МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Построение и Анализ Алгоритмов»
Тема: Жадный алгоритм и A*

Студент гр. 3384 _____ Рудаков А.Л.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2025

Цель работы.

Написание жадного алгоритма и алгоритма A* на языке программирования. Анализ созданного алгоритма.

Задание.

1. Разработайте программу, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

Пример входных данных

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

В первой строке через пробел указываются начальная и конечная вершины

Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

abcde

2. Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A*. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет

неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

Пример входных данных

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

В первой строке через пробел указываются начальная и конечная вершины

Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

ade

Выполнение работы.

Для задания 1 разработан алгоритм, находящийся в фале first.cpp.

В нем реализован класс *GreedySearch*, в котором реализована вся логика работы алгоритма. Полями класса являются *char start, char end* – начальная и конечная вершины, *std::string way* – текущий путь по вершинам, *std::map<char, bool> used* – map, в котором хранится информация о том, были ли посещена вершина или нет, если не посещена, значение равно *true*, иначе *false*, *std::map<char, std::vector<Edge>> vertexes* – map, хранящий информацию о путях из каждой вершины, записанные в виде вектора структур *Edge*, в которых записаны конечная вершина ребра и вес данного ребра.

В методе *void input()* происходит считывание поданных значений, заполняя поля класса.

Метод *void start_find()* создан для начала поиска. Вершина *start* отмечается как использованная, а также добавляется к пути. После этого вызывается метод *find(start).*

Метод *bool find(char current)* ищет путь до конечной вершины используя жадный поиск. Внутри запускается цикл *while*, в котором происходит поиск. В цикле *for* происходит проход по ребрам, исходящим из вершины *current*, который ищет ребро с наименьшим весом, в вершину, которая еще не была использована и которая уже не была просмотрена при предыдущих просмотрах ребер из данной вершины (если такие были). Если такого пути не было найдено, то возвращается *false*, что значит, что путь привел в тупик. Если путь был найден, то найденная вершина отмечается использованной и добавляется к пути. Далее проверяется на равенство текущей вершины и конечной вершины, если они равны, возвращается *true*. Если не равны, то запускается *find(..)* для найденной вершины. Если получено true, значит путь был найден, поэтому возвращается *true* для выхода из рекурсии, в противном случае вершина помечается как неиспользованная и удаляется из пути, а также добавляется к вершинам, которые уже были просмотрены из данной вершины, после чего начинается следующая итерация цикла *while*.

Метод *void print()* выводит найденный путь.

Результаты тестирования приведены в табл. Б1.

Для задания 2 разработан алгоритм, находящийся в фале second.cpp.

В нем реализован класс *AStarSearch*, в котором реализована вся логика работы алгоритма. Полями класса являются *char start, char end* – начальная и конечная вершины, *std::map<char, std::vector<Edge>> vertexes* – map, хранящий информацию о путях из каждой вершины, записанные в виде вектора структур *Edge*, в которых записаны конечная вершина ребра и вес данного ребра, *std::map<char, VertexInfo> vertex_info* – map, хранящий информацию о вершинах в виде структур *VertexInfo*, которые хранят в себе *char vertex* – название вершины, *float f_score* – текущий минимальный путь до вершины, *float*

g_score – текущее минимальное значение эвристической функции до вершины, *std::string way* – текущий путь до вершины.

В методе *void input()* происходит считывание поданных значений, заполняя поля класса.

В методе *float comp_heuristic_function(char start_vertex, char end_vertex)* подсчитывается значение эвристической функции между поданными вершинами (расстояние между символами).

Метод find выполняет поиск по алгоритму A*. Внутри инициализируется очередь с приоритетом *std::priority_queue<VertexInfo, std::vector<VertexInfo>, std::greater<VertexInfo>> queue*, в которой будут хранится элементы *VertexInfo*. Приоритет в очереди выставляется по значению *f_score*. Далее в очередь добавляется *VertexInfo* начальной вершины. После этого выполняется цикл *while*, до тех пор, пока очередь не пуста. Внутри в *current* записывается выходное значение очереди, так же оно удаляется из нее. Если вершина вытащенного значения равна конечной, то цикл завершается. Далее происходит итерация по ребрам, исходящим из текущей вершины. Рассчитывается значение предварительного значения пути (вес ребра + *g_score* текущей вершины), после чего происходит его сравнение с *g_score* вершины, в которую ведет ребро, если новое значение оказывается меньше, то у найденной вершины меняется *g_score* (становиться равен *prior_distance*), *f_score* становится равным сумме предварительного значения пути и эвристической функции между найденной и конечной вершиной, а также путь way становиться равным пути текущей вершины + найденная вершина. После этого *VertexInfo* найденной вершины добавляется в очередь.

Метод *void print()* выводит найденный путь до конечной вершины.

Результаты тестирования приведены в табл. Б2.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Дополнительное задание.

1. Написать генератор графа, который выдает граф с рандомными весами ребер в промежутке [1, 100] и переданной "полнотой" графа. Полным - 100% - считается граф, у которого у любой вершины есть ребра в каждую другую вершину графа. Далее от процента зависит с какой вероятностью будет каждое ребро

2. Реализовать алгоритм Дейкстры. Выдать кратчайшие пути от изначальной до всех остальных

3. Исследовать алгоритм на время в зависимости от полноты графа. Если до вершины нет пути от изначальной - то выдавать бесконечность.

4. Не забываем, что на каждый случай у нас 10 снятий

Исследование.

Для дополнительного задания разработан алгоритм, находящийся в фале add_exersize.cpp.

В нем реализован класс *DijkstraSearch* внутри которого реализована вся логика работы.

В методе *generate_graph()* по заданному значению количества вершин и значения полноты генерируется граф, с рандомными весами ребер.

В методе *int find_min_vertex()* происходит поиск ближайшей к началу неиспользованной вершины (вершины с минимальным весом пути).

Метод *find(char current)* обновляет веса вершин, до которых есть пути из текущей поданной, если они оказываются меньше, чем те, которые были до них.

В методе *void start_find()* происходит поиск вершин через *find_min_vertex(),* до тех пор пока все вершины не будут использованы. К найденным вершинам применяется метод *find(..).*

При замере времени использовалась библиотека chrono. Для построения графиков использовалась библиотека matplotlib.

Были проведены замеры времени работы алгоритма Дейкстры при различных условиях: различном количестве вершин и различной степени

заполненности графа. Для каждой величины выполнялось 10 замеров, после чего вычислялось их среднее значение. По полученным данным построены график зависимости времени работы алгоритма относительно заполненности графа, представленный на рис. 1 и график зависимости времени работы алгоритма относительно количества вершин (относительно полной заполненности графа), представленный на рис. 2.

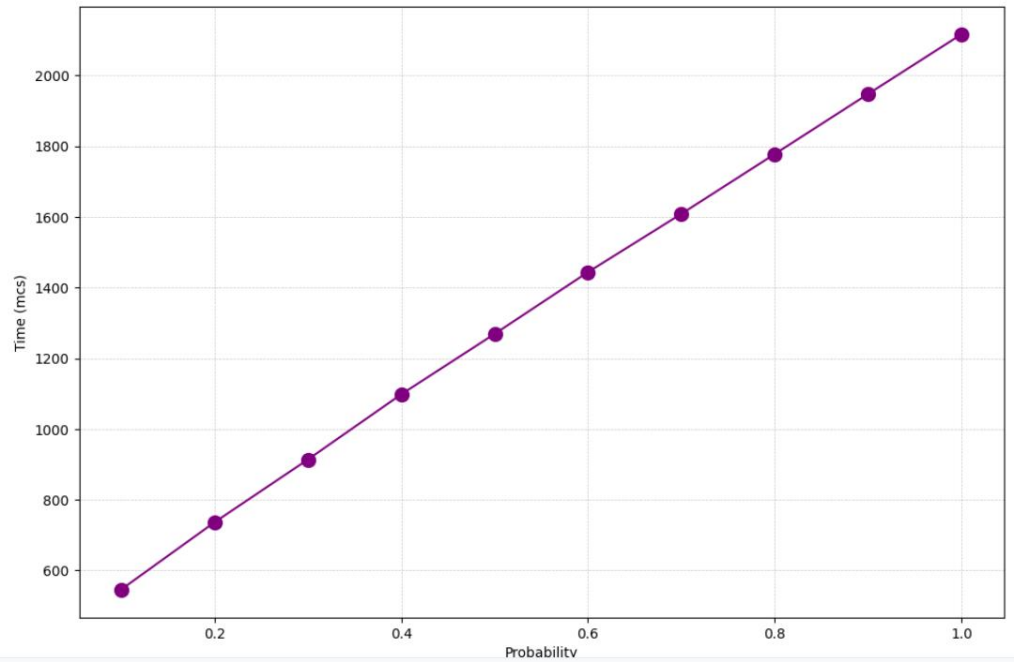

Рисунок 1 – Зависимость времени выполнения от заполненности графа

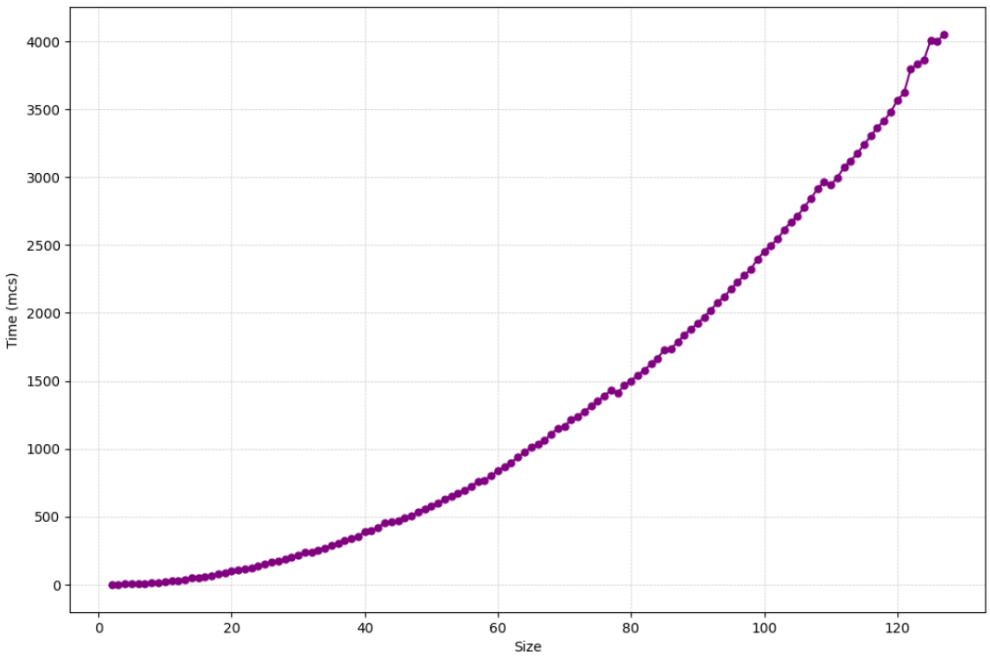

Рисунок 2 – Зависимость времени выполнения от количества вершин графа

Из графика на рис. 1 заметно, что время выполнения алгоритма Дейкстры линейно зависит от заполненности графа. Также, по графику на рис.2 можно сказать, что время выполнения алгоритма в О-нотации зависит от количества вершин как O(n^2).

Результаты тестирования приведены в табл. Б3.

Выводы.

В ходе работы был реализован алгоритм жадного поиска, позволяющий решать задачу о нахождении минимального пути от заданной начальной вершины до заданной конечной вершины, выбирая за лучший путь ребро с наименьшим весом. В алгоритме так же используется алгоритм бэктрекинге, для просмотра других вершин, в случае попадания в тупик. Также был реализован алгоритм A*, позволяющий решать задачу о нахождении минимального пути от заданной начальной вершины до заданной конечной вершины, учитывая при выборе лучшего пути не только вес ребра, но и эвристическую оценку нового пути и путей, предшествовавших ему. В алгоритме используется очередь с приоритетом для выбора новой вершины для просмотра в оптимальном для корректной работы алгоритма порядке.

Для решения дополнительной задачи был написан алгоритм Дейкстры, а также алгоритм построения графа на основе общего количества вершин и вероятности появления ребра (заполненности графа). Для алгоритма Дейкстры было проведено исследование, которое выясняет зависимость времени работы алгоритма относительно заполненности графа и относительно общего количества вершин в графе. В результате исследования было получено, что время работы алгоритма линейно зависит от заполненности графа (O(n)). Также было замечено, что время работы зависит алгоритма зависит от количества вершин в полном графе как O(n^2). Данная оценка подтверждается тем, что алгоритм проходит по каждой вершине, при этом каждый раз проходя по соседям выбранной вершины. Также есть поиск новой вершины, который не

влияет на O-нотацию, так как теоретическая оценка алгоритма O(n*n + n) = O(n^2).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: first.cpp

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <utility>
#include <limits>

float FLOAT_MAX = std::numeric_limits<float>::max();

struct Edge {
    char vertex;
    float weight;
};

class GreedySearch {
    std::map<char, std::vector<Edge>> vertexes;
    std::map<char, bool> used;
    char start;
    char end;
    std::string way;

public:

    void input() {
        std::string new_string;
        getline(std::cin, new_string);
        start = new_string[0];
        end = new_string[2];
        while (getline(std::cin, new_string)) {
            if (new_string == "") break;
            char first = new_string[0];
            char second = new_string[2];
```

```
                float value = std::stof(new_string.substr(4,
new_string.size() - 1));
                if (vertexes.count(first) == 0) {
                    vertexes[first] = std::vector<Edge>();
                    used[first] = true;
                }
                vertexes[first].push_back({ second, value });
            }
            used[end] = true;
        }

        void start_find() {
            used[start] = false;
            way.push_back(start);
            find(start);
        }

        bool find(char current) {
            std::string use_vert;
            while (true) {
                char min_vert = '\0';
                float min_val = FLOAT_MAX;
                for (auto& variant : vertexes[current]) {
                    bool used_flag = false;
                    for (char used_vert : use_vert) {
                        if (variant.vertex == used_vert) {
                            used_flag = true;
                            break;
                        }
                    }
                    if (used_flag) continue;
                    if (min_val > variant.weight || min_val ==
variant.weight && variant.vertex == end) {
                        if (used[variant.vertex]) {
                            min_val = variant.weight;
                            min_vert = variant.vertex;
                        }
                    }
```

```
            if (min_vert == '\0') return false;
            used[min_vert] = false;
            way.push_back(min_vert);
            if (min_vert == end) return true;

            if (find(min_vert)) return true;
            else {
                way.pop_back();
                used[min_vert] = true;
                use_vert.push_back(min_vert);
            }
        }
    }

    void print() {
        std::cout << way << '\n';
    }

};

int main() {
    GreedySearch greedy_search;
    greedy_search.input();
    greedy_search.start_find();
    greedy_search.print();
}
```

Название файла: second.cpp

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <utility>
#include <limits>
#include <queue>

float FLOAT_MAX = std::numeric_limits<float>::max();
```

```
    struct Edge {
        char vertex;
        float weight;
    };

    struct VertexInfo {
        char vertex;
        float f_score;
        float g_score;
        std::string way;
        bool operator>(const VertexInfo& other) const {
            return f_score > other.f_score;
        }
    };

    class AStarSearch {
        std::map<char, std::vector<Edge>> vertexes;
        std::map<char, VertexInfo> vertex_info;
        char start;
        char end;

    public:

        void input() {
            std::string new_string;
            getline(std::cin, new_string);
            start = new_string[0];
            end = new_string[2];
            while (getline(std::cin, new_string)) {
                if (new_string == "") break;
                char new_start = new_string[0];
                char new_end = new_string[2];
                float value = std::stof(new_string.substr(4,
new_string.size() - 1));
                if (vertexes.count(new_start) == 0) {
                    vertexes[new_start] = std::vector<Edge>();
                    vertex_info[new_start] = { new_start, FLOAT_MAX,
FLOAT_MAX, "" };
                }
```

```
                if (vertexes.count(new_end) == 0) {
                    vertexes[new_end] = std::vector<Edge>();
                    vertex_info[new_end] = { new_end, FLOAT_MAX,
FLOAT_MAX, "" };
                }
                vertexes[new_start].push_back({ new_end, value });
            }
            vertex_info[start] = { start, comp_heuristic_function(start,
end), 0, std::string(1, start) };
        }


        void find() {
            std::priority_queue<VertexInfo, std::vector<VertexInfo>,
std::greater<VertexInfo>> queue;
            queue.push(vertex_info[start]);

            while (!queue.empty()) {
                VertexInfo current = queue.top();
                queue.pop();

                if (current.vertex == end) return;

                for (auto& vertex : vertexes[current.vertex]) {
                    float prior_distance = current.g_score +
vertex.weight;

                    if (prior_distance < vertex_info[vertex.vertex].g_score) {
                        vertex_info[vertex.vertex].g_score =
prior_distance;
                        vertex_info[vertex.vertex].f_score =
prior_distance + comp_heuristic_function(vertex.vertex, end);
                        vertex_info[vertex.vertex].way = current.way +
vertex.vertex;

                        queue.push(vertex_info[vertex.vertex]);
                    }
```

```
                }
            }
        }

        float comp_heuristic_function(char start_vertex, char
end_vertex) {
            float a = abs(float(start_vertex - end_vertex));
            return a;
        }

        void print() {
            std::cout << vertex_info[end].way << '\n';
        }
    };

    int main() {
        AStarSearch a_star_search;
        a_star_search.input();
        a_star_search.find();
        a_star_search.print();
    }
```

Название файла: add_exersize.cpp

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <utility>
#include <random>
#include <limits>
#include <chrono>

float FLOAT_MAX = std::numeric_limits<float>::max();

struct Edge {
    int vertex;
    float weight;
};
```

```
    struct Way {
        float distance;
        std::string way;
    };

    class DijkstraSearch {
        std::map<int, std::vector<Edge>> vertexes;
        std::map<int, bool> used;
        std::map<int, Way> vertex_distance;
        int start;
        int count_vertex;
        float probability;

    public:

        void generate_graph() {
            std::random_device rd;
            std::mt19937 gen(rd());

            std::uniform_int_distribution<> int_dist(1, 100);
            std::uniform_real_distribution<> real_dist(0, 1);

            for (int i = 0; i < count_vertex; i++) {
                int current_vertex = i;
                used[current_vertex] = true;
                vertex_distance[current_vertex] = { FLOAT_MAX, "" };
                for (int j = 0; j < count_vertex; j++) {
                    int add_vertex = j;
                    if (current_vertex != add_vertex){
                        double rand_value = real_dist(gen);
                        if ((probability - rand_value) > 0.01) {
                            int weight = int_dist(gen);
                            if (vertexes.count(current_vertex) == 0) {
                                vertexes[current_vertex]                =
std::vector<Edge>();
                            }

vertexes[current_vertex].push_back({ add_vertex, float(weight) });
```

```
                            std::cout << current_vertex << " " <<
add_vertex << " " << weight << '\n';
                        }
                    }
                }
            }
            vertex_distance[start] = { 0, std::to_string(start) };
        }

        void other_input(int count_vertex, float probability, int start)
{
            this->count_vertex = count_vertex;
            this->probability = probability;
            this->start = start;
        }

        void input() {
            std::cin >> count_vertex >> probability >> start;
        }

        void start_find() {
            int current = find_min_vertex();
            while (current != -1) {
                find(current);
                current = find_min_vertex();
            }
        }

        void find(char current) {
            for (auto& variant : vertexes[current]) {
                if     (vertex_distance[variant.vertex].distance     ==
FLOAT_MAX ||
                    vertex_distance[variant.vertex].distance          >
vertex_distance[current].distance + variant.weight) {
                    vertex_distance[variant.vertex].distance          =
vertex_distance[current].distance + variant.weight;
                    vertex_distance[variant.vertex].way               =
vertex_distance[current].way + "-" + std::to_string(variant.vertex);
                }
```

```
            }
            used[current] = false;
        }

        char find_min_vertex() {
            int min_vert = -1;
            float min_val = FLOAT_MAX;
            for (auto& vertex_info : vertex_distance) {
                if (vertex_info.second.distance < min_val &&
used[vertex_info.first]) {
                    min_val = vertex_info.second.distance;
                    min_vert = vertex_info.first;
                }
            }
            return min_vert;
        }


        void print() {
            for (auto& vertex_info : vertex_distance) {
                std::cout << vertex_info.first << ": " <<
vertex_info.second.way << " " << vertex_info.second.distance << "\n";
            }
        }

    };

    int main() {
        /*DijkstraSearch dijkstra_search;
        dijkstra_search.input();
        dijkstra_search.generate_graph();
        dijkstra_search.start_find();
        std::cout << "=========\n";
        dijkstra_search.print();*/
        for (int i = 129; i < 130; i++){
            std::string out_str = "[";
            for (float j = 1.0; j > 0; j -= 0.1){
                out_str += '[';
                for (int l = 0; l < 10; l++){
```

```
                    DijkstraSearch dijkstra_search;
                    dijkstra_search.other_input(i, j, 0);
                    dijkstra_search.generate_graph();

                    auto                    start                    =
std::chrono::high_resolution_clock::now();
                    dijkstra_search.start_find();
                    auto                     end                     =
std::chrono::high_resolution_clock::now();

                    std::cout << "=========\n";
                    auto                   duration                  =
std::chrono::duration_cast<std::chrono::microseconds>(end - start);

                    out_str += std::to_string(duration.count()) + ",";
                    dijkstra_search.print();
                }
                out_str.pop_back();
                out_str += "],";
            }
            out_str.pop_back();
            out_str += "],\n";
            std::cout << out_str;
        }
    }
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Результаты тестирования для задания 1 приведены в табл. Б1.

Таблица Б.1 - Примеры тестовых случаев задания 1.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde	Ok
2.	a d a h 1 a z 8 h k 1 h p 1 h t 1 h g 1 h u 1 z d 100	azd	Ok
3.	a d a d 8.0 a b 3.0 b c 2.0 c d 1.0	abcd	Ok

Результаты тестирования для задания 2 приведены в табл. Б2.

Таблица Б.2 - Примеры тестовых случаев задания 2.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade	Ok
2.	a d a h 1 a z 8 h k 1 h p 1 h t 1 h g 1 h u 1 z d 100	azd	Ok
3.	a d a d 8.0 a b 3.0 b c 2.0 c d 1.0	abcd	Ok

Результаты тестирования для дополнительного задания приведены в табл. Б3.

Таблица Б.3 - Примеры тестовых случаев дополнительного задания.

№ п/п	Входные данные	Выходные данные	Комментарии
1.	3 1 0	0 1 72 0 2 3 1 0 89 1 2 38 2 0 17 2 1 32 ========= 0: 0 0 1: 0-2-1 35 2: 0-2 3	Ok
2.	5 0.3 0	0 1 19 1 2 62 2 4 75 3 1 12 3 2 35 ========= 0: 0 0 1: 0-1 19 2: 0-1-2 81 3: 3.40282e+38 4: 0-1-2-4 156	Ok (до третьей вершины нет пути, поэтому в ней значение бесконечности)