

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ
по лабораторной работе №2
по дисциплине «Построение и анализ алгоритмов»
Тема: Жадный алгоритм и A^*

Студент гр. 0382

Корсунов А.А.

Преподаватель

Шевская Н.В.

Санкт-Петербург

2022

Цель работы.

Применить на практике знания о жадном алгоритме и алгоритме A^* на графе. Реализовать программы, которые считывают граф и находят в нем минимальный по расстоянию путь от стартовой вершины до конечной с помощью обоих алгоритмов.

Основные теоретические положения.

Жадный алгоритм — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Алгоритм поиска “**A-звездочка**” относится к эвристическим алгоритмам поиска по первому лучшему совпадению на графе с положительными (>0) весами рёбер, который находит маршрут с наименьшей стоимостью от одной вершины в другой.

В отличие от алгоритма Дейкстры, использует эвристическую функцию.

Идея алгоритма: A^* пошагово просматривает все пути, ведущие от начальной вершины к конечной, пока не найдет минимальный путь. Как и все эвристические алгоритмы поиска, алгоритм сначала просматривает те маршруты и те ребра, которые кажутся ведущими к цели. От жадного алгоритма его отличает то, что при выборе вершины он учитывает весь путь до неё.

В начале работы просматриваются узлы, смежные с начальным. Выбирается тот, который имеет минимальное значение $f(x)$, после чего узел раскрывается. В начале работы алгоритм оперирует с множеством нераскрытых вершин. Затем $f(x)=h(x)+g(x)$ - к эвристической функции прибавляется путь до текущей вершины.

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

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

Вариант 1 — В A^* вершины именуются целыми числами (в т.ч. отрицательными).

Ход работы.

1. Был произведен анализ задания.
2. Был реализован жадный алгоритм поиска минимального пути в графе:

1)Алгоритм:

Данные, поддающиеся на вход заносятся в вектор-вектор ребер, элементы которого являются объектами с тремя полями (вершина-предок, вершина-потомок, вес ребра между ними). Затем начинается сам жадный алгоритм: на каждом рассматривается вершина (она заносится в строку, в которой последовательно хранятся вершины минимального пути), в которую был произведен переход на прошлом шаге (на первом шаге такая вершина — начальная вершина). Для этой вершины ищется потомок с минимальным весом ребра из всех ее потомков (перебирается вектор всех ребер). Если у текущей вершины нет потомков, то текущей вершиной опять становится ее потомок, из строки вершин удаляется эта вершина. Если у текущей вершины нашелся потомок (если нашелся, то он минимальный из всех), потомок становится текущей вершиной, а из вектора ребер удаляются все ребра, ведущие в новую текущую вершину. Алгоритм заканчивается, когда текущей вершиной становится конечная вершина (в этом случае искомый путь хранится в строке минимального пути) или когда путей из начальной вершины больше нет — когда начальная вершина — текущая, а вектор-ребер пуст.

2) Структура данных, которая представляет граф — vector, элементами которого являются объекты класса Edge (класс ребер).

3) Сложность алгоритма по времени — $O(|V|+|E|)$, т. к. нужно рассмотреть все ребра и найти ребро минимального веса;

Сложность алгоритма по памяти — $O(N)$, N – количество ребер, поданное на вход (вершина-предок, вершина-потомок, вес ребра между ними);

4) Функции и структуры данных:

-class Edge:

*char left_vertex – вершина-предок;

*char right_vertex – вершина-предок;

*char edge_amount – вес ребра между ними;

*Edge(char left_vertex, char right_vertex, float edge_amount) – конструктор

для класса;

*...set...(type) – сеттеры для полей;

*...get...(type) – геттеры для полей;

-class grid_alg:

*char local_start_vertex = '\0' - начальная текущая вершина

*char finish_vertex = '\0'; - конечная вершина

*std::vector<Edge> graph; - вектор всех поданных на вход ребер

*Edge local_min_edge; - текущий минимальный вес ребра

*void add_edge(char, char, float); - для начального заполнения графа

*void set_start_finish_vertexes(char, char); - для начального заполнения полей
начальной и конечной вершин

*void alg(); - основная часть "жадного" алгоритма

5) Тестирование:

а) Входные данные:

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

Выходные данные:

abcde

б) Входные данные:

a z

a w 17

w x 1

x y 18

y z 1

b w 1

a b 1

Выходные данные:

abwxyz

в) Входные данные:

a j

a b 1.0

b c 1.0

c d 1.0

d e 1.0

e j 1.0

a f 1.0

f g 1.0

g h 1.0

h i 1.0

i j 1.0

Выходные данные:

abcdej

```

a j
a b 1.0
b c 1.0
c d 1.0
d e 1.0
e j 1.0
a f 1.0
f g 1.0
g h 1.0
h i 1.0
i j 1.0
qqqqqqqqqqqq
abcdej

```

Рисунок 1 - Пример работы программы Жадного алгоритма

3. Был реализован алгоритм A*поиска минимального пути в графе:

1) Алгоритм:

Входные данные хранятся в векторе — вектор-ребер, элементами которого являются объекты класса с тремя полями (вершина-предок, вершина-потомок, вес ребра между ними). Затем производится сам алгоритм A*: в заранее созданный вектор «открытых» (далее - open) вершин заносится начальная вершина (вектор открытых вершин — вектор, элементы которого являются объекты класса с пятью полями (имя текущей вершины, путь до нее, путь от нее до конечной вершины, имя предка, весь путь), сам имя такого класса — Vertex). На каждом шаге выбирается текущая вершина, выбор происходит из всех вершин, находящихся в open вершин на основе суммы двух полей всех элементов (путь до вершины и путь до конечной вершины, путь до конечной вершины считается как модуль разницы кодов имен вершин). После чего из open удаляется текущая вершина, а в вектор «закрытых» (далее - closed) вершин заносится текущая вершина (вектор «закрытых» вершин — вектор, элементами которого являются объекты класса Vertex). Затем для текущей вершины рассматриваются все ее потомки, если потомок уже есть в close, то сравнивается его путь до его вершины с путем до текущей вершины + вес ребра между ними, если второй путь меньше — путь у потомка меняется, после чего идет переход на следующий шаг цикла, если же потомка еще нет в close, то

проверяется, есть ли он в open (если нет — заносится в open), если есть — происходит та же проверка, что была в close. Алгоритм заканчивается, когда текущей вершиной становится конечная вершина (тогда в close хранится все пройденные вершины, с помощью которых восстанавливается путь до от конечной вершины до начальной), или когда open оказывается пустым (тогда путей нет).

2) Структура данных, которая представляет граф — vector, элементами которого являются объекты класса Edge (класс ребер).

3) Сложность алгоритма по времени — зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растет 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 к конечной вершины.

Сложность алгоритма по памяти — $O(N)$, N – количество ребер, поданное на вход (вершина-предок, вершина-потомок, вес ребра между ними);

4) Функции и структуры данных:

-класс Vertex:

*int name = 10000; - имя вершины

*float way_to_name = 10000; - путь до нее

* float way_to_end = 10000; - путь от нее до конечной вершины

*int parent = 10000; - имя предка

*float way = 10000; - весь путь

*Vertex(int name, float way_to_name, float way_to_end, int parent) : name(name), way_to_name(way_to_name), way_to_end(way_to_end), parent(parent) — конструктор;

-класс Edge

*int left_vertex = 10000; - вершина-предок

*int right_vertex = 10000; - вершина-потомок

*float edge_amount = 0; - вес ребра

*Edge(int left_vertex, int right_vertex, float edge_amount) : left_vertex(left_vertex),
right_vertex(right_vertex), edge_amount(edge_amount) — конструктор

-класс ASTAR_alg

*int start_vertex = 10000; - начальная вершина

*int finish_vertex = 10000; - конечная вершина

*std::vector<Edge> graph; - вектор всех поданных на вход ребер

*std::vector<Vertex> vertexes; - вектор "закрытых" вершин

*std::vector<Vertex> open; - вектор "открытых вершин"

*void add_edge(int, int, float); - для начального заполнения графа

*void set_start_finish_vertexes(int, int); - для начального заполнения полей
начальной и конечной вершин

void alg(); - основная часть "А" алгоритма

*int hueristics(int, int); - эвристическая функция

*Vertex min_open(); - текущая минимальная вершина с из open

*void delete_vertex(int); - удаление открытой вершины

*bool check_vertexx(int, std::vector<Vertex>&); - проверка вершины, лежит ли
она в переданном векторе

*int find_vertexx(int, std::vector<Vertex>&); - найти индекс вершины в
переданном векторе

*void print_recover(); - восстановить путь и вывести его

5) Тестирование:

а) Входные данные:

1 6

1 3 17

3 4 1

4 5 18

5 6 1

2 3 1

1 2 1

Выходные данные:

1 2 3 4 5 6

б) Входные данные:

-2 9

-2 -1 1

-2 3 3

-1 0 5

-1 4 3

3 4 4

0 1 6

1 10 1

4 2 4

2 5 1

2 11 1

11 10 2

4 6 5

6 7 6

6 8 1

7 9 5

10 7 3

Выходные данные:

-2 -1 4 2 11 10 7 9

в) Входные данные:

1 5

1 2 5

2 4 5

Выходные данные:

no way

г) Входные данные:

-5 4

-5 -4 1

-4 -3 1

-3 -2 1

-2 -1 1

-1 4 1

-5 0 1

0 1 1

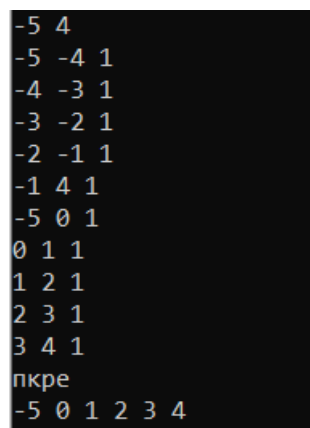
1 2 1

2 3 1

3 4 1

Выходные данные:

-5 0 1 2 3 4



```
-5 4
-5 -4 1
-4 -3 1
-3 -2 1
-2 -1 1
-1 4 1
-5 0 1
0 1 1
1 2 1
2 3 1
3 4 1
пре
-5 0 1 2 3 4
```

Рисунок 2 - Пример работы программы алгоритма A*

Выводы.

Были применины на практике знания о жадном алгоритме и алгоритме A^* на графе. Реализованы программы, которые считывают граф и находят в нем минимальный по расстоянию путь от стартовой вершины до конечной с помощью обоих алгоритмов.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл grid.cpp:

```
#include <iostream>
```

```
#include <vector>
```

```
class Edge //класс ребер
```

```
{
```

```
private:
```

```
    char left_vertex = '\0'; //первая вершина ребра
```

```
    char right_vertex = '\0'; //вторая вершина ребра
```

```
    float edge_amount = 0; //вес ребра
```

```
public:
```

```
    Edge() = default;
```

```
    Edge(char left_vertex, char right_vertex, float edge_amount) :
```

```
left_vertex(left_vertex), right_vertex(right_vertex), edge_amount(edge_amount) {}
```

```
    ~Edge() = default;
```

```
    void set_left_vertex(char left_vertex)
```

```
{
```

```
        this->left_vertex = left_vertex;
```

```
}
```

```
    void set_right_vertex(char right_vertex)
```

```
{
```

```
        this->right_vertex = right_vertex;
```

```
}
```

```
    void set_edge_amount(float edge_amount)
```

```
{
```

```
        this->edge_amount = edge_amount;
```

```
}
```

```

char get_left_vertex()
{
    return left_vertex;
}

char get_right_vertex()
{
    return right_vertex;
}

float get_edge_amount()
{
    return edge_amount;
}
};

class grid_alg
{
private:
    char local_start_vertex = '\0'; //начальная текущая вершина
    char finish_vertex = '\0'; //конечная вершина
    std::vector<Edge> graph; //вектор всех поданных на вход ребер
    Edge local_min_edge; //текущий минимальный вес ребра
public:
    void add_edge(char, char, float); //для начального заполнения графа
    void set_start_finish_vertexes(char, char); //для начального заполнения полей
    начальной и конечной вершин
    void alg(); //основная часть "жадного" алгоритма
};

void grid_alg::alg()
{

```

```

if (graph.empty()) //если граф пустой - путей нет
{
    std::cout << "no way\n";
    return;
}

std::string vertex_string; //строка для вывода пути

while (true)
{
    local_min_edge.set_edge_amount(21000); //для нахождения минимального
текущего веса ребра
    vertex_string += local_start_vertex; //запись в строку текущую
рассматриваемую вершину (из которой выходят ребра)

    if (local_start_vertex == finish_vertex) //цикл заканчивается, когда текущая
рассматриваемая вершина - конечная вершина
    {
        break;
    }

    if (vertex_string.size() == 1 && graph.size() == 0) //если из начальной
вершины нет ни одного пути в конечную вершину
    {
        std::cout << "no way\n";
        return;
    }

    for (int i = 0; i < graph.size(); i++) //поиск минимального ребра для текущей
рассматриваемой вершины

```



```

{
    if (local_start_vertex == graph[i].get_left_vertex() &&
graph[i].get_edge_amount() < local_min_edge.get_edge_amount())
    {
        local_min_edge.set_right_vertex(graph[i].get_right_vertex());
//фиксируется та вершина, в которое минимальное ребро ведет
        local_min_edge.set_edge_amount(graph[i].get_edge_amount());
//фиксируется вес минимального ребра
    }
}

if (local_min_edge.get_edge_amount() == 21000) //в случае, если у текущей
рассматриваемой вершины нет пути в другие еще не рассмотренные вершины
{
    local_start_vertex = vertex_string[vertex_string.size() - 2]; //"откат" назад
- рассматривается ранее рассмотренная вершины, но без текущего ребра
    vertex_string.erase(vertex_string.size() - 2); //удаление этой вершины из
списка
    continue;
}

local_start_vertex = local_min_edge.get_right_vertex(); //в случае, если
минимальная нашлась: переход к вершине минимального ребра

for (int i = 0; i < graph.size(); i++)
{
    if (graph[i].get_right_vertex() == local_start_vertex) //удаляются все
ребра, которые введут в рассматриваемую вершину
    {

```

```

        graph.erase(graph.begin() + i);
        i--;
    }
}

std::cout << vertex_string;
}

void grid_alg::add_edge(char left_vertex, char right_vertex, float edge_amount)
{
    graph.push_back(Edge(left_vertex, right_vertex, edge_amount));
}

void grid_alg::set_start_finish_vertexes(char start_vertex, char finish_vertex)
{
    this->local_start_vertex = start_vertex;
    this->finish_vertex = finish_vertex;
}

int main()
{
    grid_alg graph;
    char start_vertex; //начальная вершина
    char finish_vertex; //конечная вершина
    float edge_amount; //вес ребра

    std::cin >> start_vertex >> finish_vertex;
    graph.set_start_finish_vertexes(start_vertex, finish_vertex); //установка полей
    начальной и конечной вершин

```

```

while (std::cin >> start_vertex >> finish_vertex >> edge_amount)
{
    graph.add_edge(start_vertex, finish_vertex, edge_amount); //заполнение
    вектора, элементы которого являются ребра (вершины + вес ребра), что
    отражает начальный граф
}

graph.alg();

return 0;
}

```

файл ASTAR.cpp:

```

#include <iostream>
#include <vector>

```

```

class Vertex
{
private:
    int name = 10000;
    float way_to_name = 10000;
    float way_to_end = 10000;
    int parent = 10000;
    float way = 10000;
public:
    Vertex() = default;
    Vertex(int name, float way_to_name, float way_to_end, int parent) : name(name),
    way_to_name(way_to_name), way_to_end(way_to_end), parent(parent)
    {

```

```

        way = way_to_name + way_to_end;
    }
    ~Vertex() = default;

    friend class ASTAR_alg;
};

class Edge //класс ребер
{
private:
    int left_vertex = 10000; //первая вершина ребра
    int right_vertex = 10000; //вторая вершина ребра
    float edge_amount = 0; //вес ребра

public:
    Edge() = default;
    Edge(int left_vertex, int right_vertex, float edge_amount) : left_vertex(left_vertex),
right_vertex(right_vertex), edge_amount(edge_amount) {}
    ~Edge() = default;
    friend class ASTAR_alg;
};

class ASTAR_alg
{
private:
    int start_vertex = 10000;
    int finish_vertex = 10000; //конечная вершина
    std::vector<Edge> graph; //вектор всех поданных на вход ребер
    std::vector<Vertex> vertexes; //вектор "закрытых" вершин
    std::vector<Vertex> open; //вектор "открытых вершин"

```

public:

```
void add_edge(int, int, float); //для начального заполнения графа  
void set_start_finish_vertexes(int, int); //для начального заполнения полей  
начальной и конечной вершин  
void alg(); //основная часть "A*" алгоритма  
int hueristics(int, int); // эвристический алгоритм  
Vertex min_open(); //текущая минимальная вершина с F  
void delete_vertex(int); //удаление открытой вершины  
bool check_vertexx(int, std::vector<Vertex>&);  
int find_vertexx(int, std::vector<Vertex>&);  
void print_recover(); //восстановить путь и вывести его  
};
```

```
void ASTAR_alg::print_recover()  
{  
if (vertexes[vertexes.size() - 1].name != finish_vertex)  
{  
std::cout << "no way\n";  
return;  
}  
std::vector<int> result;  
Vertex current_vertex = vertexes[vertexes.size() - 1];  
while (true)  
{  
result.push_back(current_vertex.name);  
if (current_vertex.parent == 10000)  
{  
break;  
}  
}
```

```

        current_vertex = vertexes[find_vertexx(current_vertex.parent, vertexes)];
    }
    for (int i = result.size() - 1; i >= 0; i--)
    {
        std::cout << result[i] << " ";
    }
}

```

```

int ASTAR_alg::find_vertexx(int name, std::vector<Vertex>& vertexx)
{
    int key = -1;
    for (int i = 0; i < vertexx.size(); i++)
    {
        if (vertexx[i].name == name)
        {
            key = i;
        }
    }
    return key;
}

```

```

bool ASTAR_alg::check_vertexx(int name, std::vector<Vertex>& vertexx)
{
    bool key = false;
    for (int i = 0; i < vertexx.size(); i++)
    {
        if (vertexx[i].name == name)
        {
            key = true;
        }
    }
}

```

```

    }
    return key;
}

```

```

void ASTAR_alg::delete_vertex(int name)
{
    for (int i = 0; i < open.size(); i++)
    {
        if (open[i].name == name)
        {
            open.erase(open.begin() + i);
            i--;
        }
    }
}

```

```

Vertex ASTAR_alg::min_open()
{
    float min = open[0].way;
    Vertex min_F = Vertex(open[0].name, open[0].way_to_name,
open[0].way_to_end, open[0].parent);
    for (int i = 1; i < open.size(); i++)
    {
        if (open[i].way < min || open[i].way == min && open[i].name > min_F.name)
        {
            min = open[i].way;
            min_F = Vertex(open[i].name, open[i].way_to_name, open[i].way_to_end,
open[i].parent);
        }
    }
}

```

```

    }
    return min_F;
}

int ASTAR_alg::hueristics(int name, int end)
{
    return abs(name - end);
}

void ASTAR_alg::alg()
{
    if (graph.empty()) //если граф пустой - путей нет
    {
        std::cout << "no way\n";
        return;
    }

    open.push_back(Vertex(start_vertex, 0, hueristics(start_vertex, finish_vertex),
10000));
    while (!open.empty())
    {
        bool tentative_is_better;
        Vertex current_vertex = min_open();
        if (open.empty()) //если граф пустой - путей нет
        {
            std::cout << "no way\n";
            return;
        }
        if (current_vertex.name == finish_vertex)

```



```

{
    vertexes.push_back(current_vertex);
    break;
}

if (open.empty()) //если граф пустой - путей нет
{
    std::cout << "no way\n";
    return;
}

delete_vertex(current_vertex.name);
vertexes.push_back(current_vertex);

for (int i = 0; i < graph.size(); i++)
{

    if (graph[i].left_vertex == current_vertex.name)
    {
        float temp_way_to_name_child = current_vertex.way_to_name +
graph[i].edge_amount;
        if (check_vertexx(graph[i].right_vertex, vertexes))
        {
            if (temp_way_to_name_child <
vertexes[find_vertexx(graph[i].right_vertex, vertexes)].way)
            {
                vertexes[find_vertexx(graph[i].right_vertex, vertexes)].parent =
graph[i].left_vertex;
                vertexes[find_vertexx(graph[i].right_vertex, vertexes)].way_to_name
= temp_way_to_name_child;
                vertexes[find_vertexx(graph[i].right_vertex, vertexes)].way_to_end =
hueristics(graph[i].right_vertex, finish_vertex);

```

```

    }
    continue;
}
if (!check_vertexx(graph[i].right_vertex, open))
{
    open.push_back(Vertex(graph[i].right_vertex,
temp_way_to_name_child, hueristics((int)graph[i].right_vertex, (int)finish_vertex),
graph[i].left_vertex));
    tentative_is_better = true;
}
else
{
    if (temp_way_to_name_child < open[find_vertexx(graph[i].right_vertex,
open)].way)
    {
        tentative_is_better = true;
    }
    else
    {
        tentative_is_better = false;
    }
    if (tentative_is_better)
    {
        open[find_vertexx(graph[i].right_vertex, open)].parent =
graph[i].left_vertex;
        open[find_vertexx(graph[i].right_vertex, open)].way_to_name =
temp_way_to_name_child;
        open[find_vertexx(graph[i].right_vertex, open)].way_to_end =
hueristics(graph[i].right_vertex, finish_vertex);
    }
}

```

```

        }
    }
}
}
print_recover();
}

```

```

void ASTAR_alg::add_edge(int left_vertex, int right_vertex, float edge_amount)
{
    graph.push_back(Edge(left_vertex, right_vertex, edge_amount));
}

```

```

void ASTAR_alg::set_start_finish_vertexes(int start_vertex, int finish_vertex)
{
    this->start_vertex = start_vertex;
    this->finish_vertex = finish_vertex;
}

```

```

int main()
{
    ASTAR_alg graph;
    int start_vertex; //начальная вершина
    int finish_vertex; //конечная вершина
    float edge_amount; //вес ребра

    std::cin >> start_vertex >> finish_vertex;
    graph.set_start_finish_vertexes(start_vertex, finish_vertex);

    while (std::cin >> start_vertex >> finish_vertex >> edge_amount)
    {

```

```

    if (edge_amount < 0)
    {
        std::cout << "no way\n";
        return 0;
    }

    graph.add_edge(start_vertex, finish_vertex, edge_amount); //заполнение
    вектора, элементы которого являются ребра (вершины + вес ребра), что
    отражает начальный граф
}

graph.alg();

return 0;
}

```