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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Построение и анализ алгоритмов»
Тема: Алгоритмы поиска пути в графах

Студентка гр. 8382 _____ Кузина А.М.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Задание.

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
```

Вариант.

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

Описание алгоритма.

На вход программе вначале подаются две вершины: из какой и в какую необходимо найти кратчайший путь. Далее подаются наборы данных: начальная и конечная вершины ребра, длина ребра и значение эвристической функции для конечной вершины ребра. На основе входных данных формируются матрица ребер графа, массив вершин и массив значений эвристической функции для вершин.

После окончания считывания входных данных начинается процесс поиска пути из начальной вершины в конечную вершину. Начальная вершина

добавляется в список вершин из которых может быть совершен переход. Далее на каждом шаге алгоритма, до тех пор пока не будет построен путь до конечной вершины, программа совершает следующие действия:

1. Для каждой вершины из списка вершин, из которых может быть совершен переход, для всех возможных ребер, по которым еще не был совершен переход, вычисляется значение «веса» этого перехода. «Вес» перехода равняется сумме длины ребра, значению эвристической функции вершины в которую будет совершен переход и длине уже пройденного пути от начальной вершины, до вершины начала выбранного ребра.
2. Из всех полученных «весов» для ребер выбирается наименьшее и помечается как пройденное — по нему больше не может быть совершен переход. Если переход в выбранную вершину совершается первый раз, она добавляется в список вершин из которых может быть совершен переход на последующих шагах.
3. Если переход в выбранную вершину совершается первый раз, присваиваются значения минимальной длины пути от начальной вершины до данной и пути, состоящего из имен вершин от начальной, до текущей включительно. Если же в выбранную вершину переход совершается не первый раз и на текущем шаге мы попали в нее более коротким путем, то для нее и всех вершин, содержащих данную в их кратчайшем пути, изменяются значения минимальной длины пути от начальной вершины до заданной и пути, состоящего из имен вершин от начальной, до заданной включительно.
4. Производится проверка того, попали ли мы в необходимую нам конечную вершину. Если это верно, то на экран выводится строка-ответ и программа завершается. В противном случае программа продолжает поиск следующей вершины.

Сложность алгоритма.

Сложность алгоритма A* по времени зависит от эвристической функции. В худшем случае, число вершин, исследуемых алгоритмом, растёт

экспоненциально по сравнению с длиной оптимального пути. Но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию: ошибка текущей эвристики растет не быстрее, чем логарифм от оптимальной эвристики. Эвристика заданная как близость символов, обозначающих вершины графа, в таблице ASCII, не является оптимальной так как не позволяет определить действительную удаленность вершин друг от друга.

Сложность по памяти при достаточно точной эвристической функции может составлять O(кол-во ребер + кол-во вершин) так как в таком случае кратчайший путь может быть построен сразу, без возвращения к уже пройденным вершинам. В худшем случае для нахождения кратчайшего пути будет нужно просматривать все ребра графа, тогда сложность может быть экспоненциальной.

Описание структур данных и функций.

Описание класса *graf* — хранение частичных решений:

В *from* и *to* хранятся имена вершин, между которыми нужно найти путь. Массив вершин *vertex*, массив значений эвристической функции для вершин *evrist*, а также двойной массив ребер *edges* хранят основную информацию о заданном графе. Массив *open_vertex* хранит список вершин из которых может быть совершен переход в другие вершины. Двойной массив *pos* соответственно для каждого ребра из *edges* хранит 0 или 1, обозначающее возможность перехода по данному ребру. Массивы *num_way* и *way* для каждой вершины из vertex хранят соответственно минимальное численное и буквенное значение пути от начальной вершины до заданной.

Метод *print()* выводит на экран двойной массив ребер *edges*.

Метод *init()* считывает и сохраняет входные данные в соответствующие массивы.

Метод *is_new(char a)* возвращает -1, если a — новая вершина и *i*, если a располагается в массиве вершин на *i-ой* позиции.

Метод *add_vertex(char a)* добавляет вершину a в конец списка вершин.

Метод *new_connect(char first, char second, float way)* добавляет ребро длины way между вершинами first и second в массив ребер.

Метод *is_closed(int i)* возвращает 1, если вершина *i* еще не добавлена в список доступных для перехода вершин, иначе 0.

Метод *change(int tmp, int old_num, int new_num, string new_way)* для каждой вершины проверяет, входит ли в ее кратчайший путь вершина под номером *tmp* в списке вершин. При положительном результате проверки изменяет значения численного и буквенного минимального пути для данной вершины.

Метод *reboot(int tmp, int next)* обновляет, если необходимо, значение численного и буквенного минимального пути для вершины *next,* после перехода по соответствующему ребру.

Метод *find_next()* для данного шага алгоритма находит ребро по которому будет совершен переход. Возвращает вершину в которую был совершен переход.

Метод *serch()* повторяет поиск пути до тех пор, пока *find_next()* не вернет конечную вершину или -1, если пути в графе нет.

Тестирование.

Входные данные	Выходные данные
f t f b 6 20 f c 10 5 c d 1 4 c e 3 1 d e 1 1 b c 2 5 e t 50 1	Answer: fbcdet
a e a b 3.0 4 b c 1.0 3 c d 1.0 1 a d 5.0 1 d e 1.0 0	Add new vertex a at 0 Add new vertex b at 1 Add new adge: ab: 3 Add new vertex c at 2 Add new adge: bc: 1 Add new vertex d at 3 Add new adge: cd: 1 Add new adge: ad: 5 Add new vertex e at 4 Add new adge: de: 1

	edge matrix: a b c d e a 0 3 0 5 0 b 0 0 1 0 0 c 0 0 0 1 0 d 0 0 0 0 1 e 0 0 0 0 0 Find way from a to e View vertex: a weight of edge ab = 7 weight of edge ad = 6 Step from a to d Ways: a way: a b way: c way: d way: ad e way: View vertex: a weight of edge ab = 7 View vertex: d weight of edge de = 6 Step from d to e Ways: a way: a b way: c way: d way: ad e way: ade Answer: ade
a j a b 1.0 4 b c 1.0 3 c d 1.0 2 a d 3.0 2 d e 2.0 1 e j 1.0 0 b j 7.0 0 c j 6.0 0	Answer: adej
a l a b 1 10 a f 3 6 b c 5 9 b g 3 5 f g 4 5 c d 6 8 d m 1 1 g e 4 7	Answer: abgenmjl

e h 1 4 e n 1 2 n m 2 1 g i 5 3 i j 6 2 i k 1 1 j l 5 0 m j 3 2	
a l l a 3.0 0 a b 2.3 2 b c 2.0 1 c a 1 0	There is no way!

Выводы.

В ходе выполнения работы была написана программа, находящая в задаваемом пользователем ориентированном графе минимальный путь между двумя вершинами методом A*. Также были получены знания о работе жадного алгоритма поиска путей в графе и алгоритма A*

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ФАЙЛА MAIN.CPP

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>

class graf {
	char* vertex;
	int count;
	int actual_count;
	char from;
	char to;
	int* evrist;

	int* num_way;
	std::string* way;

	int open_count;
	int* open_vertex;
	double** edges;
	int** pos;
	std::string answ;

public:
	graf() {
		//Выделение памяти под все структуры и задание в них необходимых "нулевых" значений
		count = 66;
		actual_count = 0;
		vertex = new char[count];
		open_vertex = new int[count];
		way = new std::string[count];
		num_way = new int[count];
		evrist = new int[count];

		open_count = 0;
		answ = "";
		edges = (double**)malloc(sizeof(double*) * count);
		pos = (int**)malloc(sizeof(int*) * count);
		for (int i = 0; i < count; i++) {
			vertex[i] = ' ';
			open_vertex[i] = 0;
			way[i] = "";
			num_way[i] = 0;
			evrist[i] = 0;
		}
		for (int i = 0; i < count; i++) {
			edges[i] = new double[count];
			pos[i] = new int[count];
		}
		for (int i = 0; i < count; i++) {
			for (int j = 0; j < count; j++) {
				edges[i][j] = 0;
				pos[i][j] = 1;
			}
		}

	}
	int is_new(char a) {
		// проверка на то, является ли заданная вершина новой или уже существующей
		int i;
		for (i = 0; i < count; i++) {
			if (vertex[i] == a) {
				return i;
			}
```

```
                if (vertex[i] == ' ') break;
            }
            return -1;
        }
        int add_vertex(char a) {
            // добавление новой вершины в список вершин
            int i = 0;
            while (vertex[i] != ' ') i++;
            vertex[i] = a;
            actual_count++;
            std::cout << "Add new vertex " << a << " at " << i << std::endl;
            return i;
        }
        bool new_connect(char first, char second, float way) {
            // добавление ребра в матрицу ребер
            int a, b;
            if (first == second) return 0;
            a = is_new(first);
            b = is_new(second); // проверка на то, разные ли вершины и новые ли они
            if (a == -1) a = add_vertex(first);
            if (b == -1) b = add_vertex(second);
            if (a == b) return 0;
            std::cout << "Add new adge: " << first << second << ": " << way << std::endl;
            edges[a][b] = way;
            return 1;
        }
        void init() {
            // Инициализация графа: заполнение матрицы ребер, массива вершин, а также задание
эвристических значений для вершин.
            //
            scanf("%c %c\n", &from, &to);
            char first, second;
            int evr = 0;
            float way = 0;
            int enter = 3;
            int i = 0;
            char c;
            while (true) {
                // считывание  идет: вершина начала ребра, вершина конца ребра, длина ребра,
значение эвристической функции для вершины конца ребра
                std::cin >> first >> second >> way >> evr;
                enter = scanf("%c", &c);
                //cout << first << second << way << " " << evr << endl;
                new_connect(first, second, way);
                i = is_new(second);
                evrist[i] = evr;
                if (enter != 1) break;
                if (c == 'q') break;

            }
        }
        void print() {
            // Вывод на экран матрицы ребер
            std::cout << "edge matrix:\n  ";
            for (int i = 0; i < actual_count; i++) {
                std::cout << vertex[i] << " ";
            }
            std::cout << "\n";

            for (int i = 0; i < actual_count; i++) {
                std::cout << vertex[i] << " ";
                for (int j = 0; j < actual_count; j++) {
                    std::cout << edges[i][j] << " ";
                }
                std::cout << "\n";
```

```
		}
	}
	int is_closed(int i) {
		//Возвращает 1, если вершина i еще не добавлена в список доступных для перехода вершин. Иначе 0.
		for (int j = 0; j < open_count; j++) {
			if (open_vertex[j] == i) return 0;
		}
		return 1;
	}
	void change(int tmp, int old_num, int new_num, std::string new_way) {
		std::cout << "Change all ways that include " << vertex[tmp] << "\nBefore: \n";
		print_ways();
		char u = vertex[tmp];
		// обновляем пути для всех вершин, пути которых содержали в себе обновленную вершину
		std::string change = "";
		int n = 0;
		int len = 0;
		for (int i = 0; i < actual_count; i++) { // обход всех вершин
			change = "";
			// cout << way[i] <<endl;
			n = way[i].find(u);
			len = way[i].size();
			if (n != std::string::npos) { // если в пути содержится искомая вершина - заменяем все до нее на ее новый кратчайший путь
				num_way[i] = num_way[i] - old_num + new_num;

				for (int j = n; j < len - 1; j++) {
					change += way[i][n + j];
				}
				way[i] = new_way;
				way[i] += change;

			}

		}
		std::cout << "After: \n";
			print_ways();

	}
	void reboot(int tmp, int next) {
		if (next == is_new(from)) return;
		if (num_way[next] == 0) { // если в вершину мы пришли первый раз, то:
			num_way[next] = num_way[tmp] + edges[tmp][next];
			way[next] = way[tmp];
			way[next].push_back(vertex[next]);// задаем вершине кратчайшие длину и путь
			print_ways();
		}
		else if (num_way[next] > num_way[tmp] + edges[tmp][next]) { // если мы пришли в вершину более коротким путем, чем раньше, то:

			way[next] = way[tmp];
			way[next].push_back(vertex[next]);
			std::cout << "New min way for vertex " << vertex[next] << ": " << way[next] << std::endl;
			change(next, num_way[next], num_way[tmp] + edges[tmp][next], way[next]); // обновляем пути для всех вершин, содержащих данную
			num_way[next] = num_way[tmp] + edges[tmp][next]; // обновляем кратчайшие длину и путь для даной вершины
		}

	}

	void print_ways() {
```

```
		// Для каждой вершины выводит кратчайший на данный момент путь от начальной. (Если путь уже найден)
		std::cout << "Ways: \n";
		for (int i = 0; i < actual_count; i++) {
			std::cout <<"\t"<< vertex[i] << " way: " << way[i] << std::endl;
		}
		std::cout << std::endl;
		return;
	}
	int find_next() {

		float min = 666;
		float c = 0;
		int next = -1;
		int tmp = 0;

		for (int i = 0; i < open_count; i++) { // Обход всех доступных к переходу вершин
			std::cout << "View vertex: " << vertex[open_vertex[i]] << "\n";
			for (int j = 0; j < actual_count; j++) {
				if (edges[open_vertex[i]][j] > 0 && pos[open_vertex[i]][j]) {
					// если ребро между вершинами существует и этот переход еще не совершался :
					c = edges[open_vertex[i]][j] + evrist[j] + num_way[open_vertex[i]];// вычисление "веса" перехода по данному ребру: длина ребра + сумма уже пройденого пути (для вершины из которой мы идем) + значение эвристической фуункции для вершины в которую мы идем.
					std::cout << "\tweight of edge " << vertex[open_vertex[i]] << vertex[j] << " = " << c << std::endl;
					if (c <= min) {
						min = c;
						next = j;
						tmp = open_vertex[i]; // устанавливаем временный минимум для данного шага
					}
				}
			}
		}
		pos[tmp][next] = 0;
		if (is_closed(next)) { // если вершина нам еще не встречалась, добавляем в список доступных для переходов вершин
			open_vertex[open_count] = next;
			open_count++;
		}
		if (next != -1) {
			std::cout << "Step from " << vertex[tmp] << " to " << vertex[next] << std::endl;
			reboot(tmp, next);

		}//обновляем значение вершины в которую мы перешли
		return next;
	}

	void serch() {
		std::cout << "Find way from " << from << " to " << to << "\n\n";
		int tmp = is_new(from);
		int end = is_new(to);
		evrist[end] = 0;
		int check = 0;
		int flag = 0;

		if (tmp == -1 || end == -1) { // проверка на то, существуют ли исходная и конечная вершины в списке вершин
			std::cout << "There is no way!\n";
			return;
		}
```

```
                open_vertex[0] = tmp;
                open_count++;
                way[tmp] = from;
                int next = find_next();
                while (next != -1) { // пока не найден путь или не пройден весь граф
                        if (next == end) {
                                std::cout << "Answer: " << way[end] << std::endl;
                                return;
                        }

                        next = find_next(); // поиск следующей вершины
                }
                std::cout << "There is no way!\n";
                return;
        }

};


int main() {

        graf* one = new graf;
        one->init();
        one->print();
        one->serch();

        return 0;
}
```