ГРАФЫ
КРАТЧАЙШИЕ ПУТИ

Школа::Кода

Олимпиадное программирование

2020-2021 Таганрог

Рассматриваемые алгоритмы

- Алгоритм Дейкстры

  Находит кратчайшее расстояние от заданной вершины до всех остальных, базируется на обходе в ширину.

  Асимптотика $O(M \log(N))$.

- Алгоритм Флойда-Уоршелла

  Находит кратчайшее расстояние между всеми парами вершин.

  Асимптотика $O(N^3)$.

Приоритетная очередь (std::priority_queue)

1. Требуется подключение библиотеки #include <queue>
2. По умолчанию первый объект в очереди – самый наибольший.
3. Объявление объекта приоритетной очереди
   priority_queue<‘тип данных’> que;
   или
   priority_queue<‘тип данных’, ’тип контейнера’<‘тип данных’>, ‘тип компаратора’<‘тип данных’ >> que;
   По умолчанию ‘тип контейнера’ – std::vector, а ‘тип компаратора’ – std::less.
4. Добавление объекта в очередь que.push(obj);
5. Получение первого объекта из очереди que.top();
6. Удаление первого элемента из очереди que.pop();
7. Проверка, пустая ли очередь que.empty();

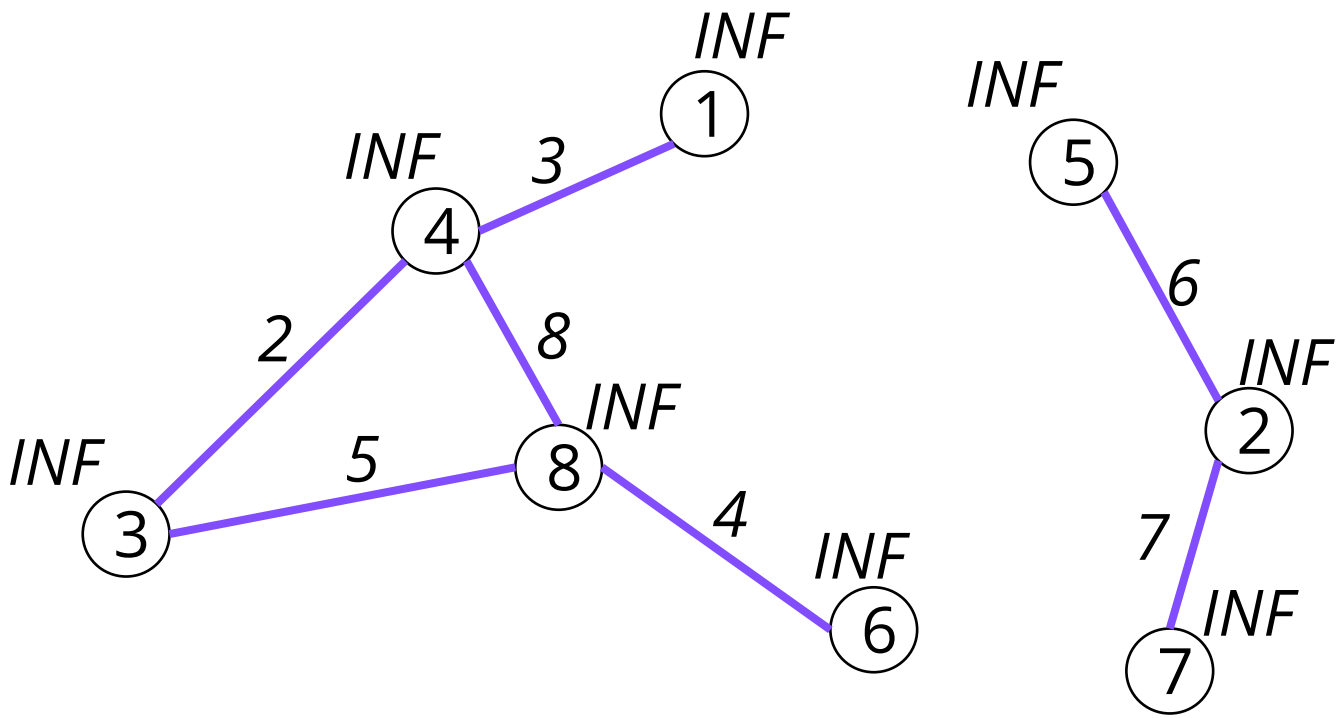
Алгоритм Дейкстры

- Для каждой вершины будем хранить расстояние до неё. Изначально расстояние до всех вершин равно INF (очень большая константа), а до стартовой – 0.
- В каждой вершине помимо списка смежных с ней вершин будем хранить и длину ребра, которым они соединены.
- Во время обхода будем использовать не обычную, а приоритетную очередь. Хранить в ней будем пары чисел <‘расстояние до вершины’,’номер вершины’>.
- Положим в очередь пару <0, ‘стартовая вершина’>, следующие пункты будем повторять, пока очередь не опустеет.
- Вынем из очереди верхнюю пару. Если расстояние в этой паре не больше, чем текущее в вершине из этой пары, то перейдём к следующему шагу. Иначе – повторим текущий.
- Переберём все смежные вершины. Если расстояние до текущей вершины + длина ребра до смежной меньше текущего расстояния в смежной вершине, то обновим в ней расстояние и добавим в очередь пару <‘новое расстояние до смежной вершины’,’номер смежной вершины’>. Если для решения задачи нужно будет знать путь, то на этом шаге нужно запомнить, что предком смежной вершины является текущая.

Алгоритм Дейкстры. Пример. Шаг 0

Приоритетная очередь:

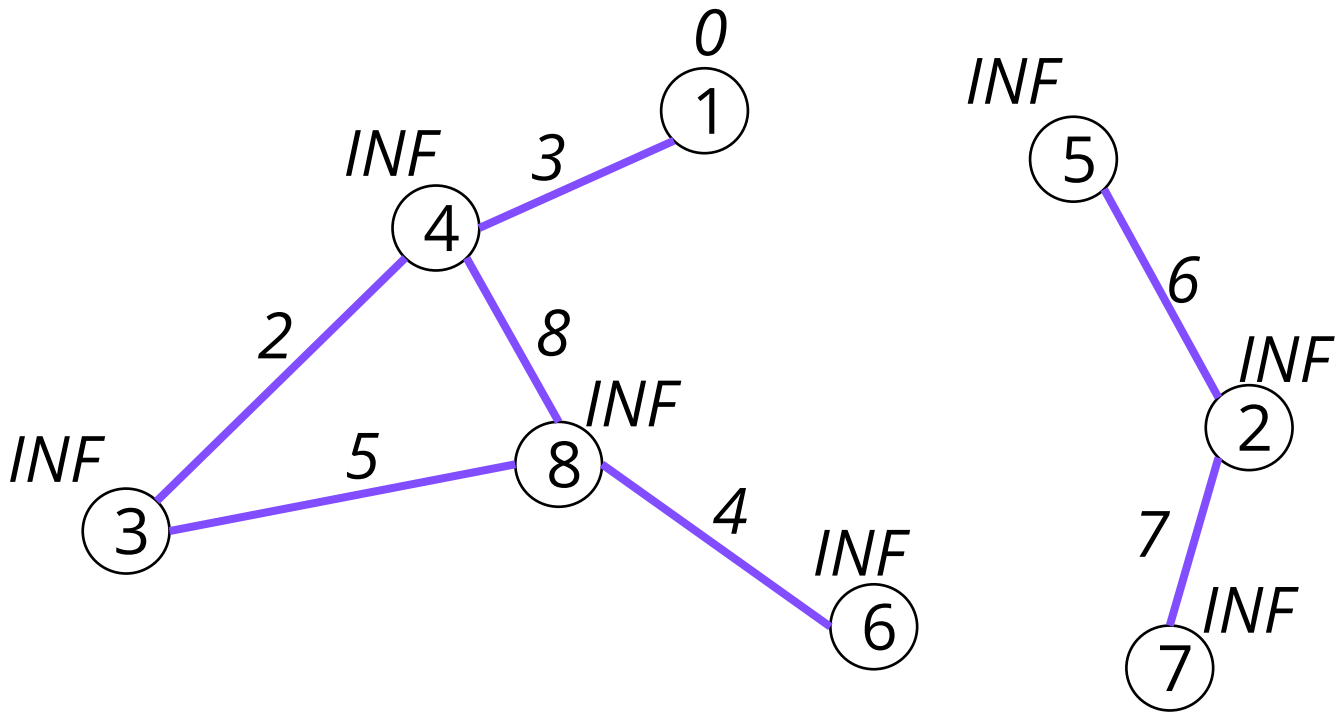
<0, 1>



Алгоритм Дейкстры. Пример. Шаг 1

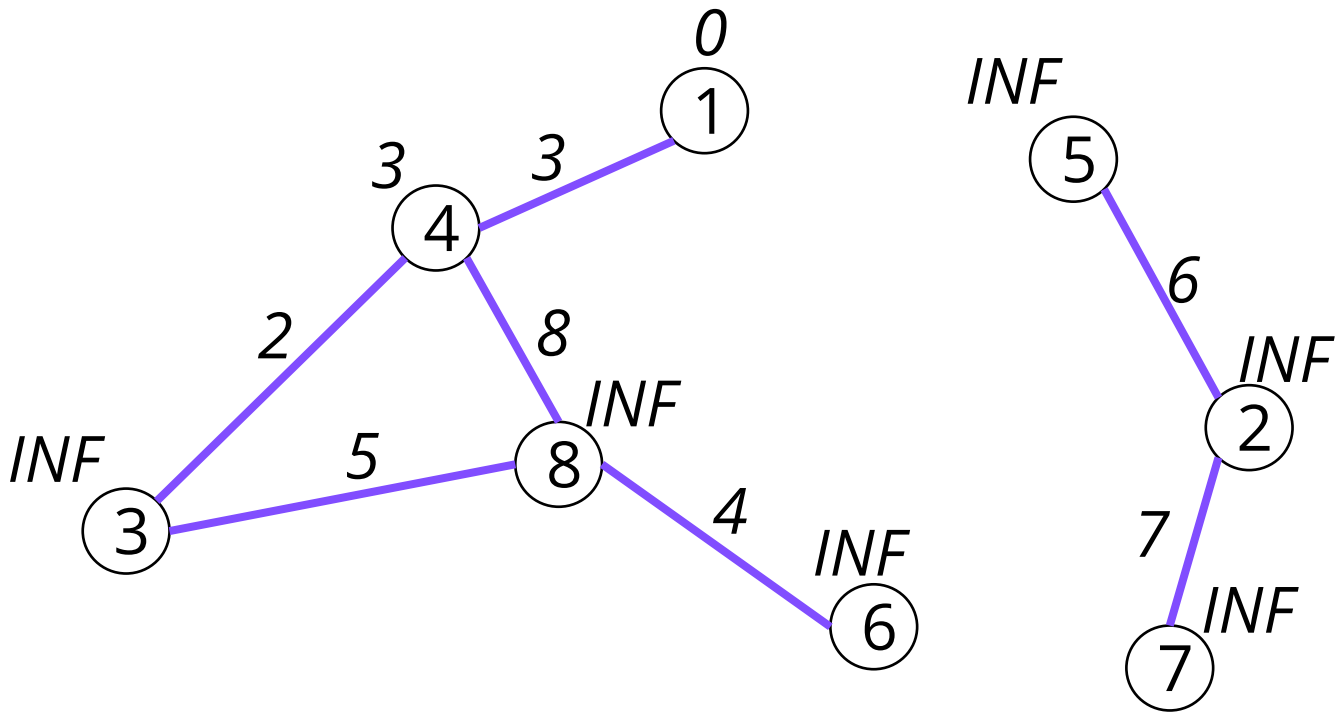
Приоритетная очередь:
<empty>

Рассматриваемая пара:
<0, 1>



Алгоритм Дейкстры. Пример. Шаг 2

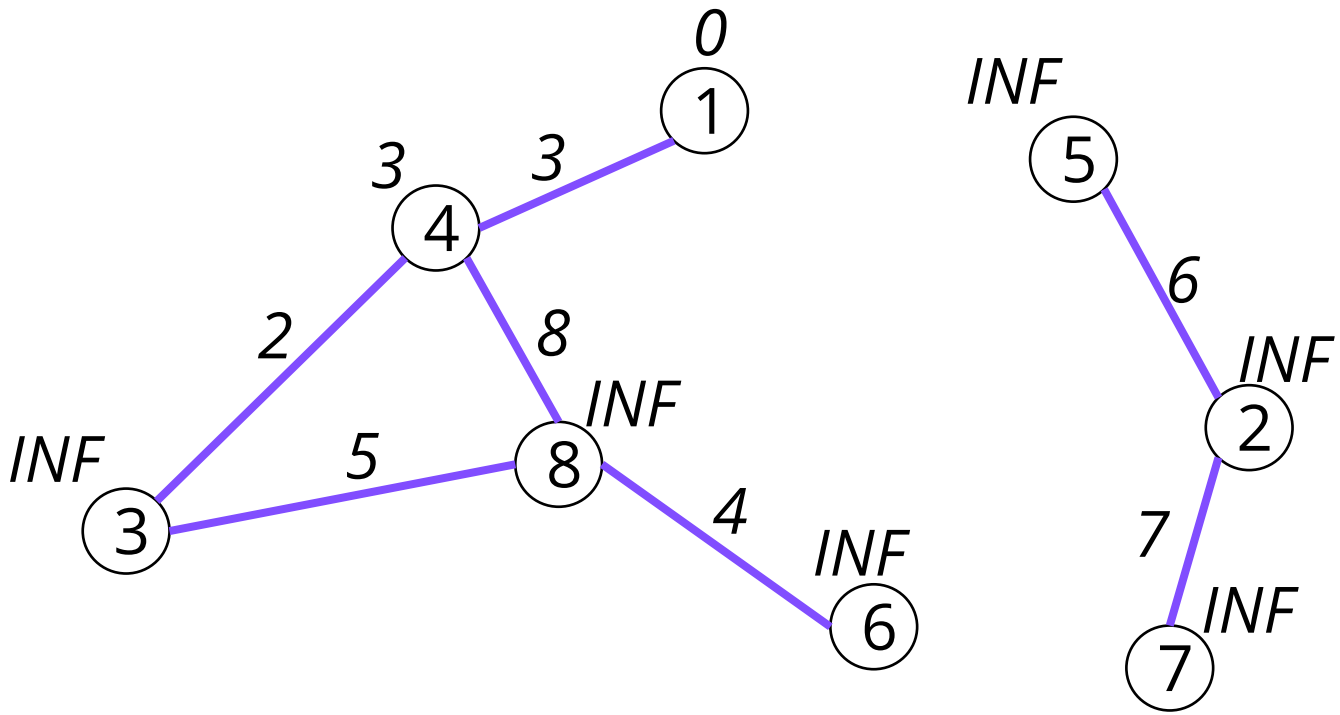
Приоритетная очередь:
<3, 4>



Алгоритм Дейкстры. Пример. Шаг 3

Приоритетная очередь:
<empty>

Рассматриваемая пара:
<3, 4>

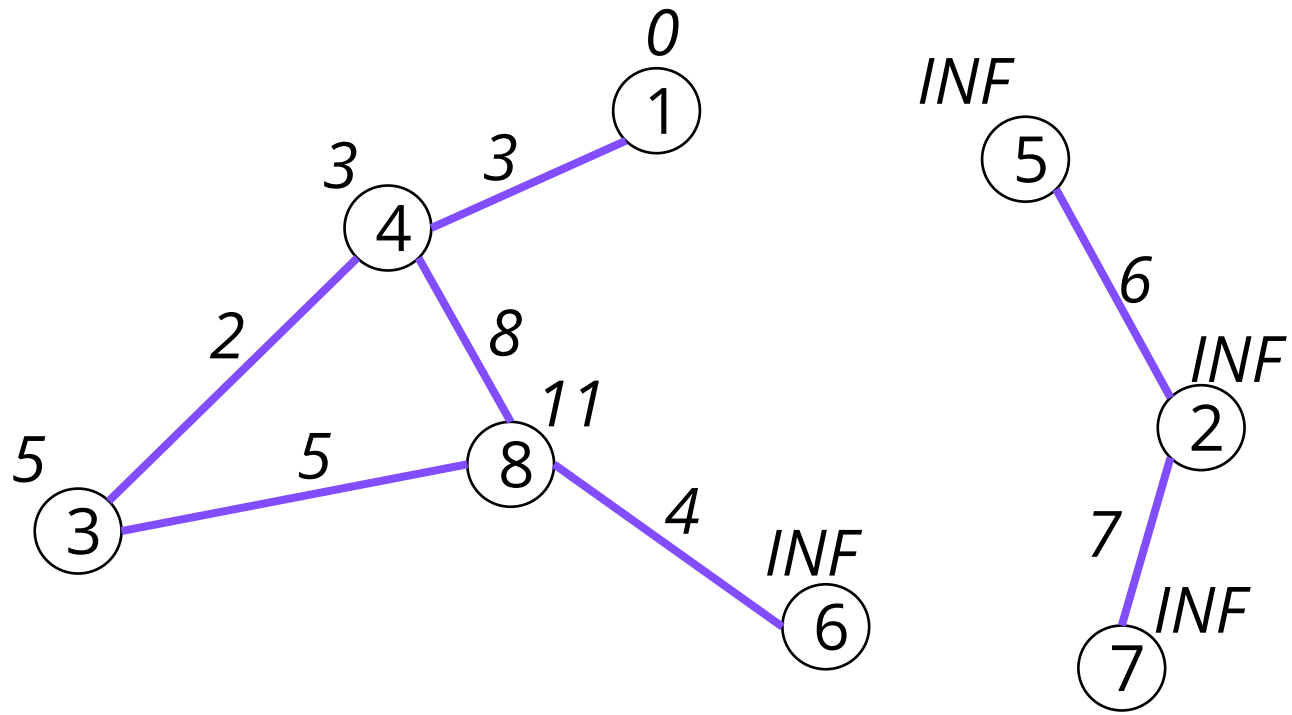


Алгоритм Дейкстры. Пример. Шаг 4

Приоритетная очередь:

<5, 3>

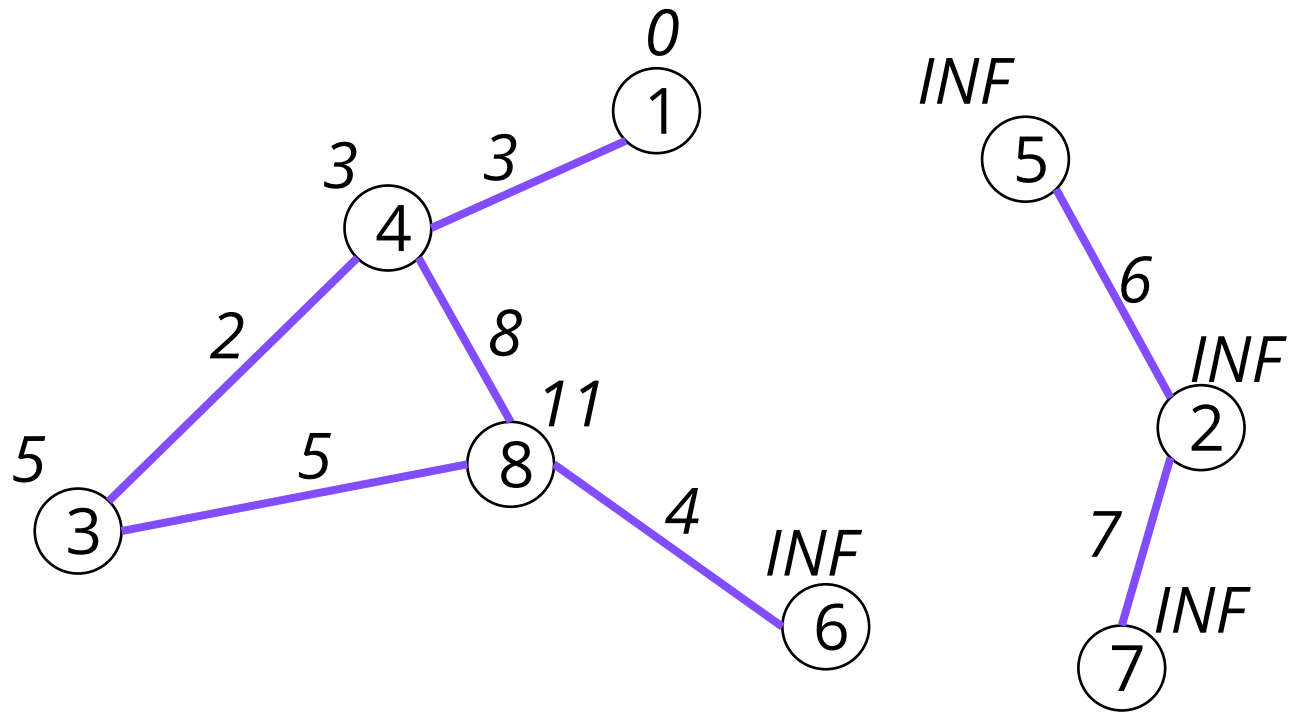
<11, 8>



Алгоритм Дейкстры. Пример. Шаг 5

Приоритетная очередь:
<11, 8>

Рассматриваемая пара:
<5, 3>

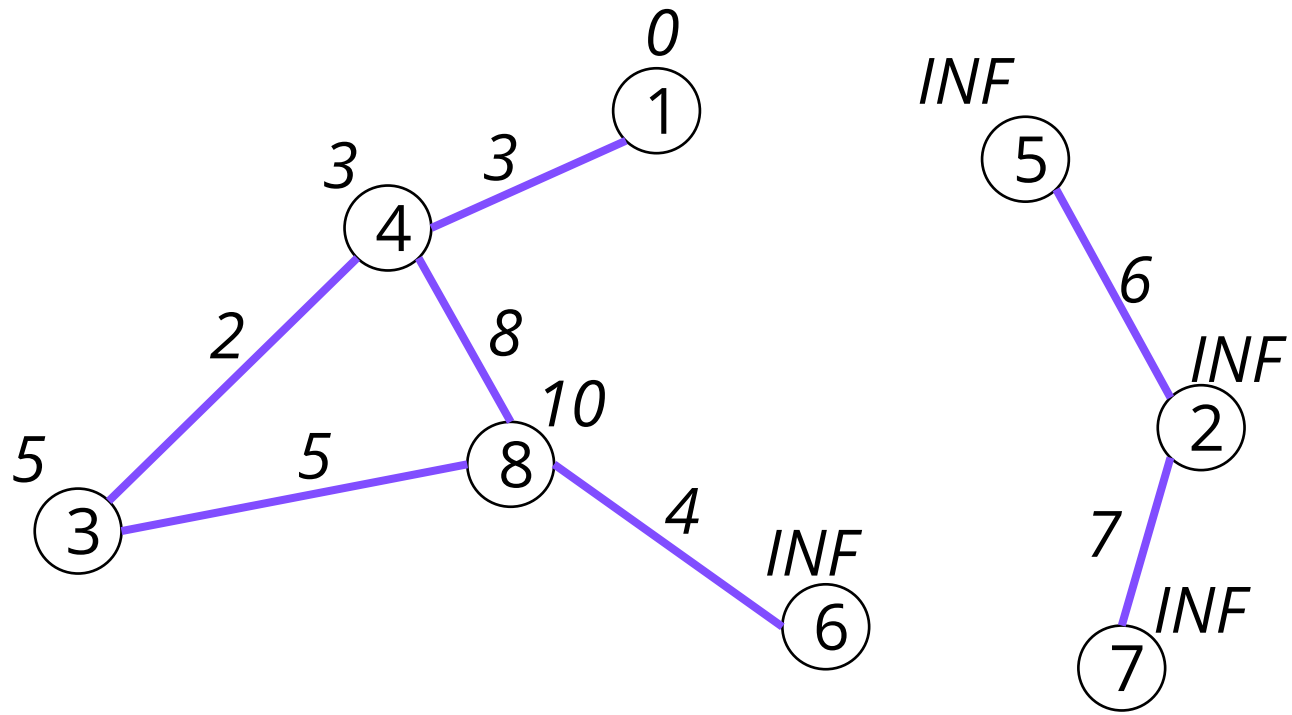


Алгоритм Дейкстры. Пример. Шаг 6

Приоритетная очередь:

<10, 8>

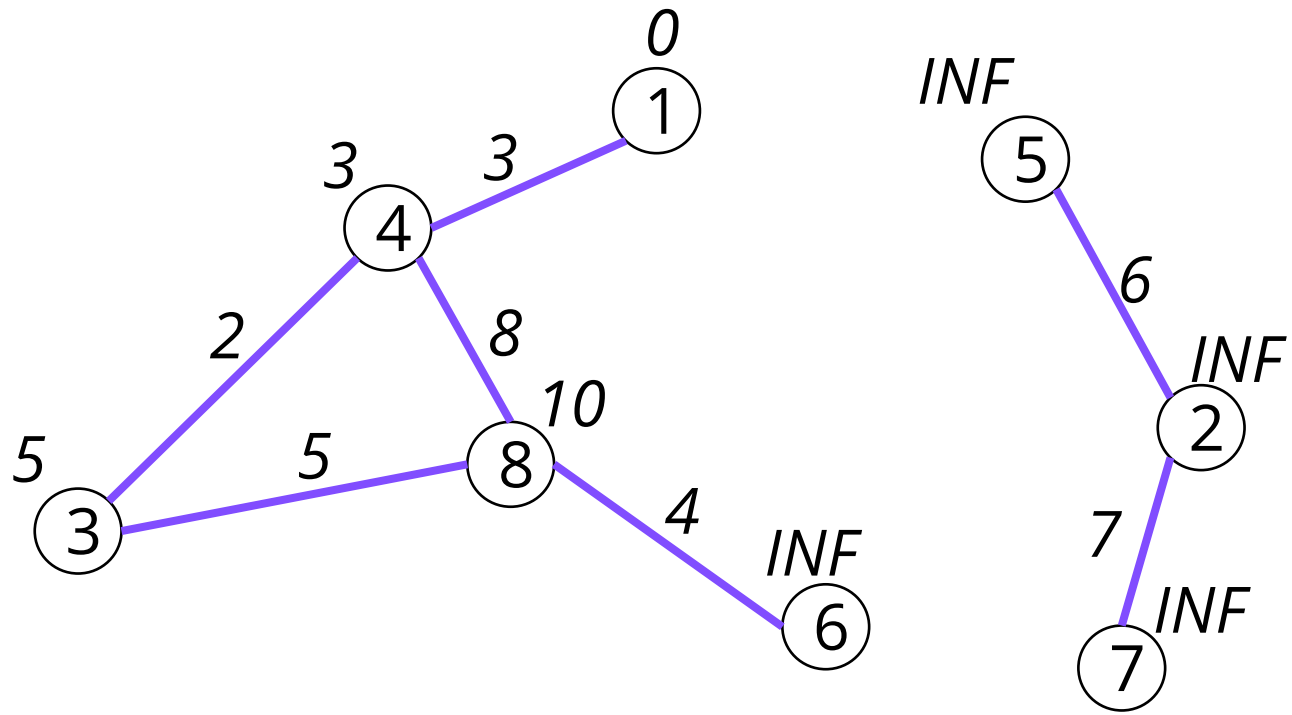
<11, 8>



Алгоритм Дейкстры. Пример. Шаг 7

Приоритетная очередь:
<11, 8>

Рассматриваемая пара:
<10, 8>

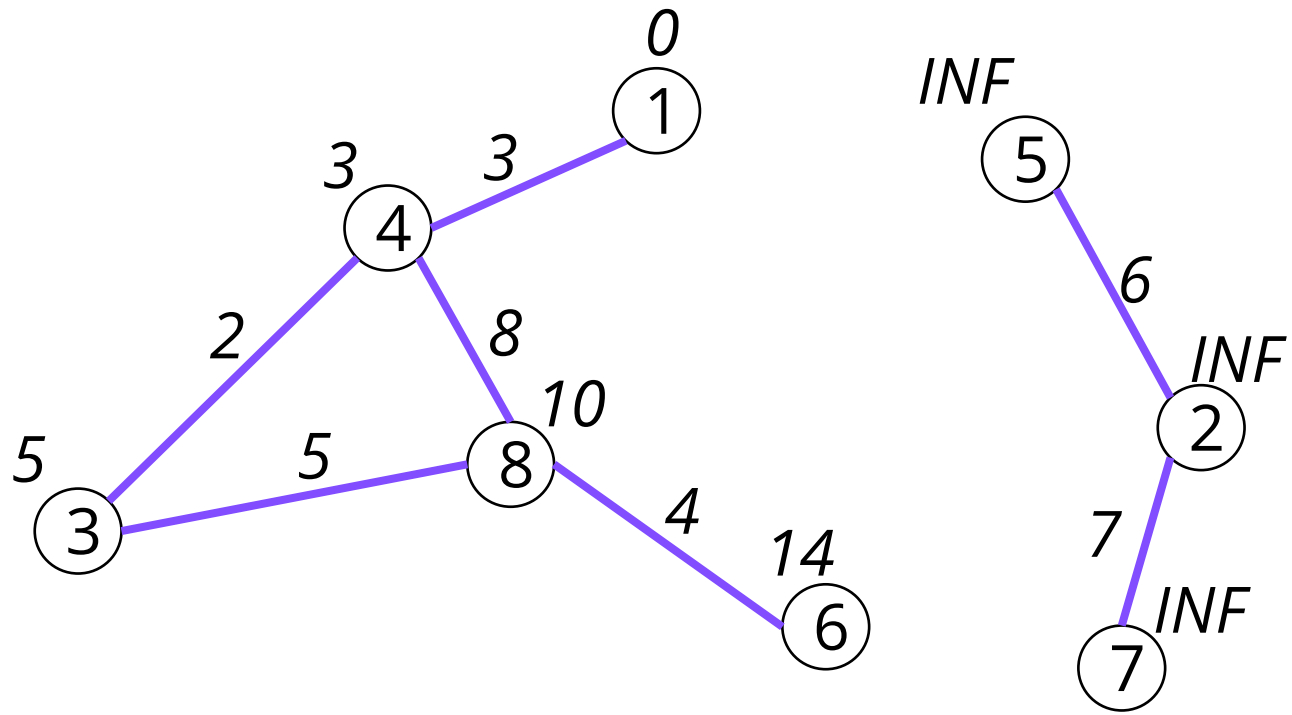


Алгоритм Дейкстры. Пример. Шаг 8

Приоритетная очередь:

<11, 8>

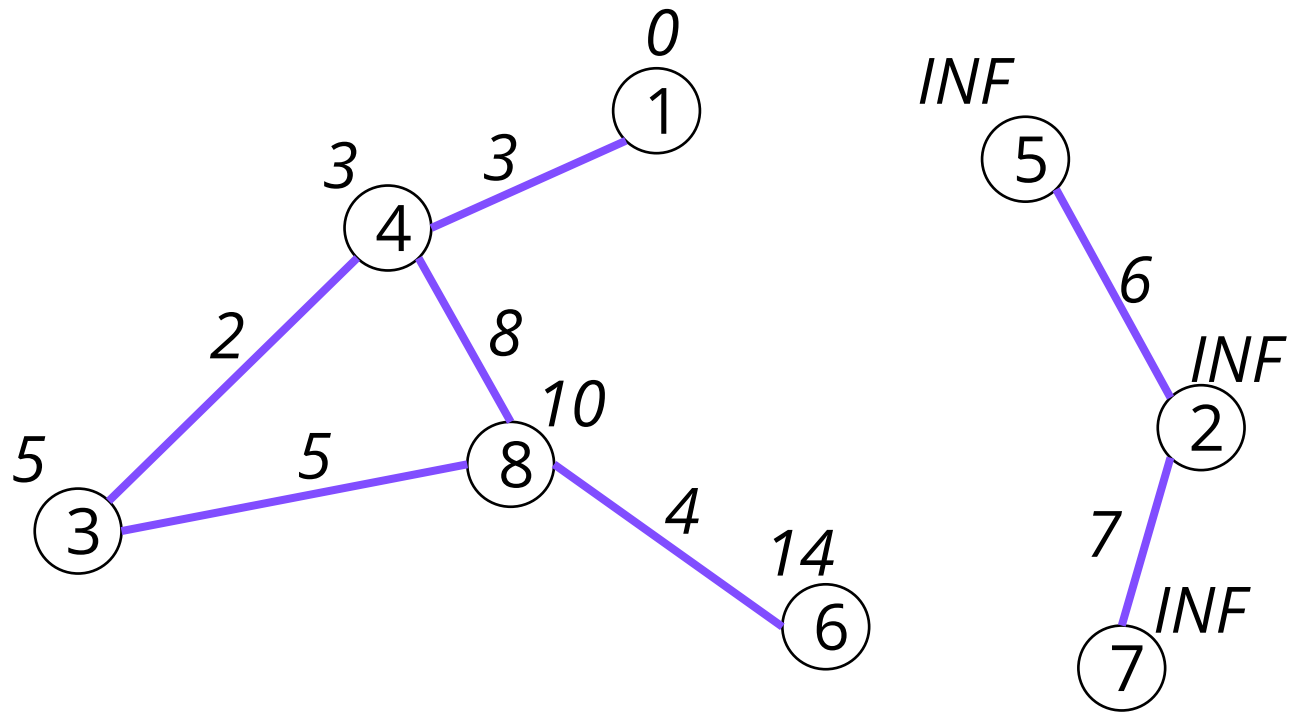
<14, 6>



Алгоритм Дейкстры. Пример. Шаг 9

Приоритетная очередь:
<14, 6>

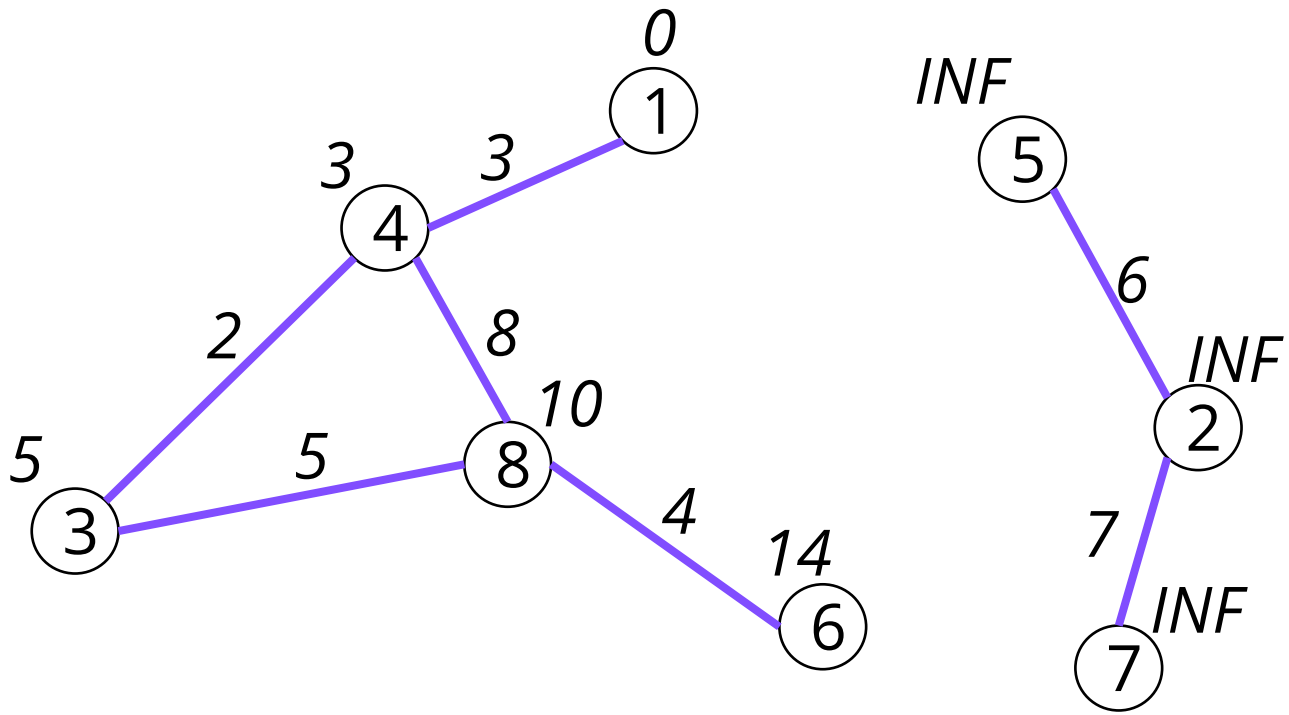
Рассматриваемая пара:
<11, 8>



Алгоритм Дейкстры. Пример. Шаг 10

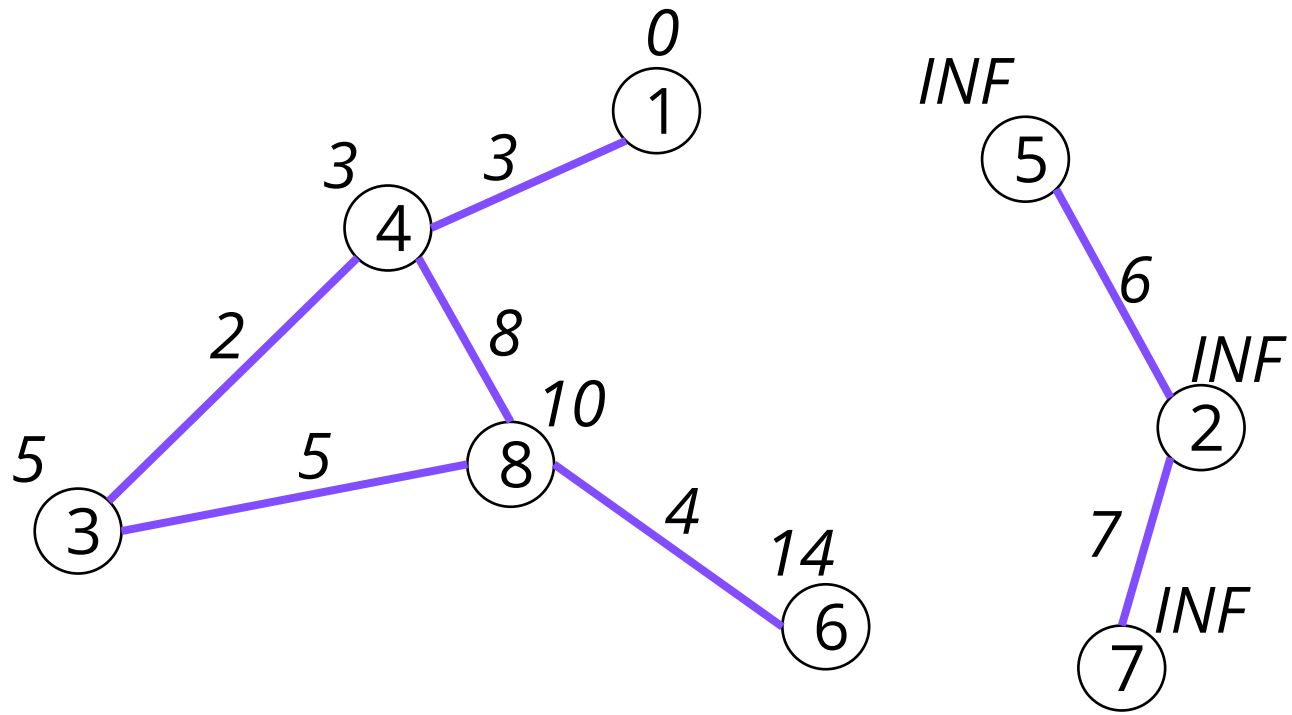
Приоритетная очередь:
<empty>

Рассматриваемая пара:
<14, 6>



Алгоритм Дейкстры. Пример. Шаг 11

Приоритетная очередь:
<empty>



Задача

Пусть дан ненаправленный взвешенный граф из N вершин и M рёбер. Рёбра описываются числами U, V и D, которые означают, что между вершинами с номерами U и V есть ребро длиной D. Далее даны два числа S и F – номера вершин, между которыми нужно найти кратчайший путь.

Решение

Напишем алгоритм Дейкстры с запоминанием предка. Запустим алгоритм из вершины S, после чего восстановим путь перемещаясь по предкам из вершины F.

Реализация

Структура графа

```
const int INF = 1e9;
struct Node
{
    vector<pair<int, int>> to;
    int dist = INF;
    int parent = -1;
};
vector<Node> g;
```

Алгоритм Дейкстры

```
void  dijkstra(int s)
{
    priority_queue<pair<int, int>> q;
    q.push(make_pair(0, s));
    g[s].dist = 0;
    while (!q.empty())
    {
        int d = -q.top().first;
        int v = q.top().second;
        q.pop();
        if (g[v].dist < d)
            continue;
        for (int i = 0; i < g[v].to.size(); ++i)
        {
            int u = g[v].to[i].first;
            int new_d = d + g[v].to[i].second;
            if (new_d < g[u].dist)
            {
                g[u].dist = new_d;
                g[u].parent = v;
                q.push(make_pair(-new_d, u));
            }
        }
    }
}
```

Реализация

Ввод графа и запуск алгоритма

```
int main()
{
    int n, m;
    cin >> n >> m;
    g.resize(n);
    for (int i = 0; i < m; ++i)
    {
        int u, v, d;
        cin >> u >> v >> d;
        --u; --v;
        g[u].to.push_back(make_pair(v, d));
        g[v].to.push_back(make_pair(u, d));
    }
    int s, f;
    cin >> s >> f;
    --s; --f;
    dijkstra(s);
```

Восстановление пути

```
if (g[f].dist == INF)
{
    cout << -1 << endl;
    return 0;
}
vector<int> path;
path.push_back(f);
while (g[path.back()].parent != -1)
    path.push_back(g[path.back()].parent);
reverse(path.begin(), path.end());
for (int i = 0; i < path.size(); ++i)
    cout << path[i] + 1 << ' ';

return 0;
}
```

Алгоритм Флойда-Уоршелла

- Будем хранить матрицу расстояний между всеми парами вершин d[N][N].
- Если в графе есть ребро из вершины i в вершину j длиной L, то d[i][j] = L. Расстояние от любой вершины до самой себя равно 0, т.е. d[i][j] = 0, если i = j. Для всех остальных пар i и j d[i][j] = INF.
- Алгоритм состоит из N фаз. Перед k-ой фазой в d[i][j] хранится минимальный путь, проходящий через вершины с номерами, меньшими k, или INF, если пути между данными вершинами не найдено.
- Во время k-ой фазы путь между вершинами i и j может либо сохраниться, либо пройти через вершину с номером k, если суммарный путь от вершины i в вершину k и из вершины k в вершину j меньше, чем d[i][j]. То есть на k-ой фазе d[i][j] = min(d[i][j], d[i][k] + d[k][j]).
- Алгоритм работает как с ориентированными так и с неориентированными графами. Алгоритм не работает, если в графе есть циклы с отрицательным весом.

Реализация

- Пусть дан взвешенный ориентированный граф из N вершин и M рёбер. Требуется найти кратчайшее расстояние между всеми парами вершин. Веса рёбер положительны.

```
int main()
{
    int n, m;
    cin >> n >> m;
    vector<vector<int>> d(n, vector<int>(n, INF));
    for (int i = 0; i < m; ++i)
    {
        int u, v, l;
        cin >> u >> v >> l;
        --u; --v;
        d[u][v] = l;
    }
    for (int i = 0; i < n; ++i)
        d[i][i] = 0;
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < n; ++j)
            if (d[i][j] != INF)
                cout << d[i][j] << ' ';
            else
                cout << -1 << ' ';
        cout << '\n';
    }
```

Восстановление пути в алгоритме Флойда-Уоршелла

- Заведём дополнительную матрицу p[N][N], заполненную -1. Когда расстояние между вершинами i и j обновляется благодаря проходу через вершину k запомним это (p[i][j] = k).
- Если d[i][j] = INF, то пути не существует, а значит восстанавливать нечего.
- Теперь мы знаем, что если путь существует p[i][j] = -1, то кратчайший путь между этими вершинами – непосредственный переход из i в j. Иначе следует восстановить пути из i в p[i][j] и из p[i][j] в j, объединение которых и будет восстановленным путём из i в j.
- Восстановление удобно реализуется рекурсивной функцией.

Реализация

```
vector<int> path;
void restore_path(int u, int v, vector<vector<int>>& p)
{
    if (p[u][v] == -1)
    {
        path.push_back(u);
        path.push_back(v);
        return;
    }
    restore_path(u, p[u][v], p);
    path.pop_back();
    restore_path(p[u][v], v, p);
}
```

Функция для восстановления

Изменения в алгоритме

```
vector<vector<int>> p(n, vector<int>(n, -1));
for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (d[i][j] > d[i][k] + d[k][j])
            {
                d[i][j] = d[i][k] + d[k][j];
                p[i][j] = k;
            }
int s, f;
cin >> s >> f;
--s; --f;
restore_path(s, f, p);
for(int i = 0; i < path.size(); ++i)
    cout << path[i] + 1 << ' ';
```