P-III-4 Stolní tenis

V řešení budeme používat terminologii teorie grafů, kterou naleznete např. v kuchařce KSP o grafech.* Máme zadán tzv. *orientovaný graf* G s $|V(G)| = n$. Naším úkolem je zkonstruovat orientovaný graf H na stejné množině *vrcholů* s co nejmenším počtem *hran* tak, aby platilo následující: Jestliže G obsahuje orientovanou hranu uv, tak H musí obsahovat posloupnost vrcholů $u = w_1, w_2, \ldots, w_t = v$ tak, aby $w_1w_2, w_2w_3, w_{t-1}w_t$ byly hrany H. Takové hraně uv v takovém případě říkáme *tranzitivní hrana* orientovaného grafu H.

Vysvětlíme rovnou řešení, které by dostalo plný počet bodů. Nejprve načteme orientovaný graf do paměti a najdeme jeho *slabě souvislé komponenty* – to jsou komponenty souvislosti grafu, který bychom z G dostali, kdybychom zapomněli na orientaci hran. Dále pro každou komponentu souvislosti ověříme, zda obsahuje *orientovaný cyklus* – to je nějaká posloupnost vrcholů $w_1, w_2, \ldots, w_t = w_1$ taková, že daná komponenta obsahuje hrany $w_1w_2, w_2w_3, \ldots, w_{t-1}w_t$. Označme $K_1, K_2, \ldots, K_k$ slabě souvislé komponenty, které žádný orientovaný cyklus neobsahují, a $L_1, L_2, \ldots, L_\ell$ komponenty, které nějaký orientovaný cyklus obsahují. Tvrdíme, že řešením úlohy je vypsat číslo $n-k$. Musíme dokázat, že pro každý graf G lze úlohu vyřešit s použitím $n-k$ hran, ale $n-k-1$ hran nikdy nestačí.

Ze všeho nejdřív ale zdůvodněme, že celý algoritmus má lineární složitost. Rozklad G na slabě souvislé komponenty lze udělat jednoduše v lineárním čase pomocí prohledávání do hloubky. Zbývá v lineárním čase ověřit, zda daný orientovaný graf G' (jedna komponenta G) obsahuje cyklus – jestli ne, jedná se o *orientovaný acyklický graf*. To lze udělat pomocí tzv. *topologického třídění*, které nyní stručně vysvětlíme, protože se bude hodit i později. Podrobnější popis naleznete třeba v sekci Topologické uspořádání ve zmiňované kuchařce.

Použijeme následující pozorování: neobsahuje-li G' cyklus, můžeme začít v libovolném jeho vrcholu a putovat proti orientaci hran G'. Nemůžeme se do stejného vrcholu dostat dvakrát, a tak nakonec skončíme v nějakém vrcholu G', do nějž nevedou žádné hrany. Každý acyklický orientovaný graf G', stejně jako každý jeho *podgraf*, obsahuje proto vrchol, do nějž nevedou žádné hrany.

Náš algoritmus si tak nejprve vytvoří seznam S vrcholů G', do nichž nevedou žádné hrany, a tento seznam následně budeme procházet. Pokaždé, když uvažujeme nějaký vrchol v na seznamu, tento vrchol vymažeme spolu se všemi hranami, které z v vycházejí. Pro vrcholy, do nichž mazané hrany vedou, ověříme, zda do nich vede alespoň jedna jiná hrana. Pokud tomu tak není, přidáme je do S. Ve chvíli, kdy se tento proces zastaví, jsme možná nevymazali všechny vrcholy G' a v takovém případě nám naše pozorování říká, že G' obsahuje orientovaný cyklus. Naopak jestliže jsme

* Viz *https://ksp.mff.cuni.cz/kucharky/grafy/*.

vymazali všechny vrcholy G', můžeme si tyto vrcholy označit jako $v_1, v_2, \ldots, v_{|V(G')|}$ v pořadí, jakém je náš algoritmus mazal. Všechny hrany grafu G' vedou z vrcholu s nižším pořadovým číslem do vrcholu s vyšším pořadovým číslem, takže G' zřejmě žádný cyklus obsahovat nemůže. Náš algoritmus tak v čase $O(|V(G')| + |E(G')|)$ ověří, zda G' obsahuje cyklus. Jestliže G' žádný cyklus neobsahuje, seřadíme navíc vrcholy G' tak, že hrany G' vedou z vrcholů s nižším pořadovým číslem do vrcholů s vyšším pořadovým číslem. Celý algoritmus může být implementován v lineárním čase $O(n + m)$ (viz zdrojový kód).

Nyní se vraťme k hlavním části důkazu. Nejprve dokážeme, že lze vždy sestrojit graf H s $n - k$ hranami, který vyhovuje zadání. Takový H může vypadat následovně: pro každou komponentu L_i s $|V(L_i)|$ vrcholy $v_1, v_2, \ldots, v_{|V(L_i)|}$ do H vložíme $|V(L_i)|$ hran $v_1v_2, v_2v_3, \ldots, v_{|V(L_i)|-1}v_{|V(L_i)|}, v_{|V(L_i)|}v_1$. Každé dva vrcholy L_i jsou v cyklu propojeny orientovanou cestou, takže speciálně každá hrana L_i je tranzitivní hranou H.

Dále ukážeme, že pro každou komponentu K_i ušetříme navíc oproti předchozí konstrukci jednu hranu. Pro každou komponentu K_i s $|V(K_i)|$ vrcholy, která tvoří acyklický orientovaný graf, použijeme topologické třídění, kterým dostaneme pořadí vrcholů $v_1, v_2, \ldots, v_{|V(K_i)|}$ takové, že všechny hrany K_i vedou z vrcholu s nižším pořadovým číslem do vrcholu s vyšším pořadovým číslem. Vložíme-li tedy do H celkem $(|V(K_i)| - 1)$ hran $v_1v_2, v_2v_3, \ldots, v_{|V(K_i)|-1}v_{|V(K_i)|}$, opět zařídíme, že každá hrana K_i je tranzitivní hranou H.

Dostali jsme tedy graf H, který splňuje podmínky zadání a obsahuje $|V(L_1)| + |V(L_2)| + \ldots + |V(L_\ell)| + (|V(K_1)| - 1) + (|V(K_2)| - 1) + \ldots + (|V(K_k)| - 1) = n - k$ hran.

Na druhou stranu je potřeba dokázat, že každý graf H splňující zadání má alespoň $n - k$ hran. Uvažme proto libovolný takový H. Nejprve si povšimněme, že všechny vrcholy komponenty K_i grafu G musí ležet ve stejné slabě souvislé komponentě H. Kdyby tomu tak nebylo, mohli bychom najít dva vrcholy $u, v \in V(K_i)$ spojené hranou v K_i takové, že tyto vrcholy v H leží ve dvou různých slabě souvislých komponentách. To ale znamená, že uv není tranzitivní hranou H, což je spor s naším předpokladem, že H splňuje zadání.

Dostáváme, že každá slabě souvislá komponenta H je sjednocením několika slabě souvislých komponent G. Komponenty grafu H si označíme $K'_1, K'_2, \ldots, K'_{k'}$ a $L'_1, L'_2, \ldots, L'_{\ell'}$, přičemž komponenty L'_i jsou ty, které jsou nadmnožinou některé komponenty L_j, a K'_i jsou ty zbylé. Z toho plyne, že $k' \leq k$.

Pro každou komponentu K'_i platí, že $|E(K'_i)| \geq |V(K'_i)| - 1$, jinak by K'_i nebyla souvislá. Tvrdíme, že pro každou L'_i platí, že $|E(L'_i)| \geq |V(L'_i)|$. Pro spor předpokládejme opak, tedy $|E(L'_i)| \leq |V(L'_i)| - 1$. To vzhledem k tomu, že L'_i je slabě souvislý, implikuje, že hrany L'_i tvoří *strom*. Komponenta L'_i grafu H je nadmnožinou nějaké komponenty L_i grafu G, která obsahuje orientovaný cyklus z hran $w_1w_2, w_2w_3, \ldots, w_tw_1$. V původním grafu G platí, že obejitím tohoto cyklu se dostaneme z vrcholu w_1 zpět do téhož vrcholu. Protože v grafu H můžeme každou z hran tohoto cyklu nahradit orientovanou cestou spojující tytéž vrcholy, musí i v grafu H

existovat procházka po směru hran začínající a končící ve w_1. To je ale ve sporu s tím, že H je strom: jakmile se z w_1 vydáme do sousedního vrcholu, musí naše procházka později použít tutéž hranu, ale v opačném směru.

Dokázali jsme tedy, že pro každý graf H, který splňuje zadání, platí, že počet hran H je alespoň $|V(L'_1)| + |V(L'_2)| + \ldots + |V(L'_{\ell'})| + (|V(K'_1)| - 1) + (|V(K'_2)| - 1) + \ldots + (|V(K'_k)| - 1) = n - k' \geq n - k$. To ukončuje důkaz správnosti našeho algoritmu.

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

using namespace std;

int n, m;
vector<vector<int>> G;   // Graf na vstupu
vector<vector<int>> Gr;  // Graf s opačnou orientací než G
vector<int> components;  // Index komponenty každého vrcholu
vector<int> indegree;    // Počet vcházejících hran každého vrcholu

void find_components (int u, int comp) { // Hledání slabě souvislých komponent
    if (components[u] != -1)
        return;

    components[u] = comp;

    for (auto v: G[u])
        find_components(v, comp);
    for (auto v: Gr[u])
        find_components(v, comp);
}

int main () {
    cin >> n >> m;
    G.resize(n), Gr.resize(n), indegree.resize(n, 0), components.resize(n, -1);

    for (int i = 0; i < m; ++i) {
        int u, v;
        cin >> u >> v;
        --u, --v;
        G[u].push_back(v);
        Gr[v].push_back(u);
        ++indegree[v];
    }

    int numcomps = 0; // Počet slabě souvislých komponent G
    for (int u = 0; u < n; ++u) {
        if (components[u] == -1) {
            // Není-li v žádné dosud nalezené komponentě, projdeme ji
            find_components(u, numcomps);
            ++numcomps;
        }
    }

    vector <int> s; // Zásobník vrcholů, do nichž již nevedou žádné hrany
    for (int u = 0; u < n; ++u) {
        if (Gr[u].empty()) {
            s.push_back(u);
        }
```

```
    }
    while (!s.empty()) {
        // Postupně odstraňujeme vrcholy, do kterých již nevede hrana
        int u = s.back();
        s.pop_back();
        for (auto v: G[u]) {
            --indegree[v];
            if (indegree[v] == 0) {
                s.push_back(v);
            }
        }
    }

    vector<bool> acyclic(numcomps, true); // Acyklické komponenty jsou nyní prázdné
    for (int u = 0; u < n; ++u) {
        if (indegree[u] > 0) {
            acyclic[components[u]] = false;
        }
    }

    int ans = n; // Od počtu vrcholů odečteme počet acyklických komponent
    for (int i = 0; i < numcomps; ++i) {
        if (acyclic[i]) {
            --ans;
        }
    }

    cout << ans << endl;
}
```

P-III-5 Elektrárny

První podúloha

První tři body šly získat za pomoci dynamického programování. Označme si a_i cenu elektrárny v i-tém městě a pro $2 \leq i \leq n$ označme b_i cenu propojení i-tého města s tím předchozím. Projdeme města od 1 do n a pro každé $1 \leq i \leq n$ si spočítáme dvě hodnoty $A[i]$ a $B[i]$ definované následovně: $A[i]$ je nejmenší cena, za kterou můžeme postavit elektrárny a spojit města na prefixu od 1 do i tak, že všechna města na tomto prefixu jsou elektrifikována. $B[i]$ je definováno stejně jako $A[i]$, ale komponenta, ve které je město i, elektrifikována být nemusí.

Platí $A[1] = a_1$ a $B[1] = 0$. Další hodnoty v těchto polích si následně můžeme spočítat následujícími rekurentními vztahy. Platí, že

$$A[i] = \min(A[i-1] + b[i], A[i-1] + a[i], B[i-1] + b[i] + a[i]),$$

protože aby bylo poslední město elektrifikováno, buď jej připojíme k jeho levému sousedu, který už je elektrifikovaný (první případ), nebo v tomto městě postavíme elektrárnu a předchozí města se elektrifikují bez naší pomoci (druhý případ), nebo v tomto městě postavíme elektrárnu a spojíme je s jeho levým sousedem, kterého tím elektrifikujeme (třetí případ). Podobně platí, že

$$B[i] = \min(B[i-1] + b[i], A[i-1]),$$

protože i když i-té město nemusí být elektrifikováno, $(i-1)$-té město buď musí být připojeno k i-tému, nebo musí být elektrifikováno.

Dynamickým programováním s využitím těchto rekurentních vztahů snadno spočítáme hodnotu $A[n]$, která je výsledkem. Celý algoritmus funguje v lineárním čase.

Dále budeme používat terminologii teorie grafů. Na vstupu jsme dostali ohodnocený graf. Naším úkolem bylo koupit některé jeho vrcholy a hrany tak, aby z každého vrcholu vedla cesta po koupených hranách do některého koupeného vrcholu. Přitom minimalizujeme celkovou cenu koupených vrcholů a hran.

Druhá podúloha

Tři body v druhé podúloze šly získat za následující řešení.

První důležité pozorování je, že kdykoliv se podíváme na jednu komponentu souvislosti nakoupených hran, musíme na této komponentě nakoupit alespoň jeden vrchol. Zároveň ale víc vrcholů kupovat nepotřebujeme, vyplatí se tedy vždy koupit nejlevnější vrchol této komponenty a žádný jiný. Pokud jsme si dále již vybrali, kterou množinu vrcholů propojíme nakoupením hran, nejlevnější možností je, když nakoupené hrany tvoří minimální kostru v podgrafu, který propojujeme.

Nyní si uvědomíme, že podmínka ze zadání zaručuje, že optimální řešení v každé komponentě souvislosti vstupního grafu najde její minimální kostru a nakoupí nejlevnější vrchol. To proto, že kdybychom v této komponentě nakoupili hrany tak, že vytvoří vícero komponent, mohli bychom nějaké dvě komponenty spojit zakoupením jedné nové hrany. Následně v jedné spojované komponentě nemusíme kupovat vrchol. Podmínka ze zadání zaručuje, že ušetříme.

Řešením je tedy najít v každé komponentě její nejlevnější vrchol a její minimální kostru. Tu najdeme třeba Kruskalovým algoritmem, který má časovou složitost $O(m \log n)$.

Třetí podúloha

I ve třetí podúloze budeme hledat minimální kostru, ovšem na jiném grafu. Představíme si, že místo toho, že máme možnost v každém městě postavit elektrárnu, existuje kromě n měst na vstupu také *hlavní město* $n+1$ a v tom již elektrárna stojí. Místo toho, abychom za cenu a_i nakoupili v i-tém městě elektrárnu, si budeme představovat, že za cenu a_i můžeme i-té město spojit s městem hlavním. Po této reformulaci je již úloha jednoduchá; ke vstupnímu grafu přidáme jeden speciální vrchol spojený hranou se všemi ostatními: cena i-té hrany je a_i. Nyní v tomto grafu chceme najít nejlevnější podmnožinu jeho hran takovou, že všechny vrcholy jsou propojeny se speciálním vrcholem, což jinak řečeno znamená, že hledáme nejlevnější souvislou podmnožinu. To je ale přesně minimální kostra, kterou nalezneme např. Kruskalovým algoritmem s časovou složitostí $O(m \log n)$.

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

int n, m;
vector<pair<int, pair<int,int>>> edges;  // Hrany ve formátu {váha, {u,v}}
```

```
// Disjoint - Find - Union
int parent[123456];
int rnk[123456];

int find (int u) {
    // Najde reprezentanta komponenty daného vrcholu, dělá kompresi cest
    if (u != parent[u]) parent[u] = find(parent[u]);
    return parent[u];
}

void merge (int u, int v) {
    // Spojí dvě komponenty do jedné, dělá Union podle ranku
    u = find(u);
    v = find(v);

    if (rnk[u] > rnk[v]) parent[v] = u;
    else parent[u] = v;

    if (rnk[u] == rnk[v]) rnk[v]++;
}

int main () {
    cin >> n >> m;
    for (int i = 1; i <= n; ++i) {
        // Ceny elektráren interpretujeme jako hrany do speciálního vrcholu 0
        int w;
        cin >> w;
        edges.push_back({w, {0, i}});
    }

    for (int i = 0; i < m; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        edges.push_back({w, {u, v}});
    }

    for (int i = 0; i <= n; i++) {
        // Inicializujeme strukturu DFU
        rnk[i] = 0;
        parent[i] = i;
    }

    long long ans = 0;  // Výsledná váha

    sort(edges.begin(), edges.end());  // Seřadíme hrany podle jejich váhy

    for (int i = 0; i < edges.size(); ++i) {
        int u = edges[i].second.first;
        int v = edges[i].second.second;
        int w = edges[i].first;

        int ru = find(u);  // Najdeme reprezentanty komponenty u a v
        int rv = find(v);

        if (ru != rv) {
            // Vrcholy zatím jsou v jiných komponentách
            ans += w;
            merge(ru, rv);
        }
    }
```

```
    cout << ans << endl;
    return 0;
}
```

P-III-6 Země živitelka

Dva body bylo možné získat za přímočarou simulaci: vyzkoušíme všechny časy vstupu mezi minutami 1 a m. Pro každý čas vstupu si jednoduše spočítáme pro každý stánek s, ve které minutě se v něm budeme vyskytovat. Pak projdeme všechny intervaly pro stánek s a zjistíme, jestli je momentálně přítomný majitel. Takto spočítáme počet získaných dárkových předmětů pro každou minutu a vezmeme maximum. Časová složitost je $\mathcal{O}(m(n+k))$.

Další dva body šlo získat chytřejším počítáním získaných dárkových předmětů pro daný čas vstupu. Místo toho, abychom procházeli každý stánek a zjišťovali, jestli je přítomný majitel, projdeme všechny trojice (s, o, d) z rozvrhu a pro každou se podíváme, jestli ke stánku s dorazíme v intervalu $\langle o, d\rangle$. Počet získaných dárečků je počet takovýchto trojic. Časová složitost je $\mathcal{O}(mk)$.

Redukce na průniky intervalů

Optimální řešení vyžaduje sofistikovanější přístup. Napřed si všimneme, že na číslu stánku vlastně moc nezáleží, protože můžeme snadno spočítat, kdy je potřeba vyrazit, abychom ke stánku přišli v daný čas. Konkrétněji, máme-li trojici (s, o, d) v rozvrhu, musíme vyrazit někdy mezi minutami $o - s + 1$ a $d - s + 1$, abychom se do intervalu strefili.

Toto pozorování nám umožňuje ignorovat čísla stánků a zredukovat úlohu na tuto: máme nějaké intervaly, kolik nejvíce intervalů se překrývá v jednom bodě? Převod uděláme tak, že každou trojici (s, o, d) konvertujeme na jeden interval $\langle o - s + 1, d - s + 1\rangle \cap \langle 1, m\rangle$. Průnik s $\langle 1, m\rangle$ odpovídá podmínce, že můžeme vyrazit pouze mezi minutami 1 a m.

Prefixové součty

Jak tedy rychle zjistit, kolik nejvíce intervalů se překrývá? Vytvoříme pomocné pole q, které inicializujeme nulami. Potom projdeme každý interval $\langle a, b\rangle$. K $q[a]$ přičteme 1 a od $q[b+1]$ naopak 1 odečteme. Všimneme si, že v čase i je počet otevřených (a dosud neuzavřených) intervalů rovný $q[1] + q[2] + \cdots + q[i]$. Chceme tedy spočítat tento součet pro každé i a vybrat maximum, čímž získáme nejvyšší počet překrývajících se intervalů.

To uděláme snadno pomocí principu prefixových součtů. Projdeme všechny minuty od 1 do m a budeme si v čase i pamatovat součet $c = q[1] + q[2] + \cdots + q[i]$. Na začátku (v pomyslné minutě 0) nastavíme $c = 0$. Když chceme přejít z minuty i na minutu $i+1$, stačí jednoduše nastavit $x \leftarrow x + q[i+1]$. Odpovědí je pak nejvyšší hodnota, kterou c v průběhu výpočtu mělo.

Celkově poběží algoritmus v čase $\mathcal{O}(m+k)$, což stačí na 7 bodů.

Optimální řešení

K vyřešení poslední sady testů použijeme stejnou myšlenku, jen ji trochu chytřeji implementujeme. Využijeme toho, že i když je m velké, většina prvků pole q bu-

de nulová: nenulových může být maximálně $2k$ prvků. Místo q použijeme seznam „událostí“ p. Události budou typu „začátek“ nebo „konec“. Za každý interval $\langle a, b\rangle$ přidáme do p událost (a, začátek) a událost ($b+1$, konec).

Pak seznam p setřídíme podle první složky (tedy podle času) a postupujeme podobně jako předtím s polem q. Budeme tedy udržovat průběžný součet c, za událost „začátek“ jej o 1 zvýšíme a za událost „konec“ zase snížíme. Výsledkem je opět nejvyšší hodnota, kterou c v průběhu výpočtu mělo. Při implementaci musíme brát v potaz to, že události se stejným časem se mají dít opravdu najednou. Například kdybychom měli události ((3, začátek), (3, začátek), (3, konec)) a před minutou 3 bychom měli $c = 2$, bude po minutě 3 platit $c = 3$, ale nesmíme si myslet, že jsme po prvních dvou událostech této minuty dosáhli hodnoty $c = 4$.

Událostí máme $2k$, takže jejich seznam p dokážeme setřídit v čase $\mathcal{O}(k \log k)$. To je i celková časová složitost, protože zbytek výpočtu už zvládneme v $\mathcal{O}(k)$. Paměťová složitost je $\mathcal{O}(k)$.

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

using namespace std;

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

    vector<pair<int, int>> p;

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

        // Konvertujeme časy
        int t1 = o - s + 1;
        int t2 = d - s + 1;

        // Ořežeme intervaly na platný rozsah
        t1 = max(t1, 1);
        t2 = min(t2, m);

        // Zbylo po ořezání něco?
        if (t1 <= t2) {
            // Typ události reprezentujeme jednoduše hodnotou, o kterou
            // se má změnit prefixový součet c.
            p.push_back({t1, +1});
            p.push_back({t2 + 1, -1});
        }
    }

    // std::pair se řadí lexikograficky, takže událost (3, -1) bude před (3, +1).
    // To vyřeší zmíněný problém s tím, že se události v jednom čase mají dít
    // "najednou", protože ‘c‘ pak můžeme jen podcenit, ale nikdy přecenit.
    sort(p.begin(), p.end());

    int vysledek = 0;
    int c = 0;

    // Projdi vzestupně všechny dvojice e == (čas, změna) v p
```

```
    for (auto& udalost : p) {
        c += udalost.second;
        vysledek = max(vysledek, c);
    }

    cout << vysledek << endl;
}
```