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: Жадный алгоритм и A*

Студент гр. 7304 _____ Шарапенков И.И.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург
2019

Цель работы.

Изучение алгоритмов построения пути в графах (Жадный алгоритм и A*)

Задача.

1. Разработайте программу, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

Пример входных данных

a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0

В первой строке через пробел указываются начальная и конечная вершины

Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

abcde

2. 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.

Пример входных данных

a e
a b 3.0

b c 1.0
c d 1.0
a d 5.0
d e 1.0

В первой строке через пробел указываются начальная и конечная вершины

Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

ade

Описание алгоритма.

1. **Жадный алгоритм**
    1. В качестве текущей вершины выбираем start (вершина из которой необходимо найти путь до вершины end).
    2. Инициализируем общий счет вершины start числом 0. Добавляем start в множество непросмотренных вершин. Общий счет вычисляется по формуле:

       Общий счет = Минимальный найденный на данном шаге вес ребра, ведущего в текущую вершину
    3. Если множество непросмотренных вершин пусто алгоритм завершает работу.
    4. Выбираем вершину с минимальным значением общего счета из множества непросмотренных вершин. Если таких вершин больше одной, то выбираем последнюю добавленную, при этом инцидентные вершины в приоритете (даже если общий счет больше). Помечаем вершину как текущую.
    5. Если текущая вершина равна end, то алгоритм завершает работу.
    6. Удаляем текущую вершину из множества непросмотренных и добавляем в множество просмотренных.

7. Для всех инцидентных вершин: если вершина просмотрена, то переходим к следующей вершине; если вершины нет в множестве просмотренных, то добавляем ее туда; если вершина уже была в множестве просмотренных, то смотрим улучшается ли общий счет, если улучшается, то обновляем общий счет для инцидентной вершины и сохраняем в ассоциативный массив came_from информацию о том из какой вершины переходим.
8. Переходим к шагу 3.

2. A*

1. В качестве текущей вершины выбираем start (вершина из которой необходимо найти путь до вершины end).
2. Инициализируем общий счет вершины start значение эвристической функции, глобальный счет – числом 0. Добавляем start в множество непросмотренных вершин. Общий и глобальный счет вычисляется по формулам:

   Гобальный счет = Минимальный найденный на данном шаге путь от start до текущей вершины

   Общий счет = Глобальный счет + Значение эвристической функции
3. Если множество непросмотренных вершин пусто алгоритм завершает работу.
4. Выбираем вершину с минимальным значением общего счета из множества непросмотренных вершин. Если таких вершин больше одной, то выбираем вершину с меньшей эвристикой. Помечаем вершину как текущую.
5. Если текущая вершина равна end, то алгоритм завершает работу.
6. Удаляем текущую вершину из множества непросмотренных и добавляем в множество просмотренных.
7. Для всех инцидентных вершин: если вершина просмотрена, то переходим к следующей вершине; если вершины нет в множестве

просмотренных, то добавляем ее туда; если вершина уже была в множестве просмотренных, то смотрим улучшается ли глобальный счет, если улучшается, то обновляем общий и глобальный счет для инцидентной вершины и сохраняем в ассоциативный массив came_from информацию о том из какой вершины переходим.

8. Переходим к шагу 3.

Пример работы.

1. **Жадный алгоритм**

   a e
   a b 3.0
   b c 1.0
   c d 1.0
   a d 5.0
   d e 1.

   abcde

2. **A***

   a e
   a b 3.0
   b c 1.0
   c d 1.0
   a d 5.0
   d e 1.0

   ade

Вывод

В ходе лабораторной работы были изучены алгоритмы поиска путей на графах: жадный алгоритм и A*. Был реализован жадный алгоритм и A* на языке C++. Жадный алгоритм выполняет поиск первого найденного пути с помощью перехода по ребрам с наименьшим весом, в то время как A* ищет минимальный путь для заданной эвристической функции, отслеживая общий пройденный путь для каждой вершины, а также используя значение эвристической функции.

Исходный код программы

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


using Vertex = char;
using Edge =  struct Edge {
    Vertex next;
    float cost;

    explicit Edge(Vertex n, float c) : next(n), cost(c) {};
};
using Graph = std::map<Vertex, std::vector<Edge>>;


float heuristic(Vertex v1, Vertex v2) {
    return abs(v1 - v2);
}


std::vector<Vertex> reconstruct(std::map<Vertex, Vertex> came from, Vertex
current) {
    std::vector<Vertex> total_path = { current };
    while(came_from.find(current) != came_from.end()) {
        current = came_from[current];
        total_path.insert(total_path.begin(), current);
    }
    return total_path;
}

using DefInfFloat = struct {
    float val = FLT_MAX;
};

std::vector<Vertex> greedy(Graph &graph, Vertex start, Vertex end) {
    std::vector<Vertex> closed_set, open_set = {start};
    std::map<Vertex, Vertex> came_from;
    std::map<Vertex, DefInfFloat> f_score;

    f_score[start].val = 0;
    while (!open_set.empty()) {
        Vertex current = *std::min_element(open_set.begin(), open_set.end(),
[&f_score, current, &graph](Vertex const &a, Vertex const &b) {
            if(std::find_if(graph[current].begin(), graph[current].end(),
[a](Edge const &e) {
                return e.next == a;
            }) != graph[current].end() && std::find_if(graph[current].begin(),
graph[current].end(), [b](Edge const &e) {
                    return e.next == b;
            }) == graph[current].end()) {
                return true;
            }
            return f_score[a].val < f_score[b].val;
        });

        if(current == end)
            return reconstruct(came_from, current);

        open_set.erase(std::remove(open_set.begin(), open_set.end(), current),
```

```
open_set.end());
        closed_set.push_back(current);

        for (auto const &neighbor: graph[current]) {
            if(std::find(closed_set.begin(), closed_set.end(), neighbor.next) !=
closed_set.end())
                continue;

            if(!(std::find(open_set.begin(), open_set.end(), neighbor.next) !=
open_set.end()))
                open_set.push_back(neighbor.next);
            else if(neighbor.cost >= f_score[neighbor.next].val)
                continue;

            came_from[neighbor.next] = current;
            f_score[neighbor.next].val = neighbor.cost;

        }
    }
}


std::vector<Vertex> astar(Graph &graph, Vertex start, Vertex end) {
    std::vector<Vertex> closed_set, open_set = {start};
    std::map<Vertex, Vertex> came_from;
    std::map<Vertex, DefInfFloat> g_score, f_score;

    g_score[start].val = 0;
    f_score[start].val = heuristic(start, end);
    while (!open_set.empty()) {
        Vertex current = *std::min_element(open_set.begin(), open_set.end(),
[&f_score, end](Vertex const &a, Vertex const &b) {
            if(f_score[a].val == f_score[b].val) return  heuristic(a, end) <
heuristic(b, end);
            return f_score[a].val < f_score[b].val;
        });

        if(current == end)
            return reconstruct(came_from, current);

        open_set.erase(std::remove(open_set.begin(), open_set.end(), current),
open_set.end());
        closed_set.push_back(current);

        for (auto const &neighbor: graph[current]) {
            if(std::find(closed_set.begin(), closed_set.end(), neighbor.next) !=
closed_set.end())
                continue;

            float tentative_g_score = g_score[current].val + neighbor.cost;

            if(!(std::find(open_set.begin(), open_set.end(), neighbor.next) !=
open_set.end()))
                open_set.push_back(neighbor.next);
            else if(tentative_g_score >= g_score[neighbor.next].val)
                continue;

            came_from[neighbor.next] = current;
            g_score[neighbor.next].val = tentative_g_score;
            f_score[neighbor.next].val = g_score[neighbor.next].val +
heuristic(neighbor.next, end);

        }
```

```
    }

}



int main() {
    Graph graph;
    Vertex start, end, v1, v2;
    float cost;

    std::cin >> start >> end;

    while (std::cin >> v1 >> v2 >> cost)
        graph[v1].push_back(Edge(v2, cost));

    std::vector<Vertex> path = astar(graph, start, end);

    for (auto const &vertex: path) {
        std::cout << vertex;
    }

    return 0;
}
```