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 А*

Студент гр. 8304	___________________	Хотяков Е.П.
Преподаватель	___________________	Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Ознакомиться с жадным алгоритмом на графе и алгоритмом А*, научиться оценивать временную сложность алгоритма и применять его для решения задач.

Постановка задачи.

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.

Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом А*. 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.

Индивидуализации для лаб. работы № 2:

Вар. 1. В A* вершины именуются целыми числами.

Описание жадного алгоритма.

Изначально рассматриваем стартовую вершину. Далее на каждом шаге рассматриваются вершины, в которые можно попасть напрямую из текущей. Выбирается вершина, расстояние до которой от текущей наименьшее. Выбранная вершина становится текущей и помечается как рассмотренная. Если из текущей вершины не существует путей в еще не рассмотренные вершины, происходит возврат в вершину, из которой был совершен переход в текущую. Алгоритм заканчивает работу, когда текущей вершиной становится конечная, либо, когда все вершины были рассмотрены, а конечная так и не была достигнута.

Сложность алгоритма:

По времени: O(|V|+|E|), т.к. нужно просмотреть все ребра и найти ребро минимального веса.

Описание алгоритма A*.

Стартовая вершина помечается «открытой». Пока существуют открытые вершины:

1) Текущая вершина – открытая вершина, с наименьшей полной стоимостью.
2) Если текущая вершина конечная – алгоритм заканчивает работу.
3) Текущая вершина становится закрытой.
4) Для каждого незакрытого ребенка текущей вершины:
   - Рассчитывается функция пути для этой вершины.
   - Если вершина еще не открыта или рассчитанная функция меньше функции, рассчитанной для этой вершины ранее, рассчитанная функция становится функцией этой вершины, вершина-предок запоминается, как вершина, из которой совершен переход в ребенка. (необходимо для восстановления пути в будущем)

Если открытых вершин не осталось, а до конечной маршрут так и не был проложен, алгоритм заканчивает работу (пути не существует).

Сложность алгоритма:

Временная сложность алгоритма A* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$|h(x) – h^*(x)| <= O(\log h^*(x))$

где h* — оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка h(x) не должна расти быстрее, чем логарифм от оптимальной эвристики.

Описание функций и структур данных.

Для описания вершин графа используется структура вершин Vertex:

Int name – имя вершины(целое число)

Int prev – номер вершины-родителя в векторе вершин

Float g – длина пути до вершины

Int h – значение эвристической функции для вершины

Float f – метка вершина(сумма g и h).

`int calcH(const int& cur, const int& end)` – функция вычисления эвристической функции для вершины cur.

`void restorePath(Vertex end, std::vector<Vertex> &Vertex_arr)` – функция вывода результирующего пути.

`void findWay(std::vector<std::pair<int, std::pair<int, float>>> &edges, int &start, int &end)` – функция, реализующая алгоритм А. Функция принимает вектор с введенными данными, стартовую и конечную позицию.

`void addToQueue(std::priority_queue<Vertex, std::vector<Vertex>, std::greater_equal<Vertex>> &open, std::vector<std::pair<int, std::pair<int, float>>> &edges, std::vector<int> &close, Vertex &cur, int &end, std::vector<Vertex> &Vertex_arr)` – функция вставки элемента в очередь с открытыми элементами.

Тестирование программы.

Ввод	Вывод алгоритма
-2 9 -2 -1 1 -2 3 3 -1 0 5 -1 4 3 3 4 4 0 1 6 1 10 1 4 2 4 2 5 1 2 11 1 11 10 2 4 6 5 6 7 6 6 8 1 7 9 5 10 7 3	-2 -1 4 2 11 10 7 9
1 5 1 2 3	No way!

2 3 1 3 4 1 1 4 5 5 6 1	
-5 4 -5 -4 1 -4 -3 1 -3 -2 1 -2 -1 1 -1 4 1 -5 0 1 0 1 1 1 2 1 2 3 1 3 4 1	-5 0 1 2 3 4

Выводы.

В ходе выполнения лабораторной работы были реализованы жадный алгоритм и алгоритм A*, дана оценка времени работы алгоритмов.

Приложение.
Код работы.

main.cpp:

```
#include <iostream>
#include <vector>
#include <string>
#include <queue>
#include <cmath>

int calcH(const int &cur, const int &end)
{
    return abs(cur - end);
}

struct Vertex
{
    int prev;
    int name;
    float f, g;
    int h;
    Vertex(int name, float g, int end) : name(name), g(g)
    {
        h = calcH(name, end);
        f = g + h;
    }
};

bool operator<=(const Vertex &a, const Vertex &b)
{
    return a.f <= b.f;
}
bool operator>=(const Vertex &a, const Vertex &b)
{
    return a.f >= b.f;
}
bool operator==(const Vertex &a, const Vertex &b)
{
    return a.name == b.name;
}

void     readData(std::vector<std::pair<int,     std::pair<int,
float>>> &edges)
{
    int edge_s, edge_f;
    float edge_l;
    while (std::cin >> edge_s >> edge_f >> edge_l)
    {
        edges.push_back(std::make_pair(edge_s,
std::make_pair(edge_f, edge_l)));
    }
```

```
    }

    bool inClose(const std::vector<int> &close, int name)
    {
        for (int i = 0; i < close.size(); i++)
            if (name == close[i])
                return true;
        return false;
    }

    void addToQueue(std::priority_queue<Vertex, std::vector<Ver-
tex>, std::greater_equal<Vertex>> &open, std::vector<std::pair<int,
std::pair<int, float>>> &edges, std::vector<int> &close, Vertex &cur,
int &end, std::vector<Vertex> &Vertex_arr)
    {
        for (int i = 0; i < edges.size(); i++)
        {
            if (cur.name == edges[i].first)
                if (inClose(close, edges[i].second.first))
                    continue;
                else
                {
                    Vertex x(edges[i].second.first, edges[i].sec-
ond.second + cur.g, end);
                    x.prev = Vertex_arr.size() - 1;
                    open.push(x);
                }
        }
    }

    void restorePath(Vertex end, std::vector<Vertex> &Vertex_arr)
    {
        Vertex cur = end;
        std::vector<int> res;
        while (1)
        {
            if (cur.prev == -1)
            {
                res.push_back(cur.name);
                break;
            }
            res.push_back(cur.name);
            cur = Vertex_arr[cur.prev];
        }
        for (int i = res.size() - 1; i >= 0; i--)
            std::cout << res[i] << ' ';
    }

    void findWay(std::vector<std::pair<int, std::pair<int, float>>>
&edges, int &start, int &end)
    {
        std::vector<int> close;
```

```
        std::priority_queue<Vertex,            std::vector<Vertex>,
std::greater_equal<Vertex>> open;
        std::vector<Vertex> Vertex_arr;
        Vertex cur(start, 0, 0);
        cur.prev = -1;
        open.push(cur);
        while (cur.name != end && !open.empty())
        {
            cur = open.top();
            if (inClose(close, cur.name))
            {
                open.pop();
                continue;
            }
            Vertex_arr.push_back(cur);
            open.pop();
            addToQueue(open, edges, close, cur, end, Vertex_arr);
        }
        if (open.empty())
        {
            std::cout << "No way!";
            return;
        }
        restorePath(cur, Vertex_arr);
    }

    int main()
    {
        int start, end;
        std::cin >> start >> end;

        std::vector<std::pair<int, std::pair<int, float>>> edges;
        readData(edges);
        findWay(edges, start, end);

        return 0;
    }
```