Гамильтонов и Эйлеров циклы. Раскраска графа. Мосты и точки сочленения

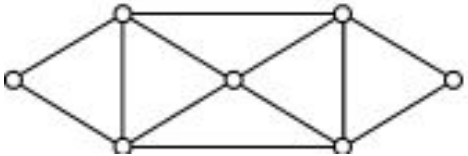
Эйлеровость графов

Вспомним немного теории из дискретной математики:

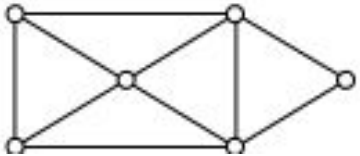
Эйлеров путь в графе – путь, который проходит по каждому ребру ровно **1** раз. Граф, в котором существует эйлеров путь, но не существует эйлерова цикла, называется **полуэйлеровым графом**.

Эйлеров цикл – замкнутый эйлеров путь. Граф, в котором существует эйлеров цикл, называется **эйлеровым графом**.

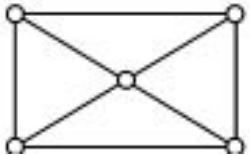
Эйлеров граф



Полуэйлеров граф



Не эйлеров граф



Эйлеровость графов

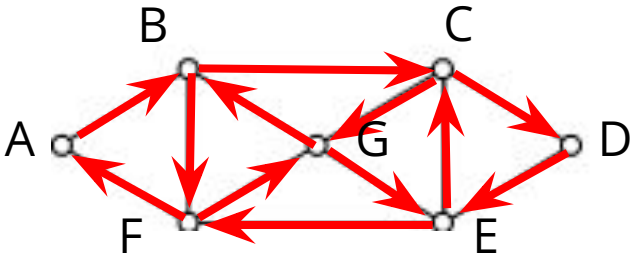
Вспомним немного теории из дискретной математики:

Эйлеров путь в графе – путь, который проходит по каждому ребру ровно **1** раз. Граф, в котором существует эйлеров путь, но не существует эйлерова цикла, называется **полуэйлеровым графом**.

Эйлеров цикл – замкнутый эйлеров путь. Граф, в котором существует эйлеров цикл, называется **эйлеровым графом**.

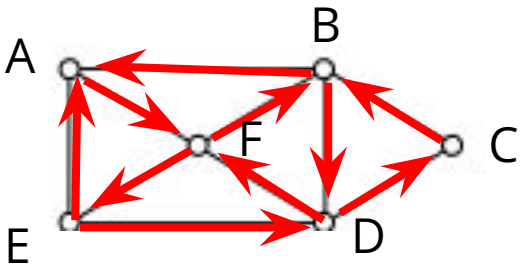
ABFGBCGECDEFA

Эйлеров граф

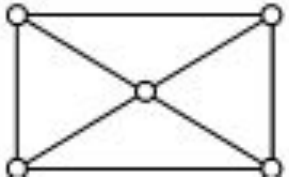


EAFEDFBDCBA

Полуэйлеров граф



Не эйлеров граф

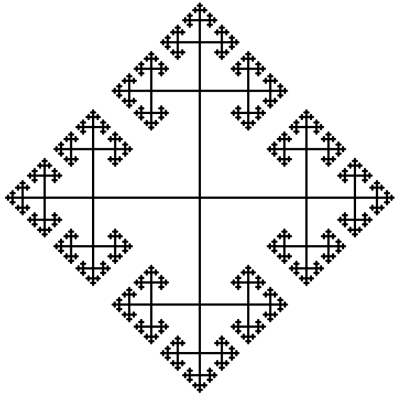


Критерий Эйлеровости

Теорема:

Для того, чтобы граф был **эйлеровым**, необходимо и достаточно, чтобы:

- Все вершины имели четную степень
- Все компоненты связности, кроме, может быть, одной, не содержали ребер



Фрактальный эйлеров граф

Доказательство необходимости:

- Допустим в графе существует вершина с нечетной степенью. Рассмотрим эйлеров обход графа. Заметим, что при попадании в вершину и при выходе из нее мы уменьшаем ее степень на два (помечаем уже пройденные ребра), если эта вершина не является стартовой(она же конечная для цикла). Для стартовой (конечной) вершины мы уменьшаем ее степень на один в начале обхода эйлерова цикла, и на один при завершении. Следовательно вершин с нечетной степенью быть не может. Наше предположение неверно.
- Если в графе существует более одной компоненты связности с ребрами, то очевидно, что нельзя пройти по их ребрам одним путем.

Доказательство достаточности остается в качестве упражнения читателю)

Критерий Эйлеровости

Теорема:

Для того, чтобы граф был **полуэйлеровым**, необходимо и достаточно, чтобы:

- В графе было не более **2** вершин, степень которых нечетна
- Все компоненты связности, кроме, может быть, одной, не содержали ребер

Доказательство примерно аналогично случаю эйлерова графа

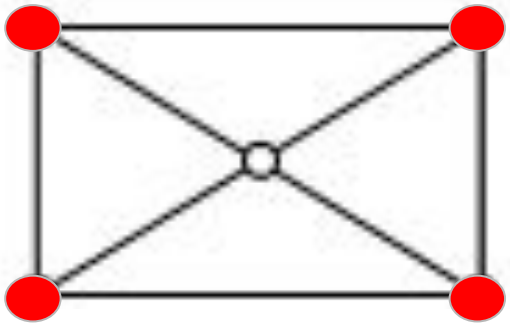
Проверка на (полу)эйлеровость

Очень напоминает dfs, однако вместо пометок вершин мы помечаем ребра

```
boolean checkForEulerPath():
    int OddVertex = 0
    for v : v ∈ V
        if deg(v) mod 2 == 1
            OddVertex++
    if OddVertex > 2// если количество вершин с нечетной степенью больше двух, то граф не является эйлеровым
        return false
    boolean visited(|V|, false) // массив инициализируется значениями false
    for v : v ∈ V
        if deg(v) > 0
            dfs(v, visited)
            break
    for v : v ∈ V
        if deg(v) > 0 and not visited[v]   // если количество компонент связности, содержащие ребра, больше одной,
            return false                   // то граф не является эйлеровым
    return true   // граф является эйлеровым
```

Проверка на (полу)эйлеровость

```
boolean checkForEulerPath():
    int OddVertex = 0
    for v : v ∈ V
        if deg(v) mod 2 == 1
            OddVertex++
    if OddVertex > 2// если количество вершин с нечетной степенью больше двух, то граф не является эйлеровым
        return false
```



В таком графе вершин с нечетной степенью **4**, что больше, чем **2**, поэтому алгоритм вернет **false**

Проверка на (полу)эйлеровость

```
boolean visited(|V|, false) // массив инициализируется значениями false
for v : v ∈ V
    if deg(v) > 0
        dfs(v, visited)
        break
for v : v ∈ V
    if deg(v) > 0 and not visited[v]   // если количество компонент связности, содержащие ребра, больше одной,
        return false                   // то граф не является эйлеровым
return true   // граф является эйлеровым
```

Оставшаяся часть кода связана только с проверкой на наличие нескольких компонент связности

Построение эйлерова пути (общий вид)

```
function findEulerPath(v : Vertex):
    for (v, u) ∈ E
        remove (v, u)
        findEulerPath(u)
    print(v)
```

- Если реализовать поиск ребер, инцидентных вершине, и удаление ребер за **O(1)**, то алгоритм будет работать за **O(E)**.
- Чтобы реализовать поиск за **O(1)**, для хранения графа следует использовать списки смежных вершин, а для удаления достаточно добавить всем ребрам **boolean флаг deleted**.

Построение эйлерова цикла

Для построения эйлерова цикла необходимо сначала проверить граф на (полу)эйлеровость, как мы делали это ранее

```
function findEulerPath(v):   // если граф является полуэйлеровым, то алгоритм следует запускать из вершины нечетной степени
   for u : u ∈ V
       if deg(u) mod 2 == 1
           v = u
           break
```

Далее следует посмотреть, есть ли в графе вершины с нечетной степенью.

- Если да, значит, **граф полуэйлеров**, и запускать обход необходимо из вершины с нечетной степенью
- Если нет, значит, **граф эйлеров**, и обход можно запускать из любой вершины

Построение эйлерова цикла

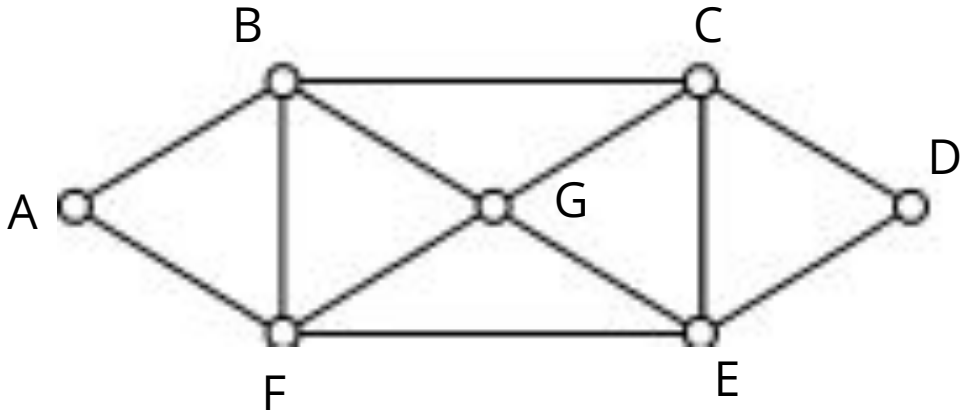
Далее ходим по ребрам, удаляя посещенные из списка ребер. Приходя в вершину, будем закидывать ее в стек. Если у вершины остались инцидентные ей непосещенные ребра, пойдем в них, а если нет - добавим вершину в ответ и удалим ее из стека.

```
S.push(v)  // S – стек
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Изначально ни одно ребро не посещено. Начнем наш обход, допустим, из вершины **A**



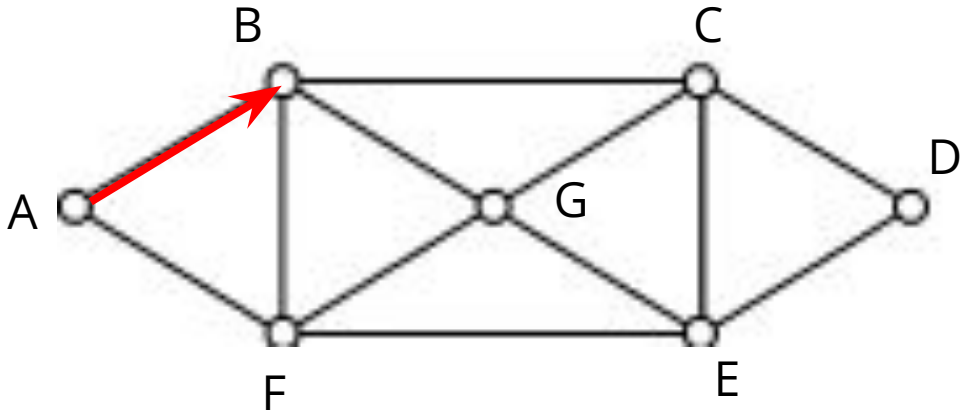
Стек: A

Ответ:

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Посетили ребро **AB**



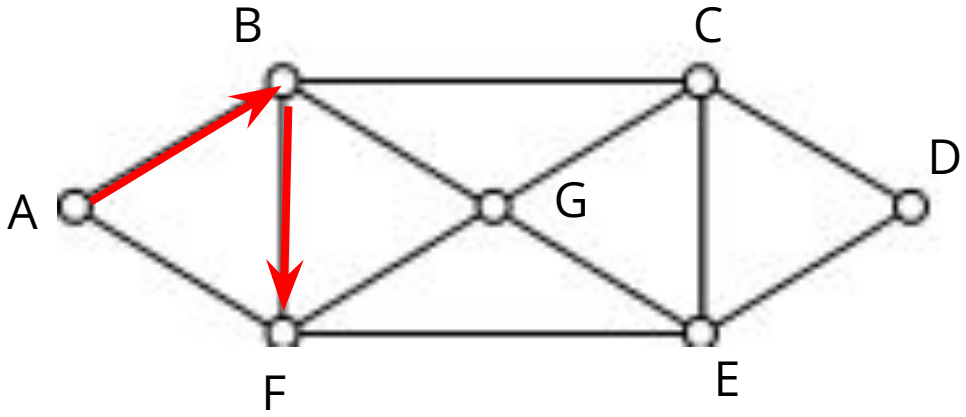
Стек: AB

Ответ:

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Посетили ребро **BF**



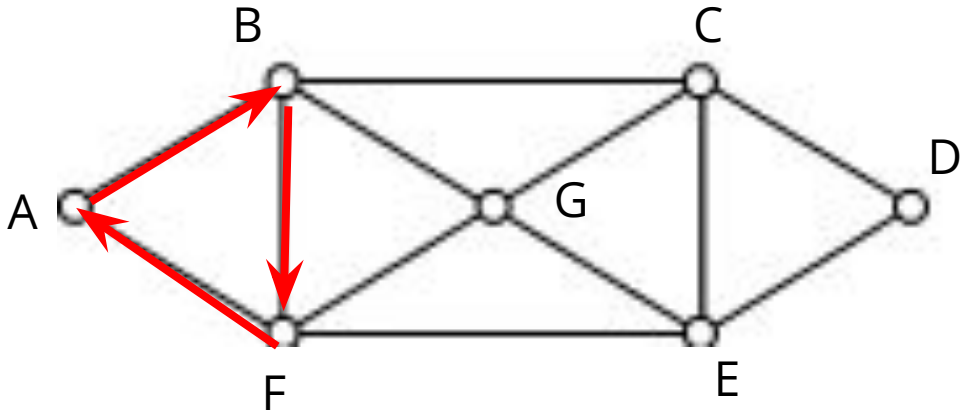
Стек: ABF

Ответ:

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Посетили ребро **FA**



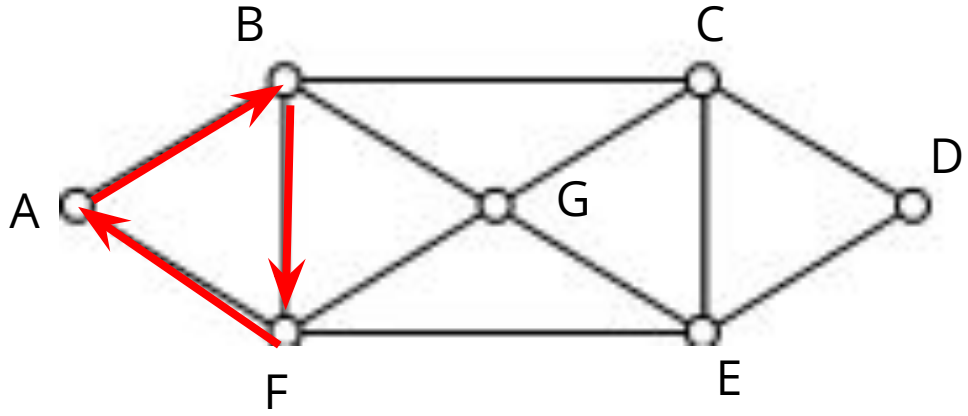
Стек: ABFA

Ответ:

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

У вершины **A** не осталось непосещенных инцидентных ей ребер. Добавляем ее в ответ.



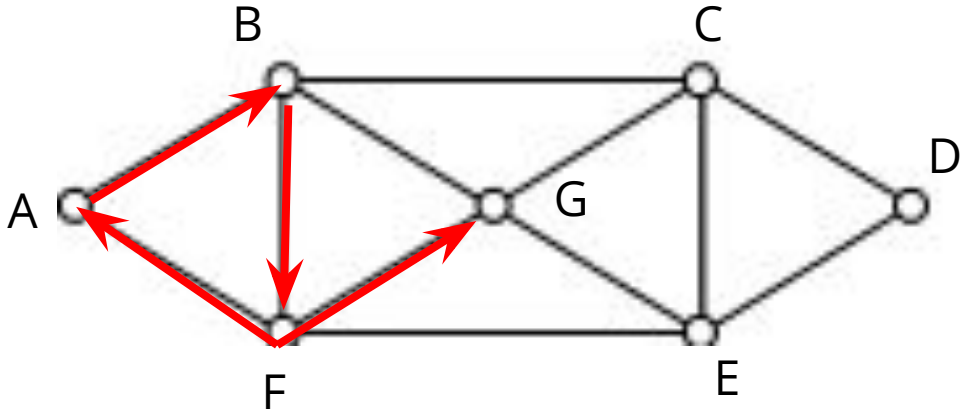
Стек: ABF

Ответ: A

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Возвращаемся в вершину **F**. У нее есть непосещенные инцидентные ребра. Посетили ребро **FG**



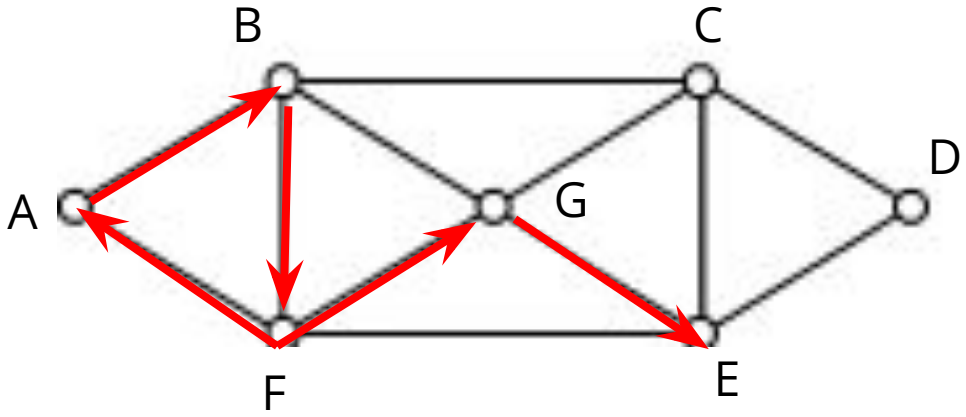
Стек: ABFG

Ответ: A

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Посетили ребро **GE**



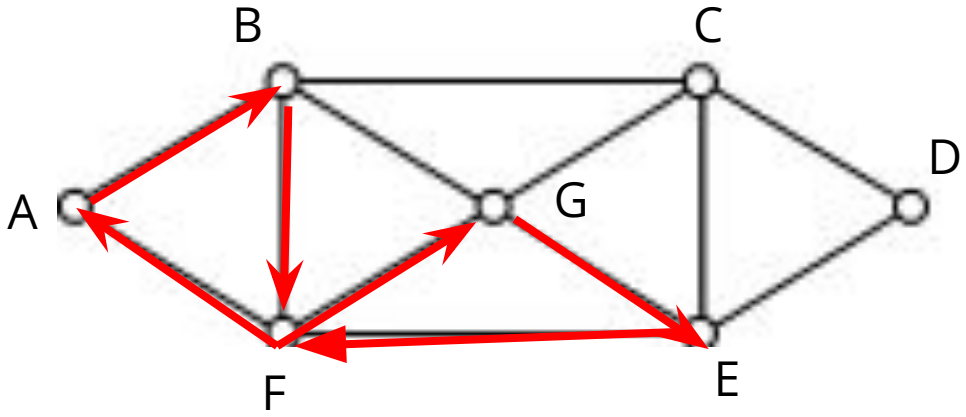
Стек: ABFGE

Ответ: A

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Посетили ребро **EF**



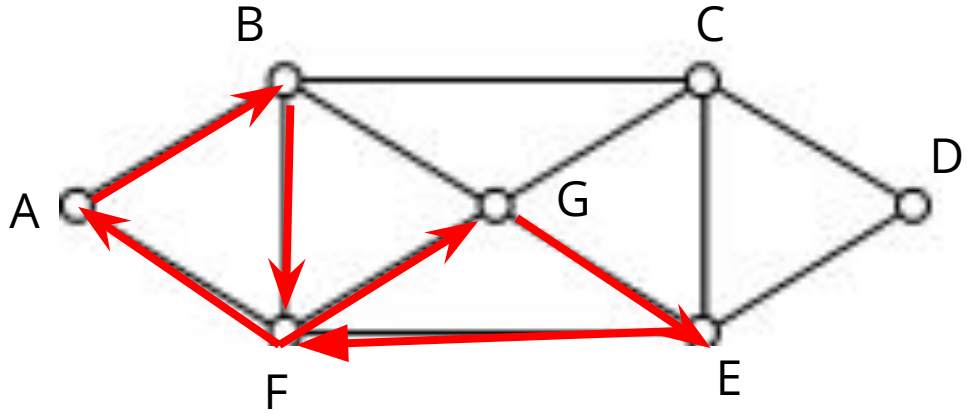
Стек: ABFGEF

Ответ: A

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

У вершины **F** не осталось непосещенных инцидентных ей ребер. Добавляем ее в ответ.



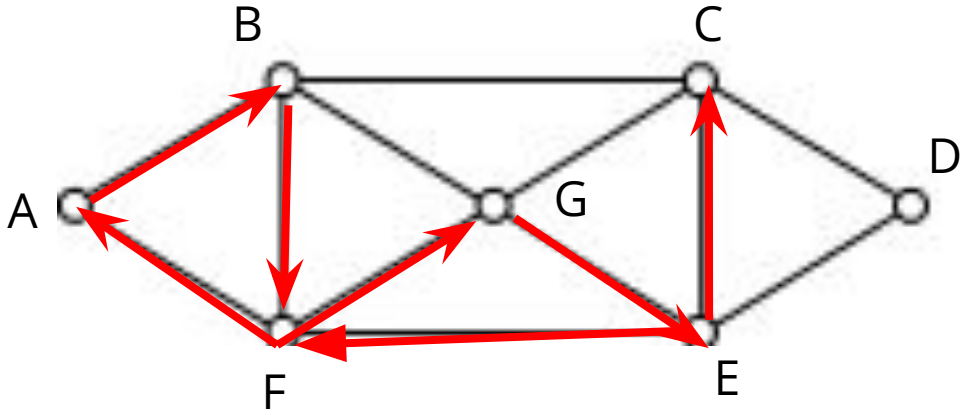
Стек: ABFGE

Ответ: AF

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Возвращаемся в вершину **E**. У нее есть непосещенные инцидентные ребра. Посетили ребро **EC**



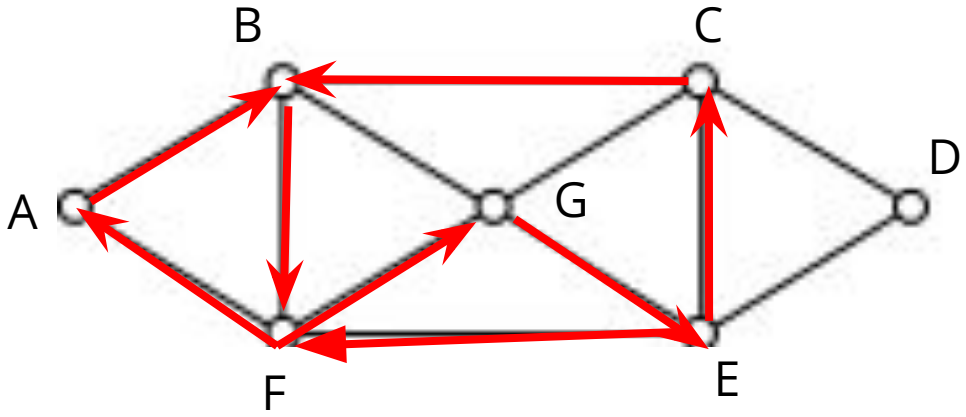
Стек: ABFGEC

Ответ: AF

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Посетили ребро **CB**



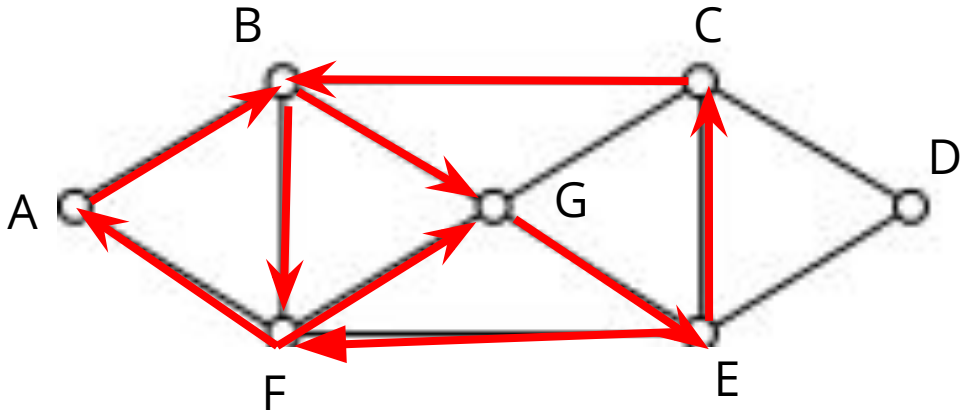
Стек: ABFGECB

Ответ: AF

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Посетили ребро **BG**



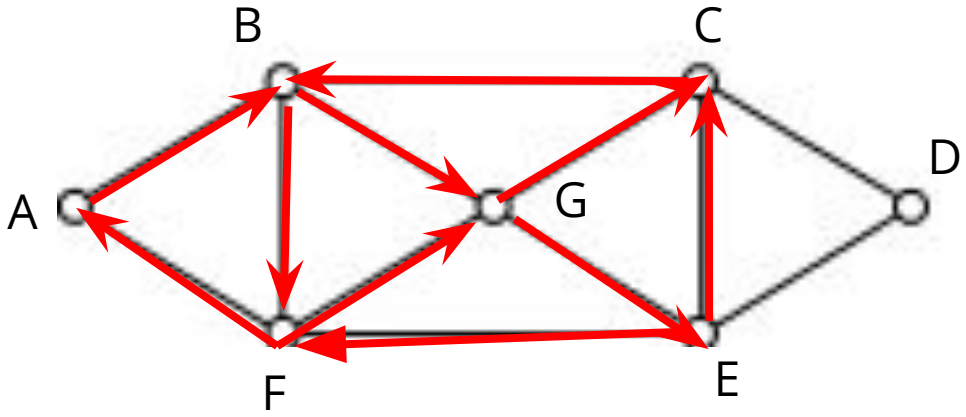
Стек: ABFGECBG

Ответ: AF

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Посетили ребро **GC**



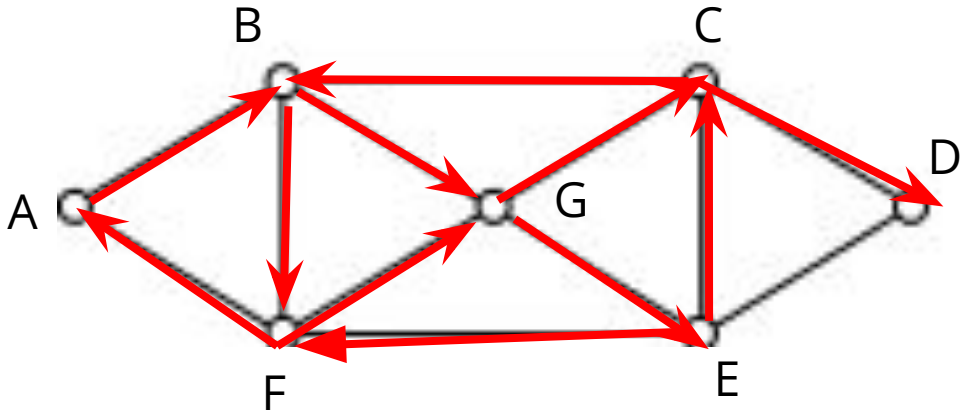
Стек: ABFGECBGC

Ответ: AF

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Посетили ребро **CD**



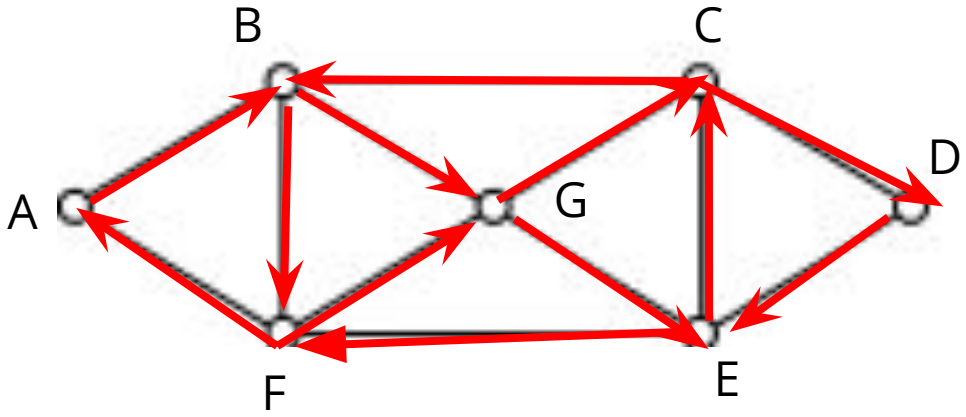
Стек: ABFGECBGCD

Ответ: AF

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Посетили ребро **DE**



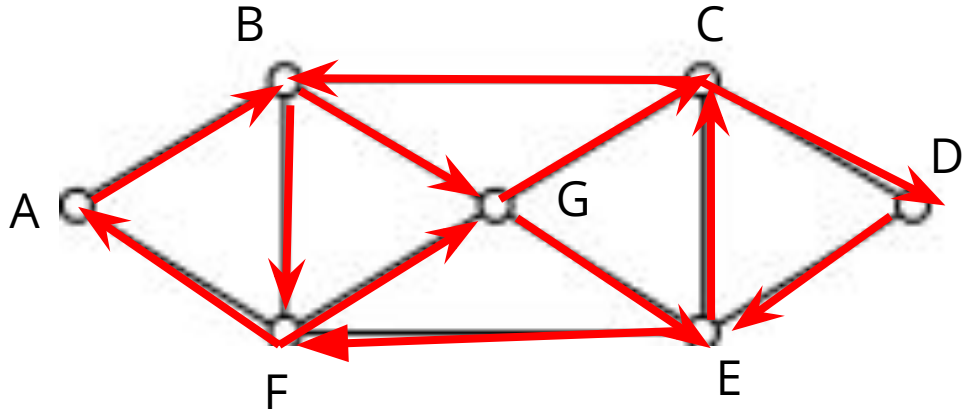
Стек: ABFGECBGCDE

Ответ: AF

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Так как непосещенных ребер не осталось, весь стек просто запишется в ответ в обратном порядке



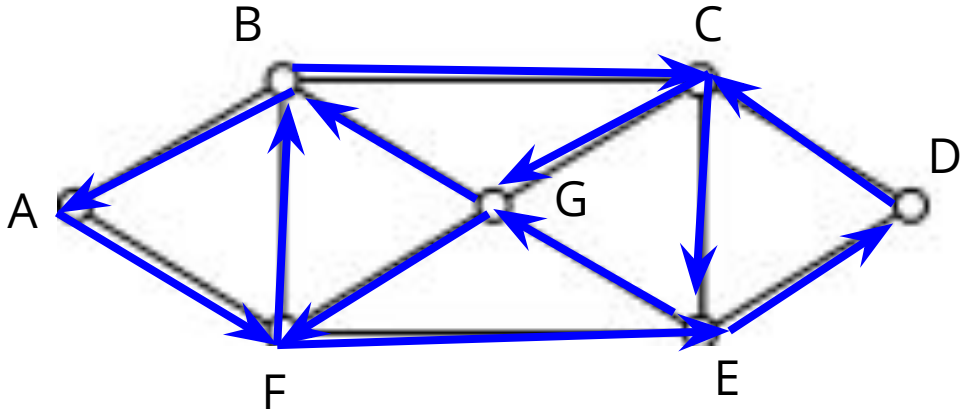
Стек:

Ответ: AFEDCGBCEGFBA

Построение эйлерова цикла

```
while not S.empty()
    w = S.top()
    found_edge = False
    for u : u ∈ V
        if (w, u) ∈ E  // нашли ребро, по которому ещё не прошли
            S.push(u)  // добавили новую вершину в стек
            E.remove(w, u)
            found_edge = True
            break
    if not found_edge
        S.pop()  // не нашлось инцидентных вершине w рёбер, по которым ещё не прошли
        print(w)
```

Получившийся эйлеров путь (который является эйлеровым циклом, потому что степени всех вершин четны):



Стек:

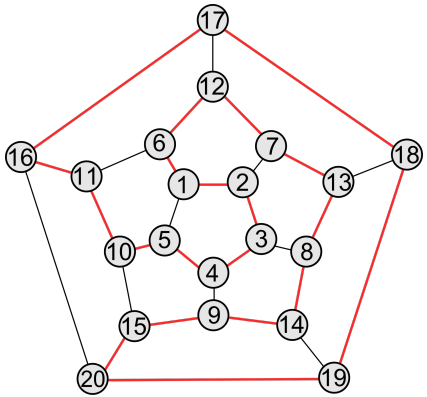
Ответ: AFEDCGBCEGFBA

Гамильтоновость графов

Вспомним немного теории из дискретной математики:

Гамильтонов путь в графе – путь, который проходит по каждой вершине ровно **1** раз. Граф, в котором существует гамильтонов путь, но не существует гамильтонова цикла, называется **полугамильтоновым графом**.

Гамильтонов цикл – замкнутый гамильтонов путь. Граф, в котором существует гамильтонов цикл, называется **гамильтоновым графом**.



Достаточные условия Гамильтоновости графа

Теорема Дирака:
Если **n >= 3** и **deg v >= n/2** для любой вершины **v** неориентированного графа **G** , то **G** — гамильтонов граф.

Теорема Оре:
Если **n >= 3** и **deg u + deg v >= n** для любых двух различных несмежных вершин *u* и *v* неориентированного графа **G**, то **G** — гамильтонов граф.

Теорема Гуйя-Ури:
Пусть **G**— сильносвязный ориентированный граф.
$\mathbf{deg_{in}(v) >= n/2}$ }
$\mathbf{deg_{out}(v) >= n/2}$ } ⇒ **G** — гамильтонов.

Задача о гамильтоновом пути

Задача о гамильтоновом пути и **задача о гамильтоновом цикле** — это задачи определения, существует ли гамильтонов путь или гамильтонов цикл в заданном графе (ориентированном или неориентированном). Обе задачи **NP-полны**.

Def: Вычислительная задача называется **NP-полной** (от англ. non-deterministic polynomial – «недетерминированные с полиномиальным временем»), если для неё не существует эффективных алгоритмов решения.

Существует несколько решений данных задач.

Наивная реализация за O(N*N!)

Самый простой подход к решению поставленной задачи - сгенерировать все возможные перестановки из **N** вершин, их количество равняется **N!**. Для каждой перестановки исследовать, является ли она правильным гамильтоновым путем, проверяя, есть ли ребро между соседними вершинами или нет. Оценка по времени **O(N*N!)**, оценка по памяти **O(1)**

```
int[n] randomPermutation(a: int[n]):   // n — длина перестановки
  for i = n downto 1
    j = random(1..i)
    swap(a[i], a[j])
  return a
```

Реализация через динамику за $O(N^2 * 2^N)$

Эффективный подход: Приведенный выше подход может быть оптимизирован с помощью динамического программирования и битовых масок, он основан на следующих наблюдениях:

- Идея заключается в том, что для каждого подмножества **S** вершин проверяется, существует ли гамильтонов путь в подмножестве **S**, который заканчивается в вершине **v**, где **v ∈ S**.
- Если у **v** есть сосед **u**, где **u ∈ S**, следовательно, существует гамильтонов путь, который заканчивается в вершине **u**.
- Эту задачу можно решить, обобщив подмножество вершин и конечную вершину гамильтонова пути.

Реализация через динамику $O(N^2 * 2^N)$

Шаги решения:

- Инициализируем булеву матрицу **dp[][]** размерности **N*2^N**, где **dp[j][i]** показывает, существует ли в подмножестве путь, представленный маской **i**, который посещает каждую вершину в **i** один раз и заканчивается в вершине **j**.

- Для базового случая обновляем **dp[i][1 << i] = true**, для **i** в диапазоне **[0, N - 1]**

```
bool Hamiltonian_path(
    vector<vector<int> >& adj, int N)
{
    int dp[N][(1 << N)];

    // Initialize the table
    memset(dp, 0, sizeof dp);

    // Set all dp[i][(1 << i)] to
    // true
    for (int i = 0; i < N; i++)
        dp[i][(1 << i)] = true;
```

Во внешнем цикле происходит проход по диапазону **[1, 2^N - 1]** с использованием переменной **i** и выполнение следующих шагов:

- Все вершины с битами, установленными в маске **i**, включаются в подмножество.
- Итерация по диапазону **[1, N]** с использованием переменной **j**, которая будет представлять конечную вершину гамильтонова пути текущего подмножества маски **i**.

```
// Iterate over each subset
// of nodes
for (int i = 0; i < (1 << N); i++) {

    for (int j = 0; j < N; j++) {

        // If the jth nodes is included
        // in the current subset
        if (i & (1 << j)) {

            // Find K, neighbour of j
            // also present in the
            // current subset
            for (int k = 0; k < N; k++) {

                if (i & (1 << k)
                    && adj[k][j]
                    && j != k
                    && dp[k][i ^ (1 << j)]) {

                    // Update dp[j][i]
                    // to true
                    dp[j][i] = true;
                    break;
                }
            }
        }
    }
}
```

Во внутреннем цикле цикле происходит проход по диапазону **[1, N]** с использованием переменной **j** и выполнение следующих шагов:

- Если значение **i** и 2^j истинно, то пройдитесь по диапазону **[1, N]**, используя переменную **k**, и если значение **dp[k][i^2^j]** истинно, то отметьте **dp[j][i]** как истинное и выйдите из цикла.
- В противном случае перейдите к следующей итерации.

```
// Iterate over each subset
// of nodes
for (int i = 0; i < (1 << N); i++) {

    for (int j = 0; j < N; j++) {

        // If the jth nodes is included
        // in the current subset
        if (i & (1 << j)) {

            // Find K, neighbour of j
            // also present in the
            // current subset
            for (int k = 0; k < N; k++) {

                if (i & (1 << k)
                    && adj[k][j]
                    && j != k
                    && dp[k][i ^ (1 << j)]) {

                    // Update dp[j][i]
                    // to true
                    dp[j][i] = true;
                    break;
                }
            }
        }
    }
}
```

Реализация через динамику $O(N^2 * 2^N)$

Проход по массиву с использованием переменной **i**, и если значение **dp[i][2^N - 1]** истинно, то существует гамильтонов путь, заканчивающийся в вершине **i**.

```
// Traverse the vertices
for (int i = 0; i < N; i++) {

    // Hamiltonian Path exists
    if (dp[i][(1 << N) - 1])
        return true;
}

// Otherwise, return false
return false;
```

Поиск Гамильтонова цикла в условиях выполнения достаточного условия

Поступим следующим образом: заведем очередь и положим в нее все вершины нашего графа (не важно в каком порядке). Пусть **n =|V|**. Тогда **n(n−1)** раз будем делать следующую операцию:

- Пусть $\mathbf{v_1}$ — это голова очереди, $\mathbf{v_2}$ — следующая за ней вершина и так далее. Если между первой и второй вершиной в очереди есть ребро в графе **G**, то перемещаем первую вершину в конец очереди и переходим к следующей итерации.

Поиск Гамильтонова цикла в условиях выполнения достаточного условия

- Если между первой и второй вершинами в очереди ребра нет, то найдем такую вершину $\mathbf{v_i}$, где $\mathbf{i > 2}$, такую, что ребра $\mathbf{(v_1, v_i)}$, $\mathbf{(v_2, v_{i+1}) \in E}$ (в условиях т Оре или т Дирака такая вершина обязательно найдется). После этого поменяем в очереди местами вершины $\mathbf{v_2}$ и $\mathbf{v_i}$, $\mathbf{v_3}$ и $\mathbf{v_{i-1}}$ и так далее, пока $\mathbf{2 + j < i - j}$. Теперь у нас появилось ребро между первыми двумя вершинами в очереди, а также между **i**-той и **(i + 1)**-ой вершинами очереди.

  После этого также, как в первом случае, переносим первую вершину в конец очереди

Поиск Гамильтонова цикла в условиях выполнения достаточного условия

Таким образом после n итераций, мы получаем последовательность (вершины лежащие в очереди), где любые 2 соседние вершины соединены ребром, все вершины графа находятся в этой последовательности, и более того, каждая ровно один раз, а также существует ребро между последней и первой вершинами очереди, а это и значит, что мы решили поставленную задачу.

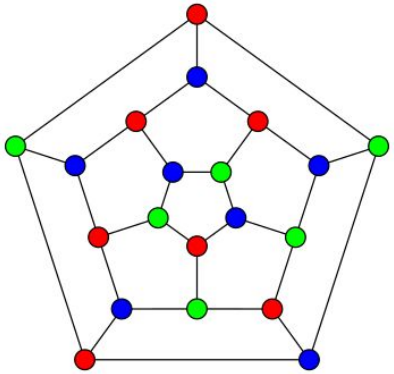
```
function findHamiltonianCycle(⟨V, E⟩):
  for v ∈ V:                                          // Добавляем все вершины графа в очередь
    queue.pushBack(v)
  for k = 0..n * (n - 1)
    if (queue.at(0), queue.at(1)) ∉ E                 // Проверяем существования ребра между первой и второй вершинами очереди
      i = 2
      while (queue.at(0), queue.at(i)) ∉ E or (queue.at(1), queue.at(i + 1)) ∉ E
          i++                                         // Ищем индекс удовлетворяющую условию вершины
      queue.swapSubQueue(1, i)                        // Разворачиваем часть перестановки от 1-й до найденной позиции включительно
    queue.pushBack(queue.top())
    queue.pop()
```

Раскраска графа

Вспомним немного теории из дискретной математики:

Правильной раскраской графа **G(V,E)** называется такое отображение **φ** из множества вершин **V** в множество красок **{c_1...c_t}**, что для любых двух смежных вершин **u** и **v** выполняется **φ(u)≠φ(v)** .
Так же её называют **t-раскраской**.

Хроматическим числом (англ. Chromatic number) **χ(G)** графа **G(V,E)** называется такое минимальное число **t** , для которого существует
t-раскраска графа.



Жадный алгоритм раскраски графа

1. Раскрасим первую вершину первым цветом.
2. Сделаем следующие для оставшихся **V-1** вершин:

Рассмотрим текущую выбранную вершину и закрасим ее цветом с наименьшим номером, который не был использован ни в одной из вершинах, смежных с ней. Если все ранее использованные цвета появляются на вершинах, смежных с **v**, присвоим ей новый цвет.

```
int *result = new int[V];

// Назначаем первый цвет первой вершине
result[0] = 0;

// Инициализируем оставшиеся вершины V-1 как неокрашенные
for (int u = 1; u < V; u++)
    result[u] = -1; // для вершины u цвет не назначен

// Временный массив для хранения доступных цветов.
// значение True  в available[cr] будет означать, что цвет cr является
// назначен одной из смежных вершин
bool *available = new bool[V];
for (int cr = 0; cr < V; cr++)
    available[cr] = false;
```

Жадный алгоритм раскраски графа

1. Раскрасим первую вершину первым цветом.
2. Сделаем следующие для оставшихся **V-1** вершин:

Рассмотрим текущую выбранную вершину и закрасим ее цветом с наименьшим номером, который не был использован ни в одной из вершинах, смежных с ней. Если все ранее использованные цвета появляются на вершинах, смежных с **v**, присвоим ей новый цвет.

```
// Присвоение цветов оставшимся вершинам V-1
for (int u = 1; u < V; u++)
{
    // Обрабатываем все смежные вершины и отмечаем их цвета
    // как недоступные
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
        if (result[*i] != -1)
            available[result[*i]] = true;

    // Находим первый доступный цвет
    int cr;
    for (cr = 0; cr < V; cr++)
        if (available[cr] == false)
            break;

    result[u] = cr; // присваиваем найденный цвет

    //  Сбросьте значения обратно в false для следующей итерации
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
        if (result[*i] != -1)
            available[result[*i]] = false;
}
```

Жадный алгоритм раскраски графа

Оценка по времени:

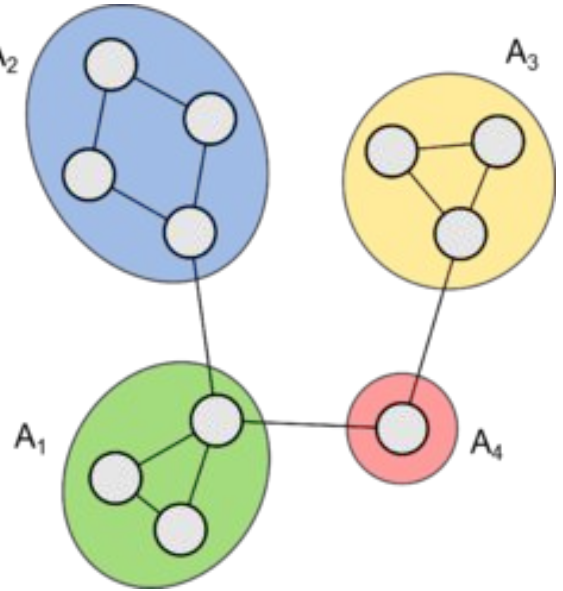
- Лучший случай **O(V^2)**
- Средний случай **O(V^2)**
- Худший случай **O(V^2 + E)**

```
// Назначаем первый цвет первой вершине
result[0] = 0;

// Инициализируем оставшиеся вершины V-1 как неокрашенные
for (int u = 1; u < V; u++)
    result[u] = -1; // для вершины u цвет не назначен

// Временный массив для хранения доступных цветов.
// значение True  в available[cr] будет означать, что цвет cr является
// назначен одной из смежных вершин
bool *available = new bool[V];
for (int cr = 0; cr < V; cr++)
    available[cr] = false;

// Присвоение цветов оставшимся вершинам V-1
for (int u = 1; u < V; u++)
{
    // Обрабатываем все смежные вершины и отмечаем их цвета
    // как недоступные
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
        if (result[*i] != -1)
            available[result[*i]] = true;

    // Находим первый доступный цвет
    int cr;
    for (cr = 0; cr < V; cr++)
        if (available[cr] == false)
            break;

    result[u] = cr; // присваиваем найденный цвет

    //  Сбросьте значения обратно в false для следующей итерации
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
        if (result[*i] != -1)
            available[result[*i]] = false;
}
```

Мосты и точки сочленения

Вспомним немного теории из дискретной математики:

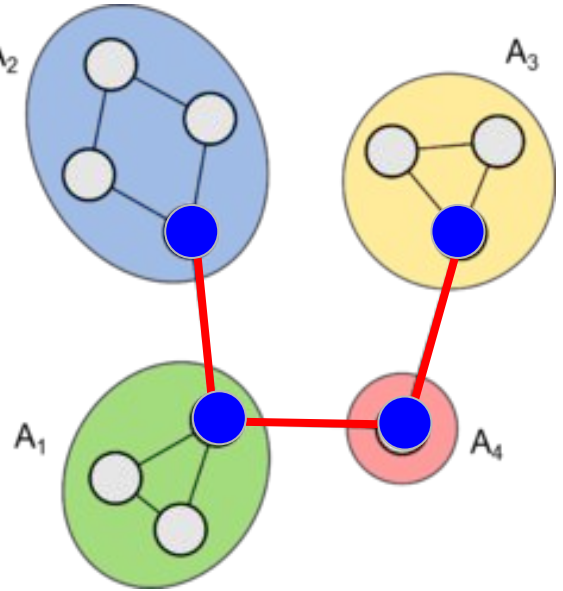
- **Точка сочленения** графа **G** — вершина, при удалении которой вместе с инцидентными ей ребрами в **G** увеличивается число компонент связности.
- Ребро графа называется **мостом**, если его удаление увеличивает число компонент связности графа.



Мосты и точки сочленения

Вспомним немного теории из дискретной математики:

- **Точка сочленения** графа **G** — вершина, при удалении которой вместе с инцидентными ей ребрами в **G** увеличивается число компонент связности.
- Ребро графа называется **мостом**, если его удаление увеличивает число компонент связности графа.

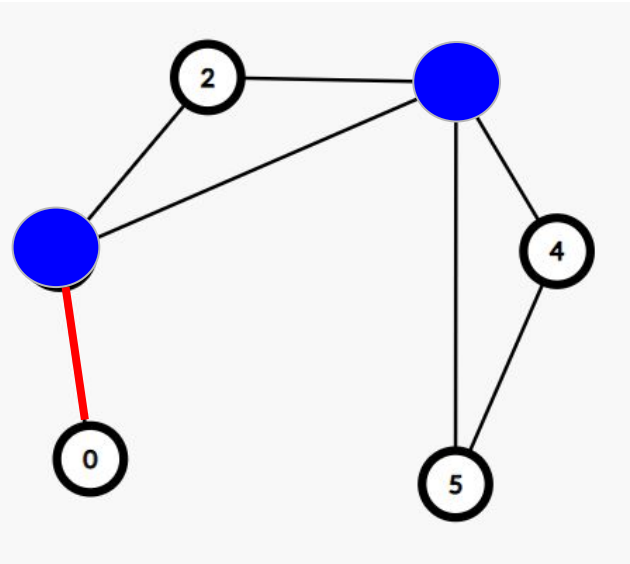


Мосты и точки сочленения

Правда ли, что можно найти все мосты, взять их концы и получить точки сочленения?

Мосты и точки сочленения

Правда ли, что можно найти все мосты, взять их концы и получить точки сочленения?



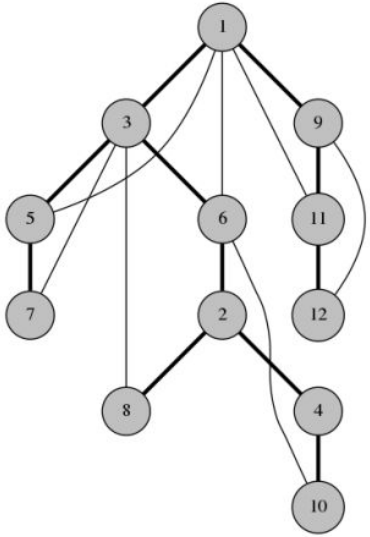
Ответ: нет, неправда

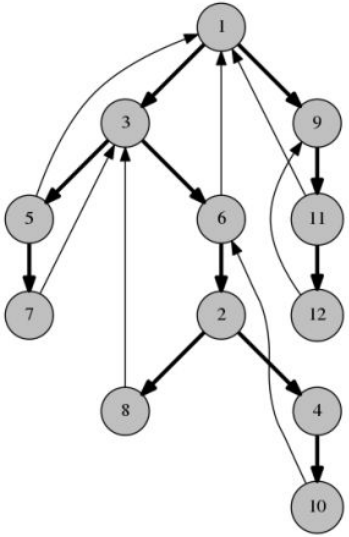
На данном рисунке не все точки сочленения являются концами мостов, и не все концы мостов являются точками сочленения

Мосты и точки сочленения

Для начала представим, что мы запустили **dfs** из произвольной вершины, введем новые понятия:

- **Прямые ребра** - те, по которым были переходы в **dfs**
- **Обратные ребра** - те, по которым не было переходов в **dfs**

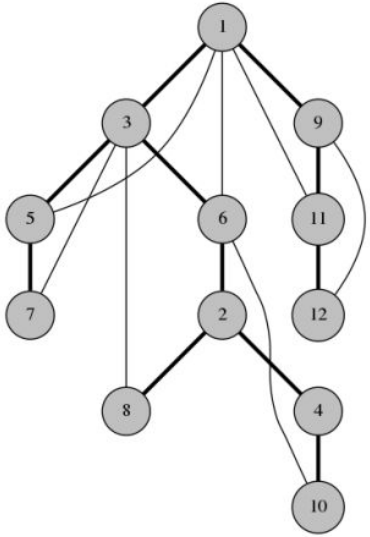


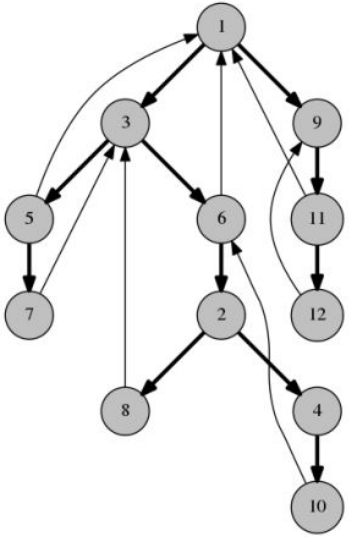


Мосты и точки сочленения

Важные замечания:

- Никакое обратное ребро не может являться мостом: если его удалить, все равно будет путь между любыми двумя вершинами, потому что подграф из прямых ребер является деревом
- Обратные ребра могут вести только “вверх” - к какому-то предку в дереве обхода графа но не в другие “ветки”





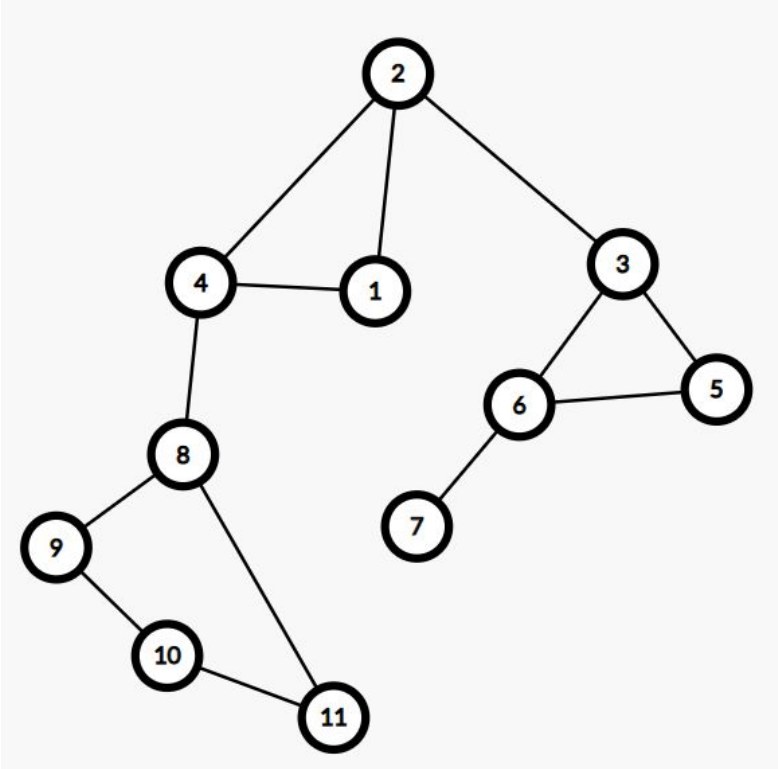
Поиск точек сочленения

Запустим **dfs** из произвольной вершины графа, назовем ее **root**. Также представим, что было сделано какое-то (возможно нулевое) количество шагов **dfs**, и мы сейчас находимся в какой-то вершине **v**.
Разберем 2 случая:

- **v == root**
  В этом случае вершина **v** является точкой сочленения тогда и только тогда, когда эта вершина имеет больше одного ребенка в дереве обхода в глубину. Другими словами, пройдя из **root** по произвольному ребру, мы не смогли обойти весь граф сразу, из чего следует, что **root** - точка сочленения.

Поиск точек сочленения

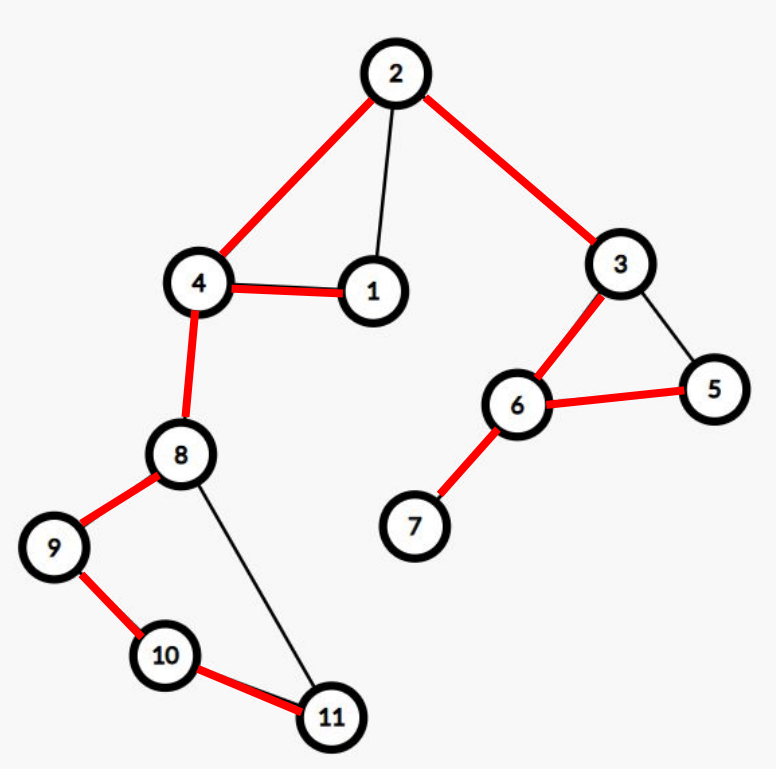
- **v != root**
  Посмотрим на все ребра **(v, to)**. Если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве **dfs** нет обратного ребра в какого-либо предка **v**, то вершина **v** является точкой сочленения, в противном случае не является. По факту, этим критерием мы проверяем, что нельзя дойти из **v** в **to** никаким другим способом, кроме как по ребру дерева **dfs**



Поиск точек сочленения

красные - ребра dfs-а
черные - обратные

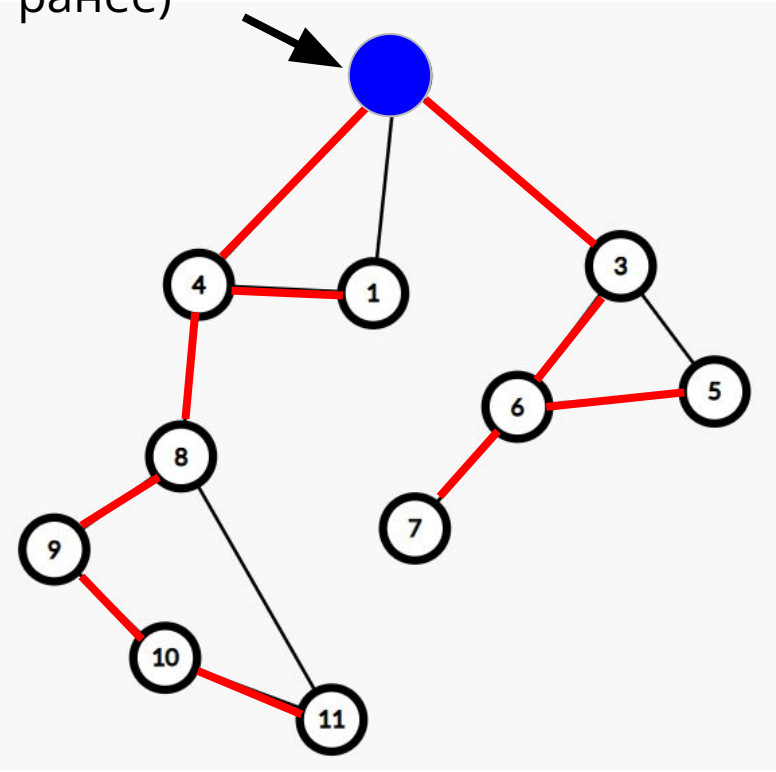
- **v != root**
  Посмотрим на все ребра **(v, to)**. Если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве **dfs** нет обратного ребра в какого-либо предка **v**, то вершина **v** является точкой сочленения, в противном случае не является. По факту, этим критерием мы проверяем, что нельзя дойти из **v** в **to** никаким другим способом, кроме как по ребру дерева **dfs**



Поиск точек сочленения

- **v != root**
  Посмотрим на все ребра **(v, to)**. Если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве **dfs** нет обратного ребра в какого-либо предка **v**, то вершина **v** является точкой сочленения, в противном случае не является. По факту, этим критерием мы проверяем, что нельзя дойти из **v** в **to** никаким другим способом, кроме как по ребру дерева **dfs**

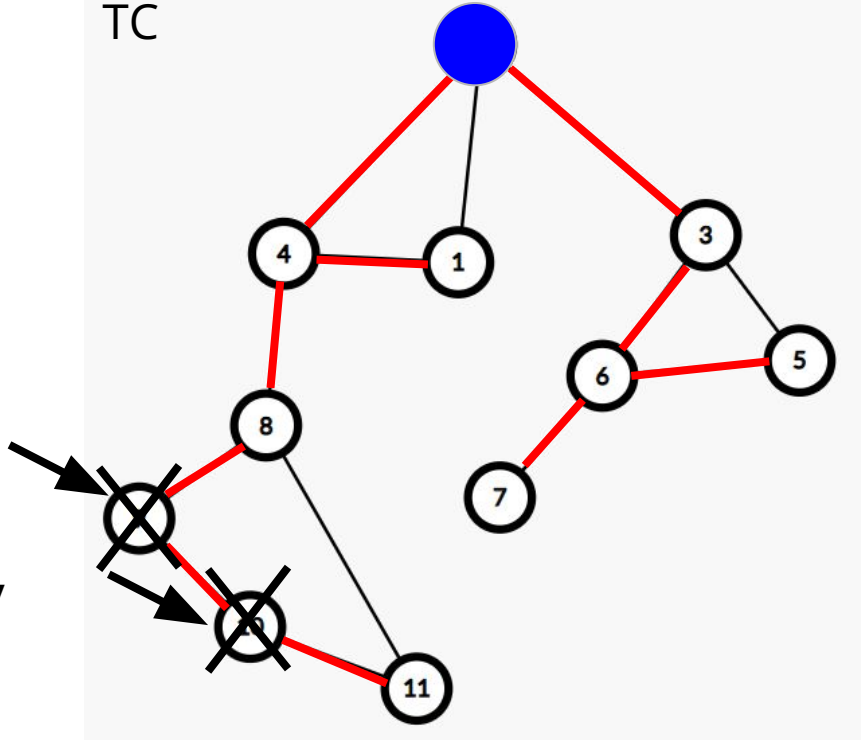
Эта вершина - root, и у нее более 1 потомка, значит она ТС (случай ранее)



Поиск точек сочленения

- **v != root**
  Посмотрим на все ребра **(v, to)**. Если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве **dfs** нет обратного ребра в какого-либо предка **v**, то вершина **v** является точкой сочленения, в противном случае не является. По факту, этим критерием мы проверяем, что нельзя дойти из **v** в **to** никаким другим способом, кроме как по ребру дерева **dfs**

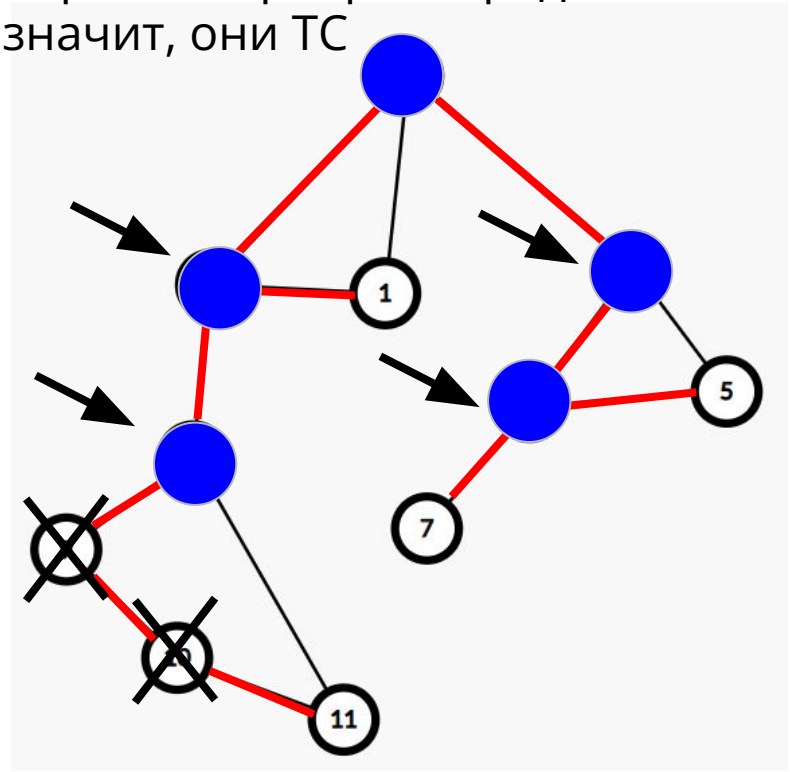
У этих вершин нет потомков, из которых нет обратного ребра в предка, поэтому они не являются ТС



Поиск точек сочленения

- **v != root**
  Посмотрим на все ребра **(v, to)**. Если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве **dfs** нет обратного ребра в какого-либо предка **v**, то вершина **v** является точкой сочленения, в противном случае не является. По факту, этим критерием мы проверяем, что нельзя дойти из **v** в **to** никаким другим способом, кроме как по ребру дерева **dfs**

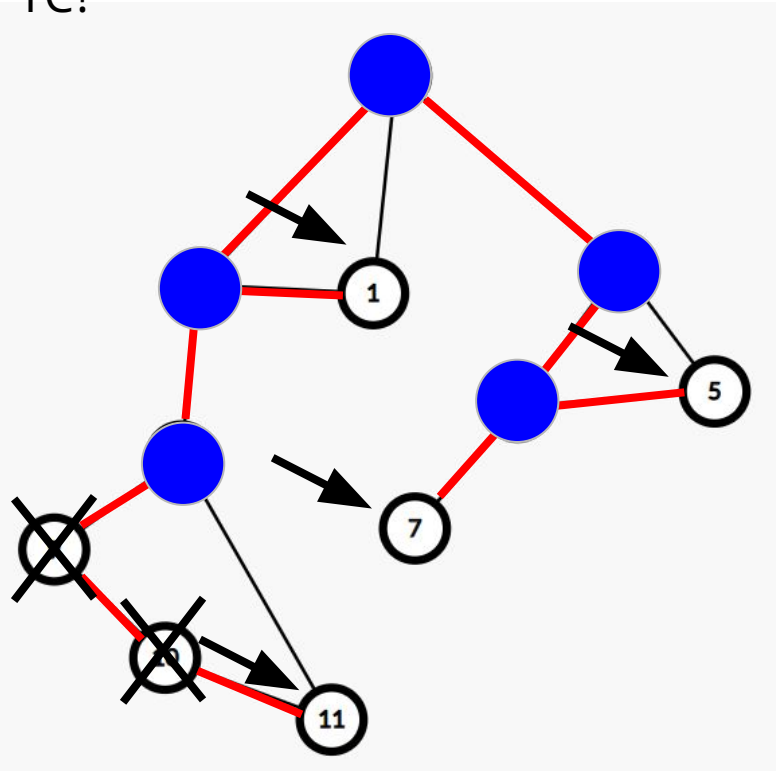
У этих вершин есть хотя бы 1 потомок, из которого нет обратного ребра в предка, значит, они ТС



Поиск точек сочленения

- **v != root**
  Посмотрим на все ребра **(v, to)**. Если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве **dfs** нет обратного ребра в какого-либо предка **v**, то вершина **v** является точкой сочленения, в противном случае не является. По факту, этим критерием мы проверяем, что нельзя дойти из **v** в **to** никаким другим способом, кроме как по ребру дерева **dfs**

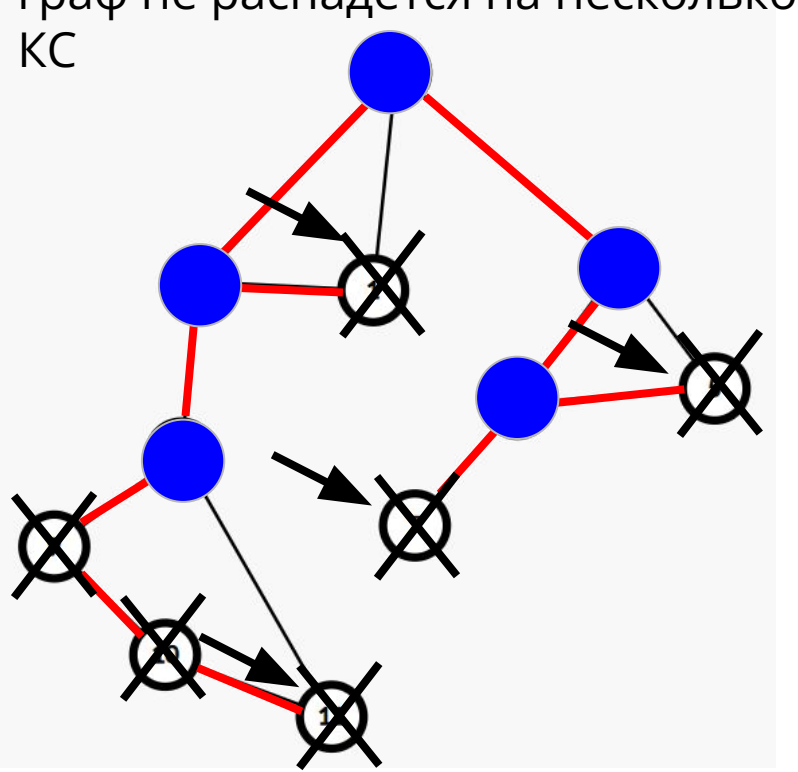
Остаются листы
Могут ли листы дерева dfs быть ТС?



Поиск точек сочленения

- **v != root**
  Посмотрим на все ребра **(v, to)**. Если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве **dfs** нет обратного ребра в какого-либо предка **v**, то вершина **v** является точкой сочленения, в противном случае не является. По факту, этим критерием мы проверяем, что нельзя дойти из **v** в **to** никаким другим способом, кроме как по ребру дерева **dfs**

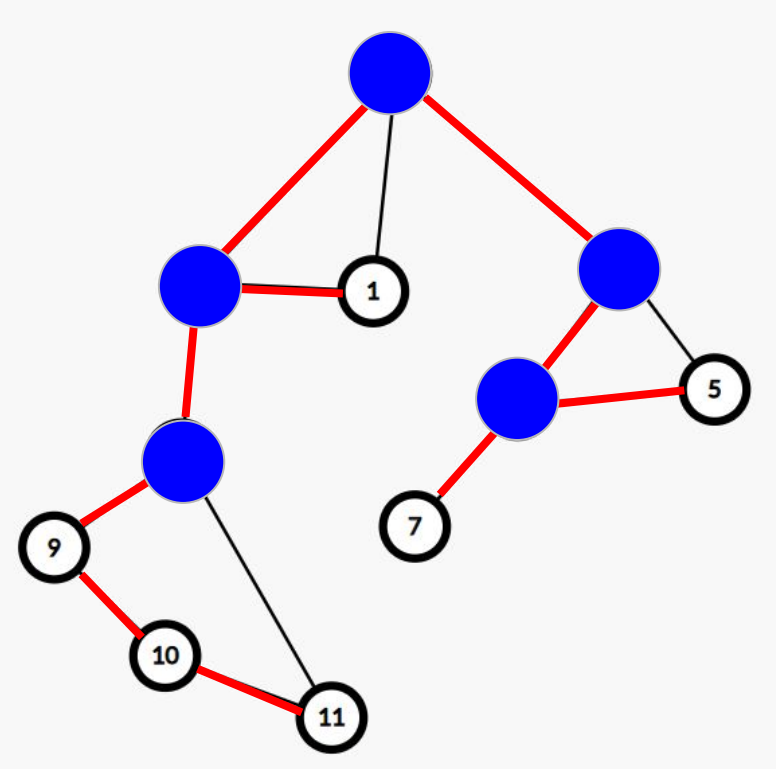
Нет, не могут, т к у них нет потомков -> при их удалении граф не распадется на несколько КС



Поиск точек сочленения

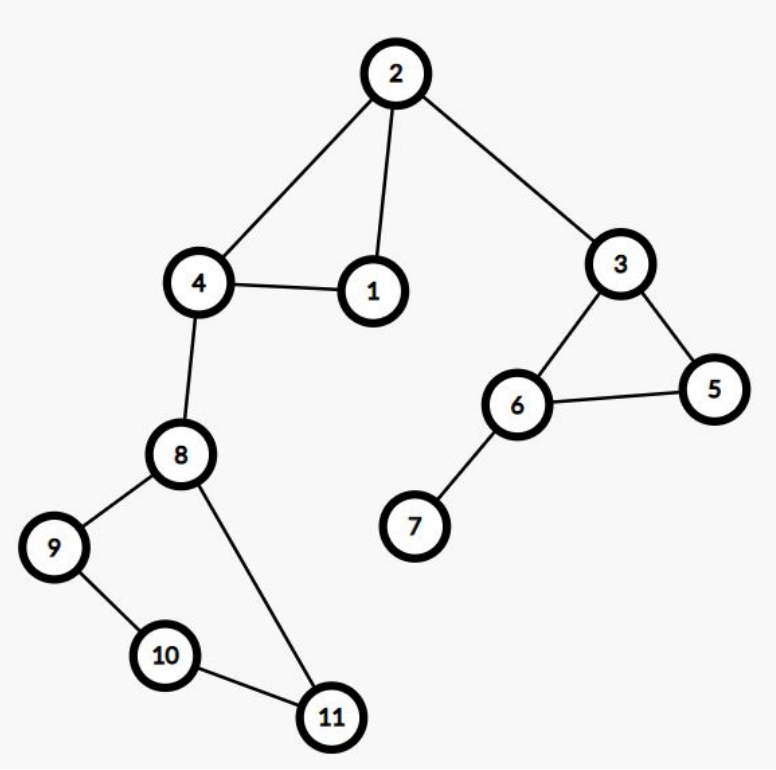
- **v != root**
  Посмотрим на все ребра **(v, to)**. Если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве **dfs** нет обратного ребра в какого-либо предка **v**, то вершина **v** является точкой сочленения, в противном случае не является. По факту, этим критерием мы проверяем, что нельзя дойти из **v** в **to** никаким другим способом, кроме как по ребру дерева **dfs**

Таким образом мы нашли все ТС графа



Поиск точек сочленения

Возникает вопрос: как эффективно проверять критерий, который мы придумали?

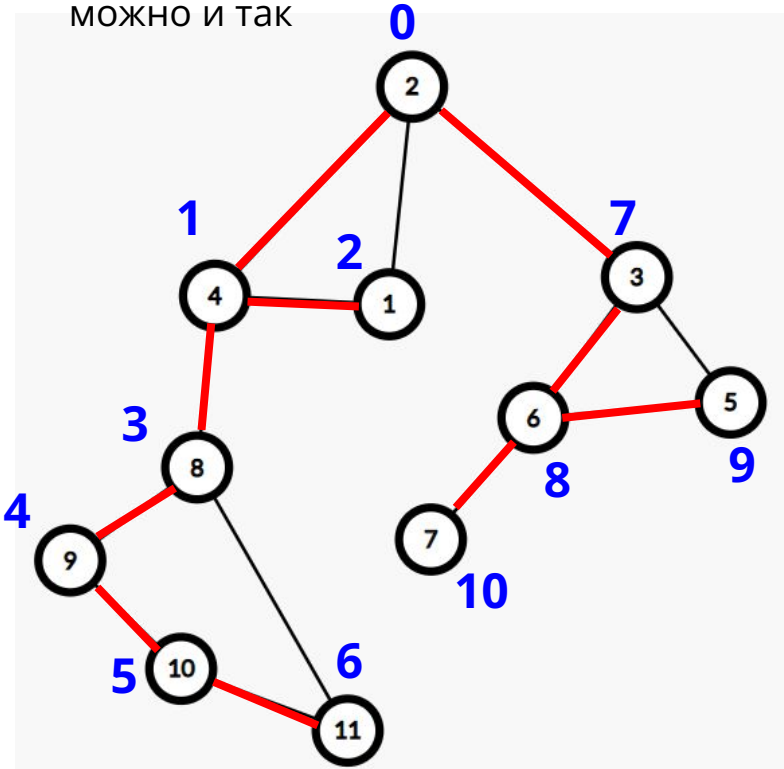


Поиск точек сочленения

Возникает вопрос: как эффективно проверять критерий, который мы придумали?

Ответ: воспользуемся временами входа в вершины (считается при помощи dfs-а). Назовем эту величину **tin[v]**

Прим.: обычно времена входа/выхода считаются немного иначе, но для данного алгоритма можно и так

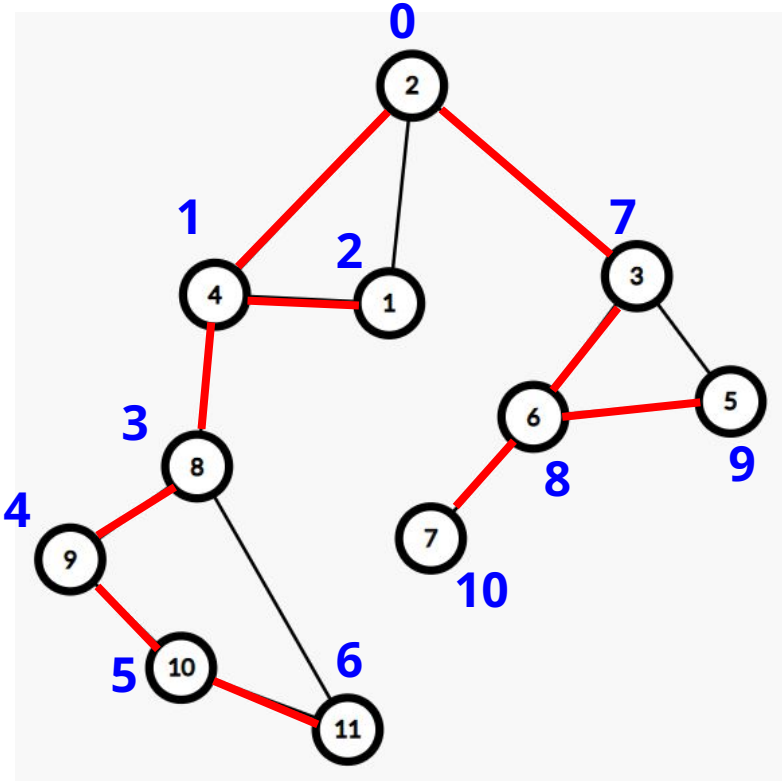


Поиск точек сочленения

Теперь определим величину **fup[v]** как минимум из:

- времени захода в саму вершину (**tin[v]**)
- времен захода в каждую вершину **p**, которая является концом **обратного** ребра **(v, p)**
- всех значений **fup[to]** для каждой вершины **to**, являющейся ребенком **v** в дереве **dfs**

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

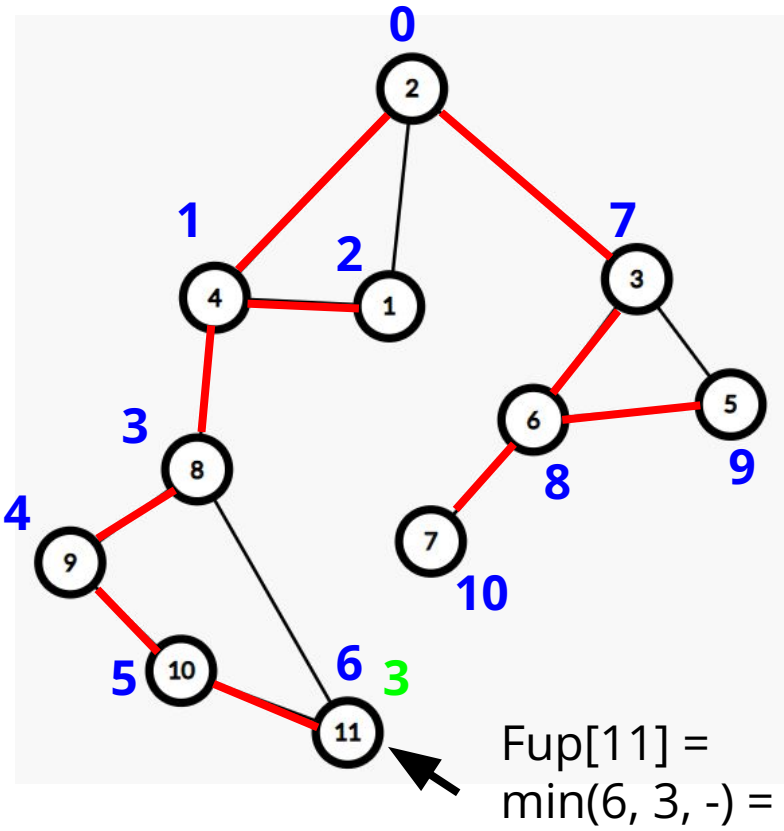


Поиск точек сочленения

Теперь определим величину **fup[v]** как минимум из:

- времени захода в саму вершину (**tin[v]**)
- времен захода в каждую вершину **p**, которая является концом **обратного** ребра **(v, p)**
- всех значений **fup[to]** для каждой вершины **to**, являющейся ребенком **v** в дереве **dfs**

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

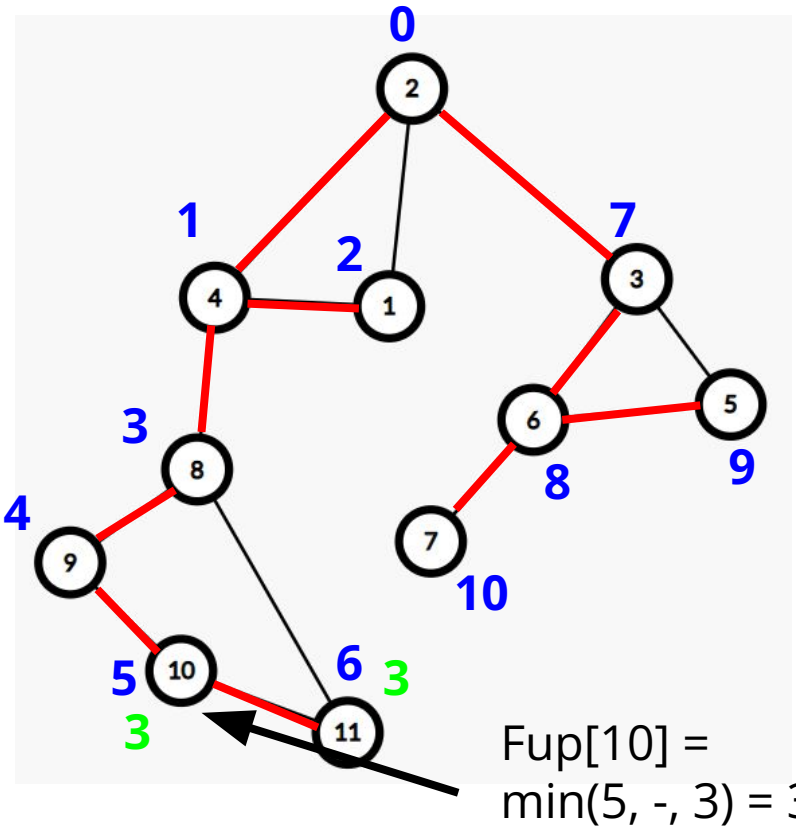


Поиск точек сочленения

Теперь определим величину **fup[v]** как минимум из:

- времени захода в саму вершину (**tin[v]**)
- времен захода в каждую вершину **p**, которая является концом **обратного** ребра **(v, p)**
- всех значений **fup[to]** для каждой вершины **to**, являющейся ребенком **v** в дереве **dfs**

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

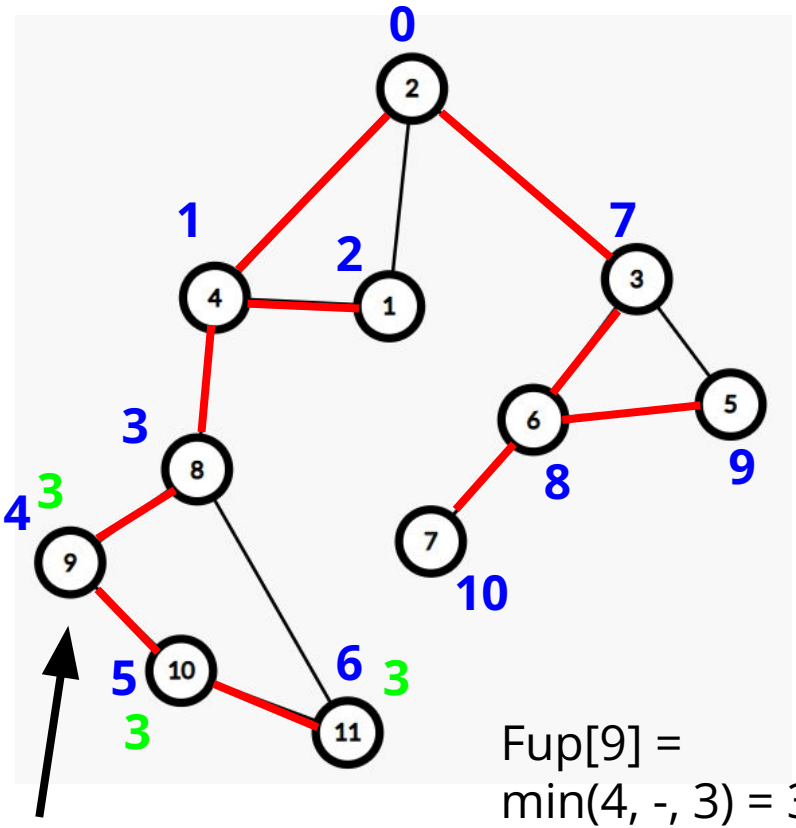


Поиск точек сочленения

Теперь определим величину **fup[v]** как минимум из:

- времени захода в саму вершину (**tin[v]**)
- времен захода в каждую вершину **p**, которая является концом **обратного** ребра **(v, p)**
- всех значений **fup[to]** для каждой вершины **to**, являющейся ребенком **v** в дереве **dfs**

$$fup[v] = \min \begin{cases} tin[v], \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$



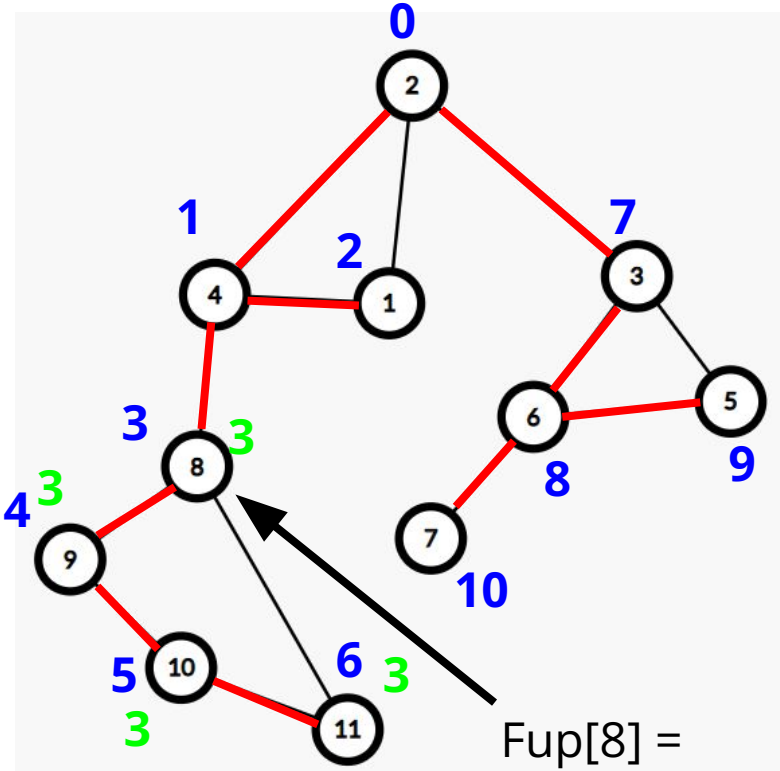
Fup[9] =
min(4, -, 3) = 3

Поиск точек сочленения

Теперь определим величину **fup[v]** как минимум из:

- времени захода в саму вершину (**tin[v]**)
- времен захода в каждую вершину **p**, которая является концом **обратного** ребра **(v, p)**
- всех значений **fup[to]** для каждой вершины **to**, являющейся ребенком **v** в дереве **dfs**

$$fup[v] = \min \begin{cases} tin[v], \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

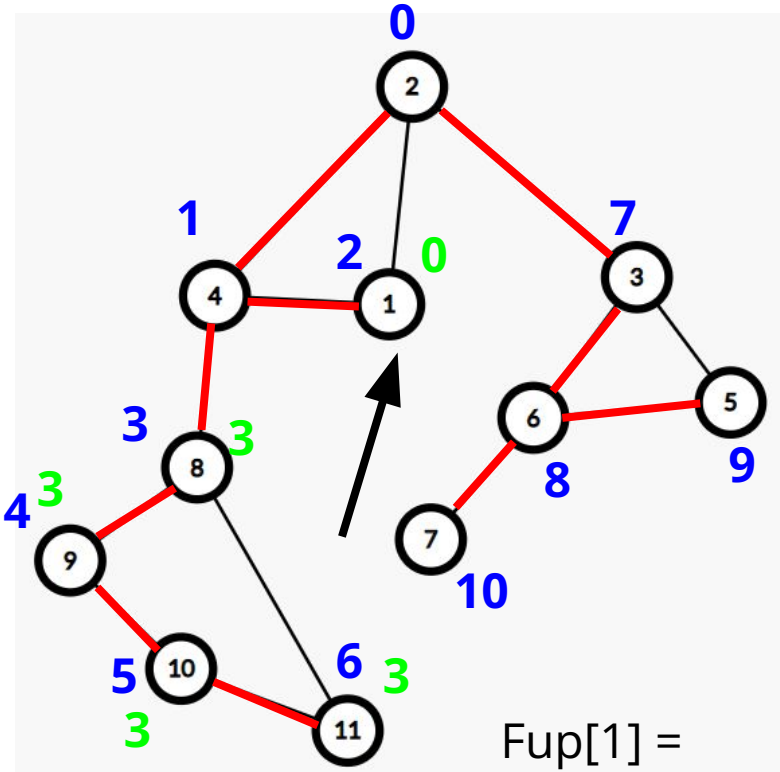


Поиск точек сочленения

Теперь определим величину **fup[v]** как минимум из:

- времени захода в саму вершину (**tin[v]**)
- времен захода в каждую вершину **p**, которая является концом **обратного** ребра **(v, p)**
- всех значений **fup[to]** для каждой вершины **to**, являющейся ребенком **v** в дереве **dfs**

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$



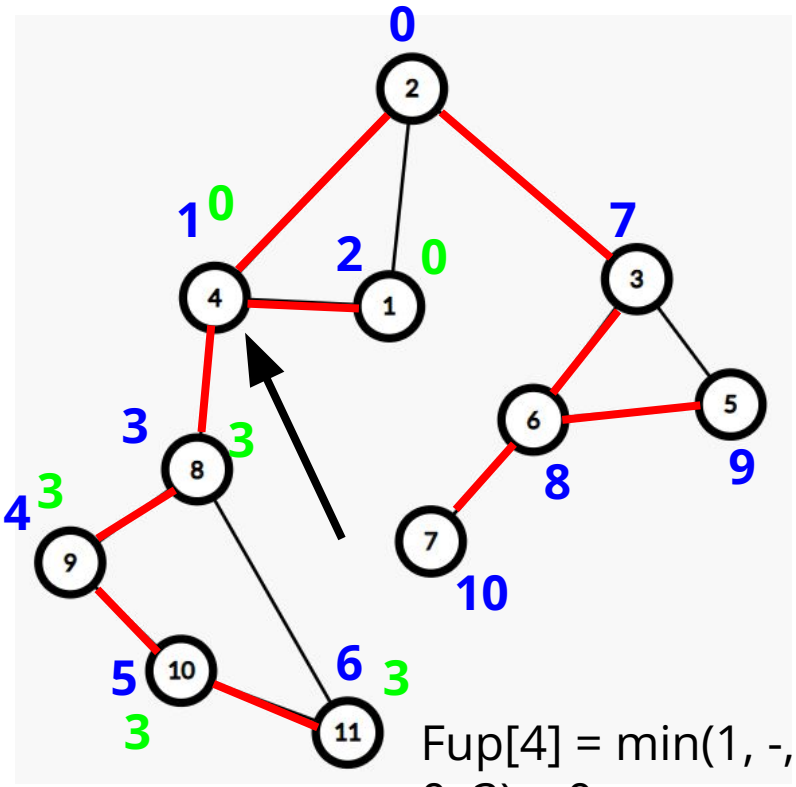
Fup[1] =
min(2, 0, -) = 0

Поиск точек сочленения

Теперь определим величину **fup[v]** как минимум из:

- времени захода в саму вершину (**tin[v]**)
- времен захода в каждую вершину **p**, которая является концом **обратного** ребра **(v, p)**
- всех значений **fup[to]** для каждой вершины **to**, являющейся ребенком **v** в дереве **dfs**

$$fup[v] = \min \begin{cases} tin[v], \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$



Fup[4] = min(1, -, 0, 3) = 0

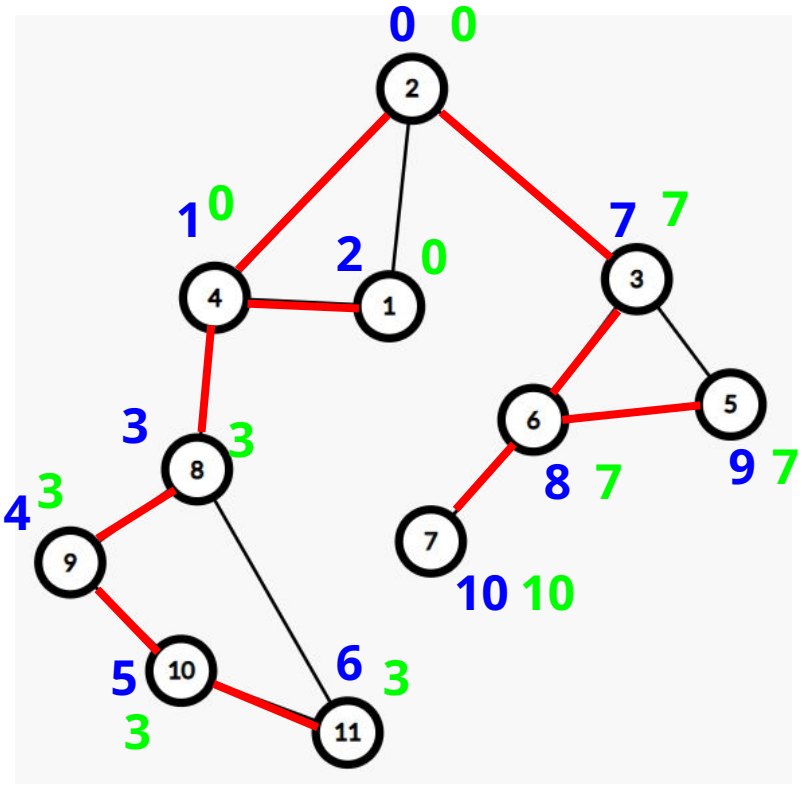
Поиск точек сочленения

Теперь определим величину **fup[v]** как минимум из:

- времени захода в саму вершину (**tin[v]**)
- времен захода в каждую вершину **p**, которая является концом **обратного** ребра **(v, p)**
- всех значений **fup[to]** для каждой вершины **to**, являющейся ребенком **v** в дереве **dfs**

$$fup[v] = \min \begin{cases} tin[v], \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

Далее считаем аналогично

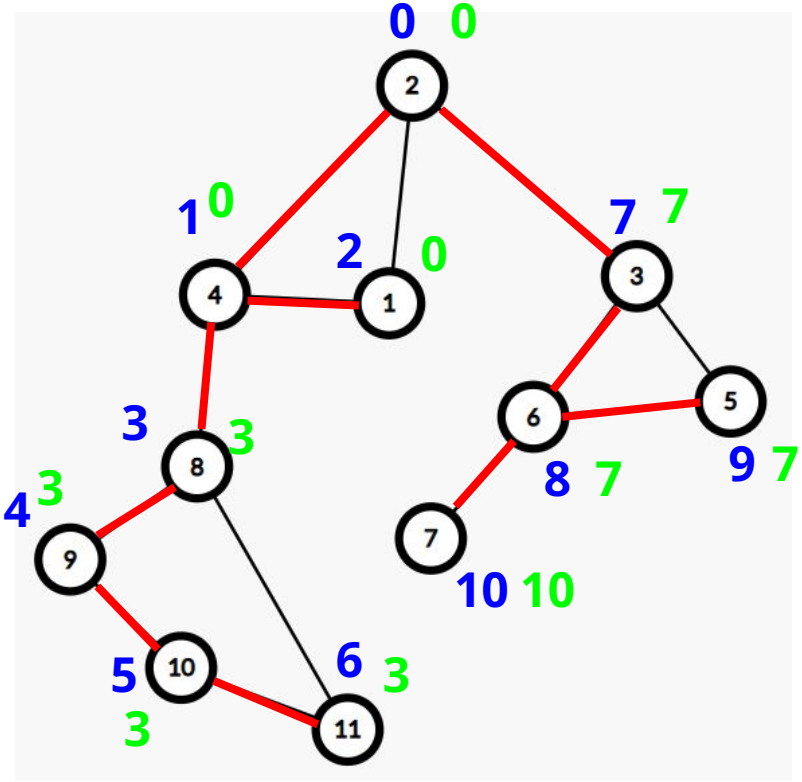


Поиск точек сочленения

Теперь можно сказать, что из вершины **v** или ее потомка есть обратное ребро в ее предка тогда и только тогда, когда у вершины **v** найдется такой сын **to**, что

fup[to] < tin[v]

$$fup[v] = \min \begin{cases} tin[v], \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$



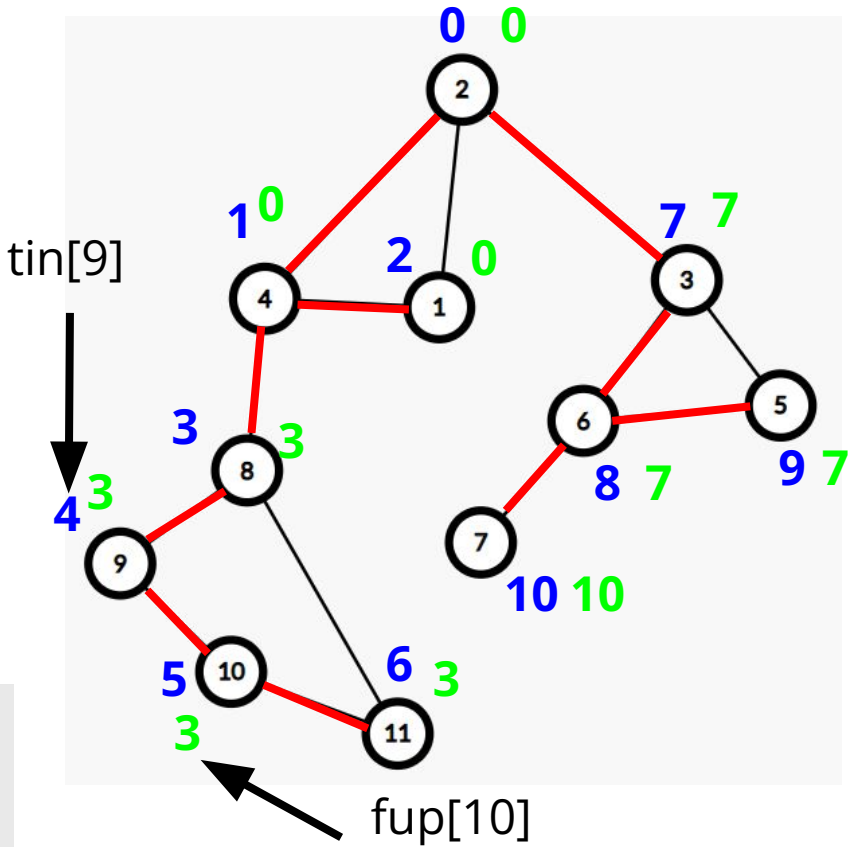
Поиск точек сочленения

Теперь можно сказать, что из вершины **v** или ее потомка есть обратное ребро в ее предка тогда и только тогда, когда у вершины **v** найдется такой сын **to**, что

fup[to] < tin[v]

Например, fup[10] < tin[9], значит, где-то в поддереве вершины 9 есть обратное ребро в ее предка

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

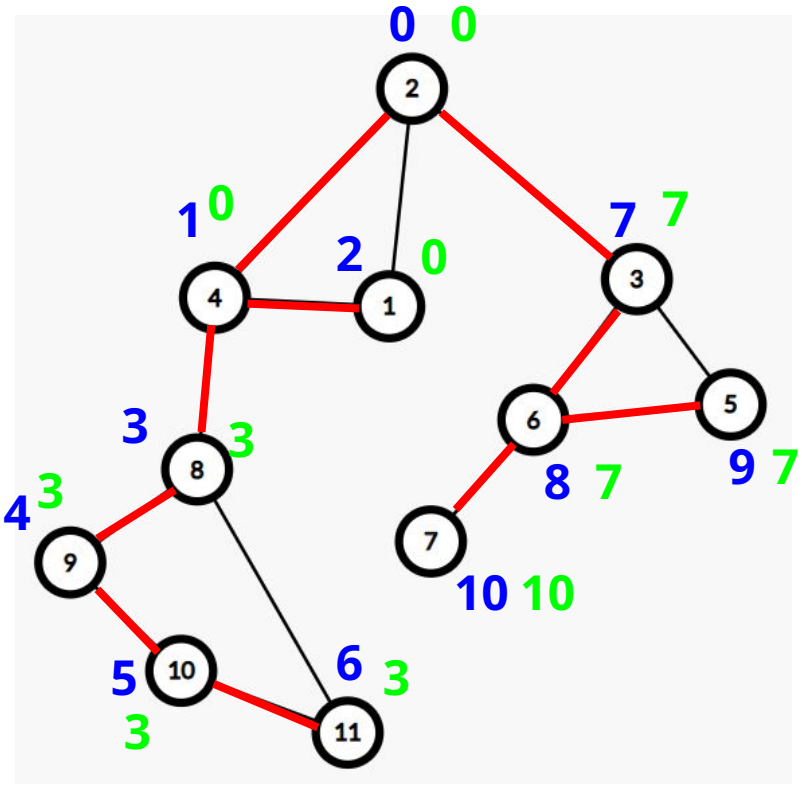


Поиск точек сочленения

Теперь критерий можно переформулировать так:

- Если **v == root**, считаем количество детей в дереве dfs
- Если **v != root**, вершина **v** является ТС, если среди ее детей найдется такая вершина **to**, что **fup[to] >= tin[v]**

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$



Поиск точек сочленения

В самой реализации нужно уметь отличать 3 случая:

- когда мы идем по ребру dfs **used[to] == false**
- когда идем по обратному ребру **used[to] == true && to != parent**
- когда пытаемся пойти по ребру дерева в обратную сторону **to == parent**

=> в функцию надо передавать предка

```
int main() {
        int n;
        ... чтение n и g ...

        timer = 0;
        for (int i=0; i<n; ++i)
                used[i] = false;
        dfs (0);
}
```

```
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

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

Поиск точек сочленения

Здесь константе **MAXN** должно быть задано значение, равное максимально возможному числу вершин во входном графе.
Функция I**S_CUTPOINT(u)** в коде — это некая функция, которая будет реагировать на то, что вершина **u** является точкой сочленения, например, выводить эту вершины на экран (надо учитывать, что для одной и той же вершины эта функция может быть вызвана несколько раз).

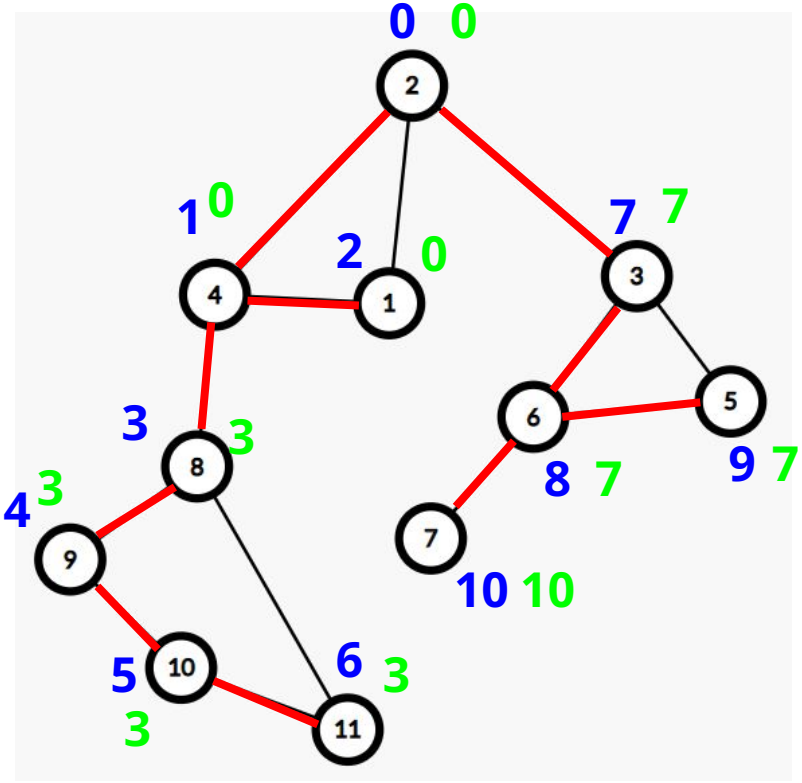
```
int main() {
        int n;
        ... чтение n и g ...

        timer = 0;
        for (int i=0; i<n; ++i)
                used[i] = false;
        dfs (0);
}
```

```
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

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

Поиск мостов

Поиск мостов очень похож на поиск ТС. Рассуждения примерно такие же:

Пусть мы находимся при обходе в глубину в какой-то вершине **v**. Тогда, если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве dfs нет обратного ребра в вершину **v** или какого-либо ее предка, то это ребро **является мостом**. В противном случае, оно мостом **не является.**

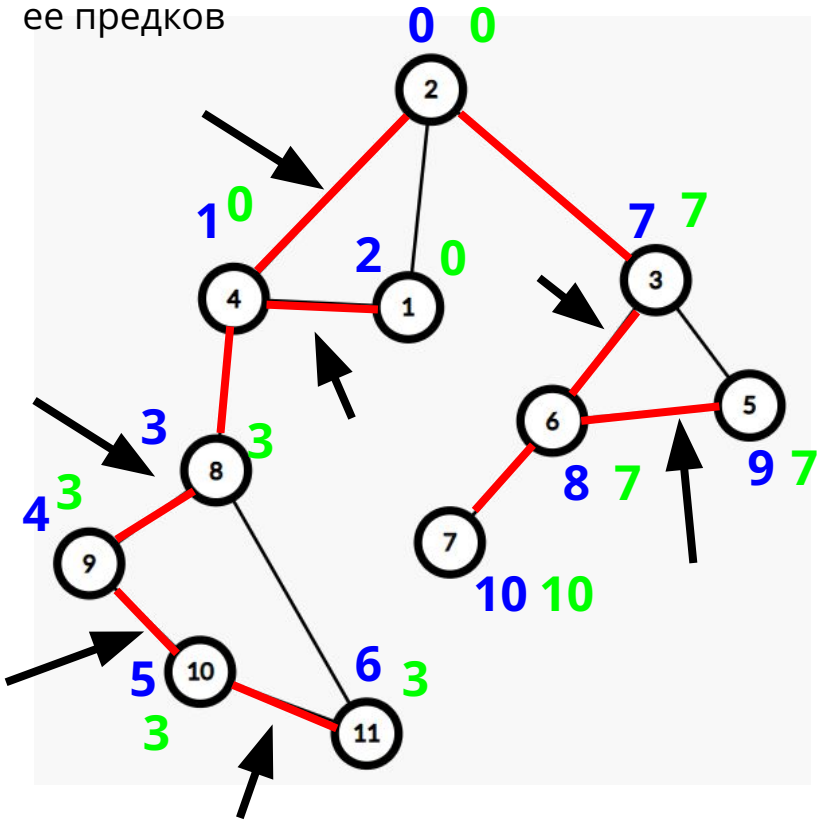


Поиск мостов

Поиск мостов очень похож на поиск ТС. Рассуждения примерно такие же:

Пусть мы находимся при обходе в глубину в какой-то вершине **v**. Тогда, если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве dfs нет обратного ребра в вершину **v** или какого-либо ее предка, то это ребро **является мостом**. В противном случае, оно мостом **не является.**

Эти ребра не являются мостами, т к у конечной вершины или ее потомков есть обратные ребра в начальную вершину или ее предков

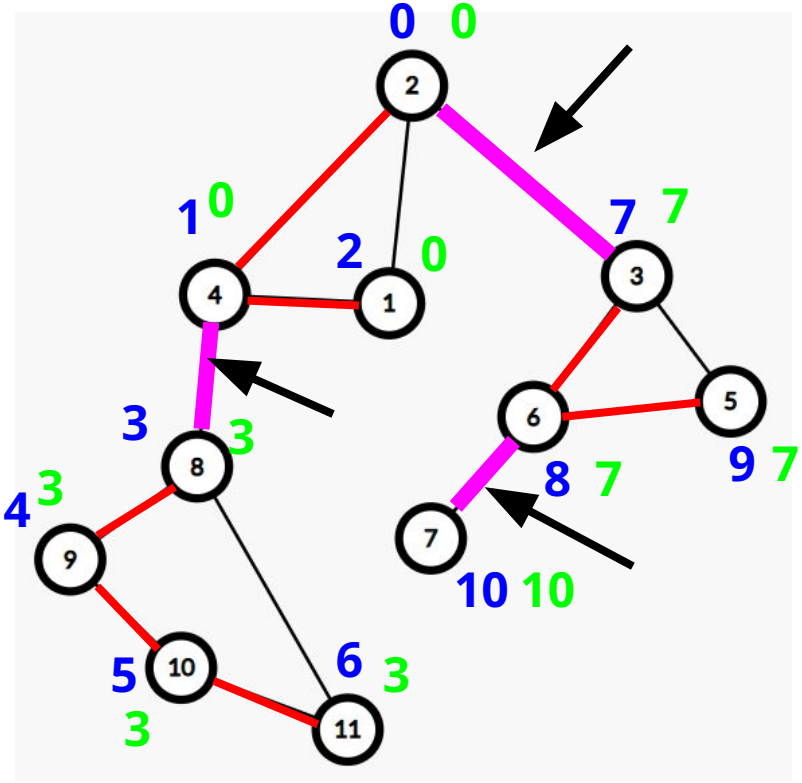


Поиск мостов

Поиск мостов очень похож на поиск ТС. Рассуждения примерно такие же:

Пусть мы находимся при обходе в глубину в какой-то вершине **v**. Тогда, если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве dfs нет обратного ребра в вершину **v** или какого-либо ее предка, то это ребро **является мостом**. В противном случае, оно мостом **не является.**

А эти ребра являются мостами, т к из поддеревьев потомков нет обратных ребер в начальную вершину или ее предков

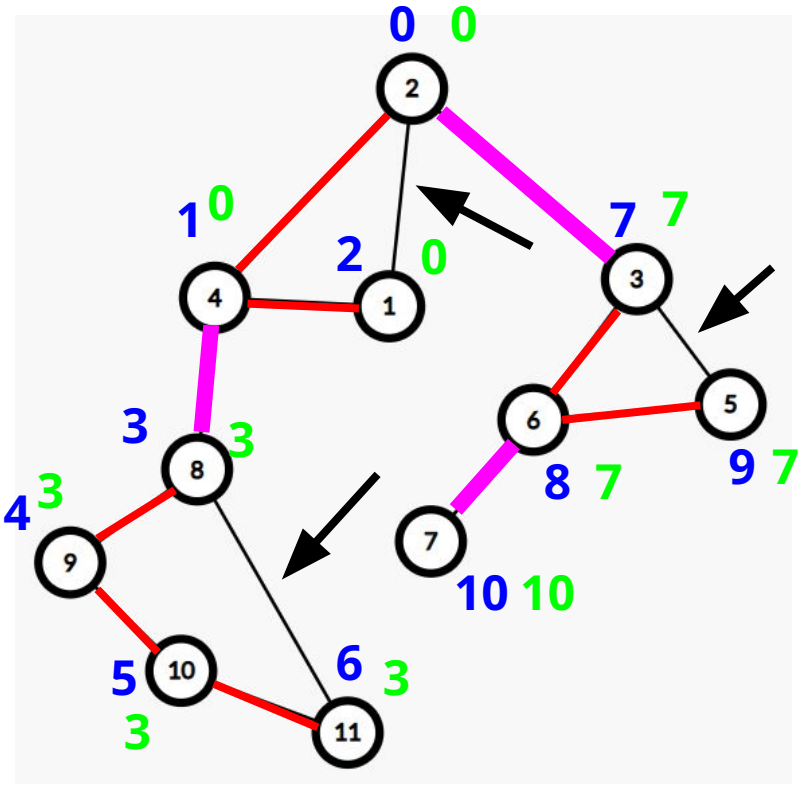


Поиск мостов

Факт, что никакое обратное ребро не может являться мостом, обсуждался ранее

Поиск мостов очень похож на поиск ТС. Рассуждения примерно такие же:

Пусть мы находимся при обходе в глубину в какой-то вершине **v**. Тогда, если текущее ребро **(v, to)** таково, что из вершины **to** и любого ее потомка в дереве dfs нет обратного ребра в вершину **v** или какого-либо ее предка, то это ребро **является мостом**. В противном случае, оно мостом **не является.**

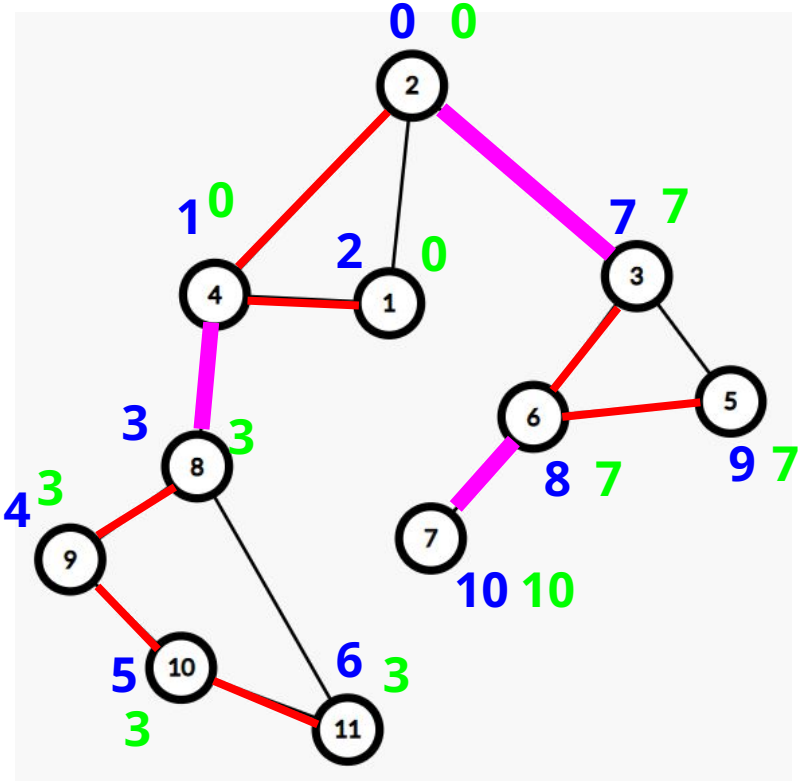


Поиск мостов

Теперь можно сказать, что из вершины **v** или ее потомка есть обратное ребро в ее предка тогда и только тогда, когда у вершины **v** найдется такой сын **to**, что

fup[to] <= tin[v]

Например, fup[9] = tin[8], значит, где-то в поддереве вершины 9 есть обратное ребро в вершину 8



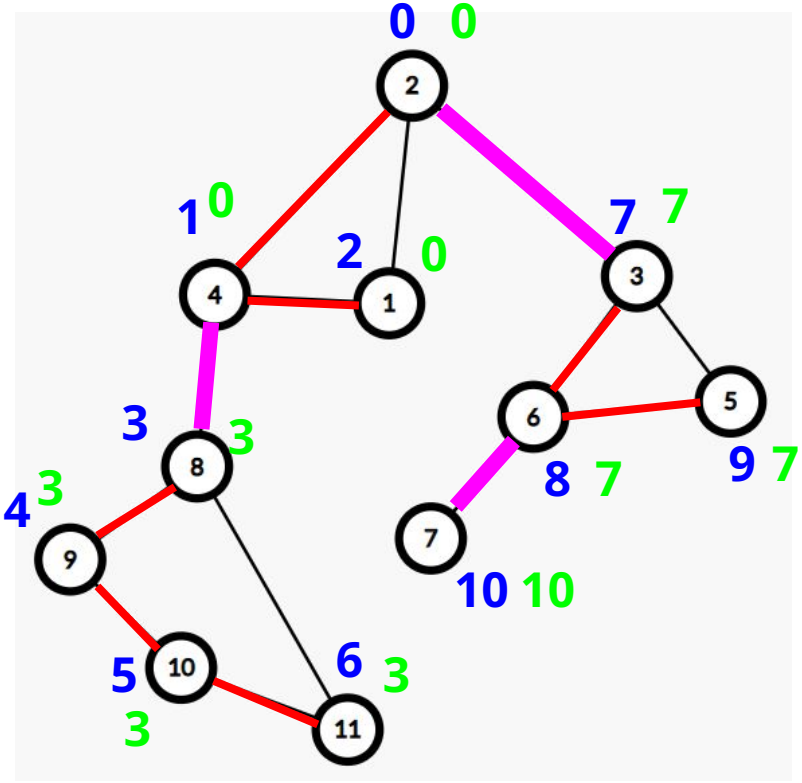
Поиск мостов

Теперь критерий можно переформулировать так:

Если для текущего ребра **(v, to)** из dfs выполняется

fup[to] > tin[v]

то это ребро является мостом. В противном случае, оно мостом не является



Поиск мостов

Реализация аналогична поиску ТС:

- когда мы идем по ребру dfs **used[to] == false**
- когда идем по обратному ребру **used[to] == true && to != parent**
- когда пытаемся пойти по ребру дерева в обратную сторону **to == parent**

=> в функцию надо передавать предка

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

```
const int MAXN = ...;
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

void dfs (int v, int p = -1) {
        used[v] = true;
        tin[v] = fup[v] = timer++;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (to == p)  continue;
                if (used[to])
                        fup[v] = min (fup[v], tin[to]);
                else {
                        dfs (to, v);
                        fup[v] = min (fup[v], fup[to]);
                        if (fup[to] > tin[v])
                                IS_BRIDGE(v,to);
                }
        }
}
```

Поиск мостов

Здесь основная функция для вызова — это **find_bridges** — она производит необходимую инициализацию и запуск обхода в глубину для каждой компоненты связности графа.

```
void find_bridges() {
        timer = 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

IS_BRIDGE(a,b) — это некая функция, которая будет реагировать на то, что ребро является мостом, например, выводить это ребро на экран.
Константе **MAXN** в самом начале кода следует задать значение, равное максимально возможному числу вершин во входном графе.

```
const int MAXN = ...;
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

void dfs (int v, int p = -1) {
        used[v] = true;
        tin[v] = fup[v] = timer++;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (to == p)  continue;
                if (used[to])
                        fup[v] = min (fup[v], tin[to]);
                else {
                        dfs (to, v);
                        fup[v] = min (fup[v], fup[to]);
                        if (fup[to] > tin[v])
                                IS_BRIDGE(v,to);
```

Стоит заметить, что эта реализация некорректно работает при наличии в графе **кратных рёбер**: она фактически не обращает внимания, кратное ли ребро или оно единственно.