

ОСНОВЫ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Н. И. Костюкова

Графы и их применение. Комбинаторные алгоритмы для программистов



Основы информационных технологий

Н. И. Костюкова

Графы и их применение. Комбинаторные алгоритмы для программистов

Учебное пособие



Интернет-Университет
Информационных Технологий
www.intuit.ru



БИНОМ.
Лаборатория знаний
www.lbz.ru

Москва
2007

УДК 519.1(07)
ББК 22.12я7+22.176я7
К72

Костюкова Н. И.

К72 Графы и их применение. Комбинаторные алгоритмы для программистов: Учебное пособие / Н. И. Костюкова. — М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. — 311 с.: ил., табл.— (Серия «Основы информационных технологий»)

ISBN 978-5-94774-545-0 (БИНОМ. ЛЗ)

ISBN 978-5-9556-0069-7 (ИНТУИТ.РУ)

Содержание учебника разделяется на две части. Первая часть посвящена изучению теории графов. Она включает в себя такие темы, как связность, деревья, эйлеровы и гамильтоновы цепи и циклы, бесконечные графы, планарные графы и раскрашивание графов, где особенно выделены вопросы, связанные с гипотезой четырех красок, теория ориентированных графов, каркасы и изоморфизм деревьев.

Содержание второй части учебника посвящено комбинаторным методам вычисления. Рассматриваются классы алгоритмов, их анализ. Большое внимание уделено последовательному распределению, связному распределению, множествам и мультимножествам, рекуррентным соотношениям, алгоритмам рекуррентных соотношений, производящим функциям, всем типам поиска и сортировок.

УДК 519.1(07)
ББК 22.12я7+22.176я7

Издание осуществлено при финансовой и технической поддержке издательства «Открытые Системы», «РМ Телеком» и Kraftway Computers.

Полное или частичное воспроизведение или размножение каким-либо способом, в том числе и публикация в Сети, настоящего издания допускается только с письменного разрешения Интернет-Университета Информационных Технологий.



По вопросам приобретения обращаться:
«БИНОМ. Лаборатория знаний»
Телефон (499) 157-1902, (495) 157-5272,
e-mail: Lbz@aha.ru, <http://www.Lbz.ru>

ISBN 978-5-94774-545-0 (БИНОМ. ЛЗ)
ISBN 978-5-9556-0069-7 (ИНТУИТ.РУ)

© Интернет-Университет
Информационных
Технологий, 2007
© БИНОМ. Лаборатория
знаний, 2007

О проекте

Интернет-Университет Информационных Технологий — это первое в России высшее учебное заведение, которое предоставляет возможность получить дополнительное образование во Всемирной сети. Web-сайт университета находится по адресу www.intuit.ru.

Мы рады, что вы решили расширить свои знания в области компьютерных технологий. Современный мир — это мир компьютеров и информации. Компьютерная индустрия — самый быстрорастущий сектор экономики, и ее рост будет продолжаться еще долгое время. Во времена жесткой конкуренции от уровня развития информационных технологий, достижений научной мысли и перспективных инженерных решений зависит успех не только отдельных людей и компаний, но и целых стран. Вы выбрали самое подходящее время для изучения компьютерных дисциплин. Профессионалы в области информационных технологий сейчас востребованы везде: в науке, экономике, образовании, медицине и других областях, в государственных и частных компаниях, в России и за рубежом. Анализ данных, прогнозы, организация связи, создание программного обеспечения, построение моделей процессов — вот далеко не полный список областей применения знаний для компьютерных специалистов.

Обучение в университете ведется по собственным учебным планам, разработанным ведущими российскими специалистами на основе международных образовательных стандартов Computer Curricula 2001 Computer Science. Изучать учебные курсы можно самостоятельно по учебникам или на сайте Интернет-Университета, задания выполняются только на сайте. Для обучения необходимо зарегистрироваться на сайте университета. Удостоверение об окончании учебного курса или специальности выдается при условии выполнения всех заданий к лекциям и успешной сдачи итогового экзамена.

Книга, которую вы держите в руках, — очередная в многотомной серии «Основы информационных технологий», выпускаемой Интернет-Университетом Информационных Технологий. В этой серии будут выпущены учебники по всем базовым областям знаний, связанным с компьютерными дисциплинами.

**Добро пожаловать в
Интернет-Университет Информационных Технологий!**

Анатолий Шкред
anatoli@shkred.ru

Об авторе

Костюкова Нина Ивановна

Кандидат технических наук, доцент кафедры «Программирование» механико-математического факультета Новосибирского государственного университета, старший научный сотрудник Института вычислительной математики и математической геофизики СО РАН. Специалист в области языков программирования, автор ряда курсов и учебных пособий по теории и языкам программирования.

Оглавление

Часть I. Графы и их применение

Лекция 1. Основные понятия теории графов	13
Определение графа	13
Определение орграфа	13
Полный граф	14
Полный ориентированный граф	14
Двудольный граф	14
Степень вершины	15
Связность графа	16
Задачи, приводящие к графам	18
Лекция 2. Некоторые определения теории графов	21
Определения и примеры	21
Удаление ребер, мосты	23
Деревья	23
Перечисление деревьев	25
Лекция 3. Представления о планарном графе	28
Плоский граф	28
Гомеоморфные графы	30
Формула Эйлера	31
Триангулированный граф	32
Задачи	33
Лекция 4. Эйлеровы графы	35
Эйлеровы графы	35
Лабиринты	39
Геометрическая постановка задачи о лабиринтах	40
Решение задачи о лабиринтах	40
Лекция 5. Гамильтоновы графы	44
Гамильтоновы графы	44
Теорема Дирака	48
Лекция 6. Бесконечные графы	49
Бесконечные графы	49
Краткий обзор свойств бесконечных эйлеровых графов	52

Лекция 7. Графы с цветными ребрами	54
Реберная раскраска	54
Задачи на графы с цветными ребрами и вытекающие из них свойства	56
Задача о несцепленных треугольниках с одноцветными сторонами	63
Лекция 8. Раскрашивание графов	66
Хроматическое число	66
Гипотеза о четырех красках	68
Раскрашивание карт	69
Лекция 9. Орграфы	71
Определения	71
Эйлеровы и гамильтоновы орграфы	74
Турниры	75
Лекция 10. Цепи Маркова	79
Еще раз об ориентированных графах	79
Задачи на круговые бескомпромиссные турниры	80
Цепи Маркова	82
Лекция 11. О деревьях	87
Представления деревьев	87
Представление с помощью матрицы смежности	87
Представление с помощью списков смежности	88
Представление с помощью списка ребер и кода Прюфера	89
Алгоритм построения кода Прюфера	89
Алгоритм раскодирования	90
Уровневые коды корневых деревьев	90
Перечисление и подсчет деревьев	91
Непомеченные деревья	91
Ориентированные деревья	92
Каркасы в неориентированном графе	92
Каркасы в ориентированных графах	93
Лекция 12. Каркасы и изоморфизм деревьев	95
Каркас неориентированного графа	95
Нахождение каркасов в графе	95
Алгоритм Краскала	96
Изоморфизм деревьев	97
Лекция 13. Деревья, вероятность и генетика	100
Отыскание кратчайшего пути	100
Вероятность и генетика	100

Лекция 14. Сетевое планирование и управление	104
Введение	104
Сетевой график	104
Правила построения сетевого графика	108
Анализ сетевой модели	110
Определение критического пути	112
Определение полного резерва времени ненапряженного пути	112
Формирование временных оценок работ	113
Лекция 15. Паросочетания и свадьбы	115
Паросочетания и свадьбы	115
Теорема Холла о свадьбах	115
Приложение теоремы Холла	118
Латинские квадраты	118
Лекция 16. Теория трансверсалей	120
Теория трансверсалей	120
Приложение теории трансверсалей	122
Лекция 17. Потоки в сетях	126
Потоки в сетях	126
Приложение	129
 Часть II. Комбинаторные алгоритмы для программистов	
Лекция 1. Комбинаторные вычисления	133
Введение	133
Проблема представления: коды, сохраняющие разности	134
Классы алгоритмов	135
Анализ алгоритмов	139
Программа	140
Лекция 2. Целые и последовательности (последовательное распределение)	142
Введение	142
Целые	142
Последовательности	145
Различные способы представлений конечных последовательностей (или начальных сегментов бесконечных последовательностей) и операции над ними	145
Лекция 3. Последовательности (связанное распределение, стеки и очереди)	149
Связанное распределение	149
Разновидности связанных списков	152

Стеки и очередь	153
Задачи	153
Программы	154
Лекция 4. Последовательности (деревья)	159
Деревья	159
Представления	161
Прохождения	163
Длина путей	165
Задача	165
Программа	165
Лекция 5. Комбинаторика разбиений	168
Введение	168
Задачи	168
Разные статистики	170
Деревья и перестановки из n элементов	171
Число сочетаний C_n^m	171
Задачи на разбиение чисел	172
Комбинаторные задачи теории информации	175
Лекция 6. Последовательности (множества и мультимножества)	176
Множества и мультимножества	176
Формула включений и исключений	180
Решето Эратосфена	180
Примеры программы	182
Лекция 7. Рекуррентные соотношения	185
Размещения без повторов	185
Перестановки	185
Сочетания	185
Рекуррентные соотношения	186
Другой метод доказательства	189
Процесс последовательных разбиений	189
Задача: «Затруднение мажордома»	190
Лекция 8. Алгоритмы рекуррентных соотношений	194
Решение рекуррентных соотношений	194
Линейные рекуррентные соотношения с постоянными коэффициентами	195
Случай равных корней характеристического уравнения	198
Производящие функции	200

Лекция 9. Комбинаторика и ряды	203
Введение	203
Деление многочленов	203
Алгебраические дроби и степенные ряды	204
Действия над степенными рядами	206
Лекция 10. Производящие функции и рекуррентные соотношения	209
Применение степенных рядов для доказательства тождеств . . .	209
Производящие функции	210
Бином Ньютона	211
Ряд Ньютона	212
Производящие функции и рекуррентные соотношения	212
Об едином нелинейном рекуррентном соотношении	214
Лекция 11. Алгоритмы на абстрактных структурах данных	215
Введение	215
Стеки	215
Очереди	219
Связанные списки	220
Двоичные деревья	225
Лекция 12. Что такое граф? Определения и примеры	228
Введение	228
Представления	229
Связность и расстояние	232
Остовные деревья	233
Клики	234
Изоморфизм	234
Планарность	235
Лекция 13. Поиск	238
Введение	238
Поиск и другие операции над таблицами	238
Последовательный поиск	241
Логарифмический поиск в статических таблицах	243
Бинарный поиск	244
Оптимальные деревья бинарного поиска	246
Логарифмический поиск в динамических таблицах	248
Сбалансированные сильно ветвящиеся деревья	249
Лекция 14. Сортировка (часть 1)	251
Введение	251
Внутренняя сортировка	252
Вставка	253
Обменная сортировка	254

Лекция 15. Сортировка (часть 2)	260
Выбор	260
Распределяющая сортировка	262
Цифровая распределяющая сортировка	264
Внешняя сортировка	265
Частичная сортировка	267
Частичная сортировка (выбор)	267
Частичная сортировка (слияние)	268
Лекция 16. Алгоритмы на графах	269
Поиск в глубину	269
Алгоритм Дейкстры нахождения кратчайшего пути	270
Алгоритм Флойда нахождения кратчайших путей между парами вершин	270
Программы	272
Лекция 17. Калейдоскоп из комбинаторных алгоритмов	280
Автоматическое построение лабиринтов	280
Бинарное дерево	285
Задача о восьми ферзях	287
Сортировки	290
Задача о назначениях (задачи выбора)	298
Ханойская башня	305
Литература	311

Часть I

Графы и их применение

Лекция 1. Основные понятия теории графов

Определение графа. Определение орграфа. Полный граф. Полный ориентированный граф. Двудольный граф. Степень вершины. Связность графа. Задачи, приводящие к графам

Ключевые слова: граф, вершины, ребра, множество вершин, семейство ребер, орграф, дуга, дуга из v в w , ориентированное ребро, петля, полный граф, дополнение графа G , двудольный граф, полный двудольный граф, звездный граф, изоморфные графы, степень вершины, валентность вершины, нечетная вершина, четная вершина, смежные вершины, вершины, инцидентные ребру, ребро, инцидентное вершинам, регулярный граф, изолированная вершина, висячая или концевая вершина, лемма о рукопожатиях, маршрут в графе, начальная вершина, конечная вершина, длина маршрута, цепь, простая цепь, цикл, связный граф, несвязный граф, компонента связности, путь, простой путь, простой цикл.

Определение графа

Графом G называется пара $(V(G), E(G))$, где $V(G)$ — непустое конечное множество элементов, называемых **вершинами**, а $E(G)$ — конечное семейство неупорядоченных пар элементов из $V(G)$ (необязательно различных), называемых **ребрами**. Употребление слова «семейство» говорит о том, что допускаются кратные ребра. Будем называть $V(G)$ **множеством вершин**, а $E(G)$ — **семейством ребер** графа G . О каждом ребре вида $\{v, w\}$ говорят, что оно соединяет вершины v и w . Каждая петля $\{v, v\}$ соединяет вершину v саму с собой.

При изображении графов на рисунках или схемах отрезки могут быть прямолинейными или криволинейными; длины отрезков и расположение точек произвольны.

Определение орграфа

Орграфом D называется пара $(V(D), A(D))$, где $V(D)$ — непустое конечное множество элементов, называемых **вершинами**, а $A(D)$ — конечное семейство упорядоченных пар элементов из $V(D)$, называемых **дугами** (или **ориентированными ребрами**). Дуга, у которой вершина v является первым элементом, а вершина w — вторым, называется **дугой из v в w** (v, w). Заметим, что дуги (v, w) и (w, v) различны. Хотя графы и орграфы — существенно различные объекты, в определенных случаях графы

можно рассматривать как орграфы, в которых каждому ребру соответствуют две противоположно ориентированные дуги.

Полный граф

Граф называется **полным**, если каждые две различные вершины его соединены одним и только одним ребром. В полном графе каждая его вершина принадлежит одному и тому же числу ребер. Для задания полного графа достаточно знать число его вершин. Полный граф с n вершинами обычно обозначается через K_n .

Граф, не являющийся полным, можно преобразовать в полный с теми же вершинами, добавив недостающие ребра. Вершины графа G и ребра, которые добавлены, тоже образуют граф. Такой граф называют дополнением графа G и обозначают его \bar{G} .

Дополнением графа G называется граф \bar{G} с теми же вершинами, что и граф G , и с теми и только теми ребрами, которые необходимо добавить к графу G , чтобы получился полный граф.

Является граф полным или нет, это его характеристика в целом.

Полный ориентированный граф

Полным ориентированным графом называется граф, каждая пара вершин которого соединена в точности одним ориентированным ребром. Если с каждого ребра полного ориентированного графа снять направление, то образуется полный граф с неориентированными ребрами.

Рассмотрим соревнование, в котором каждая из команд играет с каждой из остальных команд по одному разу. Такое соревнование называют круговым турниром или турниром в один круг.

Если каждая встреча непременно должна оканчиваться выигрышем одной из команд, то круговой турнир называют бескомпромиссным. Круговой бескомпромиссный турнир проводится, например, в волейболе и баскетболе.

$$\{v, w\}$$

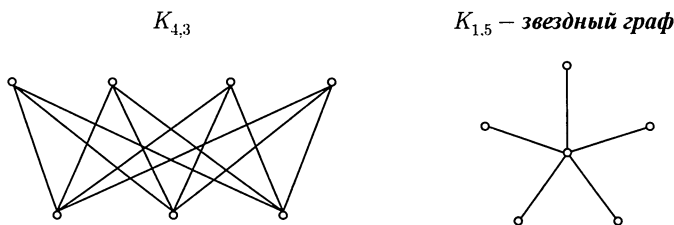
Каждому турниру соответствует полный ориентированный граф, в котором вершины представляют команды, а каждое ориентированное ребро $\{v, w\}$ выражает отношение « v победила w ».

Двудольный граф

Допустим, что множество вершин графа можно разбить на два непесекающихся подмножества V_1 и V_2 , так, что каждое ребро в G соединяет

какую-нибудь вершину из V_1 с какой-либо вершиной из V_2 , тогда G называем **двудольным графом**. Такие графы иногда обозначают $G(V_1, V_2)$, если хотят выделить два указанных подмножества. Двудольный граф можно определить и по-другому: в терминах раскраски его вершин двумя цветами, скажем, красным и синим. При этом граф называется двудольным, если каждую его вершину можно окрасить красным или синим цветом так, чтобы любое ребро имело один конец красный, а другой — синий. Следует подчеркнуть, что в двудольном графе совсем не обязательно каждая вершина из V_1 соединена с каждой вершиной из V_2 ; если же это так и если при этом граф G простой, то он называется **полным двудольным графом** и обычно обозначается $K_{m,n}$, где m, n — число вершин соответственно в V_1 и V_2 .

Пример.



Заметим, что граф $K_{m,n}$ имеет ровно $m + n$ вершин и nm ребер.

Степень вершины

Вершины в графе могут отличаться друг от друга тем, скольким ребрам они принадлежат.

Степенью вершины называется число ребер графа, которым принадлежит эта вершина. Вершина называется **четной**, если ее степень — число четное. Вершина называется **нечетной**, если ее степень — число нечетное. Две вершины графа называются **смежными**, если существует соединяющее их ребро, то есть ребро вида $\{v, w\}$; при этом **вершины** v и w называются **инцидентными** этому **ребру**, а **ребро** — **инцидентным** этим **вершинам**. Аналогично, **два различных ребра графа называются смежными**, если они имеют, по крайней мере, одну общую вершину. Иначе можно определить степень вершины. **Степенью или валентностью вершины** v графа G называется число ребер, инцидентных v ; степень вершины будем обозначать через $\rho(v)$. При вычислении степени вершины v будем учитывать петлю в v два раза, а не один. Вершина степени 0 называется **изолированной вершиной**, вершина степени 1 называется **висячей**, или **концевой**, вершиной.

Граф, у которого все вершины имеют одну и ту же степень, называется **регулярным графом**.

Два графа, G_1 и G_2 , называются **изоморфными**, если существует взаимно однозначное соответствие между множествами их вершин, обладающее тем свойством, что число ребер, соединяющих любые две вершины в G_1 равно числу ребер, соединяющих соответствующие вершины в G_2 .

Имея даже общие представления о графе, иногда можно судить о степенях его вершин. Так, степень каждой вершины полного графа на единицу меньше числа его вершин. При этом некоторые закономерности, связанные со степенями вершин, присущи не только полным графам.

1. В графе G сумма степеней всех его вершин — число четное, равное удвоенному числу ребер графа, так как каждое ребро участвует в этой сумме ровно два раза. Этот результат, известный еще двести лет назад Эйлеру, часто называют **леммой о рукопожатиях**. Из нее следует, что если несколько человек обменялись рукопожатиями, то общее число пожатых рук обязательно четно, ибо в каждом рукопожатии участвуют две руки (при этом каждая рука считается столько раз, сколько она участвовала в рукопожатиях).

2. Число нечетных вершин любого графа четно.

3. Во всяком графе с n вершинами, где $n \geq 2$, всегда найдутся по меньшей мере две вершины с одинаковыми степенями.

4. Если в графе с вершинами $n > 2$ в точности две вершины имеют одинаковую степень, то в этом графе всегда найдется либо в точности одна вершина степени 0, либо в точности одна вершина степени $n - 1$.

Связность графа

Назовем **граф связным**, если его нельзя представить в виде объединения двух графов, и **несвязным** — в противном случае.

Маршрутом в данном графе G называется конечная последовательность ребер вида

$$\{v_0, v_1\}, \{v_1, v_2\}, \dots, \{v_{m-1}, v_m\}$$

(обозначаемая также через $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_m$). Очевидно следующее свойство маршрута: любые два последовательных ребра либо смежны, либо одинаковы. Но не всякая последовательность ребер, обладающая этим свойством, является маршрутом (в качестве примера рассмотрим звездный граф и возьмем его ребра в произвольном порядке). Каждому маршруту соответствует последовательность вершин $v_0, v_1 \dots v_m$. v_0 называется **начальной вершиной**, а v_m **конечной вершиной** маршрута. Таким

образом, мы будем говорить о маршруте из v_0 в v_m . Заметим, что для любой вершины v_0 тривиальным маршрутом, вообще не содержащим ребер, является маршрут из v_0 в v_0 . *Длиной маршрута* называется число ребер в нем. Маршрут называется *цепью*, если все его ребра различны, и *простой цепью*, если все вершины v_0, \dots, v_m различны, кроме, может быть, $v_0 = v_m$. Замкнутая простая цепь, содержащая по крайней мере одно ребро, называется *циклом*. В частности, любая петля или любая пара кратных ребер образует цикл. *Путь* (последовательность ребер, ведущая от e_1 к e_2 , в которой каждые два соседних ребра имеют общую вершину и никакое ребро не встречается более одного раза) от вершины e_1 до вершины e_2 называется *простым*, если он не проходит ни через одну из вершин графа более одного раза.

Граф G называется *связным*, если для любых двух его вершин v и w существует простая цепь из v в w . Любой граф можно разбить на непересекающиеся связные графы, называемые *компонентами (связности)* графа G . Таким образом, несвязный граф имеет, по крайней мере, две компоненты. Две вершины *эквивалентны* (или *связаны*), если существует простая цепь из одной в другую. Связный граф состоит из одной компоненты. Граф называется *несвязным*, если число его компонент больше единицы.

Связный граф представляет собой *простой цикл* тогда и только тогда, когда каждая его вершина имеет степень 2.

Если $G(V, E)$ — связный граф и степень каждой его вершины 2, тогда $G(V, E)$ — простой цикл.

Доказательство. Из каждой вершины данного графа в любую другую ведет путь. Начнем путь из какой-нибудь вершины e и пройдем по одному из двух ребер, которым принадлежит эта вершина. Попад в вторую вершину, выйдем из нее по второму ребру и так далее. С необходимостью все ребра графа будут пройдены, и мы вернемся в исходную вершину.

Если граф $G(V, E)$ — простой цикл, тогда степень каждой вершины равна двум.

Доказательство. Так как граф $G(V, E)$ — замкнутый простой путь, то из каждой его вершины можно попасть в любую другую, не проходя ни через одну вершину более одного раза. Степень каждой вершины такого графа равна двум.

Покажем, что в простом цикле не может быть вершины, степень которой не равна двум.

Если какая-то вершина в графе имеет степень меньше двух, то она не принадлежит никакому простому циклу.

Если какая-то вершина имеет степень больше двух, то никакой простой цикл (по определению) не может содержать все ребра, которым принадлежит эта вершина.

Задачи, приводящие к графам

Задача 1. Лист бумаги Плюшкин (Н. В. Гоголь «Мертвые души») разрезает на три части. Некоторые из полученных листов он также разрезает на три части. Несколько новых листков он вновь разрезает на три более мелкие части и так далее. Сколько Плюшкин получает листиков бумаги, если разрезает k листов?

Решение. Будем считать листы бумаги вершинами графа. При разрезании одного листка на три части число листков увеличивается на два (появляются три новых вместо одного). Если же было разрезано k листов, то образовалось $1 + 2k$ листов.

Задача 2. Утверждают, что в одной компании из пяти человек каждый знаком с двумя другими. Возможна ли такая компания?

Решение. Каждого из этой компании будем считать вершиной графа. Двое знакомых соединим ребром. Из рассматриваемой компании нельзя выделить ни «четырехугольник», ни «треугольник», поскольку тогда из оставшихся нельзя будет составить компанию, удовлетворяющую условию. То есть схема знакомства единственная. Всякую схему, напоминающую многоугольник, принято называть циклом. Древние греки «цикл» называли «колесом».

Задача 3. Девять шахматистов проводят турнир в один круг (каждый из участников должен сыграть с каждым по одному разу). Покажите, что в любой момент найдутся двое, закончившие одинаковое число партий.

Решение. Переведем условие задачи на язык графов. Каждому из шахматистов поставим в соответствие вершину графа, соединим ребрами попарно вершины, соответствующие шахматистам, уже сыгравшим партию друг с другом. Получим граф с девятью вершинами. Степени его вершин равняются числу партий, сыгравших с соответствующими игроками. Покажем, что во всяком графе с девятью вершинами всегда найдутся хотя бы две вершины одинаковой степени.

Каждая вершина графа с девятью вершинами может иметь степень, равную $0, 1, 2, \dots, 8$. Предположим, что существует граф $G(V, E)$, все вершины которого имеют разную степень, т. е. каждое из чисел последовательности $0, 1, 2, 3, 4, 5, 6, 7, 8$ является степенью одной и только одной из его вершин. Но этого не может быть. Действительно, если в графе есть вершина A степени 0 , то в нем не найдется вершина B со степенью 8 , так

как эта вершина B должна быть соединена ребрами со всеми остальными вершинами графа, в том числе с A . Иначе говоря, в графе с девятью вершинами не могут быть одновременно вершины степени 0 и 8. Следовательно, найдутся хотя бы две вершины, степени которых равны между собой. Таким образом, доказано, что в любой момент найдутся хотя бы двое, сыгравшие одинаковое число партий.

Вывод. Во всяком графе с n вершинами, где $n \geq 2$, всегда найдутся по меньшей мере две вершины с одинаковыми степенями.

Задача 4. Девять человек проводят шахматный турнир в один круг. К некоторому моменту выясняется, что в точности двое сыграли одинаковое число партий. Нужно доказать, что либо в точности один участник еще не сыграл ни одной партии, либо в точности один сыграл все партии.

Решение. Переведем условие задачи на язык графов. Пусть вершины графа — игроки, а каждое ребро означает, что соответствующие игроки уже сыграли между собой партию. Из условия известно, что в точности две вершины имеют одинаковые степени. Требуется доказать, что в таком графе всегда найдется либо только одна изолированная вершина, либо только одна вершина степени 8.

В общем случае у графа с девятью вершинами степень каждой вершины может принимать одно из девяти значений: 0, 1, 2, ..., 7, 8. Но у такого графа степени вершин принимают только восемь разных значений, ибо ровно две вершины имеют одинаковую степень. Следовательно, обязательно либо 0, либо 8 будет значением степени одной из вершин.

Докажем, что в графах с девятью вершинами, из которых в точности две имеют одинаковую степень, не может быть двух вершин степени 0 или двух вершин степени 8.

Допустим, что все же найдется граф с девятью вершинами, в котором ровно две вершины изолированные, а все остальные имеют разные степени. Тогда, если не рассматривать эти две изолированные вершины, останется граф с семью вершинами, степени которых не совпадут. Но такого графа не существует (см. задачу 3). Значит, это предположение неверно.

Теперь допустим, что существует граф с девятью вершинами, в котором ровно две вершины имеют степень 8, а все остальные — несовпадающие степени. Тогда в дополнении данного графа ровно две вершины будут иметь степень 0, а остальные — попарно различные степени. Этого тоже не может быть (см. задачу 3), то есть и второе предположение неверно.

Следовательно, у графа с девятью вершинами, из которых в точности две имеют одинаковую степень, всегда найдется либо одна изолированная вершина, либо одна вершина степени 8.

Вернемся к задаче. Как и требовалось доказать, среди рассмотренных девяти игроков либо только один еще не сыграл ни одной партии, либо только один сыграл все партии. При решении этой задачи число 9 можно заменить любым другим натуральным числом $n > 2$.

Вывод. Если в графе с n вершинами ($n > 2$) в точности две вершины имеют одинаковую степень, то в этом графе всегда найдется либо в точности одна вершина степени 0, либо в точности одна вершина степени $n - 1$.

Лекция 2. Некоторые определения теории графов

Определения и примеры. Удаление ребер, мосты. Деревья. Перечисление деревьев.

Ключевые слова: матрица смежности, матрица инцидентий, вполне несвязный граф, пустой граф, полный граф, регулярный граф, регулярные степени r , кубические графы, трехвалентные графы, платоновы графы, объединение графов G_1, G_2 , соединение графов G_1, G_2 , мост графа, обхват графа, независимое множество ребер графа, диаметр связного графа, центр графа, дерево, лес, остовное дерево, остов графа G , каркас графа G , остовной лес, циклический ранг, циклическое число, помеченный граф, распределение меток.

Определения и примеры

Матрицей смежности графа G с множеством вершин $\{v_1, \dots, v_n\}$ (соответствующей данной нумерации вершин) называется матрица $A = (a_{ij})$ размера $n \times n$, в которой элемент a_{ij} равен числу ребер в G , соединяющих v_i и v_j . Можно получить несколько различных матриц смежности данного графа, меняя обозначения его вершин. Это приведет к изменению порядка строк и столбцов матрицы A . Но в результате всегда получится симметричная матрица из неотрицательных целых чисел, обладающая тем свойством, что сумма чисел в любой строке или столбце равна степени соответствующей вершины. Каждая петля учитывается в степени вершины один раз. Обратно, по любой заданной симметричной матрице из неотрицательных целых чисел легко построить граф, единственный с точностью до изоморфизма, для которого данная матрица является матрицей смежности. Отсюда следует, что теорию графов можно свести к изучению матриц особого типа.

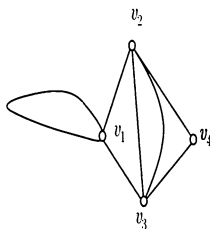
Матрицей инцидентий простого графа с множеством вершин e_1, \dots, e_m называется матрица $A = (a_{ij})$ размера $m \times n$, у которой $a_{ij} = 1$, если вершина v_j инцидентна ребру e_i , и $a_{ij} = 0$ в противном случае.

Граф, у которого множество ребер пусто, называется **вполне несвязным** или **пустым графом**. Будем обозначать вполне несвязный граф с n вершинами через N_n . Простой граф, в котором любые две вершины смежны, называется **полным графом**. Полный граф с n вершинами обычно обозначается через K_n . Граф, у которого все вершины имеют одну и ту же степень, называется **регулярным графом**. Если степень каждой вершины равна r , то граф называется **регулярным степени r** . Регулярные графы

степени 3 называются также *кубическими*, или *трехвалентными графами*. Каждый вполне несвязный граф является регулярным степени 0, а каждый полный граф K_n — регулярным степени $n - 1$. Среди регулярных графов особенно интересны *платоновы графы* — графы, образованные вершинами и ребрами пяти правильных многогранников — платоновых тел: тетраэдра, куба, октаэдра, додекаэдра и икосаэдра.

Объединение и соединение двух графов. Существует несколько способов соединения двух графов для образования нового, большего графа. Рассмотрим два из них. Пусть даны два графа $G_1 = (V(G_1), E(G_1))$, $G_2 = (V(G_2), E(G_2))$, причем множества $V(G_1)$, $V(G_2)$ не пересекаются. Тогда **объединением** $G_1 \cup G_2$ графов G_1 , G_2 называется граф с множеством вершин $V(G_1) \cup V(G_2)$ и семейством ребер $E(G_1) \cup E(G_2)$. Можно также образовать **соединение** графов G_1 , G_2 , обозначаемое $G_1 + G_2$, взяв их объединение и соединив ребрами каждую вершину графа G_1 с каждой вершиной графа G_2 .

Пример матрицы смежности. Пусть дан граф



Матрица смежности

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 2 & 1 \\ 1 & 2 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Обхватом графа называется длина его кратчайшего цикла. **Множество E ребер графа** называется **независимым**, если оно не содержит циклов, то есть никакая совокупность ребер из E не образует цикла. **Диаметром δ связного графа G** называется максимальное возможное расстояние между любыми двумя его вершинами. **Центром графа G** называется такая вершина v , что максимальное расстояние между v и любой другой вершиной является наименьшим из всех возможных. Это расстояние называется **радиусом r** . Таким образом,

$$r = \min_v (\max_w d(v, w)),$$

здесь $d(v, w)$ — расстояние между v и w .

Удаление ребер, мосты

При удалении ребра $\{a, b\}$ из графа G получается граф с теми же вершинами, что и граф G , и всеми ребрами, кроме ребра $\{a, b\}$. При удалении ребра из связного графа новый граф может оказаться как связным, так и несвязным. Ребро $\{a, b\}$ называется **мостом графа** G , если в графе, полученном после удаления из G ребра $\{a, b\}$, вершины a и b оказываются несвязными. Существует несколько признаков мостов. Известно, что признак какого-то объекта может заменить его определение, то есть по признаку можно распознать объект. Рассмотрим признаки мостов.

1. Ребро $\{a, b\}$ является мостом в том и только в том случае, если $\{a, b\}$ — единственный путь, соединяющий вершины a и b .

2. Ребро $\{a, b\}$ является мостом в том и только в том случае, если найдутся две вершины c и d , что каждый путь, соединяющий их, содержит a и b .

3. Ребро $\{a, b\}$ является мостом в том и только в том случае, если оно не принадлежит ни одному циклу.

Докажем справедливость третьего признака.

Прямая теорема. Если ребро $\{a, b\}$ не принадлежит ни одному циклу, то $\{a, b\}$ — мост.

Так как ребро $\{a, b\}$ не принадлежит ни одному циклу, при его удалении не останется пути, связывающего a и b , то есть $\{a, b\}$ является мостом.

Обратная теорема. Если ребро $\{a, b\}$ — мост, то $\{a, b\}$ не принадлежит ни одному циклу.

Если бы ребро $\{a, b\}$ принадлежало циклу, то при удалении ребра $\{a, b\}$ остался бы второй путь, связывающий a и b , то есть ребро $\{a, b\}$ не было бы мостом. Следовательно, $\{a, b\}$ не принадлежит циклу.

Деревья

Связные графы, в которых существует одна и только одна цепь, соединяющая каждую пару вершин, называются **деревьями**. Дерево можно определить и как связный граф, не содержащий циклов.

Пример. Кубок по настольному теннису разыгрывается по олимпийской системе. Встречи проводятся без «ничьих». К очередному туру допускается только победившая в предыдущем туре команда. Проигравшие команды выбывают из игры. Для завоевания кубка команда должна победить во всех турах. На участие в розыгрыше кубка поданы заявки от 16 команд.

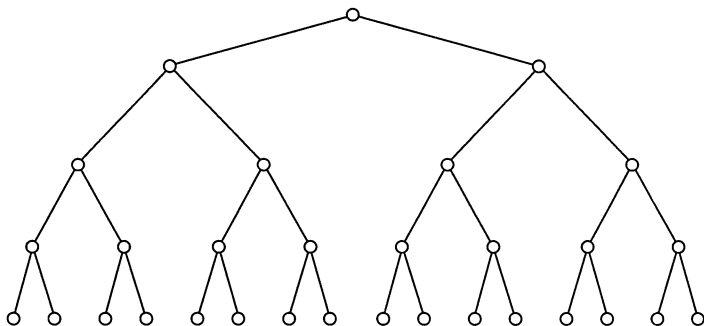


Схема проведения игр изображается графом

Вершины нижнего «яруса» дерева интерпретируем как команды, участвующие в розыгрыше кубка, вершины второго снизу яруса — как команды-победительницы в $1/16$ финала, вершины третьего яруса — как команды-победительницы в $1/8$ финала и так далее.

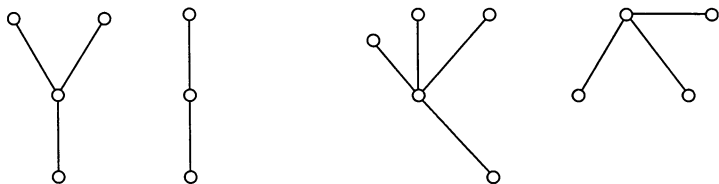
Какую информацию можно получить с помощью этого дерева?

Непосредственно с него считываются:

1. число всех участников розыгрыша кубка (число вершин нижнего «яруса»);
2. Число этапов проведения розыгрыша (число «ярусов» из вершин в дереве, не считая нижнего);
3. Число команд, участвующих в $1/8$ финала, в $1/4$ финала, в $1/2$ финала (число вершин, соответственно, в четвертом сверху ярусе, в третьем сверху ярусе, во втором сверху ярусе);
4. Число матчей, которые придется сыграть командам для выявления обладателя кубка (число вершин в графе без нижнего «яруса»). Хотя это число легко определяется и без дерева. (В каждом матче выбывает одна команда. Для того чтобы была выявлена команда-победительница, остальные должны выбыть из соревнования. Поэтому число матчей равно числу команд без одной, а именно 15).

Удобно считать, что граф, состоящий из одной изолированной вершины, тоже является деревом. Для каждой пары вершин дерева существует единственный соединяющий их путь. **Лесом** называется несвязный граф, представляющий объединение деревьев. Всякое ребро в дереве и в лесе является мостом (признак 3).

Пример. Изображен лес, состоящий из четырех компонент, каждая из которых является деревом.



Заметим, что по определению деревья и леса являются простыми графами. По многим показателям дерево представляет собой простейший нетривиальный тип графа.

Известно, что в связном графе G удаление одного ребра, принадлежащего некоторому выбранному циклу, не нарушает связности оставшегося графа. Применим эту процедуру к одному из оставшихся циклов, и так до тех пор, пока не останется ни одного цикла. В результате получим дерево, связывающее все вершины графа G , оно называется **остовным деревом** или **остовом**, или **каркасом графа G** .

В общем случае обозначим через G произвольный граф с n вершинами, m ребрами и k компонентами. Применяя описанную выше процедуру к каждой компоненте G , получим в результате граф, **называемый остовным лесом**. Число удаленных в этой процедуре ребер называется **циклическим рангом** или **циклическим числом графа G** и обозначается через $\gamma(G)$. Мы видим, что $\gamma(G) = m - n + k$ и является неотрицательным целым числом. Таким образом, циклический ранг дает меру связности графа: циклический ранг дерева равен нулю, а циклический ранг циклического графа равен единице. Удобно также определить коциклический ранг или ранг разреза графа G как число ребер в его остовном лесу. Коциклический ранг обозначается через $\chi(G)$ и равен $n - k$.

Теорема 2.1. *Дерево с n вершинами имеет $n - 1$ ребро.*

Доказательство. Для того чтобы из одного дерева G , не являющегося изолированной вершиной, получить два дерева с теми же вершинами, необходимо удалить из G одно ребро. Для образования трех деревьев необходимо удалить из G два каких-нибудь ребра. Самое большее, из дерева G с n вершинами можно получить n деревьев, каждое из которых является изолированной вершиной. Для этого необходимо удалить $n - 1$ ребро из дерева G . Итак, действительно, в дереве с n вершинами $n - 1$ ребро.

Перечисление деревьев

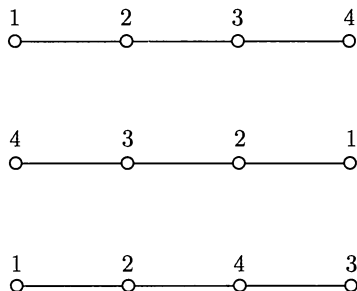
Теория перечисления графов занимается разработкой методов подсчета числа неизоморфных графов, обладающих тем или иным свой-

ством. Вероятнее всего, эта теория возникла в 70-х годах девятнадцатого столетия и связана с именем Кэли, который пытался найти число насыщенных углеводородов C_nH_{2n+2} , содержащих данное число атомов углерода. Как он обнаружил, эта задача сводится к подсчету числа деревьев, у которых степень каждой вершины равна либо четырем, либо единице. Сейчас многие задачи по перечислению графов решены.

Рассмотрим несколько определений.

Помеченный граф с n вершинами — это граф, у которого все вершины «помечены» целыми числами от 1 до n . Более точно, определим **распределение меток** в графе G с n вершинами как взаимно однозначное соответствие между множеством вершин G и множеством $\{1, \dots, n\}$; тогда **помеченным графом** называется пара $\{G, \varphi\}$, где G — граф, а φ — распределение меток в G . Числа $1, \dots, n$ будем называть **метками** графа G и обозначать вершины G через v_1, \dots, v_n . Назовем два помеченных графа $(G_1, \varphi_1), (G_2, \varphi_2)$ **изоморфными**, если существует изоморфизм между G_1 и G_2 , сохраняющий распределение меток в этих графах.

Пример. Различные распределения меток в дереве с четырьмя вершинами



Внимательное изучение рисунка позволяет заметить, что второе помеченное дерево является просто перевернутым первым, а отсюда следует, что эти два помеченных дерева изоморфны. С другой стороны, ни одно из них не изоморфно третьему помеченному дереву, достаточно посмотреть на степень вершины v_3 . Следовательно, общее число различных распределений меток в данном дереве должно равняться $1/2(4!) = 12$, поскольку «переворот» любого распределения меток не приводит к новому объекту. Аналогично, общее число различных распределений меток в дереве, изображенном на рисунке, должно равняться четырем, так как его центральная вершина может быть помечена четырьмя различными способами, каждый из которых однозначно определяет распределение меток. Отсюда следует, что число всех (неизоморфных) помеченных де-

ревью с четырьмя вершинами равно шестнадцати. Рассмотрим теорему Кэли, обобщающую этот результат на помеченные деревья с n вершинами.

Теорема 2.2 (Кэли 1889). *Существует ровно n^{n-2} различных помеченных деревьев с n вершинами.*

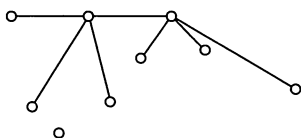
Доказательство теоремы Moon J. W. Counting labeled trees, Canadian Math. Congress, Montreal, 1970).

Лекция 3. Представления о планарном графе

Плоский граф. Гомеоморфные графы. Формула Эйлера. Триангулированный граф. Задачи

Ключевые слова: плоский граф, планарный граф, плоское представление графа, грань в плоском представлении графа, граница грани, соседние грани, бесконечная грань, перегородки, гомеоморфные графы, теорема Куратовского, элементарное стягивание, граф G , стягиваемый к графу H , максимально плоский граф, триангулированный граф, триангуляция графа.

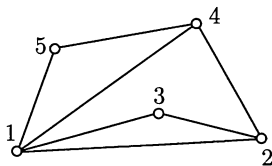
Плоский граф



Плоским графом называется граф, изображенный на плоскости так, что никакие два его ребра (или, вернее, представляющие их кривые) геометрически не пересекаются нигде, кроме инцидентной им обоим вершины.

Граф, изоморфный плоскому графу, называется *планарным*. Планарный граф можно определить еще так: граф планарен, если его можно уложить на плоскости. Рисунок графа, в котором никакие два его ребра не пересекаются, если не считать точками пересечения общие вершины, *называют плоским представлением графа*. Ясно, что плоское представление имеет только плоский граф. Обратно, у всякого плоского графа непременно найдется плоское представление. Плоские графы — это простые циклы, деревья, лес, а также граф, содержащий цикл, из вершин которого «выходят» деревья.

Пример. Примером не плоского графа может служить полный граф с пятью вершинами. Любые попытки начертить его плоское представление обернутся на неудачу.



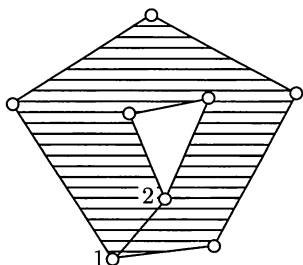
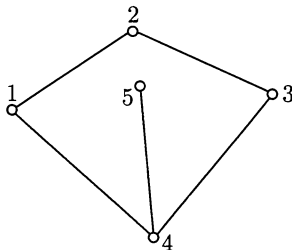
В качестве характеристики плоского представления графа вводится понятие грани. *Гранью в плоском представлении графа G* называется часть плоскости, ограниченная простым циклом и не содержащая внутри других циклов.

Пример. На рисунке показано плоское представление графа G с тремя гранями: $(1, 5, 4, 1)$, $(1, 3, 2, 4, 1)$, $(1, 2, 3, 1)$. Часть плоскости, ограниченная простым циклом

$(1, 2, 4, 1)$, гранью не является, так как содержит цикл $(1, 2, 3, 1)$. Простой цикл, ограничивающий грань, называется *границей грани*. *Две грани будем называть соседними*, если их границы имеют хотя бы одно общее ребро.

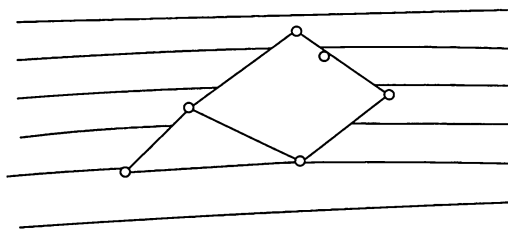
Пример. В данном графе часть плоскости, ограниченная простым циклом $(1, 2, 3, 4, 5, 1)$, является гранью, так как ребро $(4, 5)$, расположенное внутри грани, не образует цикла.

Не является гранью заштрихованная часть плоскости в данном примере, так как она содержит цикл, да к тому же эта часть плоскости не ограничена циклом. Ребро $(1, 2)$ является мостом, соединяющим циклы. Такие мосты называются *перегородками*.



В качестве грани можно рассматривать и часть плоскости, расположенную «вне» плоского представления графа. Она ограничена «изнутри» простым циклом и не содержит других циклов. Эту часть плоскости называют *бесконечной гранью*.

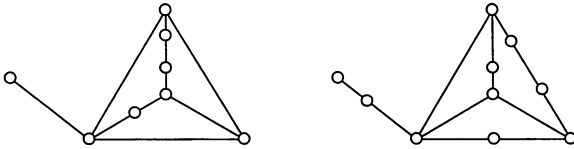
Пример.



На рисунке часть *бесконечной грани* заштрихована. Всякое плоское представление графа либо не имеет бесконечной грани, либо имеет в точности одну бесконечную грань. Как особый случай вводится бесконечная грань в плоском представлении дерева и леса. В плоском представлении дерева и леса за грань принимают всю плоскость рисунка.

Два графа гомеоморфны (или тождественны с точностью до вершин степени 2), если они оба могут быть получены из одного и того же графа «включением» в его ребра новых вершин степени 2.

Пример.



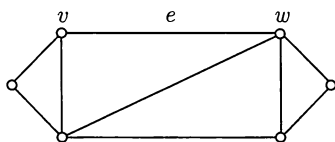
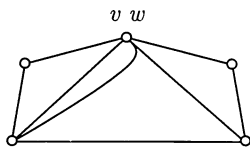
Изображенные графы гомеоморфны, и то же самое можно сказать о любых двух циклических графах. **Гомеоморфность графов** является отношением эквивалентности. Ясно, что введение термина «гомеоморфны» удобно только с технической точки зрения — включение или удаление вершин степени 2 не имеет никакого отношения к планарности. Добавление (включение) одной вершины, скажем v , в какое-нибудь ребро, например e , осуществляется следующим образом: пусть ребро e инцидентно вершинам w и x . Тогда ребро e удаляется из графа, но добавляются два новых ребра: e_1 инцидентное вершинам v и w , и e_2 , инцидентное вершинам v и x .

Введение понятия гомеоморфности графов позволяет установить следующий важный результат, известный как **теорема Куратовского** (точнее, как теорема Понтрягина–Куратовского, так как Л. С. Понтрягин доказал, но не опубликовал, эту теорему еще в 1927 году), который дает необходимое и достаточное условие планарности графа.

Гомеоморфные графы

Теорема (Куратовский 1930). *Граф планарен тогда и только тогда, когда не содержит подграфов, гомеоморфных K_5 или $K_{3,3}$.*

Поскольку доказательство теоремы Куратовского довольно длинное и сложное, здесь оно не приводится (см. Ф. Харари Теория графов. Москва «МИР». 1973). Тем не менее, воспользуемся теоремой Куратовского для получения другого критерия планарности. Рассмотрим еще два определения. **Элементарным стягиванием** называется такая процедура: берем ребро e (вместе с инцидентными ему вершинами, например, v и w) и «стягиваем» его, то есть удаляем e и отождествляем v и w . Полученная при этом вершина инцидентна тем ребрам (отличным от e), которым первоначально были инцидентны v или w .

Пример.Граф G Граф H

Граф G называется **стягиваемым к графу H** , если H можно получить из G с помощью некоторой последовательности элементарных стягиваний.

Граф планарен тогда и только тогда, когда он не содержит подграфов, стягиваемых в K_5 или в $K_{3,3}$.

Формула Эйлера

Для всякого плоского представления связного плоского графа без перегородок число вершин (V), число ребер (E) и число граней с учетом бесконечной (R) связаны соотношением: $V - E + R = 2$.

Пусть граф G — связный, плоский граф без перегородок. Определим значение алгебраической суммы $V - E + R$ для его произвольного плоского представления.

Преобразуем данный граф в дерево, содержащее все его вершины. Для этого удалим некоторые ребра графа G , разрывая поочередно все его простые циклы, причем так, чтобы граф оставался связным и без перегородок.

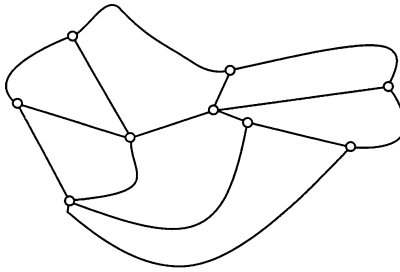
Заметим, что при таком удалении одного ребра число граней уменьшается на 1, так как при этом либо пропадет один простой цикл, либо два простых цикла преобразуются в один. Следовательно, значение разности $E - R$ при этом остается неизменным.

На рисунке ребра, которые мы удаляем, изображены кривыми. В полученном дереве обозначим число вершин — V_d число ребер — E_d , число граней — R_d . Справедливо равенство $E - R = E_d - R_d$.

В дереве одна грань, то есть $E - R = E_d - 1$. Операция удаления ребер из графа не меняет число его вершин, то есть $V = V_d$. По теореме 2.1 (см. лекцию 2) в дереве $V_d - E_d = 1$. Отсюда $V - E_d = 1$, то есть $E_d = V - 1$, а потому $E - R = V - 2$ или $V - E + R = 2$.

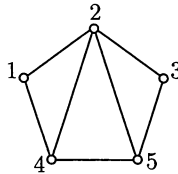
Итак, доказано, что если в плоском представлении связного графа без перегородок V вершин, E ребер и R граней, то $V - E + R = 2$. Полученная формула называется формулой Эйлера.

Пример.

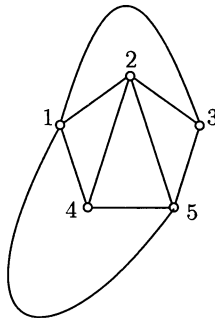


Триангулированный граф

Рассмотрим плоский граф G с пятью вершинами.



Если добавить к нему ребра $(1,3)$ и $(1,5)$, то полученный новый граф G тоже будет плоским.



К этому графу не удастся добавить ни одного ребра так, чтобы новый граф тоже был плоским.

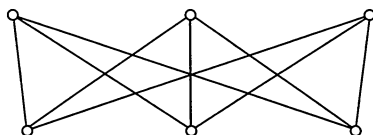
Плоский граф называется **максимально плоским**, если невозможно добавить к нему ни одного ребра так, чтобы полученный граф был плоским. Изображенный граф является максимально плоским.

Каждая грань в плоском представлении максимально плоского графа имеет 3 вершины. Поэтому максимально плоский граф называют **триангулированным**.

Операция добавления новых ребер, в результате которой в плоском представлении каждая грань имеет ровно 3 вершины, называется *триангуляцией графа*.

Задачи

Задача 1. На участке три дома и три колодца. От каждого дома к каждому колодцу ведет тропинка.



Граф G

Когда владельцы домов поссорились, они задумали проложить дороги от каждого дома к каждому колодцу так, чтобы не встречаться на пути к колодцам. Нужно показать, что их намерения не могут осуществиться.

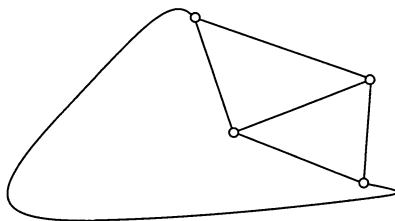
Решение. Для решения задачи достаточно доказать, что граф G , изображенный на рисунке, не плоский.

Предположим, что граф G — плоский, то есть существует его плоское представление. Граф G — связный, он не имеет ни одного моста, поэтому не имеет и перегородок. По формуле Эйлера: $V - E + R = 2$. Здесь V — число вершин, E — число ребер, R — число граней с учетом бесконечной грани. Подсчитаем число вершин и ребер: $V = 6$, $E = 9$, поэтому $R = 2 - 6 + 9 = 5$.

Теперь оценим удвоенное число ребер $2E$. Заметим, что в графе нет простых циклов длиной 3, то есть граница любой грани в плоском представлении графа G содержит не менее четырех ребер. Заметим, что каждое ребро служит границей двух граней, так как мы учитываем и бесконечную грань. При этом число $4R$ не может быть больше удвоенного числа всех ребер: $4R \leq 2E$. Если бы мы знали число ребер в границе каждой грани, то их сумма должна быть равна $2E$; но известно, что $2E = 18$, а $4R = 20$, откуда $20 \leq 18$. Полученное противоречие доказывает, что предположение было неверное, то есть граф G — не плоский. Таким образом, намерения соседей неосуществимы.

Задача 2. Каждый из четырех соседей соединил свой дом с тремя другими дорожками, которые пересекались лишь около домов.

Требуется доказать, что дом пятого соседа со всеми остальными домами соединить непересекающимися дорожками невозможно, то есть он вынужден построить мост или рыть подземный ход.

Граф G

Решение. Решение задачи сводится к доказательству того, что полный граф G с пятью вершинами не является плоским.

Предположим, что граф G плоский, то есть существует его плоское представление. Граф G — связный, он не имеет перегородок, так как не имеет ни одного моста. Для плоского представления графа G верна формула Эйлера. Подсчитаем число вершин и ребер: $V = 5$, $E = 10$, тогда $R = 2 - 5 + 10 = 7$.

Оценим удвоенное число ребер $2E$. Каждая грань ограничена не более чем тремя ребрами (граф полный). Каждое ребро принадлежит границам двух граней, поэтому число $3R$ не может быть больше числа $2E$, то есть $3R \leq 2E$. Но $2E = 20$, а $3R = 21$, то есть $20 \geq 21$. Противоречие доказывает, что предположение было неверным, то есть граф G — не плоский.

Лекция 4. Эйлеровы графы

Эйлеровы графы. Лабиринты. Геометрическая постановка задачи о лабиринтах. Решение задачи о лабиринтах

Ключевые слова: эйлеров путь, эйлеров цикл, эйлеров граф, эйлерова цепь, полуэйлеровый граф.

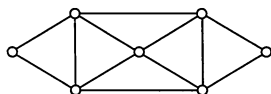
Эйлеровы графы

Эйлеровым путем в графе называется путь, содержащий все ребра графа. *Эйлеровым циклом* в графе называется цикл, содержащий все ребра графа.

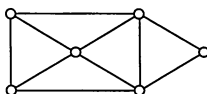
Связный граф G называется *эйлеровым*, если существует замкнутая цепь, проходящая через каждое его ребро. Такая цепь называется *эйлеровой цепью*. Отметим, что в этом определении требуется, чтобы каждое ребро проходило только один раз. Если снять ограничение на замкнутость цепи, то граф называется *полуэйлеровым*, при этом каждый эйлеров граф будет полуэйлеровым.

Примеры.

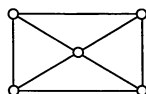
Эйлеров граф



Полуэйлеров граф

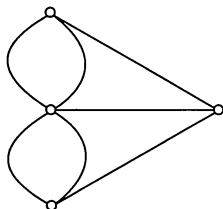


Не эйлеров граф



Заметим, что предположение о связности графа G введено только ради удобства, так как оно позволяет не рассматривать тривиальный случай графа, содержащего несколько изолированных вершин.

Задачи с эйлеровыми графами часто встречаются в книгах по занимательной математике — например, можно ли нарисовать какую-нибудь диаграмму, не отрывая карандаша от бумаги и не проходя никакую линию дважды. Название «эйлеров» возникло в связи с тем, что Эйлер первым решил знаменитую задачу о Кенигсбергских мостах, в которой нужно было узнать, имеет ли граф, изображенный на рисунке, эйлерову цепь (*не имеет!*).



Принято *всякую замкнутую линию*, если ее можно начертить, не отрывая карандаша от бумаги, проходя при этом каждый участок в точности один раз, называть *уникурсальной*. Рисунок графа, обладающего эйлеровым путем или эйлеровым циклом, является уникурсальной линией.

Теорема 4.1. *Если граф G обладает эйлеровым циклом, то он является связным, а все его вершины — четными.*

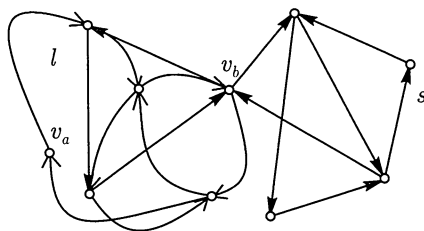
Доказательство. Связность графа следует из определения эйлерова цикла. Эйлеров цикл содержит каждое ребро и притом только один раз, поэтому, сколько раз эйлеров путь приведет конец карандаша в вершину, столько и выведет, причем уже по другому ребру. Следовательно, степень каждой вершины графа должна состоять из двух одинаковых слагаемых: одно — результат подсчета входов в вершину, другое — выходов.

Верно и обратное утверждение.

Теорема 4.2. *Если граф G связный и все его вершины четные, то он обладает эйлеровым циклом.*

Доказательство. Если начать путь из произвольной вершины графа G , то найдется цикл, содержащий все ребра графа.

Пример.



Пусть v_a — произвольная вершина графа G . Из v_a начнем путь l по одному из ребер и продолжим его, проходя каждый раз по новому ребру.

Все вершины графа имеют четные степени, поэтому если есть «выход» из v_a , то должен быть и «вход» в v_a , так же, как и для любой другой вершины. И если есть «вход» в вершину, то должен быть и «выход».

Так как число ребер конечно, этот путь должен закончиться, причем в вершине v_a , на рисунке путь l и направление его обхода показаны кривыми со стрелками.

Если путь l , замкнувшийся в v_a , проходит через все ребра графа, значит, мы получили искомым эйлеров цикл.

Если остались непройденные ребра, то должна существовать вершина v_b , принадлежащая l и ребру, не вошедшему в l .

Так как v_b — четная, то число ребер, которым принадлежит v_b и которые не вошли в путь l , тоже четно. Начнем новый путь s из v_b и используем только ребра, не принадлежащие l . Этот путь кончится в v_b . На рисунке путь s обозначен прямыми линиями со стрелками. Объединим теперь оба цикла: из v_a пройдем по пути l к v_b , затем по циклу s и, вернувшись в v_b , пройдем по оставшейся части пути обратно в v_a .

Если снова найдутся ребра, которые не вошли в путь, то найдем новые циклы. Число ребер и вершин конечно, процесс закончится.

Итак, приведен алгоритм, позволяющий отыскать эйлеров цикл, и показано, что он применим во всех случаях, допускаемых условиями теоремы.

Таким образом, замкнутую фигуру, в которой все вершины четные, можно начертить одним росчерком без повторений, начиная обводить ее с любой точки.

Если граф не обладает эйлеровым циклом, то можно поставить задачу об отыскании одного эйлерова пути или нескольких эйлеровых путей, содержащих все ребра графа.

Теорема 4.3. Если граф G обладает эйлеровым путем с концами v_a и v_b (v_a не совпадает с v_b), то граф G является связным, и v_a и v_b — единственные нечетные его вершины.

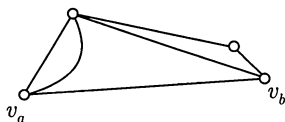
Доказательство. Связность графа следует из определения эйлерова пути. Если путь начинается в v_a , а заканчивается в другой вершине v_b , то v_a и v_b — нечетные, даже если путь неоднократно проходил через v_a и v_b . В любую другую вершину графа путь должен был привести и вывести из нее, то есть все остальные вершины должны быть четными.

Верно и обратное.

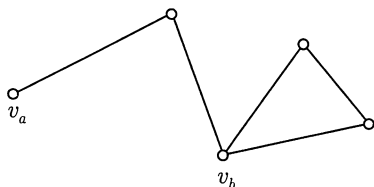
Теорема 4.4. Если граф G связный и v_a и v_b единственные нечетные вершины его, то граф G обладает эйлеровым путем с концами v_a и v_b .

Доказательство. Вершины v_a и v_b могут быть соединены ребрами в графе.

Пример.



А могут быть и не соединены.



Если v_a и v_b соединены ребром, то удалим его. Тогда все вершины станут четными. Новый граф, согласно предыдущей теореме, обладает эйлеровым циклом, началом и концом которого может служить любая вершина. Начнем эйлеров путь в вершине v_a и закончим его в вершине v_a . Добавим ребро $(v_a v_b)$ и получим эйлеров путь с началом в v_a и концом в v_b .

Если v_a и v_b не соединены ребром, то к графу добавим новое ребро $(v_a v_b)$, тогда все вершины его станут четными. Новый граф, согласно предыдущей теореме, обладает эйлеровым циклом. Начнем его из вершины v_a по ребру $(v_a v_b)$. Закончится путь тоже в вершине v_a . Если теперь удалить из полученного цикла ребро $(v_a v_b)$, то останется эйлеров путь с началом в v_b и концом в v_a или с началом в v_a и концом в v_b .

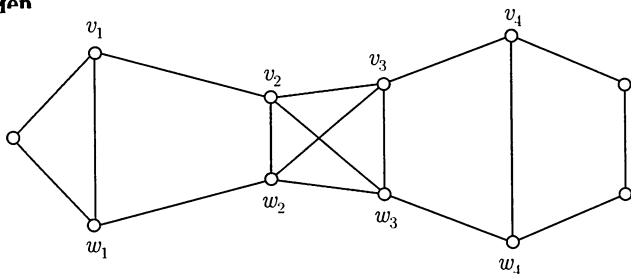
Таким образом, всякую замкнутую фигуру, имеющую в точности две нечетные вершины, можно начертить одним росчерком без повторений, начав в одной из нечетных вершин, а закончив в другой.

Теорема 4.5. Если связный граф G имеет $2k$ нечетных вершин, то найдется семейство из k путей, которые в совокупности содержат все ребра графа в точности по одному разу.

Доказательство.

Половину нечетных вершин обозначим v_1, v_2, \dots, v_k , а другую половину — w_1, w_2, \dots, w_k .

Пример



Если вершины v_i, w_i ($1 \leq i \leq k$) соединены ребром, то удалим из графа G ребро (v_i, w_i) . Если вершины v_i, w_i не соединены ребром, то добавим к G

ребро (v_i, w_i) . Все вершины нового графа будут четными, то есть в новом графе найдется эйлеров цикл. При восстановлении графа G цикл разобьется на k отдельных путей, содержащих все ребра графа.

Эйлеровым графом может быть план выставки. Это позволяет так расставить указатели маршрута, чтобы посетитель смог пройти по каждому залу в точности по одному разу.

Лабиринты

Вопрос о лабиринтах интересовал в свое время многих. В самом деле, возможно ли построить или даже начертить «безвыходный» лабиринт, то есть такой, в котором найти путь к центру и выход было бы только делом случая, а не точного математического расчета?

Разрешение этого вопроса принадлежит сравнительно позднему времени, и начало ему положено Эйлером. Результаты произведенных в этом отношении изысканий привели исследователей к заключению, что *безвыходных лабиринтов не существует*.

Решение каждого лабиринта может быть найдено, и притом сравнительно простым путем.

Способ обходить все ребра графа можно использовать, например, для поиска пути, позволяющего выбраться из лабиринта. Лабиринты, как известно, состоят из коридоров, перекрестков, тупиков (любой участок можно проходить по несколько раз), и маршруты в них могут быть представлены графами, в которых ребра соответствуют коридорам, а вершины — входам, выходам, перекресткам и тупикам.

Задача о лабиринте в общем случае сводится к построению алгоритма, позволяющего отыскать маршрут в соответствующем графе от заданной вершины v_a до заданной вершины v_b . Вершина v_a может быть входом или внутренней точкой лабиринта, из которой нужно выбраться, вершина v_b — выходом или тоже внутренней точкой, в которую необходимо попасть. Чтобы избежать бесконечного блуждания, необходимо иметь возможность запоминать пройденный путь, например, отмечать ребра графа, по которым уже прошли, и направление пути.

Если известно, что у лабиринта все «стенки» связаны друг с другом, то есть нет замкнутых маршрутов, по которым можно возвращаться в исходную точку, то такой лабиринт всегда можно обойти весь, касаясь стенки одной рукой, например правой. Если же лабиринт содержит замкнутые маршруты, то, касаясь одной рукой стенки, не всегда можно пройти все коридоры и тупики. В схемах, соответствующих таким лабиринтам, две вершины могут быть соединены и несколькими ребрами. Такие схемы называют мультиграфами.

Разработаны алгоритмы, позволяющие обойти любой лабиринт, не пользуясь его схемой.

Одно из правил обхода любого лабиринта было предложено французским математиком Тарри. Это правило о прохождении по каждому ребру графа дважды, по одному разу в каждом направлении. Согласно этому правилу, при обходе лабиринта следует, попадая в любой *перекресток* v_a *впервые* по некоторому пути s , при возвращении в этот перекресток v_a *избегать* пользоваться путем s до тех пор, пока это возможно, и лишь только в том случае *идти по пути s в обратную сторону, когда все остальные пути из v_a будут пройдены дважды.*

Геометрическая постановка задачи о лабиринтах

Аллеи, дорожки, коридоры, галереи, шахты и тому подобные лабиринты тянутся, изгибаясь во все стороны, перекрещиваются, расходятся по всевозможным направлениям, ответвляются, образуют тупики и так далее. Лабиринт — это граф. Все перекрестки обозначим вершинами, а все аллеи, дорожки, коридоры и так далее будут ребрами. Если какая-либо точка, движущаяся по ребру графа, может прийти к любой другой вершине, не покидая ребра, приняв это, докажем, что подобная движущаяся точка (например, человек) может последовательно описать все ребра без всяких скачков и перерывов и при этом по каждому ребру она пройдет ровно два раза. Другими словами, лабиринт всегда может быть разрешен.

Но еще раньше, чем приступить к этому доказательству, можно выполнить довольно интересное математическое построение, которое поможет понять все выше изложенное и будет весьма полезно для усвоения самого доказательства. На листе бумаги возьмите произвольно несколько точек и соедините их по две столько раз, сколько хотите, произвольным числом прямых или кривых линий, но так, чтобы ни одна точка системы не осталась изолированной. Итак, получается граф. Если нарисовать сеть трамваев или троллейбусов города, сеть железных дорог страны, сеть рек и каналов и так далее, прибавить к ним, границы страны — опять получается граф, или лабиринт. В данном лабиринте нужно выбрать какую-то вершину v_a и обойти все ребра два раза (пройти каждое ребро вперед и назад) и возвратиться в точку v_a .

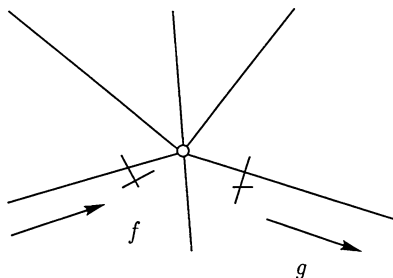
Решение задачи о лабиринтах

Правило 1. Отправляемся от выбранной вершины (первого перекрестка) и идем по любому ребру, пока не приходим или в тупик (к вершине), или к новому перекрестку (вершине).

Тогда:

1. Если окажется, что мы попали в тупик, возвращаемся назад и пройденное ребро должно быть уже отброшено, так как мы прошли его два раза (туда и обратно).
2. Если приходим к новому перекрестку, то направляемся по новому произвольному ребру, не забывая всякий раз отмечать путь, по которому прибыли, и путь, по которому отправились дальше. Как показано на рисунке.

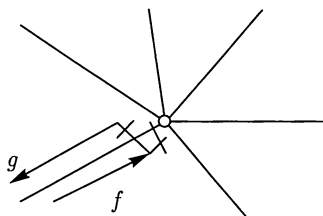
Пример.



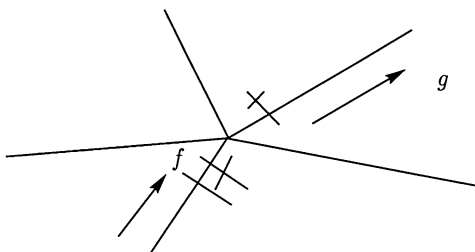
Направление движения показано стрелкой f . После прихода к пересечению путей выбирается направление, обозначенное стрелкой g . Оба пути помечаются черточкой. (Крестиками обозначаются черточки, поставленные при последнем прохождении через перекресток.)

Следуем указанному выше первому правилу всякий раз, когда приходим на такой перекресток, на котором еще не были. В конце концов мы должны прийти к перекрестку, на котором уже были, и здесь может представиться два случая. По какому пути пришли: по дороге, раз пройденной, или по новому пути.

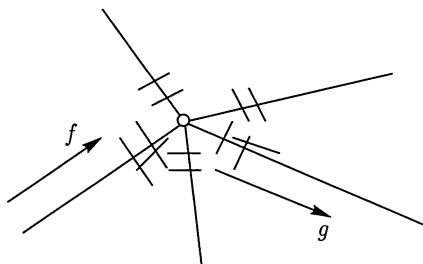
Правило 2. Прибыв на известный нам перекресток по новой дороге, мы должны сейчас же повернуть обратно, предварительно отметив этот путь двумя черточками (прибытие и обратное отправление), как показано на рисунке.



Правило 3. Если мы приходим на известный перекресток таким путем, которым уже раз прошли, то, отметив этот путь второй черточкой, отправляемся дальше путем, которым еще не проходили, если только такой путь существует. Этот случай изображен на рисунке.



Но если такого пути нет, то выбирается дорога, по которой мы прошли только один раз. Случай этот показан на рисунке.



Придерживаясь указанных правил, обойдем два раза все линии сети и придем в точку отправления. Это можно доказать, сделав предварительно такие замечания:

1. Выходя из точки отправления v_a , ставим начальный знак (поперечную черточку).
2. Прохождение через перекресток по одному из предыдущих трех правил каждый раз добавляет два знака (две поперечные черточки) на линиях, которые сходятся в этой точке.
3. В любой момент прохождения лабиринта, перед прибытием на какой-либо перекресток или после отправления из него, начальный перекресток (пункт отправления) имеет нечетное число знаков (черточек), а всякий другой перекресток имеет их четное число.

4. В любой момент, до или после прохода через перекресток, начальный перекресток имеет только один путь, обозначенный только одной черточкой. Любой другой из посещенных уже перекрестков может иметь только два пути, обозначенных одной черточкой.
5. После полного обхода лабиринта у всех перекрестков все пути должны иметь по две черточки. Это, впрочем, входит в условие задания.

Приняв во внимание все вышеизложенное, убеждаемся, что если кто-либо отправляется из начального перекрестка, скажем v_a , и прибывает в какой-либо иной перекресток v_m , то он не может встретить таких трудностей задачи, которые могли бы остановить его дальнейшее путешествие. В самом деле, в это место он приходит или новым путем, или путем, который уже один раз пройден. В первом случае прилагается первое или второе из приведенных выше правил. Во втором случае вход на перекресток v_m и остановка здесь дала бы *нечетное* число знаков около него, следовательно, за неимением нового пути надо пойти по уже пройденному один раз пути, и около перекрестка будет четное число знаков (если он не начальный) по замечанию 3.

Пусть, наконец, мы будем вынуждены закончить путь и вернуться в начальный перекресток v_a . Обозначим это ребро $(v_z v_a)$, то есть оно ведет из перекрестка v_z в начальный v_a . Этот путь должен быть тем самым, которым мы отправлялись первый раз из v_a , иначе путь можно было бы продолжать. И если теперь мы принуждены этим же путем возвратиться в начальную точку (вершину), то это значит, что из перекрестка v_z нет никакого другого пути, который бы не был уже два раза пройден. Иначе это означало бы, что забыли применить первую часть правила 3, более того, это означало бы, что в v_z есть какой-то путь $(v_v v_z)$, пройденный только один раз по замечанию 4. Итак, при последнем возвращении в v_a все пути в v_z должны быть отмечены двумя черточками. Точно так же это можно доказать для предыдущего перекрестка v_v , и для всех остальных. Другими словами, предположение доказано, и задача решена.

Лекция 5. Гамильтоновы графы

Гамильтоновы графы. Теорема Дирака

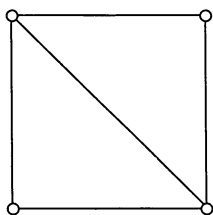
Ключевые слова: гамильтонов цикл, гамильтонов путь, гамильтонов граф, полугамильтоновый граф, теорема Дирака.

Гамильтоновы графы

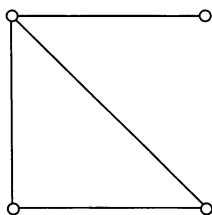
Рассмотрим проблему существования замкнутой цепи, проходящей ровно один раз через каждую вершину графа G .

Примеры:

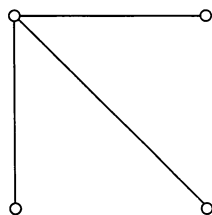
Гамильтонов граф



Полугамильтонов граф



Не гамильтонов граф



Ясно, что такая цепь должна быть циклом, исключая тривиальный случай, когда G является графом N_1 . Если такой цикл существует, то он называется **гамильтоновым циклом (путем)**, а G называется **гамильтоновым графом**. Граф, который содержит простую цепь, проходящую через каждую его вершину, называется **полугамильтоновым**. Название «гамильтонов цикл» возникло в связи с тем, что Уильям Гамильтон занимался исследованием существования таких циклов в графе, соответствующем додекаэдру. Додекаэдр — это многогранник, гранями которого служат 12 правильных пятиугольников. У него 20 вершин и 30 ребер.

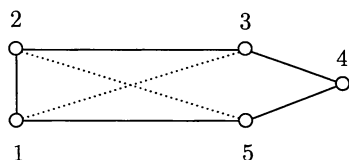
Эйлеровы и гамильтоновы пути сходны по способу задания. Первые содержат все ребра, по одному разу каждое, вторые — все вершины, по одному разу каждую. Но, несмотря на внешнее сходство, задачи их поиска резко отличаются по степени трудности. Для решения вопроса о наличии эйлерова цикла в графе достаточно выяснить, все ли его вершины четны. *Критерий же существования гамильтонова цикла в произвольном графе еще не найден.* Решение этой проблемы имеет практическую ценность,

так как к игре Гамильтона близка известная задача о коммивояжере, который должен объехать несколько пунктов и вернуться обратно. Он обязан побывать в каждом пункте в точности по одному разу и заинтересован в том, чтобы затратить на поездку как можно меньше времени. А для этого требуется определить все варианты посещения городов и подсчитать в каждом случае затрату времени. По своей математической постановке игра Гамильтона близка к задаче о порядке переналадки станков, задаче о подводке электроэнергии к рабочим местам и т. д. Подробнее об этом рассказывается, например, в книге В. И. Мудрова «Задача о коммивояжере» М., Знание, 1969.

Рассмотрим несколько достаточных условий существования гамильтоновых циклов в графе.

Во-первых, *всякий полный граф является гамильтоновым*. Действительно, он содержит такой простой цикл, которому принадлежат все вершины данного графа. Во-вторых, если граф, помимо простого цикла, проходящего через все его вершины, содержит и другие ребра, то он также является гамильтоновым.

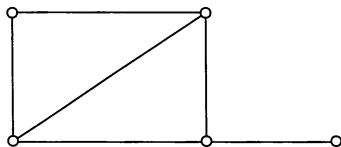
Примеры.



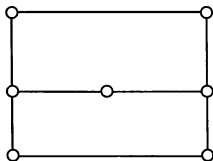
Простой (гамильтонов) цикл выделен сплошной линией (1, 2), (2, 3), (3, 4), (4, 5), (5, 1). Заметим, что если граф имеет один гамильтонов цикл, то он может иметь и другие гамильтоновы циклы.

Если гамильтонов граф объединить с еще одной вершиной ребром так, что образуется висячая вершина, то такой граф гамильтоновым не является, поскольку не содержит простого цикла, проходящего через все вершины графа.

Пример.



Не является гамильтоновым и граф, представляющий собой простой цикл с «перекладной» на которой расположены одна или несколько вершин.

Пример.

Такие графы называют «тэта-графами», поскольку они похожи на греческую букву θ («тета»). По рисунку видно, что в таком графе не удастся выделить простой цикл, содержащий все вершины.

Выведем еще два достаточных признака гамильтоновых графов.

Рассмотрим граф G с $m \geq 3$ вершинами. Пронумеруем их произвольным образом и выпишем их последовательность:

$$v_0, v_1, v_2, \dots, v_{i-1}, v_i, v_{i+1}, \dots, v_{m-1}. \quad (5.1)$$

При этом может случиться, что некоторые две соседние вершины, например, v_k и v_{k+1} , не связаны ребром. Будем говорить, что в данной последовательности имеется «разрыв» между вершинами v_k и v_{k+1} .

Очевидно, в последовательности $v_1, v_2, \dots, v_{i-1}, v_i$ не возникнут другие разрывы, если ее записать в обратном порядке, а именно: $v_i, v_{i-1}, \dots, v_2, v_1$.

Пусть для определенности разрыв в последовательности (5.1) имеет место между вершинами v_0 и v_1 . Положим теперь, что v_i — вершина графа G , связанная ребром с v_0 . Число таких вершин v_i равно $\rho(v_0)$.

Пытаясь ликвидировать разрыв в последовательности (5.1) между v_0 и v_1 , запишем ее в измененном порядке:

$$v_0, v_i, v_{i-1}, \dots, v_2, v_1, v_{i+1}, \dots, v_{m-1} \quad (5.2)$$

При этом число разрывов уменьшится на единицу в том случае, если между вершинами v_1 и v_{i+1} не возникнет новый разрыв.

Вершину v_i среди $m-1$ вершин, не совпадающих с v_0 , можно всегда найти, так, чтобы между v_1 и v_{i+1} не возник новый разрыв, если справедливо неравенство

$$\rho(v_0) \geq (m-1) - \rho(v_1)$$

(справа в этом неравенстве читаем число разрывов, которые могут произойти при всевозможных перестановках последовательности (5.1)).

Но вершины v_0 и v_1 были выбраны произвольно; можно было рассмотреть разрыв между другими соседними вершинами v_k и v_{k+1} в последовательности (5.1) можно было даже выбрать вершины v_u и v_v гра-

фа G , не стоящие рядом в последовательности (5.1). Лишь бы соблюдалось неравенство

$$\rho(v_u) \geq (m-1) - \rho(v_v) \quad (5.3)$$

Заметим, что неравенство (5.3) симметрично относительно v_u и v_v . Его можно записать в виде

$$\rho(v_u) + \rho(v_v) \geq m-1. \quad (5.4)$$

И тогда в последовательности (5.1) удастся ликвидировать все разрывы. А это означает, что в графе G найдется гамильтонов путь.

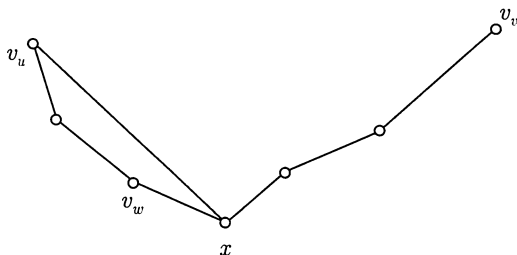
Покажем, что если для любой пары вершин v_u и v_v графа G с m вершинами справедливо неравенство

$$\rho(v_u) + \rho(v_v) \geq m, \quad (5.5)$$

то граф G обладает гамильтоновым циклом. Это один из достаточных признаков того, что данный граф является гамильтоновым.

Рассмотрим гамильтонов путь, связывающий вершины v_u и v_v графа G .

Пример.



Пусть x — одна из вершин графа G , связанная ребром с вершиной v_u . Тогда в силу неравенства (5.5), хотя бы для одной из таких вершин x найдется в гамильтоновом пути смежная с ней вершина v_w , такая, которая связана ребром с v_v .

Добавляя к гамильтонову пути ребра (v_u, x) , (v_w, v_v) и выбрасывая из него ребро (v_w, x) , получаем гамильтонов цикл, что и требовалось.

Теперь, как следствие, получаем еще один достаточный признак того, что данный граф является гамильтоновым.

Формулируется этот признак так:

Граф G с m вершинами имеет гамильтонов цикл, если для произвольной его вершины

$$\begin{aligned} v_i \quad (i = 0, 1, \dots, m-1) \\ \rho(v_i) \geq \frac{m}{2}. \end{aligned} \quad (5.6)$$

Хотя этот признак проще, чем предыдущий (при его использовании приходится меньше считать), он позволяет распознать более узкий класс гамильтоновых графов.

Проведенное доказательство справедливости достаточных признаков гамильтоновых графов было косвенным — мы не строили для данного произвольного графа, удовлетворяющего неравенству (5.5) или неравенству (5.6), гамильтоновых циклов.

Теорема Дирака

Поиск необходимого и достаточного условия для того, чтобы граф был гамильтоновым, стал одной из главных нерешенных задач теории графов! О гамильтоновых графах, в сущности, известно очень мало. Большинство известных теорем имеет вид: «если граф G имеет достаточное число ребер, то граф G является гамильтоновым графом». Вероятно, самой знаменитой из этих теорем является следующая теорема, принадлежащая Г. Э. Дираку и потому известная как *теорема Дирака*.

Теорема (Дирак 1952). *Если в простом графе с $n(\geq 3)$ вершинами $\rho(v) \geq n/2$ для любой вершины v , то граф G является гамильтоновым.*

Замечание. *Существует несколько доказательств этой широко известной теоремы, здесь мы приводим доказательство Д. Дж. Ньюмана.*

Доказательство. Добавим к нашему графу k новых вершин, соединяя каждую из них с каждой вершиной из G . Будем предполагать, что k — наименьшее число вершин, необходимых для того, чтобы полученный граф G' стал гамильтоновым. Затем, считая, что $k > 0$, придем к противоречию.

Пусть $v \rightarrow p \rightarrow w \rightarrow \dots \rightarrow v$ гамильтонов цикл в графе G' , где v, w — вершины из G , а p — одна из новых вершин. Тогда w не является смежной с v , так как в противном случае мы могли бы не использовать вершину p , что противоречит минимальности k . Более того, вершина, скажем, w' , смежная вершине w , не может непосредственно следовать за вершиной v' , смежной вершине v , потому что тогда мы могли бы заменить $v \rightarrow p \rightarrow w \rightarrow \dots \rightarrow v' \rightarrow w' \rightarrow \dots \rightarrow v$ на $v \rightarrow v' \rightarrow \dots \rightarrow w \rightarrow w' \rightarrow \dots \rightarrow v$, перевернув часть цикла, заключенную между w и v' . Отсюда следует, что число вершин графа G' , не являющихся смежными с w , не меньше числа вершин, смежных с v (то есть равно, по меньшей мере, $n/2 + k$); с другой стороны, очевидно, что число вершин графа G' , смежных с w , тоже равно, по меньшей мере, $n/2 + k$. А так как ни одна вершина графа G' не может быть одновременно смежной и не смежной вершине w , то общее число вершин графа G' , равное $n + k$, не меньше, чем $n + 2k$. Это и есть искомое противоречие.

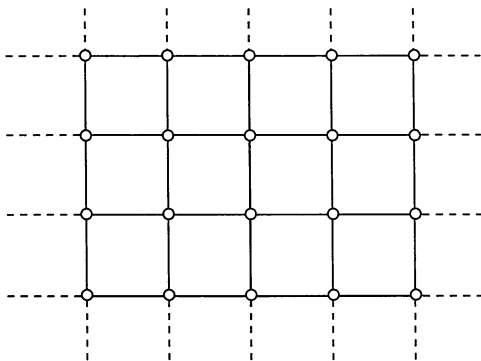
Лекция 6. Бесконечные графы

Бесконечные графы. Краткий обзор свойств бесконечных эйлеровых графов

Ключевые слова: бесконечный граф, вершины, ребра, счетный граф, степень вершины, локально конечный граф, локально счетный бесконечный граф, конечный маршрут, бесконечный в одну сторону маршрут, начальная вершина, бесконечный в обе стороны маршрут, эйлеров граф, эйлерова цепь, полуэйлеров граф.

Бесконечные графы

В данной лекции мы обобщим некоторые определения на случай бесконечных графов. **Бесконечным графом** называется пара $(V(G), E(G))$, где $V(G)$ — бесконечное множество элементов, называемое **вершинами**, а $E(G)$ — бесконечное семейство неупорядоченных пар элементов из $V(G)$, называемых **ребрами**.



Если оба множества $V(G)$ и $E(G)$ счетны, то G называется **счетным графом**. Заметим, что наши определения исключают те случаи, когда $V(G)$ бесконечно, а $E(G)$ конечно. Такие объекты являются всего лишь конечными графами с бесконечным множеством изолированных вершин. Когда $E(G)$ бесконечно, а $V(G)$ конечно, такие объекты являются конечными графами с бесконечным числом петель или кратных ребер.

Некоторые определения таких понятий как «смежный», «инцидентный», «изоморфный», «подграф», «объединение», «связный», «компо-

нента» переносятся на бесконечные графы. **Степенью вершины v бесконечного графа** называется мощность множества ребер, инцидентных v . Степень вершины может быть конечной или бесконечной. Бесконечный граф, все вершины которого имеют конечные степени, называется **локально конечным**. Хорошо известным примером такого графа является бесконечная квадратная решетка, часть которой изображена на рисунке. **Локально счетный бесконечный граф** — это граф, все вершины которого имеют счетную степень. Под счетным множеством здесь и в дальнейшем понимается бесконечное множество, допускающее взаимно однозначное отображение на множество натуральных чисел.

Теорема. *Каждый связный локально счетный бесконечный граф является счетным.*

Доказательство.

Пусть v — произвольная вершина такого бесконечного графа, и пусть v_1 — множество вершин, смежных v , v_2 — множество всех вершин, смежных вершинам из v_1 , и так далее. По условию теоремы v_1 — счетно и, следовательно, множества v_2, v_3, \dots тоже счетны. Здесь используется тот факт, что объединение не более чем счетного множества счетных множеств счетно. Следовательно, $\{v\}, v_1, v_2, \dots$ — последовательность множеств, объединение которых счетно. Кроме того, эта последовательность содержит каждую вершину бесконечного графа в силу его связности. Отсюда и следует нужный результат.

Следствие. *Каждый связный локально конечный бесконечный граф является счетным.*

Помимо этого, на бесконечный граф G можно перенести понятие маршрута, причем тремя различными способами:

1. **Конечный маршрут** в G определяется так. Маршрутом в данном графе G называется конечная последовательность ребер вида $\{v_0, v_1\}, \{v_1, v_2\}, \dots, \{v_{m-1}, v_m\}$. Маршрут можно обозначить и так $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_m$.
2. **Бесконечным в одну сторону маршрутом** в G с начальной вершиной v_0 называется бесконечная последовательность ребер вида $\{v_0, v_1\}, \{v_1, v_2\}, \dots$
3. **Бесконечным в обе стороны маршрутом** в графе G называется бесконечная последовательность ребер вида $\dots, \{v_{-2}, v_{-1}\}, \{v_{-1}, v_0\}, \{v_0, v_1\}, \{v_1, v_2\}, \dots$

Бесконечные в одну сторону и в обе стороны цепи и простые цепи определяются очевидным образом, так же, как и понятия длины цепи и расстояния между вершинами. Бесконечные простые цепи не так уж трудно обнаружить.

Теорема 6.1 (Кениг 1936). Пусть G — связный локально конечный бесконечный граф, тогда для любой вершины v существует бесконечная в одну сторону простая цепь с начальной вершиной v .

Доказательство.

Если z — произвольная вершина графа G , отличная от v , то существует нетривиальная простая цепь от v до z , отсюда следует, что в G имеется бесконечно много простых цепей с начальной вершиной v . Поскольку степень v конечна, то бесконечное множество таких простых цепей должно начинаться с одного и того же ребра. Если таким ребром является $\{v, v_1\}$, то, повторяя эту процедуру для вершины v_1 , получим новую вершину v_2 и соответствующее ей ребро $\{v_1, v_2\}$. Продолжая таким образом, получим бесконечную в одну сторону простую цепь $v \rightarrow v_1 \rightarrow v_2 \dots$

Важное значение леммы Кенига состоит в том, что она позволяет получить результаты о бесконечных графах из соответствующих результатов для конечных графов. Типичным примером является следующая теорема.

Теорема 6.2. Пусть G — счетный граф, каждый конечный подграф которого планарен, тогда и G планарен.

Доказательство. Так как G — счетный граф, его вершины можно занумеровать в последовательность v_1, v_2, v_3, \dots . Исходя из нее, построим строго возрастающую последовательность $G_1 \subset G_2 \subset G_3 \dots$ подграфов графа G , выбирая в качестве G_k подграф с вершинами v_1, \dots, v_k и ребрами графа G , соединяющими только эти вершины между собой. Далее, примем на веру тот факт, что графы G_i могут быть уложены на плоскости конечным числом, скажем $m(i)$, топологически различными способами, и построим еще один бесконечный граф H . Его вершины w_{ij} ($i \geq 1, 1 \leq j \leq m(i)$) пусть соответствуют различным укладкам графов $\{G_i\}$, а его ребра соединяют те из вершин w_{ij} и w_{kl} , для которых $k = i + 1$ и плоская укладка, соответствующей w_{ij} . Мы видим, что граф H связан и локально конечен, поэтому, как следует из леммы Кенига, он содержит бесконечную в одну сторону простую цепь. А так как граф G является счетным, то эта бесконечная простая цепь и дает требуемую плоскую укладку графа G .

Стоит подчеркнуть, что если принять дальнейшие аксиомы теории множеств, в частности, аксиому выбора для несчетных множеств, то многие результаты можно перенести и на такие бесконечные графы, которые необязательно являются счетными.

Краткий обзор свойств бесконечных эйлеровых графов

Связный бесконечный граф G называется эйлеровым, если в нем существует бесконечная в обе стороны цепь, содержащая каждое ребро графа G . Такая бесконечная цепь называется (двусторонней) *эйлеровой цепью*. Назовем граф G *полуэйлеровым*, если в нем существует бесконечная (в одну или в обе стороны) цепь, содержащая каждое ребро графа G .

Теорема 6.3. Пусть G — связный счетный граф, являющийся эйлеровым. Тогда

1. в графе G нет вершин нечетной степени;
2. для каждого конечного подграфа H графа G бесконечный граф \bar{H} (полученный путем удаления из G ребер графа H) имеет не более двух бесконечных связных компонент;
3. если, кроме того, степень любой вершины из H четна, то \bar{H} имеет ровно одну бесконечную связную компоненту.

Доказательство.

1. Предположим, что P — эйлерова цепь в графе G . Тогда при всяком прохождении цепи P через любую из вершин графа степень этой вершины увеличивается на два. А так как каждое ребро встречается в P ровно один раз, то каждая вершина должна иметь четную степень. Получим, что степень любой вершины из G должна быть либо четной, либо бесконечной.

2. Разобьем цепь P на три подцепи P_- , P_0 , P_+ так, что P_0 — конечная цепь, содержащая все ребра графа H (и, быть может, другие ребра), а P_- , P_+ — две бесконечные в одну сторону цепи. Тогда бесконечный граф K , образованный ребрами цепей P_- , P_+ (а также инцидентными вершинами), имеет не более двух бесконечных компонент. Так как \bar{H} получается из K присоединением лишь конечного множества ребер, то отсюда и следует нужный результат.

3. Пусть v, w — начальная и конечная вершины цепи P_0 . Покажем, что v, w связаны в \bar{H} . Если $v = w$, то это очевидно. Если $v \neq w$, то применяя следствие (*связный граф является полуэйлеровым тогда и только тогда, когда в нем не более двух вершин имеют нечетные степени*) к графу, полученному из P_0 путем удаления ребер графа H (предполагая, что в этом графе ровно две вершины, а именно v, w , имеют нечетные степени), получим требуемый результат.

Можно получить соответствующие необходимые условия для полуэйлеровых бесконечных графов.

Теорема 6.4. Пусть G — связный счетный граф, являющийся полуэйлеровым, но не эйлеровым. Тогда

1. G содержит либо не более одной вершины нечетной степени, либо не менее одной вершины бесконечной степени;
2. Для каждого конечного подграфа H графа G бесконечный граф \overline{H} (описанный ранее) содержит ровно одну бесконечную компоненту.

Оказывается, условие двух предыдущих теорем является не только необходимым, но и достаточным. Этот результат сформулируем в виде следующей теоремы, доказательство которой выходит за рамки данных лекций, его можно найти у Оре (Оре О., Теория графов, «Наука», М., 1968).

Теорема 6.5. Пусть G — связный счетный граф. G является эйлеровым графом тогда и только тогда, когда выполнены условия

1. в графе G нет вершин нечетной степени;
2. для каждого конечного подграфа H графа G бесконечный граф \overline{H} (полученный путем удаления из G ребер графа H) имеет не более двух бесконечных связных компонент;
3. если, кроме того, степень любой вершины из H четна, то \overline{H} имеет ровно одну бесконечную связную компоненту.

Более того, G является полуэйлеровым тогда и только тогда, когда выполнены либо эти условия, либо условия:

1. G содержит либо не более одной вершины нечетной степени, либо не менее одной вершины бесконечной степени;
 2. для каждого конечного подграфа H графа G бесконечный граф \overline{H} (описанный ранее) содержит ровно одну бесконечную компоненту.
-
1. в графе G нет вершин нечетной степени;
 2. для каждого конечного подграфа H графа G бесконечный граф \overline{H} (полученный удалением из G ребер графа H) имеет не более двух бесконечных связных компонент;
 3. если, кроме того, степень любой вершины из H четна, то \overline{H} имеет ровно одну бесконечную связную компоненту.

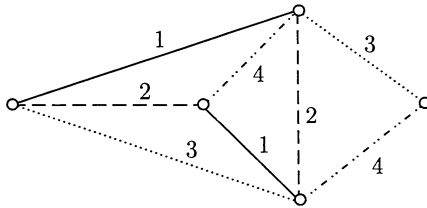
Лекция 7. Графы с цветными ребрами

Реберная раскраска. Задачи на графы с цветными ребрами и вытекающие из них свойства. Задача о несцепленных треугольниках с одноцветными сторонами

Ключевые слова: граф G реберно k — раскрашиваемый, хроматический класс, хроматический индекс, реберно-хроматическое число, два сцепленные треугольника.

Реберная раскраска

*Граф G называется **реберно k -раскрашиваемым**, если его ребра можно раскрасить k красками таким образом, что никакие два смежных ребра не окажутся одного цвета. Если граф G реберно k -раскрашиваем, но не является реберно $(k - 1)$ -раскрашиваемым, то k называется **хроматическим классом** или **хроматическим индексом**, или **реберно-хроматическим числом** графа G . При этом используется запись $\chi_e(G) = k$. На рисунке изображен граф G , для которого $\chi_e(G) = 4$.*



Ясно, что если наибольшая из степеней вершин графа G равна ρ , то $\chi_e(G) \geq \rho$. Следующий результат, известный как теорема Визинга, дает точные оценки для хроматического класса графа G . Доказательство этой теоремы можно найти у Ore (Ore O. The four-color problem, Academic Press, New York, 1967.)

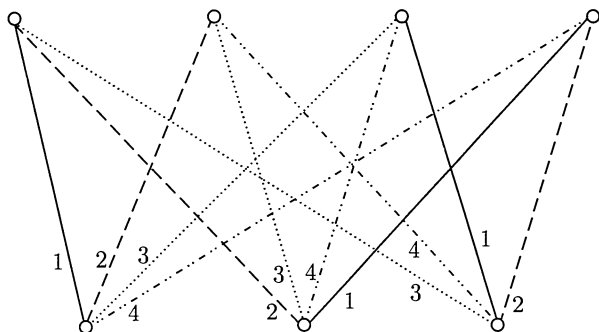
Теорема 7.1 (Визинг 1964). Пусть в графе G , не имеющем петель, наибольшая из степеней вершин равна ρ ; тогда $\rho \leq \chi_e(G) \leq \rho + 1$.

Задача, состоящая в выяснении того, какие графы имеют хроматический класс ρ , а какие $\rho + 1$, не решена. Однако в некоторых частных случаях соответствующие результаты находятся легко. Например, $\chi_e(C_n) = 2$ или 3 в зависимости от того, четно n или нечетно, а $\chi_e(W_n) = n - 1$,

при $n \geq 4$. Хроматические классы полных графов и полных двудольных графов вычисляются тоже просто.

Теорема 7.2. $\chi_e(K_{m,n}) = \rho = \max(m, n)$.

Доказательство. Без потери общности можно считать, что $m \geq n$ и что граф $K_{m,n}$ изображен так:



n вершин расположены на горизонтальной линии под m вершинами. Тогда искомая реберная раскраска получается последовательным окрашиванием ребер, инцидентных этим n вершинам, с использованием следующих групп красок:

$$\{1, 2, \dots, m\}; \{2, 3, \dots, m, 1\}; \dots; \{n, \dots, m, 1, \dots, n-1\};$$

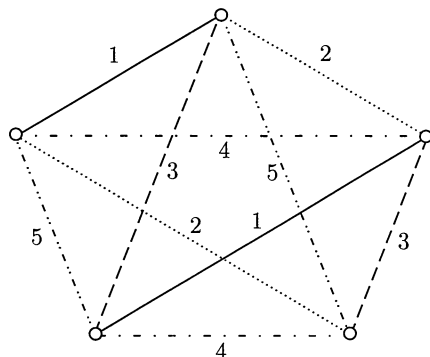
при этом краски из каждой группы располагаются по часовой стрелке, вокруг соответствующей вершины.

Теорема 7.3. $\chi_e(K_n) = n$, если n нечетно ($n \neq 1$), и $\chi_e(K_n) = n - 1$, если n четно.

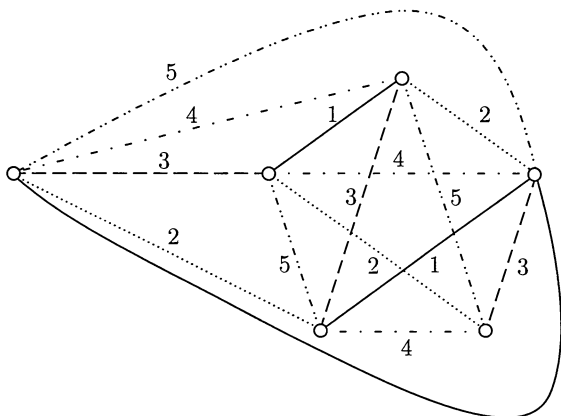
Доказательство.

В случае нечетного n расположим вершины графа K_n в виде правильного n -угольника. Тогда его ребра можно раскрасить следующим образом: сначала окрашиваем каждую сторону n -угольника в свой цвет, а затем каждое из оставшихся ребер, диагонали n -угольника, окрашиваем в тот же цвет, что и параллельная ему сторона. То, что граф K_n не является реберно $(n - 1)$ -раскрашиваемым, сразу же следует из того, что максимально возможное число ребер одного цвета равно $(n - 1)/2$.

В случае четного $n (\geq 4)$ граф K_n можно рассматривать как соединение полного $(n - 1)$ — графа K_{n-1} и отдельной вершины. Если в K_{n-1} окрасить ребра описанным выше способом, то для каждой вершины останется один неиспользованный цвет, причем все эти неиспользованные



цвета будут различными. Таким образом, чтобы получить реберную раскраску K_n , достаточно окрасить оставшиеся ребра в соответствующие «неиспользованные» цвета.



Задачи на графы с цветными ребрами и вытекающие из них свойства

Рассматриваем графы, соответствующие таким ситуациям, в которых одни пары элементов множества находятся между собой в одном отношении, другие пары этого множества — в другом отношении, третьи — в третьем, но каждая пара — в одном отношении. Например, среди участников шахматного турнира к какому-то моменту могут быть такие, которые уже сыграли партию друг с другом, и такие, которые не сыграли. Среди множества стран есть страны, установившие между собой диплома-

тические связи, и страны, между которыми не установлены дипломатические связи. Для удобства на рисунках графов ребра, соответствующие одному отношению, окрашивают в один цвет, а ребра, соответствующие другому отношению, — во второй цвет, в третий цвет и так далее. Так как мы не можем выполнить рисунок в разных цветах, то присваиваем ребрам номера. Такие графы называются графами с цветными ребрами.

Свойства полных графов с цветными ребрами

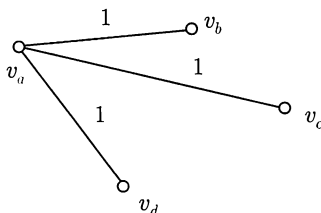
Задача 1. Шесть человек участвуют в шахматном турнире, который проводится в один круг, то есть каждый шахматист встречается со всеми участниками по одному разу. Нужно доказать, что среди них всегда найдутся три участника турнира, которые провели уже все встречи между собой или еще не сыграли друг с другом ни одной партии.

Решение. Любые два участника турнира находятся между собой в одном из двух отношений: они либо уже сыграли между собой, либо еще не сыграли.

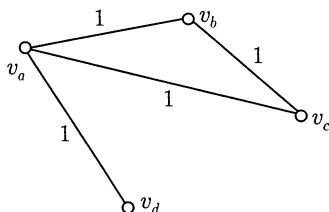
Каждому участнику поставим в соответствие вершину графа. Соединим вершины попарно ребрами двух цветов. Пусть ребро красного цвета (обозначенное цифрой 1) означает, что двое уже сыграли между собой, а синего (пронумерованное цифрой 2), что не сыграли. Получим полный граф с шестью вершинами и ребрами двух цветов.

Теперь для решения задачи достаточно доказать, что в таком графе обязательно найдется «треугольник» с одноцветными сторонами.

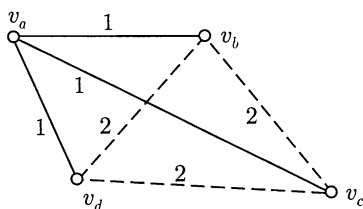
Каждая вершина полученного графа принадлежит пяти ребрам. Скольким шахматистам одного цвета может принадлежать произвольная вершина такого графа? Пять принадлежащих одной вершине ребер могут быть окрашены без учета порядка следующим образом: 22222, 12222, 11222, 11122, 11112, 11111. То есть каждая вершина принадлежит, по меньшей мере, трем шахматистам одного цвета. Пусть, например, вершина v_a принадлежит трем ребрам красного цвета:



Какого цвета ребра могут соединять вершины v_b , v_c и v_d ? Если хотя бы одно из них окажется красным, как на рисунке,



то получится треугольник с красными сторонами. Если же все эти ребра синие, как на рисунке,



то они вместе образуют «треугольник» с синими сторонами.

Задача решена. Рассмотрены все возможности. В каждом случае нашлись три шахматиста, или все сыгравшие между собой по одной партии, или не сыгравшие между собой ни одной партии.

Кроме того, при ее решении доказаны два свойства таких графов.

Свойство 1. Любая вершина полного графа с шестью или более вершинами и ребрами двух цветов принадлежит, по меньшей мере, трем ребрам одного цвета.

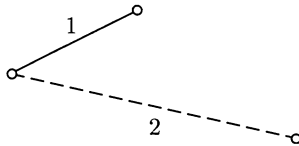
Свойство 2. В любом полном графе с шестью или более вершинами и ребрами двух цветов найдется, по меньшей мере, один треугольник с одноцветными сторонами.

Задача 2. На географической карте выбраны пять городов. Известно, что среди них из любых трех найдутся два, соединенные авиалиниями, и два — несоединенные. Требуется доказать, что:

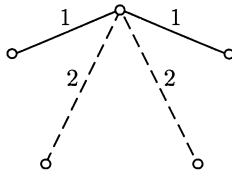
1. Каждый город соединен авиалиниями непосредственно с двумя и только с двумя другими городами.
2. Вылетев из любого города, можно облететь остальные, побывав в каждом по одному разу, и вернуться назад.

Решение. Рассматривается множество объектов — городов и два отношения, заданные для элементов этого множества. Каждые два города находятся в одном из двух отношений — они либо соединены меж-

ду собой авиалиниями, либо не соединены. Пусть вершины графа соответствуют городам: красное ребро (пронумеровано 1) соответствует наличию авиалиний, синее ребро (пронумеровано 2) соответствует отсутствию авиалиний. По условию среди трех ребер, соединяющих любые три вершины, одно — красное, второе — синее,



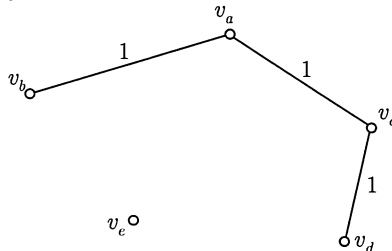
а это означает, что в графе нет ни одного треугольника с одноцветными сторонами. Тогда из решения предыдущей задачи следует, что каждая вершина непременно принадлежит двум красным ребрам и двум синим,



поскольку в противном случае образовался бы треугольник с одноцветными сторонами. А это и означает, что каждый город соединен авиалиниями с двумя и только с двумя городами.

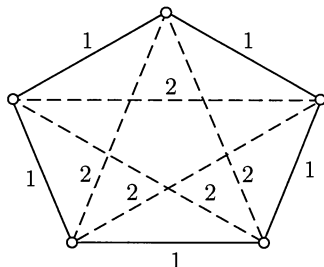
Остается показать, что в графе найдется «пятиугольник», все ребра которого — красные.

Выберем одну из вершин, например v_a , а красными будут, скажем, ребра $\{v_a, v_b\}, \{v_a, v_c\}$



Ребро $\{v_b, v_c\}$ не может быть красным, следовательно, красным является одно из ребер: либо $\{v_c, v_d\}$, либо $\{v_c, v_e\}$. Пусть красное $\{v_c, v_d\}$. Если теперь соединить красным ребром вершины v_d и v_b , то вершина v_e должна быть соединена красными ребрами с вершинами, которые принадлежат уже двум красным ребрам. По условию это невозможно. Оста-

ется соединить красными ребрами вершины v_d и v_e , v_b и v_e . Остальные ребра должны быть синими.

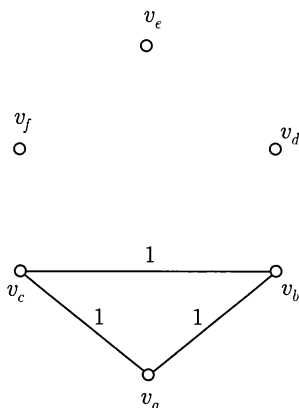


Итак, мы получили еще одно свойство.

Свойство 3. Если в полном графе с пятью вершинами и ребрами двух цветов не найдется треугольника с одноцветными сторонами, то граф можно изобразить в виде «пятиугольника» с красными сторонами и синими диагоналями.

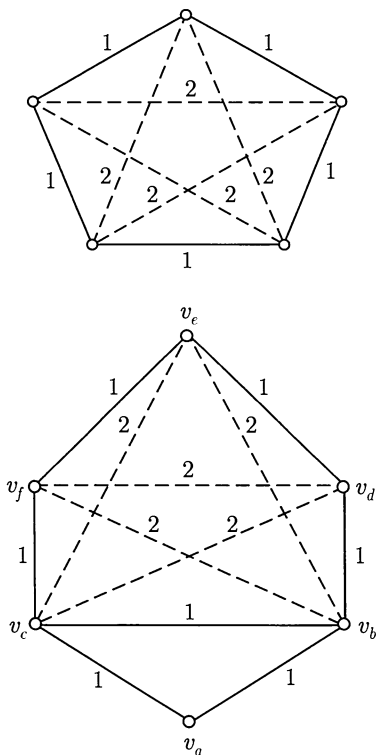
В формулировке свойства 3 можно заменить слово «красный» на «синий» и одновременно слово «синий» на «красный», то есть речь пойдет о пятиугольнике с синими сторонами и красными диагоналями. Это понятно, поскольку для пятиугольника и только для него характерно, что его диагонали образуют также пятиугольник.

Задача 3. В течение дня двое из шести телефонных абонентов могут поговорить друг с другом по телефону, а могут и не поговорить. Докажем, что всегда можно указать две тройки абонентов, в каждой из которых все переговорили друг с другом или все не переговорили.



Решение. Пусть у полного графа с шестью вершинами красные ребра соответствуют парам абонентов, которые говорили друг с другом по теле-

фону, синие — тем, кто не говорил. Тогда в графе найдется хотя бы один треугольник, $v_a v_b v_c$, с одноцветными сторонами.



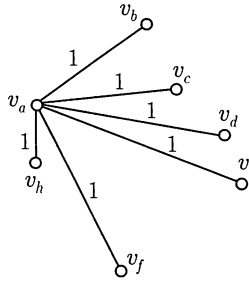
Остается показать, что обязательно найдется еще и второй такой треугольник.

Временно исключим из рассмотрения одну из его вершин, скажем v_a , вместе с ребрами, принадлежащим ей.

Найдется ли в оставшемся графе с пятью вершинами треугольник с одноцветными сторонами? Если найдется, то он содержится и в исходном графе.

В противном случае получается пятиугольник с красными сторонами и синими диагоналями. Теперь восстановим шестую вершину v_a с ее ребрами.

Если ребро $\{v_a, v_d\}$ или ребро $\{v_a, v_f\}$ будет окрашено в красный цвет, то образуется еще минимум один треугольник с красными сторонами $v_a v_d v_b$ или $v_a v_c v_f$. Если оба эти ребра будут синего цвета, то появится треугольник $v_a v_f v_d$ с синими сторонами. Вывод нетрудно перевести с языка теории графов на язык задачи.



Установлено свойство графа, являющееся обобщением свойства 2.

Свойство 4. В любом полном графе с шестью или более вершинами и ребрами одного из двух цветов всегда найдутся два разных треугольника с одноцветными сторонами. Эти два треугольника могут иметь общую вершину или даже общее ребро.

Если **два треугольника** имеют общую вершину или ребро, то их называют **сцепленными**.

Рассмотрим свойства полного графа, ребра которого окрашены в один из трех цветов, каждый цвет соответствует одному из трех отношений между объектами заданного множества.

Задача 4. Каждый из семнадцати ученых переписывается с остальными. В их переписке речь идет лишь о трех темах. Каждая пара ученых переписывается друг с другом лишь по одной теме. Нужно доказать, что не менее трех ученых переписываются друг с другом по одной и той же теме.

Решение. Условию задачи соответствует полный граф с семнадцатью вершинами и ребрами трех цветов. Из каждой вершины выходит шестнадцать ребер. Докажем, что в таком графе найдется хотя бы один треугольник с одноцветными сторонами. Заметим, что каждая вершина этого графа принадлежит хотя бы шести ребрам одного цвета. Пусть, например, вершина v_a принадлежит шести красным ребрам.

Если среди вершин $v_b, v_c, v_d, v_e, v_f, v_h$ найдутся две, которые соединены красным ребром, то получится треугольник с красными сторонами. Если не найдутся, то все шесть вершин $v_b, v_c, v_d, v_e, v_f, v_h$ соединены между собой попарно ребрами двух цветов (зеленым и синим). Как было доказано ранее, в этом графе с шестью вершинами найдется хотя бы один треугольник либо с синими, либо с зелеными сторонами. Задача решена.

Сформулируем теперь свойство, доказанное при решении этой задачи.

Свойство 5. В полном графе с семнадцатью или более вершинами и

ребрами трех цветов всегда найдется, по меньшей мере, один треугольник с одноцветными сторонами.

Заметим, что не случайно отношения, которые были найдены при решении задач, изображавшиеся цветными ребрами, симметричны, если v_a — друг v_b , то v_b — друг v_a , но не обязательно транзитивны, если v_a — друг v_b и v_b — друг v_c , то v_a может и не быть другом v_c . В случае, когда отношение между объектами было транзитивным, соответствующие ребра образовывали треугольник с одноцветными сторонами.

Задача 5. В работе международного симпозиума лингвистов участвуют n человек. Из любых четырех один может объясняться с остальными тремя хотя бы на одном языке. Нужно доказать, что найдется участник симпозиума, который может объясниться с каждым из остальных участников.

Решение. Имеем полный граф с n вершинами и ребрами двух цветов (синее ребро — двое могут объясниться на каком-нибудь языке, красное — не могут). По условию, среди любых четырех вершин графа всегда найдется, по меньшей мере, одна, синяя степень которой равна трем.

Случай, когда все ребра синие, тривиален, математически неинтересен. Пусть найдется красное ребро $\{v_a, v_b\}$. Добавим еще какие-нибудь две вершины v_c, v_d . Из четырех вершин v_a, v_b, v_c, v_d найдется хотя бы одна синяя, степень которой равна трем. Это v_c или v_d . Пусть, например, синюю степень три имеет v_c . Добавим еще одну вершину — v_e . Из вершин v_a, v_b, v_c, v_e или v_c или v_e имеет синюю степень, равную трем. В обоих случаях v_c соединена синим ребром с v_e . Переберем все вершины. В итоге окажется, что v_c соединена синим ребром со всеми вершинами графа. Во всякой четверке вершин, включая v_a и v_b . Есть вершина, соединенная синим ребром со всеми остальными вершинами графа. Отсюда, кроме v_a и v_b , существует самое большее одна вершина, не соединенная синим ребром со всеми остальными.

Задача о несцепленных треугольниках с одноцветными сторонами

Задача 6. Назовем группу людей «однородной», если любая пара из этой группы психологически совместима или, напротив, любая пара психологически несовместима. Нужно доказать, что среди восьми случайно встретившихся незнакомцев всегда найдутся две однородные группы, состоящие из трех человек каждая, причем никто из первой группы не входит во вторую.

Иначе говоря, требуется доказать, что в графе с восемью вершинами и ребрами, окрашенными в один из двух цветов, обязательно найдутся

два треугольника с одноцветными сторонами, не сцепленные между собой.

Решение. Рассмотрим в графе один из треугольников $v_k v_l v_m$ с одноцветными сторонами. По теореме 3 такой треугольник всегда найдется. Если остальные пять вершин и ребра, соединяющие их попарно, содержат еще один треугольник с одноцветными сторонами, то он и будет являться вторым искомым треугольником. (Для этого случая задача решена.) Если остальные пять вершин v_a, v_b, v_c, v_d, v_e не содержат треугольника с одноцветными сторонами, то они образуют пятиугольник с красными сторонами и синими диагоналями. На рисунке (рис. 1) изображены не все ребра графа, соответствующего задаче, а лишь треугольник $v_k v_l v_m$ с красными сторонами и пятиугольник $v_a v_b v_c v_d v_e$ с красными сторонами и синими диагоналями.

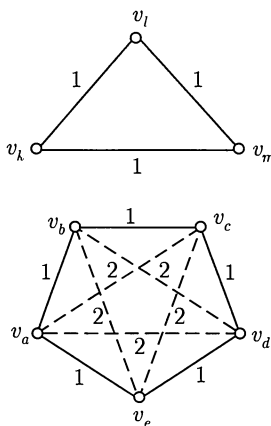


Рис. 1

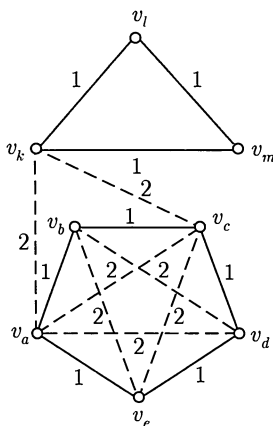


Рис. 2

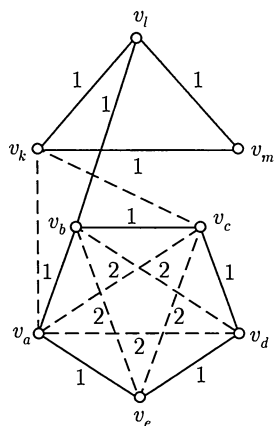


Рис. 3

Покажем, что если какая-нибудь вершина треугольника $v_k v_l v_m$ соединена синими ребрами с двумя вершинами пятиугольника, через одну, например v_k с v_a и v_c (рис. 2), то найдется еще один треугольник с одноцветными сторонами, не сцепленный с треугольником $v_k v_a v_c$.

Действительно, обратим внимание на пятиугольник $v_l v_m v_b v_d v_e$. Если он содержит треугольник с одноцветными сторонами, то второй треугольник с одноцветными сторонами, не сцепленный с первым, найден. Если не содержит, то ребра $\{v_b, v_l\}, \{v_b, v_m\}$ — красные, поскольку ребра $\{v_b, v_d\}, \{v_b, v_e\}$, по свойству 3, уже синие. То есть образован треугольник $v_b v_l v_m$ с красными сторонами, не сцепленный с треугольником $v_a v_c v_k$ (рис. 3).

Остается рассмотреть случаи, когда каждая вершина треугольни-

ка $v_k v_l v_m$ соединена красными ребрами, по меньшей мере, с тремя последовательными вершинами пятиугольника $v_a v_b v_c v_d v_e$. Тогда у пятиугольника найдутся две вершины, каждая из которых соединена красными ребрами с двумя вершинами треугольника $v_k v_l v_m$. На рисунках (рис. 4) и (рис. 5) показаны все такие случаи. На рисунке (рис. 1.) легко обнаружить два несцепленных треугольника $v_a v_b v_k$ и $v_c v_d v_m$. А на рисунке (рис. 5) хотя бы одно из ребер $\{v_a, v_l\}, \{v_c, v_l\}$ должно быть красным (иначе вершина v_l будет соединена синими ребрами с двумя вершинами пятиугольника $v_a v_b v_c v_d v_e$, взятыми через одну, а этот случай уже рассмотрен).

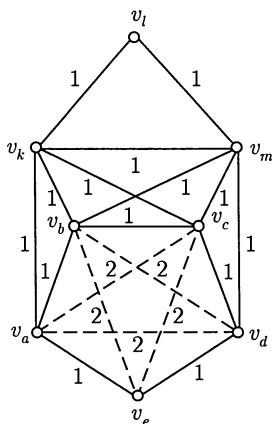


Рис. 4

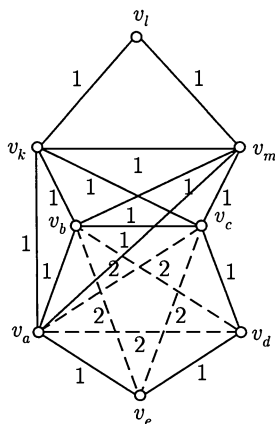


Рис. 5

Если хотя бы одно из ребер $\{v_a, v_l\}$ или $\{v_c, v_l\}$, красное, то появятся треугольники $v_c v_l v_m$ и $v_a v_b v_k$ или треугольники $v_a v_k v_l$ и $v_b v_c v_m$ с красными сторонами. Таким образом, во всех случаях найдутся два несцепленных треугольника с одноцветными сторонами. Задача решена и установлено еще одно свойство.

Свойство 6. В полном графе с восемью вершинами, ребра которого окрашены в два цвета, обязательно найдутся два треугольника с одноцветными сторонами, которые не являются сцепленными.

Лекция 8. Раскрашивание графов

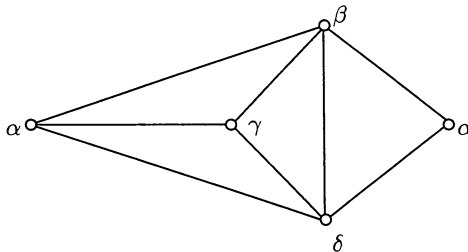
Хроматическое число. Гипотеза о четырех красках. Раскрашивание карт

Ключевые слова: k -раскрашиваемый граф, k -хроматический граф, хроматическое число графа G , карта, k -раскрашиваемая карта, вершинно k -раскрашиваемая карта, гипотеза четырех красок для карт, жорданова кривая, жорданова дуга, замкнутая жорданова кривая.

Хроматическое число

Рассмотрим задачу: при каких условиях вершины графа можно раскрасить так, чтобы каждое ребро было инцидентно вершинам разного цвета. Нас больше всего интересует вопрос, какие графы можно раскрасить с соблюдением определенных условий, чем вопрос, сколькими способами можно выполнить это раскрашивание.

Пусть граф G не имеет петель; тогда G называется *k -раскрашиваемым*, если каждой его вершине можно приписать один из k цветов таким образом, чтобы никакие две смежные вершины не оказались одного цвета. Если граф G является k -раскрашиваемым, но не является $(k - 1)$ -раскрашиваемым, назовем его *k -хроматическим*, а число k назовем *хроматическим числом графа G* и обозначим через $\chi(G)$. На рисунке изображен 4-хроматический (и, следовательно, k -раскрашиваемый граф при $k \geq 4$) граф; цвета обозначены греческими буквами



Для удобства будем предполагать, что все графы не содержат петель. Однако будем допускать существование кратных ребер, так как они не влияют на наши рассуждения.

Ясно, что $\chi(K_n) = n$, и, следовательно, легко построить графы со сколь угодно большим хроматическим числом. С другой стороны,

нетрудно видеть, что

$$\chi(G) = 1$$

тогда и только тогда, когда G — вполне несвязный граф, и что

$$\chi(G) = 2$$

тогда и только тогда, когда G — двудольный граф, отличный от вполне несвязного графа.

Теорема 8.1. *Если наибольшая из степеней вершин графа G равна ρ , то этот граф $(\rho + 1)$ -раскрашиваем.*

Доказательство. Проведем индукцию по числу вершин в G . Пусть G — граф с n вершинами; если из него удалить произвольную вершину v вместе с инцидентными ей ребрами, то в оставшемся графе будет $n - 1$ вершин, причем степени вершин по-прежнему не превосходят ρ . По предположению индукции этот граф $(\rho + 1)$ -раскрашиваем, отсюда получится $(\rho + 1)$ -раскладка для G , если окрасить вершину v цветом, отличным от тех, которыми окрашены смежные с ней вершины, а их не более чем ρ .

Теорема 8.2 (Брукса). *Пусть G — простой связный граф, не являющийся полным; если наибольшая из степеней его вершин равна ρ ($\rho \geq 3$), то он ρ -раскрашиваем.*

Доказательство. Проведем индукцию по числу вершин графа G . Предположим, что G имеет n вершин. Если при этом степень какой-нибудь его вершины меньше ρ , дальше можно рассуждать, как в доказательстве теоремы 1, и все будет закончено. Поэтому без потери общности можно считать граф G регулярным степени ρ .

Выберем произвольную вершину v и удалим ее, вместе с инцидентными ей ребрами. Останется граф с $n - 1$ вершинами, в котором наибольшая из степеней вершин не превосходит ρ . По предположению индукции этот граф ρ -раскрашиваем. Теперь окрасим вершину v в один из имеющихся ρ цветов. Как и раньше, считаем, что смежные с v вершины v_1, \dots, v_ρ расположены вокруг v по часовой стрелке и окрашены в различные цвета A_1, c_2, \dots, c_ρ .

Определяя подграфы H_{ij} ($i \neq j, 1 \leq i, j \leq \rho$), когда v_i, v_j лежат в разных компонентах графа H_{ij} . Таким образом, можно считать, что при любых данных i, j вершины v_i, v_j связаны простой цепью, целиком лежащей в H_{ij} . Обозначим компоненту графа H_{ij} , содержащую вершины v_i, v_j , через C_{ij} .

Ясно, что если вершина v_i смежная более чем с одной вершиной цвета c_j , то существует цвет, отличный от c_i , не приписанный никакой

из вершин, смежных с v_i . Тогда вершину v_i можно окрасить в этот цвет, что в свою очередь позволит окрасить вершину v в цвет c_i и закончить на этом доказательство теоремы. Если этот случай не имеет места, то используем аналогичное рассуждение, чтобы показать, что каждая вершина из C_{ij} (отличная от v_i и от v_j) должна иметь степень 2. Предположим, что w — первая вершина простой цепи из v_i в v_j , которая имеет степень больше 2; тогда w можно перекрасить в цвет, отличный от c_i или c_j , нарушая тем самым свойство, что v_i и v_j связаны простой цепью, целиком лежащей в C_{ij} . Поэтому мы можем считать, что для любых i и j компонента C_{ij} состоит только из простой цепи, соединяющей вершину v_i с v_j .

Заметим теперь, что две простые цепи вида C_{ij} и C_{jl} , где $i \neq l$, можно считать пересекающимися только в вершине v_j , так как если w

- другая точка пересечения, то ее можно перекрасить в цвет, отличный от c_i или c_j , или c_l , а это противоречит факту, что v_i, v_j связаны простой цепью.

Для завершения доказательства выберем (если это возможно) две несмежные вершины v_i, v_j и допустим, что w — вершина цвета c_j , смежная с v_i . Поскольку C_{il} — простая цепь (для любого $l \neq j$), можно поменять между собой цвета вершин в этой цепи, не затрагивая раскраску остальной части графа. Но это приводит к противоречию, потому что тогда w будет общей вершиной простых цепей C_{ij} и C_{jl} . Отсюда следует, что нельзя выбрать две вершины v_i и v_j несмежными, и поэтому G должен быть полным графом $K_{\rho+1}$. А так как это не допускается условием теоремы, то все возможные случаи рассмотрены.

Гипотеза о четырех красках

Уже сто с лишним лет математики пытаются доказать гипотезу четырех красок. В этом направлении был достигнут значительный прогресс. В печати появилось сообщение (K. Appel, W. Haken, Every planar map is four colorable, Bull. of Amer. Math. Soc., 82, № 5 (sept. 1976)), что гипотезу четырех красок удалось обосновать с использованием ЭВМ.

Сформулируем без доказательства несколько относящихся к этой проблеме результатов.

1. Если гипотеза четырех красок не верна, то любой опровергающий ее пример будет очень сложным. Известно, например, что всякий планарный граф, имеющий менее 52 вершин, 4-раскрашиваем.
2. Любой не содержащий треугольников планарный граф 3-раскрашиваем (теорема Греча).

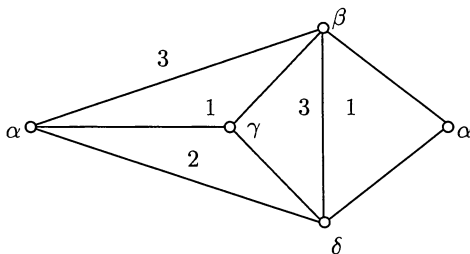
3. Если попытаться доказать гипотезу четырех красок, то достаточно доказать ее для гамильтоновых планарных графов (довольно неожиданный результат Уитни).

Раскрашивание карт

Возникновение гипотезы четырех красок исторически связано с раскрашиванием географических карт. Если имеется карта с изображением нескольких стран, то интересно узнать, сколько понадобится цветов для такой раскраски этих стран, чтобы никакие две соседние страны не были окрашены в один и тот же цвет. Возможно, самая привычная форма гипотезы четырех красок такова: любую карту можно раскрасить с помощью четырех красок.

Чтобы сделать это утверждение точным, надо определить, что означает слово «карта». Поскольку в рассматриваемых нами задачах о раскраске требуется, чтобы страны, расположенные по обе стороны ребра, были разного цвета, придется исключить карты, обладающие мостом. Таким образом, удобно определить *карту*, как связный плоский граф, не содержащий мостов. Заметим, что при таком определении карты не исключаем петель или кратных ребер.

Назовем карту *k-раскрашиваемой*, если ее грани можно раскрасить k красками так, чтобы никакие две смежные грани, то есть грани, границы которых имеют общее ребро, не были одного цвета. Там, где можно запутаться, будем использовать термин *вершинно k-раскрашиваемой*, имея в виду k -раскрашиваемость в описанном выше смысле. Например, изображенный ниже граф является 3-раскрашиваемым и вершинно 4-раскрашиваемым.



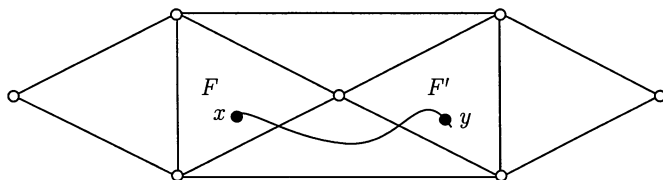
Теперь сформулируем *гипотезу четырех красок для карт*: всякая карта 4-раскрашиваема.

Теорема 8.3. *Карта G является 2-раскрашиваемой тогда и только тогда, когда G представляет собой эйлеров граф.*

Доказательство. Любую вершину v из G должно окружать четное число граней, так как их можно раскрасить в два цвета. Отсюда следует, что степень каждой вершины четна, и поэтому G — эйлеров граф.

Жордановой кривой, или **жордановой дугой**, на плоскости называется непрерывная кривая, не имеющая самопересечений; **замкнутой жордановой кривой** называется жорданова кривая, начало и конец которой совпадают.

Опишем метод, дающий нужную раскраску граней графа G . Выберем произвольную грань F и окрасим ее в красный цвет. Проведем жорданову кривую из точки x грани F в некоторую точку любой грани, причем так, чтобы эта кривая не проходила ни через какую вершину графа G . Если на пути от точки x до точки y грани F' наша кривая пересечет четное число ребер, окрасим грань F' в красный цвет; в противном случае — в синий.



Нетрудно показать, что раскрашивание определено корректно: берем «цикл», состоящий из двух таких жордановых кривых (то есть замкнутую жорданову кривую), и показываем, что он пересекает четное число ребер графа G (надо использовать индукцию по числу вершин, находящихся внутри цикла, и тот факт, что каждой вершине графа G инцидентно четное число ребер).

Лекция 9. Орграфы

Определения. Эйлеровы и гамильтоновы орграфы. Турниры

Ключевые слова: орграф, дуги или ориентированные ребра, множество вершин и семейство дуг орграфа D , основание орграфа, смежные вершины, вершины инцидентны дуге, дуга инцидентна вершинам, изоморфные орграфы, матрица смежности орграфа, простой орграф, ориентированный маршрут в орграфе, орцепи, простые орцепи и ориклы, орграф D связан, орграф слабо связан, орграф сильно связан, орсвязный, граф G ориентируемый, задание ориентации графа, приписывание направлений ребрам, эйлеров орграф, эйлерова орцепь, полустепень исхода, полустепень захода, орлема о рукопожатиях, источник орграфа, сток орграфа, гамильтонов орграф, полугамильтонов орграф, турнир.

Определения

Сначала напомним некоторые определения из первой лекции.

Орграфом D называется пара $(V(D), A(D))$, где $V(D)$ непустое конечное множество элементов, называемых **вершинами**, а $A(D)$ — конечное семейство упорядоченных пар элементов из $V(D)$, называемых **дугами** (или **ориентированными ребрами**). Дуга, у которой вершина v является первым элементом, а вершина w — вторым, называется **дугой из v в w** (v, w). Заметим, что дуги (v, w) и (w, v) различны. Хотя графы и орграфы — различные объекты, в определенных случаях графы можно рассматривать как орграфы, в которых каждому ребру соответствуют две противоположно ориентированные дуги. $V(D)$ и $A(D)$ называются соответственно множеством вершин и семейством дуг орграфа D .

На рисунке (рис. 1) представлен орграф, дугами которого являются $(u, v), (v, v), (v, w), (v, w), (v, w), (w, v), (w, u), (z, w)$. Порядок вершин на дуге указан стрелкой. Граф, полученный из орграфа D удалением стрелок, то есть заменой каждой дуги вида (v, w) на соответствующее ребро $\{v, w\}$, называется **основанием орграфа D** . Многие определения, данные для графов, можно перенести на орграфы. К примеру, две вершины v, w орграфа D называются **смежными**, если в $A(D)$ существует дуга вида (v, w) или (w, v) ; при этом вершины v, w называются **инцидентными любой такой дуге** (а дуга — **инцидентной соответствующим вершинам**). Два орграфа называются **изоморфными**, если существует изоморфизм между их основания-

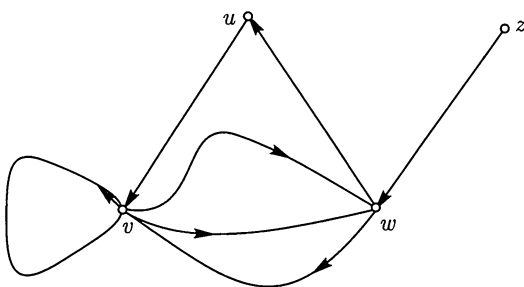


Рис. 1

ми, сохраняющий порядок вершин на каждой дуге. Матрицей смежности орграфа с множеством вершин $\{v_1, \dots, v_n\}$ является матрица, в которой a_{ij} равно числу дуг вида (v_i, v_j) в семействе $A(D)$. Матрица смежности для начерченного графа

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Орграфы, не содержащие петель и кратных ребер, называются **простыми**. **Ориентированный маршрут** в орграфе D представляет собой конечную последовательность дуг вида $(v_0, v_1), (v_1, v_2), \dots, (v_{m-1}, v_m)$. Эту последовательность можно записывать в виде $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_m$ и говорить об ориентированном маршруте из v_0 в v_m . Аналогичным образом можно определить ориентированные цепи, ориентированные простые цепи и ориентированные циклы — **орцепи, простые орцепи и орциклы**. Заметим, что хотя орцепь не может содержать данную дугу (v, w) более одного раза, она может содержать одновременно (v, w) и (w, v) . Например, на рис. 1 $z \rightarrow w \rightarrow v \rightarrow w \rightarrow u$ является орцепью.

Определим два наиболее естественных и полезных типа связности орграфов, которые возникают в соответствии с тем, хотим мы или нет принимать во внимание ориентацию дуг. Говорят, что **орграф D связан, или слабо связан**, если он не может быть представлен в виде объединения двух различных орграфов (определенных обычным образом). Это эквивалентно тому, что связно основание орграфа D . Предположим также, что для любых двух вершин v, w орграфа D существует простая орцепь из v в w , тогда D называется **сильно связным**. Этот термин настолько устоялся, что мы использовали его вместо более естественного «орсвязный». Ясно, что любой сильно связный граф связан, но обратное неверно. На рис. 1

изображен связный орграф, не являющийся сильно связным, так как не существует простой орцепи из v в z .

Различие между связным и сильно связным орграфом станет яснее, если рассмотреть план города, по всем улицам которого допускается только одностороннее движение. Тогда связность соответствующего орграфа означает, что можно проехать из любой части города в любую другую, *не обращая внимания на правила одностороннего движения*. Если же этот орграф сильно связан, то можно проехать из любой части города в любую другую, *следуя всегда «правильным путем» вдоль улиц с односторонним движением*.

Важно, чтобы система с односторонним движением была сильно связной, и естественно возникает вопрос: при каких условиях карту улиц можно превратить в систему с односторонним движением таким способом, чтобы можно было проехать из любой части города в любую другую? Если, к примеру, город состоит из двух частей, связанных одним мостом, то мы никогда не сможем сделать все его улицы односторонними, поскольку какое бы направление ни приписали мосту, одна часть города будет отрезана. Сюда включается и тот случай, когда в городе имеется тупик. С другой стороны, если мостов нет, то всегда найдется подходящая односторонняя система; это и есть основной результат, который будет установлен ниже в теореме.

Для удобства будем называть граф G **ориентируемым**, если каждое его ребро (рассматриваемое как пара вершин) может быть упорядочено таким образом, что полученный в результате орграф будет сильно связным. Этот процесс упорядочивания ребер будем называть **заданием ориентации графа** или **приписыванием направлений ребрам**. Если, например, G — граф, изображенный на рис. 2, то его можно ориентировать и получить сильно связный орграф, изображенный на рис. 3.

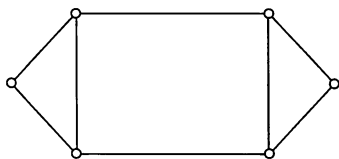


Рис. 2

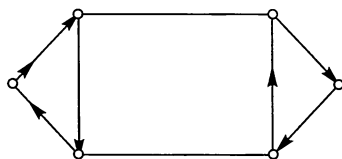


Рис. 3

Мы видим, что любой эйлеров граф ориентируем, поскольку достаточно пройти по любой эйлеровой цепи, ориентируя ребра в направлении движения по ним. Дадим необходимое и достаточное условие ориентируемого графа.

Теорема 9.1 (Роббинс). Пусть G — связный граф; он ориентируем тогда и только тогда, когда каждое его ребро содержится, по крайней мере, в одном цикле.

Доказательство. Необходимость условия очевидна. Чтобы доказать достаточность, выберем любой цикл C и ориентируем его ребра в направлении какого-либо обхода этого цикла. Если каждое ребро из G содержится в C , то доказательство завершено; если нет, то возьмем любое ребро e , не принадлежащее C , но смежное некоторому ребру из C . По предположению ребро e содержится в каком-то цикле C^1 . Ребрам цикла C^1 можно приписать ориентацию в направлении какого-либо обхода этого цикла (за исключением тех ребер, которые уже были ориентированы, то есть тех ребер из C^1 , которые принадлежат также C). Нетрудно убедиться в том, что описанная процедура за конечное число шагов приведет к сильно связному орграфу.

Эйлеровы и гамильтоновы орграфы

Связный орграф D называется **эйлеровым**, если в нем существует замкнутая орцепь, содержащая каждую его дугу. Такая **орцепь** называется **эйлеровой орцепью**.

Например, граф, изображенный на рисунке, не является эйлеровым, хотя его основание — эйлеров граф.

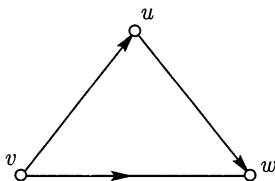


Рис. 4

Наша первая задача — найти условие, необходимое и достаточное для того, чтобы связный орграф был эйлеровым. Очевидно, что необходимым условием эйлеровости орграфа является его сильная связь. Если v вершина орграфа D , то называют **полустепенью исхода** v (обозначается $\vec{\rho}(v)$) — стрелка направлена от v) число дуг орграфа D , имеющих вид (v, w) . Аналогично, полустепенью захода v (обозначается $\overleftarrow{\rho}(v)$) называется число дуг из D вида (w, v) . Отсюда сразу следует, что сумма **полустепеней захода** всех вершин орграфа D равна сумме их полустепеней исхода, поскольку каждая дуга из D участвует в каждой сумме ровно один раз. Будем называть этот результат **орлемой о рукопожатиях**!

Источником орграфа D называют вершину, у которой полустепень захода равна нулю. **Стоком орграфа** D называют вершину, у которой полустепень исхода равна нулю. Так, на рис 4. вершина v является источником, а w — стоком. Заметим, что эйлеров орграф, кроме тривиального орграфа, не содержащего дуг, не может иметь ни источников, ни стоков.

Теорема 9.2. *Связный орграф является эйлеровым тогда и только тогда, когда $\vec{\rho}(v) = \overleftarrow{\rho}(v)$ для каждой его вершины v .*

Теорема дается без доказательства, так как оно аналогично тем, которые даны в четвертой лекции.

Орграф D называется **гамильтоновым**, если в нем существует орцикл, включающий каждую его вершину. **Орграф**, содержащий простую орцепь, проходящую через каждую вершину, называется **полугамильтоновым**. О гамильтоновых орграфах известно очень мало, к тому же некоторые теоремы о гамильтоновых графах, по-видимому, нелегко, если вообще возможно, обобщить на орграфы. Естественно спросить, обобщается ли на орграфы теорема Дирака? Одно такое обобщение принадлежит Гуйя-Ури. Доказательство этого утверждения значительно сложнее, чем доказательство теоремы Дирака, и выходит за рамки этого курса. Доказательство теоремы Гуйя-Ури можно найти в книге С. Berge, *Graphs and hypergraphs*, North-Holland, 1973, p. 196.

Теорема (Гуйя-Ури, 1973). *Пусть D — сильно связный орграф, имеющий n вершин. Если $\vec{\rho}(v) \geq n/2$ и $\overleftarrow{\rho}(v) \geq n/2$ для любой его вершины v , то D является гамильтоновым орграфом.*

Кажется, что получать результаты в этом направлении не очень просто, поэтому ограничимся рассмотрением вопроса о том, какие типы орграфов являются гамильтоновыми. В этом аспекте широко известен один тип орграфов — турниры. Для них соответствующие результаты принимают наиболее простую форму.

Турниры

Турниром называется орграф, в котором любые две вершины соединены ровно одной дугой. См. рисунок рис. 5.

Основанием для выбора такого названия служит то, что подобные орграфы можно использовать для записи результатов теннисных или любых других турниров, в которых не разрешены ничьи. Например, на рис. 5 представлены результаты турнира, в котором команда z нанесла поражение команде w , но проиграла команде v , и так далее.

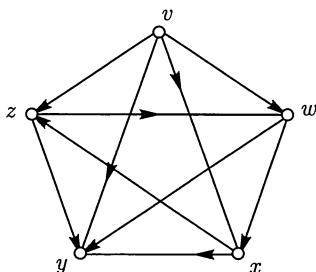


Рис. 5

Поскольку турнир может обладать источником или стоком, турниры не являются в общем случае гамильтоновыми орграфами. Однако следующая теорема (принадлежащая Реди и Камияну) показывает, что всякий турнир почти гамильтонов.

Теорема 9.3 (Реди, Камиян).

1. *Всякий турнир полугамильтонов.*
2. *Всякий сильно связный турнир гамильтонов.*

Доказательство. 1. Если турнир имеет меньше четырех вершин, то утверждение, очевидно, верно. Проведем индукцию по числу вершин. Предположим, что любой турнир с n вершинами полугамильтонов. Пусть T — турнир с $n + 1$ вершинами, и пусть турнир T' с n вершинами получен из T удалением некоторой вершины v вместе со всеми инцидентными ей дугами. Тогда по предположению индукции T обладает полугамильтоновой простой орцепью $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_n$. Рассмотрим три случая.

- 1) Если (v, v_1) — дуга в T , то искомой простой орцепью является $v \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_n$.
- 2) Если (v, v_1) не является дугой в T , это означает, что дугой является (v_1, v) и если существует такое i , что (v, v_i) — дуга в T , то выбирая первое i с таким свойством, получим, что искомой простой орцепью является (см. рис. 6) $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{i-1} \rightarrow v \rightarrow v_i \rightarrow \dots \rightarrow v_n$.
- 3) Если в T не существует дуги вида (v, v_i) , то искомой простой орцепью является $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_n \rightarrow v$.

2. Докажем более сильный результат, состоящий в том, что сильно связный турнир T с n вершинами содержит орциклы длины 3, 4, ..., n .

Сначала покажем, что T содержит орцикл длины три. Для этого выберем в T произвольную вершину v и обозначим через W множество всех

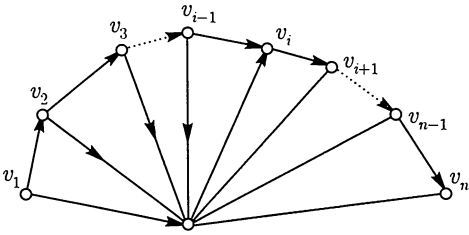


Рис. 6

вершин w , таких, что (v, w) — дуга в T , а через обозначим Z — обозначим множество всех вершин z , таких, что (z, v) — дуга в T . Так как T сильно связан, то оба множества T и Z не пусты, и поэтому в T найдется дуга вида (w', z') , где w' принадлежит W , z' принадлежит Z . Тогда требуемым циклом длины три является $v \rightarrow w' \rightarrow z' \rightarrow v$.

Осталось только показать, что если существует орцикл длины k ($k < n$), то существует и орцикл длины $k + 1$. Пусть $v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_1$ — орцикл длины k . Предположим сначала, что в T существует вершина v , не принадлежащая этому орциклу и обладающая тем свойством, что в T содержатся дуги вида (v, v_i) и вида (v_j, v) . Тогда должна существовать такая вершина v_i , что и (v_{i-1}, v) , и (v, v_i) являются дугами в T . При этом требуемым орциклом является (рис. 7)

$$v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{i-1} \rightarrow v \rightarrow v_i \rightarrow \dots \rightarrow v_k \rightarrow v_1.$$

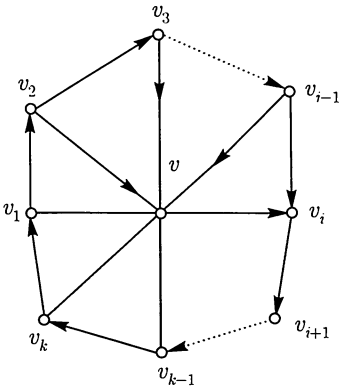


Рис. 7

Если не существует вершин, обладающих указанным выше свойством, то множество вершин, не содержащихся в орцикле, можно разбить

на два непересекающихся множества W и Z , где W есть множество таких вершин w , что (v_i, w) для любого i является дугой, а Z есть множество таких вершин z , что (z, v_i) для любого i является дугой. Так как T сильно связан, то оба множества W и Z непусты, и поэтому в T найдется дуга вида (w', z') , где w' принадлежит W , а z' принадлежит Z . Тогда требуемым орициклом будет (рис. 8)

$$v_1 \rightarrow w' \rightarrow z' \rightarrow v_3 \rightarrow \dots \rightarrow v_k \rightarrow v_1$$

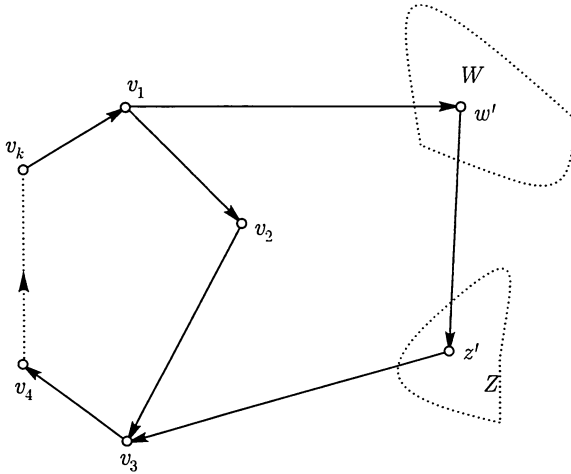


Рис. 8

Лекция 10. Цепи Маркова

Еще раз об ориентированных графах. Задачи на круговые бескомпромиссные турниры. Цепи Маркова

Ключевые слова: степень выхода, степень входа, изолированная вершина, источник, сток, путь, простой путь, замкнутый путь, длина пути, расстояние от v_a до v_b , бесконечное расстояние, полный ориентированный граф, круговой турнир, турнир в один круг, бескомпромиссный круговой турнир, случайное блуждание, вероятности перехода, вектор вероятностей, матрица перехода, дискретная стационарная цепь Маркова, состояние цепи, ассоциированный орграф данной цепи Маркова, неприводимая цепь Маркова, возвратное (или рекурсивное) состояние, состояние невозвратное, поглощающее состояние, состояние периодическое с периодом t ($t \neq 1$), состояние непериодическое, эргодическое состояние, эргодическая цепь.

Еще раз об ориентированных графах

Степенью выхода вершины v_a ориентированного графа называется число выходящих из v_a дуг (ребер). **Степенью входа** вершины v_a ориентированного графа называется число входящих в v_a дуг (ребер).

Изолированной вершиной называется вершина, у которой и степень входа и степень выхода равны 0. **Источником называется** вершина, степень выхода которой положительна, а степень входа равна 0. **Стоком** называется вершина, степень входа которой положительна, а степень выхода равна 0. **Путем** в ориентированном графе D от v_1 до v_n называется последовательность ориентированных дуг (ребер) $(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$, такая, что конец каждой предыдущей дуги (ребра) совпадает с началом следующей и ни одна дуга (ребро) не встречается более одного раза. Если в ориентированном графе D нашелся путь от v_a до v_b , то обратного пути от v_b к v_a может и не быть. **Простым путем** в ориентированном графе называется путь, в котором ни одна вершина не содержится более одного раза. **Замкнутый путь** в ориентированном графе называется ориентированным циклом. **Длиной пути** называется число дуг (ребер) в этом пути. Расстоянием от v_a до v_b в ориентированном графе называется длина наикратчайшего пути от v_a до v_b . Если пути от v_a до v_b не существует, то **расстояние от v_a до v_b называется бесконечным**. Оно обозначается ∞ .

Расстояние от v_a до v_b будем обозначать $S(v_a v_b)$. **Полным ориентированным графом** называется граф, каждая пара вершин которого соединена в точности одной ориентированной дугой (ребром). Если с каждого ребра (дуги) полного ориентированного графа снять направление, то образуется полный граф с неориентированными ребрами (дугами).

Задачи на круговые бескомпромиссные турниры

Напомним, что соревнование, в котором каждая из команд играет с каждой из остальных команд в точности по одному разу, называют **круговым турниром** или **турниром в один круг**. Если каждая встреча оканчивается непременно выигрышем одной из команд, то **круговой турнир** называется **бескомпромиссным**. Круговой бескомпромиссный турнир проводится, например, в волейболе и баскетболе. Так как мы рассматриваем исключительно бескомпромиссные круговые турниры, не возникает опасности спутать их с каким-либо другим видом турниров, такое соревнование будем называть сокращенно турниром. Каждому турниру соответствует полный ориентированный граф, в котором вершины представляют команды, а каждая ориентированная дуга (v_a, v_b) выражает отношение « v_a победила v_b ». Степень выхода любой вершины v_a есть число побед, одержанных командой v_a .

Задача 1. Турнир по волейболу проводится между n командами. Докажем, что если какие-нибудь две команды одержали в турнире одинаковое число побед, то найдутся среди участников три команды I, II, III такие, что I выиграла у II, II выиграла у III, а III выиграла у I.

Решение. Пусть v_a и v_b — две команды, одержавшие одинаковое число побед, например, p побед. Пусть к тому же v_a выиграла у v_b . Те p команд, у которых выиграла команда v_b , обозначим c_1, c_2, \dots, c_p (рис. 1).

Команда v_a не могла одержать победы над всеми командами из числа c_1, c_2, \dots, c_p , так как иначе она одержала бы больше, чем p побед.

Следовательно, среди команд c_1, c_2, \dots, c_p найдется хотя бы одна, которая одержала победу над v_a . Стрелку от нее направим v_a . Путь замкнется.

Сформулируем полученный результат на языке графов.

Теорема 10.1. Если в полном ориентированном графе с n вершинами хотя бы две вершины имеют одинаковые степени выхода, то в этом ориентированном графе найдутся 3 такие вершины, что дуги (ребра), соединяющие их, образуют ориентированный цикл.

Задача 2. Турнир между n шахматистами закончился без ничьих. Можно ли пронумеровать всех участников в таком порядке, чтобы ока-

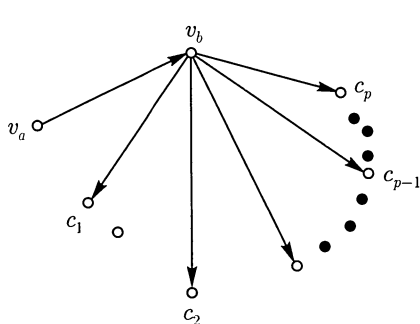


Рис. 1

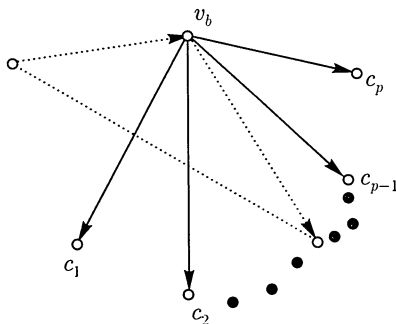


Рис. 2

залось, что каждый выиграл партию у шахматиста, имеющего номер на единицу больше?

Решение. Достаточно выяснить, что всякий полный ориентированный граф с n вершинами имеет простой путь, проходящий через все вершины орграфа. Доказательство: проведем методом математической индукции по числу вершин орграфа.

Для $n = 2$ утверждение верно. Теперь предположим, что в любом полном орграфе D с n вершинами найдется простой путь, проходящий через все вершины графа. Обозначим его $p_n = (a_1, a_2), (a_2, a_3), \dots, (a_{n-1}, a_n)$. Добавим теперь произвольную вершину a_{n+1} и ребра (дуги), соединяющие ее со всеми остальными вершинами орграфа D .

Если ребро (дуга), соединяющее a_{n+1} и a_n , направлено от a_n к a_{n+1} , то пройден путь p_n до a_{n+1} (рис. 3). Если ребро (дуга) направлено от a_{n+1} к a_n , то рассмотрим последовательность ребер (дуг), соединяющих a_{n+1} с $a_{n-1}, a_{n-2}, \dots, a_2, a_1$. Если все ребра (дуги) направлены от a_{n+1} , то к пути p_n можно добавить ребро a_{n+1}, a_1 .

Если они не все выходят из a_{n+1} , то возьмем первое ребро (дугу) этой последовательности, входящее в a_{n+1} . Пусть это будет ребро(дуга) (a_k, a_{n+1}) (рис. 4).

Прервем путь p_n в a_k и продолжим его по ребрам (дугам) (a_k, a_{n+1}) , (a_{n+1}, a_{k+1}) , после чего вновь вернемся к прежнему маршруту, то есть искомым путь будет следующим: $(a_1, a_2), \dots, (a_k, a_{n+1}), (a_{n+1}, a_{k+1}), (a_{k+1}, a_{k+2}), \dots, (a_{n-1}, a_n)$.

По принципу математической индукции утверждение верно для всякого натурального n .

А коль есть такой путь в графе, следовательно, всех игроков можно будет пронумеровать так, чтобы оказалось, что каждый выиграл партию у шахматиста, имеющего номер на единицу меньше.

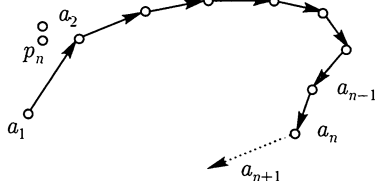


Рис. 3

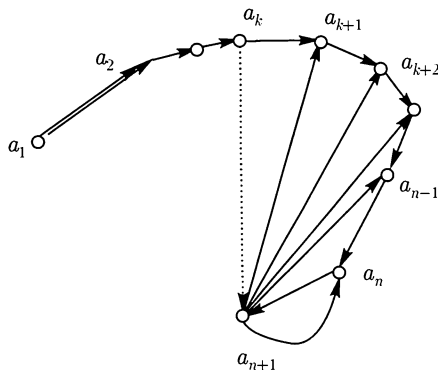


Рис. 4

Полученный результат сформулируем в виде теоремы.

Теорема 10.2. *Всякий полный орграф с n вершинами имеет простой ориентированный путь, проходящий через все вершины орграфа.*

Цепи Маркова

Орграфы возникают во многих жизненных ситуациях. Не пытаясь охватить большое число таких ситуаций, ограничимся рассмотрением одной из них. Интересующихся данными приложениями отошлем к главе 6 книги Басакера и Саати «Конечные графы и сети», «Наука», М., 1974.

Задача, которую мы рассмотрим, интересна сама по себе, а отчасти рассматриваем мы ее из-за того, что ее изложение не требует введения большого количества новых терминов.

Рассмотрим задачу об осле, стоящем точно между двумя копнами: соломы ржи и соломы пшеницы (рис. 5).

Осел стоит между двумя копнами: «Рожь» и «Пшеница» (рис. 5). Каждую минуту он либо передвигается на десять метров в сторону первой копны (с вероятностью $1/2$), либо в сторону второй копны (с вероятностью $1/3$), либо остается там, где стоял (с вероятностью $1/6$); такое поведение называется одномерным *случайным блужданием*. Будем предполагать, что обе копны являются «поглощающими» в том смысле, что если осел подойдет к одной из копен, то он там и останется. Зная расстояние между двумя копнами и начальное положение осла, можно поставить несколько вопросов, например: у какой копны он очутится с большей вероятностью и какое наиболее вероятное время ему понадобится, чтобы попасть туда?

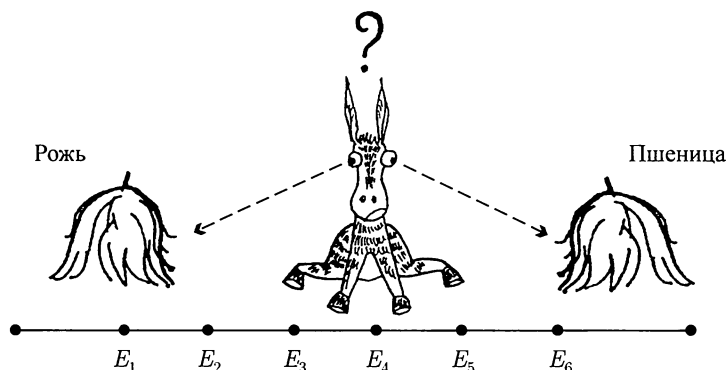


Рис. 5

Чтобы исследовать эту задачу подробнее, предположим, что расстояние между копнами равно пятидесяти метрам и что наш осел находится в двадцати метрах от копны «Пшеницы». Если места, где можно остановиться, обозначить через E_1, \dots, E_6 (E_1, E_6 — сами копны), то его начальное положение E_4 можно задать вектором $x = (0, 0, 0, 1, 0, 0)$, i -я компонента которого равна вероятности того, что он первоначально находится в E_i . Далее, по прошествии одной минуты вероятности его местоположения описываются вектором $(0, 0, 1/2, 1/6, 1/3, 0)$, а через две минуты — вектором $(0, 1/4, 1/6, 13/36, 1/9, 1/9)$. Ясно, что непосредственное вычисление вероятности его нахождения в заданном месте по прошествии k минут становится затруднительным. Оказалось, что удобнее всего ввести для этого матрицу перехода.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 1/6 & 1/3 & 0 & 0 & 0 \\ 0 & 1/2 & 1/6 & 1/3 & 0 & 0 \\ 0 & 0 & 1/2 & 1/6 & 1/3 & 0 \\ 0 & 0 & 0 & 1/2 & 1/6 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Рис. 6

Пусть p_{ij} — вероятность того, что он переместится из E_i в E_j за одну минуту. Например, $p_{23} = 1/3$ и $p_{24} = 0$. Эти вероятности p_{ij} называются

вероятностями перехода, а (6×6) -матрицу $P = (p_{ij})$ называют **матрицей перехода** (рис. 6.). Заметим, что каждый элемент матрицы P неотрицателен и что сумма элементов любой из строк равна единице. Из всего этого следует, что x — начальный вектор-строка, определенный выше, местоположение осла по прошествии одной минуты описывается вектором-строкой xP , а после k минут — вектором xP^k . Другими словами, i -я компонента вектора xP^k определяет вероятность того, что по истечении k минут осел оказался в E_i .

Можно обобщить эти понятия. Назовем **вектором вероятностей** вектор-строку, все компоненты которого неотрицательны и дают в сумме единицу. Тогда **матрица перехода** определяется как квадратная матрица, в которой каждая строка является вектором вероятностей. Теперь можно определить цепь Маркова (или просто цепь) как пару (P, x) , где P есть $(n \times n)$ -матрица перехода, а x есть $(1 \times n)$ -вектор-строка. Если каждый элемент p_{ij} из P рассматривать как вероятность перехода из позиции E_i в позицию E_j , а x — как начальный вектор вероятностей, то придем к классическому понятию **дискретной стационарной цепи Маркова**, которое можно найти в книгах по теории вероятностей (см. Феллер В., Введение в теорию вероятностей и ее приложения, т. 1, «Мир», М., 1967.) Позиция E_i обычно называется **состоянием цепи**, опишем различные способы их классификации.

Нас будет интересовать следующее: можно ли попасть из одного данного состояния в другое, и если да, то за какое наименьшее время. Например, в задаче об осле из E_4 в E_1 можно попасть за три минуты и вообще нельзя попасть из E_1 в E_4 . Следовательно, в основном мы будем интересоваться не самими вероятностями p_{ij} , а тем, положительны они или нет. Тогда появляется надежда, что все эти данные удастся представить в виде орграфа, вершины которого соответствуют состояниям, а дуги указывают на то, можно ли перейти из одного состояния в другое за одну минуту. Более точно, если каждое состояние E_i представлено соответствующей ему вершиной v_i , то, проводя из v_i в v_j для тех и только тех вершин, для которых $p_{ij} \neq 0$, мы и получим требуемый орграф. Кроме того, этот орграф можно определить при помощи его матрицы смежности, если заменить каждый ненулевой элемент матрицы P на единицу. Мы будем называть этот орграф **ассоциированным орграфом данной цепи Маркова**. Ассоциированный орграф одномерного случайного блуждания, связанного с задачей об осле, изображен на рис. 7.

Другой пример: если цепь Маркова имеет матрицу перехода, приведенную на рис. 8, то ассоциированный орграф этой цепи выглядит так, как показано на рис. 9.

Теперь ясно, что в цепи Маркова из состояния E_i в состояние E_j



Рис. 7

$$\begin{pmatrix}
 0 & 1/4 & 1/2 & 0 & 0 & 1/4 \\
 0 & 1 & 0 & 0 & 0 & 0 \\
 1/2 & 1/3 & 0 & 1/12 & 0 & 1/12 \\
 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 0 & 1 \\
 0 & 0 & 0 & 1 & 0 & 0
 \end{pmatrix}$$

Рис. 8

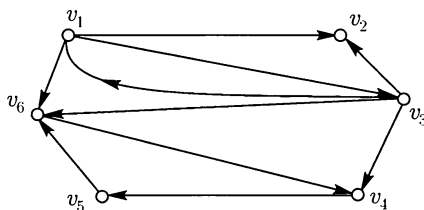


Рис. 9

можно попасть в том и только в том случае, если в ассоциированном орграфе существует орцепь из v_i в v_j , и что наименьшее возможное время попадания равно длине кратчайшей из таких орцепей. Цепь Маркова, в которой из любого состояния можно попасть в любой другой, называется **неприводимой**. Ясно, что цепь Маркова неприводима тогда и только тогда, когда ее ассоциированный орграф сильно связан. Заметим, что ни одна из описанных выше цепей не является неприводимой.

При дальнейших исследованиях принято различать те состояния, в которые мы продолжаем возвращаться независимо от продолжительности процесса, и те, в которые мы попадаем несколько раз и никогда не возвращаемся. Более точно это выглядит так: если начальное состояние есть E_i и вероятность возвращения в E_i на некотором более позднем шаге равна единице, то E_i называется **возвратным (или рекурсивным) состоянием**. В противном случае **состояние E_i называется невозвратным**.

В задаче об осле, например, очевидно, что состояния E_1 и E_6 являются возвратными, тогда как все другие состояния — невозвратными. В более сложных примерах вычисление нужных вероятностей становится очень хитрым делом, и поэтому проще бывает классифицировать состояния, анализируя ассоциированный орграф цепи. Нетрудно понять, что состояние E_i возвратно тогда и только тогда, когда существование простой орцепи из v_i в v_j в ассоциированном орграфе влечет за собой существование простой орцепи из v_j в v_i . В орграфе, изображенном на рис. 9, существует простая орцепь из v_1 в v_4 , но нет ни одной орцепи из v_4 в v_1 . Следовательно, состояния E_1 и аналогично E_3 невозвратны (E_2, E_4, E_5, E_6 возвратны). Состояние (такое, как E_2), из которого нельзя попасть ни в какое другое, называется *поглощающим состоянием*.

Другой прием классификации состояний опирается на понятие периодичности состояний. Состояние E_i цепи Маркова называется *периодическим* с периодом t ($t \neq 1$), если в E_i можно вернуться только по истечении времени, кратного t . Если такого t не существует, то состояние E_i называется *непериодическим*. Очевидно, что каждое состояние E_i , для которого $p_{ii} \neq 0$, непериодическое. Следовательно, каждое поглощающее состояние — непериодическое. В задаче об осле не только поглощающее состояние E_1, E_6 , но и все остальные являются непериодическими. С другой стороны, во втором примере (рис. 9) поглощающее состояние E_2 — единственное непериодическое состояние, поскольку E_1, E_3 имеют период два, а E_4, E_5, E_6 — период три. Используя терминологию орграфов, легко показать, что состояние E_i является периодическим с периодом t тогда и только тогда, когда в ассоциированном орграфе длина каждой замкнутой орцепи, проходящей через v_i , кратна t .

И, наконец, для полноты изложения введем еще одно понятие: назовем состояние цепи Маркова *эргодическим*, если оно одновременно возвратно и непериодично. Если любое состояние цепи Маркова является эргодическим, то назовем ее *эргодической цепью*.

Лекция 11. О деревьях

Представления деревьев. Представление с помощью матрицы смежности. Представление с помощью списков смежности. Представление с помощью списка ребер и кода Прюфера. Алгоритм построения кода Прюфера. Алгоритм раскодирования. Уровневые коды корневых деревьев. Перечисление и подсчет деревьев. Непомеченные деревья. Ориентированные деревья. Каркасы в неориентированном графе. Каркасы в ориентированных графах.

Ключевые слова: представление дерева, функция кода Прюфера, функция распаковки, уровневый код, канонический уровневый код.

Представления деревьев

Представление с помощью матрицы смежности. Представление с помощью списков смежности. Представление с помощью списка ребер и кода Прюфера. Алгоритм построения кода Прюфера. Алгоритм раскодирования. Уровневый код. Канонический уровневый код.

Каждому дереву можно поставить в соответствие некоторой код. С помощью этого кода можно восстановить дерево с точностью до изоморфизма. Существуют различные способы кодировки деревьев, которые позволяют решать конкретные задачи (подсчет деревьев, установление изоморфизма, генерирование всех неизоморфных деревьев и т. д.). *Представлением дерева* называется способ записи информации о нем, однозначно и полностью восстанавливающий структуру дерева и позволяющий вычислить его характеристики. Выбор представления зависит от решаемой задачи и способа ее решения. Рассмотрим наиболее распространенные способы задания деревьев.

Представление с помощью матрицы смежности

Это представление является общим для всех видов графов; оно задает граф с точностью до изоморфизма, но вместе с тем данное представление неэкономично, так как ненулевыми являются для n -вершинного дерева только $2 \times n - 2$ из n^2 элементов матрицы.

Задание графа *матрицей смежности*, размера $n \times n$, где n — число вершин графа. $A(i, j) = 1$, если вершины i и j смежные, в противном случае $A(i, j) = 0$. A — симметрическая матрица для неориентированного графа и несимметрическая для ориентированного.

Для неориентированного графа матрица смежности симметрична относительно главной диагонали, поэтому можно задавать только верхнюю треугольную половину матрицы, но это не улучшает ситуацию. Другой недостаток этого представления заключается в том, что трудоемкость алгоритмов, работающих с таким представлением, не может быть ниже.

Пример.

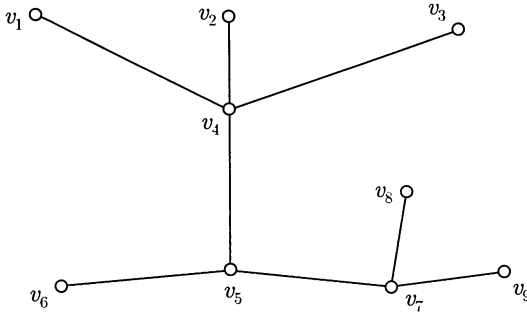


Рис. 1

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Рис. 2

Представление с помощью списков смежности

В этом представлении каждой вершине дерева сопоставляется список смежных вершин вида $v_{i_1}, v_{i_2}, \dots, v_{i_n}$.

Для дерева из предыдущего примера списки смежности имеют вид:

$$\begin{array}{l}
 v_4 : v_1, v_2, v_3, v_5; \\
 v_5 : v_4, v_6, v_7; \\
 v_7 : v_5, v_8, v_9; \\
 \left. \begin{array}{l}
 v_1 : v_4; \\
 v_2 : v_4; \\
 v_3 : v_4; \\
 v_6 : v_5; \\
 v_8 : v_7; \\
 v_9 : v_7;
 \end{array} \right\} \text{ — обязательно}
 \end{array}$$

При машинной организации списки смежности могут быть связаны между собой разными способами, например, копируя структуру дерева.

Представление с помощью списка ребер и кода Прюфера

Дерево при этом способе задается перечислением пар (v_i, v_j) или троек (v_i, v_j, u_k) , если дополнительно нужна нумерация ребер. Характер связей в списке определяется исходя из условий задачи.

Для дерева, изображенного на рис. 1, имеем:

$$(v_1, v_4), (v_2, v_4), (v_3, v_4), (v_4, v_5), (v_5, v_6), (v_5, v_7), (v_7, v_8), (v_7, v_9).$$

Алгоритм построения кода Прюфера

Пусть T — дерево с множеством вершин $\{v_1, v_2, \dots, v_n\}$. Будем считать, что номер вершины v_i равен i . Сопоставим дереву T последовательность $\{a_1, a_2, \dots, a_{n-2}\}$ по следующему правилу, представленному в виде функции: **Функция кода Прюфера** (T : дерево) =

1. Пусть n обозначает число вершин в T , а A — целочисленный вектор длины $n - 2$;
2. $B = [1 : n]$;
3. Для i от 1 до $n - 1$ цикл
4. $b = \min\{k \in B . k \text{ — номер висячей вершины}\}$;
5. $a[i]$ — номер вершины, с которой смежна вершина с номером b ;

6. $B = B - \{b\}$;
7. удалить из T вершины с номером b ;
8. возврат A .

Пример. Для дерева T (см. рис. 2. код Прюфера имеет вид: $P_2(T) = [2, 5, 5, 5, 6, 6, 10, 9, 10, 11, 13, 15, 15, 10, 13, 13, 13]$.

В случае корневого ордерова процедура получения кода Прюфера аналогична. Необходимо только на последнем месте указывать корневую вершину и при распаковке кода исключать номер этой вершины из множества B .

Алгоритм раскодирования

Распаковка кода Прюфера осуществляется следующей функцией.

Функция распаковки (A : код) =

1. Пусть T состоит из вершин $\{v_1, v_2, \dots, v_n\}$ таких, что номер вершины v_i равен i , где n — длина кода A плюс 2;
2. $B = [1 : n]$;
3. Для i от 1 до $n + 1$ цикл
4. $b = \min\{k \in B, k \neq A[j] \text{ для любого } j \geq i\}$;
5. В T добавить ребро, соединяющее вершины с номерами b и $A[i]$;
6. $B = B - \{b\}$;
7. возврат T .

Уровневые коды корневых деревьев

Пусть (T, z) обозначает корневое дерево с лежащим в его основе свободным деревом T и корнем z . Уровень вершины v в (T, z) — это расстояние от z до v плюс единица. **Уровневый код** (обозначение $L(T, z) = [l_1, l_2, \dots, l_n]$) — это последовательность целых чисел, полученная выпиыванием уровней вершин дерева (T, z) в постфиксном порядке.

Уровневый код называется **каноническим** (обозначается $L^*(T, z)$), если он является наибольшим в лексикографическом упорядочении среди всех уровневых кодов, описывающих дерево.

Пример. Для дерева T , изображенного на рисунке 3, имеем $L(T, z) = [3, 3, 2, 4, 4, 3, 2, 2, 1]$ — обычный уровневый код, а канонический уровневый код $L^*(T, z) = [4, 4, 3, 3, 2, 3, 3, 2, 2, 1]$.

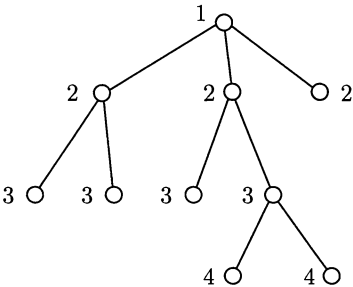


Рис. 3

Перечисление и подсчет деревьев

Теорема (Кэли). Число t_n помеченных деревьев с n вершинами равно $t_n = n^{n-2}$.

Теорема (Скойнса). Число 2-раскрашенных деревьев с t вершинами одного цвета и n вершинами другого равно $S_n = n^{t-1}t^{n-1}$.

Теорема (Рида). Число помеченных гомеоморфно несводимых деревьев равно $h_n = (n - 2)! \sum_{k=2}^n (-1)^{n-k} \binom{n}{k} \frac{k^{k-2}}{(k-2)!}$.

Непомеченные деревья

Пусть $T(x) = \sum_{n=1}^\infty T_n x^n$ — производящая функция для корневых деревьев. Таким образом, T_n представляет собой число корневых деревьев с n вершинами.

Теорема (Пойа). Перечисляющий ряд корневых деревьев удовлетворяет соотношению

$$T(x) = x \cdot \exp \left\{ \sum_{k=1}^\infty T(x^k)/k \right\}. \tag{*}$$

Из этой теоремы следует, что $T(x)$ однозначно определяется функциональным уравнением (*). Из данного уравнения выводится формула для $T(x)$. Данную зависимость получил Кэли: $T(x) = x \cdot \prod_{p=1}^{\infty} (1 - x^p)^{-T_p}$.

Следующая рекурсивная функция для вычисления $T(n)$ принадлежит Оттеру.

Пусть $t(x) = \sum_{n=1}^{\infty} t_n x^n$ — производящая функция для деревьев, так что t_n есть число деревьев с n вершинами.

Теорема (Оттера). *Ряд t_n , перечисляющий деревья, выражается через ряд $T(x)$ для корневых деревьев с помощью формулы*

$$t(x) = T(x) - \frac{1}{2} (T^2(x) - T(x^2)).$$

Ориентированные деревья

Пусть $r(x)$ и $R(x)$ — перечисляющие ряды для ориентированных и для корневых ориентированных деревьев соответственно.

Теорема (Харари—Принса). *Перечисляющие ряды $r(x)$ и $R(x)$ для ориентированных и для корневых ориентированных деревьев удовлетворяют соотношениям $T(x) = x \cdot \left(\exp \left\{ \sum_{k=1}^{\infty} R(x^k)/k \right\} \right)^2$ и $r(x) = R(x) - R^2(x)$.*

Каркасы в неориентированном графе

Число каркасов в неориентированном графе определяется с помощью следующей матричной теоремы о деревьях в графе. Пусть $M(G)$ обозначает матрицу, получаемую из матрицы $A(G)$, где $A(G)$ — матрица смежности графа G , с помощью подстановки в ней на место i -го диагонального элемента числа $\deg v_i$.

Матричная теорема о деревьях для графов. *Для всякого связного помеченного графа G все алгебраические дополнения матрицы $M(G)$ равны друг другу и их общее значение представляет собой число каркасов графа G .*

Пример. Для графа G (рис. 4) с матрицей смежности

$$A(G) = \begin{vmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{vmatrix}$$

матрица $M(G)$ имеет вид

$$M(G) = \begin{vmatrix} 2 & -1 & -1 & 0 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ 0 & 0 & -1 & 1 \end{vmatrix}.$$

Алгебраическое дополнение, например, элемента $a_{1,4}$, равно 3. Соответствующие каркасы графа G показаны на рис. 5.

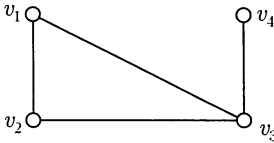


Рис. 4

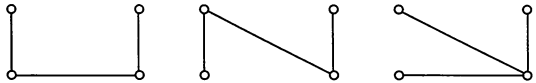


Рис. 5

Интересен также следующий результат. Пусть G — n -вершинный граф без петель и B_0 — его матрица инцидентности с одной удаленной строкой (т. е. с $n-1$ независимыми строками). Пусть B_0^t — транспонированная матрица к B_0 . Тогда определитель $|B_0 B_0^t|$ равен числу остовных деревьев графа G .

Каркасы в ориентированных графах

Число каркасов в ориентированном графе определяется с помощью аналогичной матричной теоремы о деревьях в орграфе. Пусть G — орграф с матрицей смежности $A(G)$. Определим диагональную матрицу M_{out} , у которой (i, i) -й элемент равен полу степени исхода $\deg^+ v_i$ вершины v_i . Затем положим $C_{\text{out}} = M_{\text{out}} - A(G)$. Аналогично определяется матрица $C_{\text{in}} = M_{\text{in}} - A(G)$.

Матричная теорема о деревьях для орграфов. Все алгебраические дополнения i -й строки матрицы C_{out} равны друг другу, и их общее значение есть число каркасов орграфа G , входящих в вершину v_i . Двойственным образом общее значение алгебраических дополнений i -го столбца матрицы C_{in} равно числу каркасов, выходящих из вершины v_i .

Пример. Для графа G (см. рис. 6) матрицы C_{out} и C_{in} имеют вид:

$$C_{\text{out}} = \begin{vmatrix} 2 & -1 & 0 & 0 & -1 \\ 0 & 2 & -1 & -1 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & -1 & 0 & -1 \\ -1 & 0 & 0 & 0 & 1 \end{vmatrix} \quad C_{\text{in}} = \begin{vmatrix} 1 & -1 & 0 & 0 & -1 \\ 0 & 2 & -1 & -1 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 2 & -1 \\ -1 & 0 & 0 & 0 & 2 \end{vmatrix}$$

Используя их, убеждаемся сразу, исходя из первой строки матрицы C_{out} и из первого столбца матрицы C_{in} , что орграф G имеет в точности четыре каркаса, выходящих из вершины 1, и два каркаса, входящих в эту вершину.

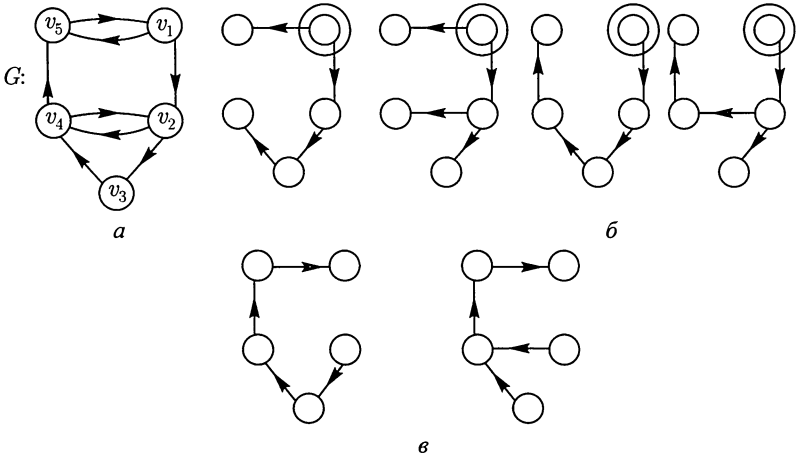


Рис. 6

Лекция 12. Каркасы и изоморфизм деревьев

Каркас неориентированного графа. Нахождение каркасов в графе. Алгоритм Краскала. Изоморфизм деревьев.

Ключевые слова: оптимальный каркас взвешенного графа, кратчайшая связывающая сеть для данного графа, изоморфные графы, инвариант графа, инцидентор, изоморфные деревья, отношение деревьев рефлексивно, симметрично, транзитивно, два дерева изоморфны, автоморфизм дерева, группа дерева, число симметрий дерева, асимметричное дерево, плоские изоморфные деревья, помеченные изоморфные деревья.

Каркас неориентированного графа

Оптимальным каркасом взвешенного графа называется каркас, минимизирующий некоторую функцию от весов входящих в него ребер. Чаще всего в качестве такой функции выступает сумма весов ребер, режé — произведение. Оптимальный каркас еще называют кратчайшей *связывающей сетью* для данного графа.

Задача о построении кратчайшей связывающей сети встречается в различных приложениях достаточно часто.

Нахождение каркасов в графе

Алгоритм нахождения каркаса на основе поиска в глубину.

Вход. Связный граф $G(V, E)$, заданный списками смежности

ЗАПИСЬ $[v]$, $v \in V$.

Выход. Каркас (V, T) графа G .

Процедура $WGD(G; \text{граф}; v: \text{вершина})$

//поиск в глубину, с нахождением ребер дерева; переменные **НОВЫЙ**, ЗАПИСЬ, T — глобальные//

Шаг 1. **НОВЫЙ** $[v]$ = ложь;

Шаг 2. Для $u \in \text{ЗАПИСЬ}[v]$ цикл

Шаг 3. если **НОВЫЙ** $[u]$ то $/(v, u)$ — новое ребро//

Шаг 4. $T = T \cup \{(v, u)\}$;

Шаг 5. $WGD(u)$;

все

все;

Шаг 6. Начало //главная программа//

Шаг 7. Для $u(V$ цикл **НОВЫЙ** $[v] =$ истина все;

Шаг 8. $T = \emptyset$ //множество найденных к этому моменту ребер//

Шаг 9. $WGD(r)$; // r — произвольная вершина графа//

все

конец.

Алгоритм нахождения каркаса на основе поиска в ширину.

Вход. Связный граф $G(V, E)$, представленный списками смежности **ЗАПИСЬ** $[v]$, $v(V)$.

Выход. Каркас (V, T) (V, T) графа G .

Процедура **КАРКАС**(G : граф; v : вершина)

Шаг 1. для $u \in V$ цикл **НОВЫЙ** $[u] =$ истина все; //инициализация//

Шаг 2. $T = \emptyset$; // — множество найденных к этому моменту ребер//

Шаг 3. **ОЧЕРЕДЬ** $= \emptyset$; **ОЧЕРЕДЬ** $\leftarrow r$; //корень каркаса//

Шаг 4. **НОВЫЙ** $[r] =$ ложь;

Шаг 5. Пока **ОЧЕРЕДЬ** $\neq \emptyset$ цикл

Шаг 6. $v \leftarrow$ **ОЧЕРЕДЬ**;

Шаг 7. Для $u \in$ **ЗАПИСЬ** $[v]$ цикл

Шаг 8. если **НОВЫЙ** $[u]$ то // (v, u) — новое ребро//

Шаг 9. **ОЧЕРЕДЬ** $\leftarrow u$;

Шаг 10. **НОВЫЙ** $[u] =$ ложь;

Шаг 11. $T = T \cup \{v, u\}$

все

все

все

Все.

Алгоритм Краскала

Вход. Неориентированный граф $G = (V, E)$ с функцией стоимости ребер c .

Выход. Оптимальный каркас $T = (V, S)$.

Метод

Процедура **Оптимальный каркас** (G : граф) =

Шаг 1. $S = \emptyset$

Шаг 2. $VS = \emptyset$

Шаг 3. Построить очередь с приоритетами Q , содержащую все ребра из E ;

Шаг 4. Для всех v из T цикла добавить $\{v\}$ к VS все;

Шаг 5. Пока $|VS| > 1$ цикл

Шаг 6. Выбрать в Q ребро (v, w) наименьшей стоимости;

Шаг 7. удалить (v, w) из Q ;

Шаг 8. Если v и w принадлежат различным множествам W_1 и W_2

из VS

Шаг 9. То заменить W_1 и W_2 на $W_1 \cup W_2$ в VS :

Шаг 10. добавить (v, w) к S

 все

 все

все.

Пример.

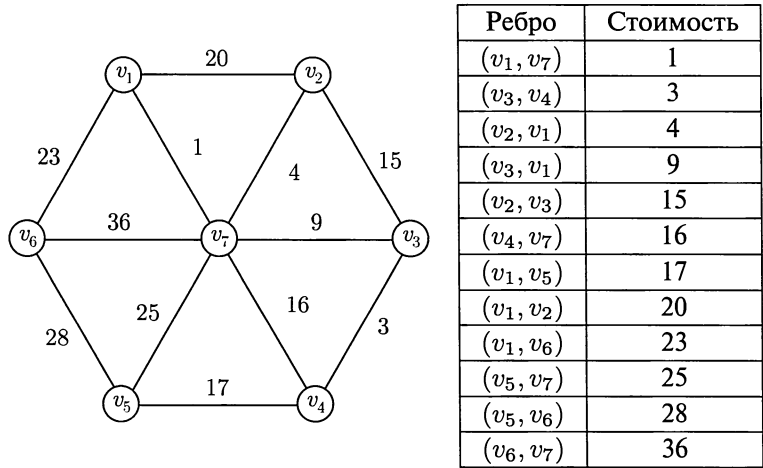


Рис. 1

Для графа, приведенного на рисунке 1, оптимальным каркасом будет — это можно проверить самостоятельно.

Изоморфизм деревьев

Два графа G и H изоморфны (записывается $G \cong H$ или иногда $G = H$), если между множествами вершин существует взаимно однозначное соответствие, сохраняющее смежность.

Инвариант графа G — это число, связанное с G , которое принимает одно и то же значение на любом графе, изоморфном G . Полный набор инвариантов определяет граф с точностью до изоморфизма.

Пусть дерево $T = (V, E)$ и его $P(e, v, w)$ **инцидентор**, т. е. предикат, равный единице в том и только в том случае, когда ребро e инцидентно вершинам v и w .

Дерево $T_1 = (V_1, E_1)$ называется изоморфным дереву $T_2 = (V_2, E_2)$, если между вершинами деревьев, а также между их ребрами можно установить взаимно однозначное соответствие $V_1 \Leftrightarrow V_2$ и $E_1 \Leftrightarrow E_2$, сохраняющее инцидентор, т. е. такое, что $(\forall v, w \in V_1) (\forall v', w' \in V_2) (\forall e \in E_1) (\forall e' \in E_2) (v \Leftrightarrow v' \& w \Leftrightarrow w' \& e \Leftrightarrow e' \Rightarrow [P_1(e, v, w)] = P_2(e', v', w'))$.

Другими словами, два дерева изоморфны, если между их вершинами можно установить взаимно однозначное соответствие, сохраняющее отношение инцидентности (смежности).

Отношение деревьев рефлексивно, симметрично, транзитивно, т. е. представляет собой транзитивность. Матрицы смежности изоморфных деревьев могут быть переведены одна в другую перестановкой рядов, т. е. одновременной одинаковой перестановкой строк и столбцов.

Пример изоморфных деревьев: нетрудно заметить, что они отличаются лишь способом представления (например, способом изображения на плоскости).

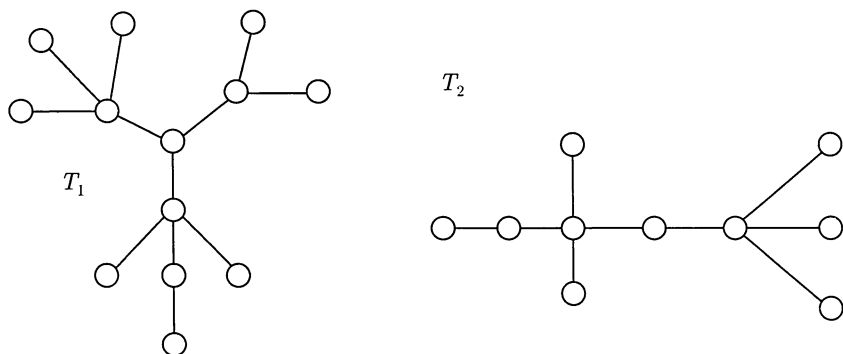


Рис. 2

Два корневых дерева называются изоморфными, если существует взаимно однозначная функция, которая отображает множество вершин одного дерева на множество другого и не только сохраняет смежность, но и переводит корень одного дерева в корень другого.

Изоморфное отображение дерева T на себя называется *автоморфизмом дерева*. Совокупность всех автоморфизмов дерева T , обозначаемая $\Gamma(T)$, образует группу, называемую *группой дерева T* . Таким образом, элементы группы $\Gamma(T)$ являются подстановками, действующими на множестве вершин V . Порядок $s(T)$ группы $\Gamma(T)$ есть *число симметрий дерева T* .

Дерево T называется *асимметричным*, если его группа автоморфизмов есть единичная группа, т. е. $s(T) = 1$.

Примеры асимметричных деревьев порядка 7, 8, 9.

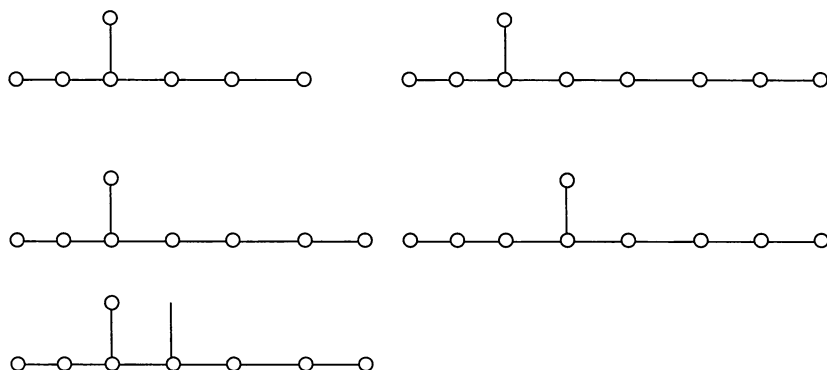


Рис. 3

Два плоских дерева называются изоморфными, если существует гомеоморфное отображение плоскости на себя, сохраняющее ориентацию, и такое, что отображает одно из этих деревьев в другое.

Два помеченных дерева называются изоморфными, если существует взаимно однозначная функция, отображающая множество вершин одного дерева на множество вершин другого, которая не только сохраняет смежность, но и переводит вершину с меткой в вершину с той же меткой.

Лекция 13. Деревья, вероятность и генетика

Поиск кратчайшего пути. Вероятность и генетика

Ключевые слова: коэффициент кровного родства.

Отыскание кратчайшего пути

На рис. 1. изображена схема местности. Передвигаться из пункта в пункт можно только по стрелкам. В каждом пункте можно бывать не более одного раза. Сколькими способами можно попасть из пункта 1 в пункт 9? У какого из этих путей наименьшая длина? У какого наибольшая? Ответить на эти вопросы помогают деревья.

Начиная с вершины 1, последовательно «расслаиваем» граф путей в дерево. При этом каждая вершина столько раз получает самостоятельное значение, сколько в нее в первоначальном графе входило путей (рис. 2). Наикратчайший путь заканчивается в меньшем «ярусе» висячей вершины дерева, самый длинный путь заканчивается в наибольшем «ярусе» (ярусы отмечены на рисунке штриховыми линиями).

Число путей равно числу висячих вершин дерева, то есть 14. Длина кратчайшего пути (1,5,9) равна 2. Длина наиболее продолжительного пути равна 7. Длину пути помогает определить размещение каждой вершины дерева в соответствующем ярусе.

Этот пример, кстати, показывает, что понятие «длина пути» в теории графов не обязательно совпадает с понятием «длина пути» в геометрии или географии.

Рисунок дерева полезен не только тем, что позволяет подсчитать число всех возможных путей, отыскать среди них кратчайший и наиболее протяженный. Он позволяет еще одновременно «увидеть» все пути и сравнить их.

Вероятность и генетика

Приведем примеры использования деревьев в генетике. С помощью дерева можно наглядно представить наследование пары генов \bar{G} и g , передаваемых родителями. Потомок получает эти гены в одной из комбинаций: $\bar{G}\bar{G}$, $g\bar{g}$ или $\bar{G}g$. Генетически комбинация $\bar{G}g$ не отличается от комбинации $g\bar{G}$.

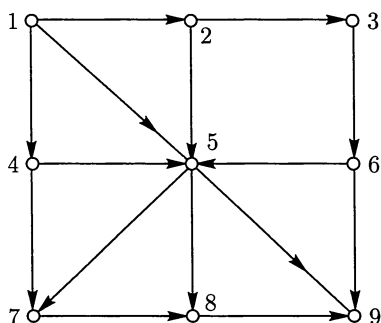


Рис. 1

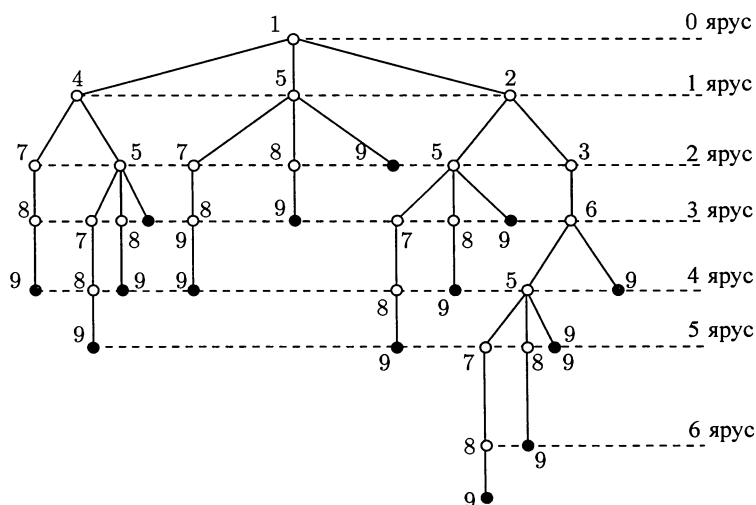


Рис. 2

В генетике допускается, что наследование данного гена происходит случайно, независимо и с равными вероятностями для всех потомков (у растений, например, их может быть очень много). Пусть ген \bar{G} наследуется (и от отца, и от матери) с вероятностью p , ген g — с вероятностью q . В этом случае отца в смысле унаследования гена можно уподобить, например, одной бросаемой монете, мать — второй (рис. 3). Тогда $p + q = 1$.

Далее будем полагать, что $p = q = \frac{1}{2}$. Заметим, что у таких «генеалогических» деревьев вершины, если они не висячие и не корневые, имеют степень 3.



Граф, описывающий ситуацию, которая нас интересует, в случаях так называемого кровного родства деревом не является — две его вершины «слипаются» (рис. 4).



Рассмотрим один из генов, который C унаследовал от своего отца A . Вероятность того, что A унаследовал этот ген от своего деда D , равна $\frac{1}{2}$.

Вероятность того, что дедушка передал копию того же гена B , также равна $\frac{1}{2}$, и вероятность того, что B передал копию этого гена C , равна $\frac{1}{2}$. Все эти события независимые, и, следовательно, $p_d = (\frac{1}{2})^3 = \frac{1}{8}$.

$$k = \frac{1}{8} + \frac{7}{8}p_0.$$

Рассмотренный пример дает некоторое представление о расчетах, связанных с проблемами сохранения в потомстве желательных признаков прародителей: вывода сортов пшеницы, пород собак, голубей, домашних животных, искусственного восстановления вымирающих пород животных. Все это проблемы разные по роли и значимости, но они имеют общую математическую суть.

Лекция 14. Сетевое планирование и управление

Введение. Сетевой график. Правила построения сетевого графика. Анализ сетевой модели. Определение критического пути. Определение полного резерва времени ненапряженного пути. Формирование временных оценок работ

Ключевые слова: проект, событие, сеть, сетевой график, работа, событие, фиктивная работа, последующее событие, предшествующее событие, предшествующая работа, исходное событие, завершающее событие, события первого ранга, события второго ранга, события $(k - 1)$ -го ранга, направленный граф, резерв времени данного события, полный резерв времени работы, критический путь, критические события, критические работы, ненапряженные пути, полное резервное время ненапряженного пути.

Введение

В нашей стране разработаны системы планирования и управления «СПУ». В основе этих систем лежат сетевые графики. Системы «СПУ» успешно применялись, например, при сооружении ТЭЦ в Лисичанске, Буштырской тепловой электростанции, Челябинского блюминга-автомата «1300», при ремонте мартеновской печи завода «Серп и молот», при реконструкции доменной печи в Запорожстали и так далее.

Сетевая модель была применена в США при создании баллистических ракет «Поларис», предназначенных для оснащения атомных подводных лодок американского военно-морского флота. В сложном комплексе работ при этом участвовало свыше 6000 фирм, работы выполнялись на территории 48 штатов Америки, а сетевой график включал в себя более 10000 событий.

Сетевой график

Всякий намеченный комплекс работ, необходимых для достижения некоторой цели, называют *проектом*. Проект (или комплекс работ) подразделяется на отдельные работы. Каждая отдельная работа, входящая в комплекс (проект), требует затрат времени. Некоторые работы могут выполняться только в определенном порядке. При выполнении комплекса работ всегда можно выделить ряд *событий*, то есть итогов какой-

то деятельности, позволяющих приступить к выполнению следующих работ. Если каждому событию поставить в соответствие вершину графа, а каждой работе — ориентированное ребро, то получится некоторый граф. Он будет отражать последовательность выполнения отдельных работ и наступление событий в едином комплексе. Если над ребрами проставить время, необходимое для завершения соответствующей работы, то получится **сеть**. Изображение такой сети называют сетевым графиком. Сетевым графиком состоит из двух типов основных элементов: работ и событий. **Работа** представляет собой выполнение некоторого мероприятия (например, погрузка боезапаса или переход корабля в пункт базирования). Этот элемент сетевого графика связан с затратой времен и расходом ресурсов. Поэтому работа всегда имеет начало и конец. Кроме того, каждая работа должна иметь определение, раскрывающее ее содержание (например, уяснение боевой задачи, приготовление корабля к походу и так далее).

На сетевом графике работа изображается стрелкой, над которой проставляется ее продолжительность или затрачиваемые ресурсы или то и другое одновременно. Работа, отражающая только зависимость одного мероприятия от другого, называется **фиктивной работой**. Такая работа имеет нулевую продолжительность (или нулевой расход ресурсов) и обозначается пунктирной стрелкой.

Начальная и конечная точки работы, то есть начало и окончание некоторого мероприятия (например, окончание приготовления корабля к бою), называются **событиями**. Следовательно, событие в отличие от работы не является процессом и не сопровождается никакими затратами времени или ресурсов.

Событие, следующее непосредственно за данной работой, называется **последующим событием** по отношению к рассматриваемой работе. **Событие**, непосредственно предшествующее рассматриваемой работе, называется **предшествующим**.

Наименования «предшествующий» и «последующий» относятся также и к работам. Каждая входящая в данное событие **работа** считается **предшествующей** каждой выходящей работе, и наоборот, каждая выходящая **работа** считается **последующей** для каждой входящей.

Из определения отношения «предшествующий—последующий» вытекают свойства сетевого графика.

Во-первых, ни одно событие не может произойти до тех пор, пока не будут закончены все входящие в него работы. Во-вторых, ни одна работа, выходящая из данного события, не может начаться до тех пор, пока не произойдет данное событие. И, наконец, ни одна последующая работа не может начаться раньше, чем будут закончены все предшествующие ей.

Событие обозначается кружком с цифрой внутри, определяющей его номер.

Из всех событий, входящих в планируемый процесс, можно выделить два специфических — событие начала процесса, получившее название *исходного события*, которому присваивается нулевой номер, и событие конца процесса (*завершающее событие*), которому присваивается последний номер. Остальные события нумеруются так, чтобы номер предыдущего события был меньше номера последующего.

Для нумерации событий применяется следующий способ. Вычеркиваются все работы, выходящие из события с номером «0», и просматриваются все события, в которых оканчиваются эти вычеркнутые работы. Среди просмотренных находятся события, которые не имеют входящих в них работ (за исключением уже вычеркнутых). Они называются *событиями первого ранга* и обозначаются (вообще, в произвольном порядке) числами натурального ряда, начиная с единицы (на рис. 1 это событие 1). Затем вычеркиваются все работы, выходящие из событий первого ранга, и среди них находятся события, не имеющие входящих работ (кроме вычеркнутых). Это — события второго ранга, которые нумеруются следующими числами натурального ряда (например, 2 и 3 на рис. 1). Продолав таким способом $(k - 1)$ шаг, определяют *события $(k - 1)$ -го ранга*, и просматривая события, в которых эти работы заканчиваются, выбирают события, не имеющие ни одной входящей в них работы (кроме вычеркнутых). Это события k -го ранга, и нумеруются они последовательными числами натурального ряда, начиная с наименьшего, еще не использованного числа при предыдущей нумерации на $(k - 1)$ -м шаге.

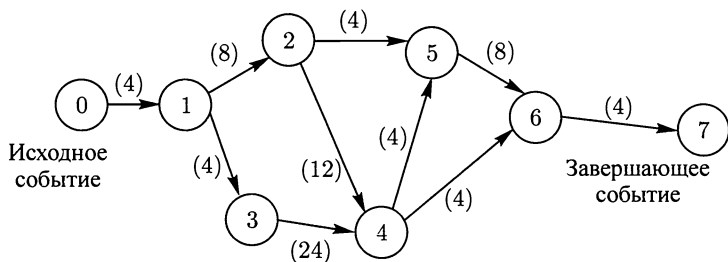


Рис. 1

Сетевой график содержит конечное число событий. Поскольку в процессе вычеркивания движение осуществляется в направлении стрелок (работ), никакое предшествующее событие не может получить номер, больший, чем любое последующее. Всегда найдется хотя бы одно собы-

тие соответствующего ранга, и все события получают номера за конечное число шагов.

Работа обычно кодируется номерами событий, между которыми они заключены, то есть парой (i, j) , где i — номер предшествующего события, j — номер последующего события.

В одно и то же событие могут входить (выходить) одна или несколько работ. Поэтому свершение события зависит от завершения самой длительной из всех входящих в него работ.

Взаимосвязь между работами определяется тем, что начало последующей работы обусловлено окончанием предыдущей. Отсюда следует, что нет работ, не связанных началом и окончанием с другими работами через события.

Последовательные работы и события формируют цепочки (пути), которые ведут от исходного события сетевого графика к завершающему. Например, путь $0 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7$ сетевого графика, показанного на рис. 1, включает в себя события 0, 1, 2, 5, 6, 7 и работы $(0 - 1)$, $(1 - 2)$, $(2 - 5)$, $(5 - 6)$, $(6 - 7)$.

На основании изложенного можно сказать, что ранг события — это максимальное число отдельных работ, входящих в какой-либо из путей, ведущих из нулевого (исходного) события в данное. Так, события первого ранга не имеют путей, состоящих более чем из одной работы, ведущих в них из 0 (например, событие 1 на рис. 1). События второго ранга связаны с 0 путями, которые состоят не более чем из двух работ, причем для каждого события второго ранга хоть один такой путь обязательно существует. Например, на рис. 1 событие 4 — событие третьего ранга, так как пути, ведущие в это событие из 0, включают только три работы — $(0 - 1)$, $(1 - 3)$ и $(3 - 4)$ или $(0 - 1)$, $(1 - 2)$ и $(2 - 4)$.

Построенный таким образом сетевой график в терминах теории графов представляет собой направленный граф.

На рисунке изображен сетевой график. Граф, не содержащий циклов и имеющий только один исток и только один сток, называется **направленным графом**. Сетевой график есть ориентированный связный асимметрический граф с одним истоком, одним стоком и без циклов, то есть это направленный граф. При этом вершинами графа служат события сетевого графика, а дугами (ребрами) — работы сетевого графика.

Продолжительность работы представляет собой, в терминах теории графов, длину дуги. Следовательно, длина пути T — это сумма длин всех дуг, образующих данный путь, то есть $T = \sum t_{i,j}$, $t_{i,j} \in T$, где символом $t_{i,j}$ обозначается дуга, которая соединяет вершины i и j направлена от вершины i к вершине j .

Правила построения сетевого графика

Обычно сетевой график строится от исходного события к завершающему, слева направо, то есть каждое последующее событие изображается несколько правее предыдущего.

В планируемых процессах часто встречаются сложные комплексные связи, когда две работы или более выполняются параллельно, но имеют общее конечное событие, или когда для выполнения одной из работ необходимо предварительно выполнить несколько работ, а для другой, выходящей из общего для них события, предварительным условием является выполнение только одной из предшествующих работ и так далее. Изображение в сетевой модели подобных параллельных или дифференцированно зависимых работ выполняется следующим образом.

В случае, когда наступление события (например, 3 на рис. 2.) возможно в результате завершения двух работ (1 – 3) и (2 – 4), но в то же время существует событие 4 (рис. 2.), зависящее от завершения только одной из этих работ (например, (2 – 4)), вводится фиктивная работа (4 – 3) — рис. 2.

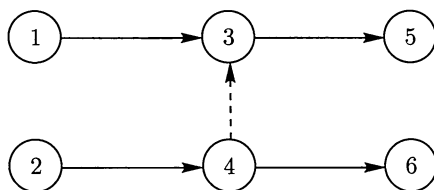


Рис. 2

Если одно событие (например, 1 на рис. 3.) служит началом двух (например, (1 – 2) и (1 – 3) или нескольких работ, заканчивающихся в другом событии (3 на рис. 3)), то для их различия также вводится фиктивная работа (2 – 3) рис. 3. С помощью фиктивной работы в сетевом графике могут быть отражены и двусторонние связи (зависимости).

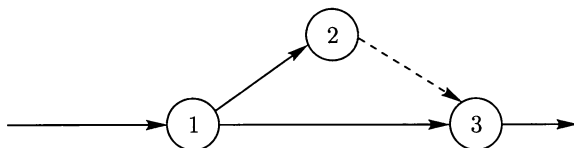


Рис. 3

Пусть, например, имеются три процесса A, B, C . При этом окончание процесса C зависит от результатов процессов A и B . В этом случае возникают двусторонние зависимости, которые можно изобразить так, как показано на рис. 4.

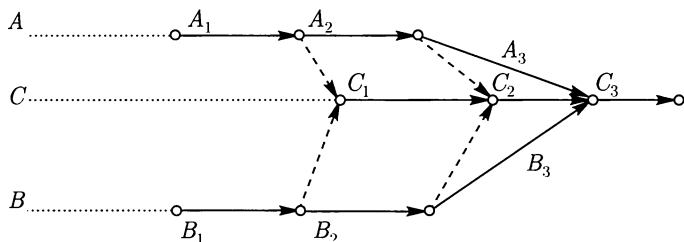


Рис. 4

Другое правило построения сетевого графика заключается в том, что если несколько работ может начаться не после полного, а после частичного выполнения определенной работы, то последнюю работу целесообразно представить как сумму ее частей, расчлененных событиями (1, 2, 3, 4 и 5 на рис. 5). И в то же время группу работ целесообразно представить одной работой, если в этой группе имеется по одному начальному и конечному событию (1 и 4 на рис. 6.).

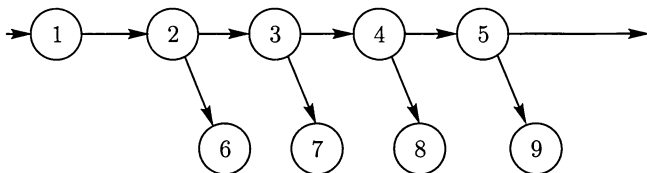


Рис. 5

Для отображения времени и места поступления дополнительных ресурсов (например, пополнение личного состава, топлива и так далее) и другой информации на сетевом графике закрашенным кружком изображаются так называемые подставки (рис. 7). При наличии двух работ и более, выходящих из события, с которым необходимо связать подставку, последняя соединяется с дополнительно введенным событием через фиктивную работу (рис. 7).

После построения сетевого графика проверяется отсутствие работ, имеющих одинаковые коды. При наличии таких работ вводятся дополни-

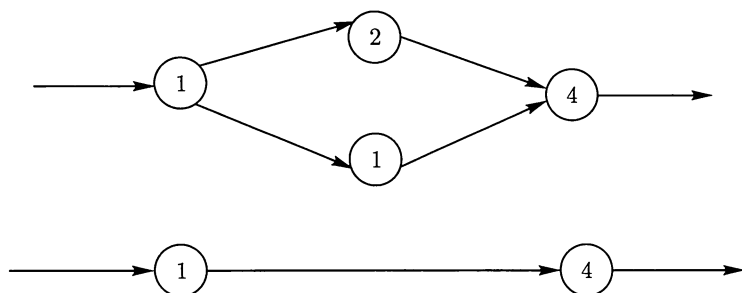


Рис. 6

тельные события и фиктивные работы. Кроме того, сетевой график должен содержать только одно исходное событие и только одно завершающее событие.

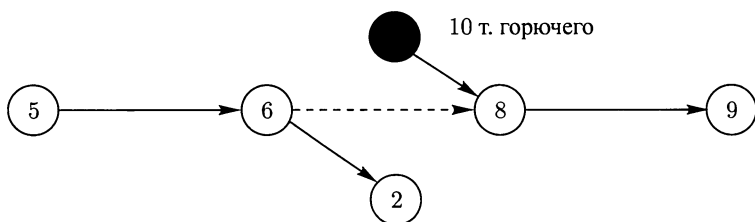


Рис. 7

Если эти условия не выполнены, то необходимо добавить еще одно исходное событие и соединить его стрелками с имеющимися несколькими начальными событиями или добавить еще одно конечное событие, к которому ведут стрелки от нескольких имеющихся конечных событий.

Сетевой график не должен иметь циклов, то есть таких путей, в которых конец последней работы совпадает с началом первой работы. Сетевой график, имеющий хотя бы один цикл, не может быть реализован, так как ни одна из работ, входящих в такой цикл, никогда не может начаться.

Анализ сетевой модели

Параметры сетевой модели. Параметрами сетевой модели являются:

- наиболее раннее возможное время наступления j -го события, обозначаемое символом $T_p(j)$;

- самое позднее допустимое время наступления i -го события, обозначаемое символом $T_n(i)$;
- резерв времени данного события, обозначаемый символом R_i ;
- полный резерв времени работы (i, j) , обозначаемый символом $r_n(i, j)$
- свободный резерв времени работы (i, j) , обозначаемый символом $r_c(i, j)$.

Наиболее раннее возможное время наступления j -го события определяется следующий рекуррентной формулой:

$$T_p(j) = \max_{i \in \Gamma_0^{-1}} \{T_p(i) + t_{ij}\}, \quad (14.1)$$

где t_{ij} — продолжительность (i, j) -й работы; Γ_j^{-1} — множество событий, предшествующих j -му событию.

Вычисления по формуле (14.1) выполняются шаг за шагом, двигаясь в порядке нумерации событий.

Самое позднее допустимое время наступления события i определяется с помощью аналогичной рекуррентной формулы, но обращаясь не к предшествующим, а к последующим событиям.

$$T_n(i) = \min_{j \in \Gamma_i} \{T_n(j) - t_{ij}\}, \quad (14.2)$$

где Γ_i — множество событий, следующих за i -м событием.

Для определения $T_n(i)$ по формуле (14.2) надо двигаться от конечного события n к исходному событию 0, при этом $T_n(n) = T_p(n)$.

Резервом времени данного события называется разность между $T_n(i)$ и $T_p(i)$, которая вычисляется по формуле

$$R_i = T_n(i) - T_p(i). \quad (14.3)$$

Полный резерв времени работы (i, j) вычисляется по формуле

$$r_n(i, j) = T_n(j) - T_p(i) - t_{ij}. \quad (14.4)$$

Свободный резерв времени работы (i, j) вычисляется по формуле

$$r_c(i, j) = T_p(j) - T_n(i) - t_{ij}. \quad (14.5)$$

Определение критического пути

Полный *путь*, суммарная продолжительность работ на котором является максимальной, называется **критическим**, то есть это самый длинный по времени путь в сетевом графике от исходного события до завершающего. Продолжительность критического пути определяет минимальное время, объективно необходимое для выполнения всего комплекса мероприятий, входящих в планируемый процесс. За время, меньше времени критического пути, весь комплекс мероприятий совершиться не может. Поэтому любая задержка на работах критического пути увеличивает время выполнения всего процесса.

События, через которые проходит критический путь, называются **критическими**. **Работы**, входящие в состав критического пути, называются **критическими**.

Задержка в выполнении работы на величину $\Delta t_{ij} > r_n(ij)$ приводит к задержке в наступлении завершающего события на величину $\Delta t_{ij} - r_n(ij)$.

Задержка в выполнении работы на величину $\Delta t_{ij} \leq r_c(ij)$ вообще не повлияет ни на один другой срок, определенный данным сетевым графиком. Следовательно, у критических работ и полные, и свободные резервы времени равны нулю. Вообще говоря, равенство нулю полного резервного времени работы является необходимым и достаточным признаком того, что данная работа критическая. Напротив, свободный резерв времени может быть равным нулю и у некритических работ.

Таким образом, критический путь находится посредством определения работ, полные резервы времени которых равны нулю.

Определение полного резерва времени ненапряженного пути

События и работы, лежащие не на критических путях (такие *пути* называются **ненапряженными**), обладают резервами времени. Выявление этих резервов наравне с определением критического пути составляет основное содержание анализа сетевой модели. С работ и путей, имеющих резервы времени, можно снять ресурсы и направить их на выполнение работ, лежащих на критических путях. Этим самым можно добиться сокращения сроков проведения критических работ, а, следовательно, и всей операции в целом, используя только внутренние резервы.

Полным резервом времени ненапряженного пути называется разница между его длиной и длиной критического пути. Полный резерв времени ненапряженного пути показывает, на сколько в сумме может быть увеличена продолжительность всех работ этого пути без изменения срока вы-

полнения всего процесса в целом. Однако при этом ненапряженный и критический пути не должны пересекаться. Если они пересекаются, то полный резерв времени определяется самым длительным участком напряженного пути, заключенным между соответствующими парами событий критического пути.

Формирование временных оценок работ

Адекватность сетевой модели отображаемому реальному процессу и соответственно оперативность руководства процессом во многом зависят от правильности временных оценок выполняемых работ. Если, например, продолжительность работ будет занижена, то это вызовет поспешность в подготовке всей операции в целом, что в свою очередь может привести к срыву и цель не будет достигнута. А завышение сроков выполнения отдельных работ может привести к потере времени, что также, как правило, ведет к срыву.

Для определения временных и других характеристик, необходимых для оценки длительности работ или расхода ресурсов, могут использоваться статистические данные, полученные опытным путем. Такие оценки однозначно определяются из нормативов. Если такие нормативы отсутствуют, то разработчиками сетевого графика даются три оценки времени:

- оптимистическая (t_{\min});
- пессимистическая (t_{\max});
- наиболее вероятная ($t_{\text{нв}}$).

Оптимистическая оценка — продолжительность работы в наиболее благоприятных условиях.

Пессимистическая оценка — продолжительность работы при самом неблагоприятном стечении обстоятельств.

Наиболее вероятная оценка — продолжительность работы при условии, что не возникнет никаких неожиданных трудностей.

На основании этих оценок вычисляются оценки t_{ij}^c и их дисперсии σ_{ij}^2 по следующим эмпирическим формулам:

$$t_{ij}^c = \frac{t_{\min} + 4t_{\text{нв}} + t_{\max}}{6}; \quad (14.6)$$

$$\sigma_{ij}^2 = \left(\frac{t_{\max} - t_{\min}}{6} \right)^2. \quad (14.7)$$

В этом случае все расчеты проводятся так, как было рассмотрено выше. Затем рассчитываются вероятности того, что полученные параметры сетевой модели (ранние сроки, поздние сроки, резервы и так далее) действительно будут находиться в тех или иных числовых границах. При этом вводится допущение, что продолжительности двух любых работ являются независимыми величинами, а величина t_{ij} определенная формулой (14.6), принимается равной математическому ожиданию продолжительности данной работы (ij). Тогда математическое ожидание любого параметра сетевой модели, являющегося суммой величин вида t_{ij} , есть сумма математических ожиданий слагаемых, то есть $\sum t_{ij}$. Точнее, это оценка снизу, так как все параметры сетевой модели носят, так сказать, экстремальный характер. Соответственно, дисперсия параметра будет $\sum \sigma_{ij}^2$. Если считать, что время выполнения работ подчиняется нормальному закону, вероятность совершения j -го события в расчетный срок можно определить по такой формуле:

$$P_i = \Phi \left(\frac{T_{зд} - T_p}{\sqrt{\sum \sigma_{ij}^2}} \right), \quad (14.8)$$

где Φ — функция Лапласа; $T_{зд}$ — директивный срок; $T_p(j)$ — время раннего свершения j -го события; $\sum \sigma_{ij}^2$ — сумма дисперсий работ, которые использовались при вычислении раннего срока наступления j -го события.

Лекция 15. Паросочетания и свадьбы

Паросочетания и свадьбы. Теорема Холла о свадьбах. Приложение теоремы Холла. Латинские квадраты

Ключевые слова: задачи о свадьбах, двудольный граф, полный двудольный граф, совершенное паросочетание из V_1 в V_2 в двудольном графе $G(V_1, V_2)$, латинский $(m \times n)$ -прямоугольник, латинский квадрат.

Паросочетания и свадьбы

Результаты этой главы носят более *комбинаторный* характер, чем результаты всех предыдущих глав, хотя они тесно связаны с теорией графов. Обсудим хорошо известную «теорему о свадьбах», принадлежащую Филиппу Холлу, и некоторые приложения этой теоремы, например, построение латинских квадратов.

Теорема Холла о свадьбах

Теорема о свадьбах, доказанная Филиппом Холлом в 1935 г., отвечает на следующий вопрос, известный под названием *задачи о свадьбах*: рассмотрим некоторое конечное множество юношей, каждый из которых знаком с несколькими девушками; спрашивается, *при каких условиях можно женить юношей так, чтобы каждый из них женился на знакомой ему девушке?* (Будем считать, что полигамия не разрешена). Например, если имеется четверо юношей $\{b_1, b_2, b_3, b_4\}$ и пять девушек $\{g_1, g_2, g_3, g_4, g_5\}$, а отношения знакомства между ними показаны в таблице 1, то возможно следующее решение: b_1 женится на g_4 , b_2 — на g_1 , b_3 — g_3 , а b_4 — на g_2 .

Эту задачу можно представить графически, взяв двудольный граф G с множеством вершин, разделенными на два непересекающихся подмно-

Таблица 1

Юноша	Девушки, с которыми знаком юноша		
b_1	g_1	g_4	g_5
b_2	g_1		
b_3	g_2	g_3	g_4
b_4	g_2	g_4	

жества V_1, V_2 , представляющих юношей и девушек, соответственно, и соединив ребром каждого юношу со знакомой ему девушкой.

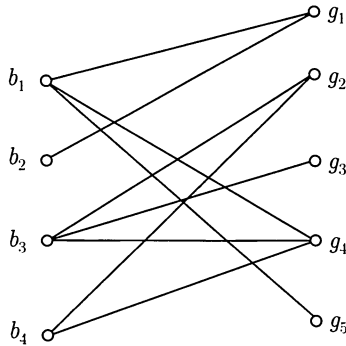


Рис. 1

Напомним определение двудольного графа. Допустим, что множество вершин графа можно разбить на два непересекающихся подмножества V_1 и V_2 так, что каждое ребро в G соединяет какую-нибудь вершину из V_1 с какой-либо вершиной из V_2 , тогда G называем **двудольным графом**. Такие графы иногда обозначают $G(V_1, V_2)$, если хотят выделить два указанных подмножества. Двудольный граф можно определить и по-другому. В терминах раскраски его вершин двумя цветами, скажем, красным и синим. При этом граф называется двудольным, если каждую его вершину можно окрасить красным или синим цветом так, чтобы любое ребро имело один конец красный, а другой — синий. Следует подчеркнуть, что в двудольном графе совсем не обязательно каждая вершина из V_1 соединена с каждой вершиной из V_2 ; если же это так и если при этом граф G простой, то он называется **полным двудольным графом** и обычно обозначается $K_{m,n}$, где m, n — число вершин соответственно в V_1 и V_2 .

Совершенным паросочетанием из V_1 в V_2 в двудольном графе $G(V_1, V_2)$ называется взаимно однозначное соответствие между вершинами из V_1 и подмножеством вершин из V_2 , обладающее тем свойством, что соответствующие вершины соединены ребром. Ясно, что задачу о свадьбах можно выразить в терминах теории графов следующим образом: *если $G = G(V_1, V_2)$ — двудольный граф, то при каких условиях в G существует совершенное паросочетание из V_1 в V_2 ?*

Используя прежнюю «матримониальную» терминологию, можно сформулировать следующее очевидное утверждение: *необходимое условие для существования решения в задаче о свадьбах в том, что любые k юно-*

шей из данного множества должны быть знакомы (в совокупности), по меньшей мере, с k девушками (для всех целых k , удовлетворяющих неравенствам $1 \leq k \leq m$, где через m обозначено общее число юношей). Необходимость этого условия сразу вытекает из того, что если оно не верно для какого-нибудь множества юношей, то мы не сможем женить требуемым способом даже этих k юношей, не говоря уже об остальных.

Поразительно, что это очевидное необходимое условие является в то же время и достаточным. В этом и состоит *теорема Холла о свадьбах*; ввиду ее важности мы приведем три доказательства. Первое из них принадлежит *Халмошу и Вогену*.

Теорема (Ф. Холл 1935). *Решение задачи о свадьбах существует тогда и только тогда, когда любые k юношей из данного множества знакомы в совокупности по меньшей мере с k девушками ($1 \leq k \leq m$).*

Доказательство. Как было отмечено выше, необходимость условия очевидна. Для доказательства достаточности воспользуемся индукцией и допустим, что утверждение справедливо, если число юношей меньше m . (Ясно, что при $m = 1$ теорема верна.) Предположим теперь, что число юношей равно m , и рассмотрим два возможных случая.

(i) Сначала будем считать, что любые k юношей ($1 \leq k \leq m$) в совокупности знакомы по меньшей мере с $k + 1$ девушками (т. е. что наше условие всегда выполняется «с одной лишней девушкой»). Тогда, если взять любого юношу и женить его на любой знакомой ему девушке, для других $m - 1$ юношей останется верным первоначальное условие. По предположению индукции мы можем женить этих $m - 1$ юношей; тем самым доказательство в первом случае завершено.

(ii) Предположим теперь, что имеются k юношей ($k < m$), которые в совокупности знакомы ровно с k девушками. По индуктивному предположению этих k юношей можно женить. Остаются еще $m - k$ юношей, но любые h из них ($1 \leq h \leq m - k$) должны быть знакомы, по меньшей мере, с h девушками из оставшихся, поскольку в противном случае эти h юношей вместе с уже выбранными k юношами будут знакомы меньше, чем с $h + k$ девушками, а это противоречит нашему предположению. Следовательно, для этих $m - k$ юношей выполнено первоначальное условие, и по предположению индукции мы можем их женить так, чтобы каждый был счастлив. Доказательство теоремы закончено. //

Теорему Холла можно также сформулировать на языке паросочетаний в двудольном графе; число элементов множества S обозначается через $|S|$.

Следствие. Пусть $G = G(V_1, V_2)$ — двудольный граф, и для любого подмножества A множества V_1 пусть $\varphi(A)$ — множество тех вершин из V_2 ,

которые смежны, по крайней мере, с одной вершиной из A . Тогда совершенное паросочетание из V_1 в V_2 существует в том и только в том случае, если $|A| \leq |\varphi(A)|$ для каждого подмножества A из V_1 .

Доказательство. Доказательство этого следствия является просто переводом изложенного выше доказательства на языке теории графов.

Приложение теоремы Холла

Рассмотрим приложения теоремы Холла в различных областях.

Латинские квадраты

Латинским $(m \times n)$ -прямоугольником называется $(m \times n)$ -матрица $M = (m_{ij})$, элементами которой являются целые числа, удовлетворяющие условиям (1) $1 \leq m_{ij} \leq n$, (2) все элементы в каждой строке и в каждом столбце различны. Заметим, что из условий (1) и (2) следует, что $m \leq n$; если $m = n$, то латинский прямоугольник называется **латинским квадратом**. К примеру, на рис. 2 и 3 изображены латинский (3×5) -прямоугольник и латинский (5×5) -квадрат. Можно задать следующий вопрос: если дан латинский $(m \times n)$ -прямоугольник, где $m < n$, когда можно присоединить к нему $n - m$ новых строк так, чтобы получился латинский квадрат? Удивительно, что ответ на этот вопрос: «всегда»!

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 5 & 3 \\ 3 & 5 & 2 & 1 & 4 \end{bmatrix}$$

Рис. 2

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 5 & 3 \\ 3 & 5 & 2 & 1 & 4 \\ 4 & 3 & 5 & 2 & 1 \\ 5 & 1 & 4 & 3 & 2 \end{bmatrix}$$

Рис. 3

Латинские квадраты долгое время были известны лишь математикам и любителям головоломок и, в основном, благодаря одной знаменитой задаче Л. Эйлера¹). В 1782 г. Эйлер предложил следующую проблему.

¹ Леонард Эйлер (1707–1783) — один из великих математиков XVIII века, создавших основы математического анализа. Швейцарец по происхождению, он жил и работал преимущественно в России. Эйлер, выделявшийся своей исключительной интуицией и разносторонностью интересов, оставил глубокий след практически во всех областях современной ему математики. Большое количество его замечательных результатов послужило основой для дальнейшего развития многих разделов математики.

Среди 36 офицеров находится по шесть офицеров шести различных званий из шести полков. Можно ли построить этих офицеров в каре так, чтобы в каждой колонне и каждой шеренге встречались офицеры всех званий и всех полков?

Лишь в 1901 г. удалось доказать, что это невозможно. Однако связанные с задачей Эйлера латинские квадраты не потеряли интереса, так как вскоре обнаружилось, что они имеют многообразные практические применения. А в конце шестидесятых годов двадцатого столетия они были применены в теории кодирования. Получающиеся на их основе коды допускают простые алгоритмы декодирования

Лекция 16. Теория трансверсалей

Теория трансверсалей. Приложение теории трансверсалей

Ключевые слова: трансверсаль, система различных представлений, частичная трансверсаль для φ , $(0, 1)$ -матрицы, матрица инцидентий, словарный ранг, общие трансверсали.

Теория трансверсалей

Приводится определения трансверсали. Используя эти понятия, дается еще одно доказательство теоремы Холла. Описывается несколько приложений в лексике трансверсалей.

Если E — непустое конечное множество и $\varphi = (S_1, \dots, S_m)$ — семейство (не обязательно различных) непустых его подмножеств, **трансверсалью** (или **системой различных представлений**) для φ называется подмножество множества E , состоящее из m элементов: по одному из каждого множества S_i .

Общие трансверсали. Если E — непустое конечное множество, а $\varphi = (S_1, \dots, S_m)$ и $\tau = (T_1, \dots, T_m)$ — два семейства его непустых подмножеств, то интересно знать, когда существует общая трансверсаль для φ и τ , то есть множество, состоящее из m различных элементов множества E и являющееся трансверсалью и для φ , и для τ .

Рассмотрим пример. Предположим, что $E = \{1, 2, 3, 4, 5, 6\}$, а $S_1 = S_2 = \{1, 2\}$, $S_3 = S_4 = \{2, 3\}$, $S_5 = \{1, 4, 5, 6\}$. Подсемейство $\varphi' = (S_1, S_2, S_3, S_5)$ имеет трансверсаль, например $\{1, 2, 3, 4\}$. Трансверсаль произвольного подсемейства семейства φ будем называть **частичной трансверсалью для φ** ; в нашем примере семейство φ имеет несколько частичных трансверсалей (например, $\{1, 2, 3, 6\}$, $\{2, 3, 6\}$, $\{1, 5\}$, \emptyset и так далее). Ясно, что любое подмножество частичной трансверсали само является частичной трансверсалью.

Естественно спросить: *при каких условиях данное семейство подмножеств некоторого множества имеет трансверсаль?* Легко увидеть связь между этой задачей и задачей о свадьбах, если взять за E множество девушек, а за S_i — множество девушек, знакомых юноше b_i ($1 \leq i \leq m$); трансверсалью в этом случае является множество из m девушек, такое, что каждому юноше соответствует ровно одна (знакомая ему) девушка. Следовательно, теорема Холла дает необходимое и достаточное условие

существования трансверсали для данного семейства множеств. Сформулируем теорему Холла для этого случая и дадим другое ее доказательство, принадлежащее *P. Радо*.

Теорема. Пусть E — непустое конечное множество и $\varphi = (S_1, \dots, S_m)$ — семейство непустых его подмножеств; тогда φ имеет трансверсаль в том и только в том случае, если для любых k подмножеств S_i их объединение содержит, по меньшей мере, k элементов ($1 \leq k \leq m$).

Доказательство. Необходимость этого условия очевидна. Для доказательства достаточности установим, что если одно из подмножеств (скажем, S_1) содержит более одного элемента, то можно удалить один элемент из S_1 , не нарушив условия теоремы. Повторением этой процедуры мы добьемся сведения задачи к тому случаю, когда каждое подмножество содержит только один элемент. Тогда утверждение станет очевидным.

Осталось обосновать законность этой «процедуры сведения». Предположим, что S_1 содержит элементы x и y , удаление каждого из которых нарушает условие теоремы. Тогда существуют подмножества A и B множества $\{2, 3, \dots, m\}$, обладающие тем свойством, что

$$\left| \bigcup_{j \in A} S_j \bigcup (S_1 - \{x\}) \right| \leq |A|$$

$$\left| \bigcup_{j \in B} S_j \bigcup (S_1 - \{y\}) \right| \leq |B|.$$

Но эти два неравенства приводят к противоречию, поскольку

$$\begin{aligned} |A| + |B| + 1 &= \left| A \bigcup B \right| + \left| A \cap B \right| + 1 \leq \\ &\leq \left| \bigcup_{j \in A \cup B} S_1 \bigcup S_j \right| + \left| \bigcup_{j \in A \cap B} S_j \right| \leq \quad (\text{по условию}) \\ &\leq \left| \bigcup_{j \in A} S_j \bigcup (S_1 - \{x\}) \right| + \left| \bigcup_{j \in B} S_j \bigcup S_1 - \{y\} \right| \leq \\ &\leq \quad (\text{так как } |S_1| \geq 2) \\ &\leq |A| + |B| \quad (\text{по предположению}). \end{aligned}$$

Прелесть этого доказательства в том, что оно проводится, по существу, лишь в один шаг, в отличие от доказательства Халмوشа—Вогена, которое предполагает исследование двух отдельных случаев. (Однако дока-

зательство Радо труднее перевести на весьма наглядный матримониальный язык!).

Следствие. В тех же обозначениях, что и выше, φ имеет частичную трансверсаль мощности t тогда и только тогда, когда для любых k подмножеств S_i их объединение содержит, по меньшей мере, $k + 1 - t$ элементов.

Доказательство. Требуемый результат можно получить, применив теорему Холла в лексике трансверсалей к семейству $\varphi' = S_1 \cup D, \dots, S_m \cup D$, где D — произвольное множество, не пересекающееся с E и состоящее из $m - t$ элементов. Заметим, что φ имеет частичную трансверсаль мощности t тогда и только тогда, когда φ' имеет трансверсаль.

Следствие. Если E и φ такие же, как и прежде, а X — любое подмножество из E , то X содержит частичную трансверсаль мощности t для φ тогда и только тогда, когда для каждого подмножества A множества $\{1, \dots, m\}$

$$\left| \left(\bigcup_{j \in A} S_j \right) \cap X \right| \geq |A| + t - m.$$

Доказательство. Достаточно применить предыдущее следствие к семейству $\varphi_x = (S_1 \cap X, \dots, S_m \cap X)$.

Приложение теории трансверсалей

Используются понятия трансверсалей, на основании чего доказывается теорема о модификации латинского прямоугольника. Вводятся определения $(0, 1)$ -матрицы, формулируются и доказываются теоремы Кенига—Эгервари и об общей трансверсали.

Теорема. Пусть M латинский $m \times n$ -прямоугольник, причем, $m < n$; тогда M можно расширить до латинского квадрата добавлением $n - m$ новых строк.

Доказательство. Докажем, что M можно расширить до латинского $(m + 1) \times n$ -прямоугольника; повторяя эту процедуру, мы придем к латинскому квадрату.

Пусть $E = \{1, 2, \dots, n\}$ и $\varphi = (S_1, \dots, S_n)$, где через S_i обозначено множество, состоящее из тех элементов множества E , которые не встречаются в i -м столбце матрицы M . Если мы сможем доказать, что φ имеет трансверсаль, то тем самым мы докажем теорему, поскольку элементы этой трансверсали и образуют дополнительную строку. По теореме Холла достаточно доказать, что объединение любых k множеств S_i содержит по

меньшей мере k различных элементов. А это очевидно ибо любое такое объединение содержит $(n - m) \times k$ элементов (включая повторения), значит, по крайней мере, один из них повторялся бы более чем $n - m$ раз, что невозможно.

Определение $(0, 1)$ -матрицы или матрицы инцидентий. Другой подход к изучению трансверсалей семейства $\varphi = (S_1, \dots, S_m)$ непустых подмножеств множества $E = \{e_1, \dots, e_n\}$ состоит в исследовании $(m \times n)$ -матрицы $A = (a_{ij})$, в которой $a_{ij} = 1$, если $e_j \in S_i$, и $a_{ij} = 0$ в противном случае. (Любую такую матрицу, все элементы которой равны 0 или 1, мы называем $(0, 1)$ -матрицей) этого семейства.

Определение словарного ранга. Назовем *словарным рангом* матрицы A наибольшее число единиц в A , никакие две из которых не лежат в одной и той же строке или в одном и том же столбце. Тогда φ имеет трансверсаль в том и только в том случае, если словарный ранг матрицы A равен m . Более того, словарный ранг матрицы A равен в точности числу элементов частичной трансверсали, обладающей наибольшей возможной мощностью. В качестве второго приложения теоремы Холла рассмотрим известный результат о $(0, 1)$ -матрицах, называемой теоремой Кенига—Эгервари.

Теорема (Кенига—Эгервари 1931). *Словарный ранг $(0, 1)$ -матрицы A равен минимальному числу μ строк и столбцов, которые в совокупности содержат все единицы из A .*

Замечание. В качестве иллюстрации этой теоремы рассмотрим матрицу

$$\begin{array}{c} e_1 \quad e_2 \quad e_3 \quad e_4 \quad e_5 \quad e_6 \\ \begin{array}{l} S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \end{array} \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}, \end{array}$$

которая является матрицей семейства $\varphi = (S_1, \dots, S_5)$. Ясно, что и ее словарный ранг, и число μ равны четырем.

Доказательство. Очевидно, что словарный ранг не может превосходить числа μ . Чтобы доказать равенство, можно без потери общности предположить, что все единицы из A содержатся в r строках и s столбцах (где $r + s = \mu$) и что строки и столбцы расположены в таком порядке, что в нижнем левом углу матрицы A находится $(m - r) \times (n - s)$ -подматрица, полностью состоящая из нулей.

Если $i \leq r$, то определим S_i как множество целых чисел $j \leq n - s$, таких, что $a_{ij} = 1$. Нетрудно проверить, что объединение любых k множеств S_i содержит по меньшей мере k целых чисел; поэтому семейство $\varphi = (S_1, \dots, S_r)$ имеет трансверсаль. Отсюда следует, что подматрица M из A содержит множество из r единиц, никакие две из которых не принадлежат одной и той же строке или одному и тому же столбцу. Аналогично, матрица N содержит множество из s единиц, обладающих тем же свойством. Таким образом, матрица A содержит множество из $r + s$ единиц, никакие две из которых не принадлежат одной и той же строке или одному и тому же столбцу. Тем самым показано, что μ не превосходит словарного ранга. //

Мы только что доказали теорему Кёнига—Эгервари с помощью теоремы Холла, а доказательство теоремы Холла с помощью теоремы Кёнига—Эгервари и того проще. Следовательно, эти две теоремы в некотором смысле эквивалентны. В лекции 17 мы докажем теорему о максимальном потоке и минимальном разрезе, которая тоже эквивалентна теореме Холла.

Общие трансверсали. Если E — непустое конечное множество, а $\varphi = (S_1, \dots, S_m)$ и $\tau = (T_1, \dots, T_m)$ — два семейства его непустых подмножеств, то интересно знать, когда существует общая трансверсаль для φ и τ , то есть множество, состоящее из m различных элементов множества E и являющееся трансверсалью и для φ , и для τ .

Сформулируем необходимое и достаточное условие для того, чтобы два семейства φ и τ имели общую трансверсаль; заметим, что эта теорема сводится к теореме Холла, если положить $T_j = E$ для $1 \leq j \leq m$.

Теорема. Пусть E — непустое конечное множество, а $\varphi = (S_1, \dots, S_m)$ и $\tau = (T_1, \dots, T_m)$ — два семейства его непустых подмножеств. Тогда φ и τ имеют общую трансверсаль в том и только в том случае, если для всех подмножеств A и B множества $\{1, \dots, m\}$

$$\left| \left(\bigcup_{i \in A} S_i \right) \cap \left(\bigcup_{j \in B} T_j \right) \right| \geq |A| + |B| - m.$$

Набросок доказательства. Рассмотрим семейство $U = \{U_i\}$ подмножеств множества $E \cup \{1, \dots, m\}$ (считаем, что E и $\{1, \dots, m\}$ не пересекаются), где множеством индексов также является $E \cup \{1, \dots, m\}$ и где $U_i = S_i$, если $i \in \{1, \dots, m\}$, и $U_i = \{i\} \cup \{i\} \cup \{j : i \in T_j\}$, если $i \in E$.

Нетрудно проверить, что φ и τ имеют общую трансверсаль тогда и только тогда, когда семейство U имеет трансверсаль. Применяя затем теорему Холла к семейству U , получим нужный результат.

Условия, при которых существует общая трансверсаль для трех семейств непустых подмножеств некоторого множества, пока что не известны, и задача нахождения таких условий кажется очень трудной. Многие попытки решения этой задачи используют теорию матроидов; и действительно, некоторые задачи теории трансверсалей становятся почти тривиальными, если рассматривать их с точки зрения теории матроидов.

Лекция 17. Потоки в сетях

Потоки в сетях. Приложение

Ключевые слова: сеть N , пропускная способность, полустепень исхода $\overleftarrow{\rho}(x)$ вершины x , полустепень захода $\overrightarrow{\rho}(x)$, поток, насыщенная дуга, ненасыщенная дуга, величина потока, максимальный поток, разрез, разрез в сети, пропускная способность разреза, минимальные разрезы.

Потоки в сетях

Деятельность современного общества тесно связана с разного рода сетями — возьмите, к примеру, транспорт, коммуникации, распределение товаров и тому подобное. Поэтому математический анализ таких сетей стал предметом фундаментальной важности.

Определим *сеть* N как оргграф, каждой дуге a которого приписано неотрицательное действительное число $\psi(a)$, называемое ее *пропускной способностью*. Другое определение сети, эквивалентное первому, звучит так: сеть представляет собой пару (D, ψ) , где D — оргграф, а ψ — функция, отображающая множество дуг оргграфа D в множество неотрицательных действительных чисел. *Полустепень исхода* $\overleftarrow{\rho}(x)$ *вершины* x определяется тогда как сумма пропускных способностей дуг вида (x, z) , и аналогичным образом определяется *полустепень захода* $\overrightarrow{\rho}(x)$. Аналог орлеммы о рукопожатиях принимает следующий вид: *сумма полустепеней исхода всех вершин в сети равна сумме их полустепеней захода*. В дальнейшем будем предполагать (если не оговорено иное), что оргграф D содержит ровно один источник v и один сток w ; общий случай, когда имеется несколько источников истоков, легко свести к этому частному.

Для данной сети $N = (D, \psi)$ определим *поток* через N как функцию φ , составляющую каждой дуге a из D неотрицательное действительное число $\varphi(a)$ (называемое потоком через a) таким образом, что $\varphi(a) \leq \psi(a)$ для любой дуги a ; по отношению к сети (D, ψ) полустепень исхода и полустепень захода любой вершины (отличной от v и w) равны между собой. Рассуждая неформально, это означает, что поток через любую дугу не превосходит ее пропускной способности и что «полный поток», входящий в любую вершину (отличную от v и w), равен «полному потоку»,

выходящему из нее. Другим потоком является нулевой поток, при котором поток через каждую дугу равен нулю (любой другой поток называется нулевым).

Для удобства назовем *дугу* a , для которой $\varphi(a) = \psi(a)$, *насыщенной*; остальные дуги называются *ненасыщенными*. Из орлеммы о рукопожатиях следует, что сумма потоков через дуги, инцидентные v , равна сумме потоков через дуги, инцидентные w ; эта сумма называется *величиной потока*. Будем в первую очередь интересоваться потоками, имеющими наибольшую возможную величину, — так называемыми *максимальными потоками*. Заметим, что в общем случае сеть может иметь несколько различных максимальных потоков, однако их величины должны совпадать.

Изучение максимальных потоков через сеть $N = (D, \psi)$ тесно связано с понятием *разреза*, т. е. такого множества A дуг орграфа D , которое обладает тем свойством, что любая простая орцепь из v в w проходит через дугу, принадлежащую A . Другими словами, *разрезом в сети* является не что иное, как vw — разделяющее множество соответствующего орграфа D . *Пропускной способностью разреза* называется сумма пропускных способностей принадлежащих ему дуг. Мы будем рассматривать главным образом такие разрезы, которые обладают наименьшей возможной пропускной способностью, — так называемые *минимальные разрезы*.

Величина любого потока не превышает пропускной способности любого разреза, и, следовательно, величина любого максимального потока не превышает пропускной способности любого минимального разреза. Однако сразу не ясно, что два последних числа всегда равны между собой; этот замечательный результат называется теоремой о максимальном потоке и минимальном разрезе. Впервые она была доказана Фордом и Фалкерсоном в 1955 г. Мы приведем здесь два доказательства; первое из них показывает, что эта теорема по существу эквивалентна теореме Менгера, а второе является прямым доказательством.

Теорема (о максимальном потоке и минимальном разрезе). *Во всякой сети величина любого максимального потока равна пропускной способности любого минимального разреза.*

Первое доказательство. Предположим сначала, что пропускная способность любой дуги является целым числом. В этом случае можно рассматривать сеть как орграф \tilde{D} , в котором пропускные способности представляют число дуг, соединяющих различные вершины. Тогда величина максимального потока соответствует в \tilde{D} полному числу непересекающихся по дугам простых орцепей из v в w , а пропускная способность минимального разреза — минимальному числу дуг в vw — разделяющем множестве. Применяя теперь теорему о целочисленности, мы сразу получим нужный результат.

Чтобы перенести этот результат на сети с рациональными пропускными способностями, умножим все пропускные способности на подходящее целое число d (например, наименьшее общее кратное всех знаменателей), чтобы получились целые числа. Тогда приходим к случаю, описанному в предыдущем абзаце, и нужный результат получаем после деления на d соответствующей величины максимального потока и пропускной способности минимального разреза.

Наконец, если некоторые из пропускных способностей иррациональны, то теорема доказывается с использованием аппроксимации этих чисел рациональными (с любой заданной точностью) и применением предыдущего результата. При этом аппроксимирующие рациональные числа можно подобрать так, чтобы разность между величиной любого максимального потока и пропускной способностью любого минимального разреза можно было сделать сколь угодно малой. Конечно, на практике иррациональные пропускные способности встречаются крайне редко, поскольку обычно пропускные способности задаются в десятичной форме.

Второе доказательство. Теперь приведем прямое доказательство теоремы о максимальном потоке и минимальном разрезе. Заметим, что поскольку величина любого максимального потока не превышает пропускной способности любого минимального разреза, достаточно доказать существование разреза, пропускная способность которого равна величине данного максимального потока.

Пусть φ — максимальный поток. Определим два множества V и W вершин сети: пусть G обозначено основание орграфа D , соответствующего рассматриваемой сети; тогда вершина z сети содержится в V в том и только в том случае, если в G существует простая цепь $v = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{m-1} \rightarrow v_m = z$, обладающая тем свойством, что любое ее ребро $\{v_i, v_{i+1}\}$ соответствует либо ненасыщенной дуге (v_i, v_{i+1}) , либо дуге (v_{i+1}, v_i) , через которую проходит ненулевой поток. (Заметим, что вершина v , очевидно, содержится в V .) Множество W состоит из всех тех вершин, которые не принадлежат V .

Покажем теперь: что W не пусто и, в частности, содержит вершину w . Если это не так, то w принадлежит V , и тогда в G существует простая цепь $v \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_{m-1} \rightarrow w$, обладающая указанным выше свойством. Выберем положительное число ε , удовлетворяющее следующим двум условиям:

- оно не превышает ни одного из чисел, необходимых для насыщения дуг первого типа;
- оно не превышает потока через любую из дуг второго типа.

Очевидно, что если потоки через дуги первого типа увеличить на ε , а потоки через дуги второго типа уменьшить на ε , то величина потока φ увеличится на ε . Но это противоречит нашему предположению о том, что φ — максимальный поток, и, следовательно, w содержится в W .

Для завершения доказательства обозначим через E множество всех дуг вида (x, z) , где x принадлежит V , а z принадлежит W . Ясно, что E является разрезом. Более того, мы видим, что каждая дуга (x, z) из E насыщена, так как в противном случае вершина z также принадлежала бы V . Следовательно, пропускная способность множества E должна равняться величине потока φ , а поэтому E и есть искомый разрез.

Приложение

Теорема о максимальном потоке и минимальном разрезе позволяет проверять, максимален данный поток или нет, но только для достаточно простых сетей. Разумеется, на практике приходится иметь дело с большими и сложными сетями, и в общем случае трудно найти максимальный поток простым подбором. Опишем один алгоритм нахождения максимального потока в любой сети с целочисленными пропускными способностями. Перенесение этого алгоритма на сети с рациональными пропускными способностями осуществляется тривиальным образом и предоставляется читателю.

Шаг 1. Сначала подберем поток φ , обладающий ненулевой величиной (если такой поток существует). Стоит отметить, что чем больше величина выбранного нами начального потока φ , тем проще будут последующие шаги.

Шаг 2. Исходя из N , строим новую сеть N' путем изменения направления потока φ на противоположное. Более точно, любая дуга a , для которой $\varphi(a) = 0$, остается в N' со своей первоначальной пропускной способностью, а любая дуга a , для которой $\varphi(a) \neq 0$, заменяется дугой a с пропускной способностью $\psi(a) - \varphi(a)$ и противоположно направленной дугой с пропускной способностью $\varphi(a)$.

Шаг 3. Если в сети N' мы сможем найти ненулевой поток из v в w , то его можно добавить к первоначальному потоку φ и получить в N новый поток φ' большей величины. Теперь можно повторить шаг 2, используя при построении сети N' новый поток φ' вместо φ . Повторяя эту процедуру, мы в конце концов придем к сети N' , не содержащей ненулевых потоков; тогда соответствующий поток φ будет максимальным потоком.

Часть II

**Комбинаторные алгоритмы
для программистов**

Лекция 1. Комбинаторные вычисления

Введение. Проблема представления: коды, сохраняющие разности. Классы алгоритмов. Анализ алгоритмов. Программа. Вопросы и ответы.

Ключевые слова: эквивалентные объекты, порог, дерево решений, класс алгоритмов, тернарные деревья решений.

Введение

Комбинаторная математика является старой дисциплиной. Она получила свое наименование в 1666 г. от Лейбница в его «Dissertation de Arte Combinatoria». Комбинаторные алгоритмы с их акцентом на разработку, анализ и реализацию практических алгоритмов являются продуктом века вычислительных машин.

Предмет теории комбинаторных алгоритмов — вычисления на дискретных математических структурах. Это новое направление исследований. Лишь в последние несколько лет из наборов искусных приемов и разрозненных алгоритмов сформировалась система знаний о разработке, реализации и анализе алгоритмов.

Комбинаторные вычисления находятся в таком же отношении к комбинаторной математике (дискретной, конечной математике), как численные методы анализа — к анализу. Комбинаторные вычисления развиваются в следующем направлении:

- интенсивно изобретаются новые алгоритмы;
- происходит быстрый прогресс (главным образом в математическом плане) в понимании алгоритмов, их разработки и анализа;
- происходит переход от изучения отдельных алгоритмов к исследованию свойств, присущих классам алгоритмов.

В отличие от некоторых других разделов математики, комбинаторные вычисления не имеют «ядра», то есть некоторого количества «фундаментальных теорем», составляющих суть предмета, из которых выводятся большинство результатов. Сначала может показаться, что в целом эта область состоит из наборов специальных методов и хитрых приемов. Однако после того, как было исследовано достаточно много комбинаторных

алгоритмов, стали вырисовываться некоторые общие принципы. Именно эти принципы делают комбинаторные вычисления связной областью знаний и позволяют изложить ее в систематизированном виде.

Проблема представления: коды, сохраняющие разности

Чрезвычайно важной проблемой в комбинаторных вычислениях является задача эффективного представления объектов, подлежащих обработке. Она возникает потому, что обычно имеется много возможных способов представления сложных объектов более простыми структурами, которые можно заложить в языки программирования, но не все такие представления в одинаковой степени эффективны с точки зрения времени и памяти. Более того, идеальное представление зависит от вида производимых операций.

Приведем примеры, в которых задаются весьма специфические операции над целыми. Целые определяются как данные простейшего типа почти во всех вычислительных устройствах и языках программирования. Таким образом, проблема представления, как правило, не возникает. Имеющееся представление почти всегда наилучшее. Однако существуют некоторые заслуживающие внимания исключения, когда выгодно или даже необходимо использовать представление целых в вычислительном устройстве иным способом. Эти исключения появляются в следующих случаях:

1. Необходимы целые, большие имеющихся непосредственно в аппаратном оборудовании.
2. Необходимы только небольшие целые, и требуется сэкономить память, упаковывая их по несколько в одну ячейку.
3. Действия с целыми производятся не общепринятыми арифметическими операциями.
4. Целые используются для представления других типов объектов, и необходимо иметь возможность легко обращать целое в соответствующий ему объект и обратно.

Проблемы кодов, сохраняющих разности, касаются случаев 2 и 3. В задачах распознавания образов и классификации для решения вопроса, будут ли два *объекта* X, Y *эквивалентными*, стандартной является следующая процедура. X, Y представляются векторами признаков

$(x_1, x_2, \dots, x_f), (y_1, y_2, \dots, y_f)$ соответственно, где каждая компонента означает признак объекта, выраженный целым значением. Считается, что X, Y эквиваленты тогда и только тогда, если $\sum_{i=1}^f |x_i - y_i| \leq t$, где t — целое, называемое порогом.

Классы алгоритмов

Одной из причин быстрого прогресса комбинаторных вычислений является усиление внимания к исследованию классов алгоритмов в противоположность изучению отдельных из них. Для того чтобы утверждать, например, что «все алгоритмы, предназначенные для выполнения того-то и того-то, должны обладать такими-то и такими-то свойствами» или «не существует алгоритма, удовлетворяющего тому-то и тому-то», необходимо иметь дело с четко определенным классом алгоритмов. Именно при таком определении становится возможным говорить, что данный алгоритм является оптимальным по отношению к некоторому свойству, если он работает по крайней мере так же хорошо (относительно этого свойства), как любой другой алгоритм из рассматриваемого класса.

Как можно строго определить, возможно, бесконечный, класс алгоритмов? Исследуем этот вопрос на примере задачи о фальшивой монете. Рассматриваемый в этом примере класс алгоритмов порождает более обширный и более важный класс алгоритмов — так называемые деревья решений.

Задача. Имеется n монет, о которых известно, что $n-1$ из них являются настоящими и не более чем одна монета, фальшивая (легче или тяжелее остальных монет). Дополнительно к группе из n сомнительных монет дается еще одна монета, причем заведомо известно, что она настоящая. Имеются также весы, с помощью которых можно сравнить общий вес любых m монет с общим весом любых других m монет и тем самым установить, имеют ли две группы по m монет одинаковый вес либо одна из групп легче другой. Задача состоит в том, чтобы найти фальшивую монету, если она есть, за наименьшее число взвешиваний, или сравнений.

Решение. Пусть сомнительные монеты занумерованы числами $1, 2, \dots, n$. Монете, о которой известно, что она настоящая, поставим в соответствие номер 0 . Пусть $\{0, 1, 2, \dots, n\}$ — множество монет. Если S_1, S_2 — непересекающиеся непустые подмножества множества S , то через $S_1 : S_2$ обозначим операцию сравнения весов множества S_1, S_2 . При сравнении возможны три исхода, которые обозначим следующим образом: $S_1 < S_2, S_1 = S_2, S_1 > S_2$ в зависимости от того, является ли вес S_1 меньшим, равным или большим веса S_2 .

Рассматриваемые алгоритмы можно представить в форме *дерева решений*.

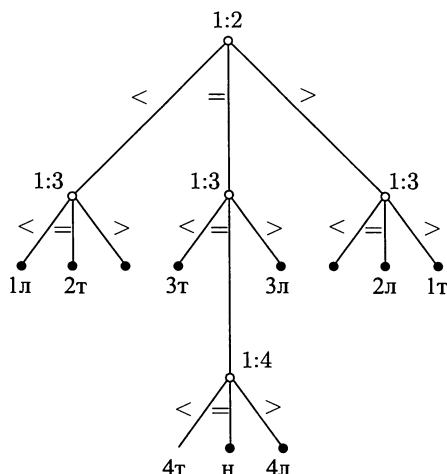


Рис. 1.1. Дерево решений для задачи о фальшивой монете с четырьмя монетами

Корень дерева на рисунке изображен полый окружностью и помечен отношением 1 : 2. Это означает, что алгоритм начинает работу сравнением весов монет с номерами 1 и 2. Три исходящие из корня ветви ведут к поддеревьям, определяющим продолжение работы алгоритма после каждого из трех возможных исходов первого сравнения. Окружности, залитые черной краской, называются листьями дерева и означают, что работа алгоритма заканчивается. Метки соответствуют исходам: «1л» — монета 1 легкая, «1т» — монета 1 тяжелая, «н» — все монеты настоящие. Непомеченная вершина дерева означает, что при наших предположениях этот случай возникнуть не может.

Алгоритм, приведенный на рис. 1.1, требует двух сравнений в одних случаях и трех — в других. Скажем, что он требует «трех сравнений в худшем случае». Обычно важно знать, сколько работы требует алгоритм в среднем, однако для этого требуется задать вероятности различных исходов. Если предположим, что все девять исходов 1л, 1т, 2л, 2т, 3л, 3т, 4л, 4т, — равновероятны, то тогда этот алгоритм требует в среднем $7/3$ сравнений.

На одну чашку весов можем положить больше одной монеты. Например, можно начать сравнения, положив на одну чашку весов монеты 1 и 2, а на другую — монеты 3 и 4 (рис. 1.2).

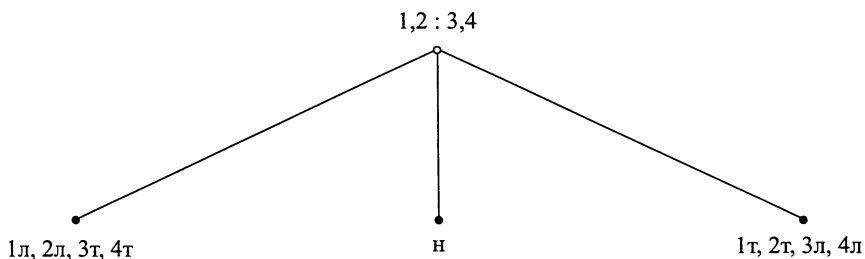


Рис. 1.2. Корень другого дерева решений для задачи о четырех монетах

Если посчастливится, задачу можно решить за одно сравнение — это может произойти, когда все монеты настоящие. Независимо от того, как дополняется это дерево решений, в худшем случае задача все равно потребует тех же трех сравнений, поскольку единственное тернарное решение не может идентифицировать один из четырех исходов, которые возможны на ветви, помеченной символом «<», так же как и один из четырех исходов на ветви, помеченной символом «>». К тому же, независимо от того, как дополняется это дерево решений, оно потребует в среднем по крайней мере 7,3 сравнений, и в этом случае оно не лучше, чем дерево на рис. 1.1.

Используя монету 0, о которой известно, что она настоящая, можно получить приведенное на рис. 1.3 дерево решений (полное двухъярусное тернарное дерево), которое и в худшем, и в среднем случае требует двух сравнений.

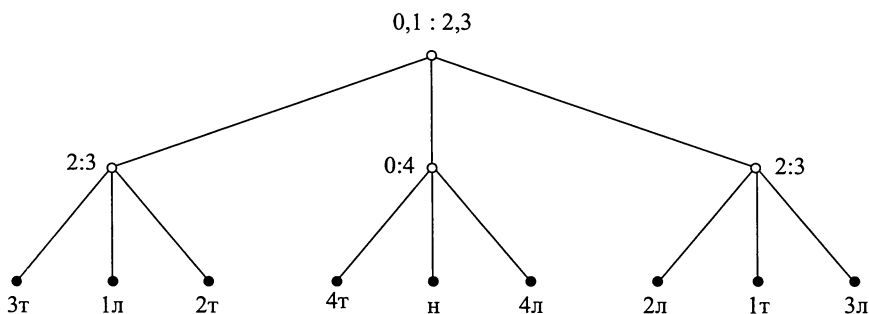


Рис. 1.3. Оптимальное дерево решений для задачи о четырех монетах

Рассматриваемый *класс алгоритмов* решения задачи о фальшивой монете есть множество *тернарных деревьев решений* (примеры на рис. 1.1, рис. 1.2, рис. 1.3), обладающих следующими свойствами:

— каждый узел помечен сравнением $S_1 : S_2$, где S_1 и S_2 — непересекающиеся непустые подмножества множества $S = \{0, 1, 2, \dots, n\}$ всех монет;

— каждый лист либо не помечен, что соответствует невозможному исходу в предположении существования не более чем одной фальшивой монеты, либо помечен одним из исходов il, iT, n , означающим соответственно, что все монеты настоящие. Четко определив подлежащий дальнейшему рассмотрению класс алгоритмов, можно исследовать свойства, которыми должно обладать каждое дерево из этого класса, и определить, как найти алгоритмы, являющиеся в некотором смысле оптимальными. Решим эту проблему в начале для четырех монет, а затем перейдем к общему случаю.

Поскольку в задаче о четырех монетах требуется различить девять возможных исходов, любое дерево решений для этой задачи должно иметь, по крайней мере, девять листьев и, следовательно, не менее двух ярусов. Поэтому дерево на рис. 1.3 является оптимальным и для худшего случая, и для среднего. Существуют ли другие оптимальные деревья? Для ответа на этот вопрос нужно рассмотреть множество всех деревьев решений для задачи о четырех монетах. Попытаемся исключить из дальнейшего рассмотрения какую-либо часть этого множества. Прежде всего видно, что путем любой перестановки множества $\{1, 2, \dots, n\}$ сомнительных монет из одного дерева, приведенного на рис. 1.3, можно получить другие оптимальные деревья. Все они будут изоморфны дереву на рис. 1.3. Исходя из этого, уточним постановку задачи и будем интересоваться попарно неизоморфными деревьями.

Рассмотрим затем, существует ли оптимальное дерево среди тех, у которых в корне не используется монета с номером n . При таком ограничении в корне дерева можно сделать только два различных сравнения, а именно $1 : 2$ и $1, 2 : 3, 4$. Рассмотрим разбиение исходов по трем ветвям, выходящим из корня, как показано на рис. 1.2. Для получения такого, как на рис. 1.3, полного двухъярусного тернарного дерева, девять возможных исходов должны были бы быть разбиты в отношении $(3, 3, 3)$. Они же вместо этого разбиваются, соответственно, в отношении $(2, 5, 2)$ и $(4, 1, 4)$. Таким образом, заключаем, что задачу для четырех монет нельзя решить за два сравнения, не используя дополнительную настоящую монету.

Наконец, рассмотрим те деревья решений, которые используют монету 0 в корне. В этом случае видно, что в корне фактически возможны только два сравнения: $(0 : 1)$ и $0, 1 : 2, 3$. Для первого сравнения набор исходов будет $(1, 7, 1)$, в связи с чем все алгоритмы, начинающиеся таким способом, для нас непригодны. Набор исходов $(3, 3, 3)$ приводит к оптимальному дереву, показанному на рис. 1.3. Аналогичным образом уста-

навливается, что для оптимального дерева сравнения в первом от корня ярусе определяются единственным образом. Отсюда заключаем, что для задачи о четырех монетах фактически существует только одно оптимальное дерево.

Когда используемые идеи анализа задачи о четырех монетах переносятся на произвольный случай, в некоторой степени все идеи обобщаются на случай любого числа монет. Однако некоторые из них не имеют практического значения, когда n значительно больше четырех. В принципе, оптимальные деревья решений всегда можно найти путем систематического поиска в множестве деревьев, поскольку для любого заданного n в качестве кандидатов требуется рассмотреть лишь конечное число деревьев решений. В лекции обсуждается техника исчерпывающего поиска в таких конечных множествах. Однако, если даже поиск организован разумно и рассматриваются лишь существенно различные (не изоморфные) деревья, эта процедура не может служить практическим способом отыскания оптимальных деревьев решений. С ростом n число деревьев растет экспоненциально, и поэтому техника исчерпывающего поиска имеет практическое значение только для малых значений n .

Поскольку число листьев в дереве решений должно быть по крайней мере таким же, как и число возможных исходов задачи ($2n + 1$ для задачи о монетах), сразу же можно получить нижнюю оценку необходимого числа сравнений (или, что эквивалентно, верхнюю оценку числа монет для данного сравнения).

Анализ алгоритмов

В процессе разработки и реализации алгоритма естественным образом раскрываются некоторые его свойства. По мере того как алгоритмы становятся все более и более сложными, все менее и менее вероятно, что их важные свойства проявятся на стадиях разработки и реализации. Как правило, некоторые важные аспекты поведения алгоритма, такие как его корректность, необходимое число операций или объем памяти, определить трудно. Поэтому обычно глубокое понимание нового алгоритма предваряется очень длинной стадией его анализа.

Из-за трудностей анализа им зачастую просто пренебрегают. Вместо этого программа выполняется для того, чтобы увидеть, что получается (например, измеряется время работы). Такой подход можно признать удовлетворительным, если есть основание полагать, что тестовые задачи достаточно хорошо характеризуют работу алгоритма в общем случае; если же это не так, то описанный подход даст мало ценной информации. Даже если тест прекрасно характеризует работу алгоритма, он никогда не даст

ответ на придиричивый вопрос, могут ли существовать лучшие алгоритмы для решения той же самой задачи. Проблему оптимальности алгоритма можно решить только путем его анализа.

В анализе алгоритмов существуют две фундаментальные проблемы:

1. Какими свойствами обладает данный алгоритм?

2. Какие свойства должен иметь любой алгоритм, решающий данную проблему?

Фундаментальная разница между этими двумя вопросами состоит в подходе к ответу на них. В первом случае алгоритм задан и заключения выводятся путем изучения свойств, присущих ему. Во втором случае задается проблема и точно определяется структура алгоритма. Заключения выводятся на основе изучения существа проблемы по отношению к данному классу алгоритмов.

Программа

Программа 1. Поиск фальшивой монеты

//Монета ищется простым перебором.

//Алгоритм реализован на языке программирования Turbo-C++.

```
#include<stdlib.h>
```

```
#include<conio.h>
```

```
#include<stdio.h>
```

```
int main() {
    clrscr();
    int k, n, flag = -1;
    int *mas;
    printf("Введите число монет: ");
    scanf("%i", &n);
    mas = (int*)malloc(sizeof(n));
    randomize();
    k = random (10) + 1;
    for (int i = 0; i < n; i++) {
        mas[i] = k;
    }
    for (i = k; i == k; i = random (20));
    mas[random (n)] = i;
    printf("Масса монет: ");
    for (i = 0; i < n; i++) {
        printf("%5i", mas[i]);
        if (mas[i] != k) {
            flag = i;

```

```
    }  
}  
printf("\nФальшивая монета под номером %i ее вес %i", flag+1,  
mas[flag]);  
getch();  
free(mas);  
return 0;  
}
```

Лекция 2. Целые и последовательности (последовательное распределение)

Введение. Целые. Последовательности. Различные способы представлений конечных последовательностей (или начальных сегментов бесконечных последовательностей) и операции над ними. Вопросы и ответы.

Ключевые слова: система счисления, основание, смешанные системы счисления, бесконечная последовательность, конечная последовательность, список, начальные сегменты бесконечных последовательностей, построчная запись матрицы, характеристический вектор.

Введение

Большинство вычислительных устройств в качестве основных объектов допускает только двоичные наборы, целые и символы, поэтому, прежде чем работать с более сложными объектами, их необходимо представить двоичными наборами, целыми или символами. Например, числа с плавающей запятой кодируются целыми — мантиссой и порядком этого числа, но такое кодирование обычно незаметно для пользователя. В противоположность этому, рассмотренные во второй и третьей лекции способы кодирования объектов (множества, последовательности, деревья) всегда адресованы пользователю.

Любой заданный класс объектов может иметь несколько возможных представлений, и выбор наилучшего из них решающим образом зависит от того, каким образом объект будет использован, а также от типа производимых над ним операций. Поэтому рассмотрим не только свойства самих представлений, но также и некоторые приложения.

Целые

Целые являются основными объектами в вычислительной комбинаторике. В различных вычислительных теоретико-числовых исследованиях изучаются сами целые числа, но мы будем использовать их главным образом при подсчете и индексировании. В последнее время установлено, что полезны различные представления. В этой лекции обсудим общий класс позиционных представлений.

Мы будем рассматривать только неотрицательные целые. Кроме того, к любому представлению неотрицательных целых легко присоединить одиночный знаковый двоичный разряд.

Позиционные системы для представления целых чисел очень широко известны, поскольку они встречаются во многих разделах математики, начиная с «новой математики» и кончая углубленным курсом теории чисел. В *системе счисления с основанием r* каждое положительное целое число имеет единственное представление в виде конечной последовательности

$$(d_0, d_1, d_2, d_3, \dots, d_k), \quad (2.1)$$

в которой каждое d_i — целое, удовлетворяющее условию $0 \leq d_i < r$ и $d_k \neq 0$. Нуль представляется последовательностью (0). r называется *основанием* системы ($r > 1$). Целое, соответствующее последовательности (2.1), имеет вид

$$N = d_0 + d_1 r + d_2 r^2 + d_3 r^3 + \dots + d_k r^k,$$

что принято выражать следующим образом:

$$N = (d_k d_{k-1} \dots d_1 d_0)_r.$$

На протяжении истории использовались различные значения r . Например, древние вавилоняне использовали $r = 60$, а индейцы племени Майя — $r = 20$. Сегодня наиболее широко используется $r = 10$ — десятичная система, которую мы унаследовали от арабов, и $r = 2$ — двоичная система, которая лежит в основе современных вычислительных устройств. В действительности она применяется лишь на самом низком уровне аппаратного оборудования; в сложных вычислительных устройствах и базисных языках удобнее использовать $r = 8$ или $r = 16$.

Единственность этого представления можно доказать методом от противного. Числа $N = 0$ и $N = 1$, очевидно, имеют единственное представление. Предположим, что представление не единственно, и пусть $N > 1$ будет наименьшим целым числом, имеющим два различных представления:

$$N = (d_k d_{k-1} \dots d_0)_r = (e_l e_{l-1} \dots e_0)_r.$$

Если $k \neq l$, то без потери общности предположим, что $k > l$. Тогда, поскольку

$$\sum_{i=0}^l (r-1)r^i = r^{l+1} - 1 < r^{l+1} < r^k$$

и поскольку $d_k \neq 0$, мы заключаем, что

$$(d_k d_{k-1} \dots d_0)_r > (e_l e_{l-1} \dots e_0)_r, \quad (2.2)$$

что невозможно. Таким образом, мы должны иметь $k = l$. Аналогично, если $d_k > e_k$, мы имели бы снова неравенство (2.2) и отсюда с необходимостью $d_k = e_k$. Следовательно, число

$$N - d_k r^k = (d_{k-1} d_{k-2}, \dots, d_0 = (e_{k-1} e_{k-2} \dots e_0)_r$$

имеет два различных представления, что противоречит предположению, что N — наименьшее из таких чисел.

Для доказательства того, что каждое положительное целое имеет представление по основанию r , достаточно задать алгоритм, конструирующий (с необходимостью единственное) представление данного числа N .

Алгоритм 1. Преобразование числа N в его представление $(d_k d_{k-1} \dots d_1 d_0)_r$ в системе счисления с основанием r .

Он строит последовательность $d_0, d_1, d_2, \dots, d_k$ путем повторения деления на r и записи остатков. Пусть на первом шаге при делении N на r остаток будет d_0 . Частное, полученное в результате первого шага, делим на r , вновь полученное частное делим на r и так далее. Полученная в результате такого процесса последовательность остатков и будет требуемым представлением N по основанию r .

Важным обобщением систем счисления с основанием r являются **смешанные системы счисления**, в которых задается не единственное основание r , а последовательность оснований r_0, r_1, r_2, \dots , и последовательность (2.2) соответствует целому

$$N = d_0 + d_1 r_0 + d_2 r_0 r_1 + d_3 r_0 r_1 r_2 + \dots + d_k \prod_{i=0}^{k-1} r_i,$$

где теперь каждое d_i удовлетворяет неравенству $0 \leq d_i < r_i$ и $d_k \neq 0$, если $N \neq 0$ — тот факт, что каждая такая последовательность соответствует единственному числу и каждое положительное целое число имеет единственное представление, следует из простого обобщения результатов для обычных систем счисления, которые являются частным случаем смешанных систем при $r_i = r, i \geq 0$.

Смешанные системы счисления могут вначале показаться странными, но в действительности в повседневной жизни они встречаются почти так же часто, как и десятичные.

Пример. Рассмотрим нашу систему измерения времени: секунды, минуты, часы, дни недели и годы. Это — в точности смешанная система с $r_0 = 60, r_1 = 60, r_2 = 24, r_3 = 7, r_4 = 52$.

Представление целого N в смешанной системе счисления (r_0, r_1, \dots) осуществляется с помощью алгоритма 2, который является простым обоб-

щением алгоритма 1. Вместо того, чтобы для получения d_k в качестве делителя всегда использовалась r , в алгоритме 2 используется r_k .

Алгоритм 2. Преобразование числа N в его представление (d_0, d_1, \dots, d_k) в смешанной системе счисления (r_0, r_1, r_2, \dots) .

Последовательности

Бесконечная последовательность

$$s_1, s_2, s_3, \dots$$

формально определяется как функция f , областью определения которой является множество положительных целых чисел: $f(i) = s_i, i \geq 1$. Во многих случаях индексирование последовательности более удобно начинать с нуля; тогда областью определения f будет множество целых неотрицательных чисел. Аналогично определим **конечную последовательность или список**

$$s_1, s_2, \dots, s_n$$

как функцию, областью определения которой является множество $\{1, 2, \dots, n\}$. Примером бесконечной последовательности являются простые числа

$$i : 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \dots \quad (2.3)$$

$$p_i : 2 \ 3 \ 5 \ 7 \ 11 \ 13 \ 17 \ 19 \ 23 \ 29 \dots,$$

а перестановка

$$\Pi(1, 2, 3, 4, 5, 6) = (6, 2, 5, 1, 3, 4)$$

представляет собой пример конечной последовательности.

В комбинаторных алгоритмах часто приходится встречаться с представлениями конечных последовательностей (или **начальных сегментов бесконечных последовательностей**) и операциями над ними.

Различные способы представлений конечных последовательностей (или начальных сегментов бесконечных последовательностей) и операции над ними

Последовательное распределение. С вычислительной точки зрения простейшим представлением конечной последовательности является список ее членов, расположенных по порядку в последовательных ячейках памяти.

Так, s_1 хранится, начиная с ячейки l_1 , s_2 хранится, начиная с ячейки $l_2 = l_1 + d$, s_3 хранится, начиная с ячейки $l_3 = l_1 + 2d$ и так далее, где d —

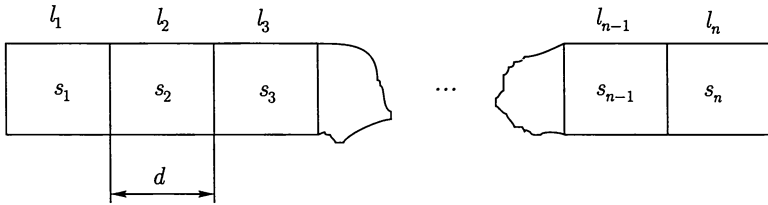


Рис. 2.1. Последовательное распределение последовательности s_1, s_2, \dots, s_n для каждого элемента которой требуется d ячеек

число ячеек, требуемых для хранения одного элемента последовательности.

Описанное выше представление последовательности имеет ряд преимуществ. Во-первых, оно легко осуществимо и требует небольших расходов в смысле памяти. Кроме того, оно полезно и потому, что существует простое соотношение между i и адресом ячейки, в которой хранится s_i :

$$l_i = l_1 + (i - 1)d.$$

Это соотношение позволяет организовать прямой доступ к любому элементу последовательности. Наконец, последовательное представление имеет достаточно широкий диапазон и включает в себя в качестве специального случая представление многомерных массивов.

Например, чтобы представить массив размером $n \times m$

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \cdot & \cdot & \cdot & \cdot \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \quad (2.4)$$

будем рассматривать его как последовательность s_1, s_2, \dots, s_n , в которой каждое s_i в свою очередь является последовательностью из m элементов i -й строки нашей матрицы. Таким образом, число ячеек, требуемых для записи элемента s_i (будем обозначать это число символом \bar{d}), равно $m\bar{d}$, где \bar{d} — число ячеек, требуемых для записи элемента a_{ij} . Поскольку последовательность s_i начинается в ячейке

$$l_i = l_1 + (i - 1)d = l_1 + (i - 1)m\bar{d},$$

ячейка для a_{ij} будет иметь следующий адрес:

$$l_i + (j - 1)\bar{d} = l_1 + [(i - 1)m + (j - 1)]\bar{d}.$$

Это представление известно как *построчная запись матрицы*; постолбцовая запись получается, если (2.4) рассматривать как последовательность t_1, t_2, \dots, t_m в которой каждое t_i в свою очередь является последовательностью из элементов i -го столбца матрицы.

Последовательное распределение, наряду с преимуществами, имеет значительные недостатки. Например, такое представление становится неудобным, если требуется изменить последовательность путем включения новых и исключения имеющихся там элементов. Включение между s_i и s_{i+1} нового элемента требует сдвига $s_{i+1}, s_{i+2}, \dots, s_n$ вправо на одну позицию; аналогично, исключение s_i требует сдвига тех же элементов на одну позицию влево. С точки зрения времени обработки, такое передвижение элементов может оказаться дорогостоящим, и в случае динамических последовательностей лучше использовать технику связанного распределения, рассматриваемую в следующей лекции.

Характеристические векторы. Важной разновидностью последовательного распределения является случай, когда такому представлению подвергается последовательность некоторой основной последовательности $s_1, s_2, s_3 \dots$.

В этом случае можно представить последовательность более удобно, используя *характеристический вектор* — последовательность из нулей и единиц, где i -й разряд равен единице, если s_i принадлежит рассматриваемой последовательности, и нулю в противном случае.

Например, характеристический вектор начального сегмента последовательности (2.3)

$$s_i : \quad 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10$$

характеристический вектор

$$\text{для простых чисел :} \quad 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0$$

Здесь основной последовательностью является последовательность целых положительных чисел. В ЭВМ с 32-разрядными словами для запоминания простых чисел, меньших 10^6 , потребуется $10^6 \cdot 32 = 31250$ слов. Замечая далее, что для $i > 1$ число $2i$ не простое, можно сэкономить половину этого поля, выписывая разряды только для чисел видов $2i+1, i \geq 1$, и запоминая, что простое число 2 отсутствует. Таким образом, простые числа, меньшие чем 10^6 , можно записать только 15625 словами. Поскольку число простых чисел, меньших 10^6 , равно 78498, последовательное представление, описанное ранее, потребовало бы поля в пять раз меньшего размера.

Характеристические векторы полезны в ряде случаев. Их полезность вытекает из их компактности, существования простого фиксированного

соотношения между i и адресом i -го разряда и возможности при таком представлении очень легко исключать элементы.

Главное неудобство характеристических векторов состоит в том, что они не экономичны. Исключение составляют «плотные» последовательности последовательностей $s_1, s_2, s_3 \dots$. Кроме того, их трудно использовать, если не существует простого соотношения между i и s_i . Если такое соотношение сложное, то использование характеристических векторов может быть очень не экономичным в смысле времени обработки. Если последовательности недостаточно плотные, то значительным может оказаться объем памяти. В случае простых чисел между i и s_i имеется простое соотношение: $s_i = i$ (или $s_i = 2i + 1$, если использовать только нечетные числа). Теорема о простых числах утверждает, что число простых чисел, меньших n , приблизительно равно $n / \ln n$; таким образом, простые числа относительно плотно распределены в множестве целых чисел.

Лекция 3. Последовательности (связанное распределение, стеки и очереди)

Связанное распределение. Разновидности связанных списков. Стеки и очередь. Задачи. Программы. Вопросы и ответы.

Ключевые слова: связанное распределение последовательности, связанный список, указатель, ссылка, узел, поле, пустой указатель, нулевой указатель, адрес, место, сбор мусора, циклический список, нециклический список, дважды связанный список, стек, вершина стека, основание стека, очередь, начало очереди, конец очереди.

Связанное распределение

В лекции 11 даны примеры и программные реализации списков, стеков и очередей. Неудобство включения и исключения элементов при последовательном распределении происходит из-за того, что порядок следования элементов задается неявно требованием, чтобы смежные элементы последовательности находились в смежных ячейках памяти. В результате многие элементы последовательности во время включения или исключения должны передвигаться. Если это требование опущено, то можно выполнить операции включения и исключения без того, чтобы передвигать элементы последовательности. При *связанном распределении последовательности (связанном списке)* каждому s_i поставлен в соответствии указатель (ссылка) P_i , отмечающий ячейку, в которой записаны s_{i+1} и P_{i+1} . Существует также указатель P_0 , который указывает начальную ячейку последовательности, то есть ячейку с символами s_1 и P_1 . Все сказанное выше проиллюстрировано на рис. 3.1.

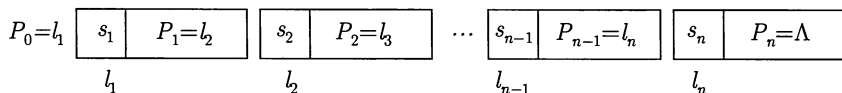


Рис. 3.1. Представление последовательности s_1, s_2, \dots, s_n в виде связанного списка. Каждый элемент списка состоит из поля *INFO* (содержащего элемент последовательности) и поля *LINK* (содержащего адрес следующего элемента)

Здесь каждый *узел* состоит из двух полей. Под узлом понимается одно или несколько последовательных слов в памяти машины, которые выступают как единое целое и разделены на части, именуемые *полями*. В поле *INFO* размещен сам элемент последовательности, а в поле *LINK* — указатель на следующий за ним элемент.

Linked list — список с использованием указателей: список, в котором каждый элемент содержит указатель на следующий элемент или два указателя — на следующий и предыдущий. Поскольку для $s_n = INFO(l_n)$ следующего элемента не существует, будем использовать обозначение $P_n = LINK(l_n) = \Lambda$, где Λ — *пустой*, или *нулевой указатель*. Так как точные значения l_1, l_2, \dots, l_n для программиста не существенны, то в более общем виде связанное представление, показанное на рис. 3.1, можно изобразить так, как показано на рис.3.2.

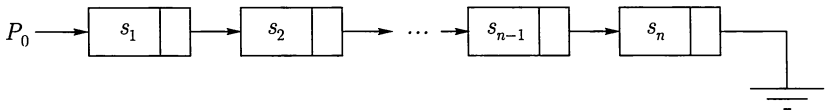


Рис. 3.2. Другой, более употребительный, способ представления списка

Связанное представление последовательностей облегчает операции включения элемента после некоторого s_i и исключения элемента s_{i+1} , если ячейка для s_i известна. Для этого необходимо лишь изменить значения некоторых указателей. Например, чтобы исключить элемент s_2 из последовательности, изображенной на рис. 3.2, необходимо только положить $LINK(l_1) = LINK(l_3)$; после такой операции элемента s_2 в последовательности больше не будет (рис. 3.3). Чтобы в последовательность, изображенную на рис. 3.2, включить новый элемент s_5 , необходимо только воспроизвести новый элемент в некоторой ячейке l_5 с $INFO(l_5) = s_5$ и $LINK(l_5) = LINK(l_1)$ и присвоить $LINK(l_1) \leftarrow l_5$. Это изображено на рис. 3.4.

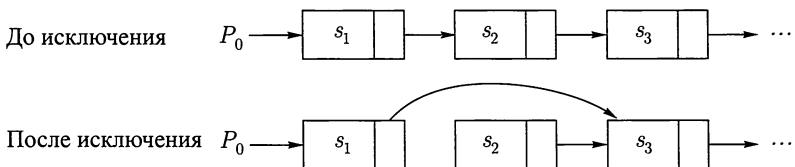


Рис. 3.3. Исключение элемента из связанного списка

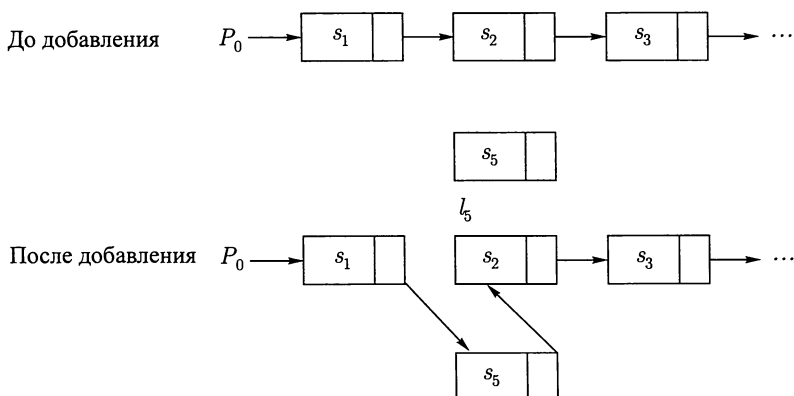


Рис. 3.4. Включение элемента в связанный список

Использование связанного представления предполагает существование некоторого механизма, который по мере надобности предоставляет новые ячейки и собирает старые ячейки, когда они освобождаются, но эта проблема выходит за рамки нашего курса.

Будем предполагать, с целью использования в различных алгоритмах, существование операции порождения ячейки **get_cell**; если эта операция присутствует в правой части оператора присваивания, то она дает *адрес (место)* новой неиспользованной ячейки памяти. Таким образом, чтобы добавить элемент s_5 , как показано на рис. 3.4 нужно фактически использовать оператор **get_cell** для того чтобы найти значение l_5 . Проблему сбора ненужных ячеек памяти будем полностью игнорировать, предполагая лишь, что их каким-то образом собирают для последующего использования; такой процесс носит название *сбора мусора*.

Недостаток при связанном распределении — приходится тратить память на указатели P_i . В приложениях при выборе последовательного или связанного представления нужно сначала проанализировать типы операций, которые будут выполняться над последовательностью. Если операции производятся преимущественно над случайными элементами, осуществляют поиск специфических элементов или производят упорядочение элементов, то обычно лучше применять последовательное распределение. Связанное распределение предпочтительнее, если в значительной степени используются операции включения и(или) исключения элементов, а также сцепления и (или) разбиения последовательностей.

Разновидности связанных списков

Тривиальной модификацией связанного списка, изображенного на рис. 3.2, будет следующее несколько более гибкое представление последовательности: если P_n указывает на s_1 , как показано на рис. 3.5, то мы имеем так называемый **циклический список**. Такая форма списка дает возможность достигнуть любой элемент из любого другого элемента последовательности. Включение и исключение элементов здесь осуществляется так же, как и в **нециклических списках**, в то время как сцепление и разбиение реализуются несколько более сложно.

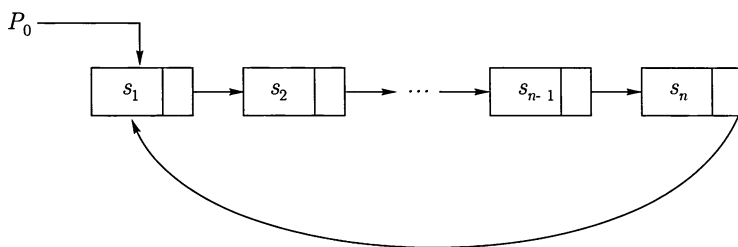


Рис. 3.5. Циклический список

Еще большая гибкость достигается, если использовать **дважды связанный список**, когда каждый элемент s_i последовательности вместо одного имеет два связанных с ним указателя. Как показано на рис. 3.6, они указывают на элементы s_{i-1} и s_{i+1} . В таком списке для любого элемента имеется мгновенный прямой доступ к предыдущему и последующему элементам, в связи с чем облегчаются такие операции, как включение нового элемента перед s_i и исключение элемента s_i без предварительного знания его предшественника. Если есть необходимость, дважды связанный список очевидным образом можно сделать циклическим.

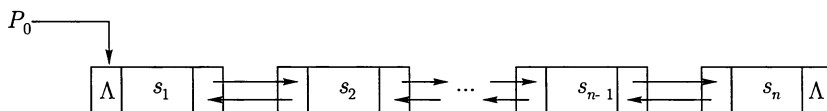


Рис. 3.6. Дважды связанный список

Стеки и очередь

В комбинаторных алгоритмах особую важность представляют две структуры данных, основанные на динамических последовательностях, т.е. последовательностях, которые изменяются вследствие включения новых и исключения имеющихся элементов. В обоих случаях операции включения и исключения, которым подвергается последовательность, имеют ограниченный вид: они производятся только в концах последовательности. **Стек** есть последовательность, у которой все включения и исключения происходят только в ее правом конце, называемом **вершиной стека** (соответственно, левый конец последовательности называется **основанием**). Таким образом, элементы включаются в стек и исключаются из него в соответствии с правилом «*Первым пришел — последним ушел*». **Очередь** — это последовательность, в которой все включения производятся на правом конце списка (**в конце очереди**), в то время как все исключения производятся на левом конце (**в начале очереди**). В противоположность стеку очередь оперирует в режиме «*Первым пришел — первым ушел*».

Стеки и очереди имеют важное значение. Для выполнения какой-либо определенной задачи может потребоваться выполнение ряда подзадач. Каждая подзадача может также привести к другим требующим выполнения подзадачам. И стеки, и очереди являются механизмом, посредством которого запоминаются подзадачи, подлежащие выполнению, а также порядок, в котором они должны быть выполнены. В некоторых случаях порядок таков: «*Первым пришел — последним ушел*»; тогда удобно использовать стеки. Если порядок подчиняется правилу «*Первым пришел — первым ушел*», то подходящим инструментом являются очереди.

Задачи

Задача 1. Создать список, элементами которого являются числа: 1 2 3 4 5 6 7 8 9. Вывести список на экран терминала. Включить в связанный список элемент 2005 после каждого элемента, который делится на 3. Модифицированный список вывести на экран терминала.

Задача 2. Очередью с приоритетом называется линейный список, который оперирует в режиме «первым включается — с высшим приоритетом исключается»; иными словами, каждому элементу очереди сопоставлено некоторое число — приоритет. Включения производятся в конец очереди, а исключения — в любом месте очереди, поскольку исключаемый элемент — это всегда элемент с высшим приоритетом. Нужно описать алгоритм (и его реализацию) включения и исключения для очередей с приоритетом.

Программы

Программа 1. Создание списка.

// Алгоритм реализован на языке программирования Turbo-C++.

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>

struct List{int i;
            List*next;
            };
List*head=NULL;

void Hed(int i)
{if(head==NULL){head=new List;
  head->i=1;
  head->next=NULL;
  }else
  {
    struct List*p,*p1;
    p=head;
    while(p->next!=NULL)
      p=p->next;

    p1=new List;
    p1->i=i;
    p1->next=NULL;
    p->next=p1;
  }
}

int s=0;

void Print(List*p)
{printf("%d",p->i);
  if(p->next!=NULL)Print(p->next);
}

void delist()
{List*p;
  while(head!=NULL)
  {p=head;
```

```
head=head->next;
delete(p);
}
}
void Vstavka(int i1,int c)
{List*p=head,*p1;
while(p->i!=i1)
p=p->next;
p1=new List;
p1->i=c;
p1->next=p->next;
p->next=p1;
}
```

```
void main()
{
clrscr();
for(int i=1;i<=10;i++)
Hed(i);
textcolor(12);
Print(head);
textcolor(1);
Vstavka(10,11);
printf("\n");
Print(head);
textcolor(11);
Vstavka(3,12);
printf("\n");
Print(head);
textcolor(14);
Vstavka(5,13);
printf("\n");
Print(head);
delist();
getch();
}
```

Программа 2. Создание стека и работа со стеком.

//Работа со стеком

// Алгоритм реализован на языке программирования Turbo-C++.

```
#include<stdio.h>
```

```
#include<dos.h>
#include<iostream.h>
#include<PROCESS.H>
#include<STDLIB.H>
#include<conio.H>
#define max_size 200
// char s[max_size]; //компоненты стека
int s[max_size];
int next=0; // позиция стека
int Empty()
{
    return next==0;
}
int Full()
{
    return next==max_size;
}
void Push()
{ if (next==max_size)
    {
        cout<<"Ошибка: стек полон"<<endl;}

    else { next++;cout<<"Добавлен"<<endl;
        cout<<"Что поместить в стек?"<<endl;
        cin>>s[next-1];
    }
}
void OUTst()
{int i=0;
if (next==0)
    {
        cout<<"Стек пуст"<<endl;}

    else { for(i=0;i<next;i++)
        cout<<s[i]<<" "<<endl;
    }
}
void Clear()
{
    next=0;
}
```

```
Poz()
{
    return next;
}
void Del()
{
    int a;
    if (next==0) cout<<"Ошибка: стек пуст"<<endl; else
        {next--;cout<<"Удален "<<endl;}
}
void menu(){
    cout<<"0: распечатать стек"<<endl;
    cout<<"1: добавить в стек"<<endl;
    cout<<"2: удалить из стека"<<endl;
    cout<<"3: узнать номер позиции в стеке"<<endl;
    cout<<"4: узнать, пуст ли стек"<<endl;
    cout<<"5: узнать, полон ли стек"<<endl;
    cout<<"6: очистить стек"<<endl;
    cout<<"7: выход"<<endl;
}
main()
{
    char c;
    clrscr();
    textcolor(11);
    do {
        menu();
        cin>>c;
        clrscr();
        switch (c) {
            case '0':OUTst();getch();break;
            case '1':Push();break;
            case '2':Del();getch();break;
            case '3':cout<<"Номер "<<Poz()<<endl;getch();break;
            case '4':if (Empty()==1) cout<<"Пуст"<<endl; else cout<<"Не
                пуст"<<endl;getch();break;
            case '5':if (Full()==1)cout<<"Полн"<<endl; else cout<<"Не
                полон"<<endl;getch();break;
            case '6':Clear();cout<<"Стек очищен"<<endl;getch();break;
            case '7':exit(1);
        }
    }
```

```
    delay(200);  
  }  
  while (c!=7);  
return 0;  
}
```

Лекция 4. Последовательности (деревья)

Деревья. Представления. Прохождения. Длина путей. Задача. Программа. Вопросы и ответы.

Ключевые слова: конечное корневое дерево, корень дерева, листья, внутренние узлы, потомки корня, предок, отец, сыновья, лес, бинарные деревья, пустое бинарное дерево, прохождение в глубину, прохождение в прямом порядке, прохождение снизу вверх, обратный порядок, конечный порядок, симметричный порядок, лексикографический порядок, внутренний порядок, горизонтальный порядок прохождения, уровень узлов, высота дерева.

Деревья

Конечное корневое дерево T формально определяется как непустое множество упорядоченных узлов, таких, что существует один выделенный узел, называемый *корнем дерева*, а оставшиеся узлы разбиты на $m \geq 0$ поддеревьев T_1, T_2, \dots, T_m . Будем рассматривать только корневые деревья. Узлы, не имеющие поддеревьев, называются *листьями*; остальные узлы называются *внутренними узлами*.

В первой лекции уже было использовано дерево — при изучении необходимого числа взвешиваний в задаче о фальшивой монете с n монетами. Так «рано» деревья появились в тексте не случайно, поскольку понятие дерева используется в различных важных аспектах данного курса. Посредством деревьев изображаются иерархические организации, поэтому они являются наиболее важными нелинейными структурами в комбинаторных алгоритмах.

В описании соотношений между узлами дерева используем терминологию, принятую в генеалогических деревьях. Так, говорят, что в дереве или поддереве все узлы являются *потомками* его корня, и наоборот, корень есть *предок* всех своих потомков. Корень именуют *отцом* корней его поддеревьев, которые в свою очередь будут *сыновьями* корня. Например, на рис. 4.1 узел A является отцом узлов B, G и I ; J, K — сыновья I , а C, E, F — братья.

Все рассматриваемые нами деревья будут упорядочены, то есть для них будет важен относительный порядок поддеревьев каждого узла. Таким образом, деревья считаются различными.

Определим *лес* как упорядоченное множество деревьев; в связи с этим можно перефразировать определение дерева: *дерево есть непустое*

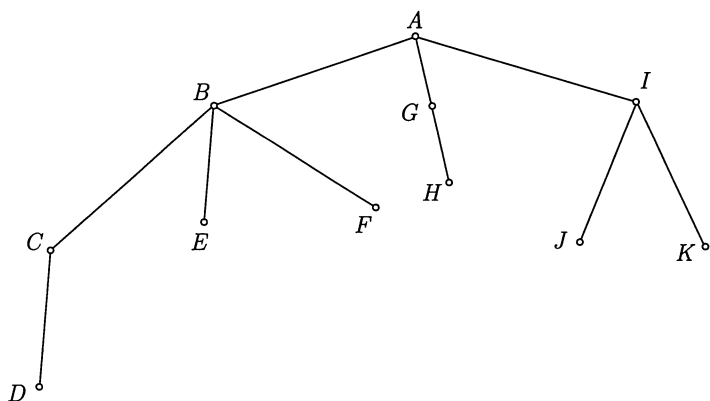


Рис. 4.1. Дерево с 11 узлами, помеченными буквами от A до K . Узлы с метками D, E, F, H, J, K являются листьями; другие узлы внутренние. Узел с меткой A — корень

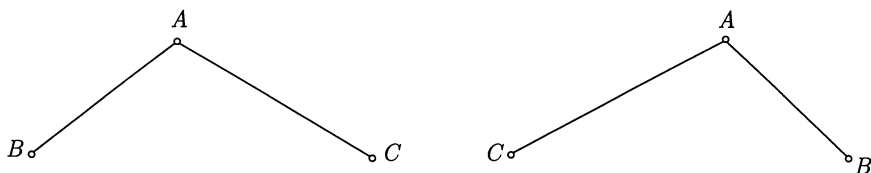


Рис. 4.2. Различные деревья

множество узлов, такое, что существует один выделенный узел, называемый *корнем дерева*, а оставшиеся узлы образуют лес с $m \geq 0$ поддеревьями корня. Важной разновидностью корневых деревьев является класс **бинарных деревьев**. **Бинарное дерево** T либо *пустое*, либо состоит из выделенного узла, называемого корнем, и двух бинарных поддеревьев: левого T_l и правого T_r . Бинарные деревья не являются подмножеством множества деревьев, они полностью отличаются по своей структуре, поскольку два следующих рисунка не изображают одно и то же бинарное дерево.

Как деревья, однако, они не отличаются от дерева, изображенного на рис. 4.4.

Различие между деревом и бинарным деревом состоит в том, что дерево *не может быть пустым*, а каждый узел дерева может иметь произ-

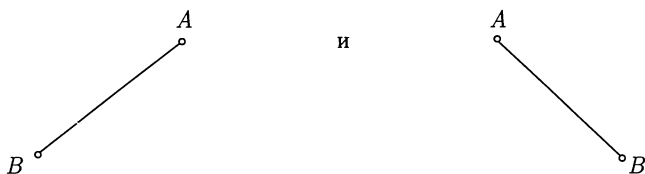


Рис. 4.3. Различные бинарные деревья



Рис. 4.4. Не бинарное дерево

вольное число поддеревьев; в то же время, бинарное дерево *может быть пустым*. Каждая из вершин бинарного дерева может иметь 0, 1 или 2 поддерева, и существует различие между левым и правым поддеревьями.

Представления

Почти все машинные представления деревьев основаны на связанных распределениях. Каждый узел состоит из поля *INFO* и нескольких полей для указателей. Например, представление, которое будет удобным для изложения множества и мультимножества, для каждого узла имеет единственное поле для указателя *FATHER*, указывающего на отца данного узла. При этом приведенное на рис. 4.1 дерево будет выглядеть так, как показано на рис. 4.6. Такое представление полезно, если необходимо подниматься по дереву от потомков к предкам. Такая операция встречается довольно редко. Чаще требуется опуститься по дереву от предков к потомкам.

Представление дерева (или леса) с использованием указателей, ведущих от предков к потомкам, довольно сложно, поскольку узел, имея не более чем одного отца, может в то же время иметь произвольно много сыновей. Другими словами, при таком представлении узлы должны различаться по размеру, что является определенным неудобством. Один из пу-

тей обхода этой трудности состоит в том, чтобы определить соответствие между деревьями и бинарными деревьями, поскольку бинарные деревья легко представить узлами фиксированного размера.

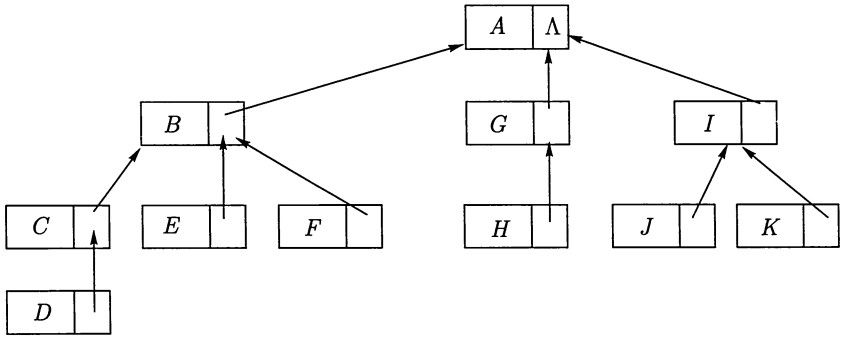


Рис. 4.5. Дерево из рис. 4.1, представленное с помощью узлов с полем *INFO* и указателем *FATHER*

Каждый узел в этом случае имеет три поля: *LEFT*, указатель местоположения корня левого поддерева, *INFO*, содержимое узла, и *RIGHT*, указатель местоположения корня правого поддерева. Все сказанное выше проиллюстрировано на рис. 4.6.

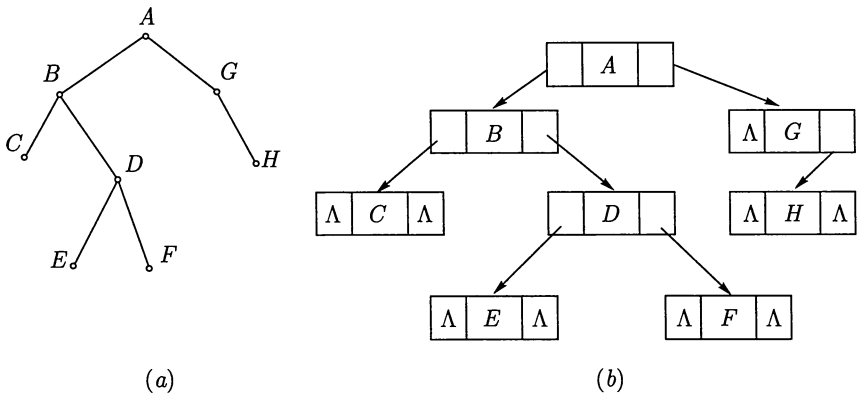


Рис. 4.6. Бинарное дерево и его представление с помощью узлов с тремя полями *LEFT*, *INFO*, *RIGHT*

Можно представлять деревья как бинарные, используя узлы фиксированного размера, представляя каждый узел леса в виде узла, состоящего из полей *LEFT*, *INFO*, *RIGHT*. При этом *LEFT* предназначается для указания самого левого сына данного узла, а поле *RIGHT* — для указания следующего брата данного узла.

Прохождения

Во многих приложениях необходимо пройти лес, заходя в узлы, то есть обрабатывая их некоторым систематическим образом. Посещение каждого узла может быть связано с простой операцией, такой как печать содержимого, или со сложной, такой как вычисление функции. Будем предполагать, что при посещении узла структура леса не меняется. Рассмотрим четыре основных способа прохождения леса: в глубину, снизу вверх, в горизонтальном порядке и для бинарных деревьев — в симметричном порядке.

При **прохождении в глубину**, известном также как **прохождение в прямом порядке**, узлы леса проходятся в соответствии со следующей рекурсивной процедурой:

1. Посетить корень первого дерева.
2. пройти в глубину поддерева первого дерева (если оно есть).
3. Пройти в глубину оставшиеся деревья, если они есть.

Например, для леса, показанного на рис. 4.7., узлы будут проходиться в следующем порядке: *A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S*.

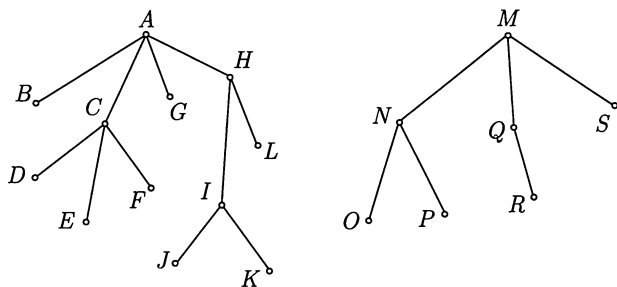


Рис. 4.7. Лес

Название «в глубину» отражает тот факт, что после посещения некоторого узла мы продолжаем прохождение в глубь дерева всякий раз, когда это возможно. Такой порядок особенно полезен в процедурах поиска.

Для бинарных деревьев эта процедура упрощается и выглядит следующим образом.

1. Посетить корень.
2. Пройти в глубину левое поддерево
3. Пройти в глубину правое поддерево.

Прохождение снизу вверх, известное также как обратный порядок или концевой порядок, осуществляется согласно следующей рекурсивной процедуре:

1. Пройти снизу вверх поддерева первого дерева, если они есть.
2. Посетить корень первого дерева.
3. Пройти снизу вверх оставшиеся деревья, если они есть.

Название «снизу вверх» связано с тем, что в момент посещения произвольного узла его потомки оказываются уже пройденными. Такой порядок прохождения полезен, в частности, потому, что он позволяет вычислять рекурсивно определенные функции на лесах. При этом порядке прохождения узлы леса, показанного на рис. 4.8, проходятся в такой последовательности: *B, D, E, F, C, G, J, K, I, L, H, A, O, P, N, R, Q, S, M*. Рекурсивная процедура прохождения снизу вверх применительно к бинарным деревьям имеет следующий вид:

1. Пройти снизу вверх левое дерево.
2. Пройти снизу вверх правое дерево.
3. Посетить корень.

Симметричный порядок для бинарных деревьев определяется рекурсивно следующим образом:

1. Пройти в симметричном порядке левое поддерево.
2. Посетить корень.
3. Пройти в симметричном порядке правое поддерево.

Такой способ прохождения известен также как **лексикографический порядок** или **внутренний порядок**. Заметим, что прохождение леса снизу вверх эквивалентно прохождению в симметричном порядке бинарного дерева, соответствующего этому лесу (при естественном соответствии).

Сравнивая рекурсивные процедуры прохождения бинарных деревьев в глубину, снизу вверх и в симметричном порядке, можно обнаружить их значительное сходство:

<i>прохождение в глубину</i>	<i>симметричный порядок</i>	<i>прохождение снизу вверх</i>
1. посетить корень	1. левое поддерево	1. левое поддерево
2. левое поддерево	2. посетить корень	2. правое поддерево
3. правое поддерево	3. правое поддерево	3. посетить корень

Это сходство позволяет построить общий нерекурсивный алгоритм, который может быть применен к каждому из этих порядков прохождения бинарных деревьев.

Горизонтальный порядок прохождения. При таком способе узлы леса проходятся слева направо, уровень за уровнем от корня вниз. Таким образом, в соответствии с этой процедурой узлы леса, показанного на рис. 4.8, будут проходиться в следующем порядке: $A, M, B, C, G, H, N, Q, S, D, E, F, I, L, O, P, R, J, K$. Такое прохождение дерева полезно в определенных алгоритмах на графах.

Длина путей

Деревья можно использовать не только как способ представления структуры данных, но также как средство для анализа поведения определенных алгоритмов. В связи с этим возникает потребность в количественных измерениях различных характеристик деревьев и, в частности, бинарных деревьев.

Наиболее важные количественные характеристики деревьев связаны с **уровнями узлов**. Уровень p определяется рекурсивно и считается равным нулю, если p корень T ; в противном случае уровень p определяется как $1 + \text{уровень}(FATHER(p))$. Понятие уровня дает возможность определить высоту $h(T)$ дерева T :

$$h(T) = \max_{p \in T} \text{уровня}(p).$$

Другими словами, высота дерева есть максимальное число ребер, образующих путь от корня к листу дерева.

Задача

Задача 1. Построить алгоритм обхода бинарного дерева (см. рис. 4.6 (а)) в глубину.

Программа

Программа 1. Обход бинарного дерева в глубину

```
//Обход ориентированных графов - поиск в глубину -
//обобщение обхода дерева в прямом порядке
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <stdlib.h>

int matr_sm[50][50];
```

```

int mark[50];
int n;

void wrod ()
{int v1,v2;
printf("Введите кол-во вершин в графе: ");
do
{
scanf("%d",&n);
if (n>51) printf("Ошибка!! Введите кол-во вершин в графе: ");
}
} while (n>51);

for (int i=0;i<50;i++) for (int j=0;j<50;j++) matr_sm[i][j]=0;

printf("\nВведите связанные вершины : \n");
do
{ scanf("%d ",&v1);
if (v1>n) //исходящая вершина
{ printf("ОШИБКА ВВОДА !!!!!!!!!");
abort();
}
if (v1==0){break;} // конец ввода
scanf(" %d ",&v2);
if (v2>n) // входящая вершина
{ printf("ОШИБКА ВВОДА !!!!!!!!!");
abort();
}
if (v2==0){break;} //конец ввода
matr_sm[v2-1][v1-1]=1;
} while(1);
}

void vivod()
{
for (int j=0;j<n;j++)
{
for (int i=0;i<n;i++) printf("%d ",matr_sm[j][i]);
printf("\n");
}
}

```

```
}  
}  
  
void dfs(int v)  
{  
    int w;  
    mark[v]=1; //посетили  
    printf("%i ",v+1); //и выдали на экран  
    for (w=0;w<n;w++)  
        if (matr_sm[v][w]==1);  
        else if (mark[w]==0) dfs(w);  
}  
  
void main()  
{ clrscr();  
    wod();  
    // printf("\n"); printf("МАТРИЦА СМЕЖНОСТИ\n");  
    // vivod();  
  
    printf("\n РЕЗУЛЬТАТ ПОИСКА В ГЛУБИНУ \n");  
    for (int v=0;v<50;v++) mark[v]=0;  
    for (v=0;v<n;v++) //если вершина не посещалась, то посетить  
        if (mark[v]==0) dfs(v);  
  
    getch();  
}
```


Лекция 5. Комбинаторика разбиений

Введение. Задачи. Разные статистики. Деревья и перестановки из n элементов. Число сочетаний C_n^m . Задачи на разбиение чисел. Комбинаторные задачи теории информации. Вопросы и ответы.

Ключевые слова: информация, теория информации.

Введение

В задачах, которые мы сейчас рассмотрим, элементы делятся на группы, и надо найти все способы такого раздела. При этом могут встретиться различные случаи. Иногда существенную роль играет порядок элементов в группах: например, когда сигнальщик вывешивает сигнальные флаги на нескольких мачтах, то для него важно не только то, на какой мачте окажется тот или иной флаг, но и то, в каком порядке эти флаги развешиваются. В других же случаях порядок элементов в группах никакой роли не играет. Когда игрок в домино выбирает кости из кучи, ему безразлично, в каком порядке они придут, а важен лишь окончательный результат.

Отличаются задачи и по тому, играет ли роль порядок самих групп. При игре в домино игроки сидят в определенном порядке, и важно не только то, как разделились кости, но и то, кому какие кости достались. Если раскладывать фотографии по одинаковым конвертам, чтобы разослать их, то существенно, как распределяются фотографии по конвертам, но порядок самих конвертов совершенно несущественен.

Играет роль и то, различаем ли мы между собой сами элементы или нет, а также различаем ли между собой группы, на которые делятся элементы. Наконец, в одних задачах некоторые группы могут оказаться пустыми, то есть не содержащими ни одного элемента, а в других такие группы недопустимы. В соответствии со всем сказанным возникает целый ряд различных комбинаторных задач на разбиение.

Задачи

Общая постановка этих задач:

Задач 1. Раскладка по ящикам

Даны n различных предметов и k ящиков. Надо положить в первый ящик n_1 предметов, во второй — n_2 предметов, ..., в k -й — n_k предметов,

где $n_1 + n_2 + \dots + n_k = n$. Сколькими способами можно сделать такое распределение?

Число различных раскладок по ящикам равно

$$P(n_1, n_2, \dots, n_k) = \frac{n!}{n_1! n_2! \dots n_k!}.$$

Эту формулу можно получить при решении следующей, на первый взгляд, совсем непохожей задачи:

Задача 2. Перестановки с повторением.

Имеются предметы k различных типов. Сколько различных перестановок можно сделать из n_1 предметов первого типа, n_2 предметов второго типа, \dots , n_k предметов k -го типа? Число элементов в каждой перестановке равно $n_1 + n_2 + \dots + n_k = n$. Поэтому если бы все элементы были различны, то число перестановок равнялось бы $n!$. Но из-за того, что некоторые элементы совпадают, получится меньшее число перестановок. В самом деле, возьмем, например, перестановку

$$\frac{aa \dots a}{n_1} \frac{bb \dots b}{n_2} \dots \frac{xx \dots x}{n_3}, \quad (5.1)$$

в которой сначала выписаны все элементы первого типа, потом все элементы второго типа, \dots , наконец, все элементы k -го типа. Элементы первого типа можно переставлять друг с другом $n_1!$ способами. Но так как все эти элементы одинаковы, то такие перестановки ничего не меняют. Точно так же ничего не меняют $n_2!$ перестановок элементов второго типа, \dots , $n_k!$ перестановок элементов k -го типа.

Перестановки элементов первого типа, второго типа и так далее можно делать независимо друг от друга. Поэтому элементы перестановки 5.1. можно переставлять друг с другом $n_1! n_2! \dots n_k!$ способами так, что она остается неизменной. То же самое верно и для любого другого расположения элементов. Поэтому множество всех $n!$ перестановок распадается на части, состоящие из $n_1! n_2! \dots n_k!$ одинаковых перестановок каждая. Значит, число различных перестановок с повторениями, которые можно сделать из данных элементов, равно

$$P(n_1, n_2, \dots, n_k) = \frac{n!}{n_1! n_2! \dots n_k!}, \quad (5.2)$$

где $n_1 + n_2 + \dots + n_k = n$.

Пользуясь формулой 5.2, можно ответить на вопрос: сколько перестановок можно сделать из букв слова «Миссисипи»? Здесь у нас одна буква «м», четыре буквы «и», три буквы «с» и одна буква «п», а всего 9

букв. Значит, по формуле 5.2 число перестановок равно

$$P(4, 3, 1, 1) = \frac{9!}{4! \cdot 3! \cdot 1! \cdot 1!} = 2520.$$

Чтобы установить связь между этими задачами, занумеруем все n мест, которые могут занимать наши предметы. Каждой перестановке соответствует распределение номеров мест на k классов. В первый класс попадают номера тех мест, на которые попали предметы первого типа, во второй — номера мест предметов второго типа и так далее. Тем самым устанавливается соответствие между перестановками с повторениями и раскладкой номеров мест по «ящикам». Понятно, что формулы решения задач оказались одинаковыми.

В рассмотренных задачах мы не учитывали порядок, в котором расположены элементы каждой части. В некоторых задачах этот порядок надо учитывать.

Задача 3. Флаги на мачтах.

Имеется n различных сигнальных флагов и k мачт, на которые их вывешивают. Значение сигнала зависит от того, в каком порядке развешены флаги. Сколькими способами можно развесить флаги, если все флаги должны быть использованы, но некоторые из мачт могут оказаться пустыми?

Каждый способ развешивания флагов можно осуществить в два этапа. На первом этапе мы переставляем всеми возможными способами данные n флагов. Это можно сделать $n!$ способами. Затем берем один из способов распределения n одинаковых флагов по k мачтам (число этих способов C_{n+k-1}^{k-1}). Пусть этот способ заключается в том, что на первую мачту надо повесить n_1 флагов, на вторую — n_2 флагов, ..., на k -ю n_k флагов, где $n_1 + n_2 + \dots + n_k = n$. Тогда берем первые n_1 флагов данной последовательности и развешиваем в полученном порядке на первой мачте; следующие n_2 флагов развешиваем на второй мачте и т.д. Ясно, что используя все перестановки n флагов и все способы распределения n одинаковых флагов по k мачтам, получим все способы решения поставленной задачи. По правилу произведения получаем, что число способов развешивания флагов равно

$$n! C_{n+k-1}^{k-1} = \frac{(n+k-1)!}{(k-1)!} = A_{n+k-1}^n. \quad (5.3)$$

Вообще, если имеется n различных вещей, то число способов распределения этих вещей по k различным ящикам равно A_{n+k-1}^n .

Разные статистики

Задачи о раскладке предметов по ящикам весьма важны для статистической физики. Эта наука изучает, как распределяются по своим свой-

ствам физические частицы; например, какая часть молекул данного газа имеет при данной температуре ту или иную скорость. При этом множество всех возможных состояний распределяют на большое число k маленьких ячеек (фазовых состояний), так что каждая из n частиц попадет в одну из ячеек.

Вопрос о том, какой статистике подчиняются те или иные частицы, зависит от вида этих частиц. В классической статистической физике, созданной Максвеллом и Больцманом, частицы считаются различимыми друг от друга. Такой статистике подчиняются, например, молекулы газа. Известно, что n различных частиц можно распределить по k ячейкам k^n способами. Если все эти k^n способов при заданной энергии имеют равную вероятность, то говорят о *статистике Максвелла-Больцмана*.

Оказалось, что этой статистике подчиняются не все физические объекты. Фотоны, атомные ядра и атомы, содержащие четное число элементарных частиц, подчиняются иной статистике, разработанной Эйнштейном и индийским ученым Бозе. В *статистике Бозе-Эйнштейна* частицы считаются неразличимыми друг от друга. Поэтому имеет значение лишь то, сколько частиц попало в ту или иную ячейку, а не то, какие именно частицы туда попали.

Однако для многих частиц, например таких как электроны, протоны и нейтроны, не годится и статистика Бозе-Эйнштейна. Для них в каждой ячейке может находиться не более одной частицы, причем различные распределения, удовлетворяющие указанному условию, имеют равную вероятность. В этом случае может быть C_k^n различных распределений. Эта статистика называется *статистикой Дирака-Ферми*.

Деревья и перестановки из n элементов

С помощью леса можно представить перестановки из n элементов множества $M = \{a; b; c; d\}$ (множество мы определяем так: множество — это неупорядоченная совокупность различных объектов или структура данных, используемая для представления множества). Подсчитаем, сколько можно получить перестановок. Для n такой лес изображен на рис. 5.1.

Число сочетаний C_n^m

Рассмотрим подмножества множества, состоящего из пяти элементов, и подсчитаем их число. При этом записывать подмножества будем не с помощью букв, как обычно, а в виде последовательностей длиной пять, составленных из нулей и единиц. Каждая из единиц указывает на наличие в подмножестве соответствующего элемента. Например, подмножества,

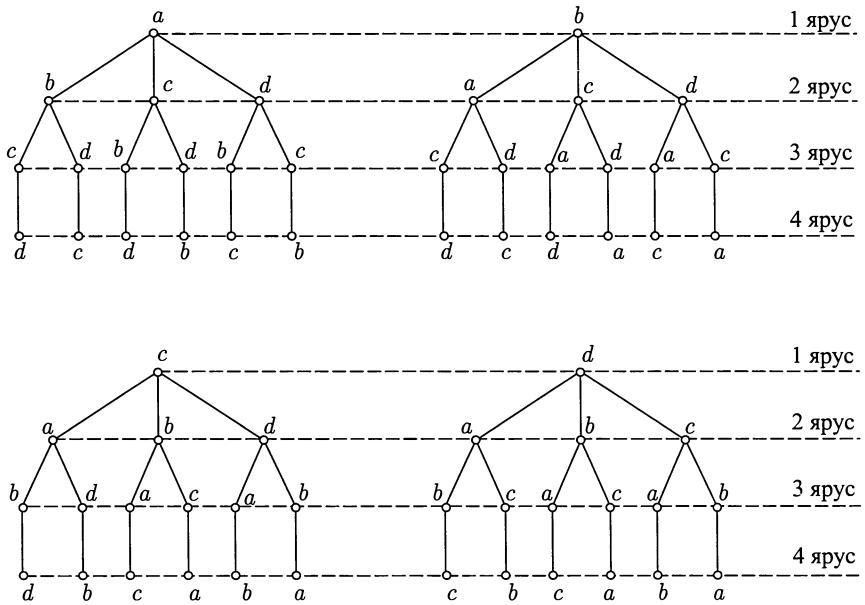


Рис. 5.1. Всевозможные перестановки прочитываются по этой схеме от корневой до висячей вершины соответствующего дерева. Ярус показывает номер места, на котором расположен элемент. Число висячих вершин леса равно числу перестановок

содержащие один элемент, будут изображаться следующими последовательностями: 10000, 01000, 00100, 00010, 00001. Пустое подмножество \emptyset будет соответствовать последовательности 00000. Подмножества, содержащие по два элемента из пяти, запишутся с помощью следующих последовательностей: 11000, 10100, 10010, 10001, 01100, 01010, 01001, 00110, 00101, 00011. Всего их $C_5^2 = 10$.

Вообще, число сочетаний из n элементов по m равно числу всевозможных последовательностей из m единиц и $n - m$ нулей.

Задачи на разбиение чисел

Теперь мы переходим к задачам, в которых все разделяемые предметы совершенно одинаковы. В этом случае можно говорить не о разделении предметов, а о разбиении натуральных чисел на слагаемые (которые, конечно, тоже должны быть натуральными числами).

Здесь возникает много различных задач. В одних задачах учитывается порядок слагаемых, в других — нет.

Задача 4. Отправка бандероли.

За пересылку бандероли надо уплатить 18 рублей. Сколькими способами можно оплатить ее марками стоимостью 4, 6, и 10 рублей, если два способа, отличающиеся порядком марок, считаются различными (запас марок различного достоинства считаем неограниченным)?

Обозначим через $f(N)$ число способов, которыми можно наклеить марки в 4, 6 и 10 рублей так, чтобы общая стоимость этих марок равнялась N . Тогда для $f(N)$ справедливо следующее соотношение:

$$f(N) = f(N - 4) + f(N - 6) + f(N - 10). \quad (5.4)$$

Пусть имеется некоторый способ наклейки марок с общей стоимостью N , и пусть последней наклеена марка стоимостью 4 рубля. Тогда все остальные марки стоят $(N - 4)$ рубля. Наоборот, присоединяя к любой комбинации марок общей стоимостью $(N - 4)$ рубля одну четырехрублевую марку, получаем комбинацию марок стоимостью N рублей. При этом из разных комбинаций стоимостью $(N - 4)$ рублей получается разные комбинации стоимостью N рублей. Итак, число искомых комбинаций, где последней наклеена марка стоимостью 4 рубля, равно $f(N - 4)$.

Точно так же доказывается, что число комбинаций, оканчивающихся на шестирублевую марку, равно $f(N - 6)$, а на десятирублевую марку оканчиваются $f(N - 10)$ комбинацией. Поскольку любая комбинация оканчивается на марку одного из указанных типов, то по правилу суммы получаем соотношение 5.4.

Соотношение 5.4 позволяет свести задачу о наклеивании марок на сумму N рублей к задачам о наклеивании марок на меньшие суммы. Но при малых значениях N задачу легко решить непосредственно. Простой подсчет показывает, что $f(0) = 1$, $f(1) = f(2) = f(3) = 0$, $f(4) = 1$, $f(5) = 0$, $f(6) = 1$, $f(7) = 0$, $f(8) = 1$, $f(9) = 0$. Равенство $f(0) = 1$ означает, что сумму в 0 рублей можно уплатить единственным образом: совсем не наклеивая марок. А сумму в 1, 2, 3, 5, 7 и 9 рублей вообще никак нельзя получить с помощью марок стоимостью 4, 6 и 10 рублей. Используя значения $f(N)$ для $N = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$, легко найти $f(10)$:

$$f(10) = f(6) + f(4) + f(0) = 3.$$

После этого находим

$$f(11) = f(7) + f(5) + f(1) = 0,$$

$$f(12) = f(8) + f(6) + f(2) = 2$$

и т. д. Наконец, получаем значение $f(18) = 8$. Таким образом, марки можно наклеить восемью способами. Эти способы таковы: 10, 4, 4; 4, 10, 4; 4, 4, 10; 6, 4, 4, 4; 4, 6, 4, 4; 4, 4, 6, 4; 4, 4, 4, 6; 6, 6, 6. Отметим, что значения $f(N)$ для $N = 1, 2, 3, 4, 5, 6, 7, 8, 9$ можно было получить иначе, не приводя непосредственно проверки. Дело в том, что при $N < 0$ имеем $f(N) = 0$, поскольку отрицательную сумму нельзя уплатить, наклеивая неотрицательное количество марок. В то же время, как мы видели, $f(0) = 1$. Поэтому $f(1) = f(-3) + f(-5) + f(-9) = 0$.

Точно так же получаем значение $f(2) = 0, f(3) = 0$, а для $N = 4$ имеем $f(4) = f(0) + f(-2) + f(-6) = 1$.

Задача 5. Общая задача о наклейке марок.

Разобранная задача является частным случаем следующей общей задачи: *Имеются марки достоинством в n_1, n_2, \dots, n_k . Сколькими способами можно оплатить с их помощью сумму в N рублей, если два способа, отличающиеся порядком, считаются различными? Все числа n_1, n_2, \dots, n_k различны, а запас марок неограничен. Здесь на первом месте мы будем указывать число слагаемых, на втором — разбиваемое число и на последнем — ограничения на величину слагаемых.*

В этом случае число $f(N)$ способов удовлетворяет соотношению

$$f(N) = f(N - n_1) + f(N - n_2) + \dots + f(N - n_k). \quad (5.5)$$

При этом $f(N) = 0$, если $f(0) = 1$ и $N < 0$. С помощью соотношения 5.5 можно найти $f(N)$ для любого N , последовательно вычисляя $f(1), f(2), \dots, f(N - 1)$.

Рассмотрим частный случай этой задачи, когда $n_1 = 1, n_2 = 2, \dots, n_k = k$. Мы получаем всевозможные разбиения числа N на слагаемые $1, 2, \dots, k$, причем разбиения, отличающиеся порядком слагаемых, считаются различными. Обозначим число этих разбиений через $\varphi(k; N)$. (На первом месте мы будем указывать число слагаемых, на втором — разбиваемое число и на последнем — ограничения на величину слагаемых.) Из соотношения 5.5 следует, что

$$\varphi(k; N - 1) + \varphi(k; N - 2) + \dots + \varphi(k; N - k). \quad (5.6)$$

При этом $\varphi(k; 0) = 1$ и $\varphi(k; N) = 0$, если $N < 0$. Вычисление $\varphi(N; k)$ можно упростить, если заметить, что

$$\varphi(N - 1; k) = \varphi(N - 2; k) + \varphi(N - k - 1; k),$$

и потому

$$\varphi(N; k) = 2\varphi(N - 1; k) - \varphi(N - k - 1; k). \quad (5.7)$$

Ясно, что слагаемые не могут быть больше N . Поэтому $\varphi(N, N)$ равно числу всех разбиений на N на натуральные слагаемые (включая и «разбиение» $N = N$). Если число слагаемых равно s , то получаем C_{N-1}^{s-1} разбиений. Поэтому

$$\varphi(N, N) = C_{N-1}^0 + C_{N-1}^1 + \dots + C_{N-1}^{N-1} = 2^{N-1}.$$

Итак, мы доказали, что натуральное число N можно разбить на слагаемые 2^{N-1} способами. Напомним, что при этом учитывается порядок слагаемых. Например, число 5 можно разбить на слагаемые $2^{5-1} = 16$ способами.

$5 = 5$	$5 = 3 + 1 + 1$	$5 = 1 + 2 + 2$
$5 = 4 + 1$	$5 = 1 + 3 + 1$	$5 = 2 + 1 + 1 + 1$
$5 = 1 + 4$	$5 = 1 + 1 + 3$	$5 = 1 + 2 + 1 + 1$
$5 = 2 + 3$	$5 = 2 + 2 + 1$	$5 = 1 + 1 + 2 + 1$
$5 = 3 + 2$	$5 = 2 + 1 + 2$	$5 = 1 + 1 + 1 + 2$
		$5 = 1 + 1 + 1 + 1 + 1$

Комбинаторные задачи теории информации

Информация — сведения, неизвестные до их получения, или данные, или значения, приписанные данным.

Теория информации — математическая дисциплина, изучающая количественные свойства информации.

Задачу, похожую на только что решенную, приходится решать в теории информации. Предположим, что сообщение передается с помощью сигналов нескольких типов. Длительность передачи сигнала первого типа равна t_1 , второго типа — t_2, \dots, k -го типа — t_k единиц времени.

Задача 6. Сколько различных сообщений можно передать с помощью этих сигналов за T единиц времени? При этом учитываются лишь «максимальные» сообщения, то есть сообщения, к которым нельзя присоединить ни одного сигнала, не выйдя за рамки отведенного для передачи времени.

Обозначим число сообщений, которые можно передать за время T через $f(T)$. Рассуждая точно так же, как и в задаче о марках, получаем, что $f(T)$ удовлетворяет соотношению

$$f(T) = f(T - t_1) + \dots + f(T - t_k). \quad (5.8)$$

При этом снова $f(T) = 0$, если $T < 0$ и $f(0) = 1$.

Лекция 6. Последовательности (множества и мультимножества)

Множества и мультимножества. Формула включений и исключений. Решето Эратосфена. Примеры программ. Вопросы и ответы.

Ключевые слова: множество, мультимножество, кратность, объединение, пересечение, имя, представитель множества, формула включений и исключений.

Множества и мультимножества

Не существует формального определения *множества*; считается что это понятие первичное и не определяется. Так, можно говорить, что множество есть объединение различных элементов, но при этом мы оставляем неопределяемыми понятия «объединение» и «элементы». Дадим следующее определение множеству: *множество* — это неупорядоченная совокупность различных объектов или структура данных, используемая для представления множества. *Мультимножество* есть объединение не обязательно различных элементов; его можно считать множеством, в котором каждому элементу поставлено в соответствие положительное целое число, называемое *кратностью*.

Конечное множество S будем записывать в следующем виде:

$$S = \{s_1, s_2, \dots, s_n\},$$

где s_1, s_2, \dots, s_n — элементы S , *обязательно различные!* Мощность множества S обозначается как $|S|$, для выписанного выше множества мощность записывается так $|S| = n$. Если S — конечное мультимножество, то будем записывать его в следующем виде:

$$S = \{ \underset{m_1 \text{ раз}}{s_1, s_1, \dots, s_1}, \underset{m_2 \text{ раз}}{s_2, s_2, \dots, s_2}, \underset{m_3 \text{ раз}}{s_3, s_3, \dots, s_3} \} = \{m_1 \bullet s_1, m_2 \bullet s_2, \dots, m_n \bullet s_n\}.$$

Здесь все s_i различны и m_i — кратность элемента s_i . В этом случае мощность S равна

$$|S| = \sum_{i=1}^n m_i.$$

Наиболее общими операциями на множествах и мультимножествах являются операции *объединения* и *пересечения*. Для множеств эти операции будем обозначать \cup и \cap , а для мультимножеств — $\overset{+}{\cup}$ и $\overset{+}{\cap}$. Последовательное и

связанное представление последовательностей можно использовать для множеств и мультимножеств очевидным способом. Индуцируя искусственный порядок элементов множества или используя собственный порядок, если он существует, можно рассматривать множество как последовательность. Аналогично, как последовательность можно рассматривать и мультимножество, или, для того чтобы сэкономить место, его можно рассматривать как последовательность пар, каждая из которых состоит из элемента и его кратности.

Как и для последовательностей, наилучший метод представления множеств или мультимножеств существенно зависит от операций, которые выполняются над ними. Предположим, например, что имеем дело с непересекающимися подмножествами множества $S = \{s_1, s_2, \dots, s_n\}$ и что над ними необходимо выполнить две следующие операции: объединение двух множеств и отыскание подмножества, содержащего данное s_i . Таким образом, в любой момент времени имеем разбиение S на непустые непересекающиеся подмножества. Рассмотрим эти операции в конце данной лекции.

С целью идентификации считаем, что каждое из непересекающихся подмножеств множества S имеет имя. **Имя** — это просто один из элементов подмножества, или, иначе, — представитель подмножества. Когда мы будем ссылаться на имя подмножества, то будем под этим подразумевать его *представителя*. Рассмотрим, например, множество

$$S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\},$$

разбитое на четыре непересекающихся подмножества

$$\begin{aligned} &\{1, 6, \langle 7 \rangle, 8, 11\} \\ &\quad \{\langle 2 \rangle\} \\ &\quad \{\langle 3 \rangle, 4, 5\} \\ &\quad \{9, \langle 10 \rangle\} \end{aligned} \tag{6.1}$$

В каждом из подмножеств, взятый в скобки элемент является его именем. Если нам нужно найти подмножество, в котором содержится восьмерка, искомым ответом будет 7, то есть имя подмножества, содержащего восьмерку. Если нужно взять объединение подмножеств с именами 2 и 10, получим разбиение множества S следующего вида:

$$\begin{aligned} &\{1, 6, \langle 7 \rangle, 8, 11\} \\ &\quad \{\langle 3 \rangle, 4, 5\} \\ &\quad \{\langle 2 \rangle\} \\ &\quad \cup \\ &\quad \{9, \langle 10 \rangle\} \end{aligned}$$

Именем множества $\{\langle 2 \rangle\} \cup \{9, \langle 10 \rangle\}$ может быть или 2, или 10. Предполагаем, что вначале имеется разбиение множества $S = \{s_1, s_2, \dots, s_n\}$ на n подмножеств, каждое из которых состоит из одного элемента

$$\{\langle s_1 \rangle\} \{\langle s_2 \rangle\} \dots \{\langle s_n \rangle\} \quad (6.2)$$

и имя каждого из них есть просто этот единственный элемент. Это разбиение преобразуется путем применения операций объединения вперемешку с операциями отыскания. Такая кажущаяся на первый взгляд надуманной задача чрезвычайно полезна в определенных комбинаторных алгоритмах; пример ее полезности виден в «жадном» алгоритме (лекция 16).

Для реализации операций и объединения, и отыскания опишем процедуры (операции) $UNION(x, y)$ и $FIND(x)$. Процедура (операция) $UNION(x, y)$ по именам двух различных подмножеств x и y образует новое подмножество, содержащее все элементы множеств x и y . Процедура (операция) $FIND(x)$ выдает имя множества, содержащего x . Например, если нужно множество, содержащее a , объединить с множеством, содержащим b , необходимо выполнить следующую последовательность операций:

$$\begin{aligned} x &\leftarrow FIND(a) \\ y &\leftarrow FIND(a) \\ \text{if } x \neq y &\text{ then } UNION(x, y). \end{aligned}$$

Предположим, что мы имеем u операций объединения, перемешанных с f операциями отыскания, и что начинаем алгоритм с множества $S = \{s_1, s_2, \dots, s_n\}$, которое разбито на подмножества, состоящие из одного элемента (см. 6.2.). Найдем такую структуру данных для представления непересекающихся подмножеств множества S , чтобы последовательность операций можно было производить эффективно. Такой структурой данных является представление в виде леса с указателями отца, как показано на рис. 4.5 лекции 4. Каждый элемент s_i множества будет узлом леса, а отцом его будет элемент из того же подмножества, что и s_i . Если элемент не имеет отца, то есть является корнем, то он будет именем своего подмножества. В соответствии с этим разбиение 5.1 может быть представлено так:

При таком представлении процедура (операция) $FIND(x)$ состоит в переходах по указателям отцов от x до корня, то есть имени, его подмножества. Процедура (операция) $UNION(x, y)$ состоит в связывании вместе некоторым образом деревьев, имеющих корни x и y . Например, такую связь можно осуществить, сделав y отцом x .

После u операций объединения наибольшее из возможных подмножеств, получающихся в результате разбиения S , будет содержать $u+1$ элементов. Поскольку каждое объединение уменьшает число подмножеств

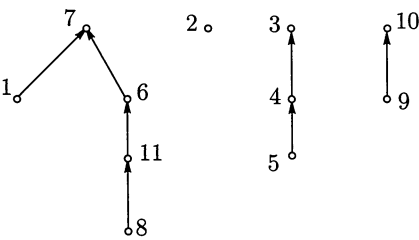


Рис. 6.1. Представление разбиения 6.1

на единицу, последовательность операций может содержать не более $n - 1$ объединений, откуда $u \leq n - 1$. Так как каждая операция объединения изменяет имя подмножества, содержащего некоторые элементы, можно считать, что каждому объединению предшествует по крайней мере одно отыскание, в связи с чем естественно предположить, что $f \geq u$. Выясним, насколько эффективно можно выполнить последовательность из $u \leq n - 1$ операций объединения, перемешанных с $f \geq u$ операциями отыскания. Время, требуемое на операции объединения, очевидно, пропорционально u , потому что необходимая для каждой операции объединения переделка некоторых указателей требует фиксированного количества работы. Поэтому сосредоточим свое внимание на времени, требуемом для f операций отыскания.

Если операция $UNION(x, y)$ выполняется путем назначения x отцом y , то после u операций объединения может получиться лес, показанный на рис. 5.2.

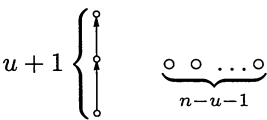


Рис. 6.2. Возможный лес, получающийся в результате применения операции $UNION$ u раз

В этом случае, если f операций отыскания выполняются после всех операций объединения и каждый поиск начинается внизу цепи из $u + 1$ элементов множества, то ясно, что время, требуемое на операции отыскания, будет пропорционально $f \cdot (u + 1)$. Очевидно, оно не может быть больше, чем константа, умноженная на $f \cdot (u + 1)$. Можно существенно уменьшить эту оценку.

Формула включений и исключений

Пусть имеется N предметов, некоторые из которых обладают свойствами $\alpha_1, \alpha_2, \dots, \alpha_n$. При этом каждый предмет может либо не обладать ни одним из этих свойств, либо обладать одним или несколькими свойствами. Обозначим через $N(\alpha_i \alpha_j \dots \alpha_k)$ количество предметов, обладающих свойствами $\alpha_i, \alpha_j, \dots, \alpha_k$ (и быть может, еще некоторыми из других свойств). Если нужно взять предметы, не обладающие некоторым свойством, то эти свойства пишем со штрихом. Например, через $N(\alpha_1 \alpha_2 \alpha'_4)$ обозначено количество предметов, обладающих свойствами α_1, α_2 , но не обладающих свойством α_4 (вопрос об остальных свойствах останется открытым). Число предметов, не обладающих ни одним из указанных свойств, обозначается по этому правилу через $N(\alpha'_1 \alpha'_2 \dots \alpha'_n)$. Общий закон состоит в том, что

$$\begin{aligned} N(\alpha'_1 \alpha'_2 \dots \alpha'_n) = & N - N(\alpha_1) - N(\alpha_2) - \dots - N(\alpha_n) + N(\alpha_1 \alpha_2) + \\ & + N(\alpha_1 \alpha_3) + \dots + N(\alpha_1 \alpha_n) + \dots + N(\alpha_{n-1} \alpha_n) - N(\alpha_1 \alpha_2 \alpha_3) - \dots - \\ & - N(\alpha_{n-2} \alpha_{n-1} \alpha_n + \dots + (-1)^n N(\alpha_1 \alpha_2 \dots \alpha_n). \end{aligned} \quad (6.3)$$

Здесь алгебраическая сумма распространена на все комбинации свойств $\alpha_1, \alpha_2, \dots, \alpha_n$ (без учета их порядка), причем знак $+$ ставится, если число учитываемых свойств четно, и знак $-$, если это число нечетно. Например, $N(\alpha_1 \alpha_3 \alpha_6 \alpha_8)$ входит со знаком $+$, а $N(\alpha_3 \alpha_4 \alpha_{10})$ со знаком $-$. Формулу 6.3 называют **формулой включений и исключений** — сначала исключаются все предметы, обладающие хотя бы одним из свойств $\alpha_1, \alpha_2, \dots, \alpha_n$, потом включаются предметы, обладающие, по крайней мере, двумя из этих свойств, затем исключаются имеющие, по крайней мере, три и т.д.

Решето Эратосфена

Одной из самых больших загадок математики является расположение простых чисел в ряду всех натуральных чисел. Иногда два простых числа идут через одно, (например, 17 и 19, 29 и 31), а иногда подряд идет миллион составных чисел. Сейчас ученые знают уже довольно много о том, сколько простых чисел содержится среди N первых натуральных чисел. В этих подсчетах весьма полезным оказался метод, восходящий еще к древнегреческому ученому Эратосфену. Он жил в третьем веке до новой эры в Александрии.

Эратосфен занимался самыми различными вопросами — ему принадлежат интересные исследования в области математики, астрономии и других наук. Впрочем, такая разносторонность привела его к некоторой поверхностности. Современники несколько иронически называли

Эратосфена «во всем второй»: второй математик после Евклида, второй астроном после Гиппарха и т.д.

В математике Эратосфена интересовал как раз вопрос о том, как найти все простые числа среди натуральных чисел от 1 до N . (Эратосфен считал 1 простым числом. Сейчас математики считают 1 числом особого вида, которое не относится ни к простым, ни к составным числам.) Он придумал для этого следующий способ. Сначала вычеркивают все числа, делящиеся на 2 (исключая само число 2). Потом берут первое из оставшихся чисел (а именно 3). Ясно, что это число — простое. Вычеркивают все идущие после него числа, делящиеся на 3. Первым оставшимся числом будет 5. Вычеркивают все идущие после него числа, делящиеся на 5, и т.д. Числа, которые уцелеют после всех вычеркиваний, и являются простыми. Так как во времена Эратосфена писали на восковых табличках и не вычеркивали, а «выкалывали» цифры, то табличка после описанного процесса напоминала решето. Поэтому метод Эратосфена для нахождения простых чисел получил название «решето Эратосфена».

Подсчитаем, сколько останется чисел в первой сотне, если мы вычеркнем по методу Эратосфена числа, делящиеся на 2, 3 и 5. Иными словами, поставим такой вопрос: сколько чисел в первой сотне не делится ни на одно из чисел 2, 3, 5? Эта задача решается по формуле включения и исключения.

Обозначим через α_1 свойство числа делиться на 2, через α_2 — свойство делимости на 3 и через α_3 — свойство делимости на 5. Тогда $\alpha_1\alpha_2$ означает, что число делится на 6, $\alpha_1\alpha_3$ означает, что оно делится на 10, и $\alpha_2\alpha_3$ — оно делится на 15. Наконец, $\alpha_1\alpha_2\alpha_3$ означает, что число делится на 30. Надо найти, сколько чисел от 1 до 100 не делится ни на 2, ни на 3, ни на 5, то есть не обладает ни одним из свойств α_1 , α_2 , α_3 . По формуле 5.3 имеем $N(\alpha'_1\alpha'_2\alpha'_3) = 100 - N(\alpha_1) - N(\alpha_2) - N(\alpha_3) + N(\alpha_1\alpha_2) + N(\alpha_1\alpha_3) + N(\alpha_2\alpha_3) - N(\alpha_1\alpha_2\alpha_3)$. Но чтобы найти, сколько чисел от 1 до N делится на n , надо разделить N на n и взять целую часть получившегося частного. Поэтому

$$N(\alpha_1) = 50, N(\alpha_2) = 33, N(\alpha_3) = 20,$$

$$N(\alpha_1\alpha_2) = 16, N(\alpha_1\alpha_3) = 10, N(\alpha_2\alpha_3) = 6, N(\alpha_1\alpha_2\alpha_3) = 3,$$

и значит,

$$N(\alpha'_1\alpha'_2\alpha'_3) = 32.$$

Таким образом, 32 числа от 1 до 100 не делятся ни на 2, ни на 3, ни на 5. Эти числа и уцелеют после первых трех шагов процесса Эратосфена. Кроме них останутся сами числа 2, 3 и 5. Всего останется 35 чисел.

А из первой тысячи после первых трех шагов процесса Эратосфена

останется 335 чисел. Это следует из того, что в этом случае

$$N(\alpha_1) = 500, N(\alpha_2) = 333, N(\alpha_3) = 200,$$

$$N(\alpha_1\alpha_2) = 166, N(\alpha_1\alpha_3) = 100, N(\alpha_2\alpha_3) = 66, N(\alpha_1\alpha_2\alpha_3) = 33.$$

Примеры программы

Программа 1. Решето Эратосфена.

{В примере, иллюстрирующем работу с множествами, реализуется алгоритм выделения из первой сотни натуральных чисел всех простых чисел. В основе алгоритма лежит прием «решета Эратосфена».

Алгоритм написан на языке программирования Turbo-Pascal.}

```
User crt;
Const
  N=100; {количество элементов исходного множества}
Type
  SetN=set of 1..N;
var
  n1, next, l : word; {вспомогательные переменные }
  BeginSet,      {исходное множество }
  PrimerSet: SetN; {множество простых чисел }
Begin
  Clrscr; {почистить экран}
  BeginSet:=[2..N]; {создать исходное множество}
  PrimerSet:=[1];   {первое простое число}
  next:=2;          {следующее простое число}
while BeginSet <> [ ] do {начало основного цикла}
  begin
    n1:=next; {n1-число, кратное очередному простому (next)}
    while n1<=N do
      {цикл удаления из исходного множества непростых чисел}
    begin
      BeginSet:=BeginSet-[n1];
      n1:=n1+next {следующее кратное}
    end;          {конец цикла удаления}
  repeat {получить следующее простое число, которое есть первое
    не вычеркнутое из исходного множества}
    inc(next)
  until(next in BeginSet) or (next > N)
  end; {конец основного цикла}
{вывод результата}
```

```
textcolor(15); {задание цвета}
for l:=1 to N do
  if l in PrimerSet then write(l:8);
  readln;
end.
```

Программа 2. Простые числа в порядке убывания от 200.

{Находит и пишет все простые числа в порядке убывания от 2 до 200.

Алгоритм написан на языке программирования Turbo-Pascal.}

```
Uses crt;
const
n=197;
var
i,q,w,e,r,t:integer;
prost:array[1..n] of integer;

begin
clrscr;
e:=1;
r:=0;
for q:=1 to n do begin
r:=0;
for w:=2 to n-1 do
if (q<>w) and (q mod w = 0) then r:=1;
{prost[e]:=q; e:=e+1;}

if r=0 then begin{begin write(q,' ');}

prost[e]:=q;
e:=e+1;
end;
end;

for i:=e downto 2 do begin
write (prost[i],' ');
if wherex>70 then writeln;

end;
readln;
end.
```


Программа 3. Поиск литер в строке.

{Поиск числа вхождений в данную строку литер а, с, е, h.

Алгоритм написан на языке программирования Turbo-Pascal.}

Uses crt;

type

liter_set = set of char;

var

c:integer;

let: liter_set;

a:char;

begin

clrscr;

let:=['a','c','e','h'];

repeat

a:=readkey;

write(a);

if a in let then c:=c+1;

until a = '.';

writeln;

writeln('Общее число вхождений литер а,с,е,h в вашу запись:',c);

readln;

end.

Лекция 7. Рекуррентные соотношения

Размещения без повторений. Перестановки. Сочетания. Рекуррентные соотношения. Другой метод доказательства. Процесс последовательных разбиений. Задача: «Затруднение мажордома». Вопросы и ответы.

Ключевые слова: размещения без повторений, перестановки из n элементов, n - перестановки, k -сочетания, метод рекуррентных соотношений, числа Фибоначчи, поиск по числам Фибоначчи.

Размещения без повторений

Имеется n различных предметов. Сколько из них можно составить k -расстановок? При этом две расстановки считаются различными, если они либо отличаются друг от друга хотя бы одним элементом, либо состоят из одних и тех же элементов, но расположенных в разном порядке. Такие расстановки называют **размещениями без повторений**, а их число обозначают A_n^k . При составлении k -размещений без повторений из n предметов нам надо сделать k выборов. На первом шагу можно выбрать любой из имеющихся n предметов. Если этот выбор уже сделан, то на втором шагу приходится выбирать из оставшихся $n - 1$ предметов. На k -м шагу $n - k + 1$ предметов. Поэтому по правилу произведения получаем, что число k -размещений без повторения из n предметов выражается следующим образом:

$$A_n^k = n(n - 1) \dots (n - k + 1).$$

Перестановки

При составлении размещений без повторений из n элементов по k мы получили расстановки, отличающиеся друг от друга и составом, и порядком элементов. Но если брать расстановки, в которые входят все n элементов, то они могут отличаться друг от друга лишь порядком входящих в них элементов. Такие расстановки называют **перестановками из n элементов**, или, короче, **n - перестановками**.

Сочетания

В тех случаях, когда нас не интересует порядок элементов в комбинации, а интересует лишь ее состав, говорят о сочетаниях. Итак, k -

сочетаниями из n элементов называют всевозможные k -расстановки, составленные из этих элементов и отличающиеся друг от друга составом, но не порядком элементов. Число k -сочетаний, которое можно составить из n элементов, обозначают через C_n^k .

Формула для числа сочетаний получается из формулы для числа размещений. В самом деле, составим сначала все k -сочетания из n элементов, а потом переставим входящие в каждое сочетание элементы всеми возможными способами. При этом получается, что все k -размещения из n элементов, причем каждое только по одному разу. Но из каждого k -сочетания можно сделать $k!$ перестановок, а число этих сочетаний равно C_n^k . Значит справедлива формула

$$k!C_n^k = A_n^k.$$

Из этой формулы находим, что

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

Рекуррентные соотношения

При решении многих комбинаторных задач пользуются методом сведения данной задачи к задаче, касающейся меньшего числа предметов. Метод сведения к аналогичной задаче для меньшего числа предметов называется **методом рекуррентных соотношений** (от латинского «гесигеге» — «возвращаться»).

Понятие рекуррентных соотношений проиллюстрируем классической проблемой, которая была поставлена около 1202 года Леонардо из Пизы, известным как Фибоначчи. Важность чисел Фибоначчи для анализа комбинаторных алгоритмов делает этот пример весьма подходящим.

Фибоначчи поставил задачу в форме рассказа о скорости роста популяции кроликов при следующих предположениях. Все начинается с одной пары кроликов. Каждая пара становится фертильной через месяц, после чего каждая пара рождает новую пару кроликов каждый месяц. Кролики никогда не умирают, и их воспроизводство никогда не прекращается.

Пусть F_n — число пар кроликов в популяции по прошествии n месяцев, и пусть эта популяция состоит из N_n пар приплода и O_n «старых» пар, то есть $F_n = N_n + O_n$. Таким образом, в очередном месяце произойдут следующие события: $O_{n+1} = O_n + N_n = F_n$. Старая популяция в $(n+1)$ -й момент увеличится на число родившихся в момент времени n . $N_{n+1} = O_n$. Каждая старая пара в момент времени n производит пару

приплода в момент времени $(n + 1)$. В последующий месяц эта картина повторяется:

$$O_{n+2} = O_{n+1} + N_{n+1} = F_{n+1},$$

$$N_{n+2} = O_{n+1}$$

Объединяя эти равенства, получим следующее рекуррентное соотношение:

$$F_{n+2} = O_{n+2} + N_{n+2} = F_{n+1} + O_{n+1},$$

$$F_{n+2} = F_{n+1} + F_n \quad (7.1)$$

Выбор начальных условий для последовательности чисел Фибоначчи не важен; существенное свойство этой последовательности определяется рекуррентным соотношением. Будем предполагать $F_0 = 0, F_1 = 1$ (иногда $F_0 = F_1 = 1$).

Рассмотрим эту задачу немного иначе.

Пара кроликов приносит раз в месяц приплод из двух крольчат (самки и самца), причем новорожденные крольчата через два месяца после рождения уже приносят приплод. Сколько кроликов появится через год, если в начале года была одна пара кроликов?

Из условия задачи следует, что через месяц будет две пары кроликов. Через два месяца приплод даст только первая пара кроликов, и получится 3 пары. А еще через месяц приплод дадут и исходная пара кроликов, и пара кроликов, появившаяся два месяца тому назад. Поэтому всего будет 5 пар кроликов. Обозначим через $F(n)$ количество пар кроликов по истечении n месяцев с начала года. Ясно, что через $n + 1$ месяцев будут эти $F(n)$ пар и еще столько новорожденных пар кроликов, сколько было в конце месяца $n - 1$, то есть еще $F(n - 1)$ пар кроликов. Иными словами, имеет место рекуррентное соотношение

$$F(n + 1) = F(n) + F(n - 1) \quad (7.2)$$

Так как, по условию, $F(0) = 1$ и $F(1) = 2$, то последовательно находим

$$F(2) = 3, F(3) = 5, F(4) = 8$$

и т.д.

В частности, $F(12) = 377$.

Числа $F(n)$ называются **числами Фибоначчи**. Они обладают целым рядом замечательных свойств. Теперь выведем выражение этих чисел через C_m^k . Для этого установим связь между числами Фибоначчи и следующей комбинаторной задачей.

Найти число n последовательностей, состоящих из нулей и единиц, в которых никакие две единицы не идут подряд.

Чтобы установить эту связь, возьмем любую такую последовательность и сопоставим ей пару кроликов по следующему правилу: единицам соответствуют месяцы появления на свет одной из пар «предков» данной пары (включая и исходную), а нулями — все остальные месяцы. Например, последовательность 010010100010 устанавливает такую «генеалогию»: сама пара появилась в конце 11-го месяца, ее родители — в конце 7-го месяца, «дед» — в конце 5-го месяца и «прадед» — в конце второго месяца. Исходная пара кроликов тогда зашифровывается последовательностью 000000000000.

Ясно, что при этом ни в одной последовательности не могут стоять две единицы подряд — только что появившаяся пара не может, по условию, принести приплод через месяц. Кроме того, при указанном правиле различным последовательностям отвечают различные пары кроликов, и обратно, две различные пары кроликов всегда имеют разную «генеалогию», так как, по условию, крольчиха дает приплод, состоящий только из одной пары кроликов.

Установленная связь показывает, что число n -последовательностей, обладающих указанным свойством, равно $F(n)$.

Докажем теперь, что

$$F(n) = C_{n+1}^0 + C_n^1 + C_{n-1}^2 + \dots + C_{n-p+1}^p, \quad (7.3)$$

где $p = \frac{n+1}{2}$, если n нечетно, и $p = \frac{n}{2}$, если n четно. Иными словами, p — целая часть числа $\frac{n+1}{2}$ (в дальнейшем будем обозначать целую часть числа α через $E(\alpha)$; таким образом, $p = E(\frac{n+1}{2})$).

В самом деле, $F(n)$ — это число всех n -последовательностей из 0 и 1, в которых никакие две единицы не стоят рядом. Число же таких последовательностей, в которые входит ровно k единиц и $n-k$ нулей, равно C_{n-k+1}^k . Так как при этом должно выполняться неравенство $k \leq n-k+1$, то k изменяется от 0 до $E(\frac{n+1}{2})$. Применяя правило суммы, приходим к соотношению (7.3).

Равенство (7.3) можно доказать и иначе. Положим

$$G(n) = C_{n+1}^0 + C_n^1 + C_{n-1}^2 + \dots + C_{n-p+1}^p,$$

где $p = \frac{n+1}{2}$. Из равенства $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ легко следует, что

$$G(n) = G(n-1) + G(n-2). \quad (7.4)$$

Кроме того, ясно, что $G(1) = 2 = F(1)$ и $G(2) = 3 = F(2)$. Так как обе последовательности $F(n)$ и $G(n)$ удовлетворяют рекуррентному соотношению $X(n) = X(n-1) + X(n-2)$, то имеем

$$G(3) = G(2) + G(1) = F(2) + F(1) = F(3),$$

и, вообще, $G(n) = F(n)$.

Другой метод доказательства

В предыдущем разделе непосредственно установлена связь между задачей Фибоначчи и комбинаторной задачей. Эту связь можно установить и иначе, непосредственно доказав, что число $T(n)$ решений комбинаторной задачи удовлетворяет тому же рекуррентному соотношению

$$T(n+1) = T(n) + T(n-1), \quad (7.5)$$

что и числа Фибоначчи. В самом деле, возьмем любую $(n+1)$ -последовательность нулей и единиц, удовлетворяющую условию, что никакие две единицы не идут подряд. Она может оканчиваться или на 0, или на 1. Если она оканчивается на 0, то, отбросив его, получим n -последовательность, удовлетворяющую нашему условию. Если взять любую n -последовательность нулей и единиц, в которой подряд не идут две единицы, и приписать к ней нуль, то получим $(n+1)$ -последовательность с тем же свойством. Таким образом доказано, что число последовательностей, оканчивающихся на нуль, равно $T(n)$.

Пусть теперь последовательность оканчивается на 1. Так как двух единиц подряд быть не может, то перед этой единицей стоит нуль. Иными словами, последовательность оканчивается на 01. Остающаяся же после отбрасывания 0 и 1 $(n-1)$ -последовательность может быть любой, лишь бы в ней не шли подряд две единицы. Поэтому число последовательностей, оканчивающихся единицей, равно $T(n-1)$. Но каждая последовательность оканчивается или на 0, или на 1. В силу правила суммы получаем, что $T(n+1) = T(n) + T(n-1)$.

Таким образом, получено то же самое рекуррентное соотношение. Отсюда еще не вытекает, что числа $T(n)$ и $F(n)$ совпадают.

Процесс последовательных разбиений

Для решения комбинаторных задач часто применяют метод, использованный в предыдущем пункте: устанавливают для данной задачи рекуррентное соотношение и показывают, что оно совпадает с рекуррентным соотношением для другой задачи, решение которой нам уже известно. Если при этом совпадают и начальные члены последовательностей в достаточном числе, то обе задачи имеют одинаковые решения.

Применим описанный прием для решения следующей задачи.

Пусть дано некоторое множество из n предметов, стоящих в определенном порядке. Разобьем это множество на две непустые части так, чтобы одна из этих частей лежала левее второй (то есть, скажем, одна часть состоит из элементов от первого до m -го, а вторая — из элементов от $(m+1)$ -го до n -го). После этого каждую из частей таким же образом разо-

бьем на две непустые части (если одна из частей состоит уже из одного предмета, она не подвергается дальнейшим разбиениям). Этот процесс продолжается до тех пор, пока не получим части, состоящие из одного предмета каждая. Сколько существует таких процессов разбиения (два процесса считаются различными, если хотя бы на одном шагу они приводят к разным результатам)?

Обозначим число способов разбиения для множества из $n + 1$ предметов через B_n . На первом шагу это множество может быть разбито n способами (первая часть может содержать один предмет, два предмета, \dots , n предметов). В соответствии с этим множество всех процессов разбиений распадается на n классов — в s -класс входят процессы, при которых первая часть состоит из s предметов.

Подсчитаем число процессов в s -м классе. В первой части содержится s элементов. Поэтому ее можно разбивать далее B_{s-1} различными процессами. Вторая же часть содержит $n - s + 1$ элементов, и ее можно разбивать далее B_{n-s} процессами. По правилу произведения получаем, что s -класс состоит из $B_{s-1}B_{n-s}$ различных процессов. По правилу суммы отсюда вытекает, что

$$B_n = B_0B_{n-1} + B_1B_{n-2} + \dots + B_{n-1}B_0. \quad (7.6)$$

Таким образом получено рекуррентное соотношение для B_n . Двоичный поиск, поиск делением пополам. **Поиском по числам Фибоначчи** называется поиск, основанный на том, что область поиска делится в точках, являющихся числами Фибоначчи.

Задача: «Затруднение мажордома»

Бывают комбинаторные задачи, в которых приходится составлять не одно рекуррентное соотношение, а систему соотношений, связывающую несколько последовательностей. Эти соотношения выражают $(n + 1)$ -у члены последовательностей через предыдущие члены не только данной, но и остальных последовательностей.

Задача: «Затруднение мажордома». Однажды мажордом короля Артура обнаружил, что к обеду за круглым столом приглашено 6 пар враждующих рыцарей. Сколькими способами можно рассадить их так, чтобы никакие два врага не сидели рядом?

Если мы найдем какой-то способ рассадки рыцарей, то, пересаживая их по кругу, получим еще 11 способов. Мы не будем сейчас считать различными способы, получающиеся друг из друга такой циклической пересадкой.

Введем следующие обозначения. Пусть число рыцарей равно $2n$. Через A_n обозначим число способов рассадки, при которых никакие два

врага не сидят рядом. Через B_n обозначим число способов, при которых рядом сидит ровно одна пара врагов, и через C_n — число способов, при которых есть ровно две пары враждующих соседей.

Выведем сначала формулу, выражающую A_{n+1} через A_n, B_n, C_n . Пусть $n + 1$ пар рыцарей посажены так, что никакие два врага не сидят рядом. Мы будем считать, что все враждующие пары рыцарей занумерованы. Попросим встать из-за стола пару рыцарей с номером $n + 1$. Тогда возможны три случая: среди оставшихся за столом нет одной пары соседей- врагов, есть одна такая пара и есть две такие пары (ушедшие рыцари могли разделять эти пары). Мы считаем, что $n > 1$. При $n = 1$ последующие рассуждения теряют силу.

Выясним теперь, сколькими способами можно снова посадить ушедших рыцарей за стол так, чтобы после этого не было одной пары соседей- врагов.

Проще всего посадить их, если за столом рядом сидят две пары врагов. В этом случае один из вновь пришедших садится между рыцарями первой пары, а другой — между рыцарями второй пары. Это можно сделать двумя способами. Но так как число способов рассадки $2n$ рыцарей, при которых две пары соседей оказались врагами, равно C_n , то всего получилось $2C_n$ способов.

Пусть теперь рядом сидит только одна пара врагов. Один из вернувшихся должен сесть между ними. Тогда за столом окажутся $2n + 1$ рыцарей, между которыми есть $2n + 1$ мест. Из них два места — рядом с только что севшим гостем — запретны для второго рыцаря, и ему остается $2n - 1$ мест. Так как первым может войти любой из двух вышедших рыцарей, то получается $2(2n - 1)$ способов рассадки. Но число случаев, когда $2n$ рыцарей сели так, чтобы ровно одна пара врагов оказалась соседями, равно B_n . Поэтому мы получаем $2(2n - 1)B_n$ способов посадить гостей требуемым образом.

Наконец, пусть никакие два врага не сидели рядом. В этом случае первый рыцарь садится между любыми двумя гостями — это он может сделать $2n$ способами. После этого для его врага останется $2n - 1$ мест — он может занять любое место, кроме двух мест, соседних с только что севшим рыцарем. Таким образом, если $2n$ рыцарей уже сидели нужным образом, то вернувшихся гостей можно посадить $2n(2n - 1)A_n$ способами. Как уже отмечалось, разработанными случаями исчерпываются все возможности. Поэтому имеет место рекуррентное соотношение

$$A_{n+1} = 2n(2n - 1)A_n + 2(2n - 1)B_n + 2C_n. \quad (7.7)$$

Этого соотношения пока недостаточно, чтобы найти A_n для всех значений n . Надо еще узнать, как выражаются B_n, C_{n+1} через A_n, B_n, C_n .

Предположим, что среди $2n+2$, $n > 1$ рыцарей оказалась ровно одна пара врагов-соседей. Мы знаем, что это может произойти в B_{n+1} случаях. Во избежание ссоры попросим их удалиться из-за стола. Тогда останется $2n$ рыцарей, причем возможно одно из двух: либо среди оставшихся нет врагов-соседей, либо есть ровно одна пара таких врагов — до ухода покинувших зал они сидели по обе стороны от них и теперь оказались рядом. Во втором случае ушедших можно посадить обратно только на старое место — иначе появится вторая пара враждующих соседей. Но так как $2n$ рыцарей можно посадить B_n способами так, чтобы была только одна пара враждующих соседей, то мы получаем $2B_n$ вариантов (возвратившихся рыцарей можно поменять местами). В первом же случае можно посадить ушедших между любыми двумя рыцарями, то есть $2n$ способами, а так как их еще можно поменять местами, то получится $4n$ способов. Комбинируя их со всеми способами посадки n пар рыцарей, при которых нет соседей врагов, получаем $4nA_n$ способов. Наконец, номер ушедшей и вернувшейся пары рыцарей мог быть любым от 1 до $n+1$. Отсюда вытекает, что рекуррентное соотношение для B_{n+1} имеет вид

$$B_{n+1} = 4n(n+1)A_n + 2(n+1)B_n. \quad (7.8)$$

Наконец, разберем случай, когда среди $2n+2$ рыцарей было две пары врагов-соседей. Номера этих пар можно выбрать $C_{n+1}^2 = \frac{n(n+1)}{2}$ способами. Заменим каждую пару одним новым рыцарем, причем будем считать новых двух рыцарей врагами. Тогда за столом будут сидеть $2n$ рыцарей, причем среди них либо не будет ни одной пары врагов-соседей (если новые рыцари не сидят рядом), либо только одна такая пара.

Первый вариант может быть в A_n случаях. Вернуться к исходной компании мы можем 4 способами благодаря возможности изменить порядок рыцарей в каждой паре. Поэтому первый вариант приводит к $4C_{n+1}^2 A_n = 2n(n+1)A_n$ способами.

Второй же вариант может быть в $\frac{1}{n}B_n$ случаях. Имеется B_n случаев, когда какая-нибудь пара врагов сидит рядом. Если указать, какая именно пара должна сидеть рядом, получим в n раз меньше случаев.

Здесь тоже можно вернуться к исходной компании 4 способами, и мы получаем всего $2(n+1)B_n$ способов. Отсюда вытекает, что при $n \geq 1$

$$C_{n+1} = 2n(n+1)A_n + 2(n+1)B_n. \quad (7.9)$$

Мы получили систему рекуррентных соотношений

$$A_{n+1} = 2n(2n-1)A_n + 2(2n-1)B_n + 2C \quad (7.10)$$

$$B_{n+1} = 4n(n+1)A_n + 2(n+1)B_n. \quad (7.11)$$

$$C_{n+1} = 2n(n+1)A_n + 2(n+1)B_n. \quad (7.12)$$

Они справедливы при $n \geq 2$. Но простой подсчет показывает, что $A_2 = 2, B_2 = 0, C_2 = 4$. Поэтому из соотношений 7.10–7.12 вытекает, что $A_3 = 32, B_3 = 48, C_3 = 24$. Продолжая далее, находим, что гостей можно посадить за стол требуемым образом $A_6 = 12771840$ способами.

Лекция 8. Алгоритмы рекуррентных соотношений

Решение рекуррентных соотношений. Линейные рекуррентные соотношения с постоянными коэффициентами. Случай равных корней характеристического уравнения. Производящие функции. Вопросы и ответы.

Ключевые слова: решение рекуррентного соотношения, общее решение, линейные рекуррентные соотношения с постоянными коэффициентами, производящая функция, формальный ряд.

Решение рекуррентных соотношений

Будем говорить, что рекуррентное соотношение имеет порядок k , если оно позволяет выразить $f(n+k)$ через $f(n), f(n+1), \dots, f(n+k-1)$. Например,

$$f(n+2) = f(n)f(n+1) - 3f^2(n+1) + 1$$

рекуррентное соотношение второго порядка, а

$$f(n+3) = 6f(n)f(n+2) + f(n+1)$$

— рекуррентное соотношение третьего порядка. Если задано рекуррентное соотношение k -го порядка, то ему удовлетворяет бесконечно много последовательностей. Дело в том, что первые k элементов последовательности можно задать совершенно произвольно — между ними нет никаких соотношений. Но если первые k элементов заданы, то все остальные элементы определяются совершенно однозначно — элемент $f(k+1)$ выражается в силу рекуррентного соотношения через $f(1), \dots, f(k)$ элемент $f(k+2)$ — через $f(2), \dots, f(k+1)$ и т.д.

Пользуясь рекуррентным соотношением и начальными членами, можно один за другим выписывать члены последовательности, причем рано или поздно получим любой ее член. Однако при этом придется выписать и все предыдущие члены — ведь не узнав их, мы не узнаем и последующих членов. Но во многих случаях нужно узнать только один определенный член последовательности, а остальные не нужны. В этих случаях удобнее иметь явную формулу для n -го члена последовательности. Некоторая последовательность является *решением данного рекуррентного соотношения*, если при подстановке этой последовательности соотношение тождественно выполняется. Например, последовательность

$$2, 4, 8, \dots, 2^n, \dots$$

является одним из решений рекуррентного соотношения

$$f(n+2) = 3f(n+1) - 2f(n).$$

В самом деле, общий член этой последовательности имеет вид $f(n) = 2^n$. Значит, $f(n+2) = 2^{n+2}$, $f(n+1) = 2^{n+1}$. Но при любом n имеет место тождество $2^{n+2} = 3 \cdot 2^{n+1} - 2 \cdot 2^n$. Поэтому 2^n является решением указанного соотношения.

Решение рекуррентного соотношения k -го порядка называется **общим**, если оно зависит от k произвольных постоянных C_1, C_2, \dots, C_k и путем подбора этих постоянных можно получить любое решение данного соотношения. Например, для соотношения

$$f(n+2) = 5f(n+1) - 6f(n) \quad (8.1)$$

общим решением будет

$$f(n) = C_1 2^n + C_2 3^n. \quad (8.2)$$

В самом деле, легко проверить, что последовательность (8.2) обращает (8.1) в тождество. Поэтому нам надо только показать, что любое решение нашего соотношения можно представить в виде (8.2). Но любое решение соотношения (8.1) однозначно определяется значениями $f(1)$ и $f(2)$. Поэтому нам надо доказать, что для любых чисел a и b найдутся такие значения C_1 и C_2 , что

$$2C_1 + 3C_2 = a$$

и

$$2^2 C_1 + 3^2 C_2 = b.$$

Но легко видеть, что при любых значениях a и b система уравнений

$$2C_1 + 3C_2 = a$$

$$4C_1 + 9C_2 = b$$

имеет решение. Поэтому (8.2) действительно является общим решением соотношения (8.1).

Линейные рекуррентные соотношения с постоянными коэффициентами

Для решения рекуррентных соотношений общих правил, вообще говоря, нет. Однако существует весьма часто встречающийся класс соотношений, решаемый единообразным методом. Это — рекуррентные соотношения вида

$$f(n+k) = a_1 f(n+k-1) + a_2 f(n+k-2) + \dots + a_k f(n), \quad (8.3)$$

где a_1, a_2, \dots, a_k — некоторые числа. Такие соотношения называют *линейными рекуррентными соотношениями с постоянными коэффициентами*.

Сначала рассмотрим, как решаются такие соотношения при $k = 2$, то есть изучим соотношение вида

$$f(n+2) = a_1 f(n+1) + a_2 f(n). \quad (8.4)$$

Решение этих соотношений основано на следующих двух утверждениях.

1. Если $f_1(n)$ и $f_2(n)$ являются решениями рекуррентного соотношения (8.4), то при любых числах A и B последовательность $f(n) = Af_1(n) + Bf_2(n)$ также является решением этого соотношения.

В самом деле, по условию, имеем

$$f_1(n+2) = a_1 f_1(n+1) + a_2 f_1(n)$$

$$f_2(n+2) = a_1 f_2(n+1) + a_2 f_2(n)$$

Умножим эти равенства на A и B соответственно и сложим полученные тождества. Получим, что

$$\begin{aligned} Af_1(n+2) + Bf_2(n+2) = \\ a_1[Af_1(n+1) + Bf_2(n+1)] + a_2[Af_1(n) + Bf_2(n)]. \end{aligned}$$

А это означает, что $Af_1(n) + Bf_2(n)$ является решением соотношения (8.4).

2. Если r_1 является корнем квадратного уравнения

$$r^2 = a_1 r + a_2,$$

то последовательность

$$1, r_1, r_1^2, \dots, r_1^{n-1}, \dots$$

является решением рекуррентного соотношения

$$f(n+2) = a_1 f(n+1) + a_2 f(n).$$

В самом деле, если $f(n) = r_1^{n-1}$, то $f(n+1) = r_1^n$ и $f(n+2) = r_1^{n+1}$. Подставляя эти значения в соотношение (8.4), получаем равенство

$$1^{n+1} = a_1 r_1^n + a_2 r_1^{n-1}.$$

Оно справедливо, так как по условию имеем $r^2 = a_1 r + a_2$. Заметим, что наряду с последовательностью $\{r_1^{n-1}\}$ любая последовательность вида

$$f(n) = r_1^{n+m}, n = 1, 2, \dots$$

также является решением соотношения (8.4). Для доказательства достаточно использовать утверждение (8.4), положив в нем $A = r_1^{m+1}$, $B = 0$.

Из утверждений 1 и 2 вытекает следующее правило решения линейных рекуррентных соотношений второго порядка с постоянными коэффициентами.

Пусть дано рекуррентное соотношение

$$f(n+2) = a_1 f(n+1) + a_2 f(n) \quad (8.5)$$

Составим квадратное уравнение

$$r^2 = a_1 r + a_2, \quad (8.6)$$

которое называется характеристическим для данного соотношения. Если это уравнение имеет два различных корня r_1, r_2 , то общее решение соотношения (8.5) имеет вид

$$f(n) = C_1 r_1^{n-1} + C_2 r_2^{n-1}.$$

Чтобы доказать это правило, заметим сначала, что по утверждению 2 $f_1(n) = r_1^{n-1}, f_2(n) = r_2^{n-1}$ являются решениями нашего соотношения. А тогда по утверждению 1 и $C_1 r_1^{n-1} + C_2 r_2^{n-1}$ является его решением. Надо только показать, что любое решение соотношения (8.5) можно записать в этом виде. Но любое решение соотношения второго порядка определяется значениями $f(1), f(2)$. Поэтому достаточно показать, что система уравнений

$$C_1 + C_2 = a,$$

$$C_1 r_1 + C_2 r_2 = b$$

имеет решение при любых a, b . Этими решениями являются

$$C_1 = \frac{b - ar_2}{r_1 - r_2},$$

$$C_2 = \frac{ar_1 - b}{r_1 - r_2}.$$

(Случай, когда оба корня уравнения (8.6) совпадут друг с другом, разберем в следующем пункте.)

Пример на доказанное правило.

При изучении чисел Фибоначчи мы пришли к рекуррентному соотношению

$$f(n) = f(n-1) + f(n-2). \quad (8.7)$$

Для него характеристическое уравнение имеет вид

$$r^2 = r + 1.$$

Корнями этого квадратного уравнения являются числа

$$r_1 = \frac{1 + \sqrt{5}}{2}, \quad r_2 = \frac{1 - \sqrt{5}}{2}.$$

Поэтому общее решение соотношения Фибоначчи имеет вид

$$f(n) = C_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n + C_2 \left(\frac{1 - \sqrt{5}}{2} \right)^n. \quad (8.8)$$

(Мы воспользовались сделанным выше замечанием и взяли показатели n вместо $n - 1$). Мы называли числами Фибоначчи решения соотношения (8.7), удовлетворяющее начальным условиям $f(0) = 1, f(1) = 2$ (то есть последовательность 1, 2, 3, 5, 8, 13, ...). Часто бывает более удобно добавить к этой последовательности вначале числа 0 и 1, то есть рассматривать последовательность 0, 1, 1, 2, 3, 5, 8, 13, ... Ясно, что эта последовательность удовлетворяет тому же самому рекуррентному соотношению (8.6) и начальным условиям $f(0) = 1, f(1) = 2$. Полагая в формуле (8.6) $n = 0, n = 1$, получаем для C_1, C_2 систему уравнений

$$\begin{aligned} C_1 + C_2 &= 0 \\ \frac{\sqrt{5}}{2}(C_1 - C_2) &= 1 \end{aligned}$$

Отсюда находим, что $C_1 = -C_2 = \frac{1}{\sqrt{5}}$ и потому

$$f(n) = \frac{1}{5} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right]. \quad (8.9)$$

На первый взгляд кажется удивительным, что это выражение при всех натуральных значениях n принимает целые значения.

Случай равных корней характеристического уравнения

Рассмотрим случай, когда оба корня характеристического уравнения совпадают: $r_1 = r_2$. В этом случае выражение $C_1 r_1^{n-1} + C_2 r_2^{n-1}$ уже не будет общим решением. Ведь из-за того, что $r_1 = r_2$, это решение можно записать в виде

$$f(n) = (C_1 + C_2) r_1^{n-1} = C r_1^{n-1}.$$

Остается только одно произвольное постоянное ; и выбрать его так, чтобы удовлетворить двум начальным условиям $f(1) = a, f(2) = b$, вообще

говоря, невозможно. Поэтому надо найти какое-нибудь второе решение, отличное от $f_1(n) = r_1^{n-1}$. Таким решением является $f_2(n) = nr_1^{n-1}$. В самом деле, если квадратное уравнение $r^2 = a_1r + a_2$ имеет два совпадающих корня $r_1 = r_2$, то по теореме Виета $a_1 = 2r_1, a_2 = -r_1^2$. Поэтому уравнение записывается так:

$$r^2 = 2r_1r - r_1^2.$$

А тогда рекуррентное соотношение имеет такой вид:

$$f(n+2) = 2r_1f(n+1) - r_1^2f(n). \quad (8.10)$$

Проверим, что $f_2(n) = nr_1^{n-1}$ действительно являются его решением. Имеем $f_2(n+2) = (n+2)r_1^{n+1}$, а $f_2(n+1) = (n+1)r_1^n$. Подставляя эти значения в соотношение (8.10), получаем очевидное тождество

$$(n+2)r_1^{n+1} = 2(n+1)r_1^{n+1} - nr_1^{n+1}.$$

Значит, nr_1^{n-1} — решение рассматриваемого соотношения.

Итак, имеются два решения $f_1(n) = r_1^{n-1}$ и $f_2(n) = nr_1^{n-1}$ заданного соотношения. Его общее решение запишется так:

$$f(n) = C_1r_1^{n-1} + C_2nr_1^{n-1} = r_1^{n-1}(C_1 + C_2n).$$

Теперь уже путем подбора C_1, C_2 можно удовлетворить любым начальным условиям.

Линейные рекуррентные соотношения с постоянными коэффициентами, порядок которых больше двух, решаются таким же способом. Пусть соотношение имеет вид

$$f(n+k) = a_1f(n+k-1) + \dots + a_nf(n). \quad (8.11)$$

Составим характеристическое уравнение

$$r^k = a_1r^{k-1} + \dots + a_k.$$

Если все корни r_1, r_2, \dots, r_k этого алгебраического уравнения k -й степени различны, то общее решение соотношения (8.3) имеет вид

$$f(n) = C_1r_1^{n-1} + C_2r_2^{n-1} + \dots + C_kr_k^{n-1}.$$

Если же, например, $r_1 = r_2 = \dots = r_s$, то этому корню соответствуют решения

$$f_1(n) = r_1^{n-1}, f_2(n) = nr_1^{n-1},$$

$$f_3(n) = n^2 r_1^{n-1}, \dots, f_s(n) = n^{s-1} r_1^{n-1}$$

рекуррентного соотношения (8.11). В общем решении этому корню соответствует часть

$$r_1^{n-1} [C_1 + C_2 n + C_3 n^2 + \dots + C_s n^{s-1}]$$

Составляя такое выражение для всех корней и складывая их, получаем общее решение соотношения (8.3).

Например, решим рекуррентное соотношение

$$f(n+4) = 5f(n+3) - 6f(n+2) - 4f(n+1) + 8f(n).$$

Характеристическое уравнение в этом случае имеет вид

$$r^4 - 5r^3 + 6r^2 + 4r - 8 = 0.$$

Решая его, получим корни

$$r_1 = 2, r_2 = 2, r_3 = 2, r_4 = -1.$$

Значит, общее решение нашего соотношения имеет следующий вид:

$$f(n) = 2^{n-1} [C_1 + C_2 n + C_3 n^2] + C_4 (-1)^{n-1}.$$

Производящие функции

В комбинаторных задачах на подсчет числа объектов при наличии некоторых ограничений искомым решением часто является последовательность a_0, a_1, a_2, \dots , где a_k — число искомых объектов «размерности» k . Например, если мы ищем число разбиений числа, то можем принять $a_k = P(k)$, если ищем число подмножеств n -элементного множества, то $a_k = \binom{n}{k}$ и т.д. В этом случае удобно последовательности a_0, a_1, a_2, \dots , поставить в соответствие формальный ряд

$$A(x) = \sum_{k=0}^{\infty} a_k x^k, \quad (8.12)$$

называемый *производящей функцией* для данной последовательности. Название *формальный ряд* для данной последовательности означает, что (8.12) мы трактуем только как удобную запись нашей последовательности — в данном случае несущественно, для каких (действительных или комплексных) значений переменной x он сходится. Поэтому мы никогда

не будем вычислять значение такого ряда для конкретного значения переменной x , мы будем только выполнять некоторые операции на таких рядах, а затем определять коэффициенты при отдельных степенях переменной x . Для произвольных рядов

$$A(x) = \sum_{k=0}^{\infty} a_k x^k, B(x) = \sum_{k=0}^{\infty} b_k x^k$$

мы определим операцию сложения:

$$A(x) + B(x) = \sum_{k=0}^{\infty} (a_k + b_k) x^k, \quad (8.13)$$

операцию умножения на число (действительное или комплексное):

$$pA(x) = \sum_{k=0}^{\infty} p a_k x^k \quad (8.14)$$

и произведение Коши

$$A(x) \cdot B(x) = \sum_{k=0}^{\infty} c_k x^k, \quad (8.15)$$

где

$$c_k = a_0 b_k + a_1 b_{k-1} + \dots + a_k b_0 = \sum_{i=0}^k a_i b_{k-i}, \quad (8.16)$$

Если $a_k = 0$ для $k > n$, то ряд (8.12) будем отождествлять с многочленом $a_n x^n + \dots + a_0$. Из математического анализа известно, что если ряд (8.12) сходится в некоторой окрестности нуля, то его сумма $A(x)$ является аналитической функцией в этой окрестности и

$$a_k = A^{(k)}(0)/k!, k = 0, 1, 2, \dots \quad (8.17)$$

$A^{(k)}(0)$ обозначает значение k -й производной функции $A(x)$ для $x = 0$; ряд 7.12— это не что иное, как ряд Маклорена функции $A(x)$). Более того, когда $A(x), B(x)$ являются аналитическими функциями в окрестности нуля, то формулы (8.13)–(8.16) будут справедливы, если $A(x), B(x)$ трактовать как значения функций A, B в точке x , а ряды понимать в обычном смысле, т. е. так, как в математическом анализе. Это сохраняющее операции взаимно однозначное соответствие между рядами, сходящимися в окрестности нуля, и функциями, аналитическими в окрестности нуля, позволяет отождествить формальный ряд (8.12) с определенной через

него аналитической функцией в случае рядов, сходящихся в окрестности нуля (несмотря на то, что ряды мы будем трактовать всегда как формальные ряды, то есть только как формальную запись их коэффициентов). Таким образом, будем писать, например,

$$\sum_{k=0}^{\infty} x^k = (1 - x)^{-1},$$

$$\sum_{k=0}^{\infty} \frac{1}{k!} x^k = e^x$$

и т. д.

Лекция 9. Комбинаторика и ряды

Введение. Деление многочленов. Алгебраические дроби и степенные ряды. Действия над степенными рядами. Вопросы и ответы.

Ключевые слова: многочлен частное, многочлен остаток, многочлен делимое, многочлен делитель, бесконечный числовой ряд сходится, сумма бесконечного ряда, расходящийся ряд, сумма степенных рядов, произведение рядов, частное при делении рядов.

Введение

Метод рекуррентных соотношений позволяет решать многие комбинаторные задачи. Но в целом ряде случаев рекуррентные соотношения довольно трудно составить, а еще труднее решить. Зачастую эти трудности удастся обойти, использовав производящие функции. Поскольку понятие производящей функции связано с бесконечными степенными рядами, познакомимся с этими рядами.

Деление многочленов

Если заданы два многочлена $f(x)$ и $\varphi(x)$, то всегда существуют **многочлены** $q(x)$ (**частное**) и $r(x)$ (**остаток**), такие, что $f(x) = \varphi(x)q(x) + r(x)$, причем степень $r(x)$ меньше степени $\varphi(x)$ или $r(x) = 0$. При этом $f(x)$ называется **делимым**, а $\varphi(x)$ — **делителем**. Если же мы хотим, чтобы деление выполнялось без остатка, то придется допустить в качестве частного не только многочлены, но и бесконечные степенные ряды. Для получения частного надо расположить многочлены по возрастающим степеням x и делить «углом», начиная с младших членов. Рассмотрим, например, деление 1 на $1 - x$

$$\begin{array}{r}
 1 \\
 \hline
 \mp 1 \pm x \\
 \hline
 x \\
 \mp x \pm x^2 \\
 \hline
 x^2 \\
 \mp x^2 \pm x^3 \\
 \hline
 x^3 \dots
 \end{array}
 \begin{array}{l}
 1 - x \\
 \hline
 1 + x + x^2 + \dots
 \end{array}$$

Ясно, что процесс деления никогда не закончится (так же, например, как при обращении числа $\frac{1}{3}$ в бесконечную десятичную дробь). С по-

мощью индукции легко убедиться, что все коэффициенты частного равны единице. Поэтому в качестве частного получается бесконечный ряд $1 + x + x^2 + \dots + x^n + \dots$. Вообще, если $f(x)$ и $\varphi(x)$ — два многочлена

$$f(x) = a_0 + \dots + a_n x^n,$$

$$\varphi(x) = b_0 + \dots + b_m x^m,$$

причем свободный член b_0 многочлена $\varphi(x)$ отличен от нуля, $b_0 \neq 0$, то при делении $f(x)$ на $\varphi(x)$ получается бесконечный ряд

$$c_0 + c_1 x + \dots + c_k x^k + \dots \quad (9.1)$$

Лишь в случае, когда $f(x)$ делится без остатка на $\varphi(x)$, ряд (9.1) обрывается и мы получаем многочлен.

Алгебраические дроби и степенные ряды

При делении многочлена $f(x)$ на многочлен $\varphi(x)$ мы получаем бесконечный степенной ряд. Возникает вопрос: как связан этот ряд с алгебраической дробью $\frac{f(x)}{\varphi(x)}$, то есть какой смысл можно придать записи

$$\frac{f(x)}{\varphi(x)} = c_0 + c_1 x + \dots + c_n x^n + \dots \quad (9.2)$$

Рассмотрим, например, разложение

$$\frac{1}{1-x} \cong 1 + x + x^2 + \dots + x^n + \dots \quad (9.3)$$

Мы не пишем здесь знака равенства, так как не знаем, какой смысл имеет стоящая справа сумма бесконечного числа слагаемых. Чтобы выяснить это, попробуем подставлять в обе части соотношения (9.3) различные значения x . Сначала положим $x = \frac{1}{10}$. Тогда левая часть соотношения примет значение $\frac{10}{9}$, а правая превратится в бесконечный числовой ряд $1+0,1+0,01+\dots+0,000\dots01+\dots$. Так как мы не умеем складывать бесконечно много слагаемых, попробуем взять сначала одно слагаемое, потом — два, потом — три и так далее слагаемых. Мы получим такие суммы: $1; 1,1; 1,11; \dots; 1,111\dots1; n$. Ясно, что с возрастанием n эти суммы приближаются к значению $\frac{10}{9} = 1,11\dots$, которое приняла левая часть соотношения (9.3) при $x = \frac{1}{10}$.

То же самое получится, если вместо x подставить в обе части (9.3) число $\frac{1}{2}$. Левая часть равенства примет значение 2, а правая превратится в бесконечный числовой ряд $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{2^n} + \dots$. Беря последовательно одно, два, три, четыре, слагаемых, мы получим числа $1; 1\frac{1}{2}; 1\frac{3}{4}; 1\frac{7}{8}; \dots, 2 - \frac{1}{2^n}$. Ясно, что с возрастанием n эти числа стремятся к числу 2.

Однако, если взять $x = 4$, то левая часть (9.3) примет значение $-\frac{1}{3}$, а в правой получим ряд $1 + 4 + 4^2 + \dots + 4^n + \dots$. Если последовательно складывать члены этого ряда, то получаются суммы 1; 5; 21; 85; ... Эти суммы неограниченно увеличиваются и не приближаются к числу $-\frac{1}{3}$.

Мы встретились, таким образом, с двумя случаями. Чтобы их различать, введем общее понятие о сходимости и расходимости числового ряда. Пусть задан бесконечный числовой ряд

$$a_1 + a_2 + \dots + a_n + \dots \quad (9.4)$$

Говорят, что **бесконечный числовой ряд сходится к числу b** , если разность $b - (a_1 + a_2 + \dots + a_n)$ стремится к нулю при неограниченном увеличении n . Иными словами, какое бы число $\varepsilon > 0$ мы ни указали, отклонение суммы $a_1 + \dots + a_n$ от b , начиная с некоторого номера N , окажется меньше ε :

$$|b - (a_1 + \dots + a_n)| < \varepsilon \text{ если } n \geq N.$$

В этом случае число b называют суммой бесконечного ряда $a_1 + a_2 + \dots + a_n + \dots$ и пишут

$$b = a_1 + a_2 + \dots + a_n + \dots$$

Если не существует числа b , к которому сходится данный ряд (9.4), то этот ряд называют **расходящимся**.

Проведенное выше исследование показывает, что

$$\begin{aligned} \frac{10}{9} &= 1 + 0,1 + 0,01 + \dots + 0,00\dots01 + \dots, \\ 2 &= 1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^n} + \dots, \end{aligned}$$

в то время как ряд $1 + 4 + 16 + \dots + 4^n + \dots$ расходится. Более тщательное исследование показывает, что если $|x| < 1$, то ряд $1 + x + \dots + x^n + \dots$ сходится к $\frac{1}{1-x}$, а если $|x| \geq 1$, то он расходится. Чтобы доказать это утверждение, достаточно заметить, что

$$1 + x + \dots + x^n = \frac{1 - x^{n+1}}{1 - x}$$

и что при $n \rightarrow \infty$ выражение x^{n+1} стремится к нулю, если $|x| < 1$, и к бесконечности, если $|x| \geq 1$. При $x = \pm 1$ получаем расходящиеся числовые ряды $1 + 1 + \dots + 1 + \dots$ и $1 - 1 + \dots + 1 - 1 + \dots$. Итак, если $|x| < 1$, то

$$\frac{1}{1-x} = 1 + x + \dots + x^n + \dots \quad (9.5)$$

Отметим, что равенство (9.5) — это известная из школьного курса математики формула для суммы бесконечно убывающей геометрической прогрессии.

Мы выяснили, таким образом, смысл записи

$$\frac{1}{1-x} = 1 + x + \dots + x^n + \dots$$

Она показывает, что для значений x , лежащих в некоторой области, а именно при $|x| < 1$, стоящий справа ряд сходится к $\frac{1}{1-x}$. Говорят, что функция $\frac{1}{1-x}$ при $|x| < 1$ разлагается в степенной ряд $1 + x + \dots + x^n + \dots$. Теперь уже можно выяснить и более общий вопрос.

Пусть при делении многочлена $f(x)$ на многочлен $\varphi(x)$ получился степенной ряд

$$c_0 + c_1x + \dots + c_nx^n + \dots \quad (9.6)$$

Оказывается, что тогда при достаточно малых значениях x ряд (9.6) сходится к $f(x)/\varphi(x)$. Размеры области сходимости зависят от корней знаменателя, то есть чисел, при которых знаменатель обращается в нуль. Именно, если эти числа равны x_1, \dots, x_k и r — наименьшее из чисел $|x_1|, \dots, |x_k|$, то ряд сходится в области $|x| < r$.

Иными словами, всегда есть область $|x| < r$, в которой выполняется равенство

$$\frac{f(x)}{\varphi(x)} = c_0 + c_1x + \dots + c_nx^n + \dots \quad (9.7)$$

В степенные ряды можно разлагать не только алгебраические дроби, но и многие другие функции. В математическом анализе доказывают, например, что

$$\begin{aligned} \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots, \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots, \\ e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \end{aligned}$$

Отметим еще следующее важное утверждение: *функция $f(x)$ не может иметь двух различных разложений в степенные ряды.*

Действия над степенными рядами

Перейдем теперь к действиям над степенными рядами. Пусть функции $f(x)$ и $\varphi(x)$ разложены в степенные ряды

$$f(x) = a_0 + a_1x + \dots + a_nx^n + \dots \quad (9.12)$$

$$\varphi(x) = b_0 + b_1x + \dots b_nx^n \dots \quad (9.13)$$

Тогда

$$f(x) + \varphi(x) = (a_0 + a_1x + \dots a_nx^n + \dots) + (b_0 + b_1x + \dots b_nx^n \dots).$$

Оказывается, что слагаемые в правой части равенства можно переставить и сгруппировать вместе члены с одинаковыми степенями x . *Это утверждение совсем не так очевидно, как кажется на первый взгляд. Ведь в правой части равенства у нас бесконечные суммы, а в бесконечных суммах переставлять слагаемые можно далеко не всегда.* После этой перегруппировки мы получим

$$f(x) + \varphi(x) = (a_0 + b_0) + (a_1 + b_1)x + \dots + (a_n + b_n)x^n + \dots \quad (9.14)$$

Ряд, стоящий в правой части равенства (9.14), называется **суммой степенных рядов** (9.12) и (9.13).

Посмотрим теперь, как разлагается в степенной ряд произведение функций $f(x)$ и $\varphi(x)$. Мы имеем

$$f(x)\varphi(x) = (a_0 + a_1x + \dots a_nx^n + \dots) \times (b_0 + b_1x + \dots b_nx^n \dots). \quad (9.15)$$

Оказывается, что как и в случае многочленов, ряды, стоящие в правой части равенства (9.15), можно почленно перемножать. Мы опускаем доказательство этого утверждения. Найдем ряд, получающийся после почленного перемножения. Свободный член этого ряда равен a_0b_0 . Члены, содержащие x , получатся дважды: при умножении a_0 на b_1x и при умножении a_1x на b_0 . Они дают

$$a_0b_1x + a_1b_0x = (a_0b_1 + a_1b_0)x.$$

Точно так же вычисляются члены, содержащие x^2 . Таким образом,

$$f(x)\varphi(x) = a_0b_0 + (a_0b_1 + a_1b_0)x + \dots + (a_0b_n + \dots + a_nb_0)x^n + \dots \quad (9.16)$$

Ряд, стоящий в правой части равенства (9.16), называется **произведением рядов** (9.12) и (9.13).

В частности, возводя ряд (9.12) в квадрат, получаем

$$f^2(x) = a_0^2 + 2a_0a_1x + (a_1^2 + 2a_0a_2)x^2 + 2(a_0a_3 + a_1a_2)x^3 + \dots \quad (9.17)$$

Посмотрим теперь, как делят друг на друга степенные ряды. Пусть свободный член ряда (9.13) отличен от нуля. Покажем, что в этом случае существует такой степенной ряд

$$c_0 + c_1x + \dots + c_nx^n + \dots, \quad (9.18)$$

что

$$(b_0 + b_1x + \dots + b_nx^n + \dots) \times (c_0 + c_1x + \dots + c_nx^n + \dots) = a_0 + a_1x + \dots + a_nx^n + \dots \quad (9.19)$$

Для доказательства перемножим ряды в левой части этого равенства. Мы получим ряд

$$b_0c_0 + (b_0c_1 + b_1c_0)x + \dots + (b_0c_n + \dots + b_nc_0)x^n + \dots$$

Для того чтобы этот ряд совпадал с рядом (9.12), необходимо и достаточно, чтобы выполнялись равенства

$$b_0c_0 = a_0,$$

$$b_0c_1 + b_1c_0 = a_1,$$

$$\dots \dots \dots$$

$$b_0c_n + \dots + b_nc_0 = a_n,$$

$$\dots \dots \dots$$

Эти равенства дают бесконечную систему уравнений для отыскания коэффициентов. Из первого уравнения системы получаем $c_0 = \frac{a_0}{b_0}$. Подставим полученное значение во второе уравнение. Мы получим уравнение

$$b_0c_1 = a_1 - \frac{b_1a_0}{b_0},$$

из которого находим, что $c_1 = \frac{a_1b_0 - b_1a_0}{b_0^2}$. Вообще, если уже найдены коэффициенты c_0, \dots, c_{n-1} , то для отыскания c_n имеем уравнение

$$b_0c_n = a_n - b_1c_{n-1} - \dots - b_nc_0.$$

Это уравнение разрешимо, поскольку $b_0 \neq 0$. Итак, мы доказали существование ряда (9.18), удовлетворяющего соотношению (9.19). Ряд (9.18) называют **частным при делении рядов** (9.12) и (9.13). Можно доказать, что он получается при разложении функции $f(x)/\varphi(x)$. Таким образом, степенные ряды можно складывать, умножать и делить (последнее — при условии, что свободный член делителя отличается от нуля). Эти действия соответствуют действиям над разлагаемыми функциями.

Лекция 10. Производящие функции и рекуррентные соотношения

Применение степенных рядов для доказательства тождеств. Производящие функции. Бином Ньютона. Ряд Ньютона. Производящие функции и рекуррентные соотношения. О едином нелинейном рекуррентном соотношении. Вопросы и ответы.

Ключевые слова: производящая функция.

Применение степенных рядов для доказательства тождеств

С помощью степенных рядов можно доказывать многие тождества. Для этого берут некоторую функцию и двумя способами разлагают ее в степенной ряд. Поскольку функция может быть представлена лишь единственным образом в виде степенного ряда, то коэффициенты при одинаковых степенях x в обоих рядах должны совпадать. Это и приводит к доказываемому тождеству.

Рассмотрим, например, известное нам разложение

$$\frac{1}{1-x} = 1 + x + x^2 + \dots + x^n + \dots$$

Возведя обе части этого разложения в квадрат, получаем

$$\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + \dots + (n+1)x^n + \dots \quad (10.1)$$

Если заменить здесь x на $-x$, то получим, что

$$\frac{1}{(1+x)^2} = 1 - 2x + 3x^2 - \dots + (-1)^n(n+1)x^n + \dots \quad (10.2)$$

Перемножив разложения (10.1) и (10.2), выводим, что

$$\begin{aligned} \frac{1}{(1-x)^2} \frac{1}{(1+x)^2} &= 1 + [1(-2) + 2 \cdot 1]x + [1 \cdot 3 + 2(-2) + 3 \cdot 1]x^2 + \\ &\dots + [1(-1)^n(n+1) + 2(-1)^{n-2}n + \dots + (-1)^n(n+1) \cdot 1]x^n + \dots \end{aligned} \quad (10.3)$$

Очевидно, что коэффициенты при нечетных степенях x обращаются в нуль, каждое слагаемое дважды входит в эти коэффициенты с противоположными знаками. Коэффициент же x^{2n} равен

$$1(2n+1) - 2 \cdot 2n + 3(2n-1) - \dots + (2n+1).$$

Но функцию $\frac{1}{(1-x)^2(1+x)^2}$ можно разложить в степенной ряд и иным образом. Мы имеем

$$\frac{1}{(1-x)^2(1+x)^2} = \frac{1}{(1-x^2)^2},$$

А разложение для $\frac{1}{(1-x^2)^2}$ получается из разложения (10.1), если заменить в нем x на x^2 :

$$\frac{1}{(1-x^2)^2} = 1 + 2x^2 + 3x^4 + \dots + (n+1)x^{2n} + \dots \quad (10.4)$$

Мы знаем, что никакая функция не может иметь двух различных разложений в степенные ряды. Поэтому коэффициент при x^{2n} разложения (10.3) должен равняться коэффициенту при x^{2n} в разложении (10.4). Отсюда вытекает следующее тождество:

$$1(2n+1) - 2 \cdot 2n + 3(2n-1) - \dots + (2n+1) = n+1.$$

Производящие функции

Пусть дана некоторая последовательность чисел $a_0, a_1, \dots, a_n, \dots$. Образуем степенной ряд

$$a_0 + a_1x + \dots + a_nx^n + \dots$$

Если этот ряд сходится в какой-то области к функции $f(x)$, то эту **функцию** называют **производящей** для последовательности чисел $a_0, a_1, \dots, a_n, \dots$. Например, из формулы

$$\frac{1}{1-x} = 1 + x + \dots + x^n + \dots$$

вытекает, что функция $\frac{1}{1-x}$ является производящей для последовательности чисел. А формула (10.1) показывает, что для последовательности чисел $1, 2, 3, 4, \dots, n, \dots$ производящей является функция $\frac{1}{(1-x)^2}$.

Нас будут интересовать производящие функции для последовательностей $a_0, a_1, \dots, a_n, \dots$, так или иначе связанных с комбинаторными задачами. С помощью таких функций удастся получить самые разные свойства этих последовательностей. Кроме того, мы рассмотрим, как связаны производящие функции с решением рекуррентных соотношений.

Бином Ньютона

Получим производящую функцию для конечной последовательности чисел $C_n^0, C_n^1, \dots, C_n^n$. Известно, что

$$(a+x)^2 = a^2 + 2ax + x^2$$

и

$$(a+x)^3 = a^3 + 3a^2x + 3ax^2 + x^3.$$

Эти равенства являются частными случаями более общей формулы, дающей разложение для $(a+x)^n$. Запишем $(a+x)^n$ в виде

$$(a+x)^n = \underbrace{(a+x)(a+x) \dots (a+x)}_{n \text{ раз}}. \quad (10.4)$$

Раскроем скобки в правой части этого равенства, причем будем записывать все множители в том порядке, в котором они нам встретятся. Например, $(a+x)^2$ запишем в виде

$$(a+x)^2 = (a+x)(a+x) = aa + ax + xa + xx, \quad (10.5)$$

а $(a+x)^3$ — в виде

$$\begin{aligned} (a+x)^3 &= (a+x)(a+x)(a+x) = \\ &= aaa + aax + axa + axx + xaa + xax + xxa + xxx. \end{aligned} \quad (10.6) \quad (())$$

Видно, что в формулу (10.5) входят все размещения с повторениями, составленные из букв x и a по две буквы в каждом размещении, а в формулу (10.6) — размещения с повторениями из тех же букв, но состоящие из трех букв каждое. То же самое и в общем случае — после раскрытия скобок в формуле (10.4) мы получим всевозможные размещения с повторениями букв x и a , состоящие из n элементов. Приведем подобные члены. Подобными будут члены, содержащие одинаковое количество букв x (тогда и букв a в них будет поровну). Найдем, сколько будет членов, в которые входит k букв x и, следовательно, $n-k$ букв a . Эти члены являются перестановками с повторениями, составленными из k букв x и $n-k$ букв a . Поэтому их число равно

$$P(k, n-k) = C_n^k = \frac{n!}{k!(n-k)!}.$$

Отсюда вытекает, что после приведения подобных членов выражение $x^k a^{n-k}$ войдет с коэффициентом $C_n^k = \frac{n!}{k!(n-k)!}$. Итак, мы доказали, что

$$\begin{aligned} (a+x)^n &= C_n^0 a^n + C_n^1 a^{n-1} x + \dots \\ &\dots + C_n^k a^{n-k} x^k + \dots + C_n^n x^n. \end{aligned} \quad (10.7)$$

Равенство (10.7) принято называть формулой бинома Ньютона. Если положить в этом равенстве $a = 1$, то получим

$$(1+x)^n = C_n^0 + C_n^1 x + \dots + C_n^k x^k + \dots + C_n^n x^n. \quad (10.8)$$

Мы видим, что $(1+x)^n$ является производящей функцией для чисел C_n^k , $k = 0, 1, \dots$. С помощью этой производящей функции можно сравнительно просто доказать многие свойства чисел C_n^k .

Ряд Ньютона

Мы назвали, как это обычно делают, формулу $(a+x)^n$ биномом Ньютона. Это наименование с точки зрения истории математики неверно. Формулу для $(a+x)^n$ хорошо знали среднеазиатские математики Омар Хайям, Гиясэдди и другие. В Западной Европе задолго до Ньютона она была известна Блэзу Паскалю. Заслуга же Ньютона была в ином — ему удалось обобщить формулу $(a+x)^n$ на случай нецелых показателей. Именно, он доказал, что если a — положительное число и $|x| < a$, то для любого действительного значения α имеет место равенство

$$\begin{aligned} (x+a)^\alpha &= a^\alpha + \alpha a^{\alpha-1} x + \frac{\alpha(\alpha-1)}{1 \cdot 2} a^{\alpha-2} x^2 + \dots \\ &\dots + \frac{\alpha(\alpha-1) \dots (\alpha-k+1)}{1 \cdot 2 \dots k} a^{\alpha-k} x^k + \dots \end{aligned} \quad (10.9)$$

Только теперь получилось не конечное число слагаемых, а бесконечный ряд. В случае, когда n — натуральное число, $(n-n)$ обращается в нуль. Но эта скобка входит в коэффициент всех членов, начиная с $(n+2)$ -го, и потому все эти члены разложения равны нулю. Поэтому при натуральном n ряд (10.9) превращается в конечную сумму.

Производящие функции и рекуррентные соотношения

Теория производящих функций тесно связана с рекуррентными соотношениями. Вернемся снова к делению многочленов. Пусть функции $f(x)$ и $\varphi(x)$ разложены в степенные ряды,

$$f(x) = a_0 + a_1 x + \dots a_n x^n + \dots$$

$$\varphi(x) = b_0 + b_1 x + \dots b_n x^n \dots$$

— два многочлена, причем $b_0 \neq 0$. Мы будем, кроме того, предполагать, что $n < m$, то есть, что алгебраическая дробь $\frac{f(x)}{\varphi(x)}$ правильна (в противном случае мы всегда можем выделить из нее целую часть). Мы знаем, что

если

$$\frac{f(x)}{\varphi(x)} = c_0 + c_1 x + \dots + c_k x^k + \dots, \quad (10.10)$$

TO

$$\begin{aligned} & a_0 + a_1x + \dots + a_nx^n = \\ & = (b_0 + b_1x + \dots + b_mx^m)(c_0 + c_1x + \dots + c_kx^k + \dots). \end{aligned}$$

Раскроем в правой части этого равенства скобки и сравним коэффициенты при одинаковых степенях x слева и справа. Сначала мы получим m соотношений такого вида:

$$\begin{aligned} b_0 c_0 &= a_0, \\ b_0 c_1 + b_1 c_0 &= a_1, \\ b_0 c_2 + b_1 c_1 + b_2 c_0 &= a_2, \\ \dots\dots\dots \\ b_0 c_{m-1} + b_1 c_{m-2} + \dots + b_{m-1} c_0 &= a_{m-1} \end{aligned} \tag{10.11}$$

(если $n < m - 1$, то мы считаем, что $a_{n+1} = \dots = a_{m-1} = 0$). А дальше все соотношения имеют один и тот же вид:

$$b_0 c_{m+k} + b_1 c_{m+k-1} + \dots + b_m c_k = 0, \quad k = 0, 1, \dots \quad (10.12)$$

(ведь в $f(x)$ нет членов, содержащих x^m, x^{m+1} и т.д.). Таким образом, коэффициенты $c_0, c_1, \dots, c_k, \dots$ ряда (10.10) удовлетворяют рекуррентному соотношению (10.12). Коэффициенты этого соотношения зависят лишь от знаменателя дроби. Числитель же дроби нужен для нахождения первых членов c_0, c_1, \dots, c_{m-1} рекуррентной последовательности.

Обратно, если дано рекуррентное соотношение (10.12) и заданы члены c_0, c_1, \dots, c_{m-1} , то мы сначала по формулам (10.11) вычислим значения a_0, \dots, a_{m-1} . А тогда производящей функцией для последовательности чисел $c_0, c_1, \dots, c_k, \dots$ является алгебраическая дробь

$$\frac{f(x)}{\varphi(x)} = \frac{a_0 + ax + \dots + a_{m-1}x^{m-1}}{b_0 + b_1x + \dots + b_mx^m}. \quad (10.13)$$

На первый взгляд кажется, что мы мало выиграли при замене рекуррентного соотношения производящей функции. Ведь все равно придется делить числитель на знаменатель, а это приведет к тому же самому рекуррентному соотношению (10.12). Но дело в том, что над дробью (10.13) можно выполнять некоторые алгебраические преобразования, а это облегчит отыскание чисел c_k .

Об едином нелинейном рекуррентном соотношении

При решении задачи о разбиении последовательности мы пришли к рекуррентному соотношению

$$T_n = T_0 T_{n-1} + T_1 T_{n-2} + \dots + T_{n-1} T_0, \quad (10.14)$$

где $T_0 = 1$. Покажем, как решить соотношение (10.14). Для этого составим производящую функцию.

$$f(x) = T_0 + T_1 x + T_2 x^2 + \dots + T_n x^n + \dots \quad (10.15)$$

Положим

$$F(x) \equiv x f(x) = T_0 x + T_1 x^2 + \dots + T_n x^{n+1} + \dots \quad (10.16)$$

и возведем $F(x)$ в квадрат. Мы получим, что

$$\begin{aligned} F^2(x) &= T_0^2 + (T_0 T_1 + T_1 T_0) x^3 + \dots \\ &\dots + (T_0 T_{n-1} + \dots + T_{n-1} T_0) x^{n+1} + \dots \end{aligned}$$

Но по рекуррентному соотношению (10.14),

$$T_0 T_{n-1} + \dots + T_{n-1} T_0 = T_n.$$

Значит,

$$F^2(x) = T_1 x^2 + T_2 x^3 + \dots + T_n x^{n+1}.$$

Полученный ряд есть не что иное, как $F(x) - T_0 x$; поскольку $T_0 = 1$, он равен

$$F^2(x) = F(x) - x. \quad (10.17)$$

Для функции $F(x)$ получилось квадратное уравнение (10.17). Решая его, находим, что

$$F(x) = \frac{1 - \sqrt{1 - 4x}}{2}.$$

Мы выбрали перед корнем знак минус, так как в противном случае при $x = 0$ мы имели бы $F(0) = 2$, а из разложения (10.16) видно, что $F(0) = 0$.

Лекция 11. Алгоритмы на абстрактных структурах данных

Введение. Стеки. Очереди. Связанные списки. Двоичные деревья.

Ключевые слова: стек, указатель стека, LIFO, очередь, FIFO, связанный список, однонаправленный связанный список, двунаправленный связанный список, граф, вершины графа, ребра графа, путь, связный граф, цикл, дерево, ориентированное дерево, двоичное дерево.

Введение

Н. Вирт определил программирование как алгоритм + структуры данных. При этом структура данных может не зависеть от конкретных языковых конструкций (абстрактная структура данных).

Рассмотрим некоторые основные структуры данных.

Стеки

Стеком называется одномерная структура данных, загрузка или увеличение элементов для которой осуществляется с помощью указателя стека в соответствии с правилом **LIFO** («last-in, first-out» — «последним введен, первым выведен»).

Указатель стека sp (steck pointer) содержит в любой момент времени индекс (адрес) текущего элемента, который является единственным элементом стека, доступным в данный момент времени для обработки.

Существуют следующие основные базисные операции для работы со стеком (для случая, когда указатель стека всегда задает ячейку, находящуюся непосредственно над его верхним элементом).

1. Начальная установка:
Sp:=1;
2. Загрузка элемента x в стек:
Stack[sp]:=x;
Sp:=sp+1;
3. Извлечение элемента из стека:
Sp:=sp-1;
X:=stack[sp];

4. Проверка на переполнение и загрузка элемента в стек:
 If $sp \leq sd$ then
 Begin $stack[sp] := x$; $sp := sp + 1$ end
 Else
 { переполнение };
 Здесь sd — размерность стека.
5. Проверка наличия элементов и извлечение элемента стека:
 If $sp > 1$ then
 Begin $sp := sp - 1$; $x := stack[sp]$ tnd
 Else
 { антипереполнение }
6. Чтение данных из указателя стека без извлечения элемента:
 $x := stack[sp - 1]$.

Программа 1. Работа со стеком.

{ Реализованы основные базисные операции для работы со стеком.
 Программа написана на языке программирования Turbo-Pascal }

```
uses crt, graph;
type PEI = ^EI;
  EI = record
    n: byte;
    next: PEI;
  end;

var ster: array[1..3] of PEI;
    number: byte;
    p: PEI;
    th, l: integer;
    i: integer;
    nhod: word;
    s: string;

procedure hod(n, f, t: integer);
begin
  if n > 1 then begin
    hod(n-1, f, 6-(f+t));
    hod(1, f, t);
    hod(n-1, 6-(f+t), t);
  end else begin
    p := ster[f];
```

```

ster[f]:=ster[f]^next;
p^.next:=ster[t];
ster[t]:=p;
inc(nhod);
str(nhod,s);
{*****}
setfillstyle(1,0);bar(0,0,50,10);
setcolor(2);outtextxy(0,0,s);
setfillstyle(1,0);setcolor(0);p:=ster[f];i:=1;
while p<>nil do begin p:=p^.next;inc(i);end;
fillellipse(160*f,460-(i-1)*th,(number-ster[t]^n+1)*l,10);
setfillstyle(1,4);setcolor(4);p:=ster[t];i:=1;
while p<>nil do begin fillellipse(160*t,460-(i-1)*th,(number-
ster[t]^n+1)*l,10);inc(i);p:=p^.next;end;
{*****}
{ readkey;}{delay(50);}
end;
end;

procedure start;
var i:integer;grD,grM: Integer;
begin
clrscr;write('Enter the number of rings, please. ');readln(number);
for i:=1 to 3 do ster[i]:=nil;
for i:=1 to number do begin new(p);
    p^.n:=i;p^.next:=ster[1];ster[1]:=p;end;
nhod:=0;
grD:=Detect;{InitGraph(grD,grM,'');}
InitGraph(grD,grM,'c:\borland\tp\bgi');
th:=20;l:=round(50/number);
setfillstyle(1,4);setcolor(4);
for i:=1 to number do begin fillellipse(160,460-(i-1)*th,(number-
i+1)*l,10);end;
end;

begin
start;
{readkey;}
hod(number,1,3);
{closegraph;}
end.

```

Программа 2. Ханойская башня.

{На стержне A в исходном порядке находится N дисков, уменьшающихся по размеру снизу вверх. Диски должны быть переставлены на стержень в исходном порядке при использовании в случае необходимости промежуточного стержня B для временного хранения дисков. В процессе перестановки дисков обязательно должны соблюдаться правила: одновременно может быть переставлен только один самый верхний диск (с одного из стержней на другой); ни в какой момент времени диск не может находиться на другом диске меньшего размера.

Программа реализована с помощью абстрактного типа данных — стек для произвольного числа дисков.

{Программа написана на языке программирования Turbo-Pascal}.

```
uses crt,graph;
type PEI=^EI;
  EI=record
    n:byte;
    next:PEI;
  end;

var ster:array[1..3] of PEI;
    number: byte;
    p:PEI;
    th,l: integer;
    i:integer;
    nhod:word;
    s:string;

procedure hod(n,f,t:integer);
begin
  if n>1 then begin
    hod(n-1,f,6-(f+t));
    hod(1,f,t);
    hod(n-1,6-(f+t),t);
  end else begin
    p:=ster[f];
    ster[f]:=ster[f]^next;
    p^.next:=ster[t];
    ster[t]:=p;
    inc(nhod);
    str(nhod,s);
    {*****}
```

```

setfillstyle(1,0);bar(0,0,50,10);
setcolor(2);outtextxy(0,0,s);
setfillstyle(1,0);setcolor(0);p:=ster[f];i:=1;
while p<>nil do begin p:=p^.next;inc(i);end;
fillellipse(160*f,460-(i-1)*th,(number-ster[t]^n+1)*l,10);
setfillstyle(1,4);setcolor(4);p:=ster[t];i:=1;
while p<>nil do begin fillellipse(160*t,460-(i-1)*th,(number-
ster[t]^n+1)*l,10);inc(i);p:=p^.next;end;
{*****}
{ readkey;}{delay(50);}
end;
end;

procedure start;
var i:integer;grD,grM: Integer;
begin
clrscr;write('Enter the number of rings, please.');
```

```

readln(number);
for i:=1 to 3 do ster[i]:=nil;
for i:=1 to number do begin new(p);p^.n:=i;
p^.next:=ster[1];ster[1]:=p;end;
nhod:=0;
grD:=Detect;{InitGraph(grD,grM,'')}
InitGraph(grD,grM,'c:\borland\tp\bgi');
th:=20;l:=round(50/number);
setfillstyle(1,4);setcolor(4);
for i:=1 to number do begin fillellipse(160,460-(i-1)*th,(number-
i+1)*l,10);end;
end;

begin
start;
{readkey;}
hod(number,1,3);
{closegraph;}
end.
```

Очереди

Очередь — одномерная структура данных, для которой загрузка или извлечение элементов осуществляется с помощью указателей начала извлечения (head) и конца (tail) очереди в соответствии с правилом **FIFO** («first-in, first-out» — «первым введен, первым выведен»).

1. Начальная установка:
Head:=1; tail:=1;
2. Добавление элемента x:
Queue[tail]:=x; tail:=tail+1;
If tail>qd then tail:=1;
Здесь qd — размерность очереди.
3. Исключение элемента x:
x:=queue[head]; head:=head+1;
if head>qd then head:=1;
4. Проверка переполнения очереди и включение в нее элемента:
Temp:=tail+1;
If temp>qd then temp:=1;
If temp=head then { переполнение }
Else begin queue[tail]:=x; tail:=temp end;
5. Проверка элементов и исключение элемента:
If head:=tail then
{очередь пуста}
else begin
x:=queue[head]; head:=head+1;
if head>qd then head:=1;
end;

Отметим, что при извлечении элемента из очереди все элементы могут также перемещаться на один шаг к ее началу.

Связанные списки

Связанный список представляет собой структуру данных, которая состоит из узлов (как правило, записей), содержащих указатели на следующий узел. Указатель, который ни на что не указывает, снабжается значением nil. Таким образом, в каждый элемент связанного списка добавляется указатель (звено связи).

Приведем основные базисные операции для работы с *однонаправленным связанным списком*.

1. Включение элемента после элемента:
Link[q]:=link[p];
Link[p]:=q;
Здесь q — индекс элемента, который должен быть вставлен в список после элемента с индексом p.

2. Исключение преемника элемента x :

```
If link[x] <> null then
```

```
Link[x] := [link[x]]
```

```
else
```

```
{Элемент  $x$  не имеет преемника};
```

Отметим, что элемент, следующий в списке за элементом x , называется преемником элемента x , а элемент, расположенный перед элементом x , называется предшественником элемента x . Если элемент x не имеет преемника, то содержащемуся в нем указателю присваивается значение nil .

3. Включение элемента y перед элементом x :

```
Prev := 0;
```

```
While (link[prev] <> nil) and (link[prev] <> x) do
```

```
Prev := link[prev];
```

```
If link[prev] = x then
```

```
Btgin link[prev] := y; link[y] := x end
```

```
Else
```

```
{Элемент  $x$  не найден};
```

```
Здесь link[0] является началом списка.
```

Отметим, что исключение последнего элемента из однонаправленного списка связано с просмотром всего списка.

В двунаправленном связанном списке каждый элемент имеет два указателя ($succlink$ — описывает связь элемента с преемником, $predlink$ — с предшественником).

Приведем основные базисные операции для работы с двунаправленным связанным списком.

Ответ 1. Включение y перед элементом x :

```
Succlink[y] := x;
```

```
Predlink[y] := predlink[x];
```

```
Succlink[predlink[x]] := y;
```

```
Predlink[x] := y;
```

Ответ 2. Включение элемента y после элемента x :

```
Succlink[y] := succlink[x];
```

```
Predlink[y] := x;
```

```
Predlink[succlink[x]] := y;
```

```
Succlink[x] := y;
```

Ответ 3. Исключение элемента x .

```
Predlink[succlink[x]] := predlink[x];
```

```
Succlink[predlink[x]] := succlink[x];
```

Программа 3. Список целых чисел.

{Создается список целых чисел. Числа выбираются случайным образом из интервала 0..9999, затем он упорядочивается, сначала - по возрастанию, затем - по убыванию.

Программа написана на языке программирования Turbo-Pascal

```

}
uses crt;
type TLink=^Link;
   Link=record
     v : integer;
     p, n : TLink
   end;

var i : integer;
    p, q, w : TLink;
    s1,s2,rs : TLink;

procedure Sort( sp : TLink; t : integer );
var temp : integer;
begin
  q:=sp;
  while q^.n<>nil do begin
    q:=q^.n;
    p:=sp;
    while p^.n<>nil do begin
      if (p^.v-p^.n^.v)*t>0 then begin
        temp:=p^.v;
        p^.v:=p^.n^.v;
        p^.n^.v:=temp;
      end;
      p:=p^.n;
    end;
  end;
end;

function CreatRndSpis(deep : integer):TLink;
begin
  new(q);
  for i:=1 to deep do begin
    if i=1 then begin
      p:=q;q^.p:=nil;

```

```

end;
q^.v:=random(9999);
new(q^.n);
q^.n^.p:=q;
q:=q^.n;
end;
q^.p^.n:=nil;
dispose(q);
CreatRndSpis:=p;
end;

```

```

function CreatSortDawnSpis(deep : integer):TLink;
begin
if deep<9999 then begin
new(q);
for i:=1 to deep do begin
if i=1 then begin
q^.p:=nil;p:=q;
end;
q^.v:=random(round(9999/deep))+round(9999*(1-i/deep));
new(q^.n);
q^.n^.p:=q;
q:=q^.n;
end;
q^.p^.n:=nil;
dispose(q);
end else p:=nil;
CreatSortDawnSpis:=p;
end;

```

```

procedure Show( s : TLink; sp: integer );
var i : integer;
begin
p:=s;
i:=1;
while p<>nil do begin
gotoxy(sp,i);write(' ' : 5); gotoxy(sp,i);writeln(p^.v);
p:=p^.n;
inc(i);
end;
end;

```



```
function min( c1, c2 : integer ) : integer;  
begin  
  case c1<c2 of  
    true : min:=c1;  
    false: min:=c2;  
  end;  
end;
```

```
function CreatConcSortUpSpis( sp1, sp2 : TLink ) : TLink;  
begin  
  q:=sp1;while q^.n<>nil do q:=q^.n;  
  w:=sp2;while w^.n<>nil do w:=w^.n;  
  new(p);
```

```
  CreatConcSortUpSpis:=p;  
  p^.p:=nil;  
  while(w<>nil)and(q<>nil)do begin  
    if(w<>nil)and(q<>nil)then begin  
      p^.v:=min(q^.v,w^.v);  
      case p^.v=q^.v of  
        true : q:=q^.p;  
        false: w:=w^.p;  
      end;  
      new(p^.n);  
      p^.n^.p:=p;  
      p^.n^.n:=nil;  
      p:=p^.n;  
    end;  
    if(w=nil)and(q<>nil)then begin  
      while q<>nil do begin  
        p^.v:=q^.v;q:=q^.p;  
        new(p^.n);  
        p^.n^.p:=p;  
        p^.n^.n:=nil;  
        p:=p^.n;  
      end;  
    end;  
    if(w<>nil)and(q=nil)then begin  
      while w<>nil do begin  
        p^.v:=w^.v;w:=w^.p;
```

```
new(p^.n);
p^.n^.p:=p;
p^.n^.n:=nil;
p:=p^.n;
end;
end;
end;
p^.p^.n:=nil;
dispose(p);
end;

begin
clrscr;
randomize;
s1:=CreatRndSpis(15);Sort(s1,-1);
s2:=CreatRndSpis(5);Sort(s2,-1);
rs:=CreatConcSortUpSpis(s1,s2);
Show(s1,10);
Show(s2,20);
Show(rs,30);
Sort(rs,-1);
Show(rs,40);
readln;
end.
```

Двоичные деревья

Основные определения и понятия о графах даются в лекции 12. В лекции 16 и 17 рассматриваются комбинаторные алгоритмы на графах. В данной лекции приведены несколько понятий, необходимых для описания абстрактной структуры данных, — двоичное дерево.

При решении многих задач математики используется понятие графа. **Граф** — набор точек на плоскости (эти точки называются **вершинами графа**), некоторые из которых соединены отрезками (эти отрезки называются **ребрами графа**). Вершины могут располагаться на плоскости произвольным образом. Причем неважно, является ли ребро, соединяющее две вершины, отрезком прямой или кривой линией. Важен лишь факт соединения двух данных вершин графа ребром. Примером графа может служить схема линий железных дорог. Если требуется различать вершины графа, их нумеруют или обозначают разными буквами. В изображении графа на плоскости могут появляться точки пересечения ребер, не явля-

ющиеся вершинами.

Говорят, что две вершины графа соединены *путем*, если из одной вершины можно пройти по ребрам в другую вершину. Путей между двумя данными вершинами может быть несколько. Поэтому обычно путь обозначают перечислением вершин, которые посещают при движении по ребрам. *Граф называется связным*, если любые две его вершины соединены некоторым путем.

Состоящий из различных ребер замкнутый путь называется *циклом*. Связный граф, в котором нет циклов, называется *деревом*. Одним из основных отличительных свойств дерева является то, что в нем любые две вершины соединены единственным путем. *Дерево называется ориентированным*, если на каждом его ребре указано направление. Следовательно, о каждой вершине можно сказать, какие ребра в нее входят, а какие — выходят. Точно так же о каждом ребре можно сказать, из какой вершины оно выходит, и в какую — входит. *Двоичное дерево* — это такое ориентированное дерево, в котором:

1. Имеется ровно одна вершина, в которую не входит ни одного ребра. Эта вершина называется корнем двоичного дерева.
2. В каждую вершину, кроме корня, входит одно ребро.
3. Из каждой вершины (включая корень) исходит не более двух ребер.

Граф задается аналогично спискам через записи и указатели.

Программа 4. Создание и работа с деревом.

```
//Алгоритм реализован на языке Turbo-C++.
//Вершины дерева задаются структурой: поле целых,
//поле для размещения адреса левого <<сына>>
//и поле для размещения адреса правого <<сына>>
//Значение целого выбирается случайным образом
//из интервала 0..99.
//Число уровней дерева равно N. В примере N = 5.
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
#define N 5
struct tree{int a;
    tree* left;
    tree* right;};
```

```
void postr(tree* root,int h)
{
    root->a=random(100);
    if (h!=0){
        if (random(3)){
            root->left=new(tree);
            postr(root->left,h-1);}
        else root->left=NULL;
        if (random(3))
            {root->right=new(tree);postr(root->right,h-1);}
        else root->right=NULL;}
    else {root->right=NULL;root->left=NULL;}
}
```

```
void DFS(tree* root)
{printf("%d ",root->a);
  if (root->left!=NULL) DFS(root->left);
  if (root->right!=NULL) DFS(root->right);}
```

```
void main()
{clrscr();
  randomize();
  tree* root1;
  root1=new(tree);
  postr(root1,N);
  DFS(root1);
  getch();
}
```

Лекция 12. Что такое граф? Определения и примеры

Введение. Представления. Связность и расстояние. Остовные деревья. Клики. Изоморфизм. Планарность. Вопросы и ответы.

Ключевые слова: конечный граф, ребро инцидентное вершинам, ориентированный граф, ребро e ориентированно из вершины v в вершину w , орграфы, неориентированный граф, петля, параллельные ребра, простой граф, матрица смежностей, матрица соединений, матрица весов, взвешенный граф, вес ребра, список ребер, структура смежности, матрица инцидентности, вершина-предшественник, вершина-последователь, вершины-соседи, смежные вершины, ребра смежны, простой путь, путь, длина пути, кратчайший путь, цикл, ациклический граф, подграф, индуцированный подмножеством вершин графа, неориентированный граф связан, точка сочленения, разделяющая вершина, двусвязный граф, неразделимый граф, двусвязная компонента, блок, ориентированный граф связан, тривиально связный граф, связная компонента, компонента, сильно связная компонента, несвязный граф, дерево, лес, остовное дерево, остовный лес, минимум остовных деревьев, минимальное остовное дерево, жадный алгоритм, алгоритм ближайшего соседа, полный граф, клика, изоморфные графы, планарный граф.

Введение

Множество самых разнообразных задач естественно формулируется в терминах графов. Так, например, могут быть сформулированы задачи составления расписаний в исследовании операций, анализа сетей в электротехнике, установления структуры молекул в органической химии, сегментации программ в программировании, анализа цепей Маркова в теории вероятностей. В задачах, возникающих в реальной жизни, соответствующие графы часто оказываются так велики, что их анализ неосуществим без ЭВМ. Таким образом, решение прикладных задач с использованием теории графов возможно в той мере, в какой возможна обработка больших графов на ЭВМ, и поэтому эффективные алгоритмы решения задач теории графов имеют большое практическое значение. В 16 и 17 лекциях мы излагаем несколько эффективных алгоритмов на графах и используем их для демонстрации некоторой общей техники решения задач на графах с помощью ЭВМ.

Конечный граф $G = (V, E)$ состоит из конечного множества вершин $V = \{v_1 v_2, \dots\}$ и конечного множества ребер $E\{e_1 e_2, \dots\}$. Каждому ребру соответствует пара вершин: если ребро (v, w) соответствует ребру e , то говорят, что e **инцидентно вершинам** v и w . Граф $G = (V, E)$ изображается следующим образом: каждая вершина представляется точкой и каждое ребро представляется отрезком линии, соединяющим его концевые вершины. **Граф** называется **ориентированным**, если пара вершин (v, w) , соответствующая каждому ребру, упорядочена. В таком случае говорят, что **ребро e ориентированно из вершины v в вершину w** , а направление обозначается стрелкой на ребре. Мы будем называть ориентированные графы **орграфами**. В **неориентированном графе** концевые вершины каждого ребра не упорядочены, и ребра не имеют направления. Ребро называется **петлей**, если оно начинается и кончается в одной и той же вершине. Говорят, что два **ребра параллельны**, если они имеют одну и ту же пару концевых вершин (и если они имеют одинаковую ориентацию в случае ориентированного графа). **Граф** называется **простым**, если он не имеет ни петель, ни параллельных ребер. Если не указывается противное, будем считать, что рассматриваемые графы являются простыми. Всюду в 16 и 17 лекции будем использовать символы $|V|$ и $|E|$ для обозначения соответственно числа вершин и числа ребер в графе $G = (V, E)$.

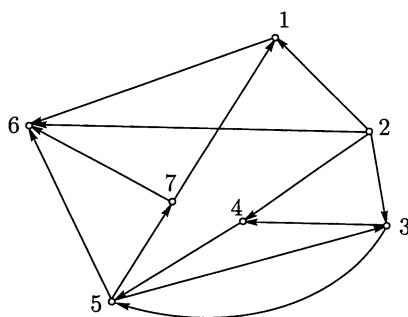
Представления

Наиболее известный способ представления графа на бумаге состоит в геометрическом изображении точек и линий. В ЭВМ граф должен быть представлен дискретным способом, причем возможно много различных представлений. Простота использования, так же как и эффективность алгоритмов на графе, зависит от подходящего выбора представления графа. Рассмотрим различные структуры данных для представления графов.

Матрица смежностей. Одним из наиболее распространенных машинных представлений простого графа является **матрица смежностей** или **соединений**. Матрица смежностей графа $G = (V, E)$ есть $|V| \times |V|$ -матрица $A = [a_{ij}]$, в которой $a_{ij} = 1$, если в G существует ребро, идущее из i -й вершины в j -ю, и $a_{ij} = 0$ в противном случае. Орграф и его матрица смежностей представлены на рис. 12.1.

Заметим, что в матрице смежностей петля может быть представлена соответствующим единичным диагональным элементом. Кратные ребра можно представить, позволив элементу матрицы быть больше 1, но это не принято, так как обычно удобно представлять каждый элемент матрицы одним двоичным разрядом.

Для задания матрицы смежностей требуется $|V|^2$ двоичных разрядов. У неориентированного графа матрица смежностей симметрична, и



G

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Рис. 12.1. Ориентированный граф и его матрица смежностей

для ее представления достаточно хранить только верхний треугольник. В результате экономится почти 50% памяти, но время вычислений может при этом немного увеличиться, потому что каждое обращение к a_{ij} должно быть заменено следующим: *if* $i > j$ *then* a_{ji} *else* a_{ij} . В случае представления графа его матрицей смежностей для большинства алгоритмов требуется время вычисления, по крайней мере пропорциональное $|V|^2$.

Матрица весов. Граф, в котором ребру (i, j) сопоставлено число w_{ij} , называется **взвешенным графом**, а число w_{ij} называется **весом ребра** (i, j) . В сетях связи или транспортных сетях эти веса представляют некоторые физические величины, такие как стоимость, расстояние, эффективность, емкость или надежность соответствующего ребра. Простой взвешенный граф может быть представлен своей матрицей весов $W = [w_{ij}]$, где w_{ij} есть вес ребра, соединяющего вершины i и j . Веса несуществующих ребер обычно полагают равными ∞ или 0 в зависимости от приложений. Когда вес несуществующего ребра равен 0, матрица весов является простым обобщением матрицы смежностей.

Список ребер. Если граф является разреженным, то возможно, что более эффектно представлять ребра графа парами вершин. Это представление можно реализовать двумя массивами $g = (g_1, g_2, \dots, g_{|E|})$ и $h =$

$(h_1, h_2, \dots, h_{|E|})$. Каждый элемент в массиве есть метка вершины, а i -е ребро графа выходит из вершины g_i и входит в вершину h_i . Например, орграф, изображенный на рис. 12.1, будет представляться следующим образом:

$$g = (1, 2, 2, 2, 2, 3, 3, 4, 5, 5, 5, 7, 7),$$

$$h = (6, 1, 3, 4, 6, 4, 5, 5, 3, 6, 7, 1, 6).$$

Ясно, что при этом легко представимы петли и кратные ребра.

Структура смежности. В ориентированном графе *вершина* y называется *последователем* другой вершины x , если существует ребро, направленное из x в y . Вершина x называется тогда *предшественником* y . В случае неориентированного графа две *вершины* называются *соседями*, если между ними есть ребро. Граф может быть описан его структурой смежности, то есть списком всех последователей (соседей) каждой вершины; для каждой вершины v задается $Adj(v)$ — список всех последователей (соседей) вершины v . В большинстве алгоритмов на графах относительный порядок вершин, смежных с вершиной v в $Adj(v)$, не важен, и в таком случае удобно считать $Adj(v)$ мультимножеством (или множеством, если граф является простым) вершин, смежных с v . Структура смежности орграфа, представленного на рис. 12.1, такова:

$v \quad Adj(v)$

1: 6

2: 1, 3, 4, 6

3: 4, 5

4: 5

5: 3, 6, 7

6:

7: 1, 6

Если для хранения метки вершины используется одно машинное слово, то структура смежности ориентированного графа требует $|V| + |E|$ слов. Если граф неориентированный, нужно $|V| + 2|E|$ слов, так как каждое ребро встречается дважды.

Структуры смежности могут быть удобно реализованы массивом из $|V|$ линейно связанных списков, где каждый список содержит последователей некоторой вершины. Поле данных содержит метку одного из последователей, и поле указателей указывает следующего последователя. Хранение списков смежности в виде связанного списка желательно для алгоритмов, в которых в графе добавляются или удаляются вершины.

Матрица инцидентности — M задает граф: $m_{ij}=1$, если ребро j выходит из вершины i , $m_{ij} = -1$, если ребро j входит в вершину i , и $m_{ij}=0$ в остальных случаях.

Связность и расстояние

Говорят, что *вершины* x и y в графе *смежны*, если существует ребро, соединяющее их. Говорят, что два *ребра смежны*, если они имеют общую вершину. *Простой путь*, или для краткости, просто *путь*, записываемый иногда как (v_1, v_2, \dots, v_k) , — это последовательность смежных ребер $(v_1, v_2), (v_2, v_3), \dots, (v_{k-2}, v_{k-1}), (v_{k-1}, v_k)$, в которой все вершины v_1, v_2, \dots, v_k различны, исключая, возможно, случай $v_1 = v_k$. В орграфе этот путь называется ориентированным из v_1 в v_k , в неориентированном графе он называется путем между v_1 и v_k . Число ребер в пути называется *длиной пути*. Путь наименьшей длины называется *кратчайшим путем*. Замкнутый путь называется *циклом*. Граф, который не содержит циклов, называется *ациклическим*.

Подграф графа $G = (V, E)$ есть граф, вершины и ребра которого лежат в G . Подграф G , *индуцированный подмножеством* S множества V вершин графа G , — это подграф, который получается в результате удаления всех вершин из $V - S$ и всех ребер, инцидентных им.

Неориентированный граф G *связен*, если существует хотя бы один путь в G между каждой парой вершин v_i и v_j . *Ориентированный граф* G *связен*, если неориентированный граф, получающийся из G путем удаления ориентации ребер, является связным. *Граф*, состоящий из единственной изолированной вершины, является (*тривиально*) *связным*. Максимальный связный подграф графа G называется *связной компонентой* или просто *компонентой* G . *Несвязный граф* состоит из двух или более компонент. Максимальный сильно связный подграф называется *сильно связной компонентой*.

Иногда недостаточно знать, что граф связан; нас может интересовать, насколько «сильно связан» связный граф. Например, связный граф может содержать вершину, удаление которой вместе с инцидентными ей ребрами разъединяет оставшиеся вершины. Такая вершина называется *точкой сочленения* или *разделяющей вершиной*. Граф, содержащий точку сочленения, называется *разделимым*. Граф без точек сочленения называется *двусвязным* или *неразделимым*. Максимальный двусвязный подграф графа называется *двусвязной компонентой* или *блоком*.

Большинство основных вопросов о графах касается связности, путей и расстояний. Нас может интересовать вопрос, является ли граф связным; если он связан, то может оказаться нужным найти кратчайшее расстояние между выделенной парой вершин или определить кратчайший путь между ними. Если граф несвязен, то может потребоваться найти все его компоненты. В нашем курсе строятся алгоритмы для решения этих и других подобных вопросов.

Остовные деревья

Связный неориентированный ациклический граф называется *деревом*, множество деревьев называется *лесом*. В связном неориентированном графе G существует по крайней мере один путь между каждой парой вершин; отсутствие циклов в G означает, что существует самое большое один такой путь между любой парой вершин в G . Поэтому, если G — дерево, то между каждой парой вершин в G существует в точности один путь. Рассуждение легко обратимо, и поэтому неориентированный граф G будет деревом тогда и только тогда, если между каждой парой вершин в G существует в точности один путь. Так как наименьшее число ребер, которыми можно соединить n вершин, равно $n - 1$ и дерево с n вершинами содержит в точности $n - 1$ ребер, то деревья можно считать минимально связными графами. Удаление из дерева любого ребра превращает его в несвязный граф, разрушая единственный путь между по крайней мере одной парой вершин.

Особый интерес представляют *остовные деревья* графа G , то есть деревья, являющиеся подграфами графа G и содержащие все его вершины. Если граф G несвязен, то множество, состоящее из остовных деревьев каждой компоненты называется *остовным лесом* графа. Для построения остовного дерева (леса) данного неориентированного графа G , мы последовательно просматриваем ребра G , оставляя те, которые не образуют циклов с уже выбранными.

Во взвешенном графе $G = (V, E)$ часто интересно определить остовное дерево (лес) с минимальным общим весом ребер, то есть дерево (лес), у которого сумма весов всех его ребер минимальна. Такое дерево *называется минимумом остовных деревьев* или *минимальное остовное дерево*. Другими словами, на каждом шаге мы выбираем новое ребро с наименьшим весом (наименьшее ребро), не образующее циклов с уже выбранными ребрами; этот процесс продолжаем до тех пор, пока не будет выбрано $|V| - 1$ ребер, образующих остовное дерево T . Этот процесс известен как *жадный алгоритм*.

Жадный алгоритм может быть выполнен в два этапа. Сначала ребра сортируются по весу и затем строится остовное дерево путем выбора наименьших из имеющихся в распоряжении ребер.

Существует другой метод получения минимума остовных деревьев, который не требует ни сортировки ребер, ни проверки на цикличность на каждом шаге, — так называемый *алгоритм ближайшего соседа*. Мы начинаем с некоторой произвольной вершины a в заданном графе. Пусть (a, b) — ребро с наименьшим весом, инцидентное a ; ребро (a, b) включается в дерево. Затем среди всех ребер, инцидентных либо a , либо b , выбираем ребро с наименьшим весом и включаем его в частично построен-

ное дерево. В результате этого в дерево добавляется новая вершина, например, s . Повторяя процесс, ищем наименьшее ребро, соединяющее a , b или s с некоторой другой вершиной графа. Процесс продолжается до тех пор, пока все вершины из G не будут включены в дерево, то есть пока дерево не станет остовным.

Наихудшим для этого алгоритма будет случай, когда G — *полный граф* (то есть когда каждая пара вершин в графе соединена ребром); в этом случае для того, чтобы найти ближайшего соседа, на каждом шаге нужно сделать максимальное число сравнений. Чтобы выбрать первое ребро, мы сравниваем веса всех $|V| - 1$ ребер, инцидентных вершине a , и выбираем наименьшее; этот шаг требует $|V| - 2$ сравнений. Для выбора второго ребра мы ищем наименьшее среди возможных $2(|V| - 2)$ ребер (инцидентных a или b) и делаем для этого $2(|V| - 2) - 1$ сравнений. Таким образом, ясно, что для выбора i -го ребра требуется $i(|V| - i) - 1$ сравнений, и поэтому в сумме потребуются

$$\sum_{i=1}^{|V|-1} [i(|V| - i) - 1] = \frac{1}{6} |V|^3 + O(|V|^2)$$

сравнений для построения минимума остовных деревьев.

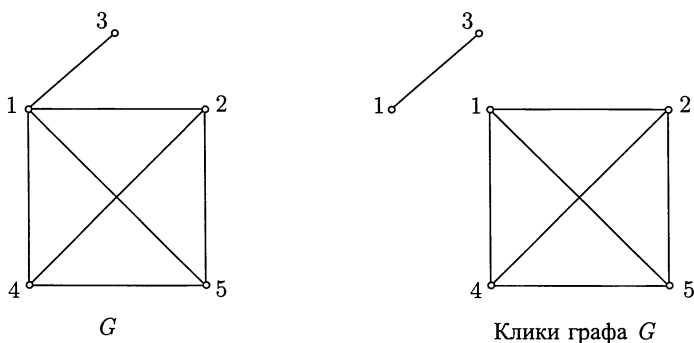
Клики

Максимальный полный подграф графа G называется *кликой* графа G ; другими словами, клика графа G есть подмножество его вершин, такое, что между каждой парой вершин этого подмножества существует ребро и, кроме того, это подмножество не принадлежит никакому большему подмножеству с тем же свойством. Например, на рис. 12.2 показан граф и его клики.

Клики графа представляют «естественные» группировки вершин, и определение клик графа полезно в кластерном анализе в таких областях, как информационный поиск и социология.

Изоморфизм

Два графа $G_A = (V_A, E_A)$, $G_B = (V_B, E_B)$ называются *изоморфными*, если существует взаимно однозначное соответствие $f : V_A \rightarrow V_B$, такое, что $(v, w) \in E_A$ тогда и только тогда, если $(f(v), f(w)) \in E_B$, то есть существует соответствие между вершинами графа G_A и вершинами графа G_B , сохраняющее отношение смежности. Например, на рис. 12.2 показаны два изоморфных орграфа: вершины a, b, c, d, e, f в орграфе G_2 соответствуют вершинам 2, 3, 6, 1, 4, 5 в указанном порядке в орграфе G_1 .

Рис. 12.2. Граф G и все его клики

Вообще говоря, между V_A и V_B может быть более чем одно соответствие, и на рис. 12.3 графы имеют на самом деле второй изоморфизм: a, b, c, d, e, f соответствуют в указанном порядке вершинам 2, 3, 6, 1, 5, 4. Изоморфные графы отличаются только метками вершин, в связи с чем задача определения изоморфизма возникает в ряде практических ситуаций, таких, как информационный поиск и определение химических соединений.

Заметим, что можно ограничиться орграфами. Любой неориентированный граф превращается в орграф заменой каждого ребра двумя противоположно направленными ребрами. Два полученных таким образом орграфа, очевидно, изоморфны тогда и только тогда, если изоморфны исходные графы.

Планарность

Граф называют планарным, если существует такое изображение на плоскости его вершин и ребер, что:

- каждая вершина v изображается отдельной точкой v' на плоскости;
- каждое ребро (v, w) изображается простой кривой, имеющей концевые точки (v', w') ;
- эти кривые пересекаются только в общих концевых точках. Задача определения того, можно ли изобразить граф на плоскости без пересечения ребер, имеет большой практический интерес (например, при конструировании интегральных схем или печатных плат необходимо выяснить, можно ли окончательную схему вложить в плоскость).

Определение планарности графа отличается от других рассмотрен-

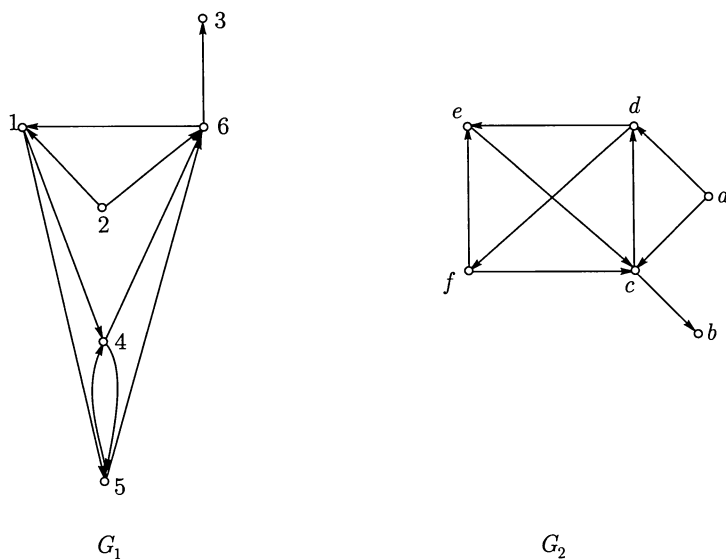


Рис. 12.3. Изоморфные орграфы

ных нами задач, поскольку при изображении точек и линий на плоскости приходится больше иметь дело с непрерывными, а не с дискретными величинами. Взаимосвязь между дискретными и непрерывными аспектами планарности заинтересовала математиков и привела к различным характеристикам планарных графов. С точки зрения математики эти характеристики изящны, но эффективных алгоритмов определения планарности они не дают. Наиболее успешный подход к определению планарности состоит просто в том, что граф разбивается на подграфы и затем делается попытка разместить его на плоскости, добавляя подграфы один за другим и сохраняя при размещении планарность.

Вначале сделаем несколько простых, но полезных наблюдений. Поскольку орграф планарен тогда и только тогда, если планарен соответствующий неориентированный граф, полученный игнорированием направления ребер, то достаточно рассматривать только неориентированные графы, поскольку неориентированный граф планарен тогда и только тогда, если все его двусвязные компоненты планарны. Поэтому если неориентированный граф является разделимым, мы можем разложить его на двусвязные компоненты и рассматривать их отдельно. Наконец, поскольку параллельные ребра и петли всегда можно добавить к графу

или удалить из него без нарушения свойства планарности, нам достаточно рассматривать только простые графы. Поэтому при определении планарности будем предполагать, что граф неориентированный, простой и двусвязный.

Наша основная стратегия состоит прежде всего в том, чтобы в графе G найти цикл C , разместить C на плоскости в виде простой замкнутой кривой, разложить оставшуюся часть $G - C$ на непересекающиеся по ребрам пути и затем попытаться разместить каждый из этих путей либо целиком внутри C , либо целиком вне C . Если нам удалось разместить так весь граф G , то он планарен, в противном случае он непланарен. Трудность этого способа заключается в том, что при размещении путей можно выбирать либо внутренность, либо внешность C , и мы должны проконтролировать, чтобы неправильный выбор области размещения на ранней стадии не устранил возможности размещения последующих путей, — это могло бы привести нас к неверному заключению, что планарный граф непланарен.

Лекция 13. Поиск

Введение. Поиск и другие операции над таблицами. Последовательный поиск. Логарифмический поиск в статических таблицах. Бинарный поиск. Оптимальные деревья бинарного поиска. Логарифмический поиск в динамических таблицах. Сбалансированные сильно ветвящиеся деревья. Вопросы и ответы.

Ключевые слова: имена, пространство имен, естественный порядок, табличный порядок, запись, поле, справочник, указатель, динамическая таблица, статическая таблица, бинарный поиск, дерево бинарного поиска, сбалансированное сильно ветвящееся дерево порядка m .

Введение

Задача поиска является фундаментальной в комбинаторных алгоритмах, так как она формулируется в такой общности, что включает в себя множество задач, представляющих практический интерес. При самой общей постановке «Исследовать множество T с тем чтобы найти элемент, удовлетворяющий некоторому условию C », о задаче поиска едва ли можно сказать что-либо стоящее. Удивительно, однако, что достаточно незначительных ограничений на структуру множества T , чтобы задача стала интересной: возникает множество разнообразных стратегий поиска различной степени эффективности. Мы сделаем некоторые предположения о структуре множества T , позволяющие исследовать T $O(\log n)$ или $O(1)$. Большинство алгоритмов поиска попадает в одну из трех категорий, характеризующихся временем поиска $O(n)$.

Поиск и другие операции над таблицами

Любой способ поиска оперирует с элементами, которые будем называть *именами*, взятыми из множества имен S — оно называется *пространством имен*. Это пространство имен может быть конечным или бесконечным. Самыми распространенными пространствами имен являются множества целых чисел с их числовым порядком (нумерацией), и множества последовательностей символов над некоторым конечным алфавитом с их лексикографическим (то есть словарным) порядком. Каждый из алгоритмов поиска, обсуждаемых в этой лекции, основан на одном из трех следующих предположений о пространстве S .

Предположение 1. На S определен линейный порядок, называемый **естественным порядком** и обозначаемый знаком $<$. Такой порядок имеет следующие свойства.

1. Любые два элемента $x, y \in S$ сравнимы, то есть должно выполняться в точности одно из трех условий: $x < y$, $x = y$, $y < x$.

2. Порядок обладает транзитивностью, то есть если $x < y$ и $y < z$, то $x < z$ для любых элементов $x, y, z \in S$. Мы используем обозначения $>$, \leq , \geq в очевидном смысле. При анализе эффективности алгоритма поиска полагаем, что исход ($<$, $=$ или $>$) зависит от сравнения.

Предположение 2. Каждое имя в S есть последовательность символов или цифр над конечным линейно упорядоченным алфавитом $A = \{a_1, a_2, \dots, a_c\}$. Естественным порядком на S является лексикографический порядок, индуцированный линейным порядком на A . Мы полагаем, что исход ($<$, $=$ или $>$) сравнения двух символов (не имен) получается за время, не зависящее от $|S|$ или n .

Предположение 3. Имеется функция $h: S \rightarrow \{0, 1, \dots, m-1\}$, которая равномерно отображает пространство имен S в множество $\{0, 1, \dots, m-1\}$, то есть все целые $i, 0 \leq i \leq m-1$, приблизительно с одинаковой частотой являются образами имен из S при отображении h . Мы полагаем, что функция h не зависела от $|S|$, это с теоретической точки зрения выглядит шатко, но с практической — довольно реально.

Как уже было отмечено, поиск производится не в самом пространстве S имен, а в конечном подмножестве $T = \{x_1, x_2, \dots, x_n\}$ множества S , называемом **таблицей**. На большинстве таблиц, которые мы рассматриваем, определен линейный порядок, называемый **табличным порядком**: ему соответствует нижний индекс имени (то есть x_1 есть первое имя в таблице, x_2 — второе и тому подобное). Табличный порядок часто совпадает с естественным порядком, определенным на пространстве имен, однако такое совпадение не обязательно.

Мощность таблицы T обычно намного меньше, чем мощность пространства имен S , даже если S конечно.

Представление о таблице как об упорядоченном множестве имен существенно для многих целей. Но иногда бывает необходимо рассматривать таблицу как множество ячеек, каждая из которых может содержать одно имя. Например, если процесс включения нового имени рассматривать более подробно, то обнаружится, что сначала нужно расширить таблицу добавлением ячейки, для того чтобы заготовить место для новой записи: только потом туда заносится новое имя.

Мы будем предполагать, что имена появляются в таблице не больше одного раза (исключение составляет переходный период, в течение которого заносится новое имя; в таблице допускается два вхождения одного

имени). В большинстве случаев вследствие такого предположения таблица с n именами имеет ровно n ячеек. Однако важный класс алгоритмов, основанных на вычислении адреса, опирается на предположение о том, что таблица содержит больше ячеек, чем имен. Эти алгоритмы должны четко принимать во внимание наличие пустых ячеек.

Существует ряд синонимов для объектов, именуемых здесь таблицей и именем. В обработке данных существуют файлы, элементами которых являются *записи*; каждая запись есть последовательность *полей*, одно из которых (участвующее в поиске) называется *ключом*. Если сам файл проходимся во время поиска, «файл» и «ключ» представляют собой то, что мы называем «таблицей» и «именем» соответственно. (Эта терминология несколько двусмысленна, поскольку понятие «ключ» можно отнести либо к самой ячейке, либо к содержимому ячейки.) Однако при поиске в большом файле часто не подразумевается просмотр самого файла; вместо этого поиск осуществляется на *справочнике* или *указателе* файла. При успешном поиске найденная в указателе отдельная запись указывает на соответствующую запись в файле. В таком типе организации файлов нашему понятию таблицы соответствует указатель или справочник.

Мы рассматриваем только четыре табличные операции: поиск, включение, исключение и распечатка. Подробное определение того, что должны делать эти операции над таблицей $T = \{x_1, x_2, \dots, x_n\}$, зависит от структуры данных, использованной для реализации таблицы.

- поиск z : если $z \in T$, то отметить его указателем, то есть переменной i присвоить такое значение, что $z = x_i$; в противном случае указать, что $z \notin T$;
- включение z : если $z \notin T$, то поместить его на соответствующее место.

Включение в общем случае предполагает прежде всего поиск соответствующего места, поэтому иногда удобно разделить операцию на две фазы. Сначала используем процедуру поиска для отыскания места, куда должно быть помещено z , и затем помещаем z на это место.

- включить z включить z сразу после имени x_{i-1} .
- на i -е место: До включения $T = \{x_1, \dots, x_{i-1}, \dots, x_{i-1}z, x_i, \dots, x_n\}$.
- исключить z : если $z \in T$, то исключить его.

Как и включение, исключение иногда реализуется процедурой поиска для получения места z и последующей процедурой:

- исключить исключить x_i из T .
- с i -го места: До исключения $T = \{x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n\}$
После исключения $T = \{x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n\}$

Таблица, в которой осуществляются включения или исключения, называется *динамической*; в противном случае она носит название *статической*.

Последней операцией, которую мы будем рассматривать для каждой табличной структуры, является

распечатка: напечатать все имена из T в их естественном порядке

Среди всех операций, которые можно производить над таблицами, четыре, рассматриваемые в этой лекции (поиск, включение, исключение и распечатка), и сортировка (лекции 14, 15) — наиболее важные.

Последовательный поиск

Под последовательным поиском мы подразумеваем исследование имен в том порядке, в котором они встречаются в таблице. При таком поиске в таблице в худшем случае получается просмотр всей таблицы; даже в среднем последовательный поиск имеет тенденцию к использованию числа операций, пропорционального n . Для больших таблиц его не следует относить к методам быстрого поиска, поскольку последовательный поиск асимптотически гораздо медленнее других алгоритмов, описанных в этой лекции. Несмотря на его низкие асимптотические возможности, имеется ряд причин, по которым этот метод следует обсудить вначале. Во-первых, хотя идея его проста, он позволяет нам ввести важные понятия и методы, применимые к поиску вообще. Во-вторых, последовательный поиск является единственным методом поиска, применимым к отдельным устройствам памяти и к тем таблицам, которые строятся на пространстве имен без линейного порядка. Наконец, последовательный поиск является быстрым для достаточно малых таблиц и для больших таблиц, организованных иерархическим способом: более быстрый метод используется для исследования окрестности верхушки иерархии, а последовательный поиск — для подтаблицы на нижнем уровне иерархии.

Для последовательного поиска по таблице $T = \{x_1, x_2, \dots, x_n\}$ мы предполагаем, что имеется указатель i , значение которого принадлежит отрезку $1 \leq i \leq n$ или, возможно, $0 \leq i \leq n + 1$. Над этим указателем разрешается производить только следующие операции; первоначальное присваивание ему значения 1 или n (или, если удобнее, 0 или $n + 1$), увеличение и /или уменьшение его на единицу и сравнение его с 0, 1, n или $n + 1$. При таких соглашениях наиболее очевидный алгоритм поиска в таблице T первого вхождения данного имени z имеет вид алгоритма 13.1. Здесь, как и во всех других алгоритмах поиска, изложенных в настоящей лекции, мы полагаем, что алгоритм останавливается немедленно по отыскании z или установлении, что z в таблице нет.

```

for  $i = 1$  to  $n$  do if  $z = x_i$  then { найдено:  $i$  указывает на  $z$  }
{ не найдено:  $z$  не входит в  $T$  }

```

Алгоритм 13.1. Последовательный поиск

Рассмотрим некоторые аспекты эффективности последовательного поиска, начиная со стандартных методов программирования. В программе, построенной в виде одного цикла, как алгоритм 13.1, любое значительное ускорение должно быть следствием улучшения кода в цикле. Для того чтобы увидеть, какие операции выполняются внутри цикла, необходимо переписать алгоритм 13.1 в форме, близкой к языку машины:

```

 $i \leftarrow 1$ 
цикл:  if  $z = x_i$  then найдено
        if  $i = n$  then не найдено
         $i \leftarrow i + 1$ 
        goto цикл

```

За каждую итерацию выполняется до четырех команд: два сравнения, одна операция увеличения и одна передача управления.

Для ускорения внутреннего цикла общим приемом является добавление в таблицу специальных строк, которые делают необязательной явную проверку того, достиг ли указатель границ таблицы. Это можно сделать в алгоритме 13.1. Если перед поиском мы добавим искомое имя z в конце таблицы, то цикл всегда будет завершаться отысканием вхождения z ; таким образом, нам не нужно в цикле каждый раз делать проверку $i = n$. В конце цикла проверка условия $i > n$, выполняемая лишь однажды, говорит о том, является ли найденное вхождение z истинным или специальным элементом таблицы. Это демонстрируется в алгоритме 13.2.

```

 $x_{n+1} \leftarrow z$ 
 $i \leftarrow 1$ 
while  $z \neq x_i$  do  $i \leftarrow i + 1$ 
if  $i \leq n$  then { найдено:  $i$  указывает на  $z$  }
               else { не найдено }

```

Алгоритм 13.2. Улучшенный последовательный поиск

Улучшение алгоритма 13.1 будет наиболее очевидным, если мы перепишем алгоритм 13.2 в тех же близких к языку машины обозначениях, которые использовались раньше:

```

 $x_{n+1} \leftarrow z$ 
 $i \leftarrow 1$ 
ЦИКЛ:    $if\ z = x_i\ then\ goto\ \text{возможно}$ 
         $i \leftarrow i + 1$ 
         $goto\ \text{цикл}$ 
возможно:  $if\ i \leq n\ then\ \{ \text{найдено: } i \text{ указывает на } z \}$ 
           $else\ \{ \text{не найдено} \}$ 

```

При каждой итерации выполняются лишь три действия вместо четырех, как это было в алгоритме 13.1. Таким образом, в большинстве вычислительных устройств цикл в алгоритме 13.2 будет выполняться гораздо быстрее, чем в алгоритме 13.1, и поскольку скорость цикла определяет скорость всей программы, такое же сравнение имеет место для двух программ.

Печально то, что фокус с добавлением z в конец таблицы перед поиском удастся, только если мы имеем прямой непосредственный доступ к концу таблицы. Это возможно, если таблица хранится в памяти с произвольным доступом, но невозможно в общем случае, когда используется связанное размещение или память с последовательным доступом.

Единственным недостатком алгоритма 13.2 является то, что при безуспешном поиске (поиске имен, которых нет в таблице) всегда просматривается вся таблица. Если такой поиск возникает часто, то имена надо хранить в естественном порядке; это позволяет завершать поиск, как только при просмотре попало первое имя, большее или равное аргументу поиска. В этом случае в конец таблицы следует добавить фиктивное имя ∞ для того, чтобы гарантировать выполнение условия завершения (∞ — это новое имя, которое по предположению больше любого имени из пространства имен S). Таким образом получаем алгоритм 13.3.

```

 $x_{n+1} \leftarrow \infty$ 
 $i \leftarrow 1$ 
     $while\ z > x_i\ do\ i \leftarrow i + 1$ 
 $if\ z = x_i\ then\ \{ \text{найдено: } i \text{ указывает на } z \}$ 
     $else\ \{ \text{не найдено} \}$ 

```

Алгоритм 13.3. Последовательный поиск по таблице, хранимой в естественном порядке

Логарифмический поиск в статических таблицах

Мы говорим о логарифмическом времени поиска, как только возникает возможность за время c , не зависящее от n , последовательно свести

задачу поиска в таблице, содержащей n имен, к задаче поиска в таблице, содержащей не более αn имен, где $\alpha < 1$ — константа. В этом случае время $t(n)$, требующееся для поиска в таблице с n именами, удовлетворяет рекуррентному соотношению

$$t(n) = c + t(\alpha n),$$

решение, которого имеет вид

$$t(n) = c \log_{1/\alpha} n + b,$$

где b определяется начальными условиями и коэффициент c при логарифме есть время, требуемое для уменьшения размера таблицы от n до αn .

Самыми распространенными предположениями, которые дают возможность уменьшить размер таблицы от n до αn за время, не зависящее от n , являются предположения о том, что пространство имен S линейно упорядочено и что сравнение двух имен x, y из S (для определения $x < y, x = y, x > y$) есть элементарная операция, требующая постоянного количества времени, не зависящего от n . В результате время, необходимое для большинства логарифмических алгоритмов поиска, естественно измеряется числом сравнений (с тремя исходами) пар имен. Для некоторых алгоритмов, однако, более естественны сравнения с большим, но фиксированным числом исходов.

В этом разделе мы рассматриваем только статические таблицы, то есть таблицы, в которых включение и исключение либо не встречаются, либо так редки, что когда они появляются, строится новая таблица. Динамические структуры таблиц, допускающие логарифмическое время поиска, так же как и эффективные алгоритмы включения и исключения, обсуждаются в конце этой лекции. Для статических таблиц нужно обсудить лишь алгоритмы поиска и построения таблицы. Алгоритмы поиска достаточно просты, но некоторые из алгоритмов построения таблицы сложны. Такая ситуация возникает потому, что в случае статической таблицы разумно считать частоты обращения известными, и, может быть, стоит затратить существенные усилия на построение оптимальной таблицы — таблицы с минимальным средним временем поиска (относительно данных частот обращения). Алгоритмы построения таблиц и их анализ являются наиболее важными темами этого раздела.

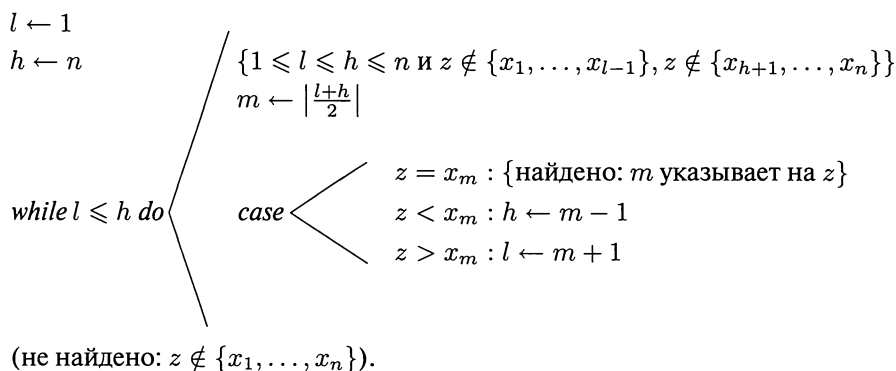
Бинарный поиск

Когда ячейки таблицы последовательно распределены в памяти с произвольным доступом и имена хранятся в таблице в их естественном

порядке, возможен **бинарный поиск** — один из наиболее широко используемых методов поиска. Идея этого метода состоит в том, чтобы искать имя z в интервале, крайними точками которого являются два заданных указателя l (для «низа») и h (для «верха»). Новый указатель m (для «средней точки») устанавливается где-то около середины интервала, и либо z с именем в этой ячейке сводит интервал поиска к одному из интервалов $[l, m - 1]$ или $[m + 1, h]$. Если интервал становится пустым, поиск завершается безуспешно.

Для получения логарифмического времени поиска существенно устанавливать указатель m за время, не зависящее от длины интервала; это требование делает непригодным бинарный поиск на большинстве вспомогательных запоминающих устройств. Требование, чтобы m помещалось точно в середине интервала, несущественно, хотя выбор средней точки в качестве m обычно дает самый эффективный алгоритм. В некоторых частных случаях полезно разбить интервал на подинтервалы длины $\alpha(h - l + 1)$ и $(1 - \alpha)(h - l + 1)$ для фиксированного значения α , отличного от $\frac{1}{2}$. Когда таблица размещена не последовательно, а хранится в виде списка древовидной структуры, доля α должна, вероятно, меняться от интервала к интервалу.

Бинарный поиск по идее прост, но с деталями условия завершения поиска нужно обращаться осторожно. Частные случаи $h - l = 1$ и $h - l = 0$ требуют пристального внимания в любой программе бинарного поиска. В алгоритме 13.4 эти случаи обрабатываются тем же кодом, что и в общем случае, и поучительно посмотреть, как это делается, проследив за выполнением алгоритма для $n = 2$ и $n = 1$.



Алгоритм 13.4. Бинарный поиск имени z в таблице $\{x_1, \dots, x_n\}$, хранящейся в естественном порядке.

Корректность алгоритма 13.4 следует из утверждения, данного в комментарии в начале тела цикла. Он устанавливает, что если z находится где-либо в таблице, то оно должно находиться в интервале $[l, h]$; иначе говоря, при нашем предположении, что имя появляется в таблице не больше одного раза, утверждается, что z не встречается ни в интервале $[1, l - 1]$, ни в интервале $[h + 1, n]$. Это утверждение очевидно первый раз, когда мы входим в цикл при $l = 1$ и $h = n$, и непосредственно по индукции проверяется, что оно выполняется при каждом проходе через цикл. Когда мы выходим из цикла, то должно быть $l > h$, и поэтому утверждение принимает вид $z \notin \{x_1, \dots, x_{l-1}\}$ и $z \notin \{x_{h+1}, \dots, x_n\}$, откуда следует, что $z \notin \{x_1, \dots, x_n\}$.

Оптимальные деревья бинарного поиска

Бинарный поиск в последовательно распределенной таблице (алгоритм 13.4.) обеспечивает очень быстрое нахождение имен, которые являются средними точками на раннем этапе процесса деления пополам, именно имен, близких к вершине дерева $T(1, n)$. Таким образом, любое имя в таблице можно выбрать примерно за $\lg n$ сравнений.

На практике в большинстве таблиц встречаются имена, к которым обращаются гораздо чаще, чем к другим, и «привилегированные» места в таблице разумно постараться использовать для наиболее часто вызываемых имен, а не для имен, выбранных для этих мест в результате бинарного поиска. Это невозможно осуществить для последовательно распределенных таблиц, поскольку место имени определяется его положением относительно естественного порядка имен в таблице. Введем структуру данных, легко приспособляемую как к месту бинарного поиска, так и к возможности выделять точки, в которых таблица делится на две части.

Деревом бинарного поиска над именами x_1, x_2, \dots, x_n называется расширенное бинарное дерево, все внутренние узлы которого помечены различными именами из списка x_1, x_2, \dots, x_n таким образом, что симметричный порядок узлов совпадает с естественным порядком. Каждый из $n + 1$ внешних узлов соответствует промежутку в таблице. На рис. 13.1 показаны четыре различных дерева бинарного поиска на множестве имен $\{A, B, C, D\}$. Деревья (а) и (б) — вырожденные, поскольку они по существу являются линейными списками, которые должны просматриваться последовательно.

Поиск имени z в дереве бинарного поиска осуществляется путем сравнения z с именем, стоящим в корне. Тогда

1. Если корня нет (дерево пусто), то z в таблице отсутствует и поиск завершается безуспешно.

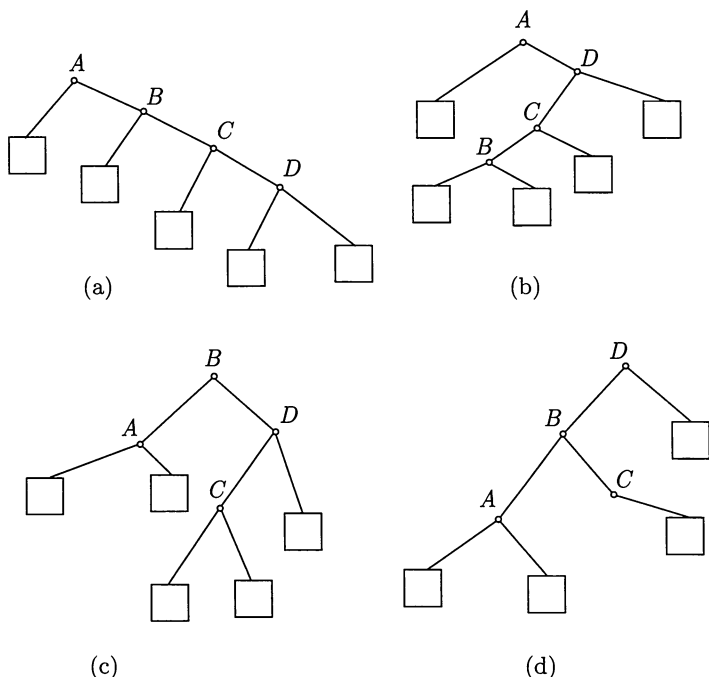


Рис. 13.1. Четыре дерева бинарного поиска над множеством имен $\{A, B, C, D\}$

2. Если z совпадает с именем в корне, поиск завершается успешно.
3. Если z предшествует имени в корне, поиск продолжается ниже в левом поддереве корня.
4. Если z следует за именем в корне, поиск продолжается ниже в правом поддереве корня.

Пусть бинарное дерево имеет вид, в котором каждый узел представляет собой тройку (LEFT, NAME, RIGHT), где LEFT и RIGHT содержат указатели левого и правого сыновей соответственно, и NAME содержит имя, хранящееся в узле. Указатели могут иметь значение Λ , означающее, что поддерево, на которое они указывают пусто. Если указатель корня дерева есть Λ , то само дерево пусто. Как и следовало ожидать, успешный поиск завершается во внутреннем узле дерева бинарного поиска и неуспешный поиск завершается во внешнем узле.

Эта процедура нахождения имени z в таблице, организованная в виде дерева бинарного поиска T , показана в алгоритме 13.5. Отметим его сходство с алгоритмом 13.4 (бинарный поиск).

$p \leftarrow \text{корень } T$

while $p \neq \Lambda$ do case $\left\{ \begin{array}{l} z = \text{NAME}(p) : \{\text{найдено: } p \text{ указывает на } z\} \\ z < \text{NAME}(p) : p \leftarrow \text{LEFT}(p) \\ z > \text{NAME}(p) : p \leftarrow \text{RIGHT}(p) \end{array} \right.$

{не найдено: z нет в таблице}

Алгоритм 13.5. Поиск в дереве бинарного поиска

Логарифмический поиск в динамических таблицах

Здесь мы рассмотрим организацию в виде деревьев для таблиц, в которых часто встречаются включения и исключения. Что происходит с временем поиска в дереве, которое модифицировалось путем включения и исключения? Если включенные и исключенные имена выбраны случайно, то оказывается, что в среднем время поиска мало изменяется; но в худшем случае поведение плохое — деревья могут вырождаться в линейные списки, поиск в которых нужно осуществлять последовательно. Проблема вырождения дерева в линейный список, приводящая к времени поиска $O(n)$ вместо $O(\log n)$ в практических применениях выражена более резко, чем это указывается теоретическим анализом. Такой анализ обычно предполагает, что включения и исключения появляются случайным образом, но на практике часто это не так.

Случайные деревья бинарного поиска. Как ведут себя деревья бинарного поиска без ограничений в качестве динамических деревьев? Другими словами, предположим, что дерево бинарного поиска меняется при случайных включениях и исключениях.

Для включения z мы используем незначительную модификацию алгоритма 13.5. Если z не было найдено, мы получаем для z новую ячейку и связываем ее с последним узлом, пройденным во время безуспешного поиска z . Соответствующее предложение нельзя однако просто добавить в конце алгоритма 13.5, поскольку при нормальном окончании цикла while указатель p не указывает больше на последний пройденный узел, а вместо этого имеет значение Λ . В связи с этим мы должны производить включение до того, как выполняется предположение $p \leftarrow \text{LEFT}(p)$ или $p \leftarrow \text{RIGHT}(p)$; мы можем осуществить это, сделав процедуру вклю-

чения рекурсивной. Процедура $INSERT(z, T)$ выдает в качестве значения указатель на дерево, в которое добавлено z . Таким образом, $T \leftarrow INSERT(z, T)$ используется для включения z в T .

$T \leftarrow INSERT(z, T)$

procedure $INSERT(z, T)$

$p \leftarrow \text{корень } T$

if $p = \Lambda$ *then* $\begin{cases} p \leftarrow \text{get-cell} \\ NAME(p) \leftarrow z \\ LEFT(p) \leftarrow RIGHT(p) \leftarrow \Lambda \end{cases}$

else $\begin{cases} \text{case} \begin{cases} z = NAME(p) : \{\text{уже в таблице}\} \\ z < NAME(p) : LEFT(p) \leftarrow INSERT(z, LEFT(p)) \\ z > NAME(p) : RIGHT(p) \leftarrow INSERT(z, RIGHT(p)) \end{cases} \end{cases}$

$INSERT \leftarrow p$

return

Алгоритм 13.6. Включение в дерево бинарного поиска

Исключение гораздо сложнее включения, и мы изложим здесь только основную идею. Если подлежащее удалению имя z имеет самое большее одного сына, то при исключении z его сын (если он вообще есть) объявляется сыном отца z . Если z имеет двух сыновей, его прямо удалить нельзя. Вместо этого мы находим в таблице либо имя y_1 , которое непосредственно предшествует z , либо имя y_2 , которое непосредственно следует за z в естественном порядке. Оба имени принадлежат узлам, которые имеют не больше одного сына, и, таким образом, z можно исключить заменой его либо именем y_1 , либо y_2 , и затем исключением узла, который содержал y_1 или y_2 , соответственно.

Сбалансированные сильно ветвящиеся деревья

Деревья бинарного поиска естественным образом обобщаются до m -арных деревьев поиска, в которых каждый узел имеет $k \leq m$ сыновей и содержит $k - 1 \leq m - 1$ имен. Имена в узле делят множество имен на k подмножеств, каждое подмножество соответствует одному из k поддеревьев узла. На рис. 13.2 показано полностью заполненное 5-арное дерево с двумя уровнями. Заметим, что мы не можем требовать, чтобы каждый узел m -арного дерева имел ровно m сыновей и включал ровно $m - 1$ имен; если мы захотим включить Z в дерево на рисунке 13.2, то должны будем создать узлы с меньше чем m сыновьями и меньше чем $m - 1$ именами. Таким образом, определение m -арного дерева утверждает только,

что каждый узел имеет не более m сыновей и содержит не более $m - 1$ имен. Ясно, что на m -арных деревьях можно осуществлять поиск так же, как и на бинарных деревьях.

Как и в деревьях бинарного поиска, полезно различать внутренние узлы и листья. Внутренний узел содержит $k \geq 1$ имен, записанных в естественном порядке, и имеет $k + 1$ сыновей, каждый из которых может быть либо внутренним узлом, либо листом. Лист не содержит имен (разве что временно в процессе включения), и, как раньше, в листьях — завершаются безуспешные поиски. Обычно за очевидностью мы на рисунке их опускаем.

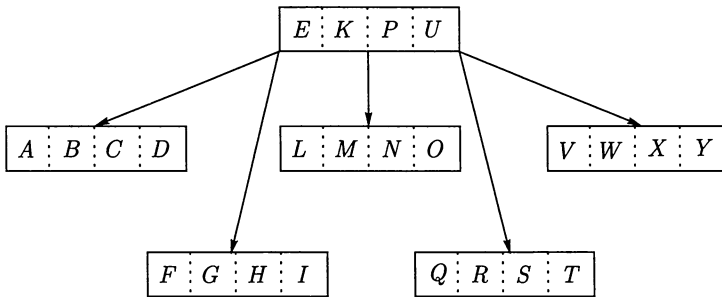


Рис. 13.2. Абсолютно сбалансированное, полностью заполненное 5-арное дерево поиска

Сбалансированное сильно ветвящееся дерево порядка m есть m -арное дерево в котором:

1. Все листья расположены на одном уровне.
2. Корень имеет k сыновей, $2 \leq k \leq m$.
3. Другие внутренние узлы имеют k' сыновей, $m/2 \leq k' \leq m$.

Лекция 14. Сортировка (часть 1)

Введение. Внутренняя сортировка. Вставка. Обменная сортировка. Вопросы и ответы.

Ключевые слова: полная сортировка, частичная сортировка, внутренняя сортировка, внешняя сортировка, вставка, пузырьковая сортировка, быстрая сортировка.

Введение

Рассматриваемые здесь задачи можно отнести к наиболее часто встречающимся классам комбинаторных задач. Почти во всех машинных приложениях множество объектов должно быть переразмещено в соответствии с некоторым заранее определенным порядком. Например, при обработке коммерческих данных часто бывает необходимо расположить их по алфавиту или по возрастанию номеров. В числовых расчетах иногда требуется знать наибольший корень многочлена, и т.д.

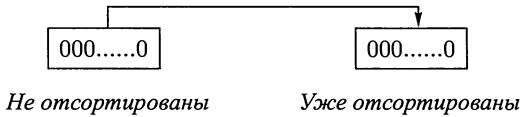
Будем считать заданной таблицу с n именами, обозначаемыми x_1, x_2, \dots, x_n . Каждое имя x_i принимает значение из пространства имен, на котором определен линейный порядок. Будем считать, что никакие два имени не имеют одинаковых значений; то есть любые x_i, x_j обладают тем свойством, что если $i \neq j$, то либо $x_i < x_j$, либо $x_i > x_j$. Ограничение $x_i \neq x_j$ при $i \neq j$ упрощает анализ без потери общности, ибо и при наличии равных имен корректность идей и алгоритмов не нарушается. Наша цель состоит в том, чтобы выяснить что-нибудь относительно перестановки $\Pi = (\pi_1, \pi_2, \dots, \pi_n)$ для которой $x_{\pi_1} < x_{\pi_2} < \dots < x_{\pi_n}$. В задаче **полной сортировки** требуется полностью определить Π , хотя обычно это делается неявно путем переразмещения имени в порядке возрастания. В задачах **частичной сортировки** требуется либо извлечь частичную информацию о Π (например, Π_i для нескольких значений i), либо полностью определить Π по некоторой заданной частичной информации о ней (так обстоит дело при слиянии двух упорядоченных таблиц).

При **внутренней сортировке** решается задача полной сортировки для случая достаточно малой таблицы, умещающейся непосредственно в адресной памяти. **Внешняя сортировка** представляет собой задачу полной сортировки для случая такой большой таблицы, что доступ к ней организован по частям, расположенным на внешних запоминающих устройствах. Частичная сортировка — задачи выбора i — наибольшего имени и слияния двух упорядоченных таблиц.

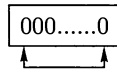
Внутренняя сортировка

Существует по крайней мере пять широких классов алгоритмов внутренней сортировки.

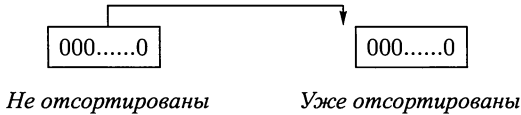
1. Вставка. На i -м этапе i -е имя помещается на надлежащее место между $i - 1$ уже отсортированными именами:



2. Обмен. Два имени, расположение которых не соответствует порядку, меняются местами (обмен). Эти процедуры повторяются до тех пор, пока остаются такие пары.

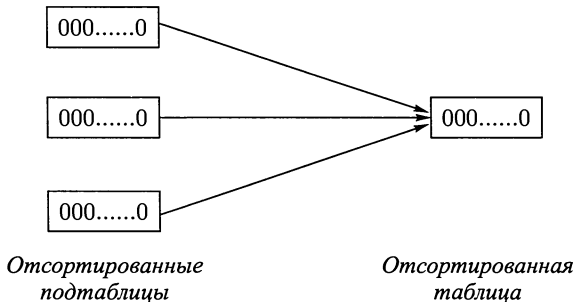


3. Выбор. На i -м этапе из неотсортированных имен выбирается i -е наибольшее (наименьшее) имя и помещается на соответствующее место.



4. Распределение. Имена распределяются по группам, и содержимое групп затем объединяется таким образом, чтобы частично отсортировать таблицу; процесс повторяется до тех пор, пока таблица не будет отсортирована полностью.

5. Слияние. Таблица делится на подтаблицы, которые сортируются по отдельности и затем сливаются в одну.



Эти классы нельзя назвать ни взаимоисключающими, ни исчерпывающими: одни алгоритмы сортировки можно с полным основанием отнести более чем к одному классу (пузырьковую сортировку можно рассматривать и как выбор, и как обмен), а другие не укладываются ни в один из классов. Тем не менее, перечисленные пять классов достаточно удобны для классификации обсуждаемых алгоритмов сортировки.

Сосредоточим внимание на первых четырех классах алгоритмов сортировки. Алгоритмы, основанные на слиянии, приемлемы для внутренней сортировки, но более естественно рассматривать их как методы внешней сортировки.

В описываемых алгоритмах сортировки имена образуют последовательность, которую будем обозначать x_1, x_2, \dots, x_n независимо от возможных пересылок данных; таким образом, значением x_i является любое текущее имя в i -й позиции последовательности. Многие алгоритмы сортировки наиболее применимы к массивам; в этом случае x_i обозначает i -й элемент массива. Другие алгоритмы более приспособлены для работы со связанными списками: здесь x_i обозначает i -й элемент списка. Следующие обозначения используются для пересылок данных:

$x_i \leftrightarrow x_j$ значения x_i и x_j меняются местами.

$x_i \leftarrow y$ значение y присваивается имени x_i .

$y \leftarrow x_j$ значение имени x_j присваивается y .

Таким образом, операция $x_i \leftarrow x_j$, которая встречается в различных алгоритмах сортировки, временно нарушает предположение о том, что никакие два имени не имеют одинаковых значений. Однако это условие всегда обязательно восстанавливается.

В каждом из рассматриваемых алгоритмов будем считать, что имена нужно сортировать на месте. Другими словами, перерасположение имен должно происходить внутри последовательности x_1, x_2, \dots, x_n ; при этом существуют одна или две дополнительные ячейки, в которых временно размещается значение имени. Ограничение «на месте» основано на предположении, будто число имен настолько велико, что во время сортировки не допускается перенос их в другую область памяти. Если в распоряжении имеется память, достаточная для такого переноса, то некоторые из обсуждаемых алгоритмов можно значительно ускорить. Эти рассмотрения заставляют нас в алгоритмах распределяющей сортировки и сортировки слиянием реализовать последовательность x_1, x_2, \dots, x_n как связанный список.

Вставка

Вставка — простейшая сортировка вставками проходит через этапы $j = 2, 3, \dots, n$: на этапе j имя x_j вставляется на свое правильное место среди x_1, x_2, \dots, x_{j-1} .

x_0	x_1	x_2	x_3	x_4	x_5	
$-\infty$	8	7	2	4	6	$j=2$
$-\infty$	7	8	2	4	6	$j=3$
$-\infty$	2	7	8	4	6	$j=4$
$-\infty$	2	4	7	8	6	$j=5$
$-\infty$	2	4	6	7	8	

Рис. 14.1. Простая сортировка вставками, используемая на таблице из $n = 5$ имен. Пунктирные вертикальные линии разделяют уже отсортированную часть таблицы и еще не отсортированную

При вставке имя x_j временно размещается в X , и просматриваются имена $x_{j-1}, x_{j-2}, \dots, x_i$; они сравниваются с X и сдвигаются вправо, если обнаруживается, что они больше X . Имеется фиктивное имя x_0 , значение которого $-\infty$ служит для остановки просмотра слева. На рис. 14.1 работа этого алгоритма проиллюстрирована на примере таблицы из пяти имен.

$x_0 \leftarrow -\infty$

```

for j = 2 to n do
    i ← j - 1
    X ← xj
    while X < xi do
        xi+1 ← xi
        i ← i - 1
    xi+1 ← X
    
```

Алгоритм 14.1. Простая сортировка вставками

Эффективность этого алгоритма, как и большинства алгоритмов сортировки, зависит от числа сравнений имен и числа пересылок данных, осуществляемых в трех случаях : худшем, среднем (в предположении, что все $n!$ перестановок равновероятны) и лучшем.

Обменная сортировка

Обменная сортировка некоторым систематическим образом меняет местами пары имен, не отвечающие порядку, до тех пор, пока такие па-

ры существуют. Фактически алгоритм 14.1 можно рассматривать как обменную сортировку, в которой имя x_j меняется местами со своим соседом слева, пока не оказывается на правильном месте. В этом разделе мы обсуждаем два типа обменных сортировок: хорошо известную, но относительно неэффективную пузырьковую сортировку и быструю сортировку — один из лучших со всех точек зрения алгоритмов внутренней сортировки.

Пузырьковая сортировка. Наиболее очевидный метод систематического обмена местами имен с неправильным порядком состоит в просмотре пар смежных имен последовательно слева направо и перемене мест тех имен, которые не отвечают порядку.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8		d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
	4	7	3	1	5	8	2	6		0	0	2	3	1	0	5	2
1-й проход	4	3	1	5	7	2	6	8		0	1	2	0	0	4	1	0
2-й проход	3	1	4	5	2	6	7	8		0	1	0	0	3	0	0	0
3-й проход	1	3	4	2	5	6	7	8		0	0	0	2	0	0	0	0
4-й проход	1	3	2	4	5	6	7	8		0	0	1	0	0	0	0	0
5-й проход	1	2	3	4	5	6	7	8		0	0	0	0	0	0	0	0
6-й проход	1	2	3	4	5	6	7	8		0	0	0	0	0	0	0	0

Рис. 14.2. Пузырьковая сортировка, примененная к таблице. Показан вектор инверсии таблицы после каждого прохода

Эта техника получила название пузырьковой сортировки, так как большие имена «пузырьками всплывают» вверх (то есть на правый конец) таблицы. В алгоритме 14.2 эта простая идея реализуется с одним небольшим усовершенствованием: ясно, что не имеет смысла продолжать просмотр для больших имен (в правом конце таблицы), про которые известно, что они находятся на своих окончательных позициях. В алгоритме 14.2 используется переменная b , значение которой в начале цикла *while* равно наибольшему индексу t , такому, что про имя x_t еще не известно, стоит ли оно в окончательной позиции. На рис. 14.2 показана работа алгоритма на примере таблицы с $n = 8$ именами.

Анализ пузырьковой сортировки зависит от трех факторов: числа проходов (то есть числа выполнений тела цикла *while*), числа сравне-

$$b \leftarrow n$$

```

while b ≠ 0 do
  t ← 0
  for j = 1 to b - 1 do if xj > xj+1 then
    xj ↔ xj+1
    t ← j
  b ← t

```

Алгоритм 14.2. Пузырьковая сортировка

ний $x_j > x_{j+1}$ и числа обменов $x_j \leftrightarrow x_{j+1}$. Число обменов равно, как в алгоритме 14.1, числу инверсий: 0 в лучшем случае, $\frac{1}{2}n(n-1)$ в худшем случае и $\frac{1}{4}n(n-1)$ — в среднем. Рисунок 14.2 дает возможность предположить, что каждый проход пузырьковой сортировки, исключая последний, уменьшает на единицу каждый ненулевой элемент вектора инверсий и циклически сдвигает вектор на одну позицию влево; легко доказать, что это верно в общем случае, и поэтому число проходов равно единице плюс наибольший элемент вектора инверсий. В лучшем случае имеется всего один проход, в худшем случае — n проходов и в среднем — $\sum kP_k$ проходов, где P_k — вероятность того, что наибольшим элементом вектора инверсии является $k-1$. Общее число сравнений имен трудно определить, но можно показать, что оно равно $n-1$ в лучшем случае, $\frac{1}{2}n(n-1)$ в худшем случае и $\frac{1}{2}(n^2 - n \ln n) + O(n)$ — в среднем.

Пузырьковую сортировку можно несколько улучшить, но при этом она все еще не сможет конкурировать с более эффективными алгоритмами сортировки. Ее единственным преимуществом является простота.

Как в простой сортировке вставками, так и в пузырьковой сортировке (алгоритм 14.2) основной причиной неэффективности является тот факт, что обмены дают слишком малый эффект, так как в каждый момент времени имена сдвигаются только на одну позицию. Такие алгоритмы непременно требуют порядка n^2 операций, как в среднем, так и в худшем случаях.

Быстрая сортировка. Идея метода быстрой сортировки состоит в том, чтобы выбрать одно из имен в таблице и использовать его для разделения таблицы на две подтаблицы, составленные соответственно из имен меньших и больших выбранного, которые затем рекурсивно сортируются с использованием быстрой сортировки. Разделение можно реализовать, одновременно просматривая таблицу и слева направо, и справа налево, меняя местами имена в неправильных частях таблицы. Имя, используемое для расщепления таблицы, затем помещается между двумя подтаблицами, и две подтаблицы сортируются рекурсивно.

В алгоритме 14.3 показаны детали быстрой сортировки для сортировки таблицы $(x_f, x_{f+1}, \dots, x_l)$, где x_j используется для разбиения та-

блицы на подтаблицы. На рис. 14.3 показано, как алгоритм 14.3 использует два указателя i и j для просмотра таблицы во время разбиения. В начале цикла «while $i < j$ » i и j указывают соответственно на первое и последнее имена, о которых известно, что они находятся не в тех частях файла, в которых требуется. Когда $i < j$ встречаются, то есть когда $i \geq j$, все имена находятся в соответствующих частях таблицы и x_f помещается между двумя частями, меняясь при этом местами с x_j , алгоритм предполагает, что имя x_{i+1} определено и больше, чем x_f, x_{f+1}, \dots, x_l .

procedure QUICKSORT(f, l)

 {отсортировать x_f, x_{f+1}, \dots, x_l }

 if $f \geq l$ then return

 {разделить таблицу}

$i \leftarrow f + 1$

 while $x_i < x_f$ do $i \leftarrow i + 1$

$j \leftarrow l$

 while $x_j < x_f$ do $j \leftarrow j - 1$

 while $i < j$ do {в этот момент имеем $i < j, x_i \geq x_f \geq x_j$

$x_i \leftrightarrow x_j$

$i \leftarrow i + 1$

 while $x_i < x_f$ do $i \leftarrow i + 1$

$j \leftarrow j - 1$

 while $x_j < x_f$ do $j \leftarrow j - 1$

$x_f \leftrightarrow x_j$

 {отсортировать подтаблицы рекурсивно}

 QUICKSORT($f, j - 1$) QUICKSORT($j + 1, l$) return

Алгоритм 14.3. Рекурсивный вариант быстрой сортировки, использующий первое имя для расщепления таблицы. Предполагается, что имя x_{i+1} определено и больше или равно x_f, x_{f+1}, \dots, x_l

Алгоритм 14.3 изящен, но непрактичен. Проблема состоит в том, что рекурсия используется для записи подтаблиц, которые рассматриваются на более поздних этапах, и в худших случаях (когда таблица уже отсортирована) глубина рекурсии может равняться n . Следовательно, для стека, реализующего рекурсию, необходима память, пропорциональная n ; для больших n такое требование становится неприемлемым. Кроме того, второе рекурсивное обращение к быстрой сортировке в алгоритме 14.3 может быть легко исключено. По этим причинам мы предлагаем алгоритм 14.4, итерационный вариант быстрой сортировки, в которой стек

	x_f	x_{f+1}	x_{f+2}										x_{l-1}	x_l
Начало	27	99	0	8	13	64	86	16	7	10	88	25	90	
		\uparrow i											\uparrow j	
Первый обмен	27	99	0	8	13	64	86	16	7	10	88	25	90	
Второй обмен	27	25	0	8	13	64	86	16	7	10	88	99	90	
						\uparrow i				\uparrow j				
Третий обмен	27	25	0	8	13	10	86	16	7	64	88	99	90	
						\uparrow i			\uparrow j					
Просмотры встретились	27	25	0	8	13	10	7	16	86	64	88	99	90	
								\uparrow ij						
x_f помещается на свое место	27	25	0	8	13	10	7	16	86	64	88	99	90	
								\uparrow j						
Разделенная таблица	16 25 0 8 13 10 7						27	86 64 88 99 90						
							\uparrow i							

Рис. 14.3. Фаза разбиения быстрой сортировки, использующей первое имя для разбиения таблицы. Значение x_{i+1} не показано, оно предполагается большим, чем другие показанные значения

ведется явно. Элементом стека является пара (f, l) : когда пара находится в стеке, это значит, что нужно сортировать соответствующие x_f, \dots, x_l . Алгоритм 14.4 помещает в стеке большую из двух подтаблиц и немедленно применяет алгоритм к меньшей подтаблице. Это уменьшает глубину стека в худшем случае примерно до $\lg n$. Заметим, что подтаблицы длины 1 игнорируются и что расщепление подтаблицы делается с использованием случайно выбранного имени в этой подтаблице.

```

 $S \leftarrow$  пустой стек
 $S \leftarrow (0, 0)$ 
 $f \leftarrow 1$ 
 $x_{n+1} \leftarrow \infty$ 
 $l \leftarrow n$ 
  while  $f < l$  do
     $x_f \leftrightarrow x_{rand(f, l)}$ 
     $i \leftarrow f + l$ 
    while  $x_i < x_f$  do  $i \leftarrow i + 1$ 
     $j \leftarrow l$ 
    while  $x_j > x_f$  do  $j \leftarrow j - 1$ 
    while  $i < j$  do
       $x_i \leftrightarrow x_j$ 
       $i \leftarrow i + 1$ 
      while  $x_i < x_f$  do  $i \leftarrow i + 1$ 
       $j \leftarrow j - 1$ 
      while  $x_j > x_f$  do  $j \leftarrow j - 1$ 
     $x_f \leftrightarrow x_j$ 
    Case
       $j - 1 \leq f$  and  $l \leq j + 1$ : {обе таблицы тривиальны}
         $(f, l) \leftarrow S$ 
       $j - 1 \leq f$  and  $l \leq j + 1$ : {нетривиальна только правая таблица}
         $f \leftarrow j + 1$ 
       $j - 1 \leq f$  and  $l \leq j + 1$ : {нетривиальна только левая таблица}
         $l \leftarrow j - 1$ 
       $j - 1 > f$  and  $l > j + 1$ : {никакая подтаблица не является: тривиальной: поместить большую в  $S$ }
        if  $j - f > l - j$  then {левая подтаблица длиннее}
           $S \leftarrow (f, j - 1)$ 
           $f \leftarrow j + 1$ 
        else {правая таблица длиннее}
           $S \leftarrow (j + 1, l)$ 
           $l \leftarrow j - 1$ 

```

Алгоритм 14.4. Итерационный вариант быстрой сортировки

Лекция 15. Сортировка (часть 2)

Выбор. Распределяющая сортировка. Цифровая распределяющая сортировка. Внешняя сортировка. Частичная сортировка. Частичная сортировка (выбор). Частичная сортировка (слияние). Вопросы и ответы.

Ключевые слова: турнир с выбыванием, пирамида, пирамидальная сортировка, лексикографический порядок, внешняя сортировка, исходные отрезки, цепочки, выбор с замещением.

Выбор

В сортировке посредством выбора основная идея состоит в том, чтобы идти по шагам $i = 1, 2, \dots, n$, находя i -е наибольшее (наименьшее) имя и помещая его на его место на i -ом шаге. Простейшая форма сортировки выбором представлена алгоритмом 15.1: i -е наибольшее имя находится очевидным способом просмотром оставшихся $n - i + 1$ имен. Число сравнений имен на i -ом шаге равно $n - i$, что приводит к общему числу сравнений имен $(n - 1) + (n - 2) + \dots + 1 = \frac{1}{2}n(n - 1)$ независимо от входа, поэтому ясно, что это не очень хороший способ сортировки.

$$\text{for } j = n \text{ to } 2 \text{ by } -1 \text{ do } \left\{ \begin{array}{l} i \leftarrow 1 \\ \text{for } k = 2 \text{ to } j \text{ do if } x_i < x_k \text{ then } i \leftarrow k \\ x_i \leftrightarrow x_j \end{array} \right.$$

Алгоритм 15.1. Простая сортировка выбором

Несмотря на неэффективность алгоритма 15.1, идея выбора может привести и к эффективному алгоритму сортировки. Весь вопрос в том, чтобы найти более эффективный метод определения i -го наибольшего имени, чего можно добиться, используя механизм *турнира с выбыванием*. Суть его такова: сравниваются $x_1 : x_2, x_3 : x_4, x_5 : x_6, \dots, x_{n-1} : x_n$, затем сравниваются «победители» (то есть большие имена) этих сравнений и т.д.; эта процедура для $n = 16$ показана на рис. 15.1. Заметим, что для определения наибольшего имени этот процесс требует $n - 1$ сравнений имен; но, определив наибольшее имя, мы обладаем большим объемом информации о втором по величине (в порядке убывания) имени: оно должно быть одним из тех, которые «потерпели поражение» от наибольшего имени. Таким образом, второе по величине имя теперь можно

определить, заменяя наибольшее имя на $-\infty$ и вновь осуществляя сравнение вдоль пути от наибольшего имени к корню. На рис. 15.2 эта процедура показана для дерева из рис. 15.1.

Идея турнира с выбыванием прослеживается при сортировке весьма отчетливо, если имена образуют пирамиду. **Пирамида** — это полностью сбалансированное бинарное дерево высоты h , в котором все листья находятся на расстоянии h или $h - 1$ от корня и все потомки узла меньше его самого; кроме того, в нем все листья уровня максимально смещены влево. На рис. 15.3 показано множество имен, организованных в виде пирамиды. Чтобы получить удобное линейное представление дерева, пирамиду можно хранить по уровням в одномерном массиве: сыновья имени из i -ой позиции есть имена в позициях $2i$ и $2i + 1$. Таким образом, пирамида, представленная на рисунке 15.3, принимает вид

$i:$	1	2	3	4	5	6	7	8	9	10	11	12
x_i	94	93	75	91	85	44	51	18	48	58	10	34

Заметим, что в пирамиде наибольшее имя должно находиться в корне и, таким образом, всегда в первой позиции массива, представляющего пирамиду. Обмен местами первого имени с n -м помещает наибольшее имя в его правильную позицию, но нарушает свойство пирамидальности в первых $n - 1$ именах. Если мы можем сначала построить пирамиду, а затем эффективно восстановить ее, то все в порядке, так как тогда можно производить сортировку следующим образом: построить пирамиду из x_1, x_2, \dots, x_n ,

for $i = n$ to 2 by -1 do $\left\{ \begin{array}{l} x_1 \leftrightarrow x_i \\ \text{восстановить пирамиду} \\ \text{в } x_1, \dots, x_{i-1} \end{array} \right.$

Это общее описание **пирамидальной сортировки**.

Процедура $RESTORE(j, k)$ восстановления пирамиды из последовательности x_j, x_{j+1}, \dots, x_k в предположении, что все поддеревья суть пирамиды, такова:

procedure $RESTOREj, k$
 if $x_j \neq \text{лucm}$ then $\left\{ \begin{array}{l} \text{пусть } x_m \text{ есть больший из сыновей } x_j \\ \text{if } x_m > x_j \text{ then } \left\{ \begin{array}{l} x + m \leftrightarrow x_j \\ RESTORE(m, k) \end{array} \right. \end{array} \right.$
 return

Переписывая это итеративным способом и дополняя деталями, мы получим алгоритм 15.2.

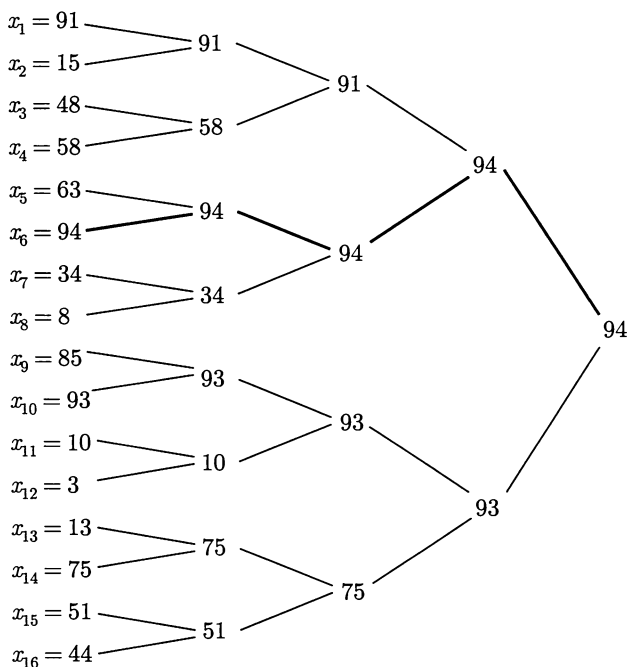


Рис. 15.1. Использование турнира с выбыванием для отыскания наибольшего имени. Путь наибольшего имени показан жирной линией

Распределяющая сортировка

Обсуждаемый здесь алгоритм сортировки отличается от рассматривавшихся до сих пор тем, что он основан не на сравнениях между именами, а на представлении имен. Мы полагаем, что каждое из имен x_1, x_2, \dots, x_n имеет вид

$$x_i = (x_{i,p}, x_{i,p-1}, \dots, x_{i,1})$$

и их нужно отсортировать в возрастающем *лексикографическом порядке*, то есть

$$x_i = (x_{i,p}, x_{i,p-1}, \dots, x_{i,1}) < (x_{j,p}, x_{j,p-1}, \dots, x_{j,1}) = x_j$$

тогда и только тогда, если для некоторого $t \leq p$ имеем $x_{i,l} = x_{j,l}$ для $l > t$ и $x_{i,t} < x_{j,t}$. Для простоты будем считать, что $0 \leq x_{i,l} < r$, и поэтому

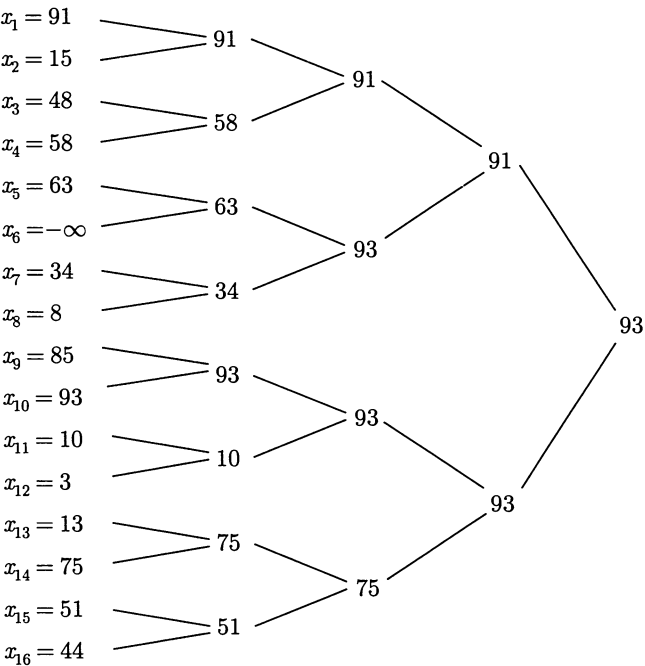


Рис. 15.2. Отыскание второго наибольшего имени путем замены наибольшего имени на $-\infty$. Проведение повторного сравнения имен, побежденных наибольшим именем

```
procedure RESTOREf, l
i ← f
while j ≤ [½l] do
    if 2j < l and x2j < x2j+1 then m ← 2j + 1
    else m ← 2j
    if xm > xj then
        xm ↔ xj
        j ← m
    else j ← m
return
```

Алгоритм 15.2. Восстановление пирамиды из дерева, поддеревья которого суть пирамиды

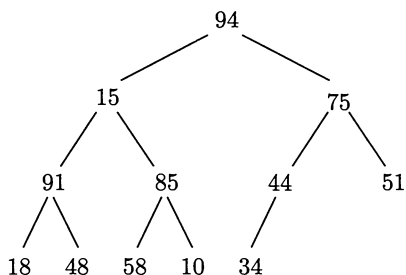


Рис. 15.3. Пирамида, содержащая 12 имен

$\text{for } i = \lceil \frac{1}{2}n \rceil \text{ to } 1 \text{ by } -1 \text{ do } \text{RESTORE}(i, n)$
 $\text{for } i = n \text{ to } 2 \text{ by } -1 \text{ do } \begin{cases} x_1 \leftrightarrow x_i \\ \text{RESTORE}(1, i-1) \end{cases}$

Алгоритм 15.3. Пирамидальная сортировка

имена можно рассматривать как целые, представленные по основанию r , так что каждое имя имеет pr -ичных цифр. Более короткие имена дополняются нулями.

Цифровая распределяющая сортировка

Цифровая распределяющая сортировка основана на наблюдении, что если имена уже отсортированы по младшим разрядам $l, l-1, \dots, 1$, то их можно полностью отсортировать, сортируя только по старшим разрядам $p, p-1, \dots, i+1$ при условии, что сортировка осуществляется таким образом, чтобы не нарушить относительный порядок имен с одинаковыми цифрами в старших разрядах. Заметим, что к самой таблице обращаются по правилу «первым включается – первым исключается», и поэтому лучшим способом представления являются очереди. В частности, предположим, что с каждым ключом x_i ассоциируется поле связи $LINK_i$; тогда эти поля связи можно использовать для сцепления всех имен в таблице вместе во входную очередь Q . При помощи полей связи можно также сцеплять имена в очереди Q_0, Q_1, \dots, Q_{r-1} , используемые для представления стопок. После того как имена распределены по стопкам, очереди, представляющие эти стопки, связываются вместе для получения вновь таблицы Q . Алгоритм 15.4. представляет эту процедуру в общих чертах (очереди описаны в лекции 11). В результате применения алгоритма оче-

редь Q будет содержать имена в порядке возрастания; то есть имена будут связаны в порядке возрастания полями связи, начиная с головы очереди Q .

Использовать поля связи $LINK_1, \dots, LINK_n$ для формирования x_1, \dots, x_n во входную очередь Q

for $j = 1$ to p do $\left\{ \begin{array}{l} \text{Вначале сделать очереди } Q_0, Q_1, \dots, Q_{r-1} \text{ пустыми} \\ \text{while } Q \text{ не пуста do } \left\{ \begin{array}{l} X \leftarrow Q \\ \text{пусть } X = (x_p, x_{p-1}, \dots, x_1) \\ Q_{x_j} \leftarrow X \end{array} \right. \\ \text{сцепить очереди } Q_0, Q_1, \dots, Q_{r-1} \text{ вместе} \\ \text{для формирования новой очереди} \end{array} \right.$

Алгоритм 15.4. Цифровая распределяющая сортировка

Внешняя сортировка

В методах сортировки, обсуждавшихся в предыдущем разделе, мы полагали, что таблица умещается в быстродействующей внутренней памяти. Хотя для большинства реальных задач обработки данных это предположение слишком сильно, оно, как правило, выполняется для комбинаторных алгоритмов. Сортировка обычно используется только для некоторого сокращения времени работы алгоритмов, в которых сортировка применяется только для некоторого сокращения времени работы алгоритмов, когда оно недопустимо велико даже для задач «умеренных» размеров. Например, часто бывает необходимо сортировать отдельные предметы во времени исчерпывающего поиска (лекция 13), но поскольку такой поиск обычно требует экспоненциального времени, маловероятно, что подлежащая сортировке таблица будет настолько большой, чтобы потребовалось использование запоминающих устройств. Однако задача сортировки таблицы, которая слишком велика для основной памяти, служит хорошей иллюстрацией работы с данными большого объема, и поэтому в этом разделе мы обсудим важные идеи *внешней сортировки*. Более того, будем рассматривать только сортировку таблицы путем использования вспомогательной памяти с последовательным доступом. Условимся называть эту память лентой.

Общей стратегией в такой внешней сортировке является использование внутренней памяти для сортировки имен из ленты по частям так, чтобы производить *исходные отрезки* (известные также как *цепочки*) имен в возрастающем порядке. По мере порождения эти отрезки распределяются по t рабочим лентам, и затем производится их слияние по t отрезков

обратно на исходную $(t+1)$ -ю ленту так, что она будет содержать меньшее число более длинных отрезков. Затем отрезки снова распределяются по остальным t лентам, и снова производится их слияние по t штук обратно на $(t+1)$ -ю ленту. Процесс продолжается до тех пор, пока не получится один отрезок, то есть пока таблица не будет полностью отсортирована. Имеются, таким образом, две отдельные проблемы: как порождать исходные отрезки и как осуществлять слияние.

Самый очевидный метод для получения исходных отрезков состоит в том, что можно просто считывать $(t+1)$ -ю ленту m имен, рассортировывать их во внутренней памяти и записывать их на ленту в виде отрезка, продолжая процесс до тех пор, пока не будут исчерпаны все имена. Все полученные таким образом исходные отрезки содержат m имен (исключая, возможно, последний отрезок). Поскольку число исходных отрезков в конце концов определяет время слияния, мы хотели бы найти некоторый метод образования более длинных исходных отрезков и, следовательно, меньшего их количества. Это можно сделать, используя для сортировки идею турнира (пирамидальную сортировку). При этом подходе m имен, которые умещаются в памяти, хранятся в виде такой пирамиды, что сыновья узла больше узла (вместо того, чтобы быть меньше его). Этот метод отвечает определению «победителя» при сравнении имен в сортировках типа турнира как меньшего из двух имен, и это позволяет нам следить за наименьшим именем.

Порождение исходных отрезков продолжается следующим образом. Из входной ленты считываются первые m имен, и затем из них формируется пирамида, как описано выше. Наименьшее имя выводится как первое в первом отрезке и заменяется в пирамиде следующим именем из входной ленты в соответствии с алгоритмом 15.2. модифицированным так, чтобы для восстановления пирамиды следить за наименьшим, а не за наибольшим именем. Процесс, известный как *выбор с замещением*, продолжается таким образом, что к текущему отрезку всегда добавляется наименьшее в пирамиде имя, большее или равное имени, которое последним добавлено к отрезку; при этом добавленное имя заменяется на следующее из входной ленты и восстанавливается пирамида. Когда в пирамиде нет имен, больших, чем последнее имя в текущем отрезке, отрезок обрывается и начинается новый. Этот процесс продолжается до тех пор, пока все имена не сформируются в отрезки.

Разумный путь реализации этой процедуры состоит в том, чтобы рассматривать каждое имя x как пару (r, x) , где r есть номер отрезка, в котором находится x . Иначе говоря, считается, что пирамида состоит из пар $(r_1, x_1), (r_2, x_2), \dots, (r_m, x_m)$; сравнения между парами осуществляются лексикографически. Когда считывается имя, меньшее последнего имени в текущем отрезке, оно должно быть в следующем отрезке, и бла-

годаря наличию номера отрезка это имя будет ниже всех имен пирамиды, которые входят в текущий отрезок.

Порождает ли этот выбор с замещением длинные отрезки? Ясно, что он делает это, по крайней мере, не хуже, чем очевидный метод, так как все отрезки (кроме, возможно, последнего) содержат не меньше m имен. На самом деле можно показать, что средняя длина отрезков, порождаемых выбором с замещением, равна $2m$, и это вполне можно считать улучшением по сравнению с очевидным методом, в котором средняя длина отрезков равна m . Конечно, в лучшем случае все заканчивается только одним исходным отрезком, то есть отсортированной таблицей.

После того, как порождены исходные отрезки, возникает задача повторного распределения их по рабочим лентам и слияния их вместе до тех пор, пока в конце концов мы не получим окончательную отсортированную таблицу. Простейший метод состоит в том, чтобы распределить отрезки по лентам $1, 2, \dots, t$ равномерно и произвести их слияние на ленту $t + 1$. Получаемые в результате отрезки снова равномерно распределить по лентам $1, 2, \dots, t$ и снова произвести слияние, образуя более длинные отрезки на ленте $t + 1$. Процесс продолжается до тех пор, пока на ленте $t + 1$ не останется только один отрезок (отсортированная таблица).

Частичная сортировка

Ранее мы изучали проблему полного упорядочения множества имен, не имея *a priori* информации об абстрактном порядке имен. Имеется два очевидных уточнения этой проблемы. Вместо полного упорядочения требуется только определить k -е наибольшее имя (то есть k -е имя в порядке убывания) (выбор) или, вместо того чтобы начинать процесс, не располагая информацией о порядке, начинать с двух отсортированных подтаблиц (слияние). Мы рассматриваем обе проблемы частичной сортировки.

Частичная сортировка (выбор)

Как при данных именах x_1, x_2, \dots, x_n можно найти k -е из наибольших в порядке убывания? Задача, очевидно, симметрична: отыскание $(n - k + 1)$ -го наибольшего (k -го наименьшего) имени можно осуществить, используя алгоритм отыскания k -го наибольшего, но меняя местами действия, предпринимаемые при результатах $<$ и $>$ сравнения имен. Таким образом, отыскание наибольшего имени ($k = 1$) эквивалентно отысканию наименьшего имени ($k = n$); отыскание второго наибольшего имени ($k = 2$) эквивалентно отысканию второго наименьшего ($k = n - 1$) и т.д.

Конечно, все перечисленные варианты задачи выбора можно решить, используя любой из методов полной сортировки имен и затем тривиально обращаясь к k -му наибольшему. Такой подход потребует порядка $n \log n$ сравнений имен независимо от значений k .

При использовании алгоритма сортировки для выбора наиболее подходящим будет один из алгоритмов, основанных на выборе: либо простая сортировка выбором (алгоритм 15.1) либо пирамидальная сортировка (алгоритм 15.3). В каждом случае мы можем остановиться после того, как выполнены первые k шагов. Для простой сортировки выбором это означает использование

$$(n-1) + (n-2) + \dots + (n-k) = kn - \frac{k(k+1)}{2}$$

сравнений имен, а для пирамидальной — использование $n + k \lg n$ сравнений имен. В обоих случаях мы получаем больше информации, чем нужно, потому что мы полностью определяем порядок k наибольших имен.

Частичная сортировка (слияние)

Вторым направлением исследования частичной сортировки является задача слияния двух отсортированных таблиц $x_1 \leq x_2 \leq \dots \leq x_n$ и $y_1 \leq y_2 \leq \dots \leq y_m$ в одну отсортированную таблицу $z_1 \leq z_2 \leq \dots \leq z_{n+m}$. Существует очевидный способ это сделать: таблицы, подлежащие слиянию, просматривать параллельно, выбирая на каждом шаге меньшее из двух имен и помещая его в окончательную таблицу. Этот процесс немного упрощается добавлением имен-сторожей $x_{n+1} = y_{m+1} = \infty$, как в алгоритме 15.5. В этом алгоритме i и j указывают, соответственно, на последние имена в двух входных таблицах, которые еще не были помещены в окончательную таблицу.

$$x_{n+1} \leftarrow y_{m+1} \leftarrow \infty$$

$$i \leftarrow j \leftarrow 1$$

$$\text{for } k = 1 \text{ to } n + m \text{ do } \left\{ \begin{array}{l} \text{if } x_i < y_j \text{ then } \left\{ \begin{array}{l} z_k \leftarrow x_i \\ i \leftarrow i + 1 \end{array} \right. \\ \text{else } \left\{ \begin{array}{l} z_k \leftarrow y_j \\ j \leftarrow j + 1 \end{array} \right. \end{array} \right.$$

Алгоритм 15.5. Прямое слияние

Лекция 16. Алгоритмы на графах

Поиск в глубину. Алгоритм Дейкстры нахождения кратчайшего пути. Алгоритм Флойда нахождения кратчайших путей между парами вершин. Программы. Вопросы и ответы.

Ключевые слова: поиск в глубину, источник, особый путь.

Поиск в глубину

При решении многих задач, касающихся ориентированных графов, необходим эффективный метод систематического обхода вершин и дуг орграфов. Таким методом является метод **поиск в глубину**. Метод поиска в глубину является основой многих эффективных алгоритмов работы с графами. Предположим, что есть ориентированный граф G , в котором первоначально все вершины помечены меткой «*unvisited*». Поиск в глубину начинается с выбора начальной вершины v орграфа G , для этой вершины метка «*unvisited*» меняется на метку «*visited*». Затем для каждой вершины, смежной с вершиной v и не посещаемой раньше, рекурсивно применяется поиск в глубину. Когда все вершины, которых можно достичь из вершины v , будут рассмотрены, поиск заканчивается. Если некоторые вершины остались не посещенными, то выбирается одна из них и алгоритм повторяется. Этот процесс продолжается до тех пор, пока не будут обойдены все вершины орграфа G .

Метод получил свое название — поиск в глубину, поскольку поиск не посещенных вершин идет в направлении вглубь до тех пор, пока это возможно. Например, пусть x является последней посещенной нами вершиной. Выбираем очередную дугу (x, y) (ребро), выходящую из вершины x . Возможна следующая альтернатива: вершина y помечена меткой «*unvisited*»; вершина y помечена меткой «*visited*». Если вершина y уже посещалась, то отыскивается другая вершина, смежная с вершиной x ; иначе вершина y метится меткой «*visited*» и поиск начинается заново от вершины y . Пройдя все пути, которые начинаются в вершине y , возвращаемся в вершину x , то есть в ту вершину, из которой впервые была достигнута вершина y . Затем процесс повторяется, то есть продолжается выбор нерассмотренных дуг, исходящих из вершины x , и так до тех пор, пока не будут исчерпаны все эти дуги.

Алгоритм Дейкстры нахождения кратчайшего пути

Рассмотрим алгоритм нахождения путей в ориентированном графе. Пусть есть ориентированный граф $G(V, E)$, у которого все дуги имеют неотрицательные метки (веса дуг), а одна вершина определена как **источник**. Задача состоит в нахождении весов кратчайших путей от источника ко всем другим вершинам графа $G(V, E)$. Здесь длина пути определяется как сумма весов дуг, составляющих путь. Эта задача часто называется задачей нахождения кратчайшего пути с одним источником. Отметим, что мы будем говорить о длине пути даже тогда, когда она измеряется в других, не линейных, единицах измерения, например, во временных единицах или в денежном эквиваленте.

Можно представить орграф $G(V, E)$ в виде карты маршрутов рейсовых полетов из одного города в другой. Каждая вершина соответствует городу, а ребро (дуга) (v, w) — рейсовому маршруту из города v в город w . Вес дуги (v, w) — это время полета из города v в город w . В этом случае решение задачи нахождения кратчайшего пути с одним источником для ориентированного графа трактуется как минимальное время перелета между различными городами.

Для решения поставленной задачи будем использовать «жадный» алгоритм, который называют алгоритмом Дейкстры (Dijkstra). Алгоритм строит множество S вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, расстояние до которой от источника меньше, чем для других оставшихся вершин. Если веса всех дуг неотрицательны, то можно быть уверенным, что кратчайший путь от источника к конкретной вершине проходит только через вершины множества S . Назовем такой **путь особым**. На каждом шаге алгоритма используется также массив M , в который записываются длины кратчайших особых путей для каждой вершины. Когда множество S будет содержать все вершины орграфа, то есть для всех вершин будут найдены особые пути, тогда массив M будет содержать длины кратчайших путей от источника к каждой вершине.

Алгоритм Флойда нахождения кратчайших путей между парами вершин

Предположим, что мы имеем помеченный орграф, который содержит время полета по маршрутам, связывающим определенные города, и мы хотим построить таблицу, где приводилось бы минимальное время перелета из одного (произвольного) города в любой другой. В этом случае мы сталкиваемся с общей задачей нахождения кратчайших путей, то есть нахождения кратчайших путей между всеми парами вершин орграфа.

Формулировка задачи. Есть ориентированный граф $G(V, E)$, каждой дуге (ребру) (v, w) этого графа сопоставлен неотрицательный вес C_{vw} . *Общая задача нахождения кратчайших путей заключается в нахождении для каждой упорядоченной пары вершин v, w любого пути от вершины v в вершину w , длина которого минимальна среди всех возможных путей от v к w .*

Можно решать эту задачу, последовательно применяя алгоритм Дейкстры для каждой вершины, объявляемой в качестве источника. Но мы для решения поставленной задачи воспользуемся алгоритмом, предложенным Флойдом (R.W. Floyd). Пусть все вершины орграфа последовательно пронумерованы от 1 до n . Алгоритм Флойда использует матрицу $A(n \times n)$, в которой находятся длины кратчайших путей:

$A_{ij} = C_{ij}$, если $i \neq j$;

$A_{ij} = 0$, если $i = j$;

$A_{ij} = \infty$ если отсутствует путь из вершины i в вершину j .

Над матрицей A выполняется n итераций. После k -й итерации A_{ij} содержит значение наименьшей длины пути из вершины i в вершину j , причем путь не проходит через вершины с номерами большими k .

Вычисление на k -ой итерации выполняется по формуле: $A_{ij}^k = \min(A_{ij}^{k-1}, A_{ik}^{k-1} + A_{kj}^{k-1})$. Верхний индекс k обозначает значение матрицы A после k -ой итерации.

Для вычисления A_{ij}^k проводится сравнение величины A_{ij}^{k-1} (то есть стоимость пути от вершины i к вершине j без участия вершины k или другой вершины с более высоким номером) с величиной $A_{ik}^{k-1} + A_{kj}^{k-1}$ (стоимость пути от вершины i к вершине k плюс стоимость пути от вершины k до вершины j). Если путь через вершину k дешевле, чем A_{ij}^{k-1} , то величина A_{ij}^k изменяется. Рассмотрим орграф:

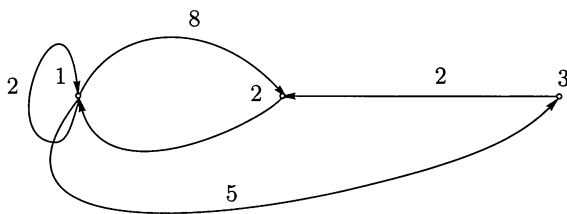


Рис. 16.1. Помеченный орграф

$$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}$$

Рис. 16.2. Матрица $A(3 \times 3)$ на нулевой итерации ($k = 0$)

$$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{bmatrix}$$

Рис. 16.3. Матрица $A(3 \times 3)$ после первой итерации ($k = 1$)

$$\begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

Рис. 16.4. Матрица $A(3 \times 3)$ после второй итерации ($k = 2$)

$$\begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}$$

Рис. 16.5. Матрица $A(3 \times 3)$ после третьей итерации ($k = 3$)

Программы

Программа 1. Построение матрицы инцидентности.

//Построение матрицы инцидентности

//Программа реализована на языке программирования Turbo-C++

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <iostream.h>
```

```
struct elem
```

```
{
    int num; /* Номер вершины */
    int suns; /* Количество сыновей */
    char str[20]; /* Строка с номерами сыновей */
    elem *next; /* Указатель на следующую вершину */
}
```

```
} *head, *w1, *w2;

int Connected(int i, int j)
{
    int k;
    char *str1;
    w2 = head;
    if(i == j) return 0;
    for(k=1; k<i; k++)
        w2 = w2->next;
    if( strchr(w2->str, j) ) return 1;
    return 0;
}

void main()
{

    int tops;
    int i,j,k,l;
    char *str1;

    clrscr();
    printf("Введите количество вершин \n");
    scanf("%d", &tops);

    head = (elem *)malloc(sizeof(elem));
    head->num = 1;
    head->suns = 0;
    head->str[0] = '\0';
    head->next = NULL;

    w1 = head;

    for(i=2;i<=tops;i++)
    {
        w2 = (elem *)malloc(sizeof(elem));
        w2->num = i;
        w2->suns = 0;
        w2->str[0] = '\0';
        w2->next = NULL;
```

```

    w1->next = w2;
    w1 = w2;
}

w1 = head;

for(i=1; i<=tops; i++)
{
//    clrscr();
    printf("Введите количество путей из вершины %d\n", i);
    scanf("%d", &k);

    for(j=1; j<=k; j++)
    {
        printf("Введите связь %d\n", j);
        scanf("%d", &l);
        if((l<=0) || (l > tops))
        {
            printf("Такой вершины нет, повторите попытку\n");
            l = 0;
            j--;
            continue;
        }
        w1->str[w1->suns++] = l;
        w1->str[w1->suns] = '\0';
        if(w1->suns == 49)
        {
            printf("Слишком много связей !");
            exit(1);
        }
    }
    w1 = w1->next;
}
clrscr();
printf("\n\n Матрица инцидентности :\n");
for(i=1; i<=tops; i++)
{
    printf("\n %d ", i);
    for(j=1; j<=tops; j++)
    {
        printf("%d ", Connected(i, j));
    }
}

```

```

    }
}
printf("\n\n Нажмите любую клавишу\ldots ");
getch();
}

```

Программа 2. Поиск вершин, недостижимых из заданной вершины графа.

//Поиск вершин, недостижимых из заданной вершины графа.

//Программа реализована на языке программирования Turbo-C++

```

#include <iostream.h>
#include <fstream.h>
//-----
int n,s;
int c[20][20];
int r[20];
//-----
int load();
int save();
int solve();
//-----
int main(){
    load();
    solve();
    save();
    return 0;
}
//-----
int load(){
    int i,j;
    ifstream in("input.txt");
    in>>n>>s;
    s--;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            in>>c[i][j];
    in.close();
    return 0;
}

int save(){
    int i;

```

```

ofstream out("output.txt");
for (i=0; i<n; i++)
  if (r[i]==0)
    out<<i+1<<" ";
out.close();
return 0;
}
//-----
int solve(){
  int i,h,t;
  int q[400];
  for (i=0; i<n+1; i++) q[i]=0;
  r[s]=1;
  h=0;
  t=1;
  q[0]=s;
  while (h<t){
    for (i=0;i<n;i++){
      if ((c[q[h]][i]>0)&(r[i]==0)){
        q[t]=i;
        t++;
        r[i]=1;
      }
      h++;
    }
    return 0;
  }
}

```

Программа 3. Поиск циклов в графе.

```

{>
Реализация на Turbo-Pascal.
Поиск циклов в графе
<}

```

```

{$R-,I-,S-,Q-}

```

```

const MAXN    = 40;
      QUERYSIZE = 600;

```

```

type vert = record x: integer; s: array [1..MAXN] of integer; end;

```

```
var c : array [1..MAXN,1..MAXN] of integer;  
    n : integer;
```

```
    wr : vert;
```

```
    res : array [1..MAXN] of string;  
    resv: integer;  
    ss : string;
```

```
procedure load;  
var i,j: integer;  
begin  
    assign(input, 'input.txt');  
    reset(input);  
    read(n);  
    for i:=1 to n do  
        for j:=1 to n do  
            read(c[i][j]);  
        close(input);  
    end;
```

```
function saveway(i:integer):string;  
var e:string;  
begin  
    str(i,e);  
    if (wr.s[i]=-1) then  
        saveway:=e+' '  
    else  
        saveway:=saveway(wr.s[i])+e+' '  
    end;
```

```
function findss(s: string): boolean;  
var i      : integer;  
    l1,l2,rs : string;  
    i1,i2,i22 : integer;
```

```
begin  
    findss:=false;  
    l2:=copy(s,1,pos(' ',s)-1);
```

```

i2:=length(l2);
i22:=length(s);
for i:=1 to resv do begin
  l1:=copy(res[i],1,pos(' ',res[i])-1);
  i1:=length(l1);
  rs:=copy(res[i],1,length(res[i])-i1)+res[i];
  if (length(res[i])+i2=i22+i1)and(pos(s,rs)>0)
  then begin
    findss:=true;
    exit;
  end;
end;
end;

```

```

procedure solve;
var h,t,i,j: integer;
    q   : array [1..QUERYSIZE] of vert;
    e   : string;
begin
  resv:=0;
  fillchar(res,sizeof(res),0);

  for i:=1 to n do begin
    fillchar(q[i],sizeof(q[i]),0);
    q[i].x:=i;
    q[i].s[i]:=-1;
  end;

  t:=n+1;
  h:=1;
  while h<t do begin
    for i:=1 to n do
      if (c[q[h].x,i]>0) then begin
        if (q[h].s[i]=-1) then begin
          wr:=q[h];
          str(i,e);
          ss:=saveway(q[h].x)+e;
          if (not findss(ss)) then begin
            inc(resv);
            res[resv]:=ss;
          end;
        end;
      end;
    end;
  end;

```

```
    end;  
    if (q[h].s[i]=0) then begin  
        q[t]:=q[h];  
        q[t].x:=i;  
        q[t].s[i]:=q[h].x;  
        inc(t);  
    end;  
end;  
inc(h);  
end;
```

```
    close(output);  
end;
```

```
procedure save;  
var i: integer;  
begin  
    assign(output,'output.txt');  
    rewrite(output);  
    for i:=1 to resv do  
        writeln(res[i]);  
    close(output);  
end;
```

```
begin  
    load;  
    solve;  
    save;  
end.
```


Лекция 17. Калейдоскоп из комбинаторных алгоритмов

Автоматическое построение лабиринтов. Бинарное дерево. Задача о восьми ферзях. Сортировки. Задача о назначениях (задачи выбора). Ханойская башня. Вопросы и ответы.

Ключевые слова: сортировка, ключ сортировки, сортирующая последовательность, способ сортировки, носитель данных.

Автоматическое построение лабиринтов

Тезей должен был найти выход из Критского лабиринта или погибнуть, убитый Минотавром. Но что поразительно: найти вход в лабиринт — задача не менее трудная.

Здесь не представляется возможным описать все мыслимые лабиринты, да это и не требуется. Мы займемся простыми лабиринтами, построенными на прямоугольнике $m \times n$, где m, n — положительные целые числа. Внутри и на границах прямоугольника поставлены стенки по ребрам покрывающей его единичной квадратной сетки. Чтобы построить из прямоугольника лабиринт, выйдем одну единичную стенку на одной из сторон прямоугольника (получится вход в лабиринт); выйдем одну единичную стенку на противоположной стороне (получится выход) и еще удалим какое-то число строго внутренних стенок. Говорят, что лабиринт имеет решение, если между входом и выходом внутри лабиринта есть путь в виде ломаной, не имеющей общих точек со стенками. Решение единственно, если любые два таких пути проходят через одни и те же внутренние ячейки сетки. На рис. 17.1 приведен пример лабиринта 5×5 .

Один из возможных подходов к решению таков. Выбираем вход; затем, начав от него, добавляем по одной ячейке к главному пути-решению, пока он не достигнет выходной стороны. После этого удаляем некоторые внутренние стенки так, чтобы все клетки оказались соединенными с главным путем. Чтобы главный путь не получился прямым коридором, следует при его построении предусмотреть случайные повороты. Программа должна также следить за тем, чтобы при построении главного пути или при открытии боковых ячеек не нарушалась единственность решения. Наблюдательный читатель заметит, что определение единственности решения не годится в случае, когда путь заходит в боковой тупик и затем возвращается.

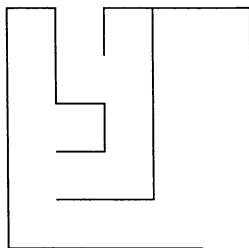


Рис. 17.1. Пример лабиринта

Программу можно написать почти на любом из процедурных языков.

Программа 1. Лабиринт.

{ Программно задаются вход и выход. Нажимая на клавишу <<Enter>>, перебираем всевозможные пути от входа до выхода в лабиринте. Выход из программы по клавише <<Esc>>.

Алгоритм реализован на языке программирования Turbo-Pascal}

```
program Maze;
```

```
uses
```

```
  Graph, Crt;
```

```
var
```

```
  m,n: Integer;
```

```
  Matrix: array [1..100,1..100] of Boolean;
```

```
  Start,Finish: Integer;
```

```
procedure PrepareGraph;
```

```
var
```

```
  Driver,Mode: Integer;
```

```
begin
```

```
  Driver:=VGA;
```

```
  Mode:=VGAHi;
```

```
  InitGraph(Driver,Mode,'c:\borland\tp\bgi');
```

```
end;
```

```
procedure DisplayMaze(x1,y1,x2,y2: Integer);
```

```
var
```

```
  i,j: Integer;
```

```
  dx,dy: Real;
```

```

begin
  SetFillStyle(1,8);
  SetColor(15);
  dx:=(x2-x1)/m;
  dy:=(y2-y1)/n;
  for i:=1 to n do
    for j:=1 to m do
      if not Matrix[i,j] then
        Rectangle(Round(x1+(i-1)*dx),Round(y1+(j-1)*dy),
          Round(x1+i*dx),Round(y1+j*dy));
    end;
  end;

```

```

function CreatesPath(i,j: Integer): Boolean;
var
  Result: Boolean;
  Count: Integer;
  ii,jj: Integer;
begin
  Count:=0;
  if (i>1) and Matrix[i-1,j] then Inc(Count);
  if (i<m) and Matrix[i+1,j] then Inc(Count);
  if (j>1) and Matrix[i,j-1] then Inc(Count);
  if (j<m) and Matrix[i,j+1] then Inc(Count);
  if Count>1 then Result:=true else Result:=false;
  CreatesPath:=Result;
end;

```

```

function DeadEnd(i,j: Integer): Boolean;
var
  Result: Boolean;
  Count: Integer;
begin
  Count:=0;
  if (i=2) or CreatesPath(i-1,j) then Inc(Count);
  if (i=m-1) or CreatesPath(i+1,j) then Inc(Count);
  if (j=2) or CreatesPath(i,j-1) then Inc(Count);
  if (j=n-1) or CreatesPath(i,j+1) then Inc(Count);
  if Count=4 then Result:=true else Result:=false;
  DeadEnd:=Result;
end;

```

```
function CreateMaze: Boolean;
var
  i,j: Integer;
  di,dj: Integer;
  Result: Boolean;
begin
  Randomize;
  for i:=1 to n do
    for j:=1 to m do Matrix[i,j]:=false;
  Start:=Random(m-2)+2;
  i:=Start;
  j:=2;
  Matrix[Start, 1]:=true;
  repeat
    Matrix[i,j]:=true;
    di:=0;
    dj:=0;
    while (di=0) and (dj=0) do begin
      di:=1-Random(3);
      if (i+di=1) or (i+di=m) then di:=0;
      if di=0 then dj:=1-Random(3);
      if j+dj=1 then dj:=0;
      if CreatesPath(i+di,j+dj) then begin
        di:=0;
        dj:=0;
      end;
    end;
    i:=i+di;
    j:=j+dj;
  until DeadEnd(i,j) or (j=n);
  Finish:=i;
  Matrix[Finish,n]:=true;
  if j<n then Result:=false else Result:=true;
  CreateMaze:=Result;
end;

begin
  m:=6;
  n:=6;

  PrepareGraph;
```

```

repeat
  ClearDevice;
  repeat until CreateMaze;
  DisplayMaze(120,40,520,440);
  repeat until KeyPressed;
until ReadKey=#27;
CloseGraph;
end.

```

Программа 2. Лабиринт.

{Лабиринт реализуется автоматически, без участия пользователя.
Алгоритм реализован на языке программирования Turbo-Pascal }

```

uses graph,crt;
var
  mpos,npos,m,n,delx,x,y,t,gd,gm,i,k:integer;

begin
  randomize;
  writeln('Input labyrinth size (x and y)');
  readln(m,n);
  writeln('Input entrance&exit coordinates (mpos<m and npos<m)');
  readln(mpos,npos);
  initgraph(gd,gm,'c:\borland\tp\bgi');
  for i:=1 to m do
    begin
      for k:=1 to n do
        begin
          rectangle(90+10*i,90+10*k,90+10*i+10,90+10*k+10);
        end;
      end;
    setfillstyle(1,0);
    setcolor(0);
    line(100+(mpos-1)*10+1,100,100+(mpos-1)*10+9,100);
    line(100+(npos-1)*10+1,100+n*10,100+(npos-1)*10+9,100+n*10);
    y:=n;
    x:=npos;
    readln;

    while y>1 do
      begin
        delx:=random(m)-x+1;

```

```

if y=2 then delx:=mpos-x;
i:=91+x*10;
if i<90+(x+delx)*10 then
begin
  while i<>90+(x+delx)*10 do
  begin
    i:=i+1;
    line(i,91+y*10,i,99+y*10);
  end;
end;

if i>91+(x+delx)*10 then
begin
  while i<>91+(x+delx)*10 do
  begin
    i:=i-1;
    line(i,91+y*10,i,99+y*10);
  end;
end;

x:=x+delx;
line(91+10*x,90+y*10,99+10*x,90+y*10);

y:=y-1;

end;

readln;
end.

```

Бинарное дерево

Строится бинарное дерево. В узлы дерева засылаются целые положительные числа, выбранные случайным образом. После задания значений вершин на каждом уровне эти значения выводятся на экран, начиная с корневой. Пользователю предлагается сделать выбор номера уровня, в нем определяется максимальное значение и выводится на экран.

Программа 3. Поиск максимального элемента.

```

{ Алгоритм реализован на языке программирования Turbo-Pascal}
uses crt;
type sp=^tree;

```

```
tree=record
  val:integer;
  l:sp;
  r:sp;
end;

var
  t:sp;
  nh, max,h,i:integer;
procedure find(t:sp; h,nh:integer);
begin
  if t=nil then exit;
  if h=nh then
    begin
      if t^.val> max then max:=t^.val;
    end
  else
    begin
      find(t^.l,h+1,nh);
      find(t^.r,h+1,nh);
    end;
  end;
end;
procedure zadtree(var t:sp; h,nh:integer);
begin
  if h=5 then
    begin
      new(t);
      t^.l:=nil;
      t^.r:=nil;
      t^.val:=random(100);
    end
  else
    begin
      new(t);
      zadtree(t^.l, h+1,nh);
      zadtree(t^.r, h+1,nh);
      t^.val:=random(100);
    end;
  end;
end;
procedure writetree(t:sp; h,nh:integer);
begin
  if t=nil then exit;
```

```
if h=nh then
  begin
    write(t^.val,' ');
  end
else
  begin
    writetree(t^.l,h+1,nh);
    writetree(t^.r,h+1,nh);
  end;
end;
begin
  clrscr;
  randomize;
  t:=nil;
  zadtrees(t,1,nh);
  for i:=1 to 5 do
    begin
      writetree(t,1,i);
      writeln;
    end;
  max:=0;
  write('vvedite uroven ');
  readln(nh);
  find(t,1,nh);
  write('max= ',max);
  readln;
end.
```

Задача о восьми ферзях

Условие задачи. Найти все такие расстановки восьми ферзей на шахматной доске, при которых ферзи не бьют друг друга.

Анализ задачи. Пусть A — множество искомым расстановок (конфигураций). Рассмотрим следующий подход к решению задачи. Будем искать множество конфигураций B со следующими свойствами:

1. $A \subset B$.
2. Имеется условие, позволяющее по элементу из B определить, принадлежит ли он A .
3. Имеется процедура, генерирующая все элементы из B .

С помощью процедуры из пункта 3 будем генерировать по очереди все элементы из B ; для элементов из B проверяем (см. пункт 2) принад-

лежит ли он A : в результате в силу 1 свойства будут порождены все элементы A .

Заметим теперь, что ферзи, которые не бьют друг друга, должны располагаться на разных горизонталях. Поэтому можно упорядочить ферзи и всегда ставить k -го ферзя на k -ю горизонталь. Тогда в качестве B можно взять множество конфигураций, в которых на каждой из первых N горизонталей стоит ровно по одному ферзю, причем никакие два ферзя не бьют друг друга.

Программа 4. Расстановка восьми ферзей на шахматной доске.

{ Программа выдает все комбинации ферзей,
которые не бьют друг друга.
Алгоритм реализован на языке программирования Turbo-Pascal }

```
program ferz;
uses crt;
const desksize=8;
type sizeint=1..desksize;
      unuses=set of sizeint;
      combines=array[shortint] of sizeint;
var num:byte;
      combine:combines;
      unuse:unuses;

function attack(combine:combines):boolean;
var i,j:byte;
      rightdiag,leftdiag:combines;
begin
  attack:=false;
  for i:=1 to desksize do
    begin
      leftdiag[i]:=i+combine[i];
      rightdiag[i]:=i-combine[i];
    end;
  for j:=1 to desksize do
    for i:=1 to desksize do
      begin
        if (i<>j) and ((leftdiag[i]=leftdiag[j])or(rightdiag[i]=rightdiag[j]))
      then
        begin
          attack:=true;
          exit;
        end;
      end;
    end;
  end;
```

```
end;  
end;  
  
procedure output(combine:combinates);  
var i,j:byte;  
begin  
  
  for i:=1 to desksize do  
    for j:=1 to desksize do  
      begin  
        gotoxy(i,j);  
        if(combine[i]=j) then write(#2) else write(' ');  
      end;  
      readln;  
    end;  
  
  procedure create(num:byte; unuse:unuses; combine:combinates);  
  var i:byte;  
  begin  
    if num<=desksize then  
      for i:=1 to desksize do  
        begin  
          if i in unuse then  
            begin  
              combine[num]:=i;  
              create(num+1,unuse-[i],combine);  
            end;  
          end  
        else if not attack(combine) then output(combine);  
      end;  
    end;  
  
  begin  
    textmode(c40);  
  
    clrscr;  
    unuse:=[1..desksize];  
    create(1,unuse,combine);  
  end.
```

Сортировки

В лекциях 14 и 15 мы рассматривали более подробно различные способы сортировки. Здесь мы напомним некоторые из них и приводим пример программ.

Сортировка упорядочивает совокупность объектов в соответствии с заданным отношением порядка. **Ключ сортировки** — поле или группа полей элемента сортировки, которые используются при сравнении во время сортировки. **Сортирующая последовательность** — схема упорядочивания. Например, можно взять последовательность символов алфавита, задающую способ упорядочения строк этого алфавита.

Способ сортировки: сортировка по возрастанию, сортировки по убыванию. Методы сортировки:

- метод прямого выбора;
- метод пузырька;
- метод по ключу;
- сортировки слиянием;
- сортировки Батчера.

Сортировка по возрастанию — это сортировка, при которой записи упорядочиваются по возрастанию значений ключевых полей.

Сортировка по убыванию — это сортировка, при которой записи упорядочиваются по убыванию значений ключевых полей.

Сортировка методом пузырька (пузырьковая сортировка) — способ сортировки, заключающийся в последовательной перестановке соседних элементов сортируемого массива.

Сортировка по ключу — это сортировка записей с упорядочением по значению указанного поля или группы полей.

Сортировка слиянием — это внешняя сортировка, при которой на первом этапе группы записей сортируются в оперативной памяти и записываются на несколько внешних носителей; на втором этапе упорядоченные группы сливаются с нескольких внешних носителей на один носитель данных. **Носитель данных** — материальный объект, предназначенный для хранения данных, или среда передачи данных.

Сортировка Батчера — это сортировка, внутренний алгоритм которой работает за время $O(N \cdot \log(N))$.

Программа 5. Сортировка массива по возрастанию методом пузырька.

```
//Данные, которые нужно отсортировать, берутся
//из файла "massiv.txt", результат записывается
```

```
//в массив int mas['K'] и выводится на экран
// Алгоритм реализован на Turbo C++.
#include <conio.h>
#include <stdio.h>
#define K 1000; //Размер массива

int mas['K'];
int n;
void puzirek()
//функция сортирует массив по возрастанию методом пузырька
{
    int i,j,t;
    for(i=0;i<n;i++)
        for(j=1;j<n-i;j++)
            if(mas[j]<mas[j-1])
            {
                t=mas[j];
                mas[j]=mas[j-1];
                mas[j-1]=t;
            }
}

int main()
{
    clrscr();
    FILE *filePointer=fopen("massiv.txt","r");
    int i=0;
    while (!feof(filePointer))
    {
        fscanf(filePointer,"%d",&mas[i]);
        i++;
    }
    n=i;
    puzirek();
    for(i=0;i<n;i++)
        printf("%d ",mas[i]);
    //scanf("%d",&n);
    getch();
    return 0;
}
```

Программа 6. Пузырьковая сортировка и сортировка методом прямого выбора.

{Сортировка. Алгоритм реализован на языке
программирования Turbo-Pascal}

uses crt;

var

M, N : array[0..10] of integer;

i:integer;

procedure Input;

begin

for i := 0 to 10 do

begin

writeln('Число');

readln(M[i]); {Ввод массива}

end;

end;

Procedure Sort1; {Пузырьковый метод сортировки}

var

q,i,x:integer;

begin

for i:=10 downto 0 do

begin

for q:=0 to 10 do

if m[q]<m[q+1] then

begin

x:=M[q];

M[q]:=M[q+1];

M[q+1]:=x

end;

end;

end;

procedure Sort2; {Метод прямого выбора}

var

i,j,k,x:integer;

begin

for i:=0 to 9 do

```

begin
  k:=i;
  x:=M[i];
  for j:=i+1 to 10 do
    if M[j] > x then begin k:=j; x:=M[k];
    end;
  M[k]:= M[i];
  m[i]:=x;
  end;
end;

{-----}
begin
  clrscr;
  input; {Ввод исходного массива}
  writeln('Исходный массив');
  for i:=0 to 10 do write(m[i], ' '); {Вывод исходного массива}
  writeln;
  Sort1; {Сортировка массива методом пузырька}
  writeln ('Сортированный массив');
  for i:=0 to 10 do write(m[i], ' '); {Вывод отсортированного массива}
  input; {Ввод исходного массива}
  writeln('Исходный массив');
  for i:=0 to 10 do write(m[i], ' '); {Вывод исходного массива}
  writeln;

  sort2;
  writeln ('Сортированный массив методом прямого выбора');
  for i:=0 to 10 do write(m[i], ' '); {Вывод отсортированного массива}
  readln;
end.

```

Программа 7. Сестры.

```

//Две строки матрицы назовем сестрами, если совпадают
//множества чисел, встречающихся в этих строках. Программа
//определяет всех сестер матрицы, если они есть,
//и выводит номера строк. Алгоритм реализован на Turbo C++.
#include <graphics.h>
#include <stdlib.h>
#include <string.h>

```

```
#include <stdio.h>
#include <conio.h>
//#include <math.h>
#include <dos.h>
#include <values.h>
#include <iostream.h>

const n=4, //кол-во строк
      m=4; //столбцов

int m1[n][m];
//исходный массив
struct mas{int i,i1;};
//i-индекс сравниваемой строки с i1 строкой
mas a[n*2];
//массив типа mas, где будут лежать сестры, пример)
//a[1].i и a[1].i1 - сестры

void main()
{clrscr();
int i,j;
randomize();
for( i=0;i<n;i++)
for( j=0;j<m;j++)
m1[i][j]=random(2);
//случайным образом в массив заносим цифры

for(i=0;i<n;i++)
{printf("\n %d ",i);
for(int j=0;j<m;j++)
printf(" %d",m1[i][j]);
//распечатываем этот массив
}
int min,
p;
//индекс минимального элемента после s-го элемента i-ой строки
//сортировка строк массива по возрастанию
for(i=0;i<n;i++)//i-сортировка i-ой строки
{
for(int s=0;s<m-1;s++)
{min=m1[i][s+1];
```

```

for(int j=s;j<m;j++)
    if(m1[i][j]<=min){min=m1[i][j];p=j;}
//запоминаем минимальный элемент в ряде после s-го элемента
if(m1[i][s]>=min)
    {m1[i][p]=m1[i][s];m1[i][s]=min;}
//меняем местами s-й и p-й элемент,если s-й>p-го(минимального)
}

}

printf("\n");
for(i=0;i<n;i++)
{printf("\n %d ",i);
for(int j=0;j<m;j++)
    printf(" %d",m1[i][j]);
//выводим отсортированный массив
}
int s=0
//сколько элементов в i-й строке совпадают с эл-ми i1 строки, k=0;
//сколько строк совпали
int i1;
for(i=0;i<n-1;i++)          //верхняя строка i
for( i1=i+1;i1<n;i1++)      //нижняя строка i1
{s=0;
for(int j=0;j<m;j++)
    //сравнение идет по j-му столбцу
//    !    !
    if(m1[i][j]==m1[i1][j])s++;
//если соответствующие элементы в i-й и i1-й строки
//совпадают, то кол-во совпавших увеличивается на 1
if(s==m){a[k].i=i;a[k].i1=i1;k++;}
//если все элементы i-й и i1-й строки совпали, то они сестры
}

printf("\nСестры :");
for(i=0;i<k;i++)
    printf("\n %d и %d",a[i].i,a[i].i1);
    //распечатываем a[i].i-ю и a[i].i1-ю сестру
getch();
}

```


Программа 8. Поиск узоров из простых чисел.

//Построить матрицу A(15 X 15)таким образом: A(8,8)=1, затем
 //по спирали против часовой стрелки, увеличивая
 //значение очередного элемента на единицу и выделяя
 //все простые числа красным цветом, заполнить матрицу
 //Алгоритм реализован на Turbo C++.

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
  clrscr();
  int mas[15][15];
  int n=1,x=6,y=6,k=1;
  int i,j;
  while(1){
    mas[x][y]=k++;
    switch(n){
      case 1: x++;break;
      case 2: y--;break;
      case 3: x--;break;
      case 4: y++;break;
    }
    if(x==15) break;

    if(x==y && x<6) n=4;
    else if(x+y==12 && x<6) n=1;
    else if(x+y==12 && x>6) n=3;
    else if(x==y+1 && x>6) n=2;

  }

  for(i=0;i<15;i++)
  {
    for(j=0;j<15;j++)
    {
      textcolor(12);
      if(mas[j][i]>2)
        for(k=2;k<mas[j][i];k++)
```

```
    if(mas[j][i]%k==0) textcolor(15);
    cprintf("%3d ",mas[j][i]);
}
printf("\n");
}

getch();
}
```

Программа 9. Сортировка строк матрицы.

//Сортировка строк матрицы. В каждой строке подсчитывается
//сумма простых чисел. Полученный вектор упорядочивается
//по возрастанию. Строки матрицы переставляются по
//по возрастанию. Алгоритм реализован на Turbo C++.

```
#include<stdio.h>
#include<conio.h>
```

```
#define n 5
```

```
struct summa
{
    int value;
    int idx;
} sum,massum[n],a;
```

```
void main(void){
    clrscr();
    int mas1[n][n],mas[n][n]={1,1,1,1,1},
        {3,16,11,6,4},
        {8,10,15,23,1},
        {3,8,10,15,3},
        {7,3,20,15,10}};
```

```
int i,j,k,flag;
```

```
for(i=0;i<n;i++){
    sum.value=0;
    for(j=0;j<n;j++){
        flag=0;
        if(mas[i][j]>2)
            for(k=2;k<mas[i][j];k++)
```

```

    if(mas[i][j]%k==0) flag=1;
    if(flag==0) sum.value=sum.value+mas[i][j];
}
sum.idx=i;
massum[i]=sum;
}

```

```

for(i=0;i<n-1;i++)
    for(j=0;j<n-1-i;j++){
        if (massum[j].value>massum[j+1].value){
            a=massum[j];
            massum[j]=massum[j+1];
            massum[j+1]=a;
        }
    }
}

```

```

for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        mas1[i][j]=mas[massum[i].idx][j];

```

```

for(i=0;i<n;i++){
    for(j=0;j<n;j++)
        printf("%3d ",mas[i][j]);
    printf("\n");
}

```

```

printf("\n\n\n");

```

```

for(i=0;i<n;i++){
    for(j=0;j<n;j++)
        printf("%3d ",mas1[i][j]);
    printf("\n");
}
getch();
}

```

Задача о назначениях (задачи выбора)

Эта задача состоит в следующем. Пусть имеется n работ и n кандидатов для выполнения этих работ. Назначение кандидата i на работу j связано с затратами c_{ij} ($i, j = 1, 2, \dots, n$). Требуется найти назначение канди-

датов на все работы, дающее минимальные суммарные затраты; при этом каждого кандидата можно назначить только на одну работу и каждая работа может быть занята только одним кандидатом.

Иначе говоря, решение этой задачи представляет собой перестановку (p_1, p_2, \dots, p_n) чисел $(1, 2, \dots, n)$; каждое из производимых назначений описывается соответствием $i \rightarrow p_i$ ($i = 1, \dots, n$). Указанные условия единственности при этом автоматически выполняются, и нашей целью является минимизация суммы

$$\sum_{i=1}^n c_{ip_i} \quad (17.1)$$

по всем перестановкам (p_1, p_2, \dots, p_n) .

Перед нами типичная экстремальная комбинаторная задача. Ее решение путем прямого перебора, то есть вычисления значений функции 17.1 на всех перестановках и сравнения, практически невозможно при сколько-нибудь больших n , поскольку число перестановок равно $n! = 1 \cdot 2 \cdot 3 \cdots (n-1)n$. Попытаемся свести дело к линейному программированию.

Конечное множество, на котором задана целевая функция 17.1, представляет собой множество всех перестановок чисел $(1, 2, \dots, n)$. Как известно, каждая такая перестановка может быть описана точкой в n^2 -мерном евклидовом пространстве; эту точку удобнее всего представить в виде $n \times n$ -матрицы $X = \|x_{ij}\|$. Элементы x_{ij} интерпретировать следующим образом:

$$x_{ij} = 1, \text{ если } i\text{-й кандидат назначается на } j\text{-ю работу,} \quad (17.2)$$

в противном случае.

Элементы матрицы должны быть подчинены двум условиям:

$$\sum_{j=1}^n x_{ij} = 1, \quad i = 1, 2, \dots, n, \quad (17.3)$$

$$\sum_{i=1}^n x_{ij} = 1, \quad j = 1, 2, \dots, n. \quad (17.4)$$

Условия 17.3 и 17.4 говорят о том, что в каждой строке и в каждом столбце матрицы X имеется ровно по одной единице. Говоря неформально, условие 17.3 означает, что каждый кандидат может быть назначен только на одну работу, а условие 17.4 — что каждая работа предназначена только для одного кандидата. (Матрицу перестановок можно получить из единичной матрицы путем некоторой перестановки ее строк.)

Теперь задача заключается в нахождении чисел x_{ij} , удовлетворяющих условиям 17.2, 17.3, 17.4 и минимизирующих суммарные затраты 17.1, которые теперь можно переписать в виде

$$\sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij}. \quad (17.5)$$

Казалось бы, что к полученной задаче методы линейного программирования непосредственно применить нельзя, ибо в силу условий 17.2 она формально является целочисленной. Заменим условие 17.2 на условие неотрицательности переменных

$$x_{ij} > 0. \quad (17.6)$$

Тем самым мы получаем обычную задачу линейного программирования. В нашем случае требование целочисленности 17.2 будет выполняться автоматически.

Программа 10. Назначение на работу.

program one; {Назначение на работу.

Рассматривается случай: 10 работ и 10 желающих.

реализовано на Turbo-Pascal}

uses crt;

const n=10;

var C : array [1..n,1..n] of integer;

 T : array [1..n] of integer;

 M : array [1..n,1..4] of integer;

 Sum,tmj,z,min,i,j,tmp:integer;

begin

 clrscr;

 randomize;

 write('work - ');

 for i:=1 to n do write(i:2, ' ');

 for i:=1 to n do begin

 writeln;

 write(i:2, ' man ');

 for j:=1 to n do begin

 C[i,j]:=random(100);

 {if M[i,j]>max then max:=M[i,j];}

 {if C[i,j]<min then begin M[1]:=C[i,j]; M[2]:=i; M[3]:=j; end; }

 write(C[i,j]:2, ' ');

```
end;
end;
writeln;

for j:=1 to n do T[j]:=0;
Sum:=0;
for i:=1 to n do begin
writeln;
write(i:2, ' man ');
min:=100;
for j:=1 to n do begin
if (C[i,j]<min) and (T[j]=0) then begin min:=C[i,j]; M[i,1]:=i;
M[i,2]:=j; M[i,3]:=C[i,j]; tmj:=j;

end;

write(C[i,j]:2, ' ');
end;
T[tmj]:=1;
{M[i,3]:=min;}
Sum:=Sum+M[i,3];
write(' $=',M[i,3]:2, ' man=',M[i,1], ' job=',M[i,2]);
end;
writeln;
{for i:=1 to n do begin
for j:=1 to n do begin
if (i<>j) and (M[i,2]=M[j,2]) then begin
M[j,3]:=C[j,1];
for z:=1 to n do begin
if (M[j,3]>C[j,z]) and (z<>M[j,2]) then begin M[j,3]:=C[j,z];
M[j,2]:=z; end;
end;
end;
end;
writeln(' $=',M[i,3]:2, ' man=',M[i,1], ' job=',M[i,2]);
end;
}
write('sum=',Sum);
readln;
end.
```

Программа 11. Назначение на работу.

```

/*
Назначение на работу.
Рассматривается случай: 6 работ и 6 желающих.
*/

//Назначение на работу. Реализовано на Turbo C++.
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <conio.h>
#define k 6
int Sum,tmj,i,j,zj,min,tmp,min2,tmj2,p,q,ki;
int M[k][4], C[k][k], T[k][2], Temper[k][2];
char a;
/*struct myst
{int cel;
float rac;
};
myst ctpyk[k];*/
main()
{

Sum=0;
min=100;
for(i=1;i<k;i++)
{ T[i][1]=0;
printf("\n");
for(j=1;j<k;j++)
{C[i][j]=rand()/1000 +1;
// printf(" %d ", C[i][j]);
}
}

for(i=1;i<k;i++)
{
min=100;
printf("\n");
for(j=1;j<k;j++)
{

if(C[i][j]<min/* && T[j][1]==0*/)

```

```

{
    if(T[j][1]==0)
    {
        min=C[i][j]; //m[i][1] - 4el, m[i][2] -job, m[i][3]
- stoimost.
        M[i][1]=i;
        M[i][2]=j;
        M[i][3]=C[i][j];
        tmj=j;
    }
/* else
{
    if(C[i][j]<C[T[j][2]][j])
    {
        ki=T[j][2];
        T[j][2]=0;
        // T[j][1]=0;
        min=C[i][j];
        M[i][1]=i;
        M[i][2]=j;
        M[i][3]=C[i][j];
        tmj=j;
        for(zj=1;zj<k;zj++)
        {
            min2=100;
            if(C[ki][zj]<min2 && zj!=tmj &&
T[zj][1]==0)
            {
                min2=C[ki][zj];
                tmj2=zj;
                M[ki][1]=ki;
                M[ki][2]=zj;
                M[ki][3]=C[ki][zj];
            }

        }
        T[tmj2][2]=ki;
        T[tmj2][1]=min2;
    }
}

```



```

    }*/

}

    printf(" %d ", C[i][j]);
}

    T[tmj][2]=i;
    T[tmj][1]=min;
//na4alo mega funkcii
/* if(C[i][j]<min && T[j][1]!=0)
{
    for(p=1;p<k;p++)
    {
        if(C[T[tmj][2]][p]

    }
}
*/
//konec.

    Sum=Sum+M[i][3];
    printf("    $= %d, man= %d, job= %d ",M[i][3],M[i][1],M[i][2]);

}

/* for(i=0;i<k;i++)
{ctpyk[i].cel=rand();
 ctpyk[i].rac=rand()/1000;
 printf("%d %f \n", ctpyk[i].cel, ctpyk[i].rac);}

*/
    scanf("%d",a);
    return 0;
}

```

Ханойская башня

В лекции 11 дана реализация алгоритма «Ханойская башня». Здесь используется другой подход. В программе реализован пользовательский интерфейс с развитым эргономическим компонентом. Пользователю предоставляется возможность самому решить поставленную задачу. В программе использована работа с видеостраницами.

Постановка задачи.

На стержне в исходном порядке находится дисков, уменьшающихся по размеру снизу вверх. Диски должны быть переставлены на стержень в исходном порядке при использовании в случае необходимости промежуточного стержня для временного хранения дисков. В процессе перестановки дисков обязательно должны соблюдаться правила: одновременно может быть переставлен только один самый верхний диск (с одного из стержней на другой); ни в какой момент времени диск не может находиться на другом диске меньшего размера.

Программа 12. Ханойская башня.

{Программа реализована с помощью абстрактного типа данных – стек для произвольного числа дисков. Число колец задается константой maxc. Программа написана на языке программирования Turbo-Pascal}

```
program Tower;  
uses Crt, Graph;
```

```
const maxc = 4; {Максимальное число колец на одной башне}
```

```
type TTower = record  
  num: byte;  
  sizes: array[1..maxc] of byte;  
  up: byte;  
end;
```

```
var Towers: array[1..3] of TTower;  
  VisVP, ActVP: byte; {видимая и активная видеостраницы}  
  ActTower: byte;  
  Message: String;  
  Win: boolean;  
  font1: integer;
```

```
function CheckEnd: boolean;  
var res: boolean;
```

```
    i: byte;  
begin  
    res:=False;  
    if (Towers[2].num=maxc) or (Towers[3].num=maxc) then res:=true;  
    CheckEnd:=res;  
end;
```

```
procedure BeginDraw;  
begin  
    SetActivePage(ActVP);  
end;
```

```
procedure EndDraw;  
begin  
  
    VisVP:=ActVP;  
    SetVisualPage(VisVP);  
    if VisVP=1 then ActVP:=0 else ActVP:=1;  
end;
```

```
procedure Init;  
var grDr, grM: integer;
```

```
    ErrCode: integer;  
    i: integer;  
begin  
    grDr:=VGA;  
    grM:=VGAMed;  
    InitGraph(grDr, grM, 'c:\borland\tp\bgi');  
    ErrCode:=GraphResult;  
    if ErrCode <> grOk then  
    begin  
        WriteLn('Graphics error:', GraphErrorMsg(ErrCode));  
        Halt;  
    end;
```

```
    Towers[1].num:=maxc;  
    Towers[1].up:=0;  
    for i:=0 to maxc-1 do  
        Towers[1].sizes[i+1]:=maxc-i;
```

```
Towers[2].num:=0;
Towers[2].up:=0;
for i:=0 to maxc-1 do
  Towers[2].sizes[i+1]:=0;

Towers[3].num:=0;
Towers[3].up:=0;
for i:=0 to maxc-1 do
  Towers[2].sizes[i+1]:=0;

ActTower:=1;
VisVP:=0; ActVP:=1;
SetVisualPage(VisVP);
SetActivePage(ActVP);
Message:='';
Win:=False;
end;

procedure Close;
begin
  closegraph;
end;

procedure DrawTower(x, y: integer; n: integer);
var i: integer;
begin
  if n=ActTower then
    SetColor(yellow);
    Line(x, y, x, y+15+maxc*15);
    for i:=1 to Towers[n].num do
      begin
        Rectangle(x-10*Towers[n].sizes[i], y+15+15*(maxc-i+1),
x+10*Towers[n].sizes[i], y+15+15*(maxc-i))
      end;

    if Towers[n].up<>0 then
      begin
        Rectangle(x-10*Towers[n].up, y-15, x+10*Towers[n].up, y-30);
      end;

    SetColor(White);
```

```
end;
```

```
procedure DrawInfo;
```

```
begin
```

```
    OutTextXY(50, 20, 'Ханойская башня.');
```

```
    OutTextXY(80, 40, 'Работа с программой: стрелки влево-вправо -  
    выбор башни');
```

```
    OutTextXY(130, 60, 'текущая башня выделяется желтым цветом');
```

```
    OutTextXY(60, 80, 'стрелка вверх - поднять кольцо, стрелка  
    вниз - положить кольцо');
```

```
    OutTextXY(80, 100, '(две последние операции выполняются для  
    активной башни)');
```

```
end;
```

```
procedure Draw;
```

```
begin
```

```
    BeginDraw;
```

```
    ClearDevice;
```

```
    OutTextXY(180, 140, Message); Message:='';
```

```
    DrawTower(150, 200, 1);
```

```
    DrawTower(300, 200, 2);
```

```
    DrawTower(450, 200, 3);
```

```
if win then
```

```
begin
```

```
    SetTextStyle(GothicFont, HorizDir, 8);
```

```
    SetColor(Red);
```

```
    Outtextxy(70, 0, 'Congratulations');
```

```
    Outtextxy(160, 70, 'You win');
```

```
    SetTextStyle(DefaultFont, HorizDir, 1);
```

```
    SetColor(White);
```

```
    OutTextXY(250, 330, 'Press any key');
```

```
end
```

```
else
```

```
    DrawInfo;
```

```
EndDraw;
```

```
end;
```

```
procedure MainCycle;
```

```
var ch: char;
```

```
ex: boolean;
up: byte;
begin
  ex:=False;
  repeat
    if KeyPressed then
      begin
        ch:=ReadKey;
        case ch of
          #27: begin
              Ex:=True;
            end;
          #77: begin
              up:=Towers[ActTower].up;
              Towers[ActTower].up:=0;
              inc(ActTower);
              if ActTower>3 then ActTower:=1;
              Towers[ActTower].up:=up;
            end;
          #75: begin
              up:=Towers[ActTower].up;
              Towers[ActTower].up:=0;
              dec(ActTower);
              if ActTower<1 then ActTower:=3;
              Towers[ActTower].up:=up;
            end;
          #80: begin {вниз}
              if Towers[ActTower].up<>0 then
                begin
                  if Towers[ActTower].num=0 then
                    begin
                      Towers[ActTower].num:=Towers[ActTower].num+1;
                    end;
                end;
            end;
        end;
      Towers[ActTower].sizes[Towers[ActTower].num]:=
      Towers[ActTower].up;
      Towers[ActTower].up:=0;
    end
  else
    begin
      if
        Towers[ActTower].sizes[Towers[ActTower].num]>Towers[ActTower].up
```

```

    then
        begin
            Towers[ActTower].num:=Towers[ActTower].num+1;

Towers[ActTower].sizes[Towers[ActTower].num]:=
Towers[ActTower].up;
            Towers[ActTower].up:=0;
        end
    else
        Message:='Это кольцо сюда опускать нельзя';
    end;
end;
end;
#72: begin    {вверх}
    if Towers[ActTower].num<>0 then
        begin
            Towers[ActTower].num:=Towers[ActTower].num-1;

Towers[ActTower].up:=
Towers[ActTower].sizes[Towers[ActTower].num+1];
            Towers[ActTower].sizes[Towers[ActTower].num+1]:=0;
        end;
    end;
end;
if CheckEnd then
    begin
        Win:=True;
        ex:=True;
    end;
    Draw;
end;
until ex;
end;

begin
    Init;
    Draw;
    MainCycle;
    if win then repeat until keypressed;
    Close;
end.

```

Литература

- [1] Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов М.: Мир, 1979.
- [2] Диниц Е.А. Алгоритм решения задачи о максимальном потоке в сети со степенной оценкой. Докл. Академии наук СССР, Серия Мат., Физ., 1970, 194, 4, с. 754-757.
- [3] Floyd R. W. Algorithm 97: Shortest path. Comm. ACM. 1962, 5, s. 345.
- [4] Floyd R. W. Algorithm 245: treesort 3. Comm. ACM. 1964, 7, s. 701.
- [5] Ford L. R. Network flow theory. The Rand Corp., P-923, August 1956.
- [6] Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
- [7] Харари Ф. Теория графов. М.: Мир, 1973.
- [8] Lipski W. More on permutation generation methods. Computing, 1979, 23, s. 357-365.
- [9] Lum V. Y. Multiattribute retrieval with combined indexes. Comm. ACM, 1970, 13, s. 660-665.

Учебное издание

Костюкова Нина Ивановна
ГРАФЫ И ИХ ПРИМЕНЕНИЯ.
КОМБИНАТОРНЫЕ АЛГОРИТМЫ ДЛЯ
ПРОГРАММИСТОВ
Учебное пособие

Литературные редакторы *Е. Петровичева, С. Перепелкина*

Корректор *Ю. Голомазова*

Компьютерная верстка *А. Ширококов*

Обложка *М. Автономова*

Подписано в печать 30.10.2006. Формат 60х90 ¹/₁₆.

Гарнитура Таймс. Бумага офсетная. Печать офсетная.

Усл. печ. л. 19,5. Тираж 2000 экз. Заказ № 6530

ООО «ИНТУИТ.ру»

Интернет-Университет Информационных Технологий, www.intuit.ru

Москва, Электрический пер., 8, стр.3.

E-mail: admin@intuit.ru, <http://www.intuit.ru>

ООО «БИНОМ. Лаборатория знаний»

Москва, проезд Аэропорта, д. 3

Телефон: (499) 157-1902, (495) 157-5272

E-mail: Lbz@aha.ru, <http://www.Lbz.ru>

При участии ООО «ПФ «Сашко»

Отпечатано в ОАО «ИПК «Ульяновский Дом печати»

432980, г. Ульяновск, ул. Гончарова, 14

**Графы и их применение.
Комбинаторные алгоритмы
для программистов
УЧЕБНОЕ ПОСОБИЕ**



В курсе лекций рассматривается широкий спектр комбинаторных и теоретико-графовых алгоритмов. Комбинаторные методы используются как в дискретной математике, так и в чисто математических вопросах – теории групп, неассоциативных алгебр и так далее. Материал лекций организован так, что знакомство с графами и комбинаторными алгоритмами происходит в процессе решения самых разнообразных задач. Если в начале рассматриваются задачи частного характера, иллюстрирующие комбинаторику и теорию графов (связь с жизнью), то в конце приводятся прикладные разделы, имеющие практическое значение в различных науках. Общие принципы построения алгоритмов иллюстрируются на различных задачах, среди которых наибольшее внимание уделяется алгоритмам поиска и сортировки и алгоритмам на графах. Описание алгоритмов дано на языках Паскаль, Си, Си ++.

Предлагаемые лекции отличаются удачное сочетание компактности изложения с весьма подробной разработкой темы, что позволяет легко и быстро ознакомиться с предметом.

Нина Костюкова

ISBN 978-5-94774-545-0



9 785947 745450