P-III-1 Básník Honzík

Pokud bychom hledali libovolné slovo s nejdelším společným koncovým úsekem, stačilo by použít datovou strukturu trie (neboli písmenkový strom – to je zakořeněný strom, v jehož kořeni se rozhodujeme podle prvního písmene slova, na další hladině podle druhého, pak podle třetího, atd.). Do trie bychom přidali všechna slova ze seznamu pozpátku a stejně tak požadované slovo bychom hledali od konce. Ve chvíli, kdy se dostaneme do uzlu u, který nemá vhodného následníka, nebo nalezneme vyhledávané slovo celé, známe nejdelší koncový úsek, kterého můžeme dosáhnout (odpovídá písmenům na cestě z kořene trie do u). Tento koncový úsek však může sdílet více slov – taková slova jsou právě ta, která končí v u nebo v jeho podstromě.

Jak ale vybrat lexikograficky nejmenší? Přímočaré řešení by spočívalo v nalezení všech slov, která se nachází v podstromu u, a následném výběru lexikografického minima. Nalezení všech slov se dá udělat průchodem do hloubky, avšak uvědomíme-li si, že počet takových slov může být srovnatelný s velikostí celého seznamu slov, zjistíme, že jsme si moc nepomohli oproti triviálnímu řešení, které vždy projde celý seznam a má složitost $\mathcal{O}(\textit{maximální délka slova} \cdot \textit{velikost seznamu})$.

Pokusme se trii upravit tak, abychom se časově náročnému výběru lexikografického minima vyhnuli. Zapamatujme si v každém uzlu u trie lexikograficky nejmenší slovo (resp. ukazatel na něj, abychom neplýtvali pamětí), mající koncový úsek odpovídající cestě z kořene trie do u. V takto upravené trii budeme moci nalézt lexikograficky nejmenší slovo s maximálním společným úsekem v čase $\mathcal{O}(\textit{délka hledaného slova})$ a vypsat v čase $\mathcal{O}(\textit{délka minimálního slova})$, což je jistě optimální, protože to odpovídá času na načtení vstupu a vypsání výsledku. Stejně tak paměťová složitost bude lineární vzhledem k velikosti seznamu, protože velikost abecedy je konstantní a v každém uzlu trie je navíc pouze jeden ukazatel.

Konstrukce takovýchto ukazatelů je celkem jednoduchá, pokud si uvědomíme následující – nechť $v_a, v_b, \ldots v_z$ jsou syny uzlu u. Pak lexikograficky nejmenší slovo v uzlu u stačí vybírat z nejmenších slov v uzlech $v_a, v_b, \ldots v_z$ a případně slova, které končí v u. Stačí tedy procházet trii do hloubky a vždy při návratu z rekurze spočítat pro daný vrchol minimální slovo (těch je maximálně 26, což odpovídá velikosti abecedy, a tedy i počtu synů uzlů trie). Jediný zádrhel je ukryt v porovnávání minimálních slov v každém uzlu trie.

Je totiž potřeba si uvědomit, že synů uzlu je sice konstantní počet, nicméně minimální slova mohou být relativně dlouhá, a tak nalezení všech minimálních slov může trvat až $\mathcal{O}(\textit{maximální délka slova} \cdot \textit{velikost seznamu})$ (velikost trie je lineární vzhledem k velikosti slovníku). To je trochu škoda. My ale naštěstí známe slova dopředu, a tak si je můžeme očíslovat podle lexikografického pořadí a porovnávat jen jejich indexy. Hledané lexikografické pořadí jsme schopni získat za pomoci jiné

trie v čase $\mathcal{O}(velikost\ seznamu)$, což je zjevně optimální, a to tak, že nejprve vložíme všechna slova do trie (tentokrát odpředu) a tu pak šikovně celou projdeme (vždy budeme volit „nejmenšího“ ještě nepoužitého syna). Tak jsme schopni předzpracovat seznam slov v celkovém čase $\mathcal{O}(velikost\ seznamu)$.

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

#define ALPHABET_SIZE ('z'-'a'+1)   // Velikost abecedy
#define MAX_WORD_LEN 10001          // Maximální velikost dotazu nebo slova v seznamu

struct WORD
{
   char *word;    // Vlastní slovo
   int lex_id;    // Jeho lexikografické pořadí
};

struct TRIE_NODE
{
   TRIE_NODE()
   {
      for (int i=0; i<ALPHABET_SIZE; i++)
         sons[i] = NULL;
      word = NULL;
   }

   TRIE_NODE *sons[ALPHABET_SIZE];  // Synové uzlu
   WORD *word;     // Obsahuje pointer na slovo, které v daném uzlu končí, nebo NULL
   WORD *min_word; // Minimální slovo v postromu uzlu
};

// Přidání slova do trie
void trie_add(TRIE_NODE *root, WORD *word)
{
   char *word_iter = word->word;
   while (*word_iter)
   {
      int son_id = *word_iter-'a';
      if (root->sons[son_id] == NULL)
         root->sons[son_id] = new TRIE_NODE;
      root = root->sons[son_id];
      word_iter++;
   }
   root->word = word;
}

// Lexikografické očíslování slov přidaných do trie
void trie_sort(TRIE_NODE *root, int &id)
{
   if (root->word)
      root->word->lex_id = id++;
   for (int i=0; i<ALPHABET_SIZE; i++)
      if (root->sons[i])
         trie_sort(root->sons[i], id);
}

// Pro každý uzel najdeme nejmenší slovo v jeho podstromu
void trie_construct_min_words(TRIE_NODE *root)
```

```
{
   WORD *min_word = NULL;
   for (int i=0; i<ALPHABET_SIZE; i++)
      if (root->sons[i])
      {
         trie_construct_min_words(root->sons[i]);
         if (min_word==NULL || min_word->lex_id>root->sons[i]->min_word->lex_id)
            min_word = root->sons[i]->min_word;
      }
   if (min_word==NULL || (root->word!=NULL && min_word->lex_id>root->word->lex_id))
      min_word = root->word;

   root->min_word = min_word;
}

// Otočení slova
void reverse_word(char *word)
{
   char *start = word;
   char *end = word + strlen(word) - 1;
   while (start < end)
   {
      char tmp = *end;
      *end = *start;
      *start = tmp;
      start++; end--;
   }
}

// Nalezení minimálního slova s nejdelším společným úsekem k danému slovu
void trie_find_min(TRIE_NODE *root, char *to_search)
{
   while (root->sons[*to_search-'a'])
      root = root->sons[*to_search++ -'a'];

   char to_print[MAX_WORD_LEN];
   strcpy(to_print, root->min_word->word);
   reverse_word(to_print);
   printf("%s\n", to_print);
}

int main()
{
   int dict_size, query_count;
   scanf("%d %d", &dict_size, &query_count);
   WORD *dictionary = new WORD[dict_size];

   TRIE_NODE *sorting_trie = new TRIE_NODE;
   char input[MAX_WORD_LEN];
   for (int i=0; i<dict_size; i++)
   {
      scanf("%s", input);
      dictionary[i].word = new char[strlen(input)+1];
      strcpy(dictionary[i].word, input);
      trie_add(sorting_trie, &dictionary[i]);
   }
   int lex_id = 0;
   trie_sort(sorting_trie, lex_id);
```

```
   TRIE_NODE *searching_trie = new TRIE_NODE;
   for (int i=0; i<dict_size; i++)
   {
      reverse_word(dictionary[i].word);
      trie_add(searching_trie, &dictionary[i]);
   }
   trie_construct_min_words(searching_trie);

   char fake_nelze[] = "EZLEN";
   WORD fake_root_min_word;
   fake_root_min_word.word = fake_nelze;
   searching_trie->min_word = &fake_root_min_word;

   for (int i=0; i<query_count;i++)
   {
      scanf("%s", input);
      reverse_word(input);
      trie_find_min(searching_trie, input);
   }

   return 0;
}
```

P-III-2 Úřad

Úlohu bychom řešit mohli přímočaře: Nejprve odstraňme úředníky, kteří nemají žádného (ani nepřímého) podřízeného schopného styku s veřejností (to lze snadno udělat jedním průchodem v čase $\mathcal{O}(N)$). Pro každého úředníka k si pak spočítáme množinu úkonů U_k, které po něm nějaký zájemce může požadovat, a množinu úkonů V_k, které může přeposlat svému nadřízenému.

Smí-li úředník vykonávat úkon u, pak zjevně $V_k = U_k \setminus \{u\}$. Má-li úředník na starosti styk s veřejností, pak po něm může být požadován libovolný úkon a $U_k = \{1, 2, 3, \ldots, M\}$. Jinak po něm může být požadován libovolný úkon, který mu přepošle jeden z jeho přímých podřízených. Proto U_k je sjednocením všech množin V_i pro úředníky i, kteří jsou přímí podřízení úředníka k. Množiny U_k a V_k tedy můžeme spočítat jedním průchodem stromu úředníků od listů.

Přímo také můžeme vypisovat úředníky, kteří nic nedělají, tj. takové, že $u \notin U_k$. Množiny U_k a V_k si můžeme reprezentovat například polem, v němž je na i-té pozici `true`, jestliže i patří do množiny, a jinak `false`. Pak sjednocení dvou množin zabere čas $\mathcal{O}(M)$ a počet provedených sjednocení je úměrný počtu úředníků, tedy celková časová složitost bude $\mathcal{O}(MN)$.

Toto řešení se dá urychlit, když si místo množin U_k a V_k budeme pamatovat jejich doplňky $\overline{U_k}$ a $\overline{V_k}$. Pak $\overline{V_k} = \overline{U_k} \cup \{u\}$ a

$$\overline{U_k} = \bigcap_{i \text{ je přímý podřízený } k} \overline{V_i}.$$

Pro zrychlení výpočtu průniku si pro každou množinu budeme pamatovat také seznam jejích prvků. Počítáme-li $\overline{U_k}$, pak si nejprve vybereme jeho libovolného podřízeného i, z jeho množiny $\overline{V_i}$ odstraníme prvky, které nepatří do množin ostatních

podřízených, a výsledek prohlásíme za $\overline{U_k}$. Časová složitost této operace je $\mathcal{O}(1)$, jestliže k má jen jednoho přímého podřízeného, a

$$\sum_{i \text{ je přímý podřízený } k} |\overline{V_i}|$$

v ostatních případech.

Situaci nám trochu komplikují úředníci, kteří nemají žádné podřízené. Pro ně bychom potřebovali vytvářet nové množiny, což by mohlo dohromady zabrat čas až $\mathcal{O}(NM)$. Jak bychom si mohli množiny reprezentovat jinak, abychom zvládli inicializaci v konstantním čase? Jednou možností je použít uspořádaný seznam prvků. V něm pak ale test přítomnosti prvku zabere logaritmický čas, a proto bychom nedosáhli časové složitosti lepší než $\mathcal{O}(N \log N)$.

Složitější, ale rychlejší variantou je reprezentace hešovací tabulkou, jejíž velikost dynamicky přizpůsobujeme velikosti reprezentované množiny (detaily najdete třeba v Kuchařce KSP na *http://ksp.mff.cuni.cz/tasks/21/cook4.html*). Tím zůstane v průměrném případě čas $\mathcal{O}(1)$ na test přítomnosti prvku či jeho přidání a ve stejné časové složitosti zvládneme množinu i inicializovat.

Ukažme si, že toto vylepšené řešení má časovou složitost $\mathcal{O}(N)$. Kdykoliv přidáme prvek do množiny $\overline{U_k}$, abychom vytvořili $\overline{V_k}$, dáme mu jednu korunu. Výpočet množiny $\overline{U_k}$ pro úředníka s nanejvýš jedním přímým podřízeným zvládneme v konstantním čase. Předpokládejme, že počítáme $\overline{U_k}$ pro úředníka s $t \geq 2$ přímými podřízenými.

Jestliže prvek u patří do všech množin $\overline{V_i}$ těchto podřízených, pak u patří i do $\overline{U_k}$ a na ověření této skutečnosti spotřebujeme čas $\mathcal{O}(t)$. Nicméně, u v množinách podřízených má dohromady t korun, jednu z nich předá prvku u v množině $\overline{U_k}$ a zbylých $t - 1 = \Omega(t)$ použije na zaplacení spotřebovaného času.

Jestliže prvek u patří jen do $r < t$ z množin podřízených, pak u nepatří do $\overline{U_k}$ a na ověření této skutečnosti spotřebujeme čas nejvýše $\mathcal{O}(r)$. Tento čas se opět zaplatí r korunami, patřícími výskytům u v množinách podřízených. Pomocí korun přiřazených jednotlivým prvkům jsme tedy ukázali, že čas strávený počítáním průniků je omezený časem stráveným přidáváním prvků do množin V, a je tedy $\mathcal{O}(N)$.

Existuje ale i jednodušeji implementovatelné řešení se stejnou časovou složitostí. Pro každého úředníka si budeme pamatovat, zda je prokazatelně užitečný, tj. už jsme pro něj dokázali, že něco dělá. Na začátku není nikdo prokazatelně užitečný, kromě ministra. Nyní budeme procházet strom úředníků do hloubky. Při průchodu si budeme udržovat následující:

- pro každý úkon u zásobník Z_u úředníků nadřízených aktuálnímu a schopných vykonat u, v pořadí dle jejich hloubky ve stromě;
- seznam S úředníků k nadřízených aktuálnímu takových, že k zatím není prokazatelně užitečný a žádný úředník mezi aktuálním a k neprovádí úkon úředníka k.

Dorazíme-li při průchodu k úředníkovi, starajícímu se o styk s veřejností, všechny úředníky v S prohlásíme za prokazatelně užitečné a položíme $S = \emptyset$. Dorazíme-li

k úředníkovi k zpracovávajícímu jiný úkon u, pak úředníka na vrcholu Z_u smažeme z S a k přidáme do Z_u a S. Vracíme-li se z takového úředníka k, pak odstraníme k z S a z Z_u, a jestliže nový úředník na vrcholu Z_u není prokazatelně užitečný, přidáme ho do S. Na konci vypíšeme úředníky, kteří nejsou prokazatelně užiteční.

V každém vrcholu strávíme pouze konstantně mnoho času, s výjimkou označování úředníků za prokazatelně užitečné. Každého úředníka ale označíme za prokazatelně užitečného nanejvýš jednou, celková časová složitost označování tedy bude jen $\mathcal{O}(N)$. Reprezentujeme-li množinu zásobníků Z_u polem, pak bude časová složitost $\mathcal{O}(N+M)$, jelikož toto pole musíme zinicializovat. Mohli bychom ho samozřejmě reprezentovat i hešovací tabulkou, čímž by se složitost zlepšila na $\mathcal{O}(N)$. Ve vzorovém programu to ale pro přehlednost neděláme.

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

#define MAXN 1000
#define MAXM 1000

/* Zásobník */
struct ze
{
  int prvek;
  struct ze *dalsi;
};

static int vrchol(struct ze *ceho)
{
  if (!ceho)
    return 0;

  return ceho->prvek;
}

static void vloz(struct ze **kam, int co)
{
  struct ze *nw = malloc(sizeof(*nw));
  nw->prvek = co;
  nw->dalsi = *kam;
  *kam = nw;
}

static void odeber(struct ze **odkud)
{
  struct ze *a = *odkud;
  *odkud = a->dalsi;
  free(a);
}

/* Popis úředníka */
typedef struct
{
  int nadr;     /* Přímý nadřízený */
  int podr;     /* Číslo prvního z podřízených, 0 jestliže žádní nejsou */
  int kolega;   /* Následující prvek v seznamu podřízených úředníka NADR,
                   0 jestliže neexistuje */
  int ukon;     /* Prováděný úkon */
```

```
  int uzit;     /* Je prokazatelně užitečný? */
  int pozvs;    /* Pozice v poli reprezentujícím množinu S; -1 jestliže není v S */
} urednik;

static int n, m;
static urednik urednici[MAXN + 1];
static struct ze *z[MAXM + 1];
static int s[MAXN];
static int slen;

static void vyrad_ze_s(int k)
{
  int pos = urednici[k].pozvs, prep;

  if (pos == -1)
    return;

  prep = s[slen - 1];
  urednici[prep].pozvs = pos;
  s[pos] = prep;
  urednici[k].pozvs = -1;
  slen--;
}

static void vloz_do_s(int k)
{
  urednici[k].pozvs = slen;
  s[slen++] = k;
}

static void pruchod(int k)
{
  int i, p = 0, u = urednici[k].ukon;

  if (u == 1)
    {
      urednici[k].uzit = 1;
      for (i = 0; i < slen; i++)
        {
          urednici[s[i]].uzit = 1;
          urednici[s[i]].pozvs = -1;
        }
      slen = 0;
    }
  else
    {
      p = vrchol(z[u]);
      if (p)
        vyrad_ze_s(p);
      vloz(&z[u], k);
      vloz_do_s(k);
    }

  for (i = urednici[k].podr; i; i = urednici[i].kolega)
    pruchod(i);

  if (u != 1)
    {
      vyrad_ze_s(k);
```

```
      odeber(&z[u]);
      if (p && !urednici[p].uzit)
        vloz_do_s(p);
    }
}

int main(void)
{
  int i;
  const char *sep = "";

  urednici[1].pozvs = -1;
  urednici[1].uzit = 1;
  urednici[1].podr = 0;
  urednici[1].kolega = 0;

  scanf("%d%d", &n, &m);
  for (i = 2; i <= n; i++)
    {
      scanf("%d%d", &urednici[i].nadr, &urednici[i].ukon);
      urednici[i].pozvs = -1;

      int nad = urednici[i].nadr;
      urednici[i].kolega = urednici[nad].podr;
      urednici[nad].podr = i;
    }

  for (i = urednici[1].podr; i; i = urednici[i].kolega)
    pruchod(i);

  for (i = 1; i <= n; i++)
    if (!urednici[i].uzit)
      {
        printf("%s%d", sep, i);
        sep = " ";
      }
  printf("\n");

  return 0;
}
```

P-III-3 Grafový počítač v potrubí

a) Inspirujeme se klasickým tříděním výběrem minima. To funguje tak, že vybíráme nejmenší číslo a přesouváme ho na výstup. Toto opakujeme tak dlouho, dokud ještě něco ve vstupu zbývá.

Na klasickém počítači je tento algoritmu kvadraticky pomalý, ale na grafovém počítači to půjde lépe. K popisu vstupu si pořídíme graf. Pro každé číslo si založíme dva vrcholy a spojíme je hranou. Hranu ohodnotíme hodnotou čísla, značky vrcholů nám budou říkat, o kolikáté číslo v pořadí se jedná.

K výběru nejkratší hrany budeme používat funkci `Find`. Pokud totiž budeme požadovat, aby nám našla libovolnou hranu (*veq* a *eeq* nastavíme na `any`), pak nám vybere hranu s nejmenším ohodnocením. Pomocí `SumE` toto ohodnocení snadno zjistíme, vypíšeme ho do výstupu a jelikož víme, o kolikáté číslo se jedná, můžeme ho z grafu odstranit.

Jinými slovy, využili jsme grafový počítač k tomu, aby nám v konstantním čase nacházel nejmenší ze zbývajících čísel. Jelikož počáteční graf vytvoříme v lineárním čase, celkově stihneme všechna čísla setřídit lineárně. Efektivněji to jistě nepůjde, protože každé číslo musíme zapsat na výstup.

```
procedure razeni(var a: array of Integer; pocet: Integer);
var Sklad, Hrana, Nalezeno: Graph;
    i: integer;
begin
   { vytvoření grafu }
   Sklad := EmptyG;
   for i := 1 to pocet do begin
      AddV(Sklad, 2*i - 1);
      AddV(Sklad, 2*i);
      AddE(Sklad, 2*i - 1, 2*i, a[i]);
   end;

   { graf, který budeme hledat: jedna neohodnocená hrana }
   Hrana := EmptyG;
   AddV(Hrana, undef);
   AddV(Hrana, undef);
   AddE(Hrana, 1, 2, undef);

   { samotné řazení }
   for i := 1 to pocet do begin
      Nalezeno := Find(Sklad, Hrana, Any, Any);
      a[i] := SumE(Nalezeno);
      DelE(Sklad, GetV(Nalezeno, 1), GetV(Nalezeno, 2));
   end;
end;
```

b) Hledaný podgraf je *minimální kostrou* zadaného grafu. Pro hledání minimální kostry existuje mnoho standardních algoritmů, my použijeme jednu z variant Jarníkova algoritmu. Ten kostru vytváří postupným rozrůstáním. Začne s jedním libovolným vrcholem. Poté v každém kroku vybere nejkratší hranu vedoucí z již sestrojeného stromu do zbytku grafu a tu přidá ke stromu. Tím strom zvětší o jeden vrchol. Toto se opakuje, dokud strom neobsahuje všechny vrcholy.

Důkaz správnosti tohoto algoritmu odložíme na konec, nejprve popíšeme, jak ho rychle implementovat na grafovém počítači.

Představme si, že máme vstupní graf rozdělený na skupinu už propojených vrcholů a skupinu těch zbývajících. Pro každou skupinu si pořídíme nový vrchol a propojíme ho hranami se všemi vrcholy dané skupiny. Potom nalezneme libovolnou cestu délky 3 mezi oběma přidanými vrcholy. Jak mohou takové cesty vypadat? Obě jejich krajní hrany budou zajisté ty námi přidané a uprostřed může být libovolná z původních hran vedoucích mezi skupinami.

Pokud tedy budou mít všechny přidané hrany stejnou délku, nejkratší cesta tohoto typu bude obsahovat nejkratší z hran mezi skupinami, což je přesně to, co potřebujeme.

Po přidání této hrany do stromu stačí nově připojený vrchol přesunout mezi

skupinami, a to zvládneme v konstantním čase – zrušíme jednu falešnou hranu a vytvoříme novou.

Jeden krok Jarníkova algoritmu tedy provedeme v konstantním čase a jelikož kostra grafu o n vrcholech obsahuje $n-1$ hran, je časová složitost našeho algoritmu lineární s počtem vrcholů, tedy $\mathcal{O}(n)$.

Ještě dodejme, že naše řešení zachová i původní ohodnocení hran i vrcholů, což zadání nepožadovalo, ale je to tak hezčí.

```
function min_kostra(g: Graph): Graph;
var Vystup, Cesta, Nalezeno: Graph;
    Propojene, Nepropojene: Integer;
    i: Integer;
begin
   if CountV(g) < 2 then begin
      min_kostra := g; exit;   { Malé a nezajímavé }
   end;

   { Výstupní graf zatím obsahuje všechny vrcholy a žádné hrany }
   Vystup := EmptyG;
   for i := 1 to CountV(g) do begin
      { Zkopírovat vrchol a nastavit poznávací značku na náš původní }
      AddV(Vystup, GetV(g, i));
      SetV(g, i, i);
   end;

   { Vytvoření skupinek }
   Propojene := AddV(g, CountV(g) + 1);
   AddE(g, 1, Propojene, 0);
   Nepropojene := AddV(g, CountV(g) + 1);
   for i := 2 to CountV(g) - 2 do
      AddE(g, i, Nepropojene, 0);

   { Vytvoříme cestičku. Značky na vrcholech zaručí, že se
     "přilepí" správným koncem na správnou skupinku. }
   Cesta := EmptyG;
   AddV(Cesta, Propojene);
   for i := 1 to 3 do begin
      AddV(Cesta, undef);
      AddE(Cesta, i, i + 1, undef);
   end;
   SetV(Cesta, 4, Nepropojene);

   { Pojďme hledat }
   for i := 1 to CountV(g) - 3 do begin
      { Nalezneme správnou hranu }
      Nalezeno := Find(g, Cesta, Value, Any);
      { Přesuneme vrchol mezi skupinkami }
      DelE(g, Nepropojene, GetV(Nalezeno, 3));
      AddE(g, Propojene, GetV(Nalezeno, 3), 0);
      { Přidáme hranu do výstupu }
      AddE(Vystup, GetV(Nalezeno, 2), GetV(Nalezeno, 3), GetE(Nalezeno, 2, 3));
   end;
   min_kostra := Vystup;
end;
```

Na závěr dodejme slíbený důkaz správnosti. Využijeme toho, že nám zadání slibuje, že délky všech hran jsou navzájem různé. Dokážeme následujicí lemma:

Lemma: Buď F *řez* v grafu – tak říkáme libovolné množině hran grafu vedoucí mezi nějakou množinou vrcholů a všemi ostatními vrcholy. Potom nejkratší hranu tohoto řezu obsahuje každá minimální kostra.

Z tohoto lemmatu ihned plyne správnost Jarníkova algoritmu: množina všech hran vedoucích mezi aktuálním stromem a zbylými vrcholy tvoří řez v grafu, vybraná hrana je nejlehčí v tomto řezu.

Důkaz lemmatu: Provedeme sporem. Nechť F je řez mezi množinami vrcholů A a B, $f = \{u, v\}$ je nejkratší hrana tohoto řezu; bez újmy na obecnosti $u \in A$, $v \in B$. Nechť T je libovolná minimální kostra, která neobsahuje hranu f. V T proto musí existovat nějaká cesta spojující vrcholy u a v. Jelikož tato cesta začíná v A a končí v B, musí alespoň jednou přejít přes řez. Označme f' libovolnou hranu, po níž přešla. Pokud nyní v kostře T vyměníme hranu f' za hranu f, vznikne opět kostra. A jelikož f je lehčí než f', nemohla být T minimální.