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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритмы поиска пути в графах

Студентка гр. 8382 _____ Кулачкова М.К.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Построить жадный алгоритм для нахождения пути в графе и алгоритм для нахождения кратчайшего пути в ориентированном графе методом A*.

Задание.

Разработайте программу, которая решает задачу построения пути в *ориентированном* графе при помощи **жадного алгоритма**, а также задачу построения кратчайшего пути **методом A***. 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. 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 ответами будут

abcde (жадный алгоритм)

и ade (алгоритм A*)

Вариант дополнительного задания.

Вар.8. Перед выполнением A* выполнять предобработку графа: для каждой вершины отсортировать список смежных вершин по приоритету.

Описание жадного алгоритма.

Жадный алгоритм осуществляет поиск пути в графе, на каждом шаге выбирая из вершин, смежных с последней просмотренной, ту, что соединена с ней ребром с наименьшим весом. Создается счётчик обработанных вершин. В строку для записи пути добавляется имя начальной вершины, она помечается как обработанная. Затем, пока не будет найдено решение или пока все вершины не будут обработаны, выполняется следующая последовательность действий.

1. Находится последняя обработанная вершина.
2. Среди смежных с ней вершин находится такая необработанная вершина, которая соединена с текущей ребром с наименьшим весом. Если есть рёбра с одинаковыми весами, выбирается то, которое соответствует вершине, идущей раньше по алфавиту. Если все смежные вершины уже обработаны, текущая вершина удаляется из пути и алгоритм возвращается к шагу 1.
3. Если смежная вершина была найдена, она помечается как обработанная и добавляется в путь.
4. Если найденная смежная вершина является концом искомого пути, путь до неё выводится на экран и алгоритм завершает работу.

Если все вершины обработаны, а путь так и не был найден, значит, его не существует.

Описание алгоритма A*.

Для предварительной обработки алгоритма, т.е. сортировки массивов инцидентных рёбер для каждой из вершин по приоритету, была использована библиотечная функция быстрой сортировки. Это было сделано по трём причинам: в задании не указано, что нужно писать функцию для сортировки

самим, быстрая сортировка обладает небольшой временной сложностью и использование библиотечных функций сокращает код. Для её работы реализован компаратор, который сравнивает рёбра по эвристической оценке расстояния между концом ребра и конечной вершиной искомого пути.

Поиск кратчайшего пути в графе реализуется с помощью алгоритма A*:

Создаётся массив для хранения индексов вершин, которые нужно рассмотреть. Для рассматриваемых вершин будет вычисляться эвристическая функция *f*, которая равна сумме кратчайшего пути из начальной вершины в текущую и эвристической оценки расстояния от текущей вершины до конечной, т.е., по условиям этой лабораторной, близости символов, обозначающих текущую и конечную вершины в таблице ASCII.

Начальная вершина добавляется в массив вершин, которые необходимо рассмотреть. Затем, до тех пор, пока этот массив не пуст, выполняется следующая последовательность действий.

1. В массиве вершин, которые необходимо рассмотреть, находится вершина, для которой значение эвристической функции *f* минимально.
2. Если таким образом найдена конечная вершина, путь до неё выводится на экран и алгоритм завершает работу.
3. Вершина помечается как обработанная и удаляется из массива вершин для рассмотрения.
4. Для каждой из смежных с ней вершин вычисляется длина пути до неё из начальной вершины: «путь до текущей вершины» + «вес ребра между текущей и смежной вершинами». Если вершина ещё не была обработана и/или если найденный путь короче текущего пути до неё, найденный путь сохраняется. Вершина добавляется в массив вершин, которые необходимо рассмотреть, если её там ещё не было.

Если путь из начальной вершины в конечную существует, то в какой-то момент работы алгоритма он будет выведен на шаге 3.

Оценка сложности алгоритма.

Временная сложность быстрой сортировки, используемой для предобработки графа, в среднем составит $O(V^2 \log V)$, где V – число вершин. Временная сложность алгоритмов поиска – как жадного, так и A* – составляет примерно $O(V^2)$. Ещё в худшем случае $O(V^3)$ составляет временная сложность инициализации графа. Просуммировав, получим сложность, приблизительно равную $O(V^3)$.

Сложность алгоритма по памяти зависит от плотности графа, т.е. числа рёбер в нем, т.к. граф хранится в виде списка смежности, в котором изменяющимся параметром будет количество рёбер, исходящих из каждой из вершин.

Описание структур данных.

В работе используются три структуры для хранения рёбер, вершин и самого графа.

Для хранения рёбер реализована структура *Edge*.

```
struct Edge {
	int vertex; //индекс конца ребра в графе
	double weight; //длина ребра
	int heuristics; //значение эвристической функции для вершины на конце
ребра
};
```

Она состоит из индекса вершины на конце ребра в массиве вершин в графе, веса ребра и эвристической оценки расстояния между вершиной на конце ребра и концом искомого пути.

Для хранения вершин реализована структура *Node*.

```
struct Node {
	char name; //обозначение вершины
	int num; //число смежных вершин
	Edge * edges; //массив исходящих ребер
	string path; //кратчайший путь до вершины
	double pathLength; //длина кратчайшего пути до вершины
	bool processed; //была ли вершина обработана алгоритмом
	Node(char name = 0); //инициализация вершины
	~Node() { delete[] edges; }
	void addNeighboor(int ind, double weight, int h); //добавление исходящего
ребра
};
```

Она содержит символьное поля для обозначения вершины, число смежных с ней вершин, массив исходящих рёбер, строку, в которой хранится кратчайший путь из начальной вершины пути в эту, длину этого пути, булеву переменную, равную *true*, если вершина была обработана алгоритмом поиска пути, и *false* в противном случае, а также конструктор, деструктор и метод, добавляющий новое ребро в массив инцидентных рёбер.

Для хранения графа реализована структура *Graph*.

```
struct Graph {
	int num; //число вершин
	Node vertices[MAX_VERTICES]; //массив вершин
	char start; //начало искомого пути
	char finish; //конец искомого пути
	Graph(char st, char fin); //инициализация графа
	int find(char name); //возвращает индекс искомой вершины в графе; если
вершины нет - возвращает -1
	void addVertex(char name); //добавление вершины в граф
	void addEdge(char from, char to, double weight); //добавление ребра в
граф
	void preprocessing(); //предобработка
	void search(); //поиск пути
	void alphSort();//сортировка вершин по алфавиту
	string path; //путь, находимый жадным алгоритмом
	void greedyAlgorithm(); //жадный алгоритм
	int h(char a) { return abs(a - finish); } //эвристическая оценка
расстояния
	double f(int ind); //значение эвристической функции "расстояние +
стоимость" для вершины с данным индексом
	void resetVertices(); //помечает все вершины как необработанные
	void print(); //вывод списка вершин на экран
};
```

Структура включает в себя число вершин в графе, статический массив вершин (число вершин в массиве равно 26, т.к. они обозначаются буквами английского алфавита), начало и конец искомого пути, строку, содержащую путь, находимый жадным алгоритмом, а также ряд методов, которые будут описаны в следующем пункте.

Описание функций и методов.

- Метод *void Node::addNeighboor(int ind, double weight, int h);* добавляет новое ребро в массив рёбер, инцидентных данной вершине. Здесь *ind* – индекс конца ребра в массиве вершин графа, *weight* – вес ребра, *h* – эвристическая оценка расстояния между концом ребра и концом искомого пути.

- Метод *int Graph::find(char name);* ищет в массиве вершин вершину с заданным именем. Возвращает индекс искомой вершины в графе; если вершины нет – возвращает -1.
- Метод *void Graph::addVertex(char name);* добавляет в граф вершину с заданным именем.
- Метод *void Graph::addEdge(char from, char to, double weight);* добавляет в граф ребро с заданными характеристиками. Если вершин на концах ребра ещё не было в массиве вершин графа, они туда добавляются. Для конечной вершины ребра рассчитывается эвристическая функция. Для начальной вершины вызывается метод *addNeighboor*.
- Метод *void Graph::preprocessing();* выполняет предобработку графа: для каждой вершины сортирует массив инцидентных ей рёбер по приоритету.
- Метод *void Graph::search();* осуществляет поиск кратчайшего пути.
- Метод *void Graph::greedyAlgorithm();* осуществляет поиск пути с использованием жадного алгоритма.
- Метод *int Graph::h(char a);* вычисляет эвристическую оценку расстояния между вершиной *a* и концом пути: возвращает разницу между именами заданной и конечной вершин в таблице ASCII.
- Метод *double Graph::f(int ind);* вычисляет значение эвристической функции для вершины с заданным индексом по формуле «кратчайший путь до вершины» + «эвристическая оценка расстояния от вершины до конца искомого пути».
- Метод *void Graph::resetVertices();* помечает все вершины как необработанные между запусками алгоритмов.

Тестирование.

№	Ввод	Вывод
1	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	Greedy algorithm: abcde A* algorithm: ade
2	b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	Greedy algorithm: bge A* algorithm: be
3	a f a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0	Greedy algorithm: abef A* algorithm: acdf
4	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0 c m 1.0 m n 1.0	Greedy algorithm: abdefg A* algorithm: ag
5	a d a b 1.0 b c 1.0 c a 1.0 a d 8.0	Greedy algorithm: ad A* algorithm: ad

Выводы.

Была реализована программа, осуществляющая поиск пути в графе с использованием жадного алгоритма, а также поиск кратчайшего пути в графе с использованием алгоритма A*. Полученные алгоритмы обладает кубической

сложностью по количеству вершин с учётом инициализации графа и его предобработки.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp

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

#define MAX_VERTICES 26

using namespace std;

//структура для хранения ребёр
struct Edge {
	int vertex; //индекс конца ребра в графе
	double weight; //длина ребра
	int heuristics; //значение эвристической функции для вершины на конце ребра
};

//компаратор для быстрой сортировки вершин по приоритету
int compare(const void * a, const void * b){
	return (((Edge*)a)->heuristics - ((Edge*)b)->heuristics);
}

//структура для хранения вершин графа
struct Node {
	char name; //обозначение вершины
	int num; //число смежных вершин
	Edge * edges; //массив исходящих ребер
	string path; //кратчайший путь до вершины
	double pathLength; //длина кратчайшего пути до вершины
	bool processed; //была ли вершина обработана алгоритмом
	Node(char name = 0); //инициализация вершины
	~Node() { delete[] edges; }
	void addNeighboor(int ind, double weight, int h); //добавление исходящего ребра
};

//структура для хранения графа
struct Graph {
	int num; //число вершин
	Node vertices[MAX_VERTICES]; //массив вершин
	char start; //начало искомого пути
	char finish; //конец искомого пути
	Graph(char st, char fin); //инициализация графа
	int find(char name); //возвращает индекс искомой вершины в графе; если  вершины нет -
возвращает -1
	void addVertex(char name); //добавление вершины в граф
	void addEdge(char from, char to, double weight); //добавление ребра в  граф
	void preprocessing(); //предобработка
	void search(); //поиск пути
	void alphSort();//сортировка вершин по алфавиту
	string path; //путь, находимый жадным алгоритмом
	void greedyAlgorithm(); //жадный алгоритм
	int h(char a) { return abs(a - finish); } //эвристическая оценка расстояния
	double f(int ind); //значение эвристической функции "расстояние + стоимость" для
вершины с данным индексом
	void resetVertices(); //помечает все вершины как необработанные
	void print(); //вывод списка вершин на экран
};
```

```
//инициализация графа
Graph::Graph(char st, char fin) {
	start = st;
	finish = fin;
	path = "";
	num = 0;
	addVertex(start);
	addVertex(finish);
}

//добавление ребра в граф
void Graph::addEdge(char from, char to, double weight) {
	//находим концы ребра в массиве вершин
	int indFrom = find(from);
	int indTo = find(to);
	//добавляем начало ребра, если его нет в массиве
	if (indFrom == -1) {
		addVertex(from);
		indFrom = num - 1;
	}
	//добавляем конец ребра, если его нет в массиве
	if (indTo == -1) {
		addVertex(to);
		indTo = num - 1;
	}
	vertices[indFrom].addNeighboor(indTo, weight, h(to));
}

//добавление вершины в граф
void Graph::addVertex(char name) {
	if (find(name) >= 0) return;
	vertices[num] = Node(name);
	num++;
}

//возвращает индекс искомой вершины в графе; если  вершины нет - возвращает -1
int Graph::find(char name) {
	for (int i = 0; i < num; i++) {
		if (vertices[i].name == name) return i;
	}
	return -1;
}

//вывод списка вершин на экран
void Graph::print() {
	cout << "List and number of vertices: ";
	for (int i = 0; i < num; i++) {
		cout << vertices[i].name << " ";
	}
	cout << endl;
}

//предобработка графа (сортировка смежных вершин по приоритету)
void Graph::preprocessing() {
	for (int i = 0; i < num; i++) {
		qsort(vertices[i].edges, vertices[i].num, sizeof(Edge), compare);
	}
}

//отметка всех вершин как необработанных
void Graph::resetVertices(){
    for (int i = 0; i < num; i++){
        vertices[i].processed = false;
    }
```

```
}

//эвристическая функция
double Graph::f(int ind) {
	return vertices[ind].pathLength + h(vertices[ind].name);
}

//жадный алгоритм поиска пути
void Graph::greedyAlgorithm() {
	int startInd = find(start);
	vertices[startInd].processed = true;
	int processed = 1;
	path = (1, start);
	while (processed < num) {
		int lastVisited = find(path.back());
		double minWeight = 10000;
		int minWeightInd = -1;
		for (int i = 0; i < vertices[lastVisited].num; i++) {
			if (!vertices[vertices[lastVisited].edges[i].vertex].processed &&
			(minWeight > vertices[lastVisited].edges[i].weight ||
			(minWeight == vertices[lastVisited].edges[i].weight &&
			vertices[vertices[lastVisited].edges[i].vertex].name <
vertices[minWeightInd].name))) {
				minWeight = vertices[lastVisited].edges[i].weight;
				minWeightInd = vertices[lastVisited].edges[i].vertex;
			}
		}
		if (minWeightInd == -1) {
			path.erase(path.length() - 1);
			continue;
		}
		vertices[minWeightInd].processed = true;
		path += vertices[minWeightInd].name;
		cout << "Intermediate path: " << path << endl;
		if (vertices[minWeightInd].name == finish) {
			cout << path << endl;
			return;
		}
		processed++;
	}

	cout << "There must be no path." << endl;
	return;
}

//алгоритм поиска кратчайшего пути
void Graph::search() {
	int startInd = find(start);
	vector<int> toVisit; //индексы вершин, которые нужно посетить
	toVisit.push_back(startInd);
	vertices[startInd].pathLength = 0;
	vertices[startInd].path = string(1, start);
	while (!toVisit.empty()) {
		double minF = 10000;
		int curr = 0, currInd = 0;
		//находим среди вершин, которые нужно обработать, вершину с наименьшим
значением
		//эвристической функции
		for (int i = 0; i < toVisit.size(); i++) {
			double fI = f(toVisit[i]);
			if (fI < minF) {
				minF = fI;
				curr = toVisit[i];
				currInd = i;
```

```
                        }
                }
                //если нашли конечную вершину - выводим путь до неё и завершаем алгоритм
                if (vertices[curr].name == finish) {
                        cout << vertices[curr].path << endl;
                        return;
                }
                //удаляем текущую вершину из множества вершин, которые нужно рассмотреть
                toVisit.erase(toVisit.begin() + currInd);
                //отмечаем её как просмотренную
                vertices[curr].processed = true;
                //рассматриваем вершины, смежные с текущей
                for (int v = 0; v < vertices[curr].num; v++) {
                        //вычисляем длину пути до рассматриваемой смежной вершины:
                        //"путь до текущей" + "вес ребра между текущей и смежной"
                        double tentScore = vertices[curr].pathLength +
vertices[curr].edges[v].weight;
                        //если вершина ещё не обработана или удалось улучшить длину пути до
неё, сохраняем новую длину пути
                        if (!vertices[vertices[curr].edges[v].vertex].processed ||
vertices[vertices[curr].edges[v].vertex].pathLength == -1
                                || tentScore <
vertices[vertices[curr].edges[v].vertex].pathLength) {
                                vertices[vertices[curr].edges[v].vertex].path =
vertices[curr].path + vertices[vertices[curr].edges[v].vertex].name;
                                vertices[vertices[curr].edges[v].vertex].pathLength = tentScore;
                                //вывод пути до рассматриваемой вершины на экран
                                cout << "Intermediate path: ";
                                cout << vertices[vertices[curr].edges[v].vertex].path << endl;
                                //добавляем смежные вершины в множество вершин, которые нужно
рассмотреть
                                toVisit.push_back(vertices[curr].edges[v].vertex);
                                //удаляем последнее добавление, если вершина там уже была
                                for (int i = 0; i < toVisit.size() - 1; i++) {
                                        if (toVisit[i] == vertices[curr].edges[v].vertex) {
                                                toVisit.pop_back();
                                                break;
                                        }
                                }
                        }
                }
        }
        //если алгоритм не завершился до этого момента, пути между вершинами нет
        cout << "There must be no path." << endl;
        return;
}

//инициализация вершины
Node::Node(char name){
        this->name = name;
        num = 0;
        edges = nullptr;
        pathLength = -1;
        path = "";
        processed = false;
}

//добавление исходящего ребра
void Node::addNeighboor(int ind, double weight, int h) {
        for (int i = 0; i < num; i++) {
                //если добавляемое ребро уже существует, можно изменить его вес
                if (edges[i].vertex == ind) {
                        cout << "This edge already exists. Do you want to rewrite it? y/n" <<
endl;
```

```
                        char answ;
                        cin >> answ;
                        if (answ == 'y' || answ == 'Y') {
                                edges[i].weight = weight;
                                return;
                        }
                        else if (answ != 'n' && answ != 'N') {
                                cout << "I'll take it as a \"No\"" << endl;
                        }
                        return;
                }
        }
        Edge * tmp = new Edge[num + 1];
        for (int i = 0; i < num; i++) {
                tmp[i].vertex = edges[i].vertex;
                tmp[i].weight = edges[i].weight;
                tmp[i].heuristics = edges[i].heuristics;
        }
        tmp[num].vertex = ind;
        tmp[num].weight = weight;
        tmp[num].heuristics = h;
        delete[] edges;
        edges = tmp;
        num++;
}


int main() {
        char start, finish;
        cout << "Enter a starting and a finishing point: " << endl;
        cin >> start >> finish;
        Graph * graph = new Graph(start, finish);
        char from, to, waste;
        double weight;
        int input = 0;
        cout << "Now enter a sequence of edges with weights: " << endl;
        while (cin >> from >> to >> weight) {
                graph->addEdge(from, to, weight);
        }
        graph->print();
        cout << "Greedy algorithm: " << endl;
        graph->greedyAlgorithm();
        graph->resetVertices();
        graph->preprocessing();
        cout << "A* algorithm: " << endl;
        graph->search();
        delete graph;
        return 0;
}
```