

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
Вариант 4

Студент гр. 8382

Ефимова М.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить работу и реализовать алгоритм Форда-Фалкерсона для нахождения максимального потока в сети.

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:

N - количество ориентированных рёбер графа

- сток

- ребро графа

- ребро графа

Выходные данные:

- величина максимального потока

- ребро графа с фактической величиной протекающего потока

- 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).

Вар. 4. Поиск в глубину. Итеративная реализация.

Описание алгоритма.

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.

2. В остаточной сети находим любой путь из источника в сток. Если такого пути нет, останавливаемся.
3. Пускаем через найденный путь максимально возможный поток:
 - 3.1. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью .
 - 3.2. Для каждого ребра на найденном пути увеличиваем поток на f , а в противоположном ему - уменьшаем на f .
 - 3.3. Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных им рёбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.
4. Возвращаемся на шаг 2.

Поиск в глубину:

1. Для реализации алгоритма используется структура данных стек. Идея алгоритма. Поиск начинается с некоторой фиксированной вершины s .
2. Далее рассматривается вершина v смежная с s .
3. Она выбирается и отмечается как посещенная.
4. Остальные смежные вершины (если они есть и они не посещены) отправляются в стек и ожидают следующего захода в родительскую вершину.
5. Далее берется вершина q смежная с v . Действия повторяются. Так процесс будет продвигаться вглубь графа пока не достигнет вершины u такой, что не окажется вершин смежных с ней и не посещенных ранее.
6. Если такая вершина получена, то осуществляется возвращение к вершине, которая была ранее (до неё) и там производится определение доступной вершины.
7. В том случае, когда мы вернулись в вершину s , а все смежные вершины с ней уже посещены то алгоритм завершает свою работу.

Описание способов хранения частичных решений.

```
struct Top {  
    char from_top;  
    char to_top;  
    int weigth;  
};
```

`vector<vector<int> > Graph` - двумерный вектор, в котором хранятся ребра.

Описание функций.

Функция `int bool One(answer a, answer b)` - компаратор для корректного вывода ребер, т.к. ребра должны быть отсортированы в лексикографическом порядке по первой вершине, потом по второй.

Функция `bool compareTop(Top a, Top b)` сортирует вершины в лексикографическом порядке по первой вершине, потом по второй

```
bool compareTop(Top a, Top b){  
    if (a.from_top < b.from_top) return true;  
    else if (a.from_top == b.from_top) {  
        if (a.to_top < b.to_top) return true;  
    }  
    return false;  
}
```

Описание функции f_Fulkerson.

Функция `int f_Fulkerson(vector<vector<int>>& graph, vector<vector<int>>& Graph, int s, int t, int U, string node)` –

на вход принимает граф `graph`, в котором хранятся ребра;

`Graph` - граф смежности,

`s` - исток, `t` - сток,

`V` - количество узлов,

`node` - названия узлов.

В начале функции `Graph` принимает значения `graph`, при этом граф `graph` обнуляется, т.к. в дальнейшем он будет использован для вывода ответа. Работа в функции производится с `Graph`.

Далее запускается цикл, который работает до тех пор, пока функция `dfs` находит путь от истока в сток в сети. Если путь найден, то он записывается в массив `parent`.

Затем просматриваются эти пути еще раз, и вычитаются из пропускной способности ребер пути значения минимальной пропускной способности и прибавляются эти значения ребрам, идущим между теми же вершинами, но в противоположную сторону.

Функция возвращает значение максимального потока в сети.

Описание функции dfs.

Функция `bool dfs(vector<vector<int> > Graph, int s, int t, vector<int>& parent, int V, string node)` на вход принимает все то же самое, что и функция `f_Fulkerson`, за исключением вектора `parent`, в который записывается путь от истока в сток.

Эта итеративная функция ищет путь обходом в глубину в сети и записывает его в массив `parent`.

Функция возвращает `true`, если путь найден, и `false`, если путь не был найден.

Сложность алгоритма по времени.

Сложность алгоритма по времени можно оценить как $O(VE^2)$

Так как каждый путь находится поиском в глубину со сложностью $O(E)$, общее число итерация в цикле `while` алгоритма не превосходит $O(VE)$, следовательно, временную сложность алгоритма можно оценить как $O(VE^2)$.

Сложность алгоритма по памяти.

Сложность алгоритма по памяти можно оценить как $O(V^2)$

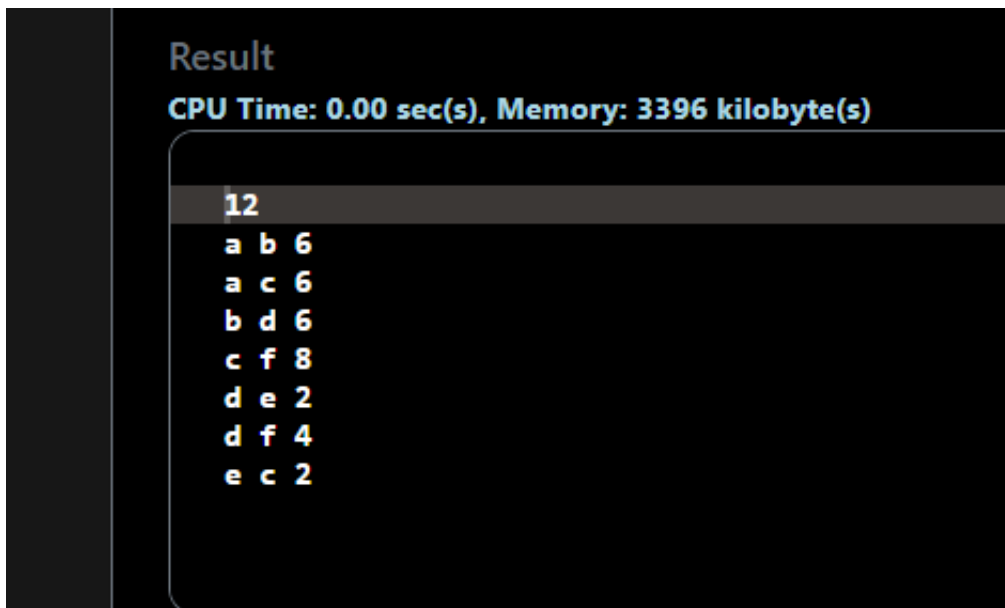
Такая оценка исходит из того, что программа хранит матрицу смежности графа.

Спецификация программы.

Программа написана на языке C++. Программа на вход получает количество ориентированных ребер графа, исток и сток. Затем вводятся ребра графа и их веса. В конце программа печатает максимальный поток в сети.

Тестирование.

Пример вывода результата для 1-го теста (читать слева направо).



```
Result
CPU Time: 0.00 sec(s), Memory: 3396 kilobyte(s)
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

```

Result
CPU Time: 0.00 sec(s), Memory: 3536 kilobyte(s)

17
a b 4
a c 13
b c 4
c d 14
c e 3
d e 14

```

№	Input	Output
1	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
2	1 a a	0

3	6	
	a	17
	e	a b 4
	a b 12	a c 13
	b c 5	b c 4
	a c 13	c d 14
	c d 14	c e 3
	c e 3	d e 14
	d e 15	

Вывод.

В ходе выполнения лабораторной работы был реализован на языке C++ алгоритм Форда-Фалкерсона для нахождения максимального потока в сети.

ПРИЛОЖЕНИЕ

КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <stack>
#include <limits.h>
#include <algorithm>
using namespace std;

//структура для вершины
struct Top{
    char from_top;//идет из вершины
    char to_top;//идет в вершину
    int weighth; //вес ребра
};

//сортировка вершин
bool compareTop(Top a, Top b){
    if (a.from_top < b.from_top) return true;
    else if (a.from_top == b.from_top) {
        if (a.to_top < b.to_top) return true;
    }
    return false;
}

//поиск в глубину
bool dfs(vector<vector<int>> Graph, int s, int t, vector<int>&parent, int U, string node){
    // массив флагов посещаемости вершин
    //создаем стек
    vector<bool> visited(U,0);
    stack <int> stak;
    //кладем исходную вершину в стек
    stak.push(s);
    //used[s] = true;
    //вектор посещенных уже вершин

    visited[s] = true;//посетили вершину
    parent[s] = -1;

    //считаем длину пути
    while (!stak.empty()) { //обработка, пока стек не пуст
        int i = stak.top(); //обработка первой вершины
        stak.pop();
        //если смежная вершина не обработана и имеет ребро с обрабатываемой вершиной
        for( int j = 0 ; j < U; j++){
            if(Graph[i][j] > 0 && visited[j] == false){
                //add смежную вершину
                stak.push(j);
                parent[j] = i;
                visited[j] = true;
            }
        }
    }
    if(visited[t] == true){
        string Sr;
        for( int i = t; i != s; i = parent[i]){
            Sr = node[i] + Sr;
        }
        Sr = node[s] + Sr;
    }
    return visited[t] == true;
}

int f_Fulkerson(vector<vector<int>>& graph, vector<vector<int>>& Graph, int s, int t, int U, string node) {
    int u, v;

    for (u = 0; u < U; u++)
        for (v = 0; v < U; v++) {
            Graph[u][v] = graph[u][v];
            graph[u][v] = 0;
        }
    //изначально поток = 0
    int max_flow = 0;
```

```

// массив для хранения пути
vector<int> parent(U, 0);
//увеличивается поток, пока есть путь от истока к стоку
while (dfs(Graph, s, t, parent, U, node)) {
    int path_flow = INT_MAX;
    for (v = t; v != s; v = parent[v]) {
        u = parent[v];
        path_flow = min(path_flow, Graph[u][v]);
    }
    //обновление пропускной способности каждого ребра
    for (v = t; v != s; v = parent[v]) {
        u = parent[v];
        Graph[u][v] -= path_flow;
        Graph[v][u] += path_flow;
        graph[u][v] += path_flow;
        graph[v][u] -= path_flow;
    }
    max_flow += path_flow;
}
return max_flow;
}

int main() {
    char start; //исток
    char finish; //сток
    char temp_from;
    char temp_to;
    int N = 0; //количество ориентированных рёбер графа
    int weighth;
    string from;
    string to;
    string node; //названия узлов
    cin >> N >> start >> finish;
    vector<int> Nw;
    node = node + start;
    for (int i = 0; i < N; i++) {
        cin >> temp_from;
        cin >> temp_to; //input
        cin >> weighth;
        from = from + temp_from;
        to = to + temp_to;
        Nw.push_back(weighth);
        if (node.length() == 0)
            node = node + temp_to;
        else if (node.find(temp_to) == string::npos) //макс/ значенит, которое может предоставить
            node = node + temp_to;
    }
    sort(node.begin(), node.end());
    int U = node.length(); //смотрим размер
    vector<vector<int>> graph(U, vector<int>(U, 0));

    //поиск всех ребер, ведущих из вершины node[q]
    for (int q = 0; q < node.length(); q++) {
        vector<int> Temp;
        for (int j = 0; j < N; j++) {
            if (from[j] == node[q]) {
                Temp.push_back(j);
            }
        }
    }
    //поиск в строке node[q] вершины, в которую ведут ребра из вектора temp
    vector<int> nodeTemp;
    for (int i = 0; i < Temp.size(); i++) {
        for (int j = 0; j < node.length(); j++) {
            if (node[j] == to[Temp[i]])
                nodeTemp.push_back(j);
        }
    }
    for (int i = 0; i < Temp.size(); i++) {
        graph[q][nodeTemp[i]] = Nw[Temp[i]];
    }
}

int start_ind = 0;
int finish_ind = 0;
for (int i = 0; i < U; i++) {

```

```

        if (node[i] == start)
            start_ind = i;
        else if (node[i] == finish)
            finish_ind = i;
    }

    vector<vector<int>> > Graph(U, vector<int>(U, 0));
    int max_flow = f_Fulkerson(graph, Graph, start_ind, finish_ind, U, node);

    vector<Top> One;
    for (int i = 0; i < U; i++){
        vector<int> pointer; //индексы
        for (int j = 0; j < N; j++) {
            if (node[i] == from[j]) pointer.push_back(j);
        }
        for (int j = 0; j < pointer.size(); j++) {
            Top tops;
            tops.from_top = from[pointer[j]];
            tops.to_top = to[pointer[j]];
            int tempF = 0;
            int tempT = 0;
            for (int k = 0; k < U; k++) {
                if (node[k] == from[pointer[j]]) tempF = k;
                else if (node[k] == to[pointer[j]]) tempT = k;
            }
            if (graph[tempT][tempF] >= 0) tops.weigth = 0;
            else tops.weigth = abs(graph[tempT][tempF]);
            One.push_back(tops);
        }
    }
    sort(One.begin(), One.end(), compareTop);
    cout << max_flow << endl;
    for (int i = 0; i < One.size(); i++){
        cout << One[i].from_top << " " << One[i].to_top << " " << One[i].weigth << endl;
    }

    return 0;
}

```