

**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: Жадный алгоритм и A\*.**

Студентка гр. 9382

\_\_\_\_\_

Круглова В.Д.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2021

## Цель работы

Ознакомиться с алгоритмом  $A^*$  и научиться применять его на практике.  
Написать программу реализовывающую поиск пути в графе.

## Постановка задачи

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

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

### **Индивидуальное задание**

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

### **Описание алгоритма**

В первой строке на вход программе подаются начальная и конечная вершины. Далее в каждой строчке указываются рёбра графа и их вес.

Связи графа хранятся в контейнере STL map, в цикле while происходит заполнение point. Цикл прекратит работу в тот момент, когда будет считана пустая строка.

Инициализируется экземпляр класса Graph, после чего вызывается метод Graph::init(). Метод читает информацию о начальной точке и о точке окончания. После происходит считывание зависимостей

графа и сохранение их в контейнер map point. Описание структур приведено ниже.

Далее вызывается метод класса `Graph::greedySearch()`. Метод ищет путь от start до end и возвращает стек результата.

После вызывается метод класса `Graph::aStar()`. Метод ищет путь от start до end методом  $A^*$  и возвращает стек результата.

После вызывается метод класса `Graph::dijkstra()`. Метод ищет путь от start до end методом dijkstra и возвращает стек результата.

### **Сложность алгоритма**

На каждой итерации выполняется поиск не просмотренного минимального ребра, выходящего из текущей вершины. Если такого ребра не нашлось, то происходит откат (текущая вершина принимает своё предыдущее значение и удаляется из результата). Если путь найден, то за вершину, из которой требуется найти путь, принимается конец ребра, предыдущая добавляется в результат. Продолжаем, пока текущая вершина не станет конечной. Алгоритм является модификацией алгоритма поиска в глубину. Его сложность -  $O(N + M \cdot \log N)$ , где  $N$  – количество вершин, а  $M$  – кол-во ребер. Поиск не просмотренного минимального значения занимает  $\log(N)$ , так как значения хранятся в ассоциативном контейнере.

Так как на каждом шаге хранится массив смежных ребер, а их количество не превышает число вершин в графе, то получаем сложность по памяти  $O(N^2)$ , где  $N$  - число вершин в графе,  $M$  – число ребер в графе.

Память. При более точной эвристической функции сложность по памяти может составлять  $O(|E| + |V|)$ , т. к. есть возможность сразу построить правильный путь без возвращения к предыдущим вершинам.

В ином случае, каждый шаг будет неверным и придётся просматривать каждое ребро графа. Тогда в худшем случае сложность

может оказаться экспоненциальной.

Дейкстра. Сложность по времени  $O(|E| \cdot \log |N|)$ , где  $E$  – кол-во ребер,  $N$  – вершин. Находясь в каждой вершине графа, пересчитывается или остается прежним длина пути до ее соседей.

### **Описание структур**

struct Triple – хранит информацию о ребре. Имеет три поля: first, second, third – имя вершины, вес ребра и флаг (проходили по ней или нет) соответственно.

Class Graph – хранит стартовую точку, точку окончания и информацию о связях.

map<int, set<Triple, SetCompare>> point – контейнер, хранящий связи вершин.

std::stack<int> res – стек, хранящий результат. Заполняется и возвращается функцией greedySearch

### **Описание основных функций**

void expand\_stack(std::stack<int>& res) – принимает стек и «переворачивает» его.

std::stack<int> Graph::greedySearch() – функция поиска пути в графе. Работает по принципу поиска в глубину. Идем по графу пока не достигнем конца (по условию), либо пока не окажемся в тупике. Если дальше пути нет (за этим

следит флаг can\_go), откатываемся на вершину назад. В итоге получаем либо стек с результатом, либо пустой стек, что означает, что требуемого пути нет.

void Graph::print\_graph() – печатает список зависимостей point

void Graph::init() - Метод читает информацию о начальной точке и о точке окончания. После происходит считывание зависимостей графа и сохранение их в контейнер map point. Описание структур приведено ниже.

`std::stack<int> Graph::aStar()` – функция поиска пути в графе. А\* пошагово просматривает пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь.

`const char find_min_vertex(list<char> open, map<char,float> G)` – вспомогательная функция для метода Дейкстры. Принимает лист открытых вершин контейнер мап длины путей до них и возвращает вершину, до которой короче путь и которая находится в листе `open`.

`std::stack<char> Graph::dijkstra()` – функция поиска пути в графе. Дейкстра пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Для каждой вершины считается расстояние до начала и формируется карта пути.

### Пример работы программы

Входные данные	Выходные данные
a e	***Info***
a b 3.0	Initialization
b c 1.0	Open list: b d c e
c d 1.0	Close list: a
a d 5.0	Map:
d e 1.0	to b from a
	to d from a
	Initialization complete
	Current vertex - b
	Open list: d c e
	Close list: a b

	<p>The path for the vertex c is recalculated. Old value = <math>1e+10</math>. New value = 4.</p> <p>Map:</p> <p>to b from a</p> <p>to c from b</p> <p>to d from a</p> <p>Current vertex - c</p> <p>Open list: d e</p> <p>Close list: a b c</p> <p>Map:</p> <p>to b from a</p> <p>to c from b</p> <p>to d from a</p> <p>Current vertex - d</p> <p>Open list: e</p> <p>Close list: a b c d</p> <p>The path for the vertex e is recalculated. Old value = <math>1e+10</math>. New value = 6.</p> <p>Map:</p> <p>to b from a</p> <p>to c from b</p> <p>to d from a</p> <p>to e from d</p> <p>Current vertex - e</p> <p>Open list:</p> <p>Close list: a b c d e</p> <p>Map:</p> <p>to b from a</p> <p>to c from b</p> <p>to d from a</p> <p>to e from d</p> <p>The algorithm has finished its work.</p> <p>Reconstruction path..</p> <p>Dijkstra answer: ade</p>
--	---

## Тестиирование

Таблица 1. Тестирование алгоритма Дейкстры.

№	Входные данные	Выходные данные -d	Выходные данные -g	Выходные данные-ass
1	p t p e 1.0 p r 1.0 p t 12.0 e t 2.0	pet	pet	pt
2	i u i l 1.0 i o 2.0 i v 3.0 i e 4.0 i u 100.0 l u 1.0	ilu	ilu	ilu
3	a b a b 10.0 a c 1.0 c b 1.0	acb	acb	acb
4	a h a b 1.0 a c 2.0 b d 5.0 b g 10.0 b e 4.0 c e 2.0 c f 1.0 d g 2.0 e d 1.0 e g 7.0 f e 3.0 f h 8.0 g h 1.0	acedgh	abedgh	abedgh
5	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0 c m 1.0 m n 1.0	ag	abdefg	ag
6	a b a b 1.12	ab	ab	ab



7	a e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	age	abge	age
---	--	-----	------	-----

### **Вывод**

В ходе выполнения лабораторной работы были изучены и применены на практике жадный алгоритм и алгоритм A\*. Также был реализован алгоритм Дейкстры.

## ПРИЛОЖЕНИЕ С КОДОМ

Название файла: main.cpp

```
#include <iostream>
#include <stack>
#include <string>
#include <map>
#include <set>
#include <list>
#include <vector>
#include <cstring>

using std::map;
using std::set;
using std::pair;
using std::string;
using std::cout;
using std::cin;
using std::endl;
using std::list;

#define INF 10000000000.0

typedef struct Triple
{
    // структура, хранящая информацию о ребре и было ли оно пройдено
    char name;
    double weight;
    mutable bool flag;
    Triple() {}
    Triple(char _name, double _weight, bool _flag=false) : name(_name),
weight(_weight), flag(_flag) {}
} Triple;

struct SetCompare
{
    bool operator()(Triple v1, Triple v2)
    {
        if (v1.weight == v2.weight)
            return v1.name < v2.name;
        return v1.weight < v2.weight;
    }
};

class Graph
{
public:
    // point хранит зависимость вершин в виде: вершина – массив смежных вершин
    // Массив смежных вершин отсортирован по возрастанию веса ребра (SetCompare)
    map<char, set<Triple, SetCompare>> point;
    char start, end;
public:
    void init();
    void print_graph();
    std::stack<char> greedySearch();
    std::stack<char> aStar();
    std::stack<char> dijkstra();
    //вспомогающие
    int heuristic(char);
    std::stack<char> reconstruction(map<char, char>);
};
```

```

void Graph::init()
/* Читаем start, end. После заполняем массив зависимостей */
{
    string input;
    //cout << "Enter start and end point: ";
    getline(cin, input);
    start = input[0];
    end = input[2];

    //cout << "Enter adjacency list:" << endl;
    while (getline(cin, input))
    {
        if (input.empty()) break;
        point[input[0]].emplace(input[2], std::stoi(input.substr(4)));
    }
}

void Graph::print_graph()
{
    for (auto var : point)
    {
        cout << var.first << ": ";
        for (auto var2 : var.second)
            cout << var2.name << " " << var2.weight << " " << var2.flag << "; ";
        cout << std::endl;
    }
}

std::stack<char> Graph::greedySearch()
{
    // В стеке храним результат. Сразу записываем первую вершину
    // curr хранит массив смежных вершин к текущей вершине
    std::stack<char> res;
    res.push(start);

    set<Triple, SetCompare> curr = point[res.top()];

    while (!res.empty() && res.top() != end)
    {
        bool can_go = false;
        char tmp;
        if (!curr.empty())
        {
            for (auto &var : point[res.top()]) //point[res.top()] == curr.
                if (!var.flag)
                {
                    can_go = true;
                    var.flag = true;
                    tmp = var.name;
                    break;
                }
        }

        if (can_go)
        {
            res.push(tmp);
            curr = point[tmp];
        } else {
            res.pop();
            if (!res.empty()) curr = point[res.top()];
        }
    }
}

```

Сделано для того, чтобы флаг изменялся

```

        //зануляем флаг, чтобы не портить массив
        for (auto &var: point)
            for (auto &var2: var.second)
                var2.flag = false;

    return res;
}

void expand_stack(std::stack<char>& res)
{
    std::stack<char> tmp;
    tmp.swap(res);
    while (!tmp.empty())
    {
        res.push(tmp.top());
        tmp.pop();
    }
}

void print_stack(std::stack<char> res)
{
    while (!res.empty())
    {
        cout << res.top();
        res.pop();
    }
    cout << endl;
}

int Graph::heuristic(char curr)
{
    return abs(curr - end);
}

char minF(list<char> open, map<char, float> F){//поиск минимального значения
f(x)
    int res = open.back();
    float min = F[res];

    for (auto var : open)
    {
        if (F[var] <= min){
            res = var;
            min = F[var];
        }
    }
    return res;
}

bool inList(list<char> _list, char x)
{
    for (auto var : _list)
        if (var == x) return true;
    return false;
}

std::stack<char> Graph::reconstruction(map<char, char> from)
{
    std::stack<char> res;
    char curr = end;
    while (curr != start)
    {
        res.push(curr);
        curr = from[curr];
    }
    res.push(start);
    return res;
}

```

```

}

std::stack<char> Graph::aStar()
{
    std::stack<char> res; //стек результата
    list<char> close; //список пройденных вершин
    list<char> open = {start}; //список рассматриваемых вершин
    map<char, char> from; //карта пути
    map<char, float> G; //хранит стоимости путей от начальной вершины
    map<char, float> F; //оценки f(x) для каждой вершины
    G[start] = 0;
    F[start] = G[start] + heuristic(start);

    while (!open.empty())
    {
        char curr = minF(open, F);

        /* Вывод для ясности */
        cout << "***Info***" << endl;
        cout << "Current: " << curr << endl;
        cout << "Close list: ";
        for (auto var : close)
            cout << var << " ";
        cout << endl << "Open list: ";
        for (auto var : open)
            cout << var << " ";
        cout << endl << "***Info end***" << endl;

        if (curr == end)
        {
            res = reconstruction(from); //восстанавливаем
            return res;
        }

        open.remove(curr);
        close.push_back(curr);

        for (auto neighbor : point[curr])
        {
            if (inList(close, neighbor.name)) //если уже проходили, дальше
                continue;

            float tmpG = G[curr] + neighbor.weight; //вычисление g(x) для
            //обрабатываемого соседа

            if (!inList(open, neighbor.name) || tmpG < G[neighbor.name])
            {
                from[neighbor.name] = curr;
                G[neighbor.name] = tmpG;
                F[neighbor.name] = G[neighbor.name] + heuristic(neighbor.name);
            }

            if (!inList(open, neighbor.name))
                open.push_back(neighbor.name);
        }
    }

    return res;
}

const char find_min_vertex(list<char> open, map<char, float> G)
{
    double min = INF;
    char ret;
    for (auto var : open)
    {

```

```

        if (G[var] < min)
        {
            min = G[var];
            ret = var;
        }
    }

    return ret;
}

std::stack<char> Graph::dijkstra()
{
    std::stack<char> res; //стек результата
    list<char> close = {start}; //список пройденных вершин
    list<char> open; //список рассматриваемых вершин
    map<char, char> from; //карта пути
    map<char, float> G; //хранит стоимости путей от начальной вершины
    G[start] = 0;
    for (auto var : point) {
        for (auto var2 : var.second)
        {
            open.push_back(var2.name);
            G[var2.name] = INF;
        }
    }
    open.remove(start);

    for (auto var : point[start])
    {
        G[var.name] = var.weight;
        from[var.name] = start;
    }

    /*for (int i=0; i<open.size(); i++)*/
    while (!open.empty())
    {
        char curr = find_min_vertex(open, G);

        close.push_back(curr);
        open.remove(curr);

        for (auto var : point[curr])
        {
            if (G[curr]+var.weight < G[var.name])
            {
                G[var.name] = G[curr]+var.weight;
                from[var.name] = curr;
            }
        }
    }

    return reconstruction(from);
}

int main(int argc, char** argv)
{
    Graph one;
    one.init();
    std::stack<char> res;

    if (argc == 2)
    {
        if (!strcmp(argv[1], "-greed\0") || !strcmp(argv[1], "-g\0"))
        {
            res = one.greedySearch();
            expand_stack(res);
        }
    }
}

```

```

        cout << "GreedySearch answer: ";
        print_stack(res);
    }
    if (!strcmp(argv[1], "-astar\0") || !strcmp(argv[1], "-as\0"))
    {
        res = one.aStar();
        cout << "aStarSearch answer: ";
        print_stack(res);
    }
    if (!strcmp(argv[1], "-dijkstra\0") || !strcmp(argv[1], "-d\0"))
    {
        res = one.dijkstra();
        cout << "Dijkstra answer: ";
        print_stack(res);
    }
}

return 0;
}

```