A-I-1 Rušenie ulíc

Poďme zostrojiť jednu možnú množinu ulíc, ktoré nechať – teda množinu ulíc, ktorá obsahuje všetky ulice, ktoré obsahovať musí, a navyše má tú vlastnosť, že z každej križovatky ich vychádza párny počet. V riešení použijeme terminológiu z teórie grafov, križovatky a ulice budeme teda nazývať vrcholmi a hranami.
Začnime tým, že do našej množiny zoberieme všetky hrany, ktoré zobrať musíme, a zmažeme ich z grafu. Tým dostaneme novú úlohu: „Daný je graf, ktorý má ku každému vrcholu predpísané, či z neho ešte chceme vybrať párny alebo nepárny počet hrán. Zistite, ktoré hrany vybrať tak, aby všetko bolo splnené.“ Inými slovami, máme vybrať nejaký podgraf, pričom máme predpísané parity stupňov vrcholov.
Je zjavné, že občas žiadne riešenie nebude existovať. Napríklad ak máme izolovaný vrchol, ktorý by chcel nepárny počet hrán, máme zjavne smolu. Tak isto máme smolu, ak máme dva vrcholy spojené jedinou hranou, pričom jeden chce mať párny a druhý nepárny počet hrán. Vo všeobecnosti je zjavné, že riešenie nemôže existovať, ak je v nejakom komponente súvislosti súčet predpísaných parít nepárny – totiž na začiatku sú všetky stupne vrcholov párne (nulové) a každým výberom hrany zmeníme paritu dvoch vrcholov, takže súčet stupňov vrcholov v komponente je vždy párny.
Ukážeme si, že vo všetkých ostatných prípadoch riešenie existuje. Navyše ukážeme, že vyššie uvedenú podmienku vôbec netreba testovať explicitne. Stačí, keď sa riešenie pokúsime zostrojiť, a ak sa nám to nepodarí, budeme vedieť, že to vôbec nešlo.
Myšlienka riešenia je jednoduchá. Zoberieme náš graf (ten, v ktorom už nie sú ulice obľúbené Kocúrkovskými radnými) a postupne každý jeho komponent *prehľadáme do hĺbky*.
Prehľadávanie do hĺbky je algoritmus, ktorý postupne navštívi všetky vrcholy daného komponentu. Jeho pseudokód je nasledovný:

```
prehľadaj(vrchol V):
    poznač si, že vo vrchole V si už bol
    pre každého suseda S vrcholu V:
        ak si ešte nebol vo vrchole S:
            prehľadaj(S)
```

Je zjavné, že na každý vrchol komponentu sa tento algoritmus pozrie práve raz, lebo akonáhle doň prvýkrát príde, označí si ho ako navštívený a odvtedy už doň znova nevkročí. Následne je zjavné, že sa tento algoritmus práve dvakrát pozrie na každú hranu v komponente – raz pri spracúvaní jej jedného konca, raz pri spracúvaní druhého. Dokopy má teda prehľadanie konkrétneho komponentu časovú zložitosť lineárnu od jeho veľkosti, čiže od súčtu počtov vrcholov a hrán v ňom.
Tento algoritmus ľahko upravíme tak, aby riešil našu úlohu. Najskôr si ale ukážme, ako ho upraviť tak, aby nám vyrobil jednu možnú *kostru* prehľadávaného komponentu – teda najmenšiu možnú množinu hrán, ktorá „ešte stále drží celý komponent pokope“. Toto sa robí ľahko, stačí si pre každý vrchol V pamätať jeho rodiča R (čiže vrchol, odkiaľ sme doň prišli) a pre každý vrchol V iný ako ten, kde sme začínali, vybrať práve hranu do jeho rodiča. V pseudokóde to vyzerá nasledovne:

```
prehľadaj(vrchol V, vrchol R):
    poznač si, že vo vrchole V si už bol
    pre každého suseda S vrcholu V:
        ak si ešte nebol vo vrchole S:
            prehľadaj(S, V)
    ak R nie je "nikto":
        zober hranu medzi V a R

# na začiatku stačí spustiť prehľadaj(V, nikto) pre ľubovoľný jeden vrchol V z komponentu
```

No a takéto prehľadávanie teraz už ľahko upravíme na riešenie našej súťažnej úlohy. Ostatné hrany okrem tých, ktoré by vybralo vyššie popísané riešenie, zahodíme. No a vždy, keď pri prehľadávaní opúšťame nejaký vrchol V, pozrieme sa, či má alebo nemá správnu paritu. Ak nemá, hranu z V do R zoberieme (a upravíme obom vrcholom paritu), ak má, hranu z V do R zahodíme. V oboch prípadoch tak zabezpečíme, že V už je hotový.

V každom komponente teda takto zaručene dosiahneme správnu paritu pre všetky vrcholy okrem toho, z ktorého sme začínali prehľadávanie. A tam už vieme, na čom sme: ak vyšiel správne, sme spokojní, a ak nie, vieme, že riešenie nemôže existovať, lebo to znamená, že v dotyčnom komponente sme museli mať predpísaný nepárny súčet stupňov.
Pri dobrej implementácii má toto riešenie časovú zložitosť $O(m+n)$, čiže lineárnu od veľkosti vstupu.

Listing programu (C++)

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

int N, M;
vector< vector<int> > graf; // pre každý vrchol zoznam jeho susedov
vector<bool> neparny;       // pre každý vrchol či mu ešte treba párny alebo nepárny počet hrán
vector<bool> navstivil;     // pre každý vrchol či sme ho už prehľadali
vector<pair<int,int> > riesenie;

void prehladaj(int v, int r) {
    navstivil[v] = true;
    for (int s : graf[v]) if (s != r) {
        if (!navstivil[s]) {
            prehladaj(s,v);
        } else {
            // túto hranu určite chceme zrušiť, ale do riešenia ju pridáme len ak s < v,
            // nech ju tam nemáme dvakrát
            if (s < v) riesenie.push_back( {s,v} );
        }
    }
    if (r != -1) {
        if (neparny[v]) {
            neparny[v] = !neparny[v]; neparny[r] = !neparny[r]; // zoberieme hranu do rodiča
        } else {
            riesenie.push_back( {v,r} ); // zrušíme hranu do rodiča
        }
    }
}

int main() {
    cin >> N >> M;
    graf.resize(N);
    neparny.resize(N,false);
    navstivil.resize(N,false);

    // načítaj prvých K hrán a zober ich -- teda len meň parity
    int K; cin >> K;
    for (int k=0; k<K; ++k) {
        int x, y;
        cin >> x >> y; --x; --y;
        neparny[x] = !neparny[x];
        neparny[y] = !neparny[y];
    }

    // načítaj ostatné hrany a ulož ich do grafu
    for (int k=0; k<M-K; ++k) {
        int x, y;
        cin >> x >> y; --x; --y;
        graf[x].push_back(y);
        graf[y].push_back(x);
    }

    // postupne sa pozri na všetky vrcholy
    // vždy, keď stretneš nenavštívený, našiel si nový komponent, prehľadaj ho
    for (int n=0; n<N; ++n) if (!navstivil[n]) {
        prehladaj(n,-1);
        if (neparny[n]) { cout << -1 << endl; return 0; } // riešenie neexistuje
    }

    // ak sme všetky vrcholy navštívili a zmenili na párne, vypíšeme riešenie
    cout << riesenie.size() << endl;
    for (auto hrana : riesenie) cout << hrana.first+1 << " " << hrana.second+1 << endl;
}
```

A-I-2 Cyklistický závod

Údaje zadané na vstupe si môžeme reprezentovať ako orientovaný graf. Pre každú križovatku budeme mať jeden vrchol a pre každú cestičku „x_i y_i l_i r_i“ si spravíme hranu z vrcholu x_i do vrcholu y_i s dĺžkou $l_i - r_i$. Dĺžka hrany nám teda hovorí, o koľko je na tomto úseku ľavá stopa dlhšia ako pravá.

Ak teraz v takomto grafe zoberieme nejaký cyklus, celkový rozdiel medzi dĺžkou ľavej a pravej stopy zistíme jednoducho tak, že sčítame dĺžky jeho hrán. Našou úlohou je teda zistiť, či v takomto grafe existuje nejaký cyklus nenulovej dĺžky, a ak áno, tak jeden nájsť.
Pre lepší popis riešenia si predstavme inú metaforu, ktorá vedie k tej istej úlohe: každý vrchol má nejakú (nám neznámu) nadmorskú výšku a každá hrana nám o dvojici vrcholov hovorí, o koľko je druhý vyššie ako prvý. Našou úlohou je overiť, či sú všetky informácie, ktoré máme, konzistentné, a ak nie sú, nájsť jeden cyklus obsahujúci spor.
S touto predstavou už začína byť jasné, ako zhruba postupovať. Vstup si rozbijeme na komponenty, ktoré sú od seba nezávislé. V každom komponente ľubovoľnému jednému vrcholu priradíme výšku 0 a postupne sa budeme pozerať na hrany, ktoré vedú do jeho susedov, do ich susedov, a tak ďalej, a odvádzať si, akú nadmorskú výšku musia mať. Ak sa nám všetky podarí odvodiť a všetky hrany budú sedieť, je všetko v poriadku. A ak nie, tak nájdeme nejakú hranu, ktorá nesedí.
Aké presne „komponenty“ ale potrebujeme a čo to znamená, že sú „nezávislé“? To, čo potrebujeme, sú presne tzv. *silno súvislé komponenty*. Dva vrcholy u a v ležia v tom istom silno súvislom komponente práve vtedy, keď sa po hranách vieme dostať aj z u do v, aj z v do u. Každý cyklus leží nutne celý v jednom silno súvislom komponente. Preto nám stačí zadaný graf rozdeliť na silno súvislé komponenty a potom o každom z nich zvlášť zistiť, či obsahuje hľadaný cyklus.

Na prvú časť úlohy, teda rozdelenie grafu na silno súvislé komponenty, existuje viacero rôznych klasických algoritmov. Viac o nich nájdete na https://en.wikipedia.org/wiki/Strongly_connected_components. Niektorí organizátori OI preferujú algoritmus, ktorý má na svedomí Tarjan, iní zase algoritmus, ktorý vymyslel Kosaraju. Oba tieto algoritmy pracujú v lineárnom čase od veľkosti grafu.

Druhá časť úlohy je potom už ľahká. Prehľadáme celý silno súvislý komponent z jedného jeho vrcholu do hĺbky. Tým si zvolíme jednu jeho kostru a podľa nej každému vrcholu priradíme jeho „nadmorskú výšku“. Teraz postupne prezrieme všetky ostatné hrany. Ak všetky sedia na vypočítané nadmorské výšky, je celý komponent v poriadku. A ak nájdeme nejakú, ktorá nesedí, potrebujeme teraz ukázať, že vždy vieme nájsť cyklus s nenulovým súčtom dĺžok.
Majme teda takúto hranu uv. Rozoberme tri možnosti vzhľadom na to, kde sa u a v nachádzajú v kostre.
Ak v je predkom u, je to jednoduché: cesta po kostre z v do u a hrana späť z u do v tvoria hľadaný cyklus.
Ak u je predkom v, nájdeme v našom komponente ľubovoľnú cestu z v do u. Túto cestu teraz vieme na cyklus doplniť dvoma rôznymi spôsobmi: buď do nej pridáme hranu uv, alebo cestu po kostre z u do v. Keďže tieto dva cykly majú rôznu dĺžku, aspoň jeden z nich môžeme vypísať.
NO a ostáva posledný prípad: uv je tzv. priečna hrana, teda ani jeden z jej vrcholov nie je predkom druhého. V takomto prípade opäť nájdeme dva cykly s rôznou cenou, a to tak, že nájdeme ľubovoľnú cestu z vrcholu v do koreňa kostry. Jeden cyklus je táto cesta + cesta po kostre z koreňa do v, druhý cyklus je táto cesta + cesta po kostre z koreňa do u + hrana uv.
Celková časová zložitosť vyššie popísaného algoritmu je $O(m + n)$.

A-I-3 Trojuholník

Na 4 body stačilo vyskúšať všetky trojice bodov, zakaždým vypočítať obsah trojuholnika a vybrať najmenší z nich. Na výpočet obsahu trojuholnika existuje viacero vzorcov. Ak poznáme súradnice jeho vrcholov, najjednoduchší je vzorec založený na vektorovom súčine. Pre dva vektory u a v v rovine vieme, že ich vektorový súčin je vektor, ktorý je na oba kolmý a ktorého veľkosť udáva (orientovaný) obsah rovnobežníka určeného týmito dvoma vektormi. Inými slovami, obsah trojuholníka ABC je rovný polovici veľkosti vektorového súčinu vektorov $B - A$ a $C - A$. Ešte inými slovami, keď máme trojuholník s vrcholmi $(0, 0)$, (x_1, y_1) a (x_2, y_2), jeho obsah je $(1/2) \cdot |x_1 y_2 - y_1 x_2|$.

Ako túto úlohu ale vyriešiť v čase požadovanom v zadaní, teda v čase $O(n^2 \log n)$?
Predstavme si, že sme si zvolili dva z troch vrcholov trojuholníka: A a B. Ktorý zo zvyšných $n - 2$ bodov má byť jeho tretím vrcholom C? Tentokrát sa oplatí použiť iný vzorec: obsah trojuholníka je $sv/2$, kde s je dĺžka

strany a v dĺžka výšky na ňu. Najmenší trojuholník teda vyrobí ten bod, z ktorého je na našu stranu najkratšia výška.
Uvažujme ľubovoľnú priamku kolmú na AB. To, čo nás zaujíma, je poradie, v akom na tejto priamke ležia priemety našich bodov. Totiž A a B sa premietnu do toho istého bodu a pre každý iný bod platí, že výška z neho na stranu AB má dĺžku rovnú vzdialenosti jeho priemetu od priemetu bodov A a B. Inými slovami, keby sme mali všetky body zoradené do poradia, v ktorom ležia na dotyčnej kolmici ich priemety, stačí nám uvažovať dva z nich: najbližší ku A a B z každej strany.
No a práve takéto poradia si vieme postupne všetky zostrojiť a pozrieť sa na ne.
Ako na to? Predstavme si, že priamku, na ktorú premietame body, budeme viesť cez začiatok sústavy súradníc. Na úvod si zvolíme nejaký smer priamky a premietneme na ňu body. Čo sa teraz stane, keď našu priamku začneme postupne okolo začiatku otáčať? Keď ju pootočíme len troška, tak sa aj priemety posunú len troška a nič sa nezmení. Kedy sa niečo zmení? Práve v tých chvíľach, ktoré nás zaujímajú – teda kedy je priamka kolmá na úsečku určenú dvoma bodmi. Vtedy sa ich priemety stretnú. A keď budeme pokračovať v otáčaní ďalej, ich priemety si vymenia poradie.
A to už nám dáva návod ako postupovať:

1. Vygenerujeme si všetkých $O(n^2)$ smerov priamky, ktorá je kolmá na niektorú z úsečiek určených dvomi zadanými bodmi.
2. Tieto smery si v čase $O(n^2 \log n)$ cyklicky usporiadame.
3. Pre jeden smer iný ako tie z minulého bodu si v čase $O(n^2)$ alebo lepšom zostrojíme (jednoznačne určené) poradie priemetov na priamke.
4. Postupne simulujeme otáčanie premietacej priamky okolo začiatku – prechádzame cez smery, kedy si body menia poradie, v usporiadanom poradí a každú zmenu vykonáme v konštantnom čase. Navyše po každej zmene sa pozrieme, ktoré dva trojuholníky obsahujú práve vymieňanú dvojicu bodov a majú najmenší obsah.

Pre zaujímavosť ešte uvedieme, že poznáme aj lepšie riešenie tejto úlohy. Presnejšie vieme ju riešiť v čase $O(n^2)$. Hlavným trikom pri tomto riešení je prechod do takzvanej duálnej roviny, v ktorom každý bod (a, b) zobrazíme na priamku $y = ax - b$ a naopak, každú nie-zvislú priamku zobrazíme na bod. Toto zobrazenie má veľa dobrých vlastností, napr. v nejakom zmysle zachováva orientáciu; veľa bodov na jednej priamke zobrazí na veľa priamok idúcich cez ten istý bod a podobne.
Zároveň existujú pomerne silné dôvody, prečo sa domnievať, že výrazne lepšie riešenie neexistuje. Táto úloha je totiž tzv. 3SUM-ťažká – teda keby sme ju vedeli riešiť efektívnejšie, vedeli by sme efektívnejšie riešiť aj nasledujúci problém: „Existujú medzi danými n číslami tri so súčtom nula?“ A tento problém síce vieme riešiť o malinký chlp lepšie ako v kvadratickom čase, ale rozhodne nečakáme, že sa objaví výrazne lepšie riešenie.
Do detailov už ale radšej nebudeme zachádzať.

Listing programu (C++)

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

struct udalost {
    int dx, dy; // smer pri ktorom nastane
    int a, b; // čísla bodov ktoré sa vymenia
};

// porovnávacia funkcia ktorá zoradí udalosti podľa uhlu z [0,360)
int polovica(const udalost &A) {
    return ( A.dy > 0 || (A.dy == 0 && A.dx > 0) ) ? 0 : 1;
}

bool skor(const udalost &A, const udalost &B) {
    if (polovica(A) != polovica(B)) return polovica(A) < polovica(B);
    return A.dx * B.dy - A.dy * B.dx > 0;
}

// funkcia pocitajuca dvojnasobok obsahu trojuholnika
```

```
int obsah(const vector<int> &X, const vector<int> &Y, int a, int b, int c) {
    int ux = X[b]-X[a], uy = Y[b]-Y[a], vx = X[c]-X[a], vy = Y[c]-Y[a];
    return abs( ux*vy - vx*uy );
}

int main() {
    // načítame vstup
    int N;
    cin >> N;
    vector<int> X(N), Y(N);
    for (int n=0; n<N; ++n) cin >> X[n] >> Y[n];

    // vygenerujeme všetky udalosti pri ktorých si dva body menia poradie
    vector<udalost> U;
    for (int a=0; a<N; ++a) for (int b=0; b<a; ++b) {
        int ux = X[b]-X[a], uy = Y[b]-Y[a];
        U.push_back( {-uy, ux, a, b} );
    }

    // usporiadame všetky udalosti
    sort( U.begin(), U.end(), skor );

    // zostrojíme poradie bodov tesne pred prvou udalosťou: premietneme body
    // na priamku idúcu cez bod (veľa,-1)
    vector< pair<int,int> > priemety;
    for (int n=0; n<N; ++n) priemety.push_back( {10047*X[n]-Y[n], n} );
    sort( priemety.begin(), priemety.end() );

    vector<int> poradie(N), kde_v_poradi(N);
    for (int n=0; n<N; ++n) { poradie[n] = priemety[n].second; kde_v_poradi[ priemety[n].second ] = n; }

    // postupne odsimulujeme všetky udalosti a zapamätáme si najlepší trojuholník
    int najlepsi_obsah = 1<<30;
    vector<int> riesenie;

    for (auto u : U) {
        // potrebujeme vymenit body s cislami u.a, u.b a najst najlepsi trojuholnik pre ne
        int ia = kde_v_poradi[u.a], ib = kde_v_poradi[u.b];

        swap( poradie[ia], poradie[ib] );
        swap( kde_v_poradi[u.a], kde_v_poradi[u.b] );

        int ic1 = min(ia,ib)-1, ic2 = max(ia,ib)+1;
        if (ic1 >= 0) {
            int o = obsah(X,Y,u.a,u.b,poradie[ic1]);
            if (o < najlepsi_obsah) { najlepsi_obsah = o; riesenie = vector<int>{u.a,u.b,poradie[ic1]}; }
        }
        if (ic2 < N) {
            int o = obsah(X,Y,u.a,u.b,poradie[ic2]);
            if (o < najlepsi_obsah) { najlepsi_obsah = o; riesenie = vector<int>{u.a,u.b,poradie[ic2]}; }
        }
    }

    for (int i : riesenie) cout << X[i] << "␣" << Y[i] << endl;
}
```

A-I-4 Online algoritmy

To, že algoritmus, ktorý dá všetky predmety jednému lupičovi, je 2-kompetitívny, dokážeme ľahko, stačí si poriadne rozmyslieť, čo vlastne hovorí definícia.
Jeden z lupičov dostane vždy aspoň polovicu celkovej ceny. Preto pre ľubovoľný vstup platí, že ak máme veci, ktoré dokopy majú cenu c, tak cena optimálneho riešenia je aspoň $c/2$. No a naše riešenie má vždy cenu presne c. Potom ale vždy platí, že $ALG(vstup) \leq 2 \cdot OPT(vstup)$.

Teraz sa pozrime na tretiu podúlohu. Ako dokázať, že ľubovoľný deterministický algoritmus musí byť aspoň 3/2-kompetitívny?
Predstavme si, že sedíme vo vreci a ideme postupne podávať lupičom veci, ktoré si majú rozdeliť. Najskôr im podáme vec s cenou 1. Je jedno, kto ju dostane. Potom im podáme druhú vec s cenou 1. Ak ju dajú tomu istému ako prvú, tak povieme, že to už je všetko. V takomto prípade máme presne situáciu z podúlohy A – teda ich algoritmus vyrobil dvakrát horšie riešenie ako optimálne.
V opačnom prípade, teda ak lupiči dajú prvú vec jednému a druhú druhému, dáme im ešte tretiu vec, tentokrát s cenou 2. Bez ohľadu na to, komu ju dajú, skončia s riešením, ktorého cena je 3. Optimálne riešenie však malo cenu 2: jeden lupič mal dostať prvé dva predmety a druhý mal potom dostať ten posledný.

Inými slovami, práve sme ukázali, že úplne každý algoritmus buď na vstupe $(1, 1)$ vyrobí 2-krát horšie ako optimálne riešenie, alebo na vstupe $(1, 1, 2)$ vyrobí 1.5-krát horšie ako optimálne riešenie. A teda úplne každý online algoritmus pre túto úlohu je aspoň 3/2-kompetitívny.

Ostáva posledná otázka: Ako by mohol vyzerať algoritmus ktorý naozaj bude 3/2-kompetitívny?
Celkom prirodzený algoritmus pre túto úlohu je algoritmus „vždy daj vec tomu lupičovi, ktorý má doteraz menej“. Ten na prvý pohľad vyzerá, že by mohol veci rozdeliť lepšie. Poďme si spočítať, ako dobrý naozaj bude.
Predstavme si, že sme vyššie popísaným algoritmom rozdali nejakú postupnosť vecí. Zoberme toho lupiča, ktorý dostal v súčte viac (v prípade rovnosti ľubovoľného) a pozrime sa na poslednú vec, ktorú dostal. Tohto lupiča nazveme prvý lupič, dotyčnú vec nazveme vec V a jej cenu si označíme c. Celkovú cenu všetkých ostatných vecí označíme d.
Tesne pred tým, ako prvý lupič dostal vec V, mal menej (alebo rovnako) ako druhý lupič. Preto tesne po tom, ako ju dostal, mal nanajvýš o c viac ako druhý lupič. No a kedže odvtedy už dostal všetky veci druhý lupič, aj na konci má prvý nanajvýš o c viac ako druhý. Inými slovami, prvý má celkovú cenu nanajvýš $c + d/2$ a druhý má celkovú cenu aspoň $d/2$.
Nami zostrojené riešenie má teda cenu nanajvýš $c + d/2$.
Teraz rozoberme dva prípady. Ak $d \leq c$, optimálne riešenie má cenu c (jeden lupič dostane vec V a druhý všetky ostatné), zatiaľ čo naše riešenie má cenu nanajvýš $d/2 + c \leq c/2 + c = (3/2)c$.
No a pre $d > c$ platí, že optimálne riešenie má cenu väčšiu alebo rovnú ako polovica vecí, čiže aspoň $(c + d)/2$. Pomer ceny nášho a optimálneho riešenia je teda nanajvýš rovný $(c + d/2)/((c + d)/2) = (2c + d)/(c + d) = 1 + c/(c + d) < 1 + c/(2c) = 3/2$.
A tým sme dôkaz skončili – práve sme ukázali, že v oboch prípadoch je nami nájdené riešenie nanajvýš 3/2-krát horšie ako optimálne.

TRIDSIATY ŠTVRTÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2018