ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт машиностроения материалов и транспорта
Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

Алгоритм Флойда – Уоршалла

по дисциплине «Объектно-ориентированное программирование»

Выполнил: студент гр. 3331506/90401 Елкин М. А.
Преподаватель: Ананьевский М. С.

Санкт-Петербург
2022 год

Содержание

1. Введение 3

2. Описание 3

2.1 Алгоритм Дейкстры 3

2.2 Алгоритм Флойда Уоршалла 6

2.3 Особенности поведения 8

3. Анализ работы 12

4. Заключение 14

5. Список использованных источников 15

Приложение 1. Программный код алгоритма 16

1. Введение

Задача о кратчайшем пути — задача поиска самого короткого пути между двумя точками (узлами) на графе.

Графом будем называть несколько точек (узлов), некоторые пары которых соединены отрезками (рёбрами). Граф связный, если от каждого узла можно дойти до любого другого по этим отрезкам. Причем, не может быть двух рёбер, соединяющих одни и те же узлы. Граф называется взвешенным, если каждому ребру соответствует какое-то число (вес). Циклом назовём какой-то путь по рёбрам графа, начинающийся и заканчивающийся в одном и том же узле. Кратчайший путь из одного узла в другой — это такой путь по рёбрам, что сумма весов рёбер, по которым мы прошли, будет минимальна.

Алгоритмы нахождения кратчайшего пути в графах могут применяться, например, для нахождения путей между физическими объектами в картографических сервисах, в гейм-девелопменте и т.п.

Существует множество алгоритмов по поиску кратчайших путей, в данной работе будет разобран алгоритм Флойда-Уоршалла, использующий метод динамического программирования. Для сравнения будет использован алгоритм Дейкстры, использующий иной метод – жадный.

2. Описание

2.1 Алгоритм Дейкстры

Алгоритм Дейкстры решает задачу поиска кратчайших путей из одного узла во взвешенном ориентированном графе в случае, когда веса ребер неотрицательны. Если дан такой граф, тогда нужно найти кратчайший путь от некоторого начального узла до всех узлов графа напрямую или через другие узлы. Данный алгоритм для решения задачи использует жадный метод, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Рассмотрим суть алгоритма Дейкстры на примере графа, представленном на рисунке 1.

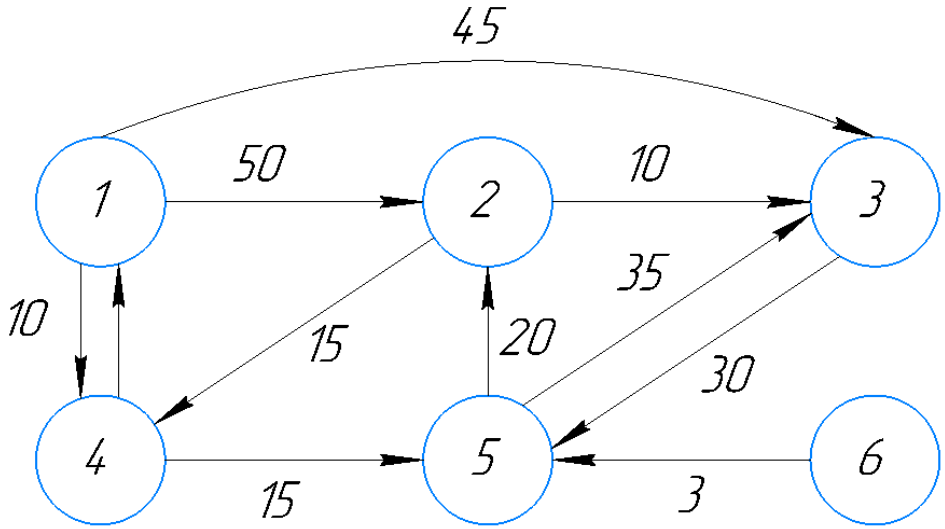

Рисунок 1 – Схема графа

Допустим, что узел 1 – начальный узел. Будем записывать в таблицу 1 расстояния от начального узла до остальных узлов. В первой строчке запишем расстояния до узлов, которые могут быть достигнуты напрямую. Если прямого пути не существует, то расстояние равняется бесконечности. Таким образом получаем таблицу 1.1.

Таблица 1.1 – Начальные расстояния

Расстояния до узлов / Выбранный узел	1	2	3	4	5	6
	0	50	45	10	∞	∞

Следующим шагом выбираем узел, расположенный ближе всего к начальному узлу. В нашем случае это узел 4 расположенный на расстоянии 10, пометим его красным цветом (расстояние до него уже изменяться не будет, не трогаем).

Далее проверяем является ли путь от 1-го узла до остальных узлов через 4-й более коротким (*d[u] + c(u,v)*), чем напрямую (*d[v]*). Если да, то меняем значение (relaxation), если нет, то оставляем как есть:

```
if (d[u] + c(u,v) < d[v]) {
       d[v] = d[u] + c(u,v);
}
```

Получаем таблицу 1.2.

Таблица 1.2 – Первая relaxation

Расстояния до узлов / Выбранный узел	1	2	3	4	5	6
4	0	50	45	10	∞	∞
	0	50	45	10	25	∞

Так как 4-й узел уже был выбран (выделим цветом, чтоб не забыть), выбираем из оставшихся узел, расстояние до которого наименьшее, т.е. 5-й. Выполняем проверку на более короткий путь, в этот раз уже через 5-й узел. Получаем таблицу 1.3.

Таблица 1.3 – Вторая relaxation

Расстояния до узлов / Выбранный узел	1	2	3	4	5	6
4	0	50	45	10	∞	∞
5	0	50	45	10	25	∞
	0	45	45	10	25	∞

Продолжаем выполнение алгоритма до тех пор, пока не будут выбраны все узлы. Тогда получается таблица 1.4.

Таблица 1.4 – Итоговая таблица

Расстояния до узлов / Выбранный узел	1	2	3	4	5	6
4	0	50	45	10	∞	∞
5	0	50	45	10	25	∞
2	0	45	45	10	25	∞
3	0	45	45	10	25	∞
6	0	45	45	10	25	∞

Итого, в нижней строчке имеем кратчайшие пути от начального узла до всех остальных.

2.2 Алгоритм Флойда Уоршалла

Алгоритм Флойда-Уоршалла решает задачу поиска кратчайших путей из одного узла во взвешенном ориентированном графе. Веса ребер могут быть как отрицательны, так и неотрицательны, но без отрицательных циклов. За одно выполнение алгоритма находятся длины (суммарные веса) кратчайших путей между всеми парами узлов. Хотя алгоритм не возвращает детали самих путей, их можно реконструировать с помощью простых модификаций алгоритма. Этот алгоритм для решения задачи использует метод динамического программирования. Рассмотрим алгоритм Флойда-Уоршалла на примере графа, изображенного на рисунке 2.

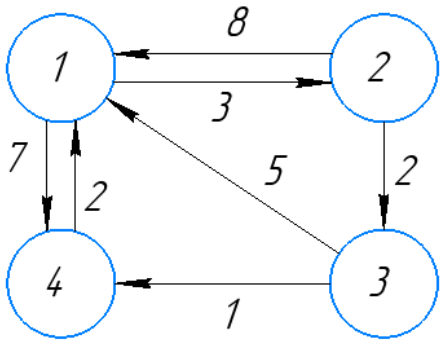

Рисунок 2 – Схема графа

Для решения задачи поиска кратчайших путей алгоритмом Флойда-Уоршалла необходимо составить матрицу расстояний. Изначально в ней содержатся только прямые пути между всеми парами узлов. Главная диагональ матрицы равна нулю. Для наглядности эта матрица представлена в таблице 2.1.

Таблица 2.1 – Исходная матрица расстояний

Узел назначения / Узел отсчета	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

$$A_0 = \begin{vmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{vmatrix}$$

Следующим шагом необходимо проверить, существует ли более короткий путь между всеми парами узлов через первый (промежуточный) узел. Результат записывается в матрицу $A_1 = dist$.

```
for (int i = 0; i < nodes; i++) {
    for (int j = 0; j < nodes; j++) {
        if (dist[i][j] > dist[i][0] + dist[0][j]) {
            dist[i][j] = dist[i][0] + dist[0][j];
        }
    }
}
```

$$A_1 = \begin{vmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{vmatrix}$$

Repeat:

$$A_2 = \begin{vmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{vmatrix}$$

Repeat:

$$A_3 = \begin{vmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{vmatrix}$$

Repeat:

$$A_4 = \begin{vmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{vmatrix}$$

В результате:

$$A_{\text{p}} = \begin{vmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{vmatrix}$$

Такой цикл повторяется число раз, равное числу узлов. С каждым разом уточняется минимальная дистанция между всеми парами узлов. Более того, при неотрицательных весах ребер значения в главной диагонали останутся равными нулю, а строка и столбец, соответствующие текущему промежуточному значению, не поменяют свои значения (сравнение числа с самим собой же, увеличенным на положительное число). В случае отрицательного веса ребра и неотрицательного цикла алгоритм также отрабатывает корректно.

Алгоритм Флойда-Уоршалла выдает правильный ответ до тех пор, пока в графе не появится отрицательный цикл.

2.3 Особенности поведения

Посмотрим на действия алгоритмов в случае возникновения отрицательных циклов, т.е. если существует такой цикл, сумма весов ребер которого отрицательна и попав в этот цикл можно получить бесконечно малое число.

Начнем с Дейкстры. Начальный узел – 1-й. Как видно на рисунке 3, ближайший узел – 2-й, поэтому выбираем его.

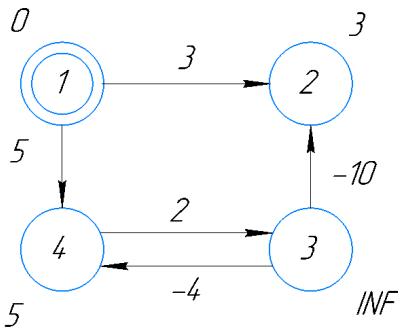

Рисунок 3 – Отрицательный цикл (алг. Дейкстры)

На рисунке 4 видно, что из узла 2 нет ни одного пути, поэтому ослаблять никакие узлы не требуется. Выбираем следующий ближайший узел – 4-й.

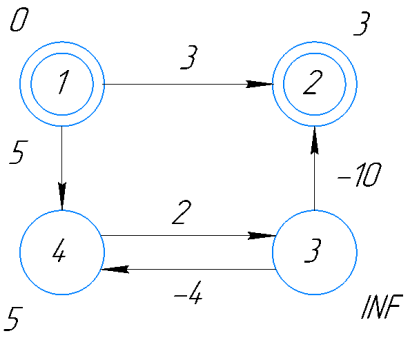

Рисунок 4 – Отрицательный цикл (алг. Дейкстры)

На рисунке 5 видно, что 3-й узел можно ослабить ($\infty > 5 + 2$). Это последний узел, выбираем его, выполнение алгоритма закончено

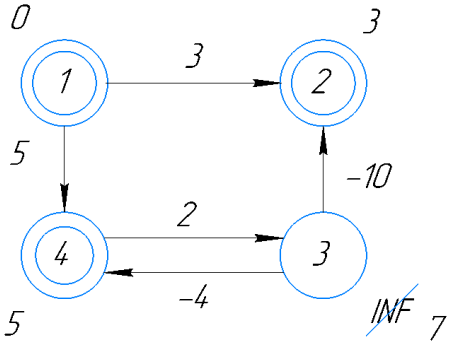

Рисунок 5 – Отрицательный цикл (алг. Дейкстры)

Однако, перепроверим еще раз, посмотрим, можно ли через 3-й узел ослабить какой-нибудь другой. И, как видно из рисунка 6, можно. Теперь путь до узла 2 составляет -3. В данном случае из узла 2 нет путей, ослабить другие не получится, т.е. это просто ребро с отрицательным весом.

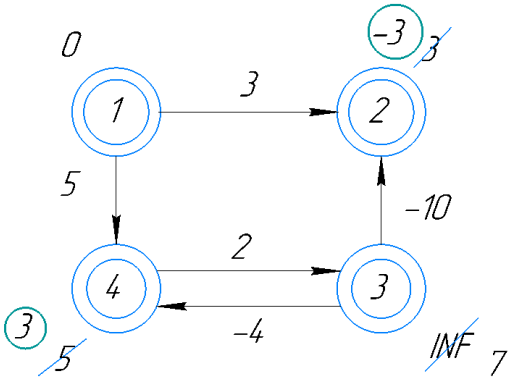

Рисунок 6 – Ошибка в алгоритме Дейкстры

Также можно ослабить путь до узла 4. А из него уже есть путь до 3, из которого в свою очередь можно прийти в 4-й узел. В данном случае имеем дело

с отрицательным циклом, который может повторяться неограниченное количество раз, приводя к некорректному ответу.

Хотя алгоритм Дейкстры не производит лишних вычислений, но выдает неправильный ответ. К тому же, после выполнения алгоритма мы не будем знать о наличии этой ошибки.

Теперь составим матрицу расстояний для алгоритма Флойда-Уоршелла.

$$A_0 = \begin{vmatrix} 0 & 3 & \infty & 5 \\ \infty & 0 & \infty & \infty \\ \infty & -10 & 0 & -4 \\ \infty & \infty & 2 & 0 \end{vmatrix}$$

По причине отсутствия путей из всех узлов до 1-го узла, матрица не меняется после первой итерации. После второй итерации также ничего не меняется.

$$A_1 = \begin{vmatrix} 0 & 3 & \infty & 5 \\ \infty & 0 & \infty & \infty \\ \infty & -10 & 0 & -4 \\ \infty & \infty & 2 & 0 \end{vmatrix}$$

$$A_2 = \begin{vmatrix} 0 & 3 & \infty & 5 \\ \infty & 0 & \infty & \infty \\ \infty & -10 & 0 & -4 \\ \infty & \infty & 2 & 0 \end{vmatrix}$$

$$A_3 = \begin{vmatrix} 0 & 3 & \infty & 5 \\ \infty & 0 & \infty & \infty \\ \infty & -10 & 0 & -4 \\ \infty & -8 & 2 & -2 \end{vmatrix}$$

На третьей итерации интересен проход по 4-й строке, так как в этом случае существует путь от 4 узла к 3-му, причем отрицательный. И при проверке условия dist[4][4] > dist[4][3] + dist[3][4], получаем $0 > 2 - 4$. И dist[4][4] = -2. То есть получаем ненулевой элемент на главной диагонали. Это сигнализирует о появлении отрицательного цикла. Дальнейшая работа алгоритма некорректна, однако ошибка выявляется простой проверкой главной диагонали на соответствие нулю в конце алгоритма. Однако, можно избежать выполнения алгоритма вхолостую (ошибка в начале, а алгоритм продолжает выполнение), добавив после каждого цикла в строке проверку на соответствие нулю элемента

dist[i][i], где i – только что пройденная строка. Или же сперва использовать алгоритм Беллмана-Форда для обнаружения отрицательных циклов за $O(mn)$, где n – количество узлов, m – количество ребер.

Также алгоритм Флойда-Уоршалла позволяет восстановить кратчайший путь между любыми узлами (если путь существует). Для этого перед выполнением алгоритма необходимо создать матрицу восстановления размера такого же, как и матрица расстояний. Далее, опираясь на матрицу расстояний, инициализировать ее по следующему принципу: если путь существует – в ячейку записывается конечный узел; если пути нет (∞) – в ячейку записывается 0.

```
vector<int> restore_matrix(grid.size());
    for (int i = 0; i < nodes; i++) {
        for (int j = 0; j < nodes; j++) {
            if (paths_matrix[i][j] == INF) {
                restore_matrix[i][j] = 0;
            }
            else
                restore_matrix[i][j] = j+1;
        }}
```

В ходе выполнения самого алгоритма, при изменении значения в матрице расстояний, в ту же (по индексу) ячейку матрицы восстановления записывается предыдущий узел.

```
if (paths_matrix[i][j] > min(paths_matrix, j, i, k)) {
  paths_matrix[i][j] = min(paths_matrix, j, i, k);
  restore_matrix[i][j] = restore_matrix[i][k];
}
```

Из полученной матрицы восстановления можно получить путь между любыми парами узлов. Схема работы такая: номер строки – начальный узел; номер столбца – конечный узел; значение в ячейке по этому индексу – следующий узел в восстанавливаемом пути (переходим в строчку с этим индексом); если в ячейке появляется конечный узел, значит путь найден. (Код в приложении 1, функция restore_path)

3. Анализ работы

Время работы алгоритма Дейкстры состоит из нахождения минимально удаленного узла и поиска минимальных путей через него. Для этого алгоритм повторяет цикл для каждой вершины, т.е. n раз. Внутри цикла каждый элемент вектора расстояний проверяется на то, был ли уже посещен узел. И если нет, то проверяется является ли расстояние до этого узла минимальным. Далее проверяется наличие более короткого пути в оставшиеся узлы. Получаем kn циклов ($k < n$) внутри первого. Итого время выполнения $O(n^2)$.

Если брать этот алгоритм для поиска кратчайших путей от всех узлов, как это делает алгоритм Флойда-Уоршалла, то время выполнения увеличится до $O(n^3)$.

Алгоритм Флойда-Уоршалла в свою очередь производит поиск кратчайших расстояний до n конечных узлов из n начальных узлов через n промежуточных узлов. Таким образом затраченное время $O(n^3)$.

```
for (int k = 0; k < nodes; k++) {
    for (int i = 0; i < nodes; i++) {
        for (int j = 0; j < nodes; j++){
            if (dist[i][j] > dist[i][k] + dist[k][j]) {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

Была получена одинаковая сложность $O(n^3)$ для обоих алгоритмов. Однако, они имеют разную реализацию, поэтому в реальности их время выполнения может отличаться. Чтобы это проверить было измерено время поиска кратчайших путей обоими способами в зависимости от количества узлов. Полученный результат представлен на рисунке 7.

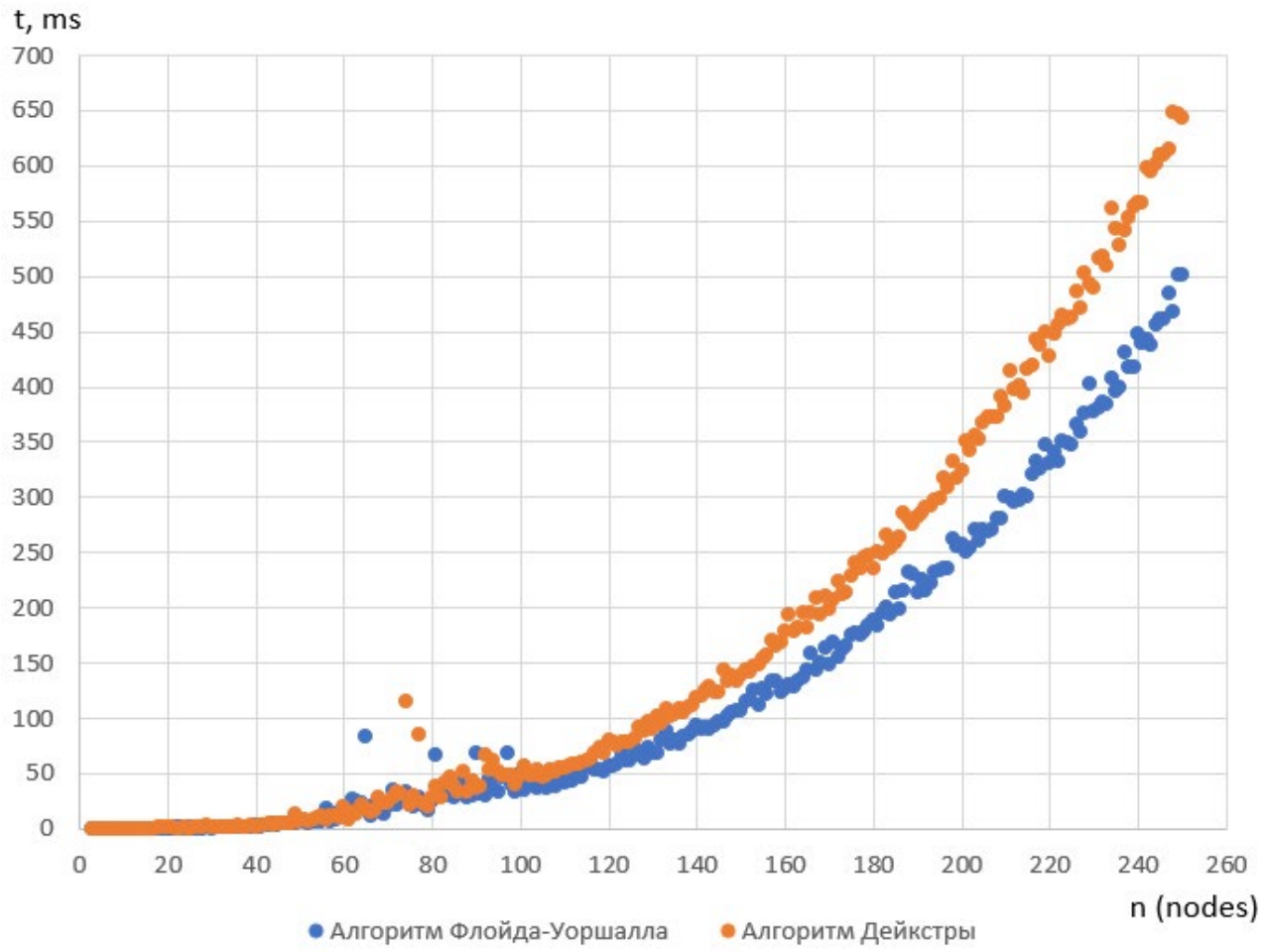

Рисунок 7 – Сравнение скорости работы алгоритмов

На вход подается случайная матрица прямых расстояний размера от 3 до 250 узлов. На выходе должна получится матрица кратчайших расстояний. Для более равных условий из алгоритма Флойда-Уоршалла была убрана функция сохранения пути в матрице восстановления, а также проверка на отрицательный цикл. В итоге был получен график зависимости времени выполнения алгоритмов от количества узлов в графе.

Как видно из графика, при количестве узлов более 100, начинается небольшое отставание алгоритма Дейкстры от алгоритма Флойда-Уоршалла по скорости выполнения. Это может быть объяснено выполнением большего количества операций и проверок. Недостаток может быть восполнен написанием более оптимального кода для данного алгоритма, например, как алгоритм Дейкстры для разреженных графов.

4. Заключение

В ходе выполнения работы были разобраны 2 популярных алгоритма для решения задачи о кратчайшем пути, а именно алгоритм Флойда-Уоршалла, использующий метод динамического программирования и алгоритм Дейкстры, использующий жадный метод. Для обоих алгоритмов был написан код реализации. Также было произведено сравнение скорости выполнения этих алгоритмов (реализованных в данной работе) для графов различных размеров. По итогам сравнения алгоритм Флойда-Уоршалла оказался быстрее. Более того, алгоритм Флойда-Уоршалла позволяет работать с ребрами с отрицательным весом, дает возможность отслеживать появление отрицательных циклов и является более простым в реализации в программном коде по сравнению с алгоритмом Дейкстры.

5. Список использованных источников

1) Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ — 3-е изд: Пер. с англ. — М.: Издательский дом «Вильямс», 2013. —1328 с.

2) Floyd–Warshall algorithm [Электронный ресурс] // URL:https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm#Example

3) Dijkstra's algorithm [Электронный ресурс] // URL: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

4) Abdul Bari. Algorithms [Электронный ресурс] // URL:https://www.youtube.com/playlist?list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O

5) Stefan Hougardy. «The Floyd-Warshall Algorithm on Graphs with Negative Cycles». Information Processing Letters. 110.

6) Базовые алгоритмы нахождения кратчайших путей во взвешенных графах [Электронный ресурс] // https://habr.com/ru/post/119158/

Приложение 1. Программный код алгоритма

```
#include <iostream>
#include <vector>
#include <cmath>
#include <tuple>
#include <random>

using namespace std;
#define INF numeric_limits<T>::max()

template<class T>
class Graph {
public:
    explicit Graph(vector<T> &);
    explicit Graph(vector<vector<T>> &);
    Graph<T> & operator= (const Graph<T> &);
    ~Graph();
public:
    ////Floyd-Warshall methods
    //Floyd-Warshall algorithm without restore matrix
    vector<vector<T>> floyd_warshall();
    //Floyd-Warshall algorithm with restore matrix
    tuple<vector<vector<T>>, vector<vector<int>>> floyd_warshall_ways();
    //Uses restore_matrix from FW_alg to restore paths
    vector<int> restore_path(int, int);
public:
    ////Dijkstra methods
    //receives start node and finds the shortest distances from it
    vector<T> dijkstra_from_one_node(int);
    //Applies Dijkstra algorithm for every node to make paths matrix
    vector<vector<T>> dijkstra();
public:
    ////auxiliary functions of class
    //getters
    vector<vector<T>> get_dist_matrix() { return dist_matrix; };
    vector<vector<T>> get_paths_matrix() { return paths_matrix; };
    vector<vector<int>> get_restore_matrix() { return restore_matrix; };
    vector<int> get_restored_path() { return restored_path; };
    vector<T> get_shortest_distances() { return shortest_distances; };
    //print functions
    void print_paths_matrix();
    void print_restored_path();

private:
    int nodes = 0;
    //initial matrix with direct distances between nodes
    vector<vector<T>> dist_matrix;
    //resulting matrix with the shortest paths between all nodes
    vector<vector<T>> paths_matrix;
    //matrix, used to restore paths with Floyd-Warshall algorithm modification
    vector<vector<int>> restore_matrix;
    //vector, used to store restored path
    vector<int> restored_path;
    //vector, used to store the shortest distances from one node in original
Dijkstra algorithm
    vector<T> shortest_distances;
    //compares path from origin node to destination with path through
intermediate node and returns the shortest
    T min(int, int, int);
    void clean();
};
```

```
template<typename T>
Graph<T>::Graph(vector<T> &grid) {
    //checking a size of input matrix
    if (grid.size() != pow((int) sqrt(grid.size()), 2)) {
        return;
    }
    //checking the main diagonal
    nodes = (int) sqrt(grid.size());
    for (int node = 0; node < nodes; node++) {
        if (grid[node * nodes + node] != 0) {
            nodes = 0;
            return;
        }
    }
    dist_matrix.assign(nodes, vector<T> (nodes));
    for (int row = 0; row < nodes; row++) {
        for (int col = 0; col < nodes; col++) {
            dist_matrix[row][col] = grid[row * nodes + col];
        }
    }
}

template<typename T>
Graph<T>::Graph(vector<vector<T>> &grid) {
    nodes = grid.size();
    //checking a size of input matrix
    for (int row = 0; row < nodes; row++) {
        if (grid[row].size() != nodes) {
            nodes = 0;
            return;
        }
    }
    //checking the main diagonal
    for (int node = 0; node < nodes; node++) {
        if (grid[node][node] != 0) {
            nodes = 0;
            return;
        }
    }
    dist_matrix = grid;
}

template<typename T>
Graph<T> & Graph<T>::operator= (const Graph<T> &other) {
    if (this == &other) {
        return *this;
    }
    this->nodes = other.nodes;
    this->paths_matrix = other.paths_matrix;
    this->shortest_distances = other.shortest_distances;
    this->restored_path = other.restored_path;
    this->restore_matrix = other.restore_matrix;
    this->dist_matrix = other.dist_matrix;
}

template<typename T>
Graph<T>::~Graph(){
    nodes = 0;
    clean();
}

template<typename T>
void Graph<T>::clean() {
```

```
    dist_matrix.clear();
    paths_matrix.clear();
    restore_matrix.clear();
    restored_path.clear();
    shortest_distances.clear();
}

////////////////////////////////////////////////
////Floyd-Warshall methods
////////////////////////////////////////////////

template<typename T>
vector<vector<T>> Graph<T>::floyd_warshall() {
    paths_matrix = dist_matrix;
    // Floyd-Warshall algorithm realization
    for (int intermediate = 0; intermediate < nodes; intermediate++) {
        for (int origin = 0; origin < nodes; origin++) {
            for (int destination = 0; destination < nodes; destination++) {
                //let paths_matrix = A, then
                //A(k)[x,y] = min(A(k-1)[x,y], A(k-1)[x,k] + A(k-1)[k,y])
                paths_matrix[origin][destination] = min(origin, destination,
intermediate);
            }
        }
    }
    return paths_matrix;
}

template<typename T>
tuple<vector<vector<T>>, vector<vector<int>>> Graph<T>::floyd_warshall_ways() {
    paths_matrix = dist_matrix;
    //initializing vector to restore the paths
    restore_matrix.assign(nodes, vector<int> (nodes));
    for (int row = 0; row < nodes; row++) {
        for (int col = 0; col < nodes; col++) {
            if (paths_matrix[row][col] == INF) {
                restore_matrix[row][col] = 0;
            } else
                restore_matrix[row][col] = col + 1;
        }
    }
    // Floyd-Warshall algorithm realization
    for (int intermediate = 0; intermediate < nodes; intermediate++) {
        for (int origin = 0; origin < nodes; origin++) {
            for (int destination = 0; destination < nodes; destination++) {
                //let paths_matrix = A, then
                //A(k)[x,y] = min(A(k-1)[x,y], A(k-1)[x,k] + A(k-1)[k,y])
                if (paths_matrix[origin][destination] > min(origin, destination,
intermediate)) {
                    paths_matrix[origin][destination] = min(origin, destination,
intermediate);
                    //saving previous nodes to restore paths
                    restore_matrix[origin][destination] =
restore_matrix[origin][intermediate];
                }
            }
        }
    }
    //Check for negative cycle
    for (int node = 0; node < nodes; node++) {
        if (paths_matrix[node][node] != 0) {
            paths_matrix.clear();
            restore_matrix.clear();
```

```
                cout << "negative cycle" << endl;
            }
        }
        return make_tuple(paths_matrix, restore_matrix);
    }

    template<typename T>
    vector<int> Graph<T>::restore_path(int from, int to) {
        restored_path.clear();
        if ((nodes < from) || (nodes < to)) return restored_path;
        if (restore_matrix.empty()) return restored_path;
        int current = from - 1;
        int destination = to - 1;
        if (paths_matrix[current][destination] == INF) return restored_path;
        while (current != destination) {
            restored_path.push_back(current);
            current = restore_matrix[current][destination] - 1;
            if (current < 0) {
                restored_path.clear();
                return restored_path;
            }
        }
        restored_path.push_back(current);
        return restored_path;
    }

    template<typename T>
    T Graph<T>::min(int origin, int destination, int intermediate) {
        T actual = paths_matrix[origin][destination];
        T alternative = 0;
        if (paths_matrix[origin][intermediate] == INF ||
    paths_matrix[intermediate][destination] == INF) alternative = INF;
        else alternative = paths_matrix[origin][intermediate] +
    paths_matrix[intermediate][destination];
        if (actual > alternative) return alternative;
        else return actual;
    }

    ////////////////////////////////////////////////
    ////Dijkstra methods
    ////////////////////////////////////////////////

    template<typename T>
    vector<T> Graph<T>::dijkstra_from_one_node(int origin) {
        origin--;
        shortest_distances.clear();
        // Checking correct input of the matrix and origin
        if (origin > nodes) return shortest_distances;
        //vector to store passed nodes
        vector<bool> passed(nodes);
        //vector to store the shortest distance from origin to all nodes
        shortest_distances.resize(nodes);
        //initializing vectors
        passed[origin] = true;
        for (int node = 0; node < nodes; node++) {
            shortest_distances[node] = dist_matrix[origin][node];
        }
        //Dijkstra algorithm realization
        for (int in_cln = 1; in_cln < nodes; in_cln++) {
            T min = INF;
            int next_node = -1;
            //Finding the nearest node
            for (int node = 0; node < nodes; node++) {
```

```
            //We can't go through passed nodes
            if (!passed[node]) {
                if (min > shortest_distances[node]) {
                    min = shortest_distances[node];
                    next_node = node;
                }
            }
        }
        if (next_node == -1) return shortest_distances;
        passed[next_node] = true;
        //Finding the shortest paths through this node
        for (int node = 0; node < nodes; node++) {
            // We can't go through passed nodes
            if (!passed[node]) {
                if (dist_matrix[next_node][node] != INF) {
                    if (min + dist_matrix[next_node][node] <
shortest_distances[node]) {
                        shortest_distances[node] = min +
dist_matrix[next_node][node];
                    }
                }
            }
        }
    }
    return shortest_distances;
}

template<typename T>
vector<vector<T>> Graph<T>::dijkstra() {
    paths_matrix.assign(nodes, vector<T> (nodes));
    for (int i = 1; i <= nodes; i++) {
        dijkstra_from_one_node(i);
        paths_matrix[i-1] = shortest_distances;
    }
    return paths_matrix;
}

////////////////////////////////////////////////
////auxiliary functions of class
////////////////////////////////////////////////

template<typename T>
void Graph<T>::print_paths_matrix() {
    if (nodes < 1) return;
    cout << "0  || ";
    for (int i = 1; i <= nodes; i++) {
        cout << i << " | ";
    }
    for (int row = 0; row < nodes; row++) {
        cout << endl << row + 1 << " || ";
        for (int col = 0; col < nodes; col++) {
            if (paths_matrix[row][col] == INF) cout << "#";
            else cout << paths_matrix[row][col];
            cout << " | ";
        }
    }
    cout << endl;
}

template<typename T>
void Graph<T>::print_restored_path() {
    if (restored_path.empty()) {
        cout << "No way" << endl;
```

```
        return;
    }
    cout << "The way is";
    for (const auto &i: restored_path) {
        cout << " - " << i + 1;
    }
    cout << endl;
}

////////////////////////////////////////////////
////other functions
////////////////////////////////////////////////

vector<int> random_grid(int nodes, int lower, int upper) {
    uniform_int_distribution<int> distr(lower, upper);
    mt19937 generator;
    vector<int> grid(nodes * nodes);
    for (int i = 0; i < grid.size(); i++) {
        grid[i] = distr(generator);
    }
    for (int i = 0; i < nodes; i++) {
        grid[i * nodes + i] = 0;
    }
    return grid;
}

void example() {
    double inf = numeric_limits<double>::max();
    vector<double> grid = {0, 50.5, 45, 10.09, inf, inf,
                           inf, 0, 10.7, 15, inf, inf,
                           inf, inf, 0, inf, 30.4, inf,
                           10, inf, inf, 0, 15.96, inf,
                           inf, 20.12, 35.62, inf, 0, inf,
                           inf, inf, inf, inf, 3.1, 0};

    cout << "========================================" << endl;
    cout << "        FLoyd-Warshall algorithm" << endl;
    cout << "========================================" << endl;
    Graph map_floyd(grid);
    map_floyd.floyd_warshall_ways();
    cout << endl << "Paths matrix:" << endl;
    map_floyd.print_paths_matrix();
    map_floyd.restore_path(6, 1);
    cout << endl << "From node 6 to 1 ";
    map_floyd.print_restored_path();
    cout << endl;

    cout << "========================================" << endl;
    cout << "           Dijkstra algorithm" << endl;
    cout << "========================================" << endl;
    Graph map_dijkstra(grid);
    map_dijkstra.dijkstra();
    cout << endl << "Paths matrix:" << endl;
    map_dijkstra.print_paths_matrix();
}

void time_compare() {
    cout << "==================" << endl;
    cout << "    Time Compare   " << endl;
    cout << "==================" << endl;
    cout << "it   F-W      Dijk " << endl;
    for (int i = 3; i <= 250; i++) {
        vector<int> grid = random_grid(i, 0, 1000);
```

```
        unsigned int start = 0;
        unsigned int end = 0;
        Graph map_floyd(grid);
        Graph map_dijkstra(grid);

        //measuring time for Floyd-Warshall algorithm
        start = clock();
        map_floyd.floyd_warshall();
        end = clock();
        unsigned int FW_time = end - start;

        //measuring time for Dijkstra algorithm
        start = clock();
        map_dijkstra.dijkstra();
        end = clock();
        unsigned int Dij_time = end - start;
        cout << i << "     " << FW_time << "         " << Dij_time << endl;
    }
}

int main() {

    return 0;
}
```