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. 8382 _____ Ершов М.И.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучение работы алгоритма Форда-Фалкерсона для нахождения максимального потока в сети.

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

v0 - исток

vn - сток

vi vj ωij - ребро графа

vi vj ωij - ребро графа

...

Выходные данные:

Pmax - величина максимального потока

vi vj ωij - ребро графа с фактической величиной протекающего потока

vi vj ωij - 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).

Индивидуализация.

Вар. 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 Форда-Фалкерсона.

Остаточная сеть — это граф с множеством ребер с положительной остаточной пропускной способностью. В остаточной сети может быть путь из u в v, даже если его нет в исходном графе (если в исходной сети есть путь (v, u) с положительным потоком).

Дополняющий путь — это путь в остаточной сети от истока до стока.

Идея алгоритма заключается в том, чтобы запускать поиск в глубину (в индивидуализации по правилу максимальной остаточной пропускной способности) в остаточной сети до тех пор, пока возможно найти новый путь от истока до стока.

Вначале алгоритма остаточная сеть — это исходный граф. Алгоритм ищет дополняющий путь в остаточной сети по следующему алгоритму:

- Находим все смежные вершины к текущей рассматриваемой
- Переходим к вершине с максимальной текущей остаточной пропускной способностью
- Повторяем шаг 1-2 для новой рассматриваемой вершины (алгоритм рекурсивный)
- Продолжаем, пока не дойдем до стока.

Если путь был найден, то остаточная сеть перестраивается, а к максимальному потоку прибавляется величина максимальной пропускной способности дополняющего пути.

Если путь от истока к стоку не был найдет, то максимальный поток найден и алгоритм завершает свою работу.

Очевидно, что максимальный поток в сети является суммой всех максимальных пропускных способностей дополняющих путей.

Описание функций и структур данных.

Структуры данных.

struct Node - структура для хранения остаточной пропускной способности основного (int f1) и дополнительного (int f2) ребра. Имеет два конструктора для удобной инициализации значений при заполнении сети.

vector<vector<Node>> network – структура для хранения остаточной сети (матрица смежности). По сути, является двумерным массивом нод – описанная ранее структура для хранения пропускной способности.

vector<pair<int, int>> graph - вектор пар для хранения исходной сети.

vector<bool> viewed – вектор для хранения уже посещенных вершин во время поиска одного из путей

int startPeak, finishPeak - сток и исток.

Функции.

int recFindPath(int v, int delta) – функция для рекурсивного поиска пути. Правило следующее: каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность. Функция принимает на вход вершину, из которой ищется путь (исток) и текущую минимальная пропускная способность на пути. Возвращает минимальную пропускную способность на пути.

void print() – функция печатает результат после того, как алгоритм заканчивает свою работу, используя граф для вывода вершин и сеть, получившуюся в результате работы recFindPath(), для вывода количества пущенного потока.

void findMaximumFlow() – функция для поиска максимального потока в сети. По сути, просто вызывает recFindPath(), пока она возвращает путь, после чего печатает результат, используя функцию print().

Сложность алгоритма.

E – множество ребер графа.

V – множество вершин графа.

F – величина максимальной пропускной способности графа.

По времени.

На каждом шаге мы ищем путь от стока к истоку, поиском в глубину с модификацией: каждый раз выполняется переход по дуге, имеющей максимальную остаточную пропускную способность.

Так как просматривать ребра нужно в порядке уменьшения пропускной способности, для этого все ребра вершины сортируются, на это приходится тратить |E| * log(|E|) операций. Помимо этого, алгоритм представляет собой обычный поиск в глубину, поэтому поиск нового дополняющего пути в сети происходит за O(|E| * log|E| * |V|).

В худшем случае, на каждом шаге мы будем находить дополняющий путь с пропускной способностью 1, тогда получим сложность по времени O(F* |E| * log|E| * |V|)

По памяти.

Для хранения остаточной сети используется матрица смежности, дающая сложность по памяти O(|V|2). Так же используется дополнительная память для хранения исходного графа в виде массива O(|E|) и массив для хранения посещенных вершин (аналогично O(|E|)). В итоге, сложность по памяти выходит O(|V|2).

Тестирование.

Ввод	Вывод
6 k k k c 10 c d 10 c b 1 b c 1 k b 10 b d 10	0 k b 0 k c 0 b c 0 b d 0 c b 0 c d 0

10 a f a b 16 a c 13 c b 4 b c 10 b d 12 c e 14 d c 9 d f 20 e d 7 e f 4	23 a b 12 a c 11 b c 0 b d 12 c b 0 c e 11 d c 0 d f 19 e d 7 e f 4
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
11 a d a b 7 a c 3 a f 5 c b 4 c d 5 b d 6 b f 3 b e 4 f b 7 f e 8 e d 10	15 a b 7 a c 3 a f 5 b d 6 b e 1 b f 0 c b 0 c d 3 e d 6 f b 0 f e 5

Вывод.

В ходе лабораторной работы была изучена работа алгоритма поиска максимального потока в сети - метод Форда-Фалкерсона, способы хранения графа и остаточной сети и сложности по времени и памяти.

Приложение А.
Исходный код программы.

```
# #include <iostream>
#include <climits>
#include <set>
#include <algorithm>
#include <vector>

using namespace std;

// Нода для хранения прямого и дополнительно ребра в остаточной сети
struct Node {
    int f1;
    int f2;

    Node() : f1(0), f2(0) {}
    Node(int f1, int f2) : f1(f1), f2(f2){}
};

vector<bool> viewed; // Посещенные вершины
int startPeak, finishPeak; // Старт и финиш
vector<vector<Node>> network(128, vector<Node>(128)); // Остаточная сеть
vector<pair<int, int>> graph; // Граф

// Поиск дополняющего пути
int recFindPath(int v, int delta) {
    char tmp_peak = static_cast<char>(v);
    cout << "Рассматриваем " << tmp_peak << endl;
    if (viewed[v]) { // Уже просмотренна, выходим
        cout << tmp_peak << " уже просмотренна!" << endl;
        return 0;
    }
    viewed[v] = true;

    // Конечная вершина, нашли дополняющий путь, возвращаемся обратно и
    // применяем найденную минимальную пропускную способность
    if (v == finishPeak) {
        cout << "Дошли до конечной, путь найден!" << endl;
        return delta;
    }

    set<pair<int, int>, greater<>> sorted; // сет для сортировки вершин по
пропускной способности

    // Ищем все смежные вершины
    cout << "Ищем все пути из " << tmp_peak << ", которые можно использовать"
<< endl;
    for (size_t u = 0; u < network[v].size(); u++) {
        if (!viewed[u] && network[v][u].f2 > 0) {
            sorted.insert(make_pair(network[v][u].f2, u));
            cout << "Нашли " << static_cast<char>(u) << "(" <<
network[v][u].f2 << ") (f2)" << endl;
        }
        if (!viewed[u] && network[v][u].f1 > 0) {
            sorted.insert(make_pair(network[v][u].f1, u));
            cout << "Нашли " << static_cast<char>(u) << "(" <<
network[v][u].f1 << ") (f1)" << endl;
        }
    }

    cout << "Вершины сейчас: ";
    for (auto item : sorted) {
```

```
        cout << static_cast<char>(item.second) << "(" << item.first << ") ";
    }
    cout << endl;

    for(auto u : sorted) { // Обходим все смежные вершины в порядке
приоретета
        if (network[v][u.second].f2 > 0) { // Если дополнительное ребро у
пути больше 0

            // После нахождения пути, здесь будет максимальное возможная
величина протекающего потока
            // Рекурсивный поиск пути
            int new_flow_delta = recFindPath(u.second, min(delta,
network[v][u.second].f2));

            if (new_flow_delta > 0){
                // Применяем минимальную пропускную способность к пути
                cout << "network[" << static_cast<char>(u.second) << "][" <<
static_cast<char>(v) <<"].f1 = " << network[u.second][v].f1 << " -> ";
                network[u.second][v].f1 += new_flow_delta;
                cout << "network[" << static_cast<char>(u.second) << "][" <<
static_cast<char>(v) <<"].f1 = " << network[u.second][v].f1 << ";" << endl;
                cout << "network[" << static_cast<char>(v) << "][" <<
static_cast<char>(u.second) <<"].f2 = " << network[v][u.second].f2 << " -> ";
                network[v][u.second].f2 -= new_flow_delta;
                cout << "network[" << static_cast<char>(v) << "][" <<
static_cast<char>(u.second) <<"].f2 = " << network[v][u.second].f2 << "; " <<
endl;

                return new_flow_delta;
            }
        }

        if (network[v][u.second].f1 > 0) { // Если прямое ребро у пути больше
0

            // После нахождения пути, здесь будет максимальное возможная
величина протекающего потока
            // Рекурсивный поиск пути
            int new_flow_delta = recFindPath(u.second, min(delta,
network[v][u.second].f1));

            if (new_flow_delta > 0) {
                // Применяем минимальную пропускную способность к пути
                cout << "network[" << static_cast<char>(u.second) << "][" <<
static_cast<char>(v) <<"].f2 = " << network[u.second][v].f2 << " -> ";
                network[u.second][v].f2 += new_flow_delta;
                cout << "network[" << static_cast<char>(u.second) << "][" <<
static_cast<char>(v) <<"].f2 = " << network[u.second][v].f2 << ";" <<endl;
                cout << "network[" << static_cast<char>(v) << "][" <<
static_cast<char>(u.second) <<"].f1 = " << network[v][u.second].f1 << " -> ";
                network[v][u.second].f1 -= new_flow_delta;
                cout << "network[" << static_cast<char>(v) << "][" <<
static_cast<char>(u.second) <<"].f1 = " << network[v][u.second].f1 <<
";"<<endl;

                return new_flow_delta;
            }
        }

    }
    return 0;
}

// Печать результата
void print(){
```

```
    sort(graph.begin(), graph.end());
    for(size_t i = 0; i < graph.size(); i++) { // Вывод ребер графа и
количества пущенного тока
        cout << static_cast<char>(graph[i].first) << ' ' <<
static_cast<char>(graph[i].second) << ' ' <<
network[graph[i].second][graph[i].first].f2 << endl;
    }
}

// Функция для поиска максимального потока
void findMaximumFlow(){
    int flow, ans;
    ans = 0;
    while (true){
        // Очищаем просмотренные
        fill(viewed.begin(), viewed.end(), false);

        // Ищем путь
        flow = recFindPath(startPeak, INT_MAX);

        // Поток равный нулю или INT_MAX, не было найденно дополняющего пути
и был найден максимальный поток
        if (flow == 0 || flow == INT_MAX) {
            cout << "Путь не найден, завершаем работу " << endl;
            break;
        } else {
            cout << "Поток увеличился на " << flow << "; ";
            ans +=flow;
            cout << "Максимальный поток сейчас: " << ans << endl;
        }
    }

    cout << ans << endl;
    print();
}

int main() {
    system("chcp 65001");

    int num_of_ribs;
    char u, v;
    int capacity;

    viewed.resize(128);

    cin >> num_of_ribs;

    cin >> u >> v;
    startPeak = static_cast<int>(u);
    finishPeak = static_cast<int>(v);

    int k = 0;
    while (k++ < num_of_ribs) {
        cin >> u >> v >> capacity;
        int i = static_cast<int>(u);
        int j = static_cast<int>(v);

        graph.emplace_back(i, j);
        network[i][j].f1 = capacity;
    }

    findMaximumFlow();
    return 0;
}
```