Санкт–Петербургский 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-Уоршелла

Студент гр. 3331506/20101 Клестов А.Д.

Преподаватель Ананьевский М.С.

Санкт-Петербург

2025

Оглавление

Введение

Алгоритм Беллмана-Форда – алгоритм, предназначенный для нахождения кратчайших путей от вершины-источника до всех остальных вершин во взвешенном графе. В отличие от алгоритма Дейкстры, он может работать с графами, которые содержат рёбра с отрицательными весами, а также находить отрицательные циклы, достижимые из исходной вершины

Алгоритм Флойда-Уоршелла — алгоритм, предназначенный для нахождения кратчайших путей между всеми парами вершин во взвешенном ориентированном графе. Также может работать с графами, которые содержат рёбра с отрицательными весами, а также находить отрицательные циклы.

В данной курсовой работе реализованы две программы: с графом, задаваемым вручную, с графом на основе базы данных аэропортов США. Рассчитано время выполнения алгоритмов.

1. Описание алгоритма Беллмана-Форда

Алгоритм Беллмана-Форда основан на идее последовательного улучшения оценок кратчайших путей путем многократного просмотра всех рёбер графа.

Шаги алгоритма:

- Инициализация: устанавливаем расстояние до начальной вершины равным 0, а до всех остальных вершин — бесконечность.
- Релаксация рёбер: производим |V|-1 итераций (где V – число вершин графа), на каждой из которых просматриваем все рёбра графа и пытаемся улучшить текущие оценки расстояний.
- Проверка на отрицательные циклы: после |V|-1 итераций выполняем дополнительную проверку — если удаётся улучшить хотя бы одну оценку расстояния, то в графе присутствует отрицательный цикл – невозможно построить кратчайший маршрут, так как при наличии отрицательных весов рёбер графа, проходя отрицательный цикл, можно бесконечно улучшать маршрут

2. Описание алгоритма Флойда-Уоршелла

Алгоритм Флойда-Уоршелла основан идее постепенного улучшении оценок кратчайших путей путем рассмотрения промежуточных вершин.

Шаги алгоритма:

- Инициализация матрицы расстояний dist, где dist[i][j] равно весу ребра (i,j) или бесконечности, если ребра нет.
- Последовательно рассматриваем каждую вершину k как потенциальную промежуточную.
- Для каждой пары вершин (i,j) проверяем, улучшится ли путь через вершину k.
- Обновляем матрицу расстояний, если найден более короткий путь
- Проверяем наличие отрицательного цикла, которое можно определить по отрицательным значениям на диагонали итоговой матрицы

3. Программная реализация алгоритмов

Было написано несколько программ: для инициализации функций, с помощью которых реализованы алгоритмы, реализация алгоритмов, пример применения на графе, задаваемым вручную, пример применения на графе, построенном на основе базы данных авиаперелётов США. Все программы представлены в приложении.

Время работы алгоритмов в разных ситуациях представлено в таблице 1

Таблица 1 – Время работы алгоритмов

Алгоритм	Ситуация	Время выполнения, мс
Беллмана-Форда	Граф, задаваемый вручную	2
Флойда-Уоршелла		2
Беллмана-Форда	Граф на основе файла с 40 перелётами	2
Флойда-Уоршелла		1
Беллмана-Форда	Граф на основе файла с 32000 перелётами	-
Флойда-Уоршелла		227159

Примечательно, что при работе с большим количеством данных, алгоритм Беллмана-Форда оказался неэффективен, а алгоритм Флойда-Уоршелла справился почти за 4 минуты. При этом с небольшим числом перелётов проблем не возникает. Это возникает из-за слишком высокой плотности графа

Алгоритмы могут работать при отрицательных весах рёбер графа, что реализовано в первой программе, представленной в приложении

Заключение

В ходе выполнения курсовой работы были рассмотрены алгоритмы Беллмана-Форда и Флойда-Уоршелла. Данные алгоритмы были применены для анализа базы данных перелётов по аэропортам США. В результате было выяснено, что для анализа больших плотных графов больше подходит алгоритм Флойда-Уоршела, так как алгоритм Беллмана-Форда не смог справиться с задачей

Список литературы

1. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы: построение и анализ. - 3-е изд. - М.: Вильямс, 2022. - 1328 с.
2. А. Ахо, Дж. Хопкрофт, Дж. Ульман. Построение и анализ вычислительных алгоритмов. - М.: Мир, 1985. - 536 с.
3. R. Sedgewick Algorithms in C, Part 5: Graph Algorithms. - 3rd edition - Addison-Wesley, 2001. - 528 p.
4. Б. Страуструп Язык программирования C++. - 4-е изд. - М.: Бином, 2023. - 1216 с.
5. С. Мейерс Эффективный и современный C++. - М.: Вильямс, 2020. - 304 с.

Приложение

1. pathfinder.h

```
#pragma once
#include <vector>
#include <string>
#include <unordered_map>

struct Edge {
    std::string destination;
    int weight;
};

struct Point {
    std::string point;
    std::vector<Edge> edges;
};

std::pair<std::vector<std::string>, int> BellmanFord(const
std::unordered_map<std::string, Point> graph,
const std::string& start, const std::string& end);

std::pair<std::vector<std::string>, int> FloydWarshall(const
std::unordered_map<std::string, Point> graph,
const std::string& start, const std::string& end);
```

2. pathfinder.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>


struct Edge {
    std::string destination;
    int weight;
};

struct Point {
    std::string point;
    std::vector<Edge> edges;
};

std::pair<std::vector<std::string>, int> BellmanFord(const
std::unordered_map<std::string, Point> graph,
const std::string& start, const std::string& end)
{
    if (graph.find(start) == graph.end() || graph.find(end) == graph.end()) {
        return { {}, std::numeric_limits<int>::max() };
    }

    // Инициализация расстояний и порядка прохождения
    std::unordered_map<std::string, int> distance;
    std::unordered_map<std::string, std::string> order;

    for (const auto& point : graph) {
        distance[point.first] = std::numeric_limits<int>::max();
    }
    distance[start] = 0;

    // Обработка графа для применения алгоритма
    std::vector<std::pair<std::string, Edge>> edges;
    for (const auto& point : graph) {
```

```
        for (const auto& edge : point.second.edges) {
            edges.emplace_back(point.first, edge);
        }
    }

    // Основной цикл алгоритма
    for (size_t idx = 0; idx < graph.size() - 1; idx++) {
        for (const auto& cur_edge : edges) {
            if (distance[cur_edge.first] != std::numeric_limits<int>::max() &&
                distance[cur_edge.first] + cur_edge.second.weight <
distance[cur_edge.second.destination]) {
                distance[cur_edge.second.destination] = distance[cur_edge.first] +
cur_edge.second.weight;
                order[cur_edge.second.destination] = cur_edge.first;
            }
        }
    }

    // Проверка на отрицательные циклы
    for (const auto& point : graph) {
        for (const auto& edge : point.second.edges) {
            if (distance[point.first] != std::numeric_limits<int>::max() &&
                distance[point.first] + edge.weight < distance[edge.destination]) {
                //throw std::runtime_error("Граф содержит отрицательный цикл");
            }
        }
    }

    if (distance[end] == std::numeric_limits<int>::max()) {
        return{{}, std::numeric_limits<int>::max() };
    }

    // Сборка пути
    std::vector<std::string> path;
    for (std::string at = end; at != ""; at = order[at]) {
        path.push_back(at);
    }
    reverse(path.begin(), path.end());

    return {path, distance[end]};
}


std::pair<std::vector<std::string>, int> FloydWarshall(const
std::unordered_map<std::string, Point> graph,
    const std::string& start, const std::string& end)
{
    // Обработка графа для применения алгоритма
    std::vector<std::string> points;

    if (graph.find(start) == graph.end() || graph.find(end) == graph.end()) {
        return {{}, std::numeric_limits<int>::max()};
    }

    for (const auto& point : graph) {
        points.push_back(point.first);
    }

    // Инициализация расстояний и порядка прохождения
    std::unordered_map<std::string, std::unordered_map<std::string, int>> distance;
    std::unordered_map<std::string, std::unordered_map<std::string, std::string>>
order;

    for (const auto& point : points) {
        for (const auto& destinatoin : points) {
```

```
            if (point == destinatoin) {distance[point][destinatoin] = 0; }
            else { distance[point][destinatoin] = std::numeric_limits<int>::max(); }
            order[point][destinatoin] = "";
        }
    }

    // Заполнение рёбер минимальными расстояниями
    for (const auto& point : graph) {
        for (const auto& edge : point.second.edges) {
            if (distance[point.first][edge.destination] > edge.weight) {
                distance[point.first][edge.destination] = edge.weight;
                order[point.first][edge.destination] = edge.destination;
            }
        }
    }

    // Основной алгоритм
    for (const auto& k : points) {
        for (const auto& i : points) {
            for (const auto& j : points) {
                if (distance[i][k] != std::numeric_limits<int>::max() &&
                    distance[k][j] != std::numeric_limits<int>::max() &&
                    distance[i][k] + distance[k][j] < distance[i][j]) {
                    distance[i][j] = distance[i][k] + distance[k][j];
                    order[i][j] = order[i][k];
                }
            }
        }
    }

    // Проверка на наличие отрицательных циклов
    if (distance[start][start] < 0 || distance[end][end] < 0) {
        throw std::runtime_error("Граф содержит отрицательный цикл");
    }

    if (order[start][end].empty()) {
        return {{}, std::numeric_limits<int>::max() };
    }

    // Сборка пути
    std::vector<std::string> path;
    for (std::string at = start; at != end; at = order[at][end]) {
        path.push_back(at);
    }
    path.push_back(end);

    return {path, distance[start][end]};
}
```

3. example custom

```
#include <fstream>
#include <iostream>
#include <chrono>

#include "pathfinder.h"

void print_result(std::unordered_map<std::string, Point> graph,
    const std::string& start, const std::string& end, const std::string&
algoritm_name) {
    auto begin_time = std::chrono::steady_clock::now();
    std::vector<std::string> path;
    int dist;
    if (algoritm_name == "Bellman-Ford") {
        path = BellmanFord(graph, start, end).first;
        dist = BellmanFord(graph, start, end).second;
```

```
    }
    else if (algoritm_name == "Floyd-Warshall") {
        path = FloydWarshall(graph, start, end).first;
        dist = FloydWarshall(graph, start, end).second;
    }
    else {
        std::cout << "Incorrect algoritm name" << std::endl;
        return;
    }
    if (path.empty() || dist == -1) {
        std::cout << "No path found" << std::endl;
    }
    else {
        std::cout << "Optimal path:" << std::endl;
        for (size_t idx = 0; idx < path.size(); idx++) {
            if (idx != 0) std::cout << " - ";
            std::cout << graph[path[idx]].point;
        }
        std::cout << "\ntotal distance: " << dist << std::endl;
    }
    auto end_time = std::chrono::steady_clock::now();
    auto time = std::chrono::duration_cast<std::chrono::milliseconds>(end_time -
begin_time);
    std::cout << "  algoritm time: " << time.count() << " ms\n";
}

int main() {
    std::unordered_map<std::string, Point> graph = {
        {"AMO", {"AMO", {{"HUH", 52}, {"GUS", 7}, {"IMP", 33}}}},
        {"GUS", {"GUS", {{"HUH", -17}, {"IMP", 3}}}},
        {"HUH", {"HUH", {{"SAD", -66}, {"AMO", 12}}}},
        {"SAD", {"SAD", {}}},
        {"IMP", {"IMP", {{"OST", 4}, {"AMO", 2}}}},
        {"OST", {"OST", {}}},
    };

    std::cout << "Bellman-Ford:" << std::endl;
    print_result(graph, "AMO", "OST", "Bellman-Ford");
    std::cout << "Floyd-Warshall:" << std::endl;
    print_result(graph, "AMO", "OST", "Floyd-Warshall");

    return 1;
}
```

4. example airports

```
#include <fstream>
#include <iostream>
#include <chrono>

#include "pathfinder.h"

std::vector<std::string> split(const std::string& line)
{
    std::vector<std::string> splitted_line;
    std::string word;
    bool in_quotes = false;

    for (char letter : line) {
        if (letter == '"') {
            in_quotes = !in_quotes;
        }
        else if (letter == ',' && !in_quotes) {
            splitted_line.push_back(word);
            word.clear();
        }
```

```
        else {
            word += letter;
        }
    }
    splitted_line.push_back(word);

    return splitted_line;
}

// получение данных из CSV и запись в граф
std::unordered_map<std::string, Point> get_airport_data(const std::string&
file_name)
{
    std::ifstream file(file_name);

    // Порядковые номера нужных позиций из CSV
    size_t distance_file_number = 7;
    size_t air_time_file_number = 9;
    size_t origin_file_number = 22;
    size_t origin_city_name_file_number = 23;
    size_t destination_file_number = 33;
    size_t destination_city_name_file_number = 34;

    size_t year_file_number = 44;
    size_t month_file_number = 46;

    std::unordered_map<std::string, Point> airports;

    if (!file.is_open()) {
        std::cerr << "File was not opened";
        throw std::invalid_argument("File was not opened");
    }

    std::string line;
    while (getline(file, line)) {
        std::vector<std::string> words = split(line);

        int air_time = 0;
        int distance = 0;
        if (words[distance_file_number] != "DISTANCE" && stoi(words[1]) == 29) {
            try {
                if (stoi(words[air_time_file_number]) == 0) {
                    air_time = std::numeric_limits<int>::max();
                }
                else {
                    air_time = stoi(words[air_time_file_number]);
                }
                distance = stoi(words[distance_file_number]);
            }
            catch (...) {
                std::cerr << "Invalid air time: " << words[air_time_file_number] <<
"or ";
                std::cerr << "Invalid distance: " << words[distance_file_number];
                continue;
            }
            std::string origin = words[origin_file_number];
            std::string origin_city_name = words[origin_city_name_file_number];
            std::string destination = words[destination_file_number];
            std::string destination_city_name =
words[destination_city_name_file_number];

            // Добавление аэропорта отправления
            if (airports.find(origin) == airports.end()) {
                airports[origin] = { origin_city_name, {} };
            }
```

```
            // Добавление аэропорта прибытия
            if (airports.find(destination) == airports.end()) {
                airports[destination] = { destination_city_name, {} };
            }

            // Добавление в граф
            airports[origin].edges.push_back({destination, air_time});
        }
    }

    file.close();
    return airports;
}

std::tuple<std::unordered_map<std::string, Point>, std::string, std::string>
request(const std::string& file_name) {
    auto airports = get_airport_data(file_name);

    std::string start_airport;
    std::string finish_airport;
    std::cout << "Enter start airport code: ";
    std::cin >> start_airport;
    std::cout << "Enter finish airport code: ";
    std::cin >> finish_airport;

    return {airports, start_airport, finish_airport};
}

void print_result(const std::string& file_name, const std::string& algoritm_name) {
    auto req = request(file_name);
    std::unordered_map<std::string, Point> airports = std::get<0>(req);
    std::string start_airport = std::get<1>(req);
    std::string finish_airport = std::get<2>(req);

    auto begin = std::chrono::steady_clock::now();

    std::vector<std::string> path;
    int total_time;
    if (algoritm_name == "Bellman-Ford") {
        path = BellmanFord(airports, start_airport, finish_airport).first;
        total_time = BellmanFord(airports, start_airport, finish_airport).second;
    }
    else if (algoritm_name == "Floyd-Warshall") {
        path = FloydWarshall(airports, start_airport, finish_airport).first;
        total_time = FloydWarshall(airports, start_airport, finish_airport).second;
    }
    else {
        std::cout << "Incorrect algoritm name" << std::endl;
        return;
    }
    if (path.empty() || total_time == -1) {
        std::cout << "No path found" << std::endl;
    }
    else {
        std::cout << "Optimal path:" << std::endl;
        for (size_t idx = 0; idx < path.size(); idx++) {
            if (idx != 0) std::cout << " - ";
            std::cout << airports[path[idx]].point;
        }
        std::cout << "\ntotal time: " << total_time << " minutes" << std::endl;
    }

    auto end = std::chrono::steady_clock::now();
    auto time = std::chrono::duration_cast<std::chrono::milliseconds>(end - begin);
```

```
    std::cout << "  algoritm time: " << time.count() << " ms\n";
}

int main() {
    std::cout << "Bellman-Ford:" << std::endl;
    print_result("T_T100_SEGMENT_ALL_CARRIER.csv", "Bellman-Ford");

    std::cout << "Floyd-Warshall:" << std::endl;
    print_result("T_T100_SEGMENT_ALL_CARRIER.csv", "Floyd-Warshall");

    return 1;
}
```