Teoriblad 1

Av: Theodor Beskow

Problem 1

För att lösa deluppgift 1 så kan man göra följande. Du sparar en stor grid av storlek N*N där rutan bestämmer ifall den x och y koordinaten är blockerad eller inte. Sedan kan man göra denna grid till en 2d prefix summa genom att först lägga till grid[i][j]+=grid[i-1][j] och efter man har gjort det med hela griden så gör man grid[i][j]+=grid[i][j-1]. När man gör detta så får man vara uppmärksam så att man inte försöker ta element som är utanför griden. Exempelvis om i = 0 så ska man inte göra grid[i][j]+=grid[i-1][j]. Efter man har gjort detta så kan man gå igenom alla N^2 rutor och räkna antalet som är lika med 0 vilket blir svaret. Denna lösning är $O(N^2+K)$ vilket är tillräckligt snabbt för deluppgift 1.

Kod:

```cpp
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define fo(i, n) for(ll i=0;i<((ll)n);i++)

int main() {
    cin.tie(0)->sync_with_stdio(0);
    ll N, K;
    cin >> N >> K;
    vector<vector<ll>> grid(N, vector<ll>(N, 0));

    fo(i, K) {
        ll x, y;
        cin >> x >> y;
        grid[x][y] = 1;
    }
    fo(i, N) {
        fo(j, N) {
            if(i > 0) grid[i][j] += grid[i-1][j];
        }
    }
    fo(i, N) {
        fo(j, N) {
            if(j > 0) grid[i][j] += grid[i][j-1];
        }
    }
    ll answer = 0;
    fo(i, N) {
        fo(j, N) {
            if(grid[i][j] == 0)answer++;
        }
    }
    cout << answer << "\n";

    return 0;
}
```

För deluppgift 2 så behöver man en lösning lite mer lik full lösningen. Till att börja så skapar du en lista med N element som är lika med N, låt oss kalla den v. Sedan går du igenom alla koordinater (x, y) och sätter v[x] = min(v[x], y). När du sedan ska räkna ut svaret så kan du spara en variabel ans = 0 och en variabel minFound = N. Sedan går du igenom alla N värden i v från 0 till N-1 och sätter först minFound = min(minFound, v[i]) och efter detta så lägger man till minFound till ans: ans+=minFound. Tidskomplexiteten av denna lösning blir O(N+K)

Kod:

```cpp
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define fo(i, n) for(ll i=0;i<((ll)n);i++)

int main() {
    cin.tie(0)->sync_with_stdio(0);

    ll N, K;
    cin >> N >> K;

    vector<ll> v(N, N);

    fo(i, K) {
        ll x, y;
        cin >> x >> y;
        v[x] = min(v[x], y);
    }

    ll ans = 0, minFound = N;
    fo(i, N) {
        minFound = min(minFound, v[i]);
        ans += minFound;
    }

    cout << ans << "\n";

    return 0;
}
```

Jämförd med lösning från deluppgift 1 med 200 testfall från detta python script:

```python
import random
rand = random.randint
n,k=rand(100,200),rand(20, 100)
print(n, k)

for i in range(k):
    print(rand(0, n-1), rand(0, n-1))
```

För deluppgift 3/full lösningen så kan man använda ungefär samma idé som den tidigare lösningen. Nu har vi dock inte råd att skapa listan av N element så vi börjar istället med att spara alla blockerade rutor i en lista av pairs {x, y} som vi kan kalla blockerade. Sedan ska vi sortera denna lista vilket går på O(KlogK). När vi sorterar denna lista så vill vi först sortera med avsikt på x(det första elementet). Det är även väldigt viktigt att vi lägger till en extra “fake” blockerad ruta på x = n och y = 0 så att det blir lättare med uträkningarna senare. Nu är det enda vi har kvar att räkna antalet tomma rutor. Detta kan vi göra genom att spara ans = 0, minFound = N och last = 0. Nu kan vi gå igenom alla dem K+1 elementen och göra följande, först så lägger vi till (blocked[i].first-last)*minFound på svaret så ans+=(blocked[i].first-last)*minFound. Sedan så sätter vi last=blocked[i].first och minFound = min(minFound, blocked.second). Med detta så får vi svaret i ans. Svaret i denna deluppgift kan dock overflowa en 64 bit integer så istället så får man använda en 128 bit integer eller bara använda python lol. Denna lösning går i O(KlogK).

Kod:

```
N, K = map(int, input().split())
blocked = []
for _ in range(K):
    x, y = map(int, input().split())
    blocked.append([x, y])

blocked.append([N, 0])
blocked.sort()

ans = 0
minFound = N
last = 0
for b in blocked:
    ans += (b[0] - last) * minFound
    last = b[0]
    minFound = min(minFound, b[1])

print(ans)
```

Jämförd med lösning från deluppgift 2 med 750 testfall från detta python script:

```
import random
rand = random.randint
n,k=rand(10000,100000),rand(10000, 100000)
print(n, k)

for i in range(k):
    print(rand(0, n-1), rand(0, n-1))
```

Den är även testad med några testfall med större N som overflowar long longs såsom:

tc1:
1000000000000000000 0

tc2:
1000000000000 1
1 100000000

Problem 2

Först så gör jag en lösning som är för långsam för alla deluppgifter men som kan användas för att verifiera att vissa andra lösningar fungerar. I denna lösning så försöker jag hitta en hamiltonisk väg vilket man kan göra på $O(N^2*2^N)$ tid med dynamisk programmering. Detta kan klara ungefär N <= 20 men eftersom grafen inte är komplett och statesen är lite sparse så kan den klara N <= 30 i praktiken.

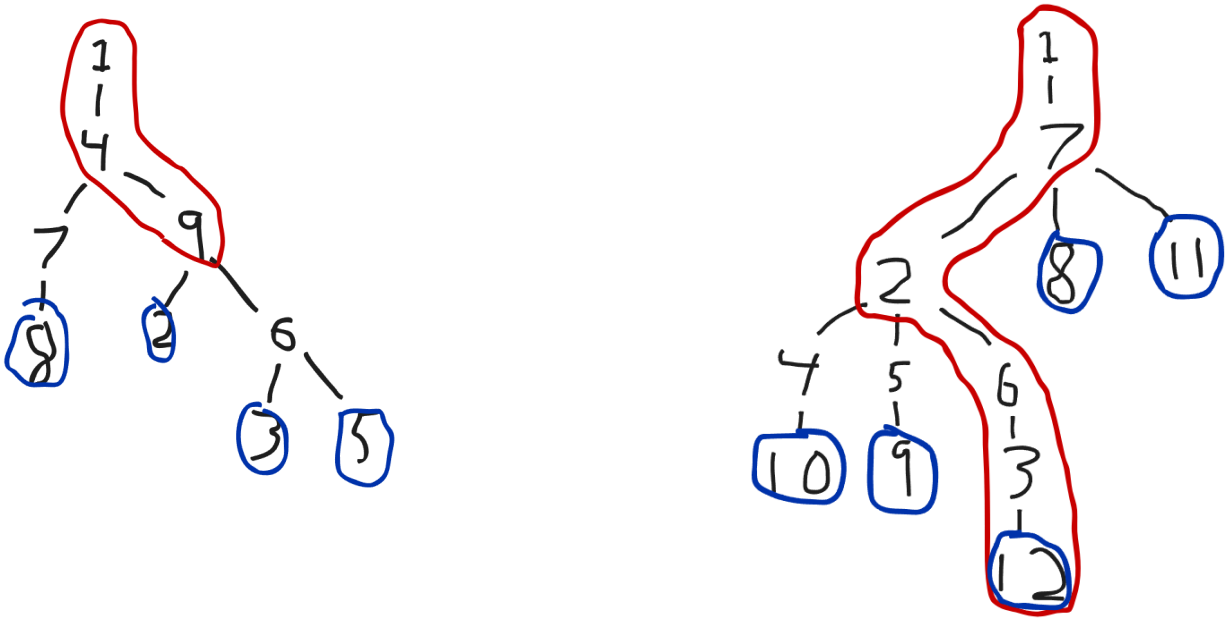
Kod:

```cpp
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define fo(i, n) for(ll i=0;i<((ll)n);i++)

vector<vector<ll>> adj;
ll N;
vector<unordered_set<ll>> seen;

bool dfs(ll pos, ll mask = 1, ll am = 1){
    if(pos == N-1) return am == N;
    if(seen[pos].count(mask)) return 0;
    seen[pos].insert(mask);
    for(auto &v : adj[pos]){
        if(!(mask&(1ll<<v))){
            if(dfs(v, mask|(1ll<<v), am+1)) return 1;
        }
        for(auto &v2 : adj[v]){
            if(!(mask&(1ll<<v2))){
                if(dfs(v2, mask|(1ll<<v2), am+1)) return 1;
            }
        }
    }
    return 0;
}

int main() {
    cin.tie(0)->sync_with_stdio(0);

    ll from, to;
    cin >> N;
    adj.resize(N);
    fo(i, N-1){
        cin >> from >> to;
        adj[--from].push_back(--to);
        adj[to].push_back(from);
    }

    seen.assign(N, {});
    if(dfs(0))cout << "possible";
    else cout << "IMPOSSIBLE";
    return 0;
}
```

Den här lösningen skriver just nu inte ut svaret ifall den hittar det men detta behövs inte som vi kommer se, då denna lösning endast används för att verifiera att andra lösningar fungerar.

Att graden på varje nod är mindre eller lika med 3 hjälper inte så mycket. Detta träd blir nästan ett binärt träd(ifall vi rotar den vid nod 1(0 i min kod)), då är roten den enda noden som kan ha tre barn. Jämfört med full lösningen så är det vissa saker som man inte behöver tänka på. Deluppgift 2 där N<=1000 ser jag ärligt talat ingen rimlig lösning som inte kan lösa problemet fullt ut.

För att lösa problemet helt så kan man använda sig av en girig strategi. Men till att börja så kan vi markera alla noder mellan och inklusive noderna mellan 1 och N. Detta kan man lätt göra med en lite modifierad dfs. Sedan vill vi också hålla koll på vilka noder som är lövnoder. Detta skulle man kunna göra med att räkna graden på noden men eftersom vi inte vill få med roten eller eventuellt några andra buggar så kan vi lika gärna göra en dfs för det också. I följande graf där N = 9 och som är rotad i nod 1 så är de blåmarkerade noderna löv/små subträd och de rödmarkerade del av vägen till noden N. Det andra trädet är exempel trädet från uppgiften:



Med detta så kan vi starta våran huvud dfs loop. Dfs'en kommer att innehålla/anropas med position(pos), senaste position(last) och ifall jag måste vara på denna nod nu eller om jag kan hoppa över den genom att hoppa två steg(isActive). Jag kommer att kalla den sist nämnda isActive och ifall den är 1 så betyder det att jag måste gå på denna nod(jag har hoppat två steg för att komma hit eller är roten). Något man kan upptäcka är att det alltid är gynnsamt att ha isActive = 0 då man kan göra samma saker som isActive = 1 plus lite mer. Den andra insikten man kan göra är att det alltid är gynnsamt att gå till nästa nod på den röda vägen sist av alla sina grannar. Detta kan man typ bevisa med den tidigare nämnda insikten.

Så först, fallet då jag måste använda denna nod nu(isActive = 1). Först lägger man uppenbarligen till den nuvarande noden. Jag kallar ett subträd stort ifall det är mer än 1 nod i subträdet och den inte är del av den röda vägen. Ifall det finns fler än 1 stort subträd så är det inte möjligt. Detta är då den första noden i det andra subträdet kommer användas på vägen ner vilket sedan gör det omöjligt att komma tillbaka i trädet med endast 2 stegs hopp(och den kan inte sluta i subträdet då den inte har en röd nod i sig). Det här stora subträdet ska anropas med isActive = 0. Däremot är det möjligt att gå till flera små subträd då de använder samma nod upp som ner. Efter allt detta så går man till nästa nod på den röda vägen. Till denna nästkommande röda nod så använder man isActive = 1 ifall man har behövt gå till någon annan nod mellan den nuvarande och den nästkommande noden, om inte så är isActive = 0.

Nu med andra fallet att isActive = 0. Först och främst så kan man gå till alla små subträd(som inte är del av den röda vägen) då man får samma förhållande och kan ha isActive = 0 igen på samma nod när man kommer tillbaka. Med de större subträden är det lite mer komplicerat nu jämfört med isActive = 1. Ifall det finns fler än 2 stora subträd så är det omöjligt. Om det finns exakt 2 styckna så är det fortfarande omöjligt ifall den nuvarande noden inte är en röd nod. Detta är då efter att man har besökt det första stora subträdet så kommer man behöva använda den nuvarande noden. Alltså så kommer det andra stora subträdet behöva hoppa tillbaka från den första noden i subträdet. Detta gör det möjligt att hoppa till nästa röda nod ifall det finns en sådan. Men det är inte möjligt att hoppa tillbaka från ett subträd som inte har en röd nod då föräldern till noden som hade isActive = 0 måste vara lika med 1 och den noden just nu också är besökt efter det första stora subträdet. Detta betyder även att ifall det finns 2 stora subträd så måste nästa röda nod ha isActive = 1.

Nu är frågan vilken av dessa två stora subträd som man ska ge isActive = 0 och isActive = 1(därmed även vilken man ska börja med). Faktum är att det inte spelar någon roll. Ifall någon av de stora subträden behöver ha isActive = 0 så har vi nyss bevisat att det inte är möjligt att ta sig tillbaka i detta tillfälle(så man kan välja en godtycklig ordning). Innan vi skriver ut svaret vi har kommit fram till så är det viktigt att kolla att svaret är av rätt längd och att svaret slutar med rätt nod. Detta löser några specialfall. Ifall något var otydligt så borde koden nedan förklara oklarheter. Tidskomplexiteten på denna lösning är O(N) vilket är mer än tillräckligt. Faktumet att lösningen verkar vara “för snabb” oroar mig lite men jag har testat lösningen med 10000 olika testfall för att verifiera att den fungerar.

Kod:

```
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define fo(i, n) for(ll i=0;i<((ll)n);i++)
#define deb(x) cout << #x << " = " << (x) << endl;
#define deb2(x, y) cout << #x << " = " << (x) << ", " << #y << " = " << (y) << endl
#define pb push_back
typedef vector<ll> vl;
typedef vector<vl> vvl;

vvl adj;
vl ans, seen, onPath, subTreeSiz;
```

```
ll N;
bool fail = 0;

bool findEnd(ll pos, ll last = -1){
    bool toEnd = (pos == N-1);
    for(auto &v : adj[pos]){
        if(v == last) continue;
        toEnd |= findEnd(v, pos);
    }
    onPath[pos] = toEnd;
    return toEnd;
}

ll dfs2(ll pos, ll last = -1){
    ll res = 1;
    for(auto &v : adj[pos]){
        if(v == last) continue;
        res+=dfs2(v, pos);
    }
    subTreeSiz[pos] = res;
    return res;
}

void dfs(ll pos, ll last = -1, bool isActive = 1){
    if(isActive){
        ans.pb(pos);
        vector<ll> bigSubtree;
        for(auto &v : adj[pos]){
            if(v == last || onPath[v]) continue;
            if(subTreeSiz[v]>1) bigSubtree.pb(v);
        }
        if(bigSubtree.size()>1) fail = 1;
        if(bigSubtree.size()) dfs(bigSubtree[0], pos, 0);
        bool nextShort = 0;
        for(auto &v : adj[pos]){
            if(v == last || onPath[v]) continue;
            nextShort = 1;
            if(subTreeSiz[v]==1) ans.push_back(v);
        }
        for(auto &v : adj[pos]){
            if(v == last || !onPath[v]) continue;
            dfs(v, pos, nextShort);
        }
    }else{
        vector<ll> bigSubtree;
        for(auto &v : adj[pos]){
            if(v == last || onPath[v]) continue;
            if(subTreeSiz[v]>1) bigSubtree.pb(v);
            else ans.push_back(v);
        }
        bool hasDoneThis = 0;
        if(bigSubtree.size()>2) fail = 1;
        fo(i, bigSubtree.size()){
            dfs(bigSubtree[i], pos, !i);
            if(!hasDoneThis){
                hasDoneThis = 1;
                ans.pb(pos);
```

```
            }
        }
        bool nextShort = 0;
        if(bigSubtree.size()==2){
            if(!onPath[pos]) fail = 1;
            nextShort = 1;
        }
        if(!hasDoneThis)ans.pb(pos);
        for(auto &v : adj[pos]){
            if(v == last || !onPath[v]) continue;
            dfs(v, pos, nextShort);
        }
    }
}

bool check(){
    if(ans[0]!=0||ans[N-1]!=N-1) return 0;
    fo(i, N-1){
        bool isPossible = 0;
        for(auto &v : adj[ans[i]]){
            if(v == ans[i+1]){
                isPossible = 1;
                break;
            }
            for(auto &v2 : adj[v]){
                if(v2 == ans[i+1]){
                    isPossible = 1;
                    break;
                }
            }
        }
        if(!isPossible) return 0;
    }
    return 1;
}


int main() {
    cin.tie(0)->sync_with_stdio(0);

    ll from, to;
    cin >> N;
    adj.assign(N, {});
    fo(i, N-1){
        cin >> from >> to;
        adj[--from].pb(--to);
        adj[to].pb(from);
    }
    onPath.assign(N, 0);
    subTreeSiz.assign(N, 0);
    findEnd(0);
    dfs2(0);
    dfs(0);
    if(fail || ans.size() != N || ans[N-1] != N-1) cout << "IMPOSSIBLE";
    else{
        // if(!check()) deb("FEEEEEEEEEEEEEEEEEEEEL!");
        // cout << "possible";
```

```
        fo(i, N) cout << ans[i]+1 << " ";
    }

    return 0;
}
```

Jämförd med den långsammare lösningen med 10000(inte typo) testfall från detta python script:

```
import random
rand = random.randint
n=rand(2,30)
print(n)

order = [i+1 for i in range(n)]
random.shuffle(order)
for i in range(n-1):
    print(order[i+1], order[rand(0, i)])
```

Eftersom ett testfall kan ha flera permutationer som är korrekta så kan jag inte jämföra direkt med min andra lösning. Istället så för jag använda funktionen check som tittar ifall permutationen är möjlig(Check är kvadratisk med är inte del av lösningen så lösningen är fortfarande linjär). Ifall lösningen säger att det inte är möjligt så checkas detta med den andra lösningen.

Problem 3

För att lösa deluppgift 1 så kan man använda en väldigt långsam lösning. Man kan testa alla möjliga fall av i och j för varje query och sedan testa ifall den är möjlig. Så om jag ska titta om det är möjligt att ta sig från A till B med i och j kan man bara göra en vanlig dfs som bara använder kanter som är mellan eller lika med i och j. Lösningen kanske fortfarande låter ganska långsam då det fortfarande finns $5*10^4$ noder. Men detta är inget problem då det inte kan finnas fler än m*2 kanter som är ihopkopplade till någon nod. Så knepet är att bara återställa seen[pos] på de positionerna jag har varit vid innan nästa dfs. Med detta så kan man svara för varje query vad svaret är genom att spara i en variabel ans vilket i och j som ger minst svar och det går att ta sig mellan A och B. Denna lösning går på $O(M^3Q)$ vilket är tillräckligt snabbt.

Kod:

```
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define fo(i, n) for(ll i=0;i<((ll)n);i++)

vector<vector<pair<ll, ll>>> adj;
vector<ll> seen, visited;

void dfs(int pos, int i, int j){
    if(seen[pos]) return;
```

```
    seen[pos] = 1;
    visited.push_back(pos);
    for(auto &[v, id] : adj[pos]){
        if(id<i || id>j) continue;
        dfs(v, i, j);
    }
}

int main() {
    cin.tie(0)->sync_with_stdio(0);
    int N, M, Q, from, to, id, A, B;
    cin >> N >> M >> Q;
    adj.assign(N, {});
    fo(i, M){
        cin >> from >> to >> id;
        adj[--from].push_back({--to, --id});
        adj[to].push_back({from, id});
    }
    seen.assign(N, 0);
    while(Q--){
        cin >> A >> B;
        A--;B--;
        ll ans = M;
        fo(i, M) {
            for(ll j = i; j < M; j++){
                dfs(A, i, j);
                if(seen[B]) ans = min(ans, j-i);
                for(auto &v : visited) seen[v] = 0;
            }
        }
        if(ans == M)cout << "IMPOSSIBLE\n";
        else cout << ans << "\n";
    }
    return 0;
}
```

För att lösa deluppgift 2 så kan man använda exakt samma lösning som som från deluppgift 1 fast istället för att använda en dfs för att checka ifall A och B är ihopkopplade så kan man använda sig av en Union Find. Detta gör att tidskomplexiteten blir $O(M^2Q\alpha(M)+MNQ)$ vilket blir $O(M^2Q\alpha(M))$ vilket är tillräckligt snabbt.

Kod:

```
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define fo(i, n) for(ll i=0;i<((ll)n);i++)

struct UnionFind{
    ll n;
    vector<ll> p, siz;
    UnionFind(ll n) : n(n){
        resetUnionFind();
    }
```

```
    void resetUnionFind(){
        p.assign(n, 1);
        siz.assign(n, 1);
        fo(i, n) p[i] = i;
    }
    ll same(ll a, ll b){
        return find(a) == find(b);
    }
    ll find(ll x){
        if(x == p[x]) return x;
        return p[x] = find(p[x]);
    }
    void unite(ll a, ll b){
        a = find(a);
        b = find(b);
        if(a == b) return;
        if(siz[b] > siz[a]) swap(a, b);
        siz[a] += siz[b];
        p[b] = a;
    }
};

int main() {
    cin.tie(0)->sync_with_stdio(0);
    int N, M, Q, from, to, id, A, B;
    cin >> N >> M >> Q;
    vector<pair<ll, ll>> edges(M);
    fo(i, M){
        cin >> from >> to >> id;
        edges[id-1] = {from-1, to-1};
    }

    UnionFind uf(N);
    while(Q--){
        cin >> A >> B;
        A--;B--;
        ll ans = M;
        fo(i, M) {
            uf.resetUnionFind();
            for(ll j = i; j < M; j++){
                uf.unite(edges[j].first, edges[j].second);
                if(uf.same(A, B)) ans = min(ans, j-i);
            }
        }
        if(ans == M)cout << "IMPOSSIBLE\n";
        else cout << ans << "\n";
    }
    return 0;
}
```

Jämförd med lösning från deluppgift 1 med 100 testfall från detta python script:

```
import random
rand = random.randint
n,m,q=rand(10,30),rand(30,40),rand(5,10)
if n<20: n = rand(10, 50000)
print(n, m, q)
```

```
for i in range(m):
    print(rand(1, n), rand(1, n), i+1)
for i in range(q):
    print(rand(1, n), rand(1, n))
```

För att lösa deluppgift 3, 4 och 5 så blir det ett stort steg fram jämfört med de tidigare lösningarna då exempelvis $O(M^2Q)$ är för långsam även för M<=1000 då den är lika med 1e9 inte mindre. Det första insikten man kan göra för att lösa dessa deluppgifter är att det är onödigt att testa alla i och j för varje query. Man kan observera att ifall i, j fungerar så kommer även i, j+1 att fungera. Med denna insikt så kan man binärsöka vad j kan vara given i. Men detta kräver en log faktor som faktiskt inte är nödvändig då man kan göra följande. Man kan testa i, j (börjar på i = 0 och j = 0) och ifall det inte går så ökar man j och om det fungerar så ökar man i. Alltså kommer man endast behöva querya varje query med unionfinden M gånger.

För att kunna göra en query på vilket intervall vi vill så kommer vi behöva skapa M styckna roll back union finds som går från varje kant k (0 till M-1) till index 0. Sedan gör vi alla queries vi behöver från index 0, sedan roll backar vi 1 steg på alla union finds och repeterar samma sak. Denna lösning går på $O(MN+M^2log(M)+QMlog(M))$ vilket går för deluppgift 3, 4 och 5.

Kod:

```
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define fo(i, n) for(ll i=0;i<((ll)n);i++)
#define deb(x) cout << #x << " = " << x << endl
#define deb2(x, y) cout << #x << " = " << x << ", " << #y << " = " << y << endl

struct UnionFind {
    ll n;
    vector<ll> p, siz;
    vector<pair<ll, ll>> hp, hsiz;

    UnionFind(ll n) : n(n) {
        p.assign(n, 1);
        siz.assign(n, 1);
        fo(i, n) p[i] = i;
    }

    bool same(ll a, ll b){
        return find(a) == find(b);
    }

    ll find(ll x){
        if(x == p[x]) return x;
        return find(p[x]);
    }
```

```
    void unite(ll a, ll b){
        a = find(a);
        b = find(b);

        hp.push_back({-1, -1});
        hsiz.push_back({-1, -1});
        if(a == b) return;
        if(siz[b] > siz[a]) swap(a, b);
        hp.pop_back();
        hsiz.pop_back();
        hsiz.push_back({a, siz[a]});
        hp.push_back({b, p[b]});
        siz[a]+=siz[b];
        p[b] = a;
    }

    void rollback(){
        if(hsiz.size() == 0) return;
        if(hsiz[((ll)hsiz.size())-1].first != -1){
            siz[hsiz[((ll)hsiz.size())-1].first] = hsiz[((ll)hsiz.size())-1].second;
            p[hp[((ll)hp.size())-1].first] = hp[((ll)hp.size())-1].second;
        }
        hsiz.pop_back();
        hp.pop_back();
    }
};

int main() {
    cin.tie(0)->sync_with_stdio(0);
    ll N, M, Q, from, to, A, B, id;
    cin >> N >> M >> Q;
    vector<tuple<ll, ll, ll>> edges;
    fo(i, M) {
        cin >> from >> to >> id;
        edges.push_back({id-1, from-1, to-1});
    }
    sort(edges.begin(), edges.end());
    vector<UnionFind> uf(M, UnionFind(N+1));
    fo(i, M){
        for(ll j = i; j>=0; j--){
            tie(id, from, to) = edges[j];
            uf[i].unite(from, to);
        }
    }
    vector<ll> ans(Q, M), current(Q, 0);
    vector<pair<ll, ll>> queries;
    fo(i, Q){
        cin >> A >> B;
        queries.push_back({A-1, B-1});
    }
    fo(i, M){
        fo(qIndex, Q){
            tie(A, B) = queries[qIndex];
            while(true){
                ll j = current[qIndex];
                if(j<i){
```

```
                    current[qIndex]++;
                    continue;
                }
                if(j == M) break;
                if(uf[j].same(A, B)){
                    ans[qIndex] = min(ans[qIndex], j-i);
                    break;
                }
                current[qIndex]++;
            }
        }
        fo(j, M) uf[j].rollback();
    }
    fo(i, Q){
        if(ans[i]==M) cout << "IMPOSSIBLE\n";
        else cout << ans[i] << "\n";
    }

    return 0;
}
```

Jämförd med lösning från deluppgift 2 med 1200 testfall från detta python script:

```
import random
rand = random.randint
n,m,q=rand(10,200),rand(50,300),rand(100,500)
print(n, m, q)

for i in range(m):
    print(rand(1, n), rand(1, n), i+1)
for i in range(q):
    print(rand(1, n), rand(1, n))
```

För att lösa deluppgift 6 så kan man använda ungefär samma lösning. Istället för att först lägga till alla kanter och sedan ta bort en från varje Union Find så lägger man bara till medan man kör. Alltså börjar du med tomma Union Finds och lägger först till en kant i den första union finden sedan andra gången lägger du till kanten på de två första osv. Med detta kan man få bort log faktorn och istället ha inverse ackerman funktion så $O(MN+M^2\alpha(M)+QM\alpha(M))$ vilket är tillräckligt snabbt.

För full lösningen så kommer vi faktiskt lösa den online och väldigt likt de tidigare lösningarna. Så istället för att generera alla möjliga sätt att querya så kommer vi istället att endast generera det som behövs för varje query. Detta kommer att blir Q*M styckna queries. Ifall grafen hade varit ett träd så hade man kunnat använda ett link cut tree som kan hantera link(sätta ihop två subträd med två noder man specificerar), cut(ta bort en kant mellan två noder man specificerar), same(är två noder i samma träd(detta kan man göra genom att titta vad roten är i trädet)). I så fall skulle du kunna expandera din range åt höger med link, korta ner din range från vänster med cut och kolla ifall noderna i querien är connectad med same. Detta är dock som sagt bara ifall grafen är ett träd (vilket den inte nödvändigtvis är). I en

generell graf är detta omöjligt att göra online(vilket det är i vår lct(link cut tree) queriesarna) utan någon speciell constrain(i log n tid). Men som tur är så vet vi i vilken ordning vi kommer att ta bort kanterna, fast inte ifall nästa query är cut eller link. Detta faktum kan man använda för att göra lct lösningen fungera i log n på vår graf.

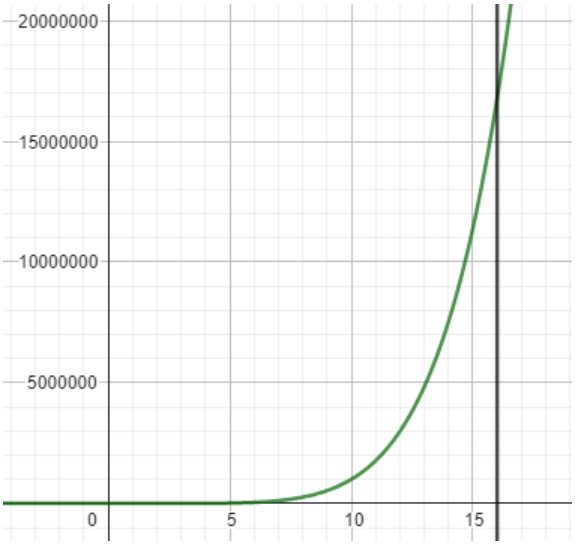
Så ifall vi ska lägga till en kant mellan v och u och de redan tillhör samma subträd så kommer detta att skapa en cykel. För att behålla lct som ett träd så kan man först ta bort den kanten i denna cykel som kommer att tas bort först. Efter att man har gjort detta så kan man bara lägga till kanten man skulle lägga till, detta kommer återigen att koppla ihop trädet. Precis som i de tidigare lösningarna så lägger man till nästa kant ifall de inte är ihopkopplade och tar bort den äldsta kanten ifall de är ihopkopplade. Sedan sparar man hur stor den minsta rangen som är möjlig är vilket blir svaret.

Nu återstår bara frågan hur man ska göra allt detta. Funktionerna cut, link, lca och find_root är väldigt standard i ett LCT och de funktionerna är baserade på en annan datastruktur som exempelvis ett splay tree. Den enda lite udda funktionen är att hitta den kant med minst index(alltså längst tid sedan vi la till kanten) i denna loop som skulle skapas. Detta går dock att göra genom att först hitta lca som är ganska standard och sedan titta vilket som är det lägsta värdet mellan lca och första noden eller lca och den andra noden(inklusive query noderna och exclusive lca). För att representera kanterna så får man säga att hörnet längst bort från roten bär på värdet av den kanten(då blir det unikt). Här är en lösning som ger summan av värden på kanterna mellan två noder och kan ändra värdet på noderna på ett träd online: https://usaco.guide/problems/ys-VertexAddPathSum/user-solutions. Som det står kommenterat i koden så är det ganska lätt att ändra denna kod från att räkna sum till att räkna min. Denna lösning går på O(QMlog(N)) vilket är tillräckligt snabbt för hela lösningen.

Problem 4

För att lösa deluppgift 1 så räcker det med en ganska långsam lösning då n<=16. Man kan skapa en adjacency matrix som är N*N stor och kan hjälpa dig att snabbt kolla ifall det finns en kant mellan två hörn. Nu gäller det bara att hitta fisken. Fisken kommer jag dela in i två delar, fenan (triangeln) och resten av kroppen (4 hörningen). Så vi kan börja med att iterera igenom alla noder och titta ifall den noden kan vara mitten noden (den noden som både triangeln och fyrhörningen använder), medan vi testar om en nod kan vara i mitten så kallar vi den för roten. För att kolla ifall en fisk kan börja där så kan vi iterera igenom alla kombinationer av fisk fenor och sedan kolla ifall denna fisk fena är möjlig. Alltså så finns det N^3 olika typer av fenor som vi kommer gå igenom.

Sedan så ska vi iterera igenom alla möjliga fyrhörningar och se om det är möjligt. För att underlätta för oss själva så kan vi bara titta på alla möjligheter av noder till resten av kroppen. För att se om en möjlig fisk graf är möjlig så får man först titta att de två fenor noderna tillsammans med roten är en komplett graf. Sedan får man kolla att det går att ta sig från roten till den första, sen till den andra, sen den tredje och tillbaka till roten igen. Man måste även ha kolla så att, förutom roten, att det inte är någon nod som används i både fenan och resten av kroppen. Denna lösning går på $O(N^6)$. Detta är tillräckligt snabbt som man kan se här:



Kod:

```cpp
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define fo(i, n) for(ll i=0;i<((ll)n);i++)

vector<vector<ll>> adj;

int main() {
    cin.tie(0)->sync_with_stdio(0);

    ll N, M, from, to;
```

```
    cin >> N >> M;
    adj.assign(N, vector<ll>(N, 0));
    fo(i, M){
        cin >> from >> to;
        from--; to--;
        adj[from][to] = 1;
        adj[to][from] = 1;
    }

    fo(root, N){
        for(ll fena1 = 0; fena1<N;fena1++){
            if(fena1 == root) continue;
            for(ll fena2 = fena1+1; fena2<N;fena2++){
                if(fena2 == root) continue;
                if(!adj[root][fena1] || !adj[root][fena2] || !adj[fena2][fena1]) continue;
                fo(head1, N){
                    if(head1 == root || head1 == fena1 || head1 == fena2) continue;
                    if(!adj[root][head1]) continue;
                    fo(head2, N){
                        if(head2 == root || head2 == fena1 || head2 == fena2 || head2 ==
head1) continue;
                        if(!adj[root][head2]) continue;
                        fo(head3, N){
                            if(head3 == root || head3 == fena1 || head3 == fena2 || head3
== head1 || head3 == head2) continue;
                            if(!adj[head2][head3] || !adj[head1][head3]) continue;
                            cout << "POSSIBLE";
                            cout << "\nKanter:\n";
                            cout << root+1 << " " << fena1+1 << "\n";
                            cout << root+1 << " " << fena2+1 << "\n";
                            cout << fena2+1 << " " << fena1+1 << "\n";
                            cout << root+1 << " " << head1+1 << "\n";
                            cout << root+1 << " " << head2+1 << "\n";
                            cout << head2+1 << " " << head3+1 << "\n";
                            cout << head1+1 << " " << head3+1 << "\n";
                            exit(0);
                        }
                    }
                }
            }
        }
    }
    cout << "IMPOSSIBLE";

    return 0;
}
```

För att lösa deluppgift 2 så kan man använda en helt annan lösning. För varje nod vill vi spara vilka cyklar som den noden är del av. För att göra detta kan vi använda en lite modifierad dfs. Så under dfs'en så har vi en seen lista som är 0 till en början 1 när jag har varit vid den noden tidigare. Dfs'en anropas med nuvarande position och tidigare position. Om den nya positionen har 1 på seen[pos] så returnerar jag pos och ett identifiernummer. Annars så fortsätter jag i dfs'en. Dfs'en består av en for loop där jag går igenom alla kanterna från noden som jag kommer gå med en dfs till om det inte var den senaste besökta

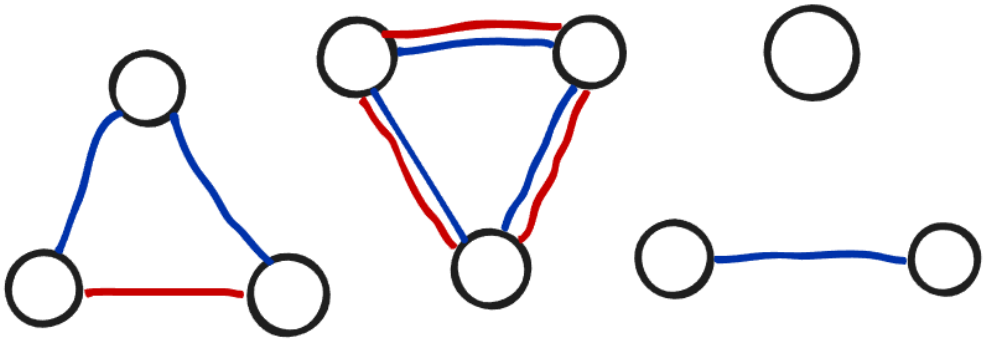
noden. Dfs anropet ska jag spara i en variabel res och om den är lika med inte är lika med -1 är den nuvarande positionen del av cykeln med identifieringsnumret som returneras. Sedan ska jag ifall res positionen inte är lika med res efter jag har sökt igenom alla kanter från det hörnet returnera res från det anropet.

Efter detta är gjort så kan man lätt gå igenom alla noder och titta vilka cykler de är med i. Ifall någon av noderna har en cykel av storlek 3 och en av storlek 4 så betyder det att vi har hittat en lösning. Denna lösning blir O(min(N, M)) vilket är tillräckligt snabbt.

För full lösningen så behövs en helt ny lösning. Vi kan börja med att hitta alla trianglar(fenor) i grafen. Detta går att göra i O(NM) genom att iterera igenom alla kanter och utifrån den kanten gå igenom alla noder och se om det går att skapa en triangel med dessa (noden får inte vara någon av noderna som kanten kopplar ihop). För att se om detta skapar en triangel utifrån en kant och ett hörn på konstant tid så kan man använda sig av ett unordered set för alla N hörn som sparar vilka andra hörn som hörnet har en kant till. Detta gör det möjligt att hitta alla trianglar O(NM) som man sedan sparar i varje nod vilka trianglar som noden är del av.

Nu måste vi göra någonting som kanske låter väldigt skumt, men det är väldigt viktigt för lösningen. Så från varje nod ska vi göra följande. Först går vi igenom alla trianglar som sitter ihop med denna nod och lägger till den i en lista kopplad till denna nod som vi kan kalla Fenor, ifall följande stämmer. Jag har inte redan lagt till 4(kan använda till exempel 5 som marginal) styckna kanter som innehåller någon av noderna som den nya kanten har. Man kan kolla antalet man har lagt in med en unordered_map. Jag behöver inte heller lägga till kanten ifall jag redan har lagt till 50 stycken på denna nod(50 är mer än tillräckligt, 16 skulle exempelvis nog räcka men 50 är en god konstant marginal). Detta gör att det finns ett konstant antal trianglar kopplade till varje nod.

Vi kan nu hitta alla dubbel kanter alltså i princip en triangel där minst 2 av kanterna finns. Sedan vill vi spara dessa med en “kant” som skulle sätta ihop dessa 3 noder till en triangel. Vi kan spara dessa i en lista men vi borde även spara dessa nya “kanter” i en unordered map(unordered_map<ll, unordered_set<ll>>) som säger vilka mitt hörn som finns från denna kant(den som inte är ihopkopplad till fake kanten som läggs till). Som nyckel kan vi stopp in kant1+kant2*N och även samma sak på kant2+kant1*N. När jag säger kant1 och kant2 menar jag indexen på kanterna från 0 till N-1(så dess riktiga index minus 1(annars kan man bara använda typ N+1 istället för N)). Så i dessa 3 exempel så är blå dem tidigare “riktiga” kanterna, och de röda kanterna de nya fake kanterna:



Precis som med trianglarna så finns det en upperbound på antalet av dessa nya kanter på N*M. Alltså går detta också på O(NM).

Nu ska vi gå igenom alla dessa nya röda kanter och göra följande. Vi kan kalla hörnen som kanten sitter ihop med för A och B, noden som är kopplad till denna kan vi kalla för roten. Vi går igenom de 4(vi kan göra 5 för marginal) första röda kanterna som har samma kant(A till B) som den vi tittar på nu(om det finns färre än 5 så går vi igenom alla som finns så man går igenom min(5, antalElement) styckna). Vi kan kalla den noden som denna nya röda kant kopplar till för C. Om det är samma kant(roten == C) så går det inte. Annars så har vi skapat fyrkanten i fisken, nu gäller det bara att se ifall det finns en fena som passar denna fyrhörning.

För att nu hitta ifall det finns en fena som passar så kan man iterera igenom de konstant antal trianglar(fenor) som är kopplade till noden. Dessa kan man hitta i Fenor[roten] och när vi går igenom dessa så kan vi kalla dem för D och E. Denna fena är inte möjlig ifall D == A eller D == B eller D == C eller E == A eller E == B eller E == C. Om ingen av dessa stämmer så är det dock möjligt. Nu definerar jag (X, Y) som en kant mellan X och Y. Då kan vi få svaret som är en kanterna (D, E), (E, roten), (D, roten), (roten, A), (roten, B), (C, A) och (C, B). Denna lösninga går på N * M * 5 * 50 men eftersom vi inte räknar konstant faktorerna så blir lösningen O(NM) vilket är tillräckligt snabbt.

Kod:

```
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define pb push_back
#define fo(i, n) for(ll i=0;i<((ll)n);i++)

ll N, M;
vector<pair<ll, ll>> edges;
vector<tuple<ll, ll, ll>> doubleEdges;
vector<vector<pair<ll, ll>>> triangles, Fenor;
vector<unordered_set<ll>> isAdjTo;
unordered_map<ll, unordered_set<ll>> fakeEdges;

void findTriangles(){
```

```
    fo(i, N){
        fo(j, M){
            ll fena1, fena2;
            tie(fena1, fena2) = edges[j];
            if(i == fena1 || i == fena2) continue;
            if(isAdjTo[i].count(fena1) && isAdjTo[i].count(fena2))
triangles[i].pb(edges[j]);
        }
    }
}

void processTriangles(){
    Fenor.assign(N, {});
    fo(i, N){
        unordered_map<ll, ll> count;
        for(auto &[v1, v2] : triangles[i]){
            if(count[v1] < 5 && count[v2] < 5 && Fenor[i].size() < 50){
                Fenor[i].pb({v1, v2});
                count[v1]++;
                count[v2]++;
            }
        }
    }
}


void findFakeEdges() {
    doubleEdges.assign(N, {});
    fo(i, N){
        fo(j, M){
            ll fena1, fena2;
            tie(fena1, fena2) = edges[j];
            if(i == fena1 || i == fena2) continue;
            if(isAdjTo[i].count(fena1)){
                ll keyVal1 = i+fena2*N;
                ll keyVal2 = fena2+i*N;
                doubleEdges.pb({i, fena1, fena2});
                fakeEdges[keyVal1].insert(fena1);
                fakeEdges[keyVal2].insert(fena1);
            }
            if(isAdjTo[i].count(fena2)){
                ll keyVal1 = i+fena1*N;
                ll keyVal2 = fena1+i*N;
                doubleEdges.pb({i, fena2, fena1});
                fakeEdges[keyVal1].insert(fena2);
                fakeEdges[keyVal2].insert(fena2);
            }
        }
    }
}

void getAns(){
    for(auto &[A, roten, B] : doubleEdges){
        ll MaxCAmount = 5;
        for(auto &C : fakeEdges[A+B*N]){
            if(C == roten) continue;
            if(MaxCAmount--<=0) break;
```

```
            for(auto &[D, E] : Fenor[roten]){
                if(D == A || D == B || D == C || E == A || E == B || E == C) continue;
                cout << "POSSIBLE";
                cout << "\nKanter:\n";
                cout << A+1 << " " << roten+1 << "\n";
                cout << B+1 << " " << roten+1 << "\n";
                cout << B+1 << " " << C+1 << "\n";
                cout << A+1 << " " << C+1 << "\n";
                cout << D+1 << " " << roten+1 << "\n";
                cout << E+1 << " " << roten+1 << "\n";
                return;
            }
        }
    }
    cout << "IMPOSSIBLE";
}


int main() {
    cin.tie(0)->sync_with_stdio(0);

    ll from, to;
    cin >> N >> M;
    isAdjTo.assign(N, {});
    triangles.assign(N, {});
    fo(i, M){
        cin >> from >> to;
        from--; to--;
        isAdjTo[from].insert(to);
        isAdjTo[to].insert(from);
        edges.pb({from, to});
    }
    findTriangles();
    processTriangles();
    findFakeEdges();
    getAns();

    return 0;
}
```

Jämförd med lösningen från deluppgift 1 med 10000 testfall från detta python script:

```
import random
rand = random.randint
n,m=rand(10,25), rand(10, 40)
print(n, m)

for i in range(m):
    a,b=0,0
    while a==b: a,b=rand(1, n), rand(1, n)
    print(a,b)
```