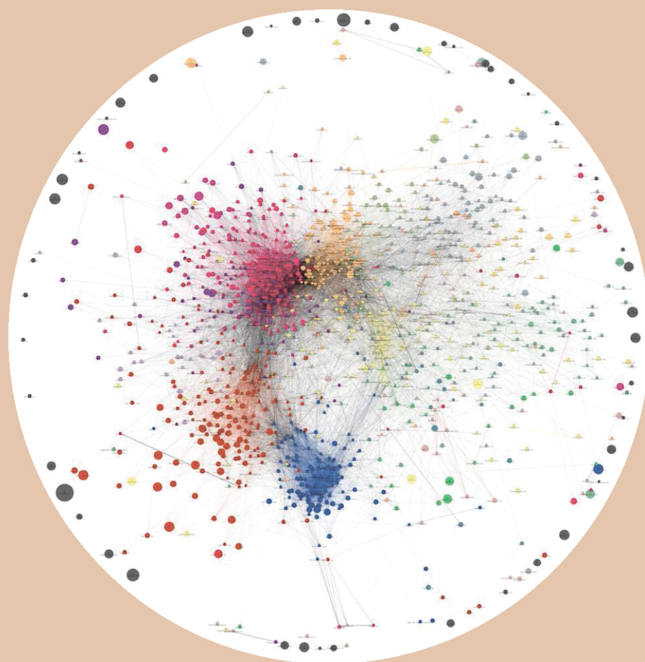
МФТИ

В. В. Прут

ГРАФЫ. АЛГОРИТМЫ НА ЯЗЫКЕ С



Министерство образования и науки Российской Федерации

Федеральное государственное автономное
образовательное учреждение высшего образования
«Московский физико-технический институт
(государственный университет)»

В. В. Прут

ГРАФЫ.
АЛГОРИТМЫ НА ЯЗЫКЕ С

Учебное пособие

Москва
МФТИ
2017

УДК 519.7(075)
ББК 22.174+22.176я73
П85

Р е ц е н з е н т ы :

Доктор физико-математических наук, профессор,
главный научный сотрудник Вычислительного центра
им. А. А. Дородницина ФИЦ «Информатика и управление» РАН *В. И. Зубов*

Доктор физико-математических наук,
начальник отдела моделирования физических процессов
и прикладных технологий Курчатовского комплекса реабилитации
и нераспространения НИЦ «Курчатовский институт» *Ю. Ю. Клосс*

Прут, В. В.

П85 Графы. Алгоритмы на языке С : учебное пособие / В. В. Прут. – М. : МФТИ, 2017. – 213 с.

ISBN 978-5-7417-0633-6

Пособие посвящено одному из наиболее интересных и практически ценных разделов информатики и дискретной математики – теории графов. Цель пособия – в весьма ограниченном объеме дать студентам достаточно широкий обзор различных задач теории графов.

Рассмотрены базовые алгоритмы решения этих задач с такой степенью доскональности, которая позволила бы использовать полученные знания в практической работе. Для большинства алгоритмов приведены С-функции.

Предназначено для студентов 1-го курса. Основное назначение учебного пособия – методическое обеспечение курса «Информатика (алгоритмы и алгоритмические языки)».

УДК 519.7(075)
ББК 22.174+22.176я73

Печатается по решению Редакционно-издательского совета Московского физико-технического института (государственного университета)

ISBN 978-5-7417-0633-6

© Прут В. В., 2017
© Федеральное государственное автономное образовательное учреждение высшего образования «Московский физико-технический институт (государственный университет)», 2017

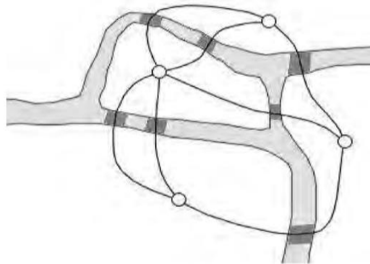
Введение

Дискретные структуры являются фундаментальной основой информатики. Одними из важнейших таких структур являются графы. Сведения из теории графов широко используются не только в организации структур данных и разработке алгоритмов, но и во всех остальных разделах информатики. По мере развития информатики, все более и более сложные методы анализа оказывают влияние на научные и практические проблемы. Графы являются удобным языком для формулировки и эффективным инструментом для решения задач, относящихся к широкому кругу этих проблем. В частности, они используются при конструировании систем автоматизированного проектирования программирования, создании операционных систем, исследовании операций и управления, во многих задачах теоретической физики, теории информации, статистики, математической лингвистики, экономики. Широкое распространение в последнее время получило использование графов в задачах распознавания образов.

Задачи на графах и алгоритмы их решения играют одну из ключевых ролей при алгоритмизации комбинаторных задач. Формулировка той или иной задачи дискретной математики на языке теории графов часто облегчает ее решение. В этом случае использование эффективных алгоритмов, существующих в теории графов, позволяет найти конструктивное решение рассматриваемой задачи.

Возможность приложения теории графов к широкому кругу задач из различных научных областей заложена уже в самом понятии графа, сочетающего в себе теоретико-множественные, комбинаторные и топологические аспекты.

История становления теории графов интересна и поучительна. Первая известная публикация была ответом Леонарда Эйлера на поставленный в письме коллеги вопрос о том, как именно можно пройти по семи мостам города Кенигсберга.



Эйлер, как он сам писал позже в одном из своих писем, после долгих размышлений нашел простое правило, позволившее ему решить предложенную и значительно более сложные задачи, придуманные им самим. Однако, несмотря на нарастающий научный авторитет, публикация Эйлера «Euler L. Solutio probtematis ad geometriam situs pertinentes. Commentarii Academiae Petropolitanae, 1736» не привлекла внимания ни современных ему ученых, ни нескольких последующих поколений исследователей. Во всяком случае, никаких следов проявления интереса к заявленной проблематике до 1856 г. не замечено.

Придуманная тогда У. Гамильтоном игрушка в виде утыканного гвоздиками деревянного додекаэдра также долгое время оставалась предметом праздных размышлений, и никто не думал, что через несколько десятков лет две эти развлекательные задачи займут достойное место в востребованной ныне теории графов. Сам термин «граф» возник как сокращение слова graphic (график) и был введен Д. Кенигом в 1936 г.

Конечно, этой актуальности в немалой степени способствовали работы Г. Кирхгофа по исследованию электрических цепей и А. Кэли при описании строения углеводородов, а также увеличивавшийся поток задач, возникавших в различных областях науки и техники.

Оказалось, что при помощи графов можно вполне успешно моделировать и решать самые разнообразные задачи.

Все это потребовало обоснований, необходимость построения которых вылилась в новую теорию. Не были забыты ни Эйлер, ни Гамильтон. Их имена носят графы с увлекательными свойствами.

Так и возникли два естественных направления работы с графами. Изучение свойств собственно графов: терминология, теоремы, доказательства, формулы, гипотезы. Применение графов в других науках и прикладных задачах.

Основные определения

Граф $G = G(V, E)$ есть совокупность двух множеств: множества *вершин* V (*vertex*) и множества *ребер* *E* (*edge*) – двух элементных подмножеств множества V.

Вершины u и v множества *V* *соединены* ребром $e = (u, v)$. Вершины *инцидентны* (*incidence*) ребру $e = (u, v)$. Если (u, v) – ребро, тогда вершины u и v есть *концы* ребра (u, v). Ребро (u, v) также *инцидентно* к вершинам u и v. Если вершина v не является концом ребра *e*, они *не инцидентны*.

Две вершины называются *смежными* (*adjacent vertices*), если они инцидентны одному ребру. Два ребра называются *смежными* (*adjacent edges*), если они инцидентны общей вершине.

Заметим, что смежность есть отношение между однородными элементами графа – между вершинами *или* между ребрами, тогда как инцидентность есть отношение между разнородными элементами – между вершинами *и* ребрами.

Число вершин графа *G* часто обозначается $|V| = n$, а число ребер $|E| = m$. Эти обозначения удобны для программирования, поэтому используются во всех приводимых программах.

Ориентированный граф, или *орграф* $G = G(V,E)$, есть граф, который состоит из множества *V* вершин и множества *E* упорядоченных пар элементов из *V*. Элемент множества *E* называется *ориентированным ребром* (*directed edge*), или *дугой* (*arc*). Если пара вершин неупорядоченная, то ребро *неориентированное* (*undirected edge*). В дуге $(u, v) \in E$, u – *начальная вершина* дуги, а v – *конечная вершина*.

Граф, содержащий только неориентированные рёбра, называется неориентированным графом. Граф, содержащий как ориентированные, так и неориентированные ребра, называется *смешанным*, или *частично ориентированным графом*.

В графе допускается наличие более одного ребра между двумя вершинами, тогда он называется *мультиграфом*. Если орграф содержит более чем одну дугу из одной вершины в другую, то называется *ориентированным мультиграфом*.

Ребро может соединять вершину саму с собой, тогда оно называется *петлей*. Причем возможно наличие нескольких петель. Если в графе допускается наличие петель, то он называется *псевдографом* (*графом с петлями*).

Примеры графов разного типа приведены на рис. 1.

Граф называется *простым*, если любая пара вершин соединена не более чем одним ребром и граф не имеет петель. По умолчанию граф простой.

Иногда граф называют сетью (*network*), вершины – узлами (*node*), рёбра – связями (*bond*).

Граф $G' = G(V', E')$ называется *подграфом*, т. е. *частью* графа G, если $E' \subseteq E$. Аналогично, как и для графа, для орграфа вводится понятие – ориентированный подграф.

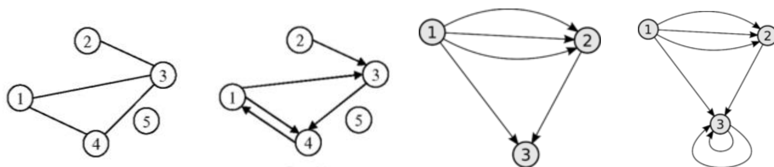


Рис. 1. Неориентированный и ориентированный графы, мультиграф, псевдограф

Степень вершины

Degree of vertex

Число ребер, инцидентных вершине v, называется степенью вершины v и обозначается $\deg v$ или $d(v)$ (от *degree*). Иногда этот термин называют *валентностью* (*valency*).

Последовательность $\{d_1, d_2, \ldots, d_n\}$ степеней вершин графа называется *степенной последовательностью* (*degree sequence*).

Граф называется *регулярным* (*regular*), или *однородным* степени *n*, если $d(v) = \text{const}$ для всех $v \in V$.

На рис. 2 приведен неориентированный граф, на вершинах которого обозначены степени. Петля увеличивает степень на 2.

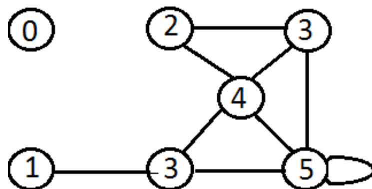


Рис. 2

Вершина степени 0 называется *изолированной*. Вершина степени 1 называется *висячей*.

Полустепень исхода (*outdegree*) вершины v для ориентированного графа есть количество дуг, для которых v является начальной вершиной, и обозначается $d^{-}(v)$, или $\deg^{-}(v)$. *Полустепень захода* (*indegree*) вершины v есть количество дуг, для которых v является конечной вершиной, и обозначается $d^{+}(v)$, или $\deg^{+}(v)$. Если $d^{+}(v)=0$, то вершина v называется *истоком*. Если $d^{-}(v)=0$, то вершина v называется *стоком*. Граф, имеющий только один источник и только один сток, называется *сетью*.

Если число ребер и число вершин бесконечно, то граф называется *бесконечным*. Если все локальные степени конечны, то граф называется *локально конечным* (*locally finite*).

Для неориентированных графов без петель и кратных ребер вводятся следующие определения и обозначения. Если u, v – вершины графа G, то через $\operatorname{dist}(u,v)$ обозначается расстояние между u и v, а через $G_i(v)$ – подграф графа G, индуцированный множеством вершин, которые находятся на расстоянии i в G от вершины v. Подграф $G_1(v)$ называется окрестностью вершины v и часто обозначается: $[v]$, или $\Gamma(v)$.

Теорема (Эйлера). *Пусть G – мультиграф с n вершинами и m ребрами, $d_i = d(v_i)$ – степень i-й вершины v_i. Тогда $\sum_{i=1}^{n} d_i = m$, т. е. сумма степеней вершин графа равна удвоенному количеству ребер.*

Доказательство. Пересчитаем число ребер в каждой вершине и сложим эти числа. Тогда каждое ребро будет подсчитано два раза. Поэтому общее число ребер мультиграфа равно половине этой суммы:

$$m = (1/2)\sum_{i=1}^{n} d_i \,.$$

Этот результат, известный еще более двухсот лет назад Эйлеру, часто называют *леммой о рукопожатиях*. Из нее следует, что если несколько человек обменялись рукопожатиями, то общее число рукопожатий обязательно четно, так как в каждом рукопожатии участвуют две руки (при этом каждая рука считается столько раз, сколько она участвует в рукопожатиях).

Следствие 1. *В каждом мультиграфе число вершин нечетной степени четно.*

Действительно, пусть $v_1,\ v_2,\ \dots,\ v_p$ – вершины с нечетной степенью, а $v_{p+1},\ v_{p+2},\dots,\ v_n$ – вершины с четной степенью. Тогда число $\sum_{i=1}^{p} d_i = \sum_{i=1}^{n} d_i - \sum_{i=p+1}^{n} d_i = 2m - \sum_{i=p+1}^{n} d_i$ – четное. Так как $d_i = 2k_i + 1$, $i = 1,\ 2,\dots,\ p$, то $\sum_{i=1}^{p} d_i = (\sum_{i=1}^{p} 2k_i) + p$ – четное число. Следовательно, число вершин с нечетной степенью *p* – четно.

Следствие 2. *Сумма полустепеней вершин орграфа равна удвоенному количеству дуг*:

$$\sum_{v\in V} d^{-}(v) + \sum_{v\in V} d^{+}(v) = 2m\,.$$

Доказательство. Сумма полустепеней вершин орграфа равна сумме степеней вершин графа, полученного из орграфа забыванием ориентации дуг.

Регулярный граф

Regular graph

В регулярном, или однородном графе, степени всех вершин равны. Регулярный граф с вершинами степени *k* называется *k-регулярным*, или *регулярным графом степени k*.

Примеры регулярных графов на рис. 3.

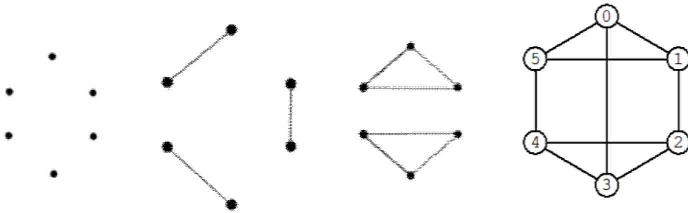


Рис. 3

0-регулярный граф состоит из одной или многих изолированных вершин (нуль-граф), 1-регулярный – из одного или многих изолированных рёбер, 2-регулярный – из треугольников (циклов).

Граф G называется *реберно регулярным графом* с параметрами (n, k, λ), если G содержит n вершин, является регулярным графом степени k и каждое ребро графа лежит точно в λ треугольниках. Граф G

называется *кореберно регулярным графом* с параметрами (n,k,μ), если G содержит n вершин, является регулярным графом степени k и подграф $[u]\cap[v]$ содержит точно μ вершин для любых не смежных вершин u и v.

Граф G называется *вполне регулярным графом* с параметрами (n,k,λ,μ), если G реберно регулярен с соответствующими параметрами и подграф $[u]\cap[v]$ содержит μ вершин в случае $\operatorname{dist}(u,v)=2$. Вполне регулярный граф диаметра 2 называется сильно регулярным графом.

Полный граф K_n является сильно регулярным для любого n.

Степень регулярности является инвариантом графа. *k*-регулярный граф на $2k+1$ вершинах имеет гамильтонов цикл.

Регулярные графы представляют особую сложность для многих алгоритмов.

Граф регулярен тогда и только тогда, когда вектор $u=(1,\ldots,1)$ есть собственный вектор матрицы смежности графа A. Его собственное число будет степенью графа. Собственные векторы, соответствующие другим собственным числам, ортогональны u, поэтому для собственных векторов $v=(v_1,\ldots,v_n)$ выполняется условие $\sum_{i=1}^{n} v_i = n$. Регулярный граф степени *k* связен тогда и только тогда, когда собственное число *k* имеет единичную кратность.

Полный граф

Complete graph

В полном неориентированном графе каждая пара вершин смежна. Полный граф с n вершинами есть регулярный граф степени $n-1$, имеет $n(n-1)/2$ рёбер. Обозначается K_n.

В полном ориентированном графе каждая пара вершин соединена парой дуг (с различными направлениями).

На рис. 4 показаны планарные графы $K_1 - K_4$, а на рис. 5 – непланарные графы K_5 и K_6. Полные графы с большим количеством вершин непланарные, если содержат подграф K_5 и, следовательно, не удовлетворяют критерию Понтрягина–Куратовского

Нуль-граф, или пустой граф, или тривиальный граф (*null graph*, *empty graph*, *edgeless graph*) – регулярный граф степени 0, т. е. граф без рёбер.

Нулевой граф O_n есть дополнительный граф для полного графа K_n, поэтому его можно обозначить $\bar{K}_n$.

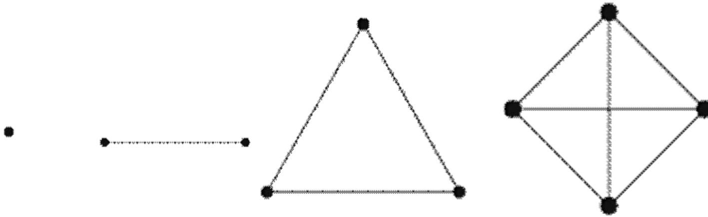


Рис. 4. Полные планарные графы: $K_1 - K_4$

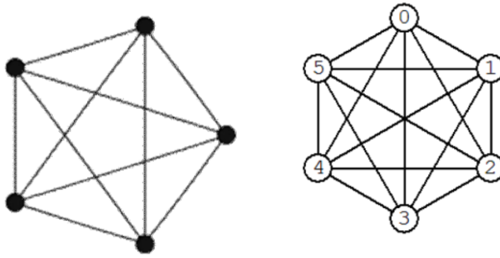


Рис. 5. Полные непланарные графы K_5 и K_6

Турниры

Tournament

Полный ориентированный граф называют *турниром*. Этот термин получил свое название от соревнований по круговой системе, графическое представление которых имеет структуру полного ориентированного графа. В турнирах по круговой системе играют несколько команд, каждая со всеми остальными по одному разу. Игра по правилам не может закончиться вничью. В графе вершины соответствуют командам, а дуга (u,v) присутствует в графе, если команда u победила команду v. В таком ориентированном графе нет параллельных дуг и петель, и между каждыми двумя вершинами имеется точно одна дуга. Граф является полным, а следовательно, и турниром.

Пример турнира приведен на рис. 6.

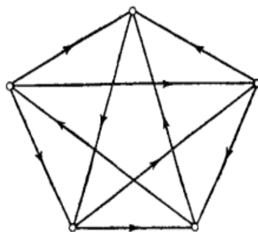


Рис. 6. Турнир

Участвующие в турнире команды можно ранжировать в соответствии с количеством очков. *Количество очков* команды соответствует числу побежденных ею противников. *Последовательностью очков* турнира на n вершинах называют последовательность $(s_1, s_2, ..., s_n)$, в которой каждое s_i является полустепенью исхода вершины турнира.

Предположим, что команды турнира по круговой системе можно упорядочить таким образом, что за каждой командой идет побежденная ею. Тогда для фиксации порядка можно будет присвоить командам целые числа $1, 2, ..., n$. Такое ранжирование всегда возможно, поскольку в турнире имеется ориентированный гамильтонов путь (см. ниже). Оно называется *ранжированием* гамильтоновым путем. Оно может отличаться от ранжирования по очкам. Более того, турнир может иметь не один ориентированный гамильтонов путь. В таком случае возможно более одного ранжирования гамильтоновым путем. Однако в транзитивном турнире существует точно один ориентированный гамильтонов путь.

Двудольный граф

Bigraph

Граф $G = (V, E)$ называется *двудольным*, или *биграфом*, если множество V его вершин разбито на два непересекающихся подмножества V_1 и V_2, так что $V = V_1 \cup V_2$ и $V_1 \cap V_2 = \varnothing$. Притом каждое ребро из E соединяет какую-нибудь вершину из V_1 с какой-нибудь вершиной из V_2. В двудольном графе необязательно, чтобы каждая вершина из V_1 соединялась с каждой вершиной V_2. Однако, если же это так, то граф называет-

ся *полным двудольным графом* и обозначается $K_{m,n}$, где m – число вершин в V_1, а n – в V_2. Граф $K_{m,n}$ имеет $(m+n)$ вершин и $m \times n$ ребер. Полный двудольный граф $K_{1,n}$ называется *звездным графом*.

Граф $K_{1,5}$ и граф $K_{4,3}$ изображены на рис. 7, а на рис. 8 – изоморфизмы графа $K_{3,3}$.

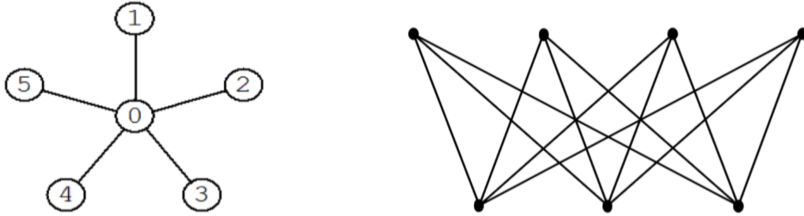


Рис. 7. Графы: $K_{1,5}$ и $K_{4,3}$

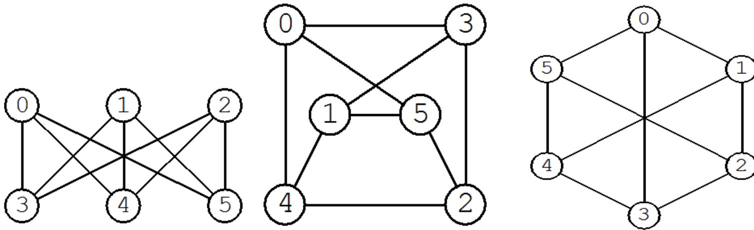


Рис. 8. Изоморфизмы графа $K_{3,3}$

Граф клешня

Claw

Клешнёй называется полный двудольный граф $K_{1,3}$ – звезда с тремя рёбрами, тремя листьями и одной центральной вершиной (рис. 9).

Граф без клешней – это граф, в котором никакой порождённый подграф не является клешнёй, то есть любое подмножество из четырёх вершин не имеет структуру звезды с тремя рёбрами, исходящими из центральной вершины. Также можно определить граф без клешней как граф, в котором окрестность любой вершины образует дополнение графа без треугольников.

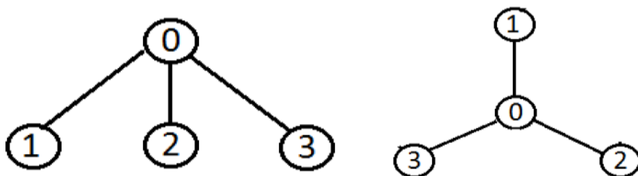


Рис. 9

Дополнительный граф

Complement of graph, complementary graph

Для произвольного графа *G* следующим образом определяется *дополнительный граф* (или *дополнение*) $\bar{G}$:

- $V(G) = V(\bar{G})$;
- любые две несовпадающие вершины смежные в $\bar{G}$ тогда и только тогда, когда они не смежные в *G* (рис. 10).

Очевидно $\bar{\bar{G}} = G$, $\bar{G} = \bar{H}$, если *G* = *H*.

Граф, изоморфный своему дополнению, называется *самодополнительным* (*self-complementary graph*). Самодополнительные графы составляют важный, хотя и экзотический, класс графов, определенным образом связанный с проблемой распознавания изоморфизма графов.

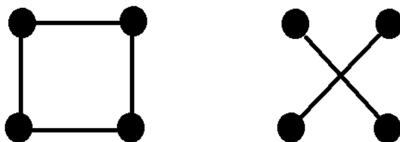


Рис. 10. Граф G и его дополнение $\bar{G}$

Рёберный граф

Рёберный граф *L*(*G*) для графа *G* есть граф со свойствами:

- любая вершина графа *L*(*G*) представляет ребро графа *G*;
- две вершины графа *L*(*G*) смежные тогда и только тогда, когда их соответствующие рёбра имеют общую вершину, т. е. смежные в *G*.

Пример построения

Рис. 11 показывает граф G и его рёберный граф L. Каждая вершина рёберного графа помечена двумя номерами вершин соответствующего ребра в исходном графе. Например, в L вершина с меткой (1,3) соответствует ребру слева между вершинами 1 и 3.

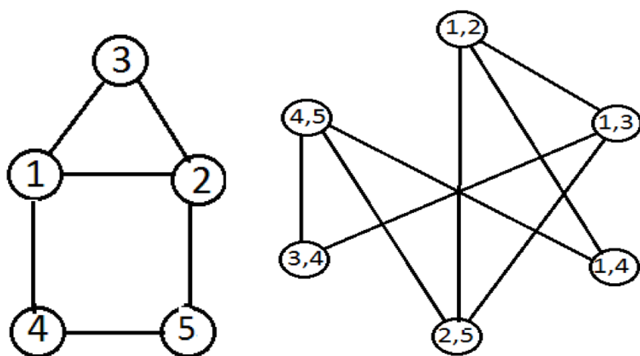


Рис. 11. Граф G и его рёберный граф L

Структура графа G полностью определяется рёберным графом: граф G может быть восстановлен из рёберного графа.

Хордальный граф, или *триангулированный граф*, есть рёберный граф полного графа K_n.

Рёберные графы выпуклых многогранников

Рёберный граф для графа тетраэдра есть граф октаэдра. Рёберный граф куба есть граф кубооктаэдра. Рёберный граф додекаэдра есть граф икосододекаэдра, и т. д. Геометрически операция состоит в отсечении всех вершин многогранника плоскостью, пересекающей все рёбра, сопряжённые с вершиной в их середине. Если многогранник не простой (то есть имеет больше трёх граней на вершину), рёберный граф не будет плоским.

Срединный граф

Срединный граф – это вариант рёберного графа для плоского графа. В срединном графе две вершины смежные в том и только в том случае, когда соответствующие рёбра исходного графа являются двумя последовательными рёбрами некоторой области плоского графа. Для простых многоугольников срединный граф и рёберный граф совпадают, но для сложных многоугольников срединный граф остаётся плоским. Так, срединные графы куба и восьмигранника изоморфны графу кубооктаэдра, а срединные графы двенадцатигранника и икосаэдра изоморфны графу икосододекаэдра.

Свойства реберных графов

Свойства графа G, зависящие только от смежности рёбер, могут быть переведены в эквивалентные свойства графа $L(G)$, зависящие только от смежности вершин. Например, паросочетание в G – это множество дуг, ни

одна из которых не смежна другой, и соответствующее множество вершин в $L(G)$, ни одна из которых не смежна другой, то есть независимое множество вершин.

Рёберный граф связного графа связен. Если G связен, он содержит путь, соединяющий любые два его ребра, что переводится в путь графа $L(G)$, содержащий любые две вершины графа $L(G)$. Тем не менее, графу G, содержащему изолированные вершины, а посему несвязному, может соответствовать связный рёберный граф. Задача о максимальном независимом множестве для рёберного графа соответствует задаче нахождения максимального паросочетания в исходном графе.

- Рёберное хроматическое число графа G равно вершинному хроматическому числу его рёберного графа $L(G)$.
- Рёберный граф рёберно-транзитивного графа является вершинно-транзитивным графом.
- Если граф G имеет эйлеров цикл (то есть G связен и имеет чётное число рёбер в каждой вершине), то его рёберный граф является гамильтоновым графом. (Однако не все гамильтоновы циклы в рёберном графе получаются из эйлеровых циклов.)
- Рёберный граф не имеет клешней.

Граф G является рёберным графом какого-либо другого графа в том и только в том случае, когда можно найти набор клик в G, разбивающих дуги графа G так, что каждая вершина G принадлежит в точности двум кликам. Может случиться, что для достижения этого потребуется отдельные вершины выделить в клики. Если G не является треугольником, может быть только одно разбиение такого рода. Если разбиение существует, можно построить граф (для которого G будет рёберным графом) путём создания вершины для каждой клики и соединением полученных вершин ребром, если вершина принадлежит обоим кликам.

Обобщение на реберные орграфы

Можно также обобщить рёберные графы на ориентированные графы. Если G – ориентированный граф, то его *ориентированный рёберный граф*, или *рёберный орграф*, имеет одну вершину для каждой дуги из G. Две вершины, соответствующие дугам (u,v) и (w,x) графа G связаны дугой (u,x) в рёберном орграфе, когда $v = w$. Таким образом, каждая дуга в рёберном орграфе соответствует пути длиной 2 в исходном графе.

Обобщение на мультиграфы

Концепция рёберного графа для графа G может быть естественным образом распространена на случай, когда G является *мультиграфом*.

Отношения в графе

Бинарное отношение R называется *рефлексивным* (*reflexive relation*), если $\forall a \in X : (aRa)$. То есть всякий элемент этого множества находится в отношении R с самим собой.

Свойство рефлексивности при отношениях, заданных графом, состоит в том, что каждая вершина имеет петлю – дугу (x, x), а матрица смежности этого графа на главной диагонали имеет единицы.

Если это условие не выполнено ни для какого элемента множества X, то отношение R называется *антирефлексивным.*

Отношение R называется *антирефлексивным* (*irreflexive relation*), если $\forall a \in X : \neg(aRa)$.

Если антирефлексивное отношение задано графом, то ни у одной вершины не будет *петли* – дуги (x, x), а в матрице смежности на главной диагонали будут нули.

Операции над графами

Существуют операции над графами, которые позволяют получать графы как с меньшим, так и с большим количеством вершин и/или ребер: унарные и бинарные операции.

Унарные операции

Unary operations

Стягивание ребра, слияние вершин

Edge contraction, vertex involving

Стягивание ребра (u, v) означает отождествление смежных вершин u и v, замена инцидентных вершин (концов ребра) одной вершиной.

Пусть u и v – две вершины графа G, граф $H = G - u - v$, затем к H присоединяется новая вершина w, соединенная ребром с каждой из вершин, входящих в объединение окружений вершин u и v в графе G.

Граф, полученный из графа G стягиванием ребра (u, v), обозначается G / uv. Говорят, что построенный граф получается из графа G *отождествлением вершин u к* v.

На рис. 12 показаны граф G и граф, полученный из G стягиванием ребра $(3, 5)$, где стянутая вершина $\overline{3} = (3, 5)$.

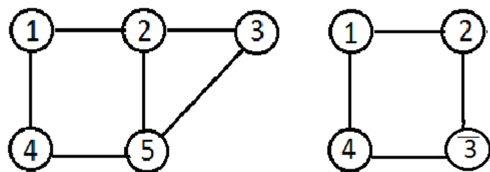


Рис. 12

Граф *G* называется *стягиваемым к графу H*, если *H* получается из *G* в результате некоторой последовательности стягиваний ребер.

Например, граф Петерсена стягиваем к K_5 и, стало быть, к любому K_n с $n < 5$. Очевидно, что любой непустой связный граф, отличный от K_1, стягиваем к K_2. Но уже не любой связный граф стягивается к графу K_3. Например, простая цепь P_n не стягивается к K_3.

Вводится параметр $\eta(G)$ – максимум порядков полных графов, к которым стягивается граф *G*. Параметр $\eta(G)$ называется *числом Хадвигера графа G*. Это число связано с проблемой четырех красок.

Расщепление вершины

Vertex splitting

В определенном смысле двойственной к операции стягивания ребра является операция *расщепления вершины*.

Пусть u – одна из вершин графа *G* (рис. 13).

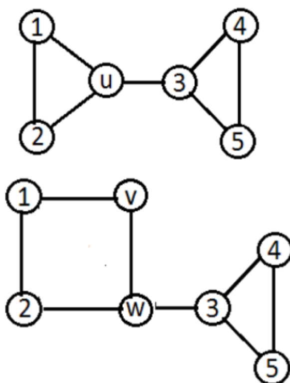


Рис. 13

Окружение вершины u заданным или произвольным образом разбивается на две части: Γ_1 и Γ_2. Выполняется следующее преобразование графа. Удаляется вершина u вместе с инцидентными ей ребрами, добавляются новые вершины v, w и соединяющее их ребро (v, w); вершина v соединяется ребром с каждой вершиной из множества Γ_1, а вершина w – с каждой вершиной из множества Γ_2.

Полученный в результате граф обозначается символом $\tilde{G}$. Используется терминология: граф $\tilde{G}$ получается из графа G *расщеплением вершины* u.

Бинарные операции

Binary operations

Пусть есть два графа $G_1(V_1, E_1)$ и $G_2(V_2, E_2)$.

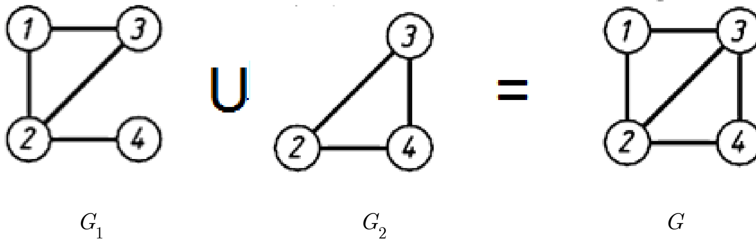
Объединение графов $\cup$

Graph union

Объединением называется граф $G = G_1 \cup G_2$, множество вершин которого есть $V = V_1 \cup V_2$ и множество ребер – $E = E_1 \cup E_2$.

Объединение $G = G_1 \cup G_2$ называется *дизъюнктным*, если $G_1 \cap G_2 = \varnothing$. Аналогично определяются *объединение* и *дизъюнктное объединение* (*disjunct union of graphs*) любого множества графов, причем никакие два из объединяемых графов не должны иметь общих вершин.

Пример операции приведен на рис. 14.



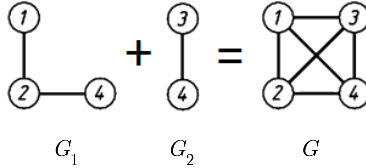
G_1 G_2 G

Рис. 14. $G = G_1 \cup G_2$

Соединение графов $+$

Join of graphs

Соединение $G = G_1 + G_2$ состоит из $V = V_1 \cup V_2$ и всех ребер, соединяющих вершины V_1 и V_2. Пример операции приведен на рис. 15.



G_1 G_2 G

Рис. 15. $G = G_1 + G_2$

Произведение графов $\times$

Product of graphs

Произведением называется граф $G = G_1 \times G_2$, множество вершин которого есть декартово произведение множеств вершин исходных графов $V = V_1 \times V_2$, а множество ребер определяется следующим образом: вершины $u = (u_1, u_2)$ и $v = (v_1, v_2)$ смежные в *G* тогда и только тогда, когда $u_1 = v_1$ и u_2 смежная v_2 или $u_2 = v_2$ и u_1 смежная v_1.

Декартово (или *прямое*) произведение двух множеств – это множество, элементами которого являются все возможные упорядоченные пары элементов исходных множеств.

В результате операции произведения

- количество вершин $V = V_1 \times V_2$,
- количество ребер $E = V_1 \times E_2 + V_2 \times E_1$.

Пример операции произведения приведен на рис. 16.

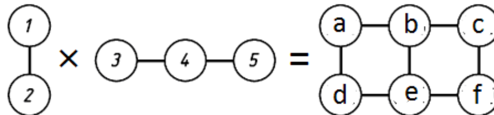


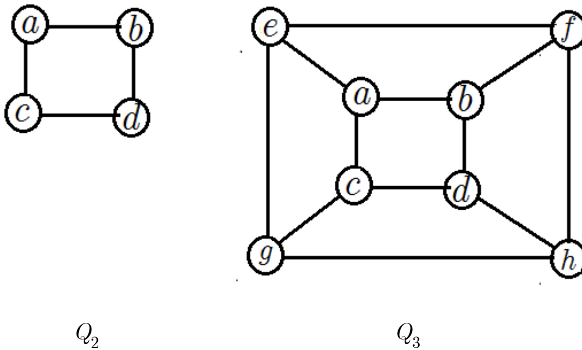
Рис. 16. $G = G_1 \times G_2$:

$a = 1 \cdot 3,\ b = 1 \cdot 4,\ c = 1 \cdot 5,\ d = 2 \cdot 3,\ e = 2 \cdot 4,\ f = 2 \cdot 5$

Операция произведения позволяет вводить класс графов, так называемые *n*-мерные кубы. *n-мерный куб* Q_n определяется рекуррентно:

$$Q_1 = K_2 \,,\; Q_n = Q_{n-1} \times K_2 \,,\; n > 1 \,.$$

Q_n – граф порядка 2^n, вершины которого можно представить (0, 1)-векторами длины *n* таким образом, что две вершины будут смежные тогда и только тогда, когда соответствующие векторы различаются ровно в одной координате. Поскольку каждая вершина *n*-мерного куба инцидентна *n* ребрам, то число его ребер $= n2^{n-1}$. На рис. 17 представлены кубы.



Q_2 Q_3

Рис. 17

Обозначения вершин. Q_2 : $a = (0,1)$; $b = (1,1)$, $c = (0,0)$, $d = (1,0)$; Q_3 : $a = (0,0,1)$, $b = (0,1,1)$, $c = (0,0,0)$, $e = (1,0,1)$, $f = (1,1,1)$, $g = (1,0,0)$, $h = (1,1,0)$.

Композиция графов

Composition of graphs

Композиция $G = G_1[G_2]$ определяется как граф *G*, имеющий множество вершин $V = V_1 \times V_2$, где вершина $u = (u_1, u_2)$ смежна $v = (v_1, v_2)$ тогда и только тогда, когда u_1 смежна v_1 и $v_1 = v_2$. $G_1[G_2] \neq G_2[G_1]$.

На рис. 18 приведен пример операции композиции $G_1[G_2]$.

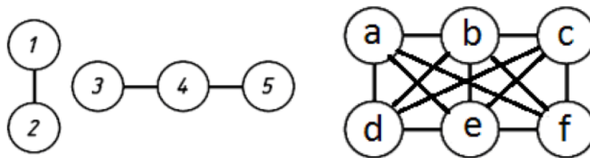


Рис. 18. $G_1[G_2]$:
$a = 1\,\&\,3$, $b = 1\,\&\,4$, $c = 1\,\&\,5$, $d = 2\,\&\,3$, $e = 2\,\&\,4$, $f = 2\,\&\,5$

Маршрут, цепь, путь, цикл

Walk, trail, path, cycle

Пусть $G = G(V,E)$ – граф с вершинами $v_1, v_2, \ldots, v_n \in V$ и ребрами $e_1, e_2, \ldots, e_m \in E$.

Как ни странно, терминология несколько отличается как в русской, так и в англоязычной литературе. Поэтому следует уточнять, какой из них пользуется автор.

Маршрут (*walk*) – это чередующаяся последовательность вершин и рёбер $v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$ в которой любые два соседних элемента инцидентны. Если два рядом стоящие ребра ориентированные, то в инцидентную им вершину ребро, стоящее слева, должно входить, а ребро, стоящее справа, из нее выходить. Маршрут соединяет вершины v_1 и v_k, которые называются соответственно *началом и концом* маршрута; вершины $v_2, \ldots, v_{k-1}$ называются *промежуточными*. Маршрут называют *замкнутым*, если $v_1 = v_k$, иначе – *открытым*. Маршрут неориентированного графа называют *неориентированным маршрутом*, а маршрут орграфа называют *ориентированным маршрутом*.

Вершины и ребра в маршруте могут повторяться. Отличие маршрута от произвольной последовательности вершин и ребер в том, что по маршруту можно непрерывно пройти от начальной вершины до конечной.

Trail (*цепь*) есть маршрут, в котором все ребра различны.

Path (*путь*) есть *trail*, в котором вершины не повторяются.

В англоязычной литературе для ориентированных графов просто добавляется слово *directed*, и не используется иная терминология.

Часто используется терминология: *trail* – путь, *path* – простой путь.

Длина пути – количество рёбер в маршруте, с возможными повторениями – равна $k-1$.

Цикл – это замкнутая цепь, в котором $v_1 = v_k$.

Контур – цикл, замкнутый путь в орграфе.

Простой цикл – цикл, в котором все вершины различны.

Цикл из k вершин обозначают C_k. Цикл – связный регулярный граф степени 2.

Граф без циклов называется *ациклическим*.

Аналогично, как и для графа, для орграфа вводятся понятия: ориентированный путь, ориентированный цикл.

Эйлеров цикл – цикл, содержащий все ребра графа. Вершины могут повторяться.

Гамильтонов цикл – простой цикл, содержащий все вершины графа, т. е. различны все ребра и все вершины.

Вершина u *достижима* из вершины v, если существует путь из v в u. *Длина пути* равна числу его ребер.

Расстояние между двумя вершинами – это длина кратчайшего пути, соединяющего эти вершины.

Пример (рис. 19).

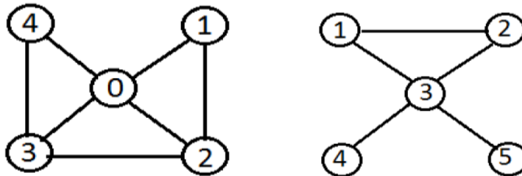


Рис. 19

Последовательность вершин: 1, 2, 4, 5 – не маршрут; 1, 2, 3, 4, 0, 3, 4 – маршрут, но *не* путь; 2, 0, 3, 4, 0, 1 – *не* простой путь; 1, 2, 0, 3, 2, 0, 1 – замкнутый маршрут, но не цикл; 1, 2, 0, 3, 4, 0, 1 – *не* простой цикл; 1, 2, 3, 4, 0, 1 – простой цикл.	The vertex sequence (4, 3, 1, 3, 5) is a walk that is not a trail; the vertex sequence (4, 3, 1, 2, 3, 5) represents a trail that is not a path; the vertex sequence (4, 3, 1, 2) represents a path.

Некоторые элементарные свойства маршрутов

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.

Если в графе степень *каждой* вершины не меньше 2, то в нем есть цикл.

Доказательство. Найдем в графе простой путь наибольшей длины. Пусть это $v_1, v_2, \ldots, v_n$. Вершина v_n смежна с v_{n-1}, а так как ее степень не меньше 2, то она смежна еще хотя бы с одной вершиной, скажем с u. Если u была бы отлична от всех вершин пути, то последовательность $v_1, v_2, \ldots, v_n$ была бы простым путем большей длины. Следовательно, u – это одна из вершин пути, $u = v_i$, где $i < n-1$. Но тогда $v_i, \ldots, v_n, v_i$ – цикл.

Изоморфизм графов

Graph isomorphism

Некоторые графы не всегда следует различать. Как в применениях теории графов, так и в самой этой теории чаще существенно лишь существование объектов (вершин графа) и связей между объектами (ребрами). Из этих воззрений графы, которые получаются один из другого изменением наименований вершин, рационально не различать. Эти представления записываются в виде следующего определения.

Пусть *G* и *H* – графы, а $\varphi : G \to H$ – биекция. (*Биекция* – это взаимно однозначное отображение одного множество в другое). Если для любых вершин u и v графа *G* их образы $\varphi(u)$ и $\varphi(v)$ смежные в *H* тогда и только тогда, когда *u* и v смежные в *G*, то эта биекция называется *изоморфизмом графа G на граф H*. Если такой изоморфизм существует, графы *G* и *H изоморфны*. Он обозначается как $G \cong H$, или проще $G = H$.

Например, три графа, представленные на рис. 8, изоморфны. Вопрос о том, изоморфны ли два данных графа, в общем случае оказывается весьма сложным.

Очевидно, что отношение изоморфизма графов является эквивалентностью, т. е. оно симметрично, транзитивно и рефлексивно.

Следовательно, множество всех графов разбивается на классы так, что графы из одного класса попарно изоморфны, а графы из разных классов не изоморфны. Изоморфные графы естественно отождествлять, т. е. считать совпадающими и можно изобразить одним рисунком. Они могли бы различаться конкретной природой своих элементов, но именно это игнорируется при введении понятия «граф».

Если каждая вершина графа и/или ребра помечена, в частности пронумерована, то такой граф называется *помеченным* (редко *нагруженным*). В качестве меток обычно используются целые числа или буквы. Если каждая дуга помечена, то это *помеченный орграф*.

Подграф графа G называется максимальным, если он получается из G удалением одной вершины и всех связанных с нею ребер. Вводится обозначение максимального подграфа, получающегося из графа G удалением вершины v: $G-v$. Через $G-H$ обозначается подграф, получающийся из графа G удалением всех вершин из H. Список максимальных подграфов графа G иногда называют его *колодой*.

В некоторых ситуациях все же приходится различать изоморфные графы, и тогда полезно понятие «помеченный граф».

Отождествив каждую из вершин графа с ее номером и, следовательно, множество вершин – с множеством чисел $\{1, 2, ..., n\}$, определяется *равенство* помеченных графов G и H одного и того же порядка: $G = H$ тогда, когда $E(G) = E(H)$. На рис. 20 изображены три разных помеченных графа.

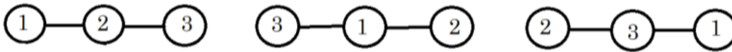


Рис. 20

Если рассматриваемые графы различаются лишь с точностью до изоморфизма, вводится термин «абстрактный граф». *Абстрактный* (или *непомеченный*) граф – это класс изоморфных графов.

Перечисление графов

Рассмотрим перечисление только помеченных графов, поскольку эти задачи решаются значительно легче, чем соответствующие задачи для непомеченных объектов.

В помеченном графе порядка n вершинам приписываются целые числа от 1 до n. Например, граф, изображённый на рис. 21, может быть помечен шестью различными способами.

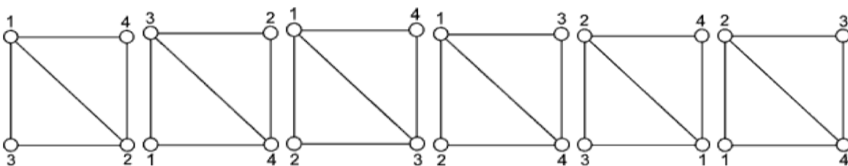


Рис. 21

Два помеченных графа G_1 и G_2 считаются изоморфными тогда и только тогда, когда существует взаимно однозначное отображение мно-

жества $V(G_1)$ на множество $V(G_2)$, сохраняющее не только смежность, но и распределение пометок.

Задача о числе помеченных графов с данным числом вершин и рёбер формулируется следующим образом. Если V – множество из n вершин, то существует $M = C_n^2 = n(n-1)/2$ различных неупорядоченных пар этих вершин. Любая пара вершин является либо смежной, либо нет. Следовательно, число помеченных графов с m рёбрами равно C_M^m.

Пусть $G_n(x)$ – многочлен, у которого коэффициент при x^m равен числу помеченных графов порядка n, имеющих m рёбер. Такой многочлен называется производящей функцией для помеченных графов с данным числом вершин и рёбер.

Производящая функция $G_n(x)$ для помеченных графов порядка *n* задаётся соотношением $G_n(x) = \sum_{m=0}^{M} C_M^m x^m = (1+x)^M$.

Так как $G_n(x) = (1+x)^M$ и число G_n помеченных графов порядка *n* равно $G_n(1)$, то $G_n(x) = 2^M$, $M = C_n^2 = n(n-1)/2$.

Например, для $n=3$ существуют 8 помеченных графов и только 4 непомеченных графа. Для $n=4$ существуют 64 помеченных графов и только 11 непомеченных графа.

Для $n=3$ эта формула иллюстрируется рис. 22.

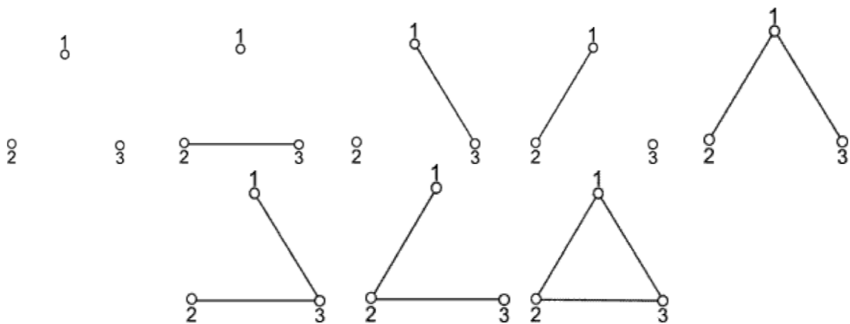


Рис. 22

Для ориентированных помеченных графов производящая функция $\bar{G}_n(x)$ получается аналогично. Однако существует $\bar{M} = 2C_n^2 = n(n-1)$

упорядоченных пар вершин, и тогда число помеченных орграфов порядка n, имеющих m рёбер, есть $C_{\bar{M}}^{m}$.

Производящая функция $\bar{G}_n(x)$ для помеченных орграфов порядка n задаётся соотношением $\bar{G}_n(x) = \sum_{m=0}^{\bar{M}} C_{\bar{M}}^{m} x^m = (1+x)^{\bar{M}}$.

Число g_n непомеченных графов порядка n определяется сложно. Известна асимптотика *Пойа* $g_n \sim 2^{C_n^2} / n!$. Поэтому число помеченных графов порядка n в $\sim n!$ раз больше числа непомеченных. Этот факт кажется естественным – существует $n!$ перестановок множества, состоящего из n вершин. Последнее утверждение совсем не означает, что из каждого непомеченного графа получается $n!$ помеченных графов. Однако в основном, каждый непомеченный граф приводит к $n!$ помеченным графам.

Реконструируемость

Граф H является *реконструкцией* графа G, если их колоды совпадают. Граф называется реконструируемым, если он изоморфен каждой своей реконструкции. Популярность задачи реконструируемости графов связана с известной гипотезой.

Гипотеза (Келли, Улам, 1945). *Каждый неориентированный граф с числом вершин, большим двух, является реконструируемым.*

Очевидно, что по колоде графа можно восстановить количество его вершин и ребер. Нетрудно доказать, что по колоде неориентированного графа можно восстановить и его вектор степеней. Исключений из гипотезы для неориентированных графов не известно, поэтому считается, что она для них выполняется. Для ориентированных графов известны бесконечные семейства пар нереконструируемых орграфов.

Представление графов в компьютере

Graph representation

Существуют четыре основных способа представления графов в памяти компьютера. Выбор оптимального представления определяется условиями конкретной задачи. Они различаются объемом занимаемой памяти и скоростью выполнения операций над графами. Во многих задачах на графах выбор представления оказывает решающее значение эффективности алгоритмов. Существует разные комбинации составляющих графа: вершина–ребра.

1. Матрица смежности графа (или матрица весов графа).
2. Структуры (списки) смежных вершин графа.
3. Матрица инцидентности графа: вершина–ребра.
4. Список ребер графа.

В основном используются матрица смежности и списки смежных вершин. Однако следует подчеркнуть, что при малых $n <\sim 10$ форма представления не имеет значения – времена слишком малы.

Матрица смежности графа

Adjacency matrix

Матрицей смежности помеченного графа с *n* вершинами называется матрица $A_{n,n}$, элементы которой есть

$$a_{ij} = \begin{cases} 1, & \text{если вершина } i \text{ смежна с вершиной } j; \\ 0, & \text{если вершина } i \text{ не смежна с вершиной } j. \end{cases}$$

Матрица смежности однозначно определяет граф. Для неориентированного графа матрица А симметрична относительно главной диагонали. Если петель нет, то диагональные элементы = 0. Иначе одна петля в матрице смежности может быть представлена соответствующим диагональным элементом, например, = 2. Для мультиграфа $a_{i\,j} =$ числу кратных ребер, соединяющих вершины v_i и v_j.

Для матрицы смежности неориентированного графа имеют место соотношения: $\sum_{i=1}^{n} a_{ij} = d(v_j)$, $\sum_{i,j=1}^{n} a_{ij} = 2m$ $(m = \mid E \mid)$.

Для ориентированного графа вершина i *смежная* с вершиной j, если дуга выходит из i вершины и входит в j вершину, иначе – *несмежная*. Для выходящей дуги $a_{ij} = 1$, а для входящей – либо $a_{ij} = 0$, либо $a_{ij} = -1$. Тогда имеют место соотношения:

$$\sum_{a_{i,j=1}>0}^{n} a_{ij} = outd(v_i)$$ – полустепень исхода,

$$\sum_{a_{i,j=1}<0}^{n} a_{ij} = ind(v_i)$$ – полустепень захода (входа),

$$\sum_{i,j=1}^{n} | a_{ij} | = 2m .$$

Матрица смежности наиболее удобное, наглядное и используемое представление. Оно позволяет получить прямой доступ к ребрам графа. Независимо от числа ребер объем занимаемой памяти составляет $n \times n$ или $n^2 / 2 - n$, если использовать симметрию и хранить только треугольную подматрицу матрицы смежности неориентированного графа. Обычно подчеркиваемый недостаток – большая требуемая память – зачастую значительно преувеличен. Даже для экзотического графа с $n = 10^4$ необходимо всего ~ 10^8 байт. Кроме того, каждый элемент матрицы часто можно представить одним битом, но при этом увеличивается время доступа. С использованием численных методов разреженных матриц теряется простота, которая делает матрицу смежности такой привлекательной, и, что более критично, время исполнения на разреженных графах остается по своей сути квадратичным.

Пример. Матрицы смежности для неориентированного, ориентированного графа, мультиграфа (с кратными ребрами), псевдографа (с петлями).

Неориентированный	Ориентированный	Мультиграф (с кратными ребрами)	Псевдограф (с петлями)
$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 1 \\ 0 & -1 & 0 & -1 \\ 1 & -1 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 0 & 3 \\ 2 & 0 & 1 & 3 \\ 0 & 1 & 0 & 1 \\ 3 & 3 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 2 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$

Матрица весов графа

Weighted matrix or weighted graph

Веса могут быть приписаны как ребрам, так и вершинам. Чаще рассматриваются веса ребер. Тогда простой взвешенный граф может быть представлен своей матрицей весов $W_{n,n}$, где w_{ij} – вес ребра, соединяющего вершины $i, j = 1,2,...,n$. Популярные обозначения: w_{ij} (от *weight* – вес) или d_{ij} (от *distance* – расстояние). Веса несуществующих ребер полагаются равными ∞ или 0 в зависимости от задачи. Матрица весов – естественное обобщение матрицы смежности.

Списки смежности вершин графа

Граф может быть представлен структурой смежности (*adjacency* – *смежность*). Такие структуры могут быть реализованы различными способами.

Распространен способ, в котором определяется массив указателей `Adj[n]`, каждый элемент которого `Adj[i]` начинает так называемый линейно связанный список смежности – соседних вершин, смежных с вершиной `i`.

Список смежных вершин графа дает компактное представление для разреженных графов с $m << n^2$. Недостаток этого представления заключается в лишних действиях. Если необходимо узнать, есть ли в графе ребро (u,v), приходится просматривать весь список $Adj[u]$ в поисках v. Объем требуемой памяти составляет $n+m$ (для неориентированных графов) и $n+2m$ (для ориентированных) единиц памяти, где n – число вершин графа, m – число ребер (дуг) графа.

Если в основе алгоритма решения задачи лежат операции добавления и удаления вершин из списков, то хранение списков смежности удобно реализовать, используя связанные списки.

Второй способ реализации структуры смежности заключается в использовании матриц вместо списков. Можно ввести два одномерных массива. В одном из них размером n записывается сумма степеней всех вершин. В другом массиве размером $2m$ записываются все смежные вершины, но индекс элемента для заданной вершины определяется элементом из первого массива. Такой способ использовался в языках, в которых списки сложно реализовать, например в фортране.

В третьем способе можно найти максимум степеней вершин maxd и определить матрицу A[n][maxd]. Самый удобный и надежный способ.

В четвертом способе используется ступенчатая матрица.

Последний элемент может определяться либо дополнительным одномерным массивом, либо характерной величиной последнего элемента. Функции создания прямоугольной и ступенчатой матриц описаны в приложении.

Список ребер графа

При описании графа списком его ребер (дуг) каждое ребро представляется парой инцидентных ему вершин. Это представление можно реализовать двумя одномерными массивами, одним двумерным массивом или одномерным массивом структур:

```
struct LISTEDGE{int u, v;};
LISTEDGE graph[m];
```

где m – количество ребер в графе.

Каждый элемент в массиве есть метка (номер) вершины, а i-е ребро графа выходит из вершины u_i и входит в вершину u_j. Объем занимаемой памяти составляет в этом случае $2m$ единиц памяти. Существенный недостаток заключается в значительном числе шагов, необходимом для получения множества вершин, к которым ведут ребра из данной вершины. Этим способом удобно вводить граф, а затем преобразовывать список в матрицу смежности.

Матрица инцидентности графа

Матрицей инцидентности называется матрица $B_{n,m}$, где n – число вершин, m – число ребер графа. Строки соответствуют вершинам, а столбцы – ребрам. Элемент матрицы в неориентированном графе равен

$$b_{ij} = \begin{cases} 1, & \text{если вершина } i \text{ инцидентна ребру } j; \\ 0, & \text{если вершина } i \text{ не инцидентна ребру } j. \end{cases}$$

В случае ориентированного графа строки матрицы также соответствуют вершинам, а столбцы – дугам. С n вершинами и m дугами элемент матрицы инцидентности равен

$$b_{ij} = \begin{cases} 1, & \text{если дуга } j \text{ выходит из вершины } i; \\ -1, & \text{если дуга } j \text{ входит в вершину } i; \\ 0, & \text{если вершина } i \text{ не инцидентна ребру } j. \end{cases}$$

Матрица инцидентности однозначно определяет структуру графа. В каждом столбце матрицы B лишь две единицы. Равных столбцов нет.

Недостаток данного представления состоит в том, что требуется $n \times m$ единиц памяти, большинство из которых будет занято нулями. Неудобен доступ к информации. Например, для ответа на вопросы: «есть ли в графе дуга (i, j)?» или «к каким вершинам ведут ребра из вершины i ?» – может потребоваться перебор всех столбцов матрицы. Этот способ, по-видимому, если и применяется, то весьма редко.

Примеры

Неориентированный граф

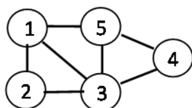


Рис. 23

Матрица смежности					
$u \setminus v$	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	0	0
3	1	1	0	1	1
4	0	0	1	0	1
5	1	0	1	1	0

Списки смежности	
	$v \rightarrow$ list
1 → | 2 | 3 | 5 |	1 → 2, 3, 5
2 → | 1 | 3 |	2 → 1, 3
3 → | 1 | 2 | 4 | 5 |	3 → 1, 2, 4, 5
4 → | 3 | 5 |	4 → 3, 5
5 → | 1 | 3 | 4 |	5 → 1, 3, 4

Списки ребер							
u,v	1,2	1,3	1,5	2,3	3,4	3,5	4,5

Матрица инцидентности							
u\(u,v)	1,2	1,3	1,5	2,3	3,4	3,5	4,5
1	1	1	1	0	0	0	0
2	1	0	0	1	0	0	0
3	0	1	0	1	1	1	0
4	0	0	0	0	1	0	1
5	0	0	1	0	0	1	1

Ориентированный граф

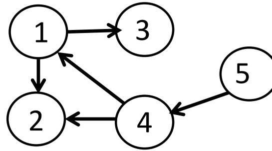
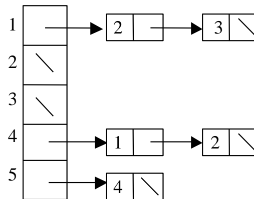


Рис. 24

Матрица смежности					
	1	2	3	4	5
1	0	1	1	–1	0
2	–1	0	0	–1	0
3	–1	0	0	0	0
4	1	1	0	0	–1
5	0	0	0	1	0

Списки смежности



Списки ребер					
u,*v*	1,2	1,3	4,1	4,2	5,4

Матрица инцидентности					
u\(u,v)	1,2	1,3	4,1	4,2	5,4
1	1	1	–1	0	0
2	–1	0	0	–1	0
3	0	–1	0	0	0
4	0	0	1	1	–1
5	0	0	0	0	1

Обходы графа

Traversal of graph

Обход графа – это прохождение его вершин и ребер по определенной системе. При обходе графа происходит перебор ребер (дуг) и перебор всех вершин. При этом происходит обработка графа (его вершин и ребер) либо в процессе прохождения, либо после прохождения. Поэтому обход графа лежит в основе многих алгоритмов исследования структуры и свойств графа. Если при посещении вершины структура графа не меняется, то используются два основных способа обхода: обход в глубину и обход в ширину.

Обход в глубину

Depth First Search, *DFS*

Пусть задан граф (неориентированный или ориентированный) и некая начальная вершина. Стратегия поиска в глубину состоит в том, чтобы, начиная с начальной вершины, идти в глубину, пока это возможно, т. е. пока не пройдены все исходящие ребра. И возвращаться и искать другой путь, когда таких ребер нет. Так делается, пока не обнаружены все вершины, достижимые из исходной. Если после этого остаются необнаруженные вершины, выбирается одна из них (как начальная) и процесс повторяется. Так делается до тех пор, пока не будут обнаружены все вершины графа. Тем самым определяются также компоненты связности.

В ориентированном графе движение вперед (вглубь) возможно только по направлению дуги. Возвращение происходит против ориентации. В неориентированном графе таких ограничений нет. В неориентированном графе обход можно начинать с любой вершины. Топология полученного остовного дерева зависит от начальной вершины.

Сложность поиска в глубину. Поскольку для каждой вершины, которую проходим впервые, выполняется обращение к функции один раз, то всего обращений будет n. При каждом обращении количество производимых действий пропорционально числу ребер, инцидентных рассматриваемой вершине. Поэтому сложность поиска составляет $O(n+m)$.

Рекурсивная функция `DFS` *обхода в глубину*

```
A - матрица смежности (adjacency matrix);
n - размер графа;
k - номер начальной вершины;
Count[j] - номер шага для j вершины;
count - текущий номер шага;
```

```
Vertex[count] - номер вершины на count шаге;
connect - номер связного подграфа;
Connect[i] - номер связного подграфа для j вершины.
```

Функция использует матрицу смежности.

```
    int DFS(int **A,int *Vertex,int n,
    int *Count,int &count,int *Connect,int connect,int i)
{   int j; count++; Count[i]=count;
    Connect[i]=connect;
    // предыдущая вершина i помечается как пройденная
    Vertex[count]=i;
    for(j=0; j<n; j++)//просмотр матрицы смежности вершины
 {  if(A[i][j]>0 && Count[j]== -1)
    // выбор из матрицы смежности непройденной вершины
  { u[count]=i; v[count]=j;   // u -> v is edge
    DFS(A,Vertex,n,Count,count,Connect,connect,j);
  } // if
 }  // j exit if j=n
    return j; } // DFS
```

Функция WalkDFS, вызывая DFS, обходит все вершины и определяет компоненты связности.

```
    int WalkDFS(int **A,int n,int v0)
 {  int i,k,count,connect,*Count,*Vertex,*Connect;
    Count=(int*)calloc(n,4);
    Vertex=(int*)calloc(n,4); Connect=(int*)calloc(n,4);
    connect=0; count= -1;
    for(i=0; i<n; i++) Count[i]= -1;
// сначала все вершины графа помечаем как непройденные
    DFS(A,Vertex,n,Count,count,Connect,connect,v0);
    while(count<n - 1)
    for(i=0; i<n; i++) //выбор непройденной вершины
    if(Count[i]==-1)
  { connect++;
    DFS(A,Vertex,n,Count,count,Connect,connect,i); }
    for(i=0; i<n; i++)
    printf("Count[%2i]=%2i %3i\n",i,Count[i],Connect[i]);
    printf("\n");
    for(k=0; k<n; k++)
    printf("Vertex[%2i]=%3i%3i\n",k,Vertex[k],Connect[i]);
    printf("\n");
    return connect; } // WalkDFS
```

Пример

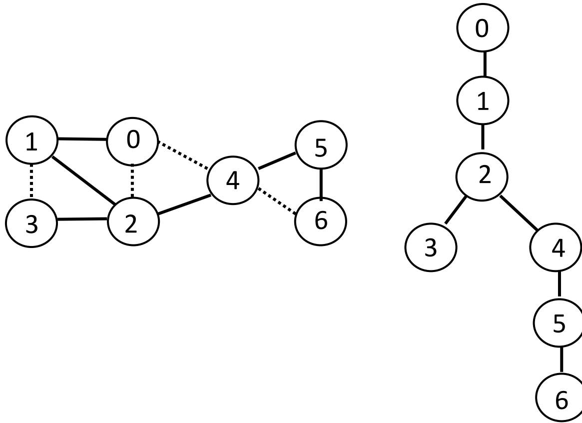


Рис. 25. Граф и его дерево обхода в глубину. Пунктирные ребра не проходятся

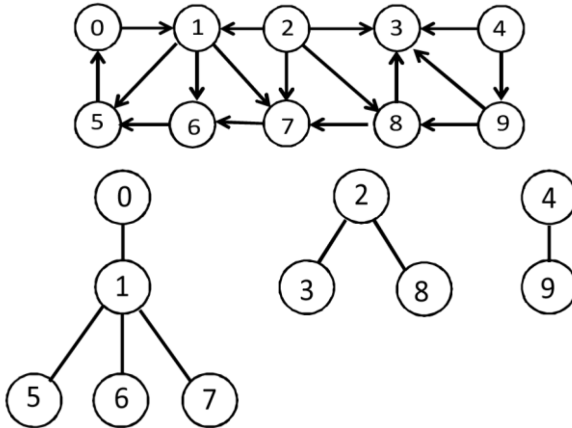


Рис. 26. Орграф и его три компоненты связности – деревья обхода в глубину

Рассмотрим особенности *поиска в глубину* в орграфе. Те дуги, которые ведут к новым вершинам, называются *дугами дерева*, или *древесными дугами*. Они формируют для данного графа либо *остовное дерево* (*дерево поиска в глубину*), либо *остовный лес* (*глубинное остовное дерево*, *глубинный остовный лес*).

Для того чтобы после завершения обхода в глубину все вершины оказались пройденными и образовалось остовное дерево, необходимо и

достаточно, чтобы обход начинался во входе графа (вершине, имеющей только исходящие дуги) и этот вход был единственным. В противном случае при обходе в глубину образуется глубинный остовной лес. Если при поиске в глубину вершины графа нумеруются в порядке их посещения, то такая нумерация называется *глубинной нумерацией графа*, или *M-нумерацией* вершин графа.

После завершения поиска в глубину все ребра связного неориентированного графа образуют одно единственное глубинное дерево. В нем различают только два класса ребер: *древесные и обратные*.

Все дуги орграфа разбиваются на 4 класса.

Класс T (*tree*) древесных дуг, порождающих дерево поиска в глубину, либо глубинный остовной лес. Для каждой дуги $(u,v) \in T$ выполняется условие на нумерацию вершин дерева $M(u) < M(v)$, где $M(u)$ – номер вершины u.

Класс F (*forward*) прямых дуг, к которому относятся дуги (u,v) такие, что $M(u) < M(v)$, и вершина u соединена с v путем, состоящим из древесных дуг. Другими словами, прямые дуги идут от предков к непосредственным потомкам, но не являются древесными дугами.

Класс B (*back*) обратных дуг, представляющих дуги (u,v) такие, что $M(u) < M(v)$, и вершина u соединена с v путем, состоящим из древесных дуг. По-другому: обратные дуги идут от потомков к предкам.

Класс C (*cross*) поперечных дуг, представляющих собой дуги (u,v) такие, что $M(u) > M(v)$, и вершины u и v не соединены путем, состоящим из древесных дуг. Поперечные дуги соединяют вершины, не являющиеся ни потомками, ни предками друг друга.

Обход в ширину

Breadth First Search, *BFS*

Заданы неориентированный или ориентированный граф и начальная вершина *S*. Процесс поиска происходит в «ширину». Сначала обрабатываются все соседние с *S* вершины, затем происходит переход к одной из соседних вершин, которая выбирается по определенному алгоритму. По умолчанию выбирается вершина с меньшим номером. Обрабатываются все соседние с этой вершиной соседние вершины, и т. д. Алгоритм поиска в ширину перечисляет все достижимые из *S* (если идти по ребрам) вершины в порядке возрастания расстояния от *S*. Расстоянием считается длина (число ребер) кратчайшего пути. При поиске из графа выделяется подграф – так называемое дерево поиска в ширину с корнем *S*. Это дерево

содержит только все достижимые из *S* вершины. Для каждой из этих вершин путь из корня (вершины *S*) в дереве поиска будет одним из кратчайших путей (из начальной вершины) в графе.

Используется очередь. Обозначения см. в [26].

```
    int BFS(int **A,int *Vertex,int n,int *Count,
    int &count, int *Connect, int connect, int v)
{   int *Q, head, tail, i, j, k;
    Q=(int*)calloc(n,4);  // queue
    tail=0; head=0;
    Q[tail]=v; tail++;    // =1
    count++; Count[v]=count; // count=0
    Vertex[count]=v;
    Connect[v]=connect;
    while(tail>head)
 {  k=Q[head]; head++;
    for(j=0; j<n; j++)
 {  if(A[k][j]>0 && Count[j]== -1) // all vertex
  { Q[tail]=j; tail++;
    count++; Count[j]=count;
    Vertex[count]=j;
    Connect[j]=connect;
  } // if
 }  // for j
 }  // while
    return count;
}   // BFS

    int WalkBFS(int **A,int n,int v)
{   int i,k,count,connect,*Count,*Vertex,*Connect;
    Count=(int*)calloc(n,4);
    Vertex=(int*)calloc(n,4);Connect=(int*)calloc(n,4);
    connect=0; count= -1;
    for(i=0; i<n; i++) Count[i]= -1;
    // сначала все вершины помечаем как непройденные
    BFS(A,Vertex,n,Count,count,Connect,connect,v);
    while(count<n - 1)
    for(i=0; i<n; i++) //выбор непройденной вершины
 {  if(Count[i]== -1)
  { connect++;
    BFS(A,Vertex,n,Count,count,Connect,connect,i);
```

```
  }
 }  // for i
    // while
    for(i=0; i<n; i++)
    printf("  Count[%2i]=%3i%3i\n",i,Count[i],Connect[i]);
    printf("\n");
    for(k=0; k<n; k++)
 {  i=Vertex[k];
    printf(" Vertex[%2i]=%3i%3i\n",k,i,Connect[i]);
 }  // for k
    return connect;
}   // WalkBFS
```

Сложность поиска в ширину. В процессе поиска каждая вершина попадает в очередь только один раз, следовательно, и извлечь ее можно только однажды. поэтому всего на операции с очередью требуется время $O(n)$. Список смежных вершин просматривается, лишь когда вершина извлекается из очереди, т. е. не более одного раза. Сумма длин этих списков равна m (и $2m$ для неориентированного графа); всего на их обработку потребуется время $O(m)$. Общая сложность поиска составит $O(n+m)$.

Графы правильных многогранников

The graphs of regular polyhedron

Представлены примеры – графы правильных многогранников и их обходы в глубину и ширину. В ребра вписываются номера просматриваемых вершин.

Многогранники: *тетраэдр, октаэдр, куб, икосаэдр, додекаэдр*

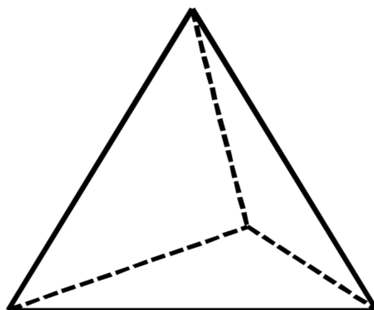


Рис. 27. Тетраэдр: $n=4, m=6$

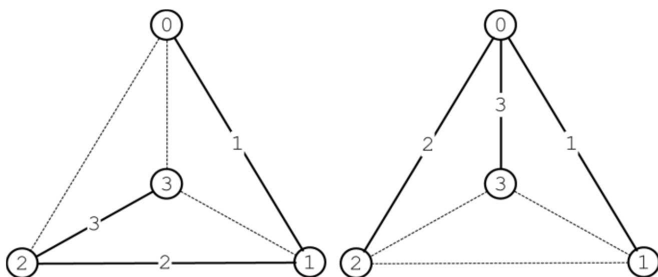
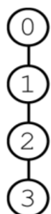


Рис. 28. Обход тетраэдра в глубину и ширину



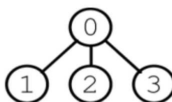


Рис. 29. Дерево обхода тетраэдра в глубину и ширину

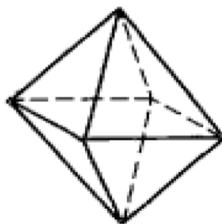


Рис. 30. Октаэдр: $n = 6, m = 12$

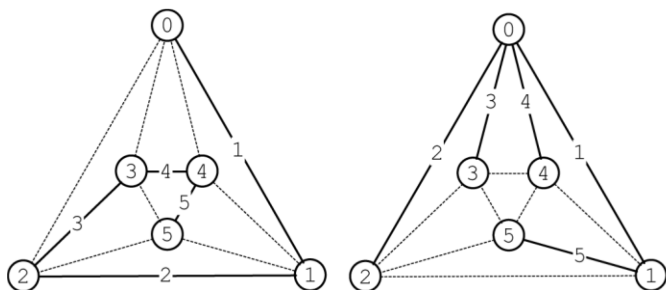


Рис. 31. Обход октаэдра в глубину и ширину

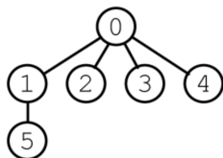


Рис. 32. Дерево обхода октаэдра в глубину и ширину

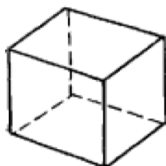


Рис. 33. Куб: $n = 8, m = 8$

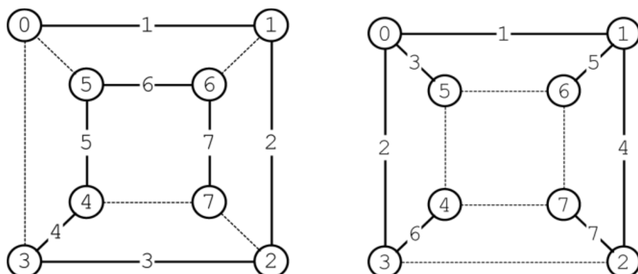


Рис. 34. Обход куба в глубину и ширину

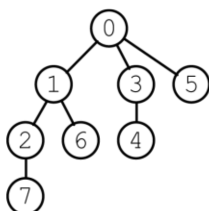


Рис. 35. Дерево обхода куба в глубину и ширину

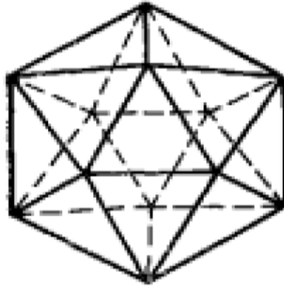


Рис. 36. Икосаэдр: $n = 12, m = 30$

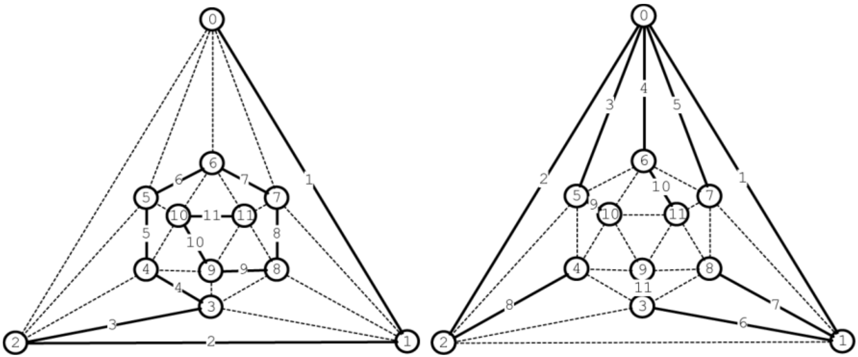


Рис. 37. Обход икосаэдра в глубину и ширину

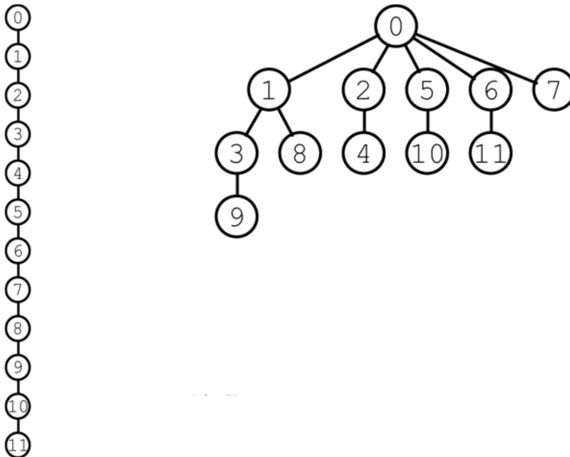


Рис. 38. Дерево обхода икосаэдра в глубину и ширину

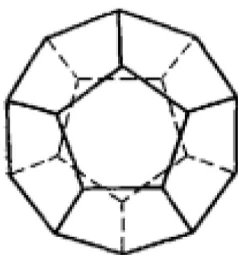


Рис. 39. Додекаэдр

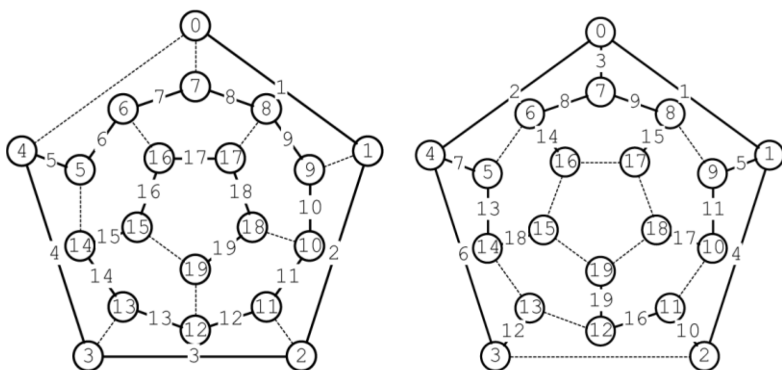


Рис. 40. Обход додекаэдра в глубину и ширину

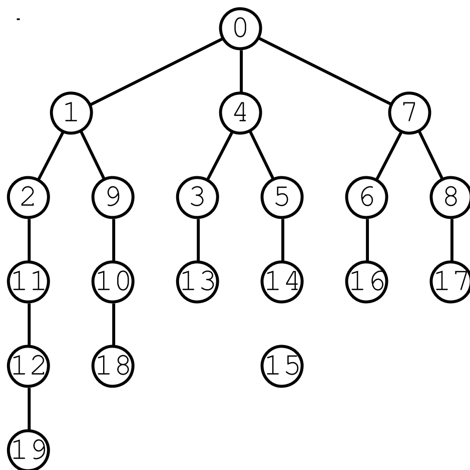


Рис. 41. Дерево обхода додекаэдра в ширину

Граф Петерсена, регулярный граф $n = 10, \deg = 3$

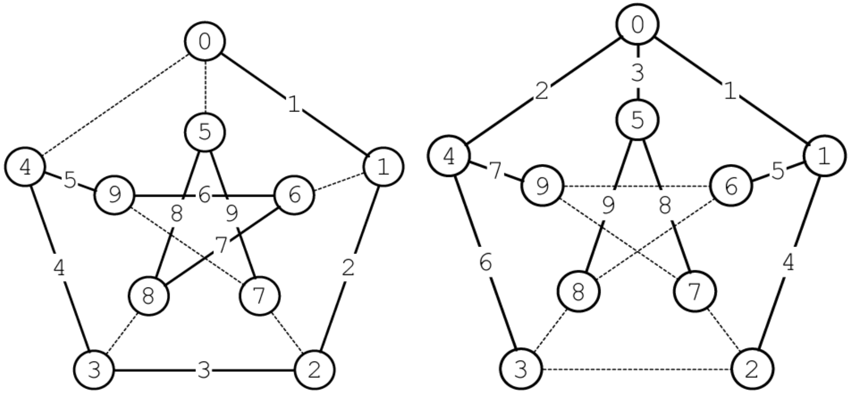


Рис. 42. Обход графа Петерсена в глубину и ширину

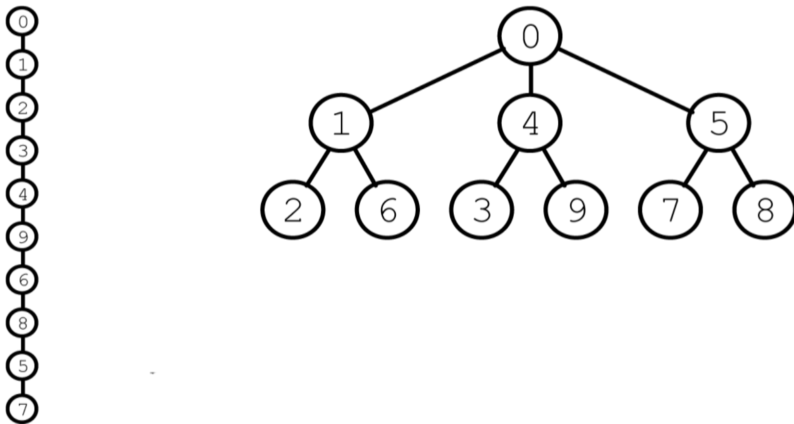


Рис. 43. Дерево обхода графа Петерсена в глубину и ширину

Граф Муна–Мозера

Moon–Moser Graph

Граф Муна–Мозера M_m имеет вершины $\{1,2,\dots,3m\}$. Вершины разбиты на триады $\{1,2,3\}$, $\{4,5,6\}$,..., $\{3m-2,3m-1,3m\}$. Граф M_m не имеет ребер внутри любой триады, но каждая вершина триады связана со всеми вершинами из остальных триад.

Можно обобщить граф Муна–Мозера, полагая произвольную длину d кластера: $\{1,\dots,d\}$, $\{d+1,\dots,2d\}$,..., $\{(m-1)d+1,\dots,md\}$. Число вершин $n=dm$. Число ребер $md(md-1)/2-(d-1)m$. Степень вершин: $d(m-1)$. Обозначение графа естественно принять в форме $M_{d,m}$.

В общем случае d может быть функцией m: $d=d(m)$.

Функция генерации обобщенного графа

```
    int MoonMoserEx(int **&A,int d,int m)
    // graph numbering 1..n
{   int i,j,k,q,p,d,n;
    n=d*m;
    A=(int**)CreateMatrix(n+1,n+1,4);
    for(k=1; k<=m; k++)   // enumeration of k cluster
    for(i=d*(k-1)+1; i<=d*(k-1)+d; i++) // inside cluster
    for(j=1; j<=n; j++)   // enumeration of vertexes
 {  A[i][j]=1;            // default
    for(q=1; q<=d; q++)
  { p=d*(k-1)+q;
    if(j==p) // inside cluster
    {A[i][j]=0; break;}
  } // for q
 }  // for j
 }  // for i
    return n;
}   // MoonMoserEx
```

«Классический» граф $M_{3,5}$, обобщенный граф $M_{5,3}$, их деревья обхода в глубину и ширину показаны на рис. 44–45.

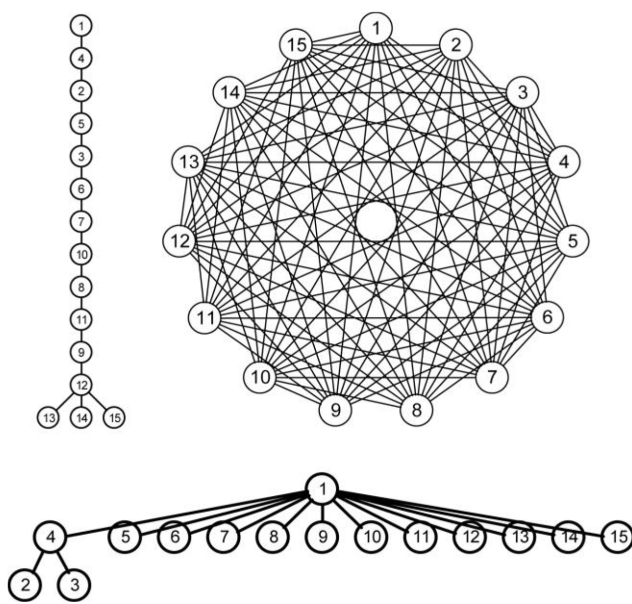


Рис. 44. Граф $M_{3,5}$

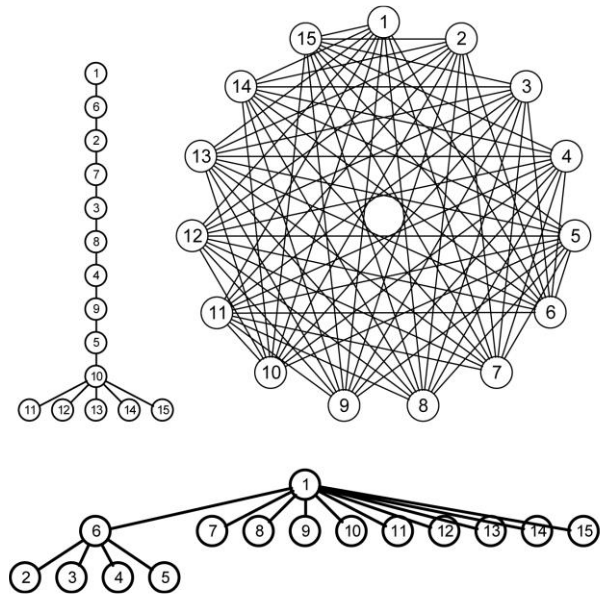


Рис. 45. Граф $M_{5,3}$

Связность графов

Graph Connectivity

Граф называется *связным* (*connected*), если существует путь между любыми двумя его вершинами. Максимальный связный подграф несвязанного графа называется *компонентой связности* данного графа.

В связном графе имеется только одна компонента связности – весь граф. Компоненты связности можно определить также как максимальные по включению связные подграфы данного графа.

Граф на рис. 46 имеет четыре компоненты связности:

$$\{1, 2, 9\}, \{3, 4\}, \{10\}, \{5, 6, 7, 8, 11, 12, 13, 14, 15\}.$$

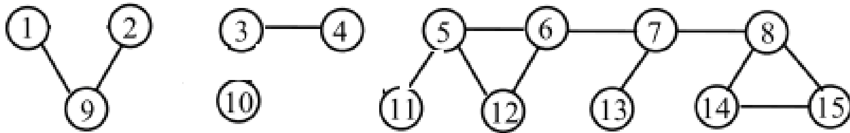


Рис. 46

Вершинная связность $\kappa(G)$

Вершинная связность (*vertex connectivity*) графа $\kappa(G)$ есть наименьшее количество вершин, удаление которых нарушает связность графа.

Точка сочленения есть вершина графа, в результате удаления которой вместе со всеми инцидентными ей рёбрами количество компонент связности в графе возрастает.

Синонимы: *точка сочленения*, *шарнир*, *разделяющая вершина*.

English notation: *articulation point*, *cutpoint*, *cutting vertex*, *cutvertex*.

Граф на рис. 46 имеет четыре шарнира: вершины 6, 7, 8, 9.

Вершина v является шарниром тогда и только тогда, когда в графе есть такие отличные от v вершины u, w, что любой путь, соединяющий u и w, проходит через v.

Блок есть максимальный связный подграф, не имеющий точек сочленения.

Реберная связность $\lambda(G)$

Реберная связность (*edge connectivity*) $\lambda(G)$ есть наименьшее число ребер, удаление которых нарушает связность.

Мост (*bridge*) графа есть ребро, удаление которого приводит к увеличению числа компонент связности.

Граф на рис. 46 имеет мосты (ребра):

(1, 9), (2, 9), (3, 4), (5, 11), (6, 7), (7, 8), (7, 13).

Ребро является мостом в том и только том случае, если в графе нет простого цикла, содержащего это ребро.

Для полных графов $\kappa(G) = n-1$ и $\lambda(G) = n-1$.

Для несвязных графов $\kappa(G) = 0$ и $\lambda(G) = 0$.

Для графа с шарниром $\kappa(G) = 1$.

Для графа с мостом $\lambda(G) = 1$.

Граф *G* называют k-связным, если $\kappa(G) \geq k$, и реберно k-связным, если $\lambda(G) \geq k$.

Связный граф, по крайней мере, 1- и 2-связен, если он не содержит шарниров.

Любой блок (граф без шарниров) 2-связен.

Между введенными характеристиками графа существует зависимость $\kappa \leq \lambda \leq \min(\text{degree})$.

Из определения связности следует несколько свойств 2-связных графов.

1. Степени вершин 2-связного графа больше единицы.
2. Если графы G и H 2-связные и имеют не менее двух общих вершин, то граф $G \cup H$ также 2-связен.
3. Если граф G 2-связен и P – простая цепь, соединяющая две его вершины, то граф $G \cup P$ также 2-связен.
4. Если вершина v не является шарниром связного графа, то любые две его вершины соединены цепью, не содержащей v.
5. В 2-связном графе для любых трех несовпадающих вершин u, v, w имеется (u, v)-цепь, не проходящая через вершину w.

Эквивалентные свойства 2-связных графов с $n \geq 3$.

1. Граф 2-связен.
2. Любые две вершины графа принадлежат простому циклу.
3. Любая вершина и любое ребро принадлежат простому циклу.
4. Любые два ребра принадлежат простому циклу.

5. Для любых двух вершин u, v и любого ребра e существует простая (u, v)-цепь, содержащая e.
6. Для любых трех вершин u, v, w существует простая (u, v)-цепь, проходящая через w.

Пример. На рис. 47 приведен пример графа и его блоков. Следует подчеркнуть, что шарнир входит во все блоки, с которыми они связаны.

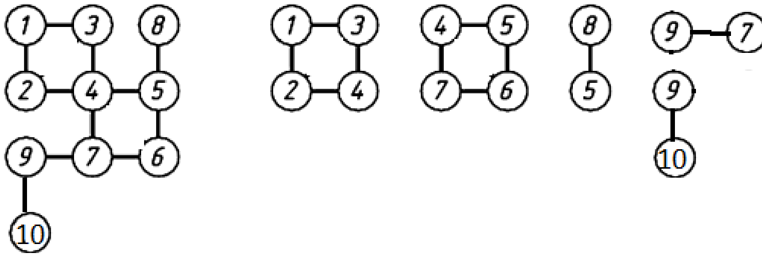


Рис. 47. Граф и его блоки. Шарниры – вершины с номерами 4, 5, 7, 9

Алгоритм нахождения блоков и шарниров графа

Вершина q является шарниром, если существуют другие вершины u, v такие, что существующий любой путь из $u \to v$ проходит через вершину q.

На рис. 48 схематично изображен граф с «квадратными» блоками B_k, $k = 1, \ldots, 5$, и «круглыми» шарнирами – u, v, x, y.

Выполняется *поиск в глубину* из произвольной вершины, принадлежащей B_1. Из блока B_1 в процессе поиска переходим в B_2, проходя через вершину u. По свойству поиска в глубину все ребра B_2 должны быть пройдены до того, как осуществляется возврат в вершину u. Поэтому блок B_2 состоит только из ребер, которые проходятся между первыми двумя заходами в вершину u. Затем из блока B_2 переходим в блоки B_3, B_4, B_5.

Последовательность прохождения ребер хранится в стеке. При возвращении ребра, находящиеся на вершине стека в момент обратного прохода через шарниры, определяют блок. Тем самым вычисляется в обратном порядке последовательность блоков.

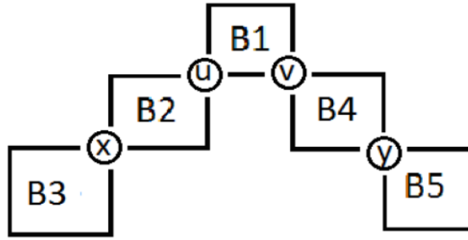


Рис. 48. Схематичное изображение графа, его блоков B_k и шарниров x, y, u, v

Функция поиска в глубину рекурсивная. Достижимые вершины записываются последовательно в массив `Count`. Для поиска шарниров и блоков необходимо учитывать информацию обратного хода, которая обычно не нужна. Вводится дополнительный массив `Block`, элементы которого первоначально определяются при входе в функцию. Запись в `Block` производится таким образом, чтобы вершины блока были помечены одним числом, равным первой записанной вершине блока. Значения элементов `Block` переопределяются исходя из двух условий.

Во-первых, `Block` обрабатывается при выходе из рекурсивной функции. После выхода из рекурсивной функции сравниваются значения `Block` текущей `j` и следующей вершины i и переопределяется `Block[j]` при условии:

```
if(Block[j]>Block[i]) Block[j]=Block[i];
```

Во-вторых, изменение `Block[j]` также происходит при условии:

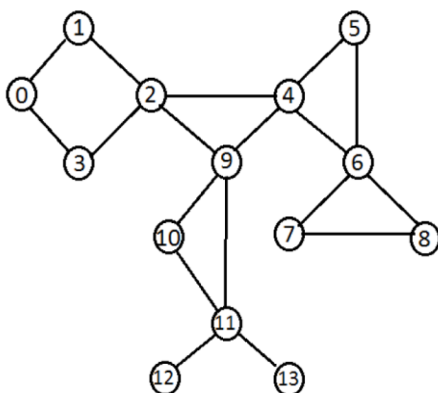
```
if(Count[i]!=-1 && j!=k && Count[j]>Count[i])
if(Count[i]<Block[j]) Block[j]=Count[i];
```

Блок заканчивается при условии `Block[i]>=Count[j]`. Выход из функции происходит при условии достижения на обратном проходе начальной вершины (здесь, в частности, значение 0). И происходит печать шарниров.

```
    int DFSblock(int **A,int n,int &count,int j) //
{   int i,k; // u=previous vertex, j=current vertex
    k=Vertex[count];
    Count[j]=++count; // вершина i помечается
    Block[j]=count;
    Vertex[count]=j;
    for(i=0; i<n; i++) //просмотр матрицы смежности
 {  if(A[j][i])
    if(Count[i]==-1)
```

```
    // выбор из матрицы смежности еще непройденной вершины
   { DFSblock(A,n,count,i);
    // back
    if(Block[j]>Block[i]) Block[j]=Block[i];
    if(Block[i]>=Count[j]) // out block
 {  block++;// 1...
    Point[block]=j; // art point
{   if(j==0) // begin v0
   { printf("\n ArtPoint\n");
    for(k=0; k<block; k++)
    printf("%2i %2i\n",k,Point[k]);
   } // if(j==0)
 } }   // if Count[j]==-1
    else // if(Count[j]>-1)
    if(i!=k && Count[j]>Count[i])
    if(Count[i]<Block[j])
    Block[j]=Count[i];
   } // j exit if j=n
    return i;
}   // DFSblock

    int TestDFSBlock()
{   int i,j,k,n,m,count,connect,**A;
    n=14;
    A=InEdge2(1,n,m,1,2,1,4,
    2,3,
    3,4,3,5,3,10,
    5,6,5,7,5,10,
    6,7,
    7,8,7,9,
    8,9,
    10,11,10,12,
    11,12,
    12,13,12,14,
    0,0);
    connect=0; count=-1;block=0;
    for(i=0; i<n; i++)
 {  Count[i]=-1;Vertex[i]=-1; }
// сначала все вершины графа помечаем как непройденные
    DFSblock(A,n,count,connect,0,-1);
    return m;
}   // TestDFSBlock
```



`Art Point: 6,4,11,9,2`

Рис. 49. Граф и его шарниры (*art point*)

Маршруты и связность в орграфах

Для ориентированного графа можно определить два типа маршрутов. *Неориентированный маршрут* (или просто *маршрут*) графа G есть последовательность вершин $v_1, v_2, ..., v_n$ такая, что для каждого $i = 1, 2, ..., n-1$ либо $(v_i, v_{i+1}) \in G$, либо $(v_{i+1}, v_i) \in G$.

Маршрут называется *ориентированным* (или *ормаршрутом*), если все $(v_i, v_{i+1}) \in G$. При движении вдоль маршрута в орграфе ребра могут проходиться как в направлении ориентации, так и в обратном направлении, а при движении вдоль ормаршрута – только в направлении ориентации. Это различие очевидным образом распространяется на пути и циклы, так что в орграфе можно рассматривать пути и орпути, циклы и орциклы.

Говорят, что маршрут $v_1, v_2, ..., v_n$ соединяет вершины v_1 и v_n, а ормаршрут $v_1, v_2, ..., v_n$ ведет из v_1 в v_n.

Соответственно двум типам маршрутов определяются и два типа связности орграфов.

Орграф называется *связным* (или *слабо связным*, *weakly connected graph*), если для каждой пары вершин в нем имеется соединяющий их маршрут. В другой формулировке. Орграф называется *связным*, или *слабо* связным, если его соотнесенный (подобный, но неориентированный) граф является связным.

Орграф называется *сильно связным* (*strongly connected*), если для каждой упорядоченной пары вершин (u, v) в нем имеется ормаршрут,

ведущий из u в v. В другой формулировке. Две вершины в ориентированном графе сильно связаны, если существует ориентированный путь из вершины u в v и из v в u. Орграф называется *сильно связным*, если все вершины сильно связаны

Максимальные по включению подмножества вершин орграфа, порождающие сильно связные подграфы, называются его *областями сильной связности*, а порождаемые ими подграфы – *компонентами сильной связности*.

Очевидно, разные области сильной связности не могут иметь общих вершин, так что множество вершин каждого орграфа разбивается на области сильной связности.

Областями сильной связности орграфа на рис. 50 являются множества {1, 2, 5}, {3, 4, 6, 7, 8}, {9}.

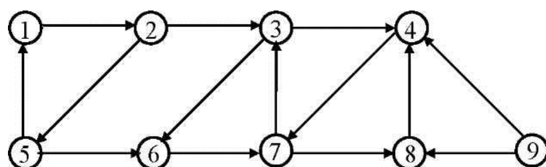


Рис. 50

Алгоритм нахождения сильно связанных компонент графа. Для каждой вершины вычисляется множество вершин, достижимых из нее, и множество вершин, из которых достигается эта вершина. Пересечение этих множеств дает множество вершин графа, образующих сильно связанную компоненту, которой принадлежит вершина

Граф конденсации H

Каждая его вершина представляет некоторую сильно связанную компоненту графа. Дуга $(\overline{u},\overline{v})$ существует в H тогда и только тогда, когда в *G* существует дуга (u,v) такая, что *u* принадлежит компоненте, соответствующей вершине $\overline{u}$, а $\overline{v}$ – компоненте, соответствующей вершине v.

В ориентированном графе каждая вершина *u* может принадлежать только одной сильно связанной компоненте.

В графе есть множество вершин *P*, из которых достижима любая вершина графа, являющееся минимальным в том смысле, что не существует подмножества в *P*, обладающего таким свойством достижимости. Это множество вершин называется *базой* графа. В *P* нет двух вершин, которые принадлежат одной и той же сильно связанной компоненте графа. Любые две базы графа *G* имеют одно и то же число вершин.

Кратчайшие пути в графе

Shortest Path. Single-source shortest paths problem

Задается ориентированный взвешенный граф $G = (V, E)$ с весами ребер.

Весом пути $p = (v_0, v_1, ..., v_n)$ называется сумма весов ребер, входящих в этот путь: $w(p) = \sum_{i=1}^{n} w(v_{i-1}, v_i)$. Вес кратчайшего пути из u в v равен $\delta(u,v) = \min\{w(p) : u \to v\}$, если существует путь $u \to v$, иначе $\delta(u,v) = \infty$.

Кратчайший путь из *u* в v – это любой путь p из u в v, для которого $w(p) = \delta(u,v)$. Если $p = (v_1, v_2, ..., v_k)$ есть кратчайший путь $v_1 \to v_k$ и $1 \le i \le j \le k$, то $p_{ij} = (v_i, v_{i+1}, ..., v_j)$ есть кратчайший путь $v_i \to v_j$.

Весами ребер могут быть расстояния, времена, силы взаимодействия, стоимости, штрафы, т. е. любая величина, которая должна быть минимизирована и которая обладает свойством аддитивности.

Алгоритм поиска в ширину можно рассматривать как решение задачи о кратчайшем пути в частном случае, когда вес каждого ребра равен единице (т. е. граф невзвешенный). Дерево кратчайших путей аналогично дереву поиска в ширину с той лишь разницей, что кратчайшими объявляются пути с наименьшим весом, а не наименьшим числом ребер. Ни кратчайшие пути, ни деревья кратчайших путей не обязательно являются единственными для графа.

Кратчайшие пути и релаксация. Релаксация является общим приемом в алгоритмах поиска кратчайших путей. Он состоит в постепенном уточнении верхней оценки на вес кратчайшего пути в данную вершину – пока неравенство не превратится в равенство. Для каждого ребра $v \in V$ сохраняется некоторое число, являющееся верхней оценкой веса кратчайшего пути из начальной вершины s в вершину t. Для краткости оно называется просто оценкой кратчайшего пути. Начальное значение оценки кратчайшего пути для вершины s $d[s] = 0$, для всех остальных вершин – $d[v] = \infty$.

Релаксация ребра $(u,v) \in E$ состоит в следующем. Значение $d[v]$ уменьшается до величины $\overline{d}[v] = d[u] + w(u,v)$, если $\overline{d}[v] < d[v]$. При этом $d[v]$ остается верхней оценкой пути из *s* в *v*, так как для всякого ребра $(u,v) \in E$ справедливо $\delta(s,v) \le \delta(s,u) + w(u,v)$.

Алгоритм Дейкстры

Dijkstra algorithm, 1959

Алгоритм Дейкстры вычисляет кратчайшие пути из одной исходной вершины s (*source*) во все другие вершины для взвешенного ориентированного графа $G(V, E)$, в котором веса всех ребер неотрицательны: $w(u, v) \geq 0$.

В процессе работы алгоритма формируется множество $S \subseteq V$, состоящее из вершин v, для которых расстояние $\delta(s, v)$ уже найдено, т. е. $d[v] = \delta(s, v)$. Осуществляя проход графа в ширину, начиная с начальной вершины s, алгоритм выбирает очередную вершину $u \in V \setminus S$ с наименьшим $d[u]$, добавляет u к множеству S и производит релаксацию всех ребер, выходящих из u, после чего процесс повторяется. Вершины, для которых $\delta(s, v)$ еще не найдено, т. е. не пройденные вершины, хранятся в очереди Q, определяемыми значениями оценок кратчайшего пути d. Так как кроме весов кратчайших путей, как правило, требуется найти и сам путь, как и при поиске в ширину, для каждой вершины $v \in V$ запоминается ее предшественник $P[v]$. Предшественник вершины – это либо вершина, либо –1, если предшественника нет.

Алгоритм Дейкстры

1. Выбирается начальная вершина v_0, от которой отыскиваются кратчайшие пути до всех остальных вершин. Вершина v_0 включается в массив выбранных вершин.
2. Определяем вершину v_1 с минимальным расстоянием от v_0. Включаем вершину v_1 в массив выбранных и в массив кратчайших путей**.**
3. Определяем расстояния от вершины v_0 до всех смежных и доступных из v_1 невыбранных вершин. Если расстояние до некоторой вершины меньше ранее определенного, то заменяем его новым значением.
4. Этот процесс выполняется за $(n-1)$ итераций, пока не будут обработаны все доступные вершины.

Восстановление путей. Чтобы получить не только длины кратчайших путей, но и сами пути необходимо записывать информацию, доста-

точную для последующего восстановления кратчайшего пути из s до любой вершины. Для этого нет необходимости вводить матрицу кратчайших путей. Достаточно одномерного массива p, в котором для каждой вершины $v \neq s$ хранится номер вершины $p[v]$, являющейся предпоследней в кратчайшем пути до вершины v. Этого вполне достаточно, потому что если в кратчайшем пути до некой вершины v удалить из этого пути последнюю вершину v, то получится путь, оканчивающийся некоторой вершиной u, и этот путь будет кратчайшим для вершины $p[u]$. Кратчайшие пути восстанавливаются по p, в котором берется предшествующая вершина от текущей вершины. Процесс заканчивается в начальной вершине, в результате чего получается искомый кратчайший путь, но записанный в обратном порядке. Массив p строится при каждой успешной релаксации, т. е. когда из выбранной вершины v происходит улучшение расстояния до некоторой вершины u, тогда записывается что для вершины u предшествующей является вершина v: $p[u] = v$.

Основное положение алгоритма Дейкстры следующее. *Утверждается, что после того как какая-либо вершина v становится помеченной, текущее расстояние до неё $d[v]$ уже является кратчайшим и, соответственно, больше меняться не будет.*

Доказательство проводится по индукции. Для первой итерации справедливость его очевидна $d[s] = 0$, что и является длиной кратчайшего пути до неё. Пусть теперь это утверждение выполнено для всех предыдущих итераций, т. е. всех уже помеченных вершин. Далее следует показать, что оно не нарушается после выполнения текущей итерации. Пусть v – вершина, выбранная на текущей итерации, т. е. вершина, которую алгоритм собирается пометить. Докажем, что $d[v]$ действительно равно длине кратчайшего пути до неё.

Рассмотрим кратчайший путь от вершины s до v длиной $h[v]$. Этот путь можно разбить на две части: a и b. a состоит только из помеченных вершин, включая стартовую. b может включать помеченные и непомеченные вершины, но начинается обязательно с непомеченной. Обозначим последнюю вершины пути a – α, а первую вершину пути b – β.

Докажем сначала утверждение для вершины β, т. е. докажем равенство $d[\beta] = h[\beta]$. Это очевидно, поскольку на одной из предыдущих итераций была выбрана вершина α и выполнялась релаксация из неё. Поскольку (в силу самого выбора вершины β) кратчайший путь до β равен кратчайшему пути до α плюс ребро (α, β), то при выполнении релакса-

ции из α величина $d[\beta]$ действительно установится в требуемое значение.

Вследствие *неотрицательности* весов рёбер длина кратчайшего пути $h[\beta]$ (а она, согласно только что доказанному, равна $d[\beta]$) не превосходит длины $h[\beta]$ кратчайшего пути до вершины v. Учитывая, что $d[\beta] \geq h[\beta]$ (ведь алгоритм Дейкстры не мог найти более короткого пути, чем это вообще возможно), в итоге получаем соотношения: $d[\beta] = h[\beta] \leq h[v] \leq d[v]$.

С другой стороны, поскольку β, v – вершины непомеченные, и так как на текущей итерации была выбрана именно вершина v, а не вершина β, справедливо другое неравенство: $d[\beta] \geq d[v]$

Из этих двух неравенств вытекает равенство $d[\beta] \geq d[v]$, а тогда из найденных до этого соотношений следует искомое $d[v] = h[v]$.

Обозначения в функции `Dijkstra`.

Аргументы:

```
n – порядок графа
A[n][n] – матрица смежности (весов, расстояний)
v0 – исходная вершина
d[n] – массив расстояний от исходной вершины v0
p[n] – предшествующая вершина в кратчайшем пути
```

Локальные переменные:

```
imin – индекс вершины с минимальным расстоянием от v0
dmin – минимальное расстояние
Mark[n] отмечает вершину, для которой уже найдено
минимальное расстояние от v0
```

```
    int Dijkstra(int **A, int *d, int *p, int n, int v0)
{   int i, j, k, h, imin, dmin,
    inf=0x7FFFFFFF,
    *Mark=(int *)calloc(n,4); // at first all Mark[i]=0
    //initialization
    for(i=0; i<n; i++)
 {  if(!A[v0][i])
  { d[i]=inf; p[i]=i;} else
  { d[i]=A[v0][i]; p[i]=v0; }
 }  // i
    d[v0]=0; Mark[v0]=1; // for v0
    // main loops
    for(k=1; k<n; k++)
```

```
 {  // выбираем вершину с индексом imin
    // с наименьшим расстоянием d[i]=dmin
    dmin=inf; imin=-1;
    for(i=0; i<n; i++)
    if(!Mark[i] && d[i]<dmin)
    {dmin=d[i]; imin=i;}
    Mark[imin]=1;
// для вершины imin уже найдено минимальное расстояние от
// v0, она отмечается mark=1 и затем исключается из
// дальнейшей обработки
    for(j=0; j<n; j++)
  { if(Mark[j] || !A[j][i]==0 || j==v0) continue;
// для всех неисключенных (mark=0) вершин с индексом j,
// смежных с найденной вершиной imin, переопределяются
// текущие расстояния d[j] исходя из условия
// d[imin]+A[imin][j]<d[j]
// также переопределяется предыдущая вершина
// кратчайшего пути
  { h=d[imin]+A[imin][j];
    if(h<d[j]) {d[j]=h; p[j]=imin;}
  } // j
 }  // k
    return n;
}   // Dijkstra
```

Вызов функции и печать путей: длин и последовательности вершин

```
    Dijkstra(A,d,p,n,v0);
    for(i=0; i<n; i++)
  { printf("%3i %3i",i,d[i]); printf("\n"); }
    printf("  i  p[i]\n");
    for(i=0; i<n ; i++)
  { if(i==v0) continue;
    j=i;
    while(j!=v0)
    {j=p[j]; printf("%3i %3i",i,j); }
     printf("\n");
  } // for i
```

После завершения алгоритма для всех вершин $u \in V$ будут выполняться равенства $d[u] = \delta(s,u)$.

Сложность алгоритма Дейкстры: $O(n^2 + m)$.

Пример. Для ориентированного взвешенного графа (рис. 52) показано выполнение алгоритма Дейкстры. В таблице указаны вычисленные длины кратчайших путей.

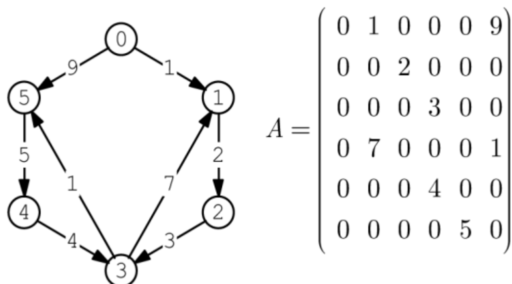


Рис. 52

i	d[i]	p[i]
1	1	0
2	3	1, 0
3	6	2, 1, 0
4	12	5, 3, 2, 1, 0
5	7	3, 2, 1, 0

Алгоритм Беллмана–Форда

Bellman–Ford algorithm

Рассматривается неориентированный или ориентированный граф с весами ребер, которые могут быть отрицательными (в отличие от алгоритма Дейкстры). Находится кратчайший путь из заданной вершины до всех остальных. Ограничение – отсутствие контуров отрицательной длины, достижимых из начальной вершины. Если граф содержит цикл отрицательной длины, то расстояния между некоторыми парами вершин становится неопределенными, поскольку, обходя этот контур достаточное число раз, можно построить путь между этими вершинами с длиной, меньшей любого числа.

Излагаемый метод часто называют динамическим программированием, а алгоритм вычисления расстояний – алгоритмом Форда–Беллмана (в англоязычной литературе Беллмана–Форда). Появился этот алгоритм в работе Форда 1956 года и в работе Беллмана 1958 года.

Основная идея алгоритма заключается в поэтапном вычислении кратчайших расстояний. Обозначим через $d_k[i]$ длину кратчайшего среди всех (s,i)-путей, содержащих не более чем k дуг. Справедливы очевидные неравенства $d_0[i] \geq d_1[i] \geq ... \geq d_{n-1}[i]$. Поскольку по предположению в графе нет контуров отрицательной длины, кратчайший путь не может содержать более чем $n-1$ дуг. Поэтому величина $d_{n-1}[i]$ дает искомое расстояние от s до i. Для вычисления $d_{n-1}[i]$ достаточно последовательно вычислять $d_k[i]$ для всех $k = 1,...,n-1$. Начальные значения $d_1[i]$ вычисляются как $d_1[i] = w(s,i)$ для всех $i \neq v$.

Затем последовательно вычисляются по рекуррентной формуле значения $d_k[i] = \min\{d_{k-1}[i], d_{k-1}[j] + w[j,i]\}$.

В программе эти вычисления проводятся с помощью одного одномерного массива d длины n.

Обозначения аналогичны функции Dijkstra.

```
    int Ford(int **A,int *d,int *p,int n,int v0)
{   int i,j,k,h,inf=0x7FFFFFFF;
    //  initialization
    for(i=0; i<n; i++)
 {  if(A[v0][i]==0)
  { d[i]=inf; p[i]=i;} else
  { d[i]=A[v0][i]; p[i]=v0; }
 }  // i
    // main loops
    for(k=1; k<n; k++)  // all steps
 {  for(i=0; i<n; i++)  // recomputed d[i]! but d[j]
  { if(i==v0) continue;
    for(j=0; j<n; j++)  // смежные с i вершиной
  { if(!A[j][i]==0 || j==v0) continue; // несмежные или v0
    h=d[j]+A[j][i];
    if(h<d[i])
    {d[i]=h; p[i]=j; }
  } // j
  } // i
 }  // k
    return 1;
}   // Ford
```

Сложность алгоритма Беллмана–Форда в силу наличия трех вложенных циклов составляет $O(n^3)$. Однако на самом деле из-за наличия условия перехода на конец цикла по j, скорость может быть значительно больше.

Алгоритм решения задачи о кратчайшем пути из начальной вершины в ориентированном взвешенном графе можно применять и для других задач.

Нахождение кратчайших путей в одну вершину: дана конечная вершина t, требуется найти кратчайшие пути в t из всех вершин $v \in V$. Если изменить направление всех дуг на противоположное, то эта задача сведется к задаче о кратчайших путях из одной вершины.

Нахождение кратчайшего пути между данной парой вершин: даны вершины u и v, найти кратчайший путь из u в v. Если найдены все кратчайшие пути из u, то, естественно, найден и кратчайший путь из u в v. Следует подчеркнуть – более быстрого способа решения этой задачи пока не найдено.

Нахождение кратчайших путей для всех пар вершин: для каждой пары вершин u и v найти кратчайший путь из u в v. Эту задачу можно решить, находя кратчайшие пути из данной вершины для всех вершин по очереди. Для этой задачи алгоритм Флойда–Уоршелла более оптимален.

Кратчайшие пути для всех пар вершин – алгоритм Флойда–Уоршелла

Floyd–Warshall algorithm

Дан *ориентированный взвешенный* граф $G = (V, E)$ с весами ребер. Для каждой пары вершин $u, v \in V$ требуется найти кратчайший путь из u в v, т. е. путь наименьшей длины. Длина пути определяется как сумма весов всех его ребер.

Алгоритм Флойда–Уоршелла использует представление графа матрицей весов (смежности) $W = (w_{ij})$:

$$w_{ij} = \begin{cases} 0, \text{ if } i = j \\ weight\ of\ edge, \text{if } i \neq j \ \ \& \ \ (i,j) \in E \\ \infty, \text{ if } i \neq j \ \ \& \ \ (i,j) \notin E \end{cases}$$

Веса могут быть отрицательны, но не может быть отрицательных циклов. Граф может быть мультиграфом (с несколькими ребрами). Если в смешанном графе есть неориентированные ребра, они могут заменяться направленными противоположными ребрами (дугами) с одинаковым ве-

сом. Такая замена невозможна для отрицательных ребер, поскольку сразу получается отрицательный контур.

Алгоритм Флойда–Уоршелла использует понятие характеристики пути – максимального номера промежуточной вершины в нем. Для данной пары вершин $i, j \in V$ рассматриваются все пути из i в j, у которых все промежуточные вершины принадлежат множеству $\{0,1,2,\ldots,k\}$, где произвольное $k \leq n$. Пусть P – путь минимального веса среди всех таких путей. Находится вес этого пути, зная веса всех таких путей (для всех пар вершин) для значений, меньших k. Для пути P есть две возможности.

Если вершина k не является промежуточной в p, то все промежуточные вершины пути P содержатся в множестве $\{0,1,2,\ldots,k-1\}$. Тогда P – кратчайший путь из i в j, промежуточные вершины которого принадлежат множеству $\{0,1,2,\ldots,k-1\}$.

Если вершина k является промежуточной вершиной пути P, она разбивает его на два участка P_1 и P_2; вершина k встречается лишь однажды, так как P – простой путь. Путь P_1 – кратчайший из i в k с промежуточными вершинами из множества $\{0,1,\ldots,k-1\}$, а путь P_2 является кратчайшим путем из k в j с промежуточными вершинами из множества $\{0,1,2,\ldots,k-1\}$.

Этот анализ позволяет написать рекуррентную формулу для длин кратчайших путей. Обозначим через d_{ij} вес кратчайшего пути из вершины i в вершину j с промежуточными вершинами из множества $\{0,1,2,\ldots,k\}$. При $k = 0$ промежуточных вершин нет, поэтому $d_{ij} = w_{ij}$. В общем случае

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, \text{ if } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right), \text{ if } k \geq 1. \end{cases}$$

Матрица $D^{(n)} = (d_{ij}^{(n)})$ содержит искомое решение. Другими словами, $d_{ij}^{(n)} = \delta(i, j)$ для всех $i, j \in V$, поскольку разрешены любые промежуточные вершины. Алгоритм Флойда–Уоршелла вычисляет веса кратчайших путей, последовательно находя значения $d_{ij}^{(k)}$ для $\{0,1,2,\ldots,n\}$, т. е. снизу вверх. Его входом является матрица W размером $n \times n$ (веса ребер гра-

фа). Последовательно вычисляются значения матриц $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$.

Результат – матрица $D^{(n)}$ весов кратчайших путей.

Время работы алгоритма $O(n^3)$ определяется тремя вложенными циклами. Константа, скрытая в *O*-обозначении, невелика, поскольку алгоритм прост и не использует сложных структур данных.

Помимо весов кратчайших путей, естественно находятся и сами пути. Для этого параллельно с вычислением матрицы D вычисляется матрица предшествования P. При этом вычисляется последовательность матриц $P^{(0)}, P^{(1)}, \ldots, P^{(n)}$. $P^{(n)}$ и есть матрица предшествования. Ее элемент $P_{ij}^{(k)}$ определяется как вершина, предшествующая вершине j на кратчайшем пути из вершины i в вершину j с промежуточными вершинами из множества $\{0,1,2,\ldots,k\}$. При $k=0$ промежуточных вершин нет, поэтому

$$P_{ij}^{(0)} = \begin{cases} 0, & \text{if } \; i = j \; \text{ or } \; w_{ij} = \infty; \\ i, & \text{if } \; i \neq j \; \& \; w_{ij} < \infty. \end{cases}$$

Пусть $k \geq 1$. Если кратчайший путь из i в j проходит через вершину ***k***, то предпоследней его вершиной будет та же самая вершина, которая будет предпоследней на кратчайшем пути из k в j с промежуточными вершинами из множества $\{0,1,2,\ldots,k-1\}$. Если путь не проходит через *k*, то он совпадает с кратчайшим путем из *i* в *j* с промежуточными вершинами из множества $\{0,1,2,\ldots,k-1\}$. Таким образом

$$P_{ij}^{(k)} = \begin{cases} P_{ij}^{(k-1)}, \text{if } \; d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}; \\ P_{kj}^{(k-1)}, \text{if } \; d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Вычисления $D_{ij}^{(k)}$ и $P_{ij}^{(k)}$ проводятся одновременно.

Матрица смежности переопределяется:

`D[i][j]=A[i][j]`, если существует дуга (i,j);

`D[i][j]=inf`, если не существует дуги (i,j);

D[i][i]=0;

Матрица путей P[i,j]=i:

```
    int Floyd(int **A, int **&D, int **&P, int n)
{   int i,j,k,h, inf=0x7FFFFFF;
    D=(int**)CreateMatrix(n,n,4);
```

```
    P=(int**)CreateMatrix(n,n,4);
    // initial
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
 {  if(i==j) D[i][j]=0;
    else if(A[i][j]<=0) D[i][j]=inf;
    else D[i][j]=A[i][j];
    if(i==j || D[i][j]==inf)
    P[i][j]=0; else P[i][j]=i;
 }
    // main loops
    for(k=0; k<n; k++)
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
 {  if(D[i][k]!=inf && D[k][j] !=inf)
 {  h=D[i][k]+D[k][j];
    if(h<D[i][j])
    {D[i][j]=h; P[i][j]=P[k][j]; }
    //previous vertex from i to j
 } // k i j
    return n;
}   // Floyd
```

Рекурсивная функция печати кратчайшего пути между вершинами `(i,j)`

```
    int PathRec(int **P,int i,int j)
{   if(P[i][j]==i)
 {  if(i==j) printf("i=%i\n",i);
    else printf("%i – %i; ",i,j);
 }
    else
 {  Path(P,i,P[i][j]);
    Path(P,P[i][j],j);
 }
    return i;
}   // PathRec
```

Итеративная функция печати кратчайшего пути между вершинами (i, j)

```
    int PathIter(int **P,int i,int j)
{   printf("P=\n");
    j=P[i][j];
    while(j!=i)
    {printf("%3i",j); j=P[i][j]; }
```

```
    printf("\n");
    return i;
}   // PathIter
```

Функция печати всех кратчайших путей

```
    int FloydPrint(int **D, int **P,int n)
{   int i,j,k;
    printf("\n");
    PrintMatrix("\n  D Floyd\n","%3i",D,n);
    PrintMatrix("\n  P Floyd\n","%3i",P,n);
    printf("\n\n");
    printf("  i   j   Dij  Pij\n\n");
    for(i=0; i<n; i++)
 {  for(j=0; j<n; j++)
  { if(i==j) continue;
    printf("%3i %3i %3i",i,j, D[i][j]);
    k=P[i][j];
    while(k!=i)
    {printf("%3i",k); k=P[i][k]; }
    printf("\n");
  } // j
    printf("\n");
 }  // i
    return n;
}   // FloydPrint
```

Для графа, представленного на рис. 53, вычислены алгоритмом Флойда–Уоршелла матрицы D и P.

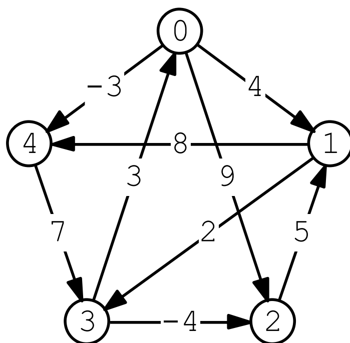


Рис. 53

$$A = \begin{pmatrix} 0 & 4 & 9 & 0 & -3 \\ 0 & 0 & 0 & 2 & 8 \\ 0 & 5 & 0 & 0 & 0 \\ 3 & 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 \end{pmatrix} \quad D^{(0)} = \begin{pmatrix} 0 & 4 & 9 & \infty & -3 \\ \infty & 0 & \infty & 2 & 8 \\ \infty & 5 & 0 & \infty & \infty \\ 3 & \infty & -4 & 0 & \infty \\ \infty & \infty & \infty & 7 & 0 \end{pmatrix} \quad D^{(5)} = \begin{pmatrix} 0 & 4 & 0 & 4 & -3 \\ 5 & 0 & -2 & 2 & 2 \\ 10 & 5 & 0 & 7 & 7 \\ 3 & 1 & -4 & 0 & 0 \\ 10 & 8 & 3 & 7 & 0 \end{pmatrix}$$

$$P^{(0)} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 2 & 0 & 0 & 0 \\ 3 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \end{pmatrix}.$$

Вершины, через которые проходят кратчайшие пути из j в i, и величина этих путей D_{ij}.

D_{ij}	j ← i
4	1 ← 0
0	2 ← 3 ← 4 ← 0
4	3 ← 4 ← 0
–3	4 ← 0
5	0 ← 3 ← 1
–2	2 ← 3 ← 1
2	3 ← 1
2	4 ← 0 ← 3 ← 1
10	0 ← 3 ← 1 ← 2

И т. д.

Интересное расширение приведенных классических задач нахождения кратчайших путей:

- веса имеют не только ребра, но и вершины;
- найти несколько неравных или равных кратчайших путей.

Транзитивное замыкание

Transitive closure

Транзитивное замыкание заданного орграфа есть вводимый орграф с теми же вершинами, в котором ребро (u,v) между вершинами u и v существует только тогда, когда в заданном орграфе имеется ориентированный путь из u в v.

В транзитивном замыкании существует ребро, исходящее из любой вершины в каждую вершину, достижимую из этой вершины в заданном орграфе. Транзитивное замыкание содержит в себе полную информацию, необходимую для решения задачи достижимости. Поэтому матрицу смежности транзитивного замыкания называют *матрицей достижимости*, или *путевой матрицей* (*path*, *reachability matrix*). Транзитивное замыкание лишь только показывает, имеется или отсутствует путь между парой вершин (или цикл в любой вершине), но не определяет все пути.

Рассмотрим матрицу смежности A, в которой элемент $A_{ij} = 1$ в том случае, если есть путь из вершины i в вершину j, иначе $A_{ij} = 0$. Диагональные элементы можно полагать равными 0 или 1, что скажется на некоторых анализируемых ниже свойствах транзитивного замыкания.

Для матрицы смежности $A_{n,n}$ (с диагональными элементами $= 0$, без петель) рассмотрим матрицу $A^2_{n,n} = A \times A$. Элементы этой матрицы определяются как $a_{ij}^{(2)} = \sum_{k=1}^{n} a_{ik} a_{kj}$ ($i, j = 1,...,n$). Очевидно, для каждого k условие $a_{ik} a_{kj} = 1$ выполняется только при выполнении условий $a_{ik} = 1$ и $a_{kj} = 1$, т. е. граф имеет ребра (i,k) и (k,j). Поэтому существует путь длиной 2 из i в j. Тогда значение $a_{ij}^{(2)}$ ($i \neq j$) равно числу *различных* путей длиной 2 из v_i в v_j, проходящих через *различные* вершины v_k.

Диагональные элементы $a_{ii}^{(2)} = \sum_{k=1}^{n} a_{ik} a_{ki}$. Следовательно, в неориентированном графе они равны степеням вершин, а в орграфе сумме неориентированных ребер (входящих и выходящих), инцидентных данной вершине.

Если в первоначальной матрице смежности диагональные элементы положить $=1$ (а не 0), то в матрице элементы матрицы $a_{ij}^{(2)}$ ($i \neq j$) равны числу *различных* путей длиной 1 и длиной 2 из v_i в v_j.

Далее обнулив диагональные элементы, вычислим $A_{n,n}^3$. Аналогично элемент матрицы $A_{n,n}^3$ определяет число путей длиной 3 из v_i в v_j, проходящих через две другие вершины.

Подобным образом матрица $A_{n,n}^q$ определяет число путей длиной q, проходящих через $q-1$ вершин из v_i в v_j.

Количество путей различной длины из v_i в v_j равно $T = A + A^2 + {} + \ldots + A^{n-1}$. Элемент t_{ij} равен числу путей из v_i в v_j длиной $\leq (n-1)$. Если элемент $t_{ij} > 0$, то вершина достижима из v_i в v_j.

Диагональные элементы a_{ii} матрицы $A_{n,n}^n$ равны количеству циклов, в которых участвует i вершина.

Можно продолжить процесс возведения в степень $> n$ и рассматривать пути длиной $> n$. Но в этом нет необходимости, поскольку любой такой путь хотя бы один раз должен повторно пройти через одну из вершин из n вершин графа. При этом не возникает какой-либо новой информации в транзитивное замыкание, поскольку обе вершины, связанные таким путем, уже соединены также другим путем, длина которого $< n$. Этот путь можно получить, удалив цикл из пути в повторно посещенную вершину.

Функция определения путевой матрицы $P_{n,n}$ с $t = O(n^3)$ использует алгоритм Флойда–Уоршелла.

```
    int Path(int **A, int **P, int n)
{   for(i=0; i<n; i++)  // copy A to P
    for(j=0; j<n; j++)
    P[i][j]=A[i][j];
    for(k=1; k<=n; k++) // calculation
    for(i=1; i<=n; i++)
    for(j=1; j<=n; j++)
    P[i][j]=P[i][j] || P[i][k] && P[k][j];
    return 1;
}
```

Метрические характеристики графа

Metric characteristics graph

Пусть $G = (V, E)$ – связный граф (или псевдограф).

Расстоянием между двумя вершинами графа называется длина кратчайшего пути, соединяющего эти вершины. Расстояние между вершинами u и v обозначается через $d(u,v)$. Если в графе нет пути, соединяющего u и v, то есть эти вершины принадлежат разным компонентам связности, то расстояние между ними считается бесконечным.

Функция $d(u,v)$ обладает свойствами:

$d(u,v) \geq 0$, причем $d(u,v) = 0$ only for $u = v$;

$d(u,v) = d(v,u)$ (в неориентированном графе);

$d(u,v) + d(v,w) \geq d(u,w)$ (неравенство треугольника);

$d(u,v) < \infty$ в связном неориентированном графе.

Функцию двух переменных, определенную на некотором множестве и удовлетворяющую вышеприведенным условиям, называют *метрикой*, а множество, на котором задана метрика – *метрическим пространством*. Поэтому множество вершин любого графа можно рассматривать как метрическое пространство.

Матрица $P_{n,n}$, где элементы матрицы $p_{ij} = d(v_i,\ v_j)$, называется *матрицей расстояний*. Матрица $P_{n,n}$ симметрична.

Максимальное удаление – эксцентриситет (*eccentricity*) от вершины u называется величина

$$\mathrm{ecc}(u) = \max_{v \in V} d(u,v).$$

Эксцентриситет $ecc(v_i)$ равен наибольшему из чисел в i-ой строке в матрице расстояний (весов).

Величина наибольшего эксцентриситета называется *диаметром* графа:

$$\mathrm{diam}(G) = \max_{u \in V} \max_{v \in V} d(u,v).$$

Величина наименьшего эксцентриситета называется *радиусом* графа:

$$\mathrm{rad}(G) = \min_{u \in V} \max_{v \in V} d(u,v).$$

Вершину с наименьшим эксцентриситетом называют *центральной*, или *центром*, а вершину с наибольшим – *периферийной*. Центры определяются неоднозначно, количество центров ≥ 1.

Наименьший диаметр имеет полный граф – его диаметр равен 1. Среди связных графов с n вершинами наибольший диаметр, равный $n-1$, имеет цепь P_n.

Если расстояние между двумя вершинами равно диаметру графа, то кратчайший путь, соединяющий эти вершины, называется *диаметральным путем*, а подграф, образованный вершинами и ребрами этого пути, называется *диаметральной цепью*.

Пример. Задан граф.

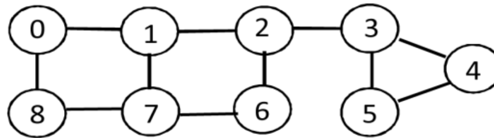


Рис. 54

Эксцентриситеты вершин приведены в таблице.

vertex	0	1	2	3	4	5	6	7	8
ecc	4	3	3	4	5	5	3	4	5

Радиус графа равен 3, диаметр 5, центры графа есть вершины 1, 2, 6; периферийные вершины – 8, 4, 5. Одна из диаметральных цепей порождается множеством вершин (8, 7, 6, 2, 3, 4).

Нахождение центральных вершин имеет практическое значение. Пусть, например, граф представляет собой сеть дорог, то есть вершины соответствуют населенным пунктам, а ребра – дорогам между ними. Требуется оптимально разместить пункты обслуживания. В подобных задачах оптимизация заключается в минимизации расстояния от места обслуживания до наиболее удаленного населенного пункта. Следовательно, местами размещения должны быть центральные вершины графа. В реальных задачах приходится учитывать и другие обстоятельства: расстояния между населенными пунктами, стоимость проезда, время проезда и т. д. Для учета этих параметров используются *взвешенные* графы.

Функция `Metrics` вычисляет эксцентриситеты E, диаметр `d`, радиус r и центры графа C.

```
    int Metrics(int **A, int n)
{   int **D,**P, *E,*C,d,r,max,i,j;
    E=(int*)calloc(n,4); C=(int*)calloc(n,4);
    Floyd(A, D, P, n);
    for(i=0; i<n; i++)
 {  for(j=0; j<n; j++)
```

```
 { if(D[i][j]==inf) D[i][j]=0;
   if(j==0) max=D[i][j]; else
   if(D[i][j]>max) max=D[i][j];
 } // for j
   E[i]=max;
   if(i==0) {d=max; r=max;}
   else
   {if(E[i]>d) d=E[i]; if(E[i]<r) r=E[i]; }
}  // for i
   printf(" d=%i r=%i\n",d,r);
   j=0;
   for(i=0; i<n; i++)
{  if(r==E[i]) {C[i]=i; j++;} else C[i]=-1;
}  // for i
   printf("centers count=%i\n",j);
   for(i=0; i<n; i++)
   {if(C[i]!=-1) printf("E[%i]=%i\n", i, E[i]);}
   free(D); free(P); free(E); free(C);
   return d; }    // Metrics
```

Основным понятием при формулировке следующих задач размещения вершин является понятие передаточного числа вершины v взвешенного графа *G*.

Пусть в графе *G* каждому ребру e сопоставлено число $w(e)$, каждой вершине v – число *q*. Передаточным числом вершины v_0 называется величина $\varphi(v_0) = \sum_{i \neq 0} \rho_\mu(v_0, v_i) q(v_i)$, где μ – кратчайшая (v_0, v_i)-цепь

$$\rho_\mu(v_0, v_i) = \sum_{v_i \in \mu} w(v_i) .$$

Медианой графа называется такая вершина v_0, для которой величина $\varphi(v_0)$ принимает минимально возможное значение.

Понятие медианы графа естественным образом обобщается. Подмножество вершин *V* называется *M* медианой, если существует такое разбиение вершин *V* на подмножества $V_1, V_2, \ldots, V_M$, что выражение

$$\varphi(V) = \sum_{v_k \in V_k} \sum_{v_i^{(k)} \in V_k} \rho(v_k, v_i^{(k)}) q(v_i^{(k)})$$

принимает минимально возможное значение $\mu(V) = \min_V \varphi(V)$.

Задача поиска *M*-медианы является *NP*-полной.

Циклы в графе

Cycle*, *circuit*, *loop

Фундаментальное множество циклов

Basic cycle set

Для того чтобы определить содержит ли граф цикл, достаточно провести поиск в глубину. Если DFS содержит обратное ребро, тогда в графе есть цикл. Причем это справедливо как для неориентированных, так и для ориентированных графов.

Существенной задачей становится определение минимального множества циклов неориентированного графа, из которого могут быть построены все циклы графа. Это так называемое фундаментальное множество циклов, или базис циклов.

После того, как создано остовное дерево T связного неориентированного графа, каждое дополнительное ребро графа, не принадлежащее дереву T, порождает один цикл. Такой цикл есть элемент фундаментального множества циклов графа относительно дерева T. В каждом остовном дереве графа $n-1$ ребер, следовательно, фундаментальное множество циклов графа относительно любого остовного дерева графа содержит $\nu = m - n + 1$ циклов.

Фундаментальное множество циклов графа полностью определяет циклическую структуру графа. Все циклы графа могут быть представлены комбинацией циклов из фундаментального множества.

Пусть $B = \{C_1, C_2, ..., C_q\}$ – фундаментальное множество циклов. Тогда любой цикл графа можно записать в виде $C_i \otimes C_j \otimes ... \otimes C_k$. Здесь символ $\otimes$ обозначает операцию симметрической разности, определяемой как $X \otimes Y = \{x : x \in X \bigcup Y, x \notin X \bigcap Y\}$. Эта операция эквивалентна сложению по модулю 2, если каждый фундаментальный цикл $C_i, i = 1, 2, ..., q$, представить k-мерным двоичным вектором, в котором j-я компонента равна 1 или 0 в зависимости от того, принадлежит или нет j-е помеченное ребро данному циклу. Элементы множества фундаментальных циклов являются *независимыми*, ибо ни один цикл из этого множества не может быть получен в результате линейной комбинации операции симметрической разности остальных циклов.

Рекурсивная функция `DFScycle` *нахождения множества циклов*

```
    int DFScycle(int **A,int *Vertex,int n,
    int *Count,int &count,int k,int i)
    // k=previus vertex i=current vertex
{   int q,j,countj,counti;
    Count[i]=++count;
    Vertex[count]=i;
    for(j=0; j<n; j++) // scan of adjacency matrix
 {  if(A[i][j]==0) continue; // there is no edge
    if(Count[j]==-1) // virgin j-vertex
    DFScycle(A,Vertex,n,Count,count,i,j);
 }
    else // if deja vu
    if(Count[j]<Count[i] && j!=k)
 {  cycle++;  // cycle number
    countj=Count[j]; counti=Count[i];
    printf("cycle=%2i\n",cycle);
    for(q=countj; q<=counti; q++)
    printf("%2i ",Vertex[q]);
    printf("\n");
  } // cycle
 }  // for j
    return count;
}   // DFScycle
```

Вызов функции `DFScycle` *нахождения множества циклов*

```
    int TestDFScycle()
{   int **A,n,m,i,j,k,count,*Count,*Vertex;
    n=11;
    A=InEdge2(1,n,m,0,1,0,2,0,3, 1,4, 2,3,2,5,
    3,4,3,5, 4,7, 5,6, 6,7, 7,8, 8,9,8,10, 9,10, 0,0);
    Count=(int*)calloc(n,4);
    Vertex=(int*)calloc(n,4);
    count=-1;
    for(i=0; i<n; i++)
    {Count[i]=-1;Vertex[i]=-1;}
    DFScycle(A,Vertex,n,Count,count,-1,k);
    return 1;
}   // TestDFScycle
```

Пример

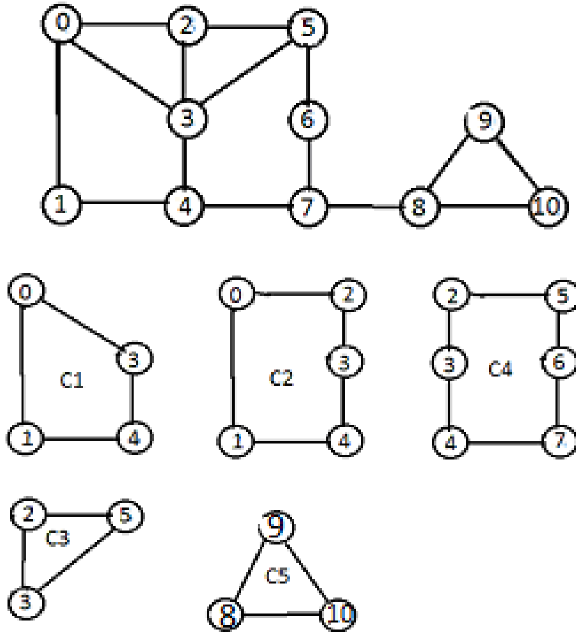


Рис. 55. Граф и его циклы

На рис. 55 показано фундаментальное множество циклов, которое генерирует приведенная программа на основе дерева DFS-поиска с начальной 0-й вершиной.

1 cycle: 0, 1, 4, 3;
2 cycle: 0, 1, 4, 3, 2;
3 cycle: 3, 2, 5;
4 cycle: 4, 3, 2, 5, 6, 7;
5 cycle: 8, 9, 10.

Например, может быть получен цикл $\{0,2,5,6,7,4,1\} = C_2 \otimes C_4$.

Следует заметить, что не каждая такая операция приводит к циклу в общепринятом определении цикла. Например, $C_3 \otimes C_5$ состоит из двух не связанных между собой циклов.

Число фундаментальных циклов всегда $\nu = m - n + 1$, но сами циклы определяются неоднозначно, и зависят от первоначально выбранного

остова *T*, ибо каждый цикл получается добавлением одного ребра, не принадлежащего *T*, к части остова или остову. Число ν называют *цикломатическим числом*, при этом полагается, что граф *G* состоит из одной связной компоненты.

Разрез графа

Cutset, separating set

Разрезом связного графа называют множество ребер, удаление которых делает граф несвязным. *Простой разрез* – это минимальный разрез, т. е. такой разрез, никакое собственное подмножество которого разрезом не является.

Простой разрез разбивает связный граф на две связные компоненты. В общем случае любой разрез является объединением некоторого числа простых разрезов. Между циклами и простыми разрезами графа существует определенная связь, поэтому простые разрезы называют *коциклами*. Действительно, остов определяется как минимальное число ребер, связывающих все вершины графа, в то время как простой разрез есть минимальное количество ребер, отделяющих одни вершины от других. Отсюда следует, что любой остов графа *G* должен иметь, по крайней мере, одно общее ребро с каждым простым разрезом, и количество простых разрезов есть $\rho(G) = n - 1$, называемое *коцикломатическим числом* графа. Подграф графа, составленный из ребер, не входящих в дерево (остов) T, называется *кодеревом* относительно дерева *T*.

Если взять любое ребро u_i из остова *T* графа *G*, то его удаление разбивает *T* на две связные компоненты K_1 и K_2. В *G* есть еще обратные ребра между вершинами из K_1 и K_2: $v_1, v_2, ..., v_j$. Множество ребер $Q = \{u_i, v_1, v_2, ..., v_j\}$ образует простой разрез, который называется *фундаментальным разрезом* *G* относительно ребра u_i остова T. Множество построенных таким образом разрезов (коциклов) $\{Q_1, Q_2, ..., Q_{n-1}\}$ относительно данного остова T называют *фундаментальным множеством разрезов*, или *базисом пространства коциклов* графа G. Их число $\rho(G)$.

Матрица фундаментальных циклов $C = \{c_{ij}\}$ графа *G* определяется как матрица, состоящая из q строк и *m* столбцов. Элемент этой матрицы $c_{ij} = 1$, если j-ребро принадлежит i-циклу, иначе $c_{ij} = 0$. Матрица фундаментальных коциклов определяется аналогично.

Ориентированные ациклические графы

Directed Acyclic Graph*, *DAG

Ориентированный ациклический граф – это орграф (*digraph*), не имеющий циклов. Ациклический орграф более общая структура, чем дерево, но менее общая по сравнению с обычным ориентированным графом.

На рис. 56 представлены дерево, ациклический орграф и ориентированный граф с циклом.

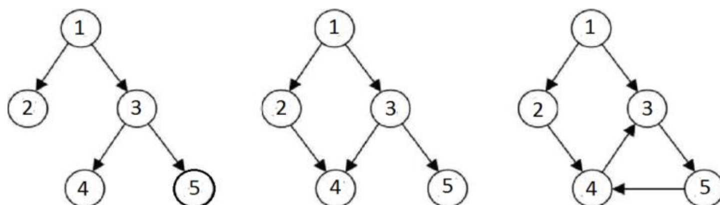


Рис. 56. Три ориентированных графа.
Дерево, ациклический орграф и ориентированный граф с циклом

Цикломатическое число графа – минимальное число ребер, которые надо удалить, чтобы граф стал ациклическим. Существует соотношение $p_1 = p_0 + m - n$, где p_1 – цикломатическое число, p_0 – число компонент связности графа, m – число рёбер, n – число вершин.

Ациклические ориентированные графы можно использовать для представления синтаксических структур арифметических выражений, имеющих повторяющиеся подвыражения. Например, на рис. 57 показан ациклический орграф для выражения $c/(a-b)+(a-b)\times(d+e)$.

Подвыражение $(a-b)$ встречается в выражении два раза, поэтому оно представлено вершинами, в которые входят две дуги.

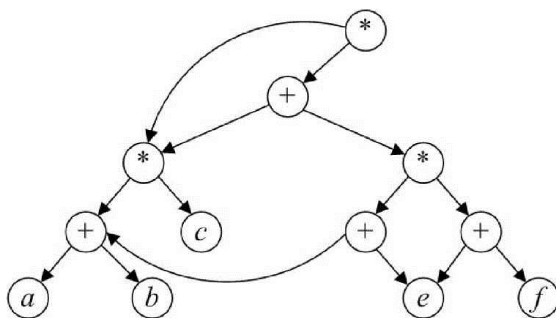


Рис. 57. Ориентированный ациклический граф
для арифметического выражения

Проверка ацикличности орграфа

Предположим, что есть ориентированный граф $G=(V,E)$. Необходимо определить, является ли он ациклическим, т. е. имеет ли он циклы. Чтобы ответить на этот вопрос, можно использовать метод поиска в глубину. Если при обходе орграфа G методом поиска в глубину встретится обратная дуга, то ясно, что граф имеет цикл. С другой стороны, если в орграфе есть цикл, тогда обратная дуга обязательно встретится при обходе этого орграфа методом поиска в глубину.

Пусть орграф G имеет цикл (рис. 58).

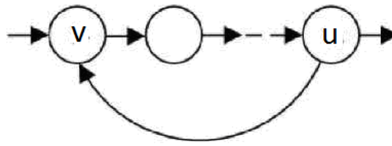


Рис. 58. Ациклический орграф,
представляющий структуру предшествований

Пусть при обходе данного орграфа методом поиска в глубину вершина v имеет наименьшее глубинное число среди всех вершин, составляющих цикл. Рассмотрим дугу $u-v$, принадлежащую этому циклу. Поскольку вершина u входит в цикл, то она должна быть потомком вершины v в глубинном остовном лесу. Поэтому дуга $u-v$ не может быть поперечной дугой. Так как глубинный номер вершины u больше глубинного номера вершины v, то отсюда следует, что эта дуга не может быть дугой дерева и прямой дугой. Следовательно, дуга $u-v$ является обратной дугой, как показано на рис. 58.

Эйлеров цикл

Euler cycle

Определения

Эйлерова цепь проходит по одному разу по всем ребрам графа, однако вершины могут повторяться. Начальная и конечная вершины цепи могут быть различными.

*Эйлеров цикл п*роходит по одному разу по всем ребрам графа, однако вершины могут повторяться. Начальная и конечная вершины цикла совпадают.

Эйлеров граф есть граф, в котором существует эйлеров цикл.

Теорема Эйлера. *Для существования эйлеровой цепи в неориентированном графе* $G=(V,E)$ *необходимо и достаточно выполнения следующих условий.*

- *Граф связный.*
- *Степени внутренних вершин* (*не являющихся началом и концом цепи*) четные.
- *Пусть вершины* u *и* v *являются началом и концом цепи.*
  Если $u=v$ (*т. е. цикл*), *степени их четные.*
  Если $u \neq v$ (*т. е. цепь*), *степени их нечетные.*

Эйлеров цикл для ориентированных графов

Сильная связность. Две вершины в орграфе *сильно связаны*, если существуют пути в обоих направлениях.

Сильносвязный орграф есть орграф, в котором все вершины сильно связаны.

Теорема Эйлера для ориентированного графа. *Для существования эйлеровой цепи в орграфе необходимо и достаточно выполнения условий.*

- *Граф сильносвязный.*
- *Для каждой вершины орграфа количество входящих дуг равно количеству выходящих дуг* (*одинаковые степени захода и исхода*).

Ориентированный граф содержит эйлеров *путь* от вершины u к вершине v тогда и только тогда, когда:

- граф сильносвязный;
- вершина u имеет степень захода на единицу меньше, чем степень исхода;
- вершина v имеет степень захода на единицу больше, чем степень исхода;
- внутренние вершины (не являющиеся началом u и концом цепи v) имеют одинаковые степени захода и исхода.

Эти свойства эйлеровых графов позволяют легко проверить наличие цикла: сначала проверяется граф на связность, выполнив обход в глубину или ширину, а потом подсчитывается количество вершин с нечетной степенью.

Алгоритм построения эйлерова цикла для графа

Пусть v_0 – некоторая произвольная вершина графа. Двигаемся в глубину от вершины v_0 по неиспользованным ребрам графа до тех пор, пока не вернемся опять в эту вершину. Эта цепь (цикл) может закончиться только в вершине v_0, так как при входе в любую другую вершину, всегда существует ребро, по которому можно выйти из нее (степени вершин чет-

ные). В результате построен некий цикл C_1. Если все ребра графа использованы, то C_1 есть полный цикл и процесс закончен.

Если же существуют неиспользованные ребра, то в силу связности графа существует такая вершина графа u, которая принадлежит циклу C_1 и является концом какого-то еще неиспользованного ребра.

Из исходного графа удаляются все ребра цикла C_1. В оставшемся графе все вершины по-прежнему будут иметь четную степень, поскольку при удалении ребер степени каждой вершины цикла уменьшаются на четное число. В полученном графе строится цикл C_2, начиная движение от вершины u.

Если все ребра использованы, то эйлеров цикл построен. Искомый эйлеров цикл – это часть цикла C_1 от вершины v_0 до вершины *u*, а затем оставшаяся часть цикла C_2 от вершины *u* до вершины v_0.

Если все ребра не использованы, то полагается $C = C_1 \cup C_2$, и затем возвращаемся к итеративному шагу алгоритма.

Рекурсивная функция построения эйлерова цикла

Пусть для заданного графа выполняются условия теоремы. Необходимо найти эйлеров цикл. Граф обходится в глубину, при этом ребра удаляются. Номера пройденных вершин запоминаются. При обнаружении вершины, из которой не выходят ребра, поскольку они удалены, ее номер записывается в стек, и просмотр продолжается от предыдущей вершины. Обнаружение вершин с нулевым числом ребер говорит о том, что найден цикл. Его можно удалить, четность вершин (количество выходящих ребер) при этом не изменится. Процесс продолжается до тех пор, пока есть ребра. В стеке после этого будут записаны номера вершин графа в порядке, соответствующем эйлерову циклу.

```
    int Euler(int i)
{   int j;
    for(j=0; j<n; j++)
    if(A[i][j]!=0)
 {  A[i][j]=0; A[j][i]=0;
    Euler(j);
 }
    Stack[top]=i; top++;
    return i;
}   // Euler
```

Итеративная функция построения эйлерова цикла

```
    int EulerIter(int i, int E[], int &topE)
{   int S[100], topS=0, j, k;
    topE=-1;
    S[topS]=i;
    while(topS>=0)
 {  k=S[topS];
    for(j=0; j<n; j++)
    if(A[k][j]!=0)
  { A[k][j]=A[j][k]=0;
    S[topS++]=j; break;
  } // if
    if(j==n)
  { k=S[topS--];
    E[topE++]=k;
  }
 }  // while (topS>0)
    return topE;
}   // EulerIter
```

Пример. Эйлеров цикл в графе:

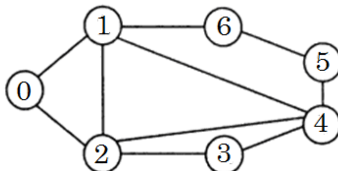


Рис. 59

Прохождение вершин в цикле: 0, 2, 4, 5, 6, 1, 4, 2, 1, 0.

Не все графы имеют эйлеровы циклы, но если эйлеров цикл существует, то это означает, что, следуя вдоль этого цикла, можно нарисовать граф на бумаге, не отрывая от нее карандаша.

Двойной эйлеров цикл

В двойном эйлеровом цикле каждое ребро графа проходится два раза в противоположных направлениях. Двойной эйлеров цикл существует в любом связном графе.

Расщепив каждое ребро графа на две дуги противоположной ориентации, получим орграф, в котором для всех вершин $\deg^+ = \deg^-$. В та-

ком графе существует эйлеров контур, который и является двойным эйлеровым циклом в исходном графе.

Для построения двойного эйлерова цикла можно применить следующий алгоритм.

1. Движение начинается с произвольной вершины.

2. По уже пройденным дугам не надо идти.

3. Когда вершина посещается первый раз, то отмечается предыдущая вершина.

4. Не возвращаться из данной вершины в вершину, меченную в п. 3, до тех пор, пока имеются другие возможности.

Пример. Нахождение двойного эйлерова цикла для графа (рис. 60).

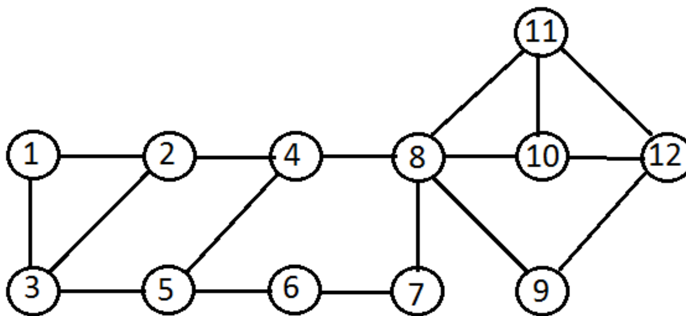


Рис. 60

Двойной эйлеров цикл:

1–3–5–4–2–3–2–1–2–4–8–7–6–5–6–7–8–9–12–10–11–12–11–8–10–8–11–10–12–9–8–4–5–3–1.

Гамильтонов цикл

Hamilton cycle

Гамильтоновым циклом на графе называется простой цикл, содержащий *все* его вершины. Граф называется гамильтоновым, если в нем имеется гамильтонов цикл.

Гамильтоновой называют и *простую цепь* (*не цикл*), содержащую каждую вершину этого графа. Граф, содержащий гамильтонову цепь, называется *полугамильтоновым*.

Аналогично определяется *гамильтонов контур* и *гамильтонов путь* в ориентированном графе.

Существуют популярные задачи, приводящие к нахождению гамильтоновых маршрутов.

Задача Гамильтона. Каждой из двадцати вершин додекаэдра приписывается название одной из столиц государства. Требуется, проходя по ребрам додекаэдра, вернуться в исходный город, посетив каждую столицу ровно один раз. Эта задача приводит к отысканию в графе додекаэдра гамильтонова цикла.

Задача о шахматном коне. Требуется обойти конем все клетки шахматной доски, побывав в каждой клетке по одному разу и последним ходом вернуться в начальную клетку. Задача сводится к нахождению гамильтонова цикла графа.

Задача коммивояжера

Коммивояжер должен объехать ряд населенных пунктов, побывав в каждом пункте только один раз и вернуться в исходный пункт. Коммивояжер должен выбрать такой маршрут, чтобы пройденный путь был наименьшей длины.

Рассматривается граф, вершины которого – населенные пункты, ребра (дуги) – дороги. Каждому ребру (дуге) (v_i, v_j) соответствует вес w_{ij} – длина соответствующей дороги. Требуется найти гамильтонов маршрут наименьшей длины. Задача коммивояжера – классическая задача дискретной оптимизации и имеет многочисленные приложения.

Задача нахождения гамильтоновых маршрутов имеет внешнее сходство с задачей нахождения эйлеровых маршрутов. Однако решение задачи Гамильтона значительно труднее. Это кажется естественным, поскольку в эйлеровом цикле ребра проходятся по одному разу, а на вершины это ограничение не распространяется. В гамильтоновом цикле единственность прохождения гораздо более жесткая, ограничена как ребрами, так и вершинами.

Сложность решения задач нахождения гамильтоновых маршрутов проявляется, во-первых, в отсутствии достаточно общих критериев их существования и, во-вторых, в отсутствии точных эффективных алгоритмов их отыскания для больших размерностей. Кажется естественным, что если граф содержит много ребер и эти ребра к тому же достаточно равномерно распределены, то граф, вероятно, является гамильтоновым. Так, например, в полном графе с n вершинами существует $(n-1)!$ гамильтоновых циклов.

Гамильтоновы циклы наименьшей длины в полном графе можно найти по следующей простой схеме. Выберем некую вершину. Из нее $(n-1)$ способами можно перейти в другую вершину. Из второй вершины

$(n-2)$ способами возможен переход в третью вершину и т. д. В результате получим $(n-1)!$ гамильтоновых циклов, из которых выбирается минимальный. Поскольку при $n \to \infty$ $n! \sim \sqrt{n}(n / e)^n$, и время решения задачи пропорционально числу возможных циклов, то нахождение гамильтоновых циклов методом *полного перебора* оказывается практически невозможным даже для сравнительно небольших $n \sim 16$. Задача нахождения гамильтоновых маршрутов принадлежит к классу NP полных задач.

Задача коммивояжера формально может быть сформулирована следующим образом. Пусть орграф с n вершинами задан матрицей расстояний W, элементы которой w_{ij} – длины дуги (i, j). Если переход из i-й вершины в j-ю не возможен, то $w_{ij} = \infty$.

Пусть $\bar{H} = \{H_1, H_2, \dots, H_m\}$ – множество гамильтоновых контуров на орграфе и $L(H_C) = w_{i_1 i_2} + w_{i_2 i_3} + \ldots + w_{i_n i_1}$ – длина гамильтонова контура H_C. Рассматривается комбинаторная оптимизационная задача: требуется найти гамильтонов контур $H_m \in \bar{H}$ наименьшей длины $L(H_m) = \min\limits_{H_C \in \bar{H}} L(H_C)$.

Такой гамильтонов контур *оптимальный*. Если на графе существует хотя бы один оптимальный гамильтонов контур, то эта задача имеет решение и называется задачей *коммивояжера*.

Существование гамильтоновых маршрутов

Рассмотрим достаточные условия гамильтоновости графа, гарантирующие существование гамильтоновых контуров и циклов.

Теорема (Кёниг). *В полном ориентированном графе* (*любая пара вершин которого соединена хотя бы в одном направлении*) *всегда существует гамильтонов путь.*

Доказательство. Пусть $d = (v_1, v_2, \ldots, v_p)$ – путь длины $(k-1)$, $k > 1$, все вершины которого различны (длина пути – это число дуг). Тогда можно показать, что для любой вершины $u \notin d$, в силу полноты орграфа, можно построить путь d_i, содержащий вершину u:

$d_i = (v_1, v_2, \ldots, v_i, u, v_{i+1}, \ldots, v_p)$ при некотором $0 \le i \le k$. В частности, $d_0 = (u, v_1, v_2, \ldots, v_k)$, $d_k = (v_1, v_2, \ldots, v_k, u)$.

Таким образом, можно шаг за шагом построить путь, содержащий все вершины графа по одному разу.

Теорема (Дирак, 1952). *Если в простом графе* G *порядка* $n \ge 3$ *для любой вершины* v *выполняется неравенство* $\deg v \ge n/2$, *то граф* G *– гамильтонов.*

Доказательство. От противного. Пусть G – не гамильтонов. Добавим к графу G минимальное количество k новых вершин $u_1, u_2, \ldots, u_k$, соединяя каждую из них с каждой вершиной графа G так, чтобы полученный граф $G' = G + u_1 + u_2 + \ldots + u_k$ стал гамильтоновым. Затем, считая, что $k > 0$, придем к противоречию.

Пусть $(v, u_1, w, \ldots, v)$ – гамильтонов цикл в графе G', где v и w – вершины из G, а $u_1 \in G'$, $u_1 \notin G$, – одна из новых вершин. Такая пара вершин v и u_1 в гамильтоновом цикле обязательно найдется, иначе граф G был бы гамильтоновым. Тогда $w \notin G$, $w \notin (u_1, u_2, \ldots, u_k)$, иначе вершина u_1 была бы не нужна. Более того, вершина w не является смежной с вершиной v, иначе вершина u_1 была бы не нужна.

Далее, если в цикле $(v, u_1, w, \ldots, v', w', \ldots, v)$ есть вершина w', смежная с вершиной w, то вершина v' не смежная с вершиной v, так как иначе можно было бы построить гамильтонов цикл $(v, v', \ldots, w, w', \ldots, v)$ без вершины u_1, взяв последовательность вершин $w, \ldots, v'$ в обратном порядке. Из этого следует, что число вершин графа G', не являющихся смежными с v, не меньше числа вершин, смежных с w, т. е. больше либо равно $n/2 + k$. С другой стороны, очевидно, что число вершин графа G', смежных с w, также больше либо равно $n/2 + k$. А так как ни одна вершина графа G' не может быть одновременно смежной и не смежной вершине w, то общее число вершин графа G', равное $n + k$, не меньше, чем $n + 2k$. Противоречие.

Теорема (Оре, 1960). *Если в графе* G *порядка* $n \geq 3$ *для любой пары несмежных вершин* u *и* v *выполняется неравенство* $\deg v + \deg u \geq n$, *то граф* G *– гамильтонов.*

Теорема Дирака является следствием теоремы Оре.

Теорема (Хватал, 1972). *Пусть* G *– обыкновенный граф,* $d_1 \leq \leq d_2 \leq ... d_n$ *– его степенная последовательность и* $n \geq 3$. *Если для любого* k *верна импликация* $d_k \leq k < n/2 \Rightarrow d_{n-k} \geq n-k$, *то граф* G *– гамильтонов.*

Пример. Нетрудно увидеть, что достаточные условия этих теорем для графа Петерсена не выполняются.

В графе Петерсена $n = 10$, $d_i = \deg v_i = 3$, поэтому $d_i < n/2 = 5$. Следовательно, теорема Дирака не выполняется. Теорема Оре также не выполняется, так как $d_i + d_j = 6 < n = 10$ при условии, что вершины v_i и v_j несмежные и $i, j = 1, 2, ..., 10,\ i \neq j$.

Проверка условия теоремы Хватала. Для $1 \leq k < 5$ значение импликации $(d_k \leq k) \to (d_{n-k} \geq n-k)$.

$$k = 1: \quad (d_1 = 3 \leq 1) \to (d_9 = 3 \geq 9) = 1;$$

$$k = 2: \quad (d_2 = 3 \leq 2) \to (d_8 = 3 \geq 8) = 1;$$

$$k = 3: \quad (d_3 = 3 \leq 3) \to (d_7 = 3 \geq 7) = 0;$$

$$k = 4: \quad (d_4 = 3 \leq 4) \to (d_6 = 3 \geq 6) = 0.$$

Таким образом, для $k = 3$ и $k = 4$ условия теоремы нарушены.

Интересно, что почти нет эйлеровых графов. А почти все графы гамильтоновы.

Теорема (Перепелица, 1969). *Пусть* $P(n)$ *– множество всех простых помеченных графов с* n *вершинами,* $P_H(n)$ *– множество всех простых помеченных гамильтоновых графов с* n *вершинами.*

Тогда $\lim_{n\to\infty}(|P_H(n)| \,/\, |P(n)|) = 1$.

Таким образом, вероятность того, что «случайный граф» с n *вершинами является гамильтоновым, стремится с увеличением* n *к единице.*

Алгоритм нахождения гамильтоновых циклов

Пусть *G* – произвольный граф с *n* вершинами. Опишем алгоритм с возвратом (полного перебора), позволяющий найти в графе *G* все гамильтоновы циклы или определить отсутствие таких циклов. Пусть v_0 – произвольная вершина графа *G*. Рассмотрим некоторый гамильтонов цикл $v_0 = u_1, u_2, ..., u_n, u_{n+1} = v_0$.

Если существует алгоритм построения всех простых максимальных цепей с началом в вершине v_0, то алгоритм нахождении гамильтоновых циклов также будет существовать. Действительно, пусть

$$P : v_0 = u_1, u_2, ..., u_k$$

– простая максимальная цепь с началом в вершине v_0. Если $k = n$ и вершина u_k смежная с вершиной v_0, то, добавляя к цепи *P* ребро (u_k, v_0), получим гамильтонов контур.

Алгоритм, позволяющий перечислить по одному разу все гамильтоновы циклы графа *G*, многократно выполняет следующие действия: имея текущую простую цепь $P : v_0 = u_1, u_2, ..., u_{k-1}$, он по очереди добавляет к ней новые вершины, продолжая ее до простых всевозможных максимальных цепей с началом в вершине v_0.

```
    int Hamilton(int **A, int n, int k) // k = step number
{   int i, j, q; static int count = 0;
    i = Vertex[k-1];      // i = number last vertex
    for(j = 0; j<n; j++)
    if(A[i][j]! = 0)      // is edge
 {  f(k == n && j == 0)
  { count++;
    printf("circuit: c = %2i\n", count);
    for(q = 0; q<n; q++)
    printf("Vertex: k = %2i j = %2i\n", q, Vertex[q]);
    printf("\n");
  }
    else if(!Mark[j])
  { Vertex[k] = j;
    Mark[j] = 1; Hamilton(A,n,k+1); Mark[j] = 0;
  }    // if(Mark[j])
 }  // for j
    return i;
}   // Hamilton
```

В вызывающей функции:

```
const int m; static int Vertex[m], Mark[m];
Mark[0] = 1;
Hamilton(H,m,1);
```

Пример. Цикл Гамильтона в графе:

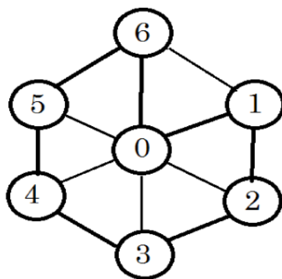


Рис. 61

Матрица смежности:

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

cycle = 1 vertex: 0 1 2 3 4 5 6
cycle = 2 vertex: 0 1 6 5 4 3 2
cycle = 3 vertex: 0 2 1 6 5 4 3
cycle = 4 vertex: 0 2 3 4 5 6 1
cycle = 5 vertex: 0 3 2 1 6 5 4
cycle = 6 vertex: 0 3 4 5 6 1 2
cycle = 7 vertex: 0 4 3 2 1 6 5
cycle = 8 vertex: 0 4 5 6 1 2 3
cycle = 9 vertex: 0 5 4 3 2 1 6
cycle = 10 vertex: 0 5 6 1 2 3 4
cycle = 11 vertex: 0 6 1 2 3 4 5
cycle = 12 vertex: 0 6 5 4 3 2 1

Тетраэдр $n = 4$ $m = 6$

cycle = 1 vertex: 0 1 2 3 0
cycle = 2 vertex: 0 1 3 2 0
cycle = 3 vertex: 0 2 1 3 0
cycle = 4 vertex: 0 2 3 1 0
cycle = 5 vertex: 0 3 1 2 0
cycle = 6 vertex: 0 3 2 1 0

Куб $n = 8$ $m = 8$

cycle = 1 vertex: 0 1 2 3 4 7 6 5 0
cycle = 12 vertex: 0 5 6 7 4 3 2 1 0

Октаэдр $n = 6$ $m = 12$

cycle = 1 vertex: 0 1 2 3 5 4 0
cycle = 32 vertex: 0 4 5 3 2 1 0

Додекаэдр $n = 20$ $m = 30$

cycle = 1 vertex: 0 1 2 3 4 5 6 16 17 18 19 15 14 13 12 11 10 9 8 7 0
cycle = 60 vertex: 0 7 8 17 18 19 15 16 6 5 14 13 12 11 10 9 1 2 3 4 0

Икосаэдр $n = 12$ $m = 30$

cycle = 1 vertex: 0 1 2 3 4 5 6 10 9 8 11 7 0
cycle = 2560 vertex: 0 7 11 10 9 8 3 1 2 4 5 6 0

Топологическая сортировка

Topological sorting

Топологическая сортировка ориентированного ациклического графа (DAG – Directed Acyclic Graph, *directed graph without cycles*) есть такое упорядочение (переименовывание) всех его вершин, при котором для дуги – ориентированного ребра (u, v) – выполняется условие: номера $u < v$.

Рассматривается также так называемая обратная топологическая сортировка, при которой выполняется обратное условие $u > v$.

Если граф имеет цикл, такая сортировка невозможна. Обычно определенный порядок не бывает уникальным, сортировка неоднозначна.

Геометрическая интерпретация топологической сортировки графа может быть представлена либо в виде горизонтальной линии, когда все ребра направлены слева направо (или справа налево для обратной сортировки); либо вершины графа расположены по окружности, а дуги направлены по часовой стрелке (или против – для обратной сортировки).

Для прямой сортировки в деревьях обхода в глубину и ширину сыновья всегда имеют номер больший, нежели родитель.

В матрице смежности после прямой сортировки все элементы выше главной диагонали либо 1, либо 0, а элементы ниже главной диагонали – либо –1, либо 0. При обратной сортировке – наоборот. Здесь используется обозначение для матрицы смежности: элемент = 1 для вершины с исходящей дугой и –1 – для вершины с входящей дугой.

Рассматриваются обычно два алгоритма сортировки, основанные на совершенно разных принципах.

Топологическая сортировка графа DAG методом удаления истоков

Исток, источник (*source*) орграфа есть вершина, которая имеет только исходящие (выходящие) дуги, а полустепень захода (входа) $d^{+}(v) = 0$.

Сток (*sink* – вход, сток) орграфа есть вершина, которая имеет только входящие дуги, а полустепень исхода $d^{-}(v) = 0$.

В каждом графе DAG имеются как минимум один исток и один сток. Это видно из следующих рассуждений. Предположим, что в графе DAG нет стоков. Исходя из любой вершины, можно построить ориентированный путь произвольной длины. Существует как минимум одно такое ребро, следуя из этой вершины вдоль любого ребра в любую другую вершину, с последующим следованием из этой вершины вдоль другого ребра и т. д. Посещение $n+1$ вершин приводит в ориентированный цикл, что

противоречит предположению о том, что имеется DAG. Таким образом, в графе DAG существует по меньшей мере один сток. Отсюда также следует, что в каждом графе DAG присутствует как минимум один исток, поскольку исток представляет собой обращение стока.

Алгоритм топологической сортировки графа DAG методом удаления истоков

Алгоритм следует из приведенного свойства. Вычисляется исток – вершина без входящих ребер, некая корневая вершина. Эта вершина и все выходящие из нее ребра удаляются из графа. После этого вычисляется другая вершина без входящих ребер. Если такая вершина не найдена, граф содержит цикл и его нельзя подвергнуть топологической сортировке. Исток – корневая вершина имеет полустепень захода = 0. При удалении корневой вершины полустепени захода всех смежных вершин уменьшаются на 1, при этом создаются новые корневые вершины (одна или более). Для поиска корневых вершин удобно использовать очередь удаляемых вершин с полустепенями захода = 0. Как только ребра удаляются и создаются новые корневые вершины, они добавляются в очередь. В цикле номер такой вершины извлекается из очереди, а степени входа всех смежных с ней вершин уменьшаются на 1. Порядок, в котором новые вершины с нулевой степенью входа добавляются в очередь, и есть результат топологической сортировки.

Для реализации данного подхода создается очередь с помощью массива. Этот массив используется для хранения новых просмотренных корневых вершин и для возвращения результата.

```
    int SourceSort(int **A,int **B,int n)
{   int i,j,q,first,free,*Q,*Out,*In;
    Q=(int*)calloc(n,4);      // queue
    Out=(int*)calloc(n,4);    // source vertexes
    In=(int*)calloc(n,4);     // sink vertexes
    printf("Source Sorting\n");
    // calculation of source and sink
    DegreeOutIn(A,Out,In,n);
    printf("Deg Out In\n");
    for(int i=0; i<n; i++)
    printf("%3i %3i %3i \n",i,Out[i],In[i]);
    // search of source v0
    for(q=0; q<n; q++)
    if(In[q]==0) break;
    Q[0]=q;
    if(q==n)
```

```
    {MessageBox(0, "Graph has not deg In==0",
                   "Source sorting is not correct",MB_OK);
    return 0;}
    first=0;   // first item in queue
    free=1;    // free item for write
    // main loop
    while(first<free) // queue  |first...|free
  { q=Q[first++];      // read, ++
    for(j=0; j<n; j++)// in adj
    if(A[j][q]==-1 && In[j]>0)
   { In[j]--;
    if(In[j]==0) Q[free++]=j;
   } // if
  } // while
    for(i=0; i<n; i++)
    printf("%3i %3i \n",i,Q[i]);
    return 1;
}   // SourceSort
```

Функция `DegreeOutIn` вычисляет количество входящих и исходящих дуг для каждой вершины и возвращает количество ребер.

```
    int DegreeOutIn(int **A,int *Out,int *In,int n)
{   int i,j,m=0;
    for(i=0; i<n; i++)
  { Out[i]=In[i]=0;
    for(j=0; j<n; j++)
    if(A[i][j]==1) {Out[i]++; m++; }
    else if(A[i][j]==-1) In[i]++;
  } // for i
    return m;
}   // DegreeOutIn
```

Алгоритм топологической сортировки графа DAG на основе поиска в глубину

Топологическая сортировка выполняется алгоритмом поиска в глубину. При завершении обработки вершины она записывается в стек вершин (`Vertex`). Запись в стек может производиться либо при (типичном) увеличении вершины стека, либо при уменьшении (как в Intel-аппаратном стеке).

При увеличении вершины стека получается обратная топологическая сортировка, в которой вершины располагаются в обратном порядке; при уменьшении – прямая (обычная) сортировка. Затем вычисляется матрица смежности отсортированного графа, по которой строится изображение графа, и вычисляются деревья поиска в глубину и ширину.

```
    int green=0, yelBlock=1, red=2;
    // it is for visual only

    int DFSup(int n,int **A,int *Vertex,
              int *Color,int &count,int i,int direct)
  // suffix in name "up" means record on recursive lifting
{   int j;
    Color[i]=yelBlock;// i vertex processing begins
    for(j=0; j<n; j++)
 {  if(A[i][j]==1 && Color[j]==green)
    DFSmy(n,A,Vertex,Color,count,j,direct);
  }  // for j
    // record to stack
    if(direct==1)
    Vertex[--count]=i; // direct sort
    else if(direct==-1)
    Vertex[count++]=i; // reverse sorting; initial count=0
    Color[i]=red; // red is not used, serves for debugging
    return j;
}   // DFSup

    int DFSsort(int **A, int **B,int n,int v0, int direct)
    // A - input adjacency matrix
    // B - output adjacency matrix
    // n - graph dimension
    // v0 - source
    // direct sorting
{   int i,j,k,ret=1,count,
    *Color=(int*)calloc(n,4),
    *Vertex=(int*)calloc(n,4),
    *Index=(int*)calloc(n,4);
    // initial
    if(direct==1) count=n;else
    if(direct==-1) count=0;
    // main
    ret=DFSup(n,A,Vertex,Color,count,v0,direct);
    // reverse of indexes
```

```
    for(i=0; i<n; i++)
    Index[Vertex[i]]=i;
    printf("\n i Vertex Index\n");
    for(i=0; i<n; i++)
    printf("%2i %3i %3i\n",i,Vertex[i],Index[i]);
    ReNorm(A,B,Index,n);
    PrintMatrix(A,n,"Graph A before dfs sort");
    printf("\n");
    PrintMatrix(B,n,"Graph B after dfs sort");
    printf("\n");
    return ret;
}   // DFSsort

    int TestSort()
{   int **A,**B,*Out,*In,n,m,ret,v0,i,direct;
    n=8; m=15;
    A=InEdge(-1,n,m,0,4,0,6,0,1,
                  1,5,
                  2,0,2,3,2,4,
                  3,1,3,7,
                  4,1,4,3,
                  6,1,6,5,
                  7,5,7,1);
    B=(int**)CreateMatrix(n,n,4);
    Out=(int*)calloc(n,4); In=(int*)calloc(n,4);
    DegreeOutIn(A,Out,In,n);
    printf("Degree Out In\n");
    for(int i=0; i<n; i++)
    printf("%3i %3i %3i \n",i,Out[i],In[i]);
    for(i=0; i<n; i++)
    if(In[i]==0) break;
    if(i<n) v0=i; else
    MessageBox(0,"Graph has not v0",
    "Topological sorting is not correct", MB_OK);
    direct=1; // or -1
    ret=DFSsort(A,B,n,v0,direct);
    if(!ret) MessageBox(0,"Graph has cycle",
    " Topological sorting is not correct",MB_OK);
    ret=SourceSort(A,B,n);
    return ret;
}   // TestSort
```

Функция ReNorm изменяет индексы: старые на новые.

```
    int ReNorm(int **A,int **B,int *X,int n)
{   for(int i=0; i<n; i++)
    for(int i=0; i<n; i++)
    B[X[i]][X[j]]=A[i][j];
    return n;
}   // ReNorm
```

Пример. Приведены результаты сортировки графа методами: DFS обходом и удалением истоков (*top sorting*).

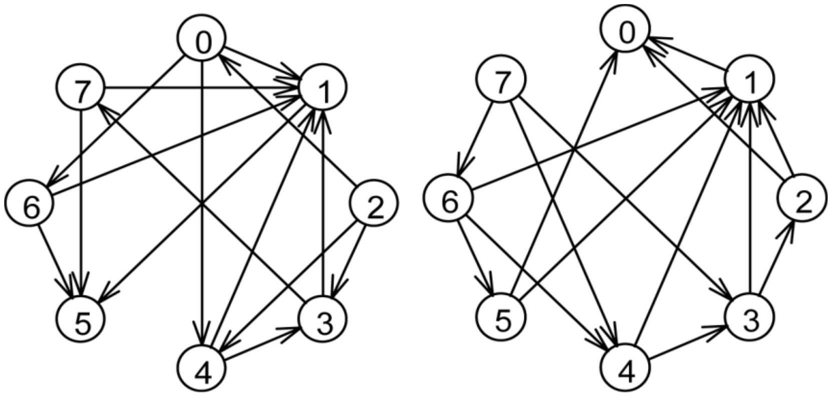


Рис. 62. Входной граф и граф после обратной DFS сортировки

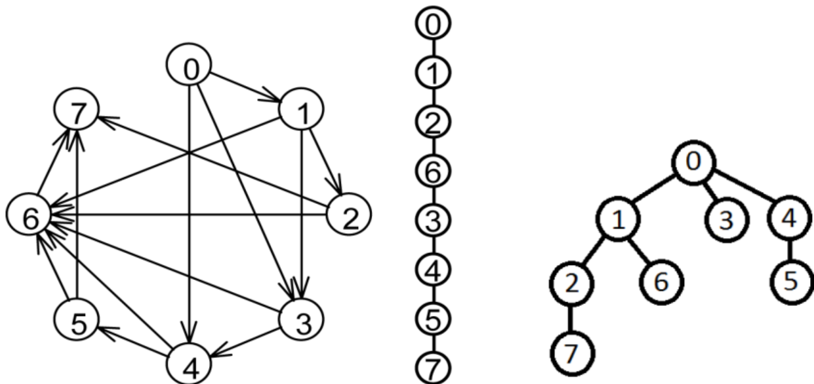


Рис. 63. Граф после прямой DFS сортировки.
Дерево DFS обхода. Дерево BFS обхода

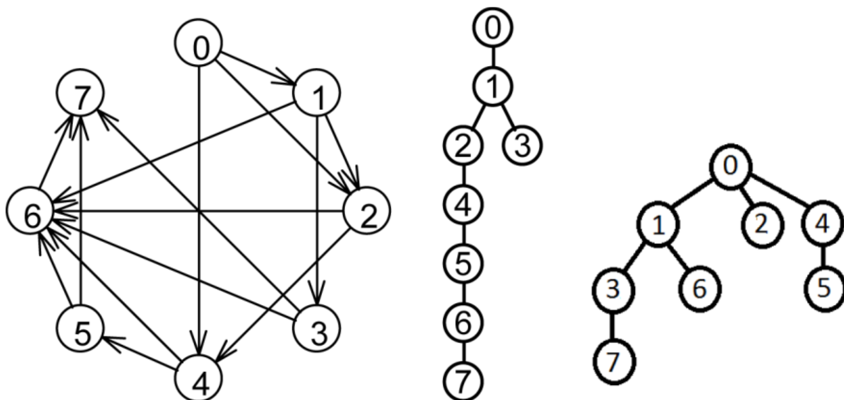


Рис. 64. Граф после source сортировки.
Дерево DFS обхода. Дерево BFS обхода

Матрица смежности – начальная.

i\j	0	1	2	3	4	5	6	7
0	0	1	-1	0	1	0	1	0
1	-1	0	0	-1	-1	1	-1	-1
2	1	0	0	1	1	0	0	0
3	0	1	-1	0	-1	0	0	1
4	-1	1	-1	1	0	0	0	0
5	0	-1	0	0	0	0	-1	-1
6	-1	1	0	0	0	1	0	0
7	0	1	0	-1	0	1	0	0

Матрицы смежности после source сортировки

i\j	0	1	2	3	4	5	6	7
0	0	1	1	0	1	0	0	0
1	-1	0	1	1	0	0	1	0
2	-1	-1	0	0	1	0	1	0
3	0	-1	0	0	0	0	1	1
4	-1	0	-1	0	0	1	1	0
5	0	0	0	0	-1	0	1	1
6	0	-1	-1	-1	-1	-1	0	1
7	0	0	0	-1	0	-1	-1	0

Матрица смежности после прямой DFS сортировки

i\j	0	1	2	3	4	5	6	7
0	0	1	0	1	1	0	0	0
1	-1	0	1	1	0	0	1	0
2	0	-1	0	0	0	0	1	1
3	-1	-1	0	0	1	0	1	0
4	-1	0	0	-1	0	1	1	0
5	0	0	0	0	-1	0	1	1
6	0	-1	-1	-1	-1	-1	0	1
7	0	0	-1	0	0	-1	-1	0

Матрица смежности после обратной DFS сортировки

i\j	0	1	2	3	4	5	6	7
0	0	-1	-1	0	0	-1	0	0
1	1	0	-1	-1	-1	-1	-1	0
2	1	1	0	-1	0	0	0	0
3	0	1	1	0	-1	0	0	-1
4	0	1	0	1	0	0	-1	-1
5	1	1	0	0	0	0	-1	0
6	0	1	0	0	1	1	0	-1
7	0	0	0	1	1	0	1	0

Остовные деревья

Spanning tree

Остовом (неориентированного) связного графа $G = (V, E)$ называется его подграф $T = (V, E)$, являющийся деревом. Остовное дерево (остовный подграф) содержит все вершины графа.

Минимальное остовное дерево графа (минимальный каркас или каркас минимального веса) есть ациклическое (не имеющее циклов) множество рёбер в связном, взвешенном и неориентированном графе, соединяющих между собой все вершины данного графа, при этом сумма весов всех рёбер в нем минимальна.

Построение всех остовных деревьев графа

Иногда возникает необходимость в построении полного списка остовных деревьев графа. Например, когда надо выбрать оптимальное дерево, а критерий, позволяющий осуществить такой отбор, является неочевидным или очень сложным, так что непосредственное решение задачи оптимизации, не использующее перечисление всех остовных деревьев, оказывается невозможным.

Число различных остовов полного связного неориентированного помеченного графа с n вершинами было найдено Кэли.

Теорема Кэли (Cayley, 1874). *Число деревьев на n вершинах равно* n^{n-2} $(n \geq 2)$.

Доказательство. Для доказательства теоремы рассмотрим *схему Прюффера* кодирования различных деревьев на *n* вершинах (*код Прюффера*).

Сопоставим каждому дереву на вершинах $\{1,2,...,n\}$ последовательность натуральных чисел $\{a_1, a_2, ..., a_{n-2}\}$ по следующему правилу. Выберем висячую вершину с минимальным номером и удалим ее, взяв в качестве a_1 номер соседней с ней вершины. Повторим операцию до тех пор, пока не останутся две вершины, соединенные ребром.

Пример

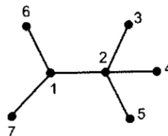


Рис. 65

Для дерева (рис. 65) последовательность Прюффера есть {2, 2, 2, 1, 1}.

Обратно. Поставим в соответствие последовательности $\{a_1, a_2, ..., a_{n-2}\}$, $a_i \le n$, дерево на n вершинах $\{1,2,...,n\}$ по следующему правилу. Найдем среди чисел $\{1,2,...,n\}$ наименьшее число κ, не встречающееся среди a_i. Соединим вершины с номерами κ и a_1 ребром. Удалим числа κ в списке $\{1,2,...,n\}$ и a_1 в списке $\{a_1, a_2, ..., a_{n-2}\}$. Повторим операцию до тех пор, пока в списке $\{1,2,...,n\}$ останется два числа, а все элементы последовательности $\{a_1, a_2, ..., a_{n-2}\}$ будут удалены. Соединим соответствующие вершины ребром.

Пример. Последовательность {2, 2, 2, 1, 1} генерирует дерево последовательным построением ребер (2, 3), (2, 4), (2, 5), (1, 2), (1, 6) и (1, 7).

Таким образом, установлена биекция между множеством всех деревьев на n вершинах и множеством всех последовательностей натуральных чисел вида $\{a_1, a_2, ..., a_{n-2}\}$, $a_i \le n$. Так как число таких последовательностей равно n^{n-2}, то таково и число искомых деревьев.

Число различных остовов неполного невзвешенного графа

Число различных остовов неполного связного неориентированного помеченного графа было найдено Кирхгофом.

Формула Кирхгофа

Определим матрицу степеней $D_{n,n}$ графа G следующим образом. Диагональные элементы $D_{i,i} = \deg(i)$ – степень i вершины, все недиагональные – $D_{i,j} = 0, i \ne j$.

Теорема Кирхгофа (Kirchhoff, 1847). *Число остовных деревьев в связном графе G равно любому алгебраическому дополнению матрицы $K = D - A$, где A – матрица смежности графа G.*

Пример. Дан граф G.

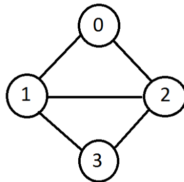


Рис. 66

Даны матрица смежности A, матрица степеней D и $K = D - A$ графа G.

$$A = \begin{bmatrix} 0,1,1,0 \\ 1,0,1,1 \\ 1,1,0,1 \\ 0,1,1,0 \end{bmatrix} \quad D = \begin{bmatrix} 2,0,0,0 \\ 0,3,0,0 \\ 0,0,3,0 \\ 0,0,0,2 \end{bmatrix} \quad K = D - A \;\; K = \begin{bmatrix} 2,-1,-1,\ 0 \\ -1,\ 3,-1,-1 \\ -1,-1,\ 3,-1 \\ 0,-1,-1,\ 2 \end{bmatrix}$$

Число остовных деревьев, равное алгебраическому дополнению матрицы K, $K_{i,j} = 8$, где i, j любые.

Число остовных деревьев полного графа $G(n = 4)$ равно 16.

Алгебраическое дополнение A_{ij} элемента a_{ij} матрицы A есть $A_{ij} =$ $= (-1)^{i+j} M_{ij}$, где M_{ij} есть дополнительный минор, т. е. определитель матрицы, получающейся из исходной матрицы A после удаления *i*-й строки и *j*-го столбца.

С-функция вычисления M_{ij} дана в приложении.

Алгоритм генерации различных остовов невзвешенного графа

Построение всех деревьев начинается с первой вершины. Множество всех деревьев одинаково независимо от выбора первой вершины. Отличается только порядок их перечисления. Для формирования очередного дерева используется только последнее. Все остовные деревья могут быть записаны в файл.

Первое дерево строится обычным поиском в ширину. Затем исключается последнее ребро и начинается попытка достроить дерево включением другого ребра. Если при этом дерево построить невозможно, исключается еще одно и т. д.

```
    static int m,edge,ktree,look,top,ntree,*tree,*Q,*u,*v;

    int CallSpanTree(int **A, int n)
{   int i, m, ktree;
    m=Edge(A,n); // кол ребер
    ktree=Kirchhoff(A,n);
    printf("ktree=%i",ktree);
    tree=(int *)calloc(m,4);
    u=(int *)calloc(m,4); v=(int *)calloc(m,4);
    Q=(int *)calloc(m,4);
```

```
    // initial
    for(i=0; i<n; i++)
    {tree[i]=0;u[i]=v[i]=0; };
    tree[0]=1;  // 0 вершина включена в дерево
    Q[0]=0;// в Q заносим первую вершину
    look=0;// занята
    top=1;  // свободна для записи
    edge=0; // количество ребер, включенных в дерево
    ntree=0; // номер сгенерированного дерева
    ntree=SpanTree(A,n,0,1); // первый вызов
    return ntree;
}   // CallSpanTree

// i – номер вершины, из которой выходит ребро;
// j – номер вершины, начиная с которой происходит поиск
// очередного ребра дерева

    int SpanTree(int **A,int n, int i, int k)
{   int j; // j=vertex
    if(look<top)
 {  j=k; // вершина начала просмотра
    while(j<n && edge<n-1)
 {  // просмотр ребер, выходящих из i вершины
    if(A[i][j] && !tree[j])
// есть ребро i-j, но j вершины еще нет в дереве
 {  // Включаем вершину j и ребро i-j в дерево
    tree[j]=1;
    edge++; // увеличиваем текущую переменную edge
// количества ребер, включенных в дерево
    u[edge]=i;v[edge]=j; // запоминаем
    Q[top]=j; top++; // записываем вершину j в дек
    SpanTree(A,n,i,j+1);
    // продолжаем построение дерева
    top--; tree[j]=0;
    edge--;// Исключаем ребро из дерева
 }  // if
    j++;
 }  // while
    if(edge==n-1) // печать построенного дерева
 {  ntree++;
    printf("ktree=%2i\n",ntree);
    printf("kij\n");
```

```
    for(int k=1; k<n; k++) // печать n-1 ребра
    printf("%3i%3i%3i\n",k,u[k],v[k]);
    printf("\n");
    return edge;
 }  // if edge
    // все ребра, выходящие из вершины с номером i,
    // просмотрены, но дерево не построено
    // переход к следующей вершине из Q
    // процесс идет, пока не будет построено дерево
    if(j==n)
 {  look++;
    i=Q[look];
    SpanTree(A,n,i,0);
    look--;
 }  //
 }  // if(look<top)
    return edge;
}   // SpanTree

    int Kirchhoff(int **A, int n)
{   int k,**D;
    D=DegreeMatrix(A,n); // матрица степеней вершин
    SubMatrix(D,A,n,n);  // D=D-A
    k=Det(D,n,0,0);  // алгебраическое дополнение "ребра"
    return k;
}   // Kirchhoff
```

Пример. Дан граф на рис. 67.

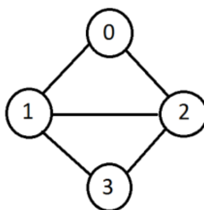


Рис. 67

tree	1		2		3		4		5		6		7		8	
edge	i	j	i	j	i	j	i	j	i	j	i	j	i	j	i	j
1	0	1	0	1	0	1	0	1	0	1	0	2	0	2	0	2
2	0	2	0	2	1	2	1	2	1	3	2	1	2	1	2	3
3	1	3	2	3	1	3	2	3	3	2	2	3	1	3	3	1

Минимальное остовное дерево

Minimal Spanning Tree, MST, or Shortest Spanning Tree, SST

Дан взвешенный неориентированный граф. Требуется найти такое дерево – подграф этого графа, которое бы соединяло все его вершины, обладая наименьшим весом, т. е. суммой весов рёбер из всех возможных вариантов таких деревьев. Такое дерево называется минимальным остовным деревом, или простым минимальным остовом, или каркасом.

Свойства минимального остова

Минимальный остов *уникален*, если веса всех рёбер различны. Иначе может существовать несколько минимальных остовов. Стандартные алгоритмы обычно получают один из возможных остовов.

Остов максимального веса аналогичен остову минимального веса. Для его поиска можно изменить знаки весов рёбер на противоположные и реализовать любой из алгоритм минимального остова. Также можно взять обратные величины.

Минимальный остов является также и *остовом с минимальным произведением* весов рёбер. Можно заменить веса всех рёбер на их логарифмы.

Алгоритм Прима

Prim, 1957

Вначале в остовное дерево включается одна вершина, которую можно выбрать произвольно. Затем находится ребро минимального веса, инцидентное этой вершине, и добавляется в дерево. В основном цикле процесса просматриваются все вершины, смежные всем вершинам, уже включенным в дерево, и ищется ребро минимального веса, которое включается в дерево. Процесс повторяется до тех пор, пока дерево не станет содержать все вершины, что тождественно включению $n-1$ ребер. В результате будет построено остовное минимальное дерево.

Если граф не связен, то необходимо строить остовные деревья для каждого связного подграфа.

Доказательство. Очевидно, что алгоритм Прима создает остовное дерево, т. к. добавление ребра между вершиной в дереве и вершиной вне дерева не может создать цикл. Открытым остается вопрос, почему именно это остовное дерево должно иметь наименьший вес из всех остовных деревьев? Существует достаточно много примеров других «жадных» эвристических алгоритмов, которые не дают точного решения. Алгоритм Прима достаточно редкий, который дает точное решение. Докажем это утверждение. Используем метод доказательства от противного. Допустим, что

существует граф, для которого алгоритм Прима не возвращает минимальное остовное дерево $T_{\min}$. Пусть уже построена часть минимального остовного дерева T_{prim}, но на некоторой итерации было принято неправильное решение и после добавления ребра (a,b) дерево больше не является минимальным.

В остовном дереве $T_{\min}$ должен быть путь от вершины a к вершине b. Этот путь должен содержать ребро (u,v), в котором вершина u находится в дереве $T_{\min}$, а вершина v – вне его. Вес этого ребра должен быть, по крайней мере, равен весу ребра (a,b), иначе алгоритм Прима выбрал бы ребро (u,v) до ребра (a,b), поскольку у него была такая возможность. Удалив из дерева $T_{\min}$ ребро (u,v) и вставив вместо него ребро (a,b), получим остовное дерево с весом не большим чем прежнее дерево. Это означает, что выбор алгоритмом Прима ребра (a,b) не был ошибочным. Следовательно, вес дерева, построенного алгоритмом Прима, минимален, т. е. создаваемое алгоритмом Прима остовное дерево минимальное.

```
    int Prim (int **G,int n,int v0)
    // G – взвешенная матрица смежности
    // n – количество вершин
    // v0 – начальная вершина
{   int i,j,k,inf,min,imin,jmin,*tree,*u,*v,*w,*p,**A;
    u=(int*)calloc(n,4); v=(int*)calloc(n,4);
    w=(int*)calloc(n,4); p=(int*)calloc(n,4);
    tree=(int*)calloc(n,4);
    inf=0x7FFFFFFF;
    // создается вспомогательная матрица A, в которой
    // 0 заменяются на inf
    A=(int**) CreateMatrix(n,n,4);
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
    {if(!G[i][j]) A[i][j]=inf; else A[i][j]=G[i][j];}
    // массив tree[v]=1 определяет
    // включение вершины v в дерево
    tree[v0]=1;
    // массивы u[k]=i v[k]=j определяют номера вершин i,j,
    // инцидентных ребру k; w[k] – вес ребра с номером k
    //p[i]=j определяет предыдущую вершину j для вершины i
```

```
   //  начальные значения:
   u[0]=-1; v[0]=v0; w[0]=inf; p[v0]=-1;
   printf("\n  i  j min\n");
   // основной цикл
   for(k=1; k<n; k++)  // step iter
 {  min=inf;
   // поиск вершины с минимальным ребром
   for(i=0; i<n; i++)
 {  if(tree[i])  // по всем вершинам дерева
   for(j=0; j<n; j++)  // по всем вершинам графа
  { if(A[i][j]<min &&  !tree[j] )
   {min=A[i][j]; imin=i; jmin=j; }
  } // j
 }  // i
   printf("%3i%3i%3i\n", imin,jmin,min);
   // включение вершины и ребра в дерево
   u[k]=imin; v[k]=jmin; w[k]=min;  p[jmin]=imin;
   tree[jmin]=1;
 }  // k
   return min;
}   // Prim
```

Время работы алгоритма существенно зависит от того, каким образом производится поиск очередного минимального ребра среди подходящих рёбер. Здесь могут быть разные подходы, приводящие к разным реализациям и, соответственно, разным асимптотикам.

Если искать каждый раз ребро простым просмотром среди всех возможных вариантов, то $O(m)$ будет асимптотическим временем выполнения для поиска ребра с наименьшим весом. Полная асимптотика составит в таком случае $O(nm)$, что в худшем случае есть $O(n^3)$.

Такую реализацию алгоритма можно улучшить, если просматривать каждый раз не все рёбра, а только по одному ребру из каждой уже выбранной вершины. Для чего можно отсортировать рёбра из каждой вершины в порядке возрастания весов и хранить индекс первого допустимого ребра. Если переопределять эти индексы при каждом добавлении ребра в дерево, суммарная асимптотика алгоритма будет $O(n^2+m)$. Однако предварительно потребуется выполнить сортировку всех рёбер, которая в лучшем случае составляет $O(m\log m)$, а для плотных графов – $O(n^2\log n)$.

Пример. Для графа G построено дерево T :

$$G=\begin{pmatrix}0&0&0&1&3\\0&0&3&5&4\\0&3&0&0&4\\1&5&0&0&2\\3&4&4&2&0\end{pmatrix} \qquad T=\begin{pmatrix}i&j&w\\0&3&1\\3&4&2\\4&1&4\\1&2&3\end{pmatrix}$$

Алгоритм Крускала

Kruskal, *1956*

Сначала все рёбра сортируются по весу в порядке возрастания (неубывания). Создается n поддеревьев, в каждом из которых по одной вершине. Эти поддеревья (компоненты будущего одного дерева) нумеруются, получая уникальный номер. Начинается основной процесс создания остова – объединения поддеревьев. Перебираются все рёбра от первого до последнего в порядке возрастания весов. Если у ребра инцидентные вершины (его концы) принадлежат разным поддеревьям, ребро соединяет эти поддеревья в более крупное поддерево. При этом номер этого поддерева для всех вершин в него входящих становится одинаковым. Если у ребра инцидентные вершины принадлежат одному поддереву, оно игнорируется, поскольку тем самым может создаться цикл, что исключено по определению дерева. По окончании перебора всех рёбер все вершины окажутся принадлежащими одному дереву и будут иметь одинаковый номер. Перебор можно закончить раньше, если количество включенных в дерево вершин $n-1$.

```
    int Kruskal(int **G,int n)
{   int i,j,k,m,x,y,q,inf,min,imin,jmin,edge,
    *tree,*u,*v,*w,*E,**A;
    m=Edge(G,n); // вычисляет количество ребер
    u=(int*)calloc(m,4); v=(int*)calloc(m,4);
    w=(int*)calloc(m,4); E=(int*)calloc(m,4);
    tree=(int*)calloc(n,4); inf=0x7FFFFFFF;
    // создается вспомогательная матрица A, в которой
    // 0 заменяются на inf
    A=(int**) CreateMatrix(n,n,4);
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
 {  if(G[i][j]==0) A[i][j]=inf; else A[i][j]=G[i][j]; }
    // сортировка ребер, инцидентные вершины которых
```

```
    // записываются в массивы u[m] и v[m], а веса в w[m]
    printf("\n  k  i  j min\n");
    for(k=0; k<m; k++)  // edge
  {  min=inf;
    for(i=0; i<n; i++)
    for(j=i+1; j<n; j++)
   { if(A[i][j]<min)
    {min=A[i][j]; imin=i; jmin=j;}
   } // i j
    if(min==inf) break; // finita
    u[k]=imin; v[k]=jmin; w[k]=min;
    printf("%3i%3i%3i%3i\n",k,imin,jmin,min);
    A[imin][jmin]=inf;
   } //  k
    // main loop
    edge=0; E[0]=0;  // число включенных ребер
    // tree[vertex] – массив, в котором индекс есть номер
    // вершины, а значение элемента есть номер поддерева
    for(i=0; i<n; i++) tree[i]=i;  // уникальные номера
    printf("\n  k  i  j min x  e\n");
    for(k=0; k<m; k++)  // step iter
  {  i=u[k]; j=v[k]; E[k]=-1;
    x=tree[i]; y=tree[j]; // номера поддеревьев
    if(x==y) goto mprint; // ребро не включается в дерево
    // из двух поддеревьев создается одно
    for(q=0; q<n; q++) // перенумерация вершин, входящих
    // в одно из поддеревьев - присваивается общий номер
    {if(tree[q]==y) tree[q]=x; }
    edge++; E[k]=edge; // количество включенных ребер
    mprint:
    printf("%3i%3i%3i%3i%3i%3i\n",k,i,j,w[k],x,edge);
    if(edge==n-1) break;
  } // k
    // дополнительная отладочная информация
    printf("\n  k  i  j min x  e\n");
    for(k=0; k<n; k++)
    printf("%3i%3i%3i%3i%3i%3i\n",
    k,u[k],v[k],w[k],tree[k],E[k]);
    return e;
}   // Kruskal
```

Пример. Дан граф: $n = 5$, $m = 7$ (рис. 68).

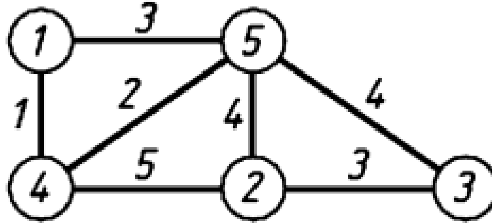


Рис. 68

$$W = \begin{pmatrix} 00013 \\ 00354 \\ 03004 \\ 15002 \\ 34420 \end{pmatrix}; \text{sort } S = \begin{pmatrix} e\,i\,j\,w \\ 0031 \\ 1342 \\ 2043 \\ 3123 \\ 4144 \\ 5244 \\ 6135 \end{pmatrix}; \text{tree } T = \begin{pmatrix} k & i & j & w & x & e \\ 0 & 0 & 3 & 1 & 1 & 1 \\ 1 & 3 & 4 & 2 & 1 & 2 \\ 2 & 0 & 4 & 3 & 1 & -1 \\ 3 & 1 & 2 & 3 & 1 & 3 \\ 4 & 1 & 4 & 4 & 1 & 4 \end{pmatrix}$$

Время выполнения определяется программой сортировки $O(mn^2)$ и основным циклом создания дерева $O(mn)$. Эти времена могут быть улучшены, используя более совершенные алгоритмы сортировки и адекватные структуры данных.

Алгоритм Борувки

Boruvka, 1926

Алгоритм создает минимальное остовное дерево во взвешенном неориентированном связном графе.

Вначале в будущее остовное дерево T – подграф графа G – включаются все вершины G, но не включаются ребра. Каждая вершина есть компонента связности, которая нумеруется. Можно считать T лесом, состоящим из деревьев – вершин.

Процесс добавления ребер в T.

1. Для каждой компоненты связности находим минимальное по весу ребро, которое связывает вершину из данной компоненты с вершиной, не принадлежащей данной компоненте.
2. Все ребра, которые хотя бы для одной компоненты связности оказались минимальными, добавляются в T .
3. Каждое добавление ребра приводит к соединению поддеревьев. При этом номер объединенного поддерева для всех вершин в него входящих становится одинаковым, равным одному из двух номеров.

Когда в T попадут $n-1$ ребер, процесс заканчивается, – минимальное остовное дерево построено.

Если в графе есть ребра равные по весу, то алгоритм выбирает первое встретившееся ребро.

Доказательство того, что алгоритм Борувки строит минимальное остовное дерево, полностью аналогично приведенному доказательству для алгоритма Прима.

```
    int Boruvka(int **G,int n)
    // G - взвешенная матрица смежности, n вершин
{   int i,j,k,m,q,inf,min,wmin,imin,jmin,iwmin,jwmin,e,
    *tree,*u,*v,*w,*del,**A;
    m=Edge(G,n);
    u=(int*)calloc(m,4); v=(int*)calloc(m,4);
    w=(int*)calloc(m,4); tree=(int*)calloc(n,4);
    del=(int*)calloc(n,4);
    inf=0x7FFFFFFF;
    A=(int**) CreateMatrix(n,n,4);
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
 {  if(G[i][j]==0) A[i][j]=inf; else A[i][j]=G[i][j]; }
    e=0;// number of added edges
    for(i=0; i<n; i++) tree[i]=i; // numbers of components
    printf("\n  k  q  i  j  w  e\n");
    edge: // loop on edges
    for(k=0; k<n; k++) // loop on tree - components
 {  if(del[k]) continue; // del[k] component is deleted
    wmin=inf;
    for(i=0; i<n; i++)
 {  if(tree[i]==k)
  { min=inf;
    for(j=0; j<n; j++)
```

```
   { if(tree[j]!=k)
     if(A[i][j]<min) {min=A[i][j]; imin=i; jmin=j;}
   } // j
     if(min<wmin) {wmin=min; iwmin=imin; jwmin=jmin;}
     // можно ввести один min
  }  // if
  }  // i
     if(wmin==inf) // there is no edge
     {wmin=-1; goto mprint; }
     q=tree[jwmin]; // or iwmin
     for(j=0; j<n; j++)
  {  if(tree[j]==q) tree[j]=k; del[q]=1;}
     // may be conversely k-q
     u[e]=iwmin; v[e]=jwmin; w[e]=wmin;
     e++;
     mprint:
     printf("%3i%3i%3i%3i%3i%3i\n",k,q,iwmin,jwmin,wmin,e);
     if(e==n-1) goto finita;
  }  // k
     goto edge;
     finita:
     printf("\n  k  i  j  w\n");
     for(k=0; k<n-1; k++)
   { i=u[k]; j=v[k]; q=w[k];
     printf("%3i%3i%3i%3i\n",k,i,j,q);
   } // k
     return e;
}    // Boruvka
```

Построение графа с заданным набором степеней вершин

Building of a graph with a given set of vertex degrees

Требуется построить неориентированный граф с n вершинами, для которого заданы степени вершин $d_k (k = 1, 2, ..., n)$.

Необходимым условием существования графа со степенями d_k есть $\sum_{k=1}^{n} d_k = N$, где по лемме о степенях N – четное число.

Это условие не является, однако, достаточным. Например, не существует графа с набором степеней (5, 1, 1, 1).

Вектор, составленный из степеней вершин графа G в порядке возрастания или убывания, называется *последовательностью*, или *вектором степеней*. Граф является *реализацией* своего вектора степеней.

Если вектор степеней имеет единственную реализацию с точностью до изоморфизма, то этот граф называют *униграф*. Все графы с числом вершин < 5 являются униграфами.

Регулярным, или *однородным*, или n-вершинным графом $R_{n,p}$ порядка p называют граф, в котором все степени вершин равны p.

Множество чисел, являющихся степенями вершин графа G, называют его *степенным множеством*. Оно отличается от вектора отсутствием совпадающих чисел.

Известно несколько критериев графичности векторов, т. е. возможности существования графа с заданным вектором степеней.

Теорема (Erdos, Gallai, 1960). *Вектор* $d = \{d_1, d_2, ..., d_n\}$, *в котором* $\{d_1 \le d_2 \le ... \le d_n\}$, *тогда и только тогда является графическим, когда для каждого* $k = 1, 2, ..., n-1$ *выполняется неравенство*

$$\sum_{i=1}^{k} d_i \le k(k-1) + \sum_{i=k+1}^{n} \min(k, d_i).$$

Теорема (Gavel, 1955; Hakimi, 1962). *Пусть дан вектор* $d = \{d_1, d_2, ..., d_n\}$, *в котором* $\{d_1 \le d_2 \le ... \le d_n\}$. *Если для какого-либо индекса* i, $1 \le i \le n$, *производный вектор* $d = \{d_1 - 1, ..., d_{i-1} - 1, d_{i+1}, ..., d_n\}$ *является графическим, то и вектор* d *является графиче-*

ским. Если вектор d *графический, то каждая последовательность* d $(i = 1,...,n)$ *является графической.*

Из теоремы непосредственно следует алгоритм построения графа с заданным вектором степеней $d = \{d_1, d_2, ..., d_n\}$.

1. Множество степеней упорядочивается по неубыванию (или невозрастанию).

2. Выбирается некоторая ведущая вершина d_{lead}. Она может иметь максимальное значение. Процесс завершается, когда все $d = 0$.

3. Ведущая вершина d_{lead} соединяется ребрами с вершинам, имеющими максимальные значения. Степень ведущей вершины полагается = 0, а у вершин, с которыми она была соединена, степень уменьшается на 1. Остальные остаются без изменения. Переход к шагу 2.

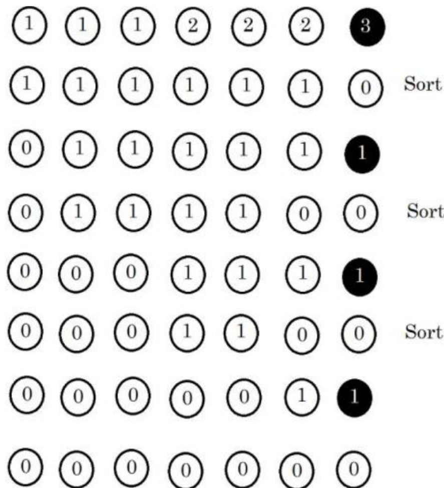
Граф может строиться либо на прямом проходе, либо на обратном, возвращаясь к исходному вектору степеней путём добавления последовательно по одной вершине.

Требуемый граф не существует, если на некотором шаге получается нереализуемая последовательность степеней, в частности, отрицательная степень.

Различным образом выбирая вершины, можно получить все неизоморфные реализации вектора степеней.

На рис. 69 показан пример построения для вектора степеней (1, 1, 1, 2, 2, 2, 3) по описанному алгоритму.

Черным цветом выделяется ведущая вершина.



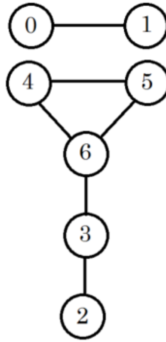


Рис. 69

Теорема. *Вектор степеней* $d = \{d_1, d_2, ..., d_n\}$ *может быть реализован связным графом тогда и только тогда, когда* $d_n > 0$ *и верно неравенство* $\sum_{i=1}^{n} d_i \geq 2(n-1)$.

Если указанные условия выполняются, то процедура, на каждом шаге которой ведущей выбирается вершина с минимальной степенью, приводит к построению связного графа.

Теорема. *Вектор степеней* $d = \{d_1, d_2, ..., d_n\}$ *может быть реализован деревом тогда и только тогда, когда выполнятся условие*

$$\sum_{i=1}^{n} d_i = 2(n-1).$$

Если указанные условия выполняются, то процедура, на каждом шаге которой ведущей является вершина с минимальной положительной меткой, приводит к построению дерева.

На рис. 70 представлен пример построения связной реализации вектора степеней (1, 1, 1, 2, 2, 2, 3) в виде дерева.

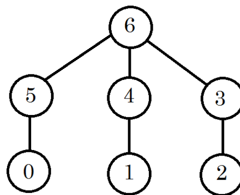


Рис. 70

Теорема (Чангфейзен, 1978). *Если существует реализация вектора степеней* $d = \{d_1, d_2, ..., d_n\}$, *имеющая гамильтонову цепь с началом в вершине степени* d_i, *то к такой реализации приведет процедура, в которой первый раз ведущей выбирается вершина степени* d_i, *а на каждом последующем – вершина с минимальной положительной степенью из вершин, соединенных с ведущей на предыдущем шаге.*

Теорема. *На последовательности* $D = \{d_i, i = 1, 2, ..., n\}$ *может быть построен граф тогда и только тогда, когда тем же свойством обладает последовательность* $\bar{D} = \{d_2 - 1, ..., d_{d_1+1} - 1, d_{d_1+2}, ..., d_n\}$.

Эта теорема дает алгоритм для построения графа с заданной последовательностью вершин, конечно, если такой граф существует.

Если же граф построить нельзя, алгоритм на некотором шаге прерывается.

Алгоритм. *Последовательность* $D = \{d_i, i = 1, 2, ..., n\}$, *где*

$$n - 1 \geq d_1 \geq d_2 \geq ... \geq d_n$$

реализует граф тогда и только тогда, когда приводимая последовательность шагов алгоритма приводит к последовательности, все элементы которой $= 0$.

Шаг 1. Вычисление последовательности $\bar{D}$ теоремы.

Шаг 2. Сортировка последовательности $\bar{D}$ по неубыванию (обозначается новая последовательность D_1).

Шаг 3. Вычисление последовательности $\bar{D}_1$ аналогично шагу 1.

Шаг 4. Сортировка последовательности $\bar{D}_1$ аналогично шагу 2 (обозначается новая последовательность D_2).

Шаг 5. Процесс продолжается до появления всех нулей – тогда можно построить граф. Либо когда для промежуточной последовательности с очевидностью можно построить граф. Либо появления хотя бы одного отрицательного элемента – тогда граф построить нельзя.

Шаг 6. Построение графа происходит на обратном ходе.

Пример. $D = \{5, 4, 4, 4, 3, 3, 2, 2\}$, $\bar{D} = \{3, 3, 3, 1, 1, 1, 2\}$.

Функция `DegreeExistConnect` определяет возможность существования и связности графа. Возвращает 0, если по заданному вектору степеней построить граф нельзя, 1 – если можно построить связный граф, –1 – если граф построить можно, но несвязный.

```
    int DegreeExistConnect(int a[], int n)
{   int i,j,k,s,min,smin,r,g,*d;
    InsertSort(a,0,n-1);
    d=(int*)calloc(n+1,4);
    for(i=0; i<n; i++)
    d[i+1]=a[i];
    g=1;
    for(k=1; k<=n-1; k++)
 {  s=0;
    for(i=1; i<=k; i++)
    s=s+d[i];
    // min
    smin=0;
    for(i=k+1; i<=n; i++)
  { if(k<d[i]) min=k; else min=d[i];
    smin=smin+min;
  } //i min
    r=k*(k-1)+smin;
    if(s>r) return 0; // there is no graph
 }  // k
    //     connect
    s=0;
    for(i=1; i<=n; i++)
    s=s+d[i];
    r=2*(n-1);
    y=s-r;
    if(y>=0) g=1; // there is connected graph
    else g=-1;     // there is no connected graph
    return g;
}   // DegreeExistConnect
```

Функции DegreeGraphA и DegreeGraphK реализуют алгоритм различным образом. DegreeGraphA строит граф на прямом проходе, а DegreeGraphK – на обратном.

```
    int DegreeGraphA(int d[],int n, int **&A)
{   int i,j,k,out=0,lead,y,q,xi,xj;
    int **B,*b,*x;
    A=(int**)CreateMatrix0(n,n,4);
    B=(int**)CreateMatrix0(n,n,4);
    b=(int*)calloc(n,4);
    x=(int*) calloc(n,4);
```

```
   for(i=0; i<n; i++)
   x[i]=i;
   InsertSort(d,x,0,n-1);
   out=-1;
   k=DegreeExistConnect(d,n);
   if(k==0)
   {printf("\n Graph is no exist\n"); return -1; }
   else {printf("\n Graph must exist\n");}
   for(k=0; k<n-1; k++)  // step
 {  j=n-1; // max
   lead=d[j]; // select lead  j=ilead
   if(lead==0) break;
   d[j]=0; xj=x[j];
   q=0;
   for(i=n-1;q<lead; i--)// i>=n-lead
 {  if(i!=j)
  { if(d[i]==0)
   {printf("j=%2i d[%2i]=%2iGraph is not\n",j,i,d[i]);
   goto not;
  }
   d[i]--;
   xi=x[i];
   A[xi][xj]=1; A[xj][xi]=1;
   q++;
 }  // if(i!=j)
 }  // i
   InsertSort(d,x,0,n-1);
   if(d[n-1]==0)break;
 }  // k
   not:
//--------------- A -------------------
   printf("\n A ");
   for(j=0; j<n; j++)
   printf("%2i",j); printf("\n\n");
   for(i=0; i<n; i++)
  { printf("%2i=",i);
   for(j=0; j<n; j++)
   printf("%2i",A[i][j]);
   printf("\n");
  } // i j
//---------------- B --------------------
```

```
    printf("\n B ");
    for(j=0; j<n; j++)
    printf("%2i",j); printf("\n\n");
    for(i=0; i<n; i++)
  { b[i]=0;
    printf("%2i=",i);
    for(j=0; j<n; j++)
  { if(A[i][j]==1)
  { B[i][b[i]]=j; b[i]++;
    printf("%2i",j);
  }
  }
    printf("\n");
  } // i
    metout0:
    return out;
}   // DegreeGraphA

    int DegreeGraphK(int *d,int n,int **&A)
{    int i,j,jj,k,ik,m,out,Dij;
    int **D,**B,*b;
    A=(int**)CreateMatrix0(n,n,4);
    B=(int**)CreateMatrix0(n,n,4);
    D=(int**)CreateMatrix0(n,n,4);
    b=(int*) calloc(n,4);
    InsertSort(d,0,n-1);
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
    D[i][j]=d[j];
    out=-1;
    for(i=1; i<n; i++)
 {  for(j=0; j<n; j++)
    D[i][j]=D[i-1][j];
    k=D[i-1][n-i];
    for(j=n-i-k; j<n-i; j++)
  { Dij=D[i][j]=D[i-1][j]-1;
    if(Dij<0) {out=0;
    printf("\n Graph is no exist");
    goto metout0; }
  } // for j
    InsertSort(D[i],0,j);
    if(k==0) {out=1; ik=i-1; goto metout1; }
```

```
 }  // for i
    metout1:
    for(i=ik-1; i>=0; i--)
 {  jj=n-i-1;
    for(j=jj-1; j>=0; j--)
  { if(D[i][j]-D[i+1][j]==1)
    {A[j][jj]=1; A[jj][j]=1;}
  } // j
 }  // i
//----------------- A -------------------
    printf("\n A ");
    for(j=0; j<n; j++)
    printf("%2i",j); printf("\n\n");
    for(i=0; i<n; i++)
  { printf("%2i=",i);
    for(j=0; j<n; j++)
    printf("%2i",A[i][j]);
    printf("\n");
  } // i j
//---------------- B ------------------
    printf("\n B ");
    for(j=0; j<n; j++)
    printf("%2i",j); printf("\n\n");
    for(i=0; i<n; i++)
  { b[i]=0;
    printf("%2i=",i);
    for(j=0; j<n; j++)
  { if(A[i][j]==1)
   {B[i][b[i]]=j; b[i]++;
    printf("%2i",j);
   }
  }
    printf("\n");
  } // i
    metout0:
    free(b); free(B); free(D);
    return out;
}   // DegreeGraphK
```

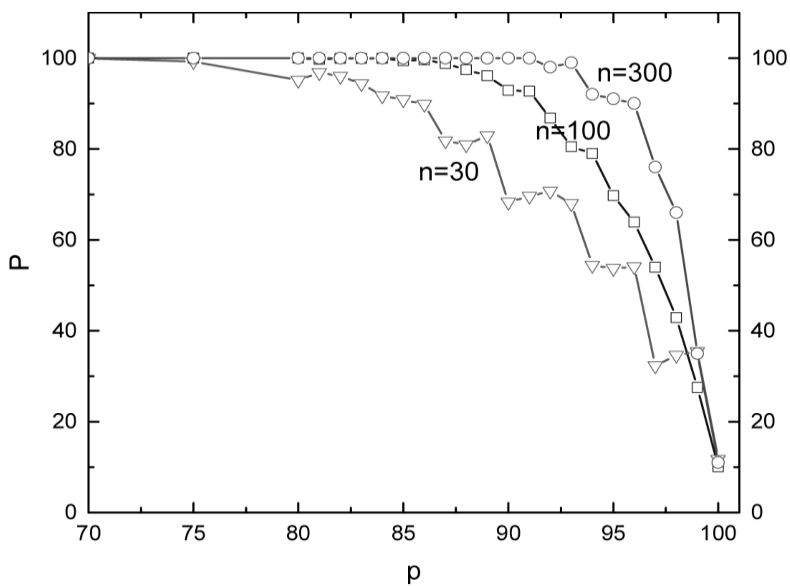


Рис. 71. Зависимость вероятности построения графа исходя из случайной последовательности степеней вершин

Случайные графы

Random graphs

Случайные графы полезны и удобны для решения проблем теории графов и множества прикладных задач, в частности анализа проблем в Интернете. Теория случайных графов была основана Эрдёшем и Реньи в 1959 г.

Популярны две модели, которые исходят из случайного выбора ребер или вершин.

Во-первых, случайный граф имеет n вершин, соединенных m ребрами, которые выбираются случайным образом из $n(n-1)/2$ возможных ребер. Всего существует $C^m_{n(n-1)/2}$ графов с n вершинами и m ребрами, которые образуют вероятностное пространство с равной вероятностью для каждой реализации.

Другое определение случайного графа называют также биноминальной моделью. В этом случае, имея n вершин, соединяем каждые две из них с вероятностью p. В конечном итоге, полученное количество ребер будет случайной величиной. Вероятность получить граф с помощью этого процесса составит $P(G) = p^m(1-p)^{n(n-1)/2}$.

Теория случайных графов изучает вероятностное пространство графов с n вершинами при $n \to \infty$. Многие свойства таких случайных графов могут быть получены с помощью случайного анализа. С этой точки зрения каждый граф обладает свойством Q, если при $n \to \infty$ вероятность выполнения Q равна 1. Среди вопросов по случайным графам некоторые имеют прямое отношение к сложным сетям. Например, является ли стандартный граф связным? Содержится ли в нем треугольник из соединенных вершин? Каким образом диаметр зависит от размеров графа?

Процесс создания случайного графа в литературе часто называют эволюцией: начиная с n изолированных вершин, граф последовательно развивается благодаря добавлению новых случайных ребер. Графы, полученные на разных стадиях этого процесса, соответствуют все большим и большим вероятностям p и в конце концов получаем полный граф, имеющий максимальное количество ребер $M = n(n-1)/2$ при $p=1$.

Многие важные свойства случайных графов начинают проявляться довольно неожиданно. Например, при заданной вероятности либо практически каждый граф обладает свойством Q, либо практически ни один граф им не обладает. Переход от вероятного к маловероятному событию может происходить при этом очень резко. Для многих из таких свойств

существует критическая вероятность $p_c(n)$. Если $p(n)$ возрастает медленнее, чем $p_c(n)$ при $n \to \infty$, то практически ни один граф не будет обладать свойством Q. Если $p(n)$ возрастает быстрее, чем $p_c(n)$, практически любой граф будет обладать свойством Q.

В теории случайных графов вероятность определена как функция от размера системы: p представляет собой дробь от наибольшего возможного количества вершин. Графы большего размера с тем же самым p будут содержать больше ребер, и в конце концов такие свойства, как появление циклов, скорее проявятся в них, чем в графах меньшего размера.

Генератор случайных графов (случайные ребра)

Рассматриваемая функция добавляет в граф произвольное (случайное) ребро путем генерации случайных пар целых чисел и интерпретации целых чисел как вершин графа.

Генератор неориентированного графа

```
#define _CRT_RAND_S
#include <stdio.h>
#include <stdlib.h>
   int **GGp(int n, int &m, double p,int *deg)
{  int i,j,M,e,err,**A; UINT r,R,mod;
   A=(int**)CreateMatrix0(n,n,4); // =0
   if(A==0) MessageBox(0,"GGp","A=0",MB_OK);
   M=n*(n-1)/2;
   mod=UINT_MAX;
   R=mod*p;
   m=0;
   for(i=0; i<n; i++)
 { for(j=0; j<i; j++)
  { err=rand_s(&r);
   if(r<R && A[i][j]==0)
   {A[i][j]=A[j][i]=1; m++;}
  } // j
 } // i
   out: printf("  n=%3i m=%3i p=%.2f\n",n,m,p);
   PrintMatrix(A,n,"GG");
   for(i=0; i<n; i++)
   for(j=0; j<n; j++)
  { if(A[i][j]) deg[i]++; }
```

```
    SelectSortMax(deg,0,n-1); // sort max..min
    for(i=0; i<n; i++)
    printf("%3i\n",deg[i]);
    printf("\n");
    return A;
}   // GGp
```

Генератор ориентированного графа

```
    int **GGdir(int n,int &m)
{   int i,j,r,**A;
    int *deg,*din,*dout;
    deg=(int*) calloc(n,4);
    din=(int*) calloc(n,4);
    dout=(int*)calloc(n,4);
    srand((unsigned)time(0));
    A=GGnm(n,m);
    for(i=0; i<n; i++)
    for(j=0; j<i; j++)
  { if(A[i][j]==1 && A[j][i]==1)
   {r=rand()%2;
    if(r==0) A[i][j]=-1; else A[j][i]=-1;}
  }
    printf(" n=%3i m=%3i\n",n,m);
    PrintMatrix(A,n,"Adir");
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
  { if(A[i][j]) deg[i]++;
    if(A[i][j]>0) dout[i]++;
    if(A[i][j]<0) din[i]++;
  }
    for(i=0; i<n; i++)
    printf("%3i%3i%3i\n",deg[i],dout[i],din[i]);
    printf("\n");
    return A;
}   // GGdir
```

Генератор неориентированного взвешенного графа

```
    int **GGw(int n,int m,int a,int b)
{   int i,j,r,**A; // e=edge
    GGnm(n,m); srand((unsigned)time(0));
    for(i=0; i<n; i++)
    for(j=0; j<i; j++)
```

```
 {  if(A[i][j]==1)
  { r=random(a,b); A[i][j]=A[j][i]=r;}
 }  // i
    printf(" n=%3i m=%3i\n",n,m);
    PrintMatrix("GG\n","%2i",A,n);
    return A;
}   // GGw
```

Генератор всех возможных графов

Использует другой алгоритм, генерирующий `i` и j.

```
    int **RandomGraph(int n,int m,int dir,int a,int b)
    // n=vertex   m=edge
{   int i,j,e,d,w,M,**A,*deg,*din,*dout;
    deg=(int*) calloc(n,4);
    din=(int*) calloc(n,4);
    dout=(int*)calloc(n,4);
    A=(int**)CreateMatrix0(n,n,4);
    M=n*(n-1)/2;
    if(m>M) m=M;
    srand((unsigned)time(0));
    if(!dir && !a && b==1)   // no direct no weight
 {  for(e=1; e<=m; e++)
  { iter:
    i=random(1,n-1); // 0 ..n-1
    j=random(0,i-1);
    if(A[i][j]) goto iter;  // djvu
    A[i][j]=A[j][i]=1;
  } // e
    for(i=0; i<n; i++)
    for(j=0; j<n; j++)
    deg[i]=deg[i]+A[i][j];
 }  //  a==0 && b==1
    if(!dir && (a || b!=1) ) // dir=0 no direct, weight!
 {  for(e=1; e<=m; e++)
  { iter2:
    i=random(1,n-1);
    j=random(0,i-1);
    if(A[i][j]) goto iter2;
    w=random(a,b);
    A[i][j]=A[j][i]=w;
  } // e
    for(i=0; i<n; i++)
```

```
   for(j=0; j<n; j++)
   if(A[i][j]) deg[i]++;
 } // a==0 && b>1
   if(dir) // direct! weight! = a<0 && b>=1
{  for(e=1; e<=m; e++)
 { iter3:
   i=random(1,n-1);
   j=random(0,i-1);
   if(A[i][j]) goto iter3;
   d=random(0,1); if(d==0) d=-1;
   w=random(a,b);
   A[i][j]=d*w; A[j][i]=-d*w;
 } // for e
   for(i=0; i<n; i++)
   for(j=0; j<n; j++)
 { if(A[i][j]) deg[i]++;
   if(A[i][j]>0) dout[i]++;
   if(A[i][j]<0) din[i]++;
 } // j i
}  // dir
   printf("\n");
   PrintMatrix(A,n,"RandomGraph");
   for(i=0; i<n; i++)
   printf("%3i%3i%3i\n",deg[i],dout[i],din[i]);
   printf("\n"); return A;
}  // RandomGraph
```

Связность, диаметр и радиус в случайном графе

Диаметр графа – наибольшее расстояние между двумя любыми его вершинами. Диаметр несвязного графа, например, образованного несколькими изолированными подграфами, обычно полагают как $\mathrm{diam} = \infty$. Однако можно переопределить его, как максимальный из диаметров его подграфов. Случайные графы имеют малый диаметр при условии, что вероятность p мала. Причина этого в том, что случайный граф кажется расширяющимся: с большой вероятностью количество вершин с расстоянием l от выбранной вершины пропорционально $\ln(n) / \ln(l)$, то есть оно логарифмически зависит только от количества вершин. Для большинства значений p, практически все графы имеют один и тот же диаметр. Это значит, что, когда рассматриваются графы

с n вершинами и вероятностью связности p, диаметры могут лишь незначительно отличаться, обычно находясь около определенного значения.

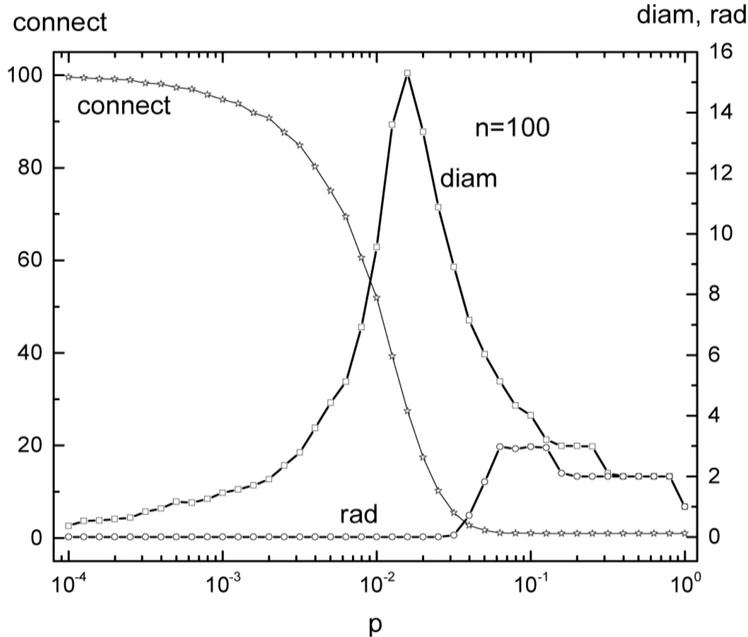


Рис. 72. Зависимость от вероятности p (количества ребер) количества связных подграфов, диаметра и радиуса

Распределение степеней в случайном графе

В случайном графе с вероятностью связности p степень d i-вершины следует биномиальному распределению с параметрами $n-1$ и p:

$$P(d) = C_{n-1}^{d} p^{d} (1-p)^{n-1-d} .$$

Эта вероятность представляет количество способов, которыми d ребер могут быть проведены из определенной вершины: вероятность для d ребер составляет p^d, вероятность отсутствия дополнительных ребер составляет $(1-p)^{n-1-d}$ и существует C_{n-1}^{d} эквивалентных способов выбора d конечных точек для этих ребер.

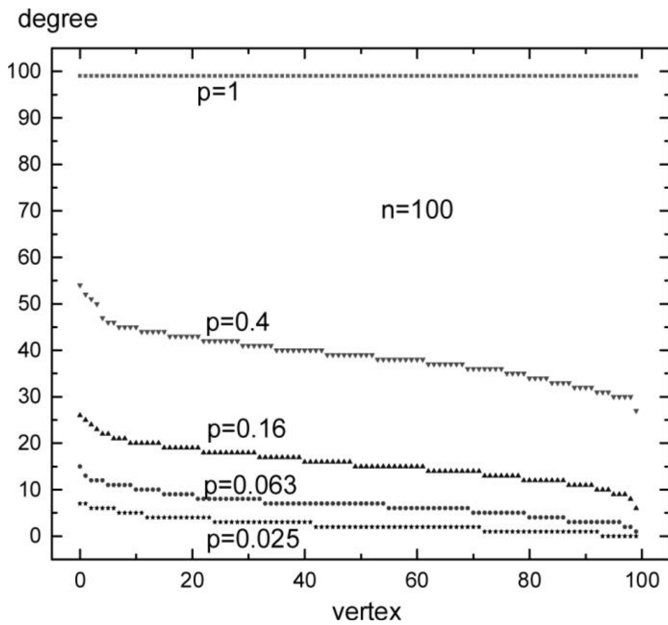


Рис. 73. Зависимость степеней вершин от вероятности p

На рис. 73 приведено распределение степеней по вершинам при некоторых вероятностях.

Клики, независимые множества, вершинные покрытия

Clique*, *Independent set*, *Covering set

Клика графа есть подмножество вершин графа, порождающее полный подграф и не являющийся подграфом никакого другого полного подграфа. В клике все вершины смежные.

Максимальная клика – клика с максимально возможным числом вершин среди клик графа.

Кликовое число (*clique number*) – число вершин в максимальной клике графа G, обозначается $\omega(G)$.

Независимое множество вершин графа есть множество несмежных вершин, не содержащееся в другом независимом множестве. Это множество вершин порождает пустой подграф.

Максимальное независимое множество содержит максимальное количество вершин среди всех независимых множеств.

Число независимости (*independent number*) *графа* – число вершин в максимальном независимом множестве графа G, обозначается $\alpha(G)$.

Независимое множество ребер графа есть множество попарно несмежных ребер, не содержащееся в другом независимом множестве.

Реберное число независимости (*edge independent number*) графа есть число ребер в максимальном независимом множестве графа G, обозначается $\alpha_e(G)$.

Вершинное покрытие графа есть множество его вершин, каждое ребро которого инцидентно хотя бы одной из этих вершин.

Число вершинного покрытия графа есть минимальное число вершин в вершинном покрытии графа G, обозначается $\beta(G)$.

Реберное покрытие графа есть множество его ребер, каждое ребро которого инцидентно хотя бы одной из вершин графа.

Число реберного покрытия графа есть минимальное число вершин в реберном покрытии графа G, обозначается $\beta_e(G)$.

Задача о независимом множестве преобразуется в задачу о клике и наоборот переходом от графа G к дополнительному графу G', так что $\alpha(G) = \omega(G')$.

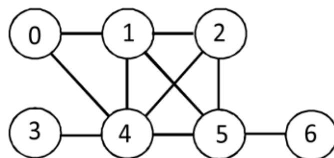
Для любого графа без изолированных вершин выполняются равенства $\alpha + \beta = \alpha_e + \beta_e = n$.

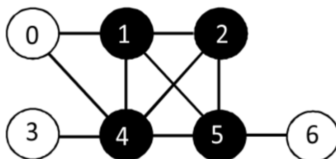
Множество U вершин графа $G=(V,E)$ является вершинным покрытием тогда и только тогда, когда $\bar{U}=V\setminus U$ есть независимое множество вершин. Минимальному независимому множеству вершин соответствует максимальное вершинное покрытие. Аналогичные утверждения справедливы и для реберного покрытия и независимого множества ребер.

Для полного графа K_n

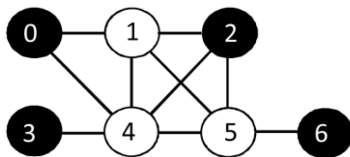
$\alpha_e = \lfloor n/2 \rfloor$, $\beta(K_n) = n-1$, $\beta_e(K_n) = \lfloor n/2 \rfloor$

Пример. Дан граф.





Максимальная клика – {1,2,4,5}



Максимальное независимое множество вершин – {0, 2, 3, 6}.
Минимальное вершинное покрытие – {1, 4, 5}

Рис. 74

Распространена также другая терминология. Клика называется *максимальной*, если она не содержится в клике с большим числом вершин, и *наибольшей*, если число вершин в ней наибольшее среди всех клик. Максимальное независимое множество называется *наибольшим*, если оно содержит наибольшее количество вершин. Независимое множество называется *максимальным*, если оно не является подмножеством другого независимого множества. Кроме того, обозначения $\alpha(G)$ и $\beta(G)$ противоположны.

Доминирующее множество

Dominating set

Доминирующее множество вершин графа $G=(V,E)$ есть множество вершин $S\subset V$ такое, что для каждой вершины v, не входящей в S, существует ребро, идущее из некоторой вершины множества S в вершину v.

Доминирующее множество называется минимальным, если нет другого доминирующего множества, содержащегося в нем.

Если F – семейство всех минимальных доминирующих множеств графа, то число $\beta(G)=\min\limits_{S\in F}|S|$ называется *числом доминирования* графа G, а множество $\overline{S}$, на котором этот минимум достигается, называется *наименьшим доминирующим множеством*. Иногда называют числом внешней устойчивости графа.

Пример. На рис. 75 приведены примеры графов.

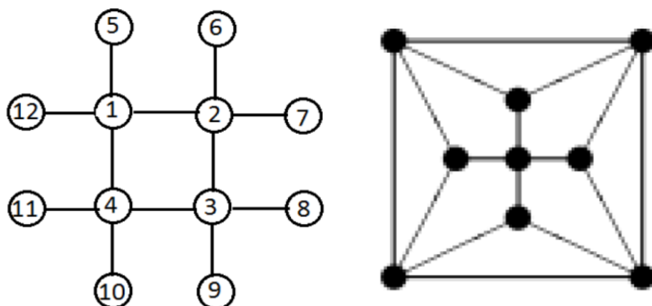


Рис. 75

G_{12}	G_9
Минимальные доминирующие множества графа: {1, 2, 3,4}, {5, 6, 7, 8, 9,10,11,12}, $\beta=4$.	$\mathrm{diam}=2,\ \beta=3$.

Независимое множество вершин графа является максимальным тогда и только тогда, когда оно является доминирующим.

Задача определения доминирующего множества вершин графа G эквивалентна задаче нахождения наименьшего множества столбцов матрицы смежности таких, что каждая строка матрицы содержит единицу хотя бы в одном из выбранных столбцов. Задачу о поиске наименьшего множества столбцов, покрывающего все строки матрицы, называют *задачей о наименьшем вершинном покрытии*.

Если введено дополнительное ограничение, чтобы любая пара столбцов не имела единиц в одних и тех же строках, то задачу называют *задачей о наименьшем разбиении*.

Обе эти задачи могут быть решены методом перебора. Решением является генерация всех подмножеств множества столбцов и проверка каждого подмножества на выполнение условий покрытия или разбиения. Оптимизация перебора связана с анализом особенностей исходных данных и поиском возможностей сокращения перебора.

Ядро графа

Для независимого и доминирующего множеств иногда используется другая терминология.

Множество внутренней устойчивости графа эквивалентно независимому множеству. *Максимальным множеством внутренней устойчивости* называется внутренне устойчивое множество, добавление любой вершины к которому, делает это множество неустойчивым. *Число внутренней устойчивости* α равно наибольшей мощности максимального внутренне устойчивого множества.

Множество внешней устойчивости графа эквивалентно доминирующему множеству. *Число внешней устойчивости* β равно наименьшей мощности из всех множеств внешней устойчивости.

Внутренне устойчивое множество является максимальным (не обязательно наибольшим) тогда и только тогда, когда оно внешне устойчиво. С другой стороны, внешне устойчивое множество не обязательно внутренне устойчиво.

Ядром графа называется подмножество множества вершин графа, если это множество является одновременно минимальным внешне устойчивым и максимальным внутренне устойчивым. Граф может не иметь ни одного ядра, а может обладать не одним ядром.

Если в графе есть ядро, то необходимым условием существования ядер в графе есть неравенство $\alpha \geq \beta$.

Процедура поиска внутренне и внешне устойчивых множеств простым перебором медленная. Для их поиска используется, в частности, *алгоритм Магу*.

Алгоритм Магу для определения множества внутренней устойчивости графа

Maghout, *1963*

Пусть дан граф $G=(V,E)$. Для него существует множество внутренней устойчивости U.

Введем булеву переменную x_i, которая определяется следующим образом: $x_i = 1$, если $x_i \in U$, т. е. вершина x_i принадлежит множеству внутренней устойчивости; $x_i = 0$, если $x_i \notin U$, т. е. вершина x_i не принадлежит множеству внутренней устойчивости.

Введем булеву переменную a_{ij}:

$a_{ij} = 1$, если *i*-я и *j*-я вершины смежные (есть ребро или дуга);

$a_{ij} = 0$, если *i*-я и *j*-я вершины несмежные.

Тогда определение внутренней устойчивости может быть представлено в следующем виде: $\forall\, i,j\ (x_i \in U, x_j \in V) \to x_j \notin U$, которое приводится к виду $\forall\, i,j\ (\, x_i \,\&\, a_{ij} \to \overline{x}_j) = 1$.

Применяя формулы равносильности, преобразуем

$$\forall\, i,j(\, x_i \,\&\, a_{ij} \to \overline{x}_j) = \mathop{\&}_{i=1}^{n} \mathop{\&}_{j=1}^{n} (x_i \,\&\, a_{ij} \to \overline{x}_j) =$$
$$= \mathop{\&}_{i=1}^{n} \mathop{\&}_{j=1}^{n} (\neg(x_i \,\&\, a_{ij}) \vee \overline{x}_j) = \mathop{\&}_{i=1}^{n} \mathop{\&}_{j=1}^{n} (\overline{x}_i \vee \overline{a}_{ij} \vee \overline{x}_j) = 1.$$

Рассмотрим уравнение

$$\mathop{\&}_{i=1}^{n} \mathop{\&}_{j=1}^{n} (\overline{x}_i \vee \overline{a}_{ij} \vee \overline{x}_j) = 1\,.$$

Если $a_{ij} = 0$, то это уравнение является тавтологией.

Если $a_{ij} = 1$, то это уравнение приводится к виду $\mathop{\&}_{a_{ij}=1} (\overline{x}_i \vee \overline{x}_j) = 1$.

Это уравнение служит обоснованием алгоритма Магу, который состоит из следующих этапов (используется матрица смежности).

1. По матрице смежности выписываются все парные дизъюнкции.
2. Выражение приводится к ДНФ.

3. Для любой элементарной конъюнкции выписываются недостающие элементы, которые и образуют множество внутренней устойчивости.

Пример. Дан граф на рис. 76.

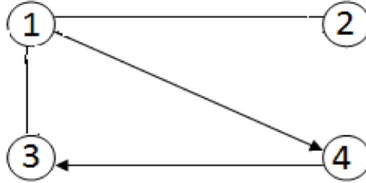


Рис. 76

Матрица смежности этого графа:

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

Для всех единиц выписываются парные дизъюнкции

$$(x_1 \vee x_2)(x_1 \vee x_3)(x_1 \vee x_4)(x_2 \vee x_1)(x_3 \vee x_1)(x_4 \vee x_3).$$

Это выражение приводится к ДНФ:

$$(x_1 \vee x_2)(x_1 \vee x_3)(x_1 \vee x_4)(x_2 \vee x_1)(x_3 \vee x_3)(x_4 \vee x_3) =$$
$$= (x_1 \vee x_2)(x_1 \vee x_3)(x_1 \vee x_4)(x_4 \vee x_3) = (x_1 \vee x_2 x_3)(x_4 \vee x_1 x_3) =$$
$$= x_1 x_4 \vee x_1 x_3 \vee x_2 x_3 x_4 \vee x_1 x_2 x_3 = x_1 x_3 \vee x_1 x_4 \vee x_2 x_3 x_4.$$

Для всех элементарных конъюнкций выписываются недостающие элементы, которые и образуют множества внутренней устойчивости

$$\{x_2, x_3\}, \{x_2, x_4\}, \{x_1\}.$$

Число внутренней устойчивости $\alpha = 2$.

Алгоритм Магу для определения множества внешней устойчивости

Пусть дан граф $G = (V, E)$. Для него существует множество внешней устойчивости W.

Вводятся булевы переменные x_i и a_{ij} по тому же правилу, что и для алгоритма Магу определения множества внутренней устойчивости.

Тогда определение множества внешней устойчивости запишется следующим образом: $\mathop{\&}\limits_{i=1}^{n}(x_i \vee \bigvee\limits_{j=1}^{n} a_{ij} \,\&\, x_j) = 1$.

При $a_{ij} = 1$ справедливо уравнение $\mathop{\&}\limits_{i=1}^{n}(x_i \vee \bigvee\limits_{a_{ij}} x_j) = 1$.

Это уравнение лежит в основе алгоритма Магу, который состоит из следующих этапов.

1. Матрица смежности дополняется 1 по главной диагонали.
2. Для каждой строки выписываются дизъюнкции.
3. Выражение приводится к ДНФ.
4. Все вершины, входящие в элементарную конъюнкцию, образуют множество внешней устойчивости.

Пример. Определить множество внешней устойчивости для графа, представленного на рис. 76.

Матрица смежности с 1 по главной диагонали

$$A = \begin{pmatrix} 1\,1\,1\,1 \\ 1\,1\,0\,0 \\ 1\,0\,1\,0 \\ 0\,0\,1\,1 \end{pmatrix}.$$

Для каждой строки выписываются дизъюнкции

$$(x_1 \vee x_2 \vee x_3 \vee x_4)(x_1 \vee x_2)(x_1 \vee x_3)(x_3 \vee x_4).$$

Это выражение приводится к ДНФ и минимизируется:

$$\begin{aligned}&(x_1 \vee x_2 \vee x_3 \vee x_4)(x_1 \vee x_2)(x_1 \vee x_3)(x_3 \vee x_4) = \\ &= (x_1 \vee x_2)(x_1 \vee x_3)(x_3 \vee x_4) = (x_1 \vee x_2x_3)(x_3 \vee x_4) = \\ &= x_1x_3 \vee x_1x_4 \vee x_2x_3 \vee x_2x_3x_4 = x_1x_3 \vee x_1x_3 \vee x_2x_3.\end{aligned}$$

Откуда получим множества внутренней устойчивости

$$\{x_1, x_3\}, \{x_1, x_4\}, \{x_2, x_3\}.$$

Число внешней устойчивости $\beta = 2$.

Граф на рис. 76 имеет ядро $\{x_2, x_3\}$.

Классические алгоритмы решения задачи нахождения клик

Maximum clique problem

Задачи нахождения клик (независимых множеств) относятся к классу NP переборных задач и очень трудны для алгоритмического решения. Эта задача имеет разнообразные приложения. В современных приложениях, таких как структурный анализ химических соединений и геномных баз данных, автоматизация проектирования сложных технических изделий и систем, кластеризация данных, поиск клик приходится осуществлять в разреженных графах очень большой размерности – до миллиона вершин. Поэтому востребованы эффективные алгоритмы, позволяющие находить точное решение задачи для таких графов за приемлемое время.

Многие из известных алгоритмов поиска требуют формирования списка всех клик (независимых множеств) данного графа исходя из списка всех клик возрастающей последовательности его подграфов. Поэтому эти алгоритмы применимы лишь при небольшом числе клик, т. е. только для малых графов.

Для нахождения точного решения задачи существует большое число алгоритмов, время выполнения которых экспоненциально зависит от числа вершин и ребер входного графа. Наиболее известными из них являются алгоритм Брона–Кербоша и алгоритм Уилфа.

Рекурсивный алгоритм Уилфа предназначен для нахождения точного решения задачи поиска независимых множеств. Краткое его изложение следующее. Для любой произвольно избранной вершины графа существуют два вида независимых множеств. Одни множества включают избранную вершину, другие – не включают избранную вершину. Исходя из этого, исходную задачу можно расщепить на две подзадачи (как уменьшенные копии исходной), соответствующие двум случаям. Формируемое независимое множество $N(v)$ содержит избранную вершину v. Тогда все вершины из $N(v)$ уже не могут входить в это независимое множество, и дальнейшее его увеличение надо осуществлять в графе $G(V \setminus N[v])$. Формируемое независимое множество не содержит вершину v. Дальнейшее увеличение этого множества следует продолжать в графе $G(V \setminus \{v\})$.

Броном и Кербошем (Bron, Kerboshf, 1971) был разработан алгоритм поиска клик и максимальных независимых множеств, значительно упрощающий процедуру перебора вершин графа. Алгоритм Брона–Кербоша является рекурсивной процедурой, которая последовательно увеличивает клику вершин. Поскольку граф с n вершинами может содержать до d^n максимальных клик (здесь $d \approx \text{const}$), то алгоритм Брона–Кербоша сопоставим по трудоемкости с процедурой полного перебора. В этом алгорит-

ме не нужно запоминать генерируемые независимые множества для проверки их на максимальность путем сравнения с ранее сформированными множествами. Этот алгоритм был опробован для большого числа графов и было установлено, что время необходимое для построения максимальных независимых множеств почти постоянно и не зависит от величины графа. По-видимому, данный алгоритм является одним из наиболее эффективных. Известны различные модификации данного алгоритма. Приведем сначала краткое описание этого алгоритма.

Алгоритм Брона–Кербоша для нахождения максимального независимого множества графа

В алгоритме использованы следующие обозначения:

S_k – независимое множество вершин графа G на k шаге;

Q_k^- – подмножество вершин графа G, которые уже использовались в процессе поиска для расширения множества;

Q_k^+ – подмножество вершин графа G, которые еще не использовались для расширения;

$\Gamma(v)$ – множество вершин графа G, смежных с вершиной v.

Начальная установка

Шаг 1. Пусть $S_0 = Q_0^- = \varnothing$, $Q_0^+ = V$, $k = 0$.

Прямой шаг

Шаг 2. Выбрать вершину $v_{ik} \in Q_k^+$ (v_{ik} – i вершина на k шаге) и сформировать множества

$$S_{k+1} = S_k \cup v_{ik};$$

$$Q_{k+1}^- = Q_k^- \setminus \Gamma(v_{ik});$$

$$Q_{k+1}^+ = Q_k^+ \setminus (\Gamma(v_{ik}) \cup (v_{ik}));$$

оставляя Q_k^- и Q_k^+ неизменными. Положить $k = k + 1$.

Проверка

Шаг 3. Если удовлетворяется условие $\exists v \in Q_k^-$ такое, что $\Gamma(v) \cap Q_k^+ = \varnothing$, то перейти к шагу 5, иначе – к шагу 4.

Шаг 4. Если $Q_k^+ = Q_k^- = \varnothing$, печатать максимальное независимое множество S_k и перейти к шагу 5. Иначе перейти к шагу 2.

Шаг возвращения

Шаг 5. Положить $k = k - 1$. Удалить v_{ik} из S_{k+1}, чтобы получить S_k. Исправить Q_k^- и Q_k^+, удалив вершину v_{ik} из Q_k^+, добавив ее к Q_k^-.

Если $k = 0$ и $Q_k^+ = \varnothing$, то остановиться.

К этому моменту будут напечатаны все максимальные независимые множества. Иначе перейти к шагу 3.

Реализация алгоритма псевдокодом представима в следующем виде.

```
while K != 0 or M != 0:
if K != 0:
v = K.first
push M, K, P, v
M = M + {v}
K = K - G(v) - {v}
P = P - G(v)
else:
if P == 0:
print M
pop v, P, K, M
K = K - {v}
P = P + {v}
```

M – текущее независимое множество;

K – множество кандидатов (вершин, способных образовать клику). На начальном этапе это множество содержит все вершины графа;

P – множество отсеянных вершин, которые не могут более добавляться в M;

v – просматриваемая вершина;

G(u) – множество вершин, смежных с вершиной u.

Алгоритм реализован в двух версиях, второй из них имеет большую скорость.

Алгоритм версии 1

Три множества играют определяющую роль в алгоритме.

sub (от *subgraph*) есть множество, которое расширяется новой вершиной или сокращается на одну вершину вдоль ветви дерева решения. Вершины, которые попадают в *sub*, связаны со всеми вершинами в *sub* и вычисляются рекурсивно.

cand (от *candidates*) есть множество всех вершин, которые будут в определенное время включены в существующее множество *sub*.

use (от *used*) есть множество всех вершин, которые уже были включены и использовались в *sub* на более ранней стадии, а затем исключены.

Алгоритм реализуется рекурсивной функцией, которая работает с описанными множествами. Процесс вычисления состоит из следующих пяти шагов.

1. Выбор кандидата – вершины для включения в *sub*.

2. Добавление отобранного кандидата в *sub*.

3. Создание *новых* множеств *cand* (*candidates*) и *use* из *старых* множеств, удаляя из этих множеств все вершины, не связанные с отобранным кандидатом, и сохраняя старые множества в итерации.

4. Вызов функции с этими новыми множествами.

5. По возвращению из этой функции удалить отобранного кандидата из *sub* и добавить его к старому множеству *use*.

Необходимое условие для окончания процедуры создания искомого подмножества (клики) состоит в том, что множество *candidates* пусто, иначе *sub* мог все еще быть расширен.

Это условие, однако, недостаточно, потому что, если теперь *use* непусто, то из определения *use* следует, что существующая конфигурация *sub* уже содержалась в другой конфигурации и поэтому не максимальна. Можно считать, что *sub* – клика, как только и *use*, и *candidates* пусты.

Если на некоторой стадии *use* содержит вершину, связанную со всеми вершинами в *candidates*, можно предсказать, что дальнейшие расширения (дальнейший выбор кандидатов) никогда не будут приводить к удалению (в шаге 3) той особой вершины от последующих конфигураций *use*.

Оба множества (*use* и *cand*) записываются в один одномерный массив с расположением: *use* / *cand* и с индексами: *use*[1:ne], *cand*[ne+1:ce]

Следующие свойства, очевидно, должны выполняться.

1. $ne <= ce$.

2. $ne = ce$: пустое множество *candidates* = 0.

3. $ne = 0$: пустое множество *use* = 0.

4. $ce = 0$: пустое (*use*) и пустое (*candidates*) = клика найдена.

Если отобранный кандидат находится в положении $ne+1$ множества, то вторая часть шага 5 осуществляется как $ne = ne + 1$.

В версии 1 используется элемент с индексом $ne + 1$ как выбранный кандидат. Эта стратегия никогда не приводит к нарушению заданного порядка элементов в *candidates*.

Алгоритм оптимальной версии 2

Эта версия не выбирает заданного кандидата в положении $ne+1$, а подбирает оптимального кандидата, обозначаемого *s* (от *select*). Чтобы иметь возможность закончить шаг 5 так, как описано выше, элементы *s* и $ne+1$ обменяются, как только выбор имел место. Этот обмен не затрагива-

ет множества *candidates* , так как нет неявного упорядочивания. Однако выбор влияет на порядок, в котором клики, в конечном счете, генерируются.

Стратегия выбора определяется условием минимизации числа итераций, т. е. шагов 1–5. Повторения заканчиваются, как только связанное условие достигнуто. Это условие сформулировано так: существует вершина в *use*, связанная со всеми вершинами в *candidates* . Желательно, чтобы существование такой вершины появилось на самой ранней стадии.

Предположим, что с каждой вершиной в *use* связан счетчик *count*, считающий число кандидатов, с которыми эта вершина связана (число несоединений). Перемещение отобранного кандидата в *use* (это происходит после вызова функции) уменьшает на 1 все счетчики вершин в *use*, с которыми она несмежна и вводит новый собственный *count*. Никакой *count* никогда не уменьшается больше чем на 1. Всякий раз, когда *count* = 0, предельное условие достигнуто.

Можно не устанавливать *count* для всех вершин, а выбрать одну особую вершину в *use*, называемую *fix*. Если выбирать кандидатов, несоединенных с этой фиксированной вершиной, то *count* фиксированной вершины будет уменьшен на 1 при каждом повторении. Никакой другой *count* не может уменьшаться быстрее. Если, для начала, у фиксированной вершины есть самый низкий *count*, никакой другой *count* не может достигнуть нуля быстрее, пока *counts* для вершин, недавно добавленных к *use*, не могут быть меньшими.

При очередном вызове функци эта фиксированная вершина должна быть определена или из *use*, или от оригинальных *кандидатов*. С этого момента отслеживается этот *count*, уменьшая его для каждого следующего выбора, как только выбираются несмежные вершины.

Граф задается матрицей смежности в симметричной форме с диагональными элементами = 1.

```
    static const int n=1024; //number of vertexes in graph
    static int **A,n,All[n+1],Sub[n+1],subsize;

    int BronKerboshf(int **A,int n,int *Old,int ne,int ce)
{   int New[n+1], nod, fixp, newne, newce, i, j, count,
    pos=-1, p, s, v, minnod,ret;
    // The latter set of integers is local in scope
    // but need not be declared recursively
    minnod=ce; i=nod=0;
    // Determine each counter value and look for minimum:
    for(i=i+1; i<=ce && minnod!=0; i++)
 {  p=Old[i]; count=0; j=ne;
    // Count disconnections:
```

```
   for(j=j+1; j<=ce && count < minnod; j++)
 {  q=Old[j];
    if(!A[p][q])
  { count++;
    // Save pos (position) of potential candidate:
    pos=j ;
  } // if
 }  // for j
    // Test new minimum
    if(count < minnod)
 {  fixp=p; minnod=count;
    if(i<=ne) s=pos; else
  { s=i; PREINCR: nod=1; }
 }  // if new min
 }  // i
    // if fixed point initially chosen from candidates
//then number of disconnections will be preincreased by 1
    //      back track cycle
    for(nod=minnod+nod; nod>=1; nod--)
{   // interchange:
    p=Old[s]; Old[s]=Old[ne+1];
    v=Old[ne+1]=p;
    // fill new set Use:
    newne=i=0;
    for(i=i+1; i<ne; i++)
    if(A[v][Old[i]])
    { newne++; New[newne] = Old[i];};
    // fill new set cand:
    newce=newne;  i=ne+1;
    for(i=i+1; i<=ce; i++)
    if(A[v][Old[i]])
    { newce++; New[newce] = Old[i]; };
    // Add to Sub:
    k++; Sub[k]=v;  // v=select vertex
    //      output of clique
    if(newce==0)
{   printf("subsize=%2i\n", subsize);
    for(int j=1; j<=subsize; j++)
    printf("%3i",Sub[j]);
```

```
    printf("\n");
 }
    else // newce!=0
    if(newne<newce) // cand!=0
    ret=BronKerboshf(A,n,New,newne,newce);
    //     back step
    // remove v from Sub:
    subsize--;
    // add v to Use:
    ne++;
    if(nod>1)
  { // select candidate disconnected to fixed point:
    s=ne;
    while(A[fixp][Old[++s]]);
  } // selection
 }  // backtrack cycle
    return 0;
}   // BronKerboshf

    int CallBronKerboshf(int **A, int n)
{   int r;
    for(int k=0; k<=n; k++) All[k]=k;
    subsize=0;
    r=BronKerboshf(A,n,All,0,n);
    return r;
}   // CallBronKerboshf
```

Пример

На рис. 77 и 78 показаны результаты расчетов для случайных графов программой – алгоритмом Брона–Кербоша.

На рис. 77 показаны зависимости (приведенные в подписи к рисунку) в функции количества вершин $n = 10 - 10^3$ при заданной вероятности $p = 20\%$ от максимального количества возможных ребер.

Время расчетов при $n > 30$: $t \approx 10^{-8} \times n^3$ с, количество клик: $\sim 10^{-3} \times n^3$, размер клик k не превышает 8, но максимальное количество клик обладает только $k_{\max}$ вершинами.

На рис. 78 показаны зависимости (приведенные в подписи к рисунку) в функции плотности ребер с вероятностью $p = (1 - 100)\%$ при $n = 50$.

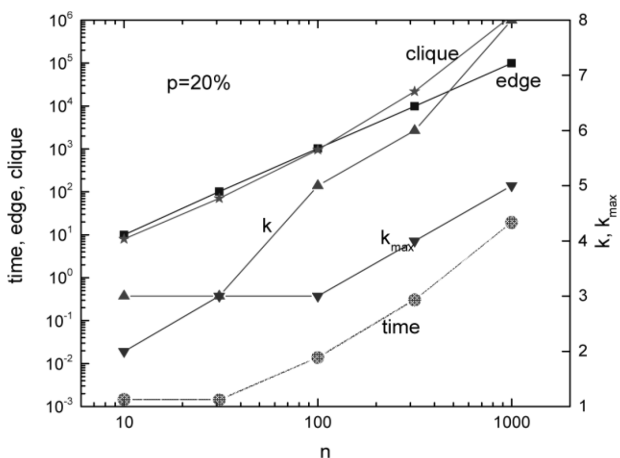


Рис. 77. Зависимости для программы – алгоритма Брона–Кербоша:
time – время работы, clique – количество клик, edge – количество ребер, k – максимально количество вершин в клике, $k_{\max}$ – количество клик, которое максимально с этим количеством вершин

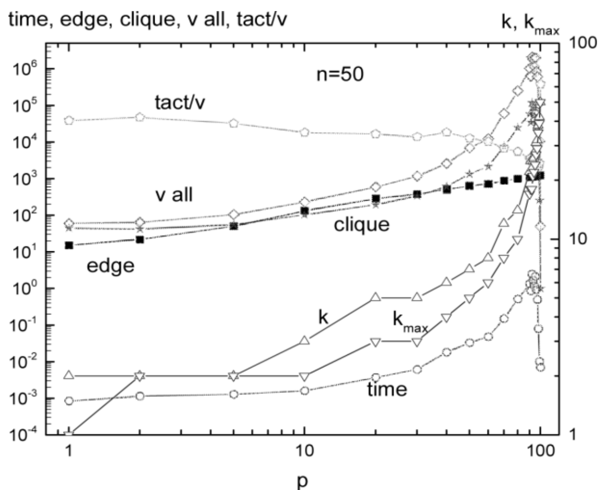


Рис. 78. Зависимости для программы – алгоритма Брона–Кербоша:
time – время работы, clique – количество клик, edge – количество ребер, k – количество вершин в клике, $k_{\max}$ – количество клик, которое максимально с этим количеством вершин, v all – полное количество вершин в кликах (вершины учитываются в разных кликах), tact/v – количество тактов, затрачиваемых на обработку каждой вершины в каждой клике

Количество клик возрастает с увеличением p, достигая величины $\approx 10^5$ при $p = 92\%$, затем резко уменьшается и, естественно, становится равным 1 при $p = 100\%$. Также ведет себя и время расчетов: изменяясь от 10^{-3} до 1 с. Величины k и $k_{\max}$ не слишком отличаются. Полное количество вершин, входящих в разные клики, достигает величины $2 \cdot 10^6$. Количество тактов, затрачиваемых на обработку каждой вершины в каждой клике, для всех p велико: $(2-50) \cdot 10^3$. За исключением $p = 100\%$ с одной кликой и её $n = 50$ вершинами, когда на обработку затрачивается $\approx 4 \cdot 10^5$ тактов.

Клики графа Муна–Мозера

Графы Муна–Мозера с $n = d \times m$ вершинами содержат d^m клик, каждая из которых содержит только m вершин. Поэтому перечисление клик возможно только при относительно малых значениях m и n.

Для $d = 4, m = 3$ генерируемые 64-мя кликами программой Брона–Кербоша показаны ниже. Таблица демонстрирует неочевидный способ выбора вершин в кликах.

1 = 1 5 9	17 = 2 5 9	33 = 3 5 9	49 = 4 5 9
2 = 1 5 10	18 = 2 5 10	34 = 3 5 10	50 = 4 5 10
3 = 1 5 11	19 = 2 5 11	35 = 3 5 11	51 = 4 5 11
4 = 1 5 12	20 = 2 5 12	36 = 3 5 12	52 = 4 5 12
5 = 1 6 9	21 = 2 6 9	37 = 3 6 9	53 = 4 6 9
6 = 1 6 10	22 = 2 6 10	38 = 3 6 10	54 = 4 6 10
7 = 1 6 11	23 = 2 6 11	39 = 3 6 11	55 = 4 6 11
8 = 1 6 12	24 = 2 6 12	40 = 3 6 12	56 = 4 6 12
9 = 1 7 9	25 = 2 7 9	41 = 3 7 9	57 = 4 7 9
10 = 1 7 10	26 = 2 7 10	42 = 3 7 10	58 = 4 7 10
11 = 1 7 11	27 = 2 7 11	43 = 3 7 11	59 = 4 7 11
12 = 1 7 12	28 = 2 7 12	44 = 3 7 12	60 = 4 7 12
13 = 1 8 9	29 = 2 8 9	45 = 3 8 9	61 = 4 8 9
14 = 1 8 10	30 = 2 8 10	46 = 3 8 10	62 = 4 8 10
15 = 1 8 11	31 = 2 8 11	47 = 3 8 11	63 = 4 8 11
16 = 1 8 12	32 = 2 8 12	48 = 3 8 12	64 = 4 8 12

Эвристические алгоритмы поиска всех клик

Можно использовать следующий эвристический алгоритм поиска всех клик. Вершины графа сортируются по степеням. Поиск клики начинается с максимальной степени. Затем клика «удаляется» – удаляются ребра и соответственно понижаются степени.

Приводятся разные варианты С-функций.

```
    int DeleteClique(int **A,int *D,int n,int *Q,int q)
{   int k,v,d;
    for(int i=0; i<q; i++)
 {  v=Q[i]; // clique vertexes
    d=0; k=0;
    for(int j=0; j<n; i++) // deg of clique vertexes
  { if(D[j] && A[v][j]==1) d++; }
    if(d<=q)  // = free vertexes & +1
 { D[v]=0; k++; }  //k= del number
}   // for i
    return k;
}   // DeleteClique

    int DeleteClique(int *D, int *Q, int q)
{   int k, v, d;
    k=0;  // vertex number
    for(int i=0; i<q; i++)
 {  v=Q[i]; // clique vertexes
    if(D[v]==q-1)
    {D[v]=0; k++;}
 }  // for i
    for(int i=0; i<q; i++)
 {  v=Q[i]; // clique vertexes
    if(D[v])
    D[v]=D[v]-k;
 }  // for i
    return k;
}   // DeleteClique

    int DeleteVertex1(int **A,int *D,int n,int m)
    // if deg<=m D=0
{   int k=0,d;
    Degree(A,n,D,1);
    for(int i=0; i<n; i++)
    d=D[i];
    for(int i=0; i<n; i++)
```

```
 {  d=D[i];
    if(!D[i]) continue;
    if(d<=m)
  { D[i]=0; k++;}
 }  // for i
    return k;  // del number
}   // DeleteVertex1

    int DeleteVertex(int **A,int *D,int *Y,int n,int m)
    // m=max del deg n=true dim
{   int k,ret,h=0,dv=0; int X[32],DD[32]; // ret
    for(k=0; k<n; k++)  //
 {  h=DeleteVertex1(A,D,n,m); // h= full del number
    if(h==0) break;
    dv=dv+h;  // del number
    printf("k=%i\n",k);
    PrintVector(D,n,"i  Degree before",0);
 }  // for k
    for(int i=0; i<n; i++)
    DD[i]=D[i];
    PrintVector(D,n,"i  Degree after",0);
    return k;
}   // DeleteVertex

    int SearchClique1(int **A,int n,int *P, int p,
    int *Q,int &q) // y=1 clique; y=0 undependent set
{   int i,j,u,v;  static int count=0;
    count++;
    //-- main loop --
    Q[0]=P[0];  // first cand
    q=1;
    for(i=1; i<p; i++)  // all cand but v by deg>1
 {  v=P[i];  // select cand
    for(j=0; j<q; j++)  // from clique
  { u=Q[j];
    if(v==u) continue;
    if(A[v][u]==0) break;
  } // for j
    if(j==q) // all coinside
    { Q[q]=v; q++; }
 }  // i
    return q;  // size
}   // SearchClique1
```

```
    int SearchCliqueAll(int **A,int n)
    // y=1 clique; y=0 independent set
{   int i,j,u,v,m,p,q,ret,D[nn],Y[nn],P[nn],Q[nn];
    m=1;
    //-- init --
    Degree(A,n,D,0); // first
    PrintVector(D,n,"Degree",0);
    ret=DeleteVertex(A,D,Y,n,m);
    //-- main loop --
    for(k=0; ; k++) // k = clique number
 {  j=0;
    for(i=0; i<n; i++)
  { if(D[i]) P[j++]=i; } // det cand=P[0..p-1]
    p=j;  // cand number
    SearchClique1(A,n,P,p,Q,q);
    printf(" k=%i q=%i\n",k,q);
    PrintVector(Q,q," Clique",0);
    ret=DeleteClique(D,Q,q);
    ret=DeleteVertex(A,D,Y,n,m);
    ret=0;
    for(i=0; i<n; i++)
    if(D[i]) { ret=1; break; }
    if(!ret) break;
 }  // k
    return k;
}   // SearchCliqueAll
```

Еще один эвристический алгоритм. Вычисляются степени вершин графа, и они упорядочиваются по убыванию. Вычисляется количество вершин, которые могли бы войти в клику. И это есть максимально возможная клика. Производится перебор исходя из генерируемых сочетаний. Если клика найдена, то степени её вершин уменьшаются и пересчитываются возможные претенденты на включение в клику. Таким образом, сначала находятся максимальные клики, а затем клики с меньшим числом вершин.

```
    int CliqueCnm2(int **A,int n)
{   inti,j,k,m,r,top,debug=0,max=0,q,w,dmin,count,
    min1,max1,x,y,clin,maxq,clique,a[dim],d[dim],t[dim],
    v[dim],u[dim],Q[dim],C[dim][dim],**B; __int64 cnm;
    printf("   CliqueCnm2\n");
```

```
// copy A->B
B=(int **)CreateMatrix(n,n,4);
for(i=0; i<n; i++)
for(j=0; j<n; j++) B[i][j]=A[i][j];
for(i=0; i<256; i++)
for(j=0; j<64; j++)
C[i][j]=-1;
dmin=0;
min=inf; max=0;
for(i=0; i<n; i++)
{  d[i]=0;
for(j=0; j<n; j++)
if(A[i][j] && d[j]) d[i]++;
deg=d[i];
if(deg>max) max=deg;  // max degree
if(deg<min) min=deg;
m=m+deg;// 1/2 number of edge
}  // i
q=0;  // max deg
for(i=max; i>=2; i--)  // num v in clique max+1
 { k=0;
for(j=0; j<n; j++)
{if(d[j]>=i) k++; }
if(k>i && q==0) q=i+1;
 } // i
maxq=q;
printf(" maxd=%3i mind=%3i maxcliq=%3i\n",
max, min, maxq);
for(i=0; i<n; i++)
printf(" i=%3i d=%3i\n",i,d[i]);
clique=-1;  // complete number +1 of clique
for(q=maxq;q>=3;q--)// q=value of vertex in clique
{  Q[q]=0;   // number of clique for q vertex
rew:
w=0; k=0;
for(j=0; j<n; j++)
if(d[j]>=q-1) {w++;
t[k]=j; k++; } // t=true vertex
if(w<q) continue;
for(i=0; i<w; i++) a[i]=i; // for combi
```

```
    for(k=0; a[0]!=w-q; k++)   // k = nomer of clique
 {  if(k) Combi0(a,w,q); // generation of combination
    for(x=0; x<q; x++)   // select vertex in cnm
   { i=t[a[x]]; // true vertex i=t[x];
    for(y=x+1; y<q; y++)
    {j=t[a[y]];   // true vertex j=t[y];
    if(!B[i][j]) goto net; // net clique
    } // y j
}   //  x i
    Q[q]++;   // number clique with c q vertex
    clique++; // complete number clique
    printf("w=%2i q=%2i k=%2i cl=%2i j=",
    w,q,k,clique);
    top=0;
    for(x=0; x<q; x++)  // select vertex in combination
   { i=t[a[x]]; d[i]=d[i]-q+1;  // true i=t[x];
    C[clique][top]=i; top++; // write all vertex in clique
    for(y=x+1; y<q; y++)
    {j=t[a[y]]; d[j]=d[j]-q+1; }  // y j
   } // x i
    for(top=0; C[clique][top]>=0; top++)
    printf("%3i", C[clique][top]); printf("\n");
    printf("a=");
    for(i=0; i<q; i++)
    printf(" %3i",a[i]);
    printf("\n");
    goto rew;
    net:;
 }  // k = nomer clique in
    printf("w=%2i q=%2i Q=%4i cnm=%4i\n",w,q,Q[q],cnm);
 }  // q =number of vertex in clique
    printf("  Q[q]\n");
    for(q=maxq; q>=3; q--)
    printf(" q=%3i Q=%3i\n",q,Q[q]);
    printf("\n");
    printf("CliqueCnm - la finita\n");
    return  clique;  //
}   // CliqueCnm2
```

Паросочетания и реберные покрытия

Matching and edge covering

Паросочетанием (*matching*) в графе называется (независимое) множество ребер, попарно не имеющих общих вершин.

Паросочетание с наибольшим числом ребер графа G обозначается $\pi(G)$.

Реберным покрытием графа называется такое множество ребер, в котором любая вершина графа инцидентна хотя бы одному из этих ребер.

Наименьшее число ребер в реберном покрытии графа G обозначается $\rho(G)$. Естественно, реберное покрытие существует только для графов без изолированных вершин.

Паросочетание называется *максимальным*, если оно не содержится в паросочетании с бóльшим числом ребер, и *наибольшим*, если число ребер в нем наибольшее.

Паросочетание называется *совершенным*, если оно одновременно является и реберным покрытием.

Если в графе есть совершенное паросочетание, то оно, естественно, является наименьшим реберным покрытием.

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

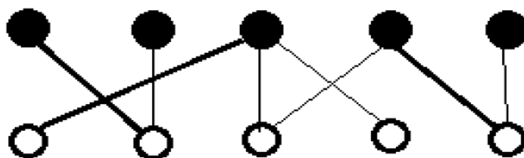


Рис. 79. Двудольный граф и паросочетание в нем (выделено жирными ребрами)

Определение паросочетания похоже на определение независимого множества вершин. Паросочетание иногда так и называют – *независимое множество ребер*. Эта аналогия усиливается еще 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 $\alpha(G) + \beta(G) = n$.

Минимальное реберное покрытие является наименьшим тогда и только тогда, когда оно содержит наибольшее паросочетание.

Максимальное паросочетание является наибольшим тогда и только тогда, когда оно содержится в наименьшем реберном покрытии.

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 рассматриваемых ниже чередующихся цепей. Отметим, что эта проблема похожа на задачи об эйлеровом и гамильтоновом циклах – реберный вариант эффективно решается, а вершинный является *NP*-полным.

Метод увеличивающих цепей

Пусть G – граф, и M – некоторое паросочетание в нем. Ребра паросочетания называются сильными, остальные ребра графа – слабыми. Вершина называется свободной, если она не принадлежит ребру паросочетания.

На рис. 80 слева показан граф и в нем выделены ребра паросочетания M = {(1, 2), (6, 8), (9, 10), (3, 7)}. Вершины 4 и 5 – свободные. Заметим, что к этому паросочетанию нельзя добавить ни одного ребра, то есть оно максимальное.

Однако оно не является наибольшим. В этом легко убедиться, если рассмотреть путь 5, 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 и состоит суть метода увеличивающих цепей.

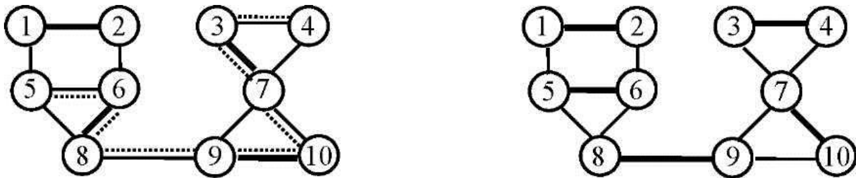


Рис. 80

Чередующейся цепью (*alternating chain*) относительно данного паросочетания называется простой путь, в котором чередуются сильные и слабые ребра. То есть за сильным ребром следует слабое, за слабым – сильное.

Чередующаяся цепь называется *увеличивающей*, если она соединяет две свободные вершины. Если M – паросочетание, P – увеличивающая цепь относительно M, то легко видеть, что $M \otimes P$ – тоже паросочетание и $| M \otimes P | = | M | + 1$.

Паросочетание является *наибольшим* тогда и только тогда, когда относительно него нет увеличивающих цепей.

Для решения задачи о паросочетании необходимо находить увеличивающие цепи или убеждаться, что таких цепей нет. Тогда, начиная с любого паросочетания (например, с пустого множества ребер), можно строить паросочетания со всё увеличивающимся количеством ребер до тех пор, пока не получится такое, относительно которого нет увеличивающих цепей. Оно и будет наибольшим.

Известны эффективные алгоритмы, которые ищут увеличивающие цепи для произвольных графов. Здесь изложены основные идеи простого алгоритма, решающего эту задачу для двудольных весьма популярных графов.

Алгоритм поиска паросочетания в двудольных графах

Пусть $G = (A, B, E)$ – двудольный граф с долями A и B; M – паросочетание в G. Всякая увеличивающая цепь, если такая имеется, соединяет вершину из множества A с вершиной из множества B.

Зафиксируем некоторую свободную вершину $a \in A$. Требуется найти увеличивающий путь, начинающийся в a, либо убедиться 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 вершинами x и a. Так как граф двудольный, то 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 неоднозначно, но любое такое дерево в двудольном графе обладает следующим свойством.

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

Для решения задачи необходимо построить дерево достижимости. Для этого можно использовать поиск в ширину из вершины a и адаптировать его к этой задаче. Отличие от типичного поиска в ширину состоит в том, что открываемые вершины группируются на четные и нечетные. Для четных вершин исследуются инцидентные им слабые ребра, а для нечетных – сильные. Через $\Gamma(x)$ обозначается множество вершин, смежных с вершиной x; Q – очередь, используемая при поиске в ширину. Если вер-

шина x не является свободной, то есть инцидентна некоторому сильному ребру, то другая вершина этого ребра обозначается через $p(x)$.

Если очередная рассматриваемая вершина x оказывается свободной, нет необходимости доводить построение дерева до конца. В этом случае путь между вершинами a и x в дереве является увеличивающим путем и можно его использовать для построения большего паросочетания. После этого снова выбирается свободная вершина (если такая еще есть) и строится дерево достижимости.

Если дерево построено и в нем нет других свободных вершин, кроме корня, то нужно выбрать другую свободную вершину и построить дерево достижимости для нее (конечно, если в графе больше двух свободных вершин). Тогда вершины первого дерева можно удалить из графа.

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

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.

Построение деревьев достижимости и удаление их вершин из графа продолжается до тех пор, пока не будет выполнено одно из двух условий.

1. Если найден увеличивающий путь, паросочетание увеличивается, граф восстанавливается, и вновь начинается поиск увеличивающего пути.
2. Если останется граф с не более чем одной свободной вершиной, имеющееся 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 есть $O(m)$. Число ребер в паросочетании не может превышать $n/2$, поэтому общая сложность алгоритма $O(mn)$.

Планарность

Planarity

Плоский граф (*plane graph*) изображается на плоскости так, что никакие два ребра не пересекаются, не имеют общих точек, кроме инцидентной им обоим вершины.

Планарный граф (*planar graph*) есть граф, изоморфный плоскому.

Граф *допускает плоскую укладку* (*layout*), если его можно изобразить как плоский.

Примеры плоских и планарных графов показаны на рис. 81.

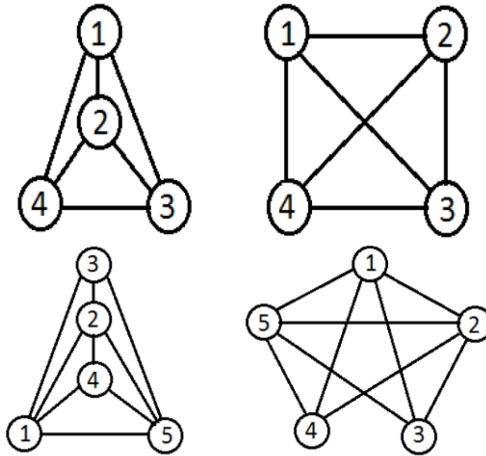


Рис. 81. Плоские и планарные графы

Примеры классических не планарных графов показаны на рис. 82: полный K_5 и полный двудольный граф $K_{3,3}$.

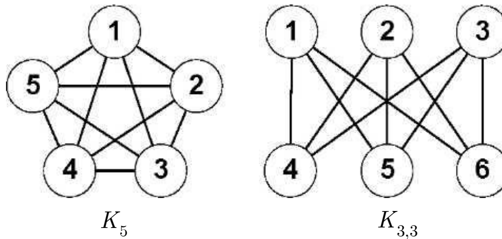


Рис. 82

В плоском представлении графа вводится понятие грани (*face*), которое определяется различными способами.

- Грань есть геометрическое место точек, каждые две из которых могут быть соединены жордановой кривой, не пересекающей ребра графа.
- Грань есть область, ограниченная ребрами в плоском графе и не содержащая внутри себя вершин и ребер.
- Грань есть часть плоскости, ограниченная простым циклом и не содержащая внутри себя других циклов.

Плоский граф имеет одну и притом единственную, неограниченную грань, называемую *внешней* (или *бесконечной*) гранью; остальные грани называются *внутренними*. Число граней плоского графа G обозначается $f(G)$, или $r(G)$.

Простой цикл, ограничивающий грань f, называется его *границей*, а число ребер этого цикла называется *степенью* грани и обозначается обычно $d(f)$. Две грани называются соседними, если они имеют общее ребро. Общее ребро двух граней иногда называют *перегородкой*.

Формула Эйлера

В связном плоском графе справедливо соотношение

$$n - m + f = 2\,.$$

Доказательство. Преобразуем граф в дерево последовательным разрывом всех простых циклов (удалением одного ребра) так, чтобы граф оставался связным; при этом число вершин не изменяется. При таком удалении одного ребра число граней уменьшается на 1, поскольку либо исчезает один простой цикл, либо два цикла превращаются в один цикл. Следовательно, выполняется условие $m - f = \mathrm{const}$. В результате всех разрывов получим дерево с количеством ребер $m_{\mathrm{tree}} = n - 1$, $f_{\mathrm{tree}} = 1$, а $n_{\mathrm{tree}} = n$. Последовательность преобразований элементарная:

$$n - m + f = n_{\mathrm{tree}} - m_{\mathrm{tree}} + f_{\mathrm{tree}} = n - n + 1 + 1 = 2\,.$$

Из формулы *Эйлера* вытекает много интересных следствий.

Следствие 1. *Граф K_5 непланарный.*

Доказательство. Предположим, что K_5 планарный. Рассмотрим его плоское представление. По формуле *Эйлера* должно быть $f = m - n + 2 = 7$, поскольку $n = 5$ и $m = 10$. Так как граф – полный,

то каждая грань в его плоском представлении ограничена точно тремя ребрами: $d(f)=3$. Тогда $21=3f=\sum_f d(f)=2m=20$, что дает противоречие.

Следствие 2. *Граф* $K_{3,3}$ *непланарный.*

Доказательство. Предположим, что $K_{3,3}$ планарный. Рассмотрим его плоское представление. По формуле *Эйлера* должно быть $f=m-n+2=5$, поскольку $n=6$ и $m=9$. Так как граф двудолен, то он не имеет циклов нечетной длины и следовательно степень любой грани $d(f)\geq 4$. Тогда $20=4f\leq \sum_f d(f)=2m=18$, что дает противоречие.

Следствие 3. *Каждый планарный граф с* $n\geq 4$ *вершинами имеет, по крайней мере, четыре вершины со степенями* $d\leq 5$.

При $n>3$ $m\leq 3n-6$. Действительно, каждая грань ограничена, по крайней мере, тремя ребрами, каждое ребро ограничивает не более двух граней, поэтому $3r\leq 2m$. Справедливо простое преобразование

$$2=n-m+r\leq n-m+2m/3,\ 3n-3m+2m\geq 6,$$

или $m\leq 3n-6$.

Плоский граф, у которого все грани, включая внешнюю, являются треугольниками, называют *плоской триангуляцией*.

Плоский максимальный граф есть граф, который перестает быть плоским при добавлении любого ребра.

Граф – плоский максимальный тогда и только тогда, когда он представляет собой плоскую триангуляцию.

Отсюда следует, что всякий плоский граф является остовным подграфом некоторой плоской триангуляции. Для всякого максимального планарного графа выполняется равенство $m=3n-6$ (вместо условия $\leq$).

Критерии планарности

Определение. Операция подразбиения ребра $e=(u,v)$ есть замена ребра e на новую вершину w и два новых ребра (u,w), (w,v). Два графа называются *гомеоморфными*, если они могут быть получены из одного графа подразбиением его ребер.

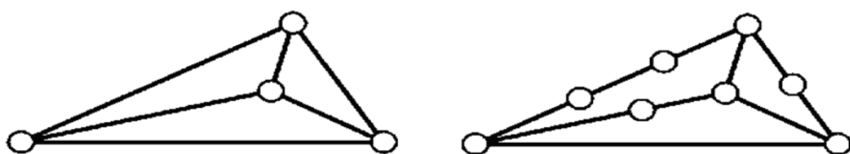


Рис. 83. Граф K_4 и гомеоморфный ему

Критерий Понтрягина–Куратовского (~1920 г.)

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

Критерий Вагнера

Определение. Граф G называется *стягиваемым* к графу H, если H можно получить из G стягиванием некоторых его ребер.

Граф планарен тогда и только тогда, когда в нем нет подграфов, стягиваемых к графам K_5 и $K_{3,3}$.

Хотя классических критических графов всего два, при росте числа вершин почти все графы непланарные.

Пример. Граф Петерсена (рис. 84) непланарный. Это доказывается следующей нетривиальной процедурой. Если убрать ребра (3, 4) и (7, 10); определить вершины 1, 8 и 9 в одну долю; 2, 5, 6 в другую, после чего появится $K_{3,3}$.

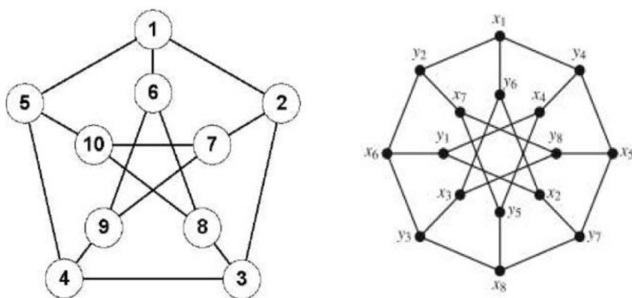


Рис. 84. Граф Петерсена G_{10} и обобщенный граф Петерсена G_{16}

Приведенные критерии очень трудны для проверки. Найти в графе подграфы K_5 и/или $K_{3,3}$ в общем случае – нетривиальная задача, и программно, по-видимому, не реализована. Данные критерии представляют лишь теоретический интерес, поэтому для практической работы используются более конструктивные критерии.

Существует несколько алгоритмов проверки на планарность, основные идеи первого из них приводятся ниже. Сама весьма громоздкая программа не приводится.

Алгоритм укладки графа на плоскости

Впервые алгоритм плоской укладки графа придумали Хопкрофт и Тарьян (J. Hopcroft & R. Tarjan, 1974). Алгоритм известен также под именем γ-*алгоритм* (*гамма-алгоритм*).

Стратегия алгоритма

1. В графе G найти цикл C и разместить его на плоскости в виде простой замкнутой кривой.

2. Оставшуюся часть графа $G \setminus C$ *разложить* на не пересекающиеся по ребрам пути.

3. Разместить каждый из выделенных путей либо целиком внутри C, либо целиком вне C. Если размещение найдено, то граф – планарный, иначе – нет. Трудность этой операции в том, что при размещении путей требуется выбирать какую-то сторону C (либо внутреннюю, либо внешнюю). При этом выборе необходимо обеспечить правильный выбор области размещения на ранней стадии, чтобы не устранять возможности размещения последующих путей. Иначе это могло бы привести к неверному заключению о непланарности планарного графа.

Пусть выделено k путей после двух шагов. Поскольку существует два способа размещения путей, существует 2^k вариантов с определением пересечения путей. Чтобы избежать простого перебора вводится определенный порядок путей, чтобы размещать очередной путь без нарушения планарности.

Производится преобразование графа, используя поиск в глубину. Вводится новая нумерация вершин, которая совпадает с порядком их просмотра при поиске в глубину. Для каждого ребра устанавливается определенный вес. Исходный граф преобразуется в ориентированный: его новое представление – списки смежностей, упорядоченные по неубыванию весов ребер.

Второй шаг алгоритма – разложение графа на один цикл и множество путей. При этом используется поиск в глубину для графа, но уже в новом представлении. Просмотр начинается с первой вершины. Сначала вычисляется цикл p_c обычным образом: встретив вершину графа, из которой проводится ребро в уже просмотренную вершину.

Далее находятся пути, которые основаны на стратегии просмотра в глубину и понятии пути. Каждый путь состоит из последовательности ребер дерева просмотра в глубину, за которой следует одно обратное реб-

ро. Если такого обратного ребра нет, происходит возврат к предыдущей вершине на последнем пути.

Такое разложение неединственно, но количество путей определено единственным образом и равно $m-n+1$. Любой из путей p_i имеет только две общие вершины с подграфом $p_c \cup p_1 \cup ... \cup p_{i-1}$. Поскольку списки связей отсортированы по неубыванию, то пути выстраиваются из конкретной вершины в порядке номеров вершин, в которых заканчиваются обратные ребра.

Третий шаг алгоритма – выделение сегментов. Фактически сегменты выделяются в процессе построения путей, но с целью достижения доступности изложения этот процесс выделяется в отдельный шаг. *Сегментом* графа относительно цикла *C* называют либо одно обратное ребро, не принадлежащее циклу, но у которого обе вершины принадлежат циклу, либо подграф, состоящий из ребра (v,w) дерева T ($v \in C, \mathrm{but}\ w \notin C$) ориентированного поддерева, имеющего корнем w, и все обратные ребра из этого поддерева. Вершину v называют *базовой вершиной сегмента*.

Очевидно, что все пути, принадлежащие одному сегменту, следует размещать вместе или внутри цикла *C*, или за пределами цикла *C*, и в этом заключается смысл объединения путей в сегменты.

Четвертый шаг алгоритма – укладка сегментов. Он начинается с размещения цикла *C*, который укладывается в виде простой замкнутой кривой, делящей плоскость на две области – внешнюю и внутреннюю. Затем наступает очередь сегментов, укладываемых в порядке их генерации. Выбирается одна из областей плоскости (внутренняя или внешняя) и делается попытка разместить путь *p*. Для этого необходимо установить *совместимость* размещения *p* с ранее размещенными путями. Если совместимость установлена, то *p* размещается и осуществляется переход к укладке следующих путей.

Если совместимости нет, то ранее размещенные во внутренней области сегменты переносятся во внешнюю область, а размещенные во внешней переносятся во внутреннюю и т. д. В том случае, когда не удается разместить *p* и после переносов, делается заключение о непланарности графа. В случае удачи – если совместимость есть – осуществляется переход к размещению следующего пути из сегмента. Аналогичные действия выполняются для всех оставшихся (неразмещенных) сегментов. Из сказанного следует, что требуется критерий совместимости – можно или нет разместить в конкретной области относительно цикла *C* первый путь сегмента.

Критерий совместимости заключается в том, что берется первый путь очередного сегмента и определяется, есть ли вхождение обратных ребер предыдущего сегмента в цикл на интервале (v_j, v_i). Если нет, то укладка

во внутренней области возможна, и проверяется на каждом новом варианте размещения выполнения критерия совместимости.

Рекурсивность логики алгоритма укладки. Предположим, что размещен первый путь $p=(s,\dots,g)$ текущего сегмента S с какой-то стороны цикла C. Теперь нужно определить, можно ли разместить оставшуюся часть $S \setminus p$ этого сегмента, не нарушая планарности уже уложенной части графа. Сделать это можно в том случае, если планарен подграф $\bar{G}=S \bigcup C$. Таким образом, возникает задача определения планарности подграфа $\bar{G}$. Поставленную задачу можно решить, применяя к $\bar{G}$ рекурсивно критерий совместимости. В этой рекурсии путь p вместе с ребрами цикла C от g (конечная вершина пути p) до s (начальная вершина пути p) будет служить начальным циклом $\bar{C}$ в $\bar{G}$. Удаление этого цикла (вместе с его начальной и конечной вершинами) из $\bar{G}$ позволит разбить оставшийся орграф $\bar{G} \setminus \bar{C}$ на сегменты (уже относительно $\bar{C}$), которые в свою очередь также обрабатываются рекурсивно. Этот процесс продолжается до тех пор, пока все пути сегмента S не разместятся на плоскости, или окажется, что некий путь не может быть уложен.

Предложена наглядная текстовая интерпретация укладки графа на плоскость. Действительно, можно получить представление о том, как нужно размещать граф, если выводить пути, учитывая два их атрибута: уровень рекурсии, на котором был порожден текущий путь, и с какой стороны от цикла очередного сегмента размещается обрабатываемый путь.

Изображение ребер плоского графа прямыми линиями

Теорема Фари. (Теорема была независимо доказана Вагнером (Wagner, 1936) и Фари (Fáry, 1948). Иногда ее называют теоремой Фари–Вагнера.) *Для любого плоского графа существует плоское представление, в котором все ребра – прямые линии.*

Доказательство. При доказательстве достаточно рассмотреть максимально плоские графы, так как если существует плоское представление с прямолинейными ребрами для максимально плоского графа, то удалением лишних ребер можно превратить его в необходимый граф.

В случае $n=1,2,3$ справедливость теоремы очевидна. При $n=3$ имеем треугольную грань – граф K_3. Добавим четвертую вершину ($n=4$), которая по отношению к треугольной грани может находиться в одном из двух положений: либо внутри нее, либо снаружи. Если четвертая вершина находится внутри треугольной грани, то очевидным образом ее можно соединить прямолинейными ребрами с тремя вершинами графа

K_3. Если четвертая вершина находится снаружи треугольной грани, то менее очевидным образом ее также можно соединить прямолинейными ребрами с тремя вершинами графа (рис. 85).

Далее доказательство следует методу математической индукции по числу вершин графа. Пусть теорема верна для плоского максимального графа с числом вершин $n > 3$, который имеет плоское представление с прямыми ребрами. Добавим еще одну вершину. Она может оказаться либо внутри одной из треугольных граней, либо снаружи.

Если дополнительная вершина попадет внутрь, то ее можно соединить прямыми ребрами с тремя вершинами этой треугольной грани. Если дополнительная вершина находится снаружи графа, то она может быть соединена с какими-либо внешними тремя вершинами триангулированного графа. Если вдруг вершина попадет на ребро, результат очевиден.

Дополнительная вершина оставляет граф максимальным, а все грани треугольными. Из принципа математической индукции следует справедливость теоремы для произвольного плоского максимального графа с любым числом вершин.

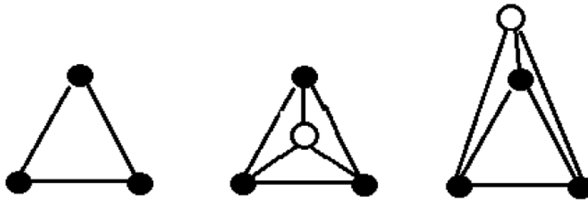


Рис. 85

Скрещивание $cr(G)$

Crossing

Планарные графы составляют весьма малую долю от всех графов, поскольку для планарности в графе должно быть мало ребер, а в среднестатистическом графе число ребер $m \sim n^2$ (число вершин).

Чтобы не рассматривать только предельные случаи планарности и непланарности, можно классифицировать графы по степени непланарности. Критерием непланарности может быть число скрещиваний.

Число скрещиваний $cr(G)$ *изображения* графа на плоскости есть число пар пересекающихся ребер (не имеющих общих вершин).

Число скрещиваний $cr(G)$ самого графа есть минимальное число скрещиваний среди всех изображений графа на плоскости.

Раскраски графа

Colouring

Рассматривается раскраска вершин и ребер графа. Раскрашиваются в основном неориентированные графы и не имеющие петель.

Вершинная раскраска

Граф G называют *r*-раскрашиваемым, или *r*-хроматическим, если его вершины могут быть раскрашены с использованием *r* цветов так, что все смежные вершины имеют разные цвета. Это так называемая правильная раскраска. Другие раскраски практически не рассматриваются. Поэтому слово «правильная» часто с очевидностью опускается. Наименьшее число *r* такое, что граф G является *r*-хроматическим, называется хроматическим числом (или индексом) графа и обычно обозначается $\chi(G)$, или $\gamma(G)$. Граф не может быть раскрашен в число цветов $< \chi$. Кроме *r*, используется часто обозначение k.

Задача нахождения хроматического числа графа называется задачей о раскраске (или задачей раскраски) графа. Соответствующая этому числу раскраска вершин разбивает множество вершин графа на *r* подмножеств, каждое из которых содержит вершины одного цвета. Эти множества независимые, поскольку в пределах одного множества нет двух смежных вершин.

Пример вершинной раскраски графа приведен на рис. 86.

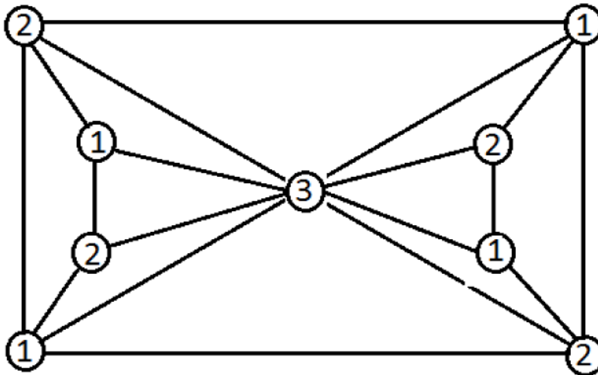


Рис. 86. Вершинная раскраска графа.
Числа на вершинах – номера цветов. Хроматический индекс $\chi = 3$

Реберная раскраска

При реберной раскраске никакие два смежных ребра не раскрашены одинаковым цветом. Такое распределение цветов называется *собственной раскраской ребер*.

Граф *r-реберно раскрашиваем*, если существует окраска, использующая r цветов.

Реберно-хроматическим числом (*индексом*) $\chi'(G)$ называется наименьшее число цветов, необходимых для реберной раскраски графа G.

Пример реберной раскраски графа приведен на рис. 87.

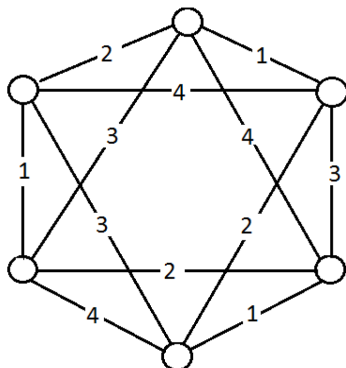


Рис. 87. Реберная раскраска графа.
Числа на ребрах – номера цветов. Хроматический индекс $\chi' = 4$

Поскольку задачу правильной раскраски точно решить очень трудно, то могут быть полезны оценки хроматического числа, выражаемые через просто вычислимые параметры графа. Вообще говоря, хроматическое число графа нельзя найти, зная только числа вершин, ребер графа и степени каждой вершины. Однако при известных этих величинах можно получить верхнюю и нижнюю оценки для хроматического числа графа.

Рассмотрим несколько оценок хроматических чисел, связанных со степенями вершин графа. Нижеприводимые верхние и нижние оценки во многих случаях столь неточны, что не имеют практического значения. Тем не менее, можно построить графы, на которых эти оценки достигаются. Однако известно, что можно построить такой граф, который не содержит даже полного подграфа третьего порядка и будет иметь произвольно большое значение хроматического числа. Эти оценки дают приблизительно одинаковую точность. Поэтому выбор из них следует осуществлять, исходя из уже вычисленных данных.

Нижние оценки $\chi(G)$

Нижние оценки хроматического числа, безусловно, более интересны, чем верхние. Очевидно, что хроматическое число графа $\chi(G)$ ограничено снизу кликовым числом графа $\omega(G)$:

$$\chi(G) \geq \omega(G).$$

Можно сделать более точную оценку. Количество вершин в максимальном независимом множестве графа G $\alpha(G) = \omega(G')$, где граф G' есть дополнение графа G.

$$\chi(G) \geq [n / \alpha(G)], \text{ или } \chi(G) \geq [n / \omega(G')],$$

где [] означает целую часть числа.

Известны еще две оценки:

$$\chi(G) \geq \,]2n^{1/2}[- \chi(G'), \; \chi(G) \geq n / \chi(G'),$$

где][означает наименьшее целое число, которое не меньше аргумента.

Существует и еще одна оценка, менее точная, но использующая лишь количество вершин и ребер графа:

$$\chi(G) \geq n^2 / (n^2 - 2m).$$

Верхние оценки $\chi(G)$

Предложены следующие легко вычисляемые оценки для любого графа:

$$\chi(G) \leq 1 + \max_H \delta(H),$$

$$\chi(G) \leq 1 + \Delta(G),$$

где $\delta(H)$ – минимальная из степеней вершин графа, H – подграфы графа G, $\Delta(G)$ – максимальная из степеней вершин графа G.

Если G – связный граф, не являющийся полным, и $\Delta(G) \geq 3$, то $\chi(G) \leq \Delta(G)$.

Для каждого графа G с m ребрами справедливо

$$\chi(G) \leq 1/2 + \sqrt{2m + 1/4}.$$

Существуют также две оценки, связывающие хроматическое число графа и хроматическое число дополнения для данного графа.

$$\chi(G) \leq n + 1 - \chi(G'), \; \chi(G) \leq [(n+1)^2 / 4] / \chi(G').$$

Для некоторых графов хроматические числа очевидны, например: $\chi(K_{n,m}) = 2$, $\chi(C_{2n}) = 2$, $\chi(C_{2n+1}) = 3$.

Естественно, граф является 1-хроматическим тогда и только тогда, когда он пустой, а 2-хроматическим – когда он двудольный и непустой. Обычно 2-хроматический граф называют *бихроматическим* (или дихроматическим). Граф является *бихроматическим* тогда и только тогда, когда он не содержит циклов нечетной длины.

Для реберной раскраски интересна более точная раскраска:

$$\Delta \le \chi' \le 1 + \Delta .$$

Хроматический индекс χ' для некоторых классов графов:

$$\chi'(K_{2n+1}) = \Delta(K_{2n+1}) + 1 = 2n + 1 , \ \chi'(K_{2n}) = \Delta(K_{2n}) = 2n - 1 .$$

Хроматический полином $\chi(G, x)$

Chromatic polynomial

Для графа G и фиксированного числа красок x количество способов правильной *x*-раскраски графа G называется *хроматическим полиномом*.

Теорема Биркгофа–Уитни. *Для каждого графа G существует точно один полином такой, что для любого x число $\chi(G, x)$ правильных раскрасок графа в x цветов равно значению этого полинома в точке x.*

Ввиду этой теоремы хроматическая функция называется хроматическим полиномом.

Основные свойства хроматического полинома

1. Степень хроматического полинома равна n.
2. Старший коэффициент равен 1.
3. Второй коэффициент равен $-m$ (числу ребер).
4. Коэффициенты знакопеременны, т. е. коэффициент при $x^{n-2k} \ge 0$, а коэффициент при $x^{n-2k+1} \le 0$ для любого целого k.
5. Третий коэффициент, считая с самого старшего, однозначно определяется набором подграфов графа, содержащих 3 вершины.
6. Число $|\chi(G, -1)|$ равно числу ациклических ориентаций графа, т. е. числу способов так расставить стрелки на его рёбрах, чтобы полученный ориентированный граф не содержал ориентированных циклов.
7. Если хроматический полином графа равен $x(x-1)^{n-1}$, то граф – дерево.

Ниже приведены интересные доказательства некоторых этих свойств.

Рекуррентные формулы для хроматических полиномов

Напомним, что граф, полученный из графа G стягиванием ребра (u,v), обозначается G / uv.

Теорема. *Пусть* u *и* v *– несмежные вершины графа* G. *Если* $G_1 = G \cup uv$, *а* $G_2 = G / uv$, *то* $\chi(G,x) = \chi(G_1,x) + \chi(G_2,x)$.

Доказательство. Рассмотрим все произвольные раскраски графа G. Рассмотрим те из них, при которых вершины u и v окрашены в разные цвета. Если добавить к графу G ребро (u,v), то они не изменятся, то есть останутся правильными. Рассмотрим раскраски, при которых u и v одного цвета. Все эти раскраски останутся правильными и для графа, полученного из G слиянием вершин u и v.

Если к некоторому произвольному графу добавлять ребра последовательно, не меняя его вершин, то на каком-то шаге получится полный граф. Аналогично, если в произвольном графе уменьшать число вершин путем их отождествления, не меняя числа ребер, то также получится полный граф.

Следствие. *Хроматический полином любого графа* G *равен сумме хроматических полиномов некоторого числа полных графов, число вершин в которых не больше, чем в графе* G.

Теорема. *Пусть* u *и* v *– смежные вершины графа* G. *Если* $G_1 = G \setminus uv$ и $G_2 = G / uv$, то $\chi(G,x) = \chi(G_1,x) - \chi(G_2,x)$.

Доказательство следует из предыдущей теоремы.

Коэффициенты хроматического полинома

Теорема. *Свободный член хроматического полинома равен 0.*

Доказательство. По определению хроматического полинома графа G, его значение в точке x равно количеству способов раскрасить вершины G правильным образом в x цветов. Количество способов раскрасить граф в 0 цветов равно 0. То есть $\chi(G,0) = 0$. Из этого следует, что $P(G,x)$ кратен x, следовательно, его свободный член равен 0.

Теорема. *Старший коэффициент хроматического полинома равен* 1.

Доказательство. Воспользуемся рекуррентной формулой $\chi(G,x) = \chi(G_1,x) + \chi(G_2,x)$, где G_1 – граф, полученный из G добавлением отсутствующего в G ребра uv, а G_2 – граф, полученный из G слиянием вершин u и v в одну вершину и удалением возникших при этом кратных ребер. Применяя рекуррентную формулу повторно, хроматический полином можно представить в виде суммы хроматических полиномов полных графов, то есть

$$\chi(G,x) = \chi(K_n,x) + a_1\chi(K_{n-1},x) + a_2\chi(K_{n-2},x) + \ldots =$$
$$= x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots$$

Из этой формулы видно, что хроматический полином имеет старший коэффициент, равный 1.

Теорема. *Коэффициенты хроматического полинома составляют знакопеременную последовательность.*

Доказательство проводится индукцией по количеству вершин.

База индукции. Теорема верна для графа G из одной вершины, потому что $\chi(G,x) = x$.

Индукционный переход: $n \to n+1$. Предположим, что теорема верна для всех графов на n вершинах. Рассмотрим графы на $n+1$ вершине. Индукционный переход доказывается индукцией по количеству ребер графа G. Если G не содержит ребер: $G = O_{n+1}$, его хроматический полином $\chi(G,x) = x^{n+1}$ обладает доказываемым свойством.

Далее предположим, что для всех $(n+1,q)$-графов теорема верна. Возьмем $(n+1,q+1)$-граф G_1 и его ребро uv. Обозначим за G граф, полученный из G_1 удалением ребра uv ($G = G_1 - uv$), а за G_2 граф, полученный из G_1 слиянием вершин u и v. Тогда из рекуррентной формулы следует $\chi(G_1,x) = \chi(G,x) - \chi(G_2,x)$.

Так как $G - (n+1,m)$ есть граф, а в G_2 – n вершин, то для G и G_2 теорема верна:

$$\chi(G,x) = x^{n+1} - a_1x^n + a_2x^{n-1} - a_3x^{n-2} + \ldots ,$$

$$\chi(G_2,x) = x^n - b_1x^{n-1} + b_2x^{n-2} + \ldots ,$$

где $a_1, a_2 \ldots a_{n+1}, b_1, b_2 \ldots b_n$ – некоторые целые числа ≥ 0. Из этих равенств следует $\chi(G_1,x) = x^{n+1} - (a_1+1)x^n + (a_2+b_1)x^{n-1} + \ldots$

Видно, что в этом полиноме коэффициенты составляют знакопеременную последовательность.

Теорема. *Второй коэффициент хроматического полинома равен по модулю количеству ребер графа.*

Доказательство. Из предыдущей теоремы видно, что при увеличении количества ребер графа на 1, второй коэффициент также увеличивается на 1. Так как для пустого графа второй коэффициент равен 0, то утверждение верно для любого графа.

Примеры

Хроматический полином полного графа

$\chi(K_n, x) = x(x-1)...(x-n+1)$, так как первую вершину полного графа K_n можно окрасить в любой из x цветов, вторую – в любой из оставшихся $x-1$ цветов и т. д. Очевидно, что если x меньше n, то и полином равен 0, так как один из его множителей 0.

Хроматический полином нуль-графа

$\chi(O_n, x) = x^n$, так как каждую из n вершин нулевого графа O_n можно независимо окрасить в любой из x цветов.

Хроматический полином простой цепи (chain)

Пусть C_n – простая цепь, состоящая из n вершин. Рассмотрим процесс раскраски простой цепи. Первую вершину можно покрасить в один из x цветов, вторую и последующие вершины – в один из $x-1$ цветов так, чтобы цвет не совпадал с предыдущей вершиной. Тогда $\chi(T_n, x) = x(x-1)^{n-1}$.

Хроматический полином цикла (cycle)

Хроматический полином цикла C_n есть

$$\chi(C_n, x) = (x-1)^n + (-1)^n(x-1).$$

Доказательство по индукции по количеству вершин.

База индукции. Сначала рассматривается $n = 3$:

$$\chi(C_3, x) = x(x-1)(x-2) =$$
$$= (x^3 - 3x^2 + 3x - 1) - (x-1) = (x-1)^3 + (-1)^3(x-1),$$

что удовлетворяет формулировке теоремы.

Индукционный переход. Пусть $\chi(C_k, x) = (x-1)^k + (-1)^k(x-1)$.

Рассматривается $n = k+1$. По теореме о рекуррентной формуле для хроматических полиномов $\chi(C_{k+1}, x) = \chi(C_{k+1} \setminus e, x) - \chi(C_{k+1} / e, x)$ (где e – любое ребро C_{k+1}).

Заметим, что граф C_{k+1} / e изоморфен C_k, а граф $C_{k+1} \setminus e$ является простой цепью. Тогда

$$\chi(C_{k+1},x) = \chi(T_{k+1},x) - \chi(C_k,x) =$$
$$= x(x-1)^k - (x-1)^k - (-1)^k(x-1) = (x-1)^{k+1} + (-1)^{k+1}(x-1)$$

Хроматический полином дерева

Теорема. *Граф G_n является деревом тогда и только тогда, когда* $\chi(G,x) = x(x-1)^{n-1}$.

Доказательство. Сначала покажем, что хроматический полином любого дерева T_n есть $x(x-1)^{n-1}$. Доказательство проводится индукцией по числу n. Для $n=1$ и $n=2$ результат очевиден. Предположим, что $\chi(T',x) = x(x-1)^{n-2}$ для любого дерева T' с количеством вершин равным $n-1$. Пусть uv – ребро дерева T такое, что v является висячей вершиной. По нашему предположению хроматический полином дерева T без ребра uv равен $\chi(T/uv,x) = x(x-1)^{n-2}$. Вершину v можно окрасить $x-1$ способом, так как её цвет должен только лишь отличаться от цвета вершины u. В результате получаем $\chi(T,x) = (x-1\chi(T/uv,x) =$ $= x(x-1)^{n-1}$. Обратно, пусть G – граф, у которого $\chi(G,x) = x(x-1)^{n-1}$. Тогда верны два следующих утверждения.

1. Граф G связен, потому что если было бы две компоненты связности (или больше), то $\chi(G,x)$ делился бы на x^2 без остатка.

2. В графе G количество рёбер равно $n-1$, так как по одной из теорем о коэффициентах хроматического полинома, количество рёбер в графе соответствует коэффициенту при x^{n-1}, взятому со знаком минус. В нашем случае, этот коэффициент удобно искать, используя бином Ньютона:

$$\chi(G,x) = x(x-1)^{n-1} = x(x^{n-1} - C_{n-1}^1 x^{n-2} + C_{n-1}^2 x^{n-3} - ... + (-1)^{n-1}) =$$
$$= x^n - (n-1)x^{n-1} + ... + (-1)^{n-1}x$$

Из этих двух утверждений (связность и $n-1$ ребро) следует, что граф G является деревом.

Оптимальная независимая раскраска

Если граф *r*-хроматический, то он может быть раскрашен с использованием $\le r$ красок с помощью следующей процедуры. Сначала в один цвет окрашивается некоторое максимальное независимое множество

$S(G)$, затем окрашивается в следующий цвет множество $S(H \setminus S(H))$. Процесс рекурсивный и продолжается до раскрашивания всех вершин.

Доказательство. Тот факт, что такая раскраска, использующая только r цветов, всегда существует, может быть установлен следующим образом. Пусть существует раскраска в r цветов такая, что одно или больше множеств, окрашенных в один и тот же цвет, не являются максимальными независимыми множествами в смысле, упомянутом выше.

Перенумеруем цвета произвольным способом. Очевидно, что можно всегда покрасить в цвет k те вершины некоего множество $\bar{V}_k$, которые не были окрашены в этот цвет и которые образуют максимальное независимое множество вместе с множеством V_k всех вершин графа, уже окрашенных в цвет k. Эта новая раскраска возможна потому, что никакая вершина из множества $\bar{V}_k$ не является смежной ни с какой вершиной из V_k и, следовательно, всякая вершина, которая смежная хотя бы с одной вершиной из $\bar{V}_k$, окрашена в цвет, отличный от цвета k, и поэтому не затрагивается процедурой перемены цвета вершин из $\bar{V}_k$. Рассматривая теперь подграф $G = G - \bar{V}_k$ и проводя с ним аналогичные действия, в цвет $k+1$ будет окрашено новое максимальное независимое множество и т. д.

Алгоритм раскрашивания максимального независимого множества

1. Положить $G_0 = G$.
2. Если граф G_k построен, то выбрать в нем максимальное независимое множество вершин S_k.
3. Окрасить вершины множества S_k одним цветом k.
4. Вычислить $G_{k+1} = G_k - S_k$.
5. Окончание процедуры при $G_K = 0$.

Алгоритм может быть реализован как итеративной, так и рекурсивной функциями.

По-видимому, алгоритм не обязательно приводит к минимальной раскраске.

Раскраска планарных графов

Проблема раскраски планарных графов является одной из самых знаменитых проблем теории графов. Первоначально – в середине XIX века – вопрос формулировался в следующем виде. Достаточно ли четырех красок для такой раскраски произвольной географической карты, при которой любые две соседние страны окрашены в различные цвета? На картах граница любой страны должна состоять из одной замкнутой линии, а соседними считаются страны, имеющие общую границу ненулевой длины.

Позднее понятия карты и ее раскраски были формализованы следующим образом. Связный плоский мультиграф без мостов называется *картой*. Грани карты, имеющие общее ребро, называются *смежными*. Функция f, ставящая в соответствие каждой грани Γ карты натуральное число (*цвет грани*) $f(\Gamma) = \{1, 2, ..., k\}$, называется *k-раскраской*, если цвета смежных граней различны. Карта называется *k-раскрашиваемой*, если для нее существует *k-раскраска*.

В 1879 г. Кэли опубликовал статью, посвященную проблеме раскраски карт, в которой сформулировал гипотезу четырех красок.

Гипотеза четырех красок: *всякая карта* 4-*раскрашиваема*.

Часто используется другая формулировка гипотезы четырех красок: *всякий планарный граф* 4-*раскрашиваем*.

Поскольку планарный граф по определению изоморфен некоторому плоскому, то эквивалентность этих двух формулировок гипотезы четырех красок вытекает из следующего утверждения.

Карта является *r*-раскрашиваемой тогда и только тогда, когда геометрически двойственный ей граф вершинно *r*-раскрашиваем.

Существуют плоские графы, которые нельзя раскрасить правильно менее чем четырьмя цветами. Например, полный граф K_4.

Теорема о 4 красках. *Каждый планарный граф G можно так раскрасить четырьмя красками, что любые две смежные вершины будут окрашены в разные цвета, т. е.* $\chi(G) \le 4$.

Теорема о 5 красках. *Каждый планарный граф G можно так раскрасить, используя пять цветов, что любые две смежные вершины будут окрашены в разные цвета, т. е.* $\chi(G) \le 5$.

В 1890 г. Хивуд (Heawood) доказал теорему пяти красок, показав, что любая плоская карта может быть раскрашена не более чем пятью цветами. В XX столетии было разработано большое количество теорий в попытках уменьшить минимальное число цветов.

Теорема четырёх красок была доказана лишь в 1977 г. Аппелем (Appel) и Хакеном (Haken). Доказательство теоремы четырёх цветов явля-

ется одним из первых доказательств, в которых был использован компьютер.

Существуют некоторые результаты раскраски планарных графов двумя и тремя цветами.

Плоский двусвязный граф является 2-*раскрашиваем* тогда и только тогда, когда граница каждой его грани содержит четное число ребер.

Плоский граф 3-*раскрашиваем* тогда и только тогда, когда он является подграфом плоской триангуляции с четными степенями вершин.

Задачи построения минимальной раскраски произвольного графа являются очень сложными, эффективные алгоритмы их решения неизвестны.

Рассмотрим простые алгоритмы и программы построения правильной раскраски, непредсказуемо приводящей к раскраске, близкой к минимальной.

Эвристические (жадные) алгоритмы раскраски

Эти алгоритмы раскраски весьма просто реализуются, но дают завышенные значения χ, и потому рассматриваются как приближенные. Они реализуются в двух вариантах. В одном из них основной цикл идет по цветам, а другой – по вершинам. Все цвета имеют С-номера: $0,1,\ldots,n-1$.

Вариант 1. Внешний цикл – по цветам, внутренний – по вершинам. Раскрашиваются в предлагаемый цвет внешнего цикла вершины, которые не имеют смежных вершин, уже раскрашенных в этот цвет. Функция возвращает максимальный цвет, а в параметрах – массив цветов C.

```
    int Paint(int **A, int n, int *C)
{   int i, j, k=0, c;
    for(i=0; i<n; i++) C[i]=-1; // initialization
    for(c=0; c<n; c++)    // loop on colors
    for(i=0; i<n; i++)    // loop on vertexes
 {  if(C[i]!=-1)continue; // i vertex is already coloured
    for(j=0; j<n; j++)   // look through adjacent vertexes
    if(A[i][j] && C[j]==c) goto conti;
// adjacent vertex j is already painted in c colour
// if all adjacent vertexes are not painted in c colour
// then it is painted in c colour
    C[i]=c;    // colour i vertex
    k++;       // number of the coloured vertexes
    if(k==n) return c;    // all vertexes are coloured
    conti:;
 }  // for i
    return c;  // return max used color
}   // Paint
```

Вариант 2. Внешний цикл – по вершинам, внутренний – по цветам.

Вводится вспомогательная функция `Color`, которая раскрашивает `i` вершину, задаваемой аргументом, в минимальный цвет, начиная с цвета `q`, также задаваемого аргументом. Функция `Color` возвращает найденный минимальный цвет.

```
    int ColorVertex(int **A, int n, int i, int *C, int q)
{   int j, c;
    for(c=q; c<q+n; c++)  // loop on colors
 {  for(j=0; j<n; j++)    // loop on vertexes
    if(A[i][j] && C[j]==c) goto contc;
    C[i]=c; return c;
    contc:;
 }  // for c
    return -1;
}   // ColorVertex
```

Фунция `ColorGraf` раскрашивает все вершины графа, используя функцию `ColorVertex`. Фунция `ColorGraf` возвращает максимальный цвет, а в параметрах – массив цветов `C[0..n-1]`.

```
    int ColorGraf(int **A, int n, int *C)
{   int i, c=-1;
    for(i=0; i<n; i++)    // loop on vertexes
 {  C[i]=ColorVertex(A,n,i,C,0);
    if(C[i]>c) c=C[i];    // c is max color
 } // i
    return c;
}   // ColorGraf
```

Классическая схема жадного алгоритма сортирует вершины по степеням и начиная с вершины с максимальной степенью последовательно присваивает каждой v_i вершине наименьший доступный цвет, не использовавшийся для окраски смежных вершин ($v_1, v_2, \ldots, v_{i-1}$), либо добавляет новый. Качество полученной раскраски зависит от выбранного порядка. Всегда существует такой порядок, который приводит жадный алгоритм к оптимальному числу красок.

Оптимизация жадного алгоритма

Приведенный выше алгоритм можно попытаться улучшить. Для этого после каждого шага нужно упорядочивать неокрашенные вершины. Один из вариантов описанной выше эвристической процедуры состоит в переупорядочивании неокрашенных вершин по невозрастанию их относительных степеней. Под относительными степенями понимаются степени

соответствующих вершин в *неокрашенном* подграфе исходного графа. Если две вершины имеют одинаковые степени, то порядок таких вершин случаен. Такие вершины можно также упорядочить, но уже по двухшаговым степеням. Двухшаговую степень можно определить как сумму относительных степеней инцидентных вершин или как максимальную степень предыдущей ступени. Аналогично можно поступать и далее. Также можно изначально упорядочивать вершины по двухшаговым или трехшаговым степеням. Следует заметить, что подобного рода программистские изыски вряд ли приведут к минимальной раскраске.

```
    int PaintSort(int **A,int n,int *C)
{   int i,j,k=0,c,*d,*x,*s,*y;
    for(i=0; i<n; i++) C[i]=-1;
    d=(int *)calloc(n,4); x=(int *)calloc(n,4);
    Degree(A,n,d);
    SelectSortMaxX(d,x,n); // d=0..n-1 =const s=sorted
    for(c=0; c<n; c++)
    for(i=0; i<n; i++)
 {  if(C[x[i]]>=0) continue; // уже окрашен
    for(j=0; j<n; j++)   // просматриваем смежные вершины
  { if(A[x[i]][x[j]] && C[x[j]]==c)
    // adjacent vertex j has painted c color
    goto conti;
  } // for j
    C[x[i]]=c;
    k++;  // количество раскрашенных вершин
    if(k==n) return c;// все раскрасили
    conti:;
 }  // for i
    return c;
}   // PaintSort

    int ColorGrafSort(int **A,int n,int *C)
{   int i,k,*d,*x,*y,c=0;  // q=max nomer color 0 1...n-1
    d=(int*)calloc(n,4); x=(int*)calloc(n,4);
    y=(int*)calloc(n,4);
    Degree(A,n,d);
    SelectSort(d,x,n);    // min .. max
    for(k=n-1; k>=0; k--) // on sorted vertices
 {  i=x[k];
    C[i]=ColorVertex(A,n,i,C,0);
    if(C[i]>c) c=C[i];
 }  // for k
    return c;
}   // ColorGrafSort
```

```
    int ColorGrafSort123(int **A,int n,int *C)
{   int i,j,k,c=0; // color=0 1...n-1
    d1=(int*)calloc(n,4); d2=(int*)calloc(n,4);
    d3=(int*)calloc(n,4); s1=(int*)calloc(n,4);
    s2=(int*)calloc(n,4); s3=(int*)calloc(n,4);
    x=(int*)calloc(n,4); y=(int*)calloc(n,4);
    DegreeSum123(A,d1,d2,d3,n);
    // DegreeMax123(A,d1,d2,d3,n);
    Bubble(s1,s2,s3,y,n); // s1,s2,s3 = sort
    for(i=0; i<n; i++) C[i]=-1;
    for(k=0; k<n; k++)// on sorted vertexes
 {  i=y[k];
    C[i]=ColorVertex(A,n,i,C,0);
    if(C[i]>c) c=C[i];
 }
    return c;
}   // ColorGrafSort123

    int DegreeSum123(int **A,int *d1,int *d2,int *d3,int n)
{   int i,j,k,kmax=0,m=0;
    // 1
    Degree(A,n,d1);
    // 2
    for(i=0; i<n; i++)
 {  d2[i]=0;
    for(j=0; j<n; j++)
    if(A[i][j]) d2[i]=d2[i]+d1[j];
 }  // for i
    // 3
    for(i=0; i<n; i++)
 {  d3[i]=0;
    for(j=0; j<n; j++)
    if(A[i][j]) d3[i]=d3[i]+d2[j];
 }  // for i
    return 1;
}   // DegreeSum123
```

Функция сортировки по убыванию. Равные элементы `s1` сортируются по данным `s2`, при равенстве элементов `s2` учитываются элементы `s3`.

```
    int Bubble123(int *s1,int *s2,int *s3,int *x,int n)
{   int count=0,k,i,h;
    // init
    for(k=0; k<n; k++) x[k]=k;
    // main loop
```

```
    for(k=0; k<n; k++) // min ->
 {  for(i=0; i<n-1-k; i++)
 {  if(s1[i]<s1[i+1] || (s1[i]==s1[i+1] &&
    (s2[i]<s2[i+1] || s2[i]==s2[i+1] && s3[i]<s3[i+1])))
  { h=s1[i]; s1[i]=s1[i+1]; s1[i+1]=h;
    h=s2[i]; s2[i]=s2[i+1]; s2[i+1]=h;
    h=s3[i]; s3[i]=s3[i+1]; s3[i+1]=h;
    h=x[i];x[i]=x[i+1];x[i+1]=h;
    count++;
  } // exchange
 }  // for i
}   // for k
    return count;
}   // Bubble123
```

Результаты расчетов, иллюстрирующие влияние сортировки, как одноступенчатой, так и трехступенчатой, показаны на рис. 88. Для исключения «случайности» расчет проводился 100 раз и затем усреднялся. Поэтому усредненные цвета – вещественные числа.

Сортировка не оказывает существенного влияния на максимальный цвет графа.

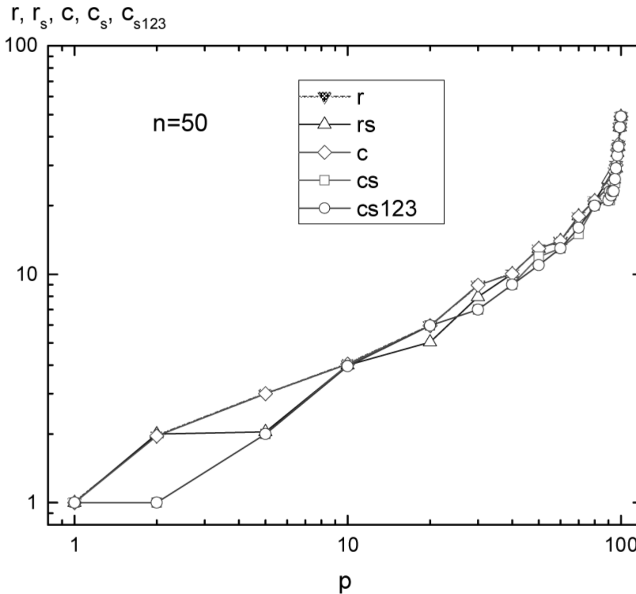


Рис. 88. Зависимости цвета для функций:
Paint (r), PaintSort (rs), Color(c), ColorSort(cs), ColorSort123(cs123)

Жадный алгоритм может и ухудшить раскраску. Например, граф-корона (рис. 89) может быть раскрашен двумя цветами, а упорядочивание может привести к числу цветов равному n .

Корона с $2n$ вершинами есть полный двудольный граф, из которого удалено совершенное паросочетание.

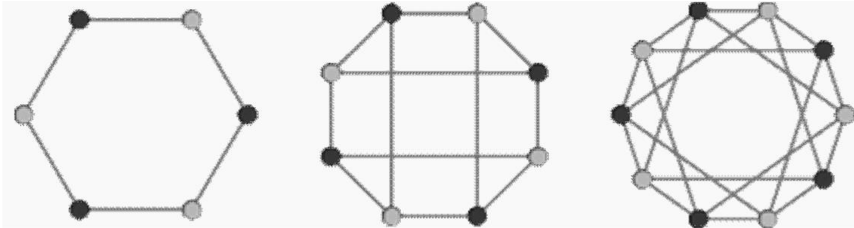


Рис. 89. Короны с шестью, восемью и десятью $2n$ вершинами и $m = n(n-1)$ рёбрами: $\chi = 2$, если $n \geq 2$; $\chi = 1$, если $n = 1$

Перекраска

Repaint algorithm

Пусть q есть число цветов, использованное для проведенной раскраски. Если существует раскраска, использующая только $q-1$ цветов, то все вершины, окрашенные в цвет q, должны быть перекрашены в цвет $p < q$. Пусть v_i – первая вершина при заданном упорядочении, которая была окрашена в цвет q. Поскольку она была так окрашена потому, что не могла быть окрашена в цвет $< q$, то ее можно перекрасить в цвет $< q$ лишь перекрасив предварительно хотя бы одну из смежных с ней вершин. Шаг возвращения из вершины v_i можно осуществить следующим образом. Из смежных с v_i вершин в множестве $\{v_0, v_1, ..., v_{i-1}\}$ определяется вершина с наибольшим индексом v_k. Если v_k окрашена в цвет c_k, то v_k необходимо перекрасить в другой допустимый цвет. Но попытаемся окрашивать смежные вершины не в минимально возможный цвет.

Если это возможно, то надо последовательно перекрасить все вершины с v_k до v_n. При этом перекрашивании, естественно, цвет q использовать нельзя. Если такая процедура осуществима, то будет найдена новая лучшая раскраска, использующая меньше, чем q цветов. В противном

случае, т. е. если встретится вершина, требующая цвет q, то нужно снова сделать шаг возвращения из такой вершины.

Если v_k невозможно перекрасить в меньший цвет, то можно сразу же делать шаг возвращения из вершины v_k. Алгоритм заканчивает работу, когда на шаге возвращения достигается вершина v_1.

```
    int MaxC(int **A, int n, int *C)
{   int i, c=-1;  // c=max number  color 0 1...n-1
    for(i=0; i<n; i++)
    if(C[i]>c) c=C[i];
    return c;
}   // MaxC

    int RePaint(int **A, int n, int *C)
{   int c, q, p, k, i, r, m, count=0;
    re0: q=MaxC(A, n, C);// first index q color
    re:
    for(m=0; m<n; m++)
  { if(C[m]==q) break; }
    re1:
    count++;
    for(k=m-1; k>=0; k--) // search last adj index
  { if(A[m][k]) break; }
    p=C[k];
    if(p+1<q)// decrease impossible
{   c=ColorVertex(A, n, k, C, p+1); // C[k]=c;
    if(c==q)  goto  re;
    for(i=k+1; i<n; i++)  // все repaint
  { c=ColorVertex(A, n, i, C, 0);
    if(c==q) goto re;
  } //  C[i]=c;
    goto re0; // select next vertex q
 }  // if(p+1<q)
    else
  { if(k==0){ r=MaxC(A,n,C);return r;}
    m=k;  goto re1;
  }
    r=MaxC(A, n, C);
    return r;
}   // RePaint
```

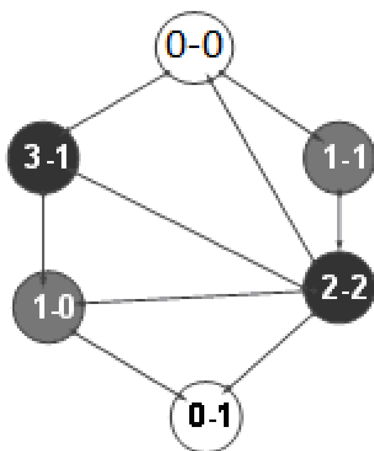


Рис. 90. RePaint: $q = 4 \to q = 3$

Рисование графов

Drawing graphs

Граф может рисоваться на плоскости или в трехмерном пространстве. Он может изображаться целиком, частично или иерархически, например, путем стягивания некоторых его подграфов в вершины, которые могут раскрываться по требованию.

Вершины могут быть нарисованы в виде точек, кругов, прямоугольников или других геометрических фигур, или представлены неявно – через имена, которыми вершины помечены.

Ребра могут быть нарисованы в виде отрезков прямых, ломаных линий или кривых.

Информация, сопоставленная элементам графовой модели, может визуализироваться с использованием текстовых меток, расположенных внутри или рядом с его изображением, различными цветами или другими визуальными элементами, такими как, например, толщина линий или размер прямоугольников.

Понятие качества изображения

Качество одного и того же изображения может по-разному оцениваться, а различные приложения могут требовать многообразные способы изображения графа.

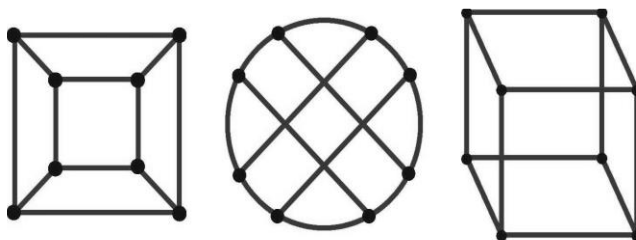


Рис. 91

Главный критерий оценки качества методов рисования является адекватность изображения графа заданному типу информации и характеру её использования. Например, при визуализации географических карт или схем дорог нужно, чтобы расположение вершин и ребер соответствовали географическим реалиям

Понятие качественного способа рисования графа формализуется с помощью таких понятий, как изобразительное соглашение, эстетический критерий и другие ограничения.

Изобразительное соглашение

Некоторые соглашения показаны на рис. 92, а также на ранее приведенных рисунках.

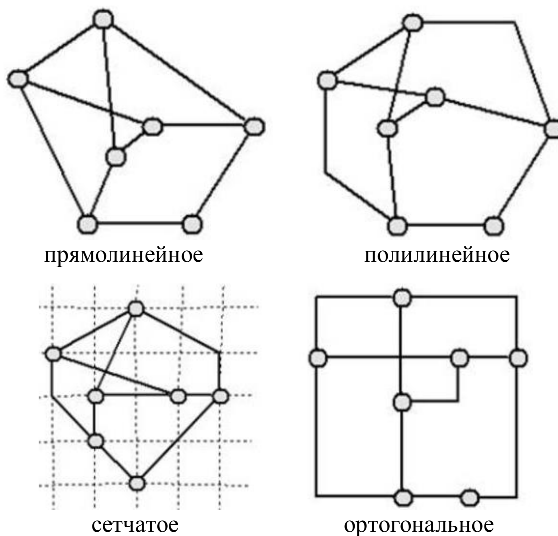


Рис. 92

Эстетические критерии

Специфицируются такие свойства изображений, которые желательно применять в возможно максимальной степени, чтобы повысить наглядность изображения:

- минимизация пересечений;
- минимизация сгибов;
- минимизация области размещения;
- максимизация разрешения;
- минимизация общей длины ребер;
- минимизация длины ребра;
- унификация длин ребер;
- минимизация числа сгибов на ребре;
- унификация сгибов;
- максимальная симметричность;
- минимизация коэффициента сторон.

Большинство из этих эстетических критериев, как задачи оптимизации, являются сложными для решения с вычислительной точки зрения, как правило, *NP*-трудными.

Критерии, как правило, противоречивы, что иллюстрирует рис. 93. Поэтому обычно при построении изображений используются различные эвристики и стратегии приближенных решений.

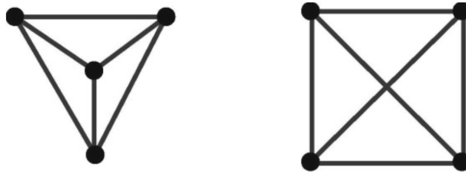


Рис. 93

Если соглашения и эвристики определяются общими правилами и критериями, которые формулируются по отношению ко всему графу и его изображению, то ограничения относятся к отдельным подграфам и частям изображений:

- центр, когда требуется разместить заданную вершину ближе к центру изображения;
- внешность, когда требуется разместить заданную вершину на внешней границе изображения;
- кластер, когда требуется разместить заданное подмножество вершин рядом друг с другом;
- последовательность слева направо (сверху вниз), когда требуется нарисовать заданный путь горизонтально.

Изоморфизм графов

Graph isomorphism

Необходимость решать задачу проверки изоморфизма графов возникает в различных областях естествознания, и методы ее решения имеют множество приложений в практической деятельности. Задача проверки изоморфизма графов имеет фундаментальное значение. Необходимо уметь отвечать на вопрос, являются ли некоторые конечные структуры принципиально различными, или же они являются лишь различными представлениями одной и той же структуры.

Отношением изоморфизма между двумя графами (неориентированными и не имеющими весов вершин и ребер) называется биекция между множествами вершин графов, сохраняющая смежность вершин. При применении понятия изоморфизма к ориентированным или взвешенным графам, накладываются дополнительные ограничения на сохранение значений весов и ориентации дуг.

Структура химических соединений естественным образом описывается помеченными графами, в которых каждая вершина представляет отдельный атом. Поиск в базе данных всех молекул, содержащих определенную функциональную группу, является задачей выявления изоморфизма подграфов.

Некоторые вопросы распознавания образов формулируются в виде задачи выявления изоморфизма графов или подграфов.

Важным применением изоморфизма графов является выявление симметрии. Отображение графа на самого себя называется *автоморфизмом*, а группа автоморфизмов содержит много информации о симметричности графа. Например, полный граф K_n содержит n автоморфизмов, в то время как произвольный граф, вероятно, будет иметь малое количество автоморфизмов (возможно, только один), поскольку граф идентичен себе самому.

Рассматривается несколько видов задачи изоморфизма графов. Например, полезно определить, является ли данный граф *G подграфом* графа *H*. Такие задачи, как задача о клике, независимом множестве и гамильтоновом цикле, являются важными частными случаями задачи выявления изоморфизма подграфов.

Выражение «граф *G* содержится в графе *H*» – неоднозначно. Во-первых, в задаче выявления *изоморфизма подграфа* требуется выяснить, содержит ли граф *H* подмножество ребер и вершин, которое является изоморфным графу G. Во-вторых, в задаче выявления изоморфизма порожденного подграфа требуется выяснить, содержит ли граф *H* подмножество

ребер и вершин, после *удаления* которого останется подграф, изоморфный графу G. В задаче выявления изоморфизма порожденного подграфа требуется, чтобы все ребра графа G присутствовали в графе H и чтобы в графе H не было никаких «не-ребер» графа G. Клика является экземпляром обоих вариантов задачи выявления изоморфизма подграфов, в то время как гамильтонов цикл является примером лишь простого изоморфизма подграфов.

Задачи выявления изоморфизма подграфов обычно сложнее, чем задачи выявления изоморфизма графов, а задачи выявления изоморфизма порожденных подграфов еще сложнее. По-видимому, очевидным разумным подходом к решению таких задач является метод перебора с возвратом. Метки и связанные с ними ограничивающие условия можно включить в любой алгоритм метода перебора с возвратом. Кроме этого, такие ограничивающие условия значительно ускоряют поиск, создавая намного больше возможностей для разрежения пространства поиска при каждом случае несовпадения меток двух вершин.

Естественно, графы G и H с одинаковым количеством вершин изоморфны тогда и только тогда, когда их вершины можно занумеровать так, что соответствующие матрицы смежности будут равны.

Пример изоморфных и неизоморфных графов приведен на рис. 94, 95. Визуально изоморфность определяется неочевидным образом.

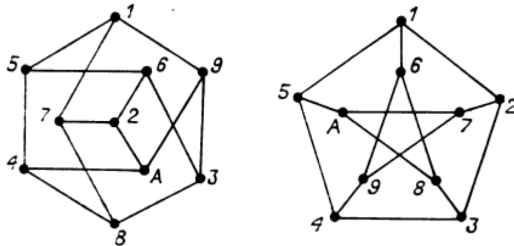


Рис. 94. Изоморфные графы

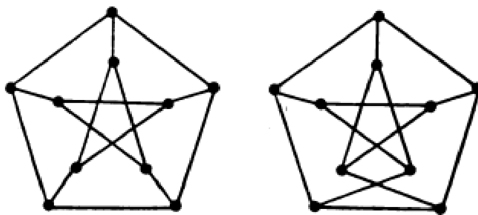


Рис. 95. Неизоморфные графы

Отношение изоморфизма является отношением эквивалентности, разбивающим множество всех графов на классы эквивалентности, которые можно рассматривать как абстрактные графы. Следовательно, изоморфные графы представляют собой один и тот же абстрактный граф. Абстрактный граф однозначно определяется матрицей смежности, но обратное утверждение, вообще говоря, неверно, так как нумерация вершин графа произвольна: каждому графу соответствует класс матриц смежности. Две матрицы смежности G и H принадлежат одному классу тогда и только тогда, когда существует матрица перестановки P такая, что $G = PHP^{-1}$. Матрица смежности G может быть получена из матрицы H при помощи некоторой совместной перестановки строк и столбцов матрицы.

Матрица перестановки P : if $(i = \varphi(j))$ $p_{ij} = 1$ else $p_{ij} = 0$. Матрица перестановки имеет в каждой строке и в каждом столбце точно по одной единице, а остальные элементы нули.

Умножение матрицы A слева на матрицу P задает перестановку ее строк, при которой *j*-строка переходит в *i*-ю. В матрице P в *i*-строке ставится 1 в $\varphi(i)$-столбец. Тогда матрица $B = PA$ получается из матрицы A такой перестановкой ее строк, что *i*-строкой становится строка с номером $\varphi(i)$-матрицы A $\{i = 1,...,n\}$, n – порядок графа.

Матрица, обратная к матрице перестановок, совпадает с транспонированной к ней: $P^{-1} = P^T$, то есть матрица со следующими элементами P^{-1} : if $(j = \varphi(i))$ $\overline{p}_{ij} = 1$ else $\overline{p}_{ij} = 0$.

Умножение $B = PA$ справа на матрицу P^{-1} задает перестановку ее столбцов, при которой *j*-столбец переходит в *i*-й. Таким образом, число, стоящее в *i*-строке и *j*-столбце матрицы PAP^{-1}, совпадает с числом, которое находится в $\varphi(i)$-строке и $\varphi(j)$-столбеце матрицы A $\{i, j = 1,...,n\}$.

Задачу проверки изоморфизма графов не удается классифицировать относительно ее сложности. Она до сих пор еще не помещена ни в один из подклассов класса *NP*. Тогда как доказано, что задача проверки изоморфизма подграфу является *NP*-полной и к ней полиномиально сводятся такие *NP*-полные задачи, как задача о максимальной клике в графе и задача проверки графа на наличие в нем гамильтонова цикла. В целом, не получено ответа на вопрос о принадлежности задачи классу *P* и *NP*-полноты задачи. Не построено алгоритмов, решавших задачу без каких-либо ограничений на структуру проверяемых на изоморфизм графов. Более того, до сих пор не известно алгоритмов, которые для графов с n вершинами

имели бы верхнюю оценку сложности, меньшую чем $O(n!/n^{\mathrm{const}})$. Сложность задачи не меняется при переходе от проверки изоморфизма неориентированных невзвешенных графов к проверке изоморфизма взвешенных графов, ориентированных графов, мультиграфов.

Полиномиальные алгоритмы разработаны для отдельных классов графов с определенными структурными свойствами, в частности для планарных графов, наиболее часто используемых в приложениях. Планарные графы представляют собой наиболее простой случай графов с ограничением на класс графа.

Одним из важных случаев задачи является определение изоморфизма деревьев. Эта задача возникает при сопоставлении языковых структур и синтаксическом анализе. Для описания структуры текста часто используется дерево синтаксического разбора. Два таких дерева будут изоморфными, если представляемые ими тексты имеют одинаковую структуру. Эффективные алгоритмы выявления изоморфизма деревьев начинают работу с листьев обоих деревьев и продвигаются к центру. Каждой вершине дерева присваивается метка, представляющая набор вершин во втором дереве, который, возможно, изоморфен поддереву с корнем в этой вершине, с учетом ограничений, накладываемых метками и степенями вершин. Например, первоначально все листья дерева T_1 потенциально эквивалентны всем листьям дерева T_2. Продвигаясь внутрь, можно разбить смежные с листьями вершины дерева T_1 на классы, в зависимости от количества смежных с ними листьев и других узлов. Отслеживая метки поддеревьев, можно убедиться, что существует одинаковое распределение помеченных поддеревьев для T_1 и T_2. Любое несовпадение означает, что $T_1 \neq T_2$. В то время как по завершении процесса все вершины оказываются разбиты на классы эквивалентности, определяющие все изоморфизмы.

Во многих приложениях обработки данных требуется выполнить поиск всех экземпляров графа с определенной структурой в большой базе данных. Задача отображения химических структур относится к этому типу задач. Такие базы данных обычно содержат большое количество сравнительно небольших графов. В связи с этим возникает необходимость индексирования базы данных графов по небольшим подструктурам (от пяти до десяти вершин каждая) и выполнения сложных проверок графов на изоморфизм только с теми, которые содержат такие же подструктуры, что и граф запроса.

Разработаны полиномиальные алгоритмы решения задачи проверки изоморфизма графов с ограниченной кратностью собственных значений

спектра матрицы смежности графа. Этот класс графов является одним из наиболее широких классов, для которых разработаны полиномиальные алгоритмы решения задачи. Алгоритм состоит из двух частей. В первой части, основанной на методах линейной алгебры, вычисляются все собственные значения матриц смежности графов и проекции базисных векторов на собственные пространства матриц смежности. Требуется значительная точность для установления равенства собственных значений, равенства проекций и равенства углов между проекциями. Во второй – комбинаторной и теоретико-групповой части алгоритма – эта информация используется для установления возможного изоморфизма.

Возникает естественный вопрос, почему на указанных классах графов задача проверки изоморфизма может быть решена за полиномиальное время? Существуют ли какие-либо общие свойства графов из этих классов, основываясь на которых можно было бы построить алгоритм, который бы охватывал как можно более широкий класс, оставаясь при этом полиномиальным? Существует ли эффективный простой алгоритм, удовлетворяющий поставленному условию на ограниченность некоего фиксированного параметра, общего для всех указанных классов? Если такой алгоритм существовал бы, то, возможно, он смог бы привести к эффективному решению задачи для новых классов графов, не содержащихся среди известных.

Эффективность проверки на изоморфизм увеличивается, если произвести предварительно разбиение множества вершин на «классы эквивалентности» таким образом, чтобы было невозможно перепутать две вершины из разных классов. Все вершины в каждом классе эквивалентности должны иметь одинаковое значение какого-либо инварианта, не зависящего от меток.

Инварианты графа

Invariants of graph

Инвариантом графа называется его характеристика, равная для изоморфных графов. Инвариант не обязательно скалярная величина, он может быть вектором и матрицей. В качестве инварианта можно рассматривать не один из частных инвариантов, а их множество.

Полный инвариант

Инвариант называется полным, если его равенство для двух графов возможно тогда и только тогда, когда графы изоморфны. Известными на сегодняшний день полными инвариантами являются так называемые мини-код $\mu_{\min}(G)$ и макси-код $\mu_{\max}(G)$ матрицы смежности, получаемые

путем выписывания двоичных значений матрицы смежности в строчку с последующим переводом полученного двоичного числа в число в любой системе счисления. Мини-коду соответствует такой порядок следования строк и столбцов, при которых полученное значение является минимально возможным. А для макси-кода полученное значение, соответственно, – максимально возможное. В настоящее время полный инвариант графа, вычислимый за полиномиальное время, неизвестен, однако не доказано, что он не существует.

Неполные инварианты

Совпадение неполных инвариантов является необходимым, но не достаточным условием наличия изоморфизма. Неполными инвариантами можно объявить все вычисляемые характеристики графа. Однако всегда остается открытым вопрос, с какой вероятностью совпадение этих инвариантов приводит к изоморфизму графов.

Популярны следующие неполные инварианты графов.

Самоочевидными инвариантами графа G являются число вершин $n(G)$ и число ребер $m(G)$.

Также очевидно, что при всяком изоморфизме графов соответствующие друг другу вершины должны иметь одинаковую степень. Упорядоченная система степеней графа (см. выше) часто называется *вектором степеней графа* $s(G)=\{s_1,s_2,...,s_n\}$, иногда *графической последовательностью*. Вектор степеней – *инвариант*.

Вектор степеней дает еще два скалярных числовых инварианта: $\min(s)$ и $\max(s)$. Такое простое разбиение может быть полезным, но не в случае с регулярными графами, вершины которых имеют одинаковую степень.

Индекс Рандича – $r=\sum_{i,j}(d_i d_j)^{-1/2}$, где d_i – степень вершины i.

Вектор степеней второго порядка. Каждый элемент этого вектора представляет собой список степеней вершин, смежных с данной вершиной.

Матрица кратчайших путей. Для каждой вершины v матрица кратчайших путей между всеми парами определяет множество из $n-1$ расстояний, представляющих расстояния между некой вершиной v и каждой из остальных вершин. Любые две одинаковые вершины должны определять одинаковые множества расстояний, поэтому можно разбить вершины на классы эквивалентности, определяющие одинаковые множества расстояний.

Количество путей длины k. Возведение в *k-ю* степень матрицы смежности графа *G* дает матрицу, в которой G_{ij}^{k} содержит количество путей от вершины i к вершине *j*. Для каждой вершины и для каждого значения k эта матрица определяет количество путей, которое можно использовать для разбиения вершин на классы, как и расстояния в предыдущем случае. Можно перепробовать все $1 < k < n$ и использовать любое отклонение как критерий для разбиения.

Диаметр, *радиус*, *медиана и центры* графа.

Индекс Винера – $\omega = \sum_{i,j} \delta_{ij}$, где δ_{ij} – минимальное расстояние между вершинами v_i и v_j.

Определитель матрицы смежности $\det(A)$.

Число компонент связности графа.

Цикломатическое число графа – минимальное число ребер, которые надо удалить, чтобы граф стал *ациклическим*.

Минимальное число вершин, которое необходимо удалить для получения несвязного графа.

Минимальное число вершин, необходимое для покрытия ребер.

Минимальное число ребер, необходимое для покрытия вершин.

Обхват графа – число ребер в составе минимального цикла.

Плотность графа – число вершин максимальной по включению клики.

Хроматическое число графа – минимальное количество цветов, требуемое для правильной раскраски вершин графа.

Характеристический многочлен матрицы смежности. Теорию графов можно отождествить с теорией матричных классов, состоящих из матриц смежности изоморфных графов, и их инвариантами. Одними из наиболее содержательных инвариантов такого матричного класса являются характеристический полином матрицы $P(\lambda) = | A - \lambda E |$ и ее спектр $\{\lambda_1, ..., \lambda_n\}$.

Существует эвристический алгоритм установления изоморфизма графов, основанный на использовании собственных значений матриц смежности графов в качестве инварианта. В вычислительной части алгоритма сравниваются инварианты исследуемых графов. В случае совпадения этих инвариантов в проверочной части осуществляется поиск подстановки, устанавливающей изоморфизм. Если ни одной такой подстановки не найдено, графы считаются неизоморфными.

Упорядоченный по возрастанию или убыванию *вектор собственных чисел матрицы смежности графа* (*спектр графа*). Сама по себе матрица

смежности не является инвариантом, так как при смене нумерации вершин она претерпевает перестановку строк и столбцов.

Вычислительная сложность инвариантов

В случае с задачей проверки изоморфизма для произвольных графов все известные алгоритмы, гарантирующие правильный ответ, экспоненциальные. Но практически для каждой из задач дискретной оптимизации существует несколько различных подходов к построению алгоритмов, предназначенных для её решения; при этом каждый из подходов эффективнее работает для своего множества входных данных.

Инварианты различаются по трудоемкости вычисления. Простые инварианты вычисляются тривиально, в то время как вычисление инвариантов $\mu_{\min}(G)$, $\mu_{\max}(G)$ вычислительно сложно.

Существуют вероятностные алгоритмы определения значений приведенных трудно вычисляемых инвариантов, однако применение подобных алгоритмов допускается не всегда.

В настоящее время полный инвариант графа, вычислимый за полиномиальное время, неизвестен, однако не доказано, что он не существует. Попытки его отыскания неоднократно предпринимались, однако не увенчались успехом.

Используя эти инварианты, часто можно разбить граф на множество небольших классов эквивалентности. После такого разбиения вершин легче использовать метод перебора с возвратом. Каждой вершине в качестве метки присваивается имя ее класса эквивалентности, и задача решается как задача паросочетания в помеченном графе. Выявить изоморфизм в высокосимметричных графах сложнее, чем в случайных, по причине снижения эффективности таких эвристических методов разбиения вершин на множество классов эквивалентности.

Одним из подходов к решению задачи проверки изоморфизма графов могла бы быть непосредственная проверка возможности получения матрицы смежности одного из графов некоторой перестановкой рядов (строк и столбцов) матрицы смежности другого графа. В наихудшем случае необходимо проверить все возможные перестановки рядов одной из матриц смежности, что неизбежно влечет экспоненциальность любого из алгоритмов такого типа. Однако рассмотрение таких инвариантов матриц смежности графов как характеристический полином, спектр и собственные векторы приводит к спектральному подходу решения задач проверки изоморфизма графов – существенно более эффективному.

Описан алгоритм установления изоморфизма графов, основанный на использовании матриц расстояний и матриц кратностей кратчайших цепей между всеми парами вершин. В том случае, когда эти средства не да-

ют возможности различать проверяемые на изоморфизм графы, происходит обращение к вычислению характеристических полиномов матриц смежности графов и, наконец, к полному перебору.

Подчеркнем, что полной системы вычисляемых за полиномиальное время инвариантов изоморфных графов построить пока не удалось.

Помимо ограничений на структуру проверяемых графов получил распространение подход, состоящий в построении алгоритмов за счет адекватного представления процесса поиска изоморфизма и отсечения заведомо неудачных путей в пространстве поиска. В этом случае не налагается никаких ограничений на структуру проверяемых на изоморфизм графов, однако трудоемкость таких алгоритмов может быть оценена только статистически.

Значительная часть эффективных алгоритмов основана на прямом построении в ходе работы алгоритма отображения множества вершин одного графа на множество вершин второго графа, являющегося изоморфизмом. В основе такого подхода лежит идея, заключающаяся в таком выборе на каждой итерации алгоритма вершин из каждого графа, который сохранял бы изоморфизм подграфов, состоящих из уже выбранных вершин и всех инцидентных им ребер проверяемых на изоморфизм графов, соединяющих выбранные вершины. Выбор пар по одной вершине из каждого графа продолжается до тех пор, пока подграфы не будут достроены до собственно самих графов, изоморфизм которых проверяется. Алгоритмы, использующие подобный подход, представляют собой некоторые вариации реализации процедуры *рекурсии с возвратом* (*backtracking*). В ходе работы подобных алгоритмов поиск происходит согласно некоторому дереву поиска и, в наихудшем случае такой даже сокращенный перебор, приводит к экспоненциальности данного алгоритма.

Предложены алгоритмы, значительно сужающие на каждой итерации пространство перебора задачи проверки изоморфизма графов, так и для задачи проверки изоморфизма подграфов. В частности, предложен алгоритм, использующий информацию, содержащуюся в матрице расстояний графа, для формирования начального разбиения вершин графа. Построенное с использованием матрицы расстояний начальное разбиение используется в дальнейшем для уменьшения дерева поиска при реализации рекурсии с возвратом.

Одним из наиболее эффективных алгоритмов является алгоритм NAUTY, предложенный McKay в 1981 г. Основа алгоритма – построение поискового дерева с вершинами, являющимися упорядоченными разбиениями множеств вершин исходного графа. Начальное разбиение состоит из одного класса – самого множества вершин графа; окончательные разбиения содержат по одной вершине в каждом классе. Следующее разбие-

ние в поисковом дереве получается из предыдущего выделением некоторой вершины в отдельный класс и применением к полученному таким образом промежуточному разбиению оператора продолжения разбиения. В качестве этого оператора используется обычная процедура многократного итерирования разбиения множества вершин графа по степеням. Алгоритм содержит некоторые средства, позволяющие исключить из рассмотрения части поискового дерева. Хотя алгоритм NAUTY считается одним из наиболее эффективных, было показано, что есть категории графов, проверка изоморфизма которых имеет для него экспоненциальную сложность.

Существует другой принципиально возможный подход к решению задачи изоморфизма. Вместо снижения сложности сравнения двух графов, можно пробовать снизить общую вычислительную сложность алгоритма, сравнивая граф с большим набором графов-прототипов. Этот метод имеет сложность $O(n^2)$ вне зависимости от количества прототипов, но объем памяти, необходимой для хранения прототипов, возрастает экспоненциально с ростом числа вершин графов, проверяемых на изоморфизм. Поэтому этот метод является эффективным только для графов с небольшим числом вершин.

Все выше приведенные алгоритмы ищут точный изоморфизм графов, то есть они предъявляют биективное отображение, являющееся изоморфизмом. В последнее время растет число алгоритмов, способных находить приближенные решения задачи. Такие алгоритмы имеют полиномиальную вычислительную сложность, но они не гарантируют нахождения точного решения поставленной задачи.

Вычисление всех собственных значений и собственных векторов с необходимой точностью является задачей, обладающей значительной вычислительной сложностью. Поэтому предложен алгоритм, который в качестве эвристики может использовать не собственные значения и собственные векторы матриц, поставленных в соответствие графам, а обратные матрицы к модифицированным матрицам смежности графов и их согласованные изменения, происходящие при возмущениях. Алгоритм работает с модифицированными до положительно определенных матрицами смежности графов. Алгоритм является прямым в том смысле, что при решении задачи проверки изоморфизма графов не происходит перебора вариантов в соответствии с некоторым деревом поиска, рост которого на некотором этапе работы алгоритма мог бы стать неконтролируемым. Изоморфны графы или нет устанавливается не более чем за n итераций алгоритма, где n – число вершин в графе. В случае изоморфизма графов предъявляется один из возможных изоморфизмов.

Методы вычисления характеристического полинома

Коэффициенты характеристического полинома матрицы смежности $P(\lambda) = \det(A - \lambda E)$ обозначаются как

$$P(\lambda) = (-1)^n (\lambda^n + a_1 \lambda^{n-1} + \ldots + a_n).$$

Вычисление этих коэффициентов непосредственным разложением определителя $P(\lambda) = \det(A - \lambda E)$ на $n!$ слагаемых неэффективно. Элементами этого разложения являются выражения, полиномиально зависящие от параметра λ. На каждом этапе вычислений возникает проблема символьных вычислений хранения полиномов и действий над ними.

Метод Гаусса – основной метод вычисления числовых определителей также неэффективен при вычислении определителя, элементы которого зависят от параметра. Источником вычислительных проблем является неудобное расположение переменной λ – на главной диагонали матрицы. Первый же шаг метода Гаусса приводит к делению на элемент $a_{11} - \lambda$, и в дальнейшем элементы преобразованной матрицы будут уже не полиномами, а рациональными функциями относительно λ. Следующие шаги метода приводят к возрастанию степеней знаменателей. Необходимость в организации хранения рациональных функций и программировании действий с ними кажется тем более неоправданной, поскольку окончательный ответ: выражение для $\det(A - \lambda E)$ должно быть полиномом по λ; т. е. знаменатели дробей в конечном ответе сократятся.

Усугубляющим обстоятельством является проблема точности вычислений коэффициентов характеристического полинома. Чувствительность его корней к возмущению коэффициентов бывает весьма высокой. Возможный выход заключается в предварительном преобразовании определителя $\det(A - \lambda E)$ к виду, когда переменная λ оказывается переведенной с диагонали на крайний ряд (в столбец или в строку). При этом допускается увеличение размеров (порядка) определителя. Такое представление дает возможность разложения определителя по этому исключительному ряду и, тем самым, позволяет свести задачу к вычислению числовых определителей, когда применение метода Гаусса вполне эффективно.

Метод Леверье

Urbain Jean Joseph Le Verrier

Метод основан на формуле $Tr(A^k) = \lambda_1^k + \ldots + \lambda_n^k = t_k$, т. е. след *k*-й степени матрицы A равен *k*-й сумме Ньютона ее характеристического полинома $f(\lambda) = \det(A - \lambda E)$.

Вычисляются последовательные степени матрицы A и их следы:

$$t_1 = Tr(A^1), t_2 = Tr(A^2), \ldots, t_n = Tr(A^n) \ .$$

Неизвестные коэффициенты $a_1, a_2, ..., a_n$ в формуле $f(\lambda) = (-1)^n \times$ $\times(\lambda^n + a_1\lambda^{n-1} + \ldots + a_n)$ вычисляются по рекуррентным формулам Ньютона:

$$\begin{aligned} a_1 &= -t_1, \\ a_2 &= -(t_2 + a_1 t_1) / 2, \\ \text{для } k \le n : a_k &= -(t_k + a_1 t_{k-1} + a_2 t_{k-2} + \ldots + a_{k-1} t_1) / k. \end{aligned}$$

Для последней матрицы A^n можно не вычислять *все* элементы – достаточно вычислить элементы ее главной диагонали.

С-функция вычисления следов *t* матрицы *A* порядка *n* и коэффициентов полинома *a*.

```
    int CharPoly(int **A,int *t,int *a,int n)
{   int **B,**C,**P,i,j,k;
    B=(int**)CreateMatrix(n+1,n+1,4);
    C=(int**)CreateMatrix(n+1,n+1,4);
    for(i=1; i<=n; i++)
    for(j=1; j<=n; j++)
    B[i][j]=A[i][j];
    k=1;
    t[k]=Trace1(A,n);
    a[k]=-t[1];
    for(k=2; k<=n; k++)
 {  MatrixMul1(A,B,C,n);
    t[k]=Trace1(C,n);
    for(i=1; i<=n; i++)
    for(j=1; j<=n; j++)
    B[i][j]=C[i][j];
    a[k]=t[k];
    for(i=1; i<k; i++)
    a[k]+=a[i]*t[k-i];
    a[k]=-a[k]/k;
 }  // k
    free(B); free(C);
    return 1;
}   // CharPoly
```

Пример. Задается случайный граф с $n = 6$ и вероятностью ребра $p = 0.6$.

A						
i\j	1	2	3	4	5	6
1	0	1	0	1	0	1
2	1	0	1	1	1	1
3	0	1	0	1	1	0
4	1	1	1	0	0	1
5	0	1	1	0	0	1
6	1	1	0	1	1	0

A^6						
i\j	1	2	3	4	5	6
1	391	534	362	462	362	462
2	534	753	498	643	498	643
3	362	498	352	414	320	446
4	462	643	414	569	446	537
5	362	498	320	446	352	414
6	462	643	446	537	414	569

Следы *t* матриц A^k и коэффициенты полинома *a*:

к	*t*	*a*
1	0	0
2	22	-11
3	42	-14
4	226	4
5	730	8
6	2986	0

Вычисление полного инварианта

Рассмотрим полный инвариант матрицы смежности с максимальным и минимальным значением кода. Для *n*-вершинного графа размер кода будет n^2 бит. Для неориентированных графов достаточно хранить только треугольную часть матрицы смежности, расположенную над (или под) главной диагональю. В этом случае размер кода будет составлять $n(n-1)/2$ бит.

С-функция `MatrixNumber` вычисляет число из верхней треугольной подматрицы задаваемой матрицы A порядка n, располагая строки подматрицы в одну битовую строку.

```
    int MatrixNumber(int **A, int n) // 0 .. n-1
{   int i,j,k,s,r=0,u=0;
    for(i=n-2; i>=0; i--)  // row loop
 {  s=0;
    for(j=n-1; j>i; j--)  // column loop
  { k=A[i][j]<<(n-1-j);   // bit shift of matrix element
    s=s | k;  // number of matrix row
  } // j
```

```
    r=r | s<<u;    // complete number
    u=u+(n-1-i);   // shift of s
 } // i
    return r;
}   // MatrixNumber
```

Пример. Задается случайный граф с $n = 6$ и вероятностью ребра $p = 0.5$.

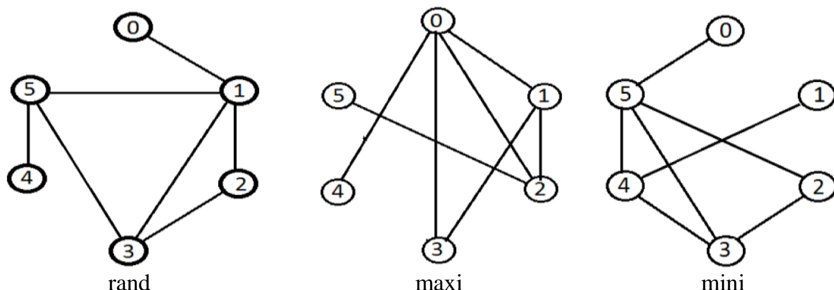


Рис. 96

Результаты вычисления показаны в таблице.

В первой строке показаны значения начального, максимального и минимального кода в десятичной и двоичной системе счисления.

Во второй строке показаны матрицы смежности.

rand=17251 rand=100001101100011	maxi=31496 max=111101100001000	mini=1199 min=10010101111
0 1 0 0 0 0 1 0 1 1 0 1 0 1 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 1 0 1 0 1 1 0	0 1 1 1 1 0 1 0 1 1 0 0 1 1 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0	0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 1 1 0 1 0 1 0 1 1 0 1 1 1 0

Потоки в сети

Основные понятия и постановка задачи

Сеть есть ориентированный граф, в котором выделены две вершины: начальная вершина *s* (*search*) и конечная вершина t (*terminal*).

Вершина *s* не обязательно должна быть истоком (вершина, у которой отсутствуют входящие ребра), а вершина *t* – стоком (вершина, у которой отсутствуют исходящие ребра), однако они рассматриваются именно в этом качестве.

Каждой дуге (i,j) приписывается некоторая *пропускная способность* $c(i,j)\geq 0$ (*capacity*, *capability*), определяющая максимальное значение потока, который может протекать по данной дуге.

Потоком в сети называют функцию $f(i,j)\geq 0$, заданную на множестве дуг E и обладающую следующими свойствами:

1) для любой дуги $(i,j)\in E$ выполняется условие $f(i,j)\leq c(i,j)$;

2) для любой вершины $v\in V, v\neq s, v\neq t,$ выполняется условие $\sum\limits_{\substack{i\in V\\(i,v)\in E}} f(i,v)=\sum\limits_{\substack{j\in V\\(v,j)\in E}} f(v,j)$. Это условие означает, что полный заходящий в v и выходящий из v потоки равны, поскольку рассматривается поток без генерации и потерь. Любой поток обладает тем свойством, что истечение вершины *s* равно притоку в вершину *t*.

Величина потока в сети есть $F=\sum\limits_{\substack{i\in V\\(i,v)\in E}} f(i,v)-\sum\limits_{\substack{j\in V\\(v,j)\in E}} f(v,j)$.

Задача заключается в определении максимального потока в сети, который можно пропустить от источника *s* к стоку *t*, и его распределение по дугам.

Теорема (Форд, Фалкерсон). *Величина каждого потока из s в t не превосходит пропускной способности минимального разреза, разделяющего s и t, причем существует поток, достигающий этого значения.*

Разрез есть множество дуг, удаление которых из сети приводит к разрыву всех путей, ведущих из *s* в *t*. Для потока f в сети поток через разрез Q определяется как $f(Q)=\sum\limits_{q\in Q} f(q)$.

Пропускная способность разреза Q есть суммарная пропускная способность составляющих его дуг $c(Q)=\sum\limits_{q\in Q} c(q)$.

Минимальный разрез, разделяющий s и t, есть произвольный разрез Q с минимальной пропускной способностью.

Теорема Форда–Фалкерсона устанавливает эквивалентность задач нахождения максимального потока и минимального разреза, однако не определяет метода их поиска.

Непосредственное решение задачи, состоящее в генерации всех подмножеств дуг, и определение, является ли очередное подмножество разрезом, приводит к экспоненциальной сложности алгоритма.

При вычитании из сети потока f получается *остаточная сеть*, в которой множество вершин остается прежним, а пропускные способности дуг изменяются по правилу $c_f(u,v) = c(u,v) - f(u,v)$.

Остаточная пропускная способность ребра показывает, насколько может быть увеличен поток по дуге без превышения его общей пропускной способности. Остаточное ребро – ребро с положительной остаточной пропускной способностью. Остаточные ребра и образуют остаточную сеть. Множество ребер остаточной сети меняется следующим образом: ребра $(u,v) \in E$ такие, что $c_f(u,v)$ уже не принадлежат E_f.

Существует другая формулировка теоремы Форда–Фалкерсона. Поток f максимален тогда и только тогда, когда в графе G_f не существует пути $p: s \to t$.

Метод решения задачи о максимальном потоке $s \to t$ был предложен Фордом–Фалкерсоном, и их метод меток составляет основу алгоритмов решения многочисленных задач, являющихся обобщениями или расширениями указанной задачи.

Метод меток заключается в последовательном (итерационном) построении максимального потока путем поиска на каждом шаге увеличивающей цепи, т. е. последовательности дуг, поток по которым можно увеличить. При этом вершины графа помечаются определенным образом.

Вводятся следующие определения. Дуга $e = (u,v)$ сети является допустимой дугой из u в v относительно потока f, если $e = (u,v)$ и $f(e) < c(e)$ (дуги первого типа будем называть их согласованными) или $e = (v,u)$ и $f(e) > 0$ (дуги второго типа – несогласованные). Второе условие говорит о том, что допустимыми являются и дуги, входящие в вершину u, по которым уже пропущен ненулевой поток.

Увеличивающая цепь – это последовательность попарно различных вершин и дуг $v_0, e_1, v_1, e_2, \dots, v_{i-1}, e_i, v_i$ такая, что $v_0 = s$, $v_i = t$, и для каждого допустимого i дуга e_i является допустимой дугой из v_{i-1} в v_i

относительно потока f. Если найдена такая цепь, то значение потока может быть увеличено на $\min q(e_i)$, где $q(e_i) = c(e_i) - f(e_i)$, если дуга первого типа, и $q(e_i) = f(e_i)$, если дуга второго типа.

В алгоритме Форда–Фалкерсона механизм построения увеличивающей цепи строго не оговаривается. В частности, может использоваться и поиск в глубину. Алгоритм является почти полиномиальным и имеет оценку $O(nmC)$, где $C = \max\{c(i,j)\}$.

Эдмондс и Карп предложили в качестве увеличивающей цепи выбирать цепь наименьшей длины – без учета пропускной способности дуг этой цепи; при этом считается, что все дуги имеют единичную длину. На практике этот выбор реализуется с помощью поиска в ширину. Такой алгоритм имеет полиномиальную оценку $O(nm^2)$.

Идея алгоритма

На каждой итерации вершины сети могут находиться в одном из трех состояний:

- вершине присвоена метка, и она просмотрена;
- вершине присвоена метка, и она не просмотрена, т. е. не все смежные с ней вершины обработаны;
- вершина не имеет метки.

На каждой итерации выбирается помеченная, но непросмотренная вершина, и происходит поиск вершины u, смежной с v, которую можно пометить. Помеченные вершины, достижимые из вершины-источника, образуют множество вершин сети *G*. Если среди этих вершин окажется вершина-сток, то это означает успешный результат поиска цепочки, увеличивающей поток; при неизменности этого множества работа заканчивается – поток изменить нельзя.

Введение в метод блокирующих потоков

Основная идея метода – алгоритм состоит из фаз (итераций), на которых поток увеличивается сразу вдоль всех кратчайших цепей определенной длины. Для этого на *i*-й фазе строится вспомогательная ациклическая (бесконтурная) сеть. Эта сеть содержит все увеличивающие цепи, длина которых не превышает k_i, где k_i – длина кратчайшего пути из *s* в *t* на *i*-й итерации. Величину k_i называют длиной вспомогательной сети.

Стратегия работы алгоритма на *i*-й итерации заключается в следующем.

Шаг 1. *Построение вспомогательной сети*

С помощью поиска в ширину происходит движение из источника сети в сток по допустимым дугам. Первоначально в очередь помещается исток. Затем, пока очередь не пуста, из нее извлекается очередная вершина u. Для этой вершины просматриваются все вершины, в которые ведут допустимые дуги из u. Если вершина v не была ранее просмотрена, то она помещается в очередь, а соответствующая дуга (u, v) добавляется во вспомогательную сеть S_k. Если вершина v уже просмотрена, то она не помещается в очередь. Однако дуга (u, v) добавляется во вспомогательную сеть S_k в том случае, когда длина кратчайшего пути от истока до текущей вершины больше (максимум на 1) длины пути от истока до вершины u, из которой в данный момент просматривается вершина v. Длина кратчайшего пути от истока до вершины v хранится как метка, значение которой присваивается вершине при записи ее в очередь. Это значение больше на единицу значения метки вершины u. Дуга $e = (u, v)$ добавляется с пропускной способностью $c_k(e) = c(e) - f(e)$, если дуга e согласованна, и $c_k(e) = f(e)$, если дуга e не согласована.

Пусть сток сети достигается и имеет метку k. Значение k становится фиксированной длиной вспомогательной сети на данной итерации. Поиск в ширину продолжается до тех пор, пока очередь непуста. Однако вершины с меткой, большей значения k, в очередь не записываются. Таким образом, вспомогательная сеть является подсетью исходной сети, содержащей все кратчайшие пути из истока в сток длины k. Если сток t не достигнут при поиске в ширину, то работа алгоритма завершается.

Шаг 2. *Поиск блокирующего потока*

Блокирующим потоком в сети называется поток, при котором любая цепь из истока в сток содержит полностью насыщенную дугу (насыщенная дуга блокирует дальнейшее увеличение величины потока по данной увеличивающей цепи). Блокирующий поток во вспомогательной сети является максимальным. В построенной вспомогательной сети длины k находится блокирующий поток (каждая увеличивающая цепь имеет длину k, ибо так строилась вспомогательная сеть). Найденный поток переносится в исходную сеть, и осуществляется переход на шаг 1 (следующая итерация).

Для построения блокирующего потока используется поиск в глубину. Пусть найден *j*-й путь из s в t и по этому пути идет поток f_j. Это значит, что по крайней мере одна дуга вспомогательной сети станет насыщенной. Удаляются все насыщенные дуги. В результате могут образо-

ваться «тупики»: вершины, не имеющие выходных дуг (кроме стока); вершины, не имеющие входных дуг (кроме источника); изолированные вершины. Удаляются из вспомогательной сети данные вершины со всеми инцидентными им дугами, что в свою очередь может привести к образованию новых «тупиков». Процесс удаления производится до тех пор, пока во вспомогательной сети не останется ни одного «тупика». Изменяются пропускные способности оставшихся дуг по формуле $c_k(e) = c_k(e) - f_j(e)$. Поиск потоков продолжается до тех пор, пока вспомогательная сеть не окажется пустой. Найденный поток $f = \sum f_j$.

После завершения работы алгоритма исходная сеть будет содержать максимальный поток.

Алгоритм имеет временную оценку $O(n^2 m)$. Итерации алгоритма начинаются с построения вспомогательной сети, строящейся обходом в ширину, до момента появления в обходе вершины-стока. Так как любой путь, не проходящий дважды через какую-либо вершину, имеет длину не более n , то именно эта величина ограничивает число вспомогательных сетей в алгоритме. Количество вершин во вспомогательной сети ограничено величиной n , а количество дуг будет не более m . При нахождении увеличивающей цепи по крайней мере одна из дуг становится насыщенной и удаляется из вспомогательной сети. Другими словами, существует не более m путей из истока в сток. Каждый путь строится за время $O(n)$, а общее время построения потока во вспомогательной сети пропорционально $O(nm)$, что дает неформальную оценку времени работы алгоритма.

Один из наиболее эффективных алгоритмов построения максимального потока в сети предложен Малхотри, Кумаром и Махешвари в 1978 г. Алгоритм является модификацией метода блокирующих потоков, предложенного Диницем. На каждой итерации на базе остаточной сети строится вспомогательная сеть тем же способом, что и в методе Диница. Затем во вспомогательной сети находится блокирующий поток. Найденный поток суммируется с текущим потоком, а остаточная сеть изменяется, и выполняется следующая итерация. После каждой итерации длина кратчайшего пути в остаточной сети от источника к стоку увеличивается. Следовательно, общее количество итераций не превышает количества вершин в сети. Алгоритм позволяет построить блокирующий поток во вспомогательной сети за время $O(n^2)$.

Приложение

Функция создания ступенчатой матрицы

```
    n – порядок матрицы, deg – степени вершин
    char** CreateMatrixDifLen(int n,int *deg,int type)
{   char **A; int i,m,r,nm;
    m=0;  // количество ребер
    for(i=0; i<n; i++)
    m=m+deg[i];
    r=n*sizeof(char*); // 32-64
    nm=n*m*type;  // размер запрашиваемой памяти
    // type - размер длины элемента
    A=(char**)malloc(r+nm);
    if(A)
 {  A[0]=(char*)(A+n); // +r !!!
    for(i=1; i<n; i++)
    A[i]=A[i-1]+deg[i-1]*type;
 }
    else
    MessageBox(0,"Failure: A=(char**)malloc=0",
    "CreateMatrixDifLen",MB_OK);
    return A;
}   // CreateMatrixDifLen
```

Функция (*с переменным количеством аргументов*) *ввода ребер и построения по ним матрицы смежности*. Вводимая нумерация вершин определяется параметром $b = 0$ или $b = 1$. Внутренняя нумерация начинается с 0.

```
    int **InEdge(int dir,int b,int n,int &m,...)
{   int *p,i,j,k,**A;
    A=(int**)CreateMatrix0(n,n,4);
    p=&m; p++;
    for(k=0; k<m; k++)
 {  i=*p-b; p++;
    j=*p-b; p++;
    A[i][j]=1;
 }  // for k
    for(i=0; i<n; i++)
```

```
    for(j=0; j<n; j++)
 {  if(A[i][j]==1)
    if(dir>=0) A[j][i]=1; else A[j][i]=-1;
 }
    return A;
}   // InEdge
```

Пример.

```
A=InEdge(-1,0,n,m,0,1,0,2,0,3,1,2,2,3,1,4,2,5,3,6,4,5);
```

Функция печати матрицы в представленном виде

```
    int PrintMatrix(int **A,int n,char *ha)
{   int i,j,k; char format[]="%3i";
    for(i=0; i<n; i++) printf(" ");
    printf(ha); printf("\n");
    printf("i\\j |");
    for(i=0; i<n; i++) printf(format,i); printf("\n");
    for(i=0; i<3*n+5; i++) printf("-"); printf("\n");
    for(i=0; i<n; i++)
 {  printf("%3i |",i); // =6
    for(j=0; j<n; j++)
    k=printf(format,A[i][j]);
    printf("\n");
 }  // i j
    return k;
}   // PrintMatrix
```

```
          in A

i\j |  0  1  2  3  4  5  6  7  8  9
-----------------------------------
  0 |  0 -1  1  0  0  0  0  0  0  0
  1 |  1  0  0  1 -1  0  0  0  0  0
  2 | -1  0  0  1  0  0  0  0  0  0
  3 |  0 -1 -1  0 -1  0  0  0  0  0
  4 |  0  1  0  1  0  1  0  0  0  0
  5 |  0  0  0  0 -1  0 -1  0 -1  0
  6 |  0  0  0  0  0  1  0  1  0 -1
  7 |  0  0  0  0  0  0 -1  0  1  1
  8 |  0  0  0  0  0  1  0 -1  0  0
  9 |  0  0  0  0  0  0  1 -1  0  0
```

Функция вычисления определителя матрицы с выбором главного элемента и без выбора

```
    double Det(double **A, int n)
{   int i,j,k,imax,select=1; // select=1 lead element
    double max,h,det,*r=(double*) calloc(n,8);
    for(i=0; i<n; i++)
 {  if(select)
  { max=abs(A[i][i]);  imax=i;
    for(j=i+1; j<n; j++)
  { h=abs(A[j][i]);
    if(h>max) {max=h; imax=j; }
  } // for j
    if(imax!=i)// swap rows
    for(j=i; j<n; j++)
 {  h=A[i][j];
    A[i][j]=A[imax][j];
    A[imax][j]=-h; }
 }  // select
    for(j=n-1; j>i; j--)
    A[i][j]=A[i][j]/A[i][i];
    for(k=i+1; k<n; k++)
    for(j=i+1; j<n; j++)
    A[k][j]=A[k][j]-A[k][i]*A[i][j];
 }  // for i
    printf("\nDet Double\n");
    for(i=0; i<n; i++)
 {  for(j=0; j<n; j++)
    printf("%5.1f ",A[i][j]);
    printf("\n");
 }  // i j
    free(r);
    det=1;
    for(i=0; i<n; i++)
    det=det*A[i][i];
    return det;
}   // Det
```

Функция вычисления алгебраического дополнения матрицы

```
    double Cofactor(double **A, int n,int p,int q)
{   int i,j,k,imax,select=1; double max,h,det;
    for(i=0; i<n; i++)
```

```
  {  if(i==p) continue;
     if(select)
   { max=abs(A[i][i]);  imax=i;
     for(j=i+1; j<n; j++)
   { if(j==q) continue;
     h=abs(A[j][i]);
     if(h>max) {max=h; imax=j; }
   } // for j
     if(imax!=i)// swap rows
     for(j=i; j<n; j++)
   { if(j==q) continue;
     h=A[i][j];
     A[i][j]=A[imax][j];
     A[imax][j]=-h;
   } // j
  }  // select
     for(j=n-1; j>i; j--)
  {  if(j==q) continue;
     A[i][j]=A[i][j]/A[i][i];
  }  // j
     for(k=i+1; k<n; k++)
  {  if(k==p) continue;
     for(j=i+1; j<n; j++)
   { if(j==q) continue;
     A[k][j]=A[k][j]-A[k][i]*A[i][j];
   } // j
  }  // k
  }  // for i
     det=1;
     for(i=0; i<n; i++)
  {  if(i==p || i==q) continue;
     det=det*A[i][i];
  }  // i
     return det;
}    // Cofactor
```

Многие рисунки выполнены авторской программой рисования графов.

Задачи

1. Написать функции взаимного преобразования 4-х способов представления неориентированного графа.
2. Написать функции взаимного преобразования 4-х способов представления ориентированного графа.
3. Напишите функции преобразования 4-х способов представления ориентированного графа в неориентированный.
4. Опишите четырьмя способами неориентированные графы, представленные на рис. 97.

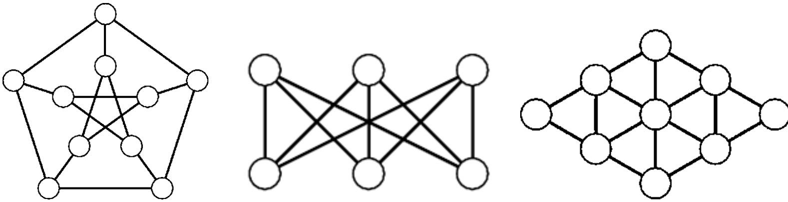


Рис. 97

5. Опишите четырьмя способами ориентированные графы, представленные на рис. 98.

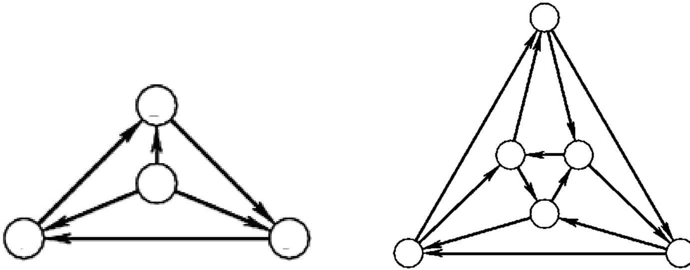


Рис. 98

6. Клетки поля $n \times n$ пронумерованы построчно от 1 до n^2. Считая клетку поля вершиной графа, описать его четырьмя способами. Вершины графа являются смежными, если клетки, которым они соответствуют: соседние по горизонтали и по вертикали; соседние по диагонали.

7. Написать программы ввода описаний графа для каждого из четырех способов его представления в памяти компьютера.
8. Написать функции, выполняющие операции над графами: объединения, соединения, произведения, отождествления, стягивания ребра.
9. Написать функции обхода графа в глубину для всех 4-х способов представления и для всех начальных вершин и построить деревья обхода.
10. Написать функции обхода графа в ширину для всех 4-х способов представления и для всех начальных вершин и построить деревья обхода.
11. Написать функции вычисления кратчайшего расстояния от начальной вершины до всех остальных вершин с формированием массива расстояний.
12. Для случайного графа написать функции вычисления эксцентриситетов, диаметра, радиуса, центра для различных способов представления.
13. Написать функцию перечисления всех подграфов заданного графа.
14. Перечислить все попарно неизоморфные графы с пятью вершинами. Графы изоморфны тогда и только тогда, когда их матрицы смежности получаются друг из друга одинаковыми перестановками строк и столбцов.
15. Даны два графа, описанные с помощью матриц смежности. Написать программу преобразования (путем одинаковых перестановок строк и столбцов) одной матрицы смежности в другую или установить, что это сделать нельзя.
16. Написать функции описания дополнения графа, описанного каким-либо компьютерным представлением.
17. Написать функцию вычисления, степеней всех вершин связного графа, описанного каким-либо компьютерным представлением, перенумеровать вершины графа в соответствии с убыванием значения степени и описать новое представление.
18. Написать функцию поиска максимального и минимального значений степеней вершин неориентированного и ориентированного графа.
19. Написать рекурсивные и нерекурсивные функции поиска в глубину и ширину в графе (орграфе).
20. Сгенерировать случайный связный граф. Найти вершины с максимальным и минимальным значениями степеней. Выполнить поиск в глубину и ширину, начиная с этих вершин.
21. Граф называется регулярным, если степени всех его вершин равны. Написать функцию построения регулярного графа.

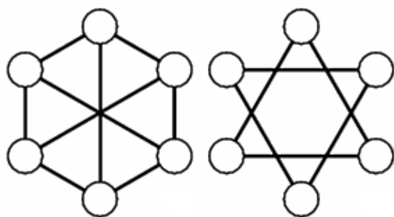


Рис. 99

22. Даны два графа на рис. 99, представленные матрицами смежности или иными способами. Написать программы формирования матрицы смежности или иные формы представления для графа, получаемого в результате выполнения операций объединения, соединения, произведения и композиции.
23. Для заданных вершин u и v найти все цепи: простые цепи и их связывающие.
24. Для заданной вершины, если существуют, построить цикл и простой цикл.
25. Построить граф, центр которого состоит из одной вершины, совпадает с множеством всех вершин, состоит из k вершин и не совпадает с множеством всех вершин.
26. Для произвольного графа, используя поиск в ширину для нахождения циклов нечетной длины, проверить, является ли он двудольным (теорема Кёнига).
27. Используя поиск в глубину и поиск в ширину, определить количество компонент связности произвольного графа.
28. Написать программу нахождения кратчайшего пути между вершинами u и v произвольного графа.
29. Написать программу поиска всех достижимых вершин из заданной вершины для произвольного ориентированного графа.
30. При поиске в глубину или в ширину в связном графе образуется дерево – остовный подграф. Написать функцию создания матрицы смежности и списков смежности этих деревьев.
31. Генерировать связный граф. Удалить все ребра, не нарушающие связности графа.
32. Вычислить все различные деревья с семью вершинами (их 11).
33. Вычислить все различные деревья с восемью вершинами (их 23).
34. Вычислить все различные помеченные деревья с четырьмя вершинами (их 16).

35. Написать функцию – процедуру перечисления деревьев графа для случая, когда он представлен в памяти с помощью списков связей.
36. Найти такое значение n_*, что при $n > n_*$ в полном графе K_n есть два остовных дерева без общих ребер.
37. Генерировать случайный взвешенный граф. Найти остовное дерево с минимальным весом методами Краскала, Прима и Борувки.
38. Изменить метод Краскала и Прима для построения остовного дерева с максимальным весом.
39. Написать функции перевода описания дерева: из матрицы смежности в числовую последовательность и из числовой последовательности в список ребер.
40. Написать функцию решения задачи Штейнера на графах методом перебора. На плоскости своими координатами задано произвольное конечное множество точек. Требуется соединить их непрерывными линиями так, чтобы любые две точки были связаны либо непосредственно, либо через другие точки и соединяющие их отрезки, при этом общая сумма длин отрезков должна быть минимальной. На множестве точек можно построить полный граф. При этом вес каждого ребра равен евклидову расстоянию между соответствующими точками. Если не допускаются пересечения любых двух линий в точках вне заданного множества, то задача сводится к нахождению остова минимального веса.
41. Генерировать произвольный граф. Написать функцию определения связности графа.
42. Генерировать произвольный граф. Определить его блоки и точки сочленения (шарниры). Построить граф блоков и граф шарниров.
43. Генерировать произвольный граф, описываемый матрицей смежности. Написать функцию формирования матриц смежности для графов блоков и точек сочленения (шарниров).
44. Написать функцию нахождения всех мостов графа.
45. Генерировать произвольный граф. Найти вершинно-непересекающиеся цепи между заданными вершинами.
46. Написать функцию поиска наибольшего количества вершинно-непересекающихся цепей между всеми парами вершин.
47. Написать функцию поиска наибольшего числа реберно-непересекающихся цепей между всеми парами вершин.
48. Дан ориентированный граф G. Написать функцию нахождения его графа конденсаций.

49. Написать функцию нахождения базы графа.

    Пояснение. База $P*$ конденсации $G*$ графа G состоит из таких вершин графа $G*$, в которые не заходят ребра. Следовательно, базы графа G можно строить так: из каждой сильно связной компоненты графа G, соответствующей вершине базы $P*$ конденсации $G*$, надо взять по одной вершине – это и будет базой графа G.

50. Граф называется транзитивным, если из существования дуг (u, v) и (v, w) следует существование дуги (u, w). Транзитивным замыканием графа $G = (V, E)$ является граф $G = (V, E \cup E')$, где E' – минимально возможное множество дуг, необходимых для того, чтобы граф G был транзитивным. Написать функцию нахождения транзитивного замыкания произвольного графа G.

51. Написать функцию поиска эйлерова цикла для (не)ориентированного графа.

52. Написать функцию поиска в неориентированном графе двойного эйлерова цикла, каждое ребро которого проходится ровно два раза в противоположных направлениях.

53. Написать функцию поиска гамильтонова цикла для ориентированного графа.

54. Генерировать последовательность случайных графов с увеличением n и разными m.

55. Генерировать случайные графы и проверить их на «эйлеровость» и «гамильтоновость».

56. Генерировать случайный граф. Найти фундаментальные циклы относительно некоторого остова.

57. Используя представление фундаментальных циклов в виде множества ребер и операцию симметрической разности, написать программу генерации всех циклов графа.

58. Генерировать случайный граф. Вычислить матрицы фундаментальных циклов и коциклов.

59. Написать функцию поиска гамильтонова цикла с помощью симметрической разности некоторого подмножества множества фундаментальных циклов.

60. Написать функцию поиска эйлерова цикла с помощью симметрической разности некоторого подмножества множества фундаментальных циклов.

61. Написать функцию нахождения числа вершинного покрытия, числа реберного покрытия, вершинное число независимости и реберное число независимости.

62. Написать функцию нахождения всех максимальных независимых множеств вершин.
63. Написать функцию нахождения всех максимальных независимых множеств вершин графа путем генерации всех подмножеств множества вершин и проверки каждого подмножества на независимость и максимальность.
64. Задача о «восьми ферзях» является задачей об отыскании максимального независимого множества. Представить «шахматную» доску в виде графа с n^2 вершинами, которые смежные, если клетки находятся на одной горизонтали, вертикали или диагонали. Решение можно получить путем поиска максимальных независимых множеств и путем прямого перебора возможных расстановок. Сравнить логику и скорость решения задач.
65. Написать функцию нахождения клик графа.
66. Написать функцию нахождения независимого множества вершин графа.
67. В произвольном связном графе нет треугольников. Написать программу поиска клик в этом графе.
68. Построить дополнение G^* графа G. Показать численно, что клики графа G соответствуют максимальным независимым множествам вершин графа G^*.
69. Написать функцию нахождения числа доминирования.
70. Дана матрица A, элементы которой равны 0 или 1. Путем перестановки столбцов преобразовать ее к блочному виду.
71. Написать функцию решения задачи о поиске всех разбиений графа. Разбиение графа – представление исходного графа в виде множества подмножеств вершин по определенным правилам.
72. Написать функцию решения задачи о поиске наименьшего разбиения графа.
73. Написать функцию поиска всех покрытий графа.
74. Написать функцию поиска покрытия с минимальным суммарным весом взвешенного графа.
75. Написать функцию проверки планарности графа.
76. Написать функцию поиска плоской укладки графа.
77. Написать функцию вычисления хроматического числа двудольного графа.
78. Написать функцию определения, раскрашиваем ли граф двумя красками.

79. Разработать программу реализации следующего эвристического алгоритма раскраски вершин графа. Первоначально вершины сортируются в порядке не возрастания их степеней, а затем используется жадный метод правильной раскраски вершин графа.
80. Написать рекурсивный и нерекурсивный варианты процедуры вывода кратчайшего пути.
81. Используя алгоритм Флойда–Уоршалла, найти кратчайшие пути между всеми парами вершин для случайных графов.
82. На основе алгоритма Флойда–Уоршалла написать функцию поиска в графе циклов с отрицательным суммарным весом.
83. Написать функцию поиска k-кратчайших простых путей между двумя заданными вершинами графа.
84. В неориентированном графе веса имеют ребра и вершины. Написать функцию поиска кратчайших путей между вершинами графа.
85. В неориентированном графе неотрицательные веса имеют ребра и вершины. Написать функцию поиска пути между заданной парой вершин с минимальным суммарным весом.
86. Существует несколько равных кратчайших путей. Написать функцию поиска всех кратчайших простых путей между вершинами графа.
87. Написать функцию поиска самого длинного пути в ациклическом ориентированном графе.
88. Написать функции рисования произвольного графа и/или дерева.
89. Написать функции динамического рисования алгоритмов графа и/или дерева, т. е. мультипликацию.
90. Написать функции, выполняющие унарные операции над графами.
91. Написать функции, выполняющие бинарные операции над графами.
92. По заданной колоде реконструировать граф.
93. Написать функцию проверки изоморфизма деревьев.
94. Написать функцию генерирования всех изоморфных графов.
95. Написать функции проверки неполных инвариантов графа.
96. Написать функции проверки полных инвариантов графа.
97. Написать функцию, которая генерирует неизоморфный граф заданному графу, но у которых совпадают все или избранные неполные инварианты.
98. Построить и нарисовать плоский граф для каждого $n \geq 4$ с максимальным m.

Литература

1. *Асанов М.О.*, *Баранский В.А.*, *Расин В.В.* Дискретная математика: графы, матроиды, алгоритмы. Ижевск: НИЦ «Регулярная и хаотическая динамика», 2001.
2. *Басакер Р.*, *Саати Т.* Конечные графы и сети. М.: Наука, 1974.
3. *Белов В.В.*, *Воробьев Е.М.*, *Шаталов В.Е.* Теория графов. М.: Высшая школа, 1976.
4. *Берж К.* Теория графов и ее применения. М.: ИЛ, 1962.
5. *Гудман С.*, *Хидетниеми Дж.*, *Ульман Дж.* Введение в разработку и анализ алгоритмов. М.: Мир, 1981.
6. *Демидович Б.П.*, *Марон И.А.* Основы вычислительной математики. М.: Наука, 1966; Физматлит, 2008.
7. *Евстигиеев В.А.* Применение теории графов в программировании. М.: Наука, 1985.
8. *Евстигнеев В.А.*, *Мельников Л.С.* Задачи и упражнения по теории графов и комбинаторике. Новосибирск: НГУ, 1981.
9. *Evstigneev V.A.*, *Kasyanov V.N.* Dictionary of graphs in computer science. Novosibirsk: OOO «Sibirskoe Nauchnoe Izdatel'stvo», 2009.
10. *Емеличев В.А.*, *Мельников О.И.*, *Сарванов В.И.*, *Тышкевич Р.И.* Лекции по теории графов. 2-е изд. М.: Книжный дом «Либроком», 2009.
11. *Зыков А.А.* Основы теории графов. М.: Вузовская книга, 2004.
12. *Иванов Б.Н.* Дискретная математика. Алгоритмы и программы. М.: Физматлит, 2007.
13. *Касьянов В.И.*, *Евстигнеев В.А.* Графы в программировании: обработка, визуализация и применение. СПб.: БХВ-Петербург, 2003.
14. *Кнут Д.Э.* Искусство программирования. Т. 1, 3. М.: Издательский дом «Вильямс», 2003.
15. *Колчин В.Ф.* Случайные графы. М.: Физматлит, 2004.
16. *Кормен Т.*, *Лейзерсон Ч.*, *Ривест Р.*, *Штайн К.* Алгоритмы: построение и анализ. М.: Издательский дом «Вильямс», 2007.
17. *Кристофидес Н.* Теория графов. Алгоритмический подход. М.: Мир, 1978.
18. *Липский В.* Комбинаторика для программистов. М.: Мир, 1988.
19. *Ловас Л.*, *Пламмер М.* Прикладные задачи теории графов. Теория паросочетаний в математике, физике, химии. М.: Мир, 1998.
20. *Макконнелл Дж.* Основы современных алгоритмов. М.: Техносфера, 2004.
21. *Новиков Ф.А.* Дискретная математика для программистов. СПб.: Питер, 2000.
22. *Носов В.А.* Комбинаторика и теория графов. М.: МГУ, 1999.
23. *Окулов С.М.* Дискретная математика. Теория и практика решения задач по информатике. М.: БИНОМ, 2008.
24. *Оре О.* Графы и их применение. М.: Едиториал УРСС, 2002.
25. *Оре О.* Теория графов. М.: Наука, 1980.
26. *Прут В.В.* Алгоритмы и структуры данных на языке С. М.: МФТИ, 2016.
27. *Райгородский А.М.* Модели случайных графов. М.: МЦНМО, 2011.

28. *Рейнгольд Э.*, *Нивергельт Ю.*, *Део Н.* Комбинаторные алгоритмы. Теория и практика. М.: Мир, 1980.
29. *Свами М.*, *Тхуласираман К.* Графы, сети и алгоритмы. М.: Мир, 1984.
30. *Сэджвик Р.* Фундаментальные алгоритмы на C++. Алгоритмы на графах. М.: Диасофт, 2002.
31. *Татт У.* Теория графов. М.: Мир, 1988.
32. *Уилсон Р.* Введение в теорию графов. М.: Мир, 1977.
33. *Хаггарти Р.* Дискретная математика для программистов. М.:
34. *Харари Ф.* Теория графов. 2-е изд. М.: Едиториал УРСС, 2003.
35. *Харари Ф.*, *Палмер* Э. Перечисление графов. М.: Мир, 1977.
36. *Bollobas B*. Random graphs. Cambridge: Cambridge University Press, 2001.
37. *Janson* S. Random Graphs. John Wiley & Sons, Inc. 2000.
38. Handbook of Graph Theory. Second edition. CRC Press, Taylor & Francis Group, 2014.

Содержание

Введение **3**

Основные определения **5**

Степень вершины 6
Регулярный граф 8
Полный граф 9
Турниры 10
Двудольный граф 11
Рёберный граф 13
Операции над графами 16
Маршрут, цепь, путь, цикл 21
Изоморфизм графов 23
Реконструируемость 26

Представление графов в компьютере **27**

Матрица смежности графа 27
Матрица весов графа 29
Списки смежности вершин графа 29
Список ребер графа 30
Матрица инцидентности графа 30

Обходы графа **33**

Обход в глубину 33
Обход в ширину 36
Графы правильных многогранников 38
Граф Муна–Мозера 44

Связность графов **46**

Вершинная связность $\kappa(G)$ 46
Реберная связность $\lambda(G)$ 47
Алгоритм нахождения блоков и шарниров графа 48

Маршруты и связность в орграфах 51

Кратчайшие пути в графе 53

Алгоритм Дейкстры 54

Алгоритм Беллмана–Форда 58

Кратчайшие пути для всех пар вершин – алгоритм Флойда–Уоршелла 60

Транзитивное замыкание 66

Метрические характеристики графа 68

Циклы в графе 71

Фундаментальное множество циклов 71

Ориентированные ациклические графы 75

Эйлеров цикл 76

Двойной эйлеров цикл 79

Гамильтонов цикл 80

Задача коммивояжера 81

Существование гамильтоновых маршрутов 82

Алгоритм нахождения гамильтоновых циклов 85

Топологическая сортировка 88

Алгоритм топологической сортировки графа DAG методом удаления истоков 89

Алгоритм топологической сортировки графа DAG на основе поиска в глубину 90

Остовные деревья 95

Построение всех остовных деревьев графа 95

Число различных остовов неполного невзвешенного графа 96

Алгоритм генерации различных остовов невзвешенного графа 97

Минимальное остовное дерево 100

Алгоритм Прима 100

Алгоритм Крускала 103

Алгоритм Борувки 105

Построение графа с заданным набором степеней вершин 108

Случайные графы 117

Генератор случайных графов (случайные ребра)...............................118

Связность, диаметр и радиус в случайном графе 121

Распределение степеней в случайном графе 122

Клики, независимые множества, вершинные покрытия 124

Доминирующее множество 126

Ядро графа 127

Алгоритм Магу для определения множества внутренней устойчивости графа 128

Алгоритм Магу для определения множества внешней устойчивости...129

Классические алгоритмы решения задачи нахождения клик 131

Алгоритм Брона–Кербоша для нахождения максимального независимого множества графа..132

Клики графа Муна–Мозера 139

Эвристические алгоритмы поиска всех клик.......................................140

Паросочетания и реберные покрытия...145

Алгоритм поиска паросочетания в двудольных графах......................147

Планарность...149

Формула Эйлера...150

Критерии планарности 151

Алгоритм укладки графа на плоскости...153

Изображение ребер плоского графа прямыми линиями 155

Скрещивание $cr(G)$ 156

Раскраски графа 157

Вершинная раскраска 157

Реберная раскраска 158

Нижние оценки $\chi(G)$ 159

Верхние оценки $\chi(G)$ 159

Хроматический полином $\chi(G,x)$ 160

Оптимальная независимая раскраска 164
Раскраска планарных графов 166
Эвристические (жадные) алгоритмы раскраски 167
Перекраска 172
Рисование графов 174

Изоморфизм графов 177

Инварианты графа 181
Методы вычисления характеристического полинома 187
Вычисление полного инварианта 189

Потоки в сети 191

Введение в метод блокирующих потоков 193

Приложение 196

Задачи 200

Литература 207

Учебное издание

Прут Вениамин Вениаминович

ГРАФЫ.
АЛГОРИТМЫ НА ЯЗЫКЕ С

Редактор *Н. Е. Кобзева*. Корректор *О. П. Котова*
Компьютерная верстка *Н. Е. Кобзева*
Дизайн обложки *Н. Е. Кобзева*

Подписано в печать 19.04.2017. Формат 60 × 84 $^1/_{16}$. Усл. печ. л. 13,3
Уч.-изд. л. 12,1. Тираж 200 экз. Заказ № 120.

Федеральное государственное автономное образовательное
учреждение высшего образования
«Московский физико-технический институт (государственный университет)»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-58-22, e-mail: rio@mipt.ru

Отдел оперативной полиграфии «Физтех-полиграф»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-84-30, e-mail: polygraph@mipt.ru

Для заметок

В. В. Прут

ГРАФЫ. АЛГОРИТМЫ НА ЯЗЫКЕ С

ISBN 978-5-7417-0633-6