**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 А\***

|  |  |  |
| --- | --- | --- |
| Студентгр. 8304 |  | Чешуин Д.И. |
| Преподаватель |  | Размочаева Н.В. |

Санкт-Петербург

2020

**Цель работы.**

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

**Постановка задачи.**

Вариант 7: "Мультипоточный" А\*: на каждом шаге из очереди с приоритетами извлекается n вершин (или все вершины, если в очереди меньше n вершин). n задаётся пользователем.

**Описание алгоритма.**

A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. При выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение F(x), 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 всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью. F(x) - эвристическая функция, представляющая собой сумму расстояния до текущей вершины и расстояние от текущей вершины до целевой, оцениваемое как раз расстояние между именами вершин в таблице символов ASCII.

**Анализ 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: ошибка эвристической функции не должна расти быстрее, чем логарифм от оптимальной эвристики.

**Описание функций и СД.**

Для решения задачи был реализован класс Graph, класс AStarPathFinder, структура отдельного ребра графа – Edge и структура вершины алгоритма A\* - Node. UML диаграмму использованных структур данных смотри на рисунке 1.

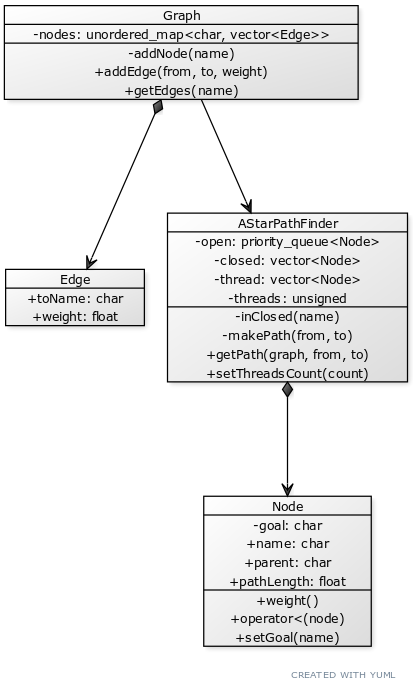


Рисунок 1 – UML диаграмма использованных структур данных

Метод поиска пути:

string getPath(const Graph& graph, char from, char to);

Принимает ссылку на граф и имена начальной и конечной вершин, возвращает строку, представляющую собой порядок вершин в пути. Если путь не найден – возвращает пустую строку

void setThreadsCount(unsigned count);

Метод устанавливает количество потоков в алгоритме.

bool inClosed(const Node& node);

Метод проверяет, была ли данная вершина уже проверена, и, если была, возвращает true, иначе – false.

string makePath(const Node& from, const Node& to);

Метод восстанавливает путь от первой from к вершине to, возвращает путь.

**Спецификация программы.**

Программа предназначена для нахождения минимального пути в графе. Программа написана на языке C++. Входными данными является число N (количество потоков), начальная и конечная вершины, перечень рёбер графа, а выходными – минимальный путь от начальной вершины до конечной

**Тестирование.**

Пример работы программы вывода для графа “a b 3.0, b c 1.0, c d 1.0, a d 5.0, d e 1.0, начальной вершины a, конечной - e и с использование двух потоков смотри на рисунке 2.

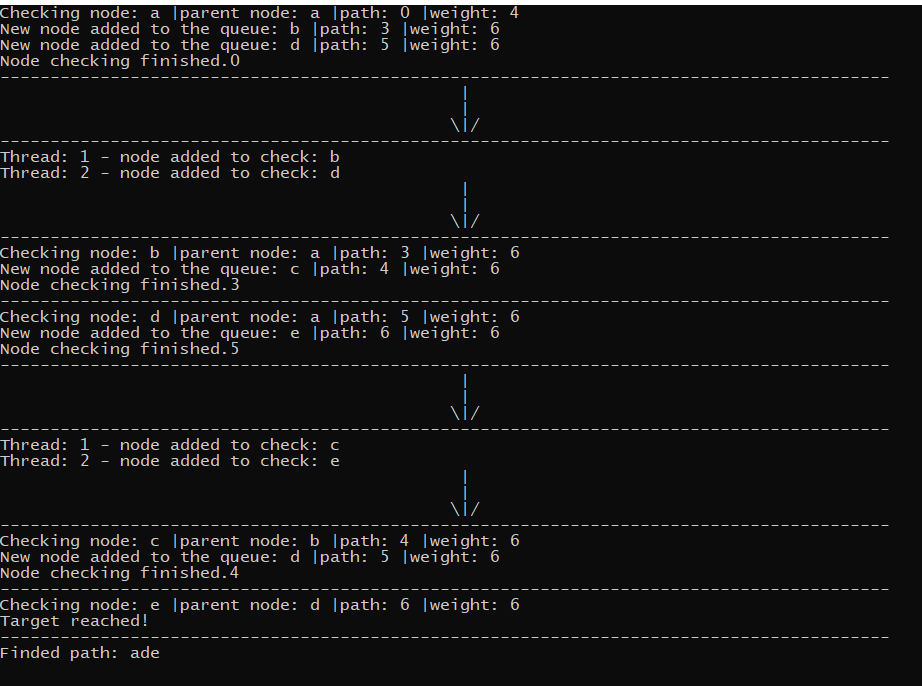


Рисунок 2 – Пример вывода для простого графа

**Выводы.**

В ходе выполнения лабораторной работы был реализован алгоритм А\*, дана оценка времени работы алгоритма, а также были получены навыки решения задач с помощью алгоритма А\*.

**ПРИЛОЖЕНИЕ A.  
ИСХОДНЫЙ КОД ПРОГРАММЫ**

main.cpp.

#include <iostream>

#include <unordered\_map>

#include <queue>

#include <cstring>

#include <fstream>

#include <cmath>

#include <vector>

#include <string>

using namespace std;

class **IOManager**

{

private:

static istream\* input;

static ostream\* output;

public:

static void **setStreamsFromArgs**(int argc, char\*\* argv)

{

if(argc > 1)

{

for(int i = 1; i < argc; i++)

{

if(strcmp(argv[i], "-infile") == 0)

{

if(i + 1 < argc)

{

input = new ifstream(argv[i + 1]);

i += 1;

}

}

if(strcmp(argv[i], "-outfile") == 0)

{

if(i + 1 < argc)

{

output = new ofstream(argv[i + 1]);

i += 1;

}

}

}

}

}

static istream& **getIS**()

{

return \*input;

}

static ostream& **getOS**()

{

return \*output;

}

static void **resetStreams**()

{

if(input != & cin)

{

delete input;

input = &cin;

}

if(output != & cout)

{

delete output;

output = &cout;

}

}

};

class **Graph**

{

public:

struct **Edge**

{

char toName;

float weight;

};

private:

unordered\_map<char, vector<Edge>> nodes;

bool **addNode**(char name);

public:

bool **addEdge**(char node1, char node2, float weight);

const vector<Edge>& **getEdges**(char name) const;

};

class **AStarPathFinder**

{

private:

struct **Node**

{

private:

static char goal;

public:

char name;

char parent;

float pathLength;

float **weight**() const

{

return ((abs(goal - name) + pathLength));

}

bool operator<(const Node& rhs) const

{

return weight() > rhs.weight();

}

static void **setGoal**(char name)

{

goal = name;

}

};

private:

priority\_queue<Node> open;

vector<Node> closed;

vector<Node> thread;

unsigned threads = 1;

private:

bool **inClosed**(const Node& node);

string **makePath**(const Node& from, const Node& to);

public:

string **getPath**(const Graph& graph, char from, char to);

void **setThreadsCount**(unsigned count);

};

int **main**()

{

Graph graph;

AStarPathFinder finder;

unsigned threads = 1;

IOManager::getOS() << "Enter threads count: " << endl;

IOManager::getIS() >> threads;

finder.setThreadsCount(2);

string from, to;

IOManager::getOS() << "Enter start and target nodes: " << endl;

IOManager::getIS() >> from >> to;

IOManager::getOS() << "Enter all edges: " << endl;

while(true)

{

string outNode, inpNode;

float weight;

if(!(IOManager::getIS() >> outNode >> inpNode >> weight))

{

break;

}

graph.addEdge(outNode[0], inpNode[0], weight);

}

IOManager::getOS() << "Trying to find path. Target: " << to << endl;

string path = finder.getPath(graph, from[0], to[0]);

IOManager::getOS() << "Finded path: " << path << endl;

return 0;

}

bool Graph::**addNode**(char name)

{

vector<Edge> emptyEdges;

return nodes.emplace(*name*, *emptyEdges*).second;

}

bool Graph::**addEdge**(char fromName, char toName, float weight)

{

auto fromNode = nodes.find(fromName);

auto toNode = nodes.find(toName);

if(fromNode == nodes.end())

{

if(!addNode(fromName))

{

return false;

}

}

if(toNode == nodes.end())

{

if(!addNode(toName))

{

return false;

}

}

auto node = nodes.find(fromName);

node->second.push\_back({toName, weight});

return true;

}

const vector<Graph::Edge>& Graph::**getEdges**(char name) const

{

return nodes.find(name)->second;

}

string AStarPathFinder::**getPath**(const Graph& graph, char from, char to)

{

Node::setGoal(to);

Node start;

start.name = from;

start.parent = from;

start.pathLength = 0;

open.push(start);

IOManager::getOS() << "Threads: " << threads << endl;

IOManager::getOS() << "-----------------------------------------------------------------------------------------" << endl;

while(!open.empty())

{

for(unsigned i = 0; i < threads && !open.empty(); i++)

{

thread.push\_back(open.top());

IOManager::getOS() << "Thread: " << i + 1 << " - node added to check: " << thread.back().name << endl;

open.pop();

}

IOManager::getOS() << " | " << endl;

IOManager::getOS() << " | " << endl;

IOManager::getOS() << " \\|/ " << endl;

IOManager::getOS() << "-----------------------------------------------------------------------------------------" << endl;

for(auto node : thread)

{

IOManager::getOS() << "Checking node: " << node.name << " |parent node: " << node.parent

<< " |path: " << node.pathLength << " |weight: " << node.weight() << endl;

if(inClosed(node))

{

IOManager::getOS() << "Node was checked previosly!" << endl;

continue;

}

if(node.name == to)

{

IOManager::getOS() << "Target reached!" << endl;

IOManager::getOS() << "-----------------------------------------------------------------------------------------" << endl;

string path = makePath(start, node);

closed.clear();

while(!open.empty())

{

open.pop();

}

while(!thread.empty())

{

thread.pop\_back();

}

return path;

}

for(auto edge : graph.getEdges(node.name))

{

Node newNode;

newNode.name = edge.toName;

newNode.parent = node.name;

newNode.pathLength = node.pathLength + edge.weight;

IOManager::getOS() << "New node added to the queue: " << newNode.name

<< " |path: " << newNode.pathLength << " |weight: " << newNode.weight() << endl;

open.push(newNode);

}

closed.push\_back(node);

IOManager::getOS() << "Node checking finished." << node.pathLength << endl;

IOManager::getOS() << "-----------------------------------------------------------------------------------------" << endl;

}

IOManager::getOS() << " | " << endl;

IOManager::getOS() << " | " << endl;

IOManager::getOS() << " \\|/ " << endl;

IOManager::getOS() << "-----------------------------------------------------------------------------------------" << endl;

thread.clear();

}

return "";

}

bool AStarPathFinder::**inClosed**(const Node& node)

{

for(auto checked : closed)

{

if(node.name == checked.name)

{

if(node.pathLength < checked.pathLength)

{

IOManager::getOS() << "New path shorter than previos, updated." << endl;

IOManager::getOS() << "Old length - " << checked.pathLength << " new length - " << node.pathLength << endl;

checked.parent = node.parent;

}

return true;

}

}

return false;

}

string AStarPathFinder::**makePath**(const Node& from, const Node& to)

{

string strPath;

strPath += from.name;

vector<char> path;

Node curNode = to;

while(curNode.name != from.name)

{

path.push\_back(curNode.name);

for(auto node : closed)

{

if(node.name == curNode.parent)

{

curNode = node;

break;

}

}

}

while(!path.empty())

{

strPath += path.back();

path.pop\_back();

}

return strPath;

}

void AStarPathFinder::**setThreadsCount**(unsigned count)

{

threads = count;

}

char AStarPathFinder::Node::goal;

istream\* IOManager::input = &cin;

ostream\* IOManager::output = &cout;