Введение в графы. Обходы в глубину и ширину.

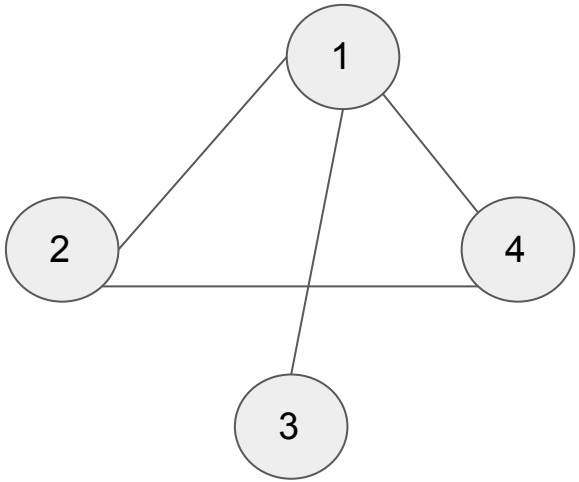
Преподаватель:
Макоян Артем Каренович

Графы. Основные определения.

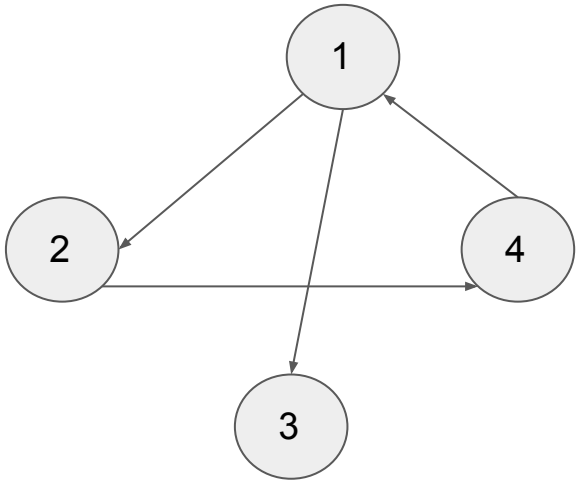
- Граф - вершины и ребра, соединяющие их.
- Примеры: города и дороги, их соединяющие; странички в вк и дружба пользователей; странички в инстаграме и подписки;
- G = (V, E) - граф, где V - множество вершин (обычно, будем нумеровать вершины числами от 1 до n), а E - множество ребер.
- Граф ориентированный, если ребра направленные (стрелки), иначе неориентированный.

По умолчанию будем считать, что граф неориентированный

Неориентированный граф G = (V, E), где V = {1, 2, 3, 4}, а E = {(1,2), (1,4), (1,3), (2,4)}



Ориентированный граф G = (V, E), где V = {1, 2, 3, 4}, а E = {(1,2), (4, 1), (1,3), (2,4)}

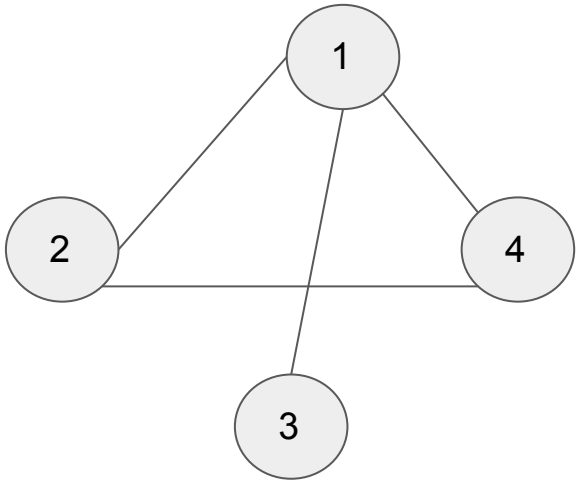


Графы. Пути и циклы.

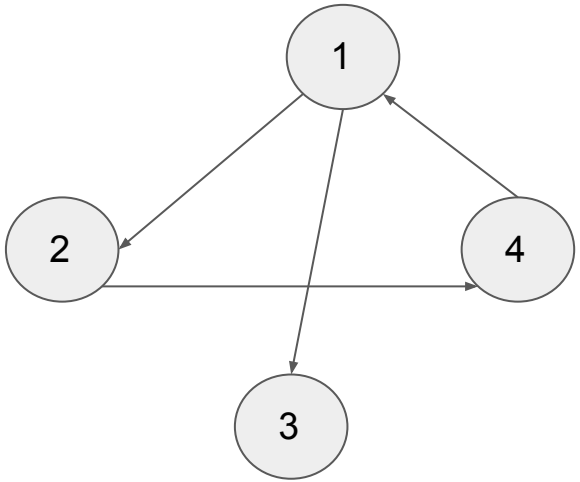
- Маршрут - это такая последовательность вершин, что из каждой можно пройти по ребру в следующую
- Путь - это маршрут, в котором каждая вершина встречается *один раз*
- Цикл - это маршрут, из последней вершины которого, можно пройти в первую
- Простой цикл - это цикл, в котором каждая вершина встречается *один раз*

В дальнейшем под циклом будем подразумевать простой цикл

Маршрут: 1, 2, 4, 1, 3
Путь: 4, 1, 3
Цикл: 2, 1, 3, 1, 4
Простой цикл: 1, 2, 4



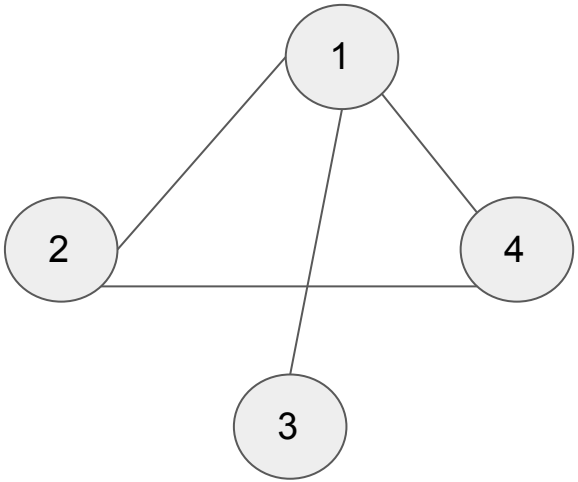
Маршрут: 1, 2, 4, 1, 3
Путь: 2, 4, 1, 3
Цикл: 1, 2, 4, 1, 2, 4
Простой цикл: 1, 2, 4



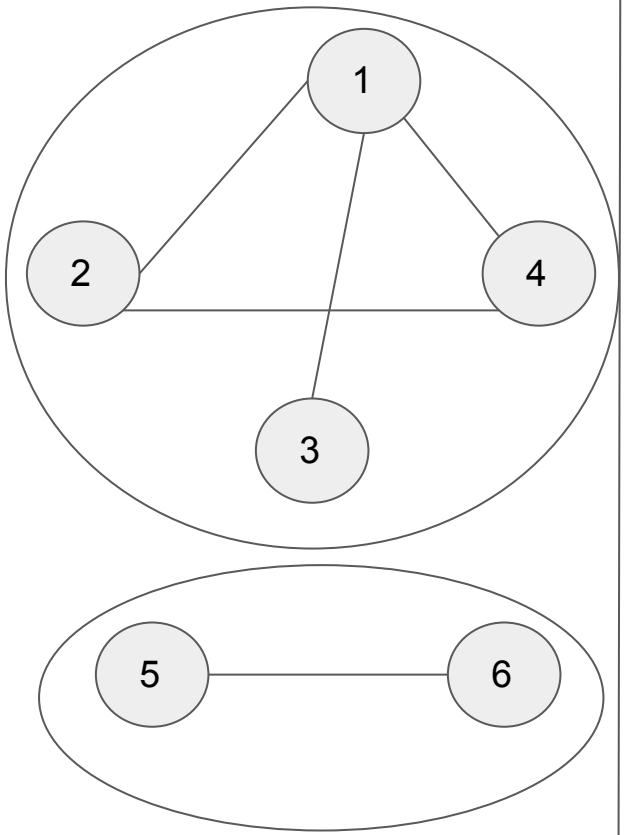
Графы. Определения.

- Связный граф - граф, между любыми двумя вершинами которого, есть маршрут
- Любой граф разбивается на связные подграфы, называемые компонентами связности
- Дерево - связный, ациклический (в котором нет циклов) граф. В дереве |E| = |V| - 1, то есть ребер на 1 меньше, чем вершин. Также между любыми 2-мя вершинами дерева есть ровно 1 путь.
- Граф называется двудольный, если его вершины можно покрасить в 2 цвета правильным образом (то есть концы любого ребра разноцветные). Вершины двудольного графа можно разбить на 2 множества (по цвету) так, чтобы внутри каждого из множеств не было ребер.

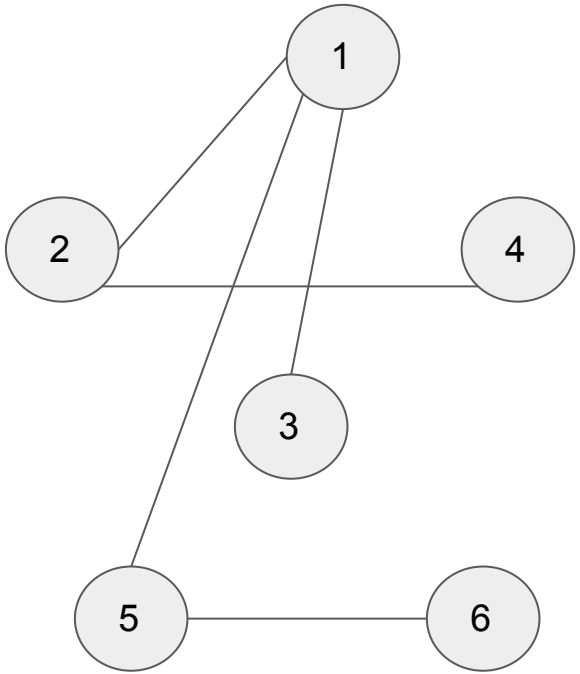
Связный граф



Несвязный граф



Дерево



Как хранить графы

- Список ребер: каждое ребро - это пара вершин, поэтому можно хранить список (вектор) пар чисел от 1 до n
- Матрица смежности: заведем матрицу nxn, где в пересечении i-ой строки и j-ого столбца будет стоять 1, если есть ребро из i в j, иначе 0.
- Список смежности: заведем двумерный вектор g, тогда g[i] - список (вектор) всех соседей i

Обычно используют список смежности, так как он занимает линейное количество памяти (в отличие от матрицы смежности), а также можно перебрать всех соседей вершины за их количество (в отличие от списка ребер и матрицы смежности).

```cpp
int readGraph() {
    // список ребер
    int n, m;
    cin >> n >> m;
    // n - количество вершин, m - количество ребер
    vector <pair <int, int> > edges(m);
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        /*
        ребро (u, v), но удобней нумерация с 0,
        то есть вершины от 0 до n - 1,
        поэтому из данных номеров вычитаем 1
        */
        edges[i] = {u - 1, v - 1};
    }
}
```

```cpp
int readGraph() {
    // матрица смежности
    int n, m;
    cin >> n >> m;
    // n - количество вершин, m - количество ребер
    vector <vector <int> > d(n, vector <int> (n, 0));
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        /*
        ребро (u, v), но удобней нумерация с 0,
        то есть вершины от 0 до n - 1,
        поэтому из данных номеров вычитаем 1
        */
        d[u - 1][v - 1] = 1;
        // ЕСЛИ ГРАФ НЕОРИЕНТИРОВАННЫЙ, ДОБАВЛЯЕМ
        {
            d[v - 1][u - 1] = 1;
        }
    }
}
```

```cpp
int readGraph() {
    // список смежности
    int n, m;
    cin >> n >> m;
    // n - количество вершин, m - количество ребер

    // изначально создаем n пустых векторов
    vector <vector <int> > g(n);
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        /*
        ребро (u, v), но удобней нумерация с 0,
        то есть вершины от 0 до n - 1,
        поэтому из данных номеров вычитаем 1
        */
        // v - сосед u, поэтому добавляем v в список соседей u - g[u - 1]
        g[u - 1].push_back(v - 1);
        // ЕСЛИ ГРАФ НЕОРИЕНТИРОВАННЫЙ, ДОБАВЛЯЕМ
        {
            g[v - 1].push_back(u - 1);
        }
    }
}
```

Обход в ширину (bfs)

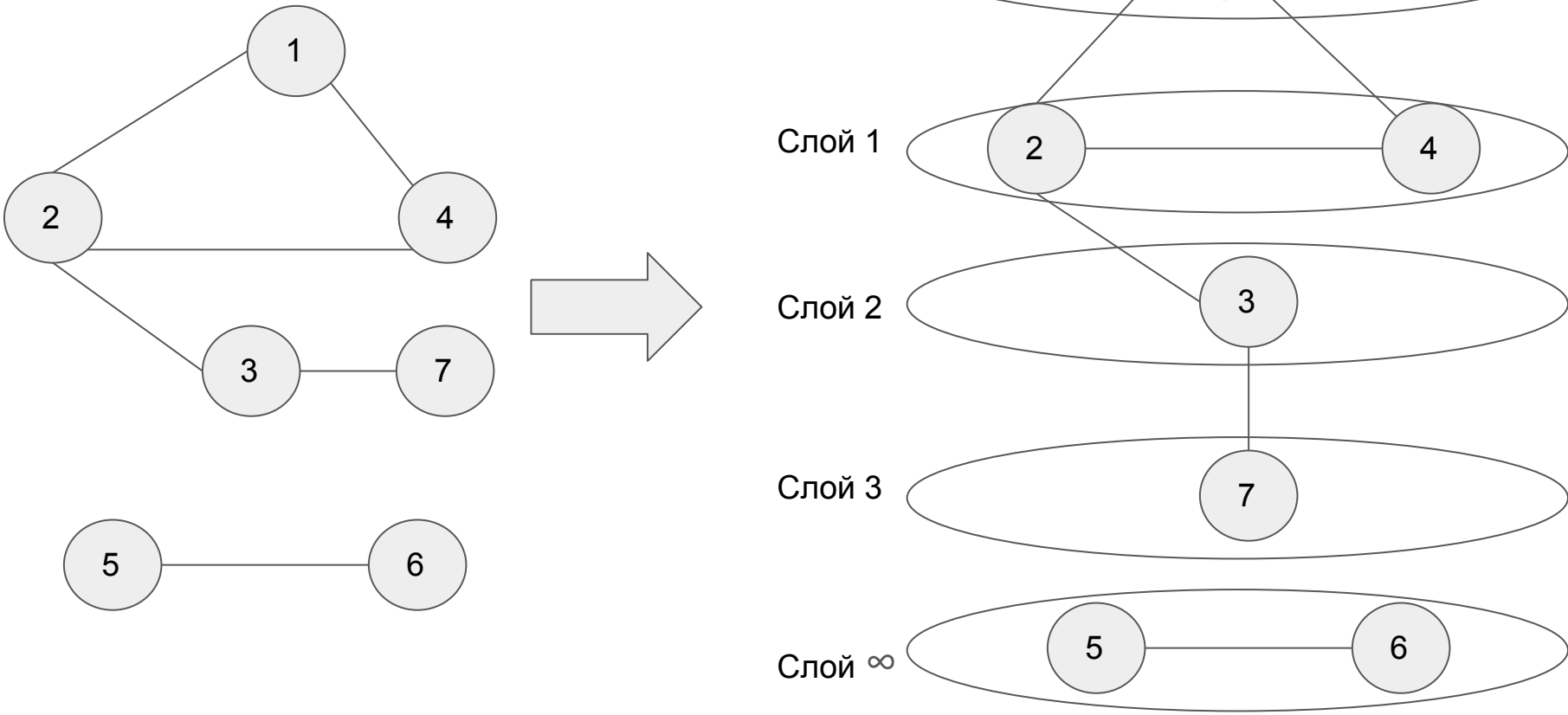
Обойти граф = побывать в каждой его вершине по 1 разу.

Пусть у нас есть мешок, в котором сложены все вершины. Возьмем из него вершину 1 и положим в нулевой слой.

Шаг n:

Посмотрим на всех соседей слоя n, находящихся в мешке (то есть всех соседей, которых мы до этого не доставали). Возьмем их из мешка и положим в слой номер n + 1. Если таких соседей нету, закончим алгоритм.

Тогда мы разделили компоненту связности 1 на слои (вершины из других компонент связности остались в мешке или в слое номер ∞).


1
2
4
3
7
5
6
Слой 0
Слой 1
Слой 2
Слой 3
Слой ∞

Алгоритм bfs

- Заведем массив d, где будем хранить номер слоя вершины (d[i] = INF, если i еще в мешке, d[i] = n, если вершина в слое n). Изначально d[1] = 0, а d[i] = INF, если i != 1.
- Нам нужно просматривать всех соседей и класть их в нужный слой, если они в мешке. Заведем очередь q, в которой будем хранить все вершины, которые мы уже достали из мешка, но еще не посмотрели из нее соседей. Изначально в q лежит только 1.
- Тогда каждым шагом (пока очередь не пустая): достаем из низа очереди вершину v, просматриваем из нее всех соседей и если сосед u лежит в мешке (то есть d[u] = INF), то кладем его в слой на 1 больше чем у v (то есть d[u] = d[v] + 1) и добавляем в конец очереди. Если мы достали u из мешка, то говорят, что мы **релаксируем u по ребру из v**.

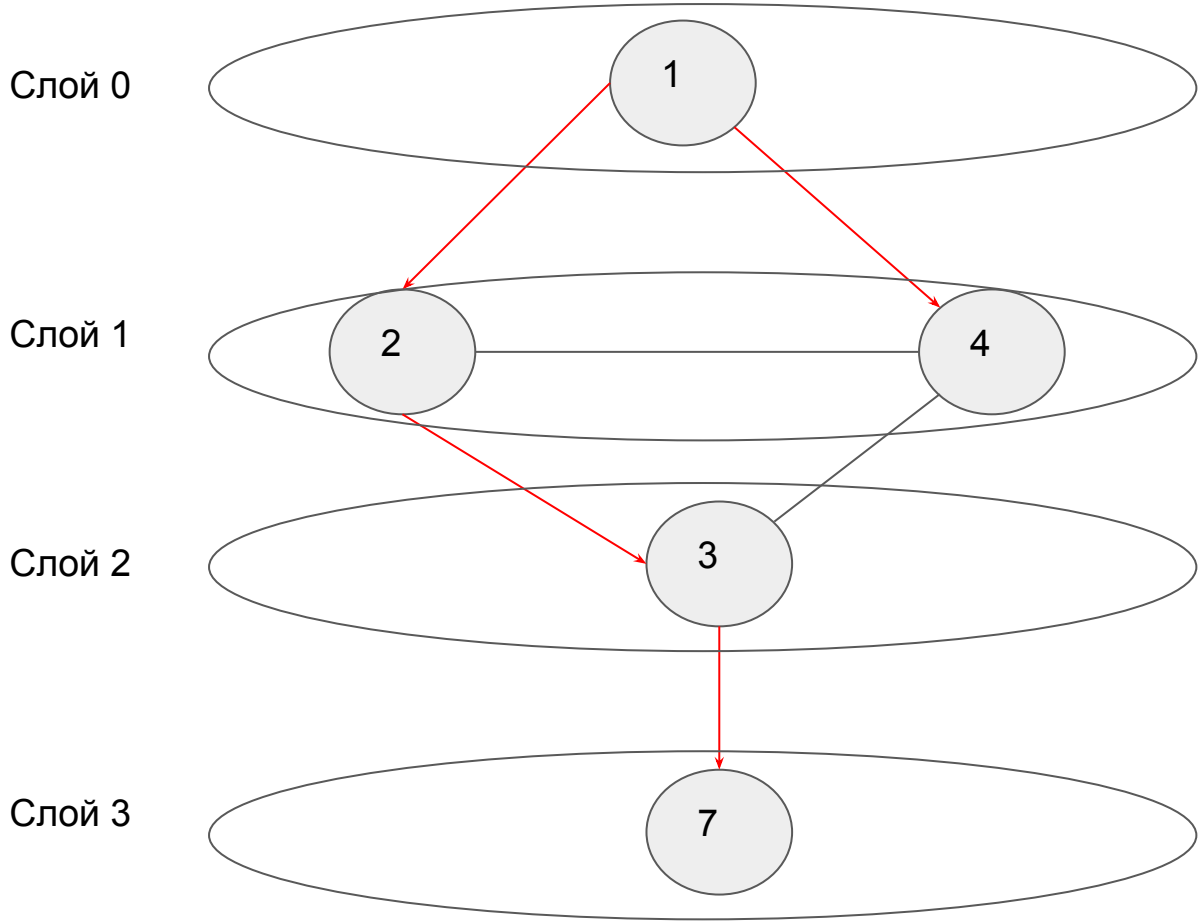
```cpp
const int INF = 1e9;

vector <int> bfs(int n, vector <vector <int> > g) {
    /*
    n - количество вершин
    g - список смежности
    */
    vector <int> d(n, INF);
    d[0] = 0;
    queue <int> q;
    q.push(0);
    while (!q.empty()) {
        // достаем вершину из очереди
        int v = q.front();
        q.pop();
        // пробегаемся по всем соседям v
        for (auto u : g[v]) {
            // проверяем, что u в мешке
            if (d[u] == INF) {
                /*
                достаем u из мешка и кладем в слой d[v] + 1
                добавляем u в очередь, чтобы потом из нее
                просмотреть всех соседей
                */
                d[u] = d[v] + 1;
                q.push(u);
            }
        }
    }
    return d;
}
```

Алгоритм bfs

Можно заметить, что номер слоя каждой вершины - это расстояние от нее до 1 (то есть длина кратчайшего пути от 1 до нее). То есть мы научились считать **расстояние от 1 до всех остальных вершин**.

Пусть граф был связный (то есть каждая вершина попала в какой-то слой). Давайте посмотрим на все ребра, по которым мы релаксировали. Тогда если мы оставим только их, то в каждую вершину (кроме 1) будет вести ровно одно ребро, причем по этим ребрам можно добраться из 1 до любой другой. Тогда образовалось **подвешенное дерево с корнем в 1**, то есть если оставить только ребра, по которым мы релаксировали, то получим дерево с корнем 1.


Слой 0
1
Слой 1
2
4
Слой 2
3
Слой 3
7

bfs - итог

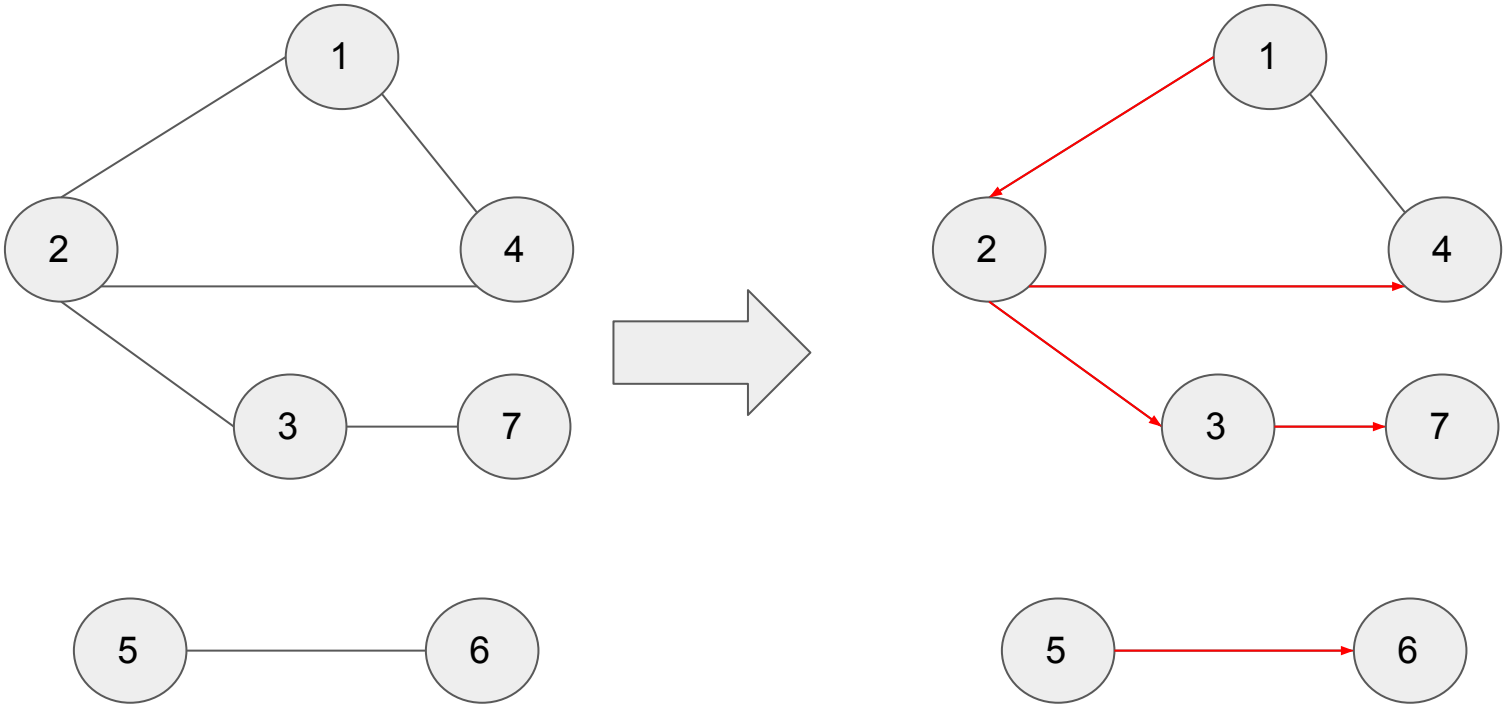
- По сути мы выделили **остовное дерево**, то есть выбросили из графа некоторые ребра и осталось дерево.
- Любое ребро не “перепрыгивает” между слоями, то есть номера слоев концов каждого ребра отличаются не больше чем на 1 (|d[u] - d[v]| <= 1)
- Расстояние от 1 до любой другой вершины в дереве bfs-а совпадает с расстоянием от 1 до этой вершины в изначальном графе и равно номеру слоя этой вершины (то есть записано в массиве d)
- Для ориентированного расстояние считается точно так же

Мы научились считать расстояние от одной вершины до другой за O(n + m)

Алгоритм dfs

Рассмотрим другой обход:

- Изначально стоим в вершине 1.
- Каждым шагом идем в соседнюю вершину, в которой не были.
- Если таких вершин нету, возвращаемся в вершину, откуда пришли.
- Если мы в вершине 1 и побывали во всех ее соседях, то просто заканчиваем алгоритм (так как мы в 1 ниоткуда не приходили)
- Если нужно обойти весь граф, а не только компоненту связности 1, будем запускать dfs от не посещенной вершины, пока такие есть


1
2
4
3
7
5
6
1
2
4
3
7
5
6

```cpp
#include <bits/stdc++.h>

using namespace std;

vector <bool> used;
vector <vector <int> > g;
/*
g - список смежности
used - массив пометок, то есть
used[u] = true, мы уже были в u
used[u] = false, в u еще не были
*/

void dfs(int u) {
    used[u] = true;
    for (auto v : g[u]) {
        if (!used[v]) {
            dfs(v);
        }
    }
}

int main() {
    int n, m;
    cin >> n >> m;
    g.resize(n);
    used.resize(n, false);
    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        u--, v--;
        g[u].push_back(v);
        g[v].push_back(u);
    }
    for (int i = 0; i < n; ++i) {
        if (!used[i]) {
            dfs(i);
        }
    }
}

```

Алгоритм dfs

- **Непосредственный предок** вершины v - это вершина u, откуда мы пришли в v в процессе dfs-а
- **Предок** вершины u - вершина, до которой можно добраться идя “вверх”, то есть идя по непосредственным предкам
- Тогда любое ребро соединяет вершину с ее предком или потомком (то есть оно “вертикальное”). Такие ребра называют прямыми.
- В ориентированном графе есть не только **прямые** ребра, но и **перекрестные** (соединяют вершины без связи “предок-потомок”).

dfs с таймером

- Все свойства здесь будут верны для ориентированного графа тоже
- Пусть в процессе dfs-а мы запустили “таймер” (то есть поддерживаем переменную и когда переходим в другую вершину - увеличиваем ее на 1). Тогда у каждой вершины есть время входа в нее и время выхода, а значит каждой вершине соответствует отрезок жизни (отрезок времени [время входа, время выхода]). Оказывается любые 2 отрезка либо не пересекаются, либо один вложен в другой (в этом случае больший отрезок является предком меньшего)
- Тогда если в текущий момент отрезок жизни вершины открыт, то мы находимся в ее потомке (то есть в ее поддереве)