

**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**

Студент гр. 8303

\_\_\_\_\_

Рудько Д.Ю.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2020

## Цель работы.

Изучение алгоритмов поиска пути в графе.

### Задание 1.

Разработайте программу, которая решает задачу построения пути в *ориентированном* графе при помощи **жадного алгоритма**. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

Пример входных данных

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

В первой строке через пробел указываются начальная и конечная вершины. Далее в каждой строке указываются ребра графа и их вес. В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

```
abcde
```

### Задание 2.

Разработайте программу, которая решает задачу построения кратчайшего пути в *ориентированном* графе **методом A\***. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

Пример входных данных

```
a e
a b 3.0
```

b c 1.0  
c d 1.0  
a d 5.0  
d e 1.0

В первой строке через пробел указываются начальная и конечная вершины. Далее в каждой строке указываются ребра графа и их вес. В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

Для приведённых в примере входных данных ответом будет  
ade

### **Индивидуализация.**

#### **Вариант 4**

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

### **Описание жадного алгоритма.**

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

На каждом шаге рассматривается ребро с минимальной длиной, выходящее из текущей вершины. Затем в качестве вершины, из которой требуется найти путь, принимается конец ребра. После чего алгоритм повторяется для новой вершины. Если из вершины нет ребер, происходит откат и рассматривается другой путь.

Данный алгоритм является модификацией алгоритма поиска в глубину, но на каждом шаге перебираются ребра для поиска ребра с минимальным весом, а значит его сложность  $O(N \cdot E + E)$ , где  $N$  – количество вершин, а  $E$  – количество ребер.

Для работы алгоритма хранится граф в виде списка смежности ( $O(N + E)$ ) и вектор посещенных вершин ( $O(N)$ ). В итоге получаем сложность по памяти  $O(2N + E)$ , где  $N$  – число вершин в графе,  $E$  – число ребер в графе.

## 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.

На каждом шаге программы в очередь добавляются новые вершины и достаётся вершина с наибольшим приоритетом. Затем, для снятой вершины, рассматриваются все ребра, которые могут быть добавлены в текущее решение. На основе каждого такого ребра создается новое частичное решение, для него рассчитывается приоритет и новое решение добавляется в очередь с приоритетом.

Сложность алгоритма A\* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

где  $h^*$  — оптимальная эвристика, то есть точная оценка расстояния из вершины  $x$  к цели. Другими словами, ошибка  $h(x)$  не должна расти быстрее, чем логарифм от оптимальной эвристики.

## Описание .

```
map<char, vector<pair<char, double>>> way;
```

Используется для хранения исходного графа.

```
map<char, int> check;
```

Используется для проверки вершин, посещена или нет.

```
priority_queue<pair<double, char>> queue;
```

Очередь с приоритетом. Хранит пары вида путь, вершина.

```
map<char, pair<string, double>> way_to_vertex;
```

Используется для хранения пути до каждой проверенной вершины.

```
string search_greed_way(char start, char finish);
```

Функция поиска пути в графе жадным алгоритмом. Принимает переменные типа char:

start — вершина из которой ищем путь

finish — вершина до которой ищем путь

```
string search_ASTAR(char start, char finish1, char finish2);
```

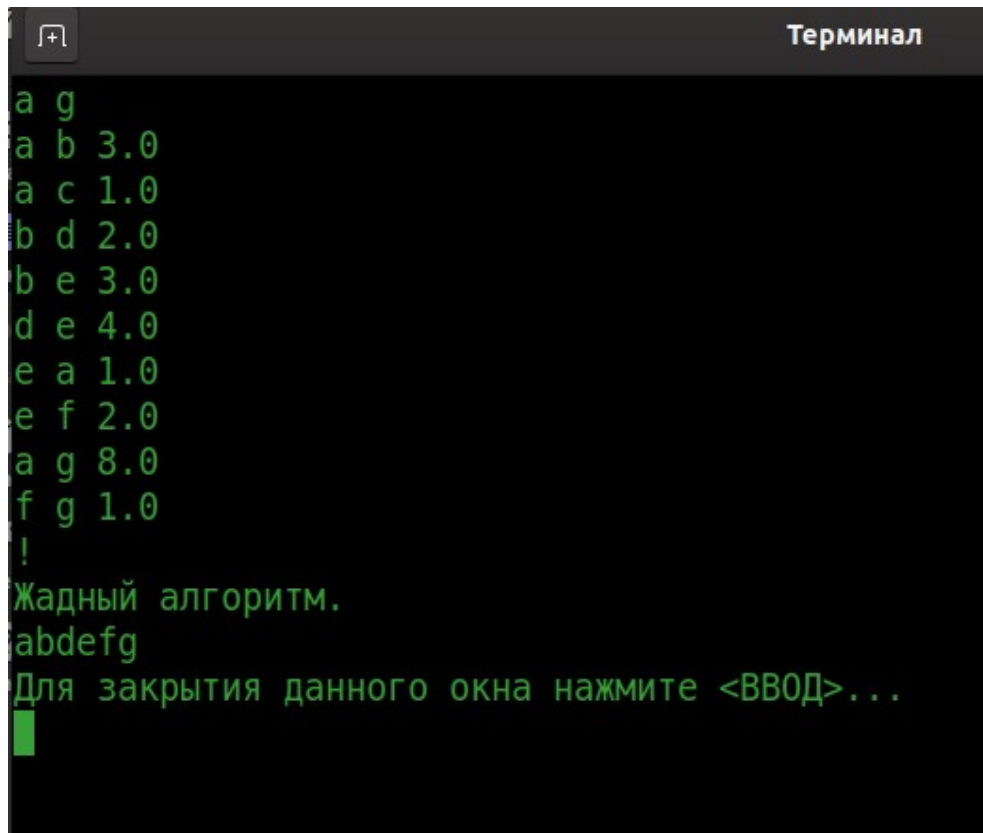
Функция поиска пути в графе алгоритмом A\*. Принимает переменные типа char:

start — вершина из которой ищем путь

finish1 — одна из вершин до которой ищем путь

finish2 — одна из вершин до которой ищем путь

### Тестирование.



```

a g
a b 3.0
a c 1.0
b d 2.0
b e 3.0
d e 4.0
e a 1.0
e f 2.0
a g 8.0
f g 1.0
!
Жадный алгоритм.
abdefg
Для закрытия данного окна нажмите <ВВОД>...
```

```
Терминал
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0
!
Жадный алгоритм.
abcde
Для закрытия данного окна нажмите <ВВОД>...
```

```
Терминал
a f
a c 1.0
a b 1.0
c d 2.0
b e 2.0
d f 3.0
e f 3.0
!
Жадный алгоритм.
acdf
Для закрытия данного окна нажмите <ВВОД>...
```

```

a c h
a b 5
a d 3
b e 8
d f 1
d c 5
e d 2
e f 3
f g 2
h g 9
g c 11
!
Алгоритм ASTAR
Очередь
a(2)
Обрабатываемая вершина: a(2)
Смежные с ней вершины:
b d
Очередь
d(-4) b(-6) d(-7) b(-11)
Обрабатываемая вершина: d(-4)
Смежные с ней вершины:
f c
Очередь
f(-6) b(-6) f(-7) d(-7) c(-8) b(-11) c(-13)
Обрабатываемая вершина: f(-6)
Смежные с ней вершины:
g
Очередь
b(-6) g(-7) f(-7) d(-7) c(-8) g(-10) b(-11) c(-13)
Обрабатываемая вершина: b(-6)
Смежные с ней вершины:
e
Очередь
g(-7) f(-7) d(-7) c(-8) g(-10) b(-11) c(-13) e(-15) e(-16)
Обрабатываемая вершина: g(-7)
Смежные с ней вершины:
c
Очередь
f(-7) d(-7) c(-8) c(-8) g(-10) b(-11) c(-13) c(-13) e(-15) e(-16)
Обрабатываемая вершина: f(-7)
Смежные с ней вершины:
g
Очередь
g(-7) d(-7) c(-8) c(-8) g(-10) g(-10) b(-11) c(-13) c(-13) e(-15) e(-16)
Обрабатываемая вершина: g(-7)
Смежные с ней вершины:
c
Очередь
d(-7) c(-8) c(-8) c(-8) g(-10) g(-10) b(-11) c(-13) c(-13) c(-13) e(-15) e(-16)
Обрабатываемая вершина: d(-7)
Смежные с ней вершины:
f c
Очередь
f(-6) f(-7) c(-8) c(-8) c(-8) c(-8) g(-10) g(-10) b(-11) c(-13) c(-13) c(-13) c(-13) e(-15) e(-16)
Обрабатываемая вершина: f(-6)
Смежные с ней вершины:
g
Очередь
g(-7) f(-7) c(-8) c(-8) c(-8) c(-8) g(-10) g(-10) g(-10) b(-11) c(-13) c(-13) c(-13) c(-13) e(-15) e(-16)
Обрабатываемая вершина: g(-7)
Смежные с ней вершины:
c
Очередь
f(-7) c(-8) c(-8) c(-8) c(-8) c(-8) g(-10) g(-10) g(-10) b(-11) c(-13) c(-13) c(-13) c(-13) c(-13) e(-15) e(-16)
Обрабатываемая вершина: f(-7)
Смежные с ней вершины:
g
Очередь
g(-7) c(-8) c(-8) c(-8) c(-8) c(-8) g(-10) g(-10) g(-10) g(-10) b(-11) c(-13) c(-13) c(-13) c(-13) c(-13) e(-15) e(-16)
Обрабатываемая вершина: g(-7)
Смежные с ней вершины:
c
Очередь
c(-8) c(-8) c(-8) c(-8) c(-8) c(-8) g(-10) g(-10) g(-10) g(-10) b(-11) c(-13) c(-13) c(-13) c(-13) c(-13) c(-13) e(-15) e(-16)
Обрабатываемая вершина: c(-8)
c - конечная вершина
adc
Для закрытия данного окна нажмите <ВВОД>...

```

```
a f b
a b 3
b c 5
c d 1
e f 4
d e 3
f g 0
g h 10
!
Алгоритм ASTAR
Очередь
a(5)
Обрабатываемая вершина: a(5)
Смежные с ней вершины:
b
Очередь
b(-3) b(-7)
Обрабатываемая вершина: b(-3)
b - конечная вершина
ab
Для закрытия данного окна нажмите <ВВОД>...
```

### **Вывод.**

В ходе выполнения лабораторной работы были изучены алгоритмы поиска пути в графе путем написания программ, реализующих жадный алгоритм и A\*.



## ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

### Жадный алгоритм

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

using namespace std;

map<char, vector<pair<char, double>>> way;
map<char, int> check;
string ans = "";

string search_greed_way(char start, char finish){
    char current = start; // текущая вершина - start
    ans += current;
    while(current != finish){ // пока не дошли до finish
        check[current] = 1;
        double min = 1000.0;
        char next;
        int num = -1;
        bool f = false;
        for(auto& i: way[current]){ // выбираем ребро с минимальным весом,
            // которое выходит из текущей вершины
            if(i.second < min && !check[i.first]){
                next = i.first;
                min = i.second;
                num++;
                f = true;
            }
        }
        if(f){
            if(!check[next]){ // если вершина не помечена, идем в нее
                current = next;
                ans += current;
            }
            else {
                current = current;
            }
        }
        else{ // если из вершины нет путей, удаляем ее из ответа
            ans.pop_back();
            current = ans[ans.size() - 1];
        }
    }
    return ans;
}

int main()
{
    char start, finish, from, to;
    double weight;
    cin >> start >> finish;
    while(cin >> from && from != '!'){
        cin >> to;
        cin >> weight;
        way[from].push_back(make_pair(to, weight));
    }
}
```

```

        check[from] = 0;
        check[to] = 0;
    }

    cout << "Жадный алгоритм." << endl;
    cout << search_greed_way(start, finish) << endl;
    return 0;
}

```

## Алгоритм A\*

```

#include <iostream>
#include <vector>
#include <map>
#include <queue>

using namespace std;

map<char, vector<pair<char, double>>> way;

string search_ASTAR(char start, char finish1, char finish2){
    priority_queue<pair<double, char>> queue; //очередь с приоритетом
    map<char, pair<string, double>> way_to_vertex; // путь до вершины
    way_to_vertex[start].first = start; // путь до start есть сама вершина start
    queue.push(make_pair(abs(finish1-start), start)); // добавляем start в
очередь
    while(!queue.empty()){ // пока очередь не пуста
        priority_queue<pair<double, char>> queue_copy(queue);
        cout << "Очередь" << endl;
        while(!queue_copy.empty()){
            cout << queue_copy.top().second << "(" << queue_copy.top().first <<
") ";
            queue_copy.pop();
        }
        cout << endl;
        pair<char, double> current_vertex;
        // берем очередной элемент очереди
        current_vertex.first = queue.top().second;
        current_vertex.second = queue.top().first;
        queue.pop();
        cout << "Обрабатываемая вершина: " << current_vertex.first << "(" <<
current_vertex.second << ")" << endl;
        if(current_vertex.first == finish1 || current_vertex.first == finish2)
            cout << current_vertex.first << " - конечная вершина" << endl;
        else{
            cout << "Смежные с ней вершины:" << endl;
            for(auto& i: way[current_vertex.first])
                cout << i.first << ' ';
            cout << endl;
        }
        if(current_vertex.first == finish1){ // если очередная вершина это
finish1, то выводим ответ
            return way_to_vertex[current_vertex.first].first;
        }
        if(current_vertex.first == finish2){ // если очередная вершина это
finish2, то выводим ответ
            return way_to_vertex[current_vertex.first].first;
        }
    }
}

```

```

        for(auto& i: way[current_vertex.first]){ // проходим по смежным вершинам
current_vertex
            double e1 = abs(finish1 - i.first); //эвристика для очередной
вершины
            double e2 = abs(finish2 - i.first); //эвристика для очередной
вершины
            if(way_to_vertex[i.first].second >
way_to_vertex[current_vertex.first].second + i.second ||
way_to_vertex[i.first].second == 0){
                way_to_vertex[i.first].second =
way_to_vertex[current_vertex.first].second + i.second;
                way_to_vertex[i.first].first =
way_to_vertex[current_vertex.first].first + i.first;
            }
            queue.push(make_pair(-(e1 + way_to_vertex[i.first].second),
i.first));
            queue.push(make_pair(-(e2 + way_to_vertex[i.first].second),
i.first));
        }
    }
    return "No path";
}

int main()
{
    char start, finish1, finish2, from, to;
    double weight;
    cin >> start >> finish1 >> finish2;
    while(cin >> from && from != '!'){
        cin >> to;
        cin >> weight;
        way[from].push_back(make_pair(to, weight));
    }

    cout << "Алгоритм ASTAR" << endl;
    cout << search_ASTAR(start, finish1, finish2) << endl;
    return 0;
}

```