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. 7383 _____ Власов Р.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2019

Содержание

Цель работы

Цель работы: познакомиться с алгоритмом поиска максимального потока в сети Форда-Фалкерсона, создать программу, осуществляющую поиск кратчайшего пути в графе с помощью алгоритма Форда-Фалкерсона.

Формулировка 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). 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 2м: граф представлен в виде матрицы смежности, поиск пути осуществляется через поиск в глубину.

Реализация задачи

Программу было решено писать на языке программирования C++.

Для реализации поставленной задачи были созданы классы graph и edge.

```
class graph
{
    char start;
    char finish;
    int max_flow;
    std::vector<char> indexes;
    edge** matrix;
    size_t reserved_indexes;
    size_t find_index(char name);
    void realloc();
    void clear_matrix();
    std::vector<char> find_path_dfs();
    int find_min_bandwidth(std::vector<char> path);
    void reduce_max_bandwidth(std::vector<char> path, int value);
public:
    graph(char start, char finish);
    ~graph();
    void add_edge(char from, char dest, int bandwidth);
    int find_flow();
    void print_answer();
};
```

Класс graph отвечает за хранение графа в виде матрицы смежности, также содержит функцию find_flow(), осуществляющую вычисление максимального патока из вершины start в вершину finish и фактическую величину потока для каждого ребра с помощью алгоритма Форда-Фалкерсона.

```
struct edge{
    int max_bandwidth = 0;
    int final_bandwidth = 0;
    bool from_input = false;
    edge& operator= (const int b)
};
```

Класс edge используется для хранения данных о ребре в матрице смежности.

Исходный код программы представлен в приложении Б.

Исследование алгоритма

Сложность алгоритма составляет O(|E|·f), где |E| – количество ребер в графе, f – максимальный поток в графе. На каждом шаге алгоритм увеличивает величину потока по крайней мере на единицу. Каждый шаг можно выполнить за O(|E|).

Тестирование

1. Процесс тестирования

Программа собрана в операционной системе Ubuntu 18.04.2 LTS bionic компилятором g++ version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04). В других ОС и компиляторах тестирование не проводилось.

2. Результаты тестирования

В результате тестирования программы ошибок выявлено не было. Тестовые случаи представлены в приложении А.

Вывод

В ходе выполнения данной работы был изучен метод поиска максимального поток в сети с помощью алгоритма Форда-Фалкерсона. Была написана программа, применяющая алгоритм Форда-Фалкерсона для поиска максимального потока в графе. Сложность реализованного алгоритма составляет O(|V|·f).

ПРИЛОЖЕНИЕ А. ТЕСТОВЫЕ СЛУЧАИ

Входные данные	Результат
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
4 a d a c 2 c d 4 a b 3 b d 1	3 a b 1 a c 2 b d 1 c d 2

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЙ КОД

```
#include <vector>
#include <stack>
#include <iostream>
#include <algorithm>

struct edge{
    int max_bandwidth = 0;
    int final_bandwidth = 0;
    bool from_input = false;

    edge& operator= (const int b)
    {
        this->max_bandwidth = b;
        return *this;
    }
};

class graph
{
    char start;
    char finish;
    int max_flow;
    std::vector<char> indexes;
    edge** matrix;
    size_t reserved_indexes;

    size_t find_index(char name)
    {
        for (size_t i = 0; i < indexes.size(); i++)
        {
            if (indexes[i] == name)
                return i;
        }

        if (indexes.size() == reserved_indexes)
            realloc();
        indexes.push_back(name);
        return indexes.size() - 1;
    }

    void realloc()
    {
        size_t res = reserved_indexes + 10;
        edge** tmp = new edge*[res];
```

```
    for (size_t i = 0; i < res; i++)
    {
        matrix[i] = new edge[res];
    }
    for (size_t i = 0; i < reserved_indexes; i++)
        for (size_t j = 0; j < reserved_indexes; j++)
        {
            tmp[i][j] = matrix[i][j];
        }
    clear_matrix();
    matrix = tmp;
    reserved_indexes = res;
}

void clear_matrix()
{
    for (size_t i = 0; i < reserved_indexes; i++)
        delete[] matrix[i];
    delete[] matrix;
}

std::vector<char> find_path_dfs()
{
    char current = start;
    std::vector<char> current_path;
    std::vector<char> visited = indexes;  // 0 - if visited
    std::stack<std::pair<char, std::vector<char>>> dfs_stack;
    dfs_stack.push(std::make_pair(current, current_path));
    visited[find_index(current)] = 0;

    while (!dfs_stack.empty())
    {
        current = dfs_stack.top().first;
        current_path = dfs_stack.top().second;
        dfs_stack.pop();
        current_path.push_back(current);
        size_t ind = find_index(current);
        for (size_t i = 0; i < indexes.size(); i++)
        {
            if (visited[i] && matrix[ind][i].max_bandwidth)
            {
                if (indexes[i] == finish)
                {
                    current_path.push_back(finish);
                    return current_path;
                }
```

```
                    visited[i] = 0;
                    dfs_stack.push(std::make_pair(indexes[i],
current_path));
                }
            }
        }
        return std::vector<char>();
    }

    int find_min_bandwidth(std::vector<char> path)
    {
        size_t ind_f, ind_d;
        ind_f = find_index(path[0]);
        ind_d = find_index(path[1]);
        int cur_min = matrix[ind_f][ind_d].max_bandwidth;

        for (size_t i = 2; i < path.size(); i++)
        {
            ind_f = ind_d;
            ind_d = find_index(path[i]);
            if (matrix[ind_f][ind_d].max_bandwidth < cur_min)
                cur_min = matrix[ind_f][ind_d].max_bandwidth;
        }
        return cur_min;
    }

    void reduce_max_bandwidth(std::vector<char> path, int value)
    {
        size_t ind_f, ind_d;
        ind_f = find_index(path[0]);
        ind_d = find_index(path[1]);
        matrix[ind_f][ind_d].max_bandwidth -= value;
        matrix[ind_f][ind_d].final_bandwidth += value;

        for (size_t i = 2; i < path.size(); i++)
        {
            ind_f = ind_d;
            ind_d = find_index(path[i]);
            matrix[ind_f][ind_d].max_bandwidth -= value;
            matrix[ind_f][ind_d].final_bandwidth += value;
        }
    }

public:
    graph(char start, char finish)
```

```
{
    this->start = start;
    this->finish = finish;
    max_flow = 0;
    reserved_indexes = 10;
    matrix = new edge*[reserved_indexes];
    for (size_t i = 0; i < reserved_indexes; i++)
        matrix[i] = new edge[reserved_indexes];
}

~graph()
{
    clear_matrix();
}

void add_edge(char from, char dest, int bandwidth)
{
    size_t index_f = find_index(from);
    size_t index_d = find_index(dest);
    matrix[index_f][index_d] = bandwidth;
    matrix[index_f][index_d].from_input = true;
}

int find_flow()
{
    std::vector<char> path = find_path_dfs();
    int cur_bandwidth = 0;

    while(path.size())
    {
        cur_bandwidth = find_min_bandwidth(path);
        max_flow += cur_bandwidth;
        reduce_max_bandwidth(path, cur_bandwidth);
        path = find_path_dfs();
    }
    return  max_flow;
}

void print_answer()
{
    size_t ind_f, ind_d;
    std::vector<char> order = indexes;
    std::sort(order.begin(), order.end());
    std::cout << max_flow << std::endl;
    for (size_t i = 0; i < indexes.size(); i++)
    {
```

```
            ind_f = find_index(order[i]);
            for (size_t j = 0; j < indexes.size(); j++)
            {
                ind_d = find_index(order[j]);
                if (matrix[ind_f][ind_d].from_input)
                    std::cout << indexes[ind_f] << ' ' <<
indexes[ind_d] << ' ' << matrix[ind_f][ind_d].final_bandwidth <<
std::endl;
            }
        }
    }
};
int main()
{
    int n;
    std::cin >> n;
    char start, finish;
    std::cin >> start >> finish;
    graph gr(start, finish);
    for (int i = 0; i < n; i++)
    {
        int l;
        std::cin >> start >> finish >> l;
        gr.add_edge(start, finish, l);
    }
    gr.find_flow();
    gr.print_answer();

    return 0;
}
```