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 на графах

Студентка гр. 7383 _____ Прокопенко Н.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2019

Цель работы

Исследовать и реализовать задачу построения кратчайшего пути в ориентированном графе помощью метода A*.

Формулировка задачи: необходимо разработать программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A* до заданной 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.

Входные данные: в первой строчке через пробел указываются начальная и две конечные вершины. Далее в каждой строке указываются ребра графа и их вес. Вариант 3м: представить граф в виде матрицы смежности и написать функцию, проверяющую эвристику на допустимость и монотонность.

Реализация задачи

В данной работе для решения поставленной цели был написан класс `Graf` и несколько методов, содержащихся в данном классе. А также написана структура для очереди с приоритетом.

Параметры, хранящиеся в структуре данных `struct Priority`:

• `prior` – приоритет нахождения в очереди;

• `double var_evristic` – разница между ASCII символов;

• `resultat` – путь в индексном виде.

Конструктор класса создает двумерный массив целых чисел, заполняет его нулями, добавляет в вектор `str` начало пути.

Ниже представлены поля класса:

`double **graf` — матрица смежности графа.

`vector<int> str` — вектор, хранящий кратчайший путь.

`double weight_from[N]` — массив, хранящий длины кратчайших путей до вершин.

`double finish_way` — переменная, хранящая значения минимального кратчайшего пути от стартовой вершины.

Далее представлены методы класса:

Метод `void matrix(char i, char j, double weight)` вставляет в массив `graf[i][j]` длину пути из `i в j`.

Метод `min_way_A_star(int first, int finish, priority_queue <Priority> &queue, double way )` который идет по графу, используя эвристическую функцию, и образует очередь с приоритетами. Под эвристической функцией понимается функция вычисляющая сумму разницы между конечной вершиной пути и вершиной, в которую ведет ребро, по которому мы идем, и длины пройденного пути. Используя свойства очереди с приоритетами строиться кратчайший путь для вершины, которая на данном шаге находится ближе. В конце работы метода если кратчайший путь до заданной вершины имеется, то он записывается в вектор `str. Если вектор str` не пуст, то для него вызывается метод проверки эвристики на допустимость и монотонность.

Метод `bool is_monotonic(int finish, int ver1, int ver2)` проверяет две вершины на монотонность (эвристическая функция первой вершины должна быть не больше эвристической функции второй вершины — потомка первой).

Метод `bool is_admissible( int finish, int ver)` проверяет вершину на допустимость (эвристическая оценка пути должна быть не больше длины минимального пути от вершины до конечной вершины).

Метод `void print_path(int first, int finish)` выводит в консоль найденный путь, проверяя эвристику на допустимость и монотонность.

Метод `double evristic(int i, int j)` возвращает эвристическую оценку для вершины, в соответствии с её определением в формулировке задачи.

В главной функции `main()` создается класс для графа и считывается начальная и конечная вершины. Долее в цикле считываются данные из какой

вершины в какую вершину есть путь определенной длины и вызывается метод, заполняющий матрицу. Вызывается метод поиска кратчайшего пути алгоритмом A*.

Тестирование

Программа собрана в операционной системе Ubuntu 17.04 с использованием компилятора g++. В других ОС и компиляторах тестирование не проводилось. Результаты тестирования показали, что поставленная цель выполнена. Результаты тестирования представлены в Приложении Б.

Так же было проведено исследование алгоритма. Функция a_star проходит по смежным ребрам вершины с наименьшей эвристической функцией. В худшем случае могут быть просмотрены все пути данного графа. Тогда сложность зависит от количества ребер и количества вершин графа. В таком случае временную сложность алгоритма можно свести к показательной. Аналогичной будет сложность по памяти, т.к. в худшем случае придется хранить в очереди приоритетов всевозможные пути.

Выводы

В ходе выполнения лабораторной работы была решена задача нахождения кратчайшего пути в графе методом A* на языке C++, и исследован алгоритм A*. Полученный алгоритм имеет сложность показательную как по времени, так и по памяти. Так же была изучена структура данных очередь с приоритетом.

Была написана программа, строящая граф в виде списка смежности, очередь с приоритетом, и вычисляющая кратчайший путь от заданной вершины до конечной, если такой существует.

ПРИЛОЖЕНИЕ А.
КОД ПРОГРАММЫ

lab2.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <queue>
#include <cmath>
#define N 26
using namespace std;
typedef struct Priority{
    vector<int> resultat;
    double prior;
    int var_evristic;
} Priority;

bool operator < (const Priority &com_ver_1, const Priority &com_ver_2){
    return com_ver_1.prior > com_ver_2.prior;
}

class Graf{
private:
    double **graf;
    vector<int> str;
    double weight_from[N];
    double finish_way;
public:
    Graf(int start){
        str.push_back(start);
        graf = new double *[N];
        for(int i = 0; i < N; i++)
            graf[i] = new double[N];
        for(int i = 0; i < N; i++){
            for(int j = 0; j < N; j++)
                graf[i][j] = 0;
        }
    }
    ~Graf(){
        for(int i = 0; i < N; i++)
            delete[] graf[i];
        delete[] graf;
        str.clear();
    }
    void matrix(char i, char j, double weight){
        graf[i][j] = weight;
    }
    void Display(){
        for(int i = 0; i < N; i++)
            for(int j = 0; j < N; j++)
                cout << graf[i][j] << " ";
```

```
            cout << endl;
    }
    void print_path(int first, int finish){
        bool admissible = 1;
        bool monotonic = 1;
        int i = 0;
        while(str[i] != finish){
            admissible = is_admissible(finish, str[i])  && admissible ?
1 : 0;
            i++;
            monotonic = is_monotonic(finish, str[i - 1], str[i]) &&
monotonic ? 1 : 0;
        }
        for(i = 0; i < str.size() ; i++)
            cout << char(str[i] + 'a');
        if(admissible)
            cout << endl << "Эвристика допустима" << endl;
        else
            cout << endl << "Эвристика не допустима" << endl;
        if(monotonic)
            cout << "Эвристика монотонна" << endl;
        else
            cout << "Эвристика не монотонна" << endl;
    }
    double evristic(int i, int j){
        return abs(i - j);
    }
    bool is_admissible(int finish, int ver){
        return finish_way - weight_from[ver] >= evristic(finish, ver);
    }
    bool is_monotonic(int finish, int ver1, int ver2){
        return graf[ver1][ver2] >= evristic(finish, ver1) -
evristic(finish, ver2);
    }
    void min_way_A_star(int first, int finish, priority_queue <Priority>
&queue, double way){
        vector<int> str1;
        weight_from[first] = 0;
        int first_prior = evristic(finish, first);
        while(true){
            for(int j = 0; j < N; j++)
                if(graf[first][j] != 0){
                    Priority elem;
                    elem.var_evristic = evristic(finish, j);
                    elem.prior = graf[first][j] + way +
elem.var_evristic;
                    for(auto i : str1)
                        elem.resultat.push_back(i);
                    elem.resultat.push_back(j);
                    queue.push(elem);
                }
            if(queue.empty())
```

```
                break;
            if(!queue.empty()){
                Priority popped;
                popped = queue.top();
                queue.pop();
                first = popped.resultat[popped.resultat.size() - 1];
                str1 = popped.resultat;
                way = popped.prior - popped.var_evristic;
                first_prior = popped.var_evristic + first_prior -
popped.var_evristic;
                weight_from[first] = way + first_prior;
            }
            if(str1[str1.size() - 1] == finish){
                if(str.size()){
                    for(auto i : str1)
                        str.push_back(i);
                    finish_way = weight_from[first]  ;
                    print_path(str[0], str[str.size() - 1]);
                    break;
                }
                else cout << "нет пути";
            }
        }
    }
};

int main(){
    char start, the_end, from, to;
    double weight;
    cin >> start >> the_end;
    priority_queue <Priority> queue;
    Graf matr(start - 'a');
    while(cin >> from >> to >> weight){
        matr.matrix(from - 'a', to - 'a', weight);
    }
    //  matr.Display();
    matr.min_way_A_star(start - 'a', the_end - 'a', queue, 0);
    return 0;
}
```

ПРИЛОЖЕНИЕ Б.

ТЕСТОВЫЕ СЛУЧАИ

Результаты тестов представлены в табл. 1.

Входные данные	Выходные данные
a f a k 5	нет пути
a l a b 1 a f 3 b c 5 b g 3 f g 4 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	abgenmjl Эвристика допустима Эвристика не монотонна
a z a x 5.0 x y 1.0 x z 1.0 a b 4.0 b z 2.0	axz Эвристика не допустима Эвристика не монотонна
a c a e 1 e c 1	aec Эвристика не допустима Эвристика не монотонна
a c a b 2 b c 1 a c 3	ac Эвристика допустима Эвристика монотонна