Тема 3.2

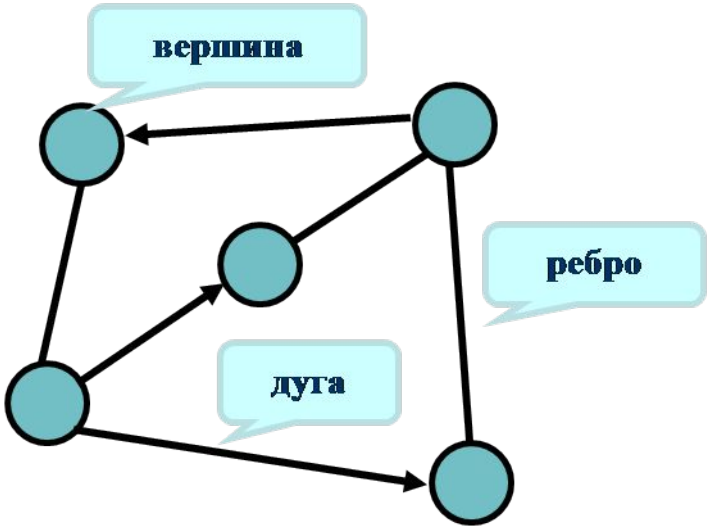
Граф. Обходы графа

Содержание

Во многих задачах, встречающихся в компьютерных науках, математике, технических дисциплинах часто возникает необходимость наглядного представления отношений между какими–либо объектами.

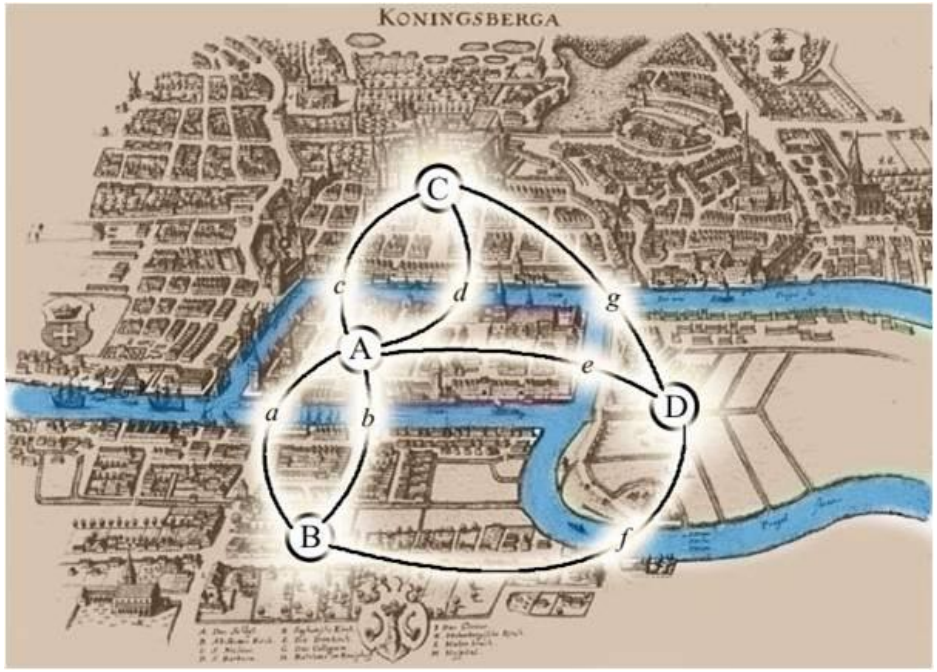
Ориентированные и неориентированные графы – естественная модель для таких отношений.

Граф – это совокупность конечного числа точек, называемых вершинами графа, и попарно соединяющих некоторые из этих вершин линий, называемых ребрами или дугами графа.



Первая работа по теории графов, принадлежащая известному швейцарскому математику Л.Эйлеру, появилась в 1736г., связанная с решением известной головоломки о мостах Кёнигсберга.

Толчок к развитию теория графов получила на рубеже XIX и XX столетий, когда резко возросло число работ в области топологии и комбинаторики.

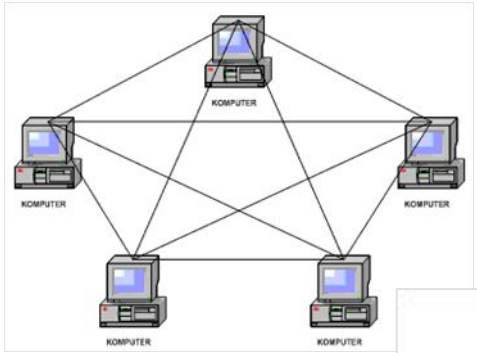


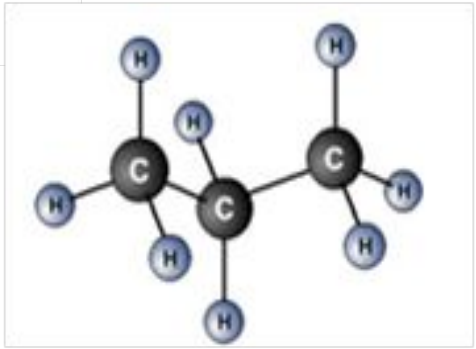
В настоящее время графы эффективно используются в

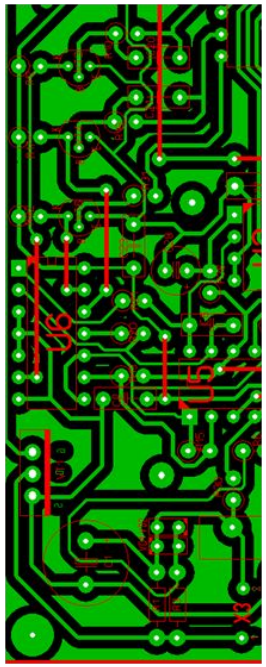
- ✔ теории планирования и управления,
- ✔ теории расписаний,
- ✔ социологии,
- ✔ экономике,
- ✔ биологии,
- ✔ медицине,
- ✔ географии.

Широкое применение находят графы в таких областях, как

- ✔ программирование,
- ✔ электроника,
- ✔ в решении вероятностных и комбинаторных задач, математических головоломок и др.







1. Графы

Определения

Ориентированный граф определяется как пара (*V, E*), где *V* – конечное множество, а *E* – бинарное отношение на *V*, т.е. подмножество множества *V*×*V.*

- Множество *V* называют ***множеством вершин*** графа; его элемент называют ***вершиной*** графа.
- Множество *E* называют ***множеством рёбер*** графа; его элементы называют ***рёбрами***.

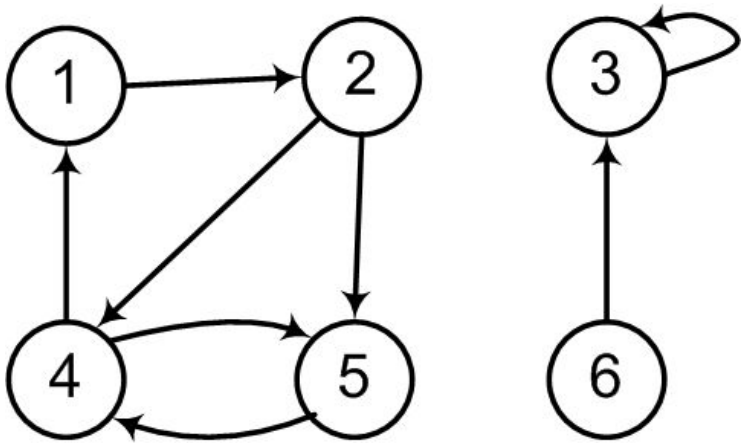
Граф может содержать рёбра–циклы – ***петли***, соединяющие вершину с собой.

Ориентированный граф иногда для краткости называют *орграфом*.

Определения

На рисунке показан ориентированный граф с множеством вершин {1, 2, 3, 4, 5, 6}.

- ✔ Вершины изображены кружками, а рёбра стрелками.
- ✔ V = {1, 2, 3, 4, 5, 6}
- ✔ E = {(1,2), (2,4), (2,5), (3,3), (4,1), (4,5), (5,4), (6,3)}.
- ✔ Ребро (3,3) является петлей.



Определения

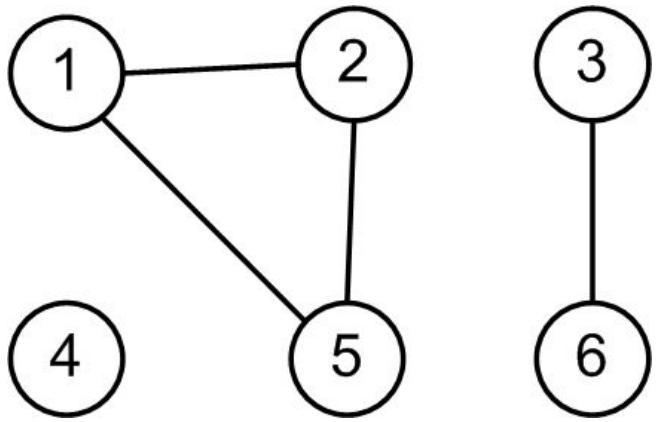
В неориентированном графе $G = (V, E)$ множество рёбер E состоит из неупорядоченных пар вершин:

- парами являются множества (u, v), где $u, v \in V$ и $u \neq v$.

В общем случае, неориентированный граф не может содержать петель, и каждое ребро состоит из двух различных вершин («соединяя» их).

На рисунке изображен неориентированный граф с множеством вершин {1, 2, 3, 4, 5, 6}.

- ✔ $V = \{1, 2, 3, 4, 5, 6\}$
- ✔ $E = \{(1,2), (1,5), (2,5), (3,6)\}$.



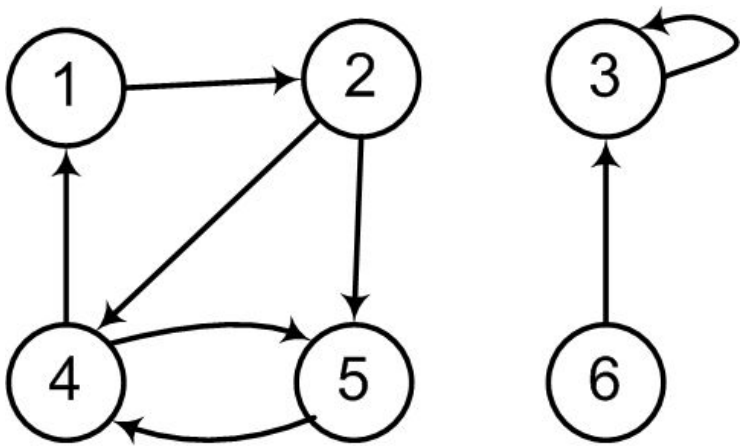
Определения

Многие понятия параллельно определяются для ориентированных и неориентированных графов (с соответствующими изменениями).

Про ребро (*u, v*) ориентированного графа говорят, что оно ***выходит*** из вершины *u* и ***входит*** в вершину *v*.

Например, на рисунке имеется

- два ребра, выходящих из вершины 2
  - ✔ рёбра (2,4), (2,5)
- и одно ребро, в неё входящее
  - ✔ рёбро (1,2).

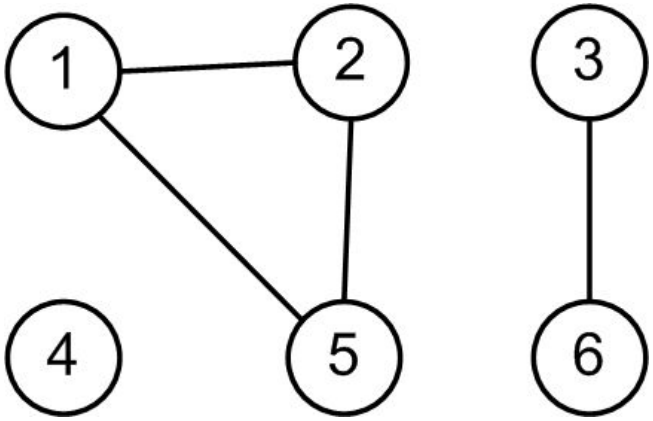


Определения

Про ребро (*u, v*) неориентированного графа говорят, что оно ***инцидентно вершинам*** *u* и *v*.

Например, на рисунке есть два ребра, инцидентные вершине 2

- ✔ рёбра (1,2) и (2, 5).



Определения

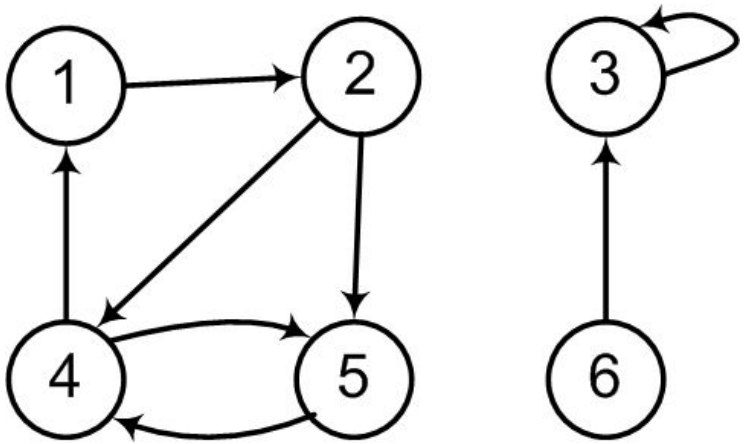
Если в графе *G* имеется ребро (*u, v*), говорят, что вершина *v* ***смежна с вершиной*** *u*.

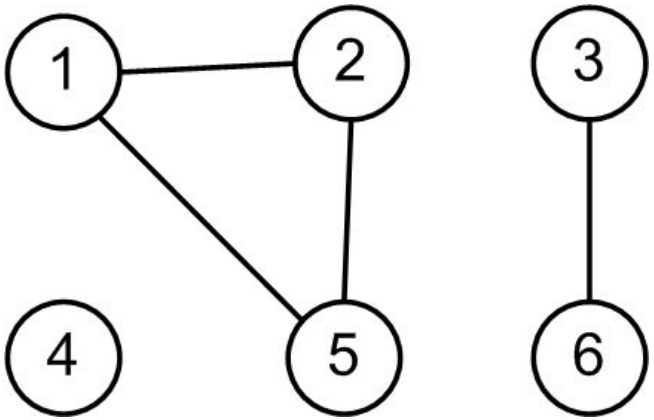
Для неориентированных графов отношение смежности является симметричным, но для ориентированных графов это не обязательно.

Если вершина *v* смежна с вершиной *u* в ориентированном графе, пишут $u \rightarrow v$.

Для обоих рисунков

- ✔ вершина 2 является смежной с вершиной 1,
- ✔ но лишь во втором из них вершина 1 смежна с вершиной 2 (в первом случае ребро (2, 1) отсутствует в графе).





Определения

Граф $G' = (V', E')$ называют ***подграфом*** графа $G = (V, E)$, если $E' \subseteq E$ и $V' \subseteq V$.

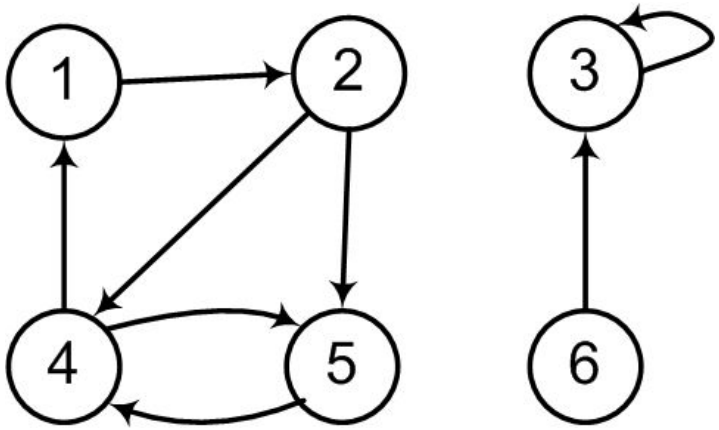
Если в графе G = (*V, E*) выбрать произвольное множество вершин *V* ', то можно рассмотреть его подграф, состоящий из этих вершин и всех соединяющих их рёбер,

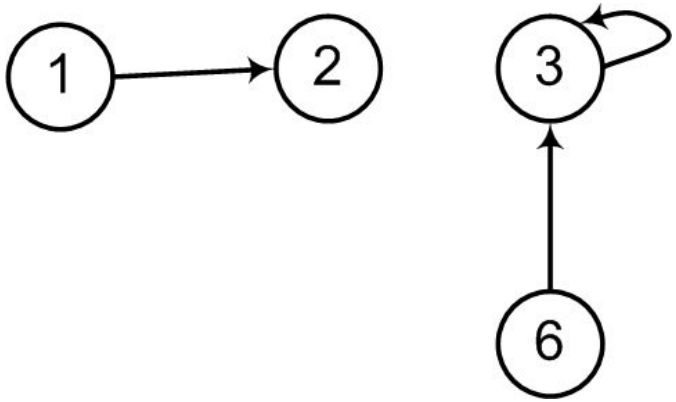
- ✔ т.е. граф *G*' = (*V* ', *E* '), для которого

$$E' = ((u, v) \in E : u, v \in V').$$

Например, для графа на рисунке с множеством вершин {1,2,3,6} можно выделить подграф

- ✔ с множеством вершин {1,2,3,6}
- ✔ и тремя ребрами (1,2), (3,3), (6,3).

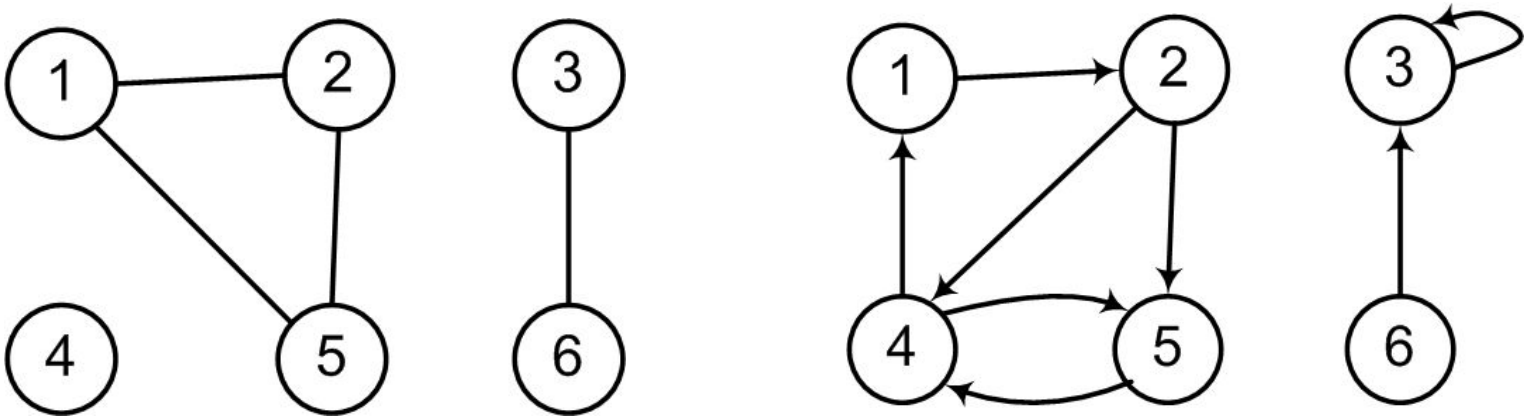




Определения

Степенью вершины в **неориентированном графе** называется число инцидентных ей рёбер.

Например, для графа на рисунке степень вершины 2 равна 2.



Для **ориентированного графа** различают ***исходящую степень***, определяемую как число выходящих из неё рёбер, и ***входящую степень***, определяемую как число входящих в неё рёбер. Сумма исходящей и входящей степеней называется ***степенью вершины***.

Например, вершина 2 в графе на рисунке имеет входящую степень 1, исходящую степень 2 и степень 3.

Определения

Путь длины *k* из вершины, *u* в вершину *v* определяется как последовательность вершин

$$\langle v_0, v_1, v_2, \ldots, v_k \rangle,$$

в которой $v_0 = u$, $v_k = v$ и $(v_{i-1}, v_i) \in E$ для всех $i = 1,2,\ldots,k$.

- Таким образом, путь длины *k* состоит из *k* рёбер.
- Этот путь содержит
  - ✔ вершины $v_0, v_1, \ldots, v_k$
  - ✔ и рёбра $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$
- Вершину v_0 называют ***началом пути***,
- вершину v_k – его ***концом***;
- говорят, что путь ведёт из v_0 в v_k.

Если для данных вершин *u* и *u'* существует путь *p* из *u* в *u',* то говорят, что ***вершина u' достижима*** из *u* по пути *p*.

- В этом случае мы пишем $u \xrightarrow{p} u'$ (для ориентированных графов) .

Определения

Путь называется ***простым***, если все вершины в нём различны.

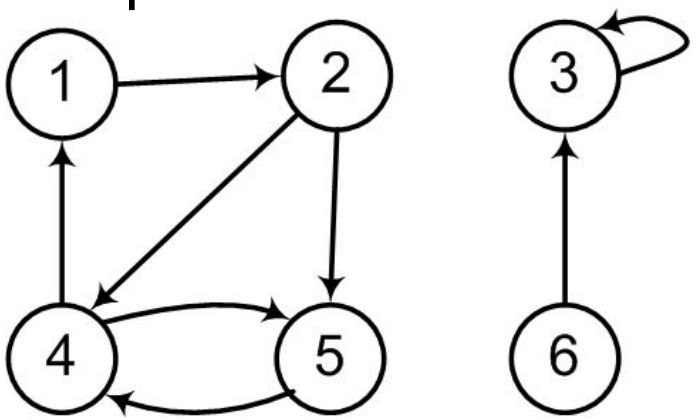
Например, на рисунке есть

- ✔ простой путь $\langle 1,2,5,4 \rangle$ длины 3,
- ✔ а также путь $\langle 2,5,4,5 \rangle$ той же длины, не являющийся простым.

Подпуть пути $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ получится, если мы возьмём некоторое количество идущих подряд вершин этого пути,

- ✔ т.е. последовательность $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ при некоторых *i, j*, для которых $0 \le i \le j \le k$.

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



Определения

Циклом в ориентированном графе называется путь,

- в котором начальная вершина совпадает с конечной
- и который содержит хотя бы одно ребро.

Цикл называется ***простым***, если в нём нет одинаковых вершин (кроме первой и последней),

✔ т.е. если все вершины различны.

Петля является циклом длины 1.

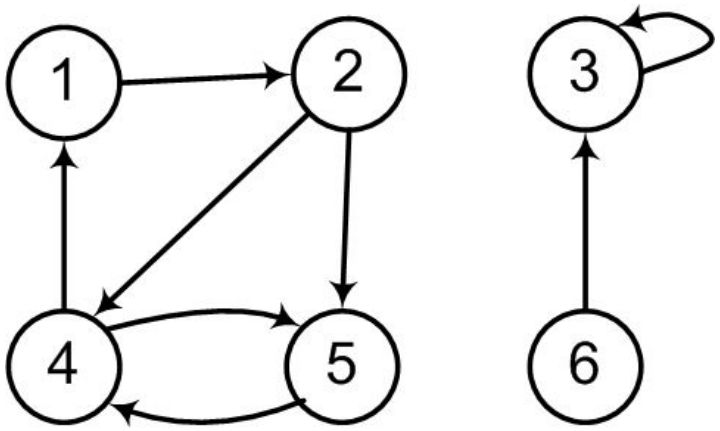
Мы отождествляем циклы, отличающиеся сдвигом вдоль цикла:

- один и тот же цикл длины *k* может быть представлен *k* различными путями (в качестве начала и конца можно взять любую из *k* вершин).

Определения

Например, на рисунке

- ✔ пути $\langle 1,2,4,1 \rangle$, $\langle 2,4,1,2 \rangle$ и $\langle 4,1,2,4 \rangle$ представляют один и тот же цикл.
- ✔ Этот цикл является простым,
- ✔ в то время как цикл $\langle 1,2,4,5,4,1 \rangle$ таковым не является.
- ✔ Есть цикл $\langle 3,3 \rangle$, образованный единственным ребром–циклом (3,3).



Определения

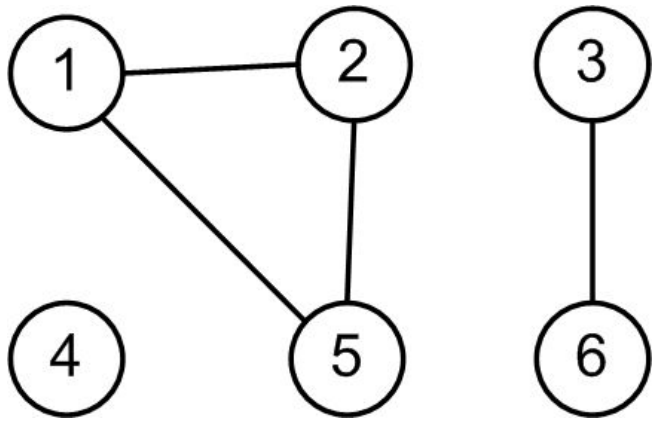
Ориентированный граф, не содержащий петель, называется ***простым***.

В неориентированном графе путь $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ называется (простым) ***циклом***, если

- $k \geq 3$,
- и все вершины различны.

Например, на рисунке имеется простой цикл $\langle 1,2,5,1 \rangle$.

Граф, в котором нет циклов, называется ***ациклическим***.



Определения

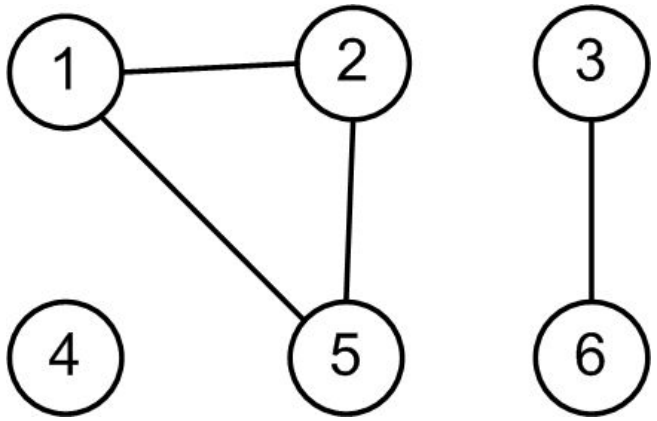
Неориентированный граф называется ***связным***, если для любой пары вершин существует путь из одной в другую.

Подграф графа, являющийся связным, называют ***компонентами связности*** графа.

Неориентированный граф связен тогда и только тогда, когда он состоит из единственной связной компоненты.

Например, на рисунке имеются три компоненты связности:

- ✔ {1, 2, 5}, {3, 6} и {4}.



Определения

Ориентированный граф называется ***сильно связным***, если из любой его вершины достижима (по ориентированным путям) любая другая.

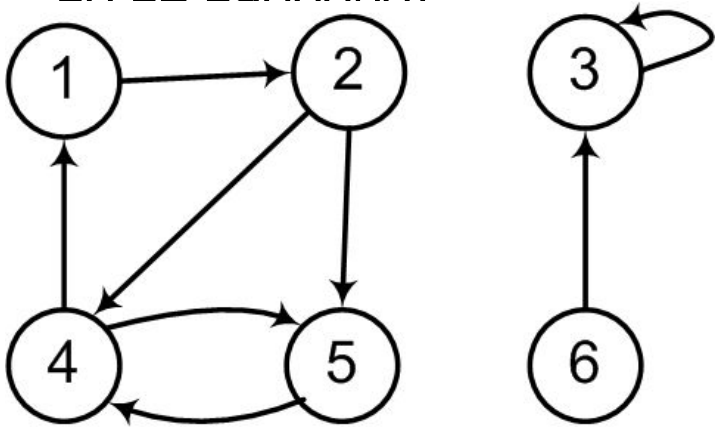
Ориентированный граф сильно связен тогда и только тогда, когда состоит из единственной сильно связной компоненты.

Граф на рисунке имеет три таких компоненты:

- ✔ {1, 2, 4, 5}, {3} и {6}.

Заметим, что вершины {3, 6} не входят в одну сильно связную компоненту,

- ✔ так как 3 достижима из 6,
- ✔ но не наоборот.



Определения

Два графа $G = (V, E)$ и $G' = (V', E')$ называются ***изоморфными***, если существует взаимно однозначное соответствие $f: V \to V'$ между множествами их вершин, при котором рёбрам одного графа соответствуют рёбра другого:

- $(u, v) \in E$ тогда и только тогда, когда $(f(u), f(v)) \in E'$.

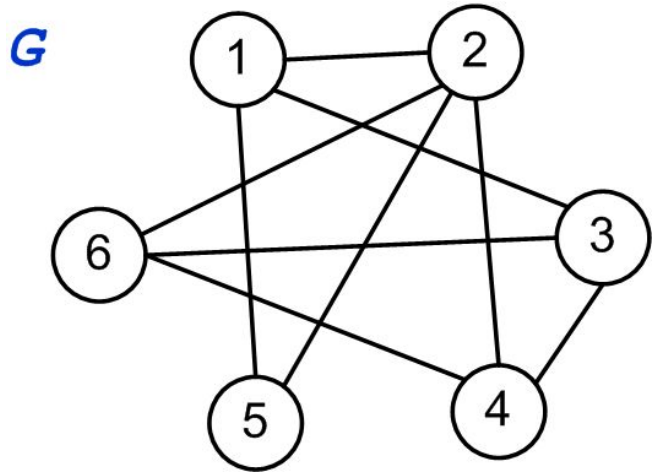
Можно сказать, что изоморфные графы − это один и тот же граф, в котором вершины названы по-разному.

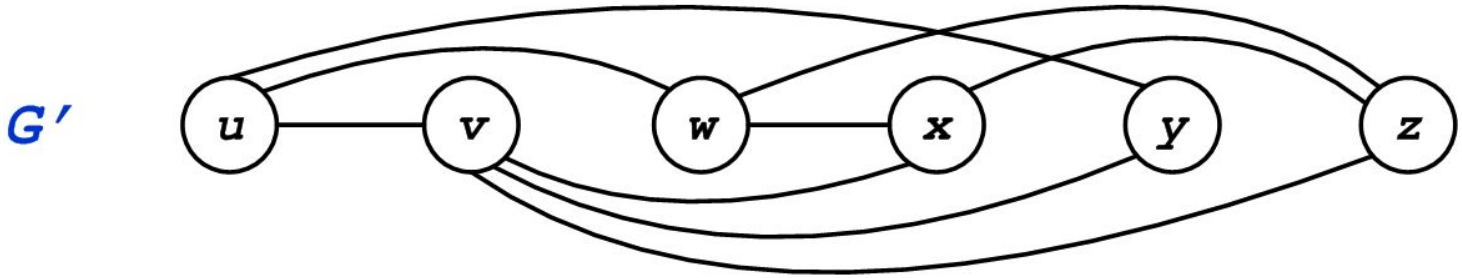
Определения

На рисунке приведен пример двух изоморфных графов *G* и *G*' с множествами вершин *V* = {1, 2, 3, 4, 5, 6} и *V* ' = {*u, v, w, x, y, z*}.

Функция $f: V \to V'$, для которой

- ✔ *f*(1) = *u*,
- ✔ *f*(2) = *v*,
- ✔ *f*(3) = *w*,
- ✔ *f*(*4*) = *x*,
- ✔ *f*(5) = *y*,
- ✔ *f*(6) = *z*.



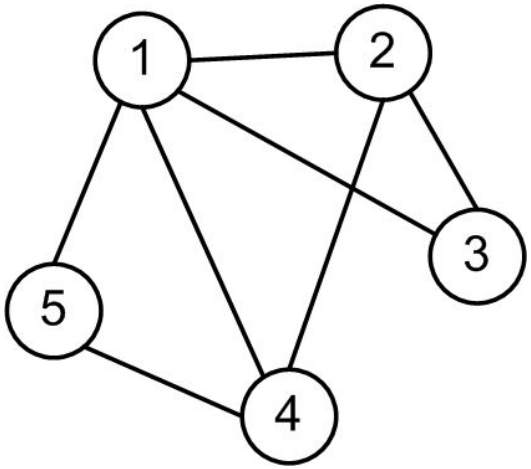


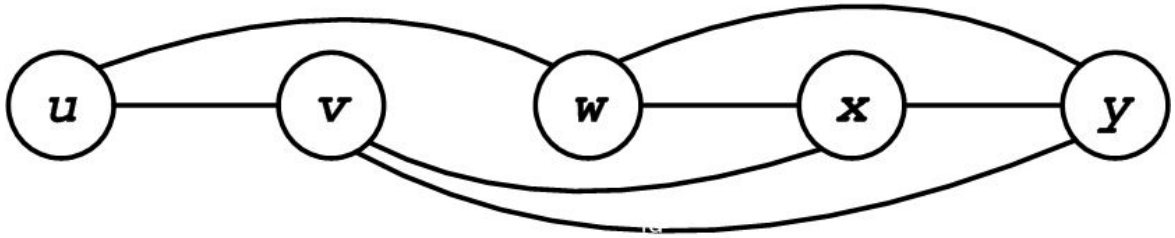
Определения

Графы на рисунке ниже не изоморфны, хотя оба имеют по 5 вершин и по 7 рёбер.

Чтобы убедиться, что они не изоморфны, достаточно отметить, что

- ✔ в верхнем графе есть вершина степени 4,
- ✔ а в нижнем − нет.





Определения

Для любого неориентированного графа *G* можно рассмотреть его *ориентированный вариант*,

- ✔ заменив каждое неориентированное ребро {*u, v*} на пару ориентированных рёбер (*u, v*) и (*v, u*), идущих в противоположных направлениях.

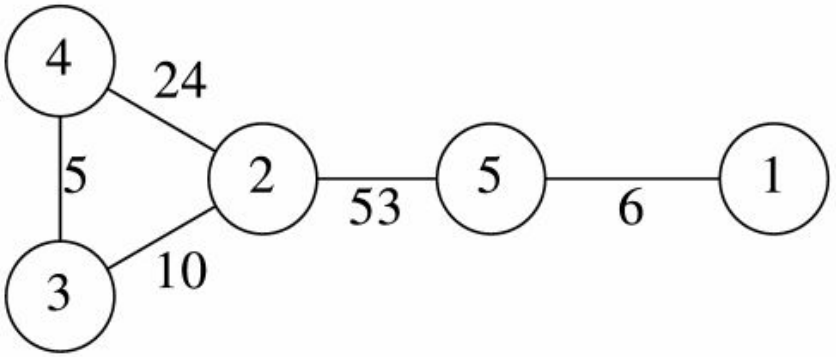
С другой стороны, для каждого ориентированного графа можно рассмотреть его *неориентированный вариант*,

- ✔ забыв про ориентацию рёбер,
- ✔ удалив петли
- ✔ и соединив рёбра (*u, v*) и (*v, u*) в одно неориентированное ребро (*u, v*).

Определения

Если задана функция $F: V \rightarrow M$, то множество M называется множеством пометок, а граф – ***помеченным***.

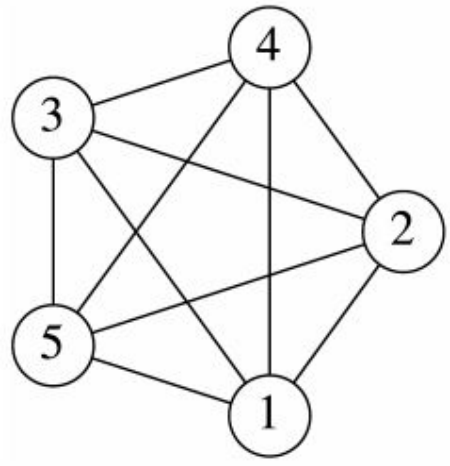
Если задана функция $F: E \rightarrow M$, т.е. ребрам графа приписаны веса, то граф называется ***взвешенным***.

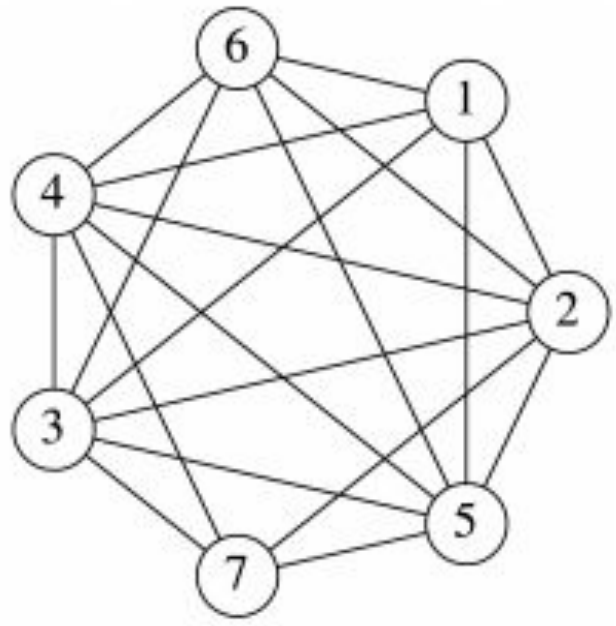


Определения

Некоторые виды графов имеют специальные названия.

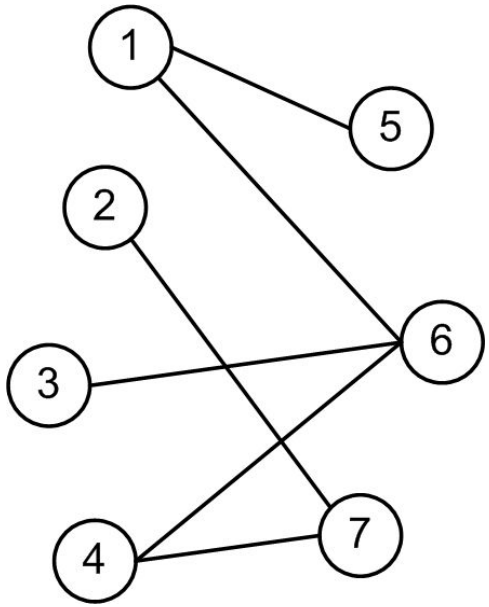
Полным графом называют неориентированный граф, содержащий все возможные рёбра для данного множества вершин (любая вершина смежна любой другой).





Определения

Неориентированный граф (V, E) называют ***двудольным***, если множество вершин V можно разбить на две части V_1 и V_2 таким образом, что концы любого ребра оказываются в разных частях.



Определения

Ациклический неориентированный граф называют ***лесом***, а связный ациклический неориентированный граф называют ***деревом без выделенного корня***.

2. Способы представления графов в программах

Представление в программе объектов математической модели – это важная составляющая программирования.

- ✔ Выбор наилучшего представления определяется требованиями конкретной задачи.

Известны различные способы представления графов в памяти компьютера. Они различаются

- ✔ объемом занимаемой памяти
- ✔ и скоростью выполнения операций над графами.

Следует заметить, что во многих задачах на графах выбор представления – решающий для эффективности алгоритмов.

Способы представления графов

Есть два стандартных способа представить граф $G = (V, E)$:

- как набор списков смежных вершин
- или как матрицу смежности.

Кроме этих методов в некоторых случаях эффективно использование

- списка ребер
- или матрицы инцидентности.

Способы представления графов

Первый обычно предпочтительнее, т.к. даёт более компактное представление для разреженных графов – тех, у которых $|E|$ много меньше $|V|^2$.

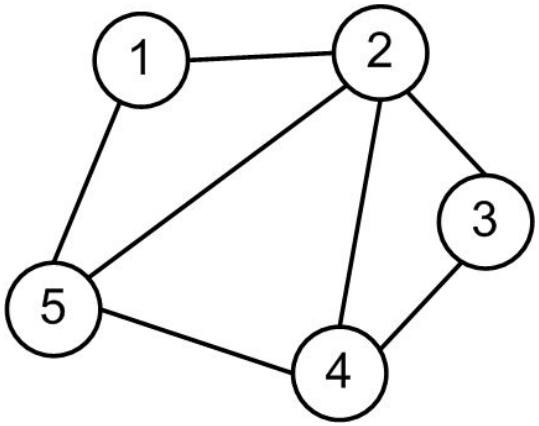
- ✔ Большинство излагаемых нами алгоритмов используют именно это представление.

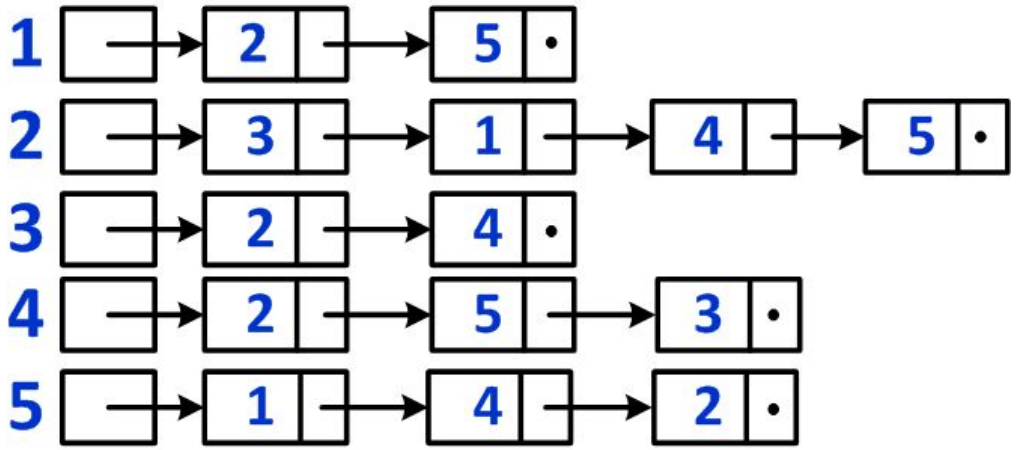
Однако в некоторых ситуациях удобнее пользоваться матрицей смежности – например, для плотных графов (близких к полному), у которых $|E|$ сравнимо с $|V|^2$.

- ✔ Матрица смежности позволяет быстро определить, соединены ли две данные вершины ребром.

Способы представления графов

Рисунок – Два представления неориентированного графа:
списки смежности, матрица смежности

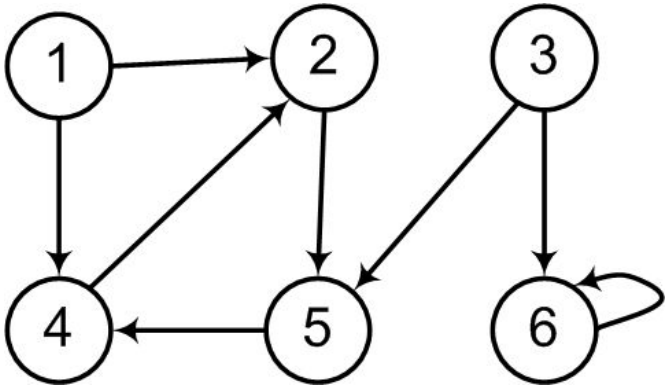


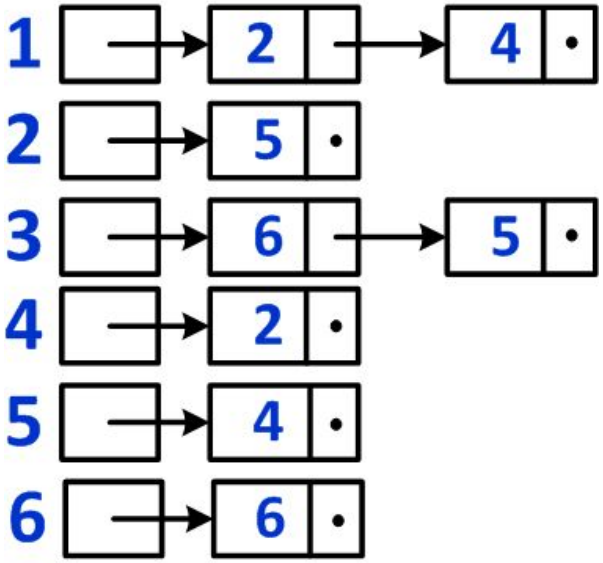


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Способы представления графов

Рисунок – Два представления ориентированного графа:
списки смежности, матрица смежности





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Список смежности графа

Представление графа $G = (V, E)$ в виде ***списков смежных вершин*** использует массив Adj из $|V|$ списков – по одному на вершину.

- Для каждой вершины $u \in V$ список смежных вершин $Adj[u]$ содержит в произвольном порядке (указатели на) все смежные с ней вершины (все вершины v, для которых $(u, v) \in E$).
- Если в основе алгоритма решения задачи лежат операции добавления и удаления вершин из списков, то хранение списков смежности удобно реализовать, используя связанное представление списков (а не в виде массивов).

Список смежности графа

Для ориентированного графа сумма длин всех списков смежных вершин равна общему числу рёбер:

- ✔ ребру (*u*, *v*) соответствует элемент *v* списка *Adj*[*u*].

Для неориентированного графа эта сумма равна удвоенному числу рёбер,

- ✔ так как ребро (*u, v*) порождает элемент в списке смежных вершин как для вершины *u,* так и для *v*.

Объем требуемой памяти составляет

- для ориентированных $n+m$
- и $n+2m$ для неориентированных графов единиц памяти,

где

- ✔ n – число вершин графа,
- ✔ а m – число ребер (дуг) графа.

Список смежности графа

Списки смежных вершин удобны для ***<u>хранения взвешенных графов</u>***, в которых каждому ребру приписан некоторый вещественный вес,

- ✔ то есть задана весовая функция $w : E \Rightarrow R$.

В этом случае удобно хранить вес *w*(*u, v*) ребра (*u, v*) ∈ *E* вместе с вершиной *v* в списке вершин, смежных с *u.*

Подобным образом можно хранить и другую информацию, связанную с графом.

Список смежности графа

Недостаток представления графа списком смежности:

- если мы хотим узнать, есть ли в графе ребро из u в v, приходится просматривать весь список $Adj[u]$ в поисках v.

Этого можно избежать, представив граф в виде матрицы смежности,

- но тогда потребуется больше памяти.

Матрица смежности графа

При использовании ***матрицы смежности*** мы

- нумеруем вершины графа (V, E) числами 1, 2, ..., $|V|$
- и рассматриваем матрицу $A = (a_{ij})$ размера $|V| \times |V|$ для которой

$$a_{ij} = \begin{cases} 1, & если \ \ (i, j) \in E \\ 0 & в \ \ противном \ \ случае. \end{cases}$$

Матрица смежности требует $\Theta(V^2)$ памяти независимо от количества рёбер в графе.

Для **неориентированного графа матрица смежности симметрична** относительно главной диагонали, поскольку (u, v) и (v, u) – это одно и то же ребро.

- Благодаря симметрии достаточно хранить только числа на главной диагонали и выше неё, тем самым мы сокращаем требуемую память почти вдвое.

Матрица смежности графа

Как и для списков смежных вершин, **хранение весов** не составляет проблемы:

- вес $w(u, v)$ ребра (u, v) можно хранить в матрице на пересечении u-й строки и v-го столбца;
- для отсутствующих рёбер можно записать специальное значение NIL (в некоторых задачах вместо этого пишут 0 или ∞).

Для **небольших графов**, когда места в памяти достаточно, матрица смежности бывает удобнее – с ней часто проще работать.

Список ребер графа

При описании графа ***списком*** его ***ребер*** каждое ребро представляется парой инцидентных ему вершин.

Это представление можно реализовать двумя массивами (или одним двумерным):

$$x = (x_0, x_1, ..., x_m) \quad \text{и} \quad y = (y_0, y_1, ..., y_m),$$

где m – количество ребер в графе.

- Каждый элемент в массиве есть метка вершины,
- а i-е ребро графа выходит из вершины x_i и входит в вершину y_i.

Объем занимаемой памяти составляет в этом случае $2m$ единиц памяти.

Неудобством является большое число шагов, необходимое для получения множества вершин, к которым ведут ребра из данной вершины.

Матрица инцидентности графа

Матрицей инцидентности называется матрица $B = [b_{ij}]$, $i = 1, 2, ..., n,\ j = 1, 2, ..., m$

где

- ✔ n – число вершин,
- ✔ а m – число ребер графа,

строки которой соответствуют вершинам, а столбцы – ребрам. Элемент матрицы b_{ij} определяется следующим образом:

- для неориентированного графа

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

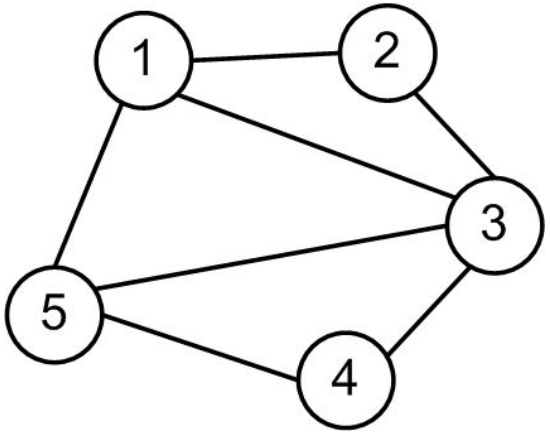
- для ориентированного графа

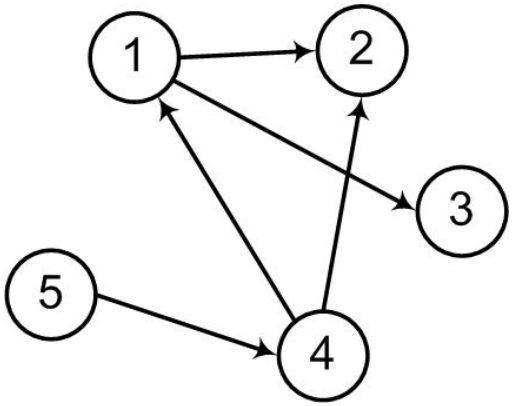
$$b_{ij} = \begin{cases} 1, & \text{если дуга выходит из вершины } i, \\ -1, & \text{если дуга входит в вершину } i, \\ 0, & \text{если вершина } i \text{ не инцидентна ребру } j. \end{cases}$$

Матрица инцидентности графа

Матрица инцидентности однозначно определяет структуру графа (рисунок).

- В каждом столбце матрицы *B* ровно две единицы.
- Равных столбцов нет.





	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

	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

Матрица инцидентности графа

Недостаток данного представления состоит в том, что

- требуется $n \times m$ единиц памяти, большинство из которых будет занято нулями;
- не всегда удобен доступ к информации.
  - ✔ Например, для ответа на вопросы
    - ✔ «есть ли в графе дуга (x, y)?»
    - ✔ или «к каким вершинам ведут ребра из вершины x?»

  может потребоваться перебор всех столбцов матрицы.

3. Обходы графа

Обходы графа

Графы могут представлять собой что угодно – карту маршрута, схему, компьютерную сеть. Но порой возникает необходимость найти нужный нам элемент в графе.

- Как же его искать?
- Какие есть способы поиска?

Обходы графа

При решении многих задач с использованием графов необходимо уметь обходить все вершины и ребра (дуги) графа.

Обойти граф – это побывать во всех вершинах точно по одному разу.

Работа всякого алгоритма обхода состоит в последовательном посещении вершин и исследовании ребер.

Какие именно действия выполняются при посещении вершины и исследовании ребра – зависит от конкретной задачи, для решения которой производится обход.

В любом случае факт посещения вершины запоминается, так что с момента посещения и до конца работы алгоритма она считается посещенной.

Обходы графа

При решении многих задач, касающихся графов, необходимы эффективные ***методы систематического обхода вершин и ребер графов***.

Обходя граф, мы двигаемся по ребрам и проходим все вершины.

- При этом можно получить много информации, которая необходима для дальнейшей обработки графа.
- Поэтому обход графа – основа многих алгоритмов исследования структуры графа.

Обходы графа

Если при посещении вершины структура графа не меняется, то наиболее полезны два основные способа обхода:

- поиск в глубину;
- поиск в ширину.

Эти методы чаще всего рассматриваются на ориентированных графах,

- ✔ но они применимы и для неориентированных, ребра которых считаются двунаправленными.

Поиск в ширину

Этот алгоритм поиска в графе также называют ***волновым алгоритмом*** из-за того, что обход графа идет по принципу распространения волны.

- Волна растекается равномерно во все стороны с одинаковой скоростью.
- На *i*-ом шаге будут помечены все вершины, достижимые за *i* ходов, если ходом считать переход из одной вершины в другую.

Поиск в ширину

Алгоритм поиск в ширину может быть описан и так.

- Пусть задан граф $G = (V, E)$ и фиксирована начальная вершина s.
- Алгоритм поиска в ширину перечисляет все достижимые из s (если идти по рёбрам) вершины в порядке возрастания расстояния от s.
  - ✔ Расстоянием считается длина (число рёбер) кратчайшего пути.
- В процессе поиска из графа выделяется часть, называемая «*деревом поиска в ширину*» с корнем s.
  - ✔ Она содержит все достижимые из s вершины (и только их).
  - ✔ Для каждой из них путь из корня в дереве поиска будет одним из кратчайших путей (из начальной вершины) в графе.

Поиск в ширину

Алгоритм применим и к ориентированным, и к неориентированным графам.

Название объясняется тем, что в процессе поиска мы идём вширь, а не вглубь:

- сначала просматриваем все соседние вершины,
- затем соседей соседей и т.д..

Поиск в ширину

Для для определения вершин, которые ранее посещались, – массив ***Visited***:

- сначала присвоить всем элементам массива ***Visited*** значение ***false***,
- затем начать поиск в ширину для каждой вершины, помеченной как ***false***.

Поиск в ширину

Алгоритм поиска в ширину формально можно записать следующим образом:

```
void WidthSearch(int v)
{
  structure Q     // Очередь
  {
    int Delayed[n];
    int rear, head;     // Хвост, голова очереди
  }
  int Cur;
  Q.rear = 0;
  Q.head = -1;
  Q.Delayed[Q.rear] = v;
  Visited[v] = true;
```

Поиск в ширину


```
  do
  {
    Q.head = Q.head +1;
    Cur = Delayed[Q.Head];
    for (каждой вершины y, смежной с Cur)
    {
      if (!Visited[y])
      {
        Q.rear = Q.rear + 1;
        Q.Delayed[Q.rear] = Graph[y];
        Visited[y] = true;
      }
    }
  } while (Q.rear != Q.head);
}
```

Поиск в ширину


```
void main()
{
  while (есть непомеченные вершины)
  {
    v = любая непомеченная вершина;
    WidthSearch(v);
  }
}
```

Поиск в ширину: программа

Функция поиска в ширину для графа с **n** вершинами, представленного матрицей смежности **A**:

```
void BFS(Matrix Graph, int n, v)
{
    // обход в ширину (V - начальная вершина)
  structure Q      //  "очередь" вершин
  {
   int el[n];
   int front, rear;
  }
  bool visited[n];  // массив посещённости вершин

  Q.front = 0;  // начало очереди
  Q.rear = 0;   // конец очереди
  Q.el[Q.rear] = v;  //  в очередь помещается исходная вершина
  visited[v] = true;  // вершина v посещена
```

Поиск в ширину: программа


```
while (Q.front <= Q.rear)      // пока очередь не пуста
{
  for (int i=0; i<n; i++)
  {
    if ((A[v][i]!=0) && (not visited[i]))
    {
     // перебираем все связные с V вершины
     Q.rear = Q.rear+1; // добавляем в очередь
        Q.el[Q.rear] = i;
        visited[i] = true; // отмечаем вершину  пройденной
    }
  }
  Q.front = Q.front+1; // переход к след. вершине в очереди
  v = Q.el[Q.front];    // и делаем её текущей
 }
}
```

Поиск в ширину: применение

- Поиск кратчайшего пути в невзвешенном графе.
- Поиск компонент связности в графе.
- Нахождения решения какой-либо задачи (игры) с наименьшим числом ходов, если каждое состояние системы можно представить вершиной графа, а переходы из одного состояния в другое – рёбрами графа.
  - ✔ Классический пример – игра, где робот двигается по полю,
  - ✔ при этом он может передвигать ящики, находящиеся на этом же поле,
  - ✔ требуется за наименьшее число ходов передвинуть ящики в требуемые позиции.
  - ✔ Решается это обходом в ширину по графу, где состоянием (вершиной) является набор координат: координаты робота, и координаты всех коробок.

Поиск в ширину: применение

- Нахождение кратчайшего цикла в ориентированном невзвешенном графе:
  - ✔ производим поиск в ширину из каждой вершины;
  - ✔ как только в процессе обхода мы пытаемся пойти из текущей вершины по какому-то ребру в уже посещённую вершину, то это означает, что мы нашли кратчайший цикл, и останавливаем обход в ширину;
  - ✔ среди всех таких найденных циклов (по одному от каждого запуска обхода) выбираем кратчайший.
- Найти все рёбра, лежащие на каком-либо кратчайшем пути между заданной парой вершин *s* и *t*.
  - ✔ Для этого надо запустить 2 поиска в ширину: из *s*, и из *t*.
  - ✔ Далее – проверить, лежит ли ребро на каком-либо кратчайшем пути.

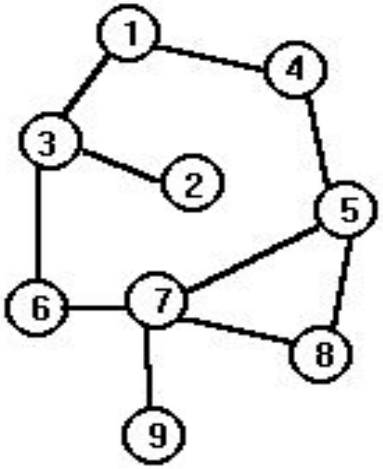
Поиск в ширину: применение

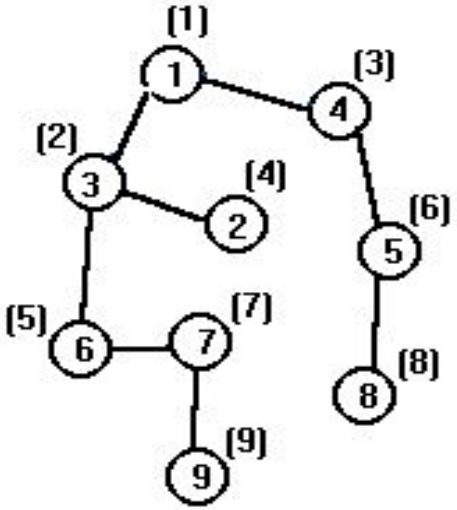
- Найти все вершины, лежащие на каком-либо кратчайшем пути между заданной парой вершин.
- Найти кратчайший чётный путь в графе (т.е. путь чётной длины).

Поиск в ширину: пример

Исходный граф на левом рисунке.

На правом рисунке рядом с вершинами в скобках указана очередность просмотра вершин графа.





Поиск в глубину

Поиск в глубину является обобщением метода обхода дерева в прямом порядке.

Предположим, что есть ориентированный граф *G*, в котором первоначально все вершины помечены как непосещенные.

Поиск в глубину

1) Поиск в глубину начинается с выбора начальной вершины *v* графа *G*, и эта вершина помечается как посещенная.
2) Затем для каждой вершины, смежной с вершиной *v* и которая не посещалась ранее, рекурсивно применяется поиск в глубину.
3) Когда все вершины, которые можно достичь из вершины *v*, будут «удостоены» посещения, поиск заканчивается.
4) Если некоторые вершины остались не посещенными, то выбирается одна из них и поиск повторяется. Этот процесс продолжается до тех пор, пока обходом не будут охвачены все вершины орграфа *G*.

Поиск в глубину

Этот метод обхода вершин орграфа называется поиском в глубину,

- ✔ т.к. поиск непосещенных вершин идет в направлении вперед (вглубь) до тех пор, пока это возможно.

Поиск в глубину

Например, пусть *x* – последняя посещенная вершина.

- Для продолжения процесса выбирается какая-либо нерассмотренная дуга $x \rightarrow y$, выходящая из вершины *x*.
- Если вершина *y* уже посещалась,
  - ✔ то ищется другая вершина, смежная с вершиной *x*.
- Если вершина *y* ранее не посещалась,
  - ✔ то она помечается как посещенная и поиск начинается заново от вершины *y*.
- Пройдя все пути, которые начинаются в вершине *y*, возвращаемся в вершину *x*,
  - ✔ т.е. в ту вершину, из которой впервые была достигнута вершина *y*.
- Затем продолжается выбор нерассмотренных дуг, исходящих из вершины *x*,
- и так до тех пор, пока не будут исчерпаны все эти дуги.

Поиск в глубину

Для определения вершин, которые ранее посещались, – массив ***Visited***:

- сначала присвоить всем элементам массива ***Visited*** значение ***false***,
- затем начать поиск в глубину для каждой вершины, помеченной как ***false***.

Поиск в глубину

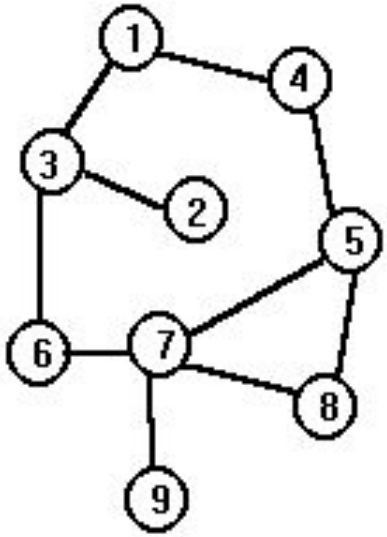
Алгоритм поиска в глубину формально можно записать следующим образом:

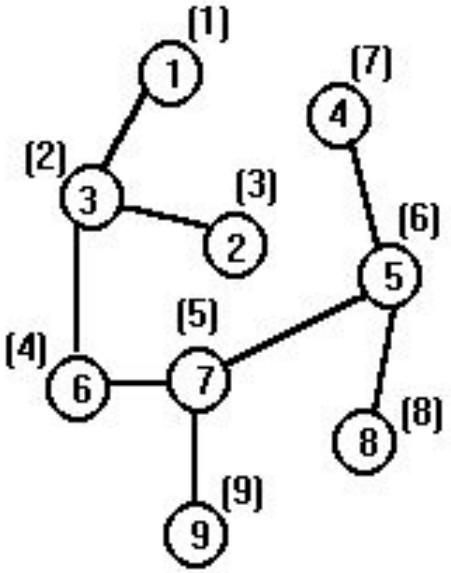
```
funcrion DepthSearch(int v)
{
  Visited[v] = true;
  for (каждой вершины y, смежной с v)
    if (!Visited[y])
      DepthSearch(y);
}

void main()
{
  while (есть непомеченные вершины)
  {
    v = любая непомеченная вершина;
    DepthSearch(v);
  }
}
```

Поиск в глубину: пример

Поиск начинается с первой вершины.

- ✔ На левом рисунке приведен исходный граф,
- ✔ а на правом рисунке у вершин в скобках указана та очередность, в которой вершины графа просматривались в процессе поиска в глубину.





Поиск в глубину: программа

Возможны две реализации алгоритма поиска в глубину:

- одна в виде рекурсивной процедуры (слайд 76),
- другая – с использованием стека (слайд 72).

Рассмотрим реализацию через стек.

Применение правила LIFO, которое характеризует работу стека, соответствует исследованию соседних коридоров в лабиринте графа:

- из всех еще не исследованных коридоров выбирается последний из тех, с которым мы столкнулись.

Словом **стратегия поиска в глубину** такова:

- идти «вглубь», пока это возможно (есть непройденные исходящие ребра),
- и возвращаться и искать другой путь, когда таких ребер нет;
- так делается, пока не обнаружены все вершины, достижимые из исходной.

Поиск в глубину: программа


```
int Graph[n][n];     // матрица смежности графа
bool Visited[n];
int stack[n]; // стек содержит номера просмотренных вершин
int top;          // вершина стека
bool flag;

void push(int v, top)
{
  top = top + 1;
  stack[top] = v;
  visited[v] = true;
}

int pop(int top)
{
  top = top - 1;
  return stack[top];
}
```

Поиск в глубину: программа


```
void DFS(Matrix Graph, int n, v)
{
  int top = 0;
  cout << "--" << v;
  push(v, top);
  visited[v] = true; // помещена в стек и посещена v
  while (top > 0)     // пока стек не пуст
  {
    int k = pop(top);
    int i = 0;  flag = false;
    do {
    if ((Graph[k][i]>0) && (!visited[i]))
      flag = true;
    else  i++;
    } while (!flag && (i<n)); // найдена новая вершина или
  // все вершины, связанные с данной вершиной, просмотрены
```

Поиск в глубину: программа


```
      if (flag)
      {   // помещена в стек и посещена новая вершина i
       push(i, top);
          visited[i] = true;
           cout << "--" << i;
      }
      else top--;     // “убираем” вершину из стека
   }
}
```

Поиск в глубину: программа

Вызов нерекурсивной процедуры:

```
for (i=0; i<n; i++)
    visited[i] = false;
for (i=0; i<n; i++)
  if  (visited[i] == false)
     DFS(i);
```

Поиск в глубину: программа

Рекурсивная процедура, реализующая алгоритм поиска в глубину:

```
void DFS(int v)
{
  // Массивы Graph и Visited те же самые описания}
  if (Visited[v] == false)
  {
    Visited[v] = true;
    cout << "--" << v;
    for (int j=0; j<n; j++)
       if (Graph[v][j]!=0)
         DFS(j);
  }
}
```

Поиск в глубину: программа

Вызов рекурсивной процедуры:

```
for (i=0; i<n; i++)
    visited[i] = false;
for (i=0; i<n; i++)
  if  (visited[i] == false)
     DFS(i);
```