

**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. 8382

\_\_\_\_\_

Ершов М.И.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2020

## Цель работы.

Исследование работы алгоритмов поиска пути в ориентированном 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 7.** "Мультипоточный" A\*: на каждом шаге из очереди с приоритетами извлекается  $n$  вершин (или все вершины, если в очереди меньше  $n$  вершин).  $n$  задаётся пользователем.

### **Жадный алгоритм.**

Один из способов поиска пути в программе – жадный алгоритм.

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.

### **Описание алгоритма $A^*$**

В программе так же реализуется 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 алгоритма выбираются  $n$  частичных решений с минимальным приоритетом (по заданию индивидуализации). Затем, для каждой из  $n$  вершин выполняется поиск смежных с минимальным весом от начальной вершины, такие вершины попадают в очередь с приоритетом, для дальнейшей обработки. Алгоритм работает, пока очередь с приоритетом не станет пустой, либо при нахождении финишной вершины.

Когда эвристическая функция идеальная и на каждом шаге указано верное направление, получаем сложность  $O(N+M)$ , где  $N$  – количество вершин,  $M$  – количество ребер. Так, максимальная длина пути –  $N$  и на каждом шаге требуется пройти по всем ребрам, выходящим из текущей вершины для того, чтобы найти путь с наименьшим приоритетом.

В худшем случае, когда вершины расположены случайным образом и эвристическая функция не идеальна, будут рассмотрены все пути. Отсюда сложность по времени  $O(N^M)$ , где  $N$  – кол-во вершин, а  $M$  – кол-во ребер в графе.

Так как промежуточные пути хранятся в контейнере, и длина пути не может превосходить число вершин в графе, то получаем сложность по памяти  $O(N!)$ .

### **Описание структур данных.**

Compare:

bool operator() – компаратор для очереди с приоритетом;

Структура SetCompare:

bool operator() – компаратор для множества рёбер;

Класс Graph:

`map <char, set<pair<char, double>, SetCompare>> ways` – список смежности;  
Класс используется для хранения информации о графе в виде списка смежности. Так, каждой вершине соответствует множество выходящих из нее ребер.

### **Описание функций.**

Класс Graph:

`set<pair<char,double>, SetCompare>& operator[] (char way)`

- `way` – вершина, к которой добавляется ребро.

Функция возвращает для соответствующей вершины список вершин, к которым она ведет

`string greedySearch(char start, char goal)`

- `start` – вершина, от которой необходимо найти путь к `goal`;
- `goal` – вершина, к которой надо найти путь из `start`. Функция возвращает найденный путь;

Функция используется для поиска пути в графе при помощи жадного алгоритма. Возвращает найденный путь.

`string aStarSearch(char start, char goal)`

- `start` – вершина, от которой необходимо найти путь к `goal`;
- `goal` – вершина, к которой надо найти путь из `start`.

Функция используется для поиска пути в графе при помощи жадного алгоритма. Возвращает найденный путь.

### **Тестирование.**

<p>Сколько вершин извлекать из очереди с приоритетами?</p> <pre> a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 a e 1.0 </pre> <p>Жадный алгоритм, промежуточный путь:</p> <pre> a ab abc abcd abcde </pre> <p>Жадный алгоритм: abcde</p> <p>A*, промежуточный путь:</p> <pre> path_from_start[a]: a path_from_start[b]: ab path_from_start[d]: ad path_from_start[e]: ade </pre> <p>A*: ade</p>	<p>Сколько вершин извлекать из очереди с приоритетами?</p> <pre> a a b 3.0 b c 1.0 c d 1.0 a d 2.0 a e 1.0 </pre> <p>Жадный алгоритм, промежуточный путь:</p> <pre> a ad ade </pre> <p>Жадный алгоритм: ade</p> <p>A*, промежуточный путь:</p> <pre> path_from_start[a]: a path_from_start[d]: ad path_from_start[b]: ab path_from_start[e]: ade </pre> <p>A*: ade</p>
<p>Сколько вершин извлекать из очереди с приоритетами?</p> <pre> a a e a b 3.0 b c 1.0 c d 3.0 a e 5.0 a e 4.0 </pre> <p>Жадный алгоритм, промежуточный путь:</p> <pre> a ab abc abcd abcde </pre> <p>Жадный алгоритм: abcde</p> <p>A*, промежуточный путь:</p> <pre> path_from_start[a]: a path_from_start[b]: ab path_from_start[e]: ae </pre> <p>A*: ae</p>	<p>Сколько вершин извлекать из очереди с приоритетами?</p> <pre> a e a b 4 b c 1 a d 5 a e 4 a d 1 </pre> <p>Жадный алгоритм, промежуточный путь:</p> <pre> a ad adc </pre> <p>Жадный алгоритм: adc</p> <p>A*, промежуточный путь:</p> <pre> path_from_start[a]: a path_from_start[d]: ad path_from_start[b]: ab path_from_start[c]: adc path_from_start[c]: abc </pre> <p>A*: abc</p>
<p>Сколько вершин извлекать из очереди с приоритетами?</p> <pre> a e a e 7 b c 8 a e 7 c d 1 a e 1 </pre> <p>Жадный алгоритм, промежуточный путь:</p> <pre> a ae aeb </pre> <p>Жадный алгоритм: aeb</p> <p>A*, промежуточный путь:</p> <pre> path_from_start[a]: a path_from_start[e]: ae path_from_start[c]: ac path_from_start[d]: acd path_from_start[b]: aeb path_from_start[b]: acdb </pre> <p>A*: acdb</p>	<p>Сколько вершин извлекать из очереди с приоритетами?</p> <pre> a e a b 1 b c 1 c d 1 c e 1000 </pre> <p>Жадный алгоритм, промежуточный путь:</p> <pre> d db dbc dbca </pre> <p>Жадный алгоритм: dbca</p> <p>A*, промежуточный путь:</p> <pre> path_from_start[d]: d path_from_start[b]: db path_from_start[c]: dbc path_from_start[d]: dbcd path_from_start[a]: dbca </pre> <p>A*: dbca</p>

## Вывод.

В ходе выполнения лабораторной работы была исследована работа алгоритмов поиска пути в ориентированном графе путем написания программ, реализующих жадный алгоритм поиска и алгоритм поиска A\*.

## Приложение А.

### Исходный код

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

using namespace std;

// Компаратор для множества ребер
struct SetCompare
{
    bool operator()(pair<char, double> v1, pair<char, double> v2)
    {
        if (v1.second == v2.second)
            return v1.first < v2.first;
```

```

        return v1.second < v2.second;
    }
};

// Компаратор для очереди с приоритетом
struct Compare
{
    bool operator()(pair<char, double> v1, pair<char, double> v2)
    {
        if (v1.second == v2.second)
            return v1.first < v2.first;
        return v1.second > v2.second;
    }
};

class Graph {
private:
    // map - отсортированный ассоциативный контейнер, порядок ключей - SetCompare
    // каждой вершине соответствует множество выходящих из нее ребер
    map <char, set<pair<char, double>, SetCompare>> ways; // список смежности

public:
    // set - ассоциативный контейнер пар (pair) с компаратором SetCompare
    // Возвращает контейнер с вершинами, к которым ведет way
    set<pair<char, double>, SetCompare>& operator[](char way)
    {
        return ways[way];
    }

    string greedySearch(char start, char goal)
    {
        map <char, bool> checked; // контейнер с просмотренными вершинами
        string res; // итоговый путь

        cout << "Жадный алгоритм, промежуточный путь:" << endl;

        res.push_back(start); // сразу добавляем стартовую вершину

```



```

checked[start] = true; // просмотрели стартовую
char curr = start; // текущая - стартовая

cout << res << endl;

do {
    bool path_from_current = false; // флаг, указывающий на то, есть ли путь из
текущей вершины в другую
    pair <char, double> temp; // смежная к curr, с минимальным весом

    // Рассматриваем первую смежную вершину
    if (!ways[curr].empty()) {
        for (auto it = ways[curr].begin(); !path_from_current && it !=
ways[curr].end(); it++) {
            temp = *it;
            if (!checked[it->first]) path_from_current = true;
        }
    }

    // Если не нашли смежных вершин - откатываемся
    if (!path_from_current) {
        cout << res << endl;
        res.pop_back();
        curr = *res.rbegin();
    } else { // Нашли - меняем текущую на найденную
        curr = temp.first;
        res.push_back(curr);
        cout << res << endl;
        checked[curr] = true;
    }
} while (curr != goal && !res.empty()); // Выполняем, пока текущая не станет
конечной

return res;
}

string aStarSearch(char start, char goal, int n = 1)

```

```

{
    priority_queue <pair<char, double>, vector<pair<char, double>>, Compare>
queue; // Очередь с приоритетом
    map <char, string> path_from_start; // Контейнер с путями от стартовой до char
    map <char, double> weight_from_start; // Контейнер с весами от стартовой до char

    path_from_start[start] = start;
    cout << "A*, промежуточный путь:" << endl;
    cout << "path_from_start[" << start << "]: " << path_from_start[start] <<
endl;
    weight_from_start[start] = 0;
    queue.push(*weight_from_start.begin());

    while (!queue.empty())
    {
        vector <pair<char, double>> peaks; // n верхних элементов очереди, для
"мультипоточности"

        auto curr = queue.top(); // Вытаскиваем верх очереди и проверяем на равенство
конечной вершине
        if (curr.first == goal) return path_from_start[goal];

        for (int i = 0; i < n && !queue.empty(); i++) {
            curr = queue.top();
            if (curr.first == goal) continue;
            queue.pop();
            peaks.push_back(curr);
        }

        for (auto &peak : peaks) { // Обрабатываем все снятые с очереди вершины
            for (auto &v : ways[peak.first]) { // Обрабатываем смежные к peak
вершины
                double new_w = weight_from_start[peak.first] + v.second; // Новый
вес для текущей смежной вершины

                // Если вес уменьшился или он еще не вычислен
                if (new_w < weight_from_start[v.first] ||
!weight_from_start[v.first]) {

```

```

        weight_from_start[v.first] = new_w; // Обновляем вес
        path_from_start[v.first] = path_from_start[peak.first] +
v.first; // Обновляем путь

        cout << "path_from_start[" << v.first << "]: " <<
path_from_start[v.first] << endl;

        // Пушим в очередь с приоритетом вершину
        queue.push({v.first, new_w + abs(goal - v.first)});
    }
}
}

return {}; // Если очередь пустая - возвращаем пустой путь
}
};

int main() {
    Graph graph;
    int n;
    char startPeak, finishPeak;

    cout << "Сколько вершин извлекать из очереди с приоритетами?";
    cin >> n;

    cin >> startPeak >> finishPeak;

    char from, to;
    double weight;

    while (cin >> from && from != '.' && cin >> to >> weight)
        graph[from].emplace(to, weight);

    string greedy = "Жадный алгоритм: " + graph.greedySearch(startPeak, finishPeak);
    cout << greedy << endl;

    string a_star = "A*: " + graph.aStarSearch(startPeak, finishPeak, n);

```

```
    cout << a_star << endl;  
  
    return 0;  
}
```