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 поиска пути в графах

Студентка гр. 9382 Пя С.

Преподаватель Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Реализовать программу, занимающуюся поиском пути в графе с помощью Жадного и A* алгоритмов, и изучить данные алгоритмы.

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

Вар. 2. В A* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных.

Описание алгоритма

Жадный алгоритм

Список ребер сортируется по начальной вершине (в алфавитном порядке ради простоты, чтобы одинаковые вершины находились вместе), при равенстве начальных вершин сортировка происходит по весу. Проходим по графу, начиная с начальной заданной вершины. Находим ребра, в которых в качестве начальной вершины выступает рассматриваемая, и выбираем первое найденное, так как оно

будет иметь меньший вес. Конечная вершина этого ребра становится следующей рассматриваемой вершиной. В случае отсутствия следующей вершины переходим к предыдущей. Просмотр графа заканчивается тогда, когда рассматриваемой вершиной будет конечная заданная.

Сложность по времени — O (NlogN + N*M).

Сортировка в среднем занимает O(NlogN) для N ребер. Рассмотрим худший случай при проходе всего графа. Для графа с M вершинами и N ребрами O(N*M).

Сложность по памяти — O(N+M).

Для хранения графа используется список ребер и список вершин. Тогда для графа с M вершинами и N ребрами O(N+M).

Алгоритм A*

Для моего варианта эвристическая оценка для каждой вершины задается во входных данных. Начинаем проход по графу с начальной заданной вершины. Помечаем рассматриваемую вершину посещенной. Для смежных вершин определяем сумму весов ребер от начальной вершины до текущей, при нахождении меньшей суммы весов ребер, заменяем текущую. Ищем вершину с минимальной суммой веса от начальной вершины до текущей и эвристической оценки и определяем ее как следующую рассматриваемую вершину, при этом сохранив в качестве ссылки начальную вершину в ребре, содержащем конечную вершину – нашу следующую рассматриваемую. Алгоритм прекратится в случае равенства текущей рассматриваемой вершины и конечной заданной.

Сложность по времени в лучшем случае - O(N+M).

Случай, когда мы имеем наиболее подходящую эвристическую функцию, с помощью которой мы выбираем вершины, которые находятся в направлении конечной вершины с доступностью к ней. Для графа с M вершинами и N ребрами O(N+M).

Сложность по времени в худшем случае.

В худшем случае, когда мы имеем плохую эвристическую функцию, алгоритм будет определять путь в последний момент, при этом он будет

проходить по всем возможным путям. Значит, сложность по времени будет расти экспоненциально в сравнении с длиной оптимального пути.

Сложность по памяти в лучшем случае и в худшем — O(2*N+M).

Для хранения графа используется список ребер и список вершин. Однако в структуре вершины хранится предыдущая. Случай, когда будет хранится путь от начала до конца, поэтому O(2*N+M).

В худшем случае сложность памяти будет такой же, потому что не реализуется стек, в ходе работы после заполнения списка ребер и вершин элементы не удаляются.

Описание функций и структур данных.

Жадный алгоритм

1. Структура Point содержит название вершины ver и isVisited, определяющий, посещена ли вершина или нет.
2. Структура Edge содержит название начальной и конечной вершин initVer и finVer, вес ребра weightVer.
3. Функция compForSort

Сигнатура: bool compForSort (const Edge& first, const Edge& second)

Назначение: предназначена для сортировки ребер.

Описание аргументов: first : const Edge и second : const Edge - ребра для сравнения.

Возвращаемое значение: true – если первое ребро остается на первом месте, иначе второе ребро является первым.

4. Класс PathSearching содержит поля initDestVer, finDestVer, хранящие начальную и конечную заданные вершины, string answer, который используется для хранения решения задачи. list<Edge> listOfEdge и list<Point*> listOfVer используются в качестве списков ребер и вершин.
5. std::list<Edge> listOfEdge – структура данных, отвечающая за хранение графа, хранит список ребер.
6. std::list<Point*> listOfVer – структура данных, список вершин.

Методы класса PathSearching:

7. Сигнатура: void readEdgeAndVer()

Назначение: ввод ребер и формирование списков вершин и ребер.

Описание аргументов: -

Возвращаемое значение: -

4. Сигнатура: void showAnswer()

Назначение: выводит результат работы алгоритма.

Описание аргументов: -

Возвращаемое значение: -

5. Сигнатура: void showListOfEdge()

Назначение: выводит отсортированный список ребер.

Описание аргументов: -

Возвращаемое значение: -

6. Сигнатура: Point* findVer(Vertical vertical)

Назначение: возвращает экземпляр структуры вершины по названию вершины.

Описание аргументов: vertical : Vertical – название вершины

Возвращаемое значение: Point* - экземпляр стуктуры вершины.

7. Сигнатура: void doGreedyAlgoritm()

Назначение: выполняет построение пути с помощью жадного алгоритма.

Описание аргументов: -

Возвращаемое значение: -

Алгоритм A*

1. Структура Point содержит название вершины ver и isVisited, определяющий, посещена ли вершина или нет. Wei, heurEst - эвристическая оценка и вес от начальной вершины до текущей, parent - начальная вершина в ребре.
2. Структура Edge содержит название начальной и конечной вершин initVer и finVer, вес ребра weightVer.

3. Класс PathSearching содержит поля initDestVer, finDestVer, хранящие начальную и конечную заданные вершины, list<Edge> listOfEdge и list<Point*> listOfVer используются в качестве списков ребер и вершин.
4. std::list<Edge> listOfEdge – структура данных, отвечающая за хранение графа, хранит список ребер.
5. std::list<Point*> listOfVer – структура данных, список вершин.

Методы класса PathSearching:

6. Сигнатура: void readEdgeAndVer()

Назначение: ввод ребер и формирование списков вершин и ребер. Ввод эвристической функции для каждой вершины.

Описание аргументов: -

Возвращаемое значение: -

7. Сигнатура: void showAnswer()

Назначение: выводит результат работы алгоритма.

Описание аргументов: -

Возвращаемое значение: -

8. Сигнатура: void showListOfNotVisitedVerAndHeur()

Назначение: выводит список непосещенных вершин и ее характеристик.

Описание аргументов: -

Возвращаемое значение: -

9. Сигнатура: void showListOfEdge()

Назначение: выводит отсортированный список ребер.

Описание аргументов: -

Возвращаемое значение: -

10. Сигнатура: Point* findVer(Vertical vertical)

Назначение: возвращает экземпляр структуры вершины по названию вершины.

Описание аргументов: vertical : Vertical – название вершины

Возвращаемое значение: Point* - экземпляр стуктуры вершины.

11. Сигнатура: void defineWeightAndHeur(Point* currentPoint)

Назначение: определяет эвристическую оценку и вес от начальной вершины до текущей для смежных вершин. В моем варианте определяется только вес.

Описание аргументов: currentPoint : Point* – текущая рассматриваемая вершина.

Возвращаемое значение: -

12. Сигнатура: Point* findNextVertical()

Назначение: ищет следующую вершину с наименьшими характеристиками

Описание аргументов: -

Возвращаемое значение: Point* - экземляр структуры следующей рассматриваемой вершины.

13. Сигнатура: Point* defineNextCurrentVer(Point* min)

Назначение: определяет следующую рассматриваемую вершину.

Описание аргументов: min : Point* – следующая рассматриваемая вершина.

Возвращаемое значение: Point*-экземпляр структуры следующей рассматриваемой вершины.

14. Сигнатура: void doAlgoritmA()

Назначение: выполняет построение кратчайшего пути с помощью метода A*.

Возвращаемое значение: -

Тестирование

Тестирование жадного алгоритма.

Нумерация	Входные данные	Выходные данные
1	g j a b 1 a f 3 b c 5 b g 3 f g 4	Хотите считать данные из файла или ввести самостоятельно?(1/2) 1 Отсортированный список ребер a b 1 a f 3

	c d 6 d m 1 g e 4 e h 1 e n 1 n m 2 g i 5 i j 6 i k 1 j l 5 m j 3	b g 3 b c 5 c d 6 d m 1 e h 1 e n 1 f g 4 g e 4 g i 5 i k 1 i j 6 j l 5 m j 3 n m 2 Рассматриваемая вершина g Следующая вершина, выбранная по минимальному весу ребра e Рассматриваемая вершина e Следующая вершина, выбранная по минимальному весу ребра h Рассматриваемая вершина h Следующая вершина не найдена, переходим к предыдущей e Рассматриваемая вершина e Следующая вершина, выбранная по минимальному весу ребра n Рассматриваемая вершина n Следующая вершина, выбранная по минимальному весу ребра m Рассматриваемая вершина m Следующая вершина, выбранная по минимальному весу ребра j Результат работы алгоритма: genmj

		Хотите продолжить?(y/n) n
2	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	Результат работы алгоритма: abcde
3	a b a b 1.0 a c 1.0	Результат работы алгоритма: ab
4	a j a b 1 b c 1 c d 1 d e 1 e j 1 a f 1 f g 1 g h 1 h i 1 i j 1	Результат работы алгоритма: abcdej

Тестирование алгоритма A*.

Нумерация	Входные данные	Выходные данные
1	g j a b 1 a f 3 b c 5 b g 3 f g 4 c d 6	Хотите считать данные из файла или ввести самостоятельно?(1/2) 1 Рассматриваемая вершина g Считаем вес для смежных вершин Список непосещенных вершин: вершина эвриситическая оценка вес

	d m 1	a 1 0
	g e 4	b 2 0
	e h 1	f 3 0
	e n 1	c 4 0
	n m 2	d 6 0
	g i 5	m 7 0
	i j 6	e 8 4
	i k 1	h 9 0
	j l 5	n 10 0
	m j 3	i 11 5
	1	j 12 0
	2	k 13 0
	3	l 14 0
	4	Конец списка.
	5	Выбранная часть пути:
	6	g e
	7	Рассматриваемая вершина e
	8	Считаем вес для смежных вершин
	9	Список непосещенных вершин:
	10	вершина эвриситическая оценка вес
	11	a 1 0
	12	b 2 0
	13	f 3 0
14		c 4 0
		d 6 0
		m 7 0
		h 9 5
		n 10 5
		i 11 5
		j 12 0
		k 13 0
		l 14 0
		Конец списка.
		Выбранная часть пути:

		e h Рассматриваемая вершина h Считаем вес для смежных вершин Список непосещенных вершин: вершина эвриситическая оценка вес a 1 0 b 2 0 f 3 0 c 4 0 d 6 0 m 7 0 n 10 5 i 11 5 j 12 0 k 13 0 l 14 0 Конец списка. Выбранная часть пути: e n Рассматриваемая вершина n Считаем вес для смежных вершин Список непосещенных вершин: вершина эвриситическая оценка вес a 1 0 b 2 0 f 3 0 c 4 0 d 6 0 m 7 7 i 11 5 j 12 0 k 13 0 l 14 0 Конец списка.

		Выбранная часть пути: n m Рассматриваемая вершина m Считаем вес для смежных вершин Список непосещенных вершин: вершина эвриситическая оценка вес a 1 0 b 2 0 f 3 0 c 4 0 d 6 0 i 11 5 j 12 10 k 13 0 l 14 0 Конец списка. Выбранная часть пути: g i Рассматриваемая вершина i Считаем вес для смежных вершин Список непосещенных вершин: вершина эвриситическая оценка вес a 1 0 b 2 0 f 3 0 c 4 0 d 6 0 j 12 10 k 13 6 l 14 0 Конец списка. Выбранная часть пути: i k Рассматриваемая вершина k

		Считаем вес для смежных вершин Список непосещенных вершин: вершина эвриситическая оценка вес a 1 0 b 2 0 f 3 0 c 4 0 d 6 0 j 12 10 l 14 0 Конец списка. Выбранная часть пути: m j Результат работы алгоритма: genmj Хотите продолжить?(y/n) n
2	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 5 4 3 2 1	Результат работы алгоритма: ade
3	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	Результат работы алгоритма: abcde

	1 2 3 4 5	
4	a j a b 1 b c 1 c d 1 d e 1 e j 1 a f 1 f g 1 g h 1 h i 1 i j 1 7 4 7 2 5 7 2 7 9 2	Результат работы алгоритма: abcdej

Выводы.

Были изучены жадный алгоритм и алгоритм A*. Реализована программа для поиска пути в ориентированном графе.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab2_1.cpp

```
#include <iostream>
#include <list>
#include <fstream>

#define Vertical char
#define Weight float

struct Point {//структура для вершины графа
    Vertical ver; //название вершины
    bool isVisited;//посещена ли вершина или нет

    Point() = default;
    Point(Vertical ver, bool isVisited) {
        this->ver = ver;
        this->isVisited = isVisited;
    }
};
struct Edge {//стуктура для ребра графа
    Point* initVer;//начальная вершина
    Point* finVer;//конечная вершина
    Weight weightVer;//вес ребра

    Edge() = default;
    Edge(Point* init, Point* fin, Weight weight) {
        initVer = init;
        finVer = fin;
        weightVer = weight;
    }
};
bool compForSort (const Edge& first, const Edge& second)//функция для сортировки
ребер
{
    if (first.initVer->ver < second.initVer->ver)//по первой вершине
        return true;
    if (first.initVer->ver > second.initVer->ver)
        return false;
    if (first.weightVer < second.weightVer)//по весу
        return true;
    return false;
}

class PathSearching {//класс для поиска пути в графе
private:
    Vertical initDestVer, finDestVer;//начальная и конечная вершины
    std::string answer;//строка для хранения пути
public:
    std::list<Edge> listOfEdge;//список ребер
    std::list<Point*> listOfVer;//список вершин
    PathSearching(Vertical initDestVer, Vertical finDestVer) {
        this->initDestVer = initDestVer;
        this->finDestVer = finDestVer;
    }
    void readEdgeAndVer(std::istream& fin) {//ввод ребер и формирование списков
        Vertical initVer, finVer;
        Weight weight = 0;
        while (fin >> initVer >> finVer >> weight) {
```

```
            if (initVer == '0')//для удобства был введен символ для остановки
ввода
                break;
            Point* initPointVer = findVer(initVer);
            Point* finPointVer = findVer(finVer);
            if (initPointVer == nullptr) {
                initPointVer = new Point(initVer, false);
                listOfVer.push_back(initPointVer);//формирование списка вершин
            }
            if (finPointVer == nullptr) {
                finPointVer = new Point(finVer, false);
                listOfVer.push_back(finPointVer);
            }
            listOfEdge.emplace_back(initPointVer, finPointVer,
weight);//формирование списка ребер
        }
    }
    void showAnswer() {//вывод пути
        std::cout << "Результат работы алгоритма:\n";
        std::cout << answer << "\n";
    }

    void showListOfEdge() {//вывод списка ребер
        std::cout << "Отсортированный список ребер\n";
        for (auto i : listOfEdge) {
            std::cout << i.initVer->ver << " " << i.finVer->ver << " " <<
i.weightVer << "\n";
        }
    }

    Point* findVer(Vertical vertical) {//поиск структуры вершины по ее названию
        for (auto i : listOfVer) {
            if (i->ver == vertical) {
                return i;
            }
        }
        return nullptr;
    }

    void doGreedyAlgoritm() {//построение пути с помощью жадного метода
        listOfEdge.sort(compForSort);//сортировка списка ребер по первой вершине
и весу
        showListOfEdge();
        Point *currentPoint;
        for (auto i : listOfVer) {//поиск начальной вершины
            if (i->ver == initDestVer) {
                currentPoint = i;
                break;
            }
        }
        while (currentPoint->ver != finDestVer) {//пока рассматриваемая вершина
не является конечной
            Vertical curVer = currentPoint->ver;//сохраняем название
рассматриаемой вершины
            std::cout << "Рассматриваемая вершина " << currentPoint->ver <<
"\n";
            answer += curVer;//и добавляем ее в ответ
            currentPoint->isVisited = true;//и добавляем в список рассмотренных
            for (auto i : listOfEdge) {
                if (i.initVer == currentPoint) {
                    if (!i.finVer->isVisited) {
                        currentPoint = i.finVer;//выбор следующей
рассматриваемой вершины, как первой найденной благодаря сортировке по весу
                        break;
```

```
                    }
                }
            }
            if (currentPoint->ver == curVer) {//если рассматриваемая вершина не
поменялась
                std::cout << "Следующая вершина не найдена, переходим к
предыдущей ";
                for (auto i : listOfEdge) {
                    if (i.finVer == currentPoint) {
                        currentPoint = i.initVer;//тогда переходим к
предпоследней
                        std::cout << currentPoint->ver << "\n";
                        answer.pop_back();
                        answer.pop_back();
                        break;
                    }
                }
            } else std::cout << "Следующая вершина, выбранная по минимальному
весу ребра " << currentPoint->ver << "\n";
        }
        answer += finDestVer;//в ответ добавляем конечную вершину
    }
};
int main() {
    char answ = 'y';
    while (answ == 'y') {
        std::cout << "Хотите считать данные из файла или ввести
самостоятельно?(1/2)\n";
        std::cin >> answ;
        Vertical initDestVer, finDestVer;
        std::ifstream fin("test1.txt");
        if (answ == '2') {
            std::cout << "Введите начальную и конечную вершину\n";
            std::cin >> initDestVer >> finDestVer;
        } else {
            fin >> initDestVer >> finDestVer;
        }
        auto answer = new PathSearching(initDestVer, finDestVer);
        if (answ == '2') {
            std::cout << "Введите ребра и вес\n";
            answer->readEdgeAndVer(std::cin);
        } else answer->readEdgeAndVer(fin);
        fin.close();
        answer->doGreedyAlgoritm();
        answer->showAnswer();
        std::cout << "Хотите продолжить?(y/n)\n";
        std::cin >> answ;
    }
    return 0;
}
```

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lab2_2.cpp

```
#include <iostream>
#include <list>
#include <fstream>

#define Vertical char
#define Weight float

struct Point {//структура для вершины графа
    Vertical ver; //название вершины
    bool isVisited;//посещена ли вершина или нет
    Weight heurEst, wei;//эвристическая оценка и вес от начальной вершины до
текущей
    Point* parent;//начальная вершина в ребре

    Point() = default;
    Point(Vertical ver, bool isVisited) {
        this->ver = ver;
        parent = nullptr;
        this->isVisited = isVisited;
        this->heurEst = 0;
        this->wei = 0;
    }
};

struct Edge {//стуктура для ребра графа
    Point* initVer;//начальная вершина
    Point* finVer;//конечная вершина
    Weight weightVer;//вес ребра

    Edge() = default;
    Edge(Point* init, Point* fin, Weight weight) {
        initVer = init;
        finVer = fin;
        weightVer = weight;
    }
};

class PathSearching {//класс для нахождения кратчайшего пути в графе по
алгоритму A*
private:
    Vertical initDestVer, finDestVer;//начальная и конечная вершины
public:
    std::list<Edge> listOfEdge;//список ребер
    std::list<Point*> listOfVer;//список вершин
    PathSearching(Vertical initDestVer, Vertical finDestVer) {
        this->initDestVer = initDestVer;
        this->finDestVer = finDestVer;
    }
    void readEdgeAndVer(std::istream& fin, bool isCin) {//ввод ребер и
формирование списков
        Vertical initVer, finVer;
        Weight weight = 0;
        while (fin >> initVer >> finVer >> weight) {
            if (initVer == '0')//для удобства был введен символ для остановки
```

```
ввода
                break;
            Point* initPointVer = findVer(initVer);
            Point* finPointVer = findVer(finVer);
            if (initPointVer == nullptr) {
                initPointVer = new Point(initVer, false);
                listOfVer.push_back(initPointVer);//формирование списка вершин
            }
            if (finPointVer == nullptr) {
                finPointVer = new Point(finVer, false);
                listOfVer.push_back(finPointVer);
            }
            listOfEdge.emplace_back(initPointVer, finPointVer,
weight);//формирование списка ребер
        }
        for (auto i : listOfVer) {
            if (isCin) {
                std::cout << "Введите эвристическую функцию для вершины " << i-
>ver << "\n";
            }
            fin >> i->heurEst;
        }
    }

    Point* findVer(Vertical vertical) {//поиск структуры вершины по ее названию
        for (auto i : listOfVer) {
            if (i->ver == vertical) {
                return i;
            }
        }
        return nullptr;
    }

    void showAnswer() {//вывод кратчайшего пути
        Point* currentPoint = findVer(finDestVer);
        std::string answer;
        while (currentPoint != nullptr) {
            answer.push_back(currentPoint->ver);
            currentPoint = currentPoint->parent;
        }
        std::cout << "Результат работы алгоритма:\n";
        for (int i = answer.length() - 1; i >= 0;i--) {
            std::cout << answer[i];
        }
        std::cout << "\n";
    }

    void showListOfNotVisitedVerAndHeur() {//вывод списка непосещенных вершин и
ее характеристик
        std::cout << "Список непосещенных вершин:\nвершина эвриситическая оценка
вес\n";
        for (auto i : listOfVer) {
            if (!i->isVisited) {
                std::cout << i->ver << " " << i->heurEst << " " << i->wei <<
"\n";
            }
        }
        std::cout << "Конец списка.\n";
    }

    void defineWeightAndHeur(Point* currentPoint) {//определение эвристической
оценки и веса от начальной вершины до текущей для смежных вершин
        std::cout << "Считаем вес для смежных вершин\n";
        for (auto i : listOfEdge) {
```

```
            if (i.initVer == currentPoint) {
                if (i.finVer->wei > i.weightVer + currentPoint->wei || i.finVer-
>wei == 0) {//при меньшей сумме характеристик вершины или их отсутствии присвоим
подсчитанные раннее
                    i.finVer->wei = i.weightVer + currentPoint->wei;
                    i.finVer->isVisited = false;
                    std::cout << "Вес ребер от начальной до текущей для вершины
" << i.finVer->ver << " посчитан: " << i.finVer->wei << "\n";
                }
            }
        }
    }

    Point* findNextVertical() {//поиск следующей вершины с наименьшими
характеристиками
        Point* min = nullptr;
        std::cout << "Выбирается вершина с наименьшей суммой эвристической
оценки и веса ребер от начальной вершины до текущей\n";
        for (auto i : listOfVer) {
            if ((min == nullptr || (min->heurEst + min->wei) > (i->heurEst + i-
>wei) ||
                 (min->heurEst + min->wei) == (i->heurEst + i->wei) && i-
>heurEst < min->heurEst) &&
                !i->isVisited && !(i->wei == 0)) {//при равенстве сумм
характеристик вершин выбирается та, у которой эвристическая оценка меньше
                min = i;
            }
        }
        std::cout << "Наименьшая вершина " << min->ver << " с " << min->heurEst
+ min->wei << "\n";
        return min;
    }
    Point* defineNextCurrentVer(Point* min) {//определение следующей
рассматриваемой вершины
        for (auto i : listOfEdge) {
            if (!i.finVer->isVisited && i.finVer->ver == min->ver && min->wei ==
i.initVer->wei + i.weightVer) {//находим начальную вершину ребра для построения
пути
                std::cout << "Выбранная часть пути:\n";
                std::cout << i.initVer->ver << " " << min->ver << "\n";
                min->parent = i.initVer;
                break;
            }
        }
        return min;
    }

    void doAlgoritmA() {//построение кратчайшего пути с помощью метода A*
        Point *currentPoint = findVer(initDestVer);
        while (currentPoint->ver != finDestVer) {//пока рассматриваемая вершина
не окажется конечной
            currentPoint->isVisited = true;//отмечаем рассмотренную вершину
            std::cout << "Рассматриваемая вершина " << currentPoint->ver <<
"\n";
            defineWeightAndHeur(currentPoint);
            showListOfNotVisitedVerAndHeur();
            Point *min = findNextVertical();
            currentPoint = defineNextCurrentVer(min);
        }
    }
};

int main() {
    char answ = 'y';
```

```
    while (answ == 'y') {
        std::cout << "Хотите считать данные из файла или ввести
самостоятельно?(1/2)\n";
        std::cin >> answ;
        Vertical initDestVer, finDestVer;
        std::ifstream fin("test2.txt");
        if (answ == '2') {
            std::cout << "Введите начальную и конечную вершину\n";
            std::cin >> initDestVer >> finDestVer;
        } else {
            fin >> initDestVer >> finDestVer;
        }
        auto answer = new PathSearching(initDestVer, finDestVer);
        if (answ == '2') {
            std::cout << "Введите ребра и вес\n";
            answer->readEdgeAndVer(std::cin, true);
        } else answer->readEdgeAndVer(fin, false);
        fin.close();
        answer->doAlgoritmA();
        answer->showAnswer();
        std::cout << "Хотите продолжить?(y/n)\n";
        std::cin >> answ;
    }
    return 0;
}
```