**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: **Поток в сети**

|  |  |  |
| --- | --- | --- |
| Студент гр. 7383 |  | Лосев М.Л. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2018

## Постановка задачи.

**Цель работы:** реализация и исследование алгоритма Форда-Фалкерсона, получение опыта и знания по его использованию. Формулировка задачи:

**Формулировка задачи:** Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

**Индивидуализация:** для поиска пути использовать поиск в ширину.

**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).

**Описание алгоритма:**

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.
2. В остаточной сети находим *кратчайший* путь из источника в сток. Если такого пути нет, останавливаемся.
3. Пускаем через найденный путь (он называется **увеличивающим путём** или **увеличивающей цепью**) максимально возможный поток:
   1. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью {\displaystyle c\_{\min }}*с*.
   2. Для каждого ребра на найденном пути увеличиваем поток на {\displaystyle c\_{\min }} *с*, а в противоположном ему — уменьшаем на {\displaystyle c\_{\min }} *с*.
   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.

Чтобы найти кратчайший путь в графе, используем [поиск в ширину](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%B2_%D1%88%D0%B8%D1%80%D0%B8%D0%BD%D1%83):

1. Создаём очередь вершин *О*. Вначале *О* состоит из единственной вершины *s*.
2. Отмечаем вершину *s* как посещённую, без родителя, а все остальные как непосещённые.
3. Пока очередь не пуста, выполняем следующие шаги:
   1. Удаляем первую в очереди вершину *u*.
   2. Для всех дуг (*u*, *v*), исходящих из вершины *u*, для которых вершина *v* ещё не посещена, выполняем следующие шаги:
      1. Отмечаем вершину *v* как посещённую, с родителем *u*.
      2. Добавляем вершину *v* в конец очереди.
      3. Если *v* = *t*, выходим из обоих циклов: мы нашли кратчайший путь.
4. Если очередь пуста, возвращаем ответ, что пути нет вообще и останавливаемся.
5. Если нет, идём от *t* к *s*, каждый раз переходя к родителю. Возвращаем путь в обратном порядке.

**Реализация**

Был использован следующие классы:

Класс Edge:

Поля:

ver\_type first;

ver\_type last;

flow\_type capacity;

flow\_type flow;

flow\_type rev\_flow;

Методы:

Edge()

Edge(ver\_type first, ver\_type last, flow\_type capacity)

void print() – вывод ребра на экран

bool empty() – возвращает true тогда и только тогда когда ребро не пусто flow\_type max\_flow() – возвращает максимальный поток, который может пройти по ребру

int first\_index() – возвращает первый идекс ребра в матрице

int last\_index() – возвращает второй идекс ребра в матрице

Класс Vertex:

Поля:

ver\_type name;

ver\_type prev;

bool visited;Методы:

Методы:

Vertex()

Vertex(ver\_type name)

Vertex(ver\_type name, ver\_type prev)

bool operator < (const Vertex &v2) const

bool operator == (const Vertex &v2) const

– операторы сравнения

Класс FlowNetwork:

Поля:

int num\_of\_vertexes;

Edge \*\* adjacency\_matrix;

Vertex sourse;

Vertex sink;

vector <Vertex> vertexex;

int num\_of\_edges;

Методы:

FlowNetwork()

~FlowNetwork()

void input\_flow\_network() – ввод сети

void print\_actual\_flow\_value() – вывод всех ребер с потоками

string BFS(Vertex start, Vertex target) – поиск у ширину

int FFA – реализация алгоритма [Форда - Фалкерсона](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A4%D0%BE%D1%80%D0%B4%D0%B0_%E2%80%94_%D0%A4%D0%B0%D0%BB%D0%BA%D0%B5%D1%80%D1%81%D0%BE%D0%BD%D0%B0)

void add\_edge(const Edge &e) – добавление ребра

Edge input\_edge() – ввод одного ребра

void clear\_visits() – обнуляет посещенности всех вершин

void zeroize\_flows() – обнуляет потоки

flow\_type edge\_max\_flow(const Edge &e) const – возвращает максимальный поток по ребру

flow\_type path\_max\_flow(string path) – возвращает максимальный поток по пути

void add\_flow(string path, flow\_type flow\_val) – добавляет поток

## Исследование

Пусть |Е| - множество ребер графа, |V| – множество его вершин. Алгоритм в худшем случае пройдет каждую вершину. При прохождении каждой вершины в очередь будут добавляться пути через эту вершину в смежные с ней из начальной. Добавление в очередь требует в худшем случае столько сравнений, сколько в очереди элементов. Но элементов в очереди не может быть больше, чем |V|, если не хранить несколько путей в одну и ту же вершину, что бессмысленно (для оптимальности достаточно рассматривать лишь более короткий из них). Получается О(|V|\*|V|). Но, с другой стороны, очевидно, что на самом деле эта величина должна зависеть не только от |V|, но и от |Е|. Некоторые исследователи полагают, что сложность алгоритма А\* составляет О(|V|\*|V|\*log(|Е|).

# **Тестирование.**

Тестирование проводилось в ОС Ubuntu 18.04 компилятором GCC. Кроме того, программа прошла тесты на Stepic. Результаты тестирования показали, что программа работает корректно.

## Вывод.

При выполнении работы был реализован и изучен Эдмондса – Карпа, который является частным случаем алгоритма  [Форда — Фалкерсона](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A4%D0%BE%D1%80%D0%B4%D0%B0_%E2%80%94_%D0%A4%D0%B0%D0%BB%D0%BA%D0%B5%D1%80%D1%81%D0%BE%D0%BD%D0%B0), который использует поиск в ширину для нахождения пути.

**ПРИЛОЖЕНИЕ А  
Код программы**

#include <iostream>

#include <queue>

#include <string>

#include <vector>

#include <algorithm> // for\_each

#include <cmath>

using namespace std;

template <typename ver\_type, typename flow\_type>

class FlowNetwork

{

public:

class Edge

{

public:

ver\_type first;

ver\_type last;

flow\_type capacity;

flow\_type flow;

flow\_type rev\_flow;

Edge() { }

Edge(ver\_type first, ver\_type last, flow\_type capacity) : first(first), last(last), capacity(capacity), flow(0), rev\_flow(0) { }

void print() { cout << first << " " << last << " " << flow << endl; }

bool empty() { return capacity == 0 && flow == 0 ? true : false; }

flow\_type max\_flow() { return capacity - flow; }

int first\_index() const { return first; }

int last\_index() const { return last; }

};

class Vertex

{

public:

ver\_type name;

ver\_type prev;

bool visited;

Vertex() { }

Vertex(ver\_type name) : name(name), visited(false) { }

Vertex(ver\_type name, ver\_type prev) : name(name), prev(prev), visited(false) { }

int index() const { return name; }

bool operator < (const Vertex &v2) const { return this->name < v2.name; }

bool operator == (const Vertex &v2) const { return this->name == v2.name; } // needed to use find func

};

int num\_of\_vertexes;

Edge \*\* adjacency\_matrix;

Vertex sourse;

Vertex sink;

vector <Vertex> vertexex;

int num\_of\_edges;

FlowNetwork() : num\_of\_vertexes(256) { // any ascii character can be the name of a vertex

for (int i = 0; i < num\_of\_vertexes; i++) {

vertexex.push\_back(Vertex(i, 0));

} // fill vertexes list with vertexes from a to z with no parent(prev)

try {

adjacency\_matrix = new Edge\* [num\_of\_vertexes];

for (int i = 0; i < num\_of\_vertexes; i++){

adjacency\_matrix[i] = new Edge [num\_of\_vertexes];

fill(adjacency\_matrix[i], adjacency\_matrix[i] + num\_of\_vertexes, Edge(0, 0, 0));

}

}

catch (...) {

if (adjacency\_matrix) {

for (int i = 0; i < num\_of\_vertexes; i++)

if (adjacency\_matrix[i])

delete [] adjacency\_matrix[i];

delete [] adjacency\_matrix;

num\_of\_vertexes = 0;

throw;

}

}

}

void input\_flow\_network() {

ver\_type tmp\_sourse, tmp\_sink;

cin >> num\_of\_edges;

cin >> tmp\_sourse;

cin >> tmp\_sink;

sourse = Vertex(tmp\_sourse);

sink = Vertex(tmp\_sink);

for (int i = 0; i < num\_of\_edges; i++){

ver\_type tmp\_first, tmp\_last;

flow\_type tmp\_capacity;

cin >> tmp\_first >> tmp\_last >> tmp\_capacity;

Edge e(tmp\_first, tmp\_last, tmp\_capacity);

adjacency\_matrix[e.first\_index()][e.last\_index()] = e;

}

}

void print\_actual\_flow\_value() {

if (adjacency\_matrix)

for (int i = 0; i < num\_of\_vertexes; i++)

if (adjacency\_matrix[i])

for (int j = 0; j < num\_of\_vertexes; j++)

if (!adjacency\_matrix[i][j].empty())

adjacency\_matrix[i][j].print();

}

~FlowNetwork() {

if (adjacency\_matrix) {

for (int i = 0; i < num\_of\_vertexes; i++)

if (adjacency\_matrix[i])

delete [] adjacency\_matrix[i];

delete [] adjacency\_matrix;

}

num\_of\_vertexes = 0;

}

string BFS(Vertex start, Vertex target) {

clear\_visits(); // set no parenst and no visits

queue <Vertex> ver\_queue;

ver\_queue.push(start); // 0 means no parent

bool tag = true;

while (!ver\_queue.empty() && tag){

Vertex curr\_ver = ver\_queue.front();

ver\_queue.pop();

for (int i = 0; i < num\_of\_vertexes; i++) { // for each vertex

Edge dir\_edge = adjacency\_matrix[curr\_ver.index()][i]; // curr\_edge

Edge rev\_edge = adjacency\_matrix[i][curr\_ver.index()];

//calculate max flow through the edge:

flow\_type max\_flow = 0;

if (!dir\_edge.empty())

max\_flow += dir\_edge.max\_flow();

if (!rev\_edge.empty())

max\_flow += rev\_edge.flow;

if (max\_flow > 0) {

Vertex tmp\_next;

if (!dir\_edge.empty()) // if the flow goes through the direct edge or both edges

tmp\_next = Vertex(dir\_edge.last);

else // if the flow goes through the reverse edge

tmp\_next = Vertex(rev\_edge.first);

auto next\_ver = find(vertexex.begin(), vertexex.end(), Vertex(tmp\_next));

if (!next\_ver->visited && max\_flow != 0) {

next\_ver->visited = true;

next\_ver->prev = curr\_ver.name;

ver\_queue.push(\*next\_ver); // push the next\_ver to queue with prev = curr

if (\*next\_ver == target) { // if a way have been found

tag = false;

break; // leave both cycles

}

}

}

}

}

if (ver\_queue.empty()) // no path case

return string("");

// restore the path:

auto curr\_ver = find(vertexex.begin(), vertexex.end(), Vertex(target));

string path("");

path = curr\_ver->name + path;;

while (curr\_ver != find(vertexex.begin(), vertexex.end(), Vertex(start))) {

curr\_ver = find(vertexex.begin(), vertexex.end(), Vertex(curr\_ver->prev));

path = curr\_ver->name + path;

}

return path;

}

int FFA() {

zeroize\_flows();

sourse.visited = true;

flow\_type common\_flow = 0;

while (true) {

string path = BFS(sourse, sink);

auto flow\_val = path\_max\_flow(path);

common\_flow += flow\_val;

add\_flow(path, flow\_val);

path = BFS(sourse, sink);

if (path == "") break;

}

return common\_flow;

}

private:

void add\_edge(const Edge &e) {

adjacency\_matrix[e.first\_index()][e.last\_index()] = e;

}

Edge input\_edge() {

ver\_type tmp\_first, tmp\_last;

flow\_type tmp\_capacity;

cin >> tmp\_first >> tmp\_last >> tmp\_capacity;

return Edge(tmp\_first, tmp\_last, (flow\_type)tmp\_capacity);

}

void clear\_visits() { // for BFS

for (int i = 0; i < num\_of\_vertexes; i++) {

vertexex[i].visited = false;

vertexex[i].prev = 0;

}

} // delete parents and visits

void zeroize\_flows() {

for (int i = 0; i < num\_of\_vertexes; i++)

for (int j = 0; j < num\_of\_vertexes; j++)

if (!adjacency\_matrix[i][j].empty())

adjacency\_matrix[i][j].flow = 0;

} // set all the flows zero

flow\_type edge\_max\_flow(const Edge &e) const {

return edge\_max\_flow(Edge(e.first, e.last, 0));

}

flow\_type edge\_max\_flow(ver\_type first, ver\_type last) const {

Edge dir\_edge = adjacency\_matrix[Vertex(first).index()][Vertex(last).index()];

Edge rev\_edge = adjacency\_matrix[Vertex(last).index()][Vertex(first).index()];

flow\_type max\_flow = 0;

if (!dir\_edge.empty())

max\_flow += dir\_edge.max\_flow();

if (!rev\_edge.empty())

max\_flow += rev\_edge.flow;

return max\_flow;

}

flow\_type path\_max\_flow(string path) {

flow\_type max\_flow = edge\_max\_flow(path[0], path[1]);

for (int i = 0; i < path.length() - 1; i++) {

flow\_type curr\_flow = edge\_max\_flow(path[i], path[i + 1]);

if (curr\_flow < max\_flow)

max\_flow = curr\_flow;

}

return max\_flow;

}

void add\_flow(string path, flow\_type flow\_val) {

for (int i = 0; i < path.length() - 1; i++) { // for each vertex in the path

Edge\* dir\_edge = &adjacency\_matrix[Vertex(path[i]).index()][Vertex(path[i + 1]).index()];

Edge\* rev\_edge = &adjacency\_matrix[Vertex(path[i + 1]).index()][Vertex(path[i]).index()];

flow\_type max\_flow = edge\_max\_flow(path[i], path[i + 1]);

flow\_type tmp\_flow = flow\_val;

if (!rev\_edge->empty())

if (rev\_edge->flow <= tmp\_flow) {

tmp\_flow -= rev\_edge->flow;

rev\_edge->flow = 0;

} // first part of a flow: through the reverse edge

if (!dir\_edge->empty() && tmp\_flow != 0)

if (dir\_edge->max\_flow() >= tmp\_flow) {

dir\_edge->flow += tmp\_flow;

tmp\_flow = 0;

} // second part of a flow: through the direct edge

if (tmp\_flow != 0) // as a precaution

cout << "ERROR: sum.(rev + dir) capacity > flow\_val" << endl;

}

}

};

int main()

{

FlowNetwork <char, int> g;

g.input\_flow\_network();

cout << g.FFA() << endl;

g.print\_actual\_flow\_value();

return 0;

}