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

Студентка гр. 8382 _____ Рочева А.К.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Разработать программы, которые решают задачи построения пути в графе при помощи жадного алгоритма и алгоритма A*.

Задание.

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 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

Индивидуализация.

Вариант 4.

Модификация A* с двумя финишами (требуется найти путь до любого из двух).

Описание жадного алгоритма.

Считывание графа из файла проходит в цикле *while()*. Для каждой вершины проверяется, состоит ли она уже в графе, и если состоит, то в контейнер *map <char, double> edges,* находящемся в ее структуре, добавляется новое ребро. Если же такой вершины нет в графе, то она создается, и в ее контейнер *edges* так же добавляется новое ребро.

Поиск пути происходит в функции *findPath(graph, start, end, path*). В вектор *path* добавляется вершина start и создается вектор *stop*, в котором будут находиться вершины, просматривать которые уже бесполезно (либо эти веришы – листья, либо все вершины, выходящие от них, уже просмотрены и так же находятся в *stop*).

Поиск происходит в цикле *while()*. Сначала проверяется, является ли веткор path пустым. Если да, то цикл прекращается (не удалось найти путь).

На каждом новом шаге выбирается последняя посещенная вершина (последняя вершина в векторе *path*). Если она является конечной вершиной, то цикл прекращается, и в функции *main()* выводится результат (все вершины в векторе *path*). Если же текущая вершина не является конечной, то проверяется, находится ли эта вершина в векторе *stop*. Если да, то она удаляется из вектора *path* и происходит переход к следующей итерации цикла. В противном случае функция *findMin()* возвращает вершину, путь к которой является минимальным из текущей вершины (поиск в *edges*). Если функция возвратила имя текущей вершины, то значит, что в текущей вершине больше нет вершин в *edges*, которые можно посетить и она заносится в вектор *stop*. Происходит переход к новой итерации.

В случае, если *findMin()* вернула имя, отличное от имени текущей вершины, то в вектор *seen* в структуре текущей вершины добавляется эта вершина (в этом векторе лежат вершины, просмотренные из текущей). Затем проверяется, находится ли новая вершина в векторе *path* (наличие цикла). Если находится, и при этом у текущей вершины все вершины из *edges* уже являются просмотренными, то она (текущая вершина) заносится в *stop* и

удаляется из *path*. Если у текущей вершины еще есть вершины, которые можно просмотреть, или если новая вершина не лежит в *path*, то она добавляется в *path.*

Описание структур данных.

1. Структура *Vertex* представляет вершину графа. В ней содержатся поля *map <char, double> edges, vector <char> seen, char name* – имя вершины.
2. *map <char, double> edges* – контейнер для хранения вершин, в которые можно попасть из текущей (они являются ключами), и длин пути до них (значения этих ключей).
3. *vector <char> seen* – вектор для хранения просмотренных вершин, в которые можно попасть из текущей.
4. *vector <Vertex*> graph* – вектор, содержащий указатели на структуры Vertex, для представления графа.
5. *vector <char> path* – вектор для представления результата работы алгоритма. Хранит названия посещенных вершин от начальной до конечной.
6. *vector <char> stop* – вектор для хранения названий вершин, которые больше не стоит посещать.

Описание функций.

1. *void findPath(vector <Vertex*>& graph, char start, char finish, vector <char> &path*) — основная функция для поиска пути в графе. Аргументы: *vector <Vertex*>& graph* – ссылка на граф, *char start, char finish* – имена начальной и конечной вешины, *vector <char> &path* – ссылка на вектор path.
2. *char findMin(Vertex &vertex, const vector <char>& stop)* — функция поиска вершины, в которую можно попасть из вершины *vertex* по ребру с минимальным значением. Аргументы: *Vertex &vertex* — ссылка на

вектор, из соседей которого нужно найти следующую вершину, *const vector <char>& stop* — ссылка на вектор, сожержащий имена вершин, которые больше не нужно посещать. Возвращает имя найденной вершины (или имя вершины *vertex*, если найти новую вершину не удастся).

3. *bool allSeen(Vertex &vert)* — функция, в которой проверяется, все ли вершины из *vert.edges* являются просмотренными (лежат в векторе *vert.seen*). Аргументы: *Vertex &vert* — сам вектор, соседи которого проверяются. Возвращает true, если все просмотрены, иначе — false.
4. *Vertex* returnVertex(vector <Vertex*> graph, char name)* — функция, возвращающая указатель на вершину в *graph* с именем *name*. Если такую вершину невозможно найти, возвращает nullptr. Аргументы: *vector <Vertex*>& graph* – ссылка на граф, *char name* – имя вершины, которую нужно возвратить.
5. *bool inVect(const vector <char>& vect, Vertex *curr*) — функция, возвращающая true, если вершина *curr* находится в векторе *vect.Аргументы: const vector <Vertex*>& vect* — ссылка на вектор, сожержащий указатели на вершины, *Vertex *curr* — указатель на вершину, наличие которой нужно проверить в векторе vect.

Сложность жадного алгоритма.

Сложность по операциям: на каждом шаге цикла просматриваются все ребра, выходящие из текущей вершины. В худшем случае придется обойти весь граф, т.е сложность O(|V|*|E|) (придется посетить все вершины и просмотреть все ребра).

Сложность по памяти: в худшем O(|V|*|E| + |V|), если все вершины соединены со всеми (в каждой вершине хранятся ребра до всех ближайших вершин) + вершины, хранящиеся в векторе path.

Описание алгоритма A*.

Считывание вершин графа происходит так же, как и в программе с жадным алгоритмом, но в отличии от первой программы на вход может подастся две конечные вершины. Если вторая конечная вершина не задана, то ей присваивается значение «\0». Так же отличием является то, что теперь абсолютно для всех поступающих вершин создается структура *Vertex* (в первой программе для листьев она не создавалась).

Поиск пути происходит в функции *findPath()*. В ней создается булевые флаги для двух конечных вершин. Если вторая вершина отсутствует, то ее флагу сразу присваивается значение false. Т.е. в функции одновременно происходит поиск для двух конечных вершин, поэтому все функции вызываются дважды — для поиска пути до первого конца и до второго. Если какой-то флаг имеет значение false, но при этом второй флаг true, то продолжается поиск только для другой конечной вершины.

Для нахождения каждой из конечной вершин создаются вектор *openset* для хранения множества вершин, которые предстоит обработать, и контейнер *map <Vertex*, Vertex*> fromset* для хранения карты пройденных вершин.

Поиск происходит в цикле *while()*. Сначала проверяются значения флагов и состояние *openset* для каждой вершины (если один из *openset* становится пустым, то значит, что нельзя найти путь до вершины, и поиск для этой конечной вершины прекращается). Затем по очереди вызываются функции *checkCurrent()* для каждой из конечных вершин.

Функция *checkCurrent* принимает ссылку на граф, указатели на *openset* и *fromset*, а так же стартовые и конечные вершины. В этой функции проверяется одна из вершин со смежными ей вершинами. Вершина выбирается в функции *findMin()*, которая из вектора *openset* возвращает вершину с самой низкой оценкой f(x) (значение поля вершины *pathFromStart* + значение эвристической оценки) расстояния от этой вершины до конечной (близость символов, обозначающих имена вершин, в таблицу ASCII). В цикле проверяются все вершины в *openset*. При нахождении вершины

функция возвращает указатель на нее, при этом она удаляется из *openset*. Если имя этой вершины совпадает с именем конечной вершины, то вызывается функция *printPath* и возвращается единица. Если имена не совпадают, то поле *isSeen* этой вершины становится true, т. е. она помечается как уже обработанная.

Затем проверяется каждое ребро, выходящее из этой вершины. Если соседняя вершина имеет значение true поля *isSeen*, то она пропускается. Для каждого еще не обработанного соседа вершины вычисляется g(x) — длина пути от стартовой вершины до этого соседа. Затем проверяется, находится ли этот сосед в *openset*. Если нет, то он добавляется туда и устанавливаем true в переменную *tentativeIsBetter* (вводим признак того, что нужно обновить свойства для этой соседней вершины). Если этот сосед уже был в *openset*, то вычисленное ранее значение g(x) сравнивается со значением *pathFromStart* этого соседа. Если значение g(x) оказалось меньше *pathFromStart*, то так же устанавливаем true в переменную *tentativeIsBetter*, т.е был найден более короткий путь от стартовой вершины до этой соседней. Затем, если *tentativeIsBetter* равен true, то мы обновляем свойства этой соседней вершины. Полю *pathFromSet* присваивается значение g(x) и ключом этого соседа в *fromset* становится текущая вершина (вершина, у которой мы смотрим соседей).

Функция возвращает 0, если вершина, которую она рассматривала, не является конечной.

Вывод пути проходит рекурсивно в функции *printPath()*.

Описание структур данных.

1. Структура *Vertex* представляет вершину графа. В ней содержатся поля *map <char, double> edges, char name* – имя вершины, *bool isSeen* – флаг, показывающий, обработа ли уже вершина и double *pathFromStart* – длина пути от стартовой вершины до текущей.

2. *map <char, double> edges* – контейнер для хранения вершин, в которые можно попасть из текущей (они являются ключами), и длин пути до них (значения этих ключей).
3. *vector <Vertex*> graph* – вектор из указателей на структуры *Vertex* для представления графа.
4. *vector <Vertex*> openset* – вектор для хранения вершин, которые еще предстоит обработать.
5. *map <Vertex*, Vertex*> fromset* – контейнер для хранения карты пройденных вершин. Ключом является потомок, значением ключа – родитель, т.е. вершина, из которой мы добрались до ключа.

Описание функций.

1. *void findPath(vector <Vertex*>& graph, char start, char end1, char end2)* — функция для поиска пути в графе. Аргументы функции: *vector <Vertex*>& graph* – ссылка на граф, *char start, char end1, char end2* – имена начальной и двух конечных вешин.
2. *int checkCurrent(vector <Vertex*>& graph, vector <Vertex*>* openset, map <Vertex*, Vertex*>* fromset, char start, char end)* — основная функция алгоритма, в которой проверяется вершина и смежные ей вершины, заполняется *openset* и *fromset*. Аргументы: *vector <Vertex*>& graph* — ссылка на граф, *vector <Vertex*>* openset* — указатель на вектор, хранящий вершины, которые нужно обработать, *map <Vertex*, Vertex*>* fromset* — указатель на карту пройденных вершин, *char start, char end* — имена начальной и конечной вершины. Функция возвращает 1, если был найден путь до конечной вершины, 0 — если еще нет.
3. *Vertex* findMin(vector <Vertex*>& openset, char end1, char end2)* — функция поиска в *opensset* вершины с самой оценкой f(x) (значение

поля вершины *pathFromStart* + значение эвристической оценки). Аргументы: *vector <Vertex*>& openset* — ссылка на вектор указателей на вершины, которые еще нужно просмотреть, *char end1, char end2* — имена конечных вершин. Возвращает указатель на найденную вершину.

4. *void printPath(map <Vertex*, Vertex*>& fromset, char start, char end, vector <Vertex*>& graph)* — рекурсивная функция вывода пути на экран. Аргументы: *map <Vertex*, Vertex*>& fromset* — карта пройденных вершин, *char start, char end* — имена начальной и конечной вершины.
5. *Vertex* returnVertex(vector <Vertex*>& graph, char name)* — функция, возвращающая указатель на вершину в *graph* с именем *name*. Если такую вершину невозможно найти, возвращает nullptr. Аргументы: *vector <Vertex*>& graph* – ссылка на граф, *char name* – имя вершины, которую нужно возвратить.
6. *bool inVect(const vector <Vertex*>& vect, Vertex *curr)* — функция, возвращающая true, если вершина *curr* находится в векторе *vect*. Аргументы: *const vector <Vertex*>& vect* — ссылка на вектор, сожержащий указатели на вершины, *Vertex *curr* — указатель на вершину, наличие которой нужно проверить в векторе vect.
7. *double h(char current, char end)* — функция, возвращающая значение эвристической оценки расстояния до конечной вершины. Аргументы: *char current* — имя текущей вершины, *char end* — имя конечной вершины.

Сложность алгоритма A*.

Сложность по операциям: в худшем случае при плохо подобранной эвристической функции сложность $O(|V|^2)$, т. к. придется просматривать все пути. При очень хорошей подобранной эвристической оценке сложность может быть $O(|V|+|E|)$, т. к. на каждом шаге будет находиться действительно

самая хорошая вершина, т. е. сразу будет строиться правильный путь без шагов назад.

Сложность по памяти: в худшем случае каждый шаг будет неверным и придется просматривать все ребра вершины. Если при этом еще каждый путь до вершины будет короче предыдущего, то придется вершину добавлять в *openset1 ,* т.е сложность будет экспонециальной.

Тестирование

Жадный алгоритм (без вывода промежуточных данных):

Входные данные	Выходные данные
a e a b 1.0 b f 2.0 f w 3.0 f e 5.0	abfe
a e a b 1.0 a g 2.0 b g 2.0 b e 3.0 g e 2.0	abge
a e a b 1.0 a c 2.0 a f 3.0 f e 1.0 b t 2.0 c q 1.0 c b 2.0	afe

Алгоритм A* (без вывода промежуточных данных):

Входные данные	Выходные данные
a w q a b 1.0 a c 1.0 b e 2.0 b t 3.0 b v 4.0 e v 1.0 v q 4.0 v i 5.0	abvq

i w 1.0	
a d f a b 1.0 a c 3.0 b c 1.0 c d 4.0 c e 1.0 e f 1.0	acef
d a z d c 8 c b 3 b a 2 d v 1 v w 1 w x 1 x y 1 y z 1	dvwxyz

Выводы.

В ходе выполнения работы были разработаны программы, решающие задачи построения пути в графе при помощи жадного алгоритма и алгоритма A*.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ФАЙЛА GREEDY.CPP

```
#include <iostream>
#include <map>
#include <vector>
#include <fstream>

using namespace std;

// вершина графа
typedef struct Vertex{
    map <char, double> edges; // название соседней вершины и путь до нее
    vector <char> seen; // просмотренные вершины
    char name;
} Vertex;

// возвращает указатель на вершину в графе с именем name
Vertex* returnVertex(vector <Vertex*> &graph, char name){
    if (graph.empty())
        return nullptr;
    for (auto & vert : graph){
        if (vert->name == name)
            return vert;
    }
    return nullptr;
}

// проверка на содержание вершины с именем name в каком-либо векторе
bool inVect(const vector <char>& vect, char name){
    if (vect.empty())
        return false;
    for (auto & v : vect){
        if (v == name)
            return true;
    }
    return false;
}

// все ли ребра просмотрены
bool allSeen(Vertex &vert){
    if ((vert.seen).empty())
        return false;
    for (auto &edge : vert.edges){
        if (!inVect(vert.seen, edge.first))
            return false;
    }
```

```
    return true;
}

char findMin(Vertex &vertex, const vector <char>& stop){

    if (vertex.edges.empty() || allSeen(vertex)){ // в вершине больше
соседних вершин, которые можно просмотреть
        cout << vertex.name << " has no free edges" << endl;
        return vertex.name;
    }

    double min = -1;
    char tmp = vertex.name; // вернет свое имя, если не найдется min

    for (auto & vert : vertex.edges){
        if (inVect(vertex.seen, vert.first)){
            // не рассматриваем вершину, которая уже просмотрена
            continue;
        }
        if (inVect(stop, vert.first)){
            // не рассматриваем вершину, которая лежит в stop
            continue;
        }
        if (min == -1 || vert.second < min){
            min = vert.second;
            tmp = vert.first;
        }
    }
    cout << "min: " << tmp << endl;
    return tmp;
}

void findPath(vector <Vertex*>& graph, char start, char finish, vector
<char> &path){

    path.push_back(start);
    vector <char> stop; // содержит вершины, в которые уже точно можно не
заходить

    while (true){
        cout << endl;
        cout << "Current path: ";
        for (auto & i : path)
            cout << i;
        cout << endl;
        cout << "Current stop list: ";
        for (auto & i : stop)
            cout << i;
```

```
        cout << endl;

        char current = path[path.size()-1]; // последняя просмотренная
вершина
        cout << "Current vertex: " << current << endl;
        if (current == finish){
            cout << "Finish!" << endl;
            break;
        }

        Vertex *v = returnVertex(graph, current);
        if (v == nullptr){
            // текущая вершина – лист
            cout << current << " is empty"<< endl;
            cout << "Pop " << current << " from current path and push it
in stop list" << endl;
            stop.push_back(current);
            path.pop_back();
            continue;
        }
        char min = findMin(*returnVertex(graph, current), stop);
        if (min == current){
            // из вершины больше некуда идти, все просмотрено
            cout << "No more ways from " << current << endl;
            stop.push_back(current);
            path.pop_back();
            continue;
        }
        returnVertex(graph, current)->seen.push_back(min);
        if (inVect(path, min)){
            if (allSeen(*v)){
                stop.push_back(current);
                path.pop_back();
                continue;
            }
        }
        path.push_back(min);
        cout << "Push " << min << endl;
    }

}

int main() {

    /*   ifstream fin("../input.txt");
       if (!fin)
           return 0;
   */
```

```
while (true){
    cout << "Enter data:" << endl;
    char start, end;
    //    if (!(fin >> start))
    //        break;
    //    fin >> end;
    vector <Vertex*> graph;
    if (!(cin >> start) || start == '!')
        break;
    cin >> end;

    while (true){
        char v1, v2;
        double weight;
        //    if (!(fin >> v1) || v1 == '.')
        //        break;
        if (!(cin >> v1) || v1 == '.' || v1 == '!')
            break;
        //    fin >> v2 >> weight;
        cin >> v2 >> weight;

        if (auto *v = returnVertex(graph, v1)){
            v->edges[v2] = weight;
        } else {
            v = new Vertex;
            v->name = v1;
            v->edges[v2] = weight;
            graph.push_back(v);
        }
    }

    cout << "Graph:" << endl;
    for (auto & vert : graph){
        cout << vert->name << ": ";
        for (auto & edge : vert->edges)
            cout << edge.first << "(" << edge.second << ") ";
        cout << endl;
    }

    vector <char> path;
    findPath(graph, start, end, path);
    cout << "\nPath from " << start << " to " << end << ":" << endl;
    for (auto & i : path)
        cout << i;
    cout << endl << endl;
}
```

```
    //  fin.close();

    return 0;
}
```

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ФАЙЛА ASTAR.CPP

```
#include <iostream>
#include <map>
#include <vector>
#include <fstream>
#include <functional>
#include <cmath>

using namespace std;

//вершина графа
typedef struct Vertex{
    map <char, double> edges; // соседние вершины с длинами путей к ним
    bool isSeen; // показывает, обработа ли вершина
    double pathFromStart; // g(x) – расстояние от начальной вершины до
текущей
    char name;
} Vertex;

// эвристическая оценка расстояния до конечной вершины
double h(char current, char end){
    return end - current;
}

// возвращает указатель на вершину их графа с именем name
Vertex* returnVertex(const vector <Vertex*>& graph, char name){
    if (graph.empty())
        return nullptr;
    for (auto & vert : graph){
        if (vert->name == name)
            return vert;
    }
    return nullptr;
}

// находится ли вершина curr в векторе vect
bool inVect(const vector <Vertex*>& vect, Vertex *curr){
    if (vect.empty())
        return false;
    for (auto & v : vect){
        if (v->name == curr->name)
            return true;
    }
    return false;
}
```

```
// ищем вершину с самой низкой оценкой f(x) = g(x) + h(x)
Vertex* findMin(vector <Vertex*>& openset, char end1, char end2){
    Vertex *cmp = openset[0];
    int ind = 0;
    int i = 0;
    double hCmpEnd1 = h(cmp->name, end1);
    double hCmpEnd2 = h(cmp->name, end2);
    double f;
    if (fabs(hCmpEnd1) < fabs(hCmpEnd2))
        f = hCmpEnd1 + cmp->pathFromStart;
    else
        f = hCmpEnd2 + cmp->pathFromStart;
    for (auto & vert : openset){
        cout << "Vertex: " << vert->name << endl;
        double hEnd1 = h(vert->name, end1);
        double hEnd2 = h(vert->name, end2);
        cout << "to (" << end1 << ") = " << hEnd1 << "; to (" << end2 <<
") = " << hEnd2 << endl;
        double hVert;
        if (fabs(hEnd1) < fabs(hEnd2))
            hVert = hEnd1;
        else
            hVert = hEnd2;
        cout << "choose " << hVert << endl;
        cout << "h(x) for: " << vert->name << " = " << hVert << ", abs ("
<< vert->name << ") = " << fabs(hVert) << endl;
        hVert = fabs(hVert);
        if (((vert->pathFromStart + hVert) < f) || (((vert->pathFromStart
+ hVert) == f) &&
                                                    vert->name > cmp-
>name)){
            f = vert->pathFromStart + hVert;
            cout << "new min: " << vert->name << endl;
            cmp = vert;
            ind = i;
        }
        i++;
    }
    openset.erase(openset.begin() + ind);
    return cmp;
}

void printPath(map <Vertex*, Vertex*>& fromset, char start, char end,
vector <Vertex*>& graph){
    if (end == start){
        cout << end;
        return;
    }
```

```
    printPath(fromset, start, (fromset[returnVertex(graph, end)])->name,
graph);
    cout << end;
}

int checkCurrent(vector <Vertex*>& graph, vector <Vertex*>* openset, map
<Vertex*, Vertex*>* fromset, char start, char end1, char end2){

    auto currentVert = findMin(*openset, end1, end2);
    cout << "Current vertex: " << currentVert->name << endl;
    if (currentVert->name == end1){
        cout << "Path to (" << end1 << ") was found" << endl;
        printPath(*fromset, start, end1, graph);
        return 1;
    }
    if (currentVert->name == end2){
        cout << "Path to (" << end2 << ") was found" << endl;
        printPath(*fromset, start, end2, graph);
        return 1;
    }

    currentVert->isSeen = true;
    cout << "Neighbours:" << endl;
    for (auto & edge : currentVert->edges) {
        cout << edge.first;
        auto neighbour = returnVertex(graph, edge.first);
        if (neighbour->isSeen) {
            cout << " has already been viewed" << endl;
            continue;
        }

        double tentative_g = currentVert->pathFromStart + edge.second;

        bool tentativeIsBetter;
        if (!inVect(*openset, neighbour)) {
            cout << " add to openset" << endl;
            openset->push_back(neighbour);
            tentativeIsBetter = true;
        } else {
            tentativeIsBetter = tentative_g < neighbour->pathFromStart;
        }

        if (tentativeIsBetter) {
            (*fromset)[neighbour] = currentVert;
            neighbour->pathFromStart = tentative_g;
        }
    }
    return 0;
```

```
}

void findPath(vector <Vertex*>& graph, char start, char end1, char end2) {

    vector<Vertex *> openset1; // вершины, которые стоит обработать
    map<Vertex *, Vertex *> fromset1; // карта пройденных вершин
    returnVertex(graph, start)->pathFromStart = 0;
    openset1.push_back(returnVertex(graph, start)); // сразу добавляем
стартовую точку

    while (!openset1.empty()) {
        cout << "============" << endl;
        if (checkCurrent(graph, &openset1, &fromset1, start, end1, end2))
{
            return;
        }
    }
}


int main(){
/*
    ifstream fin("../input.txt");
    if (!fin)
        return 0;*/

    string s;
    while (true){
        cout << "Enter data" << endl;
        char start, end1, end2;
        if (!(cin >> start) || start == '!')
            break;
        getline(cin, s);
        end1 = s[1];
        if (s[3]){
            end2 = s[3];
        } else {
            end2 = '\0';
            cout << "\nWithout second end." << endl;
        }
        vector <Vertex*> graph;

        while (true){
            char v1, v2;
            double weight;
            if (!(cin >> v1) || v1 == '.')
                break;
            cin >> v2 >> weight;
```

```
            if (auto *v = returnVertex(graph, v1)){
                v->edges[v2] = weight;
            } else {
                v = new Vertex;
                v->name = v1;
                v->edges[v2] = weight;
                v->isSeen = false;
                graph.push_back(v);
            }
            if (auto *v = returnVertex(graph, v2)){
                continue;
            } else {
                v = new Vertex;
                v->name = v2;
                v->isSeen = false;
                graph.push_back(v);
            }
        }

        cout << "Graph:" << endl;
        for (auto & vert : graph){
            cout << vert->name << ": ";
            for (auto & edge : vert->edges)
                cout << edge.first << "(" << edge.second << ") ";
            cout << endl;
        }

        vector <char> path;
        findPath(graph, start, end1, end2);
        cout << endl << endl;
    }

   // fin.close();

    return 0;
}
```