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 графах

Студент гр. 7304 _____ Пэтайчук Н.Г.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2019

Цель работы

Исследование алгоритмов, связанных с потоками на графах, на примере алгоритма Форда-Фалкерсона.

Постановка задачи

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

- Количество ориентированных рёбер;
- Исток;
- Сток;
- Список рёбер графа;

Выходные данные

- Максимальный поток;
- Список рёбер графа с фактическими величинами протекающего потока;

Ход работы

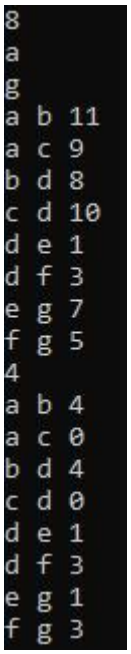
1. Объявление типов «*Vertex*» (*char*), «*Graph*» (*map<Vertex, map<Vertex, long long int>>*) и «*Path*» (*map<Vertex, Vertex>*), которые необходимы для реализации логики алгоритма и программы;
2. Реализация функции поиска в ширину в графе, которая используется для нахождения пути из истока в сток. Таким образом, будет реализована не обычная версия алгоритма Форда-Фалкерсона, а алгоритм Эдмондса-Карпа, чья сложность, в отличие от обычного алгоритма, не зависит от результата работы (то есть от максимального потока). Функция поиска в ширину возвращает булевское значение, говорящее о том, нашёлся ли путь до стока или нет, и записывает в передаваемый аргумент типа *Path* путь от истока до стока;
3. Реализация функции, использующей алгоритм Эдмондса-Карпа для нахождения максимального потока в графе. Данный алгоритм работает до тех пор, пока существуют пути до стока, не содержащие нулевые рёбра. После того, как найден такой путь, находится минимальное значение пропускной способности ребра из данного пути, это значение прибавляется к значению максимального потока в графе (которое сначала равно 0) и вычитается из весов всех рёбер данного пути;
4. Реализация головной функции, которая принимает исходные данные (число рёбер, сток и исток, а также данные о самим рёбрах) и выводит результат

работы алгоритма (максимальный поток в графе и данные о фактическом потоке, идущим через каждое ребро);

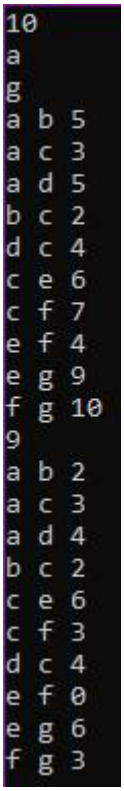
Весь исходный код программы представлен в Приложении 1.

Тестирование программы

1) Тест 1: Простой граф



2) Тест 2: Граф посложнее



Вывод

В ходе данной лабораторной работы был изучен алгоритм поиска максимального потока в графе, а именно алгоритм Форда-Фалкерсона, а также была рассмотрены модификации данного алгоритма на примере алгоритма Эдмондса-Карпа, использующего поиск в ширину для поиска пути из истока в сток и который был реализован в данной лабораторной работе. Также были закреплены навыки реализации графов и работы с ними, а также со стандартными контейнерами языка C++ (векторы, словари и очереди).

Приложение 1: Исходный код программы

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

using namespace std;

using Vertex = char;
using Graph = map<Vertex, map<Vertex, long long int>>;
using Path = map<Vertex, Vertex>;

bool findWay_BFS(Graph &graph, Vertex source, Vertex stock, Path
&stock_path)
{
    vector<Vertex> visited_vertexes = {source};
    queue<Vertex> bfs_queue;
    Path come_from;
    Vertex current_vertex;

    bfs_queue.push(source);
    while(!bfs_queue.empty())
    {
        current_vertex = bfs_queue.front();
        bfs_queue.pop();

        for (auto const &neighbor : graph[current_vertex])
        {
            if (neighbor.second > 0 &&
                    find(visited_vertexes.begin(),
visited_vertexes.end(), neighbor.first) == visited_vertexes.end())
            {
                visited_vertexes.push_back(neighbor.first);
                bfs_queue.push(neighbor.first);
                come_from[neighbor.first] = current_vertex;
            }
        }
    }
    stock_path = come_from;
    return find(visited_vertexes.begin(), visited_vertexes.end(),
stock) != visited_vertexes.end();
}

long long int findMaxFlow_EdmondsKarp(Graph &graph, Vertex source,
Vertex stock)
{
    Path stock_path;
    long long int max_flow = 0;
    Vertex from, where;
```

```
    while (findWay_BFS(graph, source, stock, stock_path))
    {
        long long int max_way_flow = LLONG_MAX;
        where = stock;
        while (where != source)
        {
            max_way_flow = min(max_way_flow,
graph[stock_path[where]][where]);
            where = stock_path[where];
        }

        max_flow += max_way_flow;
        where = stock;
        while (where != source)
        {
            from = stock_path[where];
            graph[from][where] -= max_way_flow;
            graph[where][from] += max_way_flow;
            where = stock_path[where];
        }
    }
    return max_flow;
}

int main()
{
    Graph graph, residual_graph;
    Vertex source, stock, from, where;
    long long int edge_number, bandwidth, max_flow;

    cin >> edge_number >> source >> stock;
    while ((edge_number--) && (cin >> from >> where >> bandwidth))
        graph[from][where] = bandwidth;

    residual_graph = graph;
    max_flow = findMaxFlow_EdmondsKarp(residual_graph, source,
stock);

    cout << max_flow << endl;
    for (auto const &current : graph)
        for (auto const &neighbor : graph[current.first])
            cout << current.first << " " << neighbor.first << " "
                 << (int(neighbor.second -
residual_graph[current.first][neighbor.first]) < 0 ? 0 :
                     int(neighbor.second -
residual_graph[current.first][neighbor.first])) << endl;
    return 0;
}
```