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. Жадный алгоритм и A*

Студентка гр. 0382 _____ Кривенцова Л.С.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург

2022

Цель работы.

Изучить принцип работы алгоритмов на графах. Применить знания на практике, решив поставленную задачу.

Основные теоретические положения.

Жадный алгоритм — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Поиск A* (произносится «А звезда» или «А стар», от англ. A star) — в информатике и математике, алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как f(x)). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как g(x) и может быть как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).

Функция h(x) должна быть допустимой эвристической оценкой, то есть не должна переоценивать расстояния к целевой вершине. Например, для задачи маршрутизации h(x) может представлять собой расстояние до цели по прямой линии, так как это физически наименьшее возможное расстояние между двумя точками.

Задание.

Жадный алгоритм.

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

Алгоритм A*.

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

Вар. 2. В A* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных.

Выполнение работы.

Жадный алгоритм:

Структура ребра

struct edge{

char r1; - вершина из которой идёт ребро.

char r2; - вершина, заканчивающая ребро.

float weight; - расстояние между вершинами.

};

vector<edge> current = {}; - массив текущего пути.

vector<edge> mass = {}; - массив рёбер (граф).

char start, finish; - символы начала и конца пути.

bool continiud = true; - логическая переменная, означающая нужно ли продолжать поиск пути.

Функция *mysort*() сортирует массив рёбер пузырьком.

Функция *Collision(edge point)* проверяет, есть ли рассматриваемая вершина в уже найденном пути. Если есть, возвращает *false*, иначе *true*.

int recurs(char node) – функция рекурсивного обхода графа. В качестве аргумента принимается наименование вершины, от которой нужно искать путь. Если эта вершина и есть пункт назначения, то рекурсия заканчивается. Иначе: записываем вершину, рёбра из которой будем рассматривать (это последняя точка

пути, а есть путь ещё пуст – то это стартовая вершина). В цикле *while* доходим до первого в списке ребра, идущего из вершины. Если такое ребро найдено, оно проверяется на коллизию (есть ли это ребро в найденном пути), и если всё хорошо, ребро записывается в путь и запускается рекурсия (поиск пути) от крайней вершины найденного ребра. Если ребра, идущего из рассматриваемой вершины нет в графе (в цикле просмотрен весь граф, и такого ребра не нашлось), а конечная вершина ещё не достигнута, значит путь выбран неверно. В таком случае из пути удаляется последнее ребро, и из предыдущей последней вершины ищется новый вариант пути (массив рёбер просматривается дальше).

Таким образом, в массиве текущего пути к концу рекурсии будет проложен путь. Если пути нет, то и массив окажется пуст.

Жадность алгоритма поддерживается тем, что выбор ребер в графе идёт поочередно, а сам граф сортируется в начале алгоритма – то есть сначала всегда просматривается и выбирается самое короткое ребро.

Сложность по времени — O(|E|), т. к. в худшем случае придется обойти весь граф, при чем каждое ребро будет посещена не более одного раза.

В функции *main*() производится считывание данных, вызываются функции сортировки и нахождения пути, затем данные выводятся на экран.

Алгоритм A*:

Алгоритм сохраняет граф в виде вектора структур вершин.

Описание функций и структур данных:

Структура edge – структура узла графа (одной вершины):

- *char letter;* - буквенное обозначение вершины;
- *vector <pair<char,float> > neighbours = {};* - массив соседних вершин – пар [обозначение, расстояние от вершины до этой соседней];
- *float g = 0;* - расстояние от начала пути до вершины;
- *float f = 0;* - сумма расстояния от начала + эвристический вес;
- *float h = 0;* - значение эвристической функции для вершины.
- *bool open = false;* - логическая переменная, отвечающая за нахождение вершины в очереди;

- *bool closed = false*; - логическая переменная, отвечающая за то, пройдена ли вершина;
- *int from = -1;* - индекс вершины, из которой проложен путь в нынешнюю (из какой вершины пришли в нынешнюю).

vector<edge> graph = {}; - массив вершин в графе.

vector<int> closed; - массив пройденных вершин;

vector<int> open; - очередь вершин, через которые возможен дальнейший путь;

Переменные *int start, finish; char char_start, char_finish;* - индексы и буквенные представления начальной и конечной вершин.

Функция *int find(char lttr)* производит поиск вершины в графе. Принимает символьное обозначение, возвращает индекс вершины в массиве или -1, а случае, когда вершины нет в графе.

Функция *void delete_open(int index)* удаляет элемент массива open с заданным индексом.

Функция *int min_f()* ищет вершину в очереди с минимальным весом (расстояние от старта + значение эвристической функции). Проходит массив *open*. Возвращает индекс вершины в массиве графа, найденный стандартным алгоритмом поиска наименьшего числа в массиве. Если очередь пустая, функция возвращает 0.

Функция int main().

vector<char> result; - массив символов, в котором хранится кратчайший путь в виде наименований вершин.

После считывания данных о начальной и конечной вершинах, в цикле считываются данные о вершинах. Проверяется, если вершины нет в графе, данные о ней формируются в стуктуру, которая добавляется в массив вершин. Если она уже есть в графе, то в уже существующую структуру (элемент массива графа) добавляется информация о новом соседе. После, считываются значения эвристической функции для вершин.

Вызывается функция *a_star*, которая возвращает индекс конечной вершины пути. Если пути нет, то *a_star* возвращает -1 и функция main завершается. Иначе, путь «раскручивается» с конца, и массив *result* заполняется, ставя в начало буквенное обозначение вершины, индекс которой хранится в поле from рассматриваемой вершины. Так, в цикле массив заполняется до тех пор, пока поле *from* рассматриваемой вершины положительно.

Функция a_star реализует алгоритм A*.

В очередь заносится стартовая вершина, её расстояние от начала равно нулю, а вес высчитывается как эвристическая функция.

Пока в очереди есть вершины, продолжается цикл. Если очередь оказывается пустой, значит пути нет и функция возвращает значение -1.

В цикле выбирается текущая вершина, методом определения вершины из с наименьшим весом (находит такую вершину функция *min_f*). Так, в *current* хранится индекс текущей вершины. Если она и есть конечная, то функция возвращает это значение. Вершина удаляется из очереди и заносится в массив пройденных вершин. Соответственно меняются поля элемента структуры open и *closed*.

Запускается подцикл, перебирающий соседей текущей вершины. Для рассматриваемой соседней вершины рассчитывается (но не записывается в данные структуры) расстояние от начала до неё. Если вершина не находится в очереди, или рассчитанное расстояние меньше расстояния, записанного в данных структуры, то заполняются поля элемента структуры, представляющего собой рассматриваемого «соседа» (поля *from, f, g*). Если рассматриваемая соседняя вершина не находится в очереди (и она не пройдена), встаёт в очередь. Итерация цикла завершается.

Временная сложность алгоритма A* зависит от эвристики.

Задание по вариантам.

Эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных следующим образом: считывается граф, после чего в цикле происходит обход вершин, и в каждой итерации пользователю сообщается

вершина, эвристику которой необходимо задать, и значение считывается с клавиатуры. Пример:

a d 4

a b 1.0

b c 2.0

c d 3.0

d a 1.0

Heuristics for vertex a:

>>1

Heuristics for vertex b:

>>2

Heuristics for vertex c:

>>3

Heuristics for vertex d:

>>4

abcd

Тестирование.

Тестирование проводилось с помощью встроенной библиотеки <*cassert*>. Результат не отображён в таблицах, так как команда *assert* предназначена для отлова ошибок, и выводит результат на экран только в случае провала теста.

Были написаны 5 проверяющий функций.

Testfind() – для проверки функции *find*(). Тест создаёт новые исходные данные, задавая граф. И передает *assert* результат функции и предполагаемый (верный) результат. При несовпадении на экран выводится сообщение. Пример:

Assertion failed: find('e') == 5, file

C:/Users/Serg/CLionProjects/untitled1/main.cpp, line 39

Тест *Test_delete()* проверяет функцию *delete_open*() следующим образом: Он поочередно удаляет элементы из очереди. Перед и после удаления идёт проверка с помощью функции вектора *find*. Соответственно, до удаления

функция должна вернуть указатель на элемент, а после – указатель на окончание вектора.

Тест *Test_h()* проверяет вычисление эвристической функции для задания на stepik. Функции h передаётся два символа, а *assert* сравнивает результат h и настоящую разницу позиций символов по таблице ASCII.

В функции *Test_minf*() задаётся граф и очередь, после чего функции *assert* передаётся утверждение, что результат функции min_f() равен верному ответу.

Функция *Test_a*() проверяет главный алгоритм A*. Тест состоит в том, что задаётся граф и начальные данные (начало и конец пути, верный результат). Получив ответ функции *a_star()* результат алгоритма «раскручивается» и путь записывается в переменную – вектор. Функции *assert* передаётся сравнение полученного и истинного результата алгоритма.

В конце каждого теста есть вывод строки – результата. Ведь если *assert* не вызвал ошибку, то программа выполняется дальше, и печатается строка, что определённый тест пройден. Пример:

Function find: Test OK

Function heuristic: Test OK

Function delete_open: Test OK

Function min_f: Test OK

Function a_star: Test OK

Выводы.

В результате работы была написана программа, решающая поставленную задачу при использовании изученных теоретических материалов. Программа было протестирована, результаты тестов удовлетворительны.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Жадный алгоритм.:

```
#include <iostream>
#include <vector>
#include <cstring>
using namespace std;

struct edge{
    char r1;
    char r2;
    float weight;
};

vector<edge> current = {};
vector<edge> mass = {};
char start, finish;
bool continiud = true;

bool Collision(edge point){
    for (int i = 0; i < current.size(); i++){
        if (point.r1 == current[i].r1 &&
                point.r2 == current[i].r2 &&
                point.weight == current[i].weight)
            return false;
    }
    return true;
}

void mysort(){
    edge temp;
    for (int i = 0; i < mass.size() - 1; i++) {
        for (int j = 0; j < mass.size() - i - 1; j++) {
            if (mass[j].weight > mass[j + 1].weight) {
                temp = mass[j];
                mass[j] = mass[j + 1];
                mass[j + 1] = temp;
            }
        }
    }
}

int recurs(char node) {
    float k = 0; char node1 = start;
    if (node == finish) {
        continiud = false;
        return 0;
    }
    else {
    if (current.size() != 0)
```

```
        node1 = current.back().r2;
    while (node1 != finish && k < mass.size() && continiud) {
        while (k < mass.size() && (mass[k].r1 != node)) {
            k++;
        }
        if (k < mass.size()) {
            if (Collision(mass[k])) {
                current.push_back(mass[k]);
                recurs(current.back().r2);
            }
            k++;
        }
        if (k >= mass.size() && continiud) {
            current.pop_back();
        }
    }
    }
}

int main() {
    std::cin >> start >> finish;
    while (std::cin){
        edge a;
        std::cin >> a.r1 >> a.r2 >> a.weight;
        if (mass.size() > 0 && ( a.weight < mass.front().weight))
        {
            mass.emplace(mass.begin(), move( a ));
        }
        else mass.push_back(a);
    }
    mysort();
    recurs(start);
    if (current.size() != 0) {
        if (current.back().r1 != finish &&  current.back().r2 != finish)
            current.clear();
        else {cout << current.front().r1;
            for (int i = 0; i <current.size() ;i++)
                cout << current[i].r2 ;
        }

    }

    return 0;
}
```

A*:

```
#include <iostream>
#include <vector>
#include <cstring>
#include <math.h>
#include <cassert>
```

```
using namespace std;


struct edge{
    char letter;
    float g = 0; // Расстояние от начала до вершины
    float f = 0; // Расстояние + эвристический вес
    float h = 0;
    vector <pair<char,float> > neighbours = {};
    bool open = false; //Находится ли вершина в очереди
    bool closed = false; // Пройдена ли вершина
    int from = -1;
};

vector<edge> graph = {}; //Список вершин
vector<int> closed; //Список пройдённых вершин
vector<int> open; // Очередь вершин
int start, finish;
char char_start, char_finish;

int find( char lttr){
    for (int i = 0; i < graph.size(); i++){
        if (graph[i].letter == lttr)
            return i;
    }
    return -1;
}

void Testfind(){
    graph.erase(graph.begin(), graph.end());
    graph = {{'a'},{'b'},{'c'},{'d'}};
    assert(find('k') == -1);
    graph.push_back({'e'});
    assert(find('e') == 4);
    assert(find('c') == 2);
    assert(find('a') == 0);
    cout << endl << "Function find: Test OK" << endl;
}

void delete_open( int index){
    for (int i = 0; i < open.size(); i++){
        if (open[i] == index)
            open.erase(open.begin() + i);
    }
}

void Test_delete(){
    int test = 0, i = 0;
    while (open.size() > 0){
        //i = open.size() - 1;
        test = open[i];
```

```
        assert(std::find(open.begin(),
                         open.end(), test) != open.end());
        delete_open(test);
        assert(std::find(open.begin(),
                         open.end(), test) == open.end());
    }
    cout << endl << "Function delete_open: Test OK" << endl;
}

float h(char a, char b){ // Эвристическая функция
    return abs(static_cast<float>(a) - static_cast<float>(b));
}

void Test_h(){
    assert(h('x','y') == 1);
    assert(h('a','z') == 25);
    assert(h('e','e') == 0);
    assert(h('f','b') == 4);
    cout << endl << "Function heuristic: Test OK" << endl;
}

int min_f(){
    if (open.empty()) return -1;
    float min = graph[open[0]].f;
    int index_min = open[0];
    for(int i = 0; i < open.size(); i++){
        if (graph[open[i]].f < min ||
            ( graph[open[i]].f == min &&
              static_cast<float>(graph[open[i]].letter) <
              static_cast<float>(graph[open[i]].letter))) {
            min = graph[open[i]].f;
            index_min = open[i];
        }
    }
    return index_min;
}

void Test_minf(){
    open.erase(open.begin(), open.end());
    assert(min_f() == -1);

    graph = {{'a',0, 1, 1}};
    open = {0};
    assert(min_f() == 0);

    graph = {{'a',0, 1, 1},
             {'b', 1, 2, 1},
             {'c', 2, 3, 1},
             {'d', 3, 4, 1},
            {'e', 4, 4, 0},
             {'f', 2, 5, 3},
```

```
            {'g', 8, 8, 0},
            {'h',0, 1, 1}};
    open = {1,2,3,4};
    assert(min_f() == 1);
    open = {3,4};
    assert(min_f() == 3);
    open = {1,2,3,4,5,6,7};
    assert(min_f() == 7);
    cout << endl << "Function min_f: Test OK" << endl;
}



int a_star(){
    int current;
    pair<char,float> neighbour;
    open.push_back(start);
    graph[start].open = true;
    graph[start].g = 0;  graph[start].f =  graph[start].g
                                           + graph[start].h;
    while (open.size() >= 0){
        current = min_f();
        if (current == finish) return current;
        delete_open(current);
        graph[current].open = false;
        closed.push_back(current);
        graph[current].closed = true;
        int count_neigh = 0;
        while (graph[current].neighbours.size() > count_neigh){
            neighbour = graph[current].neighbours[count_neigh];
            int neigh_index = find(neighbour.first);
                float temp_g = graph[current].g + neighbour.second;
                if (!graph[neigh_index].open && !graph[neigh_index].closed ||
temp_g < graph[neigh_index].g){
                    graph[neigh_index].from = current;
                    graph[neigh_index].g = temp_g;
                    graph[neigh_index].f = graph[neigh_index].g +
graph[neigh_index].h;
                }
                if (!graph[neigh_index].open && !graph[neigh_index].closed) {
                    graph[neigh_index].open = true;
                    open.push_back(neigh_index);
                }

            count_neigh++;
        }
    }
    return -1;
}

void Test_a(){
    vector<char> result;
    vector<char> right_result;
```

```
    closed = {}; //Список пройдённых вершин
    open = {}; // Очередь вершин
    start = 0;
    finish = 3;
     char_start = 'a';
     char_finish = 'd';
    graph = {{'a',0, 0, 1,
              {{'b', 1}}},
             {'b', 0, 0, 2,
                     {{'c', 2}}},
             {'c', 0, 0, 3,
                     {{'d', 3}}},
             {'d', 0, 0, 4,
                     {{'a', 1}}}};
    right_result = {'a','b','c','d'};
   int goal = a_star();
    if ( goal < 0 )  result = {};
    else {result.emplace(result.begin(), graph[goal].letter);
        while(graph[goal].from >= 0){
            goal = graph[goal].from;
            result.emplace(result.begin(), graph[goal].letter);}
}
    assert(result == right_result);

    result = {};
    closed = {}; //Список пройдённых вершин
    open = {}; // Очередь вершин
    start = 0;
    finish = 5;
    char_start = 'n';
    char_finish = 'a';
    graph = {{'n',0, 0, 20,
                     {{'m', 1}, {'c', 20}}},
             {'m', 0, 0, 19,
                     {{'l', 1}}},
             {'l', 0, 0, 18,
                     {{'c', 1}}},
             {'c', 0, 0, 2,
                     {{'b', 1}}},
             {'b', 0, 0, 1,
                     {{'a', 20}}},
             {'a', 0, 0, 0,
                     {}}};
    right_result = {'n', 'm', 'l', 'c', 'b', 'a'};
    goal = a_star();
    if ( goal < 0 )  result = {};
    else {result.emplace(result.begin(), graph[goal].letter);
        while(graph[goal].from >= 0){
            goal = graph[goal].from;
            result.emplace(result.begin(), graph[goal].letter);}
    }
    assert(result == right_result);
```

```
    cout << endl << "Function a_star: Test OK" << endl;
}


int main() {
    vector<char> result;
    int count;
    std::cin >> char_start >> char_finish >> count;
    for (int m = 0; m < count; m++){
        edge node;
        std::cin >> node.letter;
        char node2; float distance;
        std::cin >> node2 >> distance;
        int findletter = find(node.letter);
        if (findletter >= 0) {

graph[findletter].neighbours.emplace_back(make_pair(node2,distance));
        }
        else {
            node.neighbours.emplace_back(make_pair(node2,distance));
            graph.push_back(node);
        }
        int findletter2 = find(node2);
        if (findletter2 < 0) {
            edge new_node;
            new_node.letter = node2;
            graph.push_back(new_node);
        }
    }
    for (int l = 0; l < count; l++){
        cout << "Heuristics for vertex " << graph[l].letter << ":\n";
        cin >> graph[l].h;
    }
    start = find(char_start);
    finish = find(char_finish);
    int goal = a_star();
    if ( goal < 0 ) return 0 ;
    result.emplace(result.begin(), graph[goal].letter);
    while(graph[goal].from >= 0){
        goal = graph[goal].from;
        result.emplace(result.begin(), graph[goal].letter);
    }
    for (int loop = 0; loop < result.size(); loop++)
        cout << result[loop];
    Testfind();
    Test_h();
    Test_delete();
    Test_minf();
    Test_a();
    return 0;
}
```