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

Студент гр. 8383 _____ Ларин А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить принцип работы алгоритма Форда-Фалкерсона для нахождения максимального потока в сети на графах. Решить с их помощью задачи

Основные теоретические положения.

Алгоритм Форда — Фалкерсона решает задачу нахождения максимального потока в транспортной сети.

Идея алгоритма заключается в следующем. Изначально величине потока присваивается значение 0: f (u , v) = 0 f(u,v)=0 для всех u , v ∈ V. Затем величина потока итеративно увеличивается посредством поиска увеличивающего пути (путь от источника s к стоку t, вдоль которого можно послать больший поток). Процесс повторяется, пока можно найти увеличивающий путь.

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 (он называется увеличивающим путём или увеличивающей цепью) максимально возможный поток:
    a. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью .
    b. Для каждого ребра на найденном пути увеличиваем поток на C_{min}, а в противоположном ему — уменьшаем на C_{min}
    c. 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.

Словарь терминов:

Сеть – ориентированный взвешенный граф, имеющий один исток и один сток.

Исток – вершина, из которой рёбра только выходят*.

Сток – вершина, в которую рёбра только входят*.

Поток – абстрактное понятие, показывающее движение по графу.

Величина потока – числовая характеристика движения по графу (сколько всего выходит из истока = сколько всего входит в сток).

Пропускная способность – свойство ребра, показывающее, какая максимальная величина потока может пройти через это ребро.

Максимальный поток (максимальная величина потока) – максимальная величина, которая может быть выпущена из истока, которая может пройти через все рёбра графа, не вызывая переполнения ни в одном ребре.

Фактическая величина потока в ребре – значение, показывающее, сколько величины потока проходит через это ребро.

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

v_n - сток

v_i v_j ω_{ij} - ребро графа

v_i v_j ω_{ij} - ребро графа

...

Выходные данные:

P_{max} - величина максимального потока

v_i v_j ω_{ij} - ребро графа с фактической величиной протекающего потока

v_i v_j ω_{ij} - 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).

Sample Input:

```
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
```

Sample Output:

```
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

Вар. 4. Поиск в глубину. Итеративная реализация.

Реализация

Алгоритм работает по следующей схеме:

Проверяется существование пути из истока в сток. Через ребро можно провести пусть по направлению если его поток меньше пропускной способности, и против его направления если его поток не равен нулю.

Если путь есть, по нему вычисляется минимальная остаточная пропускная способность. Для ребер, обходимых по ходу их ориентации остаточная пропускная способность равна разности максимальной пропускной способности и пущенного потока. Для ребер, обходимых против хода их ориентации она равна пущенному потоку. Далее на найденную величину корректируются все ребра в графе — прямые — в сторону увеличения, обратные — уменьшения. После этого вновь ищется путь.

Если пути из истока в сток нет то алгоритм прекращает работу

Сложность по памяти — линейная от количества ребер O(|E|), т. к. каждой будет соответствовать метка с информацией о величине потока.

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).

Описание функций и структур данных

Реализация всех самописных структур данных представлена в коде в приложении А

```
Типу V_TYPE соответствует char – имена вершин
Типу W_TYPE соответствует double – величина потока
```

Для представления графа используются структуры данных set и map из STL

```
std::map<V_TYPE, std::set<HaEdge> > graph;
```

Словарь map, основанная на красно-черных деревьях, ставит в соответствие каждой вершине множество set ребер HaEdge.

HaEdge представляет из себя структуру содержащую информацию о инцидентных вершинах, потоке, пропускной способности, а так же флаги, показывающие является ли данное ребро обратным(для построения путей против хода) и задано ли оно явно.
set используется так же для хранения множества посещенных вершин, a vector для хранения последовательного пути, формируемого алгоритмом.

void read() - читает информацию о графе из стандартного потока ввода в глобальную переменную graph, S, T.

void write() - пишет считанный граф в стандартный вывод

W_TYPE getMaxFlow() - возвращает поток, проходящий через граф в данный момент.

void propperWrite() - Выводит граф в виде, требуемом для проверяющей системы.

bool findPath() - изет путь в графе из истока к стоку, сохраняя его в публичную переменную path. Возвращает возможно ли построить путь

void printPath() - выводит дуги, через которые проходит найденный путь

W_TYPE findBottleneck() - везвращает величину для найденного пути , на которую можно откорректировать соответствующие дуги

void augmentPath(W_TYPE amount) - корректирует путь на величину „amount“

void ff() - реализует основную логику метода Форда-Фалкерсона.

Тесты.

1.
9
a

```
d
a b 8
b c 10
c d 10
h c 10
e f 8
g h 11
b e 8
a g 10
f d 8

a : (a -> b 0/8) (a -> g 0/10)
b : (b <- a 0/8) (b -> c 0/10) (b -> e 0/8)
c : (c <- b 0/10) (c -> d 0/10) (c <- h 0/10)
d : (d <- c 0/10) (d <- f 0/8)
e : (e <- b 0/8) (e -> f 0/8)
f : (f -> d 0/8) (f <- e 0/8)
g : (g <- a 0/10) (g -> h 0/11)
h : (h -> c 0/10) (h <- g 0/11)
/------------------------------------------------\
Looking for path
Stack state:
a
  /Consider edge (a -> b 0/8)
  |Edge added(a -> b 0/8)
Stack state:
a b
  /Consider edge (b <- a 0/8)
  \Edge not augmentable
  \Edge leads back
  /Consider edge (b -> c 0/10)
  |Edge added(b -> c 0/10)
Stack state:
a b c
  /Consider edge (c <- b 0/10)
  \Edge not augmentable
  \Edge already considered
  \Edge leads back
  /Consider edge (c -> d 0/10)
  |Edge added(c -> d 0/10)
Path found successfully:
a b c d
\------------------------------------------------/
Found path:
a-->0/8--b
b-->0/10--c
```

```
c-->0/10--d
Augment path by 8:
a-->8/8--b
b-->8/10--c
c-->8/10--d
Res net:
a : (a -> b 8/8) (a -> g 0/10)
b : (b <- a 8/8) (b -> c 8/10) (b -> e 0/8)
c : (c <- b 8/10) (c -> d 8/10) (c <- h 0/10)
d : (d <- c 8/10) (d <- f 0/8)
e : (e <- b 0/8) (e -> f 0/8)
f : (f -> d 0/8) (f <- e 0/8)
g : (g <- a 0/10) (g -> h 0/11)
h : (h -> c 0/10) (h <- g 0/11)
==========================================
=======
/----------------------------------------------\
Looking for path
Stack state:
a
  /Consider edge (a -> b 8/8)
  \Edge not augmentable
  /Consider edge (a -> g 0/10)
  |Edge added(a -> g 0/10)
Stack state:
a g
  /Consider edge (g <- a 0/10)
  \Edge not augmentable
  \Edge leads back
  /Consider edge (g -> h 0/11)
  |Edge added(g -> h 0/11)
Stack state:
a g h
  /Consider edge (h -> c 0/10)
  |Edge added(h -> c 0/10)
Stack state:
a g h c
  /Consider edge (c <- b 8/10)
  |Edge added(c <- b 8/10)
Stack state:
a g h c b
  /Consider edge (b <- a 8/8)
  |Edge added(b <- a 8/8)
Stack state:
a g h c b a
  /Consider edge (a -> b 8/8)
```

```
  \Edge not augmentable
  \Edge already considered
  \Edge leads back
  /Consider edge (a -> g 0/10)
  \Edge already considered
!No new edges added from current. Popping edge
Stack state:
a g h c b
  /Consider edge (b -> c 8/10)
  \Edge already considered
  \Edge leads back
  /Consider edge (b -> e 0/8)
  |Edge added(b -> e 0/8)
Stack state:
a g h c b e
  /Consider edge (e <- b 0/8)
  \Edge not augmentable
  \Edge already considered
  \Edge leads back
  /Consider edge (e -> f 0/8)
  |Edge added(e -> f 0/8)
Stack state:
a g h c b e f
  /Consider edge (f -> d 0/8)
  |Edge added(f -> d 0/8)
Path found successfully:
a g h c b e f d
\------------------------------------------------/
Found path:
a-->0/10--g
g-->0/11--h
h-->0/10--c
c<--8/10--b
b-->0/8--e
e-->0/8--f
f-->0/8--d
Augment path by 8:
a-->8/10--g
g-->8/11--h
h-->8/10--c
c<--0/10--b
b-->8/8--e
e-->8/8--f
f-->8/8--d
Res net:
a : (a -> b 8/8) (a -> g 8/10)
```

```
b : (b <- a 8/8) (b -> c 0/10) (b -> e 8/8)
c : (c <- b 0/10) (c -> d 8/10) (c <- h 8/10)
d : (d <- c 8/10) (d <- f 8/8)
e : (e <- b 8/8) (e -> f 8/8)
f : (f -> d 8/8) (f <- e 8/8)
g : (g <- a 8/10) (g -> h 8/11)
h : (h -> c 8/10) (h <- g 8/11)
==========================================
=======
/----------------------------------------------\
Looking for path
Stack state:
a
  /Consider edge (a -> b 8/8)
  \Edge not augmentable
  /Consider edge (a -> g 8/10)
  |Edge added(a -> g 8/10)
Stack state:
a g
  /Consider edge (g <- a 8/10)
  \Edge leads back
  /Consider edge (g -> h 8/11)
  |Edge added(g -> h 8/11)
Stack state:
a g h
  /Consider edge (h -> c 8/10)
  |Edge added(h -> c 8/10)
Stack state:
a g h c
  /Consider edge (c <- b 0/10)
  \Edge not augmentable
  /Consider edge (c -> d 8/10)
  |Edge added(c -> d 8/10)
Path found successfully:
a g h c d
\----------------------------------------------/
Found path:
a-->8/10--g
g-->8/11--h
h-->8/10--c
c-->8/10--d
Augment path by 2:
a-->10/10--g
g-->10/11--h
h-->10/10--c
c-->10/10--d
```

Res net:
a : (a -> b 8/8) (a -> g 10/10)
b : (b <- a 8/8) (b -> c 0/10) (b -> e 8/8)
c : (c <- b 0/10) (c -> d 10/10) (c <- h 10/10)
d : (d <- c 10/10) (d <- f 8/8)
e : (e <- b 8/8) (e -> f 8/8)
f : (f -> d 8/8) (f <- e 8/8)
g : (g <- a 10/10) (g -> h 10/11)
h : (h -> c 10/10) (h <- g 10/11)
==========
/----------\
Looking for path
Stack state:
a
/Consider edge (a -> b 8/8)
\Edge not augmentable
/Consider edge (a -> g 10/10)
\Edge not augmentable
!No new edges added from current. Popping edge
Path does not exist!
\----------/
18
a b 8
a g 10
b c 0
b e 8
c d 10
e f 8
f d 8
g h 10
h c 10

2.
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

3.
3
a
c
a b 7
a c 6
b c 4

10
a b 4
a c 6
b c 4

4.
5
a
d
a b 20
b c 20
c d 20
a c 1
b d 1

21
a b 20
a c 1
b c 19
b d 1
c d 20

Выводы.

В результате работы была написана полностью рабочая программа решающая поставленную задачу при использовании изученных теоретических материалов. Программа было протестирована, результаты тестов удовлетворительны.

ПРИЛОЖЕНИЕ А(ЛИСТИНГ ПРОГРАММЫ)

```
#include <iostream>
#include <fstream>
#include <sstream>
//#include <tuple>
#include <set>
#include <vector>
#include <map>
#include <algorithm>
#include <string>
#include <assert.h>

#define FILE_INP
//#define DEBUG



#define V_TYPE char
#define W_TYPE double

#define CLAMP(VAL) ((VAL)<0?0:(VAL))
/*
struct Edge{
    int a;
    int b;
    double cap;
    double flow;
    bool backward;

    bool operator<(const Edge &other) const {
        return (this->a != other.a)?(this->a < other.a):(this->b < other.b);
    }

    bool cmp(const Edge &other) const {
        return (this->a != other.a)?(this->a < other.a):(this->b < other.b);
    }
};
*/

struct HaEdge {
    V_TYPE a;
    V_TYPE b;
```

```
W_TYPE cap;
mutable  W_TYPE flow;
mutable bool backward;
mutable bool fpr;

W_TYPE resCap()const{
    return backward?flow:cap-flow;
}

bool augmentable() const{
    return backward?(flow!=0):(cap!=flow);
}

void augment(std::map<V_TYPE, std::set<HaEdge> > &gr, W_TYPE amount)const {
    if(backward&&fpr){
        modFlow(-amount);
        gr[b].find({0,a,0,0,false})->modFlow(amount);
    }else{
        modFlow(amount);
        gr[b].find({0,a,0,0,false})->modFlow(-amount);
    }
}

void modFlow(W_TYPE amount)const {
    if(backward){
        flow-=amount;
    }else{
        flow+=amount;
    }
    //assert(flow>=0 && flow<=cap);
}

bool operator<(const HaEdge &other) const {//Compare by destination vertex
    return (this->b < other.b);
}

bool operator==(const HaEdge &other) const {
    return (this->b == other.b);
}

bool cmp(const HaEdge &other) const {
```

```
        if(this->b == other.b){
            if(this->backward==other.backward) {
                if (this->cap == other.cap) {
                    return this->flow < other.flow;

                } else {
                    return this->cap < other.cap;
                }
            }else{
                return this->backward<other.backward;
            }
        }else{
            return this->b < other.b;
        }

    }

    static bool cmpRes(const HaEdge &lval, const HaEdge &rval) {
        return (lval.resCap()<rval.resCap());
    }



    std::string toString()const{
        std::ostringstream strs;
        strs<<"("<<a<<" "<<(backward?"<-":"->")<<" "<<b<<"
"<<CLAMP(flow)<<"/"<<cap<<")";
        return strs.str();
    }

    std::string shortStr()const {
        std::ostringstream strs;
        strs<<a<<(backward?"<--":"-->")<<CLAMP(flow)<<"/"<<cap<<"--"<<b;
        return strs.str();
    }
};

std::map<V_TYPE, std::set<HaEdge> > graph;

V_TYPE S;//Starting vertex
V_TYPE T;//Sinc vertex

std::vector<V_TYPE> path;
```

```
void quickGreedy() {
    std::set<V_TYPE> visited;
    V_TYPE v = S;

    visited.insert(S);

    std::cout<<v<<std::endl;

    while(v!=T){
        W_TYPE minW = INT32_MAX;
        V_TYPE minV = S;
        for(auto it:graph[v]){
            if(!visited.count(it.b)) {
                if(it.cap < minW){
                    minW=it.cap;
                    minV=it.b;
                }
            }
        }
        std::cout<<minV<<" "<<minW<<std::endl;
        visited.insert(minV);
        v = minV;
        minW=INT32_MAX;
    }
}

W_TYPE getMaxFlow(){//Summ by all edges from start
    W_TYPE sum = 0;
    for(auto it:graph[S]){
        sum+=it.flow;
    }
    return sum;
}

void read() {
    int n = 0;
    V_TYPE a;
    V_TYPE b;
    W_TYPE w;
    HaEdge edge;
```

```
    std::cin>>n>>S>>T;// Here's sufficiently trivial



    while (n--) {
        std::cin >> a >> b >> w;

        edge = {a, b, w, 0, false};
        if(graph[a].count(edge)){
            graph[a].find(edge)->fpr=true;
            graph[a].find(edge)->backward=false;
        }
        graph[a].insert(edge);

        edge = {b, a, w, 0, true};
        if(graph[b].count(edge)){
            graph[b].find(edge)->fpr=true;
            graph[b].find(edge)->backward=false;
        }
        graph[b].insert(edge);
    }
}
void write(){
    for(auto it:graph){
        std::cout<<it.first<<" : ";
        for(auto it1:it.second){
            std::cout<<it1.toString()<<" ";
        }
        std::cout<<std::endl;
    }
}

//#define FLOW(IT) (((IT).flow>=0&&(IT).flow<=(IT).cap)?((IT).flow):0)
#define FLOW(IT) ((IT).flow)

void propperWrite(){
    std::cout<<getMaxFlow()<<std::endl;//Max flow forst
    for(auto it:graph){

        for(auto it1:it.second){
            if(!it1.backward ||it1.fpr )
```

```
                std::cout<<it1.a<<" "<<it1.b<<"
"<<CLAMP(FLOW(it1))<<std::endl;//Then all enges in required format
        }
    }
}

bool _findPath(){
    std::set<HaEdge> q;
    std::vector<V_TYPE> _path;
    std::set<V_TYPE> visited;
    visited.insert(S);
    bool flag = false;
    bool iterRep = true;
    q.insert({0,S,0,0,false});
    //for(auto it:graph[S]){q.insert(it);visited.insert(it.b);};
    while(iterRep){
        iterRep=false;
        for(auto it:q) {
            for (auto it1:graph[it.b]) {
                if (!q.count(it1) && it1.augmentable() && !visited.count(it1.b))
{
                    q.insert(it1);
                    visited.insert(it1.b);
                    iterRep = true;
                    if (it1.b == T) {
                        flag = true;
                        break;
                    }
                }
            }
            if (flag)break;
        }
    }
    if(flag){

        HaEdge v = {T,0,0,0,false};//*(q.find({0,T,0,0,false}));
        while(v.a!=S){
            _path.push_back(v.a);
            v = *(q.find({0,v.a,0,0,false}));
        }
        _path.push_back(v.a);
        path = std::vector<V_TYPE>(_path.rbegin(),_path.rend());
        return true;
```

```
    }else{
        return false;
    }

}

bool findPath() {

    std::set<V_TYPE> visited;
    std::vector<V_TYPE> _path;
    bool flag = true;

    _path.push_back(S);
#ifdef DEBUG

std::cout<<"/-----------------------------------------------\\"<<std::endl;
    std::cout<<"Looking for path"<<std::endl;
#endif
    auto itInit = graph[_path.back()].begin();

    while(_path.back()!=T){// Main loop. On the turns one vertex get processed
#ifdef DEBUG
        std::cout<<"Stack state:"<<std::endl;
        for(auto it:_path)std::cout<<it<<" ";
        std::cout<<std::endl;
#endif
        flag=false;
        for(auto it=itInit;it!=graph[_path.back()].end();it++){//Loop for
vertexed from current
#ifdef DEBUG
            std::cout<<"\t/Consider edge "<<it->toString()<<std::endl;
#endif
            if(it->augmentable() && !visited.count(it->b)&&(it->b!
=_path[_path.size()!=1?_path.size()-2:0])){//If current edge can be traced
#ifdef DEBUG
                std::cout<<"\t|Edge added"<<it->toString()<<std::endl;
#endif
                _path.push_back(it->b);
                visited.insert(it->b);
                flag = true;
                break;
            }else{
#ifdef DEBUG
```

```
                if(!it->augmentable()){
                    std::cout<<"\t\\Edge not augmentable "<<std::endl;
                }
                if(visited.count(it->b)){
                    std::cout<<"\t\\Edge already considered"<<std::endl;
                }
                if(!((it->b!=_path[_path.size()!=1?_path.size()-2:0]))){
                    std::cout<<"\t\\Edge leads back"<<std::endl;
                }
#endif
            }
        }
        if(!flag){//If no edges were added in this iteration
#ifdef DEBUG
            std::cout<<"!No new edges added from current. Popping
edge"<<std::endl;
#endif
            bool backtrack = true;
            //while(backtrack) {
            V_TYPE v = _path.back();
            _path.pop_back();

            if (_path.empty()){
#ifdef DEBUG
                std::cout<<"Path does not exist!"<<std::endl;

std::cout<<"\\-------------------------------------------------/"<<std::endl;
#endif
                return false;
            }
            auto it = graph[_path.back()].find({0, v, 0, 0, false});
            it++;
            /*
            if (it != graph[_path.back()].end()){
                backtrack=false;
                _path.push_back(it->b);
                visited.insert(it->b);
            }
             */
            //}
            itInit = it;
        }else{
            itInit = graph[_path.back()].begin();
```

```
        }
    }
#ifdef DEBUG
    std::cout<<"Path found successfully:"<<std::endl;
    for(auto it:_path)std::cout<<it<<" ";
    std::cout<<"\
n\\-----------------------------------------------/"<<std::endl;
#endif
    path=_path;
    return true;
}


W_TYPE findBottleneck() {//Trace found path, finding minimal rese. capacity
    W_TYPE minW = INT32_MAX;
    for(int i = 1;i<path.size();i++){
        HaEdge edge = *(graph[path[i-1]].find({0, path[i], 0, 0, false}));
        auto rCap  = edge.resCap();
        minW=rCap<minW?rCap:minW;
    }
    return minW;
}


void augmentPath(W_TYPE amount) {//Trace found path, augmenting edges with
regard to their direction
    for(int i = 1;i<path.size();i++){
        (graph[path[i-1]].find({0, path[i], 0, 0, false}))-
>augment(graph,amount);
    }
}


void printPath(){//Print found path in human readable way
    //std::cout<<S<<std::endl;
    for(int i = 1;i<path.size();i++){
        std::cout<< (graph[path[i-1]].find({0, path[i], 0, 0, false}))-
>shortStr()<<std::endl;
    }
}




void ff() {//Main Ford-Fulkerson method logic function
    bool flag = findPath();
    W_TYPE bottleNeck = 0;
```

```
    while (flag) {//While there is a path from start to sink
#ifdef DEBUG
        std::cout<<"Found path:"<<std::endl;
        printPath();
#endif
        bottleNeck = findBottleneck();//Find aug value
        augmentPath(bottleNeck);//Modify path with that value
#ifdef DEBUG
        std::cout<<"Augment path by "<<bottleNeck<<":"<<std::endl;
        printPath();
        std::cout<<"Res net:"<<std::endl;
        write();

std::cout<<"=================================================="<<std::endl;
#endif
        flag=findPath();//Find new path

    }
}



int main() {
#ifdef FILE_INP
    std::ifstream
in("/media/anton/E6D8B24FD8B21E2D/Git/txcloud/Labs/s2/Alg/3_NetFlow/in");
    std::cin.rdbuf(in.rdbuf());
#endif
    read();
#ifdef DEBUG
    write();
    //_findPath();
    //printPath();
    //quickGreedy();
#endif
    ff();
    propperWrite();

    //std::cout << "Hello, World!" << std::endl;
    return 0;
}
```