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 на графах

Студент гр. 8304 Самакаев Д.И.

Преподаватель Размочаева Н.В.

Санкт-Петербург

2019

Вариант 3.

Цель работы.

Написать функцию, проверяющую эвристику на допустимость и монотонность.

Основные теоретические положения.

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.

Описание алгоритма.

Для решения поставленной задачи был реализован алгоритм A*. В качестве эвристической функции была использована функция evristic(c1, c2), возвращающая расстояние между двумя символами. В начале каждой итерации в массиве ищется элемент приоритет, которого минимален, он удаляется из списка и начинается осмотр всех ребер выходящих из выбранного элемента. Если нашлась вершина путь до которой был больше чем найденный, то данный путь заменяется на найденный. Для хранения значений имен узлов и ребер выходящих из них был использован словарь и структура elem, хранящая ребра выходящие из текущей вершины, имя вершины из которой был найден минимальный путь и длина до начальной позиции. Сложность алгоритма: O(|V|*|V| + |E|), где V – множество вершин, а E – множество ребер.

Тестирование.

Таблица 1 – Результаты тестирование

Ввод	**Вывод**
a e a b 3 b c 1 c d 1 a d 5 d e 1	Allowable monotone ade
b m a b 1 a f 3 b c 5 b g 3 f g 4 c d 6 d m 1	Not allowable not monotone bcdm
a e a b 0 b c 1 c d0 a d 5 d e 1	Not allowable Not monotone abcde

Вывод.

В ходе работы был построен и анализирован алгоритм A* на основе решения задачи о нахождении минимального пути в графе. Исходный код программы представлен в приложении 1.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД

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

struct elem {
	std::vector<std::pair<char, int>> ways;
	int length;
};

bool cmp(const std::pair<char, int>& a, const std::pair<char, int>& b) {
	if (a.second == b.second) return a.first > b.first;
	return a.second < b.second;
}

int get_length(char a, std::map<char, elem>& my_map) { return my_map[a].length; }

int evristic(char a, char where) { return abs(a - where); }

int total(char a, char where, std::map<char, elem>& my_map) { return get_length(a, my_map) +
evristic(a, where); }

char MIN_F(std::set<char>& open, char where, std::map<char, elem>& my_map) {
	int min = std::numeric_limits<int>::max();
	char curr = 0;
	for (char i : open)
		if (total(i, where, my_map) < min) {
			min = total(i, where, my_map);
			curr = i;
		}
	return curr;
}

std::string make_path(std::map<char, char>& from, char start, char where) {
	std::string path = { where };
	char curr = where;
	while (curr != start) {
		curr = from[curr];
		path += curr;
	}
	std::reverse(path.begin(), path.end());
	return path;
}

void findWay(char start, char end, std::map<char, elem>& my_map) {
	char curr = start;
	std::set<char> closed_set;
	std::set<char> open_set = { start };
	std::map<char, char> path_syms;
	while (!open_set.empty()) {
		curr = MIN_F(open_set, end, my_map);

		if (curr == end) {
			std::cout << make_path(path_syms, start, end);
			return;
		}
		open_set.erase(curr);
		closed_set.insert(curr);
```

```
		for (auto neighbour : my_map[curr].ways) {
			bool tentative_is_better;
			if (closed_set.find(neighbour.first) != closed_set.end()) continue;
			int tentative_g_score = get_length(curr, my_map) + neighbour.second;
			if (open_set.find(neighbour.first) == open_set.end()) {
				open_set.insert(neighbour.first);
				tentative_is_better = true;
			}
			else {
				tentative_is_better = tentative_g_score <
get_length(neighbour.first, my_map);
			}

			if (tentative_is_better) {
				path_syms[neighbour.first] = curr;
				my_map[neighbour.first].length = tentative_g_score;
			}
		}
	}
	return;
}

size_t shortest_way(std::map<char, elem>& my_map,char start_ch, char end_ch, size_t min_length,
size_t current_length, size_t buff) {
	if (start_ch == end_ch) {
		return current_length;
	}
	for (size_t i = 0; i < my_map[start_ch].ways.size(); i++) {
		buff = shortest_way(my_map, my_map[start_ch].ways[i].first, end_ch, min_length,
current_length + my_map[start_ch].ways[i].second, buff);
			if (min_length > buff)
				min_length = buff;
	}
	return min_length;
}

bool check_monotony(std::map<char, elem>& my_map, char end_ch) {
	for (auto it = my_map.begin(); it != my_map.end(); ++it) {
		for (size_t i = 0; i < it->second.ways.size(); i++) {
			if (evristic(it->first, end_ch) - evristic(it->second.ways[i].first,
end_ch) > it->second.ways[i].second)
				return false;
		}
	}
	return true;
}

bool check_ambissibility(std::map<char, elem>& my_map, char end_ch) {
	if (check_monotony(my_map, end_ch))
		return true;

	for (auto it = my_map.begin(); it != my_map.end(); ++it) {
		if (evristic(it->first, end_ch) > shortest_way(my_map, it->first, end_ch,
std::numeric_limits<size_t>::max(), 0, 0))
			return false;
	}

	return true;
}

int main() {
	char start, end;
	std::cin >> start >> end;
	char a, b;
```

```
	float c = 0;
	std::map<char, elem> my_map;
	while (std::cin >> a >> b >> c) {
		if (c == -1) break;
		my_map[a].ways.push_back({ b,c });
		std::sort(my_map[a].ways.begin(), my_map[a].ways.end(), cmp);
	}

	if (check_ambissibility(my_map, end))
		std::cout << "allowable\n";
	else std::cout << "not allowable\n";
	if(check_monotony(my_map, end))
		std::cout << "monotone\n";
	else std::cout << "not monotone\n";

	findWay(start, end, my_map);
	return 0;
}
```