

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

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

Студент гр. 9382

\_\_\_\_\_

Иерусалимов Н.

Преподаватель

\_\_\_\_\_

Фирсов М. А.

Санкт-Петербург

2021

### **Цель работы.**

Познакомиться с одним из часто используемых на практике алгоритмом, поиска потоков в сети. Получить навыки решения задач на этот алгоритм.

### **Задание.**

Вар. 1. Поиск в ширину. Поочерёдная обработка вершин текущего фронта, перебор вершин в алфавитном порядке.

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

$v_i v_j w_{ij}$ - ребро графа

$v_i v_j w_{ij}$ - ребро графа

...

Выходные данные:

$P_{max}$  -величина максимального потока

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

$v_i v_j w_{ij}$ - 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).

---

**Sample Input:**

```
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
```

---

**Sample Output:**

```
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

**Описание алгоритма.**

На каждой итерации алгоритма происходит поиск в ширину, от истока до стока в начальном графе. Путь сохраняется, и дальше алгоритм с помощью него проходится по основному графу. С помощью пути мы находим нужные вершины в графе  $K1$  и высчитываем минимальную пропускную способность в этом пути. Проходя через все ребра в пути, вычитает минимальную пропускную способность от начальной. Потом ищем эти же вершины в графе  $K2$  только уже в обратном порядке, (Для чего нам рассматривать граф в

обратном порядке? Это нужно для того, что-бы направить поток в противоположную сторону от изначального.) и там уже мы прибавляем минимальную пропускную способность к изначальной. Далее мы прибавляем найденный минимальный поток к общему потоку. Алгоритм завершает свою работу, когда поиск в ширину не сможет найти еще один путь до стока

### **Сложность алгоритма**

В худшем случае алгоритм будет на каждой итерации инкрементировать поток на единицу. То есть в худшем случае, итераций в алгоритме будет  $N$ , где  $N$  - это величина потока, и так как мы на каждой итерации производим поиск в ширину который занимает в худшем случае  $O(U + V)$ , Где  $U$  – количество вершин,  $V$  – количество ребер. И тогда общая сложность алгоритма будет составлять  $O((U + V) * N)$ .

Сложность по памяти  $O(2(U + V))$  Где  $U$  – количество вершин,  $V$  – количество ребер. Двойка так как у нас два таких графа, один тот что ввел пользователь, другой такой же только ребра направлены в другую сторону.

### **Описание функций и структур данных.**

`class Graph` - класс граф

`void inputGraph()` – Метод для ввода данных в граф

`template<class T>`

`vector<pair<char, int>> findNeighbor(char sought, T whereSearch)` – Шаблонный метод который ищет соседей к интересующей нас вершине

Возвращает вектор пар имя – вес, где имя – имя вершины, вес – пропускная способность ребра до этой вершины.

Принимает искомый элемент, и где надо искать.

`bool depthSearch()` – поиск в ширину в графе.

Возвращает `True` если путь найден, `False` если путь не найден

`int findK2(vector<pair<char, int>> k2, char b)` – Ищет нужную вершину

Принимает вектор пар где искать и имя искомого элемента.

Возвращает найденное значение для обратного потока.

`void fulkerson()` – метод для поиска максимального потока.

`char source, runoff;` - имя истока и стока соответственно

`map<char, map<char, int>> graphK1` - Структура где хранится граф. Имя вершины это ключ к ее соседям, а они в свою очередь хранят свое имя которое является ключом для значения пропускной способности ребра.

`map<char, map<char, int>> graphK2` – Та же структура где хранится граф, только запись в него происходит в другую сторону от исходной.

`map<char, char> path` – Структура для записи пути, используется чтобы записать путь когда происходит поиск в ширину. Первое имя является ключом к следующему имени.

### Тестирование.

№	Входные данные	Выходные данные
1	7 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4	12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2

	e c 2	
2	9 a g a b 30 a c 1 b d 200 b e 3 d e 40 e a 3 e f 20 a g 800 f g 10	810 a b 10 a c 0 a g 800 b d 7 b e 3 d e 7 e a 0 e f 10 f g 10
3	5 a e a b 7 a c 3 b c 15 c d 82 b e 40	7 a b 7 a c 0 b c 0 b e 7 c d 0
4	6 a f a c 15 a b 17 c d 26 b e 22 d f 33 e f 35	32 a b 17 a c 15 b e 17 c d 15 d f 15 e f 17
5	6 a d a b 1 b c 9 c d 3 a d 9 a e 1 e d 3	11 a b 1 a d 9 a e 1 b c 1 c d 1 e d 1

### **Выводы.**

Был исследован алгоритм - поиск максимального потока в графе. Также были получены навыки решения задач на этот алгоритм.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

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

```
bool choise = 0;
```

```
class Graph {
public:
    void inputGraph() {
        int size;
        std::cin >> size ;
        std::cin >> source >> runoff;
        for(int i =0; i <size;++i) {
            char u, v;
            int mass;
            std::cin >> u >> v >> mass;
            graphK1[u][v] = mass;
        }
    }
}
```

```
template<class T>
std::vector<std::pair<char, int>> findNeighbor(char sought, T whereSearch) {

    std::vector<std::pair<char, int>> answ;
    std::pair<char, int> edge;
    for (auto vertex : whereSearch[sought]) {
        edge.first = vertex.first;
        edge.second = vertex.second;
        answ.push_back(edge);
    }
    return answ;
};
```

```
bool depthSearch() { //поиск в ширину
    if (choise) {
        std::cout << "\n_____Breadth First Search_____ \n";
    }
}
```



```

std::vector<char> tempVert; //Список вершин которые хотим посетить
std::map<char, bool> visitedVertex; // Пара: имя - состояние. Список посещенных вершин
char sought; //искомая вершина. Используется для поиска соседей этой вершины
visitedVertex[source] = true;
tempVert.push_back(source);

//Пока есть вершины в стеке, продолжаем рассматривать.
while (!tempVert.empty()) {

    if (choise) {
        std::cout << "\n\tThere are unconsidered vertices in the stack\n";
    }

    //Смотрим соседей текущей вершины
    sought = tempVert.back();
    tempVert.pop_back();
    if (choise) {
        std::cout << "\tWe are looking for neighbors, for the vertex: " << sought << "\n\n";
    }

    for (auto enumEdge: findNeighbor(sought, graphK1)) {

        if (choise) {
            std::cout << "\t\t\t " << sought << "->" << enumEdge.first << " \n";
        }

        //Если вес ребра больше нуля и мы не посещали ее
        if (enumEdge.second > 0 && !visitedVertex[enumEdge.first]) {
            if (choise) {
                std::cout << "\t\t The weight > 0 && not been visited\n"
                    << "\t\t Write to the path...\n\n";
            }
            tempVert.push_back(enumEdge.first); //Добавляем имя вершины для просмотра на
след.иттерации
            path[enumEdge.first] = sought; // Сохраняем путь.
            visitedVertex[enumEdge.first] = true; // Посетили текущую

            //Если текущая вершина равна истоку, выходим и возвращаем true, путь найден.
            if (enumEdge.first == runoff) {

                if (choise) {
                    char a, b;
                    std::cout << "\t\tCurrent vertex is equal to source, search is complete!\n \t\tPath is
equal: "

                    << runoff;

```

```

        for (a = this->runoff; a != this->source; a = b) {
            b = path[a];
            std::cout << "<- " << path[a];
        }
        std::cout << "\n\n";
    }

    return true;
}

}

}
if (choise) {
    std::cout << "\n\tPATH NOT FOUND!!!\n";
}
//Все вершины рассмотрены, путь не найден.
return false;
}

int findK2(std::vector<std::pair<char, int>> k2, char b) {
    for (auto i : k2) {
        if (i.first == b) {
            return i.second;
        }
    }
    return 0;
}

/*

```

(1) С помощью поиска в ширину мы находим путь до истока, параллельно записываем его.

(1.1) Если путь до истока был найден, мы проходимся по записанному пути и находим минимальную пропускную способность.

(1.1.1) После чего мы снова проходимся по всему пути и от K1 отнимаем найденную минимальную величину, а к K2 прибавляем её. ( $K1 -= \min$ ) ( $K2 += \min$ )

K1 - вес ребра, пропускная способность ребра. K2 - вес ребра в другую сторону, проходимый поток.

(1.1.2) Прибавляем минимальную пропускную способность к максимальному потоку

(2) Путь не найден. Выходим из цикла и выводим результат.

\*/

```

void fulkerson() {
    char tempRunoff, tempSource;
    int maxFlow = 0;

    //(1)
    while (depthSearch()) {
        if (choise) {
            std::cout <<
"_____ \n\t\t\tFulkerson\n";
        }
        int flowInEdge = INT32_MAX;

        //(1.1)
        if (choise) {
            std::cout << "Minimum bandwidth for path: " << runoff;
        }
        for (tempRunoff = this->runoff; tempRunoff != this->source; tempRunoff = tempSource) {
            tempSource = path[tempRunoff];
            flowInEdge = std::min(flowInEdge, graphK1[tempSource][tempRunoff]);

            if (choise) {
                std::cout << "<- " << path[tempRunoff];
            }
        }

        if (choise) {
            std::cout << ", is equal: " << flowInEdge << "\n";
        }

        //(1.1.1)
        if (choise) {
            std::cout<< "We go all the path and change the values of K1 and K2.\n"
            "For the throughput of the rib and for the throughput in the opposite direction,
respectively\n";
        }
        for (tempRunoff = this->runoff; tempRunoff != this->source; tempRunoff = tempSource) {
            tempSource = path[tempRunoff];
            if (choise) {
                std::cout<<"\t
"<<graphK1[tempSource][tempRunoff]<<"/"<<graphK2[tempRunoff][tempSource]<<"\n";
                std::cout<<"\t "<<tempSource<<" <-> "<< tempRunoff<<"\n";
                std::cout << "\tK1 = " << graphK1[tempSource][tempRunoff] << " - " << flowInEdge
<< "\n";
                std::cout << "\tK2 = " << graphK2[tempRunoff][tempSource] << " + " << flowInEdge
<< "\n";

```

```

    }
    graphK1[tempSource][tempRunoff] -= flowInEdge;
    graphK2[tempRunoff][tempSource] += flowInEdge;
    if (choise) {

        std::cout << "\tK1 {" << tempSource << " -> " << tempRunoff << " = " <<
graphK1[tempSource][tempRunoff]
        << " }\n";
        std::cout << "\tK2 {" << tempSource << " <- " << tempRunoff << " = " <<
graphK2[tempRunoff][tempSource]
        << " }\n\n";

    }

}

//(1.1.2)
if (choise) {
    std::cout<<"Add the minimum bandwidth to the maximum flow: "<<maxFlow<<" + "<<
flowInEdge<<"\n";
}

    maxFlow += flowInEdge;

}
//(2)
std::cout << maxFlow << "\n";
for (auto &vertex : graphK1) {
    for (auto neighbor : graphK1[vertex.first]) {
        auto temp = findNeighbor(neighbor.first, graphK2);
        std::cout << vertex.first << " " << neighbor.first << " " << findK2(temp, vertex.first) <<
"\n";
    }
}

}

private:
    char source, runoff; //исток, сток
    std::map<char, std::map<char, int>> graphK1; //исходный граф
    std::map<char, std::map<char, int>> graphK2; // граф с инвертированными ребрами и
проходящий через ребро поток. c->f стало f->c
    std::map<char, char> path;

```

```
};
```

```
int main() {  
    std::cout << "enable Intermediate data? 1 - Yes 0 - No\n";  
    std::cin >> choise;  
  
    if (choise) {  
        std::cout << "_____Input graph_____\\n";  
    }  
  
    Graph g;  
    g.inputGraph();  
  
    if (choise) {  
        std::cout << "\\n_____Fulkerson_____\\n";  
    }  
    g.fulkerson();  
    system("pause>nul");  
    return 0;  
}
```