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 Воропаев А.О.

Преподаватель Фирсов М. А.

Санкт-Петербург

2020

Вариант 4.

Цель работы.

Построение и анализ алгоритма A* на основе на решения задачи о нахождении минимального пути в графе.

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.

Расширение.

Модификация A* с двумя финишами.

Описание алгоритма.

Для решения поставленной задачи был реализован алгоритм A*. Все значения связанные с вершинами графа и необходимые для корректной работы алгоритма находятся в структуре node(буква, отвечающая за название вершины; вектор пар, которые хранят все пути, исходящие из вершины и их длины; длину пути от начала до данной вершины; путь от начала в виде строки из вершин). В качестве эвристической функции был использован перегруженный оператор меньше(<) структуры node. Очередь с приоритетами была реализована на основе вектора(массива), в качестве эвристической функции используется оператор меньше(<) структуры node. Сначала в пустую очередь заносится начальная вершина графа. Затем в начале каждой итерации в массиве ищется элемент, приоритет которого минимален, он удаляется из очереди, а затем в очередь добавляются все вершины, до которых можно дойти по ребрам, выходящим из этой вершины. 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. Сложность алгоритма: O(|V|*|V| + |E|), где V – множество вершин, а E – множество ребер.

Описание основных структур данных и функций.

Структуры.

struct node – структура, хранящая основную информацию о вершинах для работы алгоритма.

char vertex – название вершины

std::vector<std::pair<char, int>> destinations – вектор, хранящий все длины граней и название вершин, в которые они направляются.

double length – длина пути от начала графа до данной вершины.

std::string path – путь от начала графа до данной вершины в виде строки из названий вершин.

Функции.

void console_input/file_input(char& start, int& quantity, std::vector<char>& multiple_ends, std::map<char, node>& dict) – функция для ввода информации из файла.

start – название стартовой вершины.

quantity – кол-во выходов.

multiple_ends – вектор, содержащий все выходы.

dict – словарь, содержащий информацию о всех вершинах, включая все пути выходящие из них и их длину.

A_star_processing(std::vector<char>& multiple_ends, std::map<char, node>& dict, char& start, int& mode) – функция, отвечающая за функционал алгоритма A*.

multiple_ends - вектор, содержащий все выходы.

dict – словарь, содержащий информацию о всех вершинах, включая все пути выходящие из них и их длину.

start – название стартовой вершины.

mode – флаг способа вывода информации(1 – в файл, 2 – в консоль)

Вывод промежуточной информации.

Во время основной части работы алгоритма происходит вывод промежуточной информации а именно, выбранная на данном вершина (вершина с меньшим приоритетом), путь до которой был изменен при помощи ребра выходящего из выбранной вершины. Также выводится скорость работы

алгоритма и его сложность.

Тестирование.

Результаты тестирования представлены в таблице 1.

Таблица 1 – Результаты тестирования

Ввод	**Вывод**
a 2 l h a b 1.000000 a f 3.000000 b c 5.000000 b g 3.000000 f g 4.000000 c d 6.000000 d m 1.000000 g e 4.000000 e h 1.000000 e n 1.000000 n m 2.000000 g i 5.000000 i j 6.000000 i k 1.000000 j l 5.000000 m j 3.000000	Full path to the vertex l : abgenmjl Full path to the vertex h : abgeh
a 2	Full path to the vertex p : abgenmjp

k p a b 1.000000 a f 3.000000 b c 5.000000 b g 3.000000 f g 4.000000 c d 6.000000 d m 1.000000 g e 4.000000 e h 1.000000 e n 1.000000 n m 2.000000 g i 5.000000 i j 6.000000 i k 1.000000 j l 5.000000 m j 3.000000 j p 4.000000	Full path to the vertex k : abgik
a f i a b 0 a c 0 b d 0 c e 0 c s 0 d f 0 d s 0 s i 0 e f 0	Full path to the vertex f : acef Full path to the vertex i : acsi
a 3 e f h a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 c f 2.0 b h 4.0	Full path to the vertex e : ade Full path to the vertex h : abh Full path to the vertex f : abcf

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 <vector>
#include <utility>
#include <queue>
#include <map>
#include <string>


struct node {
    inline bool operator<(const node& a) {
        if(this->length - this->vertex == a.length - a.vertex)
            return this->vertex < a.vertex;

        return this->length - this->vertex > a.length - a.vertex;
    }

    char vertex = 0;
    std::vector<std::pair<char, int>> destinations;
    double length = 0;
    std::string path;
};


int main() {

    char start;
    int quantity = 0;
    std::vector <char> multiple_ends;

    std::cin >> start;
```

```
std::cout << "Set the number of ends\n";
std::cin >> quantity;
for(int i = 0; i < quantity; ++i) {
    char tmp_end;
    std::cin >> tmp_end;
    multiple_ends.push_back(tmp_end);
}


std::map<char, node> dict;

node tmp;
char from, to;
float len;
std::pair<char, int> to_child;

while(std::cin >> from >> to >> len) {
    if(len == -1) {
        break;
    }

    if(dict.find(from) == dict.end()) {
        tmp.vertex = from;
        to_child = {to, len};
        tmp.destinations.push_back(to_child);
        dict[from] = tmp;
    }
    else {
        to_child = {to, len};
        dict[from].destinations.push_back(to_child);
    }
    tmp.destinations.clear();
}
for(auto i : multiple_ends) {
```

```
        dict[i] = {};
        dict[i].vertex = i;
    }

    std::priority_queue<node, std::vector<node>, std::less<>>  priorityQueue;

    priorityQueue.push(dict[start]);
    while(!priorityQueue.empty()) {

        if(multiple_ends.empty())
            break;

        tmp = priorityQueue.top();
        priorityQueue.pop();

        std::cout << "Current vertex with the lowest priority : " << tmp.vertex <<
        "\nPath to the vertex(length) : " << tmp.length <<
        "\nPath to the vertex(path) : " << tmp.path << std::endl;


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

            if (tmp.vertex == multiple_ends[i]) {
                std::cout << "!!!END!!! Full path to the vertex " << tmp.vertex <<
" : " << tmp.path + multiple_ends[i] + '\n';
                multiple_ends.erase(multiple_ends.begin() + i);
                continue;
            }
        }

        std::cout << "_____________________________________________________\n";

        node elem_for_push;
        for(auto i : tmp.destinations) {
```

```
            if(dict.find(i.first) == dict.end())
                continue;

            elem_for_push = dict[i.first];
            elem_for_push.length = i.second + tmp.length;
            elem_for_push.path =  tmp.path + tmp.vertex;
            priorityQueue.push(elem_for_push);
        }
    }
    return 0;
}
```