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

ОТЧЕТ

по практической работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм A*

Студент гр. 7383 _____ Зуев Д.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Цель работы: ознакомиться с алгоритмом A* поиска кратчайшего пути из одной вершины в другую на примере построения алгоритма для выполнения задачи.

Формулировка задачи: 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. Вариант 3м: представить граф в виде матрицы смежности и написать функцию, проверяющую эвристику на допустимость и монотонность.

Реализация задачи.

Для реализации задачи был создан класс graph.

Ниже представлены поля класса:

`double edges[SIZE][SIZE]` — матрица смежности графа.

`char prev[SIZE]` — массив предыдущих вершин в кратчайшем пути.

`bool open_ver[SIZE]` — массив, хранящий информацию о том, является ли вершина открытой.

`bool closed_ver[SIZE]` — массив, хранящий информацию о том, является ли вершина закрытой.

`double d[SIZE]` — массив, хранящий длины кратчайших путей до вершин.

`char begin` — начальная вершина.

`char end` — конечная вершина.

Далее представлены методы класса:

`void readCons()` — считывает количество путей, начальную и конечную вершины и пути графа с консоли.

`double heuristic(int cur)` — возвращает эвристическую оценку для вершины, в соответствии с её определением в формулировке задачи.

int search_min_f() — находит вершину с минимальной эвристической функцией.

bool is_empty_open() — проверяет, не пустой ли массив открытых вершин.

void a_star() — непосредственно алгоритм A*. Добавляет все смежные вершины, вершины с минимальной эвристической функцией, в массив открытых вершин, а саму вершину удаляет из него и добавляет в массив закрытых вершин. Если массив открытых вершин опустошается не достигнув конца, то выводит соответствующее сообщение. Если закрываемая вершина на данной итерации является конечной, то алгоритм заканчивает свое выполнение и выводит на консоль найденный путь.

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

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

void print_path() — выводит в консоль найденный путь, проверяя эвристику на допустимость и монотонность.

В главной функции main создается класс для графа и последовательно вызываются методы считывания графа с консоли и поиска кратчайшего пути алгоритмом A*.

Код программы представлен в приложении Б.

Исследование сложности алгоритма.

Функция a_star проходит по смежным ребрам вершины с наименьшей эвристической функцией. В худшем случае она пройдет по всем вершинам и, соответственно, по всем ребрам, поэтому сложность функции a_star будет оценена как O(|E| + |V|). Но в функции a_star на каждой итерации вызывается функция поиска вершины с минимальной эвристической функцией,

проходящая по всем вершинам, поэтому итоговая сложность алгоритма будет $O(|E|^2 + |V||E|)$.

Тестирование.

Программа была собрана в компиляторе G++ в среде разработки Qt в операционной системе Linux Ubuntu 17.10.

В ходе тестирования ошибок выявлено не было.

Корректные тестовые случаи представлены в приложении А.

Выводы.

В ходе выполнения данной работы был изучен и реализован алгоритм A* поиска кратчайшего пути в графе. Сложность реализованного алгоритма составляет $O(|E|^2 + |V||E|)$.

ПРИЛОЖЕНИЕ А

ТЕСТОВЫЕ СЛУЧАИ

Входные данные	Выходные данные
1 a f a f 1	af Heuristic is not admissible Heuristic is not monotonic
3 a c a b 2 b c 1 a c 3	ac Heuristic is admissible Heuristic is monotonic
2 a c a e 1 e c 1	aec Heuristic is not admissible Heuristic is not monotonic

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД

```
#include <iostream>
#include <cstdio>
#include <cmath>

#define SIZE 26

using namespace std;

class graph
{
private:
    double edges[SIZE][SIZE];
    char prev[SIZE];
    bool open_ver[SIZE];
    bool closed_ver[SIZE];
    double d[SIZE];
    char begin;
    char end;
public:
    graph()
    {
        for(int i = 0; i<SIZE; i++)
        {
            for(int j = 0; j < SIZE; j++)
                edges[i][j] = 0;
            prev[i] = '\0';
            open_ver[i] = 0;
            closed_ver[i] = 0;
            d[i] = 0;
        }
    }
    void readCons()
    {
        int n;
        cin >> n;
        char f, s;
        cin >> begin;
        cin >> end;
        open_ver[static_cast<int>(begin) - 97] = 1;
        for(int i = 0; i<n; i++)
        {
            cin >> f;
            cin >> s;
            cin >> edges[static_cast<int>(f) - 97]
[static_cast<int>(s) - 97];
        }
    }
    double heuristic(int cur)
```

```
{
    return abs(static_cast<int>(end) - 97 - cur);
}
int search_min_f()
{
    double min = 1000;
    int min_i = 0;
    for(int i = 0; i< SIZE; i++)
    {
        if(d[i] + heuristic(i) <= min && open_ver[i])
        {

            min_i = i;
            min = d[i] + heuristic(i);
        }
    }
    open_ver[min_i] = 0;
    closed_ver[min_i] = 1;
    return min_i;
}
bool is_empty_open()
{
    for(int i = 0; i<SIZE; i++)
    {
        if(open_ver[i] == 1)
        {
            return false;
        }
    }
    return true;
}
void a_star()
{
    while(!is_empty_open())
    {
        int closing = search_min_f();
        if(closing == static_cast<int>(end)-97)
        {
            print_path();
            return;
        }
        for(int i = 0; i<SIZE; i++)
        {
            if(edges[closing][i] != 0)
            {
                if(closed_ver[i])
                    continue;
                if(open_ver[i])
                {
                    if(d[i] > d[closing] + edges[closing][i])
                    {
```

```
                            d[i] = d[closing] + edges[closing][i];
                            prev[i] = static_cast<char>(closing +
97);
                        }
                    }
                    else
                    {
                        open_ver[i] = true;
                        d[i] = d[closing] + edges[closing][i];
                        prev[i] = static_cast<char>(closing + 97);
                    }

                }
            }
        }
        cout<<"Path is not found."<<endl;
    }
    bool is_admissible(int ver)
    {
        return d[static_cast<int>(end) - 97] - d[ver] >=
heuristic(ver);
    }
    bool is_monotonic(int ver1, int ver2)
    {
        return d[ver1] + heuristic(ver1) <= d[ver2] +
heuristic(ver2);
    }
    void print_path()
    {
        string out;
        bool admissible = 1;
        bool monotonic = 1;
        int i = 0;
        out[i] = end;
        while(out[i] != begin)
        {
            admissible = is_admissible(static_cast<int>(out[i]) -
97) && admissible ? 1 : 0;
            out[i+1] = prev[static_cast<int>(out[i]) - 97];
            i++;
            monotonic = is_monotonic(static_cast<int>(out[i])-
97,static_cast<int>(out[i-1])-97) && monotonic ? 1 : 0;
        }
        admissible = is_admissible(static_cast<int>(out[i]) - 97)
&& admissible ? 1 : 0;
        for(i; i>=0 ; i--)
            cout << out[i];
        if(admissible)
            cout<<endl<<"Heuristic is admissible"<<endl;
        else
            cout<<endl<<"Heuristic is not admissible"<<endl;
```

```
        if(monotonic)
            cout<<"Heuristic is monotonic"<<endl;
        else
            cout<<"Heuristic is not monotonic"<<endl;
    }
};

int main()
{
    graph k;
    k.readCons();
    k.a_star();
}
```