

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 поиска пути в графах
Вариант 9

Студент гр. 8303

Колосова М.П.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы

Изучение алгоритмов поиска пути в графе и вывод его графического представления.

Задание 1.

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
```

Задание 2.

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

Индивидуализация

Вывод графического представления графа.

Описание жадного алгоритма

В программе реализован жадный алгоритм поиска пути. Он заключается в выборе ребра с минимальной длиной на каждом шаге.

На каждом шаге происходит переход к следующей вершине с помощью выбора ребра с минимальной длиной и изменения текущей вершины на конечную вершину выбранного ребра. Выбор происходит до тех пор, пока это возможно. Если вершина, на которой мы находимся, не является финальной и у нее нет выходящих ребер, то для предыдущей вершины выбирается другое ребро с минимально возможной длиной.

Для каждой текущей вершины обход графа начинается сначала, поэтому худший случай — переход будет выполняться к вершине, выходящие ребра которой хранятся в памяти максимально далеко от выбранного ребра. Тогда теоретическая сложность алгоритма составит $O(V \cdot N)$, где N — количество ребер, а V — количество вершин.

На протяжении всей программы используется вектор ребер. Максимальное количество ребер для V вершин равняется $V(V-1)$, так как граф ориентированный. Тогда сложность по памяти составляет $O(V(V-1))$.

Описание алгоритма A*

В программе так же используется алгоритм A*. Он заключается в выборе на каждом шаге решения с наименьшим приоритетом, где приоритет – это сумма длины текущего решения и некоторой эвристической функции, дающей оценку пути, который необходимо пройти до целевой вершины. По условию в качестве эвристической оценки используется близость символов, обозначающих вершины графа.

В качестве списка вершин для обработки с приоритетом используется вектор пар. На каждом шаге выбирается вершина с минимальным приоритетом. Цикл прекращается, когда список для проверки пуст или выбрана финальная вершина. Далее рассматриваются все еще не проверенные соседи вершины. Если путь из текущей вершины в соседнюю короче уже существующего или сосед еще не в очереди для обработки, то выполняется перерасчет приоритета соседа и текущая вершина запоминается как вершина, из которой мы пришли.

Данный алгоритм является модификацией алгоритма Дейкстры с использованием вектора пар как очереди с приоритетом. Сложность в худшем случае составит $O(V^2)$, так как данный алгоритм имеет квадратичную зависимость от количества вершин графа. Однако, если эвристическая оценка не дает правильных направлений поиска, то при поиске будут рассмотрены все пути, что означает экспоненциальную сложность по времени - $O(2^E)$, где E – количество ребер в графе.

Для любого графа и любой эвристической функции максимальная длина вектора ребер — $V(V-1)$, максимальная длина вектора пути, включающая все ребра - $0.5 * V * (V-1) + 1$, максимальная длина проверенных вершин и вершин для проверки — V , максимальная длина карты пройденных вершин и вектора минимальных длин до вершин — тоже V . Отсюда сложность по памяти $O(V^2)$.

Описание структур данных

static vector <char> path;- вектор пути, используется для хранения вершин кратчайшего пути

class Edge - используется для хранения информации о ребре, включает в себя имена начальной и конечной вершин, длину ребра и отметку о проверке, добавлена перегрузка оператора равенства

static vector <Edge> graph; - вектор ребер, используется для описания графа

map <char, bool> inChecked; - контейнер пар ключ-значение для отметок о проверке вершин, используется в алгоритме A*

vector <pair<char, double>> openset; - вектор вершин с приоритетами, используется в алгоритме A*

map <char, bool> inOpenSet; - контейнер пар ключ-значение для отметок о вхождении в список вершин с приоритетами, используется в алгоритме A*

map <char, char> vertexMap; - контейнер пар ключ-значение для записи вершины, из которой пришли, используется для построения минимального пути в алгоритме A*

map <char, double> minPathToVertex; - контейнер пар ключ-значение для записи минимальных длин путей от стартовой вершины до ключевой, используется в алгоритме A*

Описание функций

bool cmp(const Edge &a, const Edge &b); — компаратор для сравнения ребер по первой вершине и длине (возвращает истину в случае, если начальная вершина первого ребра меньше начальной вершины второго ребра или если начальные вершины равны и длина первого меньше), используется в сортировке графа после ввода данных (выходящие ребра одной вершины по возрастанию длины хранятся рядом друг с другом)

bool cmpPriority(pair <char, double> a, pair <char, double> b); - компаратор для сравнения приоритетов вершин (возвращает истину, если приоритет первой вершины меньше), используется в поиске вершины с минимальным приоритетом в алгоритме A*

bool cmpForOpenset(pair<char, double> a, pair<char, double> b); - компаратор для сравнения вершин по имени (возвращает истину, если номер первой вершины в таблице символов меньше), используется в поиске вершины в списке в алгоритме A*

void findPath(); - функция, в которой реализован жадный алгоритм, работает с глобально объявленным графом и записывает результат в вектор пути

void findPathWithAStar(); - функция, в которой реализован алгоритм A* поиска кратчайшего пути в графе

`void reconstructPath(map <char, char> vertexMap);` - функция построения пути по контейнеру, содержащему информацию о вершинах-родителях, восстанавливает с конца кратчайший путь и записывает развернутый вариант в вектор пути

`double heuristicFunction(char curVertex);` - эвристическая функция, оценивает близость передаваемого символа к финальной вершине

Вывод графического представления графа

Для визуализации графа в процессе программы создается текстовый файл с расширением «.dot». DOT — это язык описания графов. С помощью функций записи строк он заполняется информацией о графе, который хранится в памяти. Строка формируется особым образом исходя из синтаксиса dot и желаемых результатов — ориентированный граф с подписанными длинами ребер. В графическом виде описанный таким образом граф представляется с помощью команды терминала с флагами формата результата и имени создаваемого файла. Можно либо установить dot соответствующей командой, либо использовать онлайн-конвертор. Например, <http://www.webgraphviz.com>, <https://dreampuf.github.io/GraphvizOnline>, <https://onlineconvertfree.com/ru/convert-format/dot-to-png/> и другие.

Для вывода изображения была написана программа на языке C++, выполняющая через функцию `system()` необходимые терминальные команды. Так же создан скрипт командной строки для компиляции и запуска проекта с уже установленным dot, в результате которого открывается PNG-изображение заданного графа.

Тестирование жадного алгоритма

В таком виде граф хранится

```
a c 1
a b 3
a g 8
b d 2
b e 3
d e 4
e a 1
e f 2
f g 1
```

Начинаем поиск кратчайшего

Текущий вид пути:

a

Следующая вершина - c

Данная вершина не имеет вых

Текущий вид пути:

a

Следующая вершина - b

Текущий вид пути:

ab

Следующая вершина - d

Текущий вид пути:

abd

Следующая вершина - e

Текущий вид пути:

abde

Следующая вершина - a

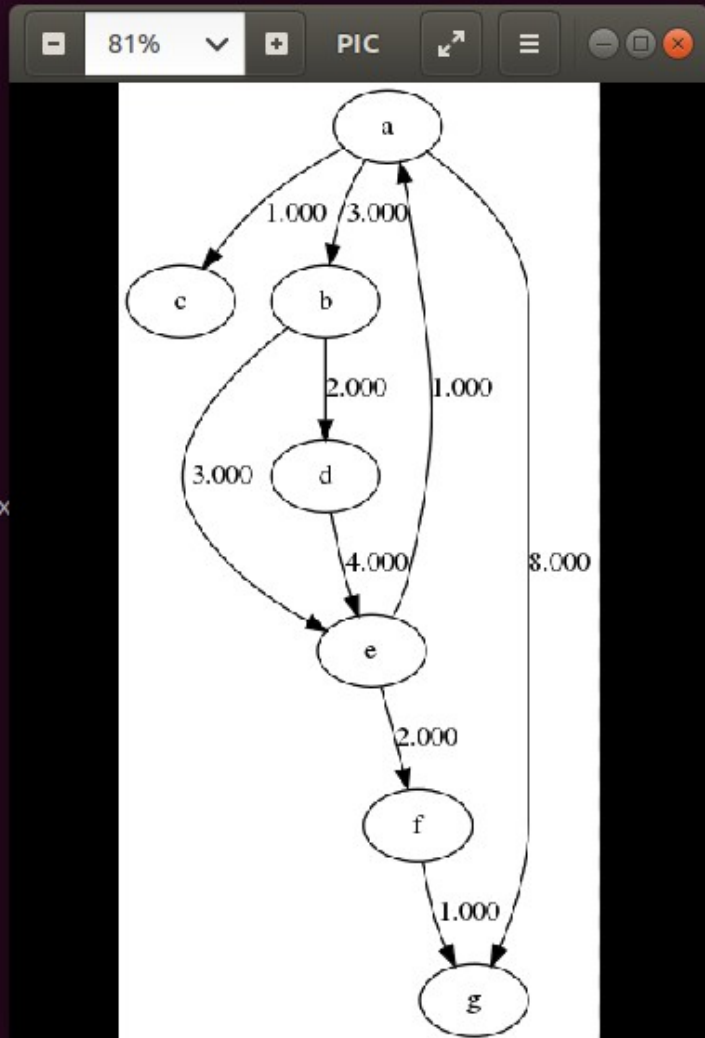
Текущий вид пути:

abdea

Следующая вершина - g

Путь найден.

abdeag



Тестирование алгоритма A*

```
a b 3.0  
b c 1.0  
c d 1.0  
a d 5.0  
d e 1.0
```

В таком виде граф хранится

```
a b 3  
a d 5  
b c 1  
c d 1  
d e 1
```

Текущая вершина - a

Начинаем обработку вершин с

Вершина с минимальным приор

Добавляем эту вершину в спи

Проверяем вершину b

Расчитываем приоритет.

Проверяем вершину d

Расчитываем приоритет.

Вершина с минимальным приор

Добавляем эту вершину в спи

Проверяем вершину c

Расчитываем приоритет.

Вершина с минимальным приор

Добавляем эту вершину в спи

Проверяем вершину e

Расчитываем приоритет.

Вершина с минимальным приор

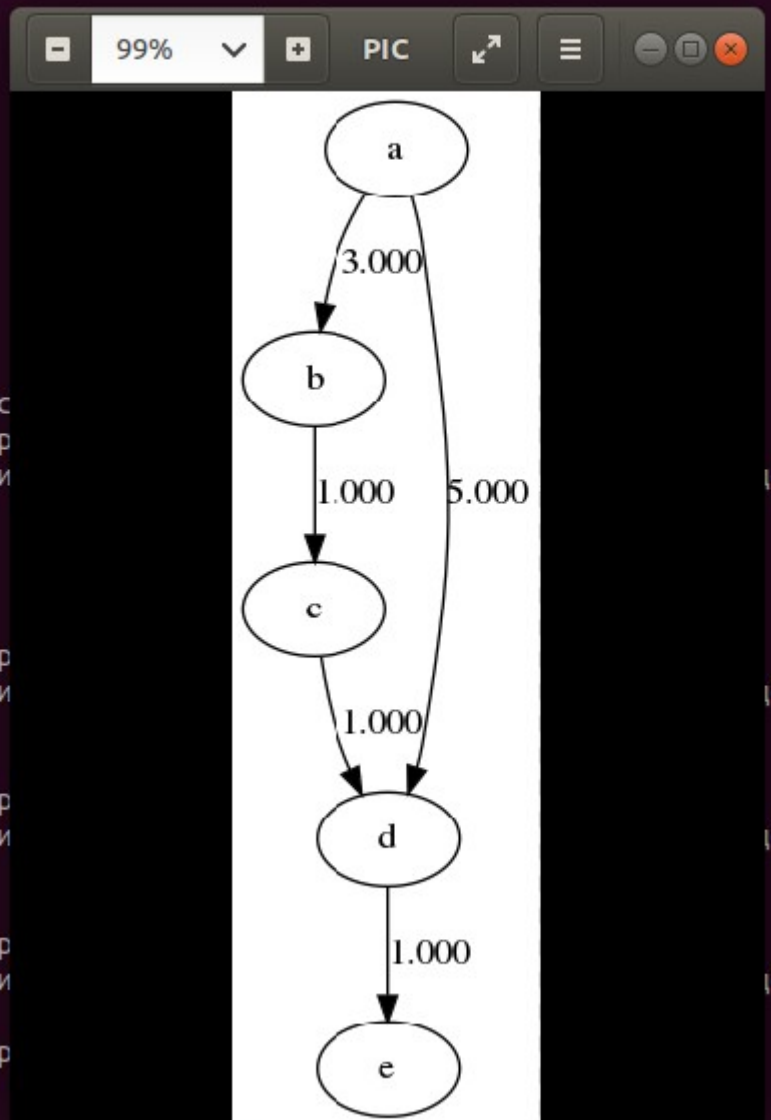
Добавляем эту вершину в спи

Вершина d уже проверена.

Вершина с минимальным приор

Путь найден.

ade



Вывод

В ходе лабораторной работы была написана программа, реализующая жадный алгоритм и алгоритм A^* , а так же написаны программы и скрипты для визуализации графа. Таким образом, были изучены алгоритмы поиска кратчайшего пути в графе и способы визуализации графов.

Приложения А. Исходный код

```
#include <iostream>
#include <map>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>
#include <queue>
#include <fstream>
#include <string.h>
#include <cstdlib>

using namespace std;

static char startVertex;
static char finalVertex;
static vector <char> path; //вектор пути
static vector <char>:: iterator vecIt; //итератор вектора пути
static char maxVertex;

class Edge{
public:
    pair<char, char> name;
    double lenght;
    bool checked; //для жадного алгоритма

    bool isCorrectEdge(char e1, char e2) const{
        return this->name.first == e1 && this->name.second == e2;
    }
    Edge(){}
    Edge(char v1, char v2, double lenght){
        this->name.first = v1;
        this->name.second = v2;
        this->lenght = lenght;
        this->checked = false;
    }
    friend bool operator==(const Edge &e1, const Edge &e2) {
        return e1.name.first == e2.name.first && e1.name.second ==
e2.name.second;
    }
};

static vector <Edge> graph;

bool cmp(const Edge &a, const Edge &b);
bool cmpPriority(pair <char, double> a, pair <char, double> b);
bool cmpForOpenset(pair<char, double> a, pair<char, double> b);
void findPath(); //обычный жадный алгоритм
void findPathWithAStar(); //A* алгоритм
void reconstructPath(map <char, char> vertexMap); //построение пути
double heuristicFunction(char curVertex); //эвристическая функция

int main()
{
    string input;
    maxVertex = 'a'; //переменная для хранения максимальной вершины
    ofstream out;
    out.open("/home/marina/Документы/piaa_2/graphFile.dot"); //очищает файл, если
в нем остались данные с предыдущего вызова
```

```

out.open("/home/marina/Документы/piaa_2/graphFile.dot", ofstream::app);
out.clear();
out.write("digraph MyGraph {\n", 18);
//считываем вершины для поиска пути
getline(cin, input);
startVertex = input[0];
finalVertex = input[2];

while(getline(cin, input) && input != ""){
    if(input[0] > maxVertex)
        maxVertex = input[0];
    if(input[2] > maxVertex)
        maxVertex = input[2];
    graph.push_back(Edge(input[0], input[2], stod(input.substr(4))));
}
sort(graph.begin(), graph.end(), cmp);
cout<<endl;

cout<<"В таком виде граф хранится в памяти:"<<endl;
vector<Edge>::iterator graphIt;
for(graphIt = graph.begin(); graphIt != graph.end(); graphIt++){
    cout<<graphIt->name.first<<" "<<graphIt->name.second<<" "<<graphIt->length<<endl;

    char *newStr = new char[50];
    strcat(newStr, " ");
    newStr[4] = graphIt->name.first;
    newStr[5] = '\\0';
    strcat(newStr, " -> ");
    newStr[9] = graphIt->name.second;
    newStr[10] = '\\0';
    strcat(newStr, " [label=");
    sprintf(newStr+18, "%f", graphIt->length);
    newStr[23] = '\\0';
    strcat(newStr, "];\n");

    out.write(newStr, strlen(newStr));
}
out.write("}\n", 2);

//findPathWithAStar();
findPath();

//вывод пути
for(vecIt = path.begin(); vecIt != path.end(); vecIt++){
    cout<<*vecIt;
}
cout<<endl;
return 0;
}

void findPath(){
    char currentVertex = startVertex; //текущая вершина
    vector<Edge>::iterator it; //итератор графа
    bool flag = false; //есть смежные вершины ли нет
    cout<<"Начинаем поиск кратчайшего пути со стартовой вершины."<<endl;

    for(it = graph.begin(); it != graph.end(); it++){

```

```

        if(currentVertex == finalVertex){//текущая вершина - финальная
            path.push_back(currentVertex);
            cout<<"Путь найден."<<endl;
            return;
        }
        if(it->name.first == currentVertex && !it->checked){//находим смежное и
еще непроверенное ребро с минимальной длиной
            flag = true;
            path.push_back(currentVertex);//добавляем текущую вершину в путь
            cout<<"Текущий вид пути:"<<endl;
            for(vecIt = path.begin(); vecIt != path.end(); vecIt++){
                cout<<*vecIt;
            }
            cout<<endl;
            currentVertex = it->name.second;
            cout<<"Следующая вершина - "<<currentVertex<<endl;
            it->checked = true;//ставим отметку о проверке
            flag = false;
            it = graph.begin();//начинаем обход сначала
        }

        if(!flag && it == graph.end() - 1){//нет путей и смежных ребер для
текущей вершины
            //ребро, выбор которого был ошибочный
            it = find(graph.begin(), graph.end(), Edge(path[path.size()-1],
currentVertex, 0));
            cout<<"Данная вершина не имеет выходящих ребер. Исключаем ее из
пути."<<endl;
            currentVertex = path[path.size()-1];//возвращаемся к предыдущей
вершине
            it->checked = true;//ставим отметку о проверке
            path.pop_back();//удаляем ошибочную вершину из пути

            it = graph.begin();//начинаем обход сначала
        }
    }

    return;
}

void findPathWithAStar(){
    vector <Edge>::iterator graphIt;//итератор графа

    //vector <char> checked;//обработанные вершины
    map <char, bool> inChecked;//для проверки вхождения в список проверенных

    vector <pair<char, double>> openset;//список вершин для обработки с
приоритетами
    vector <pair<char, double>>::iterator opensetIterator;//итератор для списка
обработки
    map <char, bool> inOpenSet;//для проверки вхождения в очередь для обработки

    char curVertex;//текущая вершина
    map <char, char> vertexMap;//карта пройденных вершин
    map <char, double> minPathToVertex;//минимальные пути до вершины

    bool needChangeValues = false;//флаг, говорящий о необходимости изменений
кратчайшего пути

    //map <char, double> priority;//приоритеты вершин

```

```

//инициализация
for(char cur = 'a'; cur <= maxVertex; cur++){
    inChecked[cur] = false;
    inOpenSet[cur] = false;
}
minPathToVertex[startVertex] = 0;//путь из вершины в саму себя
//добавляем в очередь на проверку стартовую вершину с приоритетом
openset.push_back(pair<char, double>(startVertex,
minPathToVertex[startVertex] + heuristicFunction(startVertex)));
curVertex = startVertex;

cout<<"Текущая вершина - "<<curVertex<<endl;
cout<<"Начинаем обработку вершин с приоритетами."<<endl;
while(!openset.empty()){//обработка вершин

    //выбираем текущую вершину с минимальным приоритетом
    opensetIterator = min_element(openset.begin(), openset.end(),
cmpPriority);
    pair<char, double> min_priority = *opensetIterator;
    curVertex = min_priority.first;
    cout<<"Вершина с минимальным приоритетом - "<<curVertex<<endl;

    if(curVertex == finalVertex){//достигли финальной вершины
        cout<<"Путь найден."<<endl;
        reconstructPath(vertexMap);
        return;
    }

    //Перемещаем вершину в список проверенных
    openset.erase(opensetIterator);
    inOpenSet[curVertex] = false;
    inChecked[curVertex] = true;
    //checked.push_back(curVertex);
    cout<<"Добавляем эту вершину в список проверенных и начинаем проверять
соседей."<<endl;

    //доходим до смежных ребер
    for(graphIt = graph.begin(); graphIt != graph.end(); graphIt++){
        if(graphIt->name.first == curVertex){//смежные ребра в графе
            break;
        }
    }

    //проверка соседей текущей вершины
    for(;graphIt->name.first == curVertex && graphIt != graph.end();graphIt+
+){//проверяем всех соседей текущей вершины
        char nextVertex = graphIt->name.second;

        //пропускаем соседей из закрытого списка
        if(inChecked[nextVertex]){//пропускаем проверенные вершины
            cout<<"Вершина "<<nextVertex<<" уже проверена."<<endl;
            continue;
        }
        if(!inOpenSet[nextVertex]){//соседа нет в списке для проверки
            inOpenSet[nextVertex] = true;
            openset.push_back(pair<char, double>(nextVertex,
9999));//добавляем соседа в список для проверки
            opensetIterator = openset.end();
            opensetIterator--;
            needChangeValues = true;

```

```

        }else{//сосед в очереди на проверку
            if(minPathToVertex[curVertex] + graphIt->lenght <
minPathToVertex[nextVertex]){//текущий путь меньше записанного
                needChangeValues = true;
                pair<char, double> v;
                v.first = nextVertex;
                v.second = minPathToVertex[nextVertex] +
heuristicFunction(nextVertex);
                opensetIterator = find(openset.begin(), openset.end(), v);
            }else{
                needChangeValues = false;
            }
        }
        cout<<"Проверяем вершину "<<nextVertex<<endl;
        if(needChangeValues){//нужно пересчитать путь и изменить приоритет
            cout<<"Рассчитываем приоритет."<<endl;
            vertexMap[nextVertex] = curVertex;//записывается вершина, из кт
мы прошли
            minPathToVertex[nextVertex] = minPathToVertex[curVertex] +
graphIt->lenght;
            opensetIterator->second = minPathToVertex[nextVertex] +
heuristicFunction(nextVertex);
        }
    }
    return;
}
}

void reconstructPath(map <char, char> vertexMap){
    //поиск начинается от финиша
    vector <char> buf_path;
    char curVertex = finalVertex;
    buf_path.push_back(curVertex);
    while (curVertex != startVertex) { //пока не дойдем до начальной вершины
        //получаем вершину, из которой пришли в текущую
        curVertex = vertexMap[curVertex];
        buf_path.push_back(curVertex);
    }
    //разворачиваем вектор
    for(vecIt = buf_path.end() - 1; vecIt >= buf_path.begin(); vecIt--){
        path.push_back(*vecIt);
    }
    return;
}

bool cmp(const Edge &a, const Edge &b){

    if(a.name.first == b.name.first){//сравнение по длине ребра
        if(a.lenght == b.lenght)
            return a.name.second < b.name.second;
        return a.lenght < b.lenght;
    }
    else return a.name.first < b.name.first;//сравнение по первой вершине
}

double heuristicFunction(char curVertex){//эвристичная функция - близость
символов
    return finalVertex - curVertex;
}

bool cmpPriority(pair <char, double> a, pair <char, double> b){//сравнение
вершин по приоритетам
    return a.second < b.second;
}

```

```
}  
  
bool cmpForOpenset(pair<char, double> a, pair<char, double> b){//сравнение  
вершин по названию  
    return a.first < b.first;  
}
```