ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение высшего
профессионального образования
«Уральский государственный университет им. А.М. Горького»

Математико-механический факультет

Кафедра математической экономики
Кафедра алгебры и дискретной математики

КОМБИНАТОРНЫЕ АЛГОРИТМЫ

УЧЕБНОЕ ПОСОБИЕ

Авторы:

Асанов Магаз Оразкимович, кандидат физико-математических наук, доцент, заведующий кафедрой математической экономики УрГУ им А.М.Горького

Расин Вениамин Вольфович, кандидат физико-математических наук, доцент кафедры алгебры и дискретной математики УрГУ им А.М.Горького

Екатеринбург
2008

Предисловие

Основой для данного учебного пособия послужили лекции, которые читались авторами для студентов математико-механического факультета Уральского государственного университета им. А. М. Горького, обучающихся по специальностям «Математика, прикладная математика», «Математика, компьютерные науки» и «Компьютерная безопасность».

В книге приводятся алгоритмы дискретной оптимизации на графах и сетях. Материал, посвященный алгоритмам, содержит достаточно строгое их обоснование. Разумеется, при построении и анализе алгоритмов, используются основные теоретико-графовые понятия и факты. Подбор тем, поднятых в книге, во многом определен вкусами авторов. Нам хотелось представить семейство алгоритмов дискретной оптимизации, наиболее часто используемых программистами. Мы стремились привести главные достижения, не останавливаясь на мелочах и не углубляясь в детальный обзор результатов по обсуждаемым темам.

В качестве основной литературы отметим книги [3], [6], [24], [29], [46], [49], [52], [55]. Для удобства читателям приводится достаточно полный список литературы по рассматриваемому предмету, содержащий книги, которые были опубликованы на русском языке. Мы обращаем внимание читателя на фундаментальные книги [59] – [63], русских переводов которых, к сожалению, не имеется.

Нами используется терминология, наиболее распространенная в математической литературе как за рубежом, так и в России. Теоремы и предложения нумеруются в книге двумя числами: первое – номер главы, а второе – порядковый номер утверждения данного типа в указанной главе. Аналогично нумеруются и алгоритмы.

Заметим, что при формализованной записи алгоритмов мы старались не использовать обозначений на кириллице. В этом мы следовали сложившейся российской математической традиции написания формул с помощью латинского, греческого и других иноязычных алфавитов.

Компьютерная верстка книги выполнена В.В.Расиным с использованием пакета LaTeX2$_\varepsilon$.

Уральский госуниверситет,
г. Екатеринбург
сентябрь, 2008

М.О.Асанов
В.В. Расин

1. Введение в алгоритмы

Начиная с этой главы мы переходим к систематическому обсуждению методов решения оптимизационных задач дискретной математики.

Приведем несколько примеров дискретных оптимизационных задач.

Пусть имеется несколько городов и известны попарные расстояния между городами. Два города считаются соседними, если есть дорога, соединяющая эти города и не проходящая через другой город. Требуется найти кратчайший путь между некоторой парой городов (*задача о кратчайшем пути*).

Еще один пример: есть n станков и m деталей, каждую деталь можно обрабатывать на любом станке, но время обработка детали на одном станке может отличаться от времени ее обработки на другом станке. Предположим, что для каждой пары станок — деталь эти времена заданы. Требуется так организовать производство деталей, т. е. разместить детали по станкам таким образом, чтобы суммарное время работы было наименьшим (*задача оптимального назначения*).

Наконец, одной из самых популярных дискретных оптимизационных задач является следующая задача.

Путешественник хочет объехать n городов, побывав в каждом ровно по одному разу, и вернуться в исходный город, затратив при этом минимальноую сумму на поездку. Затраты на поездку складываются из затрат на переезды между парами городов, причем эти затраты заранее известны (*задача коммивояжера*).

Все приведенные выше задачи имеют ряд общих свойств, характерных для дискретных оптимизационных задач.

Во-первых, в каждой задаче имеется лишь конечное число вариантов (путей между городами, способов распределения деталей по станкам, маршрута передвижения путешественника), из которых требуется осуществить выбор. Во-вторых, каждому варианту сопоставлена некоторая числовая характеристика (длина пути, суммарное время работы, стоимость поездки). В-третьих, требуется выбрать вариант, числовая характеристика которого достигает экстремума.

Наиболее очевидный способ решения подобных задач — это полный перебор всех вариантов. Однако этот способ наименее удачен, поскольку все интересные с практической точки зрения ситуации возникают именно тогда, когда число возможных вариантов чрезвычайно велико. Полный перебор всех вариантов потребовал бы столь большого времени, что стал бы практически нереализуем даже на самых быстродействую-

щих ЭВМ.

К счастью, для многих важных задач дискретной оптимизации существуют методы решения, намного более экономичные, чем полный перебор. Именно такие задачи и алгоритмы их решения будут рассмотрены в этой и последующих главах.

1.1. Алгоритмы и их сложность

По устоявшейся терминологии различают *массовые* и *индивидуальные* задачи. Все вышеприведенные примеры дают образцы именно массовых задач. Индивидуальная задача получается из массовой при помощи фиксации набора условий. Например, из массовой задачи коммивояжера получится индивидуальная задача, если зафиксировать число городов и определить затраты на переезды между всеми парами городов.

Каждая массовая задача (в дальнейшем просто задача) характеризуется *размером*. Размер задачи служит мерой количества входных данных и представляется одним или несколькими целочисленными параметрами. Например, размерностью задачи коммивояжера естественно считать число n городов, которые собирается посетить путешественник.

Под алгоритмом понимается общий, шаг за шагом выполнимый способ получения решения данной задачи. Заметим, что существует несколько формализаций понятия алгоритма (машины Тьюринга, рекурсивные функции, нормальные алгоритмы Маркова), однако их рассмотрение далеко выходит за рамки данной книги. Для определенности можно считать, что алгоритм является программой на некотором языке высокого уровня.

Для оценки качества алгоритмов можно применять различные критерии. Одной из важнейших характеристик алгоритма является *временная сложность в худшем случае*. Пусть размер массовой задачи определяется одним целочисленным параметром n. Рассмотрим все индивидуальные задачи размера n и обозначим через $t(n)$ максимальное число действий, которое необходимо для решения любой такой индивидуальной задачи. Функция $t(n)$ — это и есть временная сложность алгоритма в худшем случае (или просто сложность алгоритма).

Под действием, производимым алгоритмом, будем понимать выполнение простой «операции», обычно аппаратно реализованной на любой ЭВМ, а именно, любой арифметической операции, операции сравнения, пересылки и т. п. Ясно, что при таком определении действия сложность

алгоритма зависит от конкретного вида машинных команд. Поэтому нас всегда будет интересовать лишь асимптотическая сложность алгоритма, т. е. порядок роста сложности при условии, что размер задачи неограниченно возрастает.

При сравнении скорости роста двух неотрицательных функций $f(n)$ и $g(n)$ удобно использовать следующие обозначения. Будем говорить, что $f(n) = O(g(n))$, если существуют такие положительные константы c и N, что $f(n) \leqslant c \cdot g(n)$ для всех $n > N$. В этой же ситуации можно использовать и обозначение $g(n) = \Omega(f(n))$.

Например, справедливы соотношения $\log_2 n = O(n)$, $n = \Omega(\log_2 n)$, $n = O(2^n)$.

На практике алгоритм решения некоторой задачи считается достаточно хорошим, если сложность этого алгоритма есть $O(n^k)$ при некотором $k > 0$. В таком случае говорят, что задача может быть решена за *полиномиальное время*, или, короче, что задача *полиномиально разрешима*, а сам алгоритм называют *полиномиальным*. Если сложность алгоритма равна $O(n)$, то такой алгоритм называется *линейным*. Напротив, если алгоритм имеет сложность $\Omega(a^n)$ при некотором $a > 1$, то его называют *экспоненциальным*.

Отметим, что как задача о кратчайшем пути, так и задача оптимального назначения являются полиномиально разрешимыми. В то же время для задачи коммивояжера неизвестен полиномиальный алгоритм; впрочем, нет и доказательства того, что такой алгоритм не существует (см. [18]).

Важность понятия сложности алгоритма хорошо иллюстрируют следующие таблицы, заимствованные из книги [6].

Первая таблица построена в предположении, что один шаг работы алгоритма требует для своего выполнения 1 миллисекунду. Как следует из таблицы 1, при увеличении времени с 1 секунды до 1 часа, т. е. в $3,6 \cdot 10^3$ раз, размер задачи, решаемой алгоритмом сложности 2^n увеличивается только в $21/9$ раза, а для алгоритма сложности n — ровно в $3,6 \cdot 10^3$ раз.

Может показаться, что рост скорости вычислений, вызванный появлением новых ЭВМ, уменьшит значение эффективных, т. е. имеющих «небольшую» сложность алгоритмов. Однако, как это видно из таблицы 0, в действительности происходит в точности противоположное. С ростом быстродействия ЭВМ становится возможным решение задач все большего размера, и именно от сложности алгоритма зависит насколько увеличение скорости ЭВМ влияет на увеличение размера задачи

Сложность алгоритма	Максимальный размер задачи		
	за 1 сек	за 1 мин	за 1 час
n	1000	$6 \cdot 10^4$	$3,6 \cdot 10^6$
$n \log_2 n$	140	4893	$2 \cdot 10^5$
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

Таблица 1

.

Рассмотрим теперь таблицу 2. Пусть для решения некоторой задачи имеется алгоритм сложности n^3. Тогда десятикратное увеличение скорости быстродействия ЭВМ позволит увеличить размер задачи, решаемой за 1 минуту в 2,15 раза. Переход к алгоритму со сложностью n^2, позволит решить в 6 раз большего размера (см. таблицу 1). Если в качестве единицы времени взять один час, то сравнение будет еще более впечатляющим.

Сложность алгоритма	Максимальный размер задачи, разрешимой за единицу времени		
	На современных ЭВМ	На ЭВМ с 10-кратной скоростью	На ЭВМ с 1000-кратной скоростью
n	K	10K	1000K
$n \log_2 n$	L	Почти $10L$ при больших L	Почти $1000L$ при больших L
n^2	M	$3{,}16M$	$31{,}6M$
n^3	N	$2{,}15N$	$10N$
2^n	P	P+3,3	P+9,97

Таблица 2

Отметим, что для алгоритма сложности 2^n десятикратное увеличение скорости ЭВМ добавляет к размеру разрешимой задачи только три единицы, тогда как для алгоритма сложности n^2 происходит увеличение в три раза, а для алгоритма сложности n — в десять раз.

1.2. Запись алгоритмов

Предполагается, что читатель знаком с каким-нибудь языком программирования высокого уровня. Для записи алгоритмов будет исполь-

зоваться упрощенный Паскаль, являющийся неформальной версией языка Паскаль.

В упрощенном Паскале такие понятия как константа, переменная, оператор, функция и процедура имеют обычное значение. Заметим, что при описании алгоритмов тип данных явно объявляться не будет, поскольку он всегда будет понятен из контекста. Упрощенный Паскаль позволяет применять традиционные конструкции математики; в частности, разрешается использовать любые математические предписания, если они понятны и перевод их в операторы языка Паскаль (или другого языка высокого уровня) не вызывает затруднений.

Операторы упрощенного Паскаля записываются по правилам, принятым в языке Паскаль. В частности, составной оператор заключается в операторные скобки **begin** и **end**. Договоримся, что операторы, записанные в одной строке, образуют составной оператор; в такой ситуации операторные скобки будут иногда опускаться.

Для обеспечения большей наглядности при описании алгоритмов, в упрощенный Паскаль вводятся некоторые дополнительные операторы:

оператор цикла **for** $x \in X$ **do** P (для каждого элемента x из множества X выполнить оператор P);

условный оператор **if exists** x, $B(x)$ **then** P **else** Q (если существует элемент x, удовлетворяющий условию $B(x)$, то выполнить оператор P, иначе выполнить оператор Q);

оператор $x \leftrightarrow y$ (обмен значениями между x и y).

При описании алгоритмов приходится использовать такие структуры данных как списки, стеки и очереди. Список $L = (l_1, \ldots l_n)$ является конечной последовательностью однотипных элементов. В отличие от массива число элементов в списке заранее не фиксируется. Заметим также, что список может не содержать ни одного элемента, т. е. он может быть пустым. В этом случае будет использоваться обозначение $L = nil$.

Список $S = (s_1, \ldots, s_n)$ называется *стеком*, если в нем выделен элемент, называемый *вершиной* (будем считать, что в непустом стеке вершиной является последний элемент, а в пустом стеке вершина не определена) и заданы следующие две процедуры и функция:

$S \Leftarrow x$ — втолкнуть x в стек, т. е. получить список вида $(s_1, \ldots, s_n, x)$;

$x \Leftarrow S$ — вытолкнуть вершину непустого стека в переменную x. В результате выполнения этой процедуры список S принимает значение $(s_1, \ldots, s_{n-1})$, а $x = s_n$;

$top(S)$ — значением этой функции является вершина стека.

Список $Q = (q_1, \ldots, q_n)$ называется *очередью*, если в нем выделены

начало (элемент q_1) и конец (элемент q_n) и определены две процедуры:

$Q \Leftarrow x$ — втолкнуть x в конец очереди, т. е. получить список $(q_1, \ldots, q_n, x)$;

$x \Leftarrow Q$ — исключить из непустой очереди начало и передать его в переменную x. После выполнения этой процедуры $Q = (s_2, \ldots, s_n)$ и $x = s_1$.

Если v — вершина графа G, то через $list[v]$ обозначается список вершин, смежных с v. Аналогично, для вершины v ориентированного графа G, через $\overleftarrow{list}[v]$ (соответственно $\overrightarrow{list}[v]$) будет обозначаться список концов исходящих из v дуг (соответственно список начал входящих в v дуг.

При записи алгоритма строки обычно будут нумероваться с тем, чтобы можно было указать на определенный оператор и группу операторов.

Договоримся об обозначениях для часто используемых функций. Поскольку в дальнейшем будут использоваться только логарифмы по основанию 2, вместо $\log_2 x$ будем писать просто $\log x$. Для произвольного действительного числа x через $\lfloor x \rfloor$ (*целая часть* или *дно* x) будем обозначать наибольшее целое число, не превосходящее x, а через $\lceil x \rceil$ (*потолок* x) — наименьшее целое число, большее или равное x.

1.3. Корневые и бинарные деревья

Дерево $T = (V, E)$, в котором зафиксирована некоторая вершина r, называется *корневым деревом с корнем* r. Такое корневое дерево будет обозначаться через (T, r).

На множестве вершин V корневого дерева можно определить следующее отношение: $u \leqslant v$ тогда и только тогда, когда v лежит на простой (r, u)-цепи.

Очевидно, отношение $\leqslant$ на множестве вершин корневого дерева является отношением частичного порядка. Таким образом, корневое дерево с корнем r является одновременно частично упорядоченным множеством, в котором r является наибольшим элементом. Это частично упорядоченное множество удобно обозначить через $(T, \leqslant)$.

Корневые деревья принято изображать в виде диаграмм соответствующих частично упорядоченных множеств (см. рис. 1).

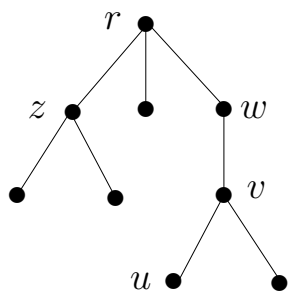


Рис. 1

Минимальные элементы частично упорядоченного множества $(T, \leqslant)$ часто называются *листьями* корневого дерева (T, r). Если $u < v$, то говорят, что u — *потомок* v, а v — *предок* u. Очевидно, что если v, w — предки вершины u, то они сравнимы в частично упорядоченном множестве $(T, \leqslant)$. Поэтому среди предков вершины u существует наименьший предок x; говорят, что x — *отец* вершины u, а u — *сын* вершины x. Для произвольной вершины v через $\hat{v}$ обозначим поддерево с корнем v; это корневое дерево состоит из самой вершины v и всех ее потомков. На рис. 1 вершина u является листом, вершина w — предок вершин u и v, причем для вершины v она является отцом, вершины v и z несравнимы.

Длина простой (r, v)-цепи называется *уровнем* вершины v. *Высота* $h(T)$ корневого дерева (T, r) — это наибольший из уровней его вершин.

Установим связь между высотой корневого дерева и числом его листьев.

Лемма 1. *Пусть (T, r) — корневое дерево высоты h, в котором каждая вершина имеет не более чем k сыновей. Тогда число листьев этого дерева не превосходит k^h.*

Доказательство. Утверждение, очевидно, выполнено при $h = 1$. Пусть $h > 1$. Для каждого сына s_i $(1 \leqslant i \leqslant p \leqslant k)$ корня r рассмотрим корневое дерево $T_i = \hat{s}_i$. Ясно, что $h(T_i) < h$. Поэтому к каждому дереву T_i, $1 \leqslant i \leqslant p$, применимо предположение индукции. Следовательно, число листьев в дереве T не превосходит $p \cdot k^{h-1} \leqslant k \cdot k^{h-1} \leqslant k^h$. $\square$

В приложениях часто используются так называемые *бинарные (двоичные)* деревья. Бинарное дерево — это корневое дерево, обладающее следующими дополнительными свойствами:

1) каждая вершина имеет не более двух сыновей;

2) для любой вершины v каждый сын имеет дополнительный признак — он является либо *левым* $(left(v))$, либо *правым* $(right(v))$.

Удобно считать, что существует пустое бинарное дерево, т. е. бинарное дерево с пустым множеством вершин. С учетом этого соглашения

для каждой вершины v определено левое поддерево $Left(v)$ и правое поддерево $Right(v)$. Если поддерево $Left(v)$ (соответственно $Right(v)$) непусто, то его корнем является левый (соответственно правый) сын вершины v. Запись $r = nil$ служит для обозначения пустого бинарного дерева. Поэтому отсутствие у вершины v данного бинарного дерева левого (соответственно правого) сына можно выразить при помощи равенства $left(v) = nil$ (соответственно $right(v) = nil$). При работе с бинарными деревьями будет использоваться процедура $Create(r)$, добавляющая вершину в пустое бинарное дерево (реализация этой процедуры в конкретном языке программирования зависит, разумеется, от способа представления бинарного дерева и возможностей языка).

Поскольку каждое бинарное дерево либо является пустым, либо состоит из корня и двух его поддеревьев — левого и правого, класс бинарных деревьев допускает рекурсивное описание. Рекурсивная природа бинарных деревьев позволяет многие алгоритмы обработки бинарных деревьев описывать при помощи рекурсии.

Остановимся на алгоритмах обхода бинарных деревьев. Рассмотрим следующие три способа обхода:

1) обход сверху вниз (обход в прямом порядке);

2) обход слева направо (обход во внутреннем порядке);

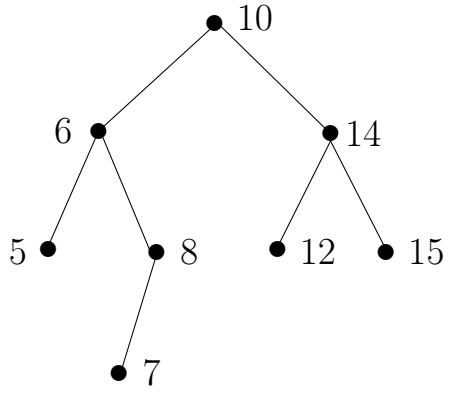
3) обход снизу вверх (обход в обратом порядке).

Эти три способа обхода (в указанном выше порядке) реализуются тремя процедурами: $PreOrder(v)$, $InOrder(v)$ и $PostOrder(v)$. В этих процедурах в каждой вершине v бинарного дерева выполняется оператор $P(v)$.

Procedure $PreOrder(v)$;
begin
  if $v \neq nil$ **then**
    $P(v)$; $PreOrder(left(v))$;$PreOrder(right(v))$;
end;
Procedure $InOrder(v)$;
begin
  if $v \neq nil$ **then**
    $InOrder(left(v))$; $P(v)$; $InOrder(right(v))$;
end;
Procedure $PostOrder(v)$;
begin
  if $v \neq nil$ **then**
    $PostOrder(left(v))$; $PostOrder(right(v))$; $P(v)$;

end;

На рис. 2 каждая из этих трех процедур применяется к бинарному дереву, вершины которого помечены числами. Справа от дерева показан тот порядок, в котором соответствующая процедура обходит бинарное дерево (1 — обход в прямом порядке, 2 — обход во внутреннем порядке, 3 — обход в обратном порядке).



1) 10,6,5,8,7,14,12,15;

2) 5,6,7,8,10,12,14,15;

3) 5,7,8,6,12,15,14,10.

Рис. 2

Пусть (T, r) — бинарное дерево, вершины которого помечены элементами некоторого линейно упорядоченного множества L. Иными словами, определено отображение λ из множества вершин V бинарного дерева в множество L. Такое бинарное дерево называется *деревом поиска*, если для любой вершины $v \in V$ выполнены условия

1) $\lambda(u) \leqslant \lambda(v)$ для произвольной вершины u из $Left(v)$;

2) $\lambda(u) \geqslant \lambda(v)$ для произвольной вершины u из $Right(v)$.

Лемма 2. *Обход дерева поиска во внутреннем порядке сортирует метки вершин в порядке возрастания.*

Читатель без труда проверит это утверждение применив индукцию по высоте дерева поиска.

Свойство дерева поиска, отмеченное в лемме 2, лежит в основе следующего способа сортировки последовательностей. Для данной последовательности $(x_1, \ldots, x_n)$ элементов из множества L сначала строим дерево поиска так, что метками вершин являются элементы последовательности, а затем обходим это дерево во внутреннем порядке. Дерево поиска изображено на рис. 2. Здесь же показано, что в результате обхода этого дерева во внутреннем порядке метки вершин сортируются в порядке возрастания.

Для построения дерева поиска из данной последовательности применяется следующая рекурсивная процедура.

Procedure $Add(r, x)$;
begin
  if r=**nil then**
    $Create(r)$; $\lambda(r) := x$
  else if $x < \lambda(r)$ **then** $Add(left(r), x)$
    else $Add(right(r), x)$
end;

Эта процедура к уже существующему дереву поиска с корнем r добавляет очередной элемент x так, что свойство быть деревом поиска сохраняется. Заметим, что процедура $Add(r, x)$ написана в предположении, что все элементы исходной последовательности попарно различны. Нетрудно понять, что такой способ сортировки последовательностей имеет сложность $O(n^2)$.

Обширный материал, относящийся к затронутым в этом разделе вопросам, читатель найдет в книгах [13], [29], [46].

1.4. Сортировка массивов

Пусть $a = [a_1, \ldots, a_n]$ — массив, составленный из элементов некоторого линейно упорядоченного множества L. Сортировка массива a состоит в нахождении такой перестановки σ множества $\{1, \ldots, n\}$, что $a_{\sigma(1)} \leqslant a_{\sigma(2)} \leqslant \ldots \leqslant a_{\sigma(n)}$. Предполагается, что всю инфориацию о данном массиве мы можем получить лишь применяя операцию сравнения к элементам массива.

Оценим снизу сложность произвольного алгоритма сортировки массива из n элементов. Для этого построим так называемое *дерево решений*. На рис. 3 изображено дерево решений для массива $[a_1, a_2, a_3]$, состоящего из трех элементов. Элементы дерева изображаются овалами и прямоугольниками. Овалы заключают проверяемые условия, в прямоугольниках записываются отсортированные массивы.

Очевидно, что дерево решений массива из n элементов является бинарным деревом, причем количество листьев в этом дереве равно $n!$.

Пусть h — высота дерева решений, построенного для массива из n элементов. Из леммы 1 разд. 1.3 следует, что $n! \leqslant 2^h$, откуда $h \geqslant \log(n!)$. Поскольку $n! > n^{n/2}$ при $n \geqslant 3$, имеем

$$\log(n!) > \frac{n \log(n)}{2}.$$

Сдедовательно, справедлива

Лемма 1. *Высота дерева решений массива из n элементов больше чем $(n \log n)/2$.*

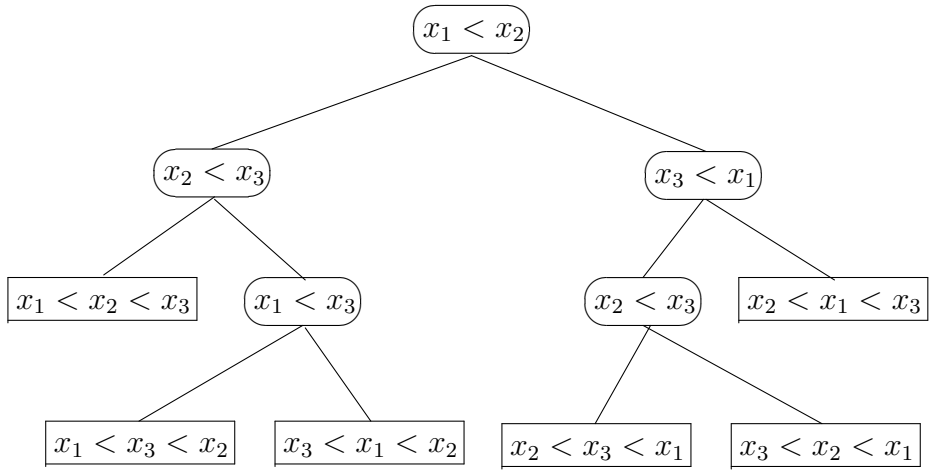


Рис. 3

Предположим теперь, что к массиву $a = [a_1, \ldots, a_n]$ применен некоторый алгоритм сортировки. В дереве решений для массива из n элементов рассмотрим простую цепь P, соединяющую корень дерева решений с листом, представляющим отсортированный массив a. Интуитивно понятно, что количество сравнений, использованых алгоритмом для сортировки массива a не меньше длины цепи P. Отсюда можно сделать вывод:

Произвольный алгоритм сортировки массива из n элементов (при помощи сравнений) имеет сложность $\Omega(n \log n)$.

Перейдем к построению простого и эффективного алгоритма сортировки массивов, сложность которого равна $O(n \log n)$. Этот алгоритм был предложен в 1964 году независимо Дж. Уильямсом (под названием $HeapSort$) и Флойдом (под названием $TreeSort$). В некоторых книгах (см., например, [13], [46]) его называют *алгоритмом пирамидальной сортировки*. Такое название более удобно, и мы будем использовать именно его.

Пусть $a = [a_1, \ldots, a_n]$ — произвольный массив. Свяжем с массивом a бинарное дерево T_n, считая, что в корне находится элемент a_1 и для каждого элемента a_i положим $a_{2i} = left(a_i)$ (если $2i \leqslant n$) и $a_{2i+1} = = right(a_i)$ (если $2i+1 \leqslant n$). Из этого определения, в частности, вытекает, что a_i лист тогда и только тогда, когда $2i > n$. На рис. 4 изображено бинарное дерево T_9.

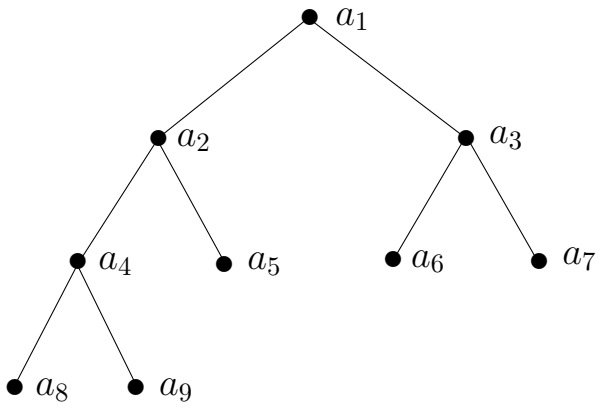


Рис. 4

Лемма 2. *Высота бинарного дерева T_n равна $\lfloor \log n \rfloor$.*

Доказательство. Пусть h — высота дерева T_n. Из определения T_n следует, что при каждом $k \leqslant h$ элемент a_{2^k} имеет уровень k. Отсюда вытекает, что элемент a_{2^h} является листом в T_n. Поэтому $2^{h+1} > n$. Следовательно, $2^h \leqslant n < 2^{h+1}$, откуда $h \leqslant \log n < h+1$, т. е. $h = \lfloor \log n \rfloor$. □

Бинарное дерево T_n будем называть *пирамидой* или *сортирующим деревом*, если при $1 \leqslant i \leqslant n/2$ элемент a_i больше или равен каждого из своих сыновей.

Займемся сначала той частью алгоритма пирамидальной сортировки, которая любое бинарное дерево T_n преобразует в пирамиду.

В основе алгоритма пирамидальной сортировки лежит следующая рекурсивная процедура $Sift(i, j)$. В этой процедуре использована функция $MaxS(i)$, вычисляющая номер наибольшего из сыновей элемента a_i.

```
1.  procedure Sift(i, j);
2.  begin
3.    if i ⩽ ⌊j/2⌋ then
4.      begin
5.        k := MaxS(i);
6.        if a[i] < a[k] then
7.          begin
8.            a[i] ↔ a[k]; Sift(k, j);
9.          end
10.     end
11. end;
```

Скажем, что дерево T_n является *частичной пирамидой с индексом* l, если при $i \geqslant l$ все поддеревья с корнями a_i являются пирамидами. Ясно, что любое дерево T_n является частичной пирамидой с индексом l, если l удовлетворяет неравенству $l > \lfloor n/2 \rfloor$.

Лемма 3. *Если дерево T_n является частичной пирамидой индекса $l+1$, то в результате выполнения процедуры $Sift(l, n)$ оно становится частичной пирамидой индекса l.*

Читатель без труда убедится в справедливости леммы 3, применив возвратную индукцию по индексу l.

Теперь нетудно понять, что последовательное применение процедуры $Sift(i, n)$, начиная с $i = \lfloor n/2 \rfloor$ и заканчивая $i = 1$, преобразует любое дерево T_n в пирамиду.

Заметим, что в пирамиде элемент, находящийся в корне, является наибольшим. Воспользовавшись этим свойством пирамиды, процесс сортировки массива можно представить следующим образом. Поменяв местами элементы a_1 и a_n, мы получим, что наибольший элемент массива поставлен на последнее место. Рассмотрим бинарное дерево T_{n-1}, соответствующее массиву $a_1, \ldots, a_{n-1}$ (тем самым последний элемент исключен из рассмотрения). Очевидно, дерево T_{n-1} является частичной пирамидой с индексом 2. В силу леммы 3 однократное применение процедуры $Sift(1, n-1)$ преобразует T_{n-1} в пирамиду. Далее нужно сделать перестановку элементов a_1 и a_{n-1} и применить процедуру $Sift(1, n-2)$ к дереву T_{n-2}. Продолжая этот процесс до тех пор, пока не останется одноэлементное дерево T_1, мы придем к отсортированному массиву.

Запишем формальную версию алгоритма пирамидальной сортировки.

Алгоритм 1.1.

```
1.  begin
2.    for i := ⌊n/2⌋ downto 1 do
3.        Sift(i, n);
4.    for i := n downto 2 do
5.        begin a[1] ↔ a[i]; Sift(1, i − 1); end
6.  end.
```

Ясно, что цикл в строках 2, 3 преобразует дерево T_n в пирамиду, а цикл в строках 4, 5 сортирует массив.

В качестве примера применим этот алгоритм к массиву $a =$

$= [2, 3, 6, 1, 4, 8]$. Сначала рассмотрим работу первого цикла (строки 2, 3). Все перестановки элементов массива будем отображать в дереве T_6.

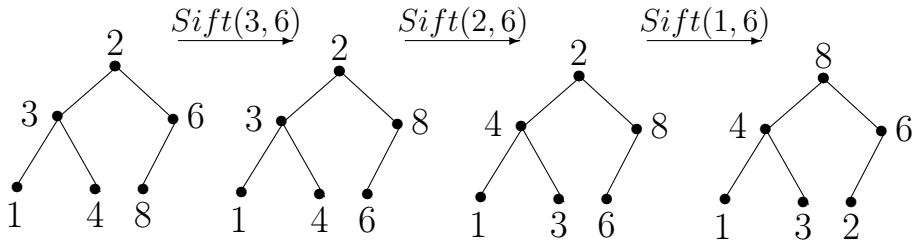


Рис. 5

Последнее из полученных деревьев представляет массив $[8, 4, 6, 1, 3, 2]$, и, очевидно, является пирамидой. Проследим теперь работу второго цикла (строки 4, 5).

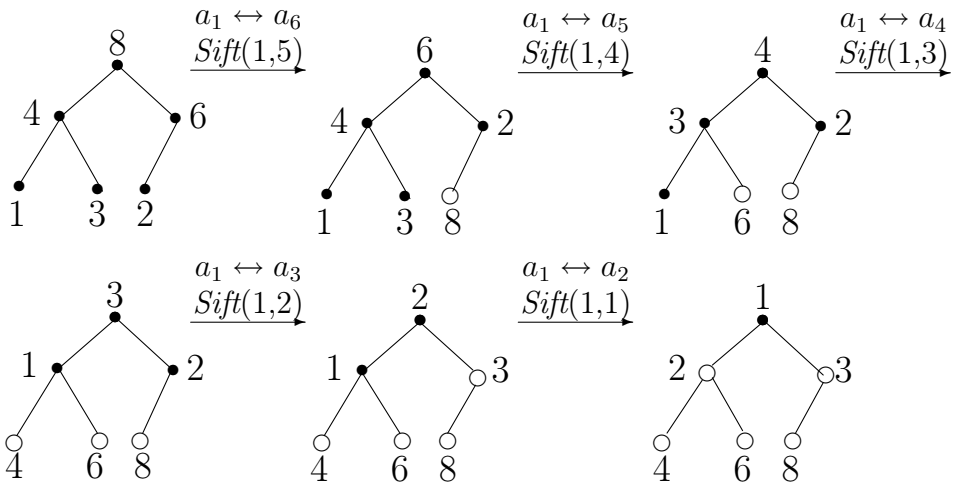


Рис. 6

На рис. 6 показаны перестановки, которые проделывает алгоритм при каждом выполнении второго цикла. Для удобства читателя вершины, метки которых уже расположены на своих местах, показаны незакрашенными.

Оценим сложность алгоритма пирамидальной сортировки.

Теорема 1.1. *Алгоритм пирамидальной сортировки имеет сложность* $O(n \log n)$.

Доказательство. Пусть $a = [a_1, \ldots, a_n]$ — данный массив, h — высота соответствующего бинарного дерева T_n. Обозначим через $t(h)$ функцию, значением которой является число операций, необходимых для сортировки массива, которому отвечает дерево T_n. Ясно, что $t(h) = = t_1(h) + t_2(h)$, где $t_1(h)$ и $t_2(h)$ — количества операций, необходимых для выполнения первого и второго циклов из алгоритма 1.1.

Оценим сначала функцию $t_1(h)$. Напомним, что процедура $Sift(i,n)$ находит номер m наибольшего из сыновей вершины a_i, и в том случае, когда $a_i < a_m$, переставляет элементы a_i и a_m, а затем вызывает процедуру $Sift(m,n)$. Ясно, что количество действий (сравнений и перестановок) c, выполненых между вызовами $Sift(i,n)$ и $Sift(m,n)$, не зависит от номера i. Убедимся, что $t_1(h)$ удовлетворяет неравенствам

$$t_1(1) \leqslant c, \qquad t_1(h) \leqslant 2t_1(h-1) + ch, \text{ если } h \geqslant 2.$$

В самом деле, преобразование в пирамиды двух поддеревьев с корнями a_2 и a_3 требует не более $2t_1(h-1)$ действий, а процедура $Sift(1,n)$ выполняет не более ch действий.

Пусть $f(h) = c(2^{h+1} - h - 2)$. Покажем, что $t_1(h) \leqslant f(h)$ при любом $h \geqslant 1$. Если $h = 1$, то

$$t_1(h) \leqslant c = f(1).$$

Пусть $h > 1$. Тогда

$$t_1(h) \leqslant 2t_1(h-1) + ch \leqslant 2c(2^h - h - 1) + ch =$$

$$= c(2^{h+1} - h - 2) = f(h).$$

Таким образом,

$$t_1(h) \leqslant c(2^{h+1} - h - 2) < c2^{h+1} \leqslant 2cn.$$

Переходя к оценке $t_2(h)$ следует лишь отметить, что второй цикл в алгоритме 1.1 выполняется $n-1$ раз, а процедура $Sift(1, i-1)$ $(2 \leqslant i \leqslant n)$ требует не более ch действий. Отсюда

$$t_2(h) \leqslant cnh \leqslant cn \log n.$$

Следовательно, $t_2(h) = O(n \log n)$. Окончательно имеем

$$t(h) = t_1(h) + t_2(h) = O(n) + O(n \log n) = O(n \log n).$$

□

2. Поиск в графе

Многие алгоритмы на графах основаны на систематическом переборе всех вершин графа, при котором каждая вершина просматривается в точности один раз. В этой главе мы рассмотрим два стандартных и широко используемых метода такого перебора: поиск в глубину и поиск в ширину. Оба метода изучаются применительно к обыкновенным графам, поскольку на произвольные графы они могут быть распространены очевидным образом. В разд. 2.3 поиск в глубину применяется для отыскания компонент сильной связности в орграфе.

2.1. Поиск в глубину

Идея этого метода состоит в следующем. Поиск в обыкновенном графе G начинается с некоторой начальной вершины v (с этого момента v считается *просмотренной*). Пусть u – последняя просмотренная вершина (этой вершиной может быть и v). Тогда возможны два случая.

1) Среди вершин, смежных с u, существует еще непросмотренная вершина w. Тогда w объявляется просмотренной, и поиск продолжается из вершины w. Будем говорим, что вершина u является *отцом* вершины w ($u = father[w]$). Ребро uw в этом случае будет называться *древесным*.

2) Все вершины, смежные с u, просмотрены. Тогда u объявляется *использованной* вершиной. Обозначим через x ту вершину, из которой мы попали в u, т. е. $x = father[u]$; поиск в глубину продолжается из вершины x.

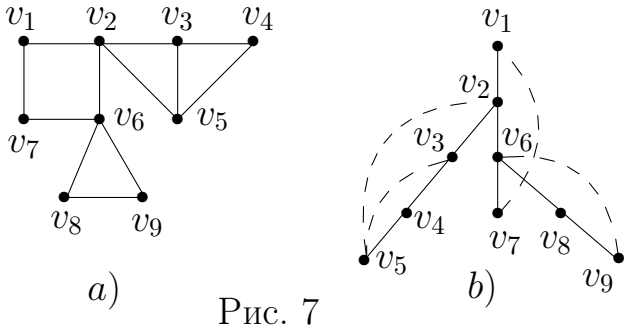
Что произойдет, когда все просмотренные вершины будут использованы? Если в графе G не осталось непросмотренных вершин, то поиск заканчивается. Если же осталась непросмотренная вершина y, то поиск продолжается из этой вершины.

Поиск в глубину просматривает вершины графа G в определенном порядке. Для того, чтобы зафиксировать этот порядок, будет использоваться массив num. При этом естественно считать, что $num[v] = 1$, если v начальная вершина, и $num[w] = num[u] + 1$, если w просматривается сразу после того, как просмотрена вершина u.

Пусть в обыкновенном графе G проведен поиск в глубину. Обозначим через T множество всех древесных ребер. Все остальные ребра графа будем называть *обратными* ребрами. Множество всех обратных ребер будем обозначать через B.

Результат применения поиска в глубину к связному графу G (рис. 7 *а*)

показан на рис. 7 *b*. Здесь сплошные линии изображают древесные ребра, а пунктирные линии — обратные ребра. Заметим, что нумерация вершин соответствует порядку, в которым они просматриваются поиском в глубину. Можно обратить внимание на то, что множество всех древесных ребер с выделенной начальной вершиной v_1 образует корневое дерево с корнем v_1. Это корневое дерево часто называют *глубинным* деревом, или, короче, *d*-деревом графа G. Следует также отметить, что каждое обратное ребро соединяет в *d*-дереве предка и потомка.



a) *b)*

Рис. 7

Пусть G — несвязный граф, $G_1, \ldots, G_k$ — множество всех его компонент связности. Обозначим через T_i множество древесных ребер, выделенных поиском в глубину в компоненте G_i, а через v_i — корневую вершину из G_i, т. е. первую просмотренную вершину подграфа G_i. Таким образом, множество всех древесных ребер несвязного графа образует остовный лес. Фиксируя в каждом поддереве этого леса корневую вершину, мы получим *глубинный лес* или, короче, *d*-лес графа G.

Представим теперь формальное описание указанного алгоритма. В алгоритме используются описанные ранее массивы num и $father$. В процессе работы алгоритма массив num используется для распознавания непросмотренных вершин, а именно, равенство $num[v] = 0$ означает, что вершина v еще не просмотрена.

Вначале изложим версию алгоритма поиска в глубину, основанную на рекурсивной процедуре $DFS(v)$ (название процедуры является аббревиатурой от англ. depth first search), осуществляющей поиск в глубину из вершины v.

Алгоритм 2.1.

1. **procedure** $DFS(v)$;
2. **begin**
3. $num[v] := i$; $i := i + 1$;
4. **for** $u \in list[v]$ **do**
5. **if** $num[u] = 0$ **then**

```
6.        begin
7.          T := T ∪ {uv}; father[u] := v; DFS(u)
8.        end
9.      else if num[u] < num[v] and u ≠ father[v] then
10.       B := B ∪ {uv}
11. end;
12. begin
13.   i := 1; T := ∅; B := ∅;
14.   for v ∈ V  do  num[v] := 0;
15.   for v ∈ V  do
16.     if num[v] = 0 then
17.       begin
18.         father[v] := ∅; DFS(v)
19.       end
20. end.
```

Алгоритм 2.1 применим к произвольному обыкновенному графу G. Если G — связный граф, то цикл в строках 15 — 19 достаточно заменить вызовом процедуры $DFS(v_0)$ применительно к начальной вершине v_0. Заметим, что в терминах алгоритма 2.1 вершина v просмотрена с началом работы процедуры $DFS(v)$; в тот момент, когда процедура $DFS(v)$ закончила работу, вершина v является использованной.

Теорема 2.1. *Пусть G — связный (n, m)-граф. Тогда*

1) *поиск в глубину просматривает каждую вершину в точности один раз;*

2) *поиск в глубину требует $O(n + m)$ операций;*

3) *подграф (V, T) графа G является деревом.*

Доказательство. 1) Проверка в строке 5 гарантирует, что каждая вершина просматривается не более одного раза. Убедимся, что поиск в глубину просматривает каждую вершину. Пусть X — множество просмотренных вершин в тот момент, когда алгоритм закончил работу, $Y = V \setminus X$. Если Y непусто, то в силу связности графа G существует такое ребро xy, что $x \in X$, $y \in Y$. Процедура $DFS(x)$ полностью проработала, поэтому смежная с x вершина y должна быть просмотрена. Получили противоречие.

2) Число повторений цикла в процедуре (начало цикла в строке 4) с учетом рекурсивных обращений равно сумме степеней всех вершин графа, т. е. оно равно $2m$; следовательно, число операций пропорционально

m. Число повторений цикла в строке 14, очевидно, пропорционально n. Отсюда вытекает, что поиск в глубину требует $O(n+m)$ операций.

3) Ясно, что условие $num[u]=0$ (строка 5) выполнится $n-1$ раз. Отсюда следует, что $|T|=n-1$. Кроме того, из 1) вытекает, что множество ребер T не содержит циклов. Таким образом (V,T) — ацикличен и число ребер в этом графе на единицу меньше числа вершин. Следовательно, граф (V,T) — дерево. □

Пусть G — связный граф, v_0 — некоторая его вершина. Предположим, что в графе G проведен поиск в глубину из вершины v_0. Дерево (V,T) с выделенной вершиной v_0 является корневым деревом. Как отмечалось выше, это корневое дерево часто называют d-деревом или глубинным деревом графа G. Для любой вершины $u \neq v_0$ вершина $father[u]$ является отцом u в d-дереве.

Рассмотрим нерекурсивную версию процедуры $DFS(v)$. Рекурсия устраняется при помощи стека S, элементами которого являются вершины графа. Вершина v является просмотренной, если $num[v] \neq 0$. Вершина v становится использованной с момента, когда $v = top(S)$ (v находится в вершине стека) и все вершины, смежные с v, уже просмотрены (в этом случае v удаляется из стека). Вычисления, связанные с множеством обратных ребер B здесь опущены; читатель, разобравшийся с рекурсивной версией процедуры $DFS(v)$, без труда восстановит их.

```
1.  procedure DFS(v);
2.  begin
3.    num[v] := i; i := i + 1;
4.    S := ∅; S ⇐ v;
5.    while S ≠ ∅ do
6.      begin
7.        v := top(S);
8.        if  exists u, u ∈ list[v] and num[u] = 0
9.        then
10.         begin
11.           num[u] := i; i := i + 1; T := T ∪ {uv};
12.           father[u] := v; S ⇐ u;
13.         end
14.       else v ⇐ S
15.     end
16. end;
```

Отметим следующее важное свойство поиска в глубину: *если xy — обратное ребро, то вершины x, y сравнимы в d-дереве, т. е. одна из этих вершин является предком другой.*

В самом деле, пусть xy — обратное ребро графа G, причем $num[x] < num[y]$. Предположим, что вершины x и y несравнимы в d-дереве. Из описания алгоритма 2.1 следует, что в промежуток времени между началом работы процедуры $DFS(x)$ и ее завершением, будут просмотрены только потомки этой вершины. Поскольку $y \in list[x]$ и в момент завершения процедуры $DFS(x)$ вершина y еще не просмотрена, получаем противоречие с описанием алгоритма 2.1.

2.2. Алгоритм отыскания блоков и точек сочленения

Пусть $G = (V, E)$ — обыкновенный связный граф. В разд. **??** было определено понятие блока графа G и получен ряд утверждений о блоках. В частности, в этом разделе определено отношение $\approx$ на множестве ребер E графа G:

$$e \approx f \Leftrightarrow e = f \text{ или } e, f \text{ лежат на некотором цикле.}$$

Это отношение является эквивалентностью, причем каждый его класс совпадает с множеством ребер некоторого блока.

Предположим, что в графе G из некоторой вершины v_0 проведен поиск в глубину. Поиск в глубину строит d-дерево (V, T) и массив num, состоящий из номеров, которые присваиваются вершинам. Получим сначала в терминах d-дерева признак того, что данная вершина является точкой сочленения.

Лемма 1. *Пусть t, v, w — вершины графа G, причем t — отец, а w — предок вершины v. Если в графе G существует ребро $e = vw$, то $e \approx f = vt$.*

Доказательство. Можно считать, что $w \neq t$. Отсюда следует, что $w > t$, т. е. существует простая $(w, t)-$цепь. Добавляя к этой цепи ребра $e = vw$ и $f = vt$, получим цикл. □

Аналогично проверяется

Лемма 2. *Пусть v, w — вершины графа G, причем w — потомок вершины v. Если в графе G существует ребро $e = vw$, то $e \approx f = vs$ для некоторого сына s вершины v.*

Пусть v — произвольная вершина. Обозначим через $\hat{v}$ поддерево d-дерева с корнем v. Для произвольного непустого множества $X \subseteq \hat{v}$ через $VB(X, v)$ будем обозначать множество всех таких вершин w, что $w > v$ и для некоторой вершины $x \in X$ существует обратное ребро xw. Если X одноэлементно, т. е. $X = \{x\}$, то вместо $VB(\{x\}, v)$ будем писать $VB(x, v)$.

Лемма 3. *Пусть t, v, s — вершины графа G, причем t — отец, а s — сын вершины v. Ребра $e = vt$ и $f = vs$ лежат на общем цикле тогда и только тогда, когда $VB(\hat{s}, v) \neq \emptyset$.*

Доказательство. Предположим, что ребра $e = vt$ и $f = vs$ лежат на общем цикле C, имеющем вид $u_1, \ldots, u_p, u_1$. Можно считать, что $u_1 = v$, $u_2 = s$, $u_p = t$. Найдется наименьшее число $q > 2$ такое, что $u_q \not\leqslant s$. Ясно, что $q \leqslant p$ и ребро $u_{q-1}u_q$ является обратным, следовательно, вершины u_{q-1} и u_q сравнимы в d-дереве. Поскольку $u_{q-1} < s$, имеем $u_q \geqslant v$. Учитывая, что C — цикл и $u_1 = v$, получаем неравенство $u_q \geqslant t$. Таким образом, $u_q \in VB(\hat{s}, v)$, поэтому $VB(\hat{s}, v) \neq \emptyset$.

Обратно, пусть $w \in VB(\hat{s}, v)$. Тогда $w \geqslant t$, и существует обратное ребро xw, причем $x \leqslant s$. Ясно, что x — потомок вершины w в d-дереве. Поэтому существует простая (w, x)-цепь P, причем P содержит ребра vt и vs. Добавляя к цепи P ребро xw, получим требуемый цикл. □

Лемма 4. *Вершина v является точкой сочленения тогда и только тогда, когда выполняется одно из двух следующих условий:*

1) v — корень d-дерева, имеющий не менее двух сыновей;

2) v не является корнем и для некоторого сына s вершины v множество $VB(\hat{s}, v)$ пусто.

Доказательство. Пусть v — точка сочленения графа G. Тогда v — общая вершина различных блоков. Следовательно, существуют вершины x, y, обе смежные с v и такие, что $vx \not\approx vy$.

Предположим, что v — корень d-дерева. Тогда x, y, очевидно, являются потомками v. В силу леммы 2 существуют сыновья s_1, s_2 вершины v такие, что $vs_1 \approx vx$, $vs_2 \approx vy$. Поскольку $vx \not\approx vy$, имеем $vs_1 \not\approx vs_2$. Отсюда, в частности, следует, что $s_1 \neq s_2$.

Пусть v не является корнем d-дерева, и t — отец вершины v. Нетрудно понять, что либо $vt \not\approx vx$, либо $vt \not\approx vy$. Без ограничения общности можно считать, что $vt \not\approx vx$. Применение леммы 1 показывает, что x потомок вершины v. В силу леммы 2 существует такой сын s вершины v,

что $vx \approx vs$. Ясно, что $vt \not\approx vs$, и из леммы 3 следует пустота множества $VB(\hat{s}, v)$.

Проверим обратное утверждение. Предположим сначала, что выполнено условие 1), т. е. вершина v — корень, имеющий двух сыновей s_1 и s_2. Убедимся, что ребра vs_1 и vs_2 принадлежат разным блокам. Допустим, рассуждая от противного, что ребра vs_1 и vs_2 лежат в одном блоке. Тогда существует цикл C, содержащий эти ребра. Если C имеет вид $u_1, \ldots, u_p, u_1$, то можно считать, что $u_1 = v$, $u_2 = s_1$, $u_p = s_2$. Поскольку s_2 не является потомком s_1, найдется наименьшее число $q \geqslant 2$ такое, что u_q не является потомком s_1. Ясно, что $q \leqslant p$ и ребро $u_{q-1}u_q$ является обратным. Следовательно, u_q — предок вершины s_1, а потому $u_q = v$. Вершина v содержится в C более одного раза, что противоречит определению цикла.

Пусть выполнено условие 2). Обозначим через t отца вершины v. Поскольку $VB(\hat{s}, v)$ пусто, из леммы 3 получаем, что ребра vs, vt лежат в разных блоках. Понятно, что v — общая вершина этих блоков, следовательно, v — точка сочленения. □

Пусть X — произвольное множество вершин графа G. Обозначим через $num(X)$ множество $\{num[v] \mid v \in X\}$.

На множестве V вершин графа G рассмотрим следующую функцию

$$L[v] = \min num(\{v\} \cup VB(\hat{v}, v)).$$

Функция L обладает двумя важными свойствами:

1) с ее помощью легко распознавать точки сочленения (см. теорему 2.2);

2) значения этой функции весьма просто и естественно вычисляются при помощи поиска в глубину (см. лемму 6).

Лемма 5. *Пусть вершина v не является корнем d-дерева, s — некоторый сын вершины v. Множество $VB(\hat{s}, v)$ пусто в том и только том случае, когда $L[s] \geqslant num[v]$.*

Доказательство. В наших рассуждениях важную роль будут играть множества $VB(\hat{s}, s)$ и $VB(\hat{s}, v)$. Нетрудно понять, что выполнены включения

$$VB(\hat{s}, v) \subseteq VB(\hat{s}, s), \qquad VB(\hat{s}, s) \setminus VB(\hat{s}, v) \subseteq \{v\}.$$

Предположим, что $VB(\hat{s}, v) = \emptyset$. Тогда $VB(\hat{s}, s) \subseteq \{v\}$. Отсюда следует, что

$$\{s\} \cup VB(\hat{s}, s) \subseteq \{s, v\}.$$

Вычисляя минимум номеров вершин, входящих в левую и правую части этого включения, получим

$$L[s] \geqslant \min(num[s], num[v]) = num[v].$$

Обратно, если множество $VB(\hat{s}, v)$ непусто, то существует $w \in \in VB(\hat{s}, v)$. Ясно, что $num[w] < num[v]$. Поскольку $VB(\hat{s}, v) \subseteq VB(\hat{s}, s)$, имеем

$$\{s\} \cup VB(\hat{s}, v) \subseteq \{s\} \cup VB(\hat{s}, s).$$

Отюда следует, что

$$num[v] > num[w] \geqslant \min num(\{s\} \cup VB(\hat{s}, v)) \geqslant L[s].$$

Следовательно, $L[s] < num[v]$. □

Из лемм 4 и 5 вытекает следующее утверждение.

Теорема 2.2. *Пусть вершина v не является корнем d-дерева. Вершина v — точка сочленения графа G тогда и только тогда, когда $L[s] \geqslant num[v]$ для некоторого сына s этой вершины.*

Лемма 6. *Пусть v — произвольная вершина графа G. Тогда*

$$L[v] = \min\left(num[v], L[s_1], \ldots, L[s_p], num(VB(v, v))\right),$$

где $s_1, \ldots, s_p$ — все сыновья вершины v.

Доказательство. Поскольку $\hat{v} = \hat{s}_1 \cup \ldots \cup \hat{s}_p \cup \{v\}$, имеем

$$VB(\hat{v}, v) = VB(\hat{s}_1, v) \cup \ldots \cup VB(\hat{s}_p, v) \cup VB(v, v).$$

Кроме того, $\{v\} \cup VB(\hat{s}_i, v) = \{v\} \cup VB(\hat{s}_i, s_i)$ при $1 \leqslant i \leqslant p$. Следовательно,

$$\{v\} \cup VB(\hat{v}, v) = \{v\} \cup VB(\hat{s}_1, s_1) \cup \ldots \cup VB(\hat{s}_p, s_p) \cup VB(v, v).$$

Обозначим через W множество, стоящее в правой части этого равенства. Так как $v \in W$ и $num[v] < num[s_i]$ $(1 \leqslant i \leqslant p)$, выполнено равенство

$$\min num(W) = \min num(W \cup \{s_1, \ldots, s_p\}).$$

Теперь нетрудно видеть, что

$$L[v] = \min\left(num[v], L[s_1], \ldots, L[s_p], num(VB(v, v))\right). \quad \square$$

Дадим формальное описание алгоритма нахождения блоков и точек сочленения.

Алгоритм 2.2.

```
1.  procedure BiComp(v);
2.  begin
3.    num[v] := i; L[v] := i; i := i + 1;
4.    for u ∈ list[v] do
5.      if num[u] = 0 then
6.        begin
7.          SE ⇐ vu; father[u] := v; BiComp(u)
8.          L[v] := min(L[v], L[u]);
9.          if L[u] ⩾ num[v] then
10.           {получить новый блок; для этого из
               стека SE надо вытолкнуть все ребра
               вплоть до ребра vu}
11.       end
12.     else if num[u] < num[v] and u ≠ father[v] then
13.       begin
14.         SE ⇐ vu; L[v] := min(L[v], num[u]);
15.       end
16. end;
17. begin
18.   i := 1; SE := nil; father[v0] := ∅;
19.   for v ∈ V do num[v] := 0;
20.   BiComp(v0)
21. end.
```

Процедура $BiComp(v)$ является является модификацией процедуры $DFS(v)$ из алгоритма 2.1. В момент завершения работы процедуры $BiComp(v)$ значение $L[v]$ уже вычислено. Вычисление $L[v]$ производится по формуле из леммы 6. В самом деле, в строке 3 выполняется присваивание $L[v] := num[v]$. Далее в строке 8 учитываются значения функции L для каждого из сыновей вершины v. Наконец, в строке 14 находится минимальное из двух чисел: текущего значения $L[v]$ и $num[u]$, где u — очередной элемент из множества $VB(v, v)$.

Необходимое и достаточное условие для вершины быть точкой сочленения, сформулированное в теореме 2.2, проверяется в строке 9. Разумеется, это условие нельзя применять к корневой вершине v_0. Чтобы

узнать, является ли v_0 точкой сочленения, нужно подсчитать количество сыновей этой вершины в d-дереве.

Для нахождения множества ребер очередного блока алгоритм 2.2 использует стек SE, элементами которого являются ребра графа G.

Теорема 2.3. *Алгоритм 2.2 правильно находит блоки графа G.*

Доказательство. Пусть граф G содержит q блоков. Проведем индукцию по числу q. Если $q = 1$, то граф G неразделим. Из леммы 4 следует, что в d-дереве корневая вершина v_0 имеет единственного сына w. Нетрудно понять, что в момент завершения процедуры $BiComp(w)$ стек SE будет содержать все ребра графа G. Поскольку $L[v_0] = num[v_0] = 1$ и граф не имеет точек сочленения, условие в строке 9 выполнится только при $u = w$. В результате все ребра графа G будут вытолкнуты из стека SE, и мы получим единственный блок.

Пусть $q > 1$. Обозначим через u_1, v_1 ту пару вершин, для которой неравенство в строке 9 выполнится в первый раз. Ясно, что v_1 — точка сочленения графа G. К этому моменту процедура $BiComp(u_1)$ завершилась и точек сочленения не обнаружила. Поэтому после проверки условия в $L[u_1] \geqslant num[v_1]$ из стека SE будут удалены ребра некоторого блока B. Теперь можно считать, что алгоритм обрабатывает граф G_1, составленный из всех блоков графа G, отличных от B. К графу G_1 применимо предположение индукции, поэтому блоки, отличные от B, также будут найдены правильно. □

2.3. Алгоритм отыскания компонент сильной связности в орграфе

Пусть $G = (V, E)$ — связный орграф. Алгоритм 2.1 из разд. 2 легко приспособить для организации поиска в глубину в орграфе G. Для этого нужно лишь список $list(v)$ заменить списком $\overleftarrow{list}(v)$, состоящим из концов всех дуг, выходящих из вершины v. Все понятия, связанные с поиском в глубину в обыкновенном графе, очевидным образом применимы и для орграфов. В частности, в результате поиска в глубину в орграфе G строится d-лес (глубинный лес), состоящий из ориентированных корневых деревьев. Тем самым, на множестве вершин орграфа G вводится отношение частичного порядка: $u < v$, если u, v принадлежат одной компоненте связности d-леса и u является потомком v. Отметим также, что поиск в глубину разбивает множество всех дуг орграфа на четыре класса:

1) древесные дуги, идущие от отца к сыну;

2) обратные дуги, идущие потомка к предку;

3) прямые дуги, идущие от предка к потомку, но не являющиеся древесными дугами;

4) поперечные дуги, соединяющие вершины, ни одна из которых не является потомком другой.

На рис. 8 показан орграф и его глубинный лес.

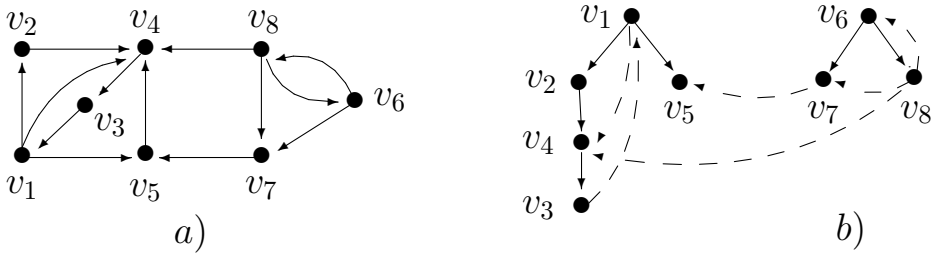


Рис. 8

Следующие два утверждения очевидны.

Лемма 1. *Если vu — поперечная дуга графа G, то $num[u] < num[v]$.*

Лемма 2. *Пусть u, v, w — такие вершины орграфа G, что $num[u] < num[v] < num[w]$ и w является потомком u. Тогда вершина v — потомок вершины u.*

Применим поиск в глубину для построения алгоритма, способного распознавать сильную связность орграфа G. На самом деле будет построен алгоритм, который сможет в орграфе G выделять *компоненты сильной связности*, являющиеся максимальными сильно связными подграфами орграфа G. Заметим, что компоненты сильной связности являются классами отношения взаимной достижимости на множестве вершин орграфа G.

Пусть $G_i = (V_i, E_i)$ $(1 \leqslant i \leqslant k)$ — компоненты сильной связности орграфа G. Обозначим через r_i такую вершину компоненты G_i $(1 \leqslant i \leqslant k)$, что $num[r_i] = \min num(V_i)$. Пусть r — преобразование множества вершин орграфа G, определенное правилом: $r(w) = r_i$ для любой вершины $w \in G_i$. Ясно, что вершины u и w лежат в одной компоненте сильной связности тогда и только тогда, когда $r(u) = r(w)$.

Лемма 3. *Если $w \neq r(w)$, то $w < r(w)$.*

Доказательство. Поскольку $r(w)$ и w взаимно достижимы, существует кратчайшая $(r(w), w)$-орцепь P длины $q \geqslant 1$. Проведем индукцию по q.

Пусть $q = 1$. Тогда существует дуга $r(w)w$. Ясно, что эта дуга древесная или прямая, поэтому w — потомок $r(w)$.

Допустим, что $q > 1$. Обозначим через x предпоследнюю вершину орцепи P. Легко проверяется, что вершины x и $r(w)$ взаимно достижимы, поэтому x и $r(w)$ лежат в одной компоненте сильной связности. Следовательно, к вершине x применимо предположение индукции. Таким образом, вершина x — потомок $r(w)$. Если xw — древесная или прямая дуга, то $w < x$ и потому $w < r(w)$. Пусть xw является обратной или поперечной дугой. Применяя лемму 1, получаем, что $num[w] < num[x]$. Так как $num[r(w)] < num[w]$ и x — потомок $r(w)$, вершины $r(w)$, w, x удовлетворяют условиям леммы 2, т. е. $w < r(w)$. □

Лемма 4. *Пусть u — произвольная вершина, удовлетворяющая неравенствам $r(w) > u > w$. Тогда $r(u) = r(w)$.*

Доказательство. Поскольку $r(w) > u > w$, существует $(r(w), w)$-орцепь P, проходящая через вершину u. Кроме того, $r(w)$ достижима из w. Следовательно, $r(w)$ и u взаимно достижимы, т. е. $r(u) = r(w)$. □

Пусть v — произвольная вершина орграфа G. Как и в разд. 2.2, через $\hat{v}$ обозначим поддерево d-леса с корнем v. Если $X \subseteq \hat{v}$, то множество всех таких вершин w, что $r(w) \geqslant v$ и для некоторой вершины $x \in X$ существует обратная или поперечная дуга xw, будем обозначать, как и раньше, через $VB(X, v)$.

Леммы 3 и 4 показывают, что вершины каждой компоненты сильной связности G_i, $1 \leqslant i \leqslant k$, образуют в глубинном лесе поддерево с корнем r_i.

Лемма 5. *Пусть v и w — такие вершины графа G, что $w \in VB(\hat{v}, v)$. Тогда $r(w) = r(v)$.*

Доказательство. Поскольку $w \in VB(\hat{v}, v)$, выполнено неравенство $r(w) \geqslant v$ и существует обратная или поперечная дуга xw, где $x \leqslant v$. Ясно, что существует $(r(w), x)$-орцепь P, содержащая вершину v. Кроме того, найдется $(w, r(w))$-орцепь Q. Из орцепей P, Q и дуги xw легко получить замкнутый ормаршрут, содержащий вершины v и w. Поэтому $r(w) = r(v)$. □

Теорема 2.4. *Пусть v — произвольная вершина орграфа G. Равенство $v = r(v)$ выполнено тогда и только тогда, когда $VB(\hat{v}, v) \subseteq \hat{v}$.*

Доказательство. Пусть v — такая вершина, что $v = r(v)$. Возьмем произвольную вершину $w \in VB(\hat{v}, v)$. В силу леммы 5 имеем $r(w) = r(v) = v$. Отсюда с учетом леммы 3 получаем, что $w \in \hat{v}$.

Предположим, что $v \neq r(v)$. Из леммы 3 следует, что $v < r(v)$. С другой стороны, существует $(v, r(v))$-орцепь P. В орцепи P найдем первую дугу xw такую, что w не принадлежит $\hat{v}$. Очевидно, xw — обратная или поперечная дуга. Кроме того, легко проверяется, что w и v лежат в одной компоненте сильной связности орграфа G. Следовательно, $r(w) = r(v) > v$. Таким образом, $w \in VB(\hat{v}, v)$, откуда вытекает, что $VB(\hat{v}, v) \not\subseteq \hat{v}$. □

На множестве V вершин графа G рассмотрим следующую функцию

$$L[v] = \min num(\{v\} \cup VB(\hat{v}, v)).$$

Теорема 2.5. *Пусть v — произвольная вершина орграфа G. Равенство $v = r(v)$ выполнено тогда и только тогда, когда $L[v] = num[v]$.*

Доказательство. Пусть $v = r(v)$. Тогда $VB(\hat{v}, v) \subseteq \hat{v}$. Поэтому $\min num(VB(\hat{v}, v)) \geqslant num[v]$. Отсюда $L[v] = num[v]$.

Предположим, что $v \neq r(v)$. Учитывая теорему 2.4, получаем, что существует $w \in VB(\hat{v}, v) \setminus \hat{v}$. Найдется такая вершина $x \in \hat{v}$, что xw — обратная или поперечная дуга. Следовательно, $num[w] < num[x]$. Проверим, что $num[w] < num[v]$. Рассуждая от противного, допустим, что $num[v] \leqslant num[w]$. Поскольку $w \neq v$, имеем строгое неравенство $num[v] < num[w]$. К вершинам v, w, x можно применить лемму 2, поэтому $w \in \hat{v}$, что невозможно. Из доказанного неравенства следует, что $L[v] \leqslant num[w] < num[v]$. □

Теорема 2.5 позволяет распознавать в каждой компоненте сильной связности вершины v, удовлетворяющие условию $v = r(v)$. Сравнивая теорему 2.5 с теоремой 2.2, мы видим, что роль таких вершин для орграфов аналогична роли точек сочленения для обыкновенных графов.

Лемма 6. *Пусть v — произвольная вершина орграфа G. Тогда $L[v] = \min\left(num[v], L[s_1], \ldots, L[s_p], num(VB(v, v))\right)$, где $s_1, \ldots, s_p$ — все сыновья вершины v.*

Эта лемма проверяется аналогично лемме 6 из разд. 2.2.

Перейдем к описанию алгоритма, позволяющего строить компоненты сильной связности орграфа G. Пусть к орграфу применен поиск в глубину. В компонентах сильной связности G_i, $1 \leqslant i \leqslant k$, ранее были выделены корневые вершины r_i. С этого момента будем считать, что компоненты сильной связности занумерованы следующим образом: если $1 \leqslant i < j \leqslant k$, то вершина r_i использована поиском в глубину раньше, чем вершина r_j. Такая нумерация компонент сильной связности обладает следующим важным свойством.

Лемма 7. *Компонента сильной связности G_i, $1 \leqslant i \leqslant k$, совпадает с множеством всех элементов из поддерева $\hat{r}_i$, не принадлежащих компонентам $G_1, \ldots, G_{i-1}$.*

Из леммы 7 легко извлекается следующий способ нахождения компонент сильной связности. Определим стек S и будем помещать в него вершины орграфа G в том порядке, в каком их просматривает поиск в глубину. Если сразу после того, как вершина r_i использована, вытолнуть из стека S все вершины до r_i включительно, мы получим компоненту G_i.

Теперь мы можем привести формальное описание алгоритма для отыскания компонент сильной связности орграфа G.

Алгоритм 2.3.

```
1.  procedure StrongComp(v);
2.  begin
3.    num[v] := i; L[v] := i; i := i + 1; S ⇐ v;
4.    for w ∈ list[v] do
5.      if num[w] = 0 then
6.        begin
7.          StrongComp(w); L[v] := min(L[v], L[w]);
8.        end
9.      else if num[w] < num[v] and w ∈ S then
10.       L[v] := min(L[v], num[w]);
11.   if L[v] = num[v] then
12.     while num[top(S)] ⩾ num[v] do
13.       begin
14.         x ⇐ S; добавить x к очердной
              компоненте сильной связности;
15.       end
```

16. **end**;
17. **begin**
18. $i := 1$; $S := nil$;
19. **for** $v \in V$ **do** $num[v] := 0$;
20. **for** $v \in V$ **do**
21. **if** $num[v] = 0$ **then** $StrongComp(v)$
22. **end**.

Процедура $StrongComp(v)$ является модифицированной версией рекурсивной процедуры $DFS(v)$ (см. разд. 8.1). Для данной вершины v значение функции $L[v]$ вычисляется циклом в строках 4 – 10 в соответствии с формулой, полученной в лемме 6. В строке 11 для вершины v проверяется условие из теоремы 2.5, означающее, что v — корневая вершина очередной компоненты сильной связности. Для нахождения компонент сильной связности использован упомянутый выше стек S, элементами которого являются вершины орграфа G (строки 12 – 15). Кроме того, этот стек участвует в формировании условия, проверяемого в строке 9. Выполнение этого условия означает, что $w \in VB(\hat{v}, v)$.

Мы изложили неформальные соображения по поводу правильности работы алгоритма 2.3. Перейдем к строгим рассуждениям.

Теорема 2.6. *Алгоритм* 2.3 *правильно строит компоненты сильной связности орграфа* G.

Доказательство. Занумеруем компоненты сильной связности G_i, $1 \leqslant i \leqslant k$, в порядке использования корневых вершин r_i поиском в глубину. В силу леммы 7 множество вершин V_1 компоненты G_1 совпадает с множеством $\hat{r}_1$, состоящим из вершины r_1 и всех ее потомков. В процессе работы процедуры $StrongComp(r_1)$ происходят обращения к процедуре $StrongComp(u)$ для каждого из потомков вершины r_1. Проверим, что ни в одной из этих процедур равенство $L[u] = num[u]$ не выполнено (здесь и далее речь идет о значениях функции L, вычисленных алгоритмом 2.3). Полагая противное, можно выбрать среди вершин, для которых равенство выполнено, минимальную вершину x (напомним, что d-лес является частично упорядоченным множеством). Ясно, что $x \neq r(x) = r_1$. Из теоремы 2.4 вытекает, что $VB(\hat{x}, x) \not\subseteq \hat{x}$. Поэтому существует обратная или поперечная дуга yz, где $y \in \hat{x}$. С использованием леммы 3 нетрудно проверить, что $num[z] < num[x]$. Выбор вершины x гарантирует, что все вершины, просмотренные поиском в глубину, содержатся в стеке S. Следовательно, при выполнении процедуры

$StrongComp(x)$ условие в строке 9 (считая, что $w = z$) выполнится. Поэтому текущее значение $L[x]$ не превосходит $num[z] < num[x]$. Отсюда следует, что значение $L[x]$, подсчитанное процедурой, будет меньше чем $num[x]$. Это противоречит выбору x.

Покажем, что в процессе выполнения процедуры $StrongComp(r_1)$ условие $L[r_1] = num[r_1]$ будет выполнено. Напомним, что множество $\hat{r}_1$ совпадает с множеством вершин компоненты G_1. Отсюда следует, что $VB(\hat{u}, u) \subseteq VB(\hat{r}_1, r_1)$ для любого потомка u вершины r_1. Из теоремы 2.4 вытекает, что $VB(\hat{r}_1, r_1) \subseteq \hat{r}_1$. При выполнении процедуры $StrongComp(u)$ все просмотренные вершины находятся в стеке S. Поэтому значения $L[u]$, уточняемые в строках 7 и 10 всегда будут не меньше чем $num[r_1]$. С учетом присваиваний в строке 3, имеем $L[r_1] = = num[r_1]$.

Для завершения доказательства применим индукцию по числу q компонент сильной связности орграфа G.

Пусть $q = 1$. Тогда $G = G_1$, т. е. все вершины орграфа G, отличные от вершины r_1, являются ее потомками. Условие в строке 11 выполнится только один раз при $v = r_1$. Поэтому из стека S будут вытолкнуты все вершины орграфа G.

Предположим, что $q > 1$. Пусть алгоритм 2.3 начинает работу с вершины v_0. В процессе работы алгоритма после завершения работы процедуры $DFS(r_1)$ из стека S будут вытолкнуты все вершины компоненты сильной связности G_1. Обозначим через G' подграф, полученный из орграфа G удалением компоненты G_1. Очевидно, к орграфу G' применимо предположение индукции. Следовательно, все компоненты, отличные от G_1, будут найдены правильно. □

2.4. Поиск в ширину

Рассмотрим еще один способ систематического обхода всех вершин обыкновенного графа G, называемый *поиском в ширину*. Для описания поиска в ширину введем в рассмотрение очередь Q, элементами которой являются вершины графа G. Поиск начинается с некоторой вершины v. Эта вершина помещается в очередь Q и с этого момента считается *просмотренной*. Затем все вершины, смежные с v включаются в очередь и получают статус просмотренных, а вершина v из очереди удаляется. Более общо, пусть в начале очереди находится вершина u. Обозначим через $u_1, \ldots, u_p$ вершины, смежные с u и еще непросмотренные. Тогда вершины $u_1, \ldots, u_p$ помещаются в очередь Q и с этого момента счита-

ются просмотренными, а вершина u удаляется из очереди и получает статус *использованной*. В этой ситуации вершина u называется *отцом* для каждой из вершин u_i ($u = father[u_i]$, $1 \leqslant i \leqslant p$). Каждое из ребер uu_i, $1 \leqslant i \leqslant p$, будем называть *древесным* ребром. В тот момент, когда очередь Q окажется пустой, поиск в ширину обойдет компоненту связности графа G. Если остались непросмотренные вершины (это возможно лишь в случае, когда граф G несвязен), поиск в ширину продолжается из некоторой непросмотренной вершины.

Поиск в ширину просматривает вершины в определенном порядке. Как и раньше, этот порядок фиксируется в массиве num. Если u — отец вершин $u_1, \ldots, u_p$, то $num[u_i] = num[u] + i$, $1 \leqslant i \leqslant p$ (для определенности мы полагаем, что сначала в очередь помещается u_1, затем u_2 и т. д.). Для начальной вершины v естественно положить $num[v] = 1$.

Поиск в ширину реализует описанная ниже процедура BFS (название процедуры является аббревиатурой от англ. breadth first search). Эта процедура использует описанные ранее массивы num и $father$. Кроме того, она вычисляет множество всех древесных ребер T. Массив num удобно использовать для распознавания непросмотренных вершин: равенство $num[u] = 0$ означает, что вершина u еще не просмотрена.

Алгоритм 2.4.

```
1.  procedure BFS(v);
2.  begin
3.    Q := nil; Q ⇐ v; num[v] := i; i := i + 1;
4.    while Q ≠ nil do
5.      begin
6.        u ⇐ Q;
7.        for w ∈ list[u] do
8.          if num[w] = 0 then
9.            begin
10.             Q ⇐ w; father[w] := u;
11.             num[w] := i; i := i + 1; T := T ∪ {uw};
12.           end
13.     end
14. end
15. begin
16.   i := 1; T := ∅;
17.   for v ∈ V do num[v] := 0;
18.   for v ∈ V do
```

19. **if** $num[v] = 0$ **then**
20. **begin** $father[v] := 0$; $BFS(v)$ **end**
21. **end**.

Полезно сравнить процедуру BFS с нерекурсивной версией процедуры DFS (см. стр. 22) и убедиться, что по существу, первая процедура получается из второй заменой стека на очередь.

Заметим также, что, применяя этот алгоритм к связному графу G, можно цикл в строках 18-20 заменить однократным обращением к процедуре BFS.

Теорема 2.7. *Пусть G — связный (n, m)-граф. Тогда*

1) *поиск в ширину просматривает каждую вершину в точности один раз;*

2) *поиск в ширину требует $O(n + m)$ операций;*

3) *подграф (V, T) графа G является деревом.*

Эта теорема доказывается аналогично теореме 2.1.

В связном графе G поиск в ширину из вершины v строит корневое дерево с множеством ребер T и корнем v. Это корневое дерево называется *деревом поиска в ширину* или, короче, *b-деревом*. Аналогично, если G произвольный обыкновенный граф, то поиск в ширину строит b-дерево в каждой компоненте связности графа G; объединяя эти деревья, мы получим остовный лес графа G, называемый в дальнейшем b-лесом этого графа.

На рис. 9 показаны связный граф G и его b-дерево.

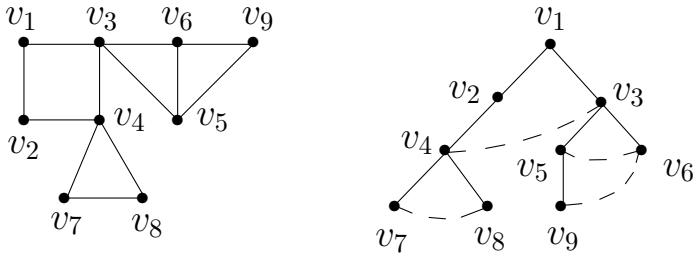


Рис. 9

Пусть в графе G проведен поиск в ширину. Занумеруем вершины графа в соответсвии с порядком, в котором поиск в ширину обходит вершины. А именно, обозначим вершины графа через w_i, $1 \leqslant i \leqslant n$, считая, что $num[w_i] = i$.

В леммах 1— 5 изучаются некоторые свойства поиска в ширину, отличающие его от поиска в глубину.

Лемма 1. *Вершина w_k является отцом вершины w_l тогда и только тогда, когда $k = \min\{i | w_i \in list(w_l)\}$.*

Доказательство. Пусть w_k — отец вершины w_l. Это значит, что непосредственно перед удалением w_k из очереди Q, вершина w_l не была просмотрена. Если w_p смежна с w_l в графе G и $p < l$, то w_p была удалена из очереди Q раньше, чем w_k. Поэтому отцом w_l оказалась бы вершина w_p, что невозможно.

Обратно, пусть k — наименьший из номеров вершин w_i, смежных с w_l в графе G. Ясно, при $p < k$ вершина w_p не смежна с вершиной w_l и потому не может быть ее отцом. Отсюда следует, что w_k — отец w_l. □

Из леммы 1 следует, что ребро, не являющееся древесным, никогда не соединяет предка с потомком в b-дереве; по этой причине такие ребра графа G будем называть *поперечными ребрами*.

Лемма 2. *Пусть вершины w_k, w_l являются отцами вершин w_p, w_q соответственно. Если $p \leqslant q$, то $k \leqslant l$.*

Доказательство. Предположим, что $l < k$. Из этого неравенства следует, что вершина w_l будет использована раньше, чем w_k. Поэтому вершина w_q, являющаяся сыном w_l, попадет в очередь раньше, чем сын w_p вершины w_k. Отсюда $q < p$, что невозможно. □

Напомним, что в корневом дереве через $h(u)$ мы обозначили уровень вершины u, равный расстоянию этой вершины от корня.

Лемма 3. *Если $1 \leqslant p \leqslant q \leqslant n$, то $h(w_p) \leqslant h(w_q)$.*

Доказательство. Требуемое неравенство очевидно, если w_p — корень b-дерева. Пусть w_p не является корнем. Обозначим через s наибольший из номеров p и q и применим индукцию по s. Рассмотрим вершины w_k, w_l, являющиеся отцами вершин w_p, w_q соответственно. В силу леммы 2 имеем $k \leqslant l$. Ясно, что к вершинам w_k, w_l применимо предположение индукции. Следовательно, $h(w_k) \leqslant h(w_l)$. Отсюда

$$h(w_p) = h(w_k) + 1 \leqslant h(w_l) + 1 = h(w_q).$$

Поскольку база индукции (при $s = 2$), очевидно, выполняется, лемма доказана. □

Лемма 4. *Если вершины w_p и w_q смежны в графе G и $p < q$, то $h(w_q) - h(w_p) \leqslant 1$.*

Доказательство. Пусть w_l — отец вершины w_q. Тогда из леммы 1 следует, что $l \leqslant p$. В силу леммы 3 имеем

$$h(w_l) \leqslant h(w_p) \leqslant h(w_q).$$

Поскольку $h(w_q) - h(w_l) = 1$, получаем

$$h(w_q) - h(w_p) \leqslant h(w_q) - h(w_l) = 1.$$

□

Лемма 5. *Расстояние в графе G от вершины w_1 (т. е. от корня b-дерева) до произвольной вершины u равно $h(u)$.*

Доказательство. Достаточно проверить, что для произвольной (w_1, u)-цепи

$$w_1 = v_0, v_1, \ldots, v_{s-1}, v_s = u$$

выполнено неравенство $s \geqslant h(u)$. Рассмотрим последовательность

$$0 = h(v_0), h(v_1), \ldots, h(v_{s-1}), h(v_s) = h(u), \qquad (1)$$

составленную из уровней вершин данной цепи. В силу леммы 4 соседние элементы последовательности (1) различаются не больше чем на 1. Отсюда вытекает, что последовательность (1) имеет наименьшую длину, если она является возрастающей. В этом случае последовательность должна иметь вид $0, 1, \ldots, h(u)$. Следовательно, для произвольной последовательности (1) выполнено неравенство $s \geqslant h(u)$. □

Пусть v_0 — корень b-дерева. Лемма (5) показывает, что простая (v_0, u)-цепь в b-дереве является кратчайшей (v_0, u)-цепью в графе G. Отсюда следует, что поиск в ширину может быть применен для решения следующей задачи: *в связном графе G найти кратчайшую цепь, соединяющую данную вершину v_0 с произвольной вершиной u.*

Для решения этой задачи необходимо в графе G из вершины v_0 провести поиск в ширину, а затем, используя массив $father$, построить требуемую кратчайшую цепь.

2.5. Алгоритм отыскания эйлеровой цепи в эйлеровом графе

С эйлеровыми графами мы познакомились в разд. **??**. Напомним, что замкнутая цепь в графе G называется эйлеровой, если она содержит все ребра и все вершины графа. Из теоремы **??** следует, что связный

неодноэлементный граф эйлеров тогда и только тогда, когда каждая его вершина имеет четную степень.

Этот раздел посвящен построению и анализу алгоритма, позволяющего в связном обыкновенном графе G с четными степенями вершин построить эйлерову цепь. В алгоритме используются два стека $SWork$ и $SRes$; элементами обоих стеков являются вершины графа G. Кроме того, введен массив $listW$, элементы которого — списки вершин. Мы считаем, что для каждой вершины v начальное значение $listW[v]$ совпадает со списком всех вершин, смежных с вершиной v, т. е. с $list(v)$.

Алгоритм 2.5.
Вход: связный граф $G = (V, E)$ без вершин нечетной степени, начальная вершина v_0.
Выход: эйлерова цепь, представленная последовательностью вершин в стеке $SRes$.

1. **begin**
2. $SWork := nil$, $SRes := nil$;
3. $SWork \Leftarrow v_0$;
4. **for** $v \in V$ **do** $listW[v] := list[v]$;
5. **while** $SWork \neq nil$ **do**
6. **begin**
7. $v := top(SWork)$;
8. **if** $listW(v) \neq \emptyset$ **then**
9. **begin**
10. $u :=$первая вершина $listW[v]$;
11. $SWork \Leftarrow u$;
12. $listW(v) := listW(v) \setminus \{u\}$;
13. $listW(u) := listW(u) \setminus \{v\}$;
14. **end**
15. **else**
16. **begin** $v \Leftarrow SWork$; $SRes \Leftarrow v$; **end**
17. **end**
18. **end**.

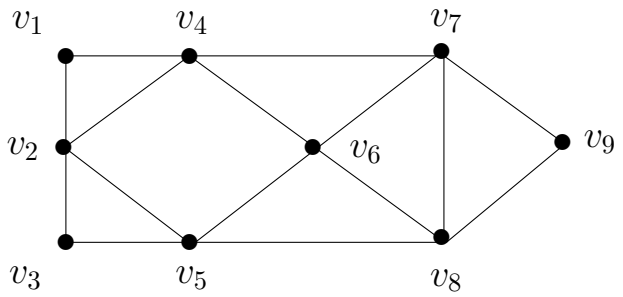
Принцип работы этого алгоритма состоит в следующем. Алгоритм начинает работу с некоторой вершины v_0 продвигаться по ребрам графа, причем каждое пройденное ребро из графа удаляется (строки 10 – 13). Ясно, что последовательное выполнение этой группы операторов позволяет выделить в графе некоторую замкнутую цепь. Затем начи-

нается выполнение группы операторов из строки 16. Эти операторы выталкивают очередную вершину из стека $SWork$ в стек $SRes$ до тех пор, пока не выполнится одно из условий

1) стек $SWork$ пуст (алгоритм заканчивает работу);

2) для вершины $v = top(SWork)$ существует непройденное ребро vu. В этом случае алгоритм продолжает работу из вершины u.

На рис. 10 изображен эйлеров граф и эйлерова цепь, построенная алгоритмом 2.5 (предполагается, что все списки вершин упорядочены по возрастанию номеров).



$v_1, v_2, v_3, v_5, v_2, v_4, v_6, v_5, v_8, v_6, v_7, v_8, v_9, v_7, v_4, v_1$

Рис. 10

Теорема 2.8. *Алгоритм 2.5 правильно строит эйлерову цепь в эйлеровом графе G.*

Доказательство. Заметим сначала, что из стека $SRes$ вершины никогда не выталкиваются. Отсюда следует, что начиная с некоторого момента работы алгоритма стек $SRes$ перестанет изменяться. Это произойдет после выполнения операторов в строке 16. Предположим, что стек $SWork$ в этот момент непуст. Если $v = top(SWork)$, операторы в строках 10 – 13 начнут добавлять вершины к стеку $SWork$ (это означает, что в графе G из вершины v можно построить цепь, состоящую из еще непройденных ребер). Ясно, что построение такой цепи должно прекратиться. Это означает, что мы прийдем в вершину u, для который все инцидентные ей ребра уже пройдены ($listW(u) = \emptyset$). После этого начнут выполняться операторы из строки 16, и, следовательно, к стеку $SRes$ добавится хотя бы одна вершина, что невозможно. Таким образом, стек $SWork$ рано или поздно обязательно станет пустым, что приведет к завершению работы алгоритма.

Пусть P — цепь, содержащаяся в стеке $SRes$ после окончания работы алгоритма. Легко понять, что вершина w помещается в стек $SRes$,

если все ребра, инцидентные с w, уже пройдены. Отсюда следует, что для любой вершины цепи P все ребра, инцидентные этой вершине, содержатся в цепи P. Поскольку G — связный граф, цепь P содержит все ребра графа G. Теперь легко понять, что P — эйлерова цепь. □

В заключение оценим сложность алгоритма 2.5. Для этого заметим, что при каждой итерации цикла либо к стеку $SWork$ добавляется вершина (это означает прохождение очередного ребра), либо вершина переносится из стека $SWork$ в $SRes$ (другими словами к строящейся эйлеровой цепи добавляется ребро). Отсюда следует, что число повторений цикла равно $O(m)$. Если позаботиться о том, чтобы время, необходимое для удаления вершины из списка $listW[v]$ было ограничено константой, то сложность алгоритма 2.5 будет равна $O(m)$.

3. Задача о минимальном остове

Пусть $G = (V, E)$ — связный обыкновенный граф. Напомним (см. разд. **??**), что его остовом называется остовный подграф, являющийся деревом. Остов (n, m)-графа легко найти поиском в глубину или поиском в ширину. Поскольку оба поиска имеют сложность $O(m + n)$ и в связном графе $m \geqslant n - 1$, остов связного (n, m)-графа можно найти за время $O(m)$.

Граф $G = (V, E)$ называется *взвешенным*, если задана функция $c : E \longrightarrow \mathbb{R}$. Это означает, что каждому ребру e такого графа поставлено в соответствие число $c(e)$, называемое *весом* или *стоимостью* ребра e. Иными словами, взвешенный граф — это тройка (V, E, c). Для произвольного ненулевого подграфа H его весом $c(H)$ будет называться сумма весов всех ребер подграфа H.

Остов T взвешенного графа G назовем *минимальным остовом*, если для любого остова T' выполнено неравенство $c(T) \leqslant c(T')$.

Этот раздел посвящен решению следующей задачи: в данном связном графе найти минимальный остов (*задача о минимальном остове*).

Пусть G — связный граф. Ациклический остовный подграф F из G будем называть *остовным лесом* графа G. В том случае, когда остовный лес графа G связен, он является остовом графа G. Ребро $e = uv$ называется *внешним* к остовному лесу F, если его концы лежат в разных компонентах связности леса F. Если H — некоторая компонента связности остовного леса F, то через $Ext(H)$ будет обозначаться множество всех внешних ребер, каждое из которых инцидентно некоторой вершине из H. Ясно, что $Ext(H)$ является сечением графа G.

Предположим, что F — остовный лес связного взвешенного графа G. Будем говорить, что F *продолжаем до минимального остова*, если существует такой минимальный остов T, что $F \leqslant T$.

Лемма 1. *Пусть остовный лес F продолжаем до минимального остова и H — одна из компонент связности леса F. Если e — ребро минимального веса из $Ext(H)$, то остовный лес $F + e$ продолжаем до минимального остова.*

Доказательство. Пусть T — такой минимальный остов графа G, что $F \leqslant T$ и $e \notin ET$. Из теоремы **??** следует, что подграф $T + e$ содержит единственный цикл C. В разд. **??** было показано, что любое сечение и любой цикл имеют четное число общих ребер. Поскольку ребро e является общим для сечения $Ext(H)$ и цикла C, найдется еще одно ребро

f, общее для $Ext(H)$ и C. В силу выбора ребра e имеем $c(f) \geqslant c(e)$. Ясно, что $T' = T + e - f$ является остовом графа G и $F + e \subseteq T'$. Кроме того,

$$c(T') = c(T) + c(e) - c(f) \leqslant c(T).$$

Учитывая, что T — минимальный остов, получаем $c(T') = c(T)$. Таким образом, T' — минимальный остов, содержащий лес $F + e$, т. е. $F + e$ продолжаем до минимального остова. □

Лемма 1 позволяет сконструировать два алгоритма построения минимального остова во взвешенном (n, m)-графе G. Пусть F_0 — остовный лес, являющийся нулевым графом. Ясно, что лес F_0 можно продолжить до остова. Оба алгоритма строят последовательность

$$F_0, F_1, \ldots, F_{n-1}, \tag{2}$$

состоящую из остовных лесов, причем $F_i = F_{i-1} + e_i$, где e_i — ребро, внешнее к остовному лесу F_{i-1}, $1 \leqslant i \leqslant n - 1$. Иногда указанную последовательность называют *растущим лесом*. Последовательность (2) строится таким образом, чтобы для каждого i, $1 \leqslant i < n - 1$ остовный лес F_i можно было продолжить до минимального остова. Ясно, что тогда F_{n-1} является минимальным остовом.

При переходе от F_{i-1} к F_i (т. е. при выборе ребра e_i) возможны две стратегии.

Стратегия 1. В качестве e_i выбираем ребро минимального веса среди всех ребер, внешних к остовному лесу F_{i-1}.

Пусть H — одна из двух компонент связности леса F_{i-1}, содержащая концевую вершину ребра e_i. Если F_{i-1} продолжаем до минимального остова, то в силу леммы 1 лес $F_i = F_{i-1} + e_i$ обладает тем же свойством.

Стратегия 2. Здесь предполагается, что каждый остовный лес F_i $(1 \leqslant i)$ из последовательности (2) имеет лишь одну неодноэлементную компоненту связности H_i. Удобно считать, что H_0 состоит из некоторой заранее выбранной вершины графа G. Таким образом, по существу, речь идет о построении последовательности

$$H_0, H_1, \ldots, H_{n-1}, \tag{3}$$

состоящей из поддеревьев графа G, причем $H_i = H_{i-1} + e_i$, где $e_i \in \in Ext(H_{i-1})$. Иногда указанную последовательность называют *растущим деревом*.

Последовательность (3) строится следующим образом.

В качестве H_0 берем произвольную вершину графа G. Пусть при некотором i, где $0 \leqslant i < n-1$, дерево H_i уже построено. В качестве e_{i+1}

выбираем ребро минимального веса из множества $Ext(H_i)$ и полагаем $H_{i+1} = H_i + e_{i+1}$.

Стратегия 1 реализуется алгоритмом Борувки-Краскала. Этот алгоритм впервые был изложен в работе О. Борувки в 1926 году. Однако, данная работа была практически забыта. Появление, а затем и широкое распространение компьютеров стимулировало интерес математиков к построению алгоритмов решения дискретных задач. В результате в 1956 году этот алгоритм был переоткрыт Краскалом.

Отметим, что алгоритм Борувки-Краскала является частным примером жадного алгоритма, описанного в разд. **??** (см. замечание, сделанное после доказательства теоремы **??**).

Одной из основных операций в алгоритме Борувки-Краскала является операция слияния деревьев. Для эффективной организации этого процесса будем использовать три одномерных массива — $name$, $next$, $size$, каждый длины n. Пусть F — произвольный член последовательности (2). Массив $name$ обладает следующим свойством: $name[u] = name[w]$ тогда и только тогда, когда вершины u и w лежат в одной компоненте связности леса F. С помощью массива $next$ задается кольцевой список на множестве вершин каждой компоненты связности леса F. Если $v = name[w]$, то $size[v]$ равно числу вершин в компоненте связности остовного леса F, содержащей вершину w.

Опишем процедуру $Merge(v, w, p, q)$, предназначенную для слияния двух деревьев $H_1 = (V_1, E_1)$ и $H_2 = (V_2, E_2)$ по ребру vw, внешнему к остовному лесу F. Предполагается, что $v \in V_1$, $w \in V_2$, $p = name[v]$, $q = name[w]$.

```
1.  procedure Merge(v, w, p, q);
2.  begin
3.    name[w] := p; u := next[w];
4.    while name[u] ≠ p do
5.      begin
6.        name[u] := p; u := next[u];
7.      end;
8.    size[p] := size[p] + size[q];
9.    x := next[v]; y := next[w];
10.   next[v] := y; next[w] := x;
11. end;
```

Отметим некоторые особенности работы этой процедуры. Объединение состоит, по-существу, в смене значений $name[w]$ для всех $w \in V_2$

(цикл 4-7). Отсюда следует несимметричность процедуры, а именно, сложности выполнения процедур $Merge(v, w, p, q)$ и $Merge(w, v, q, p)$ равны $O(|V_1|)$ и $O(|V_2|)$ соответственно. Строки 8-10 нужны для сохранения структур данных. В них происходит формирование одного кольцевого списка для элементов объединения $V_1 \cup V_2$. Для этого достаточно исправить значения двух элементов $next[v]$ и $next[w]$ и установить $size[p]$ равным числу элементов в множестве $V_1 \cup V_2$.

Теперь можно дать формальное описание алгоритма Борувки-Краскала. Предполагается, что очередь Q содержит ребра графа. Ради простоты предполагается, что очередь Q организована при помощи массива длины m. В алгоритме используется процедура $Sort(Q)$; эта процедура сортирует очередь Q по возрастанию весов ребер. Процедура $Sort(Q)$ реализует пирамидальную сортировку, поэтому ее сложность равна $O(m \log m) = O(m \log n)$, поскольку $m \leqslant n^2$.

Алгоритм 3.1 (Борувка, Краскал).
Вход: связный взвешенный граф $G = (V, E, c)$.
Выход: минимальный остов T графа G.

```
1.  begin
2.    Sort(Q);
3.    for v ∈ V  do
4.      begin
5.        name[v] := v; next[v] := v; size[v] := 1;
6.      end;
7.    T := ∅;
8.    while |T| ≠ n − 1 do
9.      begin
10.       vw ⇐ Q; p := name[v]; q := name[w];
11.       if p ≠ q then
12.         begin
13.           if size[p] > size[q] then
14.             Merge(w, v, q, p)
15.           else Merge(v, w, p, q);
16.           T := T ∪ {vw}
17.         end
18.     end
19. end.
```

Прокомментируем работу алгоритма 3.1. Цикл в строках 3-6 форми-

рует остовный лес F_0. В строке 11 проверяется принадлежность вершин v и w различным деревьям. Слияние деревьев происходит в строках 13-15.

Для данной вершины v обозначим через $r(v)$ число изменений значения $name[v]$ при работе алгоритма 3.1.

Лемма 2. *Для любой вершины v связного взвешенного $G = (V, E, c)$ выполнено неравенство $r(v) \leqslant \log |V|$.*

Доказательство. Требуемое неравенство очевидно, если $|V| = 1$. Пусть $|V| > 1$. Заметим, что при переходе от остовного леса F_{n-2} к минимальному остову F_{n-1} процедура $Merge$ срабатывает ровно один раз. Лес F_{n-2} состоит из двух деревьев. Пусть V_1, V_2 — множества вершин этих деревьев. Тогда $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$. Предположим, что $|V_1| \geqslant |V_2|$. Тогда $|V_1| \leqslant |V| - 1$, $|V_2| \leqslant |V|/2$. Нетрудно понять, что при слиянии множеств V_1 и V_2 значение $name[v]$ сохранится, если $v \in V_1$, и изменится, если $v \in V_2$. Применяя предположение индукции, получим

$$r(v) \leqslant \log |V_1| < \log |V|, \text{ если } v \in V_1,$$

$$r(v) \leqslant \log |V_2| + 1 \leqslant \log |V|, \text{ если } v \in V_2.$$

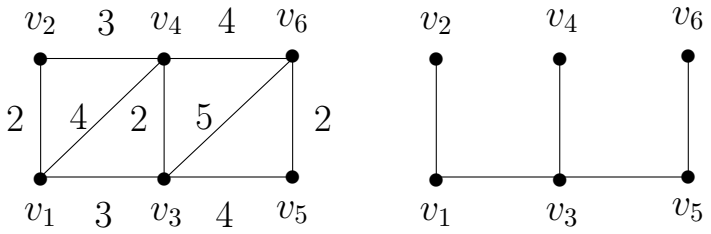
Лемма доказана. □

Теорема 3.1. *Вычислительная сложность алгоритма Борувки-Краскала для связного взвешенного (n, m)-графа равна $O(m \log n)$.*

Доказательство. Цикл в строках 8—18 проработает в худшем случае m раз. Оценим число операций, необходимых для однократного выполнения тела цикла. Заметим, что присваивание в строке 16 выполняется ровно $n - 1$ раз. Ясно, что столько же раз будет выполнена процедура $Merge$. Из леммы 2 следует, количество операций, выполненных при всех вызовах процедуры $Merge$ не превосходит $\sum\limits_{v \in V} r(v) \leqslant$ $\leqslant (n-1) \log n$. Отсюда сложность цикла 8-18 равна $O(m + n \log n) =$ $= O(m \log m) = O(m \log n)$, поскольку в связном графе выполнены неравенства $n - 1 \leqslant m \leqslant n^2$.

Осталось заметить, что процедура $Sort$ также требует $O(m \log m)$ операций. □

На рис. 11 *a*) изображен связный граф G. Числа, стоящие рядом с ребрами, означают веса ребер. Предполагается, что процедура $Sort$ упорядочивает ребра следующим образом: $v_1v_2, v_3v_4, v_5v_6, v_1v_3, v_2v_4, v_1v_4$, v_3v_5, v_4v_6, v_3v_6. Минимальный остов показан на рис. 11 *b*).



a) *b)*

Рис. 11

В приведенной ниже таблице указан порядок, в котором строился остов данного графа.

$\|T\|$	$name$	T
0	$v_1, v_2, v_3, v_4, v_5, v_6$	$\emptyset$
1	$v_1, v_1, v_3, v_4, v_5, v_6$	v_1v_2
2	$v_1, v_1, v_3, v_3, v_5, v_6$	v_1v_2, v_3v_4
3	$v_1, v_1, v_3, v_3, v_5, v_5$	v_1v_2, v_3v_4, v_5v_6
4	$v_1, v_1, v_1, v_1, v_5, v_5$	v_1v_2, v_3v_4, v_5v_6, v_1v_3
5	$v_1, v_1, v_1, v_1, v_1, v_1$	v_1v_2, v_3v_4, v_5v_6, v_1v_3, v_3v_5

Перейдем к рассмотрению алгоритма Ярника-Прима-Дейкстры, основанного на применении стратегии 2. Впервые он появился в работе Ярника, опубликованной в 1930 году, затем был переоткрыт Примом в 1957 году и независимо Дейкстрой в 1959 году. Заметим, что Дейкстра предложил очень эффективную реализацию этого алгоритма, связанную с расстановкой специальных меток.

Для ребра $e = vw$ вес $c(e)$ будет иногда обозначаться через $c(v, w)$. Напомним, что в обсуждаемом алгоритме строится последовательность (3), состоящая из деревьев, в которой дерево H_i получается из H_{i-1} поглощением ближайшей к дереву H_{i-1} вершины. Для организации эффективного выбора такой вершины используются два массива: $near[v]$ и $d[v]$. Пусть H — произвольное дерево из последовательности (3), U — множество его вершин. По определению $d[v]$ равно расстоянию от вершины v до множества U, иными словами

$$d[v] = \min\{c(v, u) | u \in U\}.$$

Пусть $d[v] = c(v, w)$, $w \in U$. Тогда $near[v] = w$. Иными словами, $near[v]$ — ближашая к v вершина из множества U.

Пусть $W = V \setminus U$. Будем считать, что если $vw \notin E$, то $c(v, w) = +\infty$. Через $Min(W)$ обозначим функцию, значением которой является вершина $v \in W$, имеющая минимальное значение метки d.

Алгоритм 3.2 (Ярник, Прим, Дейкстра).
Вход: связный взвешенный граф $G = (V, E, c)$, заданный матрицей весов $A[1..n, 1..n]$.
Выход: минимальный остов T графа G.

```
1.  begin
2.    w := произвольная вершина из V;
3.    W := V \ {w}; T := ∅;
4.    for v ∈ V  do
5.      begin
6.        near[v] := w; d[v] := A[v, w]
7.      end;
8.    while |T| ≠ n − 1 do
9.      begin
10.       v := Min(W); u := near[v];
11.       T := T ∪ {vu}; W := W \ {v};
12.       for u ∈ W  do
13.         if d[u] > A[u, v] then
14.           begin
15.             near[u] := v; d[u] = A[u, v];
16.           end
17.     end
18. end.
```

Теорема 3.2. *Алгоритм Ярника-Прима-Дейкстры применительно к связному взвешенному (n, m)-графу имеет сложность $O(n^2)$.*

Доказательство. Каждый проход цикла в строках 8-17 уменьшает на единицу число вершин во множестве W. После k проходов цикла 8-17 множество W будет содержать $n - k$ вершин. Следовательно, число операций, необходимых для выбора вершины $Min(W)$ пропорционально $n-k$ (строка 10). В строках 12-16 осуществляется пересчет меток для $n - k - 1$ вершин, т. е. число операций в цикле 12-16 пропорционально $n - k - 1$. Окончательно, число операций в алгоритме 3.2 пропорционально сумме

$$S = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2},$$

имеющей, очевидно, порядок n^2, что и доказывает теорему. □

Проиллюстрируем работу алгоритма 3.2 на графе, изображенном ранее на рис. 11. Здесь в качестве вершины w выбрана вершина v_1.

$\|T\|$	$U = V \setminus W$	$near$	d
		v_2, v_3, v_4, v_5, v_6	v_2, v_3, v_4, v_5, v_6
0	v_1	v_1, v_1, v_1, v_1, v_1	$2, 3, 4, \infty, \infty$
1	v_1, v_2	v_1, v_2, v_1, v_1	$3, 3, \infty, \infty$
2	v_1, v_2, v_3	v_3, v_3, v_3	2, 4, 5
3	v_1, v_2, v_3, v_4	v_3, v_4	4, 4
4	v_1, v_2, v_3, v_4, v_5	v_5	2
5	$v_1, v_2, v_3, v_4, v_5, v_6$		

В результате работы алгоритма получится остов

$$T = \{v_1v_2, v_1v_3, v_3v_4, v_3v_5, v_5v_6\}$$

(ребра остова перечислены в том порядке, в каком они были найдены). Этот остов изображен на рис. 11 *b*, т. е. он совпадает с остовом, построенным алгоритмом Борувки-Краскала. Заметим, что функция $Min(W)$ при наличии нескольких претендентов выбирала тот, у которого номер меньше. Например, в третьей строке таблицы вершине v_3 отдано предпочтение перед вершиной v_4. Аналогичная ситуация в предпоследней строке.

В научной литературе имеется много работ, посвященных различным вариантам постановки задачи о минимальном остове. Например, совсем недавно разработан алгоритм построения минимального остова в евклидовом графе, имеющий сложность $O(n \log n)$. Здесь под евклидовым графом понимается полный граф, вершинами которого являются точки n-мерного евклидова пространства, а вес каждого ребра равен расстоянию между его концами. Задача о минимальном остове для евклидовых графов на плоскости (случай n=2) находит непосредственное применение при проектировании радиоэлектронных изделий.

Детальное обсуждение задачи о минимальном остове содержится в книгах [29], [59] и [61].

4. Пути в сетях

4.1. Постановка задачи

Взвешенный орграф $G = (V, E, c)$ называется *сетью*. Сеть может быть представлена в компьютере матрицей весов дуг или списками смежности $\overleftarrow{list}[v]$ или $\overrightarrow{list}[v]$. В этой главе, следуя традиции, нам удобно ормаршрут называть *путем*.

Пусть P — некоторый (v, w)-путь:

$$v = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \ldots \xrightarrow{e_k} v_k = w.$$

Положим

$$c(P) = c(e_1) + c(e_2) + \ldots + c(e_k)$$

Число $c(P)$ назовем *длиной* пути P. Наименьшую из длин (v, w)-путей назовем *расстоянием* от v до w, а тот (v, w)- путь, длина которого равна расстоянию от v до w, будем называть *кратчайшим* (v, w)-путем. Ясно, что расстояние от v до w может отличаться от расстояния от w до v.

Задача о кратчайшем пути (ЗКП) между фиксированными вершинами формулируется следующим образом: в заданной сети G с двумя выделенными вершинами s и t найти кратчайший (s, t)-путь.

Отметим сразу, что ЗКП можно рассматривать и в неориентированных взвешенных графах, заменив каждое ребро vw графа двумя дугами vw, wv и считая, что веса обеих дуг равны весу ребра vw. Далее, если в произвольном графе положить вес каждого ребра равным единице, то получим данное ранее определение длины пути как числа входящих в него ребер. С этой точки зрения рассмотренная ранее (см. разд. 8.4) задача построения кратчайшего по числу ребер пути есть частный случай сформулированной только что задачи.

4.2. Общий случай. Алгоритм Форда-Беллмана

Всюду в дальнейшем будем предполагать, что если вершины v и w не являются смежными в сети $G = (V, E, c)$, то $c(v, w) = \infty$. Для удобства изложения и во избежание вырожденных случаев при оценке сложности алгоритмов будем считать, что $n \leqslant m$. Это исключает ситуации, при которых большинство вершин изолированы. Кроме того, будем рассматривать только такие сети, в которых нет контуров отрицательной длины. Понятно, что если сеть содержит контур отрицательной длины, то расстояние между некоторыми парами вершин становится неопределенным, поскольку, обходя этот контур достаточное число раз, можно

построить путь между этими вершинами с длиной, меньшей любого, наперед заданного, вещественного числа.

Метод решения ЗКП в некотором смысле аналогичен методу построения кратчайшего по числу ребер пути. Вначале размечаем все вершины данной сети (прямой ход алгоритма), вычисляя расстояния от s до всех вершин. Затем, используя специальные метки, обратным ходом строим требуемый путь. Интересно отметить, что для вычисления расстояния от s до заданной вершины t, мы вынуждены вычислять расстояния от s до всех вершин сети. В настоящее время не известен ни один алгоритм нахождения расстояния между фиксированными вершинами, который был бы существенно более эффективным, чем известные алгоритмы вычисления расстояний от одной из фиксированных вершин до всех остальных.

Начнем с первого этапа — вычисления расстояний от s до всех вершин. Метод, который мы будем здесь использовать, часто называют динамическим программированием, а алгоритм вычисления расстояний — алгоритмом Форда-Беллмана. Появился этот алгоритм в работе Форда 1956 года и в работе Беллмана 1958 года.

Основная идея алгоритма Форда-Беллмана заключается в поэтапном вычислении кратчайших расстояний. Обозначим через $d_k(v)$ длину кратчайшего среди всех (s, v)-путей, содержащих не более чем k дуг. Легко видеть, что справедливы следующие неравенства

$$d_1(v) \geqslant d_2(v) \geqslant \ldots \geqslant d_{n-1}(v).$$

Поскольку по предположению в графе нет контуров отрицательной длины, кратчайший (s, v)-путь не может содержать более чем $n-1$ дуг. Поэтому величина $d_{n-1}(v)$ дает искомое расстояние от s до v.

Для вычисления $d_{n-1}(v)$ достаточно последовательно вычислять $d_k(v)$ для всех $k = 1, ..., n-1$.

Значения $d_1(v)$ вычисляются просто:

$$d_1(v) = c(s, v) \text{ для всех } v \in V.$$

Пусть значения $d_{k-1}(v)$ вычислены для всех $v \in V$. Легко видеть, что

$$d_{k+1}(v) = \min\{d_k(v), d_k(w) + c(w, v) | w \in V\}$$

.

Организовать все эти вычисления можно с помощью всего лишь одного одномерного массива D длины n.

Положив $D[v] = c(s, v)$ для всех вершин $v \in V$, будем иметь равенства

$$D[v] = d_1(v).$$

Просматривая после этого все вершины v, произведем пересчет значений $D[v]$ по формуле

$$D[v] = \min\{D[v], D[w] + c(w, v) | w \in V\}. \quad (1)$$

После завершения первого пересчета значений $D[v]$ для всех v, будем иметь неравенства $D[v] \leqslant d_2(v)$. Почему не равенства? Пусть пересчет начинался с вершины v_1. Ясно, что тогда $D[v_1] = d_2(v_1)$. Предположим, что следующей вершиной, для которой был сделан пересчет, была вершина v_2. Тогда $D[v_2] \leqslant D[v_1] + c(v_1, v_2)$, что вытекает из формулы пересчета (1). Отсюда следует, что возможна ситуация, в которой $D[v_2] = D[v_1] + c(v_1, v_2)$, и, кроме того, значение $D[v_1]$ могло быть получено на пути, состоящем из двух дуг. Следовательно, значение $D[v_2]$ может быть получено по некоторому пути из трех дуг, т. е. $D[v_2] \leqslant d_2(v_2)$.

Повторив $n - 2$ раза процесс пересчета $D[v]$, будем иметь равенства $D[v] = d_{n-1}(v)$, т. е. $D[v]$ дает расстояние от s до v. Прежде чем подробнее обосновать равенства $D[v] = d_{n-1}(v)$, дадим формальное описание алгоритма Форда-Беллмана. Построения самих кратчайших путей удобно вести с помощью одномерного массива $Previous$ длины n, где $Previous[v]$ дает имя вершины, предпоследней в кратчайшем (s, v)-пути.

Алгоритм 4.1 (Форд, Беллман).
Вход: сеть $G = (V, E, c)$, заданная матрицей весов A порядка n; вершины s и t.
Выход: расстояния $D[v]$ от s до всех вершин $v \in V$, стек S, содержащий кратчайший (s, t)-путь, или сообщение, что искомого пути в сети не существует.

```
1.  procedure Distance
2.  begin
3.    D[s] := 0; Previous[s] := 0;
4.    for v ∈ V \ {s}  do
5.      begin D[v] := A[s, v]; Previous[v] := s end;
6.    for k := 1 to n − 2 do
```

```
7.      for v ∈ V \ {s}  do
8.        for w ∈ V  do
9.          if D[w] + A[w, v] < D[v] then
10.           begin
11.             D[v] := D[w] + A[w, v];
12.             Previous[v] := w
13.           end
14.   end;
15.   begin
16.     Distance;
17.     if D[t] < ∞ then
18.       begin
19.         S := nil; S ⇐ t; v := t;
20.         while Previous[v] ≠ 0 do
21.           begin v := Previous[v]; S ⇐ v end
22.       end
23.     else writeln ("Not exists");
24.   end.
```

Напомним, что по определению матрицы весов справедливы равенства $A[v, w] = c(v, w)$ для всех $vw \in E$ и по нашему соглашению $A[v, w] = \infty$, если в сети нет дуги vw.

Вернемся к обоснованию равенства $D[v] = d_{n-1}(v)$. Заметим, что при первом входе в цикл, начинающийся в строке 6, справедливы равенства $D[v] = d_1(v)$ для всех $v \in V$.

Предположим, что при входе в k-ю итерацию цикла 6, справедливы неравенства $D[v] \leqslant d_k(v)$ для всех $v \in V$. Докажем, что по завершению этой итерации для любой вершины $v \in V$ справедливы соотношения

$$D[v] \leqslant d_{k+1}(v).$$

Действительно, после выполнения цикла в строке 6 для произвольной вершины v выполнены неравенства

$$D[v] \leqslant D[w] + A[w, v] \text{ для всех } w \in V.$$

По предположению имеем неравенство $D[w] \leqslant d_k(w)$. Учитывая, что $A[w, v] = c(w, v)$, получаем неравенства $D[v] \leqslant d_k(w) + c(w, v)$ для всех вершин w.

В частности, это неравенство выполнено и для той вершины w, которая является предпоследней в кратчайшем (s, v)-пути, состоящем из

$k+1$ дуги. Отсюда сразу следует требуемое неравенство $D[v] \leqslant d_{k+1}(v)$.

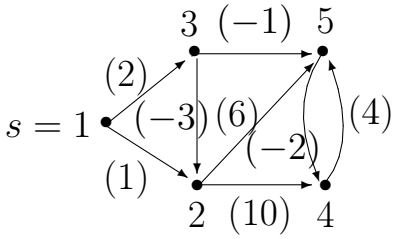
Таким образом, по завершению $(n-2)$-й итерации цикла в строке 6 справедливы неравенства $D[v] \leqslant d_{n-1}(v)$ для всех вершин v. Выше уже отмечалось, что поскольку сеть не содержит контуров отрицательной длины, то $d_{n-1}(v)$ дает длину кратчайшего (s, v)-пути. Следовательно, $D[v] = d_{n-1}(v)$. Тем самым обоснование этого равенства, а вместе с ним и обоснование корректности алгоритма Форда-Беллмана, завершено.

Теорема 4.1. *Алгоритм Форда-Беллмана имеет сложность* $O(n^3)$.

Доказательство. Понятно, что сложность алгоритма определяется сложностью процедуры *Distance*, так как основная программа требует числа операций пропорционального n. Ясно, что количество операций в цикле 4 пропорционально n. Поскольку цикл в строке 6 выполняется $n-2$ раза, в строке 7 — $n-1$ раз, а в строке 8 — n раз, и число выполнений оператора присваивания при каждом входе в цикл 8 ограниченно константой, то сложность выполнения цикла, начинающегося в строке 6, есть $O((n-2)\cdot(n-1)\cdot n) = O(n^3)$. □

Заметим, что вычисления в процедуре *Distance* можно завершить при таком выходе из цикла по k, при котором не происходит изменения ни одного значения $D[v]$. Это может произойти и при $k < n-2$, однако, такая модификация алгоритма не изменяет существенным образом его сложности, поскольку в худшем случае придется осуществить $n-2$ итерации цикла по k.

Иллюстрирует работу алгоритма Форда-Беллмана рис. 12. Веса дуг даны в скобках. Циклы в строках 7 и 8 выполнялись в порядке возрастания номеров вершин.



k	D[1]	D[2]	D[3]	D[4]	D[5]
	0	1	2	∞	∞
1	0	−1	2	9	1
2	0	−1	2	−1	1
3	0	−1	2	−1	1

Рис. 12

Рекомендуем читателю провести вычисления для этой сети в предположении, что вершины встречаются в порядке 1, 3, 2, 5, 4. (В этом случае расстояния будут вычислены за один проход, т. е. строки таблицы, соответствующие всем значениям k, будут одинаковыми.)

В случае, когда сеть задана списками смежностей $\overrightarrow{list}[v]$, для решения ЗКП в алгоритме Форда-Беллмана достаточно цикл в строке 8 процедуры $Distance$ записать следующим образом.

7. **for** $w \in \overrightarrow{list}[v]$ **do**
8. **if** $D[w] + A[w, v] < D[v]$ **then**
9. **begin** $D[v] := D[w] + c[v, w]$; $Previous[v] := w$ **end**

В таком случае алгоритм Форда-Беллмана будет иметь вычислительную сложность $O(mn)$.

4.3. Случай неотрицательных весов. Алгоритм Дейкстры

Описываемый в этом разделе алгоритм позволяет вычислять в сети с неотрицательными весами расстояния от фиксированной вершины s до всех остальных вершин и находить кратчайшие пути более эффективно, чем алгоритм Форда-Беллмана. Этот алгоритм был предложен в 1959 году Дейкстрой. В основе алгоритма Дейкстры лежит принцип «жадности», заключающийся в последовательном вычислении расстояний сначала до ближайшей к s вершине, затем до следующей ближайшей и т. д.

Для удобства изложения обозначим через $d(v)$ расстояние от s до v, т. е. длину кратчайшего (s, v)-пути в сети G.

Первая ближайшая к вершине s вершина v вычисляется просто: это сама вершина s, находящийся на нулевом расстоянии от s, т. е. $d(s) = 0$. Пусть ближайшие k вершин к вершине s определены и для всех них вычислены расстояния $d(v)$, т. е. определено множество $S = \{v_1 = s, v_2, \ldots, v_k\}$, причем выполняются неравенства:

1) $0 = d(v_1) \leqslant d(v_2) \leqslant \ldots \leqslant d(v_k)$;

2) $d(v_k) \leqslant d(v)$, для всех $v \in F$, где $F = V \setminus S$.

Здесь следует иметь ввиду, что последнее неравенство имеет «потенциальный» характер, а именно, мы считаем известными значения расстояний лишь для вершин $v_1, \ldots, v_k$, а для всех остальных вершин расстояния еще не вычислены, но для них известно, что неравенства 2) будут выполняться.

Найдем следующую ближайшую к s вершину сети G. Для каждого $w \in F$ положим

$$D(w) = \min\{d(v) + c(v, w) | v \in S\},$$

Можно отметить, что тогда $D(w)$ определяет длину минимального (s,w)-пути среди всех (s,w)-путей, все вершины в котором, кроме w, принадлежат S.

Выберем теперь такую вершину $w^* \in F$, что выполнено условие:

$$D(w^*) = \min\{D(w) | w \in F\}.$$

Оказывается, что вершина w^* является самой близкой к s среди всех вершин, не входящих в S (она является следующей $(k+1)$-ой ближайшей к s вершиной), и, более того, расстояние от вершины s до вершины w^* в точности равно $D(w^*)$, т. е. справедливо равенство $d(w^*) = D(w^*)$.

Обоснуем вначале равенство $d(w^*) = D(w^*)$. Пусть $P: \ s = w_0, w_1 \ldots, w_r = w^*$ — произвольный (s,w^*)-путь в сети G. Достаточно доказать неравенство $D(w^*) \leqslant c(P)$. Среди всех вершин пути P выберем вершину w_j с наименьшим номером среди тех, которые не входят в множество S. Так как начальная вершина пути P входит в S, а конечная — не входит в S, то такой номер j найдется. Итак $w_{j-1} \in S$, $w_j \in F$. Тогда, из определения $D[w]$, где $w \in F$, и определения стоимости пути вытекают неравенства

$$\begin{aligned} D(w_j) &\leqslant d(w_{j-1}) + c(w_{j-1}, w_j) \leqslant \\ &\leqslant c(w_0, w_1) + \ldots + c(w_{j-1}, w_j) = \mathrm{c}(Q), \end{aligned}$$

где через Q обозначен (s,w_j)-подпуть пути P. Из условия неотрицательности весов дуг вытекает, что $c(Q) \leqslant c(P)$. Кроме того, по выбору вершины w следует, что $D(w^*) \leqslant D(w_j)$. С учетом этих неравенств получаем, что $D(w^*) \leqslant c(P)$. Следовательно, $d(w^*) = D(w^*)$.

Осталось убедиться, что вершина w^* является $(k+1)$-ой ближайшей к s вершиной. Для этого достаточно доказать неравенство $d(w^*) \leqslant d(w)$ для всех $w \in F$. Зафиксируем (s,w^*)-путь P_1 и (s,w)-путь P_2, для которых $c(P_1) = d(w^*)$ и $c(P_2) = d(w)$. В силу доказанного только что равенства $D(w^*) = d(w^*)$ можно считать, что в пути P_1 все вершины, кроме w^*, лежат в S. Тогда, по выбору вершины w^*, справедливо неравенство $D(w^*) \leqslant D(w)$, и, повторяя вышеприведенные рассуждения, приходим к неравенству $D(w) \leqslant c(P)$. Отсюда $c(P_1) = d(w^*) = D(w^*) \leqslant c(P_2) = d(w)$, что и требовалось доказать.

Итак, выбирая вершину $w \in F$ с минимальным значением $D[v]$ и добавляя к S, мы расширяем множество вершин, до которых вычислено расстояние, на один элемент. Следовательно, повторяя $n-1$ раз процесс расширения множества S, мы вычислим расстояния до всех вершин сети G.

При формальной записи алгоритма Дейкстры (алгоритм 4.2) ход вычисления расстояний от s до остальных вершин v будем отражать в массиве D. По окончании работы алгоритма равенства $D[v] = d(v)$ будут выполняться для всех $v \in V$. Без формального описания используется функция $Min(F)$, которая возвращает вершину $w \in F$ такую, что справедливо равенство $D[w] = \min\{D[v] | v \in F\}$, иначе говоря, она находит тот самый элемент, который следует добавить к S и удалить из F. Мы ограничимся только вычислением расстояний и меток $Previous$, с помощью которых кратчайшие пути строятся также, как в алгоритме Форда-Беллмана.

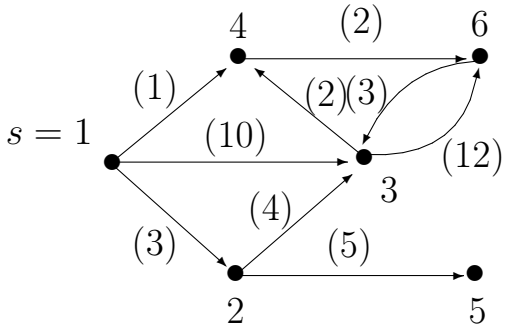
Алгоритм 4.2 (Дейкстра).
(∗ нахождение расстояний от фиксированной вершины до всех остальных в сети с неотрицательными весами ∗)
Вход: сеть $G = (V, E, c)$, заданная матрицей весов A порядка n; выделенная вершина s.
Выход: расстояния $D[v]$ от s до всех вершин $v \in V$, $Previous[v]$ — предпоследняя вершина в кратчайшем (s, v)-пути.

```
1.  begin
2.    D[s] := 0; Previous[s] := 0; F := V \ {s};
3.    for v ∈ F  do
4.      begin D[v] := A[s, v]; Previous[v] := s; end;
5.    for k := 1 to n − 1 do
6.      begin
7.        w := Min(F); F := F \ {w};
8.        for v ∈ F  do
9.          if D[w] + A[w, v] < D[v] then
10.           begin
11.             D[v] := D[w] + A[w, v];
12.             Previous[v] := w;
13.           end
14.     end
15. end.
```

Для доказательства корректности алгоритма 4.2 заметим, что при входе в очередную k-ую итерацию цикла 5-13 уже определены k ближайших к s вершин и вычислены расстояния от s до каждой из них. При этом первоначальные значения расстояний вычислены в строке 4. Как

показано выше, выполнение строки 7 правильно определяет $(k+1)$-ю ближайшую к s вершину. Удаление w из F влечет, что $D[w]$ больше меняться не будет, а по доказанному ранее пересчитывать его нет нужды, так как выполняется равенство $D[w] = d(w)$. Осталось заметить, что выполнение строки 9 в каждой итерации цикла 5-13 позволяет правильно пересчитывать значения $D[v]$ и отслеживать предпоследнюю вершину в кратчайшем пути. Следовательно, алгоритм 4.2 правильно вычисляет расстояния и кратчайшие пути от фиксированной вершины до всех остальных вершин сети.

Работа алгоритма Дейкстры показана на рис. 13.



k	$S = V \setminus F$	$D[1]$	$D[2]$	$D[3]$	$D[4]$	$D[5]$	$D[6]$
	$\{1 = s\}$	0	3	10	1	∞	∞
1	$\{1, 4\}$		3	10		∞	3
2	$\{1, 4, 2\}$			7		8	3
3	$\{1, 4, 2, 6\}$			6		8	
4	$\{1, 4, 2, 6, 3\}$					8	
5	$\{1, 4, 2, 6, 3, 5\}$						

Рис. 13

Теорема 4.2. *Алгоритм Дейкстры имеет сложность* $O(n^2)$.

Доказательство. Пусть найдены расстояния до k ближайших к s вершин. Определение расстояния до $(k+1)$-й вершины требует в строке 7 числа операций пропорционального $(n-k)$, так как именно столько вершин находится в множестве F. Проверка условия в строке 9 и, если нужно, пересчет $D[v]$ и $Previous[v]$ в строках 11, 12 требует числа операций пропорционального $(n-k)$. Окончательно, общее число операций в алгоритме Дейкстры пропорционально $n+(n-1)+\ldots+1 = n(n-1)/2$, т. е. равно $O(n^2)$. $\square$

Можно дать и другую, графическую интерпретацию работы алгоритма Дейкстры, построив *дерево кратчайших путей* K. Это дерево является орграфом на том же множестве вершин, что и G. Дуга vw включается в K, если w — ближайшая $(k+1)$-ая вершина и $v = Previous[w]$. Ясно, что K — это корневое дерево с корнем в s, поэтому для всякой вершины $v \in V$ в K существует единственный (s, v)-путь. Этот путь является кратчайшим (s, v)-путем в сети G. Для сети, изображенной ранее на рис. 13, ход построения дерева кратчайших путей показан ниже на рис. 14.

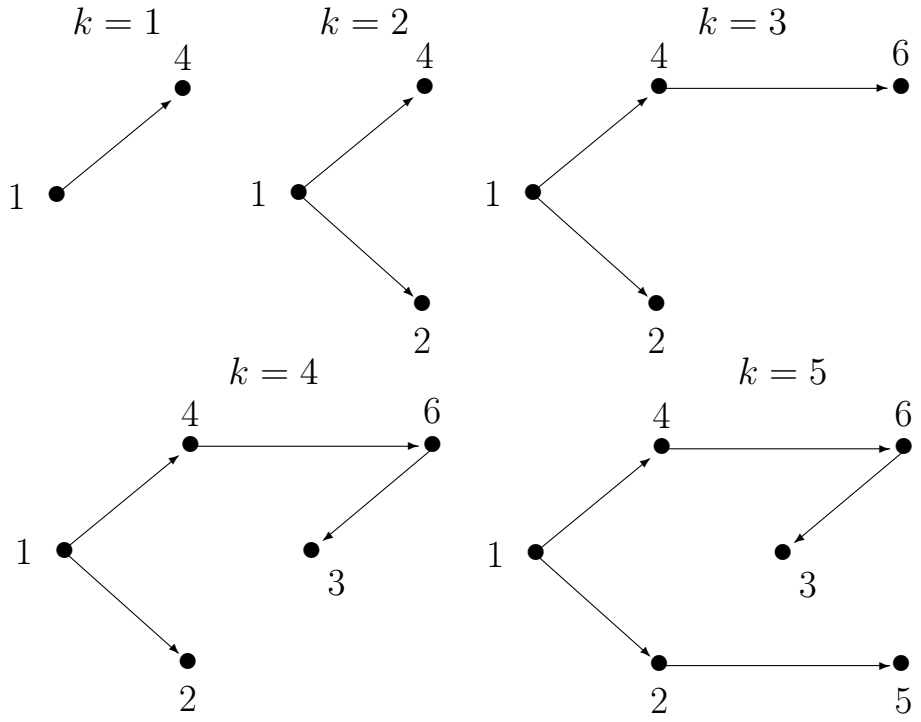


Рис. 14

4.4. Случай бесконтурной сети

В этом случае, так же как и в случае сетей с неотрицательными весами дуг, известен более эффективный алгоритм вычисления расстояний от фиксированной вершины до всех остальных, чем алгоритм Форда-Беллмана. В основе этого алгоритма лежат следующие две леммы.

Лемма 1. *В каждом бесконтурном орграфе имеется хотя бы одна вершина, полустепень исхода которой равна нулю.*

Доказательство. Пусть $G = (V, E)$ — бесконтурный орграф и w_1 — произвольная его вершина. Если ее полустепень исхода не равна нулю,

то выберем произвольную вершину w_2 такую, что $w_1w_2 \in E$, затем w_3 так, что $w_2w_3 \in E$, и т. д. до тех пор, пока подобный выбор вершины возможен. Через конечное число шагов мы дойдем до некоторой вершины w, из которой не выходит ни одна дуга, ибо в бесконтурном орграфе вершины в строящемся пути $w_1, w_2, w_3, \ldots$, не могут повторяться. Следовательно, последняя построенная в пути вершина w имеет нулевую полустепень исхода. □

Лемма 2. *Вершины бесконтурного ориентированного графа можно перенумеровать так, что каждая дуга идет из вершины с меньшим номером в вершину с большим номером.*

(Орграфы с так пронумерованными вершинами иногда называют *топологически отсортированными*, а алгоритм, осуществляющий такую нумерацию вершин — *алгоритмом топологической сортировки вершин*.)

Доказательство. Приведем алгоритм, осуществляющий топологическую сортировку. Неформально этот алгоритм можно сформулировать следующим образом:

1) объявить наибольшим неиспользованным номером число, равное количеству вершин в орграфе;

2) выбрать произвольную вершину v, полустепень исхода которой равна нулю, и присвоить вершине v наибольший из еще не использованных номеров. Номер, который получит вершина v, считать использованным;

3) удалить из орграфа вершину v вместе со всеми входящими в нее дугами;

4) повторять шаги 2 и 3 до тех пор, пока все вершины не получат свой номер.

Корректность работы приведенного алгоритма топологической сортировки вытекает из леммы 1, так как при каждом удалении вершины новый граф остается бесконтурным, и, следовательно, в нем также существует вершина с нулевой полустепенью исхода. □

При формальном описании этого алгоритма переменная $number$ дает значение самого большого из еще не использованных номеров. Переменная $DegOut[v]$ указывает на текущее значение полустепени исхода вершины v. В частности, удаление вершины v вместе со всеми выходящими из нее дугами приводит к уменьшению значения $DegOut[w]$ на единицу для всех $w \in \overrightarrow{list}[v]$. Очередь Q служит для накопления текущего множества вершин, имеющих нулевую полустепень исхода. Массив $Index$

предназначен для хранения новых номеров вершин. В этом разделе нам будет удобнее считать, что все сети и орграфы заданы списками смежностей $\overrightarrow{list}[v]$, где $w \in \overrightarrow{list}[v]$, если и только если имеется дуга из w в v.

Алгоритм 4.3.
Вход: бесконтурный орграф $G = (V, E)$, заданный списками смежностей $\overrightarrow{list}[v]$.
Выход: массив $Index$ длины n такой, что для любой дуги $vw \in E$ справедливо неравенство $Index[v] < Index[w]$.

```
1.  begin
2.    for v ∈ V  do  DegOut[v] := 0;
3.    for v ∈ V  do
4.      for w ∈ list→[v]  do  DegOut[w] := DegOut[w] + 1;
5.    Q := nil; number := n;
6.    for v ∈ V  do
7.      if DegOut[v] = 0 then Q ⇐ v;;
8.      while Q ≠ nil do
9.        begin
10.         v ⇐ Q; Index[v] := number;
11.         number := number − 1;
12.         for w ∈ list→[v]  do
13.           begin
14.             DegOut[w] := DegOut[w] − 1;
15.             if DegOut[w] = 0 then Q ⇐ v;
16.           end
17.       end
18. end.
```

В алгоритме 4.3 в цикле в строках 3 и 4 вычисляется полустепень исхода каждой вершины. Затем все вершины с нулевой полустепенью исхода помещаются в очередь Q (цикл 6-7). В строках 10 и 11 очередной вершине присваивается наибольший из неиспользованных номеров, иначе говоря, реализуется шаг 2 неформального описания алгоритма. Цикл в строках 12-15 обеспечивает удаление последней пронумерованной вершины вместе с дугами ей инцидентными, и все вершины, полустепень исхода которых в новом орграфе равна нулю, сразу же помещаются в очередь Q (шаг 3 неформального описания).

Легко видеть, что каждая вершина помещается в очередь Q либо тогда, когда ее полустепень исхода в заданном орграфе равна нулю, либо тогда, когда все вершины, следующие за ней, получат свои новые номера. Поэтому алгоритм 4.3 правильно осуществляет топологическую сортировку вершин.

Теорема 4.3. *Алгоритм* 4.3 *имеет сложность* $O(m)$.

Доказательство. Напомним, что на протяжении этой главы мы условились считать, что $n \leqslant m$. Циклы в строках 2 и 6-7 анализируют каждую вершину ровно по одному разу, а в строках 3-4 и 12-15 — каждую дугу также ровно по одному разу. Следовательно, сложность алгоритма 4.3 есть $O(n) + O(m) = O(m)$. □

В тех случаях, когда бесконтурный орграф задан списками смежностей $\overleftarrow{list}$, топологическая сортировка вершин орграфа также может быть осуществлена за время $O(m)$.

При описании алгоритма вычисления расстояний в бесконтурной сети будем считать, что все вершины заданной сети топологически отсортированы. Расстояния будем вычислять от вершины $v_1 = s$.

Пусть v_k — произвольная вершина заданной бесконтурной сети. Тогда любой (s, v_k)-путь проходит через вершины с меньшими чем k номерами. Из этого замечания следует, что для вычисления расстояний от s до всех остальных вершин сети достаточно последовательно вычислять расстояния от s до v_2, затем от s до v_3 и так далее. Пусть, как и в предыдущих разделах, $d(v)$ обозначает расстояние от s до v. Тогда $d(v_1) = 0$, и если $d(v_r)$ для всех $r < k$ вычислено, то

$$d(v_k) = \min\{d(v_r) + c(v_r, v_k) | r = 1, 2, ..., k\}. \qquad (2)$$

Корректность формулы (2) легко проверяется при помощи индукции. Именно по этой формуле вычисляет расстояния от вершины $s = v_1$ предлагаемый ниже алгоритм, в котором переменные $D[v]$ и $Previous[v]$ имеют тот же смысл, что и в алгоритмах Форда-Беллмана и Дейкстры.

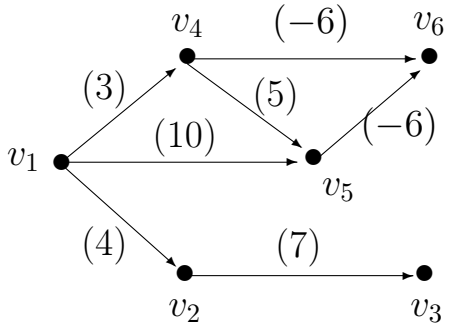
Алгоритм 4.4.
(∗ Вычисление расстояний от вершины v_1 в бесконтурной сети ∗)
Вход: бесконтурная сеть $G = (V, E, c)$ с топологически отсортированными вершинами, заданная списками $\overrightarrow{list}[v]$.
Выход: расстояния $D[v]$ от v_1 до всех $v \in V$, $Previous[v]$ — предпоследняя вершина в кратчайшем (v_1, v)-пути.

```
1.  begin
2.    D[v_1] := 0; Previous[v_1] := 0;
3.    for k := 2 to n do  D[v_k] := ∞;
4.      for k := 2 to n do
5.        for w ∈ list→[v_k]  do
6.          if D[w] + c(w, v_k) < D[v_k] then
7.            begin
8.              D[v_k] := D[w] + c(w, v_k);
9.              Previous[v_k] := w;
10.           end
11. end.
```

Теорема 4.4. *Алгоритм* 4.4 *имеет сложность* $O(m)$.

Доказательство. Цикл в строке 3 требует n операций присваивания. Цикл в строках 4-10 приводит к тому, что каждая дуга сети анализируется ровно один раз, и каждый анализ дуги приводит к выполнению числа операций, ограниченного константой в строках 6-9. Следовательно, сложность алгоритма 4.4 есть $O(n) + O(m) = O(m)$. □

Работа алгоритма 4.4 показана на рис. 15.



k	$D[v_1]$	$D[v_2]$	$D[v_3]$	$D[v_4]$	$D[v_5]$	$D[v_6]$
	0	∞	∞	∞	∞	∞
2		4	∞	∞	∞	∞
3			11	∞	∞	∞
4				3	∞	∞
5					8	∞
6						-3

Рис. 15

В случае задания бесконтурной сети списками $\overleftarrow{list}$ расстояния от v_1 до всех остальных вершин также могут быть вычислены за время $O(m)$. Такой алгоритм получается легкой модификацией алгоритма 4.4. Предоставляем читателям возможность самостоятельно подправить алгоритм 4.4 для достижения указанной цели.

В заключение отметим, что все три основных алгоритма (Форда-Беллмана, Дейкстры и алгоритм 4.4) вычисления расстояний от фиксированной вершины легко могут быть модифицированы для вычисления расстояний от фиксированной вершины в сетях со взвешенными вершинами.

4.5. Задача о максимальном пути и сетевые графики

Задача о максимальном пути формулируется следующим образом: в заданной сети $G = (V, E, c)$ с выделенной вершиной s для каждой вершины $v \in V$ найти (s, v)- путь, имеющий максимальную длину среди всех возможных (s, v)-путей в сети G.

Отметим, что имеет смысл решать эту задачу лишь в сетях, не содержащих контуров положительной длины. Рассмотрев тогда сеть, отличающуюся от исходной только изменением знаков весов дуг, получим сеть, в которой нет контуров отрицательной длины. Применяя к новой сети алгоритм Форда-Беллмана, можно построить кратчайшие (s, v)-пути, которые будут путями максимальной длины в исходной сети.

Впрочем, задачу о максимальном пути в общем случае можно решать и непосредственно, заменяя в алгоритме Форда-Беллмана знак неравенства в строке 9 на противоположный.

Разумеется, решая таким образом задачу о максимальном пути, вес несуществующих дуг следует положить равным $-\infty$.

Важный частный случай сети с неотрицательными весами дуг, не имеющей контуров положительной длины, — это сеть с неотрицательными весами дуг, в которой вообще нет контуров. В этом частном случае задача о максимальном пути может быть решена алгоритмом 4.4, в котором $+\infty$ заменяется на $-\infty$, а знак неравенства в строке 6 меняется на противоположный. Несложное доказательство корректности исправленного таким образом алгоритма 4.4 мы предоставляем читателю.

Отметим, что подобная замена знака неравенства в алгоритме Дейкстры не позволяет получить алгоритм решения задачи о максимальном пути. Для примера достаточно рассмотреть сеть, изображенную на рис. 16. Здесь такая модификация алгоритма Дейкстры, неправильно опре-

делит путь максимальной длины от вершины s до вершины 2.

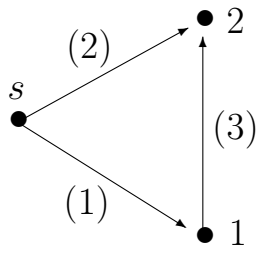


Рис. 16

Итак, алгоритм Форда-Беллмана и алгоритм 4.4 легко могут быть модифицированы для вычисления длин максимальных (s, v)-путей. Сами же пути с помощью меток $Previous[v]$ строятся также, как и в алгоритме Форда-Беллмана.

Задача о максимальном пути в бесконтурной сети имеет большое практическое значение. Она является важнейшим звеном в методах сетевого планирования работ по осуществлению проектов.

Многие крупные проекты, такие как строительство дома, изготовление станка, разработка автоматизированной системы бухгалтерского учета и т. д., можно разбить на большое количество различных операций (работ). Некоторые из этих операций могут выполняться одновременно, другие — только последовательно: одна операция после окончания другой. Например, при строительстве дома можно совместить во времени внутренние отделочные работы и работы по благоустройству территории, однако возводить стены можно только после того, как будет готов фундамент.

Задачи планирования работ по осуществлению заданного проекта состоят в определении времени возможного окончания как всего проекта в целом, так и отдельных работ, образующих проект; в определении резервов времени для выполнения отдельных работ; в определении критических работ, т. е. таких работ, задержка в выполнении которых ведет к задержке выполнения всего проекта в целом; в управлении ресурсами, если таковые имеются, и т. п.

Здесь мы разберем основные моменты одного из методов сетевого планирования, называемого *методом критического пути*. Метод был разработан в конце пятидесятых годов Дюпоном и Ремингтоном Рандом для управления работой химических заводов фирмы «Дюпон де Немур» (США).

Пусть некоторый проект W состоит из работ $v_1, \ldots, v_n$; для каждой работы v_k известно (или может быть достаточно точно оценено) время ее выполнения $tm(v_k)$. Кроме того, для каждой работы v_k известен, возможно пустой, список $Pred(v_k)$ работ, непосредственно предшествующих выполнению работы v_k. Иначе говоря, работа v_k может начать выполняться только после завершения всех работ, входящих в этот список.

Для удобства, в список работ проекта W добавим две фиктивные работы s и t, где работа s обозначает начало всего проекта W, а работа t — завершение работ по проекту W. При этом будем считать, что работа s предшествует всем тем работам $v \in W$, для которых список $Pred(v)$ пуст, иначе говоря, для всех таких работ v положим $Pred(v) = \{s\}$. Положим далее $Pred(s) = \emptyset$, $Pred(t) = \{v \in W | v$ не входит ни в один список $Pred(w)\}$, т. е. считаем, что работе t предшествуют все те работы, которые могут выполняться самыми последними. Время выполнения работ s и t естественно положить равными нулю: $tm(s) = tm(t) = 0$.

Весь проект W теперь удобно представить в виде сети $G = (V, E, c)$, где сеть $G = (V, E, c)$ определим по правилам:

1) $V = W$, т. е. множеством вершин объявим множество работ;

2) $E = \{vw | v \in Pred(w)\}$, т. е. отношение предшествования задает дуги в сети;

3) $c(v, w) = tm(w)$.

Так построенную сеть G часто называют сетевым графиком выполнения работ по проекту W. Легко видеть, что списки смежностей этой сети $\overrightarrow{list}[v]$ совпадают с заданными для проекта списками предшествующих работ $Pred(v)$.

Понятно, что сетевой график любого проекта не может содержать контуров.

Отсутствие контуров в сети G позволяет пронумеровать работы проекта W таким образом, чтобы для каждой дуги ij сети G выполнялось условие $i < j$. Поэтому в дальнейшем будем считать, что вершины в сети G топологически отсортированы.

На рис. 17 приведен пример проекта строительства дома и соответствующий сетевой график.

n	Наименование работы	Предшествующие работы	Время выполнения
1	Закладка фундамента	Нет	4
2	Возведение коробки здания	1	8
3	Монтаж электропроводки	2	2
4	Сантехмонтаж	2	3
5	Настил крыши	2	4
6	Отделочные работы	3, 4	5
7	Благоустройство территории	5	2

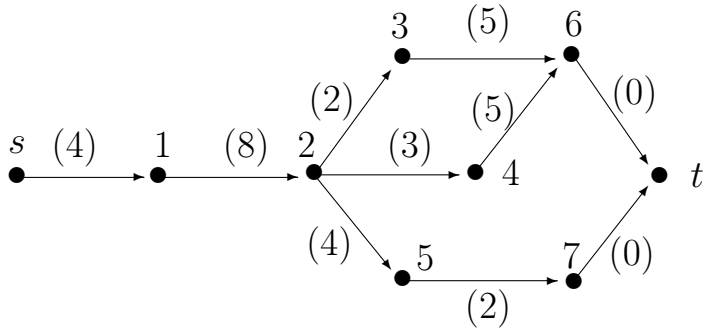


Рис. 17

Конечной целью построения сетевой модели является получение информации о возможных сроках выполнения как отдельных работ, так и о возможном сроке выполнения всего проекта в целом.

Обозначим через $efin(v)$ (соответственно $ebeg(v)$) *наиболее ранний возможный срок выполнения работы* v (соответственно *наиболее ранний возможный срок начала работы* v). Удобно считать, что $efin(s) = ebeg(s) = 0$. Поскольку начать выполнять работу v можно только после того, как будут выполнены все работы, предшествующие данной работе v, то получим следующие формулы для расчета значений $ebeg(v)$ и $efin(v)$:

$$ebeg(v) = \max(efin(w)|w \in Pred(v)),$$
$$efin(v) = ebeg(v) + tm(v).$$

Легко видеть, что значение $efin(v)$ равно длине максимального (s, v)-пути в сети G. Поэтому, для вычисления значений $efin(v)$ можно использовать алгоритм вычисления длин минимальных путей в бесконтурной сети, в котором все знаки неравенств заменены на противоположные. При этом значение $efin(t)$ дает наиболее ранний возможный

срок завершения всего проекта в целом. Ради полноты приведем здесь формальную запись алгоритма, непосредственно вычисляющего характеристики $ebeg$ и $efin$.

Алгоритм 4.5.
(∗ расчет наиболее ранних возможных сроков начала и выполнения работ ∗)
Вход: сетевой график G работ V, заданный списками $Pred(v), v \in V$.
Выход: наиболее ранние возможные сроки начала и выполнения работ $ebeg[v]$, $efin[v], v \in V$.

```
1.  begin
2.    for k := 0 to n + 1 do  ebeg[k] := efin[k] := 0;
3.    for k := 1 to n + 1 do
4.      begin
5.        for i ∈ Pred(k)  do
6.          ebeg[k] := max(efin[i], ebeg[k]);
7.        efin[k] := ebeg[k] + tm[k];
8.      end
9.  end.
```

В этом алгоритме вершины сетевого графика s и t обозначены соответственно через 0 и $n + 1$.

Значения $efin(v)$ и $ebeg(v)$ для сетевого графика, изображенного ранее на рис. 17, приведены на рис. 18. Из найденных значений следует, что этот проект не может быть завершен раньше чем через 20 единиц времени.

Пусть T — плановый срок выполнения проекта W. Ясно, что T должно удовлетворять неравенству $efin(n + 1) \leqslant T$.

Через $lfin(v)$ (соответственно $lbeg(v)$) обозначим *наиболее поздний допустимый срок выполнения (начала)* работы v, т. е. такой срок, который не увеличивает срок T реализации всего проекта. Например, для работы 3 сетевого графика из рис. 17 имеем $ebeg(3) = 12$, $efin(3) = 14$, но ясно, что начать работу 3 можно на единицу времени позже, поскольку это не повлияет на срок выполнения всего проекта, а вот задержка в реализации этой же работы на 2 единицы приведет к увеличению срока выполнения всего проекта на 1 единицу времени.

Непосредственно из определений получаем справедливость равенств $lbeg(n+1) = lfin(n+1) = T$. Поскольку произвольная работа v должна быть завершена до начала всех наиболее поздних допустимых сроков

тех работ w, которым предшествует работа v, то получаем следующие формулы:

$$lfin(v) = \min\{lbeg(w) | v \in Pred(w)\},$$

$$lbeg(v) = lfin(v) - tm(v).$$

Вычислить значения $lfin(v)$ и $lbeg(v)$ можно, двигаясь по вершинам сети G от $n+1$ к 0. Все детали вычисления названных характеристик приведены в алгоритме 4.6.

Алгоритм 4.6.
(* Расчет наиболее поздних сроков начала и окончания работ *)
Вход: сетевой график G работ V, заданный списками $Pred[v], v \in V$, плановый срок окончания проекта — T.
Выход: наиболее поздние допустимые сроки выполнения и начала работ $lfin[v]$ и $lbeg[v]$.

```
1.  begin
2.    for v ∈ V  do  lfin[v] := T;
3.    for k := n + 1 downto 1 do
4.      begin
5.        lbeg[k] := lfin[k] − tm(k);
6.        for i ∈ Pred(v)   do
7.          lfin[i] := min(lfin[i], lbeg[k]);
8.      end
9.  end.
```

Найденные значения возможных и допустимых сроков выполнения работ позволяют определить резервы времени для выполнения той или иной работы. В сетевом планировании рассматривают несколько различных и по-своему важных видов резерва работ. Мы здесь ограничимся лишь *полным резервом* (иногда его называют *суммарным*) времени выполнения работ. Он определяется по формуле

$$reserve(v) = lbeg(v) - ebeg(v).$$

Значение $reserve(v)$ равно максимальной задержке в выполнении работы v, не влияющей на плановый срок T. Понятно, что справедливо и такое равенство

$$reserve(v) = lfin(v) - efin(v).$$

Работы, имеющие нулевой резерв времени, называются *критическими*. Через каждую такую работу проходит некоторый максимальный (s,t)-путь в сети G. Поэтому такой метод нахождения критических работ и называют методом критического пути. Критические работы характеризуются тем, что любая задержка в их выполнении автоматически ведет к увеличению времени выполнения всего проекта. Численные значения введенных характеристик сетевых графиков для проекта из рис. 17 даны на рис. 18. Расчеты выполнены при $T = 20$. Критическими работами этого проекта являются работы с номерами $0, 1, 2, 4, 6, 8$, которые и образуют в сети G критический путь.

Работы	ebeg	efin	lbeg	lfin	reserve
0	0	0	0	0	0
1	0	4	0	4	0
2	4	12	4	12	0
3	12	14	13	15	1
4	12	15	12	15	0
5	12	16	14	18	2
6	15	20	15	20	0
7	16	18	18	20	2
8	20	20	20	20	0

Рис. 18

Исследование сетевых графиков на этом мы завершим. Отметим только, что помимо рассмотренных нами характеристик, часто рассматривают и большое число других, связанных, например, с неопределенностью во времени выполнения работ, с управлением ресурсами и т. д. Достаточно обширный и содержательный материал по этому поводу можно найти в книге [53].

4.6. Задача о maxmin-пути

Пусть $G = (V, E, c)$ — сеть. Для произвольного (s, v)-пути $P : s = v_0, v_1, \ldots, v_k = v$ положим

$$m(P) = \min\{c(v_{i-1}, v_i) | i = 1, 2, ..., k\}.$$

Число $m(P)$ назовем *весом* пути P.

Задача о maxmin-пути формулируется следующим образом: среди всех (s, v)-путей в сети G найти путь максимального веса.

Иногда эту задачу называют задачей об узких местах. Для иллюстрации разберем следующий пример.

Рассмотрим дорожную сеть, включающую мосты. Будем считать, что превышение грузоподъемности некоторого моста при перевозке груза приводит к разрушению данного моста. Допустим, что мы хотим определить максимальный вес груза, который может быть транспортирован в рассматриваемой дорожной сети из пункта s в пункт v без превышения грузоподъемности находящихся на пути движения транспорта мостов.

Для решения этой задачи построим сеть $G = (V, E, c)$, множеством вершин которой объявим пункты s и v, а также все перекрестки дорожной сети. Будем говорить, что дорога x, соединяющая перекрестки w и u, непосредственно их соединяет, если x не проходит через другие перекрестки. Для дороги x положим $c(x)$ равным минимальной из грузоподъемностей мостов, находящихся на дороге x; если дорога x мостов не содержит, то считаем, что $c(x) = \infty$. Будем считать, что две вершины w и u сети G соединены дугами (w, u) и (u, w), если есть хоть одна дорога x, непосредственно соединяющая перекрестки w и u.

Положим

$$c(u, w) = c(w, u) =$$

$$= \max\{c(x) | x \text{ непосредственно соединяет } w \text{ и } u\}.$$

Аналогично определяются дуги, инцидентные пунктам s и v. Будем считать, что из s дуги только исходят (т. е. исключим дуги вида (w, s), где $w \in V$), а в v дуги только входят.

На рис. 19 *a*) дан пример дорожной сети, соединяющей пункты s и v: цифрами обозначены перекрестки, буквами — мосты, а числа в скобках означают грузоподъемность соответствующего моста. Соответствующая сетевая модель приведена на рис. 19 *b*), где каждое неориентированное ребро вида i, j соответствует двум дугам (i, j) и (j, i). Обращаем внимание читателя на вес дуги $(3, v)$, который равен 80, ибо пункты 3 и v соединяют две дороги x и y, для которых $c(x) = 80$, и $c(y) = 60$. Поэтому $c(3, v) = 80$.

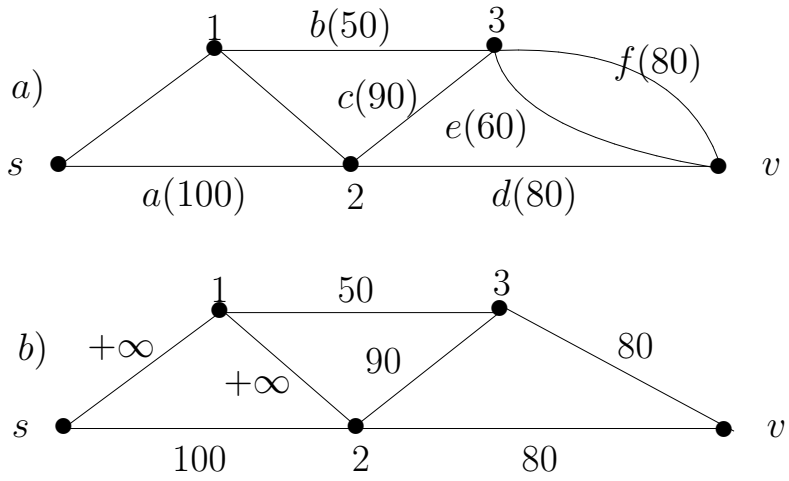


Рис. 19

Для решения задачи о maxmin-пути в произвольной сети $G = (V, E, c)$ с необязательно неотрицательными весами мы лишь немного модифицируем изложенный ранее алгоритм Дейкстры. Принцип «жадности» в вычислении maxmin-расстояний выглядит здесь следующим образом: вычисляем последовательно каждый раз maxmin-расстояние до наиболее далекой вершины от s среди всех тех вершин, до которых maxmin-расстояние еще не вычислено.

Более точно наши рассуждения здесь таковы. Обозначим через $dm(v)$ максимальный вес среди всех (s, v)-путей, т. е. maxmin-расстояние от s до v.

Положим $S = \{s\}$ и $dm(s) = +\infty$.

Будем теперь добавлять к множеству S по одной вершине так, что множество S состоит на каждом шаге из тех вершин v, для которых $dm(v)$ вычислено, и для всех $v \in S$ и $w \in F$, где $F = V \setminus S$, справедливы неравенства $dm(v) \geqslant dm(w)$. Для каждого $w \in F$ положим

$$D(w) = \max\{\min\{dm(v), c(v, w)\} | v \in S\}.$$

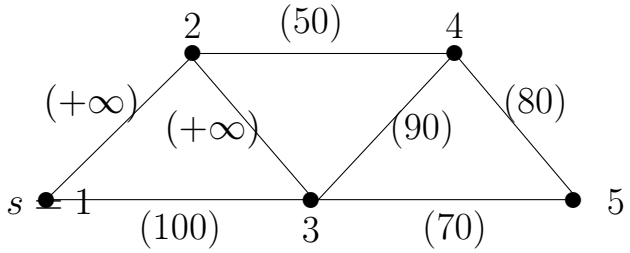
Понятно, что в этой формуле появляется максимум вместо минимума в аналогичной формуле (1) из разд. 10.2, ибо нас интересует путь максимального веса.

Очередная вершина w^*, которую следует добавить к S, выбирается в соответствии с условием

$$D(w^*) = \max\{D(w) | w \in F\}.$$

Обоснование корректности такой модификации алгоритма Дейкстры проводится аналогично доказательству корректности самого алгоритма

Дейкстры. Формально можно сказать, что вся модификация заключается в том, что все знаки суммы заменяются на минимумы, а все минимумы в алгоритме Дейкстры — на максимумы. Поэтому мы не будем приводить здесь исправленный указанным способом алгоритм 4.2.



S	D[1]	D[2]	D[3]	D[4]	D[5]
1	$+\infty$	$+\infty$	100	$-\infty$	$-\infty$
1,2			$+\infty$	50	$-\infty$
1,2,3				90	70
1,2,3,4					80
1,2,3,4,5					

Рис. 20

На рисунке 20 каждое неориентированное ребро vw представляет собой две ориентированные дуги (v, w) и (w, v). Читателя не должен смущать тот факт, что веса дуг (1,2) и (2,3) равны $+\infty$. Это обстоятельство означает, что во-первых, дуги (1,2) и (2,3) имеются в сети и, во-вторых, они имеют бесконечно большой вес. Бесконечно большой вес некоторых дуг моделирует ситуацию, встречающуюся в задаче о самом надежном пути перевозки груза по дорожной сети. Естественно считать, что дорога, по которой может быть транспортирован груз любого веса, соответствует дуге бесконечно большого веса. При рассмотрении maxmin-задачи веса несуществующих дуг естественно считать равными $-\infty$. Сами пути легко строятся с помощью меток *Previous*. Отметим только, что такая модификация алгоритма Дейкстры не гарантирует построения минимального по числу дуг наилучшего в смысле maxmin-расстояния пути. Например, в сети из рис 20 до вершины 5 будет найден путь 1-2-3-4-5, в то время как путь 1-3-4-5 имеет тот же вес. Отметим, что задача о minmax-пути может быть сформулирована и решена аналогично задаче о maxmin-пути.

4.7. Задача о кратчайших путях между всеми парами вершин

В предыдущих разделах этой главы рассматривались задачи нахождения оптимальных в том или ином смысле путей от некоторой фиксированной вершины до всех остальных вершин сети. Здесь мы рассмотрим задачу построения кратчайших путей между всеми парами вершин. Под длиной пути, как и в разделах 4.1 – 4.4, мы понимаем сумму весов дуг, образующих этот путь. Ясно, что эту задачу можно решать, используя n раз (поочередно для каждой вершины) один из описанных ранее алгоритмов нахождения расстояний от фиксированной вершины. Таким образом, мы получаем алгоритмы сложности $O(n^4)$ (при использовании алгоритма Форда-Беллмана) и $O(n^3)$ для сетей с неотрицательными весами (алгоритм Дейкстры) или для бесконтурных сетей (алгоритм 4.4). Однако, для общего случая сетей с произвольными весами имеются более эффективные алгоритмы, чем метод, основанный на многократном применении алгоритма Форда-Беллмана. Один из таких алгоритмов, предложенный в 1962 году Флойдом, мы здесь и разберем.

Пусть сеть $G = (V, E, c)$ задана матрицей весов A, где $A[i, j] = c(v_i, v_j)$, и $A[i, j] = \infty$, если дуги (v_i, v_j) в сети нет. Обозначим через $d_k(i, j)$ длину кратчайшего пути из v_i в v_j, все промежуточные вершины которого содержатся в множестве $v_1, \ldots, v_k$, т. е. содержатся в первых k вершинах. Положим

$$d_0(i, j) = A[i, j].$$

Пусть $d_k(i, j)$ вычислено при всех $i, j = 1, ..., n$ и некотором $k \geqslant 0$. Докажем равенство

$$d_{k+1}(i, j) = \min\left(d_k(i, j), d_k(i, k + 1) + d_k(k + 1, j)\right). \tag{3}$$

Действительно, рассмотрим кратчайший (v_i, v_j)-путь P с промежуточными вершинами из множества $v_1, \ldots, v_k, v_{k+1}$. Возможны две ситуации: либо вершина v_{k+1} входит в этот путь, либо нет.

Если вершина v_{k+1} не входит в путь P, то, как легко видеть, справедливо равенство $d_{k+1}(i, j) = d_k(i, j)$.

Если же вершина v_{k+1} входит в путь P, то разбивая этот путь на пути от v_i до v_{k+1} и от v_{k+1} до v_i, получаем два новых пути, все промежуточные вершины которых входят во множество $v_1, \ldots, v_k$. Поскольку всякий подпуть кратчайшего пути сам является кратчайшим путем между соответствующими вершинами, то справедливо равенство

$d_{k+1}(i,j) = d_k(i,k+1) + d_k(k+1,j))$, что завершает обоснование равенства (3).

Равенство (3) позволяет легко находить расстояния между всеми парами вершин: нужно последовательно вычислить для всех пар вершин значения $d_0(i,j), d_1(i,j), \ldots, d_n(i,j)$ и учесть, что расстояние от v_i до v_j равно $d_n(i,j)$.

Для нахождения кратчайших путей будем использовать аналог неоднократно применявшегося ранее массива $Previous$, а именно, заведем двумерный массив $Pred$ размера $[1..n, 1..n]$, считая, что $Pred[i,j]$ равен номеру вершины, являющейся предпоследней в кратчайшем (v_i, v_j)-пути (понятно, что последней вершиной в таком пути является вершина v_j). Если $k = Pred[i,j]$, то, посмотрев значение $Pred[i,k]$, получим следующую вершину в (v_j, v_i)-пути. Таким образом, двигаясь по элементам массива $Pred$, легко построить путь для любой пары вершин.

Детали изложены в приводимом ниже алгоритме, где предполагается, что $A[i,i] = 0$ и $A[i,j] = \infty$, если дуга (v_i, v_j) в сети отсутствует. Еще раз повторим, что неотрицательность весов дуг не предполагается. Однако, считается, что в сети нет контуров отрицательной длины.

Алгоритм 4.7 (Флойд).
(* Вычисление расстояний между всеми парами вершин *)
Вход: сеть $G = (V, E, c)$, заданная матрицей весов A порядка n.
Выход: расстояния $D[i,j]$ для всех пар $v_i, v_j \in V$, матрица $Pred$, в которой $Pred[i,j]$ равно номеру предпоследней вершины в кратчайшем (v_i, v_j)-пути.

```
1.  begin
2.    for i := 1 to n do
3.      for j := 1 to n do
4.        begin
5.          D[i,j] := A[i,j]; Pred[i,j] := i;
6.        end;
7.      for k := 1 to n do
8.        for i := 1 to n do
9.          for j := 1 to n do
10.           if D[i,j] > D[i,k] + D[k,j] then
11.             begin
12.               D[i,j] := D[i,k] + D[k,j];
13.               Pred[i,j] := Pred[k,j]
14.             end
```

15. **end**.

Понятно, что сложность алгоритма Флойда определяется сложностью цикла в строках 7-14, который состоит из трех вложенных циклов, выполняемых n раз каждый. Отсюда следует

Теорема 4.5. *Алгоритм Флойда имеет сложность* $O(n^3)$.

Здесь интересно отметить, что точно такую же сложность имеет алгоритм Форда-Беллмана вычисления расстояний от фиксированной вершины до всех остальных вершин сети. В настоящее время не известен ни один алгоритм вычисления расстояния между фиксированной парой вершин, который был бы существенно эффективнее (т. е. имел бы меньшую вычислительную сложность), чем алгоритм вычисления расстояний между всеми парами вершин.

Формальное описание процедуры построения самих кратчайших путей, использующее матрицу $Pred$, не составляет никаких трудностей, и мы предоставляем его читателям в качестве несложного упражнения для самостоятельной работы.

5. Задача о максимальном потоке

5.1. Основные понятия и результаты

В этой главе мы будем рассматривать сети $G = (V, E, c)$, имеющие единственную вершину s с нулевой степенью захода и единственную вершину t с нулевой степенью исхода. Вершину s будем называть *источником* , а вершину t — *стоком* сети G. Будем предполагать также, что веса $c(e)$ всех дуг неотрицательны. Число $c(e)$ будем называть *пропускной способностью* дуги e.

Для удобства введем следующие обозначения. Для произвольной вершины v через $v+$ (соответственно $v-$) будем обозначать множество вершин, к которым (из которых) идут дуги из вершины (в вершину) v.

Потоком f в сети G называется функция $f : E \to \mathbb{R}$, удовлетворяющая условиям:

1) $0 \leqslant f(e) \leqslant c(e)$ для всех $e \in E$;

2) $f(v-) = f(v+)$ для всех $v \in V$, $v \neq s, v \neq t$, где $f(v-) = \sum\limits_{w \in v-} f(w, v)$, $f(v+) = \sum\limits_{w \in v+} f(v, w)$.

Число $f(v, w)$ можно интерпретировать, например, как количество жидкости, поступающей из v в w по дуге (v, w). С этой точки зрения значение $f(v-)$ может быть интерпретировано как поток, втекающий в вершину v, а $f(v+)$ — вытекающий из v.

Условие 1) называется условием ограничения по пропускной способности, а условие 2) — условием сохранения потока в вершинах; иными словами, поток, втекающий в вершину v, отличную от s и t, равен вытекающему из нее потоку.

Положим $\|f\| = f(s+)$. Число $\|f\|$ называется *величиной потока* f. Поток f называется *максимальным*, если для любого потока f^* справедливо неравенство $\|f^*\| \leqslant \|f\|$.

Задача о максимальном потоке: в заданной сети найти поток максимальной величины.

Такая задача возникает, например, когда требуется найти максимально возможный объем некоторой жидкости или газа, который может быть перекачан по сети трубопроводов от источника до пункта потребления. При этом условие 1) в определении потока моделирует тот факт, что по каждой трубе может протекать ограниченное количество жидкости, обусловленное, например, диаметром трубы, а условие 2) — то, что потерь в промежуточных вершинах не происходит. Понятие потока в сети может моделировать также потоки транспорта в сети автострад,

перевозку товаров по железным дорогам и т. п.

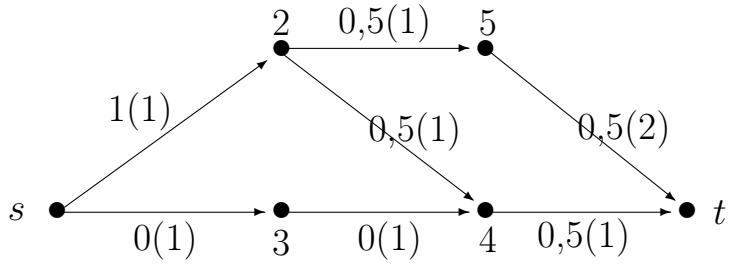


Рис. 21

На рис. 21 дан пример сети G и потока f в ней. Значение $f(e)$ на дуге e указано возле соответствующей дуги, в скобках указана пропускная способность этой же дуги. Очевидно, величина этого потока равна 1.

Задача о максимальном потоке имеет одну особенность, отличающую ее от всех рассмотренных ранее задач дискретной оптимизации. А именно, во всех предшествующих задачах искомый объект существовал очевидным образом и в принципе мог быть найден полным перебором. Например, можно было бы перебрать все пути между заданными вершинами и выбрать среди них кратчайший или перебрать все остовы и выбрать минимальный. В задаче о максимальном потоке полный перебор принципиально невозможен и существование максимального потока, вообще говоря, не очевидно. Тем не менее, справедлива следующая

Теорема 5.1. *В каждой сети существует максимальный поток.*

Доказательство. Пусть $G = (V, E, c)$ — сеть, s и t — соответственно источник и сток сети G. Занумеруем произвольным образом множество дуг E сети G. Тогда каждый поток f в сети G это просто упорядоченный набор из m чисел, т. е. точка m-мерного евклидова пространства $\mathbb{R}^m$, где m — число дуг в сети. Иначе говоря, пусть X — множество всех потоков в сети G. Тогда имеется естественное иньективное отображение ϕ множества X в $\mathbb{R}^m$.

Пусть $Y = \phi(X)$. В силу условия 1) определения потока множество Y ограничено, так как содержится в m-мерном параллелепипеде $\prod_{i=1}^{m}[0, c(e_i)]$.

Докажем замкнутость множества Y. Рассмотрим предельную точку $y = (y_1, \ldots, y_m)$ множества Y и покажем, что существует поток f в сети G такой, что $f(e_i) = y_i$ для любого $i = 1, \ldots, m$. Достаточно проверить, что для отображения, заданного равенствами $f(e_i) = y_i$ выполняются

оба условия 1) и 2) из определения потока. Пусть f_k — последовательность потоков в сети G такая, что для каждого $i = 1, \ldots, m$ последовательность $f_k(e_i)$ сходится к y_i. Поскольку неравенства $0 \leqslant f_k(e_i) \leqslant c(e_i)$ выполняются при любом k, то, переходя к пределу, получаем неравенства $0 \leqslant y_i \leqslant c(e_i)$. Итак, условие 1) выполняется. Пусть теперь v произвольная вершина сети. Обозначим через $I-$ и $I+$ множества индексов дуг, входящих (соответственно выходящих) в вершину v (из вершины v). Тогда для любого k справедливо равенство

$$\sum_{i \in I-} f_k(e_i) = \sum_{i \in I+} f_k(e_i).$$

Переходя в этом равенстве к пределу, получим

$$\sum_{i \in I-} f(e_i) = \sum_{i \in I+} f(e_i).$$

Тем самым замкнутость множества Y, а вместе с ней и компактность этого множества доказаны.

Вещественная функция $\|y\| = \sum_{i \in S} y_i (y \in Y)$, где S — множество индексов дуг, выходящих из источника s, является, как нетрудно заметить, линейной функцией. Отсюда вытекает ее непрерывность. Следовательно, по теореме Вейерштрасса данная функция имеет максимум. Поскольку эта функция дает величину потока, точка, в которой она имеет максимум, является максимальным потоком в сети G. □

Разрезом между заданными вершинами v и w в ориентированном графе обычно называют минимальное множество дуг, удаление которых из орграфа приводит к разрушению всех (v, w)-путей. Этому понятию можно придать двойственную формулировку в терминах множеств вершин. В этой главе, нам удобнее оперировать именно с такой трактовкой понятия разреза.

Более точно, (s, t)-*разрезом* (в дальнейшем просто разрезом) (V_s, V_t) в сети G называется пара множеств V_s, V_t, удовлетворяющих условиям:

(a) $s \in V_s, t \in V_t$;

(b) $V_s \cup V_t = V$;

(c) $V_s \cap V_t = \emptyset$.

Для разреза (V_s, V_t) через $E(V_s \to V_t)$ обозначим множество всех дуг e, начала которых лежат в V_s, а концы — в V_t. Аналогично,

$$E(V_t \to V_s) = \{e = vw \in E | v \in V_t, w \in V_s\}.$$

Например, для сети из рис. 21 и разреза (V_s, V_t), в котором $V_s = \{s, 4\}$, а $V_t = \{2, 3, 5, t\}$ имеем $E(V_s \to V_t) = \{(s, 2), (s, 3), (4, t)\}, E(V_t \to V_s) =$

$= \{(2,4),(3,4)\}$. Пусть

$$f(V_s \to V_t) = \sum_{e \in E(V_s \to V_t)} f(e), \quad f(V_t \to V_s) = \sum_{e \in E(V_t \to V_s)} f(e).$$

Лемма 1. *Для любого потока f и любого разреза (V_s, V_t) справедливо равенство*

$$\|f\| = f(V_s \to V_t) - f(V_t \to V_s).$$

Доказательство. Для произвольной вершины $v \in V_s$, где $v \neq s$, справедливо равенство $f(v+) - f(v-) = 0$ и $f(s+) = \|f\|$. Просуммировав эти равенства по всем вершинам $v \in V_s$, получим

$$\|f\| = \sum_{v \in V_s} f(v+) - \sum_{v \in V_s} f(v-). \tag{4}$$

Пусть $e = vw \in E$, и обе вершины v и w принадлежат V_s. Тогда значение $f(v,w)$ фигурирует в сумме $f(v+)$ как часть потока, выходящего из v, и в $f(w-)$ как часть входящего в w потока. Следовательно, в правой части равенства (4) все слагаемые вида $f(v,w)$, где $v, w \in V_s$, взаимно уничтожаются. Оставшиеся слагаемые дают требуемое равенство. □

Пусть $V_t = \{t\}$ и $V_s = V \setminus \{t\}$. Тогда $f(V_s \to V_t) = f(t-)$ и $f(V_t \to \to V_s) = 0$. Следовательно, равенство из леммы 1 запишется следующим образом

$$\|f\| = f(t-).$$

Последнее равенство выражает интуитивно понятный факт: поток, входящий в сток, в точности равен выходящему из источника потоку, так как потерь в промежуточных вершинах не происходит.

Для разреза (V_s, V_t) положим

$$c(V_s, V_t) = \sum \{c(e) | e \in E(V_s \to V_t)\}.$$

Число $c(V_s, V_t)$ называется *пропускной способностью разреза*. Из условия ограничения по пропускной способности следует, что

$$0 \leqslant f(V_s \to V_t) \leqslant c(V_s, V_t).$$

Отсюда и из леммы 1 получаем

Следствие. *Для любого потока f и любого разреза (V_s, V_t) справедливо неравенство $\|f\| \leqslant c(V_s, V_t)$.*

Разрез (V_s, V_t) называется *минимальным*, если для любого разреза (V_s^*, V_t^*) справедливо $c(V_s, V_t) \leqslant c(V_s^*, V_t^*)$.

Лемма 2. *Если для некоторого потока f^* и некоторого разреза (V_s^*, V_t^*) выполняется равенство $\|f^*\| = c(V_s^*, V_t^*)$, то поток f^* максимален, а разрез (V_s^*, V_t^*) минимален.*

Доказательство. Пусть f — максимальный поток, а (V_s, V_t) — минимальный разрез. Тогда справедлива следующая цепочка неравенств

$$\|f^*\| \leqslant \|f\| \leqslant c(V_s, V_t) \leqslant c(V_s^*, V_t^*).$$

Поскольку крайние члены в этой цепочке неравенств совпадают, она превращается в цепочку равенств, что и завершает доказательство леммы. □

Цепью из v в w в сети G называется чередующаяся последовательность попарно различных вершин и дуг $v_0 = v, e_0, v_1, \ldots, e_{k-1}, v_k = w$, в которой дуга e_r либо выходит из v_r и входит в v_{r+1}, либо наоборот выходит из v_{r+1} и входит в v_r. В первом случае, когда дуга e_r имеет вид $v_r v_{r+1}$, она называется *прямой* дугой цепи, а во втором — *обратной*. Отметим, что до этого момента понятие цепи рассматривалось только для неориентированных графов. Здесь мы вводим это понятие для ориентированных графов. Можно сказать, что цепь в орграфе — это то же самое, что цепь в неориентированном графе, если игнорировать ориентацию дуг.

Пусть P — цепь из v в w. Для каждой дуги e цепи P положим

$$h(e) = \begin{cases} c(e) - f(e), & \text{если } e \text{ — прямая дуга,} \\ f(e), & \text{если } e \text{ — обратная дуга.} \end{cases}$$

Пусть $h(P) = \min\{h(e) | e \in P\}$.

Цепь P из s в t называется *f-дополняющей*, если $h(P) > 0$. Например, (s,t)-цепь, включающая вершины $s, 3, 4, 2, 5, t$, является f-дополняющей, для потока f в сети G, изображенного на рис. 21. Причем $h(P) = 0,5$, дуга (2,4) является обратной дугой этой цепи, а остальные дуги — прямыми дугами.

Следующий результат является ключевым для построения алгоритма решения задачи о максимальном потоке в сети.

Лемма 3. *Пусть f — поток в сети G и P — f-дополняющая (s,t)-цепь. Тогда в сети G существует поток f^* такой, что $\|f^*\| = \|f\| + h(P)$.*

Доказательство. Определим в сети G функцию $f^* : E \to R$ по формуле

$$f^*(e) = \begin{cases} f(e) + h(P), & \text{если } e \in P, e \text{ — прямая дуга,} \\ f(e) - h(P), & \text{если } e \in P, e \text{ — обратная дуга,} \\ f(e), & \text{если } e \notin P. \end{cases}$$

Докажем, что функция f^* неотрицательна и удовлетворяет условиям 1) и 2) определения потока. Пусть e — прямая дуга цепи P. Используя определения f^* и $h(P)$, получим

$$0 \leqslant f(e) < f^*(e) = f(e) + h(P) \leqslant f(e) + h(e) =$$

$$= f(e) + (c(e) - f(e)) = c(e).$$

Итак, для прямых дуг цепи P имеем $0 \leqslant f^*(e) \leqslant c(e)$. Пусть e — обратная дуга цепи P. Неравенство $f^*(e) \leqslant c(e)$ очевидно. Докажем, что $f^*(e) \geqslant 0$. Действительно,

$$f^*(e) = f(e) - h(P) \geqslant f(e) - h(e) = f(e) - f(e) = 0.$$

Итак, функция f^* удовлетворяет условию 1).

Понятно, что условие 2) определения потока следует проверить лишь для вершин v, входящих в цепь P. Пусть e_1 — дуга в цепи P, по которой пришли в вершину v, e_2 — по которой ушли из v. Каждая из этих дуг может быть как прямой, так и обратной в цепи P. Следовательно, возможны четыре различных случая.

Рассмотрим случай когда обе дуги e_1 и e_2 прямые. Тогда справедливы равенства

$$f^*(v-) = f(v-) + h(P), \qquad f^*(v+) = f(v+) + h(P),$$

так как на обеих дугах e_1 и e_2 поток увеличивается на одно и то же число $h(P)$. Поскольку для потока f справедливо равенство $f(v-) = f(v+)$, получаем, что $f^*(v-) = f^*(v+)$.

В случае когда дуга e_1 — прямая, а дуга e_2 — обратная в цепи P, получаем, что обе эти дуги входят в вершину v. Следовательно, выполняется равенство $f^*(v-) = f(v-)$, ибо на дуге e_1 поток увеличится на $h(P)$, а на дуге e_2 уменьшится на $h(P)$. Поэтому величина входящего в v потока не изменится, как не меняется и величина выходящего из v потока.

Оставшиеся случаи (e_1 и e_2 — обратные дуги и e_1 — обратная, а e_2 — прямая дуга) рассматриваются аналогично.

Итак, f^* является потоком в сети G. Далее, поскольку цепь P начинается в вершине s и дуга e_2 для вершины s имеет вид sv и может быть только прямой дугой в цепи P, имеем $f^*(e_2) = f(e_2) + h(P)$. На остальных дугах, исходящих из s, поток не менялся, отсюда $\|f^*\| = f^*(s+) = = f(s+) + h(P) = \|f\| + h(P)$. $\square$

На рис. 22 показан поток f^*, полученный для потока f в сети G, изображенной на рис. 21, увеличением вдоль f-дополняющей цепи $s - 3 - 4 - 2 - 5 - t$. Величина потока f^* равна 1,5.

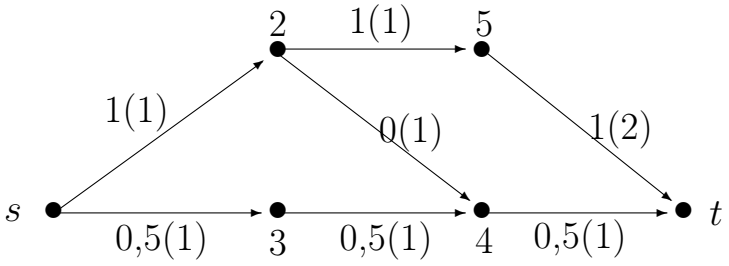


Рис. 22

Объединяет полученные выше результаты

Теорема 5.2 (Форд, Фалкерсон. 1956). *Для потока f в сети G следующие условия эквивалентны:*

a) поток f максимален;

b) не существует f-дополняющей цепи из s в t;

c) существует разрез (V_s, V_t), для которого $\|f\| = = c(V_s, V_t)$.

Доказательство. Импликации $a) \Longrightarrow b)$ и $c) \Longrightarrow a)$ уже доказаны (леммы 3 и 2 соответственно). Докажем, что $b) \Longrightarrow c)$.

Определим V_s как множество всех вершин $v \in V$, для каждой из которых существует (s, v)-цепь P такая, что $h(P) > 0$. Добавим в V_s вершину s и положим $V_t = V \setminus V_s$. Докажем, что для полученного разреза выполняется равенство $\|f\| = c(V_s, V_t)$. Пусть $e = vw$ и $e \in E(V_s \to V_t)$. Тогда $f(e) = c(e)$, так как в противном случае условие $f(e) < c(e)$ означало бы, что некоторая (s, v)-цепь P, выбранная с условием $h(P) > 0$, может быть дополнена дугой e и вершиной w до (s, w)-цепи Q, для которой $h(Q) = \min(h(P), c(e) - f(e)) > 0$. Здесь $h(e) = c(e) - f(e)$, так как e — прямая дуга в цепи Q. Поскольку $w \in V_t$, то получаем противоречие с построением множества V_s. Следовательно, выполняется равенство $f(V_s \to V_t) = c(V_s, V_t)$.

Аналогично, для всех дуг $e \in E(V_t \to V_s)$ имеем $f(e) = 0$. Следовательно, $f(V_t \to V_s) = 0$. Поскольку (лемма 1) для любого разреза

справедливо равенство $\|f\| = f(V_s \to V_t) - f(V_t \to V_s)$, для построенного разреза получаем равенство $\|f\| = c(V_s, V_t)$. $\square$

Из теоремы 5.2 и леммы 2 вытекает

Следствие. *Величина максимального потока в произвольной сети равна пропускной способности минимального разреза.*

5.2. Алгоритм Форда-Фалкерсона

Этот алгоритм построения максимального потока основан на идее, подсказываемой леммой 3. Неформально он может быть изложен следующим образом:

1) положить $f(e) = 0$ для всех дуг $e \in E$;

2) для текущего потока f искать f-дополняющую (s,t)-цепь;

3) если такая цепь P построена, то для всех прямых дуг e цепи P положить $f(e) := f(e) + h(P)$, а для всех обратных – $f(e) := f(e) - h(P)$ (в результате поток f увеличится) и вернуться на шаг 2;

4) иначе СТОП. (Поток f– максимален в силу теоремы Форда-Фалкерсона.)

Однако здесь возникает существенный вопрос. Закончится ли работа этого алгоритма за конечное число шагов? Оказывается, ответ отрицателен. Соответствующий пример такой сети привели Форд и Фалкерсон. В этой сети можно так “злоумышленно"подбирать f-дополняющие цепи, что процесс никогда не кончится; более того, величина каждого потока в течение всего времени будет меньше одной четверти величины максимального потока.

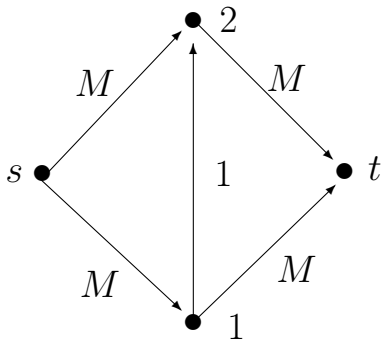


Рис. 23

Кроме того, при произвольном выборе f-дополняющей цепи количество итераций увеличения потока может не являться функцией от размерности задачи, т. е. от n и m. В этом можно убедиться на следующем простом примере (рис. 23). Здесь все дуги, кроме дуги $(1,2)$, имеют пропускную способность, равную M, где M – достаточно большое целое число, и $c(1,2) = 1$.

Действительно, пусть $f_0(e) = 0$ для всех дуг e. Выберем цепь P : $s - 1 - 2 - t$. Тогда $h(P) = 1$ и поток f_1, полученный увеличением f_0

вдоль цепи P, выглядит следующим образом

$$f_1(s,1) = f_1(1,2) = f_1(2,t) = 1, f_1(s,2) = f_1(1,t) = 0.$$

На следующем шаге выберем цепь $P : s - 2 - 1 - t$, в ней дуга $(1,2)$ является обратной. Снова $h(P) = 1$, и для нового потока f_2 получаем $f_2(1,2) = 0$ и $f(e) = 1$ для всех прочих дуг e. Далее на каждом нечетном шаге выбираем цепь $s - 1 - 2 - t$, а на каждом четном шаге – цепь $s - 2 - 1 - t$. В результате текущий поток увеличивается каждый раз ровно на единицу. Максимальный поток в этой сети равен $2M$, следовательно, понадобится $2M$ итераций увеличения потока, т. е. число итераций, а вместе с ним и сложность алгоритма, не является функцией размера задачи.

Приведенный пример показывает, что необходимо выбирать f–дополняющие цепи специальным образом. Оказывается, что если производить увеличение потока вдоль кратчайших по числу дуг f-дополняющих (s,t)-цепей, то не только можно гарантировать построение максимального потока, но и оценить сверху число итераций.

Распространим понятие f-дополняющей (s,t)-цепи и на произвольные (v,w)- цепи. Будем говорить, что некоторая (v,w)-цепь P является *f-ненасыщенной*, если $h(P) > 0$, где $h(P) = \min\{h(e) | e \in P\}$ и величина $h(e)$ для каждой дуги $e \in P$ вычисляется как прежде, а именно, $h(e) = c(e) - f(e)$ для прямых дуг и $h(e) = f(e)$ для обратных. Дуги, для которых выполняются равенства $h(P) = h(e)$, будем называть *узкими местами* цепи P.

Предположим, что алгоритм Форда-Фалкерсона строит последовательность потоков $f_0, f_1, \ldots, f_k, \ldots$. Обозначим через $d_k(v,w)$ длину (т. е. количество дуг) кратчайшей f_k-ненасыщенной (v,w)-цепи. Если такой цепи не существует, то полагаем $d_k(v,w) = \infty$.

Лемма 1. *Для всех вершин v и для всех $k \in \mathbb{N} \cup \{0\}$ выполняются неравенства*

$$d_k(s,v) \leqslant d_{k+1}(s,v), \qquad d_k(v,t) \leqslant d_{k+1}(v,t).$$

Доказательство. Докажем первое неравенство, второе доказывается аналогично. Разумеется, имеет смысл рассматривать только случай, когда существует f_{k+1}-ненасыщенная (s,v)-цепь. Пусть $P : s = v_0, e_1, v_1, \ldots, e_r, v_r = v$ – кратчайшая из таких цепей.

Докажем требуемое неравенство индукцией по длине r цепи P. Пусть $r = 1$, т. е. $d_{k+1}(s, v) = 1$. Тогда в кратчайшей (s, v)-цепи есть только одна дуга $e = sv$, которая является прямой дугой цепи P. Очевидно,

$$0 \leqslant f_k(e) \leqslant f_{k+1}(e) < c(e)$$

и, следовательно, цепь P является и f_k-ненасыщенной. Отсюда $d_k(s, v) = = 1 = d_{k+1}(s, v)$.

Пусть неравенство установлено для всех вершин w, для которых длина кратчайшей f_{k+1}-ненасыщенной (s, w)-цепи не превосходит q, и пусть $r = q + 1$ – длина кратчайшей f_{k+1}-ненасыщенной (s, v)-цепи P.

Заметим, что если для всех дуг в цепи P выполняется равенство $f_{k+1}(e) = f_k(e)$, то цепь P является также и f_k-ненасыщенной и потому $d_k(s, v) \leqslant d_{k+1}(s, v)$. Пусть теперь e_i – дуга цепи P с наибольшим номером, для которой $f_{k+1}(e_i) \neq f_k(e_i)$, т. е. на этой дуге произошло изменение потока.

По предположению индукции справедливо неравенство

$$d_k(s, v_{i-1}) \leqslant d_{k+1}(s, v_{i-1}) \tag{5}$$

Докажем, что подцепь Q цепи P от v_{i-1} до $v_r = v$ является и f_k-ненасыщенной. Пусть дуга e_i является прямой в цепи P. Тогда $f_{k+1}(e_i) < < c(e_i)$. Если дуга e_i использована при переходе от потока f_k к потоку f_{k+1} как прямая, то $f_k(e_i) \leqslant f_{k+1}(e_i) < c(e)$. Отсюда следует, что дуга e_i может быть использована как прямая в Q. Поскольку на остальных дугах цепи Q поток не менялся, то Q является и f_k-ненасыщенной. Если дуга e_i использована при переходе от потока f_k к потоку f_{k+1} как обратная, то $f_k(e_i) > 0$ и, следовательно, в Q она также может быть использована как обратная, т. е. и в этом случае Q является также и f_k-ненасыщенной.

Отсюда с учетом неравенства (5) получаем требуемый результат.

Случай, когда дуга e_i является обратной в цепи P рассматривается аналогично. □

Лемма 2. *Если все изменения потока в алгоритме Форда-Фалкерсона производятся вдоль кратчайших по числу дуг f-дополняющих (s, t)-цепей, причем $k < l$ и дуга e используется как прямая (соответственно обратная) в цепи, меняющей поток f_k на f_{k+1}, и как обратная (прямая) в цепи, меняющей поток f_l на f_{l+1}, то*

$$d_k(s, t) + 2 \leqslant d_l(s, t)$$

Доказательство. Пусть дуга e имеет вид $e = vw$. Тогда

$$d_k(s, w) = d_k(s, v) + 1,$$

поскольку e используется как прямая дуга при увеличении f_k. Ясно, что вершина w следует за v в кратчайшей f_k-дополняющей (s, t)-цепи. Далее

$$d_l(s, t) = d_l(s, w) + 1 + d_l(v, t),$$

так как e используется как обратная дуга при увеличении f_l; понятно, что вершина v следует за w в кратчайшей f_l-дополняющей (s, t)-цепи. В силу леммы 1 имеем

$$d_l(s, w) \geqslant d_k(s, w), d_l(v, t) \geqslant d_k(v, t).$$

Простые вычисления показывают, что

$$d_l(s, t) = d_l(s, w) + d_l(v, t) + 1 \geqslant d_k(s, w) + d_k(v, t) + 1 \geqslant$$

$$\geqslant d_k(s, v) + d_k(v, t) + 2 = d_k(s, t) + 2.$$

□

Теорема 5.3 (Эдмондс и Карп, 1972). *Если на каждой итерации алгоритма Форда-Фалкерсона выбирать кратчайшую по числу дуг f-дополняющую (s, t)–цепь, то построение максимального потока требует не более чем $m(n + 2)/2$ итераций.*

Доказательство. Заметим, что каждая f-дополняющая (s, t)-цепь содержит хотя бы одну дугу, являющуюся ее узким местом. Оценим, сколько раз каждая дуга может оказаться узким местом.

Пусть $e = vw$ – произвольная дуга сети и $f_{k_1}, f_{k_2}, \ldots$ $(k_1 < k_2 < \ldots)$ – последовательность потоков такая, что дуга e используется как прямая дуга при увеличении f_k и является узким местом в этот момент. Ясно, что тогда существует последовательность индексов $l_1, l_2, \ldots$ такая, что $k_1 < l_1 < k_2 < l_2 \ldots$, и дуга e используется как обратная при увеличении потока f_{l_i}.

По лемме 2 имеем

$$d_{k_i}(s, t) + 2 \leqslant d_{l_i}(s, t), \quad d_{l_i}(s, t) + 2 \leqslant d_{k_{i+1}}(s, t).$$

Отсюда $d_{k_1}(s, t) + 4(i - 1) \leqslant d_{k_i}(s, t)$ для всех i. Кроме того, $d_{k_i}(s, t) \leqslant$ $\leqslant n - 1$ и $d_{k_1}(s, t) \geqslant 1$. Поэтому $1 + 4(i - 1) \leqslant n - 1$, т. е. $i \leqslant (n + 2)/4$.

Таким образом, произвольная дуга e может быть узким местом в прямом направлении не более чем $(n+2)/4$ раз. Аналогично, она может быть узким местом в обратном направлении не более чем $(n+2)/4$ раз. Поэтому каждая дуга сети может являться узким местом не более чем $(n+2)/2$ раз. Следовательно, общее число увеличений потока, равное числу итераций, не превосходит $m(n+2)/2$. $\square$

Перейдем к формальному описанию алгоритма Форда-Фалкерсона, в котором ищутся кратчайшие по числу дуг f-дополняющие (s,t)-цепи. Метод, необходимый для построения именно кратчайшей f-дополняющей (s,t)-цепи, нами уже разработан: это поиск в ширину. Разумеется, в стандартную схему поиска в ширину необходимо внести изменения, обусловленные спецификой данной задачи.

Поскольку ищется некоторая (s,t)-цепь, то дерево поиска должно иметь в качестве корневой вершины источник s. Естественно считать, что дерево поиска содержит только те вершины сети G, через которые может проходить та или иная f-дополняющая цепь. Такие вершины будем считать помеченными. Осталось сформулировать правила помечивания. Введем массив $h[v]$. Будем считать, что $h[v] = \infty$, если вершина v не помечена. Как только вершина v становится помеченной, то $h[v] < \infty$. Более того, это будет означать, что существует f-ненасыщенная (s,v)-цепь P, для которой $0 < h(P) = h[v]$. Вершина, из которой помечена вершина v, будет обозначаться через $Previous[v]$.

При каких условиях из уже помеченной вершины w можно пометить вершину v? Это зависит от типа дуги, соединяющей w и v (в сети могут одновременно присутствовать, как дуга wv, так и дуга vw). Пусть дуга $wv \in E$. Пометить v из w можно, если выполняется условие $c(w,v) - f(w,v) > 0$. В таком случае переход от w к v совершается по прямой дуге и метка вершины v определяется по формуле

$$h[v] = \min(h[w], c(w,v) - f(w,v)).$$

Если в сети имеется дуга вида vw, т. е. обратная дуга, то помечивание возможно, если $f(v,w) > 0$. В этом случае метка вершины v определяется равенством

$$h[v] = \min(h[w], f(v,w)).$$

Если в сети имеются обе дуги vw, wv и возможно помечивание с использованием как той, так и другой дуги, то используем любую из них. Для того, чтобы различать, с помощью какой именно дуги, прямой или обратной, помечена вершина v, введем массив $choice[v]$, считая

$choice[v] = 1$, если вершина v помечена с помощью прямой дуги, и $choice[v] = -1$, если v помечена с помощью обратной дуги.

Если в процессе поиска достигнут сток t, то поиск заканчивается и по меткам $Previous$ легко строится f-дополняющая (s,t)-цепь. Если же поиск закончился, а сток t не достигнут, то f-дополняющей (s,t)-цепи не существует.

При формальном описании алгоритма будем предполагать, что сеть $G = (V, E, c)$ задана матрицей пропускных способностей, где $A[v,w] = = c(v,w)$ и $A[v,w] = 0$, если дуга vw в сети G отсутствует, а поток f – матрицей $F[v,w]$, где $F[v,w] = f(v,w)$. Просмотреть все вершины, смежные с вершиной w можно так: просмотр строки с номером w в матрице A означает просмотр всех дуг, исходящих из w, а просмотр столбца с номером w – всех дуг, входящих в w.

Опишем вначале процедуру помечивания вершин в сети G. Эта процедура является модифицированной версией процедуры поиска в ширину. В ней через Q обозначена очередь, в которую заносятся помеченные вершины.

```
1.  procedure Labeling(f);
2.  begin
3.    for v ∈ V  do  do h[v] := ∞;
4.    Q := nil; Q ⇐ s; Previous[s] := nil;
5.    while (h[t] = ∞) and (Q ≠ nil) do
6.      begin
7.        w ⇐ Q
8.        for v ∈ V   do
9.          if (h[v] := ∞) and (A[w,v] − F[w,v] > 0)
10.         then begin
11.           h[v] := min(h[w], A[w,v] − F[w,v]);
12.           Previous[v] := w; Q ⇐ v; choice[v] := 1;
13.         end;
14.         for v ∈ V \ {s}  do
15.           if (h[v] := ∞) and (F[w,v] > 0)
16.           then begin
17.             h[v] := min(h[w], F[w,v]); Q ⇐ v;
18.             father[v] := w; choice[v] := −1;
19.           end;
20.     end
21. end;
```

В этой процедуре в основном цикле 5-20 осуществляется поиск в ширину по описанным выше правилам. В цикле 8-13 осуществляется помечивание по прямым дугам, в цикле 14-19 – по обратным. Понятно, что сложность процедуры $Labeling(f)$ определяется сложностью поиска в ширину, т. е. равна $O(n^2)$. Пусть по завершению этой процедуры $h[t] < \infty$. В соответствии с правилами помечивания это означает, что существует f-дополняющая (s,t)-цепь P, для которой $h(P) = h[t]$, и, следовательно, текущий поток может быть увеличен на величину $h[t]$. Саму f-дополняющую цепь легко построить с помощью меток $Previous$. Одновременно с построением цепи можно увеличить текущий поток f. Детали приведены в алгоритме 5.1. Обращаем внимание читателя на то, что при каждом новом вызове процедуры $Labeling$ все старые метки, расставленные при предыдущем вызове этой же процедуры, исчезают.

Алгоритм 5.1 (Форд, Фалкерсон).
Вход: сеть $G = (V, E, c)$, заданная матрицей пропускных способностей $A[1..n, 1..n]$, источник s, сток t.
Выход: максимальный поток f, заданный матрицей F порядка n, для которой $F[v,w] = f(v,w)$, $\|f\|$ – величина максимального потока.

```
1.  begin
2.    for v ∈ V  do
3.      for w ∈ V  do
4.        F[v,w] := 0;
5.    ||f|| := 0;
6.    repeat
7.      Labeling(f);
8.      if h[t] < ∞ then
9.        begin
10.         ||f|| := ||f|| + h[t]; v := t;
11.         while v ≠ s do
12.           begin
13.             w := Previous[v];
14.             if choice[v] = 1
15.             then F[w,v] := F[w,v] + h[t]
16.             else F[v,w] := F[v,w] − h[t];
17.             v := w
18.           end
19.       end
20.   until h[t] = ∞;
```

21. **end**.

Теорема 5.4. *Алгоритм* 5.1 *имеет сложность* $O(n^5)$.

Доказательство. Действительно, основной цикл в строках 6-20 по теореме 5.3 проработает не более $m(n+2)/2 = n^2(n+2)/2 = O(n^3)$ раз. Каждый проход цикла 6-20 содержит вызов процедуры *Labeling*, сложность которой такая же, как и поиска в ширину в графе, заданном матрицей смежностей, т. е. $O(n^2)$. Отсюда получаем, что алгоритм 5.1 имеет сложность $O(n^5)$. □

Заметим, что если сеть $G = (V, E, c)$ задана списками смежностей $\overleftarrow{list}[v]$ и $\overrightarrow{list}[v]$, то, внося очевидные изменения в процедуру $Labeling(f)$, легко получить алгоритм сложности $O(m^2n)$, что иногда лучше, чем $O(n^5)$.

Для иллюстрации работы алгоритма 5.1 вновь рассмотрим сеть, изображенную ранее на рис. 21. Будем считать, что вершины просматриваются в порядке возрастания номеров.

После первого вызова процедуры $Labeling(f)$ будет найдена f-дополняющая цепь $s-2-4-t$, ибо вершина 2 была помечена раньше, чем вершина 3, а потому вершина 4 будет помечена из вершины 2. Ясно, что $h[t] = 1$. Следовательно, новый поток f будет таким: $f(s,2) = f(2,4) = f(4,t) = 1$ и $f(e) = 0$ для всех прочих дуг e сети G.

Второе обращение к процедуре $Labeling(f)$ определит f-дополняющую цепь $s-3-4-2-5-t, h[t] = 1$. Для этой цепи дуга $(2,4)$ является обратной. Новый поток становится таким: $f(2,4) = 0$ и $f(e) = 1$ на всех остальных дугах. Понятно, что следующий вызов процедуры $Labeling(f)$ не находит f-дополняющей (s,t)-цепи. Поток, полученный на второй итерации является максимальным.

В заключение отметим очевидное, но важное свойство алгоритма Форда-Фалкерсона.

Теорема 5.5. *Если пропускные способности всех дуг являются целыми числами, то как максимальный поток, так и все промежуточные потоки в алгоритме Форда-Фалкерсона, являются целочисленными.*

Заметим также, что в настоящее время известны алгоритмы построения максимального потока, имеющие меньшую вычислительную сложность, чем алгоритм Форда-Фалкерсона. Один из них – алгоритм Малхотры, Кумара и Махешвари сложности $O(n^3)$ подробно разобран в [36].

6. Паросочетания в двудольных графах

6.1. Основные понятия

Паросочетанием в графе называется произвольное множество его ребер такое, что каждая вершина графа инцидентна не более чем одному ребру из этого множества. Рассматривая различные задачи о паросочетаниях, мы ограничимся случаем двудольных графов. Для решения аналогичных задач в произвольных графах используются те же идеи, что и в случае двудольных, только существенно усложняется их реализация.

Напомним, что граф $G = (V, E)$ называется *двудольным*, если множество его вершин V можно разбить на непересекающиеся множества X и Y такие, что каждое ребро $e \in E$ имеет вид $e = xy$, где $x \in X$ и $y \in Y$. Двудольный граф будем обозначать либо (X, E, Y), если граф не является взвешенным, либо (X, E, c, Y), если ребрам $e \in E$ приписаны веса $c(e)$. Всюду в дальнейшем, говоря о двудольном графе, мы предполагаем, что разбиение множества V на подмножества X и Y зафиксировано.

Самые разные практические задачи связаны с построением тех или иных паросочетаний в двудольных графах. Разберем несколько примеров.

1. Пусть имеется n рабочих, каждый из которых может выполнить один или несколько из m видов работ. При этом каждый из видов работ должен быть выполнен одним рабочим. Требуется так распределить работы среди рабочих, чтобы наибольшее количество работ было выполнено.

2. Пусть в предыдущей задаче число рабочих n равно числу работ m. Спрашивается, можно ли так распределить работы между рабочими, чтобы были выполнены все виды работ?

3. Пусть сверх условий второй задачи для каждой пары рабочий-работа известна стоимость $c(x, y)$ выполнения рабочим x работы y. Требуется так подобрать каждому рабочему определенный вид работы, чтобы суммарная стоимость выполнения всех работ была минимальна.

Математическая модель всех приведенных выше задач строится очевидным образом. Определим двудольный граф $G = (X, E, Y)$, в котором в качестве X выберем множество рабочих, а в качестве Y — множество работ. Множество ребер E этого графа определим как множество всех пар (x, y) таких, что рабочий x может выполнить работу y. Пусть

$M \subseteq E$ — паросочетание в построенном графе G. Тогда каждое ребро $e = xy \in M$ можно интерпретировать как назначение рабочему x работы y. Действительно, по определению паросочетания никакие два ребра из M не могут иметь общих вершин, следовательно, на каждую работу назначается не более одного рабочего, и каждый рабочий получает не более одной работы.

В этой модели первая из рассмотренных задач означает, что в графе требуется найти паросочетание с наибольшим количеством ребер. Вторая — выяснить, существует или нет паросочетание, состоящее из n ребер. И, наконец, в третьей задаче требуется найти паросочетание из n ребер с минимальным суммарным весом его ребер.

В этой главе будут рассмотрены все три типа приведенных здесь задач. Введем необходимую терминологию. Пусть M — паросочетание в графе $G = (X, E, Y)$. Говорят, что M *сочетает* x с y и y с x, если $xy \in M$. Вершины, не принадлежащие ни одному ребру паросочетания, называются *свободными относительно* M или просто *свободными*, а все прочие — *насыщенными* в M или просто *насыщенными*. Таким образом, для каждой насыщенной в M вершины x существует y такое, что M сочетает x с y. Удобно также ребра, входящие в паросочетание M называть *M-темными* или просто *темными* ребрами, а все прочие — *M-светлыми* или *светлыми* ребрами.

Паросочетание, содержащее наибольшее число ребер, называется *наибольшим*. Паросочетание, не содержащееся ни в каком другом паросочетании, называется *максимальным*. Иначе говоря, максимальным называется паросочетание, максимальное по включению. Необходимо различать эти два понятия. Понятно, что каждое наибольшее паросочетание является максимальным, но обратное неверно. Предлагаем читателю самому построить соответствующий пример.

6.2. Задача о наибольшем паросочетании. Алгоритм Хопкрофта-Карпа

Задача о наибольшем паросочетании состоит в следующем: в заданном двудольном графе найти наибольшее паросочетание.

Оказывается, что эту задачу можно свести к задаче построения максимального потока в некоторой сети.

Пусть $G = (X, E, Y)$ — произвольный двудольный граф и $s, t \notin X \cup Y$. Построим сеть $G^* = (V^*, E^*, c)$ с источником s и стоком t. В качестве множества вершин сети G^* возьмем множество $V^* = X \cup Y \cup \{s\} \cup \{t\}$. А

множество дуг E^* определим следующим образом. Каждое ребро $e = xy$ из E, где $x \in X$ и $y \in Y$, превращаем в дугу xy, исходящую из x и входящую в y. Добавим к полученному множеству все дуги вида sx, yt, где $x \in X$ и $y \in Y$. Полученное в результате множество и есть множество дуг E^* сети G^*. Пропускную способность каждой дуги положим равной единице. На рис. 24 показаны граф G и соответствующая ему сеть G^*.

Заметим, что если f — целочисленный поток в сети G^*, то $f(e) = 0$ или $f(e) = 1$ для любой дуги $e \in E$. Кроме того, по теореме 5.5 среди таких 0-1-потоков существует максимальный поток. Оказывается, что каждому паросочетанию в графе G однозначно соответствует некоторый 0-1-поток в сети G^*. Пусть $\mathcal{P}$ — множество всех паросочетаний в графе G, а $\mathcal{F}$ — множество всех 0-1-потоков в сети G^*.

Теорема 6.1. *Существует взаимно-однозначное отображение ϕ множества $\mathcal{P}$ на множество $\mathcal{F}$, причем $|M| = \|\phi(M)\|$ для любого паросочетания M.*

(Напомним, что через $\|\phi(M)\|$ обозначается величина потока $\phi(M)$).

Доказательство. Для произвольного паросочетания M в графе G определим поток $f_M = \phi(M)$ в сети G^* по формулам $f_M(s, x) = f_M(x, y) = f_M(y, t) = 1$ для любого ребра $xy \in M$ и $f_M(e) = 0$ для остальных дуг $e \in E^*$ (соответствующий пример приведен на рис. 24). Докажем, что f_M — поток в сети G^*.

Поскольку $0 \leqslant f_M(e) \leqslant 1$, условие ограничения по пропускной способности дуг выполняется. Остается проверить условие сохранения потока в вершинах.

Пусть x — насыщенная вершина из X в паросочетании M. Тогда $f_M(s, x) = 1$ и, следовательно, $f_M(x-) = 1$, т. е. поток втекающий в вершину x равен 1. Поскольку вершина x насыщена в M, то существует ровно одно ребро $xy \in M$, инцидентное x. Для этого ребра $f_M(x, y) = 1$ на соответствующей дуге xy. Все остальные ребра графа, инцидентные вершине x , являются светлыми, т. е. не входят в паросочетание M. Поэтому $f_M(e) = 0$ для всех соответствующих дуг. Следовательно, $f_M(x+) = 1$. Тем самым равенство $f_M(x-) = f_M(x+)$ выполняется для насыщенных вершин. Если же x не насыщена, т. е. x — свободная вершина, то как входящий в x, так и выходящий из x потоки равны нулю. Следовательно, условие сохранения потока в вершинах, лежащих в X, выполняется. Для вершин $y \in Y$ это условие проверяется аналогично.

Далее, поскольку количество насыщенных в M вершин $x \in X$ в точности равно $|M|$, то $f_M(s+) = |M|$, т. е. $\|\phi(M)\| = |M|$. Легко видеть,

что разным паросочетаниям соответствуют разные потоки. Следовательно, отображение $\phi : \mathcal{P} \to \mathcal{F}$, определенное равенством $\phi(M) = f_M$, инъективно.

Обратно, пусть f — 0-1-поток в сети G^*. Положим $M_f = \{xy | f(x,y) = 1,\ x \in X,\ y \in Y\}$. Поскольку в каждую вершину $x \in X$ входит ровно одна дуга (это дуга вида sx), то имеется не более одной дуги вида xy, для которой $f(x,y) = 1$. Следовательно, каждая вершина $x \in X$ инцидентна не более чем одному ребру из M_f. Аналогично, каждая вершина $y \in Y$ инцидентна не более чем одному ребру из M_f. Отсюда следует, что M_f является паросочетанием в графе G. Легко видеть, что для паросочетания M_f справедливы равенства $|M_f| = \|f\|$ и $\phi(M_f) = f$, что завершает доказательство теоремы. □

На рис. 24 изображены двудольный граф G, сеть G^* паросочетание $M = \{x_1y_2, x_2y_3\}$ и соответствующий ему поток f_M.

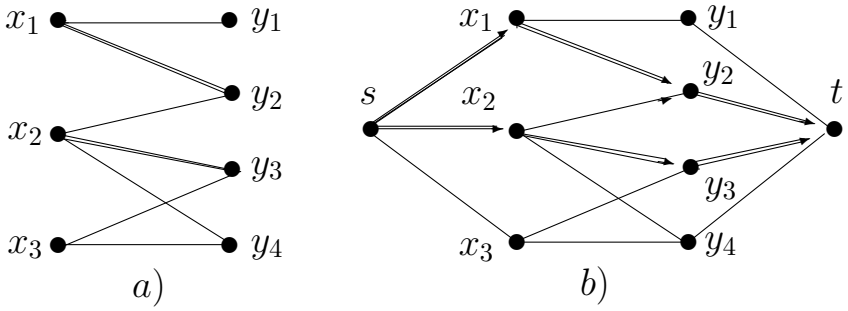


Рис. 24

Пусть f — произвольный 0-1-поток в сети G^* и P — f-дополняющая (s,t)-цепь. Тогда цепь P содержит нечетное количество дуг. Удалим из этой цепи первую дугу, а она обязательно имеет вид sx, где $x \in X$, и последнюю дугу вида yt, где $y \in Y$. Оставшиеся в цепи P дуги чередуются следующим образом: первой идет дуга вида xy, для которой $f(x,y) = 0$, поскольку эта дуга прямая в цепи, второй — дуга вида x_1y, причем эта дуга является обратной в цепи P и потому $f(x_1,y) = 1$; затем снова прямая дуга, потом обратная и т. д. Увеличение потока вдоль этой цепи по правилам, указанным в лемме 3 из разд. 5.1, приводит к тому, что новый поток становится равным единице на всех нечетных (т. е. прямых) дугах цепи P, и равным нулю на всех четных (т. е. обратных) дугах цепи P. Величина потока при этом возрастает на единицу. Понятие f-дополняющей цепи для потока f в сети G^* естественным образом соответствует понятию M-чередующейся цепи для паросочетания M в графе G.

Пусть M — паросочетание в графе G. *M-чередующейся* цепью называется такая последовательность вершин и ребер вида $x_0, x_0y_1, y_1, y_1x_2, x_2, \ldots, x_k, x_ky_{k+1}, y_{k+1}$, где $k > 0$, что все вершины этой цепи различны, x_0 и y_{k+1} — свободные, а все остальные вершины насыщенные в паросочетании M, причем каждое второе ребро принадлежит M (т. е. ребра вида $y_ix_{i+1}, i = 1, \ldots, k-1$ входят в M), а остальные ребра в M не входят. Иначе говоря, в M-чередующейся цепи цвета ребер чередуются по правилу светлое — темное и наоборот, причем первое и последнее ребра являются светлыми. Ясно, что чередующаяся цепь однозначно определяется как последовательностью ее вершин, так и последовательностью ее ребер. Например, для паросочетания M, изображенного на рис. 24 *а*), M-чередующуюся цепь можно задать последовательностью вершин $x_3, y_3, x_2, y_2, x_1, y_1$. Эта цепь содержит два темных ребра и три светлых. Соответствующая f-дополняющая цепь в сети G^*, где $f = f_M$, задается последовательностью вершин $s, x_3, y_3, x_2, y_2, x_1, y_1, t$. В этой цепи дуги x_2y_3 и x_1y_2 являются обратными, а остальные дуги — прямыми.

Увеличению потока вдоль f-дополняющей цепи соответствует увеличение количества ребер в паросочетании M вдоль M-чередующейся цепи. Для этого в M-чередующейся цепи нечетные ребра, не входившие в M, объявляются элементами M, а все четные, входившие в M, из M удаляются. Иначе говоря, все темные ребра становятся светлыми, а все светлые — темными. Такая операция приводит к увеличению количества ребер в паросочетании на единицу. Например, паросочетание M из рис. 24 *а*) заменится на паросочетание $\{x_3y_3, x_2y_2, x_1y_1\}$.

Напомним, что симметрическая разность двух множеств M и P определяется следующим образом:

$$M \oplus P = (M \setminus P) \cup (P \setminus M).$$

Процесс получения нового паросочетания M_1 из паросочетания M с помощью M-чередующейся цепи P можно выразить равенством

$$M_1 = M \oplus P$$

(здесь и далее под M-чередующейся цепью понимается последовательность ребер). Для паросочетания M_1 справедливо равенство $|M_1| = |M| + 1$, поскольку в цепи P светлых ребер на одно больше чем темных.

Из приведенных рассуждений и теоремы 5.2 вытекает следующий классический результат.

Теорема 6.2 (Берж, 1957). *Паросочетание M в двудольном графе G является наибольшим тогда и только тогда, когда в G не существует M-чередующейся цепи.*

Ради полноты изложения приведем здесь прямое доказательство этой теоремы, не опирающееся на теорему Форда-Фалкерсона.

Доказательство. Очевидно, если M — наибольшее паросочетание, то в графе $G = (X, E, Y)$ не существует M-чередующейся цепи.

Докажем обратное утверждение. Предположим, что для паросочетания M не существует M-чередующейся цепи. Рассмотрим произвольное паросочетание N и убедимся, что $|N| \leqslant |M|$. Заметим, что в графе $G_1 = (X, N \cup M, Y)$ степень каждой вершины не превосходит двух. Следовательно, каждая компонента связности графа G_1 может быть одного из следующих типов:

1) изолированная вершина;

2) цепь четной длины;

3) цепь нечетной длины;

4) цикл.

В случаях 1, 2 и 4 компонента связности содержит одинаковое число ребер из M и N. Если компонента связности — цепь нечетной длины (случай 3), то она является либо M-чередующейся, либо N-чередующейся цепью. По предположению, M-чередующихся цепей в графе нет, следовательно, могут быть только N-чередующиеся цепи. Но в каждой из этих цепей ребер из M на одно больше, чем ребер из N. Отсюда $|N| \leqslant |M|$, т. е. M — наибольшее паросочетание. □

Теорема Бержа подсказывает следующий алгоритм построения наибольшего паросочетания:

1) пустое паросочетание M объявить текущим паросочетанием;

2) искать M-чередующуюся цепь;

3) если такая цепь P найдена, то положить $M = M \oplus P$ и вернуться на шаг 2;

4) иначе СТОП (текущее паросочетание M является наибольшим).

Внимательный читатель, конечно, заметил, что предложенный алгоритм, по сути дела, является легкой модификацией алгоритма Форда-Фалкерсона. Отметим также, что поскольку каждый раз текущее паросочетание увеличивается ровно на единицу, то алгоритм завершит работу после не более чем n итераций, где $n = \min(|X|, |Y|)$ (здесь использован тот факт, что наибольшее паросочетание содержит не более чем n ребер).

Разберем подробнее процесс поиска M-чередующейся цепи. Здесь можно использовать любую схему поиска в ширину или в глубину. Чуть удобнее поиск в ширину. Сформулируем правила поиска в ширину в виде "волнового" алгоритма.

В нулевой фронт распространения волны включаем все M-свободные вершины $x \in X$.

Пусть фронт с номером k построен. Если k четно, то во фронт $k+1$ включаем все вершины $y \in Y$, не содержащиеся ни в каком из предыдущих фронтов, которые можно пометить из вершин предыдущего фронта с помощью светлых ребер. Если k нечетно, то во фронт $k+1$ включаем вершины $x \in X$, не содержащиеся ни в каком из предыдущих фронтов, которые можно пометить с помощью темных ребер из вершин предыдущего фронта.

Поиск завершается, как только будет помечена свободная вершина $y \in Y$ или очередной фронт окажется пустым. В первом случае окончания поиска M-чередующаяся цепь существует, она легко может быть построена с помощью стандартных меток $Previous$. Во втором случае M-чередующейся цепи в графе не существует и, следовательно, текущее паросочетание M является наибольшим.

Теорема 6.3. *Модифицированный алгоритм Форда-Фалкерсона построения наибольшего паросочетания в двудольном графе* $G = (X, E, Y)$ *имеет сложность* $O(pqn)$, *где* $p = |X|,\ q = |Y|,\ n = \min(p, q)$ *и граф* G *представлен матрицей* $A[1..p, 1..q]$.

(Заметим, что матрица A является подматрицей матрицы смежности двудольного графа G.)

Доказательство. Выше уже отмечалось, что процесс завершится не более чем после n итераций. Сложность каждой итерации есть $O(pq)$. Поэтому модифицированный алгоритм Форда-Фалкерсона для построения наибольшего паросочетания имеет сложность $O(pqn)$. □

Отметим, что в частном случае, когда $n = |X| = |Y|$, модифицированный алгоритм Форда-Фалкерсона имеет сложность $O(n^3)$.

В 1973 году Хопкрофт и Карп предложили более эффективный алгоритм построения наибольшего паросочетания в двудольном графе.

Пусть M — паросочетание в графе $G = (X, E, Y)$. Цепь P назовем *M-цепью*, если она начинается в свободной вершине $x \in X$, имеет нечетную длину и цвета ребер чередуются по правилу светлое — темное и наоборот. Иначе говоря, M-цепь — это почти M-чередующаяся

цепь; отличие состоит лишь в том, что M-цепь может заканчиваться в M-насыщенной вершине $y \in Y$. Понятно, что каждая M-чередующаяся цепь является M-цепью, но обратное утверждение неверно.

Пусть r — длина кратчайшей M-чередующейся цепи. Через $G(M)$ обозначим граф кратчайших M-цепей; по определению этот граф состоит из всех вершин и ребер таких, что каждое ребро и каждая вершина входят в некоторую M-цепь длины r.

Общую схему алгоритма Хопкрофта и Карпа построения наибольшего паросочетания можно описать следующим образом:

1) начать с произвольного паросочетания M в графе G;

2) построить граф $G(M)$ кратчайших M-цепей;

3) построить максимальное по включению множество $\{P_1, \ldots, P_k\}$ вершинно-непересекающихся M-чередующихся цепей в $G(\mathrm{M})$. Увеличить паросочетание M вдоль всех цепей из построенного множества по формулам

$$M_1 = M \oplus P_1, M_2 = M_1 \oplus P_2, \ldots, M_k = M_{k-1} \oplus P_k,$$

Объявить паросочетание M_k текущим, т. е. $M := M_k$. (Заметим, что $M_k = M \oplus P_1 \oplus \ldots \oplus P_k$ и $|M_k| = |M| + k$).

4) повторять шаги 2 и 3 до тех пор, пока в сети G существует хотя бы одна M-чередующаяся цепь. Если такой цепи не существует, то текущее паросочетание M является наибольшим (теорема 6.2).

Перейдем к формальному описанию алгоритма Хопкрофта-Карпа.

Двудольный граф $G = (X, E, Y)$ будем задавать матрицей A размера pq, где $p = |X|,\ q = |Y|$, в которой $A[x, y] = 1$, если ребро xy имеется в графе, и $A[x, y] = 0$, если такого ребра в графе нет.

Паросочетание M в графе G можно описать с помощью двух массивов $Xdouble$ длины p и $Ydouble$ длины q, считая, что $Xdouble[x] = y$, если x сочетается с y, и $Xdouble[x] = nil$, если вершина x свободна. Аналогично определяется массив $Ydouble$. Таким образом, пустое паросочетание задается равенствами $Xdouble[x] = Ydouble[y] = nil$ для всех $x \in X$ и $y \in Y$.

Вспомогательный граф $G(M)$ кратчайших M-цепей несложно построить, используя поиск в ширину. Напомним, что каждая M-цепь начинается в свободной вершине $x \in X$, поэтому во вспомогательный граф $G(M)$ следует отнести все свободные вершины $x \in X$ и начать поиск с них. Поскольку нас интересуют чередующиеся цепи, нужно различать шаги от X к Y и от Y к X.

В первом случае переход осуществляется по светлым ребрам, (т. е. по ребрам, не входящим в M), а во втором случае по темным ребрам (т. е. по ребрам из M). Причем, если в первом случае, находясь в вершине $x \in X$, следует отнести в $G(M)$ все вершины, смежные с x, и все ребра, инцидентные x, то во втором — выбор вершины и ребра однозначен: из вершины $y \in Y$ можно шагнуть только в вершину $x = Ydouble[y]$, используя ребро xy, которое входит в M. При этом вершину x и ребро xy следует добавить к $G(M)$. Процесс поиска завершается либо полным построением того фронта, в котором в первый раз встретится свободная вершина $y \in Y$, либо тогда, когда в граф $G(M)$ нельзя отнести ни одной новой вершины и ни одного нового ребра, но свободных вершин $y \in Y$ достичь не удалось. Последний случай означает, что M-чередующихся цепей в графе G не существует.

Через $DX \cup DY$, где $DX \subseteq X$ и $DY \subseteq Y$, будем обозначать множество вершин графа $G(M)$. Множество ребер этого графа описывается матрицей DA размера pq, где $p = |X|, q = |Y|$. При этом лишние строки и столбцы матрицы DA, соответствующие элементам $x \in X$ и $y \in Y$, не попавшим в DX и DY, будут игнорироваться.

Построение вспомогательного графа $G(M)$ представлено процедурой $Graph$(M). В ней используются две очереди. В очереди Q_1 хранится последний построенный фронт, а в очереди Q_2 накапливаются вершины нового, строящегося фронта. При этом в очередной фронт распространения волны относятся лишь вершины из множества X, ибо переход от Y к X однозначен, т. е. за один шаг строим сразу два фронта. Все достигнутые свободные вершины $y \in Y$ заносятся в $Yfree$. Через $Xfree$ обозначается множество всех свободных вершин $x \in X$. Через $front[v]$, $v \in X \cup Y$, обозначается номер фронта, в который попадает вершина v, при этом для всех непомещенных еще в какой-либо фронт вершин (в традиционной терминологии непомеченных) выполняется равенство $front[v] = \infty$.

```
1.  procedure Graph(M)
2.  begin
3.    DX := DY := ∅;
4.    for x ∈ X  do
5.      for y ∈ Y  do  DA[x, y] := 0;
6.    for v ∈ X ∪ Y  do  front[v] := ∞;
7.    Q1 := Q2 := Xfree := Yfree := nil;
8.    for x ∈ X  do
9.      if Xdouble[x] = nil  then
```

```
10.   begin
11.     Q2 ⇐ x; Xfree ⇐ x;
12.     DX := DX ∪ {x}; front[x] := 0;
13.   end;
14.     repeat
15.       Q1 := Q2; Q2 := nil;
16.       while Q1 ≠ nil do
17.         begin
18.           x ⇐ Q1;
19.           for y ∈ Y  do
20.             if (A[x, y] = 1) and (front[x] < front[y])
21.             then
22.               begin
23.                 DA[x, y] := 1;
24.                 if front[y] = ∞ then
25.                   begin
26.                     DY:=DY ∪ {y}; z := Ydouble[y];
27.                     front[y] := front[x] + 1;
28.                     if z ≠ nil then
29.                       begin
30.                         DA[z, y] := 1; DX := DX ∪ z;
31.                         front[z] := front[y] + 1; Q2 ⇐ z;
32.                       end
33.                     else Yfree ⇐ y;
34.                     end
35.                   end
36.             end
37.         until (Yfree ≠ nil) or (Q2 = nil);
38.   end.
```

Прокомментируем работу процедуры $Graph(M)$. В строках 3-5 инициализируется пустой граф $G(M)$. Цикл 8-12 означает, что все свободные вершины $x \in X$ включаются в граф G(M), и поиск в ширину начинается с них. В строке 15 инициализируется последний построенный фронт. В строке 16 начинается основной цикл поиска в ширину. При этом последний построенный фронт используется полностью. В цикле 19-35 анализируются все $y \in Y$, смежные с очередной вершиной $x \in X$ и удовлетворяющие условию $front[x] < front[y]$. Здесь возможны лишь два случая.

В первом случае $front[y] = \infty$. Это означает, что вершина y ранее не посещалась, второй — $front[y] = front[x] + 1$, т. е. вершина y уже помечена, но из вершины того же последнего построенного фронта. В этом случае в граф G(M) нужно лишь добавить одно ребро xy (строка 23). Выполнение условия в строке 28 означает, что вершина y насыщена в паросочетании M. В этом случае вершина $z = Ydouble[y]$ и ребро zy включаются в граф $G(M)$, причем z включается и в новый фронт (строка 31). В противном случае вершина y является свободной и заносится в $Yfree$. В строке 37 даны условия прекращения поиска. Случай $Yfree \neq \emptyset$ свидетельствует, что найдена хотя бы одна свободная вершина y и, следовательно, в исходном графе существует M-чередующаяся цепь. Поиск на этом прекращается и в граф $G(M)$, благодаря свойствам поиска в ширину, попадут все кратчайшие M-цепи.

Разберем теперь метод построения максимального по включению множества вершинно-непересекающихся M-чередующихся цепей и увеличения текущего паросочетания M с помощью построенного множества. Сделать это можно следующим образом. Выберем произвольную вершину $x \in Xfree$. Проведем поиск в глубину с корнем в x, помечая вершины по тем же правилам, которые использовались при построении графа $G(M)$, и продвигаясь из вершины $x \in X$ по светлым ребрам, а из вершин $y \in Y$ — по темным. Помеченные в ходе поиска вершины помещаются в стек S. Поиск в глубину из вершины x завершается либо по достижению свободной вершины $y \in Yfree$ (в этом случае существует искомая цепь, начинающаяся в x и заканчивающаяся в y), либо тогда, когда $S = nil$. Пустота стека S означает, что в $G(M)$ не существует M-чередующейся цепи, начинающейся в вершине x. Если искомая цепь существует, то после завершения поиска в S первая и последняя вершины свободны относительно M, а все промежуточные вершины насыщены в M. Отметим, что вершины чередуются — сначала вершина из DX, затем из DY и т. д.

Увеличить паросочетание M с помощью найденной цепи очень просто: считываем попарно вершины из стека S (первой считается $y \in Y$) и корректируем значения массивов $Xdouble$ и $Ydouble$. При этом у первой и последней из считанных вершин y и x значения $Ydouble[y]$ и $Xdouble[x]$, равные nil, получат значения соответствующих соседних вершин из S, а у всех прочих произойдет смена значений массивов $Ydouble$ и $Xdouble$ с одних вершин на другие. Считывая вершины из S, мы одновременно удаляем их из графа $G(M)$. В результате при построении следующей M-чередующейся цепи вновь найденная цепь не

пересекается с прежде построенными цепями по вершинам. Процесс построения цепей ведется до полного исчерпания одного из списков $Xfree$ или $Yfree$, чем обеспечивается максимальность по включению построенного множества M-чередующихся цепей.

Детали описанного процесса представлены в процедуре $Increase(M)$. В ней без формального описания используется функция $Choice(x)$, которая возвращает непомеченную в ходе поиска вершину $y \in DY$, смежную с x, если такая существует, и $Choice(x) = nil$, если все вершины из DY, смежные с x, уже помечены. Переменная $indication$ сигнализирует о том, достигнута ли свободная вершина $y \in DY$, т. е. $indication = 0$, если свободная вершина еще не достигнута, и $indication = 1$, если достигнута.

```
1.  procedure Increase(M);
2.  begin
3.    while (Xfree ≠ ∅) and (Yfree ≠ ∅) do
4.      begin
5.        x ⇐ Xfree; Xfree := Xfree \ {x};
6.        S := nil; S ⇐ x; indication := 0;
7.        while (S ≠ nil) and (indication = 0) do
8.          begin
9.            x ⇐ S; y := Choice(x);
10.           if y ≠ nil then
11.             begin
12.               S ⇐ y; z := Ydouble[y];
13.               if z ≠ nil  then S ⇐ z;
14.             else
15.               begin
16.                 indication := 1; Yfree := Yfree \ {y};
17.               end
18.             end
19.           else
20.             begin
21.               x ⇐ S; DX := DX \ {x};
22.               if S ≠ ∅ then
23.                 begin
24.                   y ⇐ S; DY := DY \ {y};
25.                 end
26.               end
27.             end;
```

```
28.          if indication = 1 then
29.            while S ≠ nil  do
30.              begin
31.                y ⇐ S; DY := DY \ {y};
32.                x ⇐ S; DX := DX \ {x};
33.                Xdouble[x] := y; Ydouble[y] := x;
34.              end
35.        end
36.    end;
```

Структура этой процедуры следующая. Основной цикл 3-35 ведется до полного опустошения одного из списков свободных в X или в Y вершин. Каждый проход этого цикла начинается с выбора произвольного элемента x из $Xfree$ и последующего поиска в глубину с корнем в x (строки 6-27). Важно отметить, что если текущая вершина x имеет непомеченную насыщенную смежную с ней вершину y, то в стек S помещаются сразу две вершины: сначала y, затем $z = Ydouble[y]$ (строки 9-13). Можно сказать, что промежуточные вершины в ходе поиска помещаются в S парами.

Именно этим обстоятельством объясняются действия в строках 20-25. Если вершины помещали в стек S парами, то и удалять их оттуда надо парами (кроме самой первой). В строках 20-25 сначала удаляется из S и из графа вершина x, все соседи которой уже посещались, а затем, если x не первой попала в S, удаляется та вершина y, для которой выполняется равенство $Ydouble[y] = x$ (впрочем, также справедливо и соотношение $y = Xdouble$[x]). В строке 28 анализируется, каким именно исходом завершился поиск в глубину.

Если поиск удачный, т. е. $indication = 1$, то в цикле 29-34 увеличивается паросочетание M и удаляются из графа $G(M)$ все вершины найденной M-чередующейся цепи.

Соберем описанные процедуры в один алгоритм построения наибольшего паросочетания в двудольном графе.

Алгоритм 6.1 (Хопкрофт, Карп).
Вход: двудольный граф $G = (X, E, Y)$, заданный матрицей $A[1..p, 1..q]$, где $p = |X|, q = |Y|$.
Выход: наибольшее паросочетание в графе G, задаваемое массивами $Xdouble[1..p]$ и $Ydouble[1..q]$.

```
1.  begin
```

2. **for** $x \in X$ **do** $Xdouble[x] := nil$;
3. **for** $y \in X$ **do** $Ydouble[y] := nil$;
4. **repeat**
5. $Graph(M)$;
6. **if** $Yfree \neq \emptyset$ **then** $Increase(M)$;
7. **until** $Yfree = \emptyset$;
8. **end**.

Дадим небольшой комментарий к этому алгоритму. В строках 2-3 строится пустое паросочетание M. После построения вспомогательного графа процедурой $Graph(M)$ анализируется (условие в строке 6), содержит ли граф $G(M)$ свободные в M вершины $y \in Y$. Если таковые имеются, то процедура $Increase(M)$ увеличивает текущее паросочетание и (условие в строке 7) фаза повторяется. Если же список достигнутых процедурой $Graph(M)$ свободных вершин $y \in Y$ пуст, то алгоритм завершает работу и по теореме Бержа текущее паросочетание является наибольшим.

Разберем работу алгоритма 6.1 на простом примере. На рис. 25 *a*) изображен исходный граф G, а на рис. 25 *b*), *c*), *d*) — паросочетания, полученные после каждой из трех фаз. Напомним, что фазой мы называем процесс построения вспомогательного графа и последующее увеличение текущего паросочетания; иначе говоря, фаза — это один проход основного цикла 4-7 в алгоритме 6.1.

Ясно, что вспомогательный граф $G(M)$, построенный в первой фазе, совпадает с исходным, ибо все вершины $y \in Y$ являются свободными и неизолированными в G. Следовательно, выполняются равенства $Xfree = X$ и $Yfree = Y$. Предположим, что поиск в глубину в процедуре $Increase(M)$ начнется с вершины x_1. Теперь все зависит от значения функции $Choice(x_1)$. Пусть $y_2 = Choice(x_1)$ (случай $y_1 = Choice(x_1)$ не так интересен). Тогда цепь x_1, x_1y_2, y_2 является M-чередующейся, и в M будет добавлено ребро x_1y_2. Пусть следующей вершиной, взятой из списка $Xfree$, была вершина x_4 и $Choice(x_4) = y_3$ (читатель заметил, что мы действуем максимально злоумышленно). Тогда в M добавится ребро x_4y_3. Больше ни одной M-чередующейся цепи в графе $G(M)$ после удаления вершин x_1, x_4, y_2, y_3 не существует. Первая фаза на этом закончится.

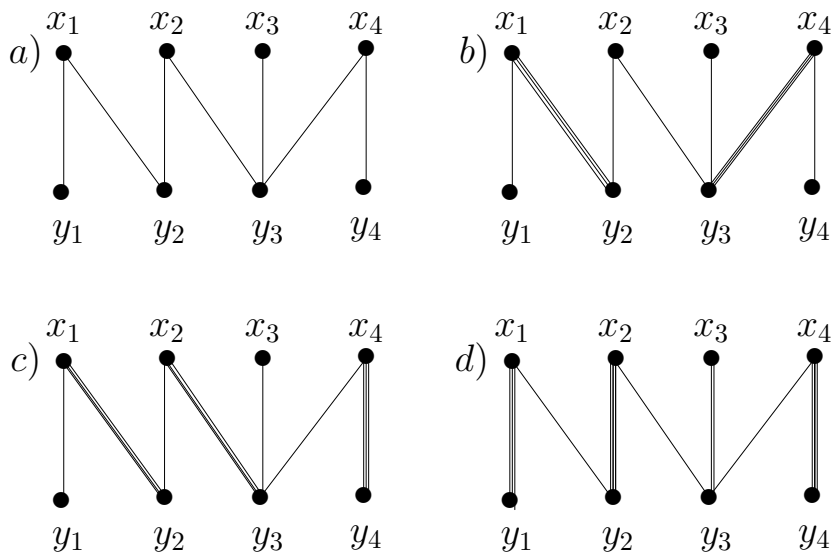


Рис. 25

Интересно отметить, что вспомогательный граф $G(M)$, построенный во второй фазе, в точности совпадает с исходным. Действительно, поиск в ширину начнется с вершин x_2 и x_3. Полный просмотр первого фронта приведет к тому, что в $G(M)$ попадут вершины y_2 и y_3, а через них — вершины x_1 и x_4. Затем из вершин x_1 и x_4 будут достигнуты вершины y_1 и y_4. Конечно, во вспомогательный граф попадут и все ребра исходного графа. Предположим, что поиск в глубину начнется с вершины x_2 и $Choice(x_2) = y_3$. Тогда будет найдена M-чередующаяся цепь с последовательностью вершин x_2, y_3, x_4, y_4. Паросочетание M увеличится вдоль найденной цепи следующим образом. Будет удалено ребро x_4y_3 и добавлены ребра x_2y_3 и x_4y_4. После удаления вершин x_2, y_3, x_4, y_4 ни одной M-чередующейся цепи в графе $G(M)$ уже не существует. Результат второй фазы изображен на рис. 25 *c*).

Наконец, в третьей фазе поиск в ширину начнется с единственной свободной вершины x_3. Вспомогательный граф представляет собой M-чередующуюся цепь, включающую вершины $x_3, y_3, x_2, y_2, x_1, y_1$. Увеличение M вдоль этой цепи приводит к паросочетанию, изображенному на рис. 25 *d*), которое является наибольшим паросочетанием в заданном графе.

Получим теперь оценку сложности алгоритма Хопкрофта-Карпа.

Лемма 1. *Пусть M и N — два паросочетания в двудольном графе $G = (X, E, Y)$ и $|M| = r < s = |N|$. Тогда симметрическая разность $M \oplus N$ содержит не менее $s - r$ непересекающихся по вершинам M-чередующихся цепей.*

Доказательство. Рассмотрим граф $\tilde{G} = (X, M \oplus N, Y)$ и обозначим через $G_1, \ldots, G_k$ компоненты связности этого графа. Поскольку M и N являются паросочетаниями, каждая вершина графа $\tilde{G}$ инцидентна не более чем одному ребру из $M \setminus N$ и не более чем одному ребру из $N \setminus M$, Отсюда следует, что каждая компонента связности имеет один из трех следующих видов:

1) изолированная вершина;

2) цикл четной длины с ребрами попеременно из $M \setminus N$ и $N \setminus M$;

3) цепь с ребрами попеременно из $M \setminus N$ и $N \setminus M$.

Обозначим через E_i множество ребер компоненты G_i. Пусть

$$d_i = |E_i \cap N| - |E_i \cap M|.$$

В тех случаях, когда компонента G_i имеет тип 1) или 2), получаем $d_i = 0$. В случае 3) либо $d_i = -1$, либо $d_i = 1$, либо $d_i = 0$. Причем случай $d_i = 1$ возможен только тогда, когда цепь начинается ребром из N и заканчивается ребром из N, т. е. является M-чередующейся. Остается показать, что $d_i = 1$ для не менее чем $s - r$ индексов. Достаточно убедиться, что сумма всех значений d_i не меньше $s - r$. В приводимых ниже вычислениях все суммы берутся по $i = 1, \ldots, k$.

$$\sum d_i = \sum (|E_i \cap N| - |E_i \cap M|) =$$

$$= |N \setminus M| - |M \setminus N| = |N| - |M| = s - r.$$

Из этого равенства следует, что в графе имеется не менее $s - r$ различных M-чередующихся цепей. □

Лемма 2. *Пусть M — паросочетание в двудольном графе G и $|M| = r < s$, где s — мощность наибольшего паросочетания в G. Тогда существует M-чередующаяся цепь длины не превосходящей* $\dfrac{2r}{s-r} + 1$.

Доказательство. Пусть N — наибольшее паросочетание в G. Тогда $|N| = s$ и по лемме 1 множество $M \oplus N$ содержит не менее $s - r$ непересекающихся по вершинам (а, следовательно, и по ребрам) M-чередующихся цепей. Пусть кратчайшая из них имеет длину $2k + 1$ (напомним, что длина каждой M-чередующейся цепи нечетна). Тогда ровно k ребер этой цепи принадлежат M. Следовательно, $(s - r)k \leqslant r$, так как каждая M-чередующаяся цепь содержит не менее k ребер из M. Из этого соотношения вытекает требуемое неравенство. □

Следующий результат является ключевым.

Лемма 3. *Пусть P и $\tilde{P}$ — чередующиеся цепи, построенные в разных фазах алгоритма Хопкрофта-Карпа, причем цепь P построена раньше, чем цепь $\tilde{P}$. Тогда*

$$|P| < |\tilde{P}|.$$

Доказательство. Достаточно рассмотреть случай, когда эти фазы непосредственно следуют друг за другом. Пусть M и N — паросочетания, которые были построены перед началом соответствующих фаз. Пусть $P_1, \ldots, P_k$ — максимальное по включению множество вершинно непересекающихся кратчайших M-чередующихся цепей, построенное алгоритмом, r — длина каждой из этих цепей. Тогда $P = P_i$ для некоторого i и $N = M \oplus P_1 \oplus \ldots \oplus P_k$.

Положим $L = N \oplus \tilde{P}$. Поскольку цепи P_i не пересекаются по вершинам (а потому не имеют общих ребер), имеем

$$M \oplus L = M \oplus M \oplus P_1 \oplus P_2 \ldots \oplus P_k \oplus \tilde{P} =$$
$$= \tilde{P} \oplus (P_1 \cup P_2 \cup \ldots \cup P_k).$$

Отсюда

$$|M \oplus L| = |\tilde{P}| + kr - 2|\tilde{P} \cap Q|,$$

где $Q = P_1 \cup P_2 \cup \ldots \cup P_k$.

С другой стороны, так как $|L| = |M| + k + 1$, множество $M \oplus L$ содержит не менее $k + 1$ реберно непересекающихся M-чередующихся цепей (лемма 1). Следовательно,

$$|M \oplus L| \geqslant (k + 1)r,$$

т. е. имеет место неравенство

$$|\tilde{P}| + kr - 2|\tilde{P} \cap Q| \geqslant (k + 1)r.$$

Отсюда

$$|\tilde{P}| \geqslant r + 2|\tilde{P} \cap Q|.$$

Так как в алгоритме Хопкрофта-Карпа выбиралось максимальное по включению множество вершинно непересекающихся цепей, цепь $\tilde{P}$ имеет общую вершину v хотя бы с одной из цепей $P_1, \ldots, P_k$. Поскольку после каждого увеличения относительно какой-либо чередующейся цепи все вершины цепи становятся насыщенными, эта общая вершина v является насыщенной, т. е. не первой и не последней в цепи $\tilde{P}$. Тогда темное относительно N ребро, инцидентое вершине v, входит в обе цепи, т. е. $\tilde{P} \cap Q \neq \emptyset$. Отсюда $|\tilde{P}| > r$. $\square$

Теорема 6.4. *Число фаз алгоритма Хопкрофта-Карпа не превышает* $2\lfloor\sqrt{s}\rfloor+1$, *где* s — *мощность наибольшего паросочетания в данном графе.*

Доказательство. Пусть $M_0 = \emptyset, M_1, \ldots, M_s$ — все паросочетания, $P_0, \ldots, P_{s-1}$ — все чередующиеся цепи, последовательно построенные алгоритмом, и

$$M_i = M_{i-1} \oplus P_{i-1}, i = 1, 2, \ldots, s.$$

Каждая цепь P_i является чередующейся для некоторого паросочетания M_j $(j \leqslant i)$, которое было построено перед началом очередной фазы. Поскольку внутри каждой фазы цепи не пересекаются по вершинам, цепь P_i является не только M_j-чередующейся, но также и M_k-чередующейся для всех k таких, что $j \leqslant k \leqslant i$. Еще раз отметим, что цепи, построенные в одной и той же фазе, имеют одинаковую длину, а в разных — разную. Таким образом, число фаз в алгоритме равно количеству различных чисел в последовательности

$$|P_0|, \ldots, |P_{s-1}|.$$

Пусть $r = \lfloor s - \sqrt{s} \rfloor$. Тогда $|M_r| = r < s$. Используя лемму 2 и несложные арифметические преобразования, получаем цепочку неравенств

$$|P_r| \leqslant \frac{2r}{s-r} + 1 = \frac{2s}{s-r} - 1 = \frac{2s}{s - \lfloor s - \sqrt{s} \rfloor} - 1 \leqslant$$

$$\leqslant \frac{2s}{\sqrt{s}} - 1 = 2\sqrt{s} - 1 \leqslant 2\lfloor\sqrt{s}\rfloor + 1.$$

Поскольку длина каждой цепи нечетна, последовательность $|P_0|, \ldots, |P_r|$ содержит не более $\lfloor\sqrt{s}\rfloor + 1$ различных чисел. Последовательность $|P_{r+1}|, \ldots, |P_{s-1}|$ может содержать не более $(s-1) - r$ других чисел. Кроме того,

$$(s-1) - r = (s-1) - \lfloor s - \sqrt{s} \rfloor \leqslant (s-1) - ((s - \sqrt{s}) - 1) = \sqrt{s}.$$

Окончательно получаем, что в последовательности чисел $|P_0|, \ldots, |P_{s-1}|$ имеется не более $2\lfloor\sqrt{s}\rfloor + 1$ различных нечетных чисел, что завершает доказательство теоремы. □

Теперь мы можем оценить вычислительную сложность алгоритма Хопкрофта-Карпа. Для данного двудольного графа $G = (X, E, Y)$ положим $n = \max(|X|, |Y|)$. Ясно, что мощность наибольшего паросочетания не превосходит n.

Теорема 6.5. *Алгоритм Хопкрофта-Карпа имеет сложность* $O(n^{5/2})$.

Доказательство. По теореме 6.4 число фаз имеет порядок $\sqrt{n}$. Остается оценить сложность выполнения каждой фазы. В процедуре $Graph(M)$ просматривается каждое ребро не более одного раза. Просмотр ребра означает просмотр соответствующего элемента матрицы смежности A, т. е. сложность этой процедуры есть $O(n^2)$. Поиск в глубину, выполняемый в процедуре $Increase(M)$, имеет сложность $O(p+q)$, поскольку использованные ребра и вершины удаляются из графа. Следовательно, сложность этой процедуры равна $O(n^2)$. Отсюда вытекает, что сложность алгоритма Хопкрофта-Карпа равна $(\sqrt{n}) \times O(n^2)$, т. е. равна $O(n^{5/2})$. $\square$

6.3. Задача о полном паросочетании. Алгоритм Куна

В этом разделе мы будем рассматривать двудольные графы $G = (X, E, Y)$, в которых множества X и Y имеют одинаковое число вершин. Пусть $m = |E|$ и $n = |X| = |Y|$. Паросочетание, насыщающее все вершины данного двудольного графа называется *полным* или *совершенным*. Задача, в которой требуется построить полное паросочетание, если оно существует, называется *задачей о полном паросочетании*.

Критерий существования полного паросочетания дает нам следствие 3 из разд. 4.11.

Теорема 6.6 (Холл, 1935). *В двудольном графе* $G = (X, E, Y)$ *полное паросочетание существует тогда и только тогда, когда для любого* $S \subseteq X$ *справедливо неравенство*

$$|S| \leqslant |E(S)|,$$

где $E(S)$ *обозначает множество всех вершин из* Y*, смежных с некоторыми вершинами из* S.

С точки зрения построения алгоритмов ценность этой теоремы невелика. В самом деле, для проверки выполнения условия существования полного паросочетания требуется просмотреть 2^n подмножеств множества X. Более того, даже если условия выполняются, теорема не дает никакого метода построения полного паросочетания. Тем не менее, эта теорема бывает зачастую полезной в различных вопросах теории графов и, кроме того, представляет самостоятельный интерес.

Одним из возможных методов решения задачи о полном паросочетании было бы применение алгоритма Хопкрофта-Карпа и выделение наибольшего паросочетания. Если это паросочетание состоит из n ребер, то оно является полным, а если наибольшее паросочетание имеет меньше чем n ребер, то в данном графе полного паросочетания не существует. Главным недостатком такого метода является то, что в случае отсутствия полного паросочетания мы узнаем об этом только после завершения работы алгоритма.

Рассмотрим теперь алгоритм, который завершает работу либо построением полного паросочетания, либо в тот момент (а он может наступить достаточно рано), когда станет ясно, что полного паросочетания не существует. Этот алгоритм составляет существенную часть метода, разработанного Куном в 1955 году для решения более общей задачи — задачи о назначениях. Кун назвал свой метод венгерским алгоритмом. Алгоритм построения полного паросочетания, который мы здесь разберем, будем называть алгоритмом Куна.

Неформально алгоритм Куна можно изложить следующим образом:

1) пустое паросочетание объявить текущим паросочетанием M;

2) если все вершины из X насыщены в M, то СТОП (M — полное паросочетание);

3) иначе выбрать произвольную свободную вершину $x \in X$ и искать M-чередующуюся цепь, начинающуюся в x;

4) если такая цепь P найдена, то положить $M = M \oplus P$ и вернуться на шаг 2;

5) иначе СТОП (полного паросочетания в заданном графе не существует).

Разберем изложенный алгоритм подробнее. Для поиска M-чередующейся цепи можно использовать как поиск в глубину, так и поиск в ширину. В данном случае удобнее использовать поиск в глубину. Правила поиска те же самые, что и в алгоритмах построения наибольшего паросочетания Форда-Фалкерсона или Хопкрофта-Карпа. Опять переход из вершин $x \in X$ к вершинам $y \in Y$ осуществляется по светлым относительно текущего паросочетания ребрам, а от $y \in Y$ к $x \in X$ — по темным.

Как всегда, возможны два исхода поиска: либо будет найдена свободная вершина $y \in Y$, т. е. найдена M-чередующаяся цепь, либо чередующейся цепи, с началом в корневой вершине поиска не существует.

В первом случае действия стандартны. Увеличиваем текущее паросочетание с помощью найденной цепи, и начинаем вновь искать M-

чередующуюся цепь из другой свободной вершины, если таковая существует.

Второй возможный исход поиска интереснее. Собственно, именно этот случай и является изюминкой предлагаемого алгоритма. Пусть поиск в глубину начинался с вершины x и M-чередующаяся цепь не была найдена. Тогда дерево поиска выглядит следующим образом. Вершина x находится в корне дерева поиска. Все вершины, уровень которых в дереве поиска есть число нечетное, принадлежат Y, а все вершины с четным уровнем — X, причем все вершины дерева за исключением x насыщены в M. Кроме того, ребра, исходящие из вершин с нечетным уровнем, соответствуют ребрам паросочетания M, а все прочие — ребрам, не входящим в M. Такое дерево часто называют *венгерским* или *чередующимся* деревом.

На рис. 26 дан пример графа и паросочетания в нем, а также дерево поиска в глубину из вершины x_4. На этом рисунке ребра паросочетания и соответствующие дуги изображены утолщенными линиями. Числа в скобках, проставленные рядом с вершинами, соответствуют тому порядку, в котором они просматривались в ходе поиска.

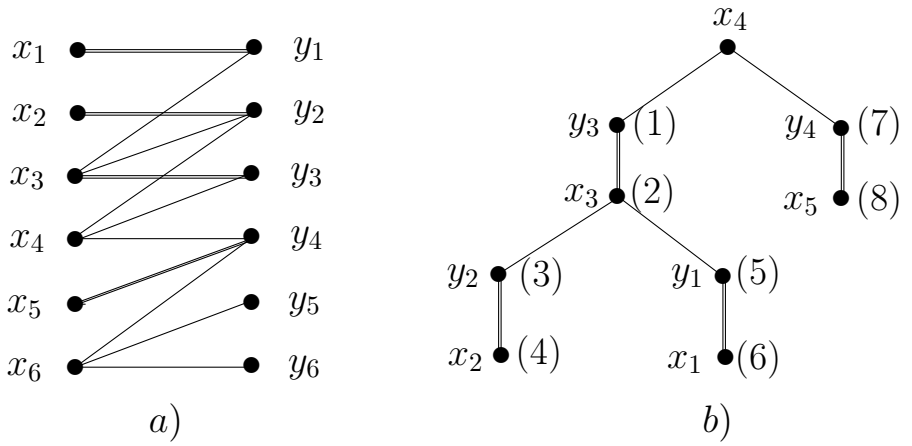


Рис. 26

Обозначим через S множество всех вершин дерева поиска, уровень k которых является четным числом. Тогда $S \subseteq X$. Пусть $E(S)$ — множество всех $y \in Y$, смежных с вершинами из S. Покажем, что все вершины $E(S)$ попадают в дерево поиска. Пусть $\tilde{y} \in E(S)$. Если вершина $\tilde{y}$ смежна с корневой вершиной поиска x, то соединяющее их ребро является светлым, так как x — свободная вершина. Тогда по правилам поиска вершина $\tilde{y}$ непременно будет помечена и, следовательно, $\tilde{y}$ входит в дерево поиска. Если $\tilde{y}$ смежна с насыщенной вершиной $\tilde{x} \in S$, то $\tilde{x}$

попадает в дерево поиска только после того, как туда попадет смежная с ней по темному ребру вершина y^*. Поскольку M — паросочетание, имеем $\tilde{y} = y^*$. Отсюда вытекает, что $\tilde{y}$ была помечена раньше чем x. Тем самым проверено, что $E(S)$ содержится в дереве поиска.

Пусть k нечетно. Тогда из каждой вершины y уровня k в дереве поиска исходит ровно одна дуга. Следовательно, если число k нечетно, то количество вершин, имеющих уровень k, равно количеству вершин уровня $k+1$. Из приведенных рассуждений следует, что $|S| = |E(S)|+1$, так как корневая вершина поиска является "лишней". Отсюда получаем $|S| > |E(S)|$, следовательно, граф G не имеет полного паросочетания в силу теоремы Холла. .

Проведенный анализ показывает, что, осуществляя поиск в глубину, мы находимся в беспроигрышной ситуации. Либо поиск завершится нахождением чередующейся цепи, и тогда текущее паросочетание можно увеличить, либо такая цепь не будет найдена, и тогда можно остановить работу алгоритма, ибо полного паросочетания не существует.

При формальной записи этого алгоритма предполагается, что двудольный граф $G = (X, E, Y)$ задан матрицей смежности $A[1..n, 1..n]$. Текущее паросочетание описывается двумя массивами $Xdouble$ и $Ydouble$ длины n каждый. Напомним, что $Xdouble[x] = y$, если x сочетается с y, и $Xdouble[x] = nil$, если x — свободная вершина относительно паросочетания M. Массив $Ydouble$ определяется аналогично.

Структура алгоритма Куна следующая. Через T обозначается множество свободных вершин относительно текущего паросочетания. Через $Start(T)$ обозначается функция, которая возвращает для непустого множества T возвращает произвольный элемент. В строках 3-4 инициализируется пустое паросочетание. В строках 6-21 осуществляется поиск в глубину из свободной вершины x. Эти строки являются почти точной копией аналогичных строк в процедуре $Increase(M)$ из предыдущего раздела. Вновь используется переменная $indication$, которая равна нулю, если свободная вершина $y \in Y$ еще не встретилась, и становится равной единице, как только достигается какая-нибудь свободная вершина $y \in Y$.

Функция $Choice(x)$ возвращает произвольную смежную с x вершину $y \in Y$, не посещавшуюся в ходе поиска из вершины x. Если такой вершины нет, то $Choice(x) = nil$. Отметим, что при программной реализации этой функции не обойтись без переменной, указывающей на то, посещалась или нет данная вершина.

Строки 6-21 показывают, что в стек S вершины помещаются парами.

Использованные в ходе поиска вершины удаляются из S тоже парами (строки 18-19). Строка 14 показывает, что сразу по достижению свободной вершины, т. е. такой вершины y, для которой $Ydouble[y] = nil$, переменная *indication* становится равной единице. После этого (условие в строке 7) процесс поиска из вершины x сразу остановится.

После завершения поиска в строке 22 анализируется, чем именно закончился поиск из данной вершины. В том случае, когда поиск завершился достижением свободной вершины, в строках 23-27 происходит увеличение текущего паросочетания. В строке 28 приведены два возможных исхода работы алгоритма 6.2.

Понятно, что условие $T = \emptyset$ означает, что все вершины множества X насыщены, т. е. текущее паросочетание является полным. Если же поиск из какой-либо вершины не завершился нахождением свободной вершины y, т. е. по окончании поиска имеем равенство $indication = 0$, то первое условие в строке 28 обеспечивает остановку алгоритма. Этот вариант окончания работы алгоритма говорит о том, что полного паросочетания не существует.

Алгоритм 6.2 (Кун).
Вход: двудольный граф $G = (X, E, Y)$, заданный матрицей $A[1..n, 1..n]$, где $n = |X| = |Y|$.
Выход: полное паросочетание, задаваемое массивами $Xdouble$ и $Ydouble$, либо сообщение о том, что такого паросочетания не существует.

```
1.  begin
2.    T := X;
3.    for x ∈ X  do  Xdouble[x] := nil;
4.    for y ∈ Y  do  Ydouble[y] := nil;
5.    repeat
6.      S := nil; x := Start(T); S ⇐ x; indication := 0;
7.      while (S ≠ nil) and (indication = 0) do
8.        begin
9.          x ⇐ S; y := Choice(x);
10.         if y ≠ nil then
11.           begin
12.             S ⇐ y; z := Ydouble[y];
13.             if z ≠ nil then S ⇐ z;
14.             else indication = 1;
15.           end
16.         else
```

```
17.          begin
18.             ⇐ S;
19.            if S ≠ ∅ then   ⇐ S;
20.          end
21.       end;
22.       if indication = 1 then
23.         while S ≠ nil do
24.          begin
25.            x ⇐ S; y ⇐ S; T := T \ {x};
26.            Xdouble[x] := y; Ydouble[y] := x;
27.          end
28.       until (indication = 0) or (T = ∅);
29.    end.
```

Так как поиск в глубину из заданной вершины имеет сложность $O(n^2)$ и основной цикл 5-28 алгоритма 6.2 выполняется не более n раз, справедлива

Теорема 6.7. *Алгоритм Куна имеет сложность* $O(n^3)$.

Вернемся к графу, изображенному на рис. 26 *а*). Паросочетание, изображенное на нем, могло быть получено в процессе работы алгоритма 6.2 следующим образом. Пусть первый раз функция *Start* выбрала вершину x_5. Тогда поиск в глубину с корнем x_5 сразу же находит свободную вершину y_4, и для текущего паросочетания M имеем $M = \{x_5y_4\}$. Пусть на втором шаге поиск велся из вершины x_2. Тогда для нового паросочетания M имеем $M = \{x_5y_4, x_2y_2\}$. Пусть на третьем шаге поиск велся из вершины x_3 и была найдена свободная вершина y_1. Тогда к текущему паросочетанию добавилось ребро x_3y_1. Предположим, что в четвертой итерации поиск начнется с вершины x_1. Тогда будет найдена свободная вершина y_3, при этом стек S после завершения поиска включает вершины x_1, y_1, x_3, y_3. В результате из паросочетания M будет удалено ребро x_3y_1 и будут добавлены два ребра x_1y_1 и x_3y_3.

Дальнейший поиск из вершины x_4, который не находит свободной вершины $y \in Y$, нами уже рассматривался. В результате работа алгоритма будет завершена, так как полного паросочетания в данном графе не существует. Отметим, что вершина x_6 при работе алгоритма даже не рассматривалась.

Завершая обсуждение алгоритма Куна, отметим, что он может быть реализован и на основе поиска в ширину. Все принципиальные моменты

построения алгоритма при такой замене сохраняются.

6.4. Задача о назначениях. Венгерский алгоритм

Пусть $G = (X, E, c, Y)$ — взвешенный двудольный граф, $|X| = |Y| = n$. Напомним,что *весом паросочетания* M называется сумма весов его ребер.

Задача о назначениях состоит в следующем: в заданном двудольном взвешенном графе найти полное паросочетание минимального веса.

Рассматривая эту задачу, мы будем предполагать, что заданный граф является полным, т. е. любые две вершины $x \in X$ и $y \in Y$ соединены ребром. Это предположение не ограничивает общности, ибо любую пару несмежных вершин x и y можно считать соединенной ребром веса, большего суммы весов всех исходных ребер графа.

Полное паросочетание минимального веса назовем для краткости *оптимальным паросочетанием.*

Лемма 1. *Если веса всех ребер графа, инцидентных какой-либо вершине, увеличить (уменьшить) на одно и то же число, то всякое оптимальное паросочетание в графе с новыми весами является оптимальным и в графе с исходными весами.*

Справедливость леммы 1 немедленно следует из того, что для каждой вершины полное паросочетание содержит ровно одно ребро, инцидентное этой вершине.

В частности, эта лемма позволяет рассматривать только такие графы, веса ребер которых неотрицательны. Действительно, пусть a — минимум весов ребер данного графа. Если $a < 0$, то увеличим вес каждого ребра на $-a$. Тогда веса всех ребер станут неотрицательными, а множество оптимальных паросочетаний не изменится.

Более того, можно рассматривать только те графы, у которых каждой вершине инцидентно хотя бы одно ребро нулевого веса. Действительно, достаточно для каждой вершины из весов всех инцидентных ей ребер вычесть минимальный из них.

Пусть $X' \subseteq X, Y' \subseteq Y$ и d — некоторое число. Будем говорить, что к графу $G = (X, E, c, Y)$ применена операция (X', d, Y'), если сначала из веса каждого ребра, инцидентного вершине из X', вычтено d, а затем к весу каждого ребра, инцидентного вершине из Y', прибавлено d. Следующий простой результат играет важную роль.

Лемма 2. *Пусть* $G = (X, E, c, Y)$ *— двудольный взвешенный граф с неотрицательными весами,* $X' \subseteq X, Y' \subseteq Y$ *и* $d = \min\{c(x, y) | x \in \in X', y \in Y \setminus Y'\}$. *Если к графу* G *применить операцию* (X', d, Y'), *то*

1) *веса всех ребер* G *останутся неотрицательными,*

2) *веса ребер вида* xy, *где* $x \in X'$, $y \in Y'$ *или* $x \in X \setminus X'$, $y \in Y \setminus Y'$ *не изменятся.*

Доказательство. Будем считать, что граф G задан квадратной матрицей весов A порядка n, в которой $A[x, y] = c(x, y)$. Не ограничивая общности, можно считать, что X' состоит из первых g элементов множества X, а Y' — из первых h элементов множества Y. Тогда, очевидно, число d равно минимальному элементу из числа тех элементов матрицы A, которые стоят в первых g строках и в последних $n - h$ столбцах. Уменьшение на число d весов всех ребер, инцидентных данной вершине $x \in X'$, означает вычитание числа d из всех элементов соответствующей строки матрицы A, а увеличение на d весов всех ребер, инцидентных данной вершине $y \in Y'$ означает прибавление числа d ко всем элементам соответствующего столбца матрицы A.

Схематично операцию (X', d, Y') изобразим с помощью рис. 27. Элементы матрицы A, находящиеся в области I, вначале уменьшаются на d, а затем увеличиваются на то же самое число d. В результате эти элементы не меняются, т. е. не меняются веса ребер вида xy, где $x \in X'$ и $y \in Y'$. Все элементы из области II останутся неотрицательными, ибо d не превосходит каждого из них. Элементы области III вообще никак не меняются, т. е. не меняются веса ребер вида xy, где $x \in X \setminus X'$ и $y \in Y \setminus Y'$. В области IV элементы увеличатся на число d. В результате они останутся неотрицательными. □

	Y'	$Y \setminus Y'$
X'	I	II $-d$
$X \setminus X'$	IV $+d$	III

Рис. 27

Следующий вспомогательный результат совершенно очевиден.

Лемма 3. *Если веса всех ребер графа неотрицательны и некоторое полное паросочетание состоит из ребер нулевого веса, то оно является оптимальным.*

Три сформулированные леммы позволяют разработать алгоритм построения полного паросочетания минимального веса в полном двудольном взвешенном графе. Этот алгоритм был предложен Куном и был назван им *венгерским алгоритмом*.

Сначала приведем неформальное описание венгерского алгоритма:

1) преобразовать веса ребер данного графа таким образом, чтобы веса всех ребер стали неотрицательными и каждой вершине стало инцидентно хотя бы одно ребро нулевого веса;

2) пустое паросочетание объявить текущим паросочетанием M;

3) если в графе все вершины насыщены относительно текущего паросочетания, то СТОП (текущее паросочетание оптимально);

4) иначе выбрать произвольную свободную вершину $x \in X$ и искать M-чередующуюся цепь, которая начинается в вершине x и состоит только из ребер нулевого веса;

5) если такая цепь P построена, то положить $M = M \oplus P$ и вернуться на шаг 3;

6) иначе для множества вершин $X' \subseteq X$ и $Y' \subseteq Y$, помеченных в ходе поиска (это вершины венгерского дерева), положить $d = \min\{c(x, y) | x \in \in X', y \in Y \setminus Y'\}$ и применить к графу операцию (X', d, Y');

7) из тех вершин $x \in X'$, которым стало инцидентно хотя бы одно ребро нулевого веса, возобновить поиск M-чередующейся цепи, используя только ребра нулевого веса; если такая цепь P будет построена, то положить $M = M \oplus P$ и вернуться на шаг 3, иначе вернуться на шаг 6 (множества X' и Y' при этом увеличатся).

Обоснуем теперь корректность алгоритма.

Докажем, что каждый поиск из очередной свободной вершины $x \in X$ завершится в конце концов построением M-чередующейся цепи, которая начинается в выбранной вершине x и состоит только из ребер нулевого веса.

Пусть выполнение шага 4 не привело к этой цели. Разберем подробнее выполнение шагов 5, 6 и 7. Поскольку поиск ведется по ребрам нулевого веса, каждое ребро вида xy, где $x \in X'$ и $y \in Y \setminus Y'$, имеет вес больше нуля, так как все эти ребра являются светлыми относительно текущего паросочетания. Поэтому $d > 0$. Тогда, очевидно, применение операции (X', d, Y') приводит к тому, что хотя бы у одной вершины $x^* \in X'$ появится инцидентное ей ребро нулевого веса и, следовательно, при выполнении шага 7 множество Y' увеличится хотя бы на одну вершину. Более того, применение операции (X', d, Y') в силу леммы 2 не меняет весов ребер дерева поиска, так как они имеют вид xy, где

$x \in X'$ и $y \in Y'$, и не меняет весов ребер текущего паросочетания, не попавших в дерево поиска, поскольку они имеют вид xy, где $x \in X \setminus X'$ и $y \in Y \setminus Y'$.

Таким образом, выполнение шага 6 не меняет нулевые значения весов ребер текущего паросочетания и в графе появляются новые ребра нулевого веса. Поскольку при выполнении шага 6 множество Y' увеличивается, этот шаг не может выполняться более n раз и, следовательно, на некотором выполнении шага 7 будет достигнута свободная вершина $y \in Y$, т. е. не более чем через n итераций будет построена M-чередующаяся цепь.

Поскольку поиск всегда ведется по ребрам нулевого веса, каждое текущее паросочетание имеет нулевой вес. Поэтому алгоритм завершит работу построением полного паросочетания нулевого веса, которое в силу леммы 3 будет оптимальным. Отметим, что в алгоритме используются те же правила поиска, что и в алгоритме Куна из предыдущего раздела: переход от вершин множества X к вершинам множества Y осуществляется по светлым ребрам , а от Y к X — по темным ребрам.

Перейдем теперь к формализованному изложению венгерского алгоритма. Будем считать, что полный двудольный взвешенный граф $G = (X, E, c, Y)$, где $|X| = |Y| = n$, задается матрицей весов A, в которой $A[x, y] = c(x, y)$ для любых $x \in X$ и $y \in Y$.

Как и раньше, текущее паросочетание M будем описывать двумя массивами $Xdouble$ и $Ydouble$. Поскольку матрица весов постоянно модифицируется с целью получения большего числа нулей, удобно иметь еще одну матрицу весов B, в которой и будут отражаться все изменения весов ребер. Так как в ходе поиска нас будут интересовать лишь ребра нулевого веса, заведем для каждой вершины $x \in X$ по списку $Bzero[x]$, который включает все такие вершины $y \in Y$, что $B[x, y] = 0$. Кроме того, для организации самого поиска удобно иметь динамически меняющуюся копию этого списка в виде стека, который будем обозначать через $S[x]$.

Опишем вначале процедуру поиска $Search(x)$. По сути дела она повторяет строки 5-15 алгоритма Куна из предыдущего раздела, но здесь нам удобнее записать ее рекурсивно. В ней используются обычным образом переменная $mark$ (для того, чтобы различать помеченные и непомеченные вершины) и массив $Previous$. Через X' и Y' обозначаются соответственно множества вершин из X и Y, помеченных в ходе поиска. Переменная $indication$ служит для прекращения поиска сразу после того, как достигается свободная вершина $y \in Y$. Поиск ведется лишь

при выполнении условия $indication = 0$. Перед первым вызовом этой процедуры выполняются равенства $S[x] = Bzero[x]$ для всех $x \in X$.

```
1.   procedure Search(x);
2.   begin
3.     while (S[x] ≠ nil) and (indication = 0) do
4.       begin
5.         y ⇐ S[x];
6.         if mark[y] = 0 then
7.           begin
8.             mark[y] := 1; Previous[y] := x; Y' := Y' ∪ {y};
9.             z := Ydouble[y];
10.            if z ≠ nil then
11.              begin
12.                mark[z] := 1; Previous[z] := y;
13.                X' := X' ∪ {z}; Search(z);
14.              end
15.            else indication := 1;
16.          end
17.        end
18.    end;
```

Теперь мы можем дать формализованное изложение венгерского алгоритма. В нем без формального описания будем использовать функции $Transform(A)$, $Start(T)$ и процедуру $Operation(X', d, Y')$.

Первая функция из исходной матрицы A сначала получает матрицу с неотрицательными значениями элементов, добавляя ко всем ее элементам достаточно большое положительное число. Затем, вычитая из каждой строки минимальный в этой строке элемент и действуя аналогично со столбцами, получает матрицу с неотрицательными значениями элементов, в которой в каждой строке и каждом столбце имеется хотя бы один нулевой элемент. После применения этой функции каждой вершине графа инцидентно хотя бы одно ребро нулевого веса. Ясно, что функция $Transform(A)$ имеет сложность $O(n^2)$. Преобразованную таким образом матрицу весов будем обозначать через B, и все дальнейшие изменения весов ребер будем отражать именно в ней.

Процедура $Operation(X', d, Y')$ сначала вычисляет значение $d = \min\{c(x, y) | x \in X', y \in Y \setminus Y'\}$, а затем применяет к графу операцию (X', d, Y') таким образом, как это описано выше. Данная операция вы-

полняется в текущей матрице весов — матрице B. Отметим, что новые нулевые значения в матрице B могут появиться только для пар $x \in X'$ и $y \in Y \setminus Y'$. Понятно, что эта процедура имеет сложность $O(n^2)$.

Через T будем обозначать множество вершин $x \in X'$, для которых существует вершина $y \in Y \setminus Y'$ такая, что $B[x, y] = 0$. Как уже отмечалось выше, после применения процедуры $Operation(X', d, Y')$ в множестве T появится хотя бы один ненулевой элемент. Функция $Start(T)$ выбирает произвольный элемент $x \in T$.

Алгоритм 6.3.
Вход: полный двудольный взвешенный граф $G = (X, E, c, Y)$, заданный матрицей весов $A[1..n, 1..n]$.
Выход: полное паросочетание минимального веса в графе G, заданное массивами $Xdouble[1..n]$ и $Ydouble[1..n]$.

```
1.      begin
2.        B := Transform(A);
3.        for x ∈ X  do  Bzero[x] := nil;
4.        for x ∈ X  do
5.          for y ∈ Y  do
6.            if B[x, y] = 0  then Bzero[x] ⇐ y;
7.        for x_0 ∈ X  do
8.          begin
9.            for y ∈ Y  do  mark[y] := 0;
10.           for x ∈ X  do
11.             begin
12.               mark[x] := 0; S[x] := Bzero[x];
13.             end;
14.         mark[x_0] := 1; X' := {x_0}; Y' := ∅;
15.         indication := 0; Search(x_0);
16.         if indication = 0  then
17.           repeat
18.             Operation(X', Y', d); T := ∅;
19.             for  x ∈ X'  do
20.               for y ∈ Y \ Y'  do
21.                 if B[x, y] = 0 then
22.                   begin
23.                     Bzero[x] ⇐ y; S[x] ⇐ y;
24.                     T := T ∪ {x};
25.                   end;
```

```
26.            while (T ≠ ∅) and (indication = 0) do
27.              begin x := Start(T); Search(x) end
28.            until indication = 1;
29.        x := Previous[y]; Xdouble[x] := y; Ydouble[y] := x;
30.        while x ≠ x0 do
31.          begin
32.            y := Previous[x]; x := Previous[y];
33.            Xdouble[x] := y; Ydouble[y] := x;
34.          end
35.      end
36.    end.
```

Дадим комментарий к венгерскому алгоритму. В основном цикле 7-35 осуществляется поиск из очередной вершины $x_0 \in X$, который ведется до тех пор, пока не будет найдена свободная вершина $y \in Y$ и, тем самым, чередующаяся цепь относительно текущего паросочетания. Этот поиск осуществляется в два этапа. Вначале, после инициализации исходных данных (строки 9-15), вызывается процедура поиска в глубину из корневой вершины x_0. Затем (условие в строке 16), если в ходе этого поиска не удалось достичь свободной вершины $y \in Y$, процедура $Operation(X', d, Y')$, выполняемая в строке 18, позволяет расширить поле поиска, ибо появятся новые ребра нулевого веса.

Новый поиск, как это следует из строк 20-27, может выполняться из любой вершины венгерского дерева. Условие в строке 28 показывает, что такой новый поиск ведется до достижения свободной вершины $y \in Y$. Как уже отмечалось выше, свободная вершина будет достигнута не более чем за n итераций цикла 17-28. В строках 29-34 увеличивается текущее паросочетание. В новом паросочетании вершина x_0 становится насыщенной и остается такой на протяжении дальнейшей работы алгоритма.

Теорема 6.8. *Алгоритм* 6.3 *имеет сложность* $O(n^4)$.

Доказательство. Функция $Transform(A)$ имеет сложность $O(n^2)$. Такую же сложность имеет цикл, выполняемый в строках 4-6. Основной цикл 7-35 выполняется ровно n раз. Остается определить сложность выполнения всех операций внутри основного цикла. Поиск в глубину, вызываемый в строке 15, имеет сложность $O(n^2)$. Цикл 17-28 имеет сложность $O(n^2)$, поскольку содержит процедуру $Operation(X', d, Y')$. Этот цикл работает, вообще говоря, n раз. Следовательно, сложность

цикла 17-28 есть $O(n^3)$. Понятно, что увеличение текущего паросочетания, выполняемое в строках 29-34, требует не более cn операций, где c — константа. Окончательно, получаем оценку сложности алгоритма $O(n^4)$.

Замечание. В венгерском алгоритме оценка $O(n^4)$ возникает по сути дела из-за того, что приходится порядка n^2 раз выполнять операцию (X', d, Y'), которая имеет сложность $O(n^2)$. Между тем, известен способ реализации этой операции со сложностью $O(n)$. Для этого достаточно применить прием, использованный нами в алгоритме Ярника-Прима-Дейкстры. А именно, при каждом входе в основой цикл 7-35 необходимо завести два массива D и $Near$ длины n, которые динамически меняются в ходе поиска таким образом, что для всех $y \in Y \setminus Y'$ выполняется равенство

$$D[y] = \min\{c(x, y) | x \in X'\},$$

и указатель $Near[y]$ дает ту вершину $x \in X'$, для которой этот минимум достигается, т. е.

$$D[y] = c(Near[y], y).$$

Тогда вычисление величины d требует порядка n операций, ибо

$$d = \min\{D[y] | y \in Y \setminus Y'\}.$$

Модифицировать можно веса не всех ребер, а только ребер вида $Near[y]y$, что также требует порядка n операций. Тем самым результат операции (X', d, Y') можно получить за время $O(n^2)$. Такая реализация венгерского алгоритма имеет сложность $O(n^3)$. Все детали изложенного метода можно найти в [44].

Разберем пример, иллюстрирующий работу алгоритма 6.3. На рис. 28 *a*) изображена матрица весов исходного графа, а на рис. 28 *b*) — результат применения к ней функции $Transform(A)$.

$$\begin{pmatrix} 1 & 4 & 4 & 3 \\ 2 & 7 & 6 & 8 \\ 4 & 7 & 5 & 6 \\ 2 & 5 & 1 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 3 & 2 \\ 0 & 2 & 4 & 6 \\ 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 2 & 1 \\ 0 & 2 & 3 & 5 \\ 0 & 0 & 0 & 1 \\ 2 & 2 & 0 & 0 \end{pmatrix}$$

a) *b*) *c*)

Рис. 28

Матрица 28 *b*) получена следующим образом. Сначала из каждой строки вычли ее минимальный элемент, а именно, из первой строки вычли 1, из второй — 2, из третьей — 4, из четвертой — 1. Затем из второго столбца (поскольку это единственный из столбцов, который не содержит нулей) вычли 3. В результате в матрице 28 *b*) в каждой строке и каждом столбце имеется хотя бы один нуль.

В дальнейшем будем предполагать, что вершины просматриваются циклами в порядке возрастания их номеров. Поэтому основной цикл в строках 7-35 венгерского алгоритма начнется с вершины x_1. Сразу же будет найдена свободная вершина y_1 и, следовательно, текущее паросочетание M приобретет вид $M = \{x_1y_1\}$. Следующей вершиной будет вершина x_2. Дерево поиска из вершины x_2 изображено на рис. 29 *a*), где цифры в скобках означают порядок, в котором вершины встречаются в ходе поиска. Дуги, соответствующие ребрам паросочетания, изображаются сплошными линиями, а прочие — пунктиром.

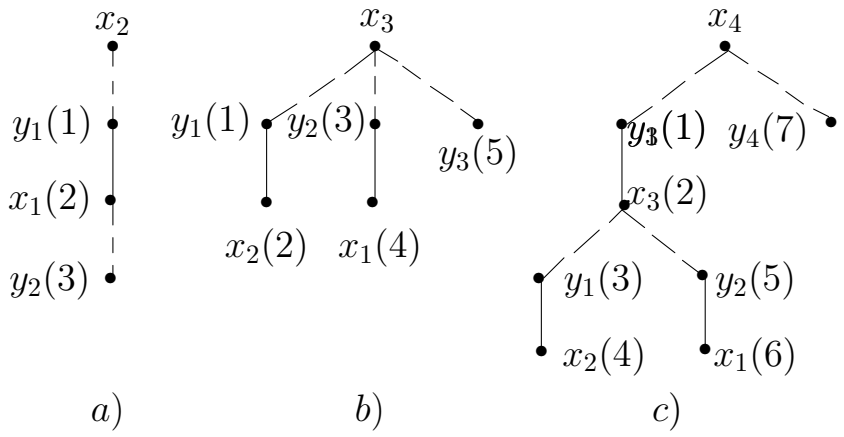


Рис. 29

Поиск из вершины x_2 завершается достижением свободной вершины y_2 и, следовательно, текущее паросочетание будет иметь вид $M = \{x_1y_1, x_2y_2\}$.

Поиск из вершины x_3 не приводит к достижению свободной вершины. Соответствующее дерево поиска изображено на рис. 29 *b*). Тогда мы имеем $X' = \{x_1, x_2, x_3\}$ и $Y' = \{y_1, y_2\}$. Поэтому d равно минимальному элементу среди элементов матрицы 28 *b*), находящихся в первых трех строках и последних двух столбцах. Следовательно, $d = 1$. Применяя к матрице 28 *b*) операцию (X', d, Y'), получаем матрицу 28 *c*). В результате у вершины x_3 появится новое инцидентное ей ребро нулевого веса — x_3y_3. Поиск возобновится из вершины x_3 и сразу же завершится до-

стижением свободной вершины y_3. Таким образом, после этой итерации получим $M = \{x_2y_1, x_1y_2, x_3y_3\}$.

Дерево поиска из вершины x_4 изображено на рис. 29 *c*). К текущему паросочетанию M добавится ребро x_4y_4. Тем самым, текущее паросочетание $M = \{x_2y_1, x_1y_2, x_3y_3, x_4y_4\}$ станет полным и будет состоять из ребер нулевого веса, ъгде веса заданы преобразованной матрицей. Это паросочетание является оптимальным, и его вес, вычисленный по исходной матрице, равен 12.

7. Задача коммивояжера

7.1. Основные понятия

Все задачи, рассмотренные нами в предыдущих главах, имеют одну общую черту: для них известны (и были нами рассмотрены) алгоритмы решения, имеющие полиномиальную сложность. Однако, для очень большого числа естественно возникающих задач оптимизации на графах эффективных (т. е. имеющих полиномиальную сложность) алгоритмов до сих пор не найдено, но одновременно доказано, что эти задачи в некотором вполне определенном смысле трудны. Назовем такие задачи труднорешаемыми. Имеются веские доводы, позволяющие предположить, что для труднорешаемых задач эффективных алгоритмов решения не существует. Рассмотрение возникающих здесь проблем выходит за рамки данной книги, и мы можем лишь порекомендовать читателям книгу [18], посвященную труднорешаемым задачам. А в этой главе мы рассмотрим методы решения одной из таких задач — задачи коммивояжера.

Пусть дан обыкновенный связный граф. Цикл, включающий все вершины графа, называется гамильтоновым. Отметим, что задача о том, существует или нет в данном графе гамильтонов цикл, является труднорешаемой (см. [18]).

Задача коммивояжера формулируется следующим образом. В данном обыкновенном взвешенном графе найти гамильтонов цикл наименьшего веса, где вес цикла определяется как сумма весов входящих в него ребер.

Разберем пример, поясняющий такое название задачи. Предположим, что некоторому коммивояжеру требуется посетить каждый город в пределах конкретной зоны обслуживания, побывав в них ровно по одному разу, и возвратиться домой. Естественно, что ему хотелось бы выбрать такой порядок обхода клиентов, при котором его путь был бы возможно короче. Построим взвешенный граф $G = (V, E, c)$, в котором каждая вершина соответствует некоторому городу, а веса ребер равны расстояниям между соответствующими городами. Гамильтонов цикл наименьшего веса в этом графе дает желаемый маршрут для коммивояжера.

Задача коммивояжера (далее ЗК) также относится к классу труднорешаемых. Известно, что ЗК остается труднорешаемой и в том случае, когда граф является полным, а матрица весов A удовлетворяет нера-

венству треугольника, т. е.

$$A[v, w] \leqslant A[v, u] + A[u, w] \text{ для всех } u, v, w \in V.$$

Более того, ЗК остается труднорешаемой и в классе евклидовых графов, т. е. графов, вершины которых являются точками евклидового пространства, а веса ребер равны расстояниям между соответствующими точками.

Отметим, что если веса всех ребер графа увеличить на одно и то же число, то гамильтонов цикл наименьшего веса в графе с измененными весами будет решением ЗК и для исходного графа. Следовательно, не ограничивая общности, можно считать, что веса всех ребер данного графа неотрицательны. Напомним, что матрица весов неориентированного графа симметрична. Этот факт будет использоваться нами при доказательстве некоторых теорем без специального упоминания.

Для удобства гамильтонов цикл будем называть *маршрутом коммивояжера*, а маршрут с наименьшим весом — *оптимальным маршрутом*.

7.2. Алгоритм отыскания гамильтоновых циклов

Пусть G — произвольный n-граф. Опишем алгоритм, позволяющий найти в графе G все гамильтоновы циклы или выдать сообщение, что таких циклов нет. Пусть v_0 — произвольная вершина графа G. Рассмотрим некоторый гамильтов цикл

$$v_0 = u_1, u_2, \ldots, u_n, u_{n+1} = v_0.$$

Удалив из этого цикла ребро $u_n v_0$, мы получим максимальную простую цепь

$$v_0 = u_1, u_2, \ldots, u_n,$$

в которой начальная вершина $v_0 = u_1$ смежна с конечной вершиной u_n.

Нетрудно понять, что если мы научимся строить все максимальные простые цепи, имеющие начало в вершине v_0, то задача о нахождении гамильтоновых циклов будет решена. В самом деле, пусть

$$P : v_0 = u_1, u_2, \ldots, u_k$$

— максимальная простая цепь с началом в вершине v_0. Если $k = n$ и вершина u_k смежна с вершиной v_0, то добаляя к цепи P ребро $u_k v_0$, мы получим гамильтонов цикл.

Пусть $\mathcal{M}$ — множество всех простых цепей, имеющих начало в вершине v_0. Для произвольных цепей $P, Q \in \mathcal{M}$ положим $P \leqslant Q$, если цепь P является началом цепи Q. Ясно, что отношение $\leqslant$ на множестве $\mathcal{M}$ является отношением частичного порядка. Максимальные элементы частично упорядоченного множества $\mathcal{M}$ мы назвали выше максимальными простыми цепями с началом в вершине v_0.

Алгоритм, позволяющий перечислить по одному разу все гамильтоновы циклы графа G, многократно выполняет следующую работу: имея текущую простую цепь

$$P : v_0 = u_0, u_1, \ldots, u_{k-1},$$

он по очереди добавляет к ней новые вершины, продолжая ее до всевозможных максимальных простых цепей с началом в вершине v_0.

Договорися об обозначениях. Вершины текущей простой цепи будем хранить в массиве x длины $n-1$ (поскольку начальная вершина v_0 зафиксирована, хранить ее в массиве x не надо). В процессе работы алгоритма каждая вершина в каждый текущий момент может находиться в одном из двух состояний: быть *включенной* или быть *невключенной*. Вершина считается включенной в текущий момент, если и только если она включена в текущую простую цепь. Массив $status$ длины n позволит отличать включенные вершины от невключенных: в любой текущий момент $status[v] = 1$, если вершина v включена, и $status[v] = 0$, если вершина v невключена.

Если текущая простая цепь имеет вид

$$v_0, x[1], x[2], \ldots, x[k-1],$$

то через S_k для $k \geqslant 1$ обозначим множество всех вершин, которые можно использовать для продолжения этой цепи. Ясно, что S_k состоит из всех невключенных вершин, смежных с вершиной $x[k-1]$. Разумеется, S_1 в начале работы алгоритма совпадает с множеством вершин $list[v_0]$, смежных с v_0.

Рассмотрим теперь следующую рекурсивную процедуру.

```
1.  procedure Hamiltonian_Cycles(k);
2.  begin
3.    for y ∈ S_k  do
4.      if k = n − 1 and y смежна с v_0 then
5.        write(v_0, x[1], x[2], ..., x[n − 2], y, v_0)
6.      else
```

```
7.        begin
8.          status[y] := 1; S_k := S_k \ {y};
9.          x[k] := y;
10.         S_{k+1} := {v|v ∈ list[y] и status[v] = 0};
11.         Hamiltonian_Cycles(k + 1);
12.         status[x[k]] := 0
13.       end
14.   end;
```

Указанная процедура методично перебирает все простые цепи, являющиеся продолжениями простой цепи $P : v_0, x[1], \ldots, x[k-1]$, добавляя по очереди новые вершины. Если найдена максимальная простая цепь, то происходит возврат, т. е. из максимальной цепи отбрасывается одна или несколько последних вершин (конечно, в случае, когда найденная максимальная простая цепь содержится в гамильтоновом цикле, перед возвратом этот цикл выводится на печать). При этом $status[v]$ принимает значение 0 для каждой из отброшенных вершин v. Алгоритм, реализованный в данной процедуре относят к классу *алгоритмов с возвратом*.

Теперь легко описать алгоритм, перечисляющий все гамильтоновы циклы, если они имеются в графе G.

Алгоритм 7.1.

```
1.  begin
2.    status[v_0] = 1; S_1 := list[v_0];
3.    for v ∈ V \ {v_0} do status[v] := 0;
4.    Hamiltonian_Cycles(1)
5.  end.
```

Указанный алгоритм полным перебором находит и выводит на печать все гамильтоновы циклы графа G, рассматривая вершину v_0 в качестве начальной. Корректность этого алгоритма очевидна. Ясно, что алгоритм экспоненциален и может быть реально применен к графам с весьма малым числом вершин.

7.3. Алгоритмы решения задачи коммивояжера с гарантированной оценкой точности

Один из возможных подходов к труднорешаемым задачам заключается в построении алгоритмов полиномиальной сложности для получе-

ния «хорошего», но, возможно, не оптимального результата. Сразу же возникает проблема: как сильно отличается найденное решение от оптимального? Обычно легче сконструировать быстрый алгоритм, дающий правдоподобное решение, чем оценить его погрешность.

Рассмотрим, например, простейший алгоритм построения маршрута коммивояжера, реализующий «жадный» алгоритм и называемый *Nearest_ vertex* или *Ближайший сосед*. В качестве начальной вершины выбираем произвольную вершину и объявляем ее последней включенной в маршрут. Далее, пусть v — последняя включенная в маршрут вершина. Среди всех еще не включенных в маршрут вершин выбираем ближайшую к v вершину w, включаем w в маршрут после вершины v и объявляем w последней включенной вершиной. Если все вершины включены в маршрут, то возвращаемся в исходную вершину.

Изложенный алгоритм легко реализовать так, чтобы он имел сложность $O(n^2)$. Понятно, что этот алгоритм иногда может находить оптимальный маршрут, но так будет далеко не всегда. Оценку возможной ошибки дает следующая

Теорема 7.1. *Пусть $G = (V, E, c)$ — полный взвешенный граф, матрица весов которого неотрицательна и удовлетворяет неравенству треугольника. Пусть $Nvt(G)$ — маршрут коммивояжера, построенный алгоритмом $Nearest\_vertex$, $Opt(G)$ — оптимальный маршрут, а $c(Nvt(G))$ и $c(Opt(G))$ — их веса. Тогда*

$$c(Nvt(G)) \leqslant \frac{1}{2}(\lfloor \log n \rfloor + 1) \cdot c(Opt(G)).$$

Доказательство данной теоремы можно найти в [46]. Эта теорема дает только верхнюю оценку отношения веса решения $Nvt(G)$ к весу оптимального решения $Opt(G)$ и не говорит о том, насколько плохим на самом деле может быть такое отношение. Имеются примеры графов, для которых оно больше чем $\frac{1}{3}\log n$. Таким образом, алгоритм *Nearest_ vertex* может иногда давать решения очень далекие от оптимальных.

Однако, можно получить лучшие результаты, используя чуть более искусную стратегию. Так алгоритм $Nearest\_insert$ или *Ближайшая вставка*, начиная с “цикла”, состоящего из одной вершины, шаг за шагом наращивает растущий цикл в полном графе до тех пор, пока он не

включит в себя все вершины графа. Пусть $T \subseteq V$. Для произвольной вершины $v \in V$ положим

$$d(v, T) = \min\{c(v, w) | w \in T\}.$$

Число $d(v, T)$ естественно назвать расстоянием от вершины v до множества T.

Неформальное описание этого алгоритма выглядит следующим образом:

1) произвольную вершину $v \in V$ объявить текущим маршрутом T;

2) если все вершины графа содержатся в T, то СТОП (T — маршрут коммивояжера);

3) иначе, среди всех вершин, не входящих в текущий маршрут T, найти такую вершину v, для которой величина $d(v, T)$ минимальна (вершина v ближе всего находится к T). Пусть w — вершина из T, для которой $d(v, T) = c(v, w)$, и u — вершина, следующая за w в маршруте T;

4) добавить вершину v в текущий маршрут T, вставив ее между w и u. Перейти на шаг 2.

Приведем формализованное изложение этого алгоритма. Текущий маршрут удобно описывать массивом $next$, где $next[v]$ дает имя вершины, которая следует за v в данном маршруте. Пусть T — множество вершин, включенных в маршрут. Для каждой вершины $v \in V \setminus T$ расстояние от v до T будем хранить в массиве с именем d. Значение $near[v]$ ($v \in V \setminus T$) дает имя вершины $w \in V \setminus T$, для которой выполняется равенство $c(w, v) = d[v]$. Иначе говоря, вершина $near[v]$ ближе всего расположена к v среди всех вершин множества T. Введение массивов d и $near$ позволяет реализовать алгоритм $Nearest\_insert$ так, чтобы он имел сложность $O(n^2)$. Такой прием был применен нами ранее в алгоритме Ярника-Прима-Дейкстры.

Через P обозначается множество вершин, не входящих в текущий маршрут. Функция $Min(P)$ дает имя вершины $v \in P$, для которой значение $d[v]$ минимально.

Алгоритм 7.2 ($Nearest\_insert$).
Вход: Полный взвешенный граф $G = (V, E, c)$, заданный матрицей весов $A[1..n, 1..n]$, причем $A[v, v] = 0$ для всех $v \in V$.
Выход: Маршрут коммивояжера, заданный массивом $next[1..n]$, S — вес найденного маршрута.

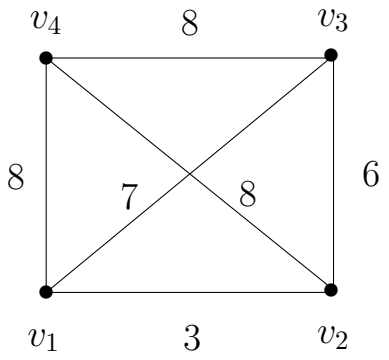
1. **begin**
2. v_1 := произвольная вершина из V;

```
3.    next[v_1] := v_1; S := 0; P := V \ {v_1};
4.    for v ∈ P do
5.      begin
6.        d[v] := c(v, v_1); near[v] := v_1;
7.      end;
8.    for k := 2 to n do
9.      begin
10.       v_k := Min(P); P := P \ {v_k};
11.       w := near[v_k]; u := next[w];
12.       next[w] := v_k; next[v_k] := u;
13.       S := S + c(w, v_k) + c(v_k, u) − c(w, u);
14.       for v ∈ P do
15.         if c(v, v_k) < d[v] then
16.           begin
17.             d[v] := c(v, v_k); near[v] := v_k;
18.           end
19.     end
20. end.
```

Алгоритм имеет следующую структуру. В строках 2-3 инициализируется маршрут, состоящий из одной вершины v_1. В строках 4-7 задаются начальные значения массивов d и $near$. В основном цикле 8-19 наращивается текущий маршрут. В строке 11 определяются те вершины, между которыми будет вставлена очередная вершина v_k. В строке 12 осуществляется эта вставка. Отметим, что вершина v_k добавляется сразу после ближайшей к ней вершине w. Это означает, что в маршрут добавляются ребра wv_k, v_ku и удаляется ребро wu. Поэтому вес S маршрута пересчитывается так, как это указано в строке 13. В цикле 14-18 пересчитываются значения массивов d и $near$.

Иллюстрирует работу алгоритма 7.2 пример, изображенный на рис. 30. Состояние текущего маршрута дано после прохождения основного цикла в строках 8-19.

Маршрут коммивояжера, найденный алгоритмом, имеет вес 26, в то время как вес оптимального маршрута v_1, v_2, v_3, v_4, v_1 равен 25. Интересно отметить, что в этом примере более грубый алгоритм *Nearest_ vertex* тем не менее находит именно оптимальный маршрут.



k	текущий маршрут
	v_1
2	v_1, v_2, v_1
3	v_1, v_2, v_3, v_1
4	v_1, v_2, v_4, v_3, v_1

Рис. 30

Поскольку каждая итерация цикла 8-19 требует порядка $n - k$ операций, то справедлива

Теорема 7.2. *Алгоритм Nearest_insert имеет сложность* $O(n^2)$.

Оказывается, что алгоритм *Nearest_insert*, несмотря на свою простоту, обладает удивительным свойством.

Теорема 7.3. *Пусть* $G = (V, E, c)$ *— полный неориентированный граф, матрица весов которого неотрицательна и удовлетворяет неравенству треугольника. Пусть* $Nins(G)$ *— маршрут коммивояжера, построенный алгоритмом Nearest_insert,* $Opt(G)$ *— оптимальный маршрут, а* $c(Nins(G))$ *и* $c(Opt(G))$ *— их веса. Тогда*

$$c(Nins(G)) \leqslant 2c(Opt(G)).$$

Доказательство. Пусть $v_1, \ldots, v_n$ — последовательность вершин графа G, занумерованных в том порядке, в котором они добавлялись в текущий маршрут алгоритмом *Nearest_insert*. Обозначим через R множество ребер, входящих в $Opt(G)$. Мы будем доказывать теорему путем построения взаимно-однозначного соответствия между вершинами $v_1, \ldots, v_n$ и ребрами из R таким образом, чтобы стоимость включения вершины v_k в маршрут $Nins(G)$ не превосходила удвоенной стоимости ребра из R, соответствующего v_k.

Поставим в соответствие вершине v_1 одно (любое) из двух инцидентных v_1 ребер, имеющихся в $Opt(G)$. Обозначим это ребро через e_1. Пусть $R_1 = R \backslash e_1$. Тогда граф $H = (V, R_1)$ состоит ровно из одной компоненты

связности, являющейся цепью, и в этой компоненте содержится вершина v_1. Кроме того, стоимость включения v_1 в текущий маршрут равна нулю, что меньше или равно $2c(e_1)$.

Пусть для вершин $v_1, \ldots, v_{k-1}$ выбраны ребра $e_1, \ldots, e_{k-1}$ из R так, что выполняются условия

1) $e_i \neq e_j$ при $1 \leqslant i < j \leqslant k-1$;

2) $R_i = R_{i-1} \setminus e_i$ для любого $i = 2, ..., k-1$;

3) граф $H_{k-1} = (V, R_{k-1})$ состоит ровно из $k-1$ компонент связности, которые являются цепями и в каждой из которых имеется точно по одной вершине из числа вершин $v_1, \ldots, v_{k-1}$.

Подберем ребро e_k для вершины v_k так, чтобы выполнялись все эти условия. Пусть в той компоненте связности H_{k-1}, которая содержит v_k, содержится вершина v_i, где $i < k$. Тогда имеется ровно одно ребро, инцидентное v_i и лежащее на цепи, соединяющей v_k и v_i. Это ребро поставим в соответствие вершине v_k и обозначим его через e_k. Заметим, что оно инцидентно точно одной вершине из множества $\{v_1, \ldots, v_k\}$, а именно v_i. Положим $R_k = R_{k-1} \setminus e_k$, и $H_k = (V, R_k)$. Легко видеть, что полученный граф имеет ровно k компонент связности (цепей) и в каждой из них имеется ровно по одной вершине из множества $\{v_1, ..., v_k\}$.

Оценим теперь стоимость включения v_k в текущий маршрут. Пусть v_k включается после вершины w и перед вершиной u. Стоимость включения v_k в маршрут равна (см. строку 13 алгоритма)

$$c(w, v_k) + c(v_k, u) - c(w, u).$$

Поскольку алгоритм выбирает для включения вершину, ближайшую к текущему маршруту, а ребро e_k инцидентно точно одной из ранее выбранных вершин, справедливо неравенство $c(w, v_k) \leqslant c(e_k)$.

Из этого неравенства и неравенства треугольника получаем

$$c(v_k, u) \leqslant c(u, w) + c(w, v_k) \leqslant c(u, w) + c(e_k).$$

Отсюда выводим

$$c(w, v_k) + c(v_k, u) \leqslant c(w, u) + 2c(e_k),$$

что эквивалентно соотношению

$$c(w, v_k) + c(v_k, u) - c(w, u) \leqslant 2c(e_k).$$

Последнее неравенство означает, что стоимость добавления новой вершины в маршрут не превосходит удвоенного веса ребра, соответствующего этой вершине. Поскольку вес маршрута $Nins(G)$ равен сумме

стоимостей включения вершин, справедливо неравенство $c(Nins(G)) \leqslant 2c(Opt(G))$. $\square$

Замечание. Если в графе существует вершина, которой инцидентны только ребра положительного веса, то справедливо неравенство $c(Nins(G)) < 2c(Opt(G))$. Для доказательства достаточно взять эту вершину в качестве v_1, и тогда уже на первом шаге стоимость включения (она равна нулю) строго меньше удвоенного веса ребра e_1.

Разберем еще один любопытный алгоритм построения маршрута коммивояжера, также имеющий гарантированную оценку точности.

Напомним, что замкнутая цепь в связном графе называется эйлеровой цепью, если она включает каждое ребро графа ровно один раз.

Пусть $G = (V, E)$ — полный граф, T — остовное дерево графа G. Построим граф H на том же множестве вершин V, используя две копии каждого ребра из ET. Степени всех вершин графа H четны, поэтому в H существует эйлерова цепь P. Выпишем все вершины v из V ровно по одному разу в том порядке, в котором они встречаются в P. Получим некоторую последовательность $v_1, \ldots, v_n$. Тогда цикл $v_1, v_1v_2, v_2, \ldots, v_n, v_nv_1, v_1$ образует маршрут коммивояжера в графе G. Будем говорить, что этот маршрут коммивояжера *вложен* в эйлерову цепь P.

Можно сказать, что минимальный остов графа является в некотором смысле приближением оптимального маршрута коммивояжера. Действительно, если в минимальном остове все вершины имеют степень не более двух, то этот остов представляет собой цепь, включающую все вершины графа. Остается дополнить эту цепь одним-единственным ребром для того, чтобы получить хороший маршрут коммивояжера. Конечно, такие остовы существуют редко, но, тем не менее, можно попытаться переделать произвольный минимальный остов в приемлемый маршрут коммивояжера. Именно на этой идее базируется алгоритм $Minspantreetravelling$ (от англ. *minimal spanning tree travalling*) или *Остовный обход* для построения маршрута коммивояжера в полном взвешенном графе $G = (V, E, c)$. Приведем сначала неформальное изложение этого алгоритма:

1) построить минимальное остовное дерево T графа G;

2) построить граф H, используя две копии каждого ребра из ET, и найти в H эйлерову цепь P;

3) построить маршрут коммивояжера, вложенный в эйлерову цепь P.

В формализованной записи алгоритма $Minspantreetravelling$ без по-

дробного описания используется процедура $Ostov(G)$, которая строит минимальное остовное дерево T графа G. Считаем, что граф H получен из T удвоением каждого ребра. Процедура $Eiler(H)$ строит эйлеров маршрут, представленный вершинами в стеке $SRes$.

Маршрут коммивояжера, вложенный в эйлерову цепь, представлен массивом mk, где $mk[i]$ дает имя i-ой вершины в маршруте коммивояжера. Массив $mark$ длины $n+1$ необходим для того, чтобы включать соответствующую вершину в маршрут коммивояжера лишь при первом ее появлении в стеке $SRes$. При этом первая и последняя вершины совпадают.

Алгоритм 7.3 (*Minspantreetravelling*).
Вход: полный взвешенный граф $G = (V, E, c)$, заданный матрицей весов $A[1..n, 1..n]$, причем $A[v, v] = 0$ для любых $v \in V$.
Выход: маршрут коммивояжера, заданный массивом $mk[1..(n+1)]$, S — вес этого маршрута.

```
1.  begin
2.    Ostov(G);
3.    Eiler(H); i := 1;
4.    for v ∈ V  do  mark[v] := 0;
5.    while SRes ≠ ∅  do
6.      begin
7.        v ⇐ SRes;
8.        if mark[v] = 0 then
9.          begin
10.           mk[i] := v; i = i + 1; mark[v] := 1;
11.         end;
12.       end;
13.     S := 0; mk[n + 1] := mk[1];
14.     for i := 1 to n do  S := S + A[mk[i], mk[i + 1]];
15.   end.
```

Теорема 7.4. *Алгоритм Minspantreetravelling имеет сложность* $O(n^2)$.

Доказательство. Напомним, что минимальный остов можно построить за время $O(n^2)$, если использовать алгоритм Ярника-Прима-Дейкстры, следовательно, процедура $Ostov(G)$ может быть реализована со сложностью $O(n^2)$. Процедура $Eiler(H)$ имеет сложность $O(m)$,

где m — число ребер в графе H. Так как H получен из остова, то $m = 2(n-1)$. Поэтому процедура построения эйлерова маршрута здесь имеет сложность $O(n)$. Количество записей в списке $SRes$ пропорционально m. Следовательно, сложность цикла 5-10 есть $O(n)$. Отсюда и вытекает оценка сложности всего алгоритма. □

Вернемся к графу, приведенному ранее на рис. 92. Возьмем минимальное остовное дерево T этого графа, которое порождается ребрами v_1v_2, v_2v_3, v_1v_4. (В данном графе есть еще одно минимальное остовное дерево). Дальнейший результат зависит от того, каким будет эйлеров маршрут в графе H. Пусть, например, взят маршрут $v_1, v_4, v_1, v_2, v_3, v_2, v_1$. Тогда маршрут коммивояжера v_1, v_4, v_2, v_3, v_1, вписанный в этот эйлеров маршрут, имеет вес 29. Если в H взять эйлеров маршрут $v_1, v_2, v_3, v_2, v_1, v_4, v_1$, то получим оптимальный маршрут коммивояжера v_1, v_2, v_3, v_4, v_1, имеющий вес 25.

Отметим, что алгоритм $Minspantreetravelling$ имеет такую же оценку точности, как и алгоритм $Nearest\_insert$.

Теорема 7.5. *Пусть $G = (V, E, c)$ — полный взвешенный граф, матрица весов которого неотрицательна и удовлетворяет неравенству треугольника. Пусть $Mstt(G)$ — маршрут коммивояжера, построенный алгоритмом $Minspantreetravelling$, $Opt(G)$ — оптимальный маршрут, а $c(Mstt(G))$ и $c(Opt(G))$ — их веса. Тогда*

$$c(Mstt(G)) \leqslant 2 \cdot c(Opt(G)).$$

Доказательство. Пусть $cmst(G)$ — вес минимального остова графа G. Удаляя из $Opt(G)$ произвольное ребро, получим некоторый остов графа G. Отсюда следует, что

$$cmst(G) \leqslant c(Opt(G)). \qquad (*)$$

Так как граф H включает каждое ребро остова ровно два раза, то сумма весов эйлерова маршрута равна $2 \cdot cmst(G)$. Пусть $v_1, \ldots, v_n, v_1$ — последовательность вершин маршрута коммивояжера, вписанного в эйлеров маршрут. Тогда

$$c(Mstt(G)) = c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_n, v_1).$$

Для произвольных последовательных вершин v_k и v_{k+1} в маршруте коммивояжера, через $w_1, w_2, \ldots, w_r$ обозначим все вершины в эйлеровом

маршруте, стоящие между ними, включая их самих, т. е. $w_1 = v_k, w_r = v_{k+1}$. Из неравенства треугольника следует, что

$$c(v_k, v_{k+1}) \leqslant c(w_1, w_2) + \ldots + c(w_{r-1}, w_r).$$

Суммируя эти неравенства по всем $k = 1, \ldots, n$ и учитывая при этом, что вес эйлерова маршрута равен $2 \cdot cmst(G)$, получим

$$c(Mstt(G)) \leqslant 2 \cdot cmst(G).$$

Отсюда и из неравенства $(*)$ следует требуемый результат. □

В заключение отметим, что известны более точные быстрые алгоритмы построения маршрута коммивояжера, чем рассмотренные нами. Например, алгоритм Кристофидеса, разработанный в 1976 году, получает маршрут коммивояжера, вес которого может быть не более чем в 1,5 раза больше веса оптимального маршрута. Алгоритмы $Nearest\_insert$ и $Minspantreetravelling$ известны исследователям давно, однако оценка их точности (теорема 7.3 и теорема 7.5) получены сравнительно недавно — в 1977 году в работе Розенкратца, Штерна и Льюиса.

7.4. Решение задачи коммивояжера методом ветвей и границ

Для решения многих труднорешаемых задач относительно успешным является применение метода ветвей и границ. Мы продемонстрируем теперь применение этого метода к задаче коммивояжера. Будем по-прежнему предполагать, что граф $G = (V, E, c)$ является полным и задан матрицей весов. Однако, будем считать, что матрица весов не обязательно симметрична. Иначе говоря, будем считать, что граф G является ориентированным и взвешенным, т. е. является сетью.

Маршрутом коммивояжера в ориентированном графе называется контур, включающий каждую вершину ровно по одному разу. Для полной сети с n вершинами имеется $(n-1)!$ вариантов маршрута коммивояжера. Естественно, что полный перебор всех вариантов является невозможным даже для не очень больших значений n. Метод ветвей и границ является хорошим способом сокращения полного перебора для получения оптимального решения.

Представим процесс построения маршрута коммивояжера в виде построения двоичного корневого дерева решений, в котором каждой вершине x соответствует некоторое подмножество $M(x)$ множества всех

маршрутов коммивояжера. Считаем, что корню дерева решений поставлено в соответствие множество всех маршрутов коммивояжера.

Пусть x — некоторая вершина этого дерева. Выберем дугу vw, которая входит хотя бы в один маршрут из $M(x)$. Тогда множество $M(x)$ разбивается на два непересекающихся подмножества, в одно из которых можно отнести все маршруты, содержащие дугу vw, а в другое — не содержащие ее. Будем считать, что первое из этих подмножеств соответствует левому сыну вершины x, а второе — правому. Тем самым описано (пока еще в общих чертах) *правило ветвления*. Вершина дерева решений, для которой строятся сыновья, будет называться *активной*. Главное достоинство метода ветвей и границ в сравнении с полным перебором заключается в том, что активными объявляются лишь те вершины, в которых может содержаться оптимальный маршрут. Следовательно, необходимо выработать *правило активизации вершин*, которое будет сводиться к *правилу подсчета границ*.

Предположим, что для вершин дерева решений вычислено значение $f(x)$ такое, что вес любого маршрута из множества $M(x)$ не меньше чем $f(x)$. Такое число $f(x)$ называется *нижней границей* маршрутов множества $M(x)$ или, короче, границей вершины x. Правило активизации вершин заключается в том, что из множества вершин, не имеющих сыновей, в качестве активной выбирается вершина с наименьшей нижней границей. Вершина, для которой построены оба сына, активной стать в дальнейшем не может.

Процесс построения дерева решений продолжается до тех пор, пока активной не будет объявлена вершина x, для которой множество $M(x)$ состоит из одного единственного маршрута, а границы всех других вершин не меньше чем вес этого маршрута. Понятно, что тогда маршрут, содержащийся в $M(x)$, является оптимальным.

Остается сформулировать правила вычисления нижних границ. Процедуру вычитания из каждого элемента строки (соответственно столбца) минимального элемента этой же строки (столбца) назовем *редукцией строки (редукцией столбца)*. Процедуру, которая сначала осуществляет редукцию каждой строки, а затем в измененной матрице — редукцию всех столбцов, назовем *редукцией матрицы*, а полученную матрицу — *редуцированной*. Заметим, что редуцированная матрица неотрицательна, причем в каждой ее строке и каждом ее столбце имеется хотя бы один нулевой элемент.

Лемма 1. *Пусть P — маршрут коммивояжера в сети G, $c(P)$ (со-*

ответственно $d(P)$) — вес этого маршрута, определяемый матрицей весов сети G (редуцированной матрицей), f — сумма всех констант, используемых при редукции. Тогда $c(P) = d(P) + f$.

Доказательство. Для всякого маршрута P соответствующая последовательность весов дуг образует набор из n элементов матрицы весов. Этот набор характеризуется тем, что в каждой строке и каждом столбце матрицы содержится ровно по одному его элементу. Следовательно, каждый элемент набора модифицируется дважды. Сначала при редукции соответствующей строки, а затем — столбца. Кроме того, каждая константа, используемая при редукции, влияет ровно на один элемент набора. Отсюда следует заключение леммы. □

Поскольку редуцированная матрица содержит только неотрицательные элементы, имеем $d(P) \geqslant 0$. Следовательно, $c(P) \geqslant f$. Это означает, что величина f является нижней границей всех маршрутов коммивояжера для исходной нередуцированной матрицы весов. Тем самым получено правило вычисления нижней границы корневой вершины дерева решений.

Это правило остается справедливым и для любой другой вершины дерева решений. Действительно, с каждой вершиной x дерева решений однозначно связывается матрица, описывающая все возможные маршруты из $M(x)$. Сумма всех констант, используемых при ее редукции и нижняя граница отца вершины x дают нужную границу вершины x. А именно, справедлива

Лемма 2. *Пусть вершина x является сыном вершины y в дереве решений, $f(y)$ — нижняя граница вершины y и f — сумма всех констант, используемых при редукции матрицы, соответствующей вершине x. Тогда равенство $f(x) = f(y) + f$ задает нижнюю границу $f(x)$ вершины x.*

Изложенную схему метода ветвей и границ разберем на конкретном примере. Пусть сеть $G = (V, E, c)$ задана матрицей весов A, которая изображена на рис. 31 *a*). Редуцированная матрица A изображена на рис. 31 *b*). В столбце g матрицы A указаны минимальные элементы для каждой строки. После их вычитания из соответствующих строк нули будут в каждой строке и в первых четырех столбцах. Минимальные элементы для новых столбцов указаны в строке h матрицы 31 *a*). После редукции столбцов получается матрица 31 *b*). Общая сумма вычтенных элементов равна 32. Это число указано в пересечении строки h и столбца

g. Следовательно, любой маршрут коммивояжера в сети G имеет вес не меньше чем 32.

	1	2	3	4	5	g
1	∞	12	9	9	12	9
2	9	∞	8	19	15	8
3	6	0	∞	16	10	0
4	5	9	12	∞	16	5
5	15	7	13	23	∞	7
h	0	0	0	0	3	32

a)

	1	2	3	4	5	r
1	∞	3	0	0	0	0
2	1	∞	0	11	4	1
3	6	0	∞	16	7	6
4	0	4	7	∞	8	4
5	8	0	6	16	∞	6
s	1	0	0	11	4	

b)

Рис. 31

Перейдем теперь к постепенному построению корневого дерева поиска решения. Для ветвления в корневой вершине дерева решений нужно выбрать дугу vw и разбить все множество маршрутов на два непересекающихся подмножества, в одно из которых включаются все маршруты, содержащие дугу vw, в другое — не содержащие эту дугу. Правило выбора такой дуги vw определяет по существу всю стратегию поиска оптимального маршрута.

Выбор дуги vw означает, что маршруты из $M(x)$ для левого сына x не содержат других дуг, выходящих из v, и других дуг, входящих в w. Иначе говоря, из матрицы весов для левого сына можно удалить строку v и столбец w. Кроме того, следует запретить возможность включения в маршрут дуги wv, для чего достаточно положить $A[w, v] = \infty$. Таким образом, размер матрицы весов левого сына уменьшается на единицу, по сравнению с размером матрицы весов отца.

Поскольку маршруты из $M(x)$ для правого сына x запрещают лишь использовать дугу vw, то все изменения в матрице весов сводятся к тому, что нужно положить $A[v, w] = \infty$.

Поэтому предпочтительно находить решение, двигаясь по левым, а не по правым сыновьям. Следовательно, дуга vw должна выбираться так, чтобы нижняя граница правого сына была как можно больше границы левого сына. В редуцированной матрице из рис. 31 *b*) в столбце r (соответственно строке s) указаны вторые минимальные по порядку числа соответствующих строк (столбцов).

Правило выбора дуги vw можно сформулировать следующим образом: выбрать тот нуль в редуцированной матрице, для которого сумма

значений элементов, находящихся на пересечении строки v со столбцом r, и столбца w со строкой s, наибольшая.

Например, для нуля, находящегося в четвертой строке первого столбца, соответствующая сумма равна 5, а для нуля в первой строке и четвертом столбце — 11. Легко видеть, что это значение является наибольшим. Таким образом, разбиение множества всех маршрутов следует осуществить по дуге $(1, 4)$.

На рис. 32 *a*) и 32 *b*) изображены исходная и редуцированная матрицы весов левого сына корневой вершины, а на рис. 32 *c*) — редуцированная матрица правого сына. Редукция матрицы правого сына заключалась лишь в вычитании числа 11 из четвертого столбца.

	1	2	3	5	g
2	1	∞	0	4	0
3	6	0	∞	7	0
4	∞	4	7	8	4
5	8	0	6	∞	0
h	1	0	0	4	9

a)

	1	2	3	5	r
2	0	∞	0	0	0
3	5	0	∞	3	3
4	∞	0	3	0	0
5	7	0	6	∞	6
s	5	0	3	0	

b)

	1	2	3	4	5	r
1	∞	3	0	∞	0	0
2	1	∞	0	0	4	0
3	6	0	∞	5	7	5
4	0	4	7	∞	8	4
5	8	0	6	5	∞	5
s	1	0	0	5	4	

c)

Рис. 32

Для удобства через x_0 обозначим корень дерева решений, через x_{00} — его левого сына, а через x_{01} — правого. Если x_k — некоторая вершина дерева решений, где k — последовательность из нулей и единиц, то левый сын вершины x_k получает индекс приписыванием справа к k значения 0, а правый — 1. Вспоминая, что границу вершины x мы уже обозначали ранее через $f(x)$, получаем $f(x_0) = 32$, $f(x_{00}) = 41$, $f(x_{01}) = 43$.

Сформулированное правило выбора нуля в редуцированной матрице позволяет надеяться (но не гарантирует), что граница правого сына увеличится больше всего. Отметим, что имеются и более сильные правила выбора дуги vw или, что тоже самое, нуля в редуцированной матрице.

После построения вершин x_{00} и x_{01} дерево решений выглядит так, как оно изображено на рис. 33, где в скобках рядом с именем вершины написано значение нижней границы. Ниже вершины дерева решений написан соответствующий вариант разбиения множества маршрутов, соответствующих данной вершине, а именно, указана дуга, по которой производится разбиение.

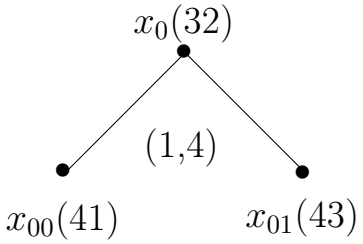


Рис. 33

Поскольку вершина x_{00} имеет границу меньше чем вершина x_{01}, теперь она становится активной. По описанному ранее правилу выбираем дугу $(5, 2)$, ибо нуль, стоящий в строке 5 и столбце 2, имеет наибольшую сумму соответствующих ему значений в строке s и столбце r (см. рис. 32 *b*)).

	1	3	5	g
2	0	0	∞	0
3	5	∞	3	3
4	∞	3	0	0
h	0	0	0	3

a)

	1	3	5	r
2	0	0	∞	0
3	2	∞	0	2
4	∞	3	0	3
s	2	3	0	

b)

	1	2	3	5	r
2	0	∞	0	0	0
3	5	0	∞	3	3
4	∞	0	3	0	0
5	1	∞	0	∞	1
s	1	0	0	0	

c)

Рис. 34

Матрицы вершин x_{000} и x_{001} изображены на рис. 34, где на рис. 34 *a*) и 34 *b*) изображены соответственно исходная и редуцированная матрицы вершины x_{000}, а на 34 *c*) — редуцированная матрица вершины x_{001}. Редукция матрицы x_{001} состояла лишь в вычитании числа 6 из последней

строки. Для нижних границ имеем равенства $f(x_{000}) = 44, f(x_{001}) = 47$. Процесс построения дерева решений рекомендуем далее отслеживать по рис. 34.

	1	3	4	5	g
1	∞	0	∞	0	0
2	1	∞	0	4	0
4	0	7	∞	8	0
5	8	6	5	∞	5
h	0	0	0	0	5

a)

	1	2	3	4	5	g
1	∞	3	0	∞	0	0
2	1	∞	0	0	4	0
3	6	∞	∞	5	7	5
4	0	4	7	∞	8	0
5	8	0	6	5	∞	0
h	0	0	0	0	0	5

b)

Рис. 35

Теперь придется вернуться назад по дереву решений, ибо активной становится вершина x_{01}. В матрице, изображенной на рис. 32 *c*), сразу несколько нулей имеют одинаковую сумму соответствующих элементов в столбцах s и r. Выберем для разбиения дугу $(3, 2)$. Нередуцированные матрицы сыновей вершины x_{01} изображены на рис. 33. Анализируя строки h и столбцы g обеих матриц, получаем $f(x_{010}) = 48$ и $f(x_{011}) = 48$.

Следующей активной вершиной становится вершина x_{000}, редуцированная матрица которой изображена на рис. 34 *b*). В этой матрице два равноценных претендента: дуга $(2, 3)$ и дуга$(4, 5)$. Выберем для разбиения дугу $(4, 5)$. Получающиеся матрицы вершин x_{0000} и x_{0001} изображены на рис. 36 *a*) и 36 *b*) соответственно. Обращаем внимание читателя на то, что в матрице 36 *a*) элемент $(2, 1)$ равен ∞. Дело в том, что все маршруты в $M(x_{0000})$ содержат дуги $(1, 4), (5, 2)$ и $(4, 5)$. Никакой маршрут коммивояжера теперь не может содержать дугу $(2, 1)$. По этой причине элемент $(2, 1)$ в матрице приравнен к ∞.

	1	3
2	∞	0
3	0	∞

a)

	1	3	5
2	0	0	∞
3	2	∞	0
4	∞	3	∞

b)

Рис. 36

На рис. 36 *a*) изображена редуцированная матрица вершины x_{0000}. Ясно, что $f(x_{0000}) = f(x_{000}) + 2 = 46$. Для матрицы 36 *b*) редукция сводится к вычитанию числа 3 из строки 4. Отсюда получаем равенство $f(x_{0001}) = 47$.

Наконец, активной становится вершина x_{0000}. Здесь выбор дуги не играет никакой роли. Есть только две возможности. Или сначала выбрать дугу (2,3) и, проводя разбиение по ней, на следующем шаге выбрать (3,1), или наоборот, сначала — (3,1), а затем — (2,3). Важным является лишь то, что при переходе к сыновьям границы левых сыновей не меняются, поэтому левые сыновья сразу становятся активными. Границы правых сыновей и в том, и в другом случае равны ∞, что означает отсутствие соответствующих маршрутов коммивояжера.

Дерево решений, построенное нами, изображено на рис. 37. Поскольку активной теперь следует объявить вершину, содержащую один единственный маршрут, алгоритм на этом заканчивает свою работу.

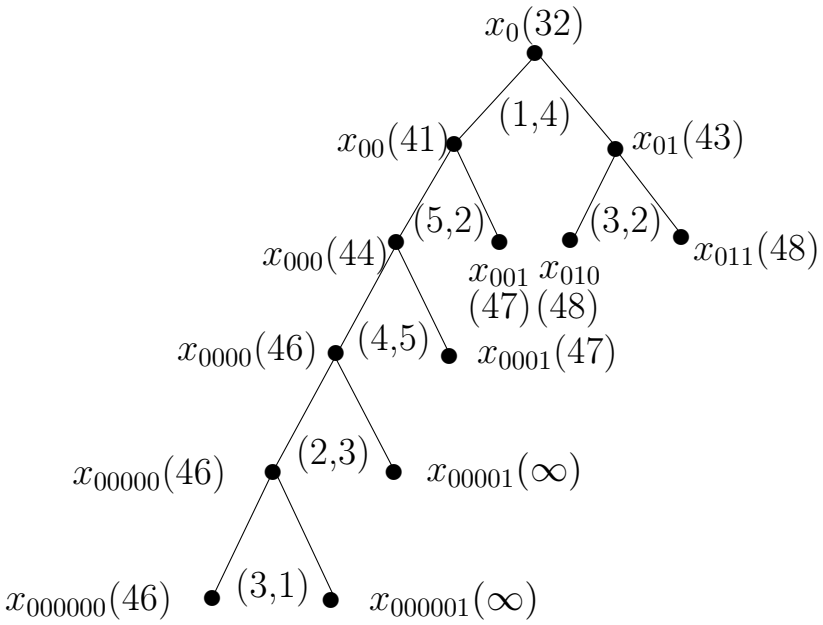


Рис. 37

Итак, оптимальным в заданном графе является маршрут 1-4-5-2-3-1, вес которого равен 46. Отметим, что в данном примере мы исследовали всего 13 вершин дерева решений. Имеющиеся экспериментальные данные позволяют утверждать, что метод ветвей и границ для ЗК является разумной альтернативой полному перебору, ибо для случайной матрицы весов число исследуемых вершин дерева решений равно $O((1,26)^n)$.

Отметим особенности ЗК, позволившие реализовать метод ветвей и границ для ее решения.

1. *Ветвление.* Множество решений, представляемое вершинами дерева решений, можно разбить на попарно непересекающиеся множества. Каждое подмножество в этом разбиении представляется сыном исходной вершины в дереве решений.

2. *Границы.* Имеется алгоритм для вычисления нижней границы веса любого решения в данном подмножестве.

Поскольку многие задачи дискретной оптимизации обладают двумя перечисленными свойствами, то метод ветвей и границ может применяться для их решения. При этом разбиение множества решений на каждом шаге может осуществляться и более чем на два непересекающихся подмножества, если для подмножеств имеется подходящий алгоритм для вычисления нижней границы входящих в них решений.

Список литературы

[1] *Абрахамс Д., Каверли Д.* Анализ электрических сетей методом графов.– М.: Мир, 1967.

[2] *Адельсон-Вельский Г.М., Диниц Е.А., Карзанов А.В.* Потоковые алгоритмы.– М.: Наука, 1975.

[3] *Айгнер М.* Комбинаторная теория.– М.: Мир, 1982.

[4] *Андерсон Дж. А.* Дискретная математика и комбинаторика.– М.: Изд. дом «Вильямс», 2003.

[5] *Асанов М.О.* Дискретная оптимизация.– Екатеринбург: УралНАУКА, 1998.

[6] *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов.– М.: Мир, 1979.

[7] *Ахо А., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы.– М.: Изд. дом "Вильямс 2000.

[8] *Басакер Р., Саати Т.* Конечные графы и сети.– М.: Наука, 1974.

[9] *Белов В.В., Воробьев Е.М., Шаталов В.Е.* Теория графов.– М.: Высш. шк., 1976.

[10] *Берж К.* Теория графов и ее применения.– М.: ИЛ, 1962.

[11] *Биркгоф Г.* Теория структур. – М:, Наука, 1984.

[12] *Болл У., Коксетер Г.* Математические эссе и развлечения. – М.: Мир, 1986.

[13] *Вирт Н.* Алгоритмы + структуры данных = программы.– М.: Мир, 1985.

[14] *Гантмахер Ф.Р.* Теория матриц.– М.: Наука, 1966.

[15] *Гретцер Г.* Общая теория решеток.– М.: Мир, 1982.

[16] *Грин Д., Кнут Д.* Математические методы анализа алгоритмов.– М.: Мир, 1987.

[17] *Гудман С., Хидетниеми С.* Введение в разработку и анализ алгоритмов. – М.: Мир, 1981.

[18] *Гэри М., Джонсон Д.* Вычислительные машины и труднорешаемые задачи.– М.: Мир, 1982.

[19] *Евстигнеев В.А.* Применение теории графов в программировании.– М.: Наука, 1985.

[20] *Евстигнеев В.А., Касьянов В.Н.* Теория графов: алгоритмы обработки деревьев.– Новосибирск: Наука, 1994.

[21] *Евстигнеев В.А., Касьянов В.Н.* Теория графов: алгоритмы обработки бесконтурных графов.– Новосибирск: Наука. Сиб. предприятие РАН, 1998.

[22] *Евстигнеев В.А., Мельников Л.С.* Задачи и упражнения по теории графов и комбинаторике.– Новосибирск: Издательство НГУ, 1981.

[23] *Емеличев В.А, Ковалев М.М., Кравцов М.К.* Многогранники, графы, оптимизация.– М.: Наука, 1981.

[24] *Емеличев В.А., Мельников О.И., Сарванов В.И., Тышкевич Р.И.* Лекции по теории графов.– М.: Наука, 1990.

[25] *Замбицкий Д.К., Лозовану Д.Д.* Алгоритмы решения оптимизационных задач на сетях.– Кишинев: Штиинца, 1983.

[26] *Зыков А.А.* Теория конечных графов.– Новосибирск: Наука, 1969.

[27] *Зыков А.А.* Основы теории графов.– М.: Наука, 1987.

[28] *Камерон П.Дж., ван Линт Дж.Х.* Теория графов, теория кодирования и блок-схемы.– М.: Наука, 1980.

[29] *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ.– М.: МЦНМО, 1999.

[30] *Кнут Д.* Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы.– М.: Мир, 1976; перераб. издание: М.: Изд. дом «Вильямс», 2000.

[31] *Кнут Д.* Искусство программирования для ЭВМ. Т. 2. Получисленные алгоритмы.– М.: Мир, 1977; перераб. издание: М.: Изд. дом «Вильямс», 2000.

[32] *Кнут Д.* Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск.– М.: Мир, 1978; перераб. издание: М.: Изд. дом «Вильямс», 2000.

[33] *Кристофидес Н.* Теория графов. Алгоритмический подход.– М.: Мир, 1978.

[34] *Кофман А.* Введение в прикладную комбинаторику.– М.: Наука, 1975.

[35] *Кук В., Бейз Г.* Компьютерная математика.– М.: Наука, 1990.

[36] *Липский В.* Комбинаторика для программистов.– М.: Мир, 1988.

[37] *Ловас Л., Пламмер М.* Прикладные залачи теории графов. Теория паросочетаний в математике, физике, химии.– М.: Мир, 1998.

[38] *Майника Э.* Алгоритмы оптимизации на сетях и графах.– М.: Мир, 1981.

[39] *Миркин Б.Г., Родин С.Н.* Графы и гены.– М.: Наука, 1977.

[40] *Новиков Ф.А.* Дискретная математика для программистов.– СПб.: Питер, 2000.

[41] *Окулов С.М., Пестов А.А., Пестов О.А.* Информатика в задачах.– Киров: Вятский гос. пед. ун-т, 1998.

[42] *Оре О.* Графы и их применение.– М.: Мир, 1965.

[43] *Оре О.* Теория графов.– М.: Наука, 1980.

[44] *Пападимитриу Х., Стайглиц К.* Комбинаторная оптимизация. Алгоритмы и сложность.– М.: Мир, 1985.

[45] *Папи Ф., Папи Ж.* Дети и графы. – М:, Педагогика, 1974.

[46] *Рейнгольд Э., Нивергельт Ю., Део Н.* Комбинаторные алгоритмы. Теория и практика.– М.: Мир, 1980.

[47] *Рингель Г.* Теорема о раскраске карт.– М.: Мир. 1977.

[48] *Сачков В.Н.* Введение в комбинаторные методы дискретной математики.– М.: Наука, 1982.

[49] *Свами М., Тхуласираман К.* Графы, сети и алгоритмы.– М.: Мир, 1984.

[50] *Сушков Ю.А.* Графы зубчатых механизмов.– Ленинград: Машиностроение, 1983.

[51] *Тамт У.* Теория графов.– М.: Мир, 1988.

[52] *Уилсон Р.* Введение в теорию графов.– М.: Мир, 1977.

[53] *Филлипс Д., Гарсиа-Диас А.* Методы анализа сетей.– М.: Мир, 1984.

[54] *Форд Л.Р., Фалкерсон Д.Р.* Потоки в сетях.– М.: Мир, 1966.

[55] *Харари Ф.* Теория графов.– М.: Мир, 1973.

[56] *Харари Ф., Палмер Э.* Перечисления графов.– М.: Мир, 1977.

[57] *Холл М.* Комбинаторика.– М.: Мир, 1970.

[58] *Цветкович Д., Дуб М., Захс Х.* Спектры графов. Теория и приложения.– Киев: Наукова думка, 1984.

[59] Handbook of Theoretical Computer Science. Vol.A: Algorithms and Complexity Theories.– North Holland Publ. Comp., Amsterdam, 1990.

[60] *Read R.C.* An Introduction to Chromatic Polynomials. J. of Comb. Theory, 4, 1968, p. 52-71.

[61] *Tarjan R.E.* Data Structures and Network Algorithms.– Soc. for Idustr. and Applied Math., Philadelphia, Pensylvania, 1983.

[62] *Truemper K.* Matroid Decomposition (Revised Edition).– Leibniz, Plano, Texas, 1998.

[63] *Welsh D.J.A.* Matroid Theory.– New York Acad. Press, 1976.

ОГЛАВЛЕНИЕ

Предисловие **3**

1. Введение в алгоритмы **4**
1.1. Алгоритмы и их сложность 5
1.2. Запись алгоритмов 7
1.3. Корневые и бинарные деревья 9
1.4. Сортировка массивов 13

2. Поиск в графе **19**
2.1. Поиск в глубину 19
2.2. Алгоритм отыскания блоков и точек сочленения 23
2.3. Алгоритм отыскания компонент сильной связности в орграфе 28
2.4. Поиск в ширину 34
2.5. Алгоритм отыскания эйлеровой цепи в эйлеровом графе 38

3. Задача о минимальном остове **42**

4. Пути в сетях **50**
4.1. Постановка задачи 50
4.2. Общий случай. Алгоритм Форда-Беллмана 50
4.3. Случай неотрицательных весов. Алгоритм Дейкстры . . 55
4.4. Случай бесконтурной сети 59
4.5. Задача о максимальном пути и сетевые графики 64
4.6. Задача о maxmin-пути 70
4.7. Задача о кратчайших путях между всеми парами вершин 74

5. Задача о максимальном потоке **77**
5.1. Основные понятия и результаты 77
5.2. Алгоритм Форда-Фалкерсона 84

6. Паросочетания в двудольных графах **92**
6.1. Основные понятия 92
6.2. Задача о наибольшем паросочетании. Алгоритм Хопкрофта-Карпа 93
6.3. Задача о полном паросочетании. Алгоритм Куна 110

6.4. Задача о назначениях. Венгерский алгоритм 116

7. Задача коммивояжера **126**
7.1. Основные понятия 126
7.2. Алгоритм отыскания гамильтоновых циклов 127
7.3. Алгоритмы решения задачи коммивояжера с гарантированной оценкой точности 129
7.4. Решение задачи коммивояжера методом ветвей и границ 138

Список литературы **147**