A-II-1 Rušenie metra

Z riešenia úloh domáceho kola vyplýva, že každý vrchol v môže byť v nejakých vyhovujúcich postupnostiach odstránený posledný. Skúsime teda najprv zistiť počet vyhovujúcich poradí odstránení, ktoré končia konkrétnym vrcholom v. Keď si náš graf zakoreníme za tento vrchol, zistíme, že v každom momente môžeme odstrániť len jeden z listov stromu. Ak by sme odstránili niečo iné, tak by sa nám graf rozpadol na viac komponentov (alebo by sme nezachovali, že v je odstránený ako posledný). To ale znamená, že ak chceme odstrániť vrchol u, tak najprv musíme odstrániť všetky vrcholy, ktoré sú v podstrome s koreňom u.

Označme si $p_v(u)$ počet vyhovujúcich poradí odstaňovania vrcholov z podstromu s koreňom u pri zakorenení za v.

Ako spočítame $p_v(u)$? Pokúsime se o to rekurzívnym algoritmom. Nech u je ľubovoľný vrchol stromu a $u_1, \ldots, u_k$ sú jeho synovia. Nech n_i pre $i = 1, \ldots, k$ označuje počet vrcholov v podstrome zakorenenom v u_i a $n = n_1 + \ldots + n_k$. Odstránenie podstromu pod u prebieha tak, že najprv odstránime všetky podstromy s koreňmi v synoch vrcholu u a úplne nakoniec odstránime u. Pritom poradie odstraňovania v jednotlivých podstromoch synou do seba môžeme ľubovoľne miešať. Voľbu poradia odoberania vrcholov si teda môžeme rozdeliť na dve nazávislé časti. Najprv si pre každé $i = 1, \ldots, n$ vyberieme, z ktorého podstromu budeme odoberať v i-tom kroku. Potom pre každý podstrom zvlášť určíme, v akom poradí budeme oboberať jeho vrcholy.

Pre prvú časť musíme zistiť počet všetkých rôznych postupností čísel od 1 do k takých, že každé číslo $i \in \{1, \ldots, k\}$ sa v postupnosti vyskytuje n_i-krát. Najprv si teda z n možných pozícií vyberieme n_1 tých, na ktoré dáme číslo 1 (to ide $\binom{n}{n_1}$ spôsobmi). Zo zvyšných $n - n_1$ pozícií vyberieme n_2 tých, na ktoré dáme číslo 2 ($\binom{n-n_1}{n_2}$). A tak ďalej. Nakoniec nám to stačí vynásobiť a dostaneme:

$$\binom{n}{n_1}\binom{n-n_1}{n_2}\cdots\binom{n-\ldots}{n_k} = \frac{n!}{n_1! \cdot n_2! \cdot \ldots \cdot n_k!}.$$

Pre druhú časť potrebujeme pre každý podstrom určiť počet poradí odoberaní jeho vrcholov. To ale vieme zistiť rekurzívne – sú to hodnoty $p_v(u_1), \ldots, p_v(u_k)$. Nakoľko prvá a druhá časť sú na sebe nezávislé, môžeme ich vynásobiť.

$$p_v(u) = \frac{n! \cdot p_v(u_1) \cdot \ldots \cdot p_v(u_k)}{n_1! \cdot \ldots \cdot n_k!}.$$

Takto ľahko spočítame $p_v(v)$. Uvedená rekurzia pre výpočet $p_v(v)$ navštívi každý vrchol práve raz a zaberie lineárny čas.

My ale musíme prejsť cez všetky možné zakorenenia (t.j. všetky možnosti, ktorý vrchol odobrať ako posledný) a tieto čísla sčítať. Nakoľko je ale zakorenení $O(n)$, tak výsledná časová zložitosť tohto algoritmu bude $O(n^2)$.

Ako to zrýchliť? Zafixujeme si až do konca výpočtu jediný pevný koreň v a spustíme naň vyššie uvedený algoritmus, takže spočítame $p_v(u)$ pre každý vrchol u.

Máme spočítaný počet poradí končiacich vrcholom v. Ako teraz spočítať tieto počty pre všetky ostatné vrcholy? Budeme postupovať po vrstvách stromu (vrstvu tvoria vrcholy s rovnakou hĺbkou), začínajúc v samotnom koreni. Prvá úroveň je vyriešená. Je tam len vrchol v a $p_v(v)$ sme už zrátali. Teraz by sme chceli z výsledkov pre k-tu úroveň zrátať výsledky pre nejaký vrchol u z $(k+1)$-ej úrovne. Čo by sa zmenilo, keby sme teraz prehlásili u za koreň?

Pôvodní synovia u zostali, kde boli a zjavne platí $p_v(u_i) = p_u(u_i)$ (ich podstromy sa totiž nezmenili). Zároveň vrcholu u pribudol jeden ďalší syn, a to jeho bývalý otec w. Ten bol v k-tej úrovni, takže preň už máme porátané $p_w(w)$. Aby sme mohli spočítať $p_u(u)$, potrebujeme spočítať $p_u(w)$. Všimnime si, že $p_u(w)$ dostaneme tak, že pri výpočte $p_w(w)$ ignorujeme podstrom pod u. Ak si označíme n_u ako počet vrcholov v podstrome s koreňom u pri zakorenení v w, dostávame

$$p_u(w) = p_w(w) \cdot \frac{(n-1-n_u)! \cdot n_u!}{(n-1)! \cdot p_w(u)}$$

a $p_u(u)$ už dopočítame ako v predchádzajúcom postupe.

V prvnej fáze tohoto riešenia sme spočítali $p_v(u)$ pre pevné v a všetky vrcholy, to zabralo čas $O(n)$. V druhej fáze sme prešli strom po úrovniach a urobili konštantne veľa výpočtov v každom vrchole, takže táto fáza tiež zabrala $O(v)$, čím sme získali lineárne riešenie.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
using namespace std;

struct vrchol {
  int cislo_vrcholu;
  vector<int> susedia;
  int velkost_podstromu;
  int p_koren_v;  // Počet spôsobov odoberania vrcholov z podstromu zakorenenom vo v.
  int p_v_v;      // Počet spôsobov odoberania takých, že v je posledný.

  int urci_velkost_podstromu(int z);
  int urci_p_koren_v(int z);
  void urci_p_v_v(int w);
};

static int N;
static vector<vrchol> graf;
static vector<int> fact;

// Rekurzívny prechod stromom, pri ktorom určujeme veľkosť podstromu.
// Do aktuálneho vrcholu sme prišli z vrchola Z.
int vrchol::urci_velkost_podstromu(int z) {
  vector<int>::iterator s;
  velkost_podstromu = 1;

  for (s = susedia.begin(); s != susedia.end(); s++) {
    if (*s == z) continue;
    velkost_podstromu += graf[*s].urci_velkost_podstromu(cislo_vrcholu);
  }
  return velkost_podstromu;
}

// Rekurzívny prechod stromom, pri ktorom určujeme počet spôsobov odoberania
// vrcholov z aktuálneho podstromu. Do aktuálneho vrcholu sme prišli z vrcholu Z.
int vrchol::urci_p_koren_v(int z) {
  vector<int>::iterator s;
  p_koren_v = fact[velkost_podstromu - 1];

  for (s = susedia.begin(); s != susedia.end(); s++) {
    if (*s == z) continue;
    p_koren_v /= fact[graf[*s].velkost_podstromu];
    p_koren_v *= graf[*s].urci_p_koren_v(cislo_vrcholu);
  }
  return p_koren_v;
}

// Rekurzívny prechod stromom, pri ktorom určujeme počet spôsobov odoberania
// vrcholov v ktorých aktuálny vrchol bol odobraný posledný. Otec aktuálneho vrcholu
// je vrchol W.
void vrchol::urci_p_v_v(int w)
{
  vector<int>::iterator s;
  if (w == -1)
    p_v_v = p_koren_v;
  else {
    p_v_v = graf[w].p_v_v * fact[velkost_podstromu];
    for (s = susedia.begin(); s != susedia.end(); s++) {
      if (*s == w) continue;
      p_v_v /= fact[graf[*s].velkost_podstromu];
      p_v_v *= graf[*s].p_koren_v;
    }
    p_v_v /= p_koren_v * (N - velkost_podstromu);
  }

  for (s = susedia.begin(); s != susedia.end(); s++) {
    if (*s == w) continue;
    graf[*s].urci_p_v_v(cislo_vrcholu);
  }
}

int main() {
  int i, f, res;

  scanf("%d\n", &N);
  graf.resize(N);
  for (i = 0; i < N; i++) graf[i].cislo_vrcholu = i;

  // Predpočítame si faktoriál.
  fact.push_back(1);
  fact.push_back(1);
  for (i = 2, f = 2; i < N; i++, f *= i) fact.push_back(f);

  // Načítame vstup.
  for (i = 0; i < N-1; i++) {
    int u, v;
```

```
    scanf("%d%d", &u, &v);
    graf[u - 1].susedia.push_back(v - 1);
    graf[v - 1].susedia.push_back(u - 1);
  }

  // Rekurzívne prechody stromom.
  graf[0].urci_velkost_podstromu(-1);
  graf[0].urci_p_koren_v(-1);
  graf[0].urci_p_v_v(-1);

  for (i = 0, res = 0; i < N; i++) res += graf[i].p_v_v;
  printf("%d\n", res);
}
```

A-II-2 Čert-prezident

Najdôležitejším pre úspešné vyriešenie úlohy bolo pozorovanie, že po každej otázke `boji_sa(a,b)` vieme o jednom z čertov a a b povedať, že určite nemôže byť čertom-prezidentom. Pokiaľ sa čert a bojí čerta b, nemôže byť prezidentom, ktorý sa nikoho nebojí. Podobne, ak sa čert a nebojí čerta b, čert b nemôže byť prezidentom, lebo prezidenta sa musia báť všetci ostatní.
A tu sa hneď ponúka prvé riešenie. V prvej fáze riešenia si budeme udržiavať množinu kandidátov na čerta-prezidenta. Kým ich máme viac ako jedného, tak vždy dvoch kandidátov zoberieme a opýtame sa na nich (v ľubovoľnom poradí). Tým vždy znížime počet kandidátov o jedna. Po $n-1$ otázkach nám ostane už len jediný kandidát. V druhej fáze riešenia ešte potrebujeme overiť, či tento kandidát naozaj spĺňa obe podmienky. Na to nám určite stačí ďalších $2n-2$ otázok, dokopy teda $3n-3$.

Tu úloha ešte nekončí, lebo toto číslo potrebujeme čo najviac znížiť. Kde môžeme ušetriť? Počas overovania kandidáta sa nemusíme znova pýtať otázky, ktoré sme sa už spýtali počas jeho hľadania. Taká otázka je určite aspoň jedna – v poslednej otázke počas hľadania kandidáta sme sa určite pýtali na neho a nejakého iného čerta. Ak túto otázku v druhej fáze preskočíme, znížime tak počet potrebných otázok na $3n-4$.
Dá sa však na to ísť ešte lepšie. Na to, aby sme v druhej fáze ušetrili čo najviac otázok, potrebujeme zabezpečiť, aby sme sa o našom kandidátovi v prvej fáze opýtali čo najviac otázok.
Presnejšie, cieľom je, aby sme sa aj pre najhorší prípad výsledkov funkcie `boji_sa` mali čo najviac informácie o našom poslednom kandidátovi. Vhodným spôsobom je pýtať sa otázky vždy o dvoch kandidátoch, o ktorých sme sa pýtali najmenej otázok (čím si zaručíme, že nikdy nestratíme kandidáta, o ktorom sme toho zistili veľa). Jedným ľahko vizualizovateľným postupom je zorganizovať im turnaj – „pavúka“, v ktorom vždy postavíme do „súboja“ dvoch čertov, o ktorých sme sa pýtali zhruba rovnako otázok.
Takýmto postupom si zaručíme, že nech sa deje čokoľvek, nášho výsledného kandidáta sa spýtame aspoň $\lfloor \log_2 n \rfloor$ otázok. Tým sa dostávame na vzorový počet spýtaných otázok, $3n - 3 - \lfloor \log_2 n \rfloor$

Listing programu (C++)

```
#include <cstdlib>
#include <cstdio>
#include <vector>

using namespace std;

int boji_sa(int a, int b)
{ ... }

typedef struct { int a, b; } otazka;

/*
 *  Overí, či kandidát c spĺňa všetky požiadavky na čerta-prezidenta
 *  položením všetkých nepoložených otázok o ňom.
 */
static bool over_kandidata(int c, int n, vector<otazka>& otazky) {
    vector <bool> c_ako_a(n, false);
    vector <bool> c_ako_b(n, false);

    for (int i = 0; i < n - 1; i++) {
        if (otazky[i].a == c) c_ako_a[otazky[i].b - 1] = true;
        if (otazky[i].b == c) c_ako_b[otazky[i].a - 1] = true;
    }
```

```
    for (int i = 1; i <= n; i++) {
        if (i == c) continue;

        if (!c_ako_a[i - 1] && boji_sa(c, i)) return false;
        if (!c_ako_b[i - 1] && !boji_sa(i, c)) return false;
    }
    return true;
}

/*
 *  Porovnáva čertov turnajovým systémom, až kým neostane jediný
 *  kandidát na čerta-prezidenta. Kladené otázky ukladá do poľa
 *  otazky.
 */
static int najdi_kandidata(int n, vector <otazka>& otazky) {
    vector <int> s(2*n - 1, -1);
    int i, a, b, z, k;

    for (i = 0; i < n; i++) s[i] = i + 1;
    z = 0;
    k = n;

    for (i = 0; i < n - 1; i++) {
        a = s[z];
        b = s[z + 1];
        z += 2;

        otazky[i].a = a;
        otazky[i].b = b;

        if (boji_sa(a, b)) s[k] = b;
        else s[k] = a;

        k++;
    }

    return s[z];
}

int prezident(int n) {
    vector<otazka> otazky(n - 1);
    int c = najdi_kandidata(n, otazky);
    return over_kandidata(c, n, otazky) ? c : 0;
}
```

Na záver si skúste rozmyslieť, prečo je vyššie uvedené riešenie optimálne. (Návod: predstavte si, že odpovede na otázky `boji_sa(a,b)` nie sú vopred dané, ale vyberá si ich počas behu vášho programu zákerný nepriateľ. Zákerný nepriateľ vie sledovať, či ste ešte v prvej fáze, a dávať vám odpovede tak, aby ste vždy z množiny kandidátov vylúčili toho čerta, ktorý už bol použitý vo viac otázkach.)

A-II-3 Okružná jazda

Najprv si rozmyslime, ako by sme okružnú trasu pre smetiarov hľadali, keby všetky ulice boli jednosmerné. Aby úloha mala riešenie, je samozrejme nutné, aby sieť ulíc bola (silno) súvislá – teda musíme sa vedieť dostať z ľubovoľnej križovatky na ľubovoľnú inú.[1] Tak isto je nutné, aby z každej križovatky von vychádzalo rovnako veľa ulíc, ako do nej vchádza. Totiž vždy keď do nejakej križovatky vôjdeme, musíme z nej aj vyjsť. (Toto platí aj pre križovatku 1. Odtiaľ navyše raz na začiatku vyjdeme, ale zase sa tam musíme aj na konci vrátiť.)
Ukážeme, že tieto podmienky sú postačujúce pre existenciu riešenia.
Vyrazme z križovatky číslo 1 a vždy poďme ľubovoľnou ešte nepoužitou ulicou v povolenom smere, pokiaľ je to možné. Kde takáto prechádzka môže skončiť? Keď prídeme na križovatku k odlišnú od križovatky 1, tak sme do križovatky k vstupili o jedna viackrát, než sme z nej odišli. Ale z každej križovatky vedie von rovnako veľa ulíc ako dnu, takže určite ešte máme ulicu, ktorou môžeme odísť.
Preto naša prechádzka musí skončiť na križovatke 1. Uvažujme teraz nejakú takú prechádzku $P = 1, k_1, k_2, \ldots, k_t, 1$. Ak sme v nej neprešli ňusetky ulice, tak vďaka súvislosti siete existuje nejaká križovatka k_i v prechádzke, z ktorej vedú ešte nepoužité ulice. Teraz zopakujme úvahu z predchádzajúceho odseku, s tým rozdielom, že začínať prechádzku budeme v križovatke k_i a počas nej budeme používať len doteraz nepoužité ulice. Opäť takto dostaneme nejakú okružnú prechádzku, ktorá nutne aj skončí v k_i
Pôvodnú prechádzku P teraz vieme predĺžiť: najprv prejdeme začiatok $1, k_1 \ \ldots, k_i$ z prechádzky P, potom celú

[1] S výnimkou križovatiek, na ktoré nevedie žiadna ulica, teda izolovaných vrcholov v grafe.

novú okružnú prechádzku z k_i naspäť do k_i, a na záver zvyšok prechádzky P, teda $k_{i+1}, \ldots, k_t, 1$.
Tento postup opakujeme a prechádzku predĺžujeme, až pokým v nej nevyskytujú všetky ulice. Keďže v každej iterácii pridáme do prechádzky aspoň nejaké ulice, je tento postup zjavne konečný.
Popisaný algoritmus sa dokonca dá implementovať v čase lineárnom od veľkosti vstupu.[2] Pri prvom prechode si čísla križovatiek ukladáme na zásobník. Potom ich odoberáme a vypisujeme od konca, pokiaľ nedojdeme ku križovatke k_i, z ktorej vedie ešte nepoužitá ulica. Z nej opäť prechádzame a pridávame križovatky na zásobník. Toto opakujeme, pokiaľ sa zásobník nevyprázdni. Takto sme našli prechádzku, ktorá používa každú ulicu práve raz, ale vypisali sme jej ulice v opačnom poradí. To môžeme opraviť ukladaním do pomocného zoznamu, ktorý nakoniec otočíme, alebo priamo drobným trikom: ulice povolíme prechádzať *iba* v protismere.

Zostáva doriešiť obojsmerne použiteľné ulice. To, že chceme každou ísť práve raz, je ekvivalentné s tým, že chceme najskôr z každej obojsmernej ulice vyrobiť jednosmerku a potom chceme vyriešiť vyššie popísanú jednoduchšiu úlohu. A my už navyše presne vieme, čo musí platiť, aby mesto tvorené samými jednosmerkami malo riešenie.
Zjavne je teda potrebné všetky obojsmerné ulice zjednosmerniť tak, aby do každej križovatky vchádzal rovnaký počet ulíc, ako z nej vychádza. Najprv ukážeme niekoľko jednoduchých pozorovaní. Ak sa na nejakej križovatke stretáva nepárny počet ulíc, tak úloha nemá riešenie. Takisto riešenie nebude existovať, ak pre nejakú križovatku k rozdiel medzi počtom vchádzajúcich a vychádzajúcich ulíc je väčší, než počet obojsmerných ulíc, susediacich s k. (Ak napríklad máme križovatku do ktorej vchádza šesť jednosmeriek a vychádzajú len dve, vôbec nám nepomôže, že z tej križovatky vedú aj dve obojsmerné cesty. Aj keby sme obe orientovali smerom von, stále bude von idúcich jednosmeriek primálo.)
Ak je rozdiel medzi počtom vchádzajúcich a vychádzajúcich jednosmeriek presne rovný počtu obojsmerných ulíc, tak máme pre obojsmerné ulice jednoznačne určenú ich orientáciu. (Všimnite si, že buď vyššie popísaná alebo táto situácia určite nastane na každej križovatke, na ktorú vedie práve jedna obojsmerná ulica.)
Takýmto spôsobom postupne priradzujeme orientáciu uliciam, pre ktoré je jednoznačne určená. Ak počas toho stretneme nejaký spor, prestaneme a podáme správu že riešenie neexistuje. (Spor môže byť buď vyššie popísaného typu, alebo ho môžeme dostať tak, že nám pre obojsmernú ulicu medzi križovatkami a a b v križovatke a vyjde opačná orientácia ako v križovatke b.)
Vyššie popísaný postup vieme tiež implementovať v lineárnom čase. Stačí na začiatku raz skontrolovať každú križovatku, a potom ešte vždy keď v križovatke a nastavím orientáciu ulici vedúcej do b, zaradím si križovatku b do zoznamu na ďalšie spracovanie. Každú križovatku takto spracujem nanajvýš dvakrát.

Predpokladajme teraz, že vyššie popísaný postup skončil a už pre žiadnu ulicu nemáme vynútenú jej orientáciu. Pozor, toto ešte neznamená, že už nemáme žiadne obojsmerné ulice! Ešte ostáva jeden možný prípad. Z niektorých križovatiek môžu ešte stále viesť dve obojsmerné ulice. Zároveň pre každú takúto križovatku platí, že počet na ňu vchádzajúcich jednosmeriek sa rovná počtu z nej vychádzajúcich.
Keď sa teraz pozrieme len na obojsmerné ulice, dostaneme graf, v ktorom má každý vrchol stupeň 0 alebo 2. Tento graf sa teda skladá z jedného alebo viacerých disjunktných cyklov. No a na doriešenie našej úlohy stačí v rámci každého cyklu orientovať všetky hrany „tým istým smerom“ – teda tak, aby sa po cykle dalo v smere hrán prejsť dokola.
Celý algoritmus je teda možné realizovať s lineárnou časovou aj pamäťovou zložitosťou.

Listing programu (C++)

```
#include <cstdio>
#include <cstdlib>
#include <vector>
using namespace std;

struct hrana
{
  int z, d;
  hrana(int _z, int _d) { z = _z; d = _d; }
};

struct krizovatka
{
  vector<hrana *> dovnitr, ven; // Jednosměrné ulice do a z křižovatky.
```

[2] Teda v čase $O(n + m)$, kde n je počet križovatiek a m počet ulíc.

```
  vector<int> obousmerne; // Obousměrné ulice sousedící s křižovatkou.
  int prvni_nepouzita;    // První ulice do křižovatky, která ještě není v procházce.
  bool navstivena;        // Značka pro prohledávání do hloubky.

  krizovatka()
    {
      prvni_nepouzita = 0;
      navstivena = false;
    }
};

static krizovatka *graf;

// Průchodem do hloubky určí počet ulic dosažitelných z křižovatky Z
// bez ohledu na jejich směr (každá ulice je počítána dvakrát, jednou
// za každý její konec).

static int pocet_dostupnych(int z)
{
  vector<hrana *>::iterator i;
  vector<int>::iterator j;
  int pocet = 0;

  if (graf[z].navstivena)
    return 0;
  graf[z].navstivena = true;

  vector<int> sousedi;
  for (i = graf[z].dovnitr.begin(); i != graf[z].dovnitr.end(); ++i)
    sousedi.push_back((*i)->z);
  for (i = graf[z].ven.begin(); i != graf[z].ven.end(); ++i)
    sousedi.push_back((*i)->d);
  for (j = graf[z].obousmerne.begin(); j != graf[z].obousmerne.end(); ++j)
    sousedi.push_back(*j);

  for (j = sousedi.begin(); j != sousedi.end(); j++)
    pocet += 1 + pocet_dostupnych(*j);

  return pocet;
}

// Přidá do grafu jednosměrnou hranu z křižovatky F do křižovatky T.

static void pridej_jednosmernou_hranu(int f, int t)
{
  hrana *h = new hrana(f, t);
  graf[f].ven.push_back(h);
  graf[t].dovnitr.push_back(h);
}

// Smaže z grafu obousměrnou hranu mezi křižovatkami F a T.

static void smaz_obousmernou_hranu(int f, int t)
{
  vector<int>::iterator i;

  for (i = graf[f].obousmerne.begin(); i != graf[f].obousmerne.end(); ++i)
    if (*i == t)
      break;

  *i = graf[f].obousmerne.back();
  graf[f].obousmerne.pop_back();
}

// Prochází obousměrné ulice a orientuje ty, u nichž je orientace vynucená.
// Vrátí false, pokud u některé křižovatky nelze naorientovat ulice tak,
// aby jich vcházel a vycházel stejný počet.

static bool naorientuj_vynucene(int n)
{
  vector<int> kontroluj;

  for (int i = 1; i <= n; i++)
    kontroluj.push_back(i);

  while (!kontroluj.empty())
    {
      int v = kontroluj.back();
      kontroluj.pop_back();

      int rozdil = graf[v].ven.size() - graf[v].dovnitr.size();
      if (rozdil == 0)
        continue;

      if ((unsigned) abs(rozdil) > graf[v].obousmerne.size())
        return false;

      bool ven;
```

```
      if (rozdil == (int) graf[v].obousmerne.size())
        ven = false;
      else if (-rozdil == (int) graf[v].obousmerne.size())
        ven = true;
      else
        continue;

      vector<int>::iterator a;
      for (a = graf[v].obousmerne.begin(); a != graf[v].obousmerne.end(); ++a)
        {
          smaz_obousmernou_hranu(*a, v);
          if (ven)
            pridej_jednosmernou_hranu(v, *a);
          else
            pridej_jednosmernou_hranu(*a, v);
          kontroluj.push_back(*a);
        }
      graf[v].obousmerne.clear();
    }

  return true;
}

// Naorientuje libovolně cyklus z obousměrných ulic obsahující křižovatku V.

static void naorientuj_cyklus(int v)
{
  vector<int> cyklus;
  vector<int>::iterator i, j;

  cyklus.push_back(v);
  int a = graf[v].obousmerne[0], p = v;
  while (a != v)
    {
      i = graf[a].obousmerne.begin();
      cyklus.push_back(a);

      if (p == *i)
        ++i;

      p = a;
      a = *i;
    }
  cyklus.push_back(v);

  for (i = cyklus.begin(), j = i + 1; j != cyklus.end(); ++i, ++j)
    {
      pridej_jednosmernou_hranu(*i, *j);
      graf[*i].obousmerne.clear();
    }
}

// Naorientuje libovolně cykly z obousměrných ulic.

static void naorientuj_cykly(int n)
{
  for (int v = 1; v <= n; v++)
    if (!graf[v].obousmerne.empty())
      naorientuj_cyklus(v);
}

// Nalezne a vypíše procházku, která projde každou ulicí právě jednou
// a v povoleném směru, za předpokladu, že žádná ulice není obousměrná.

static void vypis_eulerovsky_tah()
{
  vector<int> tah;
  const char *sep = "";

  tah.push_back(1);
  while (!tah.empty())
    {
      int v = tah.back();

      if ((unsigned) graf[v].prvni_nepouzita == graf[v].dovnitr.size())
        {
          printf("%s%d", sep, v);
          sep = "␣";
          tah.pop_back();
        }
      else
        {
          hrana *h = graf[v].dovnitr[graf[v].prvni_nepouzita];
          tah.push_back(h->z);
          graf[v].prvni_nepouzita++;
        }
    }
  printf("\n");
```

```
}

int main()
{
  int m, n;

  scanf("%d%d", &n, &m);
  graf = new krizovatka[n + 1];

  for (int i = 0; i < m; i++)
    {
      int f, t;
      char sm[2];

      scanf("%d%d%s", &f, &t, sm);
      if (sm[0] == 'J')
        pridej_jednosmernou_hranu(f, t);
      else
        {
          graf[f].obousmerne.push_back(t);
          graf[t].obousmerne.push_back(f);
        }
    }

  for (int i = 0; i < n; i++)
    if ((graf[i].dovnitr.size() + graf[i].ven.size() + graf[i].obousmerne.size())
        % 2 == 1)
      {
        printf("nelze\n");
        return 0;
      }

  if (pocet_dostupnych(1) != 2*m || !naorientuj_vynucene(n))
    {
      printf("nelze\n");
      return 0;
    }

  naorientuj_cykly(n);
  vypis_eulerovsky_tah();
  return 0;
}
```

A-II-4 Magické siete

V prvej podúlohe sme mali ukázať, že neexistuje žiadna sieť, ktorá simuluje XOR_4 a pritom sa skladá zo samých obmedzení typu XOR_3.

Toto je veľmi jednoduché. Dokážeme to sporom. Predstavme si, že máme takú sieť. Naša sieť má teda 4 vstupné premenné (w, x, y, z), niekoľko pomocných premenných, a niekoľko obmedzení typu XOR_3. Dosaďme teraz za všetky premenné hodnotu 1. Keďže každé obmedzenie je typu XOR_3, sú všetky obmedzenia splnené. Sieť teda nutne prijme vstup $(1, 1, 1, 1)$. A to je hľadaný spor, lebo obmedzenie XOR_4 pre vstup $(1, 1, 1, 1)$ nie je splnené.

V druhej podúlohe sme mali ukázať, že keď si ku XOR_3 zoberieme na pomoc aj ZERO, už budeme vedieť simulovať XOR_4.

V nasledujúcom texte použijeme označenie $\overline{x}$ ako negáciu x a označenie $\oplus$ ako operáciu xor. (Platí teda napr. $0 \oplus 1 = 1$ a $\overline{0 \oplus 1} = 0$. Zjavne tiež $\overline{x} = x \oplus 1$.)

Zaveďme si pomocnú premennú a a všimnime si, čo spraví obmedzenie $\mathrm{XOR}_3(w, x, a)$. Ak $w = x$ tak a musí byť 1 a naopak. Teda pri ľubovoľnom prípustnom ohodnotení premenných platí $a = \overline{w \oplus x} = w \oplus x \oplus 1$. Môžeme sa teda na to celé dívať tak, že sme „vypočítali“ hodnotu $w \oplus x$ a jej negáciu sme si uložili do a.
Teraz pridajme druhé obmedzenie: $\mathrm{XOR}_3(a, y, b)$. Keďže y je ďalšia vstupná premenná a hodnota a je jednoznačne určená vstupnými premennými w a x, môžeme sa aj na toto obmedzenie dívať tak, že nám „vypočíta“ hodnotu b. Presnejšie, opäť bude platiť $b = \overline{a \oplus y} = a \oplus y \oplus 1$. Po dosadení za a dostávame, že $b = w \oplus x \oplus y$. Teda b bude určite rovné xoru prvých troch vstupných premenných.
No a už sme skoro hotoví. Teraz pridáme tretie obmedzenie: $\mathrm{XOR}_3(b, z, c)$. Podobne ako vyššie, toto si vynúti že c bude negáciou xoru všetkých štyroch vstupných premenných. A keďže my chceme, aby xor všetkých šty-

roch vstupných premenných bol 1, tak vlastne chceme aby c bolo 0. A toto si vynútime posledným, štvrtým obmedzením: ZERO(c).
Tu je teda ešte raz kompletné riešenie: obmedzenie $\mathrm{XOR}_4(w, x, y, z)$ je ekvivalentné so sieťou tvorenou obmedzeniami $\mathrm{XOR}_3(w, x, a)$, $\mathrm{XOR}_3(a, y, b)$, $\mathrm{XOR}_3(b, z, c)$ a ZERO(c).

Iné riešenie: Pomocou ZERO($\heartsuit$) si vynútime, že pomocná premenná $\heartsuit$ musí mať hodnotu 0. Teraz pomocou nej vieme negovať. Ak napríklad pridáme obmedzenie $\mathrm{XOR}_3(u, v, \heartsuit)$ tak v musí mať hodnotu $\overline{u}$.
No a teraz môžeme postupne pridávať nasledovné obmedzenia:
$\mathrm{XOR}_3(w, x, a)$ – a je negáciou $w \oplus x$
$\mathrm{XOR}_3(a, b, \heartsuit)$ – b je $w \oplus x$
$\mathrm{XOR}_3(b, y, c)$ – c je negáciou $w \oplus x \oplus y$
$\mathrm{XOR}_3(c, d, \heartsuit)$ – d je $w \oplus x \oplus y$
$\mathrm{XOR}_3(d, z, e)$ – e je negáciou $w \oplus x \oplus y \oplus z$
a na záver to už len, podobne ako v predchádzajúcom riešení, ukončíme obmedzením ZERO(z).

V poslednej podúlohe sme mali obmedzenie XOR_4 simulovať pomocou siete obmedzení typu NAE.

Máme teda opäť štyri vstupné premenné (w, x, y, z) a potrebujeme si vynútiť, aby buď práve jedna z nich, alebo práve tri z nich mali hodnotu 1.
Inými slovami, potrebujeme zakázať kombinácie hodnôt $(0, 0, 0, 0)$, $(1, 1, 1, 1)$ a všetky permutácie $(0, 0, 1, 1)$.
Všimnime si obmedzenie NAE(w, x, a). Ak sú w a x rôzne, a môže mať ľubovoľnú hodnotu. Ak sú ale w a x rovnaké, a musí mať hodnotu opačnú.
Čo sa teraz stane, ak zoberieme naraz obmedzenia NAE(w, x, a) aj NAE(y, z, a)? Tieto dve obmedzenia skoro vždy vieme obe splniť vhodnou voľbou a. Existuje v princípe len jeden prípad, kedy tomu tak nie je: ak si aj jedno aj druhé obmedzenie vynúti konkrétnu hodnotu a, ale nie oba tú istú. Teda existujú len dve ohodnotenia premenných (w, x, y, z) ktoré nevyhovujú: $(0, 0, 1, 1)$ a $(1, 1, 0, 0)$.
Zvyšné kombinácie dvoch 0 a dvoch 1 zakážeme podobne, vhodnou zámenou poradia vstupných premenných. Pridali by sme teda dvojicu obmedzení NAE(w, y, b) a NAE(x, z, b) a ďalšiu dvojicu obmedzení NAE(w, z, c) a NAE(x, y, c).
Vyššie popísaných 6 obmedzení teda úspešne zakázalo všetky situácie, v ktorých práve dve zo vstupných premenných majú hodnotu 1. Ostáva nám ešte zakázať vstupy $(0, 0, 0, 0)$ a $(1, 1, 1, 1)$.
Toto spravíme veľmi podobne. Všimnime si obmedzenia NAE(w, x, d) a NAE(y, z, e). Nám sa nepáčia práve tie situácie, v ktorých má aj d aj e vynútenú hodnotu, a to obe tú istú. Tak práve túto skutočnosť zakážeme pridaním obmedzenia NAE(d, d, e). Ľahko nahliadneme, že vyhovujúce d a e vieme zvoliť práve vtedy, ak nemajú premenné w, x, y, z všetky tú istú hodnotu.

TRIDSIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Forišek, Askar Gafurov, Vladimír Macko, Jaroslav Petrucha
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2015