Úvod do složitosti a NP-úplnosti
Verze: 12. května 1999

Tento učební text původně odpovídal přednášce „Úvod do složitosti a NP-úplnosti“, kterou vedl Mirko Křivánek na MFF UK v zimním semestru 1992/1993. Kapitoly počínaje orákuly jsem začal dopisovat na konci letního semestru 1992/1993.

Naším cílem bylo sepsat skripta, která by odpovídala obsahu této přednášky a později i navazující přednášky „Složitost a NP-úplnost“. V konečné podobě budou skripta obsahovat i řadu cvičení a námětů k přemýšlení a samostatné práci.

Skripta tohoto typu vznikají na MFF UK jako experiment. Základ tvořily poznámky studentů, kteří se uvolili je připravit v elektronické podobě. Na tomto místě musíme poděkovat zvláště Radku Lučanovi, který studentskou činnost v zimním semestru 1992/1993 organizoval a vydatně se na ní podílel. Bez něj by nás k napsání skript asi nic nepřinutilo.

V původní podobě měla skripta spoustu chyb, a to nepočítaje chyby gramatické. Vzhledem k odchodu Mirko Křivánka z MFF UK nedoznal zimní semestr skript v roce 1993/1994 výrazných změn. Změny jsou především v kapitolách „Lineární programování v rovině“, kde jsem opravil mnoho nejasností, dále v kapitole „Haldy . . . “, jejíž původní verzi jsem nahradil verzí ze svých skript o datových strukturách. Drobných změn dostala i kapitola zabývající se hledáním minimální kostry, kde jsem se snažil více přiblížit datové struktury použité v jednotlivých algoritmech.

Poznámka 0.1 Snažíme se minimalizovat počet chyb, ale to neznamená, že text skript je možno brát jako dogma. Například v zimě 1992/1993 mne na zkoušce neuspokojilo, když mi studenti, kteří si dobrovolně vybrali otázku lineárního programování v rovině, tvrdili, že jim nevadí, že jimi uváděný algoritmus nefunguje, že jim stačí, že příslušný algoritmus je ve skriptech špatně popsán.

Chtěl bych aspoň tímto poděkovat Jirkovi Fialovi který ke mně přišel po absolvování zkoušky v zimě 1993/1994 a upozornil mne na poměrně velký počet jím nalezených chyb. (Nebyly to gramatické korekce ale korekce obsahové, často se jednalo pouze o překlepy, ale i ty dokáží význam textu výrazně pozměnit.)

Není v mých silách vracet se k již napsaným kapitolám tohoto textu abych v nich vyhledával chyby. Budu vděčný každému, kdo mne na jakékoli chyby upozorní. Pokusím se poopravit kapitoly, které se Vám budou zdát výrazně obtížnější než kapitoly ostatní.

V Praze dne 22.2.1994 Vladan Majerech

V létě 1994 jsem skripta „Úvod . . . “ a „Složitost . . . “ oddělil, bylo to proto, že v letním semestru jsem v přednášení Mirka vystřídal, a zvolil jsem koncepci, která nenavazovala na zimní přednášku přímočaře.

Letos se pokusím zimní přednášku navrhnout tak, aby návaznost nebyla porušena. Chci přidat kapitolu o určování dolních odhadů složitosti, a ke konci semestru více pohovořit o výpočetních modelech a jejich vzájemné simulaci.

Vzhledem k tomu, že skripta pro letní semestr doznají ještě mnoho změn, bude ještě nějakou dobu trvat, než budu moci oba texty sjednotit.

Nyní v zimním semestru se snažím držet se skript, doplňovat je o příklady, které hodlám provádět na doprovodném cvičení. Zároveň opravuji drobné nepřesnosti, které v již napsaných kapitolách nacházím.

V Jílovém u Prahy dne 20.10.1994 Vladan Majerech

Obsah

Seznam obrázků

Seznam Algoritmů

1 Spektrum výpočetní složitosti

V poslední době je v informatice velká pozornost věnována návrhu a analýze algoritmů. Především se jedná o kombinatorické algoritmy, které prokázaly svou užitečnost v praktických optimalizačních problémech. Rozvoj oblasti návrh a analýzy algoritmů předznamenaly nové metody návrhu algoritmů, které jsou založeny na používání nových abstraktních datových struktur. Nejznámějšími příklady jsou třídící a vyhledávací algoritmy na stromových datových strukturách.

Naším cílem je představit některé užitečné zásady návrhu algoritmů (rozděl a panuj, prune and search, hladová a inkrementální metoda) na vzorku zajímavých grafových, algebraických a geometrických algoritmů. Důraz klademe na matematickou podstatu takového přístupu a rádi bychom, aby si laskavý čtenář interakci matematika vs. informatika samostatně promyslel a ocenil.

Hlavním pracovním nástrojem návrhu algoritmů je analýza jejich složitosti. Z tohoto důvodu provádíme základní členění problémů – problémy zvládnutelné (ze třídy P) a problémy nezvládnutelné (ostatní).

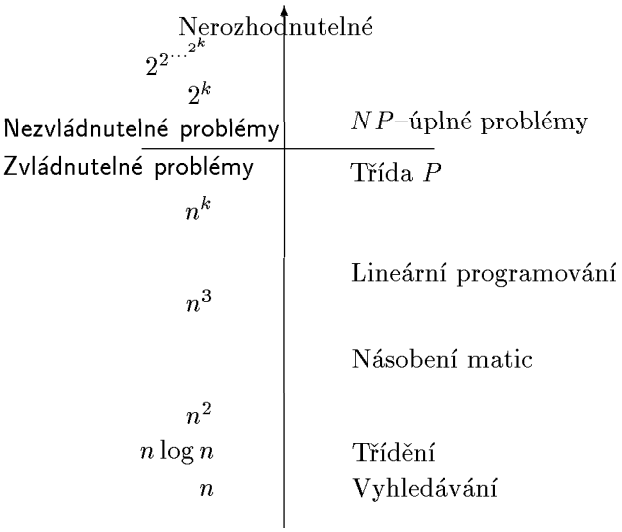
Důvod, proč za zvládnutelné problémy považujeme ty, které jsou řešitelné v polynomiálním čase, je spíše filozofický než matematický. Předně problém, řešitelný v čase n^{100} lze těžko považovat za zvládnutelný. Nicméně problémy z praxe velice zřídka vyžadují čas k řešení omezený polynomem stupně většího než 3. Dalším důvodem je fakt, že velká většina počítačových výpočetních modelů se nechá vzájemně simulovat v polynomiálním čase (např. modely z tzv. počítačové třídy $\mathcal{C}_1$: RAM, Turingův stroj, RASP, atd.). Dokonce třída problémů řešitelná na sekvenčních modelech je tatáž jako třída problémů řešitelná v polynomiálním čase na paralelních modelech s polynomiálně mnoha procesory. Posledním důvodem, který uvedeme, je příjemná vlastnost polynomiální algebry: uzavřenost na sčítání, násobení a kompozici.

U zvládnutelných problémů, tj. řešitelných v polynomiálně ohraničeném čase v délce vstupní instance, se snažíme o návrh *efektivních* algoritmů, tj. s co nejmenším stupněm polynomu. Uvědomme si, že každý navržený algoritmus poskytuje *horní odhad* složitosti. Typické odhady mají tvar funkce $n^\alpha \log^\beta n$, $\alpha, \beta \geq 0$, n je délka vstupu. Otázka dolních odhadů složitosti výpočetních problémů je mnohem komplexnější a těžší. V případě, že navrhneme algoritmus, který má časovou složitost shodnou s dolním odhadem problému, který řeší, budeme mluvit o *optimálním* algoritmu pro daný výpočetní problém. Na Obr. 1 je schematicky zachyceno uvažované spektrum výpočetní složitosti pro nejznámější problémy.

Horní asymptotické odhady budeme zapisovat pomocí operátoru O, dolní pomocí operátoru Ω. Operátor Θ bude označovat „asymptoticky optimální“, z analýzy si vypůjčíme ještě operátor o. Formálně f je $O(g)$, když existuje konstanta $c > 0$ a přirozené číslo n_0 tak, že pro všechna $n \geq n_0$ platí $f(n) \leq c \cdot g(n)$. Dále f je $\Omega(g)$ když g je $O(f)$. Konečně f je $\Theta(g)$ když f je $O(g)$ a současně f je $\Omega(g)$. Na druhé straně f je $o(g)$ když $\lim_{n\to\infty} f(n)/g(n) = 0$.

Většinou budeme časovou složitost zkoumat z hlediska *nejhoršího* případu, tj. jako funkci dat, která vedou k „nejhorší“ složitosti algoritmu. Alternativou je analýza z hlediska *průměrného* případu. Obvykle ji vztahujeme na průměrný čas daného algoritmu přes všechny možné vstupy. Hlavním problémem zde je však určení adekvátního pravděpodobnostního *rozdělení* vstupních instancí. Jinou možností je dovolit algoritmu náhodné pokračování ve výpočtu. Pak ovšem průměrnou časovou složitost algoritmu počítáme přes všechny možné průběhy algoritmu. Takové algoritmy se nazývají *randomizované*. Časovou složitost můžeme zkoumat i z hlediska nejlepšího případu.

K moderním metodám návrhu algoritmu patří specifikovat použité datové struktury až na konci celého návrhu. V průběhu návrhu specifikujeme pouze akce, jaké bude algoritmus od datové struktury (datových struktur) vyžadovat. O vhodnosti volby datové struktury často rozhoduje celkový čas práce s datovou strukturou. *Amortizovaná složitost* datové struktury je popis struktury, který nám umožňuje jednoduše počítat dobré odhady složitosti algoritmu podle četnosti použití jednotlivých operací nad datovou strukturou. Amortizovaná složitost datové struktury může být zkoumána z hlediska *nejhoršího* případu, ale i z hlediska *průměrného* případu vzhledem k pravděpodobnostnímu *rozdělení* vstupu nebo *randomizovaně*.



Obr. 1: Spektrum výpočetní složitosti

2 Rozděl a panuj, Strassenův algoritmus

Jedna z obecných metod návrhu algoritmu je metoda rozděl a panuj. Pokud se nám podaří úlohu rozdělit na menší podúlohy, každou z nich potom samostatně vyřešit a z výsledků zrekonstruovat výsledek celé úlohy, algoritmus se tím často nejen zjednoduší, ale i urychlí. Pokud se nám podaří úlohy dělit na menší a menší, až do zmenšení úlohy na triviální velikost, získali jsme tím zcela nový algoritmus.

Nás zajímá celková doba výpočtu takového algoritmu. Označíme–li $t(n)$ čas potřebný na vyřešení úlohy velikosti n, můžeme sestavit (ne)rovnici popisující chování algoritmu. Typicky získáme nerovnici následujícího typu:

$$(*) \qquad t(n) \le \sum_i^k t(\lfloor a_i n \rfloor) + f(n),$$

kde k je počet částí, na něž se nám daří úlohu dělit, $\lfloor a_i n \rfloor$ je velikost i-té části a $f(n)$ je čas strávený algoritmem na rozdělení úlohy na části a složení výsledku ($a_i < 1$).

Pro dostatečně malá n je použit triviální algoritmus, proto pro vhodné konstanty c_2 a n_0 je $t(n) \le c_2$ pro $n < n_0$.

Poznámka 2.1 Vzhledem k tomu, že velikosti částí jsou celočíselné, nejsou jejich velikosti $a_i n$, ale $[a_i n]$. Pokud by zaokrouhlování bylo směrem nahoru, můžeme způsob zaokrouhlování změnit za cenu nepatrného zvětšení konstant a_i. (Konečně mnoho špatných zaokrouhlení můžeme eliminovat posunutím počátečních podmínek)

Věta 2.1 *Nechť v rekurenci $(*)$ je $f(n)$ tvaru $c_1 n^\alpha$. Nechť β je kořen rovnice $\sum_i^k a_i^x = 1$.*

1 *Je-li $\alpha > \beta$ neboli $\sum_i^k a_i^\alpha < 1$, pak rekurence $(*)$ má pro dostatečně velikou konstantu C řešení $t(n) = Cn^\alpha$.*

2 *Je-li $\alpha = \beta$ neboli $\sum_i^k a_i^\alpha = 1$, pak rekurence $(*)$ má pro dostatečně velkou konstantu C řešení $t(n) = Cn^\alpha \log n$.*

3 *Je-li $\alpha < \beta$ neboli $\sum_i^k a_i^\alpha > 1$, pak rekurence $(*)$ má pro dostatečně velkou konstantu C řešení $t(n) = Cn^\beta$.*

Cvičení 2.1 Rozepište několikrát $t(n)$ podle rekurentního vzorce, pokuste se sledovat tvar nerovnice. Nakreslete si strom „postupného rozepisování“ rekurence a odvoďte popis funkce $t(n)$ v „řeči tohoto stromu“.

Důkaz: Představujme si, že v každém kroku nahradíme podle $(*)$ všechny výskyty funkce t. Označme s_j součet vzniklý v j-tém kroku z funkce f ($s_1 = c_1 n^\alpha$, $s_2 = c_1 \sum_i^k (a_i n)^\alpha = c_1 n^\alpha \sum_i^k a_i^\alpha, \ldots$).

V případě $\alpha > \beta$ je nejdůležitější sčítanec $s_1 = c_1 n^\alpha$, ostatní sčítance klesají geometrickou řadou s kvocientem $\sum_\ell^k a_\ell^\alpha$.

Indukcí dokážeme $t(n) \le c_2 n^\beta + c_1 n^\alpha \frac{1}{1-\sum_\ell^k a_\ell^\alpha}$. (Počáteční podmínka splněna, dokazujeme pro m, pro $n < m$ již máme dokázáno.)

$$t(m) \le \sum_i^k t(\lfloor a_i m \rfloor) + c_1 m^\alpha \overset{IP}{\le}$$

$$\overset{IP}{\le} \sum_i^k \left(c_2 (a_i m)^\beta + c_1 \frac{1}{1-\sum_\ell^k a_\ell^\alpha} \left((a_i m)^\beta - (a_i m)^\alpha \right) \right) + c_1 m^\alpha =$$

$$= c_2 m^\beta \sum_i^k a_i^\beta + c_1 m^\alpha \left(1 + \sum_i^k \frac{a_i^\alpha}{1-\sum_\ell^k a_\ell^\alpha} \right) =$$

$$= c_2 m^\beta + c_1 m^\alpha \frac{1}{1-\sum_\ell^k a_\ell^\alpha}.$$

V případě $\alpha < \beta$ je sčítanec $s_1 = c_1 n^\alpha$ málo důležitý, mnohem důležitější jsou sčítance následující, které nyní toutéž geometrickou řadou rostou. Podíváme-li se na součet zezadu, z pohledu posledních sčítanců, klesají předchozí sčítance jako geometrická řada s kvocientem $\frac{1}{\sum_\ell^k a_\ell^\alpha}$.

Indukcí dokážeme $t(n) \le c_2 n^\beta + c_1 \frac{\frac{1}{\sum_\ell^k a_\ell^\alpha}}{1-\frac{1}{\sum_\ell^k a_\ell^\alpha}} (n^\beta - n^\alpha)$. (Počáteční podmínka splněna, dokazujeme pro m, pro $n < m$ již máme dokázáno.)

$$t(m) \le \sum_i^k t(\lfloor a_i m \rfloor) + c_1 m^\alpha \overset{IP}{\le}$$

$$\overset{IP}{\le} \sum_i^k \left(c_2 (a_i m)^\beta + c_1 \frac{\frac{1}{\sum_\ell^k a_\ell^\alpha}}{1-\frac{1}{\sum_\ell^k a_\ell^\alpha}} \left((a_i m)^\beta - (a_i m)^\alpha \right) \right) + c_1 m^\alpha =$$

$$= m^\beta \sum_i^k a_i^\beta \left(c_2 + c_1 \frac{\frac{1}{\sum_\ell^k a_\ell^\alpha}}{1-\frac{1}{\sum_\ell^k a_\ell^\alpha}} \right) + c_1 m^\alpha \left(1 - \sum_i^k a_i^\alpha \frac{\frac{1}{\sum_\ell^k a_\ell^\alpha}}{1-\frac{1}{\sum_\ell^k a_\ell^\alpha}} \right)$$

$$= m^\beta \left(c_2 + c_1 \frac{\frac{1}{\sum_\ell^k a_\ell^\alpha}}{1-\frac{1}{\sum_\ell^k a_\ell^\alpha}} \right) + c_1 m^\alpha \left(1 - \frac{1}{1-\frac{1}{\sum_\ell^k a_\ell^\alpha}} \right) =$$

$$= c_2 m^\beta + c_1 \frac{\frac{1}{\sum_\ell^k a_\ell^\alpha}}{1-\frac{1}{\sum_\ell^k a_\ell^\alpha}} (m^\beta - m^\alpha).$$

V případě $\alpha = \beta$ jsou jednotlivé sčítance rovny $s_j = c_1 n^\alpha$. Sčítance přestanou být rovny $c_1 n^\alpha$ až ve chvíli, kdy se začne projevovat počáteční podmínka $n < n_0$. Ta se začne projevovat až po $\frac{\log n - \log n_0}{-\log \min_i a_i}$ krocích. Nenulových sčítanců s_j je $\frac{\log n - \log n_0}{-\log \max_i a_i}$. □

2.1 Strassenův algoritmus na násobení matic

Jako příklad odhadu časové náročnosti při řešení metodou „Rozděl a panuj“ si uveďme násobení matic. Nechť A a B jsou matice, každá řádu n a předpokládejme, že $n = 2^k$. Pak můžeme matice vynásobit takto:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$
\begin{array}{llllll}
\alpha_1 & = A_{11} - A_{21} & \beta_1 & = B_{11} - B_{21} & M_1 & = \alpha_1\beta_1 \\
\alpha_2 & = A_{11} + A_{12} & \beta_2 & = B_{11} - B_{12} & M_2 & = \alpha_2\beta_2 \\
\alpha_3 & = \alpha_1 + A_{12} & \beta_3 & = \beta_1 - B_{12} & M_3 & = \alpha_3\beta_3 \\
 & (= \alpha_2 - A_{21}) & & (= \beta_2 - B_{21}) & M_4 & = A_{12}\beta_4 \\
\alpha_4 & = \alpha_3 - A_{22} & \beta_4 & = \beta_3 + B_{22} & M_5 & = \alpha_4(-B_{21}) \\
 & & & & M_6 & = (-A_{21})(-B_{12}) \\
\sigma & = M_6 - M_3 & \gamma_1 & = \sigma + M_1 & M_7 & = (-A_{22})B_{22} \\
 & & \gamma_2 & = \sigma + M_2 & &
\end{array}
$$

$$
\begin{array}{llll}
C_{11} & = \gamma_1 + M_2 & C_{12} & = \gamma_1 + M_4 \\
 & (= \gamma_2 + M_1) & & \\
C_{21} & = \gamma_2 + M_5 & C_{22} & = M_6 - M_7
\end{array}
$$

Obr. 2: Strassenův algoritmus

kde $A_{11} \ldots A_{22}$, $B_{11} \ldots B_{22}$ a $C_{11} \ldots C_{22}$ jsou matice řádu $n/2$, přičemž pro matici C platí :

$$
\begin{array}{llll}
C_{11} & = A_{11}B_{11} + A_{12}B_{21} & C_{12} & = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} & = A_{21}B_{11} + A_{22}B_{21} & C_{22} & = A_{21}B_{12} + A_{22}B_{22}
\end{array}
$$

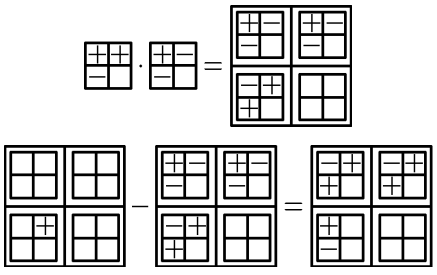
Tedy úloha vynásobit matice $n \times n$ přešla na úlohu $8\times$ vynásobit matice řádu $n/2$ a tyto pak sečíst. Tedy

$$T(n) \leq 8T\left(\frac{n}{2}\right) + \frac{wn^2}{4},$$

kde w je počet sčítání a odčítání. Použitím věty 2.1 dostaneme $T(n) \leq O(n^{\log_2 8}) = O(n^3)$. Přesto existuje postup, jak dosáhnout lepšího odhadu — tím, že nebudeme násobit 8, ale jen 7 matic řádu $n/2$, za použití $w = 15$ sčítání podle schématu na obr. 2.

Použitím věty 2.1 dostaneme, že $T(n) \leq O(n^{\log_2 7})$.

Cvičení 2.2 Dokažte korektnost Strassenova algoritmu.
Návod:



Cvičení 2.3 Rozmyslete si, jak je možno násobit „dlouhá čísla“ v čase $O(n^{\log_2 3})$, kde n je počet cifer.

Návod: $(a + Kb)(c + Kd) = ac + (bc + ad)K + bdK^2$
$(a + b)(c + d) = ac + (bc + ad) + bd$

3 Lineární algoritmus pro výpočet mediánu

neboli výběr $[n/2]$–tého prvku dle velikosti v čase $O(n)$. Nejprve provedeme analýzu funkce, která najde obecně i–tý ($i = 1, \ldots, n$) nejmenší prvek v množině o n prvcích. Pak pro $i = [n/2]$ funkce najde medián.

```
function Find(M:TSet; i:TIndex):TPrvek;
begin
  s :=Nějaký_prvek_z(M);
  M1 := {x ∈ M | x < s};
  M2 := {x ∈ M | x = s};
 (⋆ Mohou-li být v M prvky s větší násobností ⋆)
  M3 := {x ∈ M | x > s};
  if |M1| + |M2| < i then
      return (Find(M3, i − |M1| − |M2|))
  else if |M1| < i then
      return (s)
  else
      return (Find(M1, i))
end
```

Alg. 1: Algoritmus na hledání i–tého prvku v průměrném čase $O(n)$

Princip algoritmu je jistě zřejmý : Vybereme si nějaký prvek S z původní množiny M a vytvoříme tři (případně prázdné) množiny M_1, M_2 a M_3. Množina M_1 obsahuje všechny prvky z M menší než S, množina M_2 pak všechny prvky rovny S a množina M_3 všechny prvky větší než S. Je-li $|M_1 \cup M_2| < i$, znamená to, že hledaný prvek musí být v M_3, ovšem jako $(i - |M_1 \cup M_2|)$–tý nejmenší. Jinak je hledaný prvek v $M_1 \cup M_2$. Je-li $|M_1| < i$, potom hledaný prvek je v M_2, tedy S je hledaný prvek. Anebo – poslední možnost – $|M_1| > i$, a tedy hledaný prvek musí být v M_1.

Zřejmě je vidět, že v nejlepším případě, například, zvolíme-li hned v prvním kroku jako S medián, je časová náročnost $O(n)$. Zatímco v nejhorším případě, například, když za prvek S volíme vždy nejmenší prvek množiny M, je časová náročnost $O(n^2)$.

Určíme ještě náročnost v průměrném případě. Omezíme se na případ, kdy v M nejsou prvky s větší násobností. Větší násobnost může algoritmus jenom urychlit. Předpokládejme dále, že všechny vstupy algoritmu jsou stejně pravděpodobné. To znamená, že všechny možné permutace prvků množiny M mají stejnou pravděpodobnost výběru – $1/n!$.

Označme $T(n, i)$ jako průměrný čas pro **FIND**(n, i), a $T(n)$ jako maximum $\{T(n, i)\}$ přes všechna i. Pak

$$\begin{aligned} T(1) &= c \\ T(n,i) &\le \frac{1}{n}\left(\sum_{k=1}^{i-1} T(n-k, i-k) + \sum_{k=i+1}^{n} T(k-1, i)\right) \\ &\quad + cn. \end{aligned}$$

V první sumě sčítáme přes případy, kdy hledaný prvek je v množině M_3. V druhé sumě sčítáme přes případy, kdy hledaný prvek je v množině M_1. Čas potřebný na rozdělení prvků do skupin je cn. Potom

$$T(n) \le cn + \frac{1}{n} \max_i \left(\sum_{k=1}^{i-1} T(n-k) + \sum_{k=i+1}^{n} T(k-1)\right).$$

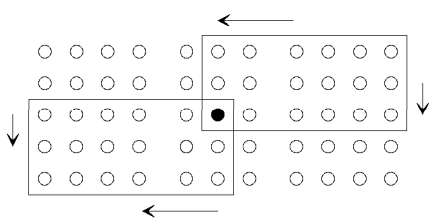
Indukcí lze z posledního vztahu ověřit, že platí $T(n) < 4cn$, tedy průměrný čas je $O(n)$.

Cvičení 3.1 Nerovnost $T(n) < 4cn$ dokažte.

Uvedeme ještě jeden algoritmus, jehož časová náročnost je také $O(n)$, ovšem na rozdíl od výše uvedeného tato náročnost zůstává i v nejhorším případě.

```
function Select(M:TSet; i:TIndex):TPrvek;
begin
  if |M| < 100 then
      return (Find(M, i));
  (⋆ na takovou ,,malou" množinu použijeme jednodušší
vyhledávání ⋆)
   n := |M|;
   Rozděl_Na_Pětice(M);
  (⋆ procedura rozdělí M na pětiprvkové množiny
M1, ..., M⌈n/5⌉ ⋆)
   for i:= 1 to ⌈n/5⌉ do
      mi:=Najdi_medián_v_pětici(Mi);
   Med:=Select({m1, ..., m⌈n/5⌉}, ⌈n/10⌉);
  (⋆ Vyhledáme medián z mediánů pětic. Toto ale není
hledaný medián celé množiny M ! ⋆)
   N1 := {x ∈ M | x <Med};
   N2 := {x ∈ M | x =Med};
   N3 := {x ∈ M | x >Med};
   if |N1| + |N2| < i then
      return (Select(N3, i − |N1| − |N2|))
   else if |N1| < i then
      return (Med)
   else
      return (Select(N1, i))
end
```

Alg. 2: Algoritmus na hledání i–tého prvku, v čase $O(n)$



Obr. 3: Činnost algoritmu *Select*

Je užitečné představit si činnost algoritmu *Select* na obrázku, srv. Obr. 3. Zde je znázorněno rozdělení do pětic,

předpokládáme, že pětice jsou setříděny vzestupně (směrem nahoru) a že střední prvky jsou mediány. Medián „Med“ z mediánů je vybarven. Dolní obdélník obsahuje prvky menší než „Med“, zatímco horní obdélník prvky větší než „Med“. Tím jsou vlastně poskytnuty dolní odhady počtu prvků v N_1 a N_3.

Lemma 3.1 $|N_1|, |N_3| \leq \lceil 7n/10 \rceil$. □

Nechť $T(n) = \max_k\{T(n,k)\}$ označuje maximální čas výpočtu procedury *Select* na datech velikosti n. Pak platí:

Lemma 3.2 *Existují konstanty a, b takové, že*

$$T(n) \leq \begin{cases} an & \text{pro } n \leq 100 \\ T(\lceil n/5 \rceil) + T(\lceil 7n/10 \rceil) + bn & \text{pro } n \geq 100 \end{cases}.$$

Důkaz: Tvrzení je zřejmé pro $n \leq 100$. Předpokládejme, že $n \geq 100$. *Select* je voláno dvakrát, jednou pro množinu velikosti $\lceil n/5 \rceil$ a jednou pro množinu velikosti nanejvýš $\lceil 7n/10 \rceil$. Celkový čas kromě rekurzivních volání je $O(n)$. □

Věta 3.3 *Algoritmus Select pracuje v lineárním čase.*

Důkaz: Tvrzení plyne přímo z věty 2.1. □

4 Lineární programování v rovině

V této přednášce si probereme určité zjemnění metody rozděl a panuj, tzv. metodu **prune & search**. V této metodě se snažíme v každém kroku zmenšit velikost úlohy z velikosti n na $c \cdot n, c < 1$. To nám pak zaručí lineární implementaci. Výklad povedeme na problému lineárního programování v rovině.

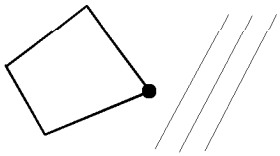
Úlohou lineárního programování rozumíme maximalizaci resp. minimalizaci vyšetřované cílové funkce vzhledem k zadané množině L omezujících nadrovin, tzv. *přípustných řešení*. Úlohu $\mathcal{L}$ lineárního programování zapisujeme ve tvaru minimalizace resp. maximalizace účelové funkce

$$\mathcal{L}(L) = \min(\max) c\vec{x}\,;\, x \in E^d,$$

kde množina L je dána soustavou lin. nerovnic

$$A\vec{x} \leq \vec{b}; A \in E^{n \times d}, d, n \in N, b \in E^n$$

Na obrázku Obr. 4 je znázorněna geometrická představa řešení úlohy lineárního programování. Konvexní oblast je množinou L přípustných řešení, rovnoběžky odpovídají účelové funkci a vyznačený extremální vrchol je řešením.



Obr. 4: Geometrická představa

Řešíme-li úlohu lineárního programování v E^1, tedy v dimenzi $d = 1$, řešení spočívá v nalezení max/min z určité množiny reálných čísel. Nás však v této kapitole bude zajímat řešení úlohy lineárního programování v dimenzi $d = 2$ vzhledem k množině přípustných řešení, kterou tvoří polygoniální konvexní oblast (může být prázdná i neomezená). Budeme zkoumat algoritmus o lineární složitosti řešící tuto úlohu.

Kromě algoritmu, který si přiblížíme, existují i další algoritmy řešící úlohu lineárního programování. Jsou to např. tzv. *simplexový* algoritmus, který prochází po vrcholech množiny přípustných řešení a hledá max/min, nebo algoritmus, který nejdříve zkonstruuje konvexní obal množiny přípustných řešení, a pak hledá v daném směru extremální vrchol, viz. Obr 4. Žádný z těchto algoritmů však neřeší úlohu lineárního programování s lineární složitostí; v prvém případě se složitostí $O(n^2)$ a ve druhém se složitostí $O(n \log n)$. Na tomto místě je vhodné připomenout nutnost předpokladu o pevné dimenzi prostoru řešení. V případě neexistence tohoto předpokladu by se zmíněné algoritmy nedaly aplikovat, neboť by se již jednalo o exponenciální algoritmy.

4.1 Lineární algoritmus lineárního programování v rovině

Nyní již přejděme k našemu zkoumanému algoritmu. Začneme tím, že si natočíme množinu přípustných řešení tak, abychom hledali minimum ve svislém směru. Toho se dá dosáhnout substitucí

$$X = x, Y = ax + by, \qquad \text{pro } b \neq 0$$

přičemž množina přípustných řešení je tvořena omezujícími nadrovinami tvaru

$$\alpha_i X + \beta_i Y + \gamma_i \leq 0, i = 1, \ldots, N,$$

kde N je počet omezujících nadrovin a $\alpha_i = a_i - (a/b)b_i$, $\beta_i = b_i/b$.

Budeme uplatňovat metodu PRUNE & SEARCH („prokousti si cestu a hledej“).

Označme si $\Sigma(L)\ldots$ prostor přípustných řešení, $D\ldots$ data.

Algoritmus Alg. 3 popisuje metodu Prune & Search.

```
begin
  if Velikost dat (D) je malá then
      Použij triviální metodu řešení
  else
      begin
        SEARCH: Následující kroky opakuj O(1) krát:
          FIND_TEST: Nalezni vhodný test t
          BISECT: Zmenši prostor Σ zodpovězením t
        PRUNE: Eliminuj data z D, která jsou
            vůči novému Σ redundantní
      end
  RECUR: Opakuj výpočet s novými D a Σ
end
```

Alg. 3: Lineární programování v rovině

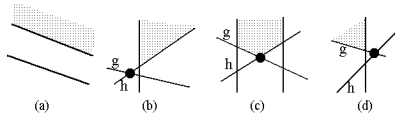
Krok **PRUNE**

Předpokládejme, že $\Sigma(L) = \{[x, y] \mid -\infty \leq a \leq x \leq b \leq \infty\}$. Uvážíme dvě množiny: $H^+\ldots$ poloroviny kladně orientované, $H^-\ldots$ poloroviny záporně orientované. Předpokládejme, že $g, h \in H^+$. Pak můžeme rozlišit tyto případy, viz Obr. 5

1. Hraniční přímky jsou rovnoběžky, tzn. že $g : y = a_1 x + b_1, h : y = a_2 x + b_2$; přičemž $a_1 = a_2, b_1 < b_2$. (viz případ (a) na Obr. 5) Je zřejmé, že nadrovina g je redundantní.

2. Průsečík nadrovin g, h leží vně pásu. Pak nadrovina g je redundantní. (viz (b) a (d) na Obr. 5)

3. Průsečík nadrovin g a h leží uvnitř pásu. Nelze nic říci. Budeme se snažit, aby tento případ nebyl moc častý. (viz (c) na Obr. 5)



Obr. 5: Postačující podmínky pro redundantnost

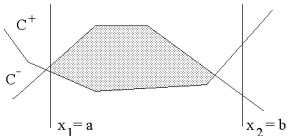
Shrnutí ke kroku PRUNE: Krok PRUNE se zabývá dvojicemi souhlasně orientovaných nadrovin. Když může, tak jednu z nich odstraní. Probere tak H^+ i H^-.

Krok **BISECT**

Funkcí tohoto kroku je zmenšení prostoru přípustných řešení asi na polovinu. Toho dosáhneme provedením určitého testu, při kterém vedeme přímku $t : x = \tau; a \leq \tau \leq b$; množinou přípustných řešení. Na základě nalezených průsečíků s hranicí množiny přípustných řešení se nám naskýtají tři možnosti, kde je třeba hledat řešení.

1. Nalevo od testu.
2. Napravo od testu.
3. Nalezeno/neexistuje.

Podrobnější prohlídka BISECTu: Zavedeme si množiny C^+, C^-. Množinu $C^+ \ldots$ definujeme jako polygoniální křivku ohraničující množinu přípustných řešení zdola, $C^+ =$ $=$ hranice$(\bigcap_{h \in H^+} \{h\})$. Symetricky C^- je polygoniální křivka ohraničující množinu přípustných řešení shora, $C^- =$ $=$ hranice$(\bigcap_{h \in H^-} \{h\})$, viz Obr. 6.



Obr. 6: Polygonální hranice C^+ a C^-

O polygonálních křivkách C^+, C^- víme jen, že jsou to množiny úseček a ty úsečky jsou, bohužel, neuspořádané. Nyní nalezneme průsečíky $(t, p^+(t))$, $(t, p^-(t))$ postupným dosazováním t za x do omezujících nadrovin. $p^+(t)$ dostaneme jako maximum přes C^+ a $p^-(t)$ jako minimum přes C^-.

Proveďme diskusi.

Je–li $p^+(t) = p^-(t)$ anebo $p^+(t) < p^-(t)$, pak $p^+(t)$ je přípustným řešením $\mathcal{L}(t)$.

Je–li $p^+(t) > p^-(t)$, potom úloha nemá řešení s x–ovou souřadnicí t.

Dále potřebujeme zjistit, zda se řešení „zlepšuje“ nalevo anebo napravo od t. Zlepšením rozumíme řešení s menší hodnotou, pokud máme přípustné řešení, nebo jakékoli přípustné řešení, pokud přípustné řešení nemáme.

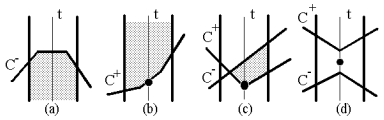
Vzhledem ke konvexitě se řešení může zlepšit jen na jedné straně od t.

Nechť $\tilde{\mathcal{L}}^+(t)$ označuje lineární program zahrnující jen ty nadroviny z H^+, které procházejí bodem $(t, p^+(t))$. Nechť $\tilde{\mathcal{L}}^-(t)$ označuje lineární program zahrnující jen ty nadroviny z H^-, které procházejí bodem $(t, p^-(t))$.

Uvažme dvě vertikální přímky $t_1 : x = \tau - 1, t_2 : x = \tau + 1$ a řešíme lineární programy $\tilde{\mathcal{L}}^+(t)$, $\tilde{\mathcal{L}}^-(t)$, pro $x = t_1$ a pro $x = t_2$. Řešení označme p_1^+, p_1^-, p_2^+, p_2^-.

Uvědomte si, že tyto programy jsou netriviální jen v degenerovaných případech, kdy existuje několik omezujících nadrovin, které procházejí $p^+(t)$ resp. $p^-(t)$. V každém případě všechny tyto jednodimenzionální programy lze vyřešit v lineárním čase.

Rozlišíme následující možnosti v závislosti na řešeních $p^+(t)$, $p^-(t)$, p_1^+, p_1^-, p_2^+, p_2^- srovnej Obr. 7:



Obr. 7: Restrikce prostoru řešení

(a) $p^+(t) = -\infty$. Pak i $\mathcal{L}$ je neomezené, viz. (a) z Obr. 7

(b) Bod $p^+(t) \leq p^-(t)$ je přípustným řešením $\mathcal{L}$

1. Je–li $p_1^+ < p^+(t)$, pak lepší řešení $\mathcal{L}$ leží nalevo od t, viz. (b) na Obr. 7.
2. Je–li $p_2^+ < p^+(t)$, pak lepší řešení $\mathcal{L}$ leží napravo od t.
3. Je–li $p_1^+ \geq p^+(t) \leq p_2^+$, pak $p^+(t)$ je optimálním řešením, viz (c) na Obr. 7.

(c) Bod $p^+(t) > p^-(t)$ je nepřípustným řešením $\mathcal{L}(t)$.

1. Je–li $p_1^+ - p_1^- < p^+(t) - p^-(t)$, pak lepší řešení $\mathcal{L}$ leží nalevo od t.
2. Je–li $p_2^+ - p_2^- < p^+(t) - p^-(t)$, pak lepší řešení $\mathcal{L}$ leží napravo od t.
3. Je–li $p_1^+ - p_1^- \geq p^+(t) - p^-(t) \leq p_2^+ - p_2^-$, pak t je „nejméně nepřípustná x–ová souřednice řešení“, úloha $\mathcal{L}$ nemá přípustné řešení, viz. (d) na Obr. 7.

Shrnutí ke kroku BISECT : Krok BISECT sestává z programů $\mathcal{L}$, $\mathcal{L}^+(t)$ a $\mathcal{L}^-(t)$, což je probírání nerovností a to stojí lineární čas. V případech, kdy budeme hledat řešení nalevo resp. napravo od t položíme $b = \tau$ resp. $a = \tau$.

Krok **FIND_TEST**

Zpárujeme nadroviny v H^+i vH^- a spočteme průsečíky párů. Pak použijeme mediánový algoritmus a spočteme

medián m z x-ových souřadnic průsečíků. Do úvahy bereme multimnožinu průsečíků, tedy i včetně multiplicity. Nakonec položíme $\tau = m$

Lemma 4.1 *V běžném kroku algoritmus buď skončí anebo se eliminuje alespoň* $\frac{n^+ + n^-}{4} - 1$ *nadrovin, kde* n^+ *je kardinalita* H^+ *a* n^- *je kardinalita* H^-.

Důkaz: Platí

$$\left\lfloor \frac{|H^+|}{2} \right\rfloor + \left\lfloor \frac{|H^-|}{2} \right\rfloor = \left\lceil \frac{n^+ - 1}{2} \right\rceil + \left\lceil \frac{n^- - 1}{2} \right\rceil.$$

Navíc

$$\left\lfloor \frac{(\lceil (n^+ - 1)/2 \rceil + \lceil (n^- - 1)/2) \rceil)}{2} \right\rfloor \geq (n^+ + n^-)/4 - 1.$$

□

Důsledek 4.1.1 *Čas algoritmu lineárního programování v rovině* $T(n)$ *vyhovuje rekurenci* $T(n) \leq T(\lceil 3n/4 \rceil) + O(n) = O(n)$. □

Věta 2.1 nám tedy zaručuje linearitu úlohy lineárního programování v rovině.

Poznamenejme na závěr, že lineární algoritmus lze rozšířit i do vyšších dimenzí E^d pro každé pevné d.

4.2 Randomizovaně lineární algoritmus lineárního programování v pevné dimenzi

V předchozí podkapitole jsme uvedli algoritmus, který je lineární v nejhorším případě. V minulé kapitole jsme ukazovali dva algoritmy hledání mediánu. Jeden z nich byl lineární v nejhorším případě, druhý byl mnohem jednodušší, a byl lineární v průměrném případě. Volíme-li v druhém algoritmu pivota pomocí náhodného generátoru a ne podle nějakého deterministického pravidla, potom neexistuje vstup, pro nějž by algoritmus pracoval nejhůř. V takovém případě hovoříme o randomizované složitosti (na rozdíl od složitosti v průměrném případě).

Následující jednoduchý randomizovaný algoritmus řeší úlohu lineárního programování v případě, kdy existuje nějaké přípustné řešení.

```
function LinProg(Omezení, Pevné : Nadroviny) : Bod;
begin
  if |Omezení| je malé then použij triviální algoritmus
  else
      begin
        Choose(h ∈ Omezení \ Pevné);
        B := LinProg(Omezení \ {h}, Pevné);
        if B vyhovuje h then return (B)
        else
            return (LinProg(Omezení, Pevné ∪ {h}))
      end
end
begin
  return (LinProg(Omezení, ∅))
end
```

Alg. 4: Randomizovaný algoritmus Lineárního programování

Myšlenka algoritmu je jednoduchá, snažíme se postupně zbavit omezujících nadrovin, které nemají vliv na optimální řešení a nalézt nadroviny, které vliv mají.

Označme d dimenzi prostoru v němž pracujeme a $t(n,k)$ randomizovaný čas funkce ***LinProg***$(Omezení, Pevné)$ pro $|Omezení| = n$ a $|Pevné| = d - k$. Uvědomme si, že počet nezávislých nadrovin, které mají vliv na optimální řešení je d.

Platí

$$t(n,k) \leq t(n-1,k) + O(1) + t(n,k-1) \cdot P,$$

kde P je pravděpodobnost, že náhodně vybraná nadrovina ovlivňuje optimální řešení. Tedy $P = k/n$ a

$$t(n,k) \leq t(n-1,k) + c + t(n,k-1) \cdot (k/n).$$

Odtud indukcí $t(n,k) \leq c \cdot c_k \cdot n$, kde $c_k \leq 1 + k \cdot c_{k-1}$ a $c_0 = 1$. Odtud $c_k \leq e \cdot k!$ a

$$t(n,k) \leq ce \cdot k!n.$$

Cvičení 4.1 Řešte tuto rekurenci, ověřte, že se jedná o optimální odhad.

Návod: $$e = \sum_{k=0}^{\infty} \frac{1}{k!}.$$

Poznámka 4.1 Velmi podobný algoritmus je možno použít na hledání nejmenšího elipsoidu obsahujícího množinu bodů.

```
function Elipsoid(Omezení, Pevné : Body) : Elipsoid;
begin
  if |Omezení| je malé then použij triviální algoritmus
  else
      begin
        Choose(x ∈ Omezení \ Pevné);
        E := Elipsoid(Omezení \ {x}, Pevné);
        if E vyhovuje x then return (E)
        else
            return (Elipsoid(Omezení, Pevné ∪ {x}))
      end
end
begin
  return (Elipsoid(Omezení, ∅))
end
```

Alg. 5: Randomizovaný algoritmus hledání nejmenšího pokrývajícího elipsoidu

Rozdíl je v tom, že zahazujeme body neurčující elipsoid a hledáme body, jenž ho určují. Je-li d počet bodů, jimiž je elipsoid určen, potom pro randomizovaný čas takového algoritmu bude opět platit $t(n,k) \leq ce \cdot k!n$, kde $t(n,k)$ je randomizovaný čas funkce ***Elipsoid***$(Omezení, Pevné)$ pro $|Omezení| = n$ a $|Pevné| = d - k$

5 Prohledávání grafů

V této přednášce se budeme zabývat metodou systematického prohledávání kombinatorických struktur. Tato činnost je známa programátorům, kteří používají metodu hrubé síly, tzv. *backtracking* při prohledávání stavového prostoru možných řešení té–které úlohy.

Naším cílem je prozkoumat systematické prohledávání grafů (multigrafů). Navrhneme optimální lineární algoritmy (v počtu hran) a na příkladech ukážeme, že tato jednoduchá metoda vyřeší na první pohled komplikované úlohy z teorie grafů.

Zkoumáním labyrintů se zabývalo mnoho matematiků, a už v roce 1895 Tarry vyslovil následující 2 pravidla pro průchod labyrinty (grafy, resp. multigrafy):

1. Každou hranou můžeme projít v 1 směru maximálně jedenkrát — (zabraňuje cyklení)

2. Hranou, kterou do uzlu vstoupíme poprvé, se můžeme vrátit, až když už není jiná možnost pokračování.

Z těchto pravidel můžeme vyvodit následující vlastnosti:
— Prohledávání je konečné. To plyne z pravidla 1.
— Časová náročnost je $O(n+m)$, kde n je počet vrcholů a m je počet hran.
— Nelze–li aplikovat žádné z pravidel 1 nebo 2, znamená to, že jsme opět na začátku a každou hranou jsme prošli právě $2\times$.

Ještě v 19. století přidal k pravidlům 1 a 2 Trámaux další pravidlo:

3 Hranou, kterou jsme vstoupili do již navštíveného uzlu, se ihned vracíme zpět.

Tolik tedy historická poznámka.

5.1 Moderní algoritmy na prohledávání grafů

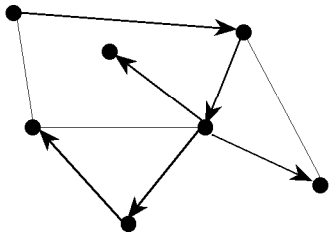
Nejpoužívanější způsoby, používané pro prohledávání grafů, jsou dva:

1. Prohledávání do hloubky (DFS - depth first search).

2. Prohledávání do šířky (BFS - breadth first search).

Předpokládejme, že prohledávaný graf je zadán takto: každý vrchol v ukazuje na seznam Adj(v) svých sousedů. Obecné zásady průchodu labyrintem pak implementujeme následujícím způsobem: navštěvované vrcholy číslujeme $1, 2, \ldots, n$ v tom pořadí jak je objevujeme. V případě, že jsme všechny vrcholy nevyčerpali, pokračujeme od prvního neobjeveného vrcholu (tj. v jiné komponentě souvislosti grafu). Při tomto postupu dostane každá hrana orientaci, šipka označuje směr průchodu. Hrany grafu můžeme rozdělit do dvou tříd: hrany *stromové*, označíme T – to jsou ty, po kterých jsme objevili nové vrcholy. Ostatní hrany nazveme *zpětné*, značíme B, viz Obr. 8.

Nyní si algoritmus popíšeme podrobněji. Začneme prohledáváním do hloubky viz algoritmus Alg. 6



Obr. 8: Průchod do hloubky. Tenké hrany jsou z B.

Hlavní program
for $x \in V$ **do** num(x) := 0;
$i := 0$; $T := B := \emptyset$
for $x \in V$ **do**
  if num(x) = 0 **then** $DFS(x)$

procedure $DFS(v)$
begin
  $i := i + 1$; num(v) := i;
  for $w \in Adj(v)$ **do**
    if num(w) = 0 **then**
      begin
        $T := T \cup (v, w)$
        $DFS(w)$
      end
    else if num(w) $\leq num(v)$**then**
      $B := B \cup (v, w)$
end

Alg. 6: Prohledávání grafu do hloubky

Uvedená rekurzivní procedura projde do hloubky daný graf v čase $O(n+m)$, kde n je počet vrcholů a m je počet hran.

V jistém smyslu opačný postup k prohledávání do hloubky je prohledávání do šířky. Zde nejdříve zkoumáme všechny hrany z průběžně navštíveného vrcholu v, pak po řadě všechny hrany ze sousedů vrcholu v, atd. V podstatě jde o rozklad vrcholové množiny do „pater“ podle vzdálenosti od startovacího vrcholu v, viz Obr. 9.

5.2 Aplikace metody DFS

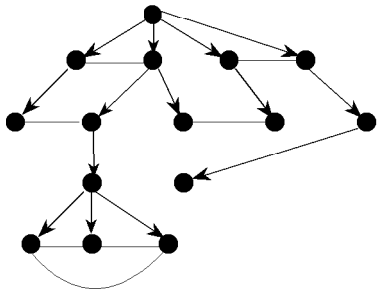
1. Hledání komponent souvislosti

2. Toplogické třídění

3. Hledání 2-souvislých komponent (vrcholově nebo hranově)

4. Hledání silně souvislých komponent

Nejjednodušší aplikací metody prohledávání do hloubky je určení počtu komponent souvislosti daného grafu G. Stačí



Obr. 9: Průchod do šířky.

si uvědomit, že množina T tvoří les v G. Jako ilustraci očíslujeme vrcholy číslem komponenty viz algoritmus Alg. 7.

Hlavní program
for $v \in V$ **do**
  comp$(v) := 0$
$c := 0$
for $v \in V$ **do**
  if comp$(v) = 0$ **then**
    $c := c + 1$; $COMP(v)$

procedure $COMP(v)$
begin
  comp$(v) := c$
  for $w \in$ Adj(v) **do**
    if comp$(w) = 0$ **then** $COMP(w)$
end

Alg. 7: Určování komponent souvislosti

Další jednoduchou aplikací DFS nalezneme v orientovaných grafech. Zde se samozřejmě vydáváme jen po orientovaných hranách grafu G. Úlohou topologického třídění je očíslovat daný orientovaný graf tak, aby platilo: Je-li $i \rightarrow j$ hrana G pak $i < j$. Samozřejmě topologicky utřídit lze jen acyklické orientované grafy. Algoritmus Alg. 8 lehce zjistí, zda daný graf je acyklický. Používá dvojí číslování label$(\cdot)$num$(\cdot)$.

Hlavní program
for $x \in V$ **do**
  num(x) :=label$(x) := 0$
$j := n + 1$; $i := 0$
for $x \in V$ **do**
  if num$(x) = 0$ **then**
    $TOPSORT(x)$

procedure $TOPSORT(v)$
begin
  $i := i + 1$;num$(v) := i$
  for $w \in Adj(v)$ ***do***
    if num$(w) = 0$ **then**
      $TOPSORT(w)$
    else if label$(w) = 0$ **then**
      Gje cyklický!
  $j := j - 1$; label$(v) := j$
end

Alg. 8: Topologické třídění

Složitější aplikací je hledání 2–souvislých komponent. Nechť $G = (E, V)$ je souvislý graf. Bod $v \in V$ nazveme artikulací, jestliže jeho vyjmutí z G zvýší počet komponent souvislosti grafu G.

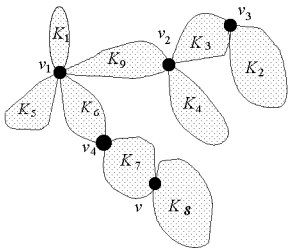
2-vrcholově-souvislá komponenta (block) K je maximální množina hran z E taková, že každé dvě hrany z K leží na prosté kružnici.

K tomu, abychom určili artikulace, využijeme následující zřejmé lemma.

Lemma 5.1 *Vrchol v je artikulací právě když existují dva vrcholy x a y a každá cesta z x do y prochází vrcholem v.*

□

Základní myšlenku algoritmu pochopíme na Obr. 10. Zde



Obr. 10: Schematické znázornění 2–souvislosti

je nakreslen graf s devíti 2–souvislými komponentami a pěti artikulacemi. Začneme–li například DFS ve vrcholu v komponentě K_9, můžeme se přes artikulaci v_2 dostat do komponenty K_4. Pak ale musíme projít celou K_4, abychom se dostali zpět do v_2. Odtud pokračujeme do další komponenty, řekněme do K_3. Tady už je situace komplikovanější, přes artikulaci v_3 můžeme pokračovat do K_2, aniž bychom prošli všechny hrany komponenty K_3. Naštěstí však procházíme strukturou, která má stromový tvar (srv. Obr. 10) a s výhodou můžeme použít zásobník na procházené hrany. Až se dostaneme zpět do komponenty K_3 budou na vrcholu již navštívené hrany K_3. Až budeme komponentu K_3 opouštět naposledy, budou v zásobníku všechny její hrany. Stačí tedy umět rozpoznávat při průchodu do hloubky artikulace.

Při průchodu budeme počítat novou funkci low(v). Její hodnotu definujeme jako nejmenší číslo num(x), kde x je předchůdce v ve stromu T a přitom existuje zpětná hrana z nějakého následníka v T vrcholu v do x (vrchol v je

sám sobě předchůdcem i následníkem). Induktivně lze low(· ·) spočítat takto: $\text{low}(v) = \min(x, y, z)$, kde

$$\begin{aligned} x &= \min\{\text{low}(w) \mid (v,w) \in T\} \\ y &= \min\{\text{num}(w) \mid (v,w) \in B\} \\ z &= \text{num}(v). \end{aligned}$$

Nyní je zřejmé, že vrchol a je artikulace právě tehdy, když existují vrcholy $v, w \in V$ takové, že $(a,v) \in T, w \neq a, w$ není následníkem v v T a $\text{low}(v) \geq \text{num}(a)$. Jediné co tedy potřebujeme, je spočítat funkci low při průchodu do hloubky. Díky její rekurzivní definici je to již snadná záležitost viz algoritmus Alg. 9.

```
Hlavní program
i := 0; S :=prázdný zásobník
for x ∈ V do num(x) := 0;
for x ∈ V do
    if num(x) = 0 then BICON(x)

procedure BICON(v)
begin
  i := i + 1; num(v) :=low(v) := i;
  for w ∈Adj(v) do
      if num(w) = 0 then
          begin
            Push(S, (v, w));
            BICON(w);
            low(v) := min(low(v),low(w));
            if low(w) ≥num(v)then
            v je artikulace nebo kořen T,
            založíme novou komponentu,
            ze zásobníku odebereme její hrany
            až po hranu (v, w), nebo konec zásobníku
          end
      else if num(w) <num(v) then
          begin
            Push(S, (v, w));
            low(v) := min(low(v),num(w))
          end
end
```

Alg. 9: Hledání dvousouvislých komponent, výpočet funkce *Low*.

Cvičení 5.1 Most (bridge) je hrana, jejímž vyjmutím z G sc zvětší počct komponcnt souvislosti grafu G.

2-hranově-souvislá komponenta (bridge-block) K, je maximální množina vrcholů z V taková, že každé dva vrcholy z K jsou spojeny dvěma hranově disjunktními cestami.

Modifikujte algoritmus tak, aby hledal 2-hranově souvislé komponenty, (popřípadě vyhledával i mosty).

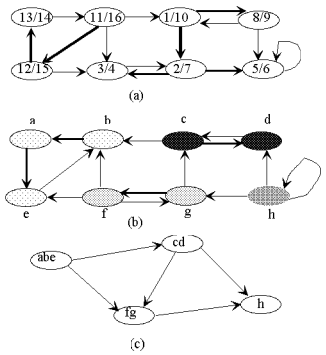
Poslední aplikací, kterou uvedeme, bude hledání silně souvislých komponent v orientovaném grafu G. Silně souvislá komponenta je maximální množina hran s vlastností, že pro libovolnou dvojici vrcholů v a w existuje orientovaná cesta z v do w a naopak z w do v. Definujeme graf G^T, který z grafu G vznikne tak, že obrátíme orientaci všech hran. Zřejmě G i G^T mají totožné silně souvislé komponenty.

Uvažme algoritmus Alg. 10, který je založený na prohledávání do hloubky. Představme si že čísluje vrcholy dvojicí čísel $(x(\cdot), y(\cdot))$, srv. (a) na Obr. 11, takto: Nechť je k dispozici inkrementální čítač, který se zvyšuje vždy po navštívení vrcholu. Poprvé když objevíme vrchol v, v přiřadíme hodnotu čítače jako první číslo $x(v)$, opouštíme–li naposledy vrchol v, přiřadíme mu druhé číslo $y(v)$.

```
begin
  Prohledej graf G do hloubky a každý
  vrchol v očísluj (x(v), y(v))
  Spočti G^T
  Prohledej do hloubky graf G^T v pořadí
    klesající posloupnosti y(·), spočítané
    v kroku 1
  Stromy v lese T jsou hledané
  silně souvislé komponenty grafu G
end
```

Alg. 10: Hledání silně souvislých komponent

Graficky je algoritmus znázorněn na Obr. 11. V části (a) silné šipky ukazují DFS les T, každý uzel v je očíslován dvojicí $(x(v), y(v))$. V části Obr. 11 je zobrazen duální graf G^T. Různými druhy rastru jsou vyznačeny stromy DFS lesa T v G^T. Část (c) ukazuje acyklický kondenzovaný graf, který vznikl ze silných komponent grafu G.



Obr. 11: Hledání silně souvislých komponent

Klíčovým pozorováním, na kterém je algoritmus založen, je fakt, že žádná cesta mezi dvěma vrcholy silně souvislé komponenty, tuto komponentu neopouští (kondenzovaný graf je acyklický). Při průchodu do hloubky tedy opouštíme silně souvislou komponentu až když ji celou projdeme a to v uzlu, ve kterém jsme do ní vstoupili. Pomocí těchto pozorování lze dokázat správnost algoritmu indukcí podle počtu nalezených stromů v lese T v grafu G^T.

Nejdůležitější aplikací prohledávání do hloubky je (dvouprůchodový) algoritmus testující rovinnost grafu, který jej

případně vnoří do roviny. Tento algoritmus by si však vzhledem ke své komplikovanosti vyžadoval samostatnou kapitolu. Prozatím nám bude stačit, že pracuje v čase lineárním v počtu vrcholů grafu (multigrafu).

5.3 Terminologie algoritmu testování rovinnosti

Prohledávání do hloubky každé hraně původně neorientovaného grafu přiřadí orientaci, zároveň jsou prohledáváním hrany každé komponenty rozděleny na stromové a zpětné.

Základní vlastností stromových hran je, že tvoří strom orientovaný z vrcholu, v němž začalo prohledávání, každá zpětná hrana (a, b) má vlastnost, že z její hlavy (z vrcholu b) vede stromová orientovaná cesta do jejího ocasu (do vrcholu a). (Takovému orientovanému grafu s rozdělením hran na stromové a zpětné, se říká palmový strom.)

Pro prvek x palmového stromu (x je hrana nebo vrchol) definujeme množinu následníků $\mathrm{desc}(x)$. Vrchol y je následníkem prvku x ($y \in \mathrm{desc}(x)$), pokud existuje orientovaná stromová cesta z x do y. Hrana (y, z) je následníkem prvku x, pokud $y \in \mathrm{desc}(x)$ nebo $(y, z) = x$. (Definice je komplikovaná, protože chceme aby zpětné hrany byly následníky. Jinak bychom mohli mluvit o podstromu.)

Pro prvek x dále definujeme množinu dotyků $\mathrm{att}(x)$. Vrchol y je dotykem x pokud je y hlavou nějaké hrany, která je následníkem x a navíc je x následníkem y. Hrana (y, z) je dotykem x, jedná-li se o zpětnou hranu a navíc je x následníkem z. Množinu $\mathrm{att}(x)$ dělíme na množiny vrcholů $\mathrm{att}^v(x)$ a hran $\mathrm{att}^e(x)$.

V prvním prohledávání do hloubky algoritmus testování rovinnosti spočítá pro každou zorientovanou hranu e při vynořování se z rekurence hodnoty $\mathrm{low}_1(e) = \min\{\mathrm{num}(y) \mid y \in \mathrm{att}^v(e)\}$ a $\mathrm{low}_2(e) = \min\{num(y) \mid y \in \mathrm{att}^v(e) \wedge \mathrm{num}(y) \neq \mathrm{low}_1(e)\}$.

Vzhledem k tomu, že chceme dosáhnout času $O(n)$, kde n značí počet vrcholů, nemůžeme si dovolit procházet všechny hrany, pokud je graf příliš hustý. Z Eulerovy věty ale víme, že v rovinném grafu nemůže být víc než $3n$ hran. Proto v průběhu algoritmu počítáme navštívené hrany, a pokud kdykoli bude trojnásobek počtu navštívených vrcholů menší než počet navštívených hran, ukončíme předčasně algoritmus s tím, že graf není rovinný. Tímto máme garantovaný čas $O(n)$ pro první fázi.

Pro druhou fázi algoritmu je potřeba připravit vhodné pořadí hran vycházejících z jednotlivých vrcholů. Ukazuje se, že vhodné pořadí je pořadí podle funkce $\varphi((a, b)) = 2\mathrm{num}((a, b)) - (\mathrm{low}_2((a, b)) \geq \mathrm{num}(a))$, kde předpokládám běžné konvence pro výsledek porovnání (pravda = 1, nepravda = 0). V mezifázi je potřeba pro každý vrchol a setřídit hrany s ocasem a vzestupně podle φ. Nechť e_1^a, e_2^a, ... je výsledný seznam, kde $\varphi(e_1^a) \leq \varphi(e_2^a) \leq \cdots$. Formálně bychom za tyto orientované hrany mohli do seznamu sousedů vrcholu a přidat všechny orientované hrany s hlavou a. Tak bychom dostali reprezentaci původního neorientovaného grafu. Při prohledávání do hloubky takto reprezentovaného grafu dostaneme stejný palmový strom jako při prvním prohledávání (cvičení).

Vzhledem k tomu, že se snažíme o algoritmus složitosti $O(n)$, nezbývá nám čas na třídění pomocí porovnávání (viz kapitola o dolních odhadech). Řešením je naalokovat si pole velikosti $2n$ (rozsah možných hodnot funkce φ), a v tomto poli na místě i uchovávat ukazatel na spojový seznam hran, kde $\varphi = i$. Postupně zařadíme všechny hrany do seznamů podle φ, poté spojíme seznamy a dostaneme tak seznam S seřazený podle φ (raději sestupně). Zatím jsme potřebovali čas $O(n)$ pro zatřídění hran do seznamů a $O(n)$ na spojení $2n$ seznamů (včetně seznamů prázdných). Abychom dostali seznamy e_i^a pro jednotlivé ocasy a, naalokujeme pole velikosti n, a v tomto poli na místě i budeme uchovávat ukazatele na spojový seznam hran s ocasem a kde $\mathrm{num}(a) = i$. Každý spojový seznam již bude setříděný podle φ. Po zařazení všech prvků seznamu S budeme mít vše připraveno pro spuštění druhého prohledávání grafu do hloubky.

Při druhém prohledávání do hloubky již budeme testovat možnost vnoření do roviny (případně tvořit vnoření). Zaveďme nyní terminologii nutnou pro druhé prohledávání. Situaci si zjednodušíme tím, že budeme testovat rovinnost zvlášť pro každou komponentu vrcholové dvousouvislosti.

Začneme definicí pojmu první cesta z prvku x. Je-li x vrchol, je první cesta x složena z x a z první cesty z e_1^x. Je-li (y, z) stromová hrana, pak je první cesta (y, z) složena z (y, z) a z první cesty ze z. Je-li (y, z) zpětná hrana, pak první cesta z (y, z) obsahuje pouze tuto hranu. Důležitým pojmem druhého prohledávání je základní cyklus $\mathrm{cycle}(x)$ daného prvku x. Základní cyklus prvku x je tvořen zpětnou hranou (y, z), do níž vede první cesta z x, a stromovou cestou ze z do y (cvičení:jednoznačnost). Cestě ze z do y se říká páteř cyklu. Páteř cyklu $\mathrm{cycle}(x)$ obsahuje první cestu z x (dvousouvislost,cvičení).

V průběhu druhého prohledávání, při práci s hranou $e = (u, v)$ budeme vytvářet strukturovanou informaci o množině $\mathrm{att}^e(e)$ na základě strukturovaných informací o $\mathrm{att}^e(e_i^y)$. Informace $\mathrm{att}^e(e)$ bude rozdělena na bloky vnoření, což jsou množiny hran z $\mathrm{att}^e(e)$, které nutně leží na stejné straně cyklu $\mathrm{cycle}(e)$. Bloky vnoření budou sdružovány do antagonistických dvojic bloků. Dva bloky jsou antagonistické, pokud hrany jednoho bloku nutně v libovolném rovinném vnoření leží na opačné straně cyklu $\mathrm{cycle}(e)$ než hrany druhého bloku. (Pokud jsou dvojice bloků (B_1, B_2) a (B_1, B_3) antagonistické, je nutně B_2 a B_3 tentýž blok.)

Bloky vnoření budeme udržovat ve struktuře, která umožní nalezení $\mathrm{first}(B) = \min\{\mathrm{num}(v) \mid (u, v) \in B\}$ a $\mathrm{last}(B) = \max\{\mathrm{num}(v) \mid (u, v) \in B\}$, spojení dvou bloků, a odstranění nespecifikované hrany, kde num hlavy je rovno $\mathrm{last}(B)$, vše v konstantním čase na operaci (k tomu stačí hrany bloku udržovat v oboustranném spojovém seznamu s rostoucím num hlav hran).

Struktura $\mathrm{att}^e(e)$ je tvořena seznamem dvojic bloků $(b(e)_1^1, b(e)_2^1)$, $(b(e)_1^2, b(e)_2^2)$, ..., $(b(e)_1^k, b(e)_2^k)$. Seznam dvojic bloků je uspořádán podle rostoucího $\mathrm{last}(b(e)_1^j)$. Dvojice bloků $(b(e)_1^j, b(e)_2^j)$ je standardizována, pokud $\mathrm{last}(b(e)_1^j) \geq \mathrm{last}(b(e)_2^j)$.

Pro stromovou hranu e je $\mathrm{att}^e(e)$ počítáno ve čtyřech fázích.

V první fázi jsou z posledních dvojic bloků odstraněny případné hrany vedoucí do hlavy hrany e.

Pokud má hrana e více než jednoho následníka, budou provedeny i fáze 2, 3 a 4.

V druhé fázi je pro každé $i > 1$ struktura $\mathrm{att}^e(e_i)$ sloučena do jednoho bloku $b(e_i)$ (celé $\mathrm{att}^e(e_i)$ leží na stejné straně cyklu $\mathrm{cycle}(e)$ jako hrana e_i). Druhá fáze skončí neúspěchem (graf není rovinný), pokud pro nějaké j se blok $b(e_i)_2^j$ dotýká i v jiném bodě než $\mathrm{low}(e_i)$ (dvojice bloků jsou standardizovány, proto se pak i $b(e_i)_1^j$ dotýká v jiném bodě než $\mathrm{low}(e_i)$).

Ve třetí fázi je $\mathrm{att}^e(e)$ inicializováno počátkem seznamu $\mathrm{att}^e(e_1)$, zbytek seznamu $\mathrm{att}^e(e_1)$ (je od konce seznamu nalezeno nejmenší j kde $\mathrm{last}(b(e_1)_1^j) > \mathrm{low}(e_2)$ jedná se o bloky $b(e_1)_1^j$, $b(e_1)_2^{j+1}$, ...) je pospojován do jednoho bloku $b(e_1)$ (celý konec seznamu musí být vnořen na druhé straně $\mathrm{cycle}(e)$ než zpětná hrana kružnice $\mathrm{cycle}(e_2)$). Třetí fáze skončí neúspěchem (graf není rovinný), pokud některý blok $b(e_2)_2^k$ $(k \geq j)$ je neprázdný.

Ve čtvrté fázi jsou bloky $b(e_i)$ zařazovány na konec seznamu $\mathrm{att}^e(e)$. V této fázi algoritmu je nutné, aby poslední dvojice bloků (b_1^ℓ, b_2^ℓ) seznamu $\mathrm{att}^e(e)$ byla standardizována. Pokud $\mathrm{low}(e_i) \geq \mathrm{last}(b_1^\ell)$, je na konec přidána dvojice bloků $(b(e_i), \emptyset)$. Pokud $\mathrm{last}(b_1^\ell) > \mathrm{low}(e_i) \geq \mathrm{last}(b_2^\ell)$, je blok $b(e_i)$ připojen k bloku b_2^ℓ a dvojici bloků (b_1^ℓ, b_2^ℓ) je potřeba opět standardizovat. Pokud $\mathrm{last}(b_2^\ell) > \mathrm{low}(e_i)$, končí fáze neúspěchem (graf není rovinný). Skončením čtvrté fáze výpočet $\mathrm{att}^e(e)$ končí.

Pro zpětnou hranu e je $\mathrm{att}^e(e)$ tvořeno jedinou dvojicí bloků $(\{e\}, \emptyset)$.

Graf je rovinný, pokud algoritmus „neskončí neúspěchem“.

Pokud bychom chtěli zároveň s testováním rovinnosti konstruovat rovinné vnoření, potom by operace spojování bloků vytvářela „nedokončené stěny“ ...přidání pomocné hrany mezi první a poslední body dotyku (jsou-li tyto body stejné, jedná se o dokončenou stěnu) by nedokončenou stěnu ozavřelo. Tato operace by vyžadovala čas úměrný počtu hran nedokončené stěny (pro hraniční zpětné hrany bychom nalezli poslední společný vrchol stromu tak, že bychom zároveň z obou z nich značili zpětnou cestu dokud bychom nenarazili na značku).

Nedokončená stěna je v algoritmu dokončena při provádění první fáze výpočtu $\mathrm{att}^e(e)$, kde hlava e byl první bod dotyku nedokončené stěny.

Vzhledem k tomu, že graf může mít více rovinných vnoření, můžeme mít více možností, jak pospojovat bloky ve druhé fázi. Nechť e_b je zpětná hrana cyklu $\mathrm{cycle}(e_i)$. Pokud je graf rovinný, pak určitě funguje metoda vytvořit první blok spojením prvních bloků každé dvojice bloků, vytvořit druhý blok spojením druhých bloků každé dvojice bloků (mají dotyky jen v bodě $\mathrm{low}(e_i)$), a připojit první blok „pod“ hranu e_b a druhý blok „naruby nad“ hranu e_b.

6 Věta o planárním separátoru

Věta 6.1 *Nechť $G = (V, E)$ je neorientovaný rovinný graf, pak je možno vrcholy rozdělit na disjunktní množiny A, B, S následujících vlastností:*

1. $|A|, |B| \leq \frac{2}{3}|V|$
2. $|S| < 4\sqrt{|V|}$
3. $(A \times B) \cap E = \emptyset$ *(S je separátor)*

Navíc takové množiny mohou být nalezeny v lineárním čase.

Označme $n = |V|$ počet vrcholů grafu. Důkazem věty bude popis algoritmu, který v čase $O(n)$ požadované množiny nalezne.

Nejprve předpokládejme, že umíme řešit úlohu pro souvislé grafy. Ukážeme, jak potom vyřešíme případ grafu o více komponentách.

Algoritmus nejprve spočítá velikosti jednotlivých komponent a nalezne největší komponentu.

- Je-li velikost největší komponenty větší než $\frac{2}{3}n$, nalezneme nejprve separátor S_1 a množiny A_1, B_1 pro tuto souvislou komponentu. Položíme $S = S_1$, spočítáme velikost A_1 a B_1, a v dalším se k A_1 a B_1 chováme jako ke komponentám grafu.
- Jinak položíme $S = \emptyset$.

Zbytek důkazu plyne z následujícího lemmatu.

Lemma 6.2 *Pokud se nám podařilo vrcholy grafu (V, E) rozdělit na množiny S, V_1, V_2, ..., V_k, tak, že*

1. $\forall i \ |V_i| \leq \frac{2}{3}n$
2. $|S| < 4\sqrt{n}$
3. $\forall i \neq j \ (V_i \times V_j) \cap E = \emptyset$,

Pak sjednocením několika množin V_i můžeme získat množinu A, a sjednocením zbývajících množin V_i množinu B požadovaných vlastností ($|A|, |B| \leq \frac{2}{3}n$).

Požadované sjednocení je možno získat v lineárním čase.

Důkaz: Budeme hledat vhodnou množinu $A \mid |A| \geq \frac{1}{3}n$. Za B potom položíme $V \setminus (A \cup S)$.

V lineárním čase zjistíme velikosti množin V_i, a nalezneme největší. (Pokud již velikosti známe, stačí nalézt největší množinu). Inicializujme A touto největší množinou. Pokud $|A| \geq \frac{1}{3}n$, pak algoritmus končí. Jinak přidáváme do A postupně množiny V_i. Přidávání končí, když $|A| \geq \frac{1}{3}n$. V tu chvíli však $|A| < \frac{2}{3}n$, protože poslední přidaná množina nebyla větší než $\frac{1}{3}n$ ($\max_i |V_i| < \frac{1}{3}n$). □

Zabývejme se nyní hledáním separátoru v souvislém rovinném grafu. Základní myšlenkou, je nalezení malé kružnice, obsahující zhruba stejně vrcholů uvnitř jako vně v nějakém vnoření grafu do roviny. Zvolíme-li S rovno vrcholům této kružnice, A vrcholům uvnitř a B vrcholům vně, budou množiny A a B množinou S podle Jordanovy věty odděleny (separovány). (Jordanova věta tvrdí, že uzavřená křivka, která se nikde neprotíná rozděluje rovinu na dvě části. Důkaz této věty, která zní tak samozřejmě je však velmi komplikovaný.)

Otázkou zůstává, jak takovou malou kružnici nalézt. Jednou z možností je zvolit vhodnou kostru grafu a hledat mezi kružnicemi, obsahujícími cestu kostry a jednu hranu (nazvěme ji příčkou). Pokud by nejdelší cesta kostry byla dlouhá l, kružnice by obsahovala nejvýš l vrcholů. Graf vnoříme do roviny, abychom mohli počítat počet vrcholů uvnitř/vně kružnice. (Věříme tomu, že existuje algoritmus vnořující rovinný graf do roviny v lineárním čase.)

Hledání kružnic tvořených jednou příčkou a cestou kostry má výhodu, že každá příčka definuje kružnici jednoznačně. Je-li určena vnější stěna grafu je určen jednoznačně i vnitřek kružnice. Můžeme tedy počet bodů uvnitř kružnice přiřadit příslušné příčce.

Ke konci kapitoly bude popsán algoritmus, který nalezne vhodnou příčku v (dotriangulovaném) grafu. Předtím ale musíme vyřešit problém, kdybychom neuměli rychle nalézt kostru, v níž by nejdelší cesta byla dostatečně krátká.

Vhodným algoritmem na hledání 'krátké' kostry je průchod grafu do šířky. Vezmeme nějaký vrchol s grafu a začneme procházet graf do šířky z vrcholu s. Nechť L_i je množina vrcholů ve vzdálenosti i od s (množině L_i říkejme vrstva). V průběhu prohledávání do šířky vytváříme příslušnou kostru, vrstvy L_i a evidujeme velikosti vrstev. Zároveň evidujeme mezisoučet jejich velikostí. Označme l_1 index, kdy tento mezisoučet překročil $\frac{n}{2}$.

Může se stát, že počet vrstev grafu je příliš velký. V tom případě jsme 'krátkou kostru' nenalezli, a musíme hledat separátor jinak.

Jakákoli vrstva L_i by mohla sloužit jako separátor vrcholů ve vzdálenosti od s menší než i od vrcholů ve vzdálenosti větší než i. Speciálně, otestujeme, je-li $|L_{l_1}| < 4\sqrt{n}$. Pokud ano, zvolíme $S = L_{l_1}$, $A = \bigcup_{i<l_1} L_i$, $B = V \setminus (A \cup S)$, a algoritmus končí.

Pokud L_{l_1} je 'velká', nalezneme dva indexy l_0, l_2 následovně: $l_0 = \max_i\{i < l_1 \mid |L_i| < \sqrt{n}\}$, $l_2 = \min_i\{i > l_1 \mid |L_i| < \sqrt{n}\}$. Poznamenejme, že indexy l_0, l_2 můžeme počítat rovnou při průchodu grafem do šířky.

Lemma 6.3 $l_2 - l_0 < \sqrt{n} - 2$

Důkaz: Sporem předpokládejme, že $l_2 - l_0 \geq \sqrt{n} - 2$. Spočítejme velikost $\bigcup_{i=l_0}^{l_2} L_i$. Kromě L_{l_0} a L_{l_2} všechny vrstvy L_i sjednocení obsahují aspoň $\sqrt{n}$ vrcholů. Vrstva L_{l_1} dokonce aspoň $4\sqrt{n}$ vrcholů. Nepočítáme-li vrstvy L_{l_0} a L_{l_2}, má sjednocení aspoň

$$(l_2 - l_0 - 2) \cdot \sqrt{n} + 4\sqrt{n} \geq (\sqrt{n} - 4) \cdot \sqrt{n} + 4\sqrt{n} \geq n.$$

Ovšem vrstva L_{l_0} je neprázdná, což je spor s tím, že počet vrcholů grafu je n. □

Označme nyní

$$V_0 = \bigcup_{i<l_0} L_i, \quad V_1 = \bigcup_{l_0<i<l_2} L_i, \quad V_2 = \bigcup_{l_2<i} L_i.$$

Pokud velikost největší z množin V_0, V_1, V_2 je nejvýš $\frac{2}{3}n$, pak zvolme $S = L_{l_0} \cup L_{l_2}$ a tvrzení věty plyne z lematu 6.2.

Vzhledem k volbě l_1 víme, že $|V_0| < \frac{n}{2}$ a $|V_2| < \frac{n}{2}$. Zbývá nám tedy jedině případ $|V_1| > \frac{2}{3}n$. Ale pro množinu V_1 jsme tímto postupem již nalezli 'dostatečně krátkou kostru'. Můžeme proto postupovat podle původního záměru.

Přidáme k podgrafu indukovanému množinou vrcholů V_1 vrchol s a spojíme jej se všemi vrcholy z L_{l_0+1}. Vzniklý graf je určitě rovinný, protože jej můžeme z grafu G zkonstruovat vypuštěním některých vrcholů, hran a kontrakcí některých hran. Vnoříme jej do roviny. Hrany z vrcholu s včetně původních hran průchodu do šířky tvoří kostru v 'nadgrafu' G' grafu indukovaného množinou vrcholů V_1, v níž nejdelší cesta je dlouhá $2(l_2 - l_0 - 1) < 2(\sqrt{n} - 3)$.

Separátor v grafu G' budeme hledat ve tvaru kružnice, tvořenou jednou příčkou a hranami kostry. Pokud bude vrchol s součástí kružnice, nemusíme jej do separátoru zahrnout. (Nejdelší cesta kostry, když do její délky nepočítáme vrchol s je dlouhá nejvýš $2\sqrt{n}-7$.) Kružnice bude separátorem na základě Jordanovy věty. Abychom tuto větu mohli použít, nepotřebujeme, aby příslušná křivka byla tvořena hranami grafu. Stačí, aby ji žádná hrana grafu neprotínala. To nám umožnilo přidat vrchol s. Dále nám to umožňuje přidat ke grafu nějaké hrany (příčky) tak, aby graf zůstal rovinný, a hledat i mezi takto vzniklými kružnicemi. Nejlépe se nám bude kružnice hledat, když celý graf dotriangulujeme. (Projdeme celý graf, a pokud stěna obsahuje víc jak 3 vrcholy, přidáme do ní úhlopříčku. To je možno provést v lineárním čase.)

Zbývá tedy popsat algoritmus, který v triangulovaném grafu G s danou kostrou T nalezne kružnici tvořenou jednou příčkou a cestou kostry tak, aby vně i uvnitř kružnice bylo nejvýš $\frac{2}{3}n$ vrcholů.

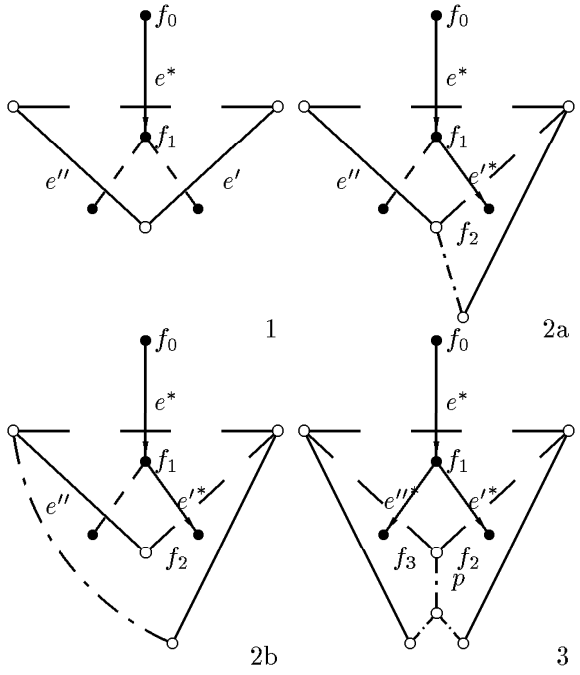
Tento algoritmus pracuje na duálním grafu. Ke každému grafu G vnořenému do roviny můžeme v lineárním čase zkonstruovat duální graf G^*, tj. graf, jehož vrcholy jsou stěny původního grafu a naopak. Každé hraně e primárního grafu odpovídá hrana e^* duálního grafu (mezi stěnami obsahujícími původní hranu). Definujme duální kostru T^* ke kostře T. Hrany kostry T^* jsou duální hrany k hranám, které nejsou v kostře T.

Cvičení 6.1 Dokažte, že „duální kostra“ je kostra.

Náš algoritmus bude pracovat s duálním grafem G^* ke triangulaci G. Proto je G^* 3-regulární graf. Zvolme libovolný list r kostry T^* (r bude 'vnější stěna'), a zorientujme hrany kostry T^* duální k T směrem od kořene r.

Z každého vrcholu grafu G^* vychází nejvýše dvě orientované hrany kostry T^*. Každá hrana e^* kostry T^* odpovídá příčce e původní kostry T. V postorder pořadí při prohledávání kostry T^* do hloubky spočítáme pro každou hranu $e^* \in T^*$ (pro příčku e) počet $I(e)$ vrcholů uvnitř kružnice příslušné příčce e. Abychom byli schopni $I(e)$ spočítat, budeme potřebovat konstruovat cestu $c(e)$ v kostře T mezi koncovými vrcholy hrany e, a počítat její délku $d(e)$ (počet hran).

Zbývá nám popsat, jak je možno spočítat $I(e)$, $c(e)$ a $d(e)$ pro hranu e^* kostry T^* na základě hodnot $I(e_i)$, $c(e_i)$ a $d(e_i)$ pro její následníky e_i^*. Potom ukážeme, že pro nějakou hranu e^* bude $\frac{1}{3}n \le I(e) \le \frac{2}{3}n$.



Obr. 12: Situace při hledání planárního separátoru

Na obrázku 12 jsou zobrazeny situace, které mohou nastat při počítání $I(e)$, $c(e)$, $d(e)$. Čerchovaně jsou naznačeny cesty v kostře T.

1 $e^* = (f_0, f_1)$ je list kostry. Označme e'^*, e''^* zbývající hrany grafu G^* obsahující f_1. Potom $I(e) = 0$, $c(e) = (e', e'')$ a $d(e) = 2$.

2 $e^* = (f_0, f_1)$ má jediného následníka $e'^* = (f_1, f_2)$ v kostře. Označme e''^* zbývající hranu grafu G^* obsahující f_1.

   a Není-li e'' krajní hranou cesty $c(e')$, pak e'' sousedí s touto cestou. Nechť $c(e)$ je cesta, která vznikne připojením e'' k cestě $c(e')$, nechť $I(e) = I(e')$, a $d(e) = d(e') + 1$.

   b Je-li e'' krajní hranou cesty $c(e')$, nechť $c(e)$ je cesta, která vznikne odebráním e'' z cesty $c(e')$, nechť $I(e) = 1 + I(e')$, a $d(e) = d(e') - 1$.

3 $e^* = (f_0, f_1)$ má dva následníky $e'^* = (f_1, f_2)$, a $e''^* = (f_1, f_3)$. Nechť $c(e)$ je cesta vzniklá zkrácením cest $c(e')$, $c(e'')$ o jejich společnou část p, a spojením vzniklých cest. Nechť $I(e) = I(e') + I(e'') + (\text{počet hran cesty } p)$, a $d(e) = d(e') + d(e'') - 2(\text{počet hran cesty } p)$.

Zbývá poslední lemma:

Lemma 6.4 *Existuje algoritmus, který v lineárním čase nalezne příčku e takovou, že*

$$I(e) \leq \frac{2}{3}n,$$

$$n - (I(e) + d(e)) \leq \frac{2}{3}n.$$

Důkaz: Protože pro hranu e_0^* vedoucí z r je $I(e_0) = n - 3$, $d(e_0) = 2$, algoritmus někdy poprvé narazí na hranu e^*, pro níž je $I(e)+d(e) \geq \frac{1}{3}n$. Rozborem případů, podle nichž bylo $I(e)$ počítáno, ukážeme, že $I(e) \leq \frac{2}{3}n$.

1 $I(e) = 0$.

2 a $I(e) = I(e') \leq I(e') + d(e') < \frac{1}{3}n$.

b Nemůže nastat protože $I(e)+d(e) = I(e')+d(e')$.

3 $I(e) = I(e') + I(e'') +$ (počet hran cesty p) $\leq I(e') +$ $+ I(e'') + d(e') + d(e'') < \frac{2}{3}n$.

□

7 Ne všechny úlohy je možno řešit v lineárním čase

Cílem této kapitoly je ukázka dokazování dolních odhadů. Nejprve ukážeme, dolní odhad pro jednu konkrétní úlohu. Později ukážeme, jak je možno pomocí „transformace“ tohoto odhadu použít i pro jinou úlohu. Nakonec ukážeme, jak nutné je přesně popsat naše „výpočetní možnosti“, aby dolní odhad odpovídal skutečnosti.

Stanovení dolního odhadu ukážeme na následující úloze: Máme n prvků, potřebujeme je seřadit. K řazení můžeme použít pouze dotazy typu „Je prvek x_i menší než x_j?“. Navíc předpokládáme tranzitivitu relace.

Ukážeme, že za takových předpokladů je k seřazení potřeba čas $\Omega(n \log n)$.

Věta 7.1 *Utřídění n prvků, založené na porovnávání vyžaduje v průměru čas $\Omega(n \log n)$. (Průměr, kde všechny permutace jsou stejně pravděpodobné.)*

Důkaz: Na začátku máme dány prvky v jednom z $n!$ možných pořadí. Ať dostaneme prvky v libovolném pořadí, musíme je převést do stejného tvaru. Pro každé pořadí (permutaci) musí algoritmus popřehazovat prvky jiným způsobem. Odhlédneme od konkrétní implementace algoritmu. Bude nás pouze zajímat způsob, jakým algoritmus jednotlivé permutace rozliší.

V každém stavu s algoritmu nás bude zajímat počet p_s permutací vyhovujících výsledkům dosavadních porovnání. Po každém porovnání může výpočet algoritmu pokračovat dvěma způsoby. Mohl-li se takto algoritmus dostat ze stavu s do stavů s_1, s_2, pak $p_s = p_{s_1} + p_{s_2}$. Tím jaké zvolíme porovnání vlastně volíme čísla p_{s_1}, p_{s_2}. V rozboru nebudeme volbu čísla p_{s_1} nijak omezovat, ačkoli v algoritmu volba čísla p_{s_1} závisí na permutacích odpovídajících stavu s.

Úlohu zjistit počet nutných porovnání můžeme pak přeformulovat takto: Máme rozlišit p_s jevů, každým porovnáním se nám jevy rozdělí do dvou (disjunktních) částí. Kolik potřebujeme porovnání?

Označme $a(k)$ průměrný počet porovnání algoritmu pro rozlišení k jevů. Označme $S(k)$ součet počtu porovnání přes jednotlivé jevy. Tedy $a(k) = \frac{S(k)}{k}$. Dostáváme rekurenci

$$S(k) \geq k + S(\ell) + S(k - \ell),$$

kde $\ell \in \langle 1, k-1 \rangle$ závisí na algoritmu. O $S(k)$ dokážeme indukcí, že $S(k) \geq k \log_2 k$.

$$S(m) \geq m + S(\ell) + S(m-\ell) \geq m + \ell \log_2 \ell + (m-\ell) \log_2 (m-\ell)$$

Tato funkce nabývá minima $m \log_2 m$ v bodě $\ell = \frac{m}{2}$. Závěr $a(k) = \frac{S(k)}{k} \geq \log_2 k$.

Co z toho plyne pro odhad počtu porovnání při třídění? Algoritmus na třídění provede v průměru aspoň $a(n!) = \log_2 n! \approx \log_2 \frac{n^n}{e^n} \approx n \log n$ porovnání. □

Cvičení 7.1 Prozkoumejte chování rekurence pro $S(k)$. **Návod:** Představte si „strom rekurence“. Tento binární strom má k listů, $S(k)$ je součet hloubek listů. Vyvažováním stromu se $S(k)$ snižuje.

Poznámka 7.1 Rozbor byl proveden tak, aby jej neovlivnilo použití náhodného generátoru.

Poznámka 7.2 Dolní odhad průměrného času je dolním odhadem nejhoršího času.

Za algoritmus s čistou aritmetikou budeme považovat algoritmus, nezávislý na přesnosti reprezentace čísel. (Považujeme čas aritmetické operace za nezávislý na přesnosti reprezentace čísla.)

Domněnka 7.2 *Rychlý třídící algoritmus s čistou aritmetikou musí být založen na porovnávání.*

Důsledek 7.2.1 *Nalézt konvexní obal množiny n bodů v rovině není možno algoritmem s čistou aritmetikou rychleji než v čase $\Omega(n \log n)$. (Konvexní obal je určen množinou hraničních úseček, ne pouze množinou hraničních bodů.)*

Důkaz: Důkaz provedeme transformací. Ukážeme, že bychom potom uměli utřídit n čísel aritmeticky čistým algoritmem v čase lepším než $\Omega(n \log n)$. Čísla $x_1, x_2, \ldots x_n$ bychom mohli utřídit tak, že bychom našli konvexní obal množiny bodů (x_1, x_1^2), (x_2, x_2^2), $\ldots (x_n, x_n^2)$. Protože x^2 je konvexní funkce, jsou všechny tyto body body konvexního obalu. Jejich pořadí (určené úsečkami konvexního obalu), určuje pořadí prvních souřadnic. Kdybychom tedy uměli nalézt konvexní obal rychleji než $\Omega(n \log n)$, uměli bychom setřídit čísla rychleji než v čase $\Omega(n \log n)$. □

Poznámka 7.3 Dříve uvedený algoritmus pracující v čase $O(n \log h)$, kde h je počet bodů na konvexním obalu není ve sporu s předchozím tvrzením, protože h může nabývat hodnot až do n, a tedy jako funkce v proměnné n máme odhad $O(n \log n)$.

Pozastavme se nyní nad „umělým“ požadavkem čisté aritmetiky.

Pokud například čísla x_i reprezentujeme b bity, pak můžeme n čísel setřídit v čase $O(n \max\{1, \frac{b}{\log n}\})$.

(Třídíme v $O(\frac{b}{\log n})$ fázích. Třídíme postupně podle více a více významných úseků čísla x_i. Na každou fázi používáme stabilní algoritmus, proto práce předchozích fází „není promarněna“. Průběh jedné fáze připomíná „zahashovávání“ do tabulky velikosti n. (případně nějaká blízká mocnina 2, kvůli snazší implementaci). Rozdíl od hashování je v tom, že běžná hashovací funkce se snaží prvky „náhodně“ rozmístit v tabulce, zatímco v tomto případě funkce pouze „vyřezává“ pro fázi významný úsek čísla. „Konflikty“ v tabulce řešíme spojovými seznamy prvků patřících do daného políčka (zde je důležité zachování pořadí prvků). Po „zahashování“ prvků do tabulky fáze končí přečtením seznamů prvků v pořadí „rostoucí vyřezávací“ funkce. Jedna fáze trvá $O(n)$.)

Uvedený algoritmus se často používá k utřídění prvků, kde třídící „klíč“ není příliš velký (typicky čísla 0, 1, 2 nebo třeba $1, \ldots, n$).

Jak je z výše uvedeného vidět, dokazovat dolní odhady složitosti úlohy není jednoduché. Neohrabanost analýzy

nyní vyplývá z toho, že dosud nemáme matematizovaný skutečný počítač.

7.1 Rozhodovací d-stromy

Rozhodovací stromy jsou modelem výpočtu, popisující možná větvení programu. V tomto modelu předpokládáme, že každý vstup je prvkem nějakého Euklidovského prostoru (vstupy jsou tedy z $\bigcup \mathbb{E}^k$). Kokrétní algoritmus pro vstup dimenze k (označme $(x_1, \ldots, x_k)$ tento vstup) je konečný orientovaný strom. Hrany stromu jsou orientovány směrem od kořene. Pokud z vrcholu stromu vede nějaká hrana, potom je vrchol ohodnocen polynomem stupně nejvýš d v proměnných $(x_1, \ldots, x_k)$ a vedou z něj právě dvě hrany, označené $\leq$ resp. $>$. Listy (vrcholy z nichž nevede žádná hrana) jsou ohodnoceny *ANO* resp. *NE*.

Výpočet začíná v kořeni stromu. Vždy pokud se nejedná o list, je vyhodnocen příslušný polynom a je-li výsledek kladný, pokračuje výpočet vrcholem, do nějž vede hrana označená $>$, jinak pokračuje výpočet vrcholem, do nějž vede hrana označená $\leq$. Výpočet končí v listu. Výsledkem je ohodnocení tohoto listu.

Čas výpočtu je počet vrcholů stromu navštívených během výpočtu.

Algoritmus (bez omezení na konkrétní dimenzi) je posloupnost algoritmů pro jednotlivé dimenze.

Věta 7.3 *Nechť $A \subseteq \mathbb{E}^k$. Nechť se množina A rozpadá na a konvexních množin. Potom rozhodovací 1-strom přijímající množinu A má průměrnou hloubku aspoň $\log_2 a$.*

Důkaz: Stačí si uvědomit, že pokud pro některé dva body x, y končí výpočet ve stejném listu, končí výpočet v tomtéž listu pro celou úsečku (x, y). Proto má rozhodovací strom aspoň a listů. Nyní stačí použít funkce $S(k)$, $a(k)$ ze začátku kapitoly k omezení průměrné hloubky stromu. □

Poznámka 7.4 Obdobná věta (s omezením $\Omega(\log a)$) prý platí pro rozhodovací d-stromy při libvolném pevném d. Při důkazu je potřeba umět odhadnout počet konvexních oblastí které mohou „sdílet“ tentýž list rozhodovacího stromu.

Důsledek 7.3.1 *Libovolný rozhodovací 1-strom zjišťující, zda daných n čísel je po dvou různých musí mít hloubku aspoň $\Omega(n \log n)$.*

Důkaz: Označme $A \subseteq \mathbb{E}^n$ množinu přijímaných vstupů. Pro libovolný vstup $x = (x_1, \ldots, x_n)$, kde daná čísla x_i jsou po dvou různá je možno definovat permutaci σ_x tak, že $x_{\sigma_x(i)} < x_{\sigma_x(j)}$ právě když $i < j$. Pokud pro vstupy x, y je $\sigma(x) \neq \sigma(y)$, potom na úsečce (x, y) leží bod nepatřící do množiny A (např. supremum z bodů kde $\sigma = \sigma(x)$). Pokud pro vstupy x, y je $\sigma(x) = \sigma(y)$, potom pro libovolný bod z úsečky (x, y) je $\sigma(z) = \sigma(x)$ a z patří do A. Množina A se rozpadá na $n!$ konvexních komponent. □

Poznámka 7.5 Uvědomme si, že model rozhodovacích stromů nepostihuje nepřímou adresaci.

7.2 Další výpočetní modely

Výpočetní model s „libovolnou adresací“ běžně označovaný **RAM** umožňuje „hashování“ do libovolně veliké tabulky. Někdy se uvažuje, že „buňkou“ paměti modelu je reálné číslo. V takovém případě přesné algoritmy musí mít čistou aritmetiku. Více se skutečnosti blíží model, kde velikost „buňky“ paměti je konstantní. To by však omezilo dosažitelný prostor na konstantu. Lepší model umožňuje reprezentovat celá čísla, ale čas operace není konstantní, ale závisí na operaci a na velikosti použitých čísel (nejlépe třetí odmocnina adresy plus polynom logaritmu čísla).

Jiným výpočetním modelem je „ukazatelový stroj“, nebo raději anglicky „pointer machine“ či zkráceně **PM**. V tomto modelu jsou pro adresaci důležité ukazatele. Větší část paměti je přístupná pouze pomocí ukazatelů. Ukazatel ukazuje vždy na konstantně velký „naalokovaný“ kus paměti. V tomto kusu paměti mohou být mimo jiné opět ukazatele.

Poněkud abstraktnějším pohledem na skutečný počítač je pohled jako na konečný automat (matematicky téměř přesné, ale počet stavů mnohonásobně převyšuje jakékoli číslo z vesmíru), nebo turingův stroj. (Nepřesnost pohledu na počítač jako na konečný automat je způsobena „nepatrností“ úloh řešených v reálném vesmíru, ale také tím, že k počítači můžeme „v průběhu výpočtu“ přidávat další a další paměť ...) Turingův stroj je velmi jednoduše zmatematizovaný počítač, který je pouze „polynomiálně“ pomalejší než **RAM**.

Pokud bychom chtěli pracovat s „přesným“ modelem počítače, museli bychom rozlišovat několik úrovní rychlosti paměti, bylo by důležité, kolik kterého druhu paměti máme. Rozbory by byly náročné, technologické inovace by měly výrazný vliv na rozbor.

Zatím jsem neuvedl paralelní výpočetní modely. Jenom poznamenám horní mez, na kterou se v odhadech reálných počítačů často zapomíná. V čase t se informace rozšíří nejvýš do koule o poloměru ct (c je rychlost světla). Proto se například v reálném paralelním počítači v čase t informace nemůže rozšířit do $O(2^t)$, ale pouze do $O(t^3)$ procesorů. Obdobně může být v čase t dosažitelná pouze některá z $O(t^3)$ buněk paměti. (Proto by měla být do času operace nad RAM zahrnuta třetí odmocnina adresy. Obdobně by měla být omezena šířka stromu modelu PM.)

8 Kleenovy algebry

Cílem této kapitoly je ukázka zobecňování algoritmů. Toto zobecňování si ukážeme na problematice hledání tranzitivního uzávěru grafu a hledání nejlevnějších cest(sledů) v grafu. Rozborem těchto algoritmů a následným zobecněním získáme algoritmus a typ objektů na kterých algoritmus funguje. Tímto typem objektů jsou Kleenovy algebry.

Tranzitivním uzávěrem orientovaného grafu G, který máme zadán maticí sousednosti E, rozumíme matici E^*, pro kterou platí: Pro každé $i, j = 1 \ldots n$, kde n je počet vrcholů grafu G popsaného maticí sousednosti $E = (a_{i,j})$, a pro každý prvek $a^*_{i,j}$ matice E^* platí:

- $a^*_{i,j} = 1$ právě tehdy, když existuje cesta v grafu G z vrcholu i do vrcholu j.
- $a^*_{i,j} = 0$ právě tehdy, když neexistuje cesta v grafu G z vrcholu i do vrcholu j.

Poznámka 8.1 Správně bychom měli psát reflexivně tranzitivní uzávěr. Tranzitivní uzávěr (neuzavřený reflexivně) bychom získali jako součin $E \times E^*$.

Řešením druhé úlohy, při které hledáme nejlevnější sledy v orientovaném grafu G, rozumíme matici E^*, pro kterou platí: Pro každé $i, j = 1 \ldots n$, kde n je počet vrcholů grafu G popsaného maticí cen $E = (a_{i,j})$, a pro každý prvek $a^*_{i,j}$ matice E^* platí:

- $a^*_{i,j} = k, k \in \mathbb{R}^+_0$ právě tehdy, když existuje alespoň jeden sled v grafu G z vrcholu i do vrcholu j a k je ohodnocení nejlevnějšího z těchto sledů.
- $a^*_{i,j} = +\infty$ právě tehdy, když neexistuje cesta v grafu G z vrcholu i do vrcholu j.

Vraťme se nyní zpět k tranzitivnímu uzávěru grafu.

Uvědomme si nejdříve, jak spolu souvisí náš problém a násobení matic.

Nechť $A = (a_{m,l})$, kde $m \in M$ a $l \in L$ a $B = (b_{l,k})$, kde $l \in L$ a $k \in K$ jsou dvě matice. $C = A \times \overset{\odot}{\oplus} B$ — součin matic je definován jako matice $C = (c_{m,k})$, kde $m \in M$ a $k \in K$ a

$$c_{m,k} = \bigoplus_{l \in L} a_{m,l} \odot b_{l,k},$$

kde $\oplus$ je operace sčítání a $\odot$ je operace násobení.

Předpokládejme, že máme tři neprázdné množiny K, L, M vrcholů a hrany mezi vrcholy množin M, L resp. L, K. Tážeme se, zda existuje cesta délky 2 z vrcholu množiny M do vrcholu množiny K přes vrcholy množiny L (zkráceně $M \to L \to K$).

Nechť matice $A = (a_{m,l})$, kde $m \in M$ a $l \in L$ popisuje všechny hrany vedoucí z vrcholů množiny M do vrcholů množiny L, a matice $B = (b_{l,k})$, kde $l \in L$ a $k \in K$, popisuje všechny hrany vedoucí z vrcholů množiny L do vrcholů množiny K.

Pak matice $C = A \times \overset{\wedge}{\vee} B, C = (c_{m,k})$ popisuje všechny cesty délky 2: $M \to L \to K$, kde $\oplus := \vee$ značí logický součet a $\odot := \wedge$ značí logický součin. (Slovem popisuje myslíme je maticí sousednosti.)

Poznámka 8.2 Podle klasické definice násobení, kde $\oplus := +$ a $\odot := \cdot$ bychom spočítali počet cest délky 2:$M \to L \to K$.

Matice $E^2 = E \times E$ definuje na množině V vrcholů grafu G nový graf. Kde každé hraně odpovídá cesta délky 2 původního grafu. Můžeme nyní vzít matici sousednosti (E^2) nového grafu a vynásobit ji maticí sousednosti E původního grafu. Dostaneme tak matici E^3 sousednosti dalšího grafu, kde každé hraně odpovídá cesta délky $(2 + 1)$ původního grafu $\ldots$, E^k je matice sousednosti grafu, kde každé hraně odpovídá cesta délky k původního grafu.

Označíme-li I matici popisující všechny cesty délky 0 v grafu G (jedná se o jednotkovou matici), pak matice $(I \vee \vee E)^k, k \in N$ popisuje všechny cesty délky nejvýše k v grafu G.

Jestliže z vrcholu u vede do vrcholu v cesta, pak existuje taková cesta délky nejvýše $n - 1$. Odtud $\forall k \geq n - 1$, $(I \vee \vee E)^k = (I \vee E)^{n-1}$. Tranzitivním uzávěrem grafu G je tedy matice $E^* = (I \vee E)^{n-1}$.

Pro čas $T(n)$ potřebný takto pro výpočet tranzitivního uzávěru grafu G tedy platí:

$$T(n) = M(n) \log n,$$

kde $M(n)$ je čas potřebný k vynásobení dvou boolovských matic typu $n \times n$, kde n je počet vrcholů grafu G.

Existuje i efektivnější algoritmus konstruující tranzitivní uzávěr v čase $M(n)$. Tento algoritmus je založen na metodě DIVIDE & IMPERA. Vrcholy orientovaného grafu G rozdělíme do dvou přibližně stejně velkých množin K, L. Na základě množin K, L můžeme matici sousednosti popisující graf G psát ve tvaru:

$$E = \begin{pmatrix} A & B \\ C & D \end{pmatrix},$$

kde A, matice typu $K \times K$, je matice sousednosti podgrafu grafu G obsahujícího vrcholy z množiny K a hrany spojující vrcholy z K, a B, matice typu $K \times L$, je matice sousednosti podgrafu grafu G obsahujícího všechny hrany vedoucí z vrcholů množiny K do vrcholů množiny L, a C, matice typu $L \times K$, je matice sousednosti podgrafu grafu G obsahujícího všechny hrany vedoucí z vrcholů množiny L do vrcholů množiny K, a D, matice typu $L \times L$, je matice sousednosti podgrafu grafu G obsahujícího vrcholy z množiny L a hrany spojující vrcholy z L.

Zamysleme se nyní, jaké cesty přicházejí v úvahu v grafu G. Mohou nastat tyto čtyři možnosti:

1. $K \to K$ Tato varianta zahrnuje všechny cesty v grafu G, které vedou z vrcholů množiny K do vrcholů množiny K přes vrcholy množin K a L. Označme F matici, která popisuje všechny takové cesty v grafu G, kde mezi hranami cesty je jen jediná vycházející z vrcholu z K.

   Pro F platí:

   $$F = (A \vee B \times D^* \times C),$$

$$E^* = \begin{pmatrix} F^* & F^* \times B \times D^* \\ D^* \times C \times F^* & D^* \vee D^* \times C \times F^* \times B \times D^* \end{pmatrix}$$

Obr. 13: Výpočet tranzitivního uzávěru

Tranzitivní uzávěr F^*, popisující všechny cesty $K \to \to K$, můžeme psát ve tvaru

$$F^* = (A \vee B \times D^* \times C)^*.$$

2. $K \to L$ Tato varianta zahrnuje všechny cesty v grafu G, které vedou z vrcholů množiny K do vrcholů množiny L přes vrcholy množin K a L. Tranzitivní uzávěr, popisující všechny tyto cesty, můžeme vyjádřit maticí

$$F^* \times B \times D^*.$$

(B zde reprezentuje poslední hranu vedoucí z K do L.)

3. $L \to K$ Tato varianta zahrnuje všechny cesty v grafu G, které vedou z vrcholů množiny L do vrcholů množiny K přes vrcholy množin K a L. Tranzitivní uzávěr, popisující všechny tyto cesty, můžeme vyjádřit maticí

$$D^* \times C \times F^*.$$

(C zde reprezentuje první hranu vedoucí z L do K.)

4. $L \to L$ Tato varianta zahrnuje všechny cesty v grafu G, které vedou z vrcholů množiny L do vrcholů množiny L přes vrcholy množin K a L. Tranzitivní uzávěr, popisující všechny cesty $L \to L$, můžeme psát ve tvaru

$$(D^* \vee D^* \times C \times F^* \times B \times D^*).$$

(cesta buď hranu z L do K neobsahuje, zde nebo C reprezentuje první takovou hranu, a B reprezentuje poslední hranu z K do L.)

Tranzitivní uzávěr grafu G pak můžeme vyjádřit maticí E^*, viz obr 13.

Jaká je časová náročnost výpočtu tranzitivního uzávěru grafu touto metodou?

$$T(n) = 2T\left(\frac{n}{2}\right) + 2\left(\frac{n}{2}\right)^2 + 10M\left(\frac{n}{2}\right) = \\ = 2T\left(\frac{n}{2}\right) + O(M(n)),$$

kde $M(n)$ je čas potřebný na vynásobení čtvercových boolovských matic typu $n \times n$, $n \in \mathbb{N}$.

Víme, že $M(n) = \Omega(n^2)$. Věta 2.1 říká, že potom $T(n) = O(M(n))$. Ukázali jsme tedy, že existuje algoritmus, který počítá tranzitivní uzávěr stejně rychle jako vynásobení boolovských matic.

Poznámka 8.3

$$\begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}^* = \begin{pmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix},$$

tedy kdybychom uměli počítat tranzitivní uzávěr rychleji, uměli bychom také rychleji násobit boolovské matice. Optimální čas na výpočet tranzitivního uzávěru je tedy stejný jako čas na vynásobení boolovských matic.

Přejděme nyní k druhému tématu této přednášky, kterým je nalezení nejlevnějších cest v grafu. Zopakujme si ze začátku této přednášky, že cílem je nalézt pro orientovaný graf G popsaný maticí cen $E = (a_{i,j})$ matici $E^* = (a^*_{i,j})$, kde

$a^*_{i,j} = k \in \mathbb{R}^+_0$ právě tehdy, když existuje alespoň jedna cesta v grafu G z vrcholu i do vrcholu j a k je ohodnocení nejlevnější z těchto cest.

$a^*_{i,j} = \infty$ právě tehdy, když neexistuje cesta v grafu G z vrcholu i do vrcholu j.

Řešení budeme hledat stejně jako v případě tranzitivního uzávěru relace pomocí násobení matic. Stejně jako v prvním případě předpokládejme, že máme tři neprázdné množiny K, L, M vrcholů a hrany mezi vrcholy množin M, L resp. L, K. Tážeme se, zda existuje cesta délky 2 z vrcholu množiny M do vrcholu množiny K přes vrcholy množiny L (zkráceně $M \to L \to K$).

Nechť matice $A = (a_{m,l})$, kde $m \in M$ a $l \in L$ popisuje všechny hrany vedoucí z vrcholů množiny M do vrcholů množiny L, s jejich ohodnoceními, a matice $B = (b_{l,k})$, kde $l \in L$ a $k \in K$, popisuje všechny hrany vedoucí z vrcholů množiny L do vrcholů množiny K, s jejich ohodnoceními.

Pak matice $C = A \times^+_{\min} B, C = (c_{m,k})$ popisuje všechny cesty délky 2: $M \to L \to K$, kde $\oplus := \min$ vybere minimální cenu a $\odot := +$ sečte dvě ceny. (Slovem popisuje myslíme je maticí cen.)

Označíme-li I matici popisující všechny cesty s jejich ohodnoceními délky 0 v grafu G tj.

$$I = \begin{pmatrix} 0 & \infty & \cdots & \cdots & \infty \\ \infty & 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 0 & \infty \\ \infty & \cdots & \cdots & \infty & 0 \end{pmatrix}$$

pak matice $(I \oplus E)^k, k \in N$ popisuje nejlevnější cesty délky nejvýše k v grafu G.

E^* můžeme spočítat jako $E^* = \lim_{k \to \infty} (I + E)^k$.

Cvičení 8.1 Dokažte, že v nezáporně ohodnoceném grafu je $E^* = (I \oplus E)^{n-1}$

Cvičení 8.2 Ukažte, že pro libovolné $M, N > 0$ konstantnost matice $(I \oplus E)^k$ pro $k \in \langle M, N \rangle$ nezaručuje rovnost $E^* = (I \oplus E)^M$, pokud v matici E byla i záporná čísla.

(1)	Operace $\oplus$:	$(a \oplus b) \oplus c$	$=$	$a \oplus (b \oplus c)$	(asociativita)
(2)		$a \oplus b$	$=$	$b \oplus a$	(komutativita)
(3)		$a \oplus a$	$=$	a	(idempotence)
(4)		$a \oplus o =$	a	$= o \oplus a$	(nulový prvek)
(5)	Operace $\odot$:	$(a \odot b) \odot c$	$=$	$a \odot (b \odot c)$	(asociativita)
(6)		$a \odot i =$	a	$= i \odot a$	(jednotkový prvek)
(7)		$a \odot o =$	o	$= o \odot a$	(nulový prvek)
(8)	Operace$\oplus$, $\odot$:	$a \odot (b \oplus c)$	$=$	$a \odot b \oplus a \odot c$	(distributivita zleva)
(9)		$(b \oplus c) \odot a$	$=$	$b \odot a \oplus c \odot a$	(distributivita zprava)

Operace $\oplus$ definuje přirozeným způsobem uspořádání $a \,\Delta\, b := (a \oplus b = b)$.

(10) Operace $*$: $a \odot b^* \odot c = \sup^{\Delta}_{n \geq 0} a \odot b^n \odot c =$ „$\sum_{n \geq 0} a b^n c$“

přičemž $b^0 = i$ a $b^{k+1} = b \odot b^k$.

(Držíme se konvence, že $\odot$ má větší prioritu než $\oplus$.)

Obr. 14: Axiomy Kleenovy algebry

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^* = \begin{pmatrix} F^* & F^* \times B \times D^* \\ D^* \times C \times F^* & D^* \oplus D^* \times C \times F^* \times B \times D^* \end{pmatrix},$$

kde $F = A \oplus B \times D^* \times C$.

Obr. 15: Výpočet E^* pro matici E

Předcházející cvičení naznačuje, že řešit takto úlohu pro graf v němž jsou i záporné ceny je obtížné.

Úlohu ale můžeme řešit podobně jako u tranzitivního uzávěru relace tak, že rozdělíme vrcholy grafu do přibližně dvou stejných množin K a L a aplikujeme postup analogický postupu při řešení první úlohy, přičemž operace $\oplus$ a $\odot$ mají výše uvedený význam.

Nyní si již definujeme **Kleenovy algebry**.

Definice 8.1 Šestice $(S, \oplus, \odot, *, o, i)$, kde S je množina, $\oplus$ a $\odot$ jsou binární operace, $*$ je unární operace a o, i jsou konstanty (nulární operace), je *Kleenova algebra*, pokud je splněno 10 axiomů viz obr. 14:

Příklady Kleenových algeber:

1. Algebra $(\{0,1\}, \vee, \wedge, 1, 0, 1)$. $(S := \{0,1\}, a \oplus b := a \vee b, a \odot b := a \wedge b, a^* := 1, o := 0, i := 1)$

2. Kleenova $(\min, +)$ algebra $(\mathbb{R}_+ \cup \{\infty\}, \min, +, 0, \infty, 0)$. $(S := \mathbb{R}_+ \cup \{\infty\}, a \oplus b := \min\{a, b\}, a \odot b := a + b, a^* := 0, o := \infty, i := 0)$

3. $(\min, +)$ algebra $(\mathbb{R} \cup \{-\infty, \infty\}, \min, +, *, \infty, 0)$, kde $a^* := \begin{cases} \infty & a = \infty \\ 0 & a \geq 0 \\ -\infty & a < 0 \end{cases}$. $(S := \mathbb{R} \cup \{-\infty, \infty\}, a \oplus b := \min\{a, b\}, a \odot b := a + b, o := \infty, i := 0$, při sčítání má ∞ 'větší váhu' než $-\infty)$

Věta 8.1 *Je-li šestice $(S, \oplus, \odot, *, o, i)$ Kleenova algebra, pak šestice $(S^{n \times n}, \oplus^{n \times n}, \times^{\odot}_{\oplus}, *^{n \times n}, O^{n \times n}_o, I^{n \times n}_{o,i})$, kde $S^{n \times n}$ jsou matice nad S, řádu $n \times n$, $A \oplus^{n \times n} B$ je sčítání po složkách, $A \times^{\odot}_{\oplus} B$ je násobení matic, $A^{*^{n \times n}}$ je definováno rekurzivně viz obr. 15,*

$$O^{n \times n}_o = \begin{pmatrix} o & \cdots & o \\ \vdots & \ddots & \vdots \\ o & \cdots & o \end{pmatrix},$$

je nulová matice řádu $n \times n$,

$$I^{n \times n}_{o,i} = \begin{pmatrix} i & o & \cdots & \cdots & o \\ o & i & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & i & o \\ o & \cdots & \cdots & o & i \end{pmatrix},$$

je jednotková matice řádu $n \times n$,

je také Kleenova algebra.

Důkaz *–8.1*: Neuvádíme. □

Dodejme ještě, že spočítat A^* umíme řádově stejně rychle jako vynásobit matice nad Kleenovou algebrou. Uvědomme si, ale, že neexistuje inverzní operace k operaci $\oplus$, nemůžeme tedy matice násobit podle rychlých algoritmů na násobení matic nad okruhy. (Strassenův algoritmus například odčítání silně využívá.) S našimi znalostmi tedy nejsme obecně schopni A^* spočítat rychleji než v čase $O(n^3)$. Výjimkou je počítání tranzitivního uzávěru, kde pro vynásobení boolovských matic můžeme použít klasické násobení nad okruhem $(\mathbb{Z}, +, \cdot)$. Vyjdeme z matic, s jedničkami na místě true a nulami na místě false. Matice vynásobíme. Tím získáme matici, jejímž nulovým hodnotám odpovídá false a nenulovým true. Na přepis do základního tvaru nám stačí čas $O(n^2)$.

Další příklady Kleenových algeber:

M1 $(\{0,1\}^{n\times n}, \vee^{n\times n}, \times^{\wedge}_{\vee}, *^{n\times n}, O_0, I_{0,1})$ je maticová Kleenova algebra nad algebrou viz příklad 1. Je-li E matice sousednosti grafu na n vrcholech, pak tranzitivní uzávěr grafu je E^*, v této maticové Kleenově algebře.

M2 $((\mathbb{R}_+ \cup \{\infty\})^{n\times n}, \min^{n\times n}, \times^{+}_{\min}, *^{n\times n}, O_\infty, I_{\infty,0})$ je maticová Kleenova algebra nad $(\min, +)$ algebrou viz příklad 2. Je-li A matice nezáporných ohodnocení hran grafu na n vrcholech, pak matice nejlevnějších cest je A^*, v této maticové Kleenově algebře.

M3 $((\mathbb{R} \cup \{-\infty, \infty\})^{n\times n}, \min^{n\times n}, \times^{+}_{\min}, *^{n\times n}, O_\infty, I_{\infty,0})$ je maticová Kleenova algebra nad algebrou viz příklad 3. Je-li A matice reálných ohodnocení hran grafu na n vrcholech, pak matice nejlevnějších cest je A^*, v této maticové Kleenově algebře.

9 Hladový algoritmus a matroidy

Další užitečnou metodou návrhu algoritmů je tzv. hladový algoritmus. Zhruba řečeno, jedná se o inkrementální algoritmus, kdy se průběžné řešení aktualizuje výběrem „nejlacinějšího, nejbližšího,...“ prvku dané datové struktury. Popíšeme si kombinatorickou strukturu, na které tato metoda nalezne optimální řešení. Poznamenejme, že hladový algoritmus je užitečný i v dalších případech, kdy rychle nalezne přibližná řešení, anebo problém řeší efektivně z hlediska průměrného případu.

Pro následující výklad si zavedeme tato označení: (S, F), kde S je konečná množina, $F \subseteq \mathcal{P}(S)$ je množina podmnožin S, $w : S \to \mathbb{R}_0^+$ je váhová funkce. Máme za úkol nalézt maximum $\{w(X) = \sum_{e \in X} w(e) \mid X \in F\}$. Duální úlohou je nalézt minimum $\{w(X) \mid X \notin F\}$.

K řešení použijeme hladový algoritmus, který postupně indukcí zkonstruuje množinu X.

Hladový algoritmus
Po řadě vybíráme prvky $e_1, \ldots, e_n$:

1. Nechť $\tilde{F} = \{X \mid \exists Y \in F, X \subseteq Y\}$ je rozšíření F o podmnožiny množin obsažených v F.
2. $E_0 = \emptyset$.
3. Množina prvků z nichž můžeme v k-tém kroku vybírat je $M_k = \{e \notin E_{k-1} \mid E_{k-1} \cup \{e\} \in \tilde{F}\}$. $k-$tý vybraný prvek je $e_k \in M_k$, kde $w(e_k) = \max_{e \in M_k} w(e)$. Množina prvků vybraných v prvních k krocích je $E_k = E_{k-1} \cup \cup \{e_k\}$.
4. Algoritmus končí s množinou $X = E_n$, právě když $M_{n+1} = \emptyset$.

Algoritmus hledá maximum z $\{w(X) \mid X \in \tilde{F}\}$. Vzhledem k tomu, že hledáme maximum ze součtu nezáporných čísel, maximum se nabývá na v inkluzi maximálních množinách, tedy na množinách z F.

Později ukážeme barvící algoritmus, což je zobecnění hladového algoritmu, nakonec si probereme uvedený algoritmus na konkrétním příkladu. Budeme hledat minimální kostru grafu.

Nyní se vraťme k hladovému algoritmu. Z předpokladu, že hladový algoritmus nalezne maximum, ať zvolíme váhy $w(e)$ jakkoli, můžeme dokázat následující vlastnost množiny $\tilde{F}$:

$$(*)\quad \forall X, Y \in \tilde{F}(|X| > |Y|) \Rightarrow \exists\, x \in X \setminus Y : Y \cup \{x\} \in \tilde{F}$$

Důkaz: Zvolme váhovou funkci w pro vhodné ε následovně:

$$w(e) = \begin{cases} 1+\varepsilon, & e \in Y \\ 1 & e \in X \setminus Y \\ 0 & \text{jinak.} \end{cases}$$

Potom $\max\{w(X) \mid X \in F\} \geq |X| > (1+\varepsilon)|Y|$. Tedy hladový algoritmus začne v Y, ale musí přibrat alespoň 1 prvek z $X \setminus Y$. □

Vlastnost $(*)$ včetně uzavřenosti systému množin na podmnožiny charakterizuje matroidy.

Definice 9.1 Matroid je dvojice (S, F), kde S je konečná množina, $F \subseteq \mathcal{P}(S)$ a platí:

1. Dědičnost: $Y \in F, X \subseteq Y \Rightarrow X \in F$
2. $X, Y \in F, |X| > |Y| \Rightarrow \exists\, x \in X \setminus Y$ tak, že $Y \cup \{x\} \in F$

Prvky F se nazývají *nezávislé* množiny. Podmnožiny z S, které nejsou z F, se nazývají závislé množiny.

Již jsme ukázali, že hladový algoritmus nalezne maximum při libovolných vahách $w(e)$, pouze v případě, že $(S, \tilde{F})$ je matroid.

Poté, co se blíže seznámíme s matroidy, ukážeme, že tato podmínka je dostatečná.

Uveďme si několik příkladů matroidů:

1. Nechť V je vektorový prostor, $S \subseteq V$ konečná množina vektorů, F jsou lineárně nezávislé podmnožiny S. Pak (S, F) je matroid.
2. Nechť $G = (V, E)$ je graf, F obsahuje takové množiny hran $E' \subseteq E$, pro které je (V, E') les. Pak (E, F) je matroid.
3. Nechť $G = (V, E)$ je graf, F obsahuje takové množiny hran $E' \subseteq E$, pro které má graf $(V, E \setminus E')$ stejný počet komponent, jako graf (V, E). Pak (E, F) je matroid.

Zadání matroidu se zjednoduší, zadáme-li pouze maximální (v inkluzi) nezávislé množiny. Nechť

$$MF = \{X \in F \mid \forall Y \in F(X \subseteq Y \Rightarrow X = Y)\}.$$

Dědičnost zaručuje, že $F = \{X \mid \exists Y \in MF, X \subseteq Y\}$. Druhá vlastnost matroidů zaručuje $X, Y \in MF \Rightarrow |X| = |Y| = |\text{maxF}|$.

Definice 9.2 Duál (S, F^*) k matroidu (S, F) je matroid jehož maximální nezávislé množiny MF^* vyhovují předpisu $MF^* = \{X \mid S \setminus X \in MF\}$.

Důkaz *–(Toho, že MF^* definuje matroid)*: Dědičnost F^* plyne z konstrukce z MF^*. Je třeba dokázat druhou vlastnost matroidu.

Nechť $X, Y \in F^*$, nechť $|X| > |Y|$. Z definice F^* pomocí MF^* máme $\exists X', Y' \in MF^*, X \subseteq X', Y \subseteq Y'$. Z definice MF^* máme $X'' = S \setminus X' \in MF$ a $Y'' = S \setminus Y' \in MF$. Z dědičnosti F máme $Y'' \setminus X \in F$. Podle druhé vlastnosti matroidu (S, F) můžeme $X'' \setminus Y$ doplnit prvky množiny Y'' na nějaké $Z \in MF$.

Z konstrukce množiny Z plyne $Z \cap Y = \emptyset$ a $Z \supseteq X'' \setminus Y$. Dále $|Z \cup Y| = |Z| + |Y| = |\text{maxF}| + |Y| < |\text{maxF}| + |X| = |X''| + |X| = |X'' \cup X|$. Proto $\exists x \in (X'' \cup X) \setminus (Z \cup Y)$. Ale $(X'' \cup X) \setminus (Z \cup Y) = \big(X'' \setminus (Z \cup Y)\big) \cup \big(X \setminus (Z \cup Y)\big) = \emptyset \cup \cup X \setminus (Z \cup Y) \subseteq X \setminus Y$. A nakonec $Y \cup \{x\} \subseteq S \setminus Z \in MF^*$. Takže x má požadované vlastnosti. □

Poznámka 9.1 Ihned je vidět, že $MF^{**} = MF$, tedy $F^{**} = F$.

Poznámka 9.2 Příklady 2 a 3 jsou navzájem duální matroidy.

U matroidů zavádíme podobné pojmy jako u grafů. Cyklus je minimální (ve smyslu inkluze) závislá množina, řezem označujeme minimální (ve smyslu inkluze) podmnožinu S, která má neprázdný průnik se všemi maximálními (ve smyslu inkluze) nezávislými množinami.

Poznámka 9.3 Cyklus je řez v duálním matroidu.

Cvičení 9.1 Nalezněte matroid, který se neliší od svého duálu.

Nechť (S, F_1) a (S, F_2) jsou navzájem duální matroidy.

Definice 9.3

Obarvení je dvojice (B, R), kde $B \in F_1$, $R \in F_2$ a množiny B a R jsou disjunktní ($B \cap R = \emptyset$).

Totální obarvení je obarvení, kde $B \cup R = S$. Neboli $B \in \in MF_1$ a $R \in MF_2$. (K zadání totálního obarvení stačí jedna z množin B nebo R.)

Obarvení (B', R') je *rozšíření obarvení* (B, R), pokud platí $B \subseteq B', R \subseteq R'$.

Mají-li prvky S určeny váhy, pak definujeme *Optimální obarvení*. Je to totální obarvení, kde B je minimální váhy. (Zároveň je R maximální váhy.)

Platí následující lemma:

Lemma 9.1 *Každé obarvení lze rozšířit na totální.*

Důkaz: Vyjdeme z obarvení (B, R). Nechť $B \subseteq U \in MF_1$ Položme $V = S \setminus U \in MF_2$. R je možno doplnit prvky V na $W \in MF_2$.

Z konstrukce W víme, že $R \subseteq W$, $W \subseteq R \cup (S \setminus U)$, tedy $B \cap W = \emptyset$. Proto je $(S \setminus W, W)$ příkladem totálního rozšíření. □

Lemma 9.2 *Řez a cyklus matroidu se nemohou protínat právě v jednom prvku.*

Důkaz: Sporem:

Nechť C je cyklus, D řez matroidu (S, F), předpokládáme pro spor $C \cap D = \{x\}$. Množiny $C \setminus \{x\}, D \setminus \{x\}$ jsou disjunktní množiny. $C \setminus \{x\} \in F$, $D \setminus \{x\} \in F^*$.

Jinými slovy $(C \setminus \{x\}, D \setminus \{x\})$ je obarvení. Nechť totální obarvení (C', D') je rozšířením $(C \setminus \{x\}, D \setminus \{x\})$. Pak je buď $C \subseteq C'$ nebo $D \subseteq D'$ (v závislosti na barvě x). To však není možné, neboť $C \notin F$ a $D \notin F^*$, a to je požadovaný spor. □

Značení 9.1 Nechť $B \in MF$, $x \notin B$. Pak existuje (právě jeden) tzv. *fundamentální* cyklus $C_{x,B}$ takový, že $C_{x,B} \subseteq B \cup \{x\}$.

Cvičení 9.2 Dokažte jednoznačnost fundamentálního cyklu.

Návod: Nechť $C_1, C_2 \subseteq B \cup \{x\}$ jsou cykly. Pak $(C_1 \cup \cup C_2) \setminus \{x\} \in F$. Odtud

$$y \in C_1 \cup C_2 \Rightarrow (C_1 \cup C_2) \setminus \{y\} \in F \Rightarrow y \in C_1 \cap C_2.$$

Lemma 9.3 (O výměně barev) *Buď (B, R) totální obarvení, $x \in R$, $y \in C_{x,B}$, pak výměnou barev x a y získáme nové totální obarvení.*

Symetricky prohozením B a R (F_1 a F_2).

Důkaz *–9.3:* $C_{x,B} \setminus \{y\} \in F_1$. Množinu $C_{x,B} \setminus \{y\}$ rozšíříme přidáváním prvků z B až na $B' \in MF$, nutně $B' = (B \setminus \setminus \{y\}) \cup \{x\}$. Výměnou barev x a y získáme totální obarvení $(B', S \setminus B')$ □

Chceme-li počítat $\min\{w(X) \mid X \in MF\}$, můžeme postupovat dvěma způsoby:

1. Zavedeme nové váhy $w'(e) = \max\{w(e) \mid e \in S\} - - w(e)$. Nalezneme $\max\{w'(X) \mid X \in F\}$. Hledané $\min\{w(X) \mid X \in MF\} = |\text{maxF}| \cdot \max\{w(e) \mid e \in \in S\} - \max\{w'(X) \mid X \in F\}$. Optimum se nabývá na stejných množinách.

2. Nalezneme $\max\{w(X) \mid X \in F^*\}$. Potom hledané $\min\{w(X) \mid X \in MF\} = w(S) - \max\{w(X) \mid \mid X \in F^*\}$. Optimum se nabývá na doplňkových množinách.

Barvící algoritmus je libovolný algoritmus, který v každém kroku postupuje podle jednoho ze dvou následujících pravidel:

1. Z řezu v matroidu (S, F_1) neobsahujícím modrou barvu prvek s nejmenší vahou w obarvi modře. (Je to prvek s největší vahou w'.) (Pokud je více minim, preferuj prvek, který není červený.)

2. Z řezu v matroidu (S, F_2) neobsahujícím červenou barvu prvek s největší vahou w obarvi červeně. (Pokud je více maxim, preferuj prvek, který není modrý.)

Algoritmus končí pokud je množina B modře obarvených prvků maximální ($B \in MF_1$), nebo pokud je maximální množina R červeně obarvených prvků ($R \in MF_2$). (Neobarvené prvky spolu s druhou barvou tvoří maximální množinu v duálním matroidu.)

Poznámka 9.4 Je vidět, že prohozením $(F_1, w', \text{modrá})$ a $(F_2, w, \text{červená})$ dostaneme tentýž algoritmus. Proto stačí dělat důkazy pouze pro modré pravidlo, zbytek plyne z této duality.

Poznámka 9.5 V barvícím pravidlu můžeme místo řezu použít sjednocení řezů protože potom vybraný prvek určí řez, na nějž by bylo možno použít původní pravidlo.

Poznámka 9.6 Hladový algoritmus je speciálním případem barvícího algoritmu, kde $(S, F_2) = (S, \tilde{F})$. Hladový algoritmus používá pouze červeného pravidla, na sjednocení M_k řezů.

V dalším dokážeme, že barvící algoritmus nalezne optimální řešení nezávisle na způsobu aplikace barvících pravidel.

Lemma 9.4 *Barvící algoritmus každý prvek obarví nejvýš jednou.*

Důkaz: Nechť (B, R) je obarvení v kroku, kdy poprvé barvíme nějaký prvek podruhé. Ukážeme, že tento prvek nemůžeme obarvit modrou barvou:

Nechť A je řez v F_1 bez modré barvy, nechť se minima $\{w(e) \mid e \in A\}$ nabývají na množině $M \subseteq A$. $M \cap B \subseteq A \cap$ $\cap B = \emptyset$. (Žádný prvek z M není obarven modře.) Nechť $N = M \cap R$ jsou minimální červené prvky řezu. Pokud $N =$ $= \emptyset$, pak obarvíme nějaký prvek z M a v tomto kroku tedy žádný prvek podruhé neobarvíme. Pokud $N \neq \emptyset$, pak nechť y je poslední obarvený prvek z N. Nechť y byl obarven pravidlem aplikovaným na řez Z v F_2 bez červené barvy. Použijeme lemma o řezu a cyklu. Dostaneme $|Z \cap A| \neq 1$, tedy $\exists z \in (Z \cap A) z \neq y$. Navíc ale $w(z) \leq w(y)$, tedy $z \in M$. Protože y je poslední obarvený prvek N, platí $z \in M \setminus N$, algoritmus tedy obarví prvek z $M \setminus N$, a v tomto kroku tedy žádný prvek podruhé neobarvíme. □

Důsledek 9.4.1 *(B, R) je v průběhu algoritmu vždy obarvení.*

Lemma 9.5 *Buď (B, R) obarvení, které má rozšíření na optimální obarvení. Pak aplikací libovolného barvícího pravidla získáme obarvení, které má také optimální rozšíření.*

Důkaz: Nechť (B, R) je obarvení, které má optimální totální rozšíření.

Vzhledem k dualitě prozkoumáme aplikaci modrého pravidla. Nechť A je řez bez modrých prvků, a x je minimální prvek řezu A. Ukážeme, že existuje optimální obarvení (B', R'), které je rozšířením (B, R) a $x \in B'$.

Ukážeme, že předpoklad, že pro každé optimální rozšířené obarvení (B', R') obarvení (B, R) platí $x \notin B'$, vede ke sporu.

Potom by totiž $|A \cap C_{x,B'}| \neq 1$ použitím lemma o řezu a cyklu. Protože $x \in A \cap C_{x,B'}$, existuje tedy $y \in A \cap C_{x,B'}$. Využijeme lemma o výměně barev. $(B'', R'') = (\{x\} \cup B' \setminus$ $\setminus \{y\}, \{y\} \cup R' \setminus \{x\})$ je totální obarvení. Víme, že $w(x) \leq$ $\leq w(y)$, protože $y \in A$ a $w(x) = \min\{w(e) \mid e \in A\}$. Odtud (B'', R'') je optimální obarvení, kde $B \cup \{x\} \subseteq B''$. □

Lemma 9.6 *Buď (B, R) obarvení, $B \notin MF_1$. Pak existuje možnost použití modrého barvícího pravidla.*

Důkaz: Nechť (B, R) je obarvení, které má totální rozšíření (B', R'). Zvolme $x \in B' \setminus B$. $C_{x,R'}$ je cyklus v F_2, tedy řez v F_1. Protože $C_{x,R'} \setminus \{x\} \subseteq R'$, není žádný prvek $C_{x,R'}$ obarven modře v B. A tak lze aplikovat modré barvící pravidlo. □

Věta 9.7 *Aplikujeme-li barvící pravidlo v matroidu dokud to je možné, obdržíme optimální obarvení.*

Dokázali jsme, že matroidy jsou právě objekty, na nichž funguje hladový algoritmus. Ukázali jsme navíc, že na nich funguje zobecnění hladového algoritmu, barvící algoritmus.

Nyní přejděme k aplikaci matroidů, k hledání minimální kostry grafu.

9.1 Hledání minimální kostry

Nechť $G = (V, E)$ je graf s množinou vrcholů V a množinou hran E. Nechť w je váhová funkce hran, $w : E \to \mathbb{R}_0^+$. *Lesem* nazveme podgraf, který neobsahuje cyklus. Lehce odvodíme, že platí následující vztahy:

Je-li F les, m udává počet hran a c počet komponent souvislosti, pak pro počet vrcholů n platí $m + c = n$. Naším úkolem je nalezení minimální kostry grafu. Kostra je maximální les. Minimální kostra je ta, pro kterou platí, že má součet vah na hranách minimální mezi všemi kostrami.

V terminologii matroidů jsou kostry maximální množiny matroidu lesů.

Řez v teorii grafů odpovídá sjednocení řezů v tomto matroidu, jsou jím všechny hrany, které spojují podgrafy X a $V - X$. (Pokud nějaké existují).

Cyklus v tomto matroidu (řez v duálu) je kružnice teorie grafů. Barvící pravidla tedy můžeme v řeči teorie grafů přeformulovat následovně.

1. Vezmi řez, který neobsahuje modrou barvu, nejmenší hranu (ve smyslu váhy) obarvi modře. (Je-li více minim, preferuj neobarvené hrany.)

2. Vezmi kružnici v původním grafu, která neobsahuje červenou hranu, největší hranu této kružnice obarvi červeně. (je-li více maxim, preferuj neobarvené hrany.)

Přehled algoritmů pro hledání minimální kostry:
Většinou hledáme minimální kostru v souvislém grafu. Pokud tomu tak není, můžeme spustit algoritmus zvlášť na každé komponentě. Z uvedených algoritmů je toto potřeba provést pouze pro algoritmus 3.

1. **O. Borůvka, Choquet, Lukaszewicz, Sollin:**

   V průběhu jedné fáze výpočtu pro každý modrý strom (v průběžném lese) vybereme minimální incidentní hranu a obarvíme ji modře (vhodné pro paralelizaci).

   Pozor! Je třeba řešit případ nejednoznačnosti volby hrany. (Vzhledem k „paralelnímu“ zpracování by totiž jinak mohly vznikat kružnice.) Stačí dodat pravidlo, že při shodě velikostí vybereme hranu s nižším pořadovým číslem (hrany nějak očíslujeme).

2. **Kruskal:**

   Barvící pravidlo aplikujeme na vzestupně uspořádanou posloupnost hran podle vah. Jestliže hrana na řadě spojuje 2 modré stromy, obarvíme ji modře, spojuje-li 2 uzly v modrém stromu, obarvíme ji červeně.

   Proces je možno zastavit, je-li modře obarveno $n - c$ hran.

   Diskuse implementace algoritmu z hlediska časové náročnosti :

   – setřídění hran — $O(m \log n)$, druhá fáze — $O(m\alpha(m, n))$ je téměř lineární.

(V druhé fázi využijeme datovou strukturu udržující komponenty souvislosti grafu – umožňuje testovat, zda dva vrcholy leží v jedné komponentě, a přidávat hrany.)

Pozor! V některých speciálních případech je možno třídění hran provést v čase $O(m)$ (malá celá čísla...). V takovém případě je Kruskalův algoritmus asymptoticky nejlepší známý algoritmus pro obecné grafy.

3. **Jarník, Prim, Dijkstra:**

   Pro $i = 1 \dots c$:

   Zvolme libovolný vrchol z_i, pěstujeme jen modrý strom T, obsahující z_i. Nalezneme minimální hranu řezu incidentní k T a obarvíme ji modře.

   — implementace se setříděním - $O(m \log n)$,

   — implementace pomocí Fibonacciho hald - $O(m + n \log n)$:

   vždy do haldy přidáme sousedy s „vybraného" vrcholu, které dosud nejsou v stromu. (resp. upravíme hodnotu vrcholu s tak, aby tato hodnota byla minimem velikostí hran spojujících vrchol s k stromu). Příští „vybraný" vrchol je vrchol haldy s minimální hodnotou, označíme jej a odebereme z haldy. (V haldě evidujeme s každým vrcholem též hranu, která „hodnotu vrcholu zaručuje". To nám umožňuje nejen spočítat velikost minimální kostry, ale také kostru nalézt.)

   Po probrání i-té komponenty začínáme z nového vrcholu z_{i+1}.

4. **round–robin (Yao), Tarjan, Cheriton:**

   Zvolíme jeden modrý strom. Nalezneme minimální hranu řezu k němu incidentní a tu obarvíme modře.

   Uvažujeme frontu stromů, na začátku jednobodových. Každý strom je reprezentován haldou hran vycházejících z vrcholů stromu. Ke stromu ze začátku fronty nalezneme strom ve frontě, který je s ním spojený nejmenší hranou. Oba stromy z fronty vyjmeme (odlišnost od Borůvka ...), spojíme a zařadíme na konec fronty, atd. Nevýhodou je, že v haldách stromů jsou i hrany vedoucí uvnitř stromu. Abychom zjistili, který strom je spojený nejmenší hranou (a zda hrana nevede uvnitř stromu), musíme umět z koncového vrcholu hrany určit příslušný strom. K tomu slouží podpůrná struktura udržující komponenty grafu. Spojení stromů odpovídá spojení jejich hald. Implementace tohoto algoritmu pomocí „leftist trees" dosahuje času $O(m \log \log n)$.

   Vnitřní hrany stromů a vícenásobné hrany je možno odstraňovat „kondenzací" grafu „ve vhodnou dobu". Tímto je možno vylepšit čas algoritmu na $O(m \log \log_{(2+m/n)} n)$. — Toto byl nejrychlejší algoritmus na hledání minimální kostry v obecném grafu, do doby než byly objeveny Fibonacciho haldy.

5. **Fredman, Tarjan:**

   Brzy po objevení Fibonacciho hald (1984) byly tyto aplikovány i na problém minimální kostry. Mimo triviální aplikace viz algoritmus 3, byl navržen algoritmus kombinující růst stromu z jednoho vrcholu (viz algoritmus 3), s metodami pracujícími s větším počtem stromů.

   Označme n počet vrcholů a m počet hran grafu. Algoritmus pracuje v několika fázích. V každé fázi nejprve zvolíme konstantu $k_i = 2^{\frac{2m}{t_i}}$, na základě počtu stromů t_i vstupujícím do fáze. Označme m_i počet hran mezi těmito stromy. (Každá fáze končí odebráním vnitřních hran stromů a vybráním minimální z vícenásobných hran mezi stromy.) Fáze začíná jako algoritmus 3, vrcholy zařazené do stromu označujeme. (Vrcholy označujeme po zařazení do stromu, což odpovídá vyřazení uzlu z haldy!)

   Ve chvíli, kdy počet uzlů v haldě dosáhne k, vybereme nějaký neoznačený vrchol a začínáme algoritmus 3 s tímto novým vrcholem. Růst stromu končí při velikosti k haldy nebo ve chvíli, kdy bychom měli označit již označený vrchol. (V takovém případě se náš strom připojil k již dostatečně velkému stromu.)

   Vzhledem k tomu, že amortizovaně Fibonacciho haldy umožňují provádět operace Findmin, Insert, Decrement a Meld v čase $O(1)$ a operaci Delete v čase $O(\log k)$, kde k je počet vrcholů haldy, je celkový čas růstu stromů během i–té fáze $O(t_i \log k_i + 2m_i)$. (t_i krát je proveden Findmin+Deletemin, vždy na haldě velikosti nejvýš k_i, pro každou hranu jsou provedeny nejvýš dvě operace Insert resp. Decrement) k_i je voleno tak, aby $t_i \log k_i$ bylo $\Theta(m)$.

   Na závěr fáze je třeba odebrat vnitřní hrany stromů, a z vícenásobných hran mezi různými stromy vybrat hranu minimální. To je možno provést následovně: Očíslujeme stromy čísly $1, \dots, t_{i+1}$, každému vrcholu přiřadíme číslo příslušného stromu. Seřadíme hrany lexikograficky podle velikosti. (To je možno v čase $O(m_i)$, vzhledem k tomu, že čísla jsou přirozená a nejvýš t_{i+1}.) Poté projdeme takto setříděné hrany, a z každého úseku hran ležících mezi týmiž stromy vybereme minimální. To je možno provést v čase $O(m_i)$.

   Jedna fáze algoritmu tedy trvá $O(m)$. Na závěr časového rozboru je třeba zjistit počet fází. Vzhledem k tomu, že na konci i–té fáze vede z každého z t_{i+1} stromů aspoň k_i hran (velikost haldy), je $t_{i+1} \cdot k_i \leq \leq 2m_i$ (každá hrana má dva konce). Odtud $k_{i+1} = = 2^{2m/t_{i+1}} \geq 2^{k_i}$. Vzhledem k tomu, že $k_1 = 2^{2m/n}$, je $k_2 \geq 2^{2^{2m/n}}$, $k_3 \geq 2^{2^{2^{2m/n}}}$ .... Etapa, při níž je $k_i \geq t_i$ je poslední. Je-li $k_i \geq n$, potom je určitě nejvýš i etap. Je–li $\log^{(i)} n \leq 2m/n$, pak je etap nejvýš i. Označíme–li $\beta(m,n) = \min\{i \mid \log^{(i)} n \leq m/n\}$, pak čas celého algoritmu je $O(m\beta(m,n))$.

6. **Gabow, Galil a Spencer:** (1984)

   popsali metodu, pomocí níž je možno algoritmy pracující postupně s „minimálními hranami kondenzovaného

grafu“ v s fázích, kde jedna fáze trvá $O(m)$, urychlit z času $O(ms)$ na čas $O(m \log s)$.

Bez využití Fibonacciho hald tím dosáhli času $O(m \log\log\log_{(2+m/n)} n)$ (algoritmus 4). Použitím na algoritmus 5 dosáhli času $O(m \log \beta(m,n))$.

Jejich metoda je založena na tom, že hrany vycházející z jednoho „zkondenzovaného“ vrcholu jsou rozděleny do balíků velikosti s. Balíky jsou reprezentovány haldou. Hledáme–li minimální hranu vedoucí z vrcholu, stačí nalézt minimum z minim balíků. V podstatě se algoritmus chová, jako by měl m/s hran. Celkový čas (nepočítáme–li práci s haldami balíků) je potom $O(m+ + s \cdot m/s) = O(m)$. Při práci s haldami balíků je nejpomalejší operace Delete, ta trvá $O(\log s)$. Odtud je možno odvodit požadovaný čas $O(m \log s)$.

Poznámka 9.7 Stále zůstává otevřena otázka rychlosti asymptoticky optimálního algoritmu hledání minimální kostry v obecně ohodnoceném grafu.

Poznámka 9.8 Pro rovinné grafy algoritmus 4 s kondenzací dosahuje asymptoticky optimálního času $O(n)$. (i–tá fáze trvá $c_1 m_i = c_2 n_i$, pro vhodné konstanty c_1, c_2, a po skončení i–té fáze zbývá $n_{i+1} \leq n/2^i$ vrcholů.)

Poznámka 9.9 V roce 1990 Dixon, Rauch a Tarjan kombinací několika technik ukázali, že je možno v asymptoticky optimálním čase $O(m)$ o daném stromu zjistit, zda jde o minimální kostru. Algoritmus je značně komplikovaný a v praxi asi těžko použitelný.

10 Dijkstrův algoritmus a amortizovaná složitost

Vraťme se k tématu minulé přednášky. Hledali jsme tranzitivní uzávěru resp. všechny nejlevnější cesty. Zamysleme se nad tím, jak by se dal urychlit výpočet, kdyby nás zajímala pouze jedna dvojice vrcholů (z, s), a chtěli bychom znát pouze údaj D_s^z týkající se této dvojice.

Tak jako v minulé přednášce i nyní nejprve navrhneme algoritmus, a poté se pokusíme zjistit za jakých obecných podmínek algoritmus pracuje. Druhým cílem přednášky je ukázat, jakým způsobem se rozhodujeme zvolit datové struktury, aby měl algoritmus co nejlepší chování.

Obě úlohy jsou řešeny algoritmem Alg. 11, kde $\odot, i, o$ jsou tytéž operátory jak byly definovány u Kleenových algeber. Operace $\oplus$ je zastoupena porovnáváním „Δ“ podle něhož hledáme minimum. V našich případech se jedná o klasické uspořádání reálných čísel („Δ“=„$\geq$“). (U tranzitivního uzávěru false reprezentuje ∞, true reprezentuje 0).

Pro libovolný vrchol u máme seznam ℓ_u, právě všech vrcholů v, pro něž $A_{u,v} \neq o$. (Hrany grafu A).

Poznámka 10.1 Ztotožňujeme zde graf s jeho maticí ohodnocení.

---

const
  $V = \{1..n\}$;
type
  Vrchol $= 1..n$;
  Řádek = **array** [*Vrchol*] **of** S;
  Množina_Vrcholů =
  Blíže nespecifikovaná struktura udržující množinu *Vrcholů* ;
  Seznam =
  Jiná blíže nespecifikovaná struktura udržující množinu *Vrcholů* ;
  Sousedé = **array** [*Vrchol*] **of** *Vrchol* ;
  Matice = **array** [*Vrchol* , *Vrchol*] **of** S;
  Hrany = **array** [*Vrchol*] **of** *Seznam*;
procedure *Dijkstra* (**var** D : *Řádek* , **var** N : *Sousedé* ,
    var A : *Matice* , **var** ℓ : *Hrany*,
    z : *Vrchol* , **var** Y : *Množina_Vrcholů*);
  ($\star$ Program mimo jiné minimalizuje $\sum D_u$ $\star$)
var
  u, v : *Vrchol* ;
  X : *Množina_Vrcholů*;
begin
  $D_z := i$; $X := \{z\}$;
  $Y := X$;
  for $v \in \ell_z$ **do**
    begin
      $D_v := A_{z,v}$;
      $N_v := z$; $Y := Y \cup \{v\}$
    end
  ($\star$ Hodnoty D_u nebudou nikdy růst $\star$)
  while $X \neq Y$ **do**
    begin
      $u := argmin_{u \in Y \setminus X}(D_u)$;
      $X := X \cup \{u\}$;
  ($\star$ X je množina vrcholů v, pro něž máme D_v a N_v s konečnou platností spočítáno, Y je množina vrcholů do nichž vede hrana z množiny X $\star$)
    for $v \in l_u \setminus X$ **do**
      if $v \notin Y$ **then**
        begin
          $D_v := D_u \odot A_{u,v}$;
          $N_v := u$; $Y := Y \cup \{v\}$
        end
      else if $D_v > D_u \odot A_{u,v}$ **then**
        begin
          $D_v := D_u \odot A_{u,v}$;
          $N_v := u$;
        end
  ($\star$ D_v je „velikost cesty $(z, \ldots, N_{N_v}, N_v, v)$“, $D_u \odot A_{u,v}$ je „velikost cesty $(z, \ldots, N_{N_u}, N_u, u, v)$“ $\star$)
    end
  ($\star$ $X = Y$ je množina vrcholů u pro něž je $D_u \neq \infty$, nebo–li Y je množína vrcholů u pro něž je (z, u) v tranzitivním uzávěru $\star$)
end

Alg. 11: Dijkstrův algoritmus

Věta 10.1 *Jsou–li váhy cest definovány předpisem*

$$w((z, v_1, v_2, v_3, \ldots, v_{k-1}, v_k)) = w((z, v_1, v_2, v_3, \ldots, v_{k-1})) \odot A_{v_{k-1}, v_k} \tag{11}$$

a pro operaci $\odot$ platí axiom

$$a \leq a \odot b \tag{12}$$

potom Dijkstrův algoritmus nalezne minimální váhy cest z vrcholu z do všech vrcholů dosažitelných ze z. Algoritmus pro každý dosažitelný vrchol u navíc nalezne N_u tak, že $(z, \ldots, N_{N_u}, N_u, u)$ je cesta v níž se minimum nabývá.

Důsledek 10.1.1 *Algoritmus Alg. 11 (Dijkstrův algoritmus) spočítá velikost nejkratší cesty mezi vrcholy (z, s), nezáporně ohodnoceného grafu. (Pokud zvolíme $\odot := +$)*

Důkaz – *10.1*: K důkazu správnosti spočítaných D_s^z využijeme toho, že algoritmus jako vedlejší efekt minimalizuje $\sum_u D_u^z$. Algoritmus Alg. 11 je hladový algoritmus, k důkazu korektnosti výpočtu $\sum_u D_u^z$ využijeme to co jsme si odvodili o matroidech. Stačí nám nalézt matroid na němž algoritmus funguje, a popsat jak se v něm používají modrá a červená barvící pravidla.

Algoritmus postupně přidává vrcholy, pro něž je D_u s konečnou platností spočítáno, do množiny X. S vrcholem u přidává i hranu (N_u, u), což je poslední hrana nejkratší cesty do u. Navíc algoritmus udržuje množinu Y vrcholů do nichž vede hrana z množiny X. Vrcholy množiny X s těmito hranami tvoří strom, jemuž by tedy nějakým způsobem měla odpovídat modrá množina matroidu.

Toto by nás mohlo inspirovat k volbě následujícího matroidu.

S je množina všech cest v grafu začínajících ve vrcholu z, F je množina takových podmnožin cest, které začínají ve vrcholu z a mají po dvou různé koncové vrcholy.

Formálně:

$$S = \{p \mid p \text{ je cesta } (z, \ldots)\}$$

$$F = \{M \mid M{=}\{p_i \mid p_i \text{ je cesta } (z, \ldots, u_i) \mid u_i{=}u_j \Rightarrow i{=}j\}\}$$

Odlišnost tohoto matroidu od dosud uvedených matroidů je v tom, že váhy w nejsou zadány hodnotami, ale je pouze uveden předpis (11) na jejich spočítání.

Matroidový algoritmus používající modré barvící pravidlo, vyžaduje výběr minima z řezu, v našem případě z množiny všech cest do koncových vrcholů z $V \setminus X$. Dijkstrův algoritmus použije místo řezu množinu

$$\mathcal{X} = \{(z, \ldots, N_{N_u}, N_u, u, v) \mid u \in X, v \in Y \setminus X\}. \quad (13)$$

Aby byl Dijkstrův algoritmus korektní stačí, aby minimum z vah cest do koncových vrcholů z množiny $V \setminus X$ bylo nabýváno na cestě z množiny $\mathcal{X}$.

Jsou–li váhy cest monotónní,

$$w((z, \ldots, u)) \leq w((z, \ldots, u, v)) \quad (14)$$

potom nutně poslední hrana některé minimální cesty (z cest do koncových vrcholů z $V \setminus X$) vede z vrcholu množiny X. K důkazu platnosti podmínky (13) stačí využít vlastnosti

$$w(p_x^1) \leq w(p_x^2) \Rightarrow w(p_x^1{\_}v) \leq w(p_x^2{\_}v) \quad (15)$$

Jsou–li váhy cest definovány předpisem (11), pak vlastnost (15) plyne z definice, a pokud pro operaci $\odot$ platí axiom (12) potom (14) platí. □

Algoritmus Alg. 11, zvolíme–li jako $\odot$ operátor zapomínající první parametr ($a \odot b := b$), je implementací algoritmu na hledání minimální kostry, který byl v přednášce o matroidech pojmenován *Jarník,Prim,Dijkstra*.

K důkazu korektnosti této varianty Dijkstrova algoritmu můžeme stěží použít větu 10.1, protože tato modifikace nepočítá velikosti cest. (V přednášce o matroidech byl tento důkaz proveden s pomocí jednoduchého matroidu (hrany,lesy).)

Co říci závěrem, než se pustíme do volby vhodných datových struktur?

Dijkstrův algoritmus je hladový algoritmus, pomocí kterého můžeme řešit některé grafové úlohy. V jeho obecnosti ale neznáme jednoduše zformulovatelné podmínky, které musí (graf,$\odot$,zadání úlohy) splňovat, aby algoritmus úlohu řešil.

Nyní zvolíme vhodné datové struktury. Nejdříve si rozmysleme, co od dosud nespecifikovaných struktur očekáváme. Od seznamů ℓ_u reprezentujících množinu vrcholů vyžadujeme pouze, aby nám postupně vydal všechny jeho prvky. Mnoho struktur umožňuje tyto operace implementovat v konstantním čase na prvek množiny.

Algoritmus pracuje dále s množinami X a Y. Postupně obě množiny zvětšuje, z množiny $Y \setminus X$ odebírá minimum a případně zmenšuje hodnoty prvků v množině $Y \setminus X$.

Je–li n_z počet vrcholů dosažitelných z vrcholu z a m_z počet hran dosažitelných z vrcholu z, potom algoritmus n_z-krát odebírá minimum z množiny $Y \setminus X$ a nejvýš m_z-krát snižuje hodnoty prvků v množině $Y \setminus X$.

Z tohoto rozboru zjišťujeme, že je přirozené pracovat s datovou strukturou udržující místo množin X, Y množinu $Y \setminus X$. Pro souvislý graf bychom algoritmus mohli naprogramovat bez struktur X a Y pro množiny. Pro nesouvislý graf hrají v algoritmu množiny X a Y i nadále užitečnou roli. Od struktury pro množiny X a Y vyžadujeme rychlé vkládání vrcholů a rychlý test přítomnosti v množině. Typem množin X a Y může být například pole booleovských hodnot.

Zbývá navrhnout datovou strukturu pro množinu $Y \setminus X$. Vhodnou strukturou jsou Fibonacciho haldy.

Podle *amortizovaného rozboru* této datové struktury, který provedeme v následující kapitole, umožňuje tato struktura provést n_z vložení prvku do množiny, n_z odebrání minima a m_z snížení hodnot klíčů jednotlivých prvků v celkovém čase $O(n_z \log n_z + m_z)$.

Algoritmus Alg. 12 je tento algoritmus přepsaný s použitím Fibonacciho hald.

---

const
  $V = \{1..n\}$;
type
  Vrchol $= 1..n$;
  Řádek=**array** [*Vrchol*] **of** *S*;
  Množina_Vrcholů =
  Nespecifikovaná struktura která umožňuje testovat přítomnost v množině *Vrcholů*a vkládat do množiny;
  Seznam_Vrcholů =
  Nespecifikovaná struktura která generuje prvky množiny *Vrcholů*;
  Sousedé = **array** [*Vrchol*] **of** *Vrchol*;
  Matice = **array** [*Vrchol*, *Vrchol*] **of** *S*;
  Hrany = **array** [*Vrchol*] **of** *Seznam_Vrcholů*
procedure *Dijkstra*(**var** *D* : *Řádek*, **var** *N* : *Sousedé*,
    var *A* : *Matice*, **var** ℓ : *Hrany*,
    z : *Vrchol*, **var** *Y* : *Množina_Vrcholů*);
var
  u, v : *Vrchol*;
  $\mathcal{H}$: *Heap*; *X* : *Množina_Vrcholů*;
begin
  $D_z := i$; $X := \{z\}$;
  $Y := X$; *MakeEmptyHeap*($\mathcal{H}$);
  ($\star$ Halda bude udržovat množinu vrcholů, a používat klíče D_v. Hodnoty D_v nesmějí být měněny jinak než pomocí funkcí *Insert* a *DecrementTo* $\star$)
($\star$ Jiným řešením je udržovat údaje D_v dvojmo $\star$)
  for $v \in \ell_z$ **do**
    begin
      Insert($\mathcal{H}$, $(v, A_{z,v})$);
      $N_v := z$; $Y := Y \cup \{v\}$
    end
  while not *EmptyHeap*($\mathcal{H}$) **do**

```
    begin
      FindMin(H, (u, D_u));
      X := X ∪ {u}; Deletemin(H);
      for v ∈ l_u \ X do
          if v ∉ Y then
              begin
                Insert(H, (v, D_u ⊙ A_{u,v}));
                N_v := u; Y := Y ∪ {v}
              end
          else if D_v > D_u ⊙ A_{u,v} then
              begin
                DecrementTo(H, (v, D_u ⊙ A_{u,v}));
                N_v := u;
              end
    end
end
```

Alg. 12: Dijkstrův algoritmus na FH

Věta 10.2 *Algoritmus Alg. 12 pracuje v celkovém čase $O(n_z \log n_z + m_z)$, kde n_z je počet vrcholů dosažitelných z vrcholu z a m_z je počet hran dosažitelných z vrcholu z.*□

Poznámka 10.2 U souvislého grafu existenci seznamů ℓ_u nemusíme předpokládat, pokud máme seznam hran grafu, jsme v čase $O(m)$ schopni seznamy l_u vytvořit (m je počet hran grafu).

Věta 10.3 *Volba Fibonacciho hald jako datové struktury pro Dijkstrův algoritmus je asymptoticky optimální.*

Důkaz: Zvolíme-li jako váhu cesty cenu koncového vrcholu ve vrcholově ohodnoceném „hvězdicovém" grafu (graf obsahuje pouze hrany typu (z, u)), pak Dijkstrův algoritmus můžeme použít na setřídění $n - 1$ čísel. Čas dijkstrova algoritmu nemůže být tedy lepší než $O(n \log n)$, což je čas nutný na toto setřídění.

Není těžké pro libovolné m a n zkonstruovat graf, v němž Dijkstrův algoritmus použije všechny hrany, než nalezne nejlevnější cestu ze z do s. Proto čas Dijkstrova algoritmu nemůže být lepší než $O(m)$. □

11 Haldy

Haldy jsou struktury podporující vyhledávání minima prvků množiny a odstraňování minima. Haldy nepodporují vyhledávání prvků množiny. Pokud potřebujeme přímo přistupovat k prvkům haldy, nic nám nebrání udržovat z vnější datové struktury ukazatele na prvky haldy.

Příkladem haldy je známá binární jednostromová halda používaná algoritmem **Heapsort**. Tato halda podporuje následující metody:

metoda	**nejhůře**	
FindMin	$\Theta(1)$	– nalezení minima
DeleteMin	$\Theta(\log n)$	– odstranění minima
Insert(i)	$\Theta(\log n)$	– vložení prvku

kde n je aktuální počet prvků haldy.

Vzhledem k tomu, že je $n!$ permutací n čísel, třídící algoritmus založený na porovnávání musí provést v průměrném případě $\log n! = O(n \log n)$ porovnání. Algoritmus **Heapsort** používající tuto haldu je asymptoticky optimální (n krát „***Insert***“, n krát ***FindMin*** a ***DeleteMin***, čas $O(n \log n + n + n \log n)$).

Použití hald k třídění je speciální případ použití hald tím, že algoritmus končí vyprázdněním haldy. Jestliže algoritmus tuto vlastnost nemá, je–li vyžadovaný počet operací ***Insert*** řádově větší než počet operací ***DeleteMin***, potom je výhodnější použít vícestromových struktur, umožňujících provádět ***Insert*** v amortizovaném čase $O(1)$.

Příkladem takových struktur jsou binomiální a Fibonacciho haldy.

11.1 Binomiální haldy

Tato struktura podporuje navíc metodu ***Meld***(h, h'), umožňující spojení dvou hald. Vzhledem k tomu, že můžeme pracovat s více haldami, je vhodné upozornit i na metodu ***MakeHeap***, umožňující založení nové haldy.

Jsou různé strategie práce s vícestromovými haldami. V následující tabulce jsou časy uvedených metod pro „zbrklou“ resp. „línou“ strategii.

metoda	**amort.**	**nejhůř**
MakeHeap(i)	$\Theta(1)$	$\Theta(1)$
FindMin(h)	$\Theta(1)$	$\Theta(1)$
Meld(h, h')	$\Theta(\lvert h' \rvert) \mid \Theta(1)$	$\Theta(\log n) \mid \Theta(1)$
DeleteMin(h)	$\Theta(\log n)$	$\Theta(\log n) \mid \Theta(s)$
Insert(h, i)	$\Theta(1)$	$\Theta(\log n) \mid \Theta(1)$

kde n je počet prvků zařazených do struktury, $t_1 \mid t_2$ jsou časy pro „zbrklou“ | „línou“ verzi, a s je počet stromů struktury. Velikost celé struktury je $O(n)$.

Poznámka 11.1 Právě uvedená tabulka je v podstatě vše co potřebuje vědět programátor používající tuto datovou strukturu (pokud není zrovna on určen k jejímu naprogramování).

Snažme se nyní přesněji popsat vlastnosti požadované struktury:

Aby operace ***FindMin*** trvala v nejhorším případě $O(1)$, je „téměř nutné“ udržovat pointer na minimální prvek haldy.

Udržovat v datové struktuře minimum zařazených prvků je jednoduché, dokud nepřijde požadavek ***DeleteMin***. (Při zařazení každého prvku porovnáme s minimem. Můžeme udržovat třeba spojový seznam prvků.)

Problémy nastávají při odstraňování minim. Je totiž třeba vždy nalézt nové minimum. Čas nutný na nalezení nového minima je úměrný počtu „kandidátů“ na minimum.

Počet kandidátů na minimum můžeme omezit tak, že prvky shlukujeme do orientovaných stromů, s pravidlem, že hodnoty prvků na cestě ke kořeni stromu klesají. Minimum potom hledáme pouze mezi kořeny stromů.

Odstraněním minima, které bylo kořenem nějakého stromu nám mezi kandidáty na nové minimum přibudou všichni synové původního minima. Odtud je vidět, že je pro nás výhodnější, aby vrcholy stromů neměly příliš mnoho synů. Vzhledem k tomu, že bychom byli rádi, aby stromů v haldě bylo co nejméně, je ale vhodné, aby stromy obsahovaly hodně vrcholů. Toho je možno dosáhnout tím, že stromy budou poměrně hluboké.

Dostatečné pro naše účely bude, aby pro nějaké pevné $2 \geq q > 1$ a $c \geq 0$ platil následující invariant:

Invariant 11.1 *Má–li vrchol v libovolného stromu haldy k synů, potom velikost podstromu s kořenem v je aspoň cq^k.*

Nazveme řádem stromu T počet synů kořene stromu T.

Kandidáti na nové minimum jsou kořeny stromů haldy a synové odebraného minima (jakožto kořeny podstromů). Při hledání nového minima můžeme výsledek každého porovnání zachytit vhodně orientovanou hranou spojující kořeny porovnaných vrcholů. Abychom zabránili přílišné šířce stromů využijeme řádu stromů (připojením jednoho vrcholu k stromu velkého řádu by mohl počet synů kořene i počet vrcholů vzrůst o 1, čímž bychom brzy porušili invariant 11.1).

Porovnáváním stromů stejného řádu dosáhneme toho, že řád výsledného stromu o jedna vzroste a nadále platí invariant 11.1. Kořeny stromů různých řádů nebudeme porovnávat za účelem spojení stromů.

Aby bylo možné porovnávat stromy stejných řádů, je potřeba, abychom byli schopni o daném stromu rychle zjistit jeho řád. To je možno zajistit tak, že v datové struktuře udržujeme pro každý prvek jakožto číslo řád podstromu zakořeněném v tomto prvku. Alternativní metodou je udržovat navíc „globální“ seznam S délky $O(\log N)$, kde N je počet prvků všech hald, a místo čísla r udržovat pointer na r–tý prvek seznamu.

Druhá metoda reprezentace řádu stromu nám umožňuje snadné hledání nového minima:

Předpokládejme, že před započetím hledání minima prvky seznam S obsahují jakožto nosnou informaci pouze pointer Null. Berme postupně stromy haldy. Pro každý strom T se snažíme opravit nosnou informaci prvku seznamu S příslušného řádu tak, aby pointer ukazoval na T. Pokud ale nosná informace není Null, ukazuje tento pointer na strom stejného řádu. Porovnáme kořeny těchto stromů a nahradíme nosnou informaci hodnotou Null. Vzniklý

strom má řád o jedna větší, takže se „řádovým“ pointrem posuneme na následující prvek seznamu S a snažíme se opravit nosnou informaci nového prvku .... Ve chvíli, kdy v původní haldě není žádný strom, máme pospojovány stromy tak, že od každého řádu existuje nejvýš jeden. Nalezneme minimum z jejich kořenů, „vrátíme“ je do haldy a opravíme příslušné nosné informace v seznamu S na Null.

Uvědomme si, že po skončení operace je počet stromů v haldě nejvýš $O(\log n)$. (Od každého řádu nejvýš jeden, nejvyšší řád je $O(\log_q n)$.)

Operace ***DeleteMin*** může trvat déle než $O(\log n)$, a to v případě, že často spojujeme stromy stejného řádu. Je–li t_t (konstantní) čas potřebný na spojení dvou stromů (včetně změn týkajících se seznamu S), a bylo–li v průběhu operace ***DeleteMin*** spojeno k stromů, potom čas operace ***DeleteMin*** může být až $O(\log n) + t_t \cdot k$.

Toto vede k definici potenciálu

$$\Phi = \text{počet stromů všech hald.}$$

Čas operace ***DeleteMin*** vůči potenciálu $t_t\Phi$ je $O(\log n)$.

Operace ***MakeHeap*** alokuje paměť pro haldu s jedním prvkem a nastaví příslušné hodnoty. Navíc tato operace může hlídat celkový počet prvků všech hald a příslušně prodlužovat seznam S. (Seznam S by mohl být prodlužován automaticky přímo procedurami ***DeleteMin***.)

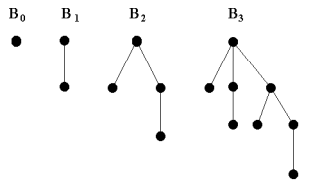
Zbývá popsat operaci ***Meld***. ***Insert***(h, i) je potom možno provést jako ***Meld***$(h,$ ***MakeHeap***$(i))$.

Zde se projeví odlišnost „zbrklé“ od „líné“ strategie. Líná strategie pouze spojí seznamy stromů a určí minimum — konstantní čas (i vzhledem k potenciálu), ale operace ***DeleteMin*** může v nejhorším případě trvat dlouho.

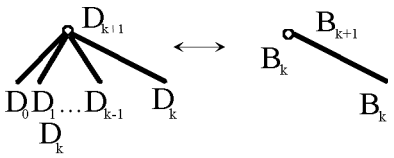
Zbrklá strategie spojí haldy tak, aby ve výsledné haldě byl od každého řádu nejvýš jeden strom (využije k tomu podobného triku jako se seznamem S) — čas úměrný celkovému počtu stromů, ten je ale stále $O(\log n)$. Při vhodnější reprezentaci (každá halda má svůj lokální seznam S_h — nemusí být „Null–ován“) je čas vzhledem k potenciálu úměrný menšímu počtu stromů spojovaných hald.

U zbrklé i u líné strategie je čas operace ***Insert*** vzhledem k potenciálu $t_t\Phi$ roven $O(1)$.

Na následujících obrázcích jsou zachyceny binomiální tvary stromů — tvary těch stromů, které vznikají v průběhu práce s datovou strukturou.



Obr. 16: Binomiální stromy řádů 0, 1, 2, 3



Obr. 17: Popis binomiálních stromů tvary B_k a D_k

Reprezentaci haldy jakožto seznamu stromů, nám může připomínat, reprezentaci bratrů jednoho stromu. Uvědomíme–li si, jaký byl důvod toho, že pracujeme s více stromy — dodržování invariantu 11.1, může nás napadnout následující modifikace struktury:

Po porovnávání hodnot kořenů stromů různých řádů jsme ztráceli informaci tím, že jsme nepřidávali hranu spojující tyto kořeny. Důvodem byl důsledek — porušení invariantu. Tuto hranu ale můžeme do struktury přidat s tím, že ji (spodní vrchol) označíme jako hranu „vnější“. Do řádu stromu vnější hrany nepočítáme, takže jimi není invariant porušen. Haldu tedy můžeme reprezentovat jedním stromem s „vnějšími“ hranami.

Definujeme–li jako potenciál místo počtu stromů počet vnějších hran, dostaneme velmi podobný amortizovaný rozbor. Chování této jednostromové haldy může být lepší, protože vnější hrany se mohou dostat hlouběji do stromu, a to nám umožňuje zbytečně neporovnávat s kořeny některých stromů. Po každé operaci ***DeleteMin*** je mezi vnějšími syny minima nejvýš jeden strom od každého řádu. Počet všech vnějších hran není jinak omezen, potenciál Φ tedy může růst, což znamená, že součet časů je ve skutečnosti ještě lepší než víme z rozboru (ze součtu časů vůči potenciálu).

Zajímavým požadavkem je možnost odebrání libovolného prvku z haldy. U binomiálních hald je to možno řešit tak, že příslušnou buňku v haldě označíme za prázdnou a vnější ukazatel prvku na buňku haldy opravíme na Null. (Používáme „externí“ reprezentaci.) Pro takovou modifikaci je nutno provést nový rozbor procedury ***FindMin***.

Dalším možným požadavkem na datovou strukturu je umožnit snížení hodnoty obecného prvku haldy. To přináší komplikace především u vnitřních prvků stromů haldy, protože po snížení hodnoty prvku by nemusela platit nerovnost reprezentovaná hranou k otci tohoto prvku.

Nabízející se řešení tuto hranu prostě zrušit bohužel může vést k relativnímu rozšiřování stromů. Fredman a Tarjan v roce 1984 publikovali popis datové struktury nazvané Fibonacciho haldy, která tento problém řeší.

11.2 Fibonacciho haldy

Fibonacciho haldy umožňují navíc metodu ***Cut***(v), která zajistí, aby v byl kořen některého stromu haldy. Pomocí

této metody je možno jednoduše implementovat metodu ***Decrement***(h, v, δ). Obecnou metodu ***Delete***(h, v) je s pomocí ***Cut*** možno provést i jinak než označením příslušné buňky za prázdnou.

V následující tabulce jsou časy uvedených metod pro „zbrklou“ resp. „línou“ strategii.

metoda	**amort.**	**nejhůř**
MakeHeap(i)	$\Theta(1)$	$\Theta(1)$
FindMin(h)	$\Theta(1)$	$\Theta(1)$
Meld(h, h')	$\Theta(\lvert h'\rvert) \mid \Theta(1)$	$\Theta(\log n) \mid \Theta(1)$
Cut(h, v)	$\Theta(1)$	$\Theta(s{+}\check{c})$
DeleteMin(h)	$\Theta(\log n)$	$\Theta(\log n) \mid \Theta(s)$
Insert(h, i)	$\Theta(1)$	$\Theta(\log n) \mid \Theta(1)$
Delete(h, v)	$\Theta(\log n)$	$\Theta(s{+}\check{c})$
Decrement(h, v, δ)	$\Theta(1)$	$\Theta(s{+}\check{c})$

kde n je počet prvků zařazených do struktury, $t_1 \mid t_2$ jsou časy pro „zbrklou“ | „línou“ verzi, s je počet stromů struktury a $\check{c}$ je počet černých vrcholů struktury. Velikost struktury je $O(n)$.

Nejprve popíšeme datovou strukturu, kde požadujeme, aby operace ***FindMin*** trvala konstantní čas. Později ukážeme, jak je možno měnit poměr časů funkcí ***FindMin*** a ***Delete***.

Základní myšlenkou Fibonacciho hald je, že stačí zajistit, aby každému vrcholu stromu haldy byl odřezán nejvýš jeden syn. To nám zajistí platnost invariantu 11.1. K zajištění této podmínky slouží „začerňování“ vrcholů, jimž byl syn odebrán. Ostatní vrcholy jsou „obarveny“ bíle.

Poznámka 11.2 Fibonacciho haldy jsou tedy přirozeným rozšířením binomiálních hald. Reprezentaci prvku binomiální haldy stačí rozšířit o jediný bit.

Operace ***Cut***(h, v) probíhá následovně:

Vrchol v obarvíme bíle. Je–li v kořen nějakého stromu haldy, operace ***Cut*** končí. Jinak zrušíme hranu od prvku v k jeho otci f a snížíme řád prvku f. Byl–li prvek f kořenem nějakého stromu haldy, operace končí. Byl–li prvek f bílý, začerníme jej a operace končí. Je–li prvek f černý, provedeme ***Cut***(h, f).

Čas operace ***Cut*** nemusí být konstantní, nekonstantnost může být způsobena tím, že na cestě od vrcholu v ke kořeni stromu haldy je příliš mnoho černých vrcholů. Je–li t_c čas nutný na zrušení hrany (v, f), změnu řádu prvku f a „obarvení“ vrcholu, a bylo–li v průběhu operace ***Cut*** odčerněno (obarveno bíle) k vrcholů, potom čas operace ***Cut*** byl $O(1) + t_c \cdot k$ a vzniklo k nových stromů.

Toto vede k definici potenciálu

$$\Psi = \text{počet černých vrcholů v haldách.}$$

Čas operace ***Cut*** vůči potenciálu $t_t\Phi + (t_c + t_t)\Psi$ je $t_c + + t_t + O(1) = O(1)$.

Metodu *Decrement* nyní již můžeme snadno implementovat pomocí metody ***Cut***. Obdobně můžeme pomocí ***Cut*** implementovat ***Delete*** (nejedná–li se o minimum). Pro ostatní metody můžeme zcela převzít algoritmy binomiálních hald.

Vzhledem k tomu, že potenciál Ψ mění pouze operace ***Cut***, zůstává časový odhad ostatních operací zachován i vůči potenciálu $t_t\Phi + (t_c + t_t)\Psi$.

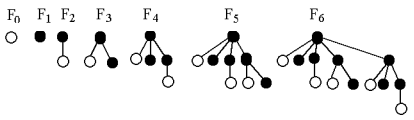
Ještě potřebujeme ukázat platnost invariantu 11.1 pro Fibonacciho stromy. K tomu slouží následující lemma:

Lemma 11.2 *Nechť v je libovolný prvek Fibonacciho haldy. Seřadíme–li syny prvku v v pořadí, v jakém byli připojováni, potom i–tý syn je řádu aspoň $i - 2$.*

Důkaz: Je–li u i–tý syn prvku v, potom v čase kdy byl prvek u připojen měl prvek v aspoň $i - 1$ synů. Proto měl prvek u také aspoň $i - 1$ synů (spojovány pouze stromy stejného řádu). Prvku u mohl ubýt nejvýš jeden syn, je tedy řádu aspoň $i - 2$. □

Označíme–li G_k minimální nutný počet prvků podstromu s kořenem stupně k, dostaneme podle předchozího lemmatu rekurenci $G_k \geq 1 + 1 + G_0 + G_1 + \cdots + G_{k-2}$. Odečtením vztahu pro k a $k + 1$ dostáváme $G_{k+1} - G_k = G_{k-1}$. Navíc $G_0 = 1$, $G_1 = 2$. Čísla G_k odpovídají známé Fibonacciho posloupnosti (odtud jméno hald), $G_k = F_{k+1}$. Pro Fibonacciho čísla platí $F_k = \left[\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{k+1}\right]$. Odtud plyne platnost invariantu 11.1.

Na následujícím obrázku jsou zachyceny minimální stromy jednotlivých řádů.



Obr. 18: Nejmenší Fibonacciho stromy řádů 0,1,2,3,4 a 5

Oprostíme–li se od požadavku, abychom stále udržovali pointer na minimální prvek haldy, můžeme urychlit operaci ***Delete*** (i ***DeleteMin***) na úkor operace ***FindMin***. (Líná stratcgic pro ***Delete***.)

Stejně jako u binomiálních hald obecnou operaci ***Delete*** můžeme provádět jednoduše tak, že prvky haldy pouze označujeme za smazané. Může se dokonce stát, že minimum celé haldy bude označeno za smazané. Ve chvíli, kdy při operaci ***Meld*** porovnáváme minima hald, by se nám mohlo stát, že jedno z minim je prohlášeno za smazané. V takovém případě jej prohlásíme za nové minimum.

Problémy nastanou až při provádění operace ***FindMin***, je–li původní minimum označeno za smazané. V tu chvíli začneme odstraňovat z jednotlivých stromů haldy prvky označené za smazané. Odstraňujeme vždy tak dlouho, až narazíme na neoznačený vrchol. Vznikne nám halda, v níž kořen žádného stromu není označen jako smazaný. Operace končí stejně jako původní *DeleteMin* pospojováním stromů stejných řádů a nalezením minima.

Lemma 11.3 *Je–li k počet odstraněných prvků v průběhu operace **FindMin**, potom čas operace **FindMin** vzhledem k potenciálu $t_t\Phi + (t_c + t_t)\Psi$ je $O(1 + k(\log(n/k) + 1))$.*

Důkaz: Pro $k = 0$ je tvrzení triviální. Pro $k > 0$ trvá první část operace ***FindMin*** $O(k)$, ale může při ní vzniknout mnoho stromů. Musíme odhadnout změnu potenciálu Φ v první fázi. Potenciál Ψ se nemění. Jsou–li d_1, d_2, ..., d_k počty neoznačených synů odstraněných prvků, potom změna potenciálu Φ je $d_1 + d_2 + \cdots + d_k$.

Tento součet můžeme odhadnout na základě lemmatu 11.2. Je–li totiž d_i počet neoznačených synů prvku p_i, potom některý ze synů prvku p_i je řádu aspoň $d_i - 2$, a podle invariantu 11.1 podstrom pod tímto prvkem obsahuje aspoň cq^{d_i-2} prvků. Odtud $nq^2 \geq cq^{d_1} + cq^{d_2} + \cdots + cq^{d_k}$, protože příslušné podstromy jsou disjunktní. Za této podmínky je součet $\sum d_i$ maximální, pokud jsou všechna d_i stejná.

$$\Delta\Phi \leq k \cdot \log_q(nq^2/kc) = O(k \log(n/k)).$$

Odhad času druhé fáze vůči potenciálu $t_t\Phi + (t_c + t_t)\Psi$ je $O(\log n)$ (původní ***DeleteMin***). Po sečtení dostáváme požadovaný odhad. □

Poznámka 11.3 Tato implementace metody ***FindMin*** umožňuje tzv. implicitní mazání: Je dán predikát (nezávislý na organizaci haldy), a pomocí tohoto predikátu je o každém prvku haldy určeno, zda má být odstraněn. (Takovým způsobem je možno například při hledání minimální kostry pomocí hald hran zneplatnit všechny hrany vedoucí uvnitř vytvořených stromů — predikátem $P((u,v)) := tree(u) = tree(v)$)

Výhoda této implementace je v její lenosti. Neztrácíme zbytečně čas odstraňováním prvků, které dosud odstranit nemusíme. Nevýhodou implementace je přílišná prostorová náročnost.

Nevýhodu přílišné prostorové náročnosti (operací ***Delete***) můžeme u Fibonacciho hald odstranit, tak, že prvky rušíme za pomoci metody ***Cut***. Pokud bychom chtěli i nadále používat implicitního (nebo zadrženého) mazání, nic nám nebrání obě metody kombinovat.

Operaci ***Delete***(h, v), není–li v minimum, provádíme tak, že provedeme ***Cut***(h, v) následovaný línou operací „***Meld***“ provedenou s množinou synů vrcholu v (s tím, že neporovnáváme minima) a zrušením prvku v.

Operaci ***Delete***, při níž mažeme minimum můžeme provést buď podle původního algoritmu s tím, že rovnou vyhledáme nové minimum (stále udržujeme pointer na minimum), nebo líně tak, že minimální prvek pouze označíme za smazaný a nové minimum hledáme až v průběhu operace ***FindMin***.

Lemma 11.4 *Čas posloupnosti k_1 operací* ***Delete****, provedených za pomoci metody* ***Cut****, následovaná (netriviální) operací* ***FindMin****, v průběhu níž je k_2–krát provedeno implicitní (odložené) mazání, je vzhledem k potenciálu $t_t\Phi + (t_c + t_t)\Psi$ nejvýš $O\big(k(\log(n/k) + 1)\big)$, kde $k = k_1 + k_2$.*

Důkaz: Důkaz je téměř stejný jako důkaz lemmatu 11.3. Opět se využije nerovnosti $nq^2 \geq cq^{d_1} + cq^{d_2} + \cdots + cq^{d_{k_1}} + cq^{d_{k_1+1}} + \cdots + cq^{d_k}$ k odhadu součtu $d_1 + d_2 + \cdots + d_k$ počtu nezrušených synů zrušených vrcholů. Je–li d počet synů vrcholu v, potom čas operace ***Delete***(h, v), naprogramované za pomoci metody ***Cut***, vůči potenciálu $t_t\Phi + (t_c + t_t)\Psi$ je $O(1) + t_t \cdot (d - 1)$. Součet časů všech k_1 operací ***Delete*** vůči tomuto potenciálu je tedy $O(k_1) + t_t(d_1 + d_2 + \cdots + d_{k_1})$.

První fáze operace ***FindMin*** trvá čas $O(k_2)$, potenciál Φ během této fáze vzroste o $d_{k_1+1} + \cdots + d_k$. Součet časů první fáze operace ***FindMin*** a úvodních k_1 operací ***Delete*** je vzhledem k potenciálu $t_t\Phi + (t_c + t_t)\Psi$ roven $O(k) + t_t(d_1 + d_2 + \cdots + d_{k_1} + d_{k_1+1} + \cdots + d_k) = O\big(k(\log(n/k) + 1)\big)$. Čas druhé fáze operace ***FindMin*** vůči tomuto potenciálu je $O(\log n)$ (odpovídá to původní operaci ***DeleteMin***). □

Na všechny uvedené implementace je možno jako u binomiálních hald použít jednostromovou modifikaci haldy.

12 *NP*–úplnost

Dosud jsme se zabývali rychlými algoritmy na řešení nejrůznějších úloh. Naskýtá se otázka, zda pro každou úlohu existuje rychlý algoritmus. Z teorie vyčíslitelnosti víme, že odpověď je ne, existují „libovolně těžké“ úlohy.

Abychom si usnadnili vyjadřování, formálně rozlišíme pojmy úloha a problém. Cílem úlohy je pro vstup (instanci) I vydat úlohou specifikovaný výstup $J = f(I)$. Problém (resp. *rozhodovací* problém) je úloha která má vydat úlohou specifikovaný výstup ANO/NE.

Pokud budeme chtít ukázat, že úloha je těžká, stačí ukázat, že je těžké rozhodnout problém: Je J řešením úlohy pro vstup I? Nalezením těžkých rozhodovacích problémů ukážeme existenci těžkých úloh.

Zaveďme pojem rychlost (čas) na řešení úlohy: Předpokládejme, že vstupy (instance) I úlohy (resp. problému) jsou zakódovány v abecedě $\{0,1\}^*$. Označme $n = |I|$ délku vstupního slova. Řekneme, že algoritmus $\mathcal{A}$ řeší úlohu (resp. problém) v čase $t_{\mathcal{A}}(n)$, pokud čas spotřebovaný algoritmem je nejvýš $t_{\mathcal{A}}(n)$ pro libovolný vstup velikosti n.

Již nyní si všimněme, že efektivita algoritmu závisí na způsobu zakódování zadání. Například zakódujeme-li přirozené číslo k jednou jako k jedniček a podruhé v binární soustavě, pak algoritmus pracující v čase $\Theta(k)$ je v prvním případě lineární a v druhém exponenciální (v počtu bitů vstupu).

12.1 Třída *P*

Nyní se pokusíme definovat rychle řešitelné úlohy. Uvědomme si, že hovoříme o rychlosti „optimálního algoritmu“ na řešení úlohy U, to znamená, o čase $t(n)$, potřebném na nalezení odpovědi $J = f(I)$ na vstup I, kde $|I| = n$. Chceme–li definovat třídu rychle řešitelných úloh, chceme vlastně definovat třídu funkcí $t(n)$. Úloha bude do třídy úloh patřit, pokud existuje algoritmus, jehož rychlost $t(n)$ patří do této třídy funkcí.

Chtěli bychom, aby naše třída funkcí měla některé následující vlastnosti.

1. Protože pro každou konstantu k, jsme principiálně schopni s předvýpočtem, pomocí tabulky odpovědí pro slova I, kde $|I| \leq k$, sepsat triviální algoritmus pro takto malá I, proto měřit rychlost algoritmu jinak než asymptoticky nemá význam. O tom, zda funkce patří do dané třídy musí rozhodovat její asymptotické chování.

2. Chceme, abychom zřetězením konstantně mnoha úloh nemohli opustit tuto třídu úloh. Vzhledem k tomu, že výstup algoritmu $\mathcal{A}$ může mít velikost až $t_{\mathcal{A}}(n)$, proto skládáním funkcí $(t_1(t_2(n)))$ nemůžeme opustit třídu funkcí.

3. Užitečná by byla i uzavřenost funkcí na konstantní počty součtů a součinů.

4. V třídě by měly být nějaké netriviální úlohy, proto by funkce $O(n)$ měly do třídy funkcí patřit.

5. Hezkou vlastností by bylo, kdyby o příslušnosti úlohy do třídy nerozhodoval výpočtový model.

Uvědomme si, že nebýt vlastností 3, 5, mohli bychom zvolit například jen všechny funkce $O(n)$. Vlastnost 3 vynucuje, aby naše třída obsahovala všechny funkce shora omezené polynomy. Chceme–li definovat neprázdnou třídu rychlých úloh, musí příslušná třída funkcí všechny polynomy obsahovat.

Filozofické otázce, zda algoritmus s časem omezeným libovolným polynomem, můžeme pokládat za rychlý, se můžeme vyhnout, nebudeme–li hovořit o rychlých, ale o polynomiálních algoritmech.

Poznámka 12.1 Opačnou otázkou je, zda bychom mezi rychlé algoritmy neměli počítat i algoritmy se superpolynomiálním časem. Exponenciální algoritmy jistě mezi rychlé počítat nebudeme. Jak by se vám ale líbily třídy asymptoticky shora omezené například funkcemi $n^{(\log n)^{O(1)}} \equiv 2^{\log n^{O(1)}}$, $n^{(\log\log n)^{O(1)}}$, $\ldots n^{\alpha(n)^{O(1)}}, \ldots$.

Zde je na místě položit ještě dvě **filozofické otázky**: Představme si, že výzkum algoritmu proběhl v následujících krocích:

1. Byl vymyslen algoritmus $\mathcal{A}$.

2. Byla nekonstruktivně dokázána existence konstanty c tak, že rychlost algoritmu $\mathcal{A}$ je $t_{\mathcal{A}}(n) = O(n^c)$.

3. Byl nalezen polynom p tak, že $t_{\mathcal{A}}(n) \leq p(n)$.

První otázka: Odkdy je algoritmus polynomiální?
Druhá otázka: Odkdy víme, že algoritmus je polynomiální?

Postavme se na stanovisko, že správné odpovědi jsou po řadě 1, 2.

V teorii NP–úplnosti se pracuje především s rozhodovacími problémy. Uvědomme si, jak je možno řešení úlohy hledat pomocí rozhodovacích problémů.

Je–li množina možných výstupů úlohy omezena polynomiálně ve velikosti vstupu, a je–li jednoduché generovat prvky této množiny, pak je úloha polynomiální, právě když je polynomiální rozhodovací problém typu je J správným výstupem na vstup I? (Stačí se zeptat postupně na všechny možné výsledky.)

12.2 Třída *NP*

Třída NP by měla obsahovat úlohy řešitelné nedeterministickým polynomiálním algoritmem.

Protože nemusí být jednoznačné, co je výstupem nedeterministického výpočtu, je jednodušší než s nedeterministickými úlohami pracovat pouze s nedeterministickými rozhodovacími problémy. Výsledek takového výpočtu je potom ANO v případě, že některý z výpočtů skončil ANO, výsledek je NE, pokud všechny výpočty skončily NE.

Čas (rychlost) nedeterministického algoritmu můžeme definovat jako čas nejpomalejšího výpočtu.

Poznámka 12.2 Jinou možností je definovat čas na základě nejrychlejšího přijímacího výpočtu. Technické komplikace potom přináší fakt, že některé úlohy nemají přijímací výpočet.

Vzhledem k tomu, že můžeme k algoritmu přidat „polynomiální počítadlo“, jsme schopni příliš pomalé nepřijímací výpočty eliminovat.

Poznámka 12.3 Úlohy bychom mohli definovat takto: výsledky nedeterministických výpočtů by byly buď NE, nebo (ANO,J). Celkový výsledek výpočtu by byl v případě existence výpočtu končícího ANO libovolný z výstupů (ANO,J).

Definice 12.1 Třída NP je třída problémů, pro něž existuje nedeterministický algoritmus, jehož rychlost je shora omezena polynomem.

Poznámka 12.4 Nedeterministický algoritmus je možno přepracovat na deterministický následující metodou. Nejprve zajistíme, aby se výpočet algoritmu v každém kroku větvil nejvýše na 2 větve. Tyto větve očíslujeme 0,1. Tím způsobíme nejvýš konstantní zpomalení algoritmu. Potom algoritmu přidáme nekonečnou vstupní pásku nad abecedou 0,1. Algoritmus opravíme tak, aby se v i–tém kroku větvil podle číslice na i–tém místě přidané vstupní pásky. Algoritmus je nyní deterministický, jeho původní nedeterminismus je plně zachycen přidanou vstupní páskou.

To vede k ekvivalentní definici třídy NP, pomocí *verifikačních* algoritmů.

Definujme *verifikační* algoritmus $\mathcal{A}$. Na vstupu má binární řetězec x (vstupní instanci) a binární řetězec y (ověření). Řekneme, že algoritmus $\mathcal{A}$ verifikuje řetězec x, jestliže existuje ověření y takové, že $\mathcal{A}(x,y)=1$.

Definice 12.2 Třída NP je třída těch problémů, které lze verifikovat v polynomiálním čase. Formálně:

$NP = \{L \subseteq \{0,1\}^* \mid$ existuje verifikační algoritmus $\mathcal{A}$ a polynom p, tak, že pro všechna $x \in \{0,1\}^*$, $x \in L$, právě když existuje ověření y, $\mathcal{A}(x,y)=1$, kde pro čas algoritmu $\mathcal{A}$ platí $t_{\mathcal{A}}(x,y) \leq p(|x|)$ $\}$.

Poznámka 12.5 Abychom pomocí verifikačního algoritmu rozhodli, zda $x \in L$, stačí probrat všechna $y \in \{0,1\}^{p(|x|)}$, protože v čase $p(|x|)$ nemůžeme větší část řetězce y přečíst.

Zavedeme poslední definici třídy NP, v níž je y polynomiální:

Definice 12.3 NP je třída těch jazyků, které je možno popsat předpisem

$$\mathcal{L} = \left\{ x \mid \exists y, |y| \leq P_1(x)\ R^P(x,y) \right\},$$

kde $R^P(x,y)$ je predikát vyčíslitelný v polynomiálním čase vůči $|x|+|y|$.

Tato definice třídy NP velmi usnadňuje ověřování, zda daný jazyk je ve třídě NP.

Věta 12.1 *Definice 12.1, 12.2 a 12.3 jsou ekvivalentní.*

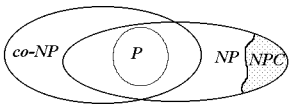
Důkaz: Poznámka 12.4 ukazuje, že definice 12.2 definuje aspoň tak velkou třídu jako definice 12.1. Poznámka 12.5 ukazuje, že definice 12.3 definuje aspoň tak velkou třídu jako definice 12.2. Abychom ukázali, že všechny tři definice definují stejnou třídu, ukážeme, že definice 12.1 definuje aspoň tak velkou třídu jako definice 12.3.

Ukážeme, že k polynomiálnímu deterministickému algoritmu na výpočet predikátu $R(x,y)$ a polynomu P_1 je možno zkonstruovat polynomiální nedeterministický algoritmus, jehož každý přijímací výpočet pro vstup x jednoznačně odpovídá právě jednomu ověření $y, |y| \leq P_1(|x|)$.

Nedeterministický algoritmus bude pracovat takto: Nejprve bude nedeterministicky hádat bity řetězce $y, |y| \leq \leq P_1(|x|)$ (počítá si, aby nevytvořil příliš dlouhý řetězec), a poté se spustí deterministický algoritmus na výpočet $R(x,y)$. (Množinu stavů můžeme rozdělit na dvě části, kde přechodová funkce je na první množině nedeterministická a na druhé deterministická, a nedovoluje přejít ze stavů druhé množiny do stavů množiny první. Navíc nedeterminismus v první množině je pouze v tom, zda hádáme jedničku, nulu nebo konec řetězce y.)

Z konstrukce algoritmu (Turingova stroje) je vidět, že počet jeho přijímacích výpočtů přesně odpovídá počtu ověření a všechny výpočty jsou polynomiálně dlouhé. □

Samozřejmě $P \subseteq NP$, ale opačná inkluze, ač nepravděpodobná, je stále otevřená. Okolo topologie třídy NP je stále mnoho otazníků. Definujme například třídu $co-NP$ jako třídu těch problémů jejichž doplňky (do $\{0,1\}^*$) jsou v NP. Pak je pravděpodobná představa znázorněná na Obr. 19.



Obr. 19: Možná topologie třídy NP

12.3 NP–úplnost a polynomiální transformace

Ve snaze rozhodnout vztah mezi třídami P a NP, byla definována třída NPC, nejtěžších NP problémů.

Pokud by se podařilo nalézt „optimální“ algoritmus pro některý z NP–úplných problémů (problém z NPC), potom by čas tohoto algoritmu otázku „P=NP?“ rozhodl.

Pokud je totiž některý z problémů třídy NPC polynomiální, potom jsou všechny problémy z NP polynomiální.

K důkazu existence NP–úplného problému budeme potřebovat nový pojem *polynomiální transformace.* Funkce $f : \{0,1\}^* \to \{0,1\}^*$ je *polynomiálně vypočitatelná*, když existuje polynomiální algoritmus $\mathcal{A}$ takový, že pro vstup

$x \in \{0,1\}^*$ dává výstup $f(x)$ (když úloha spočítat $f(I)$ je polynomiální).

Definice 12.4 Řekneme, že jazyk L_1 je polynomiálně transformovatelný (převoditelný) na jazyk L_2, píšeme $L_1 \propto$ $\propto L_2$, když existuje polynomiálně vypočítatelná funkce f : $: \{0,1\}^* \to \{0,1\}^*$ taková, že pro všechna $x \in \{0,1\}^*$

$$x \in L_1 \text{ ,právě když } f(x) \in L_2.$$

Lemma 12.2 *Relace $\propto$ je tranzitivní.*□

Polynomiální transformace nám dávají možnost definovat a ověřovat *NP*–těžkost a *NP*–úplnost.

Definice 12.5 Jazyk L je *NP*–těžký, když

$$\forall L' \in NP, L' \propto L.$$

Definice 12.6 Jazyk L je *NP*–úplný, když

1. $L \in NP$,
2. L je *NP*–těžký.

Poznámka 12.6 Kromě polynomiální transformace problému budeme později potřebovat *polynomiální redukci*. Odlišnost je v tom, že na vyřešení problému použijeme druhý problém polynomiálně krát, vždy na polynomiálně velká data. Pokud bychom měli polynomiální algoritmus řešící druhý problém, uměli bychom polynomiálně řešit i problém první.

Pro účel důkazu *P=NP* by proto lépe vyhovovala definice NP–těžkých úloh pomocí redukcí. Praktických problémů, které umíme redukovat a neumíme transformovat není mnoho.

Transformacemi jsou ale výpočetní třídy děleny mnohem jemněji, proto pro studium jejich vlastností se za předpokladu $P \neq NP$ transformace hodí lépe.

Vezměme si problém KACHL, ve kterém máme vykachlíčkovat koupelnu podle předem daných pravidel. Formálně jde o jazyk L_{KACHL}

$\{\langle B, K, S\rangle |$ B je konečná množina barev, K je konečná množina kachlíčků typu  , jejichž trojúhelníky jsou obarveny barvami z B; S je čtvercová síť rozměru $n \times n$, jejíž obvod je obarven barvami z B a existuje pokrytí sítě S kachlíčky z K tak, že jsou splněny následující podmínky

- *orientace* – kachlíčky není dovolené otáčet,
- *hranice* – trojúhelníky kachlíčků přilehlých k hranici sítě S mají s hranou hranice shodnou barvu,
- *sousednosti* – trojúhelníky kachlíčků, které spolu sousedí hranou mají shodnou barvu $\}$.

Zjištění, zda vykachlíčkovaná síť splňuje podmínky z definice problému KACHL, lze jistě provést v polynomiálním čase. Proto je problém (jazyk) KACHL ve třídě *NP*. Dokážeme větu.

Věta 12.3 *Problém KACHL je NP–těžký.*

Důkaz: Nechť L je jazyk z *NP*. Nechť známe nějaký nedeterministický Turingův stroj řešící rozhodovací problém „Je $I \in L$?“ v polynomiálním čase. Nechť známe tento polynom p.

Popíšeme polynomiální úlohu $\mathcal{U}$, která bijektivně zobrazuje každý binární řetězec I na zadání J problému KACHL, $J = f(I)$, tak, že $I \in L$ právě když $f(I) \in KACHL$.

Předpokládejme dále, že máme k dispozici následující *ekvivalentní* modifikaci $\mathcal{M}$ Turingova stroje:

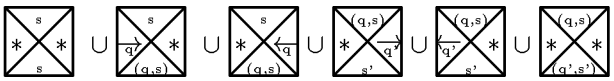
1. Stroj $\mathcal{M}$ pracuje s tzv. přetrženou páskou (je nekonečná jen v jednom směru). Na úplném začátku pásky je zapsána vstupní instance. Všude jinde je zapsán prázdný symbol $*$. Hlava stroje $\mathcal{M}$ je na začátku nastavena na první políčko pásky. Ze startovní polohy se stroj může pohybovat jen směrem doprava.
2. Stroj $\mathcal{M}$ má právě jednu přijímací koncovou konfiguraci. Kód stroje je doplněn takto: dříve než přijme, vyčistí pásku zapsáním prázdného symbolu $*$, posune hlavu řízení na 1. políčko pásky a přejde do koncového stavu q_f.
3. Stroj pracuje nad abecedou $\{0,1,*\}$.

Označme Q množinu stavů a δ přechodovou funkci stroje $\mathcal{M}$.

Úloha $\mathcal{U}$ nejprve zkonstruuje síť velikosti $p(|I|) \times p(|I|)$. Za množinu barev vezme $\{0,1,*\} \cup Q \times \{\leftarrow, \rightarrow\} \cup Q \times \{0,1,*\}$.

Nechť $y \in \{0,1\}^*$ označuje vstupní instanci. Prvních $|y|$ políček horní vodorovné hranice úloha $\mathcal{U}$ obarví $|y|$–tici, $(q_0, y_1), y_2, \ldots, y_{|y|}$, první políčko dolní vodorovné hranice obarví barvou $(q_f, *)$. Zbývající hraniční políčka jsou obarveny barvou $*$.

Konečně množinu kachlíčků K zkonstruuje takto: Kachlíčky budou 6–ti druhů:



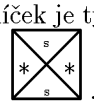
kde s je symbol na pásce, tj. $0 \vee 1 \vee *$, $q, q' \in Q$. V množině K budou zastoupeny všechny kachlíčky druhů 1, 2 a 3. Kachlíček druhu 4 je v K zastoupen, právě když $(q', s', \rightarrow) \in \delta(q, s)$. Kachlíček druhu 5 je v K zastoupen, právě když $(q', s', \leftarrow) \in \delta(q, s)$. Kachlíček druhu 6 je v K zastoupen, právě když $(q', s', \cdot) \in \delta(q, s)$.

Tím je definována vstupní instance $f(I)$ problému KACHL. Transformaci $f(I)$ je možno zkonstruovat v polynomiálním čase.

Zbývá ověřit, že Turingův stroj $\mathcal{M}$ přijímá $y \Leftrightarrow$ $\Leftrightarrow$ zkonstruovaná instance $\in L_{\text{KACHL}}$. To však je již jen technickou záležitostí. Každý krok stroje $\mathcal{M}$ můžeme zakódovat položením řádku kachlíčků a obráceně každý vykachlíčkovaný řádek jednoznačně určuje výpočtový krok stroje $\mathcal{M}$.

Jako ilustraci interpretujme krok i, kdy je stroj ve stavu q, čte j–tý symbol na pásce: s, přejde do stavu q', přepíše s na s' a přejde na políčko pásky $j+1$. To koresponduje s pokrytím řádku i: j–tý kachlíček je typu  , $(j+1)$

kachlíček je typu 

, zbývající kachlíčky na řádku jsou typu 

.

Na druhou stranu je třeba nahlédnout, že každé správně vykachlíčkované síti jednoznačně odpovídá výpočet stroje $\mathcal{M}$. Skutečně, v každém řádku vykachlíčkované sítě je právě jeden kachlíček s „horní“ barvou (q, s). Navíc v celé síti existuje jediná souvislá cesta, která se skládá z těchto kachlíčků, které se dotýkají alespoň jedním rohem. Cesta začíná v levém horním kachlíčku a končí v levém dolním kachlíčku. Důkaz je u konce. □

Cvičení 12.1 Dokažte důsledně tvrzení, že v každém řádku je právě jeden kachlíček s „horní“ barvou (q, s).

Poznámka 12.7 Pokud bychom měli polynomiální algoritmus na kachličkování, věděli bychom, že pro každý problém ze třídy NP existuje polynomiální algoritmus. Polynomiální algoritmus bychom byli schopni zkonstruovat ve chvíli, kdy bychom znali nedeterministický algoritmus i s polynomiálním odhadem jeho rychlosti.

12.4 Další *NP*–úplné problémy

S *NP*–úplnými problémy se setkáváme v nejrůznějších oblastech matematiky, ale i v praxi. Abychom ověřili, že daný problém je *NP*–úplný musíme podle definice ukázat že patří do třídy *NP* a že každý problém z *NP* je na něj polynomiálně transformovatelný. První podmínka u rozhodovacích problémů je většinou triviální, a proto její ověření je přenecháno čtenáři. Podmínku druhou zrealizujeme polynomiální transformací ze známého *NP*–úplného problému ($\propto$ je tranzitivní!).

Metodu polynomiálních transformací budeme ilustrovat na problému rozhodnutí, zda daná booleovská formule v konjunktivní normální formě je splnitelná (SAT). Booleovská formule Φ v konjunktivně normální formě se skládá z konjunkce klauzulí, kde každá klauzule obsahuje disjunkci několika proměnných (mohou být jako negace i „gace“). Například booleovská formule

$$\Phi(x_1, x_2, x_3) = (x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2) \wedge (\neg x_2 \vee x_3)$$

se skládá ze tří klauzulí a tří proměnných x_1, x_2, x_3. Formule Φ je splnitelná, jestliže existuje pravdivostní přiřazení 1(true) a 0(false) proměnným tak, že každá klauzule je vyhodnocena jako 1(true). V našem příkladě je formule Φ splnitelná, viz např. přiřazení $x_1 = x_2 = = x_3 = 1$. Samozřejmě je k dispozici 2^n pravdivostních přiřazení. Problém SAT je rozhodnout, zda daná formule v konjunktivně disjunktivním tvaru je splnitelná.

Věta 12.4 *SAT–problém splnitelnosti booleovských formulí je NP–úplný.*

Důkaz: SAT jistě patří do NP. (Hádáme ověření x lineární velikosti a vyhodnocujeme v lineárním čase formuli v konjunktivně disjunktivním tvaru.)

Důkaz NP–těžkosti provedeme polynomiální transformací z problému KACHL. Převedeme vstupní instanci I_K problému KACHL na vstupní instanci I_S problému SAT v polynomiálním čase a ověříme, že $I_K \in$ KACHL právě když $I_S \in$ SAT. Protože problém KACHL je NP–těžký, vyplyne odtud NP–těžkost problému SAT.

Buď $I_K = \langle B, K, S\rangle$ instance problému KACHL. Zaveďme proměnné $x_{i,j,k}$, které budou vyjadřovat možnost umístění kachlíčku k na místo i, j sítě S. Sestrojíme následující formule:

1. $\bigwedge_{i,j} (\bigvee_k x_{i,j,k})$, abychom zaručili, že alespoň jeden kachlíček bude umístěn v každém čtverci sítě S (rozměru $n \times n$),
2. $\bigwedge_{i,j} \bigwedge_{k \neq k'} (\neg x_{i,j,k} \vee \neg x_{i,j,k'})$, abychom zaručili, že nanejvýš jeden kachlíček bude umístěn v každém čtverci sítě S,
3. $\bigwedge_{i,j} \bigwedge_{k,k'} (\neg x_{i,j,k} \vee \neg x_{i,j+1,k'})$ pro všechny dvojice kachlíčků k, k', které nelze umístit vedle sebe v horizontálním směru – tím zaručujeme podmínku sousednosti v horizontálním směru,
4. $\bigwedge_{i,j} \bigwedge_{k,k'} (\neg x_{i,j,k} \vee \neg x_{i+1,j,k'})$ pro všechny dvojice kachlíčků k, k', které nelze umístit vedle sebe ve vertikálním směru – tím zaručujeme podmínku sousednosti ve vertikálním směru,
5. $\neg x_{1,j,k}$ pro všechny kachlíčky, které nemohou být umístěny na pozici $(1, j)$ – tím zaručujeme podmínku levé hranice
6. $\neg x_{n,j,k}$ pro všechny kachlíčky, které nemohou být umístěny na pozici (n, j) – tím zaručujeme podmínku pravé hranice
7. $\neg x_{i,1,k}$ pro všechny kachlíčky, které nemohou být umístěny na pozici $(i, 1)$ – tím zaručujeme podmínku horní hranice
8. $\neg x_{i,n,k}$ pro všechny kachlíčky, které nemohou být umístěny na pozici (i, n) – tím zaručujeme podmínku dolní hranice

Formule Φ je konjunkce formulí 1,...,8. Tímto způsobem jsme v polynomiálním čase zkonstruovali instanci I_S problému SAT. Řešení problému KACHL jednoznačně odpovídá řešení problému SAT, tedy $I_K \in$ KACHL právě když $I_S \in$ SAT. (V řeči jazyků.) □

V následující kapitole je uveden přehled nejznámějších *NP*–úplných problémů.

13 Příklady *NP*-úplných problémů a transformací

1. **KACHLíčkování**
   (a) Orientované - viz minulá přednáška.
   (b) neorientované - kachlíčky můžeme i otáčet.
   (c) průhledné - kachlíčky můžeme i převracet.
   (d) s jednobarevnou hranicí - každá hrana koupelny je jednobarevná.

2. **SAT**
   (a) **SAT**istability: Je dána formule v konjunktivně disjunktivním tvaru

$$\Phi(x) = (\bigwedge_i \bigvee_j a_{i,j}) \text{ , kde } a_{i,j} = \begin{cases} \neg x_k \\ x_k \end{cases},$$

pro $k \in \{1, \ldots, n\}$.

   Otázka: $\exists\, x = (x_1, x_2, \ldots, x_n)$ tak, že $\Phi(x)$?
   (b) **3SAT**: Je dána formule v konjunktivně disjunktivním tvaru

$$\Phi(x) = (\bigwedge_i \bigvee_{j=1}^{3} a_{i,j}) \text{ , kde } a_{i,j} = \begin{cases} \neg x_k \\ x_k \end{cases},$$

pro $k \in \{1, \ldots, n\}$.

   Otázka: $\exists\, x = (x_1, x_2, \ldots, x_n)$ tak, že $\Phi(x)$?
   (c) **3–3SAT**: Je dána formule v konjunktivně disjunktivním tvaru

$$\Phi(x) = (\bigwedge_i \bigvee_{j=1}^{l_i} a_{i,j}) \text{ , kde } a_{i,j} = \begin{cases} \neg x_k \\ x_k \end{cases},$$

pro $k \in \{1, \ldots, n\}$,
každé x_k použito nejvýš 3–krát,
$l_i \leq 3$

   Otázka: $\exists\, x = (x_1, x_2, \ldots, x_n)$ tak, že $\Phi(x)$?

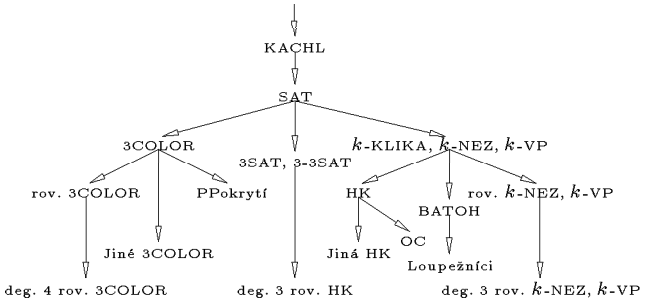
3. **KLIKA, NEZávislá množina, Vrcholové Pokrytí**
   (a) **KLIKA**: Je dán neorientovaný graf G a číslo k. Otázka: Existuje v G **klika** velikosti k? (klika: podgraf, v němž jsou každé dva vrcholy spojeny hranou)
   (b) **NEZávislá množina**: Je dán neorientovaný graf G a číslo k. Otázka: Existuje v G **nezávislá množina** velikosti k? (nezávislá množina: indukovaný podgraf, v němž není hrana)
   (c) Totéž, ale o grafu G víme, že je **rovinný**.
   (d) Totéž, ale graf G je **rovinný** a všechny vrcholy mají **stupeň nejvýš 3**.
   (e) **VP**: Je dán neorientovaný graf G a číslo k. Otázka: Existuje v G **vrcholové pokrytí** velikosti k? (vrcholové pokrytí: taková množina vrcholů, že každá hrana grafu obsahuje aspoň jeden její vrchol)

4. **HK**
   (a) **HK**: Je dán neorientovaný graf G. Otázka: Existuje v G **Hamiltonovská kružnice**? (Hamiltonovská kružnice: kružnice procházející každým vrcholem právě jednou)
   (b) **OHK**: Totéž, ale graf i **Hamiltonovská kružnice** jsou **orientované**.
   (c) **HC, OHC**: Totéž, ale dotaz na **Hamiltonovskou cestu**.
   (d) **HC/HK → HC/HK, OHC/OHK → → OHC/OHK**: Dán graf a v něm Hamiltonovská cesta/kružnice, dotaz na existenci jiné Hamiltonovské cesty/kružnice.
   (e) Orientovaný graf na vstupu je bipartitní, rovinný, v jedné partitě platí $\deg^- = 1$ a $\deg^+ = 2$ a v druhé partitě $\deg^- = 2$ a $\deg^+ = 1$, dotaz na **OHK/OHC**.
   (f) **OC**: Je dán ohodnocený graf G a číslo c. Otázka: Existuje v G sled procházející všemi vrcholy (ne nutně jednou), končící ve vrcholu ve kterém začal, jehož součet ohodnocení je nejvýš c?

5. **BATOH**
   (a) Je dáno přirozené číslo t a seznam S obsahující n přirozených čísel. Čísla jsou kódována binárně. Otázka: Existuje podseznam $S' \subseteq S$ tak, že $\sum_{s \in S'} s = t$?
   (b) **Loupežníci**: Je dán seznam S obsahující n (binárně kódovaných) přirozených čísel. Otázka: Existuje podseznam $S' \subseteq S$ tak, že $\sum_{s \in S'} s = \sum_{t \in S \setminus S'} t$?

6. **COLOR**
   (a) **COLOR**ing: Je dán neorientovaný graf G a číslo k. Otázka: Je možno graf G obarvit k barvami? (obarvení grafu: vrcholy spojené hranou musí mít různou barvu)
   (b) Totéž, ale předem víme, že $k = \mathbf{3}$.
   (c) Totéž, ale předem víme, že graf G je **rovinný**.
   (d) Totéž, ale předem víme, že **stupeň** libovolného vrcholu grafu je **nejvýš 4**.
   (e) Je dáno jedno obarvení třemi barvami, otázka na netriviálně jiné.

7. **PPokrytí**:
   (a) Máme $3n$ tříprvkových množin, jejich „multisjednocením“ je množina obsahující $3n$ různých prvků, každý třikrát. Otázka: Je možno vybrat n množin tak, aby jejich sjednocením byla množina obsahující všech $3n$ různých prvků?



Obr. 20: Graf převodů mezi *NP*–úplnými problémy

Na obrázku 20 je zobrazen graf jednoduchých převodů mezi NP–úplnými problémy.

Na obrázcích 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, a 32 jsou načrtnuty konstrukce některých polynomiálních transformací.

Ostatní uvedené transformace jsou téměř triviální, s výjimkou transformace 3COLOR na přesné pokrytí. O většině těchto transformací se můžete dočíst v knížce Luďka Kučery – Kombinatorické algoritmy, nebo v knížce Jána Plesníka – Grafové algoritmy.

Úlohu SAT bychom pomocí algoritmu na řešení 3–SAT mohli řešit takto: Postupně nahradíme každou disjunkci aspoň 4 formulí

$$a_1 \vee a_2 \vee a_3 \vee \cdots \vee a_k$$

konjunkcí s novou prvoformulí x

$$(a_1 \vee a_2 \vee x) \wedge (\neg x \vee a_3 \vee \cdots \vee a_k).$$

Formule se konstrukcí prodlouží nejvýš konstanta–krát a je splnitelná právě když původní formule je splnitelná. Pokud bychom byli pedantičtí, doplnili bychom ještě „krátké“ disjunkce „zdvojováním“. □

Úlohu 3SAT bychom pomocí 3–3SAT mohli řešit takto: Místo každé prvoformule x_k vyskytující se v formuli více než 3–krát vytvoříme prvoformule x_k^i a nahradíme i–tý výskyt prvoformule x_k prvoformulí x_k^i. Navíc přidáme konjunkci

$$(x_k^1 \vee \neg x_k^2) \wedge (x_k^2 \vee \neg x_k^3) \wedge \cdots \wedge (x_k^{l-1} \vee \neg x_k^l) \wedge (x_k^l \vee \neg x_k^1),$$

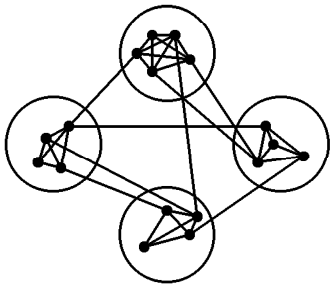
kde l označuje počet výskytů proměnné x_k. Přidaná konjunkce vynucuje stejné ohodnocení prvoformulí x_k^i (pro pevné k). Každému ohodnocení prvoformulí, pro něž je původní formule pravdivá jednoznačně odpovídá ohodnocení prvoformulí nové formule, pro něž je nová prvoformule pravdivá. Formule se prodloužila nejvýš konstanta–krát. Pozor nemůžeme použít „zdvojování“, abychom zajistili délku 3 každé disjunkce. □

K jednotlivým obrázkům:

21 Nechť formule je tvaru

$$\Phi(x) = (\bigwedge_{i=1}^{m} \bigvee_{j=1}^{l_i} a_{i,j}) \text{ , kde } a_{i,j} = \begin{cases} \neg x_k \\ x_k \end{cases},$$

pro $k \in \{1, \ldots, n\}$.

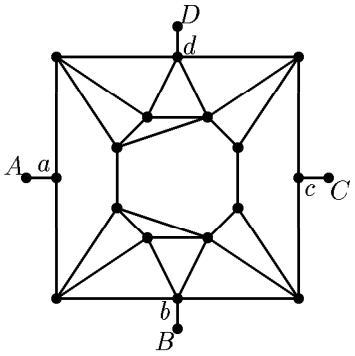


Obr. 21: Převod SAT na nezávislou množinu velikosti m

Pak obrázek obsahuje m množin po l_i vrcholech označených $a_{i,j}$.

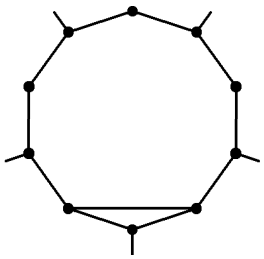
Pro právě jedno k symbolu $a_{i,j}$ odpovídá symbol x_k resp. symbol $\neg x_k$. My toto x_k resp. $\neg x_k$ využijeme ke konstrukci grafu. Vrcholy uvnitř množin jsou pospojovány hranami. Vrcholy $a_{i,j}$, $a_{i',j'}$ jsou spojeny hranou, pokud $i \neq i'$ a symbolický zápis $a_{i,j} \wedge a_{i',j'}$ je tautologicky false (gace a negace téže proměnné).

Problém jsme převedli na dotaz na nezávislou množinu velikosti m v zkonstruovaném grafu. Technickou záležitostí je dokázat přesnou korespondenci mezi odpověďmi na otázku první úlohy a odpověďmi na otázku druhou.



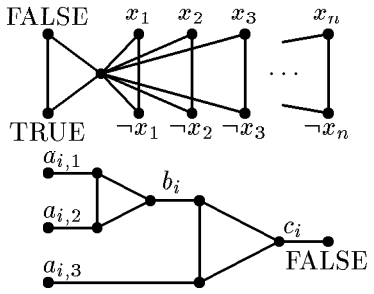
Obr. 22: Převod k nezávislá na l rovinná nezávislá

22 Nahrazením křížení hran $(A, C), (B, D)$ zobrazeným podgrafem zvětší velikost největší nezávislé množiny grafu právě o 6.



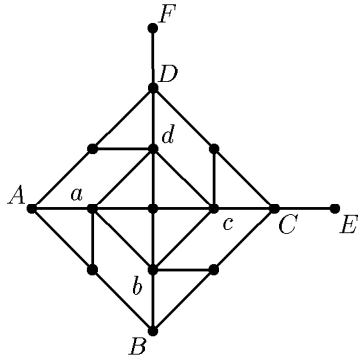
Obr. 23: Převod rovinná k nezávislá na rovinnou l nezávislou se stupni nevýš 3

23 Nahrazením vrcholu v grafu stupně většího než tři podgrafem podle tohoto schématu, se velikost největší nezávislé množiny zvětší právě o $\deg(v) - 1$.



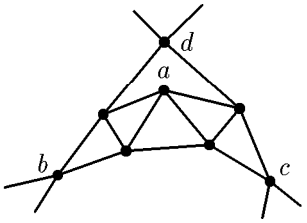
Obr. 24: Převod 3SAT na 3COLOR

24 Označme barvu vrcholu FALSE za false a barvu vrcholu TRUE za true. Graf zobrazený v horní části obrázku zajišťuje, že vrcholům x_k jsou přiřazeny pravdivostní hodnoty. Každé klauzuli odpovídá podgraf zobrazený ve spodní části obrázku. Vrcholy libovolného trojúhelníčku jsou obarveny třemi různými barvami. Pokud jsou všechny tři barvy $a_{i,j}$ false, pak je b_i jediný vrchol trojůhelníčku, jemuž není zakázána barva false. Nutně musí být b_i obarven false, ale v trojúhelníčku obsahujícím c_i je barva false zakázána všem vrcholům. Proto tento podgraf není možno správně doobarvit. Pokud máme správné obarvení grafu, pak barvy vrcholů určují řešení x problému 3SAT. Pokud existuje x, řešení problému 3SAT, pak jsou jím určeny barvy všech vrcholů s výjimkou dolních podgrafů. Víme ale, že v každém takovém podgrafu je jeden z vrcholů $a_{i,j}$ obarven true. Tento podgraf je ale v tom případě možno doobarvit.



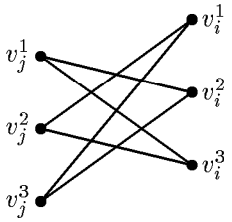
Obr. 25: Převod 3COLOR na rovinné 3COLOR

25 Nahrazením křížení hran $(A, E), (B, F)$ zobrazeným podgrafem sníží počet křížení a zachová vlastnost 3COLOR. (Vrcholy a, c mají stejnou barvu, vrcholy b, d mají stejnou barvu. Barva a není stejná jako barva b. Nemá–li A barvu b, pak B má stejnou barvu jako A ... vrcholy A, B, C, D mají stejnou barvu. Má–li A barvu d, pak D má barvu a=c, ... A, C mají barvu b a B, D mají barvu a. Vrcholy A, C mají stejnou barvu, vrcholy B, D mají stejnou barvu. Tyto barvy A, B jsou nezávislé.)



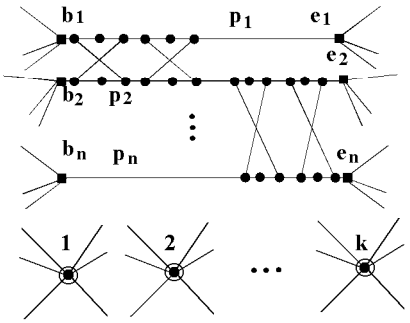
Obr. 26: Převod rovinné 3COLOR na rovinné 3COLOR se stupni nejvýš 4

26 Nahrazením vrcholu v zobrazeným podgrafem sníží součet stupňů vrcholů stupně většího než 4 a zachová vlastnost 3COLOR (vrcholy a, b, c, d mají stejnou barvu).

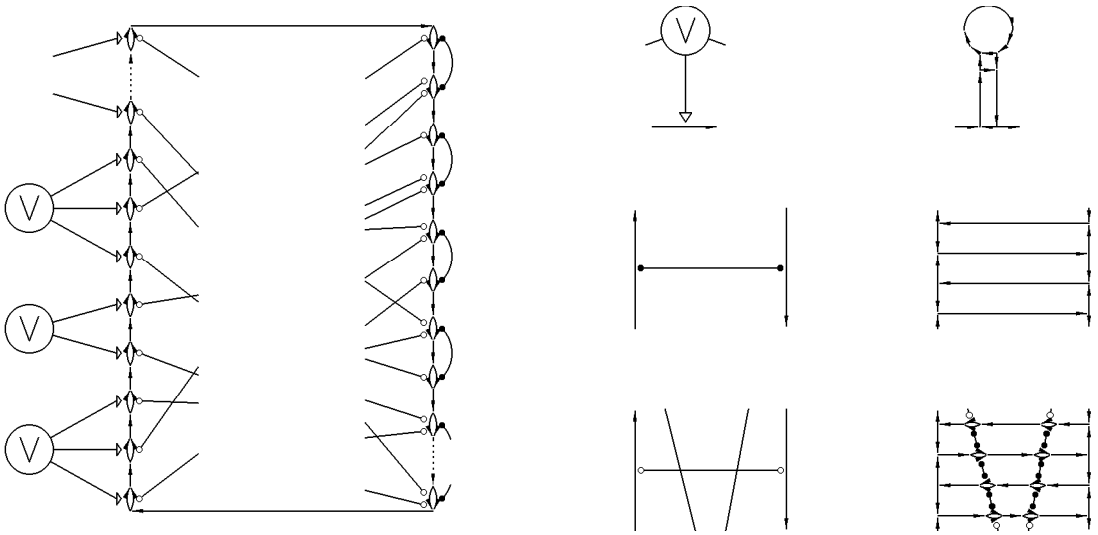


Obr. 27: Převod 3COLOR na hledání 3COLOR jiného než zadaného

27 Hraně $\{i, j\}$ původního souvislého grafu odpovídá šest hran nového grafu $(\{v_i^\ell, v_j^k\}$, kde $k \neq \ell)$. V novém grafu je triviální obarvení „po vrstvách“ $\ldots v_i^\ell$ má barvu c_ℓ. Pokud však existuje netriviální obarvení nového grafu, pak je možno obarvení původního grafu získat na základě hlasování. (Barvu vrcholu i určíme hlasováním z $\{c(v_i^1), c(v_i^2), c(v_i^3)\}$.)



Obr. 28: Převod k Vrcholového pokrytí na Hamiltonovskou kružnici



Obr. 29: Převod **SAT** na HK v rovinném bipartitním grafu s omezenými stupni

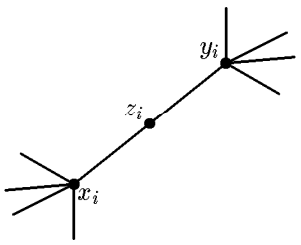
28 $i-$tému vrcholu grafu odpovídá cesta $p_i = b_i, \ldots, e_i$, hraně grafu odpovídá podgraf o 6+6 vrcholech, jehož příkladem je podgraf zobrazený na začátcích cest p_1, p_2 resp. na koncích cest p_2, p_n. Úseky cesty p_i vytnuté podgrafy reprezentujícími různé hrany $(i,j), (i,l)$ jsou disjunktní.

Číslo k je reprezentováno vrcholy $1 \ldots k$. Každý vrchol b_i resp. e_i je spojen s vrcholy $1 \ldots k$. Vrcholy b_i, e_i, jsou na konci konstrukce z grafu odstraněny tak, že jsou ztotožněny s prvním resp. posledním vnitřním vrcholem cesty. Pokud cesta p_i neměla žádný vnitřní vrchol, pak vrchol i byl izolovaný, můžeme vrcholy b_i, e_i z grafu odstranit (včetně hran do vrcholů $1, \ldots, k$). V nově zkonstruovaném grafu existuje Hamiltonovská kružnice právě když v původním grafu bylo vrcholové pokrytí velikosti k.

(Výběru vrcholového pokrytí jednoznačně odpovídá výběr prošlých dvojic vrcholů b_i, e_i. Na Hamiltonovské kružnici totiž leží buď oba nebo žádný, protože Hamiltonovská kružnice procházející levým vrcholem 6+6 podgrafu musí tento podgraf opustit pravým vrcholem 6+6 tice ležícím na téže cestě p_i. Všechny vrcholy 6+6 podgrafu mohou být na Hamiltonovské kružnici, právě pokud je aspoň jedna z odpovídající dvojice cest vybrána. Existuje–li vrcholové pokrytí, pak existuje Hamiltonovská kružnice, ale i opačně z nalezené Hamiltonovské kružnice jsme schopni sestrojit vrcholové pokrytí.)

29 Konstrukce se skládá ze „základního cyklu“, kde v levé části jsou simulovány disjunkce, pravá část slouží k „označení“ jednotlivých proměnných. Každý „dvouobloučekˇ“ odpovídá proměnné, průchod „vnějším“ obloučkem koresponduje s true, na rozdíl od průchodu „vnitřním“ obloučkem. „Vnějškem“ pravé části je zajištěno, aby $2i$-tá proměnná byla negací $(2i-1)$-ní proměnné. Vnitřek kružnice zajišťuje souvislost proměnných s jejich výskytem v disjunkcích. Na pravé části obrázku je rozkleslen „disjunktor“ a nonekvivalence. Vespod pravé části je naznačeno jak díky přidání dalších „proměnných“ můžeme zachovat rovinnost konstrukce. Uvědomte si, že nonekvivalence jsou vždy uvnitř „cyklicky orientované stěny“.

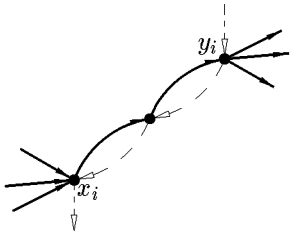
Vzniklý graf je rovinný, bipartitní, vrcholy jedné partity mají in-degree 2 a out-degree 1, u vrcholů druhé partity je to obráceně.



Obr. 30: Převod OHK na HK

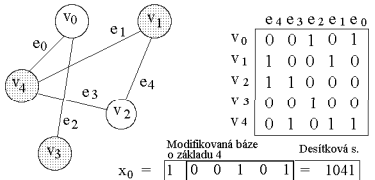
30 $i-$tému vrcholu grafu odpovídá podgraf G_i, hraně (i,j) grafu odpovídá hrana $\{y_i, x_j\}$. HK vždy obsahuje hrany $\{y_i, z_i\}, \{z_i, x_i\}$. Určením hran $\{y_i, x_j\}$ je HK definována. Odpovídající OHK původního grafu obsahuje hranu (i,j) právě když HK obsahuje hranu $\{y_i, x_j\}$. Libovolné HK v zkonstruovaném grafu odpovídá OHK v grafu původním, a libovolné OHK v původním grafu odpovídá HK v zkonstruovaném grafu.

31 $i-$tému vrcholu grafu odpovídá podgraf G_i, hraně (i,j) grafu odpovídá hrana (y_i, x_j). Kromě toho jsou přidány hrany (x_i, y_{i+1}) pro $i = 1 \ldots n-1$ a hrana (x_n, y_1)



Obr. 31: Převod OHK na hledání jiné OHK než OHK zadané

(kde n je počet vrcholů původního grafu). Podgraf G_i má vlastnost, že hamiltonovská kružnice jím projde najednou a použije buď pouze hrany nakreslené tučně, nebo pouze hrany nakreslené čárkovaně. Hamiltonovská kružnice proto nemůže obsahovat najednou jak tučné tak čárkované hrany. V zkonstruovaném grafu existuje právě jedna OHK používající pouze čárkované hrany. Každé OHK v původním grafu odpovídá OHK používající pouze tučné hrany, a opačně každé OHK používající pouze tučné hrany odpovídá OHK v původním grafu. Odpověď na existenci druhé OHK v zkonstruovaném grafu odpovídá na existenci nějaké OHK v původním grafu.



	e4	e3	e2	e1	e0
v0	0	0	1	0	1
v1	1	0	0	1	0
v2	1	1	0	0	0
v3	0	0	1	0	0
v4	0	1	0	1	1

	Modifikovaná báze o základu 4						Desítková s.
x_0 =	1	0	0	1	0	1	= 1041
x_1 =	1	1	0	0	1	0	= 1284
x_2 =	1	1	1	0	0	0	= 1344
x_3 =	1	0	0	1	0	0	= 1040
x_4 =	1	0	1	0	1	1	= 1093
y_0 =	0	0	0	0	0	1	= 1
y_1 =	0	0	0	0	1	0	= 4
y_2 =	0	0	0	1	0	0	= 16
y_3 =	0	0	1	0	0	0	= 64
y_4 =	0	1	0	0	0	0	= 256
t =	3	2	2	2	2	2	= 3754

Obr. 32: Vrcholové pokrytí s řešením v_1, v_3, v_4 a odpovídající instance BATOHu s řešením x_1, x_3, x_4, y_0, y_2, y_3, y_4

32 Klíčovou myšlenkou ke konstrukci instance problému BATOHU je použití incidenční matice grafu G. Buď $G = (E = \{e_0, \ldots, e_{|E|-1}\}, V = \{v_0, \ldots, v_{|V|-1}\})$ graf, incidenční matice G je $|V| \times |E|$ matice $B = (b_{ij})$, kde

$$b_{ij} = \begin{cases} 1 & \text{hrana } e_j \text{ sousedí s vrcholem } v_i, \\ 0 & \text{jinak.} \end{cases}$$

Množina S bude obsahovat dva typy čísel, které budou odpovídat hranám a vrcholům. Tato čísla budeme interpretovat v soustavě o základu 4. Pro každý vrchol $v_i \in V$ sestrojíme přirozené číslo x_i, které bude mít reprezentaci $(1, b_{i,0}, \ldots, b_{i,|E|-1})$, tj.

$$x_i = 4^{|E|} + \sum_{j=0}^{|E|-1} b_{ij} 4^{|E|-j-1}.$$

Pro každou hranu e_j sestrojíme přirozené číslo $y_j = 4^j$, jehož reprezentace má kromě nul právě jednu jedničku na místě e_j. Konečně

$$t = k \cdot 4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^j.$$

Všechna zkonstruovaná čísla mají velikost omezenou polynomem ve velikosti vstupní instance problému VP.

Zbývá ověřit, že graf G má vrcholové pokrytí velikosti k právě když existuje $S' \subseteq S$ a $\sum_{s \in S'} s = t$.

Předpokládejme, že graf G má vrcholové pokrytí $V' = \{v_{i_1}, \ldots, v_{i_k}\}$. Definujeme $S' = \{x_{i_1}, \ldots, x_{i_k}\} \cup \cup \{y_j \mid e_j \text{ sousedí s právě jedním vrcholem z } V'\}$.

Snadno zjistíme, že $\sum_{s \in S'} s = t$. Skutečně, součtem k čísel vrcholů a libovolného počtu čísel hran dostaneme číslo, v jehož zápisu v soustavě o základu 4 je na pozicích řádů aspoň $4^{|E|}$ číslo k. Dále zjistíme, že ostatní váhově nižší bity jsou rovny 2. Uvažme postupně bitové pozice, které odpovídají hranám e_j. Sousedí–li e_j se dvěma vrcholy z vrcholového pokrytí V', pak oba tyto vrcholy přispívají do součtu 1, ale $y_j \notin S'$ nepřispívá. V případě, že e_j sousedí jen s jedním vrcholem z V', tento vrchol přispívá do celkového součtu bitové pozice jednou 1; druhá 1 je v řádku příslušejícím $y_j \in S'$.

Nyní předpokládejme, že existuje $S' \subseteq S$ tak, že $\sum_{s \in S'=1} s = t$. Nechť $S = \{x_{i_1}, \ldots, x_{i_m}\} \cup \cup \{y_{i_1}, \ldots, y_{i_p}\}$. Pak ale $k = m$ a $V' = \{v_{i_1}, \ldots, v_{i_m}\}$ je vrcholové pokrytí G. Opravdu, na pozici e_j v bitové reprezentaci čísel z S jsou právě tři 1: jednou přispívá y_j a po jedné koncové vrcholy e_j. Vzhledem k číselné soustavě o základu 4 není možný přenos z bitové pozice e_j do pozice e_{j+1}. Takže pro $|E|$ váhově nejnižších pozic čísla t platí, že aspoň jedno a nanejvýš dva x_i přispívají do součtu na každé takové bitové pozici. Proto je V' vrcholové pokrytí.

Cvičení 13.1 Proveďte naznačené důkazy podrobně a pokuste se dokázat NP–úplnost ostatních uvedených problémů.

14 Pseudo–polynomiální algoritmy a silná *NP*–úplnost

V tomto oddíle prozkoumáme číselné problémy podrobněji. Vraťme se k problému BATOH, o kterém jsme si v minulém oddíle dokázali, že je *NP*–těžký. Buď $\langle S = \{c_1, \ldots, c_n\}, t\rangle$ instance problému BATOH. Uvažme algoritmus Alg. 13 pro problém BATOHu:

---

begin
  Zkonstruuj orientovaný graf $G = (V, A)$, kde
    $V := \{0, 1, \ldots, t\}$ a $A = A_1 \cup A_2 \cup \ldots \cup A_n$,
    kde $A_j = \{(u, v) \in V^2 \mid u - v = c_j\}$
  Označ uzel 0 (O:=0)
  for $j = 1$ **to** n **do**
    for $v \in O$ **do**
      označ uzel u takový, že $(u, v) \in A_j$
  BATOH má řešení právě když uzel t je označený.
end

Alg. 13: Pseudopolynomiální algoritmus řešící problém BATOH

Tento algoritmus je příkladem známé metody návrhu algoritmů, které se říká *dynamické programování*.

Lemma 14.1 *Nechť M_j je množina uzlů označených po j–tém opakování kroku 3 předchozího algoritmu. Pak $M_j = \{v \in V \mid$ existuje množina $S' \subseteq \{1, \ldots, j\}$ taková, že $\sum_{i \in S'} c_i = v\}$.*

Důkaz: Dokážeme indukcí podle j. V j–té iteraci je

$$M_j = M_{j-1} \cup \{u \mid \exists v \in M_{j-1} \text{ tak, že } u = v + c_j\}.$$

Užitím indukčního předpokladu na M_{j-1} a použitím předešlého vztahu dostaneme dokazovaný výsledek. □

Věta 14.2 *předchozí algoritmus řeší problém BATOH v čase $O(n \cdot t)$.*

Důkaz: Správnost algoritmu bezprostředně vyplývá z předchozího lemmatu. Krok 3 vyžaduje čas $O(t)$, neboť nanejvýš t uzlů může být označeno. Celkem je n iterací. □

Uvědomme si, že algoritmus *není* polynomiální v případě, když vstupní čísla v instanci BATOHu *nejsou* kódována *unárně*! Algoritmům, které jsou polynomiální jen v případě zakódování instance v unární soustavě říkáme *pseudopolynomiální*.

Existují však číselné problémy, které jsou *NP*–úplné bez ohledu na zakódování vstupu. Příkladem takového problému je např. problém 3–ROZKLAD. Vstupem problému je $3m$ přirozených čísel a naším úkolem je rozhodnout zda existuje rozdělení těchto čísel do m množin tak, že každá množina obsahuje právě 3 čísla a součet čísel v každé množině je totožný.

Zaveďme si následující pojmy. Nechť I je instance výpočetního problému. Definujeme number(I) jako největší číslo, které se v I vyskytuje.

Například je jistě rozumné předpokládat, že v instanci I problému baťoh je number(I) = t.

Buď Π výpočetní problém a f funkce zobrazující přirozená čísla na sebe. Označme Π_f problém Π zúžený na instance I takové, že number(I) $\leq f(|I|)$. Řekneme, že problém Π je *silně NP*–úplný, jestliže Π_p je *NP*–úplný pro nějaký polynom p.

Tak například problém KLIKA je silně *NP*–úplný, neboť např. pro $k < n$ (v n–vrcholovém grafu může být klika velikosti maximálně n) je jistě $k =$ number(I) $< |V|$.

Problém 3–ROZKLAD je taktéž silně *NP*–úplný. Avšak například problém BATOH do této kategorie nespadá.

Všimněte si, že pseudopolynomiální algoritmus lze charakterizovat takto: každou instanci I řeší v čase, který je polynomiální v $|I|$ a v number(I).

Věta 14.3 *Za předpokladu $P \neq NP$ pro žádný silně NP–úplný problém neexistuje pseudopolynomiální algoritmus.*

Důkaz: Nechť Π je silně *NP*–úplný problém, tj. $\Pi_{p(n)}$ je *NP*–úplný problém pro jistý polynom p. Předpokládejme, že existuje polynomiální algoritmus $\mathcal{A}$, který řeší v polynomiálním čase $q(|I|, \text{number}(I))$ instanci I problému Π. Pak také řeší v polynomiálním čase $q(n, p(n))$ i *NP*–úplný problém $A_{p(n)}$, spor. □

15 Aproximace *NP*–úplných problémů

Mnoho problémů, které mají zvláštní praktickou důležitost je *NP*–úplných, a proto je pošetilé pro velké instance hledat optimální řešení úloh s nimi spojených.

Nicméně v případě malých vstupních instancí lze použít i exponenciální algoritmus. Na druhé straně i v případě velkých vstupních instancí se můžeme spokojit s hledáním „skoro“ optimálních řešení, které lze nalézt v polynomiálním čase, ať už v nejhorším anebo v průměrném případě. Takovým algoritmům budeme říkat *polynomiální aproximační* algoritmy.

Předpokládejme, že hledáme řešení optimalizační úlohy, kde možné řešení má kladnou hodnotu a chceme nalézt skoro optimální řešení.

Řekneme, že aproximační algoritmus řeší úlohu s chybou $\rho(n)$, když pro každý vstup velikosti n je cena řešení C, kterou nalezl aproximační algoritmus, vůči optimálnímu řešení C^* ve vztahu

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

Tato definice je v platnosti jak pro minimalizační tak i pro maximalizační verze optimalizačních úloh. Je tedy $\rho(n) \geq \geq 1$ a hodnotu 1 nabývá právě když aproximační algoritmus nalezne optimální řešení.

Někdy je výhodnější pracovat s tzv. *relativní chybou* $\varepsilon(n)$ aproximačního algoritmu

$$\frac{|C - C^*|}{C^*} \leq \varepsilon(n).$$

Pak platí, že $\varepsilon(n) \leq \rho(n) - 1$, a v případě minimalizačních verzí nastává rovnost. U některých aproximačních algoritmů jsou funkce $\rho(n), \varepsilon(n)$ konstanty. V takovém případě budeme argument n vynechávat.

Pokusme se navrhnout polynomiální aproximační algoritmus pro optimalizační verzi problému vrcholového pokrytí grafu G. V optimalizační verzi, označme ji VPM, hledáme vrcholové pokrytí minimální mohutnosti.

Nechť $G = (V, E)$ je daný graf. Uvažme algoritmus Alg. 14:

```
begin
  C := ∅;
  while E ≠ ∅ do
      begin
        Nalezni vrchol v v G,
        který má největší stupeň;
        odstraň jej z V a přidej do C;
      end
  vrať C jako vrcholové pokrytí G
end
```

Alg. 14: Aproximace vrcholového pokrytí I

Jak je tento algoritmus dobrý? Ukážeme, že vždy můžeme najít příklad grafu, na kterém selže – relativní chyba je aspoň $O(\log n)$.

Sestrojme graf G takto. Začneme s n disjunktními hranami $\{c_i, b_i\}$. Pak b–uzly rozdělíme na dvojice a každou dvojici spojíme s novým vrcholem a (není–li b dělitelné dvěma necháme zbytek volný). Pak b–uzly rozdělíme na trojice a každou trojici spojíme s novým uzlem a. To opakujeme ... až spojíme jednu $(n-1)$–tici b–uzlů s posledním vrcholem a. Označme počet a–uzlů $\alpha(n)$. Aproximační algoritmus do C zařadí do vrcholového pokrytí všechny a– a c–uzly, tj. najde vrcholové pokrytí velikosti $\alpha(n) + n$, zatímco minimální vrcholové pokrytí tvoří b–uzly a má proto velikost n. Protože $\alpha(n) = \sum_{j=2}^{n-1} \lfloor n/j \rfloor$, je relativní chyba úměrná $n-1$ harmonickému číslu a je tedy $O(\log n)$.

Při navrhování aproximačních algoritmů je třeba postupovat uvážlivěji.

Věta 15.1 *Existuje polynomiální aproximační algoritmus, který řeší optimalizační problém VPM minimálního vrcholového pokrytí grafu s relativní chybou 1.*

Důkaz: Nechť $G = (V, E)$ je daný graf. Nalezneme polynomiální algoritmus, který nalezne vrcholové pokrytí grafu G, které je při nejhorším dvakrát větší než minimální pokrytí.

```
begin
  C := ∅;
  E' := E;
  while E' ≠ ∅ do
      begin
        Nechť (u, v) je libovolná hrana E';
        C := C ∪ {u, v};
        Odstraň z E' každou hranu
        incidentní s u nebo v
      end
  Vrať C jako vrcholové pokrytí G
end
```

Alg. 15: Aproximace vrcholového pokrytí II

Tento algoritmus je typu „greedy“. Jeho výpočtový čas je $O(|E|)$ a konstruuje vrcholové pokrytí grafu G, neboť každá hrana z E je pokryta nějakým vrcholem z C. Označme M množinu těch hran, které byly vybrány v kroku 4. Tyto hrany jsou navzájem disjunktní. Tedy v kroku 5 přidáváme k C vždy dva nové vrcholy. Proto platí $|C| = 2 \cdot |M|$. Optimální minimální vrcholové pokrytí C^* grafu G musí však pokrývat každou hranu z M, tj. musí obsahovat alespoň jeden koncový vrchol takové hrany. Tedy $|M| \leq \leq |C^*|$ a $|C| \leq 2 \cdot |C^*|$. □

V dalším budeme konstruovat aproximační algoritmus pro problém obchodního cestujícího.

Buď G ohodnocený úplný graf, kde ohodnocení vyhovuje trojúhelníkové nerovnosti. Postupujme podle algoritmu Alg. 16.

begin
Nalezni minimální kostru T grafu G;
Utvoř multigraf M vzniklý z kostry T zdvojením hran;
Nalezni Eulerovský tah E v M;
Vrať E jako cestu obchodního cestujícího.
end

Alg. 16: Aproximace problému obchodního cestujícího I

Věta 15.2 *Tento algoritmus řeší problém obchodního cestujícího s relativní chybou 1.*

Důkaz: Předchozí algoritmus je zřejmě polynomiální. Multigraf M je Eulerovský neboť je souvislý a má všechny stupně sudé. Nechť funkce γ je cenová funkce příslušných grafů a C^* cena optimálního řešení. Pak $2 \cdot \gamma(T) = \gamma(M) = \gamma(E)$. Protože $\gamma(T) \leq C^*$ platí $\cdot\gamma(E) \leq 2c^*$. □

Poznamenejme, že existují instance, které vedou k relativní chybě právě 1. Nalezněte je.

Aproximační algoritmus pro obchodního cestujícího s trojúhelníkovou nerovností lze zlepšit, ovšem na úkor výpočetního času, viz. algoritmus Alg. 17.

Předpokládejme, že ohodnocení hran vstupního grafu vyhovuje trojúhelníkové nerovnosti. To je případ například běžné verze problému, kdy je vstupní instance vnořena do roviny a vzdálenosti jsou dány Euklidovskou metrikou. Poznamenejme, že optimalizační (minimalizační) verze problému s trojúhelníkovou nerovností je stále *NP*–těžká, neboť lze na ni transformovat problém Hamiltonovské kružnice. Skutečně daný graf G přetransformujeme na vstup obchodního cestujícího takto: G doplníme na úplný graf a staré hrany ohodnotíme 1, zatímco přidané 2. Nyní je zřejmé, že řešení obchodního cestujícího má hodnotu n právě když G obsahuje Hamiltonovskou kružnici.

begin
Nalezni minimální kostru T grafu G;
Nalezni minimální úplné párování U
na uzlech lichého stupně kostry T;
Nechť $M = T \cup U$ je multigraf,
který vznikne sjednocením T s U;
Nalezni Eulerovský tah E v M;
Vrať E jako cestu obchodního cestujícího.
end

Alg. 17: Aproximace problému obchodního cestujícího II

Algoritmus je polynomiální a lze jej realizovat v čase $O(n^3)$.

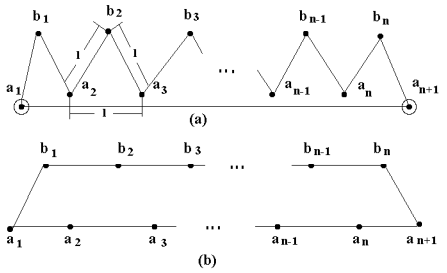
Věta 15.3 *Předchozí algoritmus řeší problém obchodního cestujícího s relativní chybou $\frac{1}{2}$.*

Důkaz: Graf M je Eulerovský, neboť je souvislý a vrchol sudého stupně v T je také sudého stupně v M a vrchol lichého stupně v T díky párování U je v $M = T \cup U$ stupně sudého. Buď C^* cena optimálního řešení, γ cenová funkce příslušných grafů. Pak

$$\gamma(E) \leq \gamma(M) = \gamma(T) + \gamma(U).$$

Dále platí, že $\gamma(T) \leq C^*$.

Označme $\{i_1, \ldots, i_{2m}\}$ množinu vrcholů lichého stupně z T v pořadí jak se poprvé objevují na optimální Hamiltonovské cestě C^*. Cesta C^* má tedy tvar $[\alpha_0 i_1 \alpha_1 \ldots \alpha_{2m-1} i_{2m} \alpha_{2m}]$, kde α_i jsou posloupnosti (třebas prázdné) vrcholů grafu G. Uvažme dvě párování: $M_1 = \{[i_1, i_2], [i_3, i_4], \ldots, [i_{2m-1}, i_{2m}]\}$, $M_2 = \{[i_2, i_3], [i_4, i_5], \ldots, [i_{2m}, i_1]\}$. Díky trojúhelníkové nerovnosti dostaneme $C^* \geq \geq \gamma(M_1) + \gamma(M_2)$. Protože U je minimální párování, je také $C^* \geq 2 \cdot \gamma(U)$. □



Obr. 33: Graf pro nějž je chyba naší aproximace úlohy OC maximální

Opět existuje příklad grafu, pro který je relativní chyba předešlého algoritmu $\frac{1}{2}$, srovnej Obr. 33. V části (a) je znázorněn graf na $2n + 1$ vrcholech. Hrana (a_1, a_{n+1}) má délku n, ostatní hrany délku 1. Minimální kostra je cesta $(a_1, b_1, \ldots, b_n, a_{n+1})$. Minimální párování tvoří jediná hrana (a_1, a_{n+1}), která spojuje jediné dva vrcholy lichého stupně. Graf je Eulerovský a celková cena je $3n$. V části (b) Obr. 33 je ale ukázáno optimální řešení ceny $2n$.

15.1 Úplně polynomiální aproximační schémata

Některé *NP*–úplné problémy (formulované jako optimalizační úlohy) připouštějí aproximační algoritmy, při kterých se relativní chyba ε snižuje na úkor zvýšení výpočetního času. Ideálně by tato závislost měla být konstantní, tj. výpočtový čas by měl být lineární v $1/\varepsilon$. My se spokojíme s tím, když bude výpočtový čas polynomiální v $1/\varepsilon$ i v n (např. $(1/\varepsilon^2)n^3$). Takovým algoritmům budeme říkat *úplně polynomiální* aproximační schémata, zkracujeme UPAS.

Sestrojíme UPAS pro optimalizační verzi problému BATOH. V této verzi je naším úkolem najít podmnožinu S' indexů takovou, že $\sigma = \sum_{i \in S'} c_i$ je maximalizována za podmínky, že $\sigma \leq t$. Připomeňme, že chceme nalézt algoritmus, který je polynomiální jak v relativní chybě $1/\varepsilon$, tak i v n. Budeme potřebovat následující notaci.

Nechť L je seznam, $L + x$ je seznam, který vznikne z L přičtením čísla x ke každému prvku. Podobně $S + x = \{s + x \mid s \in S\}$. Utříděný seznam prvků menších než x, který vznikne spojením dvou utříděných seznamů L, L' označíme $L \bigotimes L'|_x$.

Nejdříve uvedeme exponenciální algoritmus BAT(S, t):

```
begin
  n := |S|; L_0 := ⟨0⟩
  for i := 1, ..., n do
      L_i := L_{i-1} ⊗ (L_{i-1} + c_i)|_t
  Vrať největší prvek z na seznamu L_n
end
```

Alg. 18: Exponenciální algoritmus pro problém BATOH

Nechť P_i je množina všech možných součtů (včetně 0) podmnožin čísel z $\{c_1, \ldots, c_i\}$. Platí tedy $P_i = P_{i-1} \cup (P_{i-1} + c_i)$. Pak ale L_i je utříděný (vzestupně) seznam, který obsahuje všechny prvky z P_i, které nejsou větší než t. Délka seznamu $|L_i| \leq 2^i$, a proto je algoritmus BAT exponenciální. Nicméně převedeme jej na UPAS.

Klíčovou myšlenkou je využít „zkrácení" seznamu L faktorem δ. Pomocí $L^\delta, 0 < \delta < 1$ označíme seznam, který vznikne z L odstraněním maximálního počtu prvků tak, aby platila následující vlastnost:

$$y \in L \setminus L^\delta \Rightarrow \exists\, z \in L^\delta \text{ a } (1-\delta)y \leq z \leq y.$$

Říkáme, že $z \in L^\delta$ je reprezentantem těch $y \in L$, pro které platí $(1-\delta)y \leq z \leq y$. Následující procedura TRIM(L, δ) provede δ–zkrácení vzestupně setříděného seznamu $L = \langle y_1, \ldots, y_m \rangle$ v čase $\Theta(m)$:

```
procedure Trim(L, δ)
begin
  m := |L|; L' := ∅; first := -∞;
  for i ∈ {1, ..., m} do
      if first < (1-δ)y_i then
          first := y_i, L' := L' ∪ first;
  Vrať L'
end
```

Alg. 19: procedura *Trim*

Nyní už můžeme UPAS pro problém batohu sestavit pro $0 < \varepsilon < 1$ následovně:

```
1  begin
2    n := S; L_0 := ⟨0⟩;
3    for i = 1, ..., n do
4        begin
5          L_i := L_{i-1} ⨂ (L_{i-1} + c_i)|_t;
6          L_i := Trim(L_i, ε/n);
7        end
8    Vrať největší prvek z na seznamu L_n
9  end
```

Alg. 20: UPAS pro problém BATOH

Cvičení 15.1 Dokažte, že

$$1 - \varepsilon < (1 - \varepsilon/n)^n .$$

Návod: Porovnejte rozvoje logaritmů.

$$\ln(1-x) = -\sum_{i=1}^{\infty} \frac{x^i}{i} \text{ pokud má pravá strana smysl}$$

Věta 15.4 *Algoritmus Alg. 20 je úplným polynomiálním aproximačním schématem pro optimalizační verzi problému BATOH.*

Důkaz: Řádky 6 a 7 zachovávají invariant, že $L_i \subseteq P_i$. A tak hodnota z vrácená v řádku 9 je součtem jisté podmnožiny množiny S. Zbývá ukázat, že $y^*(1-\varepsilon) \leq z$, kde y^* označuje optimální řešení, a že algoritmus je polynomiální v $1/\varepsilon$ a n.

Nejprve nahlédneme, že relativní chyba algoritmu je malá. Při zkracování seznamu L_i je relativní chyba nanejvýš ε/n (mezi zbývajícími a vyškrtanými prvky). Indukcí dostaneme, že

$$\forall\, y \in P_i, y_i \leq t\ \exists\, z \in L_i \text{ tak, že } (1-\varepsilon/n)^i y \leq z \leq y.$$

Tedy speciálně

$$(1-\varepsilon/n)^n y^* \leq z \leq y^*.$$

Podle předchozího cvičení $1 - \varepsilon < (1-\varepsilon/n)^n$, a tak $(1 - -\varepsilon)y^* \leq z$.

Protože výpočtový čas algoritmu je úměrný součtu délek seznamů L_i, stačí odhadnout $|L_i|$.

Pro dva po sobě jdoucí prvky $z, z' \in L_i$ platí:

$$\frac{z}{z'} > \frac{1}{1-\varepsilon/n}.$$

Odtud a z rozvoje příslušného logaritmu vyplývá, že $|L_i|$ je nanejvýš

$$\log_{\frac{1}{1-\varepsilon/n}} t = \frac{\ln t}{-\ln(1-\varepsilon/n)} \leq \frac{n \ln t}{\varepsilon}.$$

Celkový čas je nejvýš $\frac{n^2 \ln t}{\varepsilon}$. □

Cvičení 15.2 Zdůvodněte, proč není možno v proceduře TRIM ponechávat jakožto reprezentanta „blízkých mezisoučtů" mezisoučet největší. (Zkonstruujte protipříklad, ukažte, že ani postup se seřazenými c_i nefunguje).

16 Třída #P, #P–úplné úlohy

V předchozích přednáškách jsme se zabývali rozhodovacími problémy, třídou **NP**. Připomeňme definici: Formálně můžeme třídu NP definovat jako třídu jazyků vyjádřitelných předpisem

$$\mathcal{L} = \left\{ x \mid \exists^P y \; R^P(x,y) \right\}. \tag{1}$$

Aby byl zřejmý význam symbolů $\exists^P$ a R^P, popíšeme tento předpis naprosto přesně: Pro dané dva polynomy $P_1(n)$, $P_2(n)$ a predikát $R(x,y)$ vyčíslitelný v čase $P_2(|x|+|y|)$ je $\mathcal{L}$ jazyk takových x, pro něž existuje „ověření“ y, $|y| \leq \leq P_1(|x|)$, pro nějž je hodnota predikátu $R(x,y)$ pravda.

O třídě co–NP jsme dosud příliš nemluvili. co–NP znamená “complement nondeterministic polynomial”. Do této třídy patří problémy lišící se od NP–problémů pouze znegováním otázky. Mezi typické zástupce co–NP problémů patří například problémy:

1 Je pravda, že v daném grafu neexistuje klika dané velikosti k?

2 Jsou nutné 4 barvy na obarvení daného rovinného grafu?

Formálně můžeme třídu co–NP definovat jako třídu jazyků vyjádřitelných předpisem

$$\mathcal{L} = \left\{ x \mid \forall^P y \; R^P(x,y) \right\} \tag{2}$$

Samozřejmě za předpokladu „P=NP“ bychom uměli stejně rychle hledat pozitivní i negativní odpovědi. Proto by bylo „NP=P=coNP“.

Třída **#P** je třída úloh, jejichž výsledkem je přirozené číslo. Tuto úlohu bychom mohli formálně definovat takto: Cílem je spočítat pro daný vstup x

$$V_x = \left| \left\{ y \mid |y| \leq P_1(|x|) \mid R^P(x,y) \right\} \right| \tag{3}$$

Neboli pro dané dva polynomy $P_1(n)$, $P_2(n)$, a predikát $R(x,y)$ vyčíslitelný v čase $P_2(|x|+|y|)$ je úlohou pro vstup x určit, kolik existuje „ověření“ y, $|y| \leq P_1(|x|)$, pro nějž je hodnota predikátu $R(x,y)$ pravda.

Na první pohled je vidět, že známe–li V_x — počet ověření predikátu R, umíme rozhodnout rozhodovací problémy „$V_x > 0$?“, „$V_x = 2^{P_1(|x|)} + 2^{P_1(|x|)-1} + \ldots$?“. Jinými slovy umíme řešit rozhodovací problémy rozhodující o příslušnosti x do jazyků (1), (2).

I pro třídu #P se snažíme nalézt „nejtěžší“ úlohy této třídy. Vzhledem k tomu, že hledáme počet ověření, nemůžeme používat polynomiální transformace resp. redukce, které mění neznámým způsobem počet ověření. Můžeme používat takové transformace, které počet ověření nemění—„parsimonious“, a transformace, u nichž známe funkci popisující vztah mezi počty ověření.

Ukážeme, že početní verze problému kachličkování je #P–úplná.

Věta 16.1 *Spočítat počet různých vykachlíčkování koupelny (podle dříve popsaných pravidel) je #P-m-těžká úloha.*

Důkaz: Věta 12.1 ukazuje, že pro polynom P_1 a polynomiálně vypočítatelný predikát $R(x,y)$ existuje nedeterministický Turingův stroj pracující v polynomiálním čase, jehož počet přijímacích výpočtů je roven počtu ověření y, $|y| \leq P_1(|x|)$. Mějme nedeterministický Turingův stroj těchto vlastností. Nechť má jednoznačnou koncovou konfiguraci, v níž deterministicky setrvává. Stačí si uvědomit, že v důkazu věty 12.3 jsme zkonstruovali koupelnu a kachlíčky tak, že správné vykachlíčkování koupelny jednoznačně odpovídá jednomu přijímacímu výpočtu NTS. □

Poznámka 16.1 Uvědomme si co znamená x a co znamená y: x je zadání úlohy, tedy tvar a obarvení stěn koupelny a katalog kachlíčků, y je výběr kachlíčků pro jednotlivá políčka.

Věta 16.2 *#SAT je #P-m-úplná úloha. (Počet ohodnocení prvotních formulí, pro něž je formule v konjunktivně disjunktivním tvaru splněna.)*

Důkaz: Při důkazu věty 12.4 jsme použili „parsimonious“ transformaci z problému kachlíčkování. □

Věta 16.3 *#3–SAT je #P-m-úplná úloha.*

Důkaz: Ukážeme, jak pomocí #3–SAT vyřešíme #SAT. Postupně nahradíme každou disjunkci aspoň 4 formulí

$$a_1 \vee a_2 \vee a_3 \vee \cdots \vee a_k$$

konjunkcí s novou prvoformulí x

$$(a_1 \vee a_2 \vee x) \wedge (\neg a_1 \vee \neg x) \wedge (\neg a_2 \vee \neg x) \wedge (\neg x \vee a_3 \vee \cdots \vee a_k).$$

Od transformace z kapitoly 13 se tato liší přidáním konjunkce $(\neg a_1 \vee \neg x) \wedge (\neg a_2 \vee \neg x)$, zajišťující, aby hodnota prvoformule x byla určena jednoznačně. K zakončení důkazu je třeba si uvědomit, že formule se celou konstrukcí prodloužila konstanta–krát. □

Věta 16.4 *#k–klik, #k–nezávislých množin, #(n-k) vrcholových pokrytí jsou #P-m-úplné úlohy.*

Důkaz: Počet nezávislých množin velikosti k je stejný jako počet vrcholových pokrytí velikosti $n-k$ (doplněk konkrétní NM je VP). Počet nezávislých množin velikosti k je stejný jako počet klik velikosti k v grafu, kde hrany jsou nahrazeny nehranami (tytéž množiny vrcholů).

Ukážeme, jak pomocí #k–klik spočítat počet řešení 3–SAT $\exists\, x_1, \ldots, x_n \varphi(x_1, \ldots, x_n)$, kde

$$\varphi = \bigwedge_{i=1}^{m} (a_{i,1} \vee a_{i,2} \vee a_{i,3}),$$

pro $a_{i,j}$ gace nebo negace prvoformulí x_l.

Formuli φ můžeme formálně přepsat do tvaru

$$\varphi = \bigwedge_{i=1}^{m} ((a_{i,1}\; a_{i,2}\; a_{i,3}) \vee (a_{i,1}\; a_{i,2} \neg a_{i,3}) \vee$$
$$(a_{i,1} \neg a_{i,2}\; a_{i,3}) \vee (a_{i,1} \neg a_{i,2} \neg a_{i,3}) \vee (\neg a_{i,1}\; a_{i,2}\; a_{i,3}) \vee$$

$$(\neg a_{i,1}\ a_{i,2}\neg a_{i,3}) \vee (\neg a_{i,1}\neg a_{i,2}\ a_{i,3})),$$

kde „trojice“ uvnitř závorek jsou logické součiny.

Vytvoříme graf skládající se z m sedmic vrcholů a n dvojic vrcholů, vrchol i–té sedmice je označen některou „trojicí“ s prvním indexem i. Vrcholy i–té dvojice jsou označeny x_i a $\neg x_i$.

Dva vrcholy jsou spojeny hranou, pokud logický součin jejich „trojic“ není tautologicky false.

Vidíme, že vrcholy uvnitř sedmice nejsou spojeny hranou a $m+n$–klice v uvedeném grafu jednoznačně odpovídá ohodnocení prvoformulí, pro něž je $\varphi = \mathsf{true}$. □

Věta 16.5 *#k–klik, #k–nezávislých množin, #(n-k) vrcholových pokrytí jsou #P-m-úplné úlohy i v případě, kdy víme, že pro jakékoli větší k problém nemá řešení.*

Důkaz: Stačí si uvědomit, že v důkazu vznikaly transformací pouze grafy u nichž víme, že pro větší k úloha řešení nemá. □

Věta 16.6 *Spočítat počet perfektních párování v bipartitním grafu je #P-T-úplná úloha. (Každý vrchol v právě jedné vybrané hraně.)*

Poznámka 16.2 Toto je první příklad, kde nalézt jedno ověření je jednoduché, ale spočítat je všechna je těžké. Předtím uvedené příklady byly těžké už jako rozhodovací problémy.

(Nalézt řešení je jednoduchá aplikace algoritmu na toky v celočíselných sítích.)

Než tuto větu dokážeme, ukážeme si některé jí ekvivalentní úlohy.

Věta 16.7 *Spočítat počet možných rozestavení věží na předvyznačená políčka šachovnice tak, aby se vzájemně neohrožovaly je #P-T-úplná úloha.*

Věta 16.8 *Spočítat počet rozkladů orientovaného grafu na cykly je #P-T-úplná úloha.*

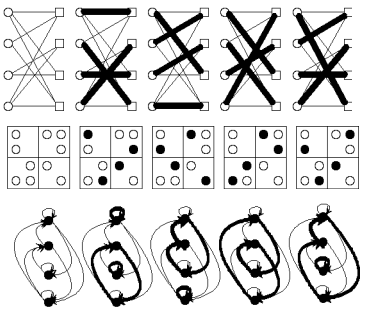
Na obrázku 34 je ukázána vzájemná korespondence mezi párováním, rozmísťováním věží a rozkladem grafu na cykly. (Jedničky na místech matice sousednosti bipartitního grafu (vrcholy jedné partity indexují řádky, vrcholy druhé partity indexují sloupce) označují přípustná políčka na položení věží, tatáž matice slouží jako incidenční matice orientovaného grafu.)

Definice 16.1 Permanent matice $A = (a_{i,j})$ typu $n \times n$ je definován předpisem

$$\mathrm{Perm}\, A = \sum_{\pi \in S_n} \prod_{i=1}^{n} a_{i,\pi(i)}$$

Permanent 0—1 matice je roven počtu možných rozmístění neohrožujících se věží na políčka označená 1. Jiná, ekvivalentní formulace věty 16.6 je věta 16.9:

Věta 16.9 *Spočítat permanent 0—1 matice je #P-T-úplná úloha.*



Obr. 34: Párování, rozmísťování věží a rozklad grafu na cykly

Důkaz věty 16.9 rozdělíme do pěti tvrzení:

Lemma 16.10 *Spočítat počet rozkladů orientovaného grafu na cykly délky větší než 2 je #P-m-úplná úloha.*

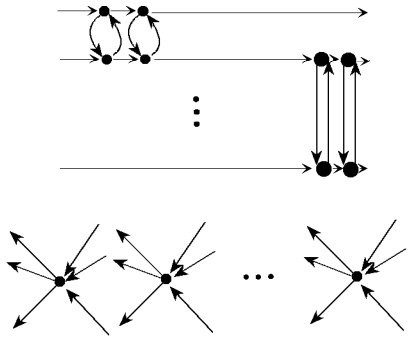
Lemma 16.11 *Spočítat permanent matice s čísly -1, $-\frac{1}{2}$, 0, $\frac{1}{2}$, 1 je #P-m-těžká úloha.*

Lemma 16.12 *Spočítat permanent matice s čísly -2, -1, 0, 1, 2 je #P-m-těžká úloha.*

Lemma 16.13 *Spočítat permanent celočíselné matice modulo součin polynomiálně mnoha prvočísel z počátečního úseku prvočísel je #P-m-těžká úloha.*

Lemma 16.14 *Spočítat permanent celočíselné matice modulo „polynomiálně velké“ prvočíslo je #P-T-těžká úloha.*

Poznámka 16.3 Všimněte si, že v důkazu lemmatu 16.14 se nepoužívá transformace, ale redukce.



Obr. 35: Převod #VP na #orientovaných HK a na #rozkladů na cykly delší než 2

Lemma 16.10 má blízkou souvislost s následující větou.

Věta 16.15 *Spočítat počet Hamiltonovských kružnic je #P-m-úplná úloha nezávisle na tom, zda se jedná o orientovaný nebo neorientovaný graf.*

Důkaz *–16.15*: V kapitole 13 je na obr. 28 uvedena konstrukce, jak vytvořit k danému grafu neorientovaný „graf cest“, v němž výběru k–vrcholového pokrytí ($k-1$ vrcholové pokrytí neexistuje) odpovídá výběr cest. Vybrané cesty můžeme mnoha způsoby spojit v hamiltonovskou kružnici. Možností, jak vybrané cesty spojit do hamiltonovské kružnice, je $2^{k-1}k!(k-1)!$. (Zafixujme jednu cestu, máme $k!$ pořadí výběrů pomocných vrcholů, $(k-1)!$ pořadí výběrů ostatních cest a 2^{k-1} orientací ostatních cest.) Ze znalosti počtu Hamiltonovských kružnic bychom uměli zjistit počet k–vrcholových pokrytí (vydělením číslem $2^{k-1}k!(k-1)!$).

Na obrázku 35 je naznačena konstrukce, jak vytvořit orientovaný „graf cest“, v němž výběru k–vrcholového pokrytí ($k-1$ vrcholové pokrytí neexistuje) opět odpovídá výběr cest. Možností, jak vybrané cesty spojit v hamiltonovskou kružnici, je nyní $k!(k-1)!$, protože orientace je již jednoznačně určena. (Počet vrcholových pokrytí získáme vydělením číslem $k!(k-1)!$.) □

Důkaz *–16.10*: Pokud chceme uvedený orientovaný „graf cest“ rozložit na cykly delší než 2, je opět jednoznačná korespondence mezi „vybranými“ cestami a vybraným k–vrcholovým pokrytím (pokud $k-1$ vrcholové pokrytí neexistuje). Možností, jak vybrané cesty doplnit na cykly, je $(k!)^2$ (můžeme cestám přiřadit libovolné pořadí vstupních a výstupních pomocných vrcholů). (Počet vrcholových pokrytí získáme vydělením číslem $(k!)^2$.) □

Důkaz *–16.11*: Ukážeme, jak pomocí permanentu matice s čísly -1, $-\frac{1}{2}$, 0, $\frac{1}{2}$, 1 počítat počet rozkladů orientovaného „grafu cest“ na cykly delší než 2.

Vezmeme si matici sousednosti „grafu cest“ a nahradíme každou „hranovou“ podmatici (velikosti 4×4) podmaticí podle níže uvedeného schématu:

$$\begin{pmatrix} \ddots & \vdots & 0 & \vdots & 0 & \vdots \\ 0 & 0 & 1 & 1 & 0 & 0 \\ \cdots & 0 & 0 & 0 & 1 & \cdots \\ 0 & 1 & 0 & 0 & 1 & 0 \\ \cdots & 0 & 1 & 0 & 0 & \cdots \\ \cdots & \vdots & 0 & \vdots & 0 & \ddots \end{pmatrix} \to \begin{pmatrix} \ddots & \vdots & 0 & \vdots & 0 & \vdots \\ 0 & 1 & 1 & -1 & 0 & 0 \\ \cdots & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 0 & \cdots \\ 0 & 0 & 0 & 0 & 1 & 0 \\ \cdots & 1 & -1 & 0 & 0 & \cdots \\ \cdots & \vdots & 0 & \vdots & 0 & \ddots \end{pmatrix}$$

Je potřeba si uvědomit, že příslušný permanent odpovídá počtu rozkladů na cykly delší než 2. □

Důkaz *–16.12*: Kdybychom uměli spočítat permanent každé -2, -1, 0, 1, 2 – matice, uměli bychom permanent matice s čísly -1, $-\frac{1}{2}$, 0, $\frac{1}{2}$, 1 počítat vynásobením prvků matice dvěma a vydělením výsledku 2^n, kde n je rozměr matice. □

Důkaz *–16.13*: Stačí si uvědomit, že celé číslo, na jehož binární zápis nám stačí polynomiálně mnoho bitů, můžeme shora odhadnout součinem polynomiálně mnoha prvočísel z počátečního úseku prvočísel. (Určitě jich stačí tolik, kolik je bitů.)

Na spočítání celého čísla od -2^k do $+2^k$ můžeme použít modulární aritmetiku modulo součin příslušných prvočísel. Máme spočítat číslo od $-2^n n!$ do $2^n n!$. Přitom $2^n n! < 2^n \cdot n^{O(1)} \cdot \frac{n^n}{e^n} < 2^n \cdot 2^{n\log n} < 2^{n^3}$. Uměli bychom spočítat permanent -2, -1, 0, 1, 2 – matice. □

Důkaz *–16.14*: Ukážeme, jak spočítat permanent modulo součin polynomiálně mnoho prvočísel z počátečního úseku prvočísel redukcí na výpočet modulo jednotlivá prvočísla.

Potřebujeme si uvědomit, že n–té prvočíslo je polynomiálně velké vůči n. To plyne z hustoty prvočísel $\frac{c}{\log n}$. (Do n^2 je asi $\frac{cn^2}{\log n} > n$ prvočísel.)

Dále je třeba si uvědomit, že ze znalosti výsledku modulo jednotlivá prvočísla jsme schopni rychle spočítat výsledek modulo jejich součin. Pracujeme s aritmetikou potřebující polynomiální počet bitů!

Nechť $q \equiv q_i \pmod{p_i}$ a nechť $r_i \cdot \prod_{j\neq i} p_j \equiv 1 \pmod{p_i}$. Potom

$$q \equiv \sum q_i \cdot r_i \prod_{j\neq i} p_j \pmod{\prod p_i}.$$

(K ověření vztahu stačí zkontrolovat, že

$$q_i \equiv \sum q_k \cdot r_k \prod_{j\neq k} p_j \pmod{p_i},$$

protože vektor zbytků modulo jednotlivá prvočísla je jednoznačný.) □

Důkaz *–16.9*: Ukážeme, jak pomocí permanentu z 0–1 matice spočítat permanent matice s malými celými nezápornými čísly ($< p$). Celkový součet čísel v matici je nejvýš pn^2. Místo každého čísla k většího než 1 vložíme k řádků a sloupců podle následujícího schématu (kde $k=3$):

$$\begin{pmatrix} & \cdots & b & \cdots \\ \vdots & \ddots & \vdots & \vdots \\ a & \cdots & 3 & \cdots \\ \vdots & \cdots & \vdots & \ddots \end{pmatrix} \to \begin{pmatrix} & \cdots & b & s_1 & s_2 & s_3 & \cdots \\ \vdots & \ddots & \vdots & 0 & 0 & 0 & \cdots \\ a & \cdots & 0 & 1 & 1 & 1 & \cdots \\ r_1 & 0 & 1 & 1 & 0 & 0 & 0 \\ r_2 & 0 & 1 & 0 & 1 & 0 & 0 \\ r_3 & 0 & 1 & 0 & 0 & 1 & 0 \\ \vdots & \cdots & \vdots & 0 & 0 & 0 & \ddots \end{pmatrix}$$

Vytvořili jsme 0–1 matici s nejvýš $n+pn^2$ řádky a sloupci, jejíž permanent je stejný jako permanent původní matice. □