**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 5

|  |  |  |
| --- | --- | --- |
| Студент гр. 8383 |  | Бабенко Н.С. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2020

**Цель работы.**

Изучение алгоритма Форда-Фалкерсона для поиска максимального потока в графе.

**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  
v0 - исток  
vn - сток  
vi vj ωij - ребро графа  
vi vj ωij - ребро графа  
...

Выходные данные:  
Pmax - величина максимального потока  
vi vj ωij - ребро графа с фактической величиной протекающего потока  
vi vj ωij - 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).

Пример входных данных

7

a

f

a b 7

a c 6

b d 6

c f 9

d e 3

d f 4

e c 2

Пример выходных данных

12

a b 6

a c 6

b d 6

c f 8

d e 2

d f 4

e c 2

**Индивидуализация.**

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.**

Остаточная сеть — это граф Gf = (V, Ef), где Ef - множество ребер с положительной остаточной пропускной способностью. В остаточной сети может быть путь из u в v, даже если его нет в исходном графе. Это выполняется, когда в исходной сети есть обратный путь (v, u) и поток по нему положительный. Дополняющий путь — это путь в остаточной сети.

В программе используется алгоритм Форда-Фалкерсона. Алгоритм подразумевает запуск поиска в глубину в остаточной сети до тех пор, пока возможно найти путь от истока к стоку.

С самого начала остаточная сеть – исходный граф. На каждом шаге находится путь от истока к стоку, при этом смежные вершины выбираются в порядке уменьшения остаточной пропускной способности C. В пути выбирается ребро с наименьшей C (далее Cmin). Для каждого ребра пути C уменьшается на Cmin, строится обратное ребро и его пропускной способности прибавляется Cmin. Если путь от истока к стоку найден не был, то значит максимальный поток был найден и алгоритм завершает свою работу. Максимальный поток в сети является суммой всех максимальных пропускных способностей дополняющих путей.

**Описание структур данных.**

**struct Edge {**

**int resultedCapacity;**

**int reversedFlow;**

**}** - структура для хранения ребер.

resultedCapacity — остаточная пропускная способность ребра

reversedFlow — обратный поток

**map<char, map<char, Edge>> network** – остаточная сеть графа.

Используется для хранения информации о графе в виде матрицы смежности. Для каждой вершины хранится карта «смежная вершина-ребро».

**vector <bool> used** — контейнер для того, чтобы отмечать посещенные вершины в поиске в глубину.

**set<pair<char, char>> graph** — контейнер для хранения списка смежности графа. Используется для упрощения сортировки выходных данных.

**set<pair<int, char>> toVisit** — контейнер для сортировки смежных вершин по остаточной пропускной способности.

**Описание функций.**

**void readGraph()**

Функция для чтения графа. Заполняет контейнеры graph и network.

**int networkTraversal(char v, int delta)**

v — вершина с которой начинаем поиск.

delta — текущая минимальная остаточная пропускная способность.

Рекурсивная функция обхода графа по заданию, заданному в индивидуализации: 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.

Возвращает минимальную пропускную способность на пути.

**void printFlow()**

Функция вывода текущего потока в консоль.

**void startFordFulkerson()**

Функция поиска максимального потока в графе.

**Сложность алгоритма.**

Время работы алгоритма ограничено O(V\*f\*E\*logE), где E — число рёбер в графе, V — число вершин, f — максимальный поток в графе, так как для каждой вершины сортируются смежные за O(E\*logE), в таком случае каждый увеличивающий путь в худшем случае находится за O(V\*E\*logE) и увеличивает поток как минимум на 1.

Для работы алгоритма хранится граф в виде матрицы смежности (O(V2)), остаточная сеть (также O(V2)) и вектор посещенных вершин (O(V)). в итоге получаем сложность по памяти O(V2).

**Пример работы с подробным выводом**

11

a

f

a b 7

a c 3

a d 5

c b 4

c f 5

b f 6

b d 3

b e 4

d b 7

d e 8

e f 10

==================

current flow = 6

a b 6(resulted capacity 1, max capacity 7)

a c 0(resulted capacity 3, max capacity 3)

a d 0(resulted capacity 5, max capacity 5)

b d 0(resulted capacity 3, max capacity 3)

b e 0(resulted capacity 4, max capacity 4)

b f 6(resulted capacity 0, max capacity 6)

c b 0(resulted capacity 4, max capacity 4)

c f 0(resulted capacity 5, max capacity 5)

d b 0(resulted capacity 7, max capacity 7)

d e 0(resulted capacity 8, max capacity 8)

e f 0(resulted capacity 10, max capacity 10)

==================

current flow = 11

a b 6(resulted capacity 1, max capacity 7)

a c 0(resulted capacity 3, max capacity 3)

a d 5(resulted capacity 0, max capacity 5)

b d 0(resulted capacity 3, max capacity 3)

b e 0(resulted capacity 4, max capacity 4)

b f 6(resulted capacity 0, max capacity 6)

c b 0(resulted capacity 4, max capacity 4)

c f 0(resulted capacity 5, max capacity 5)

d b 0(resulted capacity 7, max capacity 7)

d e 5(resulted capacity 3, max capacity 8)

e f 5(resulted capacity 5, max capacity 10)

==================

current flow = 14

a b 6(resulted capacity 1, max capacity 7)

a c 3(resulted capacity 0, max capacity 3)

a d 5(resulted capacity 0, max capacity 5)

b d 0(resulted capacity 3, max capacity 3)

b e 0(resulted capacity 4, max capacity 4)

b f 6(resulted capacity 0, max capacity 6)

c b 0(resulted capacity 4, max capacity 4)

c f 3(resulted capacity 2, max capacity 5)

d b 0(resulted capacity 7, max capacity 7)

d e 5(resulted capacity 3, max capacity 8)

e f 5(resulted capacity 5, max capacity 10)

==================

current flow = 15

a b 7(resulted capacity 0, max capacity 7)

a c 3(resulted capacity 0, max capacity 3)

a d 5(resulted capacity 0, max capacity 5)

b d 0(resulted capacity 3, max capacity 3)

b e 1(resulted capacity 3, max capacity 4)

b f 6(resulted capacity 0, max capacity 6)

c b 0(resulted capacity 4, max capacity 4)

c f 3(resulted capacity 2, max capacity 5)

d b 0(resulted capacity 7, max capacity 7)

d e 5(resulted capacity 3, max capacity 8)

e f 6(resulted capacity 4, max capacity 10)

==================

current flow = 15

a b 7(resulted capacity 0, max capacity 7)

a c 3(resulted capacity 0, max capacity 3)

a d 5(resulted capacity 0, max capacity 5)

b d 0(resulted capacity 3, max capacity 3)

b e 1(resulted capacity 3, max capacity 4)

b f 6(resulted capacity 0, max capacity 6)

c b 0(resulted capacity 4, max capacity 4)

c f 3(resulted capacity 2, max capacity 5)

d b 0(resulted capacity 7, max capacity 7)

d e 5(resulted capacity 3, max capacity 8)

e f 6(resulted capacity 4, max capacity 10)

<!New path not found!>

Resulting max flow: 15

a b 7(resulted capacity 0, max capacity 7)

a c 3(resulted capacity 0, max capacity 3)

a d 5(resulted capacity 0, max capacity 5)

b d 0(resulted capacity 3, max capacity 3)

b e 1(resulted capacity 3, max capacity 4)

b f 6(resulted capacity 0, max capacity 6)

c b 0(resulted capacity 4, max capacity 4)

c f 3(resulted capacity 2, max capacity 5)

d b 0(resulted capacity 7, max capacity 7)

d e 5(resulted capacity 3, max capacity 8)

e f 6(resulted capacity 4, max capacity 10)

Тестирование.

|  |  |
| --- | --- |
| Input | Output |
| 11  a  f  a b 7  a c 3  a d 5  c b 4  c f 5  b f 6  b d 3  b e 4  d b 7  d e 8  e f 10 | Resulting max flow: 15  a b 7  a c 3  a d 5  b d 0  b e 1  b f 6  c b 0  c f 3  d b 0  d e 5  e f 6 |
| 11  a  h  a b 3  b e 1  a c 1  c e 2  a d 2  d e 4  e g 3  e f 2  f h 3  g h 1  d f 1 | Resulting max flow: 4  a b 1  a c 1  a d 2  b e 1  c e 1  d e 1  d f 1  e f 2  e g 1  f h 3  g h 1 |
| 4  a  d  a c 1  a b 1  c b 1  b c 1 | Resulting max flow: 0  a b 0  a c 0  b c 0  c b 0 |
| 7  a  f  a b 7  b d 6  d e 3  e c 2  a c 6  c f 9  d f 4 | Resulting max flow: 12  a b 6  a c 6  b d 6  c f 8  d e 2  d f 4  e c 2 |
| 6  a  a  a c 10  c d 10  c b 1  b c 1  a b 10  b d 10 | Resulting max flow: 0  a b 0  a c 0  b c 0  b d 0  c b 0  c d 0 |

Вывод.

В ходе выполнения лабораторной работы были изучен и запрограммирован алгоритм Форда-Фалкерсона для поиска максимального потока в графе.

Приложения А. Исходный код

#include <iostream>

#include <algorithm>

#include <vector>

#include <climits>

#include <set>

#include <map>

using namespace std;

//структура ребра, хранит остаточную пропускную способность и поток, который можно пустить обратно

struct Edge {

int resultedCapacity;

int reversedFlow;

};

//остаточная сеть

map<char, map<char, Edge>> network;

//сет ребер

set<pair<char, char>> graph;

//вектор посещенных вершин

vector<bool> used;

char source, sink;

//функция для вывода найденного потока

void printFlow() {

for (auto &i : graph) {

cout << i.first << ' ' << i.second << ' ' << network[i.second][i.first].reversedFlow;

cout << "(resulted capacity " << network[i.first][i.second].resultedCapacity

<< ", max capacity " << network[i.second][i.first].reversedFlow + network[i.first][i.second].resultedCapacity << ')';

cout << endl;

}

}

//функция чтение графа из консоли. Заполняет граф и остаточную сеть

void readGraph() {

int n;

char u, v;

int weight;

cin >> n;

cin >> source >> sink;

//128 - символов в таблице ascii 128

used.resize(128);

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

cin >> u >> v >> weight;

graph.insert({u, v});

network[u][v].resultedCapacity = weight;

if (network.find(v) != network.end() && network[v].find(u) == network[v].end()){

network[v][u].resultedCapacity = 0;

}

}

}

//Функция для обхода сети по правилу из варианта индивидуализации:

/\*

\* 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.

\*/

int networkTraversal(char v, int delta) {

//если вершина уже была посещена, выходим из нее

if (used[v])

return 0;

used[v] = true;

//если текущая вершина - сток, выходим из нее

if (v == sink)

return delta;

//множество смежных вершин, сортированное по остаточной пропускной способности

set<pair<int, char>> toVisit;

for (auto u : network[v]) {

if (!used[u.first])

toVisit.insert({max(u.second.resultedCapacity, u.second.reversedFlow), u.first});

}

//обходим вершины из множества в порядке убывания остаточной пропускной способности

for (auto u = toVisit.rbegin(); u != toVisit.rend(); u++) {

//если есть поток который можно пустить обратно,

//находим минимальный вес ребра в пути и делаем это

if (network[v][u->second].reversedFlow > 0) {

int newDelta = networkTraversal(u->second, min(delta, network[v][u->second].reversedFlow));

if (newDelta > 0) {

network[u->second][v].resultedCapacity += newDelta;

network[v][u->second].reversedFlow -= newDelta;

return newDelta;

}

}

//если остаточная пропускная способность больше нуля,

//находим минимальный вес ребра в пути и пускаем поток по этому ребру

if (network[v][u->second].resultedCapacity > 0) {

int newDelta = networkTraversal(u->second, min(delta, network[v][u->second].resultedCapacity));

if (newDelta > 0) {

network[u->second][v].reversedFlow += newDelta;

network[v][u->second].resultedCapacity -= newDelta;

return newDelta;

}

}

}

return 0;

}

//запуск поиска алгоритмом Форда-Фалкерсона

void startFordFulkerson() {

int flow = 0;

int ans = 0;

while (true) {

//обнуляем вектор посещенных вершин

used.clear();

used.resize(128);

//запускаем поиск в глубину

flow = networkTraversal(source, INT\_MAX);

//если путь не найден - выходим

cout << "==================\n";

cout << "current flow = " << ans+flow << endl;

printFlow();

if (flow == 0 || flow == INT\_MAX){

cout << "<!New path not found!>\n";

break;

}

//обновляем максимальный поток

ans += flow;

}

cout << "Resulting max flow: ";

cout << ans << endl;

printFlow();

}

int main() {

readGraph();

startFordFulkerson();

return 0;

}