БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

АЛГОРИТМЫ НА ГРАФАХ

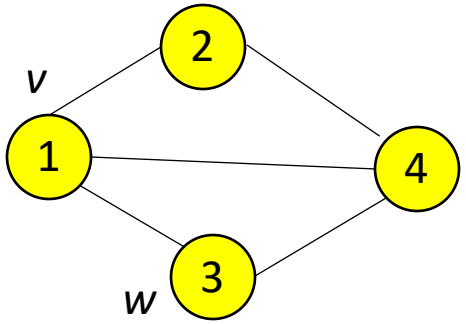
Поиск в ширину (англ. **BFS** - **B**readth **F**irst **S**earch)

Эйлеров цикл в графе

Поиск в глубину (англ. **DFS - D**epth **F**irst **S**earch)

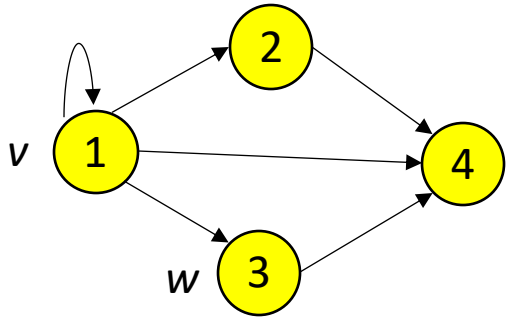
https://github.com/larandaA/alg-ds-snippets

Граф



$$G = (V, E),\ E \subseteq V^{(2)},$$
$$V^{(2)} = \{U \subseteq V : |U| = 2\}$$

Орграф



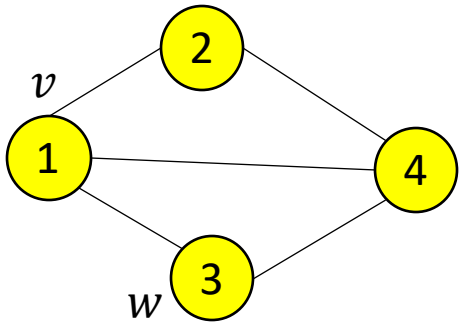
$$G = (V, E),\ E \subseteq V^2,$$
$$V^2 = V \times V = \{(u, v) : u, v \in V\}$$

$|V| = \boldsymbol{n}$ – число вершин в графе (орграфе)
$|E| = \boldsymbol{m}$ – число ребер (дуг) в графе (орграфе)

Структуры данных для представления графа (орграфа)

1. Матрица смежности
2. Списки смежности
3. Матрица инцидентности
4. Списки дуг

Матрица смежности. Списки смежности

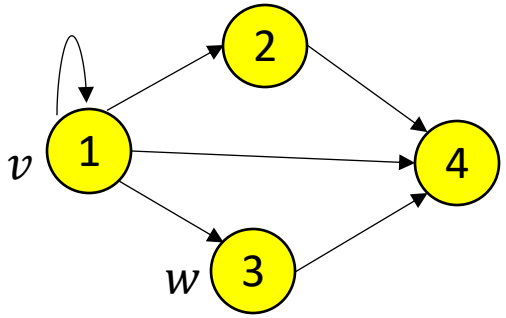


$$A_G = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 1 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

$$L_G = \begin{array}{c|l} 1 & \rightarrow \boxed{2\ 3\ 4} \\ 2 & \rightarrow \boxed{1\ 4} \\ 3 & \rightarrow \boxed{1\ 4} \\ 4 & \rightarrow \boxed{1\ 2\ 3} \end{array}$$

$\Theta(n^2)$ $\Theta(n+m)$

Если матрицу смежности A_G возвести в степень k, то $\boldsymbol{A_G^k[v,w]}$ – число попарно различных (v,w) - маршрутов длины k.

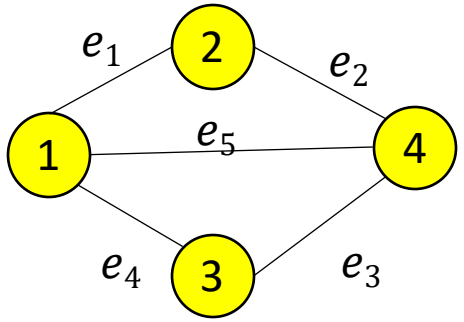


$$A_G = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 1 & 1 & 1 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 \end{array}$$

$$L_G = \begin{array}{c|l} 1 & \rightarrow \boxed{1\ 2\ 3\ 4} \\ 2 & \rightarrow \boxed{4} \\ 3 & \rightarrow \boxed{4} \\ 4 & \end{array}$$

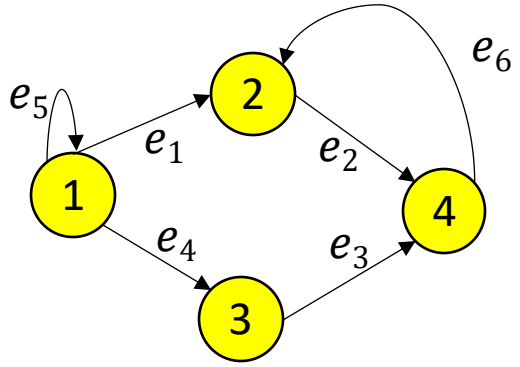
$\Theta(n^2)$ $\Theta(n+m)$

Матрица инцидентности



	e_1	e_2	e_3	e_4	e_5
1	1	0	0	1	1
2	1	1	0	0	0
3	0	0	1	1	0
4	0	1	1	0	1

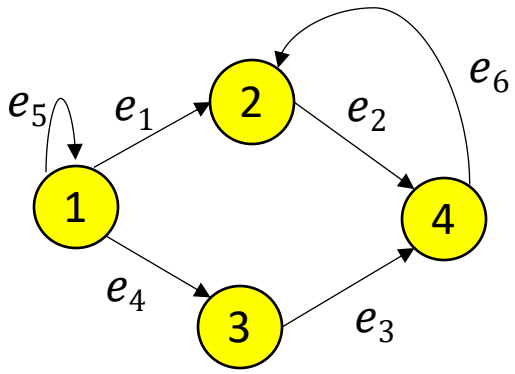
$\Theta(n \cdot m)$



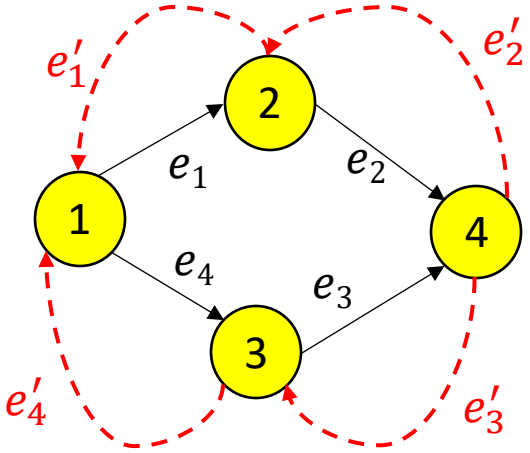
	e_1	e_2	e_3	e_4	e_5	e_6
1	1	0	0	1	2	0
2	-1	1	0	0	0	-1
3	0	0	1	-1	0	0
4	0	-1	-1	0	0	1

$\Theta(n \cdot m)$

Списки дуг



$\Theta(n+m)$



	1	2	3	4
$List(i)$	~~0~~,~~1~~,~~4~~, 5	~~0~~, 2	~~0~~, 3	~~0~~, 6

1	2	3	4
~~0~~,~~1~~, 7	~~0~~,~~2~~, 3	~~0~~, ~~5~~ 8	~~0~~, ~~4~~, 6

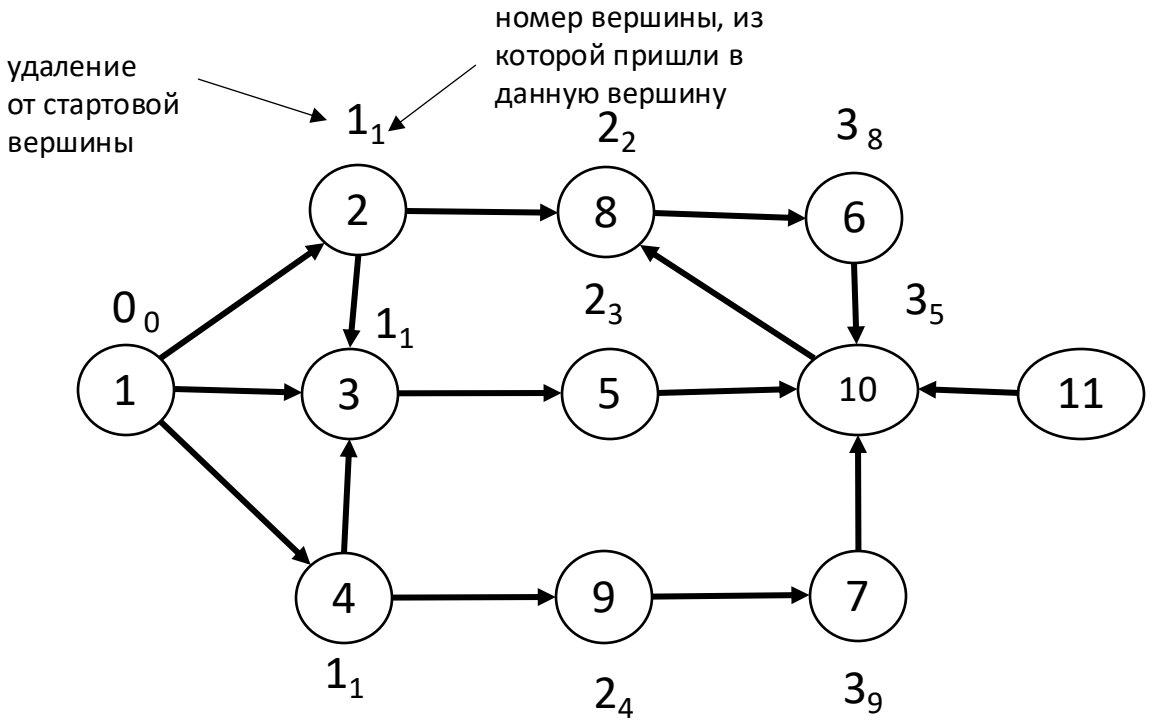
$ArcList$

	e_1 1	e_2 2	e_3 3	e_4 4	e_5 5	e_6 6
1	2	4	4	3	1	2
2	$w(e_1)$	$w(e_2)$	$w(e_3)$	$w(e_4)$	$w(e_5)$	$w(e_6)$
3	0	0	0	1	4	0

	e_1 1	e'_1 2	e_2 3	e'_2 4	e_3 5	e'_3 6	e_4 7	e'_4 8
1	2	1	4	2	4	3	3	1
2	$w(e_1)$	$w(e'1)$	$w(e_2)$	$w(e'2)$	$w(e_3)$	$w(e'3)$	$w(e_4)$	$w(e'4)$
3	0	0	2	0	0	4	1	5

Поиск в ширину
(англ. **BFS** - **B**readth **F**irst **S**earch)

Поиск в ширину: обход всех вершин в порядке удаленности от стартовой вершины.



Queue: ~~1 2 3 4 8 5 9 6 10 7~~

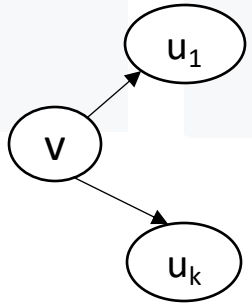
i	1	2	3	4	5	6	7	8	9	10	11
Predsessor	0	1	1	1	3	8	9	2	4	5	-1

канонический способ задания корневого дерева поиска в ширину в орграфе

Задание графа списками смежности

```
def bfs(start):
    q = queue()

    visited[start] = True
    q.enqueue(start)

    while not q.empty():
        v = q.dequeue()

        for u in g[v]:
            if not visited[u]:
                visited[u] = True
                q.enqueue(u)
```

Время работы $O(n + m)$



Задание графа матрицей смежности

```
def bfs(start):
    q = queue()

    visited[start] = True
    q.enqueue(start)

    while not q.empty():
        v = q.dequeue()

        for u in range(n):
            if a[v][u] and not visited[u]:

                visited[u] = True
                q.enqueue(u)
```

Время работы $O(n^2)$

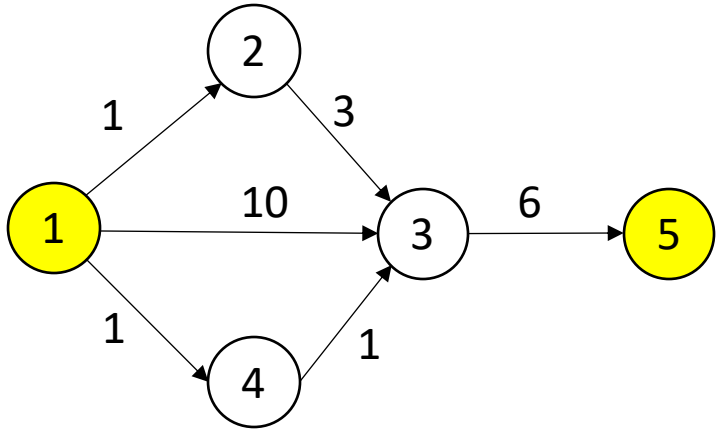
Терминология

- ✓ **кратчайший (v, w)-путь**

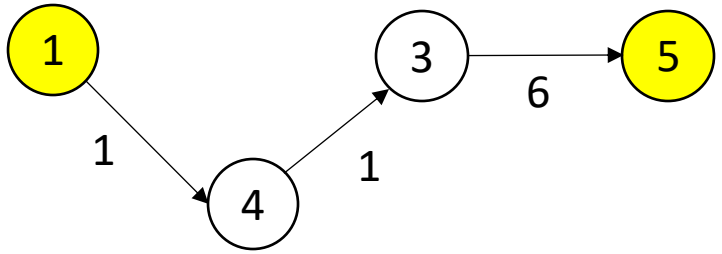
  задан взвешенный орграф; необходимо найти такой путь между заданной парой вершин v и w, для которого сумма стоимостей входящих в него дуг минимальна;

- ✓ **наименьший (v, w)-путь**

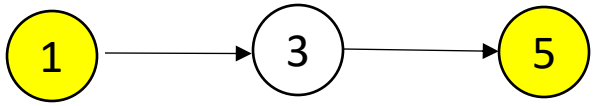
  задан орграф; необходимо найти такой путь между заданной парой вершин v и w, длина которого минимальна (длина – число дуг пути);



кратчайший путь между вершинами 1 и 5 (стоимость 8):



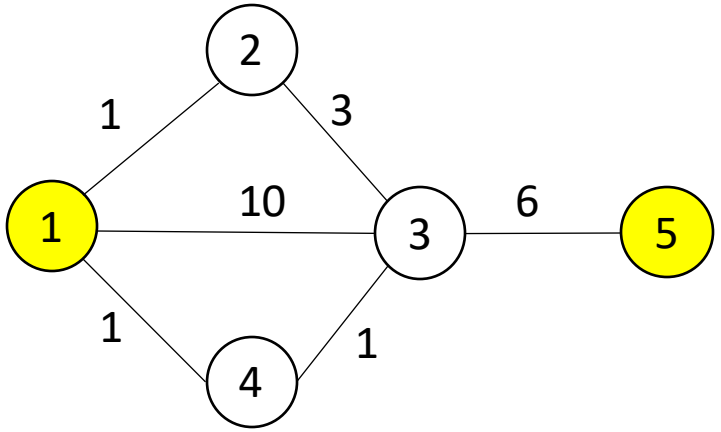
наименьший путь между вершинами 1 и 5 (длина 2):



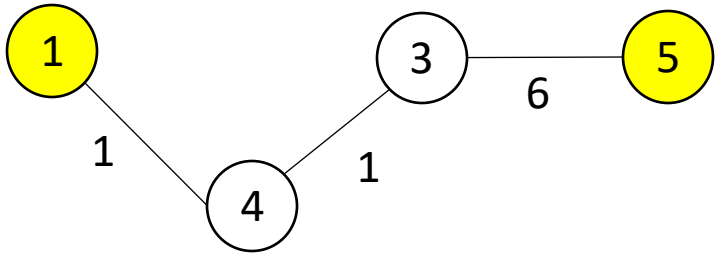
✓ **кратчайшая простая (v, w)- цепь**

задан взвешенный граф; необходимо найти такую простую цепь между заданной парой вершин v и w, для которой сумма стоимостей входящих в неё рёбер минимальна;

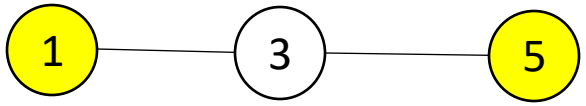
✓ **наименьшая простая (v, w)- цепь** – задан граф; необходимо найти простую цепь между заданной парой вершин v и w, длина которой минимальна (длина – число рёбер цепи);



кратчайшая простая цепь между вершинами 1 и 5 (стоимость 8):



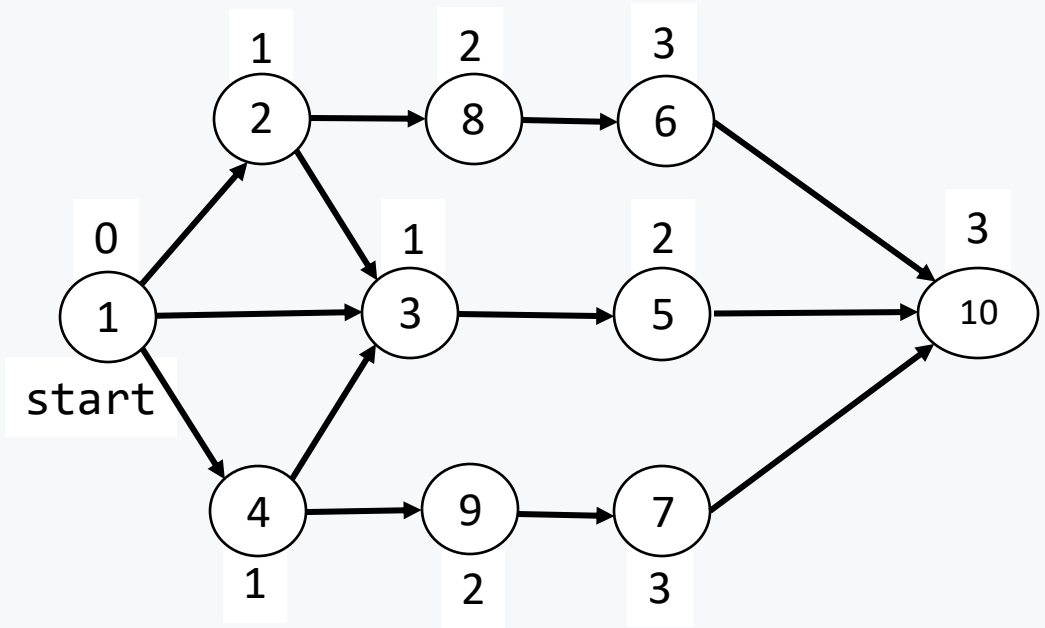
наименьшая простая цепь между вершинами 1 и 5 (длина 2):



BFS находит **наименьший путь** между стартовой вершиной поиска в ширину (`start`) и всеми, достижимыми из неё вершинами;

Если хотим найти растояние в ребрах до всех вершин:

```
def bfs(start):
    q = queue()

    dist[start] = 0 # <-- здесь!

    visited[start] = True
    q.enqueue(start)

    while not q.empty():
        v = q.dequeue()

        for u in g[v]:
            if not visited[u]:
                visited[u] = True

                dist[u] = dist[v] + 1 # <-- и здесь!

                q.enqueue(u)
```

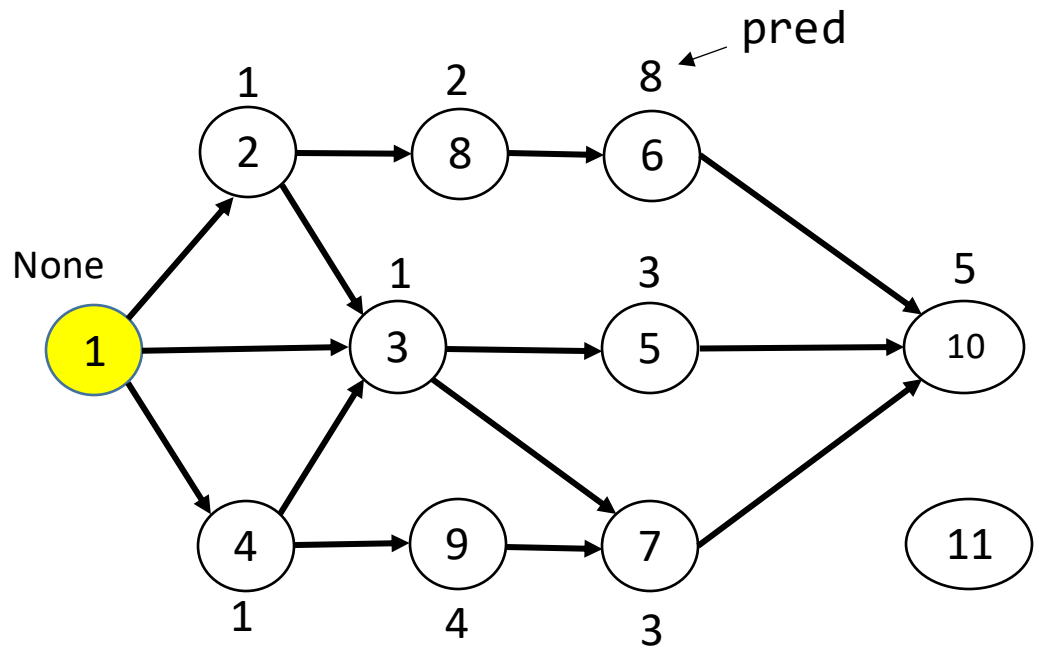


где `dist` -- массив, хранящий расстояния от стартовой вершины, изначально заполненный `-1`, `None` или `INF`.

BFS

находит **наименьший путь** между стартовой вершиной поиска в ширину (`start`) и всеми, достижимыми из неё вершинами; если необходимо восстановить последовательность вершин наименьшего пути, то для этого нужно сформировать массив **`pred`:**

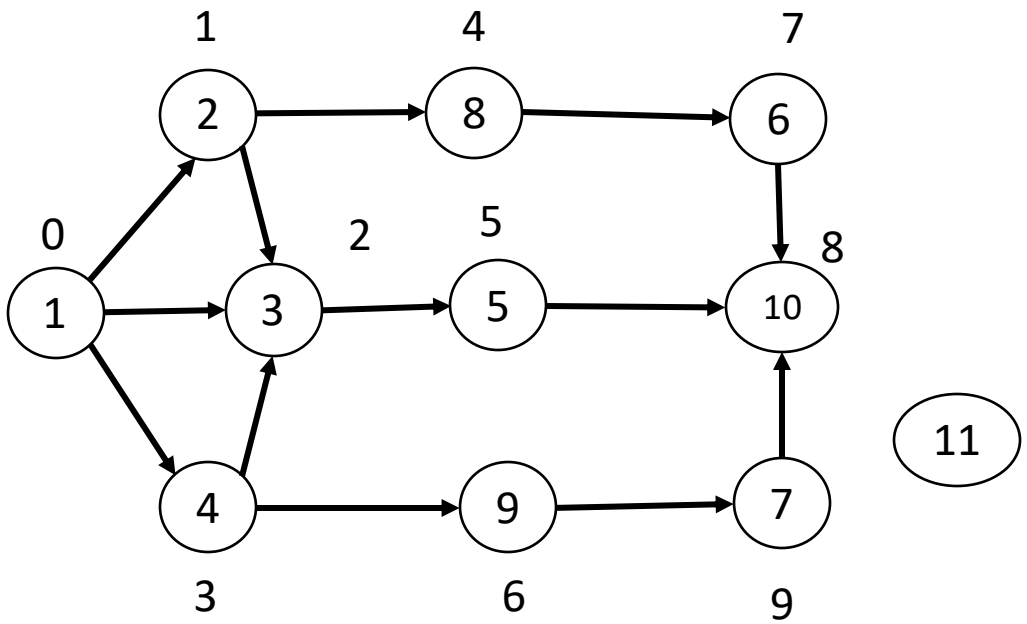
```
def bfs(start):
    q = queue()

    pred[start] = None # <-- здесь!

    visited[start] = True
    q.enqueue(start)

    while not q.empty():
        v = q.dequeue()

        for u in g[v]:
            if not visited[u]:
                visited[u] = True

                pred[u] = v # <-- и здесь!

                q.enqueue(u)
```



	1	2	3	4	5	6	7	8	9	10	11
$pred[i]$	None	1	1	1	3	8	9	2	4	5	-1

Теперь пути можно восстановить по массиву `pred`.

Пометки вершин орграфа в порядке обхода **BFS**:



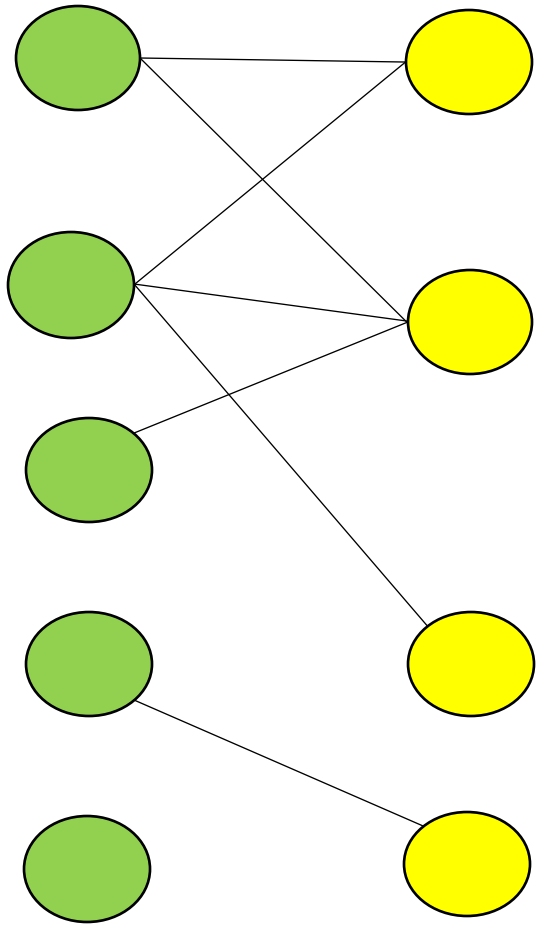

```
next_idx = 0

def bfs(start):
    q = queue()

    ord[start] = next_idx # <-- вот
    next_idx += 1         # <-- здесь!

    visited[start] = True
    q.enqueue(start)

    while not q.empty():
        v = q.dequeue()

        for u in g[v]:
            if not visited[u]:
                visited[u] = True

                ord[u] = next_idx # <-- и вот
                next_idx += 1     # <-- здесь!

                q.enqueue(u)
```

BFS можно использовать для определения **двудольности графа** (свойством двудольного графа является то, что если две вершины графа смежны, то они принадлежат разным долям)

1844-1944
венгерский математик

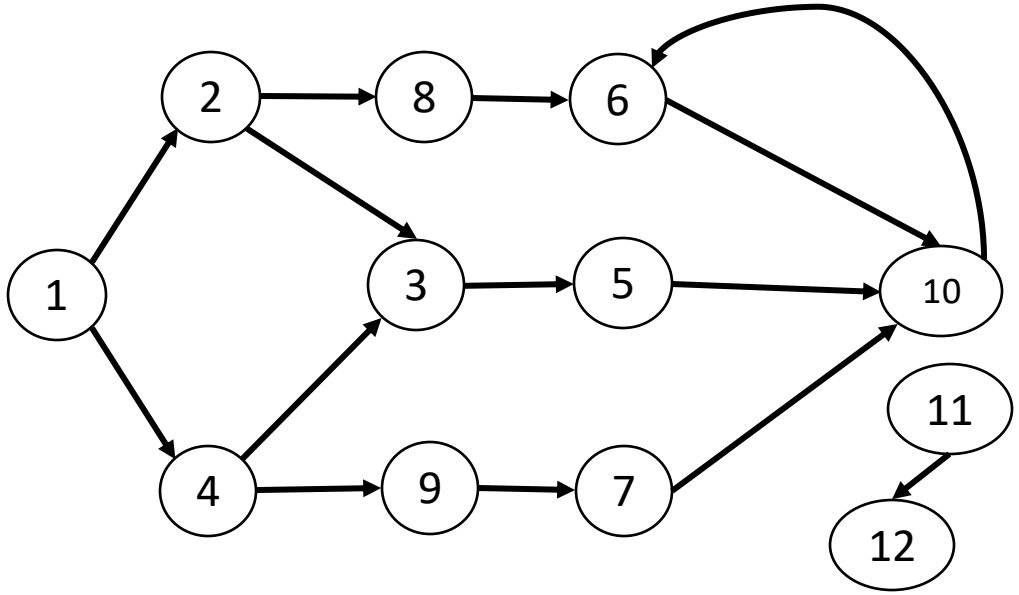
Теорема Д.Кёнига (1936 год)

Для двудольности графа необходимо и достаточно, чтобы он не содержал циклов нечётной длины.



Для **определения двудольности ориентированного графа** нужно сначала **перейти к его основанию** (т.е. дуги орграфа заменить рёбрами). Если полученный псевдограф - двудольный, то и исходный орграф - двудольный.

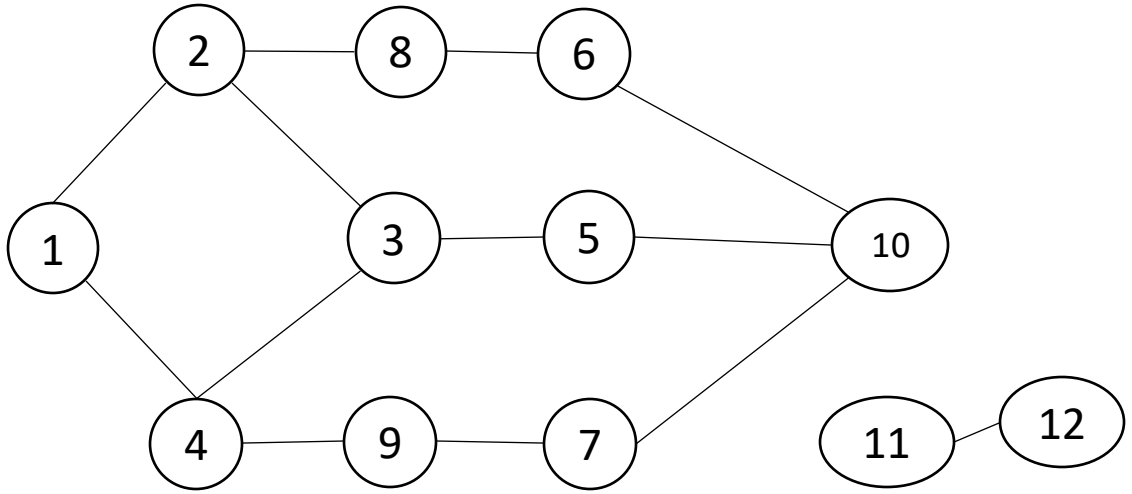
Орграф G



Если в орграфе были петли, то он не двудольный.

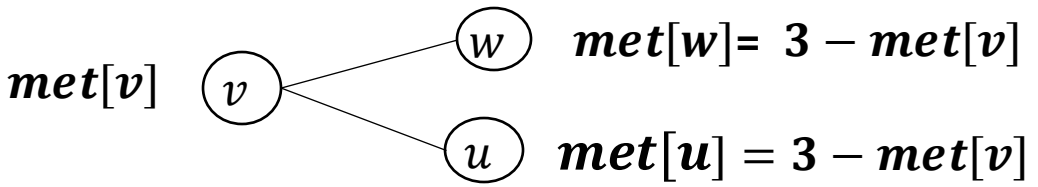
Если в орграфе были противоположно направленные дуги, то в графе им будет соответствовать одно ребро.

Основание орграфа G



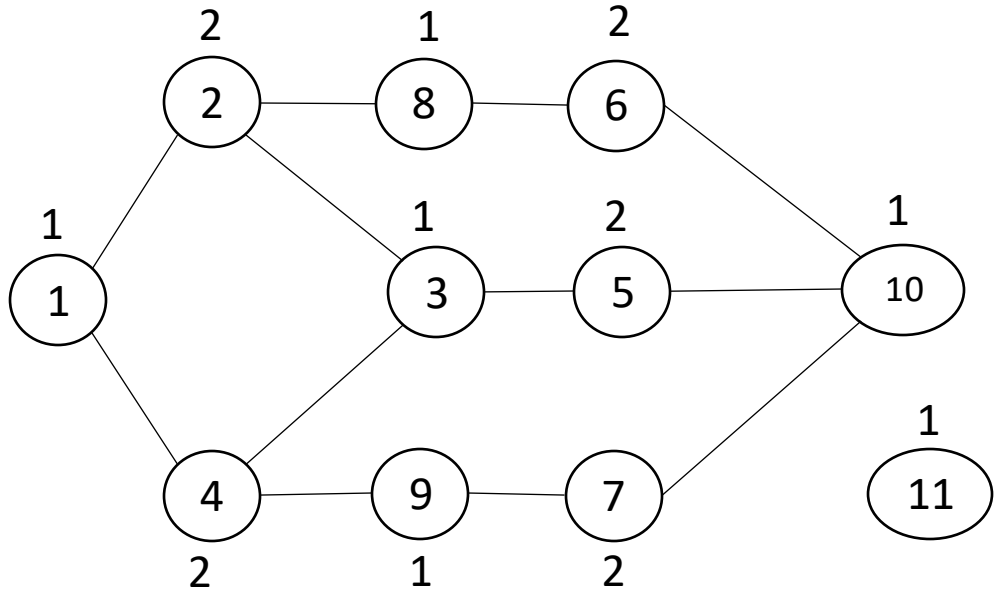
BFS можно использовать для определения **двудольности графа**

Присвоив метку стартовой вершине v, мы определим метки всех смежных с ней вершин как противоположные, и так далее:



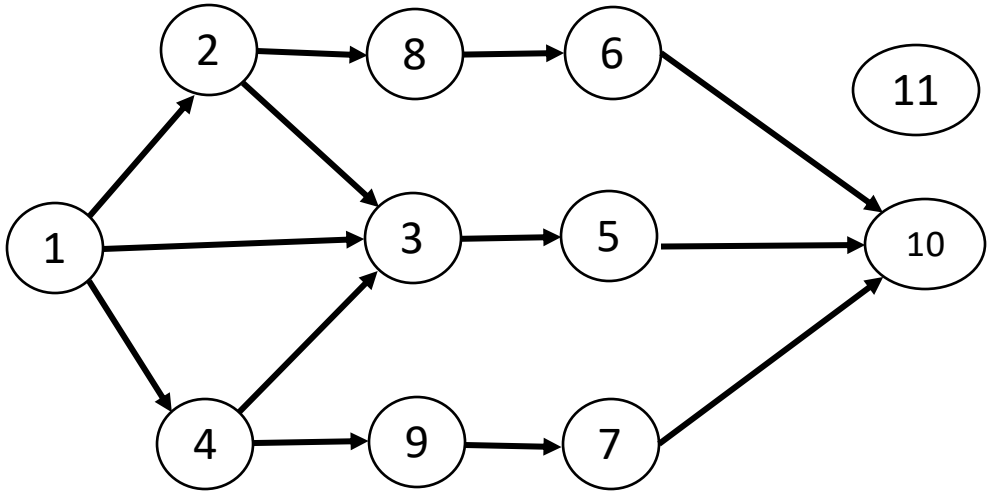
При обнаружения цикла (вершина w, которая смежна с текущей вершиной v, уже была посещена) мы сравним метки вершин v и w. Если окажется, что вершины v и w принадлежат одной доле $(met[w] = met[v])$, то граф не является двудольным.

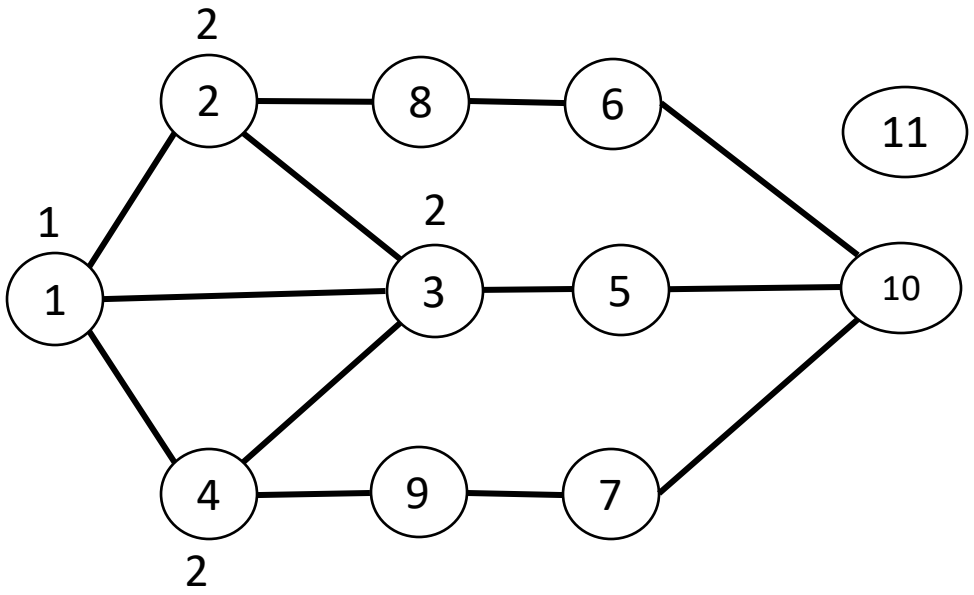
Отметим, что, если компонент связности несколько, то для двудольности графа требуется, чтобы каждая компонента связности была двудольной.



Граф является двудольным.

Определить, является ли приведённый на рисунке орграф двудольным, используя BFS.



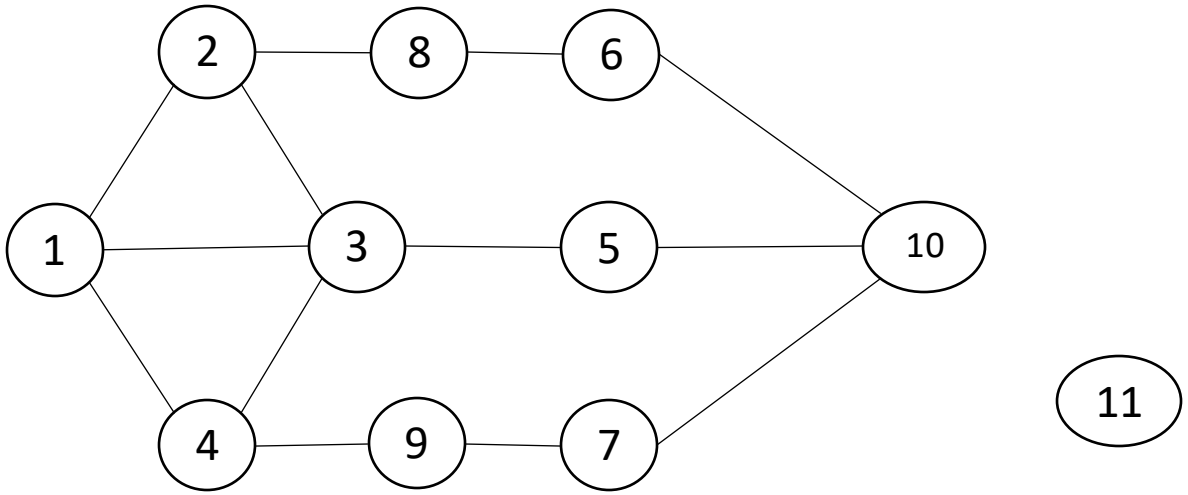


Ориентированный граф не является двудольным так как его основание не является двудольным графом.

BFS можно использовать для определения **связности графа**

Граф называется **связным**, если любые две его несовпадающие вершины соединены маршрутом.

Всякий максимальный по включению связный подграф графа G (т.е. не содержащийся в связном подграфе с большим числом элементов) называется **связной компонентой графа G.**



У графа, изображённого на рисунке, две связные компоненты.

```
for v in range(n):
    if not visited[v]:
        bfs(v)
```

BFS можно использовать для решения следующих задач

1) Поиска маршрута между заданной парой вершин.
2) Поиска наименьшего (по количеству рёбер/дуг) маршрута между заданной парой вершин.
3) Определения двудольности графа (орграфа).
4) Выделения связных компонент графа.

В программной реализации BFS используется структура данных **ОЧЕРЕДЬ (queue).**

Время работы BFS есть

$O(n + m)$**,** если граф задан **списками смежности**;

$O(n^2)$**,** если граф задан **матрицей смежности**.

Эйлеров цикл в графе

Для связного графа **эйлеров цикл** – это цикл, который содержит все рёбра графа.

Связный граф, обладающий эйлеровым циклом, называется **эйлеровым графом**.

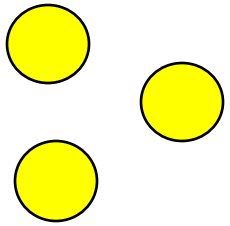
Леонард **Эйлер**
(1707-1783)
швейцарский
немецкий
российский математик

Необходимое и достаточное условие, Л. Эйлер, 1736 год.

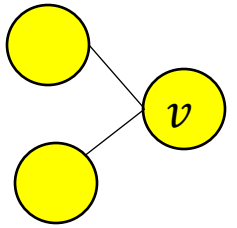
Нетривиальный связный граф содержит эйлеров цикл тогда и только тогда, когда степени всех его вершин чётны.

тривиальный

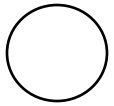
пустой



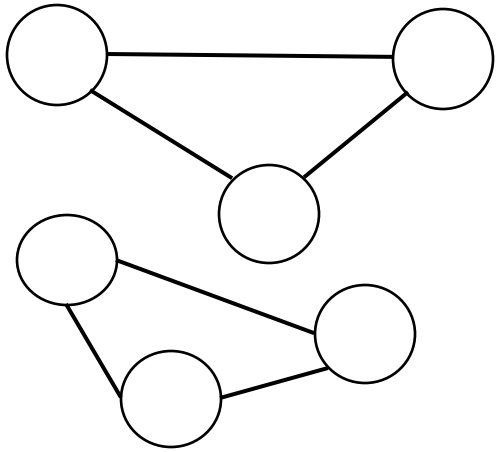
Степень вершины v равна числу инцидентных ей рёбер.



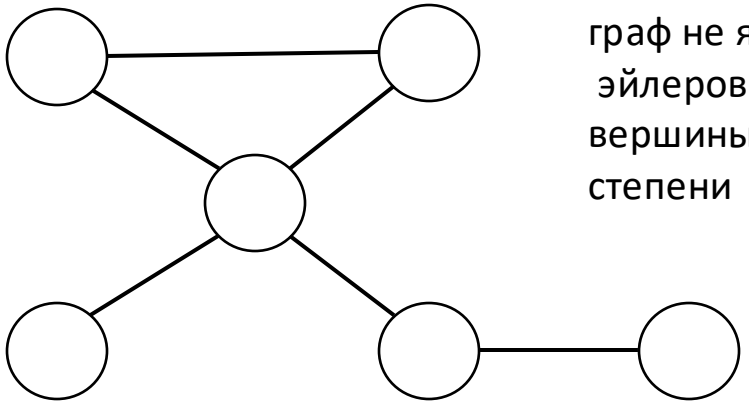
степень $(v) = 2$



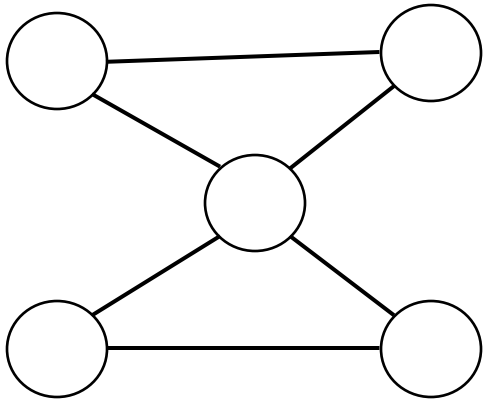
граф -
тривиальный



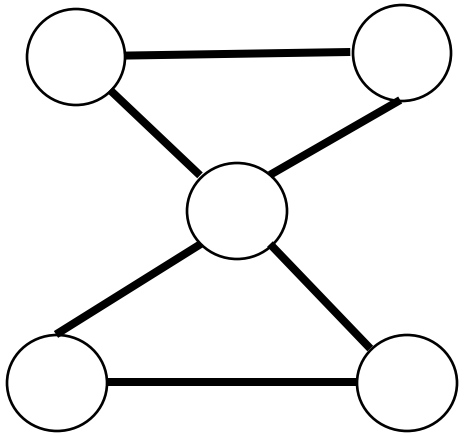
граф должен
быть связным

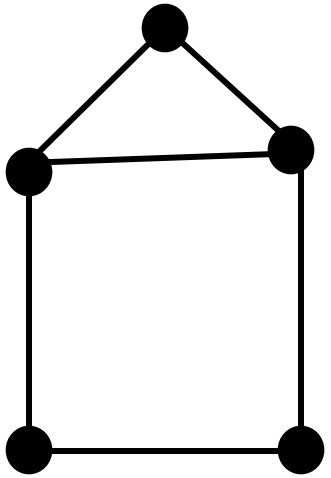


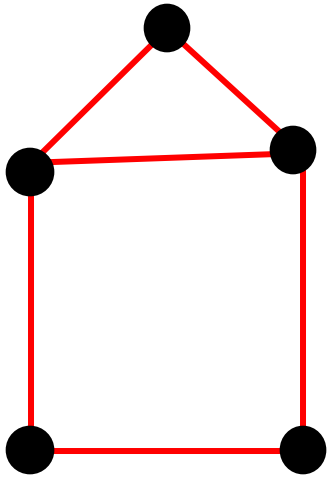
граф не является
эйлеровым т.к. есть
вершины нечётной
степени



эйлеров граф







Граф эйлеров, если мы можем его нарисовать, не отрывая руку от бумаги: стартуем из любой вершины, прорисовываем каждое ребро ровно один раз и обязательно возвращаемся в стартовую вершину.

Алгоритм построения эйлерова цикла в графе

1. Абстрактные типы данных **очередь** (queue) и **стек** (stack) – пустые.

2. Выбрать произвольную вершину графа и добавить её в **стек**.

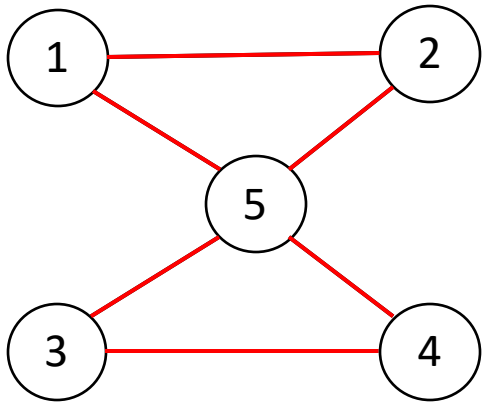
3. Пока **стек** не пуст, выполнять следующие действия:

   пусть **v** – вершина, которая последняя была занесена в **стек** :

   a) если существует ребро **(v,w)**, то добавляем вершину **w** в **стек** и удаляем ребро **(v,w)** из графа ;

   b) если не существует ребра, инцидентного вершине **v**, то удаляем вершину **v** из **стека** и добавляем её в **очередь**.

4. Последовательность вершин **очереди** задаёт порядок обхода вершин в эйлеровом цикле.



Алгоритм построения эйлерова цикла графа

1. Структуры данных **очередь** (queue) и **стек** (stack) – пустые.
2. Выбрать произвольную вершину графа и добавить её в **стек**.
3. Пока **стек** не пуст, выполнять следующие действия (пусть **v** – вершина, которая последняя была занесена в **стек** :

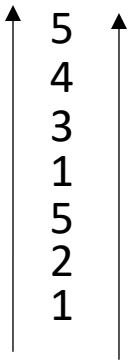
a) если существует ребро **(v,w)**, то добавляем вершину **w** в **стек** и удаляем ребро **(v,w)** из графа ;

b) если не существует ребра, инцидентного вершине **v**, то удаляем вершину **v** из **стека** и добавляем её в **очередь**.

4. Последовательность вершин **очереди** задаёт порядок обхода вершин в эйлеровом цикле.

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

для каждой строки i укажем номер столбца, начиная с которого в этой строке будет осуществляться движение при поиске вершин, смежных с вершиной i



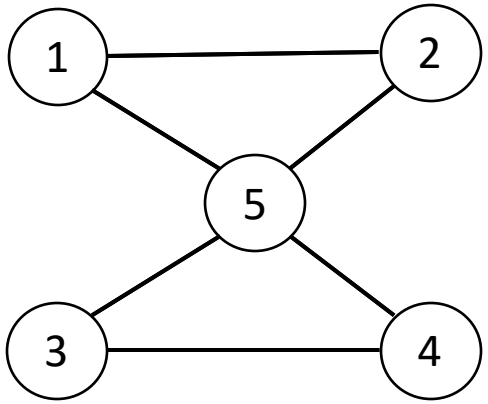
стек

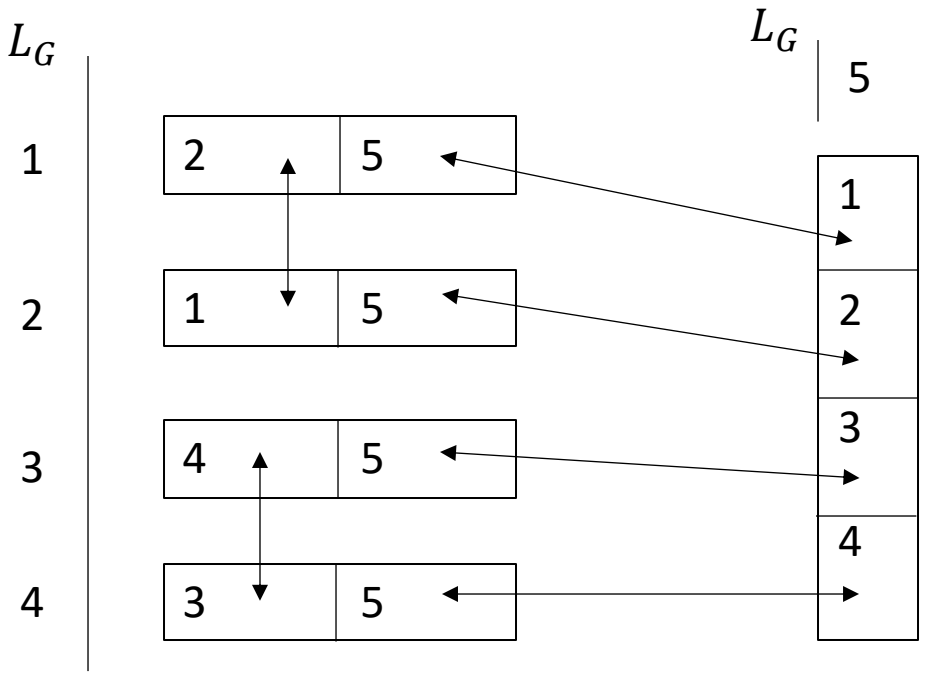
очередь 1 5 4 3 5 2 1

Время работы алгоритма построения эйлерова цикла на **матрице смежности**:

$$O(m) + O(n^2)$$

Время работы алгоритма построения эйлерова цикла на списках смежности: $O(\boldsymbol{m})$





Задача

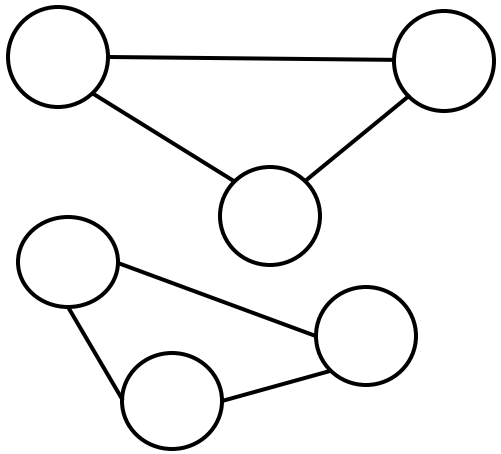
Задан граф (не обязательно связный).

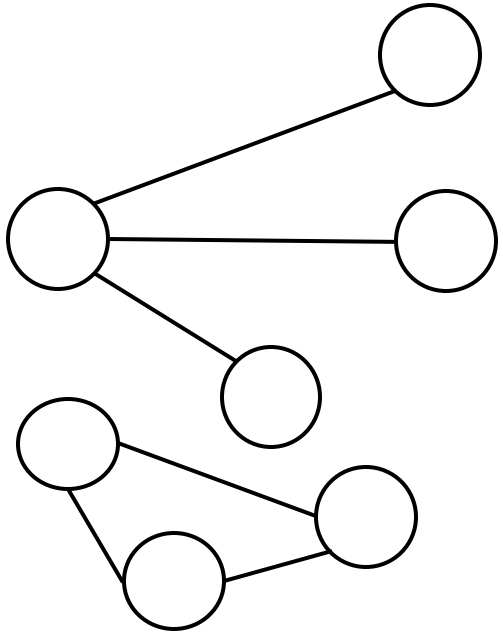
Необходимо **покрыть граф наименьшим числом рёберно-непересекающихся цепей**.

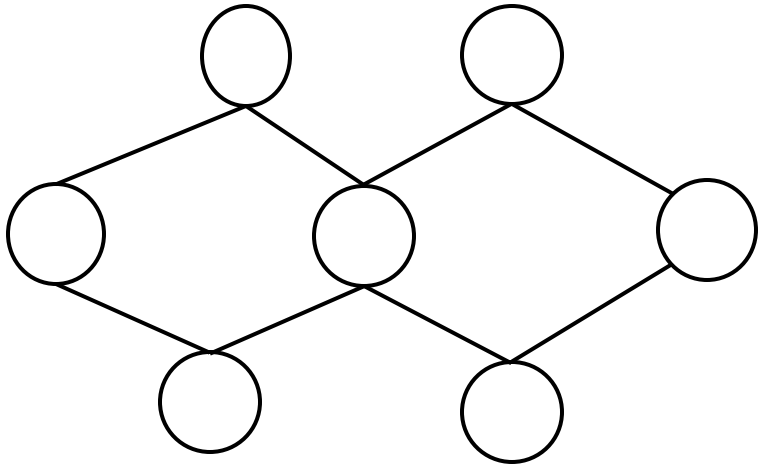
Цепи могут быть открытыми или замкнутыми.

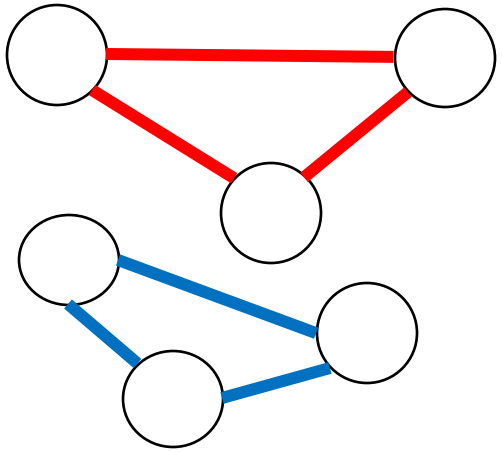
Каждое ребро графа должно входить в одну из этих цепей.

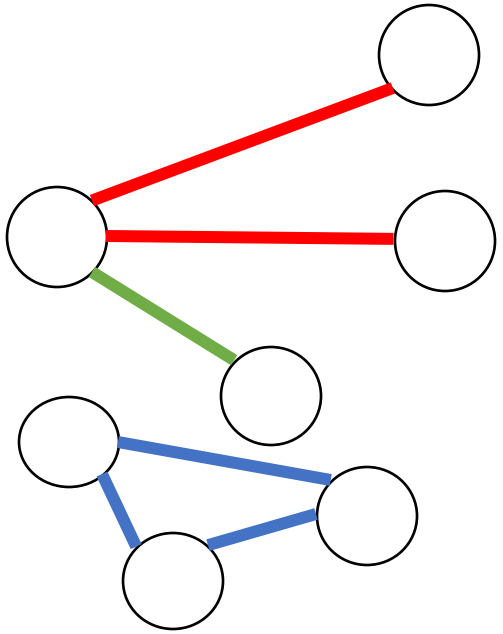
Если граф не связный, то решаем задачу оптимально для каждой компоненты связности.

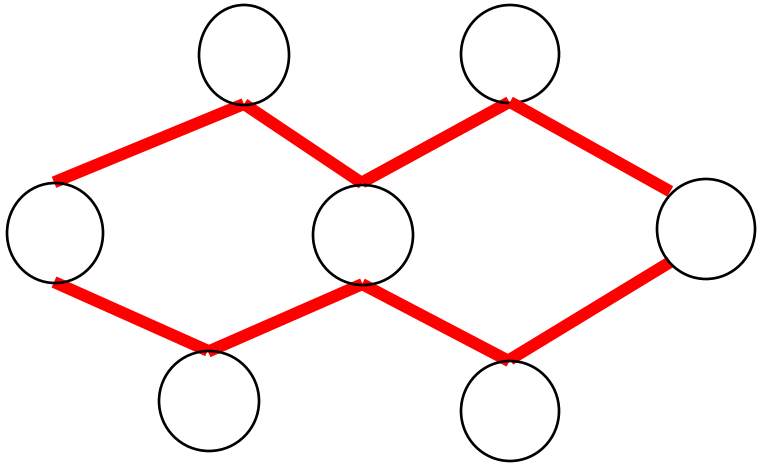


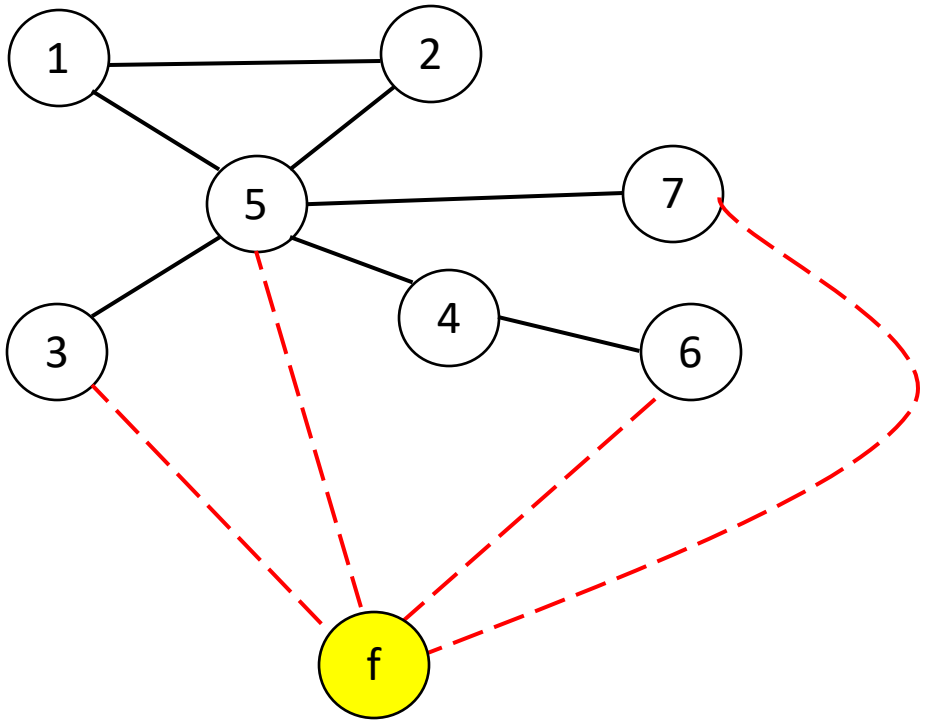










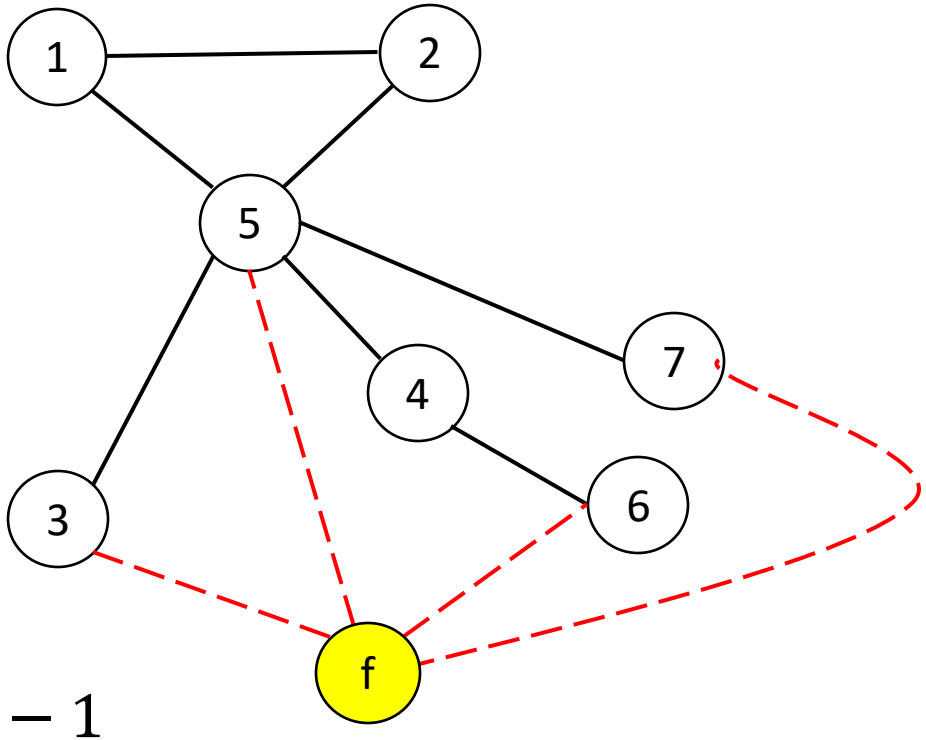


1. Если граф содержит эйлеров цикл, то задача решена – строим эйлеров цикл и покрываем граф одной замкнутой цепью эйлеровым циклом).

2. Если граф не содержит эйлеров цикл, значит в нём , есть **вершины нечётной степени**. Предположим, что их – $\boldsymbol{k}$ штук. Отметим, что число k – чётно (лемма о рукопожатиях).

3. Введём фиктивную вершину $\boldsymbol{f}$, которую соединим рёбрами со всеми вершина нечётной степени. Теперь степени всех вершин – чётны.

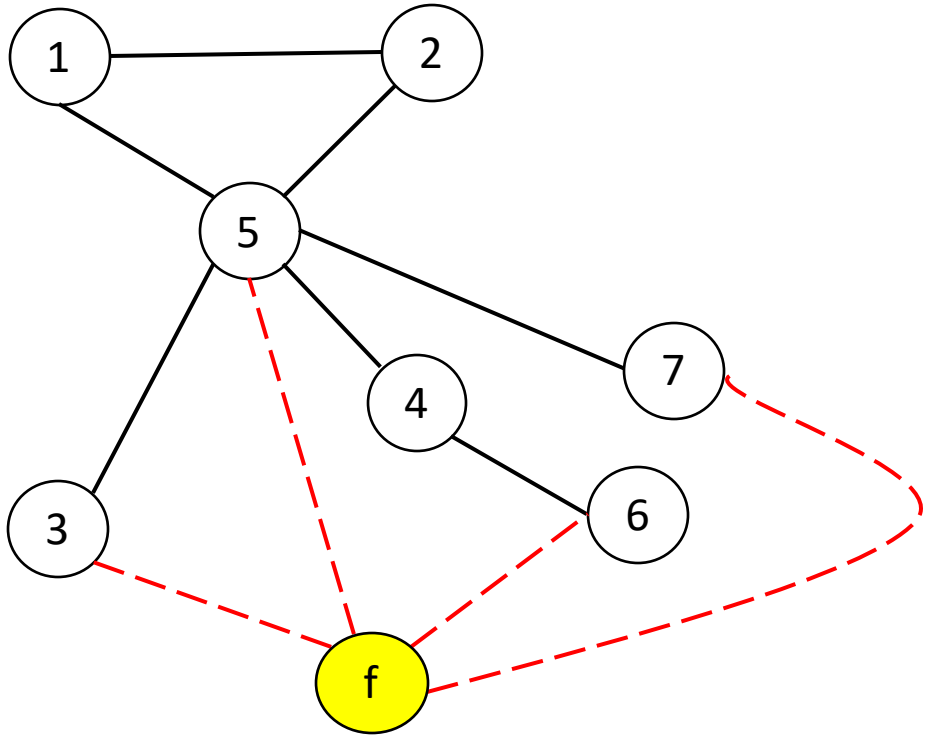
4. Построим в преобразованном графе эйлеров цикл.



Эйлеров цикл :

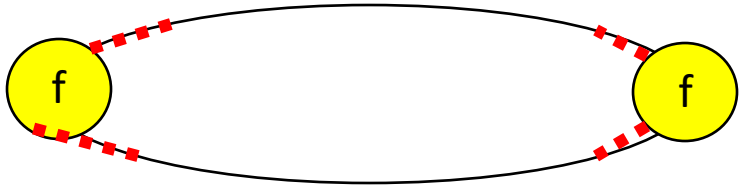
$$1 - 5 - 7 - f - 6 - 4 - 5 - f - 3 - 5 - 2 - 1$$

(продолжение)



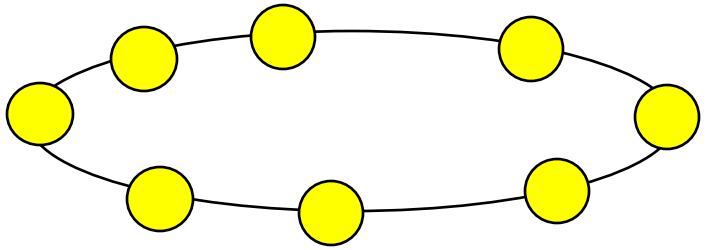
$k = 4$ – число вершин нечётной степени

5. Удалим из эйлерова цикла вершину f и инцидентные ей рёбра.



фиктивная вершина встречается в цикле $k/2$ раз

Если k – число вершин нечётной степени, то эйлеров цикл распадется на $\boldsymbol{k/2}$ **рёберно-непересекающихся цепей.**



1-е удаление — 1

2-е удаление — 1 | 2

3-е удаление — 1 | 2 | 3

$k/2$ удаление — 1 | 2 | … | $k/2$

Обоснование оптимальности

Пусть k - число вершин нечётной степени и мы покрыли граф k /2 цепями.

Предположим, что граф можно покрыть меньшим числом цепей, например,

$$\frac{k}{2} - 1.$$

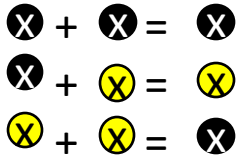
Подсчитаем для каждой цепи число вершин нечетной степени в ней.

Цепи, которыми покрываем граф, могли быть открытыми или замкнутыми.

Если цепь открытая, то в ней только 2 конечные вершины имеют нечётную степень.

Если цепь замкнутая, то в ней все вершины имеют чётную степень.

Так как каждое ребро графа входит только в одну из цепей, то степень каждой вершины графа можно найти, суммируя её степень по всем цепям покрытия.

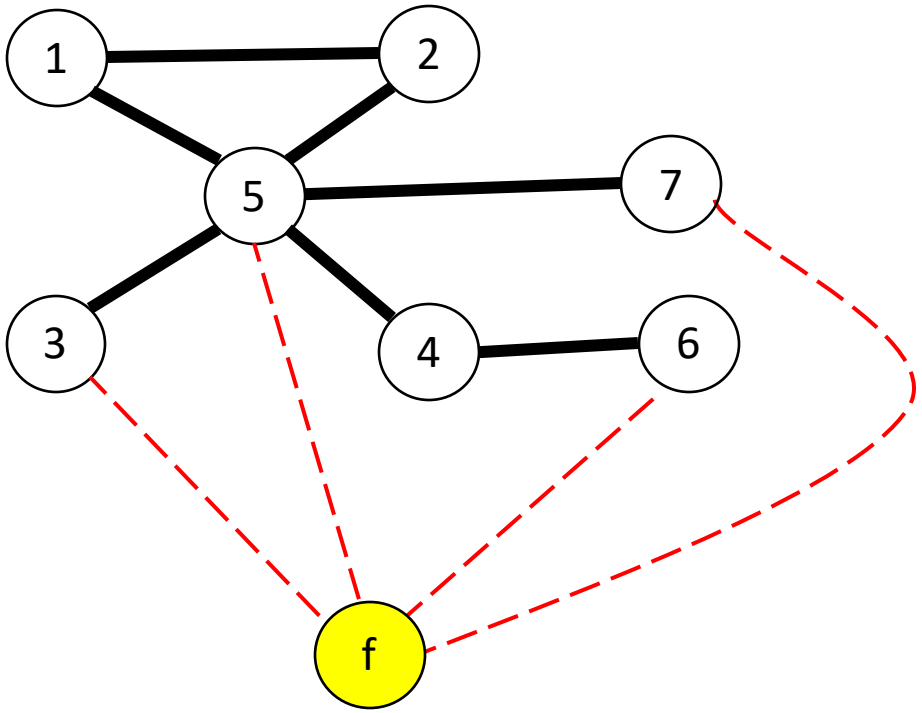


Тогда в графе **число вершин нечётной степени не больше, чем**

$$\mathbf{2 \cdot \left(\frac{k}{2} - 1 \right) = k - 2}$$

Противоречие, так как у нас число вершин нечётной степени k и $k > k - 2$.

(продолжение)



top

5
7
f
6
4
5
f
3
~~1~~
5
2
1

стек

очередь: 1 5 7 f 6 4 5 f 3 5 2 1

эйлеров цикл :

$1 - 5 - 7 - f_1 - 6 - 4 - 5 - f_2 - 3 - 5 - 2 - 1$

1-ое удаление фиктивной вершины f_1

$6 - 4 - 5 - f_2 - 3 - 5 - 2 - 1 - 5 - 7$

2-ое удаление фиктивной вершины f_2

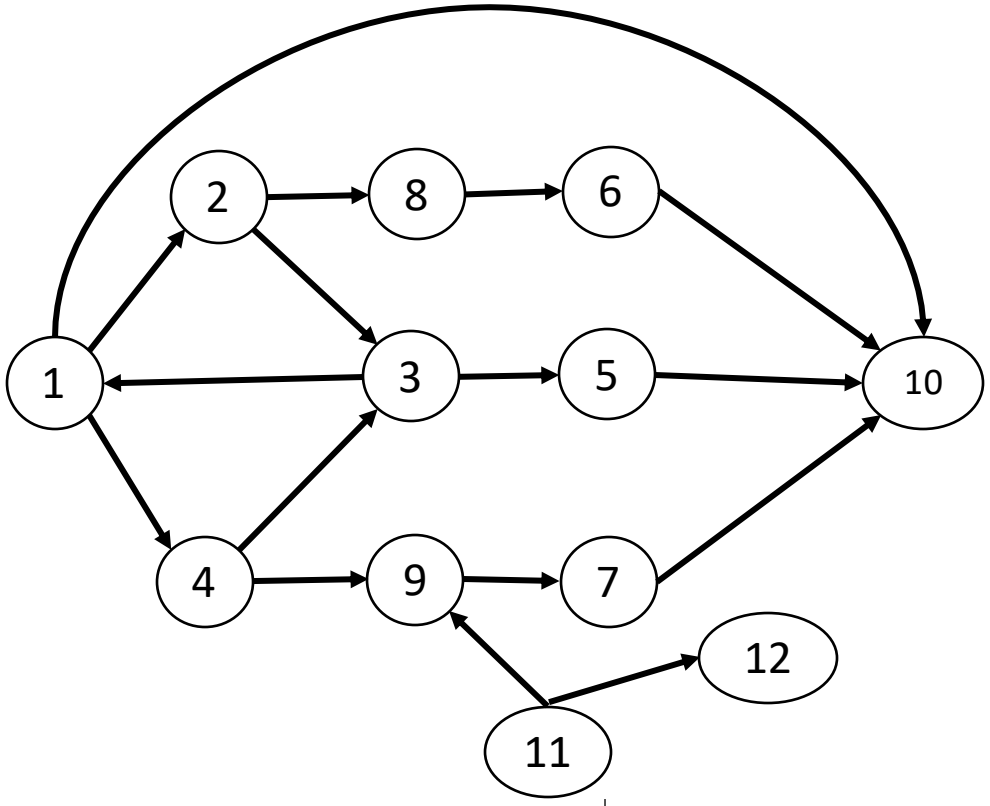
$6 - 4 - 5$

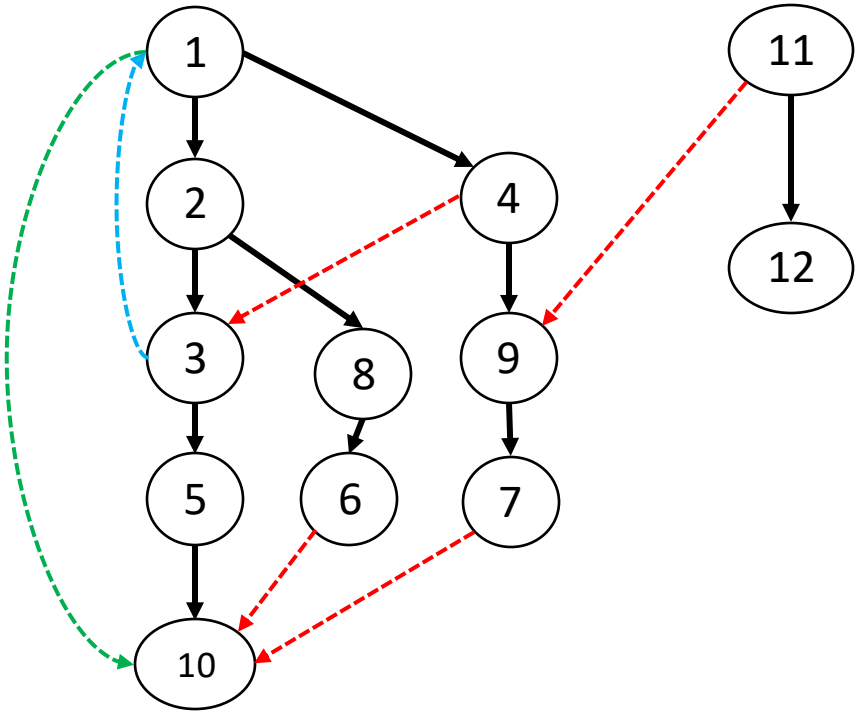
$3 - 5 - 2 - 1 - 5 - 7$

Поиск в глубину (англ. DFS - Depth First Search)

Один из способов обхода вершин и рёбер (дуг) графа (орграфа).

Поиск в глубину в орграфе:





сначала все вершины белые;

становится серой, когда первый раз приходим в неё;

становится чёрной, когда все смежные вершины уже были посещены;

древесная дуга (v,w) ведет из «серой» вершины в «белую»

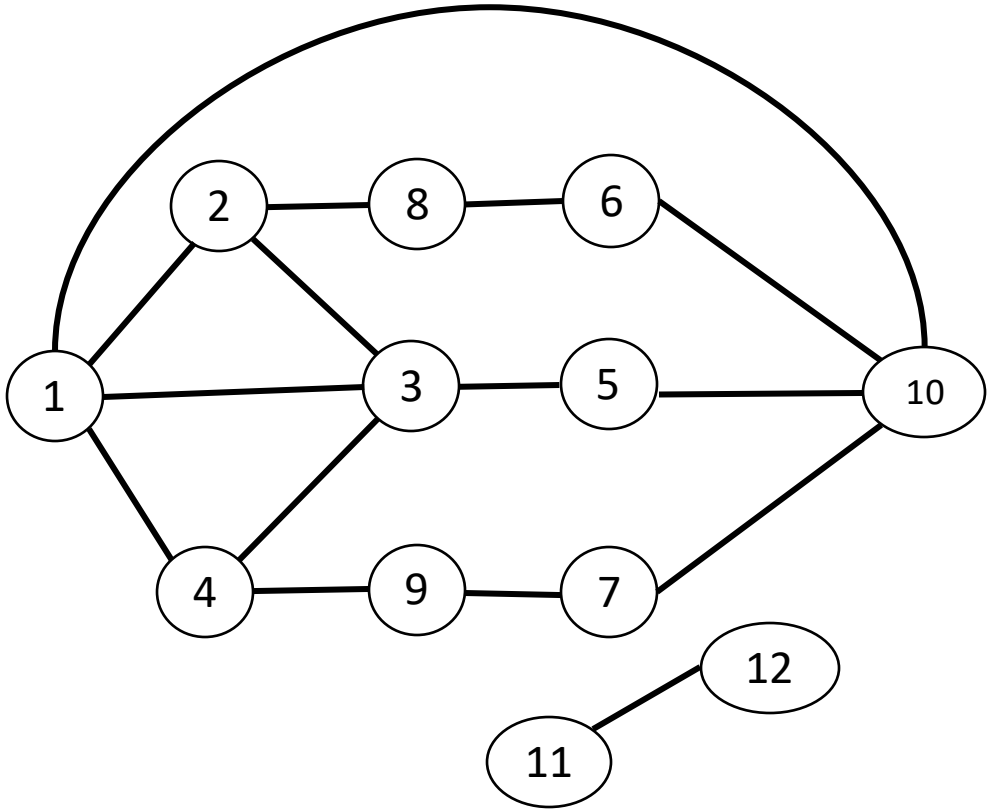
обратная дуга (v,w) ведет из «серой» вершины в «серую» (приводят к контуру)

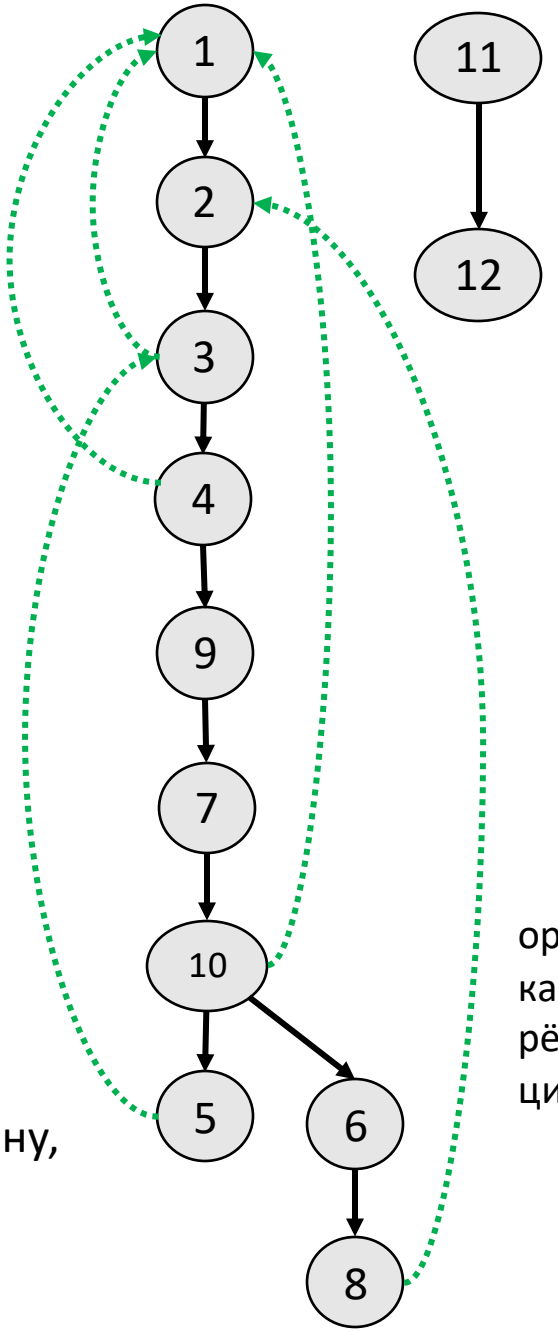
прямая дуга (v,w) ведёт из «серой» вершины в «черную» и ord[v]<ord[w]

поперечная дуга (v,w) ведёт из «серой» в «чёрную» и ord[v]>ord[w]

Поиск в глубину в графе

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





Ориентация рёбер при обходе в грубину:

древесное (ведёт в непосещенную вершину)

обратное (ещё не ориентировано и ведёт в вершину, которая ранее уже была посещена);

ориентированные, как обратные, рёбра приводят к циклу

Программная реализация DFS

Обход всех вершин, при котором в стеке лежит путь от стартовой вершины к текущей вершине.

Пусть `visited` -- массив меток посещения вершин, изначально заполненный `False`.

Рекурсивный DFS на списке смежности:

```
def dfs(v):
    visited[v] = True
    for u in g[v]:
        if not visited[u]:
            dfs(u)
```

Необходимо знать вершину, с которой начинать обход.

```
dfs(start)
```

Выходит, что все вершины, для которых метка в `visited` равна `True`, достижимы из вершины `start`.

Время работы: `O(n + m)`

DFS на матрице смежности будет немного отличаться:

```
def dfs(v):
    visited[v] = True

    for u in range(n):                    # <-- вот
        if a[v][u] and not visited[u]: # <-- здесь!

            dfs(u)
```

Рекурсивный DFS на списке смежности:

```
def dfs(v):
    visited[v] = True
    for u in g[v]:
        if not visited[u]:
            dfs(u)
```

Нерекурсивный DFS на списке смежности

```
def dfs(start):
    s = stack()
    s.push(start)
    while not s.empty():
        v = s.top()
        is_top = True
        while last[v] < len(g[v]):
            u = g[v][last[v]]
            last[v] += 1
            if not visited[u]:
                s.push(u)
                is_top = False
                break
        if is_top:
            s.pop()
```

visit[s]=True

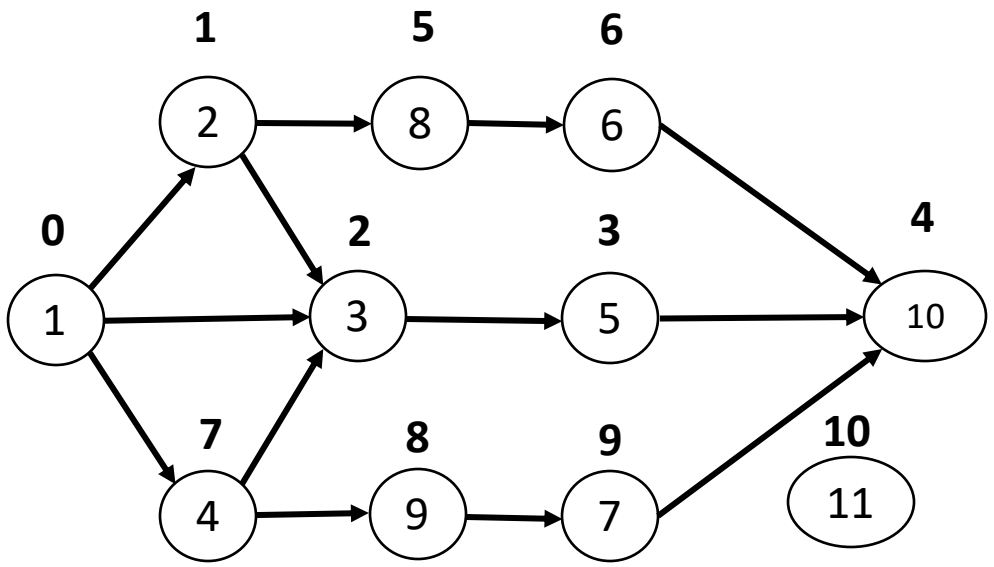
visit[u]=True

Рекурсивный DFS на списке смежности:

```
def dfs(v):
    visited[v] = True
    for u in g[v]:
        if not visited[u]:
            dfs(u)
```

Здесь **last** - массив, в котором для каждой вершины **v** хранится индекс той позиции списка смежных с ней вершин, в которой мы остановились, когда осуществляли поиск вершин смежных с **v**. Первоначально массив заполнен 0.

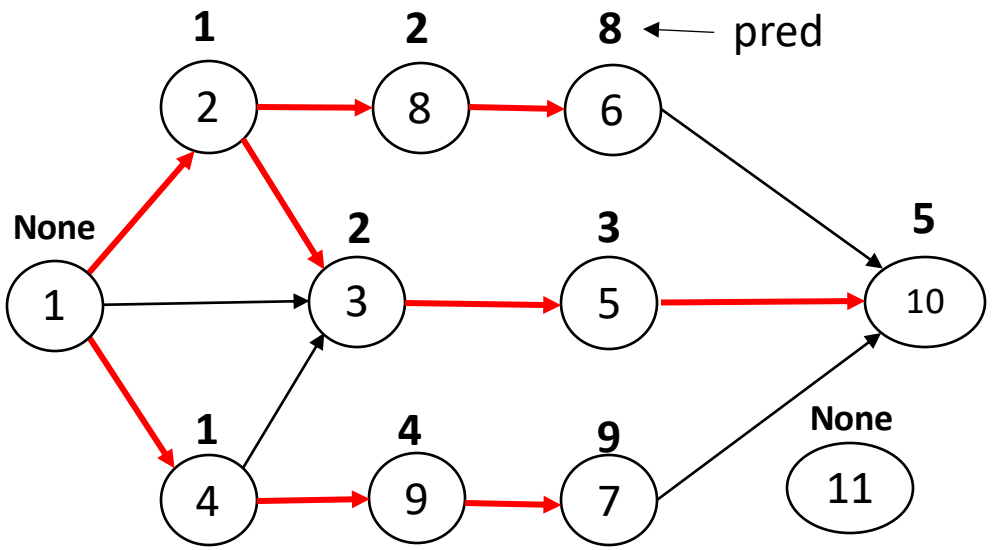
Порядок обхода вершин




```
next_idx = 0

def dfs(v):
    visited[v] = True

    ord[v] = next_idx # <-- опять
    next_idx += 1     # <-- здесь!

    for u in g[v]:
        if not visited[u]:
            dfs(u)
```

	1	2	3	4	5	6	7	8	9	10	11
Next_id[i]	0	1	2	7	3	6	9	5	8	4	10

Корневое дерево поиска в глубину




```
def dfs(v):
    visited[v] = True
    for u in g[v]:
        if not visited[u]:

            pred[u] = v # <-- здесь!

            dfs(u)


for v in range(n):
    if not visited[v]:

        pred[v] = None # <-- и здесь!

        dfs(v)
```

Вместо **None** можно использовать **-1** или **v**, если мы точно знаем, что нет петель.

Зная метки pred, можно восстановить пути из стартовой вершины во все остальные вершины.

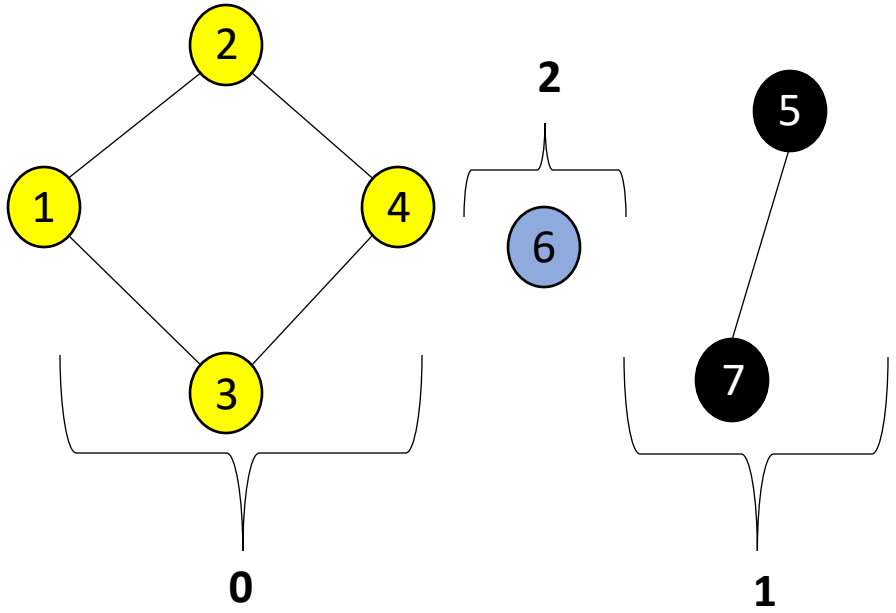
	1	2	3	4	5	6	7	8	9	10	11
pred[i]	None	1	2	1	3	8	9	2	4	5	None

Компоненты связности графа

Граф называется **связным**, если любые две его несовпадающие вершины соединены маршрутом.

```
components = 0

for v in range(n):
    if not visited[v]:
        dfs(v)
        components += 1 # <-- здесь!
```

```
def dfs(v):
    visited[v] = True

    component[v] = components # <-- здесь!

    for u in g[v]:
        if not visited[u]:
            dfs(u)
```



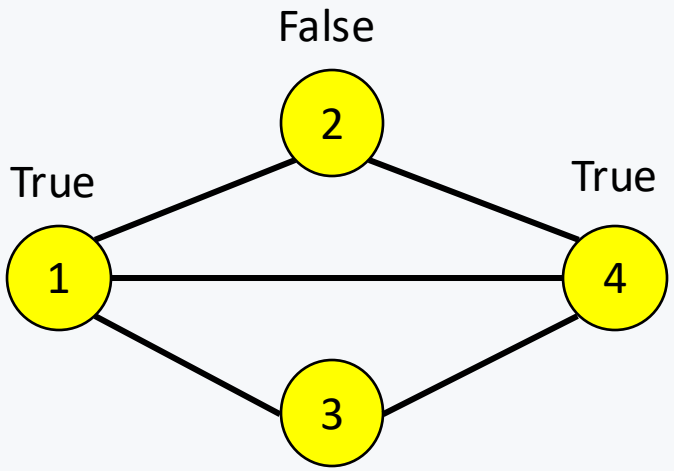
v	1	2	3	4	5	6	7
Component [v]	0	0	0	0	1	2	1

Определение двудольности графа

Свойством двудольного графа является то, что если две вершины смежны, то они принадлежат разным долям. Таким образом, присвоив метку стартовой вершине, мы определим метки всех ее смежных вершин как противоположные, и так далее. В случае обнаружения цикла мы сравним метки вершин, и если они принадлежат одной доле, то граф, очевидно, не двудолен. Если компонент связности несколько, то каждая из них должна быть двудольным графом.

Заведем массив `part`, который будет хранить метки вершин, соответсвующие долям, которым вершины принадлежат. Переменная `is_bipartite` говорит о том, является ли граф двудольным.

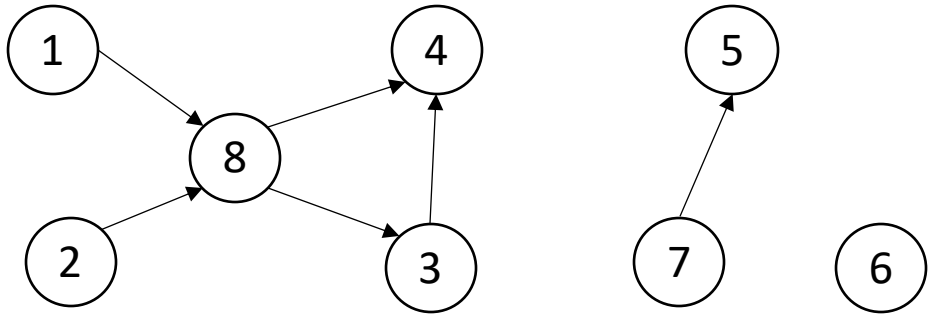
```
is_bipartite = True

def dfs(v):
    visited[v] = True

    for u in g[v]:

        if visited[u] and part[v] == part[u]: # <-- вот
            is_bipartite = False                # <-- здесь!

        if not visited[u]:

            part[u] = not part[v] # <-- и вот здесь!

            dfs(u)

for v in range(n):
    if not visited[v]:

        part[v] = True # <-- и еще здесь!

        dfs(v)
```



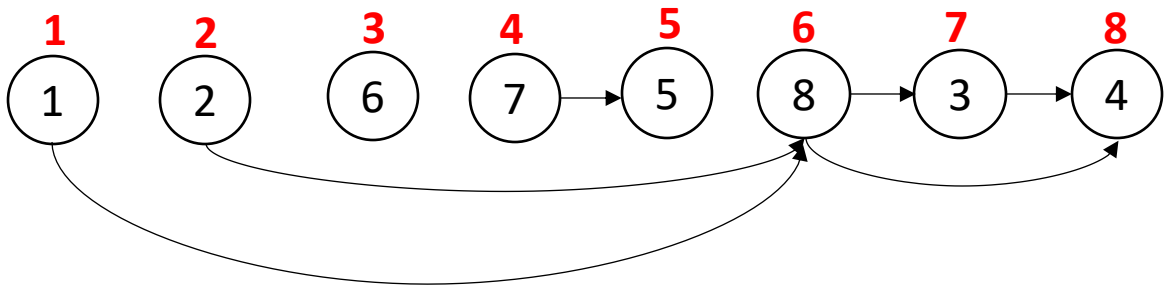
is_bipartite=False

Топологическая сортировка

Топологической сортировкой вершин ориентированного графа, который не содержит контуров, называется такая перенумерация его вершин, что у любой дуги (u, v) номер её начала u меньше, чем номер её конца v(либо наоборот).



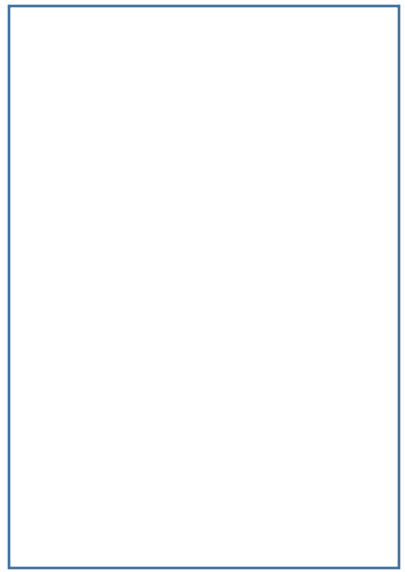
Для этого можно разместить вершины орграфа на одной линии так, чтобы все дуги исходного орграфа шли в одну сторону, например, слева направо или справа налево.
Затем присвоить новые номера вершинам, двигаясь вдоль линии слева направо.



Для выполнения топологической сортировки вершин ациклического орграфа можно использовать, алгоритмы

1) **А. Кана** (1962)
2) **Р. Тарьяна** (1976)

Артур Кан
(A. B. Kahn)



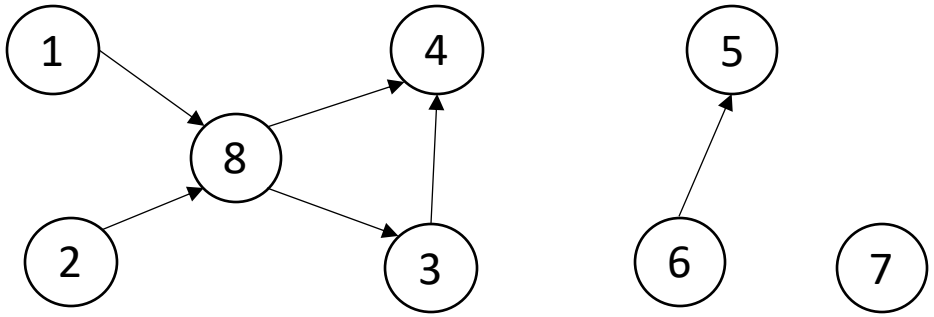
Роберт Тарьян
(*Robert Endre Tarjan),* родился в 1948 г.
американский учёный в области вычислительных систем

Алгоритм Кана

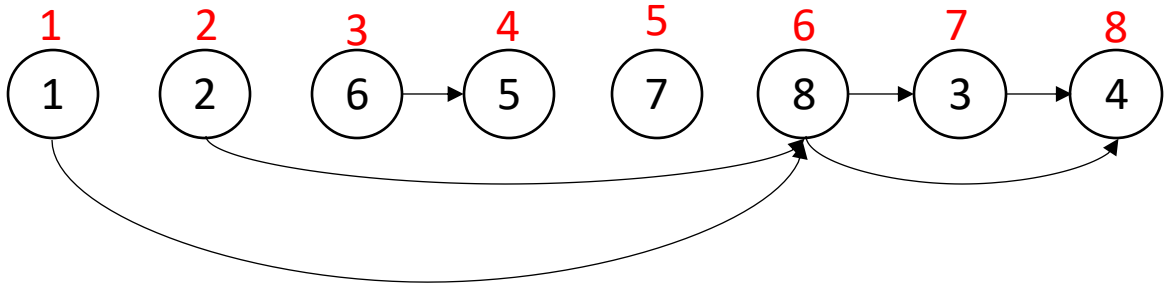
I. **Пока в графе G существуют вершины без входящих дуг**, выполнять следующие действия:
   1) найти любую вершину v без входящих дуг и присвоить ей новый номер (нумерация начинается с единицы);
   2) удалить из орграфа G вершину v вместе с выходящими из неё дугами и повторить алгоритм.

II. **Проверить, всем ли вершинам были присвоены номера**. Если не все вершины получили новые номера, то в орграфе существует контур и топологическая сортировка не может быть выполнена.

Пример



G

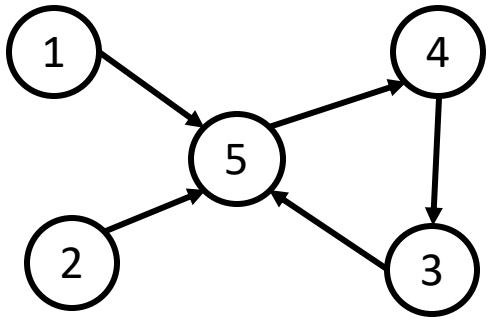


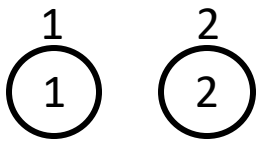
Проверить, всем ли вершинам были присвоены номера:

можно ввести счётчик числа вершин, которые были перенесены на линию:

- ✓ если нет вершин без входящих дуг и счётчик $= |V|$, то топологическая сортировка выполнена успешно;
- ✓ если счётчик $< |V|$, то в орграфе существует контур и топологическая сортировка не может быть выполнена.

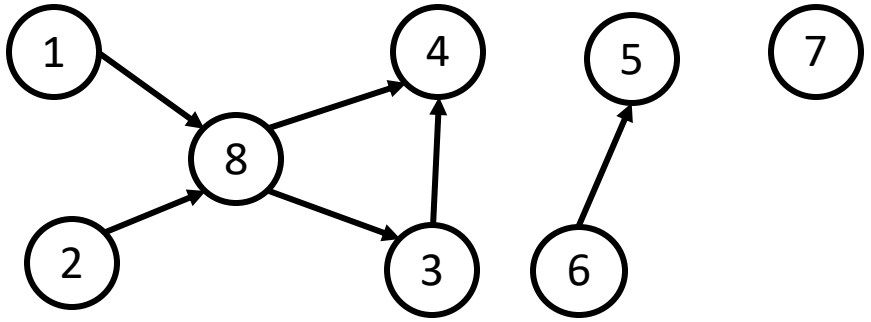
Пример



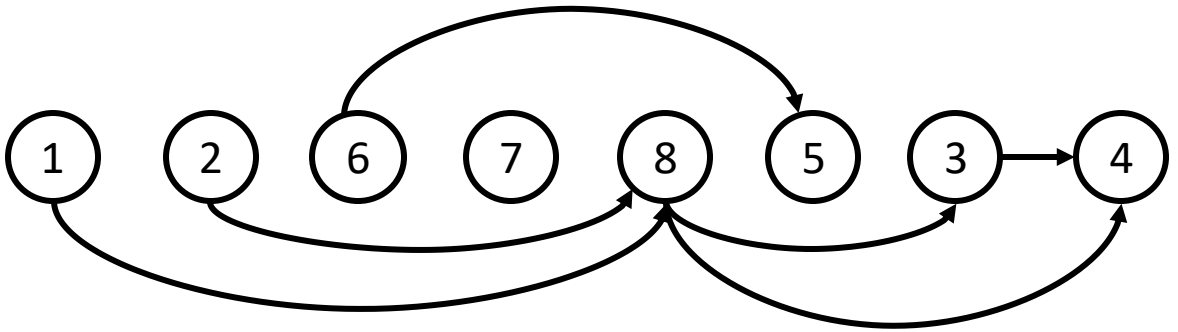


Пример

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	1
3	0	0	0	1	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	0
8	0	0	1	1	0	0	0	0
	0	**0**	**1**	**2**	**1**	**0**	**0**	**2**
								1
								0
			0					
		0	**1**					
			0					



Queue: 1 2 6 7 8 5 3 4



Время реализации алгоритма топологической сортировки Кана

- ✓ при задании орграфа матрицей смежности - $\boldsymbol{O(n^2)}$;
- ✓ при задании орграфа списками смежности - $\boldsymbol{O(n + m)}$.

Топологическая сортировка

Алгоритм Тарьяна

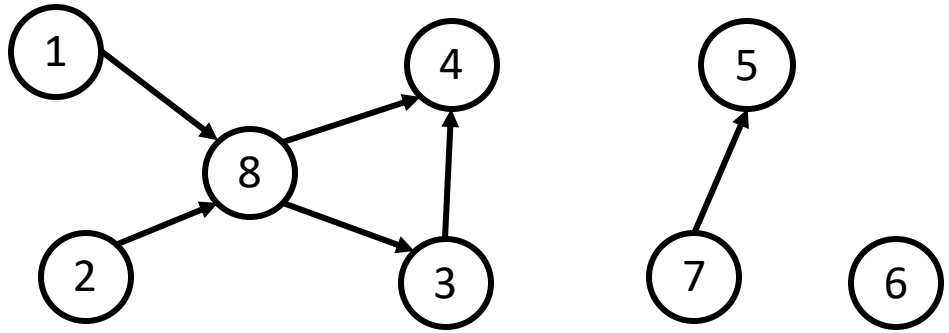
Рассмотрим проблему топологической сортировки вершин ориентированного графа. Будем говорить, что вершина `v` зависит от вершины `u`, если есть дуга `(v, u)`. Порядок вершин после такой сортировки гарантирует, что все зависимые вершины будут иметь больший индекс, чем вершины, от которых они зависят. Что же делает DFS? Он обходит все достижимые вершины из фиксированной вершины `v` перед тем, как выйти из `dfs(v)`. Это значит, что вершины, от которых зависит `v`, будут уже обработаны к моменту выхода из `dfs(v)`, и порядок выхода из функции `dfs` определяет порядок топологической сортировки.

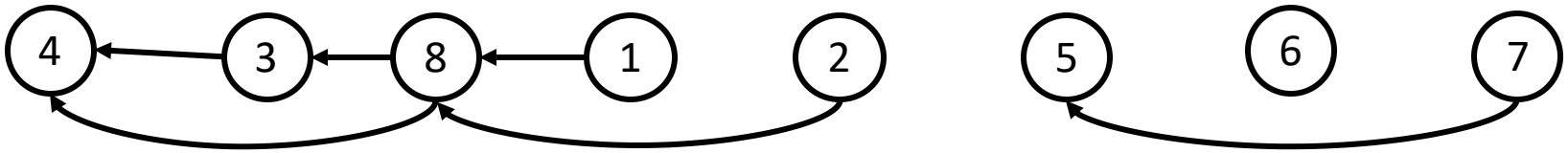
Заведем список `topsorted`, в который будем заносить вершины в порядке топологической сортировки.

```
topsorted = []

def dfs(v):
    visited[v] = True
    for u in g[v]:
        if not visited[u]:
            dfs(u)

    topsorted.append(v) # <-- здесь
```

Однако порядок топологической сортировки неопределен, если в орграфе есть ориентированные циклы. Поэтому необходимо добавить проверку на их наличие. Для этого заведем массив `in_stack`, изначально заполненный `False`, который будет хранить `True`, если вершина находится в стеке.

Алгоритм Тарьяна





topsorted: 4 3 8 1 2 5 6 7

```
topsorted = []

def dfs(v):
    visited[v] = True

    in_stack[v] = True # <-- здесь!

    for u in g[v]:

        if in_stack[u]: # <-- и еще
            panic!      # <-- здесь!

        if not visited[u]:
            dfs(u)

    topsorted.append(v)

    in_stack[v] = False # <-- и вот здесь!
```

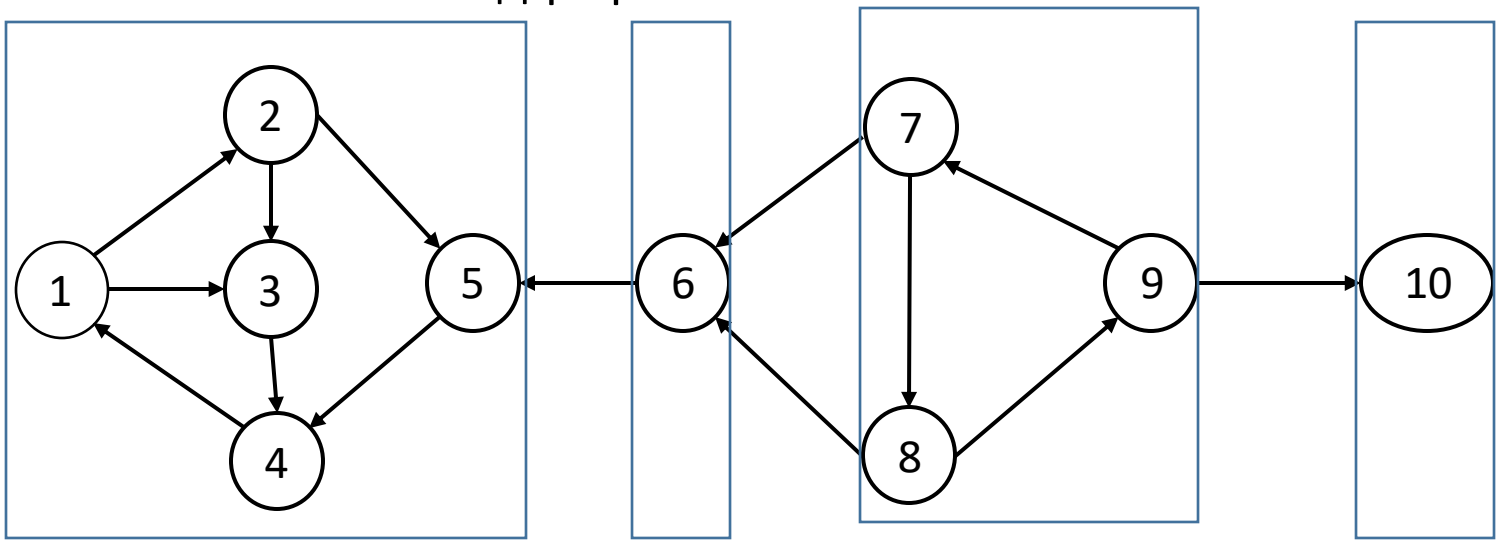
Обработка исключительной ситуации может различаться в зависимости от реализации.

Список `topsorted` можно перевернуть, чтобы дуги были ориентированы слева направо.

Выделение сильно-связных компонент орграфа

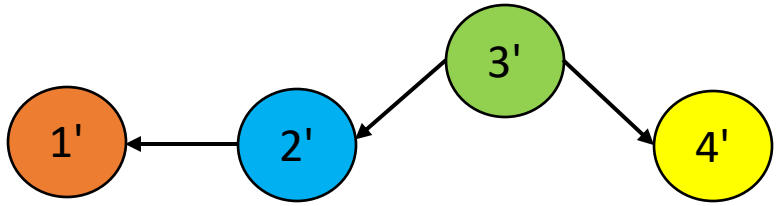
Орграф называется **сильносвязным**, если любые его две вершины достижимы друг из друга (считается, что вершина достижима сама из себя).

Сильносвязной компонентой орграфа называется любой его максимальный по включению сильносвязный подграф.



Орграф не является сильно связным.

Орграф содержит 4 сильносвязные компоненты.



Конденсация графа:
все сильносвязные компоненты свёрнуты в одну вершину.

Алгоритм **Косарайю - Шарира**
выделения сильно-связных компонент орграфа (основанный на DFS)

1978 год Самбрасива Рао Косарайю/Косараджу (Kosaraju)– американский учёный индийского происхождения, профессор информатики;
1979 год Миха Шарир (Micha Sharir) , родился в 1950 г., Израиль, профессор, специалист по вычислительной и комбинаторной геометрии;

С. Рао Косарайю

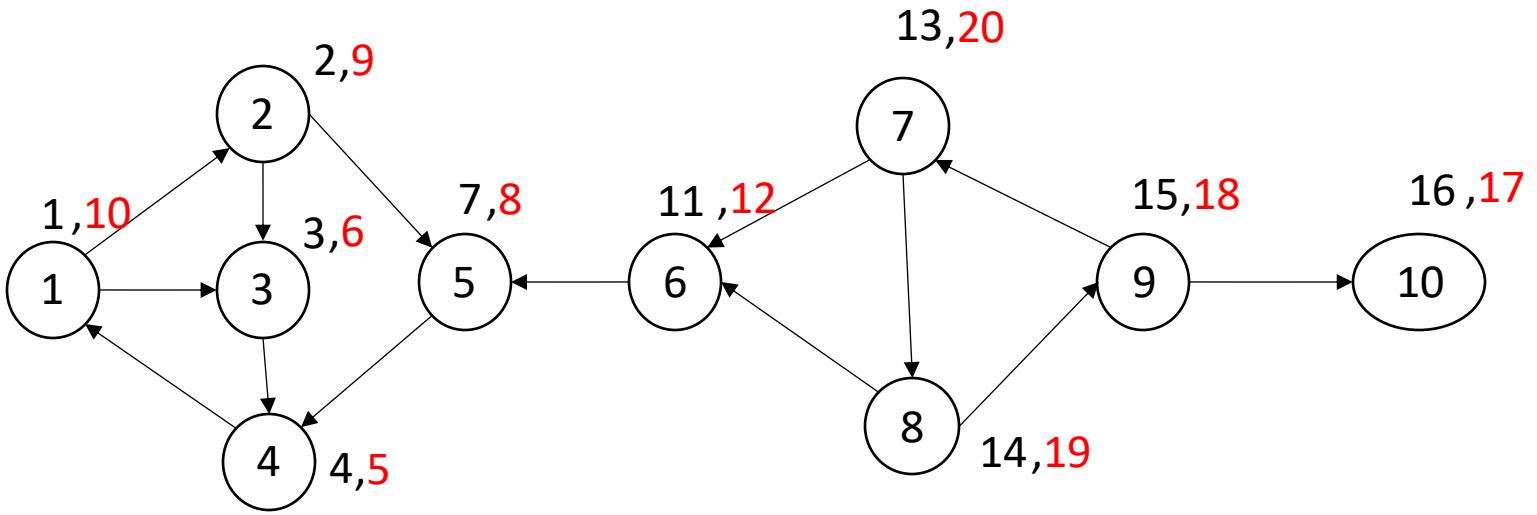
Micha Sharir

Алгоритм **Косарайю - Шарира**

Этап 1.

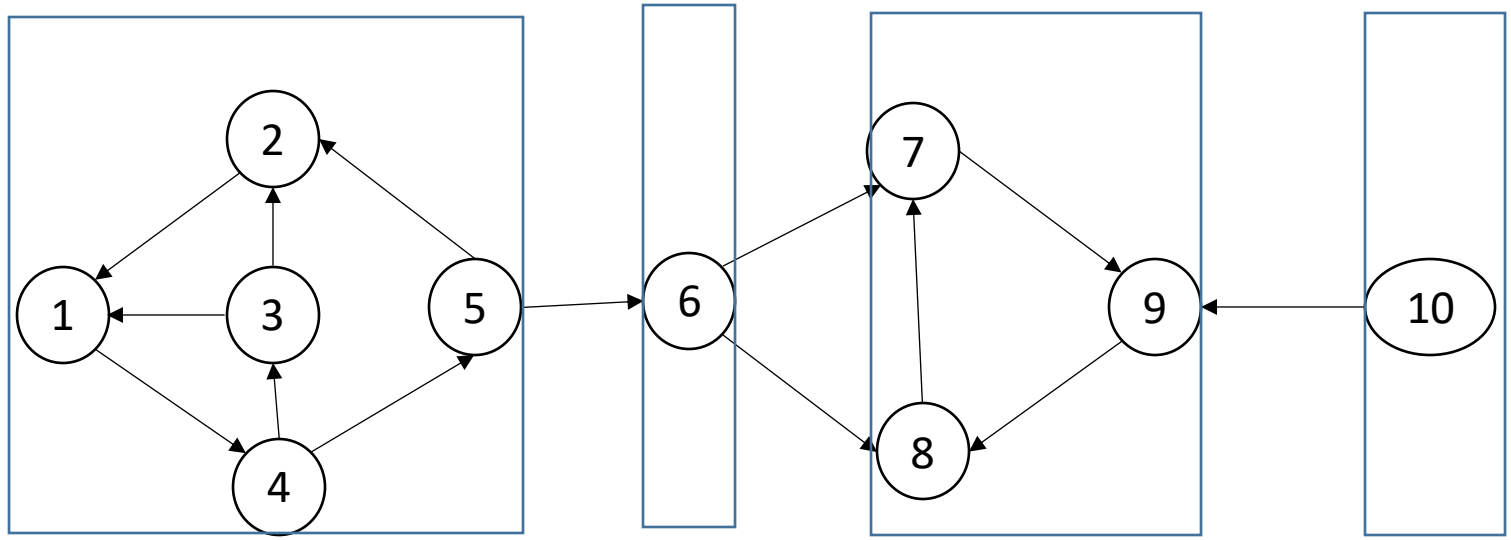
DFS. Вершинам присваивается две метки (нумерация у обеих меток – общая):

- ✓ **первая метка** присваивается, когда первый раз заходим в вершину (вершина становится серой);
- ✓ **вторая метка** присваивается, когда поиск в глубину из этой вершины завершён (осуществляется возврат из вершины и она становится чёрной).



Этап 2.

- ✓ заменить каждую дугу орграфа на противоположно направленную;
- ✓ выполнить **DFS**, начиная с вершины с самой большой второй меткой: вершины, которые при этом будут посещены, принадлежат одной сильно-связной компоненте.



Обоснование корректности алгоритма Косарайю-Шарира

Рассмотрим **конденсацию** графа G, то есть граф, в котором все сильно связные компоненты свернуты в одну вершину. Очевидно, что этот граф ацикличен, а значит, его можно отсортировать.

Если запишем вершины в порядке выхода из функции dfs, то самая последняя вершина в этом списке будет принадлежать "корню" конденсированного графа.

Воспользуемся фактом, что если транспонировать сильно связную компоненту, то мы все равно получим сильно связную компоненту. Таким образом, если транспонировать граф и запустить DFS из последней вершины, то мы посетим только вершины внутри одной сильносвязной компоненты. Соответственно, следующей непосещенной вершиной будет "корень" конденсированного графа без уже посещенной компоненты, и так далее.

Таким образом, задача решается двумя обходами DFS.

Программная реализация алгоритма Косарайю - Шарира

Пусть rg -- транспонированный орграф g, то есть орграф, в котором все дуги развернуты в обратную сторону.

```
ordered = []
components = 0

def dfs1(v):
    visited[v] = True
    for u in g[v]:
        if not visited[u]:
            dfs1(u)

    ordered.append(v) # <-- здесь!

def dfs2(v):
    visited[v] = True

    component[v] = components # <-- и еще
    for u in rg[v]:           # <-- здесь!

        if not visited[u]:
            dfs2(u)

def strongly_connected_components():
    for v in range(n):
        if not visited[v]:
            dfs1(v)

    for v in range(n):
        visited[v] = False

    for v in reversed(ordered):
        if not visited[v]:
            dfs2(v)
            components += 1
```

Время работы алгоритма Косарайю-Шарира:

O(n+m), если орграф задан списками смежности

$O(n^2)$, если орграф задан матрицей смежности.

DFS можно использовать для решения следующих задач

1) Поиск маршрута между заданной парой вершин.
2) Определение двудольности графа (орграфа).
3) Выделения связных компонент графа.
4) Выделение сильно-связных компонент ориентированного графа (алгоритм Косарайю-Шарира).
5) Топологическая сортировка вершин бесконтурного ориентированного графа (алгоритм Тарьяна).
6) Нахождение фундаментального множества циклов графа.
7) Нахождение контуров в орграфе.
8) Поиск точек сочленения и мостов.

и др.

Время работы $O(\boldsymbol{n} + \boldsymbol{m})$, если граф задан списками смежности.

Время работы $O(\boldsymbol{n^2})$, если граф задан матрицей смежности.

Общие задачи в iRunner для закрепления навыков

0.4 Матрица смежности
0.5. Канонический вид (по списку дуг)
0.6. Список смежности
0.7 Канонический вид (по матрице смежности)
0.8 BFS (поиск в ширину)
0.9 DFS (поиск в глубину)

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Спасибо за внимание