

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

Студентка гр. 8382

Кузина А.М.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

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 ребер 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
```

Smple Output:

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

Вариант.

Вариант 5. Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность. Если таких дуг несколько, то выбрать ту, которая была обнаружена раньше в текущем поиске пути.

Описание алгоритма.

На вход программе вначале подаются количество ребер графа и две вершины: исток и сток. Далее подаются наборы данных: начальная и конечная вершины ребра и пропускная способность(вес) ребра. На основе входных данных формируются матрица ребер графа, массив вершин.

После окончания считывания входных данных начинается процесс поиска максимального потока в графе с помощью алгоритма Форда-Фалкерсона. Для работы алгоритма также добавлен двойной массив ребер, который вначале работы алгоритма соответствует заданному графу. Алгоритм выполняется следующим образом:

1. В дополнительном графе производится поиск пути из истока в сток. Следующим в пути выбирается ребро, имеющее максимальную остаточную пропускную способность. Также для пути вычисляется значение веса пути — равное наименьшему весу ребра, входящего в путь.
2. В построенном пути, для каждого входящего в него ребра, вес ребра уменьшается на значение веса пути. Но также для ребра в обратном направлении, пропускная способность увеличивается на вес найденного пути. Теперь оно также доступно для построения путей в графе.
3. Значение максимального потока в графе увеличивается на полученное значение веса пути.
4. Работа алгоритма продолжается до тех пор, пока в графе можно найти хотя бы один путь из истока в сток.

По итогу работы алгоритма сумма весов ребер выходящих из стока будет равна сумме весов ребер входящих в сток, а также вычисленному алгоритмом значению максимального потока в графе.

Сложность алгоритма.

Для построения одного пути в графе в худшем случае посещаются все вершины и просматриваются все ребра графа. Таких путей будет не более чем величина максимального потока в графе. Соответственно сложность алгоритма по операциям - $O(|E| * f_{\max})$.

Сложность алгоритма по памяти зависит от количества вершин графа т. к. граф представлен в виде матрицы смежности. Следовательно сложность по памяти - $O(|V|^2)$.

Описание структур данных и функций.

Описание класса *graf* — хранение частичных решений:

В *from* и *to* хранятся имена истока и стока. Массив вершин *vertex*, а также двойной массив ребер *edges* хранят основную информацию о заданном графе. Массив *open_vertex* хранит список вершин из которых может быть совершен переход в другие вершины, он очищается в начале каждого нового поиска пути. Двойной массив *flow* хранит временную информацию о путях в графе. Строка *way* представляет найденный путь в графе. Переменная *potok* хранит значение максимального пути в графе.

Метод *print_edges()* выводит на экран двойной массив ребер *edges*.

Метод *init()* считывает и сохраняет входные данные в соответствующие массивы.

Метод *is_new(char a)* возвращает -1, если *a* — новая вершина и *i*, если *a* располагается в массиве вершин на *i*-ой позиции.

Метод *add_vertex(char a)* добавляет вершину *a* в конец списка вершин.

Метод *new_connect(char first, char second, float way)* добавляет ребро длины *way* между вершинами *first* и *second* в массив ребер.

Метод *step_back(int tmp)* при поиске пути удаляет последнее добавленное в путь ребро, если ребро не может привести к стоку.

Метод *zeroing_posible()* делает все вершины открытыми для перехода в них по ребру.

Метод *maxflow(int minflow)* для найденного пути в графе изменяет все необходимые веса ребер.

Метод *alf_egdes_print()* выводит все ребра в алфавитном порядке.

Метод *serch(int temp = 0)* производит основную работу алгоритма — нахождение путей в графе, вызов всех вспомогательных функций, проверка условий завершения работы алгоритма.

Тестирование.

Входные данные	Выходные данные
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
8 a h a d 2 a c 8 c d 16 c f 4 f h 5 g f 18 g h 5 d g 6	10 a c 8 a d 2 c d 4 c f 4 d g 6 f h 5 g f 1 g h 5
10 a e a b 2 b a 2 a c 3 c a 3 b e 4	5 a b 2 a c 3 b a 0 b c 2 b e 3 c a 0

e b 4 c e 2 e c 2 b c 4 c b 4	c b 1 c e 2 e b 0 e c 0
4 1 5 1 2 6 2 5 3 2 3 7 3 4 1	3 1 2 3 2 3 0 2 5 3 3 4 0
5 a e a b 5 b e 1 a c 1 b c 5 c e 5	Add a at 0 Add b at 1 Edge ab added to adge matrices with weight = 5 Add e at 2 Edge be added to adge matrices with weight = 1 Add e at 3 Edge ac added to adge matrices with weight = 1 Edge bc added to adge matrices with weight = 5 Edge ce added to adge matrices with weight = 5 Edges: a b e c a 0 5 0 1 b 0 0 1 5 e 0 0 0 0 c 0 0 5 0 Vertex a is watching now Next vertex will be b Vertex b is watching now Next vertex will be c Vertex c is watching now Next vertex will be e Find path: abce with min weight = 5 Changed weight of adge ce from 5 to 0 Changed weight of adge ec from 0 to 5 Changed weight of adge bc from 5 to 0 Changed weight of adge cb from 0 to 5 Changed weight of adge ab from 5 to 0 Changed weight of adge ba from 0 to 5 Adges after finding this path: Edges: a b e c a 0 0 0 1 b 5 0 1 0 e 0 0 0 5 c 0 5 0 0 Vertex a is watching now Next vertex will be c Vertex c is watching now Next vertex will be b

```

Vertex b is watching now
Next vertex will be e
Find path: acbe with min weight = 1
    Changed weight of adge be from 1 to 0
    Changed weight of adge eb from 0 to 1
    Changed weight of adge cb from 5 to 4
    Changed weight of adge bc from 0 to 1
    Changed weight of adge ac from 1 to 0
    Changed weight of adge ca from 0 to 1
Adges after finding this path:
Edges:
    a b e c
    a 0 0 0 0
    b 5 0 0 1
    e 0 1 0 5
    c 1 4 0 0
Vertex a is watching now
Can't find any more paths. Here is results:
6
a b 5
a c 1
b c 4
b e 1
c e 5

```

Выводы.

В ходе выполнения лабораторной работы была написана программа, находящая в сети максимальный поток с помощью алгоритма Форда-Фалкерсона. При поиске пути следующим дополняющим ребром выбирается ребро с наибольшей остаточной пропускной способностью. Также было проведено тестирование программы с выводом промежуточных результатов на различных тестах.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ФАЙЛА MAIN.CPP

```
#include <iostream>

class graf {
    int edge_count;
    int vertex_count;
    char* vertex;
    char* alf_vertex;
    int** edges;

    int** tmp_flow;
    int** me;
    char from, to;
    int potok;
    std::string way;
    int* posible;

public:
    graf() {
        // Инициализация и обнуление всех полей класса графа
        std::cin >> edge_count;
        vertex_count = 0;
        potok = 0;
        vertex = new char[edge_count * 2];
        alf_vertex = new char[edge_count * 2];
        posible = new int[edge_count * 2];
        way = "";
        me = (int**)malloc(sizeof(int*) * edge_count * 2);
        edges = (int**)malloc(sizeof(int*) * edge_count * 2);

        tmp_flow = (int**)malloc(sizeof(int*) * edge_count * 2);

        for (int i = 0; i < edge_count * 2; i++) {
            vertex[i] = ' ';
            alf_vertex[i] = 'Z';
            me[i] = new int[edge_count * 2];
            edges[i] = new int[edge_count * 2];
            tmp_flow[i] = new int[edge_count * 2];

            posible[i] = 1;
        }

        for (int i = 0; i < edge_count * 2; i++) {
            for (int j = 0; j < edge_count * 2; j++) {
                tmp_flow[i][j] = 0;

                me[i][j] = 0;
                edges[i][j] = 0;
            }
        }
    }

    int is_new(char a) {
        // Возвращает номер вершины, если она не новая и -1 если вершина новая
        int i;
        for (i = 0; i < vertex_count; i++) {
            if (vertex[i] == a) {
                return i;
            }
            if (vertex[i] == ' ') break;
        }
        return -1;
    }
};
```



```

    }
    int add_vertex(char a) {
        // добавляет вершину в список вершин с обычным порядком ввода и список вершин,
        // рассортированный в алфавитном порядке
        int i = 0;
        while (vertex[i] != ' ') i++;
        vertex[i] = a;
        vertex_count++;
        if (vertex_count == 1) {
            alf_vertex[0] = a;
        }
        char c = a;
        for (int j = 0; j < vertex_count; j++) {
            if (a - alf_vertex[j] < 0) {
                c = alf_vertex[j];
                alf_vertex[j] = a;
                a = c;
            }
            else if (j == vertex_count - 1) {
                alf_vertex[j] = a;
            }
        }
        std::cout << "Add " << a << " at " << i << std::endl;
        return i;
    }
    bool new_connect(char first, char second, int way) {
        // Добавление ребра в матрицу ребер
        int a, b;
        a = is_new(first);
        b = is_new(second);
        if (a == -1) a = add_vertex(first);
        if (b == -1) b = add_vertex(second);
        if (a == b) return 0;
        edges[a][b] = way;
        std::cout << "Edge " << first << second << " added to edge matrices with weight = "
        << way << "\n";
        return 1;
    }
    void init() {
        // считывание всех входных данных и запись по соответствующим полям
        std::cin >> from >> to;
        char first, second;
        int way = 0;
        int i = 0;
        while (i < edge_count) {
            std::cin >> first >> second >> way;
            new_connect(first, second, way);
            i++;
        }
        for (int i = 0; i < vertex_count; i++) {
            for (int j = 0; j < vertex_count; j++) {
                tmp_flow[i][j] = edges[i][j];
            }
        }
    }
    void alf_edges_print() {
        // Вывод ребер в алфавитном порядке
        int tmp;
        int to;
        for (int i = 0; i < vertex_count; i++) {
            tmp = is_new(alf_vertex[i]);
            for (int j = 0; j < vertex_count; j++) {
                to = is_new(alf_vertex[j]);
                if (tmp != to && edges[tmp][to] != 0) {

```

```

std::cout << alf_vertex[i] << " " << alf_vertex[j] << " " <<
me[to][tmp] << "\n";
    }
}

}

void print_edges() {
    // вывод матрицы ребер на экран
    std::cout << "Edges:\n ";
    for (int i = 0; i < vertex_count; i++) {
        std::cout << "\t" << vertex[i] << " ";
    }
    std::cout << "\n";

    for (int i = 0; i < vertex_count; i++) {
        std::cout << "\t" << vertex[i] << " ";
        for (int j = 0; j < vertex_count; j++) {
            std::cout << "\t" << tmp_flow[i][j] << " ";
        }
        std::cout << "\n";
    }
}

int step_back(int tmp) {
    //Если поиск пути зашел в тупик и не может найти сток, данная функция позволит
откатиться и найти другой путь
    way.pop_back();
    posible[tmp] = 0; // Запрет прохождения по неудачному ребру

    if (way == "") return -1;
    int prev = is_new(way[way.length() - 1]); //просто получаем номер новой вершины
    tmp_flow[prev][tmp] = 0;
    return prev;
}

void zeroing_posible() {
    //Разрешение прохождения по всем имеющимся ребрам
    for (int i = 0; i < vertex_count; i++) {
        posible[i] = 1;
    }
}

int maxflow(int minflow) {
    // Изменение весов для ребер, попавших в путь от истока до стока
    std::cout << "Find path: " << way << " with min weight = " << minflow << "\n";
    if (way == "") return 0;
    char ctmp = way[way.length() - 1];
    way.pop_back();
    char cprev;
    int tmp = is_new(ctmp);
    int prev;
    while (way != "") { //Пока не обработаем все ребра из представленных в пути
        cprev = way[way.length() - 1];
        way.pop_back();
        prev = is_new(cprev);

        std::cout << "Changed weight of edge " << vertex[prev] << vertex[tmp] << "
from " << tmp_flow[prev][tmp];
        tmp_flow[prev][tmp] = tmp_flow[prev][tmp] - minflow; //Уменьшаем исходное
ребро на вес получившегося потока
        std::cout << " to " << tmp_flow[prev][tmp] << "\n";
        std::cout << "Changed weight of edge " << vertex[tmp] << vertex[prev] << "
to " << tmp_flow[tmp][prev] << "\n";
        tmp_flow[tmp][prev] = tmp_flow[tmp][prev] + minflow; //Увеличиваем вес
обратного ребра на вес потока

```

```

std::cout << " to " << tmp_flow[tmp][prev] << "\n";

me[tmp][prev] += minflow; //это нужно для корректной работы с графами у
которых есть двунаправленные ребра
if(me[prev][tmp] > 0)
    me[prev][tmp] -= minflow;
tmp = prev;
}
}
int serch() {
    int next = 1;
    int min = 0;
    int tmp = is_new(from);
    way = vertex[tmp];
    int prevflow = 9999;
    int tmpflow = 9999;
    int h = 0;
    while (next != -1) {

        min = 0;
        next = -1;
        std::cout << "Vertex " << vertex[tmp] << " is watching now\n";
        for (int i = 0; i < vertex_count; i++) { // обход всех ребер для выбранной
вершины
            if (tmp_flow[tmp][i] > min && posible[i]) {
                // выбор ребра с наибольшим весом
                next = i;
                min = tmp_flow[tmp][i];
            }
        }
        if (next != -1) { // следующая вершина найдена - продолжаем поиск пути
            std::cout << "Next vertex will be " << vertex[next] << "\n";
            posible[tmp] = 0;
            if (min > prevflow) {
                tmpflow = prevflow;
                prevflow = tmpflow;
            }
            else {
                tmpflow = min;
                prevflow = tmpflow;
            }
            tmp = next;
            way.push_back(vertex[next]); // добавление в путь
        }
        if (next == -1) { // больше идти некуда - пробуем откатиться
            tmp = step_back(tmp);
            next = 1;
            if (tmp == -1) { // откат не удался - граф рассмотрен полностью,
больше путей нет
                std::cout << "Can't find any more paths. Here is results:" <<
"\n";

                std::cout << поток << "\n";
                return 0;
            }
            std::cout << "Next vertex will be " << vertex[tmp] << "\n"; // откат
удался, продолжаем поиск пути
        }
        if (next == is_new(to)) { // нашли какой-то путь
            поток += tmpflow;
            maxflow(tmpflow); //изменяем данные о графе
            tmpflow = 9999;
            prevflow = 9999;
            std::cout << "Adges after finding this path: \n";
            print_edges();
        }
    }
}

```

```

        way = from;
        zeroing_posible(); // разрешаем переход по всем ребрам
        tmp = is_new(from); // начинаем поиск пути заного
        next = 1;
    }
}

};

int main() {
    graf* a = new graf;
    a->init();
    a->print_edges();
    a->serch();
    a->alf_edges_print();
}

```