**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. 8383 |  | Шишкин И.В. |
| 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

- исток

- 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 , а в противоположном ему - уменьшаем на .

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.

1. Возвращаемся на шаг 2.

Чтобы найти кратчайший путь в графе, используется поиск в глубину:

1. Создается стек вершин О. Вначале О состоит из единственной вершины s.
2. Вершина s отмечается как посещенная, без родителя, а все остальные как не посещённые.
3. Пока стек не пуст, выполняются следующие шаги

3.1. Удаляется вершина i в стеке

3.2. Для всех дуг (i, j), исходящих из вершины i, для которых вершина j ещё не посещена, выполняются следующие шаги

3.2.1. Вершина i отмечается как посещенная с родителем j

3.2.2. Вершина v добавляется в стек

3.2.3. Если j = t (текущая вершина - сток), то производится выход из всех циклов, т.к. путь был найден

1. Если стек пуст, то возвращается ответ, что пути нет.
2. Если стек не пуст, то производится проход от t к s, каждый раз переходя к родителю. Путь возвращается в обратном порядке.

**Описание способов хранения частичных решений.**  struct answer {

char ansFrom;

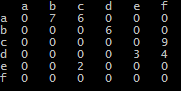
char ansTo;

int ansW;

};

Структура для ответа на степике. ansFrom - вершина, из которой идет путь; ansTo - в которую идет; ansW - вес ребра.

vector<vector<int> > graph - двумерный вектор, в котором хранятся ребра. Например, если ввести пример из степика (7; a; f; a b 7; a c 6; b d 6; c f 9; d e 3; d f 4; e c 2), то эти ребра в графе будут выглядеть следующим образом:



Для вывода графа и нужна функция void printGraph(vector<vector<int> > graph, int V, string nodeNames), которая на вход получается граф graph, количество узлов V и названия этих узлов nodeNames, хотя в программе эта функция и не используется.

vector <answer> forStepik - вектор ребер, который нужен для степика.

**Описание функции cmpForStepik.**

Функция int bool cmpForStepik(answer a, answer b) - компаратор для корректного вывода ребер на степике, т.к. стоит условие, что ребра должны быть отсортированы в лексикографическом порядке по первой вершине, потом по второй.

**Описание функции fordFulkerson.**

Функция int fordFulkerson(vector<vector<int> >& graph, vector<vector<int> >& rGraph, int s, int t, int V, string nodeNames) - на вход принимает граф graph, в котором хранятся ребра; rGraph - residual graph - граф смежности, s - исток, t - сток, V - количество узлов, nodeNames - названия узлов.

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

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> > rGraph, int s, int t, vector <int>& parent, int V, string nodeNames) на вход принимает все то же самое, что и функция fordFulkerson, 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

Так как каждый путь находится поиском в глубину со сложностью , общее число итерация в цикле while алгоритма не превосходит , следовательно, временную сложность алгоритма можно оценить как

**Сложность алгоритма по памяти.**

Сложность алгоритма по памяти можно оценить как

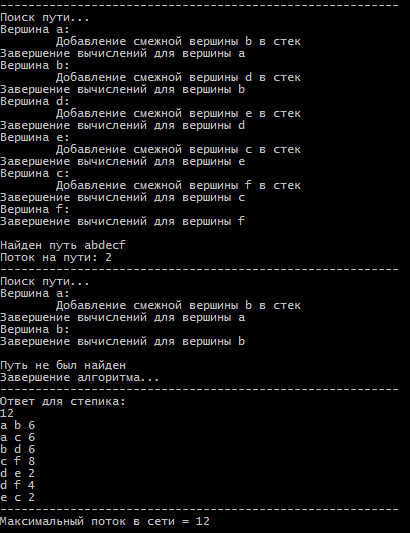
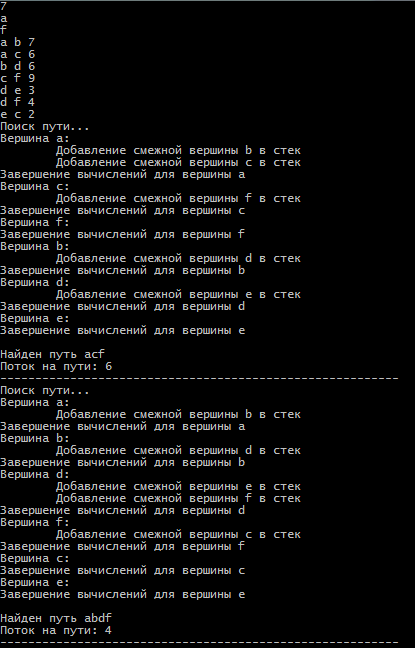
Такая оценка исходит из того, что программа хранит матрицу смежности графа.

**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).



|  |  |  |
| --- | --- | --- |
| № | 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 | Сток и исток не могут быть одной вершиной |
| 3 | 6  a  e  a b 12  b c 5  a c 13  c d 14  c e 3  d e 15 | 17  a b 4  a c 13  b c 4  c d 14  c e 3  d e 14 |

**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 <limits.h>

#include <vector>

#include <stack>

#include <iomanip>

#include <algorithm>

using namespace std;

struct answer {

char ansFrom; //вершина, из которой выходит ребро

char ansTo; //вершина, в которую входит ребро

int ansW; //вес ребра

};

bool dfs(vector<vector<int> > rGraph, int s, int t, vector <int>& parent, int V, string nodeNames); //возвращает true, если существует путь от истока s к стоку t в графе rGraph, также заполняет массив parent

int fordFulkerson(vector<vector<int> >& graph, vector<vector<int> >& rGraph, int s, int t, int V, string nodeNames); //возвращает максимальный поток от истока s к стоку t

void printGraph(vector<vector<int> > graph, int V, string nodeNames); //для печати графа

bool cmpForStepik(answer a, answer b); //компаратор для сортировки вершин

int main() {

setlocale(LC\_ALL, "RUS");

int N = 0; //количество ориентированных рёбер графа

char start; //исток

char finish; //сток

string nodeNames; //названия узлов

string from;

string to;

vector <int> w;

cin >> N;

cin >> start;

cin >> finish;

if (start == finish) {

cout << "Сток и исток не могут быть одной вершиной" << endl;

return 0;

}

char tmpFrom;

char tmpTo;

int tmpW;

nodeNames += start;

for (int i = 0; i < N; i++) {

cin >> tmpFrom >> tmpTo >> tmpW;

from += tmpFrom;

to += tmpTo;

w.push\_back(tmpW);

if (nodeNames.length() == 0) nodeNames += tmpTo;

else if (nodeNames.find(tmpTo) == string::npos) nodeNames += tmpTo;

}

sort(nodeNames.begin(), nodeNames.end());

int V = nodeNames.length();

vector<vector<int> > graph(V, vector<int>(V, 0));

string findInNodeNames;

for (int k = 0; k < nodeNames.length(); k++) {

vector <int> tmp;

for (int j = 0; j < N; j++) { //поиск всех ребер, ведущих из вершины nodeNames[k]

if (from[j] == nodeNames[k]) {

tmp.push\_back(j);

}

}

vector <int> nodeTmp;

for (int i = 0; i < tmp.size(); i++) { //поиск в строке nodeNames[k] вершины, в которую ведут ребра из вектора tmp

for (int j = 0; j < nodeNames.length(); j++) {

if (nodeNames[j] == to[tmp[i]])

nodeTmp.push\_back(j);

}

}

for (int i = 0; i < tmp.size(); i++) {

graph[k][nodeTmp[i]] = w[tmp[i]];

}

}

int startIndex = 0;

int finishIndex = 0;

for (int i = 0; i < V; i++) {

if (nodeNames[i] == start) startIndex = i;

else if (nodeNames[i] == finish) finishIndex = i;

}

vector<vector<int> > rGraph(V, vector<int>(V, 0));

int maxFlow = fordFulkerson(graph, rGraph, startIndex, finishIndex, V, nodeNames);

vector <answer> forStepik;

for (int i = 0; i < V; i++) {

vector <int> indices; //индексы

for (int j = 0; j < N; j++) {

if (nodeNames[i] == from[j]) indices.push\_back(j);

}

for (int j = 0; j < indices.size(); j++) {

answer ans;

ans.ansFrom = from[indices[j]];

ans.ansTo = to[indices[j]];

int tmpF = 0;

int tmpT = 0;

for (int k = 0; k < V; k++) {

if (nodeNames[k] == from[indices[j]]) tmpF = k;

else if (nodeNames[k] == to[indices[j]]) tmpT = k;

}

if (graph[tmpT][tmpF] >= 0) ans.ansW = 0;

else ans.ansW = abs(graph[tmpT][tmpF]);

forStepik.push\_back(ans);

}

}

sort(forStepik.begin(), forStepik.end(), cmpForStepik);

cout << "Ответ для степика:" << endl;

cout << maxFlow << endl;

for (int i = 0; i < forStepik.size(); i++) cout << forStepik[i].ansFrom << " " << forStepik[i].ansTo << " " << forStepik[i].ansW << endl;

cout << "---------------------------------------------------------" << endl;

cout << "Максимальный поток в сети = " << maxFlow << endl;

return 0;

}

bool dfs(vector<vector<int> > rGraph, int s, int t, vector <int>& parent, int V, string nodeNames) { //функция поиска пути в глубину

vector <bool> visited(V, 0); //вектор посещенных вершин (если 0, то не посещена)

stack <int> st; //создается стек, в который кладется исток, и начальная вершина помечается как посещенная

st.push(s);

visited[s] = true;

parent[s] = -1;

cout << "Поиск пути..." << endl;

while (!st.empty()) { //обработка, пока стек не пуст

int i = st.top(); //обработка первой вершины

st.pop();

cout << "Вершина " << nodeNames[i] << ":" << endl;

for (int j = 0; j < V; j++) { //если смежная вершина не обработана и имеет ребро с обрабатываемой вершиной

if (visited[j] == false && rGraph[i][j] > 0) {

st.push(j);

parent[j] = i;

visited[j] = true;

cout << "\tДобавление смежной вершины " << nodeNames[j] << " в стек" << endl;

}

}

cout << "Завершение вычислений для вершины " << nodeNames[i] << endl;

}

cout << endl;

if (visited[t] == true) {

cout << "Найден путь ";

string str;

for (int i = t; i != s; i = parent[i])

str += nodeNames[i];

str += nodeNames[s];

for (int i = str.length() - 1; i >= 0; i--)

cout << str[i];

cout << endl;

}

else cout << "Путь не был найден" << endl;

return (visited[t] == true); //если был достигнут сток, то возвращается true, иначе - false

}

int fordFulkerson(vector<vector<int> >& graph, vector<vector<int> >& rGraph, int s, int t, int V, string nodeNames) {

int u, v;

for (u = 0; u < V; u++)

for (v = 0; v < V; v++) {

rGraph[u][v] = graph[u][v];

graph[u][v] = 0;

}

vector <int> parent(V, 0); //этот массив заполняется функцией BFS и создан для хранения пути

int max\_flow = 0; //изначально поток = 0

while (dfs(rGraph, s, t, parent, V, nodeNames)) { //увеличивается поток, пока есть путь от истока к стоку

int path\_flow = INT\_MAX;

for (v = t; v != s; v = parent[v]) {

u = parent[v];

path\_flow = min(path\_flow, rGraph[u][v]);

}

cout << "Поток на пути: " << path\_flow << endl;

cout << "---------------------------------------------------------" << endl;

for (v = t; v != s; v = parent[v]) { //обновление пропускной способности каждого ребра

u = parent[v];

rGraph[u][v] -= path\_flow;

rGraph[v][u] += path\_flow;

graph[u][v] += path\_flow;

graph[v][u] -= path\_flow;

}

max\_flow += path\_flow;

}

cout << "Завершение алгоритма..." << endl;

cout << "---------------------------------------------------------" << endl;

return max\_flow;

}

void printGraph(vector<vector<int> > graph, int V, string nodeNames) {

cout << " ";

for (int i = 0; i < V; i++) {

cout << setw(3) << nodeNames[i] << " ";

}

cout << endl;

for (int i = 0; i < V; i++) {

cout << nodeNames[i];

for (int j = 0; j < V; j++) cout << setw(3) << graph[i][j] << " ";

cout << endl;

}

cout << endl;

}

bool cmpForStepik(answer a, answer b) {

if (a.ansFrom < b.ansFrom) return true;

else if (a.ansFrom == b.ansFrom) {

if (a.ansTo < b.ansTo) return true;

}

return false;

}