**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: Алгоритмы на графах**

Студентка гр. 8383 _____ Ишанина Л.Н.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

**Цель работы.**

Ознакомится с реализацией алгоритмов: жадный и A*, написать программу на языке программирования C++.

**Вариант 3**

Написать функцию, проверяющую эвристику на допустимость и монотонность.

## Жадный алгоритм.

### Описание используемого класса.

Класс путей(class Path):

Класс путей состоит из следующих свойств:

- char nameFrom – имя вершины откуда исходит путь(т.е. ребро графа);
- char nameOut – имя вершины куда входит путь(т.е. ребро графа);
- double weightPath – вес ребра графа;

Класс пути содержит следующие методы:

- Path(char nameFrom, char nameOut, double weightPath) – конструктор, принимающий имя вершины начала, имя вершины конца и вес этого ребра.
- char getNameFrom() – метод, для получения поля nameFrom класса Path.
- getNameOut() – метод, для получения поля nameOut класса Path.
- double getWeightPath() – метод, для получения поля weightPath класса Path.

### Постановка задачи.

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

**Описание алгоритма.**

Решение поставленной задачи осуществляется с помощью рекурсивной функции funk(std::vector<Path>* vector, char curChar, char endChar, std::vector<char>* answer), которая и реализует жадный алгоритм. В начале всегда происходит проверка – не является ли подаваемая начальная вершина равной конечной. Далее, пройдя проверку, создается пустой временный вектор путей(temporaryVector) и начинается проход по вектору исходному. В этом проходе происходит отбор нужных нам путей, т.е. тех, у которых nameFrom равняется текущей вершине, которую мы рассматриваем. Если они равны, то записываем в текущий временный вектор(temporaryVector). Далее, так как нужен самый “дешевый” путь, то вызываем функцию сортировки вершин(sort). И потом рекурсивно вызываем функцию func().

**Описание main ( ) :**

В функции прописан ввод вершины начальной, вершины конечной, а также ввод ребер с их весами, вызовы функций и выводы промежуточных данных на консоль.

**Описание дополнительных функций.**

Функция bool comp(Path a, Path b), принимает две переменные класса Path, и сравнивает их по весу.

**Сложность алгоритма по памяти.**

Сложность для жадного алгоритма оценивается как O(|V+E|), так как в исходном векторе может максимум хранится число всех ребер, и с каждой новой вершиной мы ‘проваливаемся’ в рекурсию и создаем новый вектор.

**Сложность алгоритма по времени.**

Сложность для жадного алгоритма оценивается как O(|E logE|), так как в худшем случае будет совершаться обход по всем ребрам, а изначальная сортировка ребер по длине имеет сложность O(|E logE|).

**Алгоритм A*.**

**Описание используемых классов.**

Класс вершин(class Top):

Класс вершин состоит из следующих свойств:

- char name – имя вершины;
- double pathToTop – путь до текущей вершины;
- double heuristicF – эврестическая функция;
- char nameFromT – имя, откуда исходит вершина.
- vector<char> coupled – вектор для исходящих из вершины вершин.

Класс вершин содержит следующие методы:

- Top(char name) – конструктор, принимающий имя вершины, используется для создания самой первой вершины.
- Top() – конструктор, для создания вершин по умолчанию.

**Постановка задачи.**

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

**Описание алгоритма.**

Вначале создаются 2 вектора: с открытыми вершинами и закрытыми вершинами. Потом, положив в вектор открытых вершин первую исходную вершину, начинается цикл. В нем изначально происходит сортировка открытых вершин по значению эвристической функции, потом совершается проход по всем детям рассматриваемой вершины, и в случае, если ребенок вершины встречается впервые, то происходит смена информации его эвристической функции, длине пути, информации о вершине родителя, а также совершается добавление ребенка в открытые вершины. Также смена информации происходит в случае, если для ребенка вершины значение кратчайшего пути меньше, чем текущее. Цикл продолжается до тех пор, пока вектор открытых вершин будет не пуст, либо пока искомая вершина не будет найдена до окончания открытой очереди. Далее, когда функция доходит до искомой конечной вершины, происходит вызов функции для вывода ответа.

**Описание дополнительных функций.**

- bool comp(Top a, Top b) – функция принимает 2 переменные типа Top, используется для сортировки открытых вершин по значению эвристической функции.
- void answer(std::vector<Top>& vectorTops, char startTop, char endTop) – функция принимает вектор вершин, а также начальную и искомую вершины. Данная функция используется для вывода ответа.
- void changeInfo(std::vector<Top>& vectorTops, std::vector<Top>& openVertexes, char a, char name, double temp_G, char endTop ) – функция принимает вектор вершин, вектор открытых вершин, имя ребенка вершины, которую рассматриваем, имя самой вершины, значение длины пути до него и имя конечной вершины, для вычисления эвристической оценки. Данная функция используется для смены информации о вершине и ребенке вершины.

- int whatNumber(char a, std::vector<Top>& vectorTops) – функция принимает имя вершины и вектор вершин. Данная функция ищет вершину в векторе и возвращает её индекс.
- bool check(std::vector<Path>& vectorPath, std::vector<Top>& vectorTops , char endTop, bool flagM, bool flagAd) – функция принимает вектор пути, вектор вершин и имя искомой вершины. Данная функции сначала выполняет проверку эвристики на монотонность, по свойству монотонности: Эвристическая функция h(v) называется монотонной, если для любой вершины v1 и её потомка v2 разность h(v1) и h(v2) не превышает фактического веса ребра c(v1,v2) от v1 до v2, а эвристическая оценка целевого состояния равна нулю. Далее, в случае, когда она не монотонна, функция выполняет проверку эвристики на допустимость по следующему свойству: Говорят, что эвристическая оценка h(v) **допустима**, если для любой вершины v значение h(v) меньше или равно весу кратчайшего пути от v до цели. Данная проверка происходит в случае, когда монотонность не выполнилась, так как существует теорема:
  Любая монотонная эвристика допустима, однако обратное неверно.

**Сложность алгоритма по памяти.**

Сложность для алгоритма A* оценивается как O(|V+E|), так как программе приходится хранить граф целиком.

**Сложность алгоритма по времени.**

Временная сложность алгоритма A* зависит от эвристики. В данном случае, N·log(N) - сложность сортировки, где N длина очереди в данный момент тогда ответ O(1 + N*log(N)(V+E)), единицу можно убрать и получим O(N*log(N)(V+E)).

**Тестирование жадного алгоритма.**

Входные данные:

a e

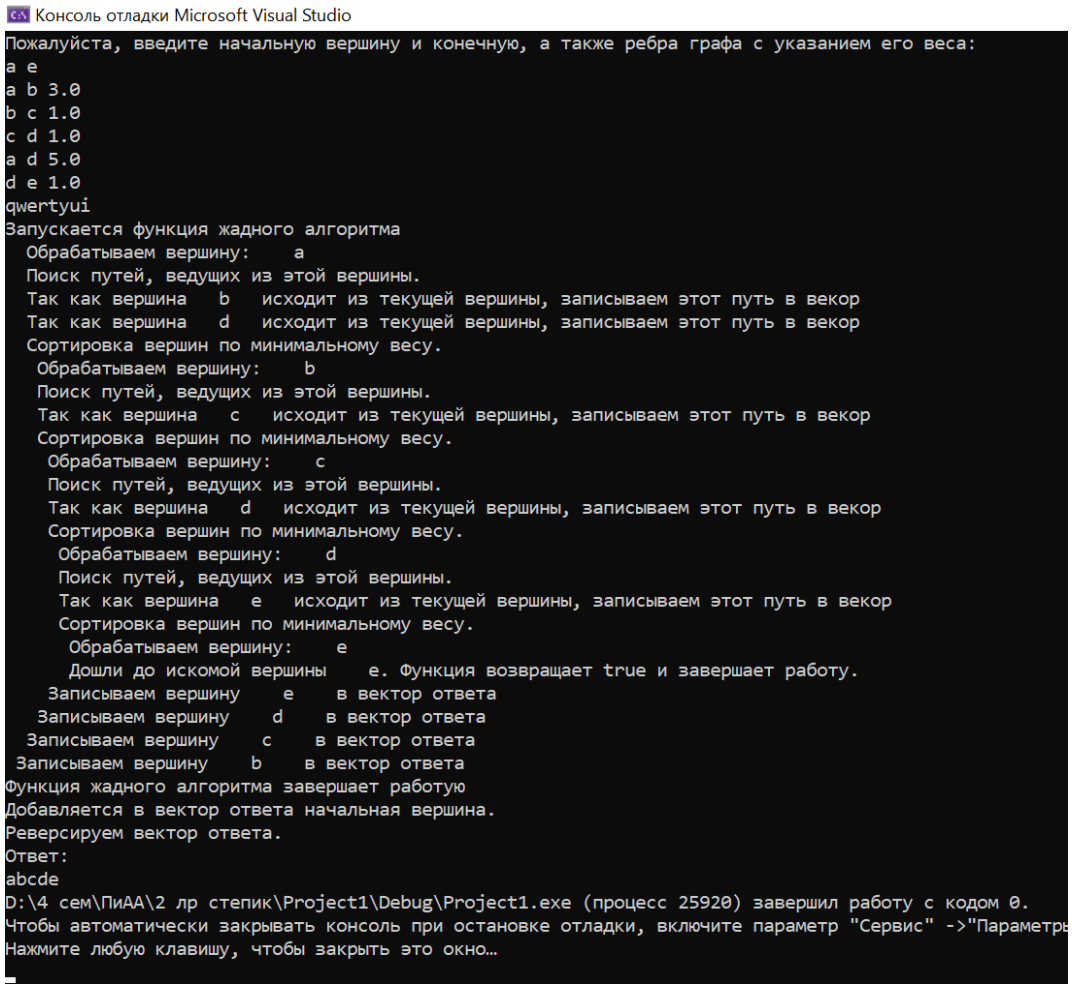
a b 3.0

b c 1.0

c d 1.0

a d 5.0

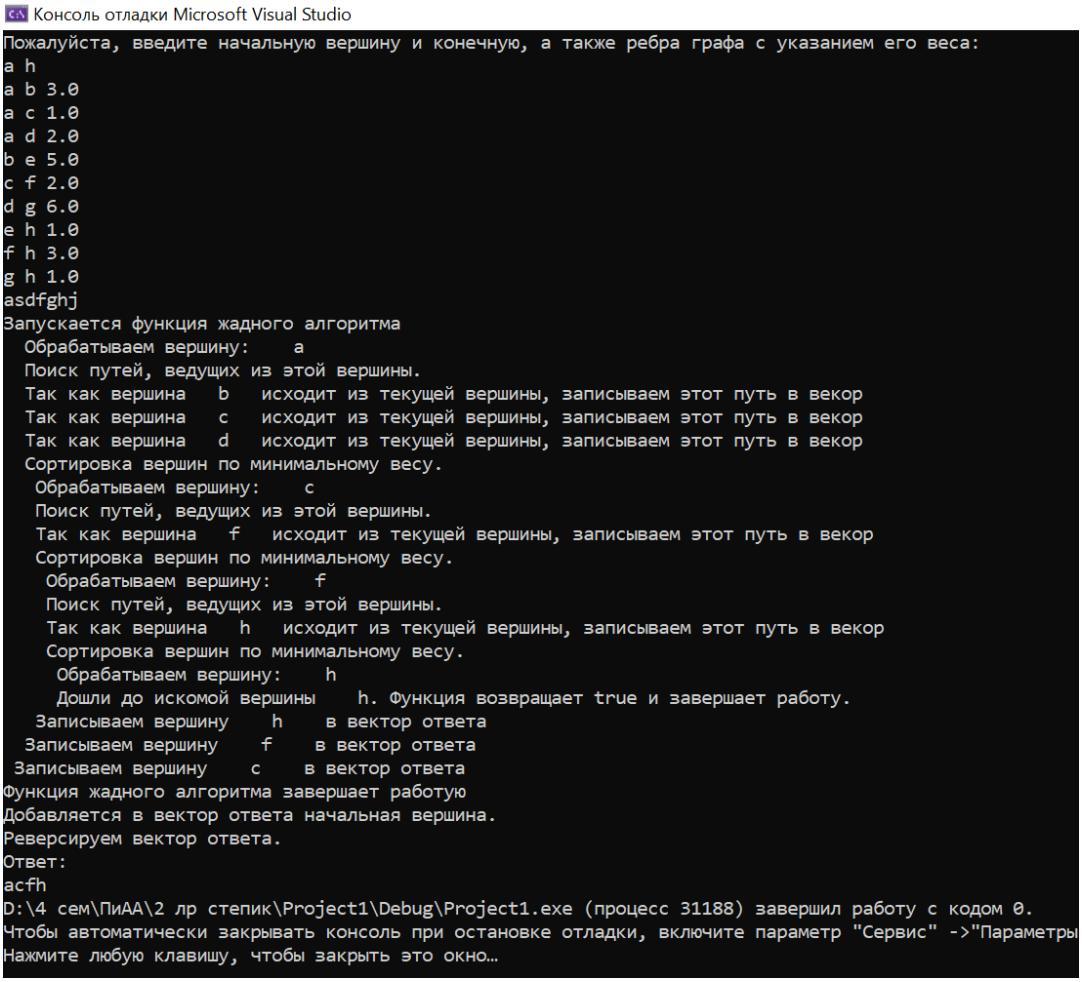
d e 1.0




Входные данные:

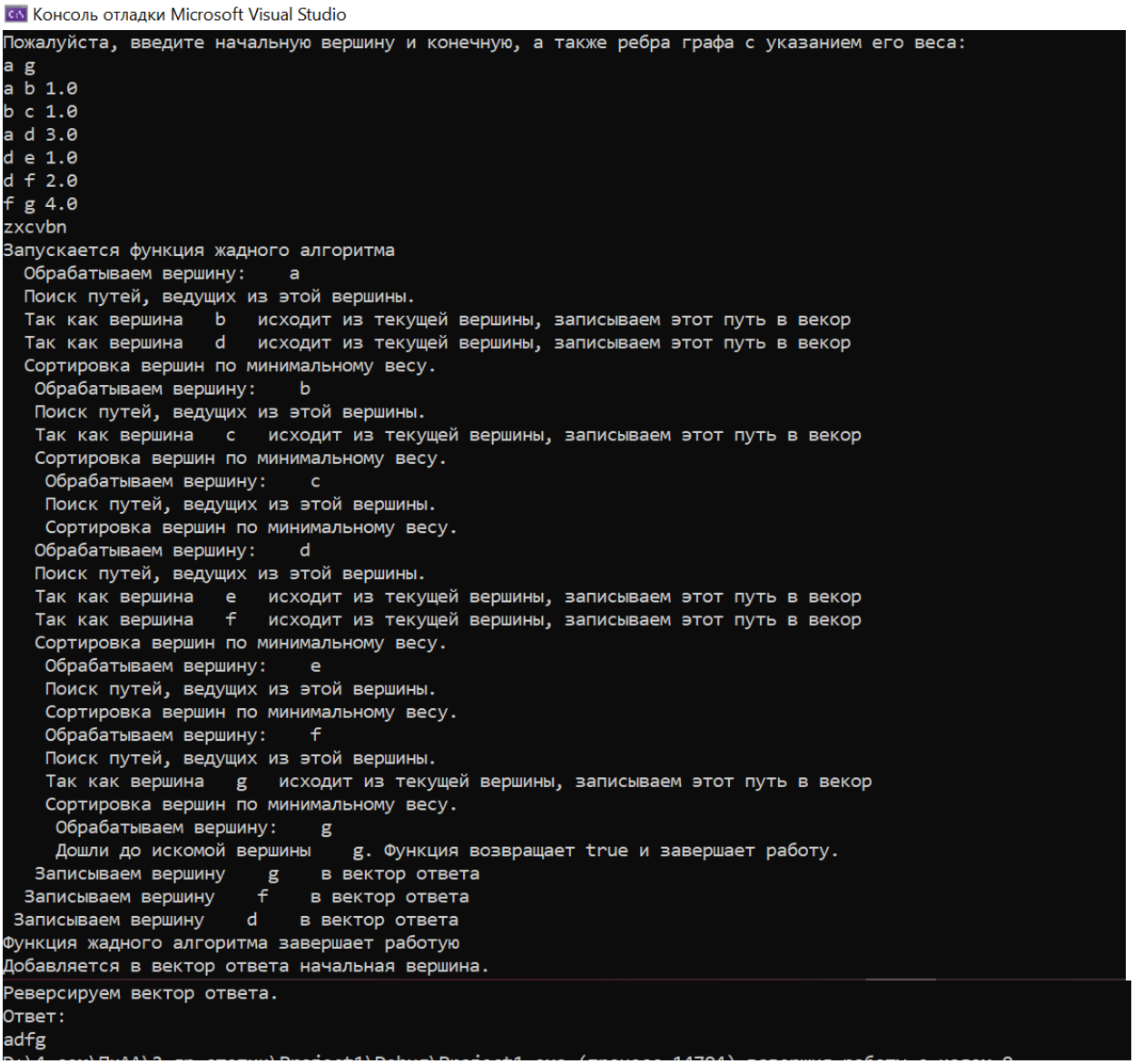
a h
a b 3.0
a c 1.0
a d 2.0
b e 5.0
c f 2.0
d g 6.0
e h 1.0

f h 3.0
g h 1.0



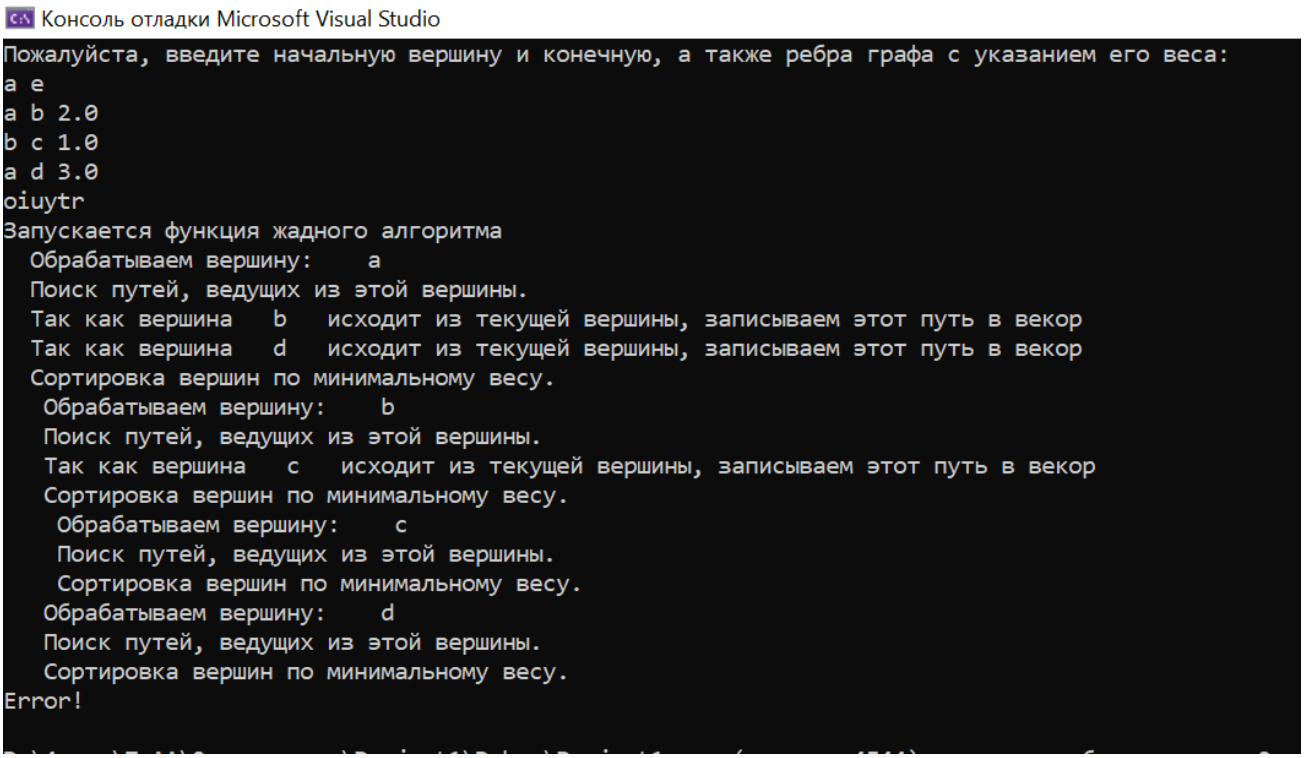

Входные данные:

a g
a b 1.0
b c 1.0
a d 3.0
d e 1.0
d f 2.0
f g 4.0




Входные данные:

a e
a b 2.0
b c 1.0
a d 3.0




**Тестирование алгоритма A*.**

Входные данные:

a e

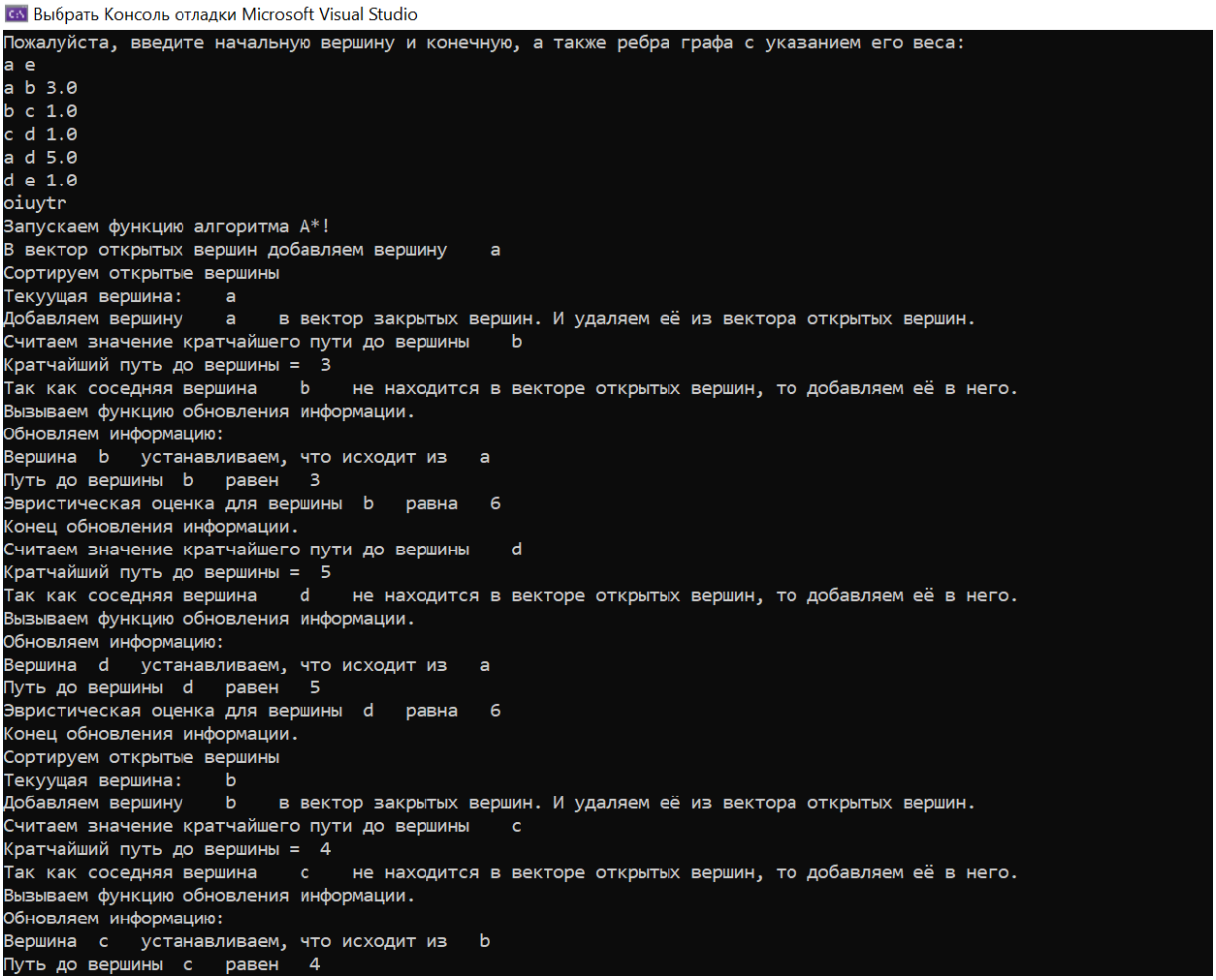
a b 3.0

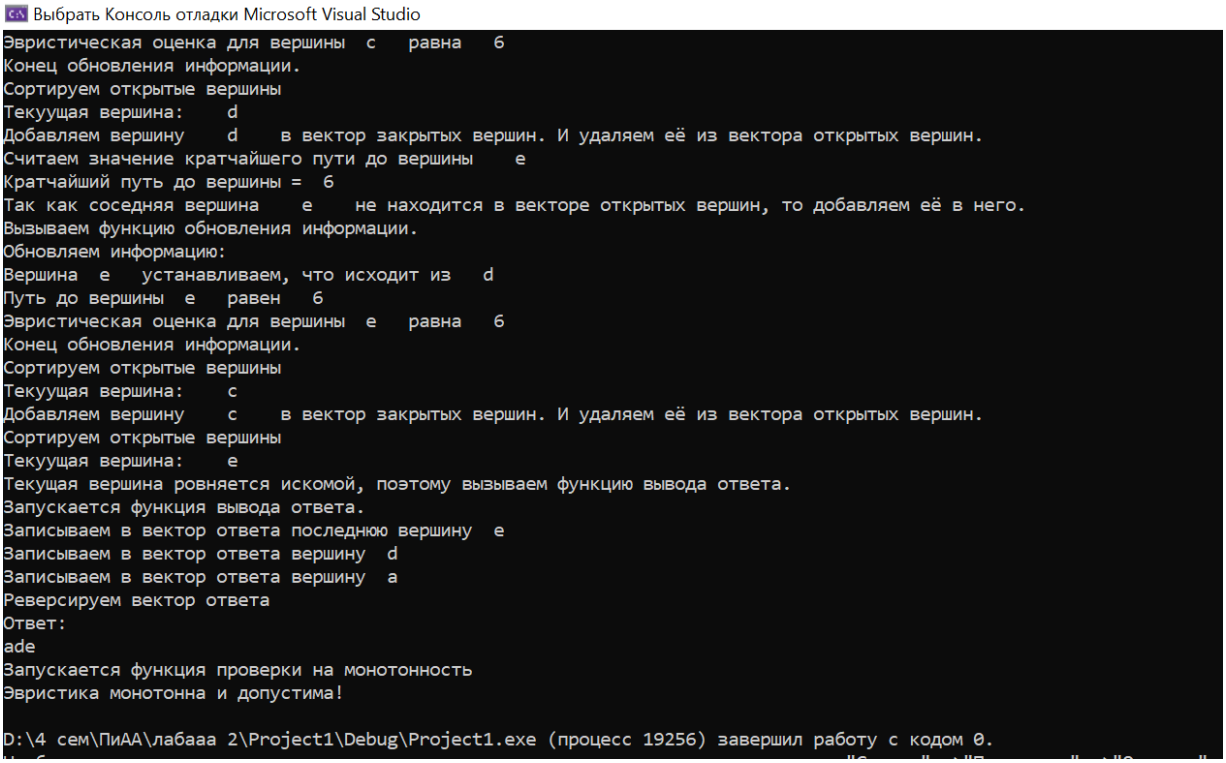
b c 1.0

c d 1.0

a d 5.0

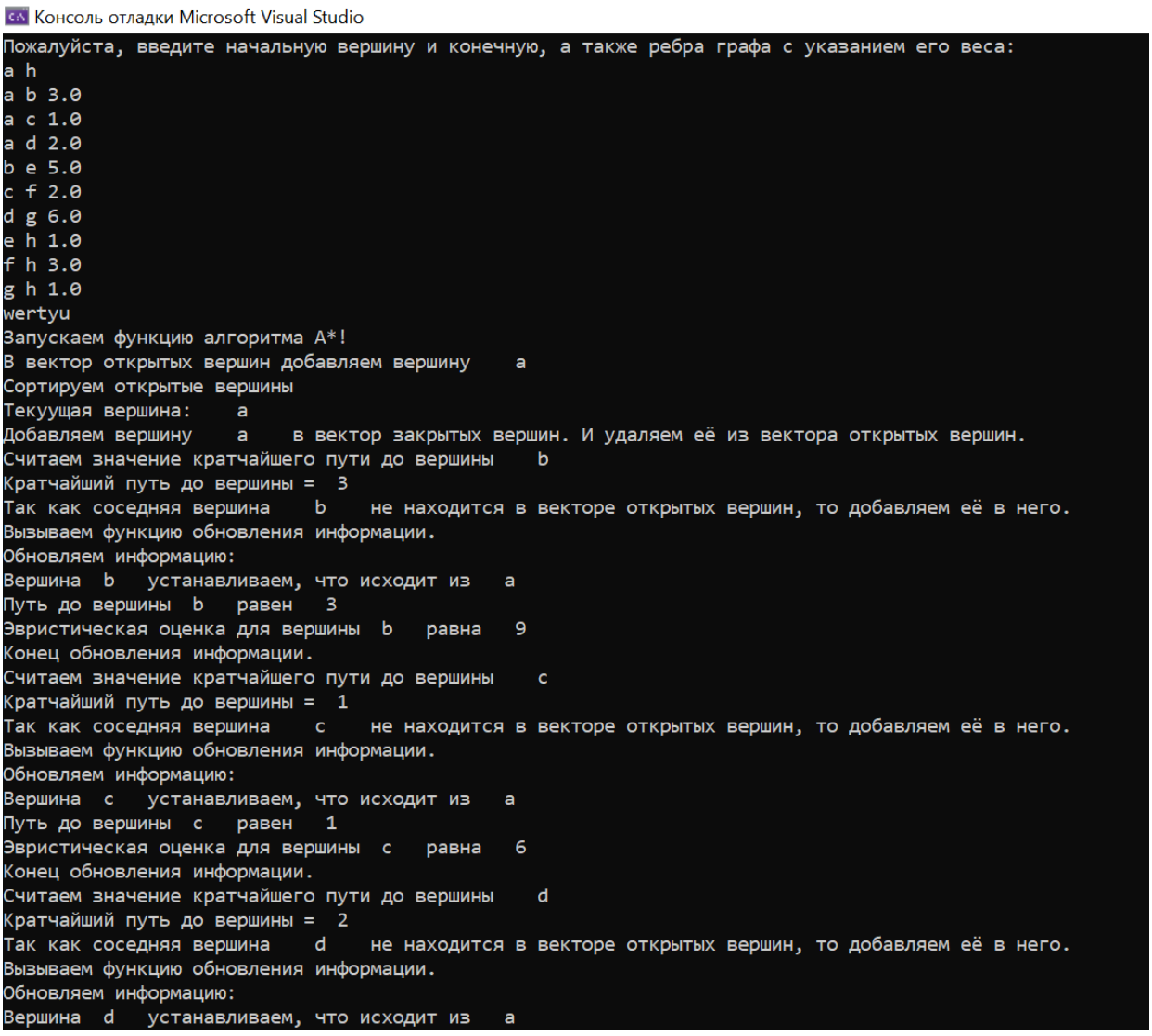
d e 1.0

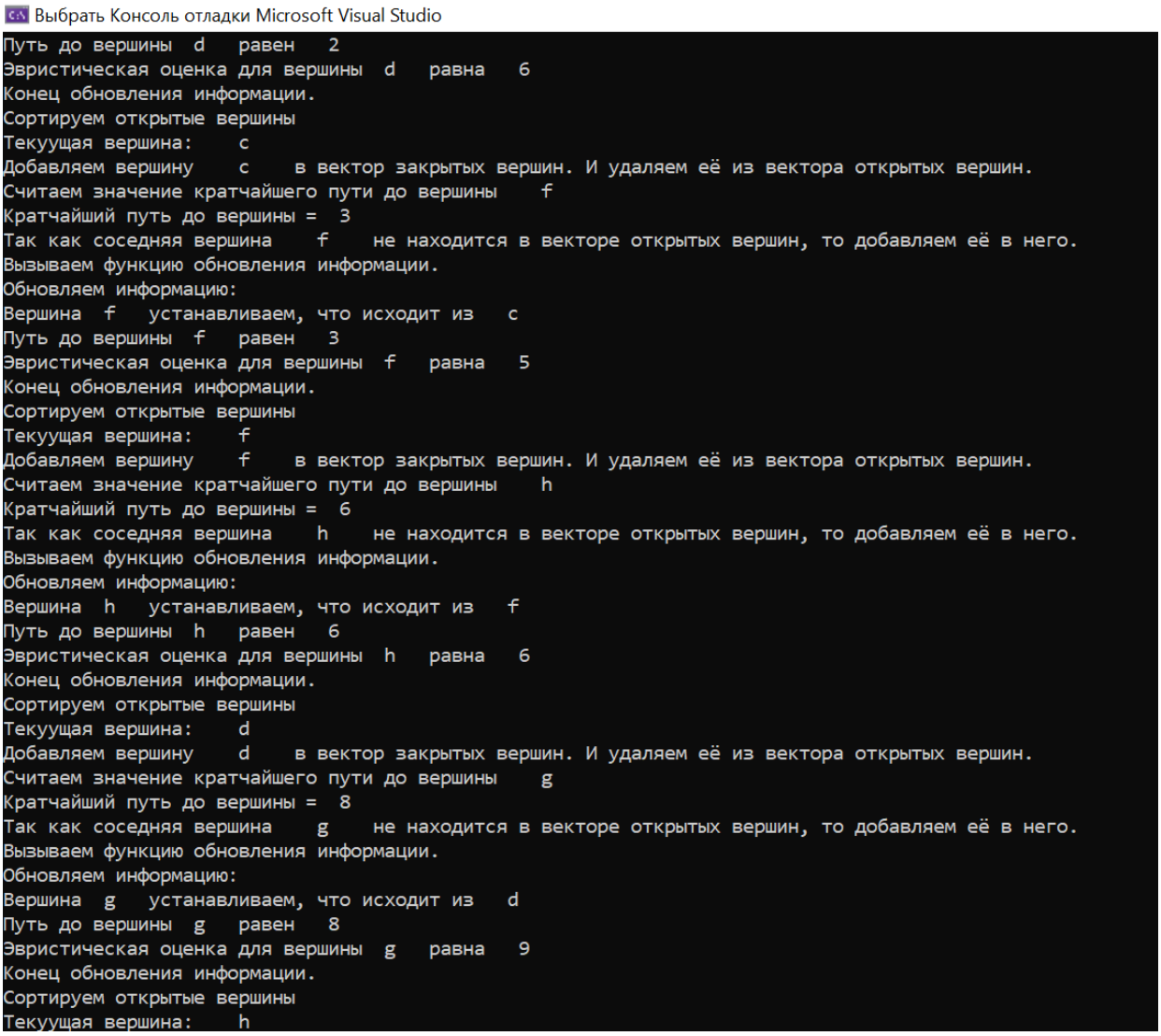


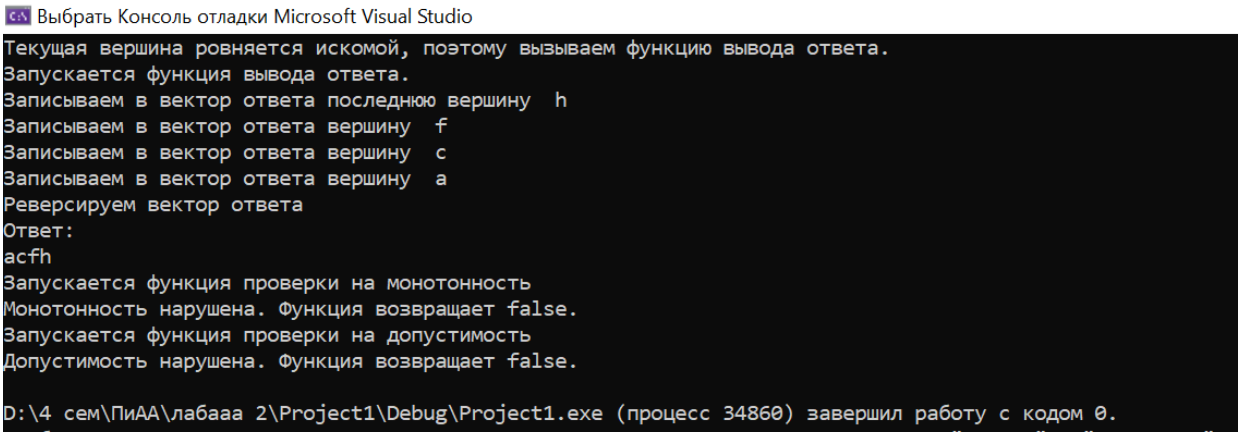




Входные данные:

a h
a b 3.0
a c 1.0
a d 2.0
b e 5.0
c f 2.0
d g 6.0
e h 1.0
f h 3.0
g h 1.0



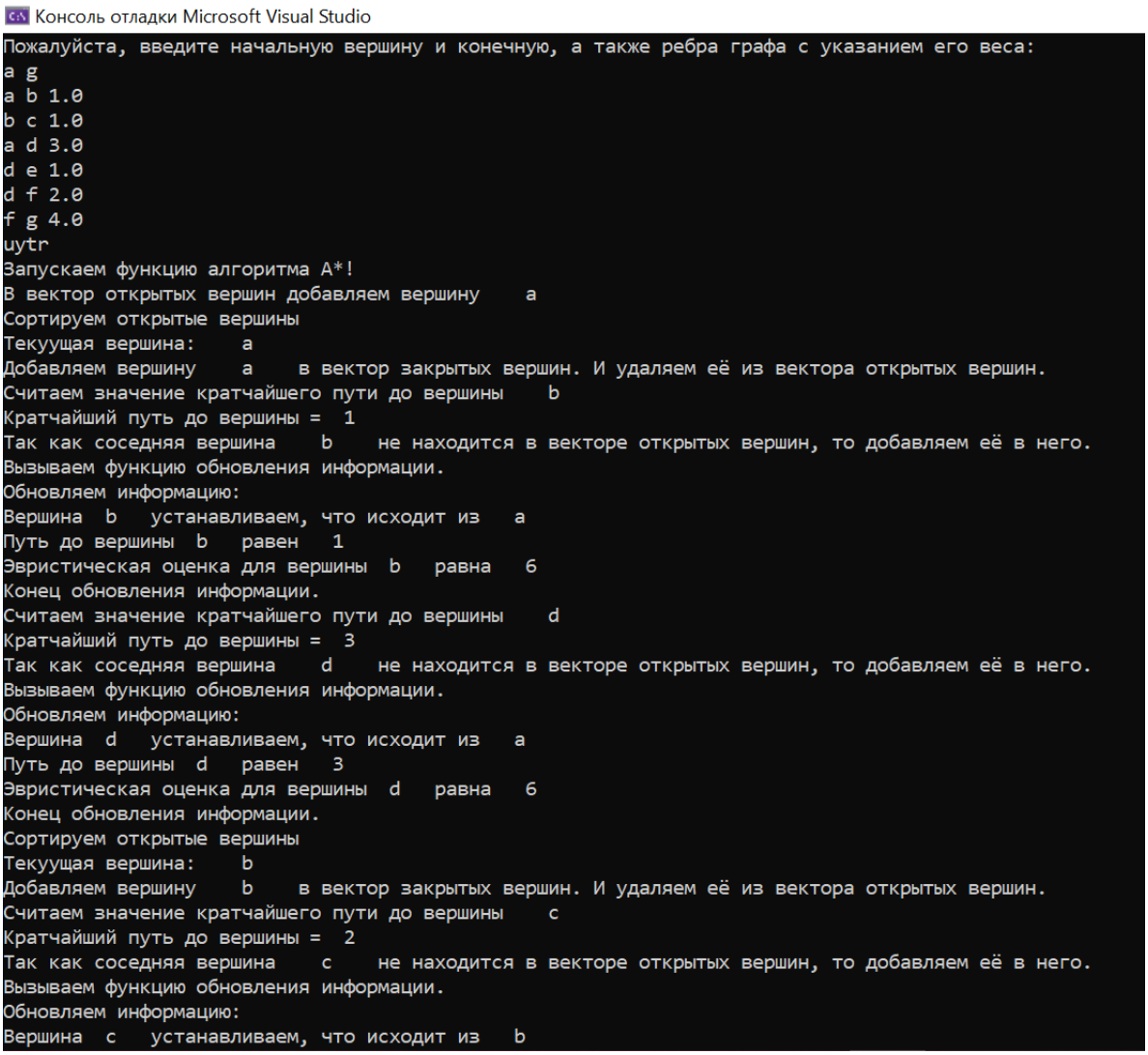


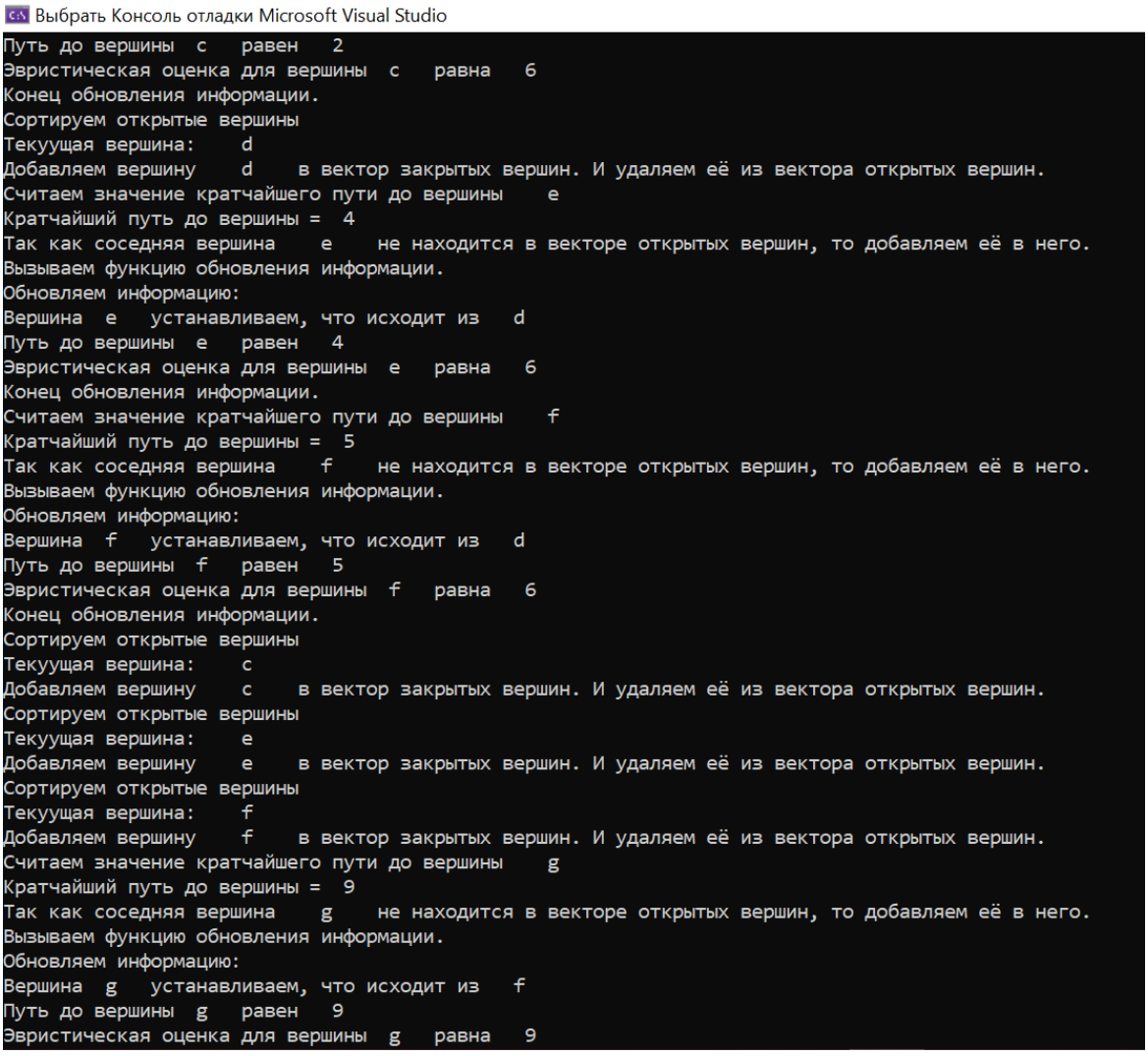



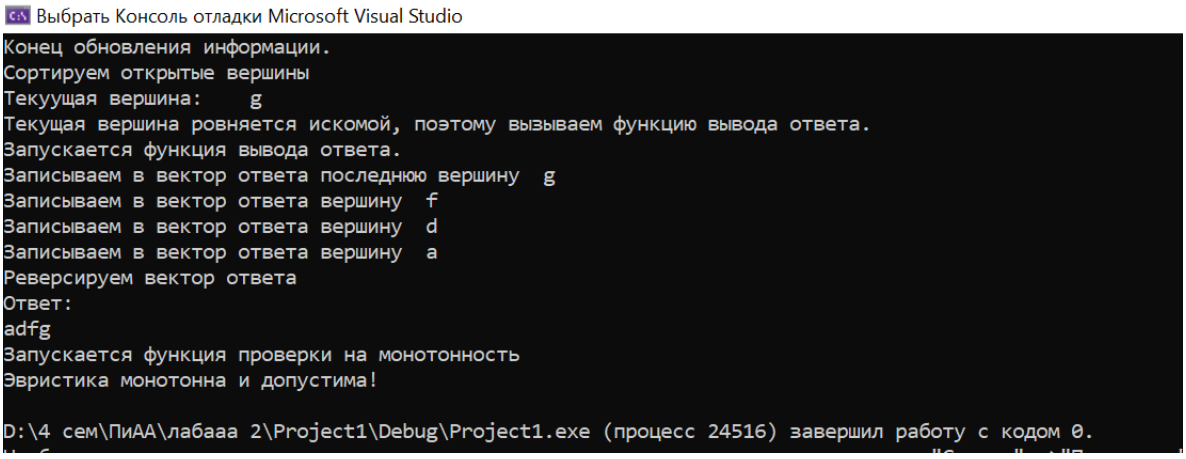

Входные данные:

a g
a b 1.0
b c 1.0
a d 3.0
d e 1.0

d f 2.0
f g 4.0

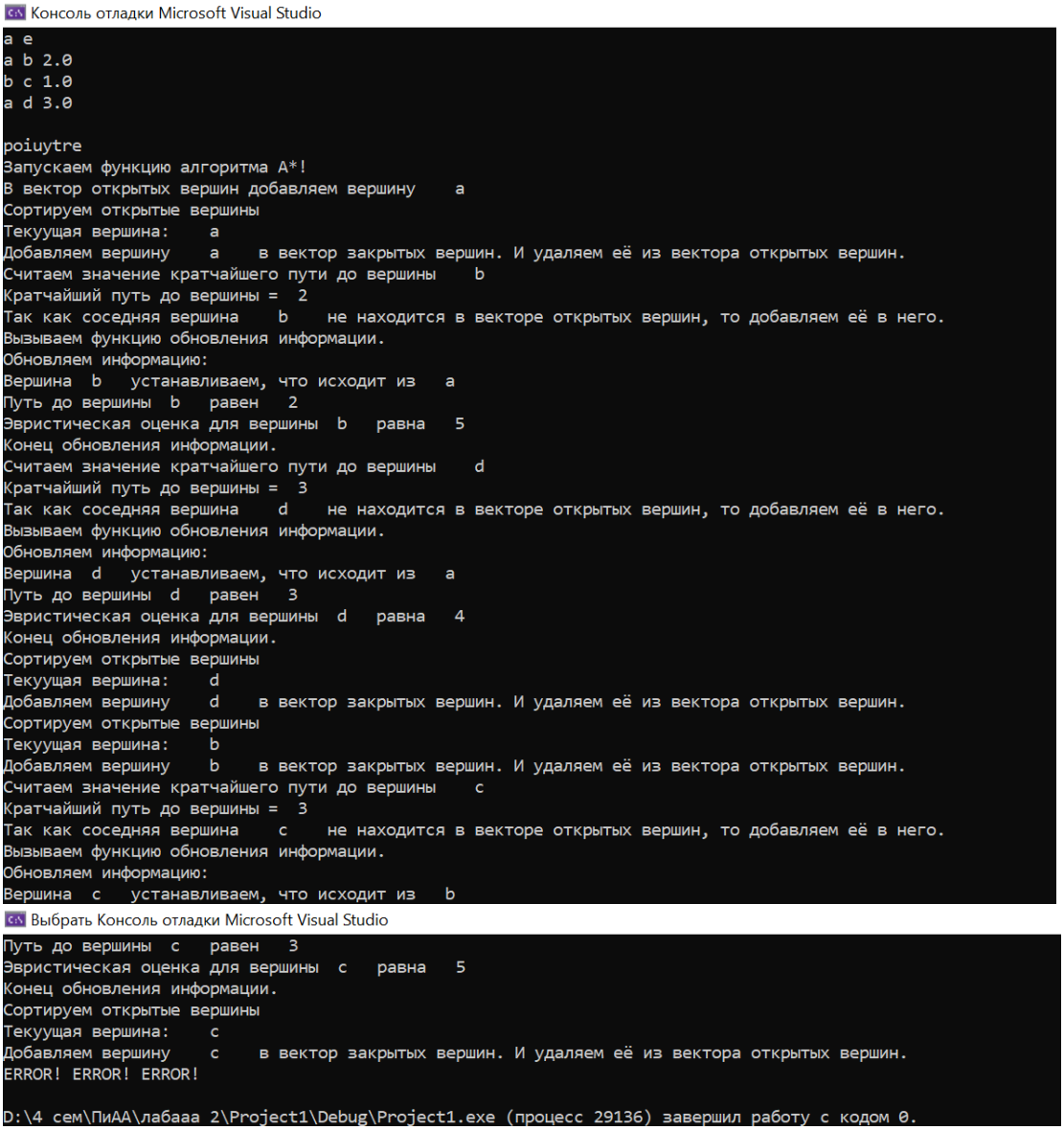

Консоль отладки Microsoft Visual Studio
Пожалуйста, введите начальную вершину и конечную, а также ребра графа с указанием его веса:
a g
a b 1.0
b c 1.0
a d 3.0
d e 1.0
d f 2.0
f g 4.0
uytr
Запускаем функцию алгоритма A*!
В вектор открытых вершин добавляем вершину a
Сортируем открытые вершины
Текуущая вершина: a
Добавляем вершину a в вектор закрытых вершин. И удаляем её из вектора открытых вершин.
Считаем значение кратчайшего пути до вершины b
Кратчайший путь до вершины = 1
Так как соседняя вершина b не находится в векторе открытых вершин, то добавляем её в него.
Вызываем функцию обновления информации.
Обновляем информацию:
Вершина b устанавливаем, что исходит из a
Путь до вершины b равен 1
Эвристическая оценка для вершины b равна 6
Конец обновления информации.
Считаем значение кратчайшего пути до вершины d
Кратчайший путь до вершины = 3
Так как соседняя вершина d не находится в векторе открытых вершин, то добавляем её в него.
Вызываем функцию обновления информации.
Обновляем информацию:
Вершина d устанавливаем, что исходит из a
Путь до вершины d равен 3
Эвристическая оценка для вершины d равна 6
Конец обновления информации.
Сортируем открытые вершины
Текуущая вершина: b
Добавляем вершину b в вектор закрытых вершин. И удаляем её из вектора открытых вершин.
Считаем значение кратчайшего пути до вершины c
Кратчайший путь до вершины = 2
Так как соседняя вершина c не находится в векторе открытых вершин, то добавляем её в него.
Вызываем функцию обновления информации.
Обновляем информацию:
Вершина c устанавливаем, что исходит из b




Входные данные:

a e
a b 2.0
b c 1.0
a d 3.0




**Вывод.**

В результате работы была написана полностью рабочая программа, выполняющая поставленную задачу.

**Приложение.**

**Файл Source.cpp(жадный алгоритм):**

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>      // std::setw
```

```
class Path {
private:
    char nameFrom;
    char nameOut;
    double weightPath;

public:
    Path(char nameFrom, char nameOut, double weightPath)
        : nameFrom(nameFrom), nameOut(nameOut), weightPath(weightPath) {}

    char getNameFrom() const {
        return nameFrom;
    }

    char getNameOut() const {
        return nameOut;
    }

    double getWeightPath() const {
        return weightPath;
    }
};

bool comp(Path a, Path b) {//компаратор
    return a.getWeightPath() < b.getWeightPath();
}

bool func(std::vector<Path>* vector, char curChar, char endChar, std::vector<char>* answer, int
depth) {
    depth++;

    std::cout << setw(depth + 1) << ' ' << "Обрабатываем вершину:    " << curChar << std::endl;

    if (curChar == endChar) {    //выход из рекурсии
        std::cout << setw(depth + 1) << ' ' << "Дошли до искомой вершины    " << endChar << ".
Функция возвращает true и завершает работу." << std::endl;
        return true;
    }

    //curChar - текущая вершина

    std::cout << setw(depth + 1) << ' ' << "Поиск путей, ведущих из этой вершины." <<
std::endl;
    std::vector<Path> temporaryVector;
    temporaryVector.reserve(0);//говоррим что пустой
    for (Path path : *vector) {// БУДУТ ПРОЙДЕНЫ ВСЕ ВЕРШИНЫ В ВЕКТОРЕ
        if (path.getNameFrom() == curChar) {//отбирает все пути из нужной вершины
            std::cout << setw(depth + 1) << ' ' << "Так как вершина   " << path.getNameOut() << "
исходит из текущей вершины, записываем этот путь в векор" << std::endl;
            temporaryVector.emplace_back(path);//записывается в вектор
        }
```

```
    }

    //т к нужен самый дешевый путь то соитируем

    std::cout << setw(depth + 1) << ' ' << "Сортировка вершин по минимальному весу." <<
std::endl;

    std::sort(temporaryVector.begin(), temporaryVector.end(), comp);


    for (Path path : temporaryVector) {//проходимся по всем вершинам
        if (func(vector, path.getNameOut(), endChar, answer, depth)) {//новая переменная - b
            depth--;
            std::cout << setw(depth + 1) << ' ' << "Записываем вершину    " << path.getNameOut()
<<    "    в вектор ответа" << std::endl;
            answer->emplace_back(path.getNameOut());
            return true;
        }
    }

    return false;
}

int main() {
    setlocale(LC_ALL, "rus");
    int depth = 0;
    int flag = 1;

    std::vector<Path> vector;
    vector.reserve(0);

    std::vector<char> answer;
    answer.reserve(0);

    char startChar;
    char endChar;
    std::cout << "Пожалуйста, введите начальную вершину и конечную, а также ребра графа
с указанием его веса: " << std::endl;

    std::cin >> startChar;
    std::cin >> endChar;


    char start, end;
    double weight;


    while (std::cin >> start >> end >> weight) {
        vector.emplace_back(Path(start, end, weight));
    }

    std::cout << "Запускается функция жадного алгоритма" << std::endl;
```

```
    if (!func(&vector, startChar, endChar, &answer, depth))
    {
        std::cout << "Error!" << std::endl;
        flag = 0;
    }

    if (flag)
    {
        std::cout << "Функция жадного алгоритма завершает работую" << std::endl;
        std::cout << "Добавляется в вектор ответа начальная вершина." << std::endl;
        answer.emplace_back(startChar);
        std::cout << "Реверсируем вектор ответа." << std::endl;
        std::reverse(answer.begin(), answer.end());

        std::cout << "Ответ:    " << std::endl;
        for (char sym : answer) {
            std::cout << sym;
        }
    }


    return 0;
          }
```

### Файл Source.cpp(алгоритм A*):

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>

class Top {//класс вершины

public:

    char name;
    double pathToTop;//путь до текущей вершины - g
    double heuristicF;//эвристическая функция - f
    char nameFromT;
    std::vector<char> coupled;//вектор для исходящих из вершины вершин


    Top(char name)//конструктор1 - необходим для заполнения начальной вершины
        : name(name) {
        heuristicF = 0;
        pathToTop = -1;//будет использоваться для необработанных вершин вместо знака
бесконечно
        nameFromT = '-';
    }

    Top() {                         //конструктор2
```

```
        name = '!';                    //
        heuristicF = 0;
        pathToTop = -1; //
        nameFromT = '-';
    }

};

class Path {//класс пути - хранит только путь: откуда куда и сколько весит путь

public:

    char nameFromP;
    char nameOutP;
    double weightPath;


    Path(char nameFromP, char nameOutP, double weightPath)
        : nameFromP(nameFromP), nameOutP(nameOutP), weightPath(weightPath) {}



    char getNameFromP() const {
        return nameFromP;
    }

    char getNameOutP() const {
        return nameOutP;
    }

    double getWeightPath() const {
        return weightPath;
    }
};

bool check(std::vector<Path>& vectorPath, std::vector<Top>& vectorTops, char endTop, bool
flagM, bool flagAd)
{
    std::cout << "Запускается функция проверки на монотонность и допустимость  " <<
std::endl;
    if (abs(endTop - endTop) != 0) {
        std::cout << "Эвристическая оценка целевого состояния не равна нулю!" << std::endl;
        flagM = false;

    }

    for (unsigned int i = 0; i < vectorPath.size(); i++) {

        if ((abs(endTop - vectorPath[i].nameFromP) - abs(endTop - vectorPath[i].nameOutP)) >
vectorPath[i].weightPath) {
            std::cout << "Монотонность нарушена. Функция возвращает false." << std::endl;
            flagM = false;
```

```
        }
    }
    //поверка на допустимость
    if (!flagM)
    {
        for (unsigned int i = 0; i < vectorTops.size(); i++) {


            if ((abs(endTop - vectorTops[i].name) > (vectorTops[vectorTops.size() - 1].pathToTop -
vectorTops[i].pathToTop)))
            {
                std::cout << "Допустимость нарушена. Функция возвращает false." << std::endl;
                flagAd = false;
            }



        }
    }
    if (flagM)
    {
        std::cout << "Эвристика монотонна и допустима!" << std::endl;
        return true;
     }
    else if (!flagM && flagAd)
    {
        std::cout << "Эвристика допустима!" << std::endl;
        return true;
    }
    else
    {
        std::cout << "Эвристика не монотонна и не допустима!" << std::endl;
        return false;
    }

}




int whatNumber(char a, std::vector<Top>& vectorTops) {

    for (unsigned int i = 0; i < vectorTops.size(); i++) {
        if (vectorTops[i].name == a) {
            return i;
        }
    }
    return -1;
}

bool comp(Top a, Top b) {//компаратор, используется для сортировки в открытом списке
```

```
    return a.heuristicF < b.heuristicF;
}

void answer(std::vector<Top>& vectorTops, char startTop, char endTop)
{
    std::cout << "Запускается функция вывода ответа.  " << std::endl;
    std::vector<Top> answer;
    answer.reserve(0);
    Top temp = vectorTops[whatNumber(endTop, vectorTops)];
    std::cout << "Записываем в вектор ответа последнюю вершину  " << endTop << std::endl;
    answer.emplace_back(temp);
    while (temp.name != startTop) {
        temp = vectorTops[whatNumber(temp.nameFromT, vectorTops)];
        std::cout << "Записываем в вектор ответа вершину  " << temp.name << std::endl;
        answer.emplace_back(temp);
    }
    std::cout << "Реверсируем вектор ответа  " << std::endl;
    std::reverse(answer.begin(), answer.end());//т к он заполнился в обратном порядке, делаем
reverse
    std::cout << "Ответ:  " << std::endl;
    for (Top ans : answer) {
        std::cout << ans.name;
    }
    std::cout << std::endl;
}

void changeInfo(std::vector<Top>& vectorTops, std::vector<Top>& openVertexes, char a, char
name, double temp_G, char endTop )
{
    std::cout << "Обновляем информацию:  " << std::endl;

    vectorTops[whatNumber(a, vectorTops)].nameFromT = name;

    vectorTops[whatNumber(a, vectorTops)].pathToTop = temp_G;
    openVertexes[whatNumber(a, openVertexes)].nameFromT = name;
    openVertexes[whatNumber(a, openVertexes)].pathToTop = temp_G;
    openVertexes[whatNumber(a, openVertexes)].heuristicF = temp_G + abs(endTop - a);
    std::cout << "Вершина  " << a << "   устанавливаем, что исходит из   " << name <<
std::endl;
    std::cout << "Путь до вершины  " << a << "   равен   " << temp_G << std::endl;
    std::cout << "Эвристическая оценка для вершины  " << a << "   равна   " << temp_G +
abs(endTop - a) << std::endl;
    std::cout << "Конец обновления информации.  " << std::endl;
}

bool A(std::vector<Path>& vectorPath, std::vector<Top>& vectorTops, char startTop, char
endTop) {

    Top temp;
    double temp_G;
    std::vector<Top> closedVertexes;
    closedVertexes.reserve(0);
```

```
    std::vector<Top> openVertexes;
    openVertexes.reserve(0);

    std::cout << "В вектор открытых вершин добавляем вершину    " << vectorTops[0].name
<< std::endl;

    openVertexes.emplace_back(vectorTops[0]);

    while (!openVertexes.empty()) {
        Top min = openVertexes[0];
        std::cout << "Сортируем открытые вершины " << std::endl;
        std::sort(openVertexes.begin(), openVertexes.end(), comp);
        temp = openVertexes[0]; //минимальная f из openVertexes
        std::cout << "Текуущая вершина:    "<< temp.name << std::endl;

        if (temp.name == endTop) {
            std::cout << "Текущая вершина ровняется искомой, поэтому вызываем функцию
вывода ответа." << std::endl;
            answer(vectorTops, startTop, endTop);
            return true;
        }
        std::cout << "Добавляем вершину    " << openVertexes[0].name << "    в вектор
закрытых вершин. И удаляем её из вектора открытых вершин." << std::endl;
        closedVertexes.emplace_back(temp); //добавляем обработанную вершину
        openVertexes.erase(openVertexes.begin()); //удаляем обработанную вершину

        for (unsigned int i = 0; i < temp.coupled.size(); i++) { //для каждого соседа
            if (whatNumber(temp.coupled[i], closedVertexes) != -1) { //если сосед находится В
closedVertexes (уже обработанном)
                continue;
            }
            int j = 0;
            while (true) {
                if (vectorPath[j].nameFromP == temp.name && vectorPath[j].nameOutP ==
temp.coupled[i]) {
                    std::cout << "Считаем значение кратчайшего пути до вершины    " <<
vectorPath[j].nameOutP << std::endl;
                    temp_G = vectorPath[j].weightPath + temp.pathToTop;
                    std::cout << "Кратчайший путь до вершины =  " << temp_G << std::endl;
                    break;
                }
                j++;
            }

            if (whatNumber(temp.coupled[i], openVertexes) == -1) { //если сосед не в openVertexes
                std::cout << "Так как соседняя вершина    " << temp.coupled[i] << "    не находится
в векторе открытых вершин, то добавляем её в него." << std::endl;
                openVertexes.emplace_back(vectorTops[whatNumber(temp.coupled[i], vectorTops)]);
//добавляем соседа
                std::cout << "Вызываем функцию обновления информации."  << std::endl;
                changeInfo(vectorTops, openVertexes, temp.coupled[i], temp.name, temp_G,
endTop);
```

```
            }
            else {
                if (temp_G < openVertexes[whatNumber(temp.coupled[i], openVertexes)].pathToTop)
{
                    std::cout << "Так как найден более короткий путь("   << temp_G <<   ") до
вершины    " << temp.coupled[i] <<".  Обновляем информацию."<< std::endl;
                    changeInfo(vectorTops, openVertexes, temp.coupled[i], temp.name, temp_G,
endTop);
                }
            }


        }
    }
    return false;
}


int main() {

    setlocale(LC_ALL, "Russian");
    bool flag = true;
    bool flagM = true;
    bool flagAd = true;
    std::vector<Path> vectorPath;//вектор путей
    vectorPath.reserve(0);
    std::vector<Top> vectorTops;//вектор вершин
    vectorTops.reserve(0);

    char startTop;
    char endTop;
    std::cout << "Пожалуйста, введите начальную вершину и конечную, а также ребра графа
с указанием его веса: " << std::endl;

    std::cin >> startTop;
    std::cin >> endTop;

    char start, end;
    double weight;


    while (std::cin >> start >> end >> weight) {
        vectorPath.emplace_back(Path(start, end, weight));
    }

    std::set<char> set;//

    set.insert(startTop);//вставляем первую вершину

    vectorTops.emplace_back(Top(startTop));//создаем вершину начальную и кладем в вектор

    int number;
```

```
    for (Path path : vectorPath) {//проход по вектору путей
        char from = path.getNameFromP();//
        char out = path.getNameOutP();

        if (set.find(from) == set.end()) {//проверяет что в множестве set нет from
            set.insert(from);
            vectorTops.emplace_back(Top(from));
        }
        if (set.find(out) == set.end()) {
            set.insert(out);
            vectorTops.emplace_back(Top(out));

        }
    }
    //вектор путей заполнен, но вектор соседей нет =>
    //выполняем проход по вектору путей снова
    for (Path path : vectorPath) {//проход по вектору путей
        char from = path.getNameFromP();//
        char out = path.getNameOutP();

        if (set.find(from) != set.end()) {//проверяет что в множестве set есть from
            number = whatNumber(from, vectorTops);
            vectorTops[number].coupled.emplace_back(out);//добавляем соседа вершины
        }

    }

    vectorTops[0].pathToTop = 0;
    vectorTops[0].heuristicF = abs(endTop - startTop);
    std::cout << "Запускаем функцию алгоритма A*!" << std::endl;

    if (!A(vectorPath, vectorTops, startTop, endTop)) {
        flag = false;
        std::cout << "ERROR! ERROR! ERROR!" << std::endl;
    }

    //chekAd(vectorPath, vectorTops, endTop);
    if (flag)
    {
        check(vectorPath, vectorTops, endTop, flagM, flagAd);
    }

}
```