Санкт-Петербугский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: объектно-ориентированное программирование

Тема: алгоритмы Беллмана-Форда и Флойда-Уоршалла

Студенты гр. 3331506/20102 Александров Г. А.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2025

Оглавление

Введение

Алгоритмы Беллмана-Форда и Флойда-Уоршалла позволяют определить длину кратчайшего пути в парах вершин взвешенного графа размером n вершин и m рёбер. Причем при работе обоих алгоритмов допускается использование рёбер отрицательного веса, но не циклов.

Под циклами отрицательного веса подразумевается маршрут из определенной вершины a_i до нее же (до a_i) через произвольное количество вершин, сумма рёбер в котором меньше нуля. Однако сейчас существуют модификации обоих алгоритмов, позволяющие не только определить наличие цикла отрицательного веса, но и определить вершины в него входящие [1, 2]. Рассмотрение этой или прочих модификаций, ускоряющих работу алгоритмов, не входит в рамки данной курсовой работы.

Алгоритмы Беллмана-Форда и Флойда-Уоршалла являются частными представителями группы алгоритмов нахождения кратчайшего маршрута в графе (наравне с волновым алгоритмом, алгоритмами поиска A*, Дейкстры и прочими). Задача определения кратчайшего пути актуальна и по сей день, не только в области оптимизации логистики, но и в любой общей ситуации, предполагающей наличие однородных целей и затрат их достижения. К примеру, модифицированный алгоритм Беллмана-Форда используется для формирования резервных путей телекоммуникационных сетей, повышающих устойчивость системы [3].

Таким образом, в данной курсовой работе были поставлены следующие задачи: изучить работу непосредственно самих алгоритмов и принципы их реализации, написать соответствующую функцию-исполнителя на языке $C{+}{+}$, а также проанализировать их быстродействие и затраты памяти.

Алгоритм Беллмана-Форда

В данном ориентированном взвешенном графе с n вершинами и m ребрами для заданной начальной вершины $start$ алгоритм находит длины кратчайших путей до всех вершин в графе.

Описание работы алгоритма

Сперва, на стадии подготовки инициализируется массив расстояний размером n, который после работы алгоритма и будет содержать ответ на поставленную задачу (поэтому при реализации его и было решено назвать ans). Далее на нулевой итерации он заполняется так, что $ans[start] = 0$, а все остальные элементы равны ∞, то есть считаются недостижимыми. На практике же используется константа MAX_COUNT, обозначающая эту бесконечность.

В основе принципа работы алгоритма Беллмана-Форда лежит метод *релаксации*, то есть на каждом шаге для произвольного ребра, характеризующего путь из вершины $source$ в $target$ стоимостью $cost$, происходит попытка улучшить (уменьшить) значение элемента $ans[target]$ значением $ans[source] + cost$.

Из описания хода работы следует, что при реализации алгоритма удобнее всего использовать описание графа со списком рёбер.

Полагается, что за $n - 1$ прохождений по всем m рёбрам алгоритм корректно определяет длины всех кратчайших путей. Доказательство чего приводится в списке литературы [5].

Таким образом, реализованный алгоритм на языке $C{+}{+}$ представлен на рисунке 1, полный же текст программы вынесен в приложение.

```cpp
Dynamic_array<unsigned int> Graph::Bellman_Ford(const unsigned int start) {

    // инициализация массива расстояний из вершины start
    Dynamic_array<unsigned int> ans = Dynamic_array<unsigned int>(vertexs.length());

    // он заполняется значениями, равными максимальной дистанции, то есть на нулевом этапе итераций все вершины считаются недостижимыми
    for (unsigned int idx = 0; idx < ans.length(); idx++){
        ans[idx] = MAX_COUNT;
    }

    // проверяется наличие данной вершины в графе
    // если её нет, то осуществляется выход из функции, иначе соответствующему значению массива присваивается 0
    if (!contains(start)) return ans;
    ans[start] = 0;

    // далее происходит заполнение массива с помощью списка ребер:
    // считается, что если путь до вершины i кратчайший, то путь до вершины j определится как сумма путей от start до i и от i до j
    // полагается, что за n итераций определятся все наикратчайшие пути до всех n вершин
    for (unsigned int idx = 0; idx < vertexs.length() - 1; idx++){
        for (unsigned int jdx = 0; jdx < ways.length(); jdx++){
            ans[ways[jdx].to] = std::min(ans[ways[jdx].to], ans[ways[jdx].from] + ways[jdx].cost);
        }
    }

    return ans;
}
```

Рисунок 1 – Реализация алгоритма Беллмана-Форда

Оценка сложности

Так как в процессе работы алгоритма создается и используется лишь один массив размером n, то и занимаемый объем памяти будет линейно ему пропорционален с коэффициентом $size\_t$, обозначаемым объем памяти выбранного типа данных для хранения длины пути.

Быстродействие замерялось эмпирически для получения численных данных (результаты представлены на рисунке 2). Теоретически же сложность алгоритма оценивается как $O(n \cdot m)$, что следует из описания принципов его работы.

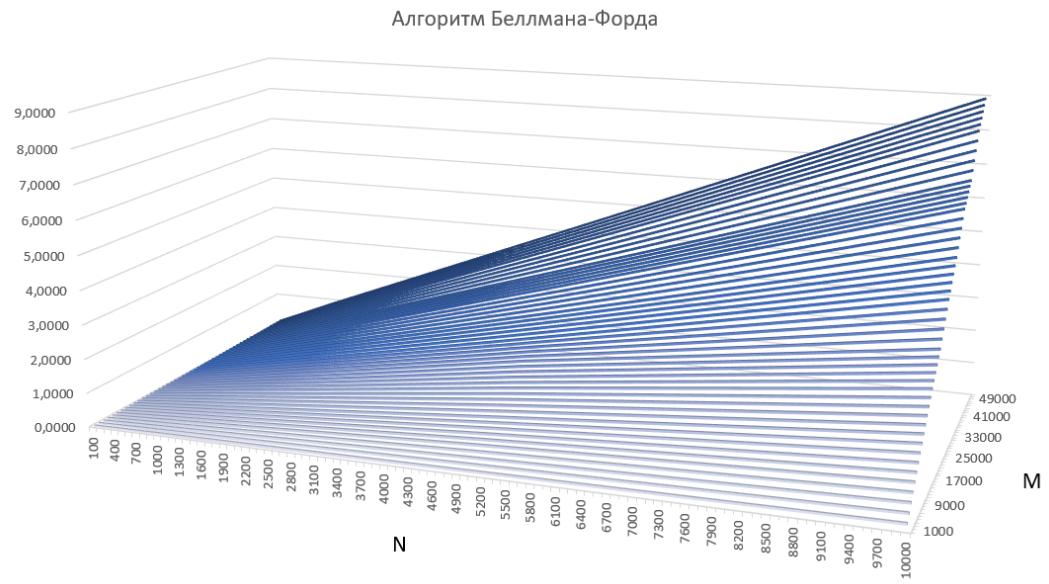

Рисунок 2 – Быстродействие алгоритма Беллмана-Форда

Так, экспериментально было получено, что алгоритм Беллмана-Форда обрабатывает граф с 10000 вершинами и 50000 рёбрами за среднее время около 9 секунд.

Преимущества и недостатки

Таким образом, алгоритм Беллмана-Форда, решающий задачу поиска кратчайшего пути из одной вершины до всех, можно назвать неуниверсальным, ведь для построения полной карты потребуется n запусков, а для получения кратчайшего пути только между выбранной парой вершин алгоритм хранит и обрабатывает много излишней информации.

Однако, в отличии от алгоритма Дейкстры, может работать с ребрами отрицательного веса. К тому же линейная зависимость сложности работы алгоритма от объемных характеристик n и m является достаточно оптимальной (с точки зрения затрат времени, опережая степенную зависимость, но уступая логарифмической), что наиболее очевидно проявляется при работе с графами со средней плотностью ребер (т.е. порядка n).

Алгоритм Флойда-Уоршалла

Пусть дан взвешенный граф с n вершинами. Алгоритм Флойда-Уоршалла позволяет в нем найти значения кратчайших путей между всеми возможными парами вершин.

Описание работы алгоритма

Из постановки задачи следует, что в работе алгоритма удобнее хранить результаты в виде матрицы / двумерного массива размером $n \times n$, на пересечении строки (обозначающей вершину $source$) и столбца (вершину $target$) которой и находится значение кратчайшего пути между ними.

На нулевой итерации результирующей матрице (при реализации обозначенной ans) присваивается матрица смежности данного графа. К слову, всем несуществующим ребрам между $source$ и $target$ присваивается условная (по аналогии с алгоритмом Беллмана-Форда) бесконечность ∞, то есть $ans[source][target] = MAX\_COUNT$.

Весь процесс нахождения значения минимального расстояния между всеми парами вершин разделяется на фазы, внутри которой происходит релаксация, аналогичная алгоритму Беллмана-Форда. Однако же на каждой k-ой фазе алгоритм пытается улучшить расстояние не вдоль ребра, а через вершину, то есть проверяется, что меньше: расстояние от $source$ до $target$ напрямую или через промежуточную для них вершину $middle$ (алгебраически это выглядит как сравнение $ans[source][target]$ и $ans[souce][middle] + ans[middle][target]$).

Таким образом, реализация простейшего алгоритма Флойда-Уоршалла на языке программирования $C++$ представлена на рисунке 3.

```
Dynamic_array<Dynamic_array<unsigned int>> Graph::Floyd_Warshall() {

    // алгоритм Флойда-Уоршелла определяет наикратчайшие пути от всех вершин до всех
    // поэтому на нулевой итерации результирующая матрица соответствует матрице смежности
    Dynamic_array<Dynamic_array<unsigned int>> ans = adjacency_matrix;

    // далее для каждой k-ой вершины проверяется следующее:
    // какой из путей от i до j (напрямую или через вершину k (от i до k + от k до j)) минимальным
    for (unsigned int kdx = 0; kdx < vertexs.length(); kdx++){
        for (unsigned int row = 0; row < vertexs.length(); row++){
            for (unsigned int col = 0; col < vertexs.length(); col++){
                ans[row][col] = std::min(ans[row][col], ans[row][kdx] + ans[kdx][col]);
            }
        }
    }

    return ans;
}
```

Рисунок 3 – Реализация алгоритма Флойда-Уоршалла

Оценка сложности

Таким образом, немудрено, что пространственная сложность соответствует хранимой матрице $O(n^2)$.

Быстродействие также было измерено эмпирически (результаты представлены на рисунке 4), однако из описания работы алгоритма следует, что временная сложность составляет $O(n^3)$ (n^2 на перебор всех пар + n на перебор промежуточных вершин).

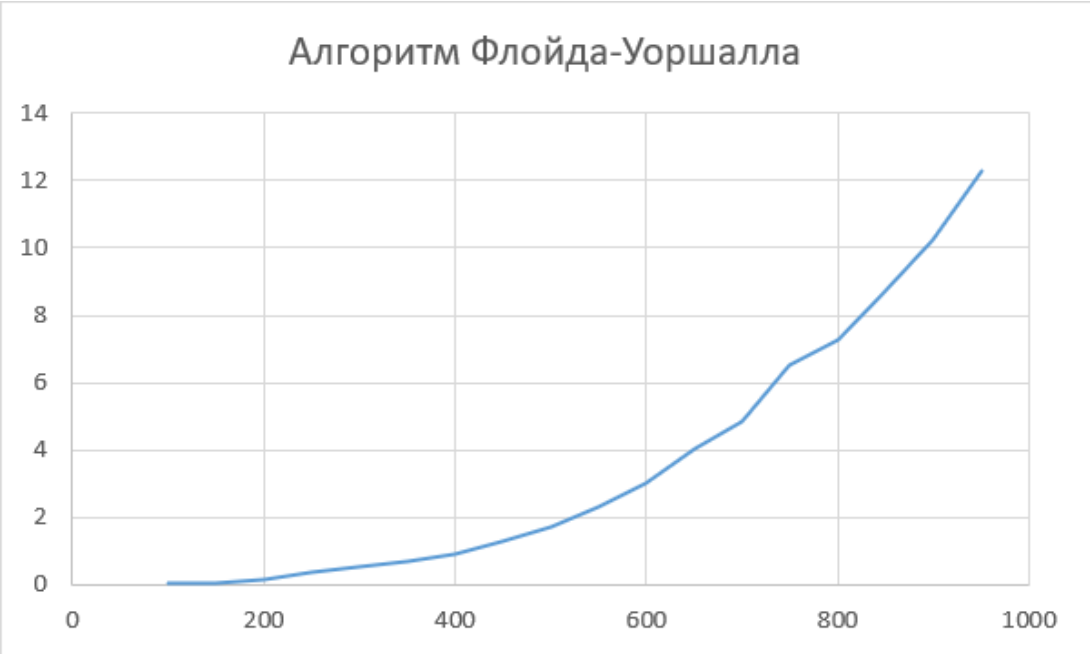

Рисунок 4 – Быстродействие алгоритма Флойда-Уоршалла

Эмпирически замерить время работы алгоритма удалось лишь для графов с количеством вершин не более 950, однако и существующие данные показывают резкий (степенной) рост затраченного времени от n.

Преимущества и недостатки

Алгоритм Флойда-Уоршалла хоть и не является единственным (алгоритм Данцига, к примеру), однако решает вполне специфическую задачу и является абсолютно избыточным при попытке его применения для решения частных случаев: о поиске кратчайшего расстояния из данной вершины до всех прочих, либо между конкретной парой вершин. Однако за счет своей простоты и работоспособности его, безусловно, можно считать лидером в данной нише.

Нельзя также не отметить степенную временную (да и пространственную) сложность, обусловленную соответствующей сложностью поставленной задачи.

Заключение

Алгоритмы Беллмана-Форда и Флойда-Уоршалла решают разные задачи определения значения кратчайшего расстояния во взвешенном графе: первый – только для пар вершин, исходящих из данной, а второй – для всех пар (то есть строит полную карту всех путей). За счет своей простоты работы и реализуемости они оба являются базовыми в программировании.

Однако же, при попытке получения полной матрицы значений всех кратчайших маршрутов алгоритмом Беллмана-Форда (запустив его n раз) пространственная сложность станет такой же, а вот временная – может оказаться значительно больше, так как в среднем ребер в графе в пару раз больше вершин. Но в целом, это показывает, что алгоритмом Беллмана-Форда может без проблем трансформироваться для решения задачи более общего случая (да, с измеримыми потерями по быстродействию), чего нельзя сказать об алгоритме Флойда-Уоршалла, оказывающимся иррациональным для решения более частных задач определения значения кратчайшего пути в графе.

Заметим также, что при наличии в графе циклов отрицательного веса, оба алгоритма (в своей простейшей реализации) сохраняют свою работоспособность, однако выдают ложный результат.

На сегодняшний день существуют различные модификации обоих алгоритмов, ускоряющие среднее время работы на случайных графах, либо же сигнализирующие о наличии циклов отрицательного веса или восстанавливающие пути, соответствующие данному кратчайшему маршруту. Всё это в очередной раз показывает базовость обоих алгоритмов.

Список литературы

1. Информационный портал по программированию и алгоритмизации MAXimal, статья посвященная алгоритму Беллмана-Форда. https://e-maxx.ru/algo/ford_bellman
2. Информационный портал по программированию и алгоритмизации MAXimal, статья посвященная алгоритму Флойда-Уоршалла. https://e-maxx.ru/algo/floyd_warshall_algorithm#3
3. Макаренко С. И., Квасов М. Н. Модифицированный алгоритм Беллмана-Форда с формированием кратчайших и резервных путей и его применение для повышения устойчивости телекоммуникационных систем //Инфокоммуникационные технологии. – 2016. – Т. 14. – №. 3. – С. 264-274.
4. Изотова Т. Ю. Обзор алгоритмов поиска кратчайшего пути в графе //Новые информационные технологии в автоматизированных системах. – 2016. – №. 19. – С. 341-344.
5. Охотин А. Математические основы алгоритмов, осень 2019 г. Лекция 4: Поиск в ориентированном графе: поиск в ширину, поиск в глубину, топологическая сортировка, нахождение компонентов сильной связности. Поиск в графе с весами: алгоритм Беллмана–Форда, алгоритм Дейкстры.
6. Образовательный иностранный портал *programiz*, посвященный программированию, статья «Floyd-Warshall Algorithm» https://www.programiz.com/dsa/floyd-warshall-algorithm

Приложение

```
#include <iostream>
#include <iomanip>
#include "algorithm"
#include "dynamic_array.h"
#include <random>


const int MAX_COUNT = 999999;


struct Way {
    unsigned int to;
    unsigned int from;
    unsigned int cost;
};


class Node {

public:
    unsigned int value;

    Dynamic_array<Node*> neighbors;
    Dynamic_array<int> weights;


public:

    Node();
    Node(const unsigned int x);
    Node(const Node& x);

    Node& operator=(const Node& x);
    friend bool operator== (const Node &x, const Node &y);

};


class Graph {

private:
    Dynamic_array<Node> vertexs;

    Dynamic_array<Way> ways;
    Dynamic_array<Dynamic_array<unsigned int>> adjacency_matrix;


public:
```

```
    void add_node(const unsigned int new_name);
    void add_path(const unsigned int from, const unsigned int to, const int cost, const bool
add_reverse = false);

    bool contains(const unsigned int node_name);

    void del_node(const unsigned int name_to_del);
    void del_path(const unsigned int from, const unsigned int to);

    void generate_ways();
    void generate_matrix();

    Dynamic_array<unsigned int> Bellman_Ford(const unsigned int start);
    Dynamic_array<Dynamic_array<unsigned int>> Floyd_Warshall();

    void output(const bool flag = false);

private:

    Node* get(const unsigned int node_name);

};


Node::Node() {
    value = 0;
    neighbors = Dynamic_array<Node*>();
    weights = Dynamic_array<int>();
}


Node::Node(const unsigned int x) {
    value = x;
    neighbors = Dynamic_array<Node*>();
    weights = Dynamic_array<int>();
}


Node::Node(const Node &x) {
    value = x.value;
    neighbors = x.neighbors;
    weights = x.weights;
}


Node& Node::operator=(const Node &x) {
    value = x.value;
    neighbors = x.neighbors;
    weights = x.weights;
}
```

```
bool operator==(const Node &x, const Node &y) {
    if (x.value == y.value) return true;
    return false;
}


void Graph::add_node(const unsigned int new_name) {
    vertexs.push_back(Node(new_name));
}


void Graph::add_path(const unsigned int from, const unsigned int to, const int cost, const bool
add_reverse) {
    if (!contains(from)) add_node(from);
    if (!contains(to)) add_node(to);

    Node* node_from = get(from);
    Node* node_to = get(to);

    unsigned int pos_of_to = node_from->neighbors.position(node_to);
    if (pos_of_to == 0) {
        node_from->neighbors.push_back(node_to);
        node_from->weights.push_back(cost);
    } else {
        node_from->weights[pos_of_to - 1] = std::min(node_from->weights[pos_of_to - 1], cost);
    }

    if (add_reverse){
        add_path(to, from, cost, false);
    }

}


void Graph::del_node(const unsigned int name_to_del) {
    Node * node_to_del = get(name_to_del);

    unsigned int pos_node_to_del_in_vertex = vertexs.position(*node_to_del);
    if (pos_node_to_del_in_vertex == 0) return;

    for (unsigned int idx = 0; idx < vertexs.length(); idx++){
        unsigned int pos_node_to_del_in_neighbours = vertexs[idx].neighbors.position(node_to_del);
        if (pos_node_to_del_in_neighbours == 0) continue;

        vertexs[idx].neighbors.del(pos_node_to_del_in_neighbours - 1);
        vertexs[idx].weights.del(pos_node_to_del_in_neighbours - 1);
    }

    vertexs.del(pos_node_to_del_in_vertex - 1);
}
```

```
void Graph::del_path(const unsigned int from, const unsigned int to) {
    unsigned int pos_node_from_in_vertex = vertexs.position(*get(from));
    if (pos_node_from_in_vertex == 0) return;

    unsigned int pos_node_to_in_froms_neighbours = vertexs[pos_node_from_in_vertex -
1].neighbors.position(get(to));
    if (pos_node_to_in_froms_neighbours == 0) return;

    vertexs[pos_node_from_in_vertex - 1].neighbors.del(pos_node_to_in_froms_neighbours - 1);
    vertexs[pos_node_from_in_vertex - 1].weights.del(pos_node_to_in_froms_neighbours - 1);

}


bool Graph::contains(const unsigned int node_name) {
    if (vertexs.length() == 0) return false;

    for (unsigned int idx = 0; idx < vertexs.length(); idx++){
        if (vertexs[idx].value == node_name) return true;
    }
    return false;
}


void Graph::output(const bool flag) {
    if (vertexs.length() == 0) {
        std::cout << "There is no elements in this graph\n\n";
        return;
    }

    for (unsigned int idx = 0; idx < vertexs.length(); idx++){
        std::cout << vertexs[idx].value << "    [";

        if (vertexs[idx].neighbors.length() != 0){
            for (unsigned int jdx = 0; jdx < vertexs[idx].neighbors.length(); jdx++){
                std::cout << " " << vertexs[idx].neighbors[jdx]->value << "{" <<
vertexs[idx].weights[jdx] << "} ";
            }
        }

        std::cout << "]\n";
    }
    std::cout << "\n";

    if (flag){
        for (unsigned int idx = 0; idx < ways.length(); idx++){
            std::cout << ways[idx].from << "\t" << ways[idx].to << "\t" << ways[idx].cost << "\n";
        }
        std::cout << "\n";

        for (unsigned int idx = 0; idx < adjacency_matrix.length(); idx++){
            for (unsigned int jdx = 0; jdx < adjacency_matrix[idx].length(); jdx++){
```

```
            if (adjacency_matrix[idx][jdx] == MAX_COUNT)
                std::cout << "*  ";
            else
                std::cout << adjacency_matrix[idx][jdx] << "  ";
        }
        std::cout << "\n";
    }
    std::cout << "\n";
  }
}


Node* Graph::get(const unsigned int node_name) {

  if (!contains(node_name)) return new Node();

  for (unsigned int idx = 0; idx < vertexs.length(); idx++){
    if (vertexs[idx].value == node_name) return &vertexs[idx];
  }

  return new Node();
}


void Graph::generate_ways() {

  for (unsigned int idx = 0; idx < vertexs.length(); idx++){
    for (unsigned int jdx = 0; jdx < vertexs[idx].neighbors.length(); jdx++){
      Way new_way;
      new_way.from = vertexs[idx].value;
      new_way.to = vertexs[idx].neighbors[jdx]->value;
      new_way.cost = vertexs[idx].weights[jdx];

      ways.push_back(new_way);
    }
  }
  std::cout << "Total N = " << vertexs.length() << "\n";
  std::cout << "Total M = " << ways.length() << "\n";
}


void Graph::generate_matrix() {
  // init of adjacency_matrix
  for (unsigned int idx = 0; idx < vertexs.length(); idx++){
    adjacency_matrix.push_back(*new Dynamic_array<unsigned int>(vertexs.length()));
  }


  for (unsigned int idx = 0; idx < vertexs.length(); idx++){
    for (unsigned int jdx = 0; jdx < vertexs.length(); jdx++){
      adjacency_matrix[idx][jdx] = MAX_COUNT;
    }
```

```
        for (unsigned int jdx = 0; jdx < vertexs[idx].neighbors.length(); jdx++){
            adjacency_matrix[idx][vertexs[idx].neighbors[jdx]->value] = vertexs[idx].weights[jdx];
        }
        adjacency_matrix[idx][idx] = 0;
    }

    std::cout << "Total N = " << vertexs.length() << "\n";
}


Dynamic_array<unsigned int> Graph::Bellman_Ford(const unsigned int start) {
    clock_t begin = clock();

    // инициализация массива расстояний из вершины start
    Dynamic_array<unsigned int> ans = Dynamic_array<unsigned int>(vertexs.length());

    // он заполняется значениями, равными максимальной дистанции, то есть на нулевом этапе итераций все вершины считаются недостижимыми
    for (unsigned int idx = 0; idx < ans.length(); idx++){
        ans[idx] = MAX_COUNT;
    }

    // проверяется наличие данной вершины в графе
    // если её нет, то осуществляется выход из функции, иначе соответствующему значению массива присваивается 0
    if (!contains(start)) return ans;
    ans[start] = 0;

    // далее происходит заполнение массива с помощью списка ребер:
    // считается, что если путь до вершины i кратчайший, то путь до вершины j определится как сумма путей от start до i и от i до j
    // полагается, что за n итераций определятся все наикратчайшие пути до всех n вершин
    for (unsigned int idx = 0; idx < vertexs.length() - 1; idx++){
        for (unsigned int jdx = 0; jdx < ways.length(); jdx++){
            ans[ways[jdx].to] = std::min(ans[ways[jdx].to], ans[ways[jdx].from] + ways[jdx].cost);
        }
    }

    clock_t end = clock();
    double rez_time = (double)(end - begin) / (double)CLOCKS_PER_SEC;
    std::cout << std::setprecision(5) << "Time taken by program is : " << rez_time << "\n";

    return ans;
}


Dynamic_array<Dynamic_array<unsigned int>> Graph::Floyd_Warshall() {
    clock_t begin = clock();

    // алгоритм Флойда-Уоршелла определяет наикратчайшие пути от всех вершин до всех
    // поэтому на нулевой итерации результирующая матрица соответствует матрице смежности
```

```
    Dynamic_array<Dynamic_array<unsigned int>> ans = adjacency_matrix;

    // далее для каждой k-ой вершины проверяется следующее:
    // какой из путей от i до j (напрямую или через вершину k (от i до k + от k до j))
минимальным
    for (unsigned int kdx = 0; kdx < vertexs.length(); kdx++){
        for (unsigned int row = 0; row < vertexs.length(); row++){
            for (unsigned int col = 0; col < vertexs.length(); col++){
                ans[row][col] = std::min(ans[row][col], ans[row][kdx] + ans[kdx][col]);
            }
        }
    }

    clock_t end = clock();
    double rez_time = (double)(end - begin) / (double)CLOCKS_PER_SEC;
    std::cout << std::setprecision(5) << "Time taken by program is : " << rez_time << "\n";

    return ans;
}



void Bellman_Ford_test(Graph A){

    A.generate_ways();
    A.output(true);

    Dynamic_array<unsigned int> distance = A.Bellman_Ford(0);
    for (unsigned int idx = 0; idx < distance.length(); idx++){
        std::cout << distance[idx] << "  ";
    }
    std::cout << "\n";
    distance = A.Bellman_Ford(1);
    for (unsigned int idx = 0; idx < distance.length(); idx++){
        std::cout << distance[idx] << "  ";
    }
    std::cout << "\n";
    distance = A.Bellman_Ford(2);
    for (unsigned int idx = 0; idx < distance.length(); idx++){
        std::cout << distance[idx] << "  ";
    }
    std::cout << "\n";
    distance = A.Bellman_Ford(3);
    for (unsigned int idx = 0; idx < distance.length(); idx++){
        std::cout << distance[idx] << "  ";
    }
    std::cout << "\n";
    distance = A.Bellman_Ford(4);
    for (unsigned int idx = 0; idx < distance.length(); idx++){
        std::cout << distance[idx] << "  ";
    }
    std::cout << "\n";
```

```
    distance = A.Bellman_Ford(5);
    for (unsigned int idx = 0; idx < distance.length(); idx++){
        std::cout << distance[idx] << "  ";
    }
    std::cout << "\n";
    distance = A.Bellman_Ford(6);
    for (unsigned int idx = 0; idx < distance.length(); idx++){
        std::cout << distance[idx] << "  ";
    }
    std::cout << "\n";

    std::cout << "\n";
}


void Floyd_Warshall_test(Graph A){

    A.generate_matrix();
    A.output(true);

    Dynamic_array<Dynamic_array<unsigned int>> dist_matrix = A.Floyd_Warshall();
    for (unsigned int idx = 0; idx < dist_matrix.length(); idx++){
        for (unsigned int jdx = 0; jdx < dist_matrix[idx].length(); jdx++){
            if (dist_matrix[idx][jdx] == MAX_COUNT)
                std::cout << "*  ";
            else
                std::cout << dist_matrix[idx][jdx] << "  ";
        }
        std::cout << "\n";
    }
    std::cout << "\n";

}


int main() {

    Graph A;

    unsigned int N = 950;
    for (unsigned int idx = 0; idx < N; idx++){
        A.add_node(idx);
    }

    unsigned int M = 2*N;
    for (unsigned int idx = 0; idx < M; idx++){
        unsigned int source = rand() % N, target;
        do{
            target = rand() % N;
        } while (target == source);
        unsigned int cost = rand() % 3 + 1;
```

```
        A.add_path(source, target, cost);
    }

    //A.output();

    A.generate_ways();
    Dynamic_array<unsigned int> distance = A.Bellman_Ford(0);

    A.generate_matrix();
    Dynamic_array<Dynamic_array<unsigned int>> dist_matrix = A.Floyd_Warshall();

    return 0;
}
```

Заголовочный файл dynamic_array.h

```
#ifndef GRAPH_DYNAMIC_ARRAY_H
#define GRAPH_DYNAMIC_ARRAY_H



template <typename T>
class Dynamic_array{

    //static_assert();

protected:

    unsigned int size;

public:
    T* data;

    Dynamic_array();
    Dynamic_array(const unsigned int size);
    Dynamic_array(const Dynamic_array<T>& x);
    ~Dynamic_array();

    T& operator[](unsigned int idx);

    void push_back(T elm);
    void pop_back();
    unsigned int position(T elm);  // возвращает позицию первого найденного элемента + 1;
    void clear();

    void shrink(unsigned int new_size);
    void extend(unsigned int new_size);
```

```
    unsigned int length();
    void output();

};


template <typename T>
Dynamic_array<T>::Dynamic_array() {
    size = 0;
    data = nullptr;
}


template <typename T>
Dynamic_array<T>::Dynamic_array(const unsigned int new_size) {
    size = new_size;
    data = new T[size];
}


template <typename T>
Dynamic_array<T>::Dynamic_array(const Dynamic_array<T>& x) {
    size = x.size;
    data = new T[size];

    for (unsigned int idx = 0; idx < x.size; idx++){
        data[idx] = x.data[idx];
    }
}


template <typename T>
Dynamic_array<T>::~Dynamic_array() {
    delete[] data;
}


template <typename T>
T& Dynamic_array<T>::operator[](unsigned int idx) {
    return data[idx];
}


template <typename T>
void Dynamic_array<T>::push_back(T elm) {
    //double temp[size + 1];
    T* temp = new T[size + 1];
    for (unsigned int idx = 0; idx < size; idx++){
        temp[idx] = data[idx];
    }
    delete[] data;
    data = temp;
```

```
    data[size++] = elm;
}


template <typename T>
void Dynamic_array<T>::pop_back() {
    // удаление последнего элемента и уменьшение размера массива на 1?;
    //double temp[size + 1];
    T* temp = new T[--size];
    for (unsigned int idx = 0; idx < size; idx++){
        temp[idx] = data[idx];
    }
    //delete[] data;
    data = temp;

}


template <typename T>
unsigned int Dynamic_array<T>::position(T elm) {
    for (unsigned int idx = 0; idx < size; idx++){
        if (data[idx] == elm){
            return idx + 1;
        }
    }
    return 0;
}


template <typename T>
void Dynamic_array<T>::clear() {
    delete[] data;
    data = nullptr;
    size = 0;
}


template <typename T>
void Dynamic_array<T>::shrink(unsigned int new_size) {
    if (new_size >= size) return;
    double* temp = new double[new_size];
    for (unsigned int idx = 0; idx < new_size; idx++){
        temp[idx] = data[idx];
    }
    delete[] data;
    data = temp;
    size = new_size;
}


template <typename T>
void Dynamic_array<T>::extend(unsigned int new_size) {
```

```
    if (new_size <= size) return;
    T* temp = new T[new_size];
    for (unsigned int idx = 0; idx < size; idx++){
        temp[idx] = data[idx];
    }
    delete[] data;
    data = temp;
    size = new_size;
}


template <typename T>
unsigned int Dynamic_array<T>::length(){
    return size;
}

#endif //GRAPH_DYNAMIC_ARRAY_H
```