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.

Студентка гр. 9382 _____ Круглова В.Д.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

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

v_0 - исток

v_n - сток

vi v_j ω_{ij} - ребро графа

vi v_j ω_{ij} - ребро графа

...

Выходные данные:

vi v_j ω_{ij} - ребро графа с фактической величиной протекающего потока

vi v_j ω_{ij} - 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).

Sample Input:

```
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
```

Sample Output:

```
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

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 итеративный)
- 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 получен, 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.

struct Node – структура хранит метку вершины, map соседних вершин и величину потока через дугу до соседней вершины. В структуре перегружен оператор [] и возвращает pair<int,int> - пропускную способность дуги.

class Graph – хранит стартовую и конечную вершину, а также map

зависимостей графа map<char, Node> point – массив зависимостей графа. Хранит информацию в формате [вершина] – [Node]. Описание Node приведено выше.

Функции.

int Graph::searchMaxFlow() – функция для поиска максимального потока в сети. Функция является методом класса Graph, поэтому может работать с private полями класса. Сначала инициализируется начальная вершина. После из map point, которая хранится в классе Graph, получаем массив инцидентных вершин. Однако не все вершины подходят, поэтому заводится контейнер string neighrors_list, в который записываются вершины, которые еще способны пропустить поток и одновременно не приводящие к «тупику» в сети. После находится приоритетная вершина и совершается переход в нее. Таким образом, получаем множество сквозных путей.

Максимальным потоком в графе будет являтся сумма потоков сквозных путей.

char Graph::max_neighbors_flow(map<char, pair<int, int>> n_mas, string n_list) – функция поиска приоритетной дуги. По условию, приоритет отдаётся той, чья пропускная способность выше.

void Graph::print_for_stepik() – печать результата. void Graph::init() – инициализация класса Graph.

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 графа используется класс Graph. Он содержит всю необходимую информацию для работы алгоритма и позволяет не хранить новые данные. Непосредственно класс состоит из map, поэтому сложность по памяти O(|E|).

Тестирование.

Ввод	Вывод
6 k d k c 10 c d 10 c b 1 b c 1	20 b c 0 b d 10 c b 0 c d 10 k b 10 k c 10

k b 10 b d 10	
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 <vector>
#include <algorithm>
#include <map>
#include <string>
#include <set>
#include <climits>

template <typename T>
class Vertex
//Хранит имя вершины, путь до нее и поток по этому пути
{
public:
    T name;
    std::string path;
    int flow_for_edge;
    int flow_for_path = 0;
    Vertex(T name_, std::string path_, int flow_f_e, int flow_f_p)
        : name(name_), path(path_), flow_for_edge(flow_f_e),
flow_for_path(flow_f_p) {}
};

template <typename T>
class Graph
{
private:
    int size; //кол-во рёбер
    T start;  //нач. вершина
    T end;    //кон. вершниа
    std::map<T, std::map<T, int>> const_edges; //Мапа рёбер (неизменяется, нужна
для вывода)
    std::map<T, std::map<T, int>> edges;       //Мапа рёбер, также хранит
обратные рёбра (изменяется)
public:
    Graph() = default;
    void init();
    void print();
    int ff_alg(); //Форд Фалкерсон
};

template <typename T>
void Graph<T>::init()
//Читаем кол-во рёбер, старт, конец и заполняем мапу рёбер
{
    T from, to;
    int weight;
    std::cin >> size;
    std::cin >> start >> end;

    for (int i=0; i<size; i++)
    {
        std::cin >> from >> to >> weight;
        const_edges[from][to] = weight;
        //Мапу для вывода заполняем всеми данными в тесте рёбрами
```

```
        if (to == start || from == end)
        //В мапу для алгоритма кладём все рёбра, кроме ведущих в начало и
исходящие из конца
            continue;

        if (edges[to].count(from) == 0)
        {
            edges[from][to] = 0;
        }
        if (edges[from].count(to) == 0)
        {
            edges[from][to] += weight;
        } else {
            edges[from][to] = weight;
        }
    }
}

template <typename T>
void Graph<T>::print()
{
    for (const auto var : const_edges)
    {
        for (const auto var2 : var.second)
        {
            int f = const_edges[var.first][var2.first] - edges[var.first]
[var2.first];
            if (f < 0) f = 0;
            std::cout << var.first << " " << var2.first << " " <<
                    f << std::endl;
        }
    }
}

template <typename T>
struct set_cmp
//кмп для set'а сортируем сначала по потоку, потом по имени вершины
{
    bool operator() (Vertex<T> a, Vertex<T> b)
    {
        if (a.flow_for_edge == b.flow_for_edge)
            return a.name < b.name;
        return a.flow_for_edge < b.flow_for_edge;
    }
};

template <typename T>
int Graph<T>::ff_alg()
{
    int max_flow = 0; //макс поток
    std::set<Vertex<T>, set_cmp<T>> open; //вершины, доступные для посещения
        //Хранит имя вершины, путь до нее, поток по последнему ребру и мин.
поток на пути
    std::string close; //хранит имена закрытых вершин

    open.insert(Vertex<T>(start, "", INT_MAX, INT_MAX));


    while (!open.empty())
```

```
    {
        Vertex<T> curr = *(--open.end());
        std::cout << "Open list:" << std::endl;
        for (auto var : open)
        {
            std::cout << var.path << "->" << var.name << "; Flow for path = " <<
                         var.flow_for_path << "; Flow for last edge = " <<
                         var.flow_for_edge << ";" << std::endl;
        }
        std::cout << "Edge selected: " << curr.path << "->" << curr.name << " "
<<
                         "; Flow for path = " << curr.flow_for_path <<
                         "; Flow for last edge = " << curr.flow_for_edge <<
                         ";" << std::endl;
        open.erase(--open.end());

        if(curr.name == end)
        // если пришли в конец
        {
            curr.path += curr.name;
            T from, to;
            for (int i=0; i< curr.path.length() - 1; i++)
            {
                from = curr.path[i];
                to = curr.path[i+1];
                edges[from][to] -= curr.flow_for_path;
                edges[to][from] += curr.flow_for_path;
            }
            max_flow += curr.flow_for_path;

            std::cout << "Reached the end! Path: " << curr.path << "; Flow for
path = " <<
                         curr.flow_for_path << "; Summ flow for graph = " <<
max_flow << std::endl;

            close.clear();
            open.clear();
            open.insert(Vertex<T>(start, "", INT_MAX, INT_MAX));
            continue;
        }
        close.push_back(curr.name);
        std::cout << "Close list: " << close << ";" << std::endl;

        for (auto var : edges[curr.name])
        {
            if (close.find(var.first) != std::string::npos || var.second <= 0)
            {
                continue; // если вершина находится в списке закрытых или по ней
нельзя пустить поток
            }
            if (var.first == start) //пропускаем вершины, ведущие в начало
                continue;
            open.insert(Vertex<T>(var.first, curr.path + curr.name, var.second,
                                  std::min(curr.flow_for_path, var.second)));

            std::cout << "Edge {" << curr.path + curr.name << "->" << var.first
<<
                         "; " << var.second << "; " <<
std::min(curr.flow_for_path, var.second) <<
                         "} add to open list" << std::endl;
```

```
        }
    }
    return max_flow;
}

int main()
{
    Graph<char> graph;
    graph.init();
    std::cout << graph.ff_alg() << std::endl;
    graph.print();

    return 0;
}
```