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: Алгоритм A*

Студент гр. 7383 _____ Власов Р.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

Содержание

Цель работы

Цель работы: познакомиться с алгоритмом поиска A*, создать программу, осуществляющую поиск кратчайшего пути в графе с помощью алгоритма A*.

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. Вариант 2c: граф представлен в виде списка смежности, эвристическая функция для каждой вершины задается положительным числом во входных данных.

Реализация задачи

Программу было решено писать на языке программирования C++.

Для реализации поставленной задачи были созданы классы adjacency_list и point.

```
class adjacency_list
{
    std::vector<std::tuple<char, char, double, double>> paths;
public:
void add_path(char from, char to, double length, double heuristic);
bool get_path(char start, char finish, std::vector<char> &answer);
double heuristic_fun(std::tuple<char, char, double, double> el):
}
```

Класс adjacency_list отвечает за хранение графа в виде списка смежности, также содержит функцию get_path(char start, char finish, std::vector<char> answer), осуществляющую поиск кратчайшего пути из вершины start в вершину finish с помощью алгоритма A*.

```
struct point
{
    char name;
    double key;
    double way;
    double heuristic;
    std::vector<char> path;
    point(char name, double way, double heuristic, std::vector<char>
path);
    friend bool operator<(const point& l, const point& r)$
};
```

Класс point используется для хранения данных в очереди с приоритетами. Элемент класса содержит имя вершины, значение эвристической функции и длину пути до неё, а также кратчайший путь и значение приоритета.

Исходный код программы представлен в приложении Б.

Исследование алгоритма

Сложность алгоритма составляет O(|V|·|E|), где |V| – количество вершин в графе, |E| - количество ребер в графе. При обработке каждое вершины алгоритм обрабатывает все ребра графа, в наихудшем случае алгоритм обработает каждую вершину.

Тестирование

1. Процесс тестирования

Программа собрана в операционной системе Ubuntu 18.04.2 LTS bionic компилятором g++ version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04). В других ОС и компиляторах тестирование не проводилось.

2. Результаты тестирования

В результате тестирования было выявлено и исправлено некорректное поведение программы в ситуации, когда после обработки вершины в очереди находился только один элемент. Тестовые случаи представлены в приложении А.

Вывод

В ходе выполнения данной работы был изучен метод поиска кратчайшего пути A*. Была написана программа, применяющая метод A* для поиска кратчайшего пути в графе. Сложность реализованного алгоритма составляет O(|V|·|E|).

ПРИЛОЖЕНИЕ А. ТЕСТОВЫЕ СЛУЧАИ

Входные данные	Результат
a e a b 3.0 3.0 b c 1.0 2.0 c d 1.0 1.0 a d 5.0 1.0 d e 1.0 0	ade
a e a b 1.0 1.0 b c 1.0 1.0 c d 1.0 1.0 d b 1.0 1.0 b e 10.0 1.0	abe

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЙ КОД

```
#include <vector>
#include <queue>
#include <tuple>
#include <iostream>

struct point
{
    char name;
    double key;
    double way;
    double heuristic;
    std::vector<char> path;

    point(char name, double way, double heuristic, std::vector<char>
path)
        : name(name), key(heuristic + way), way(way),
heuristic(heuristic), path(path) {}

    friend bool operator<(const point& l, const point& r) {return
l.key > r.key;}
};


class adjacency_list
{
    std::vector<std::tuple<char, char, double, double>> paths;

public:
    void add_path(char from, char to, double length, double heuristic)
    {
        paths.push_back(std::make_tuple(from, to, length, heuristic));
    }
    bool get_path(char start, char finish, std::vector<char> &answer)
    {
        std::priority_queue<point> queue;
        double way = 0;
        std::vector<char> cur_path;
        cur_path.push_back(start);
        do
        {
            for (auto el : paths)
            {
                if (std::get<0>(el) == start)
                {
```

```
                    cur_path.push_back(std::get<1>(el));
                    queue.push(point(std::get<1>(el), way +
std::get<2>(el), heuristic_fun(el), cur_path));
                    cur_path.pop_back();
                }
            }
            if (!queue.empty())
            {
                start = queue.top().name;
                way = queue.top().way;
                cur_path = queue.top().path;
                queue.pop();
                continue;
            }
        }
        while(start != finish);

        answer = cur_path;
        return true;

    }
    double heuristic_fun(std::tuple<char, char, double, double> el)
    {
        return std::get<3>(el);
    }
};

int main()
{
    char from, to;
    double length, heuristic;
    char start, finish;
    std::cin >> start >> finish;
    adjacency_list *head = new adjacency_list();
    while(std::cin >> from >> to >> length >> heuristic)
    {
        head->add_path(from, to, length, heuristic);
    }
    std::vector<char> ans;
    head->get_path(start, finish, ans);
    for (auto c : ans)
        std::cout << c;
    delete head;
    return 0;
}
```