A-III-4 Stolný tenis

Úlohu budeme modelovať pomocou teórie grafov.
Skutočne odohrané zápasy si môžeme predstaviť ako orientovaný graf Z, ktorého hrany hovoria, kto nad kým vyhral. Chválenie sa potom zodpovedá cestám v tomto grafe: x sa môže chváliť, že je lepší ako y, ak existuje cesta po hranách (idúc len v správnom smere) z vrcholu x do vrcholu y.
Vyššie popísaný graf nepoznáme. Na vstupe máme zadaný iný graf G. Toho orientované hrana $x \to y$ nám hovorí, že sme počuli x chváliť sa, že je lepší ako y – teda že v neznámom grafe Z musí existovať nejaká cesta z x do y.
Našou úlohou je zistiť, koľko najmenej hrán môže mať graf Z zodpovedajúci vstupu.

Terminologická odbočka

Ak poznáme graf Z, vieme si k nemu zostrojiť graf $T(Z)$ popisujúci, odkiaľ kam v Z vedie cesta. Grafu $T(Z)$ hovoríme *tranzitívny uzáver* grafu Z. Priamočiary spôsob, ako tranzitívny uzáver zostrojiť, spočíva v tom, že do Z postupne pridávame nové hrany: vždy, keď máme hrany $x \to y$ a $y \to z$, musíme mať aj hranu $x \to z$. Keď sa už žiadna nová hrana nedá týmto pravidlom pridať, sme hotoví.
(Efektívnejší algoritmus na zostrojenie tranzitívneho uzáveru sa dá založiť napr. na postupnom prehľadávaní pôvodného grafu z každého vrcholu. Inou možnosťou je Floydov algoritmus, ktorý je jednoduchšou verziou Floydovho-Warshallovho algoritmu pre výpočet najkratších ciest.)
Naša úloha sa teda dá sformulovať nasledovne: Daný je orientovaný graf G. Prípustný graf Z je taký, ktorého tranzitívny uzáver obsahuje všetky hrany G. Spomedzi všetkých prípustných Z chceme nájsť ten s najmenším počtom hrán.

Základná myšlienka

Aj keby bolo m oveľa väčšie ako n, zjavne vždy existuje prípustný graf Z s n hranami: Ak 1 vyhral nad 2, 2 nad 3, ..., $n-1$ vyhral nad n a n vyhral nad 1, môže sa chváliť každý o každom inom. Takýto graf Z je teda úplne vždy prípustným riešením. Našim cieľom bude zistiť, kedy nám stačí ešte menší počet zápasov.

Slabo súvislé komponenty

Rozdeľme si vstupný graf G na časti, ktoré „držia pokope“ – ako keby sme zabudli na smery hrán a iba našli komponenty súvislosti. Takýmto častiam orientovaného grafu niekedy hovoríme *slabo súvislé komponenty*.
Tvrdíme teraz, že každý slabo súvislý komponent G_0 grafu G sa musí celý nachádzať v tom istom slabo súvislom komponente grafu Z.
Sporom. Ak by to tak nebolo, predstavme si, že sa postupne prechádzame po celom G_0 a pozeráme sa, v ktorom slabo súvislom komponente Z sme. Keďže nie všetky vrcholy G_0 sú v tom istom komponente Z, skôr či neskôr opustíme ten, v ktorom sme začínali. V tomto okamihu sme našli dvojicu vrcholov x, y, ktoré sú v G_0 spojené hranou $x \to y$, zatiaľ čo v Z sú v rôznych slabo súvislých komponentoch. Tu ale dostávame hľadaný spor – ak sú v rôznych komponentoch Z, tak určite medzi nimi v Z nevedie cesta, a teda sa x nevie chváliť, že je lepší ako y.

Riešenie pre jeden slabo súvislý komponent

Predpokladajme zatiaľ pre jednoduchosť, že celý vstupný graf G „drží pokope“. Vtedy vieme, že musíme vyrobiť nejaký Z, ktorý tiež celý „drží pokope“.
Ak máme n vrcholov, musíme mať aspoň $n-1$ hrán. Totiž pridanie hrany zmenší počet komponentov nanajvýš o 1, takže ak sa chceme dostať od n izolovaných vrcholov ku grafu s jediným komponentom, potrebujeme mať aspoň $n-1$ hrán.
Sú teda len dve možnosti: buď je jedným optimálnym Z vyššie spomínaný cyklus idúci cez všetky vrcholy, alebo má optimálny Z presne $n-1$ hrán.
Ak G obsahuje akýkoľvek cyklus, nastáva prvý prípad. Totiž potrebujeme, aby aj Z obsahoval nejaký cyklus a taký graf musí mať aspoň n hrán.

Naopak, ak je G acyklický, vždy nám stačí len $n-1$ hrán v Z. Totiž pre acyklický graf vieme nájsť tzv. *topologické usporiadanie* jeho vrcholov – teda také ich poradie, že každá hrana vedie zo skoršieho do neskoršieho. (Dôkaz a konštrukciu popíšeme nižšie.) A keď už máme topologické usporiadanie vrcholov $a_1, \ldots, a_n$, tak stačí zobrať graf Z s hranami $a_1 \to a_2$, $a_2 \to a_3$, $\ldots$, $a_{n-1} \to a_n$.

Všeobecné riešenie

Optimálnym riešením pre všeobecné grafy je jednoducho každý ich slabo súvislý komponent vyriešiť zvlášť. Totiž ľahko overíme, že ak by sme chceli viaceré komponenty G nahradiť jedným v Z, nikdy tým neušetríme. (Ak spájané komponenty všetky obsahovali cykly, ani nič nestratíme, ak boli niektoré acyklické, tak nové riešenie bude horšie.)
To nám teda dáva návod, ako zistiť optimálny počet hrán: rozdelíme zadaný graf G na slabo súvislé komponenty, do premennej a spočítame, koľko z nich je acyklických, a následne odpovieme, že optimálny graf Z má presne $n - a$ hrán.

Overovanie acyklickosti grafu

Overiť, či je graf acyklický, vieme efektívne napríklad použitím prehľadávania do hĺbky.
Pomáha napríklad metafora, keď si predstavíme, že vrcholy grafu sú rôzne softvérové balíčky a hrany predstavujú závislosti medzi nimi: $x \to y$ hovorí, že ak chceš nainštalovať y, potrebuješ mať najskôr nainštalovaný x. Topologické usporiadanie grafu potom zodpovedá jednému možnému poradiu, v ktorom sa dá všetok softvér korektne nainštalovať.
Ako overiť, či nejaké takéto poradie existuje, a prípadne rovno aj jedno nájsť? Stačí zobrať jednoduchú rekurzívnu funkciu `nainstaluj(y)`, ktorá chce nainštalovať softvér y. Táto funkcia bude fungovať nasledovne:

1. Pozri sa, či už náhodou y nie je označený ako čakajúci na inštalovanie. Ak áno, našli sme cyklus v závislostiach – skončíme a podáme správu, že topologické usporiadanie neexistuje.
2. Označ y ako čakajúci na inštalovanie.
3. Pre každé x také, že máme závislosť (hranu) $x \to y$: rekurzívne zavolaj `nainstaluj(x)`.
4. Ak sa úspešne podarilo nainštalovať všetky závislosti, nainštaluj y.

Keď už máme túto funkciu naprogramovanú, stačí raz prejsť zoznam softvéru a vždy, keď narazíme na nejaký nenainštalovaný, zavolať ju preň. Ak je graf závislostí acyklický, týmto postupom postupne všetok softvér nainštalujeme a poradie, v ktorom sa tak stalo, je jedným z možných topologických usporiadaní. Ak je v závislostiach cyklus, algoritmus naň počas behu natrafí a podá o tom správu.
Pri dobrej implementácii má toto riešenie časovú zložitosť priamo úmernú veľkosti spracúvaného grafu, teda $O(n + m)$.
Listing programu uvedený nižšie ukazuje trochu iné možné algoritmické techniky: slabo súvislé komponenty hľadá prehľadávaním do šírky a následne orientované cykly v nich hľadá tak, že vyrába topologické usporiadanie, kým to ide (postupom „kým máš vrchol do ktorého nevchádza hrana, pridaj ho na koniec topologického poradia a odstráň ho z grafu“) a pamätá si, z ktorého komponentu koľko vrcholov už spracoval.

Listing programu (C++)

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

int main() {
    int N, M, c=-1;
    cin >> N >> M;

    vector<int> indegree(N), component(N,-1), compsize(N,0), processed(N,0);
    vector< vector<int> > G(N), UG(N);

    // nacitaj vstup a vyrob grafy
    while (M--) {
        int x, y; cin >> x >> y; --x; --y;
        G[x].push_back(y); UG[x].push_back(y); UG[y].push_back(x); ++indegree[y];
```

```
    }

    // prehladavanim do sirky ofarbi slabo suvisle komponenty a zisti ich velkosti
    for (int n=0; n<N; ++n) if (component[n] == -1) {
        queue<int> Q; Q.push(n); component[n] = ++c;
        while (!Q.empty()) {
            int x=Q.front(); Q.pop(); ++compsize[c];
            for (int y:UG[x]) if (component[y]==-1) { component[y]=c; Q.push(y); }
        }
    }

    // vyrob z topologickeho usporiadania kolko sa da
    queue<int> Q; for (int n=0; n<N; ++n) if (indegree[n]==0) Q.push(n);
    while (!Q.empty()) {
        int x=Q.front(); Q.pop(); ++processed[component[x]];
        for (int y:G[x]) { --indegree[y]; if (indegree[y]==0) Q.push(y); }
    }

    // odpoved je N minus pocet komponentov ktore cele maju topologicke usporiadanie
    int answer = N;
    for (int i=0; i<=c; ++i) if (compsize[i] == processed[i]) --answer;
    cout << answer << endl;
}
```

A-III-5 Elektrárne

Ukážeme si najskôr špecifické riešenia pre jednotlivé podúlohy a potom riešenie druhej podúlohy zovšeobecníme.

Prvá podúloha: na ceste

Pre cestu vieme úlohu riešiť dynamickým programovaním. Postupne pôjdeme po ceste zľava doprava a pre každý vrchol i si budeme počítať dve hodnoty:

- Aká je najmenšia cena, za ktorú vieme vo vrcholoch 1 až i postaviť elektrárne a medzi nimi vedenia tak, aby bolo všetko elektrifikované.
- Aká je najmenšia cena pre také postavenie elektrární a vedení, pre ktoré je elektrifikované všetko až na komponent obsahujúci i, ktorý byť elektrifikovaný ešte nemusí.

Ak máme len jeden vrchol, prvá hodnota je cena elektrárne v ňom a druhá je nula. Následne pre každý ďalší vrchol vypočítame nové hodnoty na základe nasledujúcej úvahy:

- Pre prvú hodnotu máme tri možnosti:
  - buď vyriešim prvých $i-1$ a potom vo vrchole i spravím elektráreň,
  - alebo vyriešim prvých $i-1$ a vrchol i pripojím k vrcholu $i-1$,
  - alebo vyriešim prvých $i-1$ s tým, že posledný komponent ešte nemá elektrinu, a k tomu pridám aj elektráreň v i aj vedenie medzi i a $i-1$.
- Pre druhú hodnotu mám dve možnosti:
  - buď vyriešim prvých $i-1$ a pre vrchol i nespravím nič (vznikne nový komponent),
  - alebo vyriešim prvých $i-1$ s tým, že posledný komponent nemá elektrinu, a potom pripojím i ku $i-1$

Takto celú úlohu vyriešime v lineárnom čase.

Druhá podúloha: drahé elektrárne

Už vo všeobecnej verzii úlohy platia nasledujúce pozorovania:
Prvé: Predstavme si, že sme sa už rozhodli, ktoré vedenia nakúpiť. Potom v každom z komponentov chceme postaviť práve jednu elektráreň, a to v tom jeho vrchole, v ktorom je to najlacnejšie.
Druhé: Predstavme si, že sme sa už rozhodli, ako budú vrcholy zadaného grafu rozdelené do komponentov. Potom v každom komponente chceme nakúpiť vedenia tak, aby sme ju poprepájali a v súčte nás to stálo čo najmenej – čiže v každom komponente chceme nakúpiť hrany predstavujúce ľubovoľnú jednu jeho najlacnejšiu kostru.
V tejto podúlohe si potom navyše môžeme uvedomiť, že v každom komponente grafu zadaného na vstupe stačí nájsť jeho najlacnejšiu kostru a následne postaviť jednu najlacnejšiu elektráreň.

Ak by sme totiž mali ľubovoľné riešenie, ktoré bude mať viac komponentov postaveného vedenia, budú v ňom určite existovať dva komponenty vedení, ktoré sa dajú prepojiť pridaním ďalšieho vedenia. A keď tak spravíme, môžeme ušetriť za jednu elektráreň, čo dokopy riešenie určite zlepší – keďže každé vedenie je lacnejšie ako každá elektráreň.

Všeobecné riešenie

Predstavme si, že namiesto stavania elektrární vo vrcholoch existuje nový vrchol číslo $n+1$, v ktorom už je postavená jedna centrálna elektráreň. Pre každý existujúci vrchol môžeme postaviť vedenie, ktoré ho prepojí s vrcholom $n+1$, a cena tohto vedenia zodpovedá cene, za ktorú sme pôvodne vedeli postaviť elektráreň.
Na jednej strane vidíme, že optimálne riešenie pre túto novú situáciu bude mať rovnakú cenu ako optimálne riešenie pôvodnej úlohy. No a na druhej strane vidíme, že v novej úlohe len potrebujeme všetkých $n+1$ vrcholov čo najlacnejšie prepojiť vedeniami. Takže už len stačí nájsť najlacnejšiu kostru nového grafu. Ten má $n+1$ vrcholov a $m+n$ hrán, efektívnym algoritmom to preň vieme spraviť v čase $O((m+n)\log n)$.

Algoritmy pre najlacnejšiu kostru

Dva najznámejšie algoritmy pre najlacnejšiu kostru sú Kruskalov a Jarníkov-Primov. Oba sa dajú implementovať s vyššie uvedenou časovou zložitosťou.
Oba algoritmy sú založené na nasledujúcej myšlienke: Ak vrcholy súvislého grafu rozdelíme na dve kôpky a všimneme si celkovo najlacnejšiu hranu e, ktorá má každý koniec na inej kôpke, tak existuje najlacnejšia kostra obsahujúca túto hranu. Dôkaz tohto tvrdenia vyplýva z toho, že ak zoberieme ľubovoľnú kostru, ktorá našu hranu neobsahuje, a pridáme do nej našu hranu, vznikne v nej cyklus. Na tomto cykle musí byť ešte aspoň jedna iná hrana, ktorá má každý koniec na inej kôpke. Ak túto teraz z kostry vyhodíme, tak sme dostali novú kostru, ktorá obsahuje e a je aspoň rovnako dobrá (keďže vyhodená hrana nemohla byť lacnejšia ako e).

Kruskalov algoritmus funguje tak, že všetky hrany grafu usporiadame od najlacnejšej po najdrahšiu. Teraz začneme v stave, že nič s ničím nie je prepojené, a postupne sa pozrieme na každú hranu. Ak by jej pridanie spojilo dva komponenty, spravíme to (z vyššie uvedeného tvrdenia vieme, že môžeme), a naopak, ak už oba konce práve spracúvanej hrany ležia v tom istom komponente, tak jej pridanie nič nezmení a môžeme ju zahodiť.
Na dobrú časovú zložitosť potrebujeme efektívne udržiavať, ktorý vrchol je v ktorom komponente. To vieme robiť buď tak, že vždy keď spájame komponenty, tak „prefarbíme“ menší na farbu väčšieho, alebo použitím algoritmu Union-Find.

Jarníkov-Primov algoritmus funguje tak, že na začiatku jeden vrchol prehlásime za kostru a ostatné za nepripojené. Pre každý nepripojený vrchol si pamätáme najlacnejšiu hranu z neho do kostry. Teraz dokola opakujeme nasledovnú úvahu: zoberieme najlacnejšiu hranu medzi kostrou a zvyškom grafu (čiže najlacnejšiu zo zapamätaných hrán), pridáme ju do kostry a následne prepočítame ceny zapamätaných hrán pre nepripojené vrcholy (keďže hrany z nich do vrcholu práve pridaného do kostry môžu byť lacnejšie ako tie zapamätané).
Na dobrú časovú zložitosť potrebujeme vedieť efektívne nájsť najbližší vrchol, ktorý pripojiť ku kostre. Na to vieme použiť prioritnú frontu. Výsledná implementácia je veľmi podobná efektívnej implementácii Dijkstrovho algoritmu.

V prvom listingu uvedenom nižšie je implementovaný druhý z týchto algoritmov.

Listing programu (C++)

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

typedef pair<int,int> zaznam;

int main() {
    priority_queue< zaznam, vector<zaznam>, greater<zaznam> > Q;

    int N, M;
    cin >> N >> M;
```

```
    vector<int> cena_pripojenia(N);
    for (int n=0; n<N; ++n) {
        cin >> cena_pripojenia[n];
        Q.push( { cena_pripojenia[n], n } );
    }

    vector< vector<int> > sused(N), cena_vedenia(N);
    for (int m=0; m<M; ++m) {
        int x, y, c;
        cin >> x >> y >> c;
        --x; --y;
        sused[x].push_back(y); cena_vedenia[x].push_back(c);
        sused[y].push_back(x); cena_vedenia[y].push_back(c);
    }

    long long odpoved = 0;
    vector<bool> pripojene(N, false);
    while (!Q.empty()) {
        int osada = Q.top().second;
        if (pripojene[osada]) { Q.pop(); continue; }
        pripojene[osada] = true;
        odpoved += Q.top().first;
        Q.pop();
        for (unsigned i=0; i<sused[osada].size(); ++i) {
            int s = sused[osada][i], c = cena_vedenia[osada][i];
            if (cena_pripojenia[s] > c) {
                cena_pripojenia[s] = c;
                Q.push( {c,s} );
            }
        }
    }

    cout << odpoved << endl;
}
```

A ešte stručný listing programu s Kruskalovym algoritmom a algoritmom union-find:

Listing programu (C++)

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

vector<int> boss;
int get(int x) { if (x==boss[x]) return x; return boss[x]=get(boss[x]); }

int main() {
    int N, M; cin >> N >> M;
    boss.resize(N+1); iota(boss.begin(),boss.end(),0);
    vector< tuple<int,int,int> > E;
    for (int n=1; n<=N; ++n) { int e; cin >> e; E.push_back({e,0,n}); }
    while (M--) { int x, y, c; cin >> x >> y >> c; E.push_back({c,x,y}); }
    sort( E.begin(), E.end() );
    long long answer = 0;
    for (auto e:E) {
        int c,x,y; tie(c,x,y) = e; x=get(x); y=get(y);
        if (x != y) { answer += c; if (rand()&1) swap(x,y); boss[x]=y; }
    }
    cout << answer << endl;
}
```

A-III-6 Stánky

V prvej sade funguje úplne ľubovoľné použitie hrubej sily. Na druhú sadu stačilo napríklad pre každú minútu vstupu prejsť zoznam otváracích hodín (ten je krátky) a tak zistiť, koľko stánkov stretneme otvorených.
Skôr, než sa pustíme do lepších algoritmov, si vstup upravme: keďže vieme, že ku stánku i sa dostaneme až po $i-1$ minútach od vstupu, môžeme si všetky jeho intervaly posunúť v čase o $i-1$ minút späť. Tým dostaneme novú úlohu, v ktorej akoby sme pri všetkých stánkoch naraz. V tejto novej úlohe už teda jednoducho hľadáme tú minútu, počas ktorej je naraz otvorených najviac stánkov. (Presnejšie, pre každý stánok ešte musíme zobrať prienik jeho posunutého intervalu s intervalom od 1 po t, keďže len v tieto minúty smieme vojsť na trhy.)
Túto upravenú úlohu vieme riešiť napríklad zametaním. Vyrobíme si $2m$ udalostí: všetky začiatky a konce intervalov, teda okamihy, kedy nejaký stánok otvorí alebo zavrie. Tento zoznam si usporiadame podľa času. Ak sa v tom istom okamihu stánky zatvárajú aj otvárajú, zatvorenia dáme skôr – totiž my vieme prísť len počas celej minúty, takže ak jeden stánok zavrie a druhý v tom istom okamihu otvorí, nevieme ich vidieť otvorené oba.

Následne udalosti v takto usporiadanom poradí prejdeme a tým efektívne odsimulujeme otváranie a zatváranie stánkov v čase $O(m \log m)$.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n, t, m;
    cin >> n >> t >> m;

    vector<pair<int, int>> p;

    for (int i = 0; i < m; i++) {
        int s, z, k;
        cin >> s >> z >> k;

        // Posunieme časy
        int t1 = z - s + 1;
        int t2 = k - s + 1;

        // Orežeme interval na platný rozsah
        t1 = max(t1, 1);
        t2 = min(t2, t);

        // Ak nám niečo ostalo, vytvoríme príslušné udalosti
        if (t1 <= t2) {
            p.push_back({t1,   +1});
            p.push_back({t2+1, -1});
        }
    }

    // Naše udalosti sú (čas, -1) pre zatváranie a (čas, +1) pre otváranie stánku.
    // std::pair sa porovnávajú lexikograficky, takže sort nám udalosti s rovnakým časom
    // usporiada tak, ako chceme: najskôr konce a až potom začiatky.
    sort(p.begin(), p.end());

    int vysledok = 0;
    int otvorene = 0;

    for (auto& udalost : p) {
        otvorene += udalost.second;
        vysledok = max(vysledok, otvorene);
    }

    cout << vysledok << endl;
}
```

Iná, trikovejšia a stručnejšia implementácia:

Listing programu (C++)

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

int main() {
    int N, T, M; cin >> N >> T >> M;
    map<int,int> E; E[0] = 0;
    while (M--) { int s, z, k; cin >> s >> z >> k; ++E[z-s]; --E[k-s+1]; }
    int answer = 0, in = 0;
    for (auto e : E) { in += e.second; if (0 <= e.first && e.first < T) answer = max(answer,in); }
    cout << answer << endl;
}
```

TRIDSIATY ŠIESTY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2021