

**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: Максимальный поток**

Студентка гр. 9382

\_\_\_\_\_

Сорочина М.В.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2021

### Цель работы.

Изучить принцип поиска максимального потока в сети, а также фактической величины потока, протекающей через каждое ребро, реализовать соответствующую программу.

### 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

$v_i \ v_j \ w_{ij}$  - 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
```

**Вариант.**

Вар. 6. Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, соединяющей вершины, имена которых в алфавите ближе всего друг к другу. Если таких дуг несколько, то выбрать ту, имя конца которой в алфавите ближайшее к началу алфавита.

**Описание функций и структур данных.**

```
1) struct Vertex
{
    char name;
    bool seen;
```

```

        std::pair<int, Vertex *> from;
        std::map<char, std::pair<int, int>> neighbours;
        Vertex()
        {
            seen = 0;
            from = {0, nullptr};
        }
    };

```

Структура для хранения информации о вершине графа. name - имя вершины; seen - флаг, равный 1, если при поиске пути было выбрано ребро с этой вершиной, 0 - при поиске не была использована; from - хранит длину дуги и указатель на вершину, из которой выходит эта дуга, необходима при поиске пути для обозначения из какой вершины попали в рассматриваемую; neighbours - карта для хранения смежных вершин, где ключ имя вершины, а значение - величина потока.

```
2) bool cmp(Vertex *a, Vertex *b);
```

Функция сортировки вершин в графе в лексикографическом порядке. \*a и \*b - указатели на сравниваемые вершины.

```
3) void input(int numberOfEdges, std::vector<Vertex *> &graph,
std::vector<std::pair<char, char>> &edges)
```

Функция записи ввода, а также инициализации векторов, хранящих информацию о вершинах и ребрах. numberOfEdges - число ребер; graph - вектор указателей на вершины; edges - введенные ребра графа.

```
4) bool isEdge(std::vector<std::pair<char, char>> edges, char
ver1, char ver2)
```

Функция проверки было ли ребро дано изначально. edges - введенные ребра графа; ver1 - имя вершины, предположительное начало ребра; ver2 - имя вершины, предположительный конец ребра. Возвращает 1, если такое ребро есть, и 0, если нет.

```
5) void answer(std::vector<Vertex *> graph, char from,
std::vector<std::pair<char, char>> &edges)
```

Функция подсчета максимального потока и вывода ответа. graph - вектор указателей на вершины; from - имя истока; edges - введенные ребра графа.

```
6)std::pair<Vertex *, Vertex *> chooseVer(std::vector<Vertex  
*> graph, std::vector<char> vertices)
```

Функция выбора очередного ребра. graph - вектор указателей на вершины; vertices - вектор вершин, которые уже были посещены при поиске актуального пути. Возвращает указатель на вершины, находящиеся на концах выбранной дуги.

```
7)Vertex *retVer(char name, std::vector<Vertex *> graph)
```

Функция поиска вершины в графе, возвращает указатель на вершину, если она есть в графе, и nullptr, если такой вершины нет. name - имя вершины; graph - вектор указателей на вершины.

```
8) bool flow(std::vector<Vertex *> &graph, char start, char  
end)
```

Функция поиска пути в графе. graph - вектор указателей на вершины; start и end - имена истока и стока соответственно. Возвращает 1, если путь был найден, 0, если в сети больше нет пути от истока к стоку.

```
9)int minCapacity(char end, std::vector<Vertex *> graph)
```

Функция подсчета пропускной способности найденного ранее пути, ее же и возвращает. end - имя стока; graph - вектор указателей на вершины.

```
10)void recount(int min, char end, std::vector<Vertex *>  
&graph)
```

Функция пересчета остаточных пропускных способностей. min - пропускная способность пути; end - имя стока; graph - вектор указателей на вершины.

```
11)void clearFrom(std::vector<Vertex *> &graph)
```

Функция очистки меток, проставленных во время поиска пути. graph - вектор указателей на вершины.

12)void maxFlow(std::vector<Vertex \*> &graph, char start, char end)

Функция поиска максимального потока в сети и фактической величины потока, протекающего через каждое ребро. graph - вектор указателей на вершины; start и end - имена истока и стока соответственно.

13)Для графа была использована структура данных std::vector<Vertex \*>. Вектор хранит указатели на все вершины. Каждая вершина хранит информацию:

- имя,
- была ли она использована во время поиска пути,
- метку число-символ для отметки по какому ребру пришли в вершину, если она была рассмотрена при поиске очередного пути,
- словарь соседей, содержащий имя и длину ребра до соседа, а также еще одно число, изначально равное нулю, для хранения информации о фактической пропускной способности дуги.

### **Описание алгоритма.**

Пока в сети можно найти путь, происходит поиск пути, подсчет его пропускной способности, пересчет остаточных пропускных способностей. Когда не останется путей из истока в сток, поиск прекращается, считается максимальный поток в графе, и выводится результат. Максимальный поток считается по значениям дуг, исходящих из истока.

При построении пути выбор ребра осуществляется по принципу: выбирается ребро, соединяющее вершины, имена которых находятся ближе всего друг к другу, в случае, когда таких ребер несколько, выбирается то, имя конца которого ближе к началу алфавита. Вершины, в которые попадали дуги

при поиске пути, записываются, потом при выборе нового ребра просматриваются все смежные вершины для каждой из них.

При расчете пропускной способности пути, рассматриваются все ребра, участвующие в пути, и выбирается наименьшая длина. Это значение и будет пропускной способностью рассматриваемого пути.

При пересчете пропускных способностей, рассматриваются все ребра, участвующие в пути, из длины каждого ребра вычитается пропускная способность пути, а к длине обратного ребра добавляется это же значение.

### **Оценка сложности.**

Обозначения:  $V$  - количество вершин,  $F$  - максимальный поток,  $E$  - количество ребер.

По памяти  $O(V+E)$ , так как хранится информация как о вершинах, так и о ребрах.

По времени  $O(F*V)$ , так как максимум нужно будет искать путь  $F$  раз и рассматривать все ребра.

### **Тестирование.**

| № теста | Ввод   | Вывод  |
|---------|--|--|
| 1       | 5<br>a<br>d<br>a b 2<br>a c 4<br>b c 5<br>a d 3<br>c d 1 | 4<br>a b 1<br>a c 0<br>a d 3<br>b c 1<br>c d 1 |
| 2       | 7<br>a<br>c  | 5<br>a d 3<br>a e 2                            |

|   |  |  |
|---|--|--|
|   | b a 2<br>a d 3<br>d b 1<br>d c 7<br>a e 5<br>e d 2<br>b e 10   | b a 0<br>b e 0<br>d b 0<br>d c 5<br>e d 2  |
| 3 | 7<br>a<br>e<br>a b 5<br>a d 4<br>b c 5<br>d c 7<br>c d 7<br>d e 4<br>c e 8                             | 9<br>a b 5<br>a d 4<br>b c 5<br>c d 0<br>c e 5<br>d c 0<br>d e 4                             |
| 4 | 10<br>b<br>f<br>a c 5<br>d e 7<br>b g 7<br>g f 4<br>c b 5<br>a d 3<br>d b 4<br>g a 6<br>b f 8<br>c f 2 | 14<br>a c 2<br>a d 0<br>b f 8<br>b g 6<br>c b 0<br>c f 2<br>d b 0<br>d e 0<br>g a 2<br>g f 4 |
| 5 | 10<br>a<br>e<br>a b 5<br>a f 7<br>b d 9<br>d c 11  | 12<br>a b 5<br>a f 7<br>b d 5<br>b e 0<br>c d 0<br>c e 12                                    |



|   |  |                              |
|---|--|------------------------------|
|   | d a 13<br>b e 15<br>f c 17<br>c e 19<br>b d 21<br>c d 23 | d a 0<br>d c 5<br>f c 7      |
| 6 | 3<br>a<br>b<br>a b 2<br>a c 3<br>c b 1                   | 3<br>a b 2<br>a c 1<br>c b 1 |

### **Выводы.**

В ходе выполнения работы была написана программа, реализующая поиск максимального потока в сети и вычисление фактического потока, протекающего через каждое ребро.

## ПРИЛОЖЕНИЕ А.

### ИСХОДНЫЙ КОД ПРОГРАММЫ lab3.cpp.

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

#define COMMENTS
#define PATH

struct Vertex //структура для хранения информации о вершине
{
    char name; //имя вершины
    bool seen; //флаг использования
    при поиске пути
    std::pair<int, Vertex *> from; //из какой вершины
    пришли и длина дуги
    std::map<char, std::pair<int, int>> neighbours; //хранит информацию о
    смежных вершинах
    Vertex()
    {
        seen = 0;
        from = {0, nullptr};
    }
};

bool cmp(Vertex *a, Vertex *b);
// для сортировки вершин в графе
void input(int numberOfEdges, std::vector<Vertex *> &graph,
std::vector<std::pair<char, char>> &edges);
//функция ввода и инициализации массива ребер и вершин
bool isEdge(std::vector<std::pair<char, char>> edges, char ver1, char
ver2);
//проверка было ли дано такое ребро
void answer(std::vector<Vertex *> graph, char from,
std::vector<std::pair<char, char>> &edges);
//функция вывода ответа
std::pair<Vertex *, Vertex *> chooseVer(std::vector<Vertex *> graph,
std::vector<char> vertices);
//функция выбора очередной вершины
Vertex *retVer(char name, std::vector<Vertex *> graph);
//функция поиска вершины в графе
bool flow(std::vector<Vertex *> &graph, char start, char end);
//функция поиска очередного пути в графе
int minCapacity(char end, std::vector<Vertex *> graph);
//функция нахождения пропускной способности для данного пути
void recount(int min, char end, std::vector<Vertex *> &graph);
//функция пересчета остаточных пропускных способностей
void clearFrom(std::vector<Vertex *> &graph);
//очиста меток, проставленных во время поиска пути
void maxFlow(std::vector<Vertex *> &graph, char start, char end);
```

```

//функция поиска максимального потока

int main()
{
    int numberOfEdges;
    char start, end;
    std::cin >> numberOfEdges >> start >> end;
    std::vector<Vertex *> graph;
    std::vector<std::pair<char, char>> edges;

    input(numberOfEdges, graph, edges);
    std::sort(graph.begin(), graph.end(), cmp);
    maxFlow(graph, start, end);
    answer(graph, start, edges);
    return 0;
}

bool cmp(Vertex *a, Vertex *b)
// для сортировки в графе
{
    return a->name < b->name;
}

void input(int numberOfEdges, std::vector<Vertex *> &graph,
std::vector<std::pair<char, char>> &edges)
//функция ввода и инициализации массива ребер и вершин
{
    char from, to;
    int len;
    Vertex *ver;
    for (int i = 0; i < numberOfEdges; i++)
    {
        std::cin >> from >> to >> len;
        edges.push_back({from, to});
        ver = retVer(from, graph);
        if (ver != nullptr) //если вершина есть в графе
        {
            ver->neighbours[to] = {len, 0}; //то добавляется смежная
вершина к списку смежных
        }
        else
        { //иначе добавляется новая вершина в граф
            ver = new Vertex;
            ver->name = from;
            ver->neighbours[to] = {len, 0};
            graph.push_back(ver);
        }
        ver = retVer(to, graph); //для обратного ребра
        if (!isEdge(edges, to, from)) //если ребра еще не было
        { //то либо добавляем саму
вершину(если ее не было), либо смежную ей
            if (ver == nullptr)
            {

```

```

        ver = new Vertex;
        ver->name = to;
        ver->neighbours[from] = {0, 0};
        graph.push_back(ver);
    }
    else
    {
        ver->neighbours[from] = {0, 0};
    }
}
}
}

bool isEdge(std::vector<std::pair<char, char>> edges, char ver1, char ver2)
//проверка было ли дано такое ребро
{
    for (auto i : edges)
    {
        if (i.first == ver1 && i.second == ver2)
        {
            return 1;
        }
    }
    return 0;
}

void answer(std::vector<Vertex *> graph, char from,
std::vector<std::pair<char, char>> &edges)
//функция вывода ответа
{
    auto start = retVer(from, graph);
    int max = 0;
    for (auto i : start->neighbours)
    {
        max += i.second.second;
    }
    std::cout << max << '\n';
    for (auto ver : graph)
    {
        for (auto neib : ver->neighbours)
        {
            if (isEdge(edges, ver->name, neib.first))
            {
                if (neib.second.second > 0)
                {
                    std::cout << ver->name << " " << neib.first << " " <<
neib.second.second << '\n';
                }
                else
                {
                    std::cout << ver->name << " " << neib.first << "
0\n";
                }
            }
        }
    }
}

```

```

    }
    }
}

std::pair<Vertex *, Vertex *> chooseVer(std::vector<Vertex *> graph,
std::vector<char> vertices)
//функция выбора очередной вершины
{
    int min = 26, check;
    Vertex *minV;
    Vertex *prev;
    Vertex *ver;
#ifdef COMMENTS
    std::cout << "\t\t\tВыбираем новую дугу\n";
#endif
    for (auto name : vertices)
    {
        ver = retVer(name, graph);
        for (auto neib : ver->neighbours)
        {
            if (retVer(neib.first, graph)->seen == 1) //если в вершину
уже "заходили", то пропускаем
            {
                continue;
            }
            check = abs(neib.first - name); //расстояние именами вершин
            if ((check < min || check == min && neib.first < minV->name)
&& neib.second.first > 0)
            {
                prev = ver;
                min = check;
                minV = retVer(neib.first, graph);
#ifdef COMMENTS
                std::cout << "\t\t\tНовая дуга выбрана: [" <<
prev->name << ", " << neib.first << "]\n";
#endif
            }
        }
    }
    if (min == 26)
    {
        return {nullptr, nullptr};
    }
    return {minV, prev};
}

Vertex *retVer(char name, std::vector<Vertex *> graph)
//функция поиска вершины в графе
{
    for (auto i : graph)
    {

```

```

        if (i->name == name)
        {
            return i;
        }
    }
    return nullptr;
}

bool flow(std::vector<Vertex *> &graph, char start, char end)
//функция поиска очередного пути в графе
{
    std::vector<char> vertices; //для хранения пройденных вершин
    vertices.push_back(start);
    auto ver = retVer(start, graph);
    ver->seen = 1;
#ifdef COMMENTS
    std::cout << "\t\tПоиск пути:\n";
#endif
    while (ver->name != end)
    {
#ifdef COMMENTS
        std::cout << "\t\tУже использованные вершины:\n\t\t\t[";
        for (auto q : vertices)
        {
            std::cout << q << " ";
        }
        std::cout << "]\n";
#endif
        auto newVers = chooseVer(graph, vertices); //выбор следующего
        auto newV = newVers.first;
        if (newV == nullptr)
        {
            return 0; //если потоков больше нет
        }
#ifdef COMMENTS
        std::cout << "\t\t\tВыбранная вершина: [" << newV->name << "]\n";
#endif
        auto prev = newVers.second;
        vertices.push_back(newV->name);
        newV->seen = 1;
        newV->from = {prev->neighbours[newV->name].first, prev};
        ver = newV;
    }
    return 1; //если дуга нашлась, то данные о вершинах в графе будут
    изменены, и можно будет восстановить путь
}

int minCapacity(char end, std::vector<Vertex *> graph)
//функция нахождения пропускной способности для данного пути
{
    auto ver = retVer(end, graph);
    int min = ver->from.first;
    while (ver->from.second != nullptr) //пока не дойдем до начала

```

```

    {
        if (min > ver->from.first)
        {
            min = ver->from.first;
        }
        ver = ver->from.second;
    }
    return min;
}

void recount(int min, char end, std::vector<Vertex *> &graph)
//функция пересчета остаточных пропускных способностей
{
    auto ver = retVer(end, graph)->from.second;
    char prev = end;
#ifdef PATH
    std::cout << "\t\tПропускная способность данного пути: " << min <<
    "\n";
    std::vector<char> path;
#endif
#ifdef COMMENTS
    std::cout << "\t\tНачало пересчета пропускных способностей:\n";
#endif
    while (ver->from.second != nullptr) //пока не дойдем до начала
    {
#ifdef PATH
        path.push_back(ver->name);
#endif
#ifdef COMMENTS
        std::cout << "\t\t\tИз пропускной способности ребра [" <<
ver->name << ", " << prev << "] было вычтено " << min << "\n";
        std::cout << "\t\t\tК пропускной способности ребра [" << prev <<
", " << ver->name << "] было прибавлено " << min << "\n\n";
#endif
        ver->neighbours[prev].first -= min; //вычитаем из использованного
ребра
        ver->neighbours[prev].second += min;
        auto prevVer = retVer(prev, graph);
        prevVer->neighbours[ver->name].first += min; //добавляем к
обратному ребру
        prevVer->neighbours[ver->name].second -= min;
        prev = ver->name;
        ver = ver->from.second;
    }
#ifdef COMMENTS
    std::cout << "\t\t\tИз пропускной способности ребра [" << ver->name
<< ", " << prev << "] было вычтено " << min << "\n";
    std::cout << "\t\tПересчет пропускных способностей окончен\n";
#endif
#ifdef PATH
    path.push_back(ver->name);
    std::cout << "\t\tПуть:\n\t\t";
    for (char i = path.size() - 1; i >= 0; i--)

```

```

        {
            std::cout << path[i];
        }
        std::cout << end;
    #endif
    ver->neighbours[prev].first -= min;
    ver->neighbours[prev].second += min;
    #ifdef COMMENTS
        std::cout << "\n\n";
    #endif
}

void clearFrom(std::vector<Vertex *> &graph)
//очиста меток, проставленных во время поиска пути
{
    for (auto ver : graph)
    {
        ver->from = {0, nullptr};
        ver->seen = 0;
    }
}

void maxFlow(std::vector<Vertex *> &graph, char start, char end)
//функция поиска максимального потока
{
    std::vector<char> vertices; //для хранения пройденных вершин
    int min;
    #ifdef COMMENTS
        std::cout << "Начат поиск макс. потока\n";
    #endif
    while (flow(graph, start, end))
    {
        //для поиск пропускной способности найденного пути
        min = minCapacity(end, graph);
    #ifdef COMMENTS
        std::cout << "\tПропускная способность посчитана\n";
    #endif
        //пересчет остаточных пропускных способностей
        recount(min, end, graph);
    #ifdef COMMENTS
        std::cout << "\tПересчет пропускных способностей выполнен\n";
    #endif
        //очистка from'ов
        clearFrom(graph);
    #ifdef COMMENTS
        std::cout << "\tОчистка меток
        выполнена\n-----\n";
    #endif
    }
    #ifdef COMMENTS
        std::cout << "Конец! Макс. поток найден\n";
    #endif
}

```



