УНИВЕРСИТЕТ ИТМО

ГРАФЫ: часть 1

Лекторы
Пермяков Антон Сергеевич
Ткаченко Данил Михайлович

АЛГОРИТМЫ

1. Обход в ширину
   - Подсчет длины пути
2. Обход в глубину
   - Поиск цикла
3. Топологическая сортировка
4. Поиск компонент связности (+ слабой связности)
5. Поиск компонент сильной связности
   - Конденсация графа (ориентированного)

Обход в ширину

- Какие идеи эффективной реализации?
    - *Используем очередь для отслеживания вершин, что горят*
    - *Убрали из очереди – пометили, что сгорела*
    - *Рассматриваем в один момент времени – одну вершину*
- Как найти расстояние до всех вершин от стартовой ?
    - *Использовать дополнительный массив для подсчета расстояния от стартовой до всех остальных*
    - *+ Массив предков для восстановления путей*
- Можно ли хранить граф иначе для обхода в ширину?
    - *Список смежности/Матрица инцидентности*

Обходы в ширину:
BFS (G(V, E) , s – стартовая вершина)

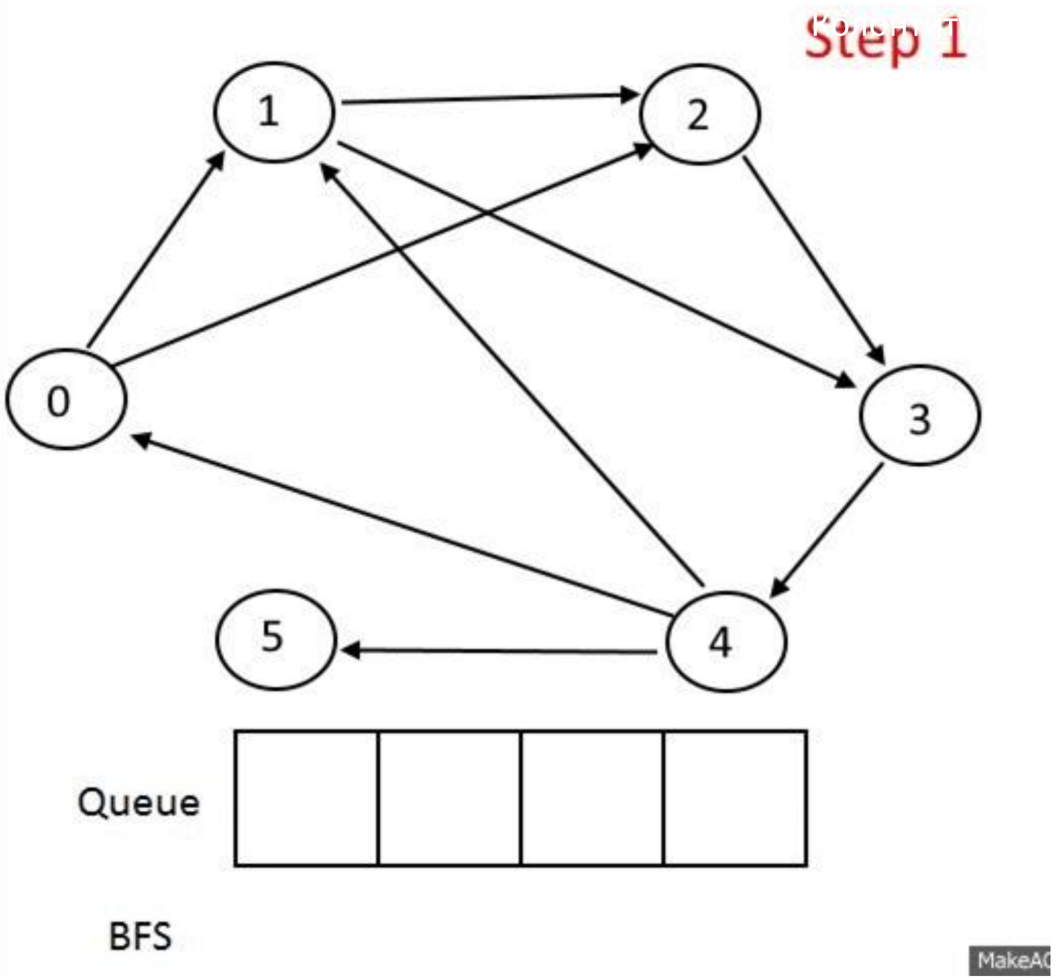
!!!

- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
  - a. Берем вершину из очереди и помечаем ее как пройдённую
  - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
  - c. Удаляем из очереди пройденную вершину

по одной вершине рассматр

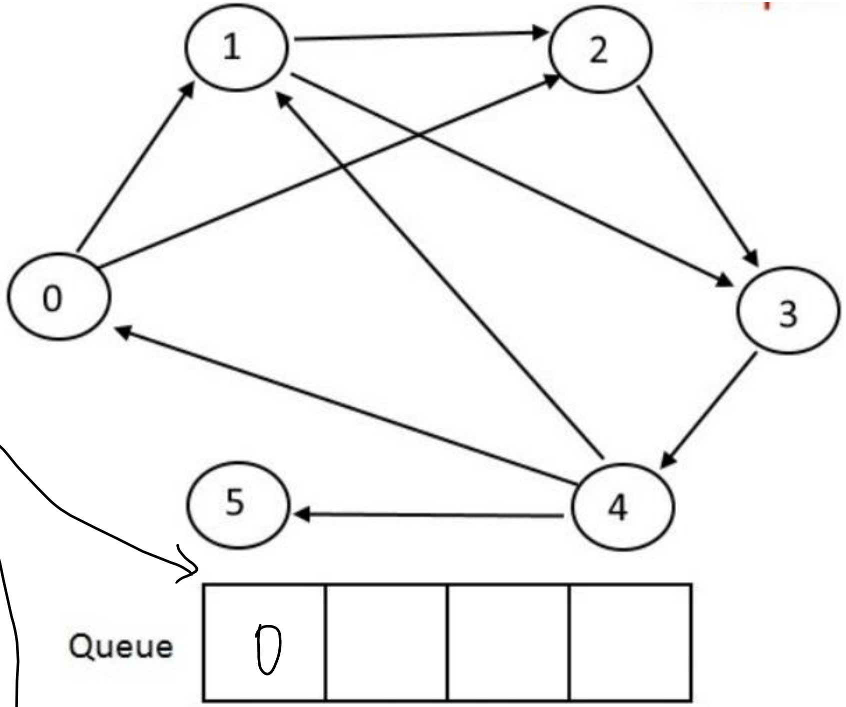
В ширину: BFS

- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
    - a. Берем вершину из очереди и помечаем ее как пройдённую
    - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
    - c. Удаляем из очереди пройденную вершину



В ширину: BFS

- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
  - a. Берем вершину из очереди и помечаем ее как пройдённую
  - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
  - c. Удаляем из очереди пройденную вершину

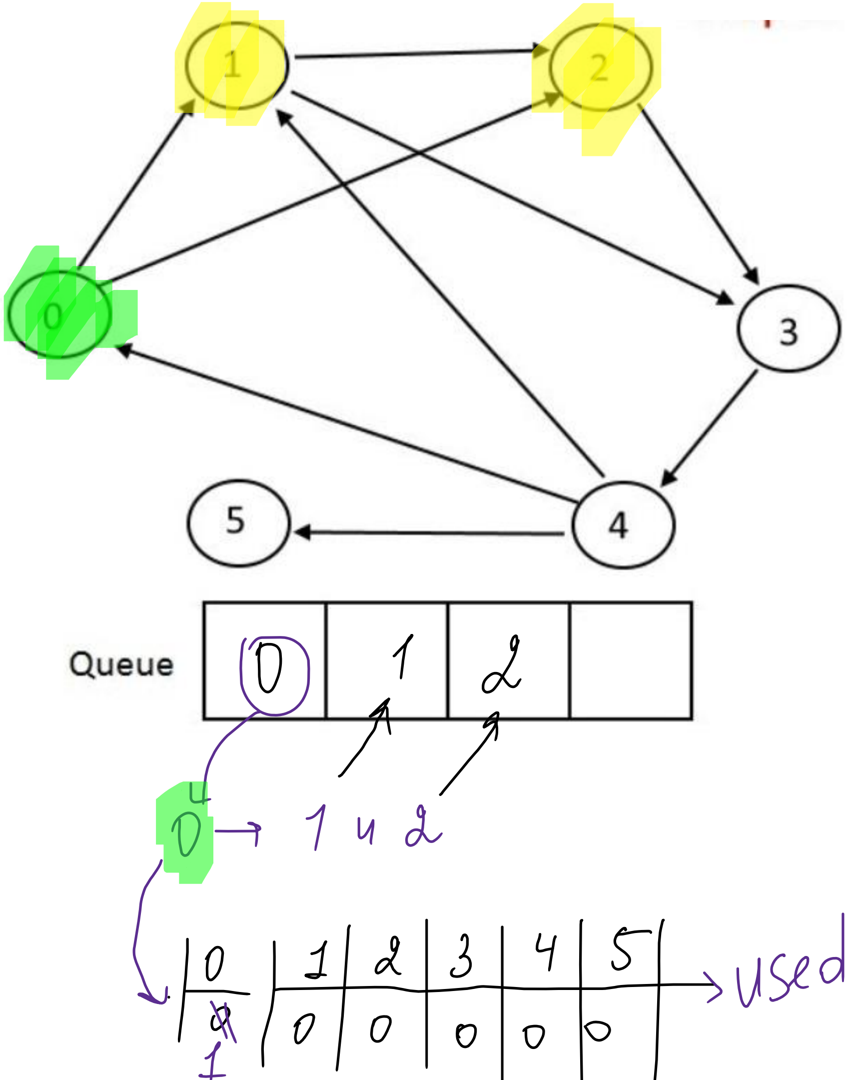


0	1	2	3	4	5
0	0	0	0	0	0

used

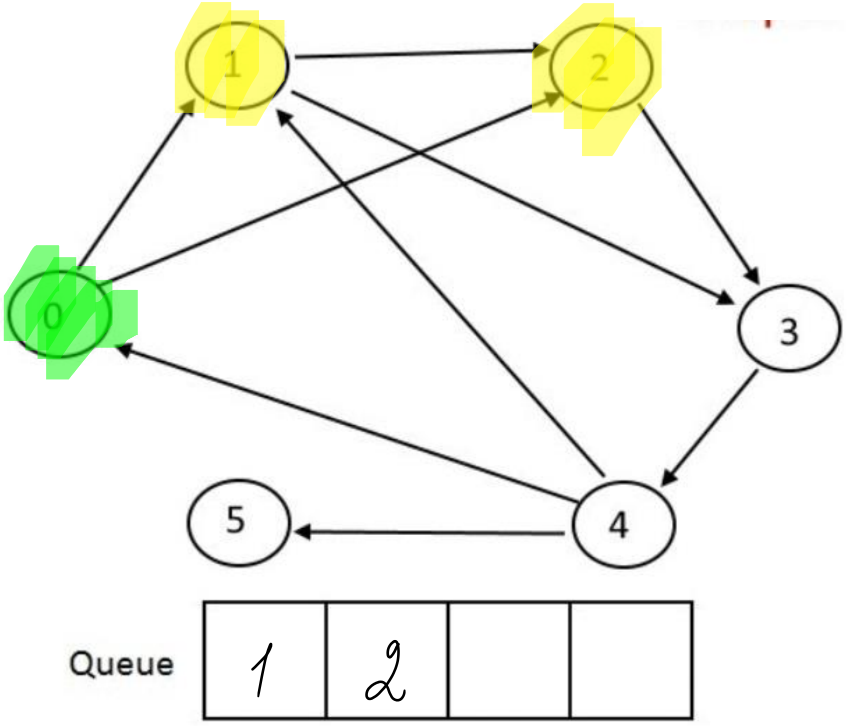
В ширину: BFS

- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
    - a. Берем вершину из очереди и помечаем ее как пройдённую
    - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
    - c. Удаляем из очереди пройденную вершину



В ширину: BFS

- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
    - a. Берем вершину из очереди и помечаем ее как пройдённую
    - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
    - c. Удаляем из очереди пройденную вершину



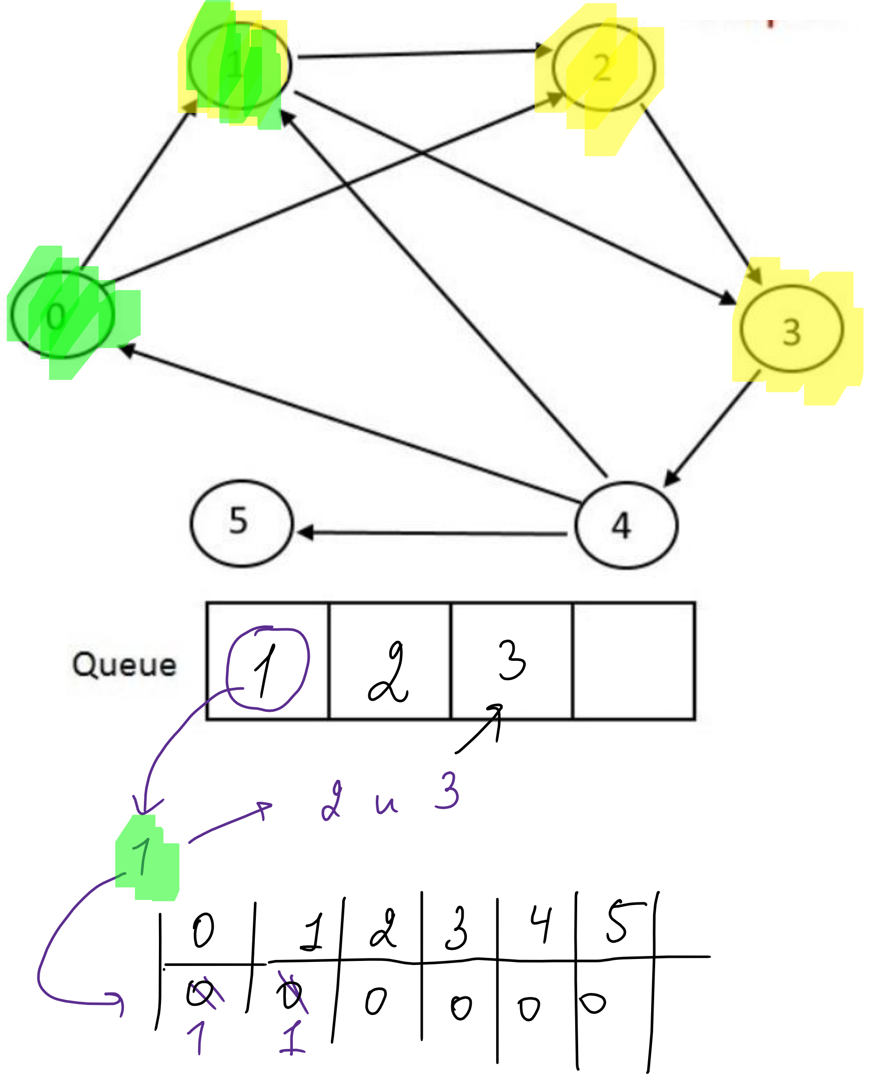
0	1	2	3	4	5
~~0~~ 1	0	0	0	0	0

→ used

В ширину: BFS

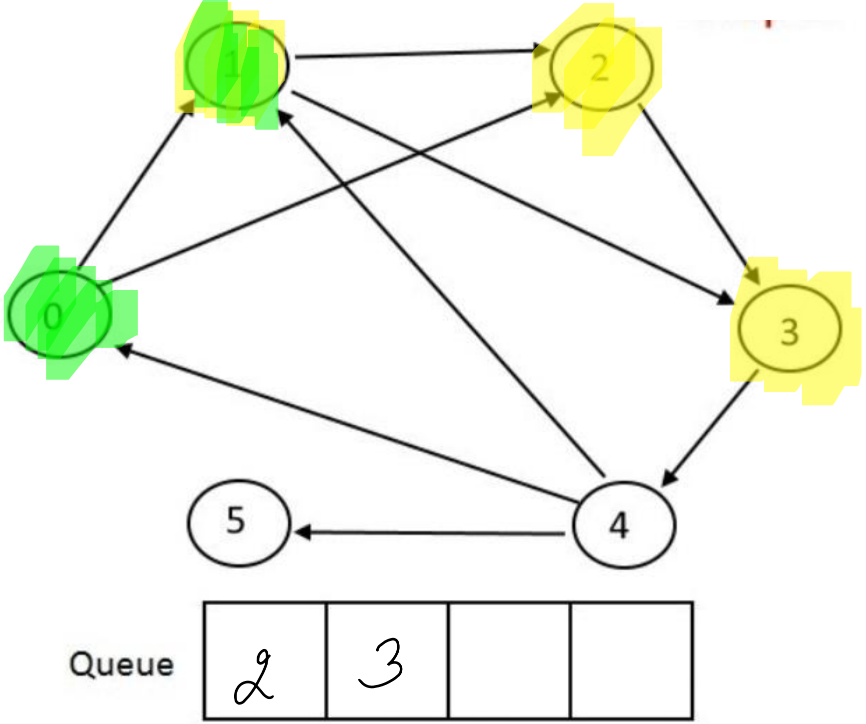
- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
  - a. Берем вершину из очереди и помечаем ее как пройдённую
  - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
  - c. Удаляем из очереди пройденную вершину

! 2 уже есть в очереди!
(уже есть)



В ширину: BFS

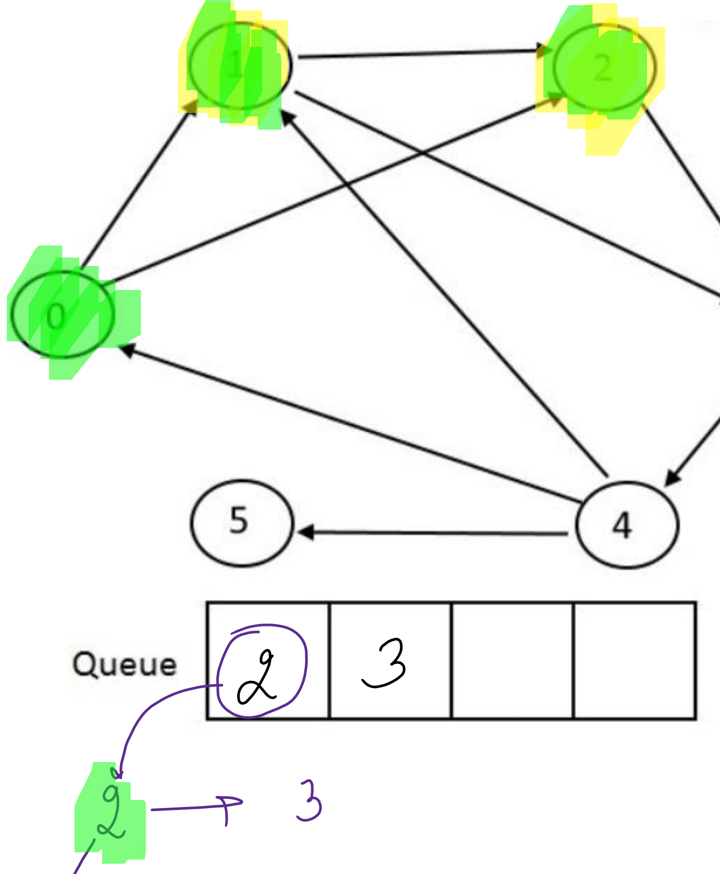
- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
    - a. Берем вершину из очереди и помечаем ее как пройдённую
    - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
    - c. Удаляем из очереди пройденную вершину



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

В ширину: BFS

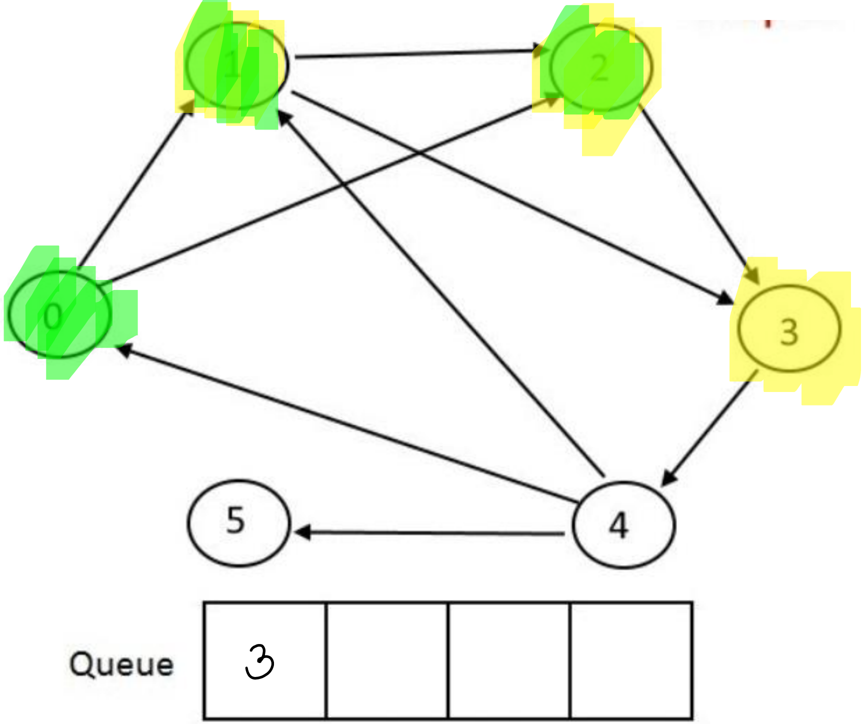
- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
    - a. Берем вершину из очереди и помечаем ее как пройдённую
    - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
    - c. Удаляем из очереди пройденную вершину



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

В ширину: BFS

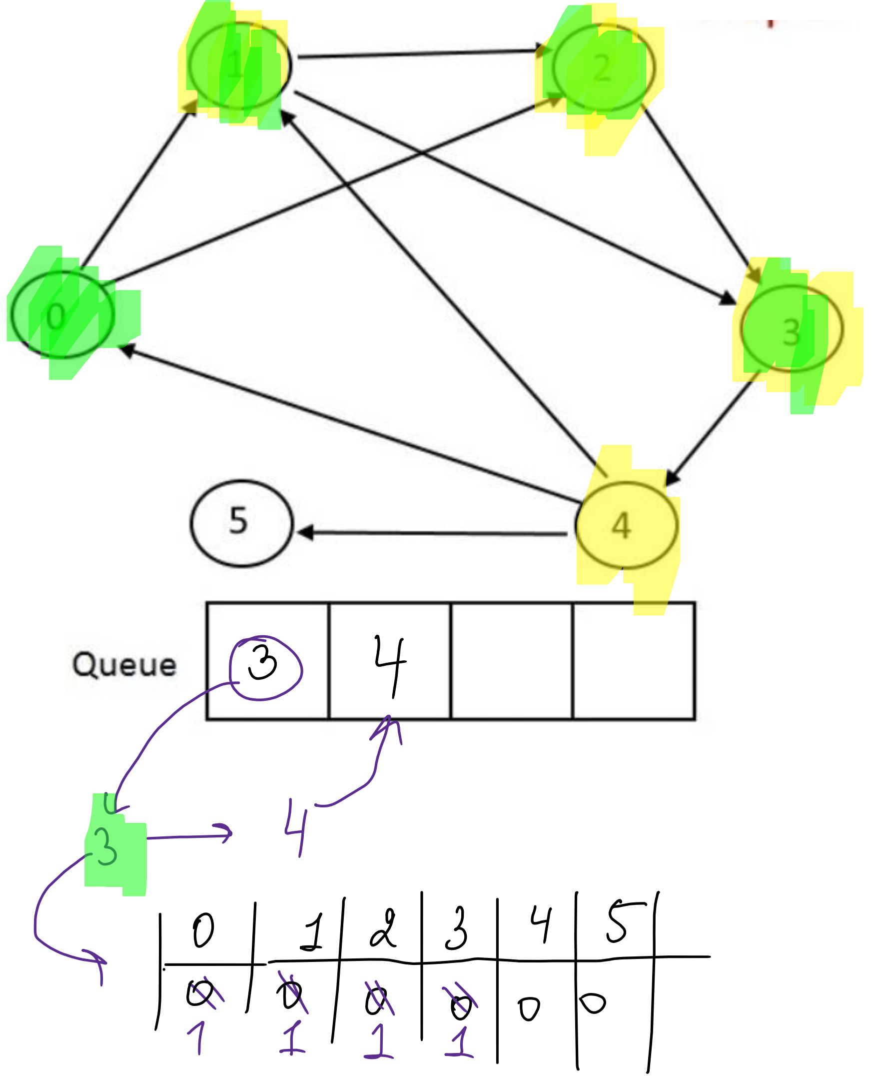
- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
    - a. Берем вершину из очереди и помечаем ее как пройдённую
    - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
    - c. Удаляем из очереди пройденную вершину



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

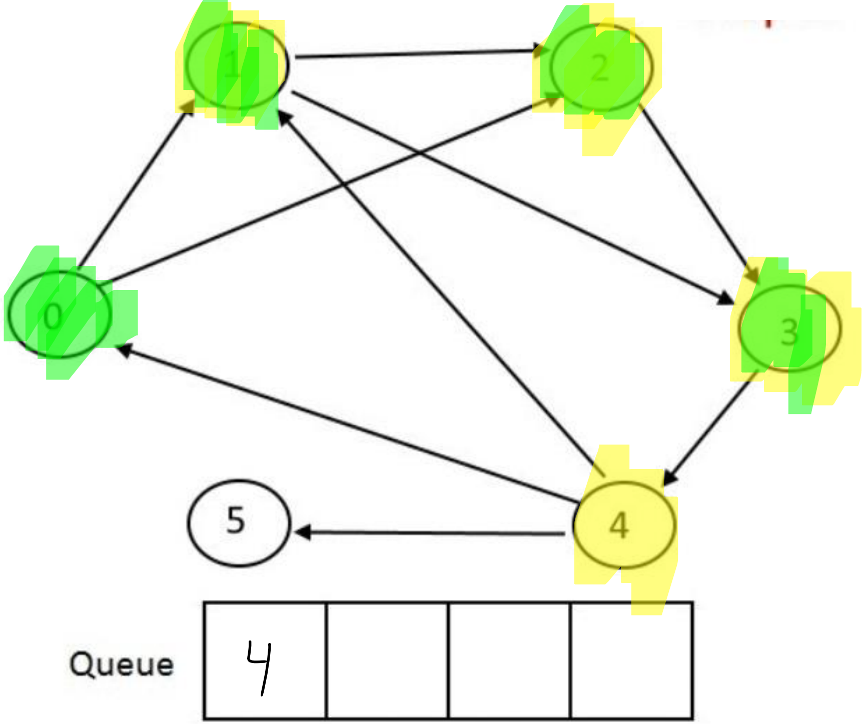
В ширину: BFS

- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
    - a. Берем вершину из очереди и помечаем ее как пройдённую
    - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
    - c. Удаляем из очереди пройденную вершину



В ширину: BFS

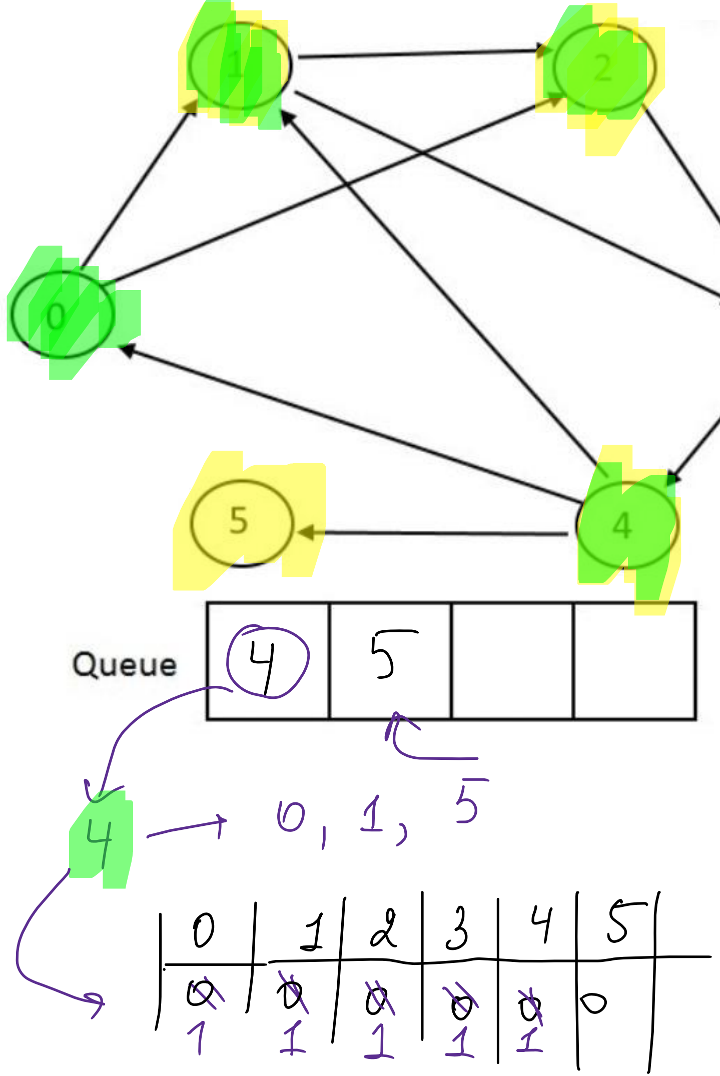
- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
  - a. Берем вершину из очереди и помечаем ее как пройдённую
  - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
  - c. Удаляем из очереди пройденную вершину



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

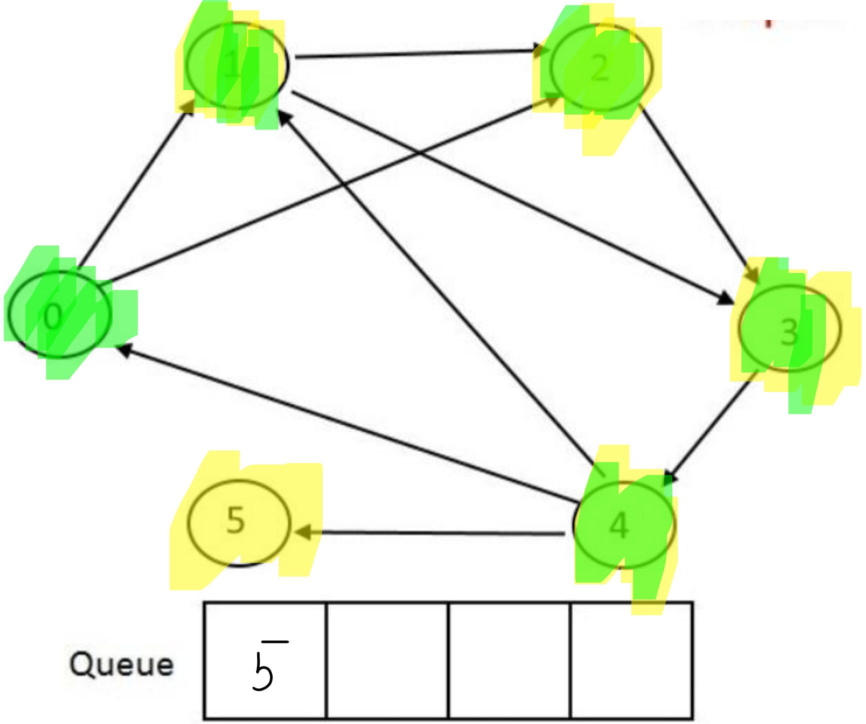
В ширину: BFS

- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
  - a. Берем вершину из очереди и помечаем ее как пройдённую
  - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
  - c. Удаляем из очереди пройденную вершину



В ширину: BFS

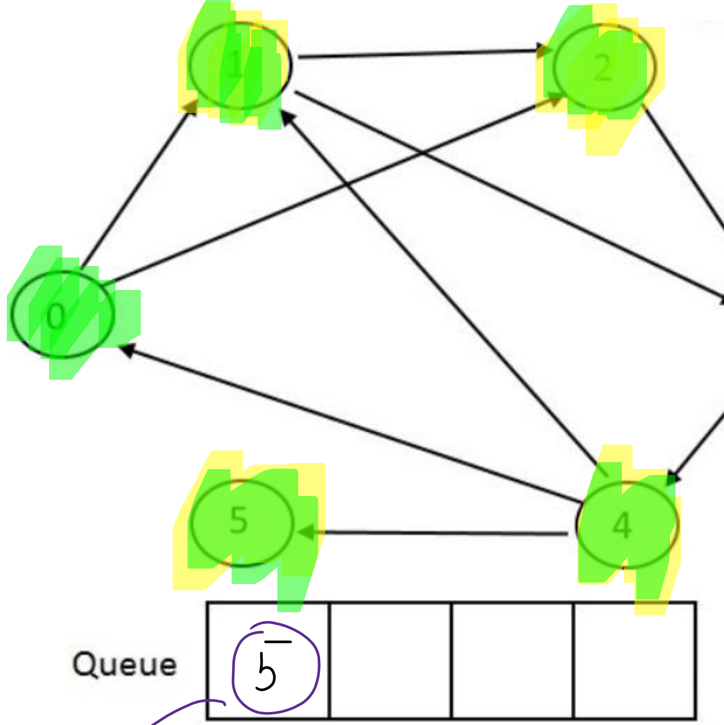
- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
  - a. Берем вершину из очереди и помечаем ее как пройдённую
  - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
  - c. Удаляем из очереди пройденную вершину



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

В ширину: BFS

- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
    - a. Берем вершину из очереди и помечаем ее как пройдённую
    - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
    - c. Удаляем из очереди пройденную вершину

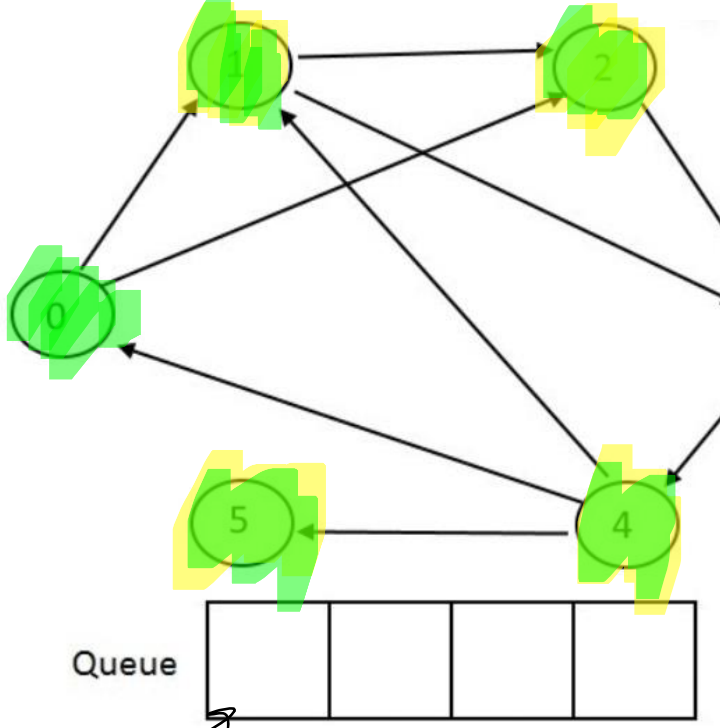


5 → ϕ

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

В ширину: BFS

- 1. Помечаем все вершины как не пройденные
- 2. Добавляем в очередь стартовую вершину
- 3. В цикле: (пока очередь не пустая)
  - a. Берем вершину из очереди и помечаем ее как пройдённую
  - b. Добавляем в очередь все смежные с ней вершины, которые не пройденные
  - c. Удаляем из очереди пройденную вершину



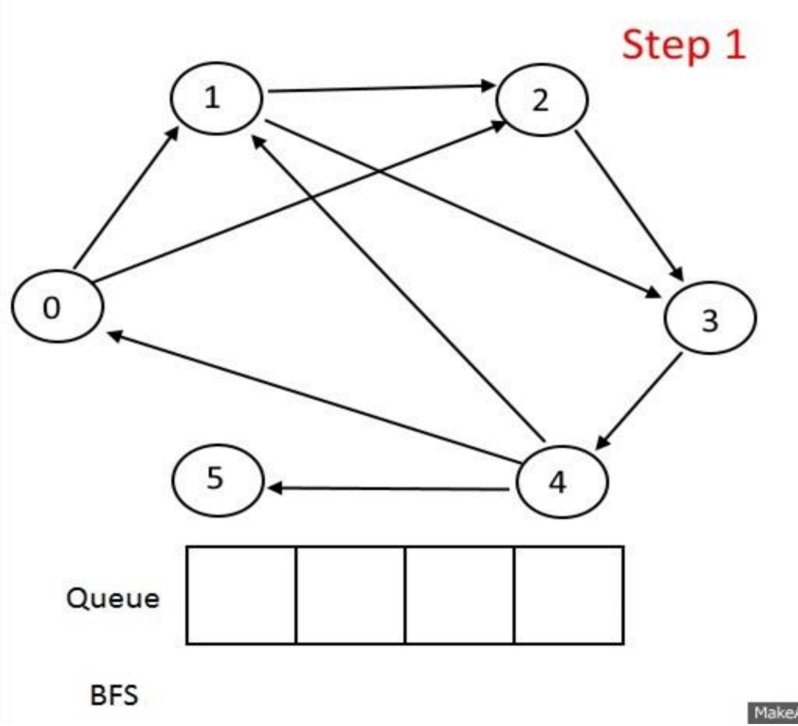
пусто → конец

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

Асимптотика: O(|V|+|E|)

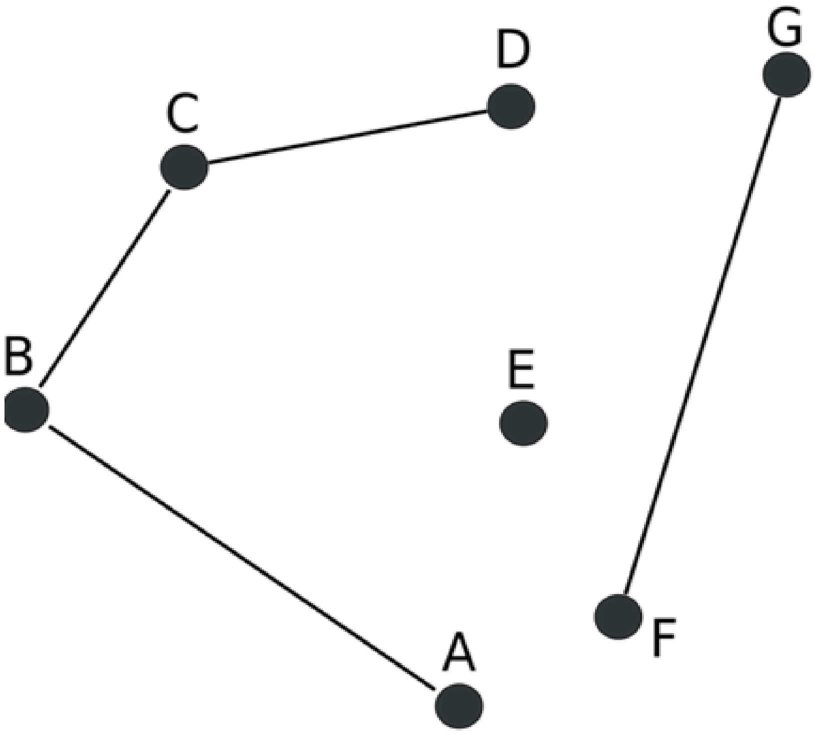
Оценим время работы для входного графа G=(V,E), где множество ребер E представлено списком смежности.

- В очередь добавляются только непосещенные вершины, поэтому каждая вершина посещается не более одного раза.
- Операции внесения в очередь и удаления из нее требуют O(1) времени, так что общее время работы с очередью составляет O(|V|) операций.
- Для каждой вершины v рассматривается не более deg(v) ребер, инцидентных ей.
- Так как ∑ deg(v)=2|E|, то время, используемое на работу с ребрами, составляет O(|E|).



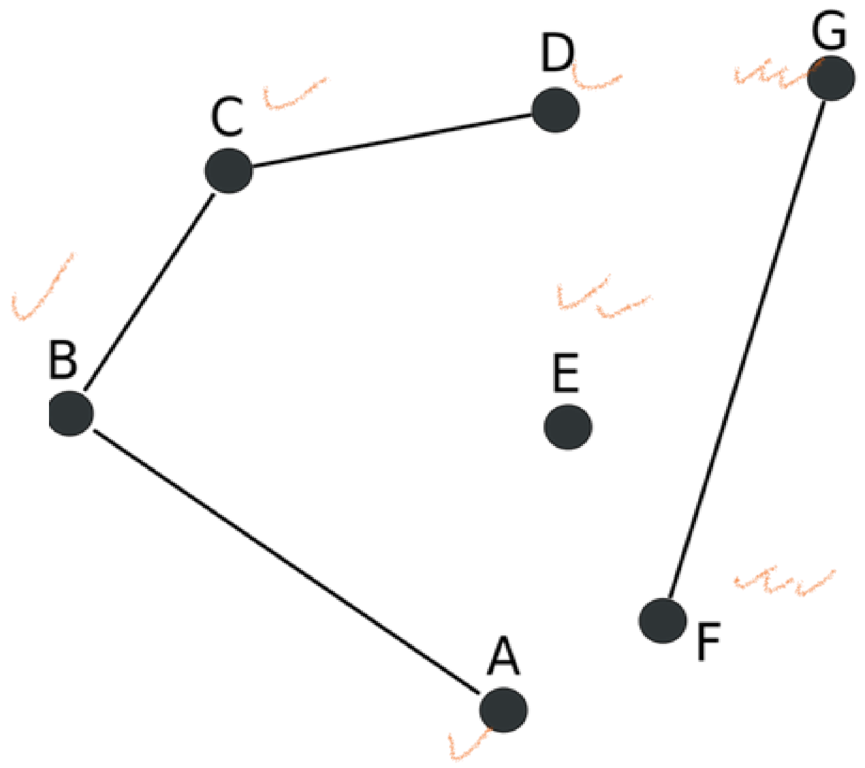
Поиск в ширину

```
int BFS(G: (V, E), source: int, destination: int):
      d = int[|V|]
      fill(d, ∞)
      d[source] = 0
      Q = Ø
      Q.push(source)
      while Q ≠Ø
          u = Q.pop()
          for v: (u, v) in E
               if d[v] == ∞
                   d[v] = d[u] + 1
                   Q.push(v)
       return d[destination]
```

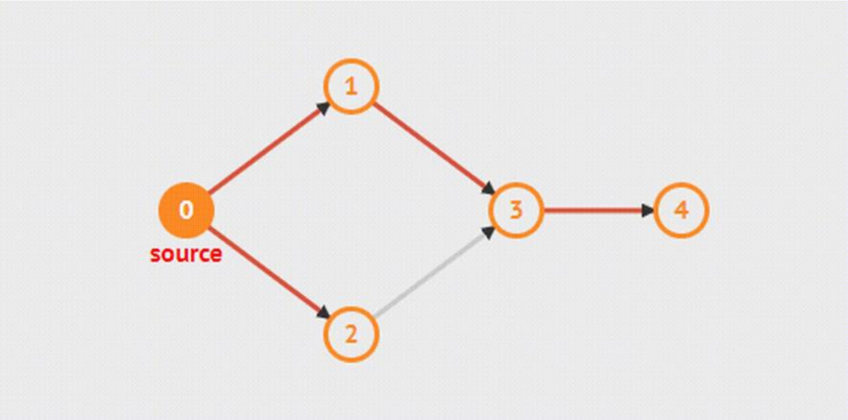


Поиск в ширину

```
int BFS(G: (V, E), source: int, destination: int):
    Q = ∅
    Q.push(source)
    while Q ≠∅
        u = Q.pop()
        for v: (u, v) in E
            if d[v] == ∞
                d[v] = d[u] + 1
                Q.push(v)
     return d[destination]

function main():
  формируем граф  G
  d = int[|V|]
  fill(d, ∞)
  d[source] = 0
  for u in V
     if d[u] == ∞
        bfsG(G, u, d)
// тут можно и к/с считать
```



Дерево поиска в ширину



Поиск в ширину также может построить дерево поиска в ширину.

- Изначально оно состоит из одного корня s.
- Когда мы добавляем непосещенную вершину в очередь, то добавляем ее и ребро, по которому мы до нее дошли, в дерево.
- Поскольку каждая вершина может быть посещена не более одного раза, она имеет не более одного родителя.
- После окончания работы алгоритма для каждой достижимой из s вершины t путь в дереве поиска в ширину соответствует кратчайшему пути от s до t в G.

Обход в ширину

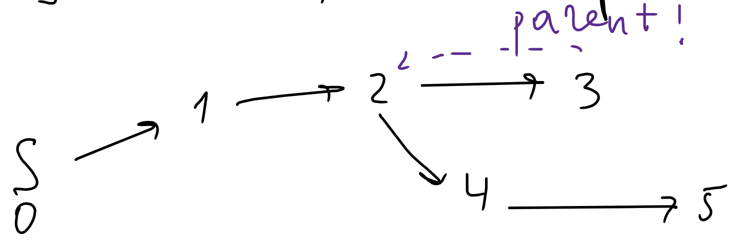
- Как найти расстояние до всех вершин от стартовой ?
  - *Использовать дополнительный массив для подсчета расстояния от стартовой до всех остальных*
  - *+ Массив предков для восстановления путей*

Обход в ширину

- <u>Как найти расстояние до всех вершин от стартовой ?</u>
  - *Использовать дополнительный массив для подсчета расстояния от стартовой до всех остальных*
  - *+ Массив предков для восстановления путей*

$d[v] = \infty$ – на старте , $d[s] = 0$ – от стартовой до самой себя

parent[] = null → какая вершина мой родитель

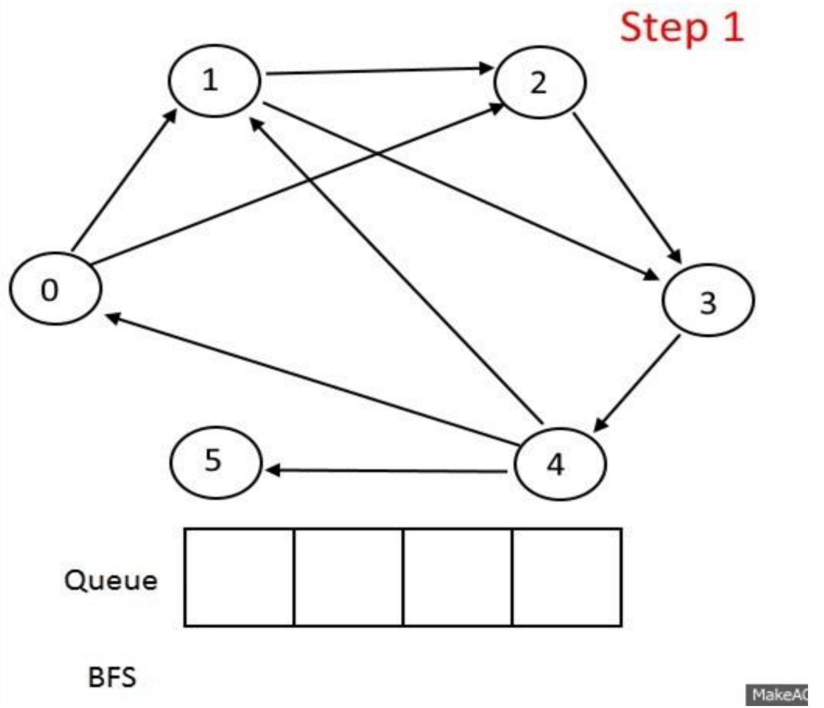


0	1	2	3	4	5
∞	0	1	2	2	4

путь от 0 до 5 : $5 \to 4 \to 2 \to 1 \to 0$ (восстановлен)

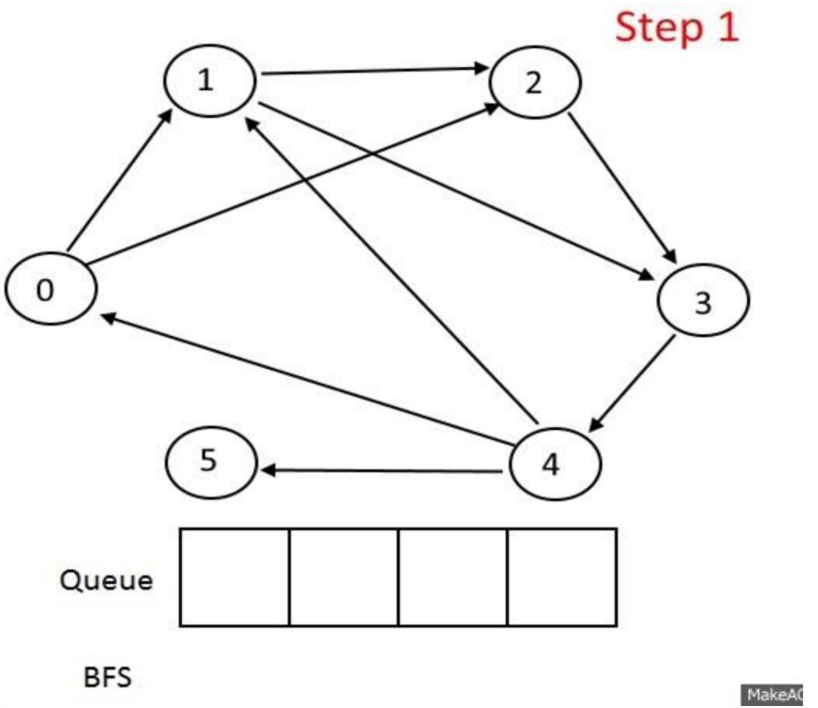
Поиск в ширину

```
int BFS(G: (V, E), source: int, destination: int):
    d, parent = int[|V|]
    fill(d, ∞)
    fill(parent, null)
    d[source] = 0
    Q = ∅
    Q.push(source)
    while Q ≠∅
        u = Q.pop()
         for v: (u, v) in E
             if d[v] == ∞
                 d[v] = d[u] + 1
                 parent[v]= u
                 Q.push(v)
     return d[destination], parent
```



Поиск в ширину

```
int BFS(G: (V, E), source: int, destination: int):
    d, parent = int[|V|]
    fill(d, ∞)
    fill(parent, null)
    d[source] = 0
    Q = Ø
    Q.push(source)
    while Q ≠Ø
        u = Q.pop()
         for v: (u, v) in E
             if d[v] == ∞
                 d[v] = d[u] + 1
                 parent[v]= u
                 Q.push(v)
     return d[destination], parent
```



	0	1	2	3	4	5
d	0	1	1	2	3	4
parent	null	0	0	1	3	4

Обход в глубину

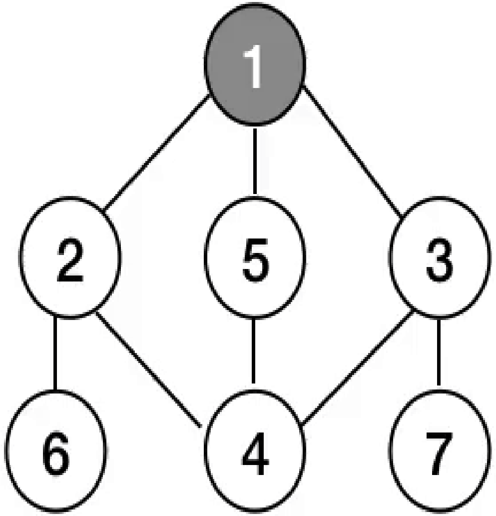
- <u>Какие идеи эффективной реализации?</u>
    - *Использовать рекурсию для прохода по графу*
    - *Использовать метки – пройдена/не пройдена*
    - *Рассматриваем в один момент времени – одну вершину*

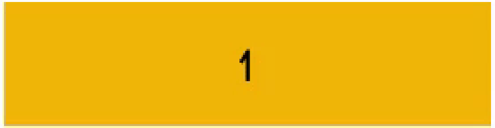
- <u>Как понять, что алгоритм закончен?</u>
    - *Все вершины помечены как пройденные, даже если несколько компонент связности*

- <u>Как лучше хранить граф?</u>
    - *Список смежности/Матрица смежности*

Обход в глубину: DFS (G(V, E))

```
DFS (G) {
    For  u из G.V do
        u.color = white
    For  u из G.V do
    If  u.color == white  then
            Visit (G, u)
}

Visit (G) {
    u.color = gray
    For  v  из G.V[u]  do
        If  v.color == white  then
            Visit (G, v)
    u.color = black
}
```





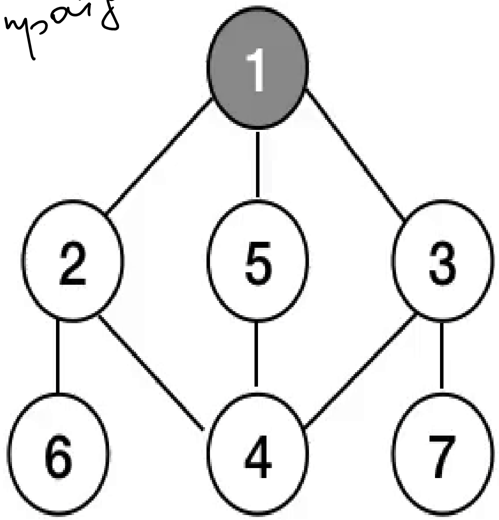
white → не пройдена
gray → в процессе dfs
black → пройдена

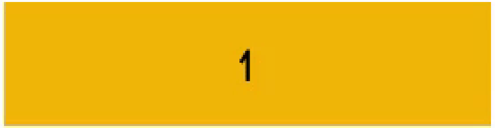
Обход в глубину: DFS (G(V, E))

```
DFS (G) {
     For  u из G.V do
         u.color = white
     For  u из G.V do
     If  u.color == white  then
             Visit (G, u)
}

Visit (G) {
      u.color = gray
      For  v  из G.V[u]  do
          If  v.color == white  then
              Visit (G, v)
      u.color = black
}
```

сперва все не пройдены

все пройдём



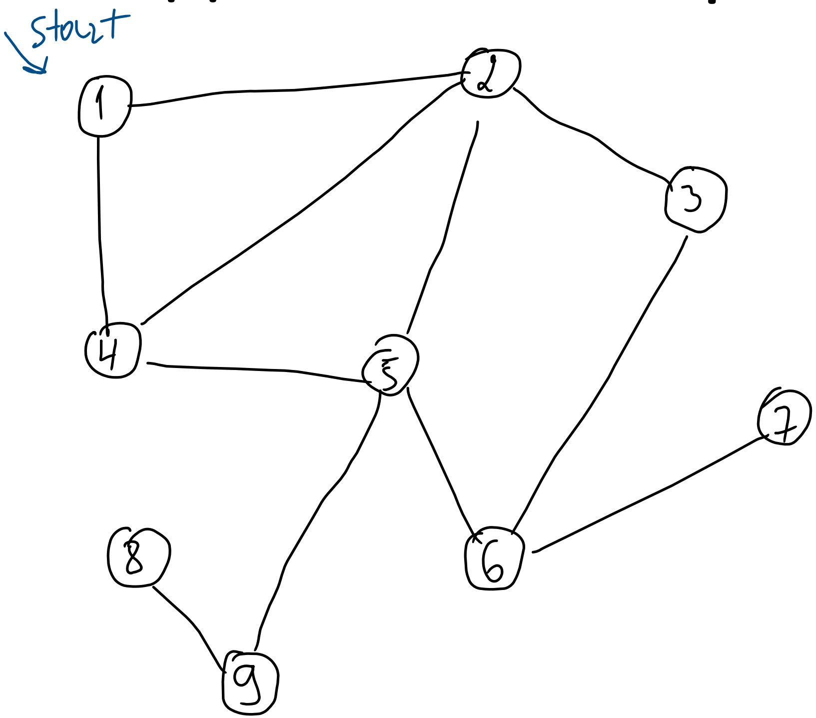


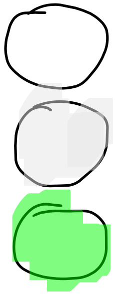
white → не пройдена

gray → в процессе dfs

black → пройдена

Обход в ГЛУБИНУ: три цвета





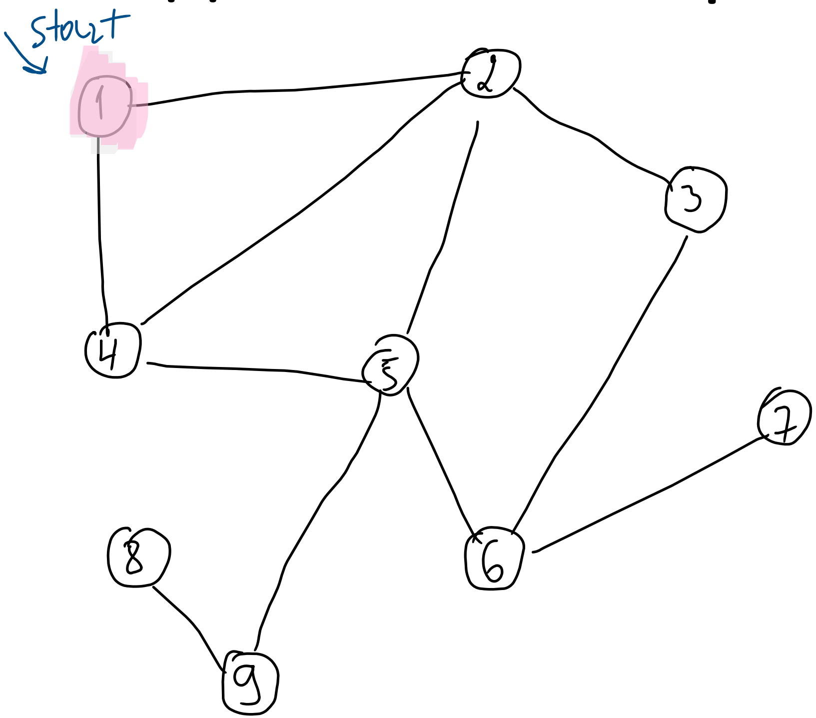
не пройдена

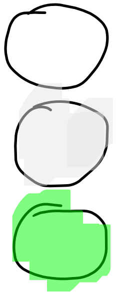
в процессе прохода

пройдена

```
DFS (G) {
    For  u из G.V do
        u.color = white
    For  u из G.V do
    If  u.color == white  then
            Visit (G, u)
}

Visit (G) {
    u.color = gray
    For  v  из G.V[u]  do
        If  v.color == white  then
            Visit (G, v)
    u.color = black
}
```

Обход в ГЛУБИНУ: три цвета





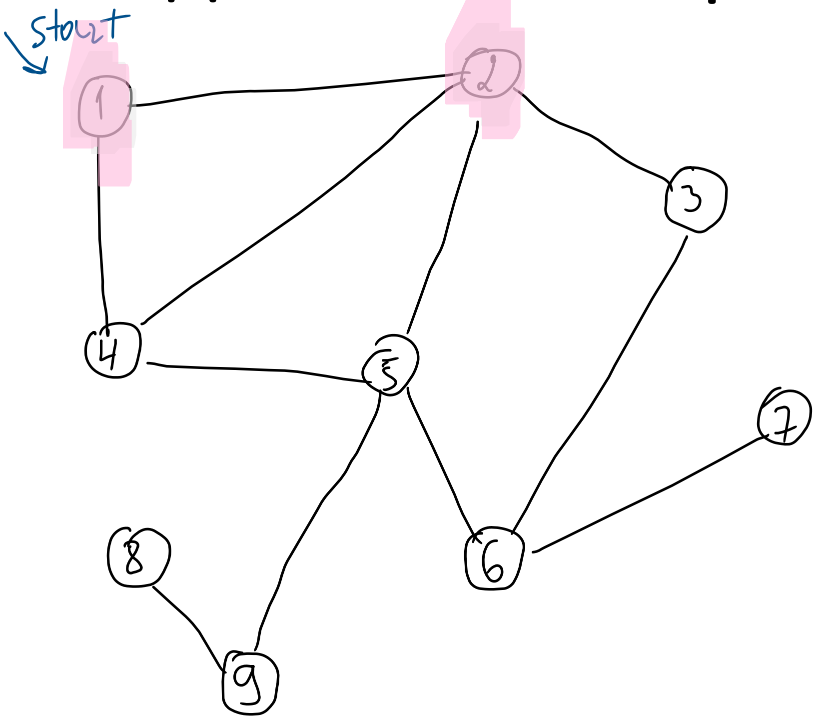
не пройдена

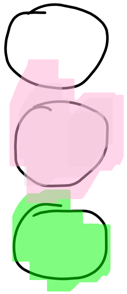
в процессе прохода

пройдена

```
DFS (G) {
    For u из G.V do
        u.color = white
    For u из G.V do
    If u.color == white then
            Visit (G, u)
}

Visit (G) {
    u.color = gray
    For v из G.V[u] do
        If v.color == white then
            Visit (G, v)
    u.color = black
}
```

Обход в ГЛУБИНУ: три цвета





не пройдена

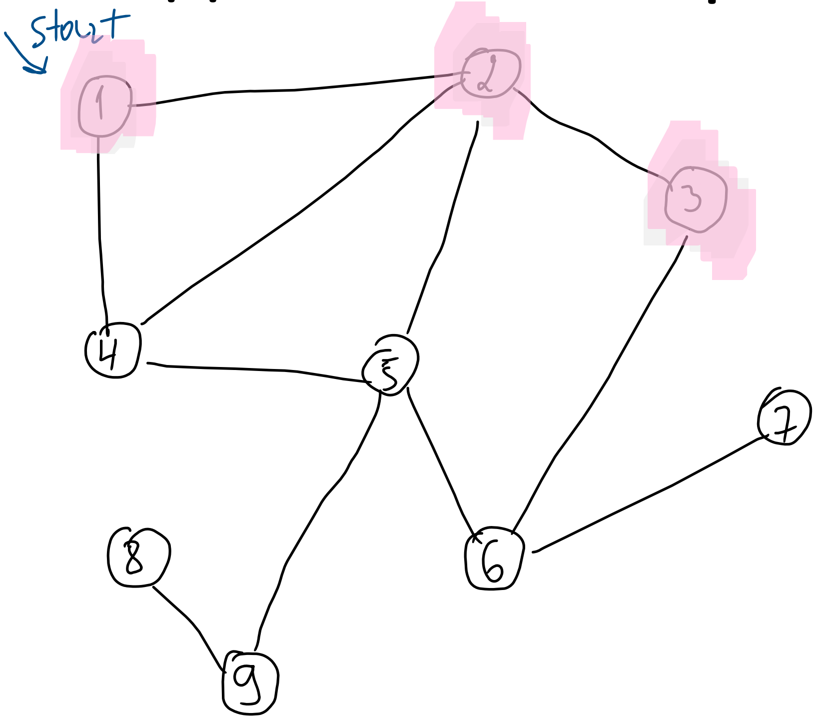
в процессе прохода

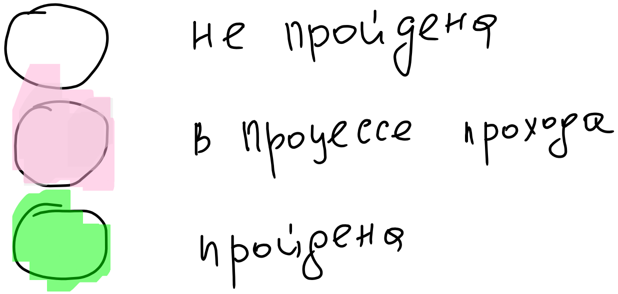
пройдена

! Следующая всегда с меньшим номером

```
DFS (G) {
    For  u из G.V do
        u.color = white
    For  u из G.V do
    If  u.color == white  then
            Visit (G, u)
}

Visit (G) {
    u.color = gray
    For  v  из G.V[u]  do
        If  v.color == white  then
            Visit (G, v)
    u.color = black
}
```

Обход в ГЛУБИНУ: три цвета

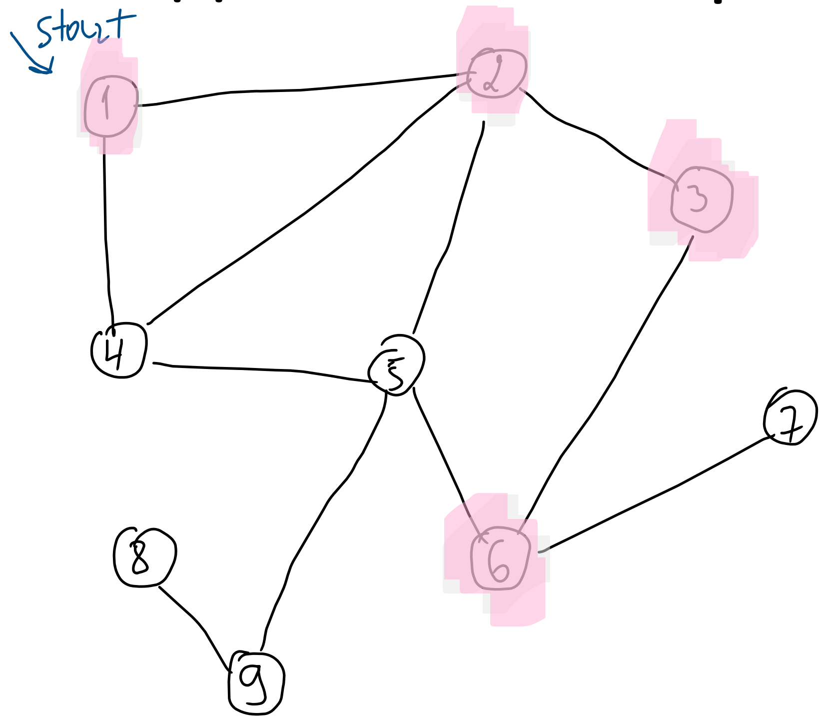




! Следующая всегда с меньшим номером

```
DFS (G) {
    For  u из G.V do
        u.color = white
    For  u из G.V do
    If  u.color == white  then
            Visit (G, u)
}

Visit (G) {
    u.color = gray
    For  v  из G.V[u]  do
        If  v.color == white  then
            Visit (G, v)
    u.color = black
}
```

Обход в ГЛУБИНУ: три цвета



не пройдена

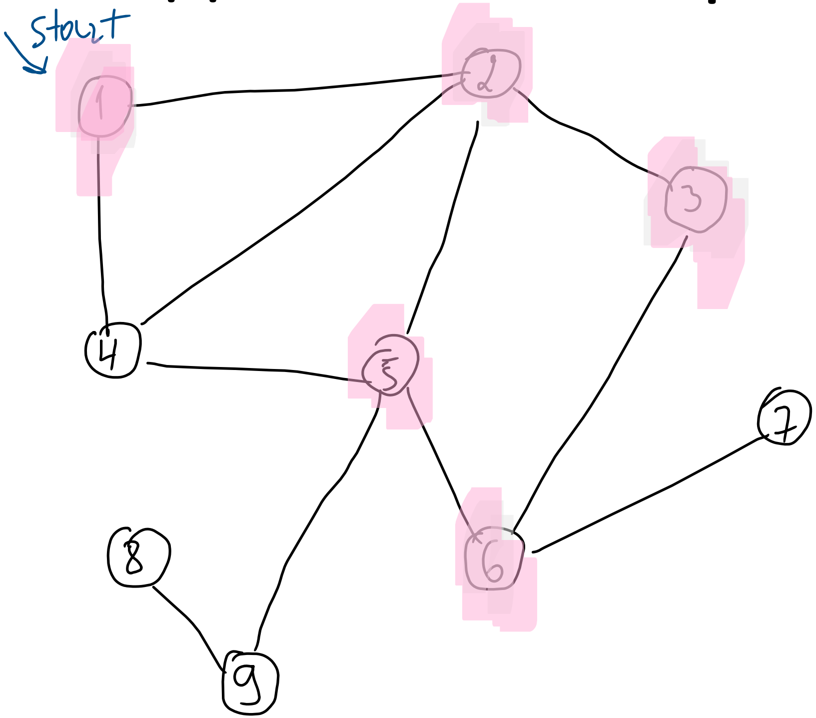
в процессе прохода

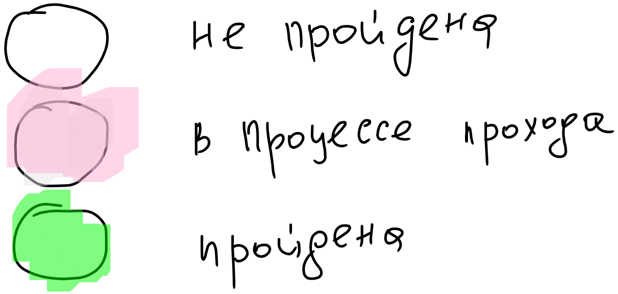
пройдена

! Следующая всегда с меньшим номером

```
DFS (G) {
    For  u из G.V do
        u.color = white
    For  u из G.V do
    If  u.color == white  then
            Visit (G, u)
}

Visit (G) {
    u.color = gray
    For  v  из G.V[u]  do
        If  v.color == white  then
            Visit (G, v)
    u.color = black
}
```

Обход в ГЛУБИНУ: три цвета

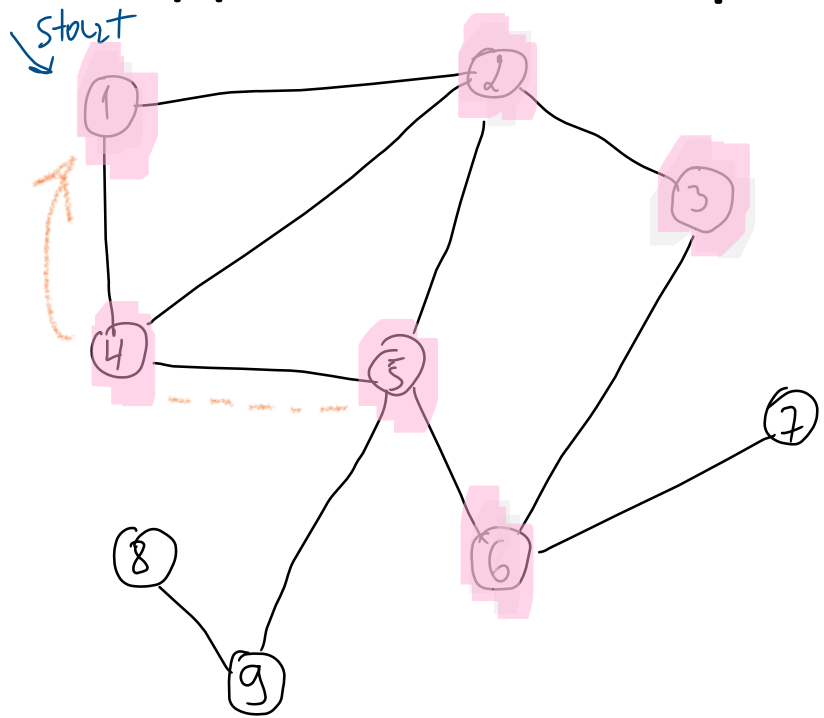




(!) Следующая всегда с меньшим номером

```
DFS (G) {
    For  u из G.V do
        u.color = white
    For  u из G.V do
    If  u.color == white  then
            Visit (G, u)
}

Visit (G) {
    u.color = gray
    For  v  из G.V[u]  do
        If  v.color == white  then
            Visit (G, v)
    u.color = black
}
```

Обход в ГЛУБИНУ: три цвета



не пройдена

в процессе прохода

пройдена

! Следующая всегда с меньшим номером

!

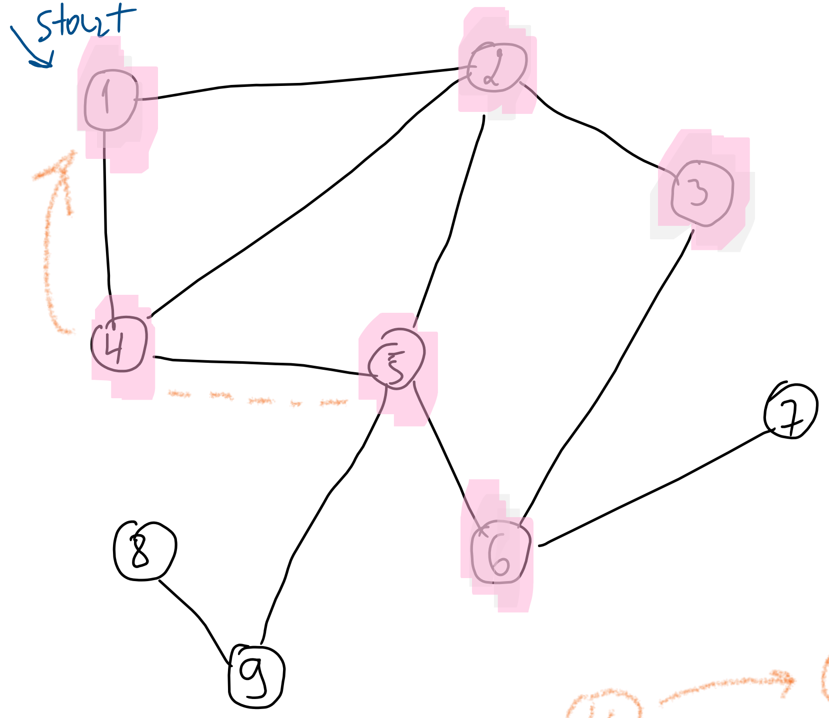
из 5-ой в 4-ую

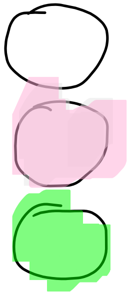
т.е. красим

ребра в обходе

в моменте!

Обход в ГЛУБИНУ: три цвета





не пройдена

в процессе прохода

пройдена

! Следующая всегда с меньшим номером

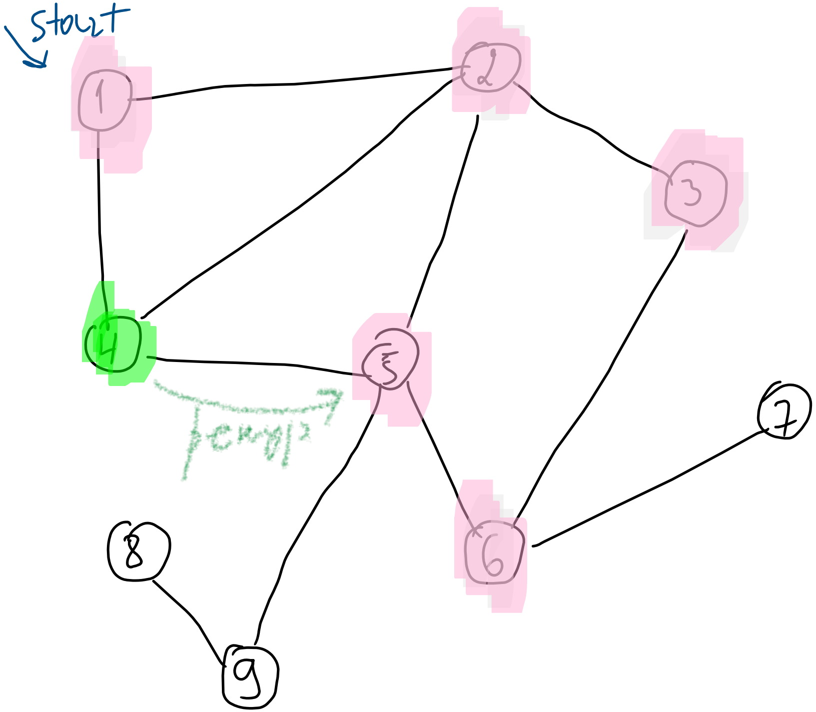
из 5ой в 4ую

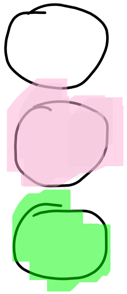
т.е. красим предка в обходе в момент?!

из серой в серую ⟹

цикл

Обход в ГЛУБИНУ: три цвета





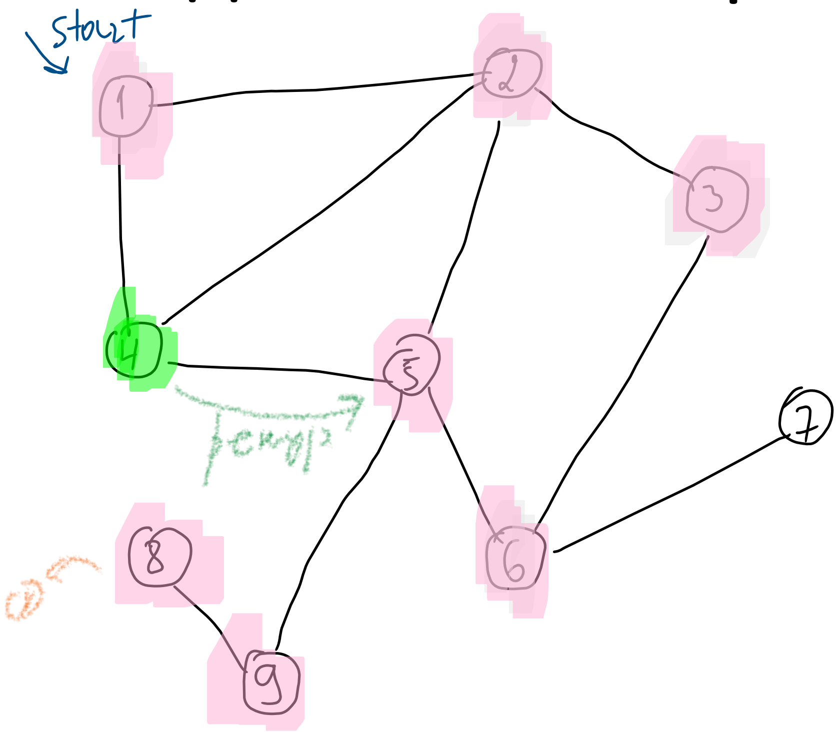
не пройдена

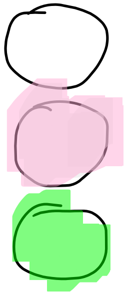
в процессе прохода

пройдена

(!) Следующая всегда с меньшим номером

Обход в ГЛУБИНУ: три цвета





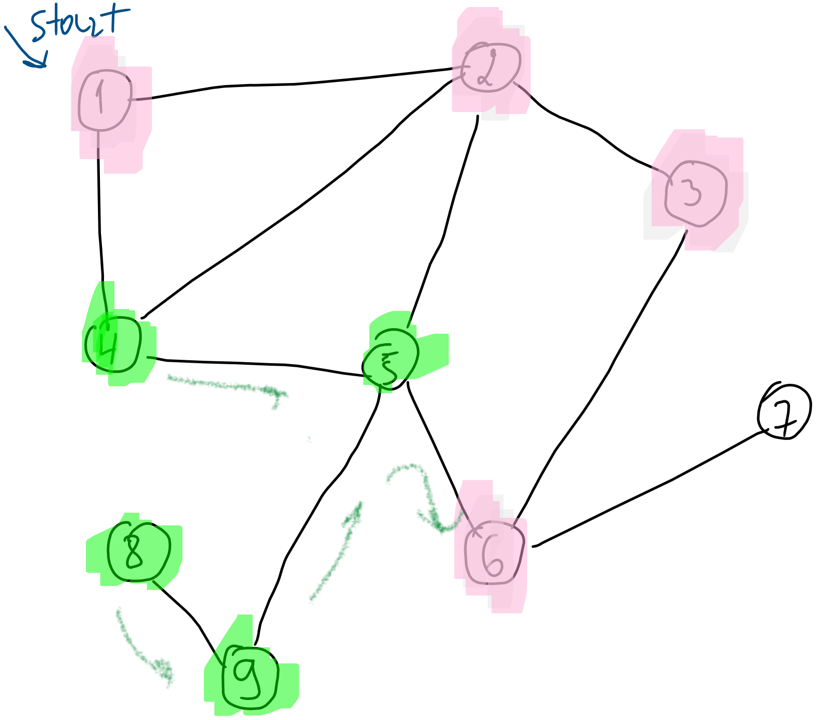
не пройдена

в процессе прохода

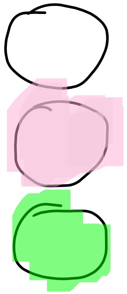
пройдена

(!) Следующая всегда с меньшим номером

Обход в ГЛУБИНУ: три цвета



рекур. возврат



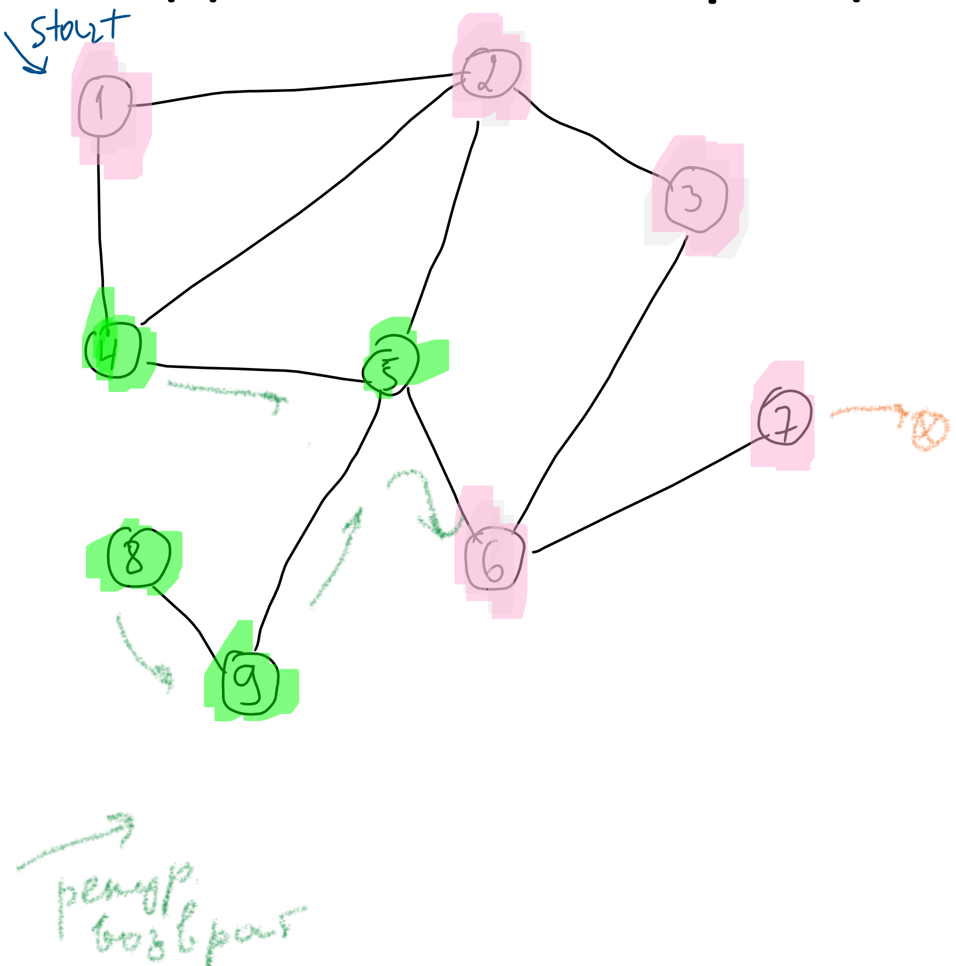
не пройдена

в процессе прохода

пройдена

! Следующая всегда с меньшим номером

Обход в ГЛУБИНУ: три цвета



не пройдена

в процессе прохода

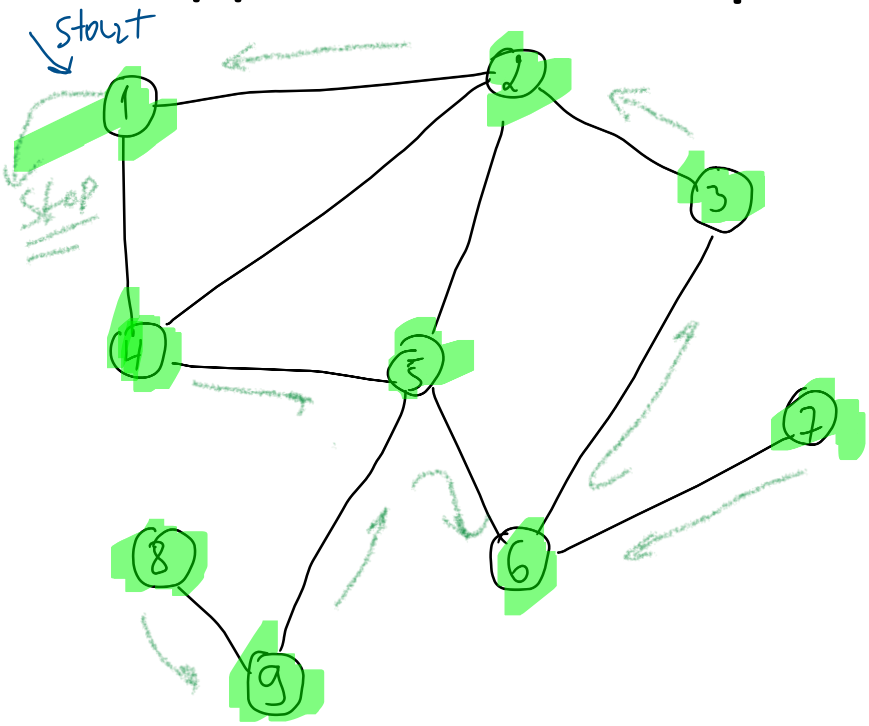
пройдена

! Следующая всегда с меньшим номером

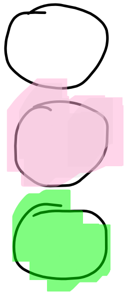
4 8 9 5 7 6 3 2 1

рез. обхода

Обход в ГЛУБИНУ: три цвета



режим возврат



не пройдена

в процессе прохода

пройдена

! Следующая всегда с меньшим номером

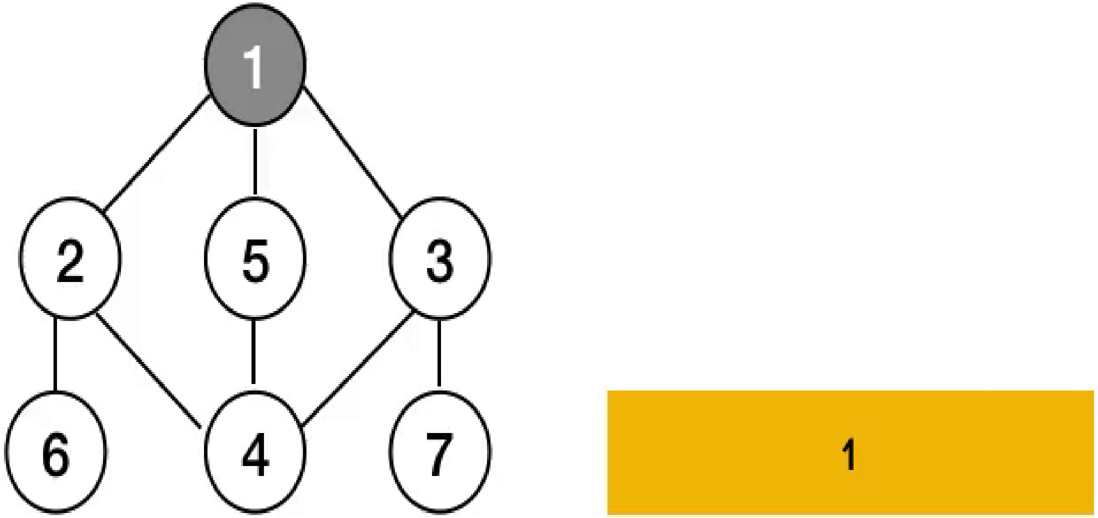
4 8 9 5 7 6 3 2 1

рез. обхода

Асимптотика: O(|V|+|E|)

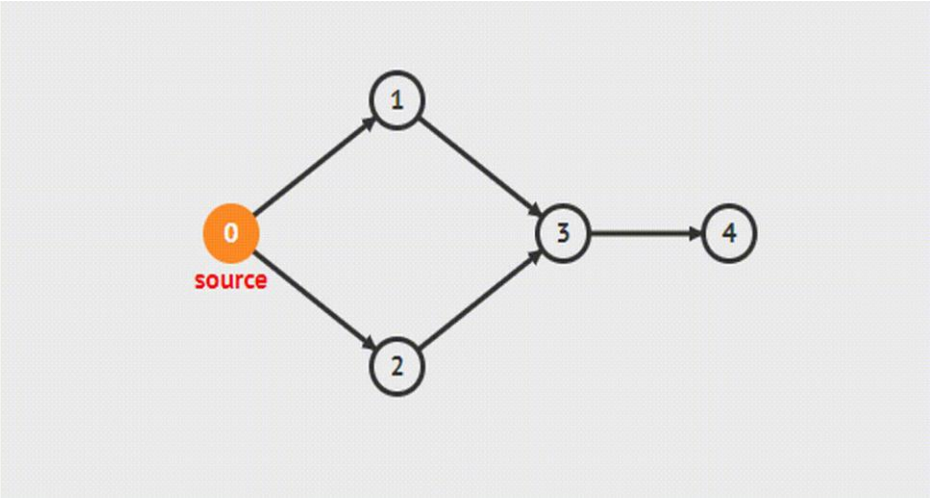
Оценим время работы для входного графа G=(V,E), где множество ребер E представлено списком смежности.

- Просматриваем все вершины.
- Просматриваем все смежные ребра у вершины для завершения обхода или покраски в черный (пройденная).



Использование обхода в глубину для поиска цикла

- В случае **ориентированного графа** произведём серию обходов.
- То есть из каждой вершины, в которую мы ещё ни разу не приходили, запустим поиск в глубину, который при входе в вершину будет красить её в серый цвет, а при выходе из нее — в чёрный.
- И, если алгоритм пытается пойти **в серую вершину,** то это означает, что цикл найден.



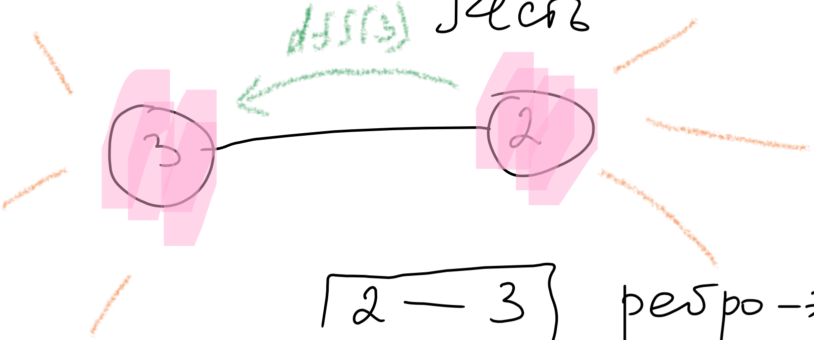
серая серая = цикл

Использование обхода в глубину для поиска цикла

```
// color — массив цветов, изначально все вершины белые
func dfs(v: vertex):                  // v — вершина, в которой мы сейчас находимся
    color[v] = grey
    for (u: vu ∈ E)
        if (color[u] == white)
            dfs(u)
        if (color[u] == grey)
            print( «цикл есть» )           // вывод ответа
    color[v] = black
```

Уже есть

```
DFS (G) {
    For  u из G.V do
        u.color = white
    For  u из G.V do
    If  u.color == white  then
            Visit (G, u)
}

Visit (G) {
    u.color = gray
    For  v  из G.V[u]  do
        If  v.color == white  then
            Visit (G, v)
    u.color = black
}
```



$\boxed{2 — 3}$ ребро – это не цикл

Использование обхода в глубину для поиска цикла

```
// color — массив цветов, изначально все вершины белые
func dfs(v: vertex):                // v — вершина, в которой мы сейчас находимся
    color[v] = grey
    for (u: vu ∈ E)
        if (color[u] == white)
            dfs(u)
        if (color[u] == grey)
            print( «цикл есть» )          // вывод ответа
    color[v] = black
```

```
DFS (G) {
    For  u из G.V do
        u.color = white
    For  u из G.V do
    If  u.color == white  then
            Visit (G, u)
}

Visit (G) {
    u.color = gray
    For  v  из G.V[u]  do
        If  v.color == white  then
            Visit (G, v)
    u.color = black
}
```

Как восстановить весь цикл?

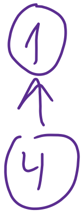
Использование обхода в глубину для поиска цикла

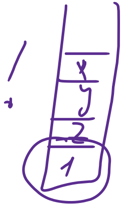
```
// color — массив цветов, изначально все вершины белые
func dfs(v: vertex):                 // v — вершина, в которой мы сейчас находимся
    color[v] = grey
    for (u: vu ∈ E)
        if (color[u] == white)
            dfs(u)
        if (color[u] == grey)
            print( «цикл есть» )          // вывод ответа
    color[v] = black
```

1) массив предков parent[] = null и заполняем по мере прохода!

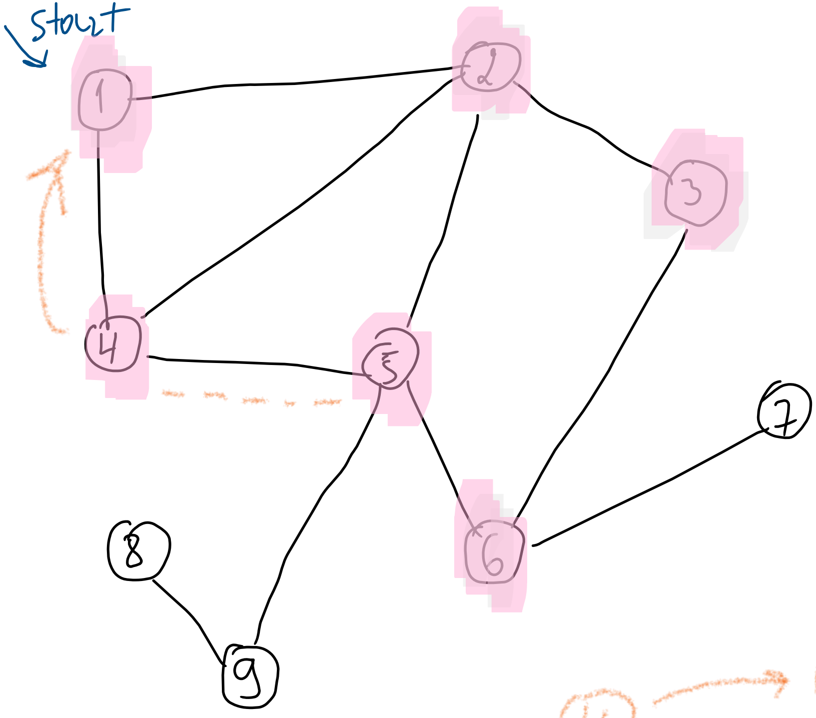
Как восстановить весь цикл? =>

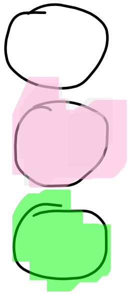
2) Используем стек серых вершин!





Обход в ГЛУБИНУ: три цвета





не пройдена

в процессе прохода

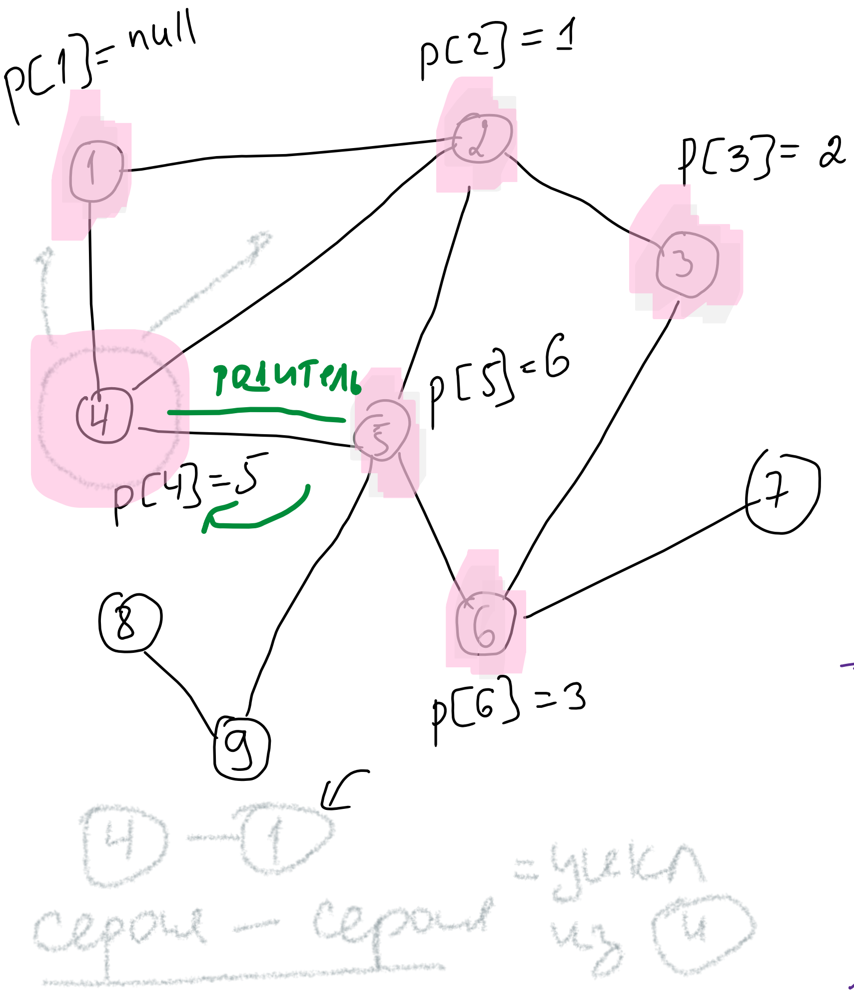
пройдена

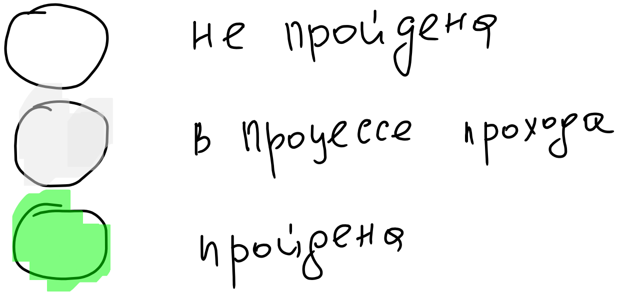
! Следующая всегда с меньшим номером

из 5-ой в 4-ую
т.е. красим ребра в обходе в иерархию!

из серой в серую ⟹

цикл





(!) Следующая всегда с меньшим номером

4 ⇒ p[4] = 5

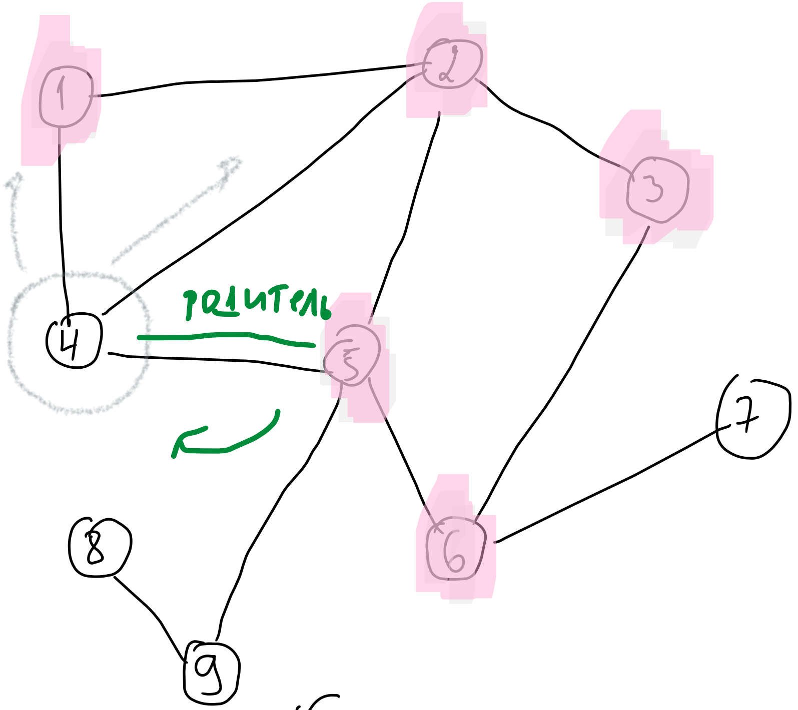
5 => p[5] – 6

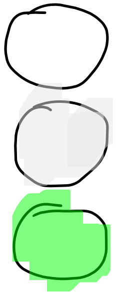
6 => p[6] = 3

3 => p[3] = 2

2 => p[2] = 1

1 → 1 дошли до 4 – 1



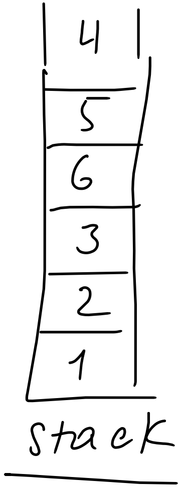


не пройдена

в процессе прохода

пройдена

(!) Следующая всегда с меньшим номером

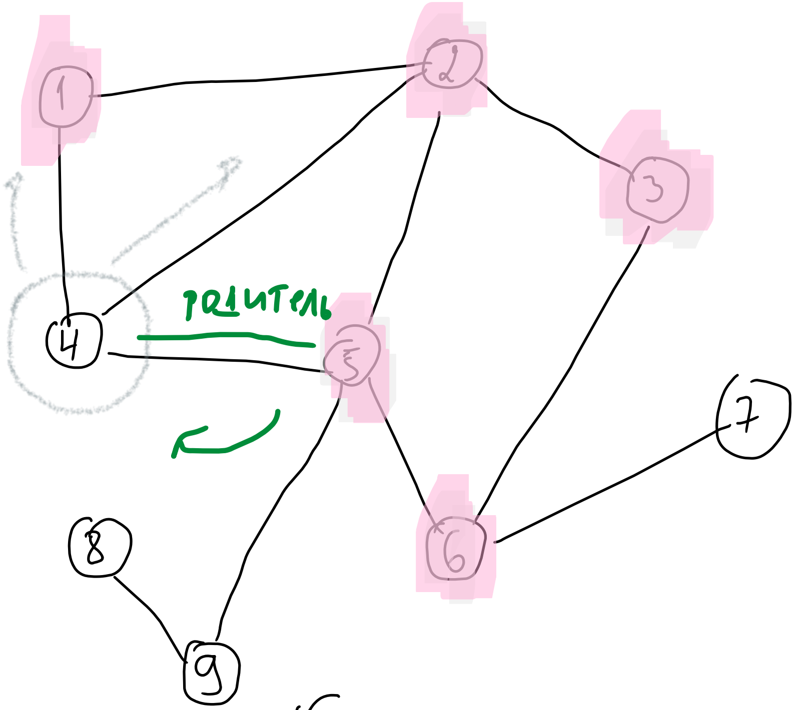


4 — 1 цикл

4 → stack.pop

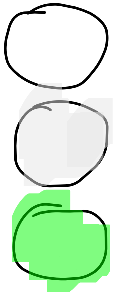
4 — 1 = цикл

серая — серая из 4



4 — 1 = цикл

серая — серая из 4

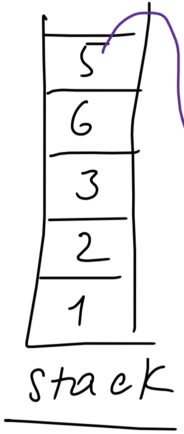


не пройдена

в процессе прохода

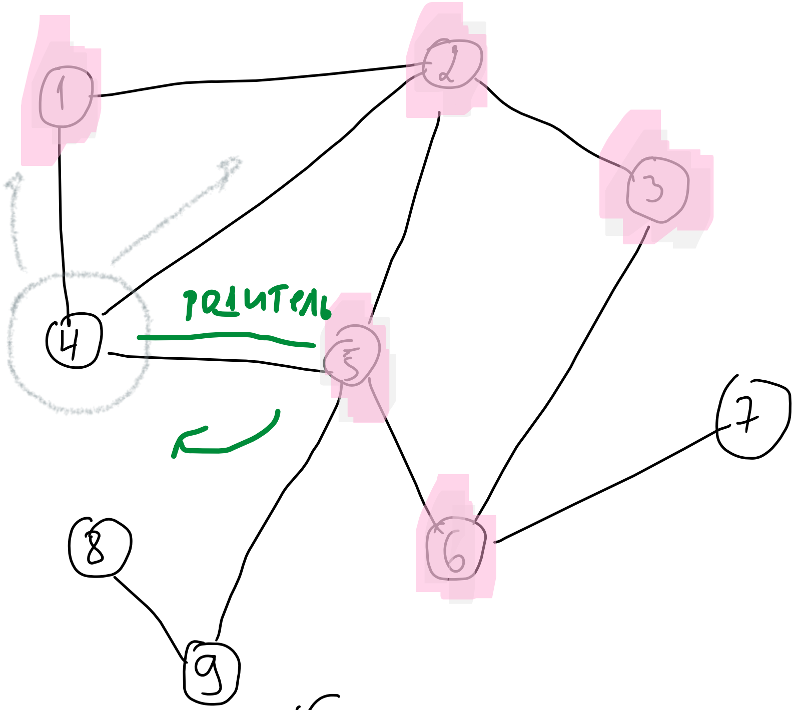
пройдена

! Следующая всегда с меньшим номером



4 — 1 цикл

4 → 5



не пройдена

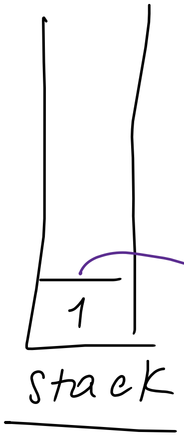
в процессе прохода

пройдена

! Следующая всегда с меньшим номером

4 — 1 цикл

4 → 5 → 6 → 3 → 2 → 1 ≡



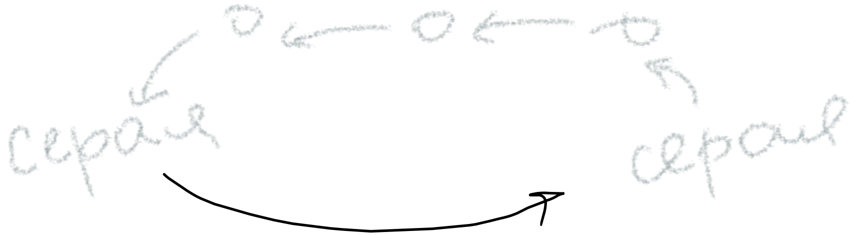
4 — 1 = цикл

серая — серая из 4

Использование обхода в глубину для поиска цикла

- Как найти другие циклы?
- Как найти цикл в цикле?
- Как найти все возможные

  циклы ?

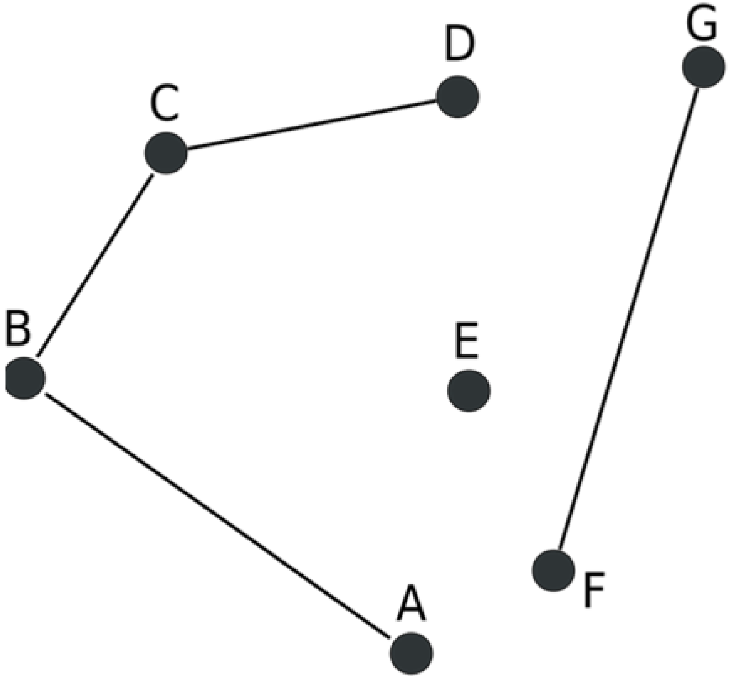


= цикл

Поиск компонент связности графа

Алгоритм:

- 1. Помечаем все вершины как не пройденные
- 2. Цикл пока есть не пройденные вершины
    - a. Запускаем обход в глубину от вершины
        - i. Все пройденные вершины собираем в первую компоненту
    - b. Ищем не пройденную вершину
- 3. Выводим все компоненты графа



Поиск компонент связности графа

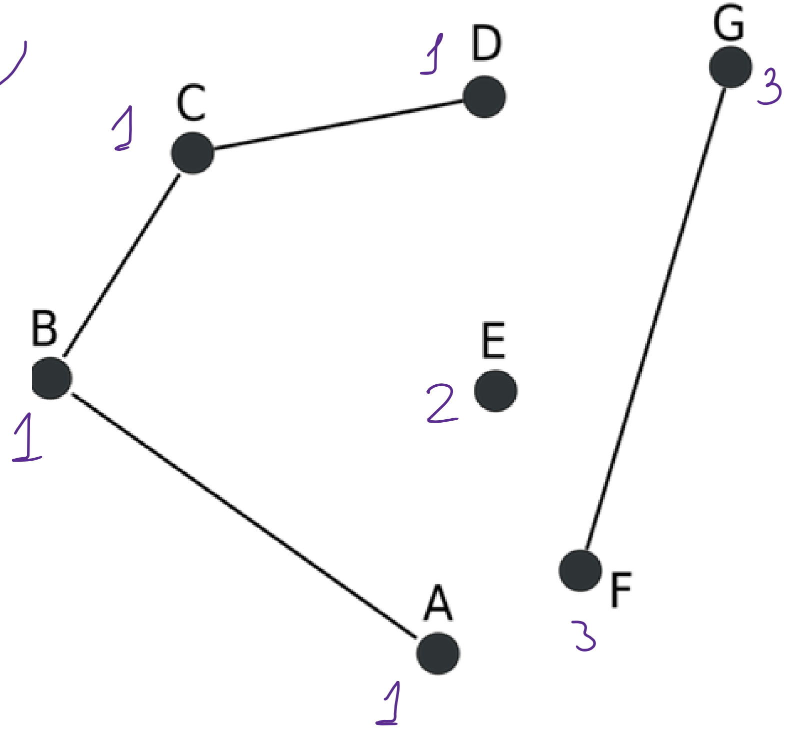
count = 1

Алгоритм:

- 1. Помечаем все вершины как не пройденные
- 2. Цикл пока есть не пройденные вершины
  - a. Запускаем обход в глубину от вершины
    - i. Все пройденные вершины собираем в первую компоненту
  - b. Ищем не пройденную вершину

count++

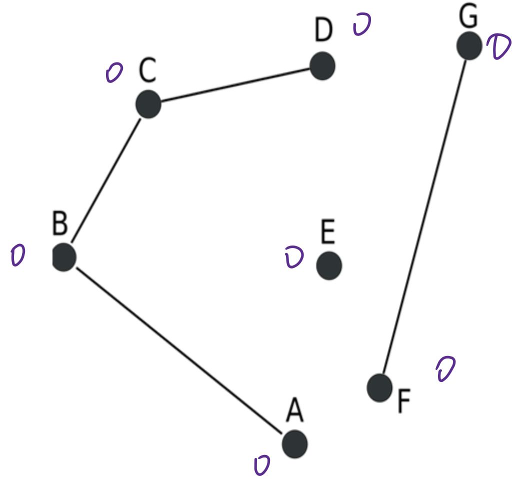
- 3. Выводим все компоненты графа



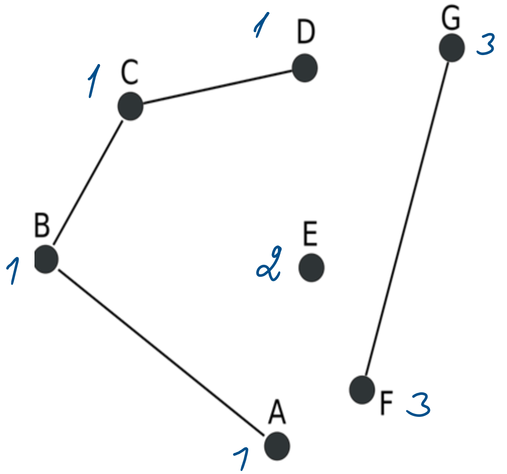
```
DFS (G) {
    For  u из G.V do
        u.color = white
        u.component=0
    count=1
    For  u из G.V do
        If  u.color == white  then
            Visit (G, u, count)
        count++
}
```

```
Visit (G, u, count) {
    u.color = gray
    u.component=count
    For  v  из G.V[u]  do
        If  v.color == white  then
            Visit (G, v)
    u.color = black
}
```

```
DFS (G) {
    For u из G.V do
        u.color = white
        u.component=0
    count=1
    For u из G.V do
        If u.color == white then
            Visit (G, u, count)
        count++
}
```




```
Visit (G, u, count) {
    u.color = gray
    u.component=count
    For v из G.V[u] do
        If v.color == white then
            Visit (G, v)
    u.color = black
}
```



Поиск компонент связности

- Как должен быть задан граф ?

- Можно ли использовать обход в ширину?

- Как найти компоненты слабой связности?

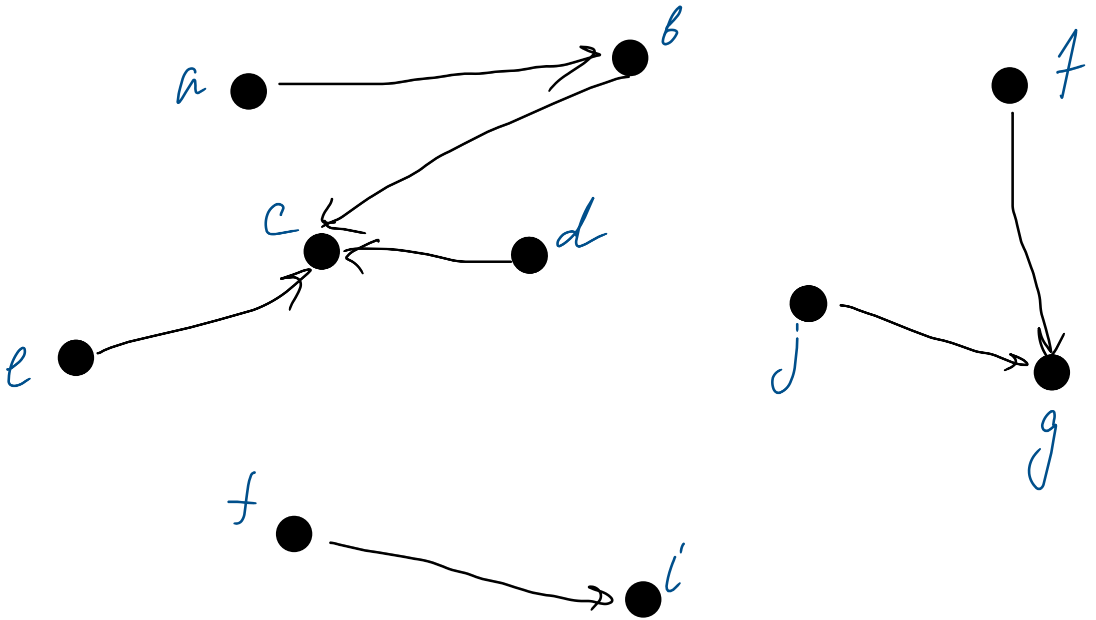
- Сложность какая?

Поиск компонент связности

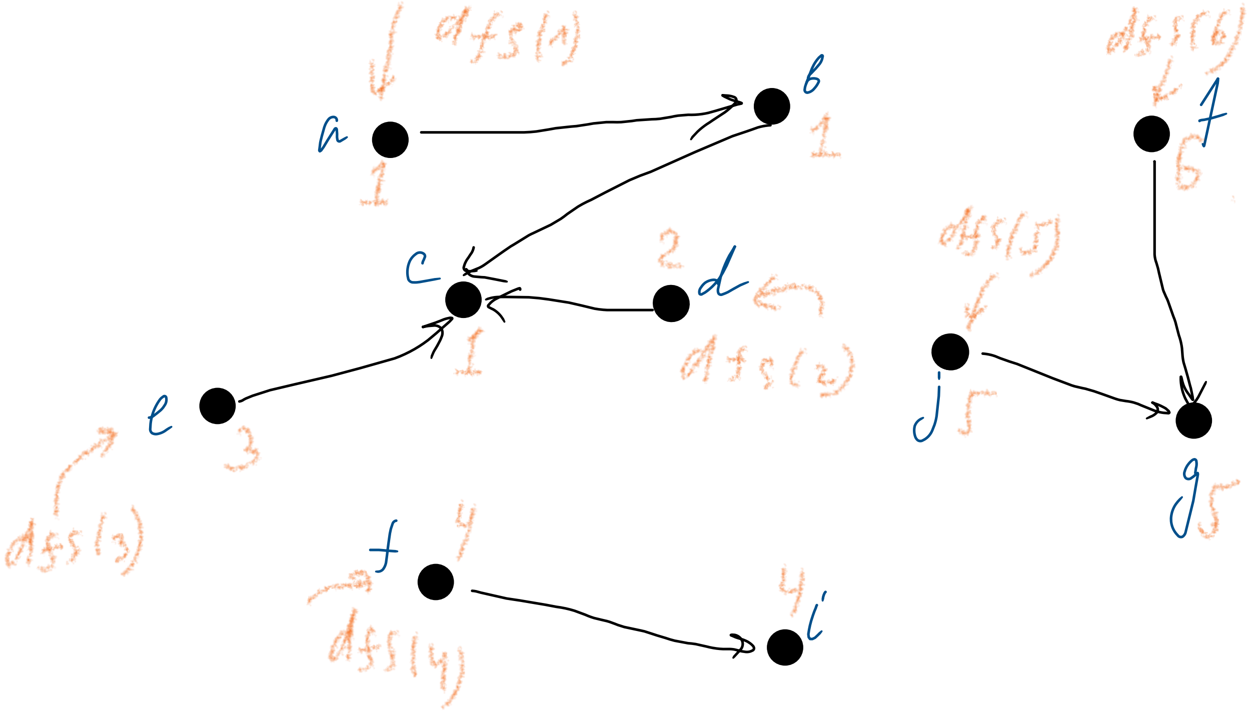
- Как должен быть задан граф ?
  - *Согласно реализации обхода*
  - *При необходимости выделяем массив с индексами вершин под запись номера компоненты*

- Можно ли использовать обход в ширину?
  - *Дополнив реализацию обхода в ширину, аналогично обходу в глубину можно найти все компоненты: просмотр всех пройденных на предмет еще не пройденных после каждого вызова*

- Как найти компоненты слабой связности?
  - *???*
- Сложность какая?
  - *Согласно сложности используемого обхода*

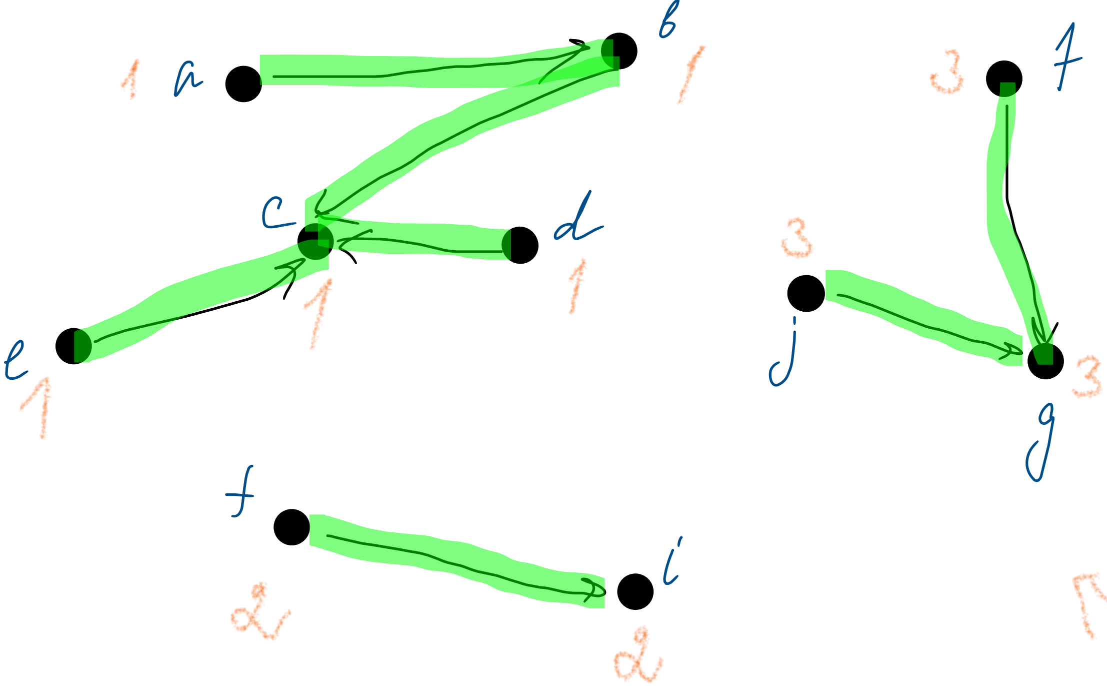


Используем обычный поиск к/с



Используем обычный поиск к/с

Не верно т.к. всего их 3



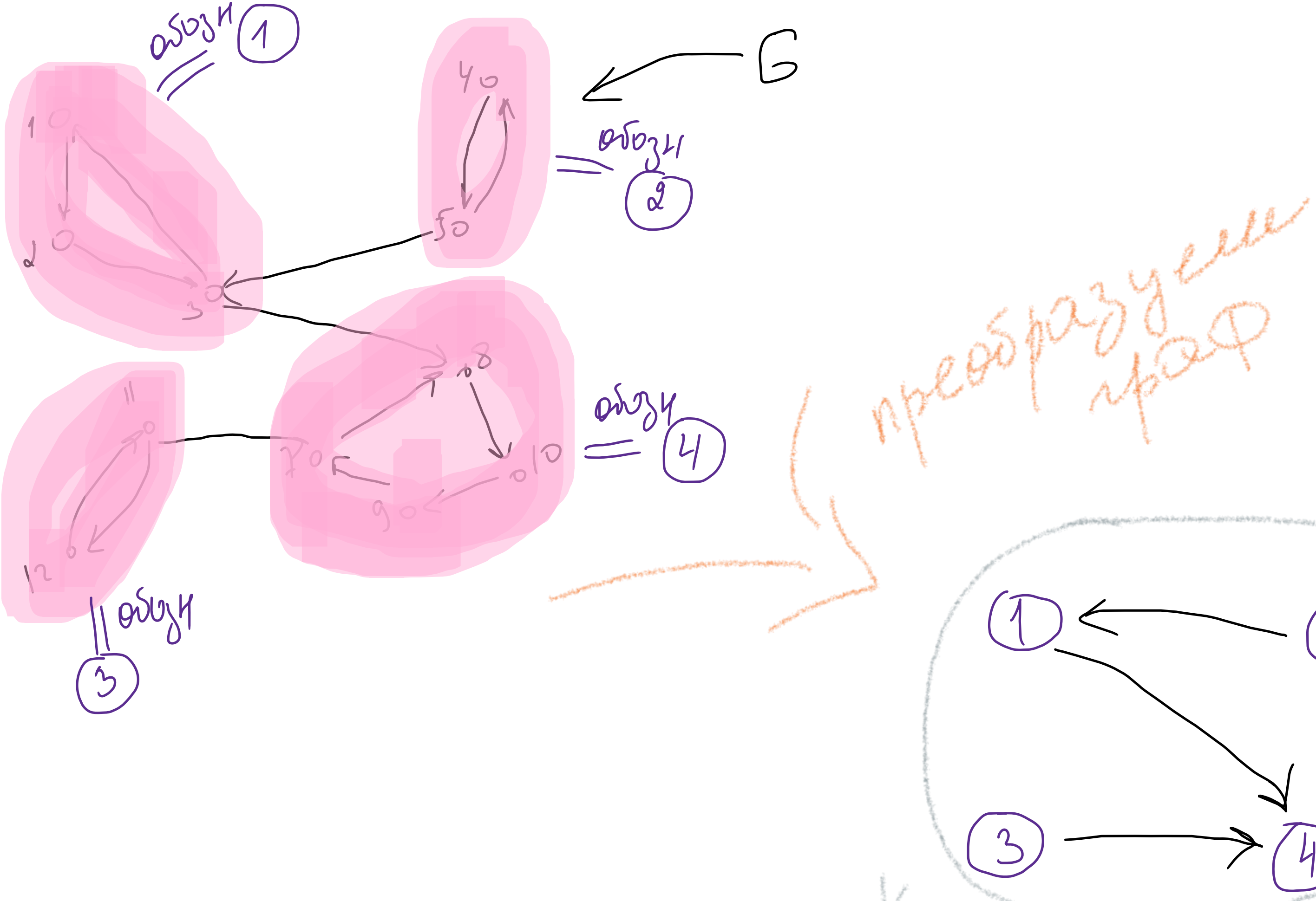
Используем обычный поиск к/с по зеленым ребрам

Верно!

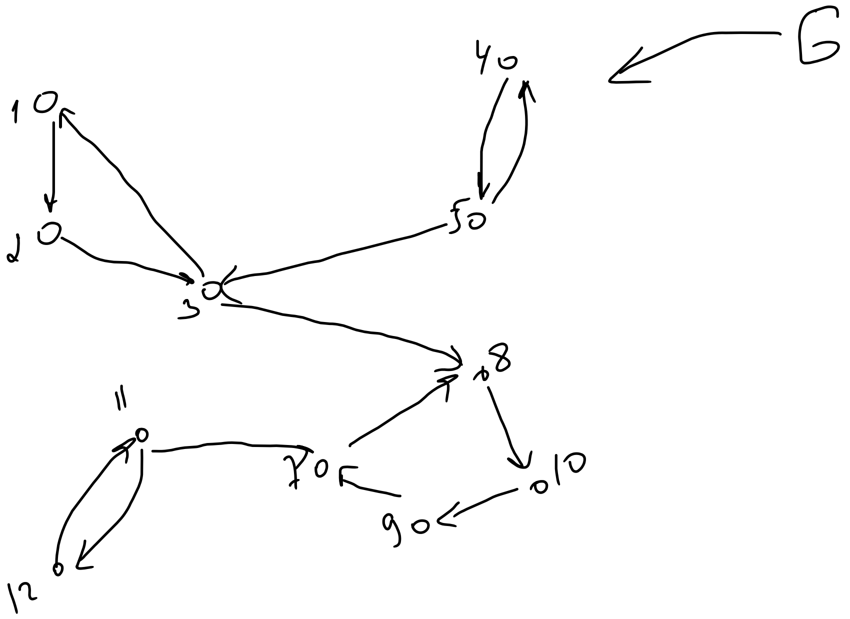
Вспомогательный граф (V×V если матр смежности)

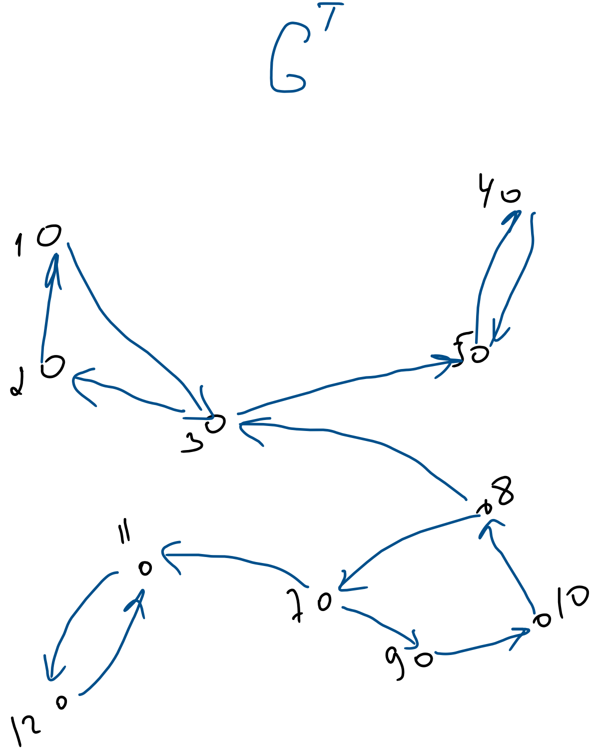
Поиск компонент связности

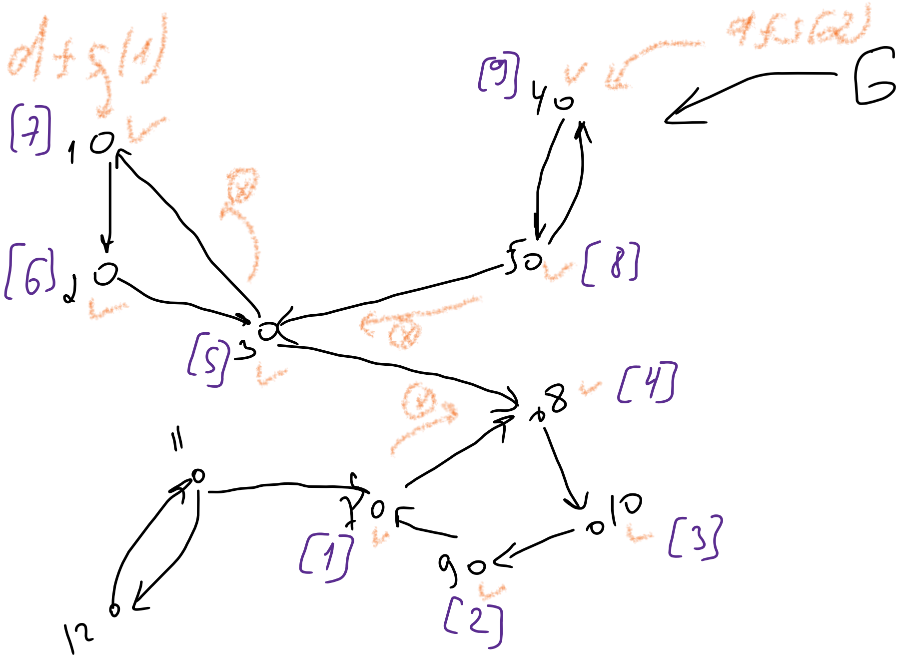
- Как найти компоненты слабой связности?
  - *Используя вспомогательный неориентированный граф и обход по нему с пометкой вершин соответствующими компонентами исходного графа*
- Сложность какая?
  - *Согласно сложности используемого обхода*
  - *У слабой связности аналогично обходу неориентированного вариант графа*
- Память?
  - *Согласно используемому способу хранения графа*
    - *Матрица VxV*
    - *Список V + E*
    - *Вспомогательные массивы и очередь + рекурсия*
  - *Для слабой связности дополнительный дубль хранения неориентированного варианта граф*



Конденсация графа G

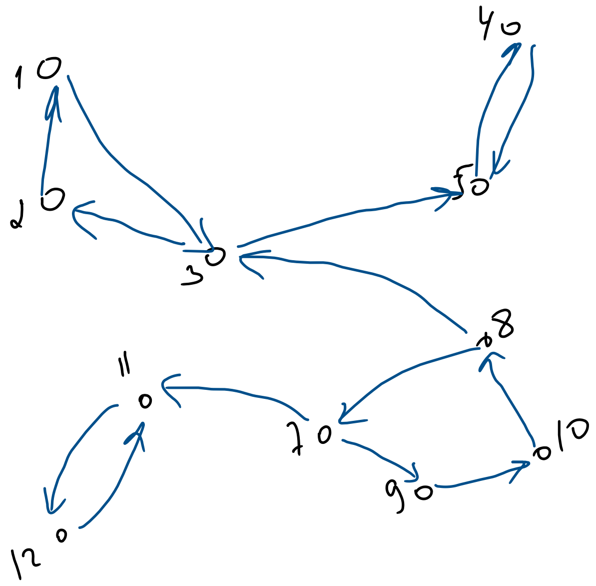

G
1
2
3
4
5
7
8
9
10
11
12

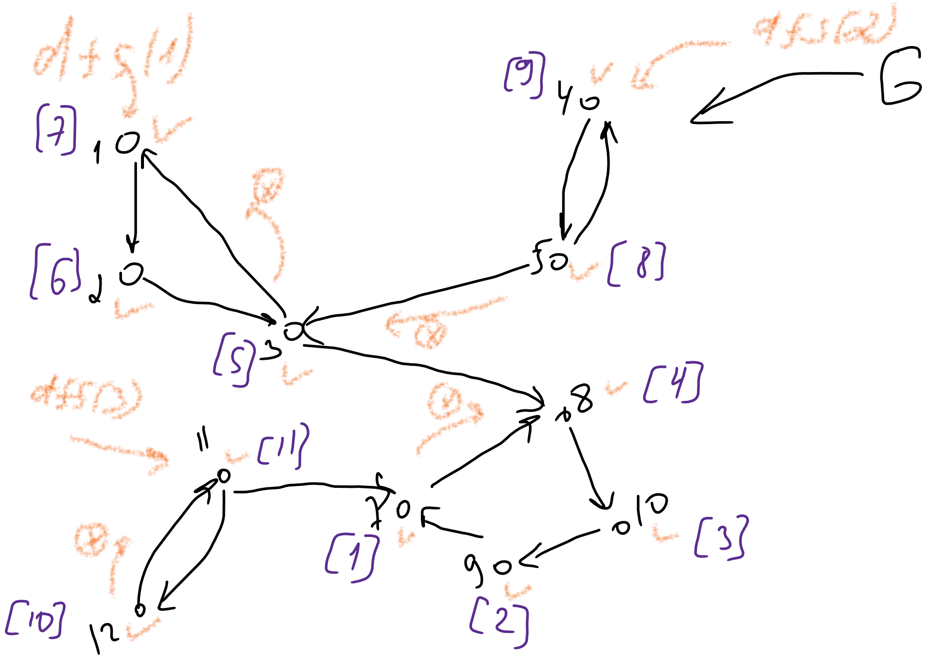

G^T
1
2
3
4
5
7
8
9
10
11
12



[№ завершение обхода]

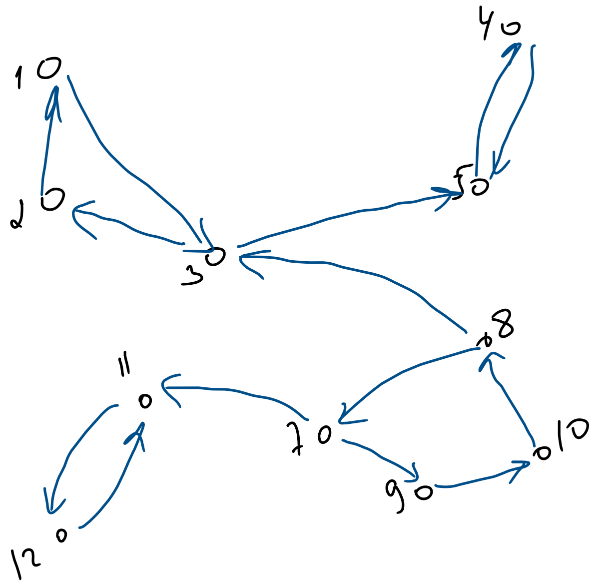
$G^T = H$

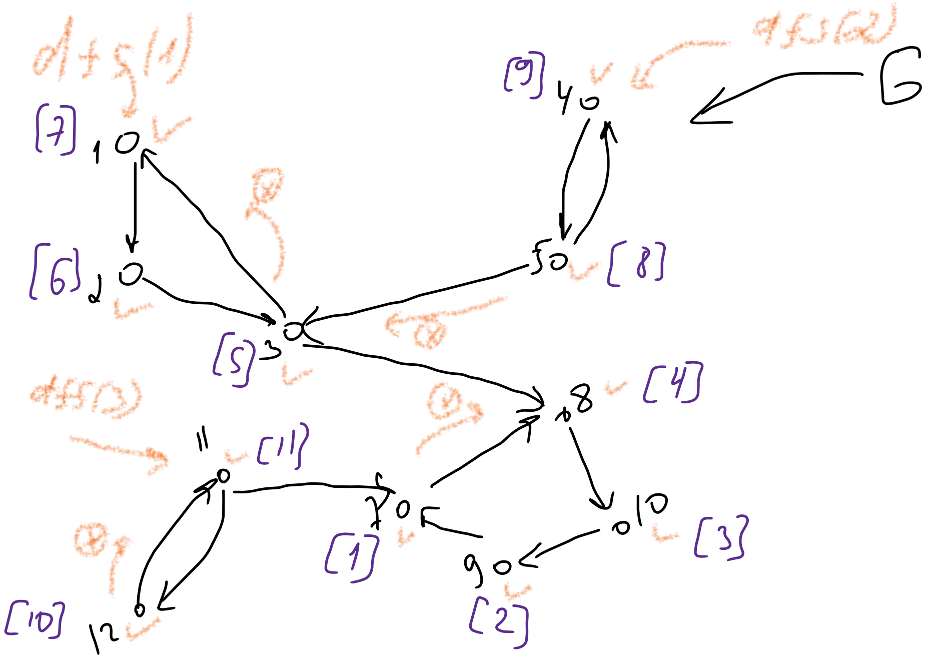




[№ завершение обхода]

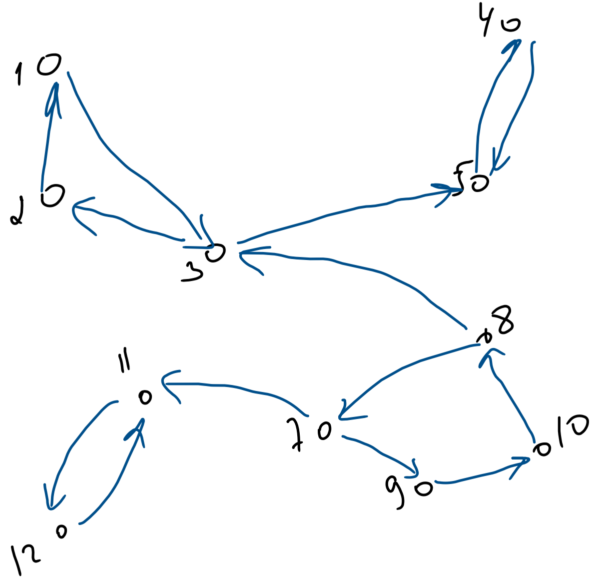
$G^T = H$





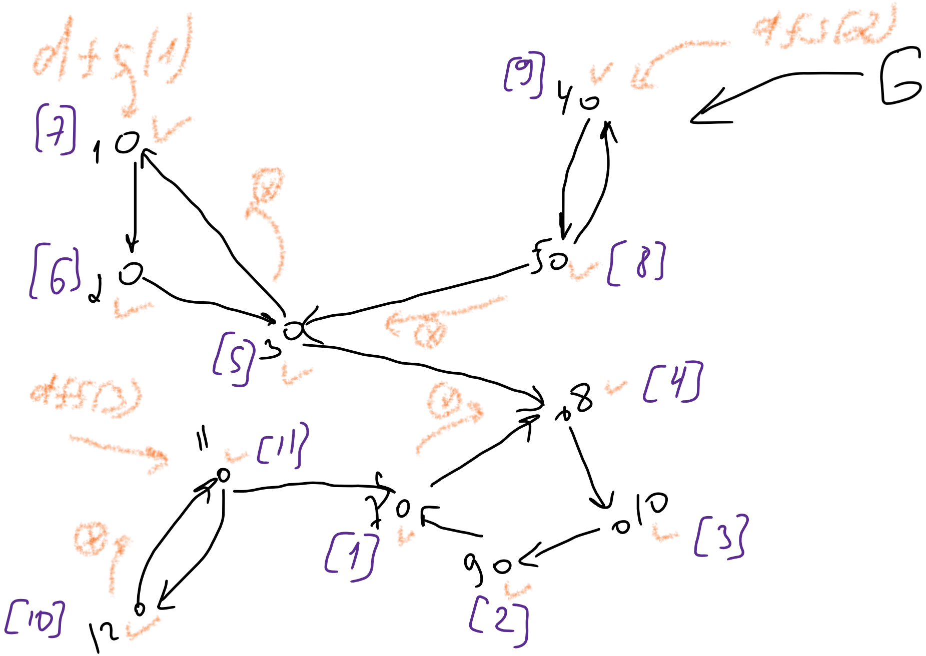
[№ завершения обхода]

$G^T = H$



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

в обратном порядке обхода

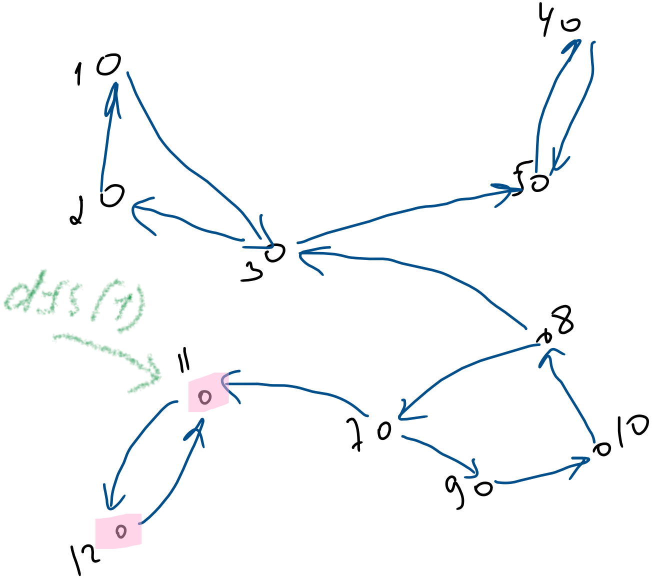


[№ завершение обхода]

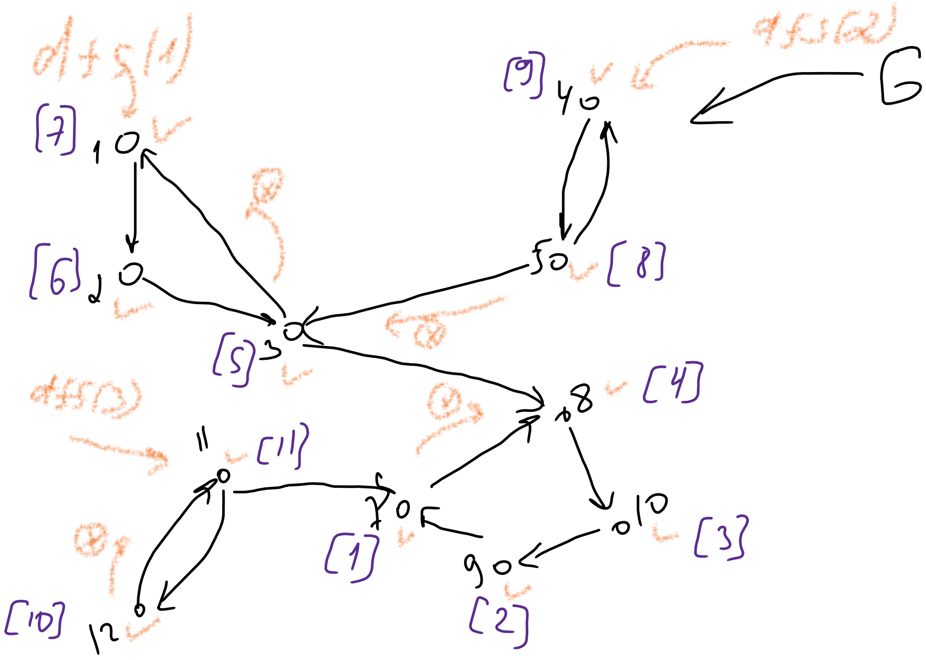
$G^T = H$

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

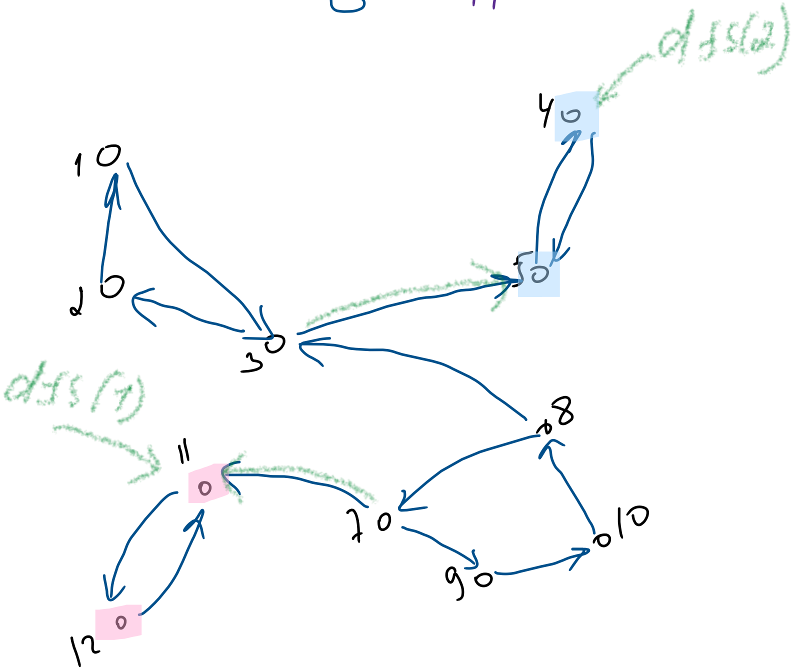
$\xrightarrow[\text{порядке обойдём}]{\text{в обратном}}$



[№ завершение обхода]



$G^T = H$

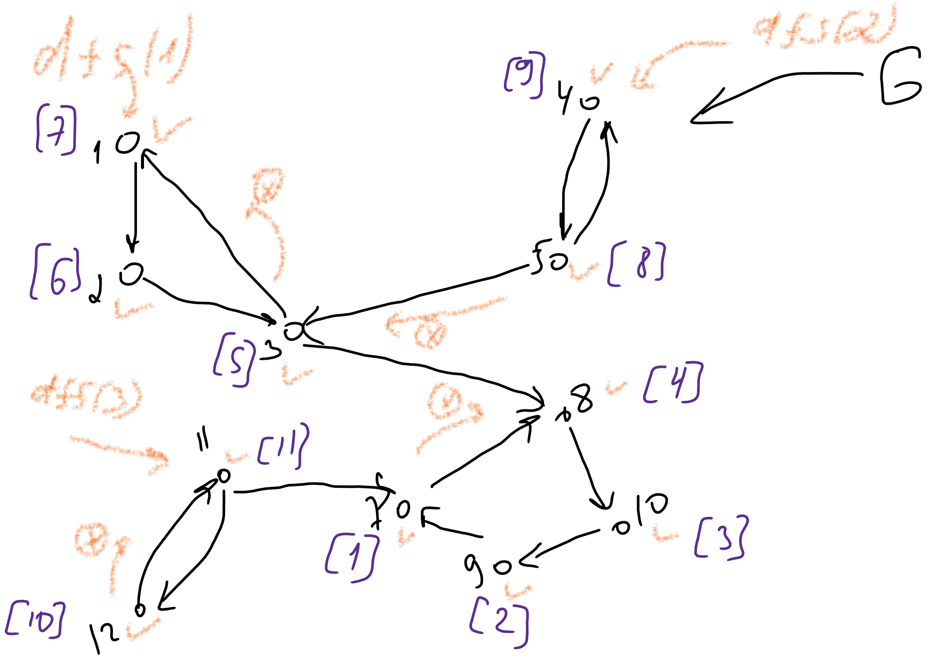


11	10	9	8	7	6	5	4	3	2	1
11	12	4	5	1	2	3	8	10	9	7

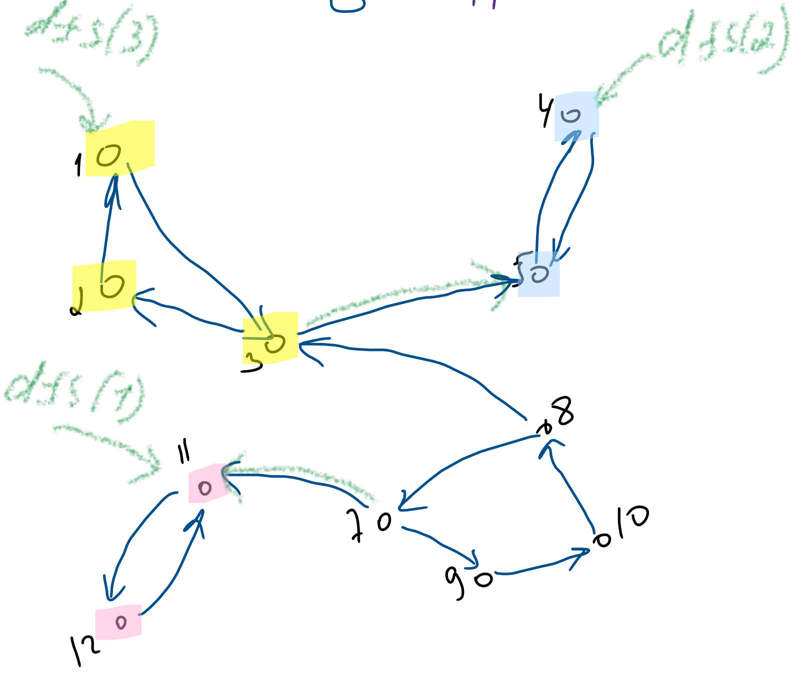
dfs

в обратном порядке обойдём

[№ завершение обхода]



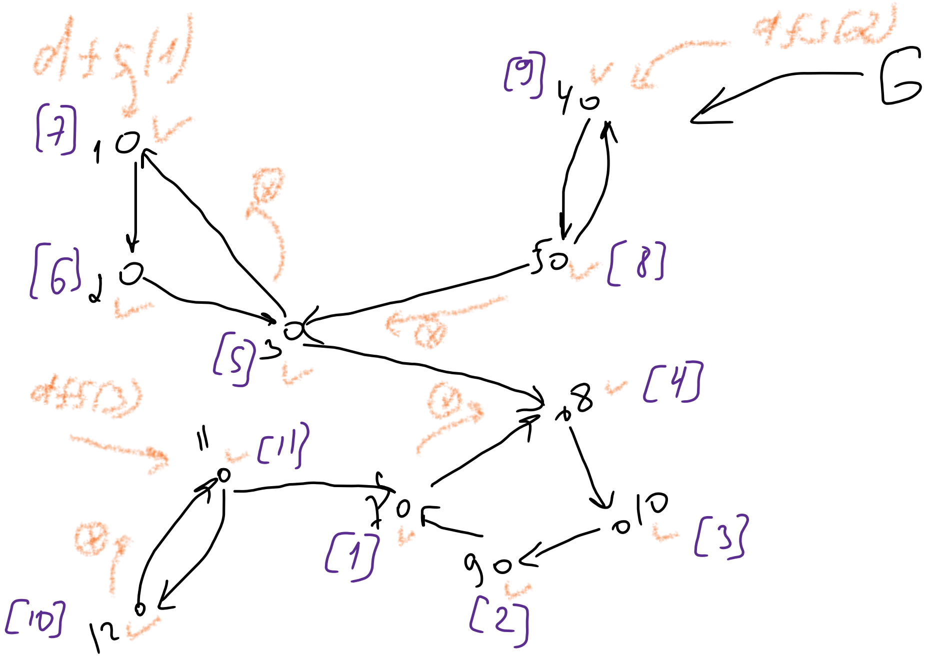
$G^T = H$



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

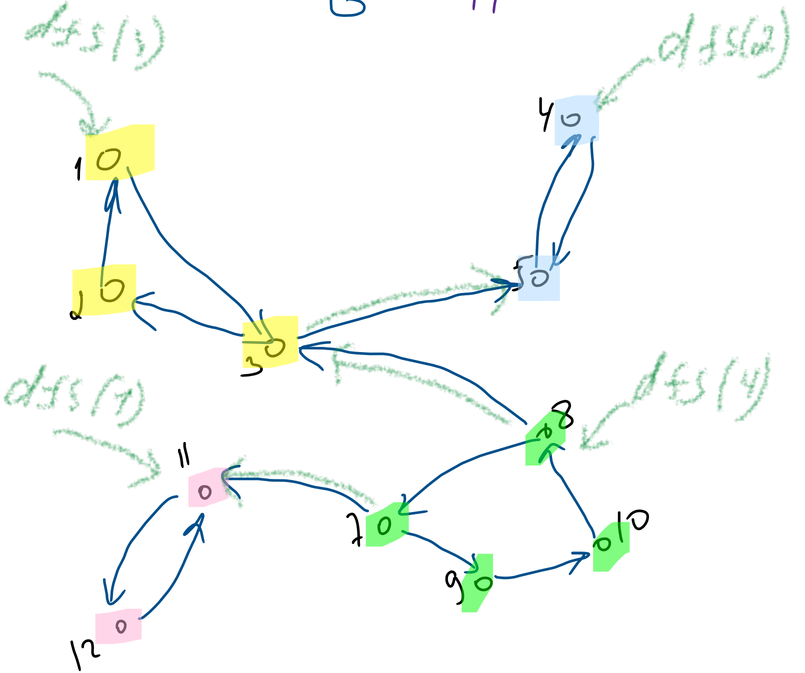
dfs dfs

в обратном порядке обойдём

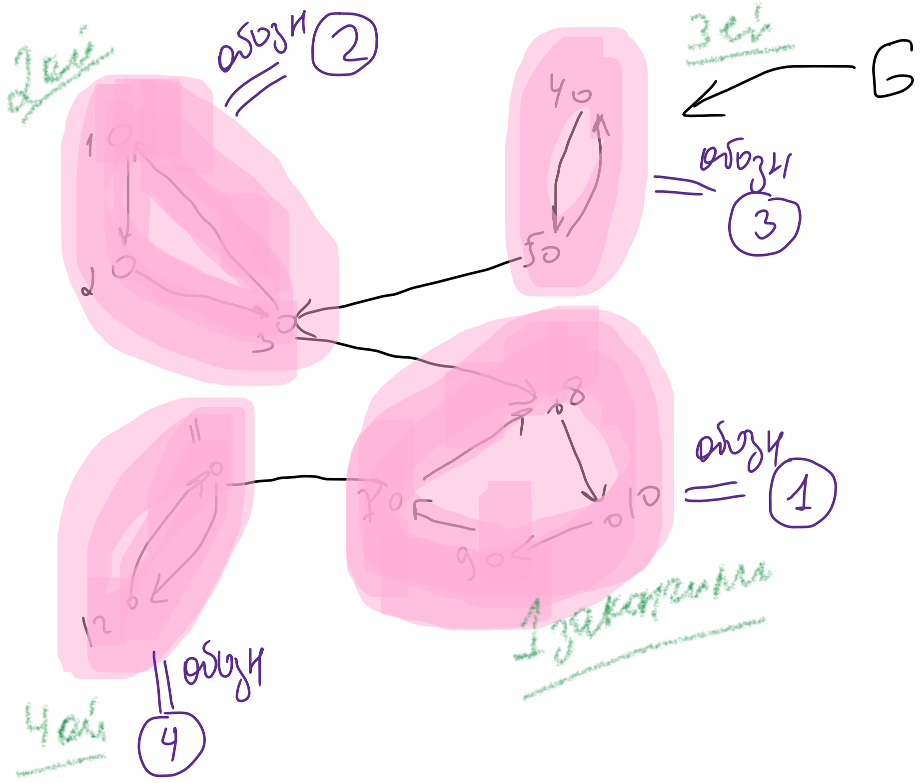


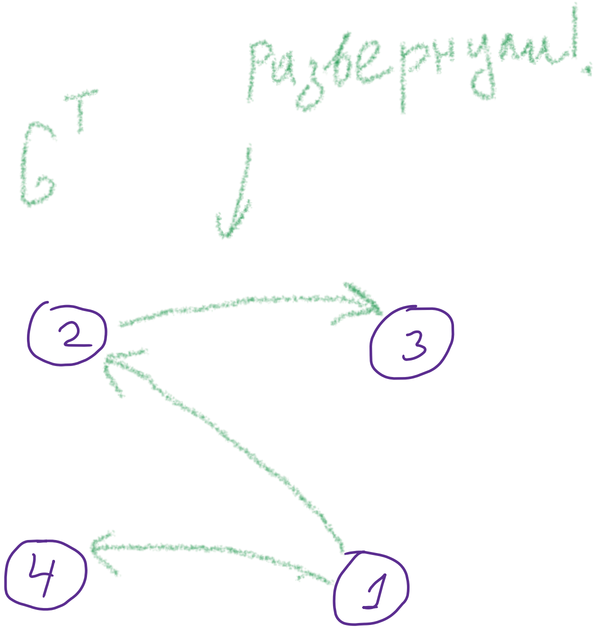
[№ завершение обхода]

$G^T = H$



в обратном порядке обойдёни


2ой
обозн 2
3ей
G
обозн 3
обозн 1
1 закончили
4ой
обозн 4
1
2
3
4
5
7
8
9
10
11
12

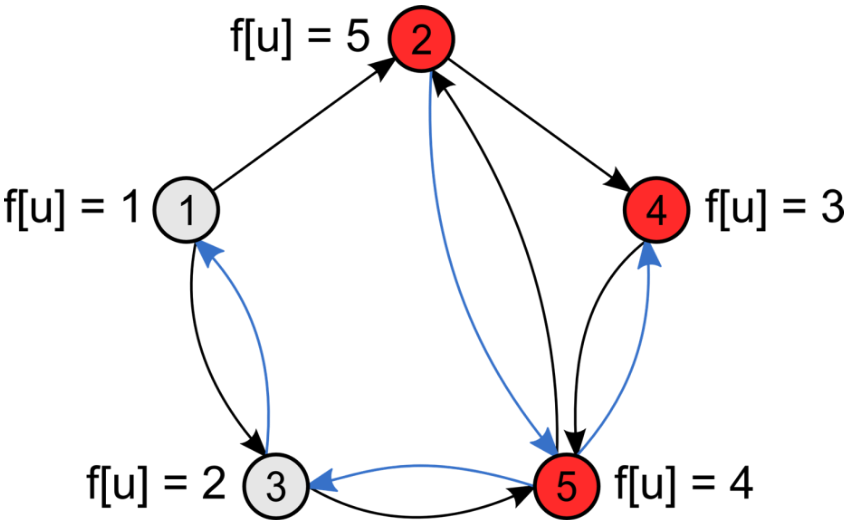

Развернули!
G^T
2
3
4
1

Конденсация: компоненты сильной связности

- Компоненты сильной связности в графе G можно найти с помощью поиска в глубину в 3 этапа:

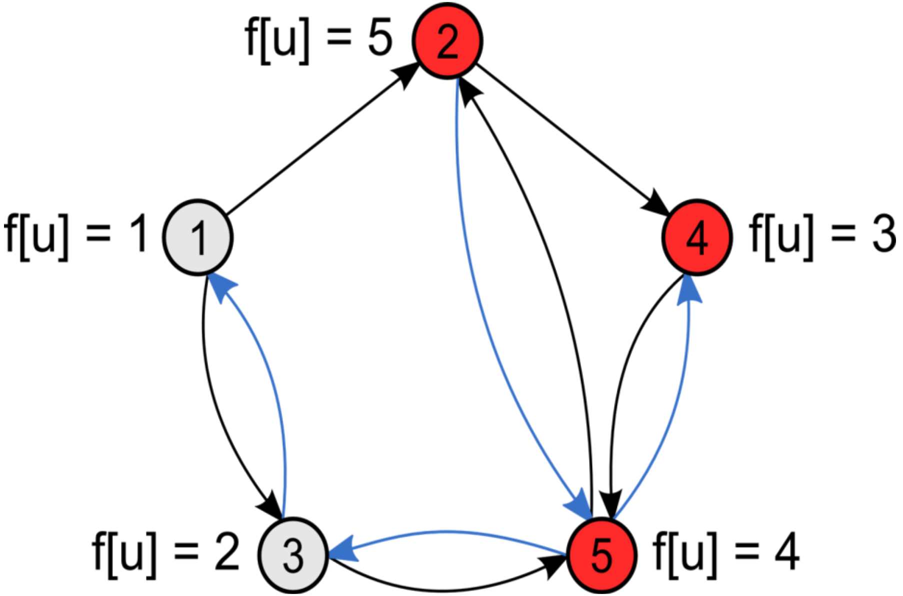
  - 1. Построить граф H с обратными (инвертированными) рёбрами
  - 2. Выполнить в H поиск в глубину и найти f[u] — время окончания обработки вершины u
  - 3. Выполнить поиск в глубину в G, перебирая вершины во внешнем цикле в порядке убывания f[u]

- Полученные на 3-ем этапе деревья поиска в глубину будут являться компонентами сильной связности графа G.
- Так как компоненты сильной связности G и H графа совпадают, то первый поиск в глубину для нахождения f[u] можно выполнить на графе G, а второй — на H.



Вершины 2, 4, 5 сильносвязаны.
Синим цветом обозначен обод DFS по инвертированным ребрам

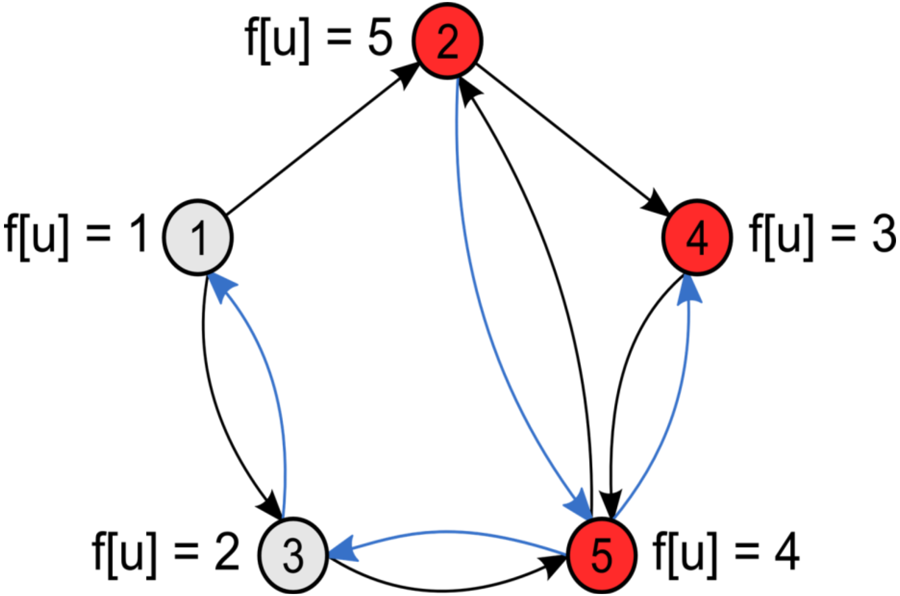
Конденсация: компоненты сильной связности



Вершины 2, 4, 5 сильносвязаны.
Синим цветом обозначен обод DFS по инвертированным ребрам

```
function dfsG(u):
        u. visited = true
        for v in G.V[u]
            if not v.visited
                dfsG(v)
        stack. push (u)

    function dfsH(u):
        component[u] = count
         for v in H.V[u]
             if component[v]==0
                 dfsH(v)

    function main():
     формируем графы  G и H
     обнулям массив component
        for u in V
            if not u.visited
                dfsG(u)
        count = 1
        for u = stack.pop
            if component[u]==0
                dfsH(u)
                count ++
```

Конденсация: компоненты сильной связности



Вершины 2, 4, 5 сильносвязаны.
Синим цветом обозначен обод DFS по инвертированным ребрам

```
function dfsG(u):
        u. visited = true
        for v in G.V[u]
            if not v.visited
                dfsG(v)
        stack. push (u)

    function dfsH(u):
        component[u] = count
         for v in H.V[u]
             if component[v]==0
                 dfsH(v)

     function main():
      формируем графы  G и H
      обнулям массив component
         for u in V
             if not u.visited
                 dfsG(u)
         count = 1
         for u = stack.pop
             if component[u]==0
                 dfsH(u)
                 count ++
```

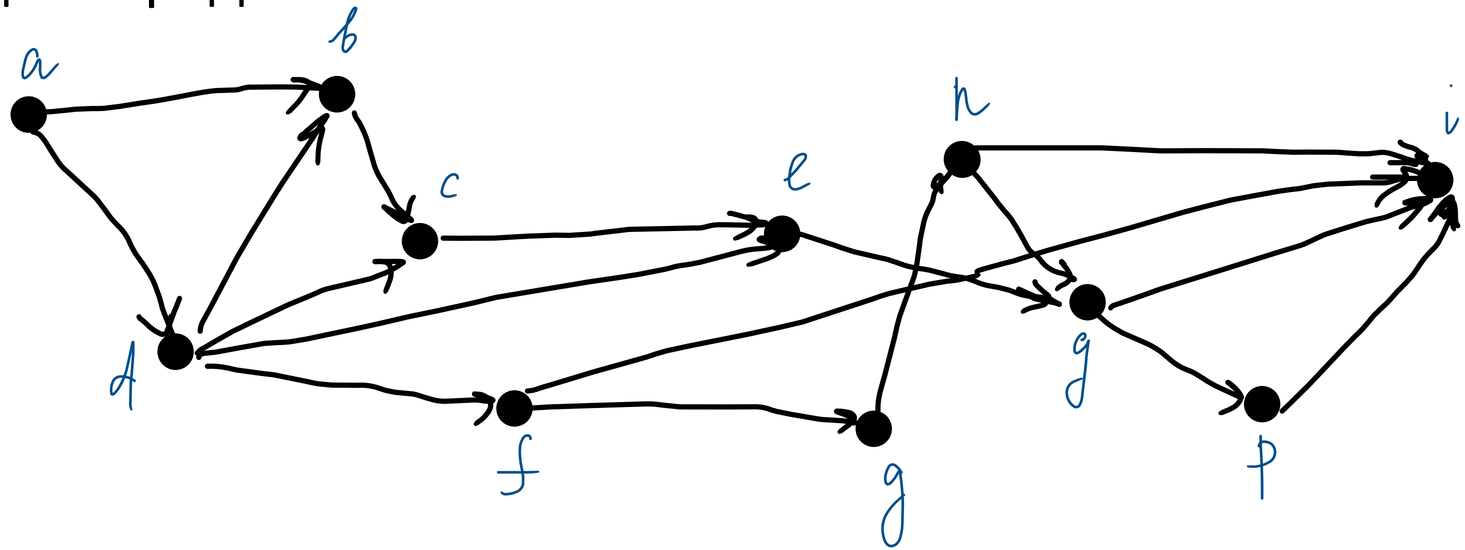
!!!

по G проходим граф и добавл в стек вершины согласн. меткам времени заверш. обхода

расставляем № к/с согласно стеку, но по G^T (обратн. дугам)

Топологическая сортировка

- Только для ациклических ориентированных графов!
- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

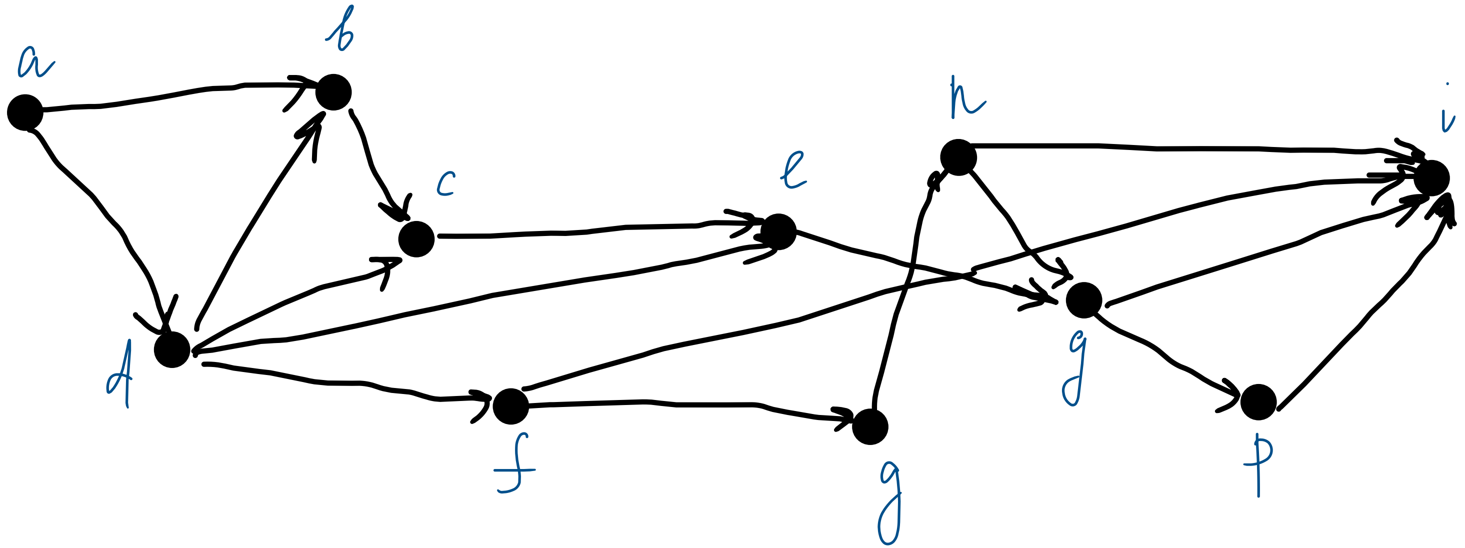


Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

1) ищем исток
2) удаляем исток и все дуги из него

Алгоритм ДЕМУКРОНА

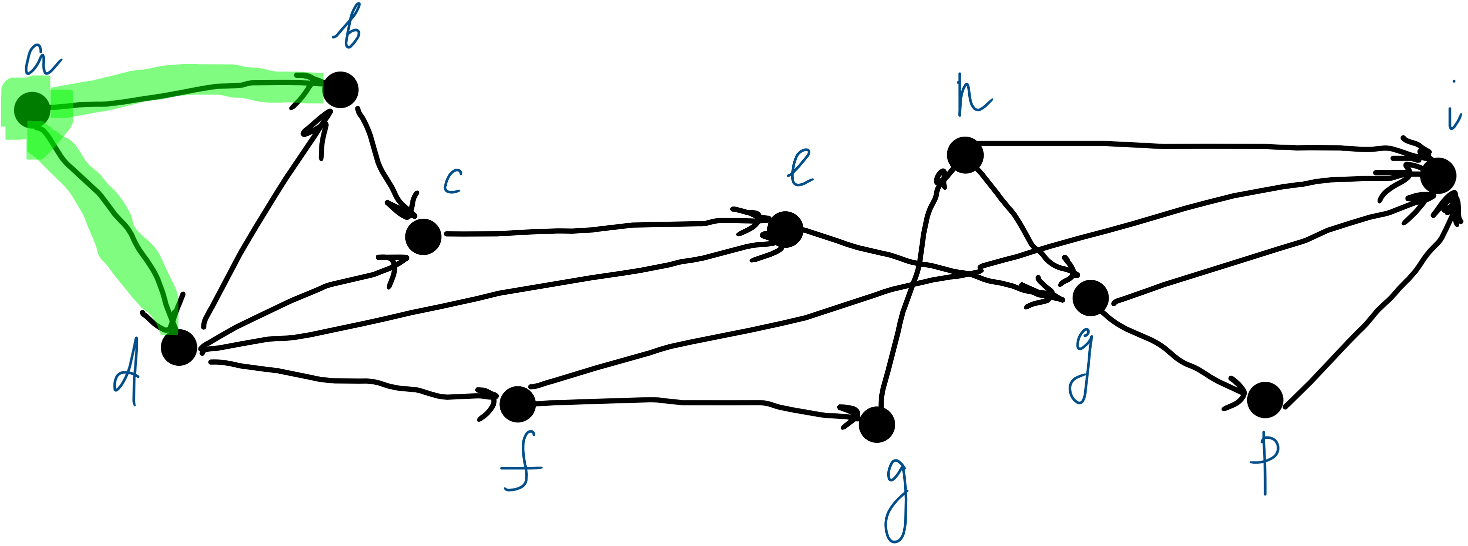


Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

Алгоритм ДЕМУКРОНА

1) ищем исток
2) удаляем исток и все дуги из него



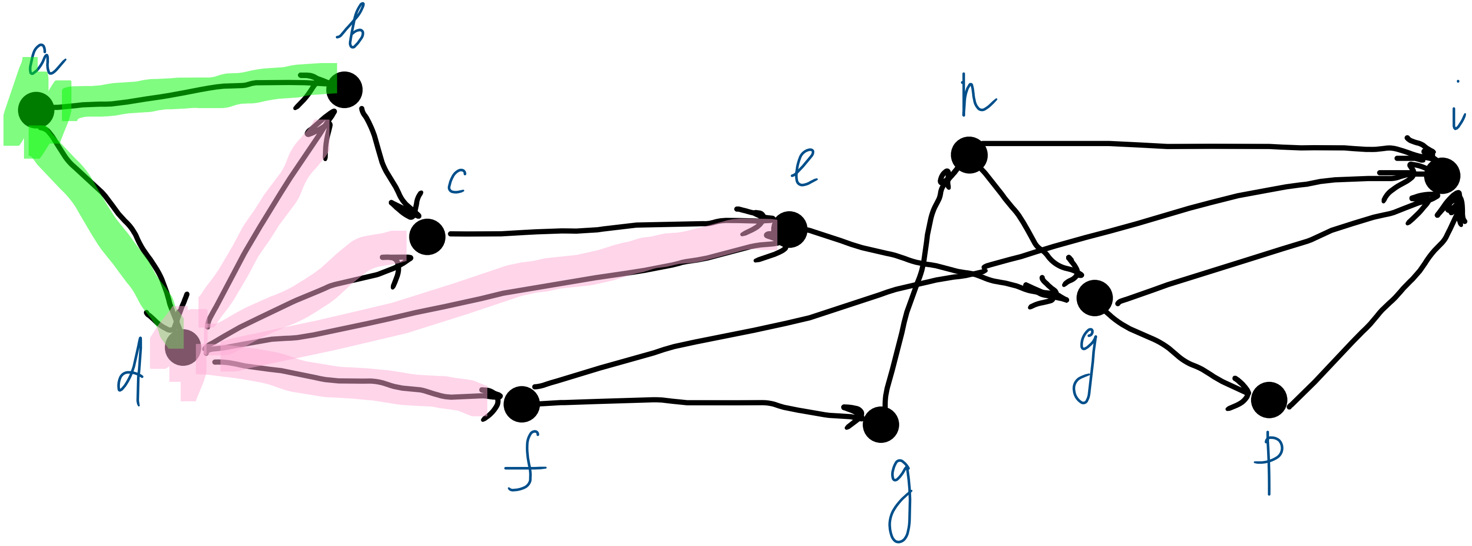
a

Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

Алгоритм ДЕМУКРОНА

1) ищем исток
2) удаляем исток и все дуги из него



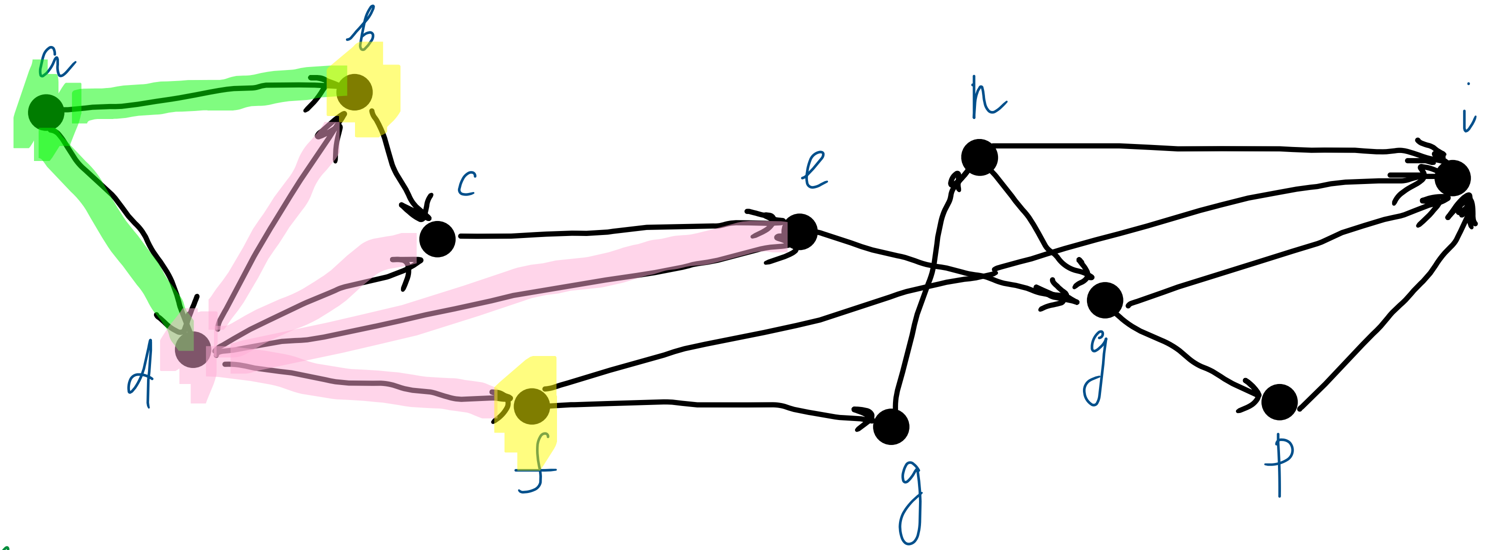
a d

Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

Алгоритм ДЕМУКРОНА

1) ищем исток
2) удаляем исток и все дуги из него



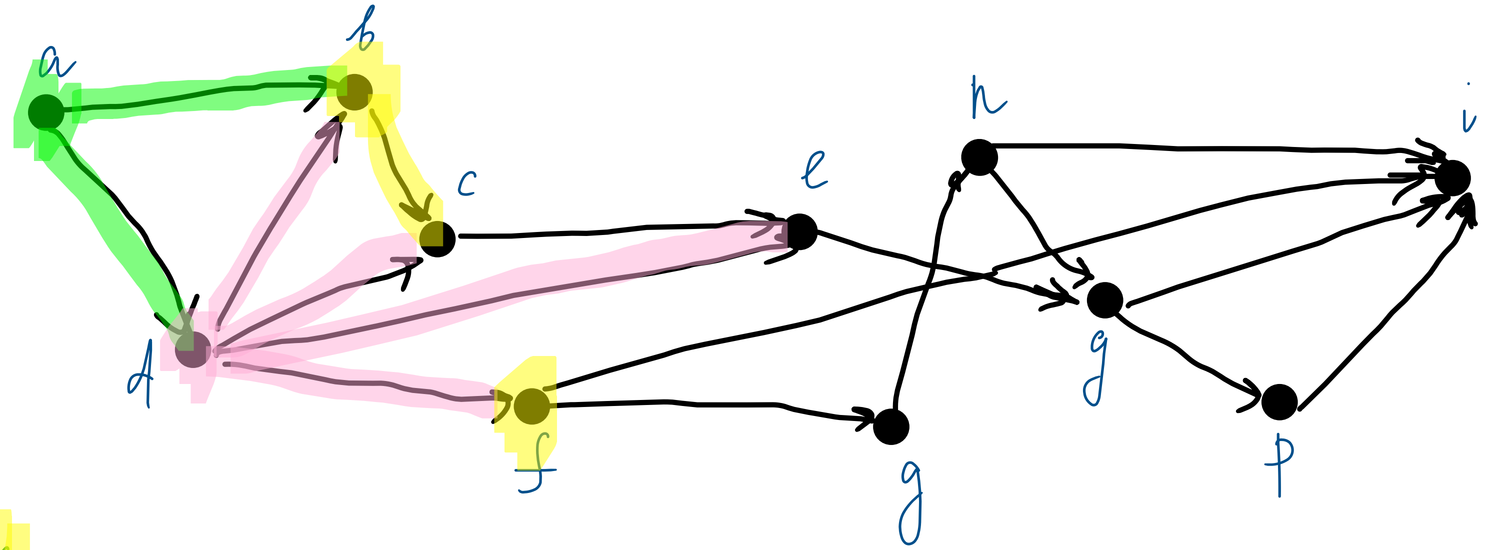
a d b f
?

Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

Алгоритм ДЕМУКРОНА

1) ищем исток
2) удаляем исток и все дуги из него



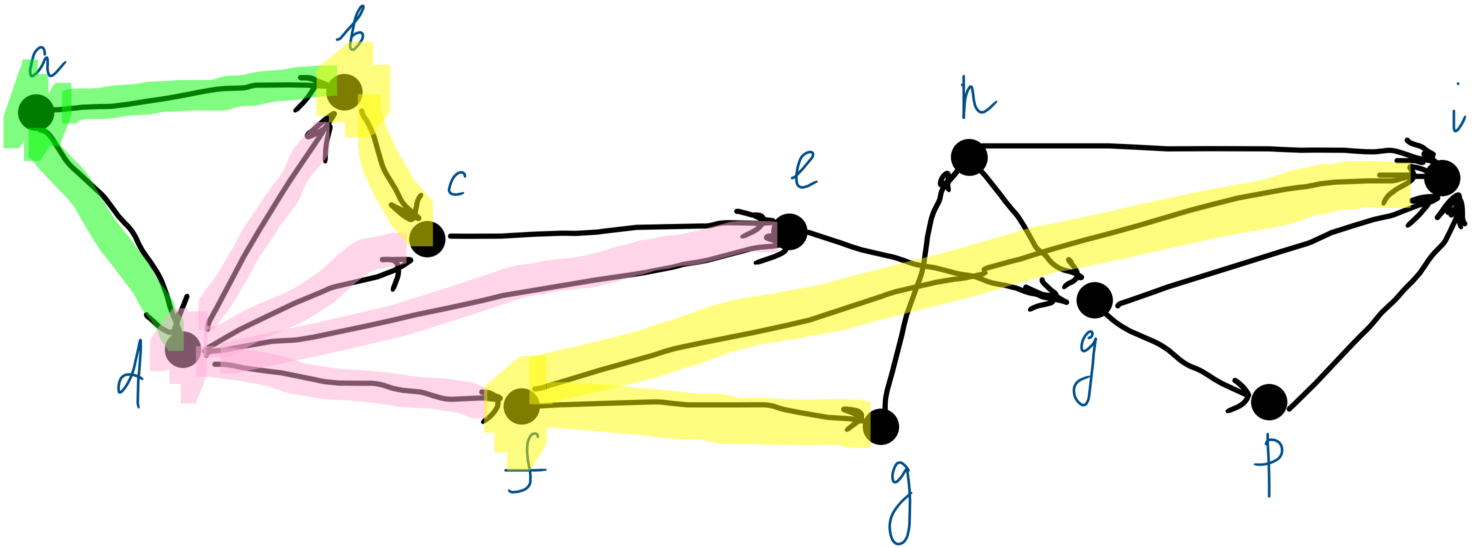
a d b f
?

Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

Алгоритм ДЕМУКРОНА

1) ищем исток
2) удаляем исток и все дуги из него



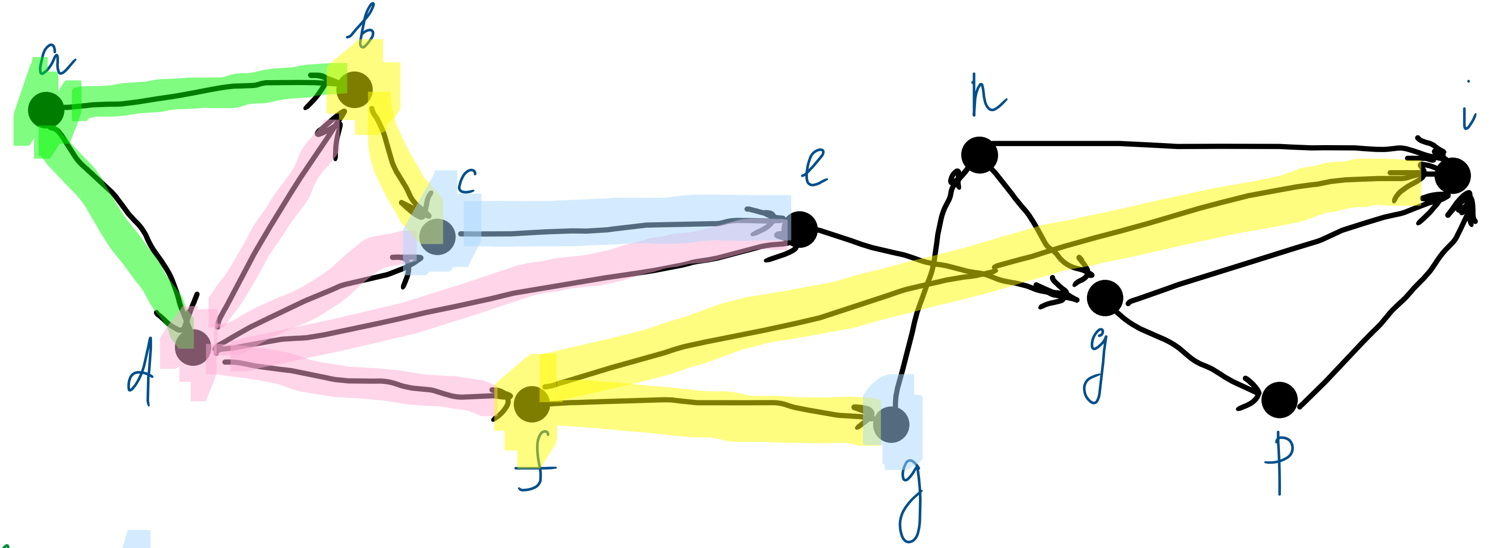
a d b f

Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

1) ищем исток
2) удаляем исток и все дуги из него

Алгоритм ДЕМУКРОНА



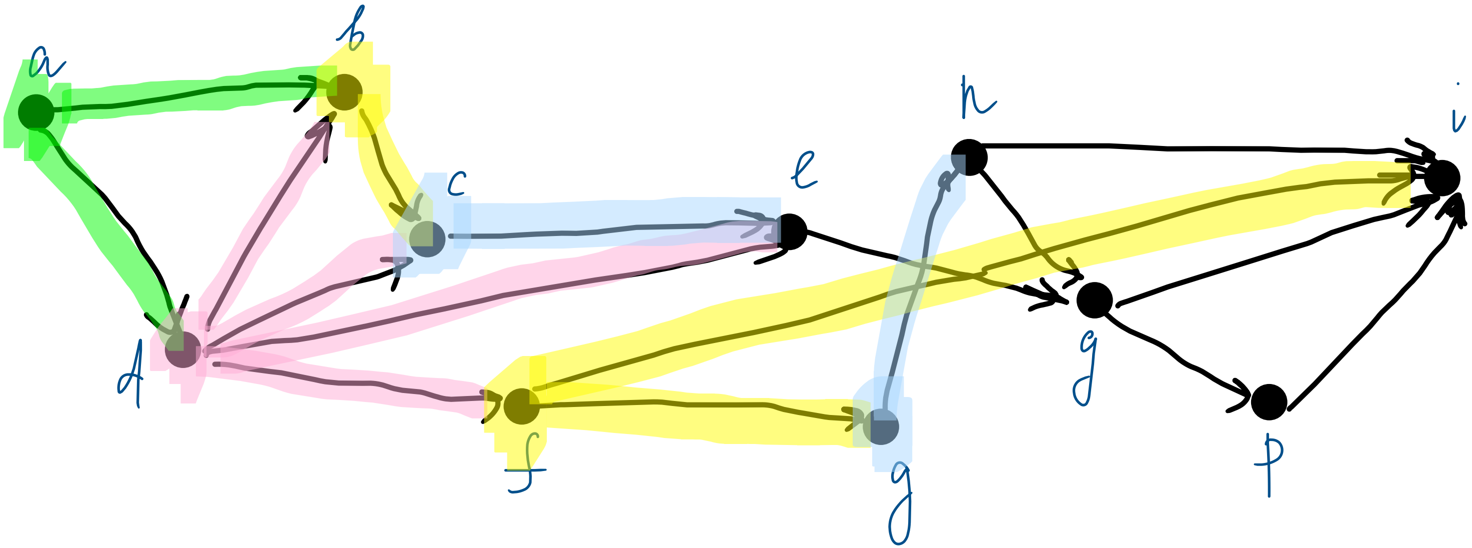
a d b f c g
?

Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

Алгоритм ДЕМУКРОНА

1) ищем исток
2) удаляем исток и все дуги из него



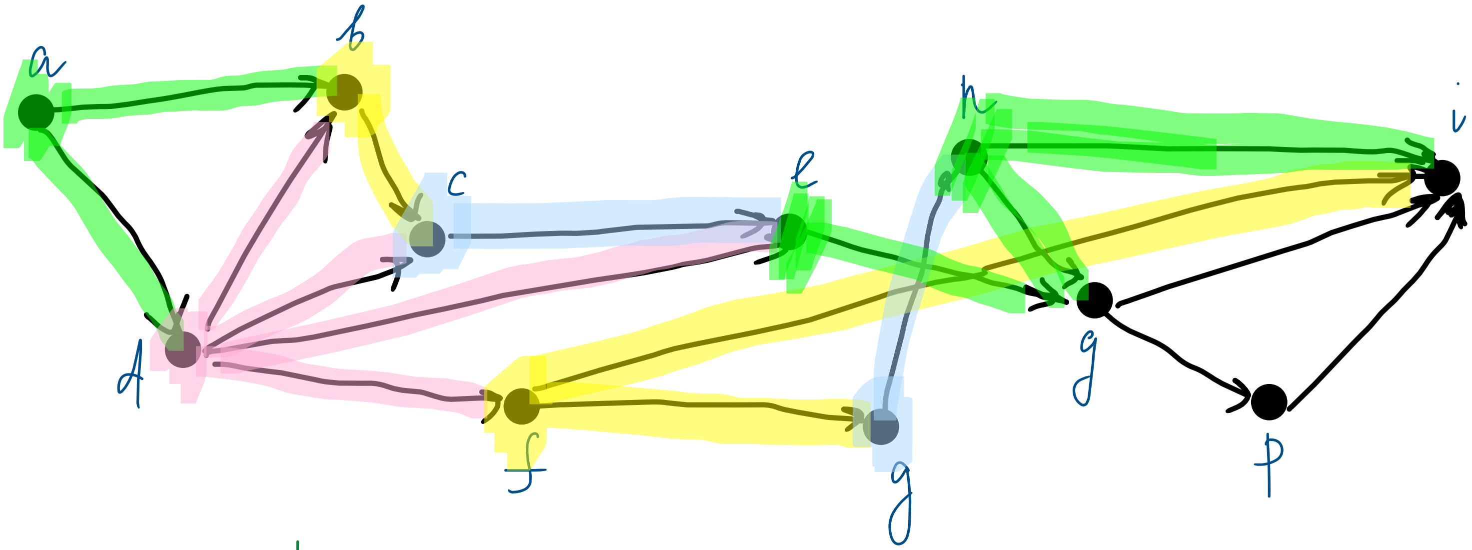
a d b f c g

Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

Алгоритм ДЕМУКРОНА

1) ищем исток
2) удаляем исток и все дуги из него



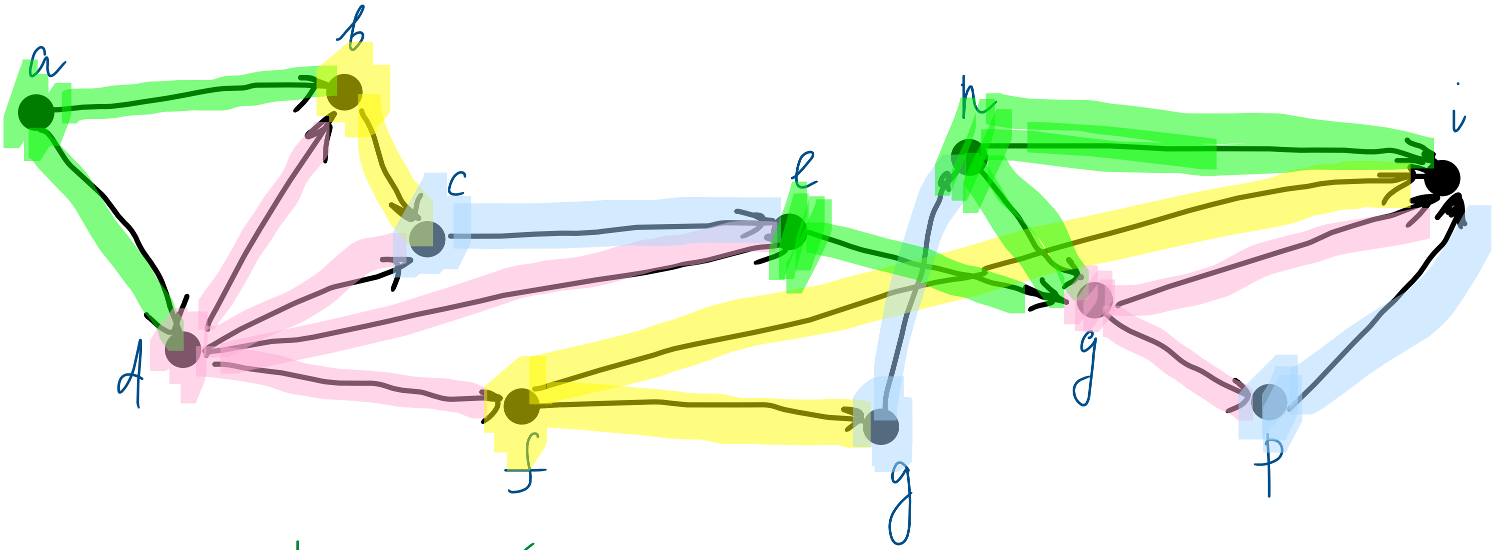
a d b f c g e h

Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

Алгоритм ДЕМУКРОНА

1) ищем исток
2) удаляем исток и все дуги из него



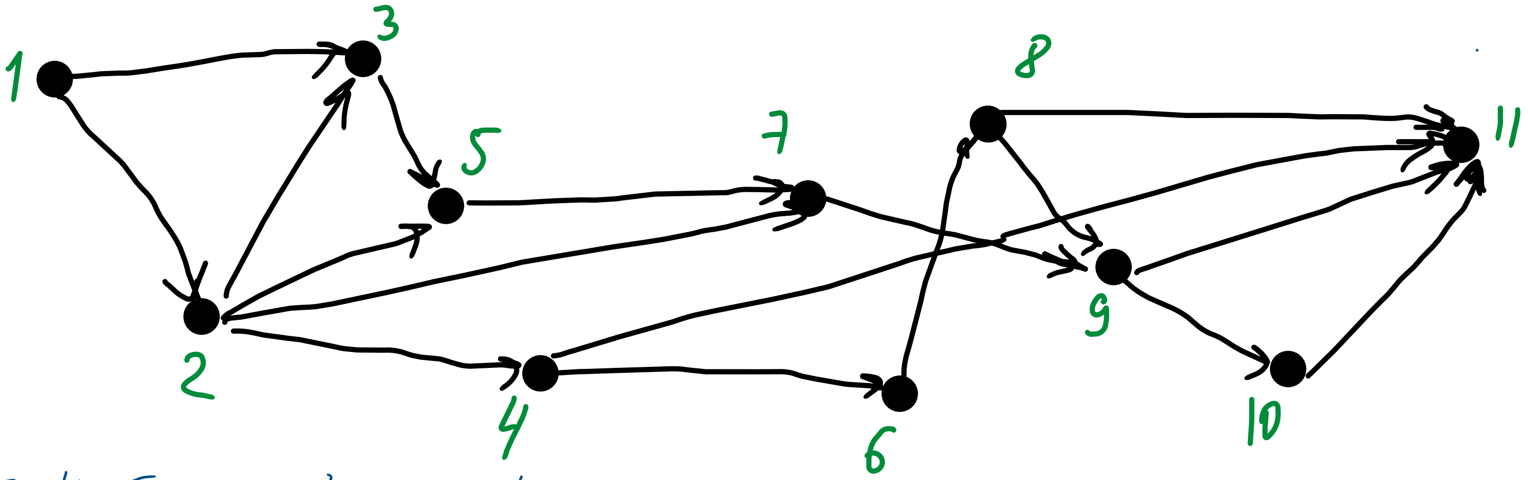
a d b f c g e h g p v

Топологическая сортировка

- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

Алгоритм ДЕМУКРОНА

1) ищем исток
2) удаляем исток и все дуги из него



1 2 3 4 5 6 7 8 9 10 11
a d b f c g e h g p v

Топологическая сортировка

- Только для ациклических ориентированных графов!
- Сортирует граф, таким образом, что для любой дуги (u, v) u будет перед v

Алгоритм

Топологическая сортировка (G(V, E))

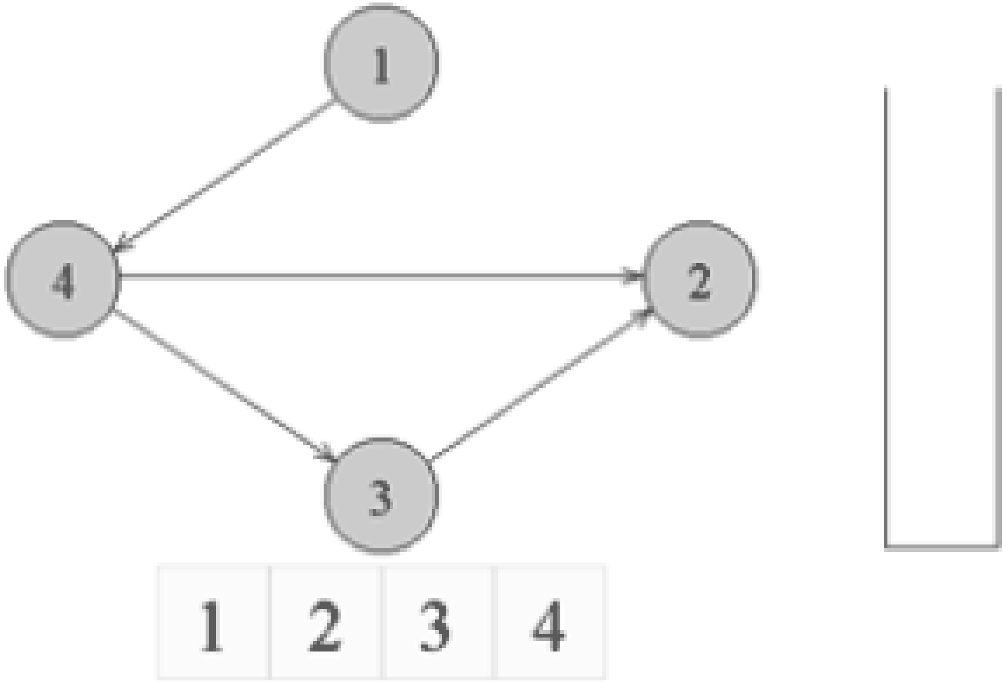
- Обход графа G(V, E) в глубину
  - Каждую пройденную (черную) вершину помещаем в стек
- Достать все вершины из стека
  - *С помощью топологической сортировки можно найти гамильтонов путь!*
  - *В направленном графе позволяет быстро найти наикратчайшие пути до всех от заданной*

Топологическая сортировка

- Топологическая сортировка (G(V, E))
  - Обход графа G(V, E) в глубину
    - Каждую пройденную (черную) вершину помещаем в стек
  - Достать все вершины из стека

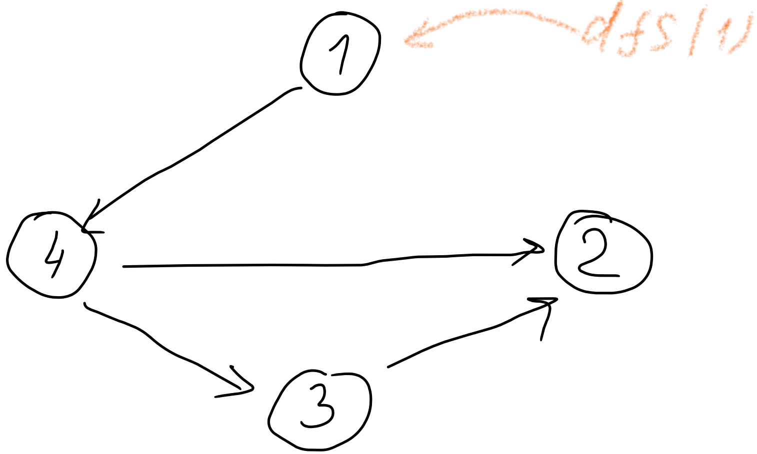
```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited,false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
    visited[u]=true
    for uv ∈ E(G)
        if not visited[v]
            dfs(v)
    ans.pushBack(u)
```

Топологическая сортировка



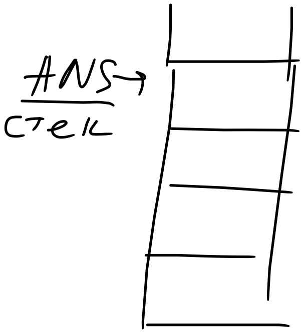

```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited, false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
    visited[u]=true
    for uv ∈ E(G)
        if not visited[v]
            dfs(v)
    ans.pushBack(u)
```

Топологическая сортировка



visited

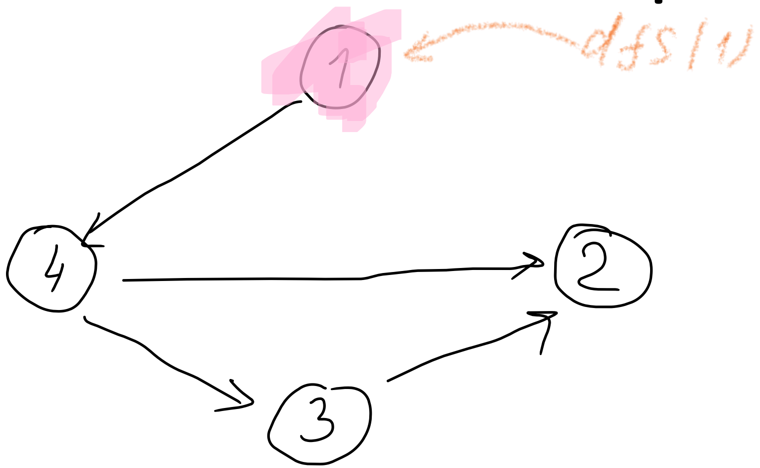
1 2 3 4




```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited,false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
     visited[u]=true
     for uv ∈ E(G)
         if not visited[v]
             dfs(v)
     ans.pushBack(u)
```

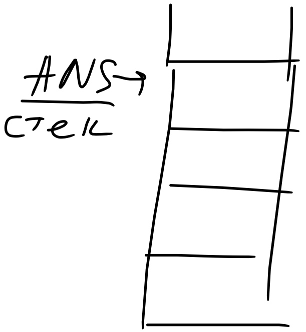
посещена

Топологическая сортировка



visited

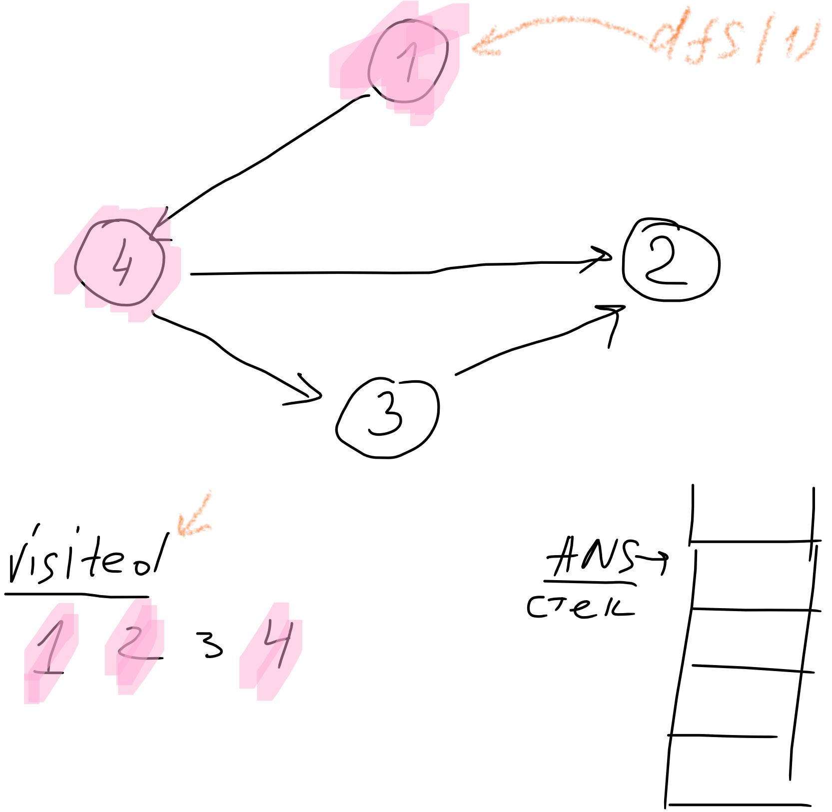
1 2 3 4




```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited, false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
    visited[u]=true
    for uv ∈ E(G)
        if not visited[v]
            dfs(v)
    ans.pushBack(u)
```

посещена

Топологическая сортировка

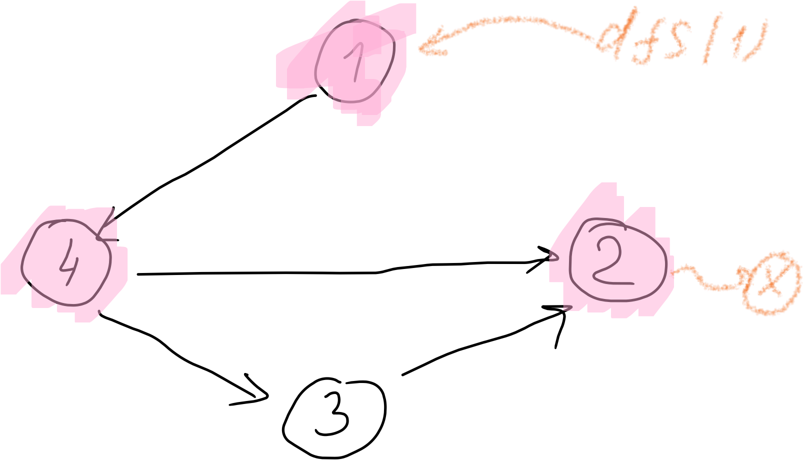



```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited, false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
     visited[u]=true
     for uv ∈ E(G)
         if not visited[v]
             dfs(v)
     ans.pushBack(u)
```

посещена

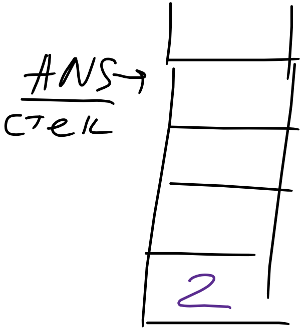
Топологическая сортировка

```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited, false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
     visited[u]=true
     for uv ∈ E(G)
         if not visited[v]
             dfs(v)
     ans.pushBack(u)
```



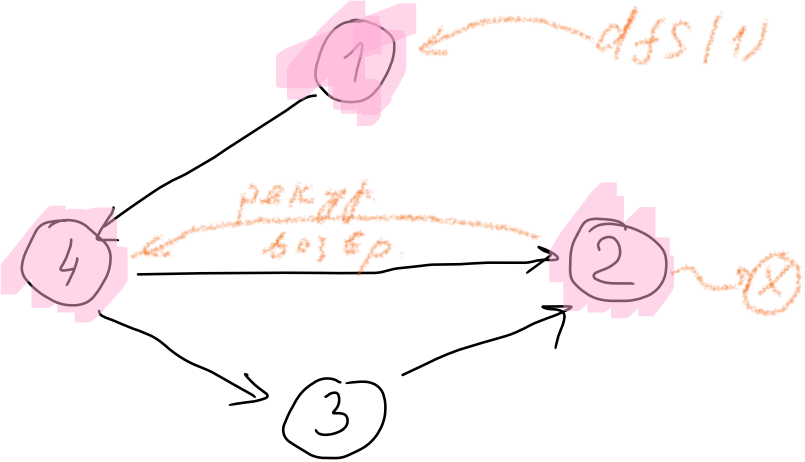
visited

1 2 3 4



посещена

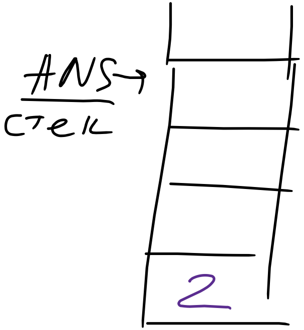
Топологическая сортировка




```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited, false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
     visited[u]=true
     for uv ∈ E(G)
         if not visited[v]
             dfs(v)
     ans.pushBack(u)
```

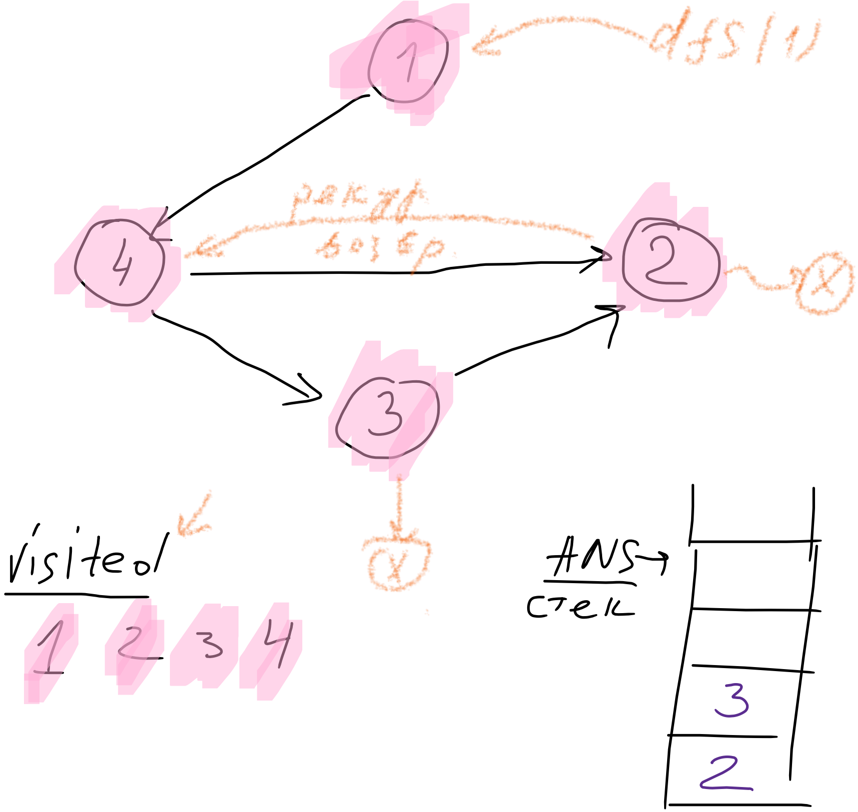
visited

1 2 3 4



посещена

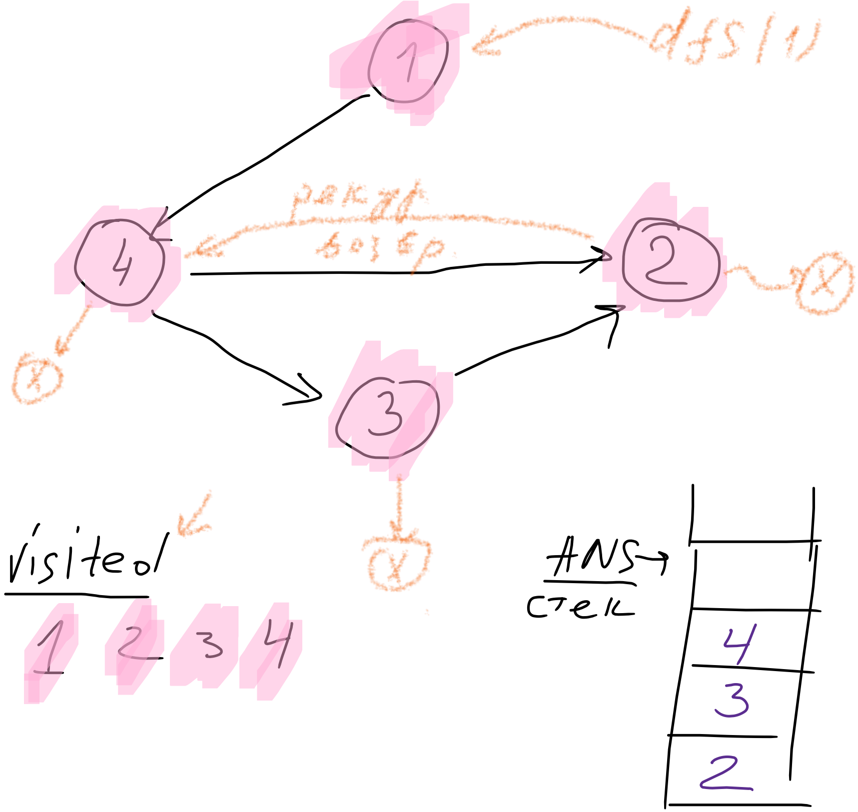
Топологическая сортировка




```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited, false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
     visited[u]=true
     for uv ∈ E(G)
         if not visited[v]
             dfs(v)
     ans.pushBack(u)
```

посещена

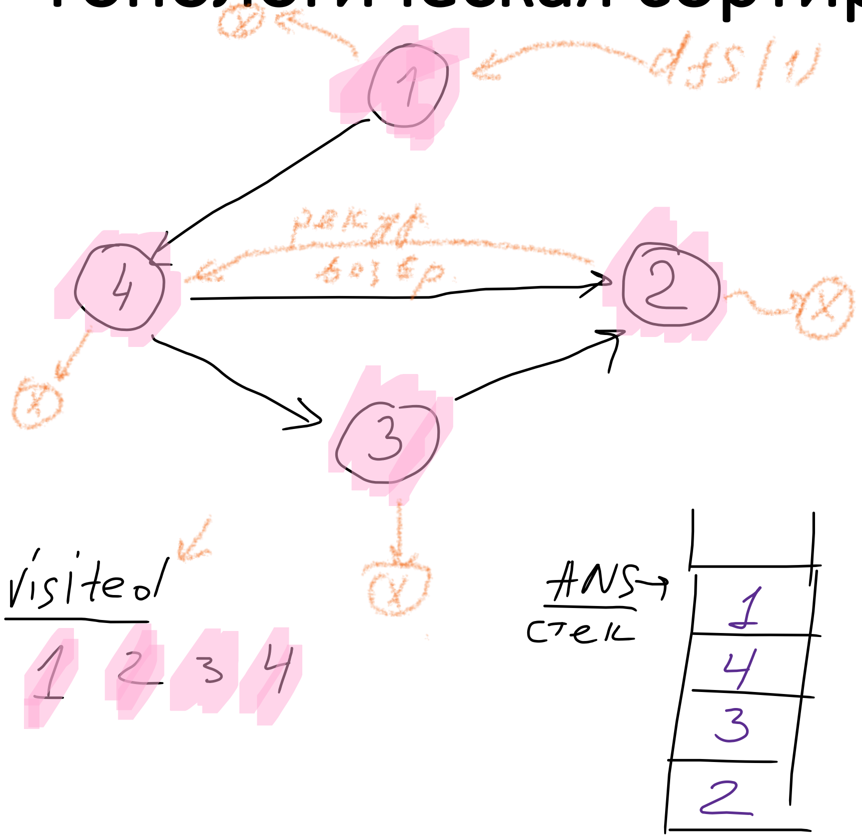
Топологическая сортировка




```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited, false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
    visited[u]=true
    for uv ∈ E(G)
        if not visited[v]
            dfs(v)
    ans.pushBack(u)
```

посещена

Топологическая сортировка

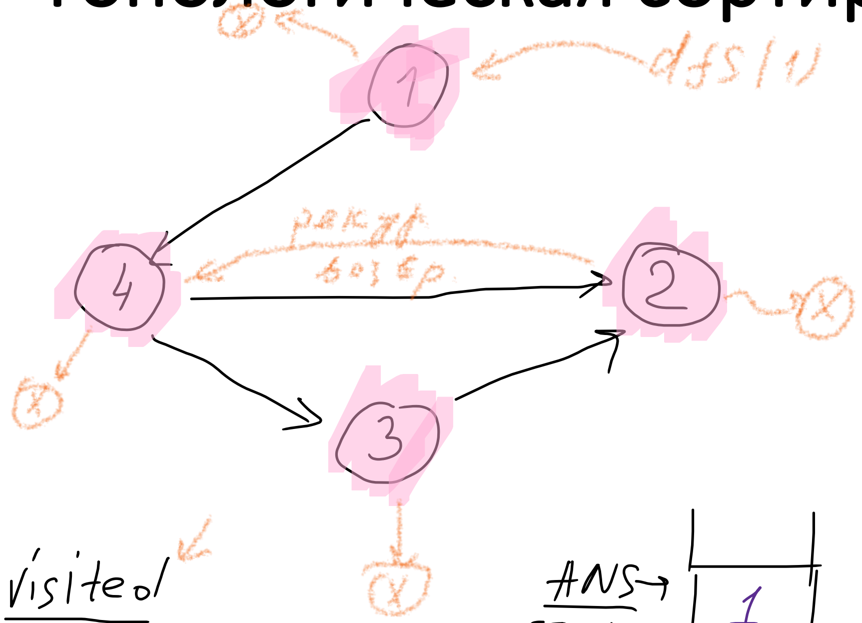



```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited, false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
     visited[u]=true
     for uv ∈ E(G)
         if not visited[v]
             dfs(v)
     ans.pushBack(u)
```

посещена

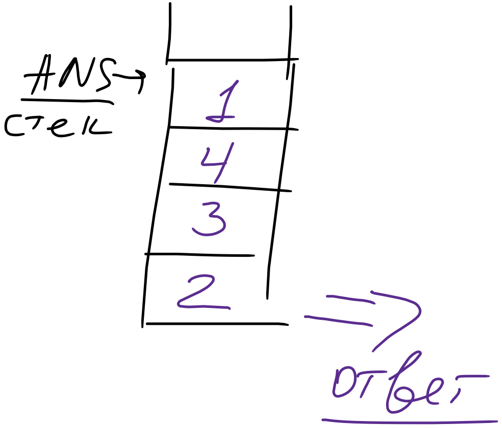
Топологическая сортировка

```
function topologicalSort():
    проверить граф G на ацикличность
    fill(visited, false)
    for v ∈ V(G)
        if not visited[v]
            dfs(v)
    ans.reverse()

function dfs(u):
    visited[u]=true
    for uv ∈ E(G)
        if not visited[v]
            dfs(v)
    ans.pushBack(u)
```



visited

1 2 3 4



~ O(V+E)

посещена

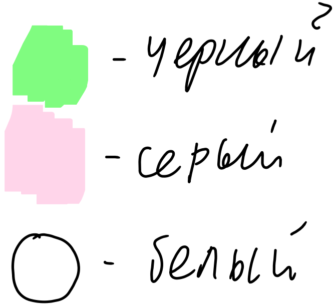
1 4 3 2

топ. отсортир V

ЛЕММА о обходе в глубину

Нет такого момента в процессе выполнения поиска в глубину, когда бы существовало ребро из черной вершины в белую.

Не может быть!



ЛЕММА о обходе в глубину

Нет такого момента в процессе выполнения поиска в глубину, когда бы существовало ребро из черной вершины в белую.

Не может быть!

Proof: допустим возможность ситуации выше

ЛЕММА о обходе в глубину

Нет такого момента в процессе выполнения поиска в глубину, когда бы существовало ребро из черной вершины в белую.

Не может быть!

- черный?
- серый
- белый

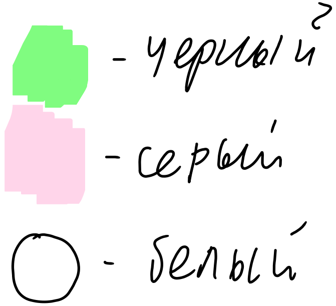
Proof: допустим возможность ситуации выше

1) Чтобы 2 стала черной; 2 сперва д.б. белой и от нее д.б. вызван dfs =>

ЛЕММА о обходе в глубину

Нет такого момента в процессе выполнения поиска в глубину, когда бы существовало ребро из черной вершины в белую.

Не может быть!



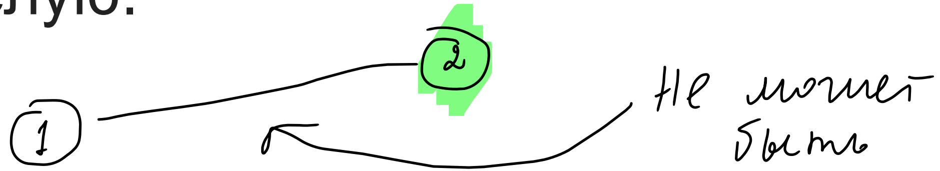
Proof: допустим возможность ситуации выше

1) чтобы 2 стала черной; 2 сперва д.б. белой и от нее д.б. вызван dfs =>

2) dfs(2) => 2 - становится серой, а далее смотрим смежные =>

ЛЕММА о обходе в глубину

Нет такого момента в процессе выполнения поиска в глубину, когда бы существовало ребро из черной вершины в белую.



- черный?
- серый
- белый

Proof: допустим возможность ситуации выше

1) Чтобы 2 стала черной; 2 сперва д.б. белой и от нее д.б. вызван dfs =>

2) dfs(2) => 2 - становится серой, а далее смотрим смежные =>

3) 1 — 2 значит нужно вызвать dfs(1) => 1

ЛЕММА о обходе в глубину

Нет такого момента в процессе выполнения поиска в глубину, когда бы существовало ребро из черной вершины в белую.

Не может быть!

- черный
- серый
- белый

Proof: допустим возможность ситуации выше

1) Чтобы 2 стала черной; 2 сперва д.б белой и от нее д.б вызван dfs =>

2) dfs(2) => 2 - становится серой, а далее смотрим смежные =>

3) 1—2 значит нужно вызвать dfs(1) => 1 =>

4) 1—2 2 станет черной после 1 согласно работе dfs => ↯

ЛЕММА **О БЕЛЫХ ПУТЯХ**

- черный
- серый
- белый

Дан граф **G**. Запустим **dfs(G)**

- остановим выполнение процедуры dfs от некоторой серой вершины **u** - первый момент времени **T1**.
- продолжим выполнение dfs(u) пока u не станет черной - второй момент времени **T2**.

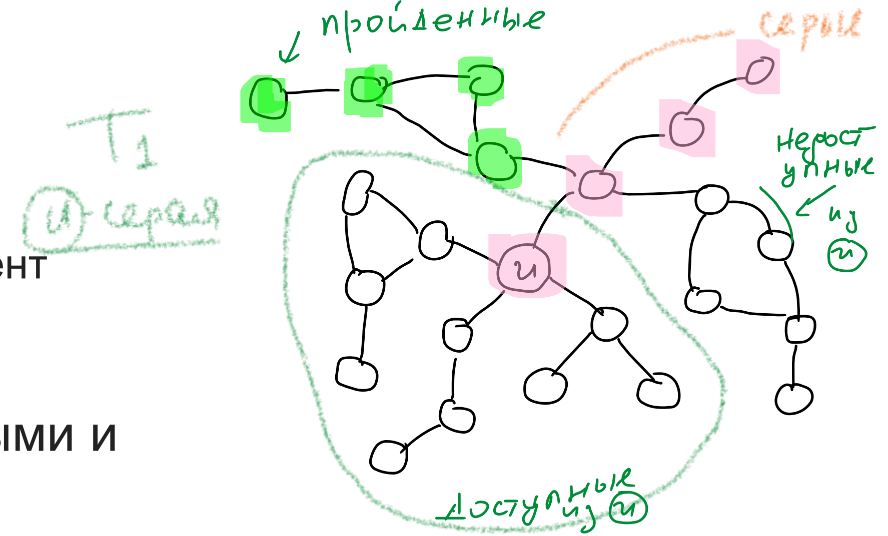
Тогда вершины графа **G\u**, бывшие черными и серыми в первый момент времени, не поменяют свой цвет ко второму моменту времени, а белые вершины либо останутся белыми, либо станут черными, причем черными станут те, что были достижимы от вершины **u** по белым путям.

ЛЕММА **О БЕЛЫХ ПУТЯХ**

Дан граф **G**. Запустим **dfs(G)**

- остановим выполнение процедуры dfs от некоторой серой вершины **u** - первый момент времени **T1**.
- продолжим выполнение dfs(u) пока u не станет черной - второй момент времени **T2**.

Тогда вершины графа **G\u**, бывшие черными и серыми в первый момент времени, не поменяют свой цвет ко второму моменту времени, а белые вершины либо останутся белыми, либо станут черными, причем черными станут те, что были достижимы от вершины **u** по белым путям.

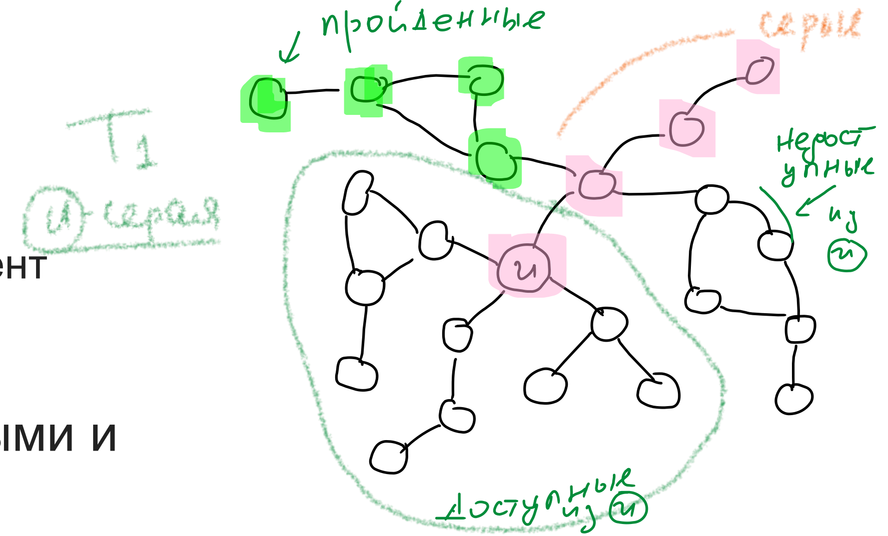


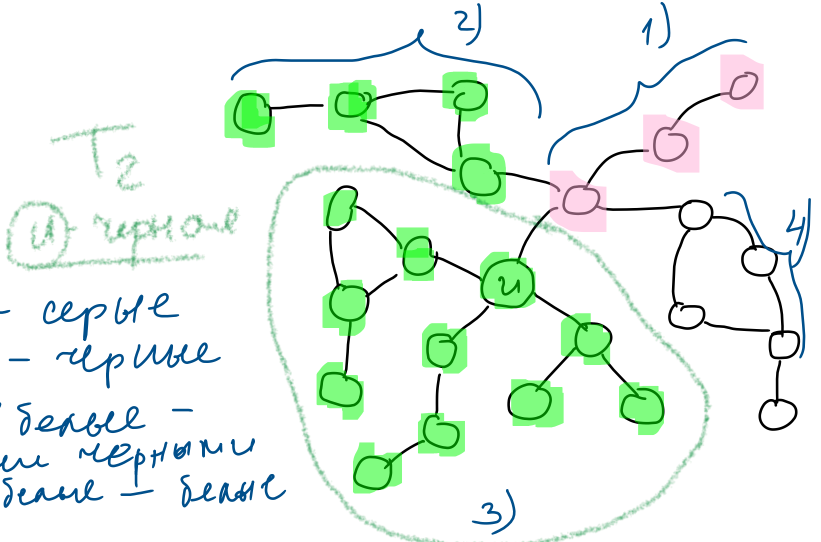
ЛЕММА **О БЕЛЫХ ПУТЯХ**

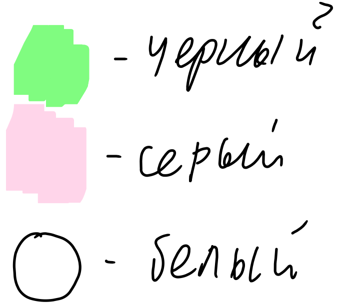
Дан граф **G**. Запустим **dfs(G)**

- остановим выполнение процедуры dfs от некоторой серой вершины **u** - первый момент времени **T1**.
- продолжим выполнение dfs(u) пока u не станет черной - второй момент времени **T2**.

Тогда вершины графа **G\u**, бывшие черными и серыми в первый момент времени, не поменяют свой цвет ко второму моменту времени, а белые вершины либо останутся белыми, либо станут черными, причем черными станут те, что были достижимы от вершины **u** по белым путям.







1) серые – серые
2) черные – черные
3) доступные белые – стали черными
4) Недостн. белые – белые

ЛЕММА **О БЕЛЫХ ПУТЯХ**

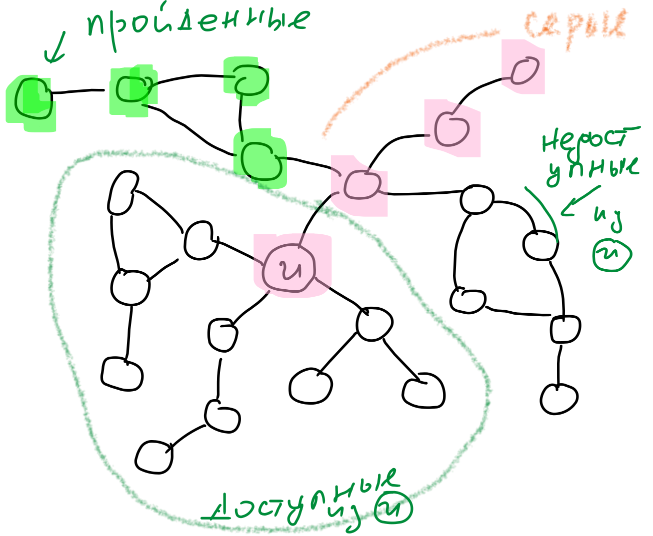
Дан граф **G**. Запустим **dfs(G)**

- остановим выполнение процедуры dfs от некоторой серой вершины **u** - первый момент времени **T1**.
- продолжим выполнение dfs(u) пока u не станет черной - второй момент времени **T2**.

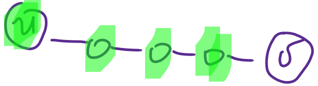
Тогда вершины графа **G\u**, бывшие черными и серыми в первый момент времени, не поменяют свой цвет ко второму моменту времени, а белые вершины либо останутся белыми, либо станут черными, причем черными станут те, что были достижимы от вершины **u** по белым путям.

T_1

u - серая

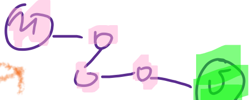


proof ⇒ Если v достижима в T1, то она станет черной в T2



по лемме

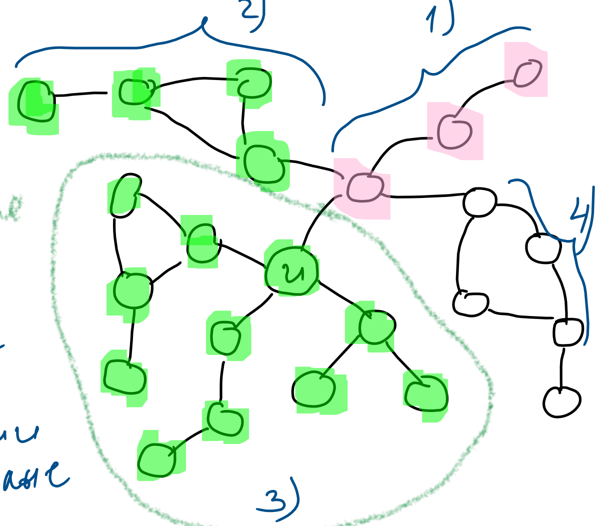
⇐ v стала черной в T2, то в T1 она была достижима из u



⇒ ∃ белый

T_2

u - черная



1) серые – серые
2) черные – черные
3) доступные белые – стали черными
4) недостижим. белые – белые

НАИКРАТЧАЙШИЕ ПУТИ

(без ребер): $v_1\ v_2\ v_5\ v_3 \dots v_k$

$\|$

$v_1 e_1 v_2 e_4 v_5 e_8 v_3 e_9 \dots v_k$

Путь (маршрут) – последовательность вершин и ребер вида

- длина пути количество ребер в нем (НЕ взвешенный граф) $= k$
- **вес пути** - сумма весом всех ребер пути (взвешенный граф) $= \sum_{i=1}^{k} W[u][v]$

$\boxed{W}$ – взвешанная матрица смежн-ти

Путь: **p = <V0, V1, V2, … Vk>** $|P| = k$

Вес пути **w(p)** $= \sum_{\substack{e_i \in P \\ i=1}} w(e_i) = \sum_{i=1}^{k} \underbrace{W[V_i][V_{i+1}]}_{V_i \text{ и } V_{i+1} \in P}$

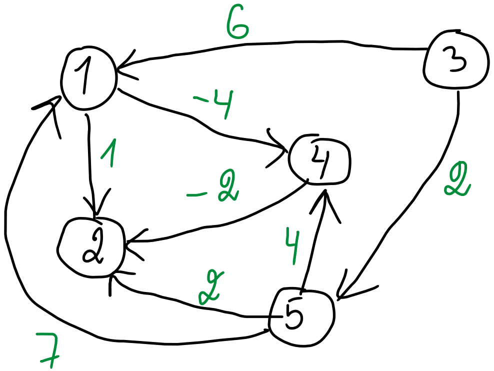
НаиКратчайший путь - путь с наименьшим весом (их может быть несколько разных, но с одним весом)

<u>Примечание</u>: вес наикратчайшего пути **из u в v** будет наименьшим из возможных или равен бесконечности, если пути из **u в v** нет

DAG: кратчайшие пути в ациклическом ориентированном графе

G	1	2	3	4	5
1	inf	**1**	inf	**-1**	inf
2	inf	inf	inf	inf	inf
3	**6**	inf	inf	inf	**2**
4	inf	**-2**	inf	inf	inf
5	**7**	**2**	inf	**4**	inf

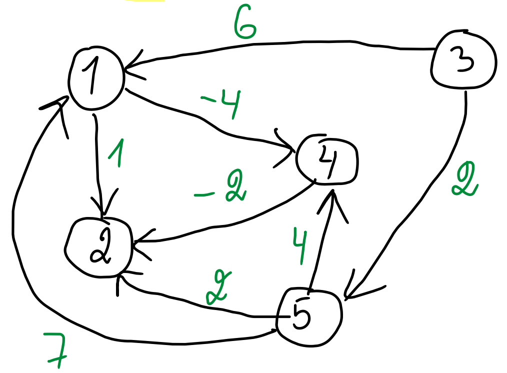
G граф



DAG: кратчайшие пути в ациклическом ориентированном графе

G	1	2	3	4	5
1	inf	1	inf	-1	inf
2	inf	inf	inf	inf	inf
3	6	inf	inf	inf	2
4	inf	-2	inf	inf	inf
5	7	2	inf	4	inf

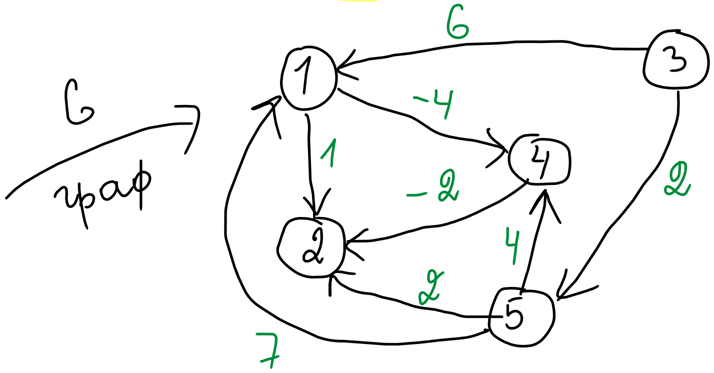
G граф



тополог. отсортируем!

DAG: кратчайшие пути в ациклическом ориентированном графе

G	1	2	3	4	5
1	inf	**1**	inf	**-1**	inf
2	inf	inf	inf	inf	inf
3	**6**	inf	inf	inf	**2**
4	inf	**-2**	inf	inf	inf
5	**7**	**2**	inf	**4**	inf

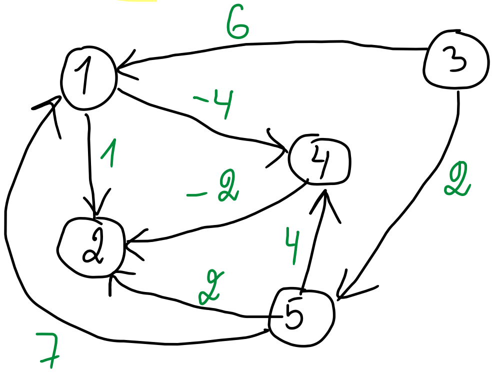


тополог. отсортируем! 3 5 1 4 2

DAG: кратчайшие пути в ациклическом ориентированном графе

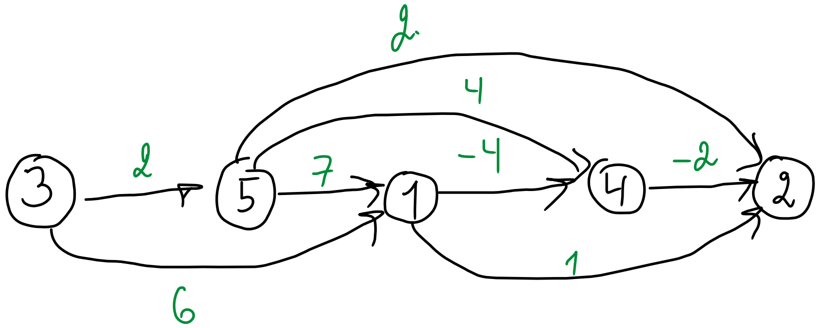
G	1	2	3	4	5
1	inf	1	inf	-1	inf
2	inf	inf	inf	inf	inf
3	6	inf	inf	inf	2
4	inf	-2	inf	inf	inf
5	7	2	inf	4	inf

G → граф



тополог. отсортируем! 3 5 1 4 2

граф G

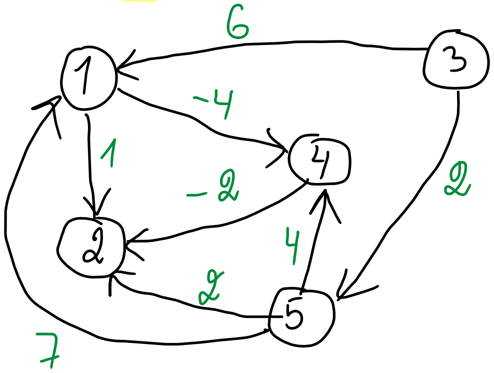


DAG: кратчайшие пути в ациклическом ориентированном графе

G	1	2	3	4	5
1	inf	1	inf	-1	inf
2	inf	inf	inf	inf	inf
3	6	inf	inf	inf	2
4	inf	-2	inf	inf	inf
5	7	2	inf	4	inf

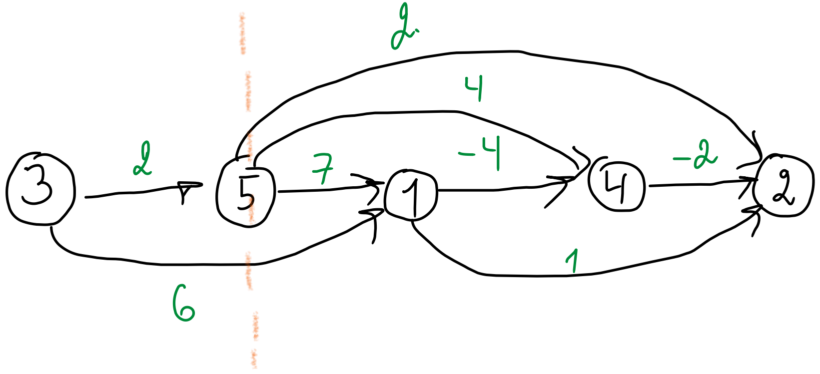
G

граф



тополог. отсортируем! 3 5 1 4 2

граф G



?Найти пути от 5 до всех?

все вершины до 5ки в топ сорт недостижимы из 5ки!

Спасибо за внимание!

www.ifmo.ru