# Московский авиационный институт (национальный исследовательский университет)

## Факультет информационных технологий и прикладной математики

## Кафедра вычислительной математики и программирования


**Лабораторная работа №9 по курсу «Дискретный анализ»**


Студент: К. С. Саженов
Преподаватель: И. Н. Симахин
Группа: М8О-308Б-19

Дата:
Оценка:
Подпись:

**Москва, 2021**

# Лабораторная работа №9

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

**Вариант 6:** Поиск кратчайших путей между всеми парами вершин алгоритмом Джонсона.

Задан взвешенный ориентированный граф, состоящий из n вершин и m ребер. Вершины пронумерованы целыми числами от 1 до $n$. Необходимо найти длины кратчайших путей между всеми парами вершин при помощи алгоритма Джонсона. Длина пути равна сумме весов ребер на этом пути. Обратите внимание, что в данном варианте веса ребер могут быть отрицательными, поскольку алгоритм умеет с ними работать. Граф не содержит петель и кратных ребер.

Входные данные: В первой строке заданы $1 \leq n \leq 2000$, $1 \leq m \leq 4000$. В следующих m строках записаны ребра. Каждая строка содержит три числа – номера вершин, соединенных ребром, и вес данного ребра. Вес ребра – целое число от $-10^9$ до $10^9$.

# 1 Описание

Алгоритм Джонсона [2] позволяет найти кратчайшие пути в ориентированном взвешенном графе без отрицательных циклов(контуров).

Он заключается в нахождении такой функции $\phi : V \to \mathbb{R}$,($V$ – все вершины графа) которая изменяла бы веса ребер в графе таким образом, чтобы длина кратчайших путей сохранялась бы(с поправкой на константу, которую мы опишем чуть ниже).

Итоговая функция весов будет представлена таким образом: $\omega_\phi(u, v) = \omega(u, v) + \phi(u) - \phi(v)$, где $\omega(u, v)$ – функция, возвращающая вес ребра от $u$ до $v$, а $\omega_\phi(u, v)$ – та же функция, но после приминения операции изменения веса.

Находить $\phi(v)$ предлагается с помощью одной дополнительной вершины $s$, от которой длина рёбер до всех остальных вершин изначально будет равна нулю(то есть $\omega(s, v) = 0$ для любых $v$) и алгоритма Форда-Беллмана, запущенного от этой вершины $s$ [3].

После исполнения алгоритма Форда-Беллмана(который, в т.ч., находит отрицательные циклы/контуры), кратчайшее расстояние от $s$ до всех остальных вершин принято брать за $\phi(u)$ для каждой вершины $u$, исключая вершину $s$.

После этого предлагается изменить функцию весов в графе по предложенной выше формуле и утверждается, что теперь все пути(равно как и ребра) в графе строго неотрицательные.

Из этого следует, что мы можем применить простой алгоритм Дейкстры [1] для поиска кратчайших путей в графе, поскольку у нас теперь нет рёбер с отрицательными весами.

После всех манипуляций алгоритмом Дейкстры, следует восстановить изначальные длины ребер по формуле $\omega(u, v) = \omega_\phi(u, v) - \phi(u) + \phi(v)$.

# 2 Исходный код

В данной лабораторной работе я решил разделить файлы для увеличения читаемости кода:

Файл `main.cpp`:

```cpp
#include "johnson.h"

#include <iostream>

int main() {
    std::size_t n, m;
    std::cin >> n >> m;
    std::vector<TEdge> edges(m);

    for(auto& edge: edges){
        std::cin >> edge.src;edge.src--;
        std::cin >> edge.dest;edge.dest--;
        std::cin >> edge.weight;
    }

    TGraph graph(edges, n);

    TGraph::AdjMatrix ans_matrix;

    try {
        ans_matrix = JohnsonAlgorithm(graph);
    }catch (TNegativeCycle& err){
        std::cout << err.what() << std::endl;
        return 0;
    }

    for(const auto& row: ans_matrix){
        for(const auto& col: row){
            if(col == WEIGHT_MAX){
                std::cout << "inf";
            } else{
                std::cout << col;
            }
            std::cout << ' ';
        }
        std::cout << std::endl;
    }

    return 0;
}
```

Файл `johnson.h`:

```cpp
//
```

```
2  // Created by sakost on 25.04.2021.
3  //
4
5  #ifndef LAB9_JOHNSON_H
6  #define LAB9_JOHNSON_H
7
8  #include "graph.h"
9
10 #include <vector>
11 #include <exception>
12 #include <limits>
13 #include <string>
14
15 class TNegativeCycle: public std::exception
16 {
17 public:
18     explicit TNegativeCycle() : msg_("Negative cycle"){}
19
20     ~TNegativeCycle() noexcept override = default;
21
22     const char* what() const noexcept override {
23         return msg_.c_str();
24     }
25
26 protected:
27     std::string msg_;
28 };
29
30 TGraph::AdjMatrix JohnsonAlgorithm(const TGraph& graph) noexcept(false);
31 TGraph::AdjMatrix BellmanFord(const TGraph& graph, std::size_t src) noexcept(false);
32 void
33 Dijkstra(const TGraph::AdjMatrix &graph, const std::size_t &start_vertex, TGraph::
      AdjMatrix &dist) noexcept;
34
35 #endif //LAB9_JOHNSON_H
```

Файл graph.h:

```
1  //
2  // Created by sakost on 20.06.2021.
3  //
4
5  #ifndef LAB9_GRAPH_H
6  #define LAB9_GRAPH_H
7
8  #include <cctype>
9  #include <vector>
10 #include <limits>
11
12
```

```
13 using TWeight = long long;
14 const TWeight WEIGHT_MAX = std::numeric_limits<TWeight>::max();
15
16 struct TEdge{
17     TWeight weight{};
18     std::size_t src{}, dest{};
19 };
20
21 class TGraph{
22 public:
23 // using AdjList = std::vector< std::vector< std::pair<std::size_t, TWeight> > >;
24     using AdjMatrix = std::vector<std::vector<TWeight>>;
25
26     AdjMatrix adjMatrix;
27
28     std::vector<TEdge> edges;
29     TGraph() = default;
30     TGraph(std::vector<TEdge> edges, std::size_t n);
31     std::size_t count_vertexes{}, count_edges{};
32 };
33
34
35 #endif //LAB9_GRAPH_H
```

# 3 Консоль

```
~/university/2 course/diskran/lab9/cmake-build-debug on master
$ cmake ../
--The C compiler identification is AppleClang 13.0.0.13000029
--The CXX compiler identification is AppleClang 13.0.0.13000029
--Detecting C compiler ABI info
--Detecting C compiler ABI info -done
--Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/
-skipped
--Detecting C compile features
--Detecting C compile features -done
--Detecting CXX compiler ABI info
--Detecting CXX compiler ABI info -done
--Check for working CXX compiler: /Applications/Xcode.app/Contents/Developer/Toolchain
-skipped
--Detecting CXX compile features
--Detecting CXX compile features -done
--Configuring done
--Generating done
--Build files have been written to: /Users/k.sazhenov/university/2 course/diskran/lab9

~/university/2 course/diskran/lab9/cmake-build-debug on master
$ cmake --build . --clean-first
[ 25%] Building CXX object CMakeFiles/lab9.dir/main.cpp.o
[ 50%] Building CXX object CMakeFiles/lab9.dir/johnson.cpp.o
[ 75%] Building CXX object CMakeFiles/lab9.dir/graph.cpp.o
[100%] Linking CXX executable lab9
[100%] Built target lab9

~/university/2 course/diskran/lab9/cmake-build-debug on master
$ cat ../input.txt
5 4
1 2 -1
2 3 2
1 4 -5
3 1 1
~/university/2 course/diskran/lab9/cmake-build-debug on master
$ ./lab9 < ../input.txt
0 -1 1 -5 inf
```

```
3 0 2 -2 inf
1 0 0 -4 inf
inf inf inf 0 inf
inf inf inf inf 0

~/university/2 course/diskran/lab9/cmake-build-debug on master
$
```

# 4 Тест производительности

Тест производительности представляет собой сравнение с алгоритмом Флойда-Уоршелла [4] , который можно считать наивным, поскольку время его работы $O(n^3)$, где $n$ – это количество вершин.

Тест производится на рандомно генерируемом графе без отрицательных циклов.

```
~/university/2 course/diskran/lab9/cmake-build-debug on master
$ cat ../input1.txt | wc -l
250

~/university/2 course/diskran/lab9/cmake-build-debug on master
$ ./lab9_benchmark <  ../input1.txt
Johnson algorithm: 0.791109
Naive algorithm: 5.46118

~/university/2 course/diskran/lab9/cmake-build-debug on master
$

~/university/2 course/diskran/lab9/cmake-build-debug on master
$ cat ../input2.txt | wc -l
10001

~/university/2 course/diskran/lab9/cmake-build-debug on master
$ ./lab9_benchmark <  ../input2.txt
Johnson algorithm: 20.3654
Naive algorithm: 27.8858

~/university/2 course/diskran/lab9/cmake-build-debug on master
$
```

Как видно из тестов,алгоритм Джонсона работает несколько быстрее алгоритма Флойда(он и асимптотически является более быстрым алгоритмом: $O(n^2 log(n) + nm)$ против $O(n^3)$,где $n$ --кол-во вершин,а $m$-- кол-во рёбер),хоть на таких числах и не происходит большого прироста.

# 5 Выводы

Задача об нахождении кратчайших путей из всех вершин до всех других – является довольно интересной с теоретической точки зрения. Алгоритм решения данной задачи, как мне кажется, может быть использован в поиске(как Google/Yandex) при решении задачи индексации страницы и определения коэффициента "важности"страницы при данном запросе. Также, как мне кажется, похожая задача может быть при проектировании различных видов сетей – будь то компьютерных, транспортных или водоканал.

Примечателен и тот факт, что данный алгоритм с практической точки зрения не слишком сложен, поскольку основные моменты лежат на других алгоритмах – на которых алгоритм Джонсона построен. Идея алгортма Джонсона достаточно проста и интуитивно понятна. Тем не менее мне было довольно трудно совместить два алгоритма, на которых основан алгоритм Джонсона, поскольку они используют немного разные представления графа в своей стандартной реализации. В общем и целом алгоритм Джонсона является одним из большого класса алгоритмов поиска на графах, что не делает его менее значимым, чем, например, алгоритм Дейкстры, поскольку данный алгоритм решает свою задачу довольно быстро в отличие, например, от алгоритма Флойда Уоршелла, с которым и было сравнение в секции теста производительности.

# Список литературы


[1] Т. Кормен, Ч Лейзерсон, Р Ривест, К Штайн. *Алгоритмы: построение и анализ, 3-е издание.* — Издательский дом «Вильямс», 2013. Перевод с английского: ООО «И.Д. Вильямс» — 1328 с. (ISBN 978-5-8459-1794-2 (рус.))

[2] *Алгоритм Джонсона – Викиконспекты*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Джонсона` (дата обращения: 16.10.2021).

[3] *Алгоритм Форда-Беллмана – Викиконспекты*
URL: `https://neerc.ifmo.ru/wiki/index.php?title= Алгоритм_Форда-Беллмана` (дата обращения: 16.10.2021)

[4] *Алгоритм Флойда – Викиконспекты*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Флойда` (дата обращения: 16.10.2021)