

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ
по лабораторной работе №3
по дисциплине «Построение и анализ алгоритмов»
Тема: Потоки в сети

Студент гр. 8382

Щеглов А.С.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

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

$v_i v_j \omega_{ij}$ - ребро графа

$v_i v_j \omega_{ij}$ - ребро графа

...

Выходные данные:

$v_i v_j \omega_{ij}$ - ребро графа с фактической величиной протекающего потока

$v_i v_j \omega_{ij}$ - 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).

Вариант дополнительного задания.

Вар. 6. Поиск не в глубину и не в ширину, а по правилу: каждый раз выполняется переход по дуге, соединяющей вершины, имена которых в алфавите ближе всего друг к другу. Если таких дуг несколько, то выбрать ту, имя конца которой в алфавите ближайшее к началу алфавита.

Описание алгоритма

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 методов.

`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 neighbors_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 являться сумма потоков сквозных путей.

`char Graph::min_neighbors_flow(map<char, pair<int, int>> n_mas, string n_list)` – функция поиска приоритетной дуги. По условию, приоритет отдаётся той, у которой вершина ближе по алфавиту к текущей.

`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	0
k	k b 0
k	k c 0
k c 10	b c 0
c d 10	b d 0
c b 1	c b 0
b c 1	c d 0
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

Выводы.

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.

```
#include <iostream>
#include <string>
#include <map>
#include <algorithm>
#include <stack>
#include <math.h>

using std::map;
using std::pair;
using std::string;
using std::cout;
using std::cin;
using std::endl;

struct Node
    //структура хранит метку вершины и map соседних вершин и величину потока через ребро
{
    bool markFlag;          //Активна ли метка
    pair<int, char> mark;    //Какой поток пришел и откуда
    map<char, pair<int, int>> neighbors; //мапа вида вершина - {поток туда / поток обратно}
    Node() : markFlag(false) {}
    pair<int, int>& operator[](const char elem)
    {
        return neighbors[elem];
    }
};

class Graph
{
private:
    map<char, Node> point;
    char start, end;
public:
    void init();
    void print_graph();
    void print_for_stepik();
    int searchMaxFlow();
    char min_neighbors_flow(map<char, pair<int, int>>, string, char);
};

void Graph::print_for_stepik()
{
    for (auto var : point)
    {
        for (auto var2 : var.second.neighbors)
            cout << var.first << " " << var2.first << " " << var2.second.second << endl;
    }
}

void Graph::init()
/* Читаем start, end. После заполняем массив зависимостей */
{
    string input;
    int n;
    cin >> n;
    //cout << "Enter start and end point: ";
    cin >> start;
```

```

cin >> end;

char from, to;
int flow;
//cout << "Enter adjacency list:" << endl;
for (int i = 0; i < n; i++)
{
    cin >> from >> to >> flow;
    point[from].neighbors[to].first = flow;
}
}

void Graph::print_graph()
{
    for (auto var : point)
    {
        cout << var.first << ": ";
        for (auto var2 : var.second.neighbors)
            cout << var2.first << " " << var2.second.first << "/" << var2.second.second << " ";
        cout << std::endl;
    }
}

char Graph::min_neighbors_flow(map<char, pair<int, int>> n_mas, string n_list, char curr)
// Ищем соседнюю вершину наиболее близкую
// Возвращает либо вершину, либо '-', если поток везде нуль
{
    int mini = abs((int)n_list[0] - (int)curr);
    char min = n_list[0]; // первый элемент в мапе
    for (auto var : n_list)
    {
        if (abs((int)var - (int)curr) < mini)
        {
            min = var;
            mini = abs((int)var - (int)curr);
        }
    }
    if (n_mas[min].first != 0)
        return min;
    else
        return '-';
}

int Graph::searchMaxFlow()
{
    char curr = start;
    point[curr].markFlag = true; // метка у начальной вершины всегда активна, чтобы не выйти за пределы
    point[curr].mark.first = 99999;
    string neighbors_list; // контейнер соседей
    int sum = 0, flow;

    while (1)
    {
        for (auto var : point[curr].neighbors)
            // заполняем контейнер соседей
        {
            if (!point[var.first].markFlag && var.second.first != 0)
                neighbors_list.push_back(var.first);
        }
    }
}

```



```

//cout << neighbors_list << endl;

if (neighbors_list.empty())
{
    if (curr == start)
    {
        return sum; //конец алгоритма
    }
    else {
        curr = point[curr].mark.second; //флаг оставляем активным, чтобы не заходить больше сюда
        continue;
    }
}
char next = min_neighbors_flow(point[curr].neighbors, neighbors_list, curr);

point[next].mark = { std::min(point[curr][next].first, point[curr].mark.first), curr };
point[next].markFlag = true;
cout << "next: " << next << "; come from " << point[next].mark.second << " with stream = " <<
point[next].mark.first << endl;
curr = next;

if (curr == end)
{
    cout << "We have reached the final peak! Through way:";
    std::stack<char> out; //стек для промежуточного вывода
    sum += point[curr].mark.first;
    flow = point[curr].mark.first;
    while (curr != start)
    {
        out.push(curr);
        next = curr;
        point[curr].markFlag = false;
        curr = point[curr].mark.second;
        point[curr][next].first -= flow;
        point[curr][next].second += flow;
    }
    out.push(start);

    while (!out.empty())
    {
        cout << " " << out.top();
        out.pop();
    }
    cout << ". Current flow: " << flow << endl;
}
neighbors_list.clear();
}
}

int main()
{
    Graph one;
    one.init();
    one.print_graph();
    cout << one.searchMaxFlow() << endl;
    one.print_for_stepik();
    cout << endl << "Last version graph " << endl;
    one.print_graph();
}

```

```
    return 0;  
}
```