**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра математического обеспечения и применения ЭВМ**

отчет

**по практической работе №3**

**по дисциплине «Построение и анализ алгоритмов»**

Тема: **Потоки в сети**

|  |  |  |
| --- | --- | --- |
| Студент гр. 7383 |  | Кирсанов А.Я. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2019

**Постановка задачи.**

**Цель работы.**

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

Вариант 4с. Представление графа с помощью списка смежности. Поиск пути по правилу: каждый раз выполняется переход по ребру, соединяющему вершины, имена которых в алфавите ближе всего друг к другу.

**Реализация задачи.**

Была создана структура node вершины графа со следующими полями:

**char value** – значение вершины.

**map<node\*, pair<int, int>> neighbours** – структура, хранящая указатели на соседние вершины и пару: (остаточный поток в ребре; максимальный поток в ребре) для соответствующей соседней вершины.

Также был создан класс **List** со следующими полями:   
 **node\* source –** указатель на исток сети.

**node\* stock** – указатель на сток сети.

**map<node\*, node\*>** –контейнер, хранящий путь из стока в исток.

**map<char, node\*>** –контейнер, хранящий указатели вершин по их значениям.

**map<char, map<char, int>> sorted** – контейнер, в котором хранятся введенные пользователем ребра, отсортированные в лексикографическом порядке по первой и второй вершинам.

**multimap<char, pair<node\*, node\*>> toprint –** контейнер, использующийся в функции print для вывода фактического потока в ребрах.

**map<char, bool> viewed** – контейнер, хранящий просмотренные вершины.

В классе List реализованы следующие функции:

**void read()** – функция считывает вводимые пользователем ребра, сортирует их в лексикографическом порядке по первой и второй вершинам и создает список, содержащий введенные вершины, указатели на соседей для каждой введенной вершины и инцидентные им ребра.

**node\* ws(char from)** – функция поиска пути по правилу: каждый раз выполняется переход по ребру, соединяющему вершины, имена которых в алфавите ближе всего друг к другу. Функция рекурсивная. На вход принимается вершина, она помечается как просмотренная, затем рассматриваются её соседи в порядке возрастания лексикографической удаленности (рассматриваются вершины, к которым ведут как прямые, так и обратные ребра), то есть сначала рассматривается самая лексикографические близкая, которая служит аргументом для рекурсивного вызова функции **ws** (если поток через инцидентное ей ребро не 0). Если вызов функции вернул нулевой указатель, вызывается следующая по удаленности вершина. **ws** возвращает нулевой указатель, если путь в сток не найден. Если путь в сток найден, **ws** для каждой вызванной вершины вернет указатель на её соседа, через который проходит путь из истока в сток.

**int fordFulkerson()** – функция, реализующая алгоритм Форда-Фалкерсона. Возвращает максимальный поток в сети. Вызывается функция поиска пути ws до тех пор, пока она не вернет нулевой указатель. В найденном пути way рассматриваются все ребра из которых выбирается то, через которое проходит минимальный поток. Затем на найденном пути последовательно берутся ребра, ведущие из стока в исток. Для каждого ребра поток уменьшается на минимальный на данном пути, а для противоположного ребра – увеличивается на минимальный поток. Когда все ребра пройдены, к максимальному потоку в сети прибавляется минимальный на данном пути. На некоторой итерации нельзя будет найти пути из истока в сток, в котором не было бы ребра с нулевым потоком через него. В таком случае функция **ws** вернет нулевой указатель и алгоритм Форда-Фалкерсона вернет максимальный найденный поток.

**void print()** – функция, выводящая фактические потоки ребер сети, отсортированные в лексикографическом порядке по первой и второй вершинам. Функция проходит по контейнеру **toprint**, выводит две вершины, затем для ребра между ними выводит разность максимального потока в ребре и остаточного потока. То есть фактический поток в ребре.

**Описание работы программы.**

Функция **main()** создает объект класса **List** и вызывает функцию   
**List :: read()**, которая считывает количество ребер и сами ребра, строит соответствующий список. Затем вызывается функция **fordFulkerson()** и выводится возвращенный ей максимальный поток. Далее вызывается функция **print()**, выводящая фактический поток в ребрах и программа завершает работу.

Исходный код программы представлен в Приложении Б.

**Исследование сложности алгоритма.**

На каждом шаге алгоритм добавляет поток увеличивающего пути к уже имеющемуся потоку. Так как поток в ребре – целое число, на каждом шаге алгоритм увеличивает поток по крайней мере на единицу, следовательно, он сойдётся не более чем за O(*f*) шагов, где *f* – максимальный поток в графе. Можно выполнить каждый шаг за время O(*E*), где *E* – число рёбер в графе, тогда общее время работы алгоритма ограничено O(*Ef*).

Если величина пропускной способности хотя бы одного из рёбер — иррациональное число, то алгоритм может работать бесконечно, даже не обязательно сходясь к правильному решению.

**Тестирование.**

Программа тестировалась в среде разработки Qt с помощью компилятора MinGW 5.3.0 в операционной системе Windows 10.

Тестовые случаи представлены в Приложении А.

**Вывод.**

В ходе выполнения задания был реализован алгоритм Форда-Фалкерсона, находящий максимальный поток в сети из истока в сток. Реализована функция поиска пути по правилу: каждый раз выполняется переход по ребру, соединяющему вершины, имена которых в алфавите ближе всего друг к другу. Оценена сложность алгоритма.

**ПРИЛОЖЕНИЕ А**

**ТЕСТОВЫЕ СЛУЧАИ**

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| 13  a h  a b 6  a c 6  c b 5  b d 4  b e 2  c e 9  e d 8  d f 4  d g 2  e g 7  f h 7  g h 4  g f 11 | 11  a b 6  a c 5  b d 4  b e 2  c b 0  c e 5  d f 4  d g 2  e d 2  e g 5  f h 7  g f 3  g h 4 |
| 5  a d  a b 10  a c 10  b c 10  b d 20  c d 30 | 20  a b 10  a c 10  b c 10  b d 0  c d 20 |
| 10  a l  n l 1  n k 1  n p 2  d n 3  b d 5  b e 2  c f 20  c g 3  a b 9  a c 5 | 1  a b 1  a c 0  b d 1  b e 0  c f 0  c g 0  d n 1  n k 0  n l 1  n p 0 |

**ПРИЛОЖЕНИЕ Б**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

#include <iostream>

#include <map>

#include <algorithm>

#include <queue>

#include <limits>

using namespace std;

struct node{

char value;

map<node\*, pair<int, int>> neighbours;

node() = default;

node(char value) : value(value){}

};

class list{

private:

node\* source;

node\* stock;

map<node\*, node\*> way;

map<char, node\*> pointers;

multimap<char, pair<node\*, node\*>> toprint;

map<char, map<char, int>> sorted;

map<char, bool> viewed;

public:

list(){

source = new node();

stock = new node();

}

~list(){

for (auto it = pointers.begin(); it != pointers.end(); it++)

delete it->second;

way.clear();

toprint.clear();

pointers.clear();

}

void read(){

int N, w;

char vi, vj, v0, vn;

cin >> N >> v0 >> vn;

pointers[v0] = source;

pointers[vn] = stock;

source->value = v0;

stock->value = vn;

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

cin >> vi >> vj >> w;

sorted[vi].insert(pair<char, int>(vj, w));

}

for (auto &it : sorted){

if(!pointers[it.first]) pointers[it.first] = new node(it.first);

for(auto &tr : it.second){

if(!pointers[tr.first]) pointers[tr.first] = new node(tr.first);

pointers[it.first]->neighbours[pointers[tr.first]] = pair<int, int>(tr.second, tr.second);

if(!pointers[tr.first]->neighbours[pointers[it.first]].first)

pointers[tr.first]->neighbours[pointers[it.first]] = pair<int, int>(0, tr.second);

toprint.insert(pair<char, pair<node\*, node\*>>(pointers[it.first]->value, pair<node\*, node\*>(pointers[it.first], pointers[tr.first])));

}

}

}

node\* ws(char from){

if(pointers[from] == stock) return stock;

viewed[from] = true;

for (auto &it : sorted[from]) {

if(pointers[from]->neighbours[pointers[it.first]].first > 0 && viewed[it.first] == false){

node\* to = ws(it.first);

if(to != nullptr){

way[pointers[to->value]] = pointers[from];

return pointers[from];

}

else continue;

}

}

return nullptr;

}

int fordFulkerson(){

node\* from;

node\* to;

int flow = 0;

while(ws(source->value) != nullptr){

int pathflow = numeric\_limits<int> :: max();

for (to = stock; to != source; to = way[to]) {

from = way[to];

pathflow = min(pathflow, from->neighbours[to].first);

}

for (to = stock; to != source; to = way[to]) {

from = way[to];

if(from->neighbours[to].first - pathflow < 0){

from->neighbours[to].first = 0;

}

else from->neighbours[to].first -= pathflow;

if(to->neighbours[from].first + pathflow > to->neighbours[from].second){

to->neighbours[from].first = to->neighbours[from].second;

}

else to->neighbours[from].first += pathflow;

}

flow += pathflow;

viewed.clear();

}

return flow;

}

void print(){

for (auto &it : toprint)

cout << it.first << " " << it.second.second->value << " " << abs(it.second.first->neighbours[it.second.second].first - it.second.first->neighbours[it.second.second].second) << endl;

}

};

int main()

{

list lst;

lst.read();

cout << lst.fordFulkerson() << endl;

lst.print();

return 0;

}