**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**

Вариант 8

|  |  |  |
| --- | --- | --- |
| Студент гр. 8383 |  | Шишкин И.В. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2020

**Цель работы.**

Ознакомиться и разработать программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом А\*.

**Постановка задачи.**

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

Вар. 8. Перед выполнением А\* выполнять предобработку графа: для каждой вершины отсортировать список смежных вершин по приоритету.

**Описание алгоритма.**

A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. В начале работы просматриваются узлы, смежные с начальным: выбирается тот из них, который имеет минимальное значение , где g(x) – функция стоимости достижения рассматриваемой вершины x из начальной, a h(x) – эвристическая оценка расстояния от рассматриваемой вершины к конечной. После чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа – множеством частных решений, – которое размещается в очереди с приоритетом. Приоритет пути определяется по значению f(x). Алгоритм продолжает свою работу до тех пор, пока значение f(x) целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока весь граф не будет просмотрен. Из множества решений выбирается решение с наименьшей стоимостью.

**Описание способов хранения частичных решений.**  struct Edge {

char connectedNode;

int heruisticNum;

double weight;

};

Структура для хранения ребра графа. connectedNode - смежный узел, heruisticNum - эвристическое значение смежного узла, weight - вес ребра.

struct Graph {

char name;

vector <Edge> edge;

};

Структура для хранения графа. name - название узла, edge - вектор смежных узлов.

struct AStar {

char node;

char prev;

double f;

double g;

double h;

bool checked = false;

};

Структура для метода А\*. node - название узла, prev - название предыдущего узла (с которым у узла node есть смежное ребро), f - значение f = g + h, необходимое для вычислений, checked - если узел уже был проверен, то это значение устанавливается в true.

**Описание функции heruisticFunc.**

Функция int heuristicFunc(char from, char to) - эвристическая функция, возвращает эвристическое число узла.

**Описание функции cmpEdge.**

Функция bool cmpEdge(Edge i, Edge j) - компаратор для сортировки ребер по эвристическому значению.

**Описание рекурсивной функции aStar.**

Функция void aStar(char start, char finish, vector <Graph> gr, string nodeNames) - на вход подается start - стартовая вершина, из которой нужно найти минимальный путь до вершины finish; gr - вектор, хранящий в себе все ребра графа; nodeNames - названия всех узлов в графе.

В начале создается vector <AStar> nodes, в котором и будут храниться все данные, необходимые для вычисления минимального пути методом А\*. В этот вектор записывается первая вершина start. Далее, из переменной nodeNames берутся названия всех узлов и записываются в nodes. По сути создается таблица, показанная на рис. 1 (все значения подобраны случайно).

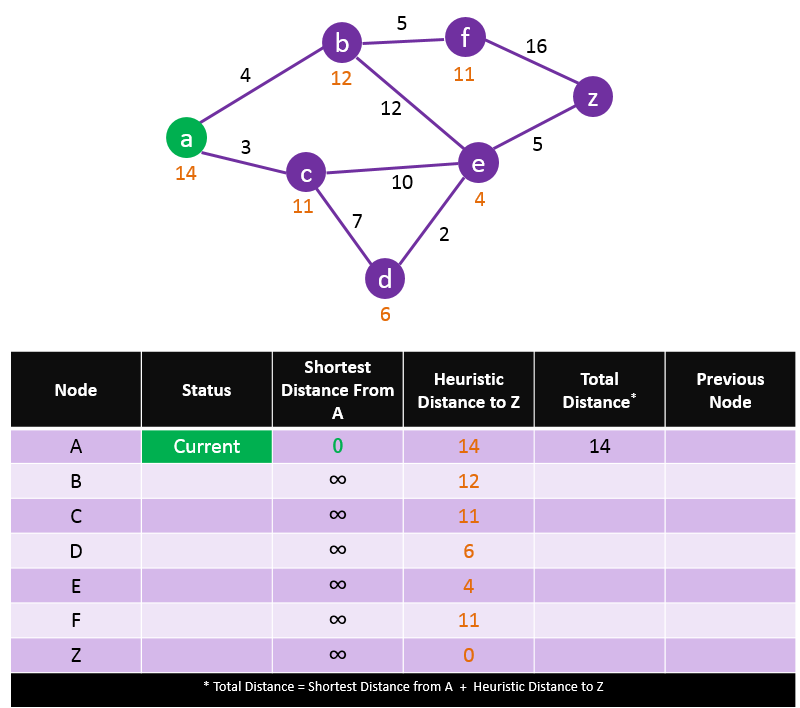


Рисунок 1 – Таблица для метода А\*

Затем идет цикл, в котором в начале ищется в nodes минимальное значение nodes.f (если есть несколько узлов с одинаковым значением nodes.f, то берется то, у которого nodes.h меньше), этот узел запоминается и значению checked присваивается true. Далее, для найденного узла ищутся смежные узлы. Для каждого смежного узла вычисляется значение (расстояние от стартовой вершины до этой) - nodes[index].g + gr[i].edge[0].weight. Если это значение меньше nodes[tmp].g, которое уже хранится в векторе nodes, то значения nodes.g и nodes.f для этой вершины меняются.

Таким образом, таблица будет заполняться, пока текущее значение в nodes не будет окончательным (=finish). Так как в таблице производилось хранение предыдущего узла (prev), то с помощью него можно восстановить путь от начальной вершины до конечной.

**Сложность алгоритма по времени.**

Сложность алгоритма по времени можно оценить как

Так как в худшем случае проверяются все узлы N и все смежные ей вершины N-1.

**Сложность алгоритма по памяти.**

Сложность алгоритма по памяти можно оценить как

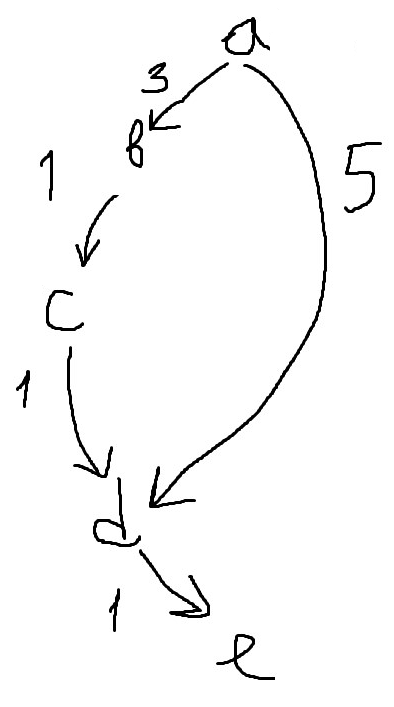
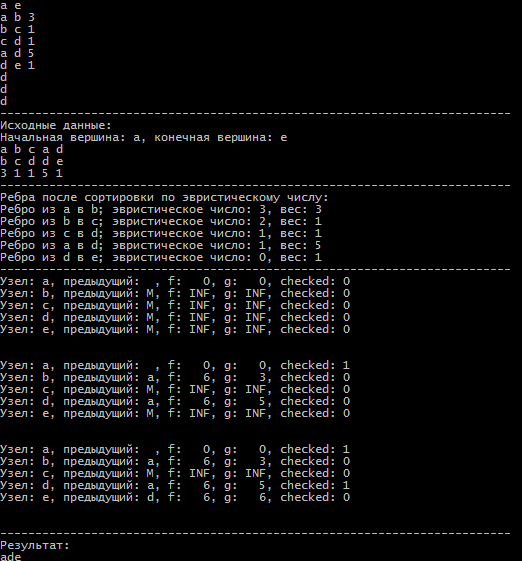
Такая оценка исходит из того, что программа хранит все вершины и все ребра, а так же есть очередь, в которой максимальное количество элементов равно количеству вершин графа.

**Спецификация программы.**

Программа написана на языке C++. Программа на вход получает вершину начала пути и конца пути. После вводятся ребра и их веса. Перед началом работы алгоритма у каждого узла сортируются смежные вершины по возрастанию по сумме эвристического числа и веса ребра.

**Тестирование.**

Пример вывода результата (INF - бесконечность).



|  |  |  |
| --- | --- | --- |
| № | Input | Output |
| 1 | a a  a a 1 | a |
| 2 | e a  a a 0.5  a b 2  a c 3  b c 2  c a 3  c e 4  e c 1  c f 5  e g 1  g a 2  g h 4  e r 2 | eca |
| 3 | a f  a a 0.5  a b 2  a c 3  b c 2  c a 3  c e 4  e c 1  c f 5  e g 1  g a 2  g h 4  e r 2 | abcf |

Граф для тестов 2 и 3 приведен на рис. 2.

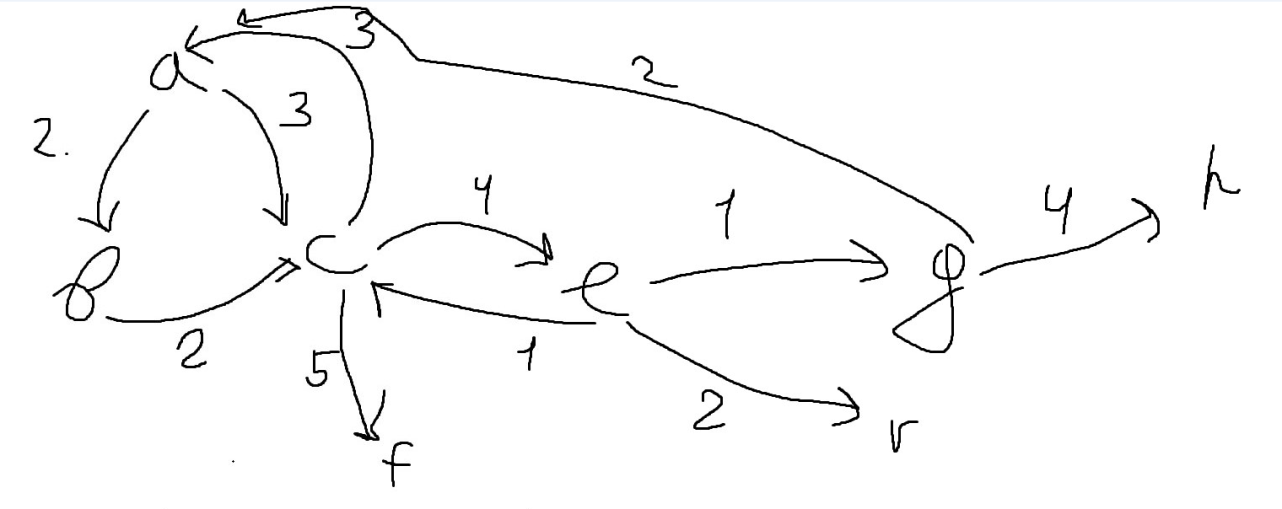


Рисунок 2 – Граф для тестов

**Вывод.**

В ходе выполнения лабораторной работы был реализован метод A\* для нахождения минимального маршрута от одного узла к другому в графе.

ПРИЛОЖЕНИЕ

КОД ПРОГРАММЫ

#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

using namespace std;

struct Edge { //структура для хранения ребра графа

char connectedNode; //смежный узел

int heruisticNum; //эвристическое число

double weight; //вес ребра

};

struct Graph { //структура для хранения графа

char name; //название узла

vector <Edge> edge; //вектор смежных узлов

};

struct AStar { //структура для метода А\*

char node; //текущий узел

char prev; //предыдущий узел

double f; //приоритет пути определяется по значению f = g + heruisticNum

double g; //значение g, необходимое для вычисления значения f

double h;

bool checked = false; //проверена ли вершина

};

int heuristicFunc(char from, char to); //эвристическая функция

bool cmpEdge(Edge i, Edge j); //компаратор для сортировки ребер по эвристическому значению

void aStar(char start, char finish, vector <Graph> gr, string nodeNames); //метод A\*

int main() {

setlocale(LC\_ALL, "RUS");

char start; //начальная вершина

char finish; //конечная вершина

double currWeight = 0;

string resStr = "";

vector <Graph> gr;

vector <char> a;

vector <char> b;

vector <double> c;

int size = 0;

char tmpa; //для записи в вектор a

char tmpb; //для записи в вектор b

double tmpc; //для записи в вектор c

cin >> start >> finish;

string nodeNames; //для хранения названий всех узлов

nodeNames += start;

while (cin >> tmpa >> tmpb >> tmpc) {

a.push\_back(tmpa);

b.push\_back(tmpb);

c.push\_back(tmpc);

size++;

Edge e;

Graph g;

e.connectedNode = tmpb;

e.weight = tmpc;

g.name = tmpa;

g.edge.push\_back(e);

gr.push\_back(g);

if (nodeNames.length() == 0) nodeNames += tmpb;

else if (nodeNames.find(tmpb) == string::npos) nodeNames += tmpb;

}

for (int i = 0; i < gr.size(); i++) {

for (int j = 0; j < gr[i].edge.size(); j++) { //сортировка смежных вершин по эвристическому числу

gr[i].edge[j].heruisticNum = heuristicFunc(gr[i].edge[j].connectedNode, finish);

}

sort(gr[i].edge.begin(), gr[i].edge.end(), cmpEdge);

}

cout << "-------------------------------------------------------------------------" << endl;

cout << "Исходные данные:" << endl;

cout << "Начальная вершина: " << start << ", конечная вершина: " << finish << endl;

for (int i = 0; i < size; i++) cout << a[i] << " ";

cout << endl;

for (int i = 0; i < size; i++) cout << b[i] << " ";

cout << endl;

for (int i = 0; i < size; i++) cout << c[i] << " ";

cout << endl;

cout << "-------------------------------------------------------------------------" << endl;

cout << "Ребра после сортировки по эвристическому числу:" << endl;

for (int i = 0; i < gr.size(); i++) {

for (int j = 0; j < gr[i].edge.size(); j++) {

cout << "Ребро ";

cout << "из " << gr[i].name << " в " << gr[i].edge[j].connectedNode << "; эвристическое число: " << gr[i].edge[j].heruisticNum << ", вес: " << gr[i].edge[j].weight << endl;

}

}

cout << "-------------------------------------------------------------------------" << endl;

aStar(start, finish, gr, nodeNames);

return 0;

}

int heuristicFunc(char from, char to) { // эвристическая функция

return abs(to - from); //возвращает близость символов, обозначающих вершины графа, в таблице ASCII

}

bool cmpEdge(Edge i, Edge j) {

if (i.heruisticNum + i.weight < j.heruisticNum + j.weight) return i.connectedNode > j.connectedNode;

return i.heruisticNum + i.weight < j.heruisticNum + j.weight;

}

void aStar(char start, char finish, vector <Graph> gr, string nodeNames) {

vector <AStar> nodes;

string resStr; //результирующая строка

AStar first; //запись первой вершины в очередь

first.node = start;

first.prev = NULL;

first.g = 0;

first.f = 0;

first.h = finish - start;

nodes.push\_back(first);

for (int k = 0; k < nodeNames.length(); k++) {

AStar tmp;

int tmpi = 0;

int tmpj = 0;

if (nodeNames[k] == start) {

continue;

}

else {

for (int i = 0; i < gr.size(); i++) {

for (int j = 0; j < gr[i].edge.size(); j++) {

if (gr[i].edge[j].connectedNode == nodeNames[k]) {

tmpi = i;

tmpj = j;

break;

}

}

}

}

tmp.f = 10000;

tmp.node = gr[tmpi].edge[tmpj].connectedNode;

tmp.g = 10000;

tmp.h = gr[tmpi].edge[tmpj].heruisticNum;

nodes.push\_back(tmp);

}

for (int i = 0; i < nodes.size(); i++) {

for (int j = 0; j < nodes.size(); j++) {

cout << "Узел: " << nodes[j].node << ", предыдущий: " << nodes[j].prev << ", f: ";

if (nodes[j].f == 10000) cout << "INF";

else cout << std::setw(3) << nodes[j].f;

cout << ", g: ";

if (nodes[j].g == 10000) cout << "INF";

else cout << std::setw(3) << nodes[j].g;

cout << ", checked: " << nodes[j].checked << endl;

}

cout << endl << endl;

int index = -1;

double minH = 1000000;

double minF = 1000000;

for (int i = 0; i < nodes.size(); i++) {

if (nodes[i].checked == false) {

if (nodes[i].f < minF) {

minF = nodes[i].f;

index = i;

minH = nodes[i].h;

}

else if (nodes[i].f == minF && nodes[i].h < minH) {

index = i;

minH = nodes[i].h;

}

}

}

if (index >= 0) {

if (nodes[index].node == finish) break;

//cout << "\tmin: " << nodes[index].node << ", index: " << index;

nodes[index].checked = true;

for (int i = 0; i < gr.size(); i++) {

if (gr[i].name == nodes[index].node) {

int tmp;

for (int j = 0; j < nodes.size(); j++) {

if (nodes[j].node == gr[i].edge[0].connectedNode) {

tmp = j;

break;

}

}

nodes[tmp].prev = nodes[index].node;

int tmpG = nodes[index].g + gr[i].edge[0].weight;

if (tmpG < nodes[tmp].g) nodes[tmp].g = tmpG;

nodes[tmp].f = nodes[tmp].g + nodes[tmp].h;

}

}

//cout << endl;

}

}

cout << "-------------------------------------------------------------------------" << endl;

cout << "Результат:" << endl;

double minH = 1000000;

double minF = 1000000;

int index = 0;

for (int i = 0; i < nodes.size(); i++) {

if (nodes[i].checked == false) {

if (nodes[i].f < minF) {

minF = nodes[i].f;

index = i;

minH = nodes[i].h;

}

else if (nodes[i].f == minF && nodes[i].h < minH) {

index = i;

minH = nodes[i].h;

}

}

}

for (int i = 0; i < nodes.size(); i++) {

resStr = nodes[index].node + resStr;

if (nodes[index].node == start)

break;

for (int j = 0; j < nodes.size(); j++) {

if (nodes[j].node == nodes[index].prev) {

index = j;

break;

}

}

}

cout << resStr << endl;

}