МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
БАШКИРСКИЙ ГОСУДАРСТВЕННЫЙ ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ
им. М.АКМУЛЛЫ

**А.С. Филиппова, С.С. Поречный,
Р.Р. Рамазанова**

ОСНОВЫ КОМБИНАТОРНЫХ АЛГОРИТМОВ

Учебное пособие

Уфа 2018

УДК 519.161(07)
ББК 22.18 (я7)
Ф53

Печатается по решению учебно-методического совета Башкирского государственного педагогического университета им. М.Акмуллы

Филиппова А.С., Поречный С. С., Рамазанова Р.Р. Основы комбинаторных алгоритмов: учеб. пособие [Текст]. – Уфа: Изд-во БГПУ, 2018. – 131 с.

Описаны типовые задачи комбинаторной оптимизации. Основное внимание уделено комбинаторным алгоритмам и задачам исследования операций: кратчайшие расстояния на графах, остовные деревья, сети, сетевое планирование, потоки в сетях, паросочетания, понятие NP-полных задач.

Предназначено для студентов очной и заочной форм обучения, обучающихся по направлению 09.03.03 Прикладная информатика, направленность (профиль) «Прикладная информатики в менеджменте», изучающих дисциплины «Дискретная математика», «Комбинаторные алгоритмы», «Прикладные методы оптимизации», «Алгоритмы и методы управления логистическими процессами». Кроме того, будет полезна студентам по направлению 09.04.03 Прикладная информатика, направленность профиль «Прикладная информатика в психологии» (уровень магистратуры) при изучении некоторых разделов дисциплины «Математические и инструментальные методы поддержки принятия решений».

Рецензенты:

Р.Ф. Маликов, *д-р. физ.-мат. наук, профессор (БГПУ им. М. Акмуллы);*

Е.Р. Колясникова, *канд. физ.-мат. наук, доцент (БГУ).*

ISBN 978-5-4221-0441-3

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Цель учебного пособия – освоение теоретического материала – знаний, необходимых для бакалавров и магистрантов в области решения прикладных задач оптимизации и принятия решений; приобретение навыков применения классических комбинаторных алгоритмов и их модификаций для поиска решения прикладных задач оптимизации; формирование у студентов умений и навыков, необходимых при исследовании и решении прикладных математических и инженерно-технических проблем.

Для достижения указанной цели решаются следующие *задачи*:

- ✓ формирование знаний классических комбинаторных алгоритмов эффективного решения прикладных задач оптимизации;
- ✓ формирование умений и навыков использования изученных комбинаторных алгоритмов для решения типовых задач, и оценки полученных результатов.

Данное учебное пособие является дополненным и переработанным в соответствии с учебными планами по направлению 09.03.03 и 09.04.03 переизданием учебного электронного издания локального доступа (Уфимск. гос. авиац. техн. ун-т; – Уфа, 2013. Гос. рег. № 0321302987) «Основы комбинаторных алгоритмов (учебное пособие)» авторов Филипповой А. С., Поречного С.С.

Учебное пособие базируется на курсах лекций по дисциплинам «Комбинаторные алгоритмы», «Прикладные методы оптимизации». Пособие посвящено классическим комбинаторным алгоритмам, которые встречаются при решении проблем в области информатики, исследовании операций и дискретной математике. Особое внимание уделяется идеям, лежащим в основе алгоритмов, для понимания принципа процесса построения решений с возможностью модификаций алгоритмов для решения прикладных задач с дополнительными практическими условиями и ограничениями. Приводится детальный разбор процесса решения численных примеров для каждого изученного алгоритма. Каждый пример проиллюстрирован, что упрощает понимание сути изучаемых алгоритмов. Основу для материалов лекций составила книга Ху Т. Ч., Шинга М. Т. «Комбинаторные алгоритмы», перевод и научное редактирование которой выполнены в 2004 г. на кафедре математической логики и высшей алгебры Нижегородского государственного университета им. Н. И. Лобачевского.

Задачи комбинаторной оптимизации встречаются во многих разделах математики и информатики. В связи с этим описание алгоритмов для их решения представлено в учебниках, отражающих рассматриваемую тематику в разной степени и в различных научных направлениях. Кроме того, подобная литература издана в конце XX века. В данном учебном пособии собраны описания, характеристики основных комбинаторных алгоритмов из разделов: комбинаторного анализа, теории графов, дискретной математики, исследования операций. Приведены примеры их использования. Пособие отличает простой

стиль изложения и минимум формализма (но не в ущерб математической строгости), отбор материала и сбалансированный объем.

ВВЕДЕНИЕ

Комбинаторика – раздел математики, посвященный решению задач выбора и расположения элементов некоторого, обычно конечного множества в соответствии с заданными правилами. Каждое такое правило определяет способ построения некоторой конструкции из элементов исходного множества, называемой *комбинаторной конфигурацией*. Поэтому можно сказать, что целью комбинаторного анализа является изучение комбинаторных конфигураций, которое включает в себя вопросы существования комбинаторных конфигураций, алгоритмы их построения, оптимизацию таких алгоритмов, а также решение задач перечисления, в частности, определение числа конфигураций данного класса.

Простейшим примером комбинаторных конфигураций являются перестановки, сочетания и размещения. Большой вклад в систематическое развитие комбинаторных методов был сделан Г. Лейбницем (диссертация «Комбинаторное искусство»), Я. Бернулли (работа «Искусство предположений»), Л. Эйлером. Можно считать, что с появлением работ Я. Бернулли и Г. Лейбница, комбинаторные методы выделились в самостоятельную часть математики. В работах Л. Эйлера по разбиениям и композициям натуральных чисел на слагаемые было положено начало одному из основных методов перечисления комбинаторных конфигураций – методу производящих функций.

Возвращение интереса к комбинаторному анализу относится к 50-м гг. XX в. в связи с бурным развитием кибернетики и дискретной математики и широким использованием электронно-вычислительной техники. В этот период активизировался интерес к классическим комбинаторным задачам. Классические комбинаторные задачи – это задачи выбора и расположения элементов конечного множества, имеющие в качестве исходной некоторую формулировку развлекательного содержания типа головоломок. Например, в 1859 г. У. Гамильтон придумал игру «Кругосветное путешествие», состоящую в отыскании такого пути, проходящего через все вершины (города, пункты назначения) графа. Это одна из самых известных задач комбинаторики – задача коммивояжера.

1. ПОСТАНОВКИ, ОСОБЕННОСТИ КОМБИНАТОРНЫХ ЗАДАЧ И МЕТОДОВ ИХ РЕШЕНИЯ

1.1. Основные понятия комбинаторики

Комбинаторика возникла как «математика досуга»: с ее помощью подсчитывались шансы участников карточных и прочих азартных игр. Появление и развитие вычислительной техники резко увеличило возможности комбинаторики и расширило круг ее приложений.

В настоящее время комбинаторные методы применяются в теории вероятностей, статистике, экономике, биологии, химии, физике и других областях науки.

В задачах комбинаторного анализа исследуются дискретные множества, то есть множества, составленные из отдельных обособленных элементов. В большинстве случаев эти множества конечные, но не исключается и рассмотрение множеств, состоящих из бесконечного числа элементов. Особенностью комбинаторных задач является то, что в них преимущественное внимание уделяется двум видам операций: отбору подмножеств и упорядочению элементов. Эти две операции и являются основными комбинаторными.

Задача перечисления состоит в выделении элементов, принадлежащих некоторому заданному конечному множеству и удовлетворяющих некоторым свойствам. Например, описание всех расположений восьми одинаковых ладей на шахматной доске, при которых ладьи не бьют друг друга.

Задача пересчета состоит в нахождении числа таких элементов. Например, нахождение количества расположений в задаче о восьми ладьях.

Обычно в подобных задачах речь идет о комбинациях некоторых объектов, поэтому такие задачи называют *комбинаторными*.

Ошибочно полагать, что комбинаторные задачи элементарны. В наше время число комбинаторных задач и их разнообразие быстро растет. К их решению прямо или косвенно приводят многие практические задачи. При этом оказывается, что несмотря на заманчивую простоту постановки, комбинаторные задачи в большинстве очень трудны; многие из них не поддаются решению до сих пор.

📖 **Теорема.** (*Основной принцип комбинаторики*). Пусть подсчитывается количество объектов, каждый из которых строится в результате последовательного выполнения n действий. Первое действие может быть выполнено a_1 способами, второе – a_2 способами, ..., последнее – a_n способами. При этом количество способов выполнить каждое действие не зависит от того, какими были предыдущие действия. Тогда общее количество объектов равно

$$a_1 \cdot a_2 \cdot \ldots \cdot a_n.$$

Приведенная теорема называется *правилом произведения*.

🕮 **Теорема.** (*Комбинаторный принцип сложения*). Пусть A_1, A_2,..., A_m – попарно непересекающиеся множества, и пусть для каждого $i \in (1,...m)$ множество A_i содержит n_i элементов. Тогда количество вариантов выбора из A_1 или A_2 или ... или A_m равно

$$n_1 + n_2 + ... + n_m.$$

✍ **Замечание.** Неформально приведенные теоремы можно сформулировать следующим образом. Пусть существуют некоторые возможности построения комбинаторной конфигурации. Если эти возможности взаимно исключают друг друга, то их количества следует складывать, а если возможности независимы, то их количества следует перемножать.

В некотором смысле комбинаторику можно понимать как синоним термина «дискретная математика», то есть исследование дискретных конечных математических структур. Вычисления на дискретных конечных математических структурах, которые часто называют *комбинаторными вычислениями,* требуют комбинаторного анализа для установления свойств и выявления оценки применимости используемых алгоритмов.

Рассмотрим простой пример.

❖***Пример 1.1.*** Пусть некоторое агентство недвижимости располагает базой данных из *n* записей, причем каждая запись содержит одно предложение (что имеется) и один запрос (что требуется) относительно объектов недвижимости. Требуется найти пары записей, в которых предложение первой записи совпадает с запросом второй и одновременно предложение второй записи совпадает с запросом первой (так называемый «подбор вариантов обмена»).

Допустим, что используемая СУБД позволяет проверить вариант за одну миллисекунду. Понятно, что при использовании «лобового» алгоритма поиска вариантов (каждая запись сравнивается с каждой) потребуется $n(n-1)/2$ сравнений. Если n=100, то ответ будет получен всего за 4,95 сек. Но на практике n может составлять 100 000, и тогда ответ будет получен за 4 999 950 сек., что составляет почти 1389 ч. и вряд ли может считаться приемлемым. При этом важно отметить то, что была оценена только трудоемкость подбора *прямых* вариантов, а существуют еще случаи, когда число участников сделки больше двух.

Этот пример показывает, что практические задачи и алгоритмы требуют предварительного анализа и количественной оценки. Задачи обычно оцениваются с точки зрения *размера*, то есть общего количества различных вариантов, среди которых нужно найти решение, а алгоритмы оцениваются с точки зрения *сложности.* При этом различают:

– *сложность по времени* (или временную сложность) – количество необходимых шагов алгоритма;

– *сложность по памяти* (или емкостную сложность) – объем памяти, необходимый для работы алгоритма.

Во всех случаях основным инструментом такого анализа оказываются формулы и методы.

Во многих практических случаях возникает необходимость подсчитать количество возможных комбинаций объектов, удовлетворяющих определенным условиям. Такие задачи называются *комбинаторными.* Разнообразие комбинаторных задач не поддается исчерпывающему описанию, но среди них есть целый ряд особенно часто встречающихся, для которых известны способы подсчета.

Для формулировки и решения комбинаторных задач используются различные модели *комбинаторных конфигураций.* Рассмотрим следующие две наиболее популярные (изложение на «языке ящиков» и «языке функций»):

а) Дано n предметов. Их нужно разместить по m ящикам так, чтобы выполнялись заданные ограничения. Сколькими способами это можно сделать?

б) Рассмотрим множество функций $F : X \to Y$, где $|X| = n$, $|Y| = m$. Не ограничивая общности, можно считать, что $X = \{1,...,n\}$, $Y = \{1,...,m\}$, $F = (F(1),...,F(n))$, $1 \le F(i) \le m$. Сколько существует функций F, удовлетворяющих заданным ограничениям?

1) *Размещения*

Число всех функций (при отсутствии ограничений) или число всевозможных способов разместить n предметов по m ящикам называется *числом размещений* и обозначается $U(m,n)$.

Перестановка с повторениями – упорядоченная выборка, в которой элементы могут повторяться.

🕮 **Теорема.** $U(m,n) = m^n$.

Доказательство. Поскольку ограничений нет, предметы размещаются независимо друг от друга, и каждый из n предметов можно разместить m способами.

❖***Пример 1.2*.** При игре в кости бросаются две кости, и выпавшие на верхних гранях очки складываются. Какова вероятность получить 12 очков? Каждый возможный исход соответствует функции *F:* {1,2} –> {1,2,3,4,5,6} (аргумент – номер кости, результат – очки на верхней грани). Таким образом, всего возможно $U(6,2) = 6^2 = 36$ различных исходов. Из них только один (6+6) дает 12 очков. Таким образом, искомая вероятность равна 1/36.

2) *Размещения без повторений*

Число инъективных функций, или число способов разместить n предметов по m ящикам, не более чем по одному в ящик, называется *числом размещений без повторений* и обозначается A_m^n.

🕮 **Теорема.** $A_m^n = \dfrac{m!}{(m-n)!}$.

Доказательство. Ящик для первого предмета можно выбрать m способами, для второго – $m-1$ способами и т. д. Таким образом

$$A_m^n = m(m-1)\cdot\ldots\cdot(m-n+1) = \frac{m!}{(m-n)!}. \qquad (1.1)$$

По определению считается, что $A_m^n=0$ при $n>m$ и $A_m^0=1$.

✍ **Замечание.** Формула (1.1) слева выглядит сложной и незавершенной, формула справа – лаконичной и «математичной». Но формула – это частный случай алгоритма. При практическом вычислении левая формула оказывается намного предпочтительнее правой. Во-первых, для вычисления по левой формуле потребуется n умножений, а по правой – m умножений и одно деление. Поскольку $n<m$, левая формула вычисляется существенно быстрее. Во-вторых, число A_m^n растет довольно быстро и при больших m и n может не поместиться в разрядную сетку. Левая формула работает правильно, если результат помещается в разрядную сетку. При вычислении же по правой формуле переполнение может наступить «раньше времени» (то есть промежуточные результаты не помещаются в разрядную сетку, в то время как окончательный результат мог бы поместиться), поскольку факториал – очень быстро растущая функция.

❖ ***Пример 1.3.*** В некоторых видах спортивных соревнований исходом является определение участников, занявших первое, второе и третье места. Сколько возможно различных исходов, если в соревновании участвуют n участников?

Каждый возможный исход соответствует функции $F: \{1,2,3\} \to \{1,\ldots,n\}$ (аргумент – номер призового места, результат – номер участника). Таким образом, всего возможно $A_n^3 = n(n-1)(n-2)$ различных исходов.

3) *Перестановки*

Если $|X|=|Y|=n$, то существуют взаимно-однозначные функции $f: X \to Y$. Число взаимно-однозначных функций или *число перестановок n* предметов обозначается $P(n)$.

🕮 **Теорема.** $P(n) = n!$.

Доказательство:

$$P(n) = A(n,n) = n(n-1)\cdot\ldots\cdot(n-n+1) = n(n-1)\cdot\ldots\cdot 1 = n!.$$

❖ ***Пример 1.4.*** Сколькими способами можно сформировать очередь из 10 человек? Количество вариантов равно 10!.

4) *Сочетания*

Число строго монотонно возрастающих функций, или число размещений n неразличимых предметов по m ящикам не более чем по одному в ящик, то есть число способов выбрать из m ящиков n ящиков с предметами, называется *числом сочетаний* и обозначается C_m^n.

🕮 **Теорема**. $C_m^n = \dfrac{m!}{n!(m-n)!}$.

Доказательство. (Обоснование формулы) Сочетание является размещением без повторений неразличимых предметов. Следовательно, число сочетаний определяется тем, какие ящики заняты предметами, поскольку перестановка предметов при тех же занятых ящиках считается одним сочетанием. При этом, согласно правилу произведения, размещение A_m^n есть произведение сочетания из m по n на перестановку $P(n)$. Таким образом, сочетание из m по n C_m^n равно

$$C_m^n = \frac{A_m^n}{P(n)} = \frac{m!}{n!(m-n)!}.$$

(Сведение моделей). Число сочетаний является числом строго монотонно возрастающих функций, потому что любая строго монотонно возрастающая функция $F:1...n \to 1...m$ определяется набором своих значений, причем $1 \le F(1) < ... < F(n) \le m$. Другими словами, каждая строго монотонно возрастающая функция определяется выбором n чисел из диапазона $1...m$. Таким образом, число строго монотонно возрастающих функций равно числу
n-элементных подмножеств m-элементного множества, которое, в свою очередь, равно числу способов выбрать n ящиков с предметами из m ящиков.

По определению $C_m^n = 0$ при $n > m$.

❖***Пример 1.5.*** В начале игры в домино каждому играющему выдается 7 костей из имеющихся 28 различных костей. Сколько существует различных комбинаций костей, которые игрок может получить в начале игры?

Очевидно, что искомое число равно числу 7-элементных подмножеств 28-элементного множества. Имеем:

$$C_{28}^7 = \frac{28!}{7!(28-7)!} = \frac{28 \cdot 27 \cdot 26 \cdot 25 \cdot 24 \cdot 23 \cdot 22}{7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} = 1184040.$$

5) Сочетания с повторениями

Число монотонно неубывающих функций $F:1...n \to 1...m$, определяемых набором своих значений, причем $1 \le F(1) \le ... \le F(n) \le m$, или

число размещений n неразличимых предметов по m ящикам, называется *числом сочетаний с повторениями* и обозначается V_m^n.

🕮 **Теорема.** $V_m^n = C_{n+m-1}^n$.

Доказательство. Монотонно возрастающей функции $f:1...n \to 1...m$ однозначно соответствует строго монотонно возрастающая функция $f':1...n \to 1...(n+m-1)$. Это соответствие устанавливается следующей формулой: $f'(k) = f(k) + k - 1$.

❖ ***Пример 1.6.*** В палитре художника 8 различных красок. Художник берет кистью наугад любую из красок и ставит цветное пятно на ватмане. Затем берет следующую кисть, окунает её в любую из красок и делает второе пятно по соседству. Сколько различных комбинаций существует для шести пятен? Порядок пятен на ватмане не важен.

Пусть количество пятен первого цвета равно k_1, второго цвета $-k_2$, третьего $-k_3$ и т.д. Запишем каждое из этих чисел последовательностью из соответствующего количества единиц, а на границах между числами поставим нули. Так, если у нас первого цвета 1 пятно, второго – 3 пятна, третьего и четвёртого – ни одного, пятого и шестого – по одному пятну, а седьмого и восьмого – снова ни одного, то запись будет выглядеть следующим образом: 1011100010100. В этой цепочке содержится $n = 6$ единиц, $m - 1 = 8 - 1 = 7$ нулей – всего $m + n - 1 = 13$ цифр. Количество перестановок с повторениями этих цифр равно

$$V_m^n = C_{n+m-1}^n = \frac{(m+n-1)!}{n!(m-1)!} = \frac{13!}{6!\cdot 7!} = 1716.$$

1.2. Особенности комбинаторных методов

К числу современных задач, решаемых комбинаторными методами, относится задача дискретного программирования:

$$f\left(x^0\right) = \min f(x),\ x \in G,$$

множество допустимых решений которой конечно, т. е. $0 \le |G| = N < \infty$, где $|G|$ – число элементов множества G. В силу конечности G все допустимые решения можно пронумеровать и вычислить значение целевой функции $f\left(x^i\right),\ i = \overline{1, N}$, и найти наименьшее значение. Однако такой метод полного перебора при решении задач реализовать не всегда возможно, так как N может оказаться настолько большим, что этот перебор невыполним на ЭВМ любой мощности. Классической задачей такого типа, которую часто рассматривают как тестовую при разработке алгоритмов комбинаторной оптимизации, является *задача о коммивояжере.* Подробное описание этой задачи и

некоторых методов и алгоритмов ее решения приведено в главе 4. Далее перечислим типы задач, для которых применимы комбинаторные методы решения:

1) *задачи на размещения*: задачи о расположении, например, на плоскости предметов, обладающих некими свойствами (размерами и др.) с нахождением наилучшего размещения по заданным критериям;

2) *задачи о покрытиях и заполнениях*: задачи о заполнении заданных пространственных фигур меньшими телами заданных форм и размеров и др.;

3) *задачи о маршрутах*: отыскание кратчайшего пути и др.;

4) *комбинаторные задачи теории графов*: задачи сетевого планирования, например, задачи транспортных и электрических сетей, задачи об окрашивании графов, задачи о перечислении вершин и др.;

5) *перечислительные задачи*. В таких задачах речь идет о числе элементов, составляемых из данного множества при соблюдении определенных правил.

Особенность многих подобных задач, например, задачи о размещении, заполнении, о маршрутах, состоит в том, что способы решения имеют переборный характер. И для эффективной реализации перебора необходимо применять специальным образом организованные алгоритмы, которые получили название комбинаторных.

Основная идея таких алгоритмов состоит в выделении из множества допустимых решений подмножеств, не содержащих оптимальных решений. Отбрасывание таких подмножеств сокращает объем перебора. Процедуры выделения таких подмножеств и их отбрасывания (отсева) составляют *содержание* комбинаторного алгоритма.

Основная идея комбинаторных методов состоит в использовании конечности множества допустимых решений и замене полного их перебора сокращенным (направленным перебором). Если каким-либо образом удается показать, что подмножество $G' \subset G$ (где G – множество допустимых решений) не может содержать оптимальных решений, то в дальнейшем задача решается на множестве $x \in G \setminus G' = \{x : x \in A, x \notin B\}$.

Таким образом, главную роль в сокращении перебора играют оценка и отбрасывание подмножеств, заведомо не содержащих оптимальных решений. Эта идея реализуется путем последовательного разбиения множества всех допустимых решений на подмножества. При этом среди подмножеств, последовательно порождаемых на каждом шаге процесса, могут обнаружиться как подмножества, не содержащие допустимых решений, так и подмножества, не содержащие оптимальных решений. Отбрасывание таких подмножеств позволяет заменить полный перебор частичным и тем самым делает реализуемым вычислительный процесс. Таким образом, комбинаторные методы основаны на двух элементах:

- последовательное разбиение на подмножества;
- оценивание получаемых подмножеств.

Подмножество, которое не может быть отсеяно, подвергается дальнейшему разбиению. Комбинаторные методы различаются способами разбиения и способами оценивания, эти способы обычно связаны со спецификой решаемых классов задач. Правила, в соответствии с которыми происходит отсев подмножеств, заведомо не содержащих оптимальных решений, называют *правилами отсева*.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Приведите пример комбинаторной задачи перечисления.
2. Приведите пример комбинаторной задачи пересчета.
3. Какие принципы комбинаторики вы знаете?
4. Дайте определения перестановки, сочетания и размещения.
5. В чем отличие в сочетаниях и сочетаниях с повторениями?
6. Какие виды сложности существуют для оценки алгоритма?
7. Верно ли, что комбинаторные алгоритмы предназначены для решения оптимизационных задач, число допустимых решений которых неограниченно?
8. Приведете примеры задач, для которых используются комбинаторные методы решения.

2. КЛАССЫ ЗАДАЧ. КЛАССЫ СЛОЖНОСТИ АЛГОРИТМОВ

2.1. Сложность алгоритма

Исследование сложности алгоритма помогает понять степень его практической приемлемости. Сравнительный анализ сложности нескольких алгоритмов решения одной и той же задачи позволяет делать обоснованный выбор лучшего из них. Например, если для решения задачи предлагается новый алгоритм, то необходимы доводы, говорящие о его преимуществах в сравнении с известными ранее алгоритмами, и анализ сложности может предоставить такие доводы.

Здесь «сложность» является математическим термином, а не общим обозначением препятствия к выполнению алгоритма. С понятием сложности связываются затраты времени или памяти (ограничимся здесь рассмотрением временных затрат), соответствующие худшему случаю, или затраты в среднем; при этом необходимо оговорить, как определяется размер входа (размер входных данных) алгоритма и в чем измерять затраты при работе алгоритма над фиксированном входом.

Часто размер входа определяют как общее число символов в представлении входа, но возможны и другие варианты. Например, в задачах сортировки размер входа – это, как правило, количество элементов n входного массива, в задачах на графах – число вершин или число ребер входного графа, но можно число вершин и число ребер рассматривать и совместно как два параметра размера входа.

При фиксированном значении размера входа сами входы алгоритма могут варьироваться, при этом меняются и затраты; алгоритм может быть охарактеризован наибольшими или средними затратами. В обоих случаях сложность алгоритма – это функция размера входа или, соответственно, нескольких параметров размера входа. Для этого понятия используется термин *вычислительная сложность*, будем называть ее просто *сложностью*.

Понятие сложности является математическим уточнением расплывчатого термина «трудоемкость», с помощью которого, наряду с «быстродействием» и «эффективностью», иногда пытаются характеризовать алгоритмы. Принятие в качестве сложности именно затрат помогает оценить достаточность имеющихся вычислительных ресурсов для выполнения алгоритма.

Сложность является функцией числового (чаще целого), иногда нескольких аргументов.

Сложности многих алгоритмов трудно или вообще нельзя представить в простом «замкнутом» виде; кроме того, точное значение сложности алгоритма для каждого конкретного значения размера входа часто не представляет особого интереса, актуальным же является исследование роста сложности при возрастании размера входа. Поэтому в теории сложности широко используются асимптотическое оценивание.

Итак, кроме правильности и корректности алгоритма и программы важным фактором является *сложность алгоритма*. Допустим, что достигнуто соглашение о том, как измеряются эти затраты, тогда можно рассмотреть функцию затрат $C^T(x)$, соответствующую вычислениям, связанным с применением алгоритма к входу x. Вид функции может быть очень непростым, ее трудно исследовать методами математического анализа, потому что аргумент x не является числом и не принадлежит какому-либо метрическому пространству. Поэтому можно принять некоторую неотрицательную числовую характеристику $\|x\|$ возможных входов алгоритма, которую назовем *размером входа*.

🕮 **Определение**. Пусть на возможных входах x некоторого алгоритма определена неотрицательная числовая функция $\|x\|$ (размер входа). Пусть также определена целочисленная неотрицательная функция $C^T(x)$ временных затрат алгоритма. Тогда *временной сложностью* называется функция числового аргумента

$$T(n) = \max_{\|x\|=n} C^T(x),$$

областью изменения n как аргумента функции $T(n)$ является множество значений размера $\|\cdot\|$. Более полно каждая такая сложность именуется сложностью в *худшем случае*.

✍ **Замечание**. Значение $C^T(x)$ есть число каких-то операций, поэтому значение этой функции целочисленное.

Нахождение $T(n)$ для программы является сложной задачей. И обычно используют асимптотическую оценку этой функции, т. е. ее примерное поведение при больших n. Эту функцию сложности алгоритма обозначают O.

Итак, сложность алгоритма характеризует, насколько быстро растет количество элементарных операций с увеличением объема входных данных.

В настоящее время принято использовать виды оценки сложности вычисления (классы сложности), приведенные в табл. 2.1.

2.2. Сложность задач

Для решения различных задач комбинаторной оптимизации некоторые из алгоритмов весьма эффективны, другие – очень медлительны. И некоторые задачи просто не допускают эффективных алгоритмов. Возникает естественный вопрос: почему? Действительно ли, некоторые задачи по существу своему более сложные, чем другие, или пока для их решения не найдено эффективных алгоритмов?

Таблица 2.1.

Классы сложности алгоритмов

Порядок от сложного к	Название класса сложности	Математи-ческая	Примеры алгоритмов

простому		формула	
1	Факториальная	$n!$	Алгоритмы комбинаторики (перестановки и др.)
2	Экспоненциальная	k^n	Алгоритмы перебора
3	Полиномиальная	n^k	Алгоритмы простой сортировки
4	Линейный логарифм	$n \log n$	Алгоритмы быстрой сортировки
5	Линейная	$k \cdot n$	Перебор элементов массива
6	Логарифмическая	$k \log n$	Бинарный поиск
7	Константная	k	Обращение к элементу массива по индексу

k – константа, n – переменные (входные данные).

Есть, по крайней мере, две причины для классификации задач по сложности. Во-первых, если известно, что задача принадлежит классу очень сложных задач, то не стоит сосредотачиваться на получении точного решения, а можно попытаться получить приближенное решение, верхние и нижние оценки, воспользоваться эвристиками и т.д. Во-вторых, если задачи принадлежат одному классу, то иногда решение одной из них удается преобразовать в метод решения другой. Сложно определить, какая задача принадлежит какому классу. Две задачи могут казаться весьма похожими, однако одна из них может быть намного сложнее другой.

Для примера рассмотрим две задачи.

(1) Задано множество n попарно различных положительных целых чисел $x_1, x_2,\ldots, x_n$ (n четно). Необходимо разбить это множество на два подмножества мощности $n/2$ каждое так, чтобы разность между суммами элементов этих подмножеств была максимальной.

(2) Задано множество n попарно различных положительных целых чисел $x_1, x_2,\ldots,x_n$. Необходимо разбить это множество на два подмножества так, чтобы разность между суммами элементов этих подмножеств была минимальной.

Задачу (1) можно решить, сортируя числа $x_1, x_2,\ldots, x_n$ и помещая $n/2$ меньших чисел в первое подмножество, а $n/2$ больших – во второе. Данная процедура выполнима за время $O(n\log n)$. Другой способ – найти медиану и использовать ее для разбиения чисел на два множества. Известен алгоритм, который находит наибольшее число среди n целых чисел за время $O(n)$, поэтому второй метод решения задачи (1) использует время $O(n)$. Так как, надо полагать, каждый алгоритм, решающий эту задачу, должен проверить каждое число, то $O(n)$ является нижней оценкой. На самом деле, можно решить обобщение этой задачи, когда требуется *разбить множество* на подмножества мощности k и $n – k$*:* для этого достаточно найти k-е наибольшее число.

Задачу (2) можно решить, разбивая при всех k исходное множество на подмножества мощности k и $(n – k)$ и записывая суммы. Рассматривая все k, найдем минимальную разность. Этот алгоритм использует время, равное $O(2^n)$. Конечно, для этой задачи существуют более быстрые алгоритмы, однако никем

не найден полиномиальный алгоритм или алгоритм с оценкой времени выполнения, в которой *n* не появлялось бы в показателе степени.

✍ **Замечание 1.** Если x_1, x_2,…, x_n заданы, то это *индивидуальная задача. Решить задачу* означает решить все индивидуальные задачи. Для конкретной индивидуальной задачи могут существовать очень эффективные алгоритмы, но нас интересует, как алгоритм ведет себя на самых худших входных данных.

✍ **Замечание 2.** Имеются в виду «большие» задачи. В качестве параметра, отвечающего за размер входных данных, можно выбирать разные величины, однако изменение параметра не должно приводить к изменению класса сложности.

✍ **Замечание 3.** Существуют многочисленные способы кодирования данных для описания одной и той же задачи. Например, граф может быть задан матрицей инцидентности «узлы-дуги» или списком всех узлов с указанием соседних смежных узлов. Время работы алгоритма не зависит от способа кодирования, т. е. сложность алгоритма никогда одновременно не будет полиномиальной для одного разумного способа кодирования и экспоненциальной для другого разумного способа кодирования. В этом смысле все разумные способы кодирования дают один и тот же результат.

✍ **Замечание 4.** Разные компьютеры выполняют один и тот же алгоритм за разное время. Самый простой компьютер называется *машиной Тьюринга.* Это искусственный компьютер, модель абстрактной машины, которая может писать символы на ленту, читать, стирать их и заканчивать свою работу. Кроме этого, имеется несколько моделей машин с произвольным доступом к памяти. Эти модели являются аналогами самых производительных современных компьютеров. Оказывается, что если алгоритм использует экспоненциальное время на машине Тьюринга, то он требует экспоненциального времени на любом реальном компьютере и наоборот. Таким образом, можно предположить, что у нас есть компьютер с одним центральным процессором и неограниченной памятью.

Далее для характеристики сложности алгоритмов рассмотрим понятие *полиномиально ограниченной сложности алгоритма* (полиномиального алгоритма), для этого дадим определение полиномиальной функции.

🕮 **Определение**. Вещественная неотрицательная функция $f(m)$, определенная для целых положительных значений аргумента, называется *полиномиально ограниченной*, если существует полином $P(m)$ с вещественными коэффициентами такой, что $f(m) \le P(m)$ для всех $m \in \mathbf{N}^+$.

Классы задач в зависимости от их трудности

1. *Алгоритмически неразрешимые задачи* (нерешаемые задачи). Это

задачи, для решения которых не существует алгоритма. Например, доказано, что задача определения, остановится или нет заданная программа на данной машине Тьюринга при заданной исходной записи на ленте, является алгоритмически неразрешимой. В данном пособии такие задачи не рассматриваются.

2. *Труднорешаемые задачи* (вероятно, сложные задачи). Это задачи, для решения которых, по-видимому, не существует полиномиального алгоритма. Иными словами, для их решения, скорее всего, существуют только экспоненциальные алгоритмы.

3. *NP-задачи* (*NP* – аббревиатура для «*недетерминированные полиномиальные*»). Это класс задач, которые можно решить за полиномиальное время, если угадать, какой путь вычислений необходимо выполнить. Т.е. этот класс включает в себя все задачи, которые имеют экспоненциальный алгоритм, но не доказано, что они не могут иметь полиномиального алгоритма.

4. *P-задачи* (*P* – аббревиатура для «полиномиальный»). Этот класс включает в себя задачи, которые можно решить полиномиальными алгоритмами. Большинство специалистов считают, что этот класс является собственным подклассом класса 3.

2.3. Полиномиальные, недетерминированные алгоритмы

Все задачи, для которых есть полиномиальные алгоритмы, составляют класс *P*. Например, задачи, требующие $n^{116}+3n^4+7739$ единиц времени, принадлежат этому классу. Сама единица времени не фиксирована. Ею может быть, например, микросекунда или час.

Причины интереса к классу *P*.

1) Полиномиальная функция не чувствительна к особенностям реализации. Реализация алгоритма с использованием другой структуры данных может понизить степень многочлена, но не найдено примера двух реализаций одного алгоритма, одна из которых требует полиномиального, а другая – экспоненциального времени.

2) Полиномиальные алгоритмы быстрее экспоненциальных, когда задача приобретает большие размеры. Рассмотрим алгоритмы *A* и *B*. Алгоритм *A* требует n^5 операций, а алгоритм *B* требует 2^n операций. Если каждая операция выполняется за микросекунду, то алгоритм *A* при $n=10$ закончит свою работу за 0.1сек., а при $n=60$ – за 13 мин. С другой стороны, алгоритм *A* при $n=10$ закончит свою работу за 0.001сек., а при $n = 60$ – за 366 веков. По этой причине полиномиальные алгоритмы называют «хорошими». Даже если при малых значениях n полиномиальный алгоритм медленнее экспоненциального, при больших n он превосходит по скорости работы экспоненциальный. Заметим, что это справедливо только для худшего случая. Кроме того, например, для конкретного практического использования (т.е. есть ожидаемый диапазон размерности задачи) экспоненциальный алгоритм будет быстрее полиномиальных.

Можно не разбивать класс P на подклассы, такие, как n, $n\log n$, n^3 и т.д., и использовать асимптотическую оценку O, не беспокоясь о коэффициенте, стоящем перед функцией.

Рассмотрим задачу определения, есть ли в графе эйлеровы циклы. Граф имеет эйлеровы циклы тогда и только тогда, когда:

1) граф связен;

2) степень каждой вершины четна.

Не будем доказывать, что эти условия необходимы и достаточны, а только оценим объем работы, необходимой для проверки этих двух условий.

Предположим, что граф задан $n \times n$ матрицей, в которой элемент i-й строки j-го столбца равен 1, если i-я вершина смежна j-й вершине, и 0 – в противном случае. (Для наших целей зададим элементы на диагонали равными 0). Чтобы проверить, все ли вершины имеют четную степень, сложим элементы каждой строки и проверим, будут ли суммы четными. Так как в графе n вершин, то таких сумм будет n. Если просмотр каждого элемента выполняется за одну операцию, то всего требуется n^2 операций. Итак, для проверки четности вершин требуется полиномиальное время. Чтобы проверить, является ли граф связным, предположим, что граф задан *списком смежности,* т. е. списком вершин вместе с соседями, указанными для каждой вершины. По известной матрице смежности можно построить список смежности за время $O(n^2)$. Затем, используя поиск в глубину, проверить граф на связность. Так как каждая дуга будет пройдена, по крайней мере, два раза (ровно два раза, если дуга принадлежит остовному дереву, и один раз, если она не принадлежит ему), то алгоритм требует $O(m+n)$, где m – число дуг. Получаем полиномиальный алгоритм в терминах числа вершин n. Подчеркнем, что в данном случае алгоритм выдает ответ «да», если граф имеет эйлеров цикл, или «нет», если эйлерова цикла в графе нет.

Обычно задачу формулируют так, что возможными ответами являются «да» или «нет». Например, задачу коммивояжера нахождения кратчайшего цикла в графе можно переформулировать следующим образом.

Имеется ли в графе цикл, длина которого меньше заданного числа B? Если при любом B на этот вопрос можно ответить за полиномиальное время, то легко за полиномиальное время решить задачу коммивояжера. (Легко установить верхнюю и нижнюю границу для кратчайшего цикла, а затем использовать бинарный поиск, испытывая разные B).

Далее предполагается, что рассматриваются только задачи с ответами «да» или «нет». Такие задачи называются *задачами распознавания*, в то время как задачи максимизации или минимизации называются *задачами оптимизации*, для них термины соответственно *NP-полные* и *NP-трудные* задачи.

Рассмотрим алгоритм как компьютерную программу. При заданных входных данных программа выполняет различные вычисления. В зависимости от промежуточных результатов программа выполняет другие вычисления до

тех пор, пока не будет получен ответ. Для задач распознавания таким ответом является «да» или «нет».

Представим весь процесс вычислений в виде корневого дерева, каждая вершина представляет некоторое вычисление, а листьям приписаны возможные ответы «да» и «нет».

Для того чтобы алгоритм был полиномиальным, дерево должно иметь полиномиальную высоту, и, кроме этого, время, затрачиваемое на вычисления в каждом узле, также должно быть полиномиальным.

Когда входные данные алгоритма известны, компьютерная программа выполняет некоторые вычисления, по результату этих вычислений определяется дальнейший путь вычислений. Полиномиальные алгоритмы обладают тем свойством, что каждую траекторию вычислений можно пройти за полиномиальное время.

Последнее свойство полиномиальных алгоритмов весьма обременительно. Например, для задач теории графов некоторый алгоритм может использовать полиномиальное время, если граф планарный, и экспоненциальное время – в противном случае. Представляется весьма важным выделение полиномиальных подклассов в задачах, которые в общей постановке требуют экспоненциального времени. Кроме того, для задач из класса P построение более эффективных алгоритмов или новых модификаций старых алгоритмов может снизить время работы программы, например, с n^3 до n^2, n^2 до $n\log n$.

Недетерминированные алгоритмы

Рассмотрим понятие недетерминированного алгоритма. Для задачи коммивояжера детерминированный алгоритм, например, поиск с возвращением, определяет, по какой дуге следует направляться из текущего узла. Для задачи распознавания – существует ли в графе цикл, длины меньшей B, поиск с возвращением неявно перебирает все циклы и дает ответ «да», если такой цикл существует, и «нет» – в противном случае.

Так как циклы перебираются один за одним, и циклом с необходимым свойством может оказаться последний, то поиск с возвращением для задачи коммивояжера требует время $O(c^n)$.

В ходе поиска решения с помощью недетерминированного алгоритма можно *правильно угадать*, какая дуга из текущей вершины должна быть следующей для построения минимального маршрута коммивояжера. Итак, если существует цикл общей длины, меньшей B, то недетерминированный алгоритм тратит время $O(n)$ на вычисление суммарной длины и проверки того, меньше ли она чем B. Таким образом, если ответ «да», то задача коммивояжера с помощью недетерминированного алгоритма может быть решена за полиномиальное время.

Грубо говоря, если ответ «да» можно проверить за полиномиальное время, то задачу можно решить недетерминированным алгоритмом за полиномиальное время, и задача принадлежит классу NP.

С другой стороны, если цикла длины, меньшей B, в графе не существует, то для ответа «нет», по-видимому, не достаточно уметь правильно угадывать нужную дугу. Таким образом, задача с ответом «нет», вероятно, не принадлежит классу *NP*.

Если некоторая задача принадлежит классу P, то дополнение к ней также принадлежит P. Скорее всего, аналогичное утверждение не верно для класса *NP* (хотя это и не доказано).

По определению, если задача принадлежит P, то она принадлежит *NP*. Итак, класс P является подклассом класса *NP*, т.е. $P \subseteq NP$. Основной вопрос современной теории сложности – это $P = NP$?

Для доказательства $P=NP$ достаточно показать, что для решения всякой задачи из *NP* существует полиномиальный детерминированный алгоритм. Никто еще не сделал этого. Для доказательства $P \neq NP$ достаточно показать, что в *NP* есть задача, которую детерминированным алгоритмом нельзя решить за полиномиальное время. Никто еще не сделал и этого.

В настоящее время большинство исследователей полагает, что $P \neq NP$. Для разрешения этого вопроса, скорее всего, необходим новый математический метод.

2.4. *NP*-полные задачи

Задача о гамильтоновом цикле заключается в определении, есть ли в графе цикл, проходящий через каждую вершину ровно один раз. В классической задаче коммивояжера надо найти самый короткий цикл. Можно поставить вопрос следующим образом: существует ли в графе достаточно короткий цикл, проходящий через каждую вершину? Алгоритм для задачи коммивояжера можно использовать для решения задачи о гамильтоновом цикле, так как задачу коммивояжера можно *свести к* первой следующим образом.

Пусть каждая дуга графа имеет длину 1, а расстояние между двумя несмежными вершинами равно ∞. Так как в графе n вершин, то существование цикла длины n эквивалентно существованию гамильтонова цикла.

Заметим, что данное сведение требует полиномиального времени, а именно, на определение расстояний между $n(n-1)/2$ парами вершин. Если удастся построить полиномиальный алгоритм для решения задачи коммивояжера, то тем самым можно найти полиномиальный алгоритм для задачи о гамильтоновом цикле.

Будем говорить, что задача L_1 *сводится* к задаче L_2, и записывать $L_1 \prec L_2$, если любой алгоритм для L_2 можно использовать для решения задачи L_1. Подразумевается, что алгоритм сведения имеет полиномиальную сложность. Если L_2 также можно свести к L_1; то говорят, что задачи L_1 и L_2 *полиноминально эквивалентны*, и пишут $L_1 \cong L_2$.

Оба отношения $\prec$ и $\cong$ транзитивны (т. е. если, например, $L_1 \cong L_2$ и $L_2 \cong L_3$, то $L_1 \cong L_3$).

Оказывается, что в *NP* существует подкласс в некотором смысле самых сложных задач (все задачи в этом подмножестве полиномиально эквивалентны), а именно: каждую задачу из *NP* можно свести к любой задаче из этого подкласса. Следовательно, если для одной из задач этого подмножества найден полиномиальный алгоритм, то любую задачу из класса *NP* можно решить за полиномиальное время.

Задачи из этого подмножества называются *NP*-полными. Итак, класс *NP* содержит два интересных подкласса: подкласс задач, которые могут быть решены за полиномиальное время детерминированными алгоритмами и наиболее сложных в классе *NP*, это символически изображено на рис. 2.1.

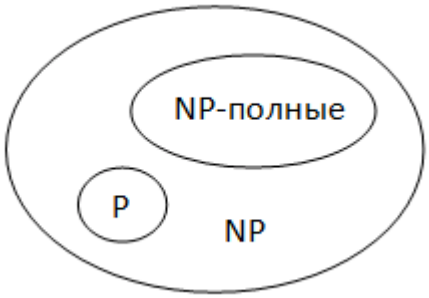


Рис. 2.1. Схематическое представление класса *NP*-задач

Если найти новую *NP*-полную задачу *L*, то есть небольшой шанс изобрести для нее полиномиальный алгоритм. Чтобы доказать, что задача *L* является *NP*-полной, достаточно доказать следующие два утверждения:

(1) Задача *L* принадлежит классу *NP*.

(2) Известная *NP*-полная задача сводится к *L*.

Рассмотрим три стандартные *NP*-полные задачи и их варианты.

1. Задача о разбиении. Существует ли в заданном числовом множестве x_1, x_2,.., x_n подмножество, такое, что сумма чисел в подмножестве равна *B*? Более общий вариант этой задачи – задача о рюкзаке.

2. Гамильтоновы циклы. Более общий вариант этой задачи – задача коммивояжера.

3а. Задача о минимальном вершинном покрытии. Во множестве вершин *V* заданного графа $G = (V, E)$ найти подмножество $V' \subseteq V$ минимальной мощности, такое, что каждое ребро графа инцидентно, по крайней мере, одной вершине из *V*.

Существует другой, по существу, эквивалентный, вариант задачи 3а:

3б. Задача о максимальном независимом множестве. Во множестве вершин *V* заданного графа $G = (V, E)$ найти подмножество $V'' \subseteq V$ максимальной мощности, такое, что никакое ребро графа не инцидентно двум вершинам из V''. Можно проверить, что $V'' = V - V'$.

Алгоритмы, полиномиальные при ограниченной величине коэффициентов (входной информации, например, для задач о загрузке это размер рюкзака и количества предметов), называются *псевдополиномиальными*.

Многие *NP*-полные задачи имеют псевдополиномиальные алгоритмы, которые достаточно эффективны на практике.

Для задачи о минимальном вершинном покрытии нет естественной границы на величину каких-либо коэфициентов, и число n является хорошим параметром, характеризующим длину входа. Поэтому псевдополиномиального алгоритма, решающего данную задачу, не существует.

Решение *NP*-полной задачи

Для решения новой задачи всегда хочется придумать точный эффективный алгоритм. В литературе приводятся списки известных *NP*-полных задач и может так оказаться, что наша задача есть в этом списке. Что делать в этом случае? Или нашей задачи нет в списке, но думается, что она могла бы там быть.

Доказательство того, что наша задача *NP*-полна, не решает проблемы. С практической точки зрения необходимо изобрести новый алгоритм или использовать старый, даже если задача *NP*-полна. Если алгоритм полиномиален, но степень полинома больше 3, то этот алгоритм, скорее всего, не достаточно эффективен. Дадим несколько советов.

1. *Придумать новый алгоритм*. Один из лучших примеров такого подхода симплекс-метод линейного программирования. Хотя симплекс-метод не является полиномиальным в худшем случае, он работает. Одной из главных проблем линейного программирования является вопрос: «Почему симплекс-метод работает так хорошо?». В формулировке задачи линейного программирования значения элементов матрицы A и векторов b и c произвольны. Вероятно, в реальных задачах A, b, c удовлетворяют некоторым условиям, делающим симплекс-метод эффективным.

2. *Рассмотреть специальные случаи задачи*. Для любой нерешенной задачи полезно сначала рассмотреть специальные или крайние случаи, а затем постепенно обобщать результат. Обычно новую задачу сводят к известной, пытаясь, тем не менее, не потерять специфики задачи. Великое множество комбинаторных задач можно свести к задачам целочисленного линейного программирования и решать общими методами. Однако для реальных задач эти общие методы, как правило, приводят к весьма медленным на практике алгоритмам. Общий алгоритм подобен одежде ходового размера: она подойдет всем, но не будет слишком удобна.

3. *Эвристический подход*. Чтобы сформулировать эвристический подход для решения конкретной задачи, обычно испытывают несколько числовых примеров, на которых тренируют интуицию. Если эвристический алгоритм приводит к успеху только в некоторых случаях, необходимо описать эти случаи.

4. *Декомпозиция задачи*. Если задача большая и сложная, то ее, возможно, удастся разложить на несколько подзадач и решать отдельно каждую. После этого необходимо как-то из частных решений составить общее решение

исходной задачи. Подход декомпозиции проиллюстрирован на задаче нахождения кратчайших путей между всеми парами вершин большой сети (глава 3 п. 3.2).

5. *Использовать общие методы*. Если нет времени и желания заниматься настоящими исследованиями, можно использовать общие подходы. Вот некоторые из них:

(1) поиск с возвращением;

(2) динамическое программирование;

(3) методы отсечений Гомори.

Специальные модификации даже таких методов, как поиск с возвращением, могут существенно повысить эффективность алгоритма, например, в игре на дереве. Метод отсечений Гомори предназначен для решения задач целочисленного линейного программирования. В некоторых случаях он хорошо зарекомендовал себя, но при решении многих других задач ведет себя крайне плохо.

Вообще говоря, нет систематического метода для создания новых комбинаторных алгоритмов.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что понимается под сложностью алгоритма?
2. Что означает функция *O*?
3. Верно ли, что сложность алгоритма определяется для каждого конкретного набора исходных данных?
4. Какие классы сложности алгоритмов вы знаете?
5. Что понимается под выражением «эффективный алгоритм»?
6. Верно ли, что при малых размерностях входных данных задачи полиномиальный алгоритм работает быстрее, чем экспоненциальный?
7. Верно ли, что алгоритм, реализованный на самом производительном современном компьютере, имеет меньшую вычислительную сложность, чем тот же алгоритм, реализованный на компьютере предыдущего поколения?
8. Для чего нужна классификация задач по сложности?
9. Какие вы знаете классы задач в зависимости от их трудности?
10. Что означает аббревиатура *NP*?
11. Приведите пример *NP*-полной задачи.
12. Верно ли, что для некоторых *NP*-трудных задач известны полиномиальные алгоритмы точного решения?
13. Верно ли, что понятия *NP*-полный и *NP*-трудный эквивалентны?
14. Верно ли, что для доказательства того, что *P=NP*, достаточно показать, что хотя бы одна задача из *NP* принадлежит *P*?
15. Существуют ли в *NP* задачи, не являющиеся *NP*-полными?
16. Что означает выражение «псевдополиномиальный алгоритм»?

3. КРАТЧАЙШИЕ ПУТИ В ГРАФАХ

Теория графов рассматривает множества с заданными на них отношениями между элементами. С помощью данного математического аппарата решаются различные задачи, в их числе задачи из области администрирования сетей, информационных потоков, планирования, проектирования и управления различными системами.

🕮 **Определение.** *Графом* $\Gamma=(V,E)$ называется пара множеств: V – множество, элементы которого называются *вершинами*, E – множество пар вершин, называемых *ребрами*.

Если вершины $v,\ w \in V$, $e=\{v,w\}\in E$, то говорят, что ребро e соединяет вершины v и w или e *инцидентно* v и w. Таким образом, $\{v,w\}$ – обозначение ребра.

Если E представляет собой упорядоченные пары (т. е. E – подмножество декартова произведения $V\times V$), то граф называется *ориентированным* (*орграфом*), а пары (v,w) называют *дугами*. Если множеству E принадлежат пары $v=w$, то такие ребра $\{v,v\}$ называют *петлями*.

🕮 **Определение.** *Подграфом* графа G (ориентированного графа D) называется граф, все вершины и ребра которого содержатся среди вершин и ребер графа G (D).

Подграф называется *собственным*, если он отличен от самого графа.

🕮 **Определение.** *Степенью* вершины v графа G называется число $\delta(v)$ ребер графа G, инцидентных вершине v.

🕮 **Определение.** Последовательность $v_1e_1v_2e_2v_3\ldots e_kv_{k+1}$, (где $k\geq 1$, $v_i\in V$, $i=1,\ldots,k+1$, $e_j\in E$, $j=1,\ldots,k$), в которой чередуются вершины и ребра (дуги) и для каждого $j=1,\ldots,k$ ребро (дуга) e_j имеет вид $\{v_j,v_{j+1}\}$ (для ориентированного графа (v_j,v_{j+1})), называется *маршрутом*, соединяющим вершины v_1 и v_{k+1} (*путем* из v_1 в v_{k+1}).

🕮 **Определение.** Граф (ориентированный граф) называется *связным* (*сильно связным*), если для любых двух его вершин v, w существует маршрут (путь), соединяющий v и w.

Маршрут (путь) называется *замкнутым*, если начальная вершина совпадает с конечной $v_1=v_{k+1}$.

🕮 **Определение.** *Цепь* − незамкнутый маршрут (путь), в котором все ребра (дуги) различны, т. е. не повторяются.

🕮 **Определение.** *Цикл* (*контур*) − замкнутый маршрут (путь), в котором все ребра (дуги) различны, т. е. не повторяются.

🕮 **Определение.** *Простая цепь* – цепь, в которой все вершины различны.

🕮 **Определение.** *Простой цикл* (*контур*) − цикл (контур), в котором все вершины (кроме первой и последней) различны, т. е. не повторяются.

🕮 **Определение.** Граф G называется *деревом*, если он является связным и не имеет циклов.

Подграф, являющийся деревом и содержащий все вершины графа, называется *остовным* (*покрывающим*) *деревом* графа.

Эти определения иллюстрируются рис. 3.1.

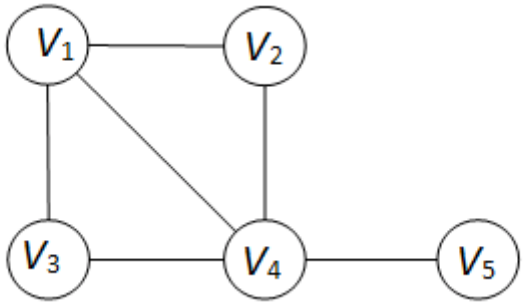


Рис. 3.1. Граф G

В графе G имеется три пути между вершинами V_1 и V_5: (V_1,V_2,V_4,V_5), (V_1,V_4,V_5),…,(V_1,V_3,V_4,V_5). Ребра $e_{14},e_{24},e_{34},e_{45}$ образуют остовное дерево, то же верно для ребер $e_{12},e_{24},e_{34},e_{45}$. Еще одно остовное дерево можно составить из ребер $e_{12},e_{13},e_{34},e_{45}$. Вершина V_1 имеет степень 3 в графе G и степень 2 в последнем остовном дереве. Если ребро e_{45} ориентировано от V_4 к V_5, то по-прежнему есть три пути из V_1 в V_5, но ни одного пути из V_5 в V_1.

В большинстве приложений с ребрами или вершинами ассоциируются некоторые числа. В этом случае граф называется *сетью*. Все определения теории графов применимы и к сетям. В теории сетей обычно используются термины «*узлы*» и «*дуги*» вместо «*вершины*» и «*ребра*».

3.1. Кратчайшие пути. Алгоритм Дейкстры

Кратчайший путь

🕮 **Определение.** Число, сопоставляемое каждой дуге сети, называют *длиной* дуги.

В большинстве случаев длины дуг положительны, но в некоторых приложениях они могут быть и отрицательными. Например, узлы могут представлять различные состояния некоторой физической системы, а длина дуги e_{ij} может означать количество энергии, поглощаемой при переходе из состояния V_i в состояние V_j. Отрицательная длина дуги тогда означает, что энергия излучается при переходе из V_i в V_j. Если суммарная длина некоторого контура или цикла в сети отрицательна, будем говорить, что сеть содержит отрицательный контур.

Длина пути есть сумма длин всех его дуг. Обычно имеется много путей между двумя узлами V_s и V_t.

🕮 **Определение.** Путь минимальной длины называется *кратчайшим путем* из V_s в V_t.

Задача нахождения кратчайшего пути является фундаментальной и часто входит как подзадача в другие оптимизационные задачи.

Обычно рассматриваются три типа задач о кратчайшем пути:

(1) кратчайший путь от одного узла до другого;

(2) кратчайшие пути от одного узла до всех других узлов;

(3) кратчайшие пути между всеми парами узлов.

Так как все алгоритмы, решающие задачу (1) и задачу (2), по существу, те же самые, рассмотрим задачу нахождения кратчайших путей от одного узла до всех остальных узлов сети.

Задача нахождения кратчайшего пути корректна, если сеть не содержит отрицательного контура.

✍ **Замечание.** Сеть может иметь ориентированные дуги отрицательной длины и не иметь отрицательных контуров.

Обозначим длину дуги из V_i в V_j через d_{ij} и предположим, что

$$d_{ij} > 0 \text{ для всех } i, j, \tag{3.1}$$

$$d_{ij} \neq d_{ji} \text{ для некоторых } i, j, \tag{3.2}$$

$$d_{ij} + d_{jk} \leq d_{ik} \text{ для некоторых } i, j, k. \tag{3.3}$$

Для удобства будем считать, что $d_{ij} = \infty$, если нет дуги из узла V_i в узел V_j, и $d_{ii} = 0$ для всех i.

Условие (3.3) делает задачу о кратчайшем пути нетривиальной. Если оно не выполняется, то кратчайший путь из V_i в V_j состоит из единственной дуги e_{ij}.

Обычно необходимо знать как длину кратчайшего пути, так и последовательность его узлов. Сделаем сначала несколько замечаний. Пусть P_k – путь из V_0 в V_k, V_i – промежуточный узел этого пути. Тогда подпуть от V_0 до V_i содержит меньше дуг, чем путь P_k. Так как длины всех дуг положительны, то этот подпуть короче, чем P_k.

✍ **Замечание 1**. Длина пути больше, чем длина любого его подпути (справедливо, только если длины всех дуг положительны).

Пусть V_i – промежуточный узел пути P_k (от V_0 до V_k). Если P_k – кратчайший путь, то подпуть от V_0 до V_i сам должен быть кратчайшим путем. В противном случае более короткий путь до V_i, дополненный отрезком исходного пути от V_i до V_k, составил бы путь, более короткий, чем P_k.

✍ **Замечание 2**. Любой подпуть кратчайшего пути сам должен быть кратчайшим путем (что не зависит от того, положительны ли длины дуг).

✍ **Замечание 3**. Любой кратчайший путь содержит не более чем *n*–1 дуг (при условии, что нет отрицательных циклов и что в сети *n* узлов).

На основании этих замечаний можно построить алгоритм для нахождения кратчайших путей из V_0 во все остальные узлы сети.

Пусть кратчайшие пути из V_0 во все остальные узлы упорядочены в соответствии с их длинами. Для простоты изложения можно переименовать узлы так, чтобы кратчайший путь в V_1 был кратчайшим среди всех кратчайших путей. Пусть пути занумерованы в порядке возрастания их длин:

$P_1 \le P_2 \le P_3 \le \ldots \le P_{n-1}$.

Алгоритм позволит найти сначала P_1, затем P_2, и т.д., пока не будет найден самый длинный из кратчайших путей.

Идея алгоритма

– Если P_1 содержал бы более одной дуги, то он исключал бы более короткий подпуть (замечание 1). Поэтому P_1 должен содержать только одну дугу.

– Если P_k содержит более чем k дуг, то он содержит, по крайней мере, k промежуточных узлов. Каждый из подпутей, ведущих в промежуточный узел, короче, чем P_k, и получается k путей, более коротких, чем P_k, что невозможно.

✍ **Замечание 4**. Кратчайший путь P_k содержит не более k дуг.

Следовательно, чтобы найти P_1, нужно только рассмотреть пути из одной дуги, минимальным среди них будет P_1. Чтобы найти P_2, нужно рассмотреть пути из одной и двух дуг. Минимальным среди них будет P_2. Если P_2 – путь из двух дуг с последней дугой e_{j2} и при этом $j \neq 1$, то дуга e_{0j} образует подпуть P_2, более короткий, чем P_2. Поэтому путь P_2 должен либо состоять из одной дуги, либо из двух дуг, последней из которых является дуга e_{12}.

Далее будем приписывать узлам числа, называемые метками. Каждая метка может быть одного из двух видов: временная или постоянная.

🕮 **Определение.** *Постоянная метка* узла – это истинное кратчайшее расстояние от начала V_0 до этого узла.

🕮 **Определение.** *Временная метка* – это длина некоторого пути от начала до этого узла. Этот путь может и не быть кратчайшим, поэтому временная метка является верхней оценкой истинного кратчайшего расстояния.

Поиск кратчайших путей от одного узла до всех остальных

1) Начиная поиск P_1, приписываем каждому узлу V_i длину дуги d_{0i}, т. е. временные метки узлов. Среди всех временных меток выбираем минимальную и превращаем ее в постоянную (V_1).

2) Чтобы найти P_2, не нужно искать все пути из двух дуг, достаточно рассмотреть те из них, у которых первая дуга есть e_{01}. В качестве временных меток приписываем узлам длины путей из одной дуги.

3) Сравниваем d_{0i} (длину одной дуги) с $d_{0i} + d_{1i}$ (длиной пути из двух дуг) и минимальное из этих двух значений приписываем как временную метку узлу V_i.

4) Минимальная среди всех временных меток есть P_2.

Постоянная метка указывает истинное кратчайшее расстояние от V_0 до V_i. Временная метка узла V_j указывает либо длину дуги e_{0j}, либо длину пути из V_0 в постоянный узел V_i, дополненного длиной дуги e_{ij}.

Правило для упрощения поиска

Как только узел V_i получает постоянную метку l_i^*, проверяем для каждого узла V_j, соседнего с узлом V_i и имеющего временную метку, верно ли, что $l_i^* + d_{ij}$ меньше, чем текущая временная метка V_j. Если верно, заменим эту временную метку значением $l_i^* + d_{ij}$. Если нет, оставим временную метку без изменения.

Чтобы найти P_{k+1}, достаточно найти минимальную временную метку всех соседей узлов и превратить эту метку в постоянную.

Теперь можно формально описать алгоритм и применить его к численному примеру. Будем применять l_i для обозначения временного кратчайшего расстояния и l_i^* для обозначения истинного кратчайшего расстояния.

Алгоритм Дейкстры
(нахождение кратчайшего пути от одного узла до всех остальных)

Шаг 0. Каждому узлу V_i (i = 1,2,..., n–1) присвоить временную метку $l_i = d_{0i}$ (если нет дуги, соединяющей V_0 и V_i, полагаем $d_{0i} = \infty$).

Шаг 1. Среди всех временных меток выбрать $l_k = \min\limits_i l_i$. Заменить l_k на l_k^*. Если нет временных меток, остановиться.

Шаг 2. Пусть V_k – узел, только что получивший постоянную метку на шаге 1. Изменить временные метки соседей V_i узла V_k в соответствии со следующим правилом: $l_i \leftarrow \min\left\{l_i, l_k^* + d_{ki}\right\}$. Перейти к шагу 1.

❖***Пример 3.1.*** Рассмотрим сеть, показанную на рис. 3.2, где числа являются длинами дуг.

Будем вписывать временные метки внутри каждого узла, а когда метка превращается в постоянную, будем помечать число звездочкой. Когда дуга применяется в некотором кратчайшем пути, будем изображать ее жирной линией.

Шаг 0. Все узлы получают временные метки, равные d_{0i}, а узел V_0 – постоянную метку 0. Это показано на рис. 3.3.

Шаг 1. Среди всех временных меток минимальное значение 2 имеет метка узла V_3, поэтому V_3 получает постоянную метку.

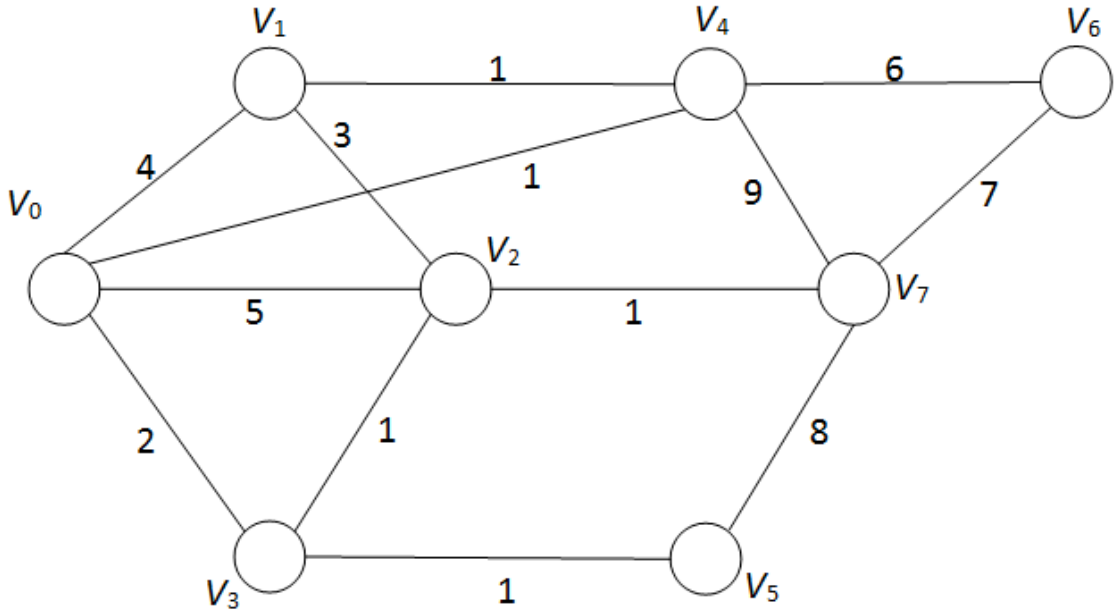


Рис. 3.2. Пример сети

Шаг 2. Узел V_3 имеет соседей V_2 и V_5.

$$l_2 \leftarrow \min\left\{l_2, l_3^* + d_{32}\right\} = \min\{5, 2+1\} = 3.$$

$$l_5 \leftarrow \min\left\{l_5, l_3^* + d_{35}\right\} = \min\{\infty, 2+11\} = 13.$$

Результат показан на рис. 3.4.

Шаг 1. Среди всех временных меток наименьшую метку 3 имеет узел V_2. Поэтому V_2 получает постоянную метку.

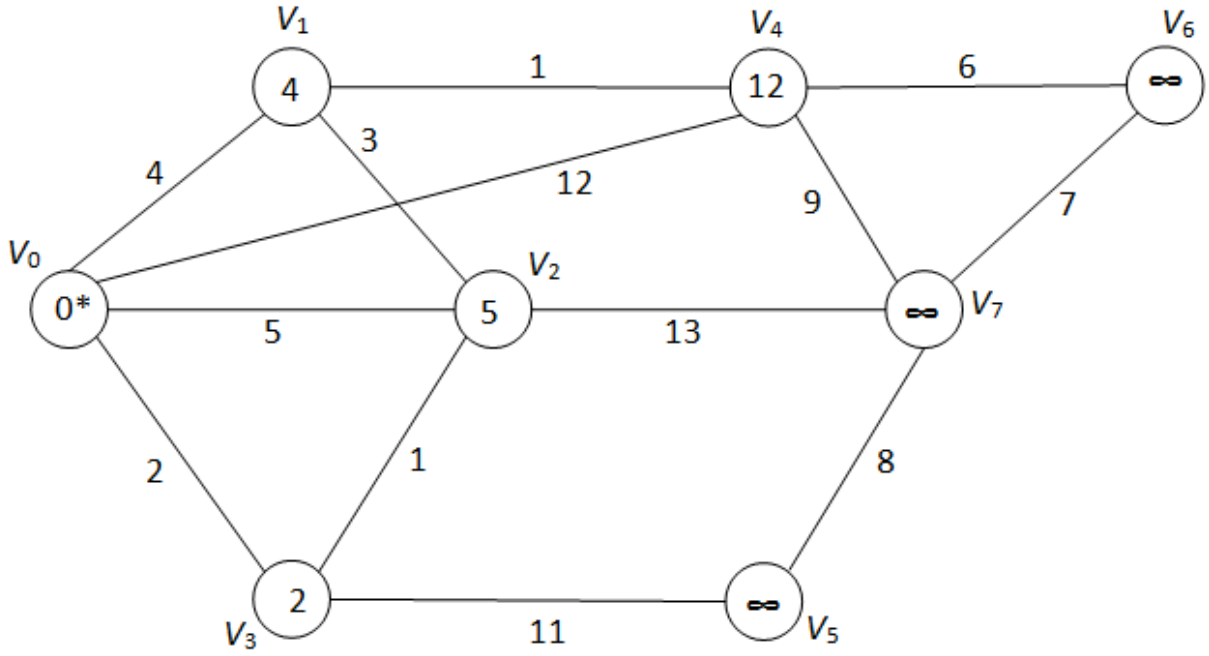


Рис. 3.3. Пример 3.1. Шаг 0

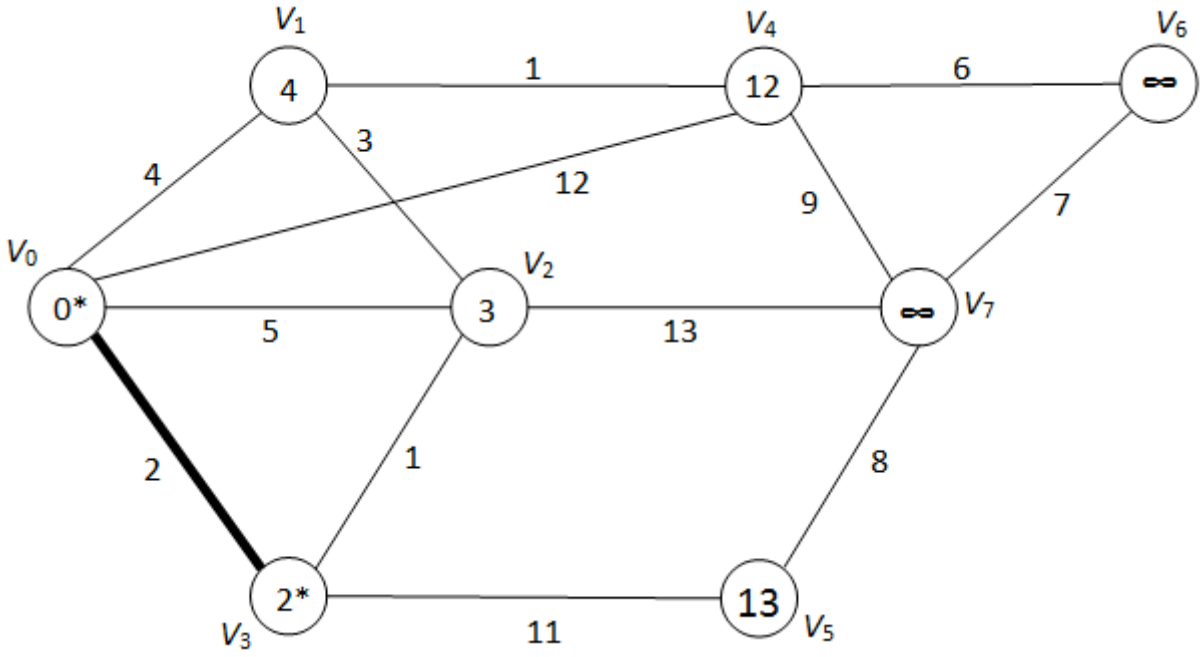


Рис. 3.4. Пример 3.1. Шаг 2

Шаг 2. Соседями узла V_2 являются V_1, V_7 (узел V_3 тоже соседний, но он стал постоянным и поэтому исключается).

$$l_1 \leftarrow \min\left\{l_1, l_2^* + d_{21}\right\} = \min\{4, 3+3\} = 4,$$

$$l_7 \leftarrow \min\left\{l_7, l_2^* + d_{27}\right\} = \min\{\infty, 3+13\} = 16.$$

Шаг 1. Узел V_1 получает постоянную метку 4.

Шаг 2. $l_4 \leftarrow \min\left\{l_4, l_1^* + d_{14}\right\} = \min\{12, 4+1\} = 5$ (рис. 3.5).

Шаг 1. V_4 получает постоянную метку 5.

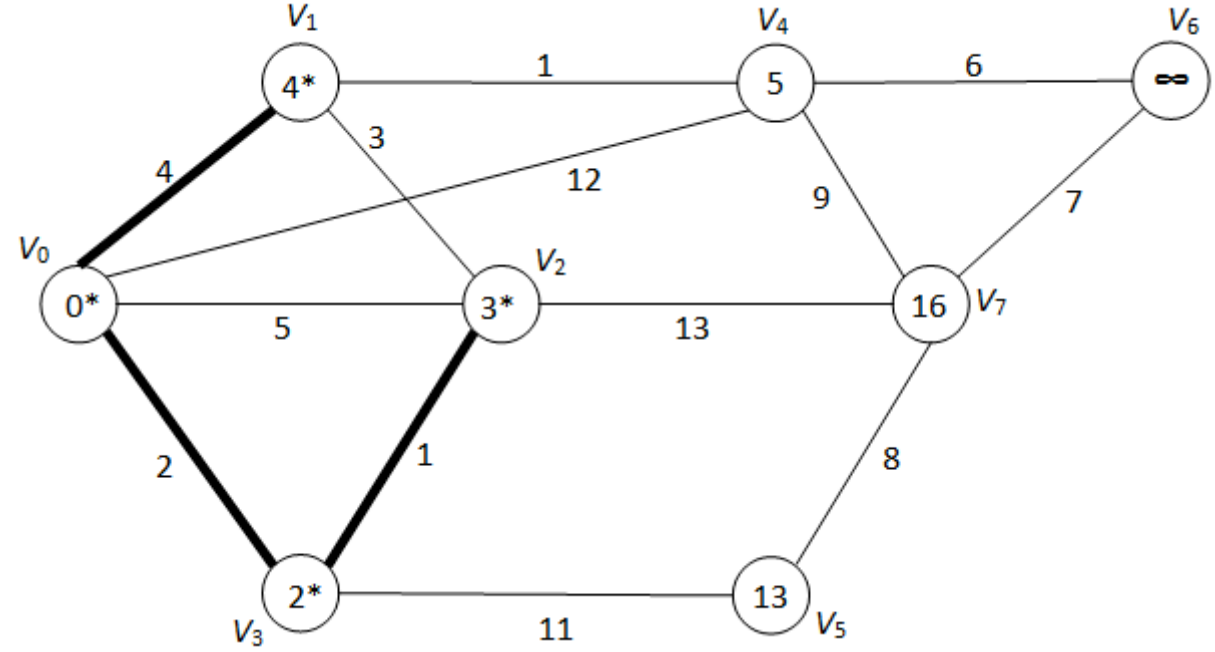


Рис. 3.5. Пример 3.1, шаг 2

Шаг 2. $l_6 \leftarrow \min\{l_6, l_4^* + d_{46}\} = \min\{\infty, 5+6\} = 11.$

$l_7 \leftarrow \min\{l_7, l_4^* + d_{47}\} = \min\{16, 5+9\} = 14.$

Шаг 1. V_6 получает постоянную метку 11.

Шаг 2. $l_7 \leftarrow \min\{l_7, l_6^* + d_{67}\} = \min\{14, 11+7\} = 14.$

Шаг 1. V_5 получает постоянную метку 13.

Шаг 2. $l_7 \leftarrow \min\{l_7, l_5^* + d_{57}\} = \min\{14, 13+8\} = 14.$

Шаг 1. V_7 получает постоянную метку 14.

Окончательный результат показан на рис. 3.6.

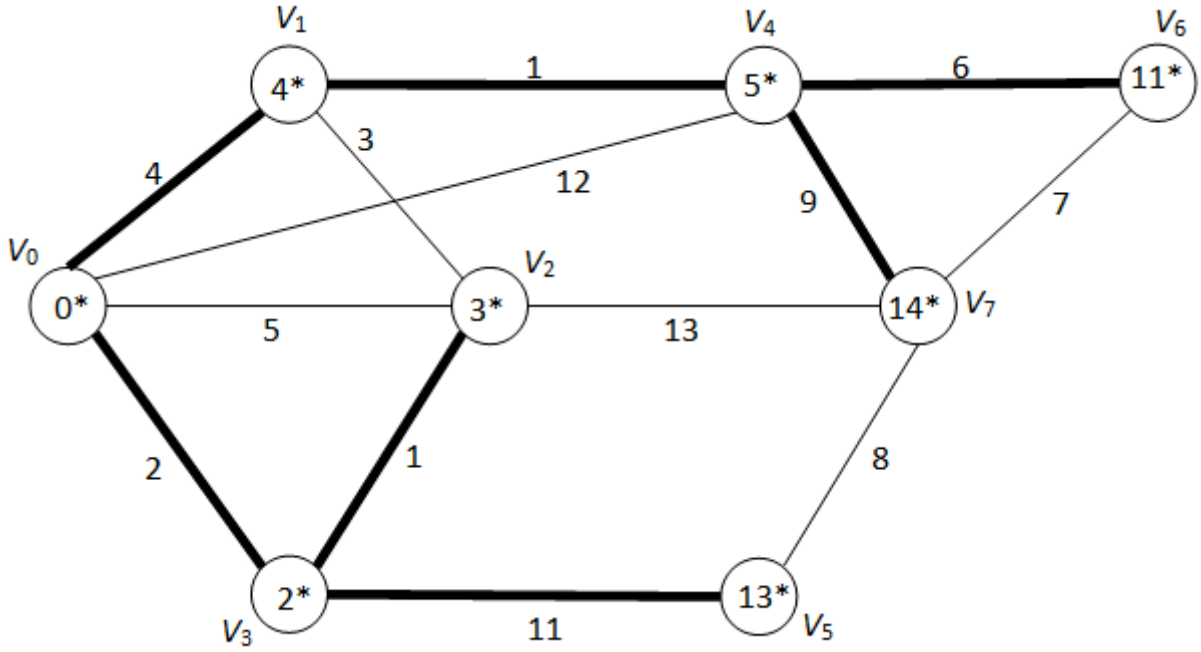


Рис. 3.6. Пример 3.1. Кратчайшие пути

Таким образом вычислены кратчайшие расстояния от V_0 до всех остальных узлов сети, но не найдены кратчайшие пути, на которых достигаются эти расстояния.

Способ проследить промежуточные узлы

Если узлу V_j приписана постоянная метка, то можно просмотреть все соседние узлы и найти среди них тот, метка которого отличается от метки узла V_i *в* точности на длину соединяющей их дуги. Таким образом, можно для каждого узла проследить в обратном направлении путь из начала в этот узел. На рис. 3.7 каждому узлу приписаны два числа. Первое число – постоянная метка, указывающая истинное кратчайшее расстояние от начала до этого узла, второе число указывает последний промежуточный узел кратчайшего пути.

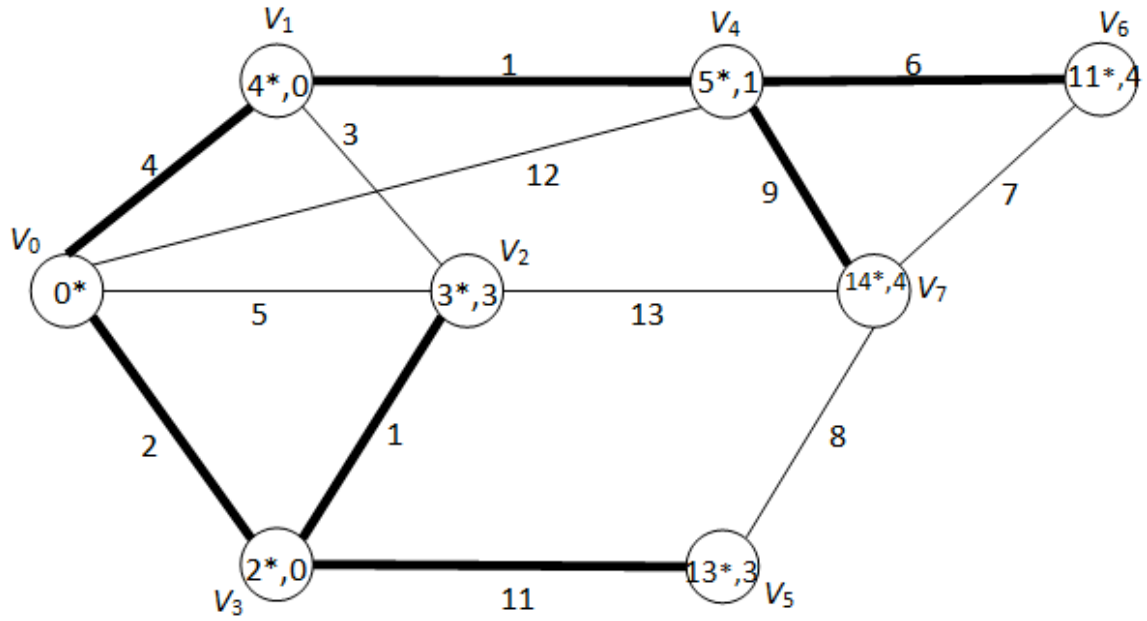


Рис. 3.7. Пример 3.1. Кратчайшие пути и промежуточные узлы

Сложность алгоритма. Так как алгоритм состоит из сравнений и сложений, подсчитаем количество этих операций.

Имеется n–2 сравнений при первом проходе, n–3 сравнений при втором проходе и т. д., таким образом, всего имеется

$(n–2)+(n–3)+ \ldots +1 = (n–1)(n–2)/2$

сравнений на шаге 1. Аналогично, имеется $(n–1)(n–2)/2$ сложений и столько же сравнений на шаге 2. Поэтому трудоемкость алгоритма есть $O(n^2)$. Так как в сети с n узлами имеется $O(n^2)$ дуг, и каждая дуга должна быть рассмотрена хотя бы один раз, то можно сказать, что не существует алгоритма, требующего в общем случае $O(n \log n)$ шагов.

3.2. Кратчайшие пути между всеми парами узлов. Алгоритм Флойда-Уоршелла

Рассмотрим задачу поиска кратчайших путей между всеми парами узлов сети.

Пусть $e_{ij}, e_{ik}, e_{kl}, \ldots, e_{pq}$ – кратчайший путь из V_i в V_q. Тогда кратчайший путь из V_i в V_j должен представлять собой единственную дугу e_{ij}, кратчайший путь из V_j в V_k – дугу e_{jk}, и т.д.

Назовем дугу e_{ij} *базисной*, если она представляет собой кратчайший путь из V_i в V_j. Из данного определения следует, что кратчайший путь состоит только из базисных дуг.

Алгоритм, который описывается ниже, заменяет все небазисные дуги базисными. Т. е. алгоритм строит дуги, соединяющие каждую пару узлов, не соединенную базисной дугой. Длина каждой построенной дуги равна кратчайшему расстоянию между двумя узлами. Для данного узла V_j рассмотрим следующую простую операцию:

$$d_{ik} \leftarrow \min\left\{d_{ik}, d_{ij} + d_{jk}\right\}, \qquad (3.4)$$

где d – длина дуги.

Операция выполняется для каждого *фиксированного* j и всевозможных i и k, не равных j. Для трех узлов V_i, V_j и V_k и трех дуг с длинами d_{ik}, d_{ij} и d_{jk} данная операция сравнивает длину дуги e_{ik} с длиной пути, состоящего из двух дуг с промежуточным узлом V_j. Этот случай показан на рис. 3.8. Операция (3.4) называется *тройственной операцией*.

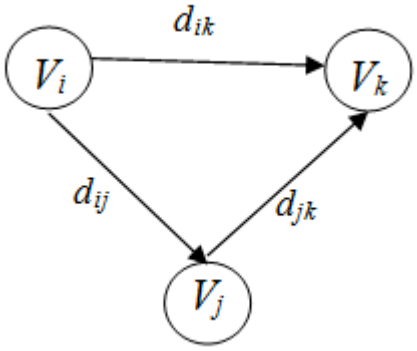


Рис. 3.8. Иллюстрация тройственной операции

Для всевозможных пар узлов V_i и V_k, смежных с V_j, выполняются следующие операции:

– если $d_{ik} \le d_{ij} + d_{jk}$, то никаких действий не производим;

– если $d_{ik} > d_{ij} + d_{jk}$, то создаем новую дугу, ведущую из V_i в V_k с $d_{ik} = d_{ij} + d_{jk}$.

Схема алгоритма

1) Фиксируем j=1 и выполняем тройственную операцию (3.4) для всех i, k = 2, 3,..., n.

2) Фиксируем j=2 и выполняем тройственную операцию (3.4) для всех i, k = 1, 3,..., n.

✍ **Замечание.** Все новые дуги, добавленные при j = 1, используются далее при j = 2 и т.д.

Как только все тройственные операции будут выполнены для $j = n$, сеть будет состоять только из базисных дуг, и числа, приписанные каждой ориентированной дуге, ведущей из V_p в V_q, будут кратчайшими расстояниями из V_p в V_q. Докажем это утверждение на примере.

В исходной сети рассмотрим любой кратчайший путь из V_p в V_q. Кратчайший путь должен состоять из базисных дуг этой сети. Если в результате тройственной операции создается дуга с длиной, равной сумме длин всех базисных дуг кратчайшего пути, то это докажет корректность алгоритма.

Рассмотрим кратчайший путь, изображенный на рис. 3.9.

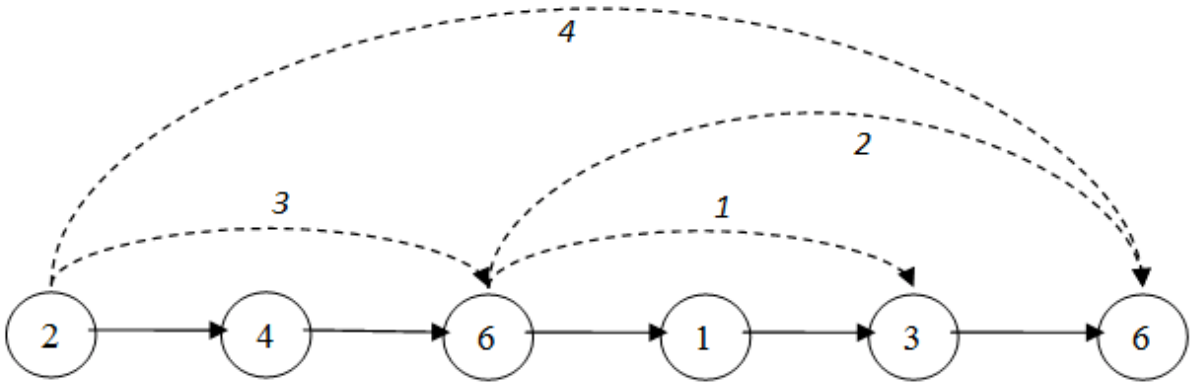


Рис. 3.9. Построение дуг в кратчайшем пути

При j=1 тройственная операция создает новую дугу с $d_{63}=d_{61}+d_{13}$.

При j=2 тройственная операция никак не скажется на данном отдельном кратчайшем пути.

При j=3 дуга с $d_{63}=d_{61}+d_{13}$ уже построена, следовательно, тройственная операция построит дугу с $d_{63}=d_{63}+d_{39}=(d_{61}+d_{13})+d_{39}$.

При j =4 будет построена дуга с $d_{26} = d_{24} + d_{46}$.

При j=6 будет построена дуга длины $d_{29}=d_{26}+d_{69}=(d_{24}+d_{46})+(d_{63}+d_{39})=$ $= d_{24}+d_{46}+d_{61}+d_{13}+d_{39}$.

На рис. 3.9 построенные дуги изображены пунктирной линией. Числа рядом указывают порядок, в котором эти дуги появлялись. Таким образом, базисная дуга e_{63} построена первой, а базисная дуга e_{69} построена второй. Некоторые базисные дуги, которые также будут построены в результате тройственной операции, например, e_{41}, не изображены на рисунке, так как они не влияют на рассматриваемый кратчайший путь.

✍ **Замечание 1.** Любая из дуг, построенная в результате тройственной операции, не может быть заменена другой дугой или путем меньшей длины. Противное противоречило бы тому, что исходный путь – кратчайший.

✍ **Замечание 2.** Если в сети отсутствуют отрицательные циклы, то любой кратчайший путь должен быть простым и должен состоять не более чем из n–1 дуг и не более чем из n–2 различных промежуточных узлов.

Приведенный алгоритм легко может быть запрограммирован на компьютере. Длины дуг сети с n узлами могут быть заданы массивом $n \times n$.

❖ ***Пример 3.2.*** Для сети, изображенной на рис. 3.10, матрица расстояний представлена в табл. 3.1.

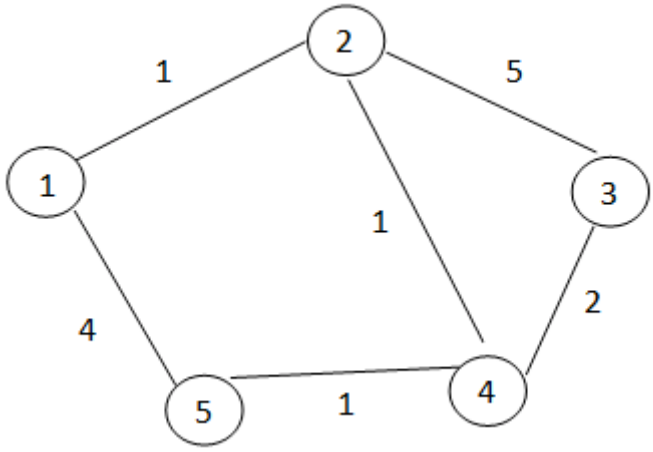


Рис. 3.10. Сеть, пример 3.2

При j =1 сравниваем каждый элемент $d_{i,k}$ $(i \neq 1, k \neq 1)$ с $d_{j,1}+d_{1,k}$.

Если $d_{i,k}>d_{j,1}+d_{1,k}$, то элемент $d_{i,k}$ заменяется на эту сумму. В противном случае элемент не меняется.

При j = 2 сравниваем каждый элемент $d_{i,k}$ $(i \neq 2, k \neq 2)$ с $d_{j,2} + d_{2,k}$. Минимум из чисел $d_{i,k}$ и $d_{j,2}+d_{2,k}$ становится новым значением элемента $d_{i,k}$.

Таблица 3.1.

Матрица расстояний, пример 3.2

узел	**1**	**2**	**3**	**4**	**5**
1	0	1	∞	∞	4
2	1	0	5	1	∞
3	∞	5	0	2	∞
4	∞	1	2	0	1
5	4	∞	∞	1	0

✍ **Замечание 3.** В описанных вычислениях при j=2 использовались результаты, полученные при j = 1.

Для фиксированного значения j необходимо просмотреть элементы в $(n-1)\times(n-1)$ матрице (диагональные элементы всегда равны 0). Каждый элемент сравнивается с суммой двух других элементов: один из той же строки и один из того же столбца. Алгоритм завершает свою работу после окончания вычислений для $j = n$.

✍ **Замечание 4.** Все длины в неориентированной сети симметричны, поэтому достаточно вычислить только одну из двух длин.

Решение примера 3.2. Выполним тройственную операцию для $j = 1, 2, 3, 4, 5$ и запишем только вычисления, которые приводят к изменениям в матрице расстояний.

При $j = 1$ $d_{25} = \min\{d_{25}, d_{21} + d_{15}\} = \min\{\infty, 1 + 4\} = 5$.

Изменения расстояний занесены в табл. 3.2.

Таблица 3.2.

Матрица расстояний при $j = 1$, пример 3.2

узел	**1**	**2**	**3**	**4**	**5**
1	0	1	∞	∞	4
2	1	0	5	1	***5***
3	∞	5	0	2	∞
4	∞	1	2	0	1
5	4	***5***	∞	1	0

При $j = 2$

$d_{13} = \min\{d_{13}, d_{12} + d_{23}\} = \min\{\infty, 1 + 5\} = 6$,

$d_{14} = \min\{d_{14}, d_{12} + d_{24}\} = \min\{\infty, 1 + 1\} = 2$,

$d_{35} = \min\{d_{35}, d_{32} + d_{25}\} = \min\{\infty, 5 + 5\} = 10$.

Изменения расстояний занесены в табл. 3.3.

При $j = 3$ изменений нет.

При $j = 4$

$d_{13} = \min\{d_{13}, d_{14} + d_{43}\} = \min\{6, 2 + 2\} = 4$,

$d_{15} = \min\{d_{15}, d_{14} + d_{45}\} = \min\{4, 2 + 1\} = 3$,

$d_{23} = \min\{d_{23}, d_{24} + d_{43}\} = \min\{5, 1 + 2\} = 3$,

$d_{25} = \min\{d_{25}, d_{24} + d_{45}\} = \min\{5, 1 + 1\} = 2$,

$d_{35} = \min\{d_{35}, d_{34} + d_{45}\} = \min\{10, 2 + 1\} = 3$.

Изменения расстояний занесены в табл. 3.4.

Таблица 3.3.

Матрица расстояний при $j = 2$, пример 3.2

узел	**1**	**2**	**3**	**4**	**5**
1	0	1	***6***	***2***	4
2	1	0	5	1	*5*
3	***6***	5	0	2	***10***

4	***2***	1	2	0	1
5	4	*5*	***10***	1	0

Таблица 3.4.

Матрица расстояний при $j = 4$, пример 3.2

узел	**1**	**2**	**3**	**4**	**5**
1	0	1	***4***	2	***3***
2	1	0	***3***	1	***2***
3	***4***	***3***	0	2	***3***
4	2	1	2	0	1
5	***3***	***2***	***3***	1	0

При $j = 5$ изменений нет.

Кратчайшие расстояния между каждой парой узлов найдены. Итоговые значения кратчайших расстояний приведены в табл. 3.4. Но еще необходимо найти промежуточные (внутренние) узлы для кратчайших путей.

Таблица 3.5.

Начальная таблица промежуточных узлов

узел	**1**	**2**	**3**	**4**	**5**
1	1	2	3	4	5
2	1	2	3	4	5
3	1	2	3	4	5
4	1	2	3	4	5
5	1	2	3	4	5

Чтобы зафиксировать порядок, в котором появляются эти узлы, используем матрицу $[p_{i,k}]$, в которой элемент i-й строки и k-го столбца указывает на *первый внутренний узел* в пути из V_i в V_j. Если $P_{ik} = j$, то кратчайший путь имеет вид $V_i,V_j,...,V_k$. Далее, если $p_{j,k}=s$, то кратчайший путь имеет вид V_i, V_j, V_s,...,V_k. Изначально полагаем $P_{i,k}=k$ для всех i, k. Например, табл. 3.1 соответствует табл. 3.5.

Таким образом, предполагается, что каждая дуга – базисная (до тех пор, пока не установлено обратное), и *первым* внутренним узлом на пути из V_i в V_k является сам V_k.

Во время выполнения тройственной операции над табл. 3.1 также обновляем данные в таблице промежуточных узлов. Элементы таблицы промежуточных узлов изменяются согласно следующему правилу:

$$p_{i,k} = \begin{cases} p_{i,k}, \ \textit{если } d_{ik} > d_{ij} + d_{jk}, \\ \textit{без изменений, если } d_{ik} \le d_{ij} + d_{jk} \end{cases}. \qquad (3.5)$$

Например, когда присваиваем элементу d_{25} значение $d_{21}+d_{15}=5$, также полагаем $p_{25}=p_{21}=1$. Когда присваиваем элементу d_{14} значение $d_{12}+d_{24}=2$, также полагаем $p_{14}=p_{12}=2$. Когда присваиваем элементу d_{15} значение $d_{14}+d_{45}=2$, также

полагаем $p_{15}=p_{14}=p_{12}=2$. Когда присваиваем элементу d_{25} значение $d_{24}+d_{45}=2$, также полагаем $p_{25}=p_{24}=4$. Итак, по окончании вычислений имеем $p_{15}= 2$, $p_{25} = 4$.

Так как дуга e_{45} – базисная, то на протяжении всех вычислений $p_{45} = 5$. $p_{15} = 2$ означает, что V_2 есть первый промежуточный узел на пути из V_1 в V_5. $p_{25} = 4$ означает, что V_4 есть первый промежуточный узел на пути из V_2 в V_5. $p_{45}=5$ означает, что V_5 есть первый промежуточный узел на пути из V_4 в V_5.

Таким образом, можно выписать все промежуточные узлы на пути из V_1 в V_5. Ими являются: V_1, V_2, V_4 и V_5.

В общем случае, чтобы найти кратчайший путь из V_g в V_t, надо найти первый промежуточный узел p_{gt}.

Если $p_{st} = a$, то p_{at} =? Если $p_{at} = b$, то p_{bt} =?

Эти действия завершаются, когда $p_{zt}=t$. Тогда V_s, V_a, V_b,..., V_z, V_t – узлы кратчайшего пути.

Элементы табл. 3.5 изменяются по (3.5) в то же самое время, когда элементы табл. 3.1 изменяются по (3.4). В конце вычислений по (3.4), когда получаем табл. 3.4 кратчайших расстояний, также получаем табл. 3.6 промежуточных узлов, вычисленную по (3.5).

Таблица 3.6.

Таблица промежуточных узлов. Пример 3.2

узел	**1**	**2**	**3**	**4**	**5**
1	1	2	2	2	2
2	1	2	4	4	4
3	4	4	3	4	4
4	2	2	3	4	5
5	4	4	4	4	5

✍ **Замечание 5.** Несмотря на то, что табл. 3.1 симметрична, табл. 3.6 таковой не является. Для получения всей табл. 3.6, надо провести все вычисления над табл. 3.1, а не половину, как в рассмотренном примере.

Например, когда полагаем $d_{25}=d_{51}+d_{12}=5$, также необходимо выполнить присваивания $p_{52} = p_{51} = 1$.

3.3. Поиск остовного дерева в ширину и поиск в глубину. Алгоритмы Прима и Краскала (жадный) для поиска минимального остовного дерева

Дана неориентированная сеть N, необходимо выбрать подмножество дуг, образующих дерево T, в котором существует путь между каждой парой узлов сети. Дерево такого типа называется *остовным деревом* сети.

Поиск остовного дерева в ширину и поиск в глубину

Во многих приложениях нужно в определенном порядке посетить все узлы графа, т. е. построить остовное дерево. Рассмотрим следующие два общих

способа обхода, называемые *поиском в ширину* (BFS, breadth-first-search) и *поиском в глубину* (DFS, depth-first-search).

Поиск в ширину BFS. Выбираем произвольно узел графа G, назовем этот узел V_0 и затем посетим всех соседей V_0 в произвольном порядке, например, это узлы V_1, V_2, …, V_i. После посещения всех соседей V_0 начать обход заново из V_1 (первого посещенного соседа узла V_0) и посетить все соседние с V_1 узлы, скажем, V_{11}, V_{12},…, V_{1j}, потом все новые узлы, соседние с V_2, скажем, V_{21}, V_{22}, …, V_{2k}. Систематически получаем:

Порядок	Соседние узлы
V_0	$V_1, V_2, \ldots, V_i$
V_1	$V_{11}, V_{12},\ldots, V_{1j}$
V_2	$V_{21}, V_{22},\ldots, V_{2j}$
…	…
V_{11}	$V_{11\,1}, V_{11\,2},\ldots, V_{11\,j}$
V_{12}	$V_{12\,1}, V_{12\,2},\ldots, V_{12\,j}$

На рис. 3.11 можно взять за V_0 узел V_a, тогда узлы можно посетить в следующем порядке:

$$V_0, V_b, V_c, V_d, V_e, V_f,$$
$$V_0, V_1, V_2, V_{11}, V_{21}, V_{11\,1}.$$

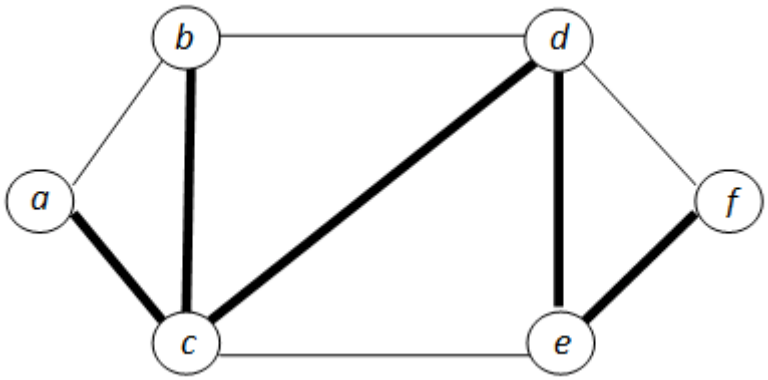


Рис. 3.11. Остовное дерево, поиск в ширину

Если взять в качестве V_0 узел V_b, то можно посетить узлы в порядке

$$V_b, V_a, V_c, V_d, V_e, V_f,$$
$$V_0, V_1, V_2, V_3, V_{21}, V_{31}.$$

✍ **Замечание 1.** После посещения нового узла можно посетить соседей нового узла в произвольном порядке. Здесь используется соглашение о том, что при необходимости выбора узлы посещаются в алфавитном порядке.

✍ **Замечание 2.** Если пометить дугу, соединяющую посещенный узел с ранее посещенным узлом, то все помеченные дуги образуют остовное дерево графа G; если же каждая дуга имеет длину 1, то остовное дерево является деревом кратчайших путей из V_0 во все остальные узлы G.

Поиск в глубину DFS. Выбираем произвольно узел V_0, а затем следуем по дуге e_{01} в узел V_i, потом следуем по дуге e_{12} в узел V_2, соседний с V_1. В общем случае, после посещения узла V_i следуем по дуге e_{ij} в узел V_j, если V_j ранее еще не был посещен. Далее применяем рекурсивно этот процесс к V_j и выбираем дугу e_{jk} в узел V_k. Если узел V_j уже был посещен, то возвращаемся в V_i и выбираем другую дугу. Если все дуги, инцидентные V_i, уже выбраны и нельзя найти ни одного нового узла, то возвращаемся из V_j в предыдущий узел, за которым идет V_i, и проверяем инцидентные ему дуги.

Если на рис. 3.11 начать с узла V_b, то можно посетить узлы в следующем порядке (упорядочение определяется не единственным образом): V_b, V_c, V_a, V_d, V_e, V_f.

Дуги, следующие в новые узлы, образуют остовное дерево. Это дуги: e_{bc}, e_{ca}, e_{cd}, e_{de}, e_{ef}.

Можно сравнить два способа посещения узлов. При BFS нужно проверить все дуги, инцидентные узлу, перед переходом к новому узлу. Таким образом, операция последовательно выполняется веером из узлов. При DFS переход к новому узлу осуществляется сразу после того, как он найден, и происходит проникновение в глубину графа. Только тогда, когда все дуги ведут в старые узлы, идет возврат к предыдущему узлу и из него опять возобновляется DFS.

Алгоритмы BFS и DFS имеют одинаковую сложность для самого неблагоприятого случая.

Сложность алгоритма для самого неблагоприятного случая – это приблизительная мера максимального числа действий, требуемых для выполнения алгоритма, это функция размера входных данных, которые непосредственно в нашем случае были представлены графом (например, $O(n)$). Так как алгоритмы имеют одинаковую сложность, то ни один из них не имеет преимуществ перед другим. Тем не менее, для целого ряда специфических графов один алгоритм мог бы производить дерево покрытия эффективнее, чем другой. Например, поиск «по глубине» эффективнее для графа «колеса», а поиск «по ширине» – для графа «Мальтийский крест».

Минимальное остовное дерево

Если дугам приписаны стоимости d_{ij}, то стоимость остовного дерева определяется как сумма d_{ij} по всем дугам дерева. Остовное дерево с наименьшей стоимостью среди всех остовных деревьев называется *минимальным остовным деревом.*

✍ **Замечание 3.** В общем случае минимальное остовное дерево отличается от дерева кратчайших путей.

Следующие две леммы, несмотря на свою очевидность, требуют детального изучения и доказательства.

🕮 **Лемма 1.** Пусть V_a – произвольный узел и e_{ax} – кратчайшая дуга среди всех дуг, смежных с V_a. Тогда существует минимальное остовное дерево T^*, содержащее дугу e_{ax}.

Доказательство. Пусть T – минимальное остовное дерево, а A – подмножество дуг, смежных с V_a, например, $A = \{e_{ab}, e_{ac}, e_{ad}, e_{ax}\}$. Предположим, что дуга e_{ax} является кратчайшей дугой, смежной с V_a, но не принадлежащей T. Поскольку T – остовное дерево, то в T должен быть путь из V_x в V_a, содержащий одну из дуг A, например, e_{ad}. Обозначим этот путь через (P_{xd}, e_{da}), где P_{xd} – путь из V_x *в* V_d. Заменяя e_{ad} на e_{ax}, получим T^*. Если e_{ax} короче, чем e_{ad}, то заключаем, что T^* – остовное дерево с меньшей стоимостью.

Во-первых, в дереве T^* V_a соединяется с V_x дугой e_{ax}, а с узлом V_d – путем (e_{ax}, P_{xd}). Оставшиеся узлы V_b, V_c по-прежнему связаны с V_a и, следовательно, с остальными узлами сети, значит, T^* является остовным деревом.

Во-вторых, стоимость T^* меньше, чем T, так как e_{ax} короче, чем e_{ad}. Это противоречит предположению о том, что T – минимальное остовное дерево. Если дуги e_{ad} и e_{ax} имеют одинаковые длины, то также можно заменить e_{ad} на e_{ax} и получить минимальное остовное дерево T^*, содержащее e_{ax}. Лемма доказана.

🕮 **Лемма 2.** Если известно, что подмножество дуг, образующих поддерево F, является частью минимального остовного дерева, то существует минимальное остовное дерево, содержащее F, и минимальную дугу, соединяющую F и $N\backslash F$.

Доказательство. Доказательство в точности совпадает с доказательством леммы 1, если заменить V_a на F.

По лемме 1 можно начать из произвольного узла и выбрать наименьшую смежную дугу. Так как известно, что только что выбранная дуга является частью минимального остовного дерева, то можно в лемме 2 взять ее в качестве F и выбрать наименьшую дугу, инцидентную F.

Можно продолжить выбор наименьшей дуги, инцидентной уже выбранной компоненте, что будет являться алгоритмом Прима для нахождения минимального остовного дерева.

Идея алгоритма следующая. Алгоритм создает последовательные связные поддеревья, путем присоединения на каждом шаге одной смежной дуги так, чтобы на каждой стадии была выбрана инцидентная дуга с самой маленькой стоимостью. Пример реализации идеи алгоритма Прима представлен на рис. 3.12. Если начальный узел V_0, то последовательность присоединяемых дуг следующая: дуга со стоимостью 1, смежная (вертикальная) дуга со стоимостью 2, дуга (горизонтальная) со стоимостью 2, дуга со стоимостью 4, дуга со стоимостью 3. Стоимость минимального остовного дерева 12.

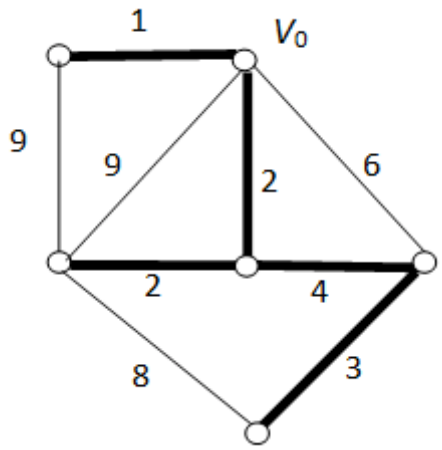


Рис. 3.12. Пример минимального остовного дерева

✍ **Замечание 4.** Алгоритм Прима для минимального остовного дерева близок к алгоритму Дейкстры для нахождения кратчайших путей.

Алгоритм Прима

Пометим узлы, соединенные дугами в остовном дереве, постоянными метками, а еще не соединенные узлы – временными метками.

Шаг 0. Выберем произвольный узел, назовем его V_1 и пометим его постоянным значением ноль (т. е. P_1=0). Пометим все остальные узлы временно значениями T_j, равными d_{1j} для V_j.

Шаг 1. Среди всех временных меток выберем одну (например, T_j) с наименьшим значением и сделаем ее постоянной. Включим дугу со значением $d_{ij} = T_{ij}$ в минимальное остовное дерево, в котором V_i – постоянный узел и $T_j = d_{ij}$.

Шаг 2. Пусть V_j – последний узел, только что ставший постоянным. Для каждого временного узла V_k пусть $T_k \leftarrow \min\{T_k, d_{jk}\}$. Если нет временных меток, то конец, иначе вернуться на шаг 1.

Анализ алгоритма Прима. На шаге 1 производится $(n–1)+(n–2)+ +…+1=O(n^2)$ сравнений. На шаге 2 осуществляется $(n–2)+...+1=O(n^2)$ сравнений. Таким образом, этот алгоритм снова является алгоритмом трудоемкости $O(n^2)$.

✍ **Замечание.** Алгоритм Прима можно использовать и для максимальных остовных деревьев, просто заменяя минимум на максимум (здесь $d_{ij} = –\infty$, если нет дуги).

❖ ***Пример 3.3.*** Проиллюстрируем алгоритм Прима для сети со значениями величин d_{ij}, указанными в табл. 3.7.

Если данные сети представлены в матричной форме, алгоритм

Таблица 3.7.

Матрица расстояний d_{ij}, пример 3.3

узел	**1**	**2**	**3**	**4**	**5**	**6**
1	0	∞	1	∞	3	∞
2	∞	0	∞	6	∞	8
3	1	∞	0	4	2	∞
4	∞	6	4	0	6	7
5	3	∞	2	6	0	∞
6	∞	8	∞	7	∞	0

Прима можно описать следующим образом:

Шаг 0. Вычеркнуть все элементы в первом столбце и пометить первую строку.

Шаг 1. Выбрать минимальный элемент среди всех элементов в помеченных строках, пусть, например, минимальный элемент есть d_{ij} (вычеркнутые элементы выбирать нельзя). Если все элементы в помеченных строках вычеркнуты, то конец.

Шаг 2. Вычеркнуть *j*-й столбец и пометить *i*-ю строку. Вернуться на шаг 1.

Если применить алгоритм Прима к табл. 3.7, то после выбора двух элементов вычисления будут выглядеть так, как показано в табл. 3.8 (выбранные элементы заключены в рамку). Дуги в порядке получения: $e_{13} - e_{35} - e_{34} - e_{42} - e_{46}$.

Таблица 3.8.

Иллюстрация работы алгоритма Прима, пример 3.3

узел	**1**	**2**	**3**	**4**	**5**	**6**
1	0	∞	[1]	∞	3	∞
2	∞	0	∞	6	∞	8
3	1	∞	0	4	[2]	∞
4	∞	6	4	0	6	7
5	3	∞	2	6	0	∞
6	∞	8	∞	7	∞	0
Порядок постоянных меток	*0*	*4*	*1*	*3*	*2*	*5*

Алгоритм Краскала или жадный для поиска минимального остовного дерева

Алгоритм начинают с пустого подграфа. Формируют последовательность из (не обязательно связных) подграфов, добавляя на каждой стадии дугу с самой маленькой стоимостью, не допуская при этом образования петель у существующего подграфа. Когда возникает ситуация, при которой дальнейшее добавление дуг оказывается невозможным, получают результирующий

подграф, представляющий собой минимальное остовное дерево. Для примера сети, изображенной на рис. 3.12, последовательность дуг, присоединяемых к минимальному остовному дереву по алгоритму Краскала следующая: первой присоединяется дуга с наименьшей стоимостью 1, далее две дуги со стоимостью 2, затем дуга со стоимостью 3 и последней дуга стоимости 4. Стоимость минимального остовного дерева составляет 12.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Приведете примеры практического использования задачи о кратчайших путях.
2. Возможно ли в сети существование нескольких кратчайших путей между одними и теми же узлами?
3. Можно ли использовать алгоритм Дейкстры для решения задачи нахождения кратчайших путей между всеми парами узлов?
4. Можно ли использовать алгоритмы нахождения кратчайшего пути от одного узла до другого для решения задачи нахождения кратчайших путей между всеми парами узлов?
5. Можно ли использовать алгоритм Дейкстры в неориентированной сети с отрицательными весами дуг?
6. Справедливо ли следующее утверждение: в неориентированной сети с положительными расстояниями кратчайшая дуга всегда принадлежит дереву кратчайших путей.
7. Справедливо ли следующее утверждение: если все длины дуг сети различны, то существует единственное дерево кратчайших путей из V_0 во все другие узлы.
8. Справедливо ли следующее утверждение: сложность алгоритма поиска «в ширину» в общем случае меньше, чем сложность поиска «в глубину».
9. Можно ли получить одинаковые остовные деревья методом в ширину и глубину, если начало в одном и том же узле?
10. Можно ли получить разные по структуре минимальные остовные деревья методом Прима и «жадным»?
11. Может ли для одной сети вес минимального оставного дерева, полученного с помощью алгоритма Прима, отличаться от веса минимального остовного дерева, полученного алгоритмом Краскала?
12. Можно ли получить одинаковые минимальные остовные деревья методом Прима и «жадным»?
13. Если разбить сеть на две части и построить для каждой части минимальное остовное дерево, затем связать эти части кратчайшей дугой, будет ли тогда результирующее дерево минимальным остовным деревом для всей сети?

4. ЗАДАЧА КОММИВОЯЖЕРА

4.1. Проблема коммивояжера

По условию задачи коммивояжер (торговец) должен посетить каждый из заданных городов и вернуться в первоначальное место. Учитывая сеть дорог, соединяющих различные города на его маршруте, проблема путешествующего коммивояжера состоит в том, чтобы найти маршрут, который минимизирует полное расстояние его путешествия. При этом такой маршрут допускает посещение некоторых городов более одного раза.

Сеть дорог может быть представлена взвешенным графом (сетью). Каждый город представляется узлом, а каждая дорога, соединяющая два города, представляется дугой, соединяющей соответствующие два узла, причем вес дуги равен длине данной дороги. Путешествие, при котором посещается каждый город и которое заканчивается в исходном стартовом положении, представляется замкнутой последовательностью дуг графа, у которого последовательность связанных узлов содержит все узлы.

В терминологии теории графов задача коммивояжера состоит в том, чтобы найти такую замкнутую последовательность дуг, полный вес которой является минимальным. Предположим, что граф связный так, что для коммивояжера действительно оказывается возможным посетить каждый город, используя дороги данной сети.

Как правило, оказывается целесообразным несколько видоизменить исходную проблему следующим образом. Граф, описанный выше, заменяется полным графом с одним узлом для каждого города (*полный граф* – это такой граф, у которого имеется единственная дуга, соединяющая каждую пару отличных узлов). Каждой дуге такого графа приписывается вес, равный наикратчайшему расстоянию между соответствующими городами, в соответствии с используемой сетью дорог. Эти самые короткие расстояния могут быть найдены на основании применения алгоритма Дейкстры к первоначальному графу. Вес дуги в полном графе может быть меньше, чем вес дуги, соединяющей те же самые узлы в первоначальном графе. Это связано с тем, что может найтись маршрут между двумя узлами, итоговая длина которого будет меньшей, чем длина единственной дуги, соединяющей эти два узла. Для сохранения информации относительно исходного графа (который может и не являться полным графом) необходимо хранить записи, связанные с информацией о том, какие пути исходного графа формировали дуги полного графа.

Замкнутая последовательность дуг, позволяющая посетить все узлы графа, представляющего дорожную сеть, соответствует цепи Гамильтона в полном графе. Таким образом, проблема коммивояжера может быть сформулирована следующим образом.

Для заданного связанного, взвешенного, полного графа требуется построить цепь Гамильтона минимального веса, т. е. минимальную цепь Гамильтона.

Напомним, что не каждый граф имеет цепь Гамильтона; однако, каждый полный граф, имеющий, по крайней мере, три узла, имеет цепь Гамильтона, что непосредственно следует из достаточного условия гамильтонова графа (если простой граф с числом вершин $n \geq 3$ и если для каждой вершины ее валентность $\geq 0{,}5n$, то такой граф является гамильтоновым). Так как наши графы конечны, то может иметься только конечное число цепей Гамильтона, и, следовательно, среди них должна быть и минимальная.

Так как веса дуг в полном графе – это самые короткие расстояния между узлами первоначального графа сети дорог, то для полного графа должно выполняться следующее неравенство треугольника.

Для каждого триплета отличных узлов (V_1, V_2, V_3) справедливо:

$$d_{12} + d_{23} \geq d_{13},$$

где d_{ij} – вес единственной дуги, соединяющей V_i и V_j.

Проблема коммивояжера ввиду своего большого практического значения получила значительное внимание со стороны теоретиков, и было разработано много различных алгоритмов для строительства минимальной цепи Гамильтона.

Одна из причин большого интереса к данной проблеме связана с тем фактом, что все известные алгоритмы, которые решают данную проблему, являются в вычислительном отношении неэффективными относительно числа узлов, т. е. являются алгоритмами неполиномиальной сложности. Тем не менее, известны различные эффективные алгоритмы, применяемые на практике, которые дают приближенное решение. Другими словами, они обеспечивают цепи Гамильтона, чей вес оказывается близким к минимально возможному, однако при этом нельзя быть уверенным в том, что данный вес является самым минимальным из возможных.

4.2. Алгоритмы «ближайшего соседа», «самой близкой вставки»

Имеется достаточно очевидный и простой «приближенный» алгоритм, так называемый алгоритмом ***«ближайшего соседа»***, являющийся своего рода модификацией алгоритма поиска по глубине.

Идея алгоритма «ближайшего соседа». Алгоритм начинается в любом узле и «идет» по дугам с наименьшим весом, которые (дуги) оказываются инцидентными узлу. На каждом шаге алгоритма начинаем развитие событий с последней из числа посещенных узлов и двигаемся вдоль дуг с наименьшим из возможных значений веса по направлению к новым узлам (могут иметься несколько возможностей выбора «минимальной дуги» на каждой стадии). Когда все узлы окажутся посещенными, возвращаемся к стартовому положению по единственной дуге полного графа от последнего узла назад к первому.

Наиболее похожим на описанный является жадный алгоритм (в том смысле, что именно ближайшие узлы посещаются на каждой стадии). Оказывается, что жадные алгоритмы чрезвычайно «бедны» в следующем смысле. Хотя они в некоторых случаях и будут производить минимальный цикл Гамильтона, как правило, цикл, произведенный такими алгоритмами, может иметь вес, значительно превышающий возможный минимум.

Фактически работа данных алгоритмов настолько плоха, насколько это можно себе представить. Взяв любое положительное целое число k (сколь угодно большое), всегда найдутся графы, для которых вес цикла Гамильтона, полученного на основании алгоритма «ближайшего соседа», окажется в k раз больше веса минимального цикла.

Рассмотрим другой «приближенный» алгоритм – алгоритм «***самой близкой вставки***». Он гарантирует нахождение цикла Гамильтона с полным весом, который не более чем в два раза превышает минимальный цикл. Причем, как правило, данный алгоритм не достигает этого двойного превышения.

Основной шаг в данном алгоритме состоит в том, чтобы взять в графе цикл и увеличить его, включая в него самые близкие к нему узлы. Этот шаг повторяется до тех пор, пока все узлы не окажутся включенными в цикл.

Алгоритм может применяться к любому полному взвешенному графу, для которого выполняется неравенство треугольника.

Алгоритм «самой близкой вставки»

Шаг 1. Выбрать любой узел. Выбрать инцидентную узлу дугу e с наименьшим весом, и пусть C является последовательностью дуг: e, e. C – стартовый «цикл», (хотя, строго говоря, это не цикл, так как данная последовательность содержит повторяющуюся дугу).

Шаг 2. Выбрать дугу с наименьшим весом, которая присоединяет инцидентный ей узел, находящийся вне цикла C, к узлу, входящему в цикл C (может иметься несколько вариантов таких дуг).

Шаг 3. Увеличение цикла в результате включения выбранного узла V_0. Чтобы решить, как вставить V_0, следует рассмотреть все пары V_1, V_2 смежных узлов цикла C и выбрать такую пару, для которой выражение: $I=d_{10}+d_{02}-d_{12}$ является минимальным, где d_{ij} – вес дуги, соединяющей V_i и V_j. Выражение I представляет увеличение полного веса цикла C после включения в него узла V_0. Увеличиваем цикл C, включая узел V_0 путем присоединения дуги, соединяющей узлы V_1 и V_0, и дуги, соединяющей узлы V_0 и V_2, и удаляя затем дугу, соединяющую узлы V_1 и V_2.

Шаг 4. Повторяем шаги 2 и 3, пока цикл не будет включать в себя все узлы графа.

❖***Пример 4.1.*** Проиллюстрируем алгоритм «самой близкой вставки» для сети, представленной на рис. 4.1 (которая удовлетворяет неравенству треугольника). Веса дуг представлены в матрице расстояний (табл. 4.1).

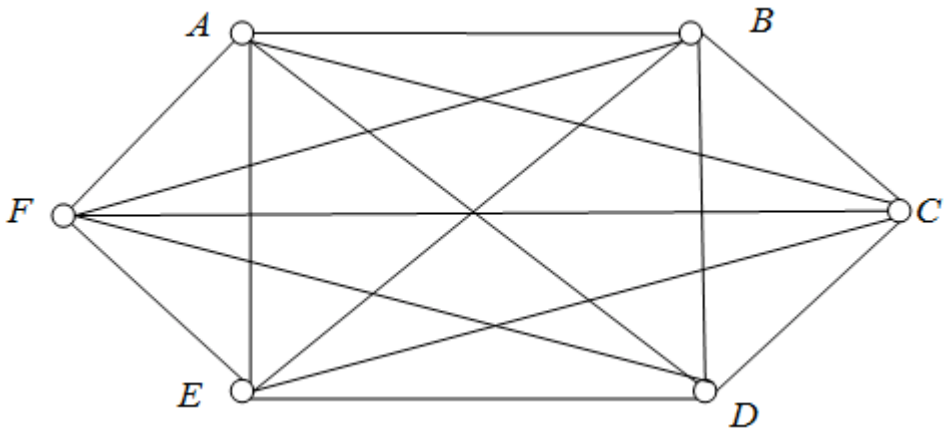


Рис. 4.1. Сеть, пример 4.1

Таблица 4.1.

Матрица расстояний d_{ij}, пример 4.1

узел	***A***	***B***	***C***	***D***	***E***	***F***
A	0	4	9	12	10	3
B	4	0	6	8	10	6
C	9	6	0	5	9	12
D	12	8	5	0	4	11
E	10	10	9	4	0	7
F	3	6	12	11	7	0

Чтобы выполнить шаг 1, сначала выберем произвольный узел, например, *A* (рис. 4.1). Дуга e_{AF} инцидентна узлу *A* и имеет наименьший вес из числа дуг, инцидентных узлу *A*. Поэтому первый цикл будет: e_{AF}, e_{FA} (от узла *A* до *F* и назад к *A*).

Дуга с наименьшим весом, инцидентная узлам *A* или *F* – это дуга e_{AB}, поэтому узел *B* – первый узел, который следует вставить. Самый короткий путь, которым узел *B* может быть вставлен в цепь, будет: от *A* до *B* и назад через *F*. Это порождает цепь e_{AB}, e_{BF}, e_{FA}, показанную на рис. 4.2.

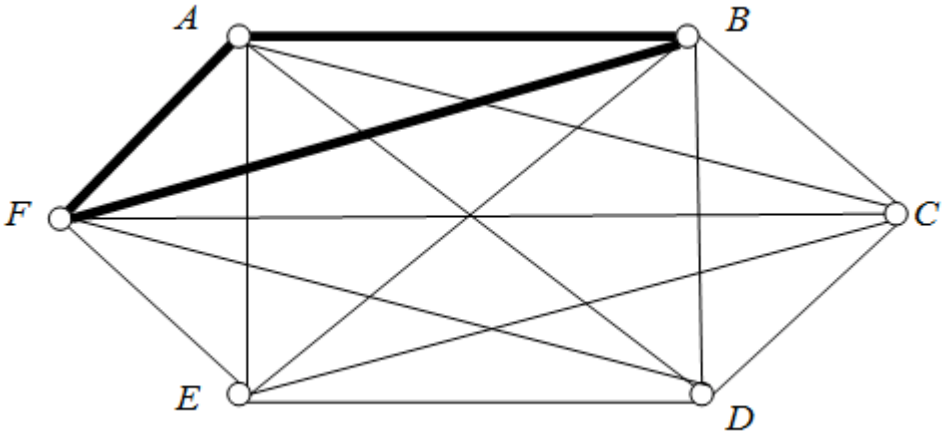


Рис. 4.2. Цепь e_{AB}, e_{BF}, e_{FA}

Узел C является самым близким к узлу этой цепи, поэтому необходимо найти лучший способ вставить его. Цены I для трех дуг текущей цепи будут:

$$I(e_{AB}) = d_{AC} + d_{CB} - d_{AB} = 9 + 6 - 4 = 11,$$
$$I(e_{BF}) = d_{BC} + d_{CF} - d_{BF} = 6 + 12 - 6 = 12,$$
$$I(e_{FA}) = d_{FC} + d_{CA} - d_{FA} = 12 + 9 - 3 = 18.$$

Таким образом, увеличиваем цепь путем удаления дуги e_{AB} и вставляя на его место дуги e_{AC}, e_{CB}. Это дает цепь, показанную на рис. 4.3.

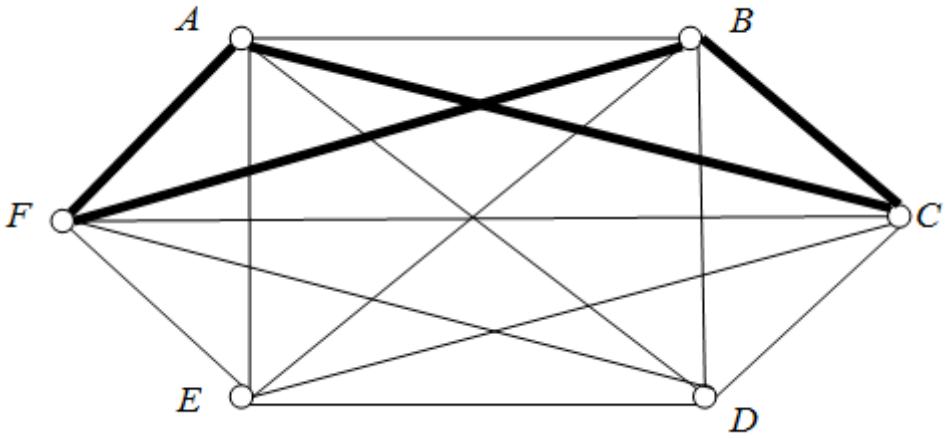


Рис. 4.3. Цепь из четырех дуг, пример 4.1

Повторяя этот процесс дважды, увеличиваем цепь, включая сначала узел D, а затем E. Заключительная цепь: e_{AC}, e_{CD}, e_{DE}, e_{EB}, e_{BF}, e_{FA} является требуемой цепью Гамильтона с полным весом 9 + 5 + 4+ 10+ 6 + 3 = 37.

Этот пример иллюстрирует тот факт, что алгоритм «самой близкой вставки» может не производить минимальную цепь Гамильтона. Граф действительно имеет единственную минимальную цепь Гамильтона – e_{AB}, e_{BC}, e_{CD}, e_{DE}, e_{EF}, e_{FA} с полным весом 29.

4.3. Метод ветвей и границ для задачи коммивояжера

Во многих комбинаторных задачах необходимо найти все конфигурации, удовлетворяющие некоторым требованиям. Один из подходов к решению таких задач состоит в генерировании всех возможных конфигураций одна за другой. Эта техника известна как *исчерпывающий поиск*. Один из способов систематического поиска всех возможных решений называется *бэктрекингом* или *поиском с возвращением*. Эта общая техника может не выдерживать конкуренции с алгоритмами, специально приспособленными для тех или иных задач. Однако бэктрекинг, благодаря его общности, получил широкое признание специалистов.

Рассмотрим вариант бэктрекинга, известный как *метод ветвей и границ*. При бэктрэкинге, когда частичный вектор решений не удовлетворяет ограничениям, этот вектор и все его потомки исключаются из поиска. Если приписать большую стоимость недопустимым векторам и нулевую стоимость

допустимым векторам, то необходимо будет искать вектор минимальной стоимости. Во многих случаях частичным векторам можно естественным образом приписать стоимости (оценки), удовлетворяющие соотношению:

стоимость$(x_1, x_2, \ldots, x_n) \le$ стоимости $(x_1, x_2, \ldots, x_n)$

для всех возможных значений переменных.

В методе ветвей и границ множество допустимых решений разбивается на подмножества частичных решений. Для этого удобно использовать дерево решений. Для каждого узла (подмножества допустимых решений) которого определенным образом подсчитывают нижнюю оценку. Оценка по определенным правилам позволяет выбирать подмножество для дальнейшего построения частичного решения и позволяет отсеять неперспективные подмножества для сокращения перебора.

Рассмотрим схему метода ветвей и границ для следующей общей задачи комбинаторной оптимизации:

$$f\left(x^0\right) = \min f(x), x \in G, |G| = N < \infty.$$

Алгоритм основан на следующих построениях, позволяющих уменьшить объем перебора.

1. *Вычисление оценки*. Пусть $G' \subset G$, тогда $\varphi(G')$ называется *нижней оценкой*, если для любого $x \subset G'$ выполняется неравенство:

$$f(x) \ge \varphi(G').$$

2. *Ветвление*. Разбиение множества G на непересекающиеся подмножества. Эти множества образуют список задач для ветвления. Выберем одно из них и снова повторим процедуру разбиения.

3. *Пересчет оценок*. Если $G_2 \subset G_1$, то $\min\limits_{x \subset G_2} f(x) \ge \min\limits_{x \subset G_1} f(x)$. Поэтому, разбивая в процессе ветвления подмножество G' на непересекающиеся подмножества G'_i, полагают

$$\varphi(G'_i) \ge \varphi(G').$$

4. *Вычисление планов* (допустимых решений). Для этого используют особенности решаемой задачи. Если на шаге ветвления с номером k известен план x^k, на шаге с номером (k+1) – план x^{k+1} и если $f(x^{k+1}) < f(x^k)$, то план x^k забывается, и вместо него сохраняется план x^{k+1}. Наилучшее из полученных допустимых решений называют *рекордом*.

5. *Признак оптимальности.* Пусть $G = \bigcup\limits_{i=1}^{s} G_i$, $\bar{x} \in G_V$. Тогда план $\bar{x}$ является оптимальным, т. е. $\bar{x} = x^0$, если выполняется условие $f(\bar{x}) = \varphi(G_V) \le \varphi(G_i), i = \overline{1, s}$.

6. *Правило отсева.* Пусть $G = \bigcup_{i=1}^{s} G_i$, x^0 – оптимум, x^k – рекорд. Если $\varphi(G_r) > f(x^k)$, то множество G_r можно отсеять, т. е. исключить из дальнейшего рассмотрения, так как оно не может содержать оптимальных решений.

7. *Конечность алгоритма* следует из конечности множества *G.*

Эффективность алгоритма ветвей и границ определяется числом решенных подзадач. Решение задачи состоит из двух основных этапов. На первом – находится оптимальное решение. На втором – производится доказательство оптимальности полученного решения. Второй этап, как правило, оказывается более трудоемким, чем первый.

Алгоритм метода ветвей и границ на примере задачи коммивояжера

Рассмотрим конкретизацию метода ветвей и границ на примере задачи коммивояжера.

Формальная постановка задачи коммивояжера представлена выше. Здесь будем использовать следующую математическую модель задачи. Пусть задано *n* точек и известна матрица $c_{ij} \geq 0$ – матрица расстояний между точками. Под маршрутом коммивояжера *z* будем понимать перестановку $i_1, i_2, ..., i_n$, длина маршрута равна

$$l(z) = \sum_{k=1}^{n} c_{i_k, i_{k+1}}, \ (i_{n+1} = i_1).$$

Пусть *Z* – множество всех возможных маршрутов ($|Z| = (n-1)!$). Требуется найти маршрут $z_0 \in Z$, такой что

$$l(z_0) = \min l(z), z \in Z.$$

Метод ветвей и границ применяется для каждого маршрута для нахождения оптимального пути. В качестве оценочной подзадачи будем использовать приведение матрицы расстояния.

Приведение матрицы расстояний. В любой маршрут входит один элемент каждой строки и каждого столбца матрицы расстояний. Поэтому если из всех элементов некоторой строки вычесть одно и то же число, то в новой матрице длины всех маршрутов уменьшатся на это число по сравнению со старой матрицей. То же можно сказать и о столбцах матрицы расстояний. Эти свойства объясняются тем, что из каждого города необходимо один раз уйти, и в каждый город один раз прийти. Поэтому найдем

$$r_i = \min_j c_{ij} \tag{4.1}$$

и образуем новую матрицу:

$$c_{ij}^{(1)} = c_{ij} - r_i. \tag{4.2}$$

В матрице $c_{ij}^{(1)}$ найдем наименьший элемент в каждом столбце

$$h_j = \min_i c_{ij} \tag{4.3}$$

и построим матрицу:

$$c_{ij}^{(2)} = c_{ij}^{(1)} - h_j. \tag{4.4}$$

Тогда

$$c_{ij} = c_{ij}^{(1)} + r_i = c_{ij}^{(2)} + h_j + r_i. \tag{4.5}$$

Матрица $c_{ij}^{(2)}$– приведенная матрица расстояний, r_i – константа приведения по строкам, h_j – константа приведения по столбцам. Матрица $c_{ij}^{(2)}$ имеет хотя бы один ноль в каждой строке и в каждом столбце.

Пусть *z* – некоторый маршрут, тогда

$$l(z) = \sum_{k=1}^{n} c_{i_k} i_{k+1} = \sum_{k=1}^{n} c_{i_k}^{(2)} i_{k+1} + \sum_{k=1}^{n} r_k + \sum_{k=1}^{n} h_k. \tag{4.6}$$

$$d_0 = \sum_{k=1}^{n} (r_k + h_k). \tag{4.7}$$

Если обозначить через $l_1(z)$ длину маршрута коммивояжера в матрице расстояний $c_{ij}^{(2)}$ и воспользоваться (4.7), то получим $l(z) = l_1(z) + d_0$; так как $l_1(z) \geq 0$, то $l(z) \geq d_0$, т. е. длина любого маршрута оценивается снизу числом d_0.

Если после применения процедуры приведения в каждой строке и в каждом столбце есть ровно один нуль, то мы решили задачу о назначении. Величина целевой функции равна d_0, а нули указывают назначение. Если нули образуют единственный цикл, то мы решили задачу о коммивояжере, и длина кратчайшего маршрута равна d_0.

Ветвление. Рассмотрим некоторый маршрут, который содержит переход из точки *i* в точку *j*. Для длины этого маршрута имеет место неравенство $l(z) \geq d_0 + c_{ij}^{(2)}$.

Для переходов, у которых $c_{ij}^{(2)} = 0$, получаем, что $l(z) \geq d_0$. Естественно предположить, что кратчайший путь коммивояжера содержит один из нулевых переходов.

Обозначим через (i, j) множество маршрутов, содержащих переход из *i* в *j* непосредственно, и через $\overline{(i, j)}$ – множество маршрутов, не содержащих такой переход, $G^0 = Z, G_1^1 = (i, j), \; G_2^1 = \overline{(i, j)}, \; Z = (i, j) \cup \overline{(i, j)}$. Таким образом,

множество Z разбивается на два подмножества (рис. 4.4). Ветвление на два подмножества называется *дихотомическим*.

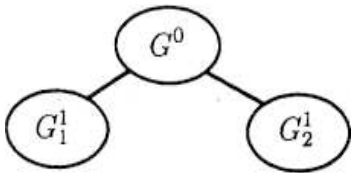


Рис. 4.4. Пример дихотомического ветвления

Рассмотрим нижнюю оценку для множества маршрутов $\overline{(i,j)}$. Для (i,j) оценка остается d_0. Для всех маршрутов из $\overline{(i,j)}$ из точки i нужно уйти в точку $k \neq j$ и в точку j прийти из точки $m \neq i$, поэтому имеем

$$l(z) \geq d_0 + c_{ik}^{(2)} + c_{mj}^{(2)}, k \neq j, m \neq i.$$

Поскольку выход из точки i добавляет к оценке, по крайней мере, наименьший элемент строки i, а вход в точку j – по крайней мере, наименьший элемент столбца j, то $l(z) \geq d_0 + \min\limits_{k \neq j} c_{ik}^{(2)} + \min\limits_{m \neq i} c_{mj}^{(2)}$.

Для каждого нулевого перехода может быть вычислена величина

$$\theta_{ij} = \min_{k \neq j} c_{ik}^{(2)} + \min_{m \neq i} c_{mj}^{(2)}. \qquad (4.8)$$

Для вычисления θ_{ij} для нулевого перехода нужно найти минимальные элементы в строке и столбце, пересекающиеся на этом переходе (минимумы находятся без учета рассматриваемого нулевого перехода (i,j)), и полученные минимумы сложить.

В качестве перехода, используемого для разветвления, выбирается тот, у которого значение θ_{ij} максимально, $l(z) \geq d_0 + \theta_{ij}$. Находим $\max\limits_{c_{ij}^{(2)}=0} \theta_{ij}$, и этот переход выбираем в качестве того, по которому происходит разветвление. Это связано с тем, что необходимо получить по возможности большую оценку для множества $\overline{(i,j)}$.

В матрице, соответствующей множеству $G_1^1 = (i,j)$, вычеркиваем строку i и столбец j и положим, $c_{ji}^{(2)} = M$, чтобы предотвратить появление цикла $i \to j \to i$. Полученную матрицу можно привести; пусть сумма констант приведения равна α_0, тогда для $z \in (i,j)$ получим $l(z) \geq d_0 + \alpha_0$.

В матрице, соответствующей множеству $G_2^1 = \overline{(i,j)}$, следует положить $c_{ij}^{(2)} = M$, и для $z \in \overline{(i,j)}$ получим $l(z) \geq d_0 + \theta_{ij}$.

Если эту матрицу привести, и сумма констант приведения равна β_0, то для $z \in \overline{(i,j)}$ получаем $l(z) \geq d_0 + \theta_{ij} + \beta_0$.

Модифицируем список множеств, не подвергшихся разбиению. Вместо множества $G_1^1 = (i,j)$ записываем множества $\overline{(k,p)}$ и (k,p), и полученные множества переобозначаем: $G_1^{t+1}, G_2^{t+1}, ..., G_{r_{t+1}}^{t+1}$, где t=1. Удаляем (или отмечаем) множества, отсеянные по правилам отсева, и переходим к множеству G_v^t с наименьшей нижней оценкой d_v^t, положив t равным t+1.

В результате применения описанного выше алгоритма получаем дерево ветвления в виде рис. 4.5

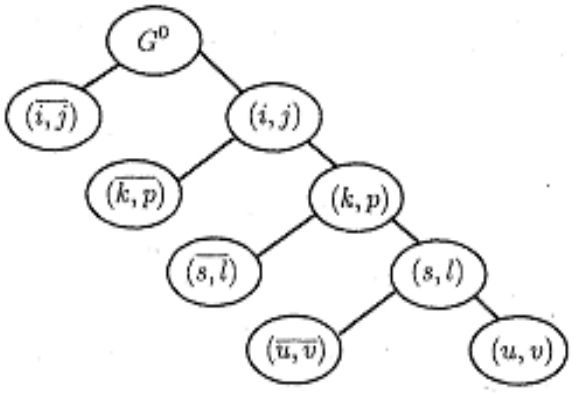


Рис. 4.5. Дерево ветвления

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем заключается проблема коммивояжера?
2. Можно ли для графа построить полный граф с действительными кратчайшими расстояниями для добавленных ребер, используя алгоритм Флойда-Уоршелла, и при этом будет ли выполняться правило «неравенство треугольника»?
3. Верно ли, что алгоритм «ближайшего соседа» имеет полиномиальную сложность? Если да, то докажите это.
4. Верно ли, что алгоритм «самой близкой вставки» имеет неполиномиальную сложность?
5. Можно ли получить различные маршруты коммивояжера с одинаковой протяженностью, если применять алгоритм «самой близкой вставки» от одного и того же начального узла?
6. Можно ли получить различные маршруты коммивояжера с разной протяженностью, если применять алгоритм «самой близкой вставки» от одного и того же начального узла?
7. Справедливо ли следующее утверждение: маршрут коммивояжера, найденный алгоритмом «самой близкой вставки», всегда лучше, чем маршрут, полученный с помощью алгоритма «ближайшего соседа».

8. Можно ли алгоритмом «ближайшего соседа» найти оптимальный маршрут коммивояжера?

9. Верно ли следующее утверждение: с помощью алгоритма «ближайшего соседа» нельзя получить маршрут коммивояжера с длиной больше чем в 10 раз длиннее оптимального?

10. Верно ли следующее утверждение: с помощью алгоритма «ближайшего соседа» всегда получаем маршрут коммивояжера с длиной больше чем в 2 раза длиннее оптимального?

11. Верно ли следующее утверждение: с помощью алгоритма «ближайшего соседа» можно получить маршрут коммивояжера с длиной больше чем в k (k любое положительное число) раз длиннее оптимального?

12. Какие модификации для улучшения алгоритма «ближайшего соседа» вы знаете?

13. Верно ли, что если начинать алгоритм «ближайшего соседа» с каждой вершины графа, то это позволит найти оптимальное решение за m идентичных шагов, где m – число вершин.

14. Может ли существовать пример задачи коммивояжера, для которого алгоритм ветвей и границ найдет оптимальное решение за один шаг? Чем будет обоснована оптимальность?

15. Чем обоснована конечность алгоритма ветвей и границ?

16. Каким образом в алгоритме ветвей и границ происходит ограничение просмотра всех допустимых решений?

17. Для чего при решении задачи коммивояжера методом ветвей и границ используется приведение матрицы расстояний?

18. Можно ли определить заранее необходимое количество оценочных задач при решении задачи коммивояжера алгоритмом ветвей и границ?

19. Как используется верхняя и нижняя оценки в методе ветвей и границ?

20. Как определить верхнюю оценку при решении задачи коммивояжера?

21. Как в процессе решения задачи коммивояжера алгоритмом ветвей и границ изменяется значение верхней оценки?

22. Как в процессе решения задачи коммивояжера алгоритмом ветвей и границ изменяется значение нижней оценки?

5. СЕТЕВОЕ ПЛАНИРОВАНИЕ

5.1. Задача о кратчайшем сроке. Задача о критическом пути

Выполнение комплексных научных исследований, а также проектирование и строительство крупных промышленных, сельскохозяйственных и транспортных объектов требуют календарной увязки большого числа взаимосвязанных работ, выполняемых различными организациями. Составление и анализ соответствующих календарных планов представляют собой весьма сложную задачу, при решении которой применяются так называемые *методы сетевого планирования.*

При использовании указанных методов зависимость между отдельными работами задается с помощью некоторого ориентированного графа (орграфа) Г, в котором каждая дуга $s = \overline{1,m}$ отвечает определенной работе. Если в этом графе дуги $s_1, s_2,...,s_l$ образуют путь из узла i_{s_1} в узел j_{s_l}, то это означает, что выполнение работ *s*, для которых $i_s = j_{s_l}$, может начинаться лишь после завершения всех работ $s_1, s_2,...,s_l$. Ввиду этого каждый узел *i* орграфа Г интерпретируют как событие, означающее возможность выполнения тех работ *s*, для которых $i_s = i$, т. е. завершение всех работ *s*, для которых $j_s = i$.

Приведенная интерпретация означает, что для выполнимости рассматриваемого комплекса работ соответствующий орграф Г не должен содержать контуров. Действительно, если бы имелся некоторый контур $s_1, s_2,...,s_l$, то для выполнения каждой работы $s_\nu, \nu = \overline{1,l}$ потребовалось бы предварительно завершить все остальные работы, что приводит к неразрешимому вопросу, подобного известному вопросу о том, кто снес яйцо, из которого вылупилась первая курица.

Следовательно, практически реализуемые комплексы работ описываются орграфами Г, не содержащими контуров. Ясно, что в каждом таком орграфе имеется, по крайней мере, один начальный узел i_0, для которого

$$N_{i_0}^{+} = \left\{ s \mid j_s = i_0 \right\} = \varnothing,$$

и, по крайней мере, один конечный узел i_{00}, для которого

$$N_{i_{00}}^{-} = \left\{ s \mid i_s = i_{00} \right\} = \varnothing.$$

В сетевом планировании обычно ограничиваются рассмотрением случая, когда i_0=1 является единственным начальным, а i_{00}=*n* – единственным конечным узлом. Первый из этих узлов интерпретируется как начальное событие, а второй – как конечное событие, означающее завершение всего комплекса работ. При этом, как нетрудно проверить, для каждого узла $i \neq 1$ в орграфе Г имеется путь с концом в этом узле и началом в i_0=1. Точно так

же для каждого узла $i \neq n$ имеется путь из этого узла в конечный i_{00}=*n*.

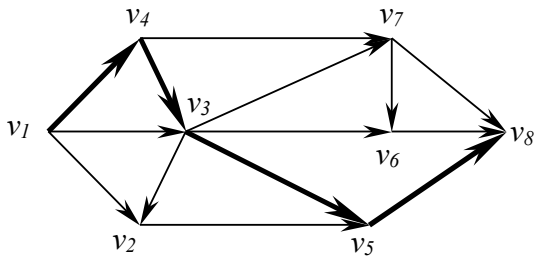


Рис. 5.1. Орграф Г

Легко проверить, что в орграфе Г, изображенном на рис. 5.1, нет контуров, и в нем имеются один начальный узел i_0=1 и один конечный i_{00}=8. Жирной линией показан путь из начального узла i_0= 1 в конечный i_{00}= 8.

Допустим теперь, что рассматриваемый комплекс работ описывается орграфом Г, обладающим указанными свойствами. Кроме того, известно время τ_s, необходимое для выполнения каждой работы $s = \overline{1,m}$. В этом случае говорят, что имеется сетевой график Г, где $T = \{\tau_s\}_{s=\overline{1,m}}$.

Календарный план выполнения сетевого графика Г определяется выбором *m*-мерного вектора

$$t = (t_1, t_2, ..., t_n), \tag{5.1}$$

компоненты которого указывают планируемые сроки выполнения соответствующих событий. Такой план является допустимым (согласован с заданными величинами τ_s), если он удовлетворяет условию

$$t_{j_s} - t_{i_s} \geq \tau_s,\ s = \overline{1,m}. \tag{5.2}$$

При этом срок выполнения всего комплекса работ определяется величиной

$$\lambda(t) = t_n - t_1. \tag{5.3}$$

Так как мы заинтересованы в минимизации общего срока выполнения работ. Таким образом, приходим к следующей задаче линейного программирования.

Задача о кратчайшем сроке. *При заданном сетевом графике Г определить вектор* (5.1), *минимизирующий линейную функцию* (5.3) *при ограничениях* (5.2).

Приведенная задача, как нетрудно проверить, является двойственной к следующей задаче, представляющей также самостоятельный интерес.

Задача о критическом пути. *При исходных данных задачи о кратчайшем сроке определить n-мерный вектор*

$$x = (x_1, x_2, ..., x_m),\ x_s \geq 0,\ s = \overline{1,m}, \tag{5.4}$$

максимизирующий линейную функцию

$$\mu(x) = \sum_{s=1}^{m} \tau_s x_s \tag{5.5}$$

при ограничениях

$$\sum_{s \in N_i^+} x_s - \sum_{s \in N_i^-} x_s = \begin{cases} -1 \text{ при } i = 1, \\ 0 \text{ при } i = \overline{2,n-1}, \\ 1 \text{ при } i = n. \end{cases} \tag{5.6}$$

Ясно, что задача о критическом пути является частным случаем сетевой транспортной задачи, в которой b_1=1, b_n=–1 и b_i=0, $i = \overline{2,n-1}$, а $d_s = \tau_s$, $s = \overline{1,m}$.

Рассмотренные задачи могут решаться простым методом, сводящимся к последовательным просмотрам дуг сетевого графика.

Для каждого узла $i \neq 1$, как уже отмечалось, имеется, по крайней мере, один путь $p = \{s_1, s_2, ..., s_l\}$ с концом в этом узле и началом в узле i_0= 1. Множество таких путей обозначим через P_i^+ и рассмотрим величины

$$t_1^0 = 0,\ t_i^0 = \max_{p \in P_i^+} \sum_{s \in p} \tau_s,\ i = \overline{2,n}. \tag{5.7}$$

Нетрудно проверить, что они удовлетворяют соотношениям

$$t_{j_s}^0 - t_{i_s}^0 \geq \tau_s,\ s = \overline{1,n},\ t_n^0 = \lambda_0, \tag{5.8}$$

где λ_0– определяемый формулой (5.3) кратчайший срок для сетевого графика Г. Но тогда в качестве решения задачи о кратчайшем сроке может быть принят вектор (5.1) с компонентами (5.7).

Покажем теперь, что для практического вычисления величин (5.7) нет необходимости в предварительном определении соответствующих множеств путей P_i^+. Для этого заметим, что интересующие нас величины (5.7) связаны между собой соотношениями

$$t_i^0 = \max_{s \in N_i^+} (t_{i_s}^0 + \tau_s),\quad i = \overline{2,n}, \tag{5.9}$$

где под N_i^+, как и выше, понимается множество дуг с концом в узле *i*. Используя приведенные формулы, можем определить все величины (5.7) за $l \leq l_0 + 1$ просмотров списка дуг сетевого графика, где через l_0 обозначена максимальная длина путей $p \in P_n^+$. Соответствующий алгоритм для вычисления величин (5.7) поясним более подробно на примере.

❖***Пример 5.1.*** Рассмотрим сетевой график Г, отвечающий числовым данным, приведенным в табл. 5.1. Заметим, что соответствующий орграф Γ совпадает с изображенным на рис. 5.1.

Таблица 5.1.

Исходные данные сетевого графика, пример 5.1

s	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i_s, j_s	1, 2	1, 3	1, 4	2, 5	3, 2	3, 5	3, 6	3, 7	4, 3	4, 7	5, 8	6, 8	7, 6	7, 8
τ_s	10	3	2	5	7	20	12	4	10	5	10	5	2	12

Предположим, что уже имеются некоторые неотрицательные величины $t_i \le t_i^0$, $i = \overline{1,8}$. Тогда при просмотре очередной дуги s новое значение t'_{j_s} определяется по формуле

$$t'_{j_s} = \max\left\{t_{j_s}, t_{i_s} + \tau_s\right\}.$$

Получаемые при этом величины t'_{j_s}, очевидно, также не превосходят $t^o_{j_s}$. Далее, если при очередном просмотре всех дуг $s = \overline{1,14}$ не происходит ни одного изменения величин t_i, то это означает, что имеющиеся t_i совпадают с искомыми величинами (5.9).

В качестве исходных принимаем t_i=0, $i = \overline{1,8}$. Затем, просматривая дуги $s = \overline{1,14}$, последовательно находим новые значения

$t_2 = 10$, $t_3 = 3$, $t_4 = 2$, $t_5 = 15$, $t_2 = 10$, $t_5 = 23$, $t_6 = 15$, $t_7 = 7$, $t_3 = 12$, $t_7 = 7$, $t_8 = 33$, $t_8 = 33$, $t_6 = 15$, $t_8 = 33$,

которые приведены во второй строке табл. 5.2.

На втором просмотре дуг $s = \overline{1,14}$ определяем новые значения величин t_i, приведенные в третьей строке табл. 5.2. На третьем просмотре дуг $s = \overline{1,14}$ ни одна из найденных величин t_i не изменяется. Следовательно, полученные при втором просмотре величины t_i совпадают с искомыми величинами t_i^o.

Таблица 5.2.

Новые значения t_i, пример 5.1

Просмотр	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Первый	0	10	12	2	23	15	7	33
Второй	0	19	12	2	32	24	16	42

Рассмотренные величины (5.7) определяют минимально возможные сроки выполнения всех событий.

При анализе сетевых графиков представляет интерес выявление имеющихся резервов в сроках выполнения отдельных событий. С этой целью через P_i^- для каждого узла $i=\overline{1,n-1}$ обозначим множество путей из этого узла в конечный *i=n.* Затем рассмотрим величины

$$t_n^{oo}=t_n^o,\ t_n^{oo}=t_n^o-\max_{p\in P_i^-}\sum_{s\in p}\tau_s,\ i=\overline{1,n-1}. \qquad (5.10)$$

Так как множество путей P_1^-, очевидно, совпадает с множеством путей P_n^+, то

$$t_1^{oo}=t_n^{oo}-\max_{p\in P_i^-}\sum_{s\in p}\tau_s=t_n^o-\max_{p\in P_n^+}\sum_{s\in p}\tau_s=t_n^o-t_n^o=0=t_1^o.$$

Для остальных узлов $i=\overline{2,n-1}$ имеют место неравенства

$$t_i^o\le t_i^{00},\ i=\overline{2,n-1}. \qquad (5.11)$$

При этом неотрицательные величины

$$\rho_i=t_i^{oo}-t_i^0,\ i=\overline{2,n-1} \qquad (5.12)$$

можно интерпретировать как имеющиеся резервы времени для выполнения соответствующих событий.

Событие *i* сетевого графика Г называется *критическим,* если в соответствующем неравенстве (5.11) достигается равенство, т. е. резерв времени (5.12) для выполнения этого события равен нулю.

Для практического вычисления величин (5.10), как и в предыдущем случае, нет надобности в предварительном определении соответствующих множеств путей P_i^-. Достаточно заметить, что интересующие нас величины (5.10) связаны между собой следующими соотношениями:

$$t_i^{oo}=\min_{s\in N_i^-}\left[t_{j_s}^{oo}-\tau_s\right],\ i=\overline{1,n-1}, \qquad (5.13)$$

где под N_i^- понимается множество дуг с началом в узле *i.*

Работу *s* сетевого графика Г называют *напряженной*, если события i_s и j_s являются критическими, и при этом $t_{j_s}^o=t_{i_s}^o+\tau_s$, или, что то же самое, $t_{i_s}^{oo}=t_{j_s}^{oo}-\tau_s$. Ясно, что для определения всех критических событий и всех напряженных дуг достаточно вычислить величины (5.7) и (5.10).

❖***Пример 5.2.*** Для сетевого графика Г, который уже рассматривался в связи с определением величин t_i^o (см. пример 5.1), найдем величины t_i^{oo}, ρ_i, $i=\overline{1,8}$, а также все критические события и напряженные работы. Исходные данные примера см. в табл. 5.1. Кроме того, во второй строке табл. 5.2 приведены величины t_i^o, $i=\overline{1,8}$, найденные при решении примера 5.1. Интересующие нас величины t_i^{oo}, $i=\overline{1,8}$ также определяются за $l<l_0+1$ просмотров дуг $s=\overline{1,14}$. Предположим, что уже имеются некоторые величины $t_i \geq t_i^o$, $i=\overline{1,8}$, причем $t_s^{oo}=t_s^o=42$. Тогда при просмотре очередной дуги s новое значение t_{i_s} определяется по формуле

$$t'_{i_s}=\min\{t_{i_s}, t_{j_s}-\tau_s\}.$$

Получаемые при этом величины t'_{i_s}, очевидно, также не меньше $t_{i_s}^{oo}$.

Далее, если при очередном просмотре всех дуг $s=\overline{1,14}$ не происходит ни одного изменения величин t_i, то это означает, что имеющиеся t_i совпадают с искомыми величинами (5.10). В качестве исходных принимаем $t_i=t_s^o=42$, $i=\overline{1,8}$. Затем за несколько просмотров дуг $s=\overline{1,14}$ находим величины t_i^{oo}, которые приведены в третьей строке табл. 5.3. В последнюю строку таблицы заносим величины ρ_i, подсчитанные по формулам (5.12). Так как $\rho_1=\rho_3=\rho_4=\rho_5=\rho_8=0$, то соответствующие события являются критическими. Что касается напряженных работ, то таковыми являются работы $s\in\{3, 6, 9, 11)$. На рис. 5.1 соответствующие дуги отмечены жирными линиями.

Таблица 5.3.

Результаты вычислений, пример 5.2

i	1	2	3	4	5	6	7	8
t_i^o	0	19	12	2	32	24	16	42
t_i^{oo}	0	27	12	2	32	37	30	42
ρ_i	0	8	0	0	0	13	14	0

🕮 **Лемма**. Путь $p = \{s_1, s_2, ..., s_l\}$ из $i_o=1$ в $i_{oo}=n$ в том и только том случае является критическим, когда соответствующим узлам отвечают критические события, а дугам – напряженные работы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Приведите примеры практического использования методов сетевого планирования.
2. Как задается с помощью орграфа зависимость между отдельными работами?
3. Что означают веса ребер в сетевом графике?
4. Можно ли использовать неориентированный граф для описания комплекса работ?
5. Верно ли, что в любом сетевом орграфе содержится контур или цикл?
6. Может ли в сетевом орграфе содержаться контур, почему?
7. Чему соответствует начальная вершина сетевого орграфа?
8. Чему соответствует конечная вершина сетевого орграфа?
9. Может ли в сетевом орграфе быть несколько начальных вершин?
10. Может ли в сетевом орграфе быть несколько конечных вершин?
11. Чем определяется календарный план выполнения сетевого графика?
12. Какие условия должны быть выполнены для допустимого плана выполнения сетевого графика?
13. Сформулируйте целевую функцию задачи о кратчайшем сроке.
14. Сформулируйте целевую функцию задачи о критическом пути.
15. Верно ли, что задача о критическом пути является частным случаем сетевой транспортной задачи?
16. Что такое напряженные работы?
17. Используются ли данные, полученные при решении задачи сетевого планирования о кратчайшем сроке, для решения задачи о критическом пути? Если да, то какие?
18. Верно ли, что в критическом пути присутствуют все дуги сетевого графика, у которых оба узла являются критическими событиями?

6. ПОТОКИ В СЕТЯХ

6.1. Максимальные потоки. Теорема Форда и Фалкерсона

Дана сеть, дуги которой ориентированы. В сети должны быть два специальных узла, называемых *источник* (*исток*) V_0 и *слив* (*сток*) V_n. V_0 – узел, у которого инвалентность равна нулю, а V_n – узел, у которого аутвалентность равна нулю. Каждой дуге сети приписывается некоторое число. Это число является не длиной дуги, а скорее ее шириной.

Рассмотрим сети, которые представляют собой поток «товара» через серию каналов («трубопроводов»). «Товар» может фактически (но не только) представлять собой жидкость, текущую через сеть труб. Дуги просто представляют собой части сети транспортирования для специфического товара, и их веса представляют собой максимальные пропускные мощности (возможности). Это могут быть, например, рельсы, дорожные или воздушные линии связи, или даже электрические или волокнисто-оптические кабели, если транспортируемый «товар» представляет собой цифровые сигналы. Таким образом, если сеть – это модель железной дороги, а узлы представляют железнодорожные станции, то число, приписанное дуге, может быть равно количеству путей между двумя станциями.

Другой пример, если сеть моделирует трубопровод, а узлы являются сочленениями, то это число может представлять площадь сечения трубы между двумя сочленениями. Оно определяет наибольшее количество жидкости, которое может пройти через дугу. Это число называют *пропускной способностью дуги.* Пропускную способность дуги e_{ij} обозначим через b_{ij}. Предполагается, что $b_{ij} > 0$ при всех i, j.

Задача ***о максимальном потоке в сети*** состоит в том, чтобы определить наибольшую величину потока, который может пройти из источника в слив.

Если D – сеть, которая имеет несколько источников и/или сливов (рис. 6.1, *а*), можно определить сеть N, добавляя два узла V_0 и V_n. к D, соединяя V_0 с помощью направленной дуги с каждым из источников и соединяя каждый слив с помощью направленных дуг с узлом V_n (рис. 6.6, *б*). N имеет единственный источник и единственный слив.

Веса, которые должны соответствовать этим дополнительным дугам, зависят от того, что собой представляет данная сеть. В случае планирования проблем, дополнительным дугам был бы присвоен вес 0, так, чтобы время на завершение всего проекта не было бы искусственно увеличено. Если D сеть для определения максимального потока, то дугам, исходящим из источника, присваивают значения пропускных способностей не менее чем суммарная пропускная способность выходящих дуг из следующего звена, а дополнительным дугам, входящим в слив, – сумму пропускных способностей входящих дуг в предыдущий узел (рис. 6.1).

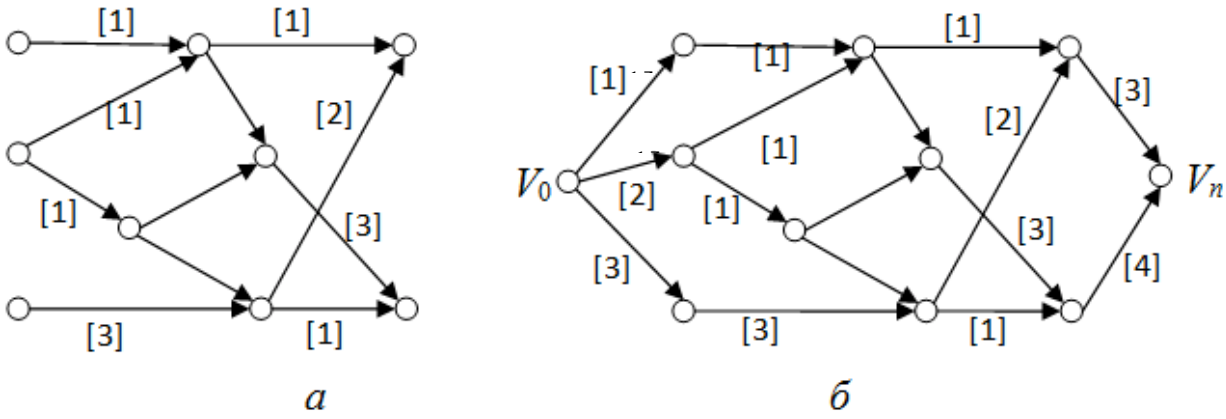


Рис. 6.1. *а* – Сеть *D*; *б* – сеть *N* с одним источником и одним сливом

Поток в сети ведет себя не совсем так, как вода или другая жидкость. Обозначим через x_{ij} поток из V_i в V_j через дугу e_{ij}. Для потока имеется ограничение

$$0 \le x_{ij} \le b_{ij}. \qquad (6.1)$$

Кроме ограничения (6.1), потребуем, чтобы приток в каждый узел был равен оттоку из него; т. е. поток сохраняется в каждом узле (за исключением источника и слива):

$$\sum_i x_{ij} = \sum_k x_{jk} \text{ при всех } j \ne 0, n. \qquad (6.2)$$

Так как поток сохраняется в каждом узле, то отток в источнике должен быть равен притоку в сливе:

$$\sum x_{0i} = v = \sum x_{jn}, \qquad (6.3)$$

где v – *величина* (*цена*) *потока*, т. е. полный поток, приходящий на слив.

Набор значений x_{ij}, определенных для всех дуг сети и удовлетворяющих (6.1)–(6.3), называется *потоком в сети*. Поток наибольшей возможной величины называется *максимальным потоком*.

Заметим, что может существовать более одного максимального потока. Например, максимальная цена любого потока через сеть, которая представлена на рис. 6.2, *а*, равна 3. В квадратных скобках дано значение пропускной способности каждой дуги, рядом поток по этой дуге. Имеются различные пути, которыми этот результат может быть достигнут. Максимальная цена всех возможных потоков равна 3, но имеется несколько различных максимальных потоков, которые достигают этого максимума. Два различных максимальных потока показаны на рис. 6.2.

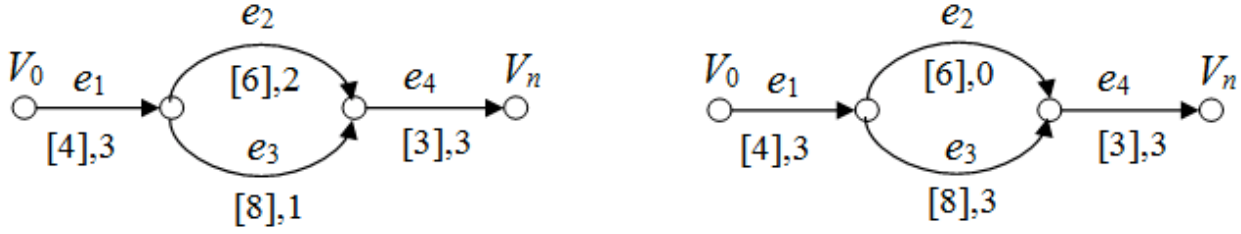


Рис. 6.2. Сеть с разными потоками и одинаковой ценой

Потоки в сетях широко используются во многих приложениях. Например, каково наименьшее число дуг, которые нужно удалить для разъединения двух заданных узлов графа? Этот вопрос является задачей теории графов, однако его можно решить, используя понятие потока.

Если сеть представляет собой цепочку V_0, V_1, V_2, …, V_k, V_n, то наибольшая величина потока, который может пройти из V_0 в V_n при выполнении (6.1) – (6.3), равна

$$\min\{b_{01}, b_{12}, b_{23}, ..., b_{kn}\}. \qquad (6.4)$$

Дуга с наименьшим значением пропускной способности является «*узким местом*» сети. Теперь определим понятие «узкого места» для произвольной сети. Узкое место сети называется *разрезом*.

Максимальная цена потока в сети тесно связана с идеей «разреза» сети, к описанию которой мы теперь переходим. Интуитивно ясно, что разрез можно рассматривать как множество дуг, которые, в случае их блокировки, полностью останавливают поток от источника к стоку. Однако если какая-либо из дуг не заблокирована, то поток может и проходить. Сеть на рис. 6.2 имеет три различных разреза: $\{e_1\}$, $\{e_2, e_3\}$ и $\{e_4\}$.

Разрезом сети N называется множество дуг, которое после удаления из N, порождает орграф с двумя компонентами, одна из которых содержит источник, а другая содержит слив.

И теперь приведем более формальное определение разреза.

🕮 **Определение.** *Разрез* – это совокупность всех дуг, идущих из некоторого подмножества узлов в его дополнение. Он обозначается через $(X, \overline{X})$, где X – заданное подмножество узлов, а $\overline{X}$ – его дополнение.

Таким образом, разрез (X, $\overline{X}$) – это множество всех таких дуг e_{ij}, что либо $V_i \in X$ и $V_j \in \overline{X}$, либо $V_j \in X$ и $V_i \in \overline{X}$. Удаление всех дуг из разреза разобьет сеть на две или более компоненты. Разрез, разделяющий узлы V_0 и V_n, – это такой разрез $(X, \overline{X})$, что $V_0 \in X$ и $V_n \in \overline{X}$.

На рис. 6.3 сеть изображена условно.

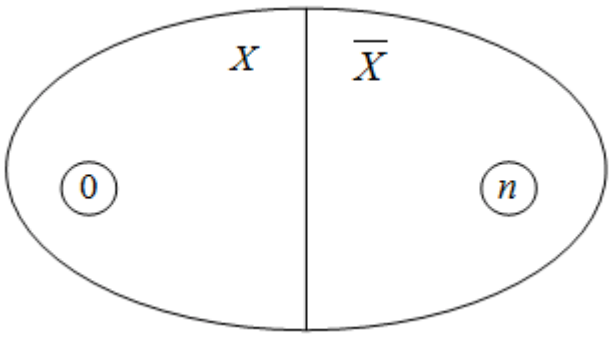


Рис. 6.3. Условное представление сети и разреза

Пропускной способностью, или *величиной (емкостью)* разреза $(X, \overline{X})$, называется сумма весов его дуг, которая определяется как $c(X, \overline{X}) = \sum_{i,j} b_{ij}$, где $V_i \in X$ и $V_j \in \overline{X}$.

Например, у сети, изображенной на рис. 6.2, разрез $\{e_1\}$ имеет емкость 4, разрез $\{e_2, e_3\}$ – 14, $\{e_4\}$ – 3.

Разрез является *минимальным***,** если его емкость меньше или равна емкости любого другого разреза.

✍ **Замечание.** В определении разреза учитываются все дуги между множествами X и $\overline{X}$, тогда как при вычислении его пропускной способности подсчитываются пропускные способности только дуг из X в $\overline{X}$, а дуги из $\overline{X}$ в X игнорируются. Поэтому в общем случае $c(X, \overline{X}) \neq c(\overline{X}, X)$.

Величина максимального потока всегда совпадает с минимальной пропускной способностью разреза, разделяющего V_0 и V_n. Разрез с наименьшей пропускной способностью, разделяющий V_0 и V_n, называется *минимальным разрезом.*

Факт совпадения величины максимального потока с пропускной способностью минимального разреза впервые был доказан А. Фордом и Д. Фалкерсоном в 1955 г. Это центральная теорема теории потоков в сетях.

🕮 **Теорема Форда и Фалкерсона (о максимальном потоке и минимальном разрезе).** В любой сети с целыми значениями пропускных способностей дуг величина максимального потока из источника в слив равна пропускной способности минимального разреза, разделяющего источник и слив.

Доказательство. Дадим конструктивное доказательство теоремы, которое строит максимальный поток и находит минимальный разрез. Доказательство показывает, что всегда существует поток, значение которого равно пропускной способности минимального разреза. Поскольку максимальный поток всегда не превосходит пропускной способности произвольного разреза, в частности, минимального разреза, то это и доказывает справедливость теоремы.

Если текущая величина потока равна пропускной способности некоторого разреза, то теорема доказана. Если величина потока не равна пропускной способности разреза, то ищем среди путей из V_0 в V_n такой, вдоль которого можно послать дополнительный поток. Это увеличивает величину потока. Продолжаем данную процедуру до тех пор, пока такого пути нельзя будет найти. Докажем, что величина потока равна пропускной способности некоторого разреза. Опишем систематический способ отыскания такого пути.

Начнем с произвольного множества величин X_{ij}, удовлетворяющих (6.1) и (6.2) (например, $X_{ij}=0$ при всех i и j). На основе текущего потока в сети рекурсивно определим подмножество узлов X при помощи следующих правил.

0. $V_0 \in X$.

1. Если $V_i \in X$ и $x_{ij} < b_{ij}$, то $V_j \in X$.

2. Если $V_i \in X$ и $x_{ji} > 0$, то $V_j \in X$.

Все узлы, не лежащие в X, принадлежат $\overline{X}$. Используя данные правила определения множества X, рассмотрим два возможных случая.

Случай 1. $V_n \in \overline{X}$. Следовательно, для всех дуг из X *в* $\overline{X}$ справедливо $x_{ij} = b_{ij}$ (в силу правила *1*), и не существует потока x_{ji} через дугу из $\overline{X}$ в X (в силу правила *2*). Поэтому

$$\sum_{\substack{i\in X \\ j\in\overline{X}}} x_{ij} = \sum_{\substack{i\in X \\ j\in\overline{X}}} b_{ij} \quad \text{и} \quad \sum_{\substack{i\in X \\ j\in\overline{X}}} x_{ji} = 0.$$

Следовательно, найден поток со значением, равным $c(X, \overline{X})$.

Случай 2. $V_n \in X$. Тогда существует путь из V_0 в V_n, образованный дугами, удовлетворяющими правилу *1* или правилу *2*. Обозначим этот путь $V_0,\ldots,V_i,e_{ij},V_j,\ldots,V_n$. Каждая дуга в данном пути должна удовлетворять либо правилу *1*, либо правилу *2*. Если дуга удовлетворяет правилу *1*, т. е. $x_{ij} < b_{ij}$, то можно отправить дополнительный поток из V_i в V_j. Это увеличивает величину потока вдоль дуги. Такой тип дуги называется *прямой дугой.*

Если же дуга удовлетворяет правилу *2*, т. е. $x_{ji} > 0$, то можно отправить поток из V_j, в V_i, фактически отменяя существующий поток вдоль дуги. Эффект состоит в сокращении величины потока вдоль дуги. Такой тип дуги называется *обратной дугой.* Данный путь по отношению к текущему потоку называется *увеличивающим путем.*

Например, рассмотрим сеть на рис. 6.4, где числа в квадратных скобках обозначают пропускные способности дуг. Если потоки на дугах равны $x_{01}=x_{12}=x_{2n}=1$, а все остальные $x_{ij}=0$, то e_{03}, e_{32}, e_{21}, e_{4n} является увеличивающим путем с обратной дугой e_{21}.

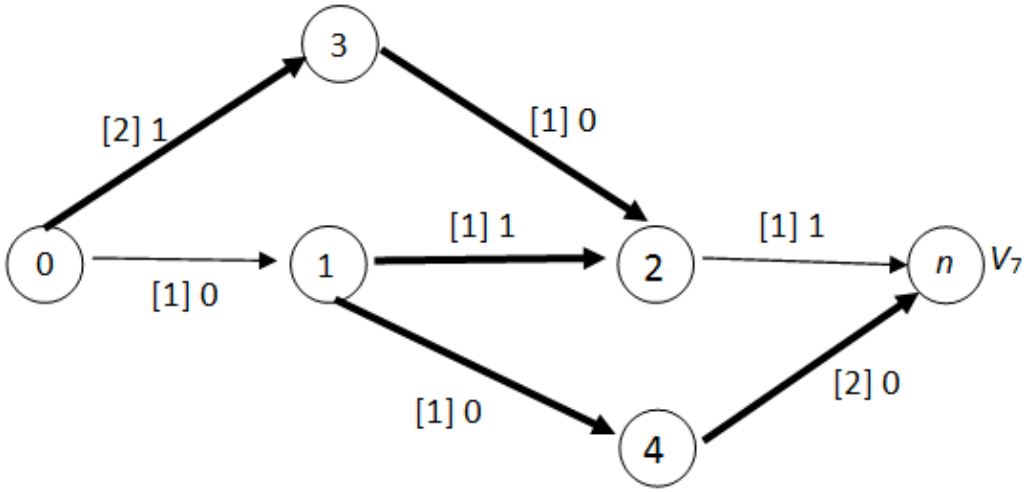


Рис. 6.4. Пример сети

Пусть ε_1 – минимум среди всех разностей $b_{ij} - x_{ij}$ в этом пути, ε_2 – минимум среди x_{ji} в нем, а $\varepsilon = \min\{\varepsilon_1, \varepsilon_2\}$ – целое положительное число. Тогда можно увеличить дуговые потоки на ε по всем прямым дугам пути и уменьшить дуговые потоки на ε по всем его обратным дугам. Таким образом, величина потока увеличивается на ε, и новые значения x_{ij} удовлетворяют всем ограничениям (6.1) и (6.2). Теперь на основе нового потока можно переопределить множество X. Если V_n все еще лежит в X, то снова увеличиваем величину потока на некоторое ε. Поскольку пропускная способность минимального разреза – конечное число, а величина потока увеличивается, по крайней мере, на единицу, то после конечного числа шагов будет получен максимальный поток. Теорема доказана.

Обозначим через F_{0n} набор неотрицательных целых чисел x_{ij}, удовлетворяющих (6.1)–(6.3).

Следствие. *Поток F_{0n} максимален тогда и только тогда, когда относительно F_{0n} не существует увеличивающего пути.*

6.2. Метод нахождения максимального потока. Теорема о максимальных разрезах

Величина максимального потока в любой сети определяется единственным образом, и может существовать несколько потоков, дающих наибольшую величину потока. Также в сети может существовать и несколько минимальных разрезов. Например, рассмотрим сеть на рис. 6.5.

Пусть пропускная способность каждой дуги равна единице. Тогда обе дуги e_{23} и e_{5n} являются минимальными разрезами. Величина максимального потока в точности равна 1, однако максимальным потоком может быть как $x_{01}=x_{12}=x_{23}=x_{34}=x_{45}=x_{5n}=1$, так и $x_{06}=x_{62}=x_{23}= =x_{37} = x_{75} =x_{5n}=1$.

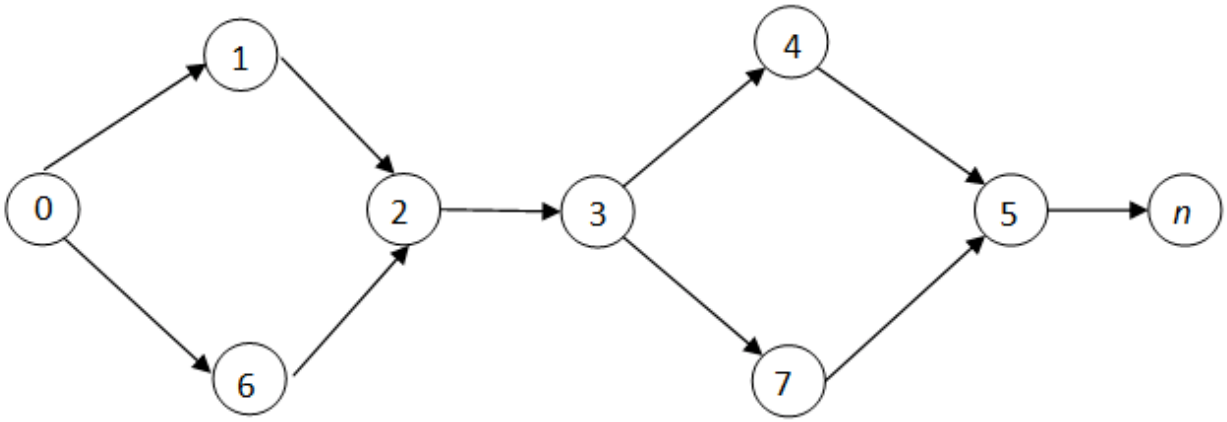


Рис. 6.5. Сеть с пропускными способностями равными единице

Метод обнаружения максимального потока. Он основан на доказательстве теоремы Форда и Фалкерсона. Основная идея состоит в том, чтобы начать с некоторого потока (относительно нетрудно найти допустимый поток), и если этот поток уже не является максимальным, улучшить его.

❖***Пример 6.1.*** Рассмотрим сеть, показанную на рис. 6.6, *а*, и предположим, что начинаем с потока, показанного на рис. 6.6, *б*, который имеет цену 15 и очевидно не является максимальным.

Чтобы улучшить поток, нам следует заменить его новым потоком, имеющим большую цену. Чтобы сделать это, прежде всего, посмотрим на направленный путь, идущий от источника к стоку и обладающий тем свойством, что поток в каждой дуге пути строго меньше его пропускной способности. Для данного пути вычислим минимальную цену $b_{ij} - x_{ij}$ для совокупности его дуг. Оказывается возможным увеличить поток в каждой дуге пути на вычисленные значения, задавая новый поток с ценой, превышающей первоначальную.

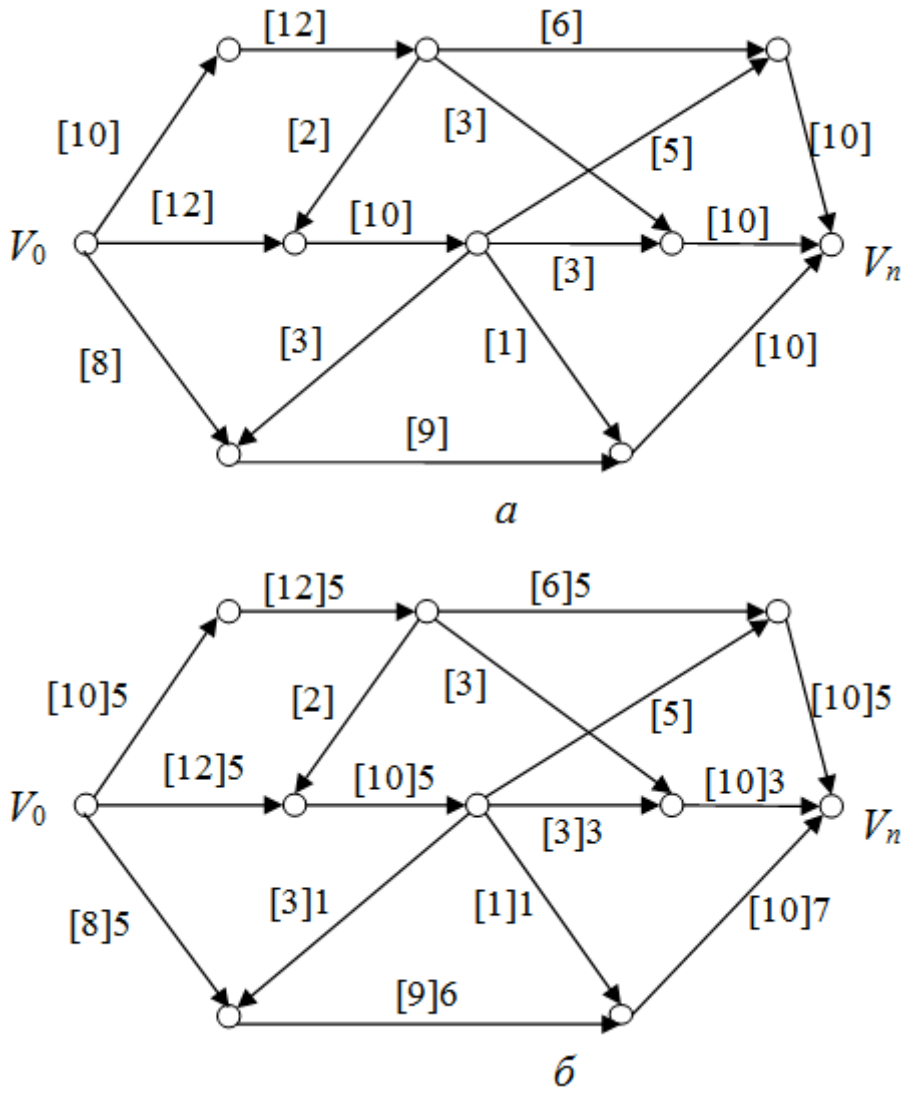


Рис. 6.6. Сеть: *а* – без потока; *б* – поток с ценой 15

Минимальная цена $b_{ij} - x_{ij} = 2 - 0 = 2$ для дуг пути, представленного на рис. 6.7, *а*, равна 2. Когда поток в каждой дуге этого пути будет увеличен на 2, получим поток с ценой 17 (рис. 6.7, *б*).

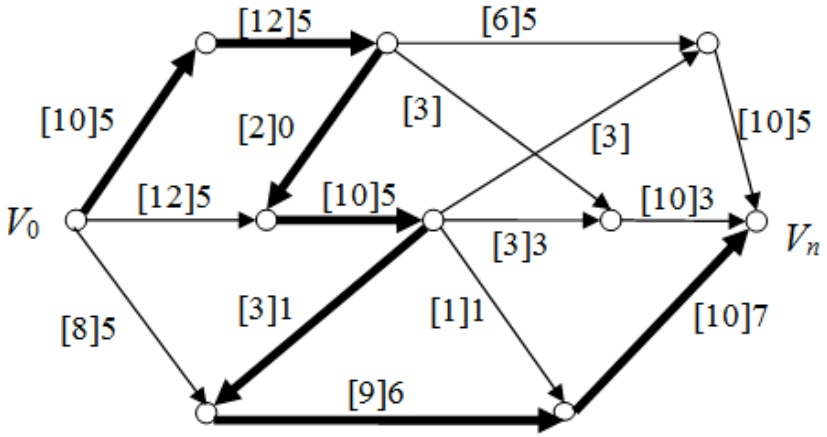
Результат нескольких повторений этого процесса производит поток с ценою 24, показанный на рис. 6.7, *в*. Для этого потока уже не существует ни одного направленного пути от источника к стоку, обладающего тем свойством, что для каждой дуги пути, поток, проходящий через дугу, оказывается строго меньшим емкости дуги. В связи с этим может показаться, что найденный поток является максимальным. Однако это не так, но дальнейшее улучшение потока требует некоторых ухищрений.

Рассмотрим путь, показанный на рис. 6.8, *а*. Это не направленный путь от источника к стоку; строго говоря, мы должны расценить его как путь в основном ненаправленном графе. Поток в каждой из трех дуг, направленных «вперед», строго меньше их емкости, и поток в дуге, направленной «назад», положителен. Если увеличивать поток в каждой из дуг с потоком «вперед» на 1

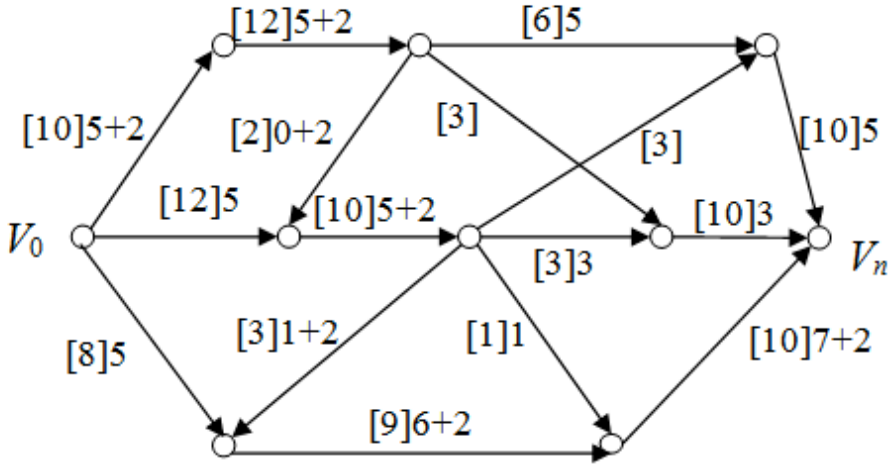
(на минимальную цену $b_{ij} - x_{ij}$ из совокупности цен для дуг, направленных «вперед») и уменьшать на 1 поток в дуге, направленном «назад», то в результате поток все еще останется консервативным, и при этом его цена будет увеличена на 1. Результирующий поток с ценой 25 показан на рис. 6.8, *б*.

Других ненаправленных путей рассмотренного типа от источника к стоку в данном примере нет. (Т. е. нет больше таких путей, у которых поток через все дуги, направленные вперед, может быть увеличен без превышения их емкости, а поток через все дуги, направленные назад, может быть уменьшен, не становясь отрицательным.) Это означает, что поток, показанный на рис. 6.7, *б*, является действительно максимальным.

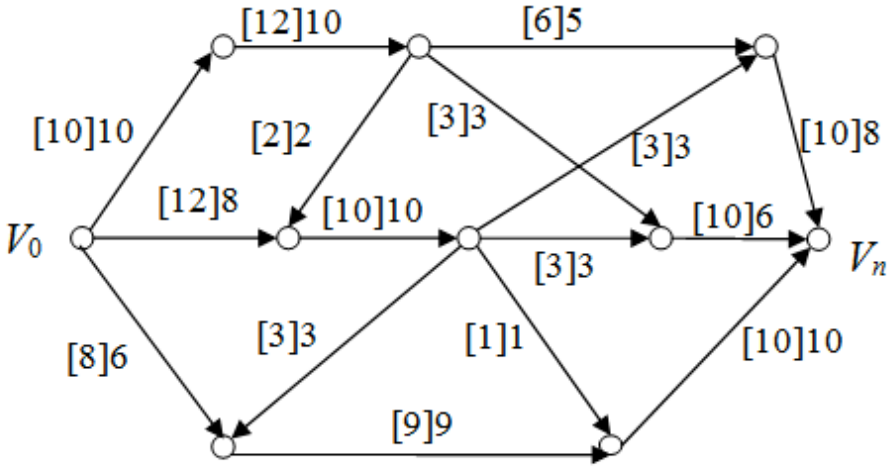
В рассмотренном примере довольно легко увидеть, что никаких других путей от источника к стоку, которые позволяют дополнительно увеличить поток, не существует. Однако в случае более большой или более сложной сети может составить трудность определение того, что таких путей, допускающих больший поток, нет. В этом случае теорема о максимальном потоке и минимальном срезе оказывается очень полезной. Так как срез – это множество таких дуг, что их блокировка полностью останавливает поток, то цена любого потока не может превысить емкость любого среза. Следовательно, цена потока может равняться емкости среза только в том случае, когда поток максимален, а срез минимален.



а



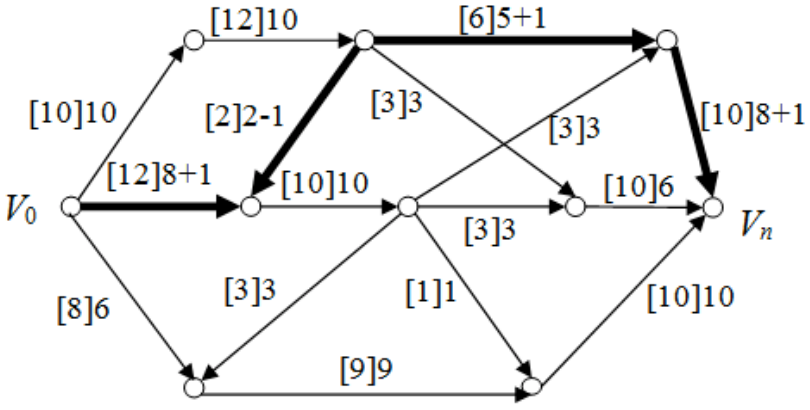
б



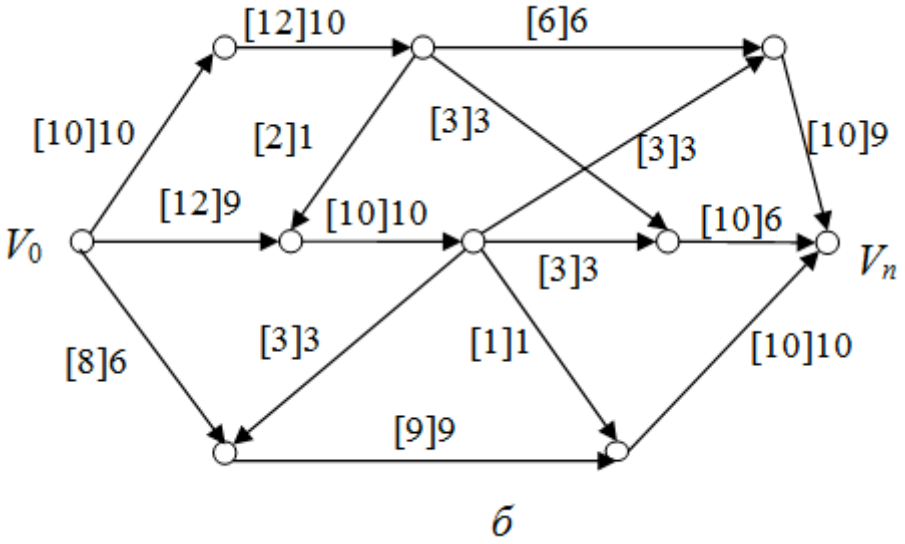
в

Рис. 6.7. Сеть: *а* – увеличивающий путь; *б* – увеличение потока на 2, вдоль увеличивающего пути; *в* – поток с ценой 24

Если можно найти срез со значением емкости, равным 25, – значению, совпадающему с ценой потока, то из этого факта можно сделать заключение, что такой поток действительно является максимальным (и, конечно, срез является минимальным). Такие разрезы показаны на рис. 6.9.



а



б

Рис. 6.8. Сеть: *а* – увеличивающий путь с обратной дугой;
б – результирующий поток с ценой 25

Таким образом, теорема максимального потока и минимального разреза гарантирует, что поток, представленный на рис. 6.8, *б*, является максимальным. При этом данный поток не является единственно возможным.

Для определения разреза удобно воспользоваться рекурсивными правилами, рассмотренными при доказательстве теоремы Форда и Фалкерсона. Так, для потока примера 6.1 получим:

0. $V_0 \in X$.

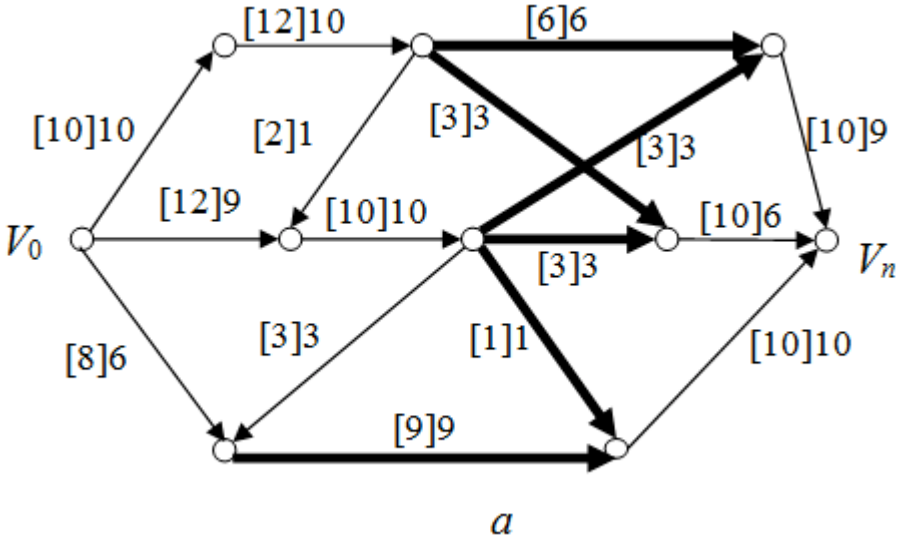
1. $V_2, V_3 \in X$.

2. $V_4, V_5 \in X$.

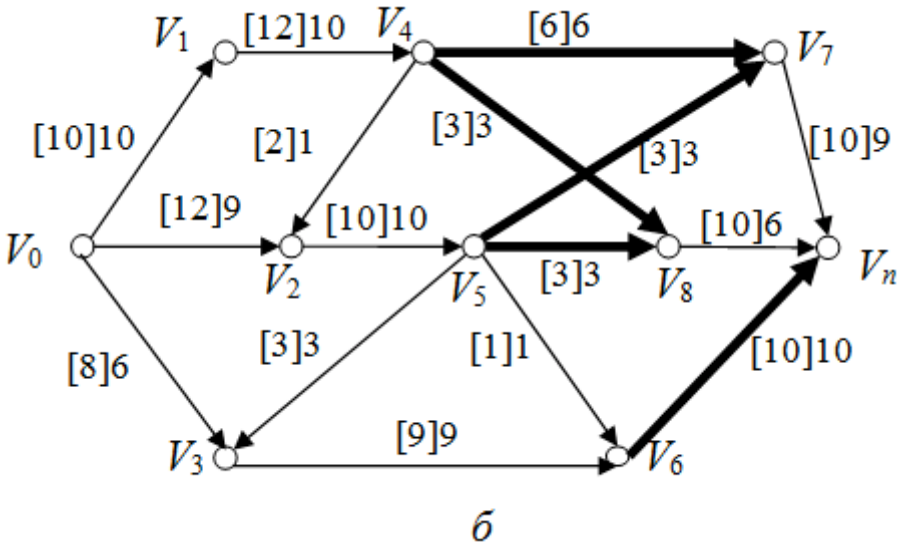
1. –.

2. $V_1 \in X$.

Получаем $X = \{0,1,2,3,4,5\}$, $\overline{X} = \{6,7,8,n\}$.



а



б

Рис. 6.9. Минимальные разрезы

Пусть $(X, \overline{X})$ и $(Y, \overline{Y})$ – два разреза. Будем говорить, что эти два разреза *скрещиваются,* если каждое из множеств $X \cap Y$, $X \cap \overline{Y}$, $\overline{X} \cap Y$, $\overline{X} \cap \overline{Y}$ не пусто.

🕮 **Теорема 2.** *Пусть* $(X, \overline{X})$ и $(Y, \overline{Y})$ – *минимальные разрезы. Тогда* $(X \cup Y, \overline{X \cup Y})$ *и* $(X \cap Y, \overline{X \cap Y})$ *также являются минимальными разрезами.*

Доказательство. Если $X \subset Y$, то $X \cup Y = Y$, $X \cap Y = X$, следовательно,

$$(X \cup Y, \overline{X \cup Y}) = (Y, \overline{Y}),\ (X \cap Y, \overline{X \cap Y}) = (X, \overline{X}).$$

Рассмотрим случай, когда $X \not\subset Y$ и $Y \not\subset X$, как показано на рис. 6.10. Таким образом, данные два разреза скрещиваются. Введем четыре различных множества: $Q = X \cap Y$, $S = X \cap \overline{Y}$, $P = \overline{X} \cap Y$, $R = \overline{X} \cap \overline{Y}$, где $V_0 \in Q$, а $V_n \in R$.

Заметим, что $X = Q \cup S$, $\overline{X} = P \cup R$, $Y = P \cup Q$, $\overline{Y} = R \cup S$.

Пусть $c(P, Q) = \sum b_{ij}$ по всем $i \in P$, и аналогично для остальных множеств.

Так как $(X, \overline{X})$ и $(Y, \overline{Y})$ – минимальные разрезы, а $(P \cup Q \cup S, R)$ и $(Q, P \cup R \cup S)$ – разрезы, разделяющие V_0 и V_n, то должно выполняться:

$$c(X, \overline{X}) + c(Y, \overline{Y}) \le c(P \cup Q \cup S, R) + c(Q, P \cup R \cup S), \quad (6.5)$$

или

$$c(Q,P) + c(Q,R) + c(S,P) + c(S,R) + c(P,R) + c(P,S) + \\ + c(Q,R) + c(Q,S) \le c(P,R) + c(Q,R) + c(S,R) +$$

$$+ c(Q,P) + c(Q,R) + c(Q,S), \tag{6.6}$$

или

$$c(P,S) + c(S,P) \le 0. \tag{6.7}$$

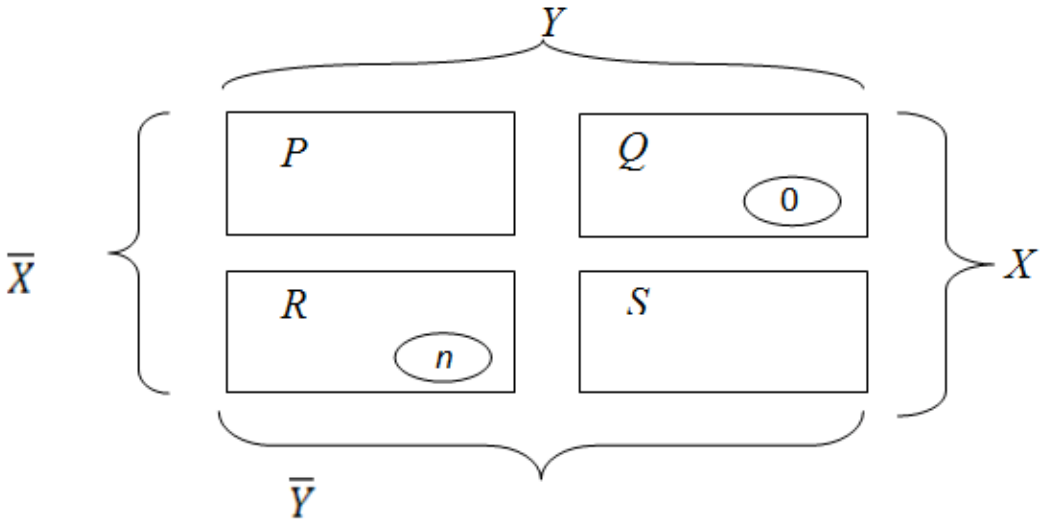


Рис. 6.10. Условное представление скрещивания разрезов

Поскольку величина $c(P,S) + c(S,P)$ должна быть неотрицательной, то видно, что (6.5) должно обращаться в равенство, т. е.

$$c(X,\overline{X}) + c(Y,\overline{Y}) = c(P \cup Q \cup S, R) + c(Q, P \cup R \cup S),$$

или

$$c(X,\overline{X}) + c(Y,\overline{Y}) = c(X \cup Y, \overline{X \cup Y}) + c(X \cap Y, \overline{X \cap Y}). \tag{6.8}$$

Так как ни одно из слагаемых правой части (6.8) не может быть строго меньше пропускной способности минимального разреза, то каждое из них должно ей равняться.

Теорема доказана.

В общем случае $c(P,S) \neq c(S,P)$, но обе величины неотрицательны. Теорема 2 устанавливает, что если имеются два скрещивающихся минимальных разреза, разделяющих V_0 и V_n, то существуют два других не скрещивающихся минимальных разреза.

6.3. Алгоритмы для нахождения максимального потока и минимального разреза

Алгоритм Форда-Фалкерсона – это систематический способ отыскания увеличивающего пути. Алгоритм называется *методом расстановки пометок* и может быть легко применен, если входные данные заданы при помощи списков смежности.

Алгоритм Форда-Фалкерсона состоит из двух шагов. На первом шаге присваиваются узлам метки для поиска увеличивающего пути. На втором шаге

увеличивается поток вдоль увеличивающего пути, найденного на первом шаге. Начинаем с нулевого значения потока (т. е. $x_{ij}=0$ при всех i, j) и повторяем шаги 1 и 2 до тех пор, пока поток не станет максимальным.

Шаг 1. Процесс расстановки пометок. Каждый узел всегда находится в одном из следующих трех состояний.

(1) Непомеченный: узел еще не получил пометки. Вначале каждый узел является непомеченным.

(2) Помеченный и неотсканированный: узел помечен, но не все его соседи являются помеченными.

(3) Помеченный и отсканированный: узел помечен, и все его соседи также помечены.

Узел V_j получает пометку $[a, c]$, состоящую из двух чисел. Число a обозначает наибольшую суммарную величину потока, который можно передать из V_0 в V_j. Число c указывает последний промежуточный узел увеличивающего пути из V_0 в V_j. (Заметим сходство этих пометок и тех, которые использовались при решении задач о кратчайших путях в главе 3, п. 3.1.)

В начале алгоритма присваиваем узлу V_0 пометку $[\infty, 0]$, а каждому соседнему с ним узлу V_j присваиваем пометку $[b_{0i}, 0]$. Пусть у помеченного узла V_i есть еще не помеченный сосед V_j. Тогда узлу V_j присваиваем пометку $[\varepsilon(j), i]$, где $\varepsilon(j) = \min\{b_{0i}, b_{ij}\}$.

В общем случае дополнительный поток из V_i в V_j можно отправить, если $x_{ij} < b_{ij}$ или $x_{ji} > 0$.

Предположим, что V_j имеет пометку $[\varepsilon(i), r^+]$, тогда V_j получает пометку $[\varepsilon(j), i^+]$, если $x_{ij} < b_{ij}$, где $\varepsilon(j) = \min\{\varepsilon(i), b_{ij} - x_{ij}\} > 0$.

Узел V_j получает пометку $[\varepsilon(j), i^-]$, если $x_{ji}>0$, где $\varepsilon(j) = \min\{\varepsilon(i), x_{ji}\} > 0$.

Оба символа i^+ и i^- указывают, что V_i – последний промежуточный узел, при этом «+» означает, что e_{ij} – прямая дуга, а «–» означает, что e_{ij} – обратная дуга. Заметим, что узел V_j получает пометку только в случае, когда $\varepsilon(j)$ строго положительно.

Сначала узел V_0 помечен и неотсканирован. Он станет помеченным и отсканированным тогда, когда будут помечены все его соседи. Соседи V_0 также становятся помеченными и отсканированными, когда будут помечены все их соседи. Процесс расстановки пометок продолжается до тех пор, пока не выполнится одно из двух условий:

(1) узел V_n помечен.

(2) узел V_n не помечен и больше нельзя присвоить никаких новых пометок.

В случае (2) все помеченные узлы (отсканированные и неотсканированные) составляют множество X, а непомеченные узлы –

множество $\overline{X}$. Разрез $(X, \overline{X})$ является минимальным, а текущий поток – максимальным. Алгоритм завершает работу.

Шаг 2. Изменение потока: если узел V_n помечен, то можно проследовать обратно из V_n в V_0 и изменить потоки через дуги вдоль увеличивающего пути. Пусть $[\varepsilon(n), k^+]$ – пометка узла V_n, $[\varepsilon(k), j^-]$, – пометка узла V_k, а $[\varepsilon(j), 0^+]$ – пометка узла V_j. Тогда добавляем $\varepsilon(n)$ к x_{kn} и x_{0j} и вычитаем $\varepsilon(n)$ из x_{kj}, т. е. добавляем величину $\varepsilon(n)$ к потокам для прямых дуг и вычитаем $\varepsilon(n)$ из потоков для обратных дуг.

Стираем все метки и возвращаемся на шаг 1.

Алгоритм Форда-Фалкерсона не определяет порядок, в котором происходит помечивание узлов или проверка помеченных и неотсканированных узлов. Если следовать правилу «первый помечен – первый отсканирован», то всегда будет использован кратчайший увеличивающий путь (кратчайший в том смысле, что число дуг в пути наименьшее). Это модификация Эдмондса и Карпа. Правило «первый помечен-первый отсканирован» – это поиск в ширину.

После построения увеличивающего пути можно построить *остаточную сеть*, порожденную существующим потоком. Пропускные способности дуг остаточной сети b'_{ij} определяются по следующему правилу:

– если дуга в увеличивающем пути прямая, то $b'_{ij} = b_{ij} - x_{ij}$;

– если дуга обратная – $b'_{ij} = b_{ij} + x_{ij}$.

Дуга из V_i в V_j называется *полезной,* если b'_{ij}=0.

Коротко алгортим Форда-Фалкерсона можно описать следующим образом.

1. На основе существующего потока строится остаточная сеть с пропускными способностями b'_{ij}.

2. В остаточной сети производим систематический поиск увеличивающего пути и увеличиваем поток вдоль увеличивающего пути.

Алгоритм Эдмондсома и Карпа. Если использовать поиск в ширину, предложенный Эдмондсом и Карпом, то поток всегда будет увеличиваться вдоль кратчайшего увеличивающего пути. Длиной пути называем число дуг в нем. Вначале используется увеличивающий путь длины 1. Если не существует увеличивающих путей длины 1, используется увеличивающий путь длины 2. Эти действия повторяются до тех пор, пока длина кратчайшего увеличивающего пути не станет равной n–1.

Разобьем вычисления на n–1 этапов. На каждом этапе происходит увеличение потоков вдоль увеличивающих путей заданной длины k (k=1,2,...,n–1). От этапа к этапу длина кратчайшего увеличивающего пути возрастает. В конце каждого этапа строится остаточная сеть, порожденная

существующим потоком. В начале каждого этапа имеется остаточная сеть с пропускными способностями дуг, равными b'_{ij}.

Разобьем все узлы сети на слои. По определению, источник V_0 лежит в нулевом слое. Узел V_i лежит в слое k, если кратчайший путь из V_0 в V_i имеет длину k.

На рис. 6.11 показана сеть, разбитая на четыре слоя. Заметим, что узел V_n лежит в слое 3. Следовательно, длина кратчайшего увеличивающего пути равна 3. Поток в сети, относительно которого нет увеличивающих путей длины 3, называется *тупиковым.* Например, если положить на рис. 6.11 пропускные способности всех дуг равными 1, то можно отправить единицу потока вдоль пути
(e_{0i}, e_{14}, e_{4n}). Тогда не существует увеличивающего пути длины 3, откуда заключаем, что текущий поток (величина которого равна 1) является тупиковым, хотя величина максимального потока в этой сети равна 2.

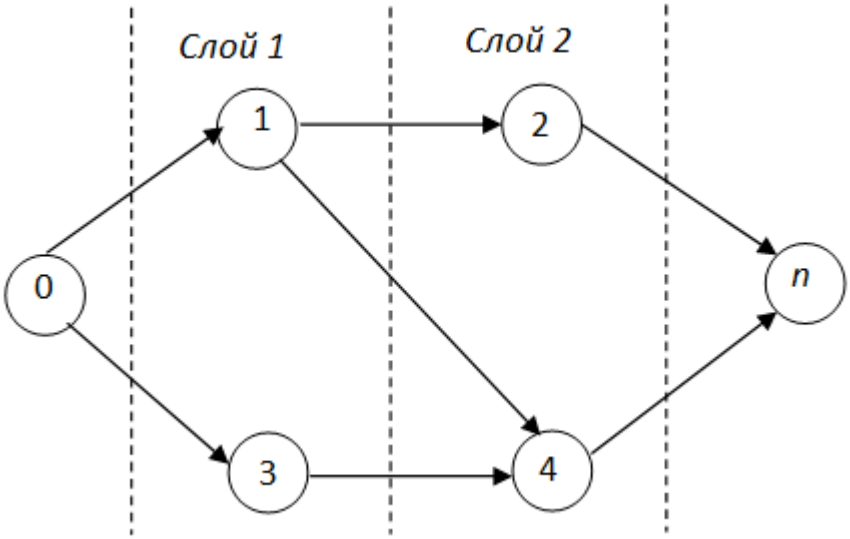


Рис. 6.11. Разделение сети на слои

Этап алгоритма:

1) Найти остаточную сеть.

2) Разбить множество узлов на слои.

3) Найти тупиковый поток.

Алгоритм состоит из последовательности таких этапов с ростом длины кратчайших увеличивающих путей.

Если длина увеличивающего пути от этапа к этапу возрастает, то число этих этапов не превосходит n–1. Предположим, что имеется m дуг, во время этапа существует не более m увеличивающих путей, и на поиск каждого такого пути требуется не более, чем $O(m)$ шагов. Следовательно, итоговая трудоемкость поиска составляет $O(m^2)$ во время этапа и $O(nm^2)$ для всего алгоритма Эдмондса и Карпа.

Оценка $O(nm^2)$ получается в предположении, что каждый увеличивающий путь в сети ищется независимо. Можно улучшить эту оценку до величины

$O(n^2m)$, если на каждом этапе скоординировать поиск увеличивающих путей заданной длины. Эта улучшенная оценка принадлежит Диницу.

Алгоритм Диница. В начале, используя поиск в ширину, строим подсеть $N(f)$. После этого, применяя поиск в глубину, в $N(f)$ находим увеличивающие пути, достигающие V_n. Если сток V_n достигнут, то увеличиваем поток вдоль найденного пути, корректируем пропускные способности дуг, проходя вдоль увеличивающего пути в обратном направлении, и удаляем дуги, пропускные способности которых стали равны нулю.

Если в процессе поиска в глубину при отыскании увеличивающего пути делается «шаг назад» из узла V_i, то все дуги, инцидентные узлу V_i, вычеркиваем из $N(f)$. Поиск в глубину затрачивает время $O(n)$ и, поскольку в течение этапа имеется не более m увеличивающих путей, итоговая трудоемкость алгоритма составляет $O(nm)$ для каждого этапа и $O(n^2m)$ в целом.

Заметим, что трудоемкость нахождения увеличивающего пути поиском в глубину не превосходит $O(n)$, и при каждом поиске удаляется, по крайней мере, одна дуга. Алгоритм Диница на каждом этапе при построении тупикового потока насыщает по одной дуге.

Алгоритм Карзанова состоит из нескольких этапов, на каждом из которых ищется тупиковый поток в многослойной сети.

При поиске тупикового потока каждый раз насыщается один узел. Каждый этап начинается с проталкивания предпотока из V_0, а затем из слоя в слой до тех пор, пока не будет достигнут узел V_n. Затем происходит балансировка предпотока в каждом узле. Таким образом, каждый этап состоит из двух шагов. Первый шаг называется продвижением предпотока, а второй – балансировкой предпотока. Эти шаги повторяются до получения тупикового потока в остаточной подсети $N(f)$.

Шаг 1. Продвижение предпотока. Цель этой подпрограммы – протолкнуть предпоток наибольшей возможной величины из источника в сток.

Начинаем процесс в источнике V_0 (в слое 0) и передаем предпоток с наибольшим возможным значением в узлы 1-го слоя, полагая $x_{0j}=b_{0j}$ для всех узлов V_j из 1-го слоя. Считаем, что предпоток передан из слоя 0 в слой 1.

Рассматриваем несбалансированные узлы из самого нижнего слоя и пытаемся протолкнуть предпоток в следующий слой и далее до V_n. Подпрограмма продвижения останавливается тогда, когда никакой предпоток невозможно передать вперед из любого узла V_j. Это обязательно произойдет, если дуги в $0(V_j)$ либо закрыты, либо насыщены.

Шаг 2. Балансировка предпотока. Цель этой подпрограммы – сделать несбалансированные узлы сбалансированными и эффективно преобразовать предпотоки в регулярные потоки.

Начинаем с наивысшего слоя, содержащего несбалансированные узлы. Для каждого такого узла V_j уменьшаем потоки, входящие в него, по принципу «вошел последним – вышел первым» до тех пор, пока узел не станет

сбалансированным. Все дуги в $a(V_j)$, относящиеся к вновь сбалансированному узлу, объявляются закрытыми. Если в данном слое все несбалансированные узлы стали сбалансированными, то возвращаемся на шаг 1 (даже если остались несбалансированные узлы в низших слоях).

Заметим, что при первом выполнении шага 1 происходит продвижение предпотока из слоя 0 в слой 1, из слоя 1 в слой 2 и т.д. до тех пор, пока не будет достигнут узел V_n, после этого переходим к шагу 2. Однако в подпрограмме балансировки рассматривается только один слой – наивысший слой j, содержащий несбалансированные узлы. При балансировке узлов слоя j могут появиться несбалансированные узлы в слое j–1. Поскольку все узлы слоев j–2,...,0 уже обработаны, можно начать шаг 1 из слоя j–1 и продвигать наибольший возможный предпоток в сток n.

В этом алгоритме есть шаги балансировки и шаги продвижения. На шаге балансировки, если поток через дугу уменьшается, то эта дуга становится закрытой, следовательно, суммарное число уменьшений потока ограничено числом дуг $O(m)$. Когда поток через дугу возрастает, она либо насыщенная, либо ненасыщенная. Но насыщение может произойти для каждой дуги не более одного раза, так как после любого уменьшения потока эта дуга станет закрытой. Следовательно, число насыщений также ограничено величиной $O(m)$. Ситуация, при которой поток через дугу возрастает, но не до предельно возможного значения, может наблюдаться не более n–1 раз при начальном продвижении предпотока и не более n–2 раз при следующем продвижении, поскольку некоторый узел балансировался при начальном продвижении предпотока и, следовательно, недоступен. Продолжая эти рассуждения, видим, что число шагов, увеличивающих поток (но не до предельно возможного значения), равно $(n–1)+(n–2)+1=O(n^2)$. Трудоемкость алгоритма Карзанова построения тупиковых потоков в многослойной сети ограничена величиной $O(n^2)$.

Следовательно, трудоемкость алгоритма ограничена величиной $O(m)+O(n^2)=O(n^2)$. Поскольку число этапов равно n, суммарное время работы алгоритма Карзанова составляет $O(n^3)$.

Итак, на каждом этапе алгоритм Карзанова затрачивает $O(n^2)$ единиц времени. Так как число этапов не превосходит n–1, то общая трудоемкость составляет $O(n^3)$.

6.4. Потоки с минимальной стоимостью. Метод анализа и оценки проекта REPT-метод

Зададим стоимость c_{ij} передачи единицы потока через каждую дугу e_{ij}. Можно поставить две задачи:

1. Какова минимальная стоимость передачи заданной величины потока из V_0 в V_n?

2. Какова максимальная величина потока, который можно передать из V_0 в V_n при заданном фиксированном бюджете?

Задачу 1 формально можно представить в виде:

$z = \sum c_{ij} x_{ij} \to \min$,

при условиях $\sum_i x_{ij} - \sum_k x_{jk} = \begin{cases} -v, j = 0; \\ 0, j \neq 0, n; \\ v, j = n; \end{cases}$

и $0 \le x_{ij} \le b_{ij}$, где v – требуемая величина потока.

Задачу 2 можно записать в виде:

$$v \to \max,$$

при условиях $\sum c_{ij} x_{ij} \le con$, где con – заданная константа,

$$\sum_i x_{ij} - \sum_k x_{jk} = \begin{cases} -v, j = 0; \\ 0, j \neq 0, n; \ 0 \le x_{ij} \le b_{ij}. \\ v, j = n; \end{cases}$$

Возможны случаи:

1) Если ограничения на пропускные способности дуг отсутствуют, то величины c_{ij} можно считать длинами дуг и найти самый дешевый (кратчайший) путь из V_0 в V_n, а затем отправить вдоль него требуемую величину потока.

2) При наложении ограничений на пропускные способности дуг метод передачи потока вдоль самого дешевого пути будет работать, если минимальная пропускная способность этого пути больше, чем v.

3) Если минимальная пропускная способность $b_{ij} < v$, то эта дуга будет насыщена, и дальнейшее продвижение вдоль этого пути станет невозможным. Это эквивалентно тому, что стоимость дуги становится бесконечной.

Отсюда возникает идея «модифицированной стоимости», когда стоимость зависит от потока, протекающего через дугу.

Схема алгоритма решения задачи 1.

Шаг 0. Начать с нулевых значений потоков на всех дугах.

Шаг 1. Определить модифицированные стоимости c^*_{ij}, порожденные существующими потоками, следующим образом:

$c^*_{ij} = c_{ij}$, если $0 \le x_{ij} \le b_{ij}$,

$c^*_{ij} = \infty$, если $x_{ij} = b_{ij}$,

$c^*_{ij} = -c_{ij}$, если $x_{ji} > 0$.

Шаг 2. Найти самый дешевый путь при модифицированных стоимостях c^*_{ij}, полученных на шаге 1, и передать вдоль него ε единиц потока, где

$\varepsilon = \min\{\varepsilon_1, \varepsilon_2\}$,

$\varepsilon_1 = \min$ среди величин $(b_{ij} - x_{ij})$ по всем прямым дугам,

$\varepsilon_2 = \min$ среди величин (x_{ji}) по всем обратным дугам.

Скорректировать текущую величину потока на ε единиц и вернуться на шаг 1. (Остановиться, если текущий поток стал равен v.)

Этот алгоритм в действительности строит поток с минимальной стоимостью величиной в p единиц для $p = 1,2,...,v$. Для нахождения самого дешевого пути на шаге 2 можно использовать алгоритм поиска кратчайших путей, в котором допускаются отрицательные длины дуг (замечание: алгоритм Дейкстры не используется в общем случае для сети с отрицательными длинами дуг).

❖ ***Пример 6.2.*** Применим вышеописанный алгоритм для передачи 4 единиц потока с минимальной стоимостью в сети, показанной на рис. 6.12. Для каждой дуги заданы пропускная способность и стоимость передачи единицы потока $[b_{ij}(c_{ij})]$.

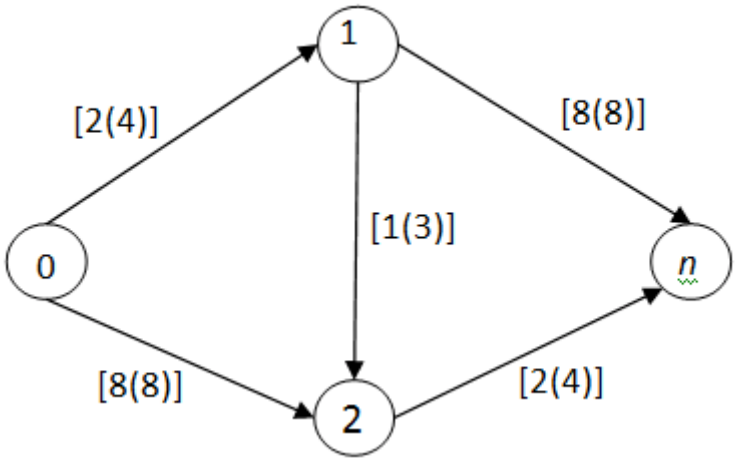


Рис. 6.12. Сеть, пример 6.2

Вначале пошлем одну единицу потока вдоль пути V_0, V_1, V_2, V_n, дуга e_{12} станет насыщенной. Остаточные пропускные способности станут равны $b'_{12} = 0$, $b'_{21} = 1$, а модифицированные стоимости примут значения $c^*_{12} = \infty$, $c^*_{21} = -3$. Стоимость передачи потока z_1=4*1+3*1+2*1=9. Остаточные пропускные способности дуг и модифицированные стоимости показаны на рис. 6.13.

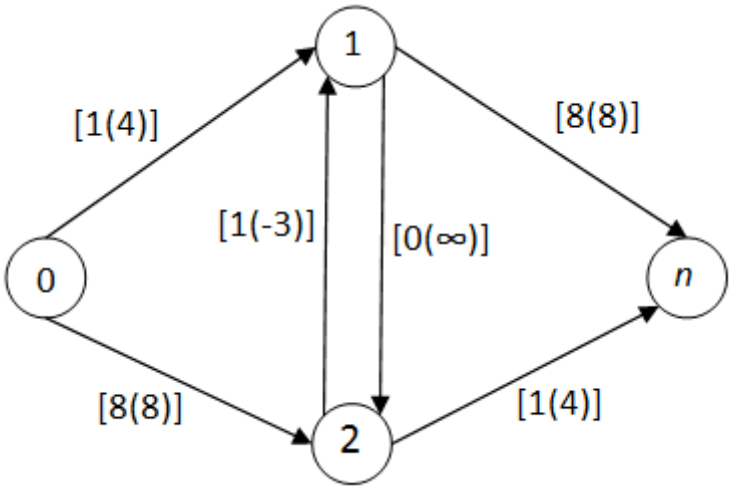


Рис. 6.13. Остаточные пропускные способности сети, пример 6.2

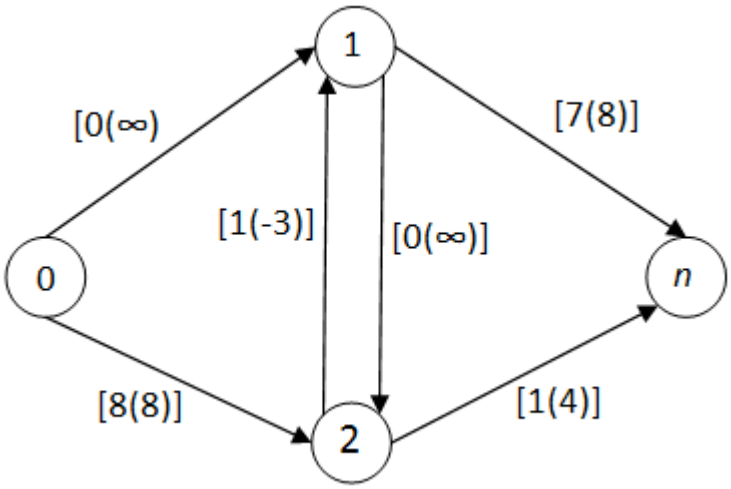


Рис. 6.14. Остаточная сеть после передачи потока по пути V_0, V_1, V_2, V_n

Посылаем еще единицу потока по пути V_0, V_1, V_n, получим остаточную сеть, показанную на рис. 6.14. Стоимость передачи потока z_2=4*1+8*1=12. После передачи еще единицы потока по пути V_0, V_2, V_n получится сеть, изображенная на рис. 6.15. Стоимость передачи потока z_3=8*1 +4*1=12.

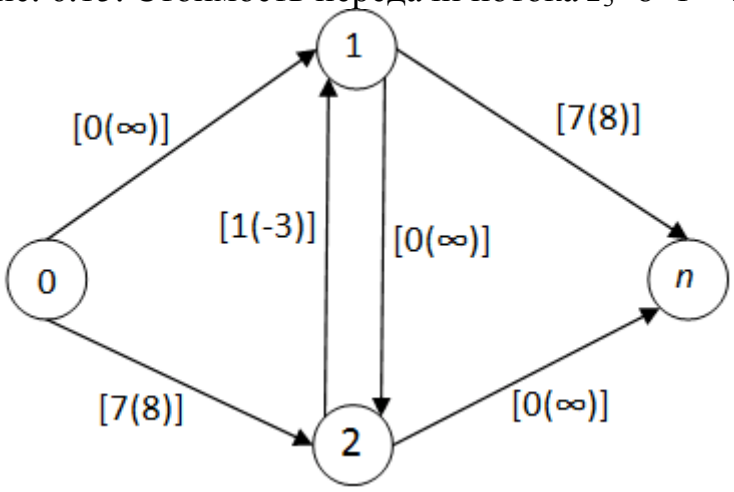


Рис. 6.15. Остаточная сеть после передачи потока по пути V_0, V_2, V_n

Далее необходимо передать одну единицу вдоль пути V_0, V_2, V_1, V_n, после чего модифицированные стоимости станут такими, как показано на рис. 6.16. Это окончательные остаточные пропускные способности дуг. Стоимость передачи потока на этом шаге z_4=8*1–3*1+8*1=13. Итоговый поток: v=1+1+1+1 = 4. Итоговая стоимость передачи этого потока $z = \sum c_{ij}x_{ij} = z_1 + z_2 + z_3 + z_4 = 48$.

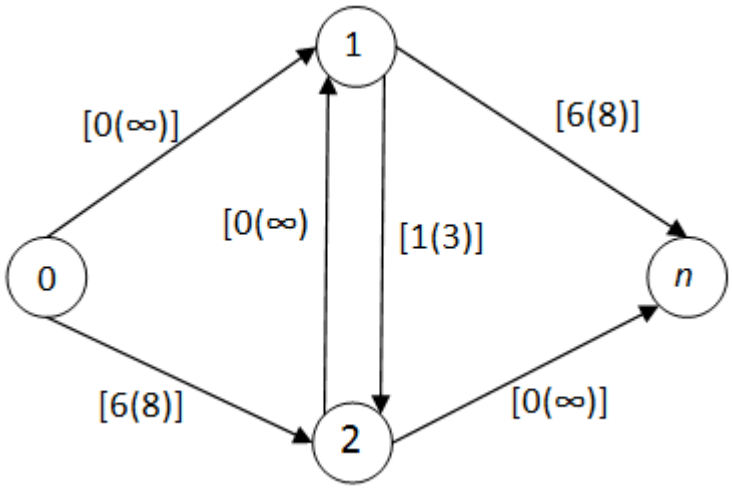


Рис. 6.16. Остаточная сеть после передачи потока по пути V_0, V_2, V_1, V_n

Заметим, что в окончательном потоке примера 6.2 не используется самый дешевый путь.

Для решения задачи 2 можно использовать тот же самый алгоритм за исключением того, что всякий раз, когда суммарная стоимость достигает бюджетного ограничения, следует останавливаться.

Метод анализа и оценки проекта REPT-метод

Одно из наиболее известных приложений теории потоков в сетях называется PERT (program evaluation and review technique – метод анализа и оценки программ)[1]. Эта техника используется для оптимального распределения денежных ресурсов среди заданий проекта, при котором проект можно завершить в кратчайшие сроки. Рассмотрим долгосрочный проект, состоящий из множества работ. Работы частично упорядочены в соответствии с технологическими ограничениями. Например, промывка должна производиться раньше сушки. Проект можно представить ориентированной сетью. Где работы – ориентированные дуги. Для каждой индивидуальной работы имеется время нормального завершения и «абсолютно» минимальное время завершения.

Варианты задачи:

1) Чем больше денег затрачивается на работу, тем быстрее эта работа будет завершена. Задача заключается в том, чтобы завершить весь проект как можно раньше при фиксированном бюджете.

[1]REPT-метод разработан в 1956 году корпорацией Lockheed Air Craft, консалтинговой компанией Booz, Allen & Hamilton и особым проектным бюро ВМС США в процессе создания ракетного комплекса Polaris. Вскоре этот метод стал повсеместно применяться для планирования проектов в вооруженных силах США.

2) В другой версии рассматриваемой задачи требуется завершить весь проект до наступления определенного срока окончания с минимальной затратой денежных ресурсов.

Предположим, что у нас достаточно денег для нормального завершения всех работ. Все дополнительные деньги следует оптимально распределить среди работ, т. е. так, чтобы весь проект был завершен в самый ранний срок.

Необходимо определить работы, которые входят в самый длинный путь, чтобы дополнительные деньги потратить на эти работы (например, какая-то работа нормально выполняется за три дня, но при затрате на нее 3-х долларов она может быть закончена за 2 дня, а при затрате 6-ти долларов – за 1 день), тем самым уменьшить время завершения всего проекта. Это называется *планированием критических путей*.

Идея алгоритма:

Для того чтобы сократить время завершения проекта, нужно сократить самый длинный путь из V_0 в V_n. Так как в самом длинном пути дуг много, то потратим деньги на дуги (работы) с наименьшими значениями c_{ij}. Это бы наиболее эффективно уменьшило длину самого длинного пути. Продолжим сокращение дуги до тех пор, пока не выполнится хотя бы одно из условий (1) или (2).

(1) Дуга сокращена до минимального времени a_{ij}.

(2) Рассматриваемый путь не длиннее самого длинного пути.

В случае (1) берем другую дугу того же самого пути со вторым по величине значением c_{ij}. В случае (2) есть несколько самых длинных путей. Для того чтобы одновременно сократить эти пути, требуется сократить дуги, лежащие на различных таких путях. Выберем множество таких дуг так, чтобы общая сумма величин c_{ij} была минимальной.

Иногда дуга e_{ij} пути *P'* сокращается до минимального времени на одном из ранних этапов. В дальнейшем сокращаются и другие дуги из *P'*, и *P'* не станет самым длинным путем, даже если дугу e_{ij} удлинить до ее исходной длины. (Заметим, что удлинение дуги означает возможность экономии денег.)

Введем некоторые понятия.

Дуга, принадлежащая хотя бы одному из самых длинных путей, называется *активной*, а не принадлежащая никакому из таких путей – *слабой*. Дуга, длина которой была уменьшена до минимального времени завершения, называется *жесткой*.

Шаг 1. На основе текущих длин дуг найти все самые длинные пути из V_0 в V_n. Сделать все дуги самых длинных путей активными.

Шаг 2. Рассматривая величины c_{ij} для активных дуг как их пропускные способности, найти максимальный поток из V_0 в V_n. (При

нахождении максимального потока получаем минимальный разрез, указывающий самый дешевый способ для сокращения длительности всего проекта.)

Шаг 3. Если дуга становится жесткой, то присваиваем ей бесконечную пропускную способность.

Шаг 4. Если на шаге 2 максимальный поток становится бесконечным, то это означает, что дальнейшее уменьшение длительности проекта невозможно.

Шаг 5. Найти расстояние от V_0 до каждого узла. Расстояние до узла V_i соответствует кратчайшему времени, за которое этот узел может быть достигнут. Если длина активной дуги короче разности расстояний между ее концевыми узлами, то увеличиваем длину этой дуги (что эквивалентно уменьшению стоимости).

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Приведите примеры практического использования задачи о максимальном потоке.
2. В чем состоит задача о максимальном потоке в сети?
3. Какие действия нужно провести, если в сети несколько источников или сливов?
4. Верно ли, что поток в сети можно интерпретировать с потоком жидкости в трубах?
5. Верно ли, что отток в источнике равен притоку в сливе?
6. Что означает величина (цена) потока?
7. Верно ли, что если в задаче о максимальном потоке пропускные способности всех дуг различны, то существует единственный минимальный разрез, разделяющий источник и сток?
8. Может ли в сети существовать несколько различных максимальных потоков?
9. Верно ли, что если пропускные способности всех дуг различны, то существует единственное множество потоков через дуги, дающее максимальное значение потока?
10. Справедливо ли следующее утверждение: если в сети больше одного источника, то алгоритм Форда-Фалкерсона для поиска максимального потока применить нельзя.
11. Справедливо ли следующее утверждение: при определении пропускной способности разреза сети учитываются все дуги, входящие в разрез.
12. Справедливо ли следующее утверждение: в сети можно найти поток с величиной, равной пропускной способности разреза с большей, чем у минимального разреза.
13. Справедливо ли следующее утверждение: для поиска минимального разреза можно использовать доказательство теоремы Форда-Фалкерсона.

14. Верно ли, что в задаче о максимальном потоке любой увеличивающий путь сети содержит только прямые дуги?

15. Верно ли, что в задаче о максимальном потоке использование обратных дуг уменьшает величину потока в сети?

16. Верно ли, что если в сети два разреза скрещиваются, то не существует других нескрещивающихся разрезов?

17. Справедливо ли следующее утверждение: в алгоритмах поиска максимального потока можно использовать принцип поиска в ширину и в глубину.

18. Справедливо ли следующее утверждение: в алгоритме поиска потока минимальной стоимости используется алгоритм Дейкстры.

19. Опишите основную идею алгоритма Эдмондсома и Карпа.

20. Опишите основную идею алгоритма Диница.

21. Для какой задачи используется метод анализа и оценки проекта REPT-метод?

22. В чем заключается идея «модифицированной стоимости» в REPT-методе.

7. НЕПЕРЕСЕКАЮЩИЕСЯ ЦЕПИ И РАЗДЕЛЯЮЩИЕ МНОЖЕСТВА

7.1. Постановки задач: непересекающиеся цепи и разделяющие множества

Интуитивно очевидно, что граф тем более связен (при использовании различных мер связности), чем больше существует различных *цепей*, соединяющих один узел с другим, и тем менее связен, чем меньше нужно удалить промежуточных узлов, чтобы отделить один узел от другого. Теорема Менгера придает этим неформальным наблюдениям точный и строгий смысл.

Пусть $G(V, E)$ – связный граф, u и v – две его несмежные вершины.

📖 **Определение.** Две цепи (u, v) называются *вершинно-непересекающимися*, если у них нет общих вершин, отличных от u и v.

📖 **Определение.** Две цепи $\langle u, v \rangle$ называются *реберно-непересекающимися*, если у них нет общих ребер.

✍ **Замечание 1.** Если две цепи вершинно не пересекаются, то они и реберно не пересекаются.

Обозначим через $P(u,v)$ множество вершинно-непересекающихся цепей $\langle u, v \rangle$.

Множество R вершин (и/или ребер) графа G *разделяет* две вершины u и v, если u и v принадлежат разным компонентам связности графа $G\backslash R$.

📖 **Определение.** *Разделяющее множество ребер* называется *разрезом.*

Разделяющее множество вершин для вершин u и v обозначим $R(u, v)$.

Для заданных вершин u и v множества $P(u, v)$ и $S(u, v)$ можно выбирать различным образом.

Из определений следует, что любое множество $P(u,v)$ вершинно-непересекающихся цепей обладает тем свойством, что

$$\bigcap_{\langle u,v\rangle \in P(u,v)} \langle u, v \rangle = \{u, v\},$$

а любое разделяющее множество вершин $R(u, v)$ обладает тем свойством, что

$$G - R = G_1 \cup G_2 \;\&\; v \in G_1 \;\&\; u \in G_2.$$

✍ **Замечание 2.** Если u и v принадлежат разным компонентам связности графа G, то $|P(u, v)| = 0$ и $|R(u, v)| = 0$.

❖ ***Пример 7.1.*** В графе, диаграмма которого представлена на рис. 7.1, можно выбрать множества вершинно-непересекающихся путей $P_1=\{(u,a,d,v), (u,b,e,v)\}$ и $P_2=\{(u,b,d,v), (u,c,e,v)\}$. Заметим, что путь (u,c,b,d,v) образует множество P_3 вершинно-непересекающихся путей, состоящее из одного элемента. Множества вершин $R_1=\{a,b,c\}$ и $R_2=\{d,e\}$ являются разделяющими для вершин u и v.

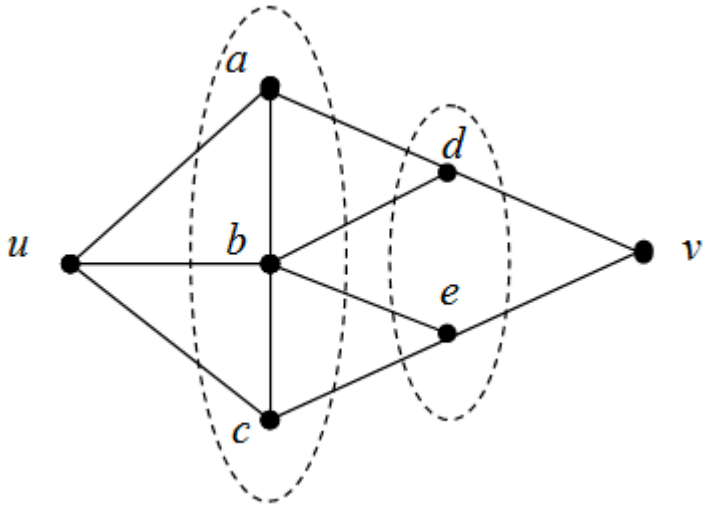


Рис. 7.1. Вершинно-непересекающиеся пути и разделяющие множества вершин

7.2. Теорема Менгера в «вершинной форме»

🕮 **Теорема** (Менгер, 1927 г.). Пусть u и v – несмежные вершины в графе G. Наименьшее число вершин в множестве, разделяющем u и v, равно наибольшему числу вершинно-непересекающихся простых $\langle u,v\rangle$-цепей:

$$\max |P(u, v)| = \min|R(u, v)|.$$

✍ **Замечание 3.** Легко видеть, что $|P| \le |R|$. Действительно, любая $\langle u,v\rangle$-цепь проходит через R. Если бы $|P| > |R|$, то в R была бы вершина, принадлежащая более чем одной цепи из P, что противоречит выбору P. Таким образом, для $\forall P$ и $\forall R\ |P| \le |R|$. Следовательно, $\max|P| \le \min|R|$. Утверждение теоремы состоит в том, что в любом графе *существуют* такие P и R, что достигается равенство $|P| = |R|$.

Доказательство. Пусть G – связный граф, u и v – несмежные вершины. Совместная индукция по p и q. База: наименьший граф, удовлетворяющий условиям теоремы, состоит из трех вершин u,w,v и двух ребер (u,w) и (w,v). В нем $P(u,v)=\{\langle u,w,v\rangle\}$ и $R(u,v)=\{w\}$. Таким образом, $|P(u,v)|=|R(u,v)|=1$. Пусть утверждение теоремы верно для всех графов с числом вершин, меньше p, и/или числом ребер, меньше q. Рассмотрим граф G с p вершинами и q ребрами. Пусть $u,v\in V$, причем u,v – не смежны и R – некоторое наименьшее множество вершин, разделяющее u и v. Обозначим n: = $|R|$. Рассмотрим три случая.

(1) Пусть в R есть вершины, несмежные с u и несмежные с v. Тогда граф $G-R$ состоит из двух *нетривиальных* графов G_1 и G_2. Образуем два новых графа G_u и G_v, стягивая нетривиальные графы G_1 и G_2 к вершинам u и v соответственно: $G_u:=G/G_1$, $G_v:=G/G_2$ (рис. 7.2).

R по-прежнему является наименьшим разделяющим множеством для u и v как в G_u, так и в G_v. Так как G_1 и G_2 нетривиальны, то G_u и G_v имеют меньше вершин и/или ребер, чем G. Следовательно, по индукционному предположению в G_u и в G_v имеется n вершинно-непересекающихся простых цепей. Комбинируя отрезки этих цепей от u до R и от R до v, получаем n вершинно-непересекающихся простых цепей в G.

(2) Пусть все вершины R смежны с u или с v (для определенности пусть с u) и среди вершин S есть вершина w, смежная одновременно с u и с v (рис. 7.3).

Рассмотрим граф $G':=G-w$. В нем $R-w$ – разделяющее множество для u и v, причем, наименьшее. По индукционному предположению в G' имеется $|R-w|=n-1$ вершинно-непересекающихся простых цепей. Добавим к ним цепь $\langle u,w,v\rangle$. Она простая и вершинно не пересекается с остальными. Таким образом, имеем n вершинно-непересекающихся простых цепей в G.

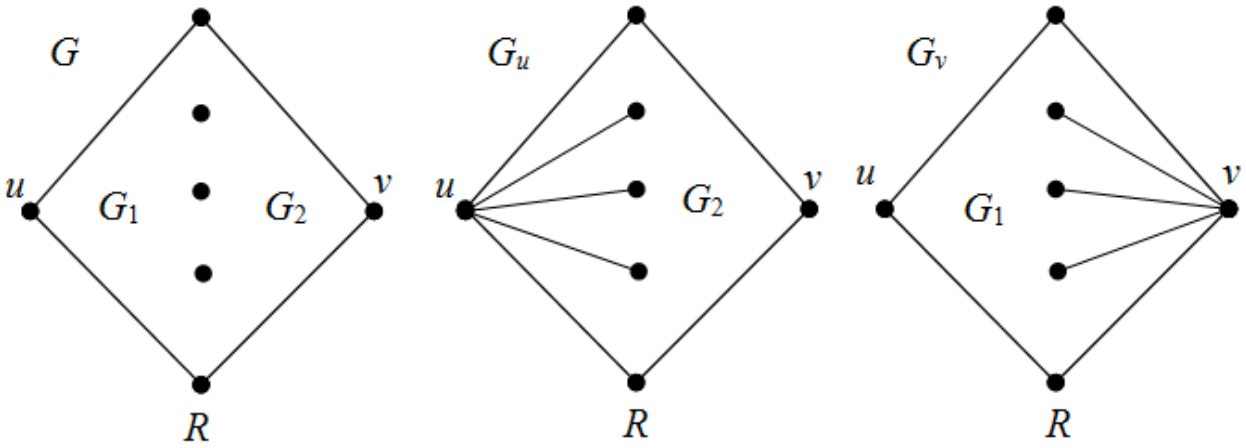


Рис. 7.2. К доказательству теоремы Менгера, случай 1

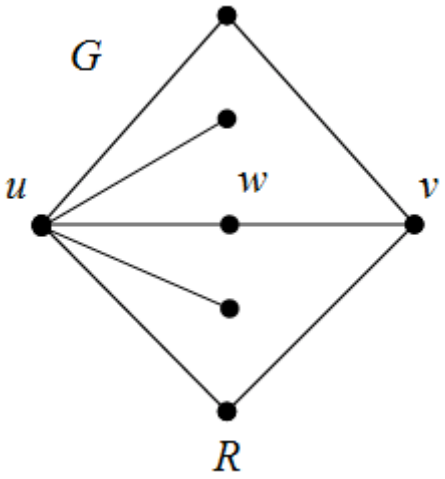


Рис. 7.3. К доказательству теоремы Менгера, случай 2

(3) Пусть теперь все вершины R смежны с u или с v (для определенности пусть с u) и среди вершин R нет вершин, смежных одновременно с вершиной u и с вершиной v. Рассмотрим *кратчайшую* $\langle u, v\rangle$-цепь $\langle u, w_1, w_2, ..., v\rangle$ $w_1 \in R$, $w_2 \neq v$ (рис. 7.4).

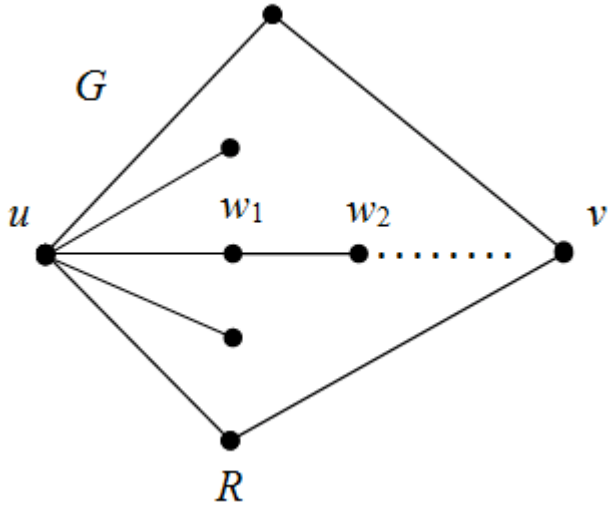


Рис. 7.4. К доказательству теоремы Менгера, случай 3

Рассмотрим граф $G' := G/\{w_1, w_2\}$, полученный из G склейкой вершин w_1 и w_2 в вершину w_1. Имеем $w_2 \notin R$, в противном случае цепь $\langle u, w_1, w_2, ..., v\rangle$ была бы еще более короткой. Следовательно, в графе G' множество R по-прежнему является наименьшим, разделяющим u и v, и граф G' имеет, по крайней мере, на одну дугу меньше. По индукционному предположению в G' существуют n вершинно-непересекающихся простых цепей. Но цепи, которые не пересекаются в G', не пересекаются и в G. Таким образом, получаем n вершинно-непересекающихся простых цепей в G. Что и требовалось доказать.

7.3. Варианты теоремы Менгера

Теорема Менгера представляет собой весьма общий факт, который в разных формулировках встречается в различных областях математики. Комбинируя вершинно- и реберно-непересекающиеся цепи, разделяя не отдельные вершины, а множества вершин, используя инварианты χ и λ, можно сформулировать и другие утверждения, подобные теореме Менгера. Например:

📖 **Теорема.** Для любых двух несмежных вершин u и v графа G наибольшее число реберно-непересекающихся $\langle u, v\rangle$-цепей равно наименьшему числу ребер в (u, v)-разрезе.

🕮 **Теорема.** Чтобы граф G был n-связным, необходимо и достаточно, чтобы любые две несмежные вершины были соединены не менее чем n вершинно-непересекающимися простыми цепями.

Другими словами, в любом графе G любые две несмежные вершины соединены не менее чем $\chi(G)$ вершинно-непересекающимися простыми цепями.

Также можно отметить и связь между теоремой Менгера и теоремой Форда и Фалкерсона.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Дайте определение вершинно-непересекающимся цепям.
2. Дайте определение реберно-непересекающимся цепям.
3. Как называется разделяющее множество ребер?
4. Каким свойством обладает любое множество вершинно-непересекающихся цепей?
5. Каким свойством обладает любое разделяющее множество вершин?
6. Верно ли, что вершинно-непересекающиеся цепи и реберно-непересекающиеся цепи всегда различны?
7. Чему равна мощность множества вершинно-непересекающихся цепей $|P(u, v)|$, если u и v принадлежат разным компонентам связности графа?
8. Чему равна мощность разделяющего множества вершин множества $|R(u, v)|$, если u и v принадлежат разным компонентам связности графа?
9. Верно ли, что наименьшее число вершин в множестве, разделяющем u и v, равно наибольшему числу вершинно- непересекающихся простых $\langle u,v\rangle$-цепей?
10. Сформулируйте теорему Менгера.
11. Какие рассматриваются случаи при доказательстве теоремы Менгера?
12. Есть ли связь между теоремой Менгера и теоремой Форда-Фалкерсона о максимальных потоках?

8. ПАРОСОЧЕТАНИЯ

8.1. Максимальные и наибольшие паросочетания

Пусть Г (*V*,*E*) – граф и $S \in E$ – некоторое множество ребер в нем.

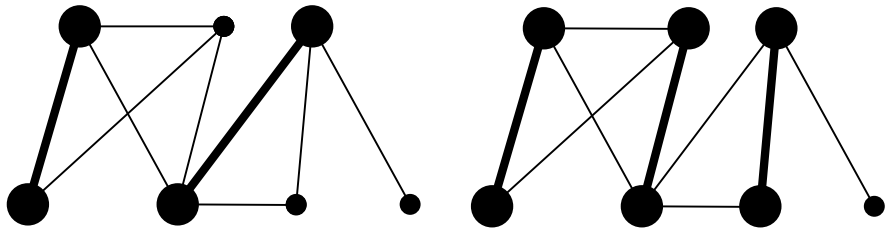
🕮 **Определение.** Множество *S* называется *паросочетанием*, если любые два ребра из него не имеют общей вершины, т. е. не являются смежными.

Множество из одного ребра тоже будем называть паросочетанием, как всякое пустое множество.

🕮 **Определение.** Паросочетание называется *максимальным,* если к нему нельзя добавить ни одного ребра так, чтобы снова получилось паросочетание.

🕮 **Определение.** Паросочетание называется *наибольшим*, если оно состоит из наибольшего возможного количества ребер.

Каждое наибольшее паросочетание является максимальным; обратное неверно (рис. 8.1).



а *б*

Рис. 8.1. Паросочетания:
а – максимальное; *б* – наибольшее и максимальное

Поиск наибольшего паросочетания в графе представляет собой классическую алгоритмическую задачу. Существуют разные варианты задач, например:

– наибольшее паросочетание по количеству дуг;
– наибольшее паросочетание по сумме весов этих дуг;
– наибольшее паросочетание в двудольных графах;
– наибольшее паросочетание в произвольном или простом графе.

Рассмотрим ее решение не для общего случая, а для графов частного вида – *двудольных*.

🕮 **Определение.** Граф Г (*V*,*E*) называется *двудольным*, если его множество вершин *A* можно представить в виде объединения двух его непустых подмножеств V_1, V_2 без общих элементов так, что любое ребро из *E*

будет иметь один конец в V_1, а другой конец – в V_2. Таким образом, нет ни одного ребра, которое соединяло бы вершины из V_1 или соединяло бы вершины из V_2.

Если считать, что $V_1 = \{x_1,...,x_r\}$, $V_2 = \{y_1,...,y_s\}$, то двудольный граф можно описать не только матрицей смежностей, но и *матрицей двудольного графа*: эта матрица – размером $r \times s$, если обозначать ее общий элемент через m_{ij}, то полагают

$$m_{ij}\begin{cases}1, \textit{если } (x_i, y_j) \in E,\\ 0, \textit{если } (x_i, y_j) \notin E.\end{cases}$$

Такая матрица описывает граф однозначно, хотя является намного меньшей по объему, чем матрица смежности графа в этом случае.

Когда множество V_1 состоит из *n* вершин, а множество V_2 состоит из *m* вершин, и в двудольном графе проведены *все* возможные ребра, то говорят, что двудольный граф является *полным двудольным графом* и обозначают как $K_{m,n}$.

8.2. Алгоритм выбора наибольшего сочетания в двудольном графе с матрицей двудольного графа

Шаг 0. По матрице $M = m_{ij}$ данного двудольного графа строим рабочую таблицу: матрица тех же размеров; в клетку с номером (*ij*) поместим символ «×» и назовем ее *недопустимой*, если в матрице двудольного графа $m_{ij} = 0$; если же $m_{ij} = 1$, то в клетку рабочей таблицы не запишем ничего и назовем такую клетку *допустимой*. Слева от рабочей таблицы расположим для удобства номера ее строк, а сверху над таблицей расположим номера ее столбцов.

Шаг 1. Просмотрим слева направо первую строку и в первую же допустимую клетку первой строки поставим символ «1». Если в первой строке все клетки недопустимы, то перейдем ко второй строке и в первую же (при просмотре слева направо) допустимую клетку поставим «1». Если же нет допустимых клеток и во второй строке, то проставим указанным выше способом «1» в третьей строке. Если окажется, что во всей таблице все клетки недопустимы, то на этом все действия заканчиваются, и выдается ответ: «в графе нет ребер». Если же все-таки удастся поставить первую «1», то после этого просмотрим все остальные строки таблицы в порядке возрастания их номеров. В каждой очередной строке просматриваем ее клетки слева направо и фиксируем первую по ходу просмотра допустимую клетку такую, которая является независимой по отношению к допустимым клеткам, в которых уже стоит символ «1», и проставляем в эту клетку «1», после чего переходим к тем же действиям в следующей строке. Если в строке такой клетки не окажется, то переходим к следующей строке и выполняем в ней ту же проверку.

Фиксируем набор ребер в графе, соответствующих проставленным в таблице символам «1». (Под ребром, *соответствующим* символу «1» в рабочей таблице, подразумевается следующее: если «1» стоит в клетке с номером (*ij*), то *соответствующим* будет ребро (x_i, y_j)) Легко заметить, что этот набор ребер – максимальное паросочетание.

Если в результате проведения всех действий на этом шаге в каждой строке рабочей таблицы окажется символ «1», то ребра из указанного только что паросочетания и составляют наибольшее паросочетание, и действия окончены. Если же в результате проведения первого шага остались строки без «1», то пометим их справа от таблицы символом «*», переходим к следующему шагу.

Шаг 2. Просмотрим все помеченные строки в порядке возрастания их номеров. Просмотр очередной строки состоит в следующем: в строке отыскиваются допустимые клетки, и столбцы, в которых эти клетки расположены, помечаются номером просматриваемой строки. При этом соблюдается принцип ($\wp$): *если пометка уже стоит, то на ее место вторая не ставится.* Эти пометки физически выставляются внизу, под таблицей.

Если в результате этого шага ни один из столбцов не будет помечен, то это означает, что уже имеющееся паросочетание (полученное на *Шаге 1*) является искомым наибольшим, и все действия прекращаются. Если же среди столбцов появятся помеченные, то переходим к следующему шагу.

Шаг 3. Просмотрим помеченные столбцы в порядке возрастания их номеров. Просмотр столбца состоит в следующем: в столбце отыскивается символ «1», и строка, в которой он расположен, помечается номером просматриваемого столбца.

Физически пометки выставляются справа от таблицы, на уровне соответствующих строк. Всегда соблюдается принцип ($\wp$).

Если в результате действий по этому шагу возникнет ситуация, когда в просматриваемом столбце нет символа «1», то действия на данном шаге прерываются и надо перейти к следующему шагу – *Шагу 4*. Если же в результате действий по этому шагу будут просмотрены все помеченные ранее столбцы и, тем самым, возникнет набор помеченных строк (одни будут помечены символом «*», другие – номерами столбцов), то следует вернуться к *Шагу 2* и повторить предусмотренные им действия.

Если в результате этих действий по *Шагу 2* не возникнет новых помеченных столбцов, то это означает, что имеющееся паросочетание является искомым, и процесс останавливается. Если же среди помечаемых столбцов возникнут новые помеченные столбцы, то следует повторить *Шаг 3* с новым набором помеченных столбцов. Если в результате этих действий не возникнет новых помеченных строк, значит, имеющееся паросочетание является искомым.

Шаг 4. Рассматривается столбец, имеющий пометку и не содержащий символа «1». Изменяется набор символов «1», расположенных в рабочей таблице.

В рассматриваемый столбец поставим символ «1» в строку, номер которой равен пометке этого столбца. Затем в этой строке отыщем «старый» символ «1» и переместим его по его столбцу в строку, номер которой равен пометке при этом столбце. (Можно доказать, что описанное действие всегда осуществимо.) Затем в строке, куда попал последний символ «1», отыщем «старый» символ «1» и с ним проделаем то же самое. В конце концов, очередной перемещаемый «старый» символ «1» окажется в строке, где больше символов «1» нет. Возник новый набор символов «1», в котором уже элементов на один больше, чем в исходном наборе символов «1». По этому новому набору можно построить паросочетание так же, как это делалось в самом начале, и после этого повторить все сначала.

В конце возникнет одно из указанных выше условий прекращения действий по алгоритму, и наибольшее паросочетание будет найдено.

❖ ***Пример 8.1.*** Найти наибольшее паросочетание в следующем двудольном графе со следующей матрицей:

0	1	0	0	0	1	0	1	0
1	0	1	1	0	0	0	0	0
0	1	0	1	0	0	0	1	0
0	1	1	1	1	1	0	1	1
0	0	0	1	0	1	0	1	0
0	1	0	0	0	1	0	1	0
1	0	1	1	0	1	1	0	1
0	1	0	1	0	1	0	0	0
1	0	0	1	0	0	0	1	0

Графическое изображение двудольный граф на рис. 8.2

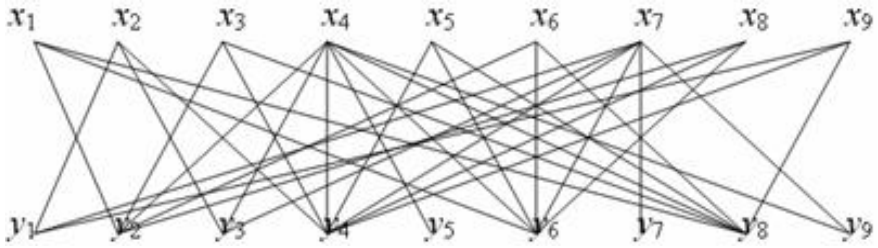


Рис. 8.2. Двудольный граф, пример 8.1

Будем реализовывать шаг за шагом описанный выше алгоритм. Рабочая таблица изображена на рис. 8.3.

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x		x	x	x		x		x	
2		x			x	x	x	x	x	

3	x		x		x	x	x		x	
4	x						x			
5	x	x	x		x		x		x	
6	x		x	x	x		x		x	
7		x			x			x		
8	x		x		x		x	x	x	
9		x	x		x	x	x		x	
метки										

Рис. 8.3. Рабочая таблица, пример 8.1

Выполняем первый шаг 1: расставляем независимые единицы и отмечаем строки, в которые единицы не попали, рис. 8.4.

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	
2	1	x			x	x	x	x	x	
3	x		x	1	x	x	x		x	
4	x		1				x			
5	x	x	x		x	1	x		x	
6	x		x	x	x		x	1	x	
7		x			x		1	x		
8	x		x		x		x	x	x	*
9		x	x		x	x	x		x	*
метки										

Рис. 8.4. Рабочая таблица, итерация № 1, этап 1, шаг 1

Паросочетание, которое соответствует выбранным единицам: (x_1, y_2); (x_2, y_1); (x_3, y_4); (x_4, y_3); (x_5, y_6); (x_6, y_8); (x_7, y_7).

Далее шаг 2, столбцы допустимых клеток помеченных строк пометим номерами этих строк, следуя принципу ($\wp$), рис. 8.5.

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	
2	1	x			x	x	x	x	x	
3	x		x	1	x	x	x		x	
4	x		1				x			
5	x	x	x		x	1	x		x	
6	x		x	x	x		x	1	x	
7		x			x		1	x		
8	x		x		x		x	x	x	*
9		x	x		x	x	x		x	*
метки	9	8		8		8		9		

Рис. 8.5. Рабочая таблица, итерация № 1, этап 1, шаг 2

Далее шаг 3, в помеченных столбцах отыщем единицы и их строки пометим номерами их столбцов (рис. 8.6). Затем снова делаем шаг 2 (второй этап), пометим столбцы, отправляясь от помеченных строк и соблюдая принцип ($\wp$) (рис. 8.7). Далее снова шаг 3, просмотрим помеченные столбцы и пометим строки (рис. 8.8).

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	2
2	1	x			x	x	x	x	x	1
3	x		x	1	x	x	x		x	4
4	x		1				x			
5	x	x	x		x	1	x		x	6
6	x		x	x	x		x	1	x	8
7		x			x		1	x		
8	x		x		x		x	x	x	*
9		x	x		x	x	x		x	*
метки	9	8		8		8		9		

Рис. 8.6. Рабочая таблица, итерация № 1, этап 1, шаг 3

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	2
2	1	x			x	x	x	x	x	1
3	x		x	1	x	x	x		x	4
4	x		1				x			
5	x	x	x		x	1	x		x	6
6	x		x	x	x		x	1	x	8
7		x			x		1	x		
8	x		x		x		x	x	x	*
9		x	x		x	x	x		x	*
метки	9	8	2	8		8		9		

Рис. 8.7. Рабочая таблица, итерация № 1, этап 2, шаг 2

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	2
2	1	x			x	x	x	x	x	1
3	x		x	1	x	x	x		x	4
4	x		1				x			3
5	x	x	x		x	1	x		x	6
6	x		x	x	x		x	1	x	8
7		x			x		1	x		
8	x		x		x		x	x	x	*
9		x	x		x	x	x		x	*
метки	9	8	2	8		8		9		

Рис. 8.8. Рабочая таблица, итерация № 1, этап 2, шаг 3

Теперь снова шаг 2 (третий этап), пометим столбцы (при принципе ($\wp$)) (рис. 8.9).

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	2
2	1	x			x	x	x	x	x	1
3	x		x	1	x	x	x		x	4
4	x		1				x			3
5	x	x	x		x	1	x		x	6
6	x		x	x	x		x	1	x	8
7		x			x		1	x		
8	x		x		x		x	x	x	*
9		x	x		x	x	x		x	*
метки	9	8	2	8	4	8		9	4	

Рис. 8.9. Рабочая таблица, итерация № 1, этап 3, шаг 2

Теперь снова делаем шаг 3: просмотр столбцов и поиск «1», но в столбце №5 с меткой 4 нет «1»: Следовательно, можно набор единиц увеличить на одну – шаг 4: новые единицы – выделены жирным курсивом, старые – зачеркнуты (рис. 8.10).

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	2
2	~~1~~	x	***1***		x	x	x	x	x	1
3	x		x	1	x	x	x		x	4
4	x		~~1~~		***1***		x			3
5	x	x	x		x	1	x		x	6
6	x		x	x	x		x	1	x	8
7		x			x		1	x		
8	x		x		x		x	x	x	*
9	***1***	x	x		x	x	x		x	*
метки	9	8	2	8	4	8		9	4	

Рис. 8.10. Рабочая таблица, итерация № 1, этап 3, шаг 3

Получим новый набор независимых единиц (рис. 8.11). Теперь вся процедура повторяется сначала – новая итерация. Шаг 1 – единственная пометка «*» у строки №8.

Шаг 2 – пометим столбцы допустимых клеток этой строки ее номером (рис. 8.12).

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
1	x	1	x	x	x		x		x
2		x	1		x	x	x	x	x
3	x		x	1	x	x	x		x

4	x				1		x		
5	x	x	x		x	1	x		x
6	x		x	x	x		x	1	x
7		x			x		1	x	
8	x		x		x		x	x	x
9	1	x	x		x	x	x		x

Рис. 8.11. Новый набор независимых единиц

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	
2		x	1		x	x	x	x	x	
3	x		x	1	x	x	x		x	
4	x				1		x			
5	x	x	x		x	1	x		x	
6	x		x	x	x		x	1	x	
7		x			x		1	x		
8	x		x		x		x	x	x	*
9	1	x	x		x	x	x		x	
метки		8		8		8				

Рис. 8.12. Рабочая таблица, итерация № 2, этап 1, шаг 1 и шаг 2

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	2
2		x	1		x	x	x	x	x	
3	x		x	1	x	x	x		x	4
4	x				1		x			
5	x	x	x		x	1	x		x	6
6	x		x	x	x		x	1	x	
7		x			x		1	x		
8	x		x		x		x	x	x	*
9	1	x	x		x	x	x		x	
метки		8		8		8				

Рис. 8.13. Рабочая таблица, итерация №2, этап 1, шаг 3

Затем шаг 3: в помеченных столбцах найдем единицы и их строки пометим номерами столбцов (рис. 8.13).

Шаг 2 второго этапа итерации №2 изображен на рис. 8.14, шаг 3 – на рис. 8.15.

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	2
2		x	1		x	x	x	x	x	
3	x		x	1	x	x	x		x	4
4	x				1		x			
5	x	x	x		x	1	x		x	6
6	x		x	x	x		x	1	x	

7		x			x		1	x		
8	x		x		x		x	x	x	*
9	1	x	x		x	x	x		x	
метки		8		8		8		1		

Рис. 8.14. Рабочая таблица, итерация №2, этап 1, шаг 2

Сложилась ситуация, когда расстановка пометок «зациклилась». Это означает, что искомое паросочетание состоит из ребер, соответствующих проставленным единицам, т. е. (x_1, y_2); (x_2, y_3); (x_3, y_4); (x_4, y_5); (x_5, y_6); (x_6, y_8); (x_7, y_7);(x_9, y_1).

y / *x*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*метки*
1	x	1	x	x	x		x		x	2
2		x	1		x	x	x	x	x	
3	x		x	1	x	x	x		x	4
4	x				1		x			
5	x	x	x		x	1	x		x	6
6	x		x	x	x		x	1	x	8
7		x			x		1	x		
8	x		x		x		x	x	x	*
9	1	x	x		x	x	x		x	
метки		8		8		8		1		

Рис. 8.15. Рабочая таблица, итерация №2, этап 1, шаг 3

8.3. Различные постановки задач о паросочетаниях

1. Задача о свадьбах. Пусть имеется конечное множество юношей, каждый из которых знаком с некоторым подмножеством конечного множества девушек. В каком случае всех юношей можно женить так, чтобы каждый женился на знакомой девушке?

2. Трансверсаль или система различных представителей (СРП). Пусть $S=\{S_i,\ldots,S_m\}$ – семейство подмножеств конечного множества E. Подмножества S_k не обязательно различны и могут пересекаться. *Системой различных представителей* в семействе S (или *трансверсалью* в семействе S) называется любое подмножество $C=\{c_i,\ldots,c_m\}$ из m элементов множества E, таких, что $\forall k \in 1,...,m\,(c_k \in S_k)$. В каком случае существует трансверсаль?

Все элементы множества C различны, откуда и происходит название «система различных представителей».

Примеры задачи СРП. В университете работает множество профессоров, которые любят создавать комитеты. После того, как профессора сформировали множество комитетов, они решают образовать комитет комитетов (КК). КК состоит из представителей, председательствующих в обычных комитетах. Действуют следующие правила:

а) от каждого комитета имеется в точности один представитель в КК;

б) никто в КК не может быть представителем более одного комитета.

Вопрос: можно ли в каждом комитете избрать председателя так, чтобы никакой профессор не был председателем более одного комитета.

3. Совершенное паросочетание

Пусть $G(V_1,V_2,E)$ – двудольный граф. *Совершенным паросочетанием* из V_1 в V_2 называется паросочетание, покрывающее вершины V_1. В каком случае существует совершенное паросочетание из V_1 в V_2?

Теорема Холла: формулировка и доказательство

Задачи *1, 2* и *3* представляют собой вариации одной и той же задачи. Действительно, задача *1* сводится к задаче *3* следующим образом. V_1 – множество юношей, V_2 – множество девушек, ребра – знакомства юношей с девушками. В таком случае совершенное паросочетание – искомый набор свадеб. Задача *2* сводится к задаче *3* следующим образом. Положим $V_1:=S$, $V_2:=E$, ребро (S_k,e_i) существует, если $e_i \in S_k$. В таком случае совершенное паросочетание – искомая трансверсаль. Таким образом, задачи *1*, *2* и *3* имеют общий ответ: в том и только том случае, когда

$$\textit{любые } k \begin{pmatrix} \textit{юношей} \\ \textit{подмножеств} \\ \textit{вершин из } V_1 \end{pmatrix} \textit{в совокупности} \begin{pmatrix} \textit{знакомы с} \\ \textit{содержат} \\ \textit{смежны с} \end{pmatrix}$$

$$\textit{не менее чем } k \begin{pmatrix} \textit{девушками} \\ \textit{элементов} \\ \textit{вершинами из } V_2 \end{pmatrix},$$

что устанавливается следующей теоремой.

Пусть $G(V, E)$ – граф, A – подмножество вершин V, т. е. $A \subset V$, тогда пусть обозначим через $\varphi(A)$ – множество всех вершин, смежных с вершинами из A.

📖 **Теорема Холла.** Пусть $G(V_1, V_2, E)$ – двудольный граф. Совершенное паросочетание из V_1 в V_2 существует тогда и только тогда, когда $\forall A \subset V_1$ $(|A| \le |\varphi(A)|)$.

Доказательство. Пусть существует совершенное паросочетание из V_1 в V_2. Тогда в $\varphi(A)$ входит $|A|$ вершин из V_2, парных к вершинам из множества A, и, возможно, еще что-то. Таким образом, $|A| \le \varphi(A)$.

Добавим в G две новые вершины u и v, так что вершина u смежна со всеми вершинами из V_1, а вершина v смежна со всеми вершинами из V_2. Совершенное паросочетание из V_1 в V_2 существует тогда и только тогда, когда

существуют $|V_1|$ вершинно-непересекающихся простых (u,v)-цепей (рис. 8.16). Ясно, что $|P(u,v)| \le |V_1|$ (так как V_1 разделяет вершины u и v).

По теореме Менгера $\max|P(u,v)|=\min|R(u,v)|=|R|$, где R – наименьшее множество, разделяющее вершины u и v. Имеем $|R| \le |V_1|$. Покажем, что $|R| \ge |V_1|$. Пусть $R = A \cup B, \quad A \subset V_1, \quad B \subset V_2$. Тогда $\varphi(V_1 \setminus A) \subset B$. Действительно, если бы $\varphi(V_1 \setminus A) \not\subset B$, то существовал бы «обходной» путь (u,v_1,v_2,v) (см. рис. 8.2) и S не было бы разделяющим множеством для u и v. Имеем: $|V_1 \setminus A| \le |\varphi(V_1 \setminus A| \le |B|$. Следовательно, $|S| = |A| + |B| \ge |A| + |V_1 \setminus A| = |V_1|$.

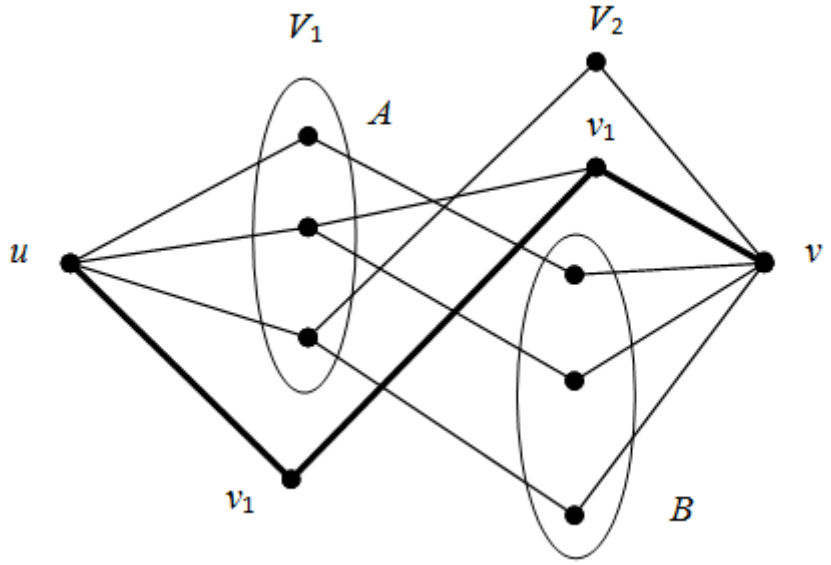


Рис. 8.16. К доказательству теоремы Холла

8.4. Задачи о назначении. Венгерский алгоритм

Задача о назначении на узкие места

Эта задача решается описанным выше алгоритмом. Имеется n рабочих мест на некотором конвейере и n рабочих, которых нужно на эти рабочие места расставить; известна производительность c_{ij} рабочего i на рабочем месте j. Тот факт, что при некотором распределении рабочий i_k попадает на рабочее место j_k, можно описать следующей таблицей:

$$\sigma = \begin{bmatrix} i_1 & i_2 \dots i_k \dots i_n \\ j_1 & j_2 \cdots j_k \cdots j_n \end{bmatrix}.$$

Имея способ σ назначения на рабочие места, можно найти конкретную для этого способа минимальную производительность c'_{ij} и заметить, что именно эта минимальная производительность и определяет скорость и производительность конвейера. То рабочее место, на котором реализуется

минимальная производительность, и называют *узким местом в назначении.* Задача состоит в том, чтобы максимизировать $s = \min c'_{ij}$.

Алгоритм решения задачи о назначении на узкие места

Шаг 0. Фиксируем матрицу производительностей $C = (c_{ij})$ и любое назначение на рабочие места. Пусть *s* – минимальная производительность при этом назначении. Построим рабочую таблицу тех же размеров, что и матрица *C*; в клетку с номером (*ij*) в этой таблице проставим символ «×», если $c_{ij} \le s$; в противном случае эту клетку оставим пустой.

Шаг 1. Рассматривая рабочую таблицу, построенную на предыдущем шаге, как рабочую таблицу в алгоритме для выбора наибольшего паросочетания в двудольном графе, найдем соответствующее наибольшее паросочетание. Если в нем окажется *n* ребер, то по ним восстанавливается новое назначение на рабочие места и с новой, более высокой, минимальной производительностью. Обозначим ее снова через *s* и вернемся к *Шагу 0.* Если же число ребер окажется меньше *n*, то имеющееся назначение на рабочие места уже оптимально.

Задача о назначениях

Примеры постановок задачи:

❖***Пример 8.2.*** Требуется распределить *m* работ (или исполнителей) по *n* станкам. Работа *i* (*i*=1,...,*m*), выполняемая на станке *j* (*j*=1,...,*n*), связана с затратами. Задача состоит в таком распределении работ по станкам (одна работа выполняется на одном станке), которое соответствует минимизации суммарных затрат c_{ij}.

❖***Пример 8.3.*** $C=(c_{ij})$ – стоимость производства детали *i* на станке *j*, нужно найти распределение станков так, чтобы суммарная стоимость производства была минимальной.

❖***Пример 8.4.*** Транспортная задача. Заданы пункты производства товара, пункты потребления товара. Требуется определить оптимальное взаимно-однозначное соответствие между пунктами производства и пунктами потребления, исходя из матрицы стоимостей перевозок *C* между соответствующими пунктами (т. е. минимизировать суммарную стоимость перевозки).

Здесь работы представляют «исходные пункты», а станки – «пункты назначения». Предложение в каждом исходном пункте равно 1. Аналогично спрос в каждом пункте назначения равен 1. Стоимость «перевозки» (прикрепления) работы *i* к станку *j* равна c_{ij}. Если какую-либо работу нельзя выполнять на некотором станке, то соответствующая стоимость берется равной очень большому числу.

Прежде чем решать такую задачу, необходимо ликвидировать дисбаланс, добавив фиктивные работы или станки в зависимости от начальных условий. Поэтому без потери общности можно положить $m = n$.

Пусть $x_{ij} = 0$, если j-я работа не выполняется на i-м станке, $x_{ij} = 1$, если j-я работа выполняется на i-м станке. Таким образом, решение задачи может быть записано в виде двумерного массива $X = (x_{ij})$. Допустимое решение называется *назначением*. Допустимое решение строится путем выбора ровно одного элемента в каждой строке матрицы $X = (x_{ij})$ и ровно одного элемента в каждом столбце этой матрицы.

✍ **Замечание 1.** Для заданного значения n существует $n!$ допустимых решений.

Математическая модель задачи: минимизировать функцию $F(X) = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \to \min$, при ограничениях:

$$\sum_{j=1}^{n} x_{ij} = 1, \quad i = \overline{1,n}, \tag{8.1}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j = \overline{1,n}, \tag{8.2}$$

$$x_{ij} \in \{0,1\}.$$

Ограничения (8.1) необходимы для того, чтобы каждая работа выполнялась ровно один раз. Ограничения (8.2) гарантируют, что каждому станку будет приписана ровно одна работа.

❖ ***Пример 8.5.*** Для иллюстрации задачи о назначениях рассмотрим таблицу с тремя работами и тремя станками. Стоимость производства детали i на станке j:

$$C = \begin{pmatrix} 5 & 7 & 9 \\ 14 & 10 & 12 \\ 15 & 13 & 16 \end{pmatrix}.$$

Нужно найти распределение станков так, чтобы суммарная стоимость производства была минимальной.

Специфическая структура задачи о назначениях позволяет использовать эффективный метод для ее решения.

✍ **Замечание 2.** Оптимальное решение задачи не изменится, если из любой строки или столбца матрицы стоимостей вычесть произвольную постоянную величину.

Приведенное замечание 2 показывает, что если можно построить новую матрицу $\overline{C}$ с нулевыми элементами, и эти нулевые элементы или их

подмножество соответствуют допустимому решению, то такое решение будет оптимальным. Например:

$$C=\begin{pmatrix}5&7&9\\14&10&12\\15&13&16\end{pmatrix}\begin{matrix}5\\10\\13\end{matrix}\Rightarrow\begin{pmatrix}0&2&4\\4&0&2\\2&0&3\\0&0&2\end{pmatrix}\Rightarrow\overline{C}=\begin{pmatrix}(0)&2&2\\4&0&(0)\\2&(0)&1\end{pmatrix}.$$

Оптимальное назначение: $x_{11}^*=1,\ x_{23}^*=1,\ x_{32}^*=1,$ остальные $x_{ij}^*=0,$ $F(X^*)=c_{11}+c_{23}+c_{32}=5+12+13=30.$

К сожалению, не всегда удается определить решение так просто. Для таких случаев рассмотрим следующий алгоритм.

Венгерский алгоритм

Шаг 1. (Редукция строк и столбцов). Цель данного шага состоит в получении максимально возможного числа нулевых элементов в матрице стоимостей. Для этого из всех элементов каждой строки вычитают минимальный элемент соответствующей строки, а затем из всех элементов каждого столбца полученной матрицы вычитают минимальный элемент соответствующего столбца. В результате получают редуцированную матрицу стоимостей и переходят к поиску назначений.

Шаг 2. (Определение назначений). На этом шаге можно использовать алгоритм поиска «наибольшего паросочетания с матрицей двудольного графа (существуют и другие возможности), если все c_{ij}=0 матрицы $\overline{C}$ заменить на «1», а c_{ij}>0 на «0». Если нельзя найти полного назначения, то необходима дальнейшая модификация матрицы стоимостей, т. е. перейти к шагу 3.

Шаг 3. (Модификация редуцированной матрицы). Для редуцированной матрицы стоимостей:

а) Вычислить число нулей в каждой невычеркнутой строке и каждом невычеркнутом столбце.

б) Вычеркнуть строку или столбец с максимальным числом нулей.

в) Выполнять пункты а) и б) до тех пор, пока не будут вычеркнуты все нули.

г) Из всех невычеркнутых элементов вычесть минимальный невычеркнутый элемент и прибавить его к каждому элементу, расположенному на пересечении двух линий.

Перейти к шагу 2.

✍ **Замечание 3.** Если исходная задача является задачей максимизации, то все элементы матрицы стоимостей следует умножить на (–1) и сложить их с достаточно большим числом так, чтобы матрица не содержала бы

отрицательных элементов. Затем задачу следует решать как задачу минимизации.

❖***Пример 8.6.*** Покажем работу венгерского алгоритма на примере задачи о назначениях со следующей матрицей стоимостей:

$$C = (c_{ij}) = \begin{pmatrix} 2 & 10 & 9 & 7 \\ 15 & 4 & 14 & 8 \\ 13 & 14 & 16 & 11 \\ 4 & 15 & 13 & 19 \end{pmatrix}.$$

Итерация 1

Шаг 1. Редукция строк и столбцов.

Значения минимальных элементов строк 1, 2, 3 и 4 равны 2, 4, 11 и 4, соответственно. Вычитая из элементов каждой строки соответствующее минимальное значение, получим следующую матрицу:

$$\begin{pmatrix} 0 & 8 & 7 & 5 \\ 11 & 0 & 10 & 4 \\ 2 & 3 & 5 & 0 \\ 0 & 11 & 9 & 15 \end{pmatrix}.$$

Значения минимальных элементов столбцов 1, 2, 3 и 4 равны 0, 0, 5 и 0, соответственно. Вычитая из элементов каждого столбца соответствующее минимальное значение, получим следующую матрицу.

$$\overline{C} = \begin{pmatrix} 0 & 8 & 2 & 5 \\ 11 & 0 & 5 & 4 \\ 2 & 3 & 0 & 0 \\ 0 & 11 & 4 & 15 \end{pmatrix}.$$

Шаг 2. Поиск допустимого решения, для которого все назначения имеют нулевую стоимость. Используем алгоритм поиска наибольшего паросочетания. Преобразуем матрицу в матрицу двудольного графа, затем в рабочую таблицу:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} & X & X & X \\ X & & X & X \\ X & X & & \\ & X & X & X \end{pmatrix}.$$

Находим паросочетание:

$$\begin{pmatrix} 1 & X & X & X \\ X & 1 & X & X \\ X & X & 1 & \\ & X & X & X \end{pmatrix}.$$

Это паросочетание не совершенное, т. е. полного назначения нет. Далее переходим на Шаг 3.

Шаг 3. Модификация редуцированной матрицы.

$$\overline{C} = \begin{pmatrix} 0 & 8 & 2 & 5 \\ 11 & 0 & 5 & 4 \\ 2 & 3 & 0 & 0 \\ 0 & 11 & 4 & 15 \end{pmatrix}.$$

а) Число нулей в строках 1, 2, 3 и 4 равно 1, 1, 2 и 1, соответственно. Для столбцов соответствующие величины равны 2, 1, 1 и 1.

б) Максимальное число нулей – по два, содержат строка 3 и столбец 1. Выбираем строку 3 и вычеркиваем все ее элементы горизонтальной линией.

в) Число невычеркнутых нулей в строках 1, 2 и 4 равно 1, 1 и 1, соответственно. Для столбцов соответствующие значения равны 2, 1, 0, и 0. Поэтому необходимо выбрать столбец 1 и вычеркнуть его вертикальной линией. После этого останется только один невычеркнутый нуль – элемент (2,2). Поэтому можно вычеркнуть либо строку 2, либо столбец 2. Вычеркивая строку 2 горизонтальной линией, получаем следующую матрицу:

$$\begin{pmatrix} 0 & 8 & 2 & 5 \\ \cancel{11} & \cancel{0} & \cancel{5} & \cancel{4} \\ \cancel{2} & \cancel{3} & \cancel{0} & \cancel{0} \\ 0 & 11 & 4 & 15 \end{pmatrix}$$

г) Значение минимального невычеркнутого элемента равно 2. Вычитая его из всех невычеркнутых элементов и складывая его со всеми элементами, расположенными на пересечении двух линий, получаем новую матрицу стоимостей:

$$\begin{pmatrix} 0 & 6 & 0 & 3 \\ 13 & 0 & 5 & 4 \\ 4 & 3 & 0 & 0 \\ 0 & 9 & 2 & 13 \end{pmatrix}.$$

Итерация 2

Шаг 2. Выполняя вновь процедуру построения допустимого решения нулевой стоимости, получаем следующее оптимальное решение:

$$\begin{pmatrix} 0 & 6 & (0) & 3 \\ 13 & (0) & 5 & 4 \\ 4 & 3 & 0 & (0) \\ (0) & 9 & 2 & 13 \end{pmatrix}.$$

Оптимальное назначение: $x_{13}^* = 1,\ x_{22}^* = 1,\ x_{34}^* = 1,\ x_{41}^* = 1,$ остальные $x_{ij}^* = 0,\ F(X^*) = 9 + 4 + 11 + 4 = 28.$

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какое паросочетание называется максимальным?
2. Какое паросочетание называется наибольшим?
3. Какие существуют варианты задач о наибольшем паросочетании в графе?
4. Справедливо ли следующее утверждение: наибольшее паросочетание – всегда четное число.
5. Возможно ли, что максимальное паросочетание является и наибольшим?
6. Какими способами можно задать двудольный граф?
7. Справедливо ли следующее утверждение: для двудольного графа матрица смежности и матрица двудольного графа совпадают.
8. Справедливо ли следующее утверждение: алгоритм выбора наибольшего сочетания в двудольном графе с матрицей двудольного графа позволяет найти максимальное паросочетание.
9. Как условие задачи о свадьбах можно представить в виде графа?
10. В каких терминах можно сформулировать теорему Холла?
11. При решении каких задач можно использовать алгоритм выбора наибольшего сочетания в двудольном графе в качестве процедуры?
12. Сформулируйте постановку задачи о назначениях.
13. Сформулируйте постановку задачи о назначении на узкие места.
14. Сколько существует допустимых решений задачи о назначениях с *n* работами и *n* станками?
15. Сформулируйте целевую функцию задачи о назначении.
16. Есть ли отличие при реализации венгерского алгоритма при решении задачи о назначении в случаях минимизации и максимизации целевой функции?
17. Какие задачи можно решать с помощью венгерского алгоритма?

9. АЛГОРИТМЫ СОРТИРОВКИ

Обычным и важным действием при обработке данных является сортировка данных в соответствии с определенным критерием. Природа данных и цели их использования определяют особенности сортировки. Чаще всего приходится иметь дело с числовой или буквенной сортировкой. Предположим, что отношение порядка представляет собой тотальный порядок (*тотальным порядком R* называется такой порядок, при котором для любой пары элементов возможно сравнение: либо $a\ R\ b$, либо $b\ R\ a$, либо и то и другое).

Пусть A – конечное множество элементов с тотальным порядком, который будем обозначать символом «≤». Будем писать $a \le b$ и говорить «*a* меньше или равно *b*» или писать aRb и говорить «*a* связано отношением *R* с *b*». Пусть элементы множества A заданы неупорядоченным списком $a_1, a_2, ..., a_N$ и необходимо отсортировать данный список, т. е. получить упорядоченный список $s_1, s_2, ..., s_N$, а значит, получить список $s_1 < s_2 < ... < s_N$ (если бы множество A было бы множеством частичного, а не тотального порядка, то получить такой список было бы невозможно).

Задача сортировки, казалось бы, носит простой характер, но несмотря на это, она является предметом серьезного и многолетнего научного исследования. Эта задача часто встречается в приложениях и до сих пор осталась интересной.

Задача становится интересной, когда количество упорядочиваемых элементов становится большим. Методы сортировки очень разнообразны. На сегодняшний день самих методов и их модификаций накопилось очень много, ограничимся лишь несколькими принципиально различными. Подробности можно найти в фундаментальной книге Д. Кнута.

9.1. Методы типа вставки

Идея метода: наращивать отсортированную часть последовательности. Сначала взять последовательность из одного первого элемента, она уже «отсортирована». Затем брать очередной элемент и, сравнивая его с предыдущими, переставлять до тех пор, пока он не займет свое место.

Пример. Вставка только одного элемента «5», сортировать в порядке возрастания (неубывания):

3, 6, 8, 10, **5**

3, 6, 8, **5,** 10

3, 6, **5,** 8, 10.

3, **5,** 6, 8, 10

Такой метод имеет трудоемкость порядка n^2. Это легко проверить, построив пример, в котором изначально последовательность задана в порядке убывания, и требуется упорядочить по возрастанию.

9.2. Метод фон Неймана

Первым предложил алгоритм трудоемкости $O(n\log_2 n)$ американский математик Дж. фон Нейман. Предложенная им сортировка основана на операции слияния двух упорядоченных массивов.

Первоначально сортируемый массив представляется в виде *n* «отсортированных» массивов, каждый длины 1. Сольем их попарно, получив примерно $n/2$ массивов длины 2, потом повторим процесс. За $\lceil\log_2 n\rceil$ шагов получим один массив длины *n*. Так как трудоемкость каждого шага равна *n*, суммарная трудоемкость сортировки по этому методу равна $n\log_2 n$.

❖ ***Пример 9.1.*** Отсортировать 11 элементов в порядке возрастания: (4, 8, 7, 5, 3, 11, 1, 9, 6, 2, 10). Данная последовательность отсортируется за 4 итерации.

	4	8	7	5	3	11	1	9	6	2	10
Итерация 1:	4	8	5	7	3	11	1	9	2	6	10
Итерация 2:	4	5	7	8	1	3	9	11	2	6	10
Итерация 3:	1	3	4	5	7	8	9	11	2	6	10
Итерация 4:	1	2	3	4	5	6	7	8	9	10	11

Этот метод очень прост и удобен в реализации, но операция слияния требует дополнительной памяти – еще одной копии массива для хранения результата слияния. Кроме того, можно отметить еще один недостаток метода – «универсальную трудоемкость», т. е. для почти отсортированных массивов тратится столько же времени, как и в совсем неотсортированном случае. Существуют модификации метода фон Неймана, в которых сортируемый массив рассматривается как последовательность упорядоченных частей, их длина не фиксирована, а определяется только сохранением порядка. Это позволяет избавиться от «универсальной трудоемкости».

9.3. Метод быстрой сортировки

Английский математик Ч. Хоар предложил эффективный метод быстрой сортировки (Quicksort). Он основан на простой идее. Основная операция его метода – взять один из элементов массива и поставить его на законное место, при этом так, чтобы элементы, предшествующие ему в упорядочении, размещались до него, а следующие за ним – после. После этого получим два массива, *до* и *после*, каждый из которых можно упорядочить тем же способом.

На каждом шаге сортируемый массив просматривается в двух направлениях от начала и от конца. С одной стороны стоит *разделяющий элемент*, место которого мы ищем, с другой стороны – *проверяемый элемент*, который сравнивается с разделяющим. Первоначально разделяющий – первый элемент массива, а проверяемый – последний. Если на каком-то шагу эти два элемента упорядочены неправильно, они меняются местами и вместе с этим изменяются роли сторон. В качестве проверяемого элемента на следующем

шаге берется следующий по направлению к разделяющему. Когда проверяемый и разделяющий совпадут, одна итерация сортировки закончена. После чего разделяющий элемент стоит на своем месте и нужно сортировать аналогично массивы «*до*» и *«после*» него.

❖ ***Пример 9.2.*** Отсортировать элементы в порядке возрастания: (4, 8, 7, 5, 3, 11, 1, 9, 6, 2, 10).

Проведем сортировку на массивы «*до*» и *«после*» относительно первого элемента 4, т. е. он будет разделяющим элементом на первой итерации. На противоположном конце массива стоит проверяемый элемент 10. Так как расположение 4 – 10 правильное, переходим к проверке следующего элемента, 2. Расположение 4 – 2 неправильное. Переставим элементы местами:

(*2*, 8, 7, 5, 3, 11, 1, 9, 6, **4**, *10*).

Теперь роли концов массива переменились, проверяем 8 – 4. Это нарушение, и снова делаем перестановку и сменяем роли концов массива:

(*2*, **4**, 7, 5, 3, 11, 1, 9, 6, *8*, *10*).

При этом зона просмотра все время сужается. Курсивом выделены элементы отсортированных массивов «*до*» и *«после*». Продолжаем:

(*2*, **4**, 7, 5, 3, 11, 1, 9, *6*, *8*, *10*);
(*2*, **4**, 7, 5, 3, 11, 1, *9*, *6*, *8*, *10*);
(*2*, *1*, 7, 5, 3, 11, **4**, 9, *6*, *8*, *10*);
(*2*, *1*, **4**, 5, 3, 11, *7*, 9, *6*, *8*, *10*);
(*2*, *1*, **4**, 5, 3, *11*, *7*, 9, *6*, *8*, *10*);
(*2*, *1*, *3*, 5, **4**, *11*, *7*, 9, *6*, *8*, *10*);
(*2*, *1*, *3*, **4**, *5*, *11*, *7*, 9, *6*, *8*, *10*).

Закончена первая итерация метода. Аналогично необходимо отсортировать массив «*до*»: (2, 1, 3) и массив «*после*» (5, 11, 7, 9, 6, 8, 10).

Эффективный вариант этого метода, в котором разделяющий элемент выбирается как средний по значению из трех элементов, например, из крайних и среднего по положению. Это позволяет избежать напрасной итерации, в случае если разделяющий элемент оказывается минимальным или максимальным в массиве. Кроме того, причиной резкого снижения эффективности этого метода сортировки может стать наличие большого числа равных элементов. Это затруднение можно легко преодолеть, если в операции сравнения элементов (в случае их равенства) поочередно вырабатывать значения «больше» и «меньше». Тогда равные элементы будут распределяться по интервалам примерно поровну.

9.4. Метод дерева сортировки

В процедурах сортировки будут использоваться двоичные деревья. Рассмотрим двоичные деревья более подробно.

Пусть T – полное двоичное дерево с высотой, превышающей ноль, и с корнем v^*. Удаление v^* и двух его инцидентных граней порождает два

непересекающихся двоичных дерева (двоичный лес), чьи корни – это вершины дерева T первого уровня. Они называются *левым поддеревом* и *правым поддеревом* корня v^*. Корни этих поддеревьев называются *левым* и *правым ребенком* корня v^*, а удаленные грани называются соответственно *левой* и *правой ветками* вершины v^*. Если T не является полным двоичным деревом, то его левое или правое поддерево может быть пустым.

Двоичное дерево включает совокупность трех множеств (L, S, R), где L и R – двоичные (или пусты) деревья, а S синглетное (однокомпонентное) множество. Единственным элементом множества S является корень. Элементы L и R называются, соответственно, *левыми* и *правыми поддеревьями корня*.

Это определение рекурсивно, потому что оно определяет двоичное дерево в терминах компонент множеств L, S и R, два из которых сами являются двоичными деревьями. Таким образом, каждое из множеств L и R, если оно непустое, определяется посредством трех множеств (L', S', R') и так далее. Такой способ определения двоичных деревьев оказывается чрезвычайно плодотворным для компьютерного представления двоичных деревьев. Следующий пример иллюстрирует, как можно распутать рекурсивное определение двоичного дерева, чтобы получить обычную диаграмму.

❖ ***Пример 9.3.*** Рассмотрите рекурсивно определенное двоичное дерево (L, $\{v^*\}$, R), где

$$L = (L_1, \{v_1\}, R_1), \text{ и } R = (L_2, \{v_2\}, R_2).$$
$$L_1 = (L_3, \{v_3\}, R_3), \text{ и } R_1 = (\varnothing, \{v_4\}, R_4).$$
$$L_2 = (L_5, \{v_5\}, R_5), \text{ и } R_2 = (\varnothing, \{v_6\}, \varnothing).$$
$$L_3 = (\varnothing, \{v_7\}, \varnothing), \text{ и } R_3 = (L_8, \{v_8\}, R_8).$$
$$R_4 = (\varnothing, \{v_9\}, \varnothing).$$
$$L_5 = (\varnothing, \{v_{10}\}, \varnothing), \text{ и } R_5 = (\varnothing, \{v_{11}\}, \varnothing).$$
$$L_8 = (\varnothing, \{v_{12}\}, \varnothing), \text{ и } R_8 = (\varnothing, \{v_{13}\}, \varnothing).$$

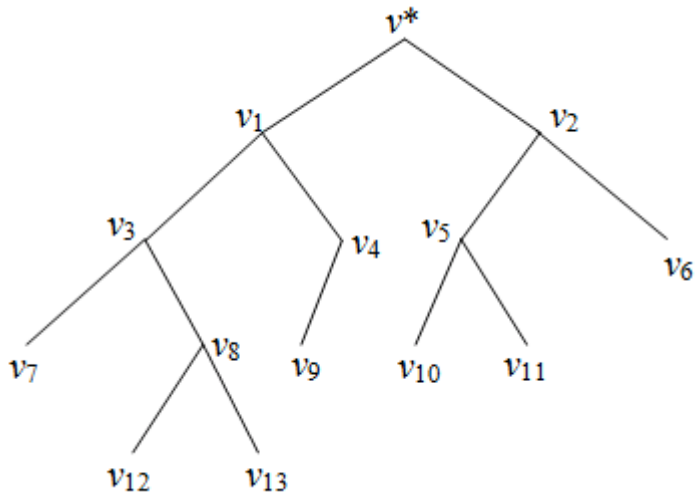


Рис. 9.1. Пример двоичного дерева

Триплексы множеств, определяющих дерево, были сгруппированы в соответствии с уровнями их корней. На рис. 9.1 показана обычная диаграмма для данного двоичного дерева. Отметим, что данное двоичное дерево не является совершенным, так как вершина v_4 имеет только одного ребенка, а также и потому, что оно имеет листовые вершины на различных уровнях (например, втором и третьем).

Процедура сортировки с помощью дерева использует специальный вид двоичного дерева. Это двоичное дерево, у которого множество вершин полностью упорядочено. Для каждой вершины v дерева, вершины в его левом поддереве меньше или равны вершине v, и сама вершина v меньше или равна вершинам правого поддерева.

Введем понятие *тотального порядка R* – это такой порядок, при котором для любой пары элементов возможно сравнение: либо $a\ R\ b$, либо $b\ R\ a$, либо и то и другое.

Дерево сортировки – это двоичное дерево T такое, что: множество вершин V тотально упорядочено (с отношения порядка, обозначаемым символом $\leq$), и для каждой вершины $v \in V$, $w_L \leq v$ для каждой вершины w_L, находящейся в левом поддереве вершины v, и $v \leq w_R$ для каждой вершины w_R, находящейся в правом поддереве вершины v.

Процедура сортировки по методу дерева происходит в два этапа. Предположим, даны элементы множества A в качестве неупорядоченного (несортированного) списка $a_1, a_2, ..., a_N$. На первом этапе список используется, чтобы сконструировать дерево сортировки, у которого вершинами являются элементы множества A и корнем является элемент a_1 – первый элемент несортированного списка. Второй этап позволяет получить сортированный список на основании дерева сортировки.

Этап 1. Последовательность деревьев сортировки получается путем добавления каждого элемента списка в определенное место (очередь) в качестве листовой вершины. Множество вершин окончательного (заключительного) дерева сортировки содержит все элементы исходного списка.

Чтобы вставить элемент x, сначала сравниваем его с корнем 6. Если $x \leq 6$ идем по левому поддереву; иначе – идем по правому поддереву. Повторяем сравнение x с корнем нового дерева. На некоторой стадии сложится такая ситуация, при которой у нас не будет ни левого, ни правого поддерева, по которому можно было бы идти, и тогда следует добавить x в качестве левого или правого ребенка. На этом заканчивается первый этап метода дерева сортировки.

Этап 2. Используем дерево сортировки, чтобы получить отсортированный список. Так как каждая вершина в левом поддереве корня меньше или равна вершине корня, то такая вершина должна быть внесена в список непосредственно перед корнем. Аналогично все вершины, находящиеся в правом от корня поддереве, должны быть внесены в список непосредственно после корня. Важно понять, что и левые и правые поддеревья корня

самостоятельно, сами по себе, также являются деревьями сортировки, и поэтому их левые и правые поддеревья также являются деревьями сортировки, и так далее вниз через дерево.

Требуемая процедура внесения в список может быть описана рекурсивно следующим образом:

1) вносим в список элементы из левого поддерева корня;

2) вносим в список непосредственно сам корень;

3) вносим в список элементы из правого поддерева корня.

Алгоритм дерева сортировки

1. Для множества, состоящего всего лишь из одного элемента a_1 (n=1), a_1 есть корень T_1; – это дерево сортировки с одним единственным элементом a_1, который является вершиной этого дерева; увеличиваем n до n+1.

2. Если $n > N$, тогда конец. Иначе сравниваем a_n с корнем.

Если $a_n \leq$ корня, то переходим к левому поддереву.

Если левое поддерево пусто, то добавляем a_n в качестве левого ребенка корня, чтобы формировать следующее дерево сортировки T_n; увеличиваем n до n+1 и повторяем шаг 2; иначе повторяем шаг 2, используя левое поддерево.

В противном случае переходим к правому поддереву.

Если правое поддерево пусто, тогда добавляем a_n в качестве правого ребенка корня, чтобы образовать следующее дерево сортировки T_n; n:=n+1 и повторяем шаг 2; в противном случае повторяем шаг 2, используя правое поддерево.

У алгоритма рекурсивный характер. Чтобы проделать шаг 2 для данного (отдельного) поддерева, может потребоваться проделать шаг 2 непосредственно для некоторых других меньших поддеревьев.

❖***Пример 9.4.*** Дан начальный список – (6, 2, 9, 4, 15, 1, 12, 7, 20, 10, 3, 11), и требуется его отсортировать по возрастающей. Первый элемент 6 определен в качестве корня. Так как 2 ≤ 6, то создаем левую от корня ветку с вершиной 2 на ее конце.

Затем, так как 9>6, то создаем правую для корня 6 ветку и размещаем 9 в ее конце.

Отсортируем следующий элемент 4. Для этого сначала сравним его с корнем 6. Так как 4≤6, то необходимо разместить 4 в левое для корня 6 поддерево. Таким образом, рассматриваем левое поддерево, у которого корень – 2. Теперь 4>2, поэтому создаем правую ветку для вершины 2 и на конце этой ветки размещаем вершину 4.

При вставке следующего элемента 15 руководствуемся следующими соображениями: 15>6, поэтому идем по правому для корня 6 поддереву к вершине – 9; 15>9, поэтому создаем правую ветку для вершины 9 и размещаем 15 на втором уровне вершин. Продолжаем процесс первого этапа. Получим

последнее дерево, содержащее все элементы исходного списка, изображенное на рис. 9.2.

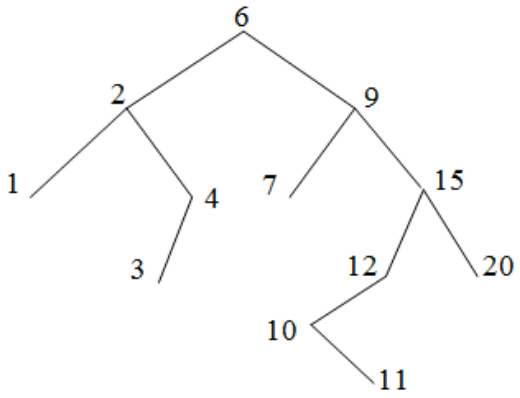


Рис. 9.2. Дерево сортировки. Этап 1, пример 9.4

Для дерева сортировки, представленного на рис. 9.2, левым поддеревом корня является дерево, изображенное на рис. 9.3.

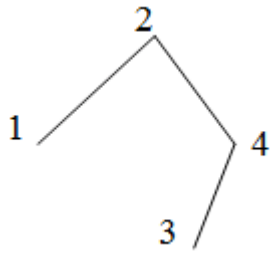


Рис. 9.3. Левое поддерево сортировки, пример 9.4

Чтобы обработать дерево сортировки по процедуре внесения вершин дерева в список (см. рис. 9.3), нужно составлять список элементов, в который сначала помещают элементы дерева сортировки, находящиеся в его левом поддереве (в данном случае первым элемент (1)), далее вносится корневой элемент (2); затем вносятся в список элементы, находящиеся в правом поддереве.

Чтобы вносить в список элементы правого поддерева, нужно снова выполнить описанные три шага: внести в список элементы из левого поддерева (3), внести в список корень (4), и затем внести в список элементы из правого поддерева (в данном случае таких просто нет). Описанный алгоритм дает список 1, 2, 3, 4, и на этом заканчивается первая стадия для главного дерева.

Вторая стадия начинается с внесения в список элемента (6) – корня основного дерева сортировки. Далее необходимо внести в список элементы правого поддерева по аналогии с описанным алгоритмом. Это требует выполнения наших трех основных шагов еще несколько раз. Окончательный список в результате сортировки будет (1, 2, 3, 4, 6, 7, 9, 10, 11, 12, 15, 20).

9.5. Метод «сортировка кучи»

Алгоритм «сортировка кучи» (Heapsort), разработал Дж. Уильямс. Используется специальный вид двоичного дерева, называемого «кучей», и в

данном случае вершины двоичного дерева – элементы списка, который нужно сортировать. «Форма» кучи выбирается такой, чтобы она имела минимальную высоту при заданном числе имеющихся вершин. Это достигается в результате такого подхода застройки дерева, при котором, если отбросить самый высокий уровень дерева, то остальная часть дерева одновременно окажется и полной, и совершенной. Кроме того, листовые вершины на самом высоком уровне оказываются расположенными на диаграмме максимально далеко слева.

Ниспадающая куча – это двоичное дерево высоты h такое, что:

1) все листовые вершины расположены на уровнях (h – 1) или h;

2) листовые вершины на уровне h оказываются расположенными на диаграмме максимально далеко слева (это имплицирует то, что любая листовая вершина на уровне (h–1) является расположенной максимально далеко справа. Т. е. поддерево, полученное из исходного в результате удаления вершин уровня h и их инцидентных граней, является совершенным двоичным деревом);

3) множество вершин имеет тотальный порядок, обозначаемый как $\leq$;

4) если «q» является родителем «p», то $p \leq q$, т. е. ребенок «p» меньше или равен родителя «q» относительно заданного тотального порядка.

Процесс сортировки кучи состоит из двух этапов. Первый этап – куча создается из исходного, неотсортированного списка, второй этап – отсортированный список получается из данной кучи. Выделим каждый этап процедуры отдельно.

Этап 1. Создание кучи

Создание кучи из исходного (неотсортированного) списка $a_1,a_2,...,a_n$ может быть получено на основании следующего алгоритма.

Алгоритм создания кучи

1. Множество a_1 – корень.

2. Добавляем следующий элемент таким (единственным) образом, чтобы дерево удовлетворяло условиям (1) и (2) из определения кучи.

3.Если необходимо, восстанавливаем кучу по методу «пузырькового» подъема новой вершины вдоль дерева.

4.Повторяем шаги 2 и 3 для каждого элемента списка.

Так же, как и в случае дерева сортировки, будем предполагать, что исходный (неотсортированный) список a_1, a_2, ..., a_N задан. Множество, состоящее из одного элемента a_1 (одинокий корень), представляет собой тривиальный случай. Предположим, что элементы a_1, a_2, ..., a_{k-1} уже были сформированы в виде кучи, и необходимо добавить следующий элемент a_k. Имеется единственная позиция, при которой, добавляя a_k, выполняются условия (1) и (2) определения кучи; это либо ближайшая правая позиция от самой правой вершины на самом высоком уровне, либо, если самый высокий уровень полностью укомплектован (совершенен), крайняя левая позиция на новом более высоком уровне.

Однако добавление a_k в такое положение еще не гарантирует образование кучи. Например, предположим, элемент a_k=17 добавлен к куче, представленной на рис. 9.4, *а*. Рассмотрим результирующее двоичное дерево рис. 9.4, *б*. Это новое дерево уже не куча, так как 17 больше, чем родитель 12. В таком случае новая вершина должна поменяться местами со своим родителем; этот результат приводит к образованию дерева, представленного на рис. 9.4, *в*. Однако такая замена все еще не решила до конца вопрос построения кучи, а только переместила проблему вверх по дереву, так как вершина 17 все еще больше, чем ее новый родитель 15. Поэтому необходимо поменять местами вершины 15 и 17, в результате чего, наконец, образуется куча (рис. 9.4, *г*). В этом процессе восстановления кучи, можно сказать, что вновь добавленная вершина, подобно пузырьку воздуха в жидкости, поднимается вверх по дереву.

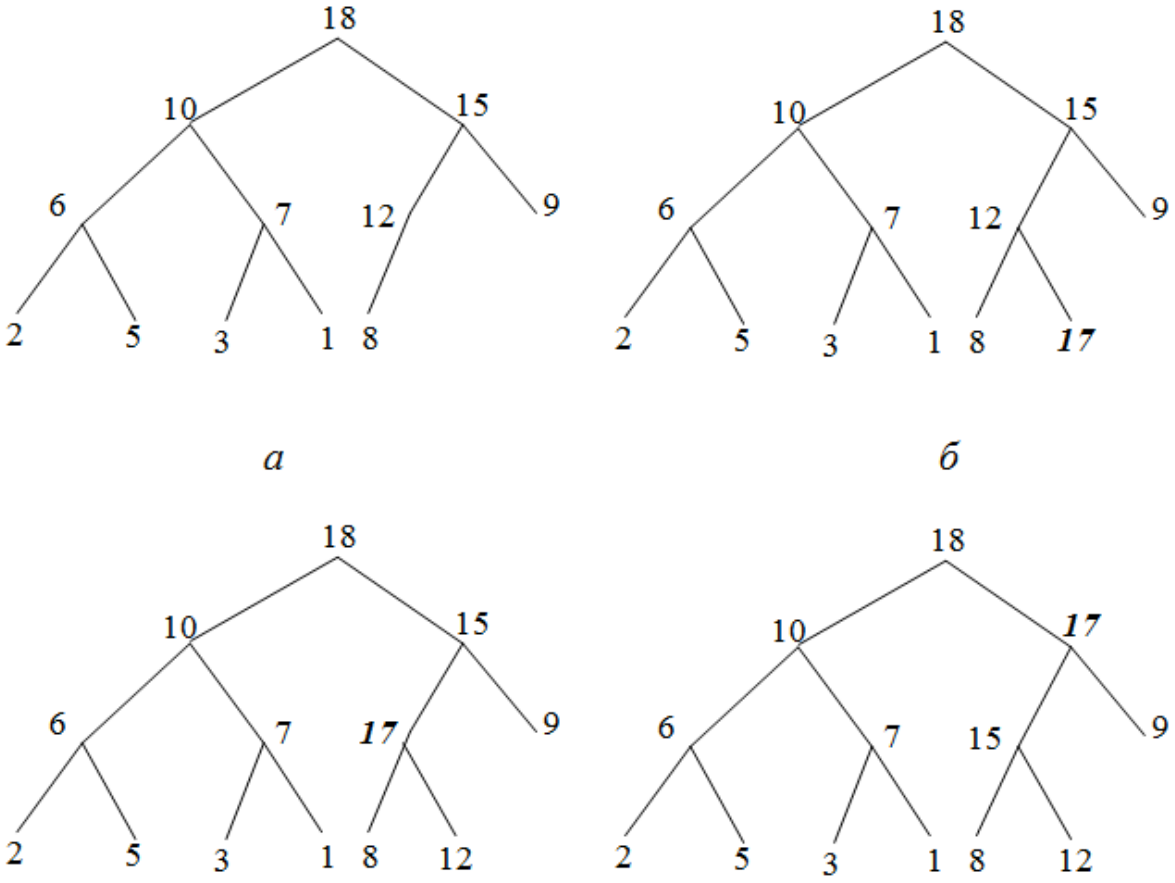


Рис. 9.4. Добавление элемента «17» к куче

Этап 2. Получение упорядоченного (отсортированного) списка на основании кучи

Процесс преобразования кучи в новое дерево, рассмотренный в примере, изображенном на рис. 9.4, может быть описан на основе следующего алгоритма.

Алгоритм преобразования кучи

1. Обменять корень с последней вершиной «*и*», т. е. с самой правой вершиной на самом высоком уровне. Зафиксировать положение (новой) последней вершины. (Под фиксацией положения вершины подразумевается, что зафиксированная вершина игнорируется при всех последующих манипуляциях дерева. В примере, на рис. 9.6, обозначим зафиксированные вершины кружками).
2. Восстановить дерево (за исключением зафиксированных вершин) до кучи, путем осаждения вниз нового корня «*и*» следующим образом.
   2.1. Поменять местами корень «*и*» с самым большим из его двух детей.
   2.2. Перейти к поддереву, меняя каждый раз вершину «*и*» с самым большим из ее детей. Если это поддерево имеет высоту 0 (при этом все зафиксированные вершины, считаются отсутствующими), то переходим к шагу 3; в противном случае повторяем шаг 2.1 для этого поддерева.
3. Повторять шаги 1 и 2 (игнорируя все зафиксированные вершины) пока все вершины не окажутся зафиксированными.

❖***Пример 9.5.*** Предположим, что куча, представленная на рис. 9.5, *а*, была получена из исходного (неотсортированного) списка чисел на основании обычного отношения порядка ≤. Нам нужна процедура, которая из кучи, представленной на рис. 9.5, *а*, создаст дерево, представленное на рис. 9.5, *б*. Отсортированный список (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) может тогда быть прочтен в результате прохождения по дереву сверху донизу и слева направо.

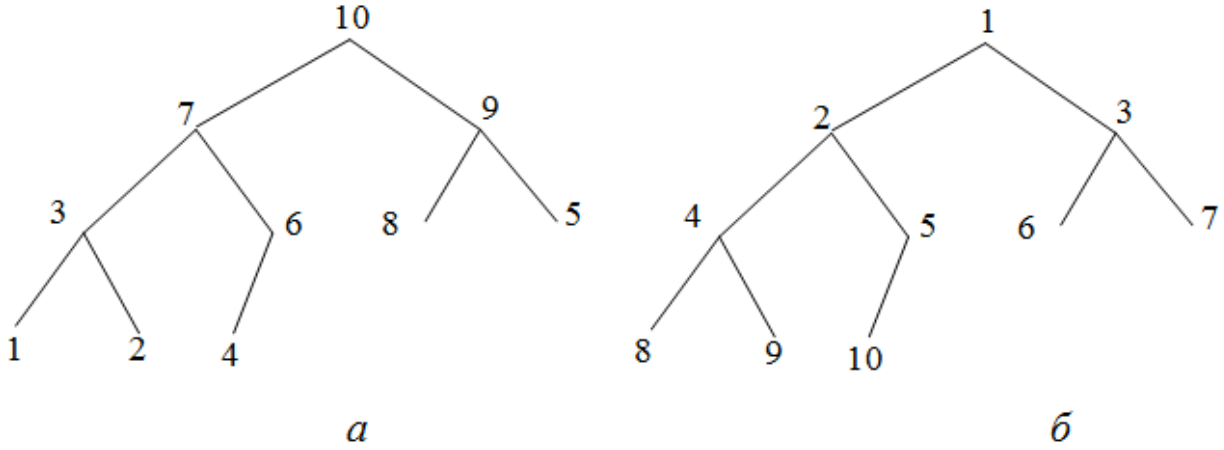


Рис. 9.5. Двоичные деревья, пример 9.5:
а – куча; *б* – отсортированный список

Корень кучи – это максимальный элемент, который поэтому должен быть помещен в конце списка. Поменяем этот корень местами с последней вершиной в куче (т. е. с самой правой вершиной на самом высоком уровне). На рис. 9.5, *а* необходимо поменять местами числа 10 и 4. Это, конечно, разрушает кучу, ибо самая большая вершина теперь становится листом. Так как теперь число 10 находится в желательной позиции дерева, то при дальнейших преобразованиях это число будет игнорироваться. Даже игнорируя это число (число 10), остаток

дерева – все еще не куча, потому что наш новый корень 4 – меньше, чем каждый из его детей. Таким образом, необходимо восстановить кучу, помня о том, что вершина 10 уже находится в своей окончательной позиции, и ее никуда не следует перемещать. Итак, чтобы восстановить кучу, надо корень 4 опустить вниз по дереву, последовательно обменивая число 4 с большим из его детей. В нашем примере сначала меняем местами корень 4 с его правым ребенком 9 и далее с его новым левым ребенком 8. Преобразования приводят к такому преобразованию дерева, при котором все его вершины, обведенные в кружочек, находятся в своем окончательном положении, а поддерево с вершинами, не обведенными в кружочек, представляет собой элемент множества куч.

Итак, повторяем процесс, игнорируя вершину 10, обведенную в кружочек, которая уже находится в своем окончательном положении. На каждой стадии рассматриваем только часть дерева, которая является кучей; т. е. та часть дерева, которая состоит из вершин, не обязательно находящихся в своих окончательных положениях (это вершины, не обведенные кружочками). Последовательные стадии процесса представлены на рис. 9.6.

✍ **Замечание.** Нельзя сказать, что какой-либо один метод сортировки во всех отношениях превосходит (лучше) все остальные методы. Из алгоритмов метод «дерево сортировки» и «кучи», по всей вероятности, первый окажется более предпочтительным для маленьких наборов данных, в то время как метод «кучи» – для больших наборов данных. С теоретической точки зрения метод «кучи» является быстрым алгоритмом. На практике другие алгоритмы, например, метод «быстрой сортировки», оказываются в среднем более эффективными, чем метод «кучи», хотя на некоторых наборах данных метод «быстрой сортировки» может работать существенно медленнее, чем метод «кучи сортировки». Таким образом, каждый из них имеет свою область применения. Метод вставки очень прост, и при малых размерах сортируемого массива, а также при ожидаемых небольших нарушениях порядка можно не беспокоиться о более эффективном методе. Наиболее практична быстрая сортировка.

Различные факторы влияют на выбор вида процесса сортировки для конкретного заданного приложения. К числу таких факторов относятся: размер данных входа, размер доступной памяти. При сортировке данных, которые размещены во внешней памяти, наиболее существенно количество обращений к внешней памяти для чтения или записи информации – каждое такое обращение расходует значительно больше времени, чем упорядочивание фрагмента данных, находящегося в оперативной памяти.

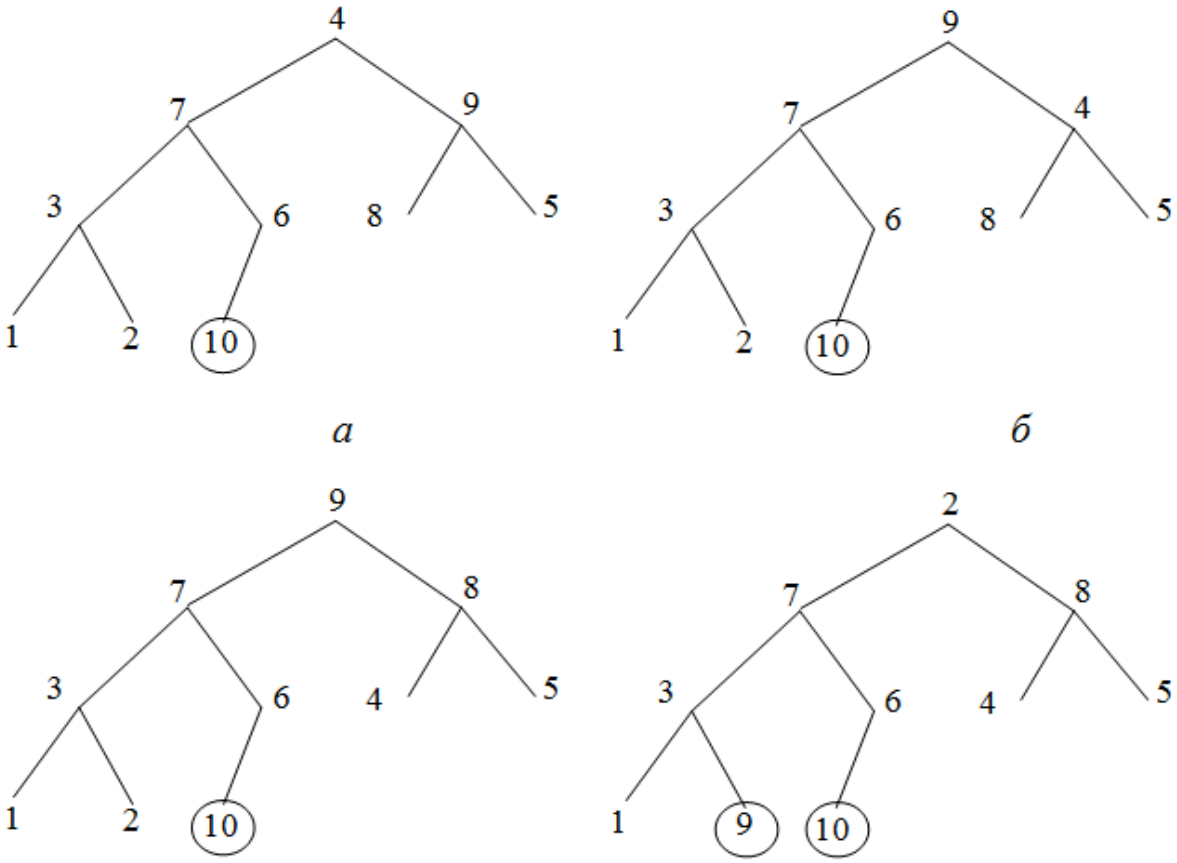


Рис. 9.6. Процесс преобразование кучи

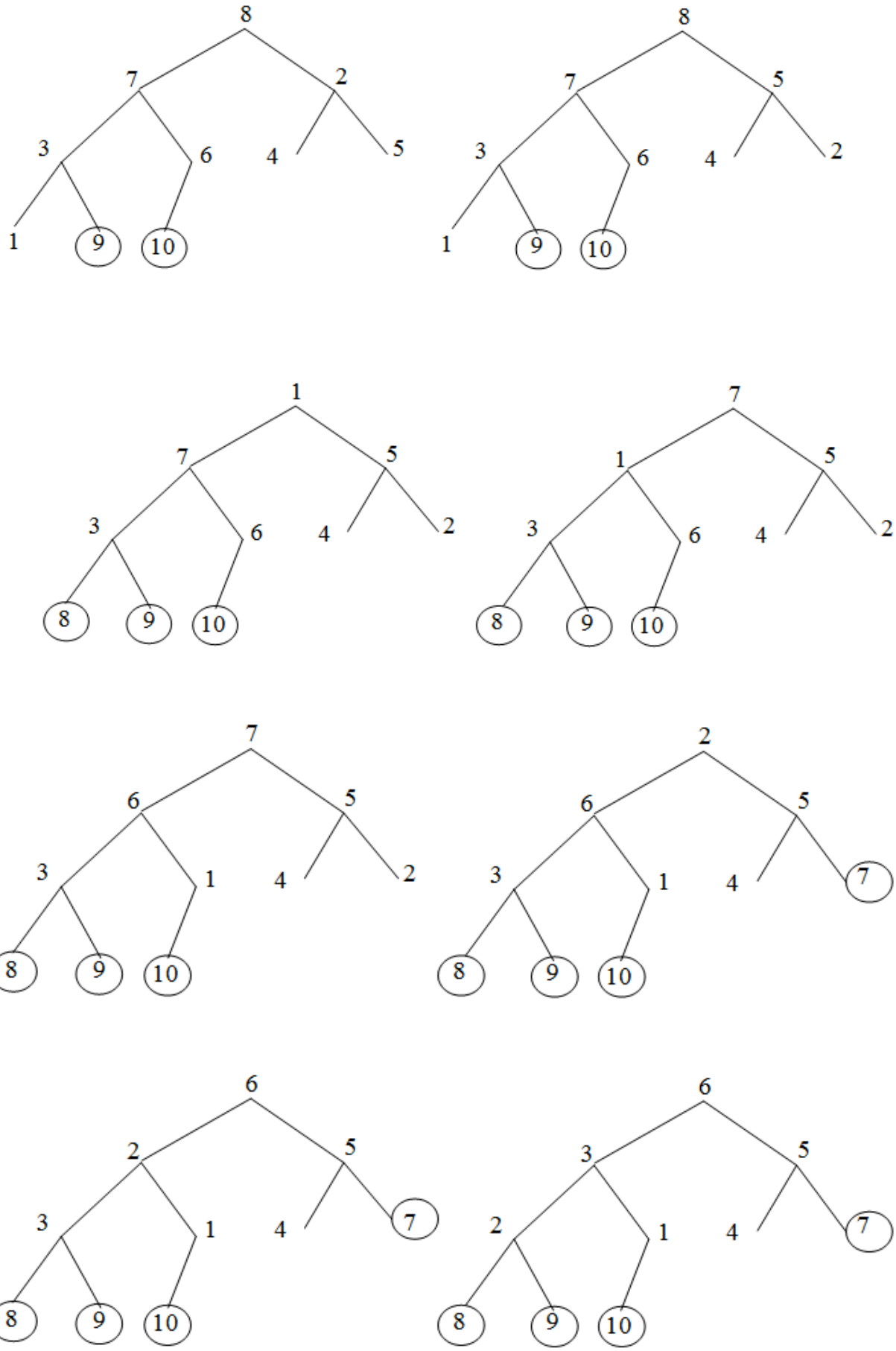


Рис. 9.6. Процесс преобразование кучи (продолжение)

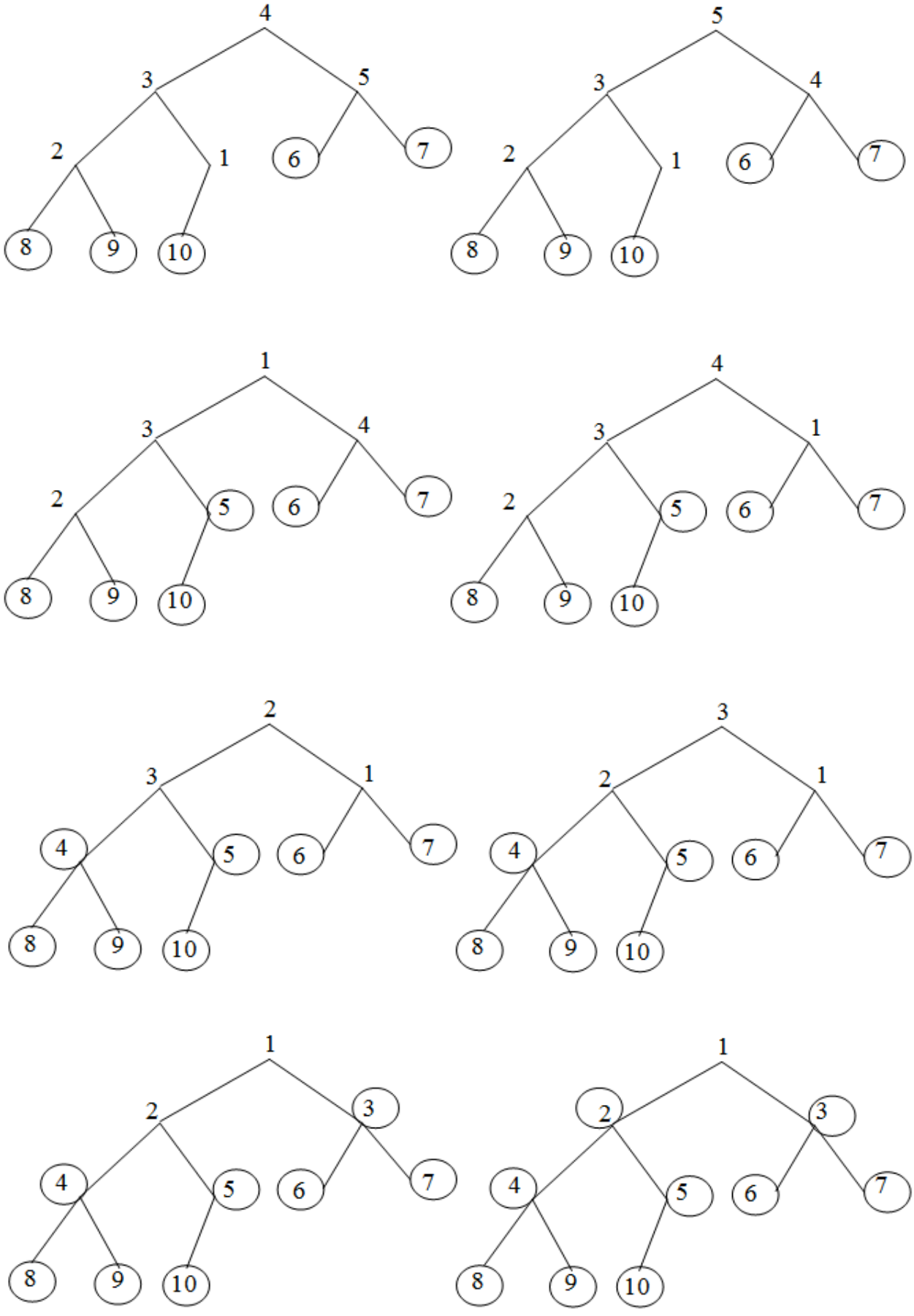


Рис. 9.6. Процесс преобразование кучи (окончание)

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем заключается идея метода вставки для сортировки данных?
2. Какая трудоемкость метода вставки?
3. Придумайте пример для алгоритма вставки, в котором бы достигалась максимальная трудоемкость.
4. Какие недостатки можно отметить для алгоритма сортировки по методу фон Неймана?
5. Как можно модифицировать метод фон Неймана для устранения недостатков?
6. В чем заключается основная операция алгоритма быстрой сортировки?
7. Что такое «разделяющий элемент» в алгоритме быстрой сортировки?
8. Какие можете предложить варианты выбора разделяющего элемента в алгоритме быстрой сортировки?
9. Зависит ли эффективность алгоритма быстрой сортировки от разделяющего элемента?
10. Какое дерево используется в методе дерева сортировки?
11. Что такое дерево сортировки?
12. Назовите количество этапов и в чем они состоят в методе дерева сортировки.
13. Верно ли, что в методе сортировки кучи используется двоичное дерево максимальной высоты?
14. Дайте определение ниспадающей кучи.
15. Какие факторы могут влиять на эффективность алгоритмов сортировки?

ЗАКЛЮЧЕНИЕ

В данном учебном пособии читатель познакомился с комбинаторными алгоритмами, которые можно в полной мере назвать классическими. Большинство из них было разработано в 50-70-х гг. XX в. Это было связано с появлением вычислительной техники, но в то время ее возможности были сильно ограничены. Современная вычислительная техника и алгоритмическое обеспечение в значительной степени позволяют успешно решать комбинаторные задачи. Хотя на сегодняшний день существует большой круг задач (пример таких задач: NP-трудные большой размерности), для которых известные комбинаторные алгоритмы требуют слишком большого вычислительного времени. В связи с этим на практике для подобных задач используют приближенное решение, а алгоритмы точного решения представляют больше теоретический интерес. Совершенствование и модификация классических комбинаторных алгоритмов направлена на снижение вычислительной трудности, увеличение точности при приближенном решении и расширение возможностей учета реальных практических ограничений. Кроме того, многие комбинаторные алгоритмы используются в качестве решения подзадач на различных этапах решения научных и практических проблем, встречающиеся в широких областях: в информатике, электронике, многочисленных вопросах оптимизации.

Для главы 1 данного учебного пособия использовалась литература [6], [7], [9].

Глава 2 основана на [1], [10].

В третьей главе определения – [4], [6], рассмотренные задачи о кратчайших путях в графах и алгоритмы их решения основаны на [10]. Алгоритмы поиска остовных деревьев – [4], [10].

Глава 4 посвящена алгоритмам решения задачи коммивояжера. Использована книга [4] для представления алгоритмов «приблизительного» решения. Точный алгоритм ветвей и границ подробно описан в [9].

Задачи сетевого планирования главы 5 представлены в классической монографии [5].

Глава 6 основана на [4], [10].

Глава 7 о непересекающихся цепях и разделяющих множествах основана на [6], [10]. Кроме того, схема доказательства теоремы Форда-Фалкерсона на основе теоремы Менгера в ориентированной реберной форме изложена в [6].

Паросочетания, варианты задач и алгоритмы их решения, представленные в главе 8, описаны в [6], алгоритм Куна (Венгерский алгоритм) в [2].

Глава 9, посвященная алгоритмам сортировки, основана на [3], [4], [8]. Методы типа вставки, фон Неймана, быстрой сортировки представлены в [3], [8], а методы дерева сортировки и кучи – в [4].

Данное учебное пособие предназначено для первого знакомства с предметом. В нем много внимания уделено терминологии и сопоставлению подходов к решению одной и той же задачи. Многие из рассматриваемых

вопросов заслуживают отдельного и глубокого рассмотрения, а здесь приведено лишь краткое их изложение.

Авторы выражают глубокую признательность профессору УГАТУ Бронштейну Ефиму Михайловичу за научное рецензирование данного пособия и ценные замечания.

СПИСОК ЛИТЕРАТУРЫ

1. *Абрамов С. А.* Лекции о сложности алгоритмов. М.: МЦНМО, 2009. 256 с.

2. *Грызина Н. Ю.*, *Мастяева И. Н.*, *Семенихина О. Н.* Математические методы исследования операций в экономике: учебно-методический комплекс. М.: Изд. центр ЕАОИ, 2008. 204 с.

3. Кнут Д. Э. Искусство программирования. Т. 3. Сортировка и поиск. М.: Издат. дом «Вильямс», 2012. 824 с.

4. Логинов Б. М. Лекции и упражнения по курсу «Введение в дискретную математику». Калуга: изд-во Калужского филиала МГТУ, 1998. 423 с.

5. Мухачева Э. А., *Рубинштейн Г. Ш.* Математическое программирование. Новосибирск: Наука, 1987. 237с.

6. *Новиков Ф. А.* Дискретная математика для программистов: учебник для вузов. 3-е изд. СПб.: Питер, 2008. 384 с.

7. *Панюкова Т. А.* Комбинаторика и теория графов: учебное пособие. М.: Книжный дом «ЛИБРОКОМ», 2012. 208 с.

8. Романовский И. В. Дискретный анализ: учебное пособие для студентов, специализирующихся на прикладной математике и информатике. СПб.: Невский Диалект; БХВ-Петербург, 2008. 336 с.

9. *Сигал И. Х.*, *Иванова А. П.* Введение в прикладное дискретное программирование. М.: Физматлит, 2007. 304 с.

10. *Ху Т. Ч.*, *Шинг М. Т.* Комбинаторные алгоритмы. Перевод с англ. В. Е. Алексеева, Н. Ю. Золотых и др. – Нижний Новгород. Изд-во Нижегородского госуниверситета им. Н. И. Лобачевского, 2004. 329 с.