Московский авиационный институт
(национальный исследовательский университет)

Институт информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №9 по курсу «Дискретный анализ»

Студент:	Л. В. Ивенкова
Преподаватель:	Н. С. Капралов
Группа:	М8О-208Б-19
Дата:	
Оценка:	
Подпись:	

Москва, 2021

Лабораторная работа №9

Задача: Поиск кратчайшего пути между парой вершин алгоритмом Дейкстры

Вариант №4

Разработать программу на языке C или C++, реализующую указанный алгоритм согласно заданию:

Задан взвешенный неориентированный граф, состоящий из **n** вершин и **m** ребер. Вершины пронумерованы целыми числами **от 1 до n**. Необходимо найти длину кратчайшего пути из вершины с номером **start** в вершину с номером **finish** при помощи алгоритма Дейкстры. Длина пути равна сумме весов ребер на этом пути. Граф не содержит петель и кратных ребер.

Формат входных данных

В первой строке заданы $1 \leq n \leq 10^5, 1 \leq m \leq 10^5, 1 \leq start \leq n$ **и** $1 \leq finish \leq n$. В следующих **m** строках записаны ребра. Каждая строка содержит три числа – номера вершин, соединенных ребром, и вес данного ребра. Вес ребра – целое число **от 0 до** 10^9.

Формат результата

Необходимо вывести одно число – длину кратчайшего пути между указанными вершинами. Если пути между указанными вершинами не существует, следует вывести строку **"No solution"** (без кавычек).

1 Описание

Требуется написать реализацию алгоритма Дейкстры поиска кратчайшего пути между парой вершин в неориентированном взвешанном графе.

Справка вики[1]: **Алгоритм Дейкстры** — находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

Неформальное объяснение:

Каждой вершине из V сопоставим метку — минимальное известное расстояние от этой вершины до a.

Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки.

Работа алгоритма завершается, когда все вершины посещены.

Инициализация.

- Метка самой вершины a полагается равной 0, метки остальных вершин — бесконечности.
- Это отражает то, что расстояния от a до других вершин пока неизвестны.
- Все вершины графа помечаются как непосещённые.

Шаг алгоритма.

- Если все вершины посещены, алгоритм завершается.
- В противном случае, из ещё не посещённых вершин выбирается вершина u, имеющая минимальную метку.
- Мы рассматриваем всевозможные маршруты, в которых u является предпоследним пунктом. Вершины, в которые ведут рёбра из u, назовём *соседями* этой вершины. Для каждого соседа вершины u, кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки u и длины ребра, соединяющего u с этим соседом.
- Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину u как посещённую и повторим шаг алгоритма.

2 Исходный код

В начале создадим маленькую стрктурку ребра - она будет состоять из номера вершины, в котрое оно ведёт от текущей, и из стоимости ребра. Далее создаём граф - вектор векторов рёбер.

Для учёта незанятых вершин используем сет, так как в нём удобно находить минимальный элемент - по указателю на первый элемент.

Сам код:

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;

const long long INF = 1e13 + 7;

struct TEdge {
    int to, cost;
};

int main() {

    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    int n, m, s, f;
    cin >> n >> m >> s >> f;
    s--;
    f--;

    int a, b, c;
    vector<vector<TEdge>> G(n);
    for (int i = 0; i < m; ++i) {
        cin >> a >> b >> c;
        a--;
        b--;
        G[a].push_back({b, c});
        G[b].push_back({a, c});
    }

    long long dis = 0;
    vector<long long> d(n, INF);
    d[s] = 0;

```

```
    set<pair<long long, int>> st; // 1 -- dist, 2 -- num
    st.insert({0, s});

    while (!st.empty()) {

        pair<long long,int> p = *st.begin();
        long long dist = p.first;
        int current = p.second;
        if (current == f) {
            dis = dist;
            break;
        }
        st.erase(st.begin());

        for (TEdge to_cost: G[current]) {
            int to = to_cost.to;
            long long cost = to_cost.cost;
            if (d[to] > d[current] + cost) {
                st.erase({d[to], to});
                d[to] = d[current] + cost;
                st.insert({d[to], to});
            }
        }
    }

    if (!st.empty()) {
        cout << dis << "\n";
    } else {
        cout << "No solution" << "\n";
    }

    return 0;
}
```

3 Консоль

```
parsifal@DESKTOP-3G70RV4:~/DA/Lab9$ make
g++ -std=c++17 -O3 -Wextra -Wall -pedantic main.cpp -o Lab9
parsifal@DESKTOP-3G70RV4:~/DA/Lab9$ cat test0.txt
5 6 1 5
1 2 2
1 3 0
3 2 10
4 2 1
3 4 4
4 5 5
parsifal@DESKTOP-3G70RV4:~/DA/Lab9$ ./Lab9 <test0.txt
8
parsifal@DESKTOP-3G70RV4:~/DA/Lab9$ cat test.txt
5 5 1 5
1 2 2
1 3 0
3 2 10
4 2 1
3 4 4
parsifal@DESKTOP-3G70RV4:~/DA/Lab9$ ./Lab9 <test.txt
No solution
```

4 Тест производительности

Алгоритм можно было бы реализовать и без использования сета: в начале мы бы создали булевский массив размера n для учёта посещённых/ не посещённых вершин. Тогда бы мы в цикле проходились по всем n вершинам и для каждой из них пробегались бы n раз по массиву использованных вершин. Плюс m раз пробежались бы по рёбрам (суммарно за цикл). В итоге сложность была бы $O(n^2 + m)$.

В случае же с сетом получаем: удаление из сета мы выполняем за логарифм, и в целом мы будем это выполнять n раз. Далее опять m раз пробегаемся по соседям, когда ты мы будем делать удаление/вставку, когда то нет, но в целом это опять же будет работать за логарифм. В итоге получим сложность $O(n * log(n) + m * log(n))$ $= O(n * log(n))$.

Создаим программу для генерации тестов:

```
import sys
import random

print(10**5, 10**5, 1, 10**5)

a = random.randint(10**5,10**5)
b = random.randint(1,10**5)
c = random.randint(1,10**5)
print(a, b, c)

for x in range(10**5 - 1):
    a = random.randint(1,10**5)
    b = random.randint(1,10**5)
    c = random.randint(1,10**5)
    print(a, b, c)
```

Посмотрим время работы для теста с $n, m = 10^5$:

```
parsifal@DESKTOP-3G70RV4:~/DA/Lab9$ time ./Square <test.txt
905090

real    2m46.507s
user    2m46.156s
sys     0m0.031s
```

```
parsifal@DESKTOP-3G70RV4:~/DA/Lab9$ time ./Log <test.txt
905090

real    0m0.097s
user    0m0.094s
sys     0m0.000s
```

5 Выводы

Выполнив девятую лабораторную работу по курсу «Дискретный анализ», я изучила различные алгоритмы поиска минимального пути в графах, разобралась в их плюсах и минусах, и чем одни лучше других. Также я научилась оценивать сложность работы этих алгоритмов и их эффективность.

Список литературы

[1] *Алгоритм Дейкстры — Википедия.*
URL: `https://ru.wikipedia.org/wiki/Алгоритм_Дейкстры` (дата обращения: 21.06.2021).

[2] *Базовые алгоритмы нахождения кратчайших путей во взвешенных графах — Habr.*
URL: `https://habr.com/ru/post/119158/` (дата обращения: 21.06.2021).