P-II-1 Rušení stanic podruhé

Z řešení úlohy domácího kola plyne, že libovolný vrchol může být ten poslední, který odstraňujeme. Je tedy přirozené si vybrat jeden vrchol v, který si necháme nakonec, a v něm strom zakořenit. Dále si všimněme, že v každém kroku odstraňujeme list, protože odstraněním vrcholu, který není listem, by se strom rozpadl na více komponent souvislosti. Proto pro každý podstrom s kořenem u musíme odstranit jako poslední z tohoto podstromu právě vrchol u.

Označme $p_v(u)$ počet pořadí vrcholů podstromu s kořenem u vzhledem k zakořenění ve v, ve kterých lze vrcholy tohoto podstromu odstraňovat, aniž by byla porušena jeho souvislost a poslední byl odstraněn u. Výsledek úlohy získáme tak, že sečteme hodnoty $p_v(v)$ pro všechny možné volby posledního odebraného vrcholu v.

Jak ale spočítáme $p_v(u)$? Pokusíme se o to rekurzivním algoritmem. Nechť u je libovolný vrchol stromu a $u_1, \ldots, u_k$ jeho synové. Nechť n_i pro $i = 1, \ldots, k$ označuje počet vrcholů v podstromu zakořeněném v u_i a $n = n_1 + \ldots + n_k$. Odstranění podstromu pod u probíhá tak, že nejprve odstraníme všechny podstromy synů u a pak odstraníme u. Přitom pořadí odstraňování v jednotlivých podstromech synů do sebe můžeme libovolně míchat. Volbu pořadí odebírání vrcholů si tedy můžeme rozložit na dvě nezávislé části: Nejprve si pro každé $i = 1, \ldots, n$ vybereme, ze kterého podstromu budeme odebírat i-tý vrchol. Poté pro každý podstrom zvlášť určíme, v jakém pořadí odebereme jeho vrcholy.

Pro první část musíme zjistit počet všech různých posloupností čísel od 1 do k takových, že každé číslo $i \in \{1, \ldots, k\}$ se v posloupnosti vyskytuje n_i-krát. Nejprve si tedy z n možných pozic vybereme n_1 těch, na které dáme číslo 1 (to lze $\binom{n}{n_1}$ způsoby). Ze zbývajících $n - n_1$ pozic vybereme n_2 těch, na které dáme číslo 2 (to lze $\binom{n-n_1}{n_2}$ způsoby). A tak dále. Celkem je tedy počet takových posloupností

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

Pro druhou část potřebujeme pro každý podstrom určit počet pořadí odebrání jeho vrcholů. Tato pořadí jsou ale právě hodnoty $p_v(u_1), \ldots, p_v(u_k)$, které získáme z rekurze. Protože všechny volby jsou na sobě nezávislé, počet všech možností získáme jejich pronásobením a

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

Nakonec takto spočítáme i $p_v(v)$. Uvedená rekurze navštíví každý vrchol právě jednou, a proto celý výpočet $p_v(v)$ zabere linearní čas.

Protože musíme určit $p_v(v)$ pro každý vrchol v, celý algoritmus běží v $\mathcal{O}(N^2)$. Jak to zrychlit? Zafixujeme si až do konce výpočtu jediný pevný kořen v a spustíme na něj výše uvedený algoritmus, takže spočítáme $p_v(u)$ pro každý vrchol u.

Máme spočten počet pořadí končících vrcholem v. My bychom nyní potřebovali zahrnout případy, kdy končíme vrcholem jiným. Budeme postupovat induktivně po hladinách stromu. První hladina je vyřešená, na ní je pouze vrchol v. Předpokládejme, že máme spočítán kompletní výsledek pro všechny vrcholy v k-té hladině a vezměme vrchol u z $(k+1)$-ní hladiny. Nyní prohlásíme u za kořen stromu. Co se změní?

Původní synové u zůstali, kde byli, a pro ně máme spočteno $p_v(u_i) = p_u(u_i)$. Zároveň vrcholu u přibyl jediný další syn, jeho bývalý otec w, který byl z k-té hladiny, takže pro něj již máme spočten celkový počet pořadí končících w, tedy $p_w(w)$. Abychom mohli spočítat $p_u(u)$, potřebujeme už jen spočítat $p_u(w)$. Všimněme si, že $p_u(w)$ získáme tak, že při výpočtu $p_w(w)$ ignorujeme podstrom pod u. Označíme-li n_u počet vrcholů v podstromě s kořenem u při zakořenění ve w, dostáváme

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

a $p_u(u)$ už nyní dopočítáme jako v předchozím postupu.

V první fázi tohoto řešení jsme spočítali $p_v(u)$ pro pevné v a všechny vrcholy, to zabralo $\mathcal{O}(N)$. V druhé fázi jsme prošli strom po hladinách a provedli konstantně mnoho výpočtů v každém vrcholu, takže tato fáze také zabrala $\mathcal{O}(N)$, čímž jsme získali lineární řešení.

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

struct vrchol
{
  int cislo_vrcholu;
  vector<int> sousedi;
  int velikost_podstromu;
  int p_koren_v;  // Počet způsobů odebírání vrcholů z podstromu zakořeněného ve v.
  int p_v_v;      // Počet způsobů odebírání tak, aby v byl poslední.

  int urci_velikost_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;

// Rekurzivní průchod stromem, při němž určujeme velikosti podstromů.
// Do aktuálního vrcholu jsme přišli z vrcholu Z.
int vrchol::urci_velikost_podstromu(int z)
{
  vector<int>::iterator s;
  velikost_podstromu = 1;
```

```
  for (s = sousedi.begin(); s != sousedi.end(); s++)
    {
      if (*s == z)
        continue;

      velikost_podstromu += graf[*s].urci_velikost_podstromu(cislo_vrcholu);
    }

  return velikost_podstromu;
}

// Rekurzivní průchod stromem, při němž určujeme počet způsobů odebírání
// vrcholů z aktuálního podstromu. Do aktuálního vrcholu jsme přišli z vrcholu Z.
int vrchol::urci_p_koren_v(int z)
{
  vector<int>::iterator s;
  p_koren_v = fact[velikost_podstromu - 1];

  for (s = sousedi.begin(); s != sousedi.end(); s++)
    {
      if (*s == z)
        continue;

      p_koren_v /= fact[graf[*s].velikost_podstromu];
      p_koren_v *= graf[*s].urci_p_koren_v(cislo_vrcholu);
    }

  return p_koren_v;
}

// Rekurzivní průchod stromem, při němž určujeme počet způsobů odebírání
// vrcholů tak, aby aktuální vrchol byl poslední. Otec aktuálního 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[velikost_podstromu];

      for (s = sousedi.begin(); s != sousedi.end(); s++)
        {
          if (*s == w)
            continue;
          p_v_v /= fact[graf[*s].velikost_podstromu];
          p_v_v *= graf[*s].p_koren_v;
        }
      p_v_v /= p_koren_v * (N - velikost_podstromu);
    }

  for (s = sousedi.begin(); s != sousedi.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;

  // Předpočítáme faktoriály.
  fact.push_back(1);
  fact.push_back(1);
  for (i = 2, f = 2; i < N; i++, f *= i)
    fact.push_back(f);

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

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

  // Rekurzivní průchody stromem.
  graf[0].urci_velikost_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);

  return 0;
}
```

P-II-2 Ďábel

Povšimněme si, že vrátí-li *bojí_se*(a, b) hodnotu 0, pak čert b není vhodný kandidát na ďábla. Naopak, vrátí-li hodnotu 1, pak čert a není vhodný kandidát na ďábla. Budeme si udržovat seznam S čertů, o kterých jsme ještě neukázali, že se nehodí na ďábla. Dokud S obsahuje alespoň dva čerty, dva z nich (a a b) si vybereme a položíme dotaz *bojí_se*(a, b). Tím právě jednoho z nich vyřadíme ze seznamu. Po $N - 1$ opakováních bude S obsahovat právě jednoho čerta c, jediného možného kandidáta na ďábla.

Zbývá nám ale ještě ověřit, zda se čerta c všichni bojí a on se nikoho nebojí. To zvládneme dalšími $2(N - 1)$ dotazy, celkový počet dotazů je tedy $3N - 3$. Tento postup lze ale ještě vylepšit: je zbytečné se v druhé části dávat dotazy, které jsme už položili v první části. Označme si jako $p(x)$ počet dotazů na čerta x v první části. Celkový počet dotazů bude $3N - 3 - p(c)$.

Potřebujeme tedy první část dotazů rozvrhnout tak, aby $p(c)$ bylo co největší možné. Vhodný způsob, jak toho dosáhnout, je vždy porovnávat čerty v S, na než jsme zatím kladli nejméně dotazů (takže nám nehrozí, že bychom z S vyřadili čerta

s mnoha dotazy). Pro jednoduchost uvažme případ, že $N = 2^k$ pro nějaké přirozené číslo k. Pak v prvních $N/2$ dotazech porovnáváme čerty, na které jsme se ještě neptali. V dalších $N/4$ dotazech porovnáváme čerty, na které jsme se ptali jednou. V dalších $N/8$ dotazech porovnáváme čerty, na které jsme se ptali dvakrát. A tak dále, až v posledním $1 = N/2^k$ dotazu porovnáváme čerty, na které jsme se ptali $(k-1)$-krát. Na konci tedy bude $p(c) = k = \log_2 N$.

V obecnosti obdobně nahlédneme, že $p(c) \geq \lfloor \log_2 N \rfloor$. Celkový počet dotazů tedy bude nejvýše $3N - 3 - \lfloor \log_2 N \rfloor$. O trochu složitější úvahou je možné ukázat, že tento počet je optimální a žádný lepší algoritmus neexistuje.

```
#include <stdlib.h>
#include <stdio.h>

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

typedef struct { int a, b; } dotaz;

/*
 * Ověří, zda C je kandidát na ďábla položením všech dotazů,
 * které na něj ještě nebyly položeny. Předchozích N-1 dotazů
 * je uloženo v poli DOTAZY.
 */
static int over_kandidata(int c, int n, dotaz dotazy[])
{
  int *c_jako_a = calloc(n, sizeof(int));
  int *c_jako_b = calloc(n, sizeof(int));
  int i, ret = 1;

  for (i = 0; i < n - 1; i++)
    {
      if (dotazy[i].a == c)
        c_jako_a[dotazy[i].b - 1] = 1;
      if (dotazy[i].b == c)
        c_jako_b[dotazy[i].a - 1] = 1;
    }

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

      if (!c_jako_a[i - 1] && boji_se(c, i))
        {
          ret = 0;
          goto end;
        }

      if (!c_jako_b[i - 1] && !boji_se(i, c))
        {
          ret = 0;
          goto end;
        }
    }

end:
  free(c_jako_a);
```

```
  free(c_jako_b);
  return ret;
}

/*
 * Porovnává čerty, dokud neurčí jediného možného kandidáta
 * na ďábla. Provedené dotazy ukládá do pole DOTAZY.
 */
static int najdi_kandidata(int n, dotaz dotazy[])
{
  int *s = calloc(2*n - 1, sizeof(int));
  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;

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

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

      k++;
    }

  free(s);
  return s[z];
}

int dabel(int n)
{
  dotaz *dotazy = calloc(n - 1, sizeof(dotaz));
  int c = najdi_kandidata(n, dotazy);
  int ret = over_kandidata(c, n, dotazy) ? c : 0;
  free(dotazy);
  return ret;
}
```

P-II-3 Okružní jízda

Nejprve si rozmysleme, jak bychom úlohu řešili, kdyby všechny ulice byly jednosměrné. Aby úloha měla řešení, je samozřejmě nutné, aby síť ulic byla souvislá. Dále je nutné, aby z každé křižovatky ven vycházelo stejně ulic, jako do ní vchází. Ukažme si, že za těchto podmínek řešení vždy existuje.

Vyražme z křižovatky číslo 1 a vždy jděme libovolnou ještě nepoužitou ulicí v povoleném směru, dokud je to možné. Kde taková procházka skončí? Když přijdeme na křižovatku k různou od křižovatky 1, pak jsme do křižovatky k vstoupili

o jedna vícekrát, než kolikrát jsme z ní odešli. Ale z každé křižovatky vede ven stejně ulic jako dovnitř, takže jistě máme kudy odejít. Proto prochazka musí skončit na křižovatce 1. Uvažujme nějakou takovou procházku $P = 1, k_1, k_2, \ldots, k_t, 1$. Jestliže jsme v ní neprošli všechny ulice, pak díky souvislosti sítě existuje nějaká křižovatka k_i v procházce, z níž vede ještě nepoužitá ulice. Nalezenou procházku tedy můžeme prodloužit: nejprve projdeme začátek $1, k_1, \ldots, k_i$ procházky P. Dále z k_i vyrazíme ulicí nepoužitou v P a dokud je to možné, jdeme v povoleném směru libovolnou ještě nepoužitou ulicí, která se nevyskytuje v procházce P. Snadno nahlédneme, že skončíme opět na křižovatce k_i. Z ní pak pokračujeme dle zbytku procházky P, tedy $k_i, k_{i+1}, \ldots, k_t, 1$.

Tento postup opakujeme a procházku prodlužujeme, dokud se v ní nevyskytují všechny ulice. Popsaný algoritmus lze implementovat v lineárním čase: Při prvním průchodu si čísla křižovatek ukládáme na zásobník. Poté je odebíráme a vypisujeme od konce, dokud nedojdeme ke křižovatce k_i, z níž vede ještě nepoužitá ulice. Z ní opět procházíme a přidáváme křižovatky na zásobník. Toto opakujeme, dokud se zásobník nevyprázdní. Takto jsme nalezli procházku, která používá každou ulici právě jednou, ale vypsali jsme ji v opačném pořadí. To můžeme opravit ukládáním do pomocného seznamu, který nakonec otočíme, nebo přímo drobným trikem: ulice povolíme procházet pouze v protisměru.

Zbývá dořešit obousměrně použitelné ulice. Zjevně je třeba ulice zjednosměrnit tak, aby do každé křižovatky vcházel stejný počet ulic, jako z ní vychází. Nejprve proveďme několik jednoduchých pozorování. Jestliže se na nějaké křižovatce sejde lichý počet ulic, úloha nemá řešení. Stejně tak řešení neexistuje, jestliže je pro nějakou křižovatku k rozdíl mezi počtem vcházejících a vycházejících ulic větší, než počet obousměrných ulic sousedících s k. Je-li tento rozdíl přesně roven počtu obousměrných ulic sousedících s k, pak tyto obousměrné ulice musíme naorientovat všechny stejným směrem tak, aby do křižovatky k vcházel stejný počet ulic, jako z ní vychází. Povšimněme si, že jedna z popsaných situací nutně nastane u každé křižovatky, s níž sousedí právě jedna obousměrná ulice. Takto postupně orientujeme ulice, pro něž je směr vynucen. Každou křižovatku stačí kontrolovat nejvýše tolikrát, kolikrát se u ní naorientuje nějaká ulice, tuto část algoritmu lze tedy realizovat v lineárním čase.

Předpokládejme nyní, že žádná z výše popsaných situací již nenastává. Pak s každou křižovatkou sousedí buď žádná nebo dvě obousměrné ulice, a počet jednosměrných vcházejících ulic je stejný jako počet vycházejících. Obousměrné ulice tedy tvoří několik cyklů, v nichž můžeme orientaci zvolit libovolně, ale stejně pro všechny ulice v cyklu.

Dohromady lze celý algoritmus implementovat s lineární časovou i paměťovou složitostí.

```
#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.
  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;
}
```

P-II-4 Magická síť

Úloha 1: Uvažme libovolnou síť, skládající se pouze z omezení XOR. Přiřadíme-li každé proměnné hodnotu 1, pak všechna omezení jsou splněna. Ale $\text{XOR}_4(w, x, y, z)$ nepřijímá ohodnocení $w = x = y = z = 1$. Proto žádná síť, která se skládá pouze z omezení XOR, nesimuluje XOR_4.

Úloha 2: Příkladem takové sítě je

$$\text{XOR}(w, x, a), \text{XOR}(a, y, b), \text{XOR}(z, b, c), \text{ZERO}(c).$$

Proměnná c má hodnotu 0, díky omezení $\text{XOR}(z, b, c)$ má tedy proměnná b opačnou hodnotu než z. První dvě omezení jsou splnitelná (vhodnou volbou hodnoty proměnné a) právě tehdy, má-li z proměnných w, x, y a b sudý počet hodnotu 1. Dohromady lze tedy všechna omezení splnit právě tehdy, má-li z proměnných w, x, y a z lichý počet hodnotu 1.

Úloha 3: Síť XOR_4 odmítá právě ta ohodnocení, v nichž se vyskytují 0, 2 nebo 4 jedničky. Nejprve si ukažme, jak pomocí NAE zakázat ohodnocení, v nichž se vyskytují právě 2 jedničky. Uvažme síť $\text{NAE}(w, x, a)$, $\text{NAE}(y, z, a)$ se vstupními proměnnými w, x, y a z. Ta vstup přijímá, jestliže první a druhé omezení nevynucují opačná ohodnocení pro proměnnou a. Síť tedy odmítá právě ohodnocení taková, že $w = x = 1$ a $y = z = 0$, nebo $w = x = 0$ a $y = z = 1$. Obdobně prohozením proměnných x, y a z získáme sítě zakazující ohodnocení $w = y = 1$ a $x = z = 0$, nebo $w = y = 0$ a $y = x = 1$, případně $w = z = 1$ a $x = y = 0$, nebo $w = z = 0$ a $x = y = 1$.

Ještě potřebujeme zakázat ohodnocení $w = x = y = z = 0$ a $w = x = y = z = 1$. To dělá síť $\text{NAE}(w, x, d)$, $\text{NAE}(y, z, e)$, $\text{NAE}(d, d, e)$. Síť $\text{XOR}_4(w, x, y, z)$ je tedy simulována sítí

$$\text{NAE}(w, x, a), \text{NAE}(y, z, a), \text{NAE}(w, y, b), \text{NAE}(x, z, b), \text{NAE}(w, z, c),$$
$$\text{NAE}(x, y, c), \text{NAE}(w, x, d), \text{NAE}(y, z, e), \text{NAE}(d, d, e).$$