

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. 8303

Стукалев А.И.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить и реализовать на языке программирования C++ жадный алгоритм поиска пути в графе и алгоритм A* поиска кратчайшего пути в графе между двумя заданными вершинами.

Формулировка задания.

Вариант 5. Реализовать алгоритм Дейкстры поиска пути в графе (на основе кода A*).

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.

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

Соответствующие выходные данные для Жадного алгоритма

abcde

Соответствующие выходные данные для A*

ade

Описание алгоритмов.

Жадный алгоритм:

В начале список рёбер сортируется по возрастанию их весов. Алгоритм начинает поиск из заданной вершины. Начальная вносится в контейнер с путём.

Затем в списке рёбер находится первое соответствие начальной вершины заданной пользователем с начальной вершиной ребра. Так как рёбра отсортированы, то первое соответственное ребро будет с минимальным весом. Конечная вершина этого ребра вносится в контейнер с путём. Затем вместо начальной вершины берётся последняя вершина, внесённая в контейнер пути. Действия по поиску соответствия повторяются, далее конечная вершина этого ребра вносится в контейнер с путём. Данная операция продолжается до тех пор, пока последняя вершина, внесённая в контейнер пути не будет соответствовать конечной вершине, заданной пользователем.

Так же, во время выполнения операций по решению задачи построения пути в ориентированном графе с помощью жадного алгоритма, происходит проверка на наличие петель, то есть конструкций вида:

a d

a b 1.0

b c 1.0

c a 1.0

a d 5.0

В данном случае алгоритм выдаст ответ `abcd`, так будет идти по минимальным вершинам, но, учитывая петлю, на выходе получится `ad`.

Так же, не исключается, что на каком-то шаге алгоритма, из последней вершины, добавленной в контейнер пути, не будет выходить никакого ребра или же ребра нерезультативного (из которого в результате не получится прийти в конечную вершину), в данном случае будет осуществляться шаг назад.

A*:

Поиск начинается из исходной вершины, заданной пользователем. Запоминаются все рёбра, по которым можно пройти из текущей вершины. Далее с помощью отдельной функции находится минимальный путь, учитывая эвристическую близость вершины к искомой (близость символов по таблице ASCII) плюс длины рёбер в этом пути. При выборе очередной вершины, выбираются рёбра, которые начинаются из последней вершины в текущем пути. Текущая вершина добавляется к этому пути, и новый путь заносится в список возможных путей, с увеличением стоимости, равной весу этого ребра. Вершина добавляется в контейнер с просмотренными после того, как будут выбраны все исходящие из неё рёбра. После того, как в текущем пути будет просмотрена последняя вершина, путь удаляется, снова осуществляется поиск минимального пути в графе.

Затем снова осуществляется поиск минимального пути в ориентированном графе, поиск осуществляется до тех пор, пока не достигается последняя вершина.

Алгоритм Дейкстры:

Алгоритм Дейкстры по условию задания должен основываться на алгоритме A*. Такие образом, внося в A* словарь вида: вершина – длина пути для неё. Когда алгоритм A* приходит в некую вершину, происходит фиксация длины для этой вершины, и если она меньше уже известной длины, то заменяется. Также изменилось условие остановки алгоритма – выполняется до

тех пор, пока не опустеет список возможных путей.

Сложность Жадного алгоритма по операциям: $O(N \log N + N)$

$N \log N$ – сортировка, N – путь по графу

Сложность алгоритма Дейкстры и A^* по операциям зависит от $|h(x) - h^*(x)| \leq O(\log h^*(x))$

В худшем случае: $O(2^{(\text{количество рёбер})})$

В лучшем случае: $O(N)$

Сложность алгоритмов по памяти: $O(N)$

N – количество ребер + количество вершин.

Описание структур данных.

1.

```
struct edge
{
    char first;
    char second;
    double weight;
};
```

Структура отражающая сущность ребра графа, так как граф ориентированный: first – начальная вершина ребра, second – конечная, weight – вес ребра.

2.

```
struct ways
{
    char start;
    char finish;
}
```

Структура, использующаяся для хранения заданных пользователем

начальной и конечной вершины, в `start` хранится начальная вершина, в `finish` – конечная (Используется в алгоритме A* и алгоритме Дейкстры).

3.

```
struct mp
{
    string curr_path;
    double length;
};
```

Структура, используемая для хранения путей и их длин. `Curr_path` – путь, состоящий из вершин, `length` – его длин, равная сумме длин рёбер от начальной, заданной пользователем, до конечной в `curr_path` и эвристической оценки расстояния от конечной вершины в `curr_path` к конечной, заданной пользователем (Используется в алгоритме A* и алгоритме Дейкстры).

4.

```
struct res
{
    char ver;
    double weigth;
};
```

Структура, используемая для хранения пары: вершины и пути до неё. `Ver` – наименование вершины, `weigth` – путь до неё (используется в алгоритме Дейкстры).

5.

```
class Graph
{
    char start;
    char finish;
    vector <edge> edges;
    vector <char> path;
}
```

Класс, необходимый для работы Жадного алгоритма, `start` – начальная вершина, заданная пользователем, `finish` – конечная вершина, заданная пользователем, `edges`- контейнер типа `edge`, используется для хранения всех рёбер, `path` - контейнер, использующийся для хранения текущего пути.

6.

```
class Graph
{
    vector <edge> edges;
    vector <char> path;
    vector <map> map_path;
}
```

Класс, необходимый для работы алгоритма A^* , `edges`- контейнер типа `edge`, используется для хранения всех рёбер, `path` - контейнер, использующийся для хранения просмотренных путей, `map_path` – контейнер типа `map`, использующийся для хранения возможных путей.

7.

```
class Graph
{
    vector <edge> edges;
    vector <char> path;
    vector <map> map_path;
    vector <res> result;
}
```

Класс, необходимый для работы алгоритма Дейкстры, `edges`- контейнер типа `edge`, используется для хранения всех рёбер, `path` - контейнер, использующийся для хранения просмотренных путей, `map_path` – контейнер типа `map`, использующийся для хранения возможных путей, `result` – контейнер типа `res`, использующийся для

хранения пар: вершина и путь до неё.

Описание функций Жадного алгоритма.

1. void input_data()

Функция класса Graph. Происходит считывание из консоли введенных пользователем начальной и конечной вершины, рёбер с их весом. Каждое введенное ребро добавляется в контейнер edges. Рёбра сортируются по возрастанию.

2. void greedy()

Функция класса Graph. Запускается цикл while, который работает до тех пор, пока текущая просматриваемая вершина не окажется конечной, введенной пользователем. Внутри запускается цикл for, который пробегается по всему контейнеру с рёбрами, в нём сначала происходит проверка на наличие петель с помощью ещё одного цикла for, который пробегается по текущему пути. Далее среди рёбер в контейнере edges ищется соответствие первой вершины ребра с последней вершиной текущего пути, если совпадение найдено, вторая вершина добавляется в текущий путь, алгоритм продолжает работу снова, если же совпадения не обнаружено, то происходит откат на одну вершину назад, и удаление последней добавленной вершины в текущий путь. Продолжается до тех пор, пока последняя добавленная вершина текущего пути не будет эквивалентна конечной вершине, введенной пользователем.

3. void print_result()

Функция класса Graph для вывода результата поиска на экран.

3. bool compare(edge one, edge two)

Функция компаратор. Принимает две переменные типа edge(ребро), сравнивает вес, возвращает результат сравнения (true либо false).

Описание функций алгоритма A*.

1. void input_data()

Функция класса Graph. Происходит считывание из консоли введенных пользователем начальной и конечной вершины, ребер с их весом. Каждое введенное ребро добавляется в контейнер edges. Вес ребер при добавлении проверяется на не отрицательность.

2. `double func_h(char one, char two)`

Функция, вычисляющая близость символов по таблице ASCII. В качестве аргументов принимает две вершины типа char. Возвращает разницу символов по таблице ASCII.

3. `void work()`

Функция класса Graph для поиска минимального пути в графе. Функция итеративно проходит по графу, на каждом шаге выбирая ребро с минимальной эвристической функцией. Алгоритм заканчивает работу если находит конечную вершину, и выводит сообщение, если такого пути нет.

4. `int min_search()`

Функция класса Graph для нахождения ребра с минимальной эвристической функцией. Для выполнения своей задачи функция проходит по всем текущим путям в векторе map_path и возвращает индекс минимального элемента.

5. `void print_result()`

Функция класса Graph для вывода результата поиска на экран.

Описание функций алгоритма Дейкстры.

1. `void input_data()`

Функция класса Graph. Происходит считывание из консоли введенных пользователем начальной и конечной вершины, ребер с их весом. Каждое введенное ребро добавляется в контейнер edges. Вес ребер при добавлении проверяется на не отрицательность. Все вершины словаря заносятся в словарь, начальная вершина принимает вес ноль, все остальные, на данный момент, - бесконечный вес.

2.

`void deikstra()`

Функция класса Graph для поиска пути в графе. Функция итеративно проходит по графу, на каждом шаге выбирая ребро с минимальной эвристической функцией. На очередном шаге проверяется длина пути для текущей вершины, если она меньше нынешней, то заменяется в словаре. Алгоритм заканчивает работу, когда контейнер с возможными путями будет пуст.

3. `double func_h(char one, char two)`

Функция, вычисляющая близость символов по таблице ASCII. В качестве аргументов принимает две вершины типа char. Возвращает разницу символов по таблице ASCII.

4. `void print_result()`

Функция класса Graph для вывода результата работы алгоритма.

5. `int min_search()`

Функция класса Graph для нахождения ребра с минимальной эвристической функцией. Для выполнения своей задачи функция проходит по всем текущим путям в векторе `map_path` и возвращает индекс минимального элемента.

Способ хранения частичных решений.

Частичные решения хранятся в векторе типа `vector<ways>` для A* и алгоритма Дейкстры и в векторе типа `vector<char>` для Жадного алгоритма.

Тестирование.

Жадный алгоритм

1.

```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
a b 1.0
b c 1.0
c a 1.0 !
Ribs before sorting:
a -> d with weight: 5
a -> b with weight: 1
b -> c with weight: 1
c -> a with weight: 1
Ribs after sorting:
a -> b with weight: 1
b -> c with weight: 1
c -> a with weight: 1
a -> d with weight: 5

algorithm start
current rib:a -> b with weight: 1
insert vertex to the path: b
removing an unnecessary edge: a -> b with weight: 1
current rib:b -> c with weight: 1
insert vertex to the path: c
removing an unnecessary edge: b -> c with weight: 1
current rib:c -> a with weight: 1
insert vertex to the path: a
removing an unnecessary edge: c -> a with weight: 1
current rib:a -> d with weight: 5
insert vertex to the path:a
insert vertex to the path: d
removing an unnecessary edge: a -> d with weight: 5
path founded!
ad
```

2.

```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0 !
abcde
```

3.

```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
a 1
a b 1
a f 3
b c 5
b g 3
f g 4
c d 6
d m 1
g e 4
e h 1
e n 1
n m 2
g i 5
i j 6
i k 1
j l 5
m j 3 !
abgenmj1
```

A-STAR

1.

```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
a d
a d 5.0
a b 1.0
b c 1.0
c a 1.0 !

algorithm start
adding possible paths from the starting vertex:
    current rib:a -> d with weight: 5
    adding new rib to map path:a -> d with weight: 5
    current rib:a -> b with weight: 1
    adding new rib to map path:a -> b with weight: 1
    current rib:b -> c with weight: 1
    current rib:c -> a with weight: 1

start search for a new path

search for a new min index:
    calculated prefix function for vertices:d and d is 0
comparing the vertex index with the current minimum: current is 5 and current min is 39
    check: if vertex is viewed d
    this vertex is not viewed!
    calculated prefix function for vertices:d and d is 0
a new current min is 5
    calculated prefix function for vertices:d and b is 2
comparing the vertex index with the current minimum: current is 3 and current min is 5
    check: if vertex is viewed b
    this vertex is not viewed!
    calculated prefix function for vertices:d and b is 2
a new current min is 3
a new min index founded: 0
final vertex is reached!
result of programm:
ad
```

2.

```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0 !
ade
```

3.

```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
a 1
a b 1
a f 3
b c 5
b g 3
f g 4
c d 6
d m 1
g e 4
e h 1
e n 1
n m 2
g i 5
i j 6
i k 1
j l 5
m j 3 !
abgenmj1
```

Алгоритм Дейкстры.

1.

```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
a e
a d 5.0
a b 3.0
b c 1.0
c d 1.0
d e 1.0!

algorithm start
adding possible paths from the starting vertex:
    current rib:a -> d with weight: 5
    adding new rib to map path:a -> d with weight: 5
    current rib:a -> b with weight: 3
    adding new rib to map path:a -> b with weight: 3
    current rib:b -> c with weight: 1
    current rib:c -> d with weight: 1
    current rib:d -> e with weight: 1

start search for a new path

search for a new min index:
    calculated prefix function for vertices:e and d is 1
comparing the vertex index with the current minimum: current is 6 and current min is 42
    calculated prefix function for vertices:e and d is 1
a new current min is 6
    calculated prefix function for vertices:e and b is 3
comparing the vertex index with the current minimum: current is 6 and current min is 6
a new min index founded: 0
search for paths from the current vertex: d
    current rib:a -> d with weight: 5
    current rib:a -> b with weight: 3
    current rib:b -> c with weight: 1
    current rib:c -> d with weight: 1
    current rib:d -> e with weight: 1
    adding new rib to map path:d -> e with weight: 1
deikstra: comparing the current path length to the vertex with the new
    - current path length for vertex e is 2147483647 and new length is 6
a new path length for vertex e is 6

start search for a new path

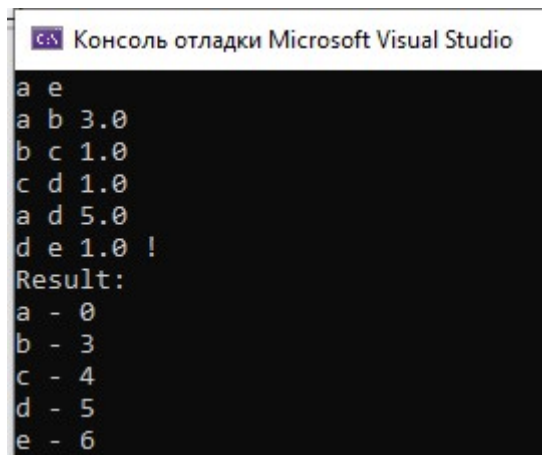
search for a new min index:
    calculated prefix function for vertices:e and b is 3
comparing the vertex index with the current minimum: current is 6 and current min is 42
    calculated prefix function for vertices:e and b is 3
a new current min is 6
    calculated prefix function for vertices:e and e is 0
comparing the vertex index with the current minimum: current is 6 and current min is 6
a new min index founded: 0
search for paths from the current vertex: b
    current rib:a -> d with weight: 5
    current rib:a -> b with weight: 3
    current rib:b -> c with weight: 1
    adding new rib to map path:b -> c with weight: 1
deikstra: comparing the current path length to the vertex with the new
    - current path length for vertex c is 2147483647 and new length is 4
a new path length for vertex c is 4
    current rib:c -> d with weight: 1
    current rib:d -> e with weight: 1

start search for a new path
```



```
search for a new min index:
    calculated prefix function for vertices:e and e is 0
comparing the vertex index with the current minimum: current is 6 and current min is 42
    calculated prefix function for vertices:e and e is 0
a new current min is 6
    calculated prefix function for vertices:e and c is 2
comparing the vertex index with the current minimum: current is 6 and current min is 6
a new min index founded: 0
final vertex is reached!
result of programm:
Result:
for vertex e length is 6
```

2.



Консоль отладки Microsoft Visual Studio

```
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0 !
Result:
a - 0
b - 3
c - 4
d - 5
e - 6
```

3.

```
Консоль отладки Microsoft Visual Studio

a 1
a b 1
a f 3
b c 5
b g 3
f g 4
c d 6
d m 1
g e 4
e h 1
e n 1
n m 2
g i 5
i j 6
i k 1
j l 5
m j 3 !
Result:
a - 0
b - 1
c - 6
d - 12
e - 8
f - 3
g - 4
h - 9
i - 9
j - 14
k - 10
l - 19
m - 11
n - 9
```

Выводы.

В ходе выполнения лабораторной работы были изучены и реализованы на языке программирования C++ жадный алгоритм поиска пути в графе и алгоритм A* поиска минимального пути в графе между двумя заданными пользователем вершинами.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

Жадный алгоритм.

```
#include <iostream>
#include <vector>
#include <algorithm>

struct edge
{
    char first;
    char second;
    double weight;
};

bool compare(edge one, edge two)
{
    return one.weight < two.weight;
}

class Graph
{
    char start;
    char finish;
    std::vector <edge> edges;
    std::vector <char> path;
public:
    Graph()
    {

    }

    void input_data()
    {
        std::cin >> start >> finish;
        path.push_back(start);
        char tmp;
        edge rib;
        std::cin >> tmp;
        while (tmp != '!') //запись рёбер в контейнер
        {
            rib.first = tmp;
            if (!(std::cin >> rib.second))
                break;
            if (!(std::cin >> rib.weight))
                break;
            edges.push_back(rib);
            std::cin >> tmp;
        }
    }

    void greedy()
    {
        int count = 0;
        bool log = false;
        std::cout << "Ribs before sorting:\n";
        for(int i = 0; i < edges.size(); i++)
        {
            std::cout << edges[i].first << " -> " << edges[i].second << " with
weight: " << edges[i].weight << "\n";
        }
    }
}
```

```

sort(edges.begin(), edges.end(), compare); //сортировка рёбер по возвратанию
std::cout << "Ribs after sorting:\n";
for(int i = 0; i < edges.size(); i++)
{
    std::cout << edges[i].first << " -> " << edges[i].second << " with
weight: " << edges[i].weight << "\n";
}
std::cout << "\nalgorithm start\n";
while (path.back() != finish)
{
    for (size_t i = 0; i < edges.size(); i++)
    {
        std::cout << "current rib:" << edges[i].first << " -> " <<
edges[i].second << " with weight: " << edges[i].weight << "\n";
        for (size_t i = 0; i < path.size(); i++)
        {
            if (path[i] == start)
                count++;
            if (count == 2)
            {
                path.clear(); //удаление петель
                path.push_back(start);
                std::cout << "insert vertex to the path:" << start << "\n";
            }
        }
        count = 0;
        if (edges[i].first == path.back())
        {
            path.push_back(edges[i].second);
            std::cout << "insert vertex to the path: " << edges[i].second <<
"\n";
            std::cout << "removing an unnecessary edge: " << edges[i].first
<< " -> " << edges[i].second << " with weight: " << edges[i].weight << "\n";
            edges.erase(edges.begin() + i); //удаление ребра из контейнера,
потому что далее его использовать нет необходимости
            log = true;
            break;
        }
        else
            log = false;
    }
    if (log == false)
    {
        std::cout << "return to the privious vertex from " << path.back();
        for (size_t i = 0; i < edges.size(); i++)
        {
            if (edges[i].second == path.back())
            {
                std::cout << "delete rib without a true path: " <<
edges[i].first << " -> " << edges[i].second << " with weight: " << edges[i].weight
<< "\n";
                edges.erase(edges.begin() + i); //удаление вершины, из
которой нет пути
            }
        }
        path.pop_back();
        std::cout << " to " << path.back() << "\n";
    }
}
std::cout << "path founded!\n";
}
void print_reuslt()
{

```

```

        for (size_t i = 0; i < path.size(); i++)
            std::cout << path[i];
    }
    ~Graph()
    {
        /*delete start;
        delete finish;
        delete <edge> edges;
        delete <char> path;*/
    }
};
int main()
{
    Graph tmp;
    tmp.input_data();
    tmp.greedy();
    tmp.print_reuslt();
}

/*
a l
a b 1
a f 3
b c 5
b g 3
f g 4
c d 6
d m 1
g e 4
e h 1
e n 1
n m 2
g i 5
i j 6
i k 1
j l 5
m j 3 !
*/
/*
a d
a b 1.0
b c 1.0
c a 1.0
a d 5.0 !
*/

```

A*.

```

#include <iostream>
#include <vector>
#include <algorithm>
#include "math.h"

```

```

struct edge
{
    char first;
    char second;
    double weight;
};

```

```

struct ways
{
    char start;
    char finish;
}way;

struct maps
{
    std::string curr_path;
    double length;
};

double func_h(char one, char two)
{
    std::cout << "\tcalculated prefix function for vertices:" << one << " and " <<
two << " is " << abs((int)(one)-(int)(two)) << "\n";
    return abs((int)(one)-(int)(two));
}
class Graph
{
    std::vector <edge> edges;
    std::vector <char> path;
    std::vector <char> close;
    std::vector <maps> map_path;
public:

    Graph()
    {

    }

    void input_data()
    {
        std::cin >> way.start >> way.finish;
        path.push_back(way.start);
        char tmp;
        edge rib;
        std::cin >> tmp;
        while (tmp != '!')//добавление рёбер в контейнер
        {
            rib.first = tmp;
            if (!(std::cin >> rib.second))
                break;
            if (!(std::cin >> rib.weight))
                break;
            edges.push_back(rib);
            std::cin >> tmp;
        }
    }

    void work()
    {
        std::cout << "\nalgorithm start\n";
        //sort(edges.begin(), edges.end(), compare);
        std::string buf = "";
        buf += way.start;
        std::cout << "adding possible paths from the starting vertex:\n";
        for (size_t i = 0; i < edges.size(); i++)
        {
            std::cout << "\tcurrent rib:" << edges[i].first << " -> " <<
edges[i].second << " with weight: " << edges[i].weight << "\n";

```

```

        if (edges[i].first == way.start)//добавление возможных путей из
начальной вершины
        {
            std::cout << "\tadding new rib to map path:" << edges[i].first << "
-> " << edges[i].second << " with weight: " << edges[i].weight << "\n";
            buf += edges[i].second;
            map_path.push_back({ buf, edges[i].weight });
            buf.resize(1);
        }
    }
    path.push_back(way.start);
    while (!map_path.empty())
    {
        std::cout << "\nstart search for a new path\n";
        size_t index = min_search();
        if (map_path[index].curr_path.back() == way.finish)//завершение работы,
если конечная вершина достигнута
        {
            std::cout << "final vertex is reached!\nresult of programm:\n";
            print_reuslt(index);
            return;
        }
        std::cout << "search for paths from the current vertex: " <<
map_path[index].curr_path.back() << "\n";
        for (size_t i = 0; i < edges.size(); i++)
        {
            std::cout << "\tcurrent rib:" << edges[i].first << " -> " <<
edges[i].second << " with weight: " << edges[i].weight << "\n";
            if (edges[i].first == map_path[index].curr_path.back())
            {
                std::cout << "\tadding new rib to map path:" << edges[i].first
<< " -> " << edges[i].second << " with weight: " << edges[i].weight << "\n";
                map_path.push_back({ map_path[index].curr_path +
edges[i].second, edges[i].weight + map_path[index].length });//добавляем минимальную
вершину
            }
        }
        path.push_back(map_path[index].curr_path.back());
        std::cout << "remove current path: " << map_path[index].curr_path <<
"\n";
        map_path.erase(map_path.begin() + index);//удаляем предыдущий путь
    }
    std::cout << "\nno path founded\n";
}

bool is_view(char value)
{
    std::cout << "\tcheck: if vertex is viewed " << value << "\n";
    for (size_t i = 0; i < path.size(); i++)
    {
        if(path[i] == value)//проверка на то, просмотрена ли вершина
        {
            std::cout << "\tthis vertex is viewed!\n";
            return true;
        }
    }
    std::cout << "\tthis vertex is not viewed!\n";
    return false;
}

size_t min_search()
{
    std::cout << "\nsearch for a new min index:\n";
    double min = 0;

```

```

        for (size_t i = 0; i < edges.size(); i++)
            min += edges[i].weight;
        min += 31; //инициализирование начального значения для сравнения
        size_t tmp;
        for (size_t i = 0; i < map_path.size(); i++)
        {
            int tmp = map_path[i].length + func_h(way.finish,
map_path[i].curr_path.back());
            std::cout << "comparing the vertex index with the current minimum:
current is "
                        << tmp;
            std::cout << " and current min is " << min << "\n";
            if (tmp < min) //сравнение индекса вершины с текущим минимальным
            {
                if (is_view(map_path[i].curr_path.back()))
                {
                    map_path.erase(map_path.begin() + i); //удаление уже просмотренных
вершин
                }
                else
                {
                    min = map_path[i].length + func_h(way.finish,
map_path[i].curr_path.back()); //запоминание индекса минимальной вершины
                    std::cout << "a new current min is " << min << "\n";
                    tmp = i;
                }
            }
            std::cout << "a new min index founded: " << tmp << '\n';
            return tmp;
        }
    }
    void print_reuslt(size_t index)
    {
        std::cout << map_path[index].curr_path; //вывод минимального пути
    }
};
int main()
{
    Graph tmp;
    tmp.input_data();
    tmp.work();
}

```

Алгоритм Дейкстры.

```

#include <iostream>
#include <vector>
#include <map>
#include <algorithm>
#include "math.h"

#define INT_MAX 2147483647

struct edge
{
    char first;
    char second;
    double weight;
}

```

```

};

struct ways
{
    char start;
    char finish;
}way;

struct mp
{
    std::string curr_path;
    double length;
};

double func_h(char one, char two)
{
    std::cout << "\tcalculated prefix function for vertices:" << one << " and " <<
two << " is " << abs((int)(one)-(int)(two)) << "\n";
    return abs((int)(one)-(int)(two));
}

class Graph
{
    std::vector <edge> edges;
    std::vector <char> path;
    std::vector <mp> map_path;
    std::map <char,int> result;
public:

    Graph()
    {

    }

    void input_data()
    {
        std::cin >> way.start >> way.finish;

        path.push_back(way.start);
        char tmp;
        edge rib;
        std::cin >> tmp;
        while (tmp != '!')//добавление рёбер в контейнер
        {
            rib.first = tmp;
            if (!(std::cin >> rib.second))
                break;
            if (!(std::cin >> rib.weight))
                break;
            if(rib.weight < 0 )//проверка на отрицательный вес ребра
            {
                std::cout << "WRONG INPUT!";
                return;
            }
            edges.push_back(rib);
            result[rib.first] = INT_MAX;//инициализирование вершин бесконечностью
            result[rib.second] = INT_MAX;
            std::cin >> tmp;
        }
    }

    void deikstra()
    {
        std::cout << "\nalgorithm start\n";
        //sort(edges.begin(), edges.end(), compare);
    }
}

```

```

        std::string buf = "";
        buf += way.start;
        result[way.start] = 0;
        std::cout << "adding possible paths from the starting vertex:\n"
        for (size_t i = 0; i < edges.size(); i++)
        {
            std::cout << "\tcurrent rib:" << edges[i].first << " -> " <<
edges[i].second << " with weight: " << edges[i].weight << "\n";
            if (edges[i].first == way.start)//добавление возможных путей из
начальной вершины
            {
                std::cout << "\tadding new rib to map path:" << edges[i].first << "
-> " << edges[i].second << " with weight: " << edges[i].weight << "\n";
                buf += edges[i].second;
                map_path.push_back({ buf, edges[i].weight });
                buf.resize(1);
            }
        }
        path.push_back(way.start);
        while (!map_path.empty())
        {
            std::cout << "\nstart search for a new path\n";
            size_t index = min_search();
            if (map_path[index].curr_path.back() == way.finish)//завершение работы,
если конечная вершина достигнута
            {
                std::cout << "final vertex is reached!\nresult of programm:\n";
                print_reuslt();
                return;
            }
            std::cout << "search for paths from the current vertex: " <<
map_path[index].curr_path.back() << "\n";
            for (size_t i = 0; i < edges.size(); i++)
            {
                std::cout << "\tcurrent rib:" << edges[i].first << " -> " <<
edges[i].second << " with weight: " << edges[i].weight << "\n";
                if (edges[i].first == map_path[index].curr_path.back())
                {
                    std::cout << "\tadding new rib to map path:" << edges[i].first
<< " -> " << edges[i].second << " with weight: " << edges[i].weight << "\n";
                    map_path.push_back({ map_path[index].curr_path +
edges[i].second, edges[i].weight + map_path[index].length });//добавляем минимальную
вершину
                    std::cout << "deikstra: comparing the current path length to the
vertex with the new\n\t- current path length for vertex "
<< edges[i].second << " is " <<
result[edges[i].second] << " and new length is " << edges[i].weight +
map_path[index].length << "\n";
                    if (result[edges[i].second] > edges[i].weight +
map_path[index].length)//запись минимального пути до вершины в контейнер
                    {
                        std::cout << "a new path length for vertex "
<< edges[i].second << " is " << edges[i].weight +
map_path[index].length << "\n";
                        result[edges[i].second] = edges[i].weight +
map_path[index].length;
                    }
                }
            }
            path.push_back(map_path[index].curr_path.back());
            map_path.erase(map_path.begin() + index);//удаление предыдущего пути
        }
        print_reuslt();
    }
}

```



```

bool is_view(char value)
{
    for (size_t i = 0; i < path.size(); i++)
        if (path[i] == value)//проверка на то, просмотрена ли уже вершина
            return true;
    return false;
}
size_t min_search()
{
    std::cout << "\nsearch for a new min index:\n";
    double min = 0;
    for (size_t i = 0; i < edges.size(); i++)
        min += edges[i].weight;
    min += 31;//инициализирование начального значения для сравнения
    size_t tmp;
    for (size_t i = 0; i < map_path.size(); i++)
    {
        int tmp = map_path[i].length + func_h(way.finish,
map_path[i].curr_path.back());
        std::cout << "comparing the vertex index with the current minimum:
current is "
                << tmp;
        std::cout << " and current min is " << min << "\n";
        if (tmp < min)//сравнение индекса вершины с текущим минимальным
        {
            if (is_view(map_path[i].curr_path.back()))
            {
                map_path.erase(map_path.begin() + i);//удаление уже просмотренных
вершин
            }
            else
            {
                min = map_path[i].length + func_h(way.finish,
map_path[i].curr_path.back());//запоминание индекса минимальной вершины
                std::cout << "a new current min is " << min << "\n";
                tmp = i;
            }
        }
        std::cout << "a new min index founded: " << tmp << '\n';
        return tmp;
    }
}

void print_result()
{
    std::cout << "Result:\n";
    std::cout << "for vertex " << way.finish << "length is" <<
result[way.finish] << "\n";
    /*std::map<char, int> ::iterator it = result.begin();
    for (int i = 0; it != result.end(); it++, i++)//вывод результата через
итератор
        std::cout << it->first << " - " << it->second << "\n";*/
}

};
int main()
{
    Graph tmp;
    tmp.input_data();
    tmp.deikstra();
}

```