

Первая работа о теории графов была опубликована в 1736 году в Санкт-Петербурге Леонардом Эйлером. Эта была знаменитая задача о кенигсбергских мостах. С тех пор графы широко исследовались как теоретически, так и практически. В данном разделе будут рассмотрены только азы теории графов.

## 1 Основные понятия теории графов

В различных источниках встречаются разные определения графов.

Формально, можно сказать, что граф — это совокупность множества  $X$ , элементы которого называются *вершинами* и множества упорядоченных пар вершин, элементы которого называются *дугами*. Граф обычно обозначается  $(X, A)$ .

Довольно часто каждому ребру графа ставят в соответствие какую-нибудь метку, например расстояние между двумя точками. Такую метку называют *весом*, а граф — *взвешенным* или *помеченным*.

В некоторых задачах имеет значение какая из вершин дуги является начальной, а какая конечной, а в некоторых нет. В первом случае граф называется *ориентированным* или *орграфом*. Во втором случае — *неориентированным*. Дуги в этом случае часто называются *ребрами*. В качестве примера ориентированного и неориентированного графов можно привести поиск правильного маршрута от точки  $A$  до точки  $B$  в городе, например, в центре Саратова с его односторонним движением на улицах. Пешеходу не важно разрешенное направление движения — маршрут будет проложен по неориентированному графу, т. е., если существует путь от точки  $A$  до точки  $B$ , то существует путь от точки  $B$  до точки  $A$ . Для водителя необходимо учитывать разрешенное направление движения и из того, что существует путь от точки  $A$  до точки  $B$ , необязательно следует, что существует путь от точки  $B$  до точки  $A$ . Графически ребра ориентированного графа представляются со стрелкой, показывающей направление от начальной точки к конечной. На рисунке 1 слева представлен неориентированный граф, справа — ориентированный.



Рис. 1: а) — неориентированный граф, б) — ориентированный граф

Изображение графа помогает понять структуру, но только для графа с малым числом вершин и ребер. Однако не всегда по чертежу можно понять, что изображен один и тот же граф. Например, на рисунке 2 представлены два графа. На первый взгляд они разные, но, если выписать все ребра для обоих графов, становится понятно, что это один и тот же граф, просто по-разному представленный. Действительно, список ребер для обоих графов:  $0 - 1$ ;  $0 - 2$ ;  $0 - 5$ ;  $0 - 6$ ;  $3 - 4$ ;  $3 - 5$ ;  $4 - 5$ ;  $4 - 6$ ;  $7 - 8$ ;  $9 - 10$ ;  $9 - 11$ ;  $9 - 12$ ;  $11 - 12$ .

Два графа называются *изоморфными*, если можно поменять метки вершин одного графа таким образом, чтобы набор ребер в итоге для обоих графов стал идентичным.

Граф называется *планарным*, если на чертеже его ребра не пересекаются.

В примерах выше и дальше, будут рассматриваться простые графы. *Простой граф* — это граф, не имеющий кратных ребер (две вершины могут быть соединены только одним ребром) и *петель* (ребро, начинающееся и заканчивающиеся в одной вершине).

Граф является *полным*, если он содержит все возможные ребра. Таких ребер для неориентированного графа может быть не более  $\frac{N(N-1)}{2}$ , где  $N$  — количество вершин графа (действительно, каждая

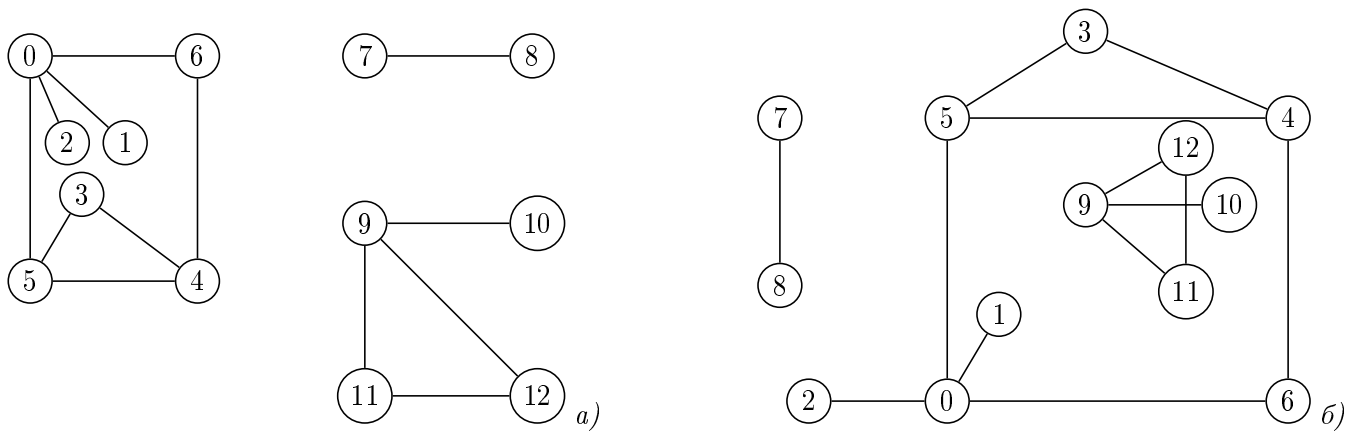
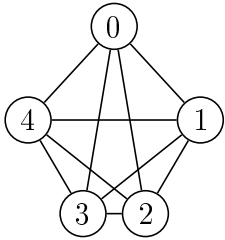


Рис. 2: Различные представления одного и того же графа

вершина может быть соединена с  $N - 1$  вершиной, на ребро должно быть одно, а оно соединяет две вершины, следовательно, делить надо пополам).

Если имеется ребро, соединяющее две вершины графа, то такие вершины называются *смежными*. А ребро — *инцидентным* этим вершинам. *Степень вершины* неориентированного графа — это количество инцидентных ей ребер. Для ориентированного графа можно говорить о *полустепени исхода* и *полустепени захода*. Это, соответственно, число ребер исходящих и заходящих в данную вершину.

Если все вершины имеют одинаковую степень, то говорят о *регулярных графах*. Например,  $K_5$  граф — полный граф с пятью вершинами, имеющий степень вершин 4.



*Подграф* — множество ребер и вершин, которые сами представляют из себя граф.

*Путь* в графе — это последовательность вершин  $a_0, a_1, a_2, \dots, a_n$ , таких что для любых  $i > 0$  вершина  $a_i$  смежна с вершиной  $a_{i-1}$ .

*Простой путь* — путь, все вершины и ребра которого различны. Если существует простой путь, начинающийся и заканчивающийся в одной вершине, такой путь называется *циклом*. Например, на рисунке 2 циклами являются:  $0 \rightarrow 6 \rightarrow 4 \rightarrow 5 \rightarrow 0$ ;  $0 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 0$ ;  $3 \rightarrow 4 \rightarrow 5 \rightarrow 3$ ;  $9 \rightarrow 11 \rightarrow 12 \rightarrow 9$ . Цикл должен содержать как минимум 3 различных вершины и три различных ребра.

*Длина пути* — количество ребер, составляющих путь (или количество вершин минус единица) для невзвешенного графа и сумма весов соответствующих ребер для взвешенного графа.

Неориентированный граф называется *связным*, если из любой вершины графа существует путь в любую другую вершину. *Несвязный граф* состоит из некоторого множества *связных компонент*, представляющих собой максимальные связные подграфы.

Например, представленный на рисунке 2 граф несвязный, содержит три связные компоненты. Первая состоит из вершин 0, 1, 2, 3, 4, 5, 6, вторая — из вершин 7, 8 и третья — из вершин 9, 10, 11, 12.

*Мост* — ребро, удаление которого увеличивает число компонент связности. Достаточно важное ребро, например, для сети дорог — наличие моста (единственная дорога) может привести к невозможности, например, добраться до какого-нибудь населенного пункта, если что-то случится с данной дорогой.

Вершина, удаление которой приводит к увеличению числа компонент связности, называется *точкой сочленения*.

В графе, представленном на рисунке 3а), ребро 2 — 3 является мостом, что видно на рисунке 3б), а вершины 2 и 3 являются точками сочленения (рисунк 3в) и г)).

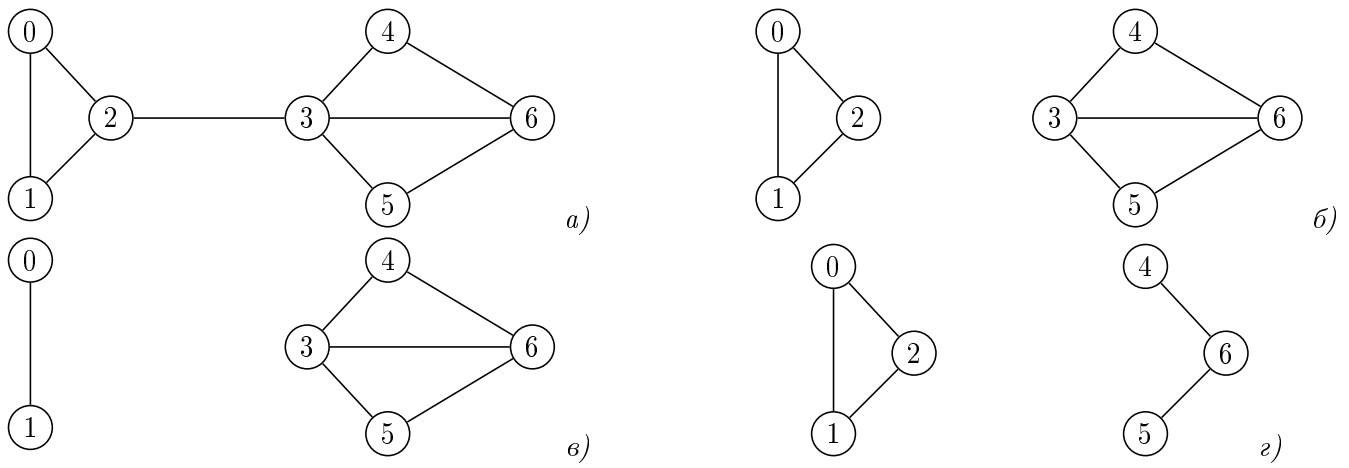


Рис. 3: а) — первоначальный граф, б) — граф с удаленным мостом; в), г) — граф с удаленными точками сочленения.

Вершина, из которой существует путь во все остальные вершины, называется *исток*ом.

Вершины, в которую существует путь из всех остальных вершин, называется *сток*ом.

В большинстве задач граф содержит малое количество из возможных ребер. Введем понятие *насыщенность* — среднее значение степени вершин,  $P = \frac{2E}{N}$ , где  $E$  — количество ребер,  $N$  — количество вершин.

Граф является *насыщенным* (*плотным*), если количество его ребер пропорционально  $N^2$  и *разреженным* в противоположном случае.

В зависимости от насыщенности определяется какой из алгоритмов необходимо использовать. Пусть есть два алгоритма, решающих одну задачу. Один имеет сложность  $O(N^2)$ , другой —  $O(E \log E)$ .

В случае плотного графа с  $N = 1000$  и  $E = 5 \times 10^5$  для первого алгоритма сложность  $O(N^2) = 10^6$ , для второго — сложность  $O(E \log E) \approx 10^7$ . Очевидно, что первый алгоритм выгоднее.

В случае разреженного графа с  $N = 10^6$  и  $E = 10^6$  для первого алгоритма сложность  $O(N^2) = 10^{12}$ , для второго — сложность  $O(E \log E) \approx 2 \times 10^7$ . Очевидно, что второй алгоритм выгоднее.

## 2 Представление графов

Графическое представление графов подходит для человека. Для программы требуется другое представление. В зависимости от того, какой граф (насыщенный или разреженный), в зависимости от задачи можно использовать различные представления графов.

### 2.1 Матрица смежности

Самый простой способ — *матрица смежности*. Матрица смежности — это матрица размера  $N \times N$ , каждый элемент которой удовлетворяет условию:

$$\begin{cases} a[i][j] = 1, & \text{если вершины } i \text{ и } j \text{ смежные,} \\ a[i][j] = 0, & \text{если вершины } i \text{ и } j \text{ несмежные.} \end{cases}$$

В случае простого неориентированного связного графа должны выполняться два условия:

1. Матрица должна быть симметричной относительно главной диагонали.
2. На диагонали матрицы должны быть нули.

Для ориентированного графа должны быть только нули на главной диагонали.

Например, для графа, изображенного на рисунке 2, матрица смежности имеет следующий вид:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Данный граф несвязный, однако можно увидеть, что для подграфов, являющихся связными компонентами  $(0 - 1 - 2 - 3 - 4 - 5 - 6)$  и  $(9 - 10 - 11 - 12)$ , подматрицы симметричны.

Недостатки матрицы смежности:

1. Занимает много лишней памяти. Для графа с  $N$  вершинами требуется  $N^2$  памяти.
2. Много лишних нулей.
3. Для проверки смежности двух вершин требуется лишняя проверка элемента матрицы (`if(a[i][j])`), что увеличивает время работы программы.

Матрица смежности может использоваться для почти полного графа, когда количество ребер почти максимально, для некоторых алгоритмов (например, алгоритм Флойда) и для проверки на смежность, вставки и удаления ребра (имеет сложность  $O(1)$ ):

1. Проверка на смежность — (`if(a[i][j]) return true;`)
2. Вставка ребра  $i - j$  — (`if(!a[i][j]) a[i][j] = 1;`)
3. Удаление ребра  $i - j$  — (`if(a[i][j]) a[i][j] = 0;`)

## 2.2 Список смежности

Намного лучше использовать вместо матрицы смежности *список смежности* — каждой вершине ставится в соответствие список смежных ей вершин. В случае, когда граф содержит  $N$  вершин и  $E$  ребер, размер списка смежности для неориентированного графа —  $O(N + 2E)$ , для ориентированного —  $O(N + E)$ .

Например, для графа, изображенного на рисунке 2 список смежности имеет вид:

0	1 → 2 → ∅
1	0 → ∅
2	0 → ∅
3	4 → 5 → ∅
4	3 → 5 → 6 → ∅
5	0 → 3 → 4 → ∅
6	0 → 4 → ∅
7	8 → ∅
8	7 → ∅
9	10 → 11 → 12 → ∅
10	9 → ∅
11	9 → 12 → ∅
12	9 → 11 → ∅

Размер каждого списка совпадает со степенью вершины. В случае списка смежности проверка на смежность не нужна, так как в списке только смежные вершины. Поиск смежной вершины, вставка и удаление ребра имеет сложность  $O(2E/N)$  (среднее значение степени вершин графа).

Создать список смежности можно разными способами, но проще всего считать ребра в виде двух вершин  $x - y$ .

**Исходные параметры:** набор ребер,  $N$  — количество вершин,  $M$  — количество ребер

**Результат:** Список смежности

**начало алгоритма**

```
Создаем вектор векторов (vector<vector<int>> Gr);
Выделяем память под  $N$  строк (Gr.resize(N));
до тех пор, пока не считали все ребра выполнять
    Считываем две вершины  $x$  и  $y$ ;
    если граф ориентированный тогда
        если  $x > N$  или  $y > N$  тогда
            пропускаем итерацию;
        иначе
            добавляем  $y$  в список вершины  $x$  (Gr[x].push_back(y));
    иначе
        если  $x > N$  или  $y > N$  тогда
            пропускаем итерацию;
        иначе
            добавляем  $y$  в список вершины  $x$  (Gr[x].push_back(y));
            добавляем  $x$  в список вершины  $y$  (Gr[y].push_back(x));

Сортируем все строки вектора;
Удаляем дубликаты (если вдруг ввели) с помощью алгоритма добавляем unique
(Gr[i].erase(unique(Gr[i].begin(), Gr[i].end()), Gr[i].end()));
```

**Алгоритм 1:** Создание списка смежности

## 2.3 Общее представление списка смежности

Основной недостаток списка смежности, представленного в виде вектора векторов — номер вершины соответствует индексу вектора, поэтому задача на удаление вершины превращается в задачу на представление указанной вершины в *изолированную*, т. е. не содержащую смежных вершин.

Более сложное представление графа в виде списка смежности с помощью `map` позволяет использовать в качестве названий вершин любые данные. Добавим и возможность вводить взвешенный граф (невзвешенный граф имеет длину ребер равную единице):

`map<int, list<pair<int, double>>>l_Adj`. Здесь ключ (первый элемент `map`) — название вершины, значение (второй элемент `map`) — список смежных вершин, каждая представляется в виде пары вершина — вес ребра.

Неудобство только в одном — необходим итератор, то есть необходимо дополнительно определять итератор, указывающий на нужную вершину (`auto it = l_Adj.find(x)`) и в дальнейшем работать именно с этим итератором, например вывод списка смежности:

```
1  for (auto it = l_Adj.begin(); it != l_Adj.end(); it++) {
2      cout << it->first << " : "; // вершина, откуда выходит ребро
3      if (isWeight) { // если взвешенный граф
4          for (auto it_1 = it->second.begin(); it_1 != it->second.end();
              it_1++) // перебираем список смежных вершин
```

```

5         cout << "{" << it_1->first << ", " << it_1->second <<
        ↪ "};";
6     }
7     else {
8         for (auto it_1 = it->second.begin(); it_1 != it->second.end();
        ↪ it_1++)
9             cout << it_1->first << "; ";
10    }
11    cout << endl;
12 }

```

Так как конструктора по умолчанию для такой сложной конструкции не существует, то в случае ориентированного графа, необходимо добавлять искусственное ребро, чтобы добавить «заходящую» вершину, если ее еще нет:

1. Проверяем, что вершины  $y$  нет.
2. Создаем «мнимое» ребро  $y \rightarrow 0$
3. После этого удаляем это «мнимое» ребро.

Создание списка смежности приведено ниже:

```

1  while (in.peek() != EOF) {
2      in >> x >> y >> w; // вводим ребро в виде x -> y -> вес
3      if (!isWeigth) //невзвешенный
4          w = 1.0;
5      auto it_x = l_Adj.find(x); //ищем вершину x
6      bool fl = true;
7      if (it_x != l_Adj.end()) { //если вершина x уже существует
8          for (auto it_y = it_x->second.begin(); it_y != it_x->second.end(); it_y++)
9              if (it_y->first == y) { //проверяем существует ли ребро x->y
10                 fl = false; //если уже существует
11                 break;
12             }
13    }
14    if (orient) { //ориентированный граф
15        if (fl) { //добавляем ребро x->y
16            l_Adj[x].push_back(make_pair(y, w)); //добавляем x -> y (y в список
            ↪ смежности для x)
17            if (l_Adj.find(y) == l_Adj.end()) { //если вершины y нет, добавляем
            ↪ "мнимое" ребро y->0,
18                l_Adj[y].push_back(make_pair(0, 0)); //добавляем 0 в список
            ↪ смежности y
19            auto it = l_Adj.find(y); //находим y
20            it->second.erase(it->second.begin()); //удаляем "мнимое" ребро
            ↪ (оно единственное)
21        }
22    }
23    }
24    else if (fl) { //неориентированный
25        l_Adj[x].push_back(make_pair(y, w)); //добавляем x->y
26        l_Adj[y].push_back(make_pair(x, w)); //добавляем y->x
27    }
28 }

```

## 3 Обходы

Как и в случае деревьев, для работы с графами необходим алгоритм для работы с вершинами. Обход — посещение каждой вершины графа по определенному алгоритму. Существует два обхода — в глубину (обходим смежные вершины до тех пор, пока это возможно) и в ширину (записываем в очередь все смежные вершины).

В идеале работа с графами описывается внутри класса. В нашем случае все переменные, необходимые для работы с графами, делаем глобальными (список смежности (**Gr**), список посещенных вершин (**used**), количество вершин (**N**), список вершин, составляющих путь (**path**) и т.д.).

Все алгоритмы ниже описаны с использованием списка смежности в виде вектора векторов.

### 3.1 Обход в глубину

Общий смысл обхода: выбираем начальную вершину, помечаем ее как посещенную, ищем смежную не посещенную вершину и вызываем обход относительно ее.

**Исходные параметры:** **x** — текущая вершина

**Результат:** тип функции — **void**

**начало алгоритма**

Помечаем вершину **x** как посещенную (**used[x] = 1**);

Записываем ее в вектор **path**;

**цикл**  $i = 0$  **до**  $i < Gr[x].size()$  **выполнять**

**если** (существует не посещенная вершина ( $Gr[x][i]$ )) **тогда**

        вызываем рекурсивно эту функцию относительно вершины  $Gr[x][i]$ ;

**Алгоритм 2:** Обход в глубину

Рассмотрим пример для графа, изображенного на рисунке 1. Зеленым цветом будет показана текущая вершина, синим — посещенные.

Список смежности:

0 | 1 → 2 → 4 → 5 → ∅

1 | 0 → ∅

2 | 0 → 5 → ∅

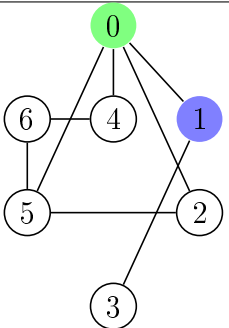
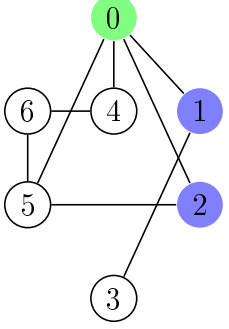
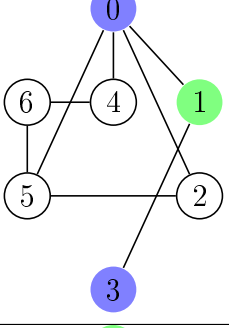
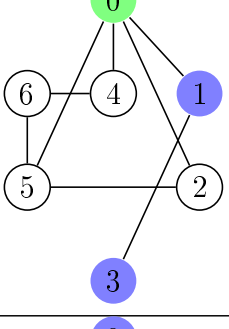
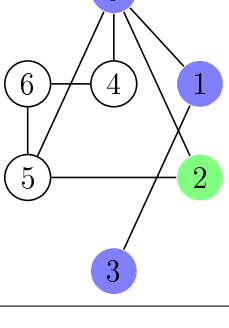
3 | 1 → ∅

4 | 0 → 6 → ∅

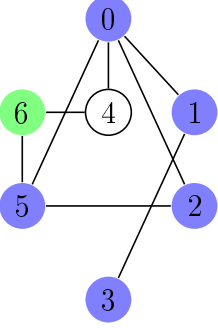
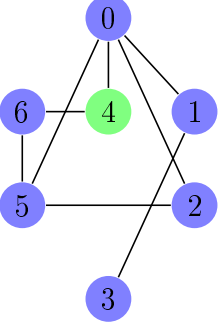
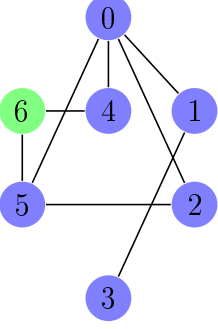
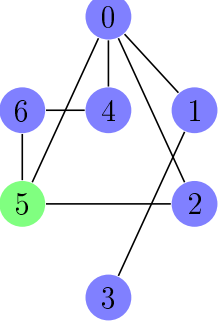
5 | 0 → 2 → 6 → ∅

6 | 4 → 5 → ∅

Граф	Текущая вершина	Смежная вершина	path	used	Стек
	0	1	[0]	[1, 0, 0, 0, 0, 0, 0]	0

	0	2	$[0, 1]$	$[1, 1, 0, 0, 0, 0, 0]$	0, 1
	0	нет	$[0, 1, 3]$	$[1, 1, 0, 1, 0, 0, 0]$	3, 1, 0
	1	нет	$[0, 1, 3]$	$[1, 1, 0, 1, 0, 0, 0]$	1, 0
	0	2	$[0, 1, 3]$	$[1, 1, 0, 1, 0, 0, 0]$	0
	2	5	$[0, 1, 3, 2]$	$[1, 1, 1, 1, 0, 0, 0]$	2, 0



	5	6	[0, 1, 3, 2]	[1, 1, 1, 1, 0, 1, 0]	5, 2, 0
	6	4	[0, 1, 3, 2, 5, 6]	[1, 1, 1, 1, 0, 1, 1]	6, 5, 2, 0
	4	нет	[0, 1, 3, 2, 5, 6, 4]	[1, 1, 1, 1, 1, 1, 1]	4, 6, 5, 2, 0
	6	нет	[0, 1, 3, 2, 5, 6, 4]	[1, 1, 1, 1, 1, 1, 1]	6, 5, 2, 0
	5	нет	[0, 1, 3, 2, 5, 6, 4]	[1, 1, 1, 1, 1, 1, 1]	5, 2, 0

	2	нет	[0, 1, 3, 2, 5, 6, 4]	[1, 1, 1, 1, 1, 1, 1]	2, 0
	0	нет	[0, 1, 3, 2, 5, 6, 4]	[1, 1, 1, 1, 1, 1, 1]	0

В данном случае под стеком понимается стек рекурсивных вызовов функции.

Таким образом, результат обхода в глубину — 0 — > 1 — > 3 — > 2 — > 5 — > 6 — > 4. Обход в глубину используется во многих алгоритмах, в частности, в случае невзвешенного графа кратчайший путь от вершины A до вершины B определяется с помощью обхода в глубину.

### 3.2 Обход в ширину

Алгоритм прост: создаем очередь, записываем в нее начальную вершину, помечая ее как посещенную. Извлекаем голову очереди и записываем в очередь все смежные не посещенные вершины головы. Повторяем процесс до тех пор, пока очередь не пуста.

**Исходные параметры:** x — текущая вершина

**Результат:** тип функции — void

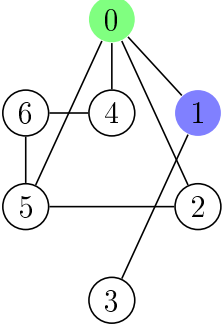
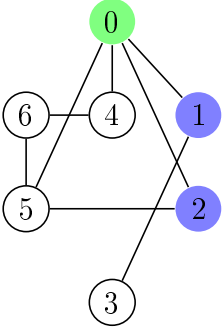
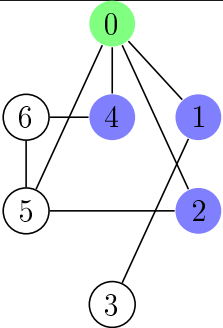
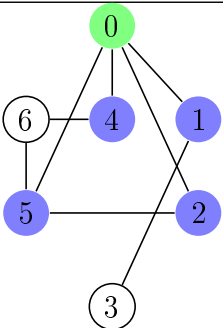
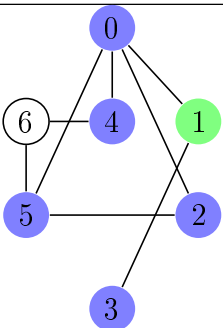
**начало алгоритма**

```
Помечаем вершину x как посещенную (used[x] = 1);
Записываем ее в вектор path;
Помещаем ее в очередь;
до тех пор, пока очередь не пуста выполнять
    Извлекаем из очереди голову (y);
    цикл (i = 0 до i < Gr[y].size()) выполнять
        если (существует не посещенная вершина (Gr[y][i])) тогда
            Помечаем вершину Gr[y][i] как посещенную;
            Записываем ее в вектор path;
            Помещаем ее в очередь;
```

Алгоритм 3: Обход в ширину

Рассмотрим в качестве примера тот же граф.

Граф	Текущая вершина	Смежная вершина	path	used	очередь
------	-----------------	-----------------	------	------	---------

	0	1	[0]	[1, 1, 0, 0, 0, 0, 0]	1
	0	2	[0, 1, 2]	[1, 1, 1, 0, 0, 0, 0]	1, 2
	0	4	[0, 1, 2, 4]	[1, 1, 1, 0, 1, 0, 0]	1, 2, 4
	0	5	[0, 1, 2, 4, 5]	[1, 1, 1, 0, 1, 1, 0]	1, 2, 4, 5
	1	3	[0, 1, 2, 4, 5, 3]	[1, 1, 1, 1, 1, 1, 0]	2, 4, 5, 3

	2	нет	[0, 1, 2, 4, 5, 3]	[1, 1, 1, 1, 1, 1, 0]	4, 5, 3
	4	6	[0, 1, 2, 4, 5, 3, 6]	[1, 1, 1, 1, 1, 1, 1]	5, 3, 6
	5	нет	[0, 1, 2, 4, 5, 3, 6]	[1, 1, 1, 1, 1, 1, 1]	3, 6
	3	нет	[0, 1, 2, 4, 5, 3, 6]	[1, 1, 1, 1, 1, 1, 1]	6
	6	нет	[0, 1, 2, 4, 5, 3, 6]	[1, 1, 1, 1, 1, 1, 1]	∅

Таким образом результат обхода в ширину — 0 — > 1 — > 2 — > 4 — > 5 — > 3 — > 6.

## 4 Простейшие алгоритмы теории графов

### 4.1 Компоненты связности

Поиск компонент связности для неориентированного графа очень прост:

1. Берем первую вершину графа и вызываем относительно ее обход в глубину.
2. Полученный вектор `path` является компонентой связности.
3. Ищем следующую непосещенную вершину и вызываем обход в глубину относительно ее.

**Исходные параметры:** нет параметров

**Результат:** тип функции — `void`

**начало алгоритма**

```
    Обнуляем вектор (used);  
    до тех пор, пока остались непосещенные вершины выполнять  
        Очищаем вектор path;  
        Вызываем обход в глубину;  
        Выводим список посещенных вершин;
```

**Алгоритм 4:** Поиск компонент связности

Например, для графа, изображенного на рисунке 2:

1. Вызываем обход в глубину относительно вершины 0. Результат —  $0 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 6$ .
2. Следующая непосещенная вершина — 7. Результат —  $7 \rightarrow 8$ .
3. Следующая непосещенная вершина — 9. Результат —  $9 \rightarrow 10 \rightarrow 11 \rightarrow 12$ .

В этом графе три компоненты связности.

## 4.2 Компоненты сильной связности

В случае ориентированного графа можно говорить о наличии сильно связных компонент.

Для работы создаем не только необходимый граф ( $Gr$ ), но и *транспонированный граф* ( $GrT$ ), т. е. ориентированный граф, все направления которого изменены на противоположные.

Очевидно, что сильно связные компоненты для транспонированного графа будут такими же, что и для обычного.

Создаем два вектора: **order**, в который будет записывать вершины в порядке выхода из обхода обычного графа (так как вызов рекурсивный, то самая первая вершина будет записана самой последней).

Потом будем вызывать для транспонированного графа обход в глубину в порядке, записанном в **order**, начиная с конца и записывать результат в **path**.

Для транспонированного графа будет простой обход в глубину. Для обычного — в **order** будет записывать уже после выхода из цикла.

**Исходные параметры:**  $x$  — текущая вершина

**Результат:** тип функции — **void**

**начало алгоритма**

```
Помечаем вершину  $x$  как посещенную;  
цикл  $i = 0$  до  $i < Gr[x].size()$  выполнять  
    если (существует не посещенная вершина ( $Gr[x][i]$ )) тогда  
        вызываем рекурсивно эту функцию относительно вершины  $Gr[x][i]$ ;  
Записываем вершину  $x$  в order;
```

**Алгоритм 5:** Обход в глубину для обычного графа (dfs1)

**Исходные параметры:**  $x$  — текущая вершина

**Результат:** тип функции — **void**

**начало алгоритма**

```
Помечаем вершину  $x$  как посещенную;  
Записываем вершину  $x$  в path;  
цикл  $i = 0$  до  $i < GrT[x].size()$  выполнять  
    если (существует не посещенная вершина ( $GrT[x][i]$ )) тогда  
        вызываем рекурсивно эту функцию относительно вершины  $GrT[x][i]$ ;
```

**Алгоритм 6:** Обход в глубину для транспонированного графа (dfs2)

Рассмотрим пример.

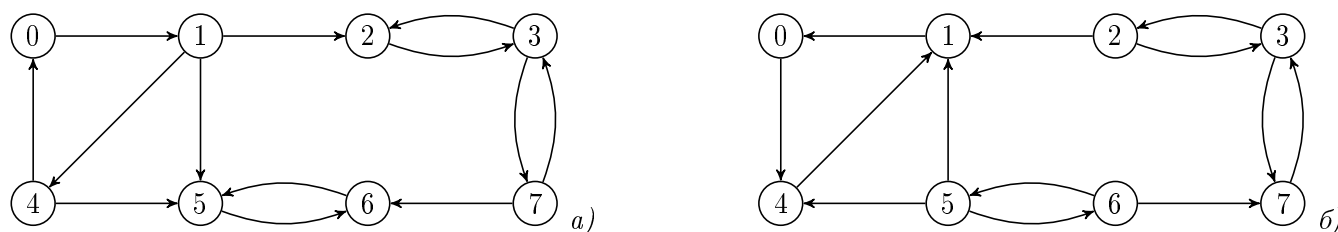


Рис. 4: а) — ориентированный граф, б) — транспонированный граф

# начало алгоритма

до тех пор, пока пока не закончились ребра выполнять

Считываем ребро  $x - y$ ;

Добавляем вершину  $y$  в  $\text{Gr}[x]$ ;

Добавляем вершину  $x$  в  $\text{GrT}[y]$ ;

Обнуляем вектор **used**;

Для всех непосещенных вершин графа вызываем  $\text{dfs1}(i)$ ;

Обнуляем вектор **used**;

цикл ( $i = 0$  до  $i < N$ ) выполнять

$v = \text{order}[n - 1 - i]$ ;

если вершина  $v$  непосещенная тогда

Вызываем  $\text{dfs2}(v)$ ;

Выводим сильно связную компоненту;

Очищаем **path**

## Алгоритм 7: Функция $\text{main}()$

Списки смежности для обоих графов и трассировки вызовов обходов в глубину приведены в таблицах.

Граф		Граф	
0	$1 \rightarrow \emptyset$	0	$4 \rightarrow \emptyset$
1	$2 \rightarrow 4 \rightarrow 5 \rightarrow \emptyset$	1	$1 \rightarrow \emptyset$
2	$3 \rightarrow \emptyset$	2	$1 \rightarrow 3 \rightarrow \emptyset$
3	$2 \rightarrow 7 \rightarrow \emptyset$	3	$2 \rightarrow 7 \rightarrow \emptyset$
4	$0 \rightarrow 5 \rightarrow \emptyset$	4	$1 \rightarrow \emptyset$
5	$6 \rightarrow \emptyset$	5	$1 \rightarrow 4 \rightarrow 6 \rightarrow \emptyset$
6	$5 \rightarrow \emptyset$	6	$5 \rightarrow 7 \rightarrow \emptyset$
7	$3 \rightarrow 6 \rightarrow \emptyset$	7	$3 \rightarrow \emptyset$

Текущая вершина	used	Смежная вершина	Стек	order
0	[1, 0, 0, 0, 0, 0, 0, 0]	1	0	$\emptyset$
1	[1, 1, 0, 0, 0, 0, 0, 0]	2	1, 0	$\emptyset$
2	[1, 1, 1, 0, 0, 0, 0, 0]	3	2, 1, 0	$\emptyset$
3	[1, 1, 1, 1, 0, 0, 0, 0]	7	3, 2, 1, 0	$\emptyset$
7	[1, 1, 1, 1, 0, 0, 0, 1]	6	7, 3, 2, 1, 0	$\emptyset$
6	[1, 1, 1, 1, 0, 0, 1, 1]	5	6, 7, 3, 2, 1, 0	$\emptyset$
5	[1, 1, 1, 1, 0, 1, 1, 1]	нет	6, 7, 3, 2, 1, 0	5
6	[1, 1, 1, 1, 0, 1, 1, 1]	нет	7, 3, 2, 1, 0	5, 6
7	[1, 1, 1, 1, 0, 1, 1, 1]	нет	3, 2, 1, 0	5, 6, 7
3	[1, 1, 1, 1, 0, 1, 1, 1]	нет	2, 1, 0	5, 6, 7, 3
2	[1, 1, 1, 1, 0, 1, 1, 1]	нет	1, 0	5, 6, 7, 3, 2
1	[1, 1, 1, 1, 1, 1, 1, 1]	4	4, 1, 0	5, 6, 7, 3, 2
4	[1, 1, 1, 1, 1, 1, 1, 1]	нет	1, 0	5, 6, 7, 3, 2, 4
1	[1, 1, 1, 1, 1, 1, 1, 1]	нет	0	5, 6, 7, 3, 2, 4, 1
0	[1, 1, 1, 1, 1, 1, 1, 1]	нет	$\emptyset$	5, 6, 7, 3, 2, 4, 1, 0

Следовательно, для транспонированного графа обход в глубину вызывается для непосещенных вершин в следующем порядке  $0 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 6 \rightarrow 5$ .

Текущая вершина	used	Смежная вершина	Стек	path
0	[1, 0, 0, 0, 0, 0, 0, 0]	4	0	0
4	[1, 0, 0, 0, 1, 0, 0, 0]	1	4, 0	0, 4
1	[1, 1, 0, 0, 1, 0, 0, 0]	нет	4, 0	0, 4, 1
4	[1, 1, 0, 0, 1, 0, 0, 0]	нет	0	0, 4, 1
0	[1, 1, 0, 0, 1, 0, 0, 0]	нет	∅	0, 4, 1

Сильно связная компонента:  $0 \longrightarrow 1 \longrightarrow 4$

2	[1, 1, 1, 0, 1, 0, 0, 0]	3	2	2
3	[1, 1, 1, 1, 1, 0, 0, 0]	7	3, 2	2, 3
7	[1, 1, 1, 1, 1, 0, 0, 1]	нет	3, 20	2, 3, 7
3	[1, 1, 1, 1, 1, 0, 0, 1]	нет	2	2, 3, 7
2	[1, 1, 1, 1, 1, 0, 0, 1]	нет	∅	2, 3, 7

Сильно связная компонента:  $2 \longrightarrow 3 \longrightarrow 7$

6	[1, 1, 1, 1, 1, 0, 1, 1]	5	6	6
5	[1, 1, 1, 1, 1, 1, 1, 1]	нет	6	6, 5
6	[1, 1, 1, 1, 1, 1, 1, 1]	нет	∅	5, 6

Сильно связная компонента:  $5 \longrightarrow 6$



## 4.3 Поиск циклов в графе

Часть алгоритмов разработана для ациклических графов, часть — для графов, имеющих циклы. Поэтому задача нахождения циклов является достаточно важной.

Поиск циклов использует обход в глубину с дополнительным условием. Будем отмечать посещенную вершину не «черным», а «серым» цветом. Если при обходе встретили вершину, обозначенную серым цветом (и это не предыдущая вершина в случае неориентированного графа), значит встретили цикл.

В данном случае в качестве глобальных переменных будем использовать **pr** — массив предков, и переменные **cycle\_st** и **cycle\_end** — для определения начальной и предпоследней вершины цикла.

Если необходимо вывести все циклы графа, но после нахождения цикла надо просто удалить последнее ребро, и начать искать цикл заново.

**Исходные параметры:**  $x$  — текущая вершина

**Результат:** нашли цикл или нет

**начало алгоритма**

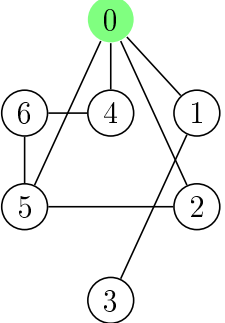
```

Помечаем вершину  $x$  как посещенную впервые ( $used[x] = 1$ );
цикл  $i = 0$  до  $i < Gr[x].size()$  выполнять
    если (существует не посещенная вершина ( $Gr[x][i]$ )) тогда
        предком для  $Gr[x][i]$  является  $x$  ( $pr[Gr[x][i]] = x$ );
        если эта функция относительно вершины  $Gr[x][i]$  истинна тогда
            возвратить true
    иначе если  $Gr[x][i]$  уже посещали тогда
        если вершина  $Gr[x][i]$  является предком  $x$  ( $pr[Gr[x][i]] == x$ ) тогда
            пропускаем итерацию;
            (только для неориентированного графа)
         $cycle\_end = x$ ;
         $cycle\_st = Gr[x][i]$ ;
        Удаляем ребро  $x - Gr[x][i]$ ;
        возвратить true
 $used[x] = 2$  (помечаем вершину  $x$  в черный цвет);
возвратить false;

```

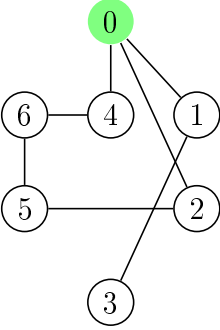
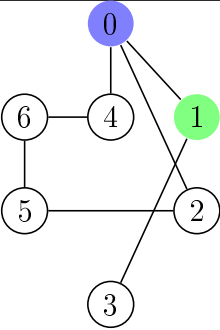
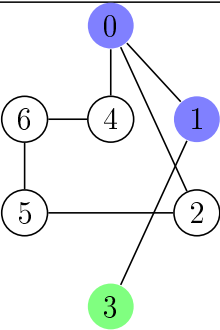
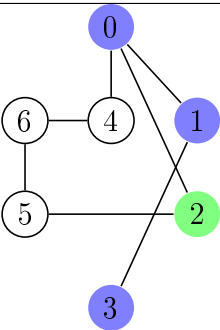
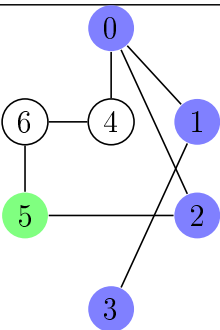
**Алгоритм 8:** Поиск цикла

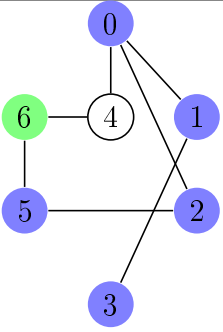
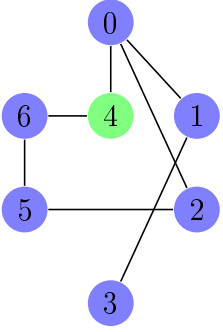
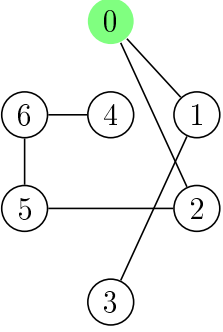
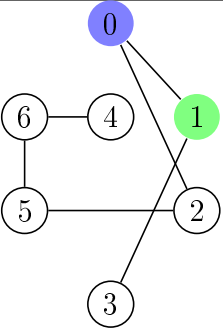
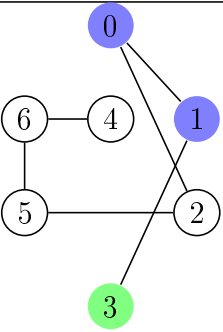
Рассмотрим в качестве примера граф, изображенный на рисунке 1.

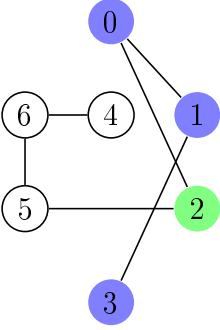
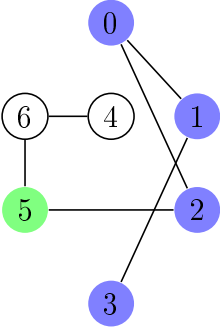
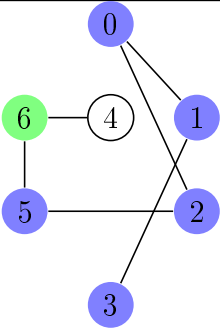
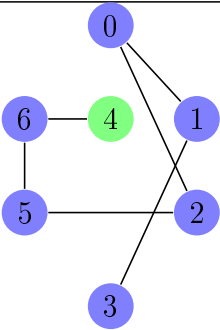
Граф	Текущая	Смежная	pr	used	Результат
	0	1	$[-1, -1, -1, -1, -1, -1, -1]$	$[1, 0, 0, 0, 0, 0, 0]$	

	1	3	$[-1, 0, -1, -1, -1, -1]$	$[1, 1, 0, 0, 0, 0, 0]$	
	3	нет	$[-1, 0, -1, 1, -1, -1, -1]$	$[1, 1, 0, 2, 0, 0, 0]$	false
	2	5	$[-1, 0, -1, 1, -1, -1, -1]$	$[1, 1, 1, 2, 0, 0, 0]$	
	5	0	$[-1, 0, -1, 1, -1, 2, -1]$	$[1, 1, 1, 2, 0, 1, 0]$	
	0	цикл	$[-1, 0, -1, 1, -1, 2, -1]$	$[1, 1, 1, 2, 0, 1, 0]$	true

цикл  $0 - 2 - 5 - 0$ . Удаляем ребро  $0 - 5$

	0	1	$[-1, -1, -1, -1, -1, -1, -1]$	$[1, 0, 0, 0, 0, 0, 0]$	
	1	3	$[-1, 0, -1, -1, -1, -1, -1]$	$[1, 1, 0, 0, 0, 0, 0]$	
	3	het	$[-1, 0, -1, 1, -1, -1, -1]$	$[1, 1, 0, 2, 0, 0, 0]$	false
	2	5	$[-1, 0, -1, 1, -1, -1, -1]$	$[1, 1, 1, 2, 0, 0, 0]$	
	5	6	$[-1, 0, -1, 1, -1, 2, -1]$	$[1, 1, 1, 2, 0, 1, 0]$	

	6	4	$[-1, 0, -1, 1, -1, 2, 5]$	$[1, 1, 1, 2, 0, 1, 1]$	
	4	0	$[-1, 0, -1, 1, 6, 2, 5]$	$[1, 1, 1, 2, 1, 1, 1]$	true
Цикл $0 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 4 \rightarrow 0$ . Удаляем ребро $0 - 4$					
	0	1	$[-1, -1, -1, -1, -1, -1, -1]$	$[1, 0, 0, 0, 0, 0, 0]$	
	1	3	$[-1, 0, -1, -1, -1, -1, -1]$	$[1, 1, 0, 0, 0, 0, 0]$	
	3	нет	$[-1, 0, -1, 1, -1, -1, -1]$	$[1, 1, 0, 2, 0, 0, 0]$	false

	2	5	$[-1, 0, -1, 1, -1, -1]$	$[1, 1, 1, 2, 0, 0, 0]$	
	5	6	$[-1, 0, -1, 1, -1, 2, -1]$	$[1, 1, 1, 2, 0, 1, 0]$	
	6	4	$[-1, 0, -1, 1, -1, 2, 5]$	$[1, 1, 1, 2, 0, 1, 1]$	
	4	het	$[-1, 0, -1, 1, 6, 2, 5]$	$[1, 1, 1, 2, 2, 1, 1]$	false

**начало алгоритма**

```
bool fl = true;
до тех пор, пока fl истина выполнять
    fl = false;
    массив предков заполняем -1;
    массив посещенных вершин обнуляем;
    cycle_st = -1;
    цикл i = 0 до i < N выполнять
        если обход в глубину от вершины i истина тогда
            fl = true;
            прерываем цикл;
        если cycle_st остался -1 тогда
            Цикла нет;
        иначе
            cycle_st записываем в вектор результата;
            цикл v = cycle_end до v != cycle_st с шагом v = pr[v] выполнять
                Записываем v в вектор результата;
            cycle_st записываем в вектор результата;
            Выводим результат ;
            Очищаем вектор результата ;
```

**Алгоритм 9:** Функция `main()`

## 4.4 Топологическая сортировка

Дан ориентированный граф с  $N$  вершинами и  $M$  ребрами. Требуется перенумеровать его вершины таким образом, чтобы каждое ребро вело из вершины с меньшим номером в вершину с большим.

Иными словами, требуется найти перестановку вершин (топологический порядок), соответствующую порядку, задаваемому всеми ребрами графа.

Топологической сортировки может не быть, если граф содержит циклы.

Задача топологической сортировки встречается очень часто. Например, при разработке учебных планов необходимо правильно расставить курсы: компьютерную графику читать до изучения алгебры и геометрии нельзя, а математический анализ и английский язык — можно.

Алгоритм очень простой — обычный обход в глубину, но результат записывается после выхода из рекурсии (аналогично алгоритму `dfs1` при поиске сильно связанных компонент).

После того, как будут пройдены все вершины, необходимо перезаписать все в обратном порядке. Это и будут вершины, размещенные в порядке топологической сортировки.

**Исходные параметры:**  $x$  — текущая вершина

**Результат:** тип функции — `void`

**начало алгоритма**

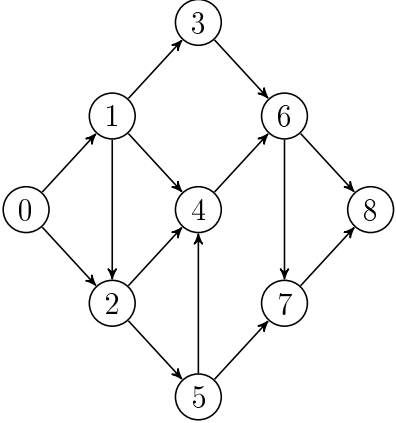
```
Помечаем вершину x как посещенную;
цикл i = 0 до i < Gr[x].size() выполнять
    если (существует не посещенная вершина (Gr[x][i])) тогда
        вызываем рекурсивно эту функцию относительно вершины Gr[x][i];
    Записываем вершину x в order;
```

**Алгоритм 10:** Обход в глубину

Вызов обхода должен быть для всех непосещенных вершин:

```
1   for(int i = 0; i < N; i++0)
2       if(!used[i])
3           dfs(i);
```

Рассмотрим пример.

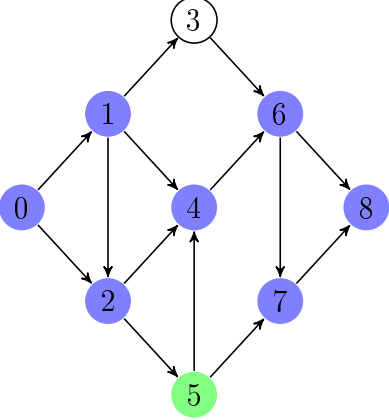
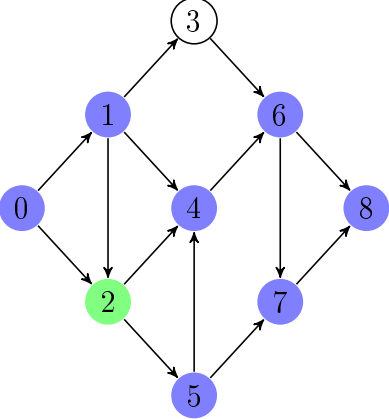
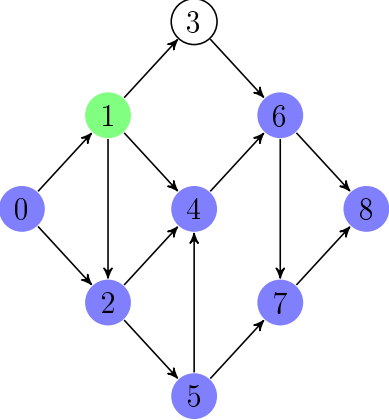
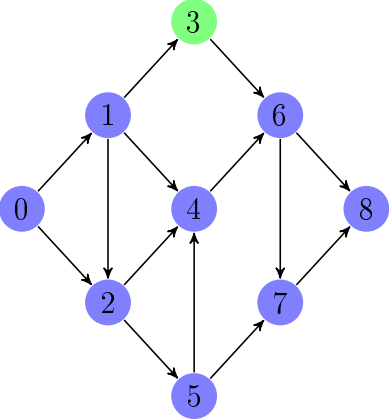


Граф	Текущая	Смежная	used	стек	order
	0	1	[1, 0, 0, 0, 0, 0, 0, 0, 0]	0	∅
	1	2	[1, 1, 0, 0, 0, 0, 0, 0, 0]	1, 0	∅
	2	4	[1, 1, 1, 0, 0, 0, 0, 0, 0]	2, 1, 0	∅

	4	6	$[1, 1, 1, 0, 1, 0, 0, 0, 0]$	4, 2, 1, 0	$\emptyset$
	6	7	$[1, 1, 1, 0, 1, 0, 1, 0, 0]$	6, 4, 2, 1, 0	$\emptyset$
	7	8	$[1, 1, 1, 0, 1, 0, 1, 1, 0]$	7, 6, 4, 2, 1, 0	$\emptyset$
	8	нет	$[1, 1, 1, 0, 1, 0, 1, 1, 1]$	8, 7, 6, 4, 2, 1, 0	8



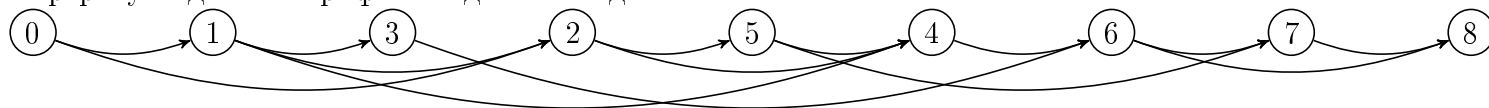
	7	нет	$[1, 1, 1, 0, 1, 0, 1, 1, 1]$	$7, 6, 4, 2, 1, 0g$	8, 7
	6	нет	$[1, 1, 1, 0, 1, 0, 1, 1, 1]$	$6, 4, 2, 1, 0$	8, 7, 6
	4	нет	$[1, 1, 1, 0, 1, 0, 1, 1, 1]$	$4, 2, 1, 0$	8, 7, 6, 4
	2	5	$[1, 1, 1, 0, 1, 0, 1, 1, 1]$	$2, 1, 0$	8, 7, 6, 4

	5	нет	[1, 1, 1, 0, 1, 1, 1, 1, 1]	2, 1, 0	8, 7, 6, 4, 5
	2	нет	[1, 1, 1, 0, 1, 1, 1, 1, 1]	2, 1, 0	8, 7, 6, 4, 5, 2
	1	3	[1, 1, 1, 0, 1, 1, 1, 1, 1]	1, 0	8, 7, 6, 4, 5, 2
	3	нет	[1, 1, 1, 1, 1, 1, 1, 1, 1]	1, 0	8, 7, 6, 4, 5, 2, 3

	1	нет	[1, 1, 1, 1, 1, 1, 1, 1, 1]	1, 0	8, 7, 6, 4, 5, 2, 3, 1
	0	нет	[1, 1, 1, 1, 1, 1, 1, 1, 1]	0	8, 7, 6, 4, 5, 2, 3, 1, 0

Следовательно, для данного графа топологическая сортировка имеет вид — 0 — > 1 — > 3 — > 2 — > 5 — > 4 — > 6 — > 7 — > 8.

Перерисуем данный граф в найденном виде:



## 4.5 Эйлеров и Гамильтонов цикл

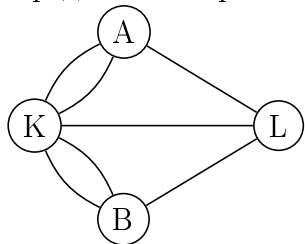
Впервые теория графов возникла в статье Эйлера в 1736 году. Это задача о кенигсбергских мостах.

На рисунке 5 представлена схема старого Кенигсберга. Есть река, на которой 2 острова. Эти острова и берега соединены семью мостами. Задача состоит в том, чтобы определить можно ли пройти по всем мостам по одному разу.



Рис. 5: Иллюстрация асимптотических обозначений.

Представим берега как вершины, а мосты — как ребра неориентированного мультиграфа.



Очевидно, что для того, чтобы пройти по каждому мосту только по одному разу, необходимо, чтобы степень всех вершин была четной (по одному мосту пришли в вершину, по другому вышли). В данном случае степени вершин:  $K=5$ ,  $A=3$ ,  $L=3$ ,  $B=3$ . Все нечетные, следовательно, задача не имеет решения.

*Эйлеров путь* — путь, проходящий по всем ребрам и притом только по одному разу.

*Эйлеров цикл* — Эйлеров путь, являющийся циклом.

*Эйлеров граф* — граф, содержащий эйлеров цикл.

*Полуэйлеров граф* — граф, содержащий эйлеров путь.

Для того, чтобы Эйлеров путь существовал, необходимо и достаточно, чтобы степень всех вершин, кроме может быть двух, была четной.

Если существует две вершины с нечетными степенями, эти вершины являются началом и концом Эйлерова пути.

Если все вершины имеют четную степень, следовательно, существует Эйлеров цикл.

Если существует только Эйлеров путь, то добавим «ребро» между вершинами с нечетными степенями, и будем искать Эйлеров цикл и только в конце удалим это ребро.

Алгоритм похож на обход в глубину, но помечаем не вершины, а удаляем посещенные ребра.

Рассмотрим пример.

## начало алгоритма

Определяем степени всех вершин;

Приравниваем вершины  $v1$  и  $v2$  минус единице;

**цикл**  $i = 0$  *до*  $i < N$  **выполнять**

**если** *степень вершины нечетная* **тогда**

**если**  $v1 == -1$  **тогда**

$v1 = i$

**иначе если**  $v2 == -1$  **тогда**

$v2 = i$

**иначе**

            цикла нет

**если** *есть вершины с нечетными степенями* ( $v1 \neq -1$ ) **тогда**

    Добавляем ребро  $v1 - v2$ ;

Записываем в стек любую начальную вершину (или  $v1$ );

**до тех пор, пока** *стек не пуст* **выполнять**

    Берем голову стека, не извлекая ее;

**если** *Степень вершины не равна 0* **тогда**

        Ищем смежную вершину;

        Записываем ее в стек;

        Удаляем соответствующее ребро;

**иначе**

        Записываем вершину в результирующий вектор;

        Удаляем ее из стека;

**если**  $v1 \neq -1$  (*есть фиктивное ребро*) **тогда**

**цикл**  $i = 0$  *до*  $i + 1 < N$  **выполнять**

**если**  $res[i] == v1$   ~~$\&\&$~~   $res[i+1] == v2$  **||**  $res[i] == v1$   ~~$\&\&$~~   $res[i+1] == v2$  **тогда**

            Сначала записываем в результат данные от  $i + 1$  до конца;

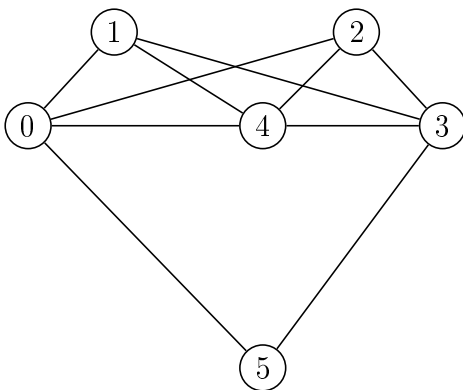
            Потом данные от 1 до  $i$ , включая  $i$ ;

**если** *в графе остались ребра* **тогда**

    он несвязный, значит Эйлера цикла нет;

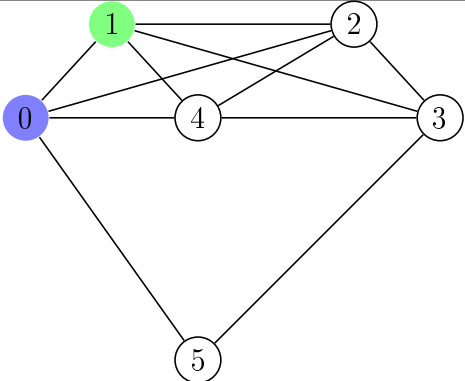
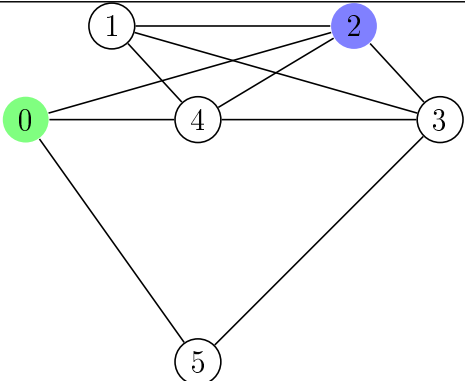
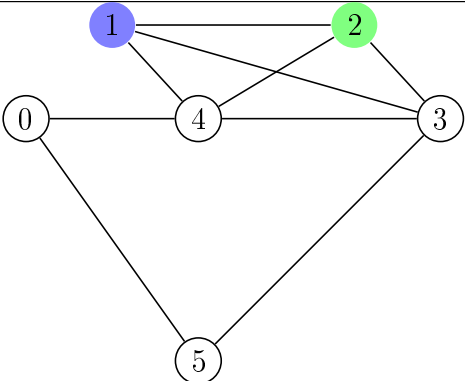
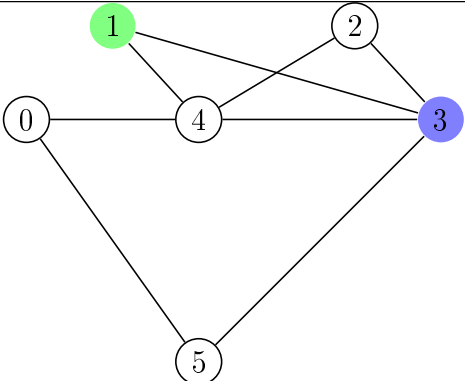
Выводим путь;

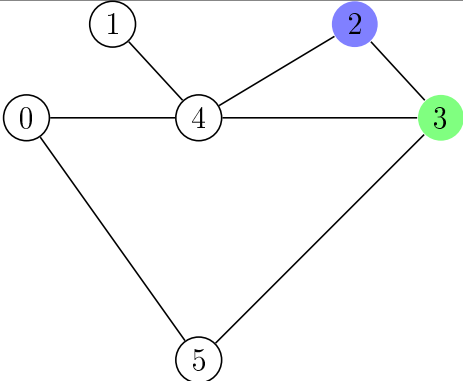
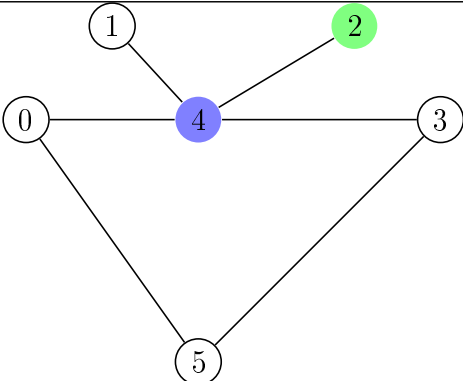
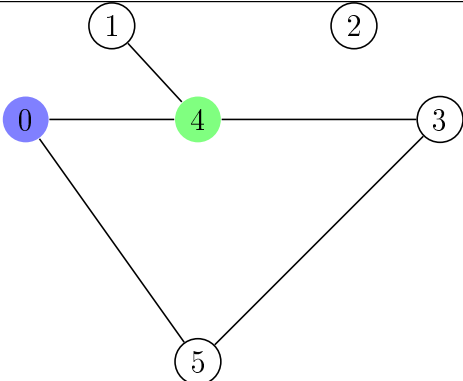
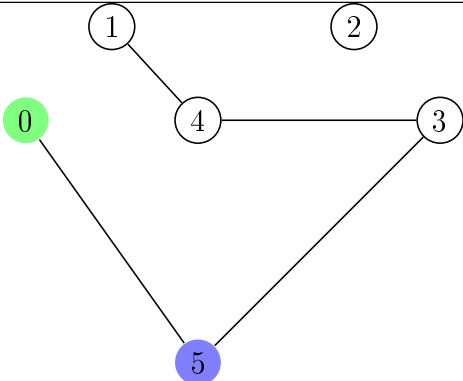
**Алгоритм 11:** Поиск Эйлера цикла для неориентированного графа

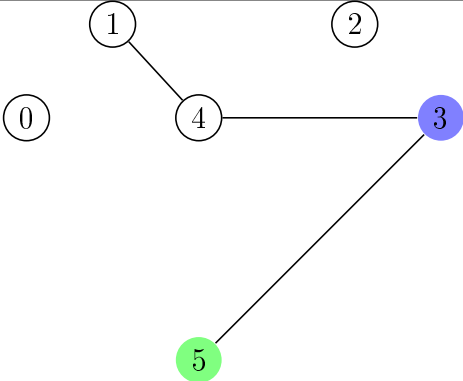
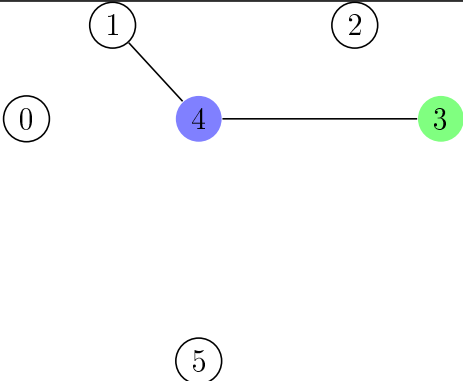
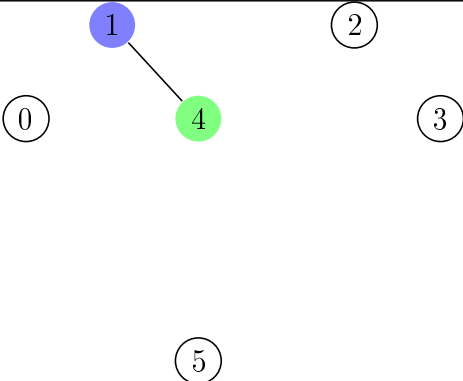
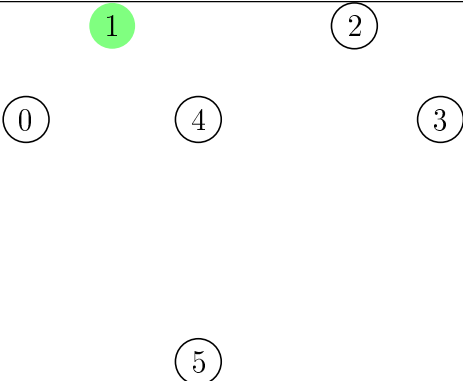


Степени вершин:  $0 - 4$ ,  $1 - 3$ ,  $2 - 3$ ,  $3 - 4$ ,  $4 - 4$ ,  $5 - 2$ . Две вершины с нечетными степенями, следовательно, в графе существует Эйлеров путь. Добавляем «мнимое ребро»  $1 - 2$ .

Граф	Текущая	Смежная	Степень	стек
------	---------	---------	---------	------

	1	0	4	1
	0	2	3	0,1
	2	1	3	2,0,1
	1	3	2	1,2,0,1

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

	5	3	1	5, 0, 4, 2, 3, 1, 2, 0, 1
	3	4	1	3, 5, 0, 4, 2, 3, 1, 2, 0, 1
	4	1	1	4, 3, 5, 0, 4, 2, 3, 1, 2, 0, 1
	1	нет	0	1, 4, 3, 5, 0, 4, 2, 3, 1, 2, 0, 1

Теперь граф содержит только изолированные вершины, дальше из стека будут доставаться вершины и записываться в результирующий вектор.

Теперь необходимо удалить «мнимое ребро».

В векторе  $1 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 0 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 0 \rightarrow 1$  индекс начала мнимого ребра  $i = 8$ . Следовательно, в результирующий вектор записываем, сначала начиная с  $i = 9$  до конца:  $2 \rightarrow 0 \rightarrow 1$ , потом начиная с  $i = 1$  до  $i = 8$ :  $2 \rightarrow 0 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 0 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 1$ .

По аналогии с Эйлеровым циклом вводится понятие *Гамильтонова цикла* — необходимо посетить



все вершины графа не более одного раза.

Примером Гамильтонова цикла является граф в виде пятиконечной звезды.

Не существует простого условия существования Гамильтонова цикла. Данная задача относится к так называемым  $NP$  полным задачам, то есть не существует доказанного алгоритма решения данной задачи, но и не доказано, что такого алгоритма не существует. К задачам поиска Гамильтонова цикла относится, например, задача коммивояжера.