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. 8383	____________________	Костарев К.В.
Преподаватель	____________________	Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с принципами работы алгоритма Форда-Фалкерсона нахождения максимального потока в сети и реализовать так, как требуется в индивидуальном варианте.

Задача.

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 ребер графа

u_0 – исток

u_n – сток

u_i u_j w_{ij} – ребро графа

u_i u_j w_{ij} – ребро графа

…

Выходные данные:

Pmax – величина максимального потока

u_i u_j w_{ij} – ребро графа с фактической величиной протекающего потока

u_i u_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 № 4.

Поиск в глубину. Итеративная реализация.

Основные теоретические сведения.

Сеть – ориентированный взвешенный граф, имеющий один исток и один сток.

Исток – вершина, из которой рёбра только выходят.

Сток – вершина, в которую рёбра только входят.

Поток – абстрактное понятие, показывающее движение по графу.

Величина потока – числовая характеристика движения по графу (сколько всего выходит из истока = сколько всего входит в сток).

Пропускная способность – свойство ребра, показывающее, какая максимальная величина потока может пройти через это ребро.

Максимальный поток (максимальная величина потока) – максимальная величина, которая может быть выпущена из истока, которая может пройти через все рёбра графа, не вызывая переполнения ни в одном ребре.

Фактическая величина потока в ребре – значение, показывающее, сколько величины потока проходит через это ребро.

Описание алгоритма.

Изначально величине фактического потока для каждого ребра присваивается нулевое значение. Далее в графе начинается поиск увеличивающего пути (пути от истока до стока, где значение пропускной способности каждого ребра в пути больше 0). В данной лабораторной работе поиск осуществляется с помощью обхода графа в глубину итеративным методом: на каждой итерации алгоритм обходит одно ребро графа, и если оно не было пройдено и его пропускная способность больше нуля, то сравнивает его пропускную способность с минимальной в уже пройденном пути и в случае, если его пропускная способность еще меньше, то обновляет значение и следующая итерация начинается с конечной вершины ребра; если ребро уже пройдено или его пропускная способность нулевая, то следующая итерация начинается с другого ребра из соседних, если таких ребер нет, то следующая итерация начинается с предыдущего ребра в уже пройденном пути.

Если ребро ведет к стоку, то значит путь найден и значение фактического потока в каждом ребре уже пройденного пути увеличивается на величину минимальной пропускной способности, а у ребер, противоположных ребрам пути, фактический поток уменьшается на эту величину. Следующая итерация начинается с предыдущего ребра в пройденном пути.

Если список вершин пройденного пути оказался пустым, значит все пути найдены и алгоритм заканчивает работу (список вершин пустой, следовательно из списка был удален исток, следовательно все вершины из истока уже пройдены). Максимальная величина потока считается как сумма потоков в ребрах, выходящих из истока или входящих в сток.

Описание структур данных.

1. class Edge. Класс, необходимый для хранения информации об одном ребре графа. Имеет поля:
   - char a, b – начальная и конечная вершины ребра;
   - int p – фактическая величина потока;

- int w – пропускная способность (вес);

2. class FordFalkerson. Класс для хранения графа и с реализацией алгоритма:
   - char s, t – исток и сток сети;
   - vector <Edge*> graph – вектор, который хранит все ребра между вершинами (представление графа);
   - vector <Edge*> path – уже пройденный путь как стек ребер;
   - string pathInChar – уже пройденный путь как стек вершин;
   - vector<string> visitedPaths – вектор путей, которые уже были пройдены обходом в глубину;
   - int wMin – минимальная пропускная способность на пройденном пути path;
   - int pMax – максимальная величина потока;
   - char u – вершина, которую обходит алгоритм на каждой итерации;
   - bool isVisitedV(char v) – функция, возвращающая true, если вершина v уже была пройдена (есть в стеке pathInChar), и наоборот;
   - bool isVisited(string p) – функция, возвращающая true, если путь p уже был пройден (есть в стеке visitedPaths), и наоборот;
   - bool haveInvert(Edge* e) – функция, возвращающая true, если у ребра e есть противоположное ребро в графе graph, и наоборот;
   - void execute() – собственно реализация описанного выше алгоритма, который до начала цикла в качестве u ставит исток графа, и пушает его на стек вершин pathInChar, и при каждой итерации проверяет одно ребро из графа graph, ведущее из вершины u, и если оно удовлетворяет условиям, заносит ребро в path, path пушает в visitedPaths, а конечную вершину ребра пушает в pathInChar (т.е обновляет пройденный путь). Окончание цикла – когда в pathInChar не останется ничего (т.е. исток был удален, а следовательно все пути из истока пройдены).

Сложность алгоритма по времени.

Так как в цикле while поиск в глубину может в худшем случае обойти все ребра и вершины, и при этом на каждой итерации значение потока увеличится как минимум на 1, значит один цикл будет длится не больше, чем сумма пропускных способностей ребер, вышедших из истока (обозначим как m):

$$O((|V|+|E|)\times m).$$

Сложность алгоритма по памяти.

Программа хранит все ребра графа как вектор структур вершин, поэтому по памяти сложность алгоритма составляет:

$$O(|V|).$$

Тестирование.

Демонстрация работы программы приведена на рис. 1. Входные данные для демонстрации:

```
5
a
c
a b 3
a c 6
b d 5
b c 5
a d 2
```

```
Начинаем с истока a
   a->b (0/3):  минимальная пропускная способность теперь 3, пройденный путь ab
   b->d (0/5):  пройденный путь abd
Все подвершины пройдены, поднимаемся, пройденный путь ab
   b->d (0/5):  пропускная способность = 0 или уже пройденный путь, пропускаем
   b->c (0/5):  пройденный путь abc
Дошли до стока, прибавляем  мнимальную пропускную способность 3 у ребер пути abc
Поднимаемся выше, пройденный путь ab
   b->d (0/5):  пропускная способность = 0 или уже пройденный путь, пропускаем
   b->c (3/5):  пропускная способность = 0 или уже пройденный путь, пропускаем
Все подвершины пройдены, поднимаемся, пройденный путь a
   a->b (3/3):  пропускная способность = 0 или уже пройденный путь, пропускаем
   a->c (0/6):  минимальная пропускная способность теперь 6, пройденный путь ac
Дошли до стока, прибавляем  мнимальную пропускную способность 6 у ребер пути ac
Поднимаемся выше, пройденный путь a
   a->b (3/3):  пропускная способность = 0 или уже пройденный путь, пропускаем
   a->c (6/6):  пропускная способность = 0 или уже пройденный путь, пропускаем
   a->d (0/2):  минимальная пропускная способность теперь 2, пройденный путь ad
   d->b (0/0):  пропускная способность = 0 или уже пройденный путь, пропускаем
Все подвершины пройдены, поднимаемся, пройденный путь a
   a->b (3/3):  пропускная способность = 0 или уже пройденный путь, пропускаем
   a->c (6/6):  пропускная способность = 0 или уже пройденный путь, пропускаем
   a->d (0/2):  пропускная способность = 0 или уже пройденный путь, пропускаем
Весь граф пройден!
9
a b 3
a c 6
a d 0
b c 3
b d 0
```

Рисунок 1 – Демонстрация работы алгоритма

Тестирование программы приведено в табл. 1. В ходе тестирования все выходные данные оказались корректными.

Таблица 1 – Тестирование программы

Входные данные	Выходные данные
7 a c a b 3 b c 2 a c 5 b d 8 a d 6 d e 4 e c 1	8 a b 3 a c 5 a d 0 b c 2 b d 1 d e 1 e c 1
7 a f a b 7	12 a b 6 a c 6 b d 6

a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	c f 8 d e 2 d f 4 e c 2
7 a e a b 5 b c 4 c d 6 d e 1 a e 3 b d 2 a d 2	4 a b 1 a d 0 a e 3 b c 1 b d 0 c d 1 d e 1
7 a e a b 7 b c 5 c d 8 d b 10 c e 9 d e 5 b e 8	7 a b 7 b c 5 b e 2 c d 5 c e 0 d b 0 d e 5
2 a c a b 4 c b 1	0 a b 0 c b 0
7 a d a b 100 b d 100 d b 60 b c 70 c b 100 c d 100 d c 50	100 a b 100 b c 0 b d 100 c b 0 c d 0 d b 0 d c 0

Выводы.

В данной лабораторной работе был изучен алгоритм Форда-Фалкерсона нахождения максимального потока в сети. Было установлено, что посредством нахождения увеличивающего пути методом обхода сети в глубину, сложность алгоритма по памяти и времени линейна.

ПРИЛОЖЕНИЕ А

КОД ПРОГРАММЫ РЕАЛИЗАЦИИ АЛГОРИТМА

```
#include <iostream>
#include <vector>
#include <string>
#include <windows.h>

class Edge{     //класс ребра
public:
    char a{};   //начальная вершина
    char b{};   //конечная
    int p;  //фактическая величина потока
    int w;  //пропускная способность (вес)
    Edge(){
        p = 0;
        w = 0;
    }
    Edge(char a, char b){
        this->a = a;
        this->b = b;
        p = 0;
        w = 0;
    }
};

class FordFalkerson{    //класс алгоритма
public:
    std::vector<Edge*> graph;   //граф для обработки
    std::vector<Edge*> path;    //пройденный путь в виде стека ребер
    std::string pathInChar;     //пройденный путь в виде стека вершин
    std::vector<std::string> visitedPaths;  //пройденные пути
    int wMin = 1000;    //минимальная пропускная сп.
    int pMax{}; //максимальный поток
    char s{};   //исток
    char t{};   //сток
    char u{};   //проходдимая вершина
    FordFalkerson() = default;
    bool isVisitedV(char v){    //является ли вершина уже пройденной
        return pathInChar.find(v) != std::string::npos;
    }
    bool isVisited(std::string p){  //является ли путь уже пройденным
        for (auto & i:visitedPaths){
            if (i == p)
                return true;
        }
        return false;
    }
    bool haveInvert(Edge* e){   //есть ли в графе противоположное от е
ребро
        for (auto &i:graph){
            if (e->a == i->b && e->b == i->a)
                return true;
        }
        return false;
    }
    void execute(){ //выполнение алгоритма
```

```
        int sze = graph.size();
        for (int i = 0; i < sze; i++){  //добавление к графу
противоположных ребер
            if (!haveInvert(graph[i])){
                auto edgeInvert = new Edge(graph[i]->b, graph[i]->a);
                graph.push_back(edgeInvert);
            }
        }
        u = s;  //начало с истока
        pathInChar.push_back(u);    //пушим в стек
        std::cout << "Начинаем с истока " << s << std::endl;
        while (true){
            bool noChild = true;
            for(auto& i:graph){
                if (i->a == u && !isVisitedV(i->b)){
                    std::cout << "   " << i->a << "->" << i->b << " (" <<
i->p << "/" << i->w << "):\t";
                    pathInChar.push_back(i->b);
                    if (i->p < i->w && i->b != s &&
!isVisited(pathInChar)){    //если ребро не пройдено и не с нулевой
пропускной
                        if (i->w - i->p < wMin){
                            wMin = i->w - i->p;
                            std::cout << "минимальная пропускная
способность теперь " << wMin << ", ";
                        } //если пропускная сп. минимальная, обновляем
значение
                        u = i->b;   //конечная вершина ребра становится
текущей
                        path.push_back(i);  //пушим ребро в стек
                        visitedPaths.push_back(pathInChar); //пушим
вершину в стек
                        std::cout << "пройденный путь " << pathInChar <<
std::endl;
                        noChild = false;
                        break;
                    }
                    std::cout << "пропускная способность = 0 или уже
пройденный путь, пропускаем" << std::endl;
                    pathInChar.pop_back();
                }
            }
            if (noChild){   //если нет непройденных ребер, возвращаемся к
предыдущему
                pathInChar.pop_back();
                if (pathInChar.empty()){ //если непройденных не осталось,
заканчиваем цикл
                    std::cout << "Весь граф пройден!" << std::endl;
                    break;
                } else{
                    std::cout << "Все подвершины пройдены, поднимаемся,
пройденный путь " << pathInChar << std::endl;
                }
                u = path.back()->a;
                path.pop_back();
                wMin = 1000;
```

```
                for(auto& i:path){  //обновляем минимальную пропускную 
сп. без учета удаленного ребра
                    if (i->w - i->p < wMin)
                        wMin = i->w - i->p;
                }
            }
            if (u == t){    //если вершина сток, возвращаемся к 
предыдущей в стеке
                std::cout << "Дошли до стока, прибавляем  мнимальную 
пропускную способность " << wMin << " у ребер пути " << pathInChar << 
std::endl;
                pathInChar.pop_back();
                std::cout << "Поднимаемся выше, пройденный путь " << 
pathInChar << std::endl;
                for(auto& i:path){  //добавляем минимальную пропускную к 
ребрам пути и отнимаем у противоположных
                    i->p += wMin;
                    for (auto & j:graph) {
                        if ((i->a == j->b) && (i->b == j->a))
                            j->p -= wMin;
                    }
                }
                u = path.back()->a;
                path.pop_back();
                wMin = 1000;
                for(auto& i:path){ //обновляем минимальную пропускную сп. 
без учета удаленного ребра
                    if (i->w - i->p < wMin)
                        wMin = i->w - i->p;
                }
            }
        }
        for (auto&i:graph){ //считаем максимальный поток
            if (i->a == s){
                pMax += i->p;
            }
        }
    }
    void printAnswer(){     //печать ответа в требуемом виде
        std::cout << pMax << std::endl;
        for (auto&i:graph){
            for (auto&j:graph){
                if ((j->a > i->a) || (j->a == i->a && j->b > i->b)){
                    auto k = new Edge();
                    k = j;
                    j = i;
                    i = k;
                }
            }
        }
        for (auto&i:graph){
            if (i->w > 0) {
                if (i->p < 0)
                    i->p = 0;
                std::cout << i->a << " " << i->b << " " << i->p 
<<std::endl;
            }
```

```
        }
    }
};

int main() {
    SetConsoleOutputCP(CP_UTF8);
    auto algorithm = new FordFalkerson();   //инициализация алгоритма
    int n;
    std::cin >> n >> algorithm->s >> algorithm->t;
    for (int i = 0; i < n; i++){    //заполняем граф ребрами
        auto edge = new Edge();
        std::cin >> edge->a >> edge->b >> edge->w;
        algorithm->graph.push_back(edge);
    }
    algorithm->execute();   //алгоритм
    algorithm->printAnswer();   //печать результатов
    return 0;
}
```