

**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. 9383

\_\_\_\_\_

Чебесова И. Д.

Преподаватель

\_\_\_\_\_

Фирсов М. А.

Санкт-Петербург

2021

## **Цель работы.**

Познакомиться с жадным алгоритмом и алгоритмом A\*, реализовать оба алгоритма на одном из языков программирования.

**Вариант -.** В данной работе не присутствует индивидуальный вариант.

## **Задание.**

### **1. 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

## **2. 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

## **Основные теоретические положения.**

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

Алгоритм  $A^*$  (англ. A star) — алгоритм поиска, который находит во взвешенном графе маршрут наименьшей стоимости от начальной вершины до выбранной конечной.

## **Описание алгоритмов.**

### **1. Жадный алгоритм.**

На каждом шаге алгоритма рассматриваются потомки текущей вершины, т.е. связанные с ней напрямую. Следующая вершина выбирается по принципу минимальности расстояния от текущей до следующей вершины. Все вершины, через которые осуществляется проход помечаются просмотренными, проход осуществляется только в не посещенные вершины. Если так получилось, что из текущей вершины нет пути в не посещенные вершины, алгоритм откатывается на шаг назад и снова пробует пройти в другую вершину. Свою работу он закончит если дойдет до финальной вершины или так и не попадет в нее.

Что касается сложности данного алгоритма, то она будет равна  $O(|V|+|E|)$  - по количеству операций, т.к. во время прохода мы просматриваем все ребра, выбирая минимальный вес среди них.

### **2. Алгоритм $A^*$ .**

Алгоритм  $A^*$  работает практически как жадный алгоритм, но есть одно очень важное отличие. Если жадный алгоритм выбивает путь в зависимости от пути от двух соседних вершин, то  $A^*$  смотрит пути от текущей до стартовой. То есть оценивается уже пройденное расстояние и расстояние, которое предположительно придется пройти. Это расстояние высчитывается с помощью приближения эвристической функции, которая обозначается

буквой  $h$ . Алгоритм использует очередь с приоритетом по значению функции оценки расстояния:  $f(v) = g(v) + h(v)$ .

## Описание функций и структур данных.

### 1. Жадный алгоритм.

Реализация жадного алгоритма в моем случае не использует отдельных функций и структур данных, так как является довольно тривиальной.

Граф хранится в структуре следующего вида:

```
std::vector<std::pair<char, std::pair<char, float>>>
```

То есть это вектор пар из имени вершины  $u$  и пары вершины  $v$  и расстояния между ними.

Работа алгоритма осуществляется в функции `main()`.

Необходимость заводить массив для просмотренных вершин была устранена удалением других ребер до вершины, в которую мы уже пришли. Так была осуществлена экономия памяти.

### 2. Алгоритм $A^*$

В данном алгоритме используются следующие структуры данных:

`struct Vertex` (который хранит имя вершины и длину пути до нее).

`using Graph = std::map<char, std::vector<Vertex>>` (структура для хранения графа, которая представляет из себя словарь вершины-массив структур `Vertex`).

Для хранения просмотренных вершин, а также для очереди используется структура:

`std::vector<std::tuple<char, char, float>>` (вектор кортежей из вершины  $u$ , вершины  $v$ , и пути между ними).

Очередь с приоритетом создается путем сортировки очереди.

### Тестирование.

Входные данные	Жадный алгоритм	Алгоритм A*
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	Граф считан корректно -----Начало работы жадного алгоритма----- Была добавлена вершина: a Была добавлена вершина: b Была добавлена вершина: c Была добавлена вершина: d Была добавлена вершина: e Результат работы алгоритма: abcde	-----Начало работы алгоритма A*----- Просмотрен новый путь: a -> b c сумарной длиной 3 Просмотрен новый путь: a -> d c сумарной длиной 5 Просмотрен новый путь: b -> c c сумарной длиной 4 Просмотрен новый путь: d -> e c сумарной длиной 6 Результат работы алгоритма: ade
a d a b 2 b d 10 a c 3 c d 1	Граф считан корректно -----Начало работы жадного алгоритма----- Была добавлена вершина: a Была добавлена вершина: b Была добавлена вершина: d Результат работы алгоритма:	-----Начало работы алгоритма A*----- Просмотрен новый путь: a -> b c сумарной длиной 2 Просмотрен новый путь: a -> c c сумарной длиной 3 Просмотрен новый путь: c -> d c сумарной длиной

	abd	4 Результат работы алгоритма: acd
a e a b 1 b c 1 c e 1 b d 2 d e 3	Граф считан корректно -----Начало работы жадного алгоритма----- Была добавлена вершина: a Была добавлена вершина: b Была добавлена вершина: c Была добавлена вершина: e Результат работы алгоритма: abce	-----Начало работы алгоритма A*----- Просмотрен новый путь: a -> b с сумарной длиной 1 Просмотрен новый путь: b -> c с сумарной длиной 2 Просмотрен новый путь: c -> e с сумарной длиной 3 Результат работы алгоритма: abce

Как видно жадный алгоритм далеко не всегда выдает минимальный результат, поэтому решать задачу нахождения минимального пути жадностью не имеет смысла.

### **Выводы.**

В ходе выполнения лабораторной работы был изучен и реализован жадный алгоритм поиска пути в ориентированном графе, а также был реализован алгоритм  $A^*$ , который уже находит кратчайшее расстояние в ориентированном графе.



## ИСХОДНЫЙ КОД ПРОГРАММЫ

### Файл grid.cpp:

```
#include <iostream>

#include <vector>

#include <climits>

int main()

{

    int length_min = INT_MAX;

    char start, end;

    std::cin >> start >> end;

    std::vector<std::pair<char, std::pair<char, float>>> graph;

    std::pair<char, float> best_way;

    char from, to;

    float graph_length;

    std::string answer;

    while (std::cin >> from >> to >> graph_length)

    {

        graph.push_back(std::make_pair(from, std::make_pair(to,

graph_length)));

    }

    std::cout << "Граф считан корректно\n";

    std::cout << "-----Начало работы жадного алгоритма-----\n";

    while (1)

    {

        answer += start;
```

```

std::cout << "Была добавлена вершина: " << start << "\n";

if (start == end)
{
    break;
}

for (int i = 0; i < graph.size(); i++)
{
    if ((graph[i].first == start) && (graph[i].second.second < length_min))
    {
        length_min = graph[i].second.second;
        best_way = graph[i].second;
    }
}

if (length_min == INT_MAX)
{
    answer.erase(answer.size()-1);
    start = answer[answer.size()-1];
    answer.erase(answer.size()-1);
    continue;
}

start = best_way.first;
length_min = INT_MAX;

```

```

        for (int i = 0; i < graph.size(); i++)
        {
            if (graph[i].second.first == start)
            {
                graph.erase(graph.begin()+i);
                i--;
            }
        }
    }

    std::cout << "Результат работы алгоритма:\n";

    std::cout << answer << '\n';

    return 0;
}

```

#### **Файл `astar.cpp`:**

```

#include <iostream>

#include <map>

#include <tuple>

#include <vector>

#include <algorithm>

bool cmp (const std::tuple<char, char, float> &first, const std::tuple<char,
char, float> &second)
{
    if (std::get<2>(first) < std::get<2>(second))
    {
        return false;
    }
}

```

```

    }

    return true;
}

int h (char vertex, char finish)
{
    return abs(finish - vertex);
};

struct Vertex
{
    char name;

    float len;

    Vertex(char name, float len)
    {
        this->name = name;

        this->len = len;
    }
};

using Graph = std::map<char, std::vector<Vertex>>;

int main()
{
    char start, finish;

    std::cin >> start >> finish;

    char from, to;

```

```

float len;

Graph graph;

while (std::cin >> from >> to >> len)
{
    Vertex new_vertex(to, len);

    graph[from].push_back(new_vertex);
}

std::cout << "-----Начало работы алгоритма A*-----\n";

char current_vertex = start;

float path = 0;

std::vector<std::tuple<char, char, float>> viewed;

std::vector<std::tuple<char, char, float>> queue;

while (current_vertex != finish)
{
    for(auto vertex = graph[current_vertex].begin(); vertex !=
graph[current_vertex].end(); vertex++)
    {
        queue.push_back(std::make_tuple(current_vertex, vertex->name,
path + vertex->len + h(vertex->name, finish)));
    }

    std::sort(queue.begin(), queue.end(), cmp);

    char from;

    while(!queue.empty())
    {
        from = std::get<0>(queue.back());

        current_vertex = std::get<1>(queue.back());

        path = std::get<2>(queue.back())-h(current_vertex, finish);
    }
}

```

```

queue.pop_back();

bool is_shorter = false;

for(auto vertex = viewed.begin(); vertex != viewed.end(); vertex++)
{
    if (std::get<1>(*vertex) == current_vertex)
    {
        if(std::get<2>(*vertex) <= path)
        {
            is_shorter = true;
            break;
        }
        else
        {
            viewed.erase(vertex);
            break;
        }
    }
}

if (is_shorter)
{
    continue;
}
else
{
    break;
}
}

```

```

        std::cout << "Просмотрен новый путь:\n";

        std::cout << from << " -> " << current_vertex << " с суммарной
длиной " << path << "\n";

        viewed.push_back(std::make_tuple(from, current_vertex, path));
    }

    std::string answer;

    answer+=finish;

    current_vertex = finish;

    while(current_vertex != start)
    {
        for (auto vertex = viewed.begin(); vertex != viewed.end(); vertex++)
        {
            if(std::get<1>(*vertex) == current_vertex)
            {
                current_vertex = std::get<0>(*vertex);
                break;
            }
        }

        answer+=current_vertex;
    }

    std::cout << "Результат работы алгоритма:\n";

    std::reverse(answer.begin(), answer.end());

    std::cout << answer << '\n';

    return 0;
}

```