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: Алгоритм Форда-Фалкерсона

Студент гр. 7383 _____ Медведев И. С.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2019

Содержание

Цель работы

Ознакомиться с алгоритмом нахождения максимального потока в сети с помощью алгоритма Форда-Фалкерсона.

Реализация задачи

Для решения поставленной задачи был использован класс Matrix. В данном классе содержатся поля: int** graph – поле для хранения матрицы смежности, std::vector<char> nodes – для хранения вершин графа, std::vector<int> way – для хранения индекса вершины из которой мы пришли, bool* is_visited – хранит информацию о том, посещена ли вершина по данному индексу, int** flows – для хранения потоков, char source; - для хранения истока, std::vector<std::tuple<char, char, int>> edges – для хранения ребер и их весов, char stock – для хранения стока, unsigned size – для хранения размера матрицы.

Также класс Matrix содержит конструктор, который считывает вершины, комплектует их в поле nodes, а также вызывает метод create_graph. void create_graph(std::vector<std::tuple<char, char, int>> edges) принимает на вход ребра графа, выделяет память и заполняет двумерные массивы flows и graph.

Метод int find_index(char node) – принимает на вход вершину, и ищет ее индекс в nodes. Метод void DFS(char node) – осуществляет поиск в глубину, заполняя вектор way. Метод bool get_way() –вызывает метод DFS, затем все вершины помечает как не просмотренные, возвращает значение way[find_index(stock)] != -1. Если возвращается false, значит мы рассмотрели все возможные пути до стока. Метод int FF() – ищет максимальный поток, проходясь в обратном порядке по вершинам, которые мы обошли с помощью поиска в глубину, меняет потоки.

Тестирование

Программа собрана в операционной системе Ubuntu 17.04 с использованием компилятора g++. В других ОС и компиляторах тестирование

не проводилось. Результаты тестирования показали, что поставленная цель выполнена. Результаты тестирования представлены в Приложении Б.

Сложность алгоритма

В данном алгоритме, в худшем случае, поток будет меняться на единицу, поэтому он пройдет за O(C), где C – максимальный поток. Также здесь реализован поиск в глубину, сложность которого O(|V|+|E|), где V – множество вершин, а E - рёбер. Следовательно общая сложность будет O(C(|V|+|E|)).

Вывод

В процессе выполнения данной лабораторной работы был изучен и реализован алгоритм Форда-Фалкерсона. Была написана программа на языке программирования C++ и оценена примерная сложность этого алгоритма.

ПРИЛОЖЕНИЕ А.

КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <cstring>
#include <algorithm>
#include <tuple>

#define N 100

class Matrix{
    int** graph;
    std::vector<char> nodes;
    std::vector<int> way;
    bool* is_visited;
    int** flows;
    char source;
    char stock;
    unsigned size;
    std::vector<std::tuple<char, char, int>> edges;
public:
    Matrix(): way(N, 0){
        char from , to;
        int number, cost;
        std::cin >> number >> source >> stock;
        for (unsigned i = 0; i < number; i++){
            std::cin >> from >> to >> cost;
            if (std::find(nodes.begin(), nodes.end(), from) == nodes.end())
                nodes.push_back(from);
            if (std::find(nodes.begin(), nodes.end(), to) == nodes.end())
                nodes.push_back(to);
            edges.push_back(std::make_tuple(from, to, cost));
        }
       size = nodes.size();
       create_graph(edges);
    }


    void create_graph(std::vector<std::tuple<char, char, int>> edges){
        int cost;
        std::vector<char>::iterator it;
        char from, to;
        is_visited = new bool[size];
        graph = new int*[size];
        flows = new int*[size];
        for (int i = 0; i < size; i++){
            way[i] = -1;
            is_visited[i] = false;
            graph[i] = new int[size];
            flows[i] = new int[size];
            for (int j = 0; j < size; j++) {
                graph[i][j] = 0;
                 flows[i][j] = 0;
            }
        }
        for(auto i : edges){
            std::tie(from, to, cost) = i;
            it = std::find(nodes.begin(), nodes.end(), from);
            int index_from = it - nodes.begin();
```

```
            it = std::find(nodes.begin(), nodes.end(), to);
            int index_to = it - nodes.begin();
            graph[index_from][index_to] = cost;
        }
    }

    int find_index(char node){
        std::vector<char>::iterator it;
        it = std::find(nodes.begin(), nodes.end(), node);
        return it - nodes.begin();
    }

    void DFS(char node) {
        is_visited[find_index(node)] = true;
        for (int i = 0; i<size; i++)
            if (!is_visited[i] && (graph[find_index(node)][i] -
flows[find_index(node)][i] > 0 && graph[find_index(node)][i] != 0 ||
flows[find_index(node)][i] < 0 && graph[i][find_index(node)] != 0)) {
                way[i] = find_index(node);
                DFS(nodes[i]);
            }
    }


    bool get_way() {
            DFS(source);
            for (size_t i = 0; i < size; i++) {
                is_visited[i] = false;
            }
            return (way[find_index(stock)] != -1);
    }

    int FF() {
        int max_flow = 0;
        while (get_way()) {
            int tmp = INT_MAX;
            for (int v = find_index(stock); way[v] >= 0; v = way[v])
                 tmp = std::min(tmp, graph[way[v]][v] - flows[way[v]][v]);
            for (int v = find_index(stock); way[v] >= 0; v = way[v]) {
                flows[way[v]][v] += tmp;
                flows[v][way[v]] -= tmp;
            }
            max_flow += tmp;
            for (int i = 0; i<size; i++)
                way[i] = -1;
        }
        return max_flow;
    }

    void print(){
        std::sort(edges.begin(), edges.end());
        for(auto i : edges){
            int it1 = find_index(std::get<0>(i));
            int it2 = find_index(std::get<1>(i));
            if(flows[it1][it2] < 0)
                flows[it1][it2] = 0;
            std::cout << nodes[it1] << ' ' << nodes[it2] << ' ' <<
flows[it1][it2] << std::endl;
        }
    }

    ~Matrix(){
        for (int i = 0; i < size; i++){
```

```
            delete [] graph[i];
            delete [] flows[i];
        }
        delete [] graph;
        delete [] flows;
        delete [] is_visited;
    }


};


int main(){

    Matrix A;
    std::cout << A.FF() << std::endl;
    A.print();
    return 0;
}
```

ПРИЛОЖЕНИЕ Б.

ТЕСТОВЫЕ СЛУЧАИ

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования.

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
3 a c a b 7 a c 6 b c 4	10 a b 4 a c 6 b c 4
8 1 4 1 2 7 1 3 6 2 4 8 2 5 1 3 5 2 3 6 4 6 5 7	13 1 2 7 1 3 6 2 4 7 2 5 0 3 5 2 3 6 4 5 4 6 6 5 4