0. Ukázka použití maker

(zapsal Martin Mareš)

Předposlední kapitola

Je svačvečer. *Lysperní jezeleni* se vírně vrtáčejí v mokřavě.[1]

- Takhle.
- Vypadají.
- Odrážky.

Druhá kapitola

Věta: Pařez není strom.

Důkaz: Strom je speciálním případem lesa. Les je (podle definice) graf bez kružnic. Pařez obsahuje alespoň jednu kružnici. Proto pařez nemůže být les, a tedy ani strom. ♡

Algoritmus: (třídění posloupnosti $a_1, \ldots, a_n$ pomocí StupidSortu)

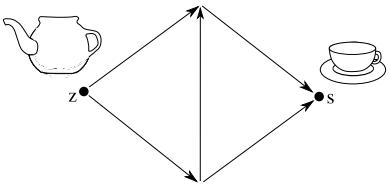
1. $\pi \leftarrow$ identická permutace na množině $\{1, \ldots, n\}$.
2. Opakuj:
3. Ověř, zda je posloupnost $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ uspořádána vzestupně. Pokud je, vrať ji jako výsledek.
4. Nahraď π jejím lexikografickým následníkem.

1. Toky v sítích

(zapsala Markéta Popelová)

První motivační úloha: Rozvod čajovodu do všech učeben.

Představme si, že by v budově fakulty na Malé Straně existoval čajovod, který by rozváděl čaj do každé učebny. Znázorněme si to orientovaným grafem, kde by jeden významný vrchol představoval čajovar a druhý učebnu, ve které sedíme. Hrany mezi vrcholy by představovaly větvící se trubky, které mají čaj rozvádět. Jak rozvést co nejefektivněji dostatek čaje do dané učebny?



Čajovod

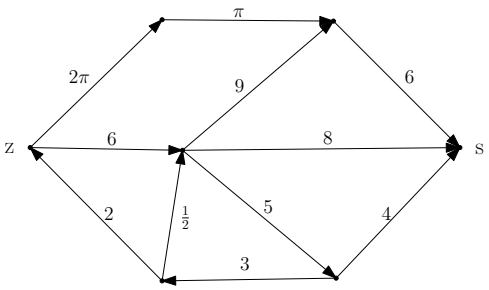
Druhá motivační úloha: Přenos dat.

[1] Viz Lewis Caroll: Jabberwocky.

Jiným příkladem může být počítačová síť na přenos dat, která se sestává z přenosových linek spojených pomocí routerů. Data se sice obvykle přenášejí po paketech, ale to můžeme při dnešních rychlostech přenosu zanedbat a považovat data za spojitá. Jak přenášet data mezi dvěma počítači v síti co nejrychleji?

Definice: *Síť* je uspořádaná pětice (V, E, z, s, c), kde platí:

- (V, E) je orientovaný graf.
- $c : E \to \mathbb{R}_0^+$ je *kapacita* hran.
- $z, s \in V$ jsou dva vrcholy grafu, kterým říkáme *zdroj* a *stok* (spotřebič).
- Graf je symetrický, tedy $\forall u, v \in V : uv \in E \Leftrightarrow vu \in E$ (tuto podmínku si můžeme zvolit bez újmy na obecnosti, neboť vždy můžeme do grafu přidat hranu, která v něm ještě nebyla, a dát jí nulovou kapacitu).



Příklad sítě. Čísla představují kapacity jednotlivých hran.

Definice: *Tok* je funkce $f : E \to \mathbb{R}_0^+$ taková, že platí:

1. Tok po každé hraně je omezen její kapacitou: $\forall e \in E : f(e) \leq c(e)$.
2. Kirchhoffův zákon:

$$\forall v \in V \setminus \{z, s\} : \sum_{u:uv \in E} f(uv) = \sum_{u:vu \in E} f(vu).$$

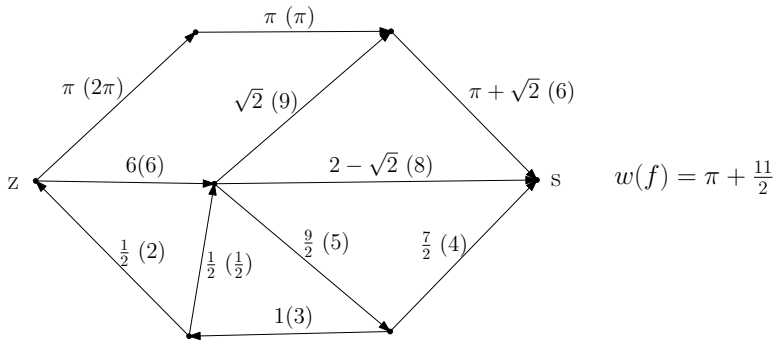
Neboli pro každý vrchol kromě zdroje a stoku platí, že to, co do něj přitéká, je stejně velké jako to, co z něj odtéká.

Poznámka: Pro zjednodušení zaveďme speciální značení:

- $f^+(v) = \sum_{u:uv \in E} f(uv)$ (to, co do vrcholu přitéká)
- $f^-(v) = \sum_{u:vu \in E} f(vu)$ (to, co z vrcholu odtéká)
- $f^\Delta(v) = f^+(v) - f^-(v)$ (rozdíl těchto hodnot)

Pak můžeme Krichhoffův zákon zapsat jednoduše jako:

$$\forall v \in V \setminus \{z, s\} : f^\Delta(v) = 0.$$



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

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

Pozorování: Nějaký tok vždy existuje. V libovolné síti můžeme vždy zvolit funkci nulovou (po žádné hraně nic nepoteče). Tato funkce splňuje podmínky toku, a tedy takovýto nulový tok je zcela korektní.

Definice: *Velikost toku* f je rozdíl součtu velikostí toku na hranách vedoucích do s a součtu velikostí toku na hranách vedoucích z s. Neboli od toho, co do stoku přitéká odečteme to, co ze stoku odtéká.

$$|f| := f^{\Delta}(s).$$

Cíl: Budeme chtít najít v zadané síti tok, jehož velikost je maximální.

Otázka: Má vůbec smysl mluvit o maximálním toku? Bude vždy existovat? Nevybíráme zde totiž z konečně mnoha případů a na první pohled není jasné, že supremum množiny všech toků bude zároveň i maximum této množiny.

Odpověď: Ano, pro každou síť existuje maximální tok. Toto poměrně překvapivé tvrzení můžeme nahlédnout za pomoci matematické analýzy. Nástin důkazu je takový, že množina toků je kompaktní a velikost toku je spojitá (dokonce lineární) funkce z množiny toků do $\mathbb{R}$. Proto nabývá velikost toku na množině všech toků svého maxima.

Poznámka: Pro naše případy předpokládejme, že kapacity jsou racionální. Poměrně nám to zjednoduší práci a příliš nám to neublíží, neboť práce s reálnými čísly je stejně pro informatika poměrně zapeklitá.

První řešení: Hledejme cestu P ze z do s takovou, že $\forall e \in P : f(e) < c(e)$ (po všech jejích hranách teče ostře méně, než jim dovolují jejich kapacity). Pak zjevně můžeme tok upravit tak, aby se jeho velikost zvětšila. Zvolme

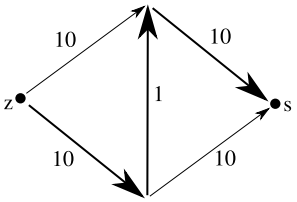
$$\varepsilon := \min_{e \in P}(c(e) - f(e)).$$

Nový tok f' pak definujme jako $f'(e) := f(e) + \epsilon$. Kapacity nepřekročíme (ϵ je největší možná hodnota, abychom tok zvětšili, ale nepřekročili kapacitu ani jedné

z hran cesty P) a Kirchhoffovy zákony zůstanou neporušeny, neboť zdroj a stok nezahrnují a každému jinému vrcholu na cestě P se přítok $f^+(v)$ i odtok $f^-(v)$ zvětší přesně o ϵ.

Otázka: Najdeme takto ovšem opravdu maximální tok?

Odpověď: Nemusíme. Např. na obrázku je vidět, že když najdeme nejdříve cestu přes hranu s kapacitou 1 (na obrázku tučně) a už hodnotu toku na této hraně nesnížíme, tak dosáhneme velikost toku nejvýše 19. Ale maximální tok této sítě má velikost 20.



Čísla představují kapacity jednotlivých hran.

Zde by ovšem situaci zachránilo, kdybychom poslali tok velikosti 1 proti směru prostřední hrany – to můžeme udělat třeba odečtením jedničky od toku po směru hrany.

Někdy je tedy potřeba poslat něco i v protisměru. Definujme si *rezervu hrany*. Ta nám říká, kolik můžeme daným směrem ještě poslat. Využijeme zde, že síť je symetrická.

Definice: *Rezerva hrany* uv je $r(uv) := c(uv) - f(uv) + f(vu)$.

Ukažme si nyní algoritmus, který rezervy využívá, a dokažme, že je konečný a že najde maximální tok každé racionální sítě.

Algoritmus (Fordův-Fulkersonův)

1. $f \leftarrow$ libovolný tok, např. všude nulový ($\forall e \in E : f(e) \leftarrow 0$).
2. Dokud $\exists P$ cesta ze z do s taková, že $\forall e \in P : r(e) > 0$, opakujeme:
3. $\quad \varepsilon \leftarrow \min\{r(e) \mid e \in P\}$.
4. $\quad$ Pro všechny hrany $uv \in P$:
5. $\quad\quad \delta \leftarrow \min\{f(vu), \varepsilon\}$
6. $\quad\quad f(vu) \leftarrow f(vu) - \delta$
7. $\quad\quad f(uv) \leftarrow f(uv) + \varepsilon - \delta$
8. Prohlásíme f za maximální tok.

Problém: Zastaví se Fordův-Fulkersonův algoritmus?

- Pro celočíselné kapacity se v každém kroku zvětší velikost toku alespoň o 1. Algoritmus se tedy zastaví po nejvíce tolika krocích, jako je nějaká horní závora pro velikost maximálního toku – např. součet kapacit všech hran vedoucích do stoku

$$\sum_{u:us \in E} c(us).$$

- Pro racionální kapacity využijeme jednoduchý trik – kapacity vynásobíme společným jmenovatelem a převedeme na původní případ. Uvědomme si, že algoritmus nikde kapacity hran ale ani toky na hranách nedělí, takže už zůstanou celočíselné. A tak jsme převedli racionální kapacity na celočíselné, pro které už víme, že se algoritmus zastaví.
- Na síti s iracionálními kapacitami se algoritmus chová mnohdy divoce, nemusí se zastavit ale dokonce ani konvergovat ke správnému výsledku. **K zamyšlení:** Zkuste vymyslet příklad takové sítě.

Otázka: Vydá algoritmus maximální tok?

Odpověď: Vydá. Abychom si to dokázali, zaveďme si řezy a použijme je jako certifikát maximality nalezeného toku.

Definice: *Řez* je uspořádaná dvojice množin vrcholů (A, B) taková, že A a B jsou disjunktní, pokrývají všechny vrcholy, A obsahuje zdroj a B obsahuje stok. Neboli $A \cap B = \emptyset$, $A \cup B = V$, $z \in A$, $s \in B$.

Definice: *Hrany řezu* $E(A, B) := E \cap A \times B$.

Poznámka: Řezy se dají definovat více způsoby, jedna z definic je, že řez je množina hran grafu takových, že po jejich odebrání se graf rozpadne na více komponent. Tuto definici splňuje i ta naše, ale ne naopak.

Definice: *Kapacita řezu* je

$$c(A, B) := \sum_{e \in E(A,B)} c(e).$$

Definice: *Tok přes řez* je

$$f(A, B) := \sum_{e \in E(A,B)} f(e) - \sum_{e \in E(B,A)} f(e).$$

Pozorování: Pro každý tok f a každý řez (A, B) platí, že $f(A, B) \leq c(A, B)$.

Důkaz:

$$f(A, B) = \sum_{e \in (A,B)} f(e) - \sum_{e \in E(B,A)} f(e) \leq \sum_{e \in E(A,B)} f(e) \leq \sum_{e \in E(A,B)} c(e) = c(A, B).$$

♡

Lemmátko: Pro každý tok f a pro každý řez (A, B) platí $f(A, B) = |f|$.

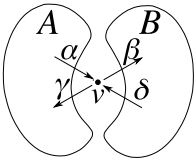
Důkaz: Indukcí a obrázkem.

Začneme s řezem $(V \setminus \{s\}, \{s\})$. Pro tento řez lemma platí z definice velikosti toku. Dále budu postupně přesouvat vrcholy z množiny B do množiny A. Libovolný řez může být takto vytvořen z toho triviálního.

Představme si, že máme již libovolný řez (A, B) a přesouváme vrchol v z A do B. Tedy $A' = A \setminus \{v\}$ a $B' = B \cup \{v\}$.

Uvědomme si, že všechny hrany jednoho typu (např. vedoucí z A do v) se chovají stejně, takže stačí uvažovat hrany pouze 4 typů (+ ostatní hrany (ty, které přesun neovlivní) označíme ε):

- α – hrany vedoucí z A do v
- β – hrany vedoucí z v do B
- γ – hrany vedoucí z v do A
- δ – hrany vedoucí z B do v



Přesun vrcholu v z A do B.

Před přesunem ($v \in A$) se $f(A, B)$ skládá z $\varepsilon + \beta - \delta$. Po přesunu ($v \in B$) se $f(A', B')$ skládá z $\varepsilon + \alpha - \gamma$. Rozdíl těchto hodnot je $\alpha + \delta - \beta - \gamma$.

Nicméně z Kirchhoffova zákonu o vrcholu v (což není ani zdroj ani stok) víme, že $\alpha + \delta - \beta - \gamma = f^{\Delta}(v) = 0$, neboť $\alpha + \delta$ je to, co do v přitéká, a $\beta + \gamma$ je to, co z v vytéká. Tedy tok přes řez před přesunem je stejně velký jako tok přes řez po přesunu. Pokud lemma platilo před přesunem, musí platit i po přesunu. ♡

Důsledek: Pro každý tok f a řez (A, B) platí, že $|f| = f(A, B) \leq c(A, B)$.

Pozorování: Pokud najdeme dvojici tok f a řez (A, B) takovou, že platí $|f| = c(A, B)$, pak tok f je maximální a řez (A, B) minimální.

Věta: Pokud se Fordův-Fulkersonův algoritmus zastaví, tak vydá maximální tok.

Důkaz:

Nechť se Fordův-Fulkersonův algoritmus zastaví. Definujme $A = \{v \in V; \exists$ cesta ze z do v jdoucí po hranách s $r > 0\}$ a $B = V \setminus A$.

Uvědomme si, že (A, B) je řez, neboť $z \in A$ (ze z do z existuje cesta délky 0) a $s \in B$ (kdyby $s \notin B$, tak by musela existovat cesta ze z do s s kladnou rezervou, tudíž by algoritmus neskončil, nýbrž tuto cestu vzal a stávající tok vylepšil).

Dále víme, že všechny hrany řezu mají nulovou rezervu, neboli $\forall uv \in E(A, B)$: $r(uv) = 0$ (kdyby měla hrana uv rezervu nenulovou, tedy kladnou, tak by vrchol v patřil do A). Proto po všech hranách řezu vedoucích z A do B teče tolik, kolik jsou kapacity těchto hran, a po hranách vedoucích z B do A neteče nic, tedy $f(uv) = c(uv)$ a $f(vu) = 0$. Máme řez (A, B) takový, že $f(A, B) = c(A, B)$. To znamená, že jsme našli maximální tok a minimální řez. ♡

Zformulujme si, co jsme zjistili a dokázali o algoritmu pánů Forda a Fulkersona.

Věta: Pro síť s racionálními kapacitami se Fordův-Fulkersonův algoritmus zastaví a vydá maximální tok a minimální řez.

Věta: (Fordova-Fulkersonova)

$$\min_{(A,B)\ \check{r}ez} c(A,B) = \max_{f\ tok} |f|.$$

Důkaz:

Již víme, že $\min_{(A,B)} c(A,B) \geq \max_f |f|$. Stačí tedy dokázat, že vždy existují tok f a řez (A,B) takové, že $c(A,B) = |f|$. Pro racionální kapacity nám Fordův-Fulkersonův algoritmus takový tok (maximální) a řez (minimální) vydá. Jak je to ale s reálnými kapacitami? Využijeme tvrzení, že maximální tok existuje vždy. Pak můžeme spustit Fordův-Fulkersonův algoritmus rovnou na tento maximální tok (místo nulového). Algoritmus se nutně ihned zastaví, neboť neexistuje cesta, která by měla alespoň jednu hranu s kladnou rezervou. A my víme, že pokud se algoritmus zastaví, tak vydá minimální řez. Proto i pro síť s reálnými kapacitami platí, že existuje maximální tok f a minimální řez (A,B) a $c(A,B) = |f|$. ♡

Věta: Síť s celočíselnými kapacitami má aspoň jeden z maximálních toků celočíselný a Fordův-Fulkersonův algoritmus takový tok najde.

Důkaz: Když dostane Fordův-Fulkersonův algoritmus celočíselnou síť, tak najde maximální tok a ten bude zase celočíselný (algoritmus nikde nedělí). ♡

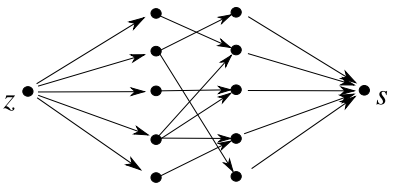
To, že umíme najít celočíselné řešení není úplně samozřejmé. (U jiných problémů takové štěstí mít nebudeme.) Ukažme si rovnou jednu aplikaci, která právě celočíselný tok využije.

Aplikace: Hledání maximálního párování v bipartitních grafech.

Definice: Množina hran $F \subseteq E$ se nazývá *párování*, jestliže žádné dvě hrany této množiny nemají společný ani jeden vrchol. Neboli $\forall e, f \in F : e \cap f = \emptyset$.

Definice: Párování je maximální, pokud obsahuje největší možný počet hran.

Mějme bipartitní graf $G = (V, E)$. V něm hledáme maximální párování. Sestrojme si síť takovou, že vezmeme vrcholy V grafu G a přidáme k nim dva speciální vrcholy z (zdroj) a s (stok) a ze zdroje přidáme hrany do všech vrcholů levé partity a ze všech vrcholů pravé partity povedeme hrany do stoku. Všechny kapacity nastavme na 1. Hrany bipartitního grafu zorientujme z levé partity do pravé. Nyní stačí jen na tuto síť spustit Fordův-Fulkersonův algoritmus (nebo libovolný jiný algoritmus, který najde maximální celočíselný tok) a až doběhne, tak prohlásit hrany s tokem 1 za maximální párování.



Hledání maximálního párování v bipartitním grafu.

Existuje totiž bijekce mezi párováním a celočíselnými toky při zachování velikosti. Z každého toku na výše zmíněném grafu (viz obrázek) lze sestrojit párování o stejné velikosti (velikost toku zde odpovídá počtu hran bipartitního grafu, po kterých poteče 1) a naopak. Důležité je si uvědomit, že definice toku (omezení toku kapacitou a Kirchhoffovy zákony) nám zaručují, že hrany s nenulovým tokem (tedy jedničkovým) budou tvořit párování (nestane se, že by dvě hrany začínaly nebo končily ve stejném vrcholu, neboť by se nutně porušila jedna ze dvou podmínek definice toku). Potom i maximální tok bude odpovídat maximálnímu párování a naopak.

V bipartitním grafu najdeme maximální párování v čase $\mathcal{O}(n \cdot (m+n))$. Fordův-Fulkersonův algoritmus stráví jednou iterací čas $\mathcal{O}(m+n)$ (za prohledání do šířky) a při jednotkových kapacitách bude iterací nejvýše n, protože každou se tok zvětší alespoň o 1 a všechny toky jsou omezené řezem kolem zdroje, který má kapacitu nejvýše n. Výsledná časová složtost hledání maximálního párování bude tedy $\mathcal{O}(n \cdot (m+n))$.

2. Dinicův algoritmus

(zapsala Markéta Popelová)

Na minulé přednášce jsme si ukázali Fordův-Fulkersonův algoritmus. Tento algoritmus hledal maximální tok tak, že začal s tokem nulovým a postupně ho zvětšoval. Pro každé zvětšení potřeboval v síti najít cestu, na které mají všechny hrany kladnou rezervu (po takovéto cestě můžeme poslat více, než po ní aktuálně teče). Ukázali jsme, že pokud takováto cesta existuje, jde tok vylepšit (zvětšit). Zároveň pokud tok jde vylepšit, pak takováto cesta existuje. Dokázali jsme, že pro racionální kapacity je algoritmus konečný a najde maximální tok.

Fordův-Fulkersonův algoritmus má ovšem značné nevýhody. Funguje pouze pro racionální kapacity a je poměrně pomalý. Nyní si ukážeme jiný algoritmus, který nevylepšuje tok pomocí cest, ale pomocí toků... Budeme k tomu potřebovat síť rezerv.

Definice: *Síť rezerv* k toku f v síti $S = (V, E, z, s, c)$ je síť $R = (S, f) = (V, E, z, s, r)$, kde $r(e)$ je rezerva hrany e v toku f.

Konvence: Pro hranu e značí $\overleftarrow{e}$ hranu opačnou. Např. pokud $e = uv$, tak $\overleftarrow{e} = vu$.

Je důležité si uvědomit, že síť rezerv je závislá jak na původní síti S, tak na nějakém toku f v síti S. Síť rezerv R se pak od sítě S liší pouze kapacitami – síť R má jako kapacitu hrany rezervu hrany v původní síti. Pro připomenutí: rezervu hrany e v síti $S = (V, E, z, s, c)$ s tokem f jsme si definovali jako $r(e) = c(e) - f(e) + f(\overleftarrow{e})$.

Než si ukážeme samotný algoritmus, dokážeme si následující lemma.

Lemma: Pro každý tok f v síti S a pro každý tok g v síti $R = (S, f)$ lze v čase $\mathcal{O}(m)$ nalézt tok f' v síti S takový, že $|f'| = |f| + |g|$.

Důkaz:

Důkaz rozdělíme do tří kroků. V prvním kroku si ukážeme, jak budeme tok f' v síti S konstruovat. V druhém kroku dokážeme, že takto zkonstruované f' je oprav-

du tok. A nakonec ukážeme, že splňuje požadovanou vlastnost, tedy že jeho velikost je součet velikostí toků f a g.

1. konstrukce f'

Pro každou dvojici hran $e, \overleftarrow{e}$ určíme $f'(e)$ a $f'(\overleftarrow{e})$ následovně:

- Pokud $g(e) = g(\overleftarrow{e}) = 0$, pak nastavme:
  - $f'(e) := f(e)$,
  - $f'(\overleftarrow{e}) := f(\overleftarrow{e})$.
- Pokud $g(e) > 0$ a $g(\overleftarrow{e}) = 0$, pak položme:
  - $\varepsilon := \min(g(e), f(\overleftarrow{e}))$,
  - $f'(e) := f(e) + g(e) - \varepsilon$,
  - $f'(\overleftarrow{e}) := f(\overleftarrow{e}) - \varepsilon$.
- Případ $g(e) = 0$ a $g(\overleftarrow{e}) > 0$ vyřešíme obdobně.
- Pokud $g(e) > 0$ a $g(\overleftarrow{e}) > 0$, pak odečteme od toku g cirkulaci po cyklu tvořeném hranami e a $\overleftarrow{e}$:
  - $\delta := \min(g(e), g(\overleftarrow{e}))$,
  - $g'(e) := g(e) - \delta$,
  - $g'(\overleftarrow{e}) := g(\overleftarrow{e}) - \delta$.

  Tok g' nyní spadá pod některý z předchozích případů, které už umíme vyřešit.

2. f' je tok

1. Nejdříve ověřme první podmínku: $\forall e \in E : 0 \leq f(e) \leq c(e)$. Vezměme libovolnou hranu $e \in E$. Podle toho, co teče po hranách e a $\overleftarrow{e}$ v toku g, jsme rozdělili konstrukci toku na tři případy:
   1. Pokud po hranách e a $\overleftarrow{e}$ netekl žádný tok g, pak jsme nastavili $f'(e) := f(e)$ a $f'(\overleftarrow{e}) := f(\overleftarrow{e})$. Tedy pokud f dodržoval kapacity, tak pro f' musí platit to samé.
   2. Pokud po hraně e tekl tok g nenulový a po opačné nulový, tak jsme zvolili: $f'(e) := f(e) + g(e) - \varepsilon$. Víme, že jsme si ε vybrali tak, že $\varepsilon \leq g(e)$. Proto $f'(e) \geq 0$.
      Teď ověřme, že $f'(e) \leq c(e)$. V případě, že $\varepsilon = g(e)$, tak $f'(e) = f(e) \leq c(e)$. V opačném případě platí, že $\varepsilon = f(\overleftarrow{e})$. Pak ovšem
      $$f'(e) = f(e) + g(e) - f(\overleftarrow{e}) \leq$$
      $$\leq f(e) + [c(e) - f(e) + f(\overleftarrow{e})] - f(\overleftarrow{e}) = c(e).$$
      Využili jsme, že g je tok v síti rezerv, tedy $g(e) \leq c(e) - f(e) + f(\overleftarrow{e})$.
      Pro tok $f'(\overleftarrow{e})$ platí, že $\varepsilon \leq f(\overleftarrow{e})$. Proto $f'(\overleftarrow{e}) = f(\overleftarrow{e}) - \varepsilon \geq 0$. Zároveň $f'(\overleftarrow{e}) \leq f(\overleftarrow{e}) \leq c(\overleftarrow{e})$.

Tím jsme dokázali, že $f'(e)$ i $f'(\overleftarrow{e})$ dodržují kapacity.

3. V posledním případě tekl po obou hranách kladný tok g. Menší tok z $g(e)$ a $g(\overleftarrow{e})$ jsme vynulovali a od většího odečetli ten menší. Tok $g'(e)$ a $g'(\overleftarrow{e})$ tedy zůstal korektní a tok f' už konstruujeme podle předchozího případu.

2. Teď musíme ještě dokázat, že nový tok neporušuje Kirchhoffovy zákony:

$$\forall v \ \in V \setminus \{z, s\} : f'^{\Delta}(v) = 0.$$

Vezměme si libovolnou hranu $e = uv \in E$. Uvědomme si, že při přechodu z $f(e)$ na $f'(e)$ a z $f(\overleftarrow{e})$ na $f'(\overleftarrow{e})$ bylo:

- $f^{\Delta}(u)$ sníženo o $g(e)$
- $f^{\Delta}(v)$ zvýšeno o $g(e)$.

Sečteme-li úpravy na všech hranách, dostaneme:

$$f'^{\Delta}(v) = f^{\Delta}(v) + \sum_{u:uv\in E} g(uv) - \sum_{u:vu\in E} g(vu) =$$

$$= f^{\Delta}(v) + g^{+}(v) - g^{-}(v) = f^{\Delta}(v) + g^{\Delta}(v).$$

Jelikož f byl tok, tak $f^{\Delta}(v) = 0$ a jelikož g byl tok, tak $g^{\Delta}(v) = 0$. Proto $f'^{\Delta}(v) = f^{\Delta}(v) + g^{\Delta}(v) = 0$.

Tím jsme dokázali, že f' je tok v síti S.

3. $|f'| = |f| + |g|$

Použijme vztah pro součet přebytků z předchozího kroku:

$$|f'| = f'^{\Delta}(s) = f^{\Delta}(s) + g^{\Delta}(s) = |f| + |g|.$$

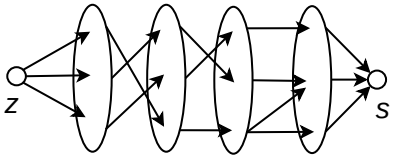
♡

Pro algoritmus budeme potřebovat vybírat kvalitní toky g v síti rezerv. Pokud se nám to bude dařit, bude se tok f' rychle zvětšovat, až bychom mohli dojít k maximálnímu toku. Nejlépe by se nám hodily co největší toky v síti rezerv. Kdybychom si dali za cíl najít vždy maximální tok v síti rezerv, výsledek by byl sice krásný (dostali bychom tak rovnou i maximální tok v původní síti), ale problém hledání maximálního toku bychom pouze přenesli na jinou síť. Naše požadavky na tento tok budou tedy takové, aby byl dostatečně velký, ale abychom během jeho hledání nestrávili moc času. Podívejme se, jak se s tímto problémem vyrovná *Dinicův algoritmus*. Nejdříve si ale zadefinujme několik pojmů.

Definice: Tok f je *blokující*, jestliže pro každou orientovanou cestu P ze z do s existuje hrana $e \in P$ taková, že $f(e) = c(e)$.

Definice: Síť je *vrstevnatá (pročištěná)*, když všechny vrcholy a hrany leží na nejkratších cestách ze z do s.

Dinicův algoritmus začíná s nulovým tokem. Potom vždy podle toku f sestrojí síť rezerv a v ní vymaže hrany s nulovou rezervou. Pokud v této promazané síti rezerv neexistuje cesta ze zdroje do stoku, tak skončí a prohlásí tok f za maximální. Jinak pročistí síť rezerv tak, aby se z ní stala vrstevnatá síť (rozdělí vrcholy do vrstev podle vzdálenosti od zdroje a odstraní přebytečné hrany). Ve vrstevnaté síti najde blokující tok, pomocí něhož zlepší tok f. Pak opět pokračuje sestrojením sítě rezerv na tomto vylepšeném toku f atd.



Pročištěná síť rozdělená do vrstev

Algoritmus (Dinicův)

1. $f \leftarrow$ nulový tok.
2. Sestrojíme síť rezerv R a smažeme $e : r(e) = 0$.
3. $l \leftarrow$ délka nejkratší cesty ze z do s v R.
4. Pokud $l = \infty$, zastavíme se a vrátíme f.
5. Pročistíme síť $R \rightarrow$ síť C.
6. $g \leftarrow$ blokující tok v C.
7. Zlepšíme tok f pomocí g.
8. GOTO 2.

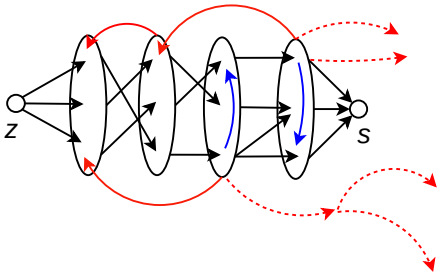
Pozorování: Pokud se algoritmus zastaví, vydá maximální tok.

Důkaz: Víme, že f je stále korektní tok (jediné, jak ho měníme je přičítání toku g, což je, jak jsme si dokázali, „neškodná operace“). Jakmile neexistuje cesta ze z do s v R, tak je f maximální tok, neboť v tuto dobu by se zastavil (a vydal maximální tok) i Fordův-Fulkersonův algoritmus, který je korektní. ♡

A teď ještě musíme ujasnit, jak budeme čistit síť rezerv a vybírat blokující tok g.

Algoritmus pročištění sítě rezerv

1. Rozdělíme vrcholy do vrstev podle vzdálenosti od z.
2. Odstraníme vrstvy za s (tedy vrcholy, které jsou od z vzdálenější než s), hrany do minulých vrstev a hrany uvnitř vrstev.
3. Odstraníme „slepé uličky“, tedy vrcholy s $\deg^{out}(v) = 0$, a to opakovaně pomocí fronty. (Nejdříve zařadíme do fronty všechny vrcholy s $\deg^{out}(v) = 0$. Pak dokud není fronta prázdná, vždy vybereme vrchol v z fronty, odstraníme v a všechny hrany uv. Pro každý takový vrchol u zkontrolujeme, zda se tím nesnížil výstupní stupeň vrcholu u na nulu ($\deg^{out}(u) = 0$). Pokud snížil, tak ho zařadíme do fronty.)



Nepročištěná síť. Obsahuje zpětné hrany, hrany uvnitř vrstvy a slepé uličky.

Hledání blokujícího toku začneme s tokem nulovým. Pak vezmeme vždy orientovanou cestu ze zdroje do stoku v síti C. V této cestě najdeme hranu s nejnižší hodnotou výrazu $r(e) - g(e)$ (neboli $c(e) - f(e)$ v původní síti). Tuto hodnotu označíme ε. Pak ke všem hranám na této cestě přičteme ε. Pokud tok g na nějaké hraně dosáhne kapacity hrany, což je zde $r(e)$, tak hranu vymažeme. Následně síť dočistíme, aby splňovala podmínky vrstevnaté sítě. A pokud ještě existuje nějaká orientovaná cesta ze zdroje do stoku, tak opět pokračujeme s touto cestou.

Algoritmus hledání blokujícího toku

1. $g \leftarrow$ nulový tok.
2. Dokud existuje orientovaná cesta P ze z do s v C, opakuj:
3. $\quad \varepsilon \leftarrow \min_{e \in P}(r(e) - g(e))$.
4. $\quad$ Pro $\forall e \in P : g(e) \leftarrow g(e) + \varepsilon$.
5. $\quad\quad$ Pokud $g(e) = r(e)$, smažeme e z C.
6. $\quad$ Dočistíme síť zase pomocí fronty.

Časová složitost Rozeberme si jednotlivé kroky algoritmu.

1. Inicializace toku f ... $\mathcal{O}(m)$.
2. Sestrojení sítě rezerv a smazání hran s nulovou rezervou ... $\mathcal{O}(m+n)$.
3. Najití nejkratší cesty (prohledáváním do šířky) ... $\mathcal{O}(m+n)$.
4. Zkontrolování délky nejkratší cesty ... $\mathcal{O}(1)$.
5. Pročištění sítě ... $\mathcal{O}(m+n)$.
    1. Rozdělení vrcholů do vrstev – provedlo již prohledávání do šířky ... $\mathcal{O}(1)$.
    2. Odstranění některých hran ... $\mathcal{O}(m+n)$.
    3. Odstranění „slepých uliček“ pomocí fronty – každou hranu odstraníme nejvýše jedenkrát, každý vrchol se dostane do fronty nejvýše jedenkrát ... $\mathcal{O}(m+n)$.
6. Najití blokujícího toku g ... $\mathcal{O}(m \cdot n)$.
    1. Inicializace toku g ... $\mathcal{O}(m)$.

2. Najití orientované cesty v pročištěné síti rezerv (stačí vzít libovolnou cestu ze zdroje, neboť každá z nich v této síti vede do stoku) ... $\mathcal{O}(n)$.
3. Výběr minima z výrazu $r(e) - g(e)$ přes všechny hrany cesty – ta může být dlouhá nejvýše n ... $\mathcal{O}(n)$.
4. Přepočítání všech hran cesty ... $\mathcal{O}(n)$.
5. Smazání hran cesty, jejichž tok $g(e)$ se zvýšil na hodnotu $r(e)$ ... $\mathcal{O}(n)$.
6. Dočišťování vyřešme zvlášť.

Vnitřní cyklus (kroky 2 až 6) provedeme nejvýše m krát, neboť při každém průchodu vymažeme alespoň jednu hranu (tak jsme si volili ε).
Čištění během celého hledání blokujícího toku g v pročištěné síti rezerv trvá dohromady $\mathcal{O}(m+n)$, neboť každou hranu a vrchol smažeme nejvýše jedenkrát.
Najití blokujícího toku bude tedy trvat $\mathcal{O}(m \cdot n + (m+n)) = \mathcal{O}(m \cdot n)$.

7. Zlepšení toku f pomocí toku g ... $\mathcal{O}(m)$.
8. Skok na 2. krok ... $\mathcal{O}(1)$.

Zbývá nám jen určit, kolikrát projdeme vnějším cyklem (fází). Dokážeme si lemma, že hodnota l vzroste mezi průchody vnějším cyklem (fázemi) alespoň o 1. Z toho plyne, že vnějším cyklem můžeme projít nejvýše n-krát, neboť cesta v síti na n vrcholech může být dlouhá nejvýše n.

Uvědomme si, že uvnitř vnějšího cyklu převládá člen $\mathcal{O}(m \cdot n)$, takže celková časová složitost bude $\mathcal{O}(n^2 \cdot m)$.

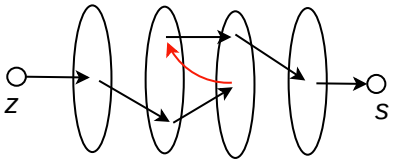
Lemma: Hodnota l (délka nejkratší cesty ze z do s v pročištěné síti) vzroste mezi fázemi alespoň o 1.

Důkaz:

Podíváme se na průběh jednoho průchodu vnějším cyklem. Délku aktuálně nejkratší cesty ze zdroje do stoku označme l. Všechny původní cesty délky l se během průchodu zaručeně nasytí, protože tok g je blokující. Musíme však dokázat, že nemohou vzniknout žádné nové cesty délky l nebo menší. V síti rezerv totiž mohou hrany nejen ubývat, ale i přibývat: pokud pošleme tok po hraně, po které ještě nic neteklo, tak v protisměru z dosud nulové rezervy vyrobíme nenulovou. Rozmysleme si tedy, jaké hrany mohou přibývat.

Hrany mohou přibývat jen tehdy, když jsme po opačné hraně něco poslali. Ale my něco posíláme po hranách pouze z vrstvy do té následující. Hrany tedy přibývají do minulé vrstvy.

Vznikem nových hran by proto mohly vzniknout nové cesty ze zdroje do stoku, které používají zpětné hrany. Jenže cesta ze zdroje do stoku, která použije zpětnou hranu, musí alespoň jednou skočit o vrstvu zpět a nikdy nemůže skočit o více než jednu vrstvu dopředu, a proto je její délka alespoň $l+2$. Pokud cesta novou zpětnou hranu nepoužije, má buď délku $> l$, což je v pořádku, nebo má délku $= l$, pak je zablokovaná.



Cesta užívající novou zpětnou hranu

Tím je lemma dokázáno. ♡

Všechna dokázaná tvrzení můžeme shrnout do následující věty:

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

Poznámka: Algoritmus se chová hezky na sítích s malými celočíselnými kapacitami, ale kupodivu i na různých jiných sítích. Často se používá, neboť se chová efektivně. A je mnoho způsobů, jak ho ještě vylepšovat, či odhadovat nižší složitost na speciálních sítích.

Poznámka: Algoritmus nevyžaduje racionální kapacity! Další z důvodů, proč maximální tok existuje i v síti s iracionálními kapacitami.

3. Goldbergův algoritmus

(zapsala Markéta Popelová)

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

Definice: Mějme síť $S = (V, E, z, s, c)$, tok f a libovolný vrchol v. Pak $f^\Delta(v)$ nazýváme *přebytek* ve vrcholu v a definujeme ho takto:

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

Přebytek ve vrcholu v je tedy součet všeho, co do vrcholu v přiteče, minus součet všeho, co z v odteče.

Definice: Dále pro libovolnou hranu $uv \in E$ definujeme její *rezervu* následovně:

$$r(uv) = c(uv) - f(uv) + f(vu).$$

Rezerva hrany značí, co ještě je možno po této hraně poslat.

Poznámka: Dále budeme označovat písmenem N počet vrcholů a M počet hran, tedy $N = |V|$ a $M = |E|$.

Goldbergův algoritmus na rozdíl od Dinicova algoritmu začíná s ohodnocením hran, které pravděpodobně není tokem (budeme ho nazývat *vlna*), a postupně ho zmenšuje až na korektní tok.

Definice: Funkce $f : E \to \mathbb{R}_0^+$ je *vlna* v síti (V, E, z, s, c) tehdy, když jsou splněny následující dvě podmínky:

1. $\forall e \in E : f(e) \leq c(e)$ (vlna na hraně nepřekročí kapacitu hrany)
2. $\forall v \in V \setminus \{z, s\} : f^\Delta(v) \geq 0$ (přebytek ve vrcholu je nezáporný).

Pozorování: Každý tok f je také vlna, ale opačně to obvykle platit nemusí.

Operace: *Převedení přebytku*

Algoritmus bude potřebovat převádět přebytky z vrcholu u na sousední vrchol v. Mějme hranu uv s kladnou rezervou $r(uv) > 0$ a kladným přebytkem ve vrcholu u: $f^\Delta(u) > 0$. Část přebytku budeme chtít poslat z vrcholu u do vrcholu v. Vezmeme $\delta := \min(f^\Delta(u), r(uv))$ a po hraně uv pošleme tok o velikosti δ. Výsledná situace bude vypadat následovně:

- $f'^\Delta(u) = f^\Delta(u) - \delta$.
- $f'^\Delta(v) = f^\Delta(v) + \delta$.
- $r'(uv) = r(uv) - \delta$.
- $r'(vu) = r(vu) + \delta$.

Kdybychom ovšem nepřidali žádnou jinou podmínku, náš algoritmus by se mohl krásně zacyklit (např. posílat přebytek z u do v a zase zpátky). Abychom se tomu vyhnuli, zavedeme *výšku vrcholu* $h : V \to \mathbb{N}$ a dovolíme převádět přebytek pouze z vyššího vrcholu u na nižší v: $h(u) > h(v)$.

Shrnutí: Podmínky pro převedení přebytku po hraně $uv \in E$:

1. Ve vrcholu u je nenulový přebytek: $f^\Delta(u) > 0$.
2. Vrchol u je výš než vrchol v: $h(u) > h(v)$.
3. Hrana uv má nenulovou rezervu: $r(uv) > 0$.

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 (Goldbergův)

1. $\forall v \in V : h(v) \leftarrow 0$ (všem vrcholům nastavíme počáteční výšku nula) a $h(z) \leftarrow N$ (zdroj zvedneme do výšky N).
2. $\forall e \in E : f(e) \leftarrow 0$ (po hranách nejdříve nenecháme protékat nic) a $\forall zu \in E : f(zu) \leftarrow c(zu)$ (ze zdroje pustíme maximální možnou vlnu).
3. Dokud $\exists u \in V \setminus \{z, s\} : f^\Delta(u) > 0$:
4. Pokud $\exists v \in V : uv \in E,\ r(uv) > 0$ a $h(u) > h(v)$, pak převedeme přebytek po hraně z u do v.
5. V opačném případě zvedneme u: $h(u) \leftarrow h(u) + 1$.
6. Vrátíme tok f jako výsledek.

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

Invariant A (základní):

1. Funkce f je v každém kroku algoritmu vlna.

2. $h(v)$ nikdy neklesá pro žádné v.
3. $h(z) = N$ a $h(s) = 0$ po celou dobu běhu algoritmu.

Důkaz: Indukcí dle počtu průchodů cyklem (3. – 5. krok algoritmu).

Na začátku je vše v pořádku (f je nulová funkce, přebytky všech vrcholů jsou nezáporné, tedy f je vlna, $h(z) = N$ a $h(s) = 0$). V průběhu se tyto hodnoty mění pouze při:

- Převedení po hraně uv: Po hraně uv se nepošle více než její rezerva. Přebytek u se sníží, ale nejméně na nulu. Přebytek v se zvýší. Tedy f zůstává vlnou. Výšky se nemění.
- Zvednutí vrcholu u: Mění pouze výšky – a to vrcholů různých od zdroje či stoku – a pouze se zvyšují. ♡

Invariant S (o Spádu): Neexistuje hrana $uv \in E : r(uv) > 0$ & $h(u) > h(v) + 1$ (s kladnou rezervou a spádem větším než jedna).

Důkaz: Indukcí dle běhu algoritmu.

Na začátku mají všechny hrany ze zdroje rezervu nulovou a všechny ostatní vedou mezi vrcholy s výškou 0. V průběhu by se tento invariant mohl pokazit pouze dvěma způsoby:

- Zvednutím vrcholu u, ze kterého vede hrana uv s kladnou rezervou a spádem 1. Tento případ nemůže nastat, neboť hranu zvedáme pouze tehdy, když neexistuje vrchol v takový, že hrana uv má kladnou rezervu a spád alespoň 1. Takový vrchol v našem případě existuje, proto se místo zvednutí vrcholu u pošle přebytek po hraně uv.
- Zvětšením rezervy hrany se spádem větším než 1. Toto také nemůže nastat, neboť rezervu bychom mohli zvětšit jedině tak, že bychom poslali něco v protisměru – a to nesmíme, jelikož bychom poslali přebytek z nižšího vrcholu na vyšší. ♡

Definice: Cestu P nazveme *nenasycenou*, pokud všechny její hrany mají kladnou rezervu. Neboli $\forall e \in P : r(e) > 0$.

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

Důkaz: Důkaz rozložme do dvou kroků. Nejdříve ukážeme, že f je tok, a pak jeho maximalitu.

1. Nechť se algoritmus zastavil. Pak nemohl existovat žádný vrchol v (kromě zdroje a stoku) s kladným přebytkem. Tedy $\forall v \in V \setminus \{z, s\} : f^{\Delta}(v) = 0$. (Víme již, že f je po celou dobu vlna, takže přebytek nemůže být nikdy záporný.) V tom případě splňuje f podmínky toku.
2. Pro spor předpokládejme, že tok f není maximální. Pak existuje nenasycená cesta ze zdroje do stoku. Vezměme si libovolnou takovou cestu. Zdroj je stále ve výšce N a spotřebič ve výšce 0 (viz invariant A). Tato cesta tedy překonává výšku N, ale může mít nejvýše $N-1$ hran. Proto existuje alespoň jedna hrana se spádem alespoň 2. Tato hrana tedy nemůže mít

kladnou rezervu (viz invariant S). Tato cesta proto nemůže být zlepšující, což je spor. Tím jsme dokázali, že f je nutně maximální tok. ♡

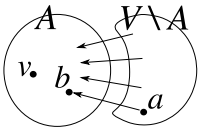
Lemma C (Cesta): Mějme vrchol $v \in V$. Pokud $f^\Delta(v) > 0$, pak existuje nenasycená cesta z vrcholu v do zdroje.

Důkaz: Pro vrchol $v \in V$ s $f^\Delta(v) > 0$ definujme množinu $A := \{u \in V : \exists \text{ nenasycená cesta z } v \text{ do } u\}$.

Sečtěme přebytky ve všech vrcholech množiny A. Přebytek každého vrcholu se spočítá jako součet toků do něj vstupujících minus součet toků z něj vystupujících. Všechny hrany, jejichž oba vrcholy leží v A, se jednou přičtou a jednou odečtou. Proto nás budou zajímat pouze hrany mezi A a $V \setminus A$.

$$\sum_{u\in A} f^\Delta(u) = \underbrace{\sum_{ab\in E\cap((V\setminus A)\times A)} f(ab)}_{=0} - \underbrace{\sum_{ab\in E\cap(A\times(V\setminus A))} f(ab)}_{\geq 0} \;\leq\; 0.$$

Ukažme si, proč je první svorka rovna nule. Mějme vrcholy $a \in V \setminus A$ a $b \in A$ takové, že $ab \in E$. O nich víme, že $r(ba) = 0$ (jinak by a patřilo do A) $\Rightarrow f(ba) = c(ba) \Rightarrow f(ab) = 0$. Proto do A nic nepřitéká.



Obrázek k důkazu lemmatu C

Proč je druhá svorka nezáporná, je zřejmé, neboť tok na hraně je vždy nezáporný a součet nezáporných čísel je nezáporné číslo.

Proto $\sum_{u\in A} f^\Delta(u) \leq 0$. Zároveň však v A je aspoň jeden vrchol s kladným přebytkem, totiž v, proto v A musí být také vrchol se záporným přebytkem – a jediný takový je zdroj. Tím je dokázáno, že $z \in A$, tedy že vede nenasycená cesta z vrcholu v do zdroje. ♡

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

Důkaz: Kdyby existoval vrchol v s výškou $h(v) > 2N$, tak by musel být někdy zvednut z výšky $2N$. Tehdy musel mít kladný přebytek $f^\Delta(v) > 0$ (jinak by nemohl být zvednut). Dle lemmatu C musela existovat nenasycená cesta z v do zdroje. Tato cesta měla spád alespoň N, ale mohla mít nejvýše $N - 1$ hran (jinak by to nebyla cesta v síti na N vrcholech). Tudíž musela na této cestě existovat hrana se spádem alespoň 2, což je spor s invariantem S (neboť všechny hrany této cesty mají z definice nenasycené cesty kladné rezervy). ♡

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. ♡

Teď nám ještě zbývá určit počet provedených převedení. Bude se nám hodit, když převedení rozdělíme na dva druhy:

Definice: Řekneme, že převedení je *nasycené*, pokud po převodu rezerva na hraně uv klesla na nulu, tedy $r(uv) = 0$. V opačném případě je *nenasycené*, a tehdy určitě klesne přebytek ve vrcholu u na nulu, tedy $f^\Delta(u) = 0$ (při nasyceném převedení se to ale může stát také).

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

Důkaz: Pro každou hranu uv spočítejme počet nasycených převedení (tedy takových převedení, že po nich klesne rezerva hrany na nulu). Abychom dvakrát nasyceně převedli přebytek (nebo jeho část) z vrcholu u do vrcholu v, tak jsme museli u mezitím alespoň dvakrát zvednout:

Po prvním nasyceném převedení z vrcholu u do vrcholu v se vynulovala rezerva hrany uv. Uvědomme si, že při této operaci muselo být u výše než v, a dokonce víme, že bylo výše přesně o 1 (viz lemma S). Po této hraně tedy nemůžeme už nic více převést. Aby došlo k druhému nasycenému převedení z u do v, musíme nejprve opět zvýšit rezervu hrany uv. Jediný způsob, jak toho lze dosáhnout, je převést část přebytku z v zpátky do u. K tomu se musí v dostat (alespoň o 1) výše než u. Po přelití bude rezerva uv opět kladná. A abychom provedli nasycené převedení znovu ve směru z u do v, musíme zase u dostat (alespoň o 1) výše než v. Proto musíme u alespoň o 2 zvednout – nejprve na úroveň v a pak ještě o 1 výše.

Ukázali jsme si tedy, že mezi každými dvěma nasycenými převedeními jsme vrchol u zvedli alespoň dvakrát. Nicméně libovolnou hranu můžeme zvednout nejvýše $2N$-krát (viz invariant V). Všech hran je M, tudíž počet všech nasycených převedení je nejvýše NM. ♡

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

Důkaz: Důkaz provedeme pomocí potenciálové metody – nadefinujme si následující funkci jako potenciál:

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

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

- Na počátku je $\Phi = 0$.
- Během celého algoritmu je $\Phi \geq 0$, neboť je součtem nezáporných členů.
- Zvednutí vrcholu zvýší Φ o jedničku (Aby byl vrchol zvednut, musel mít kladný přebytek $\Rightarrow$ vrchol do sumy již přispíval, teď jen přispěje číslem o 1 vyšším.). 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 Φ nejvýše o $2N$, protože buď po převodu hranou uv v u zůstal nějaký přebytek, takže se mohl potenciál zvýšit nejvýše

o $h(v) \leq 2N$, 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 (neboť se vynuluje přebytek ve 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 Φ 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 $v \neq z, s$ s kladným přebytkem. Neboli

$$P = \{v \in V \setminus \{z, s\} \mid f^{\Delta}(v) > 0\}.$$

Když měníme přebytek nějakého vrcholu, můžeme náš seznam v konstantním čase aktualizovat (např. tak, že si každý vrchol pamatuje pozici, na které v seznamu P je). V konstantním čase také umíme odpovědět, zda existuje nějaký vrchol s přebytkem.

Dále si pro každý vrchol $u \in V$ budeme pamatovat $L(u)$-seznam hran $uv \in E$ takových, které vedou dolů (mají spád alespoň 1) a kladnou rezervu. Neboli

$$L(u) = \{uv \in E \mid v \in V,\ r(uv) > 0,\ h(v) < h(u)\}.$$

Díky tomu můžeme přistupovat k patřičným sousedům u v čase $\mathcal{O}(1)$, stejně jako přidávat hrany do $L(u)$, resp. je mazat. Opět každá hrana si bude pamatovat pozici, na které se nachází v seznamu L.

Rozbor časové složitosti algoritmu:

1. Inicializace výšek ... $\mathcal{O}(N)$.
2. Inicializace vlny f ... $\mathcal{O}(M)$.
3. Výběr vrcholu u s kladným přebytkem – vezmeme první vrchol v P ... $\mathcal{O}(1)$.
4. Výběr vrcholu v, do kterého vede z u hrana s kladnou rezervou a který je níže než u – vezmeme první hranu z $L(u)$ ... $\mathcal{O}(1)$.

   Převedení přebytku: ... $\mathcal{O}(1)$.

   · Nasycené převedení ... $\mathcal{O}(1)$.

      · Rezerva hrany uv klesne na nulu ⇒ hrana uv vypadne z $L(u)$ ... $\mathcal{O}(1)$.
      · Přebytek vrcholu v se zvýší ⇒ pokud ještě nebyl v seznamu P, tak se tam přidá ... $\mathcal{O}(1)$.
      · Přebytek vrcholu u možná také klesne na nulu ⇒ pak by vrchol u vypadnul z P ... $\mathcal{O}(1)$.

   · Nenasycené převedení ... $\mathcal{O}(1)$.

- Rezerva hrany uv zůstane nezáporná $\Rightarrow$ hrana uv zůstane v $L(u)$ ... $\mathcal{O}(1)$.
- Vynuluje se přebytek vrcholu $u \Rightarrow$ vrchol u vypadne z P ... $\mathcal{O}(1)$.
- Přebytek vrcholu v se zvýší $\Rightarrow$ pokud ještě nebyl v seznamu P, tak se tam přidá ... $\mathcal{O}(1)$.

5. Zvednutí vrcholu u ... $\mathcal{O}(N)$.
   Musíme obejít všechny hrany do u a z u, kterých je nejvýše $2N - 2$, porovnat výšky a případně tyto hrany uv odebrat ze seznamu $L(v)$ resp. přidat do $L(u)$. Abychom pro odebrání hrany uv ze seznamu $L(v)$ nemuseli procházet celý seznam, budeme si $\forall u \in V$ pamatovat ještě $L^{-1}(u) :=$ seznam ukazatelů na hrany uv v seznamech $L(v)$.

Vidíme, že každé zvednutí je sice drahé, ale je jich zase poměrně málo. Naopak převádění přebytků je častá operace, takže je výhodné, že trvá konstantní čas.

Shrnutí:

- Všech zvednutí je $\mathcal{O}(N^2)$ (viz lemma Z), každé trvá $\mathcal{O}(N)$ ... $\mathcal{O}(N^3)$.
- Všech nasycených převedení je $\mathcal{O}(NM)$ (viz lemma S), každé trvá $\mathcal{O}(1)$ ... $\mathcal{O}(NM)$.
- Všech nenasycených převedení je $\mathcal{O}(N^2M)$ (viz lemma N), každé trvá $\mathcal{O}(1)$ ... $\mathcal{O}(N^2M)$.

Dohromady má tedy Goldbergův algoritmus časovou složitost $\mathcal{O}(N^2M)$. Vidíme, že už v tomto obecném případě to není horší než Dinicův algoritmus. Příště si ukážeme, že může mít i mnohem lepší. Nejdříve ale zformulujme všechna dokázaná tvrzení do následující věty:

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

Pozorování: Pokud bychom volili vždy nejvyšší z vrcholů s přebytkem, tak by se mohl algoritmus chovat lépe. Podívejme se na to pozorněji a vylepšený Goldebrgův algoritmus označme G'.

Algoritmus (Vylepšený Goldbergův algoritmus)

1. $\forall v \in V : h(v) \leftarrow 0$ (všem vrcholům nastavíme počáteční výšku nula) a $h(z) \leftarrow N$ (zdroj zvedneme do výšky N).
2. $\forall e \in E : f(e) \leftarrow 0$ (po hranách nejdříve nenecháme protékat nic) a $\forall zu \in E : f(zu) \leftarrow c(zu)$ (ze zdroje pustíme maximální možnou vlnu).
3. Dokud $\exists u \in V \setminus \{z, s\} : f^{\Delta}(u) > 0$:
4. Vybereme z vrcholů s přebytkem ten s nejvyšší výškou, označíme ho u.
5. Pokud $\exists v \in V : uv \in E,\ r(uv) > 0$ a $h(u) > h(v)$, pak převedeme přebytek po hraně z u do v.
6. V opačném případě zvedneme u: $h(u) \leftarrow h(u) + 1$.

7. Vrátíme tok f jako výsledek.

Rozmysleme si, o kolik bude vylepšený algoritmus G' lepší než ten původní. Ten původní měl časovou složitost $\mathcal{O}(N^2M)$ a převládal člen, který odpovídal nenasyceným převedením. Zkusme tedy právě počet nenasycených převedení odhadnout ve vylepšeném algoritmu o něco těsněji.

Lemma N' (Nenasycená převedení): Algoritmus G' provede $\mathcal{O}(N^3)$ nenasycených převedení.

Důkaz: Dokazovat budeme opět pomocí potenciálové metody. Zadefinujme si potenciál *nejvyšší hladinu s přebytkem*:

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

Rozdělíme běh algoritmu na *fáze*. Každá fáze končí tím, že se H změní. Jak se může změnit? Buď se H zvýší, což znamená, že nějaký vrchol s přebytkem v nejvyšší hladině byl o 1 zvednut, nebo se H sníží. My víme, že zvednutí je v celém algoritmu $\mathcal{O}(N^2)$. Zároveň si můžeme uvědomit, že H je nezáporný potenciál, kdy snížení i zvýšení ho změní o 1, tedy počet snížení bude stejný jako počet zvýšení, a proto obojího je $\mathcal{O}(N^2)$. Tudíž počet fází je také $\mathcal{O}(N^2)$.

Je důležité, že během jedné fáze provedeme nejvýše jedno nenasycené převedení z každého vrcholu. Po každém nenasyceném převedení po hraně uv se totiž vynuluje přebytek v u a aby se provedlo další nenasycené převedení z vrcholu u, muselo by nejdříve být co převádět. Muselo by tedy do u něco přitéci. My ale víme, že převádíme pouze shora dolů a u je v nejvyšší hladině (to zajistí právě to vylepšení algoritmu), tedy nejdříve by musel být nějaký jiný vrchol zvednut. Tím by se ale změnilo H a skončila by tato fáze.

Proto počet všech nenasycených převedení během jedné fáze je nejvýše N. A již jsme dokázali, že fází je $\mathcal{O}(N^2)$. Tedy počet všech nenasycených převedení je $\mathcal{O}(N^3)$. ♡

Tento odhad je hezký, ale stále není těsný a algoritmus se chová lépe. Dokažme si ještě jeden těsnější odhad na počet nenasycených převedení.

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

Poznámka: Tato časová složitost je výhodná například pro řídké grafy. Ty mají totiž poměrně malý počet hran.

Důkaz: Rozdělme si fáze na dva druhy: laciné a drahé podle toho, kolik se v nich provede nenasycených převedení. Zvolme si nějaké nezáporné K. Zatím nebudeme určovat jeho hodnotu. Uvidíme, že časová složitost algoritmu bude závislá na tomto parametru K. Proto jeho hodnotu zvolíme až později a to tak, aby byla složitost co nejnižší.

Laciné fáze budou ty, během nichž se provede nejvýše K nenasycených převedení. *Drahé fáze* budou ty ostatní, tedy takové, během nichž se provede více jak K nenasycených převedení.

Teď potřebujeme odhadnout, kolik nás budou stát oba typy fází. Začněme s těmi jednoduššími – s lacinými. Víme, že všech fází je $\mathcal{O}(N^2)$. Těch laciných bude tedy určitě také $O(N^2)$. Nenasycených převedení se během jedné laciné fáze provede nejvíce K. Tedy celkem se během laciných fází provede $\mathcal{O}(N^2K)$ nenasycených převedení.

Pro počet nenasycených převedení v drahých fázích si zaveďme nový potenciál definovaný následovně:

$$\Phi := \sum_{\substack{v \neq z,s \\ f^\Delta(v) \neq 0}} \frac{p(v)}{K},$$

kde $p(v)$ je počet takových vrcholů u, které nejsou výše než v. Neboli

$$p(v) = |\{u \in V \mid h(u) \leq h(v)\}|.$$

Tedy platí, že $p(v)$ je vždy nezáporné a nejvýše má hodnotu N. Dále víme, že Φ bude vždy nezáporné (neboť je to součet nezáporných členů) a nejvýše bude nabývat hodnoty $\frac{N^2}{K}$. Rozmysleme si, jak nám ovlivní tento potenciál naše tři operace:

- **Zvednutí**: Za každý zvednutý vrchol přibude nejvýše $\frac{N}{K}$ (tento vrchol může být nadzvednut nejvýše nad všechny ostatní vrcholy) a možná něco ubude (např. když vrchol vyzvedneme na úroveň k ostatním).
- **Nasycené převedení** po hraně uv: Může vynulovat přebytek ve vrcholu u, pak se Φ sníží. Může zvýšit přebytek ve v z nuly, pak se Φ zvýší. Ale nejvýše se zvýší o $\frac{N}{K}$, neboť do Φ přibude jen jeden sčítanec za vrchol v a ten přispěje nejvýše hodnotou $\frac{N}{K}$ (pod ním může být nejvíce N vrcholů).
- **Nenasycená převedení** po hraně uv v drahých fázích: Tato operace vynuluje přebytek v u, tedy Φ klesne alespoň o $\frac{p(u)}{K}$. Zároveň může zvýšit přebytek ve v z nuly, ale Φ stoupne nejvýše o $\frac{p(v)}{K}$. Celkem tedy Φ klesne alespoň o $\frac{p(u)-p(v)}{K}$.

Uvědomme si, že pokud převádíme po hraně uv, tak platí, že $h(u) = h(v) + 1$. Pak $p(u) - p(v)$ je přesně počet vrcholů na hladině H. Těch je alespoň tolik, kolik je nenasycených převedení během jedné fáze (to jsme dokázali již v lemmatu N'), a my jsme si zadefinovali, že v drahé fázi je počet nenasycených převedení alespoň K. Tedy $p(u) - p(v) > K$. Proto během jednoho nenasyceného převedení Φ klesne alespoň o $\frac{K}{K} = 1$. Nenasycená převedení potenciál nezvyšují.

Potenciál Φ se může zvětšit pouze při operacích zvednutí a nasycené převedení. Zvednutí se provede celkem $\mathcal{O}(N^2)$ a každé zvýší potenciál nejvýše o $\frac{N}{K}$. Nasycených převedení se provede celkem $\mathcal{O}(NM)$ a každé zvýší potenciál taktéž nejvýše o $\frac{N}{K}$. Celkem se tedy Φ zvýší nejvýše o

$$\frac{N}{K}\mathcal{O}(N^2) + \frac{N}{K}\mathcal{O}(NM) = \mathcal{O}\left(\frac{N^3}{K} + \frac{N^2M}{K}\right).$$

Teď využijeme toho, že Φ je nezáporný potenciál, tedy když každé nenasycené převdení v drahé fázi sníží Φ alespoň o 1, tak všech nenasycených převdení v drahých fázích je $\mathcal{O}(\frac{N^3}{K} + \frac{N^2M}{K})$. Už jsme ukázali, že nenasycených převední v laciných fázích je $\mathcal{O}(N^2K)$. Proto celkem všech nenasycených převedení je

$$\mathcal{O}\left(N^2K + \frac{N^3}{K} + \frac{N^2M}{K}\right) = \mathcal{O}\left(N^2K + \frac{N^2M}{K}\right)$$

(neboť pro souvislé grafy platí, že $M \geq N \Rightarrow N^2M \geq N^3$). A my chceme, aby jich bylo co nejméně. Tato funkce má minimum tehdy, když $N^2K = \frac{N^2M}{K}$, čili $K = \sqrt{M}$.

Proto všech nenasycených převedení je $\mathcal{O}(N^2\sqrt{M})$. ♡

4. Hradlové sítě

(zapsal: Petr Jankovský)

Výkon počítačů nelze zvyšovat donekonečna a přestože již pěkných pár let platí, že se jejich rychlost s časem exponenciálně zvětšuje, jednou určitě narazíme přinejmenším na fyzikální limity.

Když tedy nemůžeme zvyšovat rychlost jednoho procesoru, jak počítat rychleji? Řešením by mohlo být pořídit si procesorů víc. Už dnes na běžném PéCéčku máme k dispozici vícejádrové procesory, díky nimž můžeme využít pararelní počítání a úlohu řešit tak, že práci šikovně rozdělíme mezi procesory (či jádra) a zaměstnáme je při výpočtu všechny.

My se podíváme na abstraktní výpočetní model, který je ještě paralelnější. Techniky, které si ukážeme na tomto modelu, se však dají překvapivě využít i při reálném paralelizování na několika málo procesorech. Konec konců i proto, že vnitřní architektura procesoru se našemu modelu velmi podobá. Budeme se zabývat jednoduchým modelem paralelního počítače, totiž hradlovou sítí.

Hradlové sítě

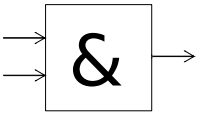
Definice: *Hradlo* je prvek, který umí vyhodnocovat nějakou funkci nad konečnou abecedou Σ.

Obecně se na hradlo díváme jako na funkci $f : \Sigma^k \to \Sigma$, která dostane k vstupů a vrátí jeden výstup, přičemž hodnoty, nad kterými pracuje, budou z nějaké konečné abecedy – tedy z nějaké konečné množiny symbolů Σ. Písmenku k zde říkáme *arita hradla*.

Příklad: Často studujeme hradla booleovská (pracující nad abecedou $\{0, 1\}$), která počítají logické funkce.

Z nich nejčastěji potkáme:

- nulární: to jsou konstanty (FALSE=0, TRUE=1),
- unární: např. negace (značíme $\neg$),
- binární: logický součin (AND, &), součet (OR, $\vee$), ...



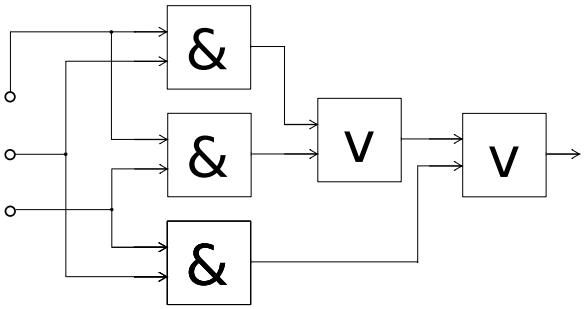
Binární hradlo provádící logickou operaci AND.

Hradla kreslíme třeba následovně:

Jednotlivá hradla můžeme navzájem určitým způsobem propojovat a vytvářet z nich *hradlové sítě*. Pokud používáme pouze booleovská hradla, říkáme takto vytvořeným sítím *booleovské obvody*. Pokud pracujeme s operacemi nad nějakou obecnější (ale konečnou) množinou symbolů (abecedou), nazývají se *kombinační obvody.*

Každá hradlová síť má nějaké vstupy, nějaké výstupy a uvnitř jsou propojovaná hradla. Každý vstup hradla je připojen buďto na některý ze vstupů sítě nebo na výstup jiného hradla. Výstupy hradel mohou být propojeny na vstupy dalších hradel (mohou se větvit), nebo na výstupy sítě. Přitom máme zakázáno vytvářet cykly.

Než si řekneme formální definici, podívejme se na obrázek.



Hradlová síť – třívstupová verze funkce *majorita*.

Obrazek znázorňuje hradlovou síť, která počítá, zda je alespoň na dvou ze vstupů jednička. Pojďme si ale *hradlovou síť* definovat formálně.

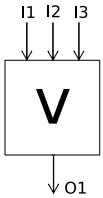
Definice: *Hradlová síť* je určena:

- *abecedou* Σ (to je nějaká konečná množina symbolů, obvykle $\Sigma = \{0, 1\}$);
- po dvou disjunktními konečnými množinami I (*vstupy*), O (*výstupy*) a H (*hradla*);
- acyklickým orientovaným multigrafem (V, E), kde $V = I \cup O \cup H$;
- zobrazením F, které každému hradlu $h \in H$ přiřadí nějakou funkci $F(h) : \Sigma^{a(h)} \to \Sigma$, což je funkce, kterou toto hradlo vykonává. 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.

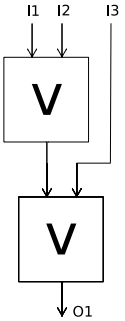
Přitom jsou splněny následující podmínky:

- $\forall i \in I : \deg^{in}(i) = 0$ (do vstupů nic nevede);
- $\forall o \in O : \deg^{in}(o) = 1$ & $\deg^{out}(o) = 0$ (z výstupů nic nevede a do každého vede právě jedna hrana);
- $\forall h \in H : \deg^{in}(v) = a(v)$ (do každého hradla vede tolik hran, kolik je jeho arita);
- $\forall h \in H \; \forall j : 1 \leq j \leq a(h)$ existuje právě jedena hrana e taková, že e končí v h a $z(e) = j$, (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, kterému bychom přiřadili funkci, která by naši úlohu rovnou vyřešila. Tento model však není ani realistický, ani pěkný. Proto přijměme omezení, že arity všech hradel budou omezeny nějakou pevnou konstantou k (ukáže se, že nám bude stačit dvojka a vystačíme si tedy pouze s nulárními, unárními a binárními hradly). Následující obrázky ukazují, jak hradla o více vstupech nahradit dvouvstupovými:



Trojvstupové hradlo OR.



Jeho nahrazení 2-vstupovými hradly.

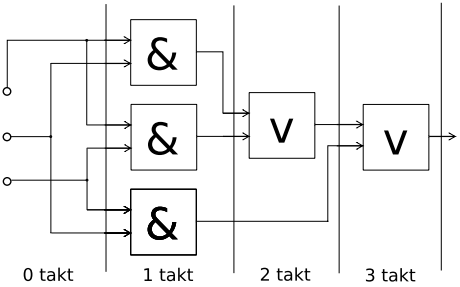
Nyní bychom ještě měli definovat, co taková hradlová síť vlastně počítá a jak její výpočet probíhá.

Definice: *Výpočet sítě* probíhá po *taktech.* V nultém taktu jsou definovány pouze hodnoty na vstupech sítě a na výstupech hradel arity 0. Můžeme si to představit tak, že na začátku nemá žádné hradlo definovánu výstupní hodnotu (až na již zmíněná hradla nulární). V každém dalším taktu pak vydají výstup hradla, která na konci minulého taktu měla definovány všechny hodnoty na vstupech. Jakmile budou po nějakém konečném počtu taktů definované i hodnoty všech výstupů, síť se zastaví a vydá výsledek.

Pozorování: Protože je síť acyklická, je jasné, že jakmile jednou nějaké hradlo vydá výstup, tak se tento výstup během dalšího výpočtu sítě již nezmění.

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

Definice: *i-tá vrstva* S_i obsahuje právě takové vrcholy v, pro které nejdelší cesta ze vstupů sítě do v má délku rovnou i.



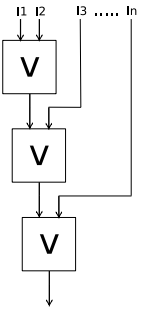
Výpočet hradlové sítě.

Pozorování: Všimněme si, že v i-tém taktu vydají hodnoty právě hradla z S_i.

Dává tedy smysl prohlásit za *časovou složitost* sítě počet jejích vrstev. Podobně *prostorovou složitost* definujeme jako počet hradel v síti.

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

První řešení: zapojíme hradla za sebe (sériově). Časová i prostorová složitost odpovídají $\Theta(n)$. Zde ovšem vůbec nevyužíváme toho, že by mohlo počítat více hradel současně.



Hradlová síť, která zjistí, zdali je na vstupu alespoň jedna jednička.

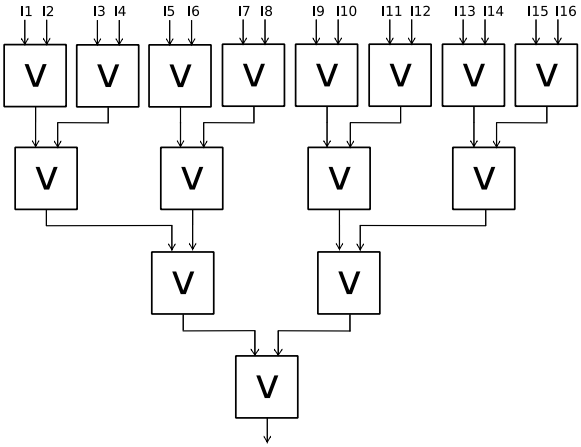
Druhé řešení: Hradla budeme spojovat do dvojic, pak výsledky z těchto dvojic opět do dvojic a tak dále. Díky paralelnímu zapojení dosáhneme časové složitosti $\Theta(\log n)$, prostorová složitost zůstane lineární.

5. Paralelní sčítání, bitonické třídění *(zapsal: Petr Jankovský)*

Minule jsme si zavedli paralelní výpočetní model, ve kterém si nyní něco naprogramujeme ...

Sčítání binárních čísel

Mějme dvě čísla x a y zapsané ve dvojkové soustavě. Jejich číslice označme $x_{n-1} \dots x_0$ a $y_{n-1} \dots y_0$, kde i-tý řád má váhu 2^i. Nyní budeme chtít tato čísla sečíst.



Chytřejší řešení stejného problému pro vstup velikosti 16.

K tomuto účelu se ihned nabízí použít starý dobrý „školní algoritmus“, který funguje ve dvojkové soustavě stejně dobře jako v desítkové. Zkrátka sčítáme čísla zprava doleva. Vždy sečteme příslušné číslice pod sebou a přičteme přenos z nižšího řádu. Tím dostaneme jednu číslici výsledku a přenos do vyššího řádu. Formálně bychom to mohli zapsat třeba takto:

$$z_i = x_i \oplus y_i \oplus c_i,$$

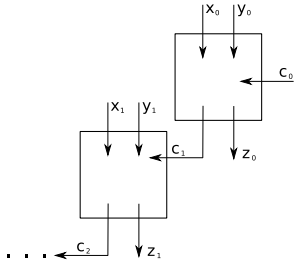
kde z_i je i-tá číslice součtu, $\oplus$ značí operaci XOR (součet modulo 2) a c_i je *přenos* z $(i-1)$-ního řádu do i-tého. Přenos přitom nastane tehdy, pokud se nám potkají dvě jedničky pod sebou, nebo když se vyskytne alespoň jedna jednička a k tomu přenos z nižšího řádu. Jinými slovy tehdy, když ze tří xorovaných číslic jsou alespoň dvě jedničky (pomocí obvodu pro majoritu z minulé přednášky lehce zkonstruujeme):

$$\begin{aligned} c_0 &= 0, \\ c_{i+1} &= (x_i \mathbin{\&} y_i) \vee (x_i \mathbin{\&} c_i) \vee (y_i \mathbin{\&} c_i). \end{aligned}$$

Takovéto sčítání sice perfektně funguje, nicméně je bohužel poměrně pomalé. Pokud bychom stavěli hradlovou síť podle tohoto předpisu, byla by složená z nějakých podsítí („krabiček“), které budou mít na vstupu x_i, y_i a c_i a jejich výstupem bude z_i a c_{i+1}.

Všimněme si, že každá krabička závisí na výstupu té předcházející. Jednotlivé krabičky tedy musí určitě ležet na různých hladinách. Celkově bychom museli použít $\Theta(n)$ hladin a jelikož je každá krabička konstantně velká, také $\Theta(n)$ hradel. To dává lineární časovou i prostorovou složitost, čili oproti sekvenčnímu algoritmu jsme si nepomohli.

Zamysleme se nad tím, jak by se proces sčítání mohl zrychlit.

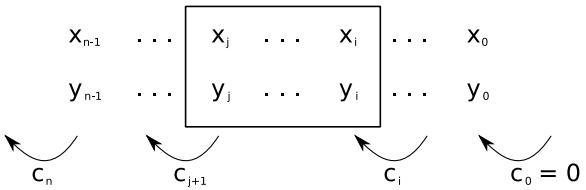


Sčítání školním algoritmem.

Přenosy v blocích

Jediné, co nás při sčítání brzdí, jsou přenosy mezi jednotlivými řády. Každý řád, aby vydal součet, musí počkat na to, až dopočítají všechny předcházející řády. Teprve pak se totiž dozví přenos. Kdybychom ovšem přenosy dokázali spočítat nějakým způsobem paralelně, máme vyhráno. Jakmile známe všechny přenosy, součet už zvládneme dopočítat na konstantně mnoho hladin – tedy v konstantním čase.

Podívejme se na libovolný *blok* v našem součtu. Tak budeme říkat číslům $x_j \ldots x_i$ a $y_j \ldots y_i$ v nějakém intervalu indexů $\langle i, j \rangle$. Přenos c_{j+1} vystupující z tohoto bloku závisí mimo hodnot sčítanců už pouze na přenosu c_i, který do bloku vstupuje.



Blok součtu.

Z tohoto pohledu se můžeme na blok také dívat jako na nějakou funkci, která dostane jednobitový vstup a vydá jednobitový výstup. To je nám příjemné, neboť takových funkcí existují jenom čtyři typy:

1. $f(x) = 0$, (0)
2. $f(x) = 1$, (1)
3. $f(x) = x$, ($<$ – kopírování)
4. $f(x) = \neg x$.

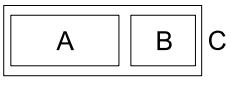
Jak se za chvíli ukáže, poslední případ, kdy by nějaký blok předával opačný přenos, než do něj vstupuje, navíc nikdy nemůže nastat. Pojďme si to rozmyslet. Jednobitové bloky se chovají velice jednoduše:

Z prvního bloku evidentně vždy vyleze 0, ať do něj vstoupí jakýkoli přenos. Poslední blok naopak sám o sobě přenos vytváří, ať již do něj vleze jakýkoliv. Bloky prostřední se chovají stejně, a to tak, že samy o sobě žádný přenos nevytvoří, ale pokud do nich nějaký přijde, tak také odejde.

$\boxed{\begin{matrix}0\\0\end{matrix}}$	$\boxed{\begin{matrix}0\\1\end{matrix}}$	$\boxed{\begin{matrix}1\\0\end{matrix}}$	$\boxed{\begin{matrix}1\\1\end{matrix}}$
0	<	<	1

Tabulka triviálních bloků.

Mějme nyní nějaký větší blok C složený ze dvou menších podbloků A a B, jejichž chování už známe. Z toho můžeme odvodit, jak se chová celý blok:



A \ B	0	1	<
0	0	0	0
1	1	1	1
<	0	1	<

Skládání chování bloků.

Pokud vyšší blok přenos pohlcuje, pak ať se už nižší blok chová jakkoli, složení těchto bloků musí vždy pohlcovat. V prvním řádku tabulky jsou tudíž nuly. Analogicky, pokud vyšší blok generuje přenos, tak ten nižší na tom nic nezmění. V druhém řádku tabulky jsou tedy samé jedničky. Zajímavější případ nastává, pokud vyšší blok kopíruje – tehdy záleží čistě na chování nižšího bloku.

Všimněme si, že skládání chování bloků je vlastně úplně obyčejné skládání funkcí. Nyní bychom mohli prohlásit, že budeme počítat nad tříprvkovou abecedou, a že celou tabulku dokážeme spočítat jedním jediným hradlem. Pojďme si však rozmyslet, jak bychom takovouto věc popsali čistě binárně. Jak tedy tyto tři stavy popisovat pouze několika bity?

Evidentně nám k tomuto binárnímu zakódování tří stavů budou stačit bity dva. Označme si je jako p a q. Tato dvojice může nabývat hned čtyř možných hodnot, kterým přiřadíme tři možná chování bloku. Toto kódování můžeme zvolit zcela libovolně, ale pokud si ho zvolíme šikovně, ušetříme si dále práci při kompozici. Zvolme si tedy kódování takto:

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

Tomu, že blok kopíruje, odpovídá dvojice $p = 1$; $q =$ *cokoliv*. V ostatních případech bude p nulové a q nám bude říkat, co je na výstupu příslušného bloku. Jinými slovy $p = 0$ znamená, že funkce je konstanta, přičemž q říká jaká; naproti tomu $p = 1$ znamená, že funkce je identita, ať už je q cokoli.

Pojďme si nyní ukázat, jak bude celé skládání bloků vypadat. Rozmysleme si, kdy je p celého dvojbloku jedničkové, tedy kdy celý dvojblok kopíruje. To nastane

tehdy, pokud kopírují obě jeho části, a tedy $p = p_a \;\&\; p_b$. Dále q bude rovno jedničce, pokud $q = (\neg p_a \;\&\; q_a) \vee (p_a \;\&\; q_b)$.

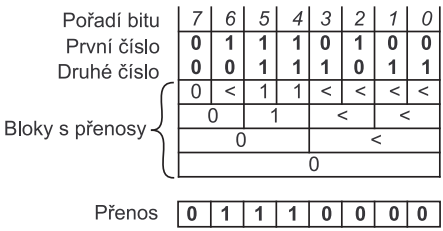
Skládání chování bloků lze tedy popsat buď ternárně – tabulkou, ale lze to i binárně výše uvedeným předpisem.

Nyní si tedy můžeme dopředu vypočítat chování bloku velikosti jedna, poté z nich skládáním bloků velikosti dva, dál velikosti čtyři, osm, atd ...

Paralelní sčítání

Paralelní algoritmus na sčítání už zkonstruujeme poměrně snadno. Bez újmy na obecnosti budeme předpokládat, že počet bitů vstupních čísel je mocnina dvojky, jinak si vstup doplníme nulami, což výsledný čas běhu algoritmu zhorší maximálně konstanta-krát.

1. Spočteme chování bloků velikosti 1. ($\mathcal{O}(1)$ hladin)
2. Postupně počítáme chování bloků[⟨2⟩] velikosti 2, 4, 8, ..., 2^k. ($\mathcal{O}(\log n)$ hladin, na nichž se skládají bloky)
3. $c_0 \leftarrow 0$ (přenos do nejnižšího řádu je vždy 0)
4. Určíme c_n podle c_0 a chování (jediného) bloku velikosti n.
5. Postupně dopočítáme přenosy na hranicích dělitelných 2^k „zahušťováním“: jakmile víme c_{2^k}, můžeme dopočítat $c_{2^k+2^{k-1}}$ podle chování bloku $\langle 2^k, 2^k + 2^{k-1}\rangle$. ($\mathcal{O}(\log n)$ hladin, na nichž se dosazuje)
6. Sečteme: $\forall i : z_i = x_i \oplus y_i \oplus c_i$.



Výpočet přenosu.

Algoritmus pracuje v čase $\mathcal{O}(\log n)$. Hradel je použito 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 dohromady se sečte na $\Theta(n)$.

Třídění

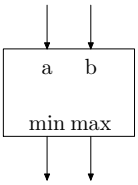
Nyní se podíváme na druhý problém, a to na problém třídění. Již známe poměrně efektivní sekvenční algoritmy, které dovedou třídit v čase $\mathcal{O}(n \log n)$. Byli

[⟨2⟩] myslíme „přirozeně zarovnané“ bloky, tedy takové, jejichž poloha je násobkem velikosti

bychom jistě rádi, kdybychom to zvládli ještě rychleji. Pojďme se podívat, zda by nám v tom nepomohlo problém paralelizovat.

Budeme při tom pracovat ve výpočetním modelu, kterému se říká komparátorová síť. Ta je postavená z hradel, kterým se říká komparátory.

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



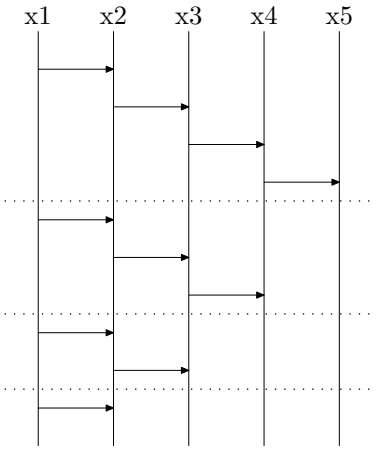
Komparátor

Komparátor umí porovnat dvě hodnoty a rozhodnout, která z nich je větší a která menší. Nevrací však booleovský výsledek jako běžné hradlo, ale má dva výstupy, přičemž na jednom vrátí menší ze vstupních hodnot a na druhém větší.

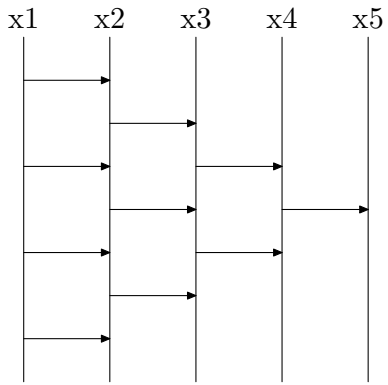
Výstupy komparátorů se nevětví. Nemůžeme tedy jeden výstup „rozdvojit“ a připojit ho na dva vstupy. (Větvení by dokonce ani nemělo smysl, protože zatímco rozdvojit bychom mohli, sloučit už ne. Pokud tedy chceme, aby síť měla n vstupů i n výstupů, rozdvojení stejně nesmíme provést, i kdybychom jej měli povolené.)

Příklad: *Bubble sort*

Obrázek Bubble 1 ilustruje použití komparátorů pro třídění Bubble sortem. Šipky představují jednotlivé komparátory. Výpočet však ještě můžeme vylepšit.



Bubble 1



Bubble 2

Snažíme se výpočet co nejvíce paralelizovat (viz obrázek Bubble 2). Jak je vidět, komparátory na sebe nemusejí čekat. Tím můžeme výpočet urychlit a místo

času $\Theta(n^2)$ docílit časové složitosti $\Theta(n)$. V obou případech je zachován kvadratický prostor.

Nyní si ukážeme ještě rychlejší třídící algoritmus. Půjdeme na něj však trochu „od lesa“. Nejdříve vymyslíme síť, která bude umět třídit jenom něco - totiž bitonické posloupnosti.

Definice: Řekneme, že posloupnost $x_0, \ldots, x_{n-1}$ je *čistě bitonická* právě tehdy, když pro nějaké $x_k, k \in \{0, \ldots, n-1\}$ platí, že všechny prvky před ním (včetně jeho samotného) tvoří rostoucí poslopnost, kdežto prvky stojící za ním tvoří poslopnost klesající. Formálně zapsáno musí platit, že:

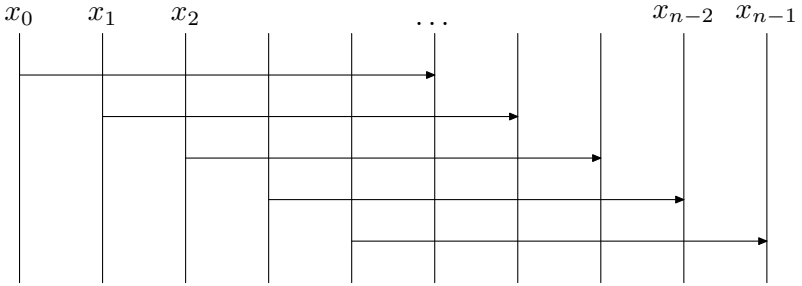
$$x_0 \leq x_1 \leq \ldots \leq x_k \geq x_{k+1} \geq \ldots \geq x_{n-1}.$$

Definice: Posloupnost $x_0 \ldots x_{n-1}$ je *bitonická*, právě když $\exists\, j \in \{0, \ldots, n-1\}$, pro které je rotace původní posloupnosti o j prvků, tedy posloupnost

$$x_j, x_{(j+1) \bmod n}, \ldots, x_{(j+n-1) \bmod n},$$

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 se pak stane i-tým, maximum $(i + n/2)$-tým prvkem výstupu.



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

Lemma: Pokud separátor dostane na vstupu bitonickou posloupnost, pak jeho výstup $y_0, \ldots, y_{n-1}$ splňuje:

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

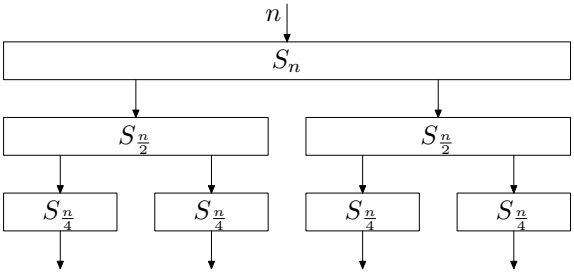
(ii) Pro všechna $i, j < n/2$ platí $y_i < y_{j+n/2}$.

Separátor nám tedy jednu bitonickou posloupnost na vstupu rozdělí na dvě bitonické posloupnosti, přičemž navíc každý prvek první posloupnosti $(y_0, \ldots, y_{n/2-1})$ je menší nebo roven prvkům druhé posloupnosti $(y_{n/2}, \ldots, y_{n-1})$.

Než přistoupíme k důkazu lemmatu, ukažme si, k čemu se nám bude hodit.

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

Třídička dostane na vstupu bitonickou posloupnost. Separátor S_n ji pak dle lemmatu rozdělí na dvě bitonické posloupnosti, kdy je každý prvek z první posloupnosti menší než libovolný prvek z druhé. Tyto poloviny pak další separátory rozdělí na čtvrtiny, …, až na konci zbudou pouze jednoduché posloupnosti délky jedna (zjevně setříděné), které mezi sebou mají požadovanou nerovnost – tedy každá posloupnost (nebo spíše prvek) nalevo je $\leq$ než prvek napravo od něj.



Bitonická třídička B_n

Jak je vidět, bitonická třídička nám libovolnou bitonickou posloupnost délky n setřídí na $\Theta(\log n)$ hladin.

Nyní se dá odvodit, že pokud umíme třídit bitonické posloupnosti, umíme setřídit všechno. Vzpomeňme si na třídění sléváním – Merge sort. To funguje tak, že začne s jednoprvkovými posloupnostmi, které jsou evidentně setříděné, a poté vždy dvě setříděné posloupnosti slévá do jedné. Kdybychom nyní uměli paralelně slévat, mohli bychom vytvořit i paralelní Merge sort. Jinými slovy, potřebujeme dvě rostoucí posloupnosti nějak efektivně slít do jedné setříděné. Uvědomme si, že to zvládneme jednoduše – stačí druhou z posloupností obrátit a „přilepit“ za první, čímž vznikne bitonická posloupnost, kterou poté můžeme setřídit naší bitonickou třídičkou.

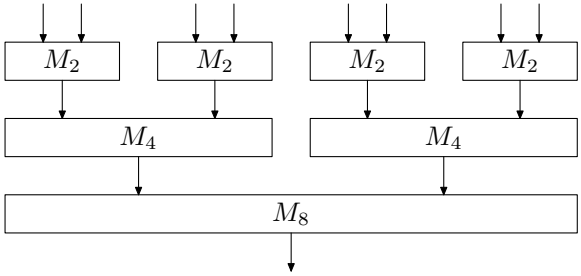
Příklad: *Merge sort*

Bitonická třídička se tedy dá použít ke slévání setříděných posloupností. Ukažme si, jak s její pomocí sestavíme součástky *slévačky* M_n:

Setříděné posloupnosti $x_0, \ldots, x_{n-1}$ a $y_0, \ldots, y_{n-1}$ spojíme do jedné bitonické posloupnosti $x_0, \ldots, x_{n-1}, y_{n-1}, \ldots, y_0$. Z této posloupnosti vytvoříme pomocí bitonické třídičky B_{2n} setříděnou posloupnost. Vytvoříme tedy blok M_{2n}, který se ovšem sestává de facto pouze z bloku B_{2n}, jehož druhá polovina vstupů je zapojena v obráceném pořadí.

Nyní se pokusme odhadnout časovou složitost. Náš MergeSort bude mít řádově hloubku bloků $\log n$. V každém bloku M_n je navíc ukryta bitonická třídička s taktéž logaritmickou hloubkou. Celková hloubka tedy bude $\log 2 + \log 4 + \ldots + \log 2^k + \ldots + \log n$. Po sečtení nakonec dostáváme výslednou časovou složitost $\Theta(\log^2 n)$.

Dodejme ještě, že existuje i třídicí algoritmus, kterému stačí jen $\mathcal{O}(\log n)$ hladin.



Paralelní MergeSort.

Jeho multiplikativní konstanta je však příliš veliká, takže je v praxi nepoužitelný.

Vraťme se nyní k důkazu lemmatu, který jsme na předminulé stránce vynechali.

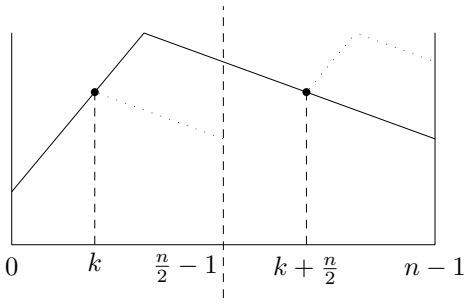
Důkaz Lemmatu:

(i) Nejprve nahlédneme, že lemma platí, je-li vstupem čistě bitonická posloupnost. Dále BÚNO předpokládejme, že vrchol posloupnosti je v první polovině (kdyby byl vrchol za polovinou, stačilo by zrcadlově obrátit posloupnost i komparátory a řešili bychom stejný problém). Nyní si definujme $k := \min j : x_j > x_{j+n/2}$. (Pokud by takové k neexistovalo, znamenalo by to, že vstupní posloupnost je monotónní. Separátor by tedy nic nedělal a pouze zkopíroval vstup na výstup, což jistě lemma splňuje.) Nyní si všiměme, že jakmile jednou začne platit, že prvek na levé straně je menší než na pravé, bude nám tato relace platit až do konce. Označme vrchol vstupní posloupnosti jako x_m. 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.

Do pozice k tedy separátor bude pouze kopírovat vstup na výstup, od pozice k dál už jen prohazuje. 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. Podobně úsek $k+n/2$ až $n-1$ nahradíme čistě bitonickou posloupností. Obě poloviny tedy budou bitonické.

Dostaneme-li na vstupu obecnou bitonickou posloupnost, představíme si, že je to čistě bitonická posloupnost zrotovaná o r prvků (BÚNO doprava). 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.

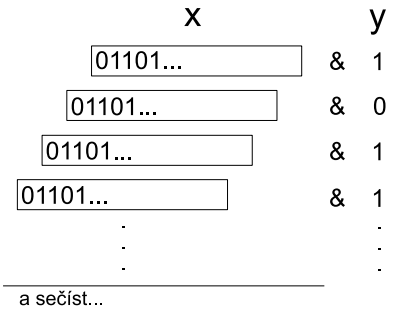
(ii) Z důkazu (i) pro čistě bitonickou posloupnost víme, že $y_0 \ldots y_{n/2-1}$ je čistě bitonická a bude rovna $x_0 \ldots x_{k-1}, x_{k+n/2} \ldots x_{n-1}$ pro vhodné k a navíc bude mít maximum v x_{k-1} nebo $x_k + n/2$. Mezi těmito body ovšem ve vstupní posloupnosti určitě neležel žádný x_i menší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



posloupnost prohozená separátorem

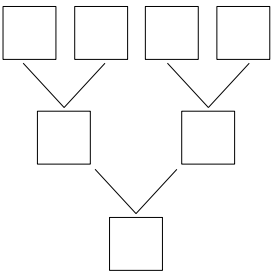
Paralelní násobení

Podobně jako u sčítání si vzpomeneme na školní algoritmus – tentokráte však pro násobení. To fungovalo tak, že jsme si jedno ze dvou binárních čísel na vstupu (říkejme mu třeba x) n-krát posouvali. Tam kde pak byly v čísle y jedničky, příslušné kopie x jsme sečetli. Jinými slovy tedy násobení umíme převést na nějaké posuny (ty lze realizovat pouze „předrátováním“ – nic nás nestojí), násobení jedním bitem (což je AND) a nakonec potřebujeme výsledných n čísel sečíst.



Školní sčítání.

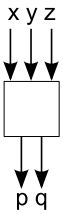
Jak nyní sečíst n n-bitových čísel..? Nabízí se využít osvědčený „stromeček“ – sčítat dvojice čísel, výsledky pak opět po dvojicích sečíst, až na konci vyjde jediný výsledek.



Stromeček

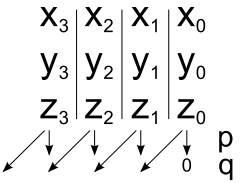
Toto řešení by však vedlo na časovou složitost $\Theta(\log^2 n)$. To je sice dle našich měřítek docela efektivní, ale překvapivě to jde ještě lépe – totiž na $\Theta(\log n)$ hladin. Této složitosti dosáhneme malým trikem.

Vymyslíme obvod konstantní hloubky, který na vstupu dostane tři čísla. Odpoví pak dvěma čísly takovými, že budou mít stejný součet jako původní tři čísla. Jinými slovy pomocí tohoto obvodu budeme umět sečtení tří čísel převést na sečtení dvou čísel.



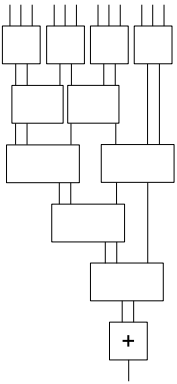
$$x + y + z = p + q$$

Všimněme si, že když sčítáme tři bity, může být přenos do vyššího řádu nula či jednička. Vezmeme si tedy bity x_i, y_i, z_i a ty sečteme. To nám dá dvoubitový výsledek, přičemž nižší bit z tohoto výsledku pošleme do čísla p, vyšší do čísla q.



Redukování sčítání

Toto zredukování sčítání nám nyní umožní opět stavit strom, byť o maličko složitější.



Složitější stromeček

Pokud jsme měli na začátku n čísel, po první redukci nám jich zbývá jen $2/3 \cdot n$ a obecně v k-té hladiné $(2/3)^k \cdot n$. Znamená to, že čísel nám ubývá exponenciálně, takže počet hladin bude logaritmický. Redukující obvod je při tom jen konstantně hluboký, takže celé redukování zvládneme v čase $\Theta(\log n)$. Na konci Složitějšího stromečku pak máme umístěnou jednu obyčejnou sčítačku, která zbývající dvě čísla sečte v logaritmickém čase.

Sečtení všech n čísel tedy zabere $\Theta(\log n)$ hladin.

Když se nyní vrátíme k násobení, zbývá nám vyřešit posouvání a ANDování. Uvědomme si, že to je plně paralelení a zvládneme ho za konstantně mnoho hladin. Celé násobení tedy zvládneme v logaritmickém čase.

6. Vyhledávání v textu

(zapsal: Petr Jankovský)

Nyní se budeme věnovat následujícímu problému: v textu délky S (seně) budeme chtít najít všechny výskyty hledaného slova délky J (jehly). Nejprve se podívejme na jeden primitivní algoritmus, který nefunguje. Je ale zajímavé rozmyslet si, proč.

Hloupý algoritmus

Začneme prvním písmenkem hledaného slova a budeme postupně procházet text, až najdeme první výskyt počátečního písmenka. Poté budeme testovat, zda souhlasí i písmenka další. Pokud nastane neshoda, v hledaném slově se vrátíme na začátek a v textu pokračujeme znakem, ve kterém neshoda nastala. Podívejme se na příklad.

Příklad: Budeme hledat slovo `jehla` v textu `jevkupcejejehla`. Vezmeme si tedy první písmenko `j` v hledaném slově a zjistíme, že v textu se nachází hned na začátku. Vezmeme tedy další písmenko `e`, které se vyskytuje jako druhé i v textu. Při třetím písmenku ale narazíme na neshodu. V tuto chvíli tedy zresetujeme a opět hledáme výskyt písmenka `j`, tentokrát však až od třetího písmene v textu. Takto postupujeme postupně dál, až narazíme na další `je`, které ovšem není následováno písmenem `h`, tudíž opět zresetujeme a nakonec najdeme shodu s celým hledaným řetězcem. V tomto případě tedy algoritmus našel hledané slovo.

Tento algoritmus však zjevně může hanebně selhat. Může se stát, že začneme porovnávat, až v jednu chvíli narazíme na neshodu. Celý tento kus tedy přeskočíme. Při tom se ale v tomto kusu textu mohl vyskytovat nějaký překrývající se výskyt hledané „jehly“. Hledejme například řetězec `kokos` v textu `clanekokokosu`. Algoritmus tedy začne porovnávat. Ve chvíli kdy najde prefix `koko` a na vstupu dostane `k`, dochází k neshodě. Proto algoritmus zresetuje a pokračuje v hledání od tohoto znaku. Najde sice ještě výskyt `ko`, ovšem s dalším písmenkem `s` již dochází k neshodě a algotimus selže. Nesprávně se totiž „upnul“ na první nalezené `koko` a s dalším `k` pak „zahodil“ i správný začátek.

Máme tedy algoritmus, který i když je špatně, tak funguje určitě kdykoli se první písmenko hledaného slova v tomto slově už nikde jinde nevyskytuje – což `jehla` splňovala, ale `kokos` už ne.

Hloupý algoritmus se na každé písmenko textu podívá jednou, tudíž časová složitost bude lineární s délkou textu ve kterém hledáme – tedy $\mathcal{O}(S)$.

Pomalý algoritmus

Zkusíme algoritmus vylepšit tak, aby fungoval správně: pokud nastane nějaká neshoda, vrátíme se zpátky těsně za začátek toho, kdy se nám to začalo shodovat. To je ovšem vlastně skoro totéž, jako brát postupně všechny možné začátky v „seně“ a pro každý z něj ověřit, jestli se tam „jehla“ nachází či nikoliv.

Tento algoritmus evidentně funguje. Běží však v čase: S možných začátků, krát čas potřebný na jedno porovnání (zda se na dané pozici nenachází „jehla“), což nám může trvat až J. Proto je časová složitost $\mathcal{O}(SJ)$. V praxi bude algoritmus často rychlejší, protože typicky velmi brzo zjistíme, že se řetězce neshodují, ale je možné vymyslet vstup, kde bude potřeba porovnání opravdu tolik.

Nyní se pokusme najít takový algoritmus, který by byl tak rychlý, jako *Hloupý algoritmus*, ale chytrý, jako ten *Pomalý*.

Chytrý algoritmus

Než vlastní algoritmus vybudujeme, zkusíme se cestou naučit přemýšlet o řetězcích občas trochu překrouceným způsobem. Podívejme se na ještě jeden příklad.

Příklad: Vezměme si například staré italské přízvisko `barbarossa`, které znamená Rudovous. Představme si, že takovéto slovo hledáme v nějakém textu, který začíná `barbar`. Víme, že až sem se nám hledaný řetězec shodoval. Řekněme, že další písmenko textu se shodovat přestane – místo `o` načteme například opět `b`. *Hloupý algoritmus* by velil vrátit se k `a` a od něj hledat dál. Uvědomme si ale, že když se vracíme z `barbar` do `arbar` (tedy řetězce, který již známe), můžeme si předpočítat, jak dopadne hledání, když ho pustíme na něj. V předpočítaném bychom tedy chtěli ukládat, že když máme řetězec `arbar`, tak `ar` a `r` nám do hledaného nepasuje a až `bar` se bude shodovat. Tedy místo toho, abychom spustili nové hledání od `a`, můžeme ho spustit až od `b`. Co víc, my dokonce víme, jak dopadne to – pokud totiž nastane neshoda po přečtení `barbar`, je to stejné, jako kdybychom přečetli pouze `bar`, na které se (původne neshodující se) `b` už navázat dá. Kdyby se nedalo navázat ani tam, tak bychom opět zkracovali… Nejen, že tedy víme, kam se máme vrátit, ale víme dokonce i to, co tam najdeme.

Myšlenka, ke které míříme, je předpočítat si nějakou tabulku, která nám bude říkat, jak se máme při hledání vracet a jak to dopadne, a pak už jenom prohlédávat s použitím této tabulky.

Aby se nám o těchto algoritmech lépe mluvilo a především psalo, pojďme si povědět několik definic.

Definice:

- *Abeceda* Σ je konečná množina znaků⟨3⟩, ze kterých tvoříme text, řetězce, slova.

⟨3⟩ Můžeme při tom jít až do extrémů. Příkladem extrémních abeced je binární

- Σ^* je množina všech slov nad abecedou Σ. Čili množina všech neprázdných konečných posloupností znaků ze Σ.

Značení: Aby se nám nepletlo značení, budeme rozlišovat proměnné pro slova, proměnné pro písmenka a proměnné pro čísla.

- *Slova* budeme značit malými písmenky řecké abecedy α, β, ...
- ι bude označovat „jehlu“
- σ bude označovat „seno“
- *Znaky* označíme malými písmeny latinky a, b, ...
- *Čísla* budeme značit velkými písmeny A, B, ...
- *Délka slova* $|\alpha|$ pro $\alpha \in \Sigma^*$ je počet jeho znaků.
- *Prázdné slovo* značíme písmenem ε, $|\varepsilon| = 0$.
- *Zřetězení* $\alpha\beta$ vznikne zapsáním slov α a β za sebe. Platí $|\alpha\beta| = |\alpha| + |\beta|$, $\alpha\varepsilon = \varepsilon\alpha = \alpha$.
- $\alpha[k]$ je k-tý znak slova α, indexujeme od 0.
- $\alpha[k : l]$ je podslovo začínající k-tým znakem a l-tý znak je první, který v něm není. Jedná se tedy o podslovo skládající se z $\alpha[k]$,$\alpha[k+1]$,...,$\alpha[l-1]$. Platí tedy: $\alpha[k : k] = \varepsilon$, $\alpha[k : k+1] = \alpha[k]$. Jednu (či obě) meze můžeme i vynechat, tento zápis pak bude znamenat buď „od začátku slova až někam“, nebo „odněkud až do konce“:
- $\alpha[: k]$ je *prefix* obsahující prvních k znaků slova α ($\alpha[0]$,...,$\alpha[k-1]$).
- $\alpha[k :]$ je *suffix* obsahující znaky slova α počínaje k-tým znakem až do konce.
- $\alpha[:] = \alpha$

Všimněme si, že prázdné slovo je prefixem, suffixem i podslovem jakéhokoliv slova včetně sebe sama. Každé slovo je také prefixem, suffixem i podslovem sebe sama. To se ne vždy hodí. Někdy budeme chtít říct, že nějaké slovo je *vlastním* prefixem nebo suffixem. To bude znamenat, že to nebude celé slovo.

α je *vlastní prefix* slova $\beta \equiv \alpha$ je prefix β & $\alpha \neq \beta$.

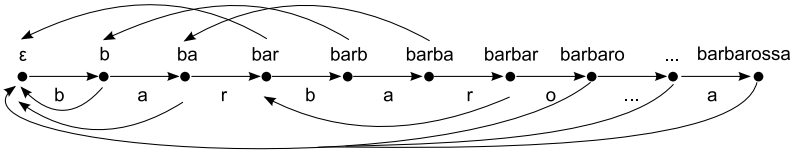
Vyhledávací automat (Knuth, Morris, Pratt)

Vyhledávací automat bude graf, jehož vrcholům budeme říkat *stavy*. Jejich jména budou prefixy hledaného slova a hrany budou odpovídat tomu, jak jeden prefix můžeme získat z předchozího prefixu přidáním jednoho písmene. Počáteční stav je

---

abeceda složená pouze z nul a jedniček. Příklad z druhého konce (který rádi dělají lingvisti) je abeceda, která má jako abecedu všechna česká slova. Všechny české věty, pak nejsou nic jiného, než slova nad touto abecedou. Použitá abeceda tedy může být i relativně obrovská. Dalším takovým příkladem může být Unicode. Pro naše potřeby ale zatím budeme předpokládat, že abeceda je nejen konstantně velká, ale i rozumně malá. Budeme si moci tedy dovolit například indexovat pole znakem abecedy (kdybychom nemohli, tak bychom místo pole použili například hashovací tabulku, či něco podobného...).

prázdné slovo ε a koncový je celá ι. Dopředné hrany grafu budou popisovat přechod mezi stavy ve smyslu zvětšení délky jména stavu (dopředná funkce $h(\alpha)$, určující znak na dopředné hraně z α). Zpětné hrany grafu budou popisovat přechod (zpětná funkce $z(\alpha)$) mezi stavem α a nejdelším vlastním suffixem α, který je prefixem ι, když nastane neshoda.



Vyhledávací automat.

Hledej(σ):

1. $\alpha \leftarrow \varepsilon$.
2. Pro $x \in \sigma$ postupně:
3. Dokud $h(\alpha) \neq x$ & $\alpha \neq \varepsilon : \alpha \leftarrow z(\alpha)$.
4. Pokud $h(\alpha) = x : \alpha \leftarrow \alpha x$.
5. Pokud $\alpha = \iota$, ohlásíme výskyt.

Vstupem je ι, hledané slovo (jehla) délky $J = |\iota|$ a σ, text (seno) délky $S = |\sigma|$. Výstupem jsou všechny výskyty hledaného slova ι v textu σ, tedy množina $\{k \mid \sigma[k : k + J] = \iota\}$

Pojďme nyní dokázat, že tento algoritmus správně ohlásí všechny výskyty.

Definice: $\alpha(\tau)$:= stav automatu po přečtení τ

Invariant: Pokud algoritmus přečte nějaký vstup, nachází se ve stavu, který je nejdelším suffixem přečteného vstupu, který je nějakým stavem. $\alpha(\tau)$ = nejdelší stav (nejdelší prefix jehly), který je suffixem τ (přečteného vstupu).

Pojďme si rozmyslet, že z tohoto invariantu ihned plyne, že algoritmus najde to, co má. Kdykoli totiž ohlásí nějaký výskyt, tak tam tento výskyt opravdu je. Kdykoli pak má nějaký výskyt ohlásit, tak se v této situaci jako suffix toho právě přečteného textu vyskytuje hledané slovo, přičemž hledané slovo je určitě stav a zároveň nejdelší ze všech existujících stavů. Takže invariant opravdu říká, že jsme právě v koncovém stavu a algoritmus nám tedy ohlásí výskyt.

Důkaz: (invariantu) Indukcí podle kroku algoritmu. Na začátku pro prázdný načtený vstup invariant triviálně platí, tedy prázdný suffix τ je prefixem ι. V kroku n máme načtený vstup τ a k němu připojíme znak x. Invariant nám říká, že nejdelší stav, který je suffixem, je nejdelší suffix, který je stavem. Nyní se ptáme, jaký je nejdelší stav, který se dá „napasovat“ na konec řetězce τx. Kdykoli však takovýto suffix máme, tak z něj můžeme x na konci odebrat, čímž dostaneme suffix slova τ.

Tedy: pokud β je neprázdným suffixem slova τx, pak $\beta = \gamma x$, kde γ je suffix τ.

Suffix, který máme sestrojit, tedy vznikne z nějakého suffixu slova τ připsáním x. Chceme najít nejdelší suffix slova τx, který je stavem, takže chceme najít

i nejdelší suffix původního slova τ, za který se dá přidat x tak, aby vyšlo jméno stavu. Stačí tedy už jen „probírat“ suffixy slova τ od nejdelšího po nejkratší, zkoušet k nim přidávat x a až to půjde, tak jsme našli nejdelší suffix τx. Přesně toto ovšem algoritmus dělá, neboť zpětná funkce mu vždy řekne nejbližší kratší suffix, který je stavem. Pokud pak nemůžeme x přidat ani do ε, pak je řešením prázdný suffix. Algoritmus tedy funguje. ♡

Nyní pojďmě zkoumat to, jak je ve skutečnosti náš algoritmus rychlý. K tomu bychom si ale nejdřív měli říct, jak přesně budeme automat reprezentovat. V algoritmu vystupují nějaká porovnávání stavů, přičemž není úplně jasné, jak zařídit, aby vše trvalo konstantně dlouho. Vyjde nám to ale docela snadno. K reprezentaci automatu nám totiž budou stačit pouze dvě pole.

Reprezentace automatu: Očíslujeme si stavy délkami příslušných prefixů, tedy čísly $0 \ldots J$. Poté ještě potřebujeme nějakým způsobem zakódovat dopředné a zpětné hrany. Vzhledem k tomu, že z každého vrcholu vede vždy nejvýše jedna dopředná a nejvýše jedna zpětná, tak nám evidentně stačí pamatovat si pro každý typ hran pouze jedno číslo na vrchol. Budeme mít tedy nějaké pole dopředných hran, které nám pro každý stav řekne, jakým písmenkem je nadepsaná dopředná hrana ze stavu I do $I+1$. To jsou ale přesně písmenka jehly, takže si stačí pamatovat jehlu samotnou. Čili z I do $I+1$ vede hrana nadepsaná $\iota[I]$. Pro zpětné hrany pak budeme potřebovat pole Z, které nám pro stav I řekne číslo stavu, do kterého vede zpětná hrana. Tedy $Z[I]$ je cíl zpětné hrany ze stavu I. S touto reprezentací již dokážeme naši hledací proceduru přímočaře přepsat tak, aby sahala pouze do těchto dvou polí:

1. $I \leftarrow 0$.
2. Pro znaky x z textu:
3. Dokud $\iota[I] \neq x$ & $I \neq 0 : I \leftarrow Z[I]$.
4. Pokud $\iota[I] = x$, pak $I \leftarrow I + 1$.
5. Pokud $I = J$, ohlásíme výskyt.

Zatím se v algoritmu ještě skrývá drobná chyba – totiž algoritmus se občas zeptá na dopřednou hranu z posledního stavu. Pokud jsme právě ohlásili výskyt (jsme tedy v posledním stavu) a přijde nějaký další znak, algoritmus se ptá, zda je roven tomu, co je na dopředné hraně z posledního stavu. Ta ale ovšem neexistuje. Jednoduše to ale napravíme tak, že si přidáme fiktivní hranu, na které se vyskytuje nějaké „nepísmenko“ – něco, co se nerovná žádnému jinému písmenku. Zajistíme tak, že se po této hraně nikdy nevydáme. Dodefinujeme tedy $\iota[J]$ odlišně od všech znaků.⟨4⟩

Lemma: Funkce *Hledej* běží v čase $\mathcal{O}(S)$.

Důkaz: Funkce *Hledej* chodí po dopředných a zpětných hranách. Dopředných hran projdeme určitě maximálně tolik, kolik je délka sena. Pro každý znak přečtený ze sena

⟨4⟩ V jazyce C se toto dodefinování provede vlastně zadarmo, neboť každý řetězec je v něm ukončen znakem s kódem nula, který se ve vstupu nevyskytne... Algoritmus bude tedy fungovat i bez tohoto dodefinování. V jiných jazycích je ale třeba na něj nezapomenout!

totiž jdeme nejvýše jednou po dopředné hraně. Se zpětnými hranami se to má tak, že na jeden přečtený znak z textu se můžeme po zpětné hraně vracet maximálně J-krát. Z tohoto by nám však vyšla složitost $\mathcal{O}(JS)$, čímž bychom si nepomohli. Zachrání nás ale přímočarý potenciál. Uvědomme si, že chůze po dopředné hraně zvýší I o jedna a chůze po zpětné hraně I sníží alespoň o jedna. Vzhledem k tomu, že I není nikdy záporné a na začátku je nulové, zjistíme, že kroků zpět může být maximálně tolik, kolik kroků dopředu. Časová složitost hledání je tedy lineární vzhledem k délce sena. ♡

Nyní nám zbývá na první pohled maličkost – totiž zkonstruovat automat. Zkonstruovat dopředné hrany zvládneme zjevně snadno, jsou totiž explicitně popsané hledaným slovem. Těžší už to bude pro hrany zpětné. Využijeme k tomu následující pozorování:

Pozorování: Představme si, že automat už máme hotový a tím, že budeme sledovat jeho chování, chceme zjistit, jak v něm vedou zpětné hrany. Vezměme si nějaký stav β. To, kam z něj vede zpětná hrana zjistíme tak, že spustíme automat na řetězec β bez prvního písmenka a stav, ve kterém se automat zastaví, je přesně ten, kam má vést i zpětná hrana z β. Jinými slovy víme, že $z(\beta) = \alpha(\beta[1:])$. Proč takováto věc funguje? Všiměme si, že definice z a to, co nám o α říká invariant, je téměř totožné – $z(\beta)$ je nejdelší vlastní suffix β, který je stavem, $\alpha(\beta)$ je nejdelší suffix β, který je stavem. Jediná odlišnost je v tom, že definice z narozdíl od definice α zakazuje nevlastní suffixy. Jak nyní vyloučit suffix β, který by byl roven β samotné? Zkrátíme β o první znak. Tím pádem všechny suffixy β bez prvního znaku jsou stejné jako všechny vlastní suffixy β.

K čemu je toto pozorování dobré? Rozmysleme si, že pomocí něj už dokážeme zkonstruovat zpětné hrany. Není to ale trochu divné, když při simulování automatu na řetězec bez prvního znaku už zpětné hrany potřebujeme? Není. Za chvíli zjistíme, že takto můžeme zjišťovat zpětné hrany postupně – a to tak, že používáme vždy jenom ty, které jsme už sestrojili.

Takovémuhle přístupu, kdy při konstruování chtěného už používáme to, co chceme sestrojit, ale pouze ten kousek, který již máme hotový, se v angličtině říká *bootstrapping*⟨5⟩. Všimněme si, že při výpočtu se vstupem β projde automat jenom prvních $|\beta|$ stavů. Automat se evidentně nemůže dostat dál, protože na každý krok dopředu (doprava) spotřebuje písmenko β. Takže kroků doprava je maximálně tolik, kolik je $|\beta|$. Jinými slovy kdybychom již měli zkonstruované zpětné hrany pro prvních $|\beta|$ stavů (tedy $0 \ldots |\beta|-1$), tak při tomto výpočtu, který potřebujeme na zkonstru-

⟨5⟩ Z tohoto slova vzniklo i *bootování* počítačů, kdy operační systém v podstatě zavádí sám sebe. Bootstrap znamená česky štruple – tedy očko na konci boty, které slouží k usnadnění nazouvání. A jak souvisí štruple s algoritmem? To se zase musíme vrátit k příběhům o baronu Prášilovi, mezi nimiž je i ten, ve kterém baron Prášil vypráví o tom, jak sám sebe vytáhl z bažiny za štruple. Stejně tak i my budeme algoritmus konstruovat tím, že se budeme sami vytahovat za štruple, tedy bootstrappovat.

ování zpětné hrany z β, ještě tuto zpětnou hranu nemůžeme potřebovat. Vystačíme si s těmi, které již máme zkonstruované.

Nabízí se tedy začít zpětnou hranou z prvního znaku (která vede evidentně do ε), pak postupně brát další stavy a pro každý z nich si spočítat, kdy spustíme automat na jméno stavu bez prvního znaku a tím získáme další zpětnou hranu. Toto funguje, ale je to kvadratické .... Máme totiž J stavů a pro každý z nich nám automat běží v čase až lineárním s J. Jak z toho ven?

Z prvního stavu povede zpětná funkce do ε. Pro další stavy chceme spočítat zpětnou funkci. Z druhého stavu $\iota[0:2]$ tedy automat spustíme na $\iota[1:2]$, dále pak na $\iota[1:3]$, $\iota[1:4]$, atd. Ty řetězce, pro které potřebujeme spoštět automat, abychom dostali zpětné hrany, jsou tedy ve skutečnosti takové, že každý další dostaneme rozšířením předchozího o jeden znak. To jsou ale přesně ty stavy, kterými projde automat při zpracovávání řetězce ι od prvního znaku dál. Jedním průchodem automatu nad jehlou bez prvního písmenka se tím pádem rovnou dozvíme všechny údaje, které potřebujeme. Z předchozího pozorování plyne, že nikdy nebudeme potřebovat zpětnou hranu, kterou jsme ještě nezkonstruovali a jelikož víme, že jedno prohledání trvá lineárně s délkou toho, v čem hledáme, tak toto celé poběží v lineárním čase. Dostaneme tedy následující algoritmus:

Konstrukce zpětné funkce:

1. $Z[0] \leftarrow ?$, $Z[1] \leftarrow 0$.
2. $I \leftarrow 0$.
3. Pro $k = 2 \ldots J$:
4. $\quad I \leftarrow \mathit{Krok}(I, \iota[k])$.
5. $\quad Z[k] \leftarrow I$.

Začínáme tím, že nastavíme zpětnou hranu z prvních dvou stavů, přičemž $z[0]$ je nedefinované, protože tuto zpětnou hranu nikdy nepoužíváme. Dále postupně simulujeme výpočet automatu nad slovem bez prvního znaku a po každém kroku se dozvíme novou zpětnou hranu. *Krokem* automatu pak není nic jiného než vnitřek (3. a 4. bod) naší hledací procedury. To, kam jsme se dostali, pak zaznamenáme jako zpětnou funkci z k. Čili pouštíme automat na jehlu bez prvního písmenka, provedeme vždy jeden krok automatu (přes další písmenko jehly) a zapamatujeme si, jakou zpětnou funkci jsme zrovna dostali. Díky pozorováním navíc víme, že zpětné hrany konstruujeme správně, nikdy nepoužijeme zpětnou hranu, kterou jsme ještě nesestrojili a víme i to, že celou konstrukci zvládneme v lineárním čase s délkou jehly.

Věta: Algoritmus KMP najde všechny výskyty v čase $O(J + S)$.

Důkaz: Lineární čas s délkou jehly potřebujeme na postavení automatu, lineární čas s délkou sena pak potřebujeme na samotné vyhledání.

Rabinův-Karpův algoritmus

Nyní si ukážeme ještě jeden algoritmus na hledání jedné jehly, který nebude mít v nejhorším případě lineární složitost, ale bude ji mít průměrně. Bude daleko

jednodušší a ukáže se, že je v praxi daleko rychlejší. Bude to algoritmus založený na hashování.

Představme si, že máme seno délky S a jehlu délky J, a vezměme si nějakou hashovací funkci, které dáme na vstup J-tici znaků (tedy podslova dlouhá jako jehla). Tato hashovací funkce nám je pak zobrazí do množiny $\{0, \ldots, N-1\}$ pro nějaké dost velké N. Jak nám toto pomůže při hledání jehly? Vezmeme si libovolné „okénko“ délky J a než budeme zjišťovat, zda se v něm jehla vyskytuje, tak si spočítáme hashovací funkci a porovnáme ji s hashem jehly. Čili ptáme se, jestli je hash ze sena od nějaké pozice I do pozice $I+J$ roven hashi jehly – formálně: $h(\sigma[I : I+J]) = h(\iota)$. Teprve tehdy, když zjistíme, že se hodnota hashovací fce shoduje, začneme doopravdy porovnávat řetězce.

Není to ale nějaká hloupost? Může nám vůbec takováto konstrukce pomoci? Není to tak, že na spočítání hashovací funkce z J znaků, potřebujeme těch J znaků přečíst, což je stejně rychlé, jako rovnou řetězce porovnávat? Použijeme trik, který bude spočívat v tom, že si zvolíme šikovnou hashovací funkci. Uděláme to tak, abychom ji mohli při posunutí „okénka“ o jeden znak doprava v konstantním čase přepočítat. Chceme umět z $h(x_1 \ldots x_j)$ spočítat $h(x_2 \ldots x_{j+1})$. Na začátku si tedy spočítáme hash jehly a první J-tice znaků sena. Pak již jenom posouváme „okénko“ o jedna, přepočítáme hashovací funkci a když se shoduje s hashem jehly, tak porovnáme. Budeme přitom věřit tomu, že pokud se tam jehla nevyskytuje, pak máme hashovací funkci natolik rovnoměrnou, že pravděpodobnost toho, že se přesto strefíme do hashe jehly, je $1/N$. Jinými slovy jenom v jednom z řádově N případů budeme porovnávat falešně – tedy provedeme porovnání a vyjde nám, že výsledek je neshoda. V průměrném případě tedy můžeme stlačit složitost až téměř k lineární.

Podívejme se teď na průměrnou časovou složitost. Budeme určitě potřebovat čas na projití jehly a sena. Navíc strávíme nějaký čas nad falešnými porovnáními, kterých bude v průměru na každý N-tý znak sena jedno porovnání s jehlou – tedy SJ/N, přičemž N můžeme zvolit dost velké na to, abychom tento člen dostali pod nějakou rozumnou konstantu… Nakonec budeme potřebovat jedno porovnání na každý opravdový výskyt, čemuž se nevyhneme. Připočteme tedy ještě $J \cdot \sharp$*výskytů*. Dostáváme tedy: $\mathcal{O}(J + S + SJ/N + J \cdot \sharp$*výskytů*$)$.

Zbývá maličkost – totiž kde vzít hashovací funkci, která toto vše splňuje. Jednu si ukážeme. Bude to vlastně takový hezký polynom:

$$h(x_1 \ldots x_j) := \left(\sum_{I=1}^{J} x_I \cdot p^{J-I} \right) \bmod N.$$

Jinak zapsáno se tedy jedná o:

$$(x_1 \cdot p^{J-1} + x_2 \cdot p^{J-2} + \ldots + x_J \cdot p^0) \bmod N.$$

Po posunutí okénka o jedna chceme dostat:

$$(x_2 \cdot p^{J-1} + x_3 \cdot p^{J-2} + \ldots + x_J \cdot p^1 + x_{J+1} \cdot p^0) \bmod N.$$

Když se ale podíváme na členy těchto dvou polynomů, zjistíme, že se liší jen o málo. Původní polynom stačí přenásobit p, odečíst první člen s x_1 a naopak přičíst chybějící člen x_{J+1}. Dostáváme tedy:

$$h(x_2 \dots x_{J+1}) = (p \cdot h(x_1 \dots x_J) - x_1 \cdot p^J + x_{J+1}) \bmod N.$$

Přepočítání hashovací funkce tedy není nic jiného, než přenásobení té minulé p, odečtení nějakého násobku toho znaku, který vypadl z okénka, a přičtení toho znaku, o který se okénko posunulo. Pokud tedy máme k dispozici aritmetické operace v konstantním čase, zvládneme konstantně přepočítávat i hashovací funkci.

Tato hashovací funkce se dokonce nejen hezky počítá, ale dokonce se i opravdu „hezky“ chová (tedy „rozumně“ náhodně), pokud zvolíme vhodné p. To bychom měli zvolit tak, aby bylo rozhodně nesoudělné s N – tedy $NSD(p, N) = 1$. Aby se nám navíc dobře projevilo modulo obsažené v hashovací funkci, mělo by být p relativně velké (lze dopočítat, že optimum je mezi $2/3 \cdot N$ a $3/4 \cdot N$). S takto zvoleným p se tato hashovací funkce chová velmi příznivě a v praxi má celý algoritmus takřka lineární časovou složitost (průměrnou).

Hledání více řetězců najednou

Nyní si zahrajeme tutéž hru, ovšem v trochu složitějších kulisách. Podíváme se na algoritmus, který si poradí i s více než jednou jehlou. Mějme tedy jehly $\iota_1 \dots \iota_n$, a jejich délky $J_i = |\iota_i|$. Dále budeme potřebovat seno σ délky $S = |\sigma|$.

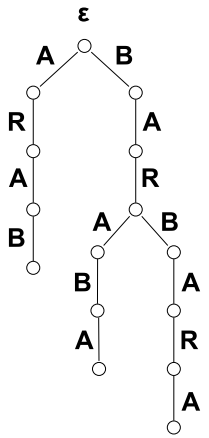
Předtím, než se pustíme do vlastního vyhledávacího algoritmu, možná bychom si měli ujasnit, co vlastně bude jeho výstupem. U problému hledání jedné jehly to bylo jasné – byla to nějaká množina pozic v seně, na kterých začínaly výskyty jehly. Jak tomu ale bude zde? Sice bychom také mohli vrátit pouze množinu pozic, ale my budeme chtít maličko víc. Budeme totiž chtít vědět i to, která jehla se na které pozici vyskytuje. Výstup tedy bude vypadat následovně: $V = \{(i, j) \mid \sigma[i : i + J_j] = \iota_j\}$.

Zde se však skrývá jedna drobná zrada. Budeme se asi muset vzdát naděje, že najdeme algoritmus, jehož složitost je lineární v celkové délce všech jehel a sena. Výstup totiž může být delší než lineární. Může se nám klidně stát, že na jedné pozici v seně se bude vyskytovat více různých jehel – pokud bude jedna jehla prefixem jiné (což jsme nikde nezakázali), tak máme povinnost ohlásit oba výskyty. Vzhledem k tomu budeme hledat takový algoritmus, který bude lineární v délce vstupu plus délce výstupu, což je evidentně to nejlepší, čeho můžeme dosáhnout.

Algoritmus, který si nyní ukážeme, vymysleli někdy v roce 1975 pan Aho a paní Corasicková. Bude to takové zobecnění Knuthova-Morrisova-Prattova algoritmu.

Algoritmus Aho-Corasicková

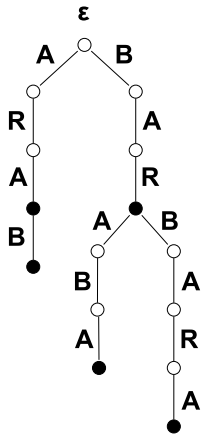
Opět se budeme snažit sestrojit nějaký vyhledávací automat a nějakým způsobem tento automat použít k procházení sena. Podívejme se nejprve na příklad. Budeme chtít vyhledávat tato slova: `ara`, `bar`, `arab`, `baraba`, `barbara`. Mějme tedy těchto pět jehel a rozmysleme si, jak by vypadal nějaký automat, který by tato slova



Vyhledávací automat – strom.

uměl zatím jenom rozpoznávat. Pro jedno slovo automat vypadal jako cesta, zde už to bude strom. (viz obrázek).

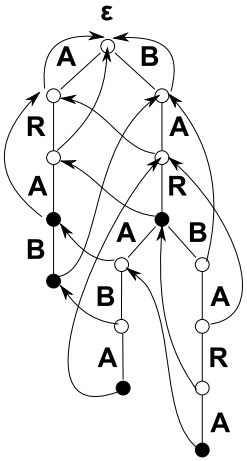
Navíc budeme muset do automatu zanést, kde nějaké slovo končí. V původním automatu pro jedno slovo to bylo jednoduché – ono jedno jediné slovo odpovídalo poslednímu vrcholu cesty. Tady se však slova mohou vyskytovat vícekrát a končit nejenom v listech ale i v nějakém vnitřním vrcholu (což se stane tehdy, pokud je jedno hledané slovo prefixem jiného hledaného slova). Formálně to nebudeme dokazovat, ale snadno nahlédneme, že listy stromu odpovídají hledaným slovům, ale opačně to neplatí.



Vyhledávací automat s konci slov.

Dále bychom měli do automatu přidat zpětné hrany. Jejich definice bude úplně stejná jako u automatu pro hledání jednoho slova. Jinými slovy z každého stavu půjde

zpětná hrana do nejdelšího vlastního suffixu, který je stavem. Čili když budeme mít nějaké jméno stavu, budeme se ho snažit co nejméně (ale alespoň o znak) zkrátit zleva, abychom zase dostali jméno stavu. Z kořene – prázdného stavu – pak evidentně žádná zpětná hrana nepovede.



Vyhledávací automat se zpětnými hranami.

Zbývá nám ještě si rozmyslet, jakým způsobem bude náš automat hlásit výstup. Opět směřujeme k tomu, aby se automat po přečtení nějakého kusu textu nacházel ve stavu odpovídajícímu nejdelšímu možnému suffixu toho textu. Zatímco u hledání jediné jehly bylo hlášení výskytů jednoduché – kdykoliv jsme se dostali na konec „automatové cestičky“ tady to bude opět složitější.

První, co se nabízí, je využít toho, že jsme si označili nějaké vrcholy, kde hledaná slova končí. Co tedy zkusit hlásit výskyt tohoto slova vždy, když přijdeme do nějakého označeného vrcholu? Tento způsob však nefunguje, pokud se uvnitř některé jehly skrývá jehla vnořená. Například po přečtení slova `bara`, nám náš současný automat neříká, že bychom měli nějaké slovo ohlásit, a přitom tam evidentě končí podřetězec `ara`. Stejně tak pokud přečteme `barbara`, už si nevšimneme toho, že tam končí zároveň i `ara`. Pouhé „hlášení teček“ tedy nefunguje.

Dále si můžeme všimnout toho, že všechna slova, která by se měla v daném stavu hlásit, jsou suffixy jména tohoto stavu. Přitom víme, že zpětná hrana jméno stavu zkracuje zleva. Takže speciálně všechny suffixy daného stavu, které jsou také stavy, se dají najít tak, že se vydáme po zpětných hranách do kořene. Nabízí se tedy vždy projít cestu po zpětných hranách až do kořene a hlásit všechny „tečky“. Tento způsob by nám však celý algoritmus značně zpomalil, protože cesta do kořene může být relativně dlouhá, ale teček na ní obvykle bude málo.

Mohli bychom také zkusit si pro každý stav β předpočítat množinu $cache(\beta)$, která by obsahovala všechna slova, která máme hlásit, když se ve stavu β nacházíme. Pokud pak do tohoto stavu vstoupíme, podíváme se na tuto množinu a vypíšeme

vše, co v ní je. Výpis nám bude evidentně trvat lineárně k velikosti množiny, celkově tedy lineárně k velikosti výstupu. Problém je ale ten, že jednotlivé cache mohou být hodně velké, takže je nestihneme sestrojit v lineárním čase. (Rozmyslete si příklad slovníku, kdy se to stane.)

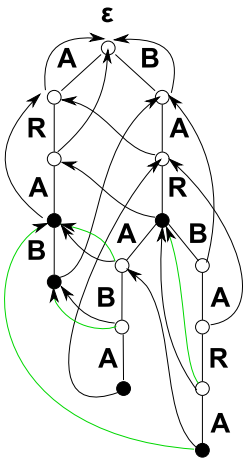
To, co nám ale již opravdu pomůže, bude zavedení zkratek. Všimli jsme si, že po zpětných hranách můžeme projít do kořene a hlásit všechny nalezené tečky. Vadilo nám ale, že se může stát, že budeme dlouho po cestě chodit a při tom žádné tečky nenalézat. Zavedeme si proto zkratky k nejbližší tečce.

Definice (zkratková hrana): Budeme mít tedy nějakou funkci $slovo(\beta) :=$ slovo, které končí ve stavu β (nebo $\emptyset$, pokud žádné takové slovo není). Dále pak funkci $out(\beta) :=$ nejbližší vrchol dosažitelný po zpětných hranách, čili nejdelší vlastní suffix stavu β, v němž je definovaná funkce *slovo*. Trochu lidštěji řečeno, ten nejbližší dosažitelný vrchol, ve kterém je tečka.

Po přidání těchto zkratkových hran již máme reprezentaci, ve které opravdu umíme v daném stavu vyjmenovat všechna slova, která máme vypsat, a to v čase lineárním s tím, kolik těch slov je.

Definice: Vyhledávací automat sestává ze stromu dopředných hran (vrcholy jsou prefixy jehel, hrany odpovídají rozšíření o písmenko), zpětných hran ($z(\beta) :=$ nejdelší vlastní suffix slova β, který je stavem) a zkratkových hran.

Automat pak bude na našem příkladu vypadat takto (zkratkové hrany jsou znázorněny zeleně):



Vyhledávací automat se zkratkovými hranami.

Nyní už nám zbývá jenom vlastní algoritmus – nejdřív popíšeme algoritmus, který bude hledat pomocí takového automatu, a potom se pustíme do toho, jak se takový automat staví.

Nejprve si nadefinujeme, jak vypadá jeden krok automatu. Bude to vlastně nějaká funkce, která dostane stav a písmenko. Ona nás pak pomocí tohoto písmenka

posune po automatu. ($f(\alpha, x)$ bude dopředná hrana ze stavu α označená písmenem x)

Krok (α, x):

1. Dokud $f(\alpha, x) = \emptyset$ & $\alpha \neq$ *kořen:* $\alpha \leftarrow z(\alpha)$.
2. Pokud $f(\alpha, x) \neq \emptyset$: $\alpha \leftarrow f(\alpha, x)$.
3. Vrátíme výsledek.

Hledání:

1. $\alpha \leftarrow$ *kořen*.
2. Pro znaky x ze slova σ:
3. $\alpha \leftarrow Krok(\alpha, x)$.
4. $\beta \leftarrow \alpha$
5. Dokud $\beta \neq \emptyset$:
6. Je-li $slovo(\beta) \neq \emptyset$:
7. Ohlásíme $slovo(\beta)$.
8. $\beta \leftarrow out(\beta)$.

Algoritmus hledání vlastně není nic jiného, než prosté projití po zelených zkratkových hranách ze stavu α, ve kterém právě jsme, a ohlášení všeho, co po cestě najdeme.

V každém okamžiku se automat nachází ve stavu, který odpovídá nejdelšímu možnému suffixu toho, co jsme už přečetli. Důkaz tohoto invariantu je stejný jako u verze automatu pro hledání pouze jedné jehly, neboť vychází pouze z definice zpětných hran. Podobně nahlédneme, že časová složitost vyhledávací procedury je lineární v délce sena plus to, co spotřebujeme na hlášení výskytů. Nejprve na chvíli zapomeneme, že nějaké výskyty hlásíme a spočítáme jenom kroky. Ty mohou vést dopředu a zpátky. Krok dopředu prodlužuje jméno stavu o jedna, krok dozadu zkracuje aspoň o jedna. Tudíž kroků dozadu je maximálně tolik, co kroků dopředu a kroků dopředu je maximálně tolik, kolik je délka sena. Všechny kroky dohromady tedy trvají $\mathcal{O}(S)$. Hlášení výskytů pak trvá $\mathcal{O}(S + |V|)$. Celé hledání tedy trvá lineárně v délce vstupu a výstupu.

Zbývá nám už jen konstrukce automatu. Opět využijeme faktu, že zpětná hrana ze stavu β vede tam, kam by se dostal automat při hledání β bez prvního písmenka. Takže zase chceme něco, jako simulovat výpočet toho automatu na slovech bez prvního písmenka a doufat v to, že si vystačíme s tou částí automatu, kterou jsme už postavili. Tentokrát to však nemůžeme dělat jedno slovo po druhém, protože zpětné hrany mohou vést křížem mezi jednotlivými větvemi automatu. Mohlo by se nám tedy stát, že při hledání nějakého slova potřebujeme zpětnou hranu, která vede do jiného slova, které jsme ještě nezkonstruovali. Takže tento postup selže. Můžeme však využít toho, že každá zpětná hrana vede ve stromu alespoň o jednu hladinu výš. Můžeme tak strom konstruovat po hladinách. Lze si to tedy představit tak, že paralelně spustíme vyhledávání všech slov bez prvních písmenek a vždycky uděláme

jeden podkrok každého z těch hledání, což nám dá zpětné hrany z dalšího patra stromu.

Konstrukce automatu:

1. Založíme prázdný strom, $r \leftarrow$ jeho kořen.
2. Vložíme do stromu slova $\iota_1 \ldots \iota_n$, nastavíme $slovo(*)$.
3. $z(r) \leftarrow \emptyset$, $out(r) \leftarrow \emptyset$.
4. Založíme frontu F a vložíme do ní syny kořene.
5. $\forall v \in F: \quad z(v) \leftarrow r, out(v) \leftarrow \emptyset$.
6. Dokud $F \neq \emptyset$:
7. Vybereme u z fronty F.
8. Pro všechny syny v vrcholu u:
9. $q \leftarrow Krok(z(u), \textit{písmeno na hraně } uv)$.
10. $z(v) \leftarrow q$.
11. Pokud $slovo(q) \neq \emptyset$, pak $out(v) \leftarrow q$.
12. Jinak $out(v) \leftarrow out(q)$.
13. Vložíme v do fronty F.

To, že tento algoritmus zkonstruuje zpětné hrany jak má, vyplývá z toho, že neděláme nic jiného, než že spouštíme výpočty po hladinách na všechna hledaná slova bez prvního písmenka. Stejně tak to, že doběhne v lineárním čase, je taktéž důsledkem toho, že efektivně spouštíme všechny tyto výpočty. Jen někdy uděláme najednou krok dvou či více výpočtů (například `araba` a `arbara` se počítají na začátku, dokud jsou stejné, jen jednou). Časová složitost této konstrukce je tedy menší nebo rovna součtu časových složitostí výpočtů nad všemi těmi slovy. To už ale víme, že je lineární v celkové délce těchto slov. Konstrukce automatu tedy trvá nejvýše tolik, co hledání všech ι_i, což je $\mathcal{O}(\sum_i \iota_i)$.

Věta: Algoritmus Aho-Corasicková najde všechny výskyty v čase

$$\mathcal{O}\left(\sum_i \iota_i \;+\; S \;+\; \sharp\textit{výskytů}\right).$$

Ještě se na závěr zamysleme, jak bychom si takový automat ukládali do paměti. Určitě se nám bude hodit si stavy nějak očíslovat (třeba v pořadí, v jakém budou vznikat). Potom funkce pro zpětné a zkratkové hrany mohou být reprezentované polem indexovaným číslem stavu. Funkce *Slovo*, která říká, jaké slovo ve stavu končí, zase může být pole indexované stavem, které nám řekne pořadové číslo slova ve slovníku. Pro dopředné hrany v každém vrcholu pak můžeme mít pole indexované písmenky abecedy, které nám pro každé písmenko řekne, buď že taková hrana není, nebo nám řekne, kam tato hrana vede. Je vidět, že takovéto pole se hodí pro poměrně malé abecedy. Už pro abecedu A-Z bude velikosti 26 a z většiny bude prázdné, takže bychom plýtvali pamětí. V praxi se proto často používá hashovací tabulka. Případně bychom mohli mít i jen jednu velkou společnou hashovací tabulku, která bude

reprezentovat funkci celou, ve které budou zahashované dvojice (stav, písmenko). Těchto dvojic je evidentně tolik, kolik hran stromu, čili lineárně s velikostí slovníku, a je to asi nejkompaktnější reprezentace.

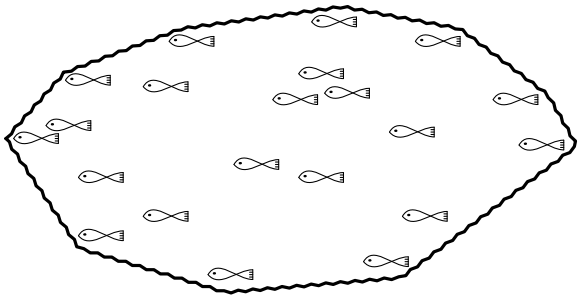
7. Geometrické algoritmy *(sepsal Pavel Klavík)*

Ukážeme si několik základních algoritmů na řešení geometrických problémů v rovině. Proč zrovna v rovině? Inu, jednorozměrné problémy bývají triviální a naopak pro vyšší dimenze jsou velice komplikované. Rovina je proto rozumným kompromisem mezi obtížností a zajímavostí.

Celou kapitolou nás bude provázet pohádka ze života ledních medvědů. Pokusíme se vyřešit jejich „každodenní" problémy ...

Hledání konvexního obalu

Daleko na severu žili lední medvědi. Ve vodách tamního moře byla hojnost ryb a jak je známo, ryby jsou oblíbenou pochoutkou ledních medvědů. Protože medvědi z naší pohádky rozhodně nejsou ledajací a ani chytrost jim neschází, rozhodli se všechny ryby pochytat. Znají přesná místa výskytu ryb a rádi by vyrobili obrovskou síť, do které by je všechny chytili. Pomozte medvědům zjistit, jaký nejmenší obvod taková síť může mít.



Problém ledních mědvědů: Jaký je nejmenší obvod sítě?

Neboli v řeči matematické, chceme pro zadanou množinu bodů v rovině nalézt její konvexní obal. Co je to konvexní obal? Množina bodů je *konvexní*, pokud pro každé dva body obsahuje i celou úsečku mezi nimi. *Konvexní obal* je nejmenší konvexní podmnožina roviny, která obsahuje všechny zadané body.⟨6⟩ Z algoritmického

⟨6⟩ Pamatujete si na lineární obaly ve vektorových prostorech? Lineární obal množiny vektorů je nejmenší vektorový podprostor, který tyto vektory obsahuje. Není náhoda, že tato definice připomíná definici konvexního obalu. Na druhou stranu každý vektor z lineárního obalu lze vyjádřit jako lineární kombinaci daných vektorů. Podobně platí i pro konvexní obaly, že každý bod z obalu je konvexní kombinací daných bodů. Ta se liší od lineární v tom, že všechny koeficienty jsou v intervalu $[0, 1]$ a navíc součet všech koeficientů je 1. Tento algebraický pohled může mnohé věci zjednodušit. Zkuste si dokázat, že obě definice konvexního obalu jsou ekvivalentní.

hlediska nás však bude zajímat jenom jeho hranice, kterou budeme dále označovat jako konvexní obal.

Naším úkolem je nalézt konvexní obal konečné množiny bodů. To je vždy konvexní mnohoúhelník, navíc s vrcholy v zadaných bodech. Řešením problému tedy bude posloupnost bodů, které tvoří konvexní obal. Pro malé množiny je konvexní obal nakreslen na obrázku, pro více bodů je však situace mnohem složitější.

$n = 1$ $n = 2$ $n = 3$ $n = 4$

Konvexní obaly malých množin.

Pro jednoduchost budeme předpokládat, že všechny body mají různé x-ové souřadnice. Tedy utřídění bodů zleva doprava je určené jednoznačně.⟨7⟩ Tím máme zajištěné, že existují dva body, nejlevější a nejpravější, pro které platí následující invariant:

Invariant: Nejlevější a nejpravější body jsou vždy v konvexním obalu.

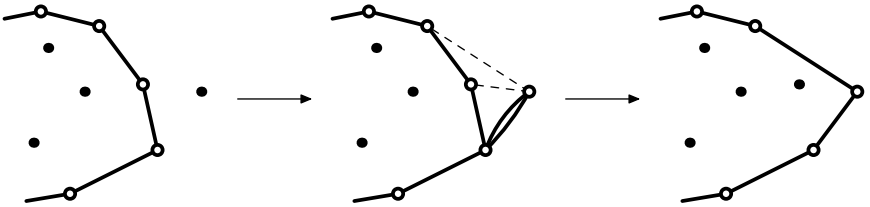
Algoritmus na nalezení konvexního obalu funguje na následujícím jednoduchém principu, kterému se někdy říká *zametání roviny*. Procházíme body zleva doprava a postupně rozšiřujeme doposud nalezený konvexní obal o další body. Na začátku bude konvexní obal jediného bodu samotný bod. Na konci k-tého kroku algoritmu známe konvexní obal prvních k bodů. Když algoritmus skončí, známe hledaný konvexní obal. Podle invariantu musíme v k-tém kroku přidat do obalu k-tý nejlevější bod. Zbývá si jen rozmyslet, jak přesně tento bod přidat.

Přidání dalšího bodu do konvexního obalu funguje, jak je naznačeno na obrázku. Podle invariantu víme, že bod nejvíc vpravo je součástí konvexního obalu. Za něj napojíme nově přidávaný bod. Tím jsme získali nějaký obal, ale zpravidla nebude konvexní. To lze však snadno napravit, stačí odebírat body, v obou směrech podél konvexního obalu, tak dlouho, dokud nezískáme konvexní obal. Na příkladu z obrázku nemusíme po směru hodinových ručiček odebrat ani jeden bod, obal je v pořádku. Naopak proti směru hodinových ručiček musíme odebrat dokonce dva body.

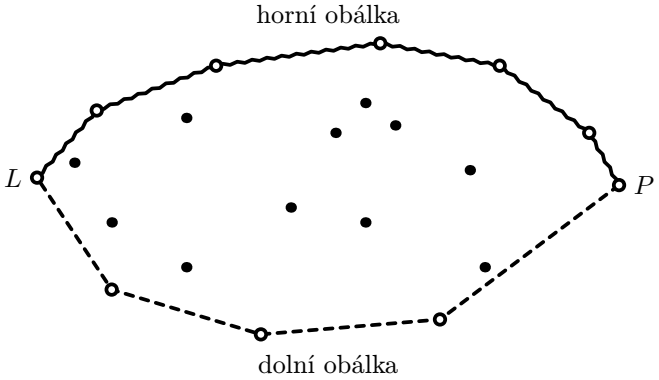
Pro případnou implementaci a rozbor složitosti si nyní popíšeme algoritmus detailněji. Aby se lépe popisoval, rozdělíme si konvexní obal na dvě části spojující nejlevější a nejpravější bod obalu. Budeme jim říkat *horní obálka* a *dolní obálka*.

Obě obálky jsou lomené čáry, navíc horní obálka pořád zatáčí doprava a dolní naopak doleva. Pro udržování bodů v obálkách stačí dva zásobníky. V k-tém kroku

⟨7⟩ To si můžeme dovolit předpokládat, neboť se všemi body stačí nepatrně pootočit. Tím konvexní obal určitě nezměníme. Avšak jednodušší řešení je naprogramovat třídění lexikograficky (druhotně podle souřadnice y) a vyřadit identické body.



Přidání bodu do konvexního obalu.



Horní a dolní obálka konvexního obalu.

algoritmu přidáme zvlášť k-tý bod do horní i dolní obálky. Přidáním k-tého bodu se však může porušit směr, ve kterém obálka zatáčí. Proto budeme nejprve body z obálky odebírat a k-tý bod přidáme až ve chvíli, kdy jeho přidání směr zatáčení neporuší.

Algoritmus:

1. Setřídíme body podle x-ové souřadnice, označme body $b_1, \ldots, b_n$.
2. Vložíme do horní a dolní obálky bod b_1: $H = D = (b_1)$.
3. Pro každý další bod $b = b_2, \ldots, b_n$:
4. Přepočítáme horní obálku:
5. Dokud $|H| \geq 2$, $H = (\ldots, h_{k-1}, h_k)$ a úhel $h_{k-1}h_kb$ je orientovaný doleva:
6. Odebereme poslední bod h_k z obálky H.
7. Přidáme bod b do obálky H.
8. Symetricky přepočteme dolní obálku (s orientací doprava).
9. Výsledný obal je tvořen body v obálkách H a D.

Rozebereme si časovou složitost algoritmu. Setřídit body podle x-ové souřadnice dokážeme v čase $\mathcal{O}(n \log n)$. Přidání dalšího bodu do obálek trvá lineárně vzhledem k počtu odebraných bodů. Zde využijeme obvyklý postup: Každý bod je odebrán nejvýše jednou, a tedy všechna odebrání trvají dohromady $\mathcal{O}(n)$. Konvexní

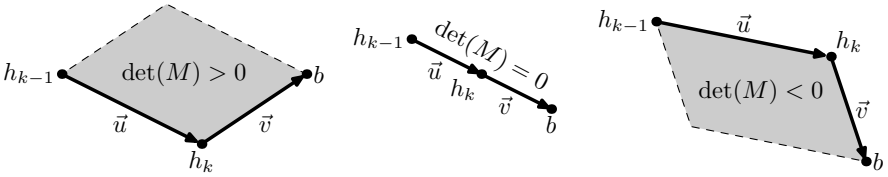
obal dokážeme sestrojit v čase $\mathcal{O}(n \log n)$, a pokud bychom měli seznam bodů již utříděný, dokážeme to dokonce v $\mathcal{O}(n)$.

Algebraický dodatek: Existuje jednoduchý postup, jak zjistit orientaci úhlu? Ukážeme si jeden založený na lineární algebře. Budou se hodit vlastnosti determinantu. Absolutní hodnota determinantu je objem rovnoběžnostěnu určeného řádkovými vektory matice. Důležitější však je, že znaménko determinantu určuje „orientaci" vektorů, zda je levotočivá či pravotočivá. Protože náš problém je rovinný, budeme uvažovat determinanty matic 2×2.

Uvažme souřadnicový systém v rovině, kde x-ová souřadnice roste směrem doprava a y-ová směrem nahoru. Chceme zjistit orientaci úhlu $h_{k-1}h_kb$. Položme $\vec{u} = (x_1, y_1)$ jako rozdíl souřadnic h_k a h_{k-1} a podobně $\vec{v} = (x_2, y_2)$ je rozdíl souřadnic b a h_k. Matice M je definována následovně:

$$M = \begin{pmatrix} \vec{u} \\ \vec{v} \end{pmatrix} = \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \end{pmatrix}.$$

Úhel $h_{k-1}h_kb$ je orientován doleva, právě když $\det M = x_1y_2 - x_2y_1$ je nezáporný,[8] a spočítat hodnotu determinantu je jednoduché. Možné situace jsou nakresleny na obrázku. Poznamenejme, že k podobnému vzorci se lze také dostat přes vektorový součin vektorů $\vec{u}$ a $\vec{v}$.



Jak vypadají determinanty různých znamének v rovině.

Šlo by to vyřešit rychleji? Také vám vrtá hlavou, zda existují rychlejší algoritmy? Na závěr si ukážeme něco, co na přednášce nebylo.[9] Nejrychlejší známý algoritmus, jehož autorem je T. Chan, funguje v čase $\mathcal{O}(n \log h)$, kde h je počet bodů ležících na konvexním obalu, a přitom je překvapivě jednoduchý. Zde si naznačíme, jak tento algoritmus funguje.

Algoritmus přichází s následující myšlenkou. Předpokládejme, že bychom znali velikost konvexního obalu h. Rozdělíme body libovolně do $\lceil \frac{n}{h} \rceil$ množin $Q_1, \ldots, Q_k$ tak, že $|Q_i| \leq h$. Pro každou z těchto množin nalezneme konvexní obal pomocí výše popsaného algoritmu. To dokážeme pro jednu v čase $\mathcal{O}(h \log h)$ a pro všechny v čase $\mathcal{O}(n \log h)$. V druhé fázi spustíme hledání konvexního obalu pomocí provázkového

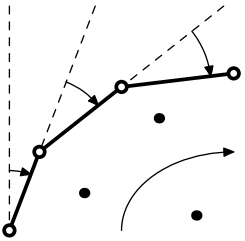
[8] Neboli vektory $\vec{u}$ a $\vec{v}$ odpovídají roztažení a zkosení vektorů báze $\vec{x} = (1, 0)$ a $\vec{y} = (0, 1)$, pro něž je determinant nezáporný.

[9] A také se nebude zkoušet.

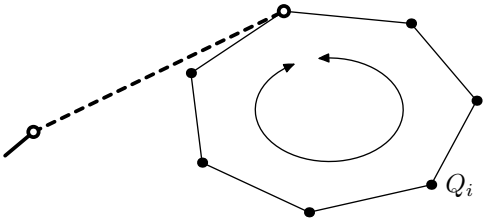
algoritmu a pro zrychlení použijeme předpočítané obaly menších množin. Nejprve popíšeme jeho myšlenku. Použijeme následující pozorování:

Pozorování: Úsečka spojující dva body a a b leží na konvexním obalu, právě když všechny ostatní body leží pouze na jedné její straně.⟨10⟩

Algoritmu se říká *provázkový*, protože svojí činností připomíná namotávání provázku podél konvexního obalu. Začneme s bodem, který na konvexním obalu určitě leží, to je třeba ten nejlevější. V každém kroku nalezneme následující bod po obvodu konvexního obalu. To uděláme například tak, že projdeme všechny body a vybereme ten, který svírá nejmenší úhel s poslední stranou konvexního obalu. Nově přidaná úsečka vyhovuje pozorování a proto do konvexního obalu patří. Po h krocích se dostaneme zpět k nejlevějšímu bodu a výpočet ukončíme. V každém kroku potřebujeme projít všechny body a vybrat následníka, což dokážeme v čase $\mathcal{O}(n)$. Celková složitost algoritmu je tedy $\mathcal{O}(n \cdot h)$.



Provázkový algoritmus.



Hledání kandidáta v předpočítaném obalu.

Provázkový algoritmus funguje, ale má jednu obrovskou nevýhodu – je totiž ukrutně pomalý. Kýženého zrychlení dosáhneme, pokud použijeme předpočítané konvexní obaly. Ty umožní rychleji hledat následníka. Pro každou z množin Q_i najdeme zvlášť kandidáta a poté z nich vybereme toho nejlepšího. Možný kandidát vždy leží na konvexním obalu množiny Q_i. Využijeme toho, že body obalu jsou „uspořádané“, i když trochu netypicky do kruhu. Kandidáta můžeme hledat metodou půlení intervalu, i když detaily jsou maličko složitější než je obvyklé. Jak půlit zjistíme podle směru zatáčení konvexního obalu. Detaily si rozmyslí čtenář sám.

Časová složitost půlení je $\mathcal{O}(\log h)$ pro jednu množinu. Množin je nejvýše $\mathcal{O}(\frac{n}{h})$, tedy následující bod konvexního obalu nalezneme v čase $\mathcal{O}(\frac{n}{h} \log h)$. Celý obal nalezneme ve slibovaném čase $\mathcal{O}(n \log h)$.

Popsanému algoritmu schází jedna důležitá věc: Ve skutečnosti většinou neznáme velikost h. Budeme proto algoritmus iterovat s rostoucí hodnotou h, dokud konvexní obal nesestrojíme. Pokud při slepování konvexních obalů zjistíme, že konvexní obal je větší než h, výpočet ukončíme. Zbývá ještě zvolit, jak rychle má h růst. Pokud by rostlo moc pomalu, budeme počítat zbytečně mnoho fází, naopak při rychlém růstu by nás poslední fáze mohla stát příliš mnoho.

⟨10⟩ Formálně je podmínka následující: Přímka ab určuje dvě poloroviny. Úsečka leží na konvexním obalu, právě když všechny body leží v jedné z polorovin.

V k-té iteraci položíme $h = 2^{2^k}$. Dostáváme celkovou složitost algoritmu:

$$\sum_{m=0}^{\mathcal{O}(\log\log h)} \mathcal{O}(n \log 2^{2^m}) = \sum_{m=0}^{\mathcal{O}(\log\log h)} \mathcal{O}(n \cdot 2^m) = \mathcal{O}(n \log h),$$

kde poslední rovnost dostaneme jako součet prvních $\mathcal{O}(\log\log h)$ členů geometrické řady $\sum 2^m$.

8. Geometrie vrací úder

(sepsal Pavel Klavík)

Když s geometrickými problémy pořádně nezametete, ony vám to vrátí! Ale když už zametat, tak určitě ne pod koberec a místo smetáku použijte přímku. V této přednášce nás spolu s dvěma geometrickými problémy samozřejmě čeká pokračování pohádky o ledních medvědech.

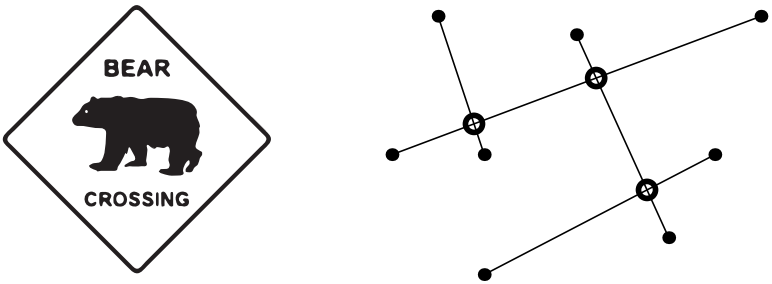
Medvědi vyřešili rybí problém a hlad je již netrápí. Avšak na severu nežijí sami, za sousedy mají Eskymáky. Protože je rozhodně lepší se sousedy dobře vycházet, jsou medvědi a Eskymáci velcí přátelé. Skoro každý se se svými přáteli rád schází. Avšak to je musí nejprve nalézt ...

Hledání průsečíků úseček

Zkusíme nejprve Eskymákům vyřešit lokalizaci ledních medvědů.

Když takový medvěd nemá co na práci, rád se prochází. Na místech, kde se trasy protínají, je zvýšená šance, že se dva medvědi potkají a zapovídají – ostatně co byste čekali od medvědů. To jsou ta správná místa pro Eskymáka, který chce potkat medvěda. Jenomže jak tato křížení najít?

Pro zjednodušení předpokládejme, že medvědi chodí po úsečkách tam a zpět. Budeme tedy chtít nalézt všechny průsečíky úseček v rovině.



Problém Eskymáků: Kde všude se kříží medvědí trasy?

Pro n úseček může existovat až $\Omega(n^2)$ průsečíků.⟨11⟩ Tedy optimální složitosti by dosáhl i algoritmus, který by pro každou dvojici úseček testoval, zda se protínají. Časovou složitost algoritmu však posuzujeme i vzhledem k velikosti výstupu p.

⟨11⟩ Zkuste takový příklad zkonstruovat.

Typické rozmístění úseček mívá totiž průsečíků spíše pomálu. Pro tento případ si ukážeme podstatně rychlejší algoritmus.

Pro jednodušší popis předpokládejme, že úsečky leží v obecné poloze. To znamená, že žádné tři úsečky se neprotínají v jednom bodě a průnikem každých dvou úseček je nejvýše jeden bod. Navíc předpokládejme, že krajní bod žádné úsečky neleží na jiné úsečce a také neexistují vodorovné úsečky. Na závěr si ukážeme, jak se s těmito případy vypořádat.

Algoritmus funguje na principu zametání roviny, popsaném v minulé přednášce. Budeme posouvat vodorovnou přímku odshora dolů. Vždy, když narazíme na nový průnik, ohlásíme jeho výskyt. Samozřejmě spojité posouvání nahradíme diskrétním a přímku vždy posuneme do dalšího zajímavého bodu.

Zajímavé události jsou *začátky úseček*, *konce úseček* a *průsečíky úseček*. Po utřídění známe pro první dva typy událostí pořadí, v jakém se objeví. Výskyty průsečíků budeme počítat průběžně, jinak bychom celý problém nemuseli řešit.

V každém kroku si pamatujeme *průřez* P – posloupnost úseček aktuálně protnutých zametací přímkou. Tyto úsečky máme utříděné zleva doprava. Navíc si udržujeme kalendář K budoucích událostí. Z hlediska průsečíků budeme na úsečky nahlížet jako na polopřímky. Pro sousední dvojice úseček si udržujeme, zda se jejich směry někde protnou. Algoritmus pro hledání průniků úseček funguje následovně:

Algoritmus:

1. $P \leftarrow \emptyset$.
2. Do K vložíme začátky a konce všech úseček.
3. Dokud $K \neq \emptyset$:
4. Odebereme nejvyšší událost.
5. Pokud je to začátek úsečky, zatřídíme novou úsečku do P.
6. Pokud je to konec úsečky, odebereme úsečku z P.
7. Pokud je to průsečík, nahlásíme ho a prohodíme úsečky v P.
8. Navíc vždy přepočítáme průsečíkové události, vždy maximálně dvě odebereme a dvě nové přidáme.

Zbývá rozmyslet si, jaké datové struktury použijeme, abychom průsečíky nalezli dostatečně rychle. Pro kalendář použijeme například haldu. Průřez si budeme udržovat ve vyhledávacím stromě. Poznamenejme, že nemusíme znát souřadnice úseček, stačí znát jejich pořadí, které se mezi jednotlivými událostmi nemění. Při přidávání úseček procházíme stromem a porovnáváme souřadnice v průřezu, které průběžně dopočítáváme.

Kalendář obsahuje vždy nejvýše $\mathcal{O}(n)$ událostí. Podobně průřez obsahuje v každém okamžiku nejvýše $\mathcal{O}(n)$ úseček. Jednu událost kalendáře dokážeme ošetřit v čase $\mathcal{O}(\log n)$. Všech událostí je $\mathcal{O}(n + p)$, a tedy celková složitost algoritmu je $\mathcal{O}((n + p) \log n)$.

Slíbili jsme, že popíšeme, jak se vypořádat s výše uvedenými podmínkami na vstup. Události kalendáře se stejnou y-ovou souřadnicí budeme třídit v pořadí začátky, průsečíky a konce úseček. Tím nahlásíme i průsečíky krajů úseček a ani vodorovné

úsečky nebudou vadit. Podobně se není třeba obávat průsečíků více úseček v jednom bodě. Úsečky jdoucí stejným směrem, jejichž průnik je úsečka, jsou komplikovanější, ale lze jejich průsečíky ošetřit a vypsat třeba souřadnice úsečky tvořící jejich průnik.

Na závěr poznamenejme, že Balaban vymyslel efektivnější algoritmus, který funguje v čase $\mathcal{O}(n \log n + p)$, ale je podstatně komplikovanější.

Hledání nejbližších bodů a Voroného diagramy

Nyní se pokusíme vyřešit i problém druhé strany – pomůžeme medvědům nalézt Eskymáky.

Eskymáci tráví většinu času doma, ve svém iglů. Takový medvěd je na své toulce zasněženou krajinou, když tu se najednou rozhodne navštívit nějakého Eskymáka. Proto se podívá do své medvědí mapy a nalezne nejbližší iglů. Má to ale jeden háček, iglů jsou spousty a medvěd by dávno usnul, než by nejbližší objevil.⟨12⟩

Popíšeme si nejprve, jak vypadá medvědí mapa. Medvědí mapa obsahuje celou Arktidu a jsou v ní vyznačena všechna iglů. Navíc obsahuje vyznačené oblasti tvořené body, které jsou nejblíže k jednomu danému iglů. Takovému schématu se říká *Voroného diagram*. Ten pro zadané body $x_1, \ldots, x_n$ obsahuje rozdělení roviny na oblasti $B_1, \ldots, B_n$, kde B_i je množina bodů, které jsou blíže k x_i než k ostatním bodům x_j. Formálně jsou tyto oblasti definovány následovně:

$$B_i = \left\{ y \in \mathbb{R}^2 \mid \forall j : \rho(x_i, y) \leq \rho(x_j, y) \right\},$$

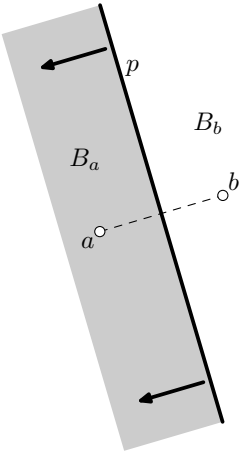
kde $\rho(x, y)$ značí vzdálenost bodů x a y.

Ukážeme si, že Voroného diagram má překvapivě jednoduchou strukturu. Nejprve uvažme, jak budou vypadat oblasti B_a a B_b pouze pro dva body a a b, jak je naznačeno na obrázku. Všechny body stejně vzdálené od a i b leží na přímce p – ose úsečky ab. Oblasti B_a a B_b jsou tedy tvořeny polorovinami ohraničenými osou p. Tedy obecně tvoří množina všech bodů bližších k x_i než k x_j nějakou polorovinu. Oblast B_i obsahuje všechny body, které jsou současně bližší k x_i než ke všem ostatním bodům x_j – tedy leží ve všech polorovinách současně. Každá z oblastí B_i je tvořena průnikem $n-1$ polorovin, tedy je to (možná neomezený) mnohoúhelník.⟨13⟩

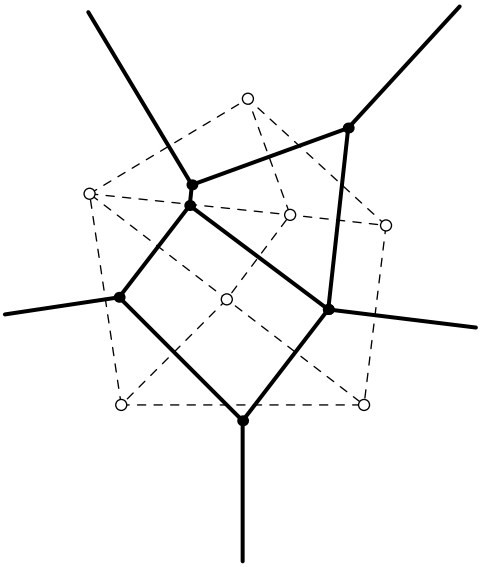
⟨12⟩ Zlí jazykové by řekli, že medvědi jsou moc líní a nebo v mapách ani číst neumí!

⟨13⟩ Slyšeli jste už o lineárním programování? Jak název vůbec nenapoví, *lineární programování* je teorii zabývající se řešením a vlastnostmi soustav lineárních nerovnic. Lineární program je popsaný lineární funkcí, kterou chceme maximalizovat za podmínek popsaných soustavou lineárních nerovnic. Každá nerovnice určuje poloprostor, ve kterém se přípustná řešení nachází. Protože přípustné řešení splňuje všechny nerovnice zároveň, je množina všech přípustných řešení (možná neomezený) mnohostěn, obecně ve veliké dimenzi $\mathbb{R}^d$, kde d je počet proměnných. Množiny B_i lze snadno popsat jako množiny všech přípustných řešení lineárních programů pomocí výše ukázaných polorovin. Na závěr poznamenejme, že dlouho otevřená otázka, zda lze nalézt optimální řešení lineárního programu v polynomiálním čase, byla pozitiv-

Příklad Voroného diagramu je naznačen na obrázku. Zadané body jsou označeny prázdnými kroužky a hranice oblastí B_i jsou vyznačené černými čárami.



Body bližší k a než b.



Voroného diagram.

Není náhoda, pokud vám hranice oblastí připomíná rovinný graf. Jeho vrcholy jsou body, které jsou stejně vzdálené od alespoň tří zadaných bodů. Jeho stěny jsou oblasti B_i. Jeho hrany jsou tvořeny částí hranice mezi dvěma oblastmi – body, které mají dvě oblasti společné. Obecně průnik dvou oblastí může být, v závislosti na jejich sousedění, prázdný, bod, úsečka, polopřímka nebo dokonce celá přímka. V dalším textu si představme, že celý Voroného diagram uzavřeme do dostatečně velkého obdélníka,⟨14⟩ čímž dostaneme omezený rovinný graf.

Poznamenejme, že přerušované čáry tvoří hrany duálního rovinného grafu s vrcholy v zadaných bodech. Hrany spojují sousední body na kružnicích, které obsahují alespoň tři ze zadaných bodů. Například na obrázku dostáváme skoro samé trojúhelníky, protože většina kružnic obsahuje přesně tři zadané body. Avšak nalezneme i jeden čtyřúhelník, jehož vrcholy leží na jedné kružnici.

Zkusíme nyní odhadnout, jak velký je rovinný graf popisující Voroného diagram. Podle slavné Eulerovy formule má každý rovinný graf nejvýše lineárně mnoho

ně vyřešena – je znám polynomiální algoritmus, kterému se říká *metoda vnitřního bodu*. Na druhou stranu, pokud chceme najít přípustné celočíselné řešení, je úloha NP-úplná a je jednoduché na ni převést spoustu optimalizačních problémů. Dokázat NP-těžkost není příliš těžké. Na druhou stranu ukázat, že tento problém leží v NP, není vůbec jednoduché.

⟨14⟩ Přeci jenom i celá Arktida je omezeně velká.

vrcholů, hran a stěn – pro v vrcholů, e hran a f stěn je $e \leq 3v-6$ a navíc $v+f = e+2$. Tedy složitost diagramu je lineární vzhledem k počtu zadaných bodů $n = f$, $\mathcal{O}(n)$. Navíc Voroného diagram lze zkonstruovat v čase $\mathcal{O}(n \log n)$, například pomocí zametání roviny nebo metodou rozděl a panuj. Tím se však zabývat nebudeme,⟨15⟩ místo toho si ukážeme, jak v již spočteném Voroného diagramu rychle hledat nejbližší body.

Lokalizace bodu uvnitř mnohoúhelníkové sítě

Problém medvědů je najít v medvědí mapě co nejrychleji nejbližší iglů. Máme v rovině síť tvořenou mnohoúhelníky. Chceme pro jednotlivé body rychle rozhodovat, do kterého mnohoúhelníku patří. Naše řešení budeme optimalizovat pro jeden pevný rozklad a obrovské množství různých dotazů, které chceme co nejrychleji zodpovědět.⟨16⟩ Nejprve předzpracujeme zadané mnohoúhelníky a vytvoříme strukturu, která nám umožní rychlé dotazy na jednotlivé body.

Ukažme si pro začátek řešení bez předzpracování. Rovinu budeme zametat přímkou shora dolů. Podobně jako při hledání průsečíků úseček, udržujeme si průřez přímkou. Všimněte si, že tento průřez se mění jenom ve vrcholech mnohoúhelníků. Ve chvíli, kdy narazíme na hledaný bod, podíváme se, do kterého intervalu v průřezu patří. To nám dá mnohoúhelník, který nahlásíme. Průřez budeme uchovávat ve vyhledávacím stromě. Takové řešení má složitost $\mathcal{O}(n \log n)$ na dotaz, což je hrozně pomalé.

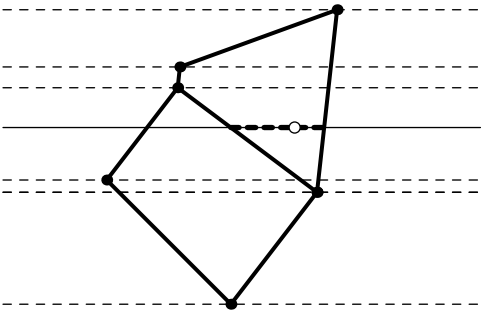
Předzpracování bude fungovat následovně. Jak je naznačeno na obrázku přerušovanými čárami, rozřežeme si celou rovinu na pásy, během kterých se průřez přímkou nemění. Pro každý z nich si pamatujeme stav stromu popisující, jak vypadal průřez při procházení tímto pásem. Když chceme lokalizovat nějaký bod, nejprve půlením nalezneme pás, ve kterém se nachází. Poté položíme dotaz na příslušný strom. Strom procházíme a po cestě si dopočítáme souřadnice průřezu, až lokalizujeme správný interval v průřezu. Dotaz dokážeme zodpovědět v čase $\mathcal{O}(\log n)$. Hledaný bod je na obrázku naznačen prázdným kolečkem a nalezený interval v průřezu je vytažený tučně.

Jenomže naše řešení má jeden háček: Jak zkonstruovat jednotlivé verze stromu dostatečně rychle? K tomu napomohou *částečně perzistentní* datové struktury. Pod perzistencí se myslí, že struktura umožňuje uchovávat svoji historii. Částečně perzistentní struktury nemohou svoji historii modifikovat.

Popíšeme si, jak vytvořit perzistentní strom s pamětí $\mathcal{O}(\log n)$ na změnu. Pokud provádíme operaci na stromě, mění se jenom malá část stromu. Například při vkládání do stromu se mění jenom prvky na jedné cestičce z kořene do listu (a

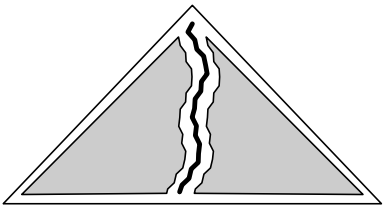
⟨15⟩ Pro zvídavé, kteří nemají zkoušku druhý den ráno: Detaily naleznete v zápiscích z předloňského ADSka.

⟨16⟩ Představujme si to třeba tak, že medvědům zprovozníme server. Ten jednou schroustá celou mapu a potom co nejrychleji odpovídá na jejich dotazy. Medvědi tak nemusí v mapách nic hledat, stačí se připojit na server a počkat na odpověď.



Mnohoúhelníky rozřezané na pásy.

případně rotací i na jejím nejbližším okolí). Proto si uložíme upravenou cestičku a zbytek stromu budeme sdílet s předchozí verzí. Na obrázku je vyznačena cesta, jejíž vrcholy jsou upravovány. Šedě označené podstromy navěšené na tuto cestu se nemění, a proto na ně stačí zkopírovat ukazatele. Mimochodem změny každé operace se složitostí $\mathcal{O}(k)$ lze zapsat v paměti $\mathcal{O}(k)$, prostě operace nemá tolik času, aby mohla pozměnit příliš velikou část stromu.

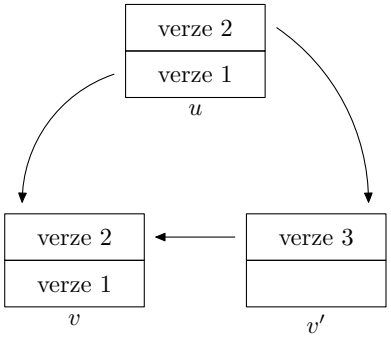


Jedna operace mění pouze okolí cesty – navěšené podstromy se nemění.

Celková časová složitost je tedy $\mathcal{O}(n \log n)$ na předzpracování Voroného diagramu a vytvoření persistentního stromu. Kvůli persistenci potřebuje toto předzpracování paměť $\mathcal{O}(n \log n)$. Na dotaz spotřebujeme čas $\mathcal{O}(\log n)$, neboť nejprve vyhledáme půlením příslušný pás a poté položíme dotaz na příslušnou verzi stromu. Rychleji to ani provést nepůjde, neboť potřebujeme utřídit souřadnice bodů.

Lze to lépe? Na závěr poznamenejme, že se umí provést výše popsaná persistence vyhledávacího stromu v amortizované paměti $\mathcal{O}(1)$ na změnu. Ve stručnosti naznačíme myšlenku. Použijeme stromy, které při insertu a deletu provádí amortizovaně jenom konstantně mnoho úprav své struktury. To nám například zaručí 2-4 stromy z přednášky a podobnou vlastnost lze dokázat i o červeno-černých stromech. Při změně potom nebudeme upravovat celou cestu, ale upravíme jenom jednotlivé vrcholy, kterých se změna týká. Každý vrchol stromu si v sobě bude pamatovat až dvě své verze. Pokud chceme vytvořit třetí verzi, vrchol zkopírujeme stranou. To však může vyvolat změny v jeho rodičích až do kořene. Situace je naznačena na obrázku. Při vytvoření nové verze 3 pro vrcholu v vytvoříme jeho kopii v', do které uložíme tuto verzi. Avšak musíme také změnit rodiče u, kterému vytvoříme novou verzi ukazující

na v'. Abychom dosáhli kýžené konstantní paměťové složitosti, pomůže potenciálový argument – změn se provádí amortizovaně jenom konstantně mnoho. Navíc si pro každou verzi pamatujeme její kořen, ze kterého máme dotaz spustit.



Vytvoření nové verze vrcholu.

9. Fourierova transformace

(K.Jakubec, M.Polák a G.Ocsovszky, V.Tůma, M.Kozák)

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

Celé polynomy označujeme velkými písmeny, jednotlivé členy polynomů příslušnými malými písmeny (př.: polynom W stupně d má koeficienty $w_0, w_1, w_2, \ldots, w_d$).

Libovolný polynom P stupně (nejvýše) d lze reprezentovat jednak jeho koeficienty, tedy čísly $p_0, p_1, \ldots, p_d$, druhak i pomocí hodnot:

Lemma: Polynom stupně nejvýše d je jednoznačně určeň svými hodnotami v $d+1$ různých bodech.

Důkaz: Polynom stupně d má maximálně d kořenů (indukcí – je-li k kořenem P, pak lze P napsat jako $(x-k)Q$ kde Q je polynom stupně o jedna menší, přitom polynom stupně 1 má jediný kořen); uvážíme-li dva různé polynomy P a Q stupně d nabývající v daných bodech stejných hodnot, tak $P-Q$ je polynom stupně maximálně d, každé z $x_0 \ldots x_d$ je kořenem tohoto polynomu $\Rightarrow$ spor, polynom stupně d má $d+1$ kořenů $\Rightarrow$ $P-Q$ musí být nulový polynom $\Rightarrow$ $P = Q$. ♡

Povšimněme si jedné skutečnosti – máme-li dva polynomy A a B stupně d a body $x_0, \ldots, x_n$, dále polynom $C = A \cdot B$ (stupně $2d$), pak platí $C(x_j) = A(x_j) \cdot$

$B(x_j)$ pro $j = 0, 1, 2, \ldots, n$. Toto činí tento druhý způsob reprezentace polynomu velice atraktivním pro násobení – máme-li A i B reprezentované hodnotami v $n \geq 2d + 1$ bodech, pak snadno (v $\Theta(n)$) spočteme takovou reprezentaci C. Problémem je, že typicky máme polynom zadaný koeficienty, a ne hodnotami v bodech. Tím pádem potřebujeme nějaký hodně rychlý algoritmus (tj. rychlejší než kvadratický, jinak bychom si nepomohli oproti hloupému algoritmu) na převod polynomu z jedné reprezentace do druhé a zase zpět.

Idea, jak by měl algoritmus pracovat:

1. Vybereme $n \geq 2d + 1$ bodů $x_0, x_1, \ldots, x_{n-1}$.
2. V těchto bodech vyhodnotíme polynomy A a B.
3. Nyní již v lineárním čase získáme hodnoty polynomu C v těchto bodech: $C(x_i) = A(x_i) \cdot B(x_i)$
4. Převedeme hodnoty polynomu C na jeho koeficienty.

Je vidět, že klíčové jsou kroky 2 a 4. Celý trik spočívá v chytrém vybrání oněch bodů, ve kterých budeme polynomy vyhodnocovat – zvolí-li se obecná x_j, tak se to rychle neumí, pro speciální x_j ale ukážeme, že to rychle jde.

Vyhodnocení polynomu metodou Rozděl a panuj (algoritmus FFT):

Mějme polynom P stupně $\leq d$ 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:

$$P(x) = (p_0 x^0 + p_2 x^2 + \ldots + p_{d-2} x^{d-2}) + (p_1 x^1 + p_3 x^3 + \ldots + p_{d-1} x^{d-1})$$

se zavedením značení:

$$P_s(t) = p_0 t^0 + p_2 t^1 + \ldots + p_{d-2} t^{\frac{d-2}{2}}$$

$$P_l(t) = p_1 t^0 + p_3 t^1 + \ldots + p_{d-1} t^{\frac{d-2}{2}}$$

bude $P(x) = P_s(x^2) + xP_l(x^2)$ a $P(-x) = P_s(x^2) - xP_l(x^2)$. Jinak řečeno, vyhodnocování polynomu P v n bodech se nám smrskne na vyhodnocení P_s a P_l v $n/2$ bodech – oba jsou polynomy stupně nejvýše $d/2$ a vyhodnocujeme je v x^2 (využíváme rovnosti $(x_i)^2 = (-x_i)^2$).

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 mocnina 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.

Komplexní intermezzo

Základní operace

- Definice: $\mathbb{C} = \{a + b\mathrm{i} \mid a, b \in \mathbb{R}\}$
- Sčítání: $(a + b\mathrm{i}) \pm (p + q\mathrm{i}) = (a \pm p) + (b \pm q)\mathrm{i}$.
  Pro $\alpha \in \mathbb{R}$ je $\alpha(a + b\mathrm{i}) = \alpha a + \alpha b\mathrm{i}$.
- Komplexní sdružení: $\overline{a + b\mathrm{i}} = a - b\mathrm{i}$.
  $\overline{\overline{x}} = x$, $\overline{x \pm y} = \overline{x} \pm \overline{y}$, $\overline{x \cdot y} = \overline{x} \cdot \overline{y}$, $x \cdot \overline{x} \in \mathbb{R}$.
- Absolutní hodnota: $|x| = \sqrt{x \cdot \overline{x}}$, takže $|a + b\mathrm{i}| = \sqrt{a^2 + b^2}$.
  Také $|\alpha x| = |\alpha| \cdot |x|$.
- Dělení: $x/y = (x \cdot \overline{y})/(y \cdot \overline{y})$.

Gaußova rovina a goniometrický tvar

- Komplexním číslům přiřadíme body v $\mathbb{R}^2$: $a + b\mathrm{i} \leftrightarrow (a, b)$.
- $|x|$ je vzdálenost od bodu $(0, 0)$.
- $|x| = 1$ pro čísla ležící na jednotkové kružnici (*komplexní jednotky*).
  Pak platí $x = \cos\varphi + \mathrm{i}\sin\varphi$ pro nějaké $\varphi \in [0, 2\pi)$.
- Pro libovolné $x \in \mathbb{C}$: $x = |x| \cdot (\cos\varphi(x) + \mathrm{i}\sin\varphi(x))$.
  Číslu $\varphi(x) \in [0, 2\pi)$ říkáme *argument* čísla x, někdy značíme $\arg x$.
- Navíc $\varphi(\overline{x}) = -\varphi(x)$.

Exponenciální tvar

- Eulerova formule: $e^{i\varphi} = \cos\varphi + \mathrm{i}\sin\varphi$.
- Každé $x \in \mathbb{C}$ lze tedy zapsat jako $|x| \cdot e^{\mathrm{i}\cdot\varphi(x)}$.
- Násobení: $xy = \left(|x| \cdot e^{\mathrm{i}\cdot\varphi(x)}\right) \cdot \left(|y| \cdot e^{\mathrm{i}\cdot\varphi(y)}\right) = |x| \cdot |y| \cdot e^{\mathrm{i}\cdot(\varphi(x)+\varphi(y))}$.
  (absolutní hodnoty se násobí, argumenty sčítají)
- Umocňování: $x^\alpha = \left(|x| \cdot e^{\mathrm{i}\cdot\varphi(x)}\right)^\alpha = |x|^\alpha \cdot e^{\mathrm{i}\alpha\varphi(x)}$.
- Odmocňování: $\sqrt[n]{x} = |x|^{1/n} \cdot e^{\mathrm{i}\cdot\varphi(x)/n}$.
  Pozor – odmocnina není jednoznačná: $1^4 = (-1)^4 = \mathrm{i}^4 = (-\mathrm{i})^4 = 1$.

Odmocniny z jedničky

- Je-li nějaké $x \in \mathbb{C}$ n-tou odmocninou z jedničky, musí platit: $|x| = 1$, takže $x = e^{\mathrm{i}\varphi}$ pro nějaké φ. Proto $x^n = e^{\mathrm{i}\varphi n} = \cos\varphi n + \mathrm{i}\sin\varphi n = 1$. Platí tedy $\varphi n = 2k\pi$ pro nějaké $k \in \mathbb{Z}$.
- Z toho plyne: $\varphi = 2k\pi/n$
  (pro $k = 0, \ldots, n-1$ dostáváme různé n-té odmocniny).
- Obecné odmocňování: $\sqrt[n]{x} = |x|^{1/n} \cdot e^{\mathrm{i}\varphi(x)/n} \cdot u$, kde $u = \sqrt[n]{1}$.
- Je-li x odmocninou z 1, pak $\overline{x} = x^{-1}$ – je totiž $1 = |x \cdot \overline{x}| = x \cdot \overline{x}$.

Primitivní odmocniny

Definice: x je *primitivní* k-tá odmocnina z $1 \equiv x^k = 1 \;\&\; \forall j : 0 < j < k \Rightarrow x^j \neq 1$.

Tuto definici splňují například čísla $\omega = e^{2\pi\mathrm{i}/k}$ a $\overline{\omega} = e^{-2\pi\mathrm{i}/k}$. Platí totiž, že $\omega^j = e^{2\pi\mathrm{i}j/k}$, což je rovno 1 právě tehdy, je-li j násobkem k (jednotlivé mocniny čísla ω postupně obíhají jednotkovou kružnici). Analogicky pro $\overline{\omega}$.

Ukažme si několik pozorování fungujících pro libovolné číslo ω, které je primitivní k-tou odmocninou z jedničky (někdy budeme potřebovat, aby navíc k bylo sudé):

- Pro $0 \leq j < l < k$ je $\omega^j \neq \omega^l$, neboť $\omega^l/\omega^j = \omega^{l-j} \neq 1$, protože $l - j < k$ a ω je primitivní.
- $\omega^{k/2} = -1$, protože $(\omega^{k/2})^2 = 1$, a tedy $\omega^{k/2}$ je druhá odmocnina z 1. Takové odmocniny jsou dvě: 1 a -1, ovšem 1 to být nemůže, protože ω je primitivní.
- $\omega^j = -\omega^{k/2+j}$ – přímý důsledek předchozího bodu, pro nás ale velice zajímavý: $\omega^0, \omega^1, \ldots, \omega^{k-1}$ jsou po dvou spárované.
- ω^2 je $k/2$-tá primitivní odmocnina z 1 – dosazením.

Konec intermezza

Vraťme se nyní k algoritmu. Z poslední části komplexního intermezza se zdá, že by nemusel být špatný nápad zkusit vyhodnocovat polynom v mocninách n-té primitivní odmocniny z jedné (tedy za $x_0, x_1, \ldots, x_{n-1}$ z původního algoritmu zvolíme $\omega^0, \omega^1, \ldots, \omega^{n-1}$). Aby nám vše vycházelo pěkně, zvolíme n jako mocninu dvojky.

Celý algoritmus bude vypadat takto:

FFT(P, ω)

Vstup: $p_0, \ldots, p_{n-1}$, koeficienty polynomu P stupně nejvýše $n-1$, a ω, n-tá primitivní 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 členy se sudými a lichými exponenty (jako v původní myšlence) a rekurzivně zavoláme FFT(P_s, ω^2) a FFT(P_l, ω^2) – P_l i P_s jsou stupně max. $n/2-1$, ω^2 je $n/2$-tá primitivní odmocnina, a mocniny ω^2 jsou stále po dvou spárované (n je mocnina dvojky, a tedy i $n/2$ je sudé; popř. $n = 2$ a je to zřejmé).
3. Pro $j = 0, \ldots, n/2 - 1$ spočítáme:
4. $\quad P(\omega^j) = P_s(\omega^{2j}) + \omega^j \cdot P_l(\omega^{2j})$.
5. $\quad P(\omega^{j+n/2}) = P_s(\omega^{2j}) - \omega^j \cdot P_l(\omega^{2j})$.

Časová složitost: $T(n) = 2T(n/2) + \Theta(n) \Rightarrow$ složitost $\Theta(n \log n)$, 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: *Diskrétní Fourierova transformace* (DFT) je zobrazení $f : \mathbb{C}^n \to \mathbb{C}^n$, kde

$$y = f(x) \equiv \forall j \; y_j = \sum_{k=0}^{n-1} x_k \cdot \omega^{jk}$$

(DFT si lze mimo jiné představit jako funkci vyhodnocující polynom s koeficienty x_k v bodech ω^j). Takovéto zobrazení je lineární a tedy popsatelné maticí Ω s prvky $\Omega_{jk} = \omega^{jk}$. Chceme-li umět převádět z hodnot polynomu na koeficienty, zajímá nás inverze této matice.

Jak najít inverzní matici?

Značme $\overline{\Omega}$ matici, jejíž prvky jsou komplexně sdružené odpovídajícím prvkům Ω, a využijme následující lemma:

Lemma:

$$\Omega \cdot \overline{\Omega} = n \cdot E.$$

Důkaz:

$$(\Omega \cdot \overline{\Omega})_{jk} = \sum_{l=0}^{n-1} \omega^{jl} \cdot \overline{\omega^{lk}} = \sum \omega^{jl} \cdot \overline{\omega}^{lk} = \sum \omega^{jl} \cdot \omega^{-lk} = \sum \omega^{l(j-k)}.$$

- Pokud $j = k$, pak $\sum_{l=0}^{n-1} (\omega^0)^l = n$.
- Pokud $j \neq k$, použijeme vzoreček pro součet geometrické řady s kvocientem $\omega^{(j-k)}$ a dostaneme $\frac{\omega^{(j-k)n}-1}{\omega^{(j-k)}-1} = \frac{1-1}{\neq 0} = 0$ ($\omega^{j-k} - 1$ je jistě $\neq 0$, neboť ω je n-tá primitivní odmoncina a $j - k < n$).

♡

Našli jsme inverzi: $\Omega(\frac{1}{n}\overline{\Omega}) = \frac{1}{n}\Omega \cdot \overline{\Omega} = E$, $\quad \Omega^{-1}_{jk} = \frac{1}{n}\overline{\omega^{jk}} = \frac{1}{n}\omega^{-jk} = \frac{1}{n}(\omega^{-1})^{jk}$, (připomínáme, ω^{-1} je $\overline{\omega}$).

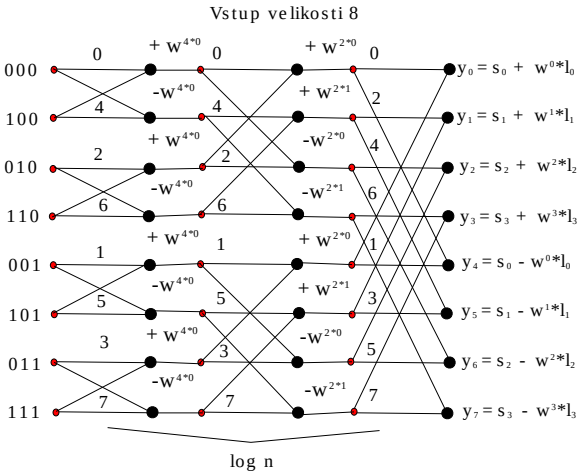
Vyhodnocení polynomu lze provést vynásobením Ω, převod do původní reprezentace vynásobením Ω^{-1}. My jsme si ale všimli chytrého spárování, a vyhodnocujeme polynom rychleji než kvadraticky (proto FFT, jakože *fast*, ne jako *fuj*). Uvědomíme-li si, že $\overline{\omega} = \omega^{-1}$ je také n-tá primitivní odmocnina z 1 (má akorát úhel s opačným znaménkem), tak můžeme stejným trikem vyhodnotit i zpětný převod – nejprve vyhodnotíme A a B v ω^j, poté pronásobíme hodnoty a dostaneme tak hodnoty polynomu $C = A \cdot B$, a pustíme na ně stejný algoritmus s ω^{-1} (hodnoty C vlastně budou v algoritmu „koeficienty polynomu“). Nakonec jen získané hodnoty vydělíme n a máme chtěné koeficienty.

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

Důsledek: Polynomy stupně n lze násobit v čase $\Theta(n \log n)$: $\Theta(n \log n)$ pro vyhodnocení, $\Theta(n)$ pro vynásobení a $\Theta(n \log n)$ pro převedení zpět.

Použití FFT:

- Zpracování signálu – rozklad na siny a cosiny o různých frekvencích ⇒ spektrální rozklad.
- Komprese dat – například formát JPEG.
- Násobení dlouhých čísel v čase $\Theta(n \log n)$.



Příklad průběhu algoritmu na vstupu velikosti 8

Paralelní implementace FFT

Zkusme si průběh algoritmu FFT znázornit graficky (podobně, jako jsme kreslili hradlové sítě). Na levé straně obrázku se nachází vstupní vektor $x_0, \ldots, x_{n-1}$ (v nějakém pořadí), na pravé straně pak výstupní vektor $y_0, \ldots, y_{n-1}$. Sledujme chod algoritmu pozpátku: Výstup spočítáme z výsledků „polovičních“ transformací vektorů $x_0, x_2, \ldots, x_{n-2}$ a $x_1, x_3, \ldots, x_{n-1}$. Černé kroužky přitom odpovídají výpočtu lineární kombinace $a + \omega^k b$, kde a, b jsou vstupy kroužku a k nějaké přirozené číslo závislé na poloze kroužku. Každá z polovičních transformací se počítá analogicky z výsledků transformace velikosti $n/4$ atd. Celkově výpočet probíhá v $\log_2 n$ vrstvách po $\Theta(n)$ operacích.

Jelikož operace na každé vrstvě probíhají na sobě nezávisle, můžeme je počítat paralelně. Náš diagram tedy popisuje hradlovou síť pro paralelní výpočet FFT v čase $\Theta(\log n \cdot T)$ a prostoru $\mathcal{O}(n \cdot S)$, kde T a S značí časovou a prostorovou složitost výpočtu lineární kombinace dvou komplexních čísel.

Cvičení: Dokažte, že permutace vektoru $x_0, \ldots, x_{n-1}$ odpovídá bitovému zrcadlení, tedy že na pozici b shora se vyskytuje prvek x_d, kde d je číslo b zapsané ve dvojkové soustavě pozpátku.

Nerekurzivní FFT

Obvod z předchozího obrázku také můžeme vyhodnocovat po hladinách zleva doprava, čímž získáme elegantní nerekurzivní algoritmus pro výpočet FFT v čase $\Theta(n \log n)$ a prostoru $\Theta(n)$:

1. *Vstup:* $x_0, \ldots, x_{n-1}$
2. Pro $j = 0, \ldots, n-1$ položíme $y_k \leftarrow x_{r(k)}$, kde r je funkce bitového zrcadlení.
3. Předpočítáme tabulku $\omega^0, \omega^1, \ldots, \omega^{n-1}$.

4. $b \leftarrow 1$ *(velikost bloku)*
5. Dokud $b < n$, opakujeme:
6. Pro $j = 0, \ldots, n-1$ s krokem $2b$ opakujeme: *(začátek bloku)*
7. Pro $k = 0, \ldots, b-1$ opakujeme: *(pozice v bloku)*
8. $\alpha \leftarrow \omega^{nk/2b}$
9. $(y_{j+k}, y_{j+k+b}) \leftarrow (y_{j+k} + \alpha \cdot y_{j+k+b}, y_{j+k} - \alpha \cdot y_{j+k+b})$.
10. $b \leftarrow 2b$
11. *Výstup:* $y_0, \ldots, y_{n-1}$

FFT v konečných tělesech

Nakonec dodejme, že Fourierovu transformaci lze zavést nejen nad tělesem komplexních čísel, ale i v některých konečných tělesech, pokud zaručíme existenci primitivní n-té odmocniny z jedničky. Například v tělese $\mathbb{Z}_p$ pro prvočíslo $p = 2^k + 1$ platí $2^k = -1$, takže $2^{2k} = 1$ a $2^0, 2^1, \ldots, 2^{2k-1}$ jsou navzájem různá, takže číslo 2 je primitivní $2k$-tá odmocnina z jedné. To se nám ovšem nehodí pro algoritmus FFT, jelikož $2k$ bude málokdy mocnina dvojky.

Zachrání nás ovšem algebraická věta, která říká, že multiplikativní grupa⟨17⟩ libovolného konečného tělesa je cyklická, tedy že všechny nenulové prvky tělesa lze zapsat jako mocniny nějakého čísla g (generátoru). Například pro $p = 2^{16} + 1 = 65\,537$ je jedním takovým generátorem číslo 3. Jelikož mezi čísly $g^1, g^2, \ldots, g^{p-1}$ se musí každý nenulový prvek tělesa vyskytnout právě jednou, je g primitivní 2^k-tou odmocninou z jedničky, takže můžeme počítat FFT pro libovolný vektor, jehož velikost je mocnina dvojky menší nebo rovná 2^k.⟨18⟩

10. Převody problémů

(zapsali Martin Chytil, Vladimír Kudelas, Michal Kozák, Vojta Tůma)

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 $L(x) =$ ANO psát prostě $x \in L$.]

Vstupy mějme zakódované jen pomocí nul a jedniček (obecně je jedno, jaký základ pro soustavu kódování zvolíme, převody mezi soustavami o nějakém základu $\neq 1$ jsou co do velikosti zápisu polynomiální). Rozhodovací problém je tedy f :

⟨17⟩ To je množina všech nenulových prvků tělesa s operací násobení.

⟨18⟩ Bližší průzkum našich úvah o FFT dokonce odhalí, že není ani potřeba těleso. Postačí libovolný komutativní okruh, ve kterém existuje příslušná primitivní odmocnina z jedničky, její multiplikativní inverze (ta ovšem existuje vždy, protože $\omega^{-1} = \omega^{n-1}$) a multiplikativní inverze čísla n. To nám poskytuje ještě daleko více volnosti než tělesa, ale není snadné takové okruhy hledat.

$\{0,1\}^* \to \{0,1\}$, to jest funkce z množiny všech řetězců jedniček a nul do množiny $\{1,0\}$, kde 1 na výstupu znamená ANO, 0 NE.

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 $A \to B$) právě tehdy, když existuje funkce f spočitatelná v polynomiálním čase taková, že pro $\forall x : A(x) = B(f(x))$. Všimněme si, že f pracující v polynomiálním čase vstup zvětší nejvíce polynomiálně.

Pozorování: $A \to B$ 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 $\mathcal{O}(a^\ell)$, kde $a = \mathfrak{q}$; takže $B(f(x))$ poběží v čase $\mathcal{O}(a^\ell + (a^\ell)^k) = \mathcal{O}(a^{k\ell})$, 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

† věrni matfyzáckým vtipům

funkce, jak už jsme zpozorovali v předchozím odstavci).

Takovýmto relacím říkáme kvaziuspořádání – nesplňují obecně antisymetrii, tedy může nastat $A \to B$ a $B \to A$. Omezíme-li se však na třídy navzájem převoditelných problémů, dostáváme již (částečné) uspořádání. Existují i navzájem nepřevoditelné problémy – například problém vždy odpovídající 1 a problém vždy odpovídající 0. Nyní se již podíváme na nějaké zajímavé problémy. Obecně to budou problémy, na které polynomiální algoritmus není znám, a vzájemnými převody zjistíme že jsou stejně těžké.

1. problém: SAT

Splnitelnost (satisfiability) logických formulí, tj. dosazení 1 či 0 za proměnné v logické formuli tak, aby formule dala výsledek 1.

Zaměříme se na speciální formu zadání formulí, *konjunktivní normální formu* (CNF), které splňují následující podmínky:

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

Formule mají tedy tvar:

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

Vstup: Formule ψ v konjunktivní normální formě.

Výstup: $\exists$ dosazení 1 a 0 za proměnné takové, že hodnota formule $\psi(\ldots) = 1$.

Převod nějaké obecné formule ψ na jí ekvivalentní χ v CNF může způsobit, že χ je exponenciálně velká vůči ψ. Později ukážeme, že lze podniknout převod na takovou formuli χ' v CNF, která sice není ekvivalentní s ψ (přibydou nám proměnné, a ne každý rozšířený model ψ je modelem χ'), ale je splnitelná právě tehdy, když je splnitelná ψ – což nám přesně stačí – a je lineárně velká vůči ψ.

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 (SAT je alespoň tak těžký jako 3-SAT)

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

Trik pro dlouhé klauzule: Každou „špatnou“ klauzuli

$$(\alpha \vee \beta), \text{ tž. } |\alpha| + |\beta| \geq 4,\ |\alpha| \geq 2,\ |\beta| \geq 2$$

přepíšeme na:

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

† bez politických konotací

kde x je nová proměnná (při každém dělení klauzule *jiná* nová proměnná).

Tento trik opakujeme tak dlouho, dokud je to třeba – formuli délky $k+l$ roztrhneme na formule délky $k+1$ a $l+1$. Pokud klauzule půlíme, dostaneme polynomiální čas (strom rekurze má logaritmicky pater – formule délky alespoň 6 se nám při rozdělení zmenší na dvě instance velikosti maximálně 2/3 původní, kratší formule nás netrápí; na každém patře se vykoná tolik co na předchozím + $2^{hloubka}$ za přidané formule). Velikost výsledné formule je tím pádem polynomiální vůči původní: v každém kroku se přidají jen dva literály, tedy celkem *čas na převod*·2 nových.

Platí-li:

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

Nabízí se otázka, proč můžeme přidanou proměnnou x nastavovat, 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.

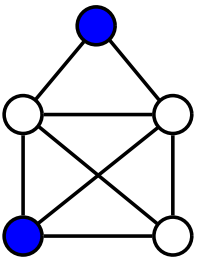
Poznámka: U 3-SAT lze vynutit právě tři literály, pro krátké klauzule použijeme stejný trik:

$$(\alpha) \to (\alpha \vee \alpha) \to (\alpha \vee x)\ \&\ (\alpha \vee \neg x).$$

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.

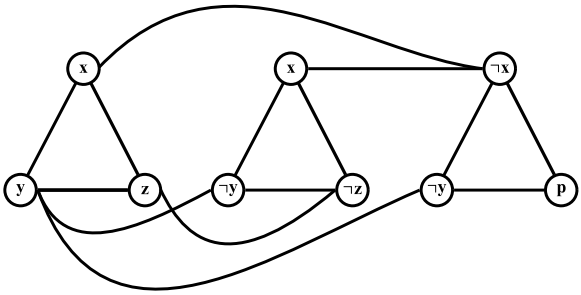
Převod 3-SAT na NzMna: Z každé klauzule vybereme jeden literál, jehož nastavením se klauzuli rozhodneme splnit. Samozřejmě tak, abychom v různých klauzulích nevybírali konfliktně, tj. x a $\neg x$.

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 $\neg x$. Počet vrcholů grafu odpovídá počtu literálů ve formuli, počet hran je maximálně kvadratický a převod je tedy polynomiální.

Existuje-li v grafu nezávislá množina velikosti k, pak z každého z k trojúhelníků vybere právě jeden vrchol, a přitom žádné dva vrcholy nebudou odpovídat literálu a jeho negaci – tedy dostaneme ohodnocení proměnných splňujících alespoň k klauzulí. Na druhou stranu, existuje-li ohodnocení k klauzulí, pak přímo odpovídá nezávislé množině velikosti k (v každém trojúhelníku zvolíme právě jednu z ohodnocených proměnných, nemůže se stát že zvolíme vrcholy konfliktní hrany). Ptáme-li se tedy na nezávislou množinu velikosti odpovídající počtu klauzulí, dostaneme odpověď ANO právě tehdy, když je formule splnitelná.

Jsou-li ve formuli i klauzule kratší než 3, můžeme je buďto prodloužit metodou výše popsanou; nebo si v grafu necháme dvoj- a jedno-úhelníky, které žádné z našich úvah vadit nebudou.



Ukázka převodu 3-SAT na nezávislou množinu

Převod NzMna na SAT:

- 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 (tedy příslušné ohodnocení proměnných bude vlastně charakteristická funkce nezávislé množiny).
- 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 klauzule, které nám řeknou, že vybrané do nezávislé množiny jsou právě ty vrcholy, které jsou touto

maticí očíslované: $\forall i, j,\ x_{ij} \Rightarrow v_j$ (jen dodejme, že $a \Rightarrow b$ je definované jako $\neg a \vee b$).

- 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 $\forall i : x_{i1} \vee x_{i2} \vee \ldots \vee x_{in}$.

Tímto vynutíme NzMnu $\geq k$, což jsme přesně chtěli. Takovýto převod je zřejmě polynomiální.

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

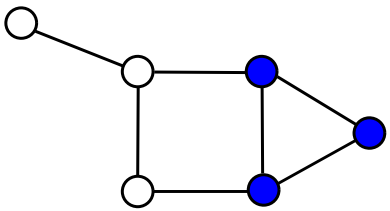
$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Vysvětlení: Jako první vrchol množiny bude vybrán vrchol v_1, proto v prvním řádku a v prvním sloupci bude 1. Jako druhý vrchol množiny bude vybrán vrchol v_4, proto na druhém 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?



Příklad kliky

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

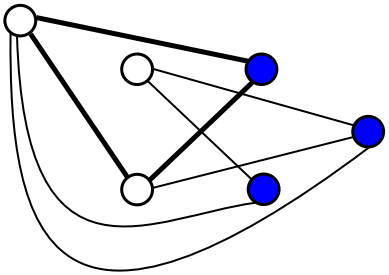
Důvod: Pokud existuje úplný graf na k vrcholech, tak v komplementárním grafu tyto vrcholy nejsou spojeny hranou, tj. tvoří nezávislou množinu, a naopak.

5. 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:

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



Prohození hran a nehran

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.

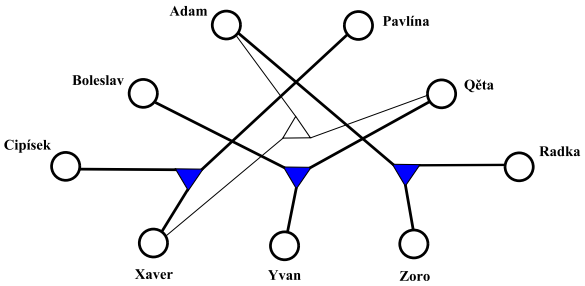
Mimochodem, můžeme rovnou zařídit, že každý literál se vyskytuje nejvíce dvakrát (tedy že každá proměnná se vyskytuje alespoň jednou pozitivně a alespoň jednou negativně). Pokud by se nějaká proměnná objevila ve třech stejných literálech, můžeme na ni také použít náš trik a nahradit ji třemi proměnnými.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

6. problém: 3D párování (3D matching)

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

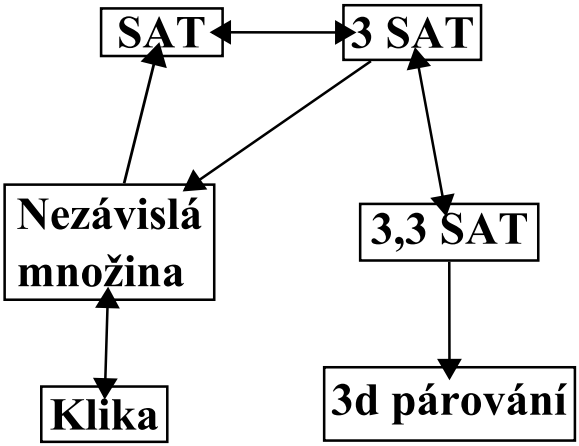
Výstup: Perfektní podmnožina trojic – tj. taková podmnožina trojic, která obsahuje všechna K, H a Z.

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



Ukázka 3D párování

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 Michal Kozák, Vojta Tůma)

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 případě 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 napověděl nějaký objekt, uměli jsme efektivně říci, zda je to ten, který hledáme. Například pokud dostaneme ohodnocení proměnných logické formule, stačí jen dosadit a spočítat, kde formule dá *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. Nadefinujeme si ji pořádně, ale nejdříve začneme trošičku jednodušší třídou.

Definice: P je *třída rozhodovacích problémů*, které jsou řešitelné v polynomiálním čase. Jinak řečeno, problém $L \in \mathrm{P} \Leftrightarrow \exists$ 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 tedy odpovídá tomu, o čem jsme se shodli, že umíme efektivně řešit. Nadefinujme nyní třídu NP:

Definice: NP je *třída rozhodovacích problémů* takových, že $L \in \mathrm{NP}$ právě tehdy, když $\exists$ problém $K \in \mathrm{P}$ 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, čili 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.

Problémy z minulé přednášky jsou všechny v NP (např. pro nezávislou množinu je onou nápovědou přímo množina vrcholů deklarující nezávislost), o jejich příslušnosti do P ale nevíme nic. 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).

Rozmyslete si, že pokud umíme řešit nějaký NP-těžký problém v polynomiálním čase, pak umíme vyřešit v polynomiálním čase vše v NP, a tedy P = NP.

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 Cookovy věty 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 nějaký problém $M \in$ NP, pak M je také NP-úplný.

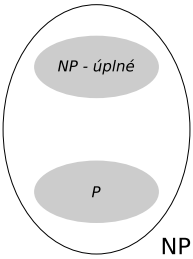
Důkaz: Tuto větičku stačí dokázat pro NP-těžkost, NP-úplnost plyne okamžitě z toho, že problémy jsou NP-těžké a leží v NP (podle předpokladu).

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 shora dolu podle obtížnosti problémů (tedy navzdor gravitaci), kde porovnání dvou problémů určuje převoditelnost (viz obrázek).

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,



Struktura třídy NP

např. každá šifra by byla jednoduše rozluštitelná. ⟨19⟩ 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 (dokonce je takových problémů nekonečně mnoho, v nekonečně třídách).

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 (existuje množina alespoň k vrcholů taková, že žádné dva nejsou propojeny hranou?)
  - Klika (existuje úplný podgraf na k vrcholech?)
  - 3D párování (tři množiny se zadanými trojicemi, existuje taková množina disjunktních trojic, ve které jsou všechny prvky?)
  - Barvení grafu (lze 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])

⟨19⟩ Poznámka o šifrách – libovolnou funkci vyčíslitelnou v polynomiálním čase bychom uměli v polynomiálním čase také invertovat.

- *číselné:*
  - Batoh (nejjednodušší verze: dána množina čísel, zjistit, zda existuje podmnožina se zadaným součtem)
  - Batoh – optimalizace (podobně jako u předchozího problému, ale místo množiny čísel máme množinu předmětů s vahami a cenami, chceme co nejdražší podmnožinu jejíž váha nepřesáhne zadanou kapacitu batohu)
  - Loupežníci (lze rozdělit danou množinu čísel 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 $Ax \leq b$?)

Nyní si ukážeme, jak převést SAT na nějaký problém. 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. Rozšiřme tedy náš katalog problémů o následující:

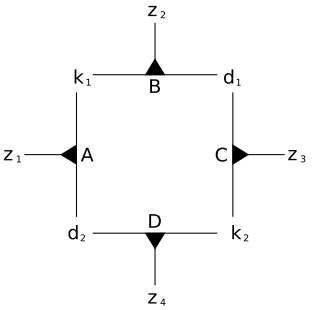
3D párování (3D matching)

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

Výstup: Perfektní podmnožina trojic $P \subseteq K \times H \times Z$ – tj. taková podmnožina trojic, že $(\forall k \in K\ \exists! p \in P : k \in p) \wedge (\forall h \in H\ \exists! p \in P : h \in p) \wedge (\forall z \in Z\ \exists! p \in P : z \in p)$ – tedy každý byl vybrán právě jednou.

Ukážeme, jak na 3D-párování převést 3,3-SAT

Uvažujme takovouto konfiguraci:

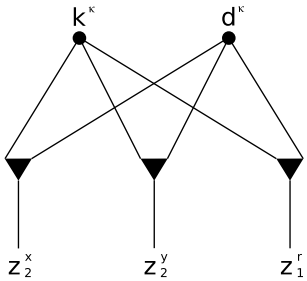


4 zvířátka, 2 kluci, 2 dívky a 4 trojice, které označíme A, B, C, D. Ještě předpokládáme, že zvířátka se mohou účastnit nějakých jiných trojic, ale tito čtyři lidé se vyskytují pouze v těchto čtyřech trojicích a nikde jinde. Všimneme si, že existují právě dvě možnosti, jak tento obrázek spárovat. Abychom 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.

Tyto konfigurace 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 bude odpovídat $x = 0$. 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 tato nespárovaná zvířátka můžeme předávat informaci, jestli proměnná x má hodnotu *true* nebo *false* do dalších částí vstupu.

Zbývá vymyslet, jak reprezentovat klauzule. Klauzule jsou trojice popř. dvojice literálů, např. $\kappa = (x \vee y \vee \neg r)$, kde 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 (podle toho, má-li se proměnná vyskytnout s negací nebo ne). 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. Pro dvojice se postupuje obdobně.

Ještě nám ale určitě zbude $2p - k$ zvířátek, kde p je počet proměnných, k počet klauzulí – každá proměnná vyrobí 4 zvířátka, klauzule zbaští jedno a samotné ohodnocení 2 zvířátka – tak přidáme ještě $2p-k$ párů kluk-děvče, kteří milují všechna zvířátka, a ti vytvoří zbývající trojice.

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 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 vše 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 (hradlovými sítěmi). 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. Začněme s pomocným lemmatem.

Lemma: Nechť L je problém v P. Potom existuje polynom p a algoritmus, který pro $\forall n \geq 0$ spočte v čase $p(n)$ hradlovou síť Bn s n vstupy a 1 výstupem takovou, že $\forall x \in \{0,1\}^n : Bn(x) = L(x)$.

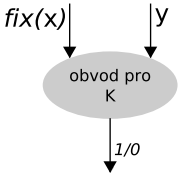
Důkaz: Náznakem. Na základě zkušeností z Principů počítačů intuitivně chápeme počítače jako nějaké složité booleovské obvody, jejichž stav se mění v čase. Uvažme tedy nějaký problém $L \in$ P a polynomiální algoritmus, který ho řeší. Pro vstup velikosti n doběhne v čase T polynomiálním v n a spotřebuje $\mathcal{O}(T)$ 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.

Poznámka: Pro důkaz následující věty si dovolíme drobnou úpravu v definici třídy NP. Budeme chtít, aby nápověda měla pevnou velikost, závislou pouze na velikosti vstupu (tedy: $|y| = g(|x|)$ namísto $|y| \leq g(|x|)$). 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.)

Věta: Obvodový SAT je NP-úplný.

Důkaz: Máme tedy nějaký problém L z NP a chceme dokázat, že L se dá převést na obvodový SAT (tj. NP-těžkost). Když nám někdo předloží nějaký vstup x (chápeme jako posloupnost $x_1, x_2, \ldots, x_n$), spočítáme velikost nápovědy $g(n)$. Víme, že kontrolní algoritmus K (který kontroluje, zda nápověda je správně) je v P. Využijeme předchozí lemma, 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 $p(g(n))$, což je také 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 pravě 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.

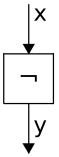
Obvodový SAT je v NP triviálně – stačí si nechat poradit vstup, síť topologicky setřídit a v tomto pořadí počítat hodnoty hradel. ♡

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

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

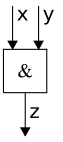
Převod hradla NOT: na vstupu hradla budeme mít nějakou proměnnou x (která přišla buďto přímo ze vstupu toho celého obvodu nebo je to proměnná, která vznikla na výstupu nějakého hradla) a na výstupu proměnnou y. Přidáme klauzule, které nám zaručí, že jedna proměnná bude negací té druhé:

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



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:

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



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 pravě 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. Aproximační algoritmy

(F. Haško, J. Menda, M. Mareš, Michal Kozák, Vojta Tůma)

Na minulých přednáškách jsme se zabývali různými těžkými rozhodovacími problémy. Tato se zabývá postupy, jak se v praxi vypořádat s řešením těchto problémů.

Co dělat, když potkáme NP-úplný problém

1. Nepanikařit.
2. Spokojit se s málem.
3. Rozmyslet, jestli opravdu potřebujeme obecný algoritmus. Mnohdy potřebujeme pouze speciálnější případy, které mohou být řešitelné v polynomiálním čase.
4. Spokojit se s přibližným řešením, (použít aproximační algoritmus).
5. Použít heuristiku – například genetické algoritmy nebo randomizované algoritmy. Velmi pomoci může i jen výhodnější pořadí při prohledávání či ořezávání některých napohled nesmyslných větví výpočtu.

První způsob: Speciální případ

Často si vystačíme s vyřešením speciálního případu NP-úplného problému, který leží v P. Například při řešení grafové úlohy nám může stačit řešení pro speciální druh grafů (stromy, bipartitní grafy, ...). Barvení grafu je lehké např. pro dvě barvy či pro intervalové grafy. 2-SAT, jako speciální případ SATu, se dá řešit v lineárním čase.

Ukážeme si dva takové případy (budeme řešení hledat, nejen rozhodovat, zda existuje)

Problém: Maximální nezávislá množina ve stromě (ne rozhodovací)

Vstup: Zakořeněný strom T.

Výstup: Maximální (co do počtu vrcholů) nezávislá množina vrcholů M v T.

BÚNO můžeme předpokládat, že v M jsou všechny listy T. Pokud by některý list l v M nebyl, tak se podíváme na jeho otce:

- Pokud otec není v M, tak list l přidáme do M, čímž se nezávislost množiny zachovala a velikost stoupla o 1.

- Pokud tam otec je, tak ho z M vyjmeme a na místo něho vložíme l. Nezávislost ani velikost M se nezměnily.

Nyní listy spolu s jejich otci z T odebereme a postup opakujeme. T se může rozpadnout na les, ale to nevadí → tentýž postup aplikujeme na všechny stromy v lese.

Algoritmus: MaxNz(T)

1. Položíme L:={listy stromu T}.
2. Položíme O:={otcové vrcholů z L}.
3. Vrátíme $L \cup$ MaxNz($T \setminus (O \cup L)$).

Poznámka: Toto dokážeme naprogramovat v $\mathcal{O}(n)$ (udržujeme si frontu listů).

Problém: Batoh

Je daná množina n předmětů s hmotnostmi $h_1, \ldots, h_n$ a cenami $c_1, \ldots, c_n$ a batoh, který unese hmostnost H. Najděte takovou podmnožinu předmětů, jejichž celková hmotnost je maximálně H a celková cena je maximální 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 $A_k(c)$ pro fixní k a všechna c, v čase $\mathcal{O}(nC)$ pak všechny $A_k(c)$.

Podle A_n snadno nalezneme maximální cenu množiny, která se vejde do batohu. To bude největší c^*, pro něž je $A_n(c^*) \leq H$. 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 může být až exponenciálně velké vzhledem k velikosti vstupu), ale pouze ve velikosti čísel na vstupu. Takovým

algoritmům se říká *pseudopolynomiální.* Ani takové algoritmy ale nejsou k dispozici pro všechny problémy (např. u problému obchodního cestujícího nám vůbec nepomůže, že váhy hran budou malá čísla).

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} — udržujme si Z_{k-1} jako setříděný spojový seznam, výpočet dalšího seznamu uděláme slitím dvou seznamů Z_{k-1} a Z_{k-1} se všemi prvky zvýšenými o hmotnost k zahazujíce duplicitní a příliš velké hodnoty — a ze Z_n vyčteme výsledek. Všechny tyto množiny mají nejvýše H prvků, takže celková časová složitost algoritmu je $\mathcal{O}(nH)$.

Druhý způsob: Aproximace

V předcházejících problémech jsme se zaměřili na speciální případy. Občas však takové štěstí nemáme a musíme vyřešit celý NP-úplný problém. Můžeme si však pomoct tím, že se ho nebudeme snažit vyřešit optimálně – namísto optimálního řešení najdeme nějaké, které je nejvýše c-krát horší pro nějakou konstantu c.

Problém: Obchodní cestující

Vstup: Neorientovaný graf G, každá hrana je ohodnocená funkcí $w : E(G) \to \mathbb{R}_0^+$.

Výstup: Hamiltonovská kružnice (obsahující všechny vrcholy grafu), a to ta nejkratší (podle ohodnocení).

Tento problém je hned na první pohled náročný – už sama existence hamiltonovské kružnice je NP-úplná. Najdeme aproximační algoritmus nejprve za předpokladu, že vrcholy splňují trojúhelníkovou nerovnost (tj. $\forall x, y, z \in V : w(xz) \leq w(xy) + w(yz)$), potom ukážeme, že v úplně obecném případě by samotná existence aproximačního algoritmu implikovala P = NP.

a) trojúhelníková nerovnost:

Existuje pěkný algoritmus, který najde hamiltonovskou kružnici o délce $\leq 2 \cdot opt$, kde opt je délka nejkratší hamiltonovské kružnice. Vedle předpokladu trojúhelníkové nerovnosti budeme potřebovat, aby náš graf byl úplný. Souhrnně můžeme předpokládat, že úlohu řešíme v nějakém metrickém protoru, ve kterém jsou obě podmínky podle definice splněny.

Najdeme nejmenší kostru grafu a obchodnímu cestujícímu poradíme, ať jde po ní – kostru zakořeníme a projdeme jako strom do hloubky, přičemž se zastavíme až v kořeni po projití všech vrcholů. Problém však je, že průchod po kostře obsahuje některé vrcholy i hrany vícekrát, a proto musíme nahradit nepovolené vracení se. Máme-li na nějaký vrchol vstoupit podruhé, prostě ho ignorujeme a přesuneme se rovnou na další nenavštívený – dovolit si to můžeme, graf je úplný a obsahuje hrany mezi všemi dvojicemi vrcholů (jinak řečeno, pořadí vrcholů kružnice bude preorder výpis průchodem do hloubky). Pokud platí trojúhelníková nerovnost, tak si těmito zkratkami neuškodíme. Nechť minimální kostra má váhu T. Pokud bychom prošli celou kostru, bude mít sled váhu $2T$ (každou hranou kostry jsme šli tam a zpátky),

a přeskakování vrcholů celkovou váhu nezvětšuje (při přeskoku nahradíme cestu xyz jedinou hranou xz, přičemž z trojúhelníkové nerovnosti máme $xz \leq xy + xz$), takže váha nalezené hamiltonovské kružnice bude také nanejvýš $2T$.

Když máme hamiltonovskou kružnici C a z ní vyškrtneme hranu, dostaneme kostru grafu G s váhou menší než C – ale každá kostra je alespoň tak těžká jako minimální kostra T. Tedy optimální hamiltonovská kružnice je určitě těžší než minimální kostra T. Když tyto dvě nerovnosti složíme dohromady, algoritmus nám vrátí hamiltonovskou kružnici T' s váhou nanejvýš dvojnásobnou vzhledem k optimální hamiltonovské kružnici ($T' \leq 2T < 2C$). Takovéto algoritmy se nazývají *2-aproximační*, když řešení je maximálně dvojnásobné od optimálního.⟨20⟩

b) bez trojúhelníkové nerovnosti:

Zde se budeme naopak snažit ukázat, že žádný polynomiální aproximační algoritmus neexistuje.

Věta: Pokud pro libovolné $\varepsilon > 0$ existuje polynomiální $(1+\varepsilon)$-aproximační algoritmus pro problém obchodního cestujícího bez trojúhelníkové nerovnosti, tak P = NP.

Důkaz: Ukážeme, že v takovém případě dokážeme v polynomiálním čase zjistit, zda v grafu existuje hamiltonovská kružnice.

Dostali jsme graf G, ve kterém hledáme hamiltonovskou kružnici. Doplníme G na úplný graf G' a váhy hran G' nastavíme takto:

- $w(e) = 1$, když $e \in E(G)$
- $w(e) = c \gg 1$, když $e \notin E(G)$

Konstantu c potřebujeme zvolit tak velkou, abychom jasně poznali, jestli je každá hrana z nalezené hamiltonovské kružnice hranou grafu G (pokud by nebyla, bude kružnice obsahovat aspoň jednu hranu s váhou c, která vyžene součet poznatelně vysoko). Pokud existuje hamiltonovská kružnice v G' složená jen z hran, které byly původně v G, pak optimální řešení bude mít váhu n, jinak bude určitě minimálně $n - 1 + c$. Když máme aproximační algoritmus s poměrem $1 + \varepsilon$, musí tedy být

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

Kdyby takový algoritmus existoval, máme polynomiální algoritmus na hamiltonovskou kružnici. ♡

Poznámka: O existenci pseudopolynomiálního algoritmu platí analogická věta, a dokáže se analogicky – existující hrany budou mít váhu 1, neexistující váhu 2.

⟨20⟩ Hezkým trikem se v obecných metrických prostorech umí 1,5-aproximace. Ve některých metrických prostorech (třeba v euklidovské rovině) se aproximační poměr dá dokonce srazit na libovolně blízko k 1. Zaplatíme ale na čase – čím přesnější výsledek po algoritmu chceme, tím déle to bude trvat.

Aproximační schéma pro problém batohu

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 $M < c_{max}$ a zobrazíme interval cen $[0, c_{max}]$ na $[0, M]$ (tedy každou cenu znásobíme M/c_{max}). Jak jsme tím zkreslili výsledek? Všimněme si, že efekt je stejný, jako kdybychom jednotlivé ceny zaokrouhlili na násobky čísla c_{max}/M (prvky z intervalu $[i \cdot c_{max}/M, (i+1) \cdot c_{max}/M)$ se zobrazí na stejný prvek). Každé c_i jsme tím tedy změnili o nejvýše c_{max}/M, celkovou cenu libovolné podmnožiny předmětů pak 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 $M \geq n/\varepsilon$.⟨21⟩

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: $\forall i : \hat{c}_i \leftarrow \lfloor c_i \cdot M/c_{max} \rfloor$.
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 $\hat{C} \leq nM = \mathcal{O}(n^2/\varepsilon)$, 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, jakou cenu budou mít předměty které daly optimální řešení v původním zadání (tedy mají v původním zadání dohromady cenu OPT),

⟨21⟩ Připoměňme, že toto ještě není důkaz, neboť velkoryse přehlížíme chyby dané zaokrouhlováním. Důkaz provedeme níže.

když jejich ceny 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 spočítejme, jak dopadne optimální 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⟨22⟩). V našem případě je dokonce složitost polynomiální i v závislosti na $1/\varepsilon$, takže schéma je *plně polynomiální* (řečené též FPTAS⟨23⟩). U některých problémů se stává, že aproximační schéma závisí na $1/\varepsilon$ exponenciálně, což tak příjemné není. Shrňme, co jsme zjistili, do následující věty:

Věta: Existuje algoritmus, který pro každé $\varepsilon > 0$ nalezne $(1-\varepsilon)$-*aproximaci* problému batohu s n předměty v čase $\mathcal{O}(n^3/\varepsilon)$.

⟨22⟩ Polynomial-Time Approximation Scheme
⟨23⟩ Fully Polynomial-Time Approximation Scheme