

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

Студент гр. 8383

Мололкин К.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить жадный алгоритм и алгоритм A^* для поиска пути в графе и решить задачи используя данные алгоритмы.

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.

Постановка задачи.

Метод A^* :

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

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

Описание жадного алгоритма.

В данном случае жадность понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. То есть при проходе графа от начальной вершины до конечной, для каждого узла выбирается ребро с наименьшим весом. В начале алгоритма кладем стартовую вершину на стек, затем от нее переходим в ближайшую вершину, так же кладем на стек, если данная вершина является конечной, то завершаем работу алгоритма, в противном случае, переходим к следующей ближайшей, если дошли до верши не имеющей инцидентных либо все инцидентные были посещены, то достаем

данную вершину из стека, и возвращаемся к той, что лежит на вершине стека.
Код программы находится в приложении А.

Описание структуры данных

Class EdgesList:

using Edge = tuple<string, string, double, bool> - ребро

Поля:

1. std::vector<Edge*> edges – вектор хранящий ребра графа;
2. std::stack<string> stackk – стек для вершин;

Методы:

1. EdgesList() = default – дефолтный конструктор класса;
2. void addEdge(Edge* edge) - добавление ребра в граф:

Функция принимает на вход переменную Edge* edge – указатель на ребро. Затем с помощью функции push_back добавляем в вектор edges.

3. void printStack() – печать стека в обратном порядке:

Функция использует второй стек newS перекладывая в него вершины из stackk, а затем достает элементы из нового стека и печатает их

4. void findWay(string startTop, string endTop) – функция поиска пути в графе:

Функция принимает на вход две переменных - string startTop – начальная вершина пути, string endTop – конечная вершина. Затем кладем начальную вершину на стек. В цикле while создаем ребро с максимальным весом, затем просматриваем инцидентные ребра, если найдем ребро связывающее с конечной вершиной, то заканчиваем поиск, если нет, то ищем не пройденное с минимальным весом, кладем его на стек и меняем startTop на найденную вершину, если нет инцидентных ребер либо все они

пройденны, то снимаем вершину со стека и возвращаемся к вершине, лежащей на вершине стека. В конце функция вызывает `printStack()`.

Сложность алгоритма

По времени жадный алгоритм можно оценить как $O(|V|)$, где V количество вершин, так как в худшем случае, алгоритм проходит по всем вершинам.

По памяти алгоритм можно оценить как $O(|E| + |V|)$, так как в графе хранятся все ребра, а стеке в худшем случае могут храниться все вершины.

Описание алгоритма A*.

После считывания графа для каждой вершины происходит сортировка всех смежных ребер по приоритету. Затем запускается алгоритм A*. Начальная вершина обозначается за текущую, затем рассматриваем все вершины в которые можно попасть из текущей, добавляем в список не посещённых со значением: расстояние от стартовой до данной + значение эвристической функции. Эвристическая функция рассчитывается как расстояние между символом данной вершины и конечной по таблице ASCII. Затем если список не пройденных вершин не пустой, то выбираем вершину с минимальным значением. Алгоритм заканчивает работу когда из текущей вершины есть ребро в конечную, либо когда список не посещённых вершин пустой. Код программы находится в приложении Б.

Описание структуры данных:

Class Pair – используется для хранения ребра

Поля:

1. `GraphTop* top` – вершина;
2. `double weight` – вес пути;

Методы:

1. `Pair(GraphTop* top, double weight)` – конструктор класса;

Class GraphTop – используется для хранения вершины

Поля:

1. char name – имя вершины;
2. double gFunc – значение функции g для заданной вершины;
3. double hFunc – значение функции h для заданной вершины;
4. double fFunc – значение функции f для заданной вершины;
5. std::vector<Pair*> adjacentEdges – вектор для хранения смежных ребер;

Методы:

1. void sortTops(char goal) – сортировка смежных вершин по приоритету:
Функция принимает на вход переменную char goal – название конечной вершины, затем для каждой смежной вершины рассчитывается fFunc и по этому признаку вершины сортируются пузырьком в возрастающем порядке.
2. void setHFunc(char b) – установка поля hFunc у вершины:
Функция принимает на вход переменную char b – название конечной вершины, затем возвращаем модуль от разности переданного символа и данной вершины.
3. void setFFunc() – установка поля fFunc у вершины;
4. void addAdjacentEdge(GraphTop* top, double weight) – функция добавляет ребро в вектор для хранения смежных ребер:
Функция принимает на вход переменные GraphTop* top – смежная вершина и double weight – вес ребра, затем вызываем конструктор ребра и добавляем в вектор смежных ребер;

Class EdgesList – используется для хранения графа**Поля:**

1. std::vector<GraphTop*> tops – вектор хранящий все вершины графа;

Методы:

1. `void sortTops(char goal)` – для каждой вершины сортируется список смежных по приоритету:
Функция принимает переменную `char goal` – конечная вершина, затем для каждой вершины вызывается функция `sortTops(goal)`
2. `void addEdge(char topName, char adjacentTopName, double weight)` – добавление ребра в граф:
Функция принимает на вход `char topName` – название начальной вершины ребра, `char adjacentTopName` – название конечной вершины ребра, `double weight` – вес ребра. Затем для `topName` и `adjacentTopName` вызывается функция `isExistTop()` и результаты записываются в новые переменные, затем для вершины с именем `topName` добавляем смежное ребро с помощью функции `addAdjacentEdge()`;
3. `GraphTop* isExistTop(char topName)` – проверяет есть ли вершина в графе:
Функция принимает на вход `char topName` – название вершины, затем проверяем есть ли заданная вершина в графе, если есть возвращаем указатель на нее, если нет, то создаем вершину и возвращаем указатель на нее;
4. `static GraphTop* findBestTop(std::vector<GraphTop*>& openSet)` возвращает вершину из `openSet` с минимальным значением `fFunc`;
5. `static bool isInOpenSet(GraphTop* top, std::vector<GraphTop*>& openSet)` – проверяет существует ли переданная вершина в переданном векторе;
6. `static void removeFromOpenSet(GraphTop* top, std::vector<GraphTop*>& openSet)` – удаляет переданную вершину из переданного вектора;
7. `static std::string reconstructPath(std::map<GraphTop*, GraphTop*>& passedTops, GraphTop* start, GraphTop* goal)` – строит путь по переданному словарию;

Функция принимает `std::map<GraphTop*, GraphTop*>& passedTops` ссылка на словарь, `GraphTop* start` – указатель на начальную вершину, `GraphTop* goal` – указатель на конечную вершину, затем создаем строку `resString` и `curr` – указатель на конечную вершину, затем пока не дойдем до начальной вершины приравниваем к `curr` значение хранящиеся в словаре по ключу `curr` и добавляем имя текущей вершины к `resString`, в конце возвращаем перевернутую `resString`.

8. `std::string findBestWay(char start, char goal)` – функция алгоритма A*:
Функция принимает `char start` – имя начальной вершины и `char goal` – имя конечной вершины, создает `closedSet` – вектор обработанных вершин, `openSet` – вектор вершин, которые предстоит обработать, `passedTops` – словарь вершин, используется для построения пути. Затем создаем переменную для хранения текущей вершины и записываем в нее стартовую, добавляем в вектор `openSet`. Потом запускает цикл `while`, который останавливается если `openSet` пустой. Затем текущей вершине приравниваем результат функции `findBestTop(openSet)`, если текущая вершина – конечная, то возвращаем результат функции `reconstructPath`, передавая туда словарь, текущую и конечную вершины, если нет то удаляем текущую вершину из `openSet` и добавляем в `closedSet`, затем проходимся по всем смежным вершинам к текущей, если она есть в списке обработанных, то пропускаем ее, если нет, то кладем в `openSet` и считаем `tentativeGScore` равный значению `gFunc` для текущей вершины и вес ребра из текущей в смежную, если `tentativeGScore` меньше значения `gFunc` для смежной, то добавляем путь в словарь из смежной в текущую, и обновляем значения `gFunc` и `fFunc` у текущей. В конце если функция не нашла пути, то возвращает строку “0”.

Сложность алгоритма

По времени сложность алгоритма A^* можно оценить как $O(|E|(|E| - 1))$, так как в худшем случае проверяются все узлы и все смежные с данным узлом вершины.

По памяти сложность можно оценить как $O(|V| + |E|)$, так как приходится хранить все вершины и все ребра.

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.

Жадный алгоритм:

```
root@DESKTOP-1605NTE:~/m...  
a p  
a q 3.0  
q p 1.0  
a b 1.0  
b c 1.0  
b d 2.0  
d c 1.0  
Start find way in graph  
Start top is (a)  
Best top for (a) is (b)  
Start top is (b)  
Best top for (b) is (c)  
Start top is (c)  
There is no neighbour or  
Return to top (b)  
Start top is (b)  
Best top for (b) is (d)  
Start top is (d)  
Best top for (d) is (c)  
Start top is (c)  
There is no neighbour or  
Return to top (d)  
Start top is (d)  
There is no neighbour or  
Return to top (b)  
Start top is (b)  
There is no neighbour or  
Return to top (a)  
Start top is (a)  
Best top for (a) is (q)  
Start top is (q)  
Find end top!  
aqp
```

A*

```

a p
a q 3.0
q p 1.0
a b 1.0
b c 1.0
b d 2.0
d c 1.0
Graph after reading and sotring
a:(q,3,3,1,4)(b,1,1,14,15)
q:(p,1,1,0,1)
p:
b:(c,1,1,13,14)(d,2,2,12,14)
c:
d:(c,1,1,13,14)
Best top in queue:a
a:(q,3)(b,1)
Check all adjacent for:a
Add to the queue:q
Add to the queue:b
Best top in queue:q
q:(p,1)
Check all adjacent for:q
Add to the queue:p
Best top in queue:p
p:
End was finded
Result :
aqp

```

| Входные данные | Выходные данные для жадного алгоритма | Выходные данные для алгоритма A* |
|--|---------------------------------------|----------------------------------|
| a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 | abcde | ade |

| | | |
|---|-------|------|
| 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 | eca |
| a f a b 1.0 a c 2.0 a d 1.0 b d 3.0 c e 2.0 d e 2.0 e f 2.0 d f 6.0 | adbef | adef |
| a e a b 10.0 a c 8.0 c b 1.0 b d 3.0 d e 7.0 b e 9.0 c e 11.0 | acdbe | acbe |
| a f a b 1.0 a c 3 c d 3.0 b e 2.0 e d 1.0 e f 2.0 | abef | abef |

Вывод.

В ходе выполнения лабораторной работы были изучены и реализованы алгоритм A^* и жадный алгоритм для нахождения пути в графе.

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ ДЛЯ ЖАДНОГО АЛГОРИТМА

```
#include <iostream>
#include <vector>
#include <tuple>
#include <stack>
#include <string>

using namespace std;

using Edge = tuple<string, string, double, bool>; //ребро

class EdgesList { //граф
private:
    vector<Edge*> edges; //вектор ребер
    stack<string> stackk; //стек вершин
public:
    EdgesList() = default;
    void addEdge(Edge* edge) { //добавление вершин в граф
        edges.push_back(edge);
    }
    void printList() { //печать ребер
        for (auto & edge : edges) {
            std::cout << get<0>(*edge) << " ";
            std::cout << get<1>(*edge) << " ";
            std::cout << get<2>(*edge) << " ";
            std::cout << get<3>(*edge) << endl;
        }
    }

    void printStack(){ //печать перевернутого стека
        std::stack<string> newS; //создаем новый стек
        while (!stackk.empty()) { //пока стек не пустой
            newS.push(stackk.top()); //добавляем вершины в новый
            stackk.pop(); //и убираем их из старого
        }
        while (!newS.empty()) { //пока нвый не пустой
            cout << newS.top(); //печатаем вершину
            newS.pop(); //и убираем ее со стека
        }
    }

    void findWay(string startTop, string endTop) {
        std::cout << "Start find way in graph" << std::endl;
        stackk.push(startTop); //кладем стартовую вершину на стек
        bool finded = false; //флаг для остановки поиска
        while(!finded) {
            std::cout << "Start top is (" << startTop << ")" << std::endl;
            Edge bestEdge = {"", "", 1000.0, false}; // будем сравнивать все ребра
            с данным
            for (auto & edge : edges) { //проходимся по каждому ребру в графе
                if ((get<0>(*edge) == startTop) && (get<2>(*edge) <
get<2>(bestEdge)) && get<3>(*edge) != true) { //если оно является не пройденным,
инцидентным данному, а также его вес меньше текущего лучшего
                    bestEdge = *edge;
                }
            }
            if(get<1>(bestEdge) == endTop) { //если смежное является конечным
                std::cout << "Find end top!" << std::endl;
                finded = true; //значит нашли конечную
            }
        }
    }
};
```

```

        stackk.push(endTop); //и добавляем ее на стек
        break;
    }
}
if(finded) break;
if(get<0>(bestEdge).empty()){ //если не нашли инцидентную или они все
пройденны
    std::cout << "There is no neighbour or there were passed for top
(" << startTop << ")"<< std::endl;
    string poppedTop = stackk.top(); //снимаем со стека с вершины и
приравниваем к новой переменной
    stackk.pop();
    std::cout << "Return to top (" << stackk.top() << ")" << std::endl;
    for(int i = 0; i < edges.size(); i++){ //ищем инцидентное ребро
между удаленной вершиной и вершиной стека
        if(get<1>(*edges[i]) == poppedTop && get<0>(*edges[i]) ==
stackk.top()) {
            get<3>(*edges[i]) = true;
            startTop = stackk.top(); //делаем верхушку стека стартовой
вершиной
        }
    }
}
else {
    std::cout << "Best top for (" << startTop << ") is (" <<
get<1>(bestEdge) << ")" << std::endl;
    stackk.push(get<1>(bestEdge)); //добавляем вершину на стек
    startTop = get<1>(bestEdge); //и делаем ее стартовой
}
}
printStack();
}
};

int main() {
    string vertex1, vertex2; // вершины для поиска пути
    cin >> vertex1 >> vertex2;
    std::string from;
    std::string to;
    double defaultEdgeLength = 1.0;
    EdgesList edgesList; //инициализируем граф
    while (cin >> from >> to >> defaultEdgeLength) { //считываем пока cin не вернет
false
        auto* currentEdge = new Edge{ from, to, defaultEdgeLength, false};
        //создаем новое ребро и добавляем его в граф
        edgesList.addEdge(currentEdge);
    }
    //edgesList.printList();
    edgesList.findWay(vertex1, vertex2); //вызов функции поиска пути в графе
    std::cout << std::endl;
    return 0;
}

```

ПРИЛОЖЕНИЕ Б

КОД ПРОГРАММЫ ДЛЯ А*

```
#include <iostream>
#include <vector>
#include <map>
#include <cmath>
#include <algorithm>
#include <utility>

class GraphTop;

class Pair { //класс ребро
public:
    GraphTop* top; // вершина
    double weight; //вес ребра
    Pair(GraphTop* top, double weight) : top(top), weight(weight){}; // конструктор
};

class GraphTop { //класс вершина
public:
    char name; //имя вершины
    double gFunc = 0; //значение функции g
    double hFunc = 0; //значение функции h
    double fFunc = 0; //значение функции f
    std::vector<Pair*> adjacentEdges; // смежные ребра
    explicit GraphTop(char name) : name(name){}; // конструктор

    void sortTops(char goal){ //сортировка всемых вершин
        for(auto & adjacentEdge : adjacentEdges){ // заплняем значений функций g,f,h
            adjacentEdge->top->gFunc = adjacentEdge->weight;
            adjacentEdge->top->setHFunc(goal);
            adjacentEdge->top->setFFunc();
        }
        for(int i = 0; i < adjacentEdges.size() ; i++){
            for (int j = i+1; j < adjacentEdges.size(); ++j) {
                if(adjacentEdges[j]->top->fFunc < adjacentEdges[i]->top->fFunc){ //сортируем
ребра пузырьком
                    std::swap(adjacentEdges[j], adjacentEdges[i]);
                }
            }
        }
    };

    void setHFunc(char b){ //эвристическа функция
        hFunc = (double)abs((int)name - (int)b);
    };

    void setFFunc(){
        fFunc = hFunc + gFunc;
    };

    void addAdjacentEdge(GraphTop* top, double weight) { //добавления смежного ребра
        adjacentEdges.push_back(new Pair(top, weight));
    };

    void print(){ //печать смежных ребер
        std::cout << name << ":";
        for(auto & adjacentEdge : adjacentEdges){
            std::cout << "(" << adjacentEdge->top->name << "," << adjacentEdge->weight <<
", " << adjacentEdge->top->gFunc << "," << adjacentEdge->top->hFunc << "," << adjacentEdge-
>top->fFunc<< ")";
        }
    };
};
```



```

void print1(){ //печать смежных ребер без h, g , f
    std::cout << name << ":";
    for(auto & adjacentEdge : adjacentEdges){
        std::cout << "(" << adjacentEdge->top->name << "," << adjacentEdge->weight <<
    ");";
    }
};

};

class EdgesList {
private:
    static GraphTop* findBestTop(std::vector<GraphTop*>& openSet){ //функция нахождения
    вершины с минимальной fFunc в векторе
        GraphTop* bestTop = openSet[0]; //приравниваем возвращаемой вершине первый элемент
    в векторе
        for(int i = 1; i < openSet.size(); i++){ //проходимся по всем вершинам в векторе
            if(openSet[i]->fFunc < bestTop->fFunc) bestTop = openSet[i];
        }
        return bestTop;
    }

    static std::string reconstructPath(std::map<GraphTop*, GraphTop*>& passedTops, GraphTop*
start, GraphTop* goal){ //функция построения пути
        std::string resString; //возвращаемая строка
        GraphTop* curr = goal; //приравниваем текущую вершину к конечной
        resString += goal->name; //добавляем имя конечной вершины к рез строке
        while(curr != start){ //пока не дойдем до начальной
            curr = passedTops[curr]; //по ключу находим следующую вершину
            resString += curr->name; //добавляем ее имя в рез строку
        }
        std::reverse(resString.begin(), resString.end()); //переворачиваем строку
        return resString;
    }

    static bool isInClosedSet(GraphTop* top, std::vector<GraphTop*>& openSet){ //находим
    врешину в векторе
        for (auto & i : openSet) { //проходимся по каждой вершине в векторе
            if(i == top) return true;
        }
        return false;
    }

    static void removeFromOpenSet(GraphTop* top, std::vector<GraphTop*>& openSet){
    //удаление вершины из вектора
        for(auto i = openSet.begin(); i <= openSet.end(); i++){ //с помощью итератора
    проходимся по вершинам в веторе
            if((*i) == top){ //сравниваем значение в итераторе и удаляемую вершину
                openSet.erase(i); //если равны то удаляем
            }
        }
    }

public:
    std::vector<GraphTop*> tops; //вектор всех вершин графа

    void sortTops(char goal){ //сортировка всех смежных вершин у каждой вершины
        for(auto & top : tops){ // проходимся по каждой вершине в векторе и вызываем для нее
    сортировку
            top->sortTops(goal);
        }
    }

    void addEdge(char topName, char adjacentTopName, double weight) { //добавление ребра

```

```

    GraphTop* top = isExistTop(topName); //ищем вершину по имени
    GraphTop* adjacentTop = isExistTop(adjacentTopName); //ищем смежную вершину по имени
    top->addAdjacentEdge(adjacentTop, weight); //вызываем функцию добавление смежного
ребра
    };

    GraphTop* isExistTop(char topName) { //вынкция проверки наличия ребра либо ее создания
        for(auto & top : tops){ //ищем переданную вершину в графе
            if(top->name == topName) return top; //если нашли то возвращаем
        }
        auto* top = new GraphTop(topName); //если не нашли то создаем, добавляем в вектор
вершин и возвращаем
        tops.push_back(top);
        return top;
    };

    void print() { //печать всех вершин
        for (auto & top : tops) {
            top->print();
            std::cout << std::endl;
        }
    };

    std::string findBestWay(char start, char goal){ //поиск пути по алгоритму А*
        std::vector<GraphTop*> closedSet; //обработанные вершины
        std::vector<GraphTop*> openSet; //вершины, которые предстоит обработать
        std::map<GraphTop*, GraphTop*> passedTops; //словарь для пути
        GraphTop* startTop = isExistTop(start); //ищем стартовую вершину
        openSet.push_back(startTop); //добавляем в вектор для обработки
        startTop->gFunc = 0.0; // заплняем значений функций g,f,h
        startTop->setHFunc(goal);
        startTop->setFFunc();
        GraphTop* curTop = startTop; //приравниваем текущей начальную
        while(!openSet.empty()) { //пока в списке есть вершины для обработки
            GraphTop* curTop = findBestTop(openSet); //ищем вершины с наименьшим значением
fFunc
            std::cout << "Current top:" << std::endl;
            curTop->print1();
            std::cout << std::endl;
            if(curTop->name == goal) {
//если нашли конечную вызываем функцию для определения пути
                std::cout << "End was finded";
                return reconstructPath(passedTops, startTop, isExistTop(goal));
            }
            removeFromOpenSet(curTop, openSet); //удаляем тек вершину из списка
обрабатываемых
            closedSet.push_back(curTop); // и добавляем в обработанные
            std::cout << "Check all adjacent for:" << curTop ->name << std::endl;
            for(int i = 0; i < curTop->adjacentEdges.size(); i++){ //проходимся по всем
смежным для текущей
                bool tentativeIsBetter; //вляг для проверки является ли лучшей
                if(isInClosedSet(curTop->adjacentEdges[i]->top, closedSet)) continue; //
если смежная уже обработана, то пропускаем
                double tentativeGScore = curTop->gFunc + curTop->adjacentEdges[i]->weight;
// по данной переменной будем определять, является ли вершина лучшей
                if(!isInClosedSet(curTop->adjacentEdges[i]->top, openSet)) { //если нет в
очереди на обработку, то добавим туда
                    openSet.push_back(curTop->adjacentEdges[i]->top);
                    tentativeIsBetter = true;
                }
                else{
                    if (tentativeGScore < curTop->adjacentEdges[i]->top->gFunc)
tentativeIsBetter = true; // если gFunc у смежной больше
                    else continue;
                }
            }
        }
    }

```

```

        }
        if (tentativeIsBetter){//если вершина лучше
            std::cout << "Adjacent top "<< curTop->adjacentEdges[i]->top->name << "
is better" << std::endl;
            passedTops[curTop->adjacentEdges[i]->top] = curTop; //то добавим ее в
путь
            curTop->adjacentEdges[i]->top->gFunc = tentativeGScore; // заполняем
значений функций g,f,h
            curTop->adjacentEdges[i]->top->setHFunc(goal);
            curTop->adjacentEdges[i]->top->setFFunc();
        }
    }

    }
    return "0";
}
};

int main() {
    EdgesList edgesList; //создание графа
    char a, b, start, goal;
    double weight;
    std::cin >> start >> goal; //считывание начальной и конечной вершин
    while(std::cin >> a >> b >> weight) { //считываем пока не cin не вернет false
        edgesList.addEdge(a, b, weight); //добавление ребра графа
    }
    edgesList.sortTops(goal); //сортируем для каждой вершины смежные по приоритету
    std::cout << "Graph after reading and sotring" << std::endl;
    edgesList.print(); //печатаем граф
    std::cout << edgesList.findBestWay(start, goal); //вызываем функцию нахождения пути в
графе
    return 0;
}

```