МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

А. Н. КОВАРЦЕВ, А. Н. ДАНИЛЕНКО

АЛГОРИТМЫ И АНАЛИЗ СЛОЖНОСТИ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебника для студентов, обучающихся по основной образовательной программе высшего образования по направлению подготовки 02.03.02 Фундаментальная информатика и информационные технологии

САМАРА
Издательство Самарского университета
2018

УДК 004.42(075)
ББК 32.973-018.2я7
К 565

Рецензенты: д-р техн. наук, проф. С. А. П р о х о р о в,
д-р техн. наук, проф. С. В. С м и р н о в

Коварцев, Александр Николаевич
К565 Алгоритмы и анализ сложности: учебник / *А.Н. Коварцев, А.Н. Даниленко*. – Самара: Изд-во Самарского университета, 2018. – 128 с.: ил.

ISBN 978-5-7883-1263-7

Приведены основные направления исследований в теории алгоритмов, определены базовые понятия и требования, предъявляемые к написанию алгоритмов и определению порядка их сложности. Описаны методы и подходы для работы с массивами, списками, деревьями, графами и другими линейными и нелинейными структурами. Введены понятия детерминированной и недетерминированной машины Тьюринга. Представлена алгоритмическая модель языка GRAPH. В учебнике содержатся задачи и упражнения, а также вопросы для самопроверки.

Предназначен для студентов, обучающихся по направлениям подготовки «Фундаментальная информатика и информационные технологии», «Информатика и вычислительная техника».

Подготовлен на кафедре программных систем.

УДК 004.42(075)
ББК 32.973-018.2я7

ISBN 978-5-7883-1263-7

Оглавление

1. ВВЕДЕНИЕ В ТЕОРИЮ АЛГОРИТМОВ

1.1. Историческая справка

Содержательные явления, которые привели к возникновению понятия «алгоритм», прослеживаются в математике в течение всего времени его существования. Со школьной скамьи молодые люди изучают алгоритм Евклида нахождения наибольшего общего кратного натуральных чисел, найденный еще в III веке до нашей эры и доживший до наших дней. В XV веке был известен алгоритм, разработанный самаркандским астрономом Аль-Каши, вычисления числа π, которое он вычислил с 17 верными значащими цифрами после запятой. Список «старинных» алгоритмов можно продолжать и далее, однако само понятие алгоритма сформировалось как предмет самостоятельного изучения лишь в XX веке.

Первоначально понятие алгоритма рассматривалось в общей форме в виде словесных правил, схем, формул (ныне называемых методиками расчета) и использовалось для описания вычислительного процесса. Они не являлись точным математическим определением, а лишь объясняли смысл слова «алгоритм». С алгоритмами, т.е. эффективными процедурами, однозначно приводящими к результату, математики имели дело всегда. Школьные методы умножения «столбиком» и деления «углом», метод исключений неизвестных при решении систем линейных уравнений – все это примеры алгоритмов. На протяжение длительного времени понятие алгоритм в своей основе не изменялось, приобретая все большую выразительность. Традиции организации вычислений складывались веками и стали составной частью общей научной культуры. Все многообразие вычислений комбинировалось из 10–15 четко определенных операций арифметики, тригонометрии и анализа. Поэтому понятие метода вычислений считалось изначально ясным, не требующим специальных исследований.

Начиная с 20 – 30-х годов XX столетия, в связи с актуализацией вопросов обоснования математики, развитием вычислительной математики и вычислительной техники возникла необходимость уточнения понятия алгоритм как объекта математической теории. Появился раздел дискретной математики, называемый *теорией алгоритмов*. Неслучайно, что основоположниками теории алгоритмов являются великие математики XX века такие как А.И. Колмогоров, А.А. Марков, А.П. Ершов, А.И. Мальцев, В.А. Успенский, А.М. Тьюринг, К. Гёдель, А. Чёрчь, А. Туэ, Э.Л. Пост, С.К. Клини и др.

Толчком к развитию теории алгоритмов послужили исследования А. Чёрча (1936 г.) связанные попыткой понять, что же мы можем вычислить с помощью функций? Чёрч уточнил понятие *вычислимой функции* и привел примеры функции, не являющихся вычислимыми. Поскольку *вычислимая функция* определяется как функция, для которой существует алгоритм её вычисления, то естественно потребовались исследования, уточняющие понятие алгоритма.

В настоящее время теория алгоритмов развивается в следующих направлениях:

1. Исследование классов вычислимых функций. В данном случае это: изучение класса *рекурсивных и перечислимых множеств*, сравнение и *классификация алгоритмической природы произвольных подмножеств множества натуральных чисел*, *теория нумераций* и т.д.;
2. Исследование механизмов вычисления. В частности, к этому направлению можно отнести *теорию конечных автоматов* как модели вычислительного механизма. *Растущие автоматы* – самовоспроизводящиеся машины, *машины с оракулом* – развитие понятия алгоритма;
3. Изучение понятия сложности алгоритма – разработка методов оценки *сложности алгоритмов и вычисления*.

Теория алгоритмов имеет множество приложений в математической логике и теории моделей, образует теоретический фундамент для ряда вопросов вычислительной математики и тесно связана с кибернетикой и информационными технологиями.

1.2. Понятие алгоритма

Первоначально понятие алгоритма отождествлялось с понятием метода вычислений. С точки зрения современной практики алгоритм – программа, а критерием алгоритмичности вычислительного процесса является возможность его запрограммировать. Именно благодаря этой реальности алгоритма, а также благодаря тому, что подход инженера к математическим методам всегда был конструктивным, понятие алгоритма в технике за короткий срок стал необычайно популярным.

Понятие алгоритма, подобно понятиям множества и натурального числа, относится к числу столь фундаментальным понятий, что оно не может быть выражено через другие понятия.

Определение 1.1. ***Алгоритм*** *(алгорифм) – точное предписание, которое задает вычислительный процесс, начинающийся с произвольного* ***исходного данного*** *и направленный на получение полностью определенного этим исходным данным* ***результата****.

Определение 1.2. ***Алгоритм*** *есть точное предписание, которое задает вычислительный процесс нахождения значений вычислимой функции по заданным значениям ее аргументов.*

Определение 1.3. ***Алгоритм*** *есть предписание, однозначно определяющее ход некоторых конструктивных процессов.*

Данные определения недостаточны для введения четкого и однозначного описания понятия алгоритма. Необходимы уточнения.

---

* МАТЕМАТИЧЕСКАЯ ЭНЦИКЛОПЕДИЯ. – М.: Советская энциклопедия, 1977. Т. 1.

1.2.1. Основные требования, предъявляемые к алгоритмам

Если затруднительно дать точное определение некоторого понятия, то всегда несложно ввести ряд требований, которому оно должно удовлетворять. В частности, для понятия *алгоритм* можно ввести следующие требования.

1. Любой алгоритм применяется к **исходным данным** и выдает результат. Т.е. всегда существует некий *конструктивный объект,* к которому применяется алгоритм. Ясно, что объекты должны быть четко определены и отличимы друг от друга Чаще всего в качестве конструктивных объектов выступают данные или структуры данных.

2. Данные для своего размещения требуют ***память***. Память обычно считается дискретной. Единицы измерения памяти и данных должны быть согласованы между собой.

3. Алгоритм состоит из отдельных ***элементарных шагов*** (действий). Множество шагов алгоритма **конечно**.

4. Последовательность шагов алгоритма **детерминирована**, т.е. после каждого шага указывается следующий шаг, либо алгоритм останавливается.

5. Каждый алгоритм должен быть **результативным,** т.е. после конечного числа шагов выдавать результат.

6. Следует различать:
   - описание алгоритма (инструкцию или программу);
   - механизм реализации алгоритма – устройство, например, ЭВМ, включая средства запуска, остановки и управления ходом вычислений;
   - процесс реализации алгоритма (алгоритмический процесс).

Сформулированные требования несколько уточняют понятие алгоритма, но не вносят полной ясности, поскольку в них используют-

ся нечеткие понятия. Поэтому при исследовании понятия алгоритма используют более строгие алгоритмические модели, такие как машина Тьюринга, частично-рекурсивные функции, машина Колмогорова, нормальные алгорифмы Маркова, канонические системы Поста и т.д.

1.2.2. Машина Тьюринга*

Неформально, машина Тьюринга (далее МТ) представляет собой автомат с конечным числом состояний и неограниченной памятью, представленной бесконечной лентой (в общем случае набором лент). Лента поделена на бесконечное число ячеек, и на ней выделена стартовая ячейка.

В каждой ячейке ленты может быть записан только один символ из некоторого конечного алфавита $A = \{ a_1, ..., a_m \}$, где предусмотрен символ λ для обозначения пустой ячейки.

На ленте имеется головка чтения-записи, и она подсоединена к «управляющему модулю» МТ — автомату с конечным множеством состояний $Q = \{ q_1, ..., q_n \}$.

Имеется выделенное стартовое состояние q_1 и состояние завершения q_z. Перед запуском МТ находится в состоянии q_1, а головка позиционируется на нулевой ячейке ленты.

На каждом шаге головка считывает информацию из текущей ячейки и посылает ее управляющему модулю МТ. В зависимости от этих символов и собственного состояния, управляющий модуль производит следующие операции:

1) посылает головке символ для записи в текущую ячейку каждой ленты;

---

* При написании этого раздела использовались материалы работы: Кузюрин Н.Н., Фомин С.А. Эффективные алгоритмы и сложность вычислений. 2008. – 347 с.

2) посылает головке одну из команд «LEFT», «RIGHT», «STAY»;

3) выполняет переход в новое состояние (которое, впрочем, может совпадать с предыдущим).

На рисунке 1.1 представлена схема машины Тьюринга.

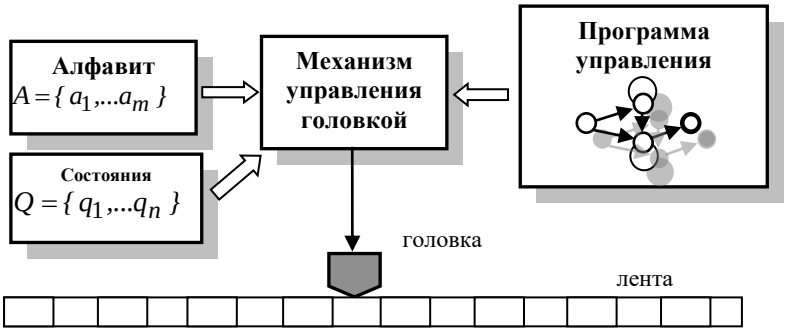


Рис. 1.1. Машина Тьюринга

Определение 1.4. ***Машина Тьюринга*** *— это набор* $T = \langle A,\ Q, \alpha, \beta, \gamma \rangle$*, где*

- A *– алфавит,* $\lambda \in A$ *– пустой символ;*
- Q *– конечное множество состояний,* $q_1,\ q_z \in Q$ *– выделенные состояния запуска машины и завершения её работы;*
- α, β, γ *– произвольные отображения:*
  - $\alpha : Q \times A \to Q$ *– задает новое состояние;*
  - $\beta : Q \times A \to A$ *– символ для записи на ленте;*
  - $\gamma : Q \times A \to \{ L, S, R \}$ *– определяет перемещение головки.*

Таким образом, машина Тьюринга определяется таблицей команд размером $|A| \times |Q|$, задающей правила работы машины в соответствии с функциями α, β, γ. Если к словарю A добавить пустой символ λ, то получим расширенный словарь $A^* = A \bigcup \{\lambda\}$.

Под ***входом*** для МТ подразумевается слово, состоящее из символов словаря A, записанного справа от стартовой позиции на ленте МТ. Договоримся, что входное слово не содержит пустых символов – иначе возникнут технические сложности, определения конца входного слова и т.п.

Результатом работы МТ над некоторым входным словом *X* считается слово, записанное на ленте после остановки МТ.

Таким образом, **память** МТ – конечное множество состояний (внутренняя память) и лента (внешняя память).

Данные МТ – слова в алфавите машины. Через $A_{исх}$ обозначим входное слово.

Элементарные **шаги** машины – это считывание и запись символов, сдвиг головки на ячейку вправо, влево, остаться на месте, а также переход управляющего устройства в следующее состояние.

«Программа управления» МТ – это задание, записанное системой правил (команд) вида: $q_i a_j \to q_i' a_j' d_k$, где d_k – направление сдвига головки $d_k \in \{L, R, S\}$.

«Программу управления» можно задать таблицей, строки которой соответствуют состояниям МТ, столбцам – входные символы, а на пересечении строки q_i и столбца a_j записана тройка символов $q_i' a_j' d_k$.

Альтернативным способом описания «программы управления» является диаграмма переходов, представленная в виде ориентированного графа, вершинами которого являются состояния q_i, а дуги помечены переходами вида $a_j \to a'_j d_k$.

Полное состояние МТ, по которому однозначно можно определить ее дальнейшее поведение, определяется её внутренним состоянием, состоянием ленты и положением головки.

Пример 1. ***Сложение***. Рассмотрим задачу сложения двух натуральных чисел. Договоримся представлять натуральные числа в единичном (*унарном*) коде, например, для числа x справедливо: $1...1 = 1^x$. Положим, что для всех числовых функций $A_{исх} = \{1\}$ либо $A_{исх} = \{1,*\}$.

Тогда числовая функция $f(x_1,...,x_n)$ правильно вычислима по Тьюрингу, если существует машина T такая, что $q_1 1^{x_1} * 1^{x_2} * ... * 1^{x_n} \underset{T}{\Rightarrow} q_z 1^y$, когда $f(x_1,...,x_n) = y$ и T работает бесконечно, начиная с $q_1 1^{x_1} * 1^{x_2} * ... * 1^{x_n}$, когда $f(x_1,...,x_n)$ не вычислима.

Рассмотрим сложение двух чисел a и b ($A_{исх} = 1^a * 1^b$) это слово необходимо переработать в 1^{a+b}, т.е. удалить разделитель и сдвинуть одно из слагаемых, скажем первое, к другому.

Это преобразование осуществляет машина T_+ с 4 состояниями и следующей системой команд:

$q_1 * \to q_z \lambda R;$

$q_1 1 \to q_2 \lambda R;$

$q_2 1 \to q_2 1 R;$

$q_2 * \to q_3 1L;$

$q_3 1 \to q_3 1L;$

$q_3 \lambda \to q_z \lambda R;$

В табличной форме программа управления выглядит так

	1	λ	*
q_1	$q_2 \lambda R$		$q_z \lambda R$
q_2	$q_2 1R$		$q_2 1L$
q_3	$q_3 1L$	$q_3 \lambda R$	

Диаграмма переходов описывается графом, представленным на рисунке 1.2.

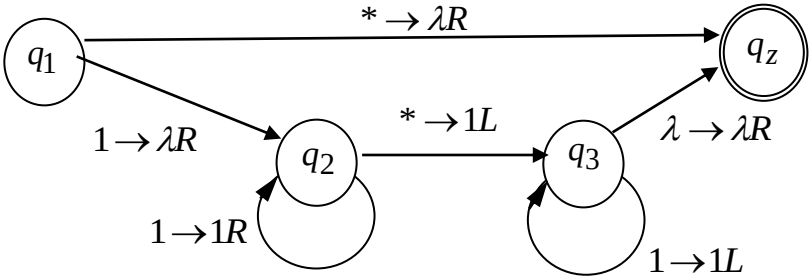


Рис. 1.2. Диаграмма переходов операции сложения МТ

Универсальная машина Тьюринга

Систему команд машины Тьюринга можно интерпретировать и как описание работы конкретного механизма, и как программу, управляющей работой механизма МТ. Обе интерпретации используются на практике. Для современного инженера это обстоятельство вполне естественно. Он хорошо знает, что любой алгоритм управления может быть реализован либо аппаратно – построением соответствующего устройства, либо программно – написанием программы для универсальной управляющей ЭВМ. В первом случае МТ «запа-

ивается» в электронную схему (часто такой подход применяется в военной технике). Во втором случае мы имеем дело с привычной практикой – разработки программного приложения на ЭВМ.

Интуитивно понятно, что правильная система команд МТ, если не делать ошибок, однозначно приводит к конечному результату. Это по существу уверенность в существовании алгоритма для реализации работы исполнительного механизма машины Тьюринга.

Словесное описание алгоритма может быть неточным, поэтому описание алгоритма работы МТ необходимо сделать с помощью машины Тьюринга, т.е. поставить задачу построения МТ, реализующей алгоритм воспроизведения работы исполнительного механизма МТ.

Для машин Тьюринга, вычисляющих функцию от одной переменной, формулировка такой задачи может выглядеть следующим образом:

необходимо построить машину Тьюринга *U*, вычисляющую функцию от двух переменных, причем такую, что для любой машины *T* с системой команд Σ_T, справедливо $U(\Sigma_T, A_{исх}) = T(A_{исх})$, если $T(A_{исх})$ определена. И $U(\Sigma_T, A_{исх})$ не останавливается, если $T(A_{исх})$ не останавливается.

Любую МТ *U,* обладающую указанными свойствами называют **универсальной машиной Тьюринга**.

1.2.3. Тезис Тьюринга

Определение 1.5. *Функция* $f : N \to N$ *является вычислимой, если существует такая машина Тьюринга T, что если ей на вход подать представленный в некоторой кодировке слово x, то*

1) если функция f определена на x, и f(x) = y, то машина T останавливается на входе x, и на выходе у нее записано y;

2) если функция f не определена на x, то машина T зацикливается (не останавливается за любое конечное число шагов) на входе x.

Со времени первого определения понятия алгоритма было предложено множество различных универсальных моделей вычислений, зачастую весьма далеких от машин Тьюринга, однако никому еще не удалось предъявить пример процесса, который можно было бы признать алгоритмическим, но который невозможно было бы создать на машине Тьюринга. Иными словами, любой вычислительный процесс может быть смоделирован на подходящей машине Тьюринга. *Это так называемый* **тезис Тьюринга** *разделяется большинством специалистов.*

Тезис Тьюринга. *Всякую вычислимую функцию (алгоритм) можно реализовать с помощью машины Тьюринга.*

Таким образом, если мы принимаем этот тезис, то можем смело говорить о вычислимости, не указывая конкретную модель. Сразу возникает вопрос: любую ли функцию $y = f(x)$, можно вычислить на МТ?

Ответ на этот вопрос отрицательный. Существует проблема остановки алгоритма. В общих чертах она сводится к вопросу о существовании алгоритма B, который для произвольного алгоритма A и данных α определял бы: приведет ли работа алгоритма A к результату или нет, т.е. $B(A,\alpha) = И$, если $A(\alpha)$ результативен и $B(A,\alpha) = Л$ в противном случае.

Эту задачу можно сформулировать, как задачу о существовании МТ T_0, которая для произвольной машины Тьюринга T и входного слова α МТ T определяла бы факт удачного завершения вычисли-

тельного процесса ($T_0(\Sigma_T, \alpha) = И$), если машина Тьюринга $T(\alpha)$ останавливается (вычислима), и $T_0(\Sigma_T, \alpha) = Л$, если $T(\alpha)$ не останавливается (не вычислима).

Теорема 1. *Не существует машины Тьюринга* T_0*, решающей проблему остановки для произвольной машины Тьюринга T.*

Полное доказательство теоремы приводится в курсе лекций «Математическая логика и теория алгоритмов» здесь же приведем некоторые неформальные соображения.

Пусть существует машина Тьюринга T_0, решающая проблему остановки для произвольной машины Тьюринга *T*. Однако она не способна решить эту проблему для самой себя. Следовательно, необходимо построить МТ T_1, которая решала бы проблему остановки МТ T_0. Нетрудно представить, что потребуется построить бесконечную последовательность машин Тьюринга $T_0, T_1, T_2, \ldots$ для решения проблемы остановки, что практически невозможно сделать*.

Приведем примеры некоторых неразрешимых проблем (вычислимости функции).

1. Проблема остановки машины Тьюринга на пустом слове неразрешима.
2. Не существует алгоритма, который для заданной машины Тьюринга *T* и ее состояния qk выясняет: попадет ли машина в это состояние хотя бы для одного входного слова *x*.

---

*В этой теореме в терминах машин Тьюринга сформулирован известный парадокс брадобрея. Рассмотрим множество *M* тех брадобреев, которые бреют тех и только тех, кто не бреют сами себя. Если такой брадобрей не бреет сам себя, то он себя должен брить (по определению множества *M*). Если же он бреет сам себя, то он себя не должен брить (опять же по определению множества *M*).

3. Не существует алгоритма, позволяющего для каждой формулы логики предикатов определить, будет ли формула выполнимой или общезначимой. Это означает, что массовые проблемы разрешимости вопроса общезначимости и выполнимости формул логики предикатов, алгоритмически неразрешимы.

4. Одной из наиболее знаменитых алгоритмических проблем математики являлась 10-я проблема Гильберта, поставленная им в числе других в 1901 г. на Международном математическом конгрессе в Париже. Требовалось найти алгоритм, определяющий для любого диафантова уравнения $F(x, y,..., z) = 0$, где $F(x, y,..., z)$ – многочлен с целыми показателями степеней и с целыми коэффициентами, имеет ли оно целочисленное решение. В общем случае эта проблема долго оставалась нерешенной, и только в 1970 г. советский математик Ю. В. Матиясевич доказал ее **неразрешимость**.

При истолковании утверждений, связанных с алгоритмической неразрешимостью, следует иметь в виду, что алгоритмическая неразрешимость означает лишь отсутствие единого способа для решения всех единичных задач данной бесконечной серии, в то время как каждая индивидуальная задача серии вполне может быть решена своим индивидуальным способом.

Более того, может оказаться разрешимой (своим индивидуальным методом) не только каждая отдельная задача этого класса, но и целые подклассы задач этого класса. Поэтому, если **проблема неразрешима в общем случае, нужно искать ее разрешимые частные случаи**. Задача в более общей постановке имеет больше шансов оказаться неразрешимой.

1.3. Граф машина

Трудно найти язык описания алгоритмов, одновременно пригодный для обучения, с одной стороны, и достаточно мощный для реализации реальных приложений (включая научные) — с другой стороны. В качестве базового языка описания алгоритмов предлагается использовать язык визуального программирования GRAPH [2].

В системе GRAPH произвольная программа интерпретируется некоторой вычислимой функцией:

$$f : in(D) \rightarrow out(D),$$

где $in(D)$ – множество входных данных программного модуля *f*,

$out(D)$ – множество выходных (вычисляемых) данных программного модуля *f*.

Определим ***граф состояний*** **G** как ориентированный помеченный граф, вершины которого – суть состояния, а дугами отмечаются переходы системы из состояния в состояние.

Каждая вершина графа помечается соответствующей локальной вычислимой функцией f_k. Одна из вершин графа, соответствующая начальному состоянию, объявляется начальной вершиной и, таким образом, граф оказывается инициальным. Дуги графа проще всего интерпретировать как *события*. С позиций данной работы *событие* – это изменение *состояния* объекта ***O*** (моделируемого алгоритма), которое влияет на развитие вычислительного процесса.

На каждом конкретном шаге работы алгоритма в случае возникновения коллизии, когда из одной вершины исходят несколько дуг, *событие* имеющее наибольший приоритет определяет дальнейшей ход развития вычислительного процесса алгоритма. Активизация того или иного события так или иначе зависит от состоя-

ния объекта, которое в свою очередь определяется достигнутой конкретизацией структур данных *D* объекта ***O***.

Для реализации *событийного управления* на графе состояний **G** введем множество предикативных функций $P = \{P_1, P_2, \ldots, P_l\}$. Под предикатом будем понимать логическую функцию $P_i(D)$, которая в зависимости от значений данных *D* принимает значение равное 0 или 1. Дугам графа ***G*** поставим в соответствие предикативные функции. Событие, реализующее переход $S_i \to S_j$ на графе состояний ***G,*** инициируется, если модель объекта **O** на текущем шаге работы алгоритма находится в состоянии S_i и соответствующий предикат $P_{ij}(D)$ (помечающий данный переход) истинен.

В общем случае предложенная концепция (без принятия дополнительных соглашений) допускает одновременное наступление нескольких событий, в том случае, когда несколько предикатов, помечающих дуги (исходящих из одной вершины), приняли значение истинности. Возникает вопрос: на какое из наступивших событий объект программирования должен отреагировать в первую очередь?

Традиционное решение этой проблемы связано с использованием механизма приоритетов. В связи с чем, все дуги, исходящие из одной вершины, помечаются различными натуральными числами, определяющими их приоритеты. Отметим, что принятое уточнение обусловлено ресурсными ограничениями, свойственными однопроцессорной ЭВМ.

Определение 1.6. *Определим универсальную алгоритмическую модель языка GRAPH четверкой* **<*D*, $\mathfrak{I}$, *P*, *G*>**, *где*

- ***D*** *– множество данных (ансамбль структур данных) некоторой предметной области, включающей объект* ***O;***

❖ $\mathfrak{I}$ *– множество вычислимых функций некоторой предметной области;*

❖ ***P*** *– множество предикатов, действующих над структурами данных D предметной области;*

❖ ***G*** *– граф состояний объекта* ***O****.*

Граф в данном случае заменяет текстовую (вербальную) форму описания алгоритма программы, при этом:

1. Реализуется главная цель – представление алгоритма в визуальной (графосимволической) форме.
2. Происходит декомпозиционное расслоение основных компонент описания алгоритма программного продукта. Так структура алгоритма представляется графом ***G,*** элементы управления собраны во множестве предикатов *P* и, как правило, значимы не только для объекта, но и для всей предметной области. Спецификация структур данных, а также установка межмодульного информационного интерфейса по данным «пространственно» отделена от описания структуры алгоритма и элементов управления.

Предложенная алгоритмическая модель **<D, $\mathfrak{I}$, P, G>** (см. рис. 1.3), в конечном счете, описывает некоторую вычислимую функцию $f_G(D)$ и в этом смысле может служить “исходным материалом” для построения алгоритмических моделей других программ. Последнее означает, что технология ГСП допускает построение иерархических алгоритмических моделей. Уровень вложенности граф-моделей в ГСП не ограничен.

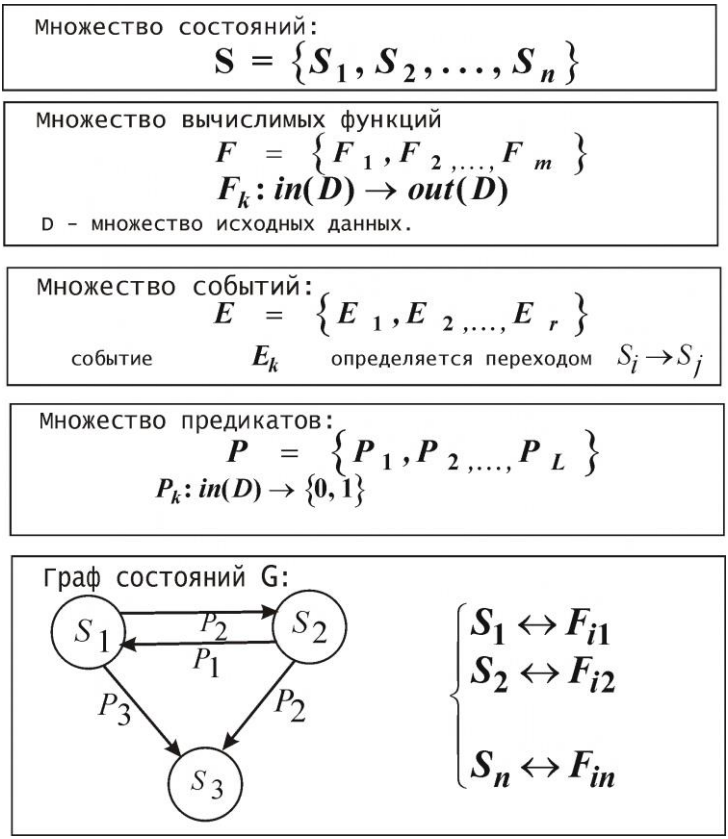


Рис.1.3. Алгоритмическая модель языка GRAPH

1.3.1. Модель данных

В качестве конструктивного объекта данных в системе GRAPH используются любые возможные структуры данных, доступные базовому языку программирования C++, на котором компилируются программы с визуальных образов GRAPH.

В системе GRAPH вводится стандарт на организацию межмодульного информационного интерфейса. Стандарт обеспечивается выполнением пяти основных правил:

1. Вводится единое для всей предметной области хранилище данных, актуальных для предметной области программирования (ПОП). Полное описание данных размещено в *словаре данных* ПОП. Любые переменные, не описанные в *словаре* данных, считаются локальными данными тех объектов ГСП, где они используются.
2. В пределах ГСП описание типов данных размещается централизовано в *архиве типов* данных.
3. В базовых модулях в качестве механизма доступа к данным допускается только передача параметров *по адресам* данных.
4. Привязка данных объектов ПОП реализована в паспортах объектов ПОП.
5. В технологии ГСП не рекомендуется использовать иные способы организации межпрограммных связей по данным.

Предложенный стандарт позволяет полностью отделить задачу построения межмодульного информационного интерфейса от кодирования процедурной части программы, а также частично автоматизировать процессы построения информационного интерфейса.

Здесь под предметной областью программирования понимается следующее.

Определение 1.7. *Под **предметной областью программирования** в дальнейшем понимается среда программирования, состоящая из общего набора данных (словарь данных) и набора программных модулей (словарь и библиотека программных модулей).*

Словарь данных представляет собой таблицу, в которой каждому данному присвоено уникальное имя, задан тип, начальное значение данного и краткий комментарий его назначения в ПОП.

Технология ГСП поддерживает жесткие стандарты на описание и документирование программных модулей, представление и поддержку информационного обеспечения программных модулей

предметной области. Таким образом, для каждой предметной области строится единая информационная среда, позволяющая унифицировать проектирование и написание программных модулей разными разработчиками.

Кроме словаря данных и каталога типов данных, информационную среду ПОП определяют объекты ГСП. Под объектом понимается, специальным образом построенный в рамках технологии ГСП, программный модуль, выполняющий определенные действия над данными ПОП.

С информационной точки зрения каждый объект ГСП f_i представляет собой функциональное отображение области определения объекта D_i^{in} на область значений D_i^{out}:

$$f_i : D_i^{in} \to D_i^{out}.$$

В общем случае $D_i^{in} \bigcap D_i^{out} \neq \varnothing$ (в объекте могут быть модифицируемые данные) и $D_i^{in}, D_i^{out} \in D$, где D – полная область данных ПОП. Для двух произвольных объектов ПОП f_i и f_j в общем случае справедливо: $(D_i^{in} \bigcup D_i^{out}) \bigcap (D_j^{in} \bigcup D_j^{out}) \neq \varnothing$.

Формально, сущность проблемы организации передачи данных между объектами в рамках некоторого модуля-агрегата f_Σ можно определить как задачу построения области данных агрегата f_Σ – $D_\Sigma = D_\Sigma^{in} \cup D_\Sigma^{out}$ и установления соответствий между данными $D_\Sigma = \{d_1, d_2, ... d_{n_\Sigma}\}$ и данными $D_i = \{d_1^i, d_2^i, ..., d_{n_i}^i\}$ объектов $f_1, f_2, ..., f_m$, из которых составлен агрегат f_Σ (см. рис.1.4).

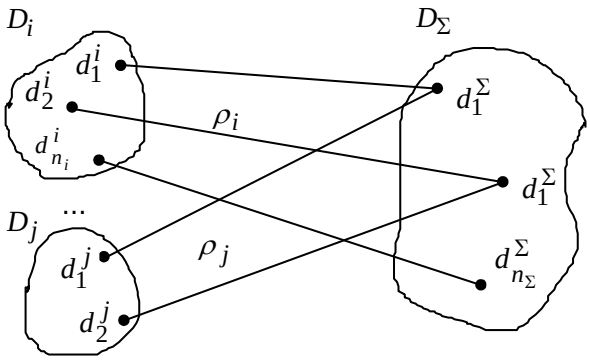


Рис.1.4. Информационный межмодульный интерфейс

1.3.2. Построение моделей алгоритмов в системе GRAPH

В качестве примера использования системы GRAPH для описания алгоритмов рассмотрим известный алгоритм сортировки «вставками».

Пусть нам задан массив натуральных чисел $A=\{a_1,a_2,...,a_n\}$. Для простоты введем фиктивный наименьший элемент $a_0=-\infty$ (для ЭВМ $a_0=-32000$).

Создадим словарь данных ПОП. В первую очередь в качестве конструктивного объекта для множества данных A массив A (в языке C++ тип массива определяется описанием: typedef int MASSIV[200];). Переменные n, i, j, w описаны в таблице 1.1.

Таблица 1.1. Словарь данных

Имя данного	Тип	Нач. значение	Комментарий
A	MASSIV	{-32000,18,4,56,65,37,63,66}	Массив, который необходимо отсортировать
n	int	6	Размерность массива A
j	int	2	Цикл
i	int	0	Счетчик
w	int	0	Промежуточный элемент

Алгоритм сортировки вставками представлен на рисунке 1.5.

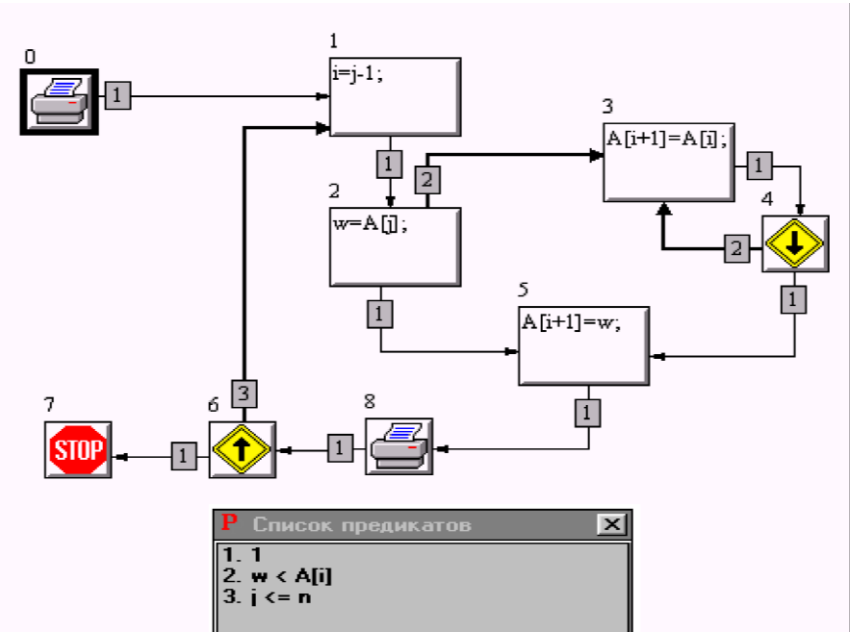


Рис. 1.5. Алгоритм сортировки вставками

Здесь:

- иконкой  обозначена вычислимая функция (объект) вывода на печать содержимого массива A (int k;printf("массив A: \n"); for (k=1;k<=n;k++) printf(" %d",A[k]); printf("\n"); getch(););
- иконкой  обозначен объект «j++»;
- иконкой  обозначен объект «i--»;
- иконкой  обозначена пустая функция «// Конец».

Нулевому элементу массива (A[0]) присвоено значение -32000.

Работа алгоритма начинается с вызова корневой вершины (на рисунке 5 обведена «жирно»). В данном случае – печать исходного массива данных. Далее последовательно, начиная с элемента A[j] (первоначально j=2) на участке массива A от j до 1 производится упорядочивание элементов в порядке возрастания их значений.

Для этого индексу «i» присваивается значение на 1 меньше j (вершина 1). В объекте 2 запоминается «старшее» (улучшаемое значение элемента) A[j]. При этом в цикле вершина 3 – вершина 4 производится перемещение элементов в направлении A[j], до тех пор, пока не выполнится логическая функция 2 (w<A[i]). В этом случае на «освободившееся» место вставляется элемент A[j] (объект 5). Очевидно, что на данный момент все элементы на участке от 1 до j оказываются упорядоченными.

В блоке 8 производится печать текущего состояния массива, в вершине 6 – увеличение индекса j на 1.

Алгоритм работает, пока не исчерпаются все числа массива A (предикат 3).

Представленный пример алгоритма сортировки данных является точным формальным описанием алгоритма, удовлетворяющим всем требованиям раздела 1.2.1. Например, данные алгоритма автоматически размещаются в памяти ЭВМ, в соответствии со спецификациями, представленными в словаре (см. табл. 1). Элементарными шагами алгоритма можно считать действия, описанные в вершинах графа. Все действия конструктивны, т.е. приводят к результату, а их количество конечно. В качестве шагов алгоритма мы рассматриваем элементарный переход алгоритма из одной вершины графа в другую. Последовательность шагов детерминирована. Описание алгоритма (фактически программы) представляется графом (см. рис. 1.5). И, наконец, алгоритм потенциально результативен, поскольку в нем есть концевая вершина.

Фактически алгоритмическая модель в системе GRAPH, описывает класс алгоритмов, определяемых четверкой **<D, $\mathfrak{J}$, P, G>**, где **D** – множество данных (ансамбль структур данных) некоторой предметной области; $\mathfrak{J}$ – множество вычислимых функций; **P** – множество предикатов, действующих над структурами данных **D**; **G** – граф состояний объекта программирования.

Универсальность представленной модели можно обосновывать универсальностью алгоритмических языков, используемых в сиситеме GRAPH (в настоящее время это язык C++), а также сводимостью предложенной алгоритмической модели к универсальной машине Тьюринга.

1.4. Вопросы для самопроверки

1. Перечислите основные направления исследований в теории алгоритмов.
2. Понятие алгоритма. Основные требования, предъявляемые к алгоритмам.
3. Машина Тьюринга. Данные машины Тьюринга. Программа управления машиной Тьюринга.
4. Результат работы МТ. Диаграмма переходов МТ.
5. Универсальная машина Тьюринга.
6. Тезис Тьюринга. Невычислимая машина Тьюринга.
7. Неразрешимая проблема. Примеры неразрешимых проблем.
8. Граф машина.
9. Алгоритмическая модель языка GRAPH.
10. Основные компоненты языка графического программирования Graph.
11. Предметная область. Словарь данных.
12. Модель данных языка GRAPH.
13. Построение моделей алгоритмов в системе GRAPH.
14. Алгоритм сортировки вставками.

1.5. Задачи

Задача 1.1. Построить машину Тьюринга, правильно вычисляющую функцию $o(x)$, где x – любое целое число в двоичном исчислении.

Задача 1.2. Построить машину Тьюринга, правильно вычисляющую функцию $o(x)$, для десятичных целых.

Задача 1.3. Какую функцию для входного слова $\alpha = 101$ вычисляет МТ со следующей программой:

	_	0	1
1	_Л2	0П1	1П1
2	_П5	_П3	_П4
3	0Л6		
4	1Л6		
5	_П5	0Н1	1Н1
6	_Л2	0Л6	1Л6
7	_Н0		

Реализует ли МТ вычислимую функцию? Как исправить МТ?

Задача 1.4. Модифицировать МТ задачи 3 для алфавита A={1, a, b}

Задача 1.5. Написать программу на машине Тьюринга, прибавляющую число 2 к введенному числу.

Задача 1.6. Постройте машину Тьюринга для распознавания строки с одинаковым количеством 0 и 1.

Задача 1.7. Перенести первый символ непустого слова P в его конец. Алфавит: A={a,b,c}.

Задача 1.8. Если первый и последний символы (непустого) слова P одинаковы, тогда это слово не менять, а иначе заменить его пустым словом. Алфавит: A={a,b,c}.

Задача 1.9. Введя в словарь данных (таблица 1), новое данное предметной области – F, int, «Факториал». Используя объекты (вычислимые функции): n--; F=1; F=F*n; printf(“Факториал: %d\n”,F);. И предикат «n==1». Построить модель алгоритма вычисления факториала n!.

Задача 1.10. Используя модуль вычисления факториала упражнения 1 построить модель алгоритма вычисления числа сочетаний из n элементов по m:

$$C_n^m = \frac{n!}{m!(n-m)!}.$$

В словарь ввести необходимые данные. Пополнить библиотеку необходимыми вычислимыми функциями и предикатами.

Задача 1.11. В условиях задачи 1.9 построить более эффективную модель алгоритма вычисления биномиального коэффициента.

2. ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ

*2.1.Временная и пространственная сложность алгоритма**

После выяснения проблемы разрешимости решаемой задачи и разработки подходящего алгоритма желательно иметь меру оценки сложности алгоритма. Здесь и дальше мы будем рассматривать только разрешимые задачи и всюду определенные (незацикливающиеся ни на одном входе) машины Тьюринга. Под временем вычисления будем понимать число шагов машины Тьюринга до получения результата.

Определение 2.1. *Пусть* $t : N \to N$. *Машина Тьюринга T имеет* ***временную сложность*** *t(n), если для каждого входного слова* α *длины n машина Тьюринга T выполняет не больше* $t(n)$ *шагов до остановки. В качестве варианта временную сложность машины Тьюринга T будем обозначать как* $timeT(n)$.

Используемой памятью будем считать число ячеек на ленте, использованных для записи, не считая длины входного слова.

Определение 2.2. ***Ленточной сложностью*** *машины Тьюринга называется функция* $s_T(\alpha)$, *которая равна мощности просматриваемой активной зоны ленты (исключая мощность входного слова).*

Обратите внимание, что пространственная сложность может быть меньше длины входного слова.

Для дальнейшего нам необходимо дать определения *детерминированной* и *недетерминированной* машины Тьюринга.

---

* При написании этого раздела использовались материалы работы: Кузюрин Н.Н., Фомин С.А. Эффективные алгоритмы и сложность вычислений. 2008. – 347 с.

Определение 2.3. *Машина Тьюринга называется* ***детерминированной****, если для каждой комбинации состояния и ленточного символа в таблице программы управления МТ записано не более одного правила.*

Определение 2.4. *Машина Тьюринга называется* ***недетерминированной****, если для каждой комбинации состояния и ленточного символа в таблице программы управления МТ может быть записано более одного правила.*

В недетерминированных алгоритмах дополнительно разрешаются недетерминированные операторы перехода вида:

goto `1 or `2.

В случае ***недетерминированной машины Тьюринга***, комбинация текущего состояния автомата и символа на ленте может допускать несколько переходов. Таким образом, для каждого входного слова имеется не один, а несколько (в общем случае — экспоненциальное число) путей, по которым может развиваться вычисление. Фактически недетерминированный алгоритм формирует «дерево вычислений». Недетерминированный алгоритм по определению выдает окончательный ответ 1, если существует, хотя бы один путь развития вычисления, на котором выдается ответ 1, и 0 — в противном случае. Таким образом, ответы ДА и НЕТ в случае недетерминированных вычислений несимметричны.

Интересной разновидностью недетерминированной машины Тьюринга является ***вероятностная машина Тьюринга***. В данном случае вместо недетерминированного перехода машина выбирает один из вариантов с некоторой вероятностью.

Следующим шагом могло бы быть введение разумного определения «оптимального» алгоритма для каждой алгоритмической за-

дачи, на основе определенных понятий временной и пространственной сложности.

К сожалению, такой подход оказался бесперспективным, и соответствующий результат (теорема об ускорении), установленный на заре развития теории сложности вычислений М. Блюмом, послужил на самом деле мощным толчком для её дальнейшего развития. Приведем этот результат без доказательства.

Теорема 2 (М. Блюма об ускорении)*. *Существует разрешимая алгоритмическая задача, для которой выполняется следующее. Для произвольного алгоритма A, решающего эту задачу и имеющего сложность в наихудшем случае timeA(n), найдется другой алгоритм B (для этой же задачи) со сложностью timeB(n), такой, что неравенство*

$$time_B(n) \le \log_2 time_A(n)$$

справедливо для почти всех n *(т.е. для всех* n*, начиная с некоторого* $n_1 > n$*).*

Эта теорема утверждает, что любой вычислительный процесс машины Тьюринга T_1 в общем случае можно улучшить с некоторого шага на МТ T_2, что в свою очередь с некоторого шага улучшается на МТ T_3 и т.д. С другой стороны, теорема Блюма не утверждает, что ускорение возможно для любой задачи. В частности, существуют задачи, интересные с практической точки зрения, для которых ускорение невозможно. Для таких задач существует оптимальный (самый быстрый) алгоритм.

---

* М. Блюм – знаменитый учёный в области информатики. В 1960-х годах Блюм разработал аксиоматическую теорию сложности вычислений, не зависящую от модели исполняющей машины, которая основывается на нумерации Гёделя. К его авторству относятся такие понятия, как: алгоритм выбора, алгоритм Блюма – Шуба, криптосистема с открытым ключом Блюма – Гольдвассера.

Тем не менее, утверждение теоремы Блюма о существовании «неудобных» задач *не позволяет нам определить общее математическое понятие «оптимального»* алгоритма, пригодное для всех задач. Поэтому развитие теории эффективных алгоритмов пошло другим путем, где одним из центральных понятий стало понятие **класса сложности**.

Классом сложности называется совокупность алгоритмических задач, для которых существует хотя бы один алгоритм с теми или иными характеристиками сложности.

2.2. Классы сложности

В теории алгоритмов **классами сложности** называют множество вычислительных задач, с примерно одинаковыми по сложности вычислениями.

При формальном определении *классов сложности* обычно рассматривают не произвольные алгоритмы, а алгоритмы для, так называемых, задач разрешения (переборных задач), когда требуется определить: принадлежит или нет некоторый элемент некоторому множеству.

Учитывая необходимость кодирования данных, подаваемых на вход машине Тьюринга, эти задачи эквивалентны задачам распознавания языков, когда на некотором алфавите Σ рассматривается подмножество слов* $L \subset \Sigma^*$, и для произвольного слова $l \in \Sigma^*$ нужно определить принадлежит ли оно языку *L*.

Суть подхода к определению наиболее сложных задач, *называемых универсальными*, состоит в сведении к ним любой переборной задачи. Решение универсальной задачи в этом смысле дает решение любой переборной задачи, и поэтому универсальная задача оказывается не проще любой из переборных задач.

---

* Σ^* – множество всех слов.

Определение 2.5. *Говорят, что неотрицательная функция* $f(n)$ ***не превосходит по порядку функцию*** $g(n)$ *и пишут* $O(f(n)) = O(g(n))$, *если существует такая константа C, что* $f(n) < Cg(n)$ *для любого* $n \in N$.

Выражениям «трудоемкость (сложность) алгоритма составляет $O(g(n))$», «решение задачи требует порядка $O(g(n))$ операций» или «алгоритм решает задачу за время $O(g(n))$» обычно придается именно этот смысл.

Например, трудоемкость $O(1)$ означает, что время работы соответствующего алгоритма ***не зависит от длины входного слова*** $(t(n) = const)$.

Алгоритмы с трудоемкостью $O(n)$ называются ***линейными***.

Алгоритм, сложность которого равна $O(n^c)$, где с – константа, называется ***полиномиальным***.

Встречаются алгоритмы, сложность которых оценивается $O(n \log_2 n)$.

Для алгоритма со сложностью $O(c^n)$ (или $O(2^{n^k})$) говорят, что он имеет **экспоненциальную сложность**.

Для более строгого определения класса сложности рассмотрим множество предикатов (любую задачу можно сформулировать с помощью соответствующим образом построенного предиката). Для переборных задач предикат формируется очевидным способом. В узком смысле каждый **класс сложности** можно определить, как множество предикатов, обладающих некоторыми свойствами.

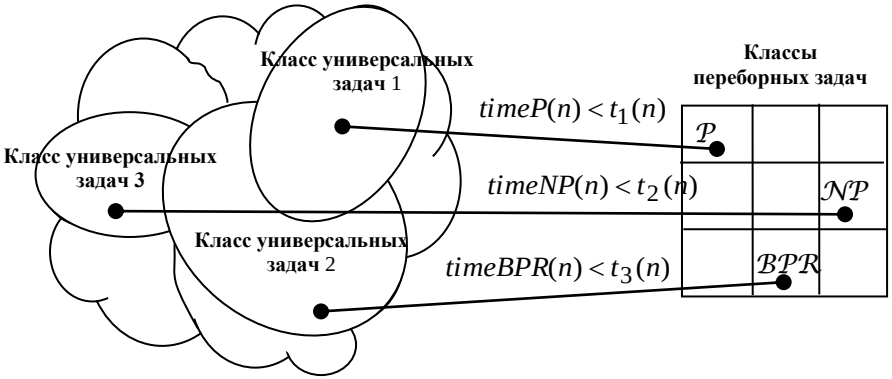


Рис. 2.1. Классификация сложности задач ($t_1(n), t_2(n), t_3(n)$ – временные сложности универсальных задач)

Классом сложности Ξ называется множество предикатов *P(x)*, вычислимых на машинах Тьюринга и использующих для вычислений $O(f(n))$ ресурсов, где ***n*** длина слова ***x*.**

В качестве ресурса обычно берут время вычисления (количество рабочих тактов МТ). Однако, можно использовать и ленточную сложность алгоритма (объем рабочей области), тем более что они связаны простыми соотношениями

$$s_T(x) \le x + 1 + time_T(x),$$

или

$$time_T(x) \le (x+1)s_T^2(x)m^{s_T(x)},$$

если в машине Тьюринга $T = \langle A, Q, \alpha, \beta, \gamma \rangle$ внешний алфавит A состоит из m элементов, множество внутренних состояний Q – из n элементов и $f(x)$ определено на x.

2.2.1. Полиномиальность и эффективность. Иерархия классов сложности

Определение 2.6. *Алгоритм называется* ***полиномиальным****, если его сложность* $t(n)$ *в наихудшем случае ограничена сверху некоторым полиномом (многочленом) от* n.

В теории алгоритмов в основном рассматриваются переборные или «распознавательные» (задачи, решение которых сводится к получению ответа «да» или «нет») массовые задачи. Произвольные задачи обычно легко сводятся к переборным или «распознавательным» задачам.

Алгоритмы, решающие **переборные задачи** с полиномиальной сложностью, часто называют **эффективными**.

Может ли не полиномиальный алгоритм быть эффективным с практической точки зрения? Ответ утвердительный.

Во-первых, может случиться так, что в реальных задачах, для которых время работы алгоритма велико, является на практике редким событием. Во-вторых, многие псевдополиномиальные алгоритмы являются эффективными, когда возникающие на практике числовые параметры не слишком велики.

Подчеркнем, что примеров задач, на которых нарушается основополагающее равенство

«полиномиальность» = «эффективность»

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

Класс всех переборных задач с полиномиальной сложностью обозначается **P**.

Классы принято обозначать прописными буквами. Дополнение к классу Ξ (то есть класс языков, дополнения которых принадлежат C) обозначается $co-\Xi$.

Иерархия классов по времени и по памяти

Класс DTIME*(f(n))* определяет класс языков, принимаемых *детерминированными машинами Тьюринга*, заканчивающими свою работу за время, не превосходящее *f(n)*.

Класс NTIME*(f(n))*, в свою очередь, определяет, как класс языков, принимаемых *недетерминированной машиной Тьюринга*, заканчивающих свою работу за время, не превосходящее *f(n)*.

Отметим, что ограничения на память при определении данных классов отсутствуют.

Класс DSPACE(*f*(*n*)) обозначает класс языков, принимаемых *детерминированными машинами Тьюринга*, использующих не более *f(n)* ячеек памяти на рабочих лентах.

Класс NSPACE*(f(n))* определяет, как класс языков, принимаемых *недетерминированными машинами Тьюринга*, использующих не более *f(n)* ячеек памяти на рабочих лентах.

Временные ограничения при определении данных классов отсутствуют.

Отсюда возникают классы сложности *PSPACE* и *NSPACE*.

Класс языков ***PSPACE*** – множество языков, допустимых детерминированной машиной Тьюринга с полиномиальным ограничением пространства.

Класс языков ***NPSPACE*** – множество языков, допустимых недетерминированной машиной Тьюринга с полиномиальным ограничением пространства.

В теории алгоритмов **классом сложности *BPP*** (от англ. *bounded-error, probabilistic, polynomial*) называется класс предикатов, быстро (за полиномиальное время) вычислимых и дающих ответ с высокой вероятностью (причём, жертвуя временем, можно добиться сколь угодно высокой точностью ответа). Задачи, решаемые веро-

ятностными методами и лежащие в *BPP*, применяются на практике очень часто*.

В теории сложности вычислений, **класс сложности *EXPTIME*** (иногда называемый просто ***EXP***) это множество задач, решаемых с помощью детерминированной машины Тьюринга за время $O(2^{p(n)})$, где *p(n)* это полиномиальная функция от *n*.

Класс сложности *APX* определяем множество оптимизационных задач, для которых существуют полиномиальные приближенные алгоритмы с константной мультипликативной точностью.

Для каждого класса существует категория задач, которые являются «самыми сложными». Действительно, в обширном множестве переборных задач (массовых задач): методы решения которых можно сформулировать в виде предиката, отвечающего «да», если имеется решение и «нет» в противном случае, имеются схожие задачи, для которых существуют или не существуют полиномиальные алгоритмы. Например, задача нахождения кратчайших маршрутов во взвешенных графах; нахождение остова минимального веса во взвешенном графе; проверки гамильтоновости цикла гамильтонового графа; задача о коммивояжере для метрического графа относятся к классу ***P*** и т.д. С другой стороны, не решаются за полиномиальное время: общая задача о коммивояжере; задача проверки гамильтоновости графа в общем случае; проверка изоморфизма графов. Они имеют экспоненциальную сложность.

---

*До 2002 года одной из наиболее известных задач, лежащих в классе *BPP*, была задача *распознавания простого* числа, для которой существовало несколько различных полиномиальных вероятностных алгоритмов, таких как *тест Миллера-Рабина*, но не существовало ни одного детерминированного. Однако в 2002 году детерминированный полиномиальный алгоритм был найден индийскими математиками Agrawal, Kayan и Saxena, которые таким образом доказали, что задача распознавания простоты числа лежит в классе *P*. Предложенный ими *алгоритм AKS* (названный по первым буквам их фамилий) распознает простоту числа длины *n* за время $O(n^{12})$.

В связи с чем, **класс всех переборных задач** обозначили через ***NP***. В этом классе содержатся все «самые сложные задачи» эквивалентной сложности, которые называют ***NP***-полными задачами.

С другой стороны, *алгоритмическая задача называется* ***труднорешаемой*** *(NP-полной), если для нее не существует полиномиального алгоритма*.

По этой причине задачи, разрешимые с экспоненциальной сложностью, относя к классу *NP*-полных задач*.

Более того, как для подавляющего большинства задач из класса *NP* в конечном итоге удается либо установить их принадлежность классу *P* (т.е. найти полиномиальный алгоритм), либо доказать *NP*-полноту.

Одним из наиболее важных исключений являются задачи вычисления дискретного логарифма и факторизации, на которых основаны многие современные криптопротоколы.

Тот факт, что большинство «естественных» массовых задач входят в класс *NP*, свидетельствует о чрезвычайной важности вопроса о *совпадении классов P* и *NP*. Безуспешным попыткам построения полиномиальных алгоритмов для *NP*-полных задач были посвящены усилия огромного числа выдающихся специалистов в данной области. Ввиду этого можно считать, что *NP*-полные задачи являются труднорешаемыми со всех практических точек зрения. Хотя, повторяем, ***строгое доказательство этого факта составляет одну из центральных открытых проблем современной математики***.

---

* Историю современной теории сложности вычислений принято отсчитывать с работ С.А. Кука (1975 года), в которых были заложены основы теории *NP*-полноты и доказано существование вначале одной, а затем достаточно большого числа (а именно, 21) естественных *NP*-полных задач. К 1979 году было известно уже более 300 наименований. К настоящему времени количество известных *NP*-полных задач выражается четырехзначным числом, и постоянно появляются новые, возникающие как в самой математике и теории сложности, так и в таких дисциплинах, как биология, социология, военное дело, теория расписаний, теория игр и т.д.

На рисунке 2.2 представлена иерархия классов сложности

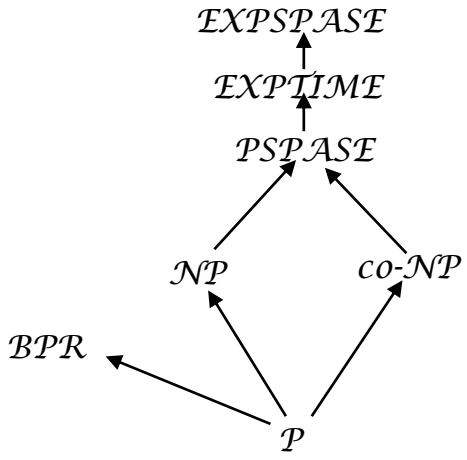


Рис. 2.2. Иерархия классов сложности

2.2.2. Алгоритмическая сводимость задач

Пусть существует алгоритм *A*, который, будучи применимым ко всякому входному слову задачи Z_1, строит некоторое входное слово $\beta = A(\alpha)$ задачи Z_2. Если при этом слово дает ответ «да»* тогда и только тогда, когда ответ «да» дает слово, то говорят что ***задача*** Z_1 ***(полиномиально) сводится к задаче*** Z_2**,** и пишут $Z_1 \Rightarrow Z_2$.

Нетрудно показать, что если $Z_1 \Rightarrow Z_2$ и $Z_2 \in \boldsymbol{P}$**,** то $Z_1 \in \boldsymbol{P}$**.**

Понятие сводимости переборных задач помогает при определении класса сложности произвольной задачи.

---

* Ответ «да/нет» понимается в смысле перевода задач z_1, z_2 к постановкам задач разрешения $P_1(\alpha)$, $P_2(\beta)$ с формулировками «Правда ли, что для входа α…». Если «да», то задача разрешима полиномиально, «нет» – неразрешима.

Доказав, что задача NP-полная, разработчик алгоритма получает достаточные основания для отказа от поиска эффективного и точного алгоритма. Дальнейшие усилия разработчика могут быть направлены, например, на получение приближенного решения либо решения важнейших частных случаев исходной задачи.

Подведем итоги. Мы определили несколько классов сложности, в частности: *P* и *NP*. Причем имеет место включение $P \subseteq NP$.

В тоже время, существует задача проверки *непустоты дополнения полурасширенного выражения,* предполагающая создание башен неограниченной высоты при фиксированном *n* объеме входной информации. Эта задача алгоритмически разрешима, но для ее решения требуется больше чем $(...((2^2)^2)...)^2$ единиц памяти (ленточная сложность). Подобные задачи не принадлежат классу *NP* и образуют класс ***труднорешаемых задач***.

Кроме того, имеется обширный класс ***неразрешимых задач,*** которые нельзя решить алгоритмически.

Таким образом, имеются четыре основных класса массовых задач: *P*, *NP*, труднорешаемые задачи и неразрешимые задачи.

2.3. Вопросы для самопроверки

1. Временная и пространственная сложность алгоритма. Дать определения.
2. Понятия детерминированной и недетерминированной МТ.
3. Оптимальность алгоритма. Теорема Блюма.
4. Классы сложности. Переборные и универсальные задачи.
5. Порядок сложности функции. Линейная, полиномиальная, экспоненциальная и другие сложности алгоритма.
6. Определение понятия класса сложности. Связь между временной и пространственной сложностью.

7. Полиномиальность и эффективность.
8. Иерархия классов сложности.
9. Класс всех переборных задач NP.
10. Теорема об иерархии.
11. Алгоритмическая сводимость.
12. Класс NP-трудных задач.
13. Класс NP-полных задач.
14. Соотношения классов. Примеры NP-полных задач.

2.4. Задачи

*Задача 2.1. О*ценить временную сложность алгоритма для следующей программы:

```
for i:=1 to N do
begin
   max:=A[i,1];
      for j:=1 to N do
      begin
         if A[i,j]>max then
         max:=A[i,j]
      end;
   writeln(max);
end;
```

*Задача 2.2. О*ценить временную сложность алгоритма для следующей программы:

```
procedure DoubleRecursive(N: integer);
begin
   if N>0 then
   begin
      DoubleRecursive(N-1);
      DoubleRecursive(N-1);
   end;
end;
```

*Задача 2.3. О*ценить временную сложность алгоритма для следующей программы:

```
procedure Slow;
var
i,j,k: integer;
begin
   for i:=1 to N do
      for j:=1 to N do
         for k:=1 to N do
            {какое-то действие}
end;
procedure Fast;
var
i,j: integer;
begin
   for i:=1 to N do
      for j:=1 to N do
         {какое-то действие}
end;
procedure Both;
begin
   Fast;
   Slow;
end;
```

Задача 2.4. Оценить временную сложность алгоритма вычисления n!.

Задача 2.5. Оценить временную сложность алгоритма вычисления n^n.

Задача 2.6. Разработать алгоритм формирования массива, составленного из сумм, пар элементов массива целых чисел A[n], в сумме дающих четное число. Оценить временную сложность алгоритма.

3. АЛГОРИТМЫ И ИХ СЛОЖНОСТЬ

3.1. Представление абстрактных объектов (последовательностей)

В определении алгоритма используется важное понятие *конструктивного объекта данных*. Без данных не существует алгоритмов. Под конструктивным объектом данных в программировании понимается *модель данных*. В большинстве языков программирования понятие модели данных часто совпадает с понятием *абстрактных типов данных*.

Выбор представления данных (типа данных, модели данных) – один из важнейших этапов разработки алгоритма. Точно также как не существует универсальных алгоритмов решения произвольных задач. Обычно невозможно предложить универсальную модель данных, пригодную для использования в разнообразных алгоритмах. Один и тот же конструктивный объект данных можно представить массивом, структурой, списком, классом, иерархией классов и т.д. Выбор модели данных зачастую зависит от решаемой задачи, используемого алгоритма, особенностей восприятия данных человеком.

Основные соображения, которыми нужно руководствоваться при выборе модели данных, состоят в следующем.

Во-первых, это естественность внешнего представления исходных данных и ответа. Их привычность для человеческого восприятия. Это требование вытекает из специфики применения ЭВМ человеком как средства автоматизации его деятельности. Польза от разработанной программы может быть сведена к нулю, если для понимания напечатанного ответа от человека требуется дополнительная сложная работа, связанная с переводом ответа в понятия исходной формулировки задачи.

Во-вторых, это возможность построения эффективного алгоритма решения задачи. Эта возможность реализуется за счет надлежащего выбора внутреннего представления исходных и промежуточных данных задачи (алгоритма). Как уже отмечалось, для построения более эффективного алгоритма наряду с внешним представлением исходных данных может потребоваться другое внутреннее представление, отличное от внешнего представления. Программа будет более эффективной, если предусмотреть перевод исходных данных в такое представление, которое обеспечит прямой доступ к нужным компонентам. Во многих задачах подобный перевод из внешнего представления во внутреннее является существенной частью процесса решения задачи; ему следует уделять должное внимание.

3.1.1. Смежное представление последовательностей

Часто приходится встречаться с представлением конечных последовательностей данных и операциями над ними (векторами, матрицами, тензорами и т.д.). С вычислительной точки зрения простейшим представлением последовательности $s_1, s_2, \ldots, s_n$ является точный список ее членов, расположенный по порядку в смежных ячейках памяти (*смежное представление данных*). В языках высокого уровня – это одномерные, двухмерные и т.д. массивы данных. Например, в языке С подобного рода абстрактные типы данных описываются следующим образом:

typedef double VECT[10]; // *описание абстрактного типа VECT как одномерного массива действительных чисел двойной точности с 10 координатами.*

typedef int MATR[10][10]; // *введение для квадратных матриц размера* 10×10 *абстрактного типа данных MATR как двухмерного массива целых чисел.*

Двухмерную матрицу действительных чисел можно определить и так:

typedef VECT MATR2[20];

Абстрактный тип данных MATR2 описывает матрицу действительных чисел (с двойной точностью) размера 20×10.

3.1.2. Связанное представление последовательностей

Модели данных со смежным размещением элементов становятся неудобными, если в алгоритме требуется изменять последовательность данных путем включения новых или исключения имеющихся элементов. В этом случае удобно пользовать **списки** (*связанное размещение данных*).

Например, пусть требуется хранить результаты «испытаний» (вычислений) некоторой функции $z = f(x, y)$ в порядке возрастаний значений функции. Введем в рассмотрение односвязанный список SFUN (см. рис. 3.1).

Рис. 3.1 Линейный односвязанный список

Каждый элемент списка занимает отдельное место в памяти (не обязательно смежное), и элементы связаны между собой с помощью указателей (адресов) (на рисунке обозначены «стрелками»). В данном случае список можно просматривать справа налево. При этом необходимо позаботится о хранении заголовка списка (указателя на первый элемент). Признаком конца списка служит «пустое» значение указателя в последнем элементе списка (NULL).

Представленный на рисунке 7 конструктивный объект данных описывается следующим образом:

```
typedef double FUNC; // ввод нового типа данных для значений функции
typedef struct LTF{
        double x; // переменная x
        double y; // переменная y
        FUNC  z; // значение функции
      LTF     *right; // указатель на следующий элемент списка
                } LFUN; // описание элемента списка
typedef LFUN* HFUN; // указатель на «голову» списка
```

Естественно, что в этом случае необходимо создать программы внесения в список нового элемента и удаления из списка существующего элемента (см. рис. 3.2).

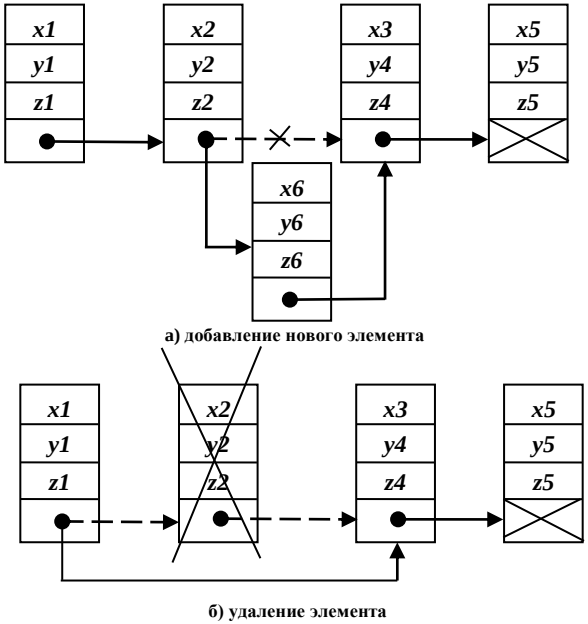


Рис. 3.2. Операции добавления и удаления элементов списка

Программа добавления элементов в однонаправленный линейный список (для языка С) в порядке возрастания значений функции имеет вид.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <malloc.h>

typedef double FUNC;
typedef struct LTF
                              {
        double x;
        double y;
        FUNC  z;
                              struct LTF   *right;
                    } LFUN;
typedef LFUN* HFUN;
int addF(HFUN *Helem, double *x,double *y, FUNC *z);

int addF(HFUN *Helem, double *x,double *y, FUNC *z)
{ LFUN *Elem, *H, *HS;
int i;
// Создание первого элемента списка
 if (*Helem==NULL)
  {
      if ((Elem = malloc(sizeof(LFUN)))== NULL)
        {printf("Невозможно выделить память для элемента списка
LFUN\n"); exit(1);}
     Elem->x = *x;
        Elem->y = *y;
        Elem->z = *z;
        Elem->right=NULL;
     *Helem=Elem;
        goto END;
  }

  H=*Helem;
  HS=H->right;
```

```
        // Добавление второго элемента
        if (HS==NULL)
          {if ((H->z)<*z)
                        {
                        if ((Elem = malloc(sizeof(LFUN)))== NULL)
                                {printf("Невозможно выделить память для элемента
списка LFUN\n");exit(1);}
                        Elem->x = *x;
                        Elem->y = *y;
                        Elem->z = *z;
                        Elem->right=NULL;
                        H->right=Elem;
                        goto END;
                        }
                        else
                        {
                        if ((Elem = malloc(sizeof(LFUN)))== NULL)
                                {printf("Невозможно выделить память для элемента
списка LFUN\n");exit(1);}
                        Elem->x = *x;
                        Elem->y = *y;
                        Elem->z = *z;
                        Elem->right= H->right;
                        H->right=NULL;
                        *Helem=Elem;
                        goto END;
                        }

            }
        BEG:
            // Добавление остальных элементов
        if (HS!=NULL)
            {if (((*Helem)->z)>*z)          // если в начало списка
                        {
                        if ((Elem = malloc(sizeof(LFUN)))== NULL)
                                {printf("Невозможно выделить память для элемента
списка LFUN\n");exit(1);}
                        Elem->x = *x;
```

```
                    Elem->y = *y;
                    Elem->z = *z;
                    Elem->right=*Helem;
                    *Helem=Elem;
                    goto END;
                    }
                    if ((H->z)<*z && HS->z>=*z)   //если в середину списка
                    {
                    if ((Elem = malloc(sizeof(LFUN)))== NULL)
                            {printf("Невозможно выделить память для элемента
списка LFUN\n");exit(1);}
                    Elem->x = *x;
                    Elem->y = *y;
                    Elem->z = *z;
                    Elem->right=HS;
                    H->right=Elem;
                    goto END;
                    }
                    if (HS->z<*z && HS->right==NULL) // если в конец списка
                    {
                    if ((Elem = malloc(sizeof(LFUN)))== NULL)
                            {printf("Невозможно выделить память для элемента
списка LFUN\n"); exit(1);}
                    Elem->x = *x;
                    Elem->y = *y;
                    Elem->z = *z;
                    Elem->right=NULL;
                    H->right=Elem;
                    goto END;
                    H=HS;
                    HS=H->right;
                    goto BEG;
                    }
          }
     END:
     //getch();

      return 1;
     }
```

```
int main(void)
{
HFUN Helem=NULL;
double x=1,y=5,z=1;
addF(&Helem,&x,&y,&z);
x=2;y=1;z=10;
addF(&Helem,&x,&y,&z);
x=2;y=4;z=3;
addF(&Helem,&x,&y,&z);
x=2;y=2;z=0;
addF(&Helem,&x,&y,&z);
//free(Helem);
return 1;
}
```

Списки – это мощный инструмент порождения разнообразных структур данных. С их помощью можно описывать деревья, графы, множества и т.д.

3.1.3. Характеристические векторы

Важной разновидностью смежного размещения является случай, когда такому представлению подвергается подпоследовательность $S_m = \{s_{k_1}, s_{k_2}, ..., s_{k_m}\}$ последовательности $S = \{s_1, s_2, ..., s_n\}$.

В этом случае подпоследовательность можно представить ***характеристическим вектором,*** т.е. последовательностью состоящей из 0 и 1. $\Theta = (\theta_1, \theta_2, ..., \theta_n)'$, $\theta_k = \begin{cases} 1, & s_k \in S_m, \\ 0, & s_k \notin S_m. \end{cases}$

Так, например, для последовательности S={1,2,3,4,5,6,7,8,9} характеристический вектор последовательности чисел, кратных 3 имеет вид $\Theta = (0,0,1,0,0,1,0,0,1)$.

3.1.4. Списки. Деревья

Использование списков для разработки алгоритма «Крестики-нолики»

Рассматривался один из возможных подходов к решению задачи «крестики-нолики» – критериальный подход. Суть которого сводится к введению некоторого критерия оценки веса каждой из клеточек игрового поля, в зависимости от текущей ситуации или перспектив развития игры.

Выберем достаточно простую схему формирования критерия оценки веса «клетки» игрового поля. Например, для всех угловых клеток поля. Итоговый (суммарный) критерий оценки веса поля K_Σ складывается из значений частных критериев ($K_\Sigma = K_1 + K_2 + K_3$) по всем исходящим из клетки зачетным линиям, на которых образуется выигрышная или проигрышная ситуация. Для угловых клеток таких линий три.

Для простоты, чтобы не вводить сложных структур данных, был предложен следующий простой алгоритм вычисления весовой оценки клетки. Для каждой клетки, смежной с клеткой, для которой производится вычисления оценочного критерия, вводится переменная, маркирующая состояние клетки, сложившаяся в процессе игры. Пустая клетка, «крестик» или «нолик», т.е. $x,y,u,v,w,z \in \{' ',' o',' x' \}$ (см. рис.3.3). Критерии K_1, K_2, K_3 вычисляются для каждой из «операционных» линий. Критерий K_1 отвечает за линию (*, x, y), критерий K_2 – за линию (*, u, v), K_3 – за линию (*, w, z).

	x	y
u	w	
v		z

Рис. 3.3. Пример весовой оценки клетки

Тогда, с помощью языка GRAPH, алгоритм вычисления критерия K_1 по линии (*, x, y) можно представит граф-программой (см. рис. 3.4). Аналогично формируются критерии K_2 и K_3.

Следует заметить, что не всегда *при разработке алгоритмов следует использовать сложные «изящные» решения*. Очень часто, для достаточно несложных задач простые решения оказываются *более результативными и качественными*.

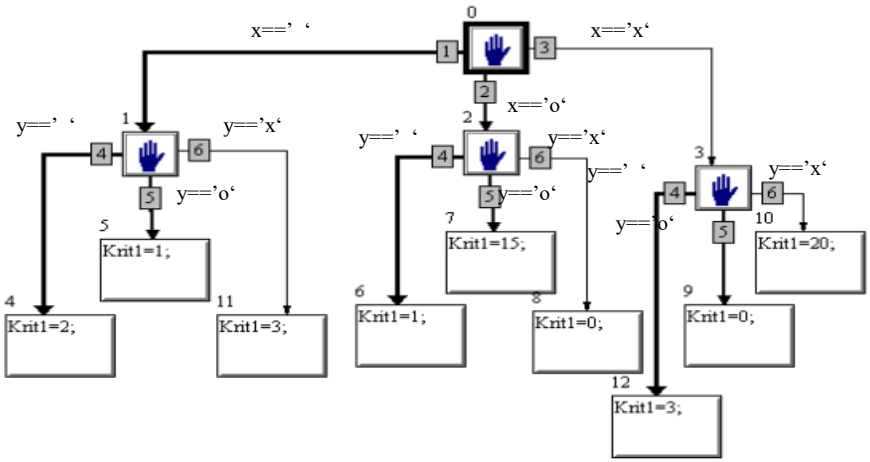


Рис. 3.4. Граф-программа вычисления критерия K_1

Однако для трехмерной игры в «крестики-нолики» предложенный подход явно неприемлем. Рассмотрим его модификацию, использующую модель данных – линейный список.

Поместим в корень дерева ссылку на элемент матрицы, содержащей координаты базовой угловой вершины, расположенной в точке (0, 0), далее в списке расположены ссылки на координаты смежных с ней операционных линий так, как это представлено на рисунке 3.5.

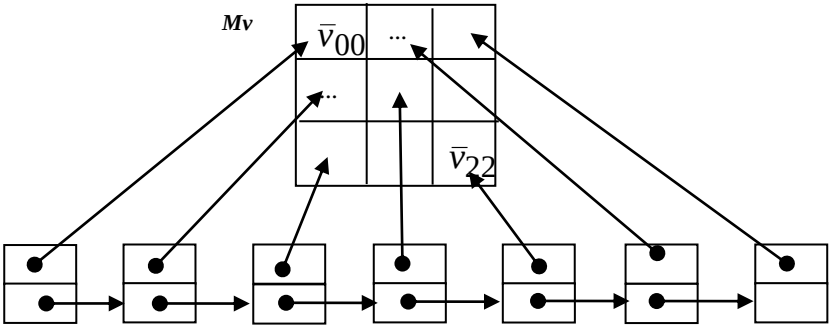


Рис. 3.5. Линейный список для решения игры «крестики-нолики»

Здесь в матрице *Mv* вектора $\bar{v}_{00},\ldots,\bar{v}_{22}$ содержатся координаты элементов матрицы игрового поля, например $\bar{v}_{00} = (0, 0)'$. Используя построенный список, можно вычислить критерии K_1, K_2, K_3 для базовой угловой вершины. Для оставшихся трех вершин можно сформировать еще три списка, но можно этого и не делать.

Если ввести линейное преобразование

$$A = \begin{Vmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{Vmatrix},$$

связанное с поворотом осей координат на заданный угол φ, и в матрице *Mv* над каждым вектором $\bar{v}_{ij}$ произвести данное преобразование $\bar{v}'_{ij} = A\bar{v}_{ij}$, то получим новую матрицу Mv' в которой все элементы повернуты на угол φ. Для того, чтобы перебрать все варианты угловых вершин необходимо сделать три поворота на углы $\varphi = \pi/2, \pi, 3\pi/2$.

3.1.5. Задачи

Задача 3.1. Составьте модель алгоритма, вычисления критерия оценки весов средних клеток рабочего поля, игры в «крестики-нолики».

Задача 3.2. Проверьте зацикленность списка за время O(n), не используя дополнительную память.

Задача 3.3. Дан список с двумя указателями у каждого элемента. Зацикленность списка не допускается. Необходимо скопировать список за время O(n) без использования дополнительной памяти.

Задача 3.4. Напишите программу удаления элемента из списка по номеру.

Задача 3.5.. Напишите программу удаления элемента из списка при совпадении значений x и y.

Задача 3.6. Напишите программу объединения двух списков.

Задача 3.7. Напишите программу выбора 12-го элемента из списка.

3.2. Сортировка и поиск

3.2.1. Сортировка вставками

В разделе 1.3.2 в качестве примера рассматривался алгоритм сортировки вставками. Оценим временную сложность этого алгоритма. Для решения этой задачи необходимо ввести понятие *инверсии* – разновидности понятия перестановки.

Инверсия

Чтобы подсчитывать эффективность различных алгоритмов сортировки нужно уметь подсчитывать число перестановок, которые повторно исполняют некоторый шаг алгоритма определенное число раз.

Пусть $A = (a_1, a_2, ..., a_n)$ – перестановка элементов множества {1,2,…,n}. Если $i<j$, а $a_i > a_j$, то пара (a_i, a_j) называется *инверсией* перестановки.

Фактически инверсия это неверное, неупорядоченное расположение двух элементов списка, который мы хотим упорядочить.

Например, перестановка $(5, 1, 4, 2)$ имеет четыре инверсии (5,1), (5,4), (5,2) и (4,2). Каждая инверсия – это пара элементов, «нарушающих порядок» следования. В совокупности они влияют на количество операций, необходимых для восстановления строгого порядка элементов.

Таблицей инверсии перестановки A называют последовательность $D = (d_1, d_2, ..., d_n)$, где d_j – число элементов в перестановке A таких, что для них выполняется условие $\forall(i = 1,2,...,k = find(A, j))\ a_i > a_k$, где функция $find(A, j)$ определяет место j-го элемента в перестановке A.

Например, для перестановки $A=(5,9,1,8,2,6,4,7,3)$ $find(A,1)=3$ по отношению к элементу «1» имеется две инверсии (5,1) и (9,1), следовательно $d_1=2$. Для второго элемента $find(A,2)=5$ и три элемента 5, 9, 8 «нарушают порядок» и, следовательно, имеем $d_2=3$. В итоге таблица инверсий приобретает вид $D=(2,3,6,4,0,2,2,1,0)$.

По определению:

$$0\le d_1\le n-1,\ 0\le d_2\le n-2,\ldots,0\le d_{n-1}\le 1,\ d_n=0.$$

М. Холл установил, что *таблица инверсий единственным образом определяет соответствующую перестановку*. Из любой таблицы инверсий можно однозначно восстановить перестановку, которая порождает данную таблицу.

Такое соответствие между перестановками и таблицами инверсий важно потому, что задачи, сформулированные в терминах перестановок, можно свести к эквивалентной ей задаче, сформулированной в терминах инверсий.

Например, для инверсий легко подсчитать число всевозможных таблиц инверсий. Так как d_1 можно выбрать *n* различными способами, d_2 независимо от d_1 – (*n-1*) способами и т.д. d_n – единственным способом. Тогда различных таблиц инверсий $n(n-1)\cdot\ldots\cdot 1=n!$

Сложность алгоритмов сортировки определяется числом проверок условия w<A[i], выполняемых в цикле. Сравнение w<A[i] для конкретного j>=2 выполняется $1+d_j$ раз, где d_j – число элементов, больших a_j и стоящих слева от него, т. е. d_j — это число инверсий, у которых второй элемент равен a_j. Числа d_j составляют таблицу инверсий:

$$0 \le d_1 \le n-1, 0 \le d_2 \le n-2, \ldots, 0 \le d_{n-1} \le 1, d_n = 0,$$

тогда в худшем случае сортировка элементов $A = (a_1, a_2, \ldots, a_n)$ потребует:

$$\sum_{j=2}^{n} (1 + d_j) \le \sum_{j=2}^{n} (1 + n - j) = \frac{n(n-1)}{2} = O(n^2)$$ сравнений.

Таким образом, сложность сортировки вставками является квадратичной.

3.2.2. Сортировка всплытия Флойда

Большинство известных алгоритмов сортировки (сортировка вставками, пузырьковая сортировка, сортировка перечислением) требуют $O(n^2)$ сравнений при сортировке элементов $A = (a_1, a_2, \ldots, a_n)$. Рассмотрим один из наиболее элегантных и эффективных методов сортировки, предложенный Флойдом, и имеющий сложность порядка $O(n \log_2 n)$.

Интересно отметить, что в методах сортировки со сложностью $O(n^2)$ при выборе наибольшего (наименьшего) элемента, обычно «забывают» информацию о других, забракованных элементах на эту роль, хотя эта проверка и выполнялась. Флойд предложил метод, в котором предшествующие проверки запоминаются и размещаются в специальной структуре данных – двоичном дереве.

Двоичные деревья на смежной памяти

Представление деревьев с помощью списочных структур данных, как правило, не представляет каких-либо трудностей. На рисунке 3.6 схематично представлена некая древовидная структура.

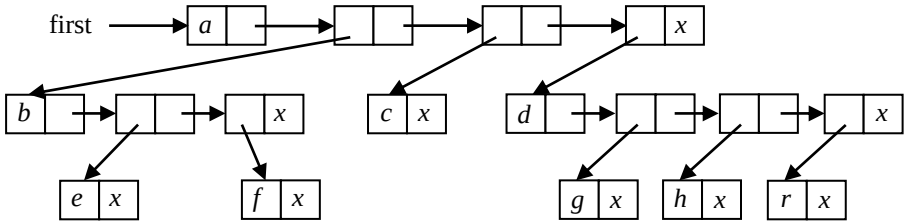


Рис. 3.6. Представление дерева на связанной памяти

Использование указателей для построения древовидных структур имеет свои неоспоримые преимущества: динамически изменяемое выделение памяти, наглядность и большая универсальность. Однако в отдельных случаях таких, как сортировка множества однородных элементов, выгоднее использовать смежное распределение элементов множества (массив).

Представление деревьев на смежной памяти (одномерный массив) предполагает неявное присутствие ребер, переходы по которым выполняются посредством выполнения арифметических операций над индексами элементов массива — смежной памяти. Формирование таких деревьев с помощью адресной арифметики можно осуществлять двумя способами. Идея первого способа применима при любом постоянном количестве ребер, выходящих из вершин (регулярное дерево). Рассмотрим данный способ формирования на примере двоичного (бинарного) дерева.

Пусть имеется одномерный массив смежных элементов

$$A = (a_1, a_2, ..., a_n).$$

Неявная структура двоичного дерева определяется как на рисунке 3.7.

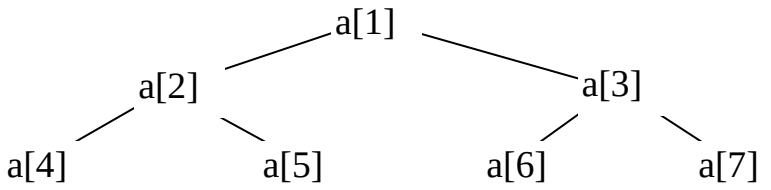


Рис. 3.7. Представление двоичного дерева на смежной памяти

По дереву на рис.3.7 легко перемещаться в любых направлениях. Переход вниз на один уровень из вершины a[k] можно выполнить, удвоив индекс k (индекс левого поддерева) или удвоив и прибавив 1 (индекс правого поддерева). Переход вверх на один уровень из вершины a[m] можно выполнить, разделив m пополам и отбросив дробную часть. Рассмотренная структура применима к любому дереву с постоянным количеством ребер, выходящих из вершин.

Определение 3.1. ***Упорядоченным бинарным деревом называют*** *дерево, у которого значение в каждой из вершин не меньше, чем значения в его дочерних вершинах.*

Метод Флойда состоит из двух этапов:

1. На первом этапе первоначальное *не упорядоченное дерево* элементов $A=(a_1,a_2,...,a_n)$ за конечное число шагов h (число уровней дерева – высота дерева) превращается в упорядоченное дерево (алгоритм всплытия).
2. На втором этапе происходит перемена местами корня дерева с его последним листом. После чего дерево уменьшается на одну вершину – последний элемент рассматриваемого массива. Теперь все готово для определения нового наибольшего (наименьшего) элемента множества с помощью применения процедуры всплытия Флойда (см. этап 1).

На рисунке 3.8 для массива $A=(3,8,1,4,6,5,2)$ показана работа первого этапа алгоритма всплытия Флойда.

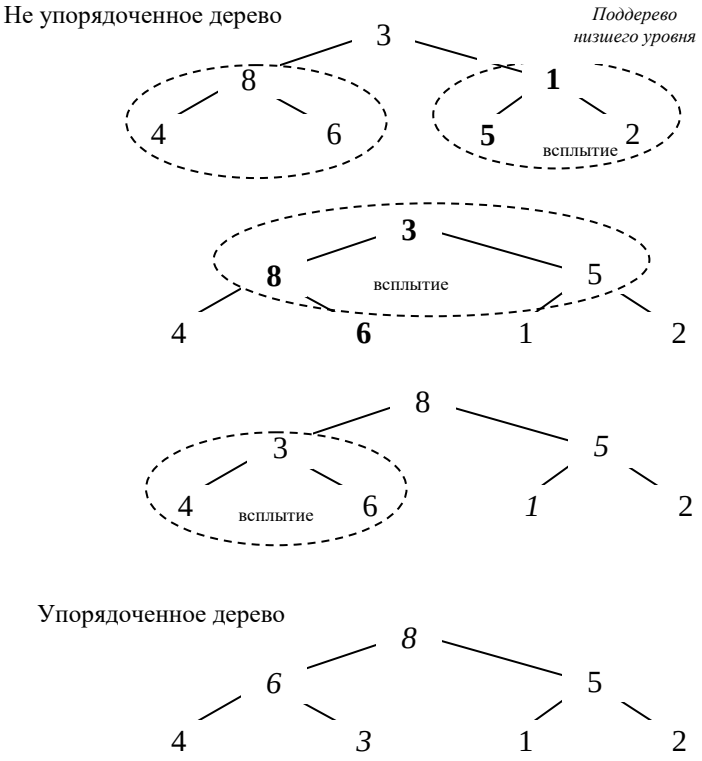


Рис. 3.8. Первый этап алгоритма

На втором этапе производится перенос значения максимальной вершины бинарного дерева (корень дерева) в последний элемент (лист дерева) массива. Соответственно последний элемент массива размещается в корне дерева. Происходит «укорачивание» дерева на один последний элемент и осуществляется поиск нового максимального элемента для новой конфигурации дерева (см. рис. 3.9).

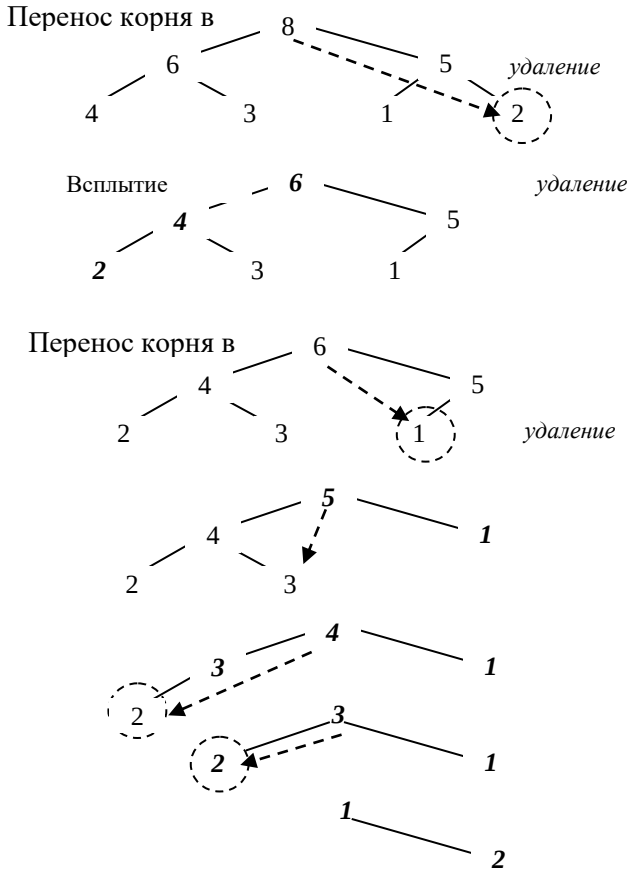


Рис. 3.9. Второй этап

Из сказанного становится ясно, что ключевым моментом алгоритма сортировки Флойда является алгоритм всплытия. На рисунке 3.10 представлен алгоритм всплытия Флойда на бинарном дереве. В сущности, он и так понятен. Сделаем несколько комментариев. В вершине 0 запоминается значение элемента, стоящего в корне дерева. Переменная m необходима для навигации по более низким уровням дерева. Ветка 2-3 необходима в случае, если рассматривается последний элемент списка при четном количестве элементов.

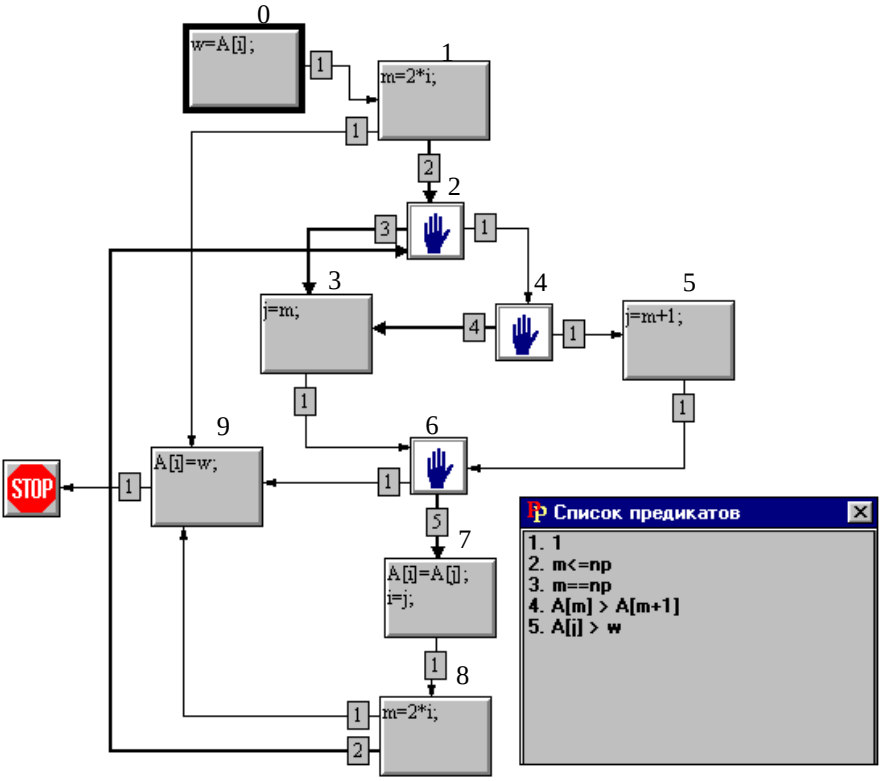


Рис. 3.10. Процедура всплытия Флойда на бинарном дереве

В вершине 7 производится текущая замена значений элементов при всплытии максимального элемента и установке корневого элемента на своем месте в последовательности вершин соответствующих элементов. Ветка алгоритма 8-2 отрабатывается, если необходима перестановка элементов на несколько уровней вниз в процессе всплытия максимального элемента.

На рисунке 3.11 полностью представлен алгоритм сортировки всплытия Флойда. Схему алгоритма, представленного на рисунке 3.11 выразительными возможностями языка GRAPH, комментировать не будем, она и так понятна.

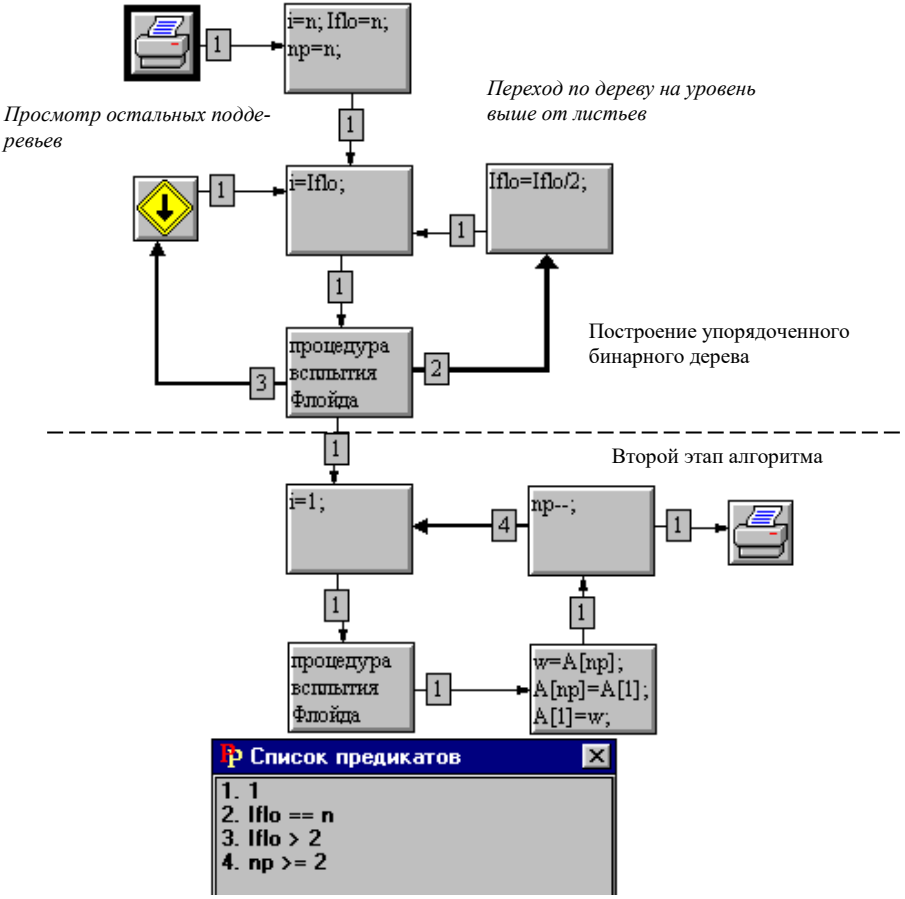


Рис.3.11. Алгоритм сортировки всплытия Флойда

Сложность алгоритма Флойда

Рассмотрим первый этап. Пусть во время всплытия на каждом уровне бинарного дерева выполняется конечное число *C* операций сравнения элементов массива *A*. Если положить, что высота дерева (число уровней в дереве) равно *h*, то сложность одного всплытия составит $C \cdot h \approx O(h)$ операций. Высота регулярного бинарного дерева из *n* вершин легко находится из соотношения $n \le 2^0 + 2^1 + ... + 2^{h-1}$, где 2^{i-1} – количество вершин на *i*-м уровне дерева, *i*=1,2,..,*n*. Отсюда* $h = [log_2(n+1)]$ и сложность процедуры всплытия имеет порядок $O(\log_2 n)$.

В полном варианте процедура всплытия Флойда на этапе 1 выполняется *n* раз и затем *n* раз для каждого шага на 2 этапе, тогда общая сложность алгоритма сортировки Флойда составит $O(nlog_2 n)$.

Оценка $O(nlog_2 n)$ вообще является наилучшей, на какую только можно надеяться при разработке алгоритмов сортировки, основанных на сравнениях элементов. Действительно, число возможных перестановок элементов множества $A = (a_1, a_2, ..., a_n)$ равно $n!$ и только одна перестановка из них удовлетворяет условию сортировки элементов. Двоичный поиск нужной перестановки среди множества $n!$ перестановок требует $log_2 n!$ числа сравнений. Воспользуемся приближенной формулой Стирлинга для вычисления $n!$ для больших n: $n! \approx \sqrt{2\pi n} n^n e^{-n}$.

Тогда $log_2 n! \approx log_2 \sqrt{2\pi n} e^{-n} + n log_2 n$, отсюда сложность гипотетического, «самого эффективного» алгоритма составляет $O(nlog_2 n)$.

---

* В данном случае используется формула суммы геометрической прогрессии

3.2.3. Задачи поиска

Последовательный поиск

Задача поиска является фундаментальной в алгоритмах на дискретных структурах. Задачи сортировки и поиска на практике «идут рука об руку». Удивительно что, накладывая незначительные ограничения на структуру исходных данных, можно получить множество разнообразных стратегий поиска с различной эффективностью.

Последовательный поиск элемента среди элементов множества $A = (a_1, a_2, ..., a_n)$ подразумевает исследование элементов множества *A* в том порядке, в каком они встречаются. Эта стратегия поиска является наиболее очевидной и самой простой.

Поиск начинается с первого элемента и продолжается до тех пор, пока не будет найден нужный элемент. После этого алгоритм останавливается.

Очевидно, что наихудшая оценка сложности алгоритма линейно зависит от мощности множества *A* и составляет $O(n)$ сравнений. Оценим *среднюю сложность* последовательного поиска.

В неотсортированном массиве, для нахождения элемента a_i требуется *i* сравнений. Для вычисления среднего времени поиска необходимо знать информацию о частоте обращений к каждому элементу множества. Предположим, что в некотором вычислительном эксперименте к каждому элементу обращаются с одинаковой частотой (частота обращений распределена равномерно). Тогда средняя сложность алгоритма для *n* независимых испытаний (*n* процедур поиска), будет равна $\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$, а *средняя сложность алгоритма поиска* $O(n)$, т.е. – линейна.

Рассмотрим распределение частот обращения к элементам в общем случае (например, в некоторой поисковой системе). Пусть ρ_i – означает приведенную частоту обращения к элементу a_i (для заданного цикла испытаний). Если для каждого элемента известно количество поисков ($m_i \quad i = 1,2,..n; \quad \sum_{i=1}^{n} m_i = n$), тогда

$$\rho_i = m_i / n, \quad \sum_{i=1}^{n} \rho_i = 1.$$

В этом случае среднюю сложность можно оценить по формуле

$$\frac{1}{n} \sum_{i=1}^{n} i\rho_i .$$

Средняя сложность алгоритма поиска в общем случае зависит от распределения частот ρ_i. Дж. Зипф заметил, что *k*-е наиболее употребительное в тексте на естественном языке слово встречается с частотой приблизительно обратно пропорционально *k*, т.е.

$$\rho_k = c / k .$$

Нормирующая константа ***c*** выбирается из условия

$$\sum_{i=1}^{n} \rho_i = 1.$$

Пусть элементы множества $A = (a_1, a_2, ..., a_n)$ **упорядочены** согласно указанным частотам, т.е. $\rho_1 > \rho_2 > ... > \rho_n$. Тогда

$$c = \frac{1}{\sum_{i=1}^{n} 1/i} \approx \frac{1}{\ln n}$$

и среднее число операций успешного поиска составит

$$\frac{1}{n} \sum_{i=1}^{n} i\rho_i = \sum_{i=1}^{n} i \frac{c}{i} = \frac{n}{\ln n} .$$

Что существенно эффективнее, чем для неотсортированного размещения данных.

Последний пример показывает, что даже простой последовательный поиск при удачном выборе структуры данных (отсортированный массив) может существенно повысить эффективность алгоритма поиска. На рисунке 3.12 приведены зависимости оценок сложностей последовательных алгоритмов поисков. Отметим, что на практике в разнообразных алгоритмах часто используется стратегия упорядочивания данных.

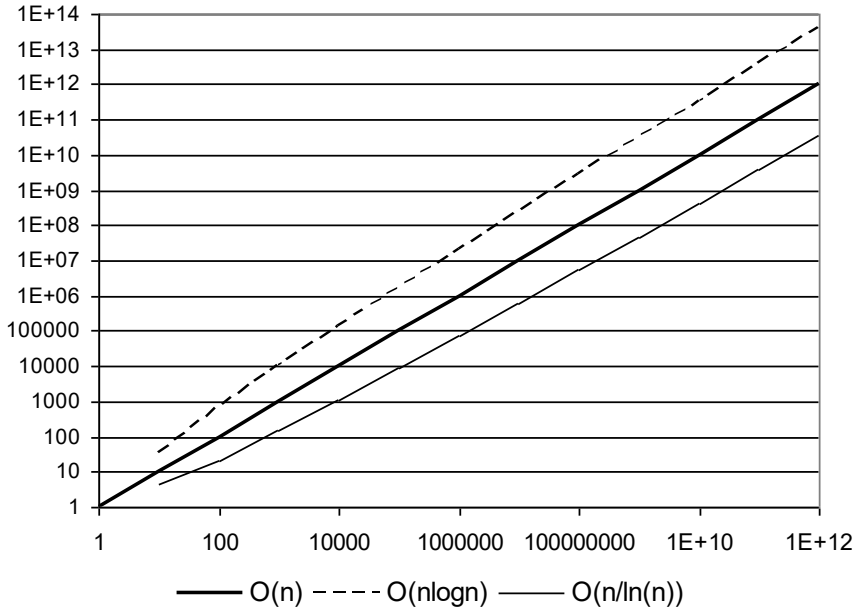


Рис. 3.12. Зависимости оценок сложностей последовательных алгоритмов поисков

Логарифмический поиск

Логарифмический поиск (бинарный, метод деления пополам) данных применим к отсортированному множеству элементов $a_1 < a_2 < \ldots < a_n$, размещение которого может быть реализовано на смежной памяти.

Идея метода заключается в том, что для большей эффективности поиска нужного элемента (например, a_{find}) необходимо построить алгоритм так, чтобы путь к элементу был как можно короче.

Последнее можно реализовать, если поиск начинать с середины множества, т.е. с элемента $a_{[(1+n)/2]}$. Если $a_{find} < a_{[(1+n)/2]}$, но нужный элемент находится справа от $a_{[(1+n)/2]}$, иначе наоборот. Поделив пополам правое или левое подмножество, мы еще раз уменьшим пространство поиска вдвое. Данную процедуру можно продолжать до тех пор, пока не найдем нужный элемент.

Естественной геометрической интерпретацией бинарного поиска является двоичное дерево сравнений.

Определение 3.2. *Двоичное дерево называется* ***деревом сравнений,*** *если для любой его вершины выполняется условие:*

{вершины левого поддерева} < вершина корня < {вершины правого поддерева}.

На рисунке 3.13 показан пример двоичного дерева сравнений для отсортированного множества *A*={3, 5, 7, 9, 12, 19, 27, 44}

Средняя сложность бинарного поиска среди элементов $a_1 < a_2 < \ldots < a_n$ сравнима с высотой двоичного дерева, которую можно оценить величиной $log_2(n+1)$. Следовательно, сложность логарифмического поиска пропорциональна величине $log_2(n+1)$.

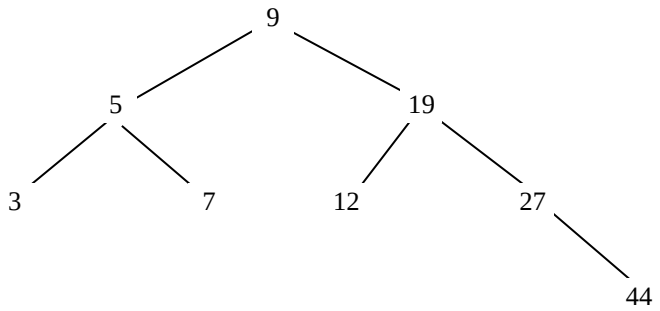


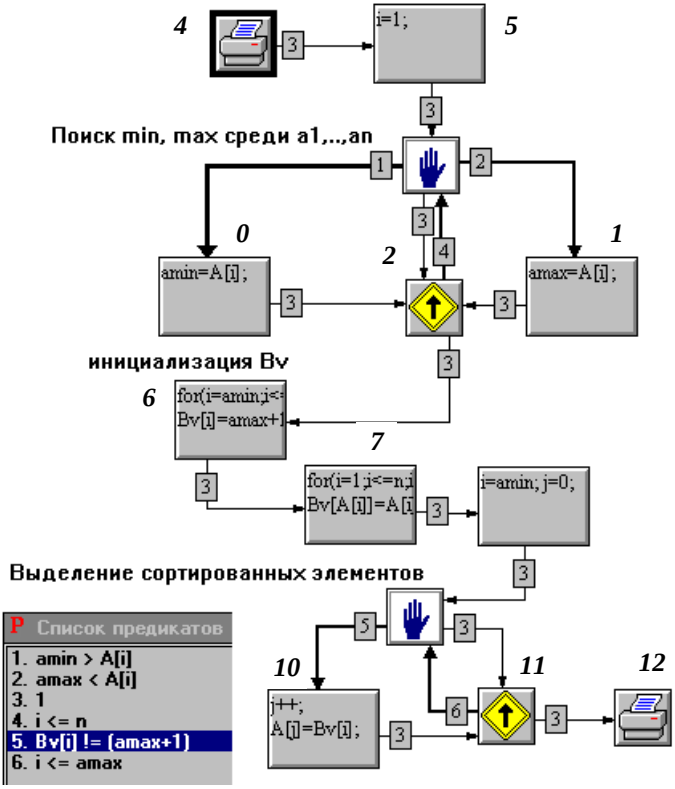
Рис. 3.13. Пример двоичного дерева сравнений для множества A

3.2.4. Сортировка с вычисляемыми адресами

В предыдущем разделе (на примере алгоритмов поиска) было показано что, учитывая особенности исходного объекта, можно существенно повысить эффективность формируемого алгоритма сортировки. В разделе 3.2.2 было показано, что для произвольного множества $A=(a_1,a_2,...,a_n)$ эффективность алгоритма сортировки не может быть лучше $O(n\log_2 n)$. Однако, если множество *A* имеет особенности, то можно построить и более эффективные алгоритмы.

Пусть $A=(a_1,a_2,...,a_n)$ – исходная последовательность сортируемых **целых чисел**. В этом случае с помощью вспомогательного множества индексов (*адресов*) $B=(b_r,b_{r+1},...,b_s)$ можно построить более эффективный алгоритм сортировки. Здесь $b_{a_i}=a_i$; $r=\min\{a_1,...,a_n\}$; $s=max\{\ a_1,...,a_n\ \}$.

Пусть все элементы множества *A* принимают различные значения, т.е. $\forall i,j \quad a_i \neq a_j$. На рисунке 3.14 представлен алгоритм сортировки с вычисляемыми адресами.



3.14. Алгоритм сортировки с вычисляемыми адресами

Библиотека вычислимых модулей алгоритма представлена в таблице 3.1.

Сделаем некоторые пояснения. На фрагменте алгоритма, в модулях 5, 0, 2, 1 осуществляется поиск максимального и минимального элемента множества *A*.

В модуле 6 производится инициализация элементов вспомогательного массива Bv значениями s+1 (amax+1). В дальнейшем это значение служит признаком отсутствия в множестве *A* соответствующего элемента, помеченного значением s+1.

Например, для множества A=(3, 6, 2, 5) в массиве Bv значение 7=6+1 будут иметь следующие элементы Bv[1] и Bv[4].

Таблица 3.1. Библиотека вычислимых модулей алгоритма сортировки с вычисляемыми адресами

№	Имя актора
1	Amin=A[i];.
2	amax=A[i];.
3	i++;.
4	//Ветвление.
5	int k;printf("массив A: \n"); for (k=1;k<=n;k++) printf(" %d",A[k]); printf("\n"); getch();.
6	i=1;. amax=A[1]; amin=A[1];
7	for(i=amin;i<=amax;i++) Bv[i]=amax+1;.
8	for(i=1;i<=n;i++) Bv[A[i]]=A[i];.
9	i=amin; j=0;.
10	//Ветвление.
11	j++; A[j]=Bv[i];.
12	i++;.
13	int k;printf("массив A: \n"); for (k=1;k<=n;k++) printf(" %d",A[k]); printf("\n"); getch();.

В вершине 7 производится запись значения a_i в массив Bv по адресу (индексу) a_i. В примере массив Bv примет вид: Bv = (7, 2, 3, 7, 5, 6).

Для построения отсортированного множества элементов *A* остается просмотреть последовательно, начиная с первого элемента, массив Bv и переписать его значения в массив A, пропуская элементы, помеченные значением s+1.

Отсортированное множество $A = (a_1 < a_2 < ... < a_n)$ является результатом последовательного просмотра массива $B = (b_r, b_{r+1}, ..., b_s)$, при условии удаления из него незанятых элементов, равных значе-

нию s+1. Алгоритм не содержит вложенных циклов, а значит сложность его – линейна $O(n)$.

Если элементы массива $A=(a_1,a_2,...,a_n)$ содержат одинаковые элементы, то описанный выше алгоритм необходимо модифицировать. Для этого достаточно ввести еще один массив $C=(c_r,c_{r+1},...,c_s)$ для подсчета кратности элементов массива *A*. В этом случае нет необходимости инициализации массива Bv значениями s+1, поскольку нулевые значения элементов массива *C* являются признаками отсутствия соответствующих чисел в массиве *A*. Отсортированный массив можно расположить в массиве Bv.

Модифицированный алгоритм не содержит вложенных циклов. Его сложность остается линейной $O(n)$.

Сортировка с вычисляемыми адресами является очень быстрым методом, но она может оказаться неэффективной с точки зрения использования оперативной памяти (ленточная сложность) при больших значениях *s*-*r*.

3.2.5. Задачи

Задача 3.8. Найдите перестановку для таблицы инверсии D = (3, 4, 4, 5, 1, 0, 2, 1, 0).

Задача 3.9. Напишите программу, которая по заданной таблице инверсии восстанавливает перестановку.

Задача 3.10. Какой перестановке соответствует таблица инверсий D = (50121200)?

Задача 3.11. Пусть перестановке $a_1,a_2,...,a_n$ соответствует таблица инверсий $D=(d_1,d_2,...,d_n)$. Какой перестановке тогда будет соответствовать следующая таблица инверсий $(n-1-d_1)(n-2-d_2)...(0-d_n)$?

Задача 3.12. Оцените реальную временную сложность работы алгоритма Флойда для массива чисел A = {5, 2, 9, 12, 1, 6, 3, 8, 7}.

Задача 3.13. Оцените реальную временную сложность работы алгоритма сортировки с вычисляемыми адресами для массива чисел упражнения 3.9.

3.3. Эффективность методов оптимизации

Численные методы решения экстремальных задач широко применяются при решении как математических задач (аппроксимация функций, решение нелинейных систем уравнений, построении нейронных сетей и т.д.), так и при построении сложных программных приложений (процедур принятия решений, при построении экспертных систем и т.п.). От эффективности методов оптимизации зачастую зависит эффективность численных методов там, где они используются. Не вдаваясь в теоретические аспекты методов оптимизации (они будут рассматриваться в отдельном курсе лекций), остановимся на оценках сложности алгоритмов оптимизации применительно к некоторым классам задач.

Рассмотрим решение задач *математического программирования*:

$$f_0(X) \to \min_x \ | \ X \in G, \quad f_j(X) \le 0, \quad j = 1,...,m, \tag{3.1}$$

где $X = (x_1, x_2, ..., x_n)'$ – оптимизируемые переменные, G – замкнутое подмножество Евклидова пространства $G \subset R_n$, $f_0(X)$ – оптимизируемая функция, $f_i(X), \quad i = 1,...,m$ – система функциональных ограничений.

На рисунке 3.15 образно представлена задача математического программирования (в данном случае выпуклого программирования). Тонкими линиями представлены изолинии (линии равного уровня) оптимизируемой функции $f_0(X)$ в пространстве двух переменных $X = (x_1, x_2)'$. Четыре ограничения представлены жирными линиями. Задача оптимизации ставится как задача поиска наименьшего (минимального) значения оптимизируемой функции, при условии выполнения всех заданных ограничений (*задача условной оптимизации*).

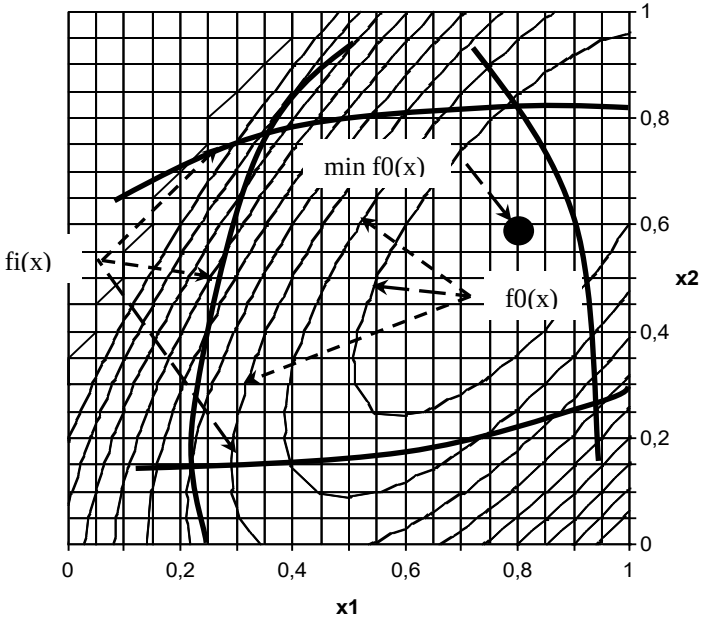


Рис. 3.15. Задача математического программирования

Эффективность алгоритмов оптимизации зависит от множества факторов. В первую очередь от свойств оптимизируемой функции и ограничений. В том числе от топологии пространства поиска, размерности вектора оптимизируемых параметров и т.д.

Для дискретной функции $y_k = f(X^k)$, заданной набором значений функции $Y = (y_1, y_2, ..., y_N)$ задача оптимизации решается достаточно просто. Для этого можно использовать любой из алгоритмов переборного поиска. При этом, если множество $Y = (y_1, y_2, ..., y_N)$ упорядоченно, то решение находится за 1 шаг (эффективность алгоритма равна $O(1)$) и не зависит от размерности вектора переменных X. В противном случае минимальное значение функции находится в худшем случае за N шагов (сложность алгоритма $O(N)$).

Значительно хуже обстоят дела для непрерывных функций. В первую очередь, в этом случае необходимо ввести понятие точности решения задачи оптимизации, поскольку *численными методами* (на ЭВМ), как правило, невозможно найти точное решение. Обычно вводят некоторую меру оценки погрешности решения поставленной задачи. Например, если X^* – точное решение задачи математического программирования, а $\tilde{X}$ – приближенное решение, полученное тем или иным алгоритмом, то оценку точности решения можно вычислить по формулам:

$$\nu = \sum_{i=1}^{n} \left| x_i^* - \tilde{x}_i \right| \text{ – для оценки абсолютной погрешности,}$$

$$\nu = \sum_{i=1}^{n} \left| x_i^* - \tilde{x}_i \right| / \left| x_i^* \right| \text{ – для оценки относительной погрешности.}$$

Заметим, что точность решения оптимизационной задачи можно оценивать и по значению функции, например, $\nu_f = \left| y^* - \tilde{y} \right|$ – абсолютная погрешность решения оптимизационной задачи по функции, где y^* – точное значение минимума функции, $\tilde{y}$ – приближенное решение.

Численные методы (алгоритмы) оптимизации формируют некоторую последовательность (траекторию) $X_1, X_2, ..., X_N$ точек области G такаю, что последняя точка $\tilde{X} = X_N$ лежит в окрестностях решения задачи математического программирования (МП) X^*.

Определение 3.3. *Траекторией алгоритма оптимизации F называется всякая последовательность*

$$X^{\infty} = \{X_1, X_2, ...\}, \quad X_i \in G.$$

Формально достаточно часто детерминированные алгоритмы оптимизации можно задать с помощью рекуррентного правила

$$X^{k+1} = F(X^k), \qquad (3.3)$$

где $F(\bullet)$ – итерационная функция.

Определение 3.4. *Трудоемкостью траектории (трудоемкостью алгоритма оптимизации) будем называть число* $l(X^{\infty})$ *членов последовательности обеспечивающую попадание последней точки траектории в* ν *– окрестность решения задачи МП.*

В методах оптимизации эффективность алгоритма традиционно оценивается по числу обращений к оптимизируемой функции. К особенности рассматриваемого класса оптимизационных задач можно отнести *приближенный характер формирования искомого решения*.

Часто невозможно найти точное решение поставленной задачи. Одним из общих подходов к решению *NP*-трудных задач, бурно развивающимся в настоящее время, является разработка приближенных алгоритмов с гарантированными оценками точности получаемого решения.

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

Определение 3.5. *Алгоритм называется С-приближенным, если при любых исходных данных он находит допустимое решение со значением целевой функции, отличающимся от оптимума не более чем в С раз.*

Заметим, что иногда говорят об С-приближенных алгоритмах, причем смысл отклонения (больше или меньше единицы) обычно ясен из контекста и содержания решаемой задачи оптимизации (максимизации или минимизации). Мультипликативная ошибка может быть константой или зависеть от параметров входной задачи. Наиболее удачные приближенные алгоритмы оптимизации позволяют управлять точностью своей работы.

Эффективность алгоритмов оптимизации зависит:

- от свойств оптимизируемой функции и в первую очередь от размерности вектора независимых переменных;
- размерности и топологии пространства независимых переменных;
- точности решения задачи оптимизации.

3.3.1. Унимодальные, непрерывные одномерные функции

Пусть при n=1 и $x \in [0;1]$ оптимизируемая функция является линейной функцией $y = ax + b$. В этом случае минимум или максимум функции находится на одном из концов отрезка [0; 1]. Для нахождения минимума функции достаточно вычислить два значения $f(0), \quad f(1)$ и выбрать минимальное значение. Сложность такого алгоритма наименьшая $O(1)$, а решение находится точно.

Пусть для одномерной задачи оптимизации ($x \in [0;1]$) задана унимодальная функция (например, функция имеющая минимум см. рис. 3.16).

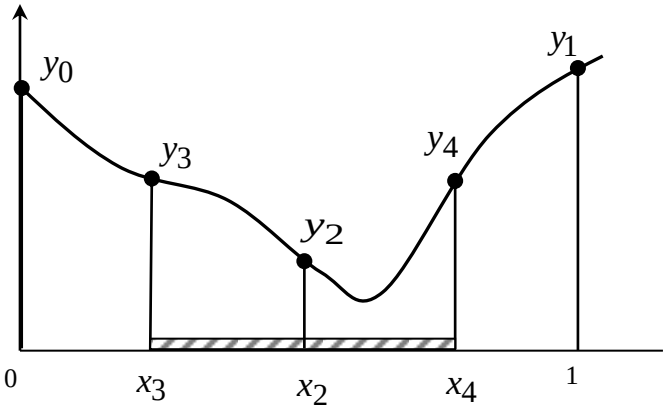


Рис. 3.16. Оптимизация унимодальной функции

В настоящее время известно большое количество алгоритмов оптимизации непрерывных унимодальных функций. Если задана непрерывная функция, то можно применить самый простой алгоритм двоичного деления *отрезка неопределенности*.

Под ***отрезком неопределенности*** будем понимать отрезок области поиска, которому принадлежит минимум функции.

Первоначально отрезок неопределенности равен исходному отрезку $X_{\min} = [0;1]$ (см. рис. 3.16). Разделим исходный отрезок пополам и вычислим еще два промежуточных значения функции в точках* $x_3 = 1/4$ и $x_4 = 3/4$. Новый отрезок неопределенности будет частью исходного отрезка, для которого выполняется условие

* Для деления отрезка *[a; b]* в заданных пропорциях $t \in [0;1]$ можно использовать формулу $x = (1-t)a + tb$. В нашем случае *t*=0,5.

$y_{i2} \le y_{i1} \le y_{i3}$. Точки с номерами $i2$, $i3$ являются смежными точками с точкой $i1$, где обнаружено наименьшее значение функции. Следовательно, отрезок неопределенности совпадает с $[x_3, x_4] = [0.25; 0.75]$ (заштрихованная линия). За пять обращений к оптимизируемой функции мы в два раза уменьшили отрезок неопределенности. Далее, вычислив два раза исходную функцию (справа и слева от средней точки), мы уменьшим отрезок неопределенности еще в 2 раза.

Число итераций N, необходимых для вычисления минимума функции с заданной точностью ν, можно определить из выражения

$$\nu \approx \left(1/2\right)^{\left[\frac{N-3}{2}\right]}$$

или

$$N \approx 3 - 2\log_2 \nu = 3 + 2\log_2 1/\nu. \qquad (3.4)$$

Из формулы (3.4) следует, что предложенный алгоритм имеет сложность равную $O(\log_2 1/\nu)$, которая для непрерывных унимодальных функций зависит от точности поиска минимума функции. Для более эффективных вариантов алгоритмов оптимизации одномерных непрерывных унимодальных функций, например, метода золотого сечения, можно получить существенно меньшую сложность $O(-\ln 1/\nu / \ln 0.382)$.

На рисунке 3.17 показаны графики трудоемкости алгоритмов двоичного деления и золотого сечения в зависимости от точности поиска минимума функции.

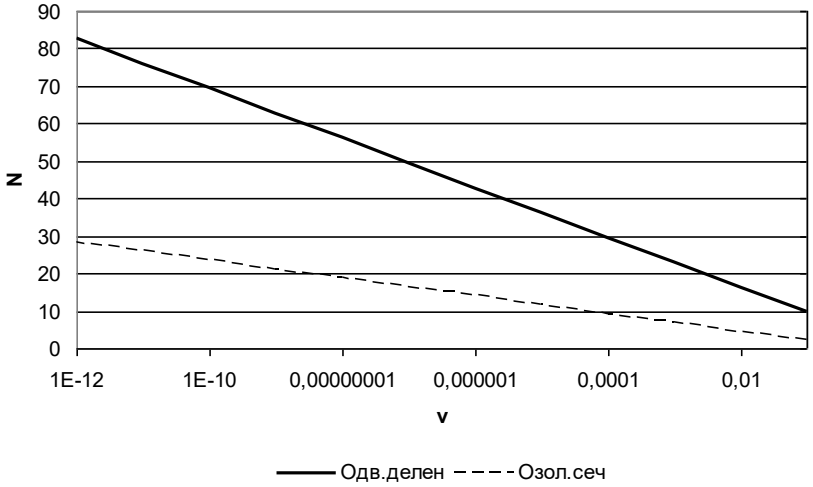


Рис. 3.17. Графики трудоемкости алгоритмов двоичного деления и золотого сечения в зависимости от точности поиска минимума функции

3.3.2. Оптимизации многоэкстремальных функций

Предложенная выше стратегия оптимизации совершенно не годится для многоэкстремальных функций. Многоэкстремальная функция содержит заранее неизвестное количество локальных минимумов и максимумов. В общем случае без дополнительных предположения, накладываемых на функцию, данная задача, по-видимому, относится к классу *неразрешимых задач.* Не вдаваясь в детали общей проблемы оптимизации многоэкстремальных функций, рассмотрим некоторые аспекты формирования оценки сложности алгоритма на простейшем примере.

Пусть оптимизируемой функцией является функция $y = a\sin(\omega x)$. Здесь a – амплитуда, ω – частота. Функция $y = a\sin(\omega x)$ является непрерывной, имеющей бесконечное число непрерывных производных любого порядка, т.е. гладкой функцией,

что, безусловно, хорошо само по себе. На отрезке $[0;2\pi]$ данная функция имеет приблизительно $[\omega]$ минимумов. Если на функцию наложить *условие Липшица*, ограничивающую скорость роста непрерывной функции, то для алгоритма, оптимизирующего функцию, появляются некоторые «ориентиры», позволяющие строить достаточно эффективные алгоритмы.

Условие Липшица задается неравенством:

$$|f(x_1) - f(x_2)| \le L|x_1 - x_2|, \qquad (3.5)$$

для любых $x_1, x_2 \in [0;1]$.

Упрощенно можно положить $|f'(x)| \le L$. Для нашего примера $|y'| = |a\omega\cos(\omega x)| < |a\omega| < L$ или $L \approx |a\omega|$, следовательно, исходный участок поиска минимума [0; 2π] будет содержать приблизительно $[\omega]$ минимумов функции, равномерно распределенных на исходном отрезке неопределенности. На котором очевидно содержатся и частные участки унимодальности функции. Для рассматриваемой функции длины участков унимодальности приблизительно равны:

$$d_1 = d_2 = ... = d_{[\omega]} = d = (2\pi/[\omega])/2 = \pi/[\omega].$$

В данном случае отбрасываются участки функции, где она принимает максимальное значение.

На каждом участке унимодальности функции можно применить алгоритм золотого сечения со сложностью для заданной длины отрезка неопределенности *d:* $N \approx \frac{\ln(\nu/d)}{\ln 0.382}$. Тогда суммарная трудоемкость с учетом всех участков унимодальности длины *d* алгоритма равна $N \approx [\omega]\frac{\ln(\nu\omega/\pi)}{\ln 0.382}$, и, следовательно, сложность алгоритма многоэкстремальной оптимизации функции $y = a\sin(\omega x)$ можно

оценить величиной $O([\omega]\frac{\ln(\omega/\nu)}{-\ln\pi\cdot 0.382})$ или $O(C\ln(1/\nu))$, где $C\approx -[\omega]/\ln(0.382)$.

В общем случае трудоемкость класса многоэкстремальной задачи математического программирования (3.1), порождаемой *k*-гладкими функциями (имеющих *k* непрерывных производных) $f_i(X),\quad i=0,...,m$ оценивается [4] снизу величиной

$$N(\nu)\geq C\left(\frac{1}{\nu}\right)^{n/k},$$

где *C* – константа, зависящая от свойств области *G*, *n* – размерность задачи оптимизации, *k* – гладкость оптимизируемых функций.

Катастрофический рост $N(\nu)$ *при* $\nu\to 0$ *и* $n\to\infty$ *показывает, что бессмысленно ставить вопрос: о построении универсальных методов решения «всех вообще» гладких задач сколько-нибудь заметной размерности*.

Последнее высказывание относится как детерминированным методам, так и стохастическим, в том числе имеет отношение и к генетическим алгоритмам, которым в последнее время незаслуженно приписываются высокие оценки эффективности. Естественно, что речь идет о методах, дающих гарантированные результаты по точности (прямые методы). В отдельных случаях может повезти любому методу оптимизации или появляется возможность разработки специализированных алгоритмов полиномиальной сложности.

3.3.3. Сложность выпуклых экстремальных задач

Если на оптимизируемые функции задачи (3.1) наложить еще более жесткие ограничения, например, положить что $f_i(X),\quad i=0,...,m$ являются выпуклыми непрерывными функциями, *G* – выпуклое множество, то сложность алгоритмов оптимизации

можно оценить величиной $O(n\ln(1/\nu))$, что существенно лучше предыдущего случая.

Интересно, что для самого простого случая, когда все $f_i(X),\quad i=0,\ldots,m$ функции являются линейными (*задача линейного программирования*), и можно было бы ожидать алгоритма точного решения. Наилучший из известных методов – симплекс метод дает только ***полиномиальную оценку сложности***. Конечно это верхняя оценка сложности. Для реальных задач методы линейного программирования работают значительно эффективнее.

3.3.4. Задачи

Задача 3.14. Пусть задана непрерывная, одномерная, унимодальная функция. Оцените сложность задачи оптимизации методом половинного деления и золотого сечения. Какой алгоритм эффективнее.

Задача 3.15. Для функции $y = x^2 + 0.5\sin(\frac{\pi}{4}x)\cos(\frac{\pi}{8}x)$ оцените реальную временную сложность алгоритма двоичного деления. Точность оптимизации $\nu = 0.01$.

Задача 3.16 .Для функции $y = 0.5(x-0.8)^2 - 2x + 1$ оцените реальную временную сложность алгоритма золотого сечения. Точность оптимизации $\nu = 0.001$.

*3.4. Задачи на графах**

3.4.1. Представление графов

Наиболее известным и популярным способом представления графов является геометрический способ, основанный на изображении графа в виде образа, состоящего из точек (вершин графа) и линий (соединяющих точки) – ребер графа. При разработке алгоритмов для задач на графе, а, в последствие для программ, необходимо вводить некоторый формальный способ представления графов. Способ основанный на использовании известные в языках программирования структуры данных. К счастью графы можно представить с помощью разнообразных алгебраических форм: матриц, ассоциируемых с графом или структур данных.

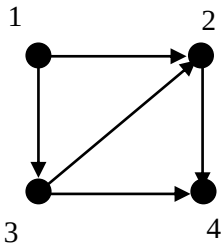
Матрица смежности графа

Определение 3.6. *Матрицей смежности ориентированного помеченного графа с n вершинами называется матрица* $A = \lfloor a_{ij} \rfloor$ $i, j = 1,2,...,n$, *в которой*

$$a_{ij} = \begin{cases} 1, \text{ если существует ребро } (v_i, v_j), \\ 0, \text{ если вершины } v_i, v_j \text{ не связаны ребром } (v_i, v_j). \end{cases}$$

Матрица смежности однозначно определяет структуру графа. Например, для графа, представленного на рисунке 3.18, матрица смежности имеет вид:

* Данный раздел подготовлен с использованием материалов курса лекций [1].



$$A=\begin{bmatrix}0 & 1 & 1 & 0\\ 0 & 0 & 0 & 1\\ 0 & 1 & 0 & 1\\ 0 & 0 & 0 & 0\end{bmatrix}.$$

Рис. 3.18. Матрица смежности графа

Матрица инцидентности графа

Определение 3.7. *Матрицей инцидентности для неориентированного графа с n вершинами и m ребрами называется матрица* $B=\lfloor b_{ij}\rfloor\ i=1,2,\ldots,n;\ j=1,2,\ldots,m$, *строки в которой соответствуют вершинам, а столбцы – ребрам. При этом*

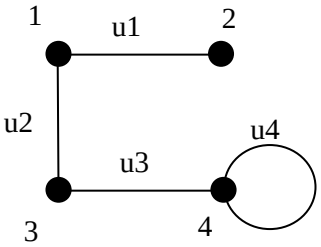
$$b_{ij}=\begin{cases}1, \text{ если вершина } v_i \text{ инцидентна ребру } u_j,\\ 0, \text{ если вершина } v_i \text{ не инцидентна ребру } u_j.\end{cases}$$

Определение 3.8. *Матрицей инцидентности для ориентированного помеченного графа с n вершинами и m ребрами называется матрица* $B=\lfloor b_{ij}\rfloor\ i=1,2,\ldots,n;\ j=1,2,\ldots,m$, *строки в которой соответствуют вершинам, а столбцы – ребрам.*

При этом:

$$b_{ij}=\begin{cases}+1, \text{ если ребро } u_j \text{ выходит из вершины } v_i,\\ -1, \text{ если ребро } u_j \text{ входит в вершину } v_i,\\ 0, \text{ иначе.}\end{cases}$$

На рисунке 3.19 показаны граф и его матрица инцидентности



$$B = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} u1 & u2 & u3 & u4 \end{array} \\ \left\| \begin{array}{cccc} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{array} \right\| \end{array}$$

3.19. Граф и его матрица инцидентности

Матрица весов графа

Определение 3.9. *Матрицей весов графа с n вершинами называется матрица* $W = \lfloor w_{ij} \rfloor \; i, j = 1,2,...,n$, *где* w_{ij} *– вес ребра, соединяющего i и j вершины.*

Веса несуществующих ребер полагают равными 0 или ∞ в зависимости от смысла решаемой задачи.

Список ребер графа

При описании графа списком ребер каждое ребро представляется парой инцидентных ему вершин. Это представление можно реализовать двумя массивами:

$$Tf = (tf_1, tf_2, ..., tf_m),$$
$$To = (to_1, to_2, ..., to_m),$$

Каждый элемент в массивах есть метка вершины, а i-е ребро выходит из вершины tf_i и входит в вершину to_i. Например, для графа (см. рис. 3.18) массивы имеют вид:

$$Tf = (1,1,3,3,2),$$
$$To = (2,3,2,4,4).$$

Интересно, что данное представление позволяет описывать петли и кратные ребра.

Структура смежности графа

Ориентированные или неориентированные графы можно однозначно представить *структурой смежности* своих вершин. Такую структуру удобно представлять массивом G линейно связанных списков Adj. Каждый список содержит вершины, смежные с вершиной, из которой исходят ребра.

Например, определим тип списка вершин

```
typedef struct LIST{
        struct LIST *right;      // указатель на следующий элемент,
        int top;                 // номер инцидентной вершины графа,
      int arc;                   // метка дуги графа,
        double wight;            // вес дуги графа.
                        } LTOP;
```

и тип массива структуры смежности

```
typedef LTOP ADJ[m];
```

тогда структуру данных, описывающей граф можно объявить следующим образом: ADJG;

Количество описателей в структуре LTOP может варьироваться в зависимости от решаемой задачи. В качестве примера опишем структуру смежности графа (рис. 3.18):

vi	LTOP
1	2,3
2	4
3	2,4
4	-

Структура смежности графа на смежной памяти

Использование списков не всегда удобно при реализации программ алгоритмов. Во-первых, при этом необходимо организовать процедуру навигации по спискам представления графа. Во-вторых, необходимо написать программы заполнения списков и нужно следить за очисткой памяти после завершения программы. Намного удобнее использовать размещение описания графа на смежной памяти.

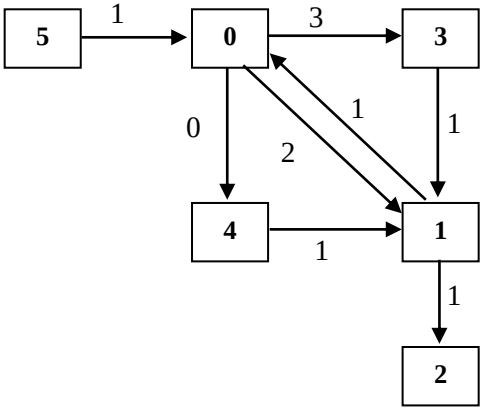


Рис. 3.20. Пример графа

В качестве примера рассмотрим граф, представленный на рисунке 3.20. Предполагается, что все вершины и дуги графа пронумерованы от *0* до *n* и *m*. В общем случае дуги графа могут иметь одинаковые метки.

Введем две структуры, предназначенные для описания вершин графа и дуг переходов на графе, сгруппированных следующим образом:

```
typedef struct _ListTop
        {       int  FirstDef;
                int  LastDef;
        } DEFTOP;
typedef struct _ListGraf
        { int NambArc;
          int NambTop;
        } DEFGRAF;
      DEFTOP    ListTop[6];
      DEFGRAF  ListGraph[8];
```

В массиве ListTop каждой вершине (строке массива) ставится в соответствие участок массива ListGraph, на котором последовательно перечислены номера смежных вершин графа. Начало и конец участка определяют параметры FirstDef и LastDef. В массиве ListGraph содержится информация о номерах дуг и вершин, смежных вершинам, описанных в массиве ListTop. Такой способ представления графа позволяет организовать быструю «навигацию» по структуре ориентированного графа, поскольку здесь используется прямая адресация связей вершин графа между собой. Например, для графа (см. рис.3.20) для определения списка вершин, исходящих из вершины 1 нашего примера, необходимо в массиве ListTop (см. рис.3.21) в строке с номером 1 определить начало и конец описания смежных вершин графа во фрагменте матрицы LitGraph, начиная с номера 3 по номер 4. В результате получим вершины 0 и 2. На рисунке в массиве ListTop число – 77 кодируется концевая вершина.

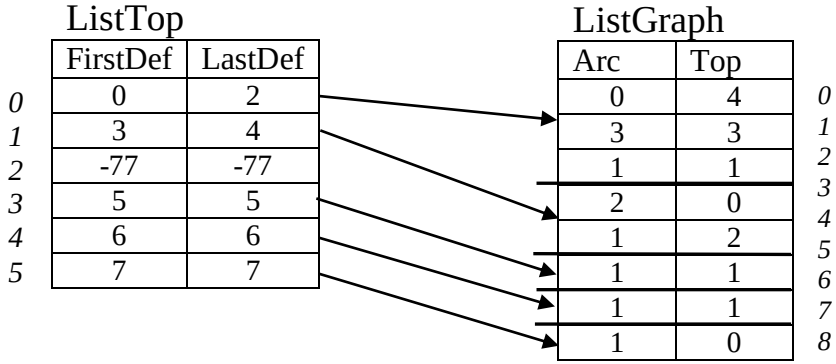


Рис. 3.21. Схема работы

3.4.2. Задача о «коммивояжере»

Классическая задача о «коммивояжере» формулируется следующим образом:

Задача о коммивояжере (TSP[a]). *Заданы неориентированный полный граф из n вершин-городов, и* $d_{ij} = d(v_i, v_j)$ *— положительные целые расстояния между городами. Чему равна наименьшая возможная длина кольцевого маршрута (Гамильтонова пути[b]), проходящего по одному разу через все города? Т.е. нужно найти минимально возможное значение суммы*

$$\min_{p \in \mathrm{P}} \sum_{i=1}^{n-1} d_{p_i, p_{i+1}} + d_{p_n, p_1}, \qquad (3.6)$$

где $p = (1, 2, \ldots, n)$ *– перестановка целых чисел,* P *– множество всех перестановок, минимум берется по всем перестановкам* p.

[a] В англоязычной литературе – Traveling Salesman Problem.

[b] Гамильтоновым циклом называется маршрут, проходящий через все вершины графа, в котором ребра встречаются не более чем один раз, где первая и последняя вершины совпадают.

Переборный алгоритм для задачи о коммивояжере просто перебирает все возможные перестановки городов с фиксированным «стартовым» городом. При анализе его сложности видно, что вычисление индивидуальной суммы (3.6) не представляет особых трудностей и требует *Cn* операций, где *C* — некоторая константа.

Проблема состоит в том, что этот процесс придется повторить *(n-1)!* раз, что дает общую сложность алгоритма $O(n!)$. Некоторые значения факториала приведены в таблице 3.2.

Таблица 3.2. Значения факториала

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

Видно, что при *n*=5 расчет всех вариантов согласно переборному алгоритму может быть произведен вручную. При *n*=8 для его проведения в разумный отрезок времени нужно привлечь программируемый калькулятор, а при *n*=10 – уже более быстродействующую вычислительную технику. Когда число городов доходит до 13, потребуется суперкомпьютер, а случай *n*=15 выходит за пределы возможностей любой современной вычислительной техники*. Число возможных вариантов при *n* = 30 превышает количество атомов на Земле.

3.4.3. Алгоритм эффективного порождения перестановок

Порождения множества всех перестановок непростая задача. В работе [5] приводится оригинальный итерационный алгоритм формирования всех перестановок. В предлагаемом алгоритме на каждом

* Конечно, разработаны различные методы сокращения перебора, кроме того, переборные задачи допускают эффективное распараллеливание для вычисления на многопроцессорных машинах или сети компьютеров. В частности, в 2004 году была решена задача «TSP» для 24978 городов.

шаге рассматривается транспозиция двух соседних элементов. Кроме текущей перестановки $\pi = (\pi_1, \pi_2, .., \pi_n)$ используется обратная[c] к ней перестановка $p = (p_1, p_2, ..., p_n)$ и вектор направлений перемещений элементов $d = (d_1, d_2, ..., d_n)$.

Компоненты вектора направленных перемещений равны 0, если элемент не перемещается, -1, если перемещается влево и +1 – вправо. Элемент перемещается до тех пор, пока не достигнет элемента, большего, чем он сам; в этом случае сдвиг прекращается. В этот момент, направление сдвига меняется на противоположенное.

Теперь для движения используется следующий меньший по значению элемент, который можно сдвинуть. Обратная перестановка используется для определения места следующего меньшего его элемента, который можно сдвинуть.

Идею метода можно проиллюстрировать на следующем простом примере. Пусть необходимо построить последовательность перестановок на множестве $(1, 2, 3, 4)$. Всего $4! = 24$ перестановки.

На рисунке 3.22 показана схема алгоритма построения перестановок.

Работа алгоритма начинается с самого правого элемента начальной перестановки. На схеме темным цветом отмечены «переставляемые» элементы. В данном случае элемент «4» смещается влево на один шаг. И это продолжается до тех пор, пока «4» не дойдет до левого края. Далее двигаться некуда, последней транспозицией в этой серии является перемена местами элементов «3» и «1».

---

[c] Обратной к исходной перестановке называют перестановку, получающуюся из исходной, если в исходной перестановке $\pi = \begin{pmatrix} \pi_1, \pi_2, ..., \pi_n \\ 1, \ 2, ..., n \end{pmatrix}$ поменять местами строки и упорядочить столбцы в возрастающем порядке $\pi^{-1} = \begin{pmatrix} 1 \ , 2, \ \ ..., n \\ \pi_1^{-1}, \pi_2^{-1}, ..., \pi_n^{-1} \end{pmatrix}$

В следующей серии транспозиций с 5 по 8 строку изменяется направление движения, при этом изменяет свои значения вектор направлений перемещений *d*. В новой серии самый левый элемент теперь двигается вправо, до тех пор, пока не упрется в правую границу. Последней транспозицией в новой серии перемена местами элементов «1» и «3». Происходит обновление вектора *d*. Далее меняется направление движение на противоположенное и алгоритм работает так, как это показано на схеме рисунка 3.22.

№	π_t				d_t			
1	1	2	3 ←	4	0	-1	-1	-1
2	1	2 ←	4	3	0	-1	-1	-1
3	1 ←	4	2	3	0	-1	-1	-1
4	4	1	2 ←	3	0	-1	-1	-1
5	4 →	1	3	2	0	-1	-1	1
6	1	4 →	3	2	0	-1	-1	1
7	1	3	4 →	2	0	-1	-1	1
8	1 →	3	2	4	0	-1	-1	1
9	3	1	2 ←	4	0	-1	-1	-1
10	3	1 ←	4	2	0	-1	-1	-1
11	3 ←	4	1	2	0	-1	-1	-1
12	4	3	1 ←	2	0	-1	-1	-1
13	4 →	3	2	1	0	-1	1	1
14	3	4 →	2	1	0	-1	1	1
15	3	2	4 →	1	0	-1	1	1
16	3 →	2	1	4	0	-1	1	1
17	2	3	1 ←	4	0	-1	1	-1
18	2	3 ←	4	1	0	-1	1	-1
19	2 ←	4	3	1	0	-1	1	-1
20	4	2	3 ←	1	0	-1	1	-1
21	4 →	2	1	3	0	-1	1	1
22	2	4 →	1	3	0	-1	1	1
23	2	1	4 →	3	0	-1	1	1
4!	2 →	1	3	4	0	-1	1	1

Рис. 3.22. Схема работы алгоритма перестановок

Удивительно, что данный алгоритм построения всех перестановок реализуется с помощью элементарных операций над векторами *p* и *d*. Остается доказать, что алгоритм не содержит повторений и перечисляет все перестановки. Попробуйте сделать это самостоятельно.

Полностью алгоритм программы приводится на рисунке 3.23 и в таблицах 3.3. – 3.5.

Таблица 3.3. Модули алгоритма эффективной перестановки

№	Имя актора
0	int k; for(k=1;k<=Ntop;k++) {Zt[k]=Pz[k]=k;Dt[k]=-1;} Dt[1]=0; Zt[0]=Zt[Ntop+1]=m=Ntop+1; Ip=0;.
1	while(Zt[Pz[m]+Dt[m]]>m) {Dt[m]=-Dt[m];m--;}.
2	Использование перестановки
3	pm=Pz[m];dm=pm+Dt[m]; w=Zt[pm];Zt[pm]=Zt[dm];Zt[dm]=w;.
4	zpm=Zt[pm];w=Pz[zpm]; Pz[zpm]=pm;Pz[m]=w;.
5	//Конец.
6	m=Ntop; Ip++;.

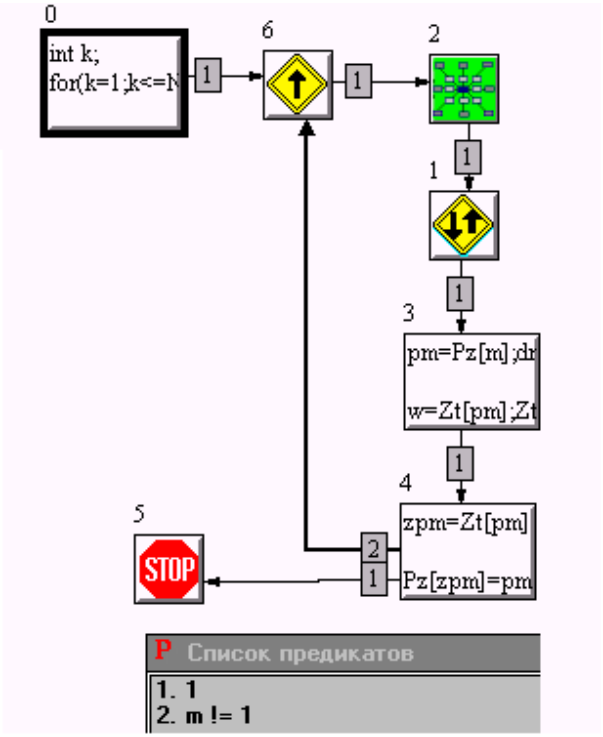


Рис. 3.23. Алгоритм построения перестановками

Таблица 3.4. Данные алгоритма эффективной перестановки

Группа инициируемых данных				
Имя данного	**Тип**	**Класс данного**	**Нач. значение**	**Комментарий**
Ntop	int	I	5	Размерность массива описания вершин графа

В модуле 1, при необходимости, производится изменение содержимого вектора *d*. В модуле 3 определяются номера транспозируемых элементов и формируется новая перестановка. В модуле 4 строится вектор обратной перестановки.

Таблица 3.5. Данные алгоритма эффективной перестановки

Группа вычисляемых данных				
Имя данного	**Тип**	**Класс данного**	**Нач. значение**	**Комментарий**
w	int	R	0	Промежуточный элемент
m	int	V	47	индекс
Ip	int	V	1	Число перестановок
Pz	MASSIV	V		Обратная перестановка
Dt	MASSIV	V		Вектор направления движения
pm	int	V	0	номер первого элемента пары перестановок
dm	int	R	0	номер второго элемента пары перестановок
zpm	int	R	0	номер элемента для обратной перестановки
Zt	MASSIV	V	{6,1,2,3,4,5,6}	Перестановки элементов

Данный алгоритм нельзя назвать эффективным с точки зрения сложности, поскольку он осуществляет полный перебор перестановок и имеет экспоненциальную сложность. Однако его можно назвать «изящным» алгоритмом, поскольку имеет очень компактный код.

3.4.4. Алгоритм полного перебора решения задачи о коммивояжере

В качестве примера рассмотрим задачу о коммивояжере*, представленную на рисунке 3.23 (описание графа на смежной памяти для данного примера приводится в таблице 3.4).

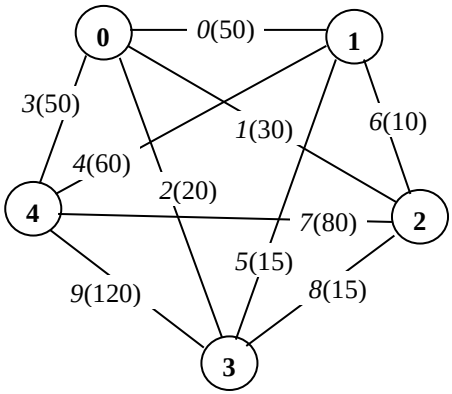


Рис. 3.24. Описание графа на смежной памяти

Для организации полного перебора гамильтоновых путей достаточно вместо вершины 2 алгоритма перебора всех перестановок подставить агрегат подсчета длины гамильтоновых путей, представленного на рисунке 3.25. Полное описание вершин графа содержится в таблице 3.6. Данные алгоритма описаны в таблицах 3.7 и 3.8.

---

* В Швеции было найдено оптимальное решение на многопроцессорном кластере (96 dual processor Intel Xeon 2.8 GHz). Если измерить время вычислений на одном процессоре «single Intel Xeon 2.8 GHz processor», то было бы потрачено 85 лет. См. отчет http://www.tsp.gatech.edu/sweden/index.html. Но это не отменяет высказанных соображений о непрактичности использования экспоненциальных алгоритмов перебора, кроме случаев для входных данных ограниченного размера.

Следует отметить, что в предложенном алгоритме учитывается, что имеется некоторое несовпадение в нумерации вершин графа в алгоритме перечисления перестановок, где вершины принципиально нумеруются от 1 до Ntop. В то время как, при описании графа удобнее нумеровать вершины от 0 до Ntop-1. Поэтому в алгоритме «Вычисление длины пути по списку вершин» двойная индексация вершин графа (m – для массива перестановок (Zt) и j – для позиционирования в описании вершин графа (Vtop)). В вершине 5 производится инициализация исходных данных (обнуляется переменная Lpath (длина пути), устанавливается начальная вершина гамильтонова цикла (Htop), индекс m). Перебор следует осуществлять, зафиксировав первую вершину, т.е организовав перестановку только для *n-1* вершин.

В вершине 0 – определяется номер (i) начала фрагмента описания (в массиве структур Gr) вершин инцидентных вершине Tp.

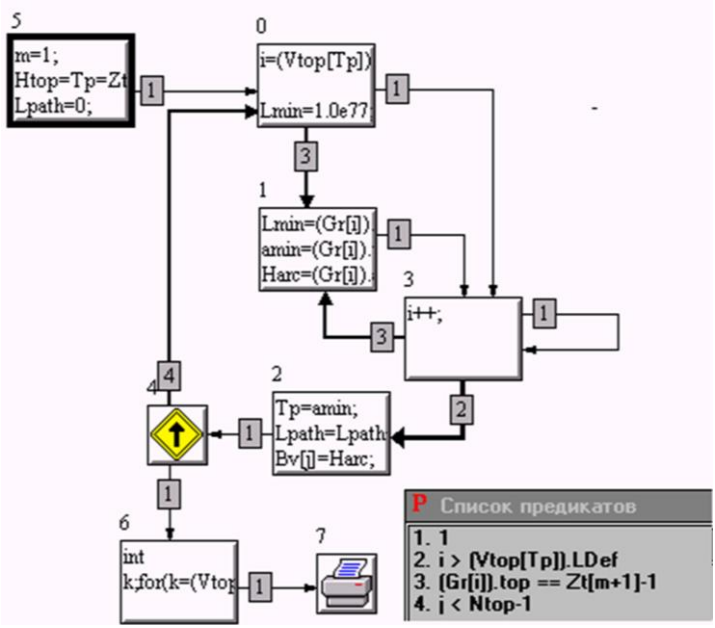


Рис. 3.25. Агрегат «Вычисление длины пути по списку вершин»

Таблица 3.6. Модули алгоритма «Вычисление длины пути по списку вершин»

№	Имя актора
0	i=(Vtop[Tp]).FDef; Lmin=1.0e77; .
1	Lmin=(Gr[i]).length; amin=(Gr[i]).top; Harc=(Gr[i]).arc;.
2	Tp=amin; Lpath=Lpath+Lmin; Bv[j]=Harc; Ztop[j+1]=Tp;.
3	i++;.
4	m++; j++;.
5	m=1; Htop=Tp=Zt[m]-1; Lpath=0; j=0;.
6	int k;for(k=(Vtop[Tp]).FDef;k<=(Vtop[Tp]).LDef;k++) {if((Gr[k]).top == Htop) {Lpath=Lpath+(Gr[k]).length; Bv[j]=(Gr[k]).arc; return 1;}}.
7	int k;printf("Gamilton path: \n"); for (k=0;k<Ntop;k++) printf(" %d",Bv[k]); printf(" Length %f \n",Lpath); getch();.

Таблица 3.7. Данные алгоритма «Вычисление длины пути по списку вершин»

Группа инициируемых данных				
Имя данного	**Тип**	**Класс данного**	**Нач. значение**	**Комментарий**
i	int	I	3	Счетчик
j	int	I	2	Цикл
Bv	MASSIV	I	{0,0,0}	Временный массив
Gr	MGRAF	I	{{0,1,50}, {1,2,30}, {2,3,20}, {3,4,50}, {0,0,50}, {4,4,60}, {5,3,15}, {6,2,10}, {6,1,10}, {1,0,30}, {7,4,80}, {8,3,15}, {8,2,15}, {5,1,15}, {2,0,20}, {9,4,120}, {9,3,120}, {7,2,80}, {4,1,60}, {3,0,50}}	Описание графа программы на смежной памяти
Vtop	MTOP	I	{{0,3}, {4,7}, {8,11}, {12,15}, {16,19}}	Описание списка вершин на смежной памяти
Ntop	int	I	5	Размерность массива описания вершин графа
Tp	int	I	0	Текущая вершина

Таблица 3.8. Данные алгоритма «Вычисление длины пути по списку вершин»

Группа вычисляемых данных				
Имя данного	**Тип**	**Класс данного**	**Нач. значение**	**Комментарий**
w	int	R	0	Промежуточный элемент
m	int	V	47	Индекс
amin	int	R	32000	Минимальный номер временного массива
Htop	int	R	0	Вершина начала Гамильтонова пути
Lmin	double	R	1.0e77	Дуга минимальной длины
Harc	int	R	0	Дуга на пути
Lpath	double	R	0	Длина пути коммивояжера
Ztop	MASSIV	R	{0, 1, 2, 3, 4}	Список запрещенных вершин
Zt	MASSIV	V	{6, 1, 2, 3, 4, 5, 6}	Перестановки элементов

Переход из модуля 0 в модуль 1 происходит только в том случае, если первая инцидентная исходной вершине, описанная в фрагменте массива Gr от (Vtop[Tp]).FDef до (Vtop[Tp]).LDef, является вершина, представленная перестановкой Zt[m]-1. В этом случае один раз выполняется модуль 1, и при этом определяется длина пути между вершинами Lmin. В противном случае в цикле 1-3 в фрагменте массива Gr от (Vtop[Tp]).FDef до (Vtop[Tp]).LDef находится нужный переход.

Модуль 2 увеличивает длину пути за счет очередного звена. Переменная Tp позиционируется на следующую вершину. В массивах Bv и Ztop записываются «пройденные» ребра и вершины гамильтонова пути.

В модуле 6 производится «замыкание» гамильтонова пути (учитывается последнее звено цикла).

Как уже говорилось сложность алгоритма полного перебора составляет $O(n!)$. Для нашего случая общее число перестановок $5!=120$. В таблице 3.9 приводится фрагмент результатов вычислительного эксперимента. Оптимальный маршрут №4 имеет длину 155. Как показали расчеты таких циклов в общем случае 10 вариантов.

Таблица 3.9. Фрагмент результатов вычислительного эксперимента

№	Путь (номера ребер)	Длина
1	0 6 8 9 3	245
2	0 6 7 9 2	280
3	0 4 7 8 2	225
4	**3 4 6 8 2**	**155**
5	3 0 6 8 9	245
	. . .	
17	0 5 9 7 1	295
	. . .	

3.4.5. Эвристический алгоритм решения задачи о коммивояжере

В разделе 3.3 было показано, что за счет изменения постановки задачи можно существенно повысить эффективность алгоритма. В частности, можно изменить и постановку задачи о коммивояжере, при этом искать не точное решение, а С-приближенное (квазиоптимальное).

Действительно, так уж ли важно найти точное решение? Во многих случаях достаточно найти пусть не оптимальное, но близкое к нему решение.

Определение 3.10. *Эвристикой, в теории алгоритмов обычно называют некоторый интуитивно-понятный алгоритм (или принцип построения алгоритмов). Эвристика может не гарантировать какую-либо точность решения, и не иметь никаких оценок времени работы, но часто применяется из-за хороших практических результатов.*

Рассмотрим алгоритм, когда из произвольной вершины ищется циклический путь, и на каждом шаге выбирается минимальное по длине ребро. При этом по мере формирования маршрута необходимо из списка инцидентных текущей вершине ребер исключать ребра, уже вошедшие в маршрут.

На первом шаге просматриваются *n-1* ребер. На втором – *n-2,* на j-м – *n-j* и т.д. Сложность выбора минимальной дуги на каждом шаге можно оценить величиной $C(n-j)$. Суммарно сложность построения квазиоптимального цикла составляет

$$\sum_{j=1}^{n-1} C(n-j) + C = C(\frac{n(n-1)+2}{2}).$$

В худшем случае необходимо просматривать *n-1* инцидентных ребер. Очевидно, что результат зависит от вершины, с которой мы начинаем строить гамильтонов цикл. Если предложенный алгоритм последовательно применить, начиная со всех вершин, то суммарная его сложность составит величину

$$Cn\frac{n(n-1)+2}{2} = C\frac{n^3 - n^2 + 2n}{2}.$$

Таким образом, предложенный элементарный эвристический алгоритм решения задачи о коммивояжере имеет полиномиальную сложность $O(n^3)$.

На рисунках 3.26, 3.27 приводится описание эвристического алгоритма поиска квази-минимального пути, где в качестве начальной рассматриваются все вершины графа. В таблицах 3.10, 3.11 – описание модулей алгоритма.

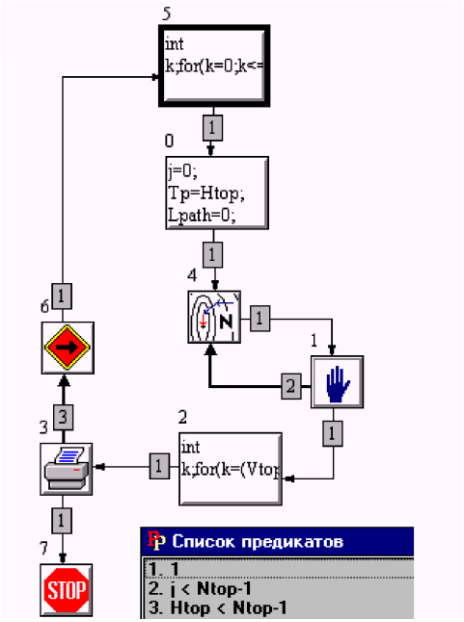


Рис. 3.26. Агрегат «Эвристический алгоритм задачи о коммивояжере»

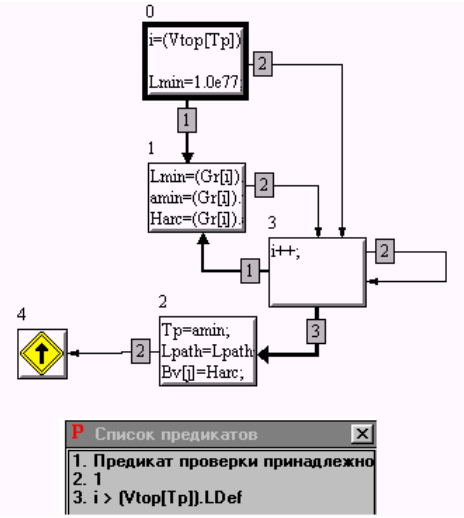


Рис. 3.27. Агрегат «Поиск следующего участка пути минимальной длины»

Таблица 3.10. Описание модуля алгоритма

№	Имя актора
0	j=0; Tp=Htop; Lpath=0; Ztop[j]=Htop;.
1	//Ветвление.
2	int k;for(k=(Vtop[Tp]).FDef;k<=(Vtop[Tp]).LDef;k++) {if((Gr[k]).top == Htop) {Lpath=Lpath+(Gr[k]).length; Bv[j]=(Gr[k]).arc; return 1;}}.
3	int k;printf("Gamilton path: \n"); for (k=0;k<Ntop;k++) printf(" %d",Bv[k]); printf(" Length %f \n",Lpath); getch();.
4	Поиск следующего участка пути минимальной дляны.
5	int k;for(k=0;k<=Ntop;k++){Ztop[k]=-1;}.
6	Htop++;.

Таблица 3.11. Описание модуля алгоритма

№	Имя актора
0	i=(Vtop[Tp]).FDef; Lmin=1.0e77; .
1	Lmin=(Gr[i]).length; amin=(Gr[i]).top; Harc=(Gr[i]).arc;.
2	Tp=amin; Lpath=Lpath+Lmin; Bv[j]=Harc; Ztop[j+1]=Tp;.
3	i++;.
4	j++;.

Предикат проверки принадлежности вершины *i* списку пройденных вершин цикла Ztop имеет вид:

```
#include "stype.h"
int predpat(MASSIV *Ztop,MGRAF *Gr,int *i, double *Lmin, int *Ntop)
 {
  int k;
  for (k=0;k<(*Ntop);k++)
      {if(((*Gr)[*i]).top == (*Ztop)[k])
                              return(0);}
  return(((*Gr)[*i]).length < (*Lmin));
 }
```

Результаты расчетов приведены в таблице 3.12 и один из гамильтоновых путей показан на рисунке 3.28.

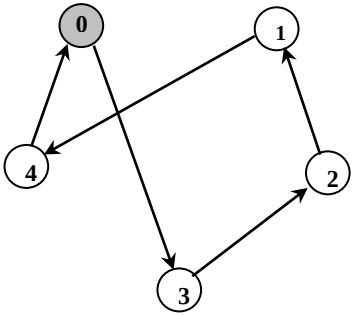


Рис. 3.28. Гамильтонов путь

Таблица 3.12. Результаты расчетов

№	Путь (номера ребер)	Длина
1	2 8 6 4 3	155
2	6 8 2 3 4	155
3	6 5 2 3 7	175
4	8 6 0 3 9	245
5	3 2 8 6 4	155

3.5. Жадные алгоритмы. Алгоритм Дейкстра

Теперь рассмотрим задачу, на первый взгляд – очень похожую на задачу «TSP».

Задача «Кратчайший путь в графе». *Заданы n вершин графа (узлов сети)* $v_1, v_2, \ldots, v_n$ *и положительные целые длины дуг* $d_{ij} = d(v_i, v_j)$ *между ними. Нужно для всех* $k \in (2,\ldots,n)$ *найти минимальную длину пути из* v_1 *в* v_k.

Разумеется, и эту задачу можно решать переборным алгоритмом, аналогичным алгоритму полного перебора для задачи TSP, но, интересно, можно ли разработать точный эффективный алгоритм, исключающий (или, по меньшей мере, минимизирующий) непосредственный перебор вариантов.

Оказывается, в данном случае можно. Здесь важным фактом является то, что, если у нас есть кратчайший путь от *v* до *w*, проходящий через вершину *y*, назовем его $(v \to w)^*$, то его первая часть от *v* до *y*, $(v \to y)^*$ тоже будет кратчайшим путем (см. рис. 3.29).

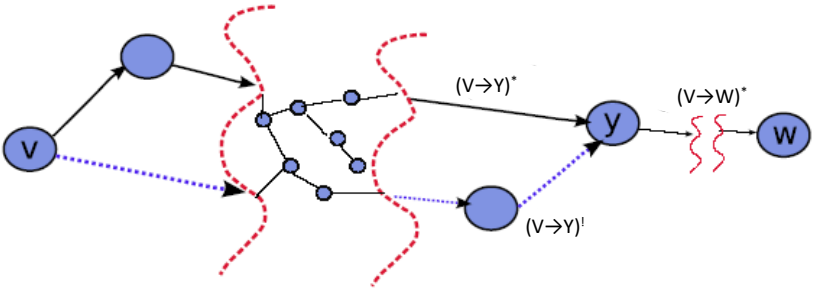


Рис. 3.29. Кратчайший путь в графе

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

Задачи с подобными свойствами, когда оптимальное решение можно легко получить из оптимальных решений подзадач, обычно хорошо решаются, так называемыми, «жадными алгоритмами». Одним из примеров, который может служить алгоритм Дейкстра построения минимального остовного дерева. Интересно, что различные варианты этого алгоритма используются при маршрутизации интернет-трафика (например, см. стандарты OSPF, Open Shortest Path First Routing Protocol, RFC 2740).

В алгоритме «Дейкстра» мы итерационно обслуживаем два множества вершин:

- *V* – множество вершин, до которых (на данном шаге алгоритма) мы уже нашли кратчайший путь. При этом вершины помечаются длинами путей, ведущих к ним от стартовой вершины.

❖ *W* – множество вершин, которые ***достижимы*** одной дугой из множества вершин *V*.

На каждом шаге алгоритма мы выбираем из достижимых вершин *W* вершину *y*, самую ближнюю к стартовой вершине *s*, при этом, если r_{ij} – ребро соединяющее вершины $v_i \leftrightarrow w_j$, а $\varphi(\cdot)$ – функция определения веса вершины или ребра, то

$$\varphi(y) = \min_{i,j} \{\varphi(v_i) + \varphi(r_{ij})\}\cdot$$

Тогда вершину *y* переносим из множества *W* в множество *V*, ($y \to V$). Далее мы увеличиваем множество «кандидатов» W за счет её соседей и пересчитываем верхнюю оценку удаленности вершин из множества W до вершины s.

На рисунке 3.30 приведен граф иллюстрирующий работу алгоритма Дейкстра. Здесь в вершинах графа записаны длины путей от исходной вершины.

Пусть, для представленного примера, остовное дерево строится, начиная из вершины 0 (корень дерева). Множество вершин *V* в этом случае состоит из одной вершины 0 с нулевой оценкой длины пути до самой себя ($V_1 = \{s_0\}$). На рисунке 3.30 в вершинах графа обозначается длина пути от текущей вершины до корневой. «Серые» вершины относятся к множеству *V*, «желтые» – к множеству *W*, «оранжевыми» обозначены вершины, для которых перевычисляется новый минимальный путь.

На первом шаге алгоритма находятся все смежные с множеством $V_1 = \{s_0\}$ (вершины графа (рис. 3.30, а), т.е. множество $W_1 = \{s_5, s_6, s_1\}$ Для каждой из вершин множества *W* (с учетом с изменившейся ситуации) определяется минимальный путь до корневой вершины. В данном случае это ребро (s_0, s_6). Вершина s_6 переносится в множество $V_2 = \{s_0, s_6\}$ и формируется часть минимального остовного дерева ($D: \{s_0 \to (s_6);\}$).

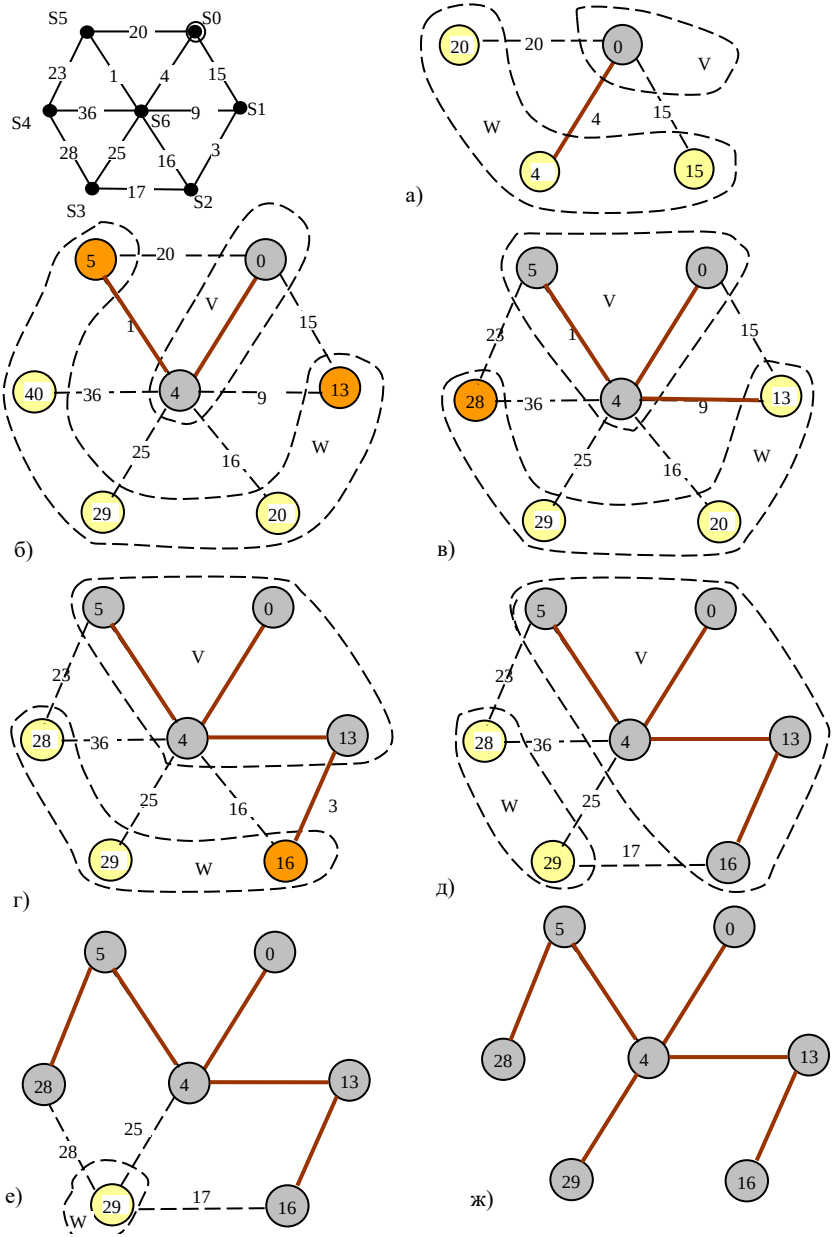


Рис. 3.30. Граф, иллюстрирующий работу алгоритма Дейкстра

На втором шаге для вершин множества *V* ищутся смежные к ним вершины (см. рис. 3.30, б) – множество $W_2 = \{s_5, s_4, s_3, s_2, s_1\}$. С учетом новых ребер перевычисляются минимальные расстояния до корневой вершины. Так, например, для вершины s_5 путь $s_5 \to s_0$ длины 20 можно заменить маршрутом $s_5 \to s_1 \to s_0$ длины 5. В связи с чем изменится метка вершины s_5. По той же причине изменится метка вершины s_1 с 15 на 13. В множестве $W_2 = \{ s_5, s_4, s_3, s_2, s_1 \}$ наименьшую метку имеет вершина s_5, поэтому она пополняет множество $V_3 = \{ s_0, s_6, s_5 \}$, а остовное дерево «разрастается» за счет ребра (s_5, s_6), обеспечивающего кратчайший путь из вершины s_5 в корень дерева. На этом шаге дерево принимает вид: $D: \{s_0 \to (s_6); s_6 \to (s_5);\}$, а $W_3 = \{s_4, s_3, s_2, s_1\}$ (см. рис. 3.30, в).

На следующем шаге с учетом новых связей между множествами V_3 и W_3 полученной маркировки вершин множества W_3 будет найдена минимальная вершина s_1 и т.д.

Минимальное остовное дерево имеет вид

$$D: \{s_0 \to (s_6);$$
$$s_6 \to (s_5, s_1, s_3);$$
$$s_5 \to (s_4);$$
$$s_1 \to (s_2);\}$$

(см. рис. 3.30, ж). По этому дереву легко найти минимальный путь из вершины s_0 в любую вершину графа. Метки вершин определяют длину пути.

Лемма 3.1. *Алгоритм «Дейкстры» корректен, т.е. все найденные им пути оптимальны.*

Доказательство

Докажем лемму методом математической индукции.

Для первого шага алгоритма (k=1), по построению множество $V_1 = \{s_0, s_{i_1}\}$ содержит корневую вершину и вершину s_{i_1} смежную с ней, имеющую минимальное расстояние. Обозначим через $\delta(v, w)$ длину **оптимального пути** из v в **w**. Следовательно $\delta(s_0, s_{i_1})$. Алгоритм Дейкстры на первом шаге порождает часть минимального дерева.

Предположим, что на k-м шаге было построено минимальное дерево путей от вершин $v \in V_k$ до корневой вершины s_0. Пусть W_k множество вершин смежных с вершинами части минимального дерева V_k. После перерасчета весов вершин $w \in W_k$, которые могут измениться за счет учета новых ребер, по алгоритму Дейкстры выбирается вершина для которой

$$w^* = \arg \min_{w \in W_k} \left(\min_{s_i \in V_k} \{\varphi(s_i) + \varphi(s_i \to w)\} \right). \qquad (3.7)$$

Вершина w^* пополняет множество V_k для следующего шага алгоритма $V_{k+1} = V_k \bigcup \{w^*\}$, т.е. содержит часть минимального остовного дерева $V_{k+1} = \{s_0, s_{i_1}, \ldots, s_{i_k}, w^*\}$.

Допустим, что существует иной более короткий путь до вершины w^* (см. рис. 3.31) через вершину y от одной из вершин множества V_k.

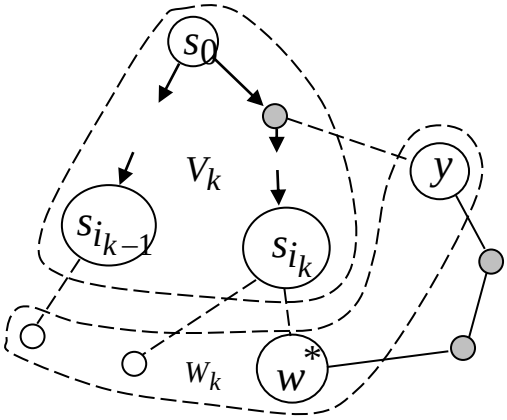


Рис. 3.31. Пути в графе

Поскольку y смежна с V_k, то $y \in W_k$ и имеет оценку $\varphi(y) > \varphi(w^*)$, иначе в качестве вершины, удовлетворяющей условию (3.7) была бы выбрана вершина *y*. Но в этих условиях дополнительные вершины смогли бы только увеличить длину пути до вершины w^*. Следовательно, других вершин нет. Тогда алгоритм Дейкстры корректен для *k+1* шага, а, следовательно, по индукции корректен в целом.

Лемма 3.2. *Трудоемкость алгоритма «Дейкстры» составляет* $O(n^2)$ *операций, где n — число вершин.*

Доказательство.

На каждом шаге работы алгоритма множество V_k увеличивается на одну вершину. При этом в худшем случае (полном графе) приходится просматривать $k-1$ смежных с минимальной вершин. Получается, что трудоемкость может быть оценена величиной $(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}$. Тогда эффективность алгоритма Дейкстры равна $O(n^2)$.

3.6. Метод ветвей и границ («поиск с возвратом» (backtracking))

Данный метод является одной из первых эффективных схем неявного (улучшенного) перебора. Идея метода состоит в том, что при решении экстремальной задачи можно избежать полного перебора путем отбрасывания заведомо неоптимальных решений [6].

В данном случае производится разбиение множества возможных вариантов решения дискретной экстремальной задачи на классы и формируется оценочная функция для каждого класса. В результате появляется возможность отбрасывать решения целыми классами, если их оценка уже хуже некоторого рекордного значения.

Основную проблему при использовании метода «ветвей и границ» составляет выбор способа разбиения множества решений на классы и выбор оценочной функции [7].

Пусть дано дискретное конечное множество $A=\{a_1,a_2,...,a_n\}$ и на нем определена вещественнозначная функция *f*. Требуется найти минимум этой функции и элемент множества на котором этот минимум достигается.

В общем случае задача решается с помощью полного перебора. Метод ветвей и границ призван сократить перебор. Он эффективен, когда выполняются специфические дополнительные условия на множество *A* и минимизируемую на нем функцию.

Предположим, что на множестве *A* существует вещественнозначная функция φ со следующими свойствами:

1. для любого i $\varphi(\{a_i\})=f(a_i)$. Здесь $\{a_i\}$ – множество, состоящее из единственного элемента a_i;
2. если $U\subseteq V\subseteq A$, то $\varphi(U)\geq\varphi(V)$.

Теперь мы можем организовать частичный перебор элементов с целью минимизации функции.

Разобьем множества *A* на части (любым способом), например, на две A_1 и A_2. Выберем ту из частей множества *A*, на котором функция φ минимальна, например, это множество A_1. Теперь множе-

ство A_1 разобьем на части и выберем ту из них, где минимальна φ и т.д., пока не придем к какому-либо одноэлементному множеству $\{a_i\}$.

Это одноэлементное множество $\{a_i\}$ называется ***рекордом***.

Функция φ, которая используется при этом выборе, называется ***оценочной***. Очевидно, что рекорд не обязан доставлять минимум функции *f*, но рекорд может служить ориентиром, с помощью которого можно сократить перебор, если создадутся соответствующие условия.

Описанный выше процесс построения рекорда фактически состоит из последовательности этапов, на каждом из которых формировалось несколько множеств и затем выбиралось одно из них. Пусть $A_1, A_2, \ldots, A_s$ – подмножества множества *A*, возникшие на последнем этапе формирования рекорда. Пусть подмножество A_1 оказалось выбранным с помощью оценочной функции, т.е. при разбиении множества A_1 возник рекорд $\{a_r\}$ и в этом случае справедливо $\varphi(A_1) \le \varphi(\{a_r\}) = f(a_r) = f^*$. Тогда $\varphi(A_1) \le \varphi(A_i) \quad i = 1, \ldots, s$

Предположим, что $\varphi(A_1) = f^* \le \varphi(A_2)$, но тогда в множестве A_2 не найдется элемента меньшего чем f^*, а, следовательно, множество A_2 не надо рассматривать. Если же условие $f^* \le \varphi(A_2)$ не будет выполнено, то все элементы из A_2 надо последовательно сравнить с найденным рекордом. Как только отыщется элемент, дающий меньшее значение оптимизируемой функции, надо им заменить рекорд и продолжить перебор. Последнее действие называется улучшением рекорда.

Идея метода ветвей и границ заключается в создании некоторой процедуры построения ограниченного дерева перебора. От исходного множества через его разбиения идут пути к рекорду и его улуч-

шениям. На дереве часть ветвей остаются «отсеченными», потому что их развитие оказалось нецелесообразным.

На самом деле метод ветвей и границ не является рабочей процедурой, а скорее идеологией, на основе которой для конкретных задач разрабатываются алгоритмы оптимизации. Любой новый способ построения дерева перебора и/или новая оценочная функция порождают новый алгоритм дискретных экстремальных задач.

3.7. Алгоритм Литтла решения задачи о коммивояжере

В качестве примера использования идей метода ветвей и границ рассмотрим задачу о коммивояжере для полного графа, представленного на рисунке 3.32.

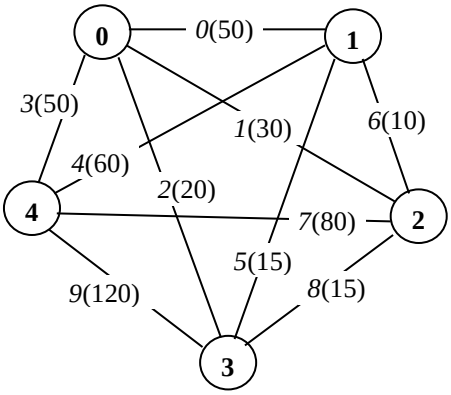


Рис. 3.32. Граф с пятью вершинами

Граф имеет следующую матрицу смежности:

$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} - & 50 & 30 & 20 & 50 \\ 50 & - & 10 & 15 & 60 \\ 30 & 10 & - & 15 & 80 \\ 20 & 15 & 15 & - & 120 \\ 50 & 60 & 80 & 120 & - \end{pmatrix} \end{array}.$$

Справедливо следующее: *вычитание любой константы из всех элементов любой строки или столбца матрицы A, оставляет минимальный тур коммивояжера минимальным.*

В связи с этим, процесс вычитания из каждой строки ее минимального элемента (назовем эту операцию **приведением по строкам**) не влияет на минимальный тур. Аналогично вводится понятие **приведения по столбцам**, обладающее тем же свойством.

Выполним операцию приведения исходной матрицы смежности по строкам, которое реализуется по формуле $\tilde{a}_{ij} = a_{ij} - \min\limits_{j} a_{ij}$.

	0	1	2	3	4	min
0	–	50	30	20	50	**20**
1	50	–	10	15	60	**10**
2	30	10	–	15	80	**10**
3	20	15	15	–	120	**15**
4	50	60	80	120	–	**50**

$A =$

и выполним операцию приведения:

	0	1	2	3	4
0	–	30	10	0	30
1	40	–	0	5	50
2	20	0	–	5	70
3	5	0	0	–	105
4	0	10	30	70	–
min	**0**	**0**	**0**	**0**	**30**

$A =$

После выполнения операции приведения матрицы смежности по столбцам имеем:

	0	1	2	3	4
0	–	30	10	0	0
1	40	–	0	5	20
2	20	0	–	5	40
3	5	0	0	–	75
4	0	10	30	70	–

$A =$. (3.8)

Смысл проделанных преобразований заключается в том, что мы таким образом выделяем множество ребер, на которых формируется минимальный тур. При этом преобразования графа (для новой матрицы расстояний) не нарушают условия формирования оптимального тура. Другими словами, оптимальный цикл останется тем же, но другой длины. При удачном стечении обстоятельств длина оптимального цикла для новой матрицы расстояний будет равна нулю. (Подумайте почему?)

В новой матрице оптимальный цикл необходимо искать на ребрах нулевой длины. Конечно, необходимо понимать, что не все ребра оптимального тура будут иметь нулевую длину. Но главное достижение предложенного Литтлом подхода заключается в возможности введения оценочной функции Φ и возможности применения для данной задачи метода ветвей и границ.

Для нулевых элементов матрицы A введем понятие **оценки нуля**.

Назовем *оценкой* нуля в позиции (i, j) в матрице сумму минимальных элементов в i-й строке и j-м столбце (не считая сам этот ноль), т.е.

$$k_{ij} = \min_{i(i \neq j)} a_{ij} + \min_{j(i \neq j)} a_{ij}.$$

Оценку нуля будем записывать в степени нуля, тогда матрицу (3.8) можно представить в виде:

$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} - & 30 & 10 & 0^{5} & 0^{20} \\ 40 & - & 0^{5} & 5 & 20 \\ 20 & 0^{5} & - & 5 & 40 \\ 5 & 0^{0} & 0^{0} & - & 75 \\ 0^{15} & 10 & 30 & 70 & - \end{pmatrix} \end{array} \qquad (3.9)$$

В нашем случае (с учетом направлений) для нулевых дуг получена конфигурация (см. рис. 3.33).

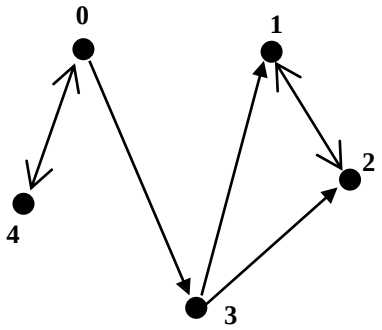


Рис. 3.33. Конфигурация нулевых дуг

Очевидно, что на этом материале невозможно построить гамильтонов цикл. Однако, с учетом преобразований уменьшающих длины ребер по строкам и столбцам исходной матрицы на (20+10+10+15+50)=105 и (0+0+0+0+30)=30 единиц, суммарная длина оптимального пути никак не может быть меньше 135.

Вернемся к оценкам нулевых ребер. Оценка *k* нуля, в позиции (*i*, *j*) означает буквально следующее: если в тур не будет включено ребро из вершины *i* в вершину *j* (стоимостью 0), то в любой тур придется доплатить как минимум *k* единиц длины. Тогда множество всевозможных туров можно разделить на два класса: туры $A_{(i-j)}$, содержащие ребро (*i*, *j*) и туры, его не содержащие $A_{\overline{(i-j)}}$. Для класса $A_{\overline{(i-j)}}$ минимальная оценка увеличится на *k* единиц.

Рассмотрим ребро, соответствующее нулю с максимальной оценкой. В данном случае это ребро (0-4). Таким образом, класс всех туров разбивается на два: содержащих ребро (0-4) и не содержащих его. Ребро с максимальной оценкой нуля выбирается из соображений достижения максимально возможной разницы для оценочной функции альтернативных классов.

В нашем случае нижняя оценка стоимости класса туров $A_{\overline{(0-4)}}$ увеличивается на 20, относительно 135 единиц до $\Phi(A_{\overline{(0-4)}}) = 155$ единиц.

Чтобы вычислить оценку для класса туров $A_{(0-4)}$ удалим из матрицы (3.9) строку 0 и столбец 4.

$$A_{(0-4)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 40 & - & 0 & 5 \\ 20 & 0 & - & 5 \\ 5 & 0 & 0 & - \\ - & 10 & 30 & 70 \end{pmatrix} \end{array} \begin{array}{c} min \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{10} \end{array}$$

После выполнения операции приведения по строкам получим:

$$A_{(0-4)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 40 & - & 0 & 5 \\ 20 & 0 & - & 5 \\ 5 & 0 & 0 & - \\ - & 0 & 20 & 60 \end{pmatrix} \end{array}.$$

$$min \quad \mathbf{5} \quad \mathbf{0} \quad \mathbf{0} \quad \mathbf{5}$$

Приведение матрицы по столбцам приводит к следующему результату:

$$A_{(0-4)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 35 & - & 0 & 0 \\ 15 & 0 & - & 0 \\ 0 & 0 & 0 & - \\ - & 0 & 20 & 55 \end{pmatrix} \end{array} \qquad (3.10)$$

При этом минимальный тур не может быть меньше, чем $135+10+5+5=155$, т.е. оценка класса туров $A_{(0-4)}$ составит $\Phi(A_{(0-4)}) = 155$ единиц.

Произведем оценку нулей матрицы (3.10).

$$A_{(0-4)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 35 & - & 0^{0} & 0^{0} \\ 15 & 0^{0} & - & 0^{0} \\ 0^{15} & 0^{0} & 0^{0} & - \\ - & 0^{20} & 20 & 55 \end{pmatrix} \end{array}$$

Далее выберем ребро с максимальной оценкой нуля: (4-1) и, вычеркнув соответствующие строки и столбцы, иполучим класс $A_{(0-4),(4-1)}$, для которой уже невозможно уменьшть длины ребер, поэтому $\Phi(A_{(0-4),(4-1)}) = 155$, а $\Phi(A_{(0-4),\overline{(4-1)}}) = 175$.

Для класса:

$$A_{(0-4),(4-1)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 2 & 3 \end{array} \\ \begin{pmatrix} 35 & 0 & 0 \\ 15 & - & 0 \\ 0 & 0 & - \end{pmatrix} \end{array} \qquad (3.11)$$

оценим нули матрицы (3.11):

$$A_{(0-4),(4-1)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 2 & 3 \end{array} \\ \begin{pmatrix} 35 & 0^{0} & 0^{0} \\ 15 & - & 0^{0} \\ 0^{15} & 0 & - \end{pmatrix} \end{array}$$

Далее, выберем ребро (3-0), тогда имеем:

$$A_{(0-4),(4-1),(3-0)} = \begin{array}{c} \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{cc} 2 & 3 \end{array} \\ \begin{pmatrix} 0 & - \\ - & 0 \end{pmatrix} \end{array}$$

В последней матрице все ребра нулевые и они без увеличения оценочной функции формируют гамильтонов цикл $P = [(0-4),(4-1),(3-0),(1-2),(2-3)]$ причем $\Phi(A_P) = 155.$ Последнее означает, что любые классы туров с оценкой $F(A_*) \le \Phi(A_P) = 155$ можно далее не рассматривать. В то время как, $\Phi(A_{(0-4),(4-1),\overline{(3-0)}}) = 170$.

Очевидно, что классы туров с оценками $\Phi(A_{(0-4),\overline{(4-1)}}) = 175$ и $\Phi(A_{(0-4),(4-1),\overline{(3-0)}}) = 170$ можно далее не рассматривать. Оптимальный гамильтонов цикл найден. Дальнейшие исследования класса туров $A_{\overline{(0-4)}}$ не может улучшить найденную оценку. Алгоритм Литтла прекращает свою работу.

На рисунке 3.34 представлено дерево классов туров, иллюстрирующее работу метода ветвей и границ. Классы туров имеют простую геометрическую интерпретацию, представленную на рисунке 3.35. Так, например, класс туров $A_{(0-4)}$ – это все гамильтоновы циклы, которые можно построить на графе $A_{(0-4)}$ рисунка 3.35, обязательно используя ребро (0-4). В тоже время, класс туров $A_{\overline{(0-4)}}$ – это все гамильтоновы циклы, не использующие ребро (0-4) (см. рис 3.35).

Удовлетворительных теоретических оценок сложности алгоритма Литтла и ему подобных нет, но практика показывает, что на современных машинах они позволяют решать задачу о коммивояжере с количеством вершин ≈ 100. Последнее подтверждает, что алгоритмы, использующие метод ветвей и границ, являются эффективными эвристическими процедурами.

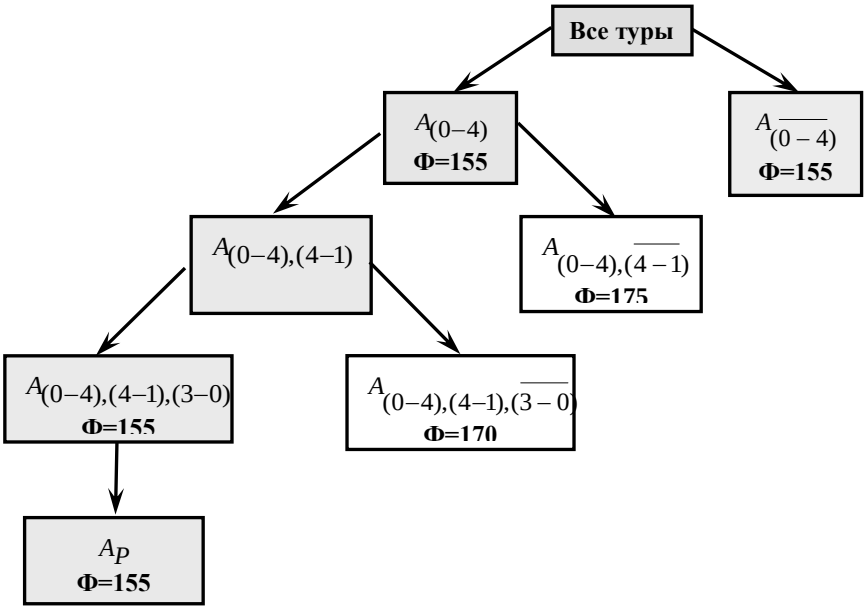


Рис. 3.34. Дерево классов туров

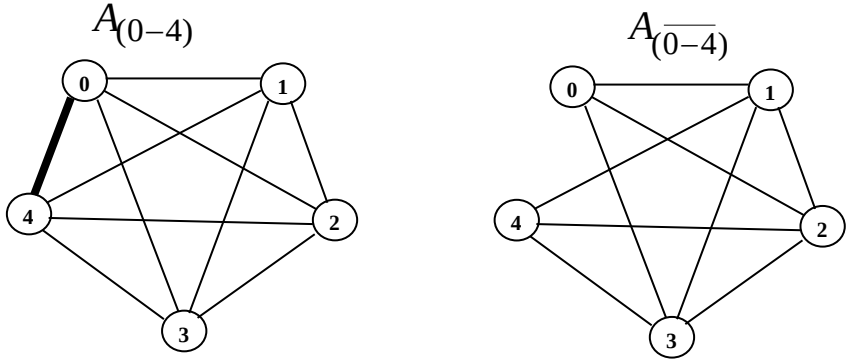


Рис. 3.35. Интерпретация классов туров

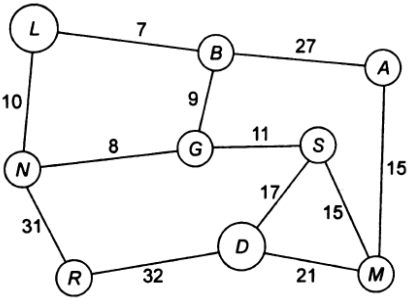
3.7.1. Задачи

Задача 3.17. Для перестановки $(2,4,1,5,3,6)$ постройте обратную перестановку.

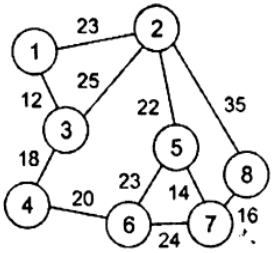
Задача 3.18. Для обратной перестановки $(7,3,6,5,4,2,1)$ постройте исходную перестановку.

Задача 3.19. Описать граф, представленный на рисунке 3.33, с помощью структур смежности графа на смежной памяти.

Задача 3.20. Найти кратчайший путь на графе от вершины L до вершины D.



Задача 3.21. Найти кратчайший путь на графе от 1-ой до 7-ой вершины.



Упражнение 3.22. С помощью алгоритма Дейкстра постройте минимальное остовное дерево для графа, представленного на рисунке 3.33.

3.8. Вопросы для самопроверки

1. Для чего нужно связанное представление последовательностей?
2. Объясните понятие характеристического вектора.
3. Идея метода и сложность алгоритма Флойда.
4. Случаи применения логарифмического поиска.
5. Эффективность алгоритмов оптимизации.
6. Траектория алгоритма оптимизации и ее трудоемкость.
7. Понятие С-приближенного алгоритма.
8. Отрезок неопределенности поиска.
9. Особенности оптимизации многоэкстремальных функций.
10. Способы представления графов.
11. Задача о коммивояжере. Способы решения.
12. Идея алгоритма эффективного порождения перестановок.
13. Понятие «жадных алгоритмов».
14. Идея алгоритма Дейкстра.
15. Суть метода ветвей и границ.

Список литературы

1. Кузюрин, Н.Н. Эффективные алгоритмы и сложность вычислений [Текст] / Н.Н. Кузюрин, С.А. Фомин. URL: http://discopal.ispras.ru/ru.book-advanced-algorithms.htm (дата обращения 22.07.2018)

2. Коварцев, А.Н. Автоматизация разработки и тестирования программных средств [Текст]: учеб. пособие / А.Н. Коварцев. – Самара: Издательство СГАУ, 1999. – 197 с.

3. Кузнецов, О.П. Дискретная математика для инженера [Текст]: учебник для вуза / О.П. Кузнецов, Г.М. Адельсон-Вельский – СПб.: Издательство «Лань», 2009. – 400 с.

4. Немировский, А.С. Сложность и эффективность методов оптимизации [Текст]: учебник / А.С. Немировский, Д.Б. Юдин – М.: Наука, 1979. – 384 с.

5. Иванов, Б.Н. Дискретная математика. Алгоритмы и программы [Текст]: учеб. пособие / Б.Н. Иванов. – М.: Лаборатория базовых знаний, 2003. – 288 с.

6. Поликарпова, Н. Дискретная математика: Алгоритмы. // Сайт Санкт-петербургского государственного университета информационных технологий, механики и оптики / Н. Поликарпова, А. Гераси-менко. URL: http://rain.ifmo.ru/cat/view.php/theory/unsorted/approx-2004 (дата обращения 14.09.2018)

7. Гагарина, Л.Р., Колдаев В.Д. Алгоритмы и структуры данных [Текст]: учеб. пособие / Л.Р. Гагарина, В.Д. Колдаев – М.: Издательство «Финансы и статистика», 2009. – 304 с.

Алфавитный указатель

Учебное издание

Коварцев Александр Николаевич
Даниленко Александра Николаевна

АЛГОРИТМЫ И АНАЛИЗ СЛОЖНОСТИ

Учебник

Редактор Т.К. Кретинина
Компьютерная вёрстка И.И. Спиридоновой

Подписано в печать 6.11.2018. Формат 60 × 84 1/16.
Бумага офсетная. Печ. л. 8,0.
Тираж 100 экз. Заказ . Арт. – 1(Р4У)/2018.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34.

Изд-во Самарского университета.
443086, Самара, Московское шоссе, 34.