Федеральное государственное бюджетное
образовательное
учреждение высшего профессионального образования
"Тверской государственный университет"
Кафедра информатики

М.И. Дехтярь

АЛГОРИТМИЧЕСКИЕ ЗАДАЧИ НА ГРАФАХ

Учебное пособие

Тверь — 2012

Содержание

Введение

Мы часто сталкиваемся с задачами, в условиях которых заданы некоторые объекты и между некоторыми их парами имеются определенные связи. Если объекты изобразить точками (вершинами), а связи — линиями (ребрами), соединяющими соответствующие пары точек, то получится рисунок, называемый *графом.* Историю теории графов принято исчислять с 1736 г., когда Эйлер исследовал "задачу о кенигсберских мостах": построить в графе циклический путь, проходящий по одному разу через каждое ребро. В середине 19-го века Гамильтон заинтересовался задачей построения циклического пути, проходящего по одному разу через каждую вершину графа[1]. К тому же времени относится использование графов для анализа электрических цепей (Кирхгоф) и химических молекул (Кэли). Начало развития современной теории графов относится к 30-м годам 20-го столетия. Графы нашли многочисленные применения в электротехнике, электронике, биологии, экономике, программировании и в других областях.

Настоящее пособие написано по материалам курсов "Проектирование эффективных алгоритмов"и "Алгоритмы на графах", которые автор неоднократно в 1990-2012гг. читал в Тверском государственном университете. Основное внимание уделено рассмотрению широко используемых в различных приложениях типовых алгоритмических задач, таких, как представления графов, отношение достижимости и транзитивное замыкание графа, компоненты сильной связности ориентированного графа и его базы, деревья и их обходы, минимальные остовные деревья, поиск в глубину, поиск кратчайших путей и потоки в транспортных сетях. Рассмотрены также многие сложные (NP-полные) задачи на графах и возможные подходы к их решению. В последней главе продемонстрировано применение теории графов к анализу социальных сетей. Каждая глава завершается разделом с задачами, которые могут служить материалом для проведения практических занятий и домашних заданий. Ряд задач содержат дополнительный материал, по тем или иным причинам не вошедший в основной текст. Список литературы заведомо неполон, он включает лишь учебники, монографии и статьи, непосредственно использованные при написании этого пособия. В них можно найти дальнейшие ссылки на многочисленные источники, посвященные алгоритмам на графах.

1 Основные понятия

Приведем основные определения теории графов.

Определение 1. Ориентированный граф – *это пара* (V, E)*, где* V *– конечное множество* **вершин** *(узлов, точек) графа, а* E *– некоторое множество пар вершин, т.е. подмножество множества* $V \times V$ *или бинарное отношение на* V*. Элементы* E *называют* **ребрами** *(дугами, стрелками, связями). Для ребра* $e = (u, v) \in E$ *вершина* u *называется началом* e*, а вершина* v *– концом* e*, говорят, что ребро* e *ведет из* u *в* v*.*

Неориентированный граф $G = (V, E)$ *– это ориентированный граф, у которого для каждого ребра* $(u, v) \in E$ *имеется противоположное ребро* $(v, u) \in E$*, т.е. отношение* E *симметрично. Такая пара* $(u, v), (v, u)$ *называется неориентированным ребром. Для его задания можно использовать обозначение для множества концов:* $\{u, v\}$*, но чаще используется указание одной из пар в круглых скобках. Если* $e = (u, v) \in E$*, то вершины* u *и* v *называются смежными в* G*, а ребро* e *и эти вершины называются инцидентными. Степенью вершины в неориентированном графе называется число смежных с ней вершин. Вершина степени 0 называется изолированной.*
В ориентированном графе полустепень исхода вершины — это число исходящих из нее ребер, а полустепень захода — это число входящих в данную вершину ребер.

[1] Интересно, что несмотря на внешнюю похожесть задача Эйлера имеет простое эффективное решение (см. задачу 2.13), а задача Гамильтона в общем случае эффективно не решается (см. раздел 8.4)

Заметим, что в ориентированном графе может быть ребро вида (u, u), называемое *петлей*, а в неориентированном петель не бывает.

Пример 1. *На рис. 1 приведены примеры ориентированного графа $G_1 = (V_1, E_1)$ и неориентированного графа $G_2 = (V_2, E_2)$. Здесь $V_1 = \{a, b, c, d\}, E_1 = \{(a, b), (a, c), (b, b), (b, d), (d, a)\}$, $V_2 = \{a, b, c, d\}, E_2 = \{(a, b), (a, c), (a, d), (b, d)\}$. В графе G_1 ребро (b, b) является петлей, полустепень исхода вершины a равна 2, а полустепень захода для нее равна 1. В графе G_2 степень вершины a равна 3, вершин b и d — 2, вершины c — 1, а вершины e — 0, т.е. вершина e является изолированной,*

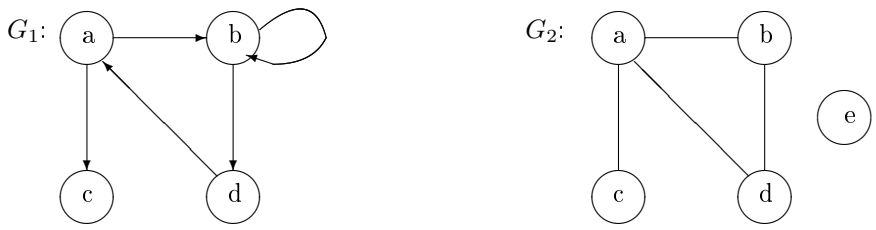


Рис. 1: Ориентированный граф G_1 и неориентированный граф G_2

Части графов называются *подграфами.*

Определение 2. *Граф $G_1 = (V_1, E_1)$ называется* **подграфом** *графа $G = (V, E)$, если $V_1 \subseteq V$ и $E_1 \subseteq E$.*

Во многих приложениях с вершинами и ребрами графов связывается некоторая дополнительная информация. Обычно она представляется с помощью функций разметки вершин и ребер.

Определение 3. Размеченный *граф — это ориентированный или неориентированный граф $G = (V, E)$, снабженный одной или двумя функциями разметки вида: $l : V \to M$ и $c : E \to L$, где M и L — множества меток вершин и ребер, соответственно.*

Упорядоченный *граф — это размеченный граф $G = (V, E)$, в котором ребра, выходящие из каждой вершины $v \in V$, упорядочены, т.е. помечены номерами $1, \ldots, k_v$, где k_v — полустепень исхода v, т.е. $k_v = |\{w \mid (v, w) \in E\}|$.*

В качестве множества меток ребер L часто выступают числа, задающие “веса”, “длины”, “стоимости” ребер. Графы с такой разметкой часто называют *взвешенными.*

Во многих задачах, использующих графы с разметкой ребер, удобно допускать несколько ребер между одной парой вершин.

Определение 4. Ориентированный мультиграф *– это пара (V, E), где V – конечное множество* **вершин** *(узлов, точек) графа, а множество ребер (дуг) E – это некоторое* **мультимножество** *пар вершин, т.е. множество, в которое каждый элемент (ребро) может входить несколько раз.*

Обычно "параллельные"ребра в размеченных мультиграфах различаются своими метками.

Неориентированный граф $G = (V, E)$ называется *полным*, если $E = V \times V \setminus \{(v, v) \mid v \in V\}$, т.е. любые две его вершины соединены ребром. Полный n-вершинный граф часто обозначается как K_n. На следующем рисунке показаны первые пять полных графов.

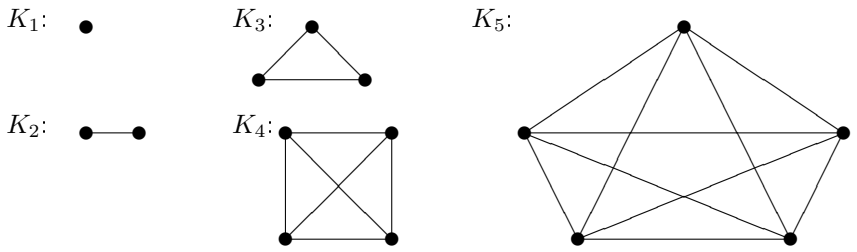


Рис. 2: Полные графы $K_1 - K_5$

Во многих случаях естественно не различать графы, отличающиеся лишь именами (порядком) вершин.

Определение 5. Изоморфизм графов. *Два графа $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$ называются изоморфными, если между их вершинами существует взаимно однозначное соответствие $\varphi : V_1 \to V_2$ такое, что для любой пары вершин u, v из V_1 ребро $(u, v) \in E_1 \Leftrightarrow$ ребро $(\varphi(u), \varphi(v)) \in E_2$.*

Для изоморфизма размеченных графов требуется также совпадение меток соответствующих вершин : $l_1(v) = l_2(\varphi(v))$ и/или ребер: $c_1((u, v)) = c_2((\varphi(u), \varphi(v)))$.

Многие приложения графов связаны с изучением путей между их вершинами.

Определение 6. Путь *в ориентированном или неориентированном графе — это последовательность ребер вида $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$. Этот путь ведет из начальной вершины v_1 в конечную вершину v_n и имеет длину $n - 1$. В этом случае будем говорить, что v_n* **достижима** *из v_1. Будем считать, что каждая вершина достижима сама из себя путем длины 0. Путь можно также определять как соответствующую последовательность вершин: $(v_1, v_2, v_3, \ldots, v_{n-1}, v_n)$, где $(v_i, v_{i+1}) \in E$ при $i = 1, 2, \ldots, n - 1$.*

Путь называется **простым**, *если все ребра и все вершины на нем, кроме, быть может, первой и последней, различны.*

Циклом *в ориентированном графе называется путь, в котором начальная вершина совпадает с конечной и который содержит хотя бы одно ребро. Цикл $(v_1, v_2, \ldots, v_{n-1}, v_n = v_1)$ называется простым, если в нем нет одинаковых вершин, кроме первой и последней, т.е. если все вершины $v_2, \ldots, v_{n-1}$ различны.*

В неориентированном графе путь $(v_1, v_2, \ldots, v_{n-1}, v_n = v_1)$ называется **циклом**, *если $n \geq 4$ и все ребра (v_i, v_{i+1}) различны.*

Из последнего определения следует, что длина цикла в неориентированном графе не меньше 3.

Если в графе нет циклов, то он называется **ациклическим.**

Неориентированный граф называется **связным,** *если любая пара вершин в нем соединена путем. При выполнении такого же условия ориентированный граф называется* **сильно связным** *или* **двусвязным**

Максимальный связный подграф неориентированного графа называется его **связной компонентой**. *Максимальный сильно связный подграф ориентированного графа называется его* **сильно связной компонентой**.

Следующее утверждение непосредственно следует из определений.

Лемма 1.1. *Если в графе G (ориентированном или неориентированном) имеется путь из вершины u в вершину v, то в нем имеется и простой путь из u в v.*

2 Представления графов

Из определения графа следует, что каждый граф $G = (V, E)$ можно задать, непосредственно перечислив его множество вершин V и множество ребер E. Однако такое представление неудобно для решения многих задач о графах. Например, чтобы проверить наличие ребра между двумя вершинами, придется, вообще говоря, просмотреть все множество E. Хорошее представление, по крайней мере, должно позволить легко переходить от вершины к ее соседу и перечислять всех ее соседей.

В этом параграфе мы рассмотрим три разных способа представления графов, которые более эффективны при решении типичных для теории графов задач.

2.1 Матрица (таблица) смежности

Определение 7. *Матрицей смежности ориентированного (или неориентированного) графа $G = (V, E)$ с n вершинами $V = \{v_1, \ldots, v_n\}$ называется булева матрица A_G размера $n \times n$ с элементами*

$$a_{ij} = \begin{cases} 1, & \text{если } (v_i, v_j) \in E \\ 0 & \text{в противном случае} \end{cases}$$

Это представление позволяет легко проверять наличие ребер между заданными парами вершин. Для поиска всех соседей, в которые ведут ребра из вершины v_i, необходимо просмотреть соответствующую ей i-ю строку матрицы A_G, а чтобы найти вершины, из которых ребра идут в v_i, необходимо просмотреть ее i-ый столбец. Требуемая для A_G память – по порядку n^2 битов – не может быть уменьшена для графов, у которых "много" ребер. Но для разреженных графов с числом ребер существенно меньшим по порядку n^2 в матрице смежности много "ненужных" нулей. Для таких графов более эффективными могут оказаться другие представления.

2.2 Матрица (таблица) инцидентности

Определение 8. *Матрицей инцидентности ориентированного (или неориентированного) графа $G = (V, E)$ с n вершинами $V = \{v_1, \ldots, v_n\}$ и m ребрами $E = \{e_1, \ldots, e_m\}$ называется матрица B_G размера $n \times m$ с элементами*

$$b_{ij} = \begin{cases} 1, & \text{если для некоторого } k \text{ ребро } e_j = (v_i, v_k), \\ -1, & \text{если для некоторого } k \text{ ребро } e_j = (v_k, v_i), \\ 2, & \text{если ребро } e_j = (v_i, v_i), \\ 0, & \text{в остальных случаях.} \end{cases}$$

Таким образом, в матрице инцидентности B_G любому ребру $e_j = (v_i, v_k)$ соответствует j-ый столбец, в котором в i-ой строке стоит 1, а в k-ой – -1. Ребра-петли выделяются числом 2. Для проверки наличия ребра между двумя вершинами v_i и v_k требуется просмотреть i-ю и k-ую строки B_G, поиск всех соседей вершины требует просмотра соответствующей строки. Если $m >> n$, то это требует существенно больше времени, чем при использовании матрицы смежности. Поэтому при практическом решении задач на графах матрица инцидентности почти не используется.

2.3 Списки смежности

Определение 9. *Пусть $G = (V, E)$ — ориентированный граф, v — вершина из V.* **Список смежности** *L_v для вершины v включает все смежные с ней вершины, т.е.*
$L_v = w_1, \ldots, w_k$, *где* $\{w_1, \ldots, w_k\} = \{w \mid (v, w) \in E\}$.

Представление графа $G = (V, E)$ с n вершинами $V = \{v_1, \ldots, v_n\}$ с помощью списков смежности состоит из списков смежности всех вершин: $L_{v_1}, L_{v_2}, \ldots, L_{v_n}$.

Размер этого представления сравним с суммой числа вершин и ребер графа. Оно позволяет легко переходить по ребрам от вершины к ее соседям. В программах списки смежности представляются массивами или списковыми структурами, которые легко реализуются во всех языках программирования.

Рассмотрим некоторые представления списков смежности для графов с ограниченной полустепенью исхода и для произвольных графов.

2.3.1 Графы с ограниченной полустепенью исхода

Пусть $G = (V, E)$ – граф с n вершинами $V = \{v_1, \ldots, v_n\}$, в котором полустепень исхода у каждой вершины ограничена числом t, т.е. для каждой вершины v_i ее список смежности $L_{v_i} = v_{j_1}, \ldots, v_{j_{k(i)}}$ имеет длину $k(i) \leq t$.

1)Такой граф G можно представить с помощью t одномерных целочисленных массивов СОСЕД1, СОСЕД2, ..., СОСЕД t длины n каждый, таких что СОСЕДi$[r] = j_r$ для всех $1 \leq i \leq n$ и $1 \leq r \leq k(i)$. При $r > k(i)$ можно положить СОСЕДi$[r] = 0$. Для некоторых задач также полезно хранить для каждой вершины информацию о числе соседей, например, в массиве СОСЕДИ размера n: СОСЕДИ$[i] = k(i)$. Это представление достаточно компактно и эффективно, если степени исхода большинства вершин равны t или близки к этому числу и в процессе работы с графом он не изменяется, в частности, не меняется множество его вершин, т.е. не требуется динамически перестраивать массивы.

2) Если же при работе с графом он может сильно изменяться, то более эффективным является представление с помощью списочной структуры, элементы которой являются записями типа *ВЕРШИНА = (Номер, ЧислоСоседей, Сосед1, ..., Соседt)*, где для элемента, представляющего вершину v_i поле *Номер* равно i, поле *ЧислоСоседей* содержит $k(i)$, и каждое поле *Соседr* $(1 \leq r \leq k(i))$ содержит ссылку на элемент, представляющий вершину v_{j_r}. Такое представление позволяет производить локальные изменения графа (удаление и вставку вершины) за время $O(1)$.

2.3.2 Произвольные графы

Пусть $G = (V, E)$ – граф с n вершинами $V = \{v_1, \ldots, v_n\}$ и полустепени исхода у вершин достаточно различны, например имеется несколько вершин с большим числом соседей, а у остальных их немного. В таком случае граф можно представить с помощью списковой структуры следующего вида:
все вершины графа связаны в линейный список с элементами-записями типа *ВЕРШИНА = (Номер, Соседи, Следующая)*. Здесь поле *Следующая* содержит ссылку на следующую вершину, а поле *Соседи* содержит ссылку на начало списка соседей данной вершины. Список соседей состоит из элементов-записей типа *СОСЕД = (Сосед, Следующий_ Сосед)*, в которых *Сосед* — это ссылка на элемент в списке вершин, являющийся соседом данной вершины, а *Следующий_ Сосед* указывает на следующий элемент в списке соседей.

Пример 2. *Рассмотрим следующий граф $G = (V, E)$:*
$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$,

$E = \{e_1 = (v_1, v_3), e_2 = (v_3, v_2), e_3 = (v_4, v_1), e_4 = (v_4, v_5), e_5 = (v_5, v_3), e_6 = (v_5, v_6), e_7 = (v_6, v_6), e_8 = (v_2, v_1)\}$. *Он показан на рис.3.*

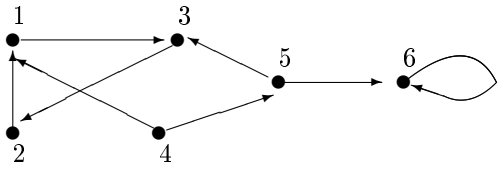


Рис. 3: Граф G

Построим для него определенные выше представления.
1) Матрица смежности.

$$A_G = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

2) Матрица инцидентности.

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

3) Списки смежности.

$$\begin{array}{ll} L_{v_1} : v_3; & L_{v_4} : v_1, v_5; \\ L_{v_2} : v_1; & L_{v_5} : v_3, v_6; \\ L_{v_3} : v_2; & L_{v_6} : v_6. \end{array}$$

4) Реализация списков смежности с помощью массивов.
Заметим, что полустепень исхода у вершин графа G не превосходит 2. Поэтому он может быть представлен с помощью двух массивов СОСЕДИ1 и СОСЕДИ2 как это показано в следующей таблице.

Вершины:	1	2	3	4	5	6
СОСЕДИ1	3	1	2	1	3	6
СОСЕДИ2	0	0	0	5	6	0

Таблица 1: Представление графа G с помощью массивов

5) Реализация списков смежности с помощью списков при фиксированном числе соседей показана на рис. 4.

6) При сильно различающихся степенях вершин граф представляется с помощью списков двух типов: ВЕРШИНА и СОСЕД. Это представление G показано на рис. 5.

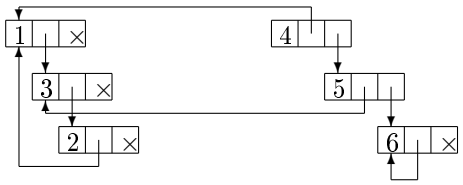


Рис. 4: Представление графа G с помощью списков (полустепень исхода $t \leq 2$)

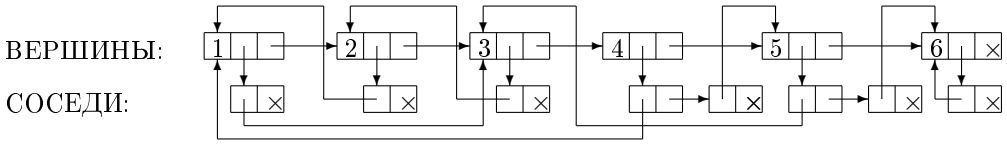


Рис. 5: Общее представление графа G с помощью списков типа ВЕРШИНА и СОСЕД

2.4 Задачи

Задача 2.1. *Докажите, что во всяком ациклическом ориентированном графе имеется хотя бы одна вершина, из которой не выходят ребра, т.е. с полустепенью исхода 0, и хотя бы одна вершина, в которую не входят ребра, т.е. с полустепенью захода 0.*

Задача 2.2. *Докажите, что, если полустепени захода у всех вершин ориентированного графа больше 0, то в этом графе имеется цикл.*

Задача 2.3. *Докажите, что сумма степеней всех вершин произвольного неориентированного графа четна.*

У этой задачи имеется популярная интерпретация: доказать, что общее число рукопожатий, которыми обменялись люди, пришедшие на вечеринку, всегда четно.

Задача 2.4. *Докажите, что в неориентированном графе число вершин нечетной степени всегда четно.*

Задача 2.5. *Перечислите все неизоморфные неориентированные графы, у которых не более четырех вершин.*

Задача 2.6. *Докажите, что в любой группе из 6 человек есть трое попарно знакомых или трое попарно незнакомых.*

Задача 2.7. *Докажите, что неориентированный связный граф остается связным после удаления некоторого ребра $\leftrightarrow$ это ребро принадлежит некоторому циклу.*

Задача 2.8. *Докажите, что неориентированный связный граф с n вершинами*
а) содержит не менее $n-1$ ребер,
б) если содержит больше $n-1$ ребер, то имеет по крайней мере хоть один цикл.

Задача 2.9. *Докажите, что, если в неориентированном графе (без петель) имеется ровно две вершины нечетной степени, то они связаны путем.*

Задача 2.10. *Докажите, что неориентированный граф $G = (V, E)$ связен $\leftrightarrow$ для каждого разбиения $V = V_1 \cup V_2$ с непустыми V_1 и V_2 существует ребро, соединяющее V_1 с V_2.*

Задача 2.11. *Докажите, что в связном неориентированном графе любые два простых пути максимальной длины имеют общую вершину.*

Задача 2.12. *Предложите алгоритмы для следующих преобразований представлений графа $G = (V, E)$ с n вершинами и m ребрами и оцените их сложность.*

1) Преобразовать матрицу смежности в списки смежности.

2) Преобразовать списки смежности в матрицу смежности.

3) Преобразовать списки смежности в матрицу инцидентности.

4) Преобразовать матрицу инцидентности в списки смежности.

Задача 2.13. *Цикл в связном неориентированном графе называется* Эйлеровым, *если он проходит по одному разу через каждое ребро графа. Докажите, что в таком графе имеется Эйлеров цикл тогда и только тогда, когда степени всех его вершин четны (такие графы называются* четными*).*

Указание. *Достаточность можно установить, доказав правильность следующей процедуры построения Эйлерова цикла.*

1) Выбрать произвольную вершину a и построить цикл, начинающийся и закачивающийся в a, следуя правилу (*): прийдя в некоторую вершину, выйти из нее по произвольному ребру, еще не включенному в цикл.

2) Если построенный цикл содержит не все ребра, то выбрать среди вершин, через которые он проходит, произвольную вершину b, инцидентную еще не пройденному ребру (почему такая вершина найдется?), и построить, начиная с нее, цикл, следуя тому же правилу (*). Объединить этот цикл с построенным ранее.

3) Повторять пункт (2) до тех пор, пока построенный цикл не станет Эйлеровым.

Задача 2.14. *Докажите, что для любого связного неориентированного графа $G = (V, E)$ описанный в предыдущей задаче алгоритм построения Эйлерова цикла можно реализовать за время $O(|E|)$.*

Задача 2.15. *Используя алгоритм из задачи 2.13, определить по неориентированному графу $G = (V, E)$ четный ли он. Если он не является четным, то удалить из него минимальное число ребер, чтобы он стал четным. Построить в исходном или в получившемся после удаления ребер четном графе Эйлеров цикл.*

$V = \{a, b, c, e, f, g, h, k, m, n\}$, $E = \{(a, c), (a, h), (a, m), (a, k), (b, c), (b, k), (b, f), (b, m), (c, k), (c, m), (e, f), (e, g), (f, k), (f, n), (g, m), (g, h), (h, k), (h, m), (k, n)\}$.

Задача 2.16. *Неориентированный граф $G = (V, E)$ называется* двудольным, *если его вершины можно разбить на две непересекающихся части X и Y ($V = X \cup Y, X \cap Y = \emptyset$) так, что каждое ребро из $e \in E$ соединяет вершину из X с вершиной из Y. Такой граф также называется* бихроматическим, *так как его вершины можно раскрасить в два цвета так, что соседние вершины будут окрашены в разные цвета.*

Докажите, что граф является двудольным тогда и только тогда, когда в нем нет циклов нечетной длины (теорема Кёнига).

Указание. *Достаточность можно установить, доказав правильность следующей процедуры разбиения V на X и Y,*

1) Выбрать произвольную вершину $v \in V$, поместить ее в X и отметить знаком +.

2) **WHILE** *имеются неотмеченные вершины с отмеченными соседями*

2.1) **DO** { *Поместить все неотмеченные вершины с соседями из X в Y и отметить их знаком - ;*

2.2) Поместить все неотмеченные вершины с соседями из Y в X и отметить их знаком +

};

3) **IF** *после завершения цикла 2 в V остались неотмеченные вершины*

THEN *поместить произвольную такую вершину v в X, отметить ее знаком +*

и снова повторить цикл 2

ELSE *выдать в качестве результата полученные множества: X – вершины, отмеченные +, и Y – вершины, отмеченные -.*

Для доказательства корректности этой процедуры установите, что в процессе разметки ни одна вершина не получит соседа с той же меткой.

Задача 2.17. *Используя результаты предыдущей задачи, определить является ли заданный ниже неориентированный граф $G = (V, E)$ двудольным. Если он не двудольный, то каково минимальное число ребер, которые нужно из него удалить, чтобы он стал двудольным? Приведите обоснование ответа.*
$V = \{a, b, c, e, f, g, h, k, m, n\}$, $E = \{(a, h), (a, n), (a, k), (b, k), (b, f), (b, m), (c, k), (c, h), (e, f), (e, g), (f, a), (f, m), (g, m), (m, n)\}$.

Задача 2.18. *Пусть имеется группа студентов, каждый из которых должен сдать два экзамена по предметам из множества $\{p_1, \ldots, p_n\}$. Экзамены должны пройти в два дня: вторник и четверг. За день студент может сдать только один экзамен.*

Определите, какие экзамены нужно назначить на вторник и на четверг, чтобы каждый студент смог сдать оба свои экзамена. Можно ли составить такое расписание, чтобы каждый экзамен принимался только в один из дней? Предложите эффективный алгоритм для составления такого расписания (если оно существует).

Задача 2.19. *Предложите алгоритм сложности $O(|V|+|E|)$, который в неориентированном графе $G = (V, E)$ удаляет каждую вершину v степени 2, заменяя инцидентные ей ребра (u, v) и (v, w) на ребро (u, w). Подчеркнем, что в результирующем графе не должно остаться вершин степени 2, т.е. каждая линейная цепь заменится на одно ребро.*

3 Неориентированные и ориентированные деревья

3.1 Основные определения

Деревья являются одним из интереснейших классов графов, используемых для представления различного рода иерахических структур.

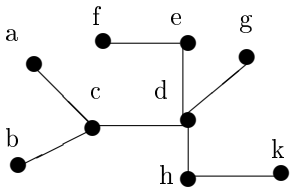
Определение 10. *Неориентированный граф называется* деревом, *если он связный и в нем нет циклов.*

Определение 11. *Ориентированный граф $G = (V, E)$ называется* (ориентированным) деревом, *если*
1) в нем есть одна вершина $r \in E$, в которую не входят ребра; она называется корнем *дерева;*
2) в каждую из остальных вершин входит ровно по одному ребру;
3) все вершины достижимы из корня.

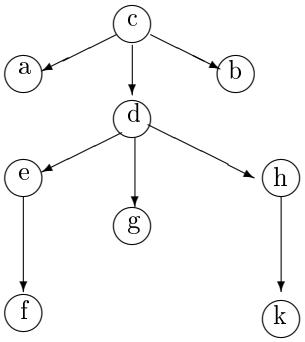
На рисунке 6 показаны примеры неориентированного дерева G_1 и ориентированного дерева G_2. Обратите внимание на то, что дерево G_2 получено из G_1 с помощью выбора вершины c в качестве корня и ориентации всех ребер в направлении “от корня".

Это не случайно. Докажите самостоятельно следующее утверждение о связи между неориентированными и ориентированными деревьями.

Лемма 3.1. *Если в любом неориентированном дереве $G = (V, E)$ выбрать произвольную вершину $v \in V$ в качестве корня и сориентировать все ребра в направлении “от корня”, т.е.*



Неориентированное дерево G_1



Ориентированное дерево G_2

Рис. 6: Неориентированное и ориентированное деревья

сделать v началом всех инцидентных ей ребер, вершины, смежные с v — началами всех инцидентных им еще не сориентированных ребер и т.д., то полученый в результате ориентированный граф G' будет ориентированным деревом.

Неориентированные и ориентированные деревья имеют много эквивалентных характеристик.

Теорема 3.1. *Пусть $G = (V, E)$ — неориентированный граф. Тогда следующие условия эквивалентны.*
(1) G является деревом.
(2) Для любых двух вершин в G имеется единственный соединяющий их путь.
(3) G связен, но при удалении из E любого ребра перестает быть связным.
(4) G связен и $|E| = |V| - 1$.
(5) G ациклический и $|E| = |V| - 1$.
(6) G ациклический, но добавление любого ребра к E порождает цикл.

Доказательство. (1) $\Rightarrow$ (2): Если бы в G некоторые две вершины соединялись двумя путями, то, очевидно, в G имеелся бы цикл. Но это противоречит определению дерева в (1).
(2) $\Rightarrow$ (3): Если G связен, но при удалении некоторого ребра $(u, v) \in E$ не теряет связности, то между u и v имеется путь, не содержащий это ребро. Но тогда в G имеется не менее двух путей, соединяющих u и v, что противоречит условию (2).
(3) $\Rightarrow$ (4): Предоставляется читателю (см. задачу 2.8).
(4) $\Rightarrow$ (5): Если G содержит цикл и является связным, то при удалении любого ребра из цикла связность не должна нарушиться, но ребер останется $|E| = |V - 2|$, а по задаче 2.8(а) в связном графе должно быть не менее $|V - 1|$ ребер. Полученное противоречие показывает, что циклов в G нет и выполнено условие (5).
(5) $\Rightarrow$ (6): Предположим, что добавление ребра (u, v) к E не привело к появлению цикла. Тогда в G вершины u и v находятся в разных компонентах связности. Так как $|E| = |V - 1|$, то в одной из этих компонент, пусть это (V_1, E_1), число ребер и число вершин совпадают: $|E_1| = |V_1|$. Но тогда в ней имеется цикл (см. задачу 2.8 (б)), что противоречит ацикличности G.
(6) $\Rightarrow$ (1): Если бы G не был связным, то нашлись бы две вершины u и v из разных компонент связности. Тогда добавление ребра (u, v) к E не привелобы к появлению цикла, что противоречит (6). Следовательно, G связен и является деревом.

Для ориентированных деревьев часто удобно использовать следующее индуктивное определение.

Определение 12. *Определим по индукции класс ориентированных графов $\mathcal{D}$, называемых деревьями. Одновременно для каждого из них определим выделенную вершину — корень.*
1) Граф $T_0 = (V, E)$, с единственной вершиной $V = \{v\}$ и пустым множеством ребер $E = \emptyset$ является деревом (входит в $\mathcal{D}$). Вершина v называется корнем этого дерева.
2) Пусть графы $T_1 = (V_1, E_1), \ldots, T_k = (V_k, E_k)$ с корнями $r_1 \in V_1, \ldots, r_k \in V_k$ принадлежат $\mathcal{D}$, а r_0 — новая вершина, т.е. $r_0 \notin \bigcup_{i=1}^{k} V_i$. Тогда классу $\mathcal{D}$ принадлежит также следующий граф $T = (V, E)$, где $V = \{r_0\} \cup \bigcup_{i=1}^{k} V_i$, $E = \{(r_0, r_i) \mid i = 1, \ldots, k\} \cup \bigcup_{i=1}^{k} E_i$. Корнем этого дерева является вершина r_0.
3) Других графов в классе $\mathcal{D}$ нет.

Рисунок 7 иллюстрирует это определение.

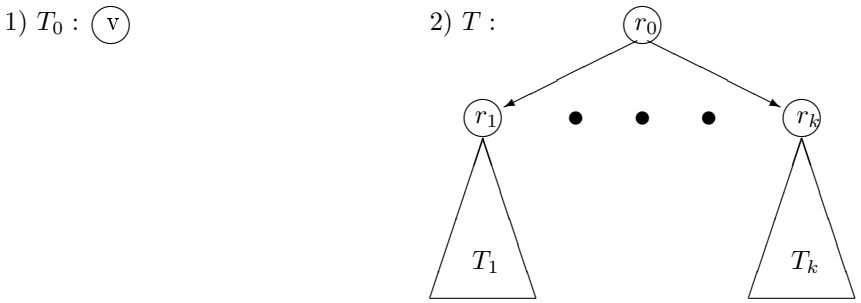


Рис. 7: Индуктивное определение ориентированных деревьев

Теорема 3.2. *Определения ориентированных деревьев 11 и 12 эквивалентны.*

Доказательство. $\Rightarrow$ Пусть граф $G = (V, E)$ удовлетворяет условиям определения 11. Покажем индукцией по числу вершин $|V|$, что $G \in \mathcal{D}$.
Если $|V| = 1$, то единственная вершина $v \in V$ является по свойству (1) корнем дерева, т.е. в этом графе ребер нет: $E = \emptyset$. Тогда $G = T_0 \in \mathcal{D}$.

Предположим, что всякий граф с $\leq n$ вершинами, удовлетворяющий определению 11 входит в $\mathcal{D}$. Пусть граф $G = (V, E)$ с $(n+1)$-й вершиной удовлетворяет условиям определения 11. По условию (1) в нем имеется вершина-корень r_0. Пусть из r_0 выходит k ребер и они ведут в вершины $r_1, \ldots, r_k$ $(k \geq 1)$. Обозначим через G_i, $(i = 1, \ldots, k)$ граф, включающий вершины $V_i = \{v \in V \mid v \text{ достижима из } r_i\}$ и соединяющие их ребра $E_i \subseteq E$. Легко понять, что G_i удовлетворяет условиям условиям определения 11. Действительно, в r_i не входят ребра, т.е. эта вершина — корень G_i. В каждую из остальных вершин из V_i входит по одному ребру как и в G. Если $v \in V_i$, то она достижима из корня r_i по определению графа G_i. Так как $|V_i| \leq n$, то по индуктивному предположению $G_i \in \mathcal{D}$. Тогда граф G получен по индуктивному правилу (2) определения 12 из деревьев $G_1, \ldots, G_k$ и поэтому принадлежит классу $\mathcal{D}$.
$\Leftarrow$ Если некоторый граф $G = (V, E)$ входит в класс $\mathcal{D}$, то выполнение условий (1)-(3) определения 11 для него легко установить индукцией по определению 11. Предоставляем это читателю

в качестве упражнения.

С ориентированными деревьями связана богатая терминология, пришедшая из двух источников: ботаники и области семейных отношений.

Корень — это единственная вершина, в которую не входят ребра, *листья* — это вершины, из которых не выходят ребра. Путь из корня в лист называется *ветвью* дерева. *Высота дерева* — это максимальная из длин его ветвей. *Глубина вершины* — это длина пути из корня в эту вершину. Для вершины $v \in V$, подграф дерева $T = (V, E)$, включающий все достижимые из v вершины и соединяющие их ребра из E, образует *поддерево* T_v дерева T с корнем v (см. задачу 3.3). *Высота вершины* v — это высота дерева T_v. Граф, являющийся объединением нескольких непересекающихся деревьев, называется *лесом.*

Если из вершины v ведет ребро в вершину w, то v называется *отцом* или *родителем* w, а w – *сыном* или *ребенком* v (в последнее время в англоязычной литературе чаще употребляется асексуальная пара терминов: родитель — ребенок). Из определения дерева непосредственно следует, что у каждой вершины кроме корня имеется единственный отец. Если из вершины v ведет путь в вершину w, то v называется *предком* w, а w – *потомком* v. Вершины, у которых общий отец, называются *братьями* или *сестрами.*

Определим важный подкласс ориентированных деревев — бинарные или двоичные деревья.

Определение 13. *Ориентированное дерево называется* бинарным *или* двоичным, *если у каждой его внутренней вершины имеется не более двух сыновей, причем ребра, ведущие к ним помечены двумя разными метками (обычно используются метки из пар: "левый" — "правый", 0 — 1, + — − и т.п.)*

Бинарное дерево называется полным, *если у каждой его внутренней вершины имеется два сына и все его ветви имеют одинаковую длину.*

Несложно подсчитать число листьев и вершин у полных бинарных деревьев.

Лемма 3.2. *Полное бинарное дерево высоты* h *содержит* 2^h *листьев и* $(2^{h+1} - 1)$ *вершин.*

Выделим еще один класс графов, обобщающий ориентированные деревья, —*ориентированные ациклические графы.* У этих графов может быть несколько корней — вершин, в которые не входят ребра, и в каждую вершину может входить несколько ребер, а не одно, как у деревьев.

3.2 Представления деревьев

Поскольку дерево – это частный случай графа, то каждое из рассмотренных в разделе 2 представлений графов является также и представлением деревьев. Кроме этого можно определить несколько специальных представлений деревьев.

3.2.1 Сылка на вершину-отца

В некоторых задачах, использующих деревья, требуется перемещаться по дереву только от вершин-потомков к вершинам-предкам. В этом случае дерево T с n вершинами $v_1, \ldots, v_n$ можно задать с помощью массива ОТЕЦ[1..n], в котором элемент ОТЕЦ[i] является номером вершины, являющейся отцом вершины v_i. Признаком корня может быть значение 0 в этом массиве.

3.2.2 Скобочное представление

Дерево — "двумерный"объект. Но его можно представить в виде линейной строки, если использовать подходящую скобочную структуру. Определим скобочное представление СП(T) дерева T, следуя индуктивному определению 12 ориентированного дерева.

Базис. Если $T = (\{v\}, \emptyset)$ — дерево из одной вершины v, то $\text{СП}(T) = v$.
Индукционный шаг. Если дерево T получено по пункту (2) определения 12 путем присоединения деревьев $T_1, \ldots, T_k$ к новому корню r, то $\text{СП}(T) = r(\text{СП}(T_1), \ldots, \text{СП}(T_k))$.

Пример 3. *Например, скобочное представление дерева G_2, показанного на рис. 6, имеет вид $СП(G_2) = c(a, (d(e(f), g, h(k)), b)$.*

Ясно, что по скобочному представлению $\text{СП}(T)$ исходное дерево T восстанавливается однозначно. Обратное справедливо с точностью до порядка сыновей каждой вершины. Если такой порядок зафиксирован, то соответствующее ему представление единственно.

3.2.3 Представление множеством путей

Множество всех путей, начинающихся в корне дерева, очевидно, однозначно задает его структуру. Хотя такое представление является избыточным (например, все пути до внутренних вершин можно восстановить по путям до листьев), оно используется на практике при представлении иерархической структуры документов и книг в их содержаниях и оглавлениях.

Пример 4. *Для дерева G_2, показанного на рис. 6, такое представление имеет вид:*
c
c.a
c.d
c.d.e
c.d.e.f
c.d.g
c.d.h
c.d.h.k
c.b

3.2.4 Стандартное представление бинарного дерева

Из каждой внутренней вершины бинарного дерева выходит не более двух дуг — одна ведет к левому сыну, другая – к правому. Стандартным представлением бинарного дерева является списковая структура, состоящая из элементов-записей типа

ВЕРШИНА = (Номер, ЛЕВ, ПРАВ),

где *ЛЕВ* — это ссылка на левого сына вершины, а *ПРАВ* – на правого. Такая структура позволяет эффективно переходить от вершин-предков к вершинам-потомкам. Если в задаче нужно также передвигаться по дереву в обратном направлении, то в запись можно добавить поле *ОТЕЦ*, указывающее на вершину-отца данной вершины.

3.2.5 Представление бинарного дерева с помощью массива

Для полных и близких к ним бинарных деревьев $T = (\{v_1, \ldots, v_n\}, E)$ эффективным является представление с помощью одного целочисленного массива $A[1..N]$, определяемого следующим образом:
$A[1]$ – корень дерева T, $A[2i]$ — левый сын вершины $A[i]$, $A[2i+1]$ — правый сын вершины $A[i]$. Если соответствующего сына нет, то значение равно 0. Это представление позволяет легко перемещаться по дереву сверху вниз и снизу вверх, используя умножение и деление на 2. Размер N массива A, представляющего дерево высоты h равен $O(2^h)$. Поэтому для "редких" деревьев или деревьев, у которых одна ветвь которых намного длиннее других, это представление неэффективно.

3.2.6 Представление произвольного дерева с помощью бинарного

Произвольное дерево можно представить с помощью бинарного дерева с сохранением числа вершин и дуг. Мы определим индуктивно такое представление $Bin(T_1, \ldots, T_n)$ для леса деревьев $T_1, \ldots, T_k$.
Базис. Если $n = 1$ и $T_1 = (\{v\}, \emptyset)$ — дерево из одной вершины v, то $Bin(T_1) = T_1$.
Индукционный шаг. Если дерево T_1 получено по пункту (2) определения 12 путем присоединения деревьев $T_{11}, \ldots, T_{1k}$ к новому корню r, то $Bin(T_1, \ldots, T_n)$ имеет корень r, левым сыном этого корня при $k > 0$ является корень бинарного дерева $Bin(T_{11}, \ldots, T_{1k})$, а правым сыном при $n > 1$ является корень бинарного дерева $Bin(T_2, \ldots, T_n)$.

Пример 5. *Применив эти реккурентные соотношения к дереву G_2, показанному на рис. 6, получим бинарное дерево $Bin(G_2)$, показанное на следующем рисунке.*

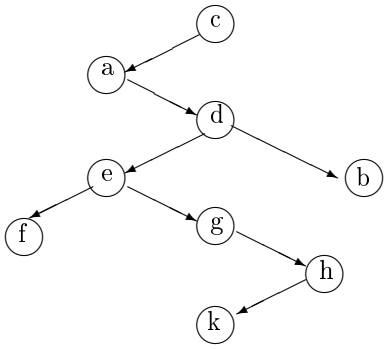


Рис. 8: Бинарное представление $Bin(G_2)$ дерева G_2 с рис. 6

Достаточно неочевидное рекурсивное правило, определяющее $Bin(T)$, можно просто переформулировать в виде локального правила, определяющего левого и правого сыновей каждой вершины: *в $Bin(T)$ левым сыном вершины v является ее первый сын в T, а правым сыном – правый брат в в T.* Это правило однозначно определяет $Bin(T)$ для упорядоченного дерева с фиксированным порядком сыновей каждой вершины.

3.3 Деревья и формулы (выражения)

Напомним общее понятие *формулы* над системой функций $\mathcal{B}$. Аналогичные синтаксические объекты в логике предикатов названются *термами*, а в языках программирования такие конструкции часто называются *выражениями.*

Итак, формула над множеством функций **F**, множеством констант **C** и множеством переменных **Var** определяется индуктивно по следующим правилам.

- Переменная из **Var** есть формула.
- Константа из **C** есть формула.
- Если $g_1, \ldots, g_k$ — формулы, а $f^{(k)}$ — k-местная функция из **F**, то $f(g_1, \ldots, g_k)$ — это формула.

Обозначим множество всех таких формул через $\mathcal{F}(\mathbf{F}, \mathbf{C}, \mathbf{Var})$.

Рассмотрим класс упорядоченных размеченных деревьев $\mathcal{T}(\mathbf{F}, \mathbf{C}, \mathbf{Var})$, листья которых помечены элементами из $(\mathbf{C} \cup \mathbf{Var})$, а внутренние вершины — функциями из $\mathbf{F}$, причем, если вершина помечена символом k-местной функции из $\mathbf{F}$, то у нее имеется k сыновей.

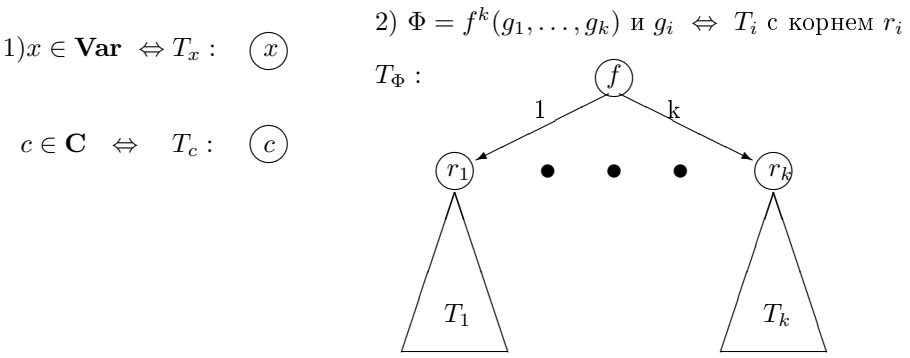


Рис. 9: Индуктивное определение связи между формулами и деревьями

Предложение 3.1. *Между множеством формул $\mathcal{F}(\mathbf{F}, \mathbf{C}, \mathbf{Var})$ и множеством деревьев $\mathcal{T}(\mathbf{F}, \mathbf{C}, \mathbf{Var})$ имеется взаимно однозначное соответствие.*

Доказательство. Это соответствие легко устанавливается индукцией по определениям формул и деревьев. Оно показано выше на рис. 9.

Пример 6. *Рассмотрим, например, класс обычных арифметических формул над множеством функций $\mathbf{F} = \{+, -, *, :\}$, целочисленных констант $\mathbf{C} = \{0, 1, 2, \ldots\}$ и переменных $\mathbf{Var} = \{x, y, z, \ldots\}$. Пусть формула $\Phi = +(*(5, +(x, 7)), (: (y, +(x, 7)))$ (ее обычное представление $\Phi = 5 * (x + 7) + y : (x + 7)$)*

Тогда в соответствии с предложением 3.1 эта формула представляется деревом T_Φ, изображенном на рис. 10.

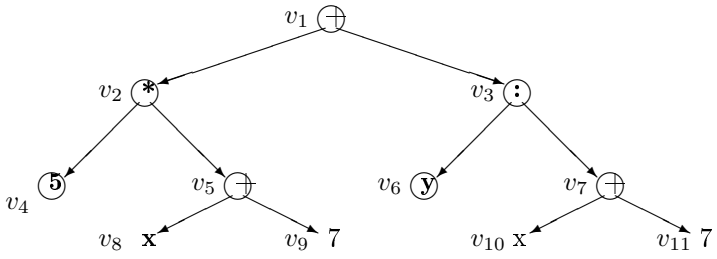


Рис. 10: Дерево T_Φ

На этом рисунке не указаны явно номера ребер, выходящих из внутренних вершин дерева, которые идентифицируют порядок аргументов операций. Предполагается, что для коммутативных операций +, * это несущественно, а для некоммутативных, таких, как :, первый аргумент расположен левее второго.

Заметим, что у деревьев, представляющих арифметические или логические (булевские) формулы, внутренние вершины имеют не более 2-х сыновей, т.е. они являются бинарными.

Ориентированные ациклические графы также используются для представления формул. Они получаются из соответствующих деревьев при склеивании вершин, представляющих одинаковые подформулы. Для формулы $\Phi = 5 * (x + 7) + y : (x + 7)$ такой граф G_Φ получается при склеивании вершин v_5 и v_7 дерева T_Φ, представляющих подформулу $(x + 7)$.

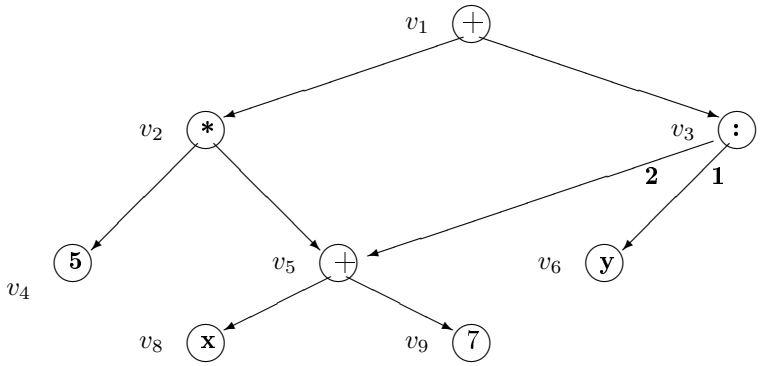


Рис. 11: Ациклический граф G_Φ

На этом рисунке явно указаны номера ребер, выходящих из вершины v_3, которые определяют порядок аргументов приписанной этой вершине операции :. Ясно, что при отсутствии такого указания и использовании порядка "по умолчанию" — первый аргумент слева — граф представлял бы другое выражение.

3.4 Обходы деревьев

Часто при обработке представленной в дереве информации требуется обойти некоторым регулярным способом все его вершины. Имеется два естественных стандартных способа обхода деревьев. Каждый из них позволяет линейно упорядочить вершины дерева и тем самым представить его "двумерную структуру" в виде линейной последовательности вершин.

Прямой (префиксный) обход дерева основан на принципе: "сначала родитель, затем дети". Определим индукцией по построению дерева T в определении 12 его прямое представление $ПР(T)$ следующим образом.

1) Если $T_0 = (\{v\}, \emptyset)$, то $ПР(T_0) = v$.

2) Если T получено из деревьев $T_1, \ldots, T_k$ и нового корня r_0 по пункту (2) определения 12 то $ПР(T) = r_0\, ПР(T_1) \ldots ПР(T_k)$.

Обратный (суффиксный) обход дерева основан на противоположном принципе: "сначала дети, затем родитель". Вот его индуктивное определение.

1) Если $T_0 = (\{v\}, \emptyset)$, то $ОБР(T_0) = v$.

2) Если T получено из деревьев $T_1, \ldots, T_k$ и нового корня r_0 по пункту (2) определения 12, то $ОБР(T) = ОБР(T_1) \ldots ОБР(T_k)\, r_0$.

Для бинарных деревьев, внутренние вершины которых имеют не более 2-х сыновей, помеченных как "левый" и "правый", можно естественно определить еще один способ обхода — *инфиксный (внутренний) обход,* основанный на принципе: "сначала левый сын, затем родитель, а затем правый сын". Он определяется следующим образом.

1) Если $T_0 = (\{v\}, \emptyset)$, то $ИНФ(T_0) = v$.

2) Если T получено из деревьев T_1, T_2 и нового корня r_0 по пункту (2) определения 12, то $ИНФ(T) = ИНФ(T_1) r_0 ИНФ(T_2)$.
(Если одно из деревьев T_1, T_2 пусто, то соответствующее ему инфиксное представление тоже пусто).

Пример 7. *Построим в соответствии с этими определениями три разных обхода бинарного дерева $T_Ф$, изображенного на рис. 10 (в скобках после вершины указана ее метка).*

$ПР(T_Ф) = v_1(+)v_2(*)v_4(5)v_5(+)v_8(x)v_9(7)v_3(:)v_6(y)v_7(+)v_{10}(x)v_{11}(7)$.
$ОБР(T_Ф) = v_4(5)v_8(x)v_9(7)v_5(+)v_2(*)v_6(y)v_{10}(x)v_{11}(7)v_7(+)v_3(:)v_1(+)$.
$ИНФ(T_Ф) = v_4(5)v_2(*)v_8(x)v_5(+)v_9(7)v_1(+)v_6(y)v_3(:)v_{10}(x)v_7(+)v_{11}(7)$.

Для упорядоченного размеченного дерева T из класса $\mathcal{T}(\mathbf{F}, \mathbf{C}, \mathbf{Var})$ по любому из указанных обходов $ПР(T)$, $ОБР(T)$ и, если дерево бинарное, – $ИНФ(T)$ можно однозначно восстановить само дерево T (см. задачу 3.9).

Замечание. Для вычислительных приложений особенно интересен обратный обход, иногда называемый обратной польской записью. За один проход по такой записи можно вычислить значение соответствующего выражения, используя стек, или откомпилировать его. Напомним правило вычисления по обратной записи: движемся по ней слева направо, при прохождении констант или переменных вталкиваем их значения в стек, а при прохождении k-местной функции заменяем k верхних элементов стека на значение этой функции от них. В результате после завершения прохода по записи на стеке останется значение вычисляемого выражения.

Каждое из указанных рекурсивных определений обходов ПР, ОБР и ИНФ можно естественно преобразовать в рекурсивную процедуру на любом современном языке программирования. Правильность таких процедур является непосредственным следствием их определений, но использование в реализации универсальных методов делает их зачастую менее эффективными, чем соответствующие нерекурсивные процедуры. Мы приведем здесь нерекурсивную процедуру для построения инфиксного обхода бинарного дерева. Будем считать, что исходное дерево представлено стандартным способом: каждая вершина — это элемент типа *ВЕРШИНА = (Номер, ЛЕВ, ПРАВ)*. Доступ к дереву осуществляется через параметр КОРЕНЬ - ссылку на его корень. В алгоритме будет использоваться стек вершин S со стандартными операциями: $PUSH(S, v)$ – поместить (ссылку на)вершину v на верх стека S, $POP(S)$ — удалить верхний элемент стека, $TOP(S)$ — функция, возвращающая верх стека S и $EMTY(S)$ — предикат, определяющий пустоту стека S. Переменная CUR будет указывать на текущую вершину в обходе, целочисленная переменная $NOMER$ будет содержать номер текущей вершины. Результат алгоритма — массив NUM, в котором для каждой вершины v указан ее номер в инфиксном порядке $NUM[v]$.

Алгоритм **ИНФ**

```
1.  CUR := КОРЕНЬ;  NOMER := 1;
2.  PUSH(S,CUR);  CUR := CUR. ЛЕВ;
3.  WHILE  (CUR ≠ NIL) or  NOT EMPTY(S)  DO
4.      {WHILE  (CUR ≠ NIL) DO
5.          { PUSH(S,CUR); CUR := CUR. ЛЕВ};
6.      CUR := TOP(S);  POP(S);
7.      NUM[CUR] :=NOMER; NOMER := NOMER+1;
8.      CUR := CUR. ПРАВ
9.      }
```

Теорема 3.3. *Алгоритм* **ИНФ** *строит инфиксный обход бинарного дерева* $T = (V, E)$ *с* n *вершинами за время* $O(n)$.

Доказательство. Если дерево T состоит из одной вершины, то в стр. 2 она будет помещена в стек S, в стр. 6 – удалена из него, в стр. 7 получит номер 1, после чего алгоритм завершится, так как S будет пуст.

Для дерева T с числом вершин $n > 1$ доказательство правильности **ИНФ** проведем индукцией по номеру t вершины в инфиксном порядке. Именно, покажем, что номер 1 **ИНФ** дается верно и что из предположения о том, что номер t присваивается верно, можно вывести, что и номер $t + 1$ также присваивается верно. Нам потребуется более сильное индуктивное предположение, в котором используется одно вспомогательное понятие. *Левый путь из корня в вершину* v — это подпоследовательность таких вершин $v_{i_1}, \ldots, v_{i_l}$ на пути $p = v_0, v_1, \ldots, v_k = v$ из корня v_0 в v, у которых левые сыновья также находятся на p.

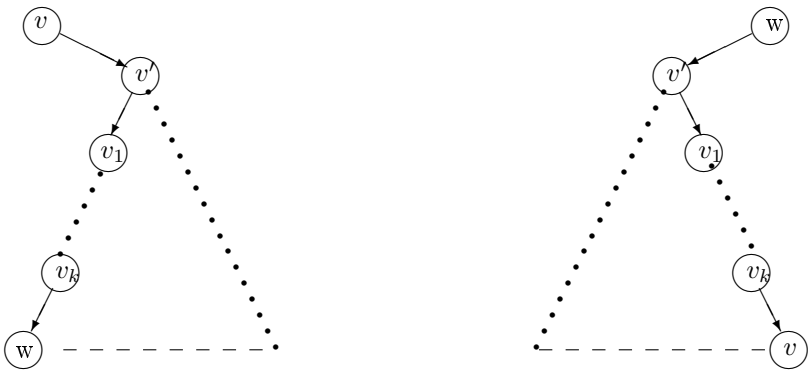
Утверждение. Для каждого $1 \leq t \leq n$ **ИНФ** *присваивает номер* t *в стр.7 вершине CUR верно и в этот момент в стеке* S *лежит левый путь в вершину CUR.*

Базис. Проверим, что номер $t = 1$ присваивается вершине, которая идет первой в инфиксном порядке. Из определения этого порядка следует, что это самая левая вершина T, т.е. последняя вершина на самом левом пути в дереве (у нее нет левого сына). Алгоритм **ИНФ** в стр. 2 поместит в S корень, затем в цикле в стр. 4-5 поместит в S самый левый путь в T, в стр. 6 сделает его последнюю вершину текущей и уберет из S, а в стр. 7 присвоит ей номер 1. В этот момент в в стеке S будет лежать левый путь в CUR.

Шаг индукции. Предположим, что для некоторого $t \geq 1$ Утверждение выполнено. Покажем, что оно будет выполнено и для $t + 1$.

Пусть номер t получила вершина v. Какая вершина w должна следовать за v в инфиксном порядке? Возможны три случая.

1) У v есть правый сын v'. Тогда w – это самая левая вершина в поддереве с корнем v' — она является первой в инфиксной нумерации этого поддерева (см. рис. 12 слева).
В этом случае алгоритм **ИНФ** в стр. 8 сделает вершину v' текущей, затем в цикле в стр. 4-5 добавит в S эту вершину и самый левый путь из нее в w. После этого в стр. 6 CUR получит значение w, w покинет стек и в стр. 7 ей присвоится правильный номер $t + 1$. В этот момент в в стеке S будет лежать левый путь в CUR=w.



Случай 1 Случай 2

Рис. 12: Инфиксный порядок на вершинах: вершина w непосредственно после v

2) У v нет правого сына, но левый путь из корня в v непуст. Тогда из определения инфиксного порядка следует, что w – это последняя вершина на левом пути из корня в v (см. рис. 12 справа).

В этом случае по предположению индукции в момент присвоения номера t вершине v наверху стека S лежит w. Алгоритм **ИНФ** в стр. 8 присвоит CUR значение NIL, вернется к началу основного цикла, пропустит цикл в стр. 4-5, в стр.6 CUR получит значение вершины стека w, w покинет стек и в стр. 7 ей присвоится правильный номер $t+1$. Как и в предыдущем случае в этот момент в в стеке S будет лежать левый путь в CUR=w.

3) У v нет правого сына и левый путь из корня в v пуст. Это означает, что v – последняя вершина на самом правом пути в T и она является последней вершиной в инфиксном порядке. В этом случае в момент присвоения номера вершине v стек S пуст. Алгоритм **ИНФ** в стр. 8 присвоит CUR значение NIL вернется к началу основного цикла, не войдет в него, так как оба его условия ложны, и завершит работу.

Для оценки времени заметим, что каждая вершина может получить номер в стр. 7 и попасть в стек в стр. 5 не более одного раза. Отсюда следует, что общее время работы алгоритма **ИНФ** не превосходит $O(n)$.

В стандартном представлении бинарных деревьев поля ЛЕВ и ПРАВ у вершин, не имеющих соответствующих сыновей содержат не несущее полезной информации значение 0 (или NIL). На самом деле эти поля можно использовать для организации эффективного инфиксного обхода дерева без применения стека. Для этого в них можно хранить ссылки на соседей данной вершины в инфиксном порядке. При этом добавляются битовые поля-признаки для определения типа информации в полях ЛЕВ и ПРАВ. Получаемое представление называется *прошитым представлением бинарного дерева*. Вершины являются элементами типа

ВЕРШИНАП = (Номер, ЛЕВ, ПрЛев, ПРАВ, ПрПрав).

При этом, если вершина имеет левого сына, то *ПрЛев=1* и поле *ЛЕВ* содержит ссылку на левого сына, а если у вершины левого сына нет, то *ПрЛев=0* и поле *ЛЕВ* содержит ссылку на вершину, предшествующую данной в инфиксном порядке. Аналогично, если вершина имеет правого сына, то *ПрПрав=1* и поле *ПРАВ* содержит ссылку на правого сына, а если у вершины правого сына нет, то *ПрПрав=0* и поле *ПРАВ* содержит ссылку на вершину, следующую за данной в инфиксном порядке.

3.5 Задачи

Задача 3.1. *Докажите, что если в связном неориентированном графе число вершин равно числу ребер, то можно выбросить одно из ребер так, что после этого граф станет деревом.*

Задача 3.2. *Пусть $G=(V,E)$ — неориентированное дерево и $v \in V$ — произвольная вершина. Докажите, что если для каждого ребра $(u,w) \in E$ выбрать ориентацию от u к w, если им заканчивается путь из v в w, и ориентацию от w к u, если им заканчивается путь из v в u, то полученный ориентированный граф будет ориентированным деревом с корнем v. Используйте это утверждение для доказательства следующего факта: если в неориентированном дереве $G=(V,E)$ имеется вершина степени $d>1$, то в нем имеется по крайней мере d вершин степени 1.*

Задача 3.3. *Пусть $T=(V,E)$ — это ориентированное дерево с корнем $v_0 \in V$. определим для каждой вершины $v \in V$ подграф $T_v=(V_v,E_v)$ следующим образом: V_v — это множество вершин, достижимых из v в T, а E_v — это множество ребер из E, оба конца которых входят в V_v. Доказать, что*
а) T_v является деревом с корнем v;
б) если две разные вершины v и u имеют одинаковую глубину, то деревья T_v и T_u не пересекаются.

Задача 3.4. *Пусть $G=(V,E)$ – ориентированный граф с $n>1$ вершинами. Докажите, что G является (ориентированным) деревом тогда и только тогда, когда в G нет циклов,*

имеется одна вершина r, в которую не входят ребра, а в каждую из остальных вершин $v \in V \setminus \{r\}$ входит ровно одно ребро.

Задача 3.5. *Пусть корень ориентированного дерева T имеет 5 сыновей, а каждая из остальных внутренних вершин имеет три или четыре сына, при этом число вершин с 3-я сыновьями вдвое больше числа вершин с 4-я. Сколько всего вершин и ребер в T, если известно, что число его листьев равно 27?*

Задача 3.6. *Докажите по индукции, что в любом бинарном дереве число вершин степени 2 на единицу меньше числа листьев.*

Задача 3.7. *Постройте дерево, представляющее следующую логическую формулу*
$\Psi = ((X \vee \neg Y) \wedge \neg(Z \to (X \wedge Y))) \vee (\neg Z + Y)$.
Для полученного дерева определите прямой, обратный и инфиксный обходы.

Задача 3.8. *Постройте дерево и ациклический ориентированный граф, представляющие следующую арифметическую формулу*
$\Phi = (a + b)/(c + a * d) + ((c + a * d) - (a + b) * (c - d))$.

Сколько вершин удалось сократить?

Задача 3.9. *Для каждого из обходов деревьев ПР(T), ОБР(T) и ИНФ(T) предложите процедуру, восстановления соответствующего дерева $T \in \mathcal{T}(\mathbf{F}, \mathbf{C}, \mathbf{Var})$.*

Задача 3.10. *Предложите нерекурсивные алгоритмы для прямого и обратного обхода деревьев (не обязательно бинарных). Оцените их сложность.*

Задача 3.11. *Реализуйте процедуру, не использующую стек, для инфиксного обхода бинарного дерева, заданного прошитым представлением. Оцените ее сложность.*

4 Минимальное остовное дерево

Пусть $G = (V, E)$ – связный неориентированный граф. Если $|E| > |V|-1$, то в G имеются циклы (см. теорему 3.1). Тогда можно, удаляя "лишние" ребра, можно получить дерево, связывающее те же вершины. Такое дерево называется остовом графа G.

Определение 14. Остовом (остовным деревом, каркасом, скелетом) *связного неориентированного графа $G = (V, E)$ называется дерево $S = (V, T)$ такое, что $T \subseteq E$.*

Пусть задана функция $c : E \to R$, приписывающая каждому ребру $e \in E$ его стоимость (вес, длину) $c(e) \in R$ (R — множество вещественных чисел). Тогда стоимость $c(S)$ дерева S определяется как сумма стоимостей всех его ребер, т.е. $c(S) = \sum_{e \in T} c(e)$.

Минимальным остовом *называется остов минимальной стоимости.*

Таким образом, минимальный остов — это самая дешевая (короткая) система путей, связывающая все вершины G. Его можно было бы найти, перебрав все возможные остовы графа. Но этот перебор мог бы потребовать экспоненциального от числа ребер G времени. К счастью, задача определения минимального остова допускает эффективное решение с помощью "жадных" алгоритмов.

4.1 Алгоритм Крускала

В этом пункте мы рассмотрим алгоритм построения минимального остова, который был предложен Крускалом в 1956г. Два других эффективных алгоритма, принадлежащих Приму и Борувке, описаны в задачах 4.6 и 4.7.

Следующая лемма является непосредственным следствием определения остовного дерева.

Лемма 4.1. *Пусть $S = (V, T)$ - остовное дерево для G. Тогда*
(а) для любых двух вершин v_1 и v_2 из V путь между v_1 и v_2 в S единственный;
(б) если к S добавить любое ребро из $E \setminus T$, то возникнет ровно один цикл.

Теоретическим обоснованием алгоритма Крускала является следующее утверждение.

Лемма 4.2. *Пусть $\{(V_1, T_1), (V_2, T_2), \ldots, (V_k, T_k)\}$ - произвольный остовной лес для G, $k > 1$, $T = \cup_{i=1}^{k} T_i$. Пусть $e = (v, w)$ — ребро наименьшей стоимости в $E \setminus T$ такое, что его концы принадлежат разным деревьям: $v \in V_i$, $w \in V_j$ и $i \neq j$. Тогда существует остовное дерево, содержащее все ребра из T и e, стоимость которого не больше стоимости любого остовного дерева для G, содержащего T.*

Доказательство. Пусть $S' = (V, T')$ — остовное дерево минимальной стоимости, содержащее все ребра T, и ребро $e \notin T'$. Тогда в S' имеется некоторый путь p из v в w, не содержащий e. Выберем на нем первое такое ребро $e' = (v', w')$, для которого $v' \in V_i$, а $w' \notin V_i$. Рассмотрим множество ребер $T_1' = (T' \cup \{e\}) \setminus \{e'\}$. Это множество задает остовное дерево, так как добавление ребра e к T' приводит по лемме 4.1 к образованию единственного цикла, а удаление ребра e' этот цикл разрушает. Так как $e' \notin T$, то выбор e гарантирует, что $c(e) \leq c(e')$. Поэтому $c(T_1') = c(T') + c(e) - c(e') \leq c(T')$. Из выбора T' тогда следует, что $c(T_1') = c(T')$ и, следовательно, $S_1 = (V, T_1')$ является требуемым остовным деревом.
□

Эта лемма лежит в основе алгоритма построения минимального остовного дерева **МинОстов**. Для ускорения в нем используются эффективные алгоритмы для реализации очередей с приоритетами (можно для этого выбрать, например, сортирующие деревья или 2-3-деревья) и структуры данных и алгоритмы для представления системы множеств и выполнения операций ОБЪЕДИНИТЬ-НАЙТИ (для этого можно выбрать, например, деревья со сжатием путей).

АЛГОРИТМ МинОстов
Вход: $G = (V, E)$- связный неориентированный граф, $c(e)$ - функция стоимости ребер.
Выход: T - множество ребер минимального остовного дерева.
Структуры данных: Q - очередь с приоритетами для ребер, VS - набор непересекающихся подмножеств вершин G, элементы VS - множества вершин остовных деревьев в текущем остовном лесу.

```
1.   T := ∅; VS := ∅;
2.   создать очередь с приоритетами Q из всех ребер из E;
3.   FOR EACH v ∈ V DO добавить {v} к VS;
4.   WHILE |VS| > 1 DO
5.       {  (v, w) := MIN(Q); УДАЛИТЬ(Q, (v, w));
6.          W1 := НАЙТИ(v);  W2 := НАЙТИ(w);
7.          IF W1 ≠ W2 THEN
8.              { ОБЪЕДИНИТЬ(W1, W2, W1); T := T ∪ {(v, w)}}
9.       }
```

Теорема 4.1. *Алгоритм* **МинОстов** *строит минимальное остовное дерево для связного графа $G = (V, E)$. Если в цикле в строках 4 - 9 рассматривается d ребер, то затрачивается время $O(d \log_2 m)$, где $m = |E|$. В худшем случае выполнение алгоритма* **МинОстов** *занимает время $O(m \log_2 m)$.*

Доказательство. Нетрудно проверить, что инвариантом цикла в строках 4-9 является следующее свойство:

система множеств $VS = \{V_1, \ldots, V_k\}$ *и набор ребер* T *задают остовной лес* $\{(V_1, T_1), (V_2, T_2), \ldots, (V_k, T_k)\}$ *исходного графа* G, $(T = \cup_{i=1}^{k} T_i)$, *который может быть расширен до минимального остовного дерева графа* G.

Это свойство, очевидно выполнено перед началом цикла, т.к. каждое из деревьев в VS содержит одну вершину, а множество ребер T пусто. Пусть оно выполнено перед очередным выполнением тела цикла. Если на этом выполнении для выбранного ребра (v, w) вершины v и w оказываются в разных деревьях, то это ребро добавляется к T, а соответствующие деревья объединяются в стр. 8. Очевидно, циклы при таком объединении не появляются и система VS остается остовным лесом для G. Так как ребро (v, w) имеет минимальный вес, то по лемме 4.2 этот лес может быть расширен до минимального остовного дерева.

Из связности G следует, что после завершения **МинОстов** лес VS будет состоять из одного дерева и оно и будет минимальным.

Оценим время работы алгоритма **МинОстов**. Для операций с очередью ребер Q выберем, например, двоичное сортирующее дерево. Тогда его инициализация в стр. 2 производится за время $O(|E|)$, а каждая операция извлечения минимального ребра в стр. 5 выполняется за время $O(\log_2 |E|)$. Для реализации системы множеств VS и операций ОБЪЕДИНИТЬ-НАЙТИ над ними выберем деревья и алгоритмы со сжатием путей. Тогда стр. 3 выполняется за время $O(|V|)$, а все операции НАЙТИ и ОБЪЕДИНИТЬ в стр. 6 и 8 – за время $O(|V|G(|V|)$, где $G(|V|) << \log_2 |V|$ – медленно растущая функция. Поэтому, если в основном цикле рассматривалось d ребер, то стр. 5 выполнялась d раз и общее время алгоритма не превосходит $O(|E|) + O(d \log_2 |E|) + O(|V|G(|V|) = O(d \log_2 |E|)$, так как $d \geq |V| - 1$.

□

Пример 8. *Рассмотрим следующий нагруженный граф* G_1 :

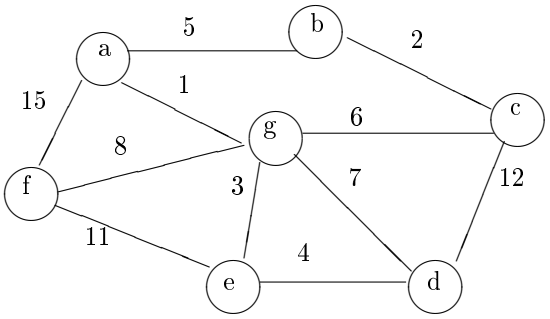


Рис. 13: Граф G_1

Применим к нему алгоритм **МинОстов**. Для простоты сразу покажем порядок, в котором будут выдаваться ребра из очереди Q в стр.5 алгоритма.

$Q = \{(a, g), (b, c), (g, e), (e, d), (a, b), (g, c), (d, g), (f, g), (e, f), (c, d), (a, f)\}$.

Выполнение основного цикла представим в виде таблицы, каждая строка которой соответствует рассмотрению очередного ребра (v, w) в стр. 5. В столбце T ребра, включаемые в T отмечены $+$. Вначале система VS состоит из семи множеств V_i $(i = 1, \ldots, 7)$, содержащих по одной вершине.

$(\mathbf{v},\mathbf{w})$	**T**	**W1**	**W2**	V_1	V_2	V_3	V_4	V_5	V_6	V_7
				$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$
(a,g)	+	1	7	$\{a,g\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	
(b,c)	+	2	3	$\{a,g\}$	$\{b,c\}$		$\{d\}$	$\{e\}$	$\{f\}$	
(g,e)	+	1	5	$\{a,e,g\}$	$\{b,c\}$		$\{d\}$		$\{f\}$	
(e,d)	+	1	4	$\{a,d,e,g\}$	$\{b,c\}$				$\{f\}$	
(a,b)	+	1	2	$\{a,b,c,d,e,g\}$					$\{f\}$	
(g,c)	−	1	1	$\{a,b,c,d,e,g\}$					$\{f\}$	
(d,g)	−	1	1	$\{a,b,c,d,e,g\}$					$\{f\}$	
(f,g)	+	1	5	$\{a,b,c,d,e,f,g\}$						

Таким образом, мы построили для G_1 минимальный остов $S = (V,T)$, где $T = \{(a,g),(b,c),(g,e),(e,d),(a,b),(f,g)\}$. Его стоимость $c(S) = 23$.

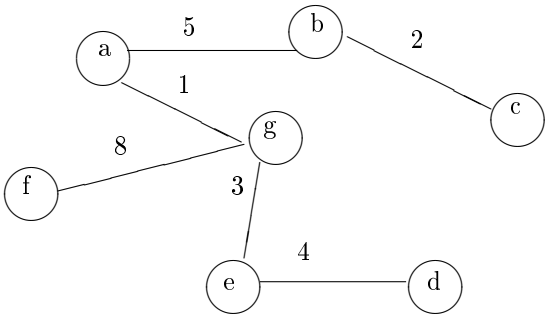


Рис. 14: Минимальный остов $S = (V,T)$ для графа G_1

4.2 Задачи

Задача 4.1. *Найти минимальное остовное дерево для неориентированного графа* $G = (V,E)$, *где* $V = \{v_1,v_2,v_3,v_4,v_5,v_6,v_7,v_8,v_9\}$, $E = \{(v_1,v_2,18),(v_1,v_3,2),(v_3,v_2,4),(v_3,v_4,6),(v_3,v_5,8),(v_4,v_6,5),(v_5,v_4,4),(v_6,v_1,7),(v_6,v_8,4),(v_6,v_7,3),(v_7,v_5,1),(v_7,v_8,7),(v_8,v_1,5),(v_8,v_9,3),(v_9,v_1,1)\}$ *(третий параметр в скобках - стоимость ребра).*

Задача 4.2. *Пусть* $S = (V,T)$ *– остовное дерево наименьшей стоимости, построенное алгоритмом* **МинОстов** *для нагруженного неориентированного графа* $G = (V,E)$ *с* n *вершинами. Пусть* $c_1 \le c_2 \le \ldots \le c_{n-1}$ *– это последовательность длин ребер из* T, *упорядоченных по возрастанию. Пусть S' – произвольное остовное дерево для* G *с длинами ребер* $d_1 \le d_2 \le \ldots \le d_{n-1}$. *Показать, что* $c_i \le d_i$ *для всех* $i : 1 \le i \le n-1$.

Задача 4.3. *Пусть* e *– ребро максимального веса в некотором цикле графа* $G = (V,E)$. *Докажите, что существует минимальный остов графа* $G' = (V, E \setminus \{e\})$, *который является также минимальным остовом графа* G.

Задача 4.4. *Пусть задано минимальное остовное дерево* $S = (V,T)$ *графа* $G = (V,E)$ *и новое ребро* (u,v) *веса* d, *добавляемое к* G. *Предложите алгоритм сложности* $O(|V|)$, *который строит минимальное остовное дерево графа* $G_1 = (V, E \cup \{(u,v)\})$.

Задача 4.5. *Предложите эффективный алгоритм для построения "минимального"остовного дерева для неориентированного взвешенного графа $G = (V, E)$ с весовой функцией $c : E \to R$, если "весом"дерева $S = (V, T)$ считать вес самого тяжелого ребра S, $c(S) = \max\{c(e)|e \in T\}$.*

Задача 4.6. Алгоритм Прима (1957) *для построения минимального остова отличается от алгоритма Крускала тем, что вместо очереди ребер Q поддерживает очередь с приоритетами для вершин QV. Приоритетом вершины в ней является минимальная длина ребра, соединяющего данную вершину с вершиной уже построенного дерева. Алгоритм начинает построение с произвольной вершины $v_0 \in V$ и помещает ее в множество вершин дерева V_T. Остальные вершины помещаются в очередь QV. На каждом этапе алгоритма Прима из QV выбирается вершина v с минимальной длиной ребра (v, u) до вершины $u \in V_T$. Эта вершина v удаляется из QV и добавляется к V_T, ее отцом в дереве становится u.*
1) Напишите реализацию алгоритма Прима на псевдокоде.
2) Докажите корректность алгоритма Прима.
3) Докажите, что его можно реализовать так, чтобы общее время работы не превосходило:
а) $O(|V|^2$ (при непосредственном поиске вершины v);
б) $O(|V| \log_2 |V| + |E| \log_2 |V|)$ (при использовании двоичной кучи для реализации очереди с приоритетами);
в) $O(|V| \log_2 |V| + |E|)$ (при использовании фиббоначиевой кучи для реализации очереди с приоритетами).

Задача 4.7. Алгоритм Борувки (1926) *основан на следующем простом утверждении:* Для каждой вершины ребро наименьшего веса, инцидентное этой вершине, должно входить в минимальное остовное дерево. *(Докажите это утверждение.)*

Поэтому ребра из этого утверждения образуют остовной лес, состоящий из не более чем $|V|/2$ деревьев. Для каждого из этих деревьев T ребро наименьшего веса (v, w), такое что $v \in T$, а $w \notin T$ также должно входить в минимальное остовное дерево. Алгоритм Борувки на первом этапе строит остовной лес из ребер наименьшего веса, инцидентных каждой вершине. Затем на очередном этапе к нему добавляюся ребра минимального веса, "торчащие"из каждого дерева. При этом число деревьев сокращается по крайней мере вдвое.
1) Напишите реализацию алгоритма Борувки на псевдокоде.
2) Докажите корректность алгоритма Борувки.
3) Докажите, что время его работы не превосходит $O(|E| \log_2 |V|)$.

5 Алгоритмы обхода графов

5.1 Поиск в глубину на неориентированном графе и задача о лабиринте

Задачу поиска выхода из лабиринта можно формализовать так: лабиринт — это неориентированный граф, вершины которого представляют "перекрестки"лабиринта, а ребра — дорожки между соседними перекрестками. Одна или несколько вершин отмечены как выходы. Задача состоит в построении пути из некоторой исходной вершины в вершину-выход.

В этом разделе мы рассмотрим метод обхода всех вершин графа, называемый *поиском в глубину*. Его идею кратко можно описать так:

находясь в некоторой вершине v, идем из нее в произвольную еще не посещенную смежную вершину w, если такой вершины нет, то возвращаемся в вершину, из которой мы пришли в v.

Алгоритм поиска в глубину

Вход: $G = (V, E)$ – неориентированный граф, представленный списками смежностей: для каждой $v \in V$ список L_v содержит перечень всех смежных с v вершин.

Выход: $NUM[v]$ – массив с номерами вершин в порядке их прохождения и множество (древесных) ребер $T \subseteq E$, по которым осуществляется обход.

Алгоритм ПОГ

1. $T = \{\}$; $NOMER = 1$;
2. для каждой вершины $v \in V$ положим $NUM[v] = 0$ и пометим v как "новую";
3. **WHILE** существует "новая"вершина $v \in V$
4. **DO** ПОИСК(v);

Основную роль в этом алгоритме играет следующая рекурсивная процедура.

procedure ПОИСК(v):

5. пометить v как "старую";
6. $NUM[v] = NOMER$; $NOMER = NOMER + 1$;
7. **FOR EACH** $w \in L_v$ **DO**
8. **IF** вершина w "новая"
9. **THEN**
10. { $T := T \cup \{(v, w)\}$;
11. ПОИСК(w)
12. }

Теорема 5.1. *Алгоритм* **ПОГ** *обходит (нумерует) все вершины неориентированного графа* $G = (V, E)$ *за время* $O(\max(|V|, |E|))$. *Если* G *— связный граф, то* $S = (V, T)$ *— это остов* G, *если граф* G *не является связным, то* $S = (V, T)$ *— это остовной лес для* G, *т.е. объединение остовных деревьев для каждой из компонент связности* G.

Доказательство. Оценка времени следует из того, что процедура ПОИСК может вызываться для каждой вершины не более одного раза, а правильность является непосредственным следствием утверждения о работе процедуры ПОИСК.

Лемма 5.1. *Пусть* $G_1 = (V_1, E_1)$ *— компонента связности графа* G *и* $v_1 \in V_1$ *— первая вершина, для которой в стр. 4 вызывается процедура ПОИСК*(v_1). *Тогда*
а) в процессе выполнения этого вызова процедура ПОИСК будет вызвана один раз для любой вершины $w \in V_1$;
б) после завершения этого вызова все вершины из V_1 *"старые"и имеют номера от* $NOMER$ *до* $NOMER + |V_1| - 1$;
в) добавленные к T *во время этого вызова ребра* T_1 *образуют остовное дерево для* G_1.

Доказательство. Пункт (а) доказывается индукцией по расстоянию от v_1 до w.

Если это расстояние равно 1, то $w \in L_{v_1}$ и рассматривается в цикле в стр.7. Если она в этот момент "старая то значит ПОИСК(w) уже вызывался. Если же w "новая то в стр.11 происходит вызов ПОИСК(w).

Предположим теперь, что ПОИСК(u) вызывается для всех вершин u, находящихся на расстоянии $k \geq 1$ от v_1, и пусть вершина $w \in L_{v_1}$ находится на рсстоянии $(k + 1)$ от v_1. Тогда имеется путь длины $(k + 1)$ от v_1 до w. Пусть u — это предпоследняя вершина на этом пути. Тогда расстояние от v_1 до u равно k и по нашему предпроложению в некоторый момент выполняется вызов ПОИСК(u). Так как $w \in L_u$, то в этом вызове вершина w в некоторый момент рассматривается в цикле в стр.7. Как и выше, если она в этот момент "старая то ПОИСК(w) уже вызывался. Если же w еще "новая то в стр.11 происходит вызов ПОИСК(w).

Пункт (б) непосредственно следует из (а), а циклы в T отсутствуют, так как каждое добавляемое ребро ведет из "старой"вершины в "новую".
□.

Дерево $S = (V, T)$, которое строится алгоритмом **ПОГ,** называется *глубинным остовом* или *глубинным остовным деревом* графа G. Ребра, попавшие в множество T, называются *прямыми,* а не попавшие в это множество ребра из множества $(E \setminus T)$ — *обратными.* Каждое обратное ребро (v, w) соединяет вершину v с ее предком w в глубинном остове (см. задачу 5.6). Поэтому в исходном графе G оно определяет цикл: от w к v по ребрам дерева T, а затем обратно от v к w по ребру (v, w). Поэтому алгоритм **ПОГ** можно использовать для проверки наличия циклов в G. Ребро (v, w) не добавляется к T, т.е. является обратным, тогда и только тогда, когда в стр.4 вызова ПОИСК(v) обнаруживается, что вершина w "старая". Поэтому, добавив в процедуру ПОИСК(v) последнюю строку

13. **ELSE** ПЕЧАТЬ (v, w),

мы получим процедуру, которая в дополнение построению к глубинного остова графа будет распечатывать список всех обратных ребер. Если этот список не пуст, то в графе имеются циклы, иначе — нет.

Алгоритм поиска в глубину часто используется как основа для различных алгоритмов обработки графов. Вместо строки 6, в котрой вершина v получает номер $NUM(v)$, можно вставить вызов любой процедуры, обрабатывающей информацию, связанную с этой вершиной (например, для задачи о лабиринте это может быть проверка того, что v является выходом из лабиринта). И тогда полученный вариант алгоритма обеспечит обработку всех вершин графа. Ниже мы рассмотрим несколько применений этого алгоритма.

Определение 15. *Ребро (v, w) неориентированного графа $G = (V, E)$ называется* мостом *G, если при его удалении из E число связных компонент графа увеличивается, т.е. в графе $G' = (V, E \setminus \{(v, w)\}$ связных компонент больше, чем в G.*

Из этого определения, в частности, следует, что ребро является мостом тогда и только тогда, когда оно не входит ни в какой цикл *(почему?).* Поиск в глубину позволяет находить все мосты графа. Во-первых, заметим, что любой мост (v, w) является ребром глубинного остова, так как другого пути, связывающего v и w нет. Во-вторых, если это ребро ориентировано от v к w, то в глубинном остове нет обратных ребер, соединяющих w и его потомки с предками w. Это условие является и достаточным, так как, если таких ребер нет, то удаление (v, w) нарушит связь между v и w и они окажутся в разных компонентах связности. Обозначим через ВЕРХ(w) минимум из $NUM[w]$ и наименьшего из номеров вершин, к которым ведут обратные ребра от вершин поддерева T_w. Тогда, учитывая, что обратные ребра соединяют потомков с предками и что номера предков меньше номеров потомков, предложенный критерий можно переформулировать в следующем виде.

Теорема 5.2. *Ребро (v, w) глубинного остова $D = (V, E)$ неориентированного графа $G = (V, E)$ является мостом G тогда и только тогда, когда $BEPX(w) > NUM[v]$ или, что эквивалентно, $BEPX(w) = NUM[w]$.*

Вычисление значения ВЕРХ(w) можно организовать в процессе обхода в глубину, используя следующее соотношение:
ВЕРХ$(w) = \min\{\{NUM[w]\} \cup \{$ВЕРХ$(z) \mid (w, z)-$ *прямое ребро* $\} \cup \{NUM[u] \mid (w, u)--$ *обратное ребро* $\}\}$.
Для этого достаточно в строку 6 алгоритма **ПОГ** добавить начальное присвоение
ВЕРХ$(v) := NUM[v]$,
в строке 11 приписать после ПОИСК(w) присвоение
ВЕРХ$(v) :=$ min{ВЕРХ(v), ВЕРХ(w)},

учитывающее прямые ребра, и добавить строку
13. **ИНАЧЕ** ВЕРХ(v) := $\min\{$ВЕРХ(v), $NUM(w)\}$,
для учета обратных ребер.
Зная значения ВЕРХ(w), нетрудно выявить все мосты, используя критерий из теоремы 5.2.

Пример 9. *Применим алгоритм* **ПОГ** *к графу G_2, изображенному на следующем рисунке.*

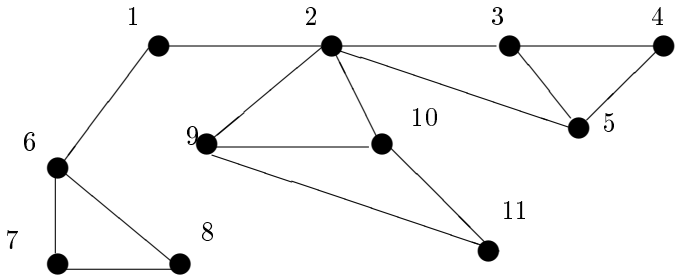


Рис. 15: Граф G_2

Его представление в виде списков смежности имеет следующий вид:

L_1 : 6, 2	L_7 : 6, 8
L_2 : 1, 9, 10, 3	L_8 : 6, 7
L_3 : 2, 5, 4	L_9 : 2, 10, 11
L_4 : 3, 5	L_{10} : 2, 9, 11
L_5 : 2, 3, 4	L_{11} : 9, 10
L_6 : 1, 7, 8	

Алгоритм **ПОГ** вызовет процедуру ПОИСК(1). Эта процедура рекурсивно вызовет ПОИСК(6) и т.д. Вот структура всех получающихся вызовов процедуры ПОИСК:
ПОИСК(1) $\Rightarrow$ ПОИСК(6) $\Rightarrow$ ПОИСК(7) $\Rightarrow$ ПОИСК(8)
$\Downarrow$
ПОИСК(2) $\Rightarrow$ ПОИСК(9) $\Rightarrow$ ПОИСК(10) $\Rightarrow$ ПОИСК(11)
$\Downarrow$
ПОИСК(3) $\Rightarrow$ ПОИСК(5) $\Rightarrow$ ПОИСК(4)

Вначале идут "горизонтальные"вызовы, затем возвраты справа налево и вызовы "по вертикали". В результате вершины G_2 получат следующие номера, отражающие порядок их прохождения:

V :	1	2	3	4	5	6	7	8	9	10	11
NUM :	1	5	9	11	10	2	3	4	6	7	8

Ребра остова T, построенные в процессе обхода графа, показаны на следующем рисунке. Стрелки указывают направление обхода.

В процессе построения этого дерева были определены следующие обратные ребра: $(8, 6)$, $(10, 2)$, $(11, 9)$, $(5, 2)$ и $(4, 3)$. Нетрудно проверить, что добавление любого из этих ребер к T приводит к образованию простого цикла.

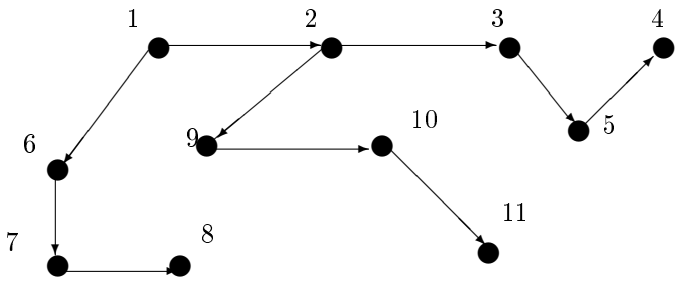


Рис. 16: Остовное "глубинное"дерево T графа G_2

Используя расширенный вариант **ПОГ** с вычислением функции ВЕРХ, мы получим следующий результат:

V:	1	2	3	4	5	6	7	8	9	10	11
NUM:	1	5	9	11	10	2	3	4	6	7	8
ВЕРХ:	1	5	5	9	5	2	2	2	5	5	6

Так как ВЕРХ(2) = $NUM[2] = 5$ и ВЕРХ(6) = $NUM[6] = 2$, то по теореме 5.2 мостами графа G_2 являются ребра $(1, 2)$ и $(1, 6)$ и других мостов у него нет.

5.2 Поиск в ширину на неориентированном графе

Идея алгоритма "поиска в ширину"состоит в том, чтобы, начав обход с некоторой вершины, в первую очередь посетить ее соседей, затем соседей соседей и т.д. Алгоритм поиска в ширину ПОШ отличается от алгоритма ПОГ только процедурой ПОИСКШ, в которой вместо стека, реализующего рекурсивные вызовы в ПОИСК, используется очередь вершин Q.

procedure ПОИСКШ(v):

5. создать пустую очередь Q;
6. ДОБАВИТЬ(Q, v); пометить v как "старую";
7. **WHILE** $Q \neq \emptyset$ **DO**
8. { $w :=$ НАЧАЛО(Q); УДАЛИТЬ(Q, w);
9. $NUM[w] := NOMER$; $NOMER := NOMER + 1$;
10. **FOR EACH** $u \in L_w$ **DO**
11. **IF** вершина u "новая"
12. **THEN** { ДОБАВИТЬ(Q, u);
13. пометить u как "старую" };
14. $T := T \cup \{(w, u)\}$
15. }

Следующее утверждение аналогично теореме 5.1 о свойствах поиска в глубину.

Теорема 5.3. *Алгоритм* **ПОШ** *обходит (нумерует) все вершины неориентированного графа* $G = (V, E)$ *за время* $O(\max(|V|, |E|))$. *Если* G *— связный граф, то* $S = (V, T)$ *— это остов* G,

если граф G не является связным, то $S = (V, T)$ — это остовной лес для G, т.е. объединение остовных деревьев для каждой из компонент связности G.

Эта теорема является непосредственным следствием следующей леммы.

Лемма 5.2. *Пусть $G_1 = (V_1, E_1)$ — компонента связности графа G и $v_1 \in V_1$ — первая вершина, для которой в стр. 4 вызывается процедура ПОИСКШ(v_1). Тогда*
а) в процессе выполнения этого вызова каждая вершина $w \in V_1$ попадет один раз в очередь Q;
б) после завершения этого вызова все вершины из V_1 "старые"и имеют номера от $NOMER$ до $NOMER + |V_1| - 1$;
в) добавленные к T во время этого вызова ребра T_1 образуют остовное дерево для G_1;
г) путь от v_1 до любой вершины $w \in V_1$ в дереве T_1 является кратчайшим (по числу ребер) путем между v_1 и w в графе G.

Доказательство. Лемма доказывается непосредственной индукцией по расстоянию от v_1 до w. Для доказательства пункта (г) предположим, что для каждой вершины u, находящейся на расстоянии $k \geq 0$ от v_1 в графе G, путь в T_1 от v_1 до u имеет длину k. Если вершина w находится на расстоянии $k + 1$ от v_1, то у нее есть соседи, находящиеся от v_1 на расстоянии k. Пусть u первая из этих вершин, попавшая (по индуктивному предположению) в очередь Q. Тогда в момент выбора u из Q вершина w еще новая и в T_1 добавится ребро (u, w). Следовательно, длина пути от v_1 до w в T_1 будет равна $k + 1$, т.е. это будет кратчайший путь.

Из этой леммы непосредственно получаем

Следствие. *Для связного графа $G = (V, E)$ алгоритм* **ПОШ** *строит остовное дерево $S = (V, T)$, которое является деревом кратчайших путей из его корня во все остальные вершины. Такие деревья иногда называют* геодезическими.

Пример 10. *Применим алгоритм* **ПОШ** *к графу G_2 из примера 9, изображенному на рис. 15. Напомним его представление в виде списков смежности:*

$$\begin{array}{ll} L_1: & 6, 2 \\ L_2: & 1, 9, 10, 3 \\ L_3: & 2, 5, 4 \\ L_4: & 3, 5 \\ L_5: & 2, 3, 4 \\ L_6: & 1, 7, 8 \end{array} \qquad \begin{array}{l} L_7 : 6, 8 \\ L_8 : 6, 7 \\ L_9 : 2, 10, 11 \\ L_{10} : 2, 9, 11 \\ L_{11} : 9, 10 \end{array}$$

Тогда при обходе в ширину получается следующая нумерация вершин G_2:

V:	1	2	3	4	5	6	7	8	9	10	11
NUM:	1	3	8	11	10	2	4	5	6	7	9

При этом обход происходит по дугам дерева T, изображенного на следующем рисунке.

5.3 Двусвязные компоненты неориентированных графов

Как мы отмечали выше, отсутствие мостов в графе означает невозможность разрушить представляемую им сеть связи за счет удаления одного ребра. Рассмотрим еще одно свойство графов - двусвязность, которое делает невозможным разрушение соответствующей сети связи за счет удаления одной вершины.

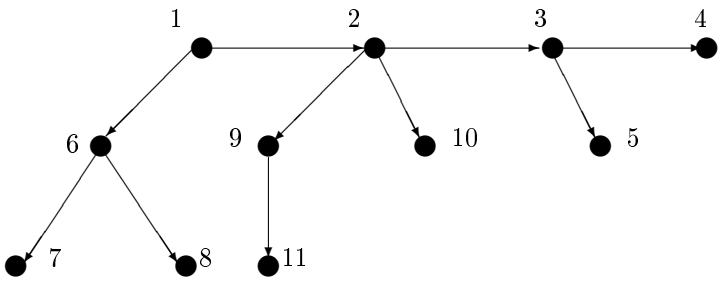


Рис. 17: Остов графа G_2, построенный алгоритмом ПОШ

Определение 16. *Неориентированный граф $G = (V, E)$ называется двусвязным, если для любых трех его попарно различных вершин w, v и a существует путь между w и v, не проходящий через a. Вершина v называется точкой сочленения (шарниром, точкой раздела) графа G, если для некоторой пары различных вершин x и y (не совпадающих с v) всякий путь между x и y проходит через v.*

Следствие. *Связный граф является двусвязным тогда и только тогда, когда в нем нет точек сочленения.*

Определение 17. *Максимальный двусвязный подграф графа G называется его двусвязной компонентой (иногда такие подграфы называются* блоками *графа).*

Заметим, что изолированные вершины и мосты являются двусвязными компонентами.

Содержательно, точки сочленения — это "слабые"места связного графа. Например, если граф представляет некоторую сеть связи, то повреждение пункта связи, соответствующего точке сочленения, приведет к разрыву сети, т.е. невозможности пересылать сообщения по всей сети.

Лемма 5.3. *Пусть $G_i = (V_i, E_i)$ $(i = 1, ..., k)$ - двусвязные компоненты графа $G = (V, E)$. Тогда*
1) граф G_i двусвязен $(i = 1, ..., k)$:
2) для любой пары $i \neq j$ $|V_i \cap V_j| \leq 1$;
3) a — точка сочленения $G \iff$ для некоторых i и j $\{a\} = V_i \cap V_j$.

Доказательство. Пункт (1) следует из определения.
(2) Если бы $|V_i \cap V_j| \geq 2$, то для любых трех попарно различных вершин w, v и a из $V_i \cup V_j$ имелся бы путь между w и v, не проходящий через a и, следовательно, эти вершины составляли бы одну двусвязную компоненту.
(3) Пусть a — точка сочленения G. Тогда существуют две вершины w и v, любой путь между которыми проходит через a. Рассмотрим произвольный такой путь $w, \ldots, w', a, v', \ldots, v$. Пусть $w' \in V_i$ и $v' \in V_j$. Тогда $V_i \neq V_j$ и $V_i \cap V_j = \{a\}$.
Обратно, пусть $\{a\} = V_i \cap V_j$. Тогда для любой пары вершин $w \in V_i$ $v \in V_j$ любой путь между ними проходит через a, т.е. a является точкой сочленения.

Следующее утверждение связывает точки сочленения с глубинным остовным деревом графа.

Лемма 5.4. *Пусть $G = (V, E)$ - связный неориентированный граф, а $S = (V, T)$ - глубинное остовное дерево. Вершина a является точкой сочленения G тогда и только тогда, когда*
1) a – корень дерева S и он имеет более одного сына или
2) a – не корень и у него есть сын s такой, что между ним и его потомками и предками a нет обратных ребер.

Доказательство. (1) Если a – корень дерева S и у него имеестя хотя бы два сына s_1 и s_2, то любой соединяющий их путь проходит через a и поэтому a является точкой сочленения G. Если же у a лишь один сын s_1, то между двумя любыми вершинами w и v, имеющими путь, проходящий через a, имеется путь, проходящий через s_1 и не включающий a. Следовательно, в этом случае a не является точкой сочленения.

(2) Пусть a – не корень и у него есть сын s такой, что между ним и его потомками и предками a нет обратных ребер. Тогда любой путь между отцом a и s проходит через a и, следовательно, a является точкой сочленения.

Предположим, что для каждого сына s вершины a имеется ребро (u_s, v_s), соединяющее его или некоторого его потомка u_s с каким-нибудь предком v_s вершины a. Пусть между w и v имеется путь, проходящий через a. Тогда, если w не является потомком a в S, а v входит в поддерево T_s с корнем s -сыном a, то имеется путь из w в корень S, оттуда в v_s, затем по ребру (v_s, u_s) в T_s, затем из u_s в s и, наконец из s в u_s. Если w и v являются потомками a в поддеревьях T_{s_1} и T_{s_2} разных сыновей s_1 и s_2 вершины a, то между ними существует путь $w \to \ldots \to s_1 \to \ldots \to u_{s_1} \to v_{s_1} \to \ldots \to v_{s_2} \to u_{s_2} \to \ldots \to s_2 \to \ldots \to v$. Во всех случаях эти пути не проходят через a. Следовательно, a не является точкой сочленения.

Пусть $NUM[v]$ - номер, присвоенный v алгоритмом ПОГ. Напомним определение функции ВЕРХ$[v]$), которая использовалась при определении мостов:
ВЕРХ$[v] = min\{\{NUM[v]\}, \{NUM[w] \mid$ существует такое обратное ребро (x, w), что x - потомок v, а w - предок v в глубинном остовном дереве $(V, T)\}\}$.

Следствие леммы 5.4: v - точка сочленения $\iff$ у v есть сын s, для которого ВЕРХ$[s] \geq NUM[v]$.

Из определения ВЕРХ следует, что
(*) ВЕРХ$[v] = min\{\{NUM[v]\} \cup \{$ВЕРХ$[s] | s-$ сын $v\} \cup \{NUM[w] |$ $(v, w) -$ обратное ребро$\}\}$.

Следующий алгоритм выделения двусвязных компонент, предложенный Хопкрофтом, основан на поиске в глубину, в процессе которого все рассматриваемые ребра попадают (по одному разу) в стек S и для каждой вершины v вычисляется значение функции ВЕРХ$[v]$ (по той же схеме, что и при поиске мостов). При обнаружении точки сочленения v все ребра, помещенные в S после перехода к ее сыну w, удовлетворяющему условию из п. (2) леммы 5.4, составляют очередную двусвязную компоненту C_i. Они выталкиваются из S и помещаются в C_i.

Алгоритм ДВУСВЯЗ
Вход: связный неориентированный граф $G = (V, E)$, заданный посредством списков смежности L_v, вершина $v_0 \in V$.
Выход: списки ребер двусвязных компонент G.

1. $NOMER := 0$; $i := 0$; СоздатьСтек(S); $T := \emptyset$;
2. **FOR EACH** $v \in V$ **DO** $\{NUM[v] := 0$; отметить v как "новую"$\}$;
3. **FOR EACH** $(v, w) \in E$ **DO** $mark(v.w) := 0$; /* (v, w) не помещалось в S
4. ПОИСКБ(v_0).

Procedure ПОИСКБ(v)

```
1.  NOMER := NOMER + 1;  NUM[v] := NOMER;   ВЕРХ[v] := NUM[v];
2.  Пометить v как "старую";
3.  FOR EACH w ∈ L_v DO
4.  {   IF mark(v,w) = 0 THEN ВТОЛК(S,(v,w));  mark(v,w) := 1   END-IF;
5.      IF w - "новая" THEN   /* (v,w) прямое ребро
6.          {   T := T ∪ (v,w) ;
7.              ОТЕЦ[w] := v;
8.              ПОИСКБ[w];
9.              IF ВЕРХ[w] ≥ NUM[v] THEN /* v -точка сочленения
        /* Очередная компонента: вытолкнуть из S все ребра до (v,w) включительно
10.             {i := i + 1; C_i = ∅;
11.             DO {(x,y) := ВЕРХ(S);  C_i := C_i ∪ {(x,y)};  ВЫТОЛК(S)}
12.             UNTIL (x,y) = (v,w)
13.             }
14.             END-IF;
15.             ВЕРХ[v] := min{ВЕРХ[v], ВЕРХ[w]}
16.         }
17.     ELSE /* (v,w) - обратное ребро
18.         IF ОТЕЦ[v] ≠ w THEN
19.         ВЕРХ[v] := min{ВЕРХ[v], NUM[w]} END-IF
20.     END-IF
21.     }
```

Теорема 5.4. *Алгоритм ДВУСВЯЗ правильно находит двусвязные компоненты графа G за время $O(|E|)$.*

Доказательство. Вначале заметим, что алгоритм для каждой вершины $v \in V$ вычисляет значение функции ВЕРХ[v] в соответствии с определением (*). Действительно, в стр. 1 оно получает значение $NUM[v]$, в стр.15 для него выбирается минимум из текущего значения и значения ВЕРХ[w] для каждого сына w вершины v, а в стр. 20 учитывается $NUM[w]$ для обратных ребер (v,w). Таким образом, тест в стр. 9 верно идентифицирует точки сочленения.

Докажем теперь индукцией по числу k двусвязных компонент графа G, что алгоритм правильно выдает двусвязные компоненты.

Базис. При $k = 1$ граф G является двусвязным. Точек сочленения в нем нет. Поэтому вершина v_0 – корень остовного дерева – имеет в нем лишь одного сына. Пусть это v_1. Тогда первым в стек помещается ребро (v_0, v_1) и вызывается ПОИСКБ[v_1]. Этот вызов возвращает ВЕРХ[v_1] = 1, так как среди потомков v_1 в дереве есть вершины, связанные ребрами с v_0. Затем в стр. 9 выясняется, что ВЕРХ[v_1] = $1 \geq NUM[v_0] = 1$, и из стека выталкиваются все ребра G, поскольку ни одно из них до этого не выталкивалось.

Шаг индукции. Предположим, что алгоритм правильно выдает двусвязные компоненты для всех графов с их числом k. Рассмотрим его работу на графе G состоящем из $(k+1)$-ой компоненты. Пусть v – это первая вершина, для которой в процессе работы будет выполнено условие в стр. 9, т.е. первая выявленная точка сочленения и пусть w – это тот ее сын, для котрого оно выполнилось. Так как после помещения в стек S ребра (v,w) ни одно ребро из него не выталкивалось, то в нем находятся все ребра графа, соединяющие между собой вершины, достижимые из w, а также ребра, соединяющие эти вершины с v. Других ребер там нет, так у вершин, достижимых из w, нет обратных ребер, ведущих "выше"v. Таким образом, в цикле в стр. 11,12 из S будут вытолкнуты все ребра компоненты двусвязности C_1, содержащей ребро (v,w). Далее

алгоритм будет работать на G так же, как он работал бы на графе G', полученном из G после удаления компоненты C_1 (только номера вершин будут далее в G больше номеров для G' на число вершин в C) . Так как G' содержит k компонент, то по предположению индукции все его компоненты будут выданы верно. Следовательно, и все компоненты G будут выданы верно.

Для оценки времени заметим, что алгоритм ДВУСВЯЗ основан на обходе в глубину, дополненном вычислением функции ВЕРХ$[v]$ для всех вершин и операциями со стеком S. Число операций взятия минимума при вычислении ВЕРХ$[v]$ не превосходит числа ребер, инцидентных v. Каждое ребро графа G вталкивается в S и выталкивается из него по одному разу. Кроме того, так как G связный, то $|V| \leq |E|+1$. Отсюда выводим, что время работы алгоритма ДВУСВЯЗ ограничено $O(|E|)$.

Пример 11. *Применим алгоритм* **ДВУСВЯЗ** *к графу G_2 из примера 9, изображенному на рис. 15. Еще раз напомним его представление в виде списков смежности:*

$L_1:\ 6,2$ $\quad$ $L_7: 6,8$
$L_2:\ 1,9,10,3$ $\quad$ $L_8: 6,7$
$L_3:\ 2,5,4$ $\quad$ $L_9: 2,10,11$
$L_4:\ 3,5$ $\quad$ $L_{10}: 2,9,11$
$L_5:\ 2,3,4$ $\quad$ $L_{11}: 9,10$
$L_6:\ 1,7,8$

В процессе работы алгоритма ДВУСВЯЗ для всех вершин будут определены те же номера и значения функции ВЕРХ, что и в примере 9:

V:	1	2	3	4	5	6	7	8	9	10	11
NUM:	1	5	9	11	10	2	3	4	6	7	8
ВЕРХ:	1	5	5	9	5	2	2	2	5	5	6

При этом при возврате из ПОИСКБ(7) в ПОИСКБ(6) в стр. 9 будет выполнено условие ВЕРХ[7] $\geq NUM[6]$, а стек будет иметь вид: $S = (1,6)(6,7)(7,8)(6,8)$ (дно стека - слева, а верх стека - справа). В стр. 11, 12 в первую компоненту C_1 из S будут перемещены ребра до $(6,7)$ включительно, т.е. $C_1 = \{(6,7),(7,8),(6,8)\}$. Затем завершится вызов ПОИСКБ(6) и будет обнаружено, что ВЕРХ[6] $\geq NUM[1]$, после чего в C_2 попадет ребро $(1,6)$, а стек S опустошится. Далее последуют вызовы ПОИСКБ(2) и ПОИСКБ(9). После завершения второго из них $S = (1,2)(2,9)(9,10)(10,2)(10,11)(11,9)$ и в стр. 9 обнаружится, что ВЕРХ[9] $\geq NUM[2]$. В результате сформируется компонента $C_3 = \{(2,9)(9,10)(10,2)(10,11)(11,9)\}$, а в стеке останется ребро $(1,2)$. Затем последует вызов ПОИСКБ(3), при завершении которого стек $S = (1,2)(2,3)(3,5)(5,2)(5,4)(4,3)$ и выполняется условие ВЕРХ[3] $\geq NUM[2]$. Поэтому в стр. 11,12 сформируется компонента $C_4 = \{(2,3)(3,5)(5,2)(5,4)(4,3)\}$, а в S останется ребро $(1,2)$. После завершения вызова ПОИСКБ(2) выполнено условие ВЕРХ[2] $\geq NUM[1]$ и создается последняя компонента $C_5 = \{(1,2)\}$.

5.4 Компоненты сильной связности и базы ориентированного графа

Понятие сильной связности для ориентированных графов соответствует отношению взаимной достижимости вершин.

Определение 18. Сильно связные компоненты ориентированных графов.
Ориентированный граф $G = (V,E)$ называется сильно связным, если для любой пары его вершин v и w в G имеется путь из v в w и путь из w в v. Сильно связной компонентой G называется максимальный сильно связный подграф графа G.

Определить все сильно связные компоненты можно, используя вариант алгоритма поиска в глубину **ПОГ+пост**, в котором в процедуре ПОИСК+пост(v) вершина v получает номер перед выходом из процедуры (т.е. операторы строки 6 стоят после строки 12).

procedure **ПОИСК+пост(v)**:

5. пометить v как "старую";
6. **FOR EACH** $w \in L_v$ **DO**
7. **IF** вершина w "новая"
8. **THEN**
9. { $T := T \cup \{(v, w)\}$;
10. ПОИСК+пост(w)
11. };
12. $NUM[v] = NOMER;\ NOMER = NOMER + 1$

Алгоритм **ПОГ+пост** вызывается на первом этапе следующего алгоритма **ССК**, возвращающего все сильно связные компоненты исходного графа. Этот алгоритм был предложен Косерейю (R. Kosaraju) в 1978 г.

Алгоритм ССК:

Вход: ориентированный граф $G = (V, E)$, заданный посредством списков смежности L_v, $v \in V$.

Выход: списки вершин сильно связных компонент G.

1. Выполнить ПОГ +пост на G и для каждой вершины v определить ее номер $NUM[v]$;
2. Построить граф $G' = (V, E')$, "перевернув"стрелки: $E' = \{(v, w) | (w, v) \in E\}$;
3. $W := V$;
4. **WHILE** $W \neq \emptyset$ **DO**
5. { выбрать $w \in W$ с максимальным номером $NUM[w]$;
6. выполнить ПОИСК(w) на G' и выдать пройденные вершины V_w в качестве очередной компоненты;
7. $W := W \setminus V_w$}.

Теорема 5.5. *Алгоритм* **ССК** *корректно выдает все сильно связные компоненты ориентированного графа* $G = (V, E)$ *за время* $O(|V| + |E|)$.

Доказательство. Доказательство правильности алгоритма **ССК** оставляем в качестве задачи (см. задачу 5.10). Для оценки времени его работы заметим, что пункты 1 - 3, очевидно, выполняются за время $O(|V| + |E|)$. В строках 6 и 7 цикла **WHILE** каждое ребро графа G' рассматривается не более двух раз, а каждая вершина проходится (нумеруется в ПОИСК) и удаляется 1 раз. Поэтому общее время выполнения этих строк также $O(|V|+|E|)$. Что касается определения очередной вершины с максимальным номером в строке 5, то для его эффективного выполнения достаточно при присвоении номера очередной вершине в алгоритме ПОГ +пост делать ссылку на вершину, получившую предыдущий номер. Таким образом по завершении этого алгоритма образуется список вершин в порядке убывания их номеров, который можно использовать для выбора очередной вершины в строке 5. При этом общее время выполнения этой строки не превысит $O(|V|)$.

Определим для ориентированного графа $G = (V, E)$ его *граф компонент* $G^K = (V^K, E^K)$ следующим образом:

$V^K = \{C \mid C -$ компонента сильной связности $G\}$,
$E^K = \{(C_1, C_2) \mid$ существует такое ребро $(v,u) \in E,$ что $v \in C_1, u \in C_2\}$.

Граф G^K называют *конденсацией* графа G. Используя алгоритм **ССК**, конденсацию G^K графа G можно построить за время $O(|V| + |E|)$ (задача 5.12).

Из определения непосредственно следует, что в графе G^K нет циклов, а отношение достижимости на нем является отношением частичного порядка $\leq_K$: оно рефлексивно, антисимметрично и транзитивно.

Назовем сильно связную компоненту K графа G *минимальной,* если она минимальна относительно порядка $\leq_K$, т.е. не достижима ни из какой другой компоненты G. Из этого определения следует, что K минимальна тогда и только тогда, когда в графе G^K нет ребер, входящих в K.

Минимальные компоненты в графе G^K, очевидно, можно найти по его спискам смежности за время $O(|V^K| + |E^K|)$.

Во многих приложениях ориентированный граф представляет собой сеть распространения некоторого ресурса: продукта, товара, информации и т.п. В таких случаях естественно возникает задача поиска минимального числа таких точек (вершин), из которых этот ресурс может быть доставлен в любую точку сети.

Определение 19. *Пусть $G = (V, E)$ — ориентированный граф. Подмножество вершин $W \subseteq V$ называется* порождающим*, если из вершин W можно достичь любую вершину графа. Подмножество вершин $W \subseteq V$ называется* **базой** *графа, если оно является порождающим, но никакое его собственное подмножество порождающим не является.*

Следующая теорема связывает базы графа с его минимальными компонентами и позволяет эффективно находить все базы графа.

Теорема 5.6. *Пусть $G = (V, E)$ — ориентированный граф. Подмножество вершин $W \subseteq V$ является базой G тогда и только тогда, когда оно содержит по одной вершине из каждой минимальной компоненты сильной связности G и не содержит никаких других вершин.*

Доказательство. Заметим вначале, что каждая вершина графа достижима из вершины, принадлежащей некоторой минимальной компоненте. Поэтому множество вершин W, содержащих по одной вершине из каждой минимальной компоненты, является порождающим а при удалении из него любой вершины перестает быть таковым, так как вершины из соответствующей минимальной компоненты становятся недостижимы. Поэтому W является базой.

Обратно, если W является базой, то оно обязано включать хотя бы по одной вершине из каждой минимальной компоненты, иначе вершины такой минимальной компоненты окажутся недоступны. Никаких других вершин W содержать не может, так как каждая из них достижима из уже включенных вершин.

Из этой теоремы вытекает следующая процедура построения одной или перечисления всех баз графа G.
1) Используя алгоритм **ССК**, найти все компоненты сильной связности G.
2) Построить конденсацию G^K и выделить среди ее вершин все минимальные компоненты G.
3) Породить одну или все базы графа, выбирая по одной вершине из каждой минимальной компоненты.

Теорема 5.7. *Существует алгоритм построения базы ориентированного графа $G = (V, E)$ за время $O(|V| + |E|)$. Все базы G можно перечислить за время $O(k|V| + |E|))$, где k – это число различных баз G.*

Пример 12. *Определим все базы ориентированного графа G, показанного на рис. 18. Будем считать, что в списках смежности вершины идут в алфавитном порядке (например, $L_e = d, g, h$).*

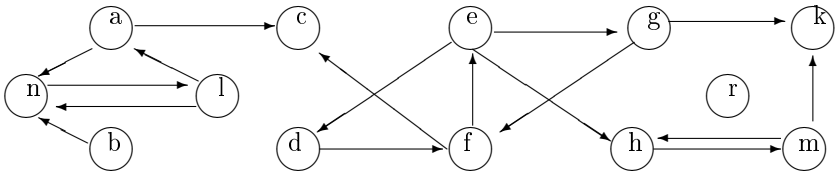


Рис. 18: Граф G

Вначале ищем компоненты сильной связности с помощью алгоритма ССК. На его первом этапе после выполнения ПОГ+пост получим следующую нумерацию вершин:

V:	a	b	c	d	e	f	g	h	k	l	m	n	r
NUM:	4	5	1	12	10	11	7	9	6	2	8	3	13

Затем строим граф $G' = (V, E')$ (см. рис. 19), обратив направления ребер в G

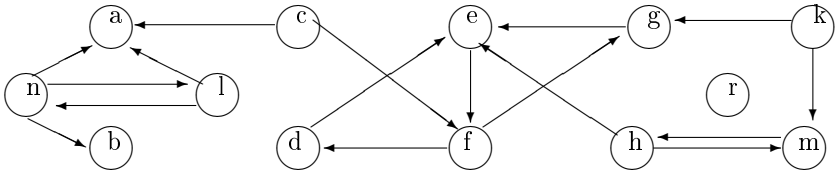


Рис. 19: Граф $G' = (V, E')$

На этом графе последовательно запускаем ПОИСК на вершинах с максимальными номерами, удаляя после каждого такого вызова достигнутые в нем вершины. Получаем следующие семь компонент сильной связности G:
$K_1 = \{r\}, K_2 = \{d, e, f, g\}, K_3 = \{h, m\}, K_4 = \{k\}, K_5 = \{b\}, K_6 = \{a, n, l\}, K_7 = \{c\}$.

На втором этапе строим конденсацию G^K на этих компонентах.

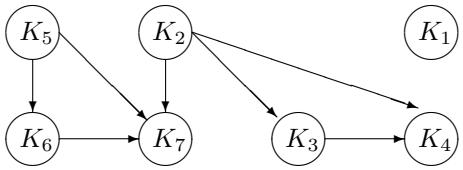


Рис. 20: Граф конденсации G^K на компонентах G

С помощью этого графа определяем минимальные компоненты:
$K_1 = \{r\}, K_2 = \{d, e, f, g\}$ и $K_5 = \{b\}$.

Наконец перечисляем все четыре базы G:
$B_1 = \{r, d, b\}$, $B_2 = \{r, e, b\}$, $B_3 = \{r, f, b\}$ и $B_4 = \{r, g, b\}$.

5.5 Задачи

Задача 5.1. *Пусть неориентированный граф $G = (V, E)$ имеет k компонент связности. Доказать, что тогда*

$$|E| \leq (|V| - k)(|V| - k + 1)/2.$$

Задача 5.2. *Модифицировать алгоритм ПОГ (ПОШ) так, чтобы он определял компоненты связности графа G (например, приписывал каждой вершине v номер ее компоненты связности КОМП[v]).*

Задача 5.3. *Доказать, что построенное ПОГ множество ребер T образует остовной лес G.*

Задача 5.4. *Пусть в результате работы ПОГ ребро (v, w) оказалось обратным, т.е. оно принадлежит $E \backslash T$. Тогда либо v — предок w, либо w — предок v в построенном ПОГ остовном лесу.*

Задача 5.5. *Рассмотрите модификацию алгоритма ПОГ (ПОШ) для ориентирован- ного графа $G = (V, E)$, в которой $L_v = \{w|\ (v, w) \in E\}$. Покажите, что полученный алгоритм нумерует все вершины и строит остовной лес.*

Задача 5.6. *Пусть в результате алгоритма ПОГ на ориентированном графе G ребро (v, w) оказалось поперечным, т.е. v и w не являются потомками друг друга в T. Доказать, что тогда $NUM[v] > NUM[w]$.*

Задача 5.7. *Пусть F - это лес, построенный алгоритмом ПОГ для ориентированного графа $G = (V, E)$. Доказать, что G - ациклический граф $\iff$ $(E \setminus F)$ не содержит дуги (v, w), в которой вершина w является предком вершины v в F.*

Задача 5.8. *Пусть в неориентированном графе $G = (V, E)$ длины всех ребер одинаковы и равны d. Модифицируйте алгоритм* **ПОШ** *так, чтобы он для заданной вершины $v \in V$ определил кратчайшие пути и расстояния до остальных вершин графа.*

Задача 5.9. Топологическая сортировка.
Для ориентированного ациклического графа $G = (V, E)$ назовем линейный порядок $<$ на множестве вершин V топологическим, если из $v < w$ следует, что в G нет пути из w в v (вообще говоря, этот порядок не единственный). Доказать, что ПОГ+пост нумерует вершины G в порядке, обратном топологическому.

Задача 5.10. *Доказать, что алгоритм ССК корректно выдает все сильно связные компоненты G.*

Задача 5.11. *Предположим, что в алгоритме ССК из предыдущей задаче в цикле 4-7 вместо графа $G' = (V, E')$ с перевернутыми стрелками используется исходный граф G, а вершины рассматриваются в порядке увеличения их номеров. Будет ли такой алгоритм работать правильно?*

Задача 5.12. *Предложите алгоритм, который по ориентированному графу $G = (V, E)$ строит его граф сильно связных компонент (конденсацию) G^K за время $O(|V| + |E|)$.*

Задача 5.13. *Определить для приведенного на рис. 21 ориентированного графа G его компоненты сильной связности, конденсацию G^K и все базы графа G.*

Задача 5.14. *Докажите, что связный неориентированный граф является деревом тогда и только тогда, когда каждое его ребро является мостом.*

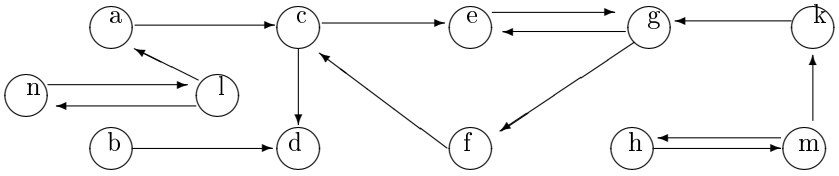


Рис. 21: Граф G

Задача 5.15. *Докажите, что двусвязная компонента - это максимальный набор ребер, любые два ребра которого принадлежат общему простому циклу.*

Задача 5.16. *Найти все мосты и двусвязные компоненты заданного неориентированного графа* $G = (V, E)$
$V = \{v_1, v_2, v_3, v_4, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$,
$E = \{(v_1, v_2), (v_1, v_4), (v_1, v_8), (v_7, v_8), (v_2, v_9), (v_9, v_{11}), (v_4, v_2), (v_3, v_8), (v_6, v_3), (v_3, v_7), (v_6, v_7), (v_{10}, v_9), (v_{10}, v_{11})\}$.

6 Задачи о путях на графе

Задачи о достижимости в графе одних вершин из других и о кратчайших путях между вершинами в графах с заданными длинами (весами) ребер занимают важное место в многочисленных приложениях теори графов. В этом разделе мы вначале рассмотрим алгоритмы для проверки достижимости и вычисления длин кратчайших путей между всеми парами вершин, а затем – алгоритмы построения дерева кратчайших путей из заданной вершины до достижимых из нее вершин графа.

6.1 Достижимость и транзитивное замыкание графа

Определение 20. *Пусть* $G = (V, E)$ *— ориентированный граф.* **Граф достижимости** $G^* = (V, E^*)$ *для* G *имеет то же множество вершин* V *и следующее множество ребер* $E^* = \{(u, v) \mid$ *в графе* G *вершина* v *достижима из вершины* $u\}$.

Иначе говоря, отношение E^* *является рефлексивным и транзитивным замыканием отношения* E.

Пример 13. *Рассмотрим следующий граф* G*:*

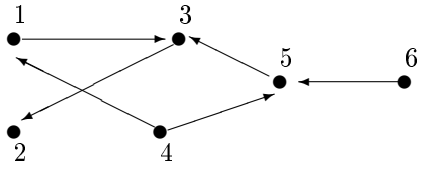


Рис. 22: Граф G

Тогда можно проверить, что граф достижимости G^* *для* G *выглядит так (ребра-петли при каждой из вершин не показаны):*

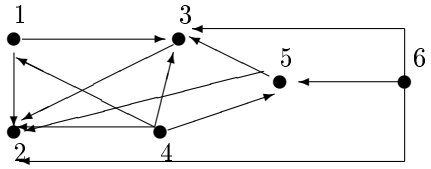


Рис. 23: Граф G^*

Каким образом по графу G можно построить граф G^*? Один способ заключается в том, чтобы для каждой вершины графа G определить множество достижимых из нее вершин, последовательно добавляя в него вершины, достижимые из нее путями длины 0, 1, 2 и т.д. Для этого можно использовать алгоритм поиска в ширину.

Другой (алгебраический) способ основан на использовании матрицы смежности A_G графа G и булевых операций. Пусть множество вершин $V = \{v_1, \ldots, v_n\}$. Напомним, что матрица смежности A_G — это булева матрица размера $n \times n$ такая, что

$$a_{ij} = \begin{cases} 1, & (v_i, v_j) \in E \\ 0 & \text{в противном случае} \end{cases}$$

Обозначим через E_n единичную матрицу размера $n \times n$. Положим $\tilde{A} = A_G \vee E_n$. Определим "булевы" степени этой матрицы, используя дизъюнкцию $\vee$ в качестве сложения, а конъюнкцию $\wedge$ в качестве умножения (будем для нее использовать обычную точку: $\cdot$).

Пусть $\tilde{A}^0 = E_n, \tilde{A}^1 = \tilde{A}, \ldots, \tilde{A}^{k+1} = \tilde{A}^k \cdot \tilde{A}$. Наша процедура построения G^* основана на следующем утверждении.

Лемма 6.1. *Пусть* $\tilde{A}^k = (a_{ij}^{(k)})$. *Тогда*

$$a_{ij}^{(k)} = \begin{cases} 1, & \textit{в графе } G \textit{ из } v_i \textit{ в } v_j \textit{ имеется путь длины } \leq k \\ 0 & \textit{в противном случае} \end{cases}$$

Доказательство проведем индукцией по k.

Базис. При $k = 0$ и $k = 1$ утверждение справедливо по определению $\tilde{A}^0$ и $\tilde{A}^1$.

Индукционный шаг. Пусть лемма справедлива для k. Покажем, что она остается справедливой и для $k + 1$. По определению $\tilde{A}^{k+1}$ имеем:

$$a_{ij}^{(k+1)} = a_{i1}^{(k)} a_{1j}^{(1)} \vee \ldots \vee a_{ir}^{(k)} a_{rj}^{(1)} \vee \ldots a_{in}^{(k)} a_{nj}^{(1)}.$$

Предположим, что в графе G из v_i в v_j имеется путь длины $\leq k + 1$. Рассмотрим кратчайший из таких путей. Если его длина $\leq k$, то по предположению индукции $a_{ij}^{(k)} = 1$. Кроме того, $a_{jj}^{(1)} = 1$. Поэтому $a_{ij}^{(k)} a_{jj}^{(1)} = 1$ и $a_{ij}^{(k+1)} = 1$. Если длина кратчайшего пути из из v_i в v_j равна $k + 1$, то пусть v_r — его предпоследняя вершина. Тогда из v_i в v_r имеется путь длины k и по предположению индукции $a_{ir}^{(k)} = 1$. Так как $(v_r, v_j) \in E$, то $a_{rj}^{(1)} = 1$. Поэтому $a_{ir}^{(k)} a_{rj}^{(1)} = 1$ $a_{ij}^{(k+1)} = 1$.

Обратно, если $a_{ij}^{(k+1)} = 1$, то хотя бы для одного r слагаемое $a_{ir}^{(k)} a_{rj}^{(1)}$ в сумме равно 1. Если это $r = j$, то $a_{ij}^{(k)} = 1$ и по индуктивному предположению в G имеется путь из v_i в v_j длины $\leq k$. Если же $r \neq j$, то $a_{ir}^{(k)} = 1$ и $a_{rj}^{(1)} = 1$. Это означает, что в G имеется путь из v_i в v_r длины $\leq k$ и ребро $(v_r, v_j) \in E$. Объединив их, получаем путь из v_i в v_j длины $\leq k + 1$.

□

Из леммы 6.1 непосредственно получаем

Следствие 1. *Пусть $G = (V, E)$ — ориентированный граф с n вершинами, а G^* — его граф достижимости. Тогда $A_{G^*} = \tilde{A}^{n-1}$.*

Доказательство. Из определения пути следует, что если в G имеется путь из u в $v \neq u$, то в нем имеется и простой путь из u в v длины $\leq n-1$. А по лемме 6.1 все такие пути представлены в матрице $\tilde{A}^{n-1}$.
□

Таким образом процедура построения матрицы смежности A_{G^*} графа достижимости для G сводится к возведению матрицы $\tilde{A}$ в степень $n-1$. Для этого достаточно выполнить $]\log n[$ возведений в квадрат:

$$\tilde{A} \Rightarrow \tilde{A}^2 \Rightarrow \tilde{A}^{2^2} \Rightarrow \ldots \Rightarrow \tilde{A}^{2^k},$$

где k — это наименьшее число такое, что $2^k \geq n-1$.

Так как стандартный алгоритм умножения $n \times n$ матриц можно выполнить за время $O(n^3)$, то мы получили алгоритм вычисления графа достижимости за время $O(n^3 \log n)$. Отметим, что это оценка числа битовых операций.

Оказывается, что время вычисления A_{G^*} можно улучшить до кубического.

Алгоритм Уоршолла

Вход: Матрица смежности A_G графа G
Выход: $B = A_{G^*}$

1. $B := A_G$;
2. **FOR** $k = 1$ **TO** n **DO**
3. **FOR** $i = 1$ **TO** n **DO**
4. **FOR** $j = 1$ **TO** n **DO**
5. $b_{ij} := b_{ij} \vee (b_{ik} \wedge b_{kj})$;

Теорема 6.1. *Алгоритм Уоршолла вычисляет матрицу $B = A_{G^*}$ за время $O(n^3)$.*

Доказательство. Введем вспомогательное понятие – l - путь.

Определение 21. *l - путем из v_i в v_j называется путь $v_i, v_{i_1}, v_{i_2}, \ldots, v_{i_k}, v_j$ в графе G, в котором все внутренние вершины принадлежат множеству $\{v_1, \ldots, v_l\}$, т.е. $1 \leq i_r \leq l$ при $1 \leq r \leq k$.*

Доказательство корректности алгоритма основывается на следующей лемме.

Лемма 6.2. *Пусть $B^{(l)}$ - матрица, полученная после l выполнений цикла по k.*
Тогда $b_{ij}^{(l)} = 1 \Leftrightarrow$ в G есть l - путь из v_i в v_j.

Доказательство. Используем индукцию по l. При $l = 0$ матрица $B^{(0)} = A_G$ и в ней представлены все 0-пути, т.е. пути состоящие из одного ребра.

Предположим, что лемма верна для некоторого $l \geq 0$ и докажем, что тогда она верна и для $l+1$. Тогда для всех i и j имеем $b_{ij}^{(l+1)} = b_{ij}^{(l)} \vee (b_{il}^{(l)} \wedge b_{lj}^{(l)})$. Если $b_{ij}^{(l+1)} = 1$, то $b_{ij}^{(l)} = 1$ или $b_{il}^{(l)} = 1$ и $b_{lj}^{(l)} = 1$. В первом случае по предположению индукции в G есть l-путь из v_i в v_j, который является и $(l+1)$-путем. Во втором случае имеется l-путь из v_i в v_l и l-путь из v_l в v_j. Объединив их, получим $(l+1)$-путь из v_i в v_j. Обратно, если в G есть $(l+1)$-путь из v_i в v_j, то либо он не проходит через вершину v_l и в этом случае $b_{il}^{(l)} = 1$, либо он включает v_l и тогда $b_{il}^{(l)} = 1$ и $b_{lj}^{(l)} = 1$. В обоих случаях $b_{ij}^{(l+1)} = 1$.
□

6.1.1 Кратчайшие пути между всеми парами вершин

Рассмотрим теперь нагруженные ориентированные графы, ребрам которых приписаны их длины (веса). Пусть $G = (V, E)$ – такой граф, в котором длина ребра $e \in E$ задается функцией $c(e)$. Предположим, что длины всех ребер неотрицательны. Тогда для любой пары вершин u и v однозначно определена длина $\rho(u, v)$ кратчайшего пути из u в v (если из u нельзя достичь v, то будем считать, что $\rho(u, v) = \infty$). Все алгоритмы определения кратчайших путей между вершинами графа используют явно или неявно следующее простое соображение.

Лемма 6.3. Критерий Беллмана *Вершина w графа $G = (V, E)$ лежит на кратчайшем пути из u в v тогда и только тогда, когда $\rho(u, v) = \rho(u, w) + \rho(w, v)$.*

Используя ту же идею, что и в алгоритме Уоршолла, можно построить эффективный алгоритм для нахождения длин кратчайших путей между всеми парами вершин в нагруженном графе. Исходными данными для этого алгоритма являются длины ребер, представленные в матрице $C = (c_{ij})$, где $c_{ij} = c(v_i, v_j)$, $c_{ij} \geq 0$ (если $(v_i, v_j) \notin E$, то $c_{ij} = \infty$).
Результатом алгоритма является матрица $D = (d_{ij})$, в которой $d_{ij} = \rho(v_i, v_j)$ – длина кратчайшего пути из v_i в v_j.

Алгоритм Уоршолла-Флойда

Вход: C
Выход: D

1. $D := C$;
2. **FOR** $k = 1$ **TO** n **DO**
3. **FOR** $i = 1$ **TO** n **DO**
4. **FOR** $j = 1$ **TO** n **DO**
5. $d_{ij} := min\{d_{ij}, d_{ik} + d_{kj}\}$;

Теорема 6.2. *Алгоритм Уоршолла-Флойда строит матрицу кратчайших расстояний за время $O(n^3)$.*

Доказательство теоремы следует из следующей леммы, аналогичной лемме 6.2.

Лемма 6.4. *Пусть $D^{(l)}$ - матрица, полученная после l выполнений цикла по k. Тогда $d_{ij}^{(l)}$ – длина кратчайшего l-пути из v_i в v_j.*

6.2 Задача о кратчайших путях из одного источника

Пусть $G = (V, E)$ — ориентированный граф, для каждого ребера $e \in E$ которого указана его (неотрицательная) длина: $c(e) \geq 0$. Тогда длина пути $p = v_1, v_2, \ldots, v_{k+1}$ определяется как сумма длин ребер, входящих в этот путь: $c(p) = \sum_{i=1}^{k} c(v_i, v_{i+1})$. Если в G имеется путь из вершины a в вершину b, то имеется и такой путь минимальной длины. Он называется *кратчайшим путем из a в b*, а его длина обозначается как $\rho(a, b)$. Конечно, может оказаться несколько различных кратчайших путей. Естественно спросить, как узнать длину кратчайшего пути из a в b и построить его? Лучшие известные на сегодняшний день алгоритмы, отвечающие на этот вопрос, решают, на самом деле, более общую задачу построения всех кратчайших путей из одного источника: *по вершине a найти длины кратчайших путей из a во все достижимые из нее вершины и построить для каждой из таких вершин некоторый кратчайший путь*

из a. Если для каждой вершины $v \in V$, достижимой из a, зафиксировать один кратчайший путь из a в v, то получившийся граф будет представлять ориентированное дерево с корнем a (докажите это!). Это дерево называется *деревом кратчайших путей из* a.

6.2.1 Алгоритм Дейкстры

Мы рассмотрим алгоритм построения дерева кратчайших путей и определения их длин, предложенный в 1959г. Е. Дейкстрой. Его идея следующая: перед каждым этапом известно множество *отмеченных вершин* S, для которых кратчайшие пути найдены ранее; тогда на очередном этапе к нему добавляется вершина w, с самым коротким путем из a, проходящим по множеству S; после этого пересчитываются длины кратчайших путей из a в оставшиеся вершины из $V \setminus S$ с учетом новой вершины w. Длина текущего кратчайшего пути из a в v, проходящего по множеству S, заносится в ячейку $D[v]$ массива D. В конце работы в этом массиве находятся длины соответствующих кратчайших путей. Для определения дерева кратчайших путей служит массив ОТЕЦ, его элемент ОТЕЦ[v] содержит ссылку на вершину, из которой кратчайший путь приходит в v.

Алгоритм Дейкстры
Вход: $G = (V, E)$ — ориентированный граф, $c(u, v) \geq 0$ —длина ребра $(u, v) \in E$ (если $(u, v) \notin E$, то считаем, что $c(u, v) = \infty$), и исходная вершина $a \in V$.

```
1.  S := {a}; % отметить a
2.  D[a] := 0; % расстояние от a до a
3.  FOR EACH v ∈ V, v ≠ a DO
4.      {D[v] := c(a, v); % расстояние от a до v через a
5.      IF  c(a, v) < ∞ THEN ОТЕЦ[v] := a  ELSE ОТЕЦ[v] := −};
%           === ОСНОВНОЙ ЦИКЛ ===
6.  WHILE  V \ S ≠ ∅ DO % есть неотмеченные вершины
7.      { выбрать неотмеченную вершину w с минимальным D[w] ;
8.      S := S ∪ {w};  % отметить w
9.      FOR EACH (неотмеченной) u ∈ V \ S DO
10.         { IF  D[u] > D[w] + c(w, u)
11.         THEN  { D[u] := D[w] + c(w, u);
12.               ОТЕЦ[u] := w}
13.     }
```

Пример 14. *Рассмотрим работу этого алгоритма на нагруженном графе* G *с множеством вершин* $(V = \{a, b, c, d, e, f\}$. *Зададим длины дуг матрицей* $C = (c_{uv})$, *где элемент* $c_{uv} = c(u, v)$, *в частности,* $c_{uv} = \infty$ *означает отсутствие дуги* (u, v).

$$C = \begin{pmatrix} & a & b & c & d & e & f \\ a & 0 & 25 & 5 & 30 & \infty & 75 \\ b & 12 & 0 & \infty & \infty & 120 & 20 \\ c & \infty & 15 & 0 & 20 & 45 & 60 \\ d & \infty & \infty & \infty & 0 & 23 & 20 \\ e & \infty & \infty & 75 & 20 & 0 & 20 \\ f & 40 & 15 & 15 & 26 & \infty & 0 \end{pmatrix}$$

Найдем кратчайшие пути из вершины a *до остальных вершин.*

Поэтапную работу алгоритма Дейкстры удобно представлять в виде таблицы, строки которой соответствуют его этапам. Первый столбец — номер этапа, второй показывает изменение множества отмеченных вершин S, третий — вершину w, добавляемую к S на текущем шаге, четвертый — длину кратчайшего пути из a в w, затем идут столбцы со значениями элементов массивов D и ОТЕЦ.

N	S	w	D[w]	**D**					О	Т	Е	Ц
				b	c	d	e	f	b	c	d	e	f
1.	a	c	5	25	5	30	∞	75	a	a	a	-	a
2.	a, c	b	20	20	-	25	50	65	c	a	c	c	c
3.	a, c, b	d	25	-	-	25	50	40	c	a	c	c	b
4.	a, c, b,d	f	40	-	-	-	48	-	c	a	c	d	b
5.	a, c, b,d, f	e	48	-	-	-	-	-	c	a	c	d	b
6.	a, c, b, d, f, e			20	5	25	48	40	c	a	c	d	b

Таблица 2: Алгоритм Дейкстры на графе G.

Дерево кратчайших путей из вершины a задается массивом ОТЕЦ. Оно представлено на следующем рисунке.

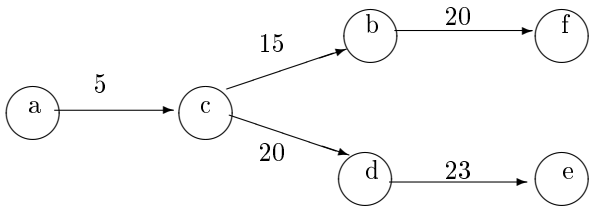


Рис. 24: Дерево кратчайших путей из вершины a в графе G

Теорема 6.3. *Алгоритм Дейкстры строит дерево кратчайших путей из вершины a графа $G = (V, E)$ во все достижимые из нее вершины и для каждой такой вершины v определяет длину $D[v] = \rho(a, v)$ кратчайшего пути в нее из a.*

Время работы алгоритма Дейкстры – $O(|V|^2)$.

Доказательство. . Докажем по индукции, что после каждого этапа алгоритма выполнены следующие условия:
а) для любой вершины $v \in S$ величина $D[v]$ равна длине $\rho(a, v)$ кратчайшего пути из a в v;
б) для любой вершины $v \in V \setminus S$ величина $D[v]$ равна длине кратчайшего пути из a в v, проходящего по множеству S;
в) для каждой вершины v дерева T, задаваемого массивом ОТЕЦ, длина пути из корня a в v равна $D[v]$.

Эти три условия, очевидно выполняются после инициализации в строках 1- 5.

Предположим теперь что они выполнены перед началом k-го этапа. Пусть w — вершина, добавляемая к S на k-ом этапе. По предположению, $D[w]$ — длина кратчайшего пути из a в w, все вершины которого, кроме w, входят в S. Предположим, что есть другой более короткий путь p из a в w. Зафиксируем на этом пути первую вершину u, не входящую в S. По выбору p $u \neq w$. Поэтому путь p разбивается на две непустые части: путь p_1 из a в u и путь p_2 из u в w. Но по выбору w мы имеем, что длина $p_1 \geq D[u] \geq D[w]$. Так как длина p_2 неотрицательна,

то длина $p \geq D[w]$, т.е. этот путь не короче пути, представленного в дереве T. Таким образом, $D[w]$ — это длина кратчайшего пути из a в w. Следовательно, условие (а) выполнено и после k-го этапа.

Рассмотрим теперь произвольную вершину $u \in V \setminus (S \cup \{w\})$. Кратчайший путь p из a в u, проходящий по множеству $S \cup \{w\}$, либо не включает вершину w и в этом случае его длина равна $D[u]$ и он имеется в текущем дереве T, либо он проходит через w и составлен из кратчайшего пути из a в w через S, продолженного ребром (w, u). В последнем случае длина пути равна $D[w]+c(w, u)$. Но в 10-ой строке алгоритма эти величины сравниваются и, если путь через w короче, то его длина становится новым значениием $D[u]$ (строка 11) и он фиксируется в дереве T (строка 12). Следовательно, условия (б) и (в) также выполнены после k-го этапа.

Так как после завершения алгоритма $S = V$, то в завершающем дереве T представлены кратчайшие пути из a во все достижимые из нее вершины, а массив D содержит длины этих путей. Значение $D[u] = \infty$ указывает на то, что вершина u не достижима из вершины a.

Для оценки времени отметим, что число этапов не превосходит $|V|$, так как на каждом этапе в S добавляется новая вершина. На каждом этапе поиск вершины w с минимальным значением $D[w]$ и последующий пересчет значений $D[v]$ требует $O(|V|)$ шагов. Отсюда, общее время ограничено $O(|V|^2)$.
□

6.2.2 О реализации алгоритма Дейкстры

Приведенная выше оценка сложности $O(|V|^2)$ основана на наивной реализации поиска вершины w с минимальным значением $D[w]$. Если реализовать D как очередь с приоритетами, используя фиббоначиевы кучи (см. [4]), то время работы алгоритма Дейкстры будет ограничено величиной $O(|E| + |V| \log |V|)$. При небольшом числе ребер $|E| = o(|V|^2)$ эта оценка по порядку лучше чем $O(|V|^2)$. Разумеется, чтобы добиться ускорения, граф следует представлять не матрицей длин ребер, а списками смежности, включив в них длины ребер. Тогда стр.9 алгоритма можно уточнить:

9. **FOR EACH** $u \in L_w \cap (V \setminus S)$ **DO**

В этом варианте для w будут просматриваться лишь смежные с ней неотмеченные вершины из L_w и общее число выполнений цикла в стр. 9-12 будет не больше числа ребер $|E|$.

Имеются также эффективные реализации алгоритма Дейкстры, использующие верхнюю границу длин ребер $C_m = \max\{c_{ij} \mid 1 \leq i, j \leq |V|\}$. В этом случае длина всякого пути в графе не превосходит $C_m|V|$. Р. Дайал (R. Dial) предложил для случая целочисленных длин ребер размещать вершины в процессе работы алгоритма в "черпаках". В черпаке B_i будут находиться вершины v с текущим значением $D[v] = i$ ($1 \leq i \leq C_m|V|$). Указатель L будет указывать на первый непустой черпак. При изменении $D[v]$ вершина v будет перемещаться в новый черпак (с меньшим номером). При опустошении черпака B_L значение L увеличивается. Для выполнения этого варианта алгоритма Дейкстры достаточно $O(|E| + C_m|V|)$ шагов.

В работе [16] предложена версия алгоритма Дейкстры-Дайала DIKDB, оптимизирующая использование памяти. В процессе работы вершины располагаются в черпаках двух видов: верхних и нижних. Фиксируется некоторое число Δ нижних черпаков $\{B_i \mid i = 1, 2, \ldots, \Delta\}$. Число верхних черпаков U_j не превосходит $C_m|V|/\Delta$. В верхнем черпаке $U_j, (L < j \leq C_m|V|/\Delta)$, находятся вершины v с текущим расстоянием $D[v] \in [j\Delta, (j + 1)\Delta - 1]$. А каждая из вершин v с $D[v] \in [L\Delta, (L + 1)\Delta - 1]$ находится в нижнем черпаке B_i с $i = D[v] - L\Delta$. Когда все нижние черпаки становятся пустыми, L увеличивается и новые нижние черпаки заполняются вершинами из первого непустого верхнего черпака.

Приведем возможную реализацию алгоритма DIKDB. В процессе работы над черпаками выполняются операции вставки и удаления элементов-вершин, последовательный просмотр

элементов и проверка непустоты. Для их представления можно выбрать двусвязные списки с одним указателем на начало, которые позволяют выполнять указанные операции за время $O(1)$.

Алгоритм DIKDB

Вход:
$G = (V, E)$ — ориентированный граф, представленный списками смежности вида
$L_v = (w_1, c(v, w_1)), \ldots, (w_{k(v)}, c(v, w_{k(v)})$, где $w_1, \ldots, w_{k(v)}$ — это все вершины, в которые из v ведут ребра, а $c(v, w_i)$ — целочисленная длина ребра (v, w_i),
исходная вершина $a \in V$, C_m – длина наибольшего ребра из E и параметр Δ – размер верхнего черпака (и число нижних).

1. **FOR EACH** $i = 0, \ldots, \Delta - 1$ **DO** создать пустой черпак B_i;
2. $M := [C_m|V|/\Delta] + 1$; % число верхних черпаков
3. **FOR EACH** $j = 1, \ldots, M$ **DO** создать пустой черпак U_j;
4. $S := \{a\}$; % отметить a
5. $D[a] := 0$; % расстояние от a до a
6. **FOR EACH** $w \in L_a$ **DO**
7. $\{D[w] := c(a, w)$;
8. ОТЕЦ$[v] := a$;
9. **IF** $D[w] < \Delta$ **THEN** ВСТАВИТЬ$(B_{D[w]}, w)$
10. **ELSE** $\{j := [D[w]/\Delta]$; ВСТАВИТЬ(U_j, w) $\}$
11. $\}$;
12. $L := \min\{\{j \mid U_j \neq \emptyset\} \cup \{M + 1\}\}$;
13. $P := \min\{\{i \mid B_i \neq \emptyset\} \cup \{\Delta\}\}$;
14. **IF** $P = \Delta$ **THEN** % все нижние черпаки пусты
15. ПЕРЕНЕСТИ-В-НИЖНИЕ(L); % перенос из первого верхнего черпака в нижние

% === ОСНОВНОЙ ЦИКЛ ===
16. **WHILE** $P < \Delta$ **DO** % есть неотмеченные вершины
17. $\{w :=$НАЧАЛО(B_P); УДАЛИТЬ(B_P, w); % w – вершина с минимальным $D[w]$;
18. $S := S \cup \{w\}$; % отметить w
19. **FOR EACH** (неотмеченной) $u \in L_w$ **DO**
20. **IF** $D[u] > D[w] + c(w, u)$
21. **THEN**
22. $\{ D[u] := D[w] + c(w, u)$;
23. ОТЕЦ$[u] := w$;
24. **IF** $u \in B_i$ **THEN** УДАЛИТЬ(B_i, u);
25. **IF** $u \in U_j$ **THEN** УДАЛИТЬ(U_j, u);
26. **IF** $D[u] < \Delta$ **THEN** ВСТАВИТЬ$(B_{D[u]}, u)$
27. **ELSE** $\{j := [D[u]/\Delta]$; ВСТАВИТЬ(U_j, u) $\}$
28. $\}$;
29. $L := \min\{\{j \mid U_j \neq \emptyset\} \cup \{M + 1\}\}$;
30. $P := \min\{\{i \mid B_i \neq \emptyset\} \cup \{\Delta\}\}$;
31. **IF** $P = \Delta$ AND $L < M$ **THEN** % все нижние черпаки пусты
32. ПЕРЕНЕСТИ-В-НИЖНИЕ(L) % перенос из первого верхнего черпака в нижние
33. $\}$.

Следующая процедура перемещает вершины из первого непустого верхнего черпака U_L в нижние (пустые) черпаки.

ПЕРЕНЕСТИ-В-НИЖНИЕ(L)
1. **FOR EACH** $w \in U_L$ **DO**
2. $\{i := D[w] - L\Delta$; УДАЛИТЬ(U_L, w); ВСТАВИТЬ$(B_i, w)\}$;
3. $L := \min\{\{j \mid U_j \neq \emptyset\} \cup \{M+1\}\}$;
4. $P := \min\{\{i \mid B_i \neq \emptyset\} \cup \{\Delta\}\}$
5. $\}$;

Прокомментируем этот алгоритм. При инициализации в стр.9 вершина-сосед a помещается в нижний черпак, если находится на расстоянии меньше Δ от a, и в стр. 10 — в верхний черпак, если она находится от a на большем расстоянии. Во время работы L указывает номер первого непустого верхнего черпака и может только возрастать. P указывает номер наименьшего непустого нижнего черпака. Если $P = \Delta$, то все нижние черпаки пусты. Таким образом, в стр. 14 и 31 проверяется непустота нижних черпаков и, если все они пусты, то вызывается процедура **ПЕРЕНЕСТИ-В-НИЖНИЕ**(L), которая перемещает вершины из первого непустого верхнего черпака U_L в нижние черпаки.

При инициализации на создание черпаков требуется $O(\Delta + [C_m|V|/\Delta])$ шагов. Каждое перемещение вершины в нижних и верхних черпаках в стр. 24-27 связано с рассмотрением входящего в нее ребра. Причем каждое ребро обрабатывается в стр. 19 только один раз. Поэтому общее число указанных перемещений не превосходит $|E|$. Число перемещений каждой вершины по нижним черпакам не превосходит их числа Δ. Поэтому общее число указанных перемещений не превосходит $|V|\Delta$. Операция переноса вершин из черпака U_L в нижние черпаки в стр. 15 и 32 всегда связана с увеличением L и поэтому может выполняться не более $M = [C_m|V|/\Delta]$ раз. При этом каждая вершина может переместиться из верхнего черпака в нижний не более одного раза. Отсюда для времени выполнения алгоритма получаем оценку $O(|E| + |V|(\Delta + C_m/\Delta))$. Выбрав в качестве Δ величину $\Theta(\sqrt{C_m})$, получим следующее утверждение.

Теорема 6.4. *Алгоритм DIKBD строит дерево кратчайших путей из вершины a графа $G = (V, E)$ во все достижимые из нее вершины и для каждой такой вершины v определяет длину $D[v] = \rho(a, v)$ кратчайшего пути в нее из a.*

Время работы алгоритма DIKBD – $O(|E| + |V|\sqrt{C_m})$.

Пример. Рассмотрим пример применения алгоритма DIKBD к поиску кратчайших путей из вершины a в графе $G = (\{a, b, c, d, e, f\}, E)$, заданном следующими списками смежности:
$L_a = \{b : 5, c : 14, d : 16\}$;
$L_b = \{c : 6, d : 10, e : 6, f : 24\}$;
$L_c = \{b : 2, d : 4, f : 17\}$;
$L_d = \{f : 15\}$;
$L_e = \{c : 3, d : 3, f : 17\}$;
$L_f = \{a : 10\}$.
Здесь в списке L_v пара $u : k$ означает что длина ребра (v, u) равна $c(v, u) = k$.

Максимальная длина C_m ребра в G равна 24. Выберем $\Delta = 4$. Тогда $M = [24 * 6/4] + 1 = 37$. Таким образом, в стр. 1-3 создаются 4 нижних черпака: B_0, B_1, B_2, B_3 и 37 верхних черпаков $U_1, U_2, \ldots, U_{36}$.

При инициализации в стр. 6-13 все нижние черпаки остаются пустыми. Соседи a попадут в три верхних черпака: $U_1 = \{b(5; a)\}, U_3 = \{c(4; a)\}, U_4 = \{d(16; a)\}$ (здесь в скобках после вершины v указаны текущие значения $D[v]$ и ОТЕЦ$[v]$). $L = 1, P = 4$.
Затем процедура **ПЕРЕНЕСТИ-В-НИЖНИЕ**(1) переместит b в U_1. После этого $P = 1$, b будет удалена из U_1 со значениями $D[b] = 5$ и ОТЕЦ$[b] = a$. Затем в цикле в стр. 16-28 будут рассмотрены все соседи b. Они попадут в три вехних черпака: $U_2 = \{c(11; b), e(11; b)\}, U_3 = \{d(15; b)\}, U_7 = \{f(29; b)\}$. После этого $L = 2, P = 4$ и происходит перенос из U_2 в нижние

черпаки, на самом деле, обе вершины попадают в $B_3 = \{c(11; b), e(11; b)\}$ и $P = 3$. После того, как в основном цикле будут удалены обе вершины из B_3 со значениями $D[c] = 11$, ОТЕЦ$[c] = b$ и $D[e] = 11$, ОТЕЦ$[e] = b$ и будут рассмотрены их соседи, получим два непустых верхних черпака: $U_3 = \{d(14; e)\}, U_7 = \{f(28; e)\}$. P снова равно 4, и $L = 3$. Затем d перемещается в B_2, а оттуда удаляется со значениями $D[d] = 11$ и ОТЕЦ$[d] = e$. После этого $L = 7, P = 4$ и f перемещается в B_0, а оттуда удаляется со значениями $D[f] = 28$ и ОТЕЦ$[f] = e$. На этом работа алгоритма завершается.

Построенное алгоритмом DIKBD дерево кратчайших путей из вершины a представлено на рисунке 25.

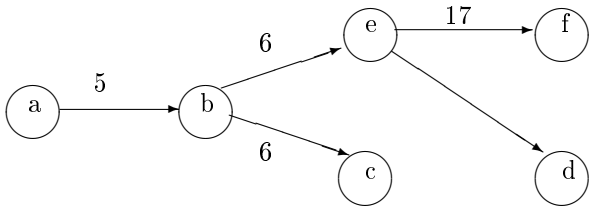


Рис. 25: Дерево кратчайших путей из вершины a, построенное алгоритмом DIKBD.

В упомянутой выше работе [16] проведено сравнение многих алгоритмов поиска кратчайших путей на различных классах графов. Среди 7 рассмотренных там вариантов алгоритма Дейкстры алгоритм DIKBD оказался наиболее эффективным и успешно справлялся с поиском кратчайших путей в графах, насчитывающих до 10^6 вершин.

6.2.3 Алгоритм Беллмана-Форда

Алгоритм Дейкстры требует, чтобы веса всех ребер были неотрицательны (см. задачу 6.6). Но бывают ситуации, когда естественно рассматривать и графы с отрицательными весами ребер. Если в них нет циклов отрицательной длины, то понятие кратчайшего пути из одной вершины в другую определено корректно. Следующий алгоритм позволяет решить задачу о кратчайших путей из одного источника для таких графов. В этом алгоритме, как и в алгоритме Дейкстры, в массиве D для каждой вершины v хранится текущее значение $D[v]$ расстояния от v_0 до v, а в массиве ОТЕЦ – вершина ОТЕЦ$[v]$, предшествующая v в текущем пути из v_0.

Алгоритм основан на следующей простой процедуре ослабления Relax, которая рассматривает ребро (u, v) и пытается уменьшить длину текущего пути из v_0 в v, заменив его на путь из v_0 в u, продолженный ребром (u, v).

procedure Relax(u, v)

1. **IF** $D[v] > D[u] + c(u, v)$ **THEN**
2. $\{D[v] := D[u] + c(u, v)$;
3. ОТЕЦ$[v] := u\}$

Следующий алгоритм основан на двух отдельных алгоритмах, предложенных Р. Беллманом (1958) и Л. Фордом (1962). Он многократно пытается применить процедуру Relax ко всем ребрам графа.

Алгоритм Беллмана-Форда

procedure Bellman_Ford$(G = (V, E), c, v_0)$

1. **FORALL** $v \in V$ **DO**
2. $\{D[v] := \infty$; ОТЕЦ$[v]$:=nil;$\}$
3. $D[v_0] := 0$;
4. **FOR** $i = 1$ **TO** $|V| - 1$ **DO**
5. **FORALL** $(u, v) \in E$ **DO** Relax(u, v);
6. **FORALL** $(u, v) \in E$ **DO**
7. **IF** $D[v] > D[u] + c(u, v)$ **THEN return** false;
8. **return** true;

Лемма 6.5. *Если в графе $G = (V, E)$ нет циклов отрицательной длины, достижимых из v_0, то алгоритм Беллмана-Форда в $D[v]$ возвращает длину кратчайшего пути из v_0 в v.*

Доказательство. Пусть $\rho = v_0, v_1, \ldots, v_k = v$ — кратчайший путь из v_0 в v. Поскольку из v_0 недостижимы циклы отрицательной длины, то можно считать, что в этом пути нет циклов. Тогда $k \leq |V| - 1$. Докажем по индукции, что
для каждого $i \leq k$ после i-й итерации цикла в строках 4-5 $D[v_i]$ равно длине $\rho(v_0, v_i)$ кратчайшего пути из v_0 в v_i.

При $i = 0$ равенство очевидно. Пусть после $(i-1)$-й итерации цикла $D[v_{i-1}] = \rho(v_0, v_{i-1})$. На i-й итерации цикла в строке 5 для ребра (v_{i-1}, v_i) будет вызвана процедура Relax(v_{i-1}, v_i). Так как в этот момент $D[v_{i-1}] = \rho(v_0, v_{i-1})$ и $\rho(v_0, v_i) = \rho(v_0, v_{i-1}) + c(v_{i-1}, v_i) = D[v_{i-1}] + c(v_{i-1}, v_i)$, то в после вызова Relax(v_{i-1}, v_i) значение $D[v_i] = \rho(v_0, v_i)$.

Следствие. *Вершина v достижима из $v_0 \Leftrightarrow$ алгоритм Беллмана-Форда выдает $D[v] < \infty$, а ОТЕЦ$[v]$ — вершина прешествующая v на некотором кратчайшем пути из v_0 в v.*

Действительно, пусть ОТЕЦ$[v] = u$. Это значит, что после некоторого вызова процедуры Relax(u, v) расстояние $D[v]$ становится равным $D[u] + c(u, v)$ и далее не изменяется. По лемме $D[v] = \rho(v_0, v)$ и $D[u] = \rho(v_0, u)$. Тогда имеется кратчайший путь из v_0 в v, включающий последнее ребро (u, v). На нем ОТЕЦ$[v] = u$ предшествует v.

Теорема 6.5.

1. *Если в графе $G = (V, E)$ нет циклов отрицательной длины, достижимых из v_0, то алгоритм Беллмана-Форда возвращает true и для всех $v \in V$ $D[v] = \rho(v_0, v)$ - длина кратчайшего пути из v_0 в v, ОТЕЦ$[v]$ - задает дерево кратчайших путей.*

2. *Если в графе $G = (V, E)$ есть цикл отрицательной длины, достижимый из v_0, то алгоритм Беллмана-Форда возвращает false (некорректная задача)*

Сложность (время работы) алгоритма – $O(|V||E|)$.

Доказательство. Пункт 1 непосредственно следует из леммы 6.5 и следствия из нее. Действительно, если из v_0 не достижимы циклы отрицательной длины, то после завершения цикла в в строках 4-5 ни для какого ребра $(u, v) \in E$ не будет выполнено неравенство $D[v] > D[u] + c(u, v)$ и алгоритм вернет значение true.

Для доказательства пункта 2 предположим, что из v_0 достижим цикл отрицательной длины $v_1, v_2, \ldots, v_k = v_1$. Предположим, что для всех $i = 1, 2, \ldots, k-1$ выполнены неравенства $D[v_{i+1}] \leq D[v_i] + c(v_i, v_{i+1})$. Тогда, сложив все эти неравенства и сократив все $D[v_i]$ слева и справа (заметим, что $D[v_k] = D[v_1]$, получим неравенство $0 \leq \sum_{i=1}^{k-1} c(v_i, v_{i+1})$. Но сумма справа и есть длина цикла, которая по его выбору должна быть отрицательной. Следовательно, хотя бы для одного i неравенство $D[v_{i+1}] \leq D[v_i] + c(v_i, v_{i+1})$ не выполняется. Алгоритм это обнаруживает в строке 7 и возвращает результат false.

Оценка времени очевидна. Отметим, что при большом количестве ребер $|E| = O(|V|^2)$ она является кубической относительно числа вершин.

Пример 15. *Рассмотрим работу этого алгоритма на графах G_1 и G_2, показанных на рис. 26. Оба имеют одинаковые множества вершин $V_1 = V_2 = \{a, b, c, d\}$ и ребер $E_1 = E_2 = \{(a, b), (a, c), (a, d), (b, c), (d, b), (c, d)\}$. Различен лишь вес ребра (c, d).*

Рис. 26: Ориентированные графы G_1 и G_2

Работа алгоритма Беллмана-Форда на графе G_1 показана в следующей таблице. Ищутся

i	(u,v)	**D**			ОТЕЦ		
		b	c	d	b	c	d
1	(a, b)	10	∞	∞	a	-	-
	(a, c)	10	20	∞	a	a	-
	(a, d)	10	20	5	a	a	a
	(b, c)	10	15	5	a	b	a
	(d, b)	7	15	5	d	b	a
	(c, d)	7	15	5	d	b	a
2	(b, c)	7	12	5	d	b	a

Таблица 3: Алгоритм Беллмана-Форда на графе G_1

кратчайшие пути из вершины a. Для $i = 1$ приведены результаты вызовов процедуры Relax(u, v) для всех ребер из E, а для $i = 2$ — только для ребра (b, c), так как вызовы для остальных ребер не меняют массивов D и ОТЕЦ. При $i = 3$ изменений в результатах не происходит. Так как отсутствуют циклы отрицательной длины, то в конце работы массив D задает длины кратчайших путей из a, а массив ОТЕЦ – сами эти пути.

В графе G_2 имеется цикл $bcdb$ отрицательной длины -3, достижимый из a. Работа первой части алгоритма Беллмана-Форда (строки 1-5) на этом графе показана в следующей таблице.

В этой таблице при $i = 2, 3$ показаны только ребра, ослабления по которым изменяют результаты. При провере во второй части алгоритма (строка 7) для ребра (b, c) обнаружится, что $D[c] = 12 > D[b] + c(b, c) = 4 + 5 = 9$. После этого алгоритм завершится выдачей результата false.

6.2.4 Кратчайшие пути в ациклических графах

Отсутствие циклов в графе позволяет за счет выбора топологического порядка перебора ребер строить кратчайшие пути из одной вершины с помощью однократного вызова процедуры Relax для каждого ребра.

Алгоритм Кратчайшие пути в ациклическом графе (КПАГ)

procedure КПАГ$(G = (V, E), c, v_0)$

i	(u,v)	**D**			ОТЕЦ		
		b	c	d	b	c	d
1	(a, b)	10	∞	∞	a	-	-
	(a, c)	10	20	∞	a	a	-
	(a, d)	10	20	5	a	a	a
	(b, c)	10	15	5	a	b	a
	(d, b)	7	15	5	d	b	a
	(c, d)	7	15	5	d	b	a
2	(b, c)	7	12	5	d	b	a
	(c, d)	7	12	2	d	b	a
3	(d, b)	4	12	2	d	b	a

Таблица 4: Алгоритм Беллмана-Форда на графе G_2

1. Отсортировать V в топологическом порядке (см. задачу 5.9)
2. Пусть в этом порядке $V = u_1, u_2, \ldots, u_n$ и $v_0 = u_k$;
3. **FOR** $i = k$ **TO** n **DO**
4. $\{D[u_i] := \infty;$ ОТЕЦ$[u_i]$:=nil$\}$;
5. $D[u_k] := 0$;
6. **FOR** $i = k$ **TO** n **DO**
7. **FORALL** $(u_i, u_j) \in E$ **DO** Relax(u_i, u_j);

Теорема 6.6. *Пусть $G = (V, E)$ – ориентированный ациклический граф с функцией веса ребер c и выделенной вершиной v_0. Тогда алгоритм КПАГ для каждой достижимой из v_0 вершины v возвращает кратчайшее расстояние $D[v]$ от v_0 до нее, а подграф, заданный массивом ОТЕЦ, является деревом кратчайших путей из v_0.*

Время работы алгоритма КПАГ не превосходит $O(|V| + |E|)$, т.е. линейно относительно размера графа.

Доказательство. Пусть $p = (u_k = v_0, u_{k_1}, \ldots, u_{k_r} = v)$ — кратчайший путь из v_0 в v (если таких путей несколько, то пусть p – это кратчайший путь с лексикографически минимальной обратной последовательностью индексов $(k_{r-1}, k_{r-2}, \ldots, k_1)$. Тогда $k < k_1 < \ldots < k_r$. Поэтому из ребер пути p в цикле в строках 6 и 7 первым будет рассмотрено ребро (u_k, u_{k_1}), после него – ребро (u_{k_1}, u_{k_2}) и т.д. Предположим по индукции, что после вызова Relax$(u_{k_{j-1}}, u_{k_j})$ в строке 7 $D[u_{k_j}]$ равно длине $\rho(u_k, u_{k_j})$ кратчайшего пути $p^{(j)} = (u_k = v_0, u_{k_1}, \ldots, u_{k_j})$ из v_0 в u_{k_j} и что массив ОТЕЦ зафиксировал этот путь. Тогда при вызове Relax$(u_{k_j}, u_{k_{j+1}})$ текущее значение $D[u_{k_{j+1}}] > D[u_{k_j}] + c(u_{k_j}, u_{k_{j+1}}) = \rho(u_k, u_{k_{j+1}})$ (равенство означало бы, что имеется кратчайший путь из u_k в $u_{k_{j+1}}$ с лексикографически меньшей чем у $p^{(j)}$ обратной последовательностью индексов). Поэтому после этого вызова $D[u_{k_{j+1}}] = D[u_{k_j}] + c(u_{k_j}, u_{k_{j+1}}) = \rho(u_k, u_{k_{j+1}})$, а ОТЕЦ$[u_{k_{j+1}}] = u_{k_j}$.

Отсюда по индукции заключаем, что после вызова Relax$(u_{k_{r-1}}, u_{k_r})$ $D[v] = D[u_r]$ является кратчайшим расстоянием от v_0 до v, а массив ОТЕЦ задает этот кратчайший путь.

Для оценки сложности заметим, что топологическая сортировка выполняется с помощью алгоритма *ПОГ+ пост* за линейное время $O(|V| + |E|)$, цикл в строках 4 и 5 выполняется за $O(|V|)$ шагов, а цикл в строках 6 и 7 требует не более $O(|V| + |E|)$ шагов.

У алгоритма КПАГ имеется интересное приложение к задаче управления проектами. Предположим, что имеется большой проект, разбитый на некоторое множество *работ*. Всему проекту соответствует граф работ, в котором каждая работа представлена ориентированным ребром.

Все работы, входящие в некоторую вершину должны быть закончены прежде, чем начнется выполнение любой выходящей из нее работы. Из этого следует, что граф работ является ациклическим. Каждой работе сопоставлен вес – плановое время ее выполнения. (Вообще говоря, не всякое отношение предшествования работ описывается графом указанного вида. Иногда для его точного задания требуется добавлять фиктивные работы нулевого веса.) Одна из основных технологий, применяемых для управления проектами – PERT (Project Evaluation and Review Technique) включает нахождение так называемого *критического пути* – самого длинного пути в графе. Содержательно, его длина равна общему времени выполнения проекта при условии максимально возможного распараллеливания работ. Поэтому основное внимание при управлении проектом следует уделять работам, находящимся на критическом пути, а работы не на нем можно и задержать. Поскольку в процессе реализации проекта реальные времена выполнения работ отличаются от плановых, граф работ часто меняется и всякий раз задачу поиска критического пути требуется решать заново. Как это делать? Достаточно просто поменять у каждого ребра знак веса на противоположный (отрицательный) и применить алгоритм КПАГ к получившемуся графу.

6.3 Задачи

Задача 6.1. *Как изменить алгоритм Уоршолла-Флойда, чтобы находить не только длины кратчайших путей, но и сами кратчайшие пути?*

Задача 6.2. *Пусть A_G – матрица смежности графа $G = (V, E)$, рассматриваемая над кольцом целых чисел. Пусть $A_G^k = (a_{ij}^{(k)})$ – ее (целочисленная) k-ая степень.*

Докажите, что $a_{ij}^{(k)}$ – это число различных путей длины k из v_i в v_j.

Задача 6.3. *Пусть p — это кратчайший путь между вершинами u и v в нагруженном ориентированном графе $G = (V, E)$. Останется ли этот путь кратчайшим, если увеличить длину каждого ребра на константу c? Докажите это утверждение или опровергните с помощью контрпримера.*

Задача 6.4. *Предложите алгоритм, который в ориентированном нагруженном графе $G = (V, E)$ с неотрицательными весами ребер находит цикл минимальной длины.*

Задача 6.5. Диаметр, радиусы и центры графа Диаметром *(не)ориентированного графа $G = (V, E)$ с весами ребер $c : E \to R^+$ называется максимальное расстояние между вершинами графа $d_G = \max\{\rho(u, v)|u, v \in V\}$.* Внутренний центр *ориентированного графа G – это такая вершина v_0, для которой величина $R(v_0) = \max\{\rho(v_0, v) \mid v \in V\}$ минимальна.* Внешний центр *ориентированного графа G – это такая вершина u_0, для которой величина $r(u_0) = \max\{\rho(v, u_0) \mid v \in V\}$ минимальна. $R_G = R(v_0)$ и $r_G = r(u_0)$ называются* внутренним *и* внешним радиусами *графа G, соответственно. Заметим, что внутренний и внешний центры могут быть не единственными. а) Докажите, что в графе G существуют внешний и внутренний центры тогда и только тогда, когда G является сильно связным.*

б) Приведите пример графа G, у которого $R_G \neq r_G$.

*в) Предложите эффективный алгоритм опеределения диаметра d_G, радиусов R_G, r_G и центров графа G. (*Указание: используйте алгоритм Уоршолла-Флойда.

Задача 6.6. *Где в доказательстве правильности алгоритма Дейкстры используется неотрицательность весов ребер? Приведите пример графа (с отрицательными весами), для которого алгоритм Дейкстры дает неверный ответ.*

Задача 6.7. *Докажите, что на каждом шаге алгоритма Дейкстры кратчайший путь из исходной вершины в любую вершину множества S проходит только через вершины множества S.*

Задача 6.8. *Сколько раз может меняться для одной вершины v значение $D[v]$ в ходе работы алгоритма Дейкстры для графа с n вершинами. Привести пример на каждый возможный случай.*

Задача 6.9. *Заменим в строке 10 алгоритма Дейкстры знак неравества на противоположный:*
10'. { **IF** $D[u] < D[w] + c(w, u)$
Будет ли такой модифицированный алгоритм находить самые длинные простые пути из заданной вершины $a \in V$? Докажите это утверждение или приведите контрпример.

Задача 6.10. Быстрейшая поездка. *Содержательно, задача состоит в том, чтобы по железнодорожному расписанию определить самый быстрый способ добраться из одного пункта в другой. Более формально, задан ориентированный граф $G = (V, E)$, вершины которого представляют станции, а ребра соединяют станции, между которыми ходят поезда. Для каждого ребра (u, v) задано расписание поездов $R(u, v)$ на ветке (u, v), представляющее последовательность троек $(n_1, t_1^u, t_1^v), \ldots, (n_r, t_r^u, t_r^v)$, в которых n_i – номер поезда, t_i^u – время его отправления со станции u, t_i^v – время его прибытия на станцию v.*
а) Предложите алгоритм, который по станции отправления $a \in V$, конечной станции $b \in V$ и времени t_0 появления пассажира на станции a находит расписание самой быстрой поездки этого пассажира из a в b. (Будем считать, что пересадка времени не занимает).
б) Предположим, что для каждой станции $u \in V$ задано время T_u, требующееся для пересадки на этой станции. Как найти самый быстрый путь в этом случае?
Указание: *модифицируйте алгоритм Дейкстры.*

Задача 6.11. Обобщенное неравенство треугольника. *Нагруженный ориентированный граф $G = (V, E), c : E \to R$ удовлетворяет* обобщенному неравенству треугольника, *если для любой пары вершин u и v и любых двух путей p и q между ними из того, что число ребер в p не больше числа ребер в q следует, что $c(p) \leq c(q)$ (т.е. самый короткий путь имеет минимальное число ребер).*
(а) Предложите алгоритм полиномиальной сложности, проверяющий удовлетворяет ли данный граф обобщенному неравенству треугольника.
(б) Покажите, что для нагруженных графов $G = (V, E), c : E \to R$, удовлетворяющих обобщенному неравенству треугольника, для задачи поиска кратчайших путей из одного источника имеется алгоритм сложности $O(|V| + |E|)$.

Задача 6.12. *Алгоритм Беллмана-Форда позволяет обнаруживать циклы отрицательной длины. Предложите алгоритм, печатающий вершины какого-нибудь такого цикла.*

Задача 6.13. *$G = (V, E)$ - ориентированный граф без циклов отрицательной длины. c - весовая функция на ребрах (ребра могут быть отрицательной длины). Построить алгоритм, который для каждой вершины $v \in V$ находит расстояние до ближайшего соседа:*

$$\rho^*(v) = \min_{u \neq v} \rho(u, v)$$

Задача 6.14. Валютные операции
Пусть на рынке имеется n валют, текущие курсы которых заданы $n \times n$ матрицей R: 1единица i-ой валюты обменивается на $R[i, j]$ единиц j-ой валюты. Разработайте алгоритм, который находит такую последовательность валют (цикл) $i_1, i_2, \ldots, i_k, i_1$, для которой $R[i_1, i_2] \cdot R[i_2, i_3] \cdot \ldots \cdot R[i_k, i_1] > 1$. Оцените время работы этого алгоритма.

Задача 6.15. Вложенные ящики.
Скажем, что d-мерный ящик размера $(x_1, x_2, \ldots, x_d)$ вкладывается в *ящик размера*

$(y_1, y_2, \ldots, y_d)$, *если существует такая перестановка* $i_1, i_2, \ldots, i_d$ *измерений* $1, 2, \ldots, d$, *для которой* $x_{i_1} \le y_1, x_{i_2} \le y_2, \ldots, x_{i_d} \le y_d$.

а) Докажите, что отношение "вкладывается в" *транзитивно.*

б) Предложите алгоритм проверки этого отношения.

в) Пусть дано n d*-мерных ящиков. Предложите алгоритм для определения самой большой подпоследовательности вложенных ящиков (матрешки из максимально возможного числа ящиков). Оцените его сложность.*

Системы ограничений на разности.
Система ограничений на разности – это система неравенств вида:

$$(*) \quad x_i - x_j \le b_k \quad 1 \le i, j \le n,\ 1 \le k \le m$$

(здесь x_i, x_j – переменные, b_k – константы). Свяжем с этой системой граф $G_* = (V_*, E_*)$, у которого $V_* = \{v_0, v_1, \ldots, v_n\}$,
$E_* = \{(v_0, v_i) \mid 1 \le i \le n\} \cup \{(v, v_j) \mid$ неравенство $x_i - x_j \le b_k$ входит в систему $(*)\}$.
Зададим веса ребер следующим образом:
$c(v_0, v_i) = 0$ $(1 \le i \le n)$, $c(v_i, v_j) = b_k$, если $x_i - x_j \le b_k$ входит в систему $(*)$.

Задача 6.16. *Докажите следующее утверждение.*

Теорема 6.7.
а) Если в графе G_* *нет циклов отрицательной длины, то* $x_1 = \rho(v_0, v_1), x_2 = \rho(v_0, v_2), \ldots, x_n = \rho(v_0, v_n)$ *является решением системы (*).*
б) Если в графе G_* *есть циклы отрицательной длины, то система (*) не имеет решений.*

Задача 6.17. *Используя предыдущую задачу, предложите алгоритм, для проверки разрешимости систем вида (*) и нахождения их решений.*

7 Потоки в сетях

7.1 Потоки и разрезы. Алгоритм Форда-Фалкерсона

Понятие (транспортной) сети служит для моделирования задач, связанных с поиском оптимальных планов перемещения или перевозки продуктов от источников (производителей) к их потребителям. Транспортные сети представляются ориентированными графами с нагруженными дугами[2]

Определение 22. Сеть $N = (V, E, s, t, c)$ - *это связный ориентированный граф без петель* $G = (V, E)$, *у которого*

1) имеется одна вершина s, *в которую не входят дуги (источник);*
2) имеется одна вершина t, *из которой не исходят дуги (сток);*
3) каждой дуге $e = (u, v) \in E$ *сопоставлена ее пропускная способность* $c(e) \ge 0$.

Содержательно, пропускная способность дуги — это максимальный объем продукта, который можно переместить по этой дуге, или максимальная скорость его перевозки.

План перевозки задается с помощью *потока* – функции, которая определяет объем (скорость) перемещения продукта по каждой дуге. Значение потока на дуге не должно превышать ее пропускной способности и для каждой вершины поток, который в нее втекает, должен совпадать с потоком, который из нее вытекает (закон Киркгофа). Такой поток может описывать поведение газа или нефти в трубопроводе, потоки автомобильных грузов в сети автострад, пересылку товаров по железной дороге, передачу информации в информационной сети и т.п.

[2] В транспортных сетях, как правило, вместо термина *ребро* используется его синоним – *дуга*.

Определение 23. Поток f в сети N - *это функция* $f : E \to R$ *такая, что*
1) $0 \le f(e) \le c(e)$ *для каждой дуги* $e \in E$;
2) для каждой вершины $v \in V \setminus \{s,t\}$
$\sum\{f(e)|$ *e входит в* $v\} = \sum\{f(e)|$ *e выходит из* $v\}$.
Величина потока f *в сети* N $val(f) = \sum_{u \in V} f(s,u)$.
Поток с наибольшим значением величины называется максимальным.

Следствие. Величина потока $val(f) = \sum_{u \in V} f(u,t)$. (Показать!)

Определение 24. Разрез A в сети N - *подмножество вершин* A *такое, что источник* $s \in A$, *а сток* $t \in V \setminus A$. *Разрез (и вообще любое подмножество вершин сети)* A *однозначно задает множество дуг* $P(A) = \{(u,v) \mid u \in A, v \in V \setminus A\}$, *ведущих из* A *"наружу".*
Пропускная способность разреза A *:*
$$c(A, V \setminus A) = \sum_{(u,v) \in P(A)} c(u,v)$$
Разрез с минимальной пропускной способностью называется минимальным.
Поток f через разрез (множество) A:
$$f(A, V \setminus A) = \sum_{(u,v) \in P(A)} f(u,v) \ .$$

Величина потока связана с его прохождением через разрез.

Лемма 7.1. *Для произвольного разреза* A *и любого потока* f
$$val(f) = f(A, V \setminus A) - f(V \setminus A, A).$$

Доказательство. Из определения потока следует, что

$$\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) = \begin{cases} val(f) & \text{если u = s} \\ 0 & \text{если } u \in A \setminus \{s\} \end{cases}$$

Просуммировав эти равенства по всем $u \in A$, получим

$$\sum_{u \in A} \sum_{v \in V} f(u,v) - \sum_{u \in A} \sum_{v \in V} f(v,u) = val(f).$$

Для любой пары вершин $u \in A, v \in A$ поток $f(u,v)$ учитывается в первой сумме в левой части с плюсом, а во второй сумме с минусом. Сократив все такие слагаемые, получим

$$\sum_{u \in A} \sum_{v \in V \setminus A} f(u,v) - \sum_{u \in A} \sum_{v \in V \setminus A} f(v,u) = val(f)$$

или $f(A, V \setminus A) - f(V \setminus A, A) = val(f)$.

Теорема 7.1 ((Форд, Фалкерсон))**.**
(1) Величина любого потока из s *в* t *не превосходит пропускной способности минимального разреза.*
(2) Существует поток, достигающий этого значения (максимальный поток).

Доказательство. (1) Пусть A — минимальный разрез. По лемме 7.1 для любого потока f имеем $val(f) = f(A, V \setminus A) - f(V \setminus A, A) \le f(A, V \setminus A) = \Sigma_{e \in P(A)} f(e) \le \Sigma_{e \in P(A)} c(e) = c(A, V \setminus A)$.

Пример 16. *Рассмотрим сеть* N *и поток* f *в ней, показанные на рис. 27. Пропускные способности дуг указаны в скобках, а значения потока на этих дугах – без скобок. Величина этого потока* $val(f) = f(s,a) + f(s,b) + f(s,c) = 3 + 6 + 4 = 13$. *Один из разрезов в сети* N *образует множество вершин* $A = \{s,a,b,c\}$. *Для него* $P(A) = \{(a,f),(b,f),(b,g),(b,e),(c,e)\}$, $P(V \setminus A) = \{(e,a)\}$, *пропускная способность этого разреза* $c(A, V \setminus A) = 6 + 3 + 2 + 2 + 5 = 18$, *поток через* A *равен* $f(A, V \setminus A) = 5 + 2 + 2 + 2 + 4 = 15$, *а поток в обратном направлении –* $f(V \setminus A, A) = 2$.

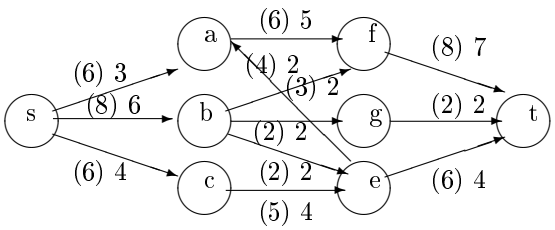


Рис. 27: Сеть N (пропускные способности в скобках) и поток f

Основные алгоритмы построения максимального потока основываются на последовательном увеличении потока, причем модификация потока, производится вдоль *увеличивающих путей (цепей)*, определенных Фордом и Фалкерсоном.

Определение 25. *Пусть N - сеть, f - поток в ней.* Увеличивающий путь (увеличивающая цепь) из s в $v \in V$ *относительно f – это любой путь P из s в v в неориентированном графе, полученном из G игнорированием направлений дуг, в котором:*

1) для любой дуги (u, v), проходимой в P в прямом направлени (прямой дуги), $f(u, v) < c(u, v)$;

2) для любой дуги (v, u), проходимой в P в обратном направлении (обратной дуги), $f(v, u) > 0$.

Увеличивающий путь из s в t будем просто называть увеличивающим путем.

Следующая теорема устанавливает связь между отсутствием увеличивающих путей, максимальностью потока и и совпадением потока через разрез с его пропускной способностью.

Теорема 7.2. *Следующие условия эквивалентны:*

1) поток f из s в t максимальный;

2) не существует увеличивающего пути для f;

3) для некоторого разреза A $val(f) = c(A, V \setminus A)$.

Доказательство. (1) $\Longrightarrow$ (2) следует из определения увеличивающего пути. Действительно, предположим, что в сети есть увеличивающий путь $p = (s = v_0 \stackrel{e_1}{\rightarrow} v_1 \stackrel{e_2}{\rightarrow} \ldots, v_{l-1} \stackrel{e_l}{\rightarrow} v_l = t)$. Положим $\Delta(e_i) = c(e_i) - f(e_i)$, если дуга e_i прямая, и $\Delta(e_i) = f(e_i)$, если e_i — обратная дуга. Пусть $\delta = \min\{\Delta(e_i) | 1 \leq i \leq l\}$. Тогда $\delta > 0$. Определим новый поток f', который совпадает с f всюду кроме p, а на дугах из p положим $f'(e_i) = f(e_i) + \delta$, если дуга e_i прямая, и $f'(e_i) = f(e_i) - \delta$, если дуга e_i обратная. Нетрудно проверить, что f' является потоком и что $val(f') = val(f) + \delta$. (2) $\Longrightarrow$ (3). Положим $A = \{v \in V |$существует увеличивающий путь из s в $v\}$. Тогда A — разрез (почему?). Для каждой дуги $e = (v, u) \in P(A)$ имеет место $f(e) = c(e)$ (иначе можно было бы продолжить увеличивающий путь до u). По той же причине для каждой дуги $e \in P(V \setminus A)$ $f(e) = 0$. Тогда по лемме 7.1 получаем, что $val(f) = f(A, V \setminus A) - f(V \setminus A, A) = c(A, V \setminus A)$. (3) $\Longrightarrow$ (1) следует по теореме 1.

Определенная в доказательстве этой теоремы для увеличивающего пути p величина $\delta = \delta(p)$ определяет максимальное возможное увеличение потока на этом пути.

Пример 17. *В сети N рис. 27 увеличивающими относительно указанного потока f будут следующие пути с соответствующими значениями δ:*

$p_1 = (s \stackrel{(6)3}{\longrightarrow} a \stackrel{(6)5}{\longrightarrow} b \stackrel{(8)7}{\longrightarrow} = t)$, $\delta(p_1) = 1$;

$p_2 = (s \stackrel{(6)3}{\longrightarrow} a \stackrel{(4)2}{\longleftarrow} e \stackrel{(6)4}{\longrightarrow} = t)$, $\delta(p_2) = 2$;

$p_3 = (s \xrightarrow{(8)6} b \xrightarrow{(3)2} f \xrightarrow{(8)7} = t)$, $\delta(p_3) = 1$;
$p_4 = (s \xrightarrow{(6)4} c \xrightarrow{(5)4} e \xrightarrow{(6)4} = t)$, $\delta(p_4) = 1$;
$p_5 = (s \xrightarrow{(6)4} c \xrightarrow{(5,)4} e \xrightarrow{(4)2} a \xrightarrow{(6)5} b \xrightarrow{(8)7} = t)$, $\delta(p_5) = 1$.

Из теоремы 7.2 можно извлечь идею следующего простого алгоритма построения максимального потока. Начиная с нулевого потока ($f(e) = 0$ для каждой дуги $e \in E$), ищем увеличивающие пути и увеличиваем вдоль них текущий поток. Завершаем вычисление, когда обнаруживаем отсутствие увеличивающих путей. Этот алгоритм был предложен в работе Форда и Фалкерсона в 1962г.

АЛГОРИТМ ФОРДА И ФАЛКЕРСОНА (Ф-Ф)

Вход: сеть $N = (V, E, s, t, c)$.
Выход: максимальный поток f в N.
Структуры данных:
$L(v)$ - предшественник v в увеличивающем пути,
$\delta(v)$ - величина дополнительного потока из s в v,
Q – очередь вершин. В качестве имен вершин используются их номера (положительные целые числа).

```
        % Инициализация:
1       готово = "нет";
2       FOR ALL  e ∈ E  DO f(e) := 0 ;
        % Основной цикл:
3       WHILE   готово="нет" DO
4           { FOR ALL v ∈ V \ {s}  DO  { L(v) := 0; δ(v) := 0 };
5           L(s) := s; δ(s) := ∞;
6           ДОБАВИТЬ(Q, s);
7           WHILE  Q ≠ ∅  DO
8           {v := НАЧАЛО(Q);
9            УДАЛИТЬ(Q);
10           ПРОСМОТРЕТЬ(v)
11           };
12           IF  L(t) ≠ 0  %  t помечена, есть увел. путь
                 % увеличить поток f вдоль увеличивающего пути;
13            THEN {v := t;
14                  WHILE  v ≠ s DO
15                    {u := L(v);
16                    IF u > 0
17                    THEN f(u, v) := f(u, v) + δ(t)    % прямая дуга
18                    ELSE f(v, u) := f(v, u) − δ(t);    % обратная дуга
19                    v := |u| }
20                    }
21           ELSE готово="да"
22            }
```

Процедура ПРОСМОТРЕТЬ(v) пытается продлить увеличивающий путь из s в v, проверяя

сначала все дуги, выходящие из v, а затем – дуги, входящие в v.

PROCEDURE ПРОСМОТРЕТЬ(v)

Поиск прямых увеличивающих дуг:
1 **FOR ALL** $(v,u) \in E$ **DO**
2 **IF** $L(u) = 0$ AND $f(v.u) < c(v,u)$,
3 **THEN** % продлить увел. путь до u:
4 { $L(u) := v$; $\delta(u) := min\{\delta(v), c(v,u) - f(v,u)\}$;
5 ДОБАВИТЬ(Q,u);
6 };
Поиск обратных увеличивающих дуг:
7 **FOR ALL** $(u,v) \in E$ **DO**
8 **IF** $L(u) = 0$ AND $f(u,v) > 0$,
9 **THEN** % продлить увел. путь до u:
10 { $L(u) := -v$; $\delta(u) := min\{\delta(v), f(u,v)\}$;
11 ДОБАВИТЬ(Q,u);
12 };

Теорема 7.3. *Если алгоритм Ф-Ф останавливается, то полученный поток максимален.*

Доказательство. Действительно, в момент остановки алгоритма Ф-Ф очередь Q пуста и $L(t) = 0$. Положим $W = \{v \mid L(v) \neq 0\}$. Тогда W – разрез. Для любой дуги $(v,u) \in P(W)$ $f((u,v) = c((u,v))$ (иначе при вызове ПРОСМОТРЕТЬ(v) в цикле 1-6) для u были бы выполнены все условия и эта вершина была бы "помечена": $L(u) := v$. Для любой же обратной дуги $(u,v) \in P(V \setminus W)$ $f((u,v) = 0$ (иначе при вызове ПРОСМОТРЕТЬ(v) в цикле 7-12) для u были бы выполнены все условия и эта вершина была бы "помечена": $L(u) := -v$). Но тогда по лемме 1 имеем $val(f) = f(W, V \setminus W) - f(V \setminus W, W) = c(W, V \setminus W)$
и по теореме 2 f – максимальный поток.

Следующий пример показывает, что сложность алгоритма Ф-Ф может зависеть от значений пропускных способностей дуг.

Пример 18. *Рассмотрим следующую сеть:*

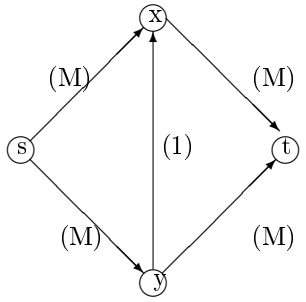


Рис. 28: Сеть с большим временем работы алгоритма Ф-Ф

Предположим, что алгоритм Ф-Ф, начав с нулевого потока, поочередно находит и использует для увеличения потока увеличивающие пути $p_1 = s \to y \to x \to t$ *и* $p_2 = s \to x \leftarrow y \to t$. *Тогда на каждом этапе алгоритма поток увеличивается на 1 и максимальный поток* $2M$ *достигается за* $2M$ *этапов.*

ЗАМЕЧАНИЕ. Форд и Фалкерсон показали, что их алгоритм может не найти максимального значения потока. Они построили примеры сетей с иррациональными значениями $c(v,u)$, на которых алгоритм не останавливается и сходится к $1/4$ от величины максимального потока. Причина этого в том, что в алгоритме Ф-Ф не учитывается порядок перебора увеличивающих путей. Эдмондс и Карп в 1972 г. предложили модификацию алгоритма Ф-Ф, в которой на каждом этапе поток увеличивается вдоль кратчайшего (по числу дуг) увеличивающего пути. Их алгоритм в худшем случае имеет сложность $O(|V||E|^2)$, что при большом количестве дуг $|E| = O(|V|^2)$ дает оценку $O(|V|^5)$.

7.2 Алгоритм построения максимального потока за кубическое время

В этом разделе мы приведем более эффективный алгоритм сложности $O(|V|^3)$, первоначальная идея которого принадлежала Е.А. Диницу (1970), а затем была уточнена А.В. Карзановым (1974). Вариант этого алгоритма, представленный здесь, является упрощенной формой этого алгоритма, описанной в работе трех авторов Малхортры (Malhotra V. M.), Кумара (Kumar M.P.) и Махешвари (Maheshwari S.N.) в 1978г. Его идея состоит в том, чтобы на каждом этапе производить увеличение потока по всем кратчайшим увеличивающим путям.

Определение 26. *Пусть* $N = (V, E, s, t, c)$ *– сеть,* f *– поток в* N. Вспомогательная сеть $AN(f) = (V(f), E(f), s, t, ac)$ *– это такая сеть, вершины которой* $V(f)$ *разбиты на* l *слоев* $V_0 = \{s\}, V_1, ...,\ t \in V_l$, *так что дуги соединяют вершины соседних слоев* V_i *и* V_{i+1} *и при этом, если* $t \notin V_i$, *то слой* V_{i+1} *и входящие в него дуги определяются следующим образом:*

1) если $(u,v) \in E,\ u \in V_i,\ v \notin \bigcup_{j=0}^{i} V_j$ *и* $f(u,v) < c(u,v)$, *то* $v \in V_{i+1},\ (u,v) \in E(f)$ *и* $ac(u,v) = c(u,v) - f(u,v)$;

2) если $(v,u) \in E,\ u \in V_i,\ v \notin \bigcup_{j=0}^{i} V_j$ *и* $f(v,u) > 0$, *то* $v \in V_{i+1},\ (u,v) \in E(f)$ *и* $ac(u,v) = f(v,u)$.

Вообще говоря, возможен и третий случай, когда поток f больше нуля на двух встречных дугах: $f(u,v) > 0$ и $f(v,u) > 0$. В этом случае можно провести эквивалентное преобразование: минимальный из этих потоков заменить на 0, а максимальный из них на значение потока $|f(u,v) - f(v,u)|$.

Алгоритм ПОВС построения вспомогательной сети основан на поиске в ширину. Пусть $L(v) = \{u \mid (v,u) \in E\}$ – список "сыновей"v, $M(v) = \{u \mid (u,v) \in E\}$ – список "отцов"v в N, а $AL(v)$ и $AM(v)$ – аналогичные списки для $AN(f)$, AV – вершины $AN(f)$, ДЛИНА(v) - расстояние от s до v, Q – очередь вершин (вначале пустая).

```
    PROCEDURE ПОВС
% Инициализация:
1       FOR ALL u ∈ V DO
2           { ДЛИНА(u) := ∞;
3             AM(u) := ∅; AL(u) := ∅;
4             FOR ALL v ∈ V DO ac(u,v) := 0;
5             FOR ALL v ∈ V DO af(u,v) := 0
6           };
```

```
7       ДОБАВИТЬ(Q, s); AV := ∅; ДЛИНА(s) := 0;
                % поиск в ширину от s:
8       WHILE  Q ≠ ∅  DO
9           {  u := НАЧАЛО(Q); УДАЛИТЬ(Q);  AV := AV ∪ {u};
10           FOR ALL  v ∈ L(u)  DO                  % поиск прямых дуг
11             IF (ДЛИНА(u) <  ДЛИНА(v) ≤ ДЛИНА(t) AND  f(u,v) < c(u,v))
12             THEN IF  ДЛИНА(v) = ∞                  % v - новая
13                    THEN {   ДОБАВИТЬ(Q, v);
14                    ДЛИНА(v) := ДЛИНА(u) + 1};
                % добавить (u,v) к сети AN:
15                    AL(u) := AL(u) ∪ {v};  AM(v) := AM(v) ∪ {u};
16                    ac(u,v) := c(u,v) − f(u,v)
17                          };
18           FOR ALL  v ∈ M(u)  DO                  % поиск обратных дуг
19             IF  (ДЛИНА(u) < ДЛИНА(v) ≤ ДЛИНА(t))  AND f(v,u) > 0)
20             THEN  IF  ДЛИНА(v) = ∞                  % v - новая
21                    THEN {   ДОБАВИТЬ(Q, v);
22                    ДЛИНА(v) := ДЛИНА(u) + 1};
23                    IF  ac(u,v) = 0   % добавить (u,v) к сети AN
24                    THEN {  AL(u) := AL(u) ∪ {v};  AM(v) := AM(v) ∪ {u} };
25                    ac(u,v) := ac(u,v) + f(v,u)
26             }                 };
27           используя "обратный"поиск в ширину от  t, удалить из  AN(f)  все вершины,
             из которых нет пути в  t  и инцидентные им дуги.
```

Следующая теорема устанавливает основные свойства алгоритма ПОВС и получаемой в его результате вспомогательной сети.

Теорема 7.4. *а) Если в результате работы алгоритма ПОВС ДЛИНА$(t) = \infty$, то в сети N нет увеличивающего пути от s к t и, следовательно, поток f максимальный.*

б) Если ДЛИНА$(t) = l \neq \infty$, то кратчайший увеличивающий путь из s в t в сети N имеет длину l. Более того, пути из s в t в сети AN взаимно однозначно соответствуют увеличивающим путям длины l в сети N.

в) Если $maxf$ – величина максимального потока в N, то величина максимального потока в $AN(f)$ равна $maxf - val(f)$.

г) Время работы процедуры ПОВС – $O(|E|) \leq O(n^2)$, где $n = |V|$.

Пример 19. *Применим алгоритм ПОВС к сети N с потоком f, приведенной на рис. 27. В результате получится сеть $AN(f)$, показанная на следующем рисунке. Отметим, что*

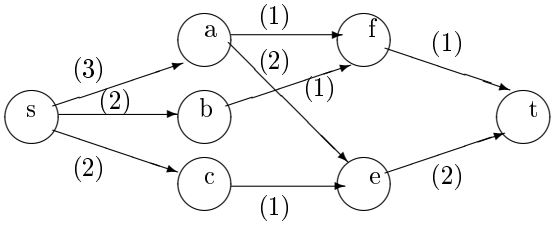


Рис. 29: Вспомогательная сеть $AN(f)$ для сети N с потоком f с рис. 27

вершина g не попала в эту сеть, так как единственное, ведущее в нее ребро (b, g) насыщено, т.е. $f(b, g) = c(b, g)$, и ДЛИНА$(g) = \infty$ до конца алгоритма.

Определение 27. *Поток g в сети AN называется* тупиковым, *если для него не существует увеличивающего пути без обратных дуг (*прямого *увеличивающего пути).*

Определение 28. Потенциал $p(v)$ *вершины v в сети N - это максимальное количество потока, которое можно протолкнуть через v, т.е. $p(v) = min(\sum_{u \in V} c(u, v),\ \sum_{u \in V} c(v, u))$ для $v \in V \setminus \{s, t\}$, $p(s) = \sum_{u \in V} c(s, u)$, $p(t) = \sum_{u \in V} c(u, t)$.*

АЛГОРИТМ МАХП ПОСТРОЕНИЯ МАКСИМАЛЬНОГО ПОТОКА
Вход: сеть $N = (V, E, s, t, c)$.
Выход: максимальный поток f в N.

```
1       FOR ALL v ∈ V DO f(u,v) := 0; готово := "нет";
2       WHILE готово="нет" DO
                % Новый этап:
3           { ПОВС;                 % построить вспомогательную сеть AN(f)
4           IF t недостижима из s в AN(f)
5           THEN готово := "да"                 % максимальный поток построен
6           ELSE
                % Построение тупикового потока:
7           { FOR ALL (u,v) ∈ E(f) DO g(u,v) := 0;          %инициал. тупикового
потока
8             WHILE p(s) > 0 AND p(t) > 0 DO
9                     { найти v с минимальным потенциалом p(v) > 0;
10                     ПРОТОЛКНУТЬ(v, p(v));
11                     ПРОТЯНУТЬ(v, p(v))
12                     WHILE есть v ∈ V(f) \ {s,t}, для которой p(v) = 0
13                           DO удалить v и все инцидентные ей дуги из AN(f);
14                     }
                % Добавление тупикового потока:
15            FOR ALL (u,v) ∈ E DO
16                     IF (u,v) ∈ E(f)    % прямая дуга
17                     THEN f(u,v) := f(u,v) + g(u,v)
18                     ELSE IF (v,u) ∈ E(f)   % обратная дуга
19                           THEN f(u,v) := f(u,v) − g(u,v)
20          }     }
```

PROCEDURE ПРОТОЛКНУТЬ(y,h)
увеличивает поток g на h единиц, проталкиваемых от y к t

```
1       создать очередь Q; ДОБАВИТЬ(Q, y);
% треб[y] и треб[u] – размер выталкиваемого из y и u потока:
2       треб[y] := h;
3       FOR ALL u ∈ V(f) \ {y} DO треб[u] := 0;
4       WHILE Q ≠ ∅ DO
5           { v := НАЧАЛО(Q); УДАЛИТЬ(Q, v);
6             FOR ALL v' ∈ AL(v) and UNTIL треб[v] = 0 DO
7                 { m := min(ac(v, v'), треб[v]);
8                   ac(v, v') := ac(v, v') − m;
```

```
9                    g(v,v') := g(v,v') + m;
10                   IF треб[v'] = 0 THEN ДОБАВИТЬ(Q,v') }
11                   треб[v] := треб[v] − m;
12                   треб[v'] := треб[v'] + m;
13                   IF ac(v,v') = 0 THEN E(f) := E(f) \ {(v,v')};
14           }
```

PROCEDURE ПРОТЯНУТЬ(y,h)
Эта процедура увеличивает поток g на h единиц, протягиваемых от s к y.
Алгоритм аналогичен процедуре ПРОТОЛКНУТЬ.

Пример 20. *Построим тупиковый поток g для вспомогательной сети $AN(f)$, приведенной на рис. 29. Минимальный потенциал, равный 1, имеют три вершины b, c и f. Выбрав в качестве v вершину b, после завершения процедур ПРОТОЛКНУТЬ(b), ПРОТЯНУТЬ(b),*

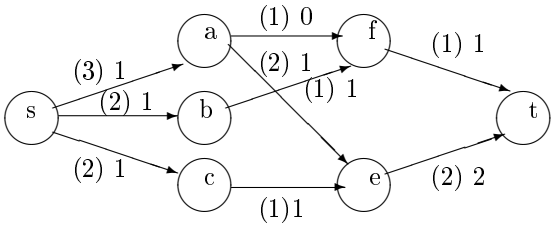


Рис. 30: Вспомогательная сеть $AN(f)$ для сети N с потоком f (рис. 27)

получим поток $g(s,b) = g(b,f) = g(f,t) = 1$. Затем будут удалены вершины b и f и инцидентные им дуги $(f,t),(b,t),(a,f)$ и (s,b). Тогда минимальный потенциал 1 останется у вершины c. После вызова процедур ПРОТОЛКНУТЬ(c), ПРОТЯНУТЬ(c) получим поток $g(s,c) = g(c,e) = g(e,t) = 1$ и будет удалена вершина c с дугами (a,c) и (c,e). После этого потенциал $p(e) = 1$ и после вызова процедур ПРОТОЛКНУТЬ(e), ПРОТЯНУТЬ(e) поток на дуге (e,t) увеличится на 1, что в результате даст $g(e,t) = 2, g(a,e) = 1, g(s,a) = 1$. После удаления e и дуги (e,t) отенциал $p(t) = 0$ и алгоритм построения тупикового потока завершается. Результирующий поток g показан на рисунке 30.

Лемма 7.2. *Дуга e сети $AN(f)$ удаляется из $E(f)$ на некотором шаге только в том случае, если в $AN(f)$ нет прямого увеличивающего пути относительно потока g, проходящего через e.*

Доказательство. Так как для любой дуги e в начале этапа $g(e) = 0$, а затем уменьшение ее пропускной способности и увеличение потока происходят одновременно и на одну и ту же величину (стр. 8 и 9 процедуры ПРОТОЛКНУТЬ), то в каждый момент (после стр. 9) $g(e) + ac(e) = ac'(e)$, где $ac'(e)$ – начальная пропускная способность дуги e. Поэтому, если эта дуга удаляется в стр. 12 процедуры ПРОТОЛКНУТЬ, то $g(e) = ac'(e)$ и e не может входить в прямой увеличивающий путь в $AN(f)$. Если же $e = (u,v)$ удаляется в стр. 13 МАХП, то либо $p(u) = 0$, либо $p(v) = 0$. В первом случае для всякой входящей в u дуги e_1 $g(e_1) = ac'(e_1)$, а во втором дл всякой выходящей из v дуги e_2 $g(e_2) = ac'(e2)$. В обоих случаях это означает, что нет прямого увеличивающего пути через e.

Лемма 7.3. *В конце каждого этапа g – тупиковый поток в $AN(f)$.*

Доказательство. Переход на новый этап происходит, когда в стр. 8 МАХП проверяется, что $p(s) = 0$ или $p(t) = 0$. В первом случае $g(e) = ac'(e)$ для всякой дуги e, выходящей из s, во втором - то же имеет место для всякой дуги e, входящей в t. В обоих случаях это означает, что в $AN(f)$ нет прямого увеличивающего пути.

Лемма 7.4. *Расстояние от s до t в $AN(f+g)$ на произвольном этапе строго больше, чем расстояние от s до t в $AN(f)$ на предыдущем этапе.*

Доказательство. Вспомогательная сеть $AN(f+g)$ совпадает со вспомогательной сетью сети $AN(f)$ относительно потока g (Доказать!). По лемме 3 поток g – тупиковый, т.е. в $AN(f)$ нет прямых увеличивающих путей относительно g. Поэтому все имеющиеся увеличивающие пути "непрямые т.е. их длины больше, чем расстояние от s до t в $AN(f)$, и лемма следует из теоремы 4.б.

Теорема 7.5. *Алгоритм МАХП корректно решает задачу о максимальном потоке для сети $N = (V, E, s, t, c)$ за время $O(n^3)$ $(n = |V|)$.*

Доказательство. Алгоритм МАХП завершается (стр. 4,5), когда в сети $AN(f)$ s и t не связаны. Но тогда в этой сети максимальный поток нулевой и по теореме 4.в $val(f) = maxf$, т.е. f – максимальный поток.

Для оценки времени отметим, что по лемме 4 число этапов не превосходит $|V|$. На каждом этапе построение вспомогательной сети $AN(f)$ алгоритмом ПОВС выполняется за $O(|V|^2)$ шагов. После выполнения процедур ПРОТОЛКНУТЬ и ПРОТЯНУТЬ для вершины v ее потенциал $p(v)$ становится нулевым *(почему ?)* и она удаляется из сети (стр. 13). Поэтому число вызовов каждой из них на одном этапе $< |V|$. Оценим теперь общее число обработок дуг на каждом этапе. При каждой такой обработке дуги (v, v') в стр. 9 процедуры ПРОТОЛКНУТЬ (ПРОТЯНУТЬ) поток $g(v, v')$ увеличивается. Назовем такое увеличение насыщающим, если после него этот поток равен исходной пропускной способности дуги, т.е. $g(v, v') = ac'(v, v')$, а остаточная пропускная способность $ac(v, v') = 0$. Ясно, что для каждой дуги насыщающее увеличение потока может произойти не более одного раза и, следовательно, общее число насыщающих увеличений не превосходит $|E|$. В каждом вызове процедур ПРОТОЛКНУТЬ и ПРОТЯНУТЬ число ненасыщающих увеличений не превосходит числа выбираемых из очереди Q вершин, поскольку для каждой из них такое увеличение может произойти лишь по одной дуге. Так как в процедуре ПРОТОЛКНУТЬ (ПРОТЯНУТЬ) каждая вершина может попасть в очередь Q только 1 раз, то число ненасыщающих увеличений при вызове каждой из этих процедур не превосходит $|V|$. Таким образом, общее число увеличений потока (в стр. 9) на одном этапе не превосходит $|E| + |V|^2$ и каждый этап можно выполнить за время $O(|V|^2)$. Отсюда общее время работы алгоритма МАХП можно оценить как $O(|V|^3)$.

ЗАДАЧА. Написать алгоритм для процедуры ПРОТЯНУТЬ.

7.3 Сети с единичными пропускными способностями

Рассмотрим работу алгоритма МАХП на сетях, в которых пропускные способности всех дуг равны 1.

Лемма 7.5. *Каждый этап алгоритма МАХП, примененного к сети $N = (V, E, s, t, c)$, такой что $c(e) = 1$ для любой дуги e из E, можно выполнить за $O(|E|)$ шагов.*

Доказательство. В этом случае и во вспомогательной сети $AN(f)$ на каждом этапе пропускные способности дуг будут равны 1. Но тогда кажда дуга может обрабатываться в процедурах ПРОТОЛКНУТЬ и ПРОТЯНУТЬ не более 1 раза, после чего она "насыщается"и удаляется из сети.

Лемма 7.6. *В сети N из леммы 5 расстояние l между s и t не может быть больше, чем $2 * |V|/\sqrt{val(fm)}$, где fm – максимальный поток.*

Доказательство. Пусть $V_i = \{v \mid$ расстояние от s до v равно $i\}$. Множество вершин $VV_i = V_0 \cup V_1 \cup ... \cup V_i$ задает разрез в N при $i < l$. Тогда $f(VV_i, V \setminus VV_i) = |\{e \mid e -$ дуга из V_i в $V_{i+1}\}| \geq val(fm)$. Но число дуг из V_i в V_{i+1} не больше, чем $|V_i| * |V_{i+1}|$. Поэтому либо $|V_i| \geq \sqrt{val(fm)}$, либо $|V_{i+1}| \geq \sqrt{val(fm)}$. Отсюда следует, что по крайней мере для каждого второго уровн V_i имеет место $|V_i| \geq \sqrt{val(fm)}$ и $l * \sqrt{val(fm)} \leq 2 * |V|$.

Теорема 7.6. *Для сетей с единичными пропускными способностями дуг время работы алгоритма МАХП не превосходит $O(|E| * |V|^{2/3})$.*

Доказательство. Покажем, что число этапов s не превосходит $|V|^{2/3}$.

а) Если $val(fm) \leq |V|^{2/3}$, то этапов "мало так как на каждом этапе поток увеличивается по крайней мере на 1.

б) Если $val(fm) > |V|^{2/3}$, то пусть i – такой этап, после которого впервые поток превосходит $val(fm) - |V|^{2/3}$. Пусть g – это поток в начале этапа i. Так как величина дополнительного потока $\leq |V|^{2/3}$, то $s - i \leq |V|^{2/3}$. С другой стороны, $i <$ (расстояние от s до t в сети $AN(g)$) $\leq 2 * |V|/\sqrt{val(fm) - val(g)}$ (по лемме 6 и так как величина максимального потока в $AN(g)$ равна $val(fm) - val(g)$ по теореме 4.в). Но по выбору g имеем перед i-ым этапом $val(fm) - val(g) \geq |V|^{2/3}$. Отсюда $i \leq 2 * |V|^{2/3}$ и $s \leq 3 * |V|^{2/3}$. Теперь теорема следует из леммы 5 и этой оценки числа этапов.

Определение 29. *Сеть, у которой пропускные способности всех дуг равны 1, назовем* простой, *если для любой вершины сети степень захода или степень исхода равны 1 или 0.*

Лемма 7.7. *В простой сети N расстояние l между s и t не может быть больше, чем $|V|/val(fm)$, где fm – максимальный поток.*

Доказательство. Определим V_i как в лемме 6. Так как через каждую вершину может проходить не более одной единицы максимального потока, а весь он проходит через каждое V_i, то $val(fm) \leq |V_i|$. Отсюда $l * val(fm) \leq |V|$.

Теорема 7.7. *Для простых сетей время работы алгоритма МАХП не превосходит $O(|E|\sqrt{|V|})$.*

Доказательство. Покажем, что число этапов s не превосходит $\sqrt{|V|}$.

а) Если $val(fm) \leq \sqrt{|V|}$, то этапов "мало так как на каждом этапе поток увеличивается по крайней мере на 1.

б) Если $val(fm) > \sqrt{|V|}$, то пусть i – такой этап, после которого впервые поток превосходит $val(fm) - \sqrt{|V|}$. Пусть g – поток в начале этапа i. Величина максимального потока в $AN(g)$ $\geq \sqrt{|V|}$. Тогда по лемме 7 $l \leq \sqrt{|V|}$. (Докажите, что $AN(f)$ – простая сеть и лемма применима). Поэтому $i \leq \sqrt{|V|}$ и $s - i \leq \sqrt{|V|}$, т.е. $s = O(\sqrt{|V|})$.

7.4 Максимальные паросочетания в двудольных графах

Определение 30. Паросочетание *в неориентированном графе $G = (V, E)$ – это произвольное подмножество ребер M такое, что никакие два ребра из него не имеют общей вершины. Паросочетание наибольшей мощности называется максимальным.*

Напомним определение двудольных графов, рассмотренных в задаче 2.16.

Определение 31. *Граф $G = (V, E)$ называется* двудольным, *если множество его вершин V можно разбить на два непересекающихся подмножества X и Y так, что каждое ребро $e \in E$ имеет вид $e = (x, y)$, где $x \in X$, а $y \in Y$.*

Пусть дан двудольный граф $B = (X, Y, E)$. Определим сеть $N(B) = (W, A, s, t, c)$:
1) вершины – $W = s, t \cup X \cup Y$;
2) дуги – $A = \{(s, x) \mid x \in X\} \cup \{(y, t) \mid y \in Y\} \cup \{(x, y) \mid (x, y) \in E\}$;
3) пропускные способности $c(e) = 1$ для всех $e \in A$.

Замечание. Построенная сеть $N(B)$ - простая.

Лемма 7.8. *Мощность максимального паросочетания в графе B равна величине максимального потока в сети $N(B)$.*

Доказательство. Пусть M – паросочетание в B. Определим поток f в $N(B)$ следующим образом. Для каждой дуги $(u, v) \in M$ положим $f(s, u) = 1$, $f(u, v) = 1$, $f(v, t) = 1$. Тогда f - допустимый поток и $val(f) = |M|$.

Пусть теперь f – максимальный поток в $N(B)$, построенный алгоритмом МАХП. Тогда он имеет целочисленные значения (0 или 1). Пусть M состоит из таких дуг (u, v), для которых $f(u, v) = 1$. Нетрудно проверить, что M – паросочетание и что $|M| = val(f)$.

Теорема 7.8. *Задачу о максимальном паросочетании для двудольных графов можно решить за время $O(|E| * \sqrt{|V|})$.*

Доказательство. Алгоритм поиска максимального паросочетания работает следующим образом.
1) По графу B строим сеть $N(B)$.
2) С помощью алгоритма МАХП находим максимальный поток в $N(B)$.
3) По этому потоку строим максимальное паросочетание в B (по лемме 8).
Так как $N(B)$ – простая сеть, то время 2-го этапа - $O(|E| * \sqrt{|V|})$. Время 1-го этапа – $O(|E|)$, а время 3-го – $O(|V|)$ (п о ч е м у ?). Отсюда следует оценка теоремы.

Пример 21. *Рассмотрим двудольный граф $B = (X, Y, E)$, в котором $X = \{x_1, x_2, x_3, x_4\}$, $Y = \{y_1, y_2, y_3, y_4\}$ $E = \{(x_1, y_1), (x_1, y_2), (x_2, y_4), (x_3, y_1), (x_4, y_3), (x_4, y_4)\}$. Соответствующая ему простая сеть $N(B)$ показана на рис. 31.*

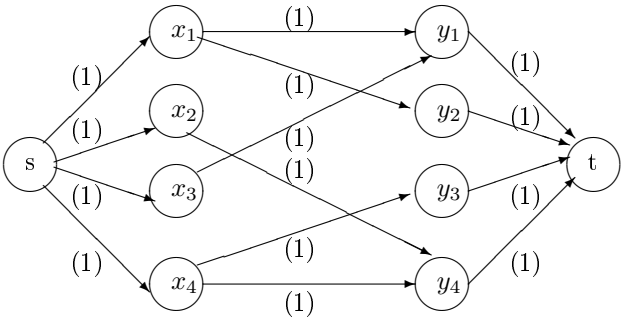


Рис. 31: Сеть $N(B)$

Вспомогательная сеть $AN(B)(0)$ для любой сети $N(B)$ относительно нулевого потока совпадает с самой $N(B)$ (почему ?). Построив тупиковый поток g на $AN(B)$ и добавив его к нулевому потоку, получим на $N(B)$ поток $f(=g)$, показанный на рис. 32.

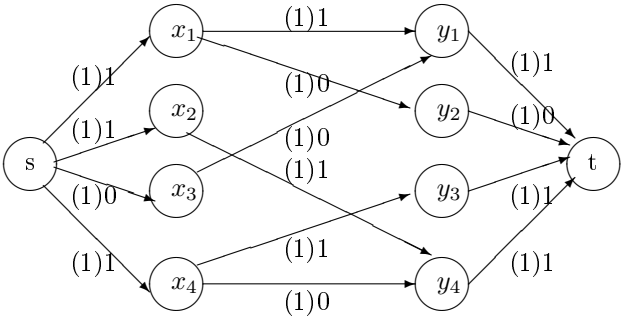


Рис. 32: Сеть $N(B)$ с текущим потоком f

Построим теперь вспомогательную сеть $AN(B)(f)$ для $N(B)$ относительно построенного потока f. Она показана на рис. 33.

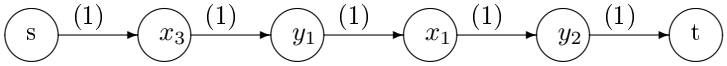


Рис. 33: Сеть $AN(B)(f)$

В этой сети ребро (y_1, x_1) обратное, а остальные прямые. Ясно, что тупиковый поток в этой сети равен 1 на всех ребрах. После его добавления к предыдущему потоку получаем сеть $N(B)$ с новым потоком f, показанную на рис. 34.

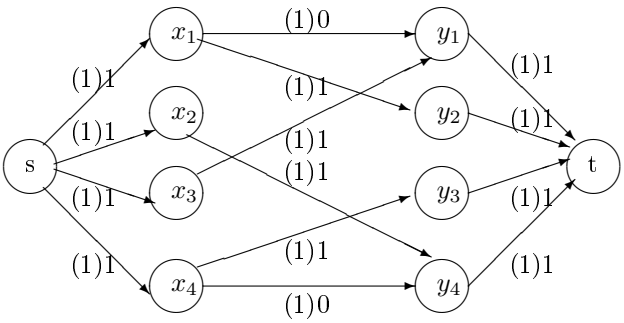


Рис. 34: Сеть $N(B)$ с максимальным потоком f

Этот поток, очевидно, является максимальным (попытка построить вспомогательную сеть поместит в нее лишь вершину s). Ему соответствует максимальное паросочетание $M = \{(x_1, y_2), (x_2, y_4), (x_3, y_1), (x_4, y_3)$ в графе B.

7.4.1 Паросочетания в общих графах

В определении 30 понятие паросочетания и максимального паросочетания было определено для произвольного неориентированного графа $G = (V, E)$. Для взвешенных графов *максимальным* называется паросочетание наибольшего веса. *Совершенным* называется паросочетание, ребра которого инцидентны всем вершинам графа. Оно, например, всегда существует в полном графе четной степени или в графе-цикле четной длины. Для взвешенных графов *минимальное совершенное паросочетание* — это совершенное паросочетание минимального веса. Для поиска максимальных и минимальных совершенных паросочетаний в произвольных неориентированных графах имеется достаточно эффективный алгоритм, предложенный Эдмондсом и Джонсоном (см. например, [5], глава 12, [9], глава 5, [8], глава 9). Его сложность не превосходит $O(|V|^4)$.

Отметим также, что теория паросочетаний в двудольных и произвольных графах имеет много важных приложений в математике, физике, химии и других областях. Многие вопросы этой теории и ее приложений рассмотрены в монографии [8].

7.5 Задачи

Задача 7.1. *Используя алгоритм Форда-Фалкерсона, построить максимальный поток для сети* $N : V = \{s, v1, v2, v3, t\}$, $E = \{(s, v1, 2), (s, v2, 2), (s, v3, 2), (v1, v2, 1), (v1, t, 1), (v2, v3, 2), (v2, t, 1), (v3, t, 3)\}$ *(формат:* $(v, u, c(v, u))$*).*

Задача 7.2. *Построить, используя алгоритм МАХП, тупиковый поток для следующей (вспомогательной) сети (формат:* $(v, u, c(v, u))$*)* : $(s, v1, 7), (s, v2, 6), (s, v3, 5), (v1, v4, 7), (v2, v5, 5), (v3, v6, 4), (v2, v6, 3), (v4, t, 4), (v5, t, 5), (v1, v5, 1), (v6, t, 7)$.

Задача 7.3. *Построить, используя алгоритм МАХП, максимальный поток для следующей сети* $N : V = \{s, v_1, v_2, v_3, v_4, v_5, v_6, t\}$, $E = (s, v_1, 5), (s, v_2, 3), (s, v_3, 5), (v_1, v_4, 7), (v_2, v_5, 5), (v_3, v_6, 4), (v_6, v_2, 3), (v_4, t, 4), (v_5, t, 5), (v_1, v_5, 2), (v_6, t, 3)$. *(формат:* $(v, u, c(v, u))$*).*

Задача 7.4. *Назовем* запасом связности *неориентированного графа минимальное число ребер, которые необходимо удалить, чтобы сделать его несвязным. Например, для дерева это число равно 1, для простого цикла — 2. Чему оно равно для полного n-вершинного графа? Какова сложность "переборного"алгоритма для нахождения запаса связности? Постройте*

алгоритм вычисления запаса связности по графу, использующий сведение этой задачи к задаче нахождения максимального потока и оцените его сложность.

Задача 7.5. *Пусть в сети $N = (V, E, s, t)$ заданы пропускные способности дуг $c(e), e \in E$ и вершин $d(v), v \in V$. Поток через вершину v не должен превышать $d(v)$. Покажите, как задачу о максимальном потоке в такой "обобщенной"сети можно свести к задаче о максимальном потоке в "обычной"сети. Используйте это сведение для решения следующей* задачи о выходе.
Имеется граф $G = (V, E)$, вершины которого расположены в вершинах прямоугольной плоской решетки $V = \{(i, j) \mid i = 1, \ldots, n,\ j = 1, \ldots, n,\ \}$. Ребра графа — это отрезки, соединяющие соседние вершины, в частности, у каждой "внутренней"вершины (i, j) имеется 4 соседа $(i-1, j), (i, j+1), (i+1, j), (i, j-1)$, у 4-х "угловых"вершин — по 2, а у вершин на "границах "графа — по 3. Скажем, что подмножество из m вершин $U \subseteq V$ имеет выход *, если существует m непересекающихся путей, соединяющих вершины U с граничными вершинами графа G. Предложите алгоритм, который по $G = (V, E)$ и подмножеству $U \subseteq V$ определяет имеет ли U выход. Оцените его сложность.*

Задача 7.6. *Предположим, что известен максимальный поток f_m в сети $N = (V, E, s, t; c)$ и пропускные способности всех ребер — целые числа.*

Предположим, что пропускная способность некоторого ребра (u, v) увеличилась на 1: $c'(u, v) = c(u, v) + 1$. Предложите алгоритм, находящий максимальный поток для измененной сети за время $O(|V| + |E|)$. Предложите также алгоритм, решающий ту же задачу в случае уменьшения пропускной способности ребра (u, v) на 1.

Задача 7.7. *Пусть обобщенная сеть - это сеть, для каждой дуги (u, v) которой указана как верхняя $c(u, v)$, так и нижняя $b(u, v)$ граница величины потока. Показать, что задачу о максимальном потоке в обобщенной сети можно свести к задаче о максимальном потоке в "обычной"сети. (Рассмотрите случай, когда для каждой v сумма $b(u, v)$ по входящим в v дугам равна сумме $b(v, w)$ по выходящим из v дугам).*

Задача 7.8. *Показать, что для простой сети N с потоком f , имеющим значения 0 или 1, вспомогательная сеть $AN(f)$ является простой.*

Задача 7.9. *Доказать, что длины всех путей из s в t во вспомогательной сети $AN(f)$ одинаковы и равны длине кратчайшего увеличивающего пути в N с потоком f.*

Задача 7.10. *По договору с рестораном поставщик должен обеспечивать поставку d_i салфеток в i-ый день в течение n дней. Салфетки можно покупать по цене a рублей за штуку или отдавать в стирку старые салфетки. Обычная стирка одной салфетки стоит b руб. и требует двух дней, экспресс-стирка одной салфетки обходится в c руб. и они готовы через один день. Определить оптимальную политику поставщика по закупке и стирке салфеток (т.е. политику, минимизирующую его затраты). Оценить сложность получившегося алгоритма.*

Задача 7.11. *Предложить алгоритм для поиска вершины с минимальным потенциалом в алгоритме МАХП (стр. 12) и оценить его сложность.*

Задача 7.12. *Доказать, что алгоритм ПОВС можно реализовать за время* $O(|V| + |E|)$ *(указание: использовать для представления* ac *и* af *не массивы, а списки, связанные с дугами сети).*

Задача 7.13. *Пусть дана сеть* $N = (V, E, s, t; c)$*, у которой некоторые пропускные способности дуг могут быть отрицательны. Построить алгоритм, который находит максимальный поток в этой сети и оценить его сложность.*
Указание. Рассмотрите сеть $N' = (V', E', s', t'; c')$*:*
$V' = V \cup \{s', t'\}$; $E' = E \cup \{(u, v) \mid (v, u) \in E\} \cup \{(s', v) \mid v \in V\} \cup \{(u, t') \mid u \in V\} \cup \{(s, t),\ (t, s)\}$. *Для каждого ребра* $(u, v) \in E$ *положим* $c'(u, v) = c'(v, u) = (c(u, v) + c(v, u))/2$. $c'(s', u) = \max\{0, (c(V, u) - c(u, V))/2\}$ *для каждой* $u \in V$ *и* $c'(u, t') = \max\{0, (c(u, V) - c(V, u))/2\}$ *для каждой* $u \in V$*. Кроме того, положим* $c'(s', t') = c'(t', s') = \infty$.
а) Докажите, что если в сети N *существует поток, то в сети* N' *все пропускные способности неотрицательны и в ней существует максимальный поток, в котором все ребра, входящие в* t' *насыщены.*
б) По любому потоку в N'*, в котором все ребра, входящие в* t' *насыщены, можно построить поток в* N.

Задача 7.14. *Предложите алгоритм, который по сети* N *и максимальному потоку в ней* f_m *находит такое ребро, увеличение пропускной способности которого приводит к увеличению максимального потока. Оцените его сложность.*

Задача 7.15. *Найти максимальное паросочетание в двудольном графе, используя сведение к задаче о максимальном потоке для простой сети.* $G =< \{v1, v2, v3, v4\}, \{u1, u2, u3, u4, u5\}, E = \{(v1, u1), (v1, u2),\ (v2, u2), (v2, u4), (v3, u1), (v3, u2), (v4, u5), (v4, u3), (v4, u4)\} >$.

Задача 7.16. *Пусть* $G = (V, E)$ *— двудольный граф с равномощными долями, т.е.* $V = X \cup Y$, $X \cap Y = \emptyset$ *и* $|X| = |Y|$. Совершенным паросочетанием *в таком графе называется паросочетание, в которое входят все вершины. Для каждого подмножества вершин* $U \subseteq V$ *определим множество соседних вершин* $N(U) = \{w | \exists u \in U[(u, w) \in E]\}$*. Докажите, что в* G *существует совершенное паросочетание тогда и только тогда, когда для любого* $A \subseteq X$ *выполнено* $|A| \leq |N(A)|$ *(теорема Холла).*

8 NP-полные задачи для графов

8.1 Полиномиальная сводимость и NP-полные задачи

В предыдущих разделах мы рассмотрели много алгоритмических задач для графов и для решения каждой из них находился достаточно эффективный алгоритм. Самые сложные из этих задач — поиск кратчайших путей в произвольном графе и поиск максимального потока в транспортной сети — решались за время $O(n^3)$, где n — число вершин графа. Однако имеется много важных для приложений задач на графах, для которых неизвестны алгоритмы полиномиальной сложности относительно размеров рассматриваемых графов. Эти задачи принадлежат классу так называемых NP-полных задач. Для задач этого класса по мнению большинства исследователей не существует алгоритмов, работающих в полиномиальное время, хотя формально эта гипотеза не доказана и с начала 70-х годов XX века является одной из самых интригующих проблем информатики.

Напомним основные понятия, связанные с NP-полнотой. В качестве алгоритмических проблем будем рассматривать проблемы разрешения, т.е. проблемы вхождения элемента во множество. *Проблемы разрешения* представляют собой пары множеств (D, Y), где D — это множество исходных данных, входов проблемы, являющихся, обычно, строками в некотором конечном алфавите, а Y — это подмножество допустимых, распознаваемых, "хороших" данных из D. Алгоритм (детерминированный) корректно решает такую проблему, если он для каждого входа из Y выдает ответ "Да", а для каждого входа из $D \setminus Y$ — ответ "Нет". Недетерминированный алгоритм для каждого входа порождает дерево вычислений. Он решает проблему, если для всякого входа из Y в этом дереве есть путь вычисления, приводящий к ответу "Да" и ни для какого входа из $D \setminus Y$ такого пути нет (предполагается, что на любом пути алгоритм либо останавливается с ответом "Да", либо вовсе не останавливается). Временем работы недетерминированного алгоритма считается минимальная длина пути, приводящего к ответу "Да".

В качестве основной формальной модели алгоритмов обычно рассматриваются детерминированные и недетерминированные машины Тьюринга (ДМТ и НМТ). Для границы сложности $f(n)$ (n — длина входа) через $DTIME(f)$ ($NTIME(f)$ обозначим класс проблем разрешения, решаемых ДТМ (НТМ) за время, ограниченное функцией $f(n)$, а через $SPACE(f)$ — класс проблем решаемых ДТМ с емкостной сложностью (в памяти), ограниченной функцией $f(n)$.

Проблемы разрешения мы будем классифицировать по сложности с помощью хорошо известных классов: $P = \cup_{k=1}^{\infty} DTIME(n^k)$ — класс проблем, решаемых за полиномиальное время ДТМ, $NP = \cup_{k=1}^{\infty} NTIME(n^k)$ — класс проблем, решаемых за полиномиальное время НТМ, $PSPACE = \cup_{k=1}^{\infty} SPACE(n^k)$ — класс проблем, решаемых в полиномиальной памяти (по теореме Цейтина-Сэвича он совпадает с соответствующим недетерминированным классом). Отметим, что эти классы проблем практически не зависят от выбора машин Тьюринга в качестве вычислительной модели алгоритмов.

В задачах разрешения, связанных с графами, нужно зафиксировать некоторое стандартное представление графов. Выберем в качестве такого представления матрицу смежности. Тогда граф $G = (V, E)$ с n вершинами и $m \leq n^2$ ребрами можно задать строкой в алфавите $\{0, 1\}$ длины n^2. Заметим, что этот выбор тоже не является существенным, так как все рассмотренные в разделе 2 представления графов могут трансформироваться одно в другое за полиномиальное от размера графа время. В частности, матрица инцидентности для G имеет размер nm, а списки смежности – $O((n + m) \log n)$.

Говорят, что проблема разрешения $\Pi = (D, Y)$ *сводится за полиномиальное время к проблеме* $\Pi' = (D', Y')$ (обозначение: $\Pi \leq_p \Pi'$), если существует вычислимая за полиномиальное время функция f такая, что для всякого входа $w \in D$ $f(w) \in D'$ и $w \in Y \Leftrightarrow f(w) \in Y'$. Проблемы Π_1 и Π_2 *полиномиально эквивалентны* ($\Pi_1 \equiv_p \Pi_2$), если $\Pi_1 \leq_p \Pi_2$ и $\Pi_2 \leq_p \Pi_1$. Проблема Π называется *трудной для сложностного класса* C (или C-трудной), если к ней сво-

дится за полиномиальное время всякая проблема из класса C. Π является *C-полной*, если она C-трудная и $\Pi \in C$. Первые примеры NP-полных проблем были построены в работах С.Кука и Л.Левина ([17, 6]. Р.Карп ([20]) расширил список таких проблем, установив, в частности, что NP-полными являются такие проблемы теории графов как размер вершинного покрытия и клики, существование гамильтонова цикла, определение хроматического числа графа. Многочисленные примеры NP-полных проблем, в том числе и задач на графах, приведены в [2].

В следующем предложении мы собрали основные свойства сводимости за полиномиальное время, которые будут далее использоваться для доказательства NP-полноты конкретных задач на графах. Его справедливость непосредственно следует из определений.

Лемма 8.1.
(1) Отношение $\leq_p$ рефлексивно и транзитивно, т.е. является отношением частичного порядка.
(2) Если $\Pi \in P$ и $\Pi' \leq_p \Pi$, то $\Pi' \in P$.
(3) Если Π является NP-полной задачей, $\Pi' \in NP$ и $\Pi \leq_p \Pi'$, то и Π' является NP-полной задачей.

Далее неоднократно будет использована задача *3-КНФ*, у которой D — это множество булевых формул в конъюнктивной нормальной форме, в которых каждая дизъюнкция содержит не более трех литералов, а Y состоит из выполнимых формул такого вида.

Ее NP-полноту установил С. Кук [17]. Естественно, NP-полной является и более общая задача ВЫП $= \{\Phi \mid \Phi$ — *выполнимая булева формула в базисе* $\{\neg, \wedge, \vee\}\}$.

Отметим, что если множество всех входов D ясно из контекста, то проблему $\Pi = (D, Y)$ обычно ассоциируют со множеством ее "решений"Y.

8.2 Полиномиальная разрешимость выполнимости 2-КНФ

Для иллюстрации понятия сводимости за полиномиальное время докажем, что задача распознавания выполнимости булевых формул в конъюнктивной нормальной форме, в которых каждая дизъюнкция содержит не более двух литералов – *2-КНФ* – разрешима за полиномиальное время. Для этого сведем ее к некоторой задаче на графах, разрешимость которой за полиномиальное время следует из результатов раздела 5.4.

Рассмотрим класс ориентированных графов $G = (V = X \cup Y, E)$, у которых множество вершин V разбито на два равномощных занумерованных подмножества $X = \{x_1, \ldots, x_n\}$ и $Y = \{y_1, \ldots, y_n\}$. Назовем такой граф *корректным*, если ни для какого $i \in [1, n]$ в нем нет цикла, включающего вершины x_i и y_i. Пусть *КОР_ГРАФ* это множество всех корректных графов.

Лемма 8.2.
КОР_ГРАФ $\in P$.

Доказательство. Пусть $G = (V = X \cup Y, E)$. В разделе 5.4 был построен алгоритм **ССК**, определяющий за время $O(|V| + |E|)$ все сильно связные компоненты ориентированного графа G. После этого достаточно проверить, что ни в одну из полученных компонент не входят одновременно вершины x_i и y_i. Очевидно, что такую проверку можно осуществить за время $O(|V|)$.

Лемма 8.3.
2-КНФ$\leq_p$ КОР_ГРАФ.

Доказательство. Пусть $\Phi = F_1 \wedge \ldots \wedge F_q$ — это такая КНФ, в которой для каждого $i \in [1,q]$ дизъюнкция $F_i = l_{i1} \vee l_{i2}$, где l_{ij} $(1 \le i \le q, 1 \le j \le 2)$ — литерал из множества $\{x_1, \neg x_1, \ldots, x_n, \neg x_n\}$. Определим по Φ граф $G_\Phi = (V = X \cup Y, E)$ следующим образом:
$X = \{x_1, x_2, \ldots, x_n\}$, $Y = \neg x_1, \neg x_2, \ldots, \neg x_n\}$, $E = \{(\neg l_{i1}, l_{i2}), (\neg l_{i2}, l_{i1}) \mid i \in [1,q]\}$.
(Как обычно, мы считаем, что $\neg\neg l = l$ для любого литерала l). Из этого определения следует, что граф G_Φ строится по формуле Φ за время $O(|\Phi|)$.

Отметим несколько свойств графа G_Φ.

(1) Если в G_Φ есть путь из l_1 в l_2, то в нем есть путь из $\neg l_2$ в $\neg l_1$.

Действительно, по определению, если в G_Φ есть ребро (l_1, l_2), то в нем есть и ребро $(\neg l_2, \neg l_1)$. Отсюда получаем (1) индукцией по длине пути из l_1 в l_2.

Из (1) непосредственно следует свойство

(2) Если вершины l_1 и l_2 лежат в одной компоненте сильной связности G_Φ, то и $\neg l_1$ и $\neg l_2$ также лежат в одной компоненте сильной связности G_Φ.

(3) Если формула Φ истинна на подстановке σ, в G_Φ есть путь из l_1 в l_2 и $\sigma(l_1) = 1$, то и $\sigma(l_2) = 1$.

Действительно, если $(l_1, l_2) \in E$, то в Φ имеется дизъюнкция $F_i = \neg l_1 \vee l_2$. Она истинна на σ только при $\sigma(l_2) = 1$. Отсюда получаем (3) индукцией по длине пути из l_1 в l_2.

Теперь лемма следует из утверждения:

$$G_\Phi \in \textit{КОР\_ГРАФ} \;\Leftrightarrow \Phi \in \text{2-КНФ}.$$

$\Leftarrow$ Предположим, что граф $G_\Phi = (V = X \cup Y, E) \in \textit{КОР\_ГРАФ}$. Рассмотрим его *конденсацию (граф компонент)* $G^K = (V^K, E^K)$. Граф G^K ациклический. Поэтому его вершины (компоненты сильной связности G) можно расположить в обратном топологическом порядке: $K_1, K_2, \ldots, K_m$ (см. задачу 5.9). Это означает, что для любого ребра $(K_p, K_s) \in E^K$ $s < p$. Определим поэтапно значения литералов из формулы Φ следующим образом.
На первом этапе всем литералам с вершинами в K_1 присвоим значение 1.
На i-м этапе $(1 < i \le m)$ присвоим всем литералам из K_i значение 0, если в K_i есть хотя бы одна вершина l, для которой $\neg l$ уже присвоено некоторое значение. Если такой вершины l в K_i нет, то присвоим всем литералам с вершинами в K_1 значение 1. Обозначим через $\sigma(l)$ – значение литерала l после этапа m.

Пусть для переменной $x \in \{x_1, \ldots, x_n\}$ литерал $x \in K_i$, а литерал $\neg x \in K_j$. Тогда при $i < j$ $\sigma(x) = 1, \sigma(\neg x) = 0$, а при $i > j$ $\sigma(x) = 0, \sigma(\neg x) = 1$.

Действительно, если бы при $i < j$ значение $\sigma(x) = 0$, то в компоненте K_i нашелся бы литерал l такой, что $\neg l$ был определен раньше, т.е. $\neg l \in K_r$ и $r < i$. Но тогда по свойству (2) и $\neg x \in K_r$, что противоречит условию $i < j$. Аналогично, при $i > j$ значение $\sigma(\neg x)$ не может быть равно 0. Таким образом, σ задает корректное присваивание значений литералам из Φ.

Покажем теперь, что для каждой дизъюнкции $F_i = l_{i1} \vee l_{i2}, (1 \le i \le m)$ $\sigma(l_{i1}) = 1$ или $\sigma(l_{i2}) = 1$.

Предположим, что $\sigma(l_{i1}) = \sigma(l_{i2}) = 0$. Пусть $l_{i1} \in K_p, l_{i2} \in K_r$. Так как $\sigma(l_{i1}) = 0$, то $\neg l_{i1} \in K_s$ для некоторого $s < p$. Аналогично, $\neg l_{i2} \in K_t$ для некоторого $t < r$. Поскольку имеется ребро $(\neg l_{i1}, l_{i2})$, то $r < s$, а из-за ребра $(\neg l_{i2}, l_{i1})$ $p < t$. Но эта система неравенств противоречива: $s < p < t < r < s$. Следовательно, и наше исходное предположение о ложности F_i на σ неверно.

Таким образом, если $G_\Phi \in \textit{КОР\_ГРАФ}$, то формула Φ выполнима.

$\Leftarrow$ Предположим, что формула Φ выполнима на корректной подстановке $\sigma : \{x_1, \ldots, x_n\} \to \{0,1\}$. Если $G_\Phi \notin \textit{КОР\_ГРАФ}$, то в G_Φ имеется цикл, содержащий некоторую переменную x_i и ее отрицание $\neg x_i$. Так как одно из значений $\sigma(x_i)$ или $\sigma(\neg x_i)$ равно 1, то по свойству (3) и другое также должно быть равно 1, что противоречит выбору σ.
□

Из лемм 8.2 и 8.3 по пункту 2 леммы 8.1 следует

Теорема 8.1. 2-КНФ $\in P$.

Более того, из доказательств лемм 8.2 и 8.3 можно извлечь следующий алгоритм для проверки выполнимости *2-КНФ*.

Алгоритм **2-КНФ-ВЫПОЛНИМОСТЬ:**

Вход: *2-КНФ* Ф с переменными $x_1, \ldots, x_n$.

1. По Ф построить граф G_Φ как в лемме 8.2;
2. С помощью алгоритма **ССК** построить *конденсацию (граф компонент)* $G^K_\Phi = (V^K, E^K)$ графа G_Φ;
3. **IF** существует $i \in [1, n]$ такое, что литералы x_i и $\neg x_i$ находятся в одной компоненте
4. **THEN Return** ("не выполнима")
5. **ELSE**
6. {Используя алгоритм **ПОГ+пост** расположить вершины V^K в обратном топологическом порядке: $K_1, K_2, \ldots, K_m$;
7. **FOR EACH** $l \in K_1$ **DO** $\sigma(l) := 1$;
8. **FOR** $i = 2$ **TO** m **DO**
9. **IF** в K_i есть литерал l, для которого $\sigma(\neg l)$ уже определено
10. **THEN FOR EACH** $l \in K_i$ **DO** $\sigma(l) := 0$
11. **ELSE FOR EACH** $l \in K_i$ **DO** $\sigma(l) := 1$
12. };
13. **Return** σ.

Теорема 8.2. *Алгоритм* **2-КНФ-ВЫПОЛНИМОСТЬ** *по произвольной выполнимой 2-КНФ* Ф *возвращает подстановку* σ, *на которой* Ф *истинна, а если* Ф *тождественно ложна, то возвращается ответ "не выполнима". Время работы алгоритма –* $O(|\Phi|)$.

Доказательство корректности алгоритма **2-КНФ-ВЫПОЛНИМОСТЬ** содержится в леммах 8.2 и 8.3. Оценка времени следует из линейной сложности алгоритмов **ПОГ+пост** и **ССК**, а также связанных с ними алгоритмов построения конденсации графа (см. задачу 5.12) и топологической сортировки (см. задачу 5.9).

8.3 Клика, независимое множество, вершинное покрытие

Определение 32. *k-кликой в неориентированном графе $G = (V, E)$ называется полный k-вершинный подграф G.*

Пусть КЛИКА $=\{(G, k) |$ *в графе G имеется $k-$клика* $\}$.

Теорема 8.3. *Задача КЛИКА является NP-полной.*

Доказательство. Недетерминированный алгоритм, который "угадывает"произвольные k вершин V, а затем детерминированно проверяет образуют ли они k-клику, очевидно, работает в полиномиальное от размера V время. Следовательно, КЛИКА$\in NP$.

Для доказательства NP-трудности покажем, что 3-КНФ $\leq_p$ КЛИКА. Пусть $\Phi = F_1 \wedge \ldots \wedge F_q$ — это такая КНФ, в которой для каждого $i \in [1, q]$ дизъюнкция $F_i = l_{i1} \vee l_{i2} \vee l_{i3}$, где l_{ij} $(1 \leq i \leq q, 1 \leq j \leq 3)$ — литерал из множества $\{x_1, \neg x_1, \ldots, x_n, \neg x_n\}$.

Определим по Ф граф $G = (V, E)$ следующим образом. Каждой дизъюнкции F_i сопоставим три вершины, соответствующие входящим в нее литералам, и соединим ребрами все пары вершин, соответствующие непротивоположным литералам из разных дизъюнкций. Более формально, пусть $V = \{(i, j) |\ 1 \leq i \leq q, 1 \leq j \leq 3\}$ и $E = \{((i, j), (k, l)) \mid i \neq k, l_{ij} \neq \neg l_{k,l}\}$. Из этого

определения следует, что $|V| \leq |\Phi|$ и $|E| \leq |\Phi|^2$ и G можно построить по Φ за полиномиальное время. Теперь теорема следует из утверждения:

$$(G, q) \in \text{КЛИКА} \Leftrightarrow \Phi \in \text{3-КНФ}.$$

Действительно, предположим, что в G имеется q-клика $G' = (V', E')$. Тогда для каждого $i \in [1, q]$ имеется ровно одна вершина вида (i, j_i), входящая в V'. Определим значения $\sigma(x_k)$ для переменных из Φ следующим образом: $\sigma(x_k) = 1 \Leftrightarrow$ существует вершина $(i, j) \in V'$ такая, что $l_{ij} = x_k$, иначе $\sigma(x_k) = 0$. Заметим, что подстановка σ определена корректно, так как в V' не могут одновременно входить вершины (i, j) и (s, l), для которых $l_{ij} = x_k$, а $l_{sl} = \neg x_k$, так как такие вершины между собой не соединены в G'. Нетрудно понять, что значение $\sigma(\Phi) = 1$, так как в каждую дизъюнкцию F_i входит литерал l_{ij_i}, для которого $\sigma(l_{ij_i}) = 1$.

Обратно, если $\sigma(\Phi) = 1$ для некоторого набора значений булевых переменных σ, то в каждой дизъюнкции F_i имеется литерал l_{ij_i}, для которого $\sigma(l_{ij_i}) = 1$. Вершины G, соответствующие этим литералам и образуют q-клику в G.□

С задачей КЛИКА тесно связаны еще две известные задачи теории графов.

Определение 33. Независимым множеством *в неориентированном графе $G = (V, E)$ называется такое подмножество вершин $V' \subseteq V$, что для любых $u, v \in V'$ ребро (u, v) не принадлежит E.*

Максимальная мощность независимого множества графа G называется числом независимости *G и обозначается как $\alpha(G)$.*

Задача НЕЗАВИСИМОЕ МНОЖЕСТВО состоит в том, чтобы по графу $G = (V, E)$ и натуральному числу $k \leq |V|$ определить, имеется ли в G независимое множество мощности $\geq k$.

Определение 34. Вершинным покрытием *в неориентированном графе $G = (V, E)$ называется такое подмножество вершин $V' \subseteq V$, что для любого ребра $(u, v) \in E$ хотя бы одна из вершин u или v принадлежит V'.*

Задача ВЕРШИННОЕ ПОКРЫТИЕ состоит в том, чтобы по графу $G = (V, E)$ и натуральному числу $k \leq |V|$ определить, имеется ли в G вершинное покрытие мощности не более k.

Следующее утверждение устанавливает связь между указанными тремя задачами.

Лемма 8.4. *Для любого неориентированного графа $G = (V, E)$ и подмножества вершин $V' \subseteq V$ следующие утверждения эквивалентны:*
(а) V' является кликой в G;
(б) V' является независимым множеством в дополнении $\bar{G}$ графа G, где $\bar{G} = (V, \bar{E}), \bar{E} = \{(u, v) | u, v \in V, (u, v) \notin E\}$.
(в) $V \setminus V'$ является вершинным покрытием в $\bar{G}$.

Доказательство предоставляется читателю (см. задачу 8.1)

Пример 22. *Проиллюстрируем лемму 8.4 на взаимно дополнительных графах G и $\bar{G}$, приведенных на рис. 35.*

В графе G имеется максимальная 4-клика $V' = \{a, b, d, e\}$. Это же множество вершин является максимальным независимым множеством в графе $\bar{G}$. В этом же графе вершины $V \setminus V' = \{c, f\}$ образуют минимальное вершинное покрытие. В обратном направлении: в $\bar{G}$ имеется две 3-клики $V_1 = \{a, c, f\}$ и $V_2 = \{d, c, f\}$. Каждое из этих множеств является максимальным независимым множеством в графе G, а их дополнения $V \setminus V_1 = \{b, d, e\}$ и $V \setminus V_2 = \{a, b, e\}$ являются вершинными покрытиями в G.

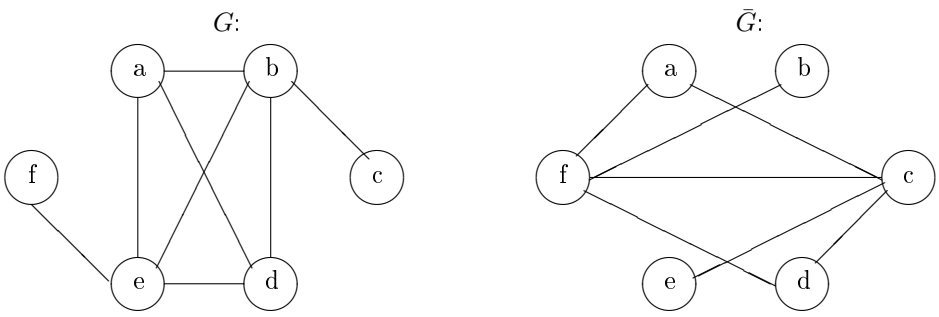


Рис. 35: Графы G и $\bar{G}$

Из леммы 8.4 непосредственно следует, что
$(G,k) \in$ КЛИКА $\Leftrightarrow (\bar{G},k) \in$ НЕЗАВИСИМОЕ МНОЖЕСТВО $\Leftrightarrow (\bar{G},|V|-k) \in$ ВЕРШИННОЕ ПОКРЫТИЕ.
Поскольку граф $\bar{G}$ строится по G за полиномиальное время, то эти три задачи полиномиально эквивалентны. Тгда из теоремы 8.3 получаем

Следствие 8.3.1. *Задачи НЕЗАВИСИМОЕ МНОЖЕСТВО и ВЕРШИННОЕ ПОКРЫТИЕ являются NP-полными.*

В приложениях эти задачи чаще всего рассматриваются как задачи на поиск максимальной клики, максимального независимого множества или минимального вершинного покрытия. Например, в социологии в графе, представляющем отношение взаимного доверия в группе людей, часто требуется выделить максимальную подгруппу взаимно доверяющих друг другу людей. В графе, вершины которого представляют места возможного расположения магазинов некоторой торговой сети, а ребра соединяют пары мест, в которых магазины будут излишне конкурировать, максимальное независимое множество определяет возможное размещение наибольшего числа взаимно не конкурирующих магазинов. На графе, представляющем сеть дорог, минимальное вершинное покрытие определяет оптимальное расположение дорожных служб, позволяющее контролировать все дороги сети.

Имея алгоритмы для соответствующих задач разрешения, можно с помощью очевидной дихотомии получить алгоритмы, определяющие размер максимальной клики, максимального независимого множества или минимального вершинного покрытия, время работы которых превосходит время распознающих алгоритмов не более чем в $O(\log(|V|)$ раз.

С их помощью можно эффективно находить и сами эти множества вершин. Действительно, пусть алгоритм *Размер-макс-клики(G)* возвращает максимальный размер клики в графе G с n вершинами за время $t_{max}(n)$. Тогда поиск максимальной клики можно провести с помощью следующего алгоритма **МАКС_КЛИКА**.

1. K:= *Размер-макс-клики(G)*; $CL := \emptyset$;
2. Удалить из G все вершины степени меньше K и инцидентные им ребра;
   Пусть $F = (\{u_1,\ldots,u_r\}, E_F)$ — это получившийся граф с вершинами степени $\geq K$;
3. $i := 0$;
4. **WHILE** $|CL| < K$ **DO**
5. { **WHILE** *Размер-макс-клики(F)* $=K$ **DO**
6. $\{i := i+1;\quad v := u_i$;
7. **IF** $v \in V_F \setminus CL$ **THEN**

8. Удалить из F вершину v и все инцидентные ей ребра
9. };
// v входит в максимальную клику:
10. $CL := CL \cup \{v\}$;
// возвращаем v в F и оставляем лишь связанные с ней вершины:
11. $V_F := CL \cup \{w \mid (v,w) \in E_F\}$;
12. $E_F := \{(u,w) \mid u,w \in V_F, (u,w) \in E_F\}$;
13. $F := (V_F, E_F)$
14. };
15. **Return** CL.

Теорема 8.4. *Алгоритм* **МАКС_КЛИКА** *вычисляет в переменной CL максимальную клику графа G с n вершинами за время $O(nt_{max}(n))$.*

Доказательство. Для доказательства корректности заметим что, ни одна вершина степени меньше K не входит в максимальную клику. Поэтому после удаления таких вершин в стр. 2 граф F содержит ту же максимальную клику, что и граф G. Далее в цикле в стр. 5-9 выявляется вершина v, удаление которой из F приводит к уменьшению размера максимальной клики. Тогда эта вершина принадлежит максимальной клике и связана с вершинами, попавшими в CL раньше. Поэтому после ее возвращения в F и удаления из F всех, не связанных с v вершин в стр. 11-13, граф F снова содержит ту же максимальную клику размера K, что и граф G. Каждое исполнение цикла в стр. 5-9 приводит к добавлению одной вершины в CL. Всего таких вызовов будет K, после чего алгоритм завершит работу, выдав клику CL максимального размера K.

Для оценки времени отметим, что алгоритм *Размер-макс-клики(F)* вызывается в стр. 5 не более $r \leq n$ раз, каждая из вершин G обрабатывается в цикле в стр. 5-9 не более одного раза и операторы в стр. 10-14 выполняются не более $K \leq n$ раз. Отсюда следует, что время поиска максимальной клики не более $O(nt_{max}(n))$.□

Таким образом, если бы нашелся алгоритм полиномиальной сложности, решающий задачу КЛИКА, то и поиск максимальной клики можно было бы осуществить за полиномиальное время. Но существование такого алгоритма представляется маловероятным.

8.4 Гамильтонов цикл

В 1859 г. У. Гамильтон придумал игру «Кругосветное путешествие», состоящую в отыскании такого пути, проходящего через все вершины (города, пункты назначения) графа додекаэдра, изображенного ниже на рис. 36. Требовалось однократно посетить каждую вершину и возвратиться в исходную. Пути, обладающие таким свойством, называются гамильтоновыми циклами. Нумерация вершин задает такой цикл на рис. 36: $1 \to 2 \to 3 \to \ldots \to 20 \to 1$.

Определение 35. *Гамильтонов цикл в (не)ориентированном графе G — это цикл, проходящий через каждую вершину G по одному разу. Гамильтонов путь в (не)ориентированном графе G между вершинами w и v — это путь, начинающийся в w, заканчивающийся в v, который проходит через каждую вершину G по одному разу.*

Нас интересует, как по графу определить, имеется ли в нем гамильтонов цикл. Пусть
ГАМ_ЦИКЛ=$\{G \mid$ *G — неориентированный граф, в котором есть гамильтонов цикл*$\}$,
ОР_ГАМ_ЦИКЛ=$\{G \mid$ *G — ориентированный граф, в котором есть гамильтонов цикл*$\}$,

Теорема 8.5. *Задача ОР_ГАМ_ЦИКЛ является NP-полной.*

Доказательство. Алгоритм, который вначале недетерминированно угадывает перестановку вершин графа G, а затем детерминированно проверяет, является ли эта перестановка циклом

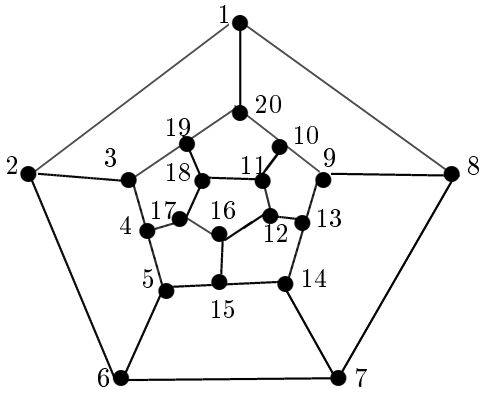


Рис. 36: Граф додекаэдра и его гамильтонов цикл

в G, решает задачу за недетерминированное полиномиальное время. Таким образом, задача ОР_ГАМ_ЦИКЛ $\in NP$.

Для доказательства нижней оценки покажем, что

ВЕРШИННОЕ ПОКРЫТИЕ $\leq_p$ ОР_ГАМ_ЦИКЛ.

Пусть $G = (V, E)$ — неориентированный граф и $k \leq |V|$ — натуральное число. Определим по ним ориентированный граф $G_D = (V_D, E_D)$ следующим образом. Пусть $V = \{v_1, \ldots, v_n\}$. Ребро из E с концами v_i и v_j будем обозначать как $e_{i,j}$. Для каждой вершины v_i упорядочим инцидентные этой вершине ребра: $e_{i,j_1}, \ldots, e_{i,j_{r(i)}}$, где $r(i)$ — это степень v_i. Множество вершин ориентированного графа определим как $V_D = \{a_1, \ldots, a_k\} \cup \{[v, e, b] \mid v \in V, e \in E, b \in \{0, 1\}, e \textit{ инцидентно } v\}$.

Таким образом, каждому ребру $e = e_{i,j} (= e_{j,i})$ графа G в G_D соответствуют 4 вершины: $A = [v_i, e_{i,j}, 0], B = [v_i, e_{i,j}, 1], C = [v_j, e_{i,j}, 0], D = [v_j, e_{i,j}, 1]$. Между ними имеются следующие ребра: $(A, C), (C, A), (B, D), (D, B), (A, B), (C, D)$. Кроме того, если $j = j_1$, то в вершину A ведут ребра из всех вершин $a_l, 1 \leq l \leq k$, а если $j = j_m, m > 1$, то в в вершину A входит ребро из вершины $[v_i, e_{i,j_{m-1}}, 1]$. Из вершины $[v_i, e_{i,j_{r(i)}}, 1]$ идут ребра во все вершины $a_l, 1 \leq l \leq k$. Иначе говоря, G_D содержит k вершин $\{a_1, \ldots, a_k\}$ и для каждой вершины $v_i \in V$ включает цепь из $2r(i)$ вершин, соответствующих ребрам инцидентным v_i: $[v_i, e_{i,j_1}, 0] \to [v_i, e_{i,j_1}, 1] \to \ldots \to [v_i, e_{i,j_m}, 0] \to [v_i, e_{i,j_m}, 1] \to \ldots \to [v_i, e_{i,j_{r(i)}}, 0] \to [v_i, e_{i,j_{r(i)}}, 1]$. В первую вершину этой цепи ведут ребра из всех $a_s, 1 \leq s \leq k$, а из последней – $[v_i, e_{i,j_{r(i)}}, 1]$ ребра ведут во все $a_s, 1 \leq s \leq k$. Кроме того, попарно соединены вершины из разных цепей, с одинаковой второй и третьей компонентами.

Покажем, что *в G имеется вершинное покрытие из k вершин тогда и только тогда, когда в G_D имеется гамильтонов цикл.*

$\Rightarrow$ Пусть в G имеется вершинное покрытие V' из k вершин. Не ограничивая общности, будем считать, что $V' = \{v_1, \ldots, v_k\}$. Определим вначале в G_D цикл p, проходящий по одному разу через все вершины $a_i (1 \leq i \leq k)$ и все вершины в цепях соответствующих $\{v_1, \ldots, v_k\}$: $p = a_1 \to [v_1, e_{1,j_1}, 0] \to [v_1, e_{i,j_1}, 1] \ldots \to [v_1, e_{1,j_{r(1)}}, 0] \to [v_1, e_{1,j_{r(1)}}, 1] \to a_2 \to \ldots \to a_k \to [v_k, e_{k,j'_1}, 0] \to [v_k, e_{k,j'_1}, 1] \to \ldots \to [v_k, e_{k,j'_{r(k)}}, 0] \to [v_k, e_{k,j'_{r(k)}}, 1] \to a_1$. Этот цикл из каждой вершины $a_i (1 \leq i \leq k)$ идет в первую вершину цепи, соответствующей v_i, проходит по этой цепи и из ее последней вершины переходит в вершину $a_{i+1 \mod k}$. Покажем, как включить в этот цикл остальные вершины из V_D. Пусть $v_j \notin V'$ и $[v_j, e, 0], [v_j, e, 1]$ — две вершины цепи, соответствующей v_j в G_D. Пусть ребро $e = (v_j, v_i)$. Так как V' вершинное покрытие, то второй конец этого ребра $v_i \in V'$. Тогда в цикле p имеется ребро $([v_i, e, 0], [v_i, e, 1])$. Заменим его на путь $([v_i, e, 0], [v_j, e, 0], [v_j, e, 1][v_i, e, 1]$. Проделав такие замены для всех не вошедших в p пар

вершин $[v_j, e, 0], [v_j, e, 1] (j > k)$, получим гамильтонов цикл в G_D.

$\Leftarrow$ Пусть в графе G_D имеется гамильтонов цикл p. Зафиксируем k чисел $l_1, \ldots, l_k$ таких, что в цикле p после каждой вершины $a_t, 1 \leq t \leq k$, идет первая вершина цепи, соответствующей v_{l_t}. Докажем, что множество вершин $V' = \{v_{l_1}, \ldots, v_{l_k}\}$ является вершинным покрытием в G.

Действительно, для каждой $v_i \in V'$ цикл p, попав в первую вершину цепи, соответствующей v_i, обязан пройти через вершины этой цепи в том же порядке, в котором они в ней расположены. При этом, для каждого ребра $e = (v_i, v_j)$, придя "сверху"в вершину типа A, он должен выйти из блока вершин A, B, C, D "вниз"из вершины B, т.е. либо по пути $A \to B$, либо по пути $A \to C \to D \to B$. Все другие варианты привели бы к тому, что одна из вершин B или C не попала бы в цикл. Поэтому для каждой пройденной в цикле p вершины вида $[v, e, b]$ хотя бы один из концов ребра e принадлежит V'. Поскольку p проходит все такие вершины, то V' является вершинным покрытием в G. $\square$

Проиллюстрируем конструкцию теоремы на следующем примере.

Пример 23. *На рис.37 показан пример неориентированного графа* $G = \{a, b, c, d\}, \{e_1 = (a, b), e_2 = (b, c), e_3 = (c, d)\}$ *и соответствующего ориентированного графа* G_D*, построенного по* G *и задаче о вершинном покрытии этого графа мощности* $k = 2$.

Вершинные покрытие из 2-х вершин в G *и гамильтоновы циклы в* G_D *соответствуют друг другу. Например, покрытию* $\{a, c\}$ *соответствует гамильтонов цикл* $a_1, [a, e_1, 0], [b, e_1, 0], [b, e_1, 1], [a, e_1, 1], a_2, [c, e_3, 0], [d, e_3, 0], [d, e_3, 1], [c, e_3, 1], [c, e_2, 0], [b, e_2, 0], [b, e_2, 1], [c, e_2, 1], a_1$ *в графе* G_D*. А гамильтоновому циклу* $a_1, [c, e_3, 0], [d, e_3, 0], [d, e_3, 1], [c, e_3, 1], [c, e_2, 0], [c, e_2, 1], a_2, [b, e_1, 0], [a, e_1, 0], [a, e_1, 1], [b, e_1, 1], [b, e_2, 0], [b, e_2, 1], a_1$ *в графе* G_D *в исходном графе* G *соответствует вершинное покрытие* $\{c, b\}$.

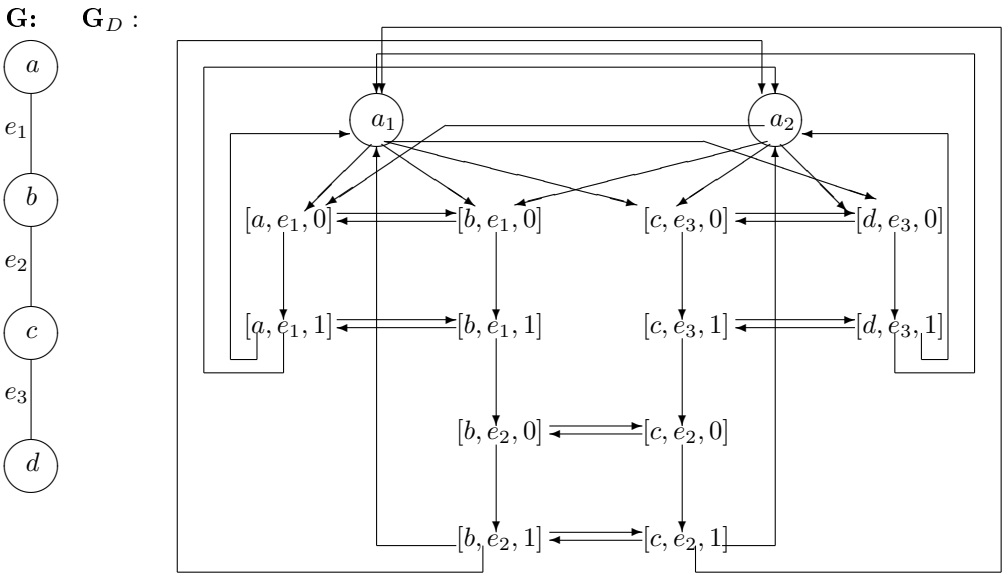


Рис. 37: Пример сведения вершинного покрытия к ориентированному гамильтонову циклу

Отметим, что граф G_D в доказательстве теоремы 8.5 является сильно связным: между любыми двумя вершинами в нем имеется (ориентированный) путь. Таким образом справедливо

Следствие. *Задача ОР_ГАМ_ЦИКЛ является NP-полной для подкласса сильно связных ориентированных графов.*

Покажем теперь, как задачу о существовании гамильтонова цикла в ориентированном графе можно свести к задаче о существовании гамильтонова цикла в неориентированном графе.

Теорема 8.6. *Задача ГАМ_ЦИКЛ является NP-полной.*

Доказательство. Алгоритм для верхней оценки не отличается от алгоритма для задачи ОР_ГАМ_ЦИКЛ. Докажем, что

$$\text{ОР\_ГАМ\_ЦИКЛ} \leq_p \text{ГАМ\_ЦИКЛ}.$$

Пусть $G = (V, E)$ — ориентированный граф.Мы хотим построить по нему неориентированный граф $G' = (V', E')$, в котором будет гамильтонов цикл тогда и только тогда, когда он есть в G. Простая идея – превратить все ориентированные ребра в неориентированные – не срабатывает, так как в G' могут образоваться "лишние"циклы из разнонаправленных ребер G. Чтобы зафиксировать "направление"ребер, утроим каждую вершину: $V' = \{[v,0],[v,1],[v,2] \mid v \in V\}$ и соединим $[v,1]$ ребрами с $[v,0]$ и $[v,2]$. Тогда гамильтонов цикл может попасть в эту тройку вершин либо через $[v,0]$, либо через $[v,2]$. В первом случае он должен далее пройти по пути $[v,0],[v,1],[v,2]$, а во вторм — по пути $[v,2],[v,1],[v,0]$ (иначе вершина $[v,1]$ останется вне цикла). Каждое ориентированное ребро $(v,u) \in E$ "промоделируем" неориентированным ребром $([v,2],[u,0])$. Таким образом, $E' = \{([v,0],[v,1]),([v,1],[v,2]) \mid v \in V\} \cup \{([v,2],[u,0]) \mid (v,u) \in E\}$.

Предположим, что в G имеется (ориентированный) гамильтонов цикл $p = v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_n, v_1$. Тогда в графе G' ему соответствует (неориентированный) гамильтонов цикл $p' = [v_1,0],[v_1,1],[v_1,2],[v_2,0],[v_2,1],[v_2,2],\ldots,[v_i,0],[v_i,1],[v_i,2],[v_{i+1},0],[v_{i+1},1],[v_{i+1},2],\ldots,[v_n,0],[v_n,1],[v_n,2],[v_1,0]$.

Обратно, предположим, что в в G' имеется гамильтонов цикл p'. Как мы выше отметили, тройка вершин $[v_1,0],[v_1,1],[v_1,2]$ должна входить в этот цикл в одном из двух порядков: (1) : $p' = \ldots,[v_1,0],[v_1,1],[v_2,2],\ldots$ или (2) : $p' = \ldots,[v_1,2],[v_1,1],[v_1,2],\ldots$. Из определения G' следует, что в обоих случаях для остальных вершин порядок будет такой же. Отсюда следует, что для некоторой перестановки вершин $p = v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_n, v_1$ в случае (1) цикл $p' = [v_1,0],[v_1,1],[v_1,2],[v_2,0],[v_2,1],[v_2,2],\ldots,[v_i,0],[v_i,1],[v_i,2],[v_{i+1},0],[v_{i+1},1],[v_{i+1},2],\ldots,[v_n,0],[v_n,1],[v_n,2],[v_1,0]$, а в случае (2) $p' = [v_1,2],[v_1,1],[v_1,0],[v_2,2],[v_2,1],[v_2,0],\ldots,[v_i,2],[v_i,1],[v_i,0],[v_{i+1},2],[v_{i+1},1],[v_{i+1},0],\ldots,[v_n,2],[v_n,1],[v_n,0],[v_1,2]$. Тогда в случае (1) имеем $(v_i, v_{i+1(\mod n)}) \in E$ при $i \in [1,n]$, а в случае (2) $(v_{i+1}, v_i) \in E$ при $i \in [1,n-1]$ и $(v_n, v_1) \in E$. Поэтому обоих случаях p задает гамильтонов цикл в графе G: в случае (1) — в прямом направлении, в случае (2) — в обратном. □

Проиллюстрируем конструкцию теоремы на следующем примере.

Пример 24. *На рис.38 показан пример ориентированного графа $G = \{a,b,c,d\},\{(a,b),(a,c),(b,c),(c,d),(d,a),(d,b)\}$ и соответствующего ориентированного графа G', построенного по G.*

В этом примере, в частности, ориентированному гамильтонову циклу a,b,c,d,a в графе G соответствует неориентированный гамильтонов цикл $[a,0],[a,1],[a,2],[b,0],[b,1],[b,2],[c,0],[c,1],[c,2],[d,0],[d,1],[d,2],[a,0]$ в графе G', а гамильтонову циклу $[c,2],[c,1],[c,0],[b,2],[b,1],[b,0],[a,2],[a,1],[a,0],[d,2],[d,1],[d,0],[c,2]$ в G' соответствует гамильтонов цикл $c \leftarrow b \leftarrow a \leftarrow d \leftarrow c$ (с точностью до сдвига он совпадает с циклом a,b,c,d,a).

NP-полными являются также задачи о существовании гамильтовых путей между двумя заданными вершинами в ориентированных и неориентированных графах (см. задачу 8.10).

8.5 Задача коммивояжера

Задача коммивояжера является одной из самых знаменитых NP-полных задач. Как задача оптимизации она формулируется так: *найти цикл минимальной длины (стоимости, веса), проходящий через все вершины нагруженного (не)ориентированного графа.*

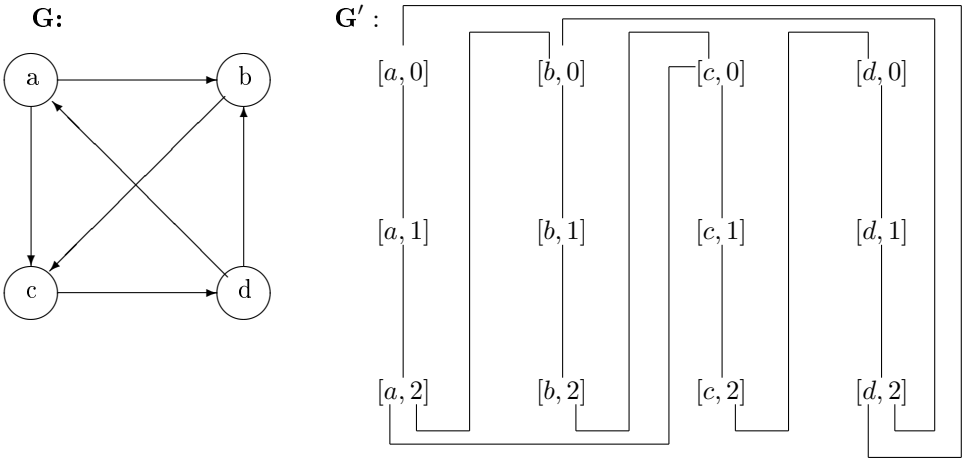


Рис. 38: Пример сведения ориентированного гамильтонова цикла к неориентированному

Соответствующая задача на распознавание задается множеством КОММИВОЯЖЕР $= \{(G,k) \mid G=(V,E)-$ *граф с функцией длин ребер* $c: E \to R$, *содержащий гамильтонов цикл длины* $\leq k\}$.

Эта задача получила свое название из-за следующей интерпретации: некоторый разъезжающий торговец (коммивояжер) планирует посетить со своим товаром определенное множество городов. Каков кратчайший маршрут, позволяющий ему объехать все города и вернуться домой, минимизировав таким образом свои затраты?

У задачи коммивояжера имеется множество приложений, связанных с транспортными задачами и задачами поиска оптимальных путей в различных ситуациях. Другая область приложения — оптимизация путей движения при сборке оборудования. Представим, например, руку робота, которая должна запаять все контакты на печатной плате. Движение руки по кратчайшему маршруту, который посещает каждую точку пайки один раз, и будет оптимальным для робота.

Еще один пример: диспетчер воздушной обстановки в аэропорту наблюдает на экране локатора светящиеся точки – воздушные суда. Чтобы определить их текущие координаты, нужно щелкнуть по каждой из этих точек курсором мыши. Оптимальным будет выбор кратчайшего пути движения курсора по точкам. Относительно него можно, например, оценивать качество работы диспетчера на тренажерах.

Используя задачу о гамильтоновом цикле нетрудно установить NP-полноту задачи о коммивояжере.

Теорема 8.7. *Задача КОММИВОЯЖЕР является NP-полной.*

Доказательство. Как и для задачи о гамильтоновом цикле верхняя оценка обеспечисвается алгоритмом, который вначале недетерминированно угадывает перестановку вершин графа G, а затем детерминированно проверяет, является ли эта перестановка циклом длины $\leq k$.

Для доказательства NP-трудности покажем, что

(*) ГАМ_ЦИКЛ $\leq_p$ КОММИВОЯЖЕР.

Пусть $G = (V, E)$ — неориентированный граф с n вершинами. Определим по нему полный нагруженный граф $G' = (V, E' = V \times V)$ со следующей функцией длины ребер:

$$c(u,v) = \begin{cases} 1, & \textit{если } (u,v) \in E \\ 2, & \textit{если } (u,v) \notin E. \end{cases}$$

Из этого определения непосредственно следует, что в графе G имеется гамильтонов цикл тогда и только тогда, когда в графе G' имеется гамильтонов цикл длины n. Это доказывает справедливость утверждения (*) и теоремы. □

Отметим, что построенный в доказательстве граф G' с длиной ребер c удовлетворяет неравенству треугольника: для любых трех вершин $u, w, v \in V'$ $\rho(u,v) \leq \rho(u,w) + \rho(w,v)$. Таким образом, справедливо

Следствие. *Задача КОММИВОЯЖЕР является NP-полной для подкласса нагруженных графов, удоволетворяющих неравенству треугольника.*

8.6 Раскраска вершин графа

В задаче о раскраске графа требуется по неориентированному графу $G = (V, E)$ найти минимальное число цветов, которыми можно раскрасить вершины графа так, чтобы смежные вершины были раскрашены в разные цвета. Назовем такую раскраску вершин графа *правильной.* Минимальное число цветов, достаточных для правильной раскраски графа G, называется *хроматическим числом графа* и обозначается $\chi(G)$. Если хроматическое число $\chi(G)$ графа G равно k, то G называется *k-хроматическим графом.*

Например, всякий цикл нечетной длины является 3-хроматическим. Полный граф K_n является n-хроматическим, так как никакие две вершины в нем не могут быть окрашены в один цвет. Очевидно также, что если G содержит n-клику (т.е. подграф, изоморфный K_n), то $\chi(G) \geq n$. Вершины графа, раскрашенные при правильной раскраске в один цвет, образуют независимое множество. Отсюда следует, что $\chi(G) \geq |V|/\alpha(G)$.

Задача о раскраске возникает, в частности, при оптимизации распределения памяти при компиляции программ. Вершинами рассматриваемого графа являются переменные программы, ребра соединяют переменные, времена существования которых пересекаются. Тогда количество цветов в минимальной раскраске равно минимальному числу ячеек памяти, достаточному для выполнения программы, поскольку все переменные, получившие одинаковый цвет, могут быть размещены в одной и той же ячейке.

Приведем еще один пример применения раскраски графа. Предположим, что на некотором химическом предприятии выпускаются вещества $P_1, \ldots, P_n$, при этом известно, что некоторые пары веществ нельзя хранить в одном помещении. Определим граф с вершинами $P_1, \ldots, P_n$ и ребрами, соединяющими вещества, которые не должны храниться вместе. Тогда хроматическое число этого графа равно минимальному числу различных помещений, необходимых для хранения продукции данного предприятия.

Задача о раскраске используется также при решении ряда задач теории расписаний и в задачах разбиения множества данных на кластеры близких в том или ином смысле данных.

Одной из самых знаменитых проблем в истории теории графов является *задача о четырех красках:* можно ли раскрасить каждую карту четырьмя красками так, чтобы соседние страны были окрашены в разные цвета? В терминах графов: верно ли, что для каждого плоского графа G его хроматическое число $\chi(G) \leq 4$? Эта задача была сформулирована в письме де Моргана Гамильтону в 1852г. и решена в 1976 Аппелем и Хакеном, которые доказали возможность такой раскраски, существенно использовав компьютерную программу. Построены также эффективные алгоритмы для раскраски плоских графов в 4 цвета.

В форме задачи на распознавание задачу раскраски можно переформулировать с помощью множества пар РАСКРАСКА $= \{(G = (V, E), k) \mid$ *существует правильная раскраска вершин неориентированного графа* G *в* k *цветов*$\} = \{(G = (V, E), k) \mid$ *существует раскраска* $c : V \to \{1, 2, \ldots, k\}$ *такая, что для любого ребра* $(u, v) \in E : c(u) \neq c(v)\}$.

При $k = 2$ имеется простой критерий бихроматичности (двудольности) графа и эффективный алгоритм раскраски в 2 цвета (см. задачу 2.16). Однако уже при $k = 3$ задача становится NP-полной даже для плоских графов. Пусть 3-РАСКРАСКА $= \{G \mid$ *существует правильная раскраска вершин* G *в 3 цвета*$\}$.

Теорема 8.8. *Задача 3-РАСКРАСКА является NP-полной.*

Доказательство. Недетерминированный алгоритм, который "угадывает"раскраску вершин G в 3 цвета, а затем детерминированно проверяет ее правильность, очевидно, работает в полиномиальное от размера G время. Следовательно, 3-РАСКРАСКА $\in NP$.

Для доказательства NP-трудности покажем, что 3-КНФ $\leq_p$ 3-РАСКРАСКА . Пусть $\Phi = F_1 \wedge \ldots \wedge F_q$ — это такая КНФ, в которой для каждого $i \in [1, q]$ дизъюнкция $F_i = l_{i1} \vee l_{i2} \vee l_{i3}$, где l_{ij} $(1 \leq i \leq q, 1 \leq j \leq 3)$ — литерал из множества $\{x_1, \neg x_1, \ldots, x_n, \neg x_n\}$.

Определим по ней граф $G = (V, E)$ следующим образом. Множество вершин V включает три специальные вершины T (истина), F (ложь) и R ("красная"), все литералы и по 5 вспомогательных вершин для каждой дизъюнкции F_i из Φ:

$$V = \{T, F, R\} \cup \{x_1, \neg x_1, \ldots, x_n, \neg x_n\} \cup \{A_i, B_i, C_i, D_i, E_i \mid 1 \leq i \leq q\}.$$

Определим множество ребер E.

1) Вершины T, F, R образуют треугольник.

2) Для каждой переменной $x_j, 1 \leq j \leq n$ вершины $x_j, \neg x_j$ и R также образуют треугольник.

Тогда с каждой переменной $x_j, 1 \leq j \leq n$ связан подграф $G(x_j)$ с множеством ребер $E(x_j)$, показанный на рис. 39 слева.

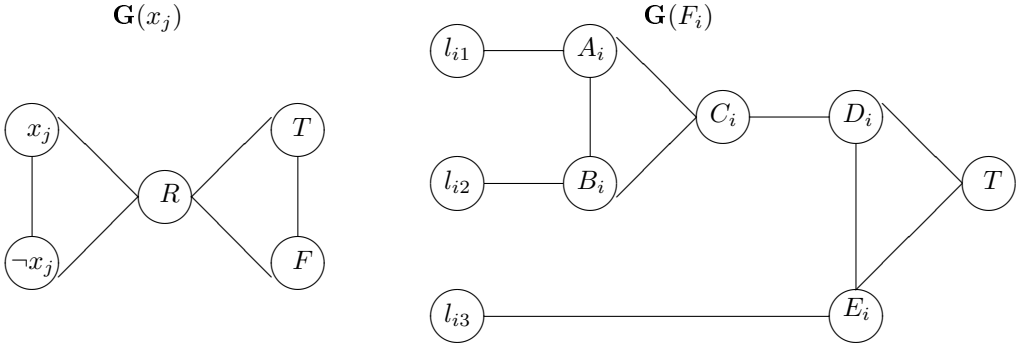


Рис. 39: Подграфы G для переменной x_j и дизъюнкции F_i

3) Для каждой дизъюнкции $F_i = l_{i1} \vee l_{i2} \vee l_{i3}$ граф G включает подграф $G(F_i)$ с множеством из 10 ребер $E(F_i)$, показанный на рис. 39 справа.

Таким образом, $E = \{(T, F), (T, R), (F, R)\} \cup \bigcup_{j=1}^{n} \{(x_j, \neg x_j), (x_j, R), (\neg x_j, R)\}$
$\cup \bigcup_{i=1}^{q} \{(l_{i1}, A_i), (l_{i2}, B_i), (l_{i3}, E_i), (A_i, B_i), (A_i, C_i), (B_i, C_i), (C_i, D_i), (D_i, E_i), (D_i, T), (E_i, T)\}$.

Покажем, что $\Phi \in$ 3-КНФ $\Leftrightarrow$ $G \in$ 3-РАСКРАСКА.

Предположим, что Φ выполняется на некоторой подстановке значений переменных $\sigma : \{x_1, \ldots, x_n\} \to \{0, 1\}$. Построим раскраску c вершин G в три цвета $\{0, 1, 2\}$ следующим образом:

(c1) $c(T) = 1,\ c(F) = 0,\ c(R) = 2$.

(c2) $c(x_j) = \sigma(x_j),\ 1 \leq j \leq n,\ c(\neg x_j) = 1 - c(x_j)$.

(c3i) Для каждой дизъюнкции $F_i, 1 \le i \le q$, определим цвета вспомогательных вершин A_i, B_i, C_i, D_i, E_i исходя из раскраски входящих в нее литералов.
(а) Если $c(l_{i3}) = 1$, то положим $c(E_i) = 0,\ c(D_i) = 2,\ c(A_i) = 1 - c(l_{i1}),\ c(B_i) = 2,\ c(C_i) = c(l_{i1})$.
(б) Если $c(l_{i3}) = 0$, то положим $c(E_i) = 2, c(D_i) = 0$. Поскольку $\sigma(F_i) = 1$ и $\sigma(l_{i3}) = 0$, то либо (б1) оба оставшихся литерала истинны, т.е. $\sigma(l_{i1}) = \sigma(l_{i2}) = 1$, либо (б2) истинен один из них, т.е. $\sigma(l_{i1}) \ne \sigma(l_{i2})$. Тогда в случае (б1) положим $c(A_i) = 0, c(B_i) = 2, c(C_i) = 1$. А в случае (б2) пусть $c(A_i) = 1 - c(l_{i1}), c(B_i) = 1 - c(l_{i2}), c(C_i) = 2$.

Непосредственно проверяется, что определения (c1) и (c2) обеспечивают правильную раскраску подграфов $G(x_j), 1 \le j \le n$, а определения (c3i) — правильную раскраску каждого из подграфов $G(F_i), 1 \le i \le q$.

Предположим теперь, что существует правильная раскраска $c : E \to \{0, 1, 2\}$ вершин графа G в три цвета. Не ограничивая общности, мы можем считать, что $c(F) = 0,\ c(T) = 1,\ c(R) = 2$. Из определения подграфов $G(x_j), 1 \le j \le n$ следует, что из каждой пары вершин-литералов $x_j, \neg x_j$ одна окрашена в цвет 0, а другая — в цвет 1. Определим значения переменных σ в соответствии с этой раскраской: $\sigma(x_j) = c(x_j), 1 \le j \le n$. Покажем, что при таких значениях формула Φ истинна, т.е. $\sigma(\Phi) = 1$. Для этого достаточно установить, что для каждой F_i в подграфе $G(F_i)$ хотя бы одна из вершин l_{i1}, l_{i2}, l_{i3} окрашена в цвет 1. Действительно, если $c(l_{i3}) = 0$, то обязательно $c(E_i) = 2$ и $c(D_i) = 0$. Если бы оба литерала l_{i1} и l_{i2} оказались ложными, то цвета их вершин $c(l_{i1})$ и $c(l_{i2})$ равнялись 0. В этом случае каждая из трех вершин треугольника $A_iB_iC_i$ должна быть окрашена в один из двух цветов 1 или 2, что противоречит правильности раскраски c. Итак, в каждой дизъюнкции F_i имеется литерал l_{ik} со значением $\sigma(l_{ik}) = 1$. Следовательно, $\sigma(\Phi) = 1$. □

Из этой теоремы немедленно следует NP-полнота и более общей задачи РАСКРАСКА. Нетрудно заметить, что построенный в доказательстве теоремы 8.8 граф G может не быть плоским. Но следующая лемма позволяет свести задачу раскраски произвольного графа в 3 цвета к аналогичной задаче для плоских графов.

Лемма 8.5. *Для любого графа G можно построить плоский граф G' такой, что G можно раскрасить в 3 цвета тогда и только тогда, когда G' можно раскрасить в 3 цвета.*

Для доказательства рассмотрим следующий плоский граф W.

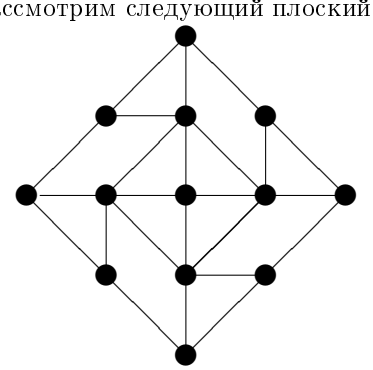


Рис. 40: Граф W

Непосредственно проверяется, что
а) при любой окраске W в 3 цвета противоположные вершины будут окрашены в одинаковый цвет;
б) любая окраска противоположных вершин в одинаковые цвета может быть расширена до корректной окраски всех вершин W в 3 цвета.

В частности, на рис. 41 показаны две раскраски W в 3 цвета: слева противоположные вершины имеют разные цвета 0 и 1, а справа они окрашены в один цвет 2. Все остальные варианты

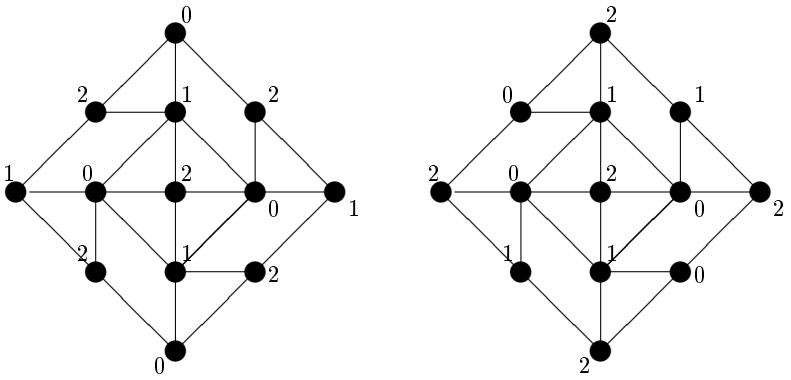


Рис. 41: Две раскраски графа W

раскрасок противоположных вершин можно получить из этих двух раскрасок перестановкой цветов.

Теперь можно использовать граф W для превращения G в плоский граф G'. Для этого для каждого ребра $(v, u) \in E$ заменим каждую точку его пересечения с другим ребром на экземпляр графа W. Склеим вершины прилегающих внутренних "углов" этх копий W, один из внешних углов склеим с v, а другой соединим ребром с u. На рис. 42 приведен пример преобразования ребра (v, u) с тремя точками пересечения с другими ребрами.

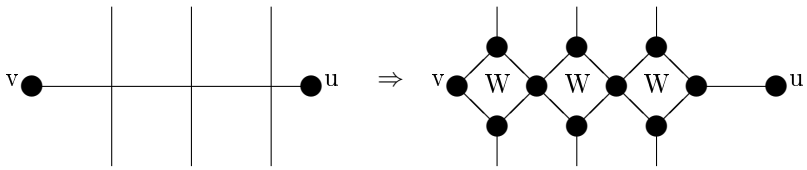


Рис. 42: Преобразование в плоский граф

Получившийся в результате описанного преобразования граф G' будет плоским и строится по G за полиномиальное время. Из свойств (а) и (б) следует, что $G \in$ 3-РАСКРАСКА $\Leftrightarrow$ $G' \in$ 3-РАСКРАСКА. Поэтому справедливо

Следствие. *Задача раскраски плоских графов в три цвета является NP-полной.*

Имеется еще большое число других интересных и важных проблем теории графов, которые оказались NP-полными. В частности, уже в вышедшей в 1976г. основополагающей монографии по теории NP-полноты [2] перечислены более 100 таких проблем. Некоторые из них приведены в следующих задачах.

Отметим, что при доказательстве NP-полноты некоторых задач на графах, кроме рассмотренных выше задач, используются известные NP-полные задачи РАЗБИЕНИЕ и 3-СОЧЕТАНИЕ. Задача РАЗБИЕНИЕ состоит в определении по набору целых чисел A существования его разбиения на два поднабора с одинаковой суммой:
РАЗБИЕНИЕ $= \{A = \{a_1, a_2, \ldots, a_n\} \mid \exists A' \subset A(\sum_{a_i \in A'} a_i = \sum_{a_i \in A \setminus A'} a_i\}$.

В задаче 3-СОЧЕТАНИЕ заданы три множества X, Y и W одинаковой мощности q и множество троек $M \subseteq X \times Y \times W$. Требуется узнать существует ли в M совершенное 3-сочетание, т.е. такое подмножество $M' \subseteq M$, что $|M'| = q$ и каждый элемент из X, Y и W входит в какую-нибудь тройку из M'.

8.7 Задачи

Задача 8.1. *Докажите лемму 8.4.*

Задача 8.2. *Какая задача КЛИКА соответствует следующей 3-КНФ:*
$\Phi = (X \vee Y \vee Z)\ \&(X \vee \neg Y \vee \neg Z)\ \&\ (\neg X \vee Y \vee Z)\ \&\ (\neg X \vee Y \vee \neg Z)$ *?*
Найдите соответствие между некоторым набором значений переменных, на котором Φ *истинна, и 4-кликой.*

Задача 8.3. *Для следующего экземпляра задачи ВЕРШИННОЕ_ПОКРЫТИЕ*
$G = (\{1,2,3,4,5\}, \{(1,2),(1,3),(2,4),(3,4),(5,3)\}),\ k = 2,$
построить эквивалентный экземпляр задачи ОРИЕНТ_ГАМИЛЬТОНОВ_ЦИКЛ и для некоторого 2-покрытия указать соответствующий гамильтонов цикл.

Задача 8.4. *Для экземпляра задачи ОРИЕНТ_ГАМИЛЬТОНОВ_ЦИКЛ*
$G = (V = \{1,2,3,4,5\},\ E = \{(1,4),(1,5),(2,3),(2,5),(4,2),(3,5),(3,1),(4,2)\})$
построить эквивалентный экземпляр G' *задачи НЕОРИЕНТ_ГАМИЛЬТОНОВ_ЦИКЛ и для некоторого гамильтонова цикла в* G *указать соответствующий гамильтонов цикл в* G'.

Задача 8.5. *Предложите эффективный алгоритм, который находит раскраску неориентированного графа в 2 цвета (если такая раскраска существует). Оцените его сложность.*

Задача 8.6. *Какая задача 3-РАСКРАСКА соответствует следующей 3-КНФ:*
$\Phi = (X \vee Y \vee Z)\ \wedge (X \vee \neg Y \vee \neg Z)\ \wedge\ (\neg X \vee Y \vee U)\ \wedge\ (\neg X \vee Y \vee \neg Z) \wedge (\neg Y \vee Z \vee \neg U)$ *?*
Найдите соответствие между некоторым набором значений переменных, на котором Φ *истинна, и раскраской графа в 3 цвета.*

Задача 8.7. Изоморфное вложение.
Докажите, что задача ИЗОМОРФИЗМ_ПОДГАРФУ= $\{(G_1, G_2) \mid$ *граф* G_1 *содержит подграф, изоморфный* $G_2\}$ *является NP-полной.*
Указание. *Используйте задачу КЛИКА.*

Задача 8.8. Наибольший общий подграф
Докажите, что задача МАКС_ОБЩИЙ_ПОДГРАФ $= \{(G_1, G_2, k) \mid G_1$ *и* G_2 *– неориентированные графы, содержащие изоморфные* k*-вершинные подграфы* $\}$ *является NP-полной.*

Задача 8.9. *Докажите, что задача нахождения простого цикла максимальной длины в нагруженном ориентированном графе является NP-полной. Пусть МАКС_ЦИКЛ* $= \{(G, k) \mid$ G *— нагруженный ориентированный граф, содержащий простой цикл длины* $\geq k\}$ *является NP-полной.*
Указание. *Сведите к МАКС_ЦИКЛ задачу ОР_ГАМ_ЦИКЛ.*

Задача 8.10. *Предложите сведение задачи ГАМ_ЦИКЛ к задаче выполнимости булевых формул ВЫП.*
Указание. *Для каждого ребра введите булеву переменную истинную, если это ребро входит в цикл, и опишите формулами условия прохождения гамильтонова цикла через каждую вершину.*

Задача 8.11. Гамильтонов путь
Докажите, что следующие задачи о гамильтоновых путях являются NP-полными.
(а) ГАМИЛЬТОНОВ_ПУТЬ_МЕЖДУ_ВЕРШИНАМИ $= \{(G = (V, E), w, v) \mid$ *в (не)ориентированном графе* G *имеется гамильтонов путь между вершинами* w *и* v $\}$.
(б) ГАМИЛЬТОНОВ_ПУТЬ$= \{(G = (V, E)) \mid$ *в графе* G *имеется имеется гамильтонов путь между некоторой парой вершин*$\}$.

Задача 8.12. Остовное дерево ограниченной степени
Докажите, что задача ОСТ_ДЕРЕВО_ОГР_СТЕПЕНИ $= \{(G = (V, E), k) \mid G$ *– неориентированный граф,* $k \in N$*, в* G *имеется остовное дерево, степени всех вершин которого* $\leq k\}$ *является NP-полной.*
Указание. *Сведите к этой задаче приведенную выше задачу ГАМИЛЬТОНОВ_ПУТЬ.*

Задача 8.13. Кратчайшие простые пути
Докажите, что задача КРАТЧАЙШИЙ_ПРОСТОЙ_ПУТЬ$= \{(G = (V, E), w, v, k) \mid$ *в (не)ориентированном графе* G *имеется простой путь между вершинами* w *и* v *длины* $\leq k\}$ *является NP-полной.*
Указание. *Сведите к этой задаче приведенную выше задачу ГАМИЛЬТОНОВ_ПУТЬ_МЕЖДУ-_ВЕРШИНАМИ.*

Примечание. *Алгоритм Беллмана-Форда из п. 6.2.2 находит длины кратчайших путей в графах без циклов отрицательной длины (либо обнаруживает наличие таких циклов). Если в графе имееются циклы отрицательной длины, то длина кратчайшего пути между двумя вершинами может оказаться равной* $-\infty$*. Тем не менее, длина кратчайшего* простого *пути определена корректно, так как имеется лишь конечное число таких путей. Однако задача поиска таких путей оказывается трудной.*

Задача 8.14. Изоморфный остов
Докажите, что задача ИЗОМОРФНЫЙ_ОСТОВ $= \{(G = (V, E), T = (V_T, E_T)) \mid$ *неориентированный граф* G *содержит остовное дерево, изоморфное* $T\}$ *является NP-полной.*

Задача 8.15. *Докажите NP-полноту задачи "Кратчайший путь с ограничениями по весу".*
Вход: *граф* $G = (V, E)$*, функция* $l(e)$*, задающая целочисленную длину ребер из* E*, функция* $w(e)$*, задающая целочисленый вес ребер из* E*, вершины* s *и* t *из* V *и положительные целые числа* K *и* W. Вопрос: *Существует ли в* G *простой путь из* s t *веса не более* W *и длины не более* K*?*
(Указание: к этой задаче сводится задача РАЗБИЕНИЕ).

Задача 8.16. Сельский почтальон
Сельский почтальон хочет минимизировать длину маршрута, который обязательно должен включить некоторые улицы, на которых проживают симпатичные девушки.

Докажите, что задача СЕЛЬСКИЙ_ПОЧТАЛЬОН $= \{< (G = (V, E), E' \subseteq E, c : E \to N, B > \mid$ *в графе* G *с длиной ребер с существует цикл длины* $\leq B$*, включающий все ребра из* $E'\}$ *является NP-полной.*
Указание. *Сведите к этой задаче задачу ГАМИЛЬТОНОВ_ЦИКЛ.*

9 Что делать с NP-полными проблемами?

Как мы установили в предыдущем разделе, многие практически важные проблемы теории графов являются NP-полными, т.е. в общем случае требуют, по-видимому, для своего решения экспоненциального времени. Такими же также являются многие алгоритмические проблемы логики, алгебры, теоретического программирования, искусственного интеллекта и др. областей (см. [2]).

Как же поступать, обнаружив, что интересующая нас проблема является NP-полной? Что можно предпринять для ее практического решения? Отметим три основных подхода для ответа на этот вопрос:

- **уточнение класса входных данных:** возможно экземпляры задач, которые требуется решать, удовлетворяют каким-либо дополнительным условиям, т.е. входят в некоторую более просто решаемую подпроблему;
- **эвристические алгоритмы:** в ряде случаев позволяют быстро находить решение проблемы, не гарантируя при этом его оптимальности (к ним относятся, в частности, алгоритмы локального поиска и их различные вариации);
- **аппроксимационные алгоритмы:** могут обеспечить достаточно эффективное и достаточно точное, в том или ином смысле, решение .

Разумеется, эти подходы можно объединять, например, для решения уточненной проблемы использовать эвристический или аппроксимационный алгоритм. Кроме того, полезными могут оказаться *переборные алгоритмы с "интеллектуальными" возвратами* и *стохастические алгоритмы*, которые обеспечивают быстрое и достаточное хорошее решение "в среднем".

Приведем несколько примеров использования указанных подходов.

9.1 Аппроксимация для задачи ВЕРШИННОЕ ПОКРЫТИЕ

Оптимизационный вариант задачи ВЕРШИННОЕ ПОКРЫТИЕ состоит в определении минимального множества вершин, в котором каждое ребро имеет хоть один конец. Рассмотрим следующий простой алгоритм для построения вершинного покрытия.

Алгоритм Аппр_ВП.
Вход:$G = (V, E)$ – неориентированный граф.
Выход: C – вершинное покрытие для G.

1. $C := \emptyset$;
2. **WHILE** $E \neq \emptyset$ **DO**
3. { взять произвольное ребро (u, v) из E;
4. $C := C \cup \{u, v\}$;
5. $E := E \setminus \{(x, y) \mid (x \in \{u, v\})$ **OR** $(y \in \{u, v\})\}$;
6. };
7. **Return**(C).

Теорема 9.1. *Алгоритм* **Аппр_ВП** *по графу* $G = (V, E)$ *строит его вершинное покрытие* C, *которое больше минимального вершинного покрытия не более чем в два раза. Время работы алгоритма* **Аппр_ВП** $- O(|V| + |E|)$.

Доказательство. Действительно, поскольку каждое ребро (x, y) удаляется из E в стр. 5 лишь тогда, когда хотя бы один из его концов входит в C, то в конце работы C является вершинныи покрытием. Для каждой пары вершин $\{u, v\}$, попадающей в C в стр. 4, хотя бы

одна из этих вершин должна принадлежать любому, в том числе и минимальному, вершинному покрытию G. Следовательно, число вершин C превосходит число вершин в минимальном вершинном покрытии не более чем в 2 раза. Оценка сложности очевидна. □

Казалось бы, естественно попытаться улучшить алгоритм **Аппр_ВП**, добавляя в C в стр.4 не два конца очередного ребра, а один, т. е. заменить стр. 4 на

4'. $C; = C \cup \{u\}$;

Но полученный с помощью такой "оптимизации" алгоритм **Аппр_ВП1** может привести к существенной потере качества аппроксимации. Рассмотрим для каждого натурального $n > 1$ двудольный граф $B_n = (X \cup Y, E)$, где $X = \{x_1, \ldots, x_n\}$, а другая доля Y состоит из n непересекающихся подмножеств $Y = \cup_{i=1}^{n} Y_i$. Каждая вершина y из Y_i связана с i вершинами X, причем никакие две вершины из Y_i не имеют общих соседей в X. Таким образом, $Y_i = \{y_{ij} \mid 1 \le j < \lfloor n/i \rfloor\}$, $E_i = \{(y_{ij}, x_k) \mid 1 \le j < \lfloor n/i \rfloor, i(j-1)+j \le k \le ij\}$, $E = \cup_{i=1}^{n} E_i$, а общее число вершин в Y равно $n + \lfloor n/2 \rfloor + \lfloor n/3 \rfloor + \ldots + 1 = O(n \log n)$.

Минимальное вершинное покрытие в графе B_n, очевидно, образует множество X и его мощность равна n. Алгоритм **Аппр_ВП** поместит в C (при любом порядке рассмотрения ребер) $2n$ вершин. Если же применить алгоритм **Аппр_ВП1** со стр. 4', то в C могут попасть все вершины из Y. Таким образом, полученное покрытие будет в $O(\log n)$ раз хуже минимального.

Все же идею добавления в C одного конца для каждого расматриваемого ребра можно использовать, если этот конец выбирать "случайно". Зафиксируем некоторый порядок ребер графа $G = (V, E)$, $E = \{e_1, \ldots, e_m\}$. Рассмотрим следующий вероятностный алгоритм.

Алгоритм Аппр_ВП_вер.
Вход:$G = (V, E = \{e_1, \ldots, e_m\})$ – неориентированный граф.
Выход: C – вершинное покрытие для G.

1. $C := \emptyset$;
2. **WHILE** $E \neq \emptyset$ **DO**
3. { взять первое по списку ребро $e_i = (u, v)$ из E;
4. с вероятностью 0.5 выбрать в качестве x одну из вершин u, v;
5. $C := C \cup \{x\}$;
6. $E := E \setminus \{(z, y) \mid (z = x)$ **OR** $(y = x)\}$;
7. };
8. **Return**(C).

Теорема 9.2. *Алгоритм* **Аппр_ВП_вер** *по графу* $G = (V, E)$ *строит его вершинное покрытие* C*, которое в среднем больше минимального вершинного покрытия не более чем в два раза. Время работы алгоритма* **Аппр_ВП_вер** – $O(|V| + |E|)$.

Доказательство. Поскольку ребро удаляется из E в стр. 6 только, когда один из его концов попал в C, а после завершения работы алгоритма $E = \emptyset$, то C является вершинным покрытием. Зафиксируем некоторое минимальное вершинное покрытие $C^* = \{x_1, \ldots, x_k\} \subseteq V$. Из каждой пары вершин u, v, рассматривающейся в стр. 3, по крайней мере одна входит в C^*. Вероятность ее успешного выбора в стр. 4 в качестве x равна 0.5. Поэтому среднее число попыток для k таких успешных выборов равно $2k$. Но это и есть средний размер C. □

Еще одна "естественная" эвристика для поиска минимального вершинного покрытия состоит в поочередном выборе и удалении вершин максимальной (оставшейся) степени. При этом на каждом шаге помещаемая в покрытие вершина будет "покрывать" максимально возможное число оставшихся ребер. Однако и эта эвристика "проваливается" на графах B_n. Действительно, согласно ей на первом этапе в покрытие попадут вершины из Y_n, на втором этапе могут

попасть вершины из Y_{n-1}, на третьем — вершины из Y_{n-2} и т.д. В результате построенное покрытие будет содержать все вершины из Y и окажется в $O(\log n)$ раз больше минимального.

9.2 Аппроксимации для задачи коммивояжера

В разделе 8.5 мы установили NP-полноту задачи распознавания КОММИВОЯЖЕР. В этом – мы рассмотрим ее как задачу оптимизации:
найти цикл минимальной длины (стоимости, веса), проходящий через все вершины полного нагруженного неориентированного графа $G = (V, E = V \times V)$ *с матрицей длин ребер* $C = (c_{ij}), 1 \leq i, j, \leq |V|)$.

Заметим, что мы всегда можем рассматривать полные графы, положив длину отсутствующих ребер равной ∞ (или числу превосходящему сумму длин всех "настояших" ребер).

Обозначим через $OPT_K(C)$ длину кратчайшего гамильтонова пути в графе с матрицей расстояний C, а через $A_K(C)$ – длину гамильтонова пути, выдаваемого алгоритмом A. Как показывает следующее утверждение, задача коммивояжера не допускает никакой сколь-нибудь эффективной полиномиальной аппроксимации.

Теорема 9.3. *Если* $P \neq NP$, *то для всякой функции* $f(n) = c^n$ *(*c *- константа) для задачи КОММИВОЯЖЕРа не существует полиномиального аппроксимационного алгоритма* A, *у которого для всякой матрицы расстояний* C *меет место неравенство*

$$A_K(C) \leq f(n) OPT_K(C)$$

(т.е. любой алгоритм, работающий в полиномиальное время, для некоторых графов выдает результат в экспоненту раз хуже оптимального).

Доказательство. Изменим немного сведение задачи ГАМ_ЦИКЛ из доказательства теоремы 8.7.

Пусть $G = (V, E)$ — неориентированный граф с n вершинами. Определим по нему полный нагруженный граф $G' = (V, E' = V \times V)$ со следующей функцией длины ребер:

$$c(u, v) = \begin{cases} 1, & \text{если } (u, v) \in E \\ nf(n), & \text{если } (u, v) \notin E. \end{cases}$$

Из этого определения непосредственно следует, что если в графе G имеется гамильтонов цикл то в графе G' имеется гамильтонов цикл длины n, а если в G нет гамильтонова цикла, то в G' кратчайший путь коммивояжера имеет длину не меньше $nf(n)+n-1$. Отметим, что для любого ребра $(u, v) \in E'$ двоичная запись его длины имеет размер $|c(u, v)| \leq |nf(n)| \leq 1 + \log_2(nc^n) = O(n \log n)$. Поэтому преобразование G в G' можно выполнить за полиномиальное (от n) время.

Если бы для задачи КОММИВОЯЖЕРа существовал полиномиальный аппроксимационный алгоритм A ошибающийся не более чем в $f(n)$ раз, то по его результату для графа G' можно было бы судить о наличии или отсутствии гамильтонова цикла в G. А именно, при ответе $A(G') < nf(n)+n-1$ в G гамильтонов цикл есть, а при $A(G') \geq nf(n)+n-1$ такого цикла нет. Таким образом, мы могли бы решить NP-полную задачу ГАМ_ЦИКЛ за полиномиальное время, из чего следовало бы, что $P = NP$ □

Заметим, что графы, к которым происходит сведение в этой теореме не удовлетворяют неравенству треугольника, которое выполнялось для графов из доказательства теоремы 8.7. Если же ограничиться классом графов $G = (V, E)$ с длинами ребер $c : E \to R^+$, для которых выполнено неравенство треугольника, т.е. для любых трех вершин $u, w, v \in V$ $c(u, v) \leq c(u, w)+ c(w, v)$, то возможны аппроксимации за полиномиальное время, которые дают обходы, которые хуже оптимального в логарифм или даже в константу раз.

Рассмотрим вначале эвристику "ближайшего соседа", которой, по-видимому, пользуется большинство людей при небходимости обойти большое число магазинов. Основанный на этой эвристике алгоритм NN (Nearest Neighbor - ближайший сосед), начав обход с некоторой вершины $v \in V$, далее всякий раз выбирает в качестве следующей вершины обхода ту, из еще не посещенных вершин, которая находится ближе всего к последней посещенной вершине. Таким образом, путь $v = v_1, v_2, \ldots, v_k$ $(k < n)$ продолжается такой вершиной $v_{k+1} \in V \setminus \{v_1, v_2, \ldots, v_k\}$, что $c(v_k, v_{k+1}) = min\{c(v_k, v') \mid v' \in V \setminus \{v_1, v_2, \ldots, v_k\}\}$.

Сложность построения пути коммивояжера этим алгоритмом, очевидно, оценивается как $\Theta(n^2)$. В работе [24] были получены логарифмические оценки качества приближения для этого алгоритма.

Теорема 9.4. *(1) Для любой $n \times n$-матрицы расстояний C, удовлетворяющей неравенству треугольника, выполнено неравенство*

$$NN(C) \leq \frac{1}{2}(\lceil \log_2 n \rceil + 1)OPT(C).$$

(2) Для сколь угодно больших n существуют $n \times n$-матрицы C, удовлетворяющие неравенству треугольника, для которых

$$NN(C) > \frac{1}{3}(\log_2(n+1) + \frac{4}{3})OPT(C).$$

Еще одна "жадная" эвристика для задачи коммивояжера MF, которую иногда называют *мульти-фрагментной*, аналогична эвристике, использованной в построении минимального остовного дерева алгоритмом Крускала. Вначале в искомый гамильтонов путь помещаем самое короткое ребро, а затем на очередном шаге добавляем к уже выбранному подмножеству ребер самое короткое из оставшихся ребер, которое не нарушает гамильтоновости (т.е. не приводит к появлению вершин степени 3). Этот алгоритм, очевидно, можно реализовать за время $\Theta(n^2 \log n)$. А оценки качества получаемых им решений такие же, как и для алгоритма NN, т.е. получаемый результат $MF(C)$ может быть хуже оптимального в $O(\log n)$ раз.

Более точные приближенные алгоритмы были приведены в упомянутой работе Розенкранца, Стирнза и Льюиса [24].

Один из них основан на нижней оценке длины оптимального пути коммивояжера через размер минимального остовного дерева. Действительно, если из любого гамильтонова цикла удалить одно ребро, то получившийся путь будет содержать все вершины графа и, следовательно, будет одним из его остовных деревьев. Обозначим через $MST(C)$ вес минимального остовного дерева графа с матрицей расстояний C. Тогда из вышесказанного следует, что $OPT(C) \geq MST(C)$.

Рассмотрим алгоритм **2МОД**, который использует двойной обход минимального остовного дерева. Он работает следующим образом.

1. По матрице длин ребер C исходного графа с n вершинами построить его минимальное остовное дерево $T = (V, E)$ (зметим, что $|E| = n - 1$).

2. Заменить в нем каждое неориентированное ребро $(u, v) \in E$ двумя противоположно направленными ориентированными ребрами (u, v) и (v, u). Обозначим получившийся ориентированный граф через $T2$.

3. В графе $T2$ полустепени захода и исхода каждой вершины равны 1. Поэтому в нем существует Эйлеров цикл, включающий все ребра. Выбрав произвольную вершину в качестве исходной, построить в $T2$ Эйлеров цикл. Пусть это путь $p_e = v_1, v_2, \ldots, v_{2n-1}(= v_1)$.

4. Превратить p_e в гамильтонов цикл $p_g = u_1, u_2, \ldots, u_{n+1}(= u_1)$ следующим образом:
a) $u_1 = v_1$
b) Пусть для $1 < i < n$ уже определен начальный отрезок пути p_g, включающий $i - 1$ ребро с

вершинами $u_1, \ldots, u_i$ и $u_i = v_j$. Тогда в качестве u_{i+1} выберем следующую еще не пройденную вершину из p_e, т.е. $u_{i+1} = v_k$, где $k = \min\{l \mid (l > j) \wedge \ (v_l \notin \{u_1, \ldots, u_i\}) \ \}$ и продолжим p_g ребром $(u_i, u_{i+1}) = (v_j, v_k)$.
c) Завершим путь p_g ребром из u_n в $u_{n+1} = v_1$.

Теорема 9.5. *Для любой $n \times n$-матрицы расстояний C, удовлетворяющей неравенству треугольника, алгоритм* **2МОД** *строит маршрут коммивояжера не более чем в 2 раза превосходящий по длине оптимальный, т.е.*

$$\textbf{\textit{2МОД}}(C) \leq 2\,OPT(C).$$

Время работы алгоритма **2МОД** *не превосходит $O(n^2 \log n)$.*

Доказательство. Действительно, из неравенства треугольника следует, что длина каждого ребра (u_i, u_{i+1}), помещаемого в путь p_g в пунктах 4(b) и 4(c), не превосходит длину соответствующего участка $v_j, \ldots, v_k$ пути p_e. Поэтому длина всего пути p_g не превосходит длину p_e. Отсюда имеем, $\textbf{2МОД}(C) = c(p_g) \leq c(p_e) = 2\,MST(C) \leq 2\,OPT(C)$.

Для построения минимального остовного дерева можно использовать, например, алгоритм Крускала имеющий сложность $O(n^2 \log n)$ (см. теорему 4.1). Построение Эйлерова цикла (задачи 2.13 и 2.14), его "удвоение" и преобразование в гамильтонов цикл в пунктах 2-4 выполняются за время $O(|E|) = O(n^2)$. $\square$

Заметим, что при использовании для построения минимального остовного дерева наиболее эффективной реализации алгоритма Прима (см. задачу 4.6), время работы алгоритма **2МОД** не превосходит $O(n^2)$.

Кристофидис показал, что, используя минимальные остовные деревья и минимальные паросочетания, можно построить более точное приближение к оптимальному пути коммивояжера. Напомним, что совершенным паросочетанием во взешенном графе $G = (V, E)$ с матрицей расстояний C называется такое подмножество ребер $M \subset E$, что никакие два из них не имеют общих концов и все вершины из V являются концами ребер из M. Минимальным называется совершенное паросочетание наименьшего веса (см. пункт 7.4.1).

Алгоритм **МОД-МП** работает следующим образом.

1. По матрице длин ребер C исходного графа с n вершинами построим его минимальное остовное дерево $T = (V, E)$ (зметим, что $|E| = n - 1$).

2. В дереве $T = (V, E_T)$ выделим подмножество вершин нечетной степени V'[3] Рассмотрим полный подграф G' графа G на вершинах V' и построим для него минимальное совершенное паросочетание M.

3. Пусть мультиграф $T1 = (V, E1)$, где множество ребер $E1$ включает все ребра из E_T и из M (некоторые ребра могут входить в $E1$ дважды). Степени всех вершин в $T1$ четны. Построим в нем Эйлеров цикл p_e.

4. Превратим p_e в гамильтонов цикл p_g так, как это сделано в п. 4 алгоритма **2МОД**.

Теорема 9.6. *Для любой $n \times n$-матрицы расстояний C, удовлетворяющей неравенству треугольника, алгоритм* **МОД-МП** *строит маршрут коммивояжера не более чем в 3/2 раза превосходящий по длине оптимальный, т.е.*

$$\textbf{\textit{МОД-МП}}(C) \leq \frac{3}{2}\,OPT(C).$$

Время работы алгоритма **МОД-МП** *не превосходит $O(n^4)$.*

[3] Заметим, что число вершин $|V'|$ четно.

Доказательство. Покажем вначале, что $c(M) = \sum_{e \in M} c(e) \leq \frac{1}{2} OPT(C)$. Действительно, пусть p_{min} — минимальный гамильтонов цикл. Удалим из него вершины из $V \setminus V'$, превратив таким образом в гамильтонов цикл p' для подграфа G'. Из неравенства треугольника следует, что $c(p') \leq c(p_{min})$. Ребра цикла p' разбиваются на два совершенных паросочетания. По крайней мере вес одного из них не превосходит $\frac{1}{2} c(p')$. Но тогда и вес минимального паросочетания $c(M)$ не превосходит $\frac{1}{2} c(p') \leq \frac{1}{2} c(p_{min}) = \frac{1}{2} OPT(C)$.

Теперь из этого неравенства и из того, что **МОД-МП**$(C) = c(p_g) \leq c(p_e) + c(M)$ и $c(p_e) \leq OPT(C)$ следует утверждение теоремы.

Основной вклад в сложность вносит построение минимального совершенного паросочетания в п. 2. Его можно построить за время $O(|V|^4)$, используя алгоритм, предложенный Эдмондсом и Джонсоном (см. например, [5], глава 12, [10], глава 11)[4] $\square$

Отметим также эвристики локального улучшения маршрута: 2-Opt, 3-Opt, ..., k-Opt. Алгоритм 2-Opt, имея некоторый маршрут коммивояжера, рассматривает всевозможные пары его ребер (a, b) и (c, d) и пытается их перекоммутировать – заменить на пару $(a, d), (b, c)$. Алгоритм 3-Opt аналогичным образом пытается улучшить текущий маршрут, рассмтаривая все тройки ребер, а алгоритм k-Opt – все подмножества из k ребер текущего маршрута. При этом даже число вариантов, рассматриваемых для проверки локальной оптимальности, имеет порядок $O(n^k)$, поэтому на практике используют эти эвристики при $k = 2, 3$.

Приведенные выше, а также и многие другие алгоритмы приближенного поиска маршрута коммивояжера подвергались интенсивной экспериментальной проверке на разных классах графов. В частности, оказалось, что на случайных графах с Эвклидовой метрикой и числом вершин порядка 10^5 алгоритм NN ошибается не более, чем на 25%, алгоритм MF – не более, чем на 15%, алгоритм Кристофидеса дает точность порядка 10%, алгоритм 2-Opt – 5%, алгоритм 3-Opt – 3% (локальные алгоритмы улучшали результаты алгоритма MF). При этом для достижения указанных результатов 2-Opt и 3-Opt для больших графов он тратят в 2-3 раза больше времени, чем более простые эвристики NN и MF, а алгоритм Кристофидеса даже в сотни раз.

9.3 Метод "ветвей и границ" для задачи коммивояжера

Рассмотрим на примере задачи коммивояжера один из методов "интеллектуального" перебора – метод "ветвей и границ". Алгоритмы этого типа применяются для поиска оптимальных решений оптимизационных задач. В худшем случае они производят полный перебор вариантов для поиска наилучшего из них и поэтому требуют экспоненциального (от размера входа) времени. Но во многих случаях число рассматриваемых вариантов удается существенно сократить.

Предположим, что для некоторой задачи A требуется из большого множества возможных решений SOL выбрать решение s с минимальным весом (длиной, стоимостью) $c(s)$. Предположим также, что SOL естественным образом представляется в виде дерева T, каждой ветви которого соответствует некоторое решение. При этом каждой вершине v дерева T соответствует некоторая подзадача, решение которой определяется путем из корня в v, и имеется легко вычисляемая функция $B(v)$, которая задает нижнюю границу качества решений s, проходящих через v, т.е. $c(s) \geq B(v)$. В переменной ТЕК-МИН будет храниться минимальный вес уже рассмотренных решений. Алгоритм осуществляет обход дерева T в глубину. При этом при достижении листа и построении некоторого "полного" решения s определяется его вес $c(s)$, который в случае $c(s) <$ ТЕК-МИН становится новым значением минимума. Перед переходом в новую вершину u вычисляется $B(u)$ и, если $B(u) \geq$ТЕК-МИН, то переход в u (и, следовательно, просмотр решений из поддерева с корнем u) не осуществляется. Ясно, что при этом минимальное решение потеряно не будет.

[4]При более эффективной реализации построение минимального паросочетания можно выполнить за время $O(|V|^4)$ ((см.[10]) .

Пусть $G = (V, E)$ — неориентированный граф с n вершинами и длинами ребер c. Зафиксируем произвольную вершину $a \in V$. В качестве $T = (V_T, E_T)$ рассмотрим дерево всех простых путей, начинающихся в a. Его вершины помечены вершинами из V. Корень имеет метку a. Пусть в вершину $w \in V_T$ с меткой $v \in V$ ведет из корня путь, метки вершин которого образуют подмножество вершин $V_w \subset V$. Тогда w соединена исходящими ребрами с вершинами T, помеченными $\{u \mid (v, u) \in E\} \cap (V \setminus V_w)$. Каждая ветвь дерева T длины n, соответствующая гамильтонову пути $s = (a = v_1, v_2, \ldots, v_n)$ в G, при наличии ребра $(v_n, a) \in E$ задает возможный маршрут коммивояжера веса $c(s) = \sum c_{i=1}^{i=n}(v_i, v_{i+1(modn)})$. Более короткие ветви T — соответствуют простым путям G, которые нельзя продолжить до гамильтоновых циклов. Имеются разные способы определения нижней оценки $B(w)$ веса маршрутов коммивояжера, проходящих через вершину $w \in V_T$. Мы рассмотрим один из самых простых. Пусть в w из корня ведет путь, соответствующий пути $s = (a = v_1, v_2, \ldots, v_k = b), k < n$, в G. Тогда $V_w = \{v_1, \ldots, v_k\}$ и всякий гамильтонов путь s', продолжающий s будет включать все вершины из множества $V' = V \setminus V_w$. Вес участка этого пути, проходящего по V', не меньше веса $MST(G')$ минимального остовного дерева подграфа G' графа G, включающего все вершины из V' и соединяющие их ребра из E. Поэтому вес любого маршрута коммивояжера, проходящего через w, будет не меньше величины

$$B(w) = B(s) = c(s) + MST(G') + \min\{c(b, u) \mid u \in V'\} + \min\{c(u, a) \mid u \in V'\}.$$

Заметим, что на самом деле оценка $B(w)$ однозначно определяется путем s в графе G. Поэтому в приведенном ниже алгоритме вершины w дерева T в явном виде не участвуют, а перебор ведется по путям s в графе G.

Теперь метод "ветвей и границ" для задачи коммивояжера можно уточнить следующим образом.

Алгоритм КОМ-ВГ

Вход: $G = (V, E)$ — (не)ориентированный граф, представленный списками смежности вида $L_v = (w_1, c(v, w_1)), \ldots, (w_{k(v)}, c(v, w_{k(v)})$, где $w_1, \ldots, w_{k(v)}$ — это все вершины, в которые из v ведут ребра, а $c(v, w_i)$ — длина ребра (v, w_i), и исходная вершина $a \in V$.

1. Быстро построить некоторый маршрут коммивояжера s, начинающийся в a;
2. ОПТ-ПУТЬ := s;
3. ТЕК-МИН := $c(s)$;
4. $v := a$; $s := \varepsilon$;
5. ОПТ_ПОИСК(v,s)

Procedure ОПТ_ПОИСК(v,s)
Вход: v – текущая вершина дерева, s путь из корня в v.

6. $W := \{w \mid w$ *входит в путь* $s\}$;
7. **IF** $W = V$ % s – гамильтонов путь
8. **THEN**
9. { $c := c(s) + c(v, a)$; % c – вес текущего маршрута
10. **IF** $c <$ ТЕК-МИН
11. **THEN** { ТЕК-МИН := c; ОПТ-ПУТЬ := s} % новый оптимум
12. }
13. **ELSE**
14. { $L := \{u \mid (v, u) \in E\} \setminus W$;
15. **FOR EACH** $u \in L$ **DO**

16. { $s := s, (v, u)$;
17. $V' := V \setminus (W \cup \{u\})$;
18. $G' :=$ *подграф* G, *порожденный* V';
19. $B(s) := c(s) + MST(G') + \min\{c(b, u) \mid u \in V'\} + \min\{c(u, a) \mid u \in V'\}$;
20. **IF** $B(s) <$ТЕК-МИН
21. **THEN** ОПТ_ПОИСК(u,s)
22. } }.

Заметим, что в качестве исходного маршрута в стр. 1 можно взять произвольную перестановку вершин из V. Но практическая эффективность алгоритма значительно улучшится, если использовать для поиска исходного пути один из быстрых эвристических алгоритмов, рассмотренных в разделе 8.5. Например, алгоритм "ближайшего соседа" NN за время $O(n^2)$ строит маршрут, который хуже оптимального не более чем в $\log n$ раз.

Теорема 9.7. *Алгоритм КОМ-ВГ всегда находит оптимальное решение задачи коммивояжера.*

Что касается сложности этого алгоритма, то нерекурсивную часть вызова процедуры ОПТ_ПОИСК(v,s) можно выполнить за время $O(|E| \log |E|)$ (основной вклад вносит время построения минимального остовного дерева при вычислении $B(s)$ в стр. 19). Число вызовов этой процедуры уменьшается за счет ранней проверки условия в стр. 20. Но это в общем случае не гарантирует полиномиальности алгоритма.

Пример 25. *Продемонстрируем применение алгоритма КОМ-ВГ к поиску оптимального маршрута коммивояжера на графе G, показанном на рис. 43.*

(

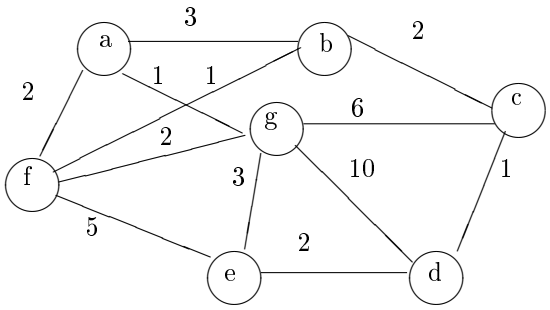


Рис. 43: Граф G

На рис. 44 представлено дерево путей графа G, которое строит в процессе работы алгоритм КОМ-ВГ. Рядом с каждой вершиной w указано значение нижней оценки $B(w)$ ля этой вершины. Мы считаем, что вершины в списках смежности упорядочены лексикографически. Пусть исходной вершиной является a, а в качестве исходного маршрута в стр. 1 выбран обход в алфавитном порядке: $s_0 = a, b, c, d, e, f, g, a$ (самая левая ветвь дерева). Его вес $c(s_0) = 16$ становится начальным значением ТЕК-МИН. Затем алгоритм обнаруживает путь $s_1 = a, b, c, d, e, g, f, a$ с меньшим весом $c(s_1) = 15$, который становится следующим значением ТЕК-МИН. Благодаря этой границе обрываются пути a, b, f, e и a, b, f, g (этот путь невозможно продолжить до гамильтонова пути). После этого находится путь $s_2 = a, f, b, c, d, e, g, a$ с весом $c(s_2) = 12$. Благодаря этому значению ТЕК-МИН рассмотрение всех остальных путей обрывается на достаточно ранней стадии. Всего в процессе работы алгоритм КОМ-ВГ рассматривает 10 путей при их общем числе в дереве, равном $6! = 720$.

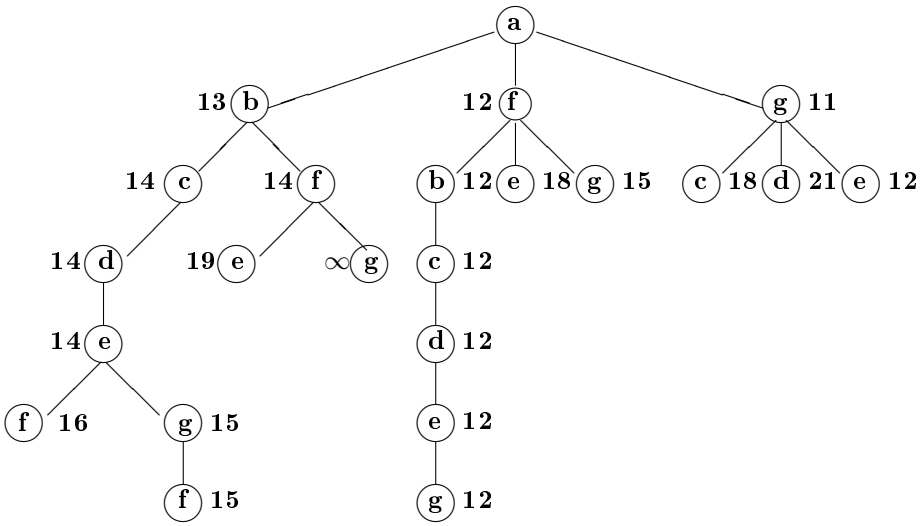


Рис. 44: Дерево обхода алгоритма КОМ-ВГ

9.4 Задачи

Задача 9.1. *Предложите алгоритм линейной сложности для нахождения максимального независимого множества вершин в неориентированном дереве.*

Задача 9.2. *Постройте "жадный"алгоритм, который находит оптимальное вершинное покрытие для графа, являющегося деревом, за линейное время.*

Задача 9.3. *Предложите алгоритм полиномиальной сложности для нахождения минимального ВЕРШИННОГО ПОКРЫТИЯ в двудольных (бихроматических) графах.*
Указание. *Установите связь между этой задачей и задачей о максимальных паросочетаниях в двудольных графах (см. раздел 7.4). Рссмотрим следующий аппроксимационный алгоритм для построения вершинного покрытия.*

1. Используя алгоритм поиска в глубину **ПОГ** *построить глубинный лес $T_G = (V, T)$ графа $G = (V, E)$.*

2. Поместить в вершинное покрытие C все внутренние вершины из T_G.

Докажите, что построенное этим алгоритмом множество C является вершинным покрытием для G и превосходит по мощности минимальное вершинное покрытие не более чем в 2 раза.

Задача 9.4. *Рассмотрим следующую задачу* **k-центры** *: для заданного нагруженного неориентированного графа $G = (V, E)$ с длинами ребер $c : E \to R$ найти подмножество $S \subseteq V$ вершин размера $|S| = k$ такое, что каждая вершина из V близка к некоторой вершине из S, т.е. S минимизирует функцию $\max_{v \in V} \min_{u \in S} c(u, v)$.*

Рассмотрите для этой задачи жадный алгоритм, который, начав с произвольной вершины, последовательно добавляет к S наиболее далекие от S вершины. Докажите, что получаемое таким образом решение отличается от оптимального не более, чем в 2 раза.

Задача 9.5. *Для нагруженного графа G, представленного матрицей C, определите гамильтонов путь p_{NN}, используя алгоритм NN (ближайшего соседа) и гамильтонов путь p_{2MOD},*

используя алгоритм 2MOD (с двойным обходом минимального остовного дерева). Сравните полученные результаты с длиной 12 минимального пути.

$$C = \begin{pmatrix} 0 & 3 & 10 & 10 & 10 & 2 & 1 \\ 3 & 0 & 2 & 12 & 12 & 2 & 10 \\ 10 & 2 & 0 & 1 & 10 & 12 & 6 \\ 10 & 12 & 1 & 0 & 2 & 12 & 10 \\ 10 & 12 & 10 & 2 & 0 & 5 & 3 \\ 2 & 1 & 12 & 12 & 5 & 0 & 2 \\ 1 & 10 & 60 & 10 & 3 & 2 & 0 \end{pmatrix}$$

Задача 9.6. *Задача* **k-поставщиков** *аналогична предыдущей задаче. В ней дополнительно задано разбиение вершин графа $G = (V, E); c : E \to, R$ на множество* поставщиков $F \subseteq V$ *и множество* потребителей $D = V \setminus F$. *Требуется найти подмножество поставщиков $S \subseteq F, |S| = k$, котолрое S минимизирует функцию $\max_{v \in D} \min_{u \in S} c(v, u)$.*

Предложите для этой задачи аппроксимационный полиномиальный алгоритм, который строит решение, отличающееся от оптимального не более, чем в 3 раза.

Задача 9.7. *Задача* **k-МНОЖЕСТВЕННОЕ СЕЧЕНИЕ** *состоит в том, чтобы по неориентированному графу $G = (V, E)$ и множеству k терминальных вершин $v_1, v_2, \ldots, v_k \in V$ найти минимальное множество ребер из E, удаление которых оставит все терминальные вершины в разных связных компонентах.*

(а) Покажите, что эта задача точно решается за полиномиальное время для $k = 2$.

(б) Для случая $k = 3$ предложите аппроксимационный полиномиальный алгоритм, который строит решение, отличающееся от оптимального не более, чем в 2 раза.

(в) Разработайте алгоритм локального поиска множественного сечения для общего случая.

Задача 9.8. *Рассмотрим следующую задачу* **МАКСИМАЛЬНОЕ СЕЧЕНИЕ** *: для заданного нагруженного неориентированного графа $G = (V, E)$ с длинами ребер $c : E \to R$ найти подмножество $S \subseteq V$ вершин такое, что сумма длин ребер между S и $V \setminus S$ максимальна. Если через $w(S)$ обозначить сумму длин ребер, один конец которых принадлежит S, а другой – $V \setminus S$, то* **МАКСИМАЛЬНОЕ СЕЧЕНИЕ** *это задача о максимизации $w(S)$ по всем подмножествам $S \subseteq V$. $S \subseteq V$.*

Рассмотрите для этой задачи локальный алгоритм, который, начав с произвольной вершины, последовательно ищет и добавляет к S такую вершину v , что $w(S \cup \{v\}) > w(S)$.

(а) Докажите,что получаемое таким образом решение отличается от оптимального не более, чем в 2 раза.

(б) Оцените время работы этого алгоритма.

10 Применение графов в анализе социальных сетй

10.1 Социальные сети

Социальные сети начали изучаться в социологии в 50-е годы XX-го века. В учебниках социологии *социальную сеть* определяют как совокупность сетевых акторов (точек, вершин, агентов), вступающих во взаимодействие друг с другом и связи между которыми преимущественно социальные, такие как дружественные отношения, совместная работа, обмен информацией и т.п.

Таким образом, социальная сеть это **граф**, вершины которого акторы, а ребра представляют связи, взаимодейсвия отношения между различными акторами.

Среди них могут быть связи, отражающие подобие акторов, их социальные отношения, взаимодействия, местоположение, атрибуты, родственные связи, роли, эмоциии, когнитивные отношения и т.п. В некоторых социальных сетях представлены симметричные отношения, например, *Быть другом кого-то*, *Работать над одним проектом с кем-то* и др. Такие сети представляются посредством неориентированных графов. Вдругих сетях представлены несимметричные отношения, например *Посылать электронное письмо кому-то*, *Помогать кому-то*, *Участвовать в некотором событии* и др. Сети с такими отношениями представляются ориентированными графами. Иногда естественно рассматривать графы со взвешенными ребрами. Числовая метка ребра может характеризовать вес, интенсивность, силу соответствующего отношения.

Размеры социальных сетей варьируются от нескольких акторов (в сетях отношений одной семьи, сотрудников малого предприятия и т.п.) до миллионов и десятков миллионов акторов в интернет-сетях таких как *Facebook, MySpace, Одноклассники, В контакте* и др. Да и вся паутина Web-страниц может рассматриваться (и на самом деле рассматривается поисковыми системами) как одна социальная сеть с отношением "ссылается на".

Главные задачи анализа социальных сетей состоят в определении их теоретико графовых свойств, которые характеризуют
а) структуру сети (анализ на уровне сети);
б) положение в сети (анализ на уровне вершин);
в) попарные свойства (анализ на уровне пар).

Анализ на уровне сети связан с двумя видами свойств: связанностью и формой. Связанность характеризуется такими свойствами как плотность, длина путей, фрагментация. Меры связанности позволяют также выделять в сетях подгруппы (“сообщества”,“клики”) – области, обладающие таким специфическими свойствами связанности как высокая плотность, небольшая длина путей между акторами и др. Форма сети связана с распределением ее соединений (ребер). Сюда относятся такие свойства как ядерность / переферийность, массивность и т.п. Анализ на уровне вершин занимается свойствами центральности вершин, связанными с важностью вершин, их доминирующим положением в сети. Например, одним из свойств центальности является *промежуточность по Фримену (Freeman)*, которая отражает свойство вершины лежать на на кратчайших путях, соединяющих две другие вершины. Это можно интерпретировать как потенциальную силу актора, который может замедлить идущие сквозь него потоки или исказить их в свою пользу. Анализ на уровне пар, как правило, связан с двумя видами свойств: парной связанностью и эквивалентностью. Связанность пары вершин означает как их близость в сети, так и наличие многих типов свзей между ними. Эквивалентность оценивает степень, с которой вершины пары играют в структуре сети одинаковые роли, например, изоморфность их окрестностей.

Мы будем рассматривать социальные сети как ориентированные или неориентированные графы в зависимости от видов представляемых ими связей, взаимодействий или отношений ([25, 26].

Ребра в графах социальных сетей называются связями (links) или соединениями (ties).

Обозначение. Пусть $I = \{i_1, \dots i_n\}$ – это некоторое множество акторов. Социальная сеть – это граф без петель $G = \langle I, E \rangle$, где E – множество (ориентированных или неориентированных) связей между парами акторов. Для конкретных сетей вершины и ребра представлящих их графов могут иметь специальные метки, содержащие дополнительную информацию. Например, имена вершин, типы и интенсивность связей, веса ребер и т.п.

Введем следующие обозначения для сети $G =< I, E >$ и актора $i \in I$ пусть:
$d(i) = |\{j \ |(i,j) \in E\}|$ обозначает степень i в неориентированном графе сети G;
$d_i(i) = |\{j \ |(j,i) \in E\}|$ обозначает (полу)степень захода в i в G (для ориентированного G);
$d_o(i) = |\{j \ |(i,j) \in E\}|$ обозначает (полу)степень исхода из i в G (для ориентированного G).

Очевидно, максимальная степень актора в сети G с n акторами равна $n-1$.

Для акторов i и j обозначим через $\rho(i,j)$ длину кратчайшего пути из i to j. Как обычно, для ненагруженных графов длина пути равна числу ребер на этом пути, а для нагруженных – сумме длин (весов, стоимостей и т.п.) ребер этого пути.

Важную роль в анализе социальных сетей играют следующие формализации в терминах теории графов понятий, содержательно описанных выше.

10.2 Параметры центральности акторов

Центральный актор – это актор, у которого много связей с другими акторами. Такие акторы играют важную роль в анализе социальных сетей. Имеется несколько формализаций центральности в терминах графов.

Степень центральности оценивает относительную степень вершины актора в сети. Для *неориентированного графа* G степень центральности актора i, обозначаемая $C_D(i)$, определяется как

$$C_D(i) = \frac{d(i)}{n-1}.$$

Для *ориентированного графа* G степень центральности актора i, также обозначаемая $C_D(i)$, определяется как

$$C_D(i) = \frac{d_o(i)}{n-1}.$$

(т.е., учитываются только выходящие из i связи).

10.2.1 Близкая центральность (closeness centrality (Sabidussi, 1966))

Эта величина оценивает насколько близки к данному актору остальные акторы сети. Как для ориентированных, так и для неориентированных сетей близкая центральность актора i обозначается $C_C(i)$ и определяется как:

$$C_C(i) = \frac{1}{\sum_{j \in I} \rho(i,j)}.$$

Для ненагруженных графов эту величину обычно нормализуют следующим образом:

$$C_C(i) = \frac{n-1}{\sum_{j \in I} \rho(i,j)}.$$

В этом случае $0 < C_C(i) \leq 1$, так как минимальное расстояние $\rho(i,j)$ для всех $j \neq i$ равно 1, а минимальное значение суммы $\sum_{j \in I} \rho(i,j)$ равно $n-1$.

Чтобы избежать бесконечности в знаменателе из i должны быть достижимы все вершины. Поэтому для определения близкой центральности для всех акторов неориентированный граф должен быть связным, а ориентированный – сильно связным.

10.2.2 Срединная центральность (Betweenness Centrality)

Срединная центральность (Freeman, 1977; Anthonisse, 1971)Срединная центральность определяет важность актора, учитывая, как часто он появляется на путях взаимодействия других акторов. Если актор i появляется на всех или почти всех путях, связывающих акторы j и k, то он может "контролировать" их взаимодействие. Срединная центральность позволяет выделять акторы, без прохождения через которые нельзя связать многие пары других акторов сети.

Неориентированные графы. Для акторов j и k пусть p_{jk} обозначает число кратчайших путей между j и k (будем считать, что $p_{jj} = 1$). Пусть i ($i \neq j$ и $i \neq k$) – это третий актор. Обозначим через $p_{jk}(i)$ число кратчайших путей между j и k, проходящих через i.

Тогда срединная центральность актора i, обозначаемая $C_B(i)$, определяется как

$$C_B(i) = \sum_{j \in I, j \neq i} \sum_{k \in I, k \neq j, k \neq i} \frac{p_{jk}(i)}{p_{jk}}.$$

Для ненагруженных графов $C_B(i)$ может принимать значения от 0 до $\frac{(n-1)(n-2)}{2}$ (это число пар акторов, не содержащих i). Поэтому $C_B(i)$ иногда нормализуют следующим образом:

$$C_B(i) = \frac{2 \sum_{j \in I, j \neq i} \sum_{k \in I, k \neq j, k \neq i} \frac{p_{jk}(i)}{p_{jk}}}{(n-1)(n-2)}.$$

Ориентированные ненагруженные графы. В этом случае (нормализовнную) срединную центральность определяют как

$$C_B(i) = \frac{\sum_{j \in I, j \neq i} \sum_{k \in I, k \neq j, k \neq i} \frac{p_{jk}(i)}{p_{jk}}}{(n-1)(n-2)}.$$

(для ориентированного графа (x, y) и (y, x) это разные ребра).

10.2.3 Алгоритмы вычисления центральности

Мы будем рассматривать вычисление центральности вершин для неориентированных графов, в которых веса (длины) всех ребер равны 1, и для нагруженных ориентированных графов $G = (I, E)$, в которых веса (длины) ребер $c(e), e \in E$, являются неотрицательными числами.

Пусть $\delta_{jk}(i) = \frac{p_{jk}(i)}{p_{jk}}$. Тогда срединная центральность определяется соотношением

$$C_B(i) = \sum_{j \neq i \neq k} \delta_{jk}(i).$$

Заметим, что для вычисления величин $p_{jk}(i)$ и $\delta_{jk}(i)$ достаточно уметь вычислять длины кратчайших путей и величины p_{jk}, так как

$$p_{jk}(i) = \begin{cases} p_{ji} p_{ik} & \text{если } \rho(j,k) = \rho(j,i) + \rho(i,k) \\ 0 & \text{в противном случае} \end{cases}$$

Поэтому срединная центральность может быть определена следующим способом.

1. *Вычислить длины и количество кратчайших путей между всеми парами вершин.*
2. *Для каждой вершины i определить и просуммировать все значения $\delta_{jk}(i)$.*

Для определения длин кратчайших путей можно использовать алгоритм Уоршолла-Флойда из пункта 6.1.1, а для определения числа таких путей утверждение о том, что элемент $a_{ij}^{(k)}$ l-ой степени A_G^l матрицы смежности A_G равен числу различных путей длины l из i в j (см. задачу 6.2).

Другой способ определения числа p_{jk} кратчайших путей связан с использованием множества вершин $P_j(k)$ – *предшественников* k на кратчайших путях из j:

$P_j(k) = \{i \mid (i,k) \in E,\ \rho(j,k) = \rho(j,i) + c(i,k)\}$.

Так как длины всех ребер неотрицательны, то все кратчайшие пути из j в k заканчиваются ребрами вида (i,k) для $i \in P_j(k)$. Отсюда следует справедливость следующего соотношения.

Лемма 10.1.

$$p_{jk} = \sum_{i \in P_j(k)} p_{ji}.$$

Для вычисления p_{jk} в соответствии с этой формулой можно адаптировать алгоритм поиска в ширину для ненагруженных графов или алгоритм Дейкстры для нагруженных ориентированных графов. Используя известные оценки сложности этих алгоритмов, получаем

Предложение 10.1. *Для заданной вершины $j \in I$ можно определить длины и число кратчайших путей из нее до остальных вершин за время:*
а) $O(|E|)$ – для ненагруженных графов;
б) $O(|E| + |I| \log |I|)$ – для нагруженных ориентированных графов.

Отсюда следует, что величины p_{jk} можно вычислить за время $O(|I| \cdot |E|)$ для ненагруженных графов и за время $O(|I| \cdot |E| + |I|^2 \log |I|)$ для нагруженных ориентированных графов.

Поскольку для вычисления срединной центральности i в п.2 требуется суммировать значения $\delta_{jk}(i)$, то общее время вычисления будет $O(|I|^3)$. Для этого вычисления требуется память размера $O(|I|^2)$, необходимая для хранения матрицы расстояний и величин p_{jk}.

В работе [15] Брандес (Brandes) предложил улучшение этого алгоритма, в котором суммирование по всем парам вершин заменяется на последовательное вычисление частичных сумм. Следуя этой работе, определим *зависимость* вершины j от вершины i как величину

$$\delta_{j\bullet}(i) = \sum_{k \in I} \delta_{jk}(i).$$

Зависимости можно вычислять, используя рекурсивные соотношения.

Лемма 10.2. *Если в каждую вершину $k \in I$ ведет единственный кратчайший путь из j, то*

$$\delta_{j\bullet}(i) = \sum_{i \in P_j(k)} (1 + \delta_{j\bullet}(k)).$$

Доказательство этой леммы предоставляется читателю (см. задачу 10.1).

В общем случае рекурсия более сложная.

Теорема 10.1. *Для любых вершин j и i справедливо соотношение*

$$\delta_{j\bullet}(i) = \sum_{i \in P_j(k)} \frac{p_{ji}}{p_{jk}} (1 + \delta_{j\bullet}(k)).$$

Доказательство. Уточним понятие зависимости, включив в него ребро, выходящее из i. Пусть $\delta_{jk}(i,(i,l)) = \frac{p_{jk}(i,(i,l))}{p_{jk}}$, где $p_{jk}(i,(i,l))$ – это число кратчайших путей из j в k, содержащих как вершину i, так и ребро (i,l). Тогда

$$\delta_{j\bullet}(i) = \sum_{k \in I} \delta_{jk}(i) = \sum_{k \in I} \sum_{l : i \in P_j(l)} \delta_{jk}(i,(i,l)) = \sum_{l : i \in P_j(l)} \sum_{k \in I} \delta_{jk}(i,(i,l)).$$

Как мы уже отмечали, для любых k и l $p_{jk}(l) = p_{jl}p_{lk}$. Число кратчайших путей из j в k, содержащих ребро (i,l), очевидно, равно произведению числа кратчайших путей из j в i на число кратчайших путей из l в k. Тогда $p_{jk}(i,(i,l)) = p_{ji}p_{lk} = \frac{p_{ji}}{p_{jl}} \cdot p_{jk}(l)$. Отсюда выводим, что

$$\delta_{jk}(i,(i,l)) = \begin{cases} \frac{p_{ji}}{p_{il}} & \text{если } k = l \\ \frac{p_{ji}}{p_{il}} \cdot \frac{p_{jk}(l)}{p_{jk}} & \text{если } k \neq l \end{cases}$$

Подставив это в выражение для $\delta_{j\bullet}(i)$, получим утверждение теоремы:

$$\sum_{l:i\in P_j(l)} \sum_{k\in I} \delta_{jk}(i,(i,l)) = \sum_{l:i\in P_j(l)} (\frac{p_{ji}}{p_{il}} + \sum_{k\neq l} \frac{p_{ji}}{p_{il}} \cdot \frac{p_{jk}(l)}{p_{jk}}) = \sum_{l:i\in P_j(l)} \frac{p_{ji}}{p_{il}} \cdot (1 + \delta_{j\bullet}(k)).$$

Следствие 10.1.1. *По заданному ациклическому графу кратчайших путей из вершины j в графе G все зависимости j от остальных вершин можно вычислить за время $O(|E|)$ и с памятью $O(|I| + |E|)$.*

Доказательство. Для вычисления требуемых зависимостей достаточно обойти вершины графа кратчайших путей в порядке неубывания их расстояний от j и суммировать зависимости в соответствии с теоремой 10.1. Для этого достаточно для каждой вершины хранить зависимость j от нее и список ее предшественников в графе кратчайших путей. Общий объем этих списков пропорционален числу ребер в графе.

Теорема 10.2. *Срединная центральность может быть вычислена за время $O(|I| \cdot |E| + |I|^2 \log |I|)$ и с памятью $O(|I| + |E|)$ для нагруженных графов. Для ненагруженных графов время вычисления сокращается до $O(|I| \cdot |E|)$.*

Доказательство. Для каждой вершины-источника $j \in I$ построим граф кратчайших путей из нее. В конце каждой итерации зависимости вершины-источника от других вершин добавляются к значениям срединной центральности этой вершины. Ниже этот алгоритм детализирован для ненагруженных графов, заданных с помошью списков смежности $L_i, i \in I$. Как и в алгоритме поиска в ширину в нем используется очередь пройденных вершин Q. В стек S вершины помещаются по мере увеличения расстояния от исходной вершины j. По ходу вычисления для каждой вершины q определяется число $p[q]$ кратчайших путей из j в q и список $P[q]$ предшественников q на этих путях.

Алгоритм BetweenCentrality(G)

1. **FOR EACH** $i \in I$ **DO** $C_B[i] := 0$;
2. **FOR EACH** $j \in I$ **DO**
3. { создать пустой стек S;
4. **FOR EACH** $i \in I$ **DO** $P[i] := \emptyset$;
5. $p[j] := 1$; $d[i] := 0$;
6. **FOR EACH** $k \in I$, $k \neq j$ **DO**
7. $\{p[k] := 0$; $d[k] := -1$; % k – "новая"
8. создать пустую очередь Q;
9. ДОБАВИТЬ(Q, j);
10. **WHILE** $Q \neq \emptyset$ **DO**
11. { $r :=$ НАЧАЛО(Q); УДАЛИТЬ(Q, r);
12. ВТОЛК(S, r);
13. **FOR EACH** $q \in L_r$ **DO**

```
14.          { IF  d(q) < 0;  % вершина v_q "новая"
15.          THEN
16.              { ДОБАВИТЬ(Q, q);
17.              d(q) := d(q) + 1;   %пометить v_q как "старую"
18.              };
                 % кратчайшие пути в q через r
19.          IF  d(q) = d(r) + 1 THEN
20.              { p[q] := p[q] + p[r];
21.              P[q] := P[q] ∪ {r}
22.          }
23.      }
24.  };
25.  FOR EACH i ∈ I DO δ[i] := 0;
                 % в S вершины упорядочены по невозрастанию расстояния от v_j
26.  WHILE S ≠ ∅ DO
27.      {  k := ВЫТОЛК(S);
28.      FOR EACH  i ∈ P[k] DO
28.          δ[i] := δ[i] + p[i]/p[k] · (1 + δ[k]);
29.      IF  k ≠ j THEN C_B[k] := C_B[k] + δ[k]
30.      }
31.  }
```

Схему этого алгоритма, связанную, с построением и использованием графов кратчайших путей, можно применить также для вычисления других мер центральности.

10.3 Престижность

Содержательно, некоторый актор имеет высокий престиж, если на него направлены связи от многих других акторов. Основное отличие престижности от центральности заключается в том, что в определении центральности участвуют *выходящие из вершины* ребра и расстояния от данной вершины до других, а престижность зависит от *входящих в вершину* ребер и расстояниях от других вершин до нее.

Престижность определяется только для ориентированных графов. Степень престижности $P_D(i)$ актора i в социальной сети определяется как: $P_D(i) = \frac{d_i(i)}{n-1}$.

Эта величина принимает значения от 0 до 1. $P_D(i) = 1$ означает, что все акторы сети непосредственно связаны с i.

10.3.1 Средняя престижность (Proximity Prestige)

Эта мера обобщает степень престижности, т.к. учитывает не только соседей актора i, но и всех других акторов, из которых есть пути в i. .

Область влияния актора i, обозначаемая I_i, это множество акторов, из которых можно достичь i в социальной сети G. Среднее расстояние от акторов из I_i до i определяется как

$$dm(I_i, i) = \frac{\sum_{j \in I_i} d(j, i)}{|I_i|}.$$

Средняя престижность актора i, обозначаемая $P_P(i)$, определяется как

$$P_P(i) = \frac{|I_i|}{(n-1)dm(I_i, i)} = \frac{|I_i|^2}{(n-1)\sum_{j \in I_i} d(j, i)}.$$

Здесь, $\frac{|I_i|}{n-1}$ это доля акторов из I, которые могут достичь i.

$P_P(i)$ находится в интервале от 0 до 1.

10.3.2 Ранжированный престиж (Rank Prestige)

Во всех предыдущих мерах считалось, что на престиж данного актора одинаково влияют все ссылающиеся на него акторы. В реальных сетях ссылки более авторитетных акторов должны иметь больший вес, чем менее авторитетных.

Ранжированный престиж позволяет оценить престиж актора через престиж его соседей. **Ранжированный престиж** $P_R(i)$ актора i определяется как

$$P_R(i) = \sum_{j \in I} A_{ij} \cdot P_R(j).$$

здесь A_{ij} – это элементы матрицы смежности графа G: $A_{ij} = 1$, если $(i, j) \in E$, и $A_{ij} = 0$ иначе.

Заметим, что значение $P_R(i)$ определено **рекурсивно**.

Пусть $\mathbf{P} = (P_R(i_1), \ldots, P_R(i_n))$ – вектор-столбец ранжированных престижей всех акторов. Тогда

$$\mathbf{P} = A^T \mathbf{P}.$$

Решениями этой системы уравнений являются **собственные векторы P** матрицы A^T.

Ранжированный престиж используется в алгоритме **PageRank**, который будет рассмотрен ниже.

10.4 Ранжирование интернет-страниц. Алгоритм PageRank

Интернет можно рассматривать как очень большую социальную сеть. *Хороший метод поиска в интернете* должен соединять поиск страниц, которые содержат все или почти все термины из запроса с **устойчивым ранжированием**, которое *передвигает важные высококачественные и надежные страницы* к началу списка ответов. Выше мы определили **престижность** как меру **важности интернет-страниц**.

PageRank. PageRank это процедура вычисления **престижа** каждой страницы в связанном множестве страниц. Она была впервые предложена основателями компании Google Л. Пейджем (L. Page) и С. Брином (S. Brin) в 1998 г. [23]. Алгоритм PageRank основан на двух содержательных соображениях:

1) ссылка на страницу является признаком престижности этой страницы;

2) чем более престижные страницы ссылаются на данную страницу, тем выше ее собственный престиж.

Существует достаточно много определений и вариантов вычисления PageRank-рейтингов страниц. Здесь мы приведем самое простое определение.

Интернет как граф. Мы рассматриваем World Wide Web как граф, вершинами которого являются **отдельные web-страницы** (urls) , а **гипертекстовые ссылки** между страницами играют роль ребер.

Более формально,рассмотрим ориентированный граф $G_{WWW} = (V, E)$. множество V его вершин — это список web-страниц (urls). Ребро $(v, w) \in E$ означает, что в теле страницы v имеется *тег* `<a href="URL>`, где `URL` это URL страницы w.

Для страницы $i \in V$ через $I(i)$ обозначим множество всех входящих в нее ребер, т.е. таких $e \in E$, что $e = (v, i)$ для некоторой $v \in V$, а через $O(i)$ – множество всех выходящих из i ребер, т.е. таких $e \in E$, что $e = (i, v)$ для некоторой $v \in V$.

Замечание: часто в I(i) и O(i) учитываются только входящие и выходящие ребра из страниц на других сайтах.

Интернет-сёрфинг. PageRank моделирует поведение пользователя интернет, в одном окне браузера. В частности, PageRank моделирует следующие действия:

Обход PageRank:

1. Пользователь начинает обход web-страниц с некоторой *случайно выбранной страницы* из V[a].
2. На каждом шаге пользователь обозревает некоторую страницу i. *С вероятностью* $d \in (0,1)$ он выбирает переход по ссылкам, расположенным на этой странице (в предположении, что хоть одна такая ссылка имеется).
3. Любая из ссылок на странице i *может быть выбрана с равной вероятностью.*
4. *С вероятностью* $1-d$ пользователь, устав от последовательного обхода страниц, перепрыгивает сразу на *случайно выбранную страницу из* V.
5. Если из текущей страницы нет ссылок на другие страницы, то пользователь просто переходит на *случайно выбранную страницу из* V.

[a]на самом деле, PageRank позволяет ослабить это условие и стартовать с некоторой странцы случайно выбранной из некоторого небольшого множества страниц.

Что вычисляет PageRank. Ранг, который **PageRank** присваивает странице $i \in V$ это *вероятность достичь в конце концов эту страницу* в описанном выше процессе обхода [23].

Вывод PageRank

Пусть $p(i)$ это *вероятность достичь web-страницу* i (т.е., ранг PageRank для страницы i). Пусть $I(i) = \{j_1, \dots j_s\}$ – множество всех страниц, которые ссылаются **на** i. Пусть $p(j_1), \dots, p(j_s)$ это вероятности достижения каждой из этих страниц. Через $O(j_k)$ обозначим множество *всех ребер, выходящих из* j_k.

Предположение: Из каждой страницы в V выходит хоть одно ребро.

- Предположим, что мы достигли страницу j_1. С вероятностью d мы выберем движение по ссылкам. Так как число разных ссылок с j_1 равно $|O(j_1)|$, то **с вероятностью**

$$p(i|j_1, \text{следуем по ссылкам}) = \frac{1}{|O(j_1)|}$$

мы можем достичь страницу i. Так как $p(\text{следуем по ссылкам}) = d$, то:

$$p(i|j_1) = d \cdot \frac{1}{|O(j_1)|}.$$

- Аналогичные рассуждения справедливы и для остальных страниц $j \in I(i)$. Тогда для $k = 1, \dots, s$ имеем

$$p(i|j_k) = d \cdot \frac{1}{|O(j_k)|}.$$

- Страницу i можно достичь одним из двух способов:
  1. следуя по некоторой ссылке на нее со страниц $j_1, \ldots, j_s$;
  2. случайно выбирая прыжок на i с любой текущей страницы.
- Отсюда получаем следующую формулу для вычисления вероятности $p(i)$:

$$p(i) = (1-d) \cdot \frac{1}{|V|} + (p(i|j_1) \cdot p(j_1) + \ldots + p(i|j_s) \cdot p(j_s)).$$

Подставляя сюда $p(i|j_k)$, получаем:

$$p(i) = (1-d) \cdot \frac{1}{|V|} + d \cdot \sum_{k=1}^{s} \frac{1}{|O_{j_k}|} \cdot p(j_k).$$

Таким образом,

$$pageRank(i) = (1-d) \cdot \frac{1}{|V|} + d \cdot \sum_{k=1}^{s} \frac{1}{|O_{j_k}|} \cdot pageRank(j_k). \tag{1}$$

Отметим, что это *рекурсивное определение.* Параметр d называется *коэффициентом затухания (demping factor)* и может принимать значения между 0 и 1. В [23] использовалось $d = 0.85$.

Вычисление рейтинга PageRank

Из формулы (1) следует, что для вычисления PageRank для некоторой страницы нам требуется знать рейтинги PageRank для всех ее "предков". Стандартный способ организовать такое вычисление состоит в последовательном итерировании.

Итеративное вычисление PageRank. Традиционный итеративный алгоритм для вычисления PageRank использует следующую процедуру:

$$pageRank^0(i) = \frac{1}{|V|} \quad \text{for all} i \in V \tag{2}$$

$$pageRank^r(i) = (1-d) \cdot \frac{1}{|V|} + d \cdot \sum_{k=1}^{s} \frac{1}{|O_{j_k}|} \cdot pageRank^{r-1}(j_k) \tag{3}$$

$$\textbf{Остановиться, когда: } \left(\sum_{i \in V} (pageRank^r(i) - pageRank^{r-1}(i)) \right) < \varepsilon \tag{4}$$

Доказано, что при некоторых условиях на граф (сильная связность и апериодичность – эти условия обеспечиваются введением прыжков с любой страницы на любую страницу) эта процедура сходится к некоторым стационарным значениям рейтингов $pageRank(i)$ (главному собственному вектору соответствующей стохастической матрицы). На самом деле, поскольку нас интересуют относительные рейтинги страниц, для определения их порядка сходимость не требуется и процедуру можно прерывать после небольшого числа итераций. В экспериментах авторов алгоритма ([23]) для графа с 322 миллионами ребер алгоритм PageRank сошелся за 52 итерации.

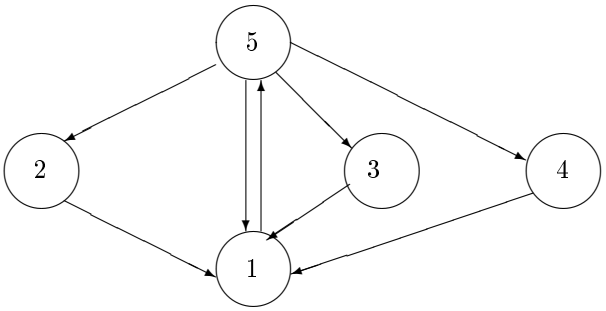


Рис. 45: Граф сети G_1

Рассмотрим сеть G_1, показанную на рис. 45. В качестве d зафиксируем значение 0.8. Тогда матрица

$$\mathbf{M} = (1-d)\frac{\mathbf{ONES}}{|V|} + d\mathbf{A}^T = \begin{pmatrix} 0,04 & 0,84 & 0,84 & 0,84 & 0,24 \\ 0,04 & 0,04 & 0,04 & 0,04 & 0,24 \\ 0,04 & 0,04 & 0,04 & 0,04 & 0,24 \\ 0,04 & 0,04 & 0,04 & 0,04 & 0,24 \\ 0,84 & 0,04 & 0,04 & 0,04 & 0,04 \end{pmatrix}$$

Начав вычисление рейтингов с равномерного распределения $\mathbf{P}^0 = (0.2, 0.2,\ 0.2, 0.2, 0.2)^T$, получаем следующую последовательность значений $\mathbf{P}^r$:

$\mathbf{P}^0$	$\mathbf{P}^1$	$\mathbf{P}^2$	$\mathbf{P}^3$	$\mathbf{P}^4$	$\mathbf{P}^5$	$\mathbf{P}^6$	$\mathbf{P}^7$
0,2	0,56	0,272	0,3296	0,42176	0,320384	0,357248	0,37641728
0,2	0,08	0,08	0,1376	0,09152	0,100736	0,1154816	0,09926144
0,2	0,08	0,08	0,1376	0,09152	0,100736	0,1154816	0,09926144
0,2	0,08	0,08	0,1376	0,09152	0,100736	0,1154816	0,09926144
0,2	0,2	0,488	0,2576	0,30368	0,377408	0,2963072	0,3257984

Квадрат расстояния между $\mathbf{P}^6$ и $\mathbf{P}^7$ меньше 0,0021. Поэтому можно использовать для ранжирования страниц ранги, полученные округлением $\mathbf{P}^7$, т.е. $pageRank(1) = 0,38; pageRank(2) = pageRank(3) = pageRank(4) = 0,1; pageRank(5) = 0,32$.

10.5 Обнаружение сообществ (Community Discovery)

Сообщество. Пусть $S = \{s_1, \ldots, s_n\}$ это множество однотипных объектов. **Сообщество** это пара $C = \langle T, G \rangle$, в которой T это объединяющая данное сообщество **тема**, а $G \subseteq S$ это множество **членов сообщества**.

Свойства. Это *весьма общее определение* **сообществ**. Алгоритмы для решения разных задач используют разные уточнения этого определения.

- Темы. Темы определяют соответствующие сообщества. Можно ожидать, что если для двух сообществ $C_1 = \langle T_1, G_1 \rangle$ и $C_2 = \langle T_2, G_2 \rangle$, $T_1 = T_2$, то также $G_1 = G_2$ и, следовательно, $C_1 = C_2$.

  (заметим, что обратное неверно. Вполне возможно, что у два разных сообщества образованы из одного и того же множества членов.)

- Темы сообществ могут быть самыми различными. Например, какие-то события, хобби, профессиональные интересы и т.д.
- Каждый объект может быть членом многих сообществ.
- Для некоторых приложений важен также временной аспект.

Задача обнаружения сообществ: для заданного множества данных, содержащего информацию об объектах, выявить (скрытые) сообщества этих объектов. Для каждого сообщества определить его тему и его членов. Обычно тема представляется набором некоторых ключевых слов.

10.6 Двудольное ядро сообществ

(i, j) **двудольное ядро** это **полный двудольный граф** $G = \langle F, C, E \rangle$, такой что:

- $F, C \subseteq S$
- $F \cap C = \emptyset$,
- $E = \{(f, c) | f \in F, c \in C\}$,
- $|F| = i$,
- $|C| = j$.

Элементы множества F называются фанами (fans), элементы множества C – центрами. На рис. 46 показано (4,5)-двудольное ядро.

Двудольное ядро представляет группу объектов (фанов), которые ссылаются на одно и то же множество центров. Поэтому двудольное ядро можно рассматривать как ядро некоторого сообщества, состоящего из фанов ядра, тема которого находится среди центров ядра.

10.6.1 Выявление двудольных ядер

Двудольные ядра можно обнаружить, используя следующую процедуру:

Вход: Граф $G_S = \langle S, E_S \rangle$, $S = \{s_1, \ldots, s_n\}$, $E \subseteq S \times S$. i, j - размер двудольного ядра.

Шаг 1. Сокращение. Множество S уменьшают дважды:

Шаг 1.1. Сокращение по степени захода. Удалить все вершины (pages) со степенью захода, превосходящей некоторую большую константу K. (например, $K = 50$). Причина этого сокращения в том, что страницы, на которые чересчур много ссылок, как правило, принадлежат "универсальным" сайтам типа Yahoo, Rambler и т.п. и не связаны с определенной темой или сообществом.

Шаг 1.2. Итеративное сокращение фанов и центров.
$S_0 = S$. Пока $S_i \neq S_{i-1}$:

- Удалить из S_{i-1} все вершины s со степенью исхода $d_o(s) < i$.
- Удалить из S_{i-1} все вершины s со степенью захода $d_i(s) < j$.

Шаг 2. Порождение двудольного ядра.
For $i = 1 \ldots k$ do:

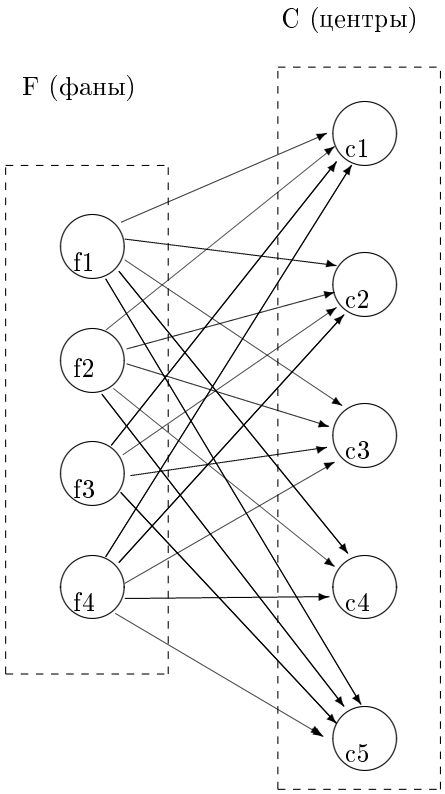


Рис. 46: (4,5)-двудольное ядро.

- Получить (i, j) двудольные сообщества.
  Базис: $(1, j)$ сообщество состоит из одной вершины s со степенью исхода $d_o(s) = j$.
  Индукционный шаг:
  - Для каждого центра $(i-1, j)$ сообщества, найти ссылающуюся на него вершину f', не входящую в сообщество. Если f' связана со всеми j центрами сообщества, то добавить f' в двудольное ядро в качестве нового фана.

Комментарий. Вообще говоря, двудольные ядра не определяют полностью сообщества. Скорее они выявляют их "центральную"часть и дают некоторое направление для поиска темы.

10.6.2 Сообщества максимального потока

Сообщества максимального потока. Пусть $G_S = \langle S, E_S \rangle$ – граф связей над множеством объектов S. Одно из определений сообщества можно сформулировать следующим образом:
Сообщество это подмножество объектов $C \subset S$ такое, что

- каждый объект $u \in C$ имеет больше ребер (входящих и исходящих), связывающих его с другими членами C, чем ребер, связывающих его с объектами из $S \setminus C$.

В общем случае, задача выделения таких сообществ является NP-полной. Поэтому в работе [18] было предложено приближенное решение этой задачи, основанное на нахождении макси-

мальных потоков. Обнаруженные таким образом сообщества называются *сообществами максимальных потоков*.

Напомним (см. раздел 7), что транспортная сеть $N =< G = (V, E), c, s, t >$, состоит из ориентированного графа G, ребра которого имеют пропускные способности $c(e) \geq 0$, источника $s \in V$, в который не входят ребра, и стока $t \in V$, из которого не выходят ребра. *Поток* f *в сети* N - это функция $f : E \to R$ такая, что

1) $0 \leq f(e) \leq c(e)$ для каждого ребра $e \in E$;

2) для каждой вершины $v \in V \setminus \{s, t\}$
$\sum\{f(e) |\ e$ *входит в* $v\} = \sum\{f(e) |\ e$ *выходит из* $v\}$.

Величина потока f в сети N $val(f) = \sum_{u \in V} f(s, u)$.

Поток с наибольшим значением величины называется *максимальным*.

В разделе 7 мы рассмотрели задачу определения максимального потока и описали алгоритм МАХП, решающий эту задачу за время $O(|V|^3)$ (). Применение этих алгоритмов для обнаружения сообществ в социальных сетях основано на следующей теореме Форда-Фалкерсона (теорема 7.1): *величина максимального потока в сети равна величине минимального разреза в ней.* Здесь *разрез* A *в сети* N - это подмножество вершин $A \subset V$ такое, что источник $s \in A$, а сток $t \in V \setminus A$. Разрез A однозначно задает множество ребер $P(A) = \{(u, v) \mid u \in A, v \in V \setminus A\}$, выходящих из A.

Если пропускные способности ребер равны 1, то для максимального потока $fmax$ его величина $val(fmax)$ равна минимальному числу ребер $|P(A)|$, соединяющих некоторый (минимальный) разрез A с остальной частью сети.

Рис. 47 иллюстрирует такие разрезы.

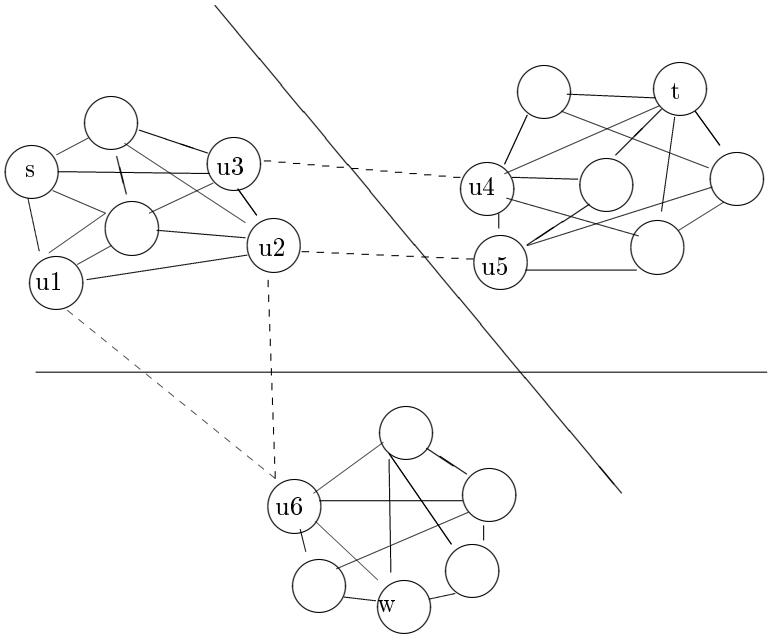


Рис. 47: Максимальный поток в сети. Сечения разделяют обнаруженные сообщества

Поток между вершинами s и t ограничен двумя ребрами "бутылочного горлышка" $(u3, u4)$ и $(u2, u5)$. Аналогично, поток между вершинами s и w ограничен ребрами $(u1, u6)$ $(u2, u6)$. Если эти ребра отсечь (удалить из сети, то мы получим три различных сообщества.

Алгоритм Find-Community(S^*, K – число итераций)

begin
  $C := S^*$;
  for $it = 1$ **to** K **do**
    { $G :=$ crawlGraph(C);
    $n := |S^*|$;
    $C^* :=$ Max-Flow-Community(G,C,n);
    ранжировать все $v \in C^*$ по числу ребер внутри C^*;
    $C := C \cup \{v \in C^* | v$ *имеет высокий ранг в* $C^*\}$
    };
  return C;
end

Function Max-Flow-Community(G= (V,E), C, k)

begin
  $V := V \cup \{s, t\}$;
  for all $v \in V$ **do** $\{E := E \cup \{(s, v)\};\ c(s, v) := \infty\}$;
  for all $(u, v) \in E$, $u \neq s$ **do**;
    { $c(u, v) = k$;
    if $(v, u) \notin E$ **then**
      $\{E := E \cup \{(v, u)\};\ c(v, u) := k\}$
    };
  for all $v \in V$, $v \notin C \cup \{s, t\}$ **do** $\{E := E \cup \{(v, t); c(v, t) = 1\}$;
  МАХП$'(G, s, t)$;
  return $\{v \in V \mid v$ *достижима из* $s\}$
end

Рис. 48: Алгоритм поиска сообщества максимального потока

Алгоритм Find-Community представлен на рис. 48. Он работает следующим образом.

- Вход. Социальная сеть G_S и множество исходных объектов (страниц) S^*. Предполагается, что пользователю известно, что исходные объекты принадлежат одному сообществу. Алгоритм будет определять границы этого сообщества.
- Выход. $C \supset S^*$ – список объектов сообщества.
- Процесс. Алгоритм работает в два этапа:
  - Этап 1. Расширение исходных страниц. Алгоритм обходит социальную сеть, начиная с исходных страниц, и собирает некоторую окрестность множества $C = S^*$. Для этого используется процедура crawlGraph(C), которая добавляет к C все вершины, находящиеся на ограниченном расстоянии от вершин C (при этом обход в ширину происходит без учета направления ребер).

– Этап 2. Максимальный поток.

Для отделения сообщества $C \supset S^*$ от остальной сети применяется алгоритм поиска максимального потока МАХП′(G, s, t). Предполагается, что этот алгоритм определения максимального потока алгоритмом МАХП удалит ребра, соединяющие минимальный разрез с остальным графом.

Для нахождения требуемого сообщества эти этапы могут повторяться несколько раз. Значения параметров K и k выбираются эмпирически. Часто выбирается значение $k = |S|$.

10.7 Задачи

Задача 10.1. *Докажите лемму 10.2.*

Задача 10.2. *Пусть $G = (I, E)$ – ориентированный граф. Предложите алгоритм для вычисления для вершины $i \in I$ значения ее близкой центральности (Closeness Centrality) $C_C(i)$ по формуле*

$$C_C(i) = \frac{n-1}{\sum_{j \in I} \rho(i,j)}.$$

Оцените сложность предложенного алгоритма.

Задача 10.3. *Пусть $G = (I, E)$ – нагруженный ориентированный граф с неотрицательными длинами ребер $c : E \to R^+$. Предложите реализацию алгоритма из теоремы 10.2, для вычисления значений срединной центральности $C_B(i)$ всех вершин $i \in V$. Оцените сложность предложенного алгоритма.*

Задача 10.4. *Графовая центральность (graph centrality (Hage, Harary, 1995)) вершины i графа $G = (I, E)$ определяется как*

$$C_G(i) = \frac{1}{\max_{k \in I} \rho(i,k)}.$$

Предложите алгоритмы для вычисления графовой центральности всех вершин для ненагруженных и нагруженных графов и оцените сложность этих алгоритмов.

Задача 10.5. *Центральность нагрузки (stress centrality (Shimbel, 1953)) определяется для вершины i графа $G = (I, E)$ как*

$$C_S(i) = \sum_{j \neq i \neq k} p_{jk}(i).$$

Предложите алгоритмы для вычисления центральности нагрузки всех вершин для ненагруженных и нагруженных графов и оцените сложность этих алгоритмов.

Задача 10.6. *Напомним, что* диаметром *(не)ориентированного графа $G = (V, E)$ с весами ребер $c : E \to R^+$ называется максимальное расстояние между вершинами графа $d_G = \max\{\rho(u, v) | u, v \in V\}$. (см. задачу 6.5). Радиальность (radiality (Valente, Foreman, 1998)) определяется для вершины i графа $G = (I, E)$ как*

$$C_R(i) = \frac{\sum_{k \in V} (d_g + 1 - \rho(i,k))}{(n-1)d_G}.$$

Предложите алгоритм для вычисления радиальности всех вершин для ненагруженных и нагруженных графов и оцените сложность этих алгоритмов.

Задача 10.7. *Пусть $G = (V, E)$ – ненагруженный ориентированный граф. Предложите алгоритм для вычисления для вершины $i \in V$ значения ее средней престижности (Proximity Prestige) $P_P(i)$, определяемой по формуле*

$$P_P(i) = \frac{|I_i|^2}{(n-1)\sum_{j \in I_i} \rho(j,i)},$$

где I_i это множество вершин, из которых можно достичь i в G, $n = |V|$. Оцените сложность предложенного алгоритма.

Задача 10.8. *Определите какие ранги присвоит алгоритм PageRank вершинам следующих графов.*

а) $G_1 = (V, E)$*:*
$V = \{a, b, c, d, e\}$
$E = \{(a, b)(b, a), (b, c), (c, a), (c, d), (d, a), (d, e), (e, a), (e, b)\}$.

б) $G_2 = (V, E)$*:*
$V = \{a, b, c, d, e\}$
$E = \{(a, b)(b, a), (c, a), (c, b), (d, a), (d, b), (e, a), (e, b)\}$.

в) $G_3 = (V, E)$*:*
$V = \{a, b, c, d\}$
$E = \{(a, b), (b, a), (b, c), (c, a), (c, d), (d, a), (d, c)\}$.

Поскольку в этих графов из каждой вершины выходят ребра, положим параметр $d = 1$.

Список литературы

[1] А. Ахо, Дж. Хопкрофт, Дж. Ульман Структуры данных и алгоритмы. М.: Вильямс, 2000.

[2] Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982, 416с.

[3] Евстигнеев В.А. Применение теории графов в программировании. М.:Наука, Физ-мат лит., 1985, 352с.

[4] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы (построение и анализ). – СПб.: Вильямс, 2005.

[5] Кристофидес Н. Теория графов. Алгоритмический подход. М. : Мир, 1978.

[6] Левин Л.А., Универсальные задачи перебора // Проблемы передачи информации, 9, 1973. С.115–116.

[7] Липский В. Комбинаторика для программистов.-М.:Мир,1988.

[8] Ловас Л., Пламмер М. Прикладные задачи теории графов. Теория паросочетаний в математике, физике, химии. – М.: Мир, 1988, 653с.

[9] Майника Э. Алгоритмы оптимизации на сетях и графах. -М.:Мир,1981.

[10] Пападимитриу Х., Стайглиц К. Комбинаторная оптимизация. Алгоритмы и сложность.-М.:Мир, 1985.

[11] Плесневич Г.С., Сапаров М.С. Алгоритмы в теории графов. Ашхабад:Ылым, 1981, 312 с.

[12] У. Татт. Теория графов. М.: Мир, 1988.

[13] Р. Уилсон. Введение в теорию графов. М.: Мир, 1977.

[14] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. The Traveling Salesman Problem: A computational study. Princeton University Press, 2007.

[15] Brandes U. A faster algorithm for betweeness centrality// Journal of Mathematical Sociology 25(2), 2001, 163-177.

[16] Cherkassky B.V., Goldberg A.V., Radzik T. Shortest paths algorithms: theory and experimental evaluation. Mathematical programming, 1996

[17] Cook S.A., The complexity of theorem-proving procedures // Proc. 3-th Ann ACM Symp. on Theory of Computing, 1971. P.151–158.
(Русск. перевод: Кук С.А., Сложность процедур вывода теорем. Киб. сб., нов.сер., вып. 12.— М.:Мир, 1975, 5-15).

[18] FlakeG. W., Lawrence S., Giles C. L., Coetzee F. Self-Organization of the Web and Identification of Communities. IEEE Computer 35(3),2002, 66–71.

[19] G. Gutin and A. Punnen. The Traveling Salesman Problem and Its Variations. Springer, 2007.

[20] Karp R.M., Reducibility among combinatorial problems, in R.E.Miller and J.W.Thatcher (eds),Complexity of Comuter Computations, Plenum Press, NY, 1972, 85-103.
(Русск. перевод: Карп Р.М. Сводимость комбинаторных задач.Киб. сб., нов.сер., вып. 12.— М.:Мир, 1975, 16-38).

[21] Kleinberg J. M. Authoritative sources in a hyperlinked environment. In Proc. of ACM-SIAM Symposium on Discrete Algorithms, 1998.

[22] E. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys. The Traveling Salesman Problem. John Wiley, 1985.

[23] Page, Lawrence; Brin, Sergey; Motwani, Rajeev and Winograd, Terry (1998). The PageRank citation ranking: Bringing order to the Web. *Technical Report, Department of Computer Science, Stanford University.*

[24] Rosenkrantz D.J., Stearns R.E., Lewis P.M. An analysis of several heuristics for the traveling salesman problem. // SIAM J. Comput. 6, 563-581.

[25] Scott, J. Social network analysis: A handbook (2nd ed.). SAGE publications, 2000.

[26] Social Network Data Analytics. Ed. C.C.Aggarwal. Springer, 2011.

[27] Xu G., Zhang Y., Li L. Web Mining and Social Networking. Techniques and Applications/ Springer Science+Business Media, 2011.

Предметный указатель

УДК 681.3.06 + 519.6
ББК 32.81

Учебное издание

Дехтярь Михаил Иосифович

Алгоритмические задачи на графах

Учебное пособие

Подписано в печать 01.10.2012. Уч.-изд.л.7,37. Электронное изд. Заказ 486
Тверской государственный университет
Факультет прикладной математики и кибернетики
Адрес: 170000, г.Тверь, пер.Садовый, 35.