1. Hradlové sítě

(zapsali Jirka Fajfr a Ján Černý)

Na této přednášce se budeme zabývat jednoduchým modelem paralelního počítače, totiž hradlovou sítí, a ukážeme si alespoň jeden efektivní paralelní algoritmus, konkrétně sčítání dvojkových čísel v logaritmickém čase vzhledem k jejich délce.

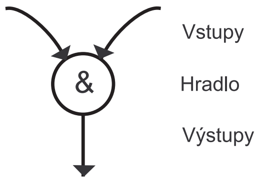
Hradlové sítě

Definice: *Hradlo* je zařízení, které počítá nějakou pevně danou funkci s k vstupy a jedním výstupem.

Příklad: Obvykle pracujeme s booleovskými hradly, ta pak odpovídají funkcím $f : \{0,1\}^k \to \{0,1\}$. Z nich nejčastěji potkáme:

- 0-vstupové: to jsou konstanty TRUE a FALSE,
- 1-vstupové: identita (ta je vcelku k ničemu) a negace (značíme $\neg$),
- 2-vstupové: logický součin (AND, &) a součet (OR, $\vee$).

Hradla kreslíme třeba následovně:



Hradlo provádějící logickou operaci AND se dvěma vstupy

Z jednotlivých hradel pak vytváříme hradlové sítě. Pokud používáme pouze booleovská hradla, říkáme takovým sítím *booleovské obvody,* pokud operace nad nějakou obecnější (ale konečnou) množinou symbolů (abecedou), nazývají se *kombinační obvody.* Každý vstup hradla je připojen buďto na některý ze vstupů sítě nebo na výstup nějakého jiného hradla. Výstupy hradel mohou být propojeny na výstupy sítě nebo přivedeny na vstupy dalších hradel, přičemž je zakázáno vytvářet cykly. Než si řekneme formální definici, podívejme se na obrázek.

TODO: *OBR*

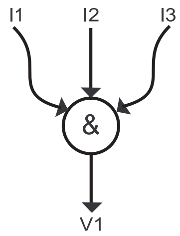
Definice: *Hradlová síť* je určena:

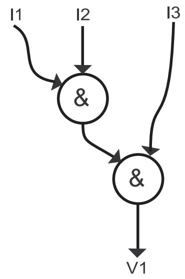
- *abecedou* Σ (to je nějaká konečná množina symbolů, obvykle $\Sigma = \{0,1\}$);
- množinou *hradel* H, *vstupních portů* I a *výstupních portů* O;
- acyklickým orientovaným grafem (V, E), kde $V = H \cup I \cup O$;
- zobrazením F, které každému hradlu $h \in H$ přiřadí nějakou funkci $F(h) : \Sigma^{a(h)} \to \Sigma$. To je funkce, kterou toto hradlo vykonává, a číslu $a(h)$ říkáme *arita* hradla h;
- zobrazením $z : E \to \mathbb{N}$, které každé hraně vedoucí do nějakého hradla přiřazuje některý ze vstupů tohoto hradla.

Přitom jsou splněny následující podmínky:

- $\forall i \in I : \deg^+(i) = 0$ (do vstupů nic nevede);
- $\forall o \in O : \deg^+(o) = 1$ & $\deg^-(o) = 0$ (z výstupů nic nevede a do každého vede právě jedna hrana);
- $\forall h \in H : \deg^+(v) = a(v)$ (do každého hradla vede tolik hran, kolik je jeho arita);
- $\forall h \in H, 1 \le j \le a(h)$ existuje právě jeden vrchol v takový, že $z(vh) = j$ (všechny vstupy hradel jsou zapojeny).

Pozorování: Kdybychom připustili hradla s libovolně vysokým počtem vstupů, mohli bychom libovolný problém se vstupem délky n vyřešit jedním hradlem o n vstupech, což není ani realistické, ani pěkné. Proto přijměme omezení, že všechna hradla budou mít maximálně k vstupů, kde k je nějaká pevná konstanta, obvykle dvojka. Následující obrázky ukazují, jak hradla o více vstupech nahradit dvouvstupovými:

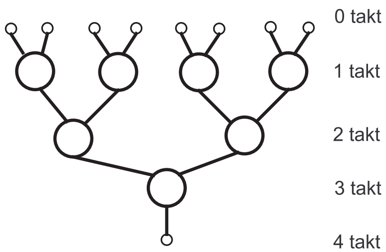




Trojvstupové hradlo AND

Jeho nahrazení 2-vstupovými hradly

Definice: *Výpočet sítě* probíhá v *taktech.* V nultém taktu jsou definovány právě hodnoty vstupních portů. V i-tém taktu vydají výsledek hradla, která jsou připojena na porty nebo na výstupy hradel, jejichž hodnota byla definována v $(i-1)$-ním taktu. Až po nějakém konečném počtu taktů budou definované i hodnoty výstupních portů, síť se zastaví a vydá výsledek.



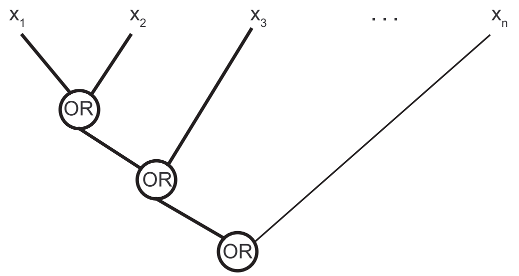
Výpočet hradlové sítě

Podle toho, jak síť počítá, si ji můžeme rozdělit na vrstvy:

Definice: *i-tá vrstva* obsahuje všechny vrcholy v takové, že nejdelší z cest z portů sítě do v má délku právě i. To jsou přesně vrcholy, které vydají výsledek poprvé v i-tém taktu výpočtu. Dává tedy smysl prohlásit za *časovou složitost* sítě počet jejích vrstev. Podobně *prostorovou složitost* definujeme jako počet hradel v síti.

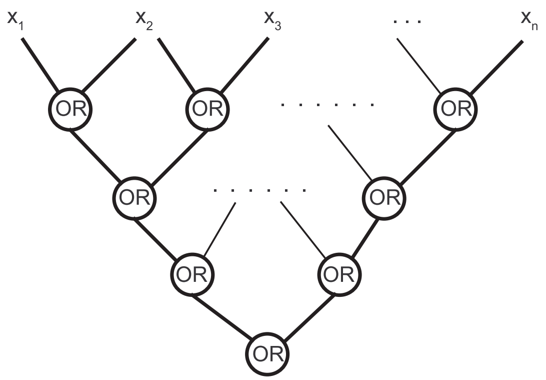
Příklad: Sestrojte síť, která zjistí, zda se mezi jejími n vstupy vyskytuje alespoň jedna jednička.

První řešení: zapojíme hradla za sebe (sériově). Časová a prostorová složitost jsou n. Zde vůbec nevyužíváme toho, že by mohlo počítat více hradel současně.



Hradlová síť, která zjistí zda-li je na vstupu alespoň jedna jednička

Druhé řešení: Budeme vrcholy spojovat do dvojic, pak výsledky z těchto dvojic opět do dvojic a tak dále. Tak dosáhneme časové složitosti $\Theta(\log n)$, prostorová složitost zůstane lineární.



Chytřejší řešení stejného problému

Sčítání binárních čísel

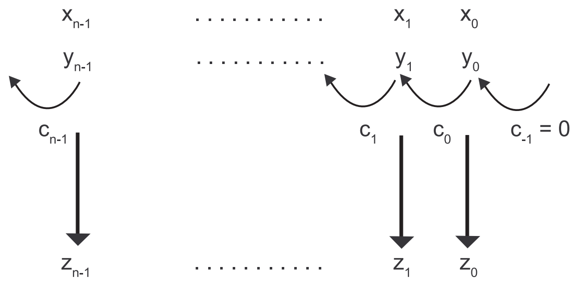
Pojďme se podívat na zajímavější problém: Mějme dvě čísla zapsané ve dvojkové soustavě jako $x_{n-1} \ldots x_1 x_0$ a $y_{n-1} \ldots y_1 y_0$. Budeme chtít spočítat jejich součet $z_n z_{n-1} \ldots z_1 z_0$.

Samozřejmě můžeme použít algoritmus „sčítání pod sebou“, který nás učili na základní škole. Formálně by se dal zapsat třeba takto:

$$z_i = x_i \oplus y_i \oplus c_{i-1},$$

kde $\oplus$ značí operaci XOR (součet modulo 2) a c_{i-1} je *přenos* z $(i-1)$-ního řádu do i-tého. Přenos přitom nastane tehdy, když ze tří xorovaných číslic jsou alespoň dvě jedničky:

$$\begin{aligned} c_{-1} &= 0 \\ c_i &= (x_i \,\&\, y_i) \vee ((x_i \vee y_i) \,\&\, c_{i-1}). \end{aligned}$$



Sčítání ze základní školy

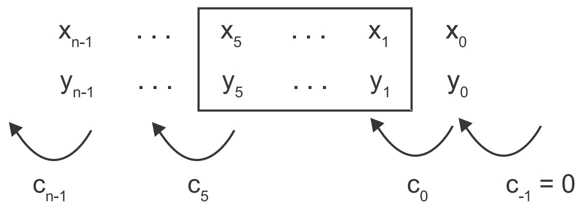
Bohužel na to, abychom spočítali c_i (a tedy z_i), musíme znát hodnotu c_{i-1}, tedy mít spočítané hodnoty pro všechny čísla menší než i. To dává lineární časovou složitost. Zamysleme se nad tím, jak by se proces sčítání mohl zrychlit.

Přenosy v blocích

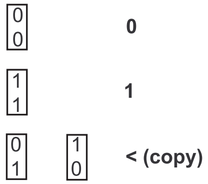
Jediné, co nás při sčítání brzdí, jsou přenosy. Kdybychom je dokázali spočítat rychle (řekněme v logaritmické hloubce), součet už zvládneme dopočítat v konstantním čase.

Podívejme se na libovolný *blok* v našem součtu. Tak budeme říkat číslům $x_a \ldots x_b$ a $y_a \ldots y_b$ v nějakém intervalu indexů $\langle a, b \rangle$. Přenos c_b vystupující z tohoto bloku závisí mimo hodnot sčítanců už pouze na přenosu c_{a-1}, který do bloku vstupuje. Záviset může pouze třemi možnými způsoby:

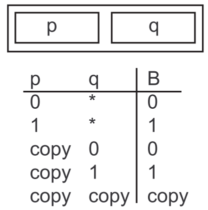
1. generuje přenos: $c_a = 1$,
2. pohlcuje přenos: $c_a = 0$,
3. kopíruje přenos: $c_a = c_{b-1}$.



Blok součtu



Tabulka triviálních bitů



Skládání chování bloků

Cvičení: Rozmyslete si, jak přesně vypadají bloky s jednotlivými typy chování.

Jednobitové bloky se chovají velice jednoduše:

Pokud máme nějaký větší blok B složený z menších bloků p a q, jejichž chování už známe, můžeme z toho odvodit, jak se chová velký blok:

Všimněme si, že skládání chování bloků je asociativní operace (je to vlastně úplně obyčejné skládání funkcí), takže pro libovolný blok můžeme jeho chování spočítat v čase $\mathcal{O}(\log n)$ postupným skládáním („stromečkovým“ způsobem).

To nám dá nějaký kombinační obvod nad trojprvkovou abecedou, ale samozřejmě můžeme chování bloků kódovat i binárně dvojicí bitů:

- $(1, *) = <$,
- $(0, 0) = 0$,
- $(0, 1) = 1$

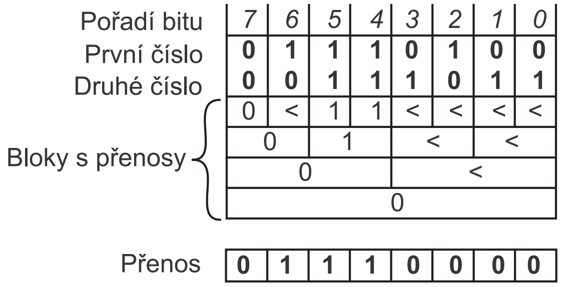
Operaci skládání $(a, x) \odot (b, y) = (c, z)$ pak definujeme takto:

$$c = a \mathbin{\&} b,$$
$$z = (\neg a \mathbin{\&} x) \vee (a \mathbin{\&} y).$$

Paralelní sčítání

Paralelní algoritmus na sčítání už zkonstruujeme poměrně snadno. Bez újmy na obecnosti budeme předpokládat, že počet bitů vstupních čísel n je mocnina dvojky, jinak si vstup doplníme nulami.

1. Spočteme chování bloků velikosti 1. ($\mathcal{O}(1)$ hladin)
2. Postupně počítáme chování bloků velikosti 2^k na pozicích dělitelných 2^k. ($\mathcal{O}(\log n)$ hladin, na nichž se skládají bloky)
3. $c_{-1} \leftarrow 0$
4. Určíme c_n podle c_{-1} a chování (jediného) bloku velikosti n.
5. Postupně počítáme přenosy na hranicích dělitelných 2^k „zahušťováním“: jakmile víme c_{2^k-1}, můžeme dopočítat $c_{2^k+2^{k-1}-1}$ podle chování bloku $\langle 2^k + 2^{k-1} - 1, 2^k \rangle$. ($\mathcal{O}(\log n)$ hladin, na nichž se dosazuje)
6. $\forall i : z_i = x_i \oplus y_i \oplus c_{i-1}$.



Výpočet přenosu

Algoritmus pracuje v čase $\mathcal{O}(\log n)$. Hradel je dokonce lineárně: na jednotlivých hladinách kroku 2 počet hradel exponenciálně klesá od n k 1, na hladinách kroku 5 exponenciálně stoupá od 1 k n, takže se sečte na $\mathcal{O}(n)$.

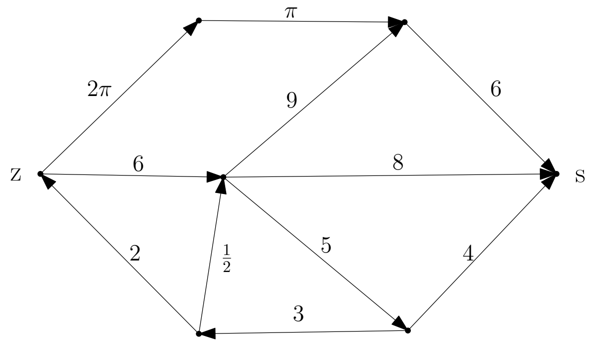
2. Toky v sítích *(přednášel T. Valla, zapsali J. Machálek a K. Vandas)*

Motivační úlohy:

- Mějme orientovaný graf se speciálními vrcholy Želivka a Kanál představující pražské vodovody. V tomto grafu budou vrcholy vodovodními stanicemi a hrany trubkami mezi nimi. Kolik vody proteče ze Želivky do Kanálu?

- Mějme orientovaný graf představující železniční síť; graf má význačné vrcholy Moskva a Fronta, každá hrana grafu má kapacitu, kterou může uvézt. Kolik vojáků je schopna síť převézt z Moskvy a spotřebovat na Frontě?

Definice: *Síť* je uspořádaná čtveřice (G, z, s, c), kde G je orientovaný graf, z a s jsou nějaké dva jeho vrcholy (*zdroj* a *stok*) a c je *kapacita hran,* kterou představuje funkce $c : E(G) \to \mathbb{R}_0^+$.



Příklad sítě. Čísla představují kapacity jednotlivých hran.

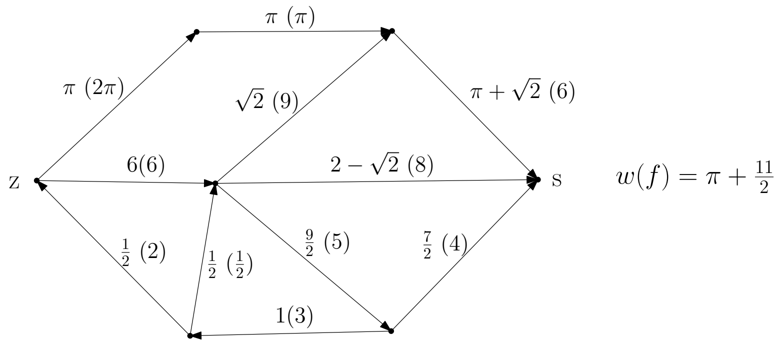
Intuice: Toky v sítích představují rozvržení, jakým suroviny sítí potečou.
Definice: *Tok* je funkce $f : E(G) \to \mathbb{R}_0^+$ taková, že platí:

1. Tok po každé hraně je omezen její kapacitou: $0 \le f(e) \le c(e)$.
2. Kirchhoffův zákon – „síť těsní“:

$$\sum_{xu \in E} f(xu) = \sum_{ux \in E} f(ux) \quad \text{pro každé } u \in V(G) \setminus \{z, s\}.$$

Poznámka: Pokud bychom se chtěli v definici toku u bodu 2 vyhnout podmínkám pro z a s, můžeme zdroj a stok vzájemně propojit (pak jde o tzv. cirkulaci).

Poznámka: V angličtině se obvykle zdroj značí „s“ a stok „t“ (jako source a target).



Příklad toku. Čísla představují toky po hranách, v závorkách jsou kapacity.

Definice: *Velikost toku* f je:

$$|f| := \sum_{zx \in E} f(zx) - \sum_{xz \in E} f(xz).$$

Budeme tedy chtít najít v zadané síti tok, jehož velikost je maximální. Musí vždycky existovat?

Věta: Pro každou síť existuje maximální tok.

Idea důkazu: Dokáže se pomocí metod matematické analýzy s tím, že množina toků je kompaktní a funkce velikosti toku je spojitá (dokonce lineární).

Dobrá, maximální tok vždy existuje. Ale když nám ho někdo ukáže, umíme poznat, že je skutečně maximální? K tomu se budou hodit řezy:

Intuice: Řez v grafu je množina hran oddělující zdroj a stok.

Definice: *Řez* R v síti (G, z, s, c) je množina hran R taková, že neexistuje orientovaná cesta ze z do s v grafu $(V(G), E(G) \setminus R)$.

Definice: *Kapacita řezu* $c(R) = \sum_{uv \in R} c(uv)$.

Někdy je lepší se na řezy dívat takto:

Definice: Pro rozklad množiny vrcholů na dvě disjunktní množiny A, B, kde $z \in A$ a $s \in B$ zavedeme *separátor* $S(A, B)$, což bude množina hran vedoucích z A do B. Pokud je g funkce definovaná na hranách (třeba tok nebo kapacita), definujeme $g(A, B) := \sum_{uv \in E, u \in A, v \in B} g(uv)$.

Pozorování: Každý separátor je řezem (libovolná orientovaná cesta ze z do s musí někdy opustit množinu A, a to jde pouze po hraně patřící do separátoru). Opačně to sice platit nemusí, ale platí, že ke každému řezu existuje separátor takový, že součet kapacit jeho hran je nejvýše kapacita daného řezu. Stačí si zvolit množinu A jako vrcholy dosažitelné ze zdroje po hranách neležících v řezu a do B dát všechny ostatní vrcholy. Separátor $S(A, B)$ pak bude tvořen výhradně hranami řezu (ne nutně všemi) a sám bude řezem.

Definice: Pro libovolnou množinu vrcholů A zavedeme její *doplněk* $\overline{A} := V(G) \setminus A$.

Nyní si všimneme, že velikost toku můžeme měřit přes libovolný separátor: je to množství tekutiny, které teče přes separátor z A do B minus to, které se vrací zpátky (zatím jsme velikost měřili u zdroje, vlastně na triviálním separátoru $A = \{z\}$, $B = \overline{A}$).

Lemma: Nechť $A \subseteq V(G), z \in A, s \notin A$ a f je libovolný tok. Potom platí, že:

$$|f| = f(A, \overline{A}) - f(\overline{A}, A).$$

Důkaz: provedeme pomocí Kirchhoffova zákona a definice velikosti toku:

$$\text{Pro každý vrchol } u \neq z, s \text{ platí:} \quad \sum_{ux \in E} f(ux) - \sum_{xu \in E} f(xu) = 0,$$

$$\text{pro zdroj pak:} \quad \sum_{zx \in E} f(zx) - \sum_{xz \in E} f(xz) = |f|.$$

Rovnice sečteme:

$$\sum_{u \in A} \left(\sum_{ux \in E} f(ux) - \sum_{xu \in E} f(xu) \right) = |f|.$$

Všimneme si, že hrany, jejíž oba koncové vrcholy leží v množině A, přispívají k této sumě jednou kladně a jednou záporně, hrany, jejíž ani jeden vrchol neleží v A, nepřispívají vůbec, a konečně hrany vedoucí z A do B, resp. opačně přispějí jen jednou (kladně, resp. záporně). Tedy:

$$f(A, \overline{A}) - f(\overline{A}, A) = \sum_{u \in A, v \notin A} f(uv) - \sum_{u \notin A, v \in A} f(uv) = |f|.$$

♡

Důsledek: Pokud f je tok, R je řez, pak platí: $|f| \leq c(R)$.

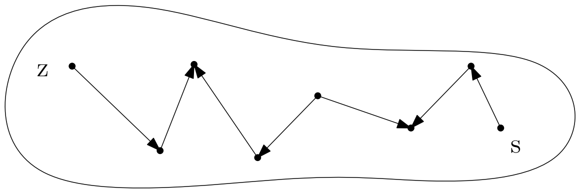
Důkaz: Dokážeme pro separátory (už víme, že ke každému řezu najdeme stejně velký nebo menší separátor):

$$|f| = f(A, \overline{A}) - f(\overline{A}, A) \leq f(A, \overline{A}) \leq c(A, \overline{A}) \leq c(R).$$

♡

Nyní víme, že velikost každého toku je shora omezena velikostí každého řezu. Kdybychom tedy k našemu toku uměli najít stejně velký řez, hned víme, že tok je maximální a řez minimální. Překvapivě platí, že se to povede vždy. K tomu se nám budou hodit zlepšující cesty:

Definice: *Zlepšující cesta* budeme říkat takové cestě mezi danými dvěma vrcholy, která používá buďto hrany sítě (hrany orientované po směru cesty) nebo hrany k nim opačné (v síti jsou tedy proti směru cesty).



Příklad zlepšující cesty.

Definice: *Zlepšující cesta* P ze z do s je *nasycená*, pokud:

$$\exists e \in P : \begin{cases} f(e) = c(e) & \text{je-li } e \text{ orientovaná po směru} \\ f(e) = 0 & \text{je-li } e \text{ orientovaná proti směru} \end{cases}$$

Jinak je zlepšující cesta nenasycená.

Definice: *Tok je nasycený,* pokud je každá zlepšující cesta P ze z do s nasycená.

Věta: Tok f je nasycený $\Leftrightarrow$ f je maximální. Navíc pro každý maximální tok f existuje řez R takový, že $|f| = c(R)$.

Důkaz:

„$\Leftarrow$“ dokážeme nepřímo – ukážeme, že pokud nějaký tok není nasycený, tak ho ještě lze zlepšit, pročež nemůže být maximální. Mějme nějaký nenasycený tok f. Existuje tedy nenasycená zlepšující cesta P. Podél této cesty budeme tok vylepšovat. Zvolíme:

$$\begin{aligned} \varepsilon_1 &:= \min_{e \in P \text{ po směru}} \{c(e) - f(e)\}, \\ \varepsilon_2 &:= \min_{e \in P \text{ proti směru}} \{f(e)\}, \\ \varepsilon &:= \min \{\varepsilon_1, \varepsilon_2\}. \end{aligned}$$

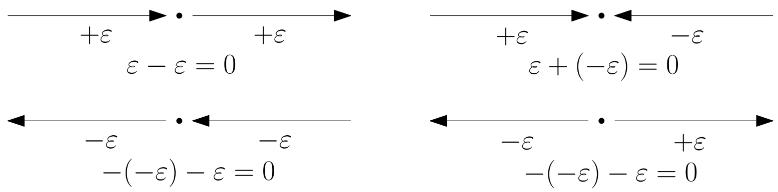
Jelikož cesta byla nenasycená, musí být $\varepsilon > 0$. Nyní z toku f vytvoříme tok f' takto:

$$f'(e) := \begin{cases} f(e) + \varepsilon & \text{je-li } e \text{ po směru cesty,} \\ f(e) - \varepsilon & \text{je-li } e \text{ proti směru,} \\ f(e) & \text{pokud } e \notin P. \end{cases}$$

Nyní je potřeba ověřit, že $f^{'}$ je skutečně tok:

$$0 \leq f^{'}(e) \leq c(e) \ldots \text{platí stále díky volbě } \varepsilon.$$

Platnost Kirchhoffova zákona ověříme rozborem případů:

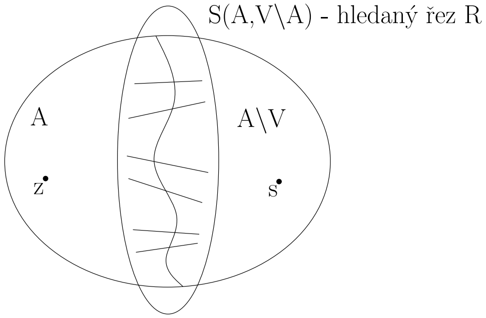


Rozbor případů.

Funkce f' je tedy tok. Jeho velikost se ovšem oproti f zvýšila o ε, takže f nebyl maximální.

„$\Rightarrow$“: Uvážíme množinu vrcholů $A \subseteq V(G)$ definovanou tak, že $v \in A$ právě tehdy, když existuje nenasycená cesta ze z do v. Všimněme si, že $z \in A$ a $s \notin A$. Potom $S(A, \overline{A})$ je řez, který je stejně velký jako tok f. Jak už víme, pokud k toku najdeme stejně velký řez, je tok maximální. ♡

To, co jsme o tocích zjistili, můžeme shrnout do následující „minimaxové“ věty:



Rozdělení $V(G)$ na množinu A a $\overline{A}$ v důkazu hlavní věty o tocích.

Věta (Hlavní věta o tocích, Ford-Fulkerson): Pro každou síť platí:

$$\max_{f \text{ tok}} |f| = \min_{R \text{ řez}} c(R).$$

Důkaz: Nerovnost „$\leq$" je náš Důsledek lemmatu o tocích a řezech, rovnost platí proto, že k maximálnímu toku existuje podle předcházející věty stejně velký řez. ♡

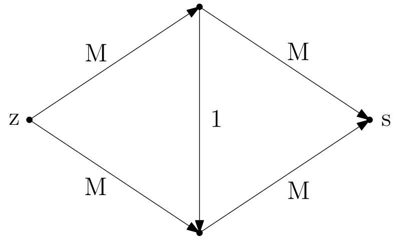
Zlepšování toků pomocí zlepšujících cest není jen pěkný důkazový prostředek, dá se pomocí něj také formulovat elegantní algoritmus na hledání maximálního toku:

Algoritmus (hledání maximálního toku v síti, Ford-Fulkerson)

1. $f \leftarrow$ libovolný tok, třeba všude nulový ($\forall e \in E : f(e) \leftarrow 0$).
2. Dokud $\exists$ zlepšující cesta P: vylepšíme f podél P jako v důkazu věty.
3. Prohlásíme f za maximální tok.

Cvičení:

- Je pro přirozené kapacity F-F algoritmus konečný? Ano – v každém zlepšujícím kroku algoritmu se celkový tok zvětší aspoň o jedna. Protože máme horní odhad na velikost maximálního toku (např. součet kapacit všech hran), máme i horní odhad na dobu běhu algoritmu.
- Je F-F algoritmus konečný pro racionální kapacity hran? Ano – všechny kapacity vynásobíme společným jmenovatelem a převedeme na předchozí případ. Algoritmus se přitom chová stejně jako s původními kapacitami.
- A pro reálné kapacity? Obecně ne – zkuste najít síť s některými kapacitami iracionální, kde se algoritmus při nešikovné volbě zlepšujících cest nikdy nezastaví a dokonce ani nekonverguje k maximálnímu toku.
- Kolik kroků bude muset algoritmus na následující síti maximálně udělat, aby úspěšně doběhl? ($2M$ kroků)
- Pokud bychom vždy hledali *nejkratší* zlepšující cestu, což je snad nejpřímočařejší možná implementace (prohledávání do šířky), algoritmus by se zastavil po $\mathcal{O}(M^2N)$ krocích (tomu se říká Edmondsův-Karpův algoritmus). To si nebudeme dokazovat a místo toho si na příští přednášce rovnou odvodíme efektivnější Dinicův algoritmus.



Příklad sítě. Kolik kroků musí maximálně udělat F-F algoritmus?

3. Dinicův algoritmus *(zapsali Jakub Melka, Petr Musil)*[1]

Na minulé přednášce jsme si ukázali *Fordův-Fulkersonův* algoritmus. Víme o něm, že když se zastaví, tak vydá maximální tok. Jenže zastavit se nemusí (například pro sítě s reálnými kapacitami), nebo trvá příliš dlouho. Ukážeme si lepší algoritmus, *Dinicův*, který má výrazně menší složitost a zastaví se vždy.

Idea je následující: v algoritmu budeme používat *síť rezerv*, která bude obsahovat rezervy – kolik ještě po dané hraně můžeme pustit, aby to nepřekročilo její kapacitu. Síť rezerv pak budeme využívat k vylepšování toku.

Definice: *Síť rezerv* R k síti $S = (V, E, z, s, c)$ a toku f v S je síť $R = (V, E \cup \overleftarrow{E}, z, s, r)$, pro $\forall e \in E$:

- $r(e) = c(e) - f(e)$,
- $r(\overleftarrow{e}) = f(e)$,

kde hrana $\overleftarrow{e}$ vznikne z hrany e tak, že se zorientuje opačným směrem. V případě, že v síti rezerv už opačně orientovaná hrana[2] je, pak vznikne multigraf[3] s multiplicitou maximálně dva.

Síť rezerv budeme používat v algoritmu k hledání vylepšujících toků. K tomu nám bude sloužit následující věta:

Věta: Je-li f tok v síti S a g tok v příslušné síti rezerv, pak $\exists$ tok f' v S takový, že $|f'| = |f| + |g|$, což znamená $\forall e \in E : f'(e) = f(e) + g(e) - g(\overleftarrow{e})$.

Důkaz: Rozebereme si jednotlivé případy pro $\forall e \in E$:

a) $g(e) = g(\overleftarrow{e}) = 0 \Rightarrow f'(e) = f(e)$.
b) $g(e) > 0$ a zároveň $g(\overleftarrow{e}) = 0 \Rightarrow f'(e) = f(e) + g(e)$.
c) $g(e) = 0$ a zároveň $g(\overleftarrow{e}) > 0 \Rightarrow f'(e) = f(e) - g(\overleftarrow{e})$.
d) Nastává cirkulace, tu snadno odstraníme: odečteme ε od obou hran e a $\overleftarrow{e}$, kde $\varepsilon = \min(g(e), g(\overleftarrow{e}))$. Převedeme tím tento případ na jeden ze tří uvedených výše.

[1] s díky Bernardovi Lidickému za obrázky

[2] vznikne tak, že v původní síti jsou dvě opačně orientované hrany mezi stejnými vrcholy.

[3] graf, který může mít mezi dvěma vrcholy více stejně orientovaných hran.

Je však f' tok? Víme, že f' určitě nemůže klesnout pod nulu, protože se odečítá jen v případě c), a tam je z definice vidět, že f' pod nulu klesnout nemůže. Kapacita také nemůže být překročena, přičítá se jen v případě b) a z definice se nepokazí, protože $g(e) = c(e) - f(e)$, tedy v nejhorším případě $f'(e) = c(e)$.

Dále dokážeme, že f' dodržuje Kirchhoffův zákon. V následujících sumách předpokládejme, že všechny vrcholy jsou rozdílné od zdroje a stoku. Musí platit, že:

$$\sum_{ab \in E} f'(ab) = \sum_{ba \in E} f'(ba).$$

Rozepíšeme si tuto rovnici dle definice:

$$\sum_{ab \in E} f'(ab) - \sum_{ba \in E} f'(ba) = 0,$$

$$\sum_{uv \in E} (f(uv) + g(uv) - g(\overleftarrow{uv})) - \sum_{vu \in E} (f(vu) + g(vu) - g(\overleftarrow{vu})) = 0.$$

Roztrhneme si to na čtyři sumy a dostaneme:

$$\underbrace{\sum_{uv \in E} f(uv) - \sum_{vu \in E} f(vu)}_{0} +$$

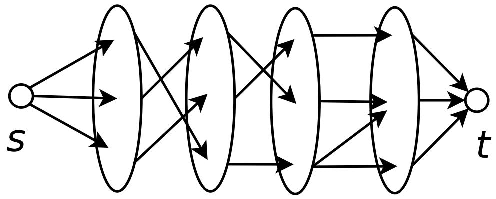
$$+ \underbrace{\sum_{uv \in E} g(uv) - g(\overleftarrow{uv}) - \sum_{vu \in E} g(vu) - g(\overleftarrow{vu})}_{0} = 0,$$

neboť f i g jsou toky a musí splňovat Kirchhoffův zákon. ♡

Tato věta nám říká, že pokud existuje nenulový tok v síti rezerv, pak lze tok v původní síti ještě zvětšit. Naopak pokud takový tok neexistuje, je tok v původní síti maximální.

Definice: f je *blokující tok*, pokud na každé orientované cestě P ze zdroje do spotřebiče $\exists e \in P : f(e) = c(e)$.

Definice: C je *pročištěná síť*, pokud obsahuje pouze vrcholy a hrany na nejkratších $z \to s$ cestách. Pročištěná síť nemá slepé uličky, ani hrany vedoucí ze stoku někam do dalšího vrcholu.



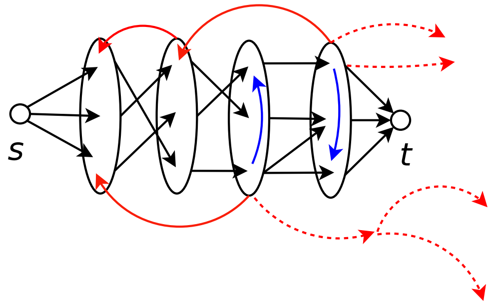
Příklad pročištěné sítě

Algoritmus (hledání maximálního toku v síti, Dinicův)

1. $f \leftarrow$ nulový tok.
2. Sestrojíme síť rezerv R, vynecháme hrany s nulovou rezervou.
3. $l \leftarrow$ délka nejkratší cesty $z \to s$ v R.
4. Když $l = \infty$, tak skončíme.
5. Sestrojíme pročištěnou síť C, a to následujícím způsobem:
6. Spustíme BFS[4] algoritmus ze zdroje.
7. BFS nám rozdělí uzly do vrstev, vyhodíme hrany za spotřebičem a slepé uličky.
8. $g \leftarrow$ blokující tok v C.
9. Zlepšíme tok f podle g a jdeme na bod 3.

Postup tvorby pročištěné sítě podrobněji: Prohledáním do šířky vytvoříme vrstvy C_i, zahodíme ty za spotřebičem, ponecháme pouze hrany mezi C_i a C_{i+1}. Ještě musíme odstranit slepé uličky – cesty, které končí v $C_m : m < l$, protože ty určitě nejsou součástí nejkratší $z \to s$ cesty.

Pročištění zvládneme v lineárním čase $\mathcal{O}(N + M)$, v případě souvislého grafu pouze $\mathcal{O}(M)$.



Příklad nepročištěné sítě

Na obrázku nepročištěné sítě vidíme, co se má smazat. Černé hrany ponecháme, ty tam jsou správně. Červené hrany jsou zpětné, ty smažeme. Modré hrany jsou hrany ve stejné vrstvě, ty rovněž smažeme. Červené tečkované hrany nevedou vůbec do stoku, takže ty rovněž smažeme.

Definice: *Fází* algoritmu označíme jeden běh cyklu – kroky 3 až 9.

Provedeme podrobnou analýzu algoritmu z hlediska složitosti a uvidíme, že má složitost $\mathcal{O}(N^2 M)$. Nejprve analyzujeme hledání blokujícího toku, pak se podíváme, kolik fází maximálně může Dinicův algoritmus mít.

Algoritmus hledání blokujícího toku

[4] Breadth-First Search, standardní prohledávání do šířky.

1. $g \leftarrow$ nulový tok.
2. Dokud $\exists z \to s$ cesta P v pročištěné síti C:
3. $\quad \varepsilon \leftarrow \min_{e \in P}(c(e) - f(e))$.
4. $\quad \forall e \in P : g(e) \leftarrow g(e) + \varepsilon$, pokud $g(e)$ vzroste na $r(e)$, tak smažeme hranu e.
5. $\quad$ Dočistíme síť tím, že odstraníme slepé uličky, které mohly vzniknout smazáním hrany e.

Při každém průchodu se smaže vždy alespoň jedna hrana, tedy maximálně M-krát provádíme $\mathcal{O}(N)$ – právě tolik trvá nalezení cesty P, protože délka cesty bude kratší nebo rovna N. Čištění pak maximálně smaže celý graf, jedno mazání nás stojí konstantní čas, tedy celková složitost tohoto algoritmu bude $\mathcal{O}(MN)$.

Dokážeme si, že počet fází je menší nebo roven N. Algoritmus se ukončí, pokud $l > N$, protože pak už neexistuje nejkratší $z \to s$ cesta, prošli jsme všechny vrcholy.

Lemma: Při každé fázi vzroste l alespoň o jedna.

Důkaz: Uvažme síť R, rozdělenou na vrstvy, ještě před pročištěním. Po pročištění některé hrany zmizí. Přibýt[5] mohou jen zrcadlové protějšky již existujících hran.

Uvažme cestu P délky l nebo menší ze $z \to s$ a novou hranu e vzniklou při poslání toku po hraně s nulovým tokem:

a) Hrana $e \notin P \Rightarrow$ zablokování, taková cesta neexistuje.
b) Hrana $e \in P \Rightarrow$ délka $> l$, protože hrana e vede z nějakého vrcholu ve vrstvě C_i do vrcholu ve vrstvě C_{i-1}. ♡

Věta: Dinicův algoritmus najde maximální tok v čase $\mathcal{O}(MN^2)$.

Důkaz: Složitost plyne přímo z předchozího lemmatu a složitosti algoritmu hledání blokujícího toku.

Korektnost (najde vždy maximální tok) dokážeme takto: Nechť algoritmus proběhne. Skončit může jedině tehdy, když nenajde žádnou cestu ze zdroje do spotřebiče v síti rezerv, kterou by mohl být tok vylepšen. Tedy tok musí být nutně maximální. ♡

Takto napsaný algoritmus je ještě příliš pomalý, ale jde výrazně zrychlit. Například namísto přepočítávání toků na rezervy a naopak můžeme minimálně vnitřní cyklus počítat jenom v rezervách. Pročištění se dá dělat jednou namísto dvakrát, můžeme prohledávat do hloubky a mazat při vracení se z vrcholů. Dokonce i hledání blokujícího toku lze řešit v rámci prohledávání do hloubky. Cesta si pamatuje minimum rezerv a při vracení se rezerva snižuje.

Problematika toků v sítích má velké uplatnění v kombinatorice a teorii grafů. Zde uvedeme jeden příklad:

Hledání maximálního párování v bipartitních grafech: Zorientujeme všechny hrany zleva doprava a přidáme zdroj, z něhož vedou hrany do první partity, a stok, do něhož vedou hrany z druhé partity. Hrany mezi partitami mají jednotkovou kapacitu.

[5] Přibudou tak, že po hraně s nulovým tokem pošleme nějaký tok, v opačném směru v síti rezerv vytvoříme z nulové hrany nenulovou.

4. Goldbergův algoritmus *(zapsali R. Tupec, J. Volec, J. Záloha)*

Představíme si nový algoritmus pro hledání maximálního toku v síti, který se ukáže stejně dobrý jako *Dinicův algoritmus* ($\mathcal{O}(MN^2)$) a po několika vylepšeních bude i lepší.

Tento algoritmus narozdíl od Dinicova algoritmu začíná s přebytky v sousedních vrcholech zdroje a snaží se jich zbavit pomocí převádění. Pokud bychom toto převádění dělali „tupým způsobem“, mohl by se algoritmus zacyklit. Proto pro každý vrchol budeme definovat výšku, a jak uvidíme, s její pomocí se vyhneme zacyklení.

Definice: Funkce $f : E \to \mathbb{R}_0^+$ je *vlna* v síti (V, E, z, s, c) tehdy, když $\forall uv \in E : f(uv) \leq c(uv)$, kde $c(uv)$ je kapacita hrany uv, a $\forall v \neq z, s : f^\Delta(v) \geq 0$. Funkcí $f^\Delta(v)$ pro libovolný vrchol v rozumíme *přebytek* v tomto vrcholu, což je součet všeho, co do vrcholu v přiteče, minus součet všeho, co z v odteče. To můžeme zapsat jako:

$$f^\Delta(v) := \sum_{uv \in E} f(uv) - \sum_{vu \in E} f(vu).$$

Každý tok je vlna, kde $\forall v \neq z, s : f^\Delta(v) = 0$.

Algoritmus používá sít rezerv, kterou jsme nadefinovali již v předchozí kapitole věnované Dinicovi.

Dále budeme provádět následující dvě operace na vrcholech sítě. K tomu budeme potřebovat přiřadit všem vrcholům výšku pomocí funkce $h : V \to \mathbb{N}$.

Operace: Pro hranu $uv \in E$ definujme *převedení přebytku*:

Pokud platí, že:

1. ve vrcholu u je nenulový přebytek, tj. $f^\Delta(u) > 0$,
2. vrchol u je výš než vrchol v, tj. $h(u) > h(v)$, a
3. hrana uv má nenulovou rezervu, tj. $r(uv) > 0$,

převedeme tok o velikosti $\delta := \min(f^\Delta(u), r(uv))$ z u do v tímto způsobem:

1. $f^\Delta(u) \leftarrow f^\Delta(u) - \delta$ a $f^\Delta(v) \leftarrow f^\Delta(v) + \delta$,
2. $r(uv) \leftarrow r(uv) - \delta$ a $r(vu) \leftarrow r(vu) + \delta$.

Definice: Řekneme, že převedení je *nasycené*, pokud je po převodu rezerva na hraně uv nulová, tedy $r(uv) = 0$. Naopak převedení je *nenasycené*, pokud po převodu $f^\Delta(u) = 0$. Pokud $r(uv) = 0$ a $f^\Delta(u) = 0$, budeme převedení považovat za *nasycené*.

Operace: Pro vrchol $u \in V$ definujme *zvednutí vrcholu*: Pokud během výpočtu narazíme ve vrcholu u na přebytek, který nelze nikam převést, zvětšíme výšku vrcholu u o jedničku, tj. $h(u) \leftarrow h(u) + 1$.

Algoritmus (hledání maximálního toku v síti, Goldberg)

1. $\forall v \in V : h(v) \leftarrow 0$ (všem vrcholům nastavíme počáteční výšku nula).
2. $h(z) \leftarrow N$ (zdroj zvedneme do výšky N).
3. $\forall e \in E : f(e) \leftarrow 0$ (po hranách na počátku nenecháme protékat nic).

4. $\forall zu \in E : f(zu) \leftarrow c(zu)$ (ze zdroje pustíme maximální možnou vlnu).
5. Dokud $\exists u \in V \setminus \{z, s\}, f^{\Delta}(u) > 0$:
6. Pokud $\exists uv \in E, r(uv) > 0$ a $h(u) > h(v)$: převedeme přebytek po hraně uv.
7. V opačném případě zvedneme u.
8. Vrátíme tok f jako výsledek.

Nyní bude následovat několik lemmat a invariantů, jimiž dokážeme správnost a časovou složitost výše popsaného algoritmu.

Invariant A: Funkce $f : E \to \mathbb{R}$ je v každém kroku algoritmu vlna, $h(v)$ nikdy neklesá, $h(z) = N$ a $h(s) = 0$.

Důkaz: Pro první část invariantu si stačí rozmyslet, že v žádném kroku algoritmu nepřekročíme kapacity hran a nevytvoříme záporný přebytek. Pro $v \in V \setminus \{z, s\}$ skutečně výšku pouze zvyšujeme a z podmínky v pátém kroku algoritmu vyplývá, že nás přebytky v z a s v podstatě nezajímají, tudíž se $h(z)$ a $h(s)$ nemění. ♡

Invariant S (o Spádu): Neexistuje hrana se spádem větším než jedna a nenulovou rezervou, neboli $\forall uv \in E, r(uv) > 0 : h(u) \leq h(v) + 1$.

Důkaz: Podívejme se, kdy by mohla vzniknout nenasycená hrana se spádem větším než jedna. Během inicializace k tomu evidentně nedojde, protože všechny hrany jsou nenasycené nebo mají kapacitu nula, proto je můžeme vypustit. Během práce algoritmu k tomu však také nedojde, jak uvidíme z rozebrání následujících případů. Pokud již existuje vrchol v s kladným přebytkem, dále existuje nenasycená hrana vu a $h(v) = h(u) + 1$, vrchol v algoritmus nezvedne, ale přebytek pošle po této hraně. Uvažme tedy ještě druhý případ, kdy existuje nasycená hrana uv se spádem větším než jedna a tuto hranu se pokusíme odsytit. Jenže pokud bychom chtěli něco poslat v protisměru, snažili bychom se o přelití proti směru funkce h. ♡

Lemma K (o Korektnosti): Když se algoritmus zastaví, je f maximální tok.

Důkaz: Nejprve ukážeme, že f je tok, a pak jeho maximalitu. Vyjděme z toho, že f je vlna a algoritmus se může zastavit, jen pokud nastanou oba následující případy současně:

- Ve vrcholech grafu nejsou žádné přebytky (mimo z a s), protože jinak by se algoritmus nezastavil a pokračoval dále ve výpočtu. Tudíž f je tok.
- Neexistuje nenasycená cesta ze zdroje do stoku, čímž z *Ford-Fulkersonovy věty* okamžitě vyplývá, že f je tok maximální. A jak tuto neexistenci nahlédneme? Pro spor předpokládejme, že nějaká nenasycená cesta P ze z do s existuje. Tato cesta může mít maximálně $N - 1$ hran. O nich víme, že všechny mají kladnou rezervu, a dále víme, že po celou dobu výpočtu je výška zdroje N a výška stoku 0. Takže celkový spád cesty P je N, což ale znamená, že na cestě P existuje hrana s kladnou rezervou, která má spád alespoň 2. To je však v rozporu s invariantem S. ♡

Invariant C (Cesta domů, do zdroje): Je-li $v \in V \setminus \{z, s\}$ a $f^{\Delta}(v) > 0$, pak existuje nenasycená cesta z v do z.

Důkaz: Mějme nějaký vrchol $v \in V$ takový, že $f^{\Delta}(v) > 0$. Potom definujme množinu $A := \{u \in V : \exists$ nenasycená cesta z v do $u\}$. Mějme vrcholy $a \in A$ a $b \in V \setminus A$ takové, že $ba \in E$. O nich víme, že $f(ba) = 0$, protože pokud by tomu tak nebylo, muselo by platit $r(ab) > 0$, a tudíž by b patřilo do množiny A.

Sečtěme přebytky ve všech vrcholech množiny A. Protože přebytek každého vrcholu se spočítá jako součet toků do něj vstupujících minus součet toků z něj vystupujících a všechny hrany, jejichž oba vrcholy leží v A, se jednou přičtou a jednou odečtou, platí:

$$\sum_{u \in A} f^{\Delta}(u) = \sum_{\substack{ab \in E \cap \mathbb{A} \\ \mathbb{A} = \bar{A} \times A}} f(a, b) - \sum_{\substack{ba \in E \cap \mathbb{A} \\ \mathbb{A} = A \times \bar{A}}} f(b, a).$$

My však víme, že do A nic neteče, a proto $\sum_{v \in A} f^{\Delta}(v) \leq 0$. Zároveň však v A je vrchol s kladným přebytkem, totiž v, proto v A musí být také vrchol se záporným přebytkem a jediný takový je z. ♡

Invariant V (Výška): $\forall v \in V$ platí $h(v) \leq 2N$.

Důkaz: Víme, že počet hran na cestě ze z do $\forall v \in V$ je maximálně $N - 1$. Pokud by existoval vrchol v s výškou $h(v) > 2N$, museli jsme tento vrchol zvednout alespoň $(2N + 1)$-krát. Snadno si uvědomíme, že z nikdy nezvedáme, a tudíž by na cestě ze z do v musela být hrana se spádem větším než jedna, což je spor s invariantem S. ♡

Lemma Z (počet Zvednutí): Počet všech zvednutí je maximálně $2N^2$.

Důkaz: Stačí si uvědomit, že každý vrchol můžeme zvednout maximálně $2N$-krát a vrcholů je N. ♡

Lemma S (naSycená převedení): Počet všech nasycených převedení je nejvýš NM.

Důkaz: Mějme hranu $uv \in E$, kterou jsme právě nasytili. Tedy platí $h(v) < h(u)$ a zároveň $r(uv) = 0$. Aby se rezerva této hrany změnila, musel by ji někdo odsytit. Pro odsycení hrany se musí otočit nerovnost mezi výškami koncových vrcholů. Tedy $h(v) > h(u)$. Proto, abychom tuto hranu opět nasytili, musíme opět změnit nerovnost výšek na $h(v) < h(u)$. Mezi dvěma nasyceními hranami uv proto proběhla minimálně dvě zvednutí vrcholu u. Algoritmus nikdy výšku vrcholu nesnižuje, a tedy počet všech nasycených převedení je skutečně NM. ♡

Lemma N (Nenasycená převedení): Počet všech nenasycených převedení je $\mathcal{O}(N^2M)$.

Důkaz: Důkaz provedeme pomocí potenciálu – nadefinujme si následující funkci jako potenciál:

$$\psi := \sum_{\substack{v: f^{\Delta}(v) > 0 \\ v \neq z, s}} h(v).$$

Nyní se podívejme, jak se náš potenciál během algoritmu vyvíjí a jaké má vlastnosti:

- Během celého algoritmu je $\psi \geq 0$, neboť je součtem nezáporných členů.
- Na počátku je $\psi = 0$.

- Zvednutí vrcholu zvýší ψ o jedničku. Již víme, že za celý průběh algoritmu je všech zvednutí maximálně $2N^2$, proto zvedáním vrcholů zvýšíme potenciál dohromady nejvýše o $2N^2$.
- Nasycené převedení zvýší ψ nejvýše o $2N$, protože buď po převodu hranou uv v u zůstal nějaký přebytek, takže se mohl potenciál zvýšit až o $2N$, nebo je přebytek v u po převodu nulový a potenciál se dokonce o jedna snížil. Za celý průběh tak dojde k maximálně NM takovýmto převedením, díky nimž se potenciál zvýší maximálně o $2N^2M$.
- Konečně když převádíme po hraně uv nenasyceně, tak od potenciálu určitě odečteme výšku vrcholu u a možná přičteme výšku vrcholu v. Jenže $h(v) = h(u)-1$, a proto nenasycené převedení potenciál vždy sníží alespoň o jedna.

Z tohoto rozboru chování potenciálu ψ v průběhu algoritmu získáváme, že počet všech nenasycených převedení může být nejvýše $2N^2 + 2N^2M$, což je $\mathcal{O}(N^2M)$. ♡

Implementace: Budeme si pamatovat seznam P všech vrcholů $v \neq z, s$ takových, že $f^{\triangle}(v) > 0$. Když měníme přebytek nějakého vrcholu, můžeme náš seznam v konstantním čase aktualizovat (např. tak, že si každý vrchol pamatuje pozici, na které v seznamu je). A v konstantním čase také umíme odpovědět, zda existuje nějaký vrchol s přebytkem. Dále si $\forall u \in V$ budeme pamatovat $L(u) :=$ seznam $uv \in E$ takových, že $r(uv) > 0$ a $h(v) < h(u)$. Díky tomu můžeme přistupovat k patřičným sousedům u v čase $\mathcal{O}(1)$, stejně jako provádět operace přidání do $L(u)$, resp. smazání v něm. Každé převedení po hraně uv nás stojí konstantní čas na aktualizaci rezerv hran uv a vu, stejně tak i na aktualizaci přebytků ve vrcholech u a v. V případě, že se jedná o nasycené převedení, musíme ještě odstranit hranu uv z $L(u)$, což také stihneme v čase $\mathcal{O}(1)$. A konečně zvedání vrcholu v nám zabere čas $\mathcal{O}(N)$, protože musíme obejít všechny hrany uv, kterých je $\mathcal{O}(N)$, porovnat výšky a případně odebrat uv z seznamu $L(u)$ resp. přidat do $L(v)$. Abychom pro odebrání hrany uv ze seznamu $L(u)$ nemuseli procházet celý seznam, budeme si $\forall v \in V$ pamatovat ještě $L^{-1}(v) :=$ seznam ukazatelů na hrany uv v seznamech $L(u)$.

Věta: Goldbergův algoritmus najde maximální tok v čase $\mathcal{O}(N^2M)$.

Důkaz: Z lemmatu Z vyplývá, že celkový počet zvednutí je maximálně $2N^2$, přičemž každé zvednutí jsme schopni provést v čase $\mathcal{O}(N)$. Takže dohromady pro zvedání spotřebujeme čas $\mathcal{O}(N^3)$, což je pro souvislé sítě určitě $\mathcal{O}(N^2M)$. Z lemmatu S pro změnu vyplývá, že nasycená převedení nás stojí $\mathcal{O}(NM)$, a na závěr z lemmatu N dostáváme časovou složitost $\mathcal{O}(N^2M)$ pro převedení nenasycená. Proto celková složitost algoritmu je $\mathcal{O}(N^2M)$. ♡

Dokázali jsme, že algoritmus má časovou složitost $\mathcal{O}(N^2M)$ pro libovolnou posloupnost zvedání a převádění. Nabízí se otázka, zda není možné vhodným výběrem těchto operací výpočet zrychlit. Ukážeme, že pokud v 5. kroku algoritmu budeme vždy brát vrchol u takový, že $h(u)$ je maximální, počet nenasycených převedení se sníží.

Lemma N': Počet nenasycených převedení v upravené verzi Goldbergova algoritmu je $\mathcal{O}(N^2\sqrt{M})$, což je maximálně $\mathcal{O}(N^3)$. Díky tomu je i složitost celého algoritmu $\mathcal{O}(N^3)$.

Důkaz: Viz příští přednášku.

5. Třídicí sítě *(zapsaly T. Klimošová a K. Böhmová)*

Goldbergův algoritmus – pokračování

Algoritmus upřesníme tak, že místo libovolného vrcholu s přebytkem budeme vždy pracovat s takovým vrcholem v, jehož výška $h(v)$ je největší. Jak dokážeme v následujícím lemmatu, sníží se tím počet potřebných nenasycených převedení. Takto modifikovaný algoritmus budeme značit G'.

Lemma N': V algoritmu G' je počet nenasycených převedení $\mathcal{O}(N^3)$.

Důkaz: Definujme H jako maximální výšku vrcholů s přebytkem:

$$H := \max\{h(v) : v \neq z, s, f^{\triangle}(v) > 0\}.$$

Běh algoritmu G' rozdělíme na fáze tak, že fáze skončí po každé změně H. Odhadneme počet nenasycených převedení v jedné fázi: bude jich nanejvýš stejně jako vrcholů, které se na začátku fáze nacházely na nejvyšší hladině – z jiných vrcholů v průběhu fáze nic nepřevádíme a nenasycené převedení můžeme provést z každého vrcholu nejvýše jednou. Počet nenasycených převedení za fázi je proto nejvýše N.

Odhadneme počet fází: rozdělíme fáze podle toho, jestli končí snížením nebo zvýšením H a odhadneme jejich počty. Maximálně $2N^2$ fází může končit zvýšením H, protože dle lemmatu Z provedeme nanejvýš $2N^2$ zvednutí. Snížením H může končit nanejvýš $2N^2$ fází, protože H klesne vždy alespoň o jedna, původně bylo nula a nikdy nemůže být záporné – klesnout tedy nemůže víckrát než vzrůst.

Máme maximálně $4N^2$ fází o nejvýše N nenasycených převedení, celkem tedy algoritmus G' provede $\mathcal{O}(N^3)$ nenasycených převedení. ♡

Ve skutečnosti je algoritmus ještě lepší (alespoň pro řídké grafy):

Lemma N'': V algoritmu G' je počet nenasycených převedení $\mathcal{O}(N^2\sqrt{M})$.

Důkaz: (Nezkouší se.) Algoritmus rozdělíme na fáze stejně jako v důkazu předchozího lemmatu a zvolíme přirozené číslo K (jak velké ho zvolíme, se rozhodneme až na konci důkazu). Fáze tentokrát budeme dělit na drahé, v nichž se provede více než K nenasycených převedení, a laciné, v nichž se nenasycených převedení provede maximálně K.

Nyní budeme zkoumat počty převedení v obou typech fází. Jak víme z minulého lemmatu, všech fází je nanejvýš $4N^2$, takže v laciných se provede nanejvýš $4N^2K$ nenasycených převedení. Za účelem zkoumání drahých fází definujeme potenciál:

$$\psi := \sum_{v \neq z,s;\, f^{\triangle}(x)>0} \frac{p(v)}{K},$$

kde $p(v) := |\{u : h(u) \leq h(v)\}|$, čili počet vrcholů ve stejné nebo menší výšce než v. Víme, že na začátku bude $\psi \leq N^2/K$ a po celou dobu bude $\psi \geq 0$.

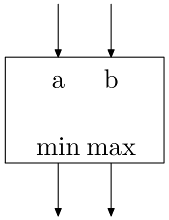
Zvednutím vrcholu v se hodnota $p(v)$ zvýší maximálně o N, u libovolného jiného vrcholu w jeho $p(w)$ klesne nebo se nezmění, potenciál tedy vzroste maximálně o N/K. Při sytém převedení po hraně z u do v (z minula víme, že se vždy provádí po hraně spádu nejvýše jedna) se mohl potenciál zmenšit o $p(u)/K$ a mohl se zvětšit o $p(v)/K$. Zvětší se tedy nanejvýš o N/K Při nenasyceném převedení z potenciálu určitě ubude $p(u)/K$ a možná přibude $p(v)/K$. Celkově se tedy sníží nanejvýš o $p(u)/K - p(v)/K$, což je $1/K$ počtu prvků na nejvyšší hladině. Z minulého lemmatu víme, že počet nenasycených převedení v jedné fázi je menší nebo roven počtu vrcholů na nejvyšší hladině na začátku fáze. Vzhledem k tomu, že zkoumáme drahé fáze, v nichž proběhne více než K nenasycených převedení, vrcholů na nejvyšší hladině musí být na začátku fáze také více než K, potenciál se tedy sníží o více než $K/K = 1$.

Potenciál ψ tedy přijme na začátku nanejvýš N^2/K, při zvedání nevzroste o víc než $(N/K)2N^2$, při sytém převádění nevzroste o víc než $(N/K)NM$, takže za celý průběh algoritmu potenciál vzroste maximálně o $(N/2)(N + 2N^2 + NM) = \mathcal{O}(N^2M/2)$, v drahých fázích tak může proběhnout nanejvýš $\mathcal{O}(N^2M/2)$ nenasycených převedení. Celkem algoritmus vykoná $\mathcal{O}(N^2K + N^2M/K)$ nenasycených převedení. Když za K dosadíme $\sqrt{M}$, dostaneme slíbený počet $\mathcal{O}(N^2\sqrt{M})$. ♡

Implementace: Narozdíl od původní verze algoritmu si ve verzi se zvedáním nejvyššího vrcholu nebudeme pamatovat seznam vrcholů s kladným přebytkem, ale setříděný seznam přihrádek. V každé přihrádce budou jen vrcholy s přebytkem s určitou výškou. Vyhledání nejvyššího vrcholu tedy zvládneme v konstantním čase, stejně pro zvýšení vrcholu nám stačí $\mathcal{O}(1)$ (buď vrchol přesuneme do vedlejší přihrádky, nebo pro něj založíme novou). Převádíme-li přebytek do vrcholu, kde předtím nebyl, pak musí mít výšku o 1 nižší, než vrchol, ze kterého přebytek převádíme (jinak by existovala nenasycená hrana se spádem dva, což nejde). Najít (případně vytvořit) přihrádku nově vzniklému vrcholu s přebytkem tak také stihneme v konstantním čase. Pro zvednutí nám tedy stále stačí čas $\mathcal{O}(N)$ a libovolné převedení přebytku zvládneme v $\mathcal{O}(1)$.

Třídění

Definice: *Komparátorová síť* je kombinační obvod, jehož hradla jsou komparátory:

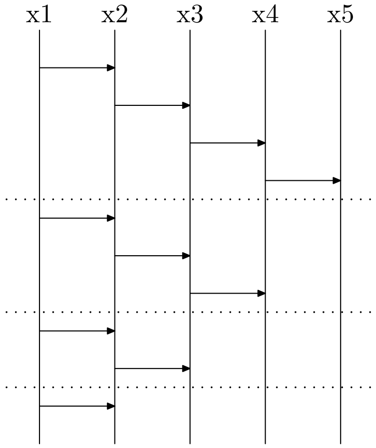


Komparátor dostane na vstupu dvě čísla, porovná je a na levý výstup vrátí menší z nich, na pravý výstup naopak vrací větší číslo ze zadané dvojice.

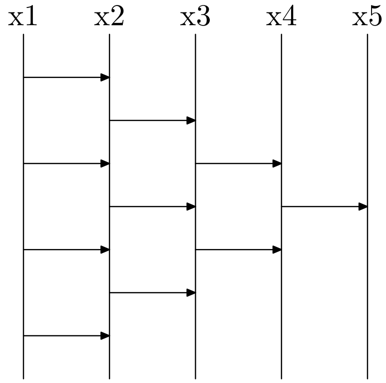
Výstupy komparátorů se nevětví.

Příklad: *Bubble sort*

Obrázek Bubble.1 ilustruje použití komparátorů pro třídění bubble sortem. Šipky představují jednotlivé komparátory.



Bubble.1



Bubble.2

Snažíme se výpočet co nejvíce paralelizovat (viz obrázek Bubble.2). Takto se nám podařilo výpočet provést pomocí $\mathcal{O}(n^2)$ komparátorů rozmístěných na $\mathcal{O}(n)$ hladinách. Třídíme v čase n a prostoru n^2.

Definice: Řekneme, že posloupnost $x_0, \ldots, x_{n-1}$ je *čistě bitonická*, pokud pro nějaké $x_j \in \{1, \ldots, n-1\}$ platí:

$$x_0 \leq x_1 \leq \ldots \leq x_j \geq x_{j+1} \geq \ldots \geq x_{n-1}.$$

Posloupnost je *bitonická*, pokud existuje $k \in \{1, \ldots, n-1\}$, pro které je rotace původní posloupnosti o k prvků, tedy posloupnost $x_k, x_{(k+1) \bmod n}, \ldots, x_{(k+n-1) \bmod n}$, čistě bitonická.

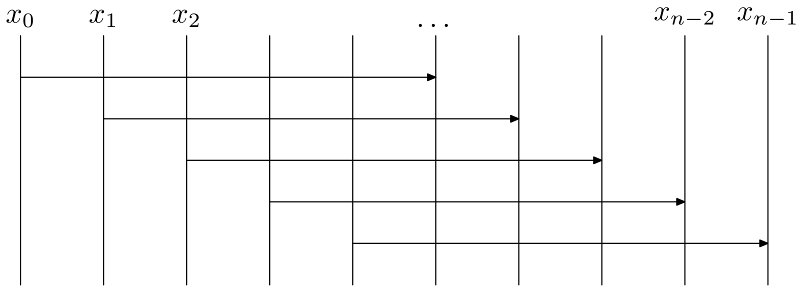
Definice: *Separátor S_n* je síť, ve které jsou vždy i-tý a $(i + n/2)$-tý prvek vstupu (pro $i = 0, \ldots, n/2 - 1$) propojeny komparátorem, minimum bude i-tým, maximum $(i + n/2)$-ním prvkem výstupu.

Lemma: Pokud vstup S_n obvodu je bitonická posloupnost, pak výstup $y_0, \ldots, y_{n-1}$ je posloupnost, která splňuje:

(i) $y_0, \ldots, y_{n/2-1}$ a $y_{n/2}, \ldots, y_{n-1}$ jsou bitonické posloupnosti,

(ii) Pro všechna $i, j < n/2$ platí $y_i < y_{j+n/2}$.

Důkaz: (i) Nejprve nahlédneme, že lemma platí, je-li vstupem čistě bitonická posloupnost. Tehdy najdeme nejmenší k takové, že x_k a $x_{k+n/2}$ se prohodí. (Pokud

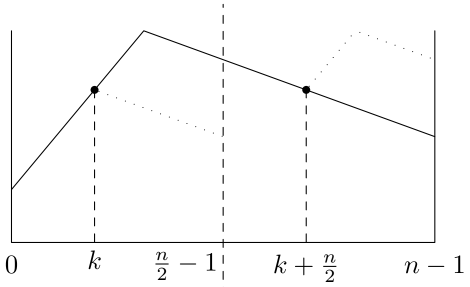


$(y_i, y_{i+n/2}) = CMP(x_i, x_{i+n/2})$

takové k neexistuje, separátor pouze zkopíruje vstup na výstup a obě tvrzení lemmatu zřejmě platí.) Řekněme, že x_m je maximum vstupní posloupnosti. Pak k bude jistě menší než m a $k + n/2$ bude větší než m, mezi k a m je tedy vstupní posloupnost neklesající, mezi $k + n/2$ a $n - 1$ nerostoucí. Uvědomíme si, že pro každé i, $k \leq i \leq n/2 - 1$ se prvky x_i a $x_{i+n/2}$ prohodí. Úsek mezi k a $n/2 - 1$ tedy nahradíme nerostoucí posloupností, první polovina výstupu tedy bude (dokonce čistě) bitonická. Úsek $k+n/2$ a $n-1$ nahradíme čistě bitonickou posloupností, která bude neklesající, je-li $m \geq n/2$, v opačném případě je úsek $n/2-1$ až $k+n/2$ nerostoucí. V obou případech budou v posloupnosti maximálně dva zlomy a mezi $x_{n/2}$ a x_{n-1} bude správná nerovnost na to, aby posloupnost byla bitonická.

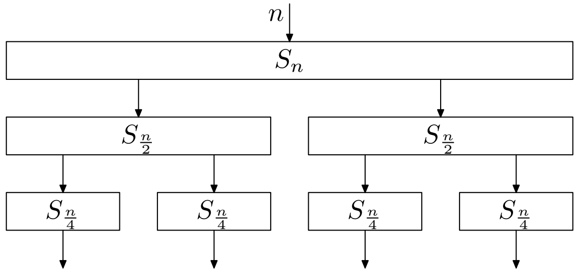
Dostaneme-li na vstupu obecnou bitonickou posloupnost, představíme si, že je to čistě bitonická posloupnost zrotovaná o r prvků (BÚNO doprava), a zjistíme, že v komparátorech se porovnávají tytéž prvky jako kdyby zrotovaná nebyla. Výstup se od výstupu čistě bitonické posloupnosti zrotovaného o r bude lišit prohozením úseků x_0 až x_{r-1} a $x_{n/2}$ až $x_{n/2+r-1}$. Obě výstupní posloupnosti tedy budou zrotované o r prvků, ale na jejich bitoničnosti se nic nezmění.

(ii) Z důkazu (i) pro čistě bitonickou posloupnost víme, že $y_0 \dots y_{n/2-1}$ čistě bitonická a bude rovna $x_0 \dots x_{k-1}, xk + n/2 \dots x_{n-1}$ pro vhodné k a navíc bude mít maximum v x_{k-1} nebo $x_k + n/2$. Mezi těmito body ovšem ve vstupní posloupnosti určitě neležel žádný x_i menší než $x_k - 1$ nebo $x_k + n/2$ (jak je vidět z obrázku) a posloupnost $x_k \dots x_{k-1+n/2}$ je rovna $y_{n/2} \dots y_{n-1}$. Pro obecné bitonické posloupnosti ukážeme stejně jako v (i). ♡



posloupnost prohozená separátorem

Definice: *Bitonická třídička* B_n je obvod sestavený ze separátorů, který dostane-li na vstupu bitonickou posloupnost délky n (konstruujeme třídičku pro $n = 2^k$), vydá setříděnou posloupnost délky n.

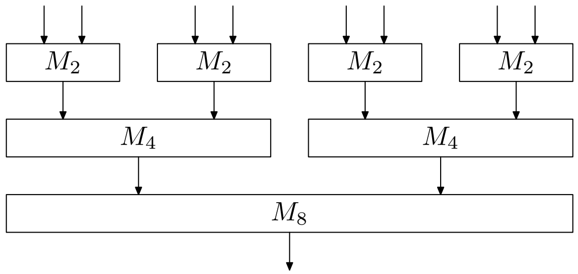


Bitonická třídička B_n

Separátor má jednu hladinu s $\mathcal{O}(n)$ hradly, třídička tedy bude mít $\log n$ hladin s $\mathcal{O}(n \log n)$ hradly.

Příklad: *Merge sort*

Bitonická třídička se dá použít ke slévání setříděných posloupností. S její pomocí sestavíme součástky mergesortové sítě. Setříděné posloupnosti $x_0, \dots, x_{n-1}$ a $y_0, \dots, y_{n-1}$ spojíme do jedné bitonické $x_0, \dots, x_{n-1}, y_{n-1}, \dots, y_0$. Z takové posloupnosti pomocí B_{2n} vytvoříme setříděnou posloupnost. Blok M_{2n} sestává z bloku B_{2n}, jehož druhá polovina vstupů je zapojena v obráceném pořadí.



Z bitonických třídiček tedy můžeme postavit mergesortovou síť, která bude mít $\mathcal{O}(\log^2 n)$ hladin a $\mathcal{O}(n \log^2 n)$ hradel.

Existuje třídicí algoritmus, kterému stačí $\mathcal{O}(\log n)$ hladin, ale jeho multiplikativní konstanta je příliš veliká, takže je v praxi nepoužitelný.

6. Vyhledávání v textu *(zapsal K. Kaščák, M. Klaučo, M. Vachna)*

Úkol: V textu s délkou S najít všechny výskyty hledaného slova s délkou J.

Hloupý algoritmus

Algoritmus prochází sekvenčně textem a hledaným vzorovým slovem. Při neshodě se ve vzorovem slově vrací na začátek a v textu pokračuje znakem, ve kterém nastala neshoda. Časová složitost je $\mathcal{O}(S)$. Tento algoritmus funguje pouze pro vzorová slova, ve kterých se neopakuje první znak.

Příklad: Hledání vzorového slova `jehla` v textu `vkupcejejehla`. Ve chvíli kdy máme prefix `je` a na vstupu dostaneme `j`, dochází k neshodě a pokračujeme v hledání od tohoto znaku. Pro tenhle případ algoritmus najde vzorové slovo. To ale již neplatí pro vzorové slovo `kokos` v textu `clanekokokosu`. Ve chvíli kdy máme prefix `koko` a na vstupu dostaneme `k`, dochází k neshodě a pokračujeme v hledání od tohoto znaku, tím ale zahodíme potřebnou část a algoritmus selže.

Neefektivní algoritmus

Algoritmus prochází text od začátku až do konce a pro každou pozici v textu zkontroluje, zda na této pozici nezačíná hledané slovo. Tak pro každou pozici provede až S porovnání znaků, čili celkem až SJ porovnání. Proto je časová složitost $\mathcal{O}(SJ)$.

Chytrý algoritmus

Algoritmus je vylepšením Neefektivního algoritmu, konkrétně způsobu, jakým sa vrací v textu při neshodě mezi znakem textu a znakem vzorového slova.

Příklad: Pro vzorové slovo `ajaajak` jsme našli v textu prefix `ajaaja`. Očekávame `k`.

- Když ale dostaneme `a` a budeme mít prefix `ajaajaa`, vracíme se v textu za první `aja`, tedy prefix zkrátíme na `ajaa` a pokračujeme v hledání.
- Když je nasledující znak `j` a budeme mít prefix `ajaajaj`, vracíme se v textu za `ajaaj`, tedy prefix zkrátíme na `aj` a pokračujeme v hledání.
- V případě, že dostaneme jiný znak, se v textu nevracíme a pokračujeme dalším znakem textu.

Definice a značení pro řetězce (slova):

Definice:

- *Abeceda* Σ je konečná množina znaků, ze kterých tvoříme text, řetězce, slova jako konečné posloupnosti znaků z Σ. Příkladem extrémních abeced je binární abeceda složená z nul a jedniček. Příklad z druhého konce je abeceda, která má jako znaky slova českého jazyka. V algoritmech nebudeme uvažovat velikost abecedy (počet znaků), budeme předpokládat, že je to konstanta.

- Σ^* je množina všech slov nad abecedou Σ.

Značení:

- *Slova* budeme značit malými písmeny řecké abecedy α,β… a *znaky* malými písmeny latinky a,b… .
- *Prázdné slovo* značíme písmenem ε.
- *Délka slova* $|\alpha|$ pro $\alpha \in \Sigma^*$ je počet jeho znaků.
- *Zřetězení* $\alpha\beta$ vznikne zapsáním slov α a β za sebe. Platí $\alpha\varepsilon = \varepsilon\alpha = \alpha$, $|\alpha\beta| = |\alpha| + |\beta|$.
- $\alpha[i]$ je i-té písmeno slova α, indexuje se od 0.
- $\alpha[i:j]$ je podslovo tvořené písmeny $\alpha[i],\ldots,\alpha[j-1]$. Příklady: $\alpha[i:i+1] = \alpha[i]$, $\alpha[i:i] = \varepsilon$. Vynechaním první meze získame prefix ($\alpha[:j]$), druhé meze suffix ($\alpha[i:]$), obou mezí dostaneme celé slovo ($\alpha[:]=\alpha$).
- $\alpha[:j]$ je *prefix* obsahující prvních j znaků slova α.
- $\alpha[i:]$ je *suffix* obsahující znaky slova α počínaje i-tým znakem.
- Každé slovo je prefixem i suffixem sebe sama, takovému prefixu/suffixu říkáme *nevlastní*. Všem ostatním *vlastní*.
- Prázdné slovo je podslovem, prefixem i suffixem každého slova včetně prázdného slova.

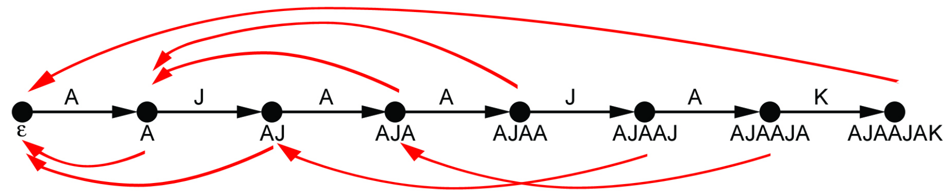
Problém:

Vstupem je ι hledané slovo (jehla) délky $J = |\iota|$ a σ text (seno) délky $S = |\sigma|$.

Výstupem jsou všechny vyskyty hledaného slova ι v textu σ: $\{i | \sigma[i:i+J] = \iota\}$

Vyhledávací automat (Knuth, Morris, Pratt)

Vyhledávací automat bude graf, jehož vrcholům říkame stavy automatu. Jména stavů budou všechny prefixy slova ι. Počáteční stav je prázdné slovo ε a koncový je celá ι. Dopředné hrany grafu budou popisovat přechod mezi stavy ve smyslu zvětšení délky jména stavu (dopředná funkce $d(\alpha, X)$), tedy každá taková hrana bude označena písmenem X a bude popisovat dané zvětšení délky jména stavu, tedy $\alpha \rightarrow \alpha X$. Zpětné hrany grafu budú popisovat přechod (zpětná funkce $z(\alpha)$) mezi stavem α a nejdelším vlastním suffixem α, který je prefixem ι, když nastane neshoda.



Vyhledávací automat

Vyhledávání:

1. $\alpha \leftarrow \varepsilon$.
2. Pro $c \in \Sigma$ postupně:
3. Dokud $\neg\exists d(\alpha, c) \wedge \alpha \neq \varepsilon : \alpha \leftarrow z(\alpha)$.
4. Když $\exists d(\alpha, c) \Rightarrow \alpha \leftarrow d(\alpha, c)$.
5. Když $\alpha = \iota \Rightarrow$ hledané slovo je v textu.

Alternatíva: Automat může být reprezentovaný i polem. Při této reprezentaci odpadá starost o dopřední hrany (stačí zvětšit hodnotu, kterou v poli indexujeme). Hodnota na dané pozici v poli určuje kam směruje zpětná hrana (index v poli).

Alternatívní vyhledávání:

1. $k \leftarrow 0$.
2. pro $c \in \Sigma$ postupně:
3. Dokud $c \neq \iota[k] \wedge k > 0 : k \leftarrow z[k]$
4. Je-li $c = \iota[k] \Rightarrow k \leftarrow k + 1$
5. Když $k = J \Rightarrow$ hledané slovo je v textu

Invariant: Nejdelší suffix β, který je prefixem $\iota = \alpha(\beta)$. Kde β je přečtení vstup. Z invariantu vyplýva korektnost vyhledávací části algoritmu KMP.

Důkaz: Indukcí podle $|\beta|$. Na začátku pro prázdný načtený vstup platí invariant, tedy prázdny suffix β je prefixem ι. V kroku n máme načtený vstup β a k němu načteme znak c. Jestli si odmyslíme c, tedy když si od jména stavu odmyslíme poslední písmenko, dostaneme znovu jméno stavu. Tak stav, který pasuje na konec vstupu bez toho c je stav, který pasuje na konec původního vstupu, toho o jeden znak kratšího. Tím pádem to musí být něco, co je maximálně tak dlouhé jako původní stav, u kterého jsme byli, protože to byl nejdelší, který pasoval. Stačí procházet postupně všechny stavy, které pasují na konec toho vstupu od nejdelšího k nejkratšímu a vzít první, který se dá rozšířit o c. To je přesně to, co algoritmus dělá. Protože zpětná funkce řekne nejbližší kratší jméno stavu. Takže algoritmus iteruje přes stavy, které tam pasují, až najde jeden, který se dá rozšířit o c a jelikož iteroval od toho nejdelšího, tak to je logicky ten nejdelší, který tam pasuje. ♡

Lemma: Vyhledávání doběhne v čase $\mathcal{O}(S)$.

Důkaz: Pro každý znak vstupního textu mohou nastat dva případy. Znak rozširuje aktuální prefix, nebo musíme použít zpětnou funkci (zpětnou hranu). Rozširování trvá konstantně mnoho času, zatímco zpětná funkce muže být pro jeden znak volána až J-krát. Při každém volání klesne délka aktuálního stavu minimálně o jedna a zároveň platí, že kdykoliv stav prodlužujeme, roste právě o jeden znak. Proto všech zkrácení dohromady muže být nejvýše tolik, kolik bylo všech prodloužení, t.j. kolik jsme přečetli znaku textu. Celkem je tedy počet kroků lineární vzhledem k délce textu. ♡

Konstrukce zpětné funkce:

1. Sestrojíme dopředné hrany

2. $z(\varepsilon) \leftarrow 0$, $z(\iota[0]) \leftarrow \varepsilon$
3. $\alpha \leftarrow \varepsilon$
4. pro $i = 1$ do J
5. $\alpha \leftarrow krok(\alpha, \iota[i])$
6. $z(\iota[0 : i + 1]) \leftarrow \alpha$

Vysvětlení: Všimněte si, že $z(i)$ je přesně stav, do nejž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec $\iota[2 : i]$, čili na i-tý prefix bez prvního písmenka. Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco α označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku. Takže $z()$ získáme tak, že spustíme vyhledávání na část samotného slova ι. Jenže k vyhledávání zase potřebujeme zpětnou funkci z. Proto budeme zpětnou funkci vytvářet postupne od nejkratších prefixů. Zřejmě $z(1) = \varepsilon$. Pokud již máme $z(i)$, pak výpočet $z(i+1)$ odpovídá spuštení automatu na slovo délky i a pritom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku, protože $(i+1)$-ní prefix je prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na celý řetězec ι a sledovat, jakými stavy bude procházet. To budou přesně hodnoty zpětné funkce. Vytvoření zpětné funkce se tak nakonec zredukovalo na jediné vyhledávání v textu o délce $J - 1$, a proto poběží v čase $\mathcal{O}(J)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(S + J)$.

Algoritmus Rabin & Karp

Tento algoritmus funguje tak, že porovnává hash hledaného řetězce s hashem aktuálního podřetězce („posuvné okénko“ stejné délky jako hledaný řetězec) v textu a aktuální podřetězec porovná se vzorkem pouze v případě, když mají shodný hash. Když si zvolíme tu správnou hashovací funkci, budeme moci vypočítat hash následujícího podřetězce na základe hashe toho aktuálního. Jako hashovací funkci $h : \Sigma^J \to \mathbb{Z}$ použijeme následující: $h(x_0, ..., x_{J-1}) = (\sum_{i=0}^{J-1} x_i.p^{J-1-i}) \bmod N$, kde N je velikost prostoru, do kterého hashujeme. Jak zjistíme hash následujícího podřetězce?

- $h = x_0.p^J + x_1.p^{J-1} + ... + x_{J-1}.p^1$
- $h' = x_1.p^J + x_2.p^{J-1} + ... + x_J.p^1$
- $h' = (h - x_0.p^J).p + x_J.p^1$

Tady můžeme vidět, že hash následujícího řetězce lze přepočítat na základě toho předchozího v konstantním čase. Časová složitost je v nejlepším případě lineární vzhledem k délce textu, zatímco nejhorší případ múže trvat až $\Theta(JS)$.

7. Vyhledávání v textu *(zapsali J. Kunčar, M. Demin a J. Chludil)*

Na minulých prednáškách jsme si ukázali, jak se v textu (seně) vyhledává slovo (jehlu). Teď si ovšem úlohu zobecníme a ukážeme si, jak v kupce sena hledat současně více než jednu jehlu.

Hledání výskytu všech slov

- $\iota_1, \ldots, \iota_k$ – vyhledávaná slova (jehly) délek $J_i = |\iota_i|$
- σ – text (seno) délky $S = |\sigma|$

Nejprve si řekneme, jak chceme, aby vypadal výstup. Výstupem pro nás budou všechny uspořádané dvojice (i, j) (i je index jehly, kterou jsme nalezli, a j je počáteční pozice v seně, kde se jehla nachází) takové, že

$$\iota_i = \sigma[j : j + J_i].$$

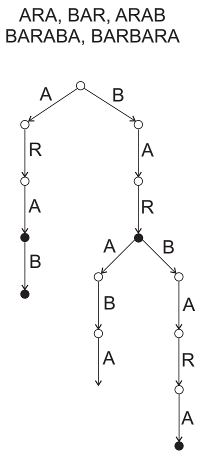
Postavme si proto vyhledávací automat podobný tomu, který jsme viděli na minulé přednášce. Tento automat nám všechny takové uspořádané dvojice najde.

Vyhledávací automat

Vyhledávací automat je strom[6], kde každý vrchol může mít stupeň až do velikosti abecedy a kde jednotlivé hrany odpovídají písmenům této abecedy. Vrcholy, ve kterých končí slovo, jsou označené (na obrazcích černě). Dále si časem do tohoto vyhledávacího stromu přidáme zpětné hrany a „zkratky“.

Stavy automatu jsou určeny vrcholy stromu, pro které platí rovněž stejný *invariant* z předchozí přednášky.

Zpětná hrana $z(\alpha)$:= nejdelší vlastní suffix[7] slova α, který je stavem.



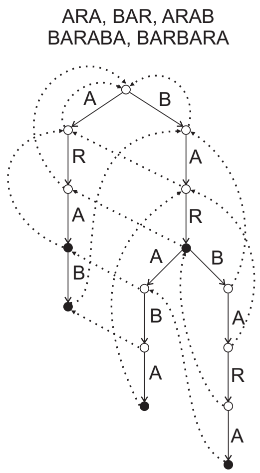
Vyhledávací automat

[6] http://en.wikipedia.org/wiki/Trie
[7] definováno na minulé přednášce

Výstup z automatu

Při vypisování výsledků mužeme narazit na určité problémy, které jsou dobře vidět na následujícím obrázku. První problém určitě nastane, protože v automatu není přesně řečeno, které slovo končí v jakém vrcholu. Například ve stavu, kde končí slovo BARBARA, končí také slovo ARA, ale o tom nevíme. Druhý problém nastává, když v automatu není informace o konci slova. Příkladem je seno BARAB (jednoduché k nahlédnutí, viz obrázek). Teď nám nezbývá nic jiného, než najít řešení těchto záludných problémů. Řešeší se nám naskýtá hned několik:

- Projdeme všechny zpětné hrany a vypíšeme slova, jež v daných stavech končí. Toto řešení funguje, ale je pomalé, protože pokaždé procházíme všechny zpětné hrany.
- Pro následující řešení, jenž spočívá v nalezení zkratek ve stromě, si zavedeme toto značení:
  slovo(s) = index slova ι, které končí ve stavu s, nebo $\emptyset$
  out(s) = nejbližší vrchol, do kterého se lze z s dostat po zpětných hranách a *slovo(v)* $\neq 0$ (končí tam slovo)



Vyhledávací automat se zpětnými hranami

Podle posledního bodu vytvoříme algoritmus na vyhledávání „jehel v seně“.

1. $s \leftarrow$ *kořen* (s bude aktuální stav vyhledávacího automatu).
2. Procházíme všechny písmena c v seně σ:
3. $\quad s \leftarrow krok(s, c)$.
4. $\quad$ Je-li $slovo(s) \neq 0$, vypíšeme $slovo(s)$.
5. $\quad v \leftarrow out(s)$.
6. $\quad$ Dokud $v \neq 0$:
7. $\quad\quad$ Vypíšeme $slovo(v)$.

8. $\qquad v \leftarrow out(v)$.

krok(s,c):= jeden *krok* vyhledávacího automatu:

1. Dokud $\nexists f(s,c) \wedge s \neq$ *kořen*: $s \leftarrow z(s)$.
2. Pokud $\exists f(s,c)$: $s \leftarrow f(s,c)$.
3. Vrátíme s.

Reprezentace v paměti

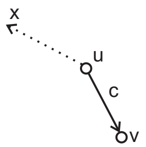
První možnost, jak reprezentovat vyhledávací automat, je jednorozměrné pole vrcholů stromu, v němž ukládáme seznam synů pro každý vrchol. Je to jednoduchá varianta, ale má nevýhodu pro velké abecedy, protože procházení seznamu synů může trvat neúměrně dlouho. Proto se nabízí druhá možnost a to hashovací tabulka $(stav, znak) \rightarrow f(stav, znak)$, kde se „ztratí“ používání hashovací funkce.

Složitost

- Kroky 2.–5. mají časovou složitost $\mathcal{O}(|\sigma|)$, kterou jednoduše dokážeme pomocí potenciálu – počet kroků nahoru je menší nebo roven počtu kroků dolů. A to je maximálně S.
- Kroky 6.–8. mají časovou složitost $\mathcal{O}(\textit{počet výskytů})$, protože rychleji doopravdy nelze všechny výskyty vypsat.

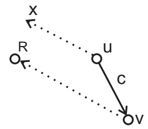
Konstrukce automatu (Aho, Corasicková)

1. Postavíme strom dopředných hran, $r \leftarrow$ kořen stromu.
2. Spočteme $slovo(*)$ – označíme si stavy, kde končí slova.
3. Spočteme $z(*)$: $z(\beta) = \alpha(\beta[1:])$:
   - $z(\beta) = \alpha(\beta[1:])$ – všechny zpětné hrany vedou do vyšších hladin
   - $z(v) = krok(z(u), c)$

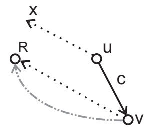


$z(v) = krok(z(u), c)$

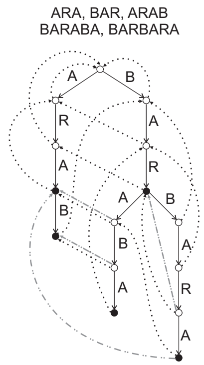
4. $z(r) \leftarrow 0$, do fronty Q přiřadíme všechny syny r, pro všechny v prvky $Q : z(v) \leftarrow r$.
5. Dokud fronta Q není prázdná:
6. $\qquad u \leftarrow$ vybereme z Q.
7. $\qquad$ Pro syny v vrcholu u:
8. $\qquad\qquad R \leftarrow krok(z(u))$ [znak na hraně uv].



$z(v) = R$



Nastavení $out(v)$



Vyhledávací automat – kompletní

9. $\qquad z(v) \leftarrow R$.
10. $\qquad$ Je-li $slovo(R) \neq 0 \Rightarrow out(v) \leftarrow R$, jinak $out(v) \leftarrow out(R)$.

Věta: Algoritmus Aho-Corasicková najde všechny výskyty slov $\iota_1, \dots, \iota_k$ ve slově σ v čase $\mathcal{O}(\sum_i |\iota_i| + |\sigma| + \textit{počet výskytů})$.

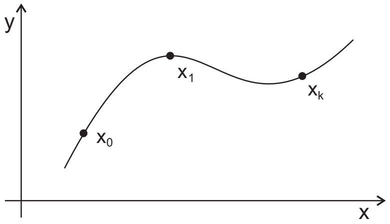
Polynomy a násobení

Mějme dva polynomy definované jako:

$$P(x) = \sum_{j=0}^{n-1} p_j x^j, \quad Q(x) = \sum_{j=0}^{n-1} q_j x^j.$$

Násobení dvou polynomů $R = P \cdot Q$ je ekvivalentní s operací $R = \sum_{j,k} p_j q_k x^{j+k}$. Přičemž na vypočítání členu $r_l = \sum_{j=0}^{l} p_j q_{l-j}$ použijeme $\Theta(n)$ operací, tedy na spočítaní celého polynomu R potřebujeme $\Theta(n^2)$ operací.

Podíváme se na jinou možnost, jak tento problém řešit. Poslouží nám k tomu následující věta o jednoznačné existenci polynomu nejvýše k-tého stupně, pokud známe hodnoty ve více než k bodech.



Polynom

Věta: Jsou-li $x_0, \dots, x_k \in \mathbb{R}$ navzájem ruzná a $y_0, \dots, y_k \in \mathbb{R}$, pak $\exists!$ polynom P stupně $\le k : \forall j : P(x_j) = y_j$.

Plán:

Nechť $k = 2^{n-1}$. Zvolíme čísla $x_0, \dots, x_k$ libovolná, ale různá, a spočteme $P(x_0)$, $\dots$, $P(x_k)$ a $Q(x_0), \dots, Q(x_k)$. Poté $\forall j : y_j = P(x_j)Q(x_j)$ musíme najít polynom R stupně $\le k : \forall j : R(x_j) = y_j$.

Vyhodnocování polynomů (metodou Rozděl a panuj)

BÚNO $n = 2^m$. Uvažme polynom:

$$P(x) = p_0x^0 + p_1x^1 + \ldots + p_{n-1}x^{n-1}.$$

Tento polynom si mužeme rozdělit na dvě časti. V levé budeme mít členy se sudými exponenty a v pravé budou členy s exponenty lichými:

$$P(x) = (p_0 + p_2x^2 + \ldots + p_{n-2}x^{n-2}) + (p_1x^1 + p_3x^3 + \ldots + p_{n-1}x^{n-1}).$$

Z pravé strany můžeme vytknout x a dostaneme:

$$P(x) = (p_0 + p_2x^2 + \ldots + p_{n-2}x^{n-2}) + x(p_1 + p_3x^2 + \ldots + p_{n-1}x^{n-2}),$$

$$\vdots$$

$$P(x) = L(x^2) + xN(x^2),$$
$$P(-x) = L(x^2) - xN(x^2),$$

kde $L(x)$ a $N(x)$ jsou polynomy stupně $n/2$. Umocněním x^2 se nám poruší párování x a $-x$, proto musíme počítat v $\mathbb{C}$ místo $\mathbb{R}$. V tomto případě jsme z polynomu s n koeficienty v n bodech dostali 2 polynomy s $n/2$ koeficienty v $n/2$ bodech. Z toho vyplývá časová složitost definována vztahem:

$$T(n) = 2T(n/2) + \mathcal{O}(n).$$

Ten můžeme vyřešit s použitím Master Theoremu z ADS I a dostaneme:

$$T(n) = \mathcal{O}(n \log n).$$

8. Fourierova transformace *(K.Jakubec, M.Polák a G.Ocsovszky)*

Násobení polynomů může mnohým připadat jako poměrně (algoritmicky) snadný problém. Asi každého hned napadne „hloupý“ algoritmus – vezmeme koeficienty prvního polynomu a vynásobíme každý se všemi koeficienty druhého polynomu a příslušně u toho sečteme i exponenty (stejně jako to děláme, když násobíme polynomy na papíře). Pokud stupeň prvního polynomu je n a druhého m, strávíme tím čas $\Omega(mn)$. Pro $m = n$ je to kvadraticky pomalé. Na první pohled se může zdát, že rychleji to prostě nejde (přeci musíme vždy vynásobit „každý s každým“). Ve skutečnosti to ale rychleji fungovat může, ale k tomu je potřeba znát trochu tajemný algoritmus FFT neboli *Fast Fourier Transform*.

Trochu algebry na začátek:

Libovolný polynom P stupně n může být reprezentován dvěma různými způsoby:

- svými koeficienty, čili čísly $p_0, p_1, \dots, p_n$, nebo
- svými hodnotami v n různých bodech $x_0, x_1, \dots, x_n$, čili čísly $P(x_0)$, $P(x_1)$, $\dots$, $P(x_n)$.

Konvence:

Celé polynomy označujeme velkými písmeny, jednotlivé členy polynomů pak příslušnými malými písmeny. (Př.: Polynom W stupně n má koeficienty $w_0, w_1, w_2, \dots, w_n$.)

Povšimněme si jedné skutečnosti – máme-li dva polynomy A a B stupně n a body $x_0, \dots, x_k$, dále polynom $C = A \cdot B$ (stupně $2n$), pak platí $C(x_k) = A(x_k) \cdot B(x_k), k = 0, 1, 2, \dots, n$. Toto činí tento druhý způsob reprezentace polynomu velice atraktivním pro násobení. Problémem je, že typicky máme polynom zadaný koeficienty a ne hodnotami v bodech. Tím pádem potřebujeme nějaký hodně rychlý algorimtus (tj. rychlejší než kvadratický, jinak bychom si nepomohli oproti hloupému algoritmu) na převod polynomu z jedné reprezentace do druhé a zase zpět.

Dále bychom si měli uvědomit, že stupeň našeho výsledného polynomu C bude $\le 2n$ (kde n je stupeň výchozích polynomů). Pokud chceme polynom C reprezentovat pomocí jeho hodnot v bodech, musíme tedy vzít alespoň $2n$ bodů. Tímto končí malá algebraická vsuvka.

Idea, jak by měl algoritmus pracovat:

1. Vybereme $2n$ bodů $x_0, x_1, \dots, x_{2n-1}$.
2. V těchto bodech vyhodnotíme polynomy A a B.
3. Nyní již v lineárním čase získáme hodnoty polynomu C v těchto bodech (viz výše).
4. Převedeme hodnoty polynomu C na jeho koeficienty.

Je asi vidět, že klíčové jsou kroky 2 a 4. Vybrání bodů jistě stihneme pohodlně v lineárním čase a vynásobení samotných hodnot též (máme $2n$ bodů a $C(x_k) = A(x_k) \cdot B(x_k), k = 0, 1, 2, \dots, 2n-1$, takže na to nepotřebujeme více než $2n$ násobení).

Celý trik spočívá v chytrém vybrání oněch bodů, ve kterých budeme polynomy vyhodnocovat. Je na to potřeba vědět pár zajímavostí o komplexních číslech, na webové stránce přednášky jsou k dispozici slajdy, zde to bude zapsáno o trochu stručněji.

Vyhodnocení polynomu metodou Rozděl a panuj (algoritmus FFT):

Mějme polynom P řádu n a chtějme jej vyhodnotit v n bodech. Vybereme si body tak, aby byly spárované, čili $\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$. To nám výpočet urychlí, protože pak se druhé mocniny x_j shodují s druhými mocninami $-x_j$.

Polynom P rozložíme na dvě části, první obsahuje členy se sudými exponenty, druhá s lichými: $P(x) = (p_0x^0+p_2x^2+\ldots+p_{n-2}x^{n-2})+(p_1x^1+p_3x^3+\ldots+p_{n-1}x^{n-1})$.

$S(x^2) = p_0x^0 + p_2x^2 + \ldots + p_{n-2}x^{n-2}$, $L(x^2) = p_1x^1 + p_3x^3 + \ldots + p_{n-1}x^{n-1}$

Takže obecně $P(x) = S(x^2) + xL(x^2)$ a $P(-x) = S(x^2) - xL(x^2)$. Jinak řečeno, vyhodnocování P v n bodech se nám smrskne na vyhodnocení $S(x)$ a $L(x)$ (oba jsou polynomy stupně $n/2$ a vyhodnocujeme je nyní v x^2) v $n/2$ bodech (protože $(x_i)^2 = (-x_i)^2$).

Příklad: $3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$.

Teď nám ovšem vyvstane problém s oním párováním – druhá mocina přece nemůže být záporná a tím pádem už v druhé úrovni rekurze body spárované nebudou. Z tohoto důvodu musíme použít komplexní čísla – tam druhé mocniny záporné býti mohou. Jako $x_0, \ldots, x_{n-1}$ si zvolíme mocniny n-té primitvní odmocniny z jedné (označíme si ji jako ω). Máme n n-tých primitivních odmocnin z jedničky, rovnoměrně rozesetých po jednotkové kružnici, BÚNO $n = 2^k, k \in N$ (jinak viz slajdy Martina Mareše). Jednotlivé mocniny vypadají takto: $1, \omega, \omega^2, \ldots, \omega^{n-1}$, kde $\omega = e^{2\pi i/n}$.

Dvě poznámky:

- primitivní n-té odmocniny z jedničky jsou spárované, čili $\omega^j = -\omega^{n/2+j}$,
- umocníme-li všechny na druhou, vznikne nám $n/2$ $n/2$-tých odmocnin z jedné, které jsou i nadále spárované.

Celý algoritmus bude vypadat takto:

FFT(P, ω)

Vstup: $p_0, \ldots, p_{n-1}$, koeficienty polynomu P, a ω, $n-$tá odmocina z jedné.

Výstup: Hodnoty polynomu v bodech $1, \omega, \omega^2, \ldots, \omega^{n-1}$, čili čísla $P(1), P(\omega), P(\omega^2)$, $\ldots, P(\omega^{n-1})$.

1. Pokud $n = 1$, vrátíme P_0 a skončíme.
2. Jinak rozdělíme P na sudé a liché koeficienty a rekurzivně zavoláme FFT(S, ω^2) a FFT(L, ω^2).
3. Pro $j = 0, \ldots, n/2 - 1$ spočítáme:

$P(\omega^j) = S(\omega^{2j}) + \omega^j \cdot L(\omega^{2j})$.
$P(\omega^{j+n/2}) = S(\omega^{2j}) - \omega^j \cdot L(\omega^{2j})$.
j je z intervalu $[0, \frac{n}{2} - 1]$

Časová složitost: $T(n) = 2T(n/2) + \mathcal{O}(n) \Rightarrow$ složitost $\mathcal{O}(n \log n)$, stejně jako MergeSort.

Máme tedy algoritmus, který převede koeficienty polynomu na hodnoty tohoto polynomu v různých bodech . Potřebujeme ale také algoritmus, který dokáže reprezentaci polynomu pomocí hodnot převést zpět na koeficienty polynomu. K tomu nám pomůže podívat se na náš algoritmus trochu obecněji.

Definice: *Diskretní Fourierova transformace* (DFT) je funkce $f : \mathbb{C}^n \to \mathbb{C}^n$, kde $y = f(x) \equiv \forall j\ y_j = \sum\limits_{k=0}^{n-1} x_k \cdot \omega^k$.

Jak najít inverzní matici? Víme, že $\Omega = \Omega^T$ protože $\omega^{jk} = \omega^{kj}$.

Využijeme následující lemma:

Lemma:

$$\Omega_j \cdot \Omega_k = \begin{cases} 0 \ldots j \neq k \\ 1 \ldots j = k \end{cases}.$$

Poznámka: $\Omega_j \cdot \Omega_k$ myslíme skalární součin. Jsou-li $x = (x_0, \ldots, x_n)$ a $y = (y_0, \ldots, y_n)$ dva komplexní vektory, pak jejich skalární součin definujeme jako: $x^*y = \sum\limits_{i=0}^{n} \overline{x_i} y_i$.

V této definici $\overline{x}$ označuje číslo komplexně sdružené k číslu x.

Důkaz: Součin

$$\Omega_j\Omega_k = \sum_{l=0}^{n-1} \Omega_{jl}\overline{\Omega_{kl}} = \sum_l \omega^{jl}\overline{\omega^{kl}} = \sum_l \omega^{jl}\omega^{-kl} = \sum_l \omega^{(j-k)l} = \sum_{l=0}^{n-1} (\omega^{j-k})^l,$$

protože $\overline{\omega^{kl}} = \overline{\omega}^{kl} = (\frac{1}{\omega})^{kl} = \omega^{-kl}$.

- Pokud $j \neq k$, použijeme vzoreček pro součet geometrické posloupnosti, kde $a_1 = 1$ a $q = \omega^{(j-k)}$ a dostaneme $\frac{\omega^{(j-k)n}-1}{\omega^{(j-k)}-1} = \frac{1-1}{r-1} = \frac{0}{\neq 0} = 0$. Kde r je číslo různé od jedničky.
- Pokud $j = k$, pak $\sum\limits_{l=0}^{n-1} (\omega^0)^l = n$.

♡

Důsledek: $\Omega \cdot \overline{\Omega} = nE$.

Jedná se o násobení matic, čili prvek na pozici ij je 0 nebo n. $\Rightarrow \Omega^{-1} = \frac{1}{n}\overline{\Omega}$.

Našli jsme inverzi:

$\Omega(\frac{1}{n}\overline{\Omega}) = \frac{1}{n}\Omega \cdot \overline{\Omega} = E$, $\quad \Omega^{-1}_{jk} = \frac{1}{n}\overline{\omega^{jk}} = \frac{1}{n}\omega^{-jk} = \frac{1}{n}(\omega^{-1})^{jk}$, $\quad$ kde ω^{-1} je $\overline{\omega}$ a ω_n je n-tá primitivní odmocnina z jedničky.

Náš algoritmus počítá tedy i inverzní transformaci, pouze místo ω_n použijeme komplexně združené $\overline{\omega_n}$ a matici vynásobíme $(1/n)$. Což je skvělé – stačí znát pouze jeden algoritmus u kterého stačí v jednom případě použít transformovanou matici a vydělit n.

Výsledek: Pro $n = 2^k$ lze DFT na $\mathbb{C}^n$ spočítat v čase $\mathcal{O}(n \log n)$ a DFT^{-1} taktéž.

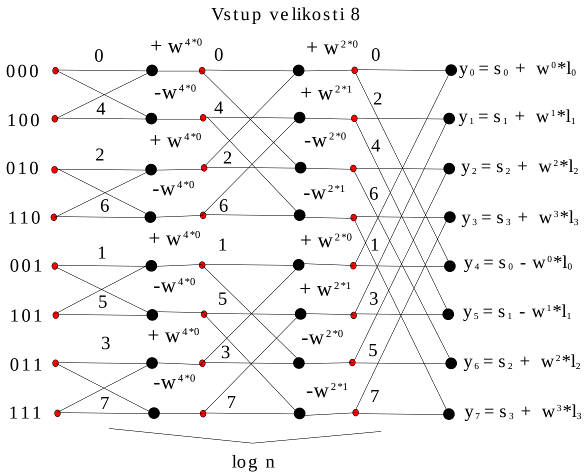
Důsledek:

Polynomy stupně n lze násobit v čase $\mathcal{O}(n \log n)$: $\mathcal{O}(n \log n)$ pro vyhodnocení, $\mathcal{O}(n)$ pro vynásobení a $\mathcal{O}(n \log n)$ pro převedení zpět.

Použití:

- Zpracování signálu – rozklad na siny a cosiny o různých frekvencích $\Rightarrow$ spektrální rozklad.
- komprese dat – například formát JPEG.
- Násobení dlouhých čísel v čase $\mathcal{O}(n \log n)$.

Hardwarová implementace FFT



Příklad průběhu algoritmu na vstupu velikosti 8

Obrázek ukazuje zapojení kombinačního obvodu pro DFT pro vstup velikosti 8. Hladin bude vždy $\log_2 n$, tj. v našem případě $\log_2 8 = 3$ hladiny.

Podívejme se na pravou část obrázku, tedy výstup celého obvodu. Černá kolečka představují podobvody, rovnice vedle nich operaci, kterou provádějí. Hodnoty y_j značí hodnotu polynomu P v bodě ω^j kde ω^j je $j - tá$ mocnina primitivní n-té odmocniny z jedničky. K jejímu spočtení ale potřebujeme znát hodnoty s_k a l_k kde k je z intervalu $[0, n/2 - 1]$ a s_k a l_k jsou hodnoty polynomu stupně $n/2$ v bodě ω^{2k}. V polynomu s jsou sudé koeficienty a v polynomu l liché koeficienty polynomu P. Vidíme ze se jedná přesně o náš rekurzivní algoritmus pro počítání FFT a tímto způsobem postavíme celou síť. Tímto obvodem jsme tedy získali nerekurzivní algoritmus pro počítání FFT. Všiměme si pořadí vstupních hodnot (koeficientů). Čísla jsou v binárním tvaru 0–7 přečtená pozpátku. Pro představu jaké koeficienty polynomu P se objevují v různých hladinách, na obrázku jsou naznačena jejich čísla spolu s příslušnými mocninami primitivní n-té odmocniny z jedničky.

Z toho:

- Kombinační obvod pro DFT s $\mathcal{O}(\log n)$ hladinami a $\mathcal{O}(n)$ hradly na hladině.
- Nerekurzivní algoritmus (postupujeme zleva) v čase $\mathcal{O}(n \log n)$.

9. Geometrické algoritmy *(B. Maslowski, J. Návrat, M. Staša)*

Budeme se teď bavit o geometrických problémech v rovině. Většina algoritmů, které zde uvedeme, má sice své obdoby i pro prostory vyšší nebo nižší dimenze, ale jednorozměrné případy bývají triviální a vícerozměrné jsou zase většinou moc složité.

Budeme se tedy zabývat tím, jak tyto problémy řešit v dimenzi dva (v Euklidovské rovině).

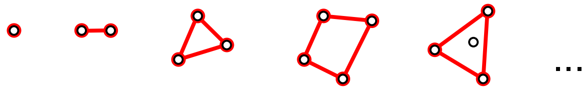
Hledání konvexního obalu

Ptáte se o co půjde? Zkusme si to přiblížit na problému ledních medvědů :) *Lední medvědi si po dlouhé době zmapovali vody severního moře a zjistili přesně místa, kde se nacházejí jejich oblíbené ryby. No a protože to jsou medvědi chytří, rozhodli se všechny tyto rybky pochytat najednou do jedné velké sítě. A problém, který tady mají, je následující: jaký nejmenší obvod může mít taková síť, aby se dovnitř vešly všechny rybky?!*

Neboli budeme řešit, jak nějakou zadanou množinu bodů v rovině obalit co nejkratší uzavřenou křivkou, do které se ještě všechny body vejdou.

Intuice nám napovídá že výsledek bude nějaký konvexní⟨8⟩ mnohoúhelník, který bude mít vrcholy v některých uvedených bodech. Ostatní vrcholy pak budou buď někde na hranách mnohoúhelníku, nebo uvnitř. Tomuto mnohoulehníku se říká *konvexní obal* dané množiny.

Možná by se teď hodilo předvést názorně, jak vypadají nejmenší konvexní obaly:



Základní obaly.

- Konvexní obal prázdné množiny je prázdná množina.
- Konvexní obal 1 bodu je bod samotný.
- Konvexní obal 2 bodů je úsečka spojující tyto body.
- Konvexní obal 3 bodů je trojúhleník s vrcholy v těchto bodech.
- Konvexní obal 4 bodů . . . to už je složitější. . .

Konvexní obaly 4 a více bodů, jak si můžeme všimnout, už nejsou jednoznačné. Pro N-prvkovou množinu bude konvexní obal mnohoúhelník se třemi až N vrcholy.

⟨8⟩ Množina bodů v rovině je konvexní, pokud platí, že pro každé dva body této množiny leží úsečka spojující tyto dva body také celá v této množině.

Jeden dobrý způsob, jak tento konvexní obal sestrojit se nazývá *Zametání roviny.*

Algoritmus funguje tak, že si v rovině zvolíme nějaký směr, a v tomto směru začneme posouvat přímku. Budeme takto potkávat body ležící v naší množině. V každém okamžiku budeme chtít, aby body v části, kterou jsme již zametli, už měli spočítaný konvexní obal. Vždy když pak zametací přímkou narazíme na nový bod, už si jen rozmyslíme, jak ho do konvexního obalu přidat.

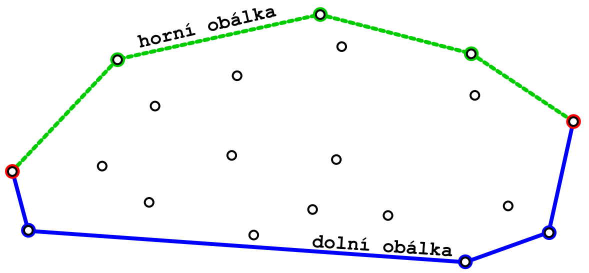
BÚNO předpokládáme body v obecné poloze, tedy takové, že žádné tři neleží na jedné přímce. Dále také budeme předpokládat, že budeme zametat ve směru x-ové osy a že všechny body mají různou x-ovou souřadnici.

Je také vidět, že bod s nejmenší a největší x-ovou souřadnicí bude ležet na konvexním obalu.

Myšlenka algoritmu:

1. Setřídíme body podle jejich x-ové souřadnice.
2. Vezmeme první tři body a sestrojíme jejich konvexní obal.
3. Opakuj: Najdeme další bod a podíváme se, jestli ho můžeme do konvexního obalu rovnou přidat:
4. Pokud jej můžeme rovnou přidat, tak jej přidáme.
5. Pokud jej přidat rovnou nemůžeme, pak je potřeba nejdříve nějaké body odzadu odebrat a pak teprve připojit náš nový bod.

Každá iterace tedy bude probíhat tak, že nějaké body z původního konvexního obalu pozapomínáme a přidáme nový bod. Aby se to lépe popisovalo, tak celý konvexní obal rozdělíme na *horní obálku* a *dolní obálku.*



Obrázek obálek.

Vidíme teď, že dolní obálka je nějaká lomená čára, která zatáčí doleva. Horní obálka zatáčí doprava. V našem algoritmu si budeme obálky pamatovat jako dva seznamy vrcholů. Když pak v algoritmu narazíme na nový bod, budeme zvlášť řešit jak to ovlivní horní obálku a jak ovlivní dolní. Je vidět že bod nejvíce vlevo a bod nejvíce vpravo leží v obou obálkách. Ostatní body buď leží jen v jedné z obálek, nebo neleží v žádné z nich (tedy nejsou součástí konvexního obalu).

Algoritmus:

1. Setřídíme body podle souřadnice x, dostaneme množinu bodů $b_1 - b_n$.
2. Spočítáme konvexní obal b_1, b_2, b_3, z toho získáme horní a dolní obálku[9].
3. Pro b postupně zpracováváme $b_3 - b_n$:
4. Přepočítáme Horní obálku:
5. Dokud ($|H| \geq 2$) a úhel $(H[-2], H[-1], b)$ je orientovaný doleva:
6. Odebereme poslední prvek z obálky.
7. Přidáme do obálky nový vrchol.
8. Přepočítáme dolní obálku:
9. Dokud ($|D| \geq 2$) a úhel $(D[-2], D[-1], b)$ je orientovaný doprava:
10. Odebereme poslední prvek z obálky.
11. Přidáme do obálky nový vrchol.

Setřídit body podle x-ové souřadnice a sestrojit konvexní obal prvních třech bodů stihneme v čase $\mathcal{O}(n \log n)$. Zbytek pak už uděláme dokonce v čase lineárním $\mathcal{O}(n)$[10]. Platí tedy:

Věta: Konvexní obal dokážeme sestrojit v čase $\mathcal{O}(n \log n)$.

Naši lední medvědi se tedy již naučili, jak si efektivně obstarat potravu a mohly se pustit do řešení dalšího velmi důležitého problému. Pojďme se na něj podívat s nimi. A o co že to půjde?

Lední medvědi nejsou na antarktidě sami, kromě nich tam taky bydlí kamarádi eskymáci ve svých iglů. Medvědi by si teď rádi udělali mapu, podle které by hned poznali, ke kterému ekymákovi to mají nejblíže na návšťevu.

My tuhle medvědí mapu od teď budeme nazývat *Voroneho diagramem.*

Voroného diagramy

Před tím, než vás vystraším nějakou definicí, si řekneme, co jsi pod tímto, na první pohled ne zřejmým pojmem, představit. Mějme množinu teček T rozmístěných náhodně po papíru. Ke každému bodu nakreslíme okraje tak, aby vniklá ploška obsahovala body, které jsou nejblíže právě té naší vybrané tečce. Samozřejmě „sousední“ tečky budou mít tyto hranice společné. Výsledkem našeho dlouhého snažení pak bude právě Voroného diagram. V dalších odstavcích se budeme zajímat o to, jak takový útvar správně popsat, jak ho sestrojit a jaké datové struktury k tomu použít.

Definice: *Voroného diagram* pro konečnou množinu $M = \{m_1, \dots, m_n\} \in \mathbb{R}^2$ míst je systém množin $O_1, \dots, O_n$ takových, že pro všechna i a j a pro všechna $x \in M_i$ je

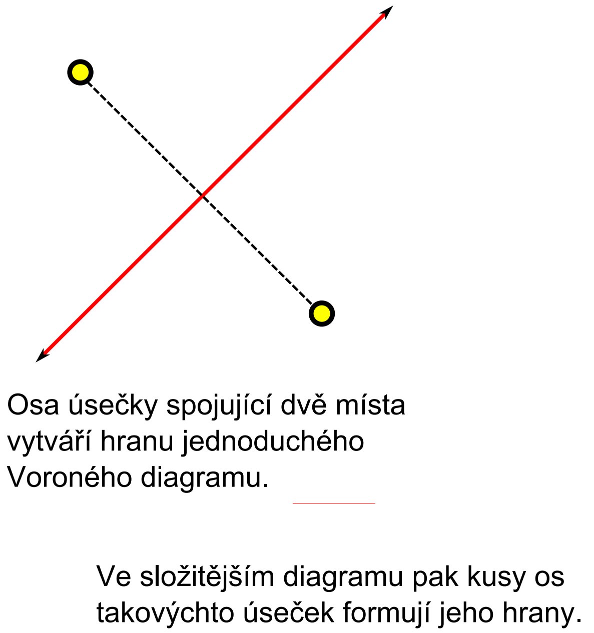
[9] Body b_1 a b_3 budou v obou obálkách. Bod b_2 bude v horní obálce pokud leží nad přímkou spojující b_1 a b_3, v dolní obálce bude pokud leží pod přímkou.

[10] V této části už jen do obálek přidáváme a odebíráme body.Přidáváme jich N. A odebrat jich můžeme maximálně tolik kolik jsme jich přidali. Tedy zase maximálně N.

vzdálenost x od m_i menší nebo rovna vzdálenosti x od m_j a zároveň sjednocení O_i přes všechna i je celý prostor $\mathbb{R}^2$, neboli:

$$d(x, m_i) \leq d(x, m_j) \wedge \bigcup_i O_i = \mathbb{R}^2.$$

Jednoduchý Voroného diagram:



Voroneho diagramu pro dvě místa.

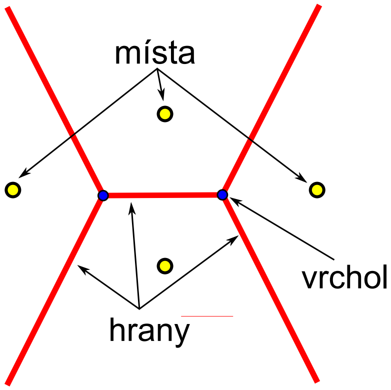
Voroného diagram se tedy skládá z nějakých míst, oblastí a hran, které ty oblasti oddělují.

Definice: Řekneme, že *hrana* H je taková množina bodů, že pro každý bod $x \in H$ platí, že existují dvě místa m_i a m_j, od kterých má bod x stejnou vzdálenost. Tyto dvě místa jsou pro všechny body x stejná a platí, že všechny ostatní místa mají od každého bodu x delší vzdálenost.

Definice: Řekneme, že *vrchol* je takový bod, kde se potkávají alespoň dvě hrany.

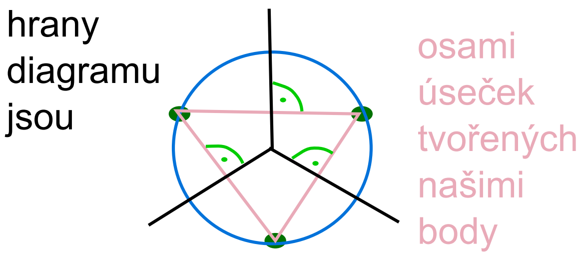
Pozorování:

- Voroneho diagramem pro dvě místa jsou dvě poloroviny odělené takovou přímkou, že každý bod přímky je stejně vzdálený od obou míst.
- Každá množina M_i je ohraničena konvexní lomenou čarou, takže oblasti mají tvar konvexních mnohoúhelníků, ale je možné, že jsou oteveřené do neknečna.



Části Voroneho diagramu.

- Pro každou hranu h ve Voroného diagramu existuje i a j takové, že když $x \in H$, pak vzdálenost $d(x, m_i) = d(x, m_j)$.
- Pro každý vrchol v Voroného diagramu existují alespoň tři místa ležící na kružnici se středem v.



Body na kružnici.

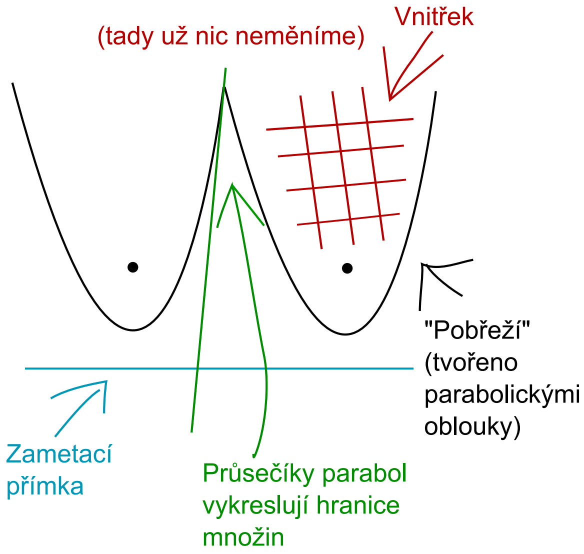
- Počet vrcholů a hran je lineární k počtu míst[11].
- Počet krajních oblastí je tak velký, jak velký je konvexní obal té zadané množiny. (Je to dobré vědět, ale asi to nebudeme potřebovat.)

Pojďme teď vymyslet, jak takový diagram vyrobit. Mohli bychom zkonstruovat všechny dělící přímky a poslepovat je, ale vznikl by nám kvadratický algoritmus a to nám nemůže stačit.

Mluvili jsme o zametání roviny, a tak bychom tento trik mohli využít právě při řešení našeho problému. Ovšem tentokrát má zametání jeden podstatný háček. Když

[11] Voroneho diagram si lze představit jako graf, kde místa Voroneho diagramu odpovídají stěnám, vrcholy diagramu vrcholům a hrany odpovídají hranám grafu. Pokud si teď někam mimo graf přidáme ještě jeden vrchol a všechny přímky vedoucí do nekonečna navedeme do toho bodu, vidíme, že náš graf je roviný. Pro roviný graf platí Eulerova formule a z ní už plyne že naše linearita.

si vezmeme nějakou zametací přímku a pojedeme s ní shora dolů, tak nad ní máme nějakou už zkonstruovanou část diagramu a když narazíme na další bod, tak se nám může právě tato část diagramu poměrně složitě změnit. Pomůžeme si malým trikem. Nebudeme považovat za hotovou celou oblast nad zametací přímkou, ale jen takové body, které mají blíže k místům (m_i) než k zametací přímce. Tak dostaneme nějakou posloupnost (množinu) parabol. Všechno, co jsme spočítali uvnitř této oblasti nám už nikdo nepokazí (ani nevylepší), je tam bezpečno. Vezmeme si tedy dolní obálku těchto parabol, budeme jí říkat *pobřeží*.



Pobřežní linie.

Pobřeží je tedy nějaká posloupnost parabolických oblouků s tím, že nejlevější a nejpravější jdou do nekonečna. Průsečíky těchto oblouků vykreslují hrany diagramu. Proč? Odpověď na tuto otázku není těžká, stačí vyjít z definice paraboly tak, jak jí zde používáme. Nebo-li je to množina bodů, která je od ohniska (pro nás místa) stejně vzdálená jako od přímky (řídící). A tedy průnik dvou parabol je místo, které je stejně vzdálené od obou ohnisek, což je vlastně definice bodu ležícího na nějaké hraně.

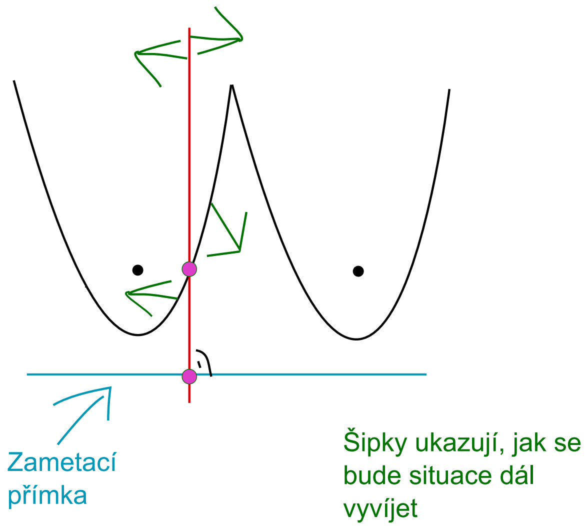
Kdykoliv v průběhu zametání narazíme na nějaký bod, může nám ovlivnit už jen část diagramu pod pobřežím. Dostáváme se tedy k tomu, co se děje, když hýbeme zametací přímkou. Pakliže nenarážíme na žádné body, tak se v podstatě neděje nic zajímavého. Zajímavá situace nastává až tehdy, narazíme-li na další bod. V tom okamžiku vzniká nová parabola. V tuto chvíli je značně degenerovaná. Je to totiž zatím jen polopřímka kolmá na zametací přímku. S dalším pohybem se začne parabola rozevírat. Všimněme si, že průnik oné přímky a pobřeží je vlastně vrchol Voroného diagramu.

Může nastat ještě jeden problém. Nějaká parabola se může natolik rozevřít, že pohltí jiné a ty zmizí z pobřežní linie. V takovém případě, se nám ale netriviálně změní vzhled pobřeží, a proto se této situaci budeme muset více věnovat.

Shrneme-li naše úvahy, mohou se dít celkem tři věci. Jedna z nich je posun přímky. To se vlastně děje pořád. Pobřeží se téměř nemění a průsečíky parabol nám kreslí hrany. To můžeme počítat najednou. Navíc nejen že bychom mohli s přímkou skočit o nějaké epsilon, ale my dokonce můžeme skočit o pořádný kus a prostě pouze dopočítat, jak se pobřeží změnilo a co se vykreslilo. Důležitým místům, kde se budeme zastavovat, budeme říkat *události*.

Místní událost

Pokud narazíme na bod, musíme najít místo, kde pobřeží rozetnou a kam vklínit další výběžek (parabolu). Takovéto události budeme říkat místní událost.

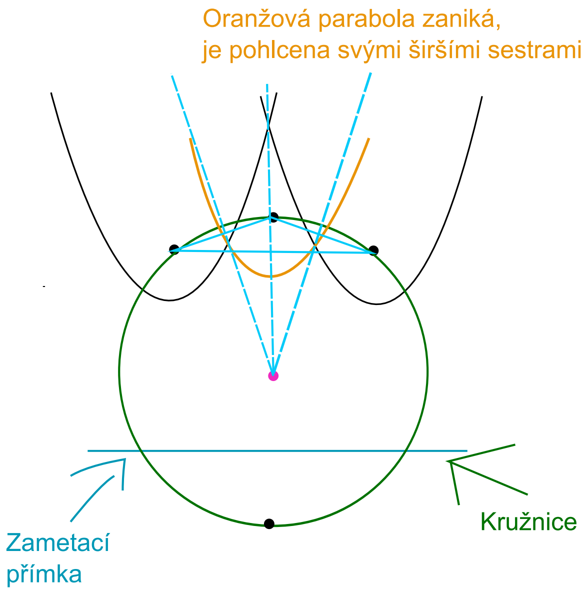


Místní událost – červená kolmice je nově vznikající parabola, při postupu zametací přímky dále se bude rozevírat a vytvoří další parabolu.

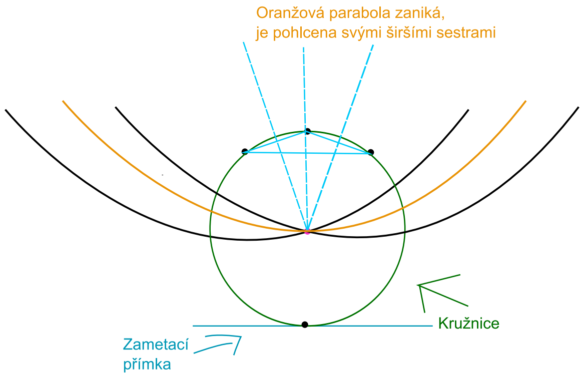
Kružnicová událost

Poslední situace, která může nastat, je, že se nějaká parabola schová za jiné. Koukněme se na první obrázek níže, fialový bod je střed kružnice opsané trojúhelníku tvořenému třemi místy. Jak víme, ten leží na osách stran takového trojúhelníku. Po těchto osách se však budou i pohybovat průsečíky parabol a s posunováním řídící přímky se pak dostanou všechny tři do středu této kružnice. Stane se to právě tehdy, když se zametací přímka dotkne kružnice zespodu. Je možné nahlédnout, že při postupu dále se pak dvě krajní paraboly rozšíří natolik, že prostřední pohltí a ta zanikne. Takto vzniklé události budeme říkat kružnicová.

Pojďme si to ukázat lépe na následujících dvou obrázcích. První představuje situaci před kružnicovou událostí a druhý právě kružnicovou událost. Mimo jiné by tedy z obrázků mělo být patrné, že kružnicové události jsou určeny trojicemi sousedních oblouků v pobřeží.



Před kružnicovou událostí.



Kružnicová událost.

Datové struktury

Budeme potřebovat haldu událostí (místních i kružnicových dohromady).

Dále bude zapotřebí udržovat si pobřežní linii, neboli posloupnost míst v ohniscích parabolických oblouků. Zde je potřeba si definovat operace *Insert*, *Delete* a

FindX, jinak řečeno přidat a odebrat oblouk a najít oblouk podle x-sové souřadnice nebo-li oblouk do kterého jsme se trefili při místní události. Navíc budeme potřebovat vyhledávací strom nad průsečíky s implicitní reprezentací, což znamená, že si ve vrcholech nebudeme pamatovat přímo souřadnice průsečíků, ale jen instrukci, jak je spočítat. Takže jakkoli se mění poloha průsečíků, tak struktura stromu zůstává stejná.

S haldou událostí lze pracovat s logaritmickou časovou složitostí na operaci a ve stejném čase dokážeme pracovat s pobřežní linií. Když si pobřeží ještě reprezentujeme zvlášť jako seznam tak Insert a Delete budou v konstatním čase a operace se stromem pak $\mathcal{O}(\log n)$.

Poslední datovou strukturou bude samotný diagram, reprezentovaný grafem se souřadnicemi a vazbami hran na průsečík.

Fortunův algoritmus

1. Připravíme si haldu H a vložíme do ní všechny místní události.
2. Připravíme pobřežní linii $P \leftarrow 0$.
3. Dokud existuje $h \leftarrow DeleteMin(H)$:
4. Je-li na řadě místní událost:
5. Najdeme průsečík s $P(FindX(P))$.
6. Vložíme do P novou parabolu.
7. Poznamenáme do D vznik hran.
8. Přepočítáme kružnicové události.
9. Je-li na řadě kružnicová událost:
10. Smažeme oblouk z P.
11. Poznamenáme vznik a zánik do D.
12. Přepočítáme kružnicové události.

Složitost:

Počet místních událostí je roven n (na každé místo narazíme právě jednou). Počet kružnicových událostí není větší než n, protože kružnicová událost smaže parabolu a ty vznikají jen při místních událostech, takže kružnicových událostí není více než místních. Speciálně z toho plyne, že velikost pobřežní linie je vždy lineární, protože s každou místní událostí přibudou dva úseky do pobřežní linie, takže velikost pobřežní linie je maximálně $2n$. Velikost haldy je pak také $2n$, takže pak operace určitě zvládneme v čase $\mathcal{O}(\log n)$. Jelikož diagram je lineárně velký tak i jeho struktura je lineárně velká. Operace se strukturami nás stojí nejvíce $\mathcal{O}(\log n)$. Takže místní i kružnicové události zvládneme v čase $\mathcal{O}(\log n)$ na jednu na konstantním počtu struktur. Halda má velikost $2n$, takže maximálně provedeme $\mathcal{O}(2n \log n)$ operací. Celý algoritmus potřebuje na inicializaci maximálně $\mathcal{O}(n \log n)$ (i kdybychom ji dělali neefektivně) a $\mathcal{O}(2n \log n)$ výpočet.

Pokud tedy shrneme všechny naše odhady, pak časová složitost algoritmu je $\mathcal{O}(n \log n)$ a prostorová $\mathcal{O}(n)$.

10. Převody problémů *(zapsali Martin Chytil, Vladimír Kudelas)*

Na této přednášce se budeme zabývat rozhodovacími problémy a jejich obtížností. Za jednoduché budeme trochu zjednodušeně považovat ty problémy, na něž známe algoritmus pracující v polynomiálním čase.

Definice: *Rozhodovací problém* je takový problém, jehož výstupem je vždy ANO, nebo NE. [Formálně bychom se na něj mohli dívat jako na množinu L vstupů, na které je odpověď ANO, a místo $L(x) = \text{ANO}$ psát prostě $x \in L$.]

Příklad: Je dán bipartitní graf G a $k \in \mathbb{N}$. Existuje v G párování, které obsahuje alespoň k hran?

To, co bychom ve většině případů chtěli, je samozřejmě nejen zjistit, zda takové párování existuje, ale také nějaké konkrétní najít. Všimněme si ale, že když umíme rozhodovat existenci párování v polynomiálním čase, můžeme ho polynomiálně rychle i najít:

Mějme černou skříňku (fungující v polynomiálním čase), která odpoví, zda daný graf má nebo nemá párování o k hranách. Odebereme z grafu libovolnou hranu a zeptáme se, jestli i tento nový graf má párovaní velikosti k. Když má, pak tato hrana nebyla pro existenci párování potřebná, a tak ji odstraníme. Když naopak nemá (hrana patří do každého párování požadované velikosti), tak si danou hranu poznamenáme a odebereme nejen ji a její vrcholy, ale také hrany, které do těchto vrcholů vedly. Toto je korektní krok, protože v původním grafu tyto vrcholy byly navzájem spárované, a tedy nemohou být spárované s žádnými jinými vrcholy. Na nový graf aplikujeme znovu tentýž postup. Výsledkem je množina hran, které patří do hledaného párování. Hran, a tedy i iterací našeho algoritmu, je polynomiálně mnoho a skříňka funguje v polynomiálním čase, takže celý algoritmus je polynomiální.

A jak náš rozhodovací problém řešit? Nejsnáze tak, že ho převedeme na jiný, který už vyřešit umíme. Tento postup jsme (právě u hledání párování) už použili v kapitole o Dinicově algoritmu. Vytvořili jsme vhodnou síť, pro kterou platilo, že v ní existuje tok velikosti k právě tehdy, když v původním grafu existuje párování velikosti k.

Takovéto převody mezi problémy můžeme definovat obecně:

Definice: Jsou-li A, B rozhodovací problémy, pak říkáme, že A lze *redukovat* (neboli převést) na B (píšeme $A \to B$) $\Leftrightarrow$ existuje funkce f spočitatelná v polynomiálním čase taková, že pro $\forall x : A(x) = B(f(x))$.

Všimněte si, že $A \to B$ také znamená, že problém B je alespoň tak těžký jako problém A (tím myslíme, že pokud lze B řešit v polynomiálním čase, lze tak řešit i A): Nechť problém B umíme řešit v čase $\mathcal{O}(b^k)$, kde b je délka jeho vstupu. Nechť dále funkce f převádějící A na B pracuje v čase $\mathcal{O}(a^\ell)$ pro vstup délky a. Spustíme-li tedy $B(f(x))$ na nějaký vstup x problému A, bude mít $f(x)$ délku $\mathcal{O}(a^\ell)$, takže $B(f(x))$ poběží v čase $\mathcal{O}(a^{k\ell})$, což je polynomiální v délce vstupu a.

Pozorování: Převoditelnost je:

- reflexivní (úlohu můžeme převést na tu stejnou identickým zobrazením): $A \to A$,
- tranzitivní: Je-li $A \to B$ funkcí f, $B \to C$ funkcí g, pak $A \to C$ složenou funkcí $g \circ f$ (složení dvou polynomiálních funkcí je zase polynomiální funkce, jak už jsme zpozorovali v předchozím odstavci).

Podívejme se nyní na převody mezi dalšími zajímavými problémy:

1. problém: SAT

Splnitelnost logických formulí, tj. dosazení *true* či *false* za proměnné v logické formuli tak, aby formule dala výsledek *true*.

Zaměříme se na speciální formu zadání formulí, *konjunktivní normální formu* (CNF).

$$(\ldots \vee \ldots \vee \ldots \vee \ldots) \,\&\, (\ldots \vee \ldots \vee \ldots \vee \ldots) \,\&\, \ldots$$

Vstup: Formule φ v konjunktivní normální formě.

Výstup: $\exists$ dosazení *true* a *false* za proměnné takové, že hodnota formule $\varphi(\ldots) =$ *true*.

Pro formuli platí následující podmínky:

- *formule* je zadána pomocí *klauzulí* oddělených &,
- každá *klauzule* je složená z *literálů* oddělených $\vee$,
- každý *literál* je buďto proměnná nebo její negace.

Ukážeme, že stačí vyřešit jednodušší problém 3-SAT.

2. problém: 3-SAT

Definice: 3-SAT je takový SAT, v němž každá klauzule obsahuje nejvýše tři literály.

Převod 3-SAT na SAT: Vstup není potřeba nijak upravovat, 3-SAT splňuje vlastnosti SATu, proto 3-SAT $\to$ SAT (3-SAT je alespoň tak těžký jako SAT)

Převod SAT na 3-SAT: Musíme formuli převést tak, abychom neporušili splnitelnost.

Trik pro dlouhé klauzule: Každou klauzuli

$$(\alpha \vee \beta), \text{ tž. } |\alpha| + |\beta| \geq 4$$

přepíšeme na:

$$(\alpha \vee x) \,\&\, (\beta \vee \neg x),$$

kde x je nová proměnná, kterou nastavíme tak, abychom neovlivnili splnitelnost formule.

Platí-li:

- $\alpha \Rightarrow x = 0$ (zajistí splnění druhé poloviny nové formule),
- $\beta \Rightarrow x = 1$ (zajistí splnění první poloviny nové formule),
- $\alpha, \beta / \neg\alpha, \neg\beta \Rightarrow x = 0/1$ (je nám to jedno, celkové řešení nám to neovlivní).

Tento trik opakujeme tak dlouho, dokud je to třeba.

Nabízí se otázka, proč můžeme proměnnou x nastavit, jak se nám zlíbí. Vysvětlení je prosté, proměnná x nám původní formuli nijak neovlivní, protože se v ní nevyskytuje, proto ji můžeme nastavit tak, jak chceme.

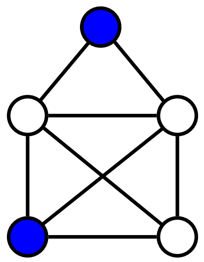
Poznámka: U 3-SAT lze vynutit právě tři literály, pro krátké klauzule použijeme následující trik:

$$(\alpha) \rightarrow (\alpha \vee x) \;\&\; (\alpha \vee \neg x).$$

3. problém: Hledání nezávislé množiny v grafu

Existuje nezávislá množina vrcholů z G velikosti alespoň k?

Definice: *Nezávislá množina* (NzMna) budeme říkat každé množině vrcholů grafu takové, že mezi nimi nevede žádná hrana.



Příklad nezávislé množiny

Vstup: Neorientovaný graf G, $k \in \mathbb{N}$.

Výstup: $\exists A \subseteq V(G)$, $|A| \geq k$: $\forall u, v \in A \Rightarrow uv \notin E(G)$?

Poznámka: Každý graf má minimálně jednu nezávislou množinu, a tou je prázdná množina. Proto je potřeba zadat i minimální velikost hledané množiny.

Ukážeme, jak na tento probém převést 3-SAT.

Převod: Z každé klauzule vybereme jeden literál tak, abychom v různých klauzulích nevybírali konfliktně, tj. x a $\neg x$.

Příklad: $(x \vee y \vee z) \;\&\; (x \vee \neg y \vee \neg z) \;\&\; (\neg x \vee \neg y \vee p)$.

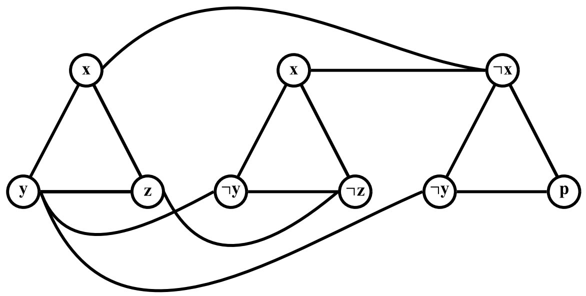
Pro každou klauzuli sestrojíme graf (trojúhelník) a přidáme „konfliktní“ hrany, tj. x a $\neg x$.

Princip je takový, že z každé klauzule si vybereme literál, který danou klauzuli splní, a to tak, aby literály, které si vybereme, nekolidovaly. Kolize ošetříme hranami mezi proměnnými a jejich negacemi.

Existuje nezávislá množina velikosti rovné počtu klauzulí? Pokud ano, tak dostaneme seznam proměnných, pomocí kterých splníme danou formuli.

Převod NzMna na SAT: Máme proměnné $v_1, \ldots, v_n$ pro vrcholy.

Nyní ukážeme, jak převést problém hledání nezávislé množiny, na SAT.



Ukázka převodu 3-SAT na nezávislou množinu

- Pořídíme si proměnné $v_1, \ldots, v_n$ odpovídající vrcholům grafu. Proměnná v_i bude indikovat, zda se i-tý vrchol vyskytuje v nezávislé množině.
- Pro každou hranu $ij \in E(G)$ přidáme klauzuli $(\neg v_i \vee \neg v_j)$. Tyto klauzule nám ohlídají, že vybraná množina je vskutku nezávislá.
- Ještě potřebujeme zkontrolovat, že je množina dostatečně velká, takže si její prvky očíslujeme čísly od 1 do k. Očíslování popíšeme maticí proměnných x_{ij}, přičemž x_{ij} bude pravdivá právě tehdy, když v pořadí i-tý prvek nezávislé množiny je vrchol v_j.
- Přidáme tedy klauzuje, které nám řeknou, že vybrané do nezávislé množiny jsou právě ty vrcholy, které jsou touto maticí očíslované: $\forall i, j,\; x_{ij} \Rightarrow v_j$.
- Ještě potřebujeme zajistit, aby byla v každém řádku i sloupci nejvýše jedna jednička: $\forall j, i, i^{'}, i \neq i^{'} : x_{ij} \Rightarrow \neg x_{i^{'}j}$ a $\forall i, j, j^{'}, j \neq j^{'} : x_{ij} \Rightarrow \neg x_{ij^{'}}$.
- A nakonec si ohlídáme, aby v každém řádku byla alespoň jedna jednička, klauzulí $\forall i : x_{i1} \vee x_{i2} \vee \ldots \vee x_{in}$.

Příklad matice: Jako příklad použijeme nezávislou množinu z ukázky nezávislé množiny. Nechť jsou vrcholy grafu očíslované zleva a ze zhora. Hledáme nezávislou množinu velikosti 2. Matice pak bude vypadat následovně:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Vysvětlení: Jako první vrchol množiny bude vybrán vrchol v_1, proto v prvním řádku a v prvním sloupci bude 1. Jako druhý (k-tý) vrchol množiny bude vybrán vrchol v_4, proto na druhém (k-tém) řádku a ve čtvrtém sloupci bude 1. Na ostatních místech bude 0.

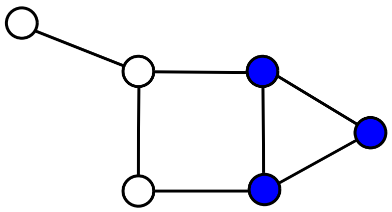
4. problém: Klika

Vstup: Graf $G, k \in N$.

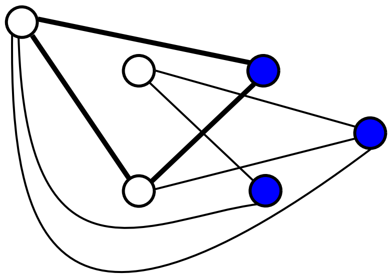
Výstup: $\exists$ úplný podgraf grafu G na k vrcholech?

Převod: Prohodíme v grafu G hrany a nehrany $\Rightarrow$ hledání nezávislé množiny.

Důvod: Pokud existuje úplný graf na k vrcholech, tak v „invertovaném“ grafu tyto vrcholy nejsou spojeny hranou, tj. tvoří nezávislou množinu.



Příklad kliky



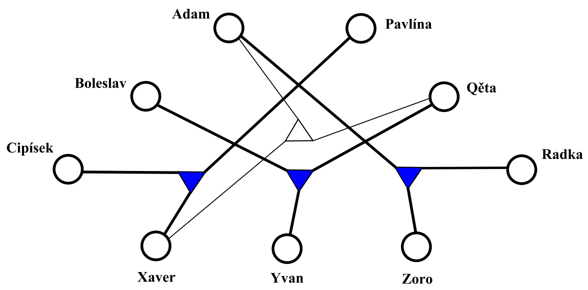
Prohození hran a nehran

5. problém: 3D párování (3D matching)

Vstup: Tři množiny, např. K (kluci), H (holky), Z (zvířátka) a množina kompatibilních trojic (těch, kteří se spolu snesou).

Výstup: Perfektní podmnožina trojic - tj. taková podmnožina trojic, která obsahuje všechna K, H a Z.

Ukážeme, jak tento problém převést na 3,3-SAT (ovšem to až na další přednášce).



Ukázka 3D párování

6. problém: 3,3-SAT

Definice: 3,3-SAT je speciální případ 3-SATu, kde každá proměnná se vyskytuje v maximálně třech literálech.

Převod 3-SAT na 3,3-SAT: Pokud se proměnná x vyskytuje v $k > 3$ literálech, tak nahradíme výskyty novými proměnnými $x_1, \ldots, x_k$ a přidáme klauzule:

$$(\neg x_1 \vee x_2)(\neg x_2 \vee x_3)(\neg x_3 \vee x_4) \ldots (\neg x_{k-1} \vee x_k)(\neg x_k \vee x_1),$$

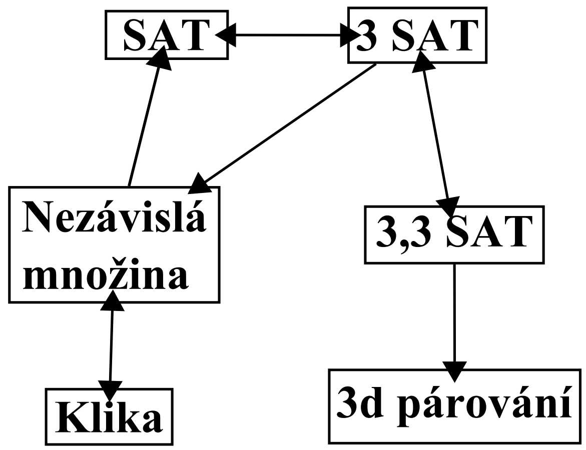
což odpovídá:

$$(x_1 \Rightarrow x_2)(x_2 \Rightarrow x_3)(x_3 \Rightarrow x_4) \ldots (x_{k-1} \Rightarrow x_k)(x_k \Rightarrow x_1).$$

Tímto zaručíme, že všechny nové proměnné budou mít stejnou hodnotu.

Mimochodem, můžeme rovnou zařídit, že každý literál se vyskytuje nejvíce dvakrát (tedy že každá proměnná se vyskytuje alespoň jednou pozitivně a alespoň jednou negativně). Pokud by se nějaká proměnná nějaká proměnná objevila ve třech stejných literálech, můžeme na ni také použít náš trik a nahradit ji třemi proměnnými. V nových klauzulích se pak bude vyskytovat jak pozitivně, tak negativně.

Závěr: Obrázek ukazuje problémy, jimiž jsme se dnes zabývali, a vztahy mezi těmito problémy.



Převody mezi problémy

11. NP-úplné problémy *(zapsali F. Kačmarik, R. Krivák, D. Remiš)*

Dosud jsme zkoumali problémy, které se nás ptaly na to, jestli něco existuje. Například jsme dostali formuli a problém splnitelnosti se nás ptal, zda existuje ohodnocení proměnných takové, že formule platí. Nebo v případe nezávislých množin jsme dostali graf a číslo k a ptali jsme se, jestli v grafu existuje nezávislá množina, která obsahuje alespoň k vrcholů. Tyto otázky měly společné to, že když nám někdo zadal nějaký objekt, uměli jsme efektivně říci, zda je to objekt, který hledáme. Například pokud dostaneme ohodnocení proměnných logické formule, stačí jen dosadit a spočítat, že formule je *true* nebo *false*. Zjistit, že nějaký objekt je ten, který hledáme, umíme efektivně. Těžké na tom je takový objekt najít. Což vede k definici obecných vyhledávacích problémů, kterým se říká třída problémů NP. Definujeme si ji pořádně, ale nejdříve začneme trošičku jednodušší třídou.

Definice: P je *třída rozhodovacích problémů*, které jsou řešitelné v polynomiálním čase. Jinak řečeno, problém $L \in P \Leftrightarrow \exists$ polynom f a $\exists$ algoritmus A takový, že $\forall x : L(x) = A(x)$ a $A(x)$ doběhne v čase $\mathcal{O}(f(x))$.

Třída P odpovídá tomu, o čem jsme se shodli, že umíme efektivně řešit. Nadefinujme tedy třídu NP:

Definice: NP je *třída rozhodovacích problémů* takových, že $L \in$ NP právě tehdy, když $\exists$ problém $K \in$ P a $\exists$ polynom g takový, že pro $\forall x$ platí $L(x) = 1 \Leftrightarrow \exists$ nápověda $y : |y| \leq g(|x|)$ a současně $K(x, y) = 1$.

Pozorování: Splnitelnost logických formulí je v NP. Stačí si totiž nechat napovědět, jak ohodnotit jednotlivé proměnné a pak ověřit, jestli je formule splněna. Nápověda je polynomiálně velká (dokonce lineárně), splnění zkontrolujeme také v lineárním čase. Odpovíme tedy ano právě tehdy, existuje-li nápověda, která nás přesvědčí, tedy pokud je formule splnitelná.

Pozorování: Třída P leží uvnitř NP. V podstatě říkáme, že když máme problém, který umíme řešit v polynomiálním čase bez nápovědy, tak to zvládneme v polynomiálním čase i s nápovědou.

Nasbírali jsme problémy, které jsou v NP, ale nevíme, jestli jsou v P. Brzy ukážeme, že to jsou v jistém smyslu nejtěžší problémy v NP. Nadefinujme si:

Definice: Problém L je NP-*těžký* právě tehdy, když je na něj převoditelný každý problém z NP. (Viz definici převodů z minulé přednášky)

Také platí, že pokud umíme řešit nějaký NP-těžký problém v polynomiálním čase, pak umíme vyřešit v polynomiálním čase vše v NP, a tedy P = NP. (To už také víme z minulé přednášky.)

My se budeme zabývat problémy, které jsou NP-těžké a samotné jsou v NP. Takovým problémům se říká NP-úplné.

Definice: Problém L je NP-*úplný* právě tehdy, když L je NP-těžký a $L \in$ NP.

NP-úplné problémy jsou tedy ve své podstatě nejtěžší problémy, které leží v NP. Kdybychom uměli vyřešit nějaký NP-úplný problém v polynomiálním čase, pak všechno v NP je řešitelné v polynomiálním čase. Bohužel to, jestli nějaký NP-úplný problém lze řešit v polynomiálním čase, se neví. Otázka, jestli P = NP, je asi nejznámější otevřený problém v celé teoretické informatice.

Kde ale nějaký NP-úplný problém vzít? K tomu se nám bude velice hodit následující věta:

Věta (Cookova): SAT je NP-úplný.

Důkaz je značně technický, přibližně ho naznačíme později. Přímým důsledkem je, že cokoli v NP je převoditelné na SAT. K dokazování NP-úplnosti dalších problémů použijeme následující větičku:

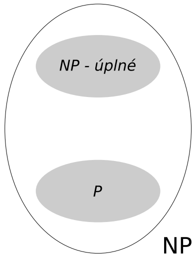
Větička: Pokud problém L je NP-úplný a L se dá převést na $M \in$ NP ($L \to M$), pak M je také NP-úplný.

Důkaz: Tuto větičku stačí dokázat pro NP-těžkost, NP-úplnost plyne okamžitě z toho, že problémy jsou NP-těžké a leží v NP (podle předpokladu).

Víme, že L se dá převést na M nějakou funkcí f. Jelikož L je NP-úplný, pak pro každý problém $Q \in$ NP existuje nějaká funkce g, která převede Q na L. Stačí tedy složit funkci f s funkcí g, čímž získáme funkci pracující opět v polynomiálním čase, která převede Q na M. Každý problém z NP se tedy dá převést na problém M. ♡

Důsledek: Cokoliv, na co jsme uměli převést SAT, je také NP-úplné. Například nezávislá množina, různé varianty SATu, klika v grafu …

Jak taková třída NP vypadá? Představme si všechny problémy třídy NP, jakoby seřazené zhora nadolu podle obtížnosti problémů, kde porovnání dvou problémů určuje převoditelnost (viz obrázek).



Struktura třídy NP

Obecně mohou nastat dvě situace. Protože nevíme, jestli P = NP. Jestli ano, pak všechno je jedna a ta samá třída. To by bylo v některých případech nepraktické, např. každá šifra by byla jednoduše rozluštitelná. Jestli ne, NP-úplné problémy určitě neleží v P, takže P a NP-úplné problémy jsou dvě disjunktní části NP. Také se dá dokázat (to dělat nebudeme, ale je dobré to vědět), že ještě něco leží mezi nimi, tedy že existuje problém, který je v NP, není v P a není NP-úplný.

Katalog NP-úplných problémů

Ukážeme si několik základních NP-úplných problémů. O některých jsme to dokázali na minulé přednášce, o dalších si to dokážeme nyní, zbylým se na zoubek podíváme na cvičeních.

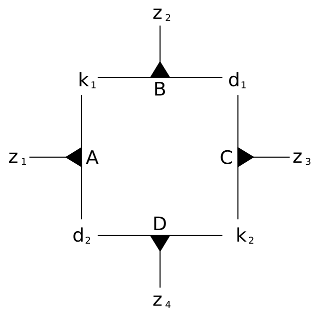
- *logické:*
  - SAT (splnitelnost logických formulí v CNF)
  - 3-SAT (každá klauzule obsahuje max. 3 literály)
  - 3,3-SAT (a navíc každá proměnná se vyskytuje nejvýše třikrát)
  - SAT pro obecné formule (nejen CNF)
  - Obvodový SAT (není to formule, ale obvod)
- *grafové:*
  - Nezávislá množina (množina alespoň k vrcholů taková, že žádné dva nejsou propojeny hranou)
  - Klika (úplný podgraf na k vrcholech)

- 3D párování (tři množiny se zadanými trojicemi, najít takovou množinu disjunktních trojic, ve které jsou všechny prvky)
- Barvení grafu (obarvit vrcholy k barvami tak, aby vrcholy stejné barvy nebyly nikdy spojeny hranou; NP-úplné už pro $k = 3$)
- Hamiltonovská cesta (cesta obsahující všechny vrcholy [právě jednou])
- Hamiltonovská kružnice (kružnice, která navštíví všechny vrcholy [právě jednou])

- *číselné:*
  - Batoh (nejjednodušší verze: dána množina čísel, zjistit, zda existuje podmnožina se zadaným součtem)
  - Loupežníci (rozdělit množinu na dvě podmnožiny se stejným součtem)
  - $Ax = b$ (soustava celočíslených lineárních rovnic; x_i mohou být pouze 0 nebo 1; NP-úplné i pokud $A_{ij} \in \{0, 1\}$ a $b_i \in \{0, 1\}$)
  - Celočíselné lineární programování (existuje vektor nezáporných celočísených x takový, že $Ax \leq b$)

Převoditelnost 3,3-SAT na 3D-párování

Když chceme ukázat, že na něco se dá převést SAT, potřebujeme obvykle dvě věci. Konstrukci, která bude simulovat proměnné, tedy něco, co nabývá dvou stavů *true*/*false*, a něco, co bude reprezentovat klauzule a umí zařídit, aby každá klauzule byla splněna alespoň jednou proměnnou. Jenom pro připomenutí, máme množinu kluků, děvčat, zvířátek a nějaké trojice, kdo se s kým snese, a chceme vybrat trojice tak, aby se v nich každý kluk, holka, zvířátko vyskytovalo právě jednou. Najdeme si takovouto konfiguraci:

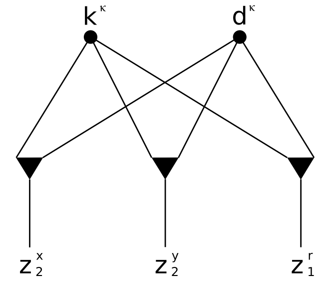


4 zvířátka, 2 kluci, 2 dívky a takové 4 trojice, které označíme A, B, C, D. Ještě předpokládáme, že zvířátka se mohou účastnit nějakých jiných trojic, ale tito čtyři lidé se vyskytují pouze v těchto čtyřech trojicích a nikde jinde. Všimneme si, že existují právě dvě možnosti, jak tento obrázek spárovat. Abychom spárovali kluka k_1, tak si musíme vybrat A nebo B. Když si vybereme A, k_1 i d_2 už jsou spárovaní takže si nesmíme vybrat B ani D. Pak jediná možnost, jak spárovat d_1 a k_2 je C.

Jedna možnost je tedy vybrat si A a C a jelikož je obrázek symetrický, tak když vybereme místo A trojici B, dostaneme B a D. Vždy si tedy vybereme dvě protější trojice v obrázku.

Takovýto obrázek budeme používat k reprezentaci proměnných. Pro každou proměnnou si nakreslíme takový obrázek a to, že A bude spárované s C, bude odpovídat tomu, že $x = 1$, a spárování B a D odpovídá $x = 0$. Pokud jsme použili A a C, zvířata se sudými čísly, tj. z_2 a z_4, horní a dolní jsou nespárovaná a pokud jsme použili B a D, zvířátka z_1 a z_3 zůstala nespárovaná. Přes tyto nespárovaná zvířátka můžeme předávat informaci, jestli proměnná x má hodnotu *true* nebo *false* do dalších částí grafu.

Zbývá vymyslet, jak reprezentovat klauzule. Klauzule budou vypadat jako trojice literálů: $\kappa = (x \vee y \vee \neg r)$ Potřebujeme zajistit, aby x bylo nastavené na 1 nebo y bylo nastavené na 1 nebo r na 0.



Pro takovouto klauzuli si pořídíme dvojici kluk-dívka, kteří budou figurovat ve třech trojicích se třemi různými zvířátky, což mají být volná zvířátka z obrázků pro příslušné proměnné. A zařídíme to tak, aby každé zvířátko bylo použité maximálně v jedné takové trojici, což jde proto, že každý literál se vyskytuje maximálně dvakrát a pro každý literál máme dvě volná zvířátka, z čehož plyne, že zvířátek je dost pro všechny klauzule.

Ještě nám ale určitě zbude $2p - k$ zvířátek, kde p je počet proměnných, tak přidáme ještě $2p - k$ párů kluk-děvče, kteří milují všechna zvířátka, a ti vytvoří zbývající páry.

Pokud formule byla splnitelná, pak ze splňujícího ohodnocení můžeme vyrobit párování s naší konstrukcí. Obrázek pro každou proměnnou spárujeme podle ohodnocení, tj. proměnná je 0 nebo 1 a pro každou klauzuli si vybereme některou z proměnných, kterými je ta klauzule splněna. Funguje to také ale i opačně. Když nám někdo dá párovaní v naší konstrukci, pak z něho dokážeme vyrobit splňující ohodnocení dané formule. Podíváme se, v jakém stavu je proměnná, a to je všechno. Z toho, že jsou správně spárované klauzule, už okamžitě víme, že jsou všechny splněné.

Zbývá ověřit, že naše redukce funguje v polynomiálním čase. Pro každou klauzuli spotřebujeme konstantně mnoho času, $2p - k$ je také polynomiálně mnoho a když to sečteme, máme polynomiální čas vzhledem k velikosti vstupní formule. Tím je převod hotový a můžeme 3D-párování zařadit mezi NP-úplné problémy.

Náznak důkazu Cookovy věty

Abychom mohli budovat teorii NP-úplnosti, potřebujeme alespoň jeden problém, o kterém dokážeme, že je NP-úplný, z definice. Cookova věta říká o NP-úplnosti SAT-u, ale nám se to hodí dokázat o trošku jiném problému – *obvodovém SAT-u*.

Obvodový SAT je splnitelnost, která nepracuje s formulemi, ale s booleovskými obvody. Každá formule se dá přepsat do booleovského obvodu, který ji počítá, takže dává smysl zavést splnitelnost i pro obvody. Naše obvody budou mít nějaké vstupy a jenom jeden výstup. Budeme se ptát, jestli se vstupy tohoto obvodu dají nastavit tak, abychom na výstupu dostali *true*.

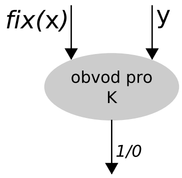
Nejprve dokážeme NP-úplnost *obvodového SAT-u* a pak ukážeme, že se dá převést na obyčejný SAT v CNF. Tím bude důkaz Cookovy věty hotový.

Věta: Obvodový SAT je NP-úplný.

Důkaz: Náznakem. Na základě zkušeností z Principů počítačů intuitivně chápeme počítače jako nějaké složité booleovské obvody, jejichž stav se mění v čase. Uvažme nějaký problém $L \in \mathrm{P}$ a polynomiální algoritmus, který ho řeší. Pro vstup velikosti n tedy doběhne v čase T polynomiálním v n a spotřebuje $\mathcal{O}(T)$ buněk paměti. Stačí nám tedy „počítač s pamětí velkou $\mathcal{O}(T)$“, což je nějaký booleovský obvod velikosti polynomiální v T, a tedy i v n. Vývoj v čase ošetříme tak, že sestrojíme T kopií tohoto obvodu, každá z nich bude odpovídat jednomu kroku výpočtu a bude propojena s „minulou“ a „budoucí“ kopií. Tím sestrojíme booleovský obvod, který bude řešit problém L pro vstupy velikosti n a bude polynomiálně velký vzhledem k n.

Ještě si dovolíme drobnou úpravu v definici třídy NP. Budeme chtít, aby nápověda byla měla pevnou velikost, závislou pouze na velikosti vstupu (tedy: $|y| = g(|x|)$). Proč je taková úprava BÚNO? Jistě si dovedete představit, že původní nápovědu doplníme na požadovanou délku nějakými „mezerami“, které program ignoruje. (Tedy upravíme program tak, aby mu nevadilo, že dostane na konci nápovědy nějak kódované mezery.)

Máme tedy nějaký problém L z NP a chceme dokázat, že L se dá převést na obvodový SAT. Když nám někdo předloží nějaký vstup x (chápeme jako vektor $(x_1, x_2, \ldots, x_n)$), spočítáme velikost nápovědy $g(|x|)$. Víme, že kontrolní algoritmus K (který kontroluje, zda nápověda je správně) je v P. Využijeme intuice o obvodech, abychom získali obvod, který pro konkrétní velikost vstupu x počítá to, co kontrolní algoritmus K. Na vstupu tohoto obvodu bude x (vstup problému L) a nápověda y. Na výstupu nám řekne, jestli je nápověda správná. Velikost vstupu tohoto obvodu bude tedy $|x| + g(|x|)$, což je polynom.



V tomto obvodu zafixujeme vstup x (na místa vstupu dosadíme konkrétní hodnoty z x). Tím získáme obvod, jehož vstup je jen y a ptáme se, zda za y můžeme dosadit nějaké hodnoty tak, aby na výstupu bylo *true*. Jinými slovy, ptáme se, zda je tento obvod splnitelný.

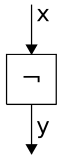
Pro libovolný problém z NP tak dokážeme sestrojit funkci, která pro každý vstup x v polynomiálním čase vytvoří obvod, který je splnitelný právě tehdy, když odpověď tohoto problému na vstup x má být *true*. Tedy libovolný problém z NP se dá v polynomiálním čase převést na obvodový SAT. ♡

Lemma: Obvodový SAT se dá převést na 3-SAT.

Důkaz: Budeme postupně budovat formuli v konjunktivní normální formě. Pro každé hradlo v obvodu zavedeme novou proměnnou popisující jeho výstup. Přidáme klauzule, které nám kontrolují, že toto hradlo máme ohodnocené konzistentně. Každý booleovský obvod se dá převést na ekvivalentní obvod, ve kterém se vyskytují jen hradla AND a NOT, takže stačí najít klauzule odpovídající těmto hradlům.

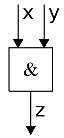
Převod hradla NOT: na vstupu hradla budeme mít nějakou proměnnou x (která přišla buďto přímo ze vstupu toho celého obvodu nebo je to proměnná, která vznikla na výstupu nějakého hradla) a na výstupu proměnnou y. Přidáme klauzule, které nám zaručí, že jedna proměnná bude negací té druhé:

$$(x \vee y),\quad (\neg x \vee \neg y).$$



Převod hradla AND: Hradlo má vstupy x, y a výstup z. Potřebujeme přidat klauzule, které nám popisují, jak se má hradlo AND chovat. Tyto vztahy přepíšeme do konjunktivní normální formy:

$$\left.\begin{array}{c} x \mathbin{\&} y \Rightarrow z \\ \neg x \Rightarrow \neg z \\ \neg y \Rightarrow \neg z \end{array}\right\} \quad \begin{array}{c} (z \vee \neg x \vee \neg y) \\ (\neg z \vee x) \\ (\neg z \vee y) \end{array}$$



Když chceme převádět obvodový SAT na 3-SAT, obvod nejdříve přeložíme na takový, ve kterém jsou jen hradla AND a NOT, a pak hradla tohoto obvodu přeložíme na klauzule. Formule vzniklá z takovýchto klauzulí je splnitelná právě tehdy, když je splnitelný daný obvod. Převod pracuje v polynomiálním čase. ♡

Poznámka: Když jsme zaváděli SAT, omezili jsme se jen na formule, které jsou v konjunktivní normální formě (CNF). Teď už víme, že splnitelnost obecné booleovské formule dokážeme převést na obvodovou splnitelnost a tu pak převést na 3-SAT. Opačný převod je samozřejmě triviální, takže obecný SAT je ve skutečnosti ekvivalentní s naším „standardním“ SATem pro CNF.

12. Aproximačné algoritmy *(F. Haško, J. Menda, M. Mareš)*

Na minulých prednáškach sme sa zaoberali rôzne ťažkými rozhodovacími problémami. Táto sa zaoberá postupmi ako sa v praxi vysporiadať s riešením týchto problémov.

Prvý spôsob: Špeciálny prípad

Často si vystačíme s vyriešením špeciálneho prípadu NP-úplného problému, ktorý leží v P. Napríklad, ak riešime grafovú úlohu, tak nám môže stačiť riešenie pre špeciálny druh grafov (strom, bipartitný graf, ...). Farbenie grafu je ľahké pre nejaký malý počet farieb. 2SAT, ako špeciálny prípad SAT-u, sa dá riešiť v lineárnom čase.

Problém: Maximálna nezávislá množina v strome (nie rozhodovacia)

Vstup: zakorenený strom T

Výstup: Maximálna (čo do počtu vrcholov) nezávislá množina vrcholov M

BUNV môžeme predpokladať, že v M sú všetky listy T. Ak by nejaký list l nebol v M, tak sa pozrieme na jeho otca:

- Ak otec nie je v M, tak vytvoríme novú nezávislú množinu M' obsahujúcu aj l (veľkosť nezávislej množiny stúpla o 1).
- Ak tam otec je, tak ho z M vyjmeme a namiesto neho vložíme l (veľkosť nezávislej množiny sa nezmenšila).

Tieto listy aj ich otcov z T odstránime a postup opakujeme. T sa môže rozpadnúť na les; potom tento postup aplikujeme na všetky stromy v lese.

Algoritmus:

1. Položíme M_1:={listy stromu T}.
2. Položíme M_2:={otcovia vrcholov z M_1}.
3. Vrátime $M_1 \cup$ MaxNz$(T \setminus (M_1 \cup M_2)$.

Poznámka: Toto dokážeme naprogramovať v $\mathcal{O}(n)$ (vrcholy máme vo fronte a prechádzame).

Problém: Batoh

Je daná množina n predmetov s hmotnosťami $h_1, \ldots, h_n$ a cenami $c_1, \ldots, c_n$ a batoh, ktorý unesie hmotnosť H. Nájdite takú podmnožinu predmetov, ktorých celková hmotnosť je najviac H a celková cena je maximálna možná.

Tento problém je zobecněním problému batohu z minulé přednášky dvěma směry: Jednak místo rozhodovacího problému řešíme optimalizační, jednak předměty mají ceny (předchozí verze odpovídala tomu, že ceny jsou rovny hmotnostem). Ukážeme si algoritmus pro řešení tohoto obecného problému, jehož časová složitost bude polynomiální v počtu předmětů n a součtu všech cen $C = \sum_i c_i$.

Použijeme dynamické programování. Představme si problém omezený na prvních k předmětů. Označme si $A_k(c)$ (kde $0 \leq c \leq C$) minimální hmotnost podmnožiny, jejíž cena je právě c. Tato A_k spočteme indukcí podle k: Pro $k = 0$ je určitě $A_0(0) = 0$,

$A_0(c) = \infty$ pro $c > 0$. Pokud již známe A_{k-1}, spočítáme A_k následovně: $A_k(c)$ odpovídá nějaké podmnožině předmětů z $1, \ldots, k$. V této podmnožině jsme buďto k-tý předmět nepoužili (a pak je $A_k(c) = A_{k-1}(c)$), nebo použili a tehdy bude $A_k(c) = A_{k-1}(c-c_k)+h_k$ (to samozřejmě jen pokud $c \geq c_k$). Z těchto dvou možností si vybereme tu, která dává množinu s menší hmotností. Tedy:

$$A_k(c) = \min(A_{k-1}(c), A_{k-1}(c - c_k) + h_k).$$

Tímto způsobem v čase $\mathcal{O}(C)$ spočteme jednu množinu, v čase $\mathcal{O}(nC)$ pak všechny.

Podle A_n snadno nalezneme maximální cenu množiny, která se vejde do batohu. To bude největší c^*, pro něž je $A_n(c^*) < \infty$. Jeho nalezení nás stojí čas $\mathcal{O}(C)$.

A jak zjistit, které předměty do nalezené množiny patří? Upravíme algoritmus, aby si pro každé $A_k(c)$ pamatoval $B_k(c)$, což bude index posledního předmětu, který jsme do příslušné množiny přidali. Pro nalezené c^* tedy bude $i = B_n(c^*)$ poslední předmět v nalezené množině, $i' = B_{i-1}(c^* - c_i)$ ten předposlední a tak dále. Takto v čase $\mathcal{O}(n)$ rekonstruujeme celou množinu od posledního prvku k prvnímu.

Ukázali jsme tedy algoritmus s časovou složitostí $\mathcal{O}(nC)$, který vyřeší problém batohu. Jeho složitost není polynomem ve velikosti vstupu (C muže být až exponenciálně velké vzhledem k velikosti vstupu), ale pouze ve velikosti čísel na vstupu. Takovým algoritmům se říká *pseudopolynomiální.*

Verze bez cen: Na verzi s cenami rovnými hmotnostem se dá použít i jiný algoritmus založený na dynamickém programování: počítáme množiny Z_k obsahující všechny hmotnosti menší než H, kterých nabývá nějaká podmnožina prvních k prvků. Přitom $Z_0 = \{0\}$, Z_k spočteme ze Z_{k-1} a ze Z_n vyčteme výsledek. Všechny tyto množiny mají nejvýše H prvků, takže celková časová složitost algoritmu je $\mathcal{O}(nH)$.

Druhý spôsob: Aproximácia

V predchádzajúcich problémoch sme sa zamerali na špeciálne prípady. Občas však také štastie nemáme a musíme vyriešiť celý NP-úplný problém. Môžeme si však pomôcť tým, že sa nebudeme snažiť vyriešiť ho optimálne – iba v nejakom pomere k optimálnosti (*aproximácia*), t.j. budeme vedieť, o koľko maximálne je naše riešenie horšie ako optimálne.

Problém: Obchodný cestujúci

Vstup: neorientovaný graf G, popisujúci nejaku krajinu a každá hrana je ohodnotená funkciou $w : E(G) \to \mathbb{R}_0^+$

Vystup: Hamiltonovská kružnica (všetky vrcholy grafu), a to tá najkratšia (podľa ohodnotenia).

Tento problém je hneď na prvý pohľad náročný – už problém, či existuje Hamiltonovská kružnica, je NP-úplný. BUNV nech graf G je úplný (doplnime zvyšné hrany ohodnotené $max(w)+1$ alebo viac, nie však nekonečnom, lebo by neplatila trojuholníková nerovnosť, ktorú neskôr budeme potrebovať). Vyriešme tento problém najprv za predpokladu, že vrcholy grafu spĺňajú trojuholníkovú nerovnosť, potom bez nej.

a) trojuholníková nerovnosť: $\forall x, y, z \in V : w(xz) \leq w(xy) + w(yz)$

Existuje pekný algoritmus, ktory nájde Hamiltonovsku kružnicu, čo je maximálne dvakrát tak veľká ako optimálna.

Nájdeme najmenšiu kostru a obchodnému cestujúcemu poradíme, nech ide po nej (stačí zakoreniť a prejsť do hĺbky). Problémom však je, že daný sled obsahuje každý vrchol viackrát a preto musíme nahradiť nepovolené vracania sa, t.j. pre každý vrchol nájsť ešte nenavštívený vrchol v našom slede a ísť priamo naň. Keďže platí trojuholníková nerovnosť, tak si týmito skratkami neuškodíme. Nech minimálna kostra má váhu T. Váha obídeného sledu tak bude $2T$. Skrátenia určite nezväčšujú, takže váha nájdene Hamiltonovskej kružnice bude nanajvýš $2T$.

Ak máme Hamiltonovskú kružnicu C a z nej vyškrtneme hranu, tak máme kostru grafu G s váhou najviac $w(C)$, teda to aspoň takú, aká je váha minimálnej kostry – T. To je optimálny prípad Hamiltonovskej kružnice. Ak to teda zložíme dohromady, algoritmus nám vráti Hamiltonovskú kružnicu s váhou najviac dvojnásobnou od optimálnej Hamiltonovskej kružnice. Takéto algoritmy sa nazývajú *2-aproximačné*, keďže riešenie je maximálne dvojnásobné od optimálneho.

b) bez trojuholníkovej nerovnosti:

Tu sa budeme naopak snažiť ukázať, že žiaden polynomiálny aproximačný algoritmus neexistuje.

Veta: Ak existuje polynomiálny $(1+\varepsilon)$-aproximačný algoritmus pre algoritmus obchodného cestujúceho bez trojuholníkovej nerovnosti pre ľubovoľné $\varepsilon > 0$, tak potom $P = NP$.

Důkaz: Ukážeme, že v tom prípade dokážeme v polynomiálnom čase nájsť Hamiltonovskú kružnicu.

Dostali sme graf G, v ktorom hľadáme Hamiltonovskú kružnicu. Doplníme G na uplný graf G' a váhy hrán G' nastavíme takto:

- $w(e) = 1$, ak $e \in E(G)$
- $w(e) = c \ll 1$, ak $e \notin E(G)$

Ak existuje Hamiltonovská kružnica v G' zložená iba z hrán, ktoré boli pôvodne v G, tak optimálné riešenie bude mať váhu n, inak bude určite minimálne $n - 1 + c$. Ak máme aproximačný algoritmus s pomerom $1 + \varepsilon$, musí byť

$$(1+\varepsilon).n < n - 1 + c$$
$$\varepsilon n + 1 < c$$

Ak by taký algoritmus existoval, tak na neho máme polynomiálny algoritmus na Hamiltonovsku kružnicu. Inak neexistuje ani pseudo-polynomialny algoritmus. ♡

Aproximační schéma pro problém batohu

POZOR: Verze algoritmu, kterou jsem říkal na přednášce, obsahovala jednu poměrně zásadní chybu, které jsem si nevšiml: Verze se zaokrouhlováním dolů mohla produkovat nepřípustná (příliš těžká) řešení, verze se zaokrouhlováním nahoru pro změnu

někdy spočítala řešení příliš daleká od optima. Algoritmus lze opravit (budemeli zvlášť zpracovávat lehké a těžké předměty), ale raději budeme místo hmotností kvantovat ceny. Tak dojdeme k následujícímu aproximačnímu algoritmu. –M.M.

Již víme, jak optimalizační verzi problému batohu vyřešit v čase $\mathcal{O}(nC)$, pokud jsou hmotnosti i ceny na vstupu přirozená čísla a C je součet všech cen. Jak si poradit, pokud je C obrovské? Kdybychom měli štěstí a všechny ceny byly dělitelné nějakým číslem p, mohli bychom je tímto číslem vydělit. Tím bychom dostali zadání s menšími čísly, jehož řešením by byla stejná množina předmětů jako u zadání původního.

Když nám štěstí přát nebude, můžeme přesto zkusit ceny vydělit a výsledky nějak zaokrouhlit. Řešení nové úlohy pak sice nebude přesně odpovídat optimálnímu řešení té původní, ale když nastavíme parametry správně, bude alespoň jeho dobrou aproximací.

Základní myšlenka:

Označíme si c_{max} maximum z cen c_i. Zvolíme si nějaké přirozené číslo M a zobrazíme interval cen $[0, c_{max}]$ na $[0, M]$. Jak jsme tím zkreslili výsledek? Všimněme si, že efekt je stejný, jako kdybychom jednotlivé ceny zaokrouhlili na násobky čísla c_{max}/M. Každé c_i jsme tím změnili o nejvýše c_{max}/M, celkovou cenu libovolné podmnožiny předmětů tedy nejvýše o $n \cdot c_{max}/M$. Teď si ještě všimněme, že pokud ze zadání odstraníme předměty, které se samy nevejdou do batohu, má optimální řešení původní úlohy cenu $OPT \geq c_{max}$, takže chyba v součtu je nejvýše $n \cdot OPT/M$. Má-li tato chyba být shora omezena $\varepsilon \cdot OPT$, musíme zvolit $M \geq n/\varepsilon$.

Algoritmus:

1. Odstraníme ze vstupu všechny předměty těžší než H.
2. Spočítáme $c_{max} = \max_i c_i$ a zvolíme $M = \lceil n/\varepsilon \rceil$.
3. Kvantujeme ceny: $\hat{c}_i = \lfloor c_i \cdot M/c_{max} \rfloor$.
4. Vyřešíme dynamickým programováním problém batohu pro upravené ceny $\hat{c}_1, \ldots, \hat{c}_n$ a původní hmotnosti i kapacitu batohu.
5. Vybereme stejné předměty, jaké použilo optimální řešení kvantovaného zadání.

Kroky 1–3 a 5 jistě zvládneme v čase $\mathcal{O}(n)$. Krok 4 řeší problém batohu se součtem cen $\hat{C} \leq nM \leq n^2/\varepsilon$, což stihne v čase $\mathcal{O}(n\hat{C}) = \mathcal{O}(n^3/\varepsilon)$. Zbývá dokázat, že výsledek našeho algoritmu má opravdu relativní chybu nejvýše ε.

Nejprve si rozmyslíme, jak dopadne optimální řešení OPT původního zadání, když ceny v něm použitých předmětů nakvantujeme (množinu indexů těchto předmětů si označíme Y):

$$\widehat{OPT} = \sum_{i \in Y} \hat{c}_i = \sum_i \left\lfloor c_i \cdot \frac{M}{c_{max}} \right\rfloor \geq \sum_i \left(c_i \cdot \frac{M}{c_{max}} - 1 \right) \geq$$
$$\geq \left(\sum_i c_i \cdot \frac{M}{c_{max}} \right) - n = OPT \cdot \frac{M}{c_{max}} - n.$$

Nyní naopak spočítejme, jak dopadne řešení Q nakvantovaného problému při přepočtu na původní ceny (to je výsledek našeho algoritmu):

$$ALG = \sum_{i \in Q} c_i \geq \sum_i \hat{c}_i \cdot \frac{c_{max}}{M} = \left(\sum_i \hat{c}_i\right) \cdot \frac{c_{max}}{M} \geq^* \widehat{OPT} \cdot \frac{c_{max}}{M}.$$

Nerovnost $\geq^*$ platí proto, že $\sum_{i \in Q} \hat{c}_i$ je optimální řešení kvantované úlohy, zatímco $\sum_{i \in Y} \hat{c}_i$ je nějaké další řešení téže úlohy, které nemůže být lepší. Teď už stačí složit obě nerovnosti a dosadit za M:

$$ALG \geq \left(\frac{OPT \cdot M}{c_{max}} - n\right) \cdot \frac{c_{max}}{M} \geq OPT - \frac{n \cdot c_{max}}{n/\varepsilon} \geq OPT - \varepsilon c_{max} \geq$$
$$\geq OPT - \varepsilon OPT = (1 - \varepsilon) \cdot OPT.$$

Algoritmus tedy vždy vydá řešení, které je nejvýše $(1 - \varepsilon)$-krát horší než optimum, a dokáže to pro libovolné ε v čase polynomiálním v n. Takovému algoritmu říkáme *polynomiální aproximační schéma* (jinak též PTAS⟨12⟩). V našem případě je dokonce složitost polynomiální i v závislosti na $1/\varepsilon$, takže schéma je *plně polynomiální* (řečené též FPTAS⟨13⟩).

13. Z teorie čísel *(zapsali L. Banáková, O. Hoferek, J. Břečka)*

Na této přednášce se budeme zabývat různými problémy okolo teorie čísel. Zopakujme si některé ze základních pojmů:

- $a \setminus b$ (a dělí b) $\Leftrightarrow \exists c : b = a \cdot c$.
- $\gcd(a, b)$ je označení největšího společného dělitele čísel a a b.
- $a \equiv_n b \Leftrightarrow n \perp (a - b)$ (nebo také $a \bmod n = b \bmod n$).
- $a \perp b$ (a a b jsou nesoudělná) $\Leftrightarrow \gcd(a, b) = 1$.

Dále si zopakujeme, jakou časovou složitost mají základní operace s čísly, které budeme potřebovat. Pro N-bitová čísla:

- $a + b$, $a - b \ldots \mathcal{O}(N)$
- $a * b$, a/b, $a \bmod b \ldots \mathcal{O}(N^2)$
- $\gcd(a, b) \ldots \mathcal{O}(N^3)$

Definice: *Komutativní (Abelovská) grupa* je čtveřice $(G, \cdot, 1, ^{-1})$, kde G je nosná množina prvků, $\cdot$ je binární operace $G^2 \to G$, 1 je prvkem G, $^{-1}$ je unární operace $G \to G$ a platí následující axiomy:

1. $\cdot$ je komutativní a asociativní
2. $\forall a \in G : a \cdot 1 = a$

⟨12⟩ Polynomial-Time Approximation Scheme
⟨13⟩ Fully Polynomial-Time Approximation Scheme

3. $\forall a \in G : a \cdot a^{-1} = 1$

Zkusme si pro následujících několik kandidátů určit, zda jsou komutativními grupami:

- $(\mathbb{Z}, +, 0, -x)$ (sčítání na množině celých čísel, nula a změna znaménka) je komutativní grupa.
- $(\mathbb{Z}_n, + \bmod n, 0, -x)$ ($\mathbb{Z}_n = \{0, \ldots, n-1\}$, sčítání modulo n) je komutativní grupa, navíc konečná.
- $(\mathbb{Q} - \{0\}, *, 1, 1/x)$ (násobení nad racionálními čísly) je komutativní grupa.
- $(\mathbb{Z}_n - \{0\}, * \bmod n, 1, ?)$ (násobení modulo n) nemusí být vždy grupa, protože se nám nepovede vždy najít inverzní prvek. (např. pro $n = 4$ neexistuje inverzní prvek pro dvojku).
- (permutace na $\{1, \ldots, n\}, \circ, \mathrm{id}, ^{-1}$) (permutace se skládáním a inverzní permutací) je grupa, ale není komutativní.

Definice: $(H, \cdot, 1, ^{-1})$ je *podgrupa* grupy $(G, \cdot, 1, ^{-1})$ právě tehdy, pokud $H \subseteq G$ a H je grupa.

Příkladem podgrupy může být $(2 * \mathbb{Z}_n, + \bmod n, 0, -x) \subseteq (\mathbb{Z}_n, + \bmod n, 0, -x)$ (grupa tvořená jen sudými čísly).

Věta (Lagrangeova): Pokud H je podgrupa G, pak počet prvků G je dělitelný počtem prvků H.

Definice: Umocňování prvků grupy definujeme takto: $x^0 = 1$, $x^{n+1} = x^n \cdot x$.

Definice: Grupa G je *cyklická* právě tehdy, pokud $\exists g \in G : g^0, g^1, \ldots = G$. Pak g je *generátor* grupy G.

- $(\mathbb{Z}, +, 0, -x)$ není cyklická.
- $(\mathbb{Z}_n, + \bmod n, 0, -x)$ je cyklická, $g = 1$.
- $(\mathbb{Q}, \ldots)$ není cyklická.

Definice: *Multiplikativní grupa modulo n* je grupa $\mathbb{Z}_n^* = (\{x \mid 1 \leq x < n$ a zároveň $\exists y : xy \equiv_n 1\}, * \bmod n, 1, ^{-1})$

Multiplikativní grupa $\mathbb{Z}_n^*$ obsahuje pouze invertibilní prvky. Snadno lze nahlédnout, že toto je opravdu grupa, pokud ověříme, že množina prvků je uzavřená na násobení:

$x_1, x_2 \in \mathbb{Z}_n^* \Rightarrow \exists y_1, y_2 \in \mathbb{Z}_n^* : x_1 y_1 \equiv_n 1, x_2 y_2 \equiv_n 1$

$x \equiv_n x_1 x_2, y \equiv_n y_1 y_2 \ldots xy \equiv_n x_1 x_2 y_1 y_2 \equiv_n 1 \cdot 1 = 1 \Rightarrow$ množina je uzavřená vzhledem k operaci $* \bmod n$.

Otázkou zůstává, jak najít tyto invertibilní prvky v $\mathbb{Z}_n$. Předpokládejme nejprve, že n je prvočíslo. Pomůžeme si větou, kterou dokážeme později.

Věta (Malá Fermatova): Pro každé prvočíslo n a každé číslo a, které je nesoudělné s n, platí: $a^{n-1} \equiv_n 1$.

Pokud je tedy n prvočíslo, tak z *Malé Fermatovy věty* vyplývá, že invertibilní jsou všechny prvky $1, \ldots, n - 1$. Snadno totiž nahlédneme, že $\forall a \in \mathbb{Z}_n$ platí $a^{-1} \equiv_n$

a^{n-2}, protože $a \cdot a^{-1} \equiv_n 1 \equiv_n a^{n-1} \equiv_n a \cdot a^{n-2}$. Jak to však bude s n, která nejsou prvočísla? Invertibilní jsou prvky $a \in \mathbb{Z}_n$, pro které existuje $x \in \mathbb{Z}_n$ tak, že $a \cdot x \equiv_n 1$. Jejich zkoumání začneme následující větou:

Věta: Rovnice $a \cdot x \equiv_n b$ má řešení pro $a, b, n \in \mathbb{Z} \iff g = \gcd(a, n) \perp b$. Navíc existuje algoritmus s časovou složitostí $\mathcal{O}(N^3)$, který řešení najde.

Důkaz: Rovnice $a \cdot x \equiv_n b$ je ekvivalentní s rovnicí $a \cdot x - n \cdot y = b$.

„$\Rightarrow$“: Dokážeme sporem. $g \setminus a \cdot x$ a $g \setminus n \cdot y \Rightarrow g \setminus a \cdot x - n \cdot y$. Zároveň však platí g nedělí b, což vede ke sporu.

„$\Leftarrow$“: Pokud $b = g$, pak lze úlohu hledání x, y řešení rovnice $a \cdot x - n \cdot y = g$ převést na Eukleidův algoritmus. V Eukleidově algoritmu se vstupem (x, y) jsou totiž všechny mezivýsledky i konečný výsledek lineární kombinace typu $\alpha \cdot x - \beta \cdot y$. Pokud $b = k \cdot g$, pak najdeme x_0, y_0 taková, že $a \cdot x_0 - n \cdot y_0 = g$. Pro $x = k \cdot x_0$ a $y = k \cdot y_0$ pak platí $a \cdot x - n \cdot y = b$. ♡

Z předchozí věty vyplývá, že invertibilní jsou právě taková $a \in \mathbb{Z}_n$, která jsou nesoudělná s n. Zbývá už jenom určit, jak vypadají prvky k nim inverzní.

Definice: *Eulerova funkce* $\varphi(n) = |\{x \mid 1 \leq x < n \text{ a zároveň } x \perp n\}|$.

Pozorování: *(vlastnosti Eulerovy funkce)*

- Je-li n prvočíslo, pak $\varphi(n) = n - 1$.
- $\varphi(n^k) = (n-1) \cdot n^{k-1}$.
- Pro $a \perp b$ platí $\varphi(a \cdot b) = \varphi(a) \cdot \varphi(b)$.
- $|\mathbb{Z}_n^*| = \varphi(n)$.

Věta (Eulerova): Pro každá dvě přirozená čísla n a a nesoudělná platí: $a^{\varphi(n)} \equiv_n 1$.

Důkaz: Uvažme posloupnost $a^0, a^1, a^2, \ldots$. V této posloupnosti se určitě vyskytuje jednička, protože možných hodnot a^m je nejvýše n. Proto existují i a j ($i < j$) taková, že $a^i \equiv_n a^j$, tedy $a^{j-i} \equiv_n 1$. Zvolme $m > 0$ nejmenší takové, že $a^m \equiv_n 1$. Čísla $a^0, a^1, \ldots, a^{m-1}$ tvoří podgrupu multiplikativní grupy $\mathbb{Z}_n^*$. Proto podle Lagrangeovy věty platí m dělí $\varphi(n)$, tedy $\varphi(n) = k \cdot m$ pro nějaké k. Snadno nahlédneme, že $a^{\varphi(n)} \equiv_n a^{k \cdot m} \equiv_n 1^k = 1$. ♡

Tato věta nám dokázala i dříve zmíněnou Malou Fermatovu větu a snadno díky ní nahlédneme, že pro $a \perp n$ platí $a^{-1} \equiv_n a^{\varphi(n)-1}$.

Testování prvočíselnosti

Nyní využijeme naše nově získané poznatky k ověřování prvočíselnosti. Tzv. *faktorizace*, neboli rozklad čísla na součet prvočísel, je určitě v *NP*, avšak zatím se neví, zda to je nebo není *NP*-úplný problém. Naproti tomu je již znám polynomiální algoritmus, který přesně ověří, zda je zadané číslo prvočíslem (s časovou složitostí $\mathcal{O}(N^{6,5})$). Tento algoritmus je však velmi komplikovaný a stejně tak odhad jeho časové složitosti. My se proto spokojíme s tzv. Monte Carlo testováním prvočísel. Pokud takto označený test prvočíselnosti odpoví, že číslo n zadané na vstupu je složené, je to pravda. Pokud odpoví, že se jedná o prvočíslo, mýlí se s pravděpodobností menší, než $\frac{1}{2}$, čili celková pravděpodobnost, že se mýlí je menší než $\frac{1}{4}$. Pokud takový test

opakujeme k-krát za sebou, je pravděpodobnost, že se test zmýlil $(\frac{1}{4})^k$, což je už pro $k = 100$ dostačující. Ukažme si tedy první z takových testů.

Algoritmus: *(Fermatův test)*

1. Zvolme náhodně $a \in \{2, \ldots, n-1\}$.
2. Pokud $\gcd(a, n) \neq 1$, je n složené.
3. Pokud $a^{n-1} \not\equiv_n 1$, je n složené (a a je *Fermatův svědek*).
4. Jinak je n prvočíslo.

Pozorování: Pokud odpoví Fermatův test, že zadané číslo je složené, nemýlí se.

Poznámka: Všimněme si, že druhý krok algoritmu je zbytečný. Pokud totiž není $a \perp n$, neboli existuje $g > 1$ společný dělitel n a a, pak g dělí i $a^{n-1} \bmod n$ nebo $a^{n-1} \bmod n = 0$ a třetí krok algoritmu prohlásí n za složené číslo.

Špatná zpráva pro tento algoritmus je, že existují tzv. *Carmichaelova čísla*, což jsou složená čísla, která Fermatova svědka nemají. Je jich sice „řídko“, ale zato nekonečně mnoho. Pokud se ale zrovna do Carmichaelova čísla nestrefíme, mají složená čísla Fermatových svědků dostatek.

Věta: Pokud n není Carmichaelovo číslo ani prvočíslo, pak existuje alespoň $\frac{\varphi(n)}{2}$ Fermatových svědků.

Důkaz: Zvolme $H = \{a \in \{1, \ldots, n-1\} \mid a^{n-1} \equiv_n 1, a \perp n\}$, tedy množinu všech čísel, která nejsou Fermatovými svědky. Všechna čísla v H jsou invertibilní a jejich součin je opět v H. H je tedy podgrupa multiplikativní grupy $\mathbb{Z}_n^*$. Podle Lagrangeovy věty tak existuje k tak, že $|\mathbb{Z}_n^*| = \varphi(n) = k \cdot |H|$. Navíc n není Carmichaelovo, takže platí $|\mathbb{Z}_n^*| \neq |H|$. Proto $k \geq 2$ a čísel, která nemají Fermatova svědka, je $|H| \leq \frac{\varphi(n)}{2}$. ♡

Vidíme tedy, že pokud n není Carmichaelovo číslo, pak je Fermatův test Monte Carlo test prvočíselnosti. Na závěr přednášky si uvedeme Monte Carlo test, který řeší problém Carmichaelových čísel (což není zrovna jednoduché):

Algoritmus: *(Rabin-Millerův test)*

1. Zvolme náhodně $a \in \{2, \ldots, n-1\}$.
2. Pokud $a^{n-1} \not\equiv_n 1$, je n složené (a a je *Fermatův svědek*).
3. Pro $i = 1, 2, \ldots$ dokud 2^i dělí $n - 1$:
4. Spočteme $t_i \equiv_n a^{(n-1)/2^i}$.
5. Pokud je $t_i \equiv_n -1$, je n prvočíslo.
6. Pokud je $t_i \not\equiv_n 1$, je n složené (a a je *Riemannův svědek*).
7. Jinak je n prvočíslo.