Н. Д. Бубнова

АЛГОРИТМЫ НА ГРАФАХ

Учебно-методическое пособие

по дисциплине

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

Барнаул 2018

Бубнова Н.Д. Алгоритмы на графах : Учебно-методическое пособие по дисциплине «Алгоритмы и структуры данных». − Барнаул / Алтайский государственный технический университет им. И.И. Ползунова, 2018. − 95 с.

Данное пособие содержит варианты заданий для разработки и отладки программ; теоретические основы для их выполнения; примеры текстов программ для их чтения и понимания студентами; вопросы для самоконтроля.

Задания ориентированы на освоение обучающимися основных компетенций:

- готовность к применению основных методов и инструментов разработки программного обеспечения,
- готовность к оцениванию его временной и емкостной сложности;
- овладение навыками чтения, понимания и выделения главной идеи прочитанного исходного кода.

Учебно-методическое пособие предназначено для бакалавров, обучающихся по направлению подготовки «Программная инженерия».

Содержание

1 Разработка и анализ программ обхода графа

Лабораторная работа № 1
(работа в аудитории 6 ч., СРС 6 ч.)

Цели работы

– закрепление теоретических знаний по теме работы;

– углубление навыков по применению методов разработки программного обеспечения;

– углубление навыков оценки временной и емкостной сложности программного обеспечения;

– углубление навыков чтения, понимания и выделения главной идеи прочитанного исходного кода;

– приобретение практических навыков в разработке компьютерных программ, реализующих алгоритмы обхода графа (в ширину и в глубину).

Задания

Перед выполнением работы необходимо повторить теоретический материал по теме. Основные теоретические положения изложены ниже в данном параграфе пункте «Теоретические сведения». После этого следует ответить на тестовые вопросы по теме.

В результате выполнения лабораторной работы требуется написать и отладить программу на алгоритмическом языке, которая получает на входе числовые данные, выполняет их обработку в соответствии с требованиями задания и выводит результат на экран в графической форме. Для обработки данных необходимо реализовать алгоритмы обхода графа в соответствии с постановкой задачи. Ввод данных осуществляется из файла с учетом требований к входным данным, с учетом постановки задачи. Ограничениями на входные данные является допустимый диапазон значений используемых числовых типов в выбранной системе программирования. Работа состоит из двух указанных ниже этапов.

Этап 1. Выполнить приведенные ниже задания (работа в аудитории и СРС).

- Выбрать алгоритм решения поставленной задачи по заданному варианту (варианты заданий приведены ниже).
- Разработать тестовые примеры для отладки соответствующего алгоритма и программы. Согласовать с преподавателем.
- Разработать и обосновать структуры данных для реализации алгоритма.
- С учетом выполнения предыдущих пунктов задания написать и отладить программу на множестве тестовых примеров.
- Оформить отчет по выполненной работе.
- Защитить работу, ответив на вопросы преподавателя и студентов.

Этап 2. Выполнить приведенные ниже задания (работа в аудитории с использованием интерактивных технологий обучения):

1. Разделиться на команды по 2-3 человека в команде
2. Каждая команда выбирает один из отчетов, оформленный членом данной команды для передачи отчета другой команде.
3. Проанализировать полученный от другой команды отчет; прочитать исходный код; протестировать соответствующую программу на своих тестах; подготовить отзыв, осветив следующие вопросы по рецензируемому отчету:

– идея алгоритма;

– корректность выбора структур данных;

– полнота и качество реализации алгоритма в программе;

– грамотность оценки временной и емкостной сложности программы;

– качество тестов;
– полнота и качество оформления отчета в целом;
Дать итоговое заключение и оценку, указав достоинства и недостатки работы.
Результаты работы команд оцениваются преподавателем и другими командами.

Варианты алгоритмов

1. Найти самый длинный путь, соединяющий две данные вершины графа.
2. Найти количество путей между двумя заданными узлами по заданной матрице смежности.
3. Проверить граф на полноту. Граф задан матрицей смежности.
4. Найти самый длинный простой путь в графе.
5. Найти кратчайший путь между двумя заданными вершинами графа.
6. Найти все простые пути между двумя заданными вершинами, если граф задан списковой структурой.
7. Найти длину максимального по длине простого пути графа.
8. Перенумеровать узлы направленного ациклического графа так, чтобы получившаяся матрица смежности была бы нижней треугольной. Выполнить проверку графа на ацикличность и указать новую нумерацию узлов для ациклического графа.
9. Определить компоненты связности графа.
10. Проверить, будет ли в графе существовать простой путь, проходящий через все его вершины. Граф задан матрицей смежности.
11. Перейти от задания графа списком ребер к его заданию матрицей смежности.
12. Указать множество вершин графа, доступных из заданной вершины.
13. По матрице смежности построить матрицу путей.
14. Найти все простые пути начинающиеся в заданной вершине графа.
15. Найти максимальный полный подграф исходного графа.
16. Получить матрицу путей для графа, заданного связными списками.
17. Найти минимальный простой путь между двумя заданными вершинами графа, определенного списковой структурой.
18. Проверить, будет ли граф несвязным.
19. Проверить два графа на изоморфность.
20. Проверить, является ли граф, заданный списком, деревом.
21. Найти максимальную сильную компоненту графа.
22. Найти базу для заданного графа.
23. Подсчитать число путей заданной длины в графе и перечислить их.
24. Проверить, является ли граф, заданный списком ребер, полным. И если да, то получить матрицу смежности графа.
25. Перейти от задания графа матрицей смежности к его заданию списком вершин и ребер.
26. Найти самый короткий простой цикл во взвешенном графе, заданном взвешенной матрицей смежности.

Теоретические сведения

Графическая интерпретация различных моделей графов дана на рис. 1. В виде ориентированных графов можно представить логическую или функционально - логическую схему. На графовых моделях можно, например, оценить быстродействие схемы. Блок-схема алгоритма может быть представлена вероятностным графом для оценки временных характеристик алгоритма, затрат процессорного времени. Графом типа "дерево" можно отобразить практически любую структуру организации или предприятия.
Широкое применение теория графов получила при исследовании проблемы оптимизации, возникающей при конструировании больших как технических, так и программных систем например, таких, как компиляторы.

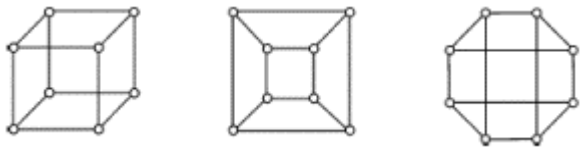


Рисунок 1 – Примеры моделей графа

Определение графа. Для описания строения различных систем, состоящих из связанных между собой элементов, часто используют графические схемы, изображая элементы точками (кружками, прямоугольниками и т.д.), а связи между ними – ориентированными или неориентированными линиями, соединяющими элементы. При этом получаются диаграммы, подобные представленной на рис 2.

На таких диаграммах важно лишь то, какие именно пары элементов соединены линиями, а не способ изображения элементов, форма или длина линий. Эту же структуру можно описать, не прибегая к графическому изображению, а просто перечислив пары связанных между собой элементов.

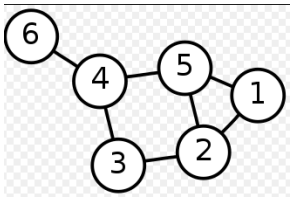


Рисунок 2 – Пример представления графа

Определение. Обыкновенным графом называется пара G = (V,E), где V - конечное множество, E - множество неупорядоченных пар различных элементов из V. Элементы множества V называются вершинами графа, элементы множества E - его ребрами.

Слегка модифицируя это определение, можно получить определения других типов графов без кратных ребер: если заменить при описании элементов множества E слово "неупорядоченных" словом "упорядоченных", получится определение ориентированного графа без петель, если убрать слово "различных", получится определение графа с петлями. Ориентированный граф часто называют орграфом.

В дальнейшем термин "граф" мы будем употреблять в смысле "обыкновенный граф", а рассматривая другие типы графов, будем специально это оговаривать.
Множество вершин графа G будем обозначать через VG, множество ребер - EG, число вершин - n(G), число ребер – m(G).

Термин "граф" неоднозначен, это легко заметить, сравнивая приводимые в разных книгах определения. Однако во всех этих определениях есть кое-что общее. В любом случае граф состоит из двух множеств - множества вершин и множества ребер, причем для каждого ребра указана пара вершин, которые это ребро соединяет. Вершины и ребра называются элементами графа. Будем рассматривать только конечные графы, то есть такие, у которых мно-

жества вершин и ребер конечны. Чтобы получить законченное определение графа того или иного типа, необходимо уточнить еще три момента.

Рассмотрим **виды графов.**

Ориентированный граф (кратко **орграф**) – (мульти) граф –это граф, рёбрам которого присвоено направление. Направленные рёбра именуются также *дугами*, а в некоторых источниках и просто рёбрами.

Ориентированный или неориентированный? Зададим ребро парой конечных вершин ребра. Прежде всего, договоримся, считаем ли мы пары (a,b) и (b,a) **различными.** Если да, то говорят, что рассматриваются упорядоченные пары (порядок элементов в паре важен), если нет - неупорядоченные. Если ребро соединяет вершину *a* с вершиной *b* и пара (*a, b*) считается упорядоченной, то это ребро называется ориентированным, вершина *a* - его началом, вершина *b* - концом. Если же эта пара считается неупорядоченной, то ребро называется неориентированным, а обе вершины – его концами. Чаще всего рассматривают графы, в которых все ребра имеют один тип – либо ориентированные, либо неориентированные. Соответственно и весь граф называют ориентированным или неориентированным. На рисунках ориентацию ребра (направление от начала к концу) указывают стрелкой.

Связный граф – граф, содержащий ровно одну компоненту связности. Это означает, что между любой парой вершин этого графа существует как минимум один путь.

Граф называется **односвязным (связным)**, если:

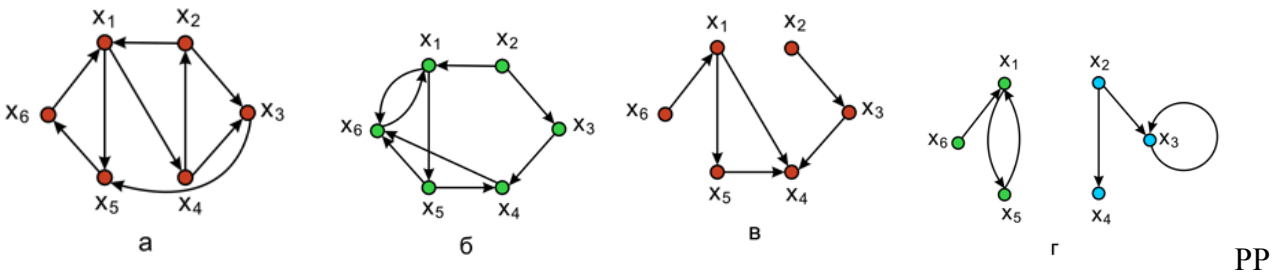
1. У него одна компонента связности.
2. Существует путь из любой вершины в любую другую вершину
3. Существует путь из заданной вершины в любую другую вершину
4. Содержит связный подграф, включающий все вершины исходного графа
5. Содержит в качестве подграфа дерево, включающее все вершины исходного графа (такое дерево называется остовным)
6. При произвольном делении его вершин на 2 группы всегда существует хотя бы 1 ребро, соединяющее пару вершин из разных групп

Орграф называется сильно связным, или сильным, если для двух любых различных его вершин x_i и x_j существует, по крайней мере, один путь, соединяющий эти вершины. Это определение означает также, что любые две вершины сильно связного графа взаимодостижимы. Пример данного графа показан на рис. 3а.

Орграф называется односторонне связным, или односторонним, если для любых двух различных его вершин x_i и x_j существует, по крайней мере, один путь из x_i в x_j или из x_j в x_i или оба пути существуют одновременно. Пример односторонне связного графа приведен на рис. 3б.

Орграф называется слабо связным, или слабым, если для любых двух различных вершин графа существует по крайней мере один маршрут, соединяющий их. Пример слабо связного графа – рис. 3в.

Орграф называется несвязным, если для некоторой пары вершин орграфа не существует маршрута, соединяющего их (рис. 3г).



РР

исунок 3 – Виды орграфов

Связный граф, не имеющий циклов, либо граф, в котором каждая пара вершин соединена одной и только одной простой цепью, называется **деревом.**

Степень вершины x (*degree)* в теории графов — количество рёбер графа, которым принадлежит (инцидентна) вершина x.

Полустепень захода в орграфе для вершины x – число дуг, входящих в вершину.

Полустепень исхода в орграфе для вершины x– число дуг, исходящих из вершины.

Взвешенные графы

Если графы используются для моделирования реальных систем, их вершинам, или ребрам, или и тем, и другим приписываются некоторые числа. Природа этих чисел может быть разнообразна. Например, если граф представляет собой модель железнодорожной сети, то число, приписанное ребру, может указывать длину перегона между двумя станциями, или наибольший вес состава, который допустим для этого участка пути, или среднее число поездов, проходящих через этот участок в течение суток и т.п. Сложилась традиция называть эти числа весами, а граф с заданными весами вершин и /или ребер - взвешенным графом.

Иногда дугам графа сопоставляют числа a_i -> c_i , называемые весом или длиной, или стоимостью или ценой. В каждом конкретном случае выбирается то слово, которое ближе подходит по смыслу задачи.

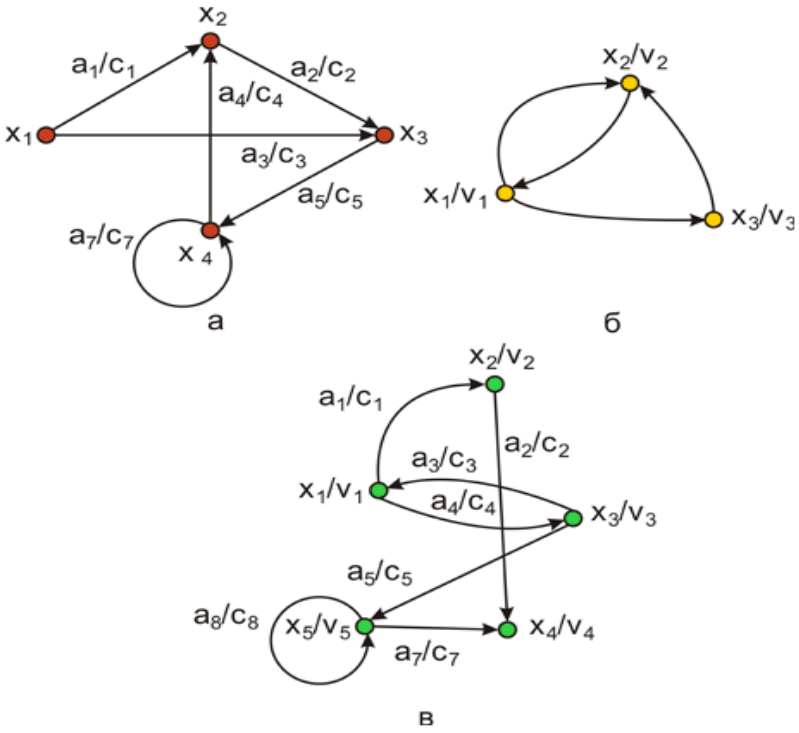


Рисунок 4 – Взвешенные графы: а – граф со взвешенными дугами; б – граф со взвешенными вершинами; в – взвешенный граф

Граф G, описываемый тройкой вида G = (X, A, C),
где X = { x_i }, i =1, 2, 3, ..., n – множество вершин,
A = { a_i }, i = 1, 2, 3, ..., m – множество дуг,
C = {C_i}, i = 1, 2, 3, ..., m – множество характеристик дуг, называется графом со взвешенными дугами.

Пример такого графа приведен на рис. 4.

Граф со взвешенными вершинами – это граф, описываемый тройкой G = (X, A, V),
где X = { x_i }, i = 1, 2, ..., n – множество вершин графа;
A = { a_i }, i = 1, 2, ..., m – множество дуг графа;
V ={ v_i }, i = 1, 2, ..., n – множество характеристик вершин.

В качестве характеристик вершин могут выступать "стоимость", "мощность", "вес" и т. п. Для графа со взвешенными вершинами в случае представления пути последовательностью вершин весом пути является сумма весов, входящих в этот путь вершин.

И наконец, взвешенный граф определяется четверкой вида G = (X, A, V, C), т. е. и дуги, и вершины этого графа имеют некоторые характеристики.

Областями применения взвешенных графов в качестве моделей являются транспортные задачи, задачи оптимизации сети связи и системы перевозок и др. Одной из известнейших оптимизационных задач является нахождение кратчайших путей в графе со взвешенными дугами.

Понятие пути в графе. Путем в орграфе называется последовательность дуг, в которой конечная вершина всякой дуги, кроме последней, является начальной вершиной следующей дуги.

Например, для графа на рис 5 последовательности дуг

M_1: a_6, a_5, a_9, a_8, a_4 ,

M_2: a_1, a_6, a_5, a_9, a_7 ,

M_3: a_1, a_6, a_5, a_9, a_{10}, a_6, a_4

являются путями. Пути могут быть различными.

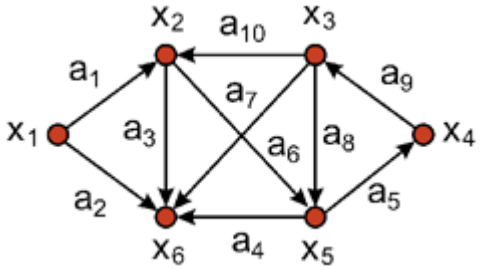


Рисунок 5 – Орграф

Маршрутом в орграфе называют чередующуюся последовательность вершин и *дуг*, вида $v_0\{v_0,v_1\}v_1\{v_1,v_2\}v_2...v_n$ (вершины могут повторяться). **Длина маршрута** — количество дуг в нем.

Путь есть *маршрут* в орграфе без повторяющихся дуг, **простой путь** — без повторяющихся вершин. Если существует путь из одной вершины в другую, то вторая вершина **достижима** из первой.

Контур есть замкнутый *путь*.

Для **полумаршрута** снимается ограничение на направление дуг, аналогично определяются **полупуть** и **полуконтур**.

Вес и длина пути. Если дугам графа сопоставлены числа a_i -> c_i , называемые весом или длиной, или стоимостью или ценой, то при рассмотрении пути M, представленного последовательностью дуг (a_1, a_2, ..., a_q), за его вес (или длину, или стоимость) принимается число L(M), равное сумме весов всех дуг, входящих в путь, т. е. $L(M) = \Sigma c_i$ для всех c_i □ M.

Длиной (или мощностью) пути называется число дуг, входящих в него. Чаще всего термин "длина" употребляется, когда все дуги, входящие в путь, имеют веса, равные 1, т. е. когда вес пути совпадает с его длиной (мощностью).

Каркасы/ Пусть G - обыкновенный граф. Его каркасом называется остовный подграф, в котором нет циклов, а области связности совпадают с областями связности графа G. Таким образом, каркас связного графа - дерево, а в общем случае - лес.

У любого графа есть хотя бы один каркас. Действительно, если в G нет циклов, то он сам является собственным каркасом. Если же циклы есть, то можно удалить из графа любое ребро, принадлежащее какому-нибудь циклу. Такое ребро не является мостом (мост – это ребро графа, удаление которого увеличивает число компонент связности), поэтому при его удалении области связности не изменятся. Продолжая действовать таким образом, после удаления некоторого количества ребер получим остовный подграф, в котором циклов уже нет, а об-

ласти связности - те же, что у исходного графа, то есть этот подграф и будет каркасом. Можно сказать, сколько ребер необходимо удалить для получения каркаса. Если в графе n вершин, m ребер и k компонент связности, то в каркасе будет тоже n вершин и k компонент связности. Но в любом лесе с n вершинами и k компонентами связности имеется ровно n-k ребер. Значит, удалено будет m-n+k ребер. Это число называется цикломатическим числом графа и обозначается через ν(G).

Способы задания графов. Ниже рассмотрены наиболее часто используемые способы задания графов.

Теоретико-множественное представление графов. Граф описывается перечислением множества вершин и дуг. Пример описания для орграфа приведен на рис 6: $G_4 = (X, A)$, где $X = \{x_i\}$, i = 1, 2, 3, 4 – множество вершин; $A = \{a_i\}$, i = 1, 2, ..., 6 – множество дуг, причем $A = \{(x_1, x_2), (x_4, x_2), (x_2, x_4), (x_2, x_3), (x_3, x_3), (x_4, x_1)\}$.

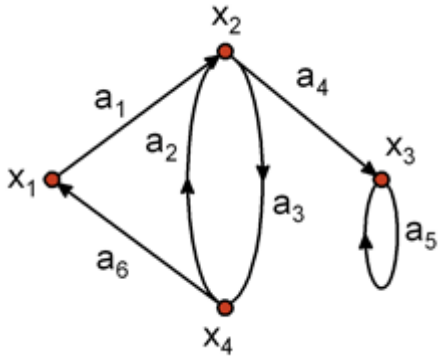


Рисунок 6 – орграф

Задание графов соответствием. Описание графов состоит в задании множества вершин X и соответствия Γ, которое показывает, как между собой связаны вершины.

Соответствием Γ называется отображение множества X в X, а граф в этом случае обозначается парой G = (X, Γ).

Отображением вершины x_i — $\Gamma(x_i)$ является множество вершин, в которые существуют дуги из вершины x_i, т. е. $\Gamma(x_i)=\{x_j : \square\text{дуга } (x_i, x_j) \square A\}$.

Так для орграфа на рис. 6 описание заданием множества вершин и соответствия выглядит следующим образом:

$G_4=(X, \Gamma)$,

где $X = \{x_i\}$, I = 1, 2, ..., 4 – множество вершин, $\Gamma(x_1) = \{ x_2 \}$, $\Gamma(x_2) = \{ x_3, x_4 \}$, $\Gamma(x_3) = \{ x_3 \}$, $\Gamma(x_4) = \{ x_1, x_2 \}$ – отображения.

Для неориентированного или смешанного графов предполагается, что соответствие Γ задает такой эквивалентный ориентированный граф, который получается из исходного графа заменой каждого неориентированного ребра двумя противоположно направленными дугами, соединяющими те же самые вершины.

Матричное представление графов. Для обработки на ЭВМ графы удобно представлять в виде матриц смежности и инциденций.

Матрица смежности – это квадратная матрица размерностью n x n, (где n – число вершин графа), однозначно представляющая его структуру.

$A = \{a_{ij}\}$, i, j = 1, 2, ..., n, а каждый элемент матрицы определяется следующим образом:

$a_{ij} = 1$, если существует дуга (x_i, x_j),

$a_{ij} = 0$, если нет дуги (x_i, x_j).

Матрица инциденций представляет собой прямоугольную матрицу размером n x m, где n – количество вершин графа, а m – количество дуг графа. Обозначается матрица инциденций $B = \{b_{ij}\}$, i = 1, 2, ..., n, j = 1, 2, ..., m .

Каждый элемент матрицы определяется следующим образом:

$b_{ij} = 1$, если x_i является начальной вершиной дуги a_j,

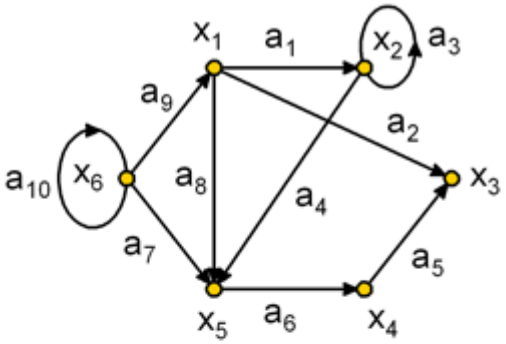
$b_{ij} = -1$, если x_i является конечной вершиной дуги a_j,

$b_{ij} = 0$, если x_i не является концевой вершиной дуги a_j или если a_j является петлей.

На рис. 7 (а,б) приведен граф и его матрица смежности, по которой можно найти характеристики вершин. Так сумма элементов i -ой строки матрицы дает полустепень исхода вершины xi, а сумма элементов i -го столбца дает полустепень захода вершины xi. По матрице смежности можно найти прямые и обратные отображения. Рассмотрим i -ю строку матрицы. Если элемент aij=1, то элемент графа xj входит в отображение Г(xi). Например, во 2-й строке матрицы A (рис 7,б) единицы стоят в 2-м и 5-м столбцах, следовательно, Г(x2) = { x2, x5}.

Для графа на рис. 7а матрица инциденций приведена на рис. 7в. Поскольку каждая дуга инцидентна двум различным вершинам, за исключением того случая, когда дуга образует петлю, то каждый столбец либо содержит один элемент равный 1 и один – равный – 1, либо все элементы столбца равны 0.

Для неориентированного графа, матрица инциденций определяется так же, за исключением того, что все элементы, равные –1, заменяются на 1.



а

A=	x_1	x_2	x_3	x_4	x_5	x_6
x_1	0	1	1	0	1	0
x_2	0	1	0	0	1	0
x_3	0	0	0	0	0	0
x_4	0	0	1	0	0	0
x_5	0	0	0	1	0	0
x_6	1	0	0	0	1	1

б

A=	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
x_1	1	1	0	0	0	0	0	1	-1	0
x_2	-1	0	0	1	0	0	0	0	0	0
x_3	0	-1	0	0	-1	0	0	0	0	0
x_4	0	0	0	0	1	-1	0	0	0	0
x_5	0	0	0	-1	0	1	-1	-1	0	0
x_6	0	0	0	0	0	0	1	0	1	0

в

Рисунок 7 – Орграф и его матричное представление: а – орграф;
б – матрица смежности; в – матрица инциденций

Обходы графа. Существует множество алгоритмов на графах, в основе которых лежит систематический перебор вершин графа, такой что каждая вершина просматривается (посещается) в точности один раз.

Под обходом графов (поиском на графах) понимается процесс систематического просмотра всех ребер или вершин графа с целью отыскания ребер или вершин, удовлетворяющих некоторому условию.

При решении многих задач, использующих графы, необходимы методы регулярного обхода вершин и ребер графов. К стандартным и наиболее распространенным методам относятся:

- поиск в глубину (Depth First Search, DFS);

- поиск в ширину (Breadth First Search, BFS).

Эти методы чаще всего рассматриваются на ориентированных графах, но они применимы и для неориентированных, ребра которых будем считать двунаправленными. Алгоритмы обхода в глубину и в ширину используются при решении различных задач обработки графов, например, построения остовного леса, проверки связности, ацикличности, вычисления расстояний между вершинами и других.

Поиск в глубину. При поиске в глубину посещается первая вершина, затем выполняется переход по ребрам графа, до тупика. Вершина графа является тупиковой, если все смежные с ней вершины уже посещены. После попадания в тупиковую вершину выполняется возврат по пройденному пути, пока не будет обнаружена вершина, у которой есть соседние, еще не посещенные вершины, а затем выполняется движение в этом новом направлении. Процесс оказывается завершенным при том возвращении в начальную вершину, при котором все смежные с ней вершины уже посещены.

Алгоритм обхода в глубину состоит из трех основных шагов.

Шаг 1. Всем вершинам графа присваивается значение не посещенная. Выбирается первая вершина и помечается как посещенная.

Шаг 2. Для последней помеченной как посещенная вершины выбирается смежная вершина, являющаяся первой помеченной как не посещенная, и ей присваивается значение посещенная. Если таких вершин нет, то берется предыдущая помеченная вершина.

Шаг 3. Повторять шаг 2 до тех пор, пока все вершины не будут помечены как посещенные (рис. 8).

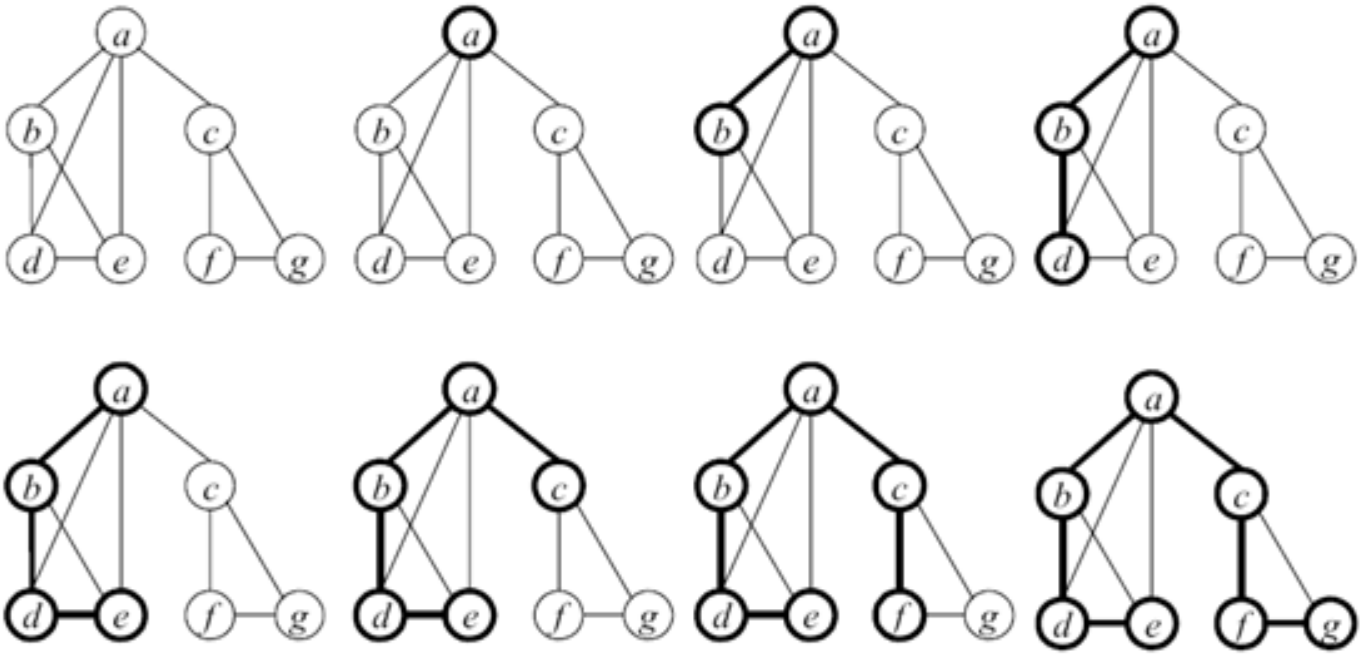


Рисунок 8 – иллюстрация схемы «обход в глубину»

Временная сложность зависит от представления графа. Если используется матрица смежности, то временная сложность равна $O(n^2)$, при нематричном представлении графа сложность –O(n+m), рассматриваются все вершины и все ребра.

Обход в ширину. При обходе в ширину, после посещения первой вершины, посещаются все соседние с ней вершины. Потом посещаются все вершины, находящиеся на расстоянии двух ребер от начальной. На каждом новом шаге посещаются вершины, расстояние от которых до начальной на единицу больше предыдущего. Чтобы предотвратить повторное посещение вершин, необходимо вести список посещенных вершин. Для хранения временных данных, необходимых для работы алгоритма, используется очередь – упорядоченная после-

довательность элементов, в которой новые элементы добавляются в конец, а старые удаляются из начала.

При использовании этого алгоритма обхода для каждой вершины сразу находится длина кратчайшего маршрута от начальной вершины.

Алгоритм обхода в ширину также состоит из трех основных шагов.

Шаг 1. Всем вершинам графа присваивается значение не посещенная. Выбирается первая вершина и помечается как посещенная (и заносится в очередь).

Шаг 2. Посещается первая вершина из очереди (если она не помечена как посещенная). Все соседние с ней вершины заносятся в очередь. После этого она удаляется из очереди.

Шаг 3. Повторяется шаг 2 до тех пор, пока очередь не пуста (рис. 9).

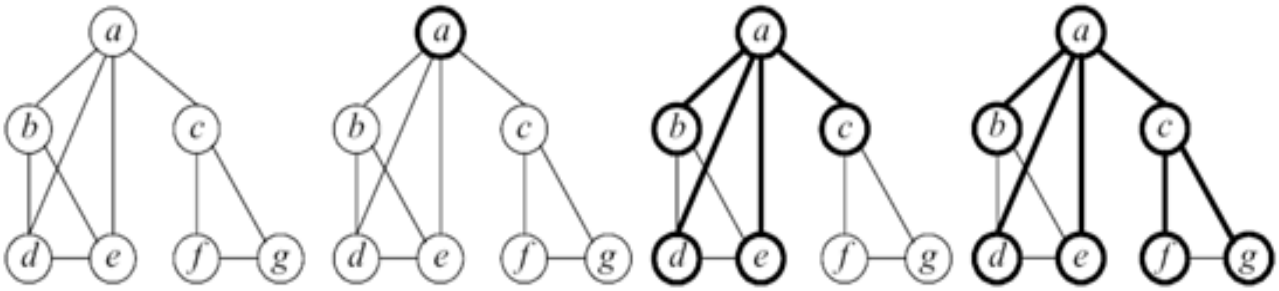


Рисунок 9 – иллюстрация схемы «обход в ширину»

Не трудно обосновать, что сложность поиска в ширину при нематричном представлении графа равна O(n+m), так как рассматриваются все n вершин и m ребер. Если граф задан матрицей смежности сложность поиска оценивается как $O(n^2)$

Тестовые вопросы по теме

1. Как от вида или представления графа зависит временная сложность алгоритмов?
2. Как от вида или представления графа зависит временная сложность алгоритмов поиска в глубину?
3. Как от вида или представления графа зависит временная сложность алгоритмов поиска в ширину?
4. Как при реализации в коде выполняется возвращение из тупиковых вершин при обходе графа?
5. Как выполняется обход в несвязном графе?
6. Распространяются ли понятия "поиск в глубину" и "поиск в ширину" на несвязный граф? Ответ обоснуйте.
7. Охарактеризуйте трудоемкость рекурсивного и нерекурсивного алгоритмов обхода графа.
8. Сформулируйте идею алгоритма нахождения самого длинного пути между двумя данными вершина ми графа.
9. Сформулируйте идею алгоритма проверки графа на полноту, если граф задан матрицей смежности.
10. Сформулируйте идею алгоритма нахождения самого длинного простого цикла в графе.
11. Сформулируйте идею алгоритма нахождения компонент связности графа.
12. Сформулируйте идею алгоритма нахождения пути, проходящего через все его вершины. Граф задан матрицей смежности.
13. Сформулируйте идею алгоритма нахождения множества вершин графа, доступных из заданной вершины.
14. Сформулируйте идею алгоритма нахождения матрицы путей по матрице смежности

15. Сформулируйте идею алгоритма нахождения максимального полного подграфа исходного графа.
16. Сформулируйте идею алгоритма нахождения матрицы путей для графа, заданного связными списками.
17. Сформулируйте идею алгоритма нахождения минимального простого пути между двумя заданными вершинами взвешенного графа, определенного списковой структурой.
18. Сформулируйте идею алгоритма проверки графа на несвязность.
19. Сформулируйте идею алгоритма проверки двух графов на изоморфность.
20. Сформулируйте идею алгоритма проверки: является ли граф, заданный списком, деревом.
21. Сформулируйте идею алгоритма нахождения базы для заданного графа.
22. Как связаны между собой различные способы представления графов?

Источники по теме 1

1 Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. –М.: МЦНМО, 2000. – 960 с.
2 Кристофидес Н. Теория графов. Алгоритмический подход. М.: Мир, 1978. – 432 с.
3 https://www.intuit.ru/studies/courses/1033/241/lecture/6218 – дата съема информации 31.01.2018
4 https://www.intuit.ru/studies/courses/58/58/lecture/1708 – дата съема информации 31.01.2018
5 https://www.intuit.ru/studies/courses/648/504/lecture/11474 – дата съема информации 31.01.2018

2 Разработка и анализ программ нахождения кратчайших путей в графах

Лабораторная работа № 2
(работа в аудитории 4 ч., СРС 4 ч.)

Цели работы

– закрепление теоретических знаний по теме работы;

– углубление навыков по применению методов разработки программного обеспечения;

– углубление навыков оценки временной и емкостной сложности программного обеспечения;

– углубление навыков чтения, понимания и выделения главной идеи прочитанного исходного кода;

– приобретение практических навыков в разработке компьютерных программ, реализующих алгоритмы нахождения кратчайших путей в графах.

Задания

Перед выполнением работы необходимо повторить теоретический материал по теме. Основные теоретические положения изложены ниже в данном параграфе пункте «Теоретические сведения». После этого следует ответить на тестовые вопросы по теме.

В результате выполнения лабораторной работы требуется написать и отладить программу на алгоритмическом языке (C++ или ином), которая получает на входе числовые данные, выполняет их обработку в соответствии с требованиями задания и выводит результат на экран в графической форме. Для обработки данных необходимо реализовать алгоритмы обхода графа в соответствии с постановкой задачи. Ввод данных осуществляется из фай-

ла с учетом требований к входным данным, с учетом постановки задачи. Ограничениями на входные данные является допустимый диапазон значений используемых числовых типов в выбранной системе программирования. Работа состоит из двух указанных ниже этапов.

Этап 1. Выполнить приведенные ниже задания (работа в аудитории и СРС).

1. Выбрать алгоритм решения поставленной задачи по заданному варианту (варианты заданий приведены ниже).
2. Разработать тестовые примеры для отладки соответствующего алгоритма и программы. Согласовать с преподавателем.
3. Разработать и обосновать структуры данных для реализации алгоритма.
4. С учетом выполнения предыдущих пунктов задания написать и отладить программу на множестве тестовых примеров.
5. Оформить отчет по выполненной работе.
6. Защитить работу, ответив на вопросы преподавателя и студентов.

Этап 2. Выполнить приведенные ниже задания (работа в аудитории с использованием интерактивных технологий обучения)

1. Разделиться на команды по 2-3 человека в команде
2. Каждая команда выбирает один из отчетов, оформленный членом данной команды для передачи отчета другой команде.
3. Проанализировать полученный от другой команды отчет; прочитать исходный код; протестировать соответствующую программу на своих тестах; подготовить отзыв, осветив следующие вопросы по рецензируемому отчету:

– идея алгоритма;
– корректность выбора структур данных;
– полнота и качество реализации алгоритма в программе;
– грамотность оценки временной и емкостной сложности программы;
– качество тестов;
– полнота и качество оформления отчета в целом;

Дать итоговое заключение и оценку, указав достоинства и недостатки работы.
Результаты работы команд оцениваются преподавателем и другими командами.

Варианты алгоритмов

1. Найти самый короткий путь, соединяющий две данные вершины графа. Граф задан матрицей весов. Все веса неотрицательные.
2. Написать программу для Нахождения числа кратчайших путей между двумя заданными вершинами по заданной матрице смежности. Все веса неотрицательные.
3. Найти самый короткий путь, соединяющий две данные вершины графа. Граф задан множеством вершин и ребер. Веса ребер заданы. Все веса неотрицательные.
4. Найти самый длинный путь, соединяющий две данные вершины графа. Граф задан матрицей весов. Все веса неотрицательные
5. Найти самый длинный путь, соединяющий две данные вершины графа. Граф задан матрицей весов. Возможны отрицательные веса ребер.
6. Найти все кратчайшие пути между двумя заданными вершинами, если граф задан списковой структурой. Все веса ребер неотрицательные.
7. Найти все кратчайшие пути между всеми вершинами, если граф задан списковой структурой. Все веса ребер неотрицательные.
8. Найти все кратчайшие пути между всеми вершинами, если граф задан списковой структурой. Возможны ребра с отрицательными весами.
9. Написать программу для определения компонент связности графа.

10. Проверить, будут ли в графе существовать кратчайшие пути одинаковых весов между различными парами вершин. Граф задан списковой структурой. Возможны ребра с отрицательными весами.
11. Проверить, будут ли в графе существовать кратчайшие пути одинаковых весов между различными парами вершин. Граф задан списковой структурой. Все ребра имеют неотрицательные веса.
12. Проверить, будут ли в графе существовать кратчайшие пути одинаковых весов между различными парами вершин. Граф задан матрицей смежности. Возможны ребра с отрицательными весами.
13. Проверить, будут ли в графе существовать кратчайшие пути одинаковых весов между различными парами вершин. Граф задан матрицей смежности. Все ребра имеют неотрицательный вес.
14. Указать множество вершин графа, доступных из заданной вершины на путях, веса которых не превышают заданной величины. Граф задан матрицей смежности. Все ребра имеют неотрицательный вес.
15. Указать множество вершин графа, доступных из заданной вершины на путях, веса которых не превышают заданной величины. Граф задан матрицей смежности. Возможен отрицательный вес ребер.
16. Указать множество вершин графа, доступных из заданной вершины на путях, веса которых не превышают заданной величины. Граф задан списковой структурой. Возможен отрицательный вес ребер.
17. Указать множество вершин графа, доступных из заданной вершины на путях, веса которых не превышают заданной величины. Граф задан списковой структурой. . Все ребра имеют неотрицательный вес.
18. Найти вершины, максимально и минимально удаленные от заданной вершины. Граф задан списковой структурой. Все ребра имеют неотрицательный вес.
19. Найти вершины, максимально и минимально удаленные от заданной вершины. Граф задан списковой структурой. Возможен отрицательный вес ребер.
20. . Найти вершины, максимально и минимально удаленные от заданной вершины Граф задан матрицей смежности. Все ребра имеют неотрицательный вес.
21. Найти вершины, максимально и минимально удаленные от заданной вершины Граф задан матрицей смежности. Возможен отрицательный вес ребер.

Теоретические сведения

Постановка задачи нахождения кратчайшего пути в графах. Пусть дан граф $G = (X, \Gamma)$, дугам которого приписаны веса (стоимости), задаваемые матрицей $C = [c_{ij}]$. Задача о кратчайшем пути состоит в нахождении кратчайшего пути от заданной начальной вершины $s \in X$ до заданной конечной вершины $t \in X$, при условии, что такой путь существует, т.е. при условии $t \in R(s)$. Здесь $R(s)$ - множество, достижимое из вершины *s*. Элементы c_{ij} матрицы весов *C* могут быть положительными, отрицательными или нулями. Единственное ограничение состоит в том, чтобы в *G* не было циклов с отрицательным суммарным весом. Если такой цикл все же существует и x_i - некоторая его вершина, то, двигаясь от *s* к x_i, обходя затем цикл достаточно большое число раз и попадая, наконец, в *t*, мы получим путь со сколь угодно малым весом, означающим, что кратчайшего пути не существует.

Следующие задачи являются обобщениями сформулированной выше задачи о кратчайшем пути.

1. Для заданной начальной вершины *s* найти кратчайшие пути между *s* и всеми другими вершинами $x_i \in X$.

2. Найти кратчайшие пути между всеми парами вершин.

На практике часто требуется найти не только кратчайший путь, но также второй, третий и т.д. кратчайшие пути в графе. Располагая этими результатами, можно решить, какой путь выбрать в качестве наилучшего (указанный подход полезен при использовании таких критериев, которые являются субъективными по своей природе или не могут быть непосредственно включены в алгоритм).

Алгоритм Дейкстры (случай неотрицательной матрицы весов). Эффективный алгоритм решения задачи о кратчайшем (*s*-*t*)-пути первоначально дал Дейкстра. Этот метод основан на приписывании вершинам временных пометок, причем пометка вершины дает верхнюю границу длины пути от *s* к этой вершине. Значения пометок постепенно уменьшаются с помощью некоторой итерационной процедуры, и на каждом шаге итерации одна из временных пометок становится постоянной. Последнее указывает на то, что пометка уже не является верхней границей, а дает точную длину кратчайшего пути от *s* к рассматриваемой вершине.

Описание алгоритма Дейкстра:

Пусть $l(x_i)$ - пометка вершины x_i,

Г(р) – множество вершин, доступных из вершины р на пути длины 1.

Шаг 1. Положить $l(s) = 0$ и считать эту пометку постоянной. Положить $l(x_i) = \infty$ для всех $x_i \neq s$ и считать эти пометки временными. Положить $p = s$.

Шаг 2. Для всех $x_i \in \Gamma(p)$, пометки которых временные, изменить пометки в соответствии со следующим правилом:

$$l(x_i) = \min[l(x_i), l(p) + c(p, x_i)] \tag{2.1}$$

Шаг 3. Среди всех вершин с временными пометками найти такую, для которой $l(x_i^*) = \min[l(x_i)]$.

Шаг 4. Считать пометку вершины x_i^* постоянной и положить $p = x_i^*$.

Шаг 5.

1. Если $p = t$, то $l(p)$ является длиной кратчайшего пути. Если $p \neq t$, перейти к шагу 2 (для случая поиска пути от *s* к *t*.)

2. Если все вершины отмечены как постоянные, то эти пометки дают длины кратчайших путей. Если некоторые пометки являются временными перейти к шагу 2 (для случая поиска путей от *s* ко всем остальным вершинам).

Как только длины кратчайших путей от *s* будут найдены, то сами пути можно получить при помощи рекурсивной процедуры с использованием соотношения (2.2). Так как вершина x'_i непосредственно предшествует вершине x_i в кратчайшем пути от *s* к x_i, то для любой вершины x_i соответствующую вершину x'_i можно найти как одну из оставшихся вершин, для которой

$$l(x'_i) + c(x'_i, x_i) = l(x_i). \tag{2.2}$$

Если кратчайший путь от *s* до любой вершины x_i является единственным, то дуги (x'_i, x_i) этого кратчайшего пути образуют ориентированное дерево с корнем *s*. Если существует несколько "кратчайших" путей от *s* к какой-либо другой вершине, то при некоторой

фиксированной вершине x'_i соотношение (2) будет выполняться для более чем одной вершины x_i. В этом случае выбор может быть либо произвольным (если нужен какой-то один кратчайший путь между *s* и x_i), либо таким, что рассматриваются все дуги (x'_i, x_i), входящие в какой-либо из кратчайших путей, и при этом совокупность всех таких дуг образует не ориентированное дерево, а общий граф, называемый базой относительно *s*.

Пример. Рассмотрим граф *G*, изображенный на рис. 2.1, где каждое неориентированное ребро рассматривается как пара противоположно ориентированных дуг равного веса. Матрица весов *C* приведена ниже.

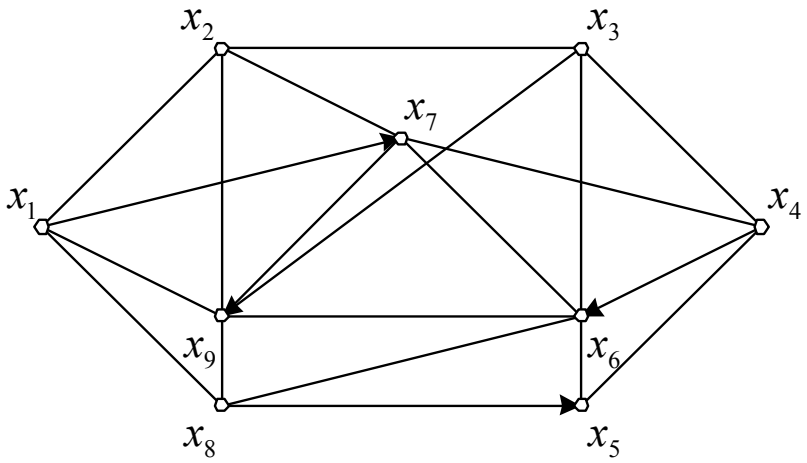


Рисунок 2.1 – Граф *G*.

Требуется найти все кратчайшие пути от вершины x_1 ко всем остальным вершинам. Постоянные пометки будем снабжать знаком +, остальные пометки рассматриваются как временные.

$C=$

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
x_1	0	12	0	0	0	0	5	2	10
x_2	12	0	14	0	0	0	4	0	15
x_3	0	14	0	20	0	25	0	0	9
x_4	0	0	20	0	2	15	5	0	0
x_5	0	0	0	2	0	8	0	0	0
x_6	0	0	25	0	8	0	12	11	10
x_7	0	4	0	5	0	12	0	0	22
x_8	2	0	0	0	24	11	0	0	7
x_9	10	15	0	0	0	10	22	7	0

Воспользуемся алгоритмом Дейкстры.

Шаг 1. $l(x_1) = 0^+$, $l(x_i) = \infty \ \forall x_i \neq x_1$, $p = x_1$.

Первая итерация

Шаг 2. $\Gamma(p) = \Gamma(x_1) = \{x_2, x_7, x_8, x_9\}$ - все пометки временные. Возьмем сначала x_2. Из (1.1) получаем

$$l(x_2) = \min[\infty, 0^+ + 12] = 12,$$

аналогично $l(x_7) = 5$, $l(x_8) = 2$, $l(x_9) = 10$.

Шаг 3. $\min\left[\underset{x_2}{12},\ \underset{x_7}{5}\ ,\ \underset{x_8}{2}\ ,\underset{x_9}{10},\ \underset{x_3,x_4,x_5,x_6}{\infty}\right]=2$ соответствует x_8 .

Шаг 4. x_8 получает постоянную пометку $l(x_8)=2^+$, $p=x_8$.

Шаг 5. Не все вершины имеют постоянные пометки, поэтому переходим к шагу 2. Пометки в начале следующей итерации показаны на рис. 1.2(а).

Вторая итерация

Шаг 2. $\Gamma(p)=\Gamma(x_8)=\{x_5,x_6,x_9\}$ − все пометки временные. Из соотношения (1.1) имеем

$$l(x_5)=\min[\infty,\ 2^+ +24]=26,$$

аналогично $l(x_6)=13$, $l(x_9)=9$. Пометки изображены на рис. 1.2(б).

Шаг 3. $\min\left[\underset{x_2}{12},\underset{x_5}{26},\underset{x_6}{13},\ \underset{x_7}{5}\ ,\ \underset{x_9}{9}\ ,\ \underset{x_3,x_4}{\infty}\right]=5$ соответствует x_7 .

Шаг 4. x_7 получает постоянную пометку $l(x_7)=5^+$, $p=x_7$.

Шаг 5. Перейти к шагу 2.

Третья итерация

Шаг 2.

$\Gamma(p)=\Gamma(x_7)=\{x_2,x_4,x_6,x_9\}$

из соотношения (1.1) получаем

$l(x_2)=\min[12,\ 5^+ +4]=9$,

и аналогично:

$l(x_4)=10$

$l(x_6)=13$

$l(x_9)=9$

Шаг 3.

$$\min\left[\underset{x_2}{9},\underset{x_4}{10},\underset{x_5}{26},\underset{x_6}{13},\underset{x_9}{9},\underset{x_3}{\infty}\right]=9$$

что соответствует x_2.

Шаг 4. x_2 получает постоянную пометку

$l(x_2)=9^+$

$p=x_2$.

Шаг 5. Перейти к шагу 2.

Четвертая итерация

Шаг 2.

$\Gamma(p)=\Gamma(x_2)=\{x_3,x_7,x_9\}$

здесь не все пометки временные, поэтому из соотношения (1.1) получаем

$l(x_3)=\min[\infty,\ 9^+ +14]=23$,

и аналогично:

$$l(x_9) = 9$$

Шаг 3.

$$\min\left[\underset{x_3}{23},\underset{x_4}{10},\underset{x_5}{26},\underset{x_6}{13},\underset{x_9}{9}\right] = 9$$

что соответствует x_9.

Шаг 4. x_9 получает постоянную пометку

$$l(x_9) = 9^+, \quad p = x_9.$$

Шаг 5. Перейти к шагу 2.

Пятая итерация

Шаг 2.

$$\Gamma(p) = \Gamma(x_9) = \{x_2, x_6, x_7, x_8\}$$

здесь не все пометки временные, поэтому из соотношения (1.1) получаем

$$l(x_6) = \min[13, 9^+ + 10] = 13$$

Шаг 3.

$$\min\left[\underset{x_3}{23},\underset{x_4}{10},\underset{x_5}{26},\underset{x_6}{13}\right] = 10$$

что соответствует x_4.

Шаг 4. x_4 получает постоянную пометку:

$$l(x_4) = 10^+, \quad p = x_4.$$

Шаг 5. Перейти к шагу 2.

Шестая итерация

Шаг 2.

$$\Gamma(p) = \Gamma(x_4) = \{x_3, x_5, x_6\}$$

здесь все пометки временные, из соотношения (1.1) получаем

$$l(x_3) = \min[23, 10^+ + 20] = 23,$$

и аналогично:

$$l(x_5) = 12, \quad l(x_6) = 13.$$

Шаг 3.

$$\min\left[\underset{x_3}{23},\underset{x_5}{12},\underset{x_6}{13}\right] = 12$$

что соответствует x_5.

Шаг 4. x_5 получает постоянную пометку:

$$l(x_5) = 12^+, \quad p = x_5.$$

Шаг 5. Перейти к шагу 2.

Седьмая итерация

Шаг 2.

$$\Gamma(p) = \Gamma(x_5) = \{x_4, x_6\}$$

здесь не все пометки временные, поэтому из соотношения (1.1) получаем

$$l(x_6) = \min[13, 12^+ + 8] = 13.$$

Шаг 3.

$$\min\left[\underset{x_3}{23},\underset{x_6}{13}\right]=13$$

соответствует x_6.

Шаг 4. x_6 получает постоянную пометку

$$l(x_6)=13^+,\ p=x_6.$$

Шаг 5. Перейти к шагу 2.

Восьмая итерация

Шаг 2.

$$\Gamma(p)=\Gamma(x_6)=\{x_3,x_5,x_8,x_9\}$$

здесь не все пометки временные, поэтому из соотношения (1.1) получаем

$$l(x_3)=\min[23,13^+ + 25]=23.$$

Шаг 3.

$$\min\left[\underset{x_3}{23}\right]=23$$

что соответствует x_3.

Шаг 4. x_3 получает постоянную пометку:

$$l(x_6)=23^+,\ p=x_3.$$

Шаг 5. Все вершины имеют постоянные пометки. Конец работы алгоритма. Пометки, полученные в результате работы алгоритма, показаны на рис. 1.2(в).

Найдем кратчайший путь между вершиной x_2 и начальной вершиной x_1, последовательно используя соотношение (1.2).

Полагая $x_1 = x_2$, находим вершину x'_2, непосредственно предшествующую x_2 в кратчайшем пути от x_1 к x_2: вершина x'_2, должна удовлетворять соотношению

$$l(x'_2)+c(x'_2,x_2)=l(x_2)=9.$$

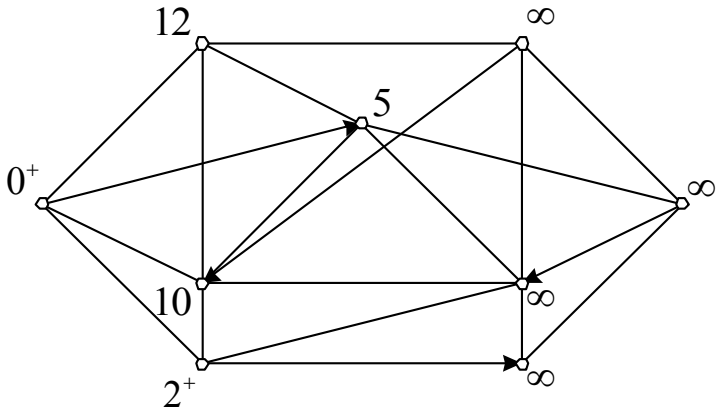
Единственной такой вершиной является x_7. Далее, применяем второй раз соотношение (1.2), беря $x_i = x_7$; получаем вершину x'_7, непосредственно предшествующую x_7 в кратчайшем пути от x_1 к x_2. Вершина x'_7 удовлетворяет соотношению

$$l(x'_7)+c(x'_7,x_7)=l(x_7)=5.$$

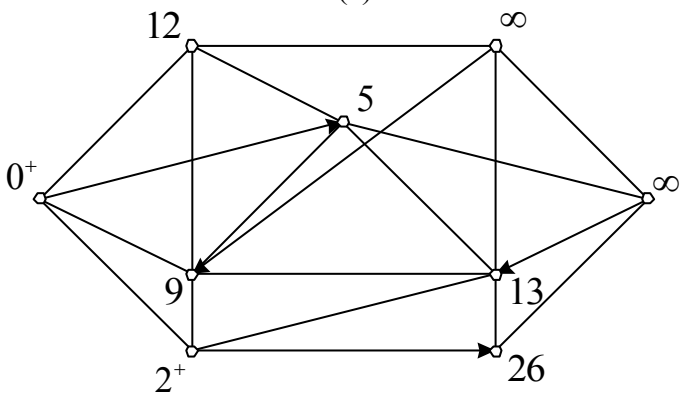
Единственной такой вершиной является x_1 и поэтому кратчайший путь от x_1 к x_2 есть:

$$(x_1, x_7, x_2)$$

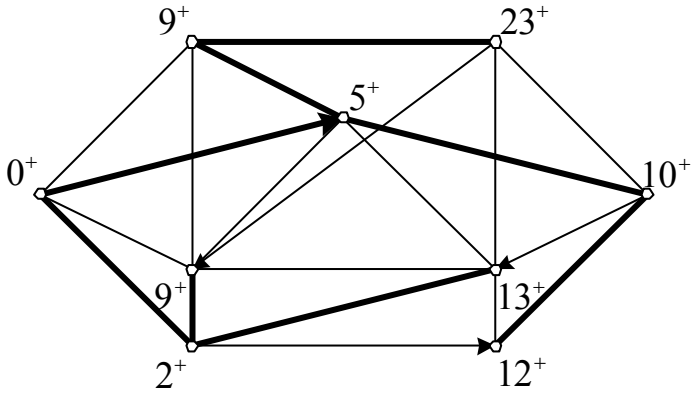
x_1- база, дающая все кратчайшие пути от x_1, представляет собой дерево, изображенное жирными линиями на рис. 1.2 (в).



(а)



(б)



(в)

Рисунок 2.2: а) пометки в конце 1-й итерации; б) пометки в конце шага 2 на 2-й итерации; в) окончательные пометки вершин

Алгоритм Форда (случай общей матрицы весов). Алгоритм Дейкстры применим лишь в том случае, когда $c_{ij} \geq 0$ для всех *i* и *j*. Однако если матрица *C* является матрицей стоимостей, то дуги, приносящие доход, должны иметь отрицательные «стоимости». В этом случае для нахождения кратчайших путей между вершиной *s* и всеми другими вершинами можно воспользоваться описанной ниже процедурой. Этот метод также является итерационным и основан на пометках вершин, причем в конце *k*-й итерации пометки равны длинам тех кратчайших путей (от s ко всем остальным вершинам), которые содержат не более *k*+1 дуг. В отличие от алгоритма Дейкстры никакая из пометок во время этого процесса не рассматривается как окончательная. Описываемый метод был первоначально предложен Фордом.

Описание алгоритма Форда. Пусть $l^k(x_i)$ - пометка вершины x_i в конце (k+1)–й итерации,

Г(р) – множество вершин, доступных из вершины р на пути длины 1.

Шаг 1. Положить S=Г(s), k=1, $l^1(s)=0$, $l^1(x_i)=c(s, x_i)$ для всех $x_i \in \Gamma(s)$ и $l^1(x_i)=\infty$ для всех остальных x_i.

Шаг 2. Для каждой вершины $x_i \in \Gamma(s)$ ($x_i \neq s$) изменить ее пометку следующим образом:

$$l^{k+1}(x_i)=\min\left[l^k(x_i), \min_{x_j \in T_i}\left\{l^k(x_j)+c(x_j, x_i)\right\}\right],$$

где $T_i = \Gamma^{-1}(x_i) \cap S$ (множество T_i содержит те вершины, для которых текущие кратчайшие пути из s состоят из k дуг и для которых существуют дуги к вершине x_i.). Для вершин $x_i \notin \Gamma(s)$ положим $l^{k+1}(x_i)=l^k(x_i)$.

Шаг 3.

1. Если $k \leq n-1$ и $l^{k+1}(x_i)=l^k(x_i)$ для всех x_i, то получен оптимальный ответ и пометки равны длинам кратчайших путей.

2. Если $k<n-1$ и $l^{k+1}(x_i) \neq l^k(x_i)$ для некоторой вершины x_i, то перейти к шагу 4.

3. Если $k=n-1$ и $l^{k+1}(x_i) \neq l^k(x_i)$ для некоторой вершины x_i, то в графе существует цикл отрицательного веса и задача не имеет решения.

Шаг 4. Обновить множество S следующим образом:

$$S=\left\{x_i \mid l^{k+1}(x_i) \neq l^k(x_i)\right\}.$$

Шаг 4. Положить $k=k+1$ и перейти к шагу 2.

Также можно привести общую матрицу весов к неотрицательной, увеличив вес каждого ребра на величину:

$$\max[\ \ abs\ \ (c_{ij}\ )]$$

и найти кратчайшие пути между вершиной s и всеми другими вершинами с помощью алгоритма Дейкстры.

Алгоритм Флойда (кратчайшие пути между всеми парами вершин). Пусть требуется найти кратчайшие пути между всеми парами вершин графа. Очевидный способ получить ответ на этот вопрос заключается в n-кратном применении алгоритма Дейкстры или алгоритма Форда, причем каждый раз в качестве начальной вершины s берутся различные вершины. В случае полного графа с неотрицательной матрицей весов С время, необходимое для вычислений, пропорционально n^3, а для произвольной матрицы весов оно пропорционально n^4.

Опишем совершенно иной подход к задаче нахождения кратчайших путей между всеми парами вершин. Этот метод применим как к неотрицательным, так и к произвольным

матрицам весов и время, необходимое для вычислений, пропорционально n^3. Если этот метод применим к графам с неотрицательной матрицей весов, то он сэкономит почти 50 % времени по сравнению с n-кратным применением алгоритма Дейкстры. Метод был предложен первоначально Флойдом. Он базируется на использовании последовательности из n преобразований (итераций) начальной матрицы весов C. При этом на k-й итерации матрица представляет длины кратчайших путей между каждой парой вершин с тем ограничением, что путь между x_i и x_j (для любых x_i и x_j) содержит в качестве промежуточных только вершины из множества:

$$\{x_1, x_2, \ldots, x_k\}.$$

Описание алгоритма Флойда.

Предположим, что в начальной матрице весов $c_{ij} = 0$ для всех i = 1,2,… n и $c_{ij} = \infty$, если в графе отсутствует дуга (x_i , x_j).

Шаг 1. Присвоить k =0.

Шаг 2. k = k +1.

Шаг 3. Для всех i ≠ k, таких, что $c_{ij} \neq \infty$, и для всех j ≠ k, , таких, что $c_{ij} \neq \infty$, введем операцию

$$c_{ij} = \min[c_{ij}, c_{ik} + c_{kj}]. \qquad (2.3)$$

Шаг 4.

1. Если $c_{ij} < 0$, то в графе G существует цикл отрицательного веса, содержащий вершину x_i, и решения нет.

2. Если все $c_{ij} \geq 0$ и $k=n$, то получено решение. Длины всех кратчайших путей содежатся в матрице:

$$[c_{ij}]$$

3. Если все $c_{ij} \geq 0$, но $k<n$, то вернуться к шагу 2.

Сами кратчайшие пути можно найти по их длинам с помощью рекурсивной процедуры, подобной той, которая выше определялась соотношением (2.2). С другой стороны, можно использовать информацию о самих путях (наряду с информацией о длинах путей). Этот последний метод полезен в тех случаях, когда требуется найти в графе цикл отрицательного веса (если такой существует). В этом методе в дополнение к матрице весов C хранится и обновляется вторая (n x n) –матрица^

$$\Theta = [\theta_{ij}].$$

Элемент Θ_{ij} указывает вершину, непосредственно предшествующую вершине x_j в кратчайшем пути от x_i к x_j . Матрице Θ присваиваются начальные значения $\Theta_{ij} = x_i$ для всех x_i и x_j. В соответствии с (1.3) на шаге 3 алгоритма обновление матрицы происходит так:

$$\theta_{ij} = \begin{cases} \theta_{kj}, \text{ если } c_{ik} + c_{kj} < c_{ij} \\ \text{не изменяется, если } c_{ik} + c_{kj} \geq c_{ij} \end{cases}.$$

В конце алгоритма кратчайшие пути получаются непосредственно из заключительной матрицы Θ. Таким образом, кратчайший путь между двумя вершинами x_i и x_j определяется следующей последовательностью вершин:

$$x_i, x_\nu, \ldots, x_\gamma, x_\beta, x_\alpha, x_j,$$

где $x_\alpha = \theta_{ij}$, $x_\beta = \theta_{i\alpha}$, $x_\gamma = \theta_{i\beta}$

и т.д. до $x_i = \theta_{i\nu}$.

Здесь следует отметить, что если всем c_{ij} придать начальные значения ∞ (а не 0), то конечное значение величины c_{ij} будет равно весу цепи, проходящей через вершину x_i. Исхо-

дя из структуры матрицы Θ, полученной в процессе той итерации, когда элемент c_{ij} становится отрицательным, можно найти цикл отрицательного веса, соответствующий этому элементу.

Применение к моделям сетевого планирования и управления. Допустим, что нужно реализовать некоторый проект, состоящий из большого числа взаимосвязанных этапов (работ). Событие, связанное с началом каждого этапа можно изобразить вершиной некоторого графа и построить дугу от вершины *i* к вершине *j*, чтобы показать, что событие *i* должно предшествовать событию *j*. Каждой дуге приписывается некоторый вес c_{ij}, равный продолжительности этапа *(i;j)*.

В задаче требуется найти минимальное время, необходимое для реализации всех работ (этапов) проекта. Иными словами, нужно найти в графе самый длинный путь между вершиной s, изображающей начало, и вершиной t, изображающей завершение всех необходимых для реализации проекта работ. Самый длинный путь называется критическим путем, так как этапы, относящиеся к этому пути, определяют полное время реализации проекта, и всякая задержка с началом выполнения любого из этих этапов приведет к задержке выполнения проекта в целом.

Данную задачу можно решить как задачу нахождения кратчайшего пути, используя, в частности, алгоритм Дейкстры, заменив операцию min на max. Можно для этого использовать также специальные алгоритмы, разработанные для расчета параметров сетевых графиков.

Тестовые вопросы по теме

1. Какие алгоритмы нахождения кратчайших путей Вы знаете?
2. Существуют ли ограничения на веса путей? Если существуют, то какие?
3. Будет ли правильно работать алгоритм Дейкстры для графа, из каждой вершины которого выходит одно ребро и оно имеет отрицательный вес?
4. Можно ли использовать для нахождения кратчайшего пути в ациклическом графе алгоритм Дейкстры, если в графе ребра, исходящие из истока, являются единственными ребрами с отрицательными весами?
5. Применим ли алгоритм Флойда для полных орграфов?
6. Какие алгоритмы применимы для нахождения всех кратчайших путей в графах?
7. Можно ли алгоритм для нахождения кратчайших путей в орграфах применять для неориентированных графов?
8. Как найти второй кратчайший путь после нахождения первого?
9. Как найти ребро, удаление которого вызывает максимальное возрастание длины кратчайшего пути из одной заданной вершины в другую в заданном ациклическом графе.
10. Приведите пример графа с V вершинами и E ребрами, для которых время работы алгоритма Дейкстры будет наихудшим.
11. Приведите оценку временной сложности алгоритма Флойда.
12. Приведите оценку временной сложности алгоритма Дейкстры.
13. Приведите оценку временной сложности алгоритма Форда.
14. В каких случаях следует отдать предпочтение алгоритму Флойда, а не Дейкстры?
15. Оцените с точностью до порядка наибольший размер (количество вершин) графа, который ваш компьютер и система программирования могут обработать за 10 секунд, если для вычисления кратчайших путей использовать алгоритм Флойда.
16. Можно ли найти кратчайший путь в графе с несколькими истоками? Если можно, то какой алгоритм следует использовать и как его применить?
17. Всегда ли с помощью алгоритма Флойда можно найти кратчайший путь?
18. Оцените временную сложность алгоритмов Дейкстры, Флойда и Форда.
19. Какие еще алгоритмы можно применить для нахождения кратчайших путей?

Рекомендуемые источники по теме 2

1 Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. – М.: МЦНМО, 2000. – 960 с.
2 Кристофидес Н. Теория графов. Алгоритмический подход. – М.: Мир, 1978. – 432 с.
3 https://www.intuit.ru/studies/courses/1033/241/lecture/6218 – дата съема информации 31.01.2018
4 https://www.intuit.ru/studies/courses/58/58/lecture/1708 – дата съема информации 31.01.2018
5 https://www.intuit.ru/studies/courses/648/504/lecture/11474 – дата съема информации 31.01.2018

3 Исследование программ, реализующих алгоритмы нахождения каркасов графа

Лабораторная работа № 3
(работа в аудитории 2 ч., СРС 2 ч.)

Цели работы

– закрепление теоретических знаний по теме работы;

– углубление навыков чтения, понимания и выделения главной идеи прочитанного исходного кода;

– углубление навыков оценки временной и емкостной сложности программного обеспечения;

– закрепление практических навыков работы в команде и навыков использования интерактивных технологий.

Задания

1 Типовые задания

Перед выполнением работы необходимо повторить теоретический материал по теме. Основные теоретические положения изложены ниже в данном параграфе пункте «Теоретические сведения». После этого следует ответить на тестовые вопросы по теме.

Рекомендуется также во время подготовки к аудиторному занятию предварительно ознакомиться с текстом программы (см. ниже «**Пример текста программы на построение каркасов**»).

В результате выполнения лабораторной работы требуется прочитать, проанализировать и выделить главную идею прочитанного исходного кода, реализующего алгоритм нахождения каркаса в графе.

Для выполнения работы нужно выполнить следующие этапы.

1. Разделиться на команды по 2-3 человека в команде
2. Каждая команда получает отчет, предложенный преподавателем.
3. Проанализировать полученный отчет; прочитать исходный код; протестировать соответствующую программу на своих тестах; подготовить отзыв, осветив следующие вопросы по рецензируемому отчету:

– идея алгоритма;

– корректность выбора структур данных;

– полнота и качество реализации алгоритма в программе;

– грамотность оценки временной и емкостной сложности программы;

– качество тестов;

– полнота и качество оформления отчета в целом;

– имеющиеся в программе ошибки (если они есть).

Дать итоговое заключение и оценку, указав достоинства и недостатки работы.

Результаты работы команд оцениваются преподавателем и другими командами.

2 Индивидуальные задания

Индивидуальные задания по данной работе выполняются по желанию студентов – для получения рейтинга, превышающего 75 баллов.

При выполнении индивидуального задания по выданному преподавателем варианту (варианты заданий см. ниже) требуется написать программу на алгоритмическом языке (C++ или ином), которая получает на входе числовые данные, выполняет их обработку в соответствии с требованиями задания и выводит результат на экран в графической форме. Для обработки данных необходимо реализовать алгоритм нахождения каркаса графа в соответствии с постановкой задачи. Ввод данных осуществляется из файла с учетом требований к входным

данным, с учетом постановки задачи. Ограничениями на входные данные является допустимый диапазон значений используемых числовых типов в выбранной системе программирования.

Варианты индивидуальных заданий по теме 3

1. В Алтайский государственный Технический Университет приезжает делегация, которой хотят продемонстрировать все помещения университета. Там, где она пойдет, должна быть постелена дорожка. Определить, по какому маршруту следует расстелить дорожки так, чтобы их суммарная длина была бы минимальна (уменьшение финансовых затрат). Входные данные заданы в «input.txt». В первой строке содержится количество комнат. Далее каждая строка содержит описание одной комнаты. Сначала следуют координаты комнаты, далее следует количество коридоров выходящих из этой комнаты и следуют номера комнат, с которыми эти коридоры соединят текущую комнату. Нумерация комнат начинается с 0. В выходном файле «output.txt» должны содержаться коридоры, по которым необходимо проложить дорожку (коридоры указываются номерами комнат). Для решения задачи использовать алгоритм Прима.

2. Директор Алтай Сбербанка решил произвести upgrade локальной компьютерной сети с минимальными затратами на кабели. Сеть представлена графом, который задается матрицей смежности в файле «input.txt». Для решения задачи использовать алгоритм Прима.

3. Чтобы разнообразить жизнь на туристической базе, было решено обновить подвесной комплекс, при этом площадки уже были построены. Определить, какой длины веревочную бухту необходимо купить, чтобы затраты на покупку были бы минимальны, и все площадки были бы доступны. Для решения задачи использовать алгоритм Прима.

4. Задано число N, где N – количество точек на плоскости. Построить стягивающее дерево так, чтобы оно было оптимальным. Оптимальное дерево имеет минимальную сумму длин входящих в него ребер. Все ребра – это расстояния между вершинами, заданными координатами точек на плоскости. Для решения задачи использовать алгоритм Прима.

5. Сеть состоит из N компьютеров, некоторые из них соединены прямыми двухсторонними каналами. Для повышения секретности было решено сократить число каналов до минимума и повысить степень защищенности каналов, но при этом любые компьютеры должны сохранить возможность обмена информацией или напрямую, или через другие компьютеры сети. Какие каналы было решено оставить, если сумма дополнительных вложений по каждому каналу пропорциональна длине канала и общая сумма вложений минимизируется. Для решения задачи использовать алгоритм Прима.

6. В далекой стране не очень большой бюджет, но власти хотят, чтобы все города были соединены дорогами. Требуется указать, между какими городами нужно строить дороги. Входные данные заданы в «input.txt». В первой строке записывается количество городов. Далее следует матрица смежности, в которой 0 означает невозможность построить дорогу, а другое число означает стоимость постройки. В выходном файле «output.txt» должны содержаться дороги, которые необходимо построить (дороги указываются номерами городов). Нумерация городов начинается с 0. Для решения задачи использовать алгоритм Прима.

7. Представьте, что Вы начальник отдела автоматизации новой фирмы, Вам предстоит соединить компьютеры в сеть. Но разумеется начальник не будет Вами доволен, если Вы не сделаете это за минимальную цену. Входные данные заданы в «input.txt». На первой строке содержится количество компьютеров. Далее каждая строка содержит описание одного компьютера. Сначала следуют координаты компьютера, далее следует количество возможных сетевых трасс от этого компьютера и следуют номера компьютеров, с которыми эти трассы соединяют текущий компьютер. Нумерация компьютеров начинается с 1. В выходном файле «output.txt» должны содержаться трассы, по которым необходимо проложить сетевые кабели (трассы указываются номерами компьютеров). Для решения задачи использовать алгоритм Прима.

8. Вы нарядили новогоднюю елку, и выяснили, что гирлянды не соединены проводом. От Вас требуется определить, хватит ли у Вас провода, чтобы их соединить, и если хватит, то как это сделать. Входные данные заданы в «input.txt». На первой строке содержится количество гирлянд. На второй строке содержится длина провода, который Вы имеете. Далее каждая строка содержит описание одной гирлянды. Сначала следуют координаты гирлянды, далее следует количество возможных соединений данной гирлянды с другими, и следуют номера гирлянд, с которыми данная может соединяться. Нумерация гирлянд начинается с 0. В выходном файле «output.txt» должен содержаться 0, если Вам не хватит провода, или должны содержаться соединения (соединения указываются номерами гирлянд). Для решения задачи использовать алгоритм Прима.

9. Необходимо соединить некоторым образом между собой определенные факты, но соединение одного факта с разными имеет различную стоимость и необходимо суммарную стоимость минимизировать. Входные данные заданы в «input.txt». В первой строке содержится количество фактов. Далее следует матрица смежности, в которой 0 означает отсутствие возможности соединения фактов, а число отличное от нуля означает стоимость соединения. В выходном файле «output.txt» должны содержаться соединения, которые указываются номерами фактов. Нумерация фактов начинается с 0. Для решения задачи использовать алгоритм Прима.

10. Задано N точек на плоскости и требуется проложить между ними граф путей, так, чтобы минимизировать сумму длин путей. Для решения задачи использовать алгоритм Прима.

11. В Алтайский государственный Технический Университет приезжает делегация, которой хотят продемонстрировать все помещения университета. Там, где она пойдет, должна быть постелена дорожка. Определить, по какому маршруту следует расстелить дорожки так, чтобы их суммарная длина была бы минимальна (уменьшение финансовых затрат). Входные данные заданы в «input.txt». В первой строке содержится количество комнат. Далее каждая строка содержит описание одной комнаты. Сначала следуют координаты комнаты, далее следует количество коридоров выходящих из этой комнаты и следуют номера комнат, с которыми эти коридоры соединят текущую комнату. Нумерация комнат начинается с 0. В выходном файле «output.txt» должны содержаться коридоры, по которым необходимо проложить дорожку (коридоры указываются номерами комнат). Для решения задачи использовать алгоритм Борувки.

12. Директор Алтай Сбербанка решил произвести upgrade локальной компьютерной сети с минимальными затратами на кабели. Сеть представлена графом, который задается матрицей смежности в файле «input.txt». Для решения задачи использовать алгоритм Борувки.

13. Чтобы разнообразить жизнь на туристической базе, было решено обновить подвесной комплекс, при этом площадки уже были построены. Определить, какой длины веревочную бухту необходимо купить, чтобы затраты на покупку были бы минимальны, и все площадки были бы доступны. Для решения задачи использовать алгоритм Борувки.

14. Задано число N, где N – количество точек на плоскости. Построить стягивающее дерево так, чтобы оно было оптимальным. Оптимальное дерево имеет минимальную сумму длин входящих в него ребер. Все ребра – это расстояния между вершинами, заданными координатами точек на плоскости. Для решения задачи использовать алгоритм Борувки.

15. Сеть состоит из N компьютеров, некоторые из них соединены прямыми двухсторонними каналами. Для повышения секретности было решено сократить число каналов до минимума и повысить степень защищенности каналов, но при этом любые компьютеры должны сохранить возможность обмена информацией или напрямую, или через другие компьютеры сети. Какие каналы было решено оставить, если сумма дополнительных вложений по каждому каналу пропорциональна длине канала и общая сумма вложений минимизируется. Для решения задачи использовать алгоритм Борувки.

16. В далекой стране не очень большой бюджет, но власти хотят, чтобы все города были соединены дорогами. Требуется указать, между какими городами нужно строить дороги. Входные данные заданы в «input.txt». В первой строке записывается количество городов. Далее следует матрица смежности, в которой 0 означает невозможность построить дорогу, а другое число означает стоимость постройки. В выходном файле «output.txt» должны содержаться дороги, которые необходимо построить (дороги указываются номерами городов). Нумерация городов начинается с 0. Для решения задачи использовать алгоритм Борувки.

17. Представьте, что Вы начальник отдела автоматизации новой фирмы, Вам предстоит соединить компьютеры в сеть. Но разумеется начальник не будет Вами доволен, если Вы не сделаете это за минимальную цену. Входные данные заданы в «input.txt». На первой строке содержится количество компьютеров. Далее каждая строка содержит описание одного компьютера. Сначала следуют координаты компьютера, далее следует количество возможных сетевых трасс от этого компьютера и следуют номера компьютеров, с которыми эти трассы соединяют текущий компьютер. Нумерация компьютеров начинается с 1. В выходном файле «output.txt» должны содержаться трассы, по которым необходимо проложить сетевые кабели (трассы указываются номерами компьютеров). Для решения задачи использовать алгоритм Борувки.

18. Вы нарядили новогоднюю елку, и выяснили, что гирлянды не соединены проводом. От Вас требуется определить, хватит ли у Вас провода, чтобы их соединить, и если хватит, то как это сделать. Входные данные заданы в «input.txt». На первой строке содержится количество гирлянд. На второй строке содержится длина провода, который Вы имеете. Далее каждая строка содержит описание одной гирлянды. Сначала следуют координаты гирлянды, далее следует количество возможных соединений данной гирлянды с другими, и следуют номера гирлянд, с которыми данная может соединяться. Нумерация гирлянд начинается с 0. В выходном файле «output.txt» должен содержаться 0, если Вам не хватит провода, или должны содержаться соединения (соединения указываются номерами гирлянд). Для решения задачи использовать алгоритм Борувки.

19. Необходимо соединить некоторым образом между собой определенные факты, но соединение одного факта с разными имеет различную стоимость и необходимо суммарную стоимость минимизировать. Входные данные заданы в «input.txt». В первой строке содержится количество фактов. Далее следует матрица смежности, в которой 0 означает отсутствие возможности соединения фактов, а число отличное от нуля означает стоимость соединения. В выходном файле «output.txt» должны содержаться соединения, которые указываются номерами фактов. Нумерация фактов начинается с 0. Для решения задачи использовать алгоритм Борувки.

20. Задано N точек на плоскости и требуется проложить между ними граф путей, так, чтобы минимизировать сумму длин путей. Для решения задачи использовать алгоритм Борувки.

21. В Алтайский государственный Технический Университет приезжает делегация, которой хотят продемонстрировать все помещения университета. Там, где она пойдет, должна быть постелена дорожка. Определить, по какому маршруту следует расстелить дорожки так, чтобы их суммарная длина была бы минимальна (уменьшение финансовых затрат). Входные данные заданы в «input.txt». В первой строке содержится количество комнат. Далее каждая строка содержит описание одной комнаты. Сначала следуют координаты комнаты, далее следует количество коридоров выходящих из этой комнаты и следуют номера комнат, с которыми эти коридоры соединят текущую комнату. Нумерация комнат начинается с 0. В выходном файле «output.txt» должны содержаться коридоры, по которым необходимо проложить дорожку (коридоры указываются номерами комнат). Для решения задачи использовать алгоритм Крускала.

22. Директор Алтай Сбербанка решил произвести upgrade локальной компьютерной сети с минимальными затратами на кабели. Сеть представлена графом, который задается

матрицей смежности в файле «input.txt». Для решения задачи использовать алгоритм Крускала.

23. Чтобы разнообразить жизнь на туристической базе, было решено обновить подвесной комплекс, при этом площадки уже были построены. Определить, какой длины веревочную бухту необходимо купить, чтобы затраты на покупку были бы минимальны, и все площадки были бы доступны. Для решения задачи использовать алгоритм Крускала.

24. Задано число N, где N – количество точек на плоскости. Построить стягивающее дерево так, чтобы оно было оптимальным. Оптимальное дерево имеет минимальную сумму длин входящих в него ребер. Все ребра – это расстояния между вершинами, заданными координатами точек на плоскости. Для решения задачи использовать алгоритм Крускала.

25. Сеть состоит из N компьютеров, некоторые из них соединены прямыми двухсторонними каналами. Для повышения секретности было решено сократить число каналов до минимума и повысить степень защищенности каналов, но при этом любые компьютеры должны сохранить возможность обмена информацией или напрямую, или через другие компьютеры сети. Какие каналы было решено оставить, если сумма дополнительных вложений по каждому каналу пропорциональна длине канала и общая сумма вложений минимизируется. Для решения задачи использовать алгоритм Крускала.

26. В далекой стране не очень большой бюджет, но власти хотят, чтобы все города были соединены дорогами. Требуется указать, между какими городами нужно строить дороги. Входные данные заданы в «input.txt». В первой строке записывается количество городов. Далее следует матрица смежности, в которой 0 означает невозможность построить дорогу, а другое число означает стоимость постройки. В выходном файле «output.txt» должны содержаться дороги, которые необходимо построить (дороги указываются номерами городов). Нумерация городов начинается с 0. Для решения задачи использовать алгоритм Крускала.

27. Представьте, что Вы начальник отдела автоматизации новой фирмы, Вам предстоит соединить компьютеры в сеть. Но разумеется начальник не будет Вами доволен, если Вы не сделаете это за минимальную цену. Входные данные заданы в «input.txt». На первой строке содержится количество компьютеров. Далее каждая строка содержит описание одного компьютера. Сначала следуют координаты компьютера, далее следует количество возможных сетевых трасс от этого компьютера и следуют номера компьютеров, с которыми эти трассы соединяют текущий компьютер. Нумерация компьютеров начинается с 1. В выходном файле «output.txt» должны содержаться трассы, по которым необходимо проложить сетевые кабели (трассы указываются номерами компьютеров). Для решения задачи использовать алгоритм Крускала.

28. Вы нарядили новогоднюю елку, и выяснили, что гирлянды не соединены проводом. От Вас требуется определить, хватит ли у Вас провода, чтобы их соединить, и если хватит, то как это сделать. Входные данные заданы в «input.txt». На первой строке содержится количество гирлянд. На второй строке содержится длина провода, который Вы имеете. Далее каждая строка содержит описание одной гирлянды. Сначала следуют координаты гирлянды, далее следует количество возможных соединений данной гирлянды с другими, и следуют номера гирлянд, с которыми данная может соединяться. Нумерация гирлянд начинается с 0. В выходном файле «output.txt» должен содержаться 0, если Вам не хватит провода, или должны содержаться соединения (соединения указываются номерами гирлянд). Для решения задачи использовать алгоритм Крускала.

29. Необходимо соединить некоторым образом между собой определенные факты, но соединение одного факта с разными имеет различную стоимость и необходимо суммарную стоимость минимизировать. Входные данные заданы в «input.txt». В первой строке содержится количество фактов. Далее следует матрица смежности, в которой 0 означает отсутствие возможности соединения фактов, а число отличное от нуля означает стоимость соединения. В выходном файле «output.txt» должны содержаться соединения, которые указываются номерами фактов. Нумерация фактов начинается с 0. Для решения задачи использовать алгоритм Крускала.

30. Задано N точек на плоскости и требуется проложить между ними граф путей, так, чтобы минимизировать сумму длин путей. Для решения задачи использовать алгоритм Крускала.

Теоретические сведения

Минимальные покрывающие деревья. Пусть даны n контактов на печатной плате, которые мы хотим электрически соединить. Для этого достаточно использовать n -1 проводов, каждый из которых соединяет два контакта. При этом мы обычно стремимся сделать суммарную длину проводов как можно меньше.

Упрощая ситуацию, можно сформулировать задачу так. Пусть имеется связный неориентированный граф G = (V, E), в котором V — множество контактов, а E — множество их возможных попарных соединений. Для каждого ребра графа (u,v) задан вес w(u, v) (длина провода, необходимого для соединения u и v). Задача состоит в нахождении подмножества T $\subset$ E, связывающего все вершины, для которого суммарный вес

$$w(T)= \sum_{(u,v)\in T} w(u,v)$$

минимален. Такое подмножество T будет деревом (поскольку не имеет циклов: в любом цикле один из проводов можно удалить, не нарушая связности). Связный подграф графа G, являющийся деревом и содержащий все его вершины, называют покрывающим деревом (spanning tree) этого графа. (Иногда используют термин "остовое дерево"; для краткости мы будем говорить просто "остов".)

Далее мы рассмотрим задачу о минимальном покрывающем дереве (minimum-spanning-tree problem). Здесь слово "минимальное" означает "имеющее минимально возможный вес".

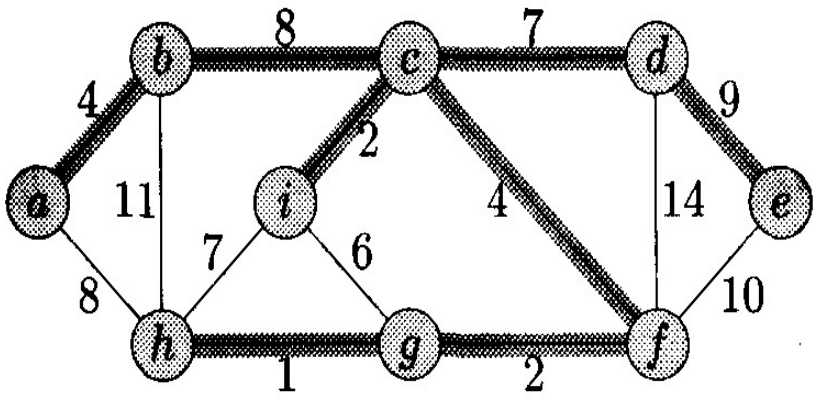


Рисунок 3.1 – Минимальное покрывающее дерево

\

На рис 3.1 показан пример минимального покрывающего дерева. На каждом ребре графа указан вес. Выделены ребра минимального покрывающего дерева (суммарный вес 37). Такое дерево не единственно: заменяя ребро (b, c) ребром (a, h), получаем другое дерево того же веса 37.

Рассмотрим два способа решения задачи о минимальном покрывающем дереве: алгоритмы Крускала и Прима. Каждый их них легко реализовать со временем работы O(E log2V), используя обычные двоичные кучи. Применив фибоначчиевы кучи, можно сократить время работы алгоритма Прима до O(E+V log2V) (что меньше E log2V, если |V| много меньше |E|).

Оба алгоритма следуют "жадной" стратегии: на каждом шаге выбирается "локально наилучший" вариант. Не для всех задач такой выбор приведёт к оптимальному решению, но для задачи о покрывающем дереве это так. Здесь будет описана общая схема алгоритма построения минимального остова (добавление рёбер одного за другим.

Построение минимального остова. Пусть дан связный неориентированный графа G = (V, E) и весовая функцией w : E $\rightarrow R$.Будем искать минимальное покрывающее дерево (остов), следуя жадной стратегии.

Общая схема алгоритма. Искомый остов строится постепенно: к изначально пустому множеству A на каждом шаге добавляется одно ребро. Множество A всегда является подмножеством некоторого минимального остова. Ребро (u, v), добавляемое на очередном шаге, выбирается так, чтобы не нарушить этого свойства: A $\cup$ {(u, v)} тоже должно быть подмножеством минимального остова. Мы называем такое ребро безопасным ребром (safe edge) для A.

Generic-MST(G,w)
1 A$\leftarrow \varnothing$
2 while A не является остовом
3 do найти безопасное ребро (u,v) для A
4 A $\leftarrow$ A $\cup$ {(u,v)}
5 return A

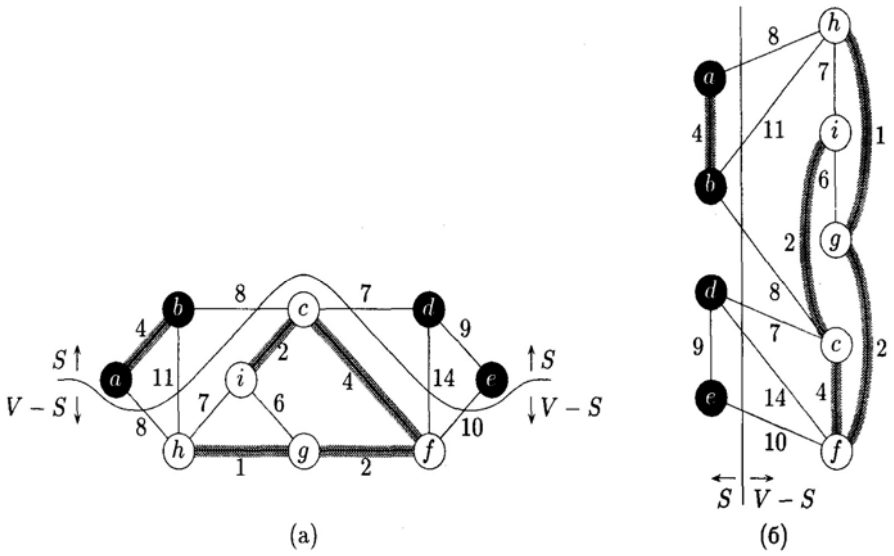


Рисунок 3.2 – Два изображения одного и того же разреза графа с Рис. 3.1

На рисунке 3.2 (а) вершины множества S изображены чёрными, его дополнения V\S — белым. Рёбра, пересекающие разрез, соединяют белые вершины с черными. Единственное лёгкое ребро, пересекающее разрез — ребро (d, c). Множество A состоит из серых ребер. Разрез (s, V \S) согласован с A (ни одно ребро из A не пересекает разрез).

На рисунке 3.2 (б) вершины множества S изображены слева, вершины V \ S — справа. Ребро пересекает разрез, если оно пересекает вертикальную прямую.

По определению безопасного ребра свойство "A является подмножеством некоторого минимального остова" (для пустого множества это свойство, очевидно, выполнено) остаётся истинным для любого числа итераций цикла, так что в строке 5 алгоритм выдаёт минимальный остов. Конечно, главный вопрос в том, как искать безопасное ребро в строке 3. Такое ребро существует (если A является подмножеством минимального остова, то любое ребро этого остова, не входящее в A, является безопасным). Заметим, что множество A не может содержать циклов (поскольку является частью минимального остова). Поэтому добавляемое в строке 4 ребро соединяет различные компоненты графа Ga = (V,A), и с каждой итерацией цикла число компонент уменьшается на 1. Вначале каждая точка представляет собой отдель-

ную компоненту; в конце весь остов — одна компонента, так что цикл повторяется |V| — 1 раз.

Приведем правило отыскания безопасных ребер (теорема 3.1). Начнём с определений.

Разрезом (cut) (S,V \S) неориентированного графа G = (V, E) называется разбиение множества его вершин на два подмножества (рис. 3.2). Говорят, что ребро (u, v) $\in$ E пересекает (crosses) разрез (S, V \ S), если один из его концов лежит в S, а другой — в V\S. Разрез согласован с множеством рёбер A (respects the set A), если ни одно ребро из A не пересекает этот разрез. Среди пересекающих разрез рёбер выделяют рёбра наименьшего веса (среди пересекающих этот разрез), называя их лёгкими (light edges).

Теорема 3.1. Пусть G = (V, E) — связный неориентированный граф, на множестве вершин которого определена вещественная функция w. Пусть A — множество рёбер, являющееся подмножеством некоторого минимального остова графа G. Пусть (S, V \ S) — разрез графа G, согласованный с A, а (u, v) — лёгкое ребро для этого разреза. Тогда ребро (u, v) является безопасным для A.

Вершины S — чёрные, вершины V \S — белые. Изображены только рёбра минимального остова (назовём его T). Рёбра множества A выделены серым цветом; (u, v) — лёгкое ребро, пересекающее разрез (S, V \ S); (x, y) — ребро единственного пути p от u к v в T.

Доказательство:

Пусть T — минимальный остов, содержащий A. Предположим, что T не содержит ребра (u,v), поскольку в противном случае доказываемое утверждение очевидно. Покажем, что в этом случае существует другой минимальный остов T', содержащий ребро (u, v), так что это ребро является безопасным. Остов T связный и потому содержит некоторый путь p из u в v (рис. 3.3).

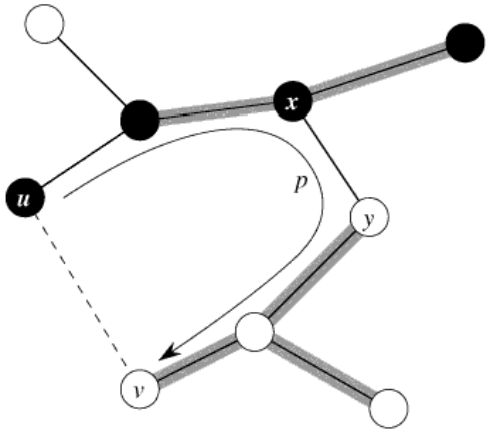


Рисунок 3.3 – Схема к доказательству теоремы 3.1

Поскольку вершины u и v принадлежат разным частям разреза (S, V \ S), в пути p есть по крайней мере одно ребро (x,y), пересекающее разрез. Это ребро не лежит в A, так как разрез согласован с A. Удаление из дерева T ребра (x, y) разбивает его на две компоненты. Добавление (u,v) объединяет эти компоненты в новый остов

T'= T\ {(x, y)}U{(u, v)}. Покажем, что T' — минимальный остов. Поскольку (u, v) — лёгкое ребро, пересекающее разрез (S, V \ S), изъятое из T ребро (x, y) имеет не меньший вес, чем добавленное вместо него ребро (u, v), так что вес остова мог только уменьшиться. Но остов был минимальным, значит, вес его остался прежним, и новый остов T' будет другим минимальным остовом (того же веса). Поэтому ребро (u, v), содержащееся в T', является безопасным.

Следующее следствие теоремы 3.1 будет использовано далее.

Следствие. Пусть G = (V, E) — связный неориентированный граф и на множестве E определена вещественная функция w. Пусть A — множество рёбер графа, являющееся под-

множеством некоторого минимального остова. Рассмотрим лес Ga = (V,A). Пусть дерево C — одна из связных компонент леса Ga. Рассмотрим все рёбра графа, соединяющие вершины из C с вершинами не из C, и возьмём среди них ребро наименьшего веса. Тогда это ребро безопасно для A. Очевидно: разрез (C, V \ C) согласован с A, а ребро (u, v) — лёгкое ребро для этого разреза.

Доказательство. Очевидно: разрез (C,V \ C) согласован с A, а ребро(u, v) — лёгкое ребро для этого разреза.

Описание алгоритмов нахождения min покрывающего дерева. Рассмотрим три часто используемых алгоритма.

Алгоритмы Крускала и Прима. Оба этих алгоритма следуют описанной схеме, но по-разному выбирают безопасное ребро. В алгоритме Крускала множество рёбер A представляет собой лес, состоящий из нескольких связных компонент (деревьев). Добавляется ребро минимального веса среди всех рёбер, концы которых лежат в разных компонентах. В алгоритме Прима множество A представляет собой одно дерево, Безопасное ребро, добавляемое к A, выбирается как ребро наименьшего веса, соединяющее это уже построенное дерево с некоторой новой вершиной.

Алгоритм Крускала. В любой момент работы алгоритма Крускала множество A выбранных рёбер (часть будущего остова) не содержит циклов. Оно соединяет вершины графа в несколько связных компонент, каждая из которых является деревом. Среди всех рёбер, соединяющих вершины из разных компонент, берётся ребро наименьшего веса. Надо проверить, что оно является безопасным.

Пусть (u, v) — такое ребро, соединяющее вершины из компонент C_1 и C_2. Это ребро является лёгким ребром для разреза $(C_1, V \setminus C_1)$, и можно воспользоваться теоремой 4 (или её следствием).

Реализация алгоритма Крускала использует структуры данных для непересекающихся множеств. Элементами множеств являются вершины графа. Напомним, что Find-Set(u) возвращает представителя множества, содержащего элемент u. Две вершины u и v принадлежат одному множеству (компоненте), если Find-Set(u) = Find-Set(v). Объединение деревьев выполняется процедурой Union. (Строго говоря, процедурам Find-Set и Union должны передаваться указатели на u и v)

MST-Kruskal(G,w)
1 $A \leftarrow \varnothing$
2 **for** каждой вершины $v \in V[G]$
3 **do** Make-Set(v)
4 упорядочить рёбра E по весам
5 **for** $(u,v) \in E$ (в порядке возрастания веса)
6 **do if** Find-Set$(u) \neq$ Find-Set(v)
7 **then** $A := A \cup \{(u,v)\}$
8 Union(u,v)
9 **return** A

На Рис 3.4 показана работа алгоритма. Сначала (строки 1-3) множество A пусто, и есть |V| деревьев, каждое из которых содержит по одной вершине. В строке 4 рёбра из E упорядочиваются по неубыванию веса. В цикле (строки 5-8) проверяется, лежат ли концы ребра в одном дереве. Если да, то ребро нельзя добавить к лесу (не создавая цикла), и оно отбрасывается. Если нет, то ребро добавляется к A (строка 7), и два соединённых им дерева объединяются в одно (строка 8).

Подсчитаем время работы алгоритма Крускала. Будем считать, что для хранения непересекающихся множеств используется метод с объединением по рангу и сжатием путей — самый быстрый из известных. Инициализация занимает время $O(V)$, упорядочение рёбер в

строке 4 — O*(E log₂E)*. Далее производится O*(E)* операций, в совокупности занимающих время O*(E* α *(E,V))*. (основное время уходит на сортировку). Рёбра растущего леса (A) выделены серым цветом. Рёбра рассматриваются в порядке неубывания весов (текущее ребро показано стрелкой). Если ребро соединяет два различных дерева, оно добавляется к лесу, а деревья сливаются.

Алгоритм Прима. Как и алгоритм Крускала, алгоритм Прима следует общей схеме алгоритма построения минимального остова. В этом алгоритме растущая часть остова представляет собой дерево (множество рёбер которого есть *A)*.

Как показано на рис 3.5, формирование дерева в алгоритме Прима начинается с произвольной корневой вершины *r*. На каждом шаге добавляется ребро наименьшего веса среди рёбер соединяющих вершины этого дерева с вершинами не из дерева. По следствию такие рёбра являются безопасными для *A,* так что в результате получается минимальный остов.

При реализации важно быстро выбирать лёгкое ребро. Алгоритм получает на вход связный граф *G* и корень *r* минимального покрывающего дерева. В ходе алгоритма все вершины, ещё не попавшие в дерево, хранятся в очереди с приоритетами. Приоритет вершины *v* определяется значением *key[u]*, которое равно минимальному весу рёбер, соединяющих *v* с вершинами дерева *A*. (Если таких рёбер нет, полагаем *key[V]* = ∞). Поле π [*v*] для вершин дерева указывает на родителя, а для вершины *v* $\in$ *Q* указывает на вершину дерева, в которую ведёт ребро веса *key[v]* (одно из таких рёбер, если их несколько).

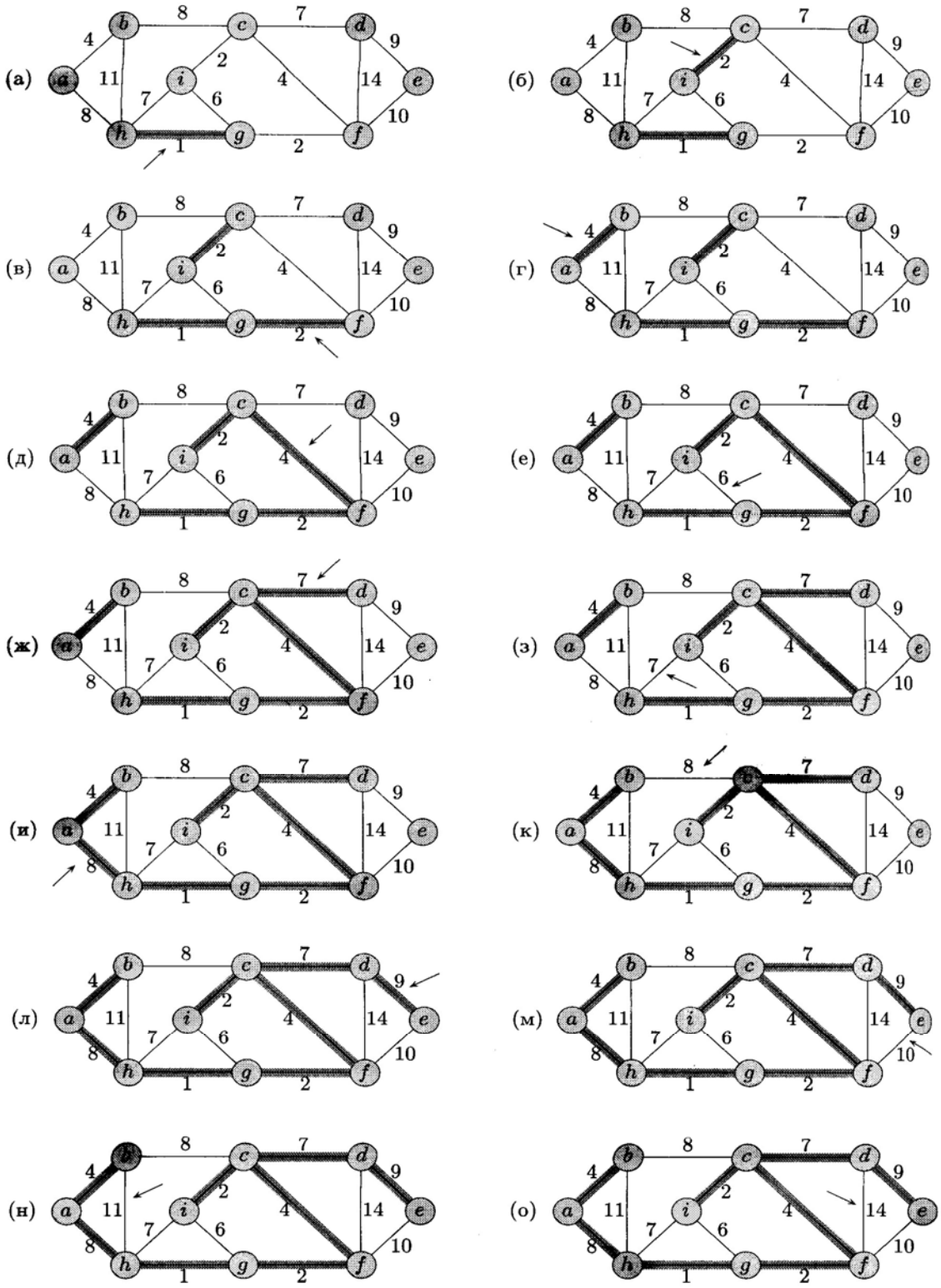


Рисунок 3.4 – Работа алгоритма Крускала на графе рис. 3.1.

Следует отметить, что в рассматриваемом алгоритме не хранится множество A вершин строящегося дерева явно; его можно восстановить как

$$A = \{(v, \pi[v]) : v \in V \setminus \{r\} \setminus Q\}.$$

В конце работы алгоритма очередь Q пуста, и множество ребер покрывающего дерева это множество A:

$$A = \{(v, \pi[v]) : v \in V \setminus \{r\}\}.$$

MST-Prim*(G,W,r)*

1 $Q \leftarrow V[G]$
2 **for** для каждой вершины $u \in Q$
3 **do** $key[u] \leftarrow \infty$
4 $key[r] \leftarrow 0$
5 π [r] $\leftarrow$ nil
6 **while** $Q \neq \varnothing$
7 **do** $u \leftarrow$ Extract-Min(Q)
8 **for** для каждой вершины $v \in Adj[u]$
9 **do if** $v \in Q$ и $w(u,v)<key[v]$
10 **then** $\pi [v] \leftarrow u$
11 $key(v) \leftarrow w(u,v)$

Рёбра, входящие в дерево A. выделены серым; вершины дерева — чёрным. На каждом шаге к A добавляется ребро, пересекающее разрез между деревом и его дополнением. Например, на втором шаге можно было бы добавить любое из рёбер (b, c) и (a,h).

На рис. 3.5 показана работа алгоритма Прима. После исполнения строк 1-5 и первого прохода цикла в строках 6 - 11 дерево состоит из единственной вершины r, все остальные вершины находятся в очереди, и значение key[v] для них равно весу ребра из r в v или ∞, если такого ребра нет(в первом случае π [v] = r). Таким образом, выполнен описанный выше инвариант (дерево есть часть некоторого остова, для вершин дерева поле π указывает на родителя, а для остальных вершин на "ближайшую" вершину дерева — вес ребра до неё хранится в key[v].

Время работы алгоритма Прима зависит от того, как реализована очередь Q. Время инициализации в строках 1-4 O(V).Далее цикл выполняется |V| раз, и каждая операция Extract-Min занимает время O(log2V), всего O(V log2V). Цикл for в строках 8-11 выполняется в общей сложности O(E) раз, поскольку сумма степеней вершин графа равна 2 |E|. Проверку принадлежности в строке 9 внутри цикла for можно реализовать за время O(1). Таким образом, всего получаем O(V log2V + E log2V) = O(E log2V) — та же самая оценка, что была для алгоритма Крускала.

Алгоритм Борувки. Третий подход к построению остова — алгоритм Борувки (Boruvka). На первом шаге к минимальному покрывающему -дереву добавляются ребра, которые соединяют каждую вершину с ее ближайшим соседом. Если веса ребер различны, этот шаг порождает лес из связных поддеревьев (мы докажем этот факт и рассмотрим усовершенствование, которое позволяет работать даже при наличии ребер с равными весами). Затем мы добавляем ребра, которые соединяют каждую вершину с ее ближайшим соседом (минимальное ребро, соединяющее вершину одного дерева с вершиной другого), и повторяем этот процесс, пока не останется только одно дерево.

Лемма 3.1. Алгоритм Борувки вычисляет минимальное остовное дерево для любого связного графа.

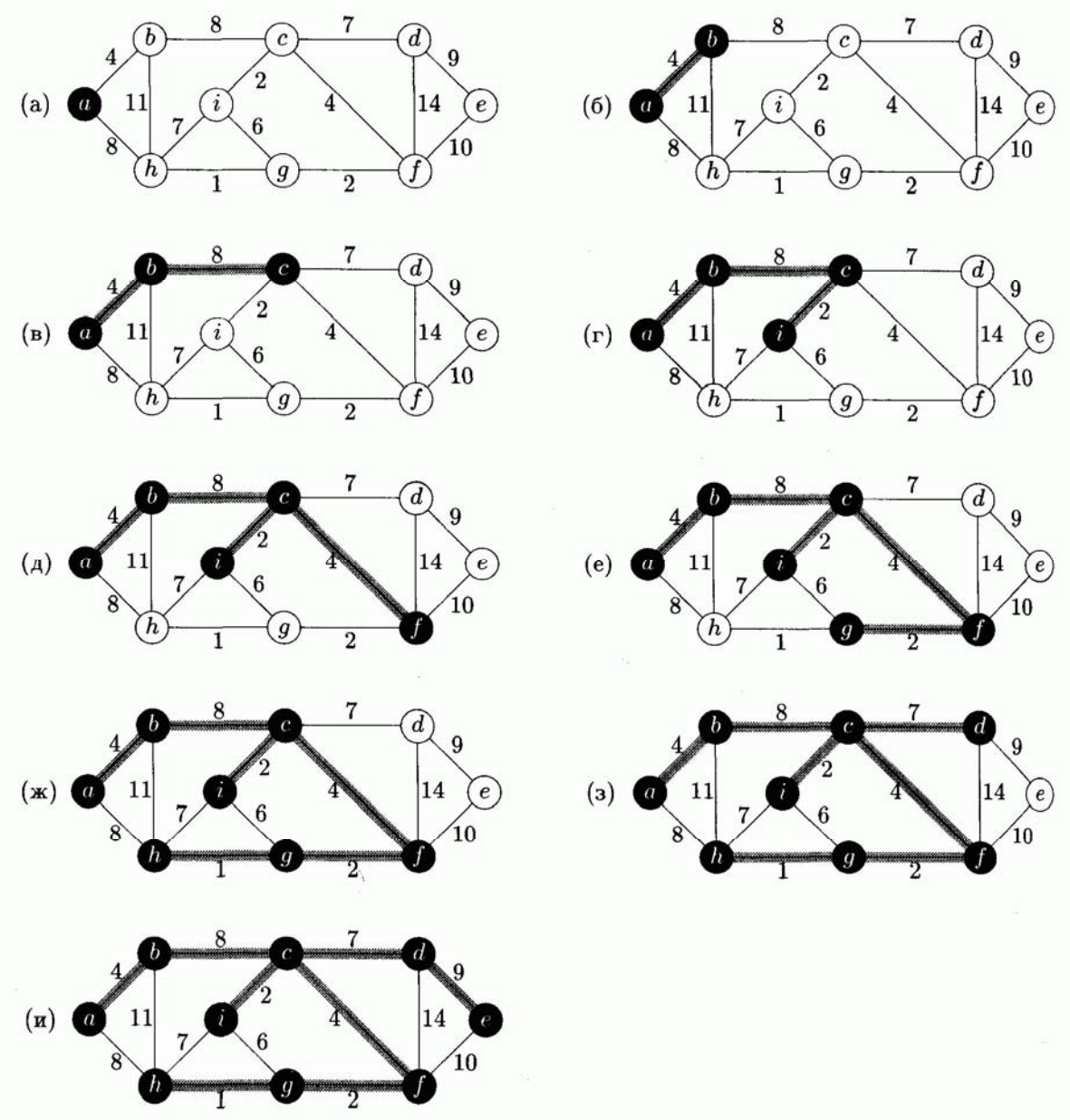


Рисунок 3.5 – Работа алгоритма Прима на графе рис. 3.1
с корневой вершиной a.

Сначала предположим, что веса всех ребер различны. В этом случае у каждой вершины имеется единственный ближайший сосед, минимальное остовное дерево уникально, и мы знаем, что каждое добавленное ребро — это ребро минимального остовного дерева (согласно свойству сечения это самое короткое ребро, пересекающее сечение из рассматриваемой вершины во все остальные вершины). Поскольку каждое ребро выбрано из уникального минимального остовного дерева, циклов не может быть, каждое добавленное ребро соединяет два дерева из леса в большее дерево, и процесс продолжается, пока не останется единственное дерево — минимальное остовное дерево.

При наличии одинаковых ребер ближайших соседей может быть несколько, и при добавлении ребра к ближайшему соседу возможно появление цикла (см. рис. 3.6). Другими словами, мы можем выбрать для некоторой вершины два ребра из множества минимальных секущих ребер, хотя дереву принадлежит лишь одно. Чтобы избежать подобных ситуаций, необходимо подходящее правило разрыва связей. Одно из них — выбор из множества минимальных соседей вершины с наименьшим номером. Тогда любой цикл приводит к противоречию: если v — вершина с наибольшим номером в цикле, то ни одна из вершин, соседних с

v, не выберет ее в качестве ближайшего соседа, и вершина v должна выбрать только одного из своих соседей с минимальным номером.

В графе с четырьмя вершинами(рис. 3.6) все четыре ребра имеют одинаковую длину. Перед соединением каждой вершины с ближайшим соседом необходимо решить, какое ребро выбрать из множества минимальных ребер. В верхнем примере мы выбираем 1 из вершины 0, 2 из 1, 3 из 2 и 0 из 3, что приводит к образованию цикла в заготовке дерева. Каждое из ребер входит в некоторое дерево, но не все они входят в каждое дерево. Поэтому мы используем правило разрыва связей (внизу): выбираем минимальное ребро в вершину с наименьшим индексом. Тогда из 1 мы выбираем 0, 0 из 1, 1 из 2 и 0 из 3, что и дает остовное дерево. Цикл разорван, т.к. вершина с максимальным индексом 3 не выбрана ни из одного из ее соседей (2 или 1), а она может выбрать только одного из них (0).

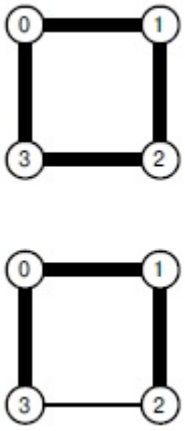


Рисунок 3.6 –Циклы в алгоритме Борувки

Все три рассматриваемых алгоритма можно свести к одному обобщенному алгоритму. Этот алгоритм начинается с выбора леса поддеревьев, состоящих из одиночных вершин (и не содержащих ребер), затем выполняется шаг добавления минимального ребра, соединяющего два любых поддерева леса, и эти шаги повторяются V— 1 раз, пока не останется единственное дерево. Согласно свойству сечения, ни одно из ребер, порождающих цикл, не стоит рассматривать в качестве кандидата на включение в дерево, поскольку ранее одно из ребер уже было минимальным ребром, пересекающим некоторое сечение между MST-поддеревьями, содержащими его вершины. В алгоритме Прима ребра добавляются по одному к единственному дереву; алгоритмы Крускала и Борувки объединяют деревья в лесе.

Вначале мы строим вектор, индексированный именами вершин, который для каждого MST-поддерева определяет ближайшего соседа. Затем для каждого ребра графа мы выполняем следующие операции. Если оно соединяет две вершины одного и того же дерева, отбрасываем его; иначе проверяем расстояния между вершинами деревьев, которые соединяет это ребро и при необходимости изменяем их.

На верхней диаграмме (рис. 3.7) показаны ориентированные ребра, проведенные из каждой вершины к ближайшему соседу. Из этой диаграммы следует, что ребра 0-2, 1-7 и 3-5 являются самыми короткими ребрами, инцидентными обеим их вершинам, 6-7 — кратчайшее ребро для вершины 6, а 4-3 — кратчайшее ребро для вершины 4. Все эти ребра принадлежат минимальному остову и образуют лес минимальных остовных поддеревьев (в центре), вычисляемый на первом этапе выполнения алгоритма Борувки. На втором этапе алгоритм завершает вычисление минимальных остовных поддеревьев (внизу). Для этого добавляется ребро 0-7 — кратчайшее ребро, инцидентное каждой вершине тех поддеревьев, которые оно соединяет; и ребра 4-7 — кратчайшее ребро, инцидентное каждой вершине нижнего поддерева.

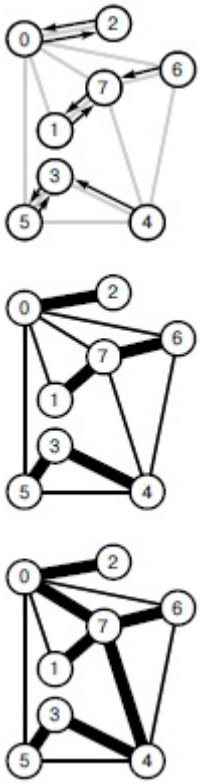


Рисунок 3.7 – Алгоритм Борувки нахождения минимального остовного дерева

Алгоритм Боровки можно описать следующим образом.

1. Изначально, пусть T — пустое множество рёбер (представляющее собой остовный лес, в который каждая вершина входит в качестве отдельного дерева).

2. Пока T не является деревом (что эквивалентно условию: пока число рёбер в T меньше, чем V-1, где V— число вершин в графе):

- Для каждой компоненты связности (то есть, дерева в остовном лесе) в подграфе с рёбрами T, найдём самое легкое ребро, связывающее эту компоненту с некоторой другой компонентой связности. (Предполагается, что веса рёбер различны, или как-то дополнительно упорядочены так, чтобы всегда можно было найти единственное ребро с минимальным весом).
- Добавим все найденные рёбра в множество T.

3. Полученное множество рёбер T является минимальным остовным деревом входного графа.

Сложность алгоритма. На каждой итерации число деревьев в остовном лесу уменьшается по крайней мере в два раза, поэтому всего алгоритм совершает не более O(log V) итераций. Каждая итерация может быть реализована со сложностью O(E), поэтому общее время работы алгоритмы составляет O(E log V) времени (здесь V и E— число вершин и рёбер в графе, соответственно).

Пример текста программы на построение каркасов

```
#include "stdafx.h"
#include "graph.h"
#include <iostream>
#include <fstream>
#include <string>
#include <stdlib.h>
using namespace std;
#define n 10
```

```
class graph
{
private:
	int a[n][n]; //Матрица смежности.
	int num_ver; //Количество вершин.
	int fl; //указатель создания файлов
	int mark[n];  //компоненты вершин при поиске минимального каркаса
	int numb;  //текущее количество рёбер в остовном дереве
	int t[n][n]; //матрица остовного дерева
	int up;   //верхний и
	int down; //нижний указатель очереди при поиске всех остовных деревьев
	int tr[n]; //очередь для вершин
	bool vernew[n]; //массив посещенных вершин
	int kol_os_gr; // количество остовных графов
	int weight; //вес минимального каркаса
	int weight_os; //вес остовного каркаса
public:
	graph(void);
	void init_ver( int num ); //Инициализация количества вершин.
	void init_graph( int gr[][n] );  //Инициализация матрицы смежности графа.
	void change_mark ( int l, int m );
	void tree();   //поиск минимального каркаса
	void solve( int v, int q ); //поиск всех остовных деревьев
	void print_mat();   //Вывод матрицы
	int GetKol();
	~graph(void);
};
graph::graph(void)
{
	kol_os_gr = 0;
	tr[0] = 0;
	down = 0;
	up = 1;
	numb = 0;
	weight = 0;
	weight_os = 0;
	memset(t, 0, sizeof(t)); //Обнуление массивов.
	memset(tr, 0, sizeof(tr));
	memset(vernew, true, sizeof(vernew));
	vernew[0] = false;
	fl = 1;
}
void graph::init_ver ( int num )
{
	num_ver = num;
}
void graph::init_graph( int gr[][n] )
{
	int i, j;
	for ( i = 0; i < num_ver; i ++ )
		for ( j = 0; j < num_ver; j ++ )
			a[i][j] = gr[i][j];
```

```
}
void graph::change_mark ( int l, int m )
{
	int i, t;
	if ( m < l )
	{
		t = l;
		l = m;
		m = t;
	}
	for ( i = 0; i < num_ver; i ++ )
		if ( mark[i] == m )
			mark[i] = l;
}
void graph::tree()
{
	int i, j, kolreb = 0;
	for ( i = 1; i < num_ver; i ++ )
		for ( j = 0; j < i; j ++ )
			if ( a[i][j] != 0 )
				kolreb ++;
	int **h; //массив уворядоченных ребёр
	h = new int* [kolreb];
	for ( i = 0; i < kolreb; i ++ )
		h[i] = new int [3];
		int b = 0;
	for ( i = 1; i < num_ver ; i ++ )
		for ( j = 0; j < i; j ++ )
			if ( a[i][j] != 0 )
			{
				h[b][0] = i;
				h[b][1] = j;
				h[b][2] = a[i][j];
				b ++;
			}
	//----сортировка ребер по возрастанию веса
	int f, d, s, v;
	for ( i = 0; i < kolreb - 1; i ++ )
		for ( j = 0; j < kolreb - 1; j ++ )
			if ( h[j][2] > h[j + 1][2] )
			{
				f = h[j][2];
				h[j][2] = h[j+1][2];
				h[j+1][2] = f;
				d = h[j][0];
				h[j][0] = h[j+1][0];
				h[j+1][0] = d;
				s = h[j][1];
				h[j][1] = h[j+1][1];
				h[j+1][1] = s;
			}
	//поиск минимального каркаса
```

```
	for ( i = 2; i < num_ver; i ++ )
		mark[i] = i;
	s = 0;
	d = kolreb;
	while ( s < num_ver - 1 )
	{
		i = 0;
		while ( i < d && mark[h[i][0]] == mark[h[i][1]] )
			i ++;
		s ++;
		weight += h[i][2];
		change_mark (mark[h[i][0]], mark[h[i][1]]);
	}
	cout << "Min weight: " << weight << "." << endl;
}
void graph::solve( int v, int q )   //v - номер вершины, из которой выходит ребро, q - номер вершины, начиная с которой следует искать очередное ребро каркаса
{
	int j;
	char s[5];
	string name = "file";
	string name1 = ".dot";
	if ( down >= up )
		return;
	j = q;
		while ( j < num_ver && numb < num_ver - 1 )
	{
		/*Просмотр ребер, выходящих из вершины с номером v.*/
		if ( a[v][j] != 0 && vernew[j] ) //Есть ребро, и вершины с номером j еще нет в каркасе. Включаем ребро в каркас.
		{
			vernew[j] = false;
			numb ++;
			t[v][j] = t[j][v] = a[v][j];
			weight_os += t[j][v];
			tr[up] = j;
			up ++; //Включаем вершину с номером j в очередь
			solve ( v, j ); //Продолжаем построение каркаса.
			up --;
			vernew[j] = true;
			weight_os -= t[j][v];
			t[v][j] = t[j][v] = 0;
			numb --; //Исключаем ребро из каркаса.
		}
		j ++;
	}
	if ( numb == num_ver - 1 )
	{
		if ( weight == weight_os )
		{
			kol_os_gr ++;
			itoa(fl, s, 10);
```

```
                        ofstream fout(name+s+name1);
                        if (fout.is_open())
                        {
                                fl ++;
                                fout << "graph G {" << endl;
                                for ( int i = 0; i < num_ver; i ++ )
                                        for ( int k = i; k < num_ver; k ++ )
                                                if ( t[i][k] != 0 )
                                                        fout << i + 1 << " -- " << k + 1 << " [label = \""
<< t[i][k] << "\"];" << endl;
                                fout << "}";
                        }
                        else
                        {
                                cout << "Error!" << endl;
                                system ("pause");
                        }
                        fout.close();
                }
                return;
        }
        /*Все ребра, выходящие из вершины с номером v, просмотрены. Переходим к сле-
дующей вершине
        из очереди и так до тех пор, пока не будет построен каркас.*/
        if ( j == num_ver )
        {
                down ++;
                solve ( tr[down], 0 );
                down --;
        }
}
void graph::print_mat()
{
        int i, j;
        cout << "The matrix of graph: " << endl;
        for ( i = 0; i < num_ver; i ++ )
        {
                for ( j = 0; j < num_ver; j ++ )
                        cout << a[i][j] << " ";
                cout << endl;
        }
        cout << "The list of adjacent vertices: " << endl;
        for ( i = 0; i < num_ver; i ++ )
        {
                cout << i + 1 << ": ";
                for ( j = 0; j < num_ver; j ++ )
                        if ( a[i][j] > 0 )
                                cout << j + 1 << "(" << a[i][j] << ") ";
                cout << endl;
        }
        /*Ввод графа в файл.*/
        ofstream fout("file.dot");
```

```
	if (fout.is_open())
	{
		fout << "graph G {" << endl;
		for ( i = 0; i < num_ver; i ++ )
			for ( j = i; j < num_ver; j ++ )
				if ( a[i][j] != 0 )
					fout << i + 1 << " -- " << j + 1 << " [label = \"" << a[i][j] <<
"\"];" << endl;
		fout << "}";
	}
	else
	{
		cout << "Error!" << endl;
		system ("pause");
	}
	fout.close();
}
int graph::GetKol()
{
	return kol_os_gr;
}
graph::~graph(void)
{
}
int main()
{
	int ent; //Способ ввода.
	int num_ver;   //Количество вершин.
	int gr[n][n];   //Массив для ввода.
	memset(gr, 0, sizeof(gr)); //Обнуление массива.
	graph Graph1;
	cout << "Hello!\n";
	do
	{
		cout << "File(1) or keyboard(2)? ";
		cin >> ent;
	}
	while ( ent < 1 || ent > 2 );
	//Ввод списка рёбер с клавиатуры.
	if ( ent == 2 )
	{
		int ver1, ver2; //Вершины ребра.
		int weight; //вес ребра
		int id = 0;
		int f;
		do
		{
			cout << "Enter the number of vertices of the graph. ";
			cin >> num_ver;
		}
		while ( num_ver < 1 || num_ver > 10 );
		Graph1.init_ver( num_ver );
```

```
            cout << "Enter the edge between nodes and the weight of edge (for ex., 8 4 10 or 1 2
1): ";
            do
            {
                  do
                  {
                        f = true;
                        cin >> ver1 >> ver2 >> weight;
                        if ( ver1 > num_ver || ver2 > num_ver || ver1 < 1 || ver2 < 1 || weight <
1 || weight > 2515 )
                              f = false;
                        if ( f )
                              gr[ver1-1][ver2-1] = gr[ver2-1][ver1-1] = weight;
                        if ( !f )
                              cout << "Vertices are entered incorrectly. Try again." << endl;
                  }
                  while ( !f );
                  do
                  {
                        cout << "Will be more ribs? Yes(1) or no(2). ";
                        cin >> id;
                  }
                  while ( id < 1 || id > 2 );
            }
            while ( id == 1 );
      }
      //Ввод списка рёбер из файла.
      if ( ent == 1 )
      {
            char k[10]; //Для считывания из файла.
            int ver1, ver2; //Вершины ребра.
            int weight; //вес ребра
            int f;
            ifstream fin("tfile.txt");
            if (fin.is_open())
            {
                  fin >> k;
                  num_ver = atoi(k);
                  if ( num_ver < 1 || num_ver > 10 )
                  {
                        cout << "Error in the file. Please correct to continue.\n";
                        system ("pause");
                        return 2;
                  }
                  Graph1.init_ver( num_ver );
                  do
                  {
                        f = true;
                        fin >> ver1 >> ver2 >> weight;
                        if ( ver1 > num_ver || ver2 > num_ver || ver1 < 1 || ver2 < 1 || weight <
1 || weight > 1000 )
                              f = false;
```

```
                        if ( f )
                                gr[ver1-1][ver2-1] = gr[ver2-1][ver1-1] = weight;
                }
                while ( !fin.eof() );
        }
        else
        {
                cout << "Error!" << endl;
                system ("pause");
                return 1;
        }
        fin.close();
    }
    Graph1.init_graph(gr);
    Graph1.print_mat();
    Graph1.tree();
    Graph1.solve( 0, 0 );
    cout << "The number of minimal spanning graphs: " << Graph1.GetKol() << "." << endl;
    system ("pause");
    return 0;
}
```

Рекомендуемые источники по теме 3

1 Кристофидес Н. Теория графов. Алгоритмический подход. – М.: Мир, 1978. – 432 с.
2 https://ru.wikipedia.org/wiki/Алгоритм_Борувки - дата съема информации 01.02.2018

4 Алгоритмы нахождения циклов в графах и их программная реализация

Лабораторная работа № 4
(работа в аудитории 4 ч., СРС 4 ч.)

Цели работы

– закрепление теоретических знаний по теме работы;

– углубление навыков по применению методов разработки программного обеспечения;

– углубление навыков оценки временной и емкостной сложности программного обеспечения;

– углубление навыков чтения, понимания и выделения главной идеи прочитанного исходного кода;

– приобретение практических навыков в разработке компьютерных программ, реализующих алгоритмы нахождения циклов в графах.

Задания

Перед выполнением работы необходимо повторить теоретический материал по теме. Основные теоретические положения изложены ниже в данном параграфе пункте «Теоретические сведения». После этого следует ответить на тестовые вопросы по теме.

В результате выполнения лабораторной работы требуется написать и отладить программу на алгоритмическом языке (C++ или ином), которая получает на входе числовые данные, выполняет их обработку в соответствии с требованиями задания и выводит резуль-

тат на экран в графической форме. Для обработки данных необходимо реализовать алгоритмы в соответствии с постановкой задачи. Ввод данных осуществляется из файла с учетом требований к входным данным, с учетом постановки задачи. Ограничениями на входные данные является допустимый диапазон значений используемых числовых типов в выбранной системе программирования. Работа состоит из двух указанных ниже этапов.

Этап 1. Выполнить приведенные ниже задания (работа в аудитории и СРС).

1 Выбрать вариант алгоритм решения поставленной задачи (варианты алгоритмов приведены ниже) или получить вариант от преподавателя.

2 Разработать тестовые примеры для отладки соответствующего алгоритма и программы. Согласовать с преподавателем.

3 Разработать и обосновать структуры данных для реализации алгоритма.

4 С учетом выполнения предыдущих пунктов задания написать и отладить программу на множестве тестовых примеров.

5 Оформить отчет по выполненной работе.

6 Защитить работу, ответив на вопросы преподавателя и студентов.

Этап 2. Выполнить приведенные ниже задания (работа в аудитории с использованием интерактивных технологий обучения):

4. Разделиться на команды по 2-3 человека в команде

5. Каждая команда выбирает один из отчетов, оформленный членом данной команды для передачи отчета другой команде.

6. Проанализировать полученный от другой команды отчет; прочитать исходный код; протестировать соответствующую программу на своих тестах; подготовить отзыв, осветив следующие вопросы по рецензируемому отчету:

– идея алгоритма;

– корректность выбора структур данных;

– полнота и качество реализации алгоритма в программе;

– грамотность оценки временной и емкостной сложности программы;

– качество тестов;

– полнота и качество оформления отчета в целом;

Дать итоговое заключение и оценку, указав достоинства и недостатки работы.

Результаты работы команд оцениваются преподавателем и другими командами.

Варианты алгоритмов

1. Найти самый длинный простой цикл в графе, заданном матрицей смежности.
2. Найти все простые циклы в графе, заданном матрицей смежности.
3. Найти самый короткий простой цикл во взвешенном графе. Граф задан взвешенной матрицей смежности.
4. В некоторой деревне скучают люди. Чтобы разнообразить свою жизнь они распространяют слухи. Для каждого жителя известно кому он передает слух, попавший к нему (возможно, нескольким людям). Если слух попадет обратно к тому, кто его «запустил», тогда он обрастает новыми деталями и распространяется дальше. Вам требуется определить максимальный круг людей, внутри которого циркулирует один слух.
5. Найти базовые циклы в графе, заданном списком дуг.
6. Город состоит из перекрестков и соединяющих их дорог. Требуется определить: сможет ли таксист подзаработать, провезя вас по кругу несколько раз. Найти все кольцевые дороги. Перекрестки пронумерованы. Город задается матрицей смежности, где каждая дуга – дорога, соединяющая перекрестки.
7. Найти базовые циклы в графе, заданном матрицей смежности
8. Найти эйлеров цикл в графе, заданном матрицей смежности.
9. Ребрам графа G приписаны положительные веса. Требуется найти цикл, проходящий через каждое ребро графа G по крайней мере один раз и такой, что для него общий вес

(а именно сумма величин nj*c(aj), где число nj показывает, сколько раз проходилось ребро aj, а c(aj) — вес ребра) минимален. Очевидно, что если G содержит эйлеров цикл, то любой такой цикл будет оптимальным, так как каждое ребро проходится только один раз и вес этого цикла равен тогда ∑ c(aj).

10. Рассмотрим проблему сбора домашнего мусора. Допустим, что определенный район города обслуживается единственной машиной. Ребра графа G представляют дороги, а вершины — пересечения дорог. Величина c(aj) — вес ребра — будет соответствовать длине дороги. Тогда проблема сбора мусора в данном районе сводится к нахождению цикла в графе G, проходящего по каждому ребру G по крайней мере один раз. Требуется найти цикл с наименьшим километражем. С учетом того, как на самом деле курсирует машина, возникает задача: найти Q циклов, которые вместе покроют все ребра графа и при этом их суммарная длина будет минимальной. Машина будет ездить по расписанию и в i-ый день объезжать i-ый цикл.

11. Определить маршрут, минимизирующий общий километраж и проходящий хотя бы один раз по каждой из улиц при доставке молока

12. Разработать наилучший маршрут проверки электрических линий. Проблема инспектирования распределенных систем связана с непременным требованием проверки всех "компонент".

13. Разработать наилучший маршрут проверки телефонных линий. Проблема инспектирования распределенных систем связана с непременным требованием проверки всех "компонент".

14. Разработать наилучший маршрут проверки железнодорожных линий. Проблема инспектирования распределенных систем связана с непременным требованием проверки всех "компонент".

15. Разработать наилучший маршрут для проверки работы автоматических вентиляционных устройств. Проблема инспектирования распределенных систем связана с непременным требованием проверки всех "компонент".

16. Разработать наилучший маршрут для уборки помещений и коридоров в большом учреждении.

17. Найти гамильтонов цикл в графе.

18. На планете «Cube», имеющей форму куба, города расположены в вершинах куба и в серединах граней. Любые два города, расположенные на одном ребре, соединены дорогой. Также города, расположенные в серединах граней, соединены с вершинами. Космонавт хочет совершить путешествие по этой планете, пройдя по каждой из её дорог ровно один раз. Сможет ли он это сделать?

19. Найти самый длинный простой цикл во взвешенном графе.

20. Найти все простые циклы взвешенного графа.

21. Проверить, будет ли в графе существовать путь, проходящий через все его вершины. Граф задан матрицей смежности.

22. Найти самый короткий простой цикл во взвешенном графе, заданном взвешенной матрицей смежности.

Теоретические сведения

Ознакомьтесь с теоретическим материалом по теме «Циклы графа», ответьте на тестовые вопросы по теме.

Основные определения.

Определение 1. Пусть *G(V, E)* – неориентированный граф. Рассмотрим конечную последовательность вершин такую, что любые две соседние вершины инцидентны одному ребру. Эта последовательность называется *маршрутом графа*.

Определение 2. *Маршрутом (путем)* для графа *G(V,E)* называется последовательность $v_1e_1v_2e_2v_3...e_kv_{k+1}$. Маршрут называется *замкнутым*, если его начальная и конечная

точки совпадают. Число ребер (дуг) маршрута (пути) графа называется *длиной* маршрута (пути).

Определение 3. Незамкнутый маршрут (путь) называется *цепью*. Цепь, в которой все вершины попарно различны, называется *простой цепью*.

Определение 4. Замкнутый маршрут (путь) называется *циклическим маршрутом* или *циклом (контуром***)**. Цикл, в котором все вершины попарно различны, называется *простым циклом*.

Эйлеровы пути и циклы. Первая теорема теории графов была доказана задолго до того, как стало употребляться *словосочетание* "теория графов". В 1736 г. появилась работа Эйлера, в которой не только была решена предложенная ему задача о кенигсбергских мостах, но и сформулировано *общее правило*, позволяющее решить любую задачу такого рода. Интересно, что в одном из писем Эйлер писал по этому поводу: "... это решение по своему характеру, по-видимому, имеет мало отношения к математике, и мне непонятно, почему следует скорее от математика ожидать этого решения, нежели от какого-нибудь другого человека ...".

На языке теории графов задача состоит в том, чтобы определить, имеется ли в графе *путь*, проходящий через все его ребра (*путь*, по определению, не может дважды проходить по одному ребру). Такой *путь* называется *эйлеровым путем*, а если он замкнут, то *эйлеровым циклом*. В графе, изображенном на рис. 4.1-а, *эйлеров цикл* существует - например, последовательность вершин 1, 2, 4, 5, 2, 3, 5, 6, 3, 1 образует такой цикл. В графе же на рис. 4.1-б эйлерова *цикла* нет, есть *эйлеровы пути*, например, 2, 4, 5, 2, 1, 3, 5, 6, 3.

а) *б)*

Рисунок 4.1 − Эйлеров цикл и эйлеров путь.

Рассмотрим сначала условия существования эйлерова *цикла* в обыкновенном графе. Очевидно, что в *несвязном графе эйлеров цикл* может существовать только в том случае, когда все его ребра принадлежат одной компоненте связности, а все остальные компоненты – просто *изолированные вершины*. Поэтому достаточно рассматривать связные графы.

Теорема 1. В любом маршруте, соединяющем две различные вершины, содержится простой путь, соединяющий те же вершины. В любом цикле, проходящем через некоторое ребро, содержится простой цикл, проходящий через это ребро.

Теорема 2. *Эйлеров цикл* в связном графе существует тогда и только тогда, когда в нем степени всех вершин четны.

Доказательство.

Необходимость условия очевидна, так как при каждом прохождении цикла через какую-либо вершину используются два ребра - по одному из них маршрут входит в вершину, по другому выходит из нее (это относится и к стартовой вершине - в ней ведь маршрут должен закончиться). Докажем его достаточность.

Пусть G - связный граф, в котором больше одной вершины и степени всех вершин четны. Значит, степень каждой вершины не меньше 2, поэтому в графе G имеется цикл Z_1. Если удалить все ребра этого цикла из графа G , то получится граф G_1 , в котором степени вершин также четны. Если в G_1 нет ни одного ребра, то Z_1. - *эйлеров цикл*. В противном случае, применяя теорему 1 к графу, полученному из G_1 удалением всех изолированных вершин, заключаем, что в G_1 имеется цикл Z_2. Удалив из G_1 все ребра цикла Z_2, получим

граф G_2. Продолжая действовать таким образом, пока не придем к пустому графу, получим в итоге систему циклов $Z_1, Z_2, \ldots, Z_k$, причем каждое ребро графа принадлежит в точности одному из них. Покажем теперь, что из этих циклов можно составить один цикл. Действительно, из того, что исходный граф связен, следует, что хотя бы один из циклов $Z_1, Z_2, \ldots, Z_{k-1}$ имеет общую вершину с Z_k. Допустим, для определенности, что таков цикл Z_{k-1}.

Пусть $Z_k = x_1, x_2, \ldots x_p$; , $Z_{k-1} = y_1, y_2, \ldots y_q$, и $x_i = y_j$ для некоторых i и j. Тогда последовательность вершин

$$Z'_{k-1} = x_1, x_2, \ldots x_i, y_{j+1}, y_{j+2}, \ldots y_q, y_2, \ldots y_j, x_{i+1}, x_{i+2}, \ldots x_p$$

очевидно, является циклом, а множество ребер этого цикла есть объединение множеств ребер циклов Z_{k-1} и Z_k. Таким образом, получаем систему из меньшего числа циклов, попрежнему обладающую тем свойством, что каждое ребро графа принадлежит в точности одному из них. Действуя далее таким же образом, в конце концов, получим один цикл, который и будет эйлеровым.

Теорема 2 верна и для *мультиграфов* (кстати, в задаче о кенигсбергских мостах ситуация моделируется именно *мультиграфом*). Она остается верной и при наличии петель, если при подсчете степеней вершин каждую петлю считать дважды.

Теперь нетрудно получить и *критерий существования* эйлерова пути.

Теорема 3. Эйлеров путь в связном графе существует тогда и только тогда, когда в нем имеется не более двух вершин с нечетными степенями.

Доказательство. Если в графе нет вершин с нечетными степенями, то, по предыдущей теореме, в нем имеется *эйлеров цикл*, он является и эйлеровым путем. Не может быть точно одной вершины с нечетной степенью - это следует из теоремы 2 «Начальные понятия теории графов». Если же имеются точно две вершины с нечетными степенями, то построим новый граф, добавив ребро, соединяющее эти вершины. В новом графе степени всех вершин четны и, следовательно, существует *эйлеров цикл* (возможно, что при добавлении нового ребра получатся кратные ребра, но, как отмечалось выше, теорема об эйлеровом цикле верна и для *мультиграфов*). Так как циклический сдвиг цикла - тоже цикл, существует и такой *эйлеров цикл*, в котором добавленное ребро - последнее. Удалив из этого цикла последнее ребро, получим эйлеров путь в исходном графе.

В ориентированном графе под эйлеровым путем (циклом) понимают ориентированный *путь* (цикл), проходящий через все ребра графа. Ориентированный вариант критерия существования эйлерова *цикла* формулируется следующим образом.

Теорема 4. *Эйлеров цикл* в связном орграфе существует тогда и только тогда, когда у каждой его вершины число входящих в нее ребер равно числу выходящих.

Тестовые вопросы по теме 4

1. Что такое эйлеров цикл?
2. Что такое эйлеров путь?
3. Сформулируйте условия существования эйлерова цикла в обыкновенном графе
4. Сформулируйте необходимое и достаточное условие *Эйлерова цикла* в связном графе
5. Как от вида или представления графа зависит временная сложность алгоритмов?
6. Как от вида или представления графа зависит временная сложность алгоритмов нахождения циклов?
7. Как от вида или представления графа зависит временная сложность алгоритма нахождения эйлерова цикла?
8. Как от вида или представления графа зависит временная сложность алгоритма нахождения гамильтонова цикла?
9. Существует ли цикл в несвязном графе? Ответ обоснуйте.
10. Существует ли гамильтонов цикл в несвязном графе? Ответ обоснуйте.

11. Сформулируйте идею алгоритма нахождения самого длинного цикла, которому принадлежат две данные вершины графа.

12. Сформулируйте идею алгоритма проверки графа на цикличность, если граф задан матрицей смежности.

13. Сформулируйте идею алгоритма нахождения самого длинного простого цикла графа.

14. Сформулируйте идею алгоритма нахождения фундаментального множества циклов графа.

15. Сформулируйте идею алгоритма нахождения цикла, проходящего через все его вершины. Граф задан матрицей смежности.

16. Сформулируйте идею алгоритма нахождения множества вершин графа, доступных из заданной вершины.

17. Сформулируйте идею алгоритма нахождения минимального простого цикла между двумя заданными вершинами взвешенного графа, определенного списковой структурой.

18. Сформулируйте идею алгоритма проверки графа на ацикличность.

19. Сформулируйте идею алгоритма проверки: является ли граф, заданный списком, ацикличным.

20. Как связаны между собой различные способы представления графов?

Рекомендуемые источники по теме 4

1 Кристофидес Н. Теория графов. Алгоритмический подход. – М.: Мир, 1978. – 432 с.

2 Дольников В.Л., Якимова О.П. Основные алгоритмы на графах : Текст лекций – Ярославль.: ЯрГУ, 2011. – 80 с.

5 Алгоритмы нахождения независимых множеств, клик, вершинных покрытий и их программная реализация

Лабораторная работа № 5

(работа в аудитории 4 ч., СРС 4 ч.)

Цели работы

– закрепление теоретических знаний по теме работы;

– углубление навыков по применению методов разработки программного обеспечения;

– углубление навыков оценки временной и емкостной сложности программного обеспечения;

– углубление навыков чтения, понимания и выделения главной идеи прочитанного исходного кода;

– приобретение практических навыков в разработке компьютерных программ, реализующих алгоритмы нахождения независимых множеств, клик, вершинных покрытий в графах.

Задания

Перед выполнением работы необходимо повторить теоретический материал по теме. Основные теоретические положения изложены ниже в данном параграфе пункте «Теоретические сведения». После этого следует ответить на тестовые вопросы по теме.

В результате выполнения лабораторной работы требуется написать и отладить программу на алгоритмическом языке (C++ или ином), которая получает на входе числовые

данные, выполняет их обработку в соответствии с требованиями задания и выводит результат на экран в графической форме. Для обработки данных необходимо реализовать алгоритмы в соответствии с постановкой задачи. Ввод данных осуществляется из файла с учетом требований к входным данным, с учетом постановки задачи. Ограничениями на входные данные является допустимый диапазон значений используемых числовых типов в выбранной системе программирования. Работа состоит из двух указанных ниже этапов.

1 Выбрать вариант алгоритм решения поставленной задачи (варианты алгоритмов приведены ниже) или получить вариант от преподавателя.

2 Разработать тестовые примеры для отладки соответствующего алгоритма и программы. Согласовать с преподавателем.

3 Разработать и обосновать структуры данных для реализации алгоритма.

4 С учетом выполнения предыдущих пунктов задания написать и отладить программу на множестве тестовых примеров.

5 Оформить отчет по выполненной работе.

6 Защитить работу, ответив на вопросы преподавателя и студентов.

Варианты заданий

1. Найти минимальные доминирующие множества и числа доминирования для графов правильных многогранников.
2. Шахматному коню можно поставить в соответствие граф, вершины которого расположены на 64 полях доски, а ребра соответствуют ходам этой фигуры. Определить число независимости для графа данной фигуры.
3. Шахматному коню можно поставить в соответствие граф, вершины которого расположены на 64 полях доски, а ребра соответствуют ходам этой фигуры. Определить число доминирования для графа данной фигуры
4. Разместить на шахматной доске минимальное число ферзей так, чтобы они держали под боем каждую клетку доски.
5. Ладейный граф n × m представляет допустимые ходы ладьи на доске n × m. Вершинам графа можно задать координаты (x,y), где $1 \le x \le n$ и $1 \le y \le m$. Какому количеству ребер инцидентна вершина I,j?
6. Ладейный граф n × m представляет допустимые ходы ладьи на доске n × m. Вершинам графа можно задать координаты (x,y), где $1 \le x \le n$ и $1 \le y \le m$. Какому количеству циклов из четырех вершин принадлежат две несмежные вершины?
7. Определить число доминирования для ладейного графа на доске m×n
8. Выбрать в неориентированном графе минимальное (по количеству вершин) вершинное покрытие.
9. Найти клику в неориентированном графе.
10. Имеется *n* проектов, которые должны быть выполнены. Для выполнения проекта *xi* требуется некоторое подмножетсво *Ri* наличных ресурсов из множества $\{1, \ldots, p\}$. Пусть каждый проект, задаваемый совокупностью средств, необходимых для его реализации, может быть выполнен за один и тот же промежуток времени. Построим граф *G*, каждая вершина которого соответствует некоторому проекту, а ребро (*xi*, *xj*) наличию общих средств обеспечения у проектов *xi* и *xj*. Какое максимальное множество проектов, которое можно выполнить одновременно за один и тот же промежуток времени?
11. Выбор переводчиков. Предположим, что организации нужно нанять переводчиков с французского, немецкого, греческого, итальянского, испанского, русского и китайского языков на английский и что имеется пять кандидатур A, B, C, D и E. Каждая кандидатура владеет только некоторым собственным подмножеством из указанного выше множества языков и требует вполне определенную зарплату. Необходимо решить, каких переводчиков (с указанных выше языков на английский) надо нанять, чтобы затраты на зарплату были наименьшими. Очевидно, что это — задача о наименьшем покрытии.

Если, например, требования на оплату труда у всех претендентов одинаковые и группы языков, на которых они говорят, указаны ниже в матрице T, то решение задачи будет таким: нужно нанять переводчиков B , C и D.

12 Информационный поиск. Предположим, что некоторое количество единиц информации хранится в N массивах длины cj, j = 1, 2, . . ., N , причем на каждую единицу информации отводится по меньшей мере один массив. В некоторый момент делается запрос о M единицах информации. Они могут быть получены различными способами при помощи поиска в массиве. Для того чтобы получить все M единиц информации и при этом произвести просмотр массивов наименьшей длины, надо решить задачу о наименьшем покрытии, в которой элемент t_{ij} матрицы T равен 1, если информация i находится в массиве, и 0 в противном случае.

13 Маршруты полетов самолетов. Предположим, что вершины неориентированного графа G представляют аэропорты, а дуги графа G — этапы полетов (беспосадочные перелеты), которые осуществляются в заданное время. Любой маршрут в этом графе (удовлетворяющий ряду условий, которые могут встретиться на практике) соответствует некоторому реально выполнимому маршруту полета. Пусть имеется N таких возможных маршрутов и для каждого из них каким-то способом подсчитана его стоимость (например, стоимость j-го маршрута равна C_j). Задача нахождения множества маршрутов, имеющего наименьшую суммарную стоимость и такого, что каждый этап полета содержится хотя бы в одном выбранном маршруте, является задачей о наименьшем покрытии с матрицей $T = [t_{ij}]$, в которой элемент t_{ij} равен 1, если i-й этап содержится в j - м маршруте, и равен 0 в противном случае.

14 Маршруты полетов самолетов. Предположим, что вершины неориентированного графа G представляют аэропорты, а дуги графа G — этапы полетов (беспосадочные перелеты), которые осуществляются в заданное время. Любой маршрут в этом графе (удовлетворяющий ряду условий, которые могут встретиться на практике) соответствует некоторому реально выполнимому маршруту полета. Пусть имеется N таких возможных маршрутов и для каждого из них каким-то способом подсчитана его стоимость (например, стоимость j-го маршрута равна Cj). Задача нахождения множества маршрутов, имеющего наименьшую суммарную стоимость и такого, что каждый этап полета содержится хотя бы в одном выбранном маршруте, является задачей о наименьшем покрытии с матрицей $T = [t_{ij}]$, в которой элемент t_{ij} равен 1, если i-й этап содержится в j - м маршруте, и равен 0 в противном случае. Известно, что каждый этап содержится только в одном маршруте.

15 Граф представляет сеть дорог, одна его вершина представляет склад, а все остальные вершины изображают потребителей. Транспорт, покидающий склад, снабжает товаром некоторых потребителей, после чего возвращается на склад. Пусть «стоимость» маршрута j равна c_j (например, c_j может быть километражем маршрута или временем, необходимым для его прохождения). Спрашивается, сколько машин следует использовать на разных маршрутах, чтобы в один и тот же день доставлять всем потребителям товары (каждому потребителю поставляется сразу все необходимое) и чтобы суммарная стоимость проходимых маршрутов была наименьшей. Эту задачу, очевидно, можно рассматривать как частный случай транспортной задачи (без перевозки грузов), в которой столбцы представляют всевозможные осуществимые (с учетом практических ограничений) замкнутые маршруты, начинающиеся и кончающиеся на складе. Строки представляют потребителей.

16 Граф представляет сеть кабелей для подачи электроэнергии от подстанции к потребителям в «кольцевых схемах электроснабжения», несколько его вершина представляет подстанции, а все остальные вершины изображают потребителей.. Пусть «стоимость» маршрута j равна c_j (например, c_j может быть длиной маршрута или стоимостью необходимого количества кабеля). Спрашивается, как проложить маршруты, чтобы доставлять всем потребителям электроэнергию (каждому потребителю поставляется необходимое ее количество) и чтобы суммарная стоимость была наименьшей. Эту задачу, очевидно, можно рассматривать как частный случай транспортной задачи, в которой столбцы представляют всевозможные

осуществимые (с учетом практических ограничений) замкнутые маршруты, начинающиеся и кончающиеся на подстанциях. Строки представляют потребителей.

17 Пусть G — неориентированный граф. Обосновать неравенство $\alpha [G] \geq \beta [G]$, показав, что каждое максимальное независимое множество есть доминирующее множество.

18 Пусть G — неориентированный граф. Решить задачу о размещении телевизионных станций на некоторой территории. Территория разделена на районы, размер которых известен. Дальность передачи задана. Известно, какие районы являются соседними. Требуется найти наименьшее возможное число телевизионных станций и места для их размещения так, чтобы была охвачена вся территория.

19 Пусть G — неориентированный граф. Решить задачу о размещении военных баз, контролирующих данную территорию. Территория разделена на районы, размер которых известен. Известно, на максимальное расстоянии от базы, на котором район может контролироваться. Известно, какие районы являются соседними. Требуется найти наименьшее возможное число военных баз и места для их размещения так, чтобы вся территория контролировалась.

20 Пусть G — неориентированный граф. Решить задачу о размещении поликлиник на некоторой территории. Одна поликлиника может обслуживать несколько районов(количество обслуживаемых районов задано), размер районов известен. Для каждого района известны соседние. . Требуется найти наименьшее возможное число поликлиник и места для их размещения так, чтобы вся территория охватывалась медицинским обслуживанием

21 Пусть G — неориентированный граф. Решить задачу о размещении центров торговли на некоторой территории. Один центр может обслуживать несколько районов(количество обслуживаемых районов задано), размер районов известен. Для каждого района известны соседние. . Требуется найти наименьшее возможное число торговых центров и места для их размещения так, чтобы вся территория охватывалась торговым обслуживанием.

22 Пусть G — неориентированный граф. Найти минимальное доминирующее множество для заданного графа.

23 Пусть G — неориентированный граф. Найти наибольшую клику.

24 Пусть G — неориентированный граф. Найти максимальную клику.

25 Пусть G — неориентированный граф. Найти максимальное независимое множество.

26 Пусть G — неориентированный граф. Найти наибольшее независимое множество.

27 Пусть G — неориентированный граф. Найти число независимости графа.

28 Пусть G — неориентированный граф. Найти кликовое число графа.

29 Пусть G — неориентированный граф. Найти наименьшее доминирующее множество для заданного графа

30 Пусть G — неориентированный граф. Найти число доминирования графа

Теоретические сведения

Введение. Пусть дан граф $G = (X, \Gamma)$. Довольно часто возникает задача поиска таких подмножеств множества вершин X графа G, которые обладают определенным, наперед заданным свойством. Например, какова максимально возможная мощность такого подмножества S множества X, для которого порожденный подграф является полным? Или какова максимальная мощность подмножества S, такого, что порожденный граф— вполне несвязный? Ответ на первый вопрос дает так называемое *кликовое число* графа G, а на второй — *число независимости*. Еще одна задача состоит в нахождении минимально возможной мощности таких подмножеств S множества X, что любая вершина из $X — S$ достижима из S с помощью путей единичной длины. Решение этой задачи дает так называемое *число доминирования* графа G.

Эти числа и связанные с ними подмножества вершин описывают важные структурные свойства графа и имеют разнообразные непосредственные приложения при ведении проектного планирования исследовательских работ, в кластерном анализе и численных методах

таксономии, параллельных вычислениях на ЭВМ, при размещении предприятий обслуживания, а также источников и потребителей в энергосистемах.

Рассмотрим алгоритмы определения указанных выше чисел и обсудим некоторые их приложения. Кроме того, рассмотрим задачу о наименьшем покрытии, которая является обобщением задачи о нахождении числа доминирования графа, и метод ее решения. Последняя задача очень важна не только потому, что она имеет большое число прямых приложений, но и в связи с тем, что она часто возникает как подзадача в ряде разделов теории графов. В частности, большую роль она играет при вычислении хроматических чисел и других.

Независимые множества. Рассмотрим неориентированный граф $G = (X, Г)$. *Независимое множество вершин* (известное также как *внутренне устойчивое множество*) есть множество вершин графа *G,* такое, что любые две вершины в нем не смежны, т. е. никакая пара вершин не соединена ребром. Следовательно, любое множество $S \subset X$, которое удовлетворяет условию

$$S \cap Г(S) = \varnothing \qquad (5.1)$$

является независимым множеством вершин. Например, для графа, приведенного на рисунке 5.1, множества вершин {x7, x8, x2}, {x3, x1}, {x7, x8, x2, x5} — независимые. Если не возникают недоразумения, эти множества будут называться просто независимыми множествами (вместо независимые множества вершин).

Независимое множество называется *максимальным,* когда нет *другого* независимого множества, в которое оно бы входило. Таким образом, множество *S* является *максимальным независимым множеством,* если оно удовлетворяет условию (5.1) и условию:

$$H \cap Г(H) \neq \varnothing \qquad \forall H \supset S. \qquad (5.2)$$

Следовательно, для графа, приведенного на рис. 5.1, множество {x7; x8, x2, x5} является максимальным, а {x7, x8, x2} не является таковым. Множества {xl, x3, x7} и (x4, x6} также являются максимальными независимыми множествами, и, значит, в данном графе больше одного независимого множества. Следует также отметить, что число элементов (вершин) в разных максимальных множествах, как следует из приведенного примера, не обязательно одинаковое.

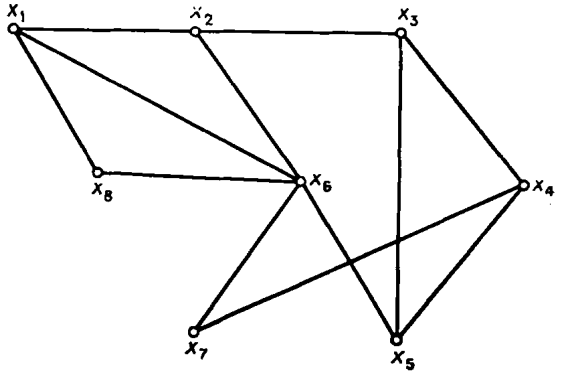


Рисунок 5.1. – Пример графа

Если Q является семейством всех независимых множеств графа G, то число

$$\alpha(G) = \max_{S \in Q} |S| \qquad (5.3)$$

называется числом независимости графа G, а множество S*, на котором этот максимум достигается, называется наибольшим независимым множеством.

Для графа, приведенного на рисунке 5.1, семейство максимальных независимых множеств таково: {x8,x7,x2,x5}, {x1,x3,x7}, {x2,x4,x8}, {x6,x4}, {x6,x3}, {x7,x5,x1}, {x1,x4}, {x3,x7,x8}.

Наибольшее из этих множеств имеет 4 элемента и, следовательно, $\alpha[G] = 4$. Множество {x8, x7, x2, x5} является наибольшим независимым множеством.

Кликой графа называется множество вершин, порождающее полный подграф, т.е. множество вершин, каждые две из которых смежные. Число вершин в клике наибольшего размера называется кликовым числом графа и обозначается через ω(G). Очевидно, задача о независимом множестве преобразуется в задачу о клике и наоборот простым переходом от данного графа G к дополнительному графу , так что α(G) = ω($\overline{G}$).

Вершинное покрытие графа – это такое множество вершин, что каждое ребро графа инцидентно хотя бы одной из этих вершин. Наименьшее число вершин в вершинном покрытии графа G обозначается через β(G) и называется числом вершинного покрытия графа. В графе на рисунке 5.2 наибольшим независимым множеством является множество {1,3,4,7}, наибольшей кликой - множество {2,3,5,6}, наименьшим вершинным покрытием – множество {2,5,6}..

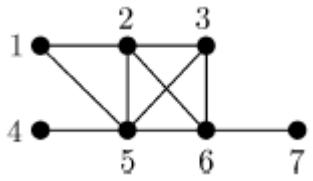


Рисунок 5.2. – Пример графа

Между задачами о независимом множестве и о вершинном покрытии тоже имеется простая связь благодаря следующему факту.

Теорема 1. Подмножество U множества вершин графа G является вершинным покрытием тогда и только тогда, когда $\overline{U}$ = VG-U - независимое множество.

Доказательство. Если U - вершинное покрытие, то всякое ребро содержит хотя бы одну вершину из множества U и, значит, нет ни одного ребра, соединяющего две вершины из множества $\overline{U}$. Следовательно, $\overline{U}$ - независимое множество. Обратно, если $\overline{U}$ - независимое множество, то нет ребер, соединяющих вершины из $\overline{U}$ и, значит, у каждого ребра одна или обе вершины принадлежат множеству U. Следовательно, U - вершинное покрытие.

Из этой теоремы следует, что α(G)+ β(G) = n для любого графа G с n вершинами.

Таким образом, все три задачи тесно связаны друг с другом, так что достаточно научиться решать одну из них, и мы будем уметь решать остальные две. Вместе с тем известно, что эти задачи NP-полны. Для таких задач не известно эффективных алгоритмов, а накопленный к настоящему времени опыт делает правдоподобным предположение о том, что таких алгоритмов и не существует. Тем не менее, алгоритмы для подобных задач разрабатывались и продолжают разрабатываться, и в некоторых случаях они могут быть полезны. Все эти алгоритмы в той или иной форме осуществляют перебор вариантов (число которых может быть очень большим). Далее рассмотрим один из способов такого перебора для задачи о независимом множестве.

Стратегия перебора для задачи о независимом множестве. Пусть G -граф, в котором требуется найти наибольшее независимое множество. Выберем в нем произвольную вершину a. Обозначим через G_1 подграф, получающийся удалением из графа G вершины a, т.е. $G_1 = G – \{ a\}$, а через G_2 - подграф, получающийся удалением из G всех вершин, смежных с a. На рисунке 5.3 показаны графы G_1 и G_2 , получающиеся из графа G , изображенного на рисунке 5.2 при a=1.

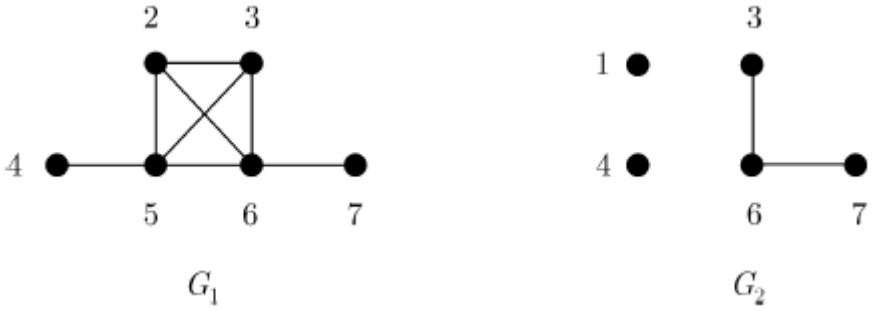


Рисунок 5.3. – Примеры графов

Пусть X- какое-нибудь независимое множество графа G. Если оно не содержит вершины a, то оно является независимым множеством графа G_1 . Если же a □ X то никакая вершина, смежная с a, не принадлежит X. В этом случае множество X является независимым множеством графа G_2 . Заметим, что в графе G_1 на одну вершину меньше, чем в исходном графе G. Если вершина a не является изолированной, то и в графе G_2 вершин меньше, чем в графе G. Таким образом, задача о независимом множестве для графа G свелась к решению той же задачи для двух графов меньшего размера. Это приводит к рекуррентному соотношению для числа независимости: $\alpha(G) = \max\{ \alpha(G1),\alpha(G2)\}$ и к рекурсивному алгоритму для нахождения наибольшего независимого множества графа G: найдем наибольшее независимое множество X_1 графа G_1 , затем наибольшее независимое множество X_2 графа G_2 и выберем большее из этих двух множеств.

В целом процесс решения задачи при этом можно рассматривать как исчерпывающий поиск в возникающем дереве подзадач. Чтобы не путать вершины дерева и вершины графа, вершины дерева будем называть узлами. Узел, не являющийся листом, называется внутренним узлом. Каждому внутреннему узлу дерева соответствует некоторый граф H и некоторая вершина этого графа x. Вершину x можно выбирать произвольно, но она не должна быть изолированной вершиной графа H. Внутренний узел имеет двух сыновей - левого и правого. Левому сыну соответствует подграф графа H, получаемый удалением вершины x, а правому -подграф, получаемый удалением всех вершин, смежных с x. Корню дерева соответствует исходный граф. Листьям соответствуют подграфы, не имеющие ребер, то есть подграфы, у которых все вершины изолированные. Множества вершин этих подграфов - это независимые множества исходного графа.

Для нахождения наибольшего независимого множества не обязательно строить все дерево полностью, а достаточно обойти его в том или ином порядке, запоминая на каждом шаге только небольшую часть информации об устройстве этого дерева. Можно, например, применить поиск в глубину для обхода дерева: сначала пройти от корня до некоторого листа, затем вернуться к предку этого листа и искать следующий лист, и т.д.

Для одного и того же графа могут получиться разные деревья в зависимости от того, как выбирается активная вершина x в каждом узле дерева. Может быть различным и число листьев в этих деревьях, а значит, и трудоемкость алгоритма, основанного на обходе дерева. Однако в любом случае листьев в дереве будет не меньше, чем максимальных независимых множеств у графа, так как каждое из этих множеств будет соответствовать некоторому листу. Так, для графа pK_2 , т.е. графа, состоящего из p компонент связности, каждая из которых изоморфна графу K_2 в дереве подзадач будет 2p листьев.

Эвристики для задачи о независимом множестве. Поиск в дереве вариантов в общем случае неэффективен, а приемы сокращения перебора, подобные описанному далее сжатию по включению, применимы далеко не ко всем графам. Одним из выходов из этого положения является применение эвристических алгоритмов, или эвристик. Так называются алгоритмы, основанные на каких-нибудь интуитивных соображениях, которые, как кажется, ведут к получению хорошего решения. Такие алгоритмы могут иногда не давать вообще ни-

какого решения или давать решение далеко не оптимальное. Но они, как правило, очень быстро работают и иногда (а может быть, и очень часто) дают решение, близкое к оптимальному или приемлемое для практики. Рассмотрим две простые эвристики для задачи о независимом множестве.

Одна из эвристических идей состоит в том, чтобы рассмотреть только один путь от корня до листа в дереве вариантов в надежде, что этому листу соответствует достаточно большое независимое множество. Для выбора этого единственного пути могут применяться разные соображения. В уже описанном дереве вариантов у каждого внутреннего узла имеются два сына. Одному из них соответствует подграф, получающийся удалением некоторой произвольно выбранной вершины a, а другому -подграф, получающийся удалением окрестности этой вершины. Чтобы вместо дерева получился один путь, достаточно каждый раз выполнять какую-нибудь одну из этих двух операций. Рассмотрим оба варианта.

Допустим, мы решили каждый раз удалять выбранную вершину. Эти удаления производятся до тех пор, пока не останется граф без ребер, т.е. независимое множество. Оно и принимается в качестве решения задачи. Для полного описания алгоритма необходимо еще сформулировать правило выбора активной вершины a. Мы хотим получить граф без ребер, в котором было бы как можно больше вершин. Чем меньше вершин будет удалено, тем больше их останется. Значит, цель - как можно быстрее удалить все ребра. Кажется, мы будем двигаться в нужном направлении, если на каждом шаге будем удалять наибольшее возможное на этом шаге число ребер. Это означает, что в качестве активной вершины всегда нужно выбирать вершину наибольшей степени. Алгоритмы такого типа называются жадными или градиентными. К сожалению, как будет показано дальше, оптимальный выбор на каждом шаге не гарантирует получения оптимального решения в конечном итоге.

Другой вариант - каждый раз удалять окрестность активной вершины a. Это повторяется до тех пор, пока оставшиеся вершины не будут образовывать независимого множества. Удаление окрестности вершины a равносильно тому, что сама эта вершина включается в независимое множество, которое будет получено в качестве ответа. Так как мы хотим получить в итоге как можно большее независимое множество, следует стараться удалять на каждом шаге как можно меньше вершин. Это означает, что в качестве активной вершины всегда нужно выбирать вершину наименьшей степени. Получается еще один вариант жадного алгоритма.

Имеется немало графов, для которых каждая из этих эвристик дает близкое к оптимальному, а иногда и оптимальное решение. Но, как это обычно бывает с эвристическими алгоритмами, можно найти примеры графов, для которых найденные решения будут весьма далеки от оптимальных. Рассмотрим граф Gk , у которого множество вершин V состоит из трех частей мощности k каждая: V = A $\cup$ B1 $\cup$ B2 , причем A является независимым множеством, каждое из множеств B1 , B2 - кликой, и каждая вершина из множества A смежна с каждой вершиной из множества B1 $\cup$ B2. С помощью операций суммы и соединения графов этот граф можно представить формулой Gk =(2Kk) °Ok. Степень каждой вершины из множества A в этом графе равна 2k, а степень каждой вершины из множества B1 $\cup$ B2. Равна 2k-1. . Первый алгоритм, выбирающий вершину наибольшей степени, будет удалять вершины из множества A до тех пор, пока не удалит их все. После этого останется граф, состоящий из двух клик, и в конечном итоге будет получено независимое множество из двух вершин. Второй алгоритм на первом шаге возьмет в качестве активной одну из вершин множества B1 $\cup$ B2. и удалит всю ее окрестность. В результате получится граф, состоящий из этой вершины и клики, а после второго шага получится независимое множество, состоящее опять из двух вершин. Итак, при применении к этому графу любой из двух эвристик получается независимое множество из двух вершин. В то же время в графе имеется независимое множество A мощности k.

Приближенный алгоритм для задачи о вершинном покрытии. Иногда для алгоритма, не гарантирующего точного решения, удается получить оценку степени приближения, т.е. отклонения получаемого решения от точного. В таком случае говорят о приближенном

алгоритме. Рассмотрим один простой приближенный алгоритм для задачи о вершинном покрытии.

Работа алгоритма начинается с создания пустого множества X и состоит в выполнении однотипных шагов, в результате каждого из которых к множеству X добавляются некоторые вершины. Допустим, перед очередным шагом имеется некоторое множество вершин X. Если оно покрывает все ребра (т.е. каждое ребро инцидентно одной из этих вершин), то процесс заканчивается и множество X принимается в качестве искомого вершинного покрытия. В противном случае выбирается какое-нибудь непокрытое ребро (a,b) и вершины a и b добавляются к множеству X.

Для полного описания алгоритма нужно дополнительно сформулировать правило выбора ребра (a,b). Однако для оценки степени приближения, которая будет сейчас получена, это не имеет значения. Можно считать, что какое-то правило выбрано.

Обозначим через $\beta`(G)$ мощность вершинного покрытия, которое получится при применении этого алгоритма к графу G, и докажем, что $\beta`(G) \le 2\beta(G)$. Иначе говоря, полученное с помощью этого алгоритма решение не более чем в два раза отличается от оптимального.

Действительно, допустим, что до окончания работы алгоритм выполняет k шагов, добавляя к множеству X вершины ребер $(a_1,b_1), \ldots,(a_k,b_k)$. Тогда $\beta`(G) = 2k$. Никакие два из этих k ребер не имеют общей вершины. Значит, чтобы покрыть все эти ребра, нужно не меньше k вершин. Следовательно, $\beta(G) \ge k$ и $\beta`(G) \le 2\beta(G)$.

Перебор максимальных независимых множеств. Основной недостаток изложенного выше способа организации перебора состоит в том, что при нем часто рассматриваются не только максимальные независимые множества. Это означает, что выполняется лишняя работа, так как наибольшее независимое множество находится среди максимальных. Кроме того, в некоторых случаях бывает необходимо знать все максимальные независимые множества. Рассмотрим алгоритм, который строит все максимальные и только максимальные независимые множества графа.

Предположим, что вершинами заданного графа G являются числа 1, 2, …, n. Рассматривая любое подмножество множества вершин, будем выписывать его элементы в порядке возрастания. Лексикографический порядок на множестве получающихся таким образом кортежей порождает линейный порядок на множестве всех подмножеств множества вершин, который тоже будем называть лексикографическим. Например, множество {2,5,7,9} предшествует в этом порядке множеству {2,5,8}, а множество {2,5,7,10} занимает промежуточное положение между ними.

Нетрудно найти лексикографически первое максимальное независимое множество: нужно на каждом шаге брать наименьшую из оставшихся вершин, добавлять ее к построенному независимому множеству, а все смежные с ней вершины удалять из графа. Если граф задан списками смежности, то это построение выполняется за время O(m).

Допустим теперь, что $U \subseteq VG$, G` - подграф графа G, порожденный множеством U, и пусть имеется список L всех максимальных независимых множеств графа G. Тогда однократным просмотром списка L можно получить список L` всех максимальных независимых множеств графа G` . Это основано на следующих очевидных утверждениях:

1. каждое максимальное независимое множество графа G` содержится в некотором максимальном независимом множестве графа G;
2. для каждого максимального независимого множества N графа G` имеется точно одно максимальное независимое множество M графа G, такое, что $N \subseteq M$ и (M –N) – лексикографически первое среди максимальных независимых множеств подграфа, порожденного множеством $VG-(U \cup V(N))$ (последнее утверждение верно и в том случае, если $VG-(U \cup V(N)) = \varnothing$, если считать, что пустое множество является максимальным независимым множеством в графе с пустым множеством вершин).

Будем теперь рассматривать множества из L одно за другим и пусть M - очередное такое множество. Положим $N = M \cap U$. Если N не является максимальным независимым

множеством графа G` , то переходим к следующему элементу списка L. Если же N- максимальное независимое множество в G`, то рассматриваем множество M-N. Если оно является лексикографически первым среди максимальных независимых множеств подграфа, порожденного множеством VG-(U $\cup$ V(N)), то включаем N в список L`.

Выберем в графе G произвольную вершину a и пусть A - множество всех вершин графа, смежных с a (окрестность вершины a), B - множество всех вершин, не смежных с a и отличных от a. Обозначим через G_1 подграф, получающийся удалением из графа G вершины a, а через G_2 подграф, получающийся удалением из G всех вершин множества A $\cup$ {a}. Иначе говоря, G_1 – подграф графа G, порожденный множеством A $\cup$ B, а G_2 - подграф, порожденный множеством B.

Допустим, что имеется список L_1 всех максимальных независимых множеств графа G_1 . На основании вышеизложенного можно предложить следующую процедуру получения списка L всех максимальных независимых множеств графа G.

1. Взять очередной элемент M списка L_1 .
2. Если M$\subseteq$B, то добавить к списку L множество M $\cup$ {a} и перейти к 1, иначе добавить к списку L множество M.
3. Если множество M $\cap$ B не является максимальным независимым множеством в графе G_2, то перейти к 1.
4. Если множество N = M $\cap$ A является лексикографически первым максимальным независимым множеством подграфа, порожденного множеством A – V(M $\cap$ B), то добавить к списку L множество M $\cup$ {a}.
5. Если список L_1 не исчерпан, перейти к 1.

Начиная с одновершинного графа (у которого список максимальных независимых множеств состоит из одного элемента), добавляя последовательно по одной вершине, получаем последовательность графов G_1 , G_2 , … G_n = G. Применяя для каждого i = 1,2, …,(n – 1) описанный алгоритм для построения списка всех максимальных независимых множеств графа G_{i+1} по такому списку для графа G_i , в конце концов, получим список всех максимальных независимых множеств графа G. Этот алгоритм представляет собой поиск в ширину в дереве вариантов. Для того чтобы не хранить все получающиеся списки, его можно преобразовать в поиск в глубину. Заметим, что приведенная процедура для каждого максимального независимого множества графа G_i находит одно или два максимальных независимых множества графа G_{i+1}. Одно из этих новых множеств рассматривается на следующем шаге, другое, если оно есть, рекомендуется запомнить в стеке.

Изложенный алгоритм можно применить для поиска наибольших независимых множеств в графах, про которые известно, что в них мало максимальных независимых множеств. Одним из классов графов с таким свойством является класс всех графов, не содержащих $2K_2$ в качестве порожденного подграфа. Известно, что в графе с m ребрами из этого класса число максимальных независимых множеств не превосходит m+1.

Тестовые вопросы по теме 5

1. Что такое порожденный граф?
2 Что такое кликовое число?
3. Что такое число независимости?
4 Что такое число доминирования графа?
5. Что такое независимое множество вершин?
6 Какое независимое множество является максимальным независимым множеством? Приведите примеры
7. Что такое клика графа? Приведите примеры
8. Сформулируйте и докажите необходимое и достаточное условие существования вершинного покрытия графа
9. Охарактеризуйте стратегию перебора для задачи о независимом множестве.

10. Постройте алгоритм для нахождения независимого множества

11. Приведите примеры эвристических алгоритмов решения задачи о независимом множестве

12. Алгоритм решения задачи о вершинном покрытии?

13. Алгоритм решения задачи нахождения максимальных независимых множеств?

Рекомендуемые источники по теме 5

1. Кристофидес, Н. Теория графов. Алгоритмический подход. – М.: Мир, 1978.
2. https://studfiles.net/preview/5683649/page:9/

6 Алгоритмы раскраски на графах и их программная реализация

Лабораторная работа № 6
(работа в аудитории 6 ч., СРС 6 ч.)

Цели работы

– закрепление теоретических знаний по теме работы;

– углубление навыков по применению методов разработки программного обеспечения;

– углубление навыков оценки временной и емкостной сложности программного обеспечения;

– углубление навыков чтения, понимания и выделения главной идеи прочитанного исходного кода;

– приобретение практических навыков в разработке компьютерных программ, реализующих алгоритмы раскраски вершин и ребер графа.

Задание

Перед выполнением работы необходимо повторить теоретический материал по теме. Основные теоретические положения изложены ниже в данном параграфе пункте «Теоретические сведения». После этого следует ответить на тестовые вопросы по теме.

В результате выполнения лабораторной работы требуется написать и отладить программу на алгоритмическом языке (C++ или ином), которая получает на входе числовые данные, выполняет их обработку в соответствии с требованиями задания и выводит результат на экран в графической форме. Для обработки данных необходимо реализовать алгоритмы обхода графа в соответствии с постановкой задачи. Ввод данных осуществляется из файла с учетом требований к входным данным, с учетом постановки задачи. Ограничениями на входные данные является допустимый диапазон значений используемых числовых типов в выбранной системе программирования. Работа состоит из двух указанных ниже этапов.

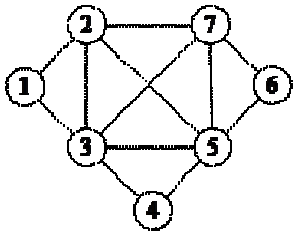
- Выбрать алгоритм решения поставленной задачи по заданному варианту (варианты заданий приведены ниже).
- Разработать тестовые примеры для отладки соответствующего алгоритма и программы. Согласовать с преподавателем.
- Разработать и обосновать структуры данных для реализации алгоритма.
- С учетом выполнения предыдущих пунктов задания написать и отладить программу на множестве тестовых примеров.
- Оформить отчет по выполненной работе.
- Защитить работу, ответив на вопросы преподавателя и студентов.

Варианты заданий

1. Найти хроматическое число графа. Граф задан матрицей смежности.
2. Найти хроматический индекс графа. Граф задан матрицей смежности.
3. Найти хроматическое число графа. Граф задан списком ребер.
4. Проверить граф на двудольность, используя алгоритм раскраски. Граф задан матрицей смежности.
5. Найти хроматический индекс двудольного графа. Граф задан списком ребер.
6. Найти хроматический индекс полного графа. Граф задан списком ребер.
7. Найти хроматическое число графа. Можно ли по результату утверждать, что граф является полным? Граф задан списком ребер.
8. Найти хроматическое число графа, являющегося циклом.
9. Найти хроматический индекс графа. Граф задан множеством вершин и множеством ребер.
10. Найти хроматическое число плоского графа. Граф задан списком ребер. Плоским графом называется граф, изображенный на плоскости так, что никакие два его ребра (или, вернее, представляющие их кривые) геометрически не пересекаются нигде, кроме инцидентной им обоим вершины.
11. Найти хроматический индекс плоского графа. Граф задан матрицей смежности. Плоским графом называется граф, изображенный на плоскости так, что никакие два его ребра (или, вернее, представляющие их кривые) геометрически не пересекаются нигде, кроме инцидентной им обоим вершины.
12. Найти хроматическое число заданного графа, используя алгоритм нахождения независимых множеств, указать, какие вершины в какой цвет окрашиваются.
13. Найти хроматический индекс заданного графа, используя алгоритм нахождения независимых множеств, указать, какие ребра в какой цвет окрашиваются.
14. Пусть для выполнения каких-то n работ надо распределить m имеющихся в наличии ресурсов. Считаем, что каждая из работ выполняется за некоторый (одинаковый для всех работ) промежуток времени и что для выполнения i-й работы требуется подмножество ресурсов Si. Построим граф G: каждой работе соответствует определенная вершина графа, а ребро (xi, xj) существует в графе тогда и только тогда, когда для выполнения i-й и j-й работ требуется хотя бы один общий ресурс, т. е. когда $Si \cap Sj \neq \varnothing$. Это означает, что i-я и j-я работы не могут выполняться одновременно. Раскраска графа G определяет тогда некоторое распределение ресурсов (по выполняемым работам), причем такое, что работы, соответствующие вершинам одного цвета, выполняются одновременно. Наилучшее использование ресурсов (т. е. выполнение всех n работ за наименьшее время) достигается при оптимальной раскраске вершин графа G.
15. В задачах теории расписаний осмотры представляются в виде временных интервалов. Каждому осмотру можно сопоставить вершину некоторого графа, причем две любые вершины графа будут соединены ребром лишь тогда, когда соответствующие им осмотры нельзя осуществлять одновременно. Требуется составить такой график осмотра, который связан с наименьшими временными затратами (с учетом приведенных выше ограничений на «совместимость» осмотров). Эта задача эквивалентна задаче о раскраске вершин графа с использованием наименьшего числа цветов. Хроматическое число графа как раз и соответствует осмотру, требующему наименьших временных затрат.
16. Для детей была построена игровая площадка, состоящая из n платформ, соединенных дорожками. Платформы было решено правильно покрасить в k цветов, исходя из их наличного количества. Под правильной раскраской понимается вершинная раскраска графа. Проверить, возможно ли это. Граф задан матрицей смежности.

17. Для детей была построена игровая площадка, состоящая из n платформ, соединенных дорожками. Дорожки было решено правильно покрасить в k цветов, исходя из их наличного количества. Под правильной раскраской понимается реберная раскраска графа. Проверить, возможно ли это. Граф задан списком ребер.

18. Примените алгоритм последовательной раскраски к графу, изображенному на рисунке, если вершины упорядочиваются по возрастанию номеров



19. Примените алгоритм последовательной раскраски к графу, изображенному на рисунке задачи 18, если вершины упорядочиваются по убыванию номеров.

20. Для каких из следующих графов алгоритм последовательной раскраски дает точный результат при любом упорядочении вершин: P3 , P4, C4 , C5 , K2,4 ?

21. Найти хроматическое число графа, являющегося дополнением к C_7.

Теоретические сведения

Раскраска вершин. *Раскраской* вершин графа называется назначение цветов его вершинам. Обычно цвета - это числа 1,2,…,k. Тогда *раскраска* является функцией, определенной на множестве вершин графа и принимающей значения в множестве {1,2,…,k}. Раскраску можно также рассматривать как *разбиение множества* вершин графа $V = V_1 \cup V_2 \cup \ldots \cup V_k$, где V_i - *множество вершин* цвета i. *Множества* V_i называют цветными классами. *Раскраска* называется *правильной*, если каждый цветной *класс* является независимым множеством. Иначе говоря, в правильной раскраске любые две смежные вершины должны иметь разные цвета. Задача о раскраске состоит в нахождении правильной раскраски данного графа G в наименьшее число цветов. Это число называется *хроматическим числом* графа и обозначается $\chi(G)$.

В правильной раскраске полного графа K_n все вершины должны иметь разные цвета, поэтому $\chi(K_n) = n$. Если в каком-нибудь графе имеется полный *подграф* с k вершинами, то для раскраски этого подграфа необходимо k цветов. Отсюда следует, что для любого графа выполняется *неравенство* $\chi(G) \geq \omega(G)$.

Но хроматическое число может быть и строго больше кликового числа. Например, для цикла длины 5 $\omega(C_5) = 2$, а $\chi(G) = 3$. Другой пример показан на рис. 6.1. На нем изображен граф, вершины которого раскрашены в 4 цвета (цвета вершин показаны в скобках). Нетрудно проверить, что трех цветов для правильной раскраски этого графа недостаточно. Следовательно, его хроматическое число равно 4. Очевидно также, что кликовое число этого графа равно 3.

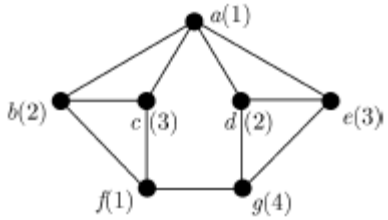


Рисунок 6.1. – Пример графа, кликовое число которого меньше хроматического числа

Очевидно, что $\chi(G) = 1$тогда и только тогда, когда G – пустой граф. Нетрудно охарактеризовать и графы с хроматическим числом 2 (точнее, не больше 2). По определению, это

такие графы, у которых множество вершин можно разбить на два независимых подмножества. Но это совпадает с определением двудольного графа. Поэтому двудольные графы называют бихроматическими. Согласно теореме Кенига граф является бихроматическим тогда и только тогда, когда в нем нет циклов нечетной длины.

Для графов с хроматическим числом 3 такого простого описания мы не знаем. Неизвестны и простые алгоритмы, проверяющие, можно ли данный граф раскрасить в 3 цвета. Более того, такая задача проверки (вообще, задача проверки возможности раскрасить граф в k цветов при любом фиксированном $k \geq 3$) является NP-полной.

Переборный алгоритм для раскраски. Рассмотрим алгоритм решения задачи о раскраске, похожий на алгоритм для задачи о нахождении независимого множества. Сходство заключается в том, что задача для данного графа сводится к той же задаче для двух других графов. Поэтому снова возникает дерево вариантов, обход которого позволяет найти решение. Но есть и одно существенное различие, состоящее в том, что два новых графа не будут подграфами исходного графа.

Выберем в данном графе G две несмежные вершины x и y и построим два новых графа: G_1, получающийся добавлением ребра (x,y) к графу G, и G_2, получающийся из G слиянием вершин x и y. Операция слияния состоит в удалении вершин x и y и добавлении новой вершины z и ребер, соединяющих ее с каждой вершиной, с которой была смежна хотя бы одна из вершин x, y. На рисунке 6.2 показаны графы G_1 и G_2, получающиеся из графа G, изображенного на рисунке. 6.1, с помощью этих операций, если в качестве x и y взять вершины a и f.

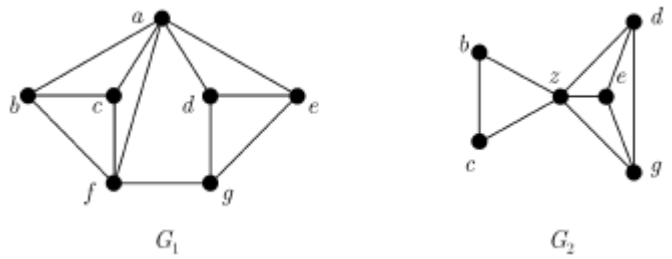


Рисунок 6.2. Графы, полученные из графа рисунка 6.1 операциями добавления ребра(G_1) и слиянием вершин(G_2)

Если в правильной раскраске графа G вершины x и y имеют разные цвета, то она будет правильной и для графа G_1. Если же цвета вершин x и y в раскраске графа G одинаковы, то граф G_2 можно раскрасить в то же число цветов: новая вершина z окрашивается в тот цвет, в который окрашены вершины x и y, а все остальные вершины сохраняют те цвета, которые они имели в графе G. И наоборот, раскраска каждого из графов G_1, G_2 , очевидно, дает раскраску графа G в то же число цветов. Поэтому

$$\chi(G) = \min\{ \chi(G_1), \chi(G_2)\},$$

что дает возможность рекурсивного нахождения раскраски графа в минимальное число цветов. Заметим, что *граф* G_1 имеет столько же вершин, сколько исходный *граф*, но у него больше ребер. Поэтому *рекурсия* в конечном счете приводит к полным графам, для которых задача о раскраске решается тривиально.

Раскраска ребер. Наряду с задачей о раскраске вершин существует и задача о раскраске ребер графа, когда цвета назначаются ребрам. Раскраска ребер (или реберная раскраска) называется правильной, если любые два ребра, имеющие общую вершину, окрашены в разные цвета. Минимальное число цветов, необходимое для правильной раскраски ребер графа G, называется хроматическим индексом графа и обозначается через $\chi'(G)$.

Обозначим через $\Delta(G)$ максимальную степень вершины графа. При правильной реберной раскраске все ребра, инцидентные одной вершине, должны иметь разные цвета. От-

сюда следует, что для любого графа выполняется неравенство $\chi'(G) \geq \Delta(G)$. Для некоторых графов имеет место строгое неравенство, например, $\Delta(C_3) = 2$, а $\chi'(C_3) = 3$.

Следующая теорема, доказанная В.Г. Визингом в 1964 г., показывает, что $\chi'(G)$ может отличаться от $\Delta(G)$ не более чем на 1.

Теорема 1. Для любого графа G справедливы неравенства $\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$.

Доказательство. Приводимое ниже доказательство дает и план алгоритма для раскрашивания ребер графа не более чем в $\Delta(G) + 1$ цветов. Оно основано на двух операциях перекрашивания Опишем их. Далее будут рассматриваться частичные реберные раскраски, т.е. правильные раскраски, при которых некоторые ребра остаются неокрашенными.

Допустим, ребра графа G правильно (может быть, частично) раскрашены. Пусть α и β- два из использованных в этой раскраске цветов. Рассмотрим подграф H, образованный всеми ребрами, имеющими цвета α или β. В этом подграфе степень каждой вершины не превосходит 2, следовательно, каждая компонента связности в нем является цепью или циклом. Такую компоненту будем называть (α, β) - компонентой. Если в какой-нибудь (α, β) - компоненте поменять местами цвета α и β (т.е. все ребра, окрашенные в цвет α, перекрасить в цвет β и наоборот), то полученная раскраска тоже будет правильной. Эту операцию назовем перекраской (α, β) -компоненты.

Другая операция применяется к частично раскрашенному подграфу, называемому веером. Будем говорить, что при данной раскраске цвет α отсутствует в вершине x, если ни одно из ребер, инцидентных вершине x, не окрашено в этот цвет. Веером называется подграф $F(x,y_1, \ldots, y_k, \alpha_1, \ldots, \alpha_k)$, состоящий из вершин $x,y_1, \ldots, y_k$ и ребер $(x,y_1),\ldots, (x,y_k)$, в котором:

- ребро (x,y_1) не окрашено;
- ребро (x,y_i) окрашено в цвет α_{i-1}, $i = 2,\ldots,k$;
- в вершине y_i отсутствует цвет α_i, $i = 1,\ldots,k$;
- $\alpha_1, \ldots, \alpha_{k-1}$ все попарно различны.

Перекраска веера состоит в том, что ребра $(x,y_1), \ldots, (x,y_{k-1})$ окрашиваются соответственно в цвета $\alpha_1, \ldots, \alpha_{k-1}$, а ребро (x,y_k) становится неокрашенным. Очевидно, новая частичная раскраска тоже будет правильной. На рисунке . 6.3 слева показан веер, а справа - результат его перекраски. Цвета ребер представлены числами, а отсутствующие цвета в вершинах - числами со знаком минус. Неокрашенное ребро изображено пунктиром.

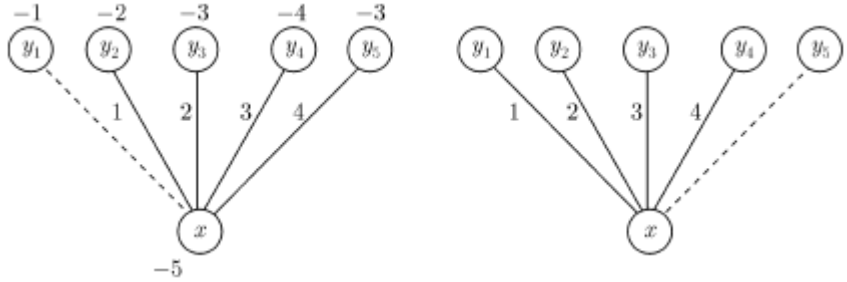


Рисунок. 6.3. Пример веерной перекраски левого графа в правый граф

Покажем, что с помощью этих двух процедур перекрашивания можно ребра любого графа G окрасить в не более чем $\Delta(G) + 1$ цвет. Допустим, что уже построена частичная правильная раскраска, использующая не более чем $\Delta(G) + 1$ цвет, и имеется неокрашенное ребро (x,y). Так как число разрешенных цветов больше, чем максимальная степень вершины, то в каждой вершине какой-нибудь цвет отсутствует. Допустим, в вершине x отсутствует цвет β.

Будем строить веер следующим образом. Положим $y_1 = y$ и пусть α_1- цвет, отсутствующий в вершине y. Получаем веер $F(x,y_1,\alpha_1)$. Допустим, веер $F(x,y_1, \ldots, y_k, \alpha_1, \ldots, \alpha_k)$, уже построен. Если цвет α_k отличен от $\alpha_1, \ldots, \alpha_{k-1}$ и существует инцидентное вершине x ребро (x,z) этого цвета, то увеличиваем k на 1 и полагаем $y_k = z$, α_k - цвет, отсутствующий в

вершине z. Этот процесс построения веера продолжается до тех пор, пока не наступит одно из следующих событий.

(А) Нет ребра цвета α_k, инцидентного вершине x. Перекрашиваем веер, в результате ребро (x,y) становится окрашенным, а ребро (x,y_k) - неокрашенным, причем цвет α_k отсутствует и в вершине y_k, и в вершине x. Но тогда можно это ребро окрасить в цвет α_k, и получится правильная раскраска, в которой на одно окрашенное ребро больше.

(Б) Цвет α_k совпадает с одним из цветов $\alpha_1, \ldots, \alpha_{k-1}$ (именно этот случай изображен на рисунке 5.3). Пусть $\alpha_k = \alpha_i$. Рассмотрим вершины x, y_i, y_k. В каждой из них отсутствует какой-нибудь из цветов β или α_k. Значит, в подграфе, образованном ребрами этих двух цветов, степень каждой из этих вершин не превосходит 1. Следовательно, все три вершины не могут принадлежать одной (α_k,β) -компоненте. Рассмотрим две возможности.

(Б1) Вершины x и y_i принадлежат разным (α_k,β) -компонентам. Перекрасим веер $F(x,y_1, \ldots, y_i, \alpha_1, \ldots, \alpha_i)$. Ребро (x, y_i) станет неокрашенным. Теперь перекрасим (α_k,β) -компоненту, содержащую вершину y_i. После этого цвет β будет отсутствовать в вершине y_i и ребро (x, y_i) можно окрасить в этот цвет.

(Б2) Вершины x и y_k принадлежат разным (α_k,β) -компонентам. Перекрасим веер $F(x,y_1, \ldots, y_k, \alpha_1, \ldots, \alpha_k)$. Ребро (x, y_k) станет неокрашенным. Теперь перекрасим (α_k,β) -компоненту, содержащую вершину y_k. После этого цвет β будет отсутствовать в вершине y_k и ребро (x, y_k) можно окрасить в этот цвет.

Итак, в любом случае получаем правильную раскраску, в которой добавилось еще одно раскрашенное ребро (x, y).

На рисунке 6.4 иллюстрируются случаи (Б1) и (Б2) на примере веера из рисунка 6.3. Здесь k = 5, i = 3. Левое изображение соответствует случаю (Б1): вершины x и y_3 принадлежат разным (3,5) -компонентам. После перекраски веера $F(x, y_1, y_2, y_3, 1,2,3)$ и (3,5) -компоненты, содержащей вершину y_3, появляется возможность окрасить ребро (x, y_3) в цвет 5. Случай (Б2) показан справа: здесь вершины x и y_5 принадлежат разным (3,5)-компонентам, поэтому после перекраски веера $F(x, y_1, y_2, y_3, y_4, y_5, 1,2,3,4,3)$ и (3,5) -компоненты, содержащей вершину y_5, появляется возможность окрасить ребро (x, y_5) в цвет 5.

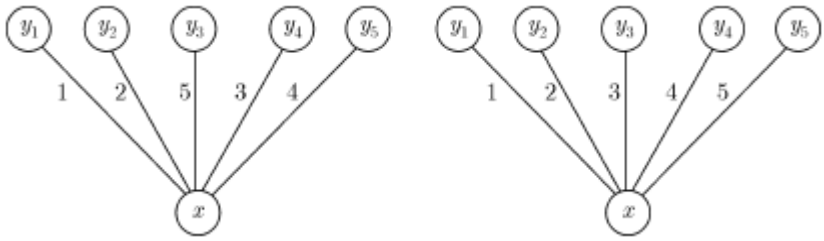


Рисунок 6.4. – Пример для случаев (Б1), (Б2)

Итак, все графы делятся на два класса: у одних *хроматический индекс* равен максимальной *степени вершины*, у других он на единицу больше. Оказывается, *определение* принадлежности графа к тому или иному классу является NP-трудной задачей. *Алгоритм*, который можно извлечь из доказательства теоремы 1, за полиномиальное время находит раскраску в не более чем $\Delta(G) + 1$ цвет. Его можно назвать "идеальным" приближенным алгоритмом, так как более высокую *точность* имеет только точный *алгоритм*.

Остановимся на вопросах рационализации алгоритмов решения задач раскраски и нахождения независимого множества.

Рационализация поиска наибольшего независимого множества. Известны различные приемы сокращения перебора при использовании описанной стратегии исчерпывающего поиска. Один из них основан на следующем наблюдении.

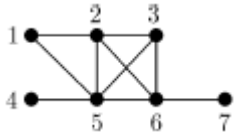


Рисунок 6.5. Пример графа, у которого вершина 2 смежно поглощает вершины 1 и 3. Вершины 5 и 6 – смежно поглощающие

Допустим, в графе G, для которого нужно найти наибольшее независимое множество, имеются две вершины a и b , такие, что каждая *вершина*, отличная от b и смежная с вершиной a , смежна и с вершиной b. Иначе говоря, множество V(a) – {b} является подмножеством множества V(b). Возможно совпадение. Будем говорить в этом случае, что *вершина* b поглощает вершину a. Если при этом вершины a и b смежны, то скажем, что *вершина* b смежно поглощает вершину a. Вершину b в этом случае назовем смежно поглощающей. Например, в графе, изображенном на рисунке 6.5, *вершина* 2 смежно поглощает вершины 1 и 3. Вершины 5 и 6 в этом графе тоже являются смежно поглощающими.

Теорема 1. Если вершина b является смежно поглощающей в графе G, то $\alpha(G-b) = \alpha(G)$.

Доказательство. Допустим, вершина b смежно поглощает вершину a в графе G. Пусть X - наибольшее независимое множество графа G. Если X не содержит вершину b, то оно является наибольшим независимым множеством и в графе G - b, так что в этом случае $\alpha(G-b) = \alpha(G)$. Предположим, что множество X содержит вершину b. Тогда ни одна вершина из множества V(b) не принадлежит X. Значит, X не содержит вершину a и ни одну вершину из множества V(a). Но тогда множество (X – {b}) U {a} тоже будет независимым, причем оно целиком содержится в графе G - b, а число элементов в нем такое же, как в множестве X. Значит, и в этом случае $\alpha(G-b) = \alpha(G)$.

Итак, если мы удалим из графа смежно поглощающую вершину b , то получим граф с тем же числом независимости. Так как новый граф является порожденным подграфом исходного графа G, то каждое наибольшее независимое множество нового графа будет наибольшим независимым множеством исходного. Этот прием называется "сжатием по включению". Исследование применимости и применение операции сжатия по включению к каждому встречающемуся подграфу требуют, конечно, дополнительных расходов времени, но могут привести к существенному сокращению дерева подзадач. Для некоторых графов задача о независимом множестве может быть решена с помощью одних только сжатий по включению. Таков, например, граф pK_2, и вообще любой лес. Действительно, любая вершина, смежная с листом, поглощает этот лист. Рассмотрим более широкий класс графов, для которых этот прием эффективен.

Хордальные графы. Граф называется хордальным (или триангулированным), если в нем нет порожденных простых циклов длины ≥ 4. Иначе говоря, в хордальном графе для каждого простого цикла длины 4 или больше имеется хотя бы одна хорда - ребро, не принадлежащее циклу, но соединяющее две вершины цикла.

Теорема 2. В любом непустом хордальном графе имеется смежно поглощающая вершина.

Доказательство. Пусть G - непустой граф, в котором нет смежно поглощающих вершин. Докажем, что G - не хордальный. Рассмотрим в нем простой путь $P = x_1, x_2, \ldots, x_k$ наибольшей длины, не имеющий хорд, то есть ребер, соединяющих две вершины пути и не принадлежащих пути. Так как граф непустой, то $k \geq 2$. Рассмотрим вершину x_k .Так как она не поглощает вершину x_{k-1}, то существует вершина $y \neq x_{k-1}$, смежная с вершиной x_k, но не смежная с x_{k-1}. Вершина y не принадлежит пути P , так как иначе ребро (x_k,y) было бы хордой этого пути. Таким образом, последовательность $P' = x_1, x_2, \ldots x_k, y$ является простым путем. Но длина этого пути больше, чем длина пути P, поэтому, в силу выбора пути P , у пути P' должна существовать хорда. Такой хордой может быть только ребро вида (y, x_i) , где $i \leq k - 2$. Пусть i - наибольшее, при котором ребро (y, x_i) является хордой пути P'. Тогда последовательность $y, x_i, x_{i+1}, \ldots, x_k, y$ является циклом без хорд длины не менее 4.

Итак, для хордального графа наибольшее независимое множество можно найти с помощью одних только сжатий по включению. Нужно только находить смежно поглощающие

вершины и удалять их из графа до тех пор, пока оставшийся граф не станет пустым. Множество оставшихся вершин и является наибольшим независимым множеством.

Рационализация алгоритма для задачи о раскраске вершин

В описанную схему решения задачи о раскраске можно включить тот же прием сжатия по включению, что и для задачи о независимом множестве. Небольшое отличие состоит в том, что теперь вершины a и b должны быть несмежны. Итак, пусть в графе G имеются две несмежные вершины a и b, такие, что V(a) подмножество V(b). Будем говорить, что *вершина* b несмежно поглощает вершину a , а вершину a называть несмежно поглощаемой. В графе на рисунке 5.5 *вершина* 1 несмежно поглощает вершину 4, а *вершина* 3 - вершину 7.

Теорема 3. Если вершина a является несмежно поглощаемой в графе G, то $\chi(G-a) = \chi(G)$.

Доказательство. Допустим, вершина a несмежно поглощается вершиной b. Рассмотрим правильную раскраску графа G – a в наименьшее число цветов. Применим эту же раскраску к графу G, окрасим вершину a в тот цвет, который имеет вершина b. Так как вершина a смежна только с такими вершинами, с которыми смежна b, то получится правильная раскраска графа G в то же самое число цветов. Следовательно, $\chi(G) = \chi(G-a)$.

Как и для задачи о независимом множестве, для некоторых графов этот прием позволяет находить решение, совсем не прибегая к перебору. Допустим, вершина b смежно поглощает вершину a в графе G. Тогда в дополнительном графе для графа G, очевидно,вершина a будет несмежно поглощать вершину b. Верно и обратное утверждение. Поэтому из теоремы 2 следует

Теорема 4. В любом графе, дополнительном к хордальному и не являющемся полным, имеется несмежно поглощаемая вершина.

Таким образом, для графов, дополнительных к хордальным, раскраска в минимальное число цветов может быть найдена с помощью одних только сжатий по включению. Оказывается, и для хордальных графов существует эффективное решение задачи о раскраске.

Установим сначала некоторые свойства хордальных графов. Подмножество множества вершин графа называется разделяющим множеством, если удаление всех этих вершин приводит к увеличению числа компонент связности. Таким образом, понятие разделяющего множества является обобщением понятия шарнира. Разделяющее множество называется минимальным, если оно не содержится в большем разделяющем множестве.

Теорема 5. В хордальном графе всякое минимальное разделяющее множество является кликой.

Доказательство. Допустим, что в некотором графе G есть минимальное *разделяющее множество* X, не являющееся кликой. Это означает, что в X имеются несмежные вершины a и b. При удалении множества X образуется не менее двух новых компонент связности. Пусть G_1 и G_2- такие компоненты. Вершина a смежна, по крайней мере, с одной вершиной в каждой из этих компонент. Действительно, если a была бы не смежна, скажем, ни с одной из вершин компоненты G_1, то множество $X – \{a\}$ тоже было бы разделяющим, а это противоречит минимальности разделяющего множества X. То же относится к вершине b. Выберем в компоненте G_1 такие вершины x_1 и y_1, чтобы x_1 была смежна с вершиной a, y_1 - с вершиной b и при этом расстояние между x_1 и y_1 в C_1 было минимальным (возможно $x_1 = y_1$). Аналогично выберем x_2 и y_2 в C_2 - компоненте. Пусть P_1 - кратчайший путь из x_1 в y_1 в компоненте C_1, а P_2- кратчайший путь из y_2 в x_2 в компоненте C_2 (каждый из этих путей может состоять из одной вершины). Тогда последовательность a,P_1, b , P_2 , a является простым циклом без *хорд* длины не менее 4. Следовательно, граф G - не хордальный.

Вершина графа называется *симплициальной*, если множество всех смежных с ней вершин является кликой или пустым множеством.

Теорема 6. В любом хордальном графе имеется симплициальная вершина.

Доказательство. В полном графе любая вершина является симплициальной. Докажем индукцией по числу вершин n, что в любом хордальном графе, не являющемся полным, есть

две несмежные симплициальные вершины. При n = 2 это, очевидно, так. Пусть G - хордальный граф с n вершинами, n > 2, не являющийся полным. Если G несвязен, то, по предположению индукции, во всех компонентах связности есть симплициальные вершины. Допустим, что граф G связен. Так как он не полный, то в нем есть разделяющее множество, а по теореме 5 есть разделяющая клика. Пусть C - такая клика, A и B - две новые компоненты связности, появляющиеся при удалении из графа всех вершин клики C. Рассмотрим подграф G_A, порожденный множеством A U C. Если он полный, то в нем любая вершина симплициальна. Если же он не полный, то по предположению индукции в нем есть две несмежные симплициальные вершины. Хотя бы одна из этих двух вершин принадлежит множеству A. Итак, в любом случае в множестве A имеется вершина a, являющаяся симплициальной в графе G_A. Окрестность вершины a во всем графе G совпадает с ее окрестностью в подграфе G_A. Следовательно, a - симплициальная вершина графа G. Аналогично, в множестве B имеется симплициальная вершина графа G и она не смежна с вершиной a.

Существование симплициальных вершин можно использовать для создания эффективного алгоритма раскрашивания хордального графа в наименьшее число цветов. План такого алгоритма содержится в доказательстве следующей теоремы:

Теорема 7. Для любого хордального графа $\chi(G) = \omega(G)$.

Доказательство. Пусть G- хордальный граф с n вершинами и $\omega(G) = k$. Покажем, что граф G можно правильно раскрасить в k цветов. Найдем в нем симплициальную вершину и обозначим ее через x_n, а граф, полученный удалением этой вершины, через G_{n-1}. Этот граф тоже хордальный, значит, в нем тоже есть *симплициальная* вершина. Пусть x_{n-1}- *симплициальная* вершина в графе G_{n-1}, а G_{n-2}- граф, получаемый из него удалением этой вершины. Продолжая действовать таким образом, получим последовательность вершин $x_n , x_{n-1} , \ldots, x_1$ и последовательность графов $G_n , G_{n-1} , \ldots , G_1$, (здесь $G_n = G$), причем при каждом i вершина x_i является *симплициальной* в графе G_i, а граф G_{i-1} получается из G_i удалением этой вершины.

Допустим, что граф G_{i-1} правильно раскрашен в k цветов. Покажем, что вершину x_i можно покрасить в один из этих цветов, сохраняя правильность раскраски. Действительно, x_i – симплициальная вершина графа G_i, значит, множество C всех смежных с ней в этом графе вершин является кликой. Так как при добавлении к множеству C вершины x_i тоже получается клика, а мощность наибольшей клики в графе G равна k, то $|C| \leq k-1$. Значит, для окрашивания вершин множества C использовано не более k – 1 цвета. Поэтому для вершины x_i можно использовать один из оставшихся цветов.

Итак, каждый из графов G_i, а значит, и исходный граф G, можно правильно раскрасить в k цветов. Отсюда следует, что $\chi(G) \leq \omega(G)$. Обратное неравенство было установлено в ранее.

Одним из примеров использования алгоритмов раскраски является решение задачи о нахождении паросочетания и реберных покрытий.

Паросочетания и реберные покрытия. *Паросочетанием* в графе называется множество ребер, попарно не имеющих общих вершин. Задача о *паросочетании* состоит в том, чтобы в данном графе найти *паросочетание* с наибольшим числом ребер. Это число для графа G будем обозначать через $\pi(G)$. *Реберным покрытием* графа называется такое множество ребер, что всякая *вершина* графа инцидентна хотя бы одному из этих ребер. Наименьшее число ребер в реберном покрытии графа G обозначим через $\rho(G)$. Заметим, что реберное покрытие существует только для графов без изолированных вершин.

Определение паросочетания похоже на определение независимого множества вершин, паросочетание иногда так и называют - независимое множество ребер. Эта аналогия еще усиливается тесной связью между реберными покрытиями и паросочетаниями, подобно тому, как связаны между собой вершинные покрытия и независимые множества. Даже равенство, количественно выражающее эту связь, имеет точно такой же вид (напомним, что числа независимости $\alpha(G)$ и вершинного покрытия $\beta(G)$ связаны равенством $\alpha(G) + \beta(G) = n$).

Приводимое ниже доказательство этого факта имеет алгоритмическое значение, так как показывает, каким образом каждая из двух задач может быть сведена к другой.

Теорема 1. Для любого графа G с n вершинами, не имеющего *изолированных вершин*, справедливо равенство $\pi(G) + \rho(G) = n$.

Доказательство. Пусть M – наибольшее *паросочетание* в графе G. Обозначим через W множество всех вершин графа, не покрытых ребрами этого *паросочетания*. Тогда $|W| = n - 2\pi(G)$. Очевидно, что W - независимое множество (иначе M не было бы наибольшим). Выберем для каждой вершины из W какое-нибудь инцидентное ей ребро. Пусть F - множество всех выбранных ребер. Тогда M U F - реберное покрытие и $|M \cup F| = n - \pi(G)$, следовательно, $\rho(G) \le n - \pi(G)$.

Обратно, пусть C - наименьшее реберное *покрытие графа* G. Рассмотрим подграф H графа G , образованный ребрами этого покрытия. В графе H один из концов каждого ребра является вершиной степени 1 (ребро, каждая вершина которого инцидентна, по крайней мере, еще одному ребру, можно было бы удалить из C, и оставшиеся ребра по-прежнему покрывали бы все вершины). Отсюда следует, что каждая *компонента связности* графа H является звездой (звезда - это дерево, у которого не более одной вершины степени больше 1). Так как в любом лесе сумма количеств ребер и *компонент связности* равна числу вершин, то число *компонент связности* в графе H равно $n - \rho(G)$. Выбрав по одному ребру из каждой компоненты, получим *паросочетание*. Отсюда следует, что $\pi(G) \ge n - \rho(G)$.

Несмотря на такое сходство между "вершинными" и "реберными" вариантами независимых множеств и покрытий, имеется кардинальное различие в сложности соответствующих экстремальных задач. "Вершинные" задачи, как уже отмечалось, являются NP-полными. Для реберных же известны полиномиальные алгоритмы. Они основаны на методе чередующихся путей, к рассмотрению которого мы теперь переходим. Отметим только еще, что ситуация похожа на ту, что наблюдается для задач об эйлеровом и гамильтоновом циклах - реберный вариант эффективно решается, а вершинный является NP-полным.

Метод увеличивающих путей. Пусть G - *граф*, M – некоторое *паросочетание* в нем. Ребра *паросочетания* будем называть *сильными*, остальные ребра графа - *слабыми*. Вершину назовем *свободной*, если она не принадлежит ребру *паросочетания*. На рисунке 6.6 слева показан *граф* и в нем выделены ребра *паросочетания* M = {(1,2),(6,8),(3,7),(9,10)}. Вершины 4 и 5 - свободные. Заметим, что к этому паросочетанию нельзя добавить ни одного ребра, т.е. оно максимальное. Однако оно не является наибольшим. В этом легко убедиться, если рассмотреть путь5,6,8,9,10,7,3,4 (показан пунктиром). Он начинается и заканчивается в свободных вершинах, а вдоль пути чередуются сильные и слабые ребра. Если на этом пути превратить каждое сильное *ребро* в слабое, а каждое слабое - в сильное, то получится новое *паросочетание*, показанное на рисунке справа, в котором на одно *ребро* больше. Увеличение *паросочетания* с помощью подобных преобразований - в этом суть метода увеличивающих путей.

Рисунок 6.6. – Паросочетания графа

Сформулируем необходимые понятия и докажем теорему, лежащую в основе этого метода. *Чередующимся путем* относительно данного *паросочетания* называется простой *путь*, в котором чередуются сильные и слабые ребра (т.е. за сильным ребром следует слабое, за слабым - сильное). Чередующийся *путь* называется *увеличивающим*, если он соединяет две свободные вершины. Если M - *паросочетание*, P - увеличивающий *путь* относительно M, то легко видеть, что *паросочетанием* являются также:

$$M \oplus P$$
$$|M \oplus P| = |M| + 1$$

Теорема 2. *Паросочетание* является наибольшим тогда и только тогда, когда относительно него нет увеличивающих путей.

Доказательство. Если есть увеличивающий путь, то, поступая так, как в рассмотренном примере, то-есть заменяя вдоль этого пути сильные ребра на слабые и наоборот, получим большее *паросочетание*. Для доказательства обратного утверждения рассмотрим *паросочетание* M в графе G и предположим, что M - не наибольшее. Покажем, что тогда имеется увеличивающий путь относительно M. Пусть M'- другое *паросочетание* и | M '| > | M |. Рассмотрим подграф H графа G, образованный теми ребрами, которые входят в одно и только в одно из паросочетаний M, M '. Иначе говоря, множеством ребер графа H является симметрическая разность M и M '. В графе H каждая вершина инцидентна не более чем двум ребрам (одному из M и одному из M '), т.е. имеет степень не более двух. В таком графе каждая *компонента связности* путь или цикл. В каждом из этих путей и циклов чередуются ребра из M и M '. Так как | M '| > | M, имеется компонента, в которой ребер из M ' содержится больше, чем ребер из M. Это может быть только путь, у которого оба концевых ребра принадлежат M '. Легко видеть, что относительно M этот путь будет увеличивающим.

Для решения задачи о *паросочетании* остается научиться находить увеличивающие пути или убеждаться, что таких путей нет. Тогда, начиная с любого *паросочетания* (можно и с пустого *множества* ребер), можем строить *паросочетания* со все увеличивающимся количеством ребер до тех пор, пока не получим такое, относительно которого нет увеличивающих путей. Оно и будет наибольшим. Известны эффективные алгоритмы, которые ищут увеличивающие пути для произвольных графов. Рассмотрим сначала более простой *алгоритм*, решающий эту задачу для *двудольных графов*.

Паросочетания в двудольных графах. Пусть G=(A,B,E) - *двудольный граф* с долями A и B, M – *паросочетание* в G. Всякий увеличивающий *путь*, если такой имеется, соединяет вершину из *множества* A с вершиной из *множества* B.

Зафиксируем некоторую свободную вершину a из A. Пытаемся найти увеличивающий *путь*, начинающийся в a , либо убедимся в том, что таких путей нет. Оказывается, нет необходимости рассматривать все чередующиеся пути, начинающиеся в вершине a, для того, чтобы установить, какие вершины достижимы из вершины a чередующимися путями.

Вершину x назовем *четной* или *нечетной* в зависимости от того, четно или нечетно *расстояние* между нею и вершиной a. Поскольку *граф двудольный*, любой *путь*, соединяющий вершину a с четной (нечетной) вершиной, имеет четную (нечетную) длину. Поэтому в чередующемся пути, ведущем из вершины a в четную (нечетную) вершину, последнее *ребро* обязательно сильное (слабое).

Определим дерево достижимости как максимальное дерево с корнем a, в котором каждый путь, начинающийся в корне, является чередующимся. Дерево достижимости определено не однозначно, но любое такое дерево в двудольном графе обладает следующим свойством, сформулированным в лемме 7.1.

Лемма 7.1. Вершина x принадлежит дереву достижимости тогда и только тогда, когда существует чередующийся путь, соединяющий вершины a и x.

Доказательство. Рассмотрим некоторое дерево достижимости T и докажем, что всякая вершина x , достижимая из вершины a чередующимся путем, принадлежит этому дереву. Проведем индукцию по длине кратчайшего чередующегося пути из a в x. Пусть y- предпоследняя (т.е. предшествующая x) вершина такого пути. По предположению индукции, вершина y принадлежит дереву T. Если она четная, то любой чередующийся путь из вершины a в вершину y заканчивается сильным ребром. Следовательно, в дереве T вершину y с ее отцом соединяет сильное ребро, а ребро (x,y) - слабое. Поэтому, если добавить к дереву вершину x и ребро (x,y), то путь в дереве, соединяющий a с x, будет чередующимся. Значит, если предположить, что дерево T не содержит вершины x, то окажется, что оно не максимально, а это

противоречит определению. Аналогично рассматривается случай, когда вершина y- нечетная.

Итак, для решения задачи остается научиться **строить дерево достижимости**. Для этого можно использовать слегка модифицированный поиск в ширину из вершины a. Отличие от стандартного поиска в ширину состоит в том, что открываемые вершины делятся на четные и нечетные. Для четных вершин исследуются инцидентные им слабые ребра, а для нечетных - сильные. Через V(x), как обычно, обозначается множество вершин, смежных с вершиной x, Q - очередь, используемая при поиске в ширину. Если вершина x не является свободной, т.е. инцидентна некоторому сильному ребру, то другая вершина этого ребра обозначается через p(x). Ниже приведен пример реализации данного алгоритма.

Алгоритм построения дерева достижимости:

1. объявить все вершины новыми
2. объявить вершину a четной
3. a →Q
4. создать дерево T из одной вершины a
5. **while Q ≠ ∅ do**
6. x←Q
7. **if** вершина x нечетная
8. **then if** вершина x не свободная
9. **then y = p(x)**
10. y→Q
11. объявить вершину y четной
12. добавить к дереву T вершину y и ребро (x,y)
13. **else for y из V(x) do**
14. **if** вершина y новая
15. **then** y→Q
16. объявить вершину y нечетной
17. добавить к T вершину y и ребро (x,y)

Если очередная рассматриваемая *вершина* x оказывается свободной (это выясняется при проверке в строке 8), нет необходимости доводить построение дерева до конца. В этом случае *путь* между вершинами a и x в дереве является увеличивающим путем и его можно использовать для построения большего *паросочетания*. После этого снова выбирается свободная *вершина* (если такая еще есть) и строится *дерево* достижимости. В приведенном тексте алгоритма соответствующий *выход* отсутствует, но его легко предусмотреть, добавив *ветвь* **else** к оператору **if** в строке 8.

Если дерево построено и в нем нет других свободных вершин, кроме корня, то нужно выбрать другую свободную вершину и построить дерево достижимости для нее (если в графе больше двух свободных вершин). При этом, как показывает следующая лемма, вершины первого дерева можно удалить.

Лемма 7.2. Если дерево достижимости содержит хотя бы одну вершину увеличивающего пути, то оно содержит увеличивающий путь.

Доказательство. Пусть дерево достижимости T с корнем a имеет общие вершины с увеличивающим путем P, соединяющим свободные вершины b и c. Покажем, что в T есть увеличивающий путь (это не обязательно путь P). Достаточно доказать, что хотя бы одна из вершин b, c принадлежит дереву T. Если одна из них совпадает с вершиной a, то из теоремы 1 следует, что и другая принадлежит дереву, так что в этом случае в дереве имеется увеличивающий путь между вершинами b и c. Допустим, обе вершины b и c отличны от a. Пусть R-простой путь, начинающийся в вершине a, заканчивающийся в вершине x, принадлежащей пути P, и не содержащий других вершин пути P. Очевидно, что последнее ребро пути R слабое. Вершина x делит путь P на два отрезка – P_a и P_c, - содержащие соответственно вершины b и c. В одном из этих отрезков, скажем, в P_b, ребро, инцидентное вершине x, – сильное. То-

гда объединение путей R и P_b образует чередующийся путь, соединяющий вершины a и b. По теореме 1, вершина b принадлежит дереву T.

Из этой леммы следует, что если полностью построенное дерево достижимости не содержит других свободных вершин, кроме корня, то ни одна вершина этого дерева не принадлежит никакому увеличивающему пути. Поэтому, приступая к построению следующего дерева достижимости, вершины первого дерева можно временно удалить из графа. Этот процесс - построение деревьев достижимости и удаление их вершин из графа - продолжается до тех пор, пока либо будет найден увеличивающий путь, либо останется граф не более чем с одной свободной вершиной. В первом случае паросочетание увеличивается, граф восстанавливается, и вновь начинается поиск увеличивающего пути. Во втором случае полученное паросочетание является наибольшим.

Так как при поиске в ширину каждое ребро исследуется не более чем дважды, то общее время поиска увеличивающего пути для данного паросочетания – O(m). Число ребер в паросочетании не может превышать n/2, поэтому общая сложность алгоритма – O(m*n).

Паросочетания в произвольных графах (алгоритм Эдмондса). Для графа, не являющегося двудольным, утверждение леммы 1 может быть неверным. Пример этого показан на рисунке 6.7.

Рисунок 6.7. – Пример графа с нечетным циклом, для которого не найден увеличивающий путь

В графе, изображенном слева, с *паросочетанием* из двух ребер, имеется увеличивающий *путь*5,3,1,2,4,6. Справа показано *дерево достижимости*, построенное для вершины 5. *Вершина* 6 не вошла в это *дерево*, хотя имеется чередующийся *путь*, соединяющий ее с вершиной 5. В результате увеличивающий *путь* не будет найден. Причина этого - наличие в графе ребра (1,2), соединяющего вершины, находящиеся на одинаковом расстоянии от корня дерева. В тот момент, когда исследуется это *ребро*, обе вершины 1 и 2 уже присутствуют в дереве, поэтому построение дерева заканчивается. Наличие такого ребра означает, что в графе есть нечетный цикл.

Тем не менее, и для графов с нечетными циклами задачу о наибольшем паросочетании можно решать эффективно. Рассмотрим алгоритм построения наибольшего паросочетания в произвольном графе, предложенный Эдмондсом.

Сначала, как и в случае двудольного графа, методом поиска в ширину строится дерево достижимости для некоторой свободной вершины a. Построение дерева для двудольного графа прекращалось, если к дереву нельзя было добавить ни одной вершины, либо если к нему добавлялась свободная вершина, т.е. обнаруживался увеличивающий путь. Для произвольного графа будем прекращать построение дерева еще и в том случае, когда исследуемое ребро соединяет две четные вершины дерева. При поиске в ширину это может быть только тогда, когда эти две вершины находятся на одинаковом расстоянии от корня. Обнаружение такого ребра означает, что найден подграф, называемый цветком (рисунок 6.8). Он состоит из чередующегося пути P, соединяющего корень дерева a с некоторой вершиной b, и нечетного цикла C. При этом b является единственной общей вершиной пути P и цикла C, а C можно рассматривать как замкнутый чередующийся путь, начинающийся и заканчивающийся в вершине b. На рисунке показаны также смежные четные вершины x и y, находящиеся на одинаковом расстоянии от вершины a.

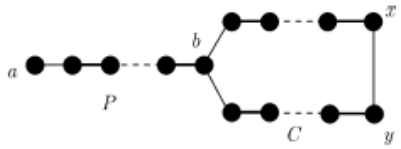


Рисунок 6.8. Пример графа с подграфом-цветком.

5

Выявление цветков не представляет трудности - нужно только добавить ветвь else к оператору if в строке 14 алгоритма 1. Первое, что мы сделаем, обнаружив цветок, - превратим все сильные ребра пути P в слабые, а слабые - в сильные. После этого преобразования множество сильных ребер является паросочетанием той же мощности, но вместо вершины a свободной вершиной станет вершина b. Таким образом, на цикле C будет одна свободная вершина и этот цикл является чередующимся путем, который начинается и заканчивается в этой вершине. Покажем, что такой цикл можно стянуть в одну вершину, не теряя информации о существовании увеличивающих путей.

Операция стягивания подграфа H в графе G состоит в следующем. Все вершины этого подграфа удаляются из графа, а вместо них добавляется новая вершина. Эта новая вершина соединяется ребрами с теми и только теми из оставшихся вершин графа, которые были смежны хотя бы с одной вершиной подграфа H. Полученный граф будем обозначать G/H.

Теорема 3. Пусть M – паросочетание в графе G, C - цикл длины 2k+1 в этом графе, причем на цикле имеется k сильных ребер и одна свободная вершина. Пусть M' – паросочетание в графе G' = G/C, составленное из всех ребер паросочетания M, не принадлежащих циклу C. Паросочетание M является наибольшим в графе G тогда и только тогда, когда M' – наибольшее паросочетание в графе G'.

Доказательство. Докажем, что из существования увеличивающего пути относительно паросочетания M в графе G следует, что существует увеличивающий путь относительно паросочетания M' в графе G' и обратно.

Пусть b - свободная вершина цикла C. Новую вершину, образованную в графе G' при стягивании цикла C, обозначим через c. Отметим, что она является свободной вершиной относительно паросочетания M'.

Пусть P - увеличивающий путь в графе G. Если он не содержит вершин цикла C, то он будет увеличивающим путем и в графе G'. В противном случае рассмотрим отрезок P' пути P, начинающийся в свободной вершине, отличной от b, заканчивающийся в вершине x, лежащей на цикле C, и не содержащий других вершин цикла C. Если в пути P' заменить вершину x вершиной c, то, очевидно, получится увеличивающий путь в графе G'.

Обратно, пусть P' - увеличивающий путь в графе G'. Если P' не проходит через вершину c , то он будет увеличивающим путем и в графе G. В противном случае рассмотрим путь P'', получающийся удалением вершины c из пути P'. Можно считать, что вершина c была последней вершиной пути P', а путь P'' заканчивается в предпоследней вершине x. Так как вершина x смежна с вершиной c в графе G', то в графе G на цикле C существует вершина y, смежная с x. Добавим к пути P'' тот из отрезков цикла C, соединяющих вершину y с вершиной b, который начинается сильным ребром. В результате получится увеличивающий путь в графе G.

Из доказательства видно, что увеличивающий путь в графе G при известном увеличивающем пути в графе G' находится за линейное время. Для получения оценок времени работы алгоритма в целом требуется еще проработка ряда деталей, например подробностей выполнения операции стягивания и т.д.

Тестовые вопросы по теме

1. Перечислить способы нахождения хроматического числа.
2. Дать определение хроматического индекса.
3. Какая раскраска вершин называеся правильной? Минимальной?
4. Какие виды перекрасок ребер Вы знаете?

5. Чему равно хроматическое число графа K_6?

6. Какая из задач нахождения минимальной правильной раскраски в графе является наиболее сложной по времени? Обоснуйте ответ.

7. Какой граф называется хордальным?

8. Понятие и примеры смежно и несмежно поглощающих вершин.

9. Какая вершина графа называется симплициальной?

10. Определить хроматический индекс графа C_5, C_{18}, O_8/

11. Какой граф называется К-раскрашиваемым?+

12. Как определяется хроматический класс графа?

13. Сформулировать задачу о четырех красках?

Рекомендуемые источники по теме 6

1 Кристофидес, Н. Теория графов. Алгоритмический подход. – М.: Мир, 1978.

7 Алгоритмы нахождения максимального потока в графе

Лабораторная работа № 7
(работа в аудитории 6 ч., СРС 6 ч.)

Цели работы

– закрепление теоретических знаний по теме работы;

– углубление навыков по применению методов разработки программного обеспечения;

– углубление навыков оценки временной и емкостной сложности программного обеспечения;

– углубление навыков чтения, понимания и выделения главной идеи прочитанного исходного кода;

– приобретение практических навыков в разработке компьютерных программ, реализующих алгоритмы нахождения максимального потока в графе

Задание

Перед выполнением работы необходимо повторить теоретический материал по теме. Основные теоретические положения изложены ниже в данном параграфе пункте «Теоретические сведения». После этого следует ответить на тестовые вопросы по теме.

В результате выполнения лабораторной работы требуется написать и отладить программу на алгоритмическом языке (С++ или ином), которая получает на входе числовые данные, выполняет их обработку в соответствии с требованиями задания и выводит результат на экран в графической форме. Для обработки данных необходимо реализовать алгоритмы обхода графа в соответствии с постановкой задачи. Ввод данных осуществляется из файла с учетом требований к входным данным, с учетом постановки задачи. Ограничениями на входные данные является допустимый диапазон значений используемых числовых типов в выбранной системе программирования. Работа состоит из этапов.

- Выбрать алгоритм решения поставленной задачи по заданному варианту (варианты заданий приведены ниже).
- Разработать тестовые примеры для отладки соответствующего алгоритма и программы. Согласовать с преподавателем.
- Разработать и обосновать структуры данных для реализации алгоритма.
- С учетом выполнения предыдущих пунктов задания написать и отладить программу на множестве тестовых примеров.

- Оформить отчет по выполненной работе.
- Защитить работу, ответив на вопросы преподавателя и студентов.

Варианты заданий

1. Найти максимальный поток методом Форда-Фалкерсона. Найти минимальный разрез.
2. Найти максимальный поток методом Форда-Фалкерсона. Найти все разрезы и их пропускную способность
3. Найти максимальный поток методом Форда-Фалкерсона. Указать остаточную сеть.
4. Найти максимальный поток методом Форда-Фалкерсона. Указать дополняющие пути.
5. Найти максимальный поток методом Форда-Фалкерсона. Указать минимальную и максимальную пропускные способности.
6. Найти максимальный поток методом Форда-Фалкерсона. Найти минимальный и максимальный разрезы.
7. Найти максимальный поток методом Эдмондса-Карпа и минимальный разрез.
8. Найти максимальный поток методом Эдмондса-Карпа и длины всех дополняющих путей.
9. Найти методом Форда-Фалкерсона максимальное паросочетание в двудольном графе.
10. Найти максимальный поток методом проталкивания предпотока. Найти число вершину с максимальной высотой
11. Найти максимальный поток методом проталкивания предпотока. Найти число подъемов. Найти максимальный поток методом проталкивания предпотока. Напечатать высоты вершин на каждом шаге.
12. Найти максимальный поток методом проталкивания предпотока и подсчитать число насыщающих проталкиваний.
13. Найти максимальный поток методом проталкивания предпотока и подсчитать число ненасыщающих проталкиваний.
14. Найти максимальный поток методом проталкивания предпотока и подсчитать общее число операций подъема и проталкивания.
15. Найти максимальный поток методом проталкивания предпотока и найти минимальный разрез в сети.
16. Найти максимальный поток методом проталкивания предпотока и расстояние между заданными вершинами в остаточной сети.
17. Найти максимальный поток методом "поднять и в начало".
18. Найти максимальный поток методом "поднять и в начало". Перечислить допустимые ребра.
19. Найти максимальный поток методом "поднять и в начало". Напечатать списки соседей.
20. Найти максимальный поток методом "поднять и в начало". Перечислить все переполнявшиеся вершины.
21. Эксперименты в космосе(Кормен, стр.575)
22. Найти максимальный поток методом "поднять и в начало". Увеличить пропускную способность заданного ребра на 1 и подсчитать максимальный поток. Сравнить число операций.
23. Найти максимальный поток методом "поднять и в начало". Уменьшить пропускную способность заданного ребра на 1 и подсчитать максимальный поток. Сравнить число операций.
24. Найти максимальный поток методом "поднять и в начало". Увеличить пропускную способность заданного ребра в с раз и подсчитать максимальный поток. Сравнить число операций.

Теоретические сведения

Каждое ориентированное ребро графа можно рассматривать как канал, по которому движется продукт. Каждый канал имеет заданную пропускную способность, которая харак-

теризует максимальную скорость перемещения продукта по каналу, например 200 литров жидкости в минуту для трубопровода. Вершины являются точками пересечения каналов; через вершины, отличные от истока и стока, продукт проходит, не накапливаясь. Иными словами, скорость поступления продукта в вершину должна быть равна скорости его удаления из вершины. Это свойство называется свойством сохранения потока.

В задаче о максимальном потоке мы хотим найти максимальную скорость пересылки продукта от истока к стоку, при которой не будут нарушаться ограничения пропускной способности. Это одна из простейших задач, возникающих в транспортных сетях, и, как будет показано далее, существуют эффективные алгоритмы ее решения.

Рассмотрим несколько алгоритмов решения задачи: классический метод Форда-Фалкерсона (Ford- Fulkerson) для поиска максимального потока. Дополнительно рассмотрим метод проталкивания предпотока ("push-relabel"), который лежит в основе многих наиболее быстрых алгоритмов для решения задач в транспортных сетях. и метод «поднять и в начало».

Сети и потоки. Пусть G = (V, E) представляет собой ориентированный граф, в котором каждое ребро (u, v) принадлежит E имеет неотрицательную пропускную способность (capacity) c(и, v) > 0. Далее мы потребуем, чтобы в случае, если E содержит ребро (u, v), обратного ребра (v, и) не было (вскоре мы увидим, как обойти это ограничение). Если (и, v) не принадлежит E, то c(u,v) = 0, а также запретим петли. В транспортной сети выделяются две вершины: исток (source) s и сток (sink) t. Для удобства предполагается, что каждая вершина лежит на пути от истока к стоку, т.е. для любой вершины транспортная сеть содержит путь s в t, содержащий эту вершину. Таким образом, граф является связным и, поскольку каждая вершина, отличная от s, содержит как минимум одно входящее ребро На рис. 7.1 показан пример сети, которую еще называют транспортной сетью.

На рисунке 7.1. (а) представлена сеть ***G*** = (V, *E)* для задачи о грузовых перевозках. Исток s, сток t. На каждом ребре указано максимальное число единиц груза, например, ящиков, которое можно отправить за день. На рис. 7.1 (б) дан пример потока f сети G (до косой черты на ребрах изображен поток, после – пропускная способность).

Потоком (flow) в сети ***G*** является действительная функция f:*V* x *V* —» R, удовлетворяющая следующим трем условиям.

Ограничение пропускной способности. Для всех *u, v* *V* должно выполняться условие: **0 <** $f(u, v) < c(u, v)$**.**

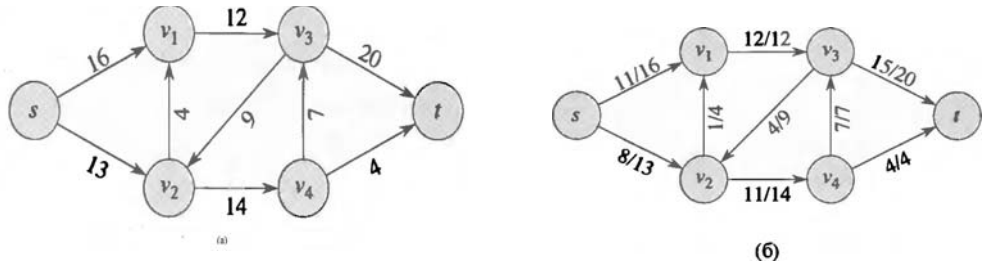


Рисунок 7.1 – Варианты сетей грузовых превозок

Сохранение потока. Для всех *u* *V* — {s, *t*} должно выполняться:

$$\sum_{v \in V} f(u, v) = 0$$

Кососимметричность: f(u,v) = - f(v,u) для всех u, v из V.

Величина f(u,v) определяет, сколько вещества движется из u в v.

Величина потока из истока определяется как:

$$\sum_{v \in V} f(s, v).$$

Задача о максимальном потоке формулируется так: для данной сети G с истоком s и стоком t найти поток максимальной величины.

Сети с несколькими истоками и стоками

В задаче о максимальном потоке может быть несколько истоков и стоков. Например, у компании может быть ***m*** фабрик {s1, ***s2,***..., ***sm}*** и ***n*** складов ***{t1,t2,…,tn},*** как показано на рис.7.2(а), на котором приведен пример транспортной сети с пятью истоками и тремя стоками. К счастью, эта задача не сложнее, чем обычная задача о максимальном потоке.

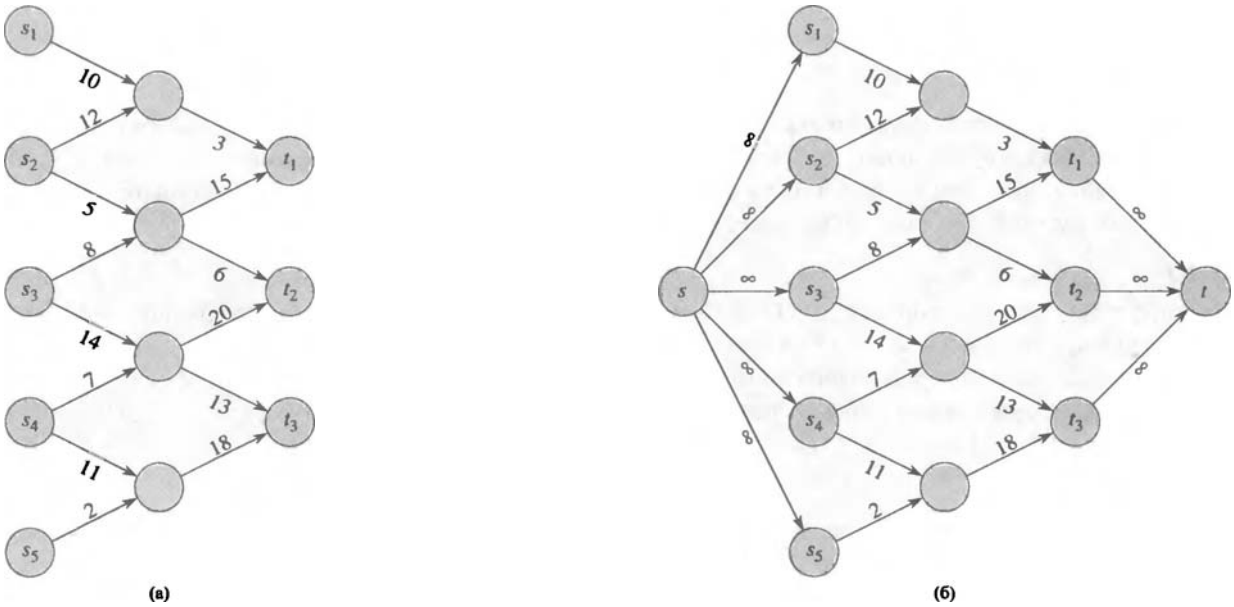


Рисунок 7.2. – Преобразование задачи о максимальном потоке с несколькими истоками и стоками в задачу с одним истоком и одним стоком

На рис.7.2 (а) – транспортная сеть с пятью истоками S = {s1, *s2, s3, s*4, s5} и тремя стоками T = {t1, t2, t3}. На рис. 7.2 (б) – эквивалентная сеть с одним истоком и одним стоком. Добавлен фиктивный исток *s* и ребра с бесконечной пропускной способностью от *s* до каждого из исходных истоков. Кроме того, мы добавляем фиктивный сток t и ребра с бесконечной пропускной способностью из каждого из исходных стоков в t

Метод Форда Фалкерсона. Рассмотрим метод Форда-Фалкерсона для решения задачи о максимальном потоке. Мы называем его методом, а не алгоритмом, поскольку он допускает несколько реализаций с различным временем выполнения. Метод Форда-Фалкерсона базируется на трех важных идеях, это — остаточные сети, увеличивающие пути и разрезы. Данные концепции лежат в основе важной теоремы о максимальном потоке и минимальном разрезе, которая определяет значение максимального потока через разрезы сети.

Поиск максимального потока методом Форда-Фалкерсона выполняется по шагам. В начале поток нулевой. На каждом шаге величина потока увеличивается. Для этого находим «дополняющий путь», по которому можно пропустить еще немного вещества, увеличивая поток. Этот процесс повторяется, пока находятся дополняющие пути. Теорема о максимальном потоке и минимальном разрезе показывает, что найденный поток – максимальный.

FORD-FULKERSON-METHOD (G, *s*, *t)*

1 Инициализация потока f нулевым значением
2 **while** существует увеличивающий путь ***p*** в остаточ ce ***G***
3 увеличиваем поток f вдоль пути ***p***
4 **return f**

Остаточные сети. Пусть есть транспортная се G (***V***, ***E)*** с истоком ***s*** и стоком ***t.*** Пусть в G имеется поток f, и рассмотрим пару вершин ***u,v*** ∈ ***V.***

Определим ***остаточную пропускную способность*** $C_f(u,v)$ *как*

$$C_f(u,v) = c(u,v) - f(u,v).$$

Определение. Сеть $G_f = (V,E)$, где $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$, назовем остаточной сетью сети G, порожденной потоком f. Ее ребра, называемые остаточными ребрами допускают положительный поток.

Отметим, что остаточное ребро (u,v) не обязано быть ребром сети G. Ребер (v1,s), (v2,v3) на рисунке 7.4 в исходной сети не было.

Такая сеть определяет, сколько еще вещества можно направить из u в v.

Рассмотрим примеры, в которых на рисунках 7.3-7.6 поясняются варианты построения остаточных сетей**.**

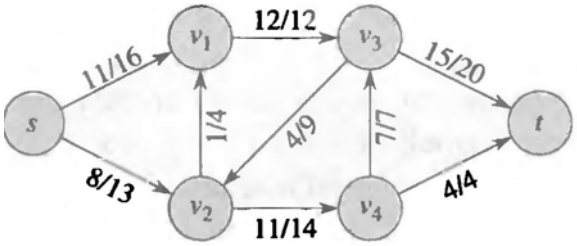


Рисунок 7.3.– Поток f сети G (см. рисунок 7.1)

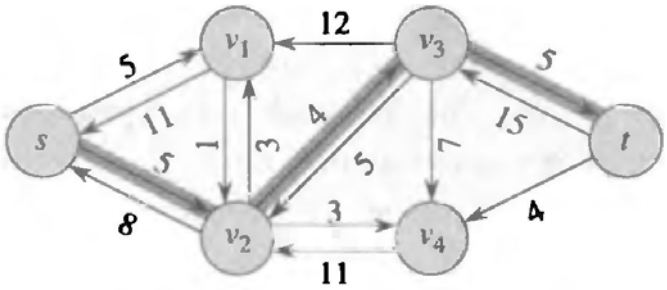


Рисунок 7.4. – Остаточная сеть G_f. Выделен дополняющий путь p с остаточной пропускной способностью $c_f = c(v_2,v_3) = 4$.

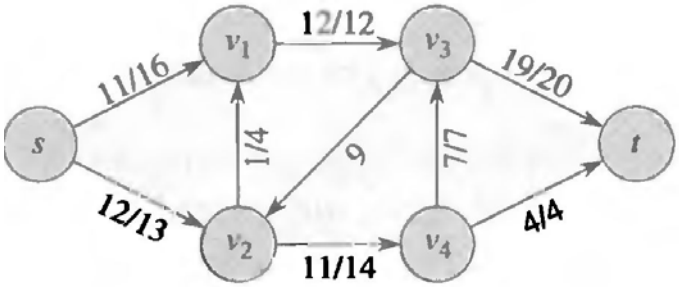


Рисунок 7.5. – Результат добавления потока величины 4 по пути p.

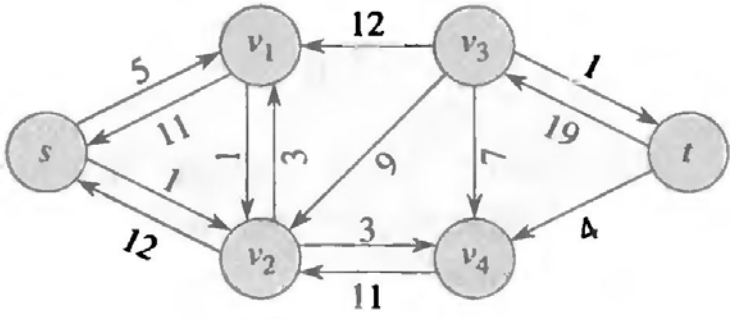


Рисунок 7.6. – Остаточная сеть, порожденная потоком рисунка 7.5.

Лемма 7.1. Пусть G = (V,E) – сеть с истоком s и стоком t, а f – поток в ней. Пусть G_f – остаточная сеть сети G, порожденная потоком f. Пусть f' – поток в G_f. Тогда сумма f + f' – поток в сети G величины |f + f'| = |f| + |f'|.

Доказательство. Сначала убедимся, что f + f' будет потоком. Проверим кососимметричность. Для всех u,v ∈V выполнено

(f + f')(u,v) = f(u,v) + f'(u,v) = - f(v,u) - f'(v,u) = -(f(v,u)+ f'(v,u)) = -(f + f')(v,u).

Проверим условие, связанное с ограниченной пропускной способностью. Заметим, что f'(v,u) ≤ cf(u,v) для всех u,v ∈V, поэтому (f + f')(u,v) = f(u,v) + f'(u,v) ≤ f(u,v) +(c(u,v) – f(u,v))= c(u,v).

Проверим закон сохранения потока. Для всех u ∈ V \ {s,t} выполнено равенство (f + f')(u,V) = f(u,V) + f'(u,V) = 0+0=0.

Наконец, найдем суммарный поток:

|f + f'| = (f + f')(s,V) = f(s,V) + f'(s,V) =|f| + |f'| (7.1)

Дополняющие пути. Для заданных транспортной сети ***G = (V,E)*** и потока f ***дополняющим путем*** является простой путь из истока ***s*** в сток ***t*** в остаточной сети ***Gf.*** Согласно определению остаточной сети мы можем увеличить поток в ребре (u, ***v***) дополняющего пути до ***Cf{u,v***) без нарушения ограничения пропускной способности соответствующего ребра в исходной сети.

Выделенный путь на рисунке 7.4, является дополняющим путем. Рассматривая представленную на рисунке остаточную сеть ***G***f как некоторую транспортную сеть, можно увеличивать поток вдоль каждого ребра данного пути вплоть до четырех единиц, не нарушая ограничений пропускной способности, поскольку наименьшая остаточная пропускная способность на данном пути составляет ***Cf(v***2, vз) = 4.

Максимальная величина, на которую можно увеличить поток в каждом ребре дополняющего пути ***p,*** называется ***остаточной пропускной способностью*** (residual capacity) пути ***p*** и задается формулой

Cf(p) = min{cf(u, ***v)*** : ***(u,v)*** принадлежит ***p}*** .

Следующая лемма, доказательство которой предлагается провести в качестве упражнения, более строго формулирует приведенные выше рассуждения.

Лемма 7.2. Пусть ***G*** = (V, ***E)*** является сетью, а f представляет собой поток в ***G,*** и пусть ***p*** является дополняющим путем в ***Gf.*** Определим функцию fp : ***V*** x ***V*** -> R так

$$f_p(u,v)=\begin{cases} c_f(p), & \text{если } (u,v)\in p \\ -c_f(p), & \text{если } (v,u)\in p \\ 0, & \text{в остальных случаях} \end{cases} \qquad (7.2)$$

Тогда f_p – поток в сети G_f и $|f_p| = c_f(p) > 0$.

Теперь ясно, что если добавить поток f_p к потоку f, получится поток в сети G с большим значением.

Следствие 7.1. Пусть f – поток в сети G=(V,E), а p – дополняющий путь в сети G_f. Рассмотрим поток f_p, заданный соотношением 7.2. Тогда функция

f' = f + f_p является потоком в сети G величины |f'| = |f| +|fp|>|f|.

Доказательство. Следует из лемм 7.1 и 7.2.

Разрезы в сетях. В методе Форда-Фалкерсона проводится многократное увеличение потока вдоль дополняющих путей до тех пор, пока не будет найден максимальный поток. В

теореме о максимальном потоке и минимальном разрезе, которую мы вскоре докажем, утверждается, что поток является максимальным тогда и только тогда, когда его остаточная сеть не содержит дополняющих путей. Для доказательства теоремы необходимо ввести понятие разреза сети.

Разрезом (cut) сети ***G*** = (V, ***E)*** называется разбиение множества вершин ***V*** на множества ***S*** и ***T = V \ S,*** такие, что ***s*** ∈ ***S,*** а ***t*** ∈ ***T.*** ***Пропускной способностью*** (capacity) разреза (S, T) является сумма c(S,T) пропускных способностей пересекающих разрез ребер. Для заданного потока f величина потока через разрез (S,T) определяется как сумма f(S,T) по пересекающим разрез ребрам. ***Минимальным разрезом*** (minimum cut) сети является разрез, пропускная способность которого минимальна.

На рисунке 7.7 изображен разрез ({s,v1,v2},{v3,v4,t}) сети 7.1(б). Поток через этот разрез равен f(v1, v3)+ f(v2, v3)+ f(v2, v4) = 12+(-4)+11=19, а пропускная способность разреза равна c(v1, v3)+ c(v2, v4) = 12+14=26.

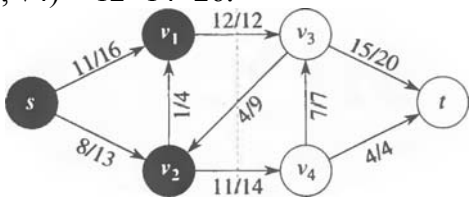


Рисунок 7.7. **Разрез (*S, T*) в сети, показанной на рис.7.1,(6), где *S* = {s,v1,v2}, а *T* — *{v3,v4, t}*. Вершины в *S* показаны черными, а вершины в *T* — белыми. Поток через разрез (S, *T)* равен *f(S, T)* = 19, а пропускная способность c*(S, T)* = 26**

Лемма 7.3. Пусть f – поток в сети G с истоком s и стоком t, а (S,T) – разрез сети G. Тогда поток f(S,T) через разрез (S,T) равен |f|.

Доказательство. Многократно используя лемму 7.1, получим f(S,T)= f(S,V)- f(S,S)= f(S,V)-= f(s,V)+f(S\s,V)=f(s,V)=|f|.

Доказанное выше равенство(величина потока равна потоку в сток) следует из этой леммы.

Следствие 7.2. Величина любого потока f в сети G не превосходит пропускной способности любого разреза сети G.

Доказательство. Пусть (S,T) – произвольный разрез сети G. В силу леммы 7.3. и ограничений на потоки по ребрам

$|f|= f(S,T)=\sum_{u\in S}\sum_{v\in T} f(u,v) \le \sum_{u\in S}\sum_{v\in T} c(u,v)$=c(S,T)/

Теорема 7.1. О максимальном потоке и минимальном разрезе.

Пусть f - поток в сети ***G = (V,E)*** с истоком ***s*** и стоком ***t,*** то следующие утверждения равносильны.

1. f - максимальный поток в сети ***G.***
2. Остаточная сеть G_f не содержит дополняющих путей.
3. |f| = ***c(S,T***) для некоторого разреза (S, ***T)*** сети ***G.***

Доказательство. (1) => (2). Рассуждая от противного, допустим, что f является максимальным потоком в G, но G_f содержит увеличивающий путь ***p.*** Тогда, согласно следствию 26.3, поток, полученный путем увеличения потока f на поток fp, где fp задается уравнением (7.1), представляет собой поток в G с величиной, строго большей, чем |f|, что противоречит предположению о том, что f является максимальным потоком.

(2) => (3): предположим, что G_f не содержит увеличивающего пути, т.е. что в G_f нет пути из ***s*** в ***t.*** Рассмотрим множество S = {v∈ V: в G_f существует путь из s в v}.

Пусть T=V\S. Очевидно, что s∈S, t∈T, так как в Gf нет пути из s в t. Поэтому пара (S, ***T) – разрез.*** *Ни для каких u∈S, v∈T ребро (u,v) не принадлежит E_f(иначе вершина v попала бы в в S). Поэтому f(u,v)=c(u,v). По лемме 7.3* |*f*|= f(S,T)=c(S,T).

(3) => (1): для любого разреза (S, ***T) выполнено*** $|f| \le c($S,T). Поэтому из равенства $|f| =$ c(S,T) следует, что поток f максимален.

Общая схема алгоритма Форда-Фалкерсона.

При выполнении каждой итерации метода Форда-Фалкерсона находится ***некоторый*** увеличивающий путь ***p*** и поток f увеличивается на пропускную способность пути p. Приведенная далее реализация данного метода вычисляет максимальный поток в транспортной сети $G = (V, E)$ путем обновления величины потока f (u, v). каждого ребра $(u, v) \in E$. Считаем, что $c(u, v)$ пропускные способности ребер и $c(u,v) = 0$, если (u,v) не является ребром.

```
0    for каждого ребра (u, v) графа
1      {f[u,v] = 0
2       f[v,u] = 0}
3      while существует путь p из s в t в остаточной сети G_f
4      do { c_f(p) = min {c_f(u, v) : (u, v) содержится в p}
5      for каждого ребра (u, v) пути p
6  do { f(u,v) = f(u,v) + c_f(p)
7       f(v,u) = - f(u,v) }
```

Метод Форда Фалкерсона следует рассмотренной схеме алгоритма. В строках 1-3 задаются начальные значения потока. В цикле (строки 3-7) на каждом шаге находится дополняющий путь p в G_f и увеличивается поток f. Если дополняющего пути нет, найденный поток максимален.

Анализ алгоритма Форда-Фалкерсона

Время работы зависит от способа нахождения пути p. При неудачном выборе метода поиска алгоритм может даже не завершиться: величина потока будет последовательно увеличиваться, но она не обязательно будет сходиться к максимуму. Если увеличивающий путь выбирается с использованием поиска в ширину, алгоритм выполняется за полиномиальное время. Прежде чем доказать этот результат, получим простую границу времени выполнения для случая, когда увеличивающий путь выбирается произвольным образом, а все значения пропускных способностей являются целыми числами.

Для указанного случая(пропускные способности – целые числа) Рассмотренный алгоритм выполняется за время O(E|f*|), где f* - максимальный поток. Действительно, выполнение строк 1-3 требует времени O(E). Цикл в строках 4-8 выполняется не более |f*| раз, так как после каждого выполнения величина потока увеличивается по крайней мере на единицу. Оценим время одного шага этого цикла, которое зависит от способа хранения данных о потоке. Рассмотрим ориентированный граф G' = (V,E'), в котором E' = {(u,v) : (u,v)∈E или (v,u) ∈E }. Каждому ребру будем приписывать поток и пропускную способность. Остаточная сеть для текущего потока состоит из тех ребер (u,v) графа G', для которых c(u,v) – f(u,v) ≠0. Поиск дополняющего пути в остаточной сети займет время O(E) = O(E'). Поэтому время работы будет O(E| f*|).

Из полученной оценки следует, что при небольших значениях |f*| и целых пропускных способностях время работы алгоритма невелико. Но при большом |f*| время работы может быть большим даже для простой сети, как показывает пример на рисунке 7.8.

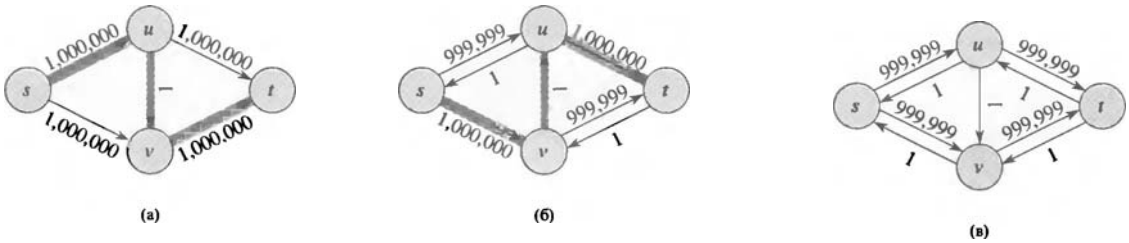


Рисунок 7.8. (а)Сеть, для обработки которой алгоритму **FORD-FULKERSON** может потребоваться время $O(E\,|f^*|)$, где f* - максимальный поток и |f*| = 2000000. Штриховкой выделен дополняющий путь с остаточной пропускной способностью 1. (б) Полученная остаточная сеть с другим дополняющим путем с остаточной пропускной способностью 1. (в) Полученная в результате остаточная сеть.

Можно получить лучшую оценку времени, если в строке 4 для поиска пути использовать поиск в ширину. Эта реализация метода Форда-Фалкерсона называется алгоритмом Эдмондса-Карпа. Можно доказать, что время работы алгоритма Эдмондса-Карпа равно O(VE2).

Алгоритм проталкивания предпотока

Рассмотрим подход к вычислению максимальных потоков, основанный на «проталкивании предпотока». В настоящее время многие асимптотически наиболее быстрые алгоритмы поиска максимального потока принадлежат данному классу, и на этом методе основаны реальные реализации алгоритмов поиска максимального потока. С помощью методов проталкивания предпотока можно решать и другие связанные с потоками задачи, например задачу поиска потока с минимальной стоимостью. Разработанный Голдбергом (Goldberg) “обобщенный” алгоритм поиска максимального потока, для которого существует простая реализация с временем выполнения $0(V^2E)$, что лучше времени работы алгоритма Эдмондса-Карпа $0(VE^2)$. Кроме того, рассмотрим усовершенствование этого обобщенного алгоритма, время выполнения которого составляет $0(V^3)$.

Алгоритмы проталкивания предпотока работают более локальным способом, чем метод Форда-Фалкерсона. Вместо того чтобы для поиска увеличивающего пути анализировать всю остаточную сеть, алгоритмы проталкивания предпотока обрабатывают вершины по одной, рассматривая только соседей данной вершины в остаточной сети. Кроме того, в отличие от метода Форда-Фалкерсона, алгоритмы проталкивания предпотока не поддерживают в процессе работы свойства сохранения потока. При этом, они поддерживают ***предпоток*** (preflow), который представляет собой функцию f: $V \times V \longrightarrow \boldsymbol{R}$, удовлетворяющую ограничениям пропускной способности и следующему ослабленному условию сохранения потока:

$f(V,u) \geq 0$ для всех вершин $u \in V \setminus \{s\}$. Таким образом, в каждой вершине u, кроме истока, есть некоторый неотрицательный избыток $e(u) = f(V,u)$.

Вершину, отличную от истока и стока, с положительным избытком назовем переполненной. Рассмотрим идею метода и опишем две основные операции: «проталкивание» предпотока и «подъем» вершины. Далее докажем правильность алгоритма и оценим время его работы.

Пусть ориентированные ребра соответствуют трубам. Вершины, в которых трубы пересекаются, обладают двумя интересными свойствами. Во-первых, чтобы принять избыточный поток, каждая вершина снабжена выходящей трубой, ведущей в произвольно большой резервуар, способный накапливать жидкость. Во-вторых, каждая вершина, ее резервуар и все трубные соединения находятся на платформе, высота которой увеличивается по мере работы алгоритма.

Высота вершины определяет, как проталкивается поток. Он может проталкиваться только вниз, т.е. от более высокой вершины к более низкой. Поток может быть направлен и

от нижестоящей вершины к вышестоящей, но операции проталкивания потока проталкивают его только вниз. Высота истока является фиксированной и составляет $|V|$, а фиксированная высота стока равна 0. Высота всех других вершин сначала равна 0 и увеличивается со временем. Алгоритм сначала посылает максимально возможный поток вниз от истока к стоку. Посылается количество, в точности достаточное для заполнения всех выходящих из истока труб до достижения их пропускной способности; таким образом посылается поток, равный пропускной способности разреза (s, V — {s}). Когда поток впервые входит в некоторую транзитную вершину, он накапливается в ее резервуаре. Отсюда он со временем проталкивается вниз.

Может случиться так, что все трубы, выходящие из вершины u и еще не заполненные потоком, ведут к вершинам, которые лежат на одном уровне с u или находятся выше нее. В этом случае, чтобы избавить переполненную вершину u от избыточного потока, необходимо увеличить ее высоту — провести операцию подъема (relabeling) вершины u. Ее высота увеличивается и становится на единицу больше, чем высота самой низкой из смежных с ней вершин, к которым ведут незаполненные трубы. Следовательно, после подъема вершины существует по крайней мере одна выходящая труба, по которой можно протолкнуть дополнительный поток.

В конечном итоге весь поток, который может пройти к стоку, оказывается там. Больше пройти не может, поскольку трубы подчиняются ограничениям пропускной способности. Чтобы сделать предпоток “нормальным” потоком, алгоритм после этого посылает избытки, содержащиеся в резервуарах переполненных вершин, обратно к истоку, продолжая менять метки вершин, чтобы их высота превышала фиксированную высоту истока $|V|$. Как будет показано, после того как резервуары окажутся пустыми, предпоток станет не только нормальным, но и максимальным потоком.

Основные операции

Как следует из предыдущих рассуждений, в алгоритме проталкивания предпотока выполняются две основные операции: проталкивание избытка потока от вершины к одной из соседних с ней вершин и подъем вершины. Применение этих операций зависит от высот вершин, которым мы сейчас дадим более точные определения.

Пусть $G = (V, E)$ представляет собой сеть с истоком s и стоком t, а f является предпотоком в графе. Функция является функцией высоты,
если $h(s) = |V|$, $h(t) = 0$ и $h(u) < h(v) + 1$ для каждого остаточного ребра (u, v) остаточной сети E_f. Сразу же можно сформулировать лемму 7.4.

Лемма 7.4. Пусть $G = (V, E)$ представляет собой сеть, а f является некоторым предпотоком в G, и пусть h — функция высоты, заданная на множестве V. Для любых двух вершин u и v графа справедливо следующее утверждение: если $h(u) > h(v) + 1$, то (u, v) не является ребром остаточной сети.

Определим основные операции.

Операция проталкивания

Основная операция **PUSH**(u, v) может применяться тогда, когда u является переполненной вершиной, $C_f(u, v) > 0$ и $h(u) = h(v) + 1$. Представленный ниже псевдокод обновляет предпоток f и избыточный поток для u и v;. Предполагается, что остаточные пропускные способности $C_f(u, v)$ при заданных f и c можно вычислить за фиксированное время. Излишний поток в вершине u –c[u], а высота вершины u — $h[u]$. Выражение $d_f(u, v)$ - временная переменная, в которой хранится количество потока, которое можно протолкнуть из u в v.

PUSH (u, v)

```
1                  // Применяется если вершина u переполнена,
//                 Cf(u, v) > 0 и h[u] = h[v] + 1.
2                  //Действие:  проталкивает df(u, v) = min(e[u], Cf(u, v))
//                 единиц потока из u в v.
3       df(u, v) = min(e[u], Cf(u, v))
```

4 $f(u, v) = f(u, v) + d_f(u,v)$
5 $f(v,u) = -f(u, v)$
6 $e[u] = e[u] - d_f(u,v)$
7 $e[v] = e[v] + d_f(u,v)$

Вершина u имеет положительный избыток $e[u]$ и остаточная пропускная способность ребра (u,v) положительна, поэтому можно увеличить поток из u в v на величину $d_f(u,v) = \min(e[u], C_f(u, v))$, при этом избыток $e[u]$ не становится отрицательным и не будет превышена пропускная способность $c(u,v)$. В строке 3 вычисляется значение $d_f(u,v)$, после чего в строках 4-5 обновляется f. В строке 4 увеличивается поток в ребре (u,v), поскольку мы проталкиваем поток через остаточное ребро, которое также является и исходным ребром. В строке 5 уменьшается поток в ребре (v, u), поскольку остаточное ребро в действительности обратно ребру в исходной сети. Наконец в строках 6 и 7 обновляются избыточные потоки в вершины u и v. Таким образом, если f являлся предпотоком перед вызовом процедуры **PUSH.** он останется предпотоком и после ее выполнения.

Условие $h[u] = h[v] + 1$ гарантирует, что дополнительный поток идет по ребрам идущим вниз с единичной разницей высот.

Операция PUSH называется ***проталкиванием*** (push) из u в v. Если операция проталкивания применяется к некоторому ребру (u, v), выходящему из вершины u, будем говорить, что операция проталкивания применяется к u. Если в результате ребро (u, v) становится ***насыщенным*** (saturated) (после проталкивания $C_f(u, v) = 0$), то это ***насыщающее проталкивание*** (saturating push), в противном случае это ***ненасыщающее проталкивание*** (nonsaturating push). Если ребро становится насыщенным, оно исчезает из остаточной сети. Один из результатов ненасыщающего проталкивания характеризует следующая лемма.

Лемма 7.5. После ненасыщающего проталкивания *из u в v* вершина *u* более не является переполненной.

Доказательство. Поскольку проталкивание ненасыщающее, фактическое количество посланного потока $d_f(u,v)$ должно быть равно величине $e[u]$ непосредственно перед проталкиванием. Поскольку избыток $e[u]$ уменьшается на эту величину, после проталкивания он становится равным 0.

Операция подъема

Основная операция LIFT(U) применяется, если вершина u переполнена и если h(u)< h(u) для всех ребер остаточной сети.

Иными словами, переполненную вершину u можно подвергнуть подъему, если все вершины v, для которых имеется остаточная пропускная способность от u к v, расположены не ниже u, так что протолкнуть поток из u нельзя. (Напомним, что по определению ни исток s, ни сток t не могут быть переполнены; следовательно, ни s, ни t нельзя подвергать подъему).

LIFT (u)

Применяется при:

Вершина u переполнена и для всех v – вершин графа, таких,

// что (u,v) е E_f, имеем $h[u] < h[v]$.
// Действие: увеличивает высоту u.
$h[u] = 1 + \min \{h[v] : (u, v) \in E_f\}$

Когда выполняется подъем *u*, E_f должно содержать хотя бы одно ребро, выходящее из *u*, чтобы минимизация в коде операции осуществлялась по непустому множеству. Это свойство вытекает из предположения о том, что вершина u переполнена(f(V,u)=e[u]>0). Получаем $c_f(u,v) = c(u,v)-f[u,v]>0$, это означает, что (u, v) принадлежит E_f.

Общая ема алгоритма

Алгори начинается с ALIZE-PREFLOW, ДЛЯ ЗАДАНИЯ **НАЧАЛЬНОГО ПРЕДПОТОКА.**

INITIA E-PREFLOW(G

1 каждой вершин
2 *{h* = 0
3 *[* = 0}
4 каждого ребра *(v, ι)*
5 *′u, v) = 0*
6 *, u) = 0}*
7 *] = |V[G]|*
8 **for** каждой вершины *v* ∈ *Adj[s]*
9 *{ f(s,v) = c(s,v)*
10 *f[v,s]= - c(s,v)*
11 e[s]= c(s, v)

В массиве h хранятся высоты, в массиве e – избытки, *c(v,u)* – пропускные способности. Поток записывается в массив f.

Каждое ребро, выходящее из истока ***s,*** заполняется до его пропускной способности, а все остальные ребра не несут потока. Для каждой вершины ***v,*** смежной с истоком, изначально e[***v]*** = ***c(s,v)*** и инициализируем ***e[s]*** суммой этих значений с обратным знаком. Обобщенный алгоритм начинает работу с начальной функцией высоты ***h,*** задаваемой следующим образом: h[s]=|V|, для остальных вершин h равна 0. Это высотная функция, поскольку единственными ребрами ***(u, v),*** для которых h[**u] > *h*[v]+** 1, являются те, для которых ***u = s,*** и эти ребра заполнены, а это означает, что их нет в остаточной сети.

Программа Generic-Preflow-Push дает общую схему алгоритма, основанного на проталкивании предпотока.

Generic-Preflow-Push(G)
1. Initialize-Preflow (G,s)
2. while возможны операции подъема или проталкивания
3. выполнить одну из этих операций

В следующей лемме утверждается, что до тех пор, пока существует хотя бы одна переполненная вершина, применима хотя бы одна из этих операций.

Лемма 7.6. Пусть ***G*** = (V, ***E)*** - сеть с истоком ***s*** и стоком ***t,*** а f - предпоток, и пусть ***h*** – высотная функция для f. ***u*** - переполненная вершина. Тогда в u возможно либо проталкивание, либо подъем.

Доказательство. Для любого остаточного ребра (***u***, ***v)*** выполняется соотношение ***h(u***) ≤***h(v)*** + 1, поскольку ***h*** представляет собой функцию высоты. Если в переполненной вершине ***u*** не применима операция проталкивания, то для всех остаточных ребер (***u***, ***v***) должно выполняться условие ***h(u) < h(v)*** +1, из чего следует, что ***h(u) ≤ h(v).*** В таком случае в ***u*** можно применить операцию подъема.

Корректность метода проталкивания предпотока

Чтобы показать, что обобщенный алгоритм проталкивания предпотока позволяет решить задачу нахождения максимального потока, сначала докажем, что если он завершается, то предпоток f является максимальным потоком. Затем докажем, что алгоритм завершается. Начнем с рассмотрения некоторых свойств функции высоты ***h.***

Лемма 7.7. (Высота вершины никогда не уменьшается) При выполнении процедуры Generic-Preflow-Push над транспортной сетью ***G*** = (V, ***E)*** для любой вершины графа ее высота ***h[u]*** никогда не уменьшается. И всякий раз, когда к вершине ***u*** применяется операция подъема, ее высота ***h[u]*** увеличивается как минимум на 1.

Лемма 7.8. Пусть ***G*** = (***V,E)*** представляет собой сеть с истоком ***s*** и стоком ***t.*** Во время выполнения процедуры Generic-Preflow-Push над сетью ***G*** атрибут ***h*** сохраняет свойства функции высоты.

Доказательство.

При подъеме вершины выходящие из u остаточные ребра не нарушают определение высотной функции. Для входящих ребер не может быть проблем, так как высота вершины u только возрастает.

Теперь рассмотрим операцию **PUSH(U, v).** Эта операция может добавить ребро ***(v, u)*** к ***Ef,*** и может удалить ребро ***(u, y)*** из ***Ef.*** В первом случае ***h[v] = h[u]*** — 1, так что ***h*** остается функцией высоты. Во втором случае удаление ребра ***(u, v)*** из остаточной сети приводит к удалению соответствующего ограничения, так что ***h*** по-прежнему остается функцией высоты

Лемма 7.9. Пусть ***G = (V, E)*** - сеть с истоком ***s*** и стоком ***t,*** f - предпоток в ***G,*** а ***h*** — функция высоты, определенная на множестве ***V.*** Тогда в остаточной сети G_f не существует пути из истока ***s*** в сток ***t.***

Доказательство. Предположим, что в ***Gf*** имеется некоторый путь ***p*** из ***s*** в ***t,*** покажем, что это приводит к противоречию. Без потери общности можно считать, что ***p*** — простой путь, так что ***его длина меньше*** \ ***V*** \. При этом вдоль пути высота падает от |V| до 0. Следовательно, в пути есть ребро, где высота падает по крайней мере на 2. Такое ребро не может входить в остаточную сеть.

Теорема 7.1 (Корректность обобщенного алгоритма проталкивания предпотока)

Если алгоритм Generic-Preflow-Push**,** выполняемый над сетью ***G = (V, E)*** с истоком ***s*** и стоком ***t,*** завершается, то вычисленный им предпоток f является максимальным потоком в ***G.***

Доказательство. Воспользуемся следующим инвариантом цикла.

Всякий раз, когда в строке 2 процедуры Generic-Preflow-Push выполняется проверка условия цикла **while,** f является предпотоком.

Инициализация. INITIALIZE-PREFLOW делает f предпотоком**.**

Сохранение. Внутри цикла **while** в строках 2 и 3 выполняются только операции проталкивания и подъема. Операции подъема влияют только на атрибуты высоты, но не на величины потока, следовательно, от них не зависит, будет ли f предпотоком. Анализируя работу процедуры **PUSH,** мы доказали**,** что если f является предпотоком перед выполнением операции проталкивания, он остается предпотоком и после ее выполнения.

Завершение. По завершении процедуры каждая вершина из множества ***V*** — {s, ***t}*** должна иметь нулевой избыток, поскольку из леммы 7.6 и утверждения, что f всегда остается предпотоком, следует, что переполненных вершин нет. Следовательно, f является потоком. Лемма 7.8 показывает, что при завершении ***h*** является функцией высоты; таким образом, согласно лемме 7.9 в остаточной сети G_f не существует пути из ***s*** в ***t.*** Согласно теореме о максимальном потоке и минимальном разрезе f является максимальным потоком.

Анализ метода проталкивания предпотока

Чтобы показать, что обобщенный алгоритм проталкивания предпотока действительно завершается, найдем верхнюю границу количества выполняемых им операций для каждого вида операций (подъем, насыщающее проталкивание и ненасыщающее проталкивание). Зная эти границы, несложно построить алгоритм, время работы которого — $O(V^2E)$.

Лемма 7.10. Пусть ***G = (V, E)*** представляет собой сеть с истоком ***s*** и стоком ***t,*** а f является предпотоком в ***G.*** Тогда в остаточной сети G_f для любой переполненной вершины ***u*** существует простой путь из ***x*** в ***s.***

Доказательство. Для переполненной вершины ***u*** существует простой путь из ***s*** в u, по ребрам которого идет положительный поток. Возьмем путь из s в u, по всем ребрам которого идет положительный поток и обратим его ребра. Обратные ребра входят в остаточную сеть.

Лемма 7.11. При выполнении программы Generic-Preflow-Push высота любой вершины графа не превосходит 2|V|-1.

Доказательство. По определению высоты h[s]=|V| и h[t]=0. Подъем применим только к переполненным вершинам. Пусть u –переполненная вершина. По лемме 7.10 в G_f существует простой путь из этой вершины в s. По определению высотной функции высота не может убывать более чем на 1 вдоль ребер остаточной сети, а высота конечной вершины пути

равна |V|. Так как путь простой, то содержит не более |V|-1 ребер, поэтому высота его начала не превосходит 2|V|-1.

Следствие . (Оценка числа подъемов)

При выполнении программы Generic-Preflow-Push общее число операций подъема не превосходит $2|V|^2$.

***Доказательство. Высота вершины при подъеме увеличивается, но не может стать больше* 2|V|-1**, поэтому любую вершину из V\{s,t} можно поднять максимум 2|V|-1 раз. Всего таких вершин |V|-2, поэтому общее число подъемов не превышает $(2|V|-1)(|V|-2)< 2|V|^2$.

Лемма 7.12. (Оценка числа насыщающих проталкиваний) При выполнении программы Generic-Preflow-Push количество насыщающих проталкиваний не превосходит 2|V||E|.

Доказательство. Рассмотрим насыщающие проталкивания между вершинами $u,v \in V$ (в обе стороны). Если хотя бы одно проталкивание было, то хотя бы одно ребро (u,v) или (v,u) принадлежит E. Пусть проталкивание из u в v было насыщающим. После него ребро (u,v) исчезло из остаточной сети. Для того, чтобы это ребро появилось, необходимо проталкивание из v в u, но это возможно только при условии, что h[v]=h[u]+1, т.е. h[v] необходимо увеличить хотя бы на 2. По лемме об ограничении значения высоты высота вершины не превышает 2|V|-1 , следовательно, количество раз, когда высота вершины может увеличится на 2, меньше |V|. Поскольку между двумя насыщающими проталкиваниями высота одной из вершин должна увеличится по меньшей мере на 2, то между вершинами u и v их будет не более 2. Тогда суммарно по всем ребрам во время работы алгоритма произойдут не более 2|V||E| насыщающих проталкиваний.

Лемма 7.13. Количество ненасыщающих проталкиваний при выполнение алгоритма Generic-Preflow-Push не превосходит $4|V|^2(|V|+|E|)$

Доказательство. Пусть $\Phi = \sum_{u:в(u)>0} h(u)$. После завершения я алгоритма ни одна из вершин не является переполненной, то и величина Ф после выполнения алгоритма должна равняться нулю.

Для начала рассмотрим, каким образом может увеличиваться величина Ф . Первое, что может увеличить Ф, это подъем, поскольку, осуществляя данную операцию, мы не изменяем избыточный поток ни у одной вершины, а лишь увеличиваем высоту одной из них. При каждой операции подъема Ф увеличивается менее чем на 2|V|, так как подъем не может увеличить высоту вершины больше, чем ее максимальная высота, которая согласно лемме 7.11 может быть не более 2|V|-1. А поскольку из леммы (7.12) известно, что число подъемов не превышает $2|V|^2$, то суммарно подъемы всех вершин могут увеличить Ф не более чем на $4|V|^3$.

Во-вторых, величина Ф может увеличится при насыщающем проталкивании из u в v, потому что e(u)>0 и после насыщающего проталкивания, а вершина v может стать переполненной. Увеличение меньше 2|V|, так как изменения высот не происходит, а высота вершины v не превосходит 2|V|-1. Известно, что количество насыщающих проталкиваний за все время выполнения алгоритма не превосходит 2|V||E|, следовательно, за счет насыщающих проталкиваний Ф увеличится не более чем на 4|V|2|E|.

Итого, получаем, что величина Ф не может быть больше 4|V|2(|V|+|E|). Теперь покажем, что ненасыщающее проталкивание уменьшает Ф как минимум на единицу. Пусть произошло ненасыщающее проталкивание из вершиныu в v. Согласно лемме (7.5) после ненасыщающего проталкивания вершина u перестает быть переполненной, следовательно, Ф уменьшается на величину ее высоты. После проталкивания вершина v является переполненной, и поэтому Ф могла увеличится на h(v). Поскольку h(u)=h(v)-1, то при каждом ненасыщающем проталкивании Ф уменьшается по меньшей мере на единицу.

Зная верхнюю границу величины Ф, ее значение после выполнения алгоритма и то, что при каждом ненасыщающем проталкивании Ф уменьшается минимум на единицу, то

можно сделать вывод, что количество ненасыщающих проталкиваний не больше чем 4|V|2(|V|+|E|) .

Теорема 7.2. Общее число операций подъема и проталкивания при исполнении программы Generic-Preflow-Push на сети G=(V,E) равно O(V2E).

Следствие. Алгоритм, основанный на проталкивании предпотока, можно реализовать так, чтобы на сети G=(V,E) время его работы было $O(V^2E)$.

Алгоритм «поднять и в начало». Пользуясь методом проталкивания предпотока, мы применяли операции подъёма и проталкивания в более или менее произвольном порядке. Более продуманный порядок выполнения этих операций позволяет уменьшить время работы алгоритм (по сравнению с оценкой $O(V^2E)$). Рассмотрим алгоритм «поднять-и-в-начало» (lift-to-front algorithm), использующий эту идею; время его работы есть $O(V^3)$, что асимптотически по крайней мере не хуже, чем $O(V^2E)$.

Алгоритм «поднять-и-в-начало» хранит все вершины сети в виде списка. Алгоритм просматривает этот список, начиная с головы, и находит в нем переполненную вершину и. Затем алгоритм «обслуживает» эту вершину, применяя к ней операции подъёма и проталкивания до тех пор, пока избыток не станет равным нулю. Если для этого вершину пришлось поднять, её перемещают в начало списка (отсюда и название алгоритма), и просмотр списка начинается вновь.

При анализе алгоритма используем понятие допустимого ребра — ребра остаточной сети, по которому возможно проталкивание. Сначала мы изучим некоторые их свойства и рассмотрим процесс «обслуживания» вершины.

Допустимые рёбра. Пусть f — предпоток в сети G = (V, E), а h — высотная функция. Назовем ребро (и,v) допустимым (admissible), если оно входит в остаточную сеть ($c_f(u,v) > 0$) и h(u) = h(v) + 1. Остальные рёбра мы будем называть недопустимыми (inadmissible). Обозначим через Ef,h множество допустимых рёбер; сеть Gf,h = (V,Ef,h) назовём сетью допустимых рёбер (admissible network). Она состоит из рёбер, по которым возможно проталкивание. Поскольку вдоль допустимого ребра высота уменьшается, имеет место такая :

Лемма 7.14. (Допустимые рёбра образуют ациклический граф)

Пусть f — предпоток в сети G = (V,E); пусть h — высотная функция. Тогда сеть допустимых рёбер $G_{f,h} = (V,E_{f,h})$ не содержит циклов.

Посмотрим, как изменяют сеть допустимых рёбер операции подъёма и проталкивания.

Лемма 7.15. Пусть f — предпоток в сети G =(V,E) и h — высотная функция. Пусть (u,v) — допустимое ребро и вершина u переполнена. Тогда по (u,v) возможно проталкивание. В результате выполнения этой операции новые допустимые рёбра не появляются, но ребро (u, v) может стать недопустимым.

Доказательство. В результате проталкивания в остаточной сети может появиться только ребро (u,v). Поскольку ребро (u, v) допустимо, то h(v) = h(u) — 1, и потому ребро (v,u) недопустимо. Если проталкивание оказывается насыщающим, то в результате $C_f(u,v) = 0$ и ребро (u, v) исчезает из остаточной сети (и становится недопустимым).

Лемма 7.16. Пусть f — предпоток в сети *G* = (V, *E)* и *h* — высотная функция. Если вершина *и* переполнена и из нее не выходит допустимых рёбер, то возможен подъём вершины *и.* После подъёма появится по крайней мере одно допустимое ребро, выходящее из вершины *и* и не будет рёбер, входящих в *и.*

Доказательство. Как мы видели, в переполненной вершине *и* возможно либо проталкивание, либо подъём. Так как из *и* допустимые рёбра не выходят, то проталкивание в ней невозможно, и возможен подъём. При этом высота вершины увеличивается так, что проталкивание становится возможным, то есть появляется допустимое ребро.

Проверим второе утверждение леммы. Предположим, что после подъёма имеется допустимое ребро (w, v). Тогда h(w) = h(v) + 1, — а до подъёма было выполнено h(w) = h(v) +

1. По определению высотной функции ребро (w, v) должно быть насыщенным (до и после подъёма — подъём не меняет потоков), и потому не входит в остаточную сеть и не является допустимым.

Списки соседей. Алгоритм «поднять-и-в-начало» использует специальный способ хранения рёбер сети G = (V, E). Именно, для каждой вершины и ∈ V имеется односторонне связанный список соседей (neighbor list) N[u]. Вершина v фигурирует в этом списке, если (u, v) ∈E или если (v,u) ∈E. Таким образом, список N[u] содержит все вершины v, для которых (u,v) имеет шанс появиться в остаточной сети. Первый элемент этого списка обозначается head[N[u]]; следующий за вершиной v сосед: next- neighbor[v]. Если вершина v последняя в списке, то next-neighbor[v] =NIL.

Порядок в списке соседей может быть произвольным; он не меняется в ходе работы (всякий раз алгоритм просматривает список соседей в одном и том же порядке). Для каждой вершины u хранится указатель current[u] на очередной элемент списка N[u]. Изначально current[u] установлен на head[N[u]].

Обработка переполненной вершины

Обработка переполненной вершину *и* состоит в том, что её *разгружают*(discharge), проталкивая весь избыток потока в соседние по допустимым рёбрам. Иногда для этого необходимо создать новые допустимые рёбра, подняв вершину *и.*

Discharge(u)

1. **while** e[u]>0
2. **do** v ←current [u]
3. **if** v==nil
4. **then** Lift(u)
5. current[u] ← head[N[u]]
6. **else if** c_f(u,v)>0 **and** h[u]=h[v]+l
7. **then** Push(u,v)
8. **else** current[u] ←next-neighbor [v]

Каждая итерация цикла **while** производит одно из трёх действий:

1. Если мы дошли до конца списка *(v* =nil), то мы поднимаем вершину *и* (строка 4) и переходим к началу списка *N[u]* (строка 5). Мы увидим, что подъём возможен.
2. Если мы не дошли до конца списка и ребро (u, *v)* — допустимое (проверка в строке 6), то проталкиваем поток из u в v; (строка 7).
3. Если мы не дошли до конца списка, но ребро (u,*v)*— недопустимое, то сдвигаем указатель *current[u]* на одну позицию в списке (строка 8).

Заметим, что при вызове процедуры discharge указатель *current[u]* находится в позиции, «унаследованной» от предыдущего вызова. Последним действием этой процедуры может быть лишь проталкивание: процедура останавливается, если избыток *e[u]* обращается в нуль, но ни подъём вершины, ни сдвиг указателя не меняют эту величину.

Надо проверить, что процедура discharge выполняет подъём и проталкивание тогда, когда это действительно можно сделать по нашим правилам.

Лемма 7.17. При вызове операции push (u, v) в процедуре discharge (строка 7) по ребру (u,*v)* возможно проталкивание. При вызове операции Lift(u) в процедуре discharge (строка 4) возможен подъём вер шины *и.*

Доказательство. Возможность проталкивания гарантируется проверками в строках 1 и 6, так что первое утверждение очевидно.

Докажем второе утверждение. Для этого достаточно доказать, что все выходящие из и рёбра недопустимы. Заметим, что при вызовах процедуры discharge указатель current[u] перемещается по списку N[u] от его начала head[N[u]] до конца.

В конце вершину u поднимают и начинается новый проход. Каждый раз, прежде чем сдвинуть указатель с произвольной позиции v мы убеждаемся (строка 6), что ребро (u, v) недопустимо. Таким образом, в конце прохода все выходящие из u рёбра были просмотрены и оказались недопустимыми. Могли ли они затем стать допустимыми (до конца прохода)? Очевидно, проталкивания вообще не создают допустимых рёбер. Их могут породить только операции подъёма. Но вершина u не поднималась (в течение прохода по списку), а подъёмы других вершин создают лишь выходящие из них допустимые рёбра. Поэтому в конце прохода все выходящие из вершины и рёбра недопустимы, поэтому её можно поднять.

Алгоритм «поднять-и-в-начало». Алгоритм «поднять-и-в-начало» хранит множество V \ {s,t} вершин (отличных от истока и стока) в виде списка. При этом существенно то, что список этот оказывается «корректно упорядоченным» в следующем смысле: конец любого допустимого ребра находится дальше в списке, чем начало этого ребра (напомним, что допустимые рёбра образуют ациклический граф). Задачу о поиске корректного упорядочения для произвольного ациклического графа мы называли задачей топологической сортировки.

Следующую в этом списке за u вершину обозначим next[u]; если вершина u последняя в списке, то next[u]=NIL.

Lift-To-Front(G,s,t)

1. Initialize-Preflow(G,s)
2. L ←V[G] \{s,t\} (в любом порядке)
3. for (для) каждой вершины u ∈ V[G] \{s,t\}
4. do current[u] ← head[N[u]]
5. u ← head[L]
6. while u ≠nil
7. do old-height ← h[u]
8. Discharge(u)
9. if h[u]>old-height
10. then переместить u в начало списка L
11. u ← next[u]

Алгоритм формирует начальный предпоток (строка 1), список L(строка 2) (точно так же, как это делалось раньше). Затем (строки 3-4) он устанавливает указатели current[u] в начало списка соседей каждой вершины и (считаем, что для всех вершин и списки соседей N[u] уже созданы.)

Работа цикла (строки 6-11) происходит так: мы просматриваем все элементы списка L, начиная с начала (строка 5). Всякий раз мы разгружаем текущую вершину и (строка 8). Если при этом высота вершины и увеличилась (что определяется сравнением с сохранённым в строке 7 прежним значением), то мы перемещаем её в начало списка (строка 10). После этого мы переходим к следующему элементу списка L. Заметим, что если мы переместили и в начало списка, то очередным будет элемент, следующий за u в её новой позиции.

Докажем, что алгоритм LlFT-To-FRONT находит максимальный поток. Для этого убедимся, что его можно рассматривать как реализацию общей схемы проталкивания предпотока. Сначала заметим, что программа применяет подъёмы и проталкивания только там, где они возможны. Осталось доказать, что после завершения алгоритма никакие подъёмы или проталкивания невозможны, останавливается, возможных подъёмов и проталкиваний нет.

Когда мы в последний раз в программе просмотрели список L, мы разгрузили каждую вершину u, не подняв её. Как мы вскоре увидим, список L во время выполнения программы остаётся корректно упорядоченным, то есть конец любого допустимого ребра идёт после его начала. Поэтому процедура discharge(m), проталкивая поток по допустимым рёбрам, создаёт избыток в вершинах, идущих в списке после u, и не трогает вершины, предшествующие u, в которых избыток остаётся равным нулю. Таким образом, по завершению работы избыток в каждой вершине равен нулю (и ни проталкивание, ни подъём невозможны).

Лемма 7.18 . При исполнении алгоритма LlFT-To-FRONT список L остаётся корректно упорядоченным относительно (текущего) графа допустимых рёбер $G_{f,h} = (V, E_{f,h})$ после любого числа итераций цикла в строках 6—11.

Доказательство. Перед первым выполнением цикла допустимых рёбер нет, так как высота истока h[s] равна |V| ≥ 2 (множество V содержит по крайней мере исток s и сток t), а высота остальных вершина равна 0 (и единичного перепада высот быть не может).

Докажем, что свойство топологической упорядоченности сохраняется после каждого выполнения тела цикла. Сеть допустимых рёбер меняют только подъёмы и проталкивания. Проталкивания не создают допустимых рёбер. После подъёма произвольной вершины и все рёбра, входящие в эту вершину — недопустимые. Поэтому после перемещения u в начало списка порядок элементов в списке становится корректным.

Анализ алгоритма. Покажем, что время работы алгоритма «поднять-и-в-начало» на сети G = (V,E) равно O (V^3). Сначала напомним некоторые уже известные нам факты. Этот алгоритм является реализацией метода проталкивания предпотока, поэтому действует верхняя оценка O (V) на число подъёмов каждой вершины, а общее число подъёмов есть O (V^2). На все подъёмы уходит время O(VE).Общее число насыщающих проталкиваний есть O(VE).

\ **Теорема.** Время работы программы LlFT-To-FRONT на сети G = (V,E) равно O (V^3).

Доказательство. Разобьём время работы программы LlFT-To-FRONT на периоды между двумя подъёмами. Тогда всего периодов будет (как и подъёмов) O (V^2). Покажем, что за каждый период мы вызываем процедуру discharge O (V) раз. Действительно, за один период число вызовов процедуры discharge (поскольку внутри периода мы не поднимаем вершину, список остаётся неизменным) не превосходит длины списка, и тем самым |V|.

Следовательно, процедура discharge вызывается O (V3) раз (строка 8), и время работы программы LlFT-To-FRONT равно O (V3) плюс суммарное время, уходящее на выполнение вызовов discharge. Оценим второе слагаемое.

При каждом повторении цикла в процедуре discharge совершается ровно одно из трёх действий: подъём вершины, перемещение указателя и проталкивание предпотока. Оценим количество операций каждого из этих типов.

Начнём с подъёмов (строки 4-5). На все O (V^2) подъёмов требуется время O(VE).

Оценим количество перемещений указателя current[u] (строка 8). Обозначим через degree(u) степень вершины u. На каждый подъём вершины u приходятся O(degree(u)) перемещений указателя, поэтому всего производится O(V • degree(u)) перемещений для каждой вершины. Таким образом, общее число перемещений указателей равно O(VE).

Оценим число проталкиваний (строка 7). Мы уже знаем, что количество насыщающих проталкиваний равно O(VE). Заметим, что сразу после выполнения ненасыщающего проталкивания процедура discharge прекращает работу, так как избыток потока обращается в ноль. Таким образом, при каждом вызове процедуры выполняется не более одного ненасыщающего проталкивания; всего вызовов O(V^3), поэтому ненасыщающих проталкиваний не более 0(V^3).

Таким образом, время работы программы LlFT-To-FRONT оценивается как: O(V^3 + VE]| = O(V^3).

Тестовые вопросы по теме

1. Как от вида или представления графа зависит временная сложность алгоритмов?
2. Покажите, что любой поток в сети с несколькими истоками и стоками соответствует потоку той же величины в сети с единственным истоком и стоком.
3. Дать определение остаточной сти.
4. Как формируется остаточная сеть?
5. Какова временная сложность алгоритма Форда-Фалкерсона?
6. Какой путь называется увеличивающим?
7. Дать определение разреза.
8. Какой разрез называется минимальным? Каковы его свойства?
9. Сформулировать алгоритм Форда Фалкерсона.
10. Сформулировать идею алгоритма Эдмондса-Карпа. Какова временная сложность этого алгоритма?
11. Перечислить наиболее эффективные алгоритмы нахождения максимального потока.
12. Привести пример сети, найти ее минимальный разрез и проверить, как соотносятся максимальный поток и пропускная способность найденного разреза.
13. Можно ли применить алгоритм для нахождения потока в сети с одним стоком и одним истоком к сети, количество стоков и истоков которой больше 1? Если можно, то как это сделать?
14. Каково назначение операции подъема?
15. Дать определение предпотока.
16. Какая функция называется высотной?
17. Сформулировать идею алгоритма «поднять и в начало»? Каково его быстродействие?
18. Как доказывается корректность алгоритма проталкивание предпотока?
19. Оценить число подъемов при выполнении алгоритма проталкивание предпотока.
20. Оценить число насыщающих проталкиваний алгоритма проталкивание предпотока.
21. Какое проталкивание называется ненасыщающим?
22. Какое ребро называется допустимым?
23. Как формируется список вершин при нахождении максимального потока методом поднять и в начало?
24. Какова временная сложность алгоритма поднять и в начало?

Рекомендуемые источники по теме 7

1 Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. –М.: МЦНМО, 2000. – 960 с.

2 Кристофидес Н. Теория графов. Алгоритмический подход. М.: Мир, 1978. – 432 с.