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: Потоки в сети

Студент гр. 8304 _____ Бутко А.М.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург
2020

Цель работы.

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

Постановка задачи.

(Вариант 4)

Поиск в глубину. Итеративный метод

Реализация алгоритма.

Изначально величине потока присваивается значение 0 для всех ребер из графа. Затем величина потока итеративно увеличивается посредством поиска увеличивающего пути (путь от источника vSource к стоку vSink, вдоль которого можно послать ненулевой поток). Алгоритм осуществляет этот поиск с помощью итеративного обхода в глубину (DFS). Процесс повторяется, пока можно найти увеличивающий путь.

Оценка сложности алгоритма.

Сложность алгоритма O(EF), где E — количество ребер в графе, а F — величина максимального потока.

Описание структур данных и функций.

Граф хранится в словаре вида `std::map<char, Vertex> Graph`, структура `Vertex` хранит в себе информацию о всех «назначениях» из текущей вершины, о вершине из которой был совершен переход, информацию о том, является ли вершина частью пути, а так же информацию о каждом переходе (вес ребра и поток, которые являются полями структуры `VertexInfo`).

`bool iterativeDFS(std::map<char, Vertex> Graph, char vSource, char vSink, std::vector<char>& path)` – функция итеративного поиска в глубину, возвращает false, если пути из истока в сток не осталось. Поиск «доступной» вершины для перехода выполняет функция `char vAvailable(std::vector<std::pair<char, VertexInfo>> const& edges,std::map<char, Vertex> Graph)`.

`void algorithmFordFulkerson(std::map<char, Vertex>& Graph, char vSource, char vSink)` – функция-реализация алгоритма Форда-Фалкерсона, которая ищет увеличивающий путь, путь в свою очередь хранится в векторе `path`.

Тестирование.

Ввод	Вывод
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
5 A D A B 50 B C 50 C D 50 A C 5 B D 5	55 A B 50 A C 5 B C 45 B D 5 C D 50
10 A W A B 1 B C 2 C D 3 A C 4 C F 5 A F 6 F W 7 A F 8 A I 9 I W 10	21 A B 0 A C 0 A F 6 A F 6 A I 9 B C 0 C D 0 C F 0 F W 6 I W 9

Вывод.

В ходе выполнения работы был найден максимальный поток в сети, а также фактическая величиа потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД.

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


struct VertexInfo
{
    int weight;
    int flow;
};

struct Vertex
{
    bool isVisited;
    char vParent;
    std::vector<std::pair<char, VertexInfo>> edges;
};

char vAvailable(std::vector<std::pair<char, VertexInfo>> const&
edges,std::map<char, Vertex> Graph)
{
    char vAvailable = 0;
    for(auto vertex : edges)
        if(!Graph[vertex.first].isVisited && vertex.second.weight > 0)
            vAvailable = vertex.first;
    return vAvailable;
}

bool iterativeDFS(std::map<char, Vertex> Graph, char vSource, char vSink,
std::vector<char>& path)
{
    char vCurrent = vSource;
    char vSelected = vAvailable(Graph[vCurrent].edges, Graph);
    while(vCurrent != vSink && !(vCurrent == vSource && vSelected == 0))
    {
        Graph[vCurrent].isVisited = true;

        if(vSelected != 0)
        {
            Graph[vSelected].vParent = vCurrent;
            vCurrent = vSelected;
            path.push_back(vCurrent);
        }
        else
        {
            path.pop_back();
            vCurrent = Graph[vCurrent].vParent;
        }
        vSelected = vAvailable(Graph[vCurrent].edges, Graph);
    }
    return path == std::vector<char>{vSource};
}

void algorithmFordFulkerson(std::map<char, Vertex>& Graph, char vSource, char
vSink)
```

```
{
    std::vector<char> path{vSource};
    std::vector<char> tmp;
    int minFlow;
    while(!iterativeDFS(Graph, vSource, vSink, path))
    {
        minFlow = INT_MAX;
        tmp = path;

        while(path.size() != 1)
        {
            for(auto vertex : Graph[path[0]].edges)
                if(vertex.first == path[1] && vertex.second.weight < minFlow)
                    minFlow = vertex.second.weight;
            path.erase(path.begin());
        }

        path = tmp;

        while(path.size() != 1)
        {
            for (auto& vParent : Graph[path[0]].edges)
                if(vParent.first == path[1])
                {
                    vParent.second.flow += minFlow;
                    vParent.second.weight -= vParent.second.flow;
                    for(auto& vChild : Graph[path[1]].edges)
                        if(vChild.first == path[0])
                        {
                            vChild.second.flow -= minFlow;
                            vChild.second.weight -= vChild.second.flow;
                        }
                }
            path.erase(path.begin());
        }

        path[0] = vSource;
    }
}

bool compare(std::pair<char, VertexInfo> const& a, std::pair<char, VertexInfo>
const& b)
{
    return a.first < b.first;
}

void output(std::map<char, Vertex>& Graph, char vSource)
{
    int maxFlow = 0;
    for(auto i : Graph[vSource].edges)
        maxFlow += i.second.flow;
    std::cout << maxFlow <<std::endl;

    for(auto vFrom : Graph)
    {
        std::sort(vFrom.second.edges.begin(), vFrom.second.edges.end(),
compare);
        for(auto vTo : vFrom.second.edges)
            std::cout << vFrom.first << " " << vTo.first << " " << std::max(0,
vTo.second.flow) << std::endl;
```

```
    }
}

int main()
{
    std::map<char, Vertex> Graph;

    char vSource;
    char vSink;
    int vEdges;

    std::cin >> vEdges;
    std::cin >> vSource;
    std::cin >> vSink;

    char vFrom;
    char vTo;
    int vWeight;

    while(vEdges)
    {
        std::cin >> vFrom >> vTo >> vWeight;
        Graph[vFrom].edges.push_back({vTo,{vWeight, 0}});
        vEdges--;
    }

    algorithmFordFulkerson(Graph, vSource, vSink);
    output(Graph, vSource);
    return 0;
}
```