МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

Тема: Потоки в сети

Студент гр. 8304 Птухов Д.А.

Преподаватель Размочаева Н.В.

Санкт-Петербург

2019

Вариант 1.

Цель работы.

Построение и анализ алгоритма Форда-Фалкерсона на основе на решения задачи о нахождении максимального потока в сети.

Основные теоретические положения.

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).

Описание алгоритма.

Для решения поставленной задачи был реализован класс Network, содержащий в себе следующие методы – findWay, findMin, modifyFlow, modifyCapacity. findWay – функция, находящая при помощи поиска в ширину любой путь от истока до стока. Хранение пути осуществляется при помощи хранения имени предыдущей вершины. findMin – функция, осуществляющая поиск ребра с минимальной пропускной способностью. modifyFlow – функция, увеличивающая и уменьшающая (для противоположных ребер) величину потока в каждом ребре, входящем в состав ранее найденного пути. modifyCapacity – функция, осуществляющая пересчет пропускной способности каждого ребра, входящего в ранее найденный путь, на основе потока, протекающего в нем. Алгоритм заканчивает свою работу, когда пути от истока в сток не было найдено. Сложность алгоритма O(|V|*$|E|^2$).

Вывод промежуточной информации.

Во время основной части работы алгоритма происходит вывод промежуточной информации, а именно, выбранная на данном шаге вершина (поиск в ширину), величину минимальной пропускной способности для данного пути, поток в ребрах, входящих в состав ранее найденного пути.

Тестирование.

Таблица 1 – Результаты 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 20 b c 20 c d 20 a c 1 b d 1	21 a b 20 a c 1 b c 19 b d 1 c d 20
9 a d a b 8 b c 10 c d 10 h c 10 e f 8 g h 11	18 a b 8 a g 10 b c 0 b e 8 c d 10 e f 8 f d 8

b e 8 a g 10 f d 8	g h 10 h c 10
16 a e a b 20 b a 20 a d 10 d a 10 a c 30 c a 30 b c 40 c b 40 c d 10 d c 10 c e 20 e c 20 b e 30 e b 30 d e 10 e d 10	60 a b 20 a c 30 a d 10 b a 0 b c 0 b e 30 c a 0 c b 10 c d 0 c e 20 d a 0 d c 0 d e 10 e b 0 e c 0 e d 0

Вывод.

В ходе работы был построен и анализирован алгоритм Форда-Фалкерсона на основе решения задачи о нахождении максимального потока в сети. Исходный код программы представлен в приложении А.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <queue>
#include <map>
#include <stack>

struct Network
{
public:
    struct WayInfo
    {
        char to;
        int capacity;
        int flow;
    };

    struct VertexInfo
    {
        char prev = 0;
        bool is_used = false;
        std::vector<WayInfo> ways;
    };

private:
    char start = 0;
    char end = 0;
    size_t min_capacity = 0;

    std::map<char, VertexInfo> d;

public:
    Network() = default;
    Network(Network const& other)
    {
        start = other.start;
        end = other.end;
        d = other.d;
    }

    void addElem(char vertexName, WayInfo way)
    {
        d[vertexName].ways.push_back(way);
    }

    void setEnds(char start, char end)
    {
        this->start = start;
        this->end = end;
    }

    bool findWay()
    {
        std::queue<char> q;
        q.push(start);

        while (!q.empty())
        {
```

```
            char cur = q.front();
            q.pop();

            d[cur].is_used = true;

            for (auto& next : d[cur].ways)
            {
                if (d[next.to].is_used == true)
                    continue;

                if (next.capacity <= 0)
                    continue;

                if (next.to == end)
                {
                    d[next.to].prev = cur;
                    return true;
                }

                q.push(next.to);
                d[next.to].prev = cur;
            }
        }

        return false;
    }

    void findMin()
    {
        char cur = end;
        while (cur != start)
        {
            char prev = d[cur].prev;
            for (auto& way : d[prev].ways)
            {
                if (way.to == cur)
                {
                    if (min_capacity == 0 || min_capacity > way.capacity)
                        min_capacity = way.capacity;
                    break;
                }
            }

            cur = prev;
        }
    }

    void modifyFlow()
    {
        char cur = end;
        while (cur != start)
        {
            char prev = d[cur].prev;
            for (auto& way : d[prev].ways)
            {
                if (way.to == cur)
                {
                    way.flow += min_capacity;
                    break;
                }
            }

            for (auto& reverse_way : d[cur].ways)
```

```
            {
                if (reverse_way.to == prev)
                {
                    reverse_way.flow -= min_capacity;
                    break;
                }
            }

            cur = prev;
        }
    }

    void modifyCapacity()
    {
        char cur = end;
        while (cur != start)
        {
            char prev = d[cur].prev;
            for (auto& way : d[prev].ways)
            {
                if (way.to == cur)
                {
                    way.capacity -= way.flow;
                    break;
                }
            }

            for (auto& reverse_way : d[cur].ways)
            {
                if (reverse_way.to == prev)
                {
                    reverse_way.capacity -= reverse_way.flow;
                    break;
                }
            }

            cur = prev;
        }

        min_capacity = 0;

        for (auto& i : d)
            i.second.is_used = false;
    }

    friend std::ostream& operator<<(std::ostream& out, Network net)
    {
        net.sort();

        size_t res_flow = 0;
        for (auto& start_way : net.d[net.start].ways)
            res_flow += std::max(0, start_way.flow);
        std::cout << res_flow << "\n";

        for (auto& elem : net.d)
        {
            for (auto& way : elem.second.ways)
                out << elem.first << " " << way.to << " " << std::max(0, way.flow)
<< "\n";
        }

        return out;
    }
```

```
    void sort()
    {
        for (auto& i : d)
        {
            std::sort(i.second.ways.begin(), i.second.ways.end(), [](WayInfo i1,
WayInfo i2) {return i1.to < i2.to; });
        }
    }

    void reverse_sort()
    {
        for (auto& i : d)
        {
            std::sort(i.second.ways.begin(), i.second.ways.end(), [](WayInfo i1,
WayInfo i2) {return i1.capacity > i2.capacity; });
        }
    }

};

int main()
{
    Network StartNetwork;

    char start = 0;
    char end = 0;
    int n = 0;

    std::cin >> n >> start >> end;
    StartNetwork.setEnds(start, end);

    char p1 = 0;
    char p2 = 0;
    int len = 0;

    for (int i = 0; i < n; ++i)
    {
        std::cin >> p1 >> p2 >> len;

        Network::WayInfo way = { p2, len, 0 };
        StartNetwork.addElem(p1, way);
    }

    Network CurrentNetwork(StartNetwork);
    CurrentNetwork.reverse_sort();

    while (true)
    {
        bool res = CurrentNetwork.findWay();
        if (res == false)
            break;

        CurrentNetwork.findMin();
        CurrentNetwork.modifyFlow();
        CurrentNetwork.modifyCapacity();
    }

    std::cout << CurrentNetwork;

    return 0;
}
```