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 реброа"a", "b" 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 реброа"a", "b", "c"...), каждое ребро, " 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 теоретический сложность алгоритма составит o(V*N), где N — количество), где N) 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*V), 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*V).

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, double> minPathToVertex; - контейнер пар ключ-значение для записи минимальных длин путей от стартовой вершины до ключевой, 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*

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

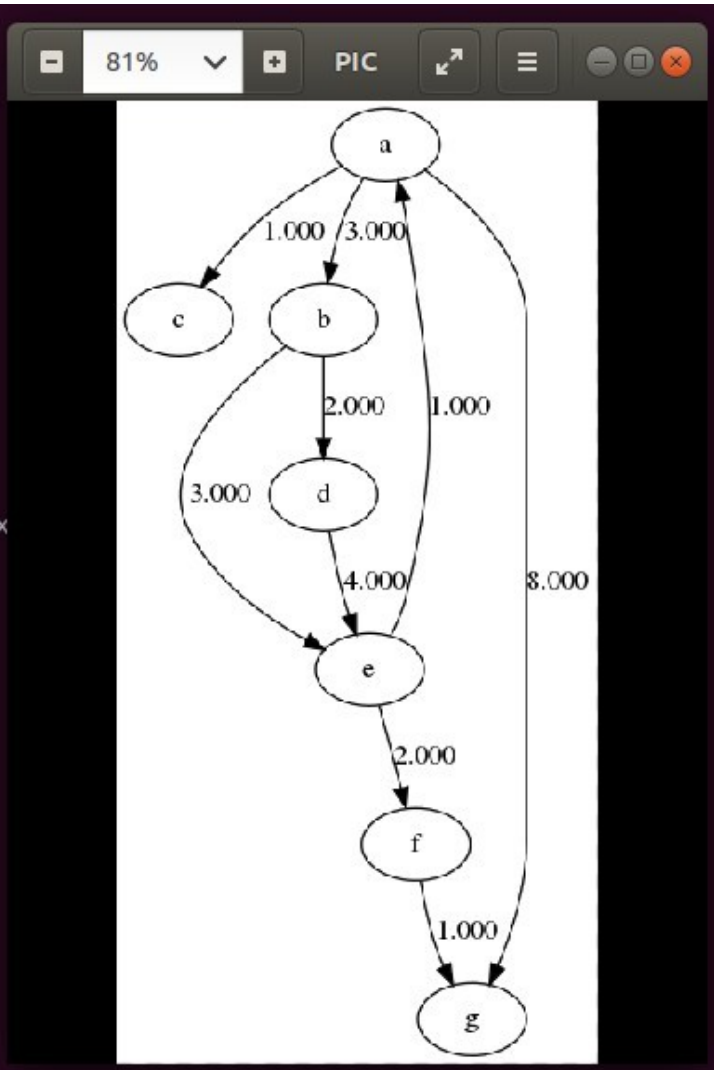
Вывод графического представления графа

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 соответсвующей 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.

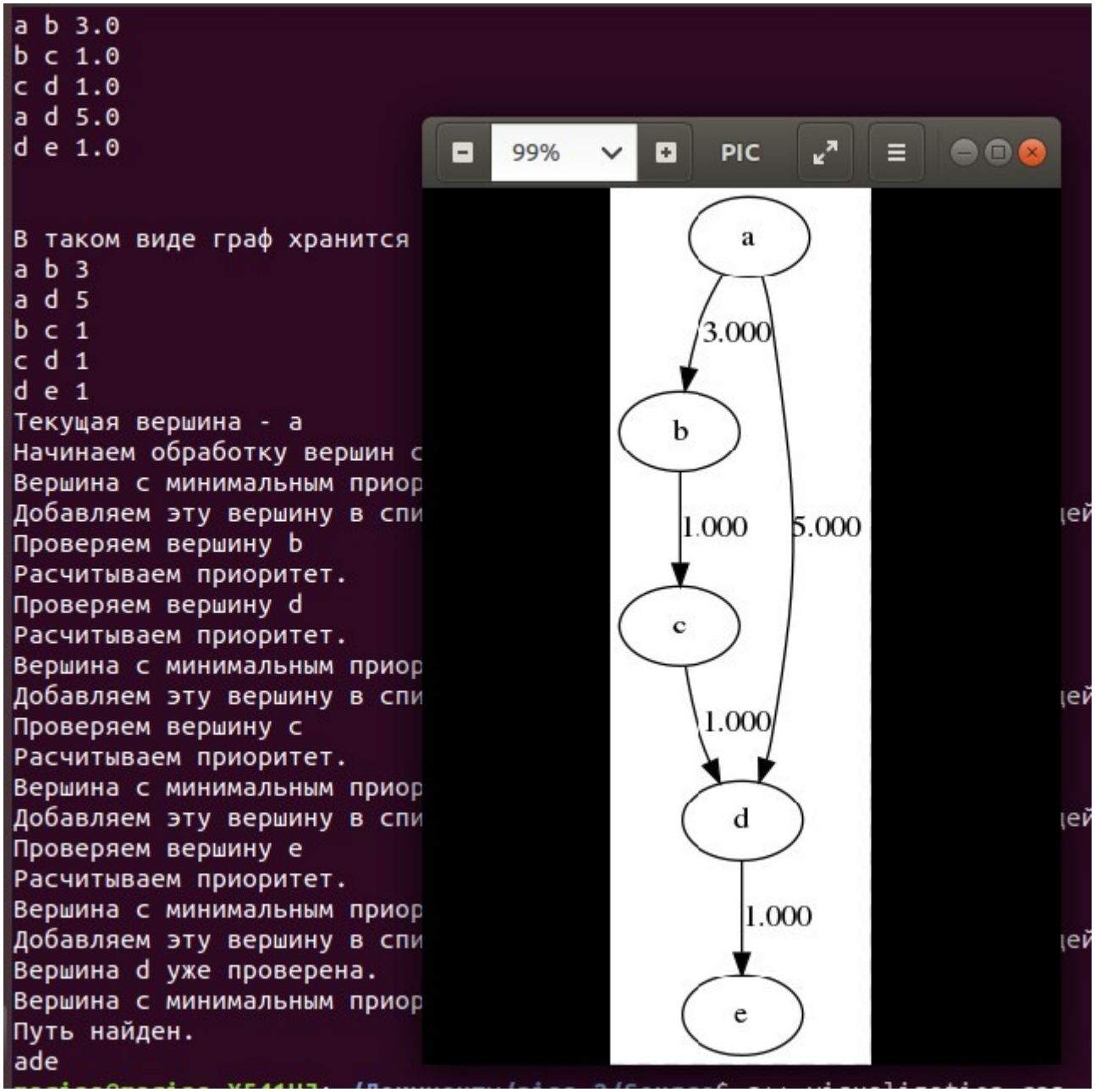
Для вывода изображения была написана программа на языке C++, выполняющая через функцию system() необходимые терминальные команды. Так же создан скрипт командной строки для компиляции и запуска проекта с уже установленным dot, в результате которого открывается PN), где N — количество заданного графа.

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*



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.

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


using namespace std;


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;
    }

};

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(char startVertex, char finalVertex, vector<Edge> &graph);//обычный жадн
void findPathWithAStar(char startVertex, char finalVertex, vector<Edge> &graph);//A* ал
void reconstructPath(map <char, char> vertexMap, char startVertex, char finalVertex);//по
double heuristicFunction(char curVertex, char finalVertex);//эвристическая функция
void printPath(vector<char> path);
```

```
int main(int argc, char* argv[])
{
   if(argc > 1)
      return  0;
   string input;
   vector <Edge> graph;
   char startVertex, finalVertex;//переменная для хранения максимальной вершины
   ofstream out;
   char *curDir = new char[200];
   ifstream dir;
   dir.open("path");
   dir>>curDir;
   strcat(curDir, "/Source/graphFile.dot");
   out.open(curDir);
   out.open(curDir, ofstream::app);

   out.clear();
   out.write("digraph MyGraph {\n", 18);
   getline(cin, input);
   startVertex = input[0];
   finalVertex = input[2];

   while(getline(cin, input) && input != ""){
      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->lenght<<en

     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->lenght);
     newStr[23] = '\0';
     strcat(newStr, "];\n");
```

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


  findPathWithAStar(startVertex, finalVertex, graph);
  //findPath(startVertex, finalVertex, graph);

  system("g++ ./Source/visualization.cpp -o showGraph");

  return 0;
}

void findPath(char startVertex, char finalVertex, vector<Edge> &graph){

   cout<<"Начинаем поиск кратчайшего пути со стартовой вершины "<<startVertex<<

   {
      char currentVertex = startVertex;//текущая вершина
      bool flag = false;//есть смежные вершины ли нет
      vector<char> path;
      auto it = graph.begin();

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

         if(currentVertex == finalVertex){//текущая вершина - финальная
            path.push_back(currentVertex);
            cout<<"Путь найден."<<endl;
            printPath(path);
            return;
         }
         if(it->name.first == currentVertex && !it->checked){//находим смежное и еще не
            flag = true;
            path.push_back(currentVertex);//добавляем текущую вершину в путь
            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 вер

            //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(char startVertex, char finalVertex, vector<Edge> &graph){


    map <char, double> openset;//список вершин для обработки с приоритетам
    map <char, double> minPathToVertex;//минимальные пути до вершины

    minPathToVertex[startVertex] = 0;//путь из вершины в саму себя
    openset[startVertex] = minPathToVertex[startVertex] + heuristicFunction(startVertex, f
    cout<<"Текущая вершина - "<<startVertex<<endl;
    cout<<"Начинаем обработку вершин с приоритетами."<<endl;

    {
        map <char, char> vertexMap;//карта пройденных вершин
        vector<char> inChecked;//для проверки вхождения в список проверенных
        while(!openset.empty()){//обработка вершин

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

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

```
cout<<"Добавляем эту вершину в список проверенных и начинаем проверять соседей
        openset.erase(curVertex);
        inChecked.push_back(curVertex);

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

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

          //пропускаем соседей из закрытого списка
          if(find(inChecked.begin(), inChecked.end(), nextVertex) != inChecked.end()){//п
            cout<<"Вершина "<<nextVertex<<" уже проверена."<<endl;
            continue;
          }

          bool needChangeValues = false;
          if(openset.find(nextVertex) == openset.end()){//соседа нет в списке для провер
            needChangeValues = true;
          }else{//сосед в очереди на проверку
            if(minPathToVertex[curVertex] + graphIt->lenght < minPathToVertex[nextVe
              needChangeValues = true;
            }else{
              needChangeValues = false;
            }
          }
          cout<<"Проверяем вершину "<<nextVertex<<endl;
          if(needChangeValues){//нужно пересчитать путь и изменить приоритет
            cout<<"Расчитываем приоритет."<<endl;
            vertexMap[nextVertex] = curVertex;//записывается вершина, из кт мы прош
            minPathToVertex[nextVertex] = minPathToVertex[curVertex] + graphIt->leng
            openset[nextVertex] = minPathToVertex[nextVertex] + heuristicFunction(nex
          }
        }
      }
    }

    return;
}
```

```
void reconstructPath(map <char, char> vertexMap, char startVertex, char finalVertex){

    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
    vector<char> path;
    for(auto it = buf_path.end() - 1; it >= buf_path.begin(); it--){
        path.push_back(*it);
    }
    printPath(path);
    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, char finalVertex){//эвристичнкая функция - бл
    return finalVertex - curVertex;
}

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

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

void printPath(vector<char> path){
    for(auto it = path.begin(); it != path.end(); it++){
        cout<<*it;
    }
    cout<<endl;
}
```