**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 А\***

Студент гр. 7383 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Рудоман В.А.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Жангиров Т.Р.

Санкт-Петербург

2019

**Цель работы.**

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 А\*:

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

**Описание алгоритма.**

Вариант индивидуализации: 3с. Списки смежности. Написать функцию, проверяющую эвристику на допустимость и монотонность.

Жадный алгоритм в ходе своей работы выбирает оптимальное на данном шаге решение, которое, в общем случае, совсем не обязано быть глобально лучшим. В текущей реализации на каждом шаге в списке смежности находится ближайшая вершина (относительно последней, добавленной в дек) и помещается в конец дека. Если текущая цепочка приводит в тупик, происходит откат (=последовательное извлечение некоторого числа элементов из конца дека).

Алгоритм А\* является модификацией алгоритма Дейкстры. Его эффективность напрямую зависит от задаваемой на графе эвристической функции. Алгоритм последовательно «раскрывает» вершины графа, хранящиеся в очереди с приоритетом. Приоритетом является оценочное расстояние до конечной вершины (расстояние от начальной вершины + эвристическая оценка расстояния до конечной вершины). «Раскрытие» подразумевает добавление всех смежных с текущей вершин в очередь. Алгоритм заканчивает свою работу в тот момент, как только происходит попытка «раскрытия» конечной вершины.

**Описание функций и структур данных.**

В лабораторной работе реализовано несколько небольших вспомогательных классов для работы со списками смежности.

**class AdjPair**

Представляет собой пару: вершина и расстояние до нее.

**class AdjPair\_prior**

Вспомогательный класс для определения элементов очереди с приоритетом.

**class VertexInAdjList**

Представляет собой пару: вершина и список смежности, причем элементами списка являются классы AdjPair.

**class AdjacencyList**

Представляет собой граф на основе контейнера std::vector, элементами которого являются списки смежности.

Методы класса:

**void addEdge(char start, char end, double length)**

Аргументы: вершины **start, end** и длина **length** соответствующего ребра между ними.

Возвращаемое значение: отсутствует.

Описание: добавляет ребро в граф.

**void addVertex(char name)**

Аргументы: вершина **name.**

Возвращаемое значение: отсутствует.

Описание: добавляет вершину в граф.

**bool isConsistent(char end)**

Аргументы: конечная вершина **end.**

Возвращаемое значение: истина, если эвристическая функция монотонна, ложь – иначе.

Описание: проверяет эвристическую функцию на монотонность. Существует теорема о том, что любая монотонная функция допустима, поэтому необходимость отдельно проверять эвристику на допустимость отсутствует. Формально необходимо доказать, что для любых двух смежных вершин a и b верно следующее соотношение h(a) — h(b) <= d(a, b), где h – эвристическая функция, а d-- длина ребра. Задача решается полным перебором по всем вершинам.

**void aStar(char start, char end)**

Аргументы: начальная вершина **start** и конечная вершина **end.**

Возвращаемое значение: отсутствует.

Описание: реализует алгоритм А\* с помощью стандартной очереди с приоритетом (для хранения вершин, которые предстоит «раскрыть»), и двух map-контейнеров (один для хранения цепочек вершин, другой для хранения минимального расстояния до каждой из вершин). Вершины графа извлекаются из очереди с приоритетом и последовательно «раскрываются». Пока очередь не пуста или же мы не пытаемся раскрыть конечную вершину.

**std::deque <char> Greedy(char start, char end)**

Аргументы: начальная вершина **start** и конечная вершина **end.**

Возвращаемое значение: дек, содержащий путь.

Описание: реализует жадный алгоритм, последовательно вызывая поиск минимального ребра функцией **findMinEdge**.

**std::pair<bool,char>findMinEdge(std::vector<VertexInAdjList> &tmpAdjList, char cur)**

Аргументы:  **tmpAdjList –** список смежности**, cur –** вершина, относительно которой ищется минимальное ребро.

Возвращаемое значение: пара: (найдено ли минимальное ребро, вершина).

Описание: реализует жадный алгоритм, последовательно вызывая поиск минимального ребра функцией **findMinEdge**.

**Оценка сложности алгоритма.**

Сложность алгоритма А\* очень сильно зависит от заданной эвристики. В данной задаче она «бесплатна», выполняется за короткое время. В худшем случае алгоритм А\* превращается в алгоритм Дейкстры, который, как известно, имеет сложность O(|E| + |V|log |V|).

А\* в некотором смысле «направленный» BFS, который экспоненциален по использованию памяти в отношении длины пути (в данном случае являющимся решением).

**Тестирование.**

Таблица 1 – пример работы алгоритма А\*

|  |  |
| --- | --- |
| Входные данные | Ответ |
| a e  a b 8  a c 9  c d 11  a d 9  d e 15 | ade |
| b f  b c 3  b d 5  c d 7  d f 1 | bdf |
| a z  a b 3  b z 4  a x 3  w z 4 | abz |

**Выводы.**

В ходе выполнения данной лабораторной работы были изучены и реализованы жадный алгоритм и алгоритм А\*, а так же исследована сложность по памяти и операциям каждого из них. Помимо этого представление графа в памяти было реализовано с помощью списков смежности. Для заданной эвристики были написаны функции, проверяющие ее на допустимость и монотонность. Детально изучена необходимость каждого из этих свойств для эвристический функций в целом.

**ПРИЛОЖЕНИЕ A**

**РЕАЛИЗАЦИЯ АЛГОРИТМА А\***

#include <iostream>

#include <list>

#include <vector>

#include <algorithm>

#include <cstring>

#include <map>

#include <queue>

using namespace std;

class AdjPair {

public:

AdjPair(char name, double weight): name(name), weight(weight) {};

char name = 0;

double weight = 0;

};

class VertexInAdjList {

public:

VertexInAdjList(char name) : name(name){};

std::list <AdjPair> AdjacentVertices;

char get\_name() const {

return name;

}

private:

char name;

};

class AdjacencyList {

public:

std::vector <VertexInAdjList> AdjList;

void aStar(char start, char end);

double heuristic(char curVertex, char end) {

return abs(end - curVertex);

}

void addVertex(char name){ // добавляет вершину в граф

for (auto &cur\_vertex : AdjList)

if (cur\_vertex.get\_name() == name)

return;

AdjList.emplace\_back(name);

}

void addEdge(char start, char end, double length){ //добавление ребра в граф

addVertex(start);

addVertex(end);

for (auto &vectorElem : AdjList){

if (vectorElem.get\_name() == start){

vectorElem.AdjacentVertices.emplace\_back(end, length);

}

}

}

// returns a copy of adjacency list of given vertex

std::list <AdjPair> operator[](char name){

for (auto vertex : AdjList)

if (vertex.get\_name() == name)

return vertex.AdjacentVertices;

}

//need to check that for each rib (a,b) h(a) - h(b) <= d(a,b)

bool isConsistent(char end){

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

char a = AdjList[i].get\_name();

double ha = heuristic(a, end);

std::list <AdjPair> AdjV = (\*this)[a];

for(auto cur : AdjV){

if(ha - heuristic(cur.name, end) > cur.weight)

return false;

}

}

return true;

}

};

class AdjPair\_prior { //Вспомогательный класс для определения элементов очереди с приоритетом.

public:

AdjPair\_prior(char vert, double prior): vert(vert), prior(prior){};

char vert;

double prior;

bool operator<(const AdjPair\_prior b) const {

return prior > b.prior;

}

};

void AdjacencyList::aStar(char start, char end) { // реализация алгоритма A\*

std::priority\_queue<AdjPair\_prior> toExpandPriorQueue;

toExpandPriorQueue.emplace(start, 0);

std::map <char, char> prevChain;

std::map <char, double> curCost;

prevChain[start] = start;

curCost[start] = 0;

while(!toExpandPriorQueue.empty()) { // cout << ""<<endl;

const char curVertex = toExpandPriorQueue.top().vert;

cout << "Достали вершину " << curVertex <<endl;

toExpandPriorQueue.pop();

if(curVertex == end) {

break;

}

for(const AdjPair& child : (\*this)[curVertex]) { // placing in queue each child of cur vertex

cout << "Новый потомок " << child.name <<endl;

double new\_cost = curCost[curVertex] + child.weight;

if(curCost.find(child.name) == curCost.end() || new\_cost < curCost[child.name]) { //not found or smaller than existing one

curCost[child.name] = new\_cost;

double priority = new\_cost + heuristic(child.name, end);

cout << "Ее приоритет "<<priority <<endl;

prevChain[child.name] = curVertex;

cout << "В prevChain записываем " << child.name << "и" << curVertex <<endl;

toExpandPriorQueue.emplace(child.name, priority);

cout << "Записываем в очередь" << child.name <<endl;

}

}

}

// reconstructing the path

std::vector<char> path;

char elem = end;

path.push\_back(elem);

while(elem!= start) {

elem = prevChain[elem];

path.push\_back(elem);

}

reverse(path.begin(), path.end());

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

std::cout << path[i];

std::cout << std::endl;

// std::cout << isConsistent(end) << std::endl;

return;

}

int main(){

AdjacencyList graph;

double length;

char start, end;

std::cin >> start >> end;

char u, v;

while (std::cin >> u >> v >> length){

graph.addEdge(u, v, length);

}

graph.aStar(start, end);

return 0;

}