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

Студент гр. 9383 Мосин К.К.

Преподаватель Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Задание.

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

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).

Вариант 7. Производить поиск пути с обеих сторон графа.

Выполнение работы.

Алгоритм:

1. Копия оригинального графа для работы с остаточной сетью

Необходимо для вычисления фактической величины протекающего потока.

2. Пока есть путь от истока к стоку

Строится любой путь от истока к стоку, проходя по ребрам с потоком больше нуля.

3. Для всех ребер в пути выбирается минимальная пропускная способность

По построенному пути выбирается ребро с минимальной пропускной способностью, чтобы скорректировать остаточную сеть.

4. Вычитание и прибавление для каждого ребра и обратного соответственно

Непосредственная корректировка остаточной сети, чтобы на п.2 не построилась аналогичная остаточная сеть и алгоритм выполнился успешно.

Для реализации индивидуального варианта производился поиск пути с двух сторон. Если вектор пути от истока пересекался с вектором пути от стока, возвращалась комбинация двух векторов.

Тестирование.

В первом случае на вход подается обычный граф. Во втором кейсе этот же граф обнуляется для ребер, ведущих в сток. В третий раз удаляются ребра, ведущие в сток. В четвертом случае, имена вершин заменяются цифрами. Результаты тестирования представлены в таблице 1.

Табл. 1 - Результаты тестирования алгоритма

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
7 a f a b 7 a c 6 b d 6 c f 0 d e 3 d f 0 e c 2	0 a b 0 a c 0 b d 0 c f 0 d e 0 d f 0 e c 0
7 a	0 a b 0

f a b 7 a c 6 b d 6 c d 9 d e 3 d b 4 e c 2	a c 0 b d 0 c f 0 d e 0 d f 0 e c 0
7 1 6 1 2 7 1 3 6 2 4 6 3 6 9 4 5 3 4 6 4 5 3 2	12 1 2 6 1 3 6 2 4 6 3 6 8 4 5 2 4 6 4 5 3 2

Анализ алгоритма.

Для нахождения пути в остаточной сети необходимо E операций, где E - количество ребер в графе. При каждой итерации пропускная способность ребра уменьшается хотя бы на 1, соответственно необходимо O(FE) времени для выполнения алгоритма, где F - максимальный поток в графе.

Вывод

В ходе выполнения лабораторной работы был посчитан максимальный поток графа, используя алгоритм Форда-Фалкерсона.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: API.h

```
#pragma once

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

bool input(int& count, char& start, char& end, std::map<char, std::map<char, int>>& graph, std::istream& stream);
static std::map<char, std::map<char, int>> transpose_graph(std::map<char, std::map<char, int>>& graph);
static std::pair<std::vector<char>, int> get_path(std::map<char, std::map<char, int>>& rGraph, char s, char t);
static std::pair<std::vector<char>, int> restore_path(std::map<char, std::pair<char, int>>& graph1, std::map<char, std::pair<char, int>>& graph2, char s, char t, char node);
std::pair<int, std::map<char, std::map<char, int>>> FordFulkerson(std::map<char, std::map<char, int>>& graph, char s, char t);
```

Название файла: API.cpp

```
#include "API.h"

bool input(int& count, char& start, char& end, std::map<char, std::map<char, int>>& graph, std::istream& stream){
    stream >> count >> start >> end;
```

```
    char node1, node2;
    int weight = 0;
    for (int i = 0; i < count; ++i) {
        stream >> node1 >> node2 >> weight;
        //Some check for return false
        graph[node1][node2] = weight;
    }

    return true;
}

std::pair<std::vector<char>, int> path(std::map<char, std::map<char, int>>& rGraph,
char s, char t) {
    std::pair<std::vector<char>, int> path_flow;

    std::map<char, std::map<char, int>> transposedGraph = transpose_graph(rGraph);

    std::queue<char> queue_from_start;
    queue_from_start.push(s);

    std::vector<char> visited_from_start;
    visited_from_start.push_back(s);

    std::queue<char> queue_from_end;
    queue_from_end.push(t);

    std::vector<char> visited_from_end;
    visited_from_end.push_back(t);

    char node;
```

```
    std::map<char, std::pair<char, int>> graph1, graph2;
    while (!queue_from_start.empty() && !queue_from_end.empty()) {
        node = queue_from_start.front();
        queue_from_start.pop();
        if (std::find(visited_from_end.begin(), visited_from_end.end(), node) != visited_from_end.end()) {
            return restore_path(graph1, graph2, s, t, node);
        }

        for (auto it : rGraph[node]) {
            if (std::find(visited_from_start.begin(), visited_from_start.end(), it.first) == visited_from_start.end() && it.second > 0) {
                queue_from_start.push(it.first);
                visited_from_start.push_back(it.first);
                graph1[it.first] = std::make_pair(node, it.second);
            }
        }

        node = queue_from_end.front();
        queue_from_end.pop();
        if (std::find(visited_from_start.begin(), visited_from_start.end(), node) != visited_from_start.end()) {
            return restore_path(graph1, graph2, s, t, node);
        }

        for (auto it : transposedGraph[node]) {
            if (std::find(visited_from_end.begin(), visited_from_end.end(), it.first) == visited_from_end.end() && it.second > 0) {
                queue_from_end.push(it.first);
                visited_from_end.push_back(it.first);
```

```
                graph2[it.first] = std::make_pair(node, it.second);
            }
        }
    }

    return path_flow;
}

std::map<char, std::map<char, int>> transpose_graph(std::map<char, std::map<char, int>>& graph) {
    std::map<char, std::map<char, int>> transposedGraph;
    for (auto it : graph) {
        for (auto vertex : it.second) {
            transposedGraph[vertex.first][it.first] = vertex.second;
        }
    }
    return transposedGraph;
}

std::pair<std::vector<char>, int> restore_path(std::map<char, std::pair<char, int>>& graph1, std::map<char, std::pair<char, int>>& graph2, char s, char t, char node) {
    std::vector<char> path;
    int flow = __INT_MAX__;

    path.push_back(node);

    char vertex = node;
    while (vertex != s) {
        path.push_back(graph1[vertex].first);
        flow = std::min(flow, graph1[vertex].second);
```

```
        vertex = graph1[vertex].first;
    }
    std::reverse(path.begin(), path.end());

    vertex = node;
    while (vertex != t) {
        path.push_back(graph2[vertex].first);
        flow = std::min(flow, graph2[vertex].second);
        vertex = graph2[vertex].first;
    }

    return std::make_pair(path, flow);
}

std::pair<int, std::map<char, std::map<char, int>>> FordFulkerson(std::map<char, std::map<char, int>>& graph, char s, char t) {
    std::map<char, std::map<char, int>> rGraph = graph;

    int flow = 0;
    for(std::pair<std::vector<char>, int> path_flow = path(rGraph, s, t); !path_flow.first.empty(); path_flow = path(rGraph, s, t)) {
        for (int i = 0; i < path_flow.first.size() - 1; ++i) {
            rGraph[path_flow.first[i]][path_flow.first[i + 1]] -= path_flow.second;
            rGraph[path_flow.first[i + 1]][path_flow.first[i]] += path_flow.second;
        }

        flow += path_flow.second;
    }

    return std::make_pair(flow, rGraph);
```

```
}
```

Название файла: main.cpp

```
#include "API.h"

int main(int argc, char *argv[]) {
    int count;
    char start, end;
    std::map<char, std::map<char, int>> graph;

    if (input(count, start, end, graph, std::cin)) {
        std::pair<int, std::map<char, std::map<char, int>>> path_flow = FordFulkerson(graph, start, end);
        std::cout << path_flow.first << std::endl;
        for (auto i : graph) {
            for (auto j : graph[i.first]) {
                std::cout << i.first << " " << j.first << " ";
                if (graph[i.first][j.first] - path_flow.second[i.first][j.first] > 0) {
                    std::cout << graph[i.first][j.first] - path_flow.second[i.first][j.first] << std::endl;
                }
                else {
                    std::cout << 0 << std::endl;
                }
            }
        }
    }

    return 0;
}
```

ПРИЛОЖЕНИЕ В
ТЕСТЫ

Название файла: test.cpp

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

#include "API.h"

TEST_CASE("FordFulkerson algorithm") {
    int count;
    char start, end;
    std::map<char, std::map<char, int>> graph;
    std::stringstream stream;
    SECTION("Path found and flow calculated") {
        stream << "7\na\nf\na b 7\na c 6\nb d 6\nc f 9\nd e 3\nd f 4\ne c 2\n";
        input(count, start, end, graph, stream);
        REQUIRE(FordFulkerson(graph, start, end).first == 12);
    }
    SECTION("Path found but flow is zero") {
        stream << "7\na\nf\na b 7\na c 6\nb d 6\nc f 0\nd e 3\nd f 0\ne c 2\n";
        input(count, start, end, graph, stream);
        REQUIRE(FordFulkerson(graph, start, end).first == 0);
    }
    SECTION("Path not found") {
        stream << "7\na\nf\na b 7\na c 6\nb d 6\nc d 9\nd e 3\nd b 4\ne c 2\n";
        input(count, start, end, graph, stream);
        REQUIRE(FordFulkerson(graph, start, end).first == 0);
    }
    SECTION("Numbers only") {
        stream << "7\n1\n6\n1 2 7\n1 3 6\n2 4 6\n3 6 9\n4 5 3\n4 6 4\n5 3 2\n";
```

```
        input(count, start, end, graph, stream);
        REQUIRE(FordFulkerson(graph, start, end).first == 12);
    }
}
```