

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

**ОТЧЕТ**  
**по лабораторной работе №3**  
**по дисциплине «Построение и анализ алгоритмов»**  
**ТЕМА: Поток в сети**

Студент гр. 8303		Сенюшкин Е.В.
Преподаватель		Фирсов М.А.

Санкт-Петербург

2020

## Цель работы.

Изучение алгоритма Форда-Фалкерсона для нахождения максимального потока в графе.

## Задание.

Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Сеть (ориентированный взвешенный граф) представляется в виде триплета из имён вершин и целого неотрицательного числа - пропускной способности (веса).

Входные данные:

$N$  - количество ориентированных рёбер графа

$v_0$  - исток

$v_n$  - сток

$v_i v_j \omega_{ij}$  - ребро графа

$v_i v_j \omega_{ij}$  - ребро графа

...

Выходные данные:

$P_{max}$  - величина максимального потока

$v_i v_j \omega_{ij}$  - ребро графа с фактической величиной протекающего потока

$v_i v_j \omega_{ij}$  - ребро графа с фактической величиной протекающего потока

...

В ответе выходные рёбра отсортируйте в лексикографическом порядке по первой вершине, потом по второй (в ответе должны присутствовать все указанные входные рёбра, даже если поток в них равен 0).

## Индивидуализация.

Вар. 5. Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность. Если таких дуг несколько, то выбрать ту, которая была обнаружена раньше в текущем поиске пути.

## Описание алгоритма Форда-Фалкерсона.

*Остаточная сеть* — это граф  $G_f = (V, E_f)$ , где  $E_f$  - множество ребер с положительной остаточной пропускной способностью. В остаточной сети может быть путь из  $u$  в  $v$ , даже если его нет в исходном графе. Это выполняется, когда в исходной сети есть обратный путь  $(v, u)$  и поток по нему положительный.

*Дополняющий путь* — это путь в остаточной сети.

Вначале алгоритма *остаточная сеть* — это исходный граф. Алгоритм пытается найти *дополняющий путь* пока может, величина потока, идущая по *дополняющему пути* равна минимальной пропускной способности ребра, находящегося в пути, если путь найти не удалось значит был найден максимальный поток в графе и алгоритм останавливается.

## Описание функций и структур данных.

Структуры данных.

struct Node - структура для хранения остаточной пропускной способности основного и вспомогательного ребра.

vector<vector<Node>> network(128, vector<Node>(128)) - матрица смежности для хранения остаточной сети.

vector<pair<int, int>> graph - вектор пар в котором хранится исходный граф.

vector<bool> mark - вектор, в котором отмечаются уже посещенные вершины.

int source, sink - сток и исток.

Функции.

int dfs(int v, int delta) - рекурсивный поиск в глубину с модификацией: каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность.

Функция принимает на вход вершину, из которой ищется путь и текущая минимальная пропускная способность на пути.

Возвращает минимальную пропускную способность на пути.

`void print()` - функция выводит ребра исходного графа с фактической величиной протекающего потока

`void readGraph()` - функция считывает граф и создает начальную остаточную сеть.

`void FFA()` - функция, которая находит максимальный поток в графе.

### **Сложность алгоритма.**

$E$  – множество ребер графа.

$V$  – множество вершин графа.

$F$  – величина максимальной пропускной способности графа.

### **По времени.**

На каждом шаге мы ищем путь от стока к истоку, поиском в глубину с модификацией это работает за  $O(|E| + |E| * \log|E| * |V|)$  так как в каждой вершине мы сортируем все исходящие из нее ребра.

В худшем случае, на каждом шаге мы будем находить дополняющий путь с пропускной способностью 1, тогда получим сложность по времени  $O(F * |E| * \log|E| * |V|)$ .

### **По памяти.**

Для хранения остаточной используется матрица смежности, так что сложность по памяти  $O(|V|^2)$ . Так же используется дополнительная память для хранения исходного графа в виде массива  $O(|E|)$  и массив для хранения посещенных вершин, тоже от  $O(|E|)$ . Таким образом, сложность по памяти получается  $O(|V|^2)$ .

### Тестирование.

Input	Output
6 a a a c 10 c d 10 c b 1 b c 1 a b 10 b d 10	0 a b 0 a c 0 b c 0 b d 0 c b 0 c d 0
11 a d a b 7 a c 3 a f 5 c b 4 c d 5 b d 6 b f 3 b e 4 f b 7 f e 8 e d 10	15 a b 7 a c 3 a f 5 b d 6 b e 1 b f 0 c b 0 c d 3 e d 6 f b 0 f e 5
7 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2

10	23
a	a b 12
f	a c 11
a b 16	b c 0
a c 13	b d 12
c b 4	c b 0
b c 10	c e 11
b d 12	d c 0
c e 14	d f 19
d c 9	e d 7
d f 20	e f 4
e d 7	
e f 4	

### **Вывод.**

В ходе лабораторной работы был изучен алгоритм поиска максимального потока в сети - метод Форда-Фалкерсона.

## Приложение А.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <climits>
#include <set>

using namespace std;

struct Node{ // структура для хранения прямого и дополнительно ребра в
остаточной сети
    int f1;
    int f2;

    Node() : f1(0), f2(0) {}
    Node(int f1, int f2) : f1(f1), f2(f2){}
};

vector<vector<Node>> network(128, vector<Node>(128)); // остаточная сеть
vector<pair<int, int>> graph; // исходный граф
vector<bool> mark; // посещенные вершины
int source, sink; // исток и сток

int dfs(int v, int delta){ // функция для поиска дополняющего пути
    //cout << static_cast<char>(v);
    if (mark[v]) // если вершина уже была посещена выходим из нее
        return 0;
    mark[v] = true; // помечаем вершину как посещенную

    if (v == sink) // если текущая вершина является конечная, то мы нашли
дополняющий путь, возвращаемся обратно и по пути применяем найденную
минимальную пропускную способность
        return delta;
```

```

    set<pair<int, int>, greater<>> q; // сет для сортировки вершин по
пропускной способности

    for (size_t u = 0; u < network[v].size(); u++)
    {
        if (!mark[u] && network[v][u].f2 > 0)
            q.insert(make_pair(network[v][u].f2, u));
        if (!mark[u] && network[v][u].f1 > 0)
            q.insert((make_pair(network[v][u].f1, u)));
    }

    for(auto u : q){ // обходим все смежные вершины в порядке приоритета
        if (network[v][u.second].f2 > 0){
            int newDelta = dfs(u.second, min(delta,
network[v][u.second].f2));
            if (newDelta > 0){
                network[u.second][v].f1 += newDelta; //применяем
минимальную пропускную способность
                network[v][u.second].f2 -= newDelta;
                return newDelta;
            }
        }
        if (network[v][u.second].f1 > 0){
            int newDelta = dfs(u.second, min(delta,
network[v][u.second].f1));
            if (newDelta > 0) {
                network[u.second][v].f2 += newDelta; //применяем
минимальную пропускную способность
                network[v][u.second].f1 -= newDelta;
                return newDelta;
            }
        }
    }

    return 0;
}

```



```

void print(){
    sort(graph.begin(), graph.end());

    for(size_t i = 0; i < graph.size(); i++){ // выводим ребра графа и то
        сколько по ним было пущено потока

        cout << static_cast<char>(graph[i].first) << ' ' <<
            static_cast<char>(graph[i].second) << ' ' <<
            network[graph[i].second][graph[i].first].f2 << endl;

    }

}

void readGraph(){
    int N;
    char u, v;
    int c; // capacity

    cin >> N;

    cin >> u >> v;
    source = static_cast<int>(u);
    sink = static_cast<int>(v);

    mark.resize(128);

    for(size_t _ = 0; _ < N; _++){
        cin >> u >> v >> c;

        int i = static_cast<int>(u); //u
        int j = static_cast<int>(v); //v

        graph.emplace_back(i, j);
        network[i][j].f1 = c;

    }

}

```

```

void FFA(){
    int flow = 0;
    int ans = 0;
    while (true){
        fill(mark.begin(), mark.end(), false);
        flow = dfs(source, INT_MAX);
        cout << endl;

        if (flow == 0 || flow == INT_MAX) { // если функция вернула поток
равный нулю или INT_MAX, значит не было найдено дополняющего пути и был
найден максимальный поток
            //cout << "Path not found " << endl;
            break;
        }else {
            //cout << "A new path increased flow by: " << flow << endl;
        }
        ans +=flow;
    }
    //cout << endl;
    // cout << "Max flow: ";
    cout << ans << endl;
    print();
}

int main(){
    readGraph();
    //cout << endl;
    FFA();
    return 0;
}

```