Лекция 5
Невзвешенные графы

Алгоритмы и структуры данных

Крымов А.Ю.

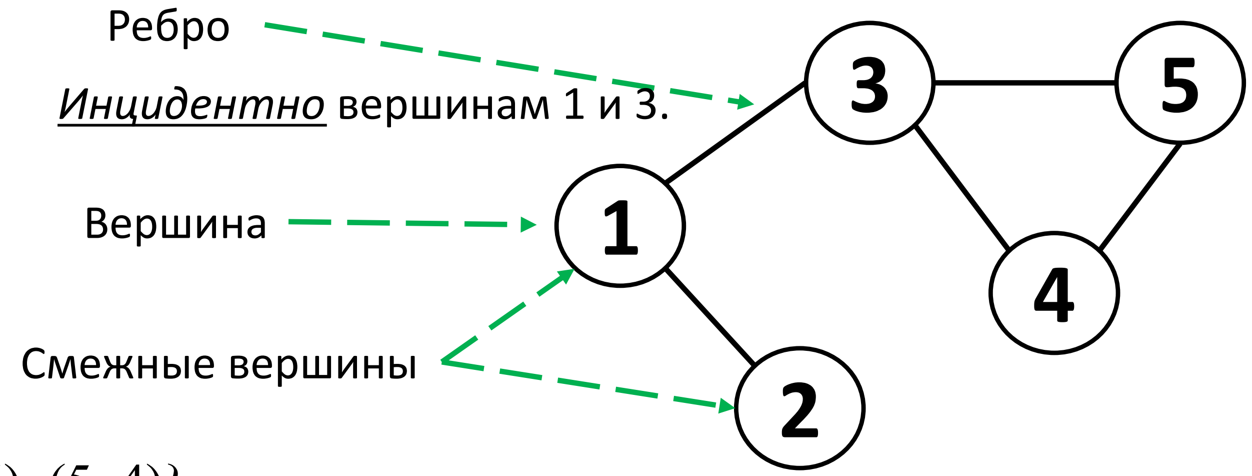
План лекции

- Терминология.
- Обход в глубину.
- Времена входа-выхода. Лемма о белых путях.
- Поиск циклов.
- Проверка связности.
- Топологическая сортировка.
- Поиск сильносвязных компонент. Алгоритм Косарайю.
- Обход в ширину.
- Поиск кратчайших путей.
- Поиск мостов
- Поиск точек сочленения
- Эйлеровы графы

Граф

Граф – это совокупность непустого множества вершин V и множества ребер E.

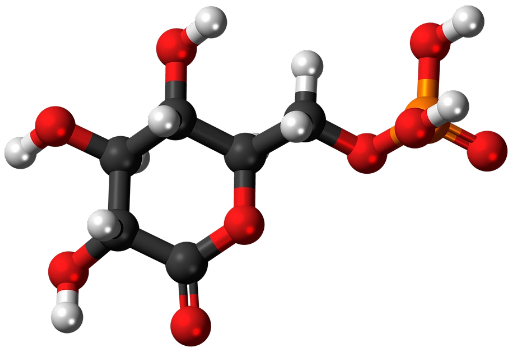
$$G = (V, E)$$
$$n = |V|, m = |E|$$
$$V = \{1, 2, \dots, n\}, E \subset \{\{v, u\}, v, u \in V\}$$

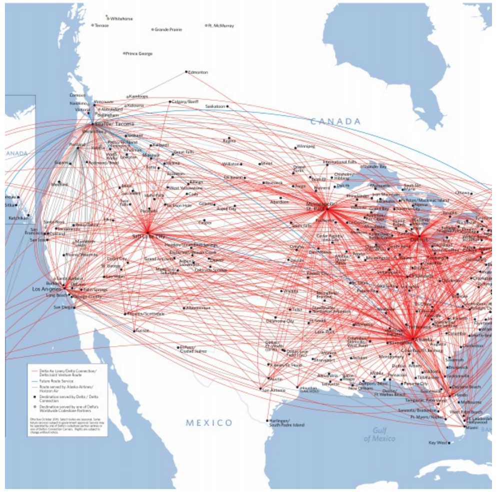


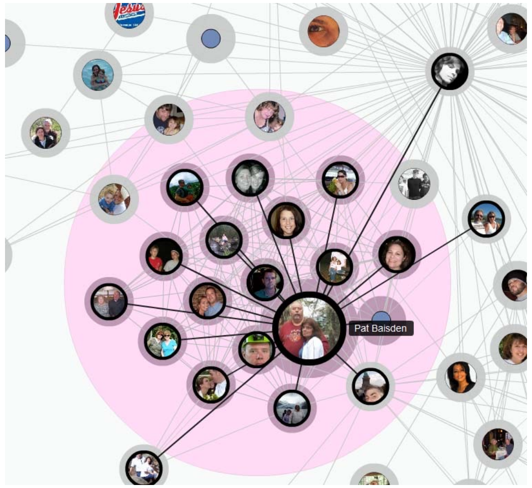
V = {1, 2, 3, 4, 5}
E = {(1, 2), (1, 3), (3, 4), (3, 5), (5, 4)}

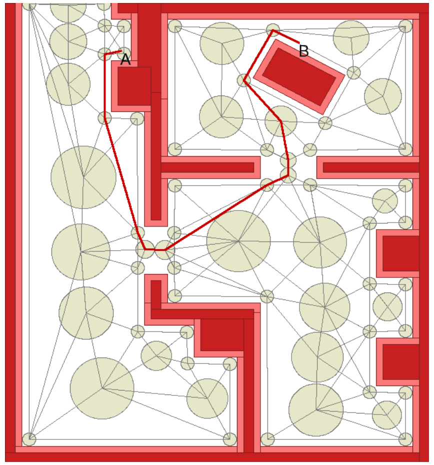
Графы в реальной жизни

- Страницы в интернете с ссылками
- Дороги
- Самолетные маршруты
- Друзья в соцсетях
- Генеалогическое древо
- Химические элементы
- Зависимости в исходниках
- Сетка перемещений в играх
- Печатные платы



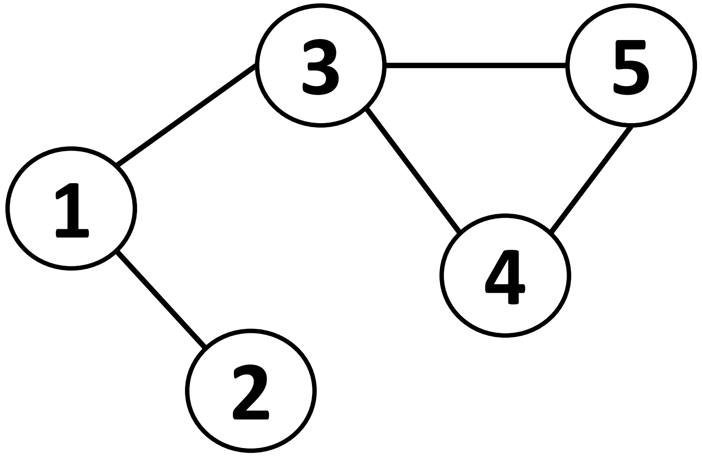






Виды графов

Неориентированные графы

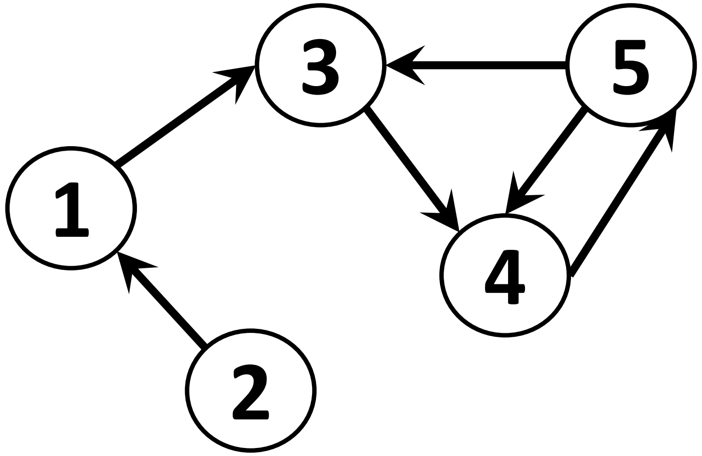


$$G = (V, E), \text{ где } E \subset \{\{v, u\}, v, u \in V\}$$

Ребра не имеют направлений
(4,5) и (5, 4) – одно и то же ребро

Ориентированные графы

(орграфы)



$$G = (V, E), \text{ где } E \subset V \times V$$

Ребра (дуги) имеют направления
(4,5) и (5, 4) – <u>разные</u> дуги

Мультиграф. Псевдограф.

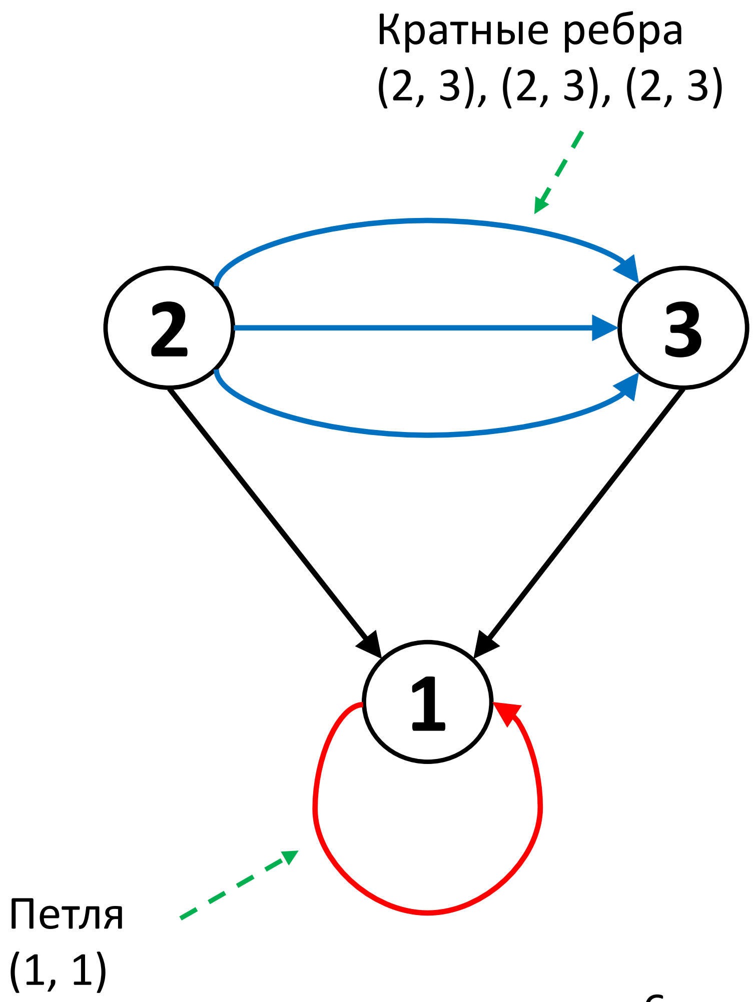
Мультиграф – граф с кратными ребрами.

Кратные рёбра (параллельные рёбра, мультирёбра) — это два и более **рёбер**, инцидентных одним и тем же двум вершинам.

Псевдограф – мультиграф с петлями.

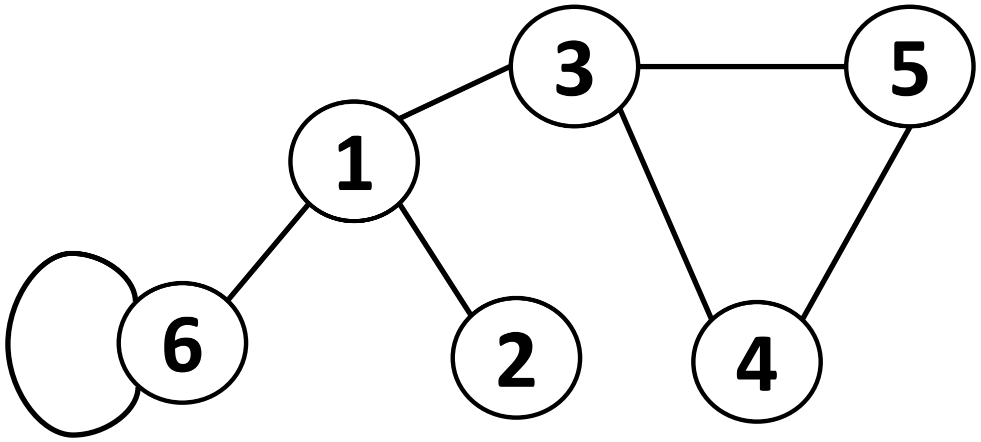
Петля – это ребро, инцидентное одной и той же вершине (v, v).

Простой граф – граф, в котором нет кратных ребер и петель.



Степень вершины

Степень вершины $\deg v$ – число ребер, инцидентных v, причем петля добавляет степень 2.



__Лемма (о рукопожатиях)__

$$\sum_{v \in V} \deg v = 2|E|.$$

Доказательство. Индукция по числу ребер.

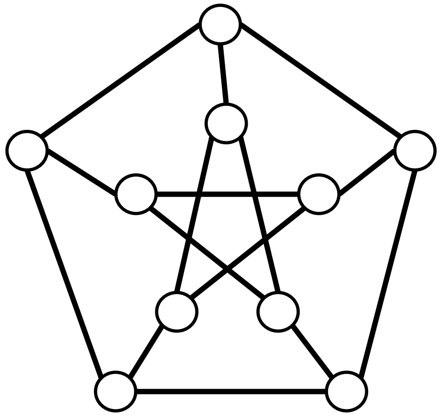
Следствие 1. Число вершин нечетной степени – четно.
Следствие 2. Число ребер в полном графе равно

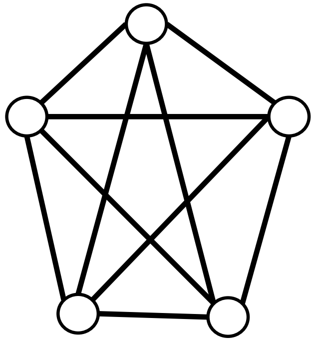
$$\frac{|V||V-1|}{2}.$$

Регулярный и полный графы

Регулярный граф – граф, в котором степени всех его вершин равны.

В таком графе $|E| = \frac{k|V|}{2}$.





Полный граф – граф, в котором каждая пара вершин смежна (все вершины соединены со всеми).

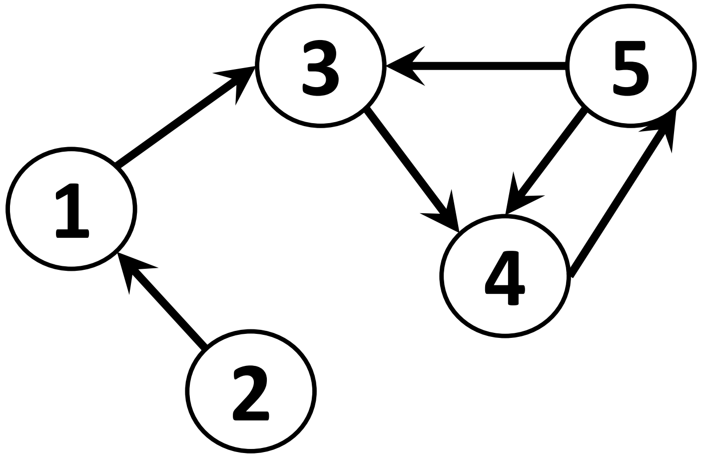
В таком графе $|\mathrm{E}| = \frac{|V||V-1|}{2}$

Представление графов в памяти

Матрица смежности – это матрица $n \times n$ элементов, в которой значение a_{ij} равно количеству рёбер из i-й вершины графа в j-ю вершину.

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

i

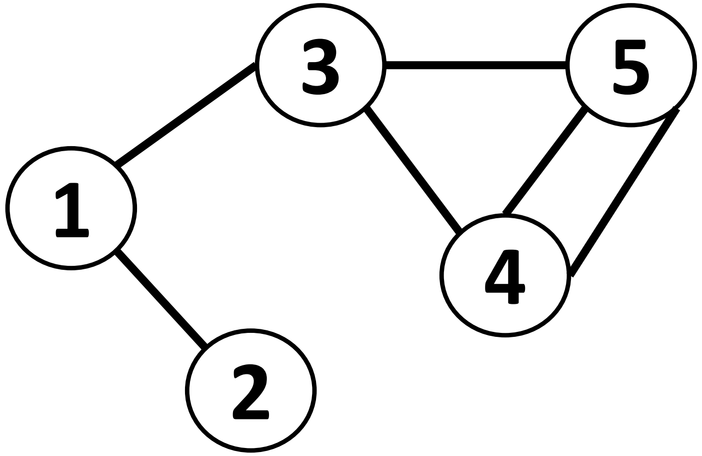


- Требует $\mathrm{O}(|V^2|)$ памяти
- Определение наличия ребра в графе за $\mathrm{O}(1)$
- Эффективна для хранения насыщенных графов ($|E| = O(|V|^2)$)

Представление графов в памяти

Матрица смежности – это матрица $n \times n$ элементов, в которой значение a_{ij} равно количеству рёбер из i-й вершины графа в j-ю вершину.

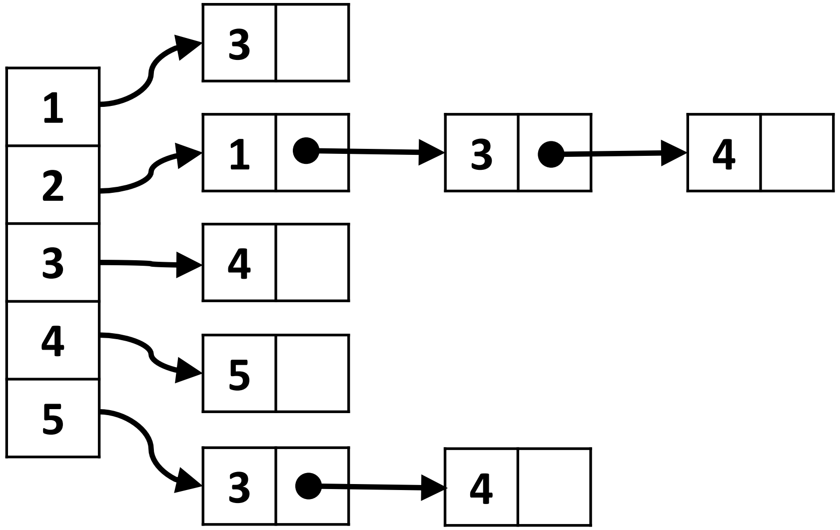
$$A = \begin{array}{c} \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 & 0 \end{pmatrix} \\ \begin{array}{ccccc} & & \mathrm{i} & & \end{array} \end{array}$$

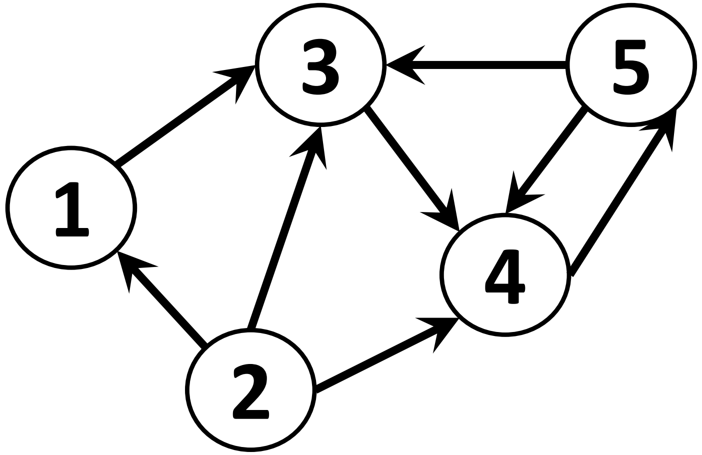


- Требует $\mathrm{O}(|V^2|)$ памяти
- Определение наличия ребра в графе за $\mathrm{O}(1)$
- Получение списка смежных вершин за $\mathrm{O}(|V|)$
- Эффективна для хранения насыщенных графов ($|E| = O(|V|^2)$)

Представление графов в памяти

Списки смежных вершин – для каждой вершины хранится список смежных с ней вершин.



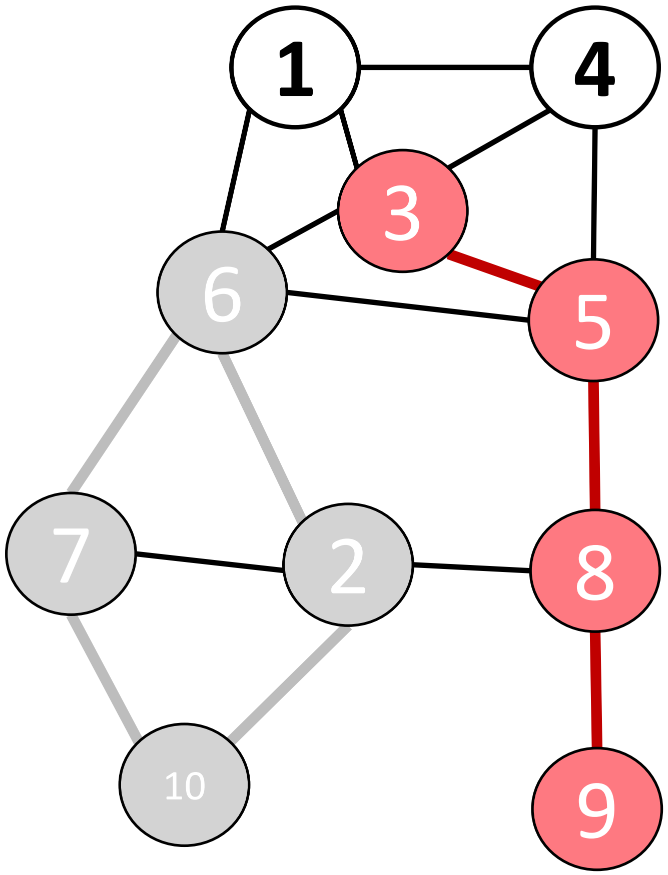


- Требует $O(|V| + |E|)$ памяти
- Определение наличия ребра в графе за $O(|E|)$
- Получение списка смежных вершин за $O(|E|)$
- Эффективна для хранения разреженных графов ($|E| = O(|V|)$)

Пути и циклы

Путь – последовательность $v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k$, где $e_i \in E$, $v_i \in V$, $e_i = (v_{i-1}, v_i)$, k – *длина* пути.

Длина пути – кол-во рёбер, задающих этот путь



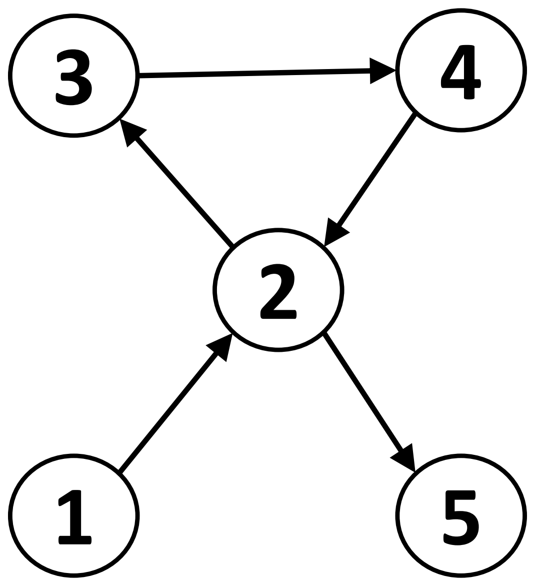
Циклический путь (цикл) в ориентированном графе – путь, в котором $v_0 = v_k$.

Циклический путь (цикл) в неориентированном графе – путь, в котором $v_0 = v_k$ и $e_i \neq e_{i+1}$, $e_1 \neq e_k$.

Пути и циклы

Простой (вершинно-простой) путь – путь, в котором каждая из вершин встречается не более одного раза.

Путь (1, 2, 3, 4, 2, 5) не является вершинно-простым.



Реберно-простой путь – путь, в котором каждое ребро встречается не более одного раза.

Путь (1, 2, 3, 4, 2, 3, 4, 2, 5) не является реберно-простым.

Связность графа

Вершины u и v в неориентированном графе ***связны***, если существует путь $u \rightsquigarrow v$.

Теорема. Связность – отношение эквивалентности.

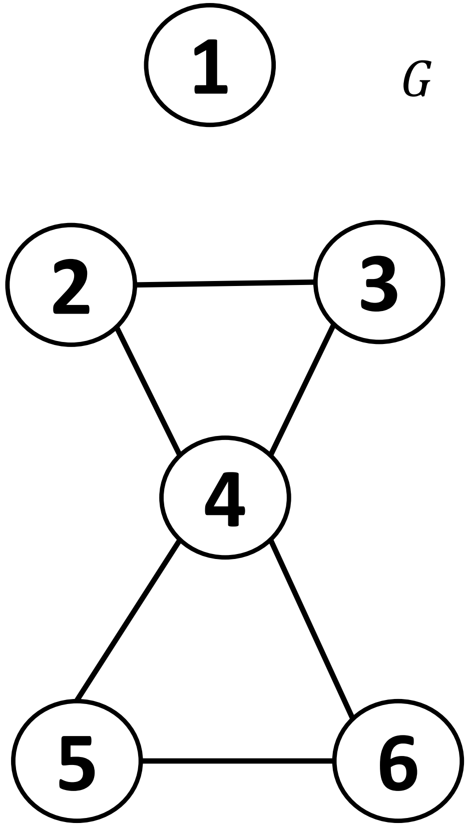
Доказательство. Надо доказать:

Рефлексивность. $\forall u \in V\ u \rightsquigarrow u$

Симметричность. $u \rightsquigarrow v \Rightarrow v \rightsquigarrow u$

Транзитивность. $v \rightsquigarrow a \land a \rightsquigarrow u \Rightarrow v \rightsquigarrow u$

Компонента связности – класс эквивалентности отношения связности.



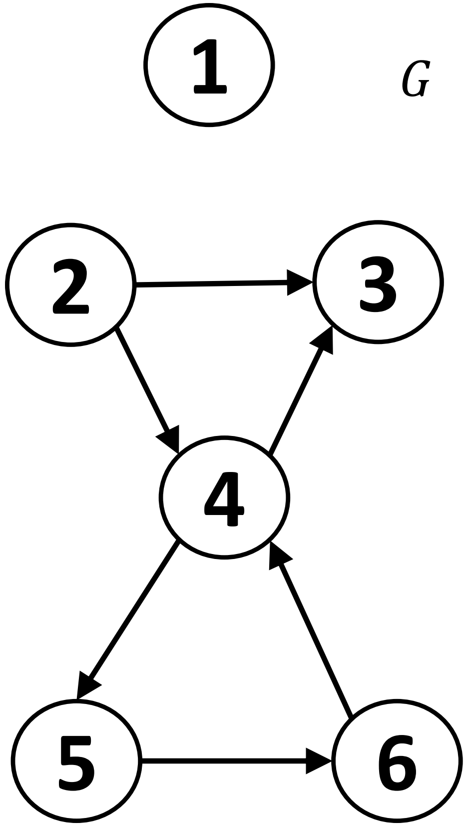
V = {1, 2, 3, 4, 5, 6}
E = {(2, 4), (2, 3), (4, 3), (4, 5), (5, 6), (6, 4)}

Сильная связность

Вершины u и v в ориентированном графе G **сильно связны**, если существуют пути $u \rightsquigarrow v$ и $v \rightsquigarrow u$.

Сильная связность – отношение эквивалентности.

Компонента сильной связности (КСС) – класс эквивалентности отношения сильной связности.



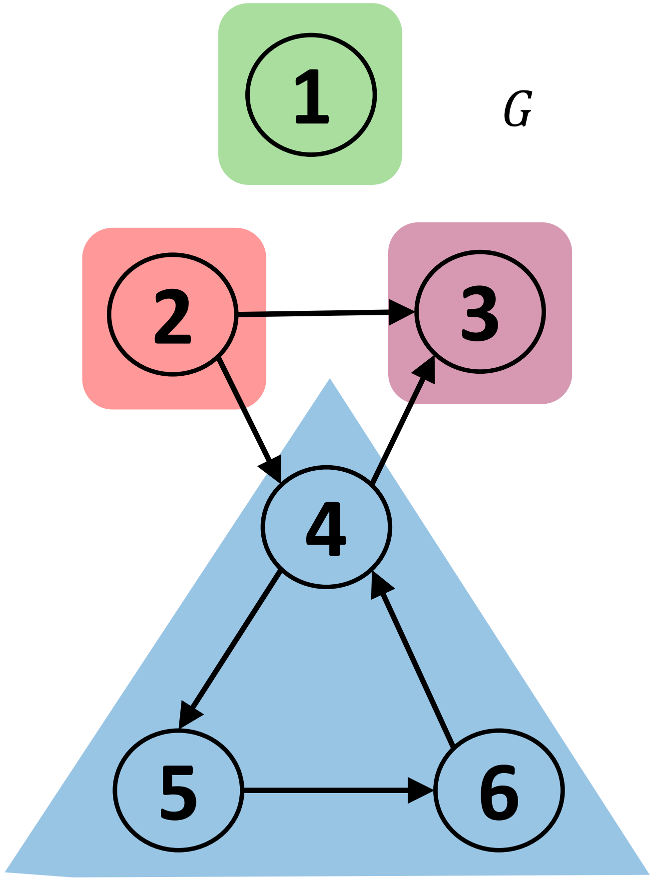
Сколько компонент сильной связности в примере?

Сильная связность

Вершины u и v в ориентированном графе G **сильно связны**, если существуют пути $u \rightsquigarrow v$ и $v \rightsquigarrow u$.

Сильная связность – отношение эквивалентности.

Компонента сильной связности (КСС) – класс эквивалентности отношения сильной связности.



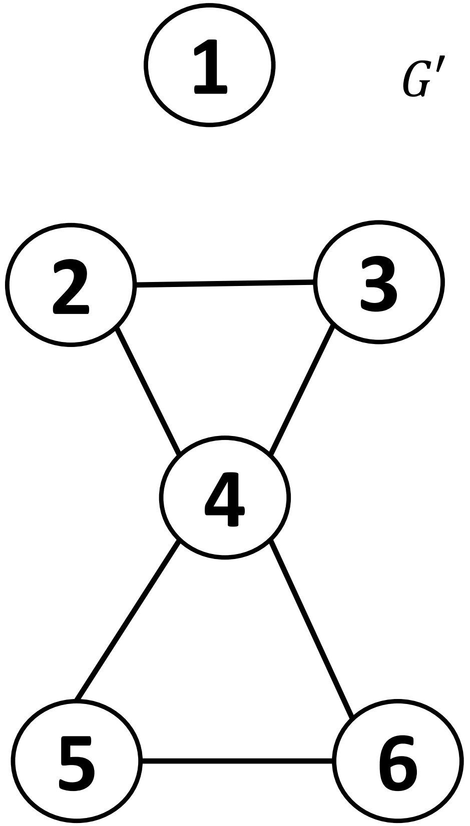
Сколько компонент сильной связности в примере?

Слабая связность

Вершины u и v в ориентированном графе G **слабо связны**, если они связны в графе G', полученном из G удалением ориентации ребер и повторяющихся ребер.

Слабая связность – отношение эквивалентности.

Компонента слабой связности – класс эквивалентности отношения слабой связности.



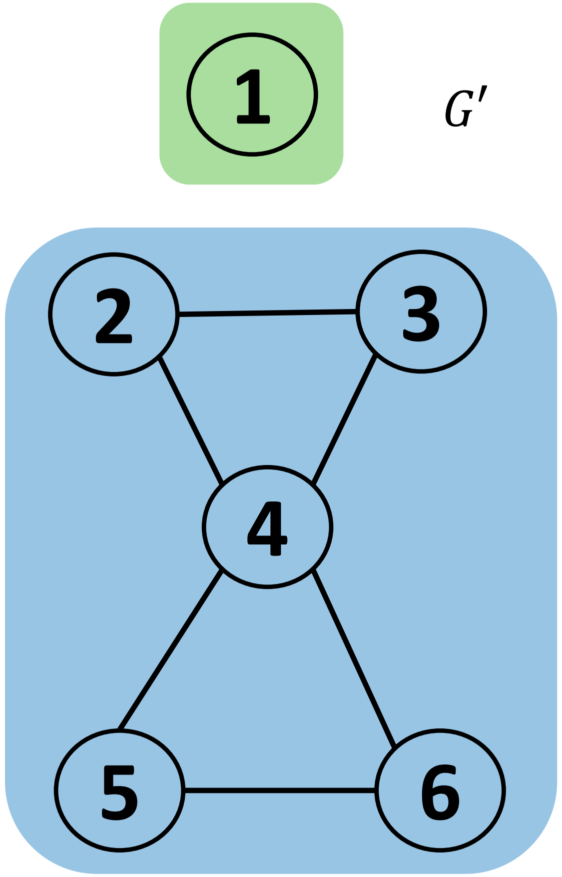
Сколько компонент слабой связности в примере?

Слабая связность

Вершины u и v в ориентированном графе G **слабо связны**, если они связны в графе G', полученном из G удалением ориентации ребер и повторяющихся ребер.

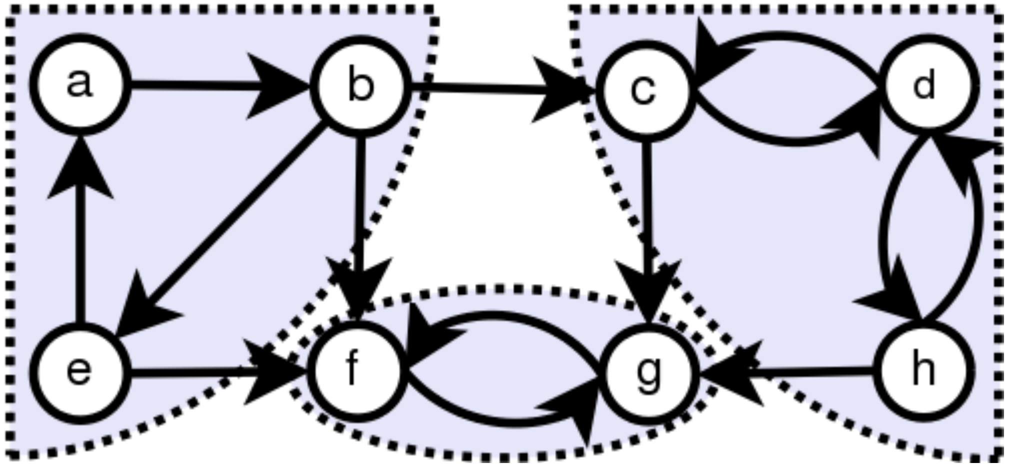
Слабая связность – отношение эквивалентности.

Компонента слабой связности – класс эквивалентности отношения слабой связности.



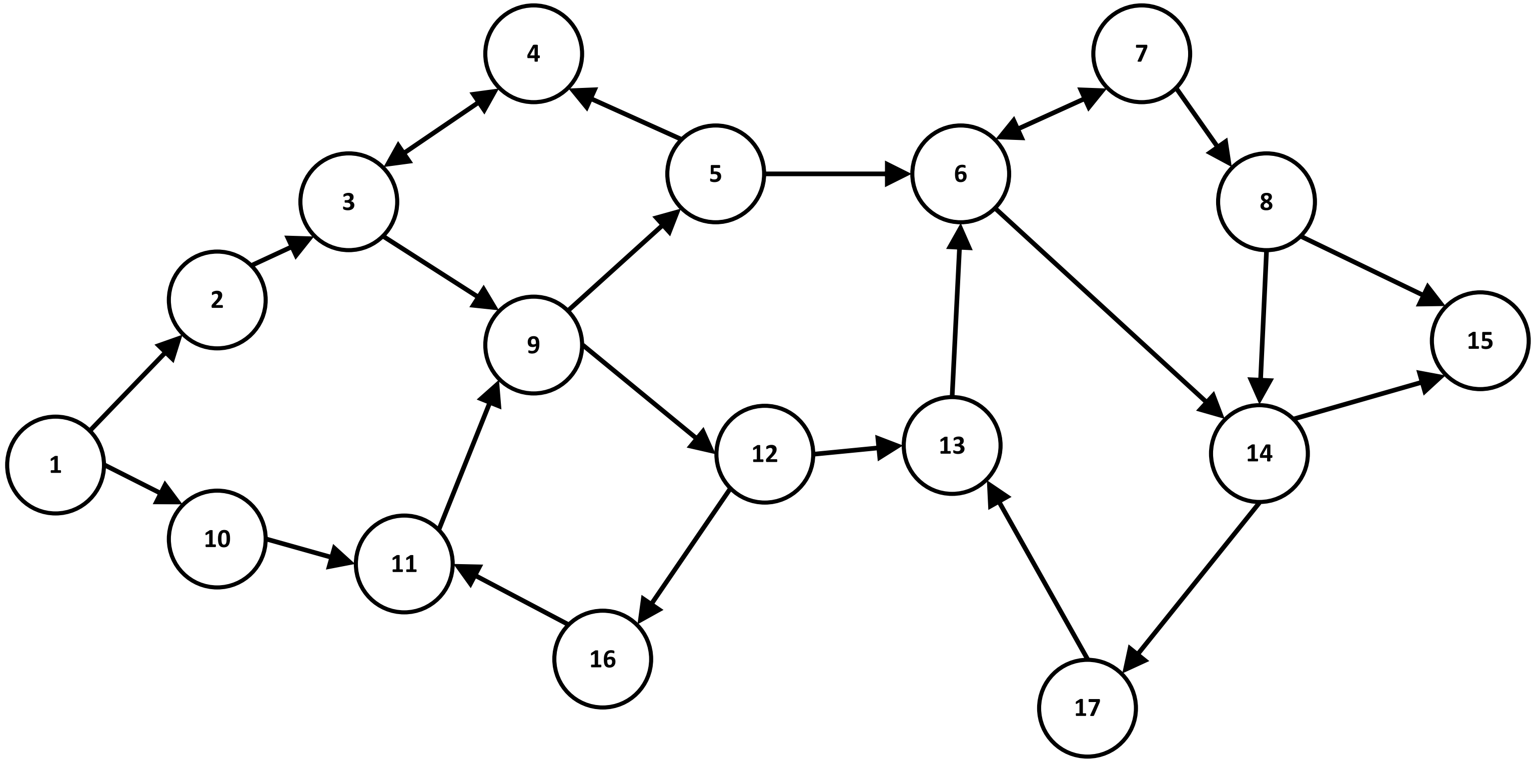
КСС

Интересно искать сильно связные компоненты графа.



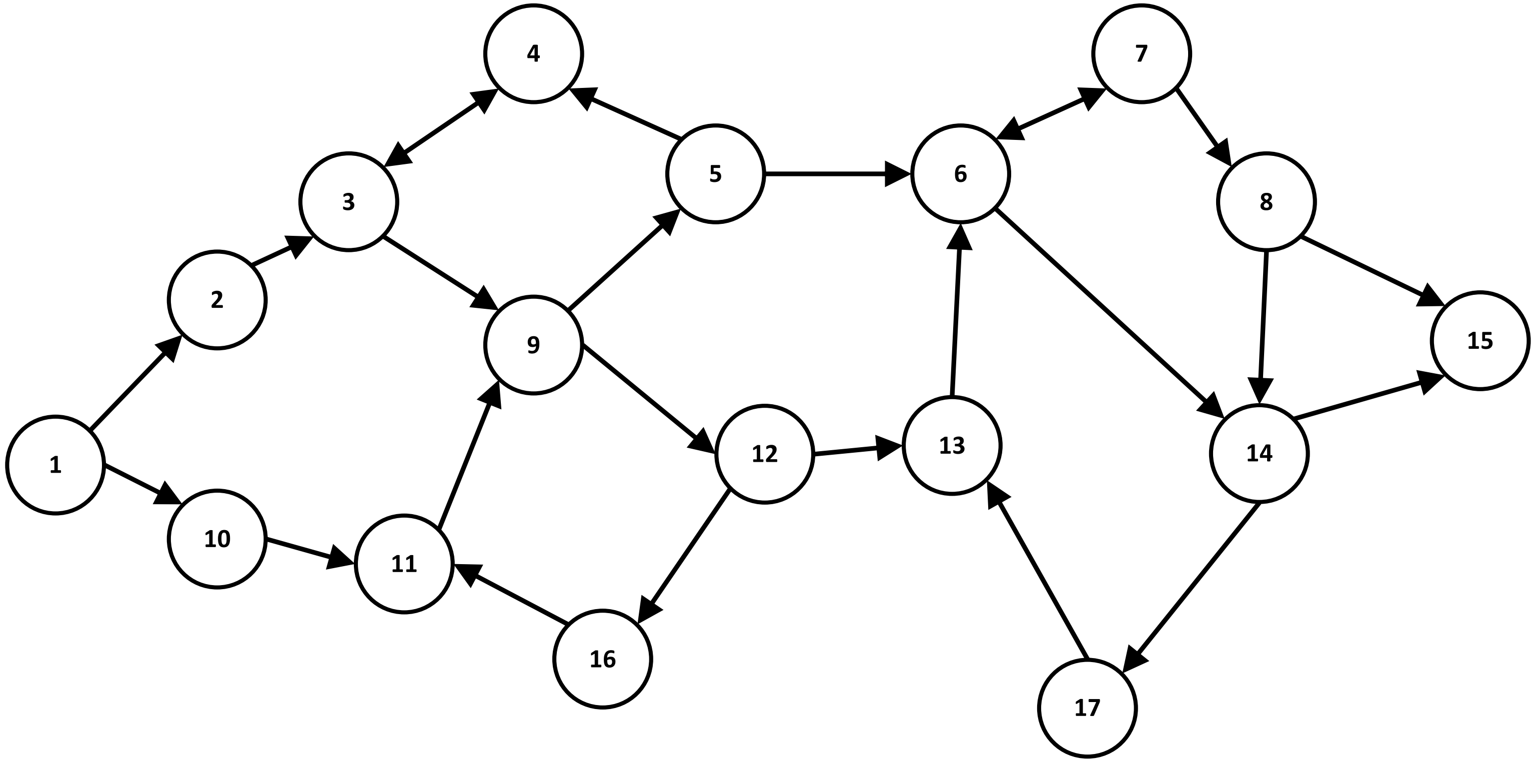
КСС

Сколько КСС в этом графе?



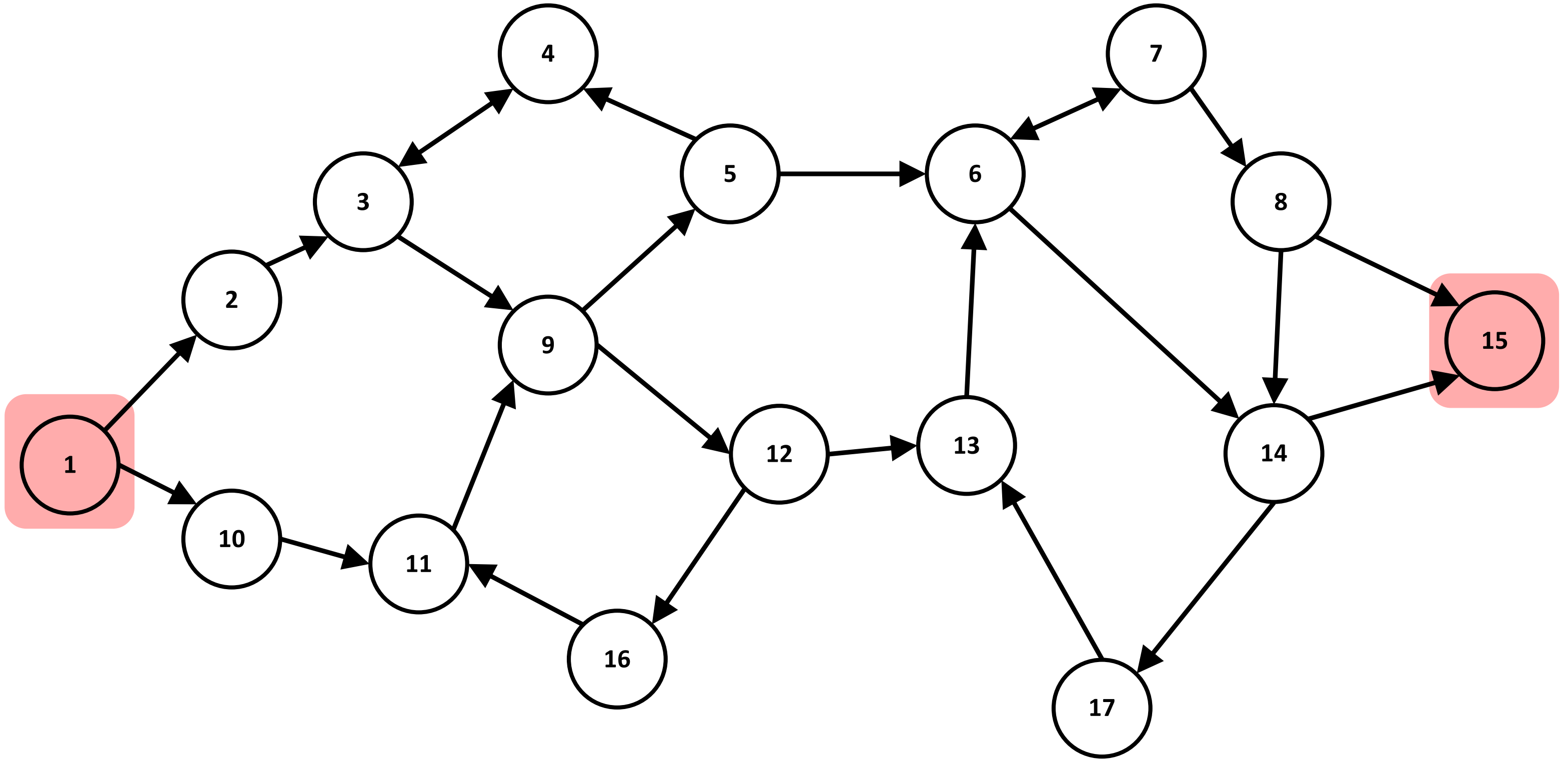
КСС

Шаг 1. Поиск стоков и истоков



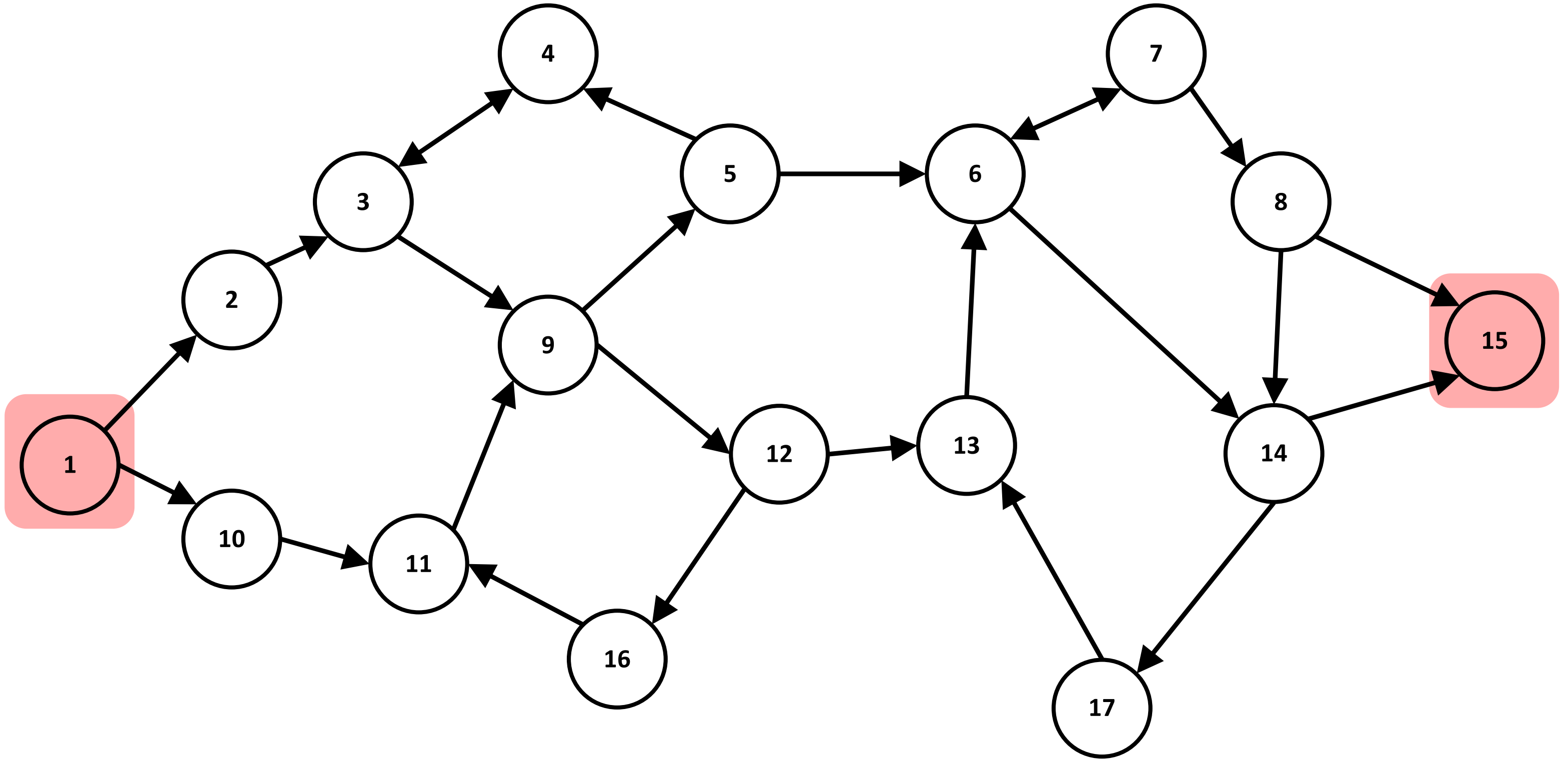
КСС

Шаг 1. Поиск стоков и истоков



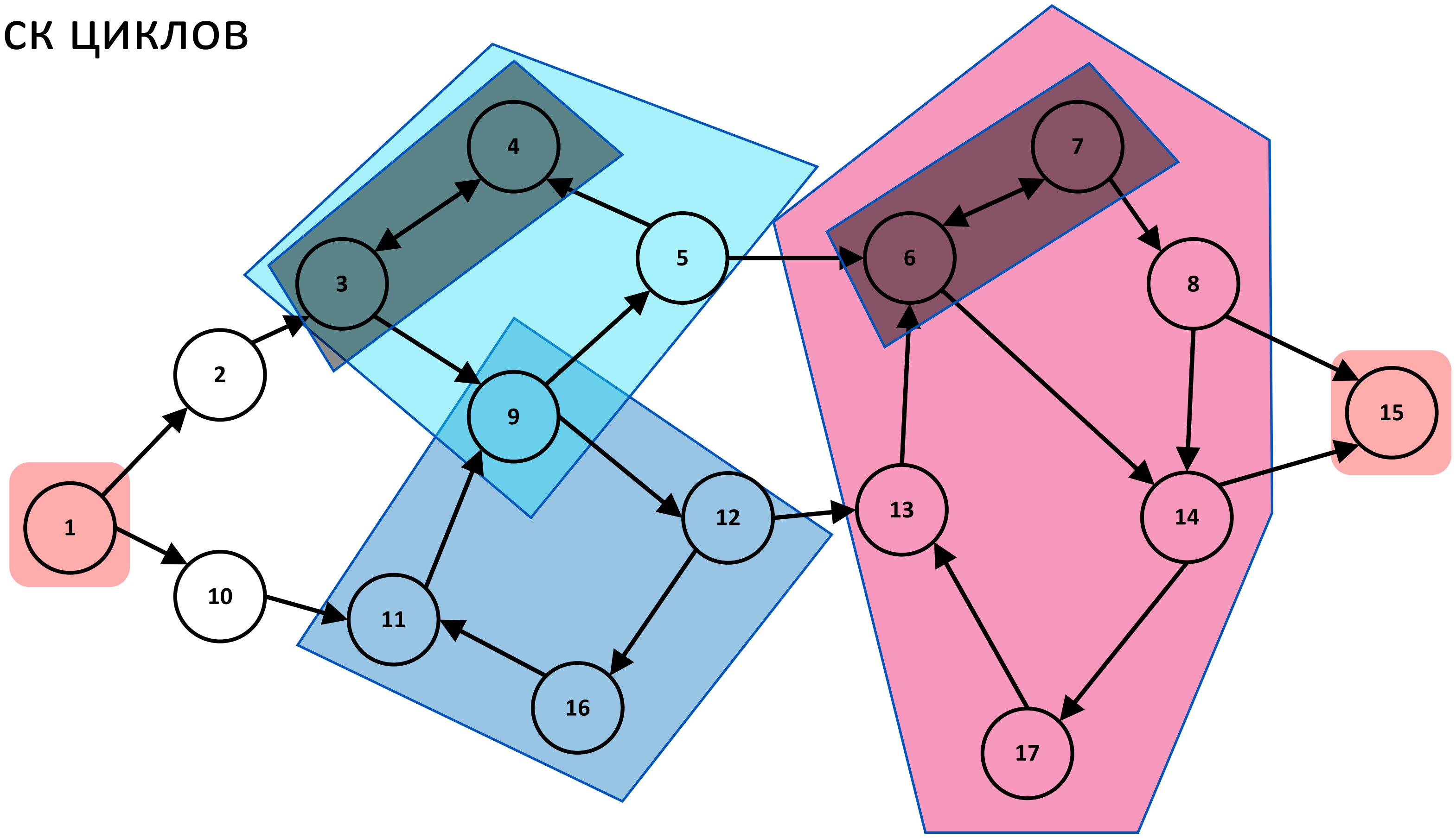
КСС

Шаг 2. Поиск циклов



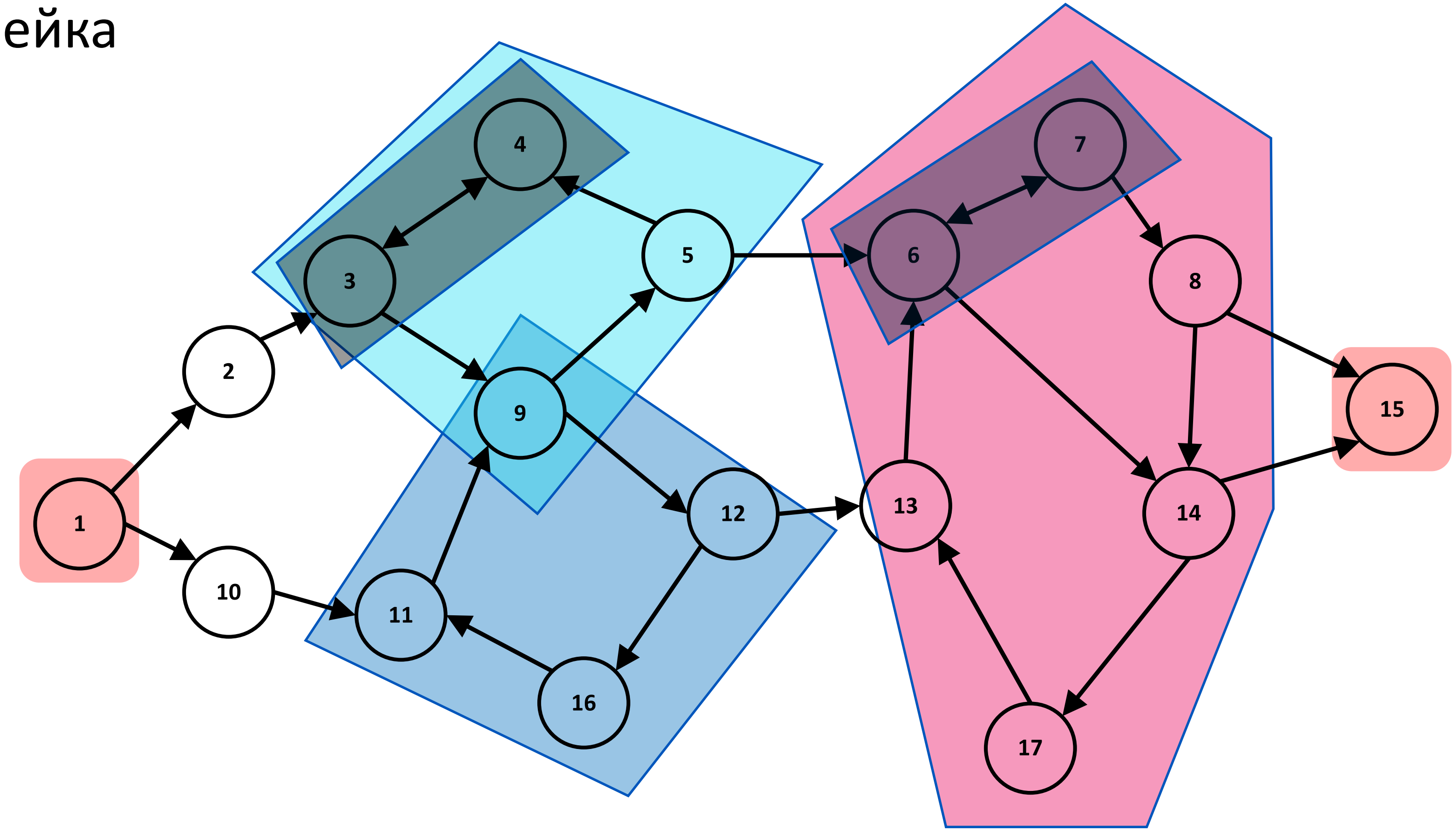
КСС

Шаг 2. Поиск циклов



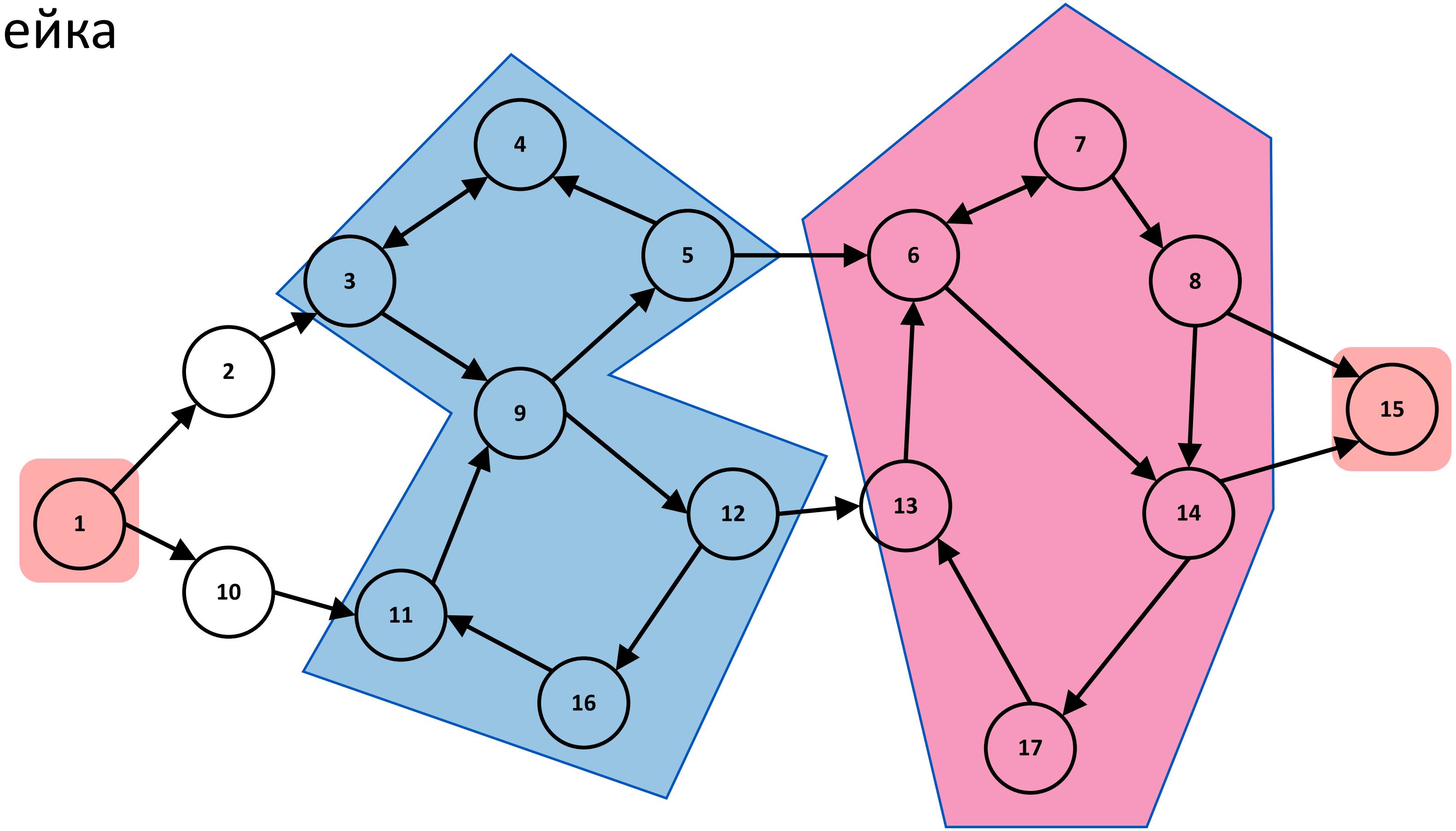
КСС

Шаг 3. Склейка



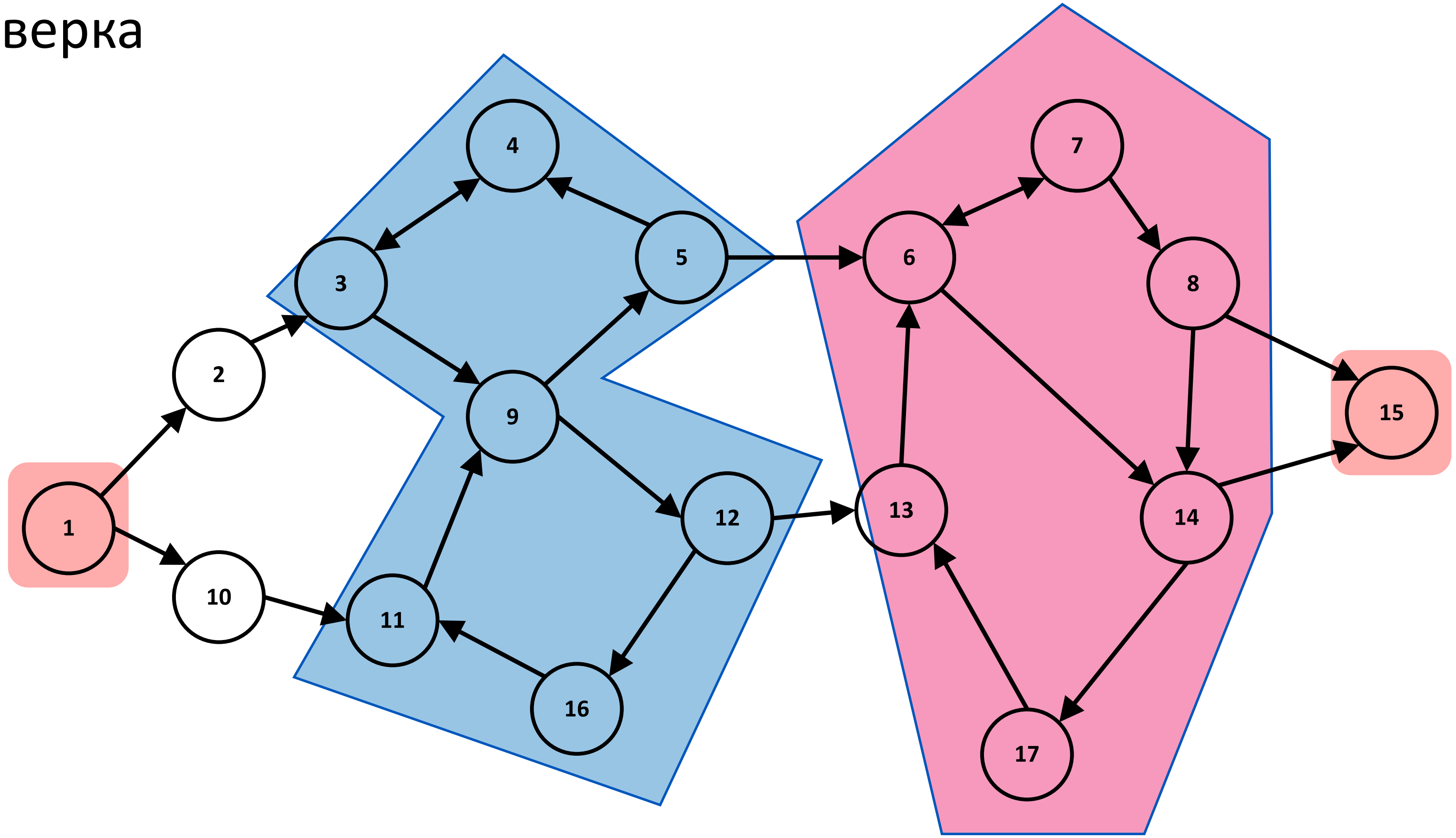
КСС

Шаг 3. Склейка



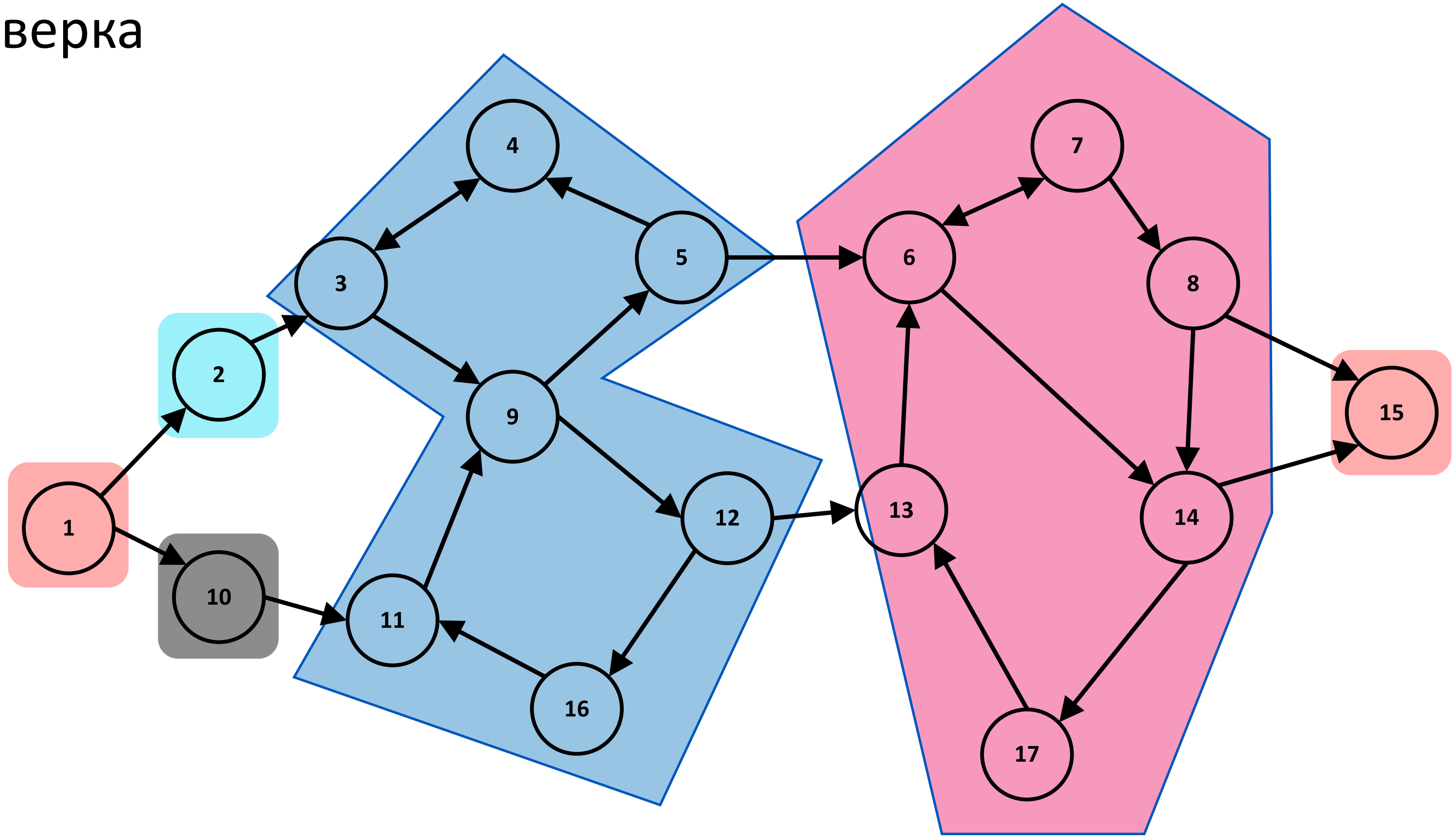
КСС

Шаг 4. Проверка



КСС

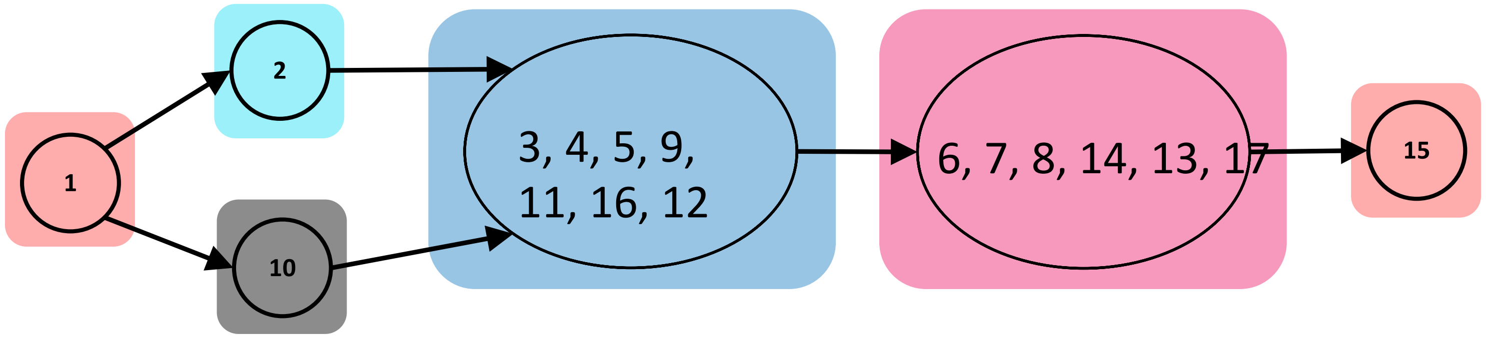
Шаг 4. Проверка



Конденсат графа

Конденсат графа – это ориентированный граф, в котором вершины соответствуют компонентам сильной связности, а дуги отражают достижимость компонент друг из друга.

Конденсат графа **ацикличен**.



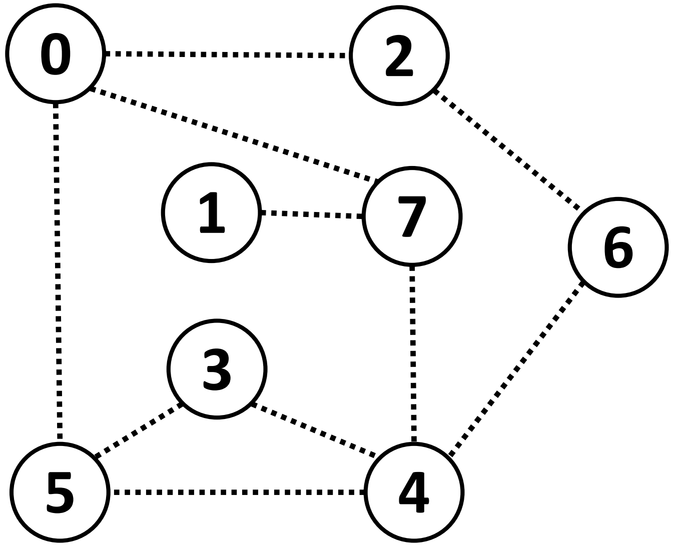
Обход в глубину

DFS – Depth First Search – обход ориентированного или неориентированного графа, при котором рекурсивно обходятся все вершины, достижимые из текущей вершины.

1) Выбираем непосещённую вершину u.
2) Запускаем dfs(u):
   - Помечаем u,
   - Запускаем dfs(v) для всех $(u, v) \in E$.
3) Повторяем 1) и 2) пока есть непосещенные вершины.

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

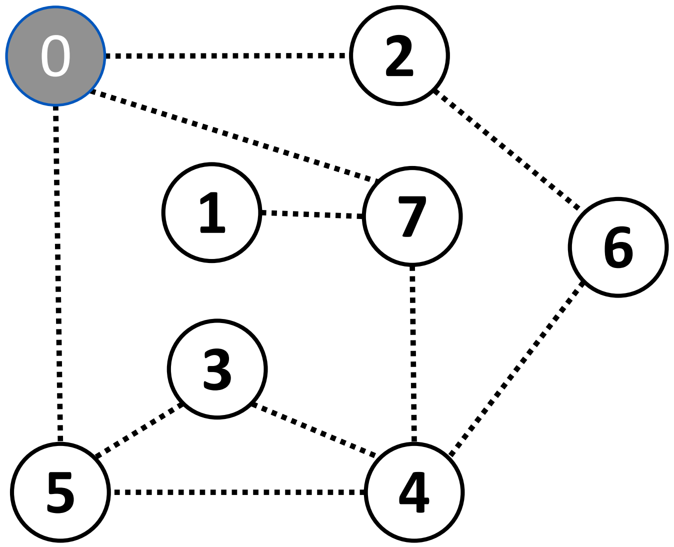
Обход в глубину



Индекс	0	1	2	3	4	5	6	7
Visited	0	0	0	0	0	0	0	0

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

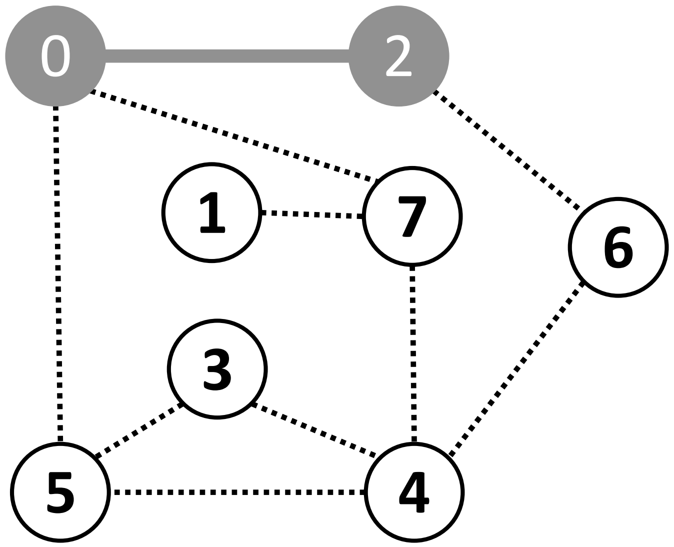


[0] Порядок посещения вершин

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	0	0	0	0	0	0

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

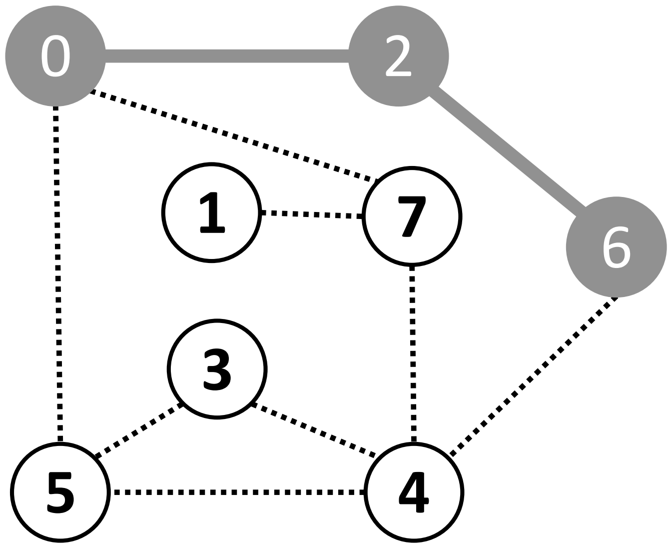


[0, 2]

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	0	0	0	0	0

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

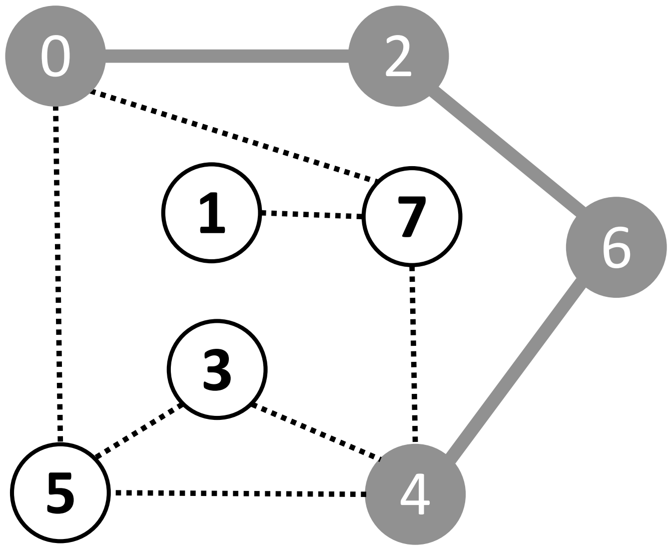


[0, 2, 6]

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	0	0	0	1	0

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

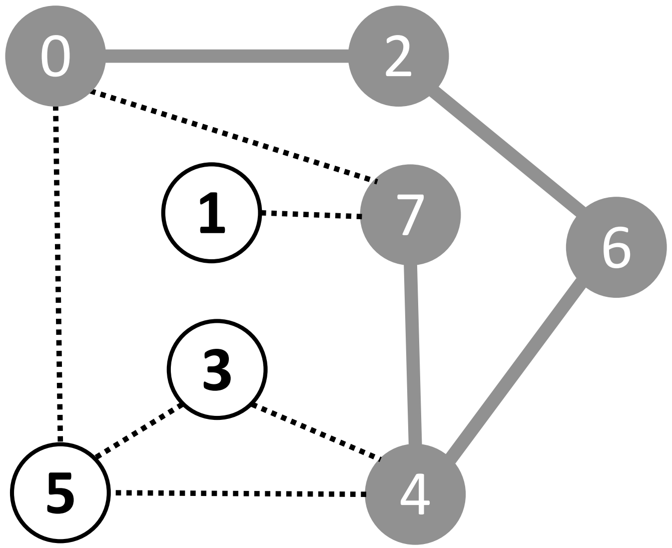


[0, 2, 6, 4]

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	0	1	0	1	0

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

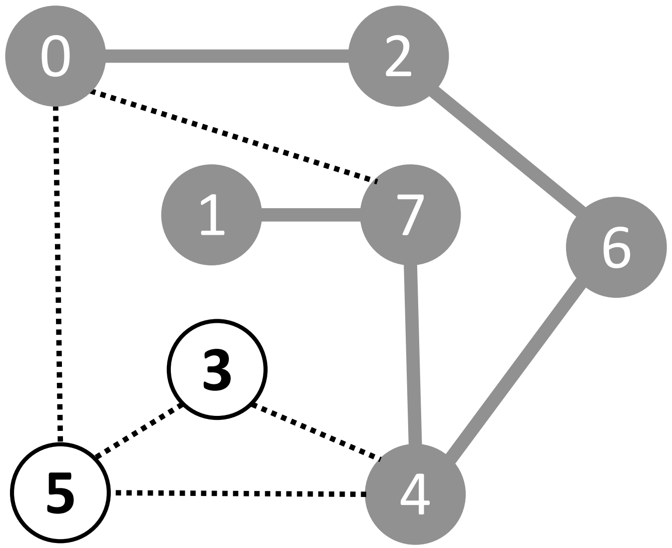


[0, 2, 6, 4, 7]

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	0	1	0	1	1

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

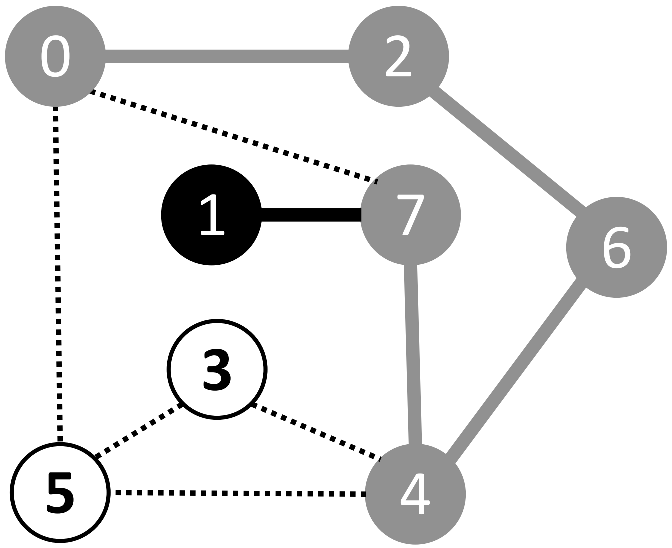


[0, 2, 6, 4, 7, 1]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	0	1	0	1	1

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

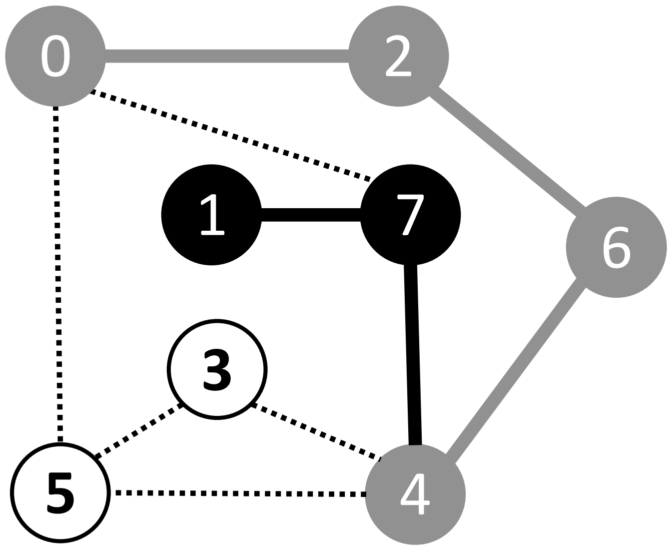


[0, 2, 6, 4, 7, 1]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	0	1	0	1	1

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

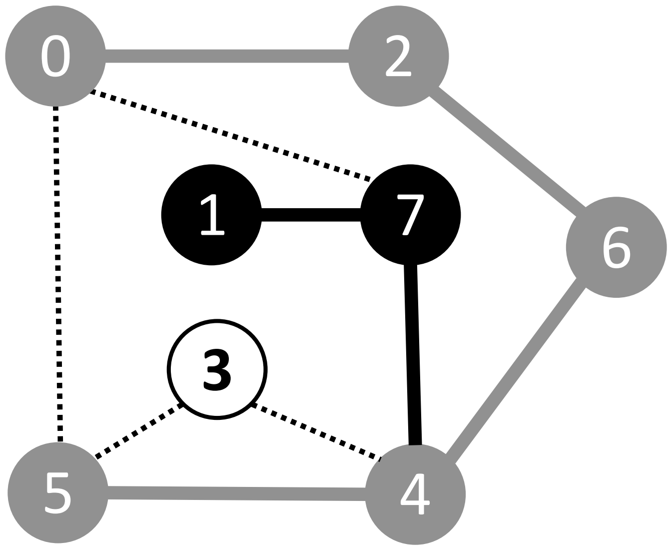


[0, 2, 6, 4, 7, 1]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	0	1	0	1	1

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

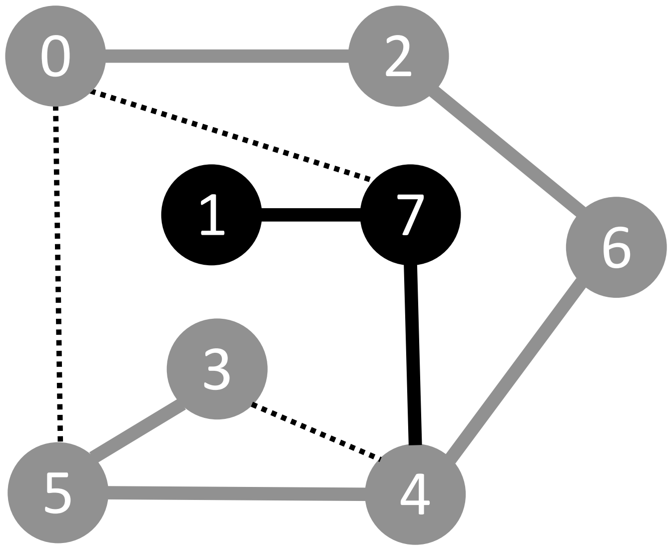


[0, 2, 6, 4, 7, 1, 5]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	0	1	1	1	1

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

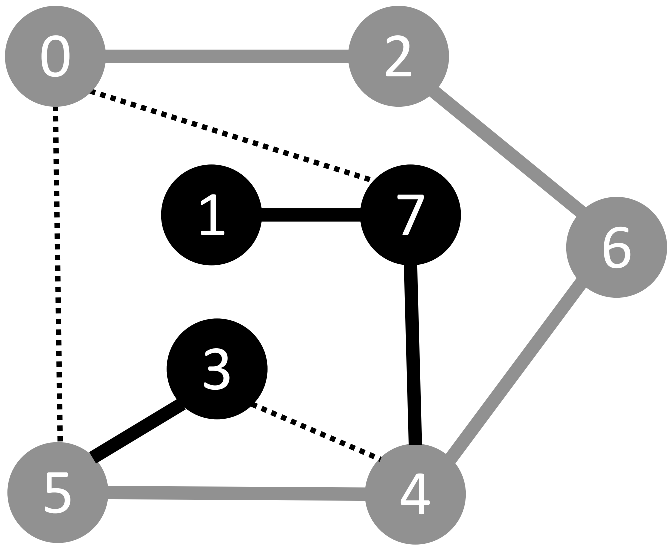


[0, 2, 6, 4, 7, 1, 5, 3]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	1	1	1	1	1

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

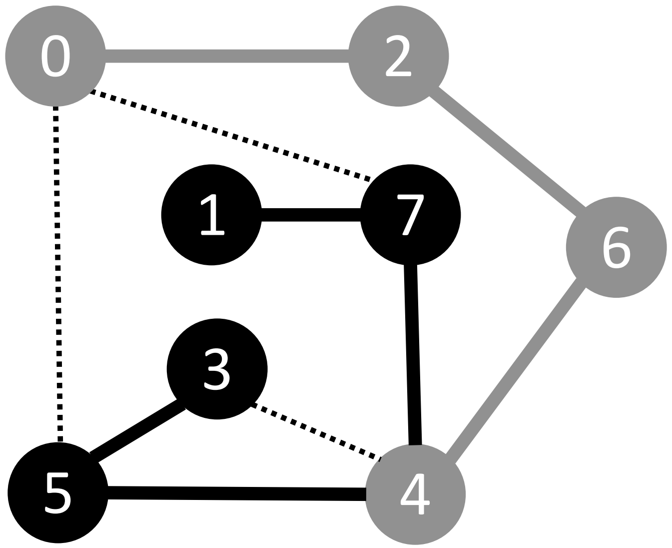


[0, 2, 6, 4, 7, 1, 5, 3]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	1	1	1	1	1

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину

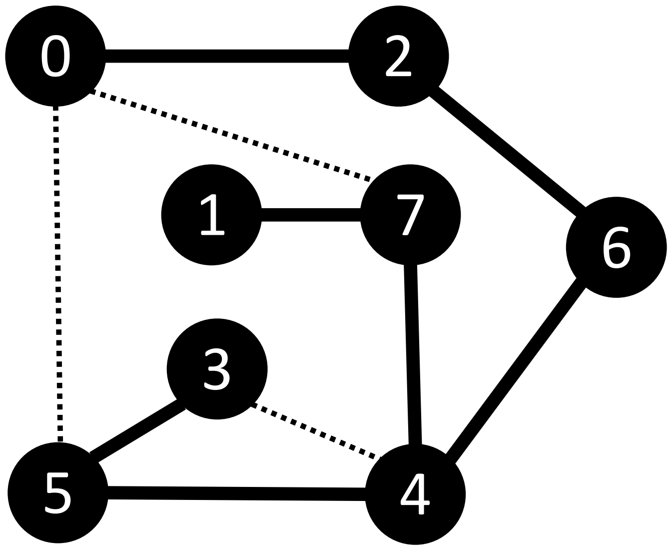


[0, 2, 6, 4, 7, 1, 5, 3]

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	1	1	1	1	1

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Обход в глубину



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

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	1	1	1	1	1

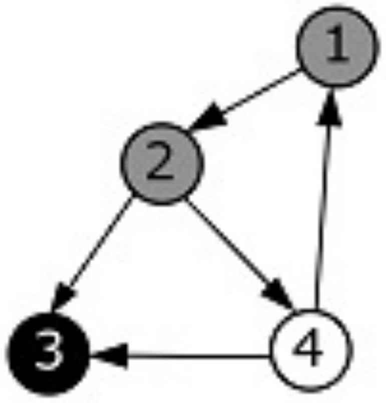
```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Цвета вершин

Белая – в ней еще не были.

Серая – проходимся текущим вызовом dfs.

Черная – пройдена, итерации завершены.



Подграф предшествования.

$G_P = (V, E_P)$, где $E_P = \{(p[u], u)\}$, $p[u]$ – вершина, от которой был вызван dfs(u).

G_P – лес обхода в глубину, состоящий из нескольких деревьев.

Типы ребер графа относительно dfs

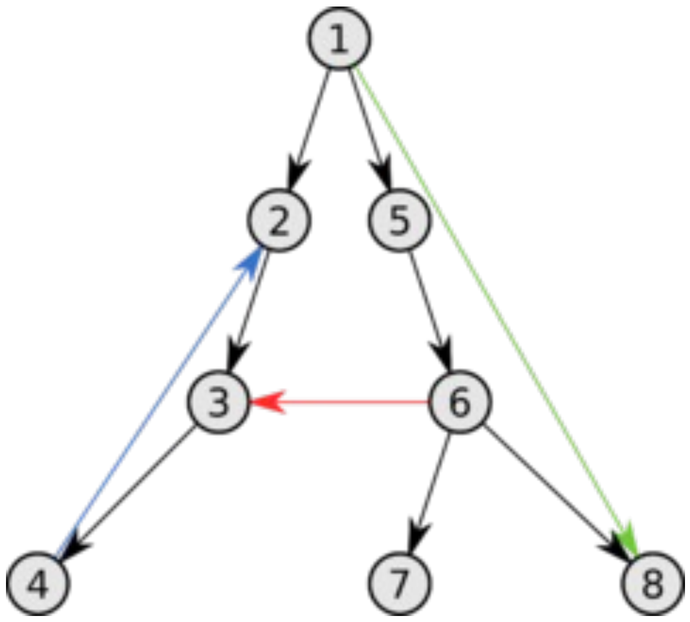
1. **Ребра дерева** $\in G_P$.
2. Ребра (u, v), соединяющие u с предком v – **обратные** ребра.
3. Ребра (u, v), соединяющие u с потомком v – **прямые** ребра.
4. Остальные ребра (u, v) – **перекрестные** ребра.

Переход в белую вершину в dfs – ребро дерева.

Переход в серую вершину в dfs – обратное ребро.

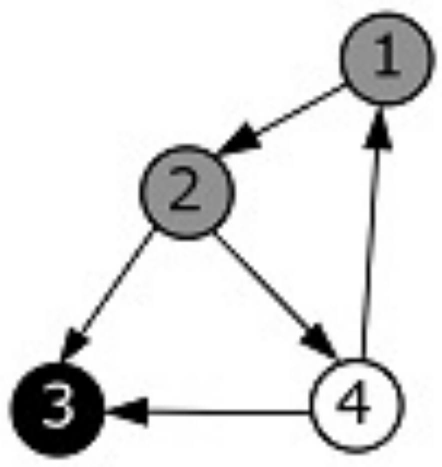
Переход в черную вершину в dfs – прямое или перекрестное.

Последние можно различить по времени входа и выхода.



1) ребра дерева
2) обратные ребра
3) прямые ребра
4) перекрестные ребра

Времена входа и выхода




```
dfs( u ) {
    entry[u] = time++;

    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );

    leave[u] = time++;
}
```

В дереве dfs вершина u – предок v (ребро (u, v) - прямое), если

$$entry[u] < entry[v] \quad \text{и} \quad leave[u] > leave[v].$$

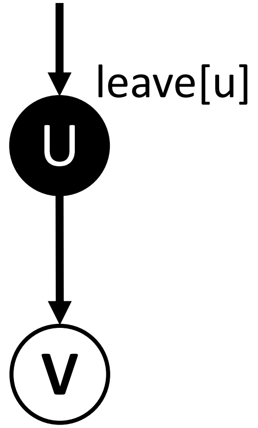
Простая лемма

Лемма. Не существует момента поиска в глубину такого, в котором существует ребро из черной вершины в белую.

Доказательство. От противного. Пусть такое ребро (u, v) и момент *time* существуют.
Рассмотрим момент *leave[u]*. Этот момент – первый, в котором вершина u – черная. Т.е.

$$leave[u] \le time.$$

Следовательно, вершина v в момент *leave[u]* – белая, т.к. она белая в момент *time*.
Но это означается, что на момент выхода из вершины u есть необработанное ребро (u, v). Противоречие.



Лемма о белых путях

Лемма о белых путях. Пусть есть некоторый обход dfs в графе G.

entry[u] и *leave[u]* – моменты входа и выхода из вершины u. Тогда между этими моментами:

Черные и серые вершины $G\backslash u$ не поменяют свой цвет.

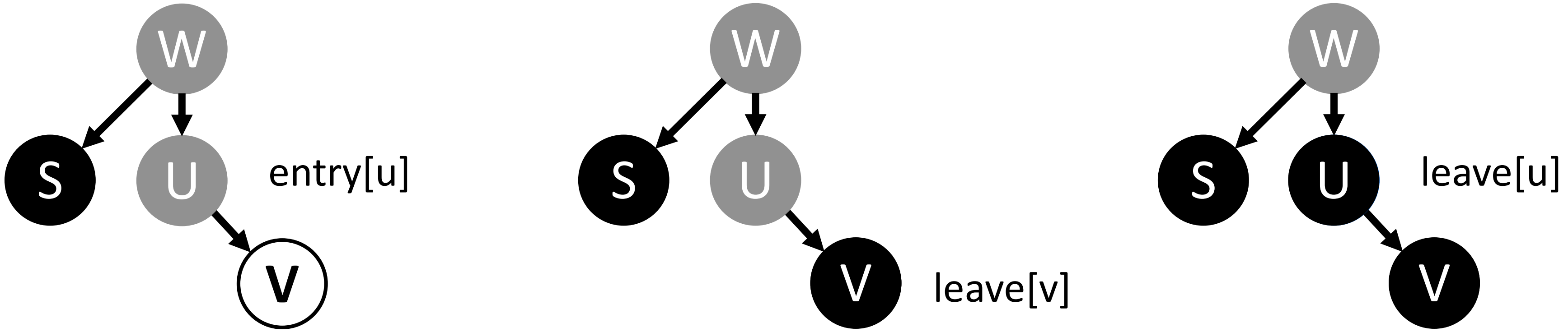
Белые вершины $G\backslash u$ либо останутся белыми, либо станут черными. Причем черными станут те, которые были достижимы из u по белым путям и только они.

Доказательство. Черная вершина останется черной.

Серая вершина останется серой, т.к. находится в стеке рекурсии.

Достижимая белая вершина станет черной. Иначе на пути к ней в момент *leave[u]* будет ребро из некоторой черной вершины в некоторую белую, чего не может быть по лемме.

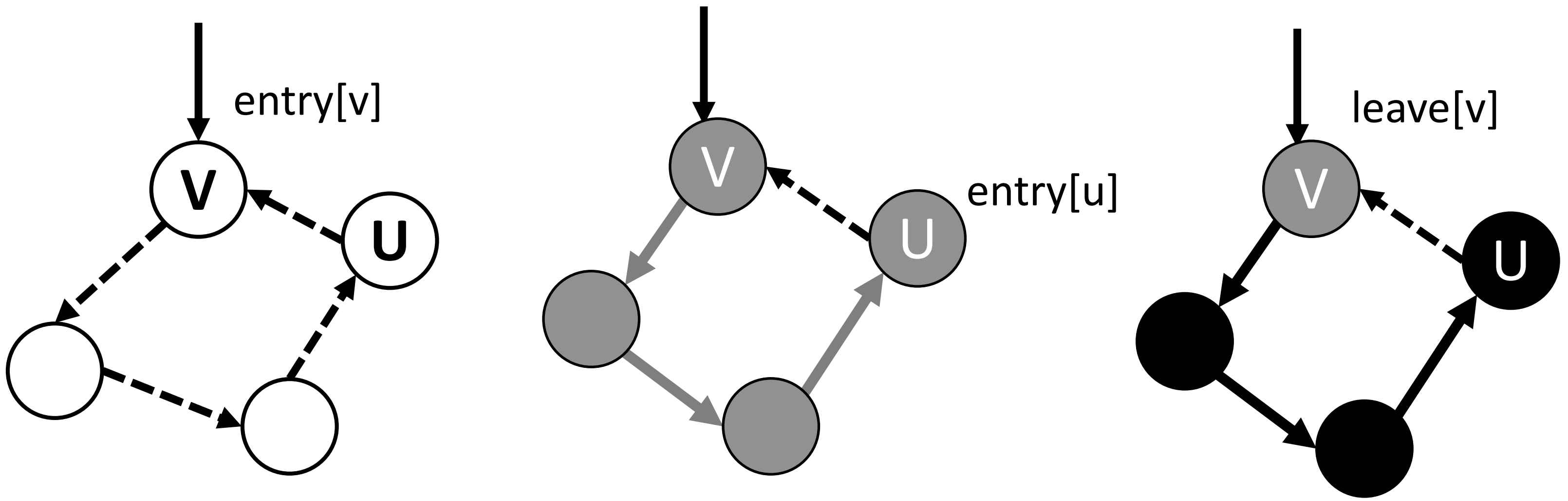
Если вершина стала черной к моменту *leave[u]* , значит, она была достижима из u по белому пути.



Проверка наличия циклов

Задача. Есть ориентированный или неориентированный граф G. Проверить наличие циклов в графе и, если циклы есть, найти какой-нибудь цикл.

Решение. Если в некоторый момент некоторого обхода dfs нашли обратное ребро (ведущее из текущей вершины в серую), то цикл существует. Иначе цикла нет.



Поиск цикла

Время работы проверки наличия цикла – O(V + E).

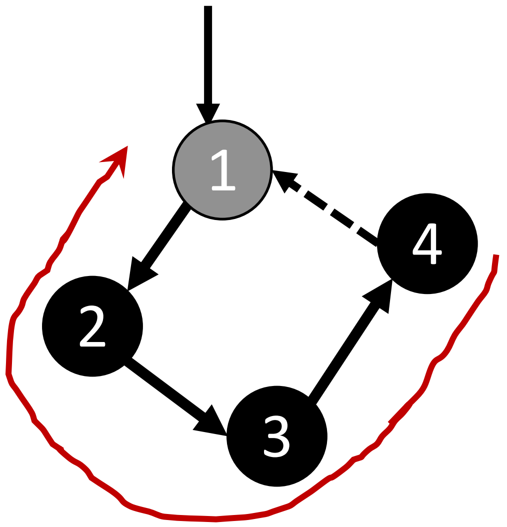
Задача. Найти какой-нибудь цикл в графе с циклами.

Решение.

Пусть в момент time в dfs была найден переход (u, v) в серую вершину v.

Цикл восстанавливается по предкам (стек вызовов в момент *entry[u]*):

$$v, u, p[u], p\big[p[u]\big], \dots, v.$$



Проверка связности

Задача. Проверить, является ли неориентированный граф G связным.

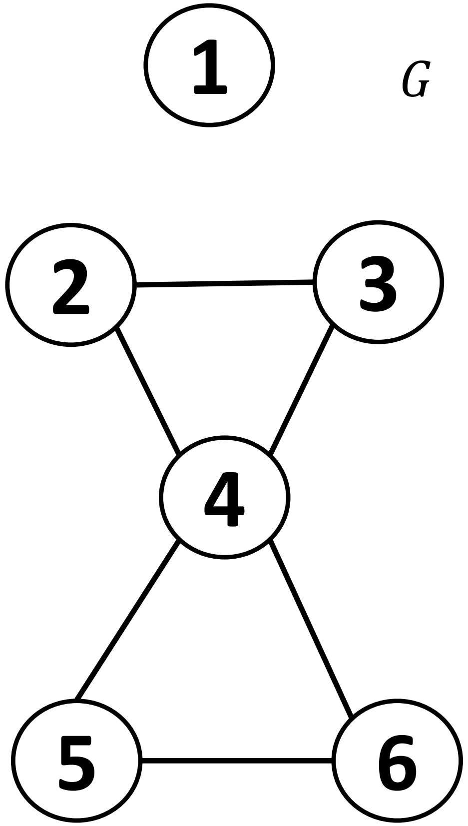
Решение.

Запустим dfs(v).

Если после выхода все вершины посетили ⇔ связность.

После выхода из dfs(v) все вершины можно не проверять на visited[u], если использовать переменную для подсчета числа обработанных вершин.

Время работы $O(V + E)$.



Топологическая сортировка

Топологическая сортировка ациклического графа $G = (V, E)$ – такое упорядочивание всех вершин V, что если $(u, v) \in E$, то u располагается до v.
Формально: упорядочивание

$$\phi: V \to \{1, \dots, n\}, \phi(u) < \phi(v).$$

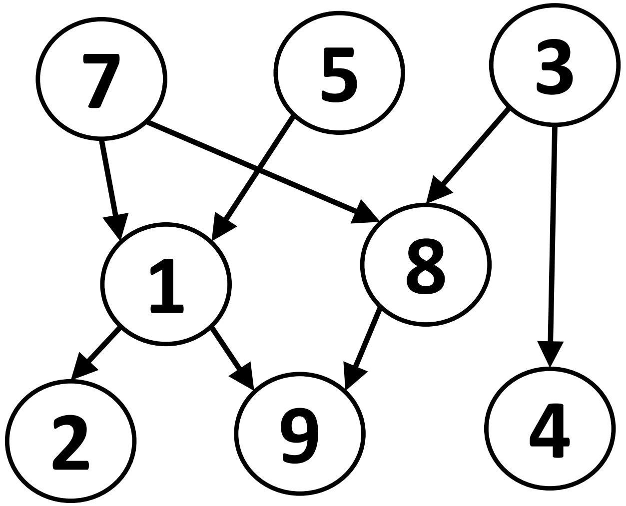
Топологическая сортировка для графов с циклами невозможна.

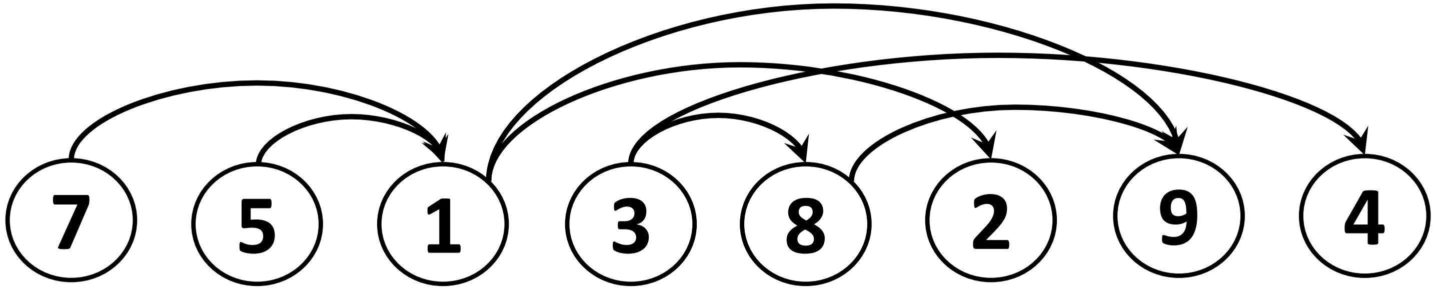
Топологическая сортировка

Строится корректная последовательность зависимых действий.

Например:

- Сборка исходников в правильном порядке
- Порядок прохождения обучающих курсов
- Порядок выполнения технологических операций





Топологическая сортировка

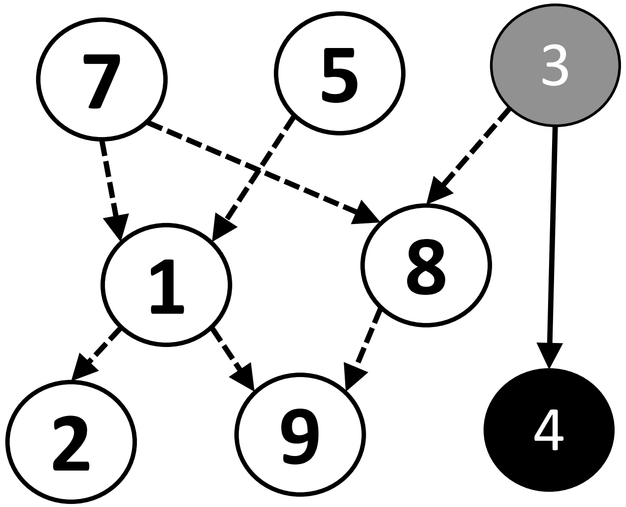
Алгоритм топологической сортировки:

- Запустить DFS, считать $leave$.
- $\phi(v) = |V| + 1 - leave[v]$

Время работы $T = O(V + E)$.

```
vector<bool> visited;
list<int> sorted;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
    sorted.push_front(u);
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Топологическая сортировка

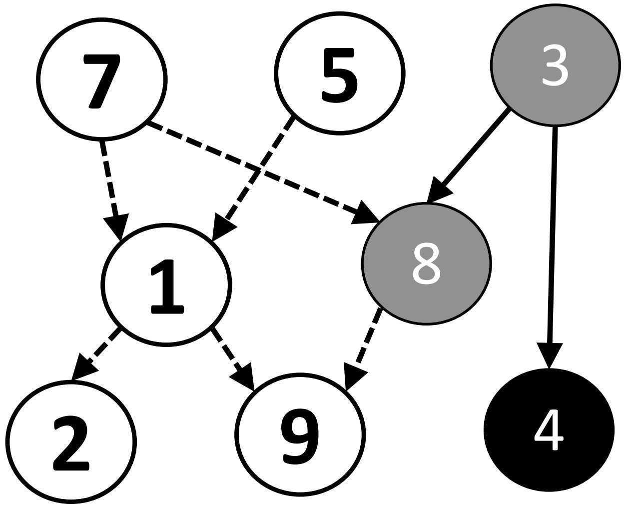



```
vector<bool> visited;
list<int> sorted;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
    sorted.push_front(u);
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Добавляем элемент в начало списка в момент окраски в чёрный

sorted

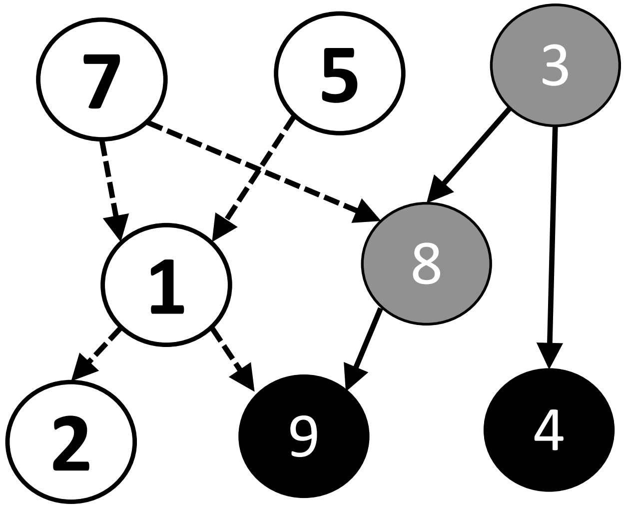
Топологическая сортировка



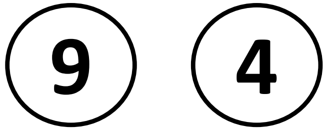

```
vector<bool> visited;
list<int> sorted;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
    sorted.push_front(u);
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

sorted

Топологическая сортировка

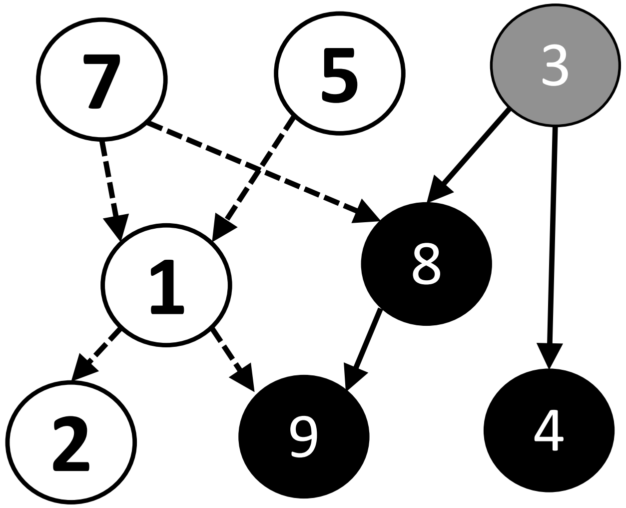



```
vector<bool> visited;
list<int> sorted;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
    sorted.push_front(u);
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

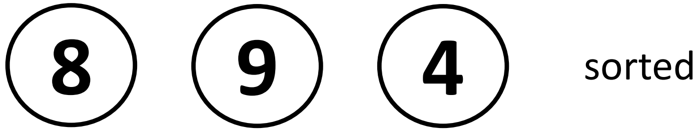


sorted

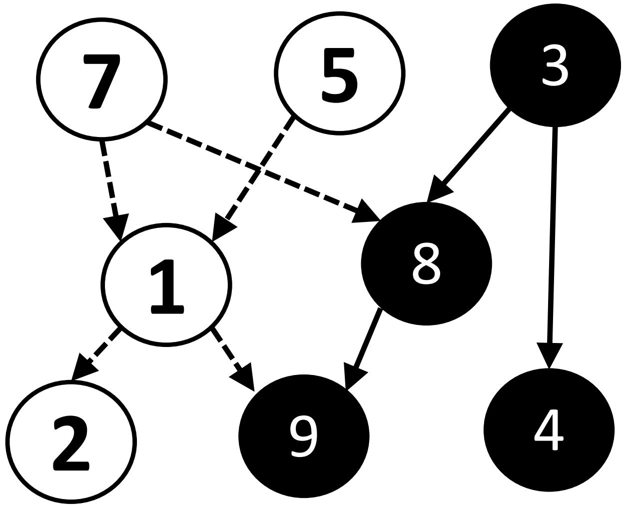
Топологическая сортировка




```
vector<bool> visited;
list<int> sorted;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
    sorted.push_front(u);
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

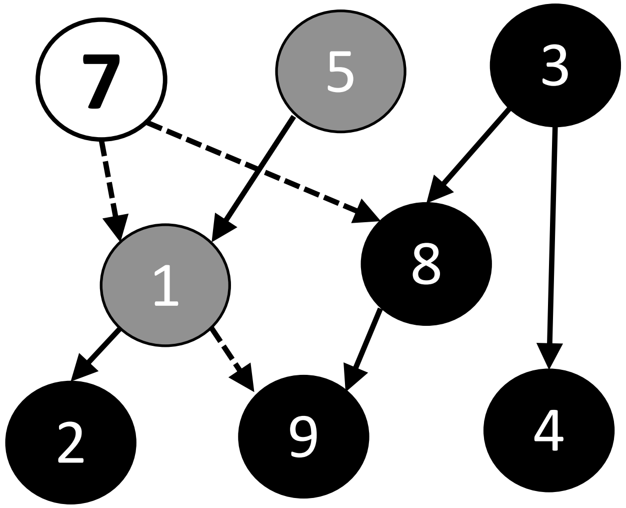


Топологическая сортировка




```
vector<bool> visited;
list<int> sorted;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
    sorted.push_front(u);
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

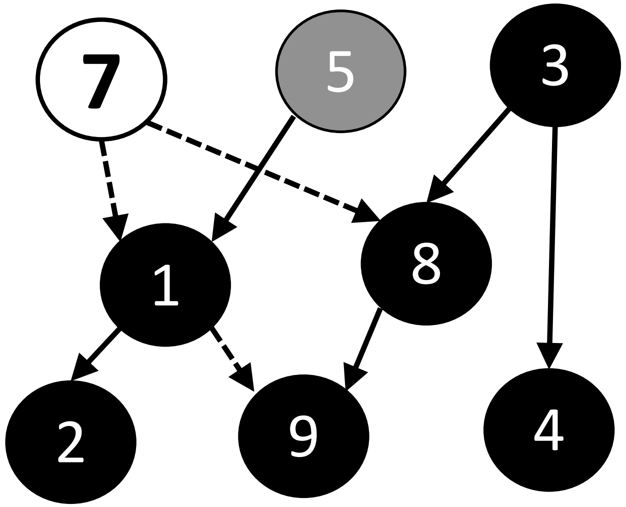
Топологическая сортировка




```
vector<bool> visited;
list<int> sorted;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
    sorted.push_front(u);
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

2 3 8 9 4 sorted

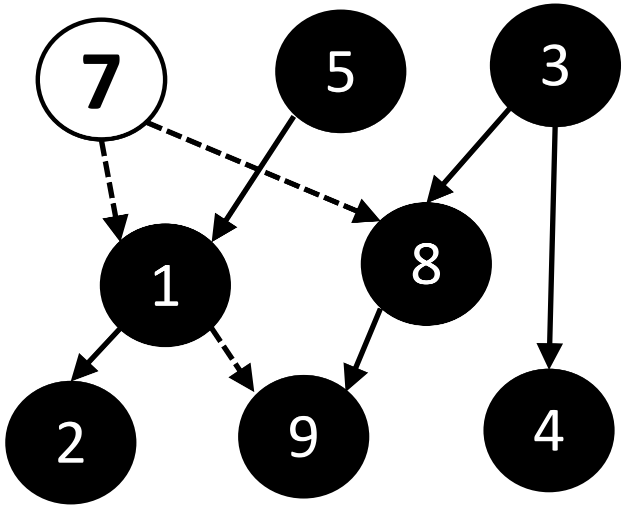
Топологическая сортировка




```
vector<bool> visited;
list<int> sorted;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
    sorted.push_front(u);
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

1 2 3 8 9 4 sorted

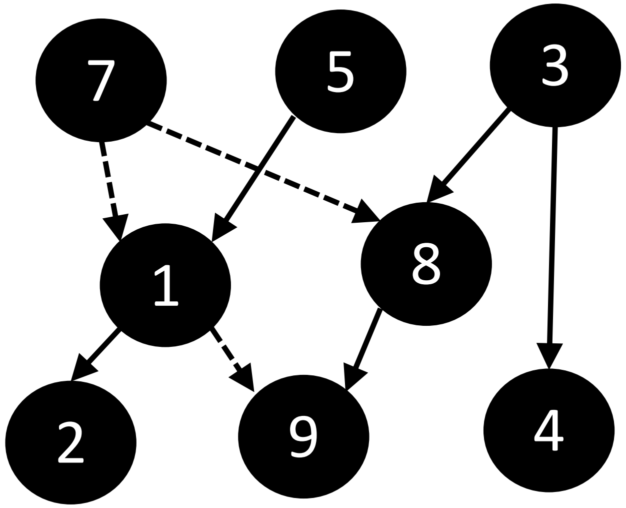
Топологическая сортировка



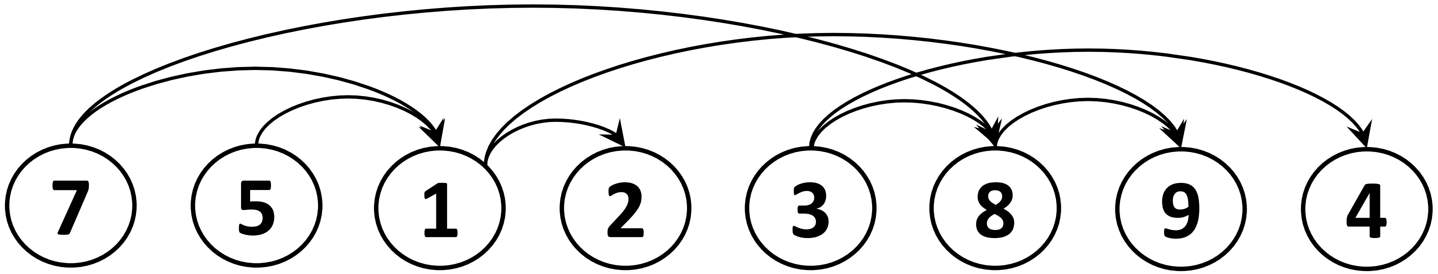

```
vector<bool> visited;
list<int> sorted;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
    sorted.push_front(u);
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

sorted

Топологическая сортировка




```
vector<bool> visited;
list<int> sorted;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
    sorted.push_front(u);
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```



sorted

Алгоритм Косарайю

Алгоритм Косарайю (1978г) – алгоритм поиска сильно связных компонент.

Пусть $G = (V, E)$.

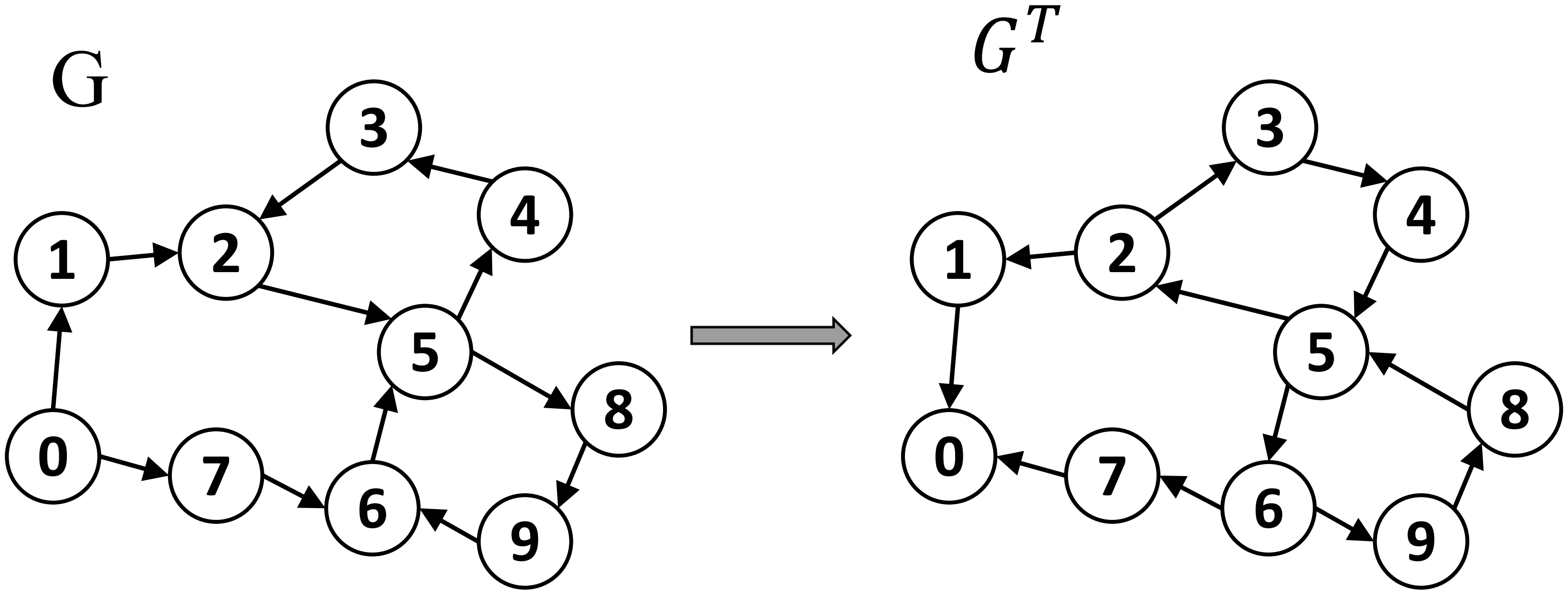
Алгоритм:

1) Построим $H = G^T$ – граф, являющийся инвертированным к G.
2) DFS(H)
3) DFS(G), перебирая вершины в MainDFS в порядке убывания $leave_H$.

Деревья, полученные запусками dfs на шаге 3 – компоненты сильной связности.

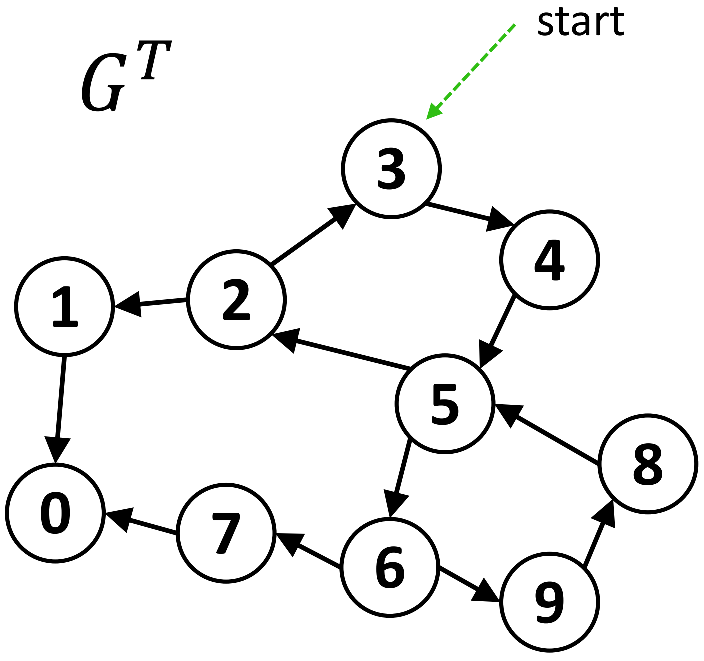
Алгоритм Косарайю

1) Построим $H = G^T$ – граф, являющийся инвертированным к G.



Алгоритм Косарайю

2) Обходим в глубину граф $H = G^T$, для всех вершин запоминаем $leave_H$



$leave_H: 0$

Порядок обхода:

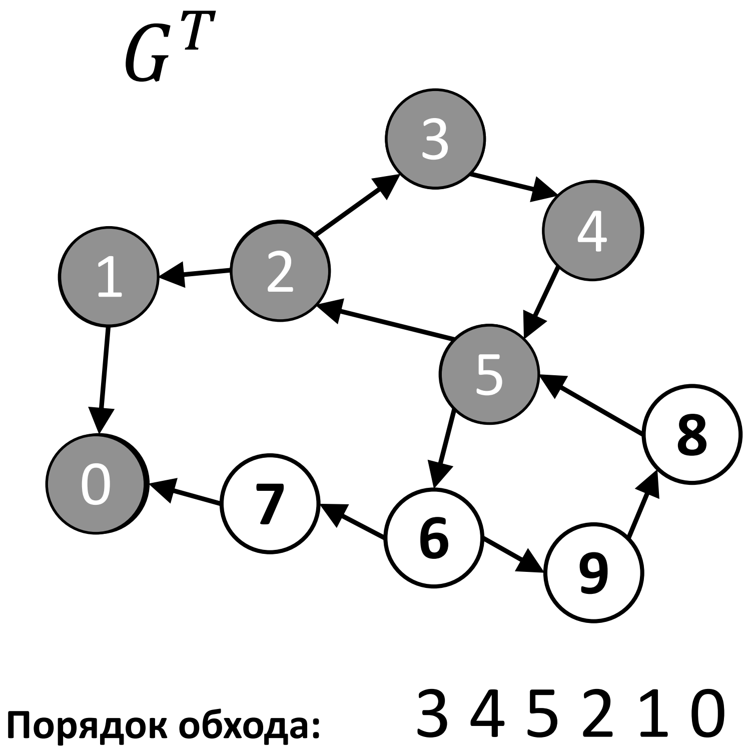
	Visited
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

	Order
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Алгоритм Косарайю

2) Обходим в глубину граф $H = G^T$, для всех вершин запоминаем $leave_H$

$leave_H: 0$



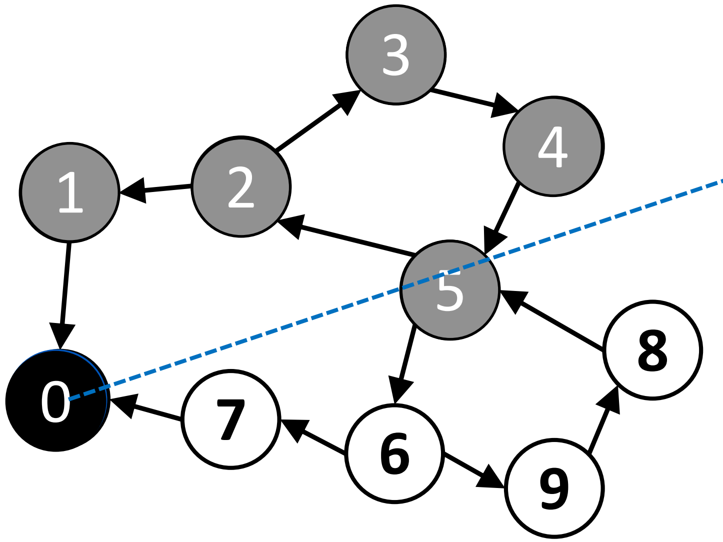
	Visited
0	1
1	1
2	0
3	1
4	1
5	1
6	0
7	0
8	0
9	0

	Order
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Алгоритм Косарайю

В момент, когда вершина становится чёрная, пишем order[$leave_H$] = v

G^T



$leave_H: 0$

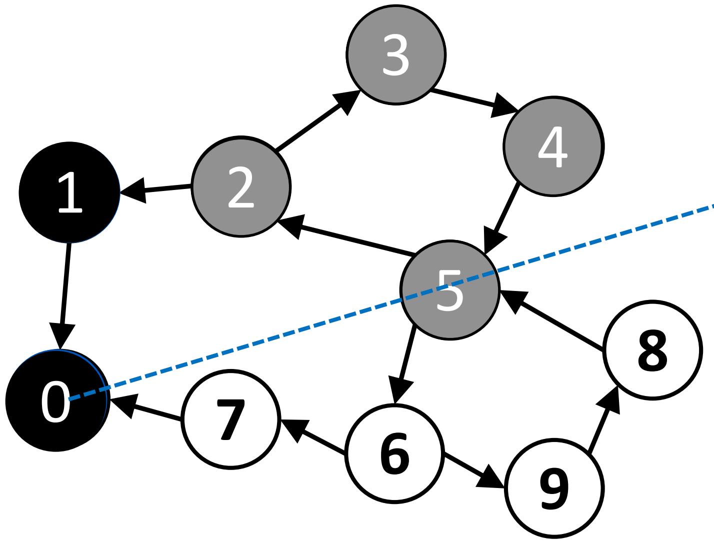
	Visited		Order
0	1	0	0
1	1	1	
2	1	2	
3	1	3	
4	1	4	
5	1	5	
6	0	6	
7	0	7	
8	0	8	
9	0	9	

Порядок обхода: 3 4 5 2 1 0

Алгоритм Косарайю

В момент, когда вершина становится чёрная, пишем order[$leave_H$] = v

G^T



$leave_H$: 1

	Visited	Order
0	1	0
1	1	1
2	1	
3	1	
4	1	
5	1	
6	0	
7	0	
8	0	
9	0	

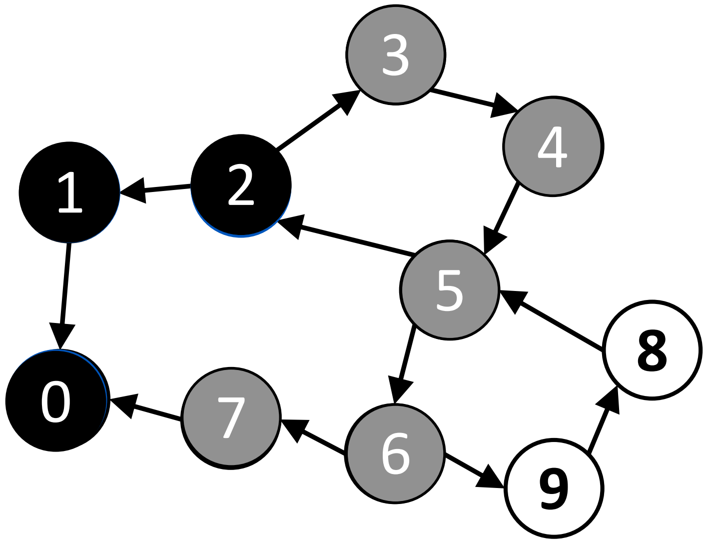
Порядок обхода: 3 4 5 2 1 0

Алгоритм Косарайю

В момент, когда вершина становится чёрная, пишем order[$leave_H$] = v

$leave_H$: 3

G^T



Порядок обхода: 3 4 5 2 1 0 6 7

Visited

0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	0
9	0

Order

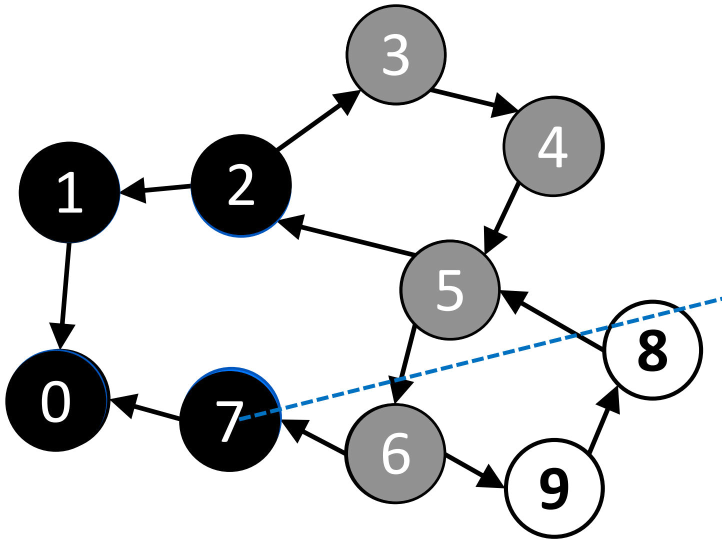
0	0
1	1
2	2
3	
4	
5	
6	
7	
8	
9	

Алгоритм Косарайю

В момент, когда вершина становится чёрная, пишем order[$leave_H$] = v

$leave_H: 3$

G^T



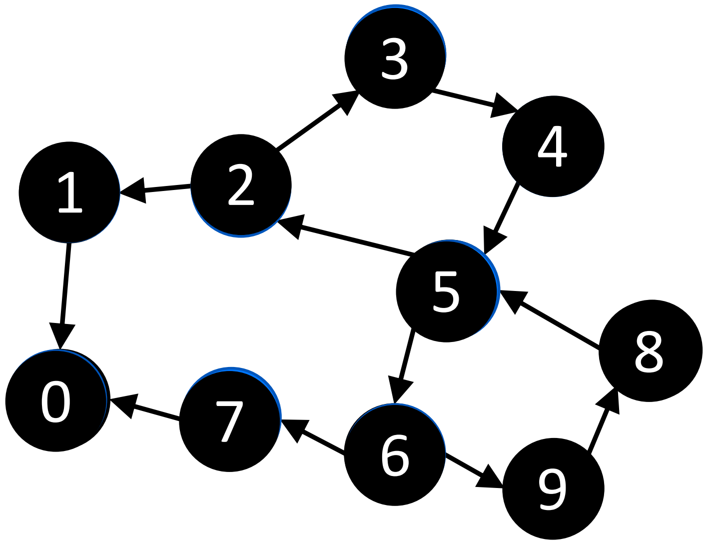
	Visited	Order
0	1	0
1	1	1
2	1	2
3	1	7
4	1	
5	1	
6	1	
7	1	
8	0	
9	0	

Порядок обхода: 3 4 5 2 1 0 6 7

Алгоритм Косарайю

По сути в Order получилась топологическая сортировка

G^T



$leave_H: 9$

Порядок обхода: 3 4 5 2 1 0 6 7 9 8

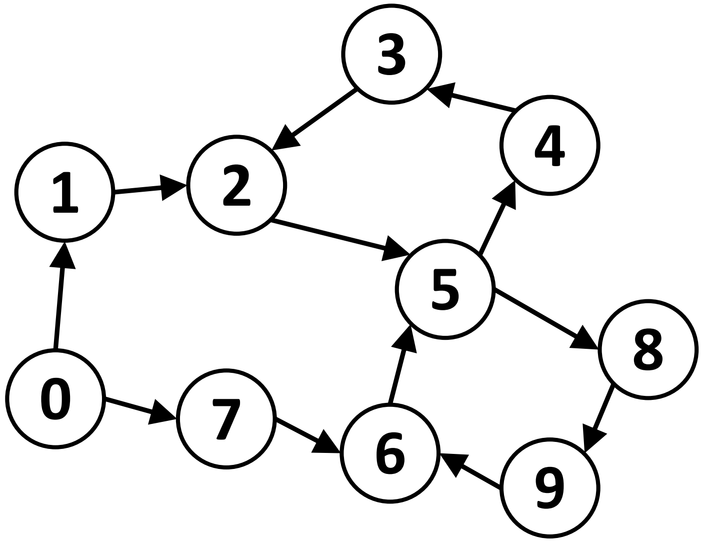
	Visited
0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1

	Order
0	0
1	1
2	2
3	7
4	8
5	9
6	6
7	5
8	4
9	3

Алгоритм Косарайю

3) Обходим в глубину исходный граф G перебирая вершины в MainDFS в порядке убывания $leave_H$.

G



Visited

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Order

0	0
1	1
2	2
3	7
4	8
5	9
6	6
7	5
8	4
9	3

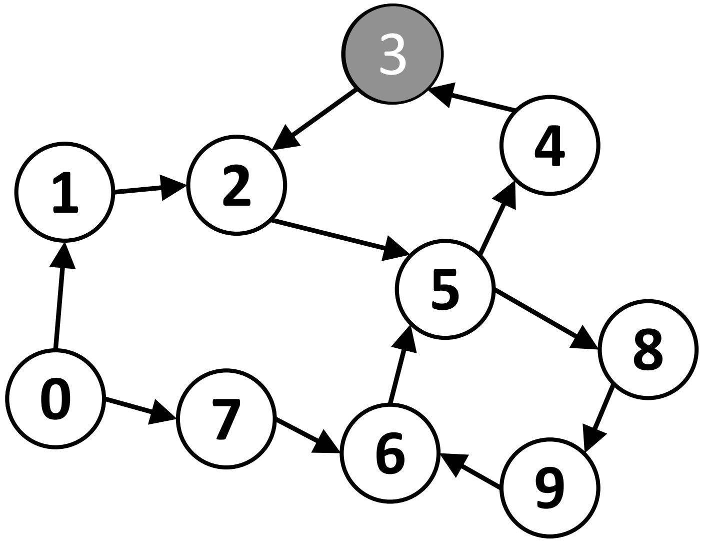
КСС

Порядок перебора вершин

Алгоритм Косарайю

Начинаем с вершины 3, у нее самое большое $leave_H$

G



	Visited
0	0
1	0
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

	Order
0	0
1	1
2	2
3	7
4	8
5	9
6	6
7	5
8	4
9	3

КСС

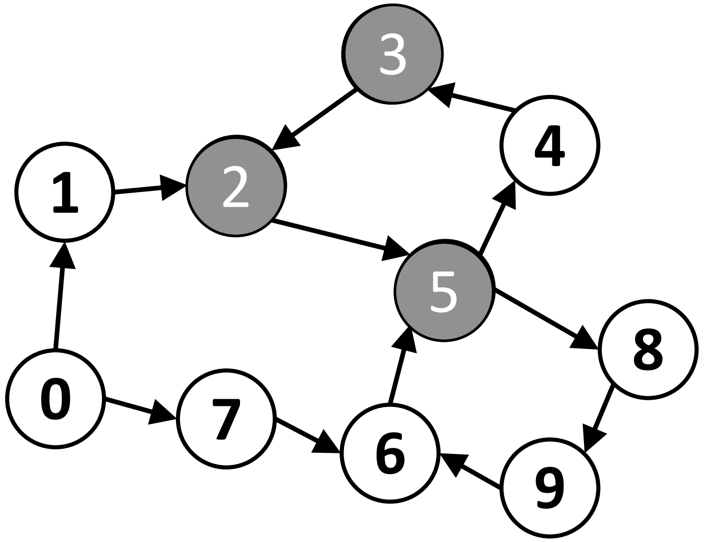
КСС1: 3

Порядок перебора вершин

Алгоритм Косарайю

Начинаем с вершины 3, у нее самое большое $leave_H$

G



	Visited		Order
0	0	0	0
1	0	1	1
2	1	2	2
3	1	3	7
4	0	4	8
5	1	5	9
6	0	6	6
7	0	7	5
8	0	8	4
9	0	9	3

КСС

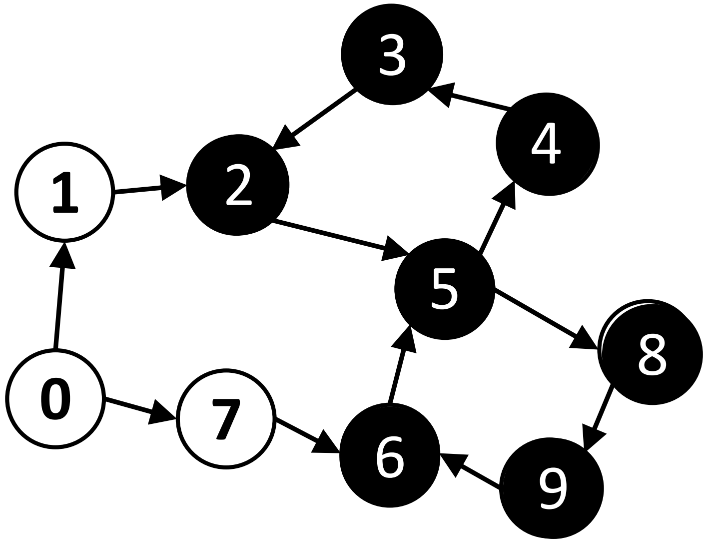
КСС1: 3 2 5

Порядок перебора вершин

Алгоритм Косарайю

Пропускаем вершины 4, 5, 6, 9, 8, так как уже посетили их.

G



	Visited	Order
0	0	0
1	0	1
2	1	2
3	1	7
4	1	8
5	1	9
6	1	6
7	1	5
8	1	4
9	1	3

Уже посетили

КСС

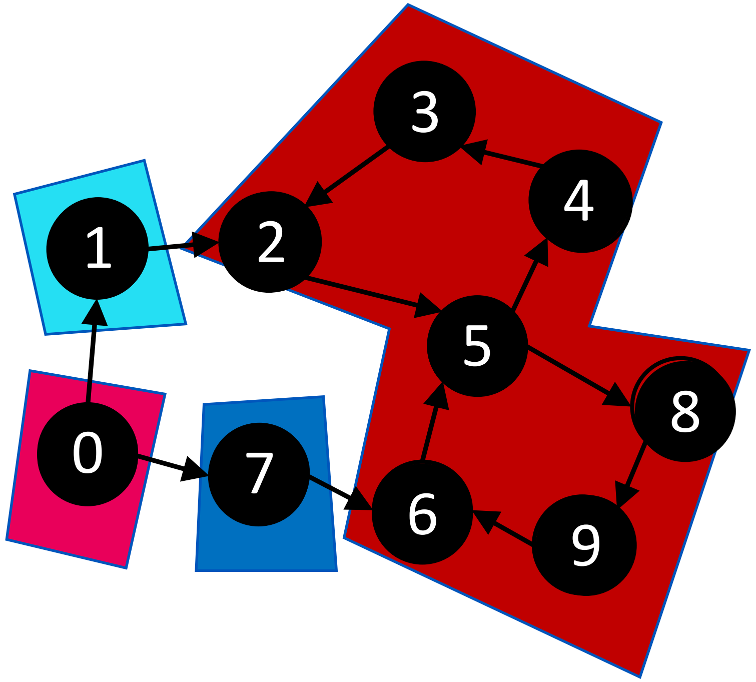
КСС1: 3 2 5 8 9 6 4

КСС2: 7

Алгоритм Косарайю

В итоге получаем список КСС.

G



Visited

0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1

Order

0	0
1	1
2	2
3	7
4	8
5	9
6	6
7	5
8	4
9	3

КСС

КСС1: 3 2 5 8 9 6 4

КСС2: 7

КСС3: 1

КСС4: 0

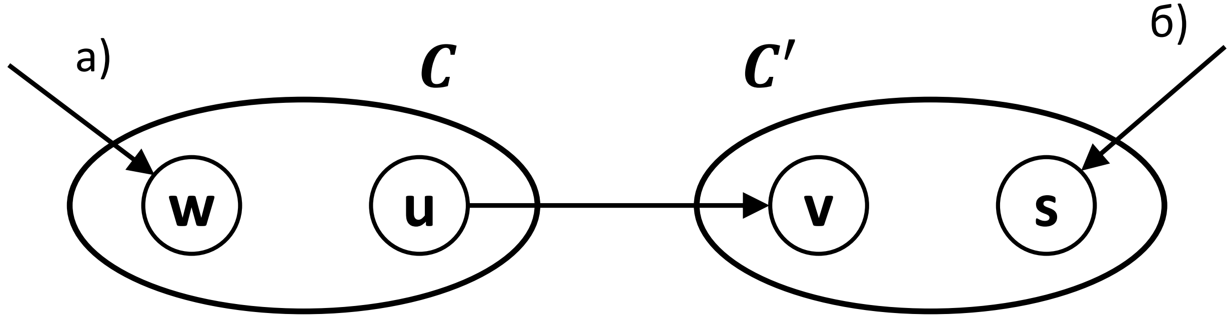
Алгоритм Косарайю

Пусть C – КСС. Обозначим leave[C] – максимальное время выхода leave[v], $v \in C$.

Лемма. Пусть C, C’ – две различные КСС, и есть ребро (u, v) между ними, $u \in C, v \in C'$. Тогда $leave[C] > leave[C']$.

Доказательство леммы. а) Первой была достигнута КСС C – вершина w. Тогда в момент входа в C вся компонента C’ – белая и достижима из C. По лемме о белых путях в момент $leave[w] > leave[C']$.

б) Первой была достигнута КСС C’. В этом случае вся компонента C’ будет пройдена до обхода C, т.к. не существует пути из C’ в C. То есть $leave[C] > leave[C']$.



Алгоритм Косарайю

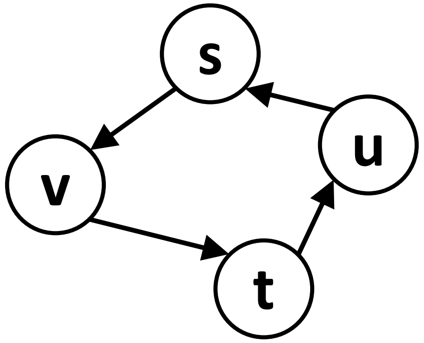
Теорема. Деревья, полученные в п. 3 алгоритма Косарайю, – КСС.

Доказательство. <=

Пусть вершины s и t из одной КСС.

Тогда существуют пути $s \rightsquigarrow t$ и $t \rightsquigarrow s$. Значит, вершины s и t попадут в одно дерево в шаге 3).

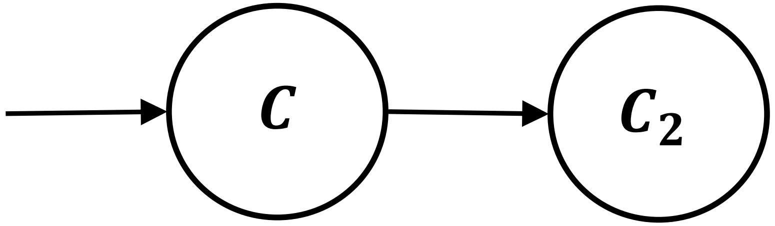
Это следует из следующего рассуждения:
Пусть v – первая вершина из цикла $s \rightsquigarrow t \rightsquigarrow s$ в обходе DFS(G). Тогда в момент $entry[v]$ вершины s и t достижимы из v по белым путям. По лемме о белых путях они будут обработаны в dfs(v).



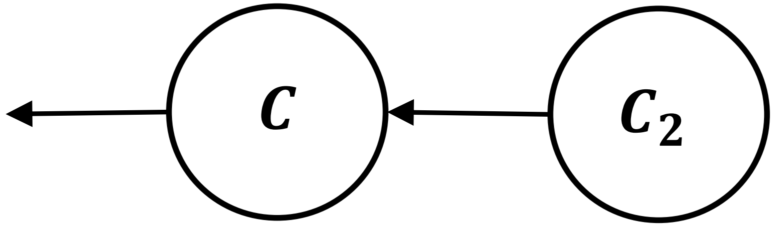
Далее рассмотрим дерево T – дерево обхода dfs на этапе 3.
Докажем, что T – компонента сильной связности.

Алгоритм Косарайю

Если можем пройти по ребру при обходе dfs в п. 3, значит вершины из C_2 не отмечены как visited



Но такого быть не может, потому что при обходе инвертированного графа по лемме $leave_H[C] < leave_H[C_2]$



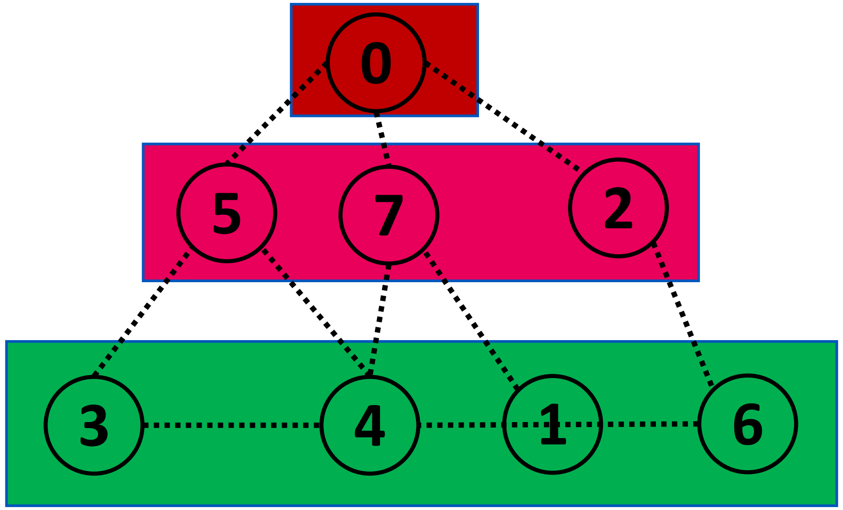
В п.3 перебираем вершины в порядке убывания $leave_H$. Следовательно, вершины из C_2 будут отмечены как visited до начала обхода C.

Обход в ширину

BFS – Breadth First Search – обход в ширину.

Обход, при котором вершины обходятся в порядке увеличения расстояния от стартовой вершины.

Обход, при котором вершины обходятся «по слоям».



Как и в обходе в ширину деревьев используется очередь.

Обход в ширину

BFS – Breadth First Search – обход в ширину.

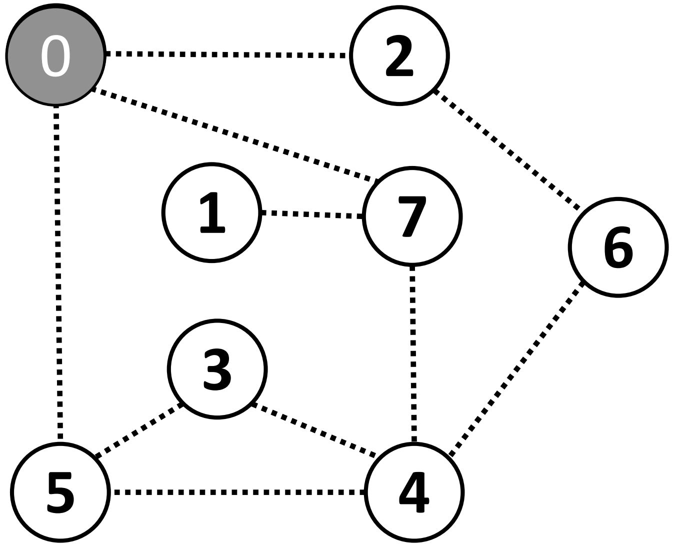
Обход, при котором вершины обходятся в порядке увеличения расстояния от стартовой вершины.

Обход, при котором вершины обходятся «по слоям».

Как и в обходе в ширину деревьев используется очередь.

```
vector<bool> visited;
void bfs( int u ) {
    std::queue<int> q;
    q.push( u ); visited[q] = true;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E ) {
            if( !visited[w] ) {
                visited[w] = true;
                q.push( w );
}   }   }   }
int MainBFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            bfs( i );
}
```

Обход в ширину

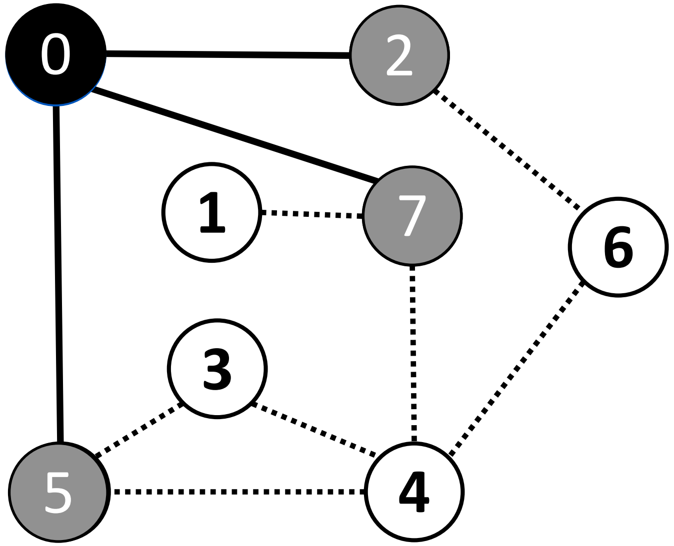


Queue: 0

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	0	0	0	0	0	0

```
vector<bool> visited;
void bfs( int u ) {
    std::queue<int> q;
    q.push( u ); visited[q] = true;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E ) {
            if( !visited[w] ) {
                visited[w] = true;
                q.push( w );
}   }   }   }
int MainBFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            bfs( i );
}
```

Обход в ширину

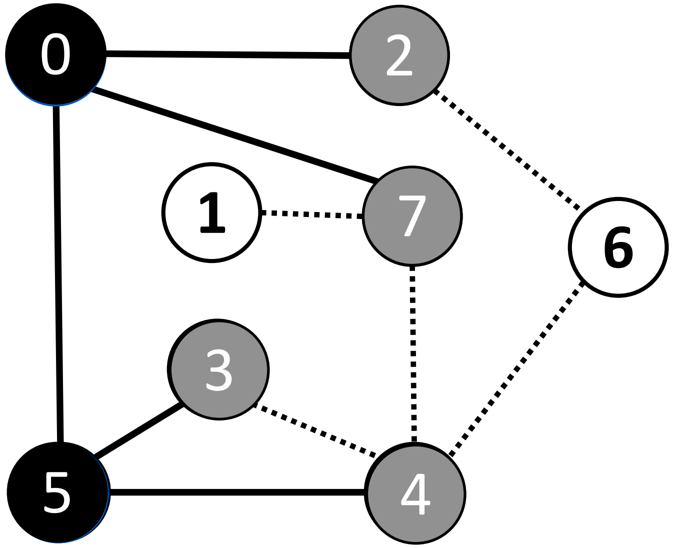


Queue: 5 7 2

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	0	0	1	0	1

```
vector<bool> visited;
void bfs( int u ) {
    std::queue<int> q;
    q.push( u ); visited[q] = true;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E ) {
            if( !visited[w] ) {
                visited[w] = true;
                q.push( w );
}   }   }   }
int MainBFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            bfs( i );
}
```

Обход в ширину

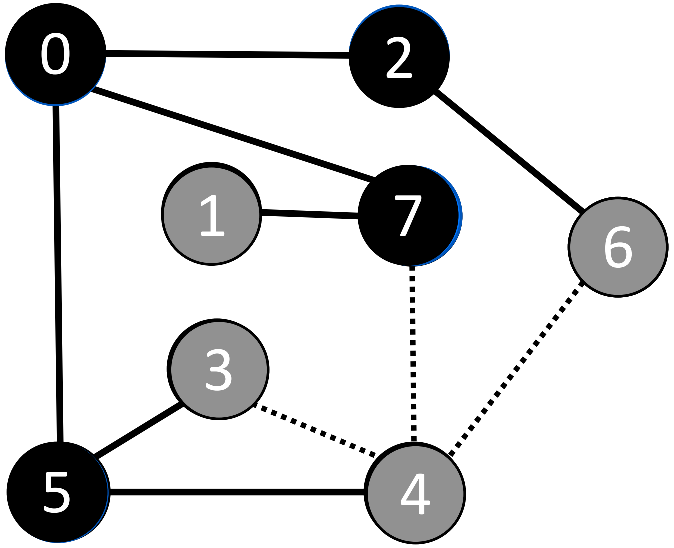


Queue: 7 2 3 4

Индекс	0	1	2	3	4	5	6	7
Visited	1	0	1	1	1	1	0	1

```
vector<bool> visited;
void bfs( int u ) {
    std::queue<int> q;
    q.push( u ); visited[q] = true;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E ) {
            if( !visited[w] ) {
                visited[w] = true;
                q.push( w );
}   }   }   }
int MainBFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            bfs( i );
}
```

Обход в ширину



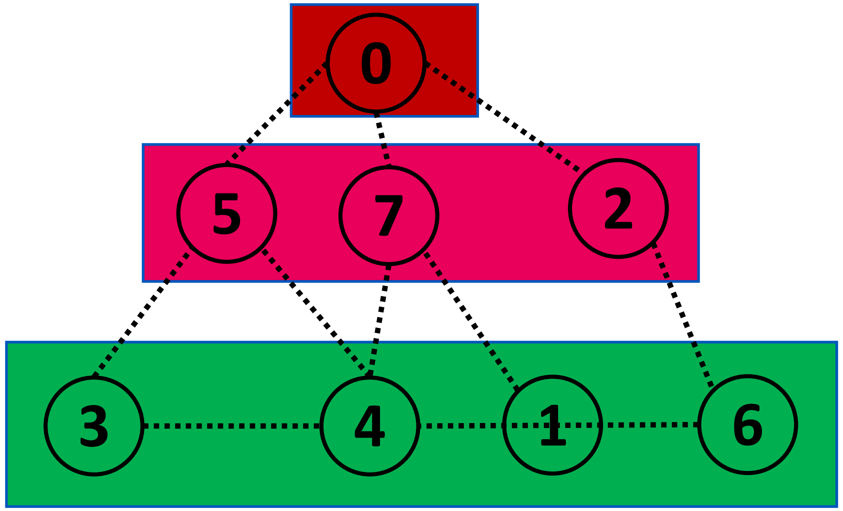
Queue: 3 4 1 6

Индекс	0	1	2	3	4	5	6	7
Visited	1	1	1	1	1	1	1	1

```
vector<bool> visited;
void bfs( int u ) {
    std::queue<int> q;
    q.push( u ); visited[q] = true;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E ) {
            if( !visited[w] ) {
                visited[w] = true;
                q.push( w );
}   }   }   }
int MainBFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            bfs( i );
}
```

Обход в ширину

BFS - обход по «слоям»



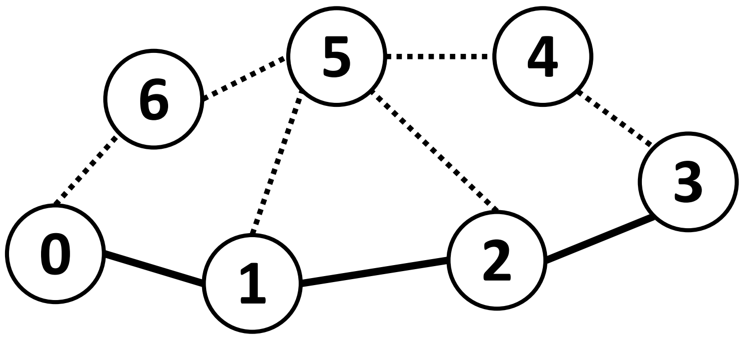
Вершины в одном слое, если у них одинаковое расстояние от стартовой вершины.

Поиск кратчайших путей

Алгоритм:

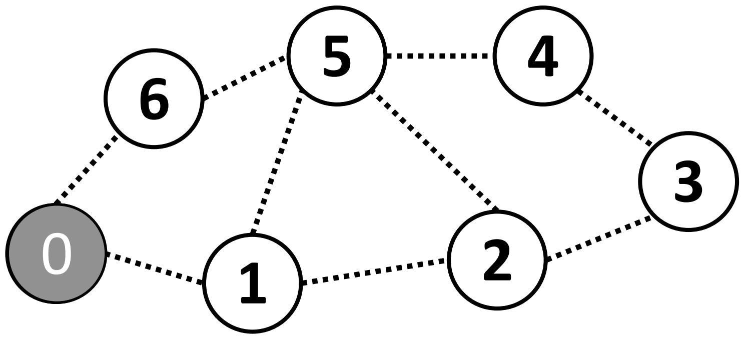
1. Обходим граф в ширину.
2. Для непосещенных вершин запоминаем кратчайший путь.

Сложность $O(V+E)$




```
vector<int> r, pi;
void bfs( int s ) {
    r.assign( n, INT_MAX );
    std::queue<int> q;
    q.push( s ); r[s] = 0; pi[s] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( r[w] > r[v] + 1 ) {
                r[w] = r[v] + 1;
                pi[w] = v;
                q.push( w );
            }
    }
}
```

Поиск кратчайших путей



r

0	0
1	
2	
3	
4	
5	
6	

pi

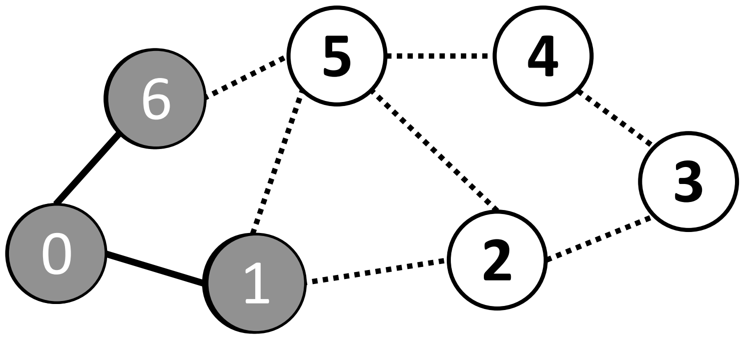
0	-1
1	
2	
3	
4	
5	
6	

Queue

0

```
vector<int> r, pi;
void bfs( int s ) {
    r.assign( n, INT_MAX );
    std::queue<int> q;
    q.push( s ); r[s] = 0; pi[s] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( r[w] > r[v] + 1 ) {
                r[w] = r[v] + 1;
                pi[w] = v;
                q.push( w );
            }
    }
}
```

Поиск кратчайших путей



	r
0	0
1	1
2	
3	
4	
5	
6	1

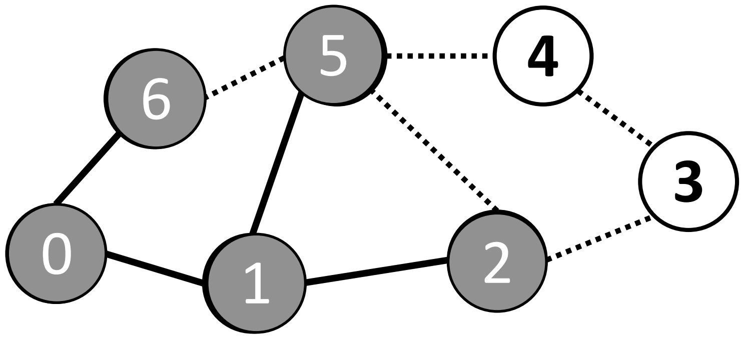
	pi
0	-1
1	0
2	
3	
4	
5	
6	0

Queue

1 6

```
vector<int> r, pi;
void bfs( int s ) {
    r.assign( n, INT_MAX );
    std::queue<int> q;
    q.push( s ); r[s] = 0; pi[s] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( r[w] > r[v] + 1 ) {
                r[w] = r[v] + 1;
                pi[w] = v;
                q.push( w );
            }
    }
}
```

Поиск кратчайших путей



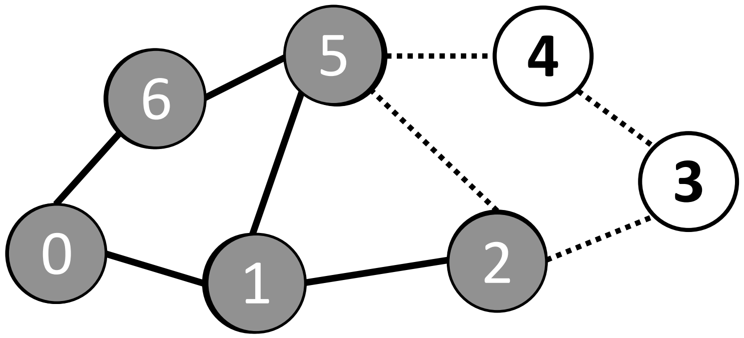
	r		pi
0	0	0	-1
1	1	1	0
2	2	2	1
3		3	
4		4	
5	2	5	1
6	1	6	0

Queue

6 5 2

```
vector<int> r, pi;
void bfs( int s ) {
    r.assign( n, INT_MAX );
    std::queue<int> q;
    q.push( s ); r[s] = 0; pi[s] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( r[w] > r[v] + 1 ) {
                r[w] = r[v] + 1;
                pi[w] = v;
                q.push( w );
            }
    }
}
```

Поиск кратчайших путей



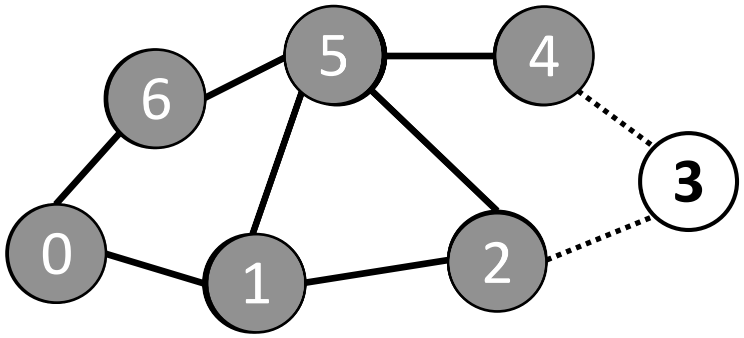
	r		pi
0	0	0	-1
1	1	1	0
2	2	2	1
3		3	
4		4	
5	2	5	1
6	1	6	0

Queue

5 2

```
vector<int> r, pi;
void bfs( int s ) {
    r.assign( n, INT_MAX );
    std::queue<int> q;
    q.push( s ); r[s] = 0; pi[s] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( r[w] > r[v] + 1 ) {
                r[w] = r[v] + 1;
                pi[w] = v;
                q.push( w );
            }
    }
}
```

Поиск кратчайших путей



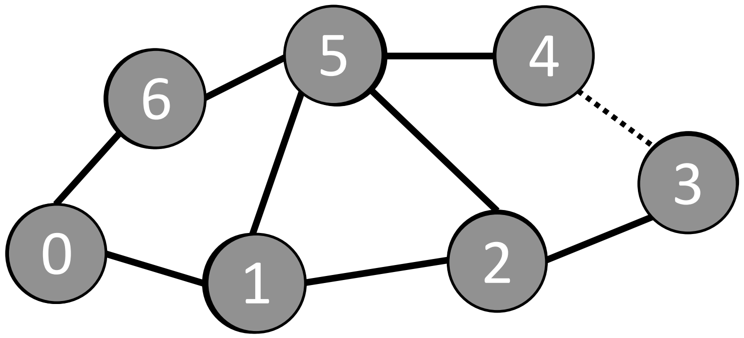
	r	pi
0	0	-1
1	1	0
2	2	1
3		
4	3	5
5	2	1
6	1	0

Queue

2 4

```
vector<int> r, pi;
void bfs( int s ) {
    r.assign( n, INT_MAX );
    std::queue<int> q;
    q.push( s ); r[s] = 0; pi[s] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( r[w] > r[v] + 1 ) {
                r[w] = r[v] + 1;
                pi[w] = v;
                q.push( w );
            }
    }
}
```

Поиск кратчайших путей



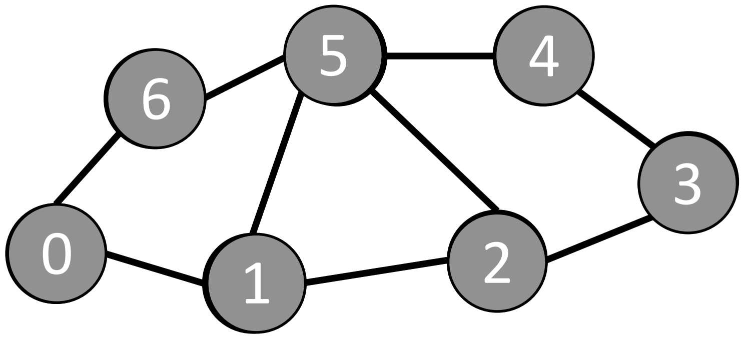
	r		pi
0	0	0	-1
1	1	1	0
2	2	2	1
3	3	3	2
4	3	4	5
5	2	5	1
6	1	6	0

Queue

4 3

```
vector<int> r, pi;
void bfs( int s ) {
    r.assign( n, INT_MAX );
    std::queue<int> q;
    q.push( s ); r[s] = 0; pi[s] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( r[w] > r[v] + 1 ) {
                r[w] = r[v] + 1;
                pi[w] = v;
                q.push( w );
            }
    }
}
```

Поиск кратчайших путей



	r		pi
0	0	0	-1
1	1	1	0
2	2	2	1
3	3	3	2
4	3	4	5
5	2	5	1
6	1	6	0

Queue

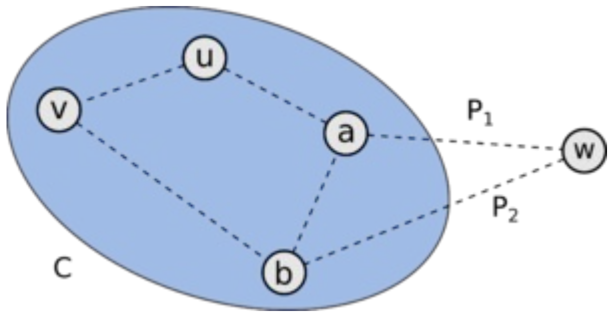
3

```
vector<int> r, pi;
void bfs( int s ) {
    r.assign( n, INT_MAX );
    std::queue<int> q;
    q.push( s ); r[s] = 0; pi[s] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( r[w] > r[v] + 1 ) {
                r[w] = r[v] + 1;
                pi[w] = v;
                q.push( w );
            }
    }
}
```

Реберная двусвязность

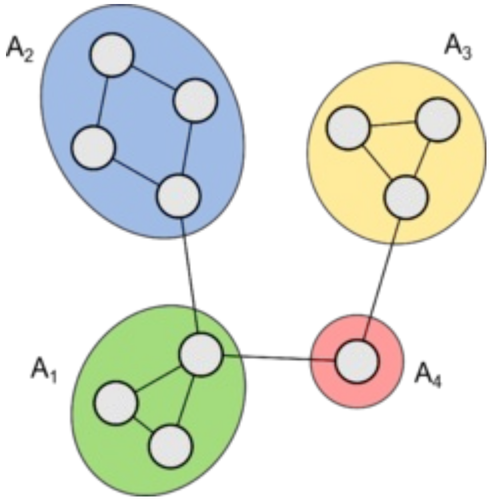
Вершины u и v в неориентированном графе G **реберно двусвязны**, если между этими вершинами существуют два реберно непересекающихся пути.

Реберная двусвязность – отношение эквивалентности.



Реберная двусвязность

Вершины неориентированного графа разбиваются на **компоненты реберной двусвязности**.



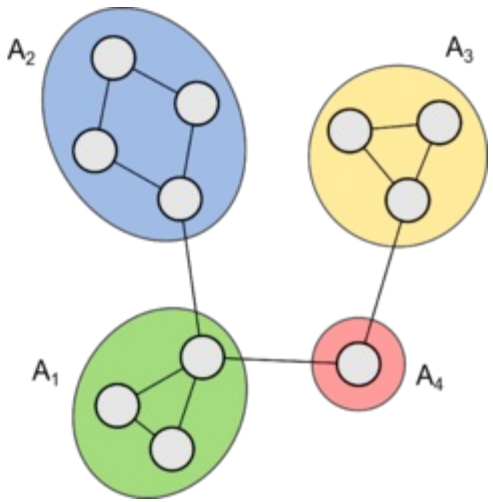
Мост

Мост – ребро, связывающее две различные компоненты реберной двусвязности.

Мост – ребро, при удалении которого компонента связности распадается.

Мост – ребро (u, v), лежащее на любом пути, соединяющем u и v.

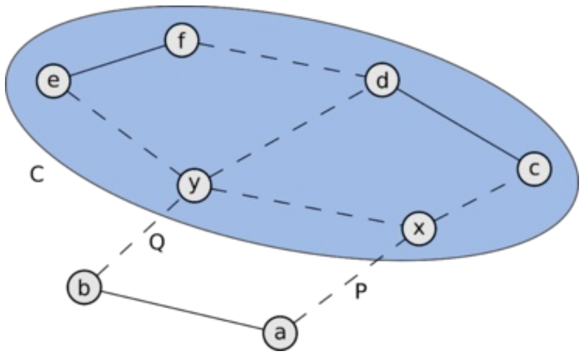
Все три определения эквивалентны.



Вершинная двусвязность

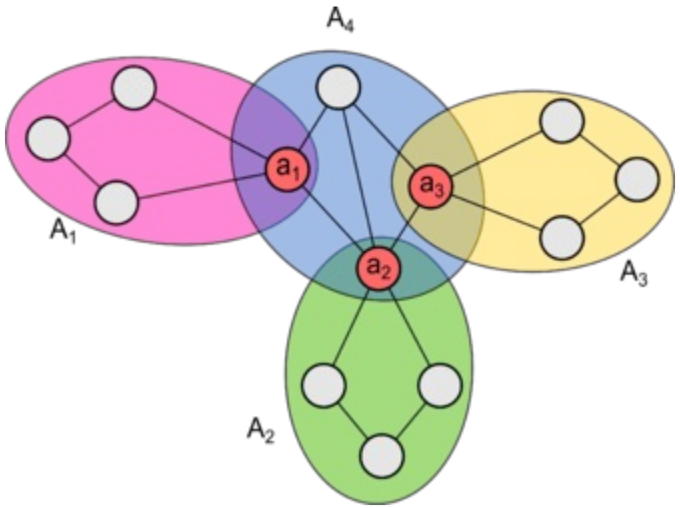
Два ребра в неориентированном графе G **вершинно двусвязны**, если существуют вершинно непересекающиеся пути, соединяющие их концы.

Вершинная двусвязность – отношение эквивалентности на ребрах.



Вершинная двусвязность

Ребра неориентированного графа разбиваются на **компоненты вершинной двусвязности**.

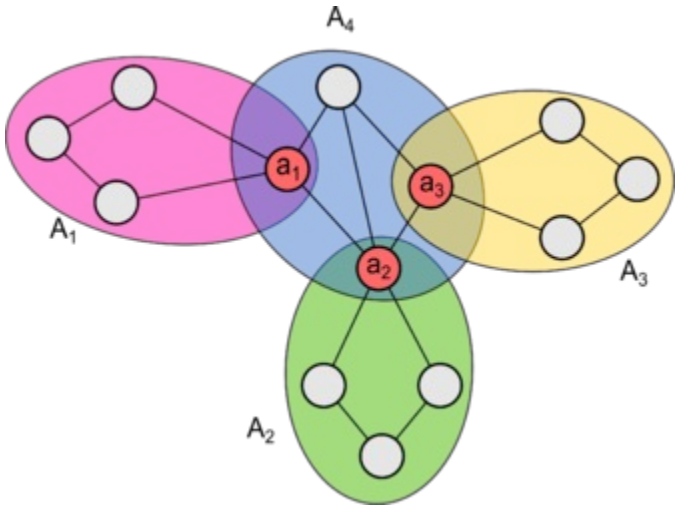


Точка сочленения

Точка сочленения – вершина, при удалении которой вместе с инцидентными ей ребрами, компонента связности распадается.

Точка сочленения – вершина, инцидентная рёбрам, принадлежащим двум или более компонентам вершинной двусвязности.

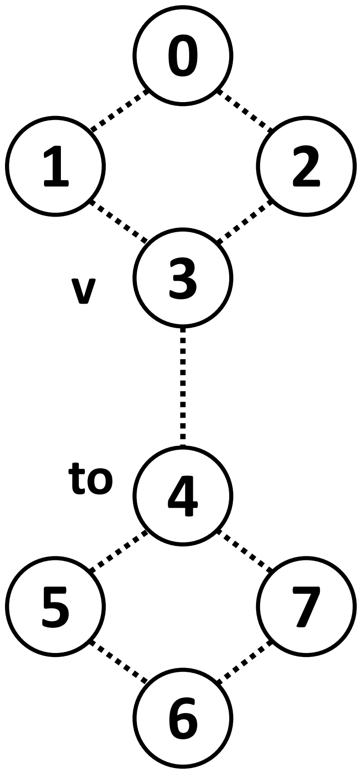
Эти определения эквивалентны.



Поиск мостов

Если из вершины **to** или ее потомком нет ребра в вершину **v** или в ее родителей, то ребро **(v, to)** – мост.

Мы этим условием проверяем, нет ли другого пути из **v** в **to**, кроме как спуск по ребру **(v, to)** дерева обхода в глубину.



Поиск мостов

Проход по дереву поиска в обратную сторону
Пропускаем ребро

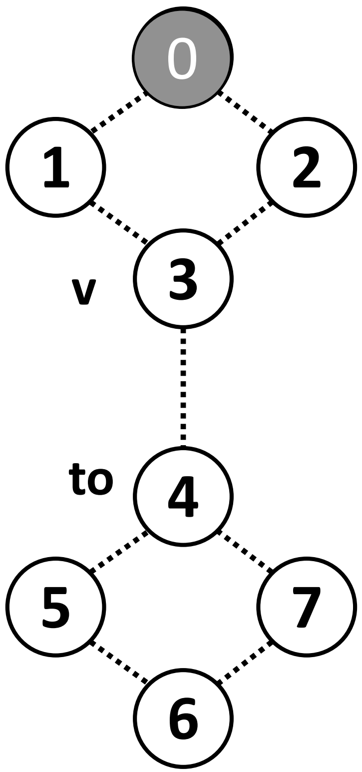
Критерий обратного ребра
Берем минимальное время захода

Ребро дерева поиска
Обрабатываем всех потомков вершины to
Берем минимальное время захода из всех потомков to

Критерий моста
Если время входа в to и в любого из её потомком больше времени входа в v.

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p )
            continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to]);
            if( lowest[to] > entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}
void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++I )
        if( !used[i] )
            dfs( i );
}
```

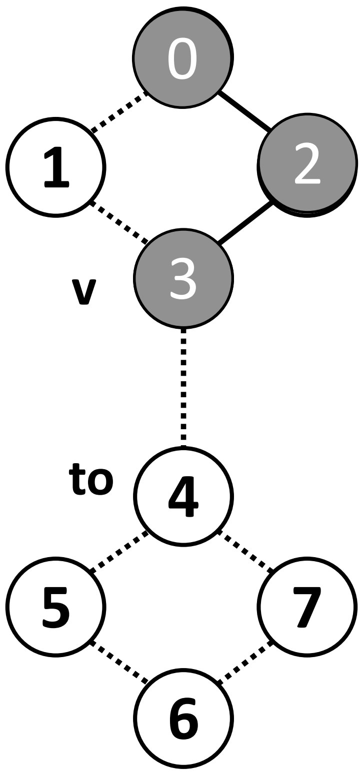
Поиск мостов



	used	entry	lowest
0	1	0	0
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p ) continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to]);
            if( lowest[to] > entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}
void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++I )
        if( !used[i] )
            dfs( i );
}
```

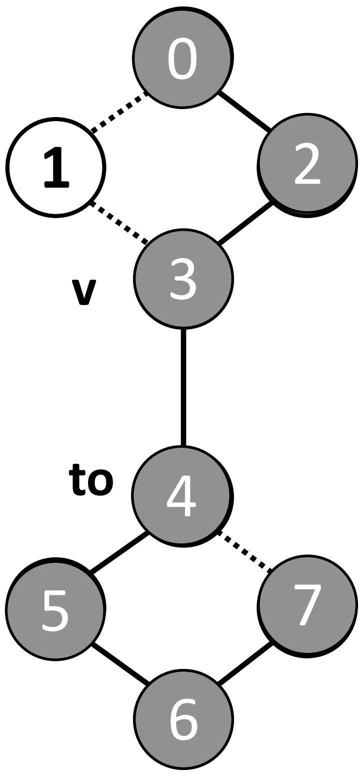
Поиск мостов



	used	entry	lowest
0	1	0	0
1	0		
2	1	1	1
3	1	2	2
4	0		
5	0		
6	0		
7	0		

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p ) continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to]);
            if( lowest[to] > entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}
void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++I )
        if( !used[i] )
            dfs( i );
}
```

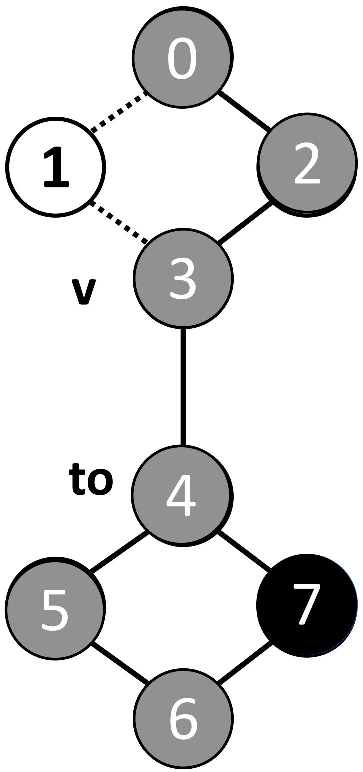
Поиск мостов



	used	entry	lowest
0	1	0	0
1	0		
2	1	1	1
3	1	2	2
4	1	3	3
5	1	4	4
6	1	5	5
7	1	6	3

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p ) continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to]);
            if( lowest[to] > entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}
void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++I )
        if( !used[i] )
            dfs( i );
}
```

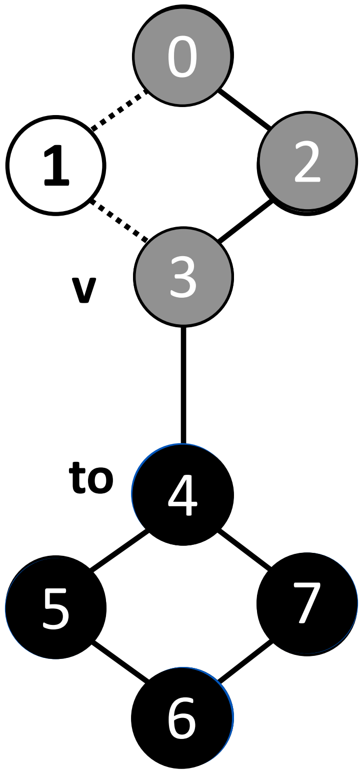
Поиск мостов



	used		entry		lowest
0	1	0	0	0	0
1	0	1		1	
2	1	2	1	2	1
3	1	3	2	3	2
4	1	4	3	4	3
5	1	5	4	5	4
6	1	6	5	6	3
7	1	7	6	7	3

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p ) continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to]);
            if( lowest[to] > entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}
void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++I )
        if( !used[i] )
            dfs( i );
}
```

Поиск мостов



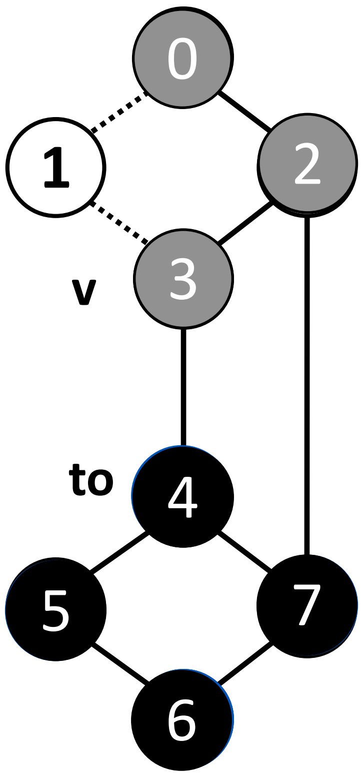
	used	entry	lowest
0	1	0	0
1	0		
2	1	1	1
3	1	2	2
4	1	3	3
5	1	4	3
6	1	4	3
7	1	5	3

```
lowest[to] = 3
entry[v] = 2
lowest[to] > entry[v]
```

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p ) continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to]);
            if( lowest[to] > entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}
void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++I )
        if( !used[i] )
            dfs( i );
}
```

Мост найден

Поиск мостов



	used	entry	lowest
0	1	0	0
1	0		
2	1	1	1
3	1	2	2
4	1	3	1
5	1	4	1
6	1	5	1
7	1	6	1

```
lowest[to] = 1
entry[v] = 2
lowest[to] < entry[v]
```

Моста нет

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p ) continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to]);
            if( lowest[to] > entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}
void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++I )
        if( !used[i] )
            dfs( i );
}
```

Поиск точек сочленения

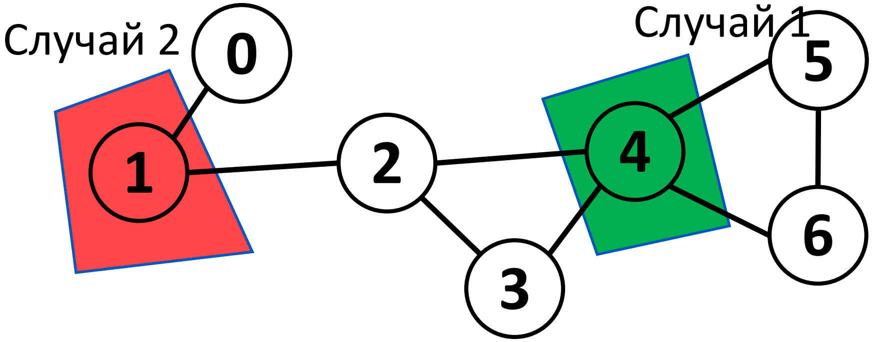
Рассматриваются два случая:

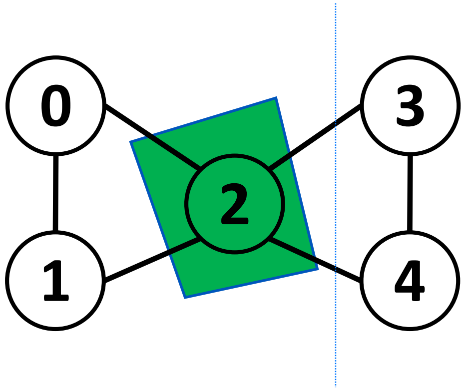
1) Вершина **v** не корень. Если из вершины **to** или ее потомком нет ребра в *родителей* **v**, то вершина **v** – **точка сочленения**.

2) Вершина **v** – корень. Тогда она является **точкой сочленения**, если в дереве обхода в глубину имеет более одного потомка.

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    int children = 0;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p )
            continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to]);
            if( lowest[to] >= entry[v] && p != -1 )
                IS_CUTPOINT( v, to );
            children++;
        }
    }
    if( children > 1 && p == -1 )
        IS_CUTPOINT( v );

}
void find_cutpoints() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++I )
        if( !used[i] )
            dfs( i );
}
```

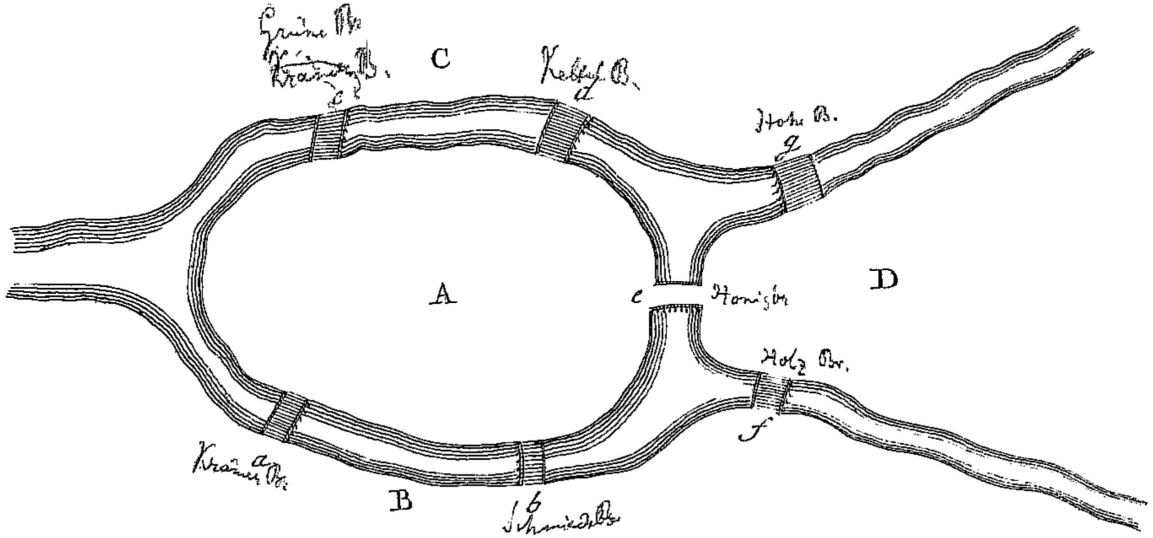




Эйлеровы графы

Задача Эйлера: Найти путь (цикл), проходящий по всем ребрам графа один раз.

Кёнигсбергские мосты



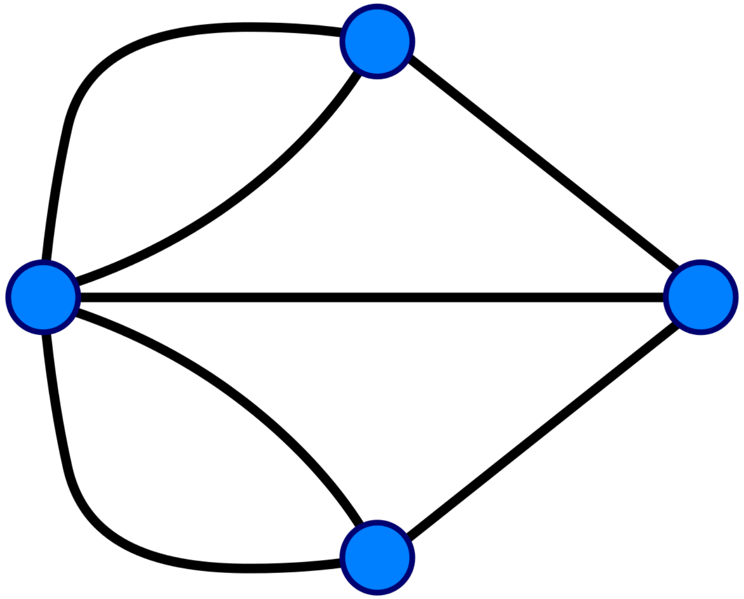
Эйлеровы графы

Эйлеров путь – это путь, проходящий по всем ребрам графа, притом по одному разу.

Эйлеров цикл – это замкнутый эйлеров путь.

Эйлеров граф – это граф, содержащий эйлеров цикл.

Полуэйлеров граф – это граф, содержащий эйлеров путь.



G – связный неориентированный граф.

- Эйлеров путь существует тогда и только тогда, когда в G не более двух нечетных вершин и они являются началом и концом пути.
- Эйлеров цикл существует тогда и только тогда, когда в G все вершины четные.

Эйлеровы графы

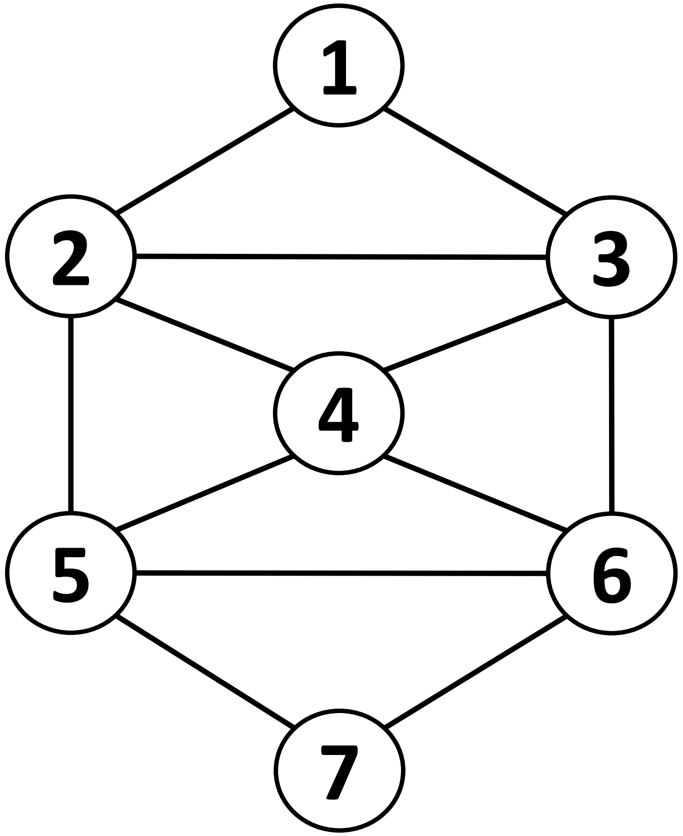
Поиск эйлерова пути

Алгоритм запускаем из вершины с нечетной степенью

Перебираем все ребра, выходящие из V

- Удаляем это ребро из графа
- Вызываем п.1 для второго конца этого ребра

Добавляем V в ответ



Вопросы?

Спасибо за внимание!