Třídy složitosti

Složitost problémů

- Ukazuje se, že různé (algoritmické) problémy jsou různě těžké.
- Obtížnější jsou ty problémy, k jejichž řešení potřebujeme více času a paměti.
- Obtížnost problémů chceme nějak posuzovat, a to jak
  - absolutně — kolik času a kolik paměti potřebujeme k jejich řešení, tak
  - relativně — o kolik je jejich řešení obtížnější nebo naopak jednodušší oproti jiným problémům.
- Proč se u některých problémů nedaří nalézt efektivní algoritmy? Může vůbec nějaký efektivní algoritmus pro daný problém existovat?
- Kde přesně jsou limity toho, co je možné prakticky zvládnout?

Složitost problémů

Je potřeba rozlišovat **složitost algoritmu** a **složitost problému**.

Pokud například zkoumáme časovou složitost v nejhorším případě, mohli bychom neformálně říct:

- **složitost algoritmu** — funkce, která vyjadřuje, jaká bude pro daný algoritmus maximální doba výpočtu pro vstup velikosti n
- **složitost problému** — jaká je časová složitost „nejefektivnějšího“ algoritmu, který řeší daný problém

Zavedení pojmu „složitost problému“ ve výše uvedeném smyslu naráží na značné technické obtíže. Pojem „složitost problému“ se tedy jako takový nedefinuje, ale obchází se zavedením tzv. **tříd složitosti**.

Třídy složitosti

Třídy složitosti jsou podmnožiny množiny všech (algoritmických) **problémů**.

Daná konkrétní třída složitosti je vždy charakterizována nějakou vlastností, kterou mají problémy do ní patřící.

Typickým příkladem takové vlastnosti je vlastnost, že pro daný problém existuje nějaký algoritmus s určitým omezením (např. časové nebo prostorové složitosti):

- Do dané třídy pak patří všechny problémy, pro které takovýto algoritmus existuje.
- Naopak do ní nepatří problémy, pro které žádný takový algoritmus neexistuje.

Poznámka: V následujícím popisu se budeme soustředit prakticky jen na třídy **rozhodovacích** problémů.

Třídy složitosti

Definice

Pro libovolnou funkci $f : \mathbb{N} \to \mathbb{N}$ definujeme třídu $\mathsf{DTIME}(f(n))$ jako třídu obsahující právě ty rozhodovací problémy, pro něž existuje algoritmus s časovou složitostí $\mathcal{O}(f(n))$.

Příklad:

- $\mathsf{DTIME}(n)$ – třída všech rozhodovacích problémů pro něž existuje algoritmus s časovou složitostí $\mathcal{O}(n)$
- $\mathsf{DTIME}(n^2)$ – třída všech rozhodovacích problémů pro než existuje algoritmus s časovou složitostí $\mathcal{O}(n^2)$
- $\mathsf{DTIME}(n \log n)$ – třída všech rozhodovacích problémů pro než existuje algoritmus s časovou složitostí $\mathcal{O}(n \log n)$

Třídy složitosti

Definice

Pro libovolnou funkci $f : \mathbb{N} \rightarrow \mathbb{N}$ definujeme třídu $\mathrm{DSPACE}(f(n))$ jako třídu obsahující právě ty rozhodovací problémy, pro něž existuje algoritmus s prostorovou složitostí $\mathcal{O}(f(n))$.

Příklad:

- $\mathrm{DSPACE}(n)$ – třída všech rozhodovacích problémů pro něž existuje algoritmus s prostorovou složitostí $\mathcal{O}(n)$
- $\mathrm{DSPACE}(n^2)$ – třída všech rozhodovacích problémů pro než existuje algoritmus s prostorovou složitostí $\mathcal{O}(n^2)$
- $\mathrm{DSPACE}(n \log n)$ – třída všech rozhodovacích problémů pro než existuje algoritmus s prostorovou složitostí $\mathcal{O}(n \log n)$

Třídy složitosti

Poznámka:

Všimněte si, že u tříd DTIME(f) a DSPACE(f) může to, které problémy do dané třídy patří, záviset na použitém výpočetním modelu (zda je to stroj RAM, jednopáskový Turingův stroj, vícepáskový Turingův stroj, ...).

Třídy složitosti

Pomocí tříd DTIME$(f(n))$ a DSPACE$(f(n))$ můžeme definovat třídy PTIME a PSPACE jako

$$\text{PTIME} = \bigcup_{k \geq 0} \text{DTIME}(n^k) \qquad\qquad \text{PSPACE} = \bigcup_{k \geq 0} \text{DSPACE}(n^k)$$

- PTIME je třída všech rozhodovacích problémů, pro které existuje algoritmus s polynomiální časovou složitostí, tj. s časovou složitostí $\mathcal{O}(n^k)$, kde k je nějaká konstanta.

- PSPACE je třída všech rozhodovacích problémů, pro které existuje algoritmus s polynomiální prostorovou složitostí, tj. s prostorovou složitostí $\mathcal{O}(n^k)$, kde k je nějaká konstanta.

Třídy složitosti

Vzhledem k tomu, že všechny „rozumné" výpočetní modely jsou schopné se navzájem simulovat tak, že při dané simulaci nevzroste počet kroků ani množství použité paměti víc než polynomiálně, není definice tříd PTIME a PSPACE závislá na použitém výpočetním modelu.
Pro jejich zadefinování můžeme použít kterýkoliv výpočetní model.

Říkáme, že tyto třídy jsou **robustní** — jejich definice nezávisí na použitém výpočetním modelu, pro všechny „rozumné" sekvenční výpočetní modely obsahují tytéž problémy.

Poznámka: Za „rozumné" sekvenční výpočetní modely jsou považovány ty, které se umí vzájemně simulovat s Turingovými stroji tak, že doba výpočtu vzroste při takové simulaci nanejvýš polynomiálně.

„Rozumné“ sekvenční výpočetní modely

Příklady výpočetních modelů považovaných z tohoto hlediska za „rozumné“:

- různé varianty Turingových strojů (jednopáskové, vícepáskové, ...)
- stroje RAM při použití logaritmické míry
- stroje RAM, které nemají operace násobení a dělení, při použití jednotkové míry
- stroje RAM, které sice mají operace násobení a dělení, při použití jednotkové míry, pokud je zároveň pro daný stroj zaručeno, že velikosti všech čísel ve všech buňkách během výpočtu je možné omezit nějakým polynomem vzhledem k době výpočtu

Výpočetní modely, které „rozumné“ nejsou

Příklady výpočetních modelů, které „rozumnými“ sekvečními výpočetními modely z tohoto hlediska nejsou:

- stroje RAM s operacemi násobení a dělení v jednotkové míře (bez omezení na velikosti čísel, na kterých je možné v jednom kroku provádět aritmetické operace) — mohou v jediném kroku provádět aritmetické operace na číslech, která mají počet bitů exponenciální vzhledem k počtu provedených instrukcí
- Minského stroje — jsou příliš pomalé, provedení jednoduchých operací trvá příliš dlouhou dobu; při simulaci činnosti Turingova stroje vzrůstá čas výpočtu oproti původnímu Turingovu stroji exponenciálně

Třídy složitosti

Analogicky můžeme zavést další třídy:

EXPTIME – množina všech rozhodovacích problémů, pro které existuje algoritmus s časovou složitostí $2^{\mathcal{O}(n^k)}$, kde k je nějaká konstanta

EXPSPACE – množina všech rozhodovacích problémů, pro které existuje algoritmus s prostorovou složitostí $2^{\mathcal{O}(n^k)}$, kde k je nějaká konstanta

LOGSPACE – množina všech rozhodovacích problémů, pro které existuje algoritmus s prostorovou složitostí $\mathcal{O}(\log n)$

Poznámka: Místo $2^{\mathcal{O}(n^k)}$ bychom mohli psát také $\mathcal{O}(c^{n^k})$, kde c a k jsou nějaké konstanty.

Třídy složitosti

Při definici třídy LOGSPACE musíme přesněji specifikovat, co považujeme za prostorovou složitost algoritmu.

Uvažujeme například Turingův stroj, který pracuje se třemi páskami:

- **Vstupní páskou**, na které je na začátku výpočtu zapsán vstup. Z této pásky je možno pouze číst.
- **Pracovní páskou**, která je na začátku výpočtu prázdná. Z této pásky je možno číst i na ni zapisovat.
- **Výstupní páskou**, která je také na začátku výpočtu prázdná a na kterou je možno pouze zapisovat.

Množství použité paměti je pak definováno jako počet použitých políček na pracovní pásce.

Třídy složitosti

Další příklady tříd složitosti:

2-EXPTIME – množina všech problémů, pro které existuje algoritmus s časovou složitostí $2^{2^{\mathcal{O}(n^k)}}$, kde k je nějaká konstanta

2-EXPSPACE – množina všech problémů, pro které existuje algoritmus s prostorovou složitostí $2^{2^{\mathcal{O}(n^k)}}$, kde k je nějaká konstanta

ELEMENTARY – množina všech problémů, pro které existuje algoritmus s časovou (či prostorovou) složitostí

$$2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{2^{\mathcal{O}(n^k)}}}}}}}}$$

kde k je konstanta a počet exponentů je omezen konstantou.

Vztahy mezi třídami složitosti

Pokud Turingův stroj provede m kroků, tak použije maximálně m políček na pásce.

Pokud tedy existuje pro nějaký problém algoritmus s časovou složitostí $\mathcal{O}(f(n))$, má tento algoritmus prostorovou složitost (nejvýše) $\mathcal{O}(f(n))$.

Je tedy zřejmé, že platí následující vztah.

Pozorování

Pro libovolnou funkci $f : \mathbb{N} \rightarrow \mathbb{N}$ platí $\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n))$.

Poznámka: Analogicky bychom mohli argumentovat například pro stroj RAM.

Vztahy mezi třídami složitosti

Z předchozího okamžitě plyne:

$$\begin{aligned}\text{PTIME} &\subseteq \text{PSPACE}\\ \text{EXPTIME} &\subseteq \text{EXPSPACE}\\ \text{2-EXPTIME} &\subseteq \text{2-EXPSPACE}\\ &\vdots\end{aligned}$$

Vzhledem k tomu, že polynomiální funkce rostou pomaleji než exponenciální a logaritmické pomaleji než polynomiální, zjevně platí:

$$\text{PTIME} \subseteq \text{EXPTIME} \subseteq \text{2-EXPTIME} \subseteq \cdots$$

$$\text{LOGSPACE} \subseteq \text{PSPACE} \subseteq \text{EXPSPACE} \subseteq \text{2-EXPSPACE} \subseteq \cdots$$

Vztahy mezi třídami složitosti

Při zkoumání vztahů mezi třídami složitosti se ukazuje jako užitečný pojem **konfigurace**.

Konfigurací budeme rozumět celkový stav, ve kterém se během jednoho kroku nachází stroj, provádějící nějaký daný algoritmus.

- U Turingova stroje je konfigurace dána stavem jeho řídící jednotky, obsahem pásky (resp. pásek) a pozicí hlavy (resp. hlav).
- U stroje RAM je konfigurace dána obsahem paměti, obsahem všech registrů (včetně IP), obsahem vstupní a výstupní pásky a pozicemi čtecí a zapisovací hlavy.

Vztahy mezi třídami složitosti

Mělo by být jasné, že konfigurace (resp. jejich popisy) můžeme zapisovat jako slova v nějaké abecedě.

Navíc můžeme konfigurace zapisovat tak, že délka těchto slov bude zhruba stejná jako množství paměti použité algoritmem (tj. počet políček na pásce použitých Turingovým stojem, počet bitů paměti použitých strojem RAM apod.).

Poznámka: Pokud máme abecedu Σ, kde $|\Sigma| = c$, tak:

- Počet slov délky n je c^n, tj. $2^{\Theta(n)}$.
- Počet slov délky nejvýše n je

$$\sum_{i=0}^{n} c^n = \frac{c^{n+1} - 1}{c - 1}$$

tj. také $2^{\Theta(n)}$.

Vztahy mezi třídami složitosti

Je jasné, že během výpočtu korektního algoritmu se žádná konfigurace nemůže zopakovat, protože jinak by se algoritmus zacyklil a běžel by donekonečna.

Pokud tedy víme, že prostorová složitost nějakého algoritmu je $\mathcal{O}(f(n))$, znamená to, že počet různých konfigurací dosažitelných během výpočtu je $2^{\mathcal{O}(f(n))}$.

Protože se konfigurace během žádného výpočtu neopakují, je i časová složitost daného algoritmu maximálně $2^{\mathcal{O}(f(n))}$.

Pozorování

Pro libovolnou funkci $f : \mathbb{N} \to \mathbb{N}$ platí, že pokud je nějaký problém P řešený algoritmem s prostorovou složitostí $\mathcal{O}(f(n))$, pak časová složitost tohoto algoritmu je v $2^{\mathcal{O}(f(n))}$.

Pokud je tedy problém P ve třídě $\mathrm{DSPACE}(f(n))$, pak je i ve třídě $\mathrm{DTIME}(2^{c \cdot f(n)})$ pro nějaké $c > 0$.

Vztahy mezi třídami složitosti

Z předchozího plynou následující důsledky:

$$\begin{aligned}
\text{LOGSPACE} &\subseteq \text{PTIME} \\
\text{PSPACE} &\subseteq \text{EXPTIME} \\
\text{EXPSPACE} &\subseteq \text{2-EXPTIME} \\
&\vdots
\end{aligned}$$

Shrnutí:

$$\begin{aligned}
&\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE} \subseteq \\
&\quad \subseteq \text{2-EXPTIME} \subseteq \text{2-EXPSPACE} \subseteq \cdots \subseteq \text{ELEMENTARY}
\end{aligned}$$

Horní a dolní odhady složitosti problémů

Horním odhadem složitosti problému rozumíme to, že složitost problému není vyšší než nějaká uvedená.

Většinou je to formulováno tak, že daný problém patří do nějaké určité třídy složitosti.

Příklady tvrzení, které se týkají horních odhadů složitosti:

- Problém dosažitelnosti v grafu je v PTIME.
- Problém ekvivalence dvou regulárních výrazů je v EXPSPACE.

Pokud chceme zjistit nějaký horní odhad složitosti problému, stačí ukázat, že existuje algoritmus s danou složitostí.

Horní a dolní odhady složitosti problémů

Dolním odhadem složitosti problému rozumíme to, že složitost problému je alespoň taková jako nějaká uvedená.

Obecně je zjišťování (netriviálních) dolních odhadů složitosti problémů mnohem obtížnější než zjišťování horních odhadů.

Pro odvození dolního odhadu musíme totiž ukázat, že **každý** algoritmus řešící daný problém má danou složitost.

Horní a dolní odhady složitosti problémů

Problém „Třídění“

Vstup: Posloupnost prvků $a_1, a_2, \ldots, a_n$.

Výstup: Prvky $a_1, a_2, \ldots, a_n$ setříděné od nejmenšího po největší.

Dá se dokázat, že každý algoritmus, který řeší problém “Třídění” a na prvcích tříděné posloupnosti používá pouze operaci porovnávání (tj. nezkoumá obsah těchto prvků), má časovou složitost v nejhorším případě v $\Omega(n \log n)$ (tj. pro každý takový algoritmus existují konstanty $c > 0$ a $n_0 \geq 0$ takové, že pro každé $n \geq n_0$ existuje vstup velikosti n, pro který provede algoritmus nejméně $cn \log n$ operací).

Nedeterministické algoritmy a třídy složitosti

Nedeterminismus

Zatím jsme uvažovali jen deterministické algoritmy.

Můžeme ale uvažovat i **nedeterministické** algoritmy realizované nedeterministickými variantami nejrůznějších druhů strojů:

- Turingovými stroji
- stroji RAM
- ...

Obecně jsou nedeterministické algoritmy takové, kde:

- V každém kroku si může algoritmus vybrat z několika možností, kterou instrukcí pokračovat.
- Pokud ze všech možných výpočtů takového stroje nad zadaným vstupem alespoň jeden skončí s odpovědí Ano, je odpověď Ano.
- Pokud všechny výpočty skončí s odpovědí Ne, je odpověď Ne.

Nedeterminismus

Například u jednopáskového Turingova stroje se bude deterministická a nedeterministická varianta lišit pouze v definici přechodové funkce δ:

- **deterministický**: $\delta : (Q - F) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$
- **nedeterministický**: $\delta : (Q - F) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{-1, 0, +1\})$

Nedeterministický stroj RAM:

- Je definován velice podobně jako deterministický RAM.
- Navíc má instrukci

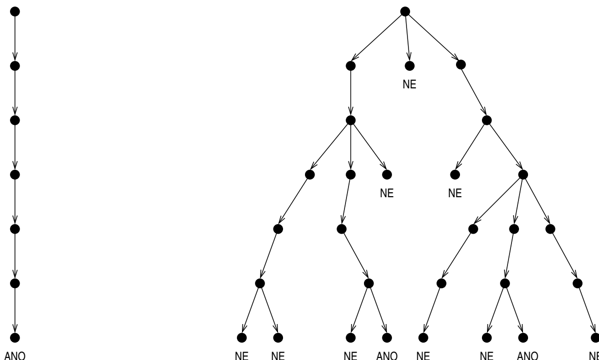
  $$\textbf{nd_goto}\ \ell_1,\ \ell_2$$

  která umožňuje stroji vybrat si jedno z možných pokračování.

Podobně můžeme definovat nedeterministické varianty libovolných jiných výpočetních modelů.

Nedeterminismus

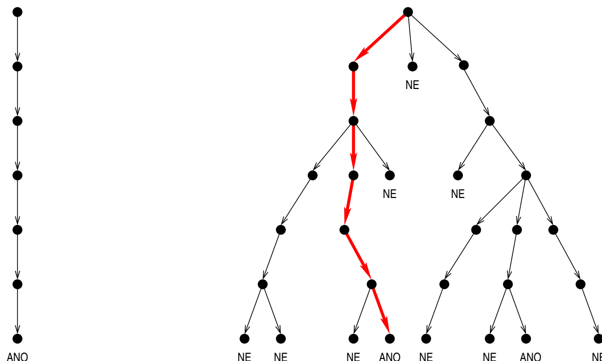
Nedeterministický algoritmus dává pro daný vstup x odpověď ANO, jestliže **existuje** alespoň jeden jeho výpočet, který vede k odpovědi ANO.



- Doba výpočtu nedeterministického stroje RAM (nebo jiného nedeterministického stroje) nad zadaným vstupem je definována jako délka nejdelšího možného výpočtu nad tímto vstupem.

Nedeterminismus

Nedeterministický algoritmus dává pro daný vstup x odpověď Ano, jestliže **existuje** alespoň jeden jeho výpočet, který vede k odpovědi Ano.



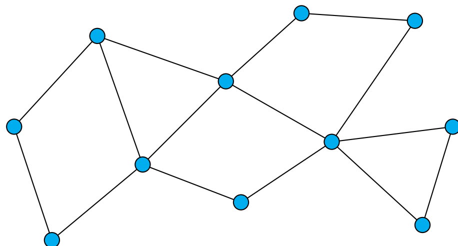
- Doba výpočtu nedeterministického stroje RAM (nebo jiného nedeterministického stroje) nad zadaným vstupem je definována jako délka nejdelšího možného výpočtu nad tímto vstupem.

Nedeterminismus

Problém „Barvení grafu k barvami“

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Je možné obarvit vrcholy grafu G k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?



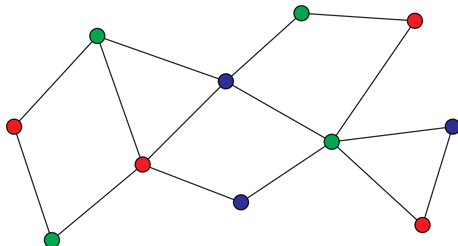
$k = 3$

Nedeterminismus

Problém „Barvení grafu k barvami“

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Je možné obarvit vrcholy grafu G k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?



$k = 3$

Nedeterminismus

Problém „Barvení grafu k barvami“

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Je možné obarvit vrcholy grafu G k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

Nedeterministický algoritmus pracuje následovně:

1. Každému vrcholu grafu G nedeterministicky přiřadí jednu z k barev.
2. Projde všechny hrany grafu G a u každé z nich zkontroluje, že oba její koncové vrcholy jsou obarveny různými barvami. Pokud ne, skončí s odpovědí NE.
3. Pokud prošel všechny hrany a u všech byly koncové vrcholy obarveny různými barvami, skončí s odpovědí ANO.

Nedeterminismus

Problém „Isomorfismus grafů“

Vstup: Neorientované grafy $G_1 = (V_1, E_1)$ a $G_2 = (V_2, E_2)$.

Otázka: Jsou grafy G_1 a G_2 isomorfní?

Poznámka: Grafy G_1 a G_2 jsou isomorfní, jestliže existuje nějaká bijekce $f : V_1 \to V_2$ taková, že pro libovolné dva vrcholy $u, v \in V_1$ platí $(u, v) \in E_1$ právě když $(f(u), f(v)) \in E_2$.

Nedeterministický algoritmus pracuje následovně:

1. Nedeterministicky zvolí hodnoty funkce f pro všechny $v \in V_1$.
2. Deterministicky ověří, že f je bijekce a že pro všechny dvojice vrcholů je splněna výše uvedená podmínka.
3. Pokud je některá z podmínek porušena, skončí s odpovědí Ne, v opačném případě s odpovědí Ano.

Nedeterminismus

Booleovská formule φ je **splnitelná**, jestliže existuje nějaké pravdivostní ohodnocení ν, při kterém je formule φ pravdivá, tj. nabývá pravdivostní hodnoty 1.

SAT (splnitelnost booleovských formulí)

Vstup: Booleovská formule φ.

Otázka: Je φ splnitelná?

Příklad:
Formule $\varphi_1 = x_1 \wedge (\neg x_2 \vee x_3)$ je splnitelná:
např. při ohodnocení ν, kde $\nu(x_1) = 1$, $\nu(x_2) = 0$, $\nu(x_3) = 1$, je formule φ_1 pravdivá.

Formule $\varphi_2 = (x_1 \wedge \neg x_1) \vee (\neg x_2 \wedge x_3 \wedge x_2)$ není splnitelná:
je nepravdivá při každém ohodnocení ν.

Nedeterminismus

Nedeterministický algoritmus řešící problém SAT v polynomiáním čase:

- Načte formuli φ.
- Postupně v cyklu projde všechny proměnné $x_1, x_2, \ldots, x_k$, které se ve formuli φ nachází.

  Pro každou z nich nedeterministicky zvolí, zda jí přiřadí hodnotu 0 nebo 1.
- Vyhodnotí pravdivostní hodnotu formule φ při daném ohodnocení.

  Skončí s odpovědí Ano, pokud je formule při daném ohodnocení pravdivá.

  V opačném případě skončí s odpovědí Ne.

Nedeterminismus

- Z hlediska rozhodnutelnosti nepřináší nedeterministické algoritmy oproti deterministickým nic dalšího navíc:
  Pokud je nějaký problém možné řešit nedeterministickým strojem RAM nebo TS, tak je ho možné řešit i deterministickým, který postupně vyzkouší všechny možné výpočty nedeterministického stroje nad daným vstupem.

- Nedeterminismus má význam především při zkoumání výpočetní složitosti problémů.

Nedeterminismus

- Při výše uvedené přímočaré simulaci činnosti nedeterministického algoritmu pomocí deterministického, který systematicky zkouší všechny možné výpočty, je časová složitost deterministického algoritmu exponenciálně vyšší než u nedeterministického.
- Pro řadu problémů je zjevné, že pro ně existuje nedeterministický algoritmus s polynomiální časovou složitostí, ale není vůbec jasné, jestli pro ně existuje také deterministický algoritmus s polynomiální časovou složitostí.

Nedeterminismus

Na nedeterminismus můžeme nahlížet následujícími způsoby:

1. Ve chvíli, kdy má stroj nedeterministicky zvolit mezi několika možnostmi, tak „uhodne“, která z těchto možností povede k odpovědi Ano (pokud taková možnost existuje).
2. Ve chvíli, kdy má stroj nedeterministicky zvolit mezi několika možnostmi, rozdělí se do tolika kopií, kolik je těchto možností, a každá z těchto kopií pokračuje ve výpočtu odpovídající jedné z možností, přičemž pracují všechny paralelně.

   Odpověď je Ano právě tehdy, když alespoň jedna z kopií stroje odpoví Ano.

Ani jedno z toho není něco, co by se dalo efektivně realisticky implementovat.

Nedeterminismus

Další možný pohled na nedeterminismus:

- Druh algoritmu, který sice neřeší daný problém, ale s použitím dodatečné další informace — **svědka** (**witness**) — umí **ověřit**, že pro danou instanci je odpověď Ano.

  Předpokládejme, že v původním problému je vstupem nějaké x z množiny instancí In a otázka je, zda má dané x nějakou specifikovanou vlastnost P.

  Pro daný vstup x je dána množina **potenciálních svědků** $\mathcal{W}(x)$, přičemž právě tehdy, když x má vlastnost P, tak existuje nějaký **skutečný svědek** $y \in \mathcal{W}(x)$ toho, že x tuto vlastnost P skutečně má.

  Vezměme si **deterministický** algoritmus Alg, který jako vstup dostane dvojici (x, y) (kde $y \in \mathcal{W}(x)$) a ověří, zda y je svědkem toho, že x má vlastnost P.

Nedeterminismus

Příklad: Problém „Barvení grafu k barvami“:

- *Vstup*: Neorientovaný graf $G = (V, E)$ a číslo k.
- *Potenciální svědci*: Všechna možná obarvení vrcholů grafu G s použím k barev, tj. všechny možné funkce $c : V \rightarrow \{1, \ldots, k\}$.
- *Skuteční svědci*: Taková obarvení c, kde pro každou hranu $(u, v) \in E$ platí, že $c(u) \neq c(v)$.

Nedeterminismus

- Ke každému takovému **deterministickému** algoritmu Alg, který pro danou dvojici (x, y) umí ověřit, zda y je svědkem toho, že x má vlastnost P, je možné snadno sestrojit odpovídající **nedeterministický** algoritmus, který řeší původní problém:
  - Pro dané $x \in In$ nejprve neterministicky vygeneruje potenciálního svědka $y \in \mathcal{W}(x)$.
  - Použije algoritmus Alg jako podprogram k (deterministickému) ověření toho, zda je y skutečným svědkem.

Nedeterminismus

- Naopak ke každému **nedeterministickému** algoritmu můžeme snadno vytvořit **deterministický** algoritmus ověřující svědky:
  - Potenciálním svědkem bude posloupnost udávající pro jednotlivé kroky původního nedeterministického algoritmu, která možnost se má v daném kroku zvolit.
  - Deterministický algoritmus simuluje jeden konkrétní výpočet (jednu větev stromu) původního algoritmu, přičemž v krocích, kdy má na výběr z více možností, tak nehádá, ale postupuje podle toho, co je určeno v zadané posloupnosti.

Nedeterminismus

Zejména nás budou zajímat ty případy, kdy časová složitost algoritmu pro ověřování svědka je polynomiální vzhledem k velikosti vstupu x.

Mimo jiné to znamená, že daný svědek y, dosvědčující, že pro x je odpověď ANO, musí být polynomiálně velký.

Nedeterministickým algoritmem s polynomiální časovou složitostí se tedy dají řešit ty rozhodovací problémy, kde:

- pro daný vstup x existuje příslušný (polynomiálně velký) svědek právě tehdy, když pro x je odpověď ANO,
- je možné deterministickým algoritmem v polynomiálním čase ověřit, že daný potenciální svědek je skutečně svědkem.

Nedeterminismus

Mnohdy je existence takových polynomiálně velkých svědků a deterministických algoritmů, které je ověřují, očividná a je triviální ukázat, že existují — např. u problémů jako „Barvení grafu k barvami“, „Isomorfismus grafů“ nebo u následujícího problému:

Testování složenosti

Vstup: Přirozené číslo x.

Otázka: Je číslo x složené?

Poznámka: Číslo x je **složené**, když existují přirozená čísla a a b taková, že $a > 1$, $b > 1$ a $x = a \cdot b$.

Například číslo 15 je složené, protože $15 = 3 \cdot 5$.

Číslo $x \in \mathbb{N}$ je tedy složené, pokud $x > 1$ a x není prvočíslo.

Existence takových polynomiálně velkých svědků ale nutně neznamená, že je snadné je najít.

Nedeterminismus

U některých problémů může být ale ukázání existence takových polynomiálně velkých svědků, které je možné deterministicky v polynomiálním čase ověřovat, značně netriviálním výsledkem.

Příkladem je následující problém:

Testování prvočíselnosti

Vstup: Přirozené číslo x.

Otázka: Je číslo x prvočíslo?

S využitím různých netriviálních poznatků z teorie čísel se dá ukázat existence takových svědků i pro tento problém — svědci zde mají podobu rekurzivně definované datové struktury.

Poznámka: Tento výsledek ukázal V. Pratt v roce 1975.

Mnohem později bylo ukázáno, že „Testování prvočíselnosti“ je ve skutečnosti v PTIME (Agrawal–Kayal–Saxena, 2002).

Nedeterministické třídy složitosti

Definice

Pro funkci $f : \mathbb{N} \rightarrow \mathbb{N}$ rozumíme **třídou časové složitosti** NTIME(f) množinu těch rozhodovacích problémů, které jsou řešeny nedeterministickými RAMy s časovou složitostí v $\mathcal{O}(f(n))$.

Definice

Pro funkci $f : \mathbb{N} \rightarrow \mathbb{N}$ rozumíme **třídou prostorové složitosti** NSPACE(f) množinu těch rozhodovacích problémů, které jsou řešeny nedeterminictickými RAMy s prostorovou složitostí v $\mathcal{O}(f(n))$.

Poznámka: Ve výše uvedených definicích mohou být samozřejmě místo strojů RAM uvedeny třeba Turingovy stroje či nějaký jiný výpočetní model.

Třída NPTIME

Definice

$$\text{NPTIME} = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k)$$

- NPTIME (někdy se píše jen NP) je třída všech problémů, pro které existuje nedeterministický algoritmus s polynomiání časovou složitostí.
- Do NPTIME tedy patří problémy, u kterých je možné pro daný vstup rychle ověřit, že odpověď je Ano, pokud nám ten, kdo nás o tom chce přesvědčit, dodá nějakou dodatečnou informaci.

Třídy NPSPACE, NEXPTIME, NEXPSPACE, ...

Podobně můžeme definovat další třídy složitosti:

NPSPACE – množina všech rozhodovacích problémů, pro které existuje nedeterministický algoritmus s polynomiání prostorovou složitostí

NEXPTIME – množina všech rozhodovacích problémů, pro které existuje nedeterministický algoritmus s časovou složitostí $2^{\mathcal{O}(n^k)}$, kde k je nějaká konstanta

NEXPSPACE – množina všech rozhodovacích problémů, pro které existuje nedeterministický algoritmus s prostorovou složitostí $2^{\mathcal{O}(n^k)}$, kde k je nějaká konstanta

NLOGSPACE – množina všech rozhodovacích problémů, pro které existuje nedeterministický algoritmus s prostorovou složitostí $\mathcal{O}(\log n)$

Názvy tříd složitosti

Poznámky: V literatuře se běžně používají pro některé třídy složitosti také následující kratší označení.

Tato kratší označení budeme někdy používat i v tomto předmětu:

L	–	LOGSPACE
NL	–	NLOGSPACE
P	–	PTIME
NP	–	NPTIME
EXP	–	EXPTIME
NEXP	–	NEXPTIME

Vztahy mezi třídami složitosti

Je zřejmé, že na deterministické algoritmy se můžeme dívat jako na speciální případ nedeterministických.

Očividně tedy platí:

$$
\begin{aligned}
\mathrm{LOGSPACE} &\subseteq \mathrm{NLOGSPACE}\\
\mathrm{PTIME} &\subseteq \mathrm{NPTIME}\\
\mathrm{PSPACE} &\subseteq \mathrm{NPSPACE}\\
\mathrm{EXPTIME} &\subseteq \mathrm{NEXPTIME}\\
\mathrm{EXPSPACE} &\subseteq \mathrm{NEXPSPACE}\\
&\vdots
\end{aligned}
$$

Vztahy mezi třídami složitosti

Rovněž je zřejmé, že jak u deterministických, tak u nedeterministických algoritmů, algoritmus během výpočtu nemůže použít řádově více buněk paměti, než kolik udělá kroků.

Prostorová složitost daného algoritmu je tedy vždy nejvýše taková, jaká je jeho časová složitost.

Z toho plyne:

$$\begin{aligned} \mathsf{PTIME} &\subseteq \mathsf{PSPACE} \\ \mathsf{NPTIME} &\subseteq \mathsf{NPSPACE} \\ \mathsf{EXPTIME} &\subseteq \mathsf{EXPSPACE} \\ \mathsf{NEXPTIME} &\subseteq \mathsf{NEXPSPACE} \\ &\vdots \end{aligned}$$

Vztahy mezi třídami složitosti

Věta

Nedeterministický algoritmus s **časovou** složitostí $\mathcal{O}(f(n))$ je možné simulovat **deterministickým** algoritmem s **prostorovou** složitostí $\mathcal{O}(f(n))$.

Důkaz: Vezměme si nějaký nedeterministický algoritmus s časovou složitostí $\mathcal{O}(f(n))$.

Deteriministický algoritmus bude simulovat jeho činnost tím způsobem, že bude systematicky procházet všechny jeho výpočty:

- Bude procházet strom všech výpočtů procházením do hloubky.
- Bude používat zásobník, na který si bude ukládat informace nutné k tomu, aby se v každém kroku mohl vrátit k předchozí konfiguraci.
  — aby bylo možné z konfigurace α' obnovit předchozí konfiguraci α, stačí si uložit konstatní množství informace — jen to, co se při přechodu z α do α' změnilo

Vztahy mezi třídami složitosti

Tento simulující algoritmus bude potřebovat následující paměť:

- paměť, kde je uložena aktuální konfigurace simulovaného stroje — má velikost $\mathcal{O}(f(n))$ (protože pokud tento simulovaný nedeterministický stroj udělá maximálně $\mathcal{O}(f(n))$ kroků, tak jeho konfigurace budou používat nanejvýš $\mathcal{O}(f(n))$ buněk paměti)
- paměť pro uložení zásobníku, který bude používat k tomu, aby se mohl vracet k předchozím konfiguracím

  Vzhledem k tomu, že délka větví je $\mathcal{O}(f(n))$, množství potřebné paměti pro zásobník je $\mathcal{O}(f(n))$.

Celkově tedy tento deterministický algoritmus vystačí s množstvím paměti, které je nejvýše $\mathcal{O}(f(n))$.

Vztahy mezi třídami složitosti

Z výše uvedeného vyplývá:

$$\text{NPTIME} \subseteq \text{PSPACE}$$
$$\text{NEXPTIME} \subseteq \text{EXPSPACE}$$
$$\vdots$$

Vztahy mezi třídami složitosti

Vezměme si nějaký **nedeterministický** algoritmus s **prostorovou** složitostí $\mathcal{O}(f(n))$:

- Připomeňme, že konfigurací velikosti nejvýše $\mathcal{O}(f(n))$ je $\mathcal{O}(c^{f(n)})$, kde c je nějaká konstanta, což můžeme psát jako $2^{\mathcal{O}(f(n))}$.
- Počet kroků tohoto nedeterministického algoritmu v rámci jedné větve výpočtu tedy může být až $2^{\mathcal{O}(f(n))}$.

  (Pozn.: Žádná konfigurace se během výpočtu nemůže zopakovat, protože jinak by mohly být výpočty nekonečné.)
- Simulace výše popsaným způsobem by tedy měla časovou složitost až $2^{2^{\mathcal{O}(f(n))}}$.

Vztahy mezi třídami složitosti

Při simulaci ale můžeme postupovat o něco chytřeji — představme si orientovaný graf, kde:

- **vrcholy** — všechny konfigurace simulovaného stroje, jejichž velikost je nejvýše $\mathcal{O}(f(n))$
  — těchto konfigurací je $2^{\mathcal{O}(f(n))}$
- **hrany** — mezi vrcholy, které reprezentují konfigurace α a α' vede hrany právě tehdy, když simulovaný stroj může přejít jedním krokem z konfigurace α do konfigurace α'
  — z každého vrcholu povede počet hran omezený shora nějakou kostantou — hran tedy bude také řádově $2^{\mathcal{O}(f(n))}$

Stačí umět zjistit, zda ve výše uvedeném grafu existuje cesta z vrcholu, který odpovídá počáteční konfiguraci (pro daný vstup x), do některého vrcholu, který odpovídá koncové konfiguraci, kdy daný stroj dává odpověď Ano.

Pro zjištění existence takové cesty je možné použít libovolný algoritmus na **procházení grafu** — ať už procházení do šířky nebo procházení do hloubky:

- Algoritmus si musí ukládat a značit, které konfigurace již navštívil. Další paměť potřebuje pro uložení fronty či zásobníku, apod.
- Časová i prostorová složitost tohoto algoritmu bude lineárně úměrná velikosti daného grafu, tj. $2^{\mathcal{O}(f(n))}$.

Vztahy mezi třídami složitosti

Dostáváme tedy následující:

Věta

Činnost nedeterministického algoritmu, jehož prostorová složitost je $\mathcal{O}(f(n))$, je možné simulovat deterministickým algoritmem, jehož časová složitost je $2^{\mathcal{O}(f(n))}$.

Z toho vyplývá:

$$\begin{aligned} \text{NLOGSPACE} &\subseteq \text{PTIME} \\ \text{NPSPACE} &\subseteq \text{EXPTIME} \\ \text{NEXPSPACE} &\subseteq \text{2-EXPTIME} \\ &\vdots \end{aligned}$$

Vztahy mezi třídami složitosti

Uvažujeme opět nějaký **nedeterministický** algoritmus s **prostorovou** složitostí $\mathcal{O}(f(n))$. Teď nám ale pro změnu půjde o co nejmenší **prostorovou** složitost simulujícího deterministického algoritmu.

Věta (Savitch, 1970)

Činnost nedeterministického algoritmu s prostorovou složitostí $\mathcal{O}(f(n))$ je možné simulovat deterministickým algoritmem s prostorovou složitostí $\mathcal{O}(f(n)^2)$.

Myšlenka důkazu:

- Opět si představme výše popsaný graf konfigurací, který má $2^{\mathcal{O}(f(n))}$ vrcholů (i hran).
- Algoritmus bude zjišťovat, zda existuje cesta z počáteční konfigurace do některé přijímající konfigurace.

Vztahy mezi třídami složitosti

Základem bude rekurzivní funkce $F(\alpha, \alpha', i)$, která pro libovolné zadané konfigurace α a α' a číslo $i \in \mathbb{N}$ zjistí, zda ve výše uvedeném grafu existuje cesta z α do α' délky nejvýše 2^i:

- Pokud je $i = 0$, zjistí, zda existuje cesta z α do α' délky nejvýše 1:
  - buď je to cesta délky 0, tj. $\alpha = \alpha'$,
  - nebo je to cesta délky 1, tj. je možné přejít z α do α' jedním krokem
- Pokud je $i > 0$, bude systematicky probírat všechny možné konfigurace α'' a testovat, jestli:
  - existuje cesta délky nejvýše $2^i/2$ z α do α'' — zavolá rekurzivně $F(\alpha, \alpha'', i-1)$
  - existuje cesta délky nejvýše $2^i/2$ z α'' do α' — zavolá rekurzivně $F(\alpha'', \alpha', i-1)$

  Pokud obojí vrátí TRUE, vrátí TRUE, jinak pokračuje zkoušením dalšího α''.

Vztahy mezi třídami složitosti

Analýza prostorové složitosti daného algoritmu:

- V rámci jednoho rekurzivního volání funkce F je třeba mít uložené:
  - tři konfigurace α, α', α'' — všechny jsou velikosti $\mathcal{O}(f(n))$
  - hodnotu čísla i, které je řádově $\mathcal{O}(f(n))$ — proto na jeho uložení stačí zhruba $\mathcal{O}(\log f(n))$ bitů
  - další pomocné proměnné, jejichž hodnoty jsou proti velikosti výše uvedených položek zanedbatelné
- Množství paměti potřebné v rámci jednoho rekurzivního volání je tedy $\mathcal{O}(f(n))$.
- Hloubka zanoření rekurze je také $\mathcal{O}(f(n))$.
- Celková prostorová složitost daného algoritmu je tedy $\mathcal{O}(f(n)^2)$.

Vztahy mezi třídami složitosti

Z výše uvedené věty vyplývá:

$$\text{NPSPACE} \subseteq \text{PSPACE}$$
$$\text{NEXPSPACE} \subseteq \text{EXPSPACE}$$
$$\vdots$$

Spolu s triviálními fakty, že PSPACE ⊆ NPSPACE, EXPSPACE ⊆ NEXPSPACE, atd., nám to tedy dává:

$$\text{PSPACE} = \text{NPSPACE}$$
$$\text{EXPSPACE} = \text{NEXPSPACE}$$
$$\vdots$$

Poznámka: Všimněte si, že z výše uvedeného **nevyplývá**, že by muselo platit LOGSPACE = NLOGSPACE.

Vztahy mezi třídami složitosti

Celkově tak dostáváme následující **hierarchii tříd složitosti**:

$$
\begin{gathered}
\mathsf{LOGSPACE} \subseteq \mathsf{NLOGSPACE} \subseteq \\
\subseteq \mathsf{PTIME} \subseteq \mathsf{NPTIME} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} \subseteq \\
\subseteq \mathsf{EXPTIME} \subseteq \mathsf{NEXPTIME} \subseteq \mathsf{EXPSPACE} = \mathsf{NEXPSPACE} \subseteq \\
\vdots
\end{gathered}
$$

NP-úplné problémy

NP-úplné problémy

Existuje velká skupina algoritmických problémů označovaných jako **NP-úplné** problémy, které:

- patří do třídy NPTIME, tj. jsou řešitelné v polynomiálním čase **nedeterministickým** algoritmem
- jsou tedy řešitelné v exponenciálním čase
- není pro ně znám žádný algoritmus s polynomiální časovou složitostí
- na druhou stranu není ani dokázáno, že daný pro daný problém nemůže algoritmus s polynomiální časovou složitostí existovat
- jsou všechny navzájem polynomiálně převeditelné

Poznámka: Toto není definice NP-úplných problémů. Ta bude uvedena později.

Polynomiální převody mezi problémy

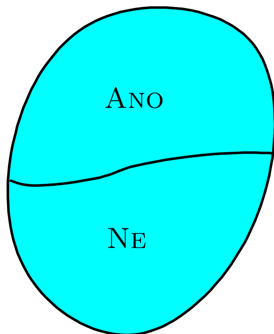
Problém P_1 je **polynomiálně převeditelný** na problém P_2, jestliže existuje algoritmus Alg s polynomiální časovou složitostí, který převádí problém P_1 na problém P_2.

Tento algoritmus Alg:

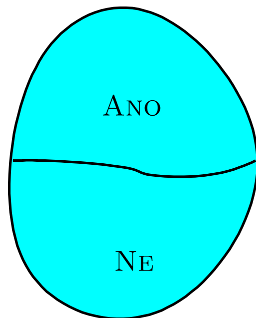
- Jako vstup může dostat libovolnou instanci problému P_1.
- K instanci problému P_1, kterou dostane jako vstup (označme ji x), vyprodukuje jako svůj výstup instanci problému P_2 (označme ji $Alg(x)$).
- Platí, že pro vstup x je v problému P_1 odpověď Ano právě tehdy, když pro vstup $Alg(x)$ je v problému P_2 odpověď Ano.

Polynomiální převody mezi problémy

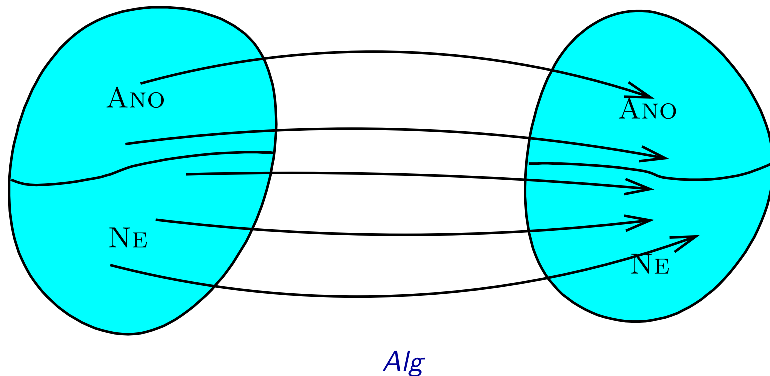
vstupy problému P_1



vstupy problému P_2



Polynomiální převody mezi problémy



Polynomiální převody mezi problémy

Řekněme, že problém P_1 je polynomiálně převeditelný na problém P_2, tj. existuje polynomiální algoritmus Alg realizující tento převod.

Pokud pro problém P_2 existuje polynomiální algoritmus, pak i pro problém P_1 existuje polynomiální algoritmus.

Řešení problému P_1 pro vstup x:

- Zavoláme Alg se vstupem x, vrátí nám hodnotu $Alg(x)$.
- Zavoláme algoritmus řešící problém P_2 se vstupem $Alg(x)$. Hodnotu, kterou nám vrátí, vypíšeme jako výsledek.

Z toho plyne:

Pokud neexistuje polynomiální algoritmus pro problém P_1, tak neexistuje ani polynomiální algoritmus pro problém P_2.

Polynomiální převody mezi problémy

Ukážeme si příklad polynomiálního převodu problému 3-SAT, což je jedna z variant problému SAT, na problém nezávislé množiny (IS).

Poznámka: Jak 3-SAT, tak IS jsou příklady NP-úplných problémů.

Problém 3-SAT

3-SAT je varianta problému SAT, ve které se omezujeme na formule určitého speciálního typu:

3-SAT

Vstup: Formule φ v konjunktivní normální formě, kde každá klauzule obsahuje právě 3 literály.

Otázka: Je φ splnitelná?

Problém 3-SAT

Připomenutí některých pojmů:

- **Literál** je formule tvaru x nebo $\neg x$, kde x je atomický výrok.
- **Klauzule** je disjunkce literálů.

  Příklady: $x_1 \vee \neg x_2$ $\neg x_5 \vee x_8 \vee \neg x_{15} \vee \neg x_{23}$ x_6
- Formule je v **konjunktivní normální formě (KNF)**, jestliže je konjunkcí klauzulí.

  Příklad: $(x_1 \vee \neg x_2) \wedge (\neg x_5 \vee x_8 \vee \neg x_{15} \vee \neg x_{23}) \wedge x_6$

V případě problému 3-SAT tedy vyžadujeme, aby formule φ byla v KNF a navíc, aby každá klauzule obsahovala právě tři literály.

Příklad:

$$(x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

Problém 3-SAT

Následující formule je splnitelná:

$(x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_3) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$

Je pravdivá např. při ohodnocení ν, kde

$$\nu(x_1) = 0$$
$$\nu(x_2) = 1$$
$$\nu(x_3) = 0$$
$$\nu(x_4) = 1$$

Naproti tomu následující formule není splnitelná:

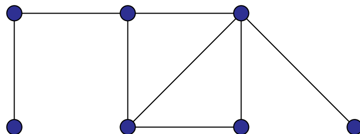
$$(x_1 \vee x_1 \vee x_1) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_1)$$

Problém nezávislé množiny (IS)

Problém nezávislé množiny (IS)

Vstup: Neorientovaný graf G, číslo k.

Otázka: Existuje v grafu G nezávislá množina velikosti k?



$k = 4$

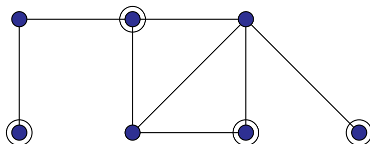
Poznámka: **Nezávislá množina** v grafu je podmnožina vrcholů grafu taková, že žádné dva vrcholy z této podmnožiny nejsou spojeny hranou.

Problém nezávislé množiny (IS)

Problém nezávislé množiny (IS)

Vstup: Neorientovaný graf G, číslo k.

Otázka: Existuje v grafu G nezávislá množina velikosti k?

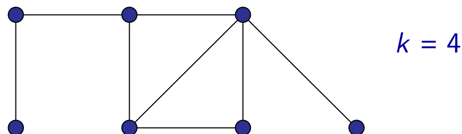


$k = 4$

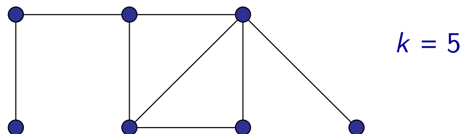
Poznámka: **Nezávislá množina** v grafu je podmnožina vrcholů grafu taková, že žádné dva vrcholy z této podmnožiny nejsou spojeny hranou.

Problém nezávislé množiny (IS)

Příklad instance, kde je odpověď Ano:



Příklad instance, kde je odpověď Ne:



Převod 3-SAT na IS

Popíšeme (polynomiální) algoritmus, který bude mít následující vlastnosti:

- **Vstup:** Libovolná instance problému 3-SAT, tj. formule φ v konjunktivní normální formě, kde každá klauzule obsahuje právě tři literály.
- **Výstup:** Instance problému IS, tj. neorientovaný graf G a číslo k.
- Navíc bude pro libovolný vstup (tj. pro libovolnou formuli φ ve výše uvedeném tvaru) zaručeno následující:

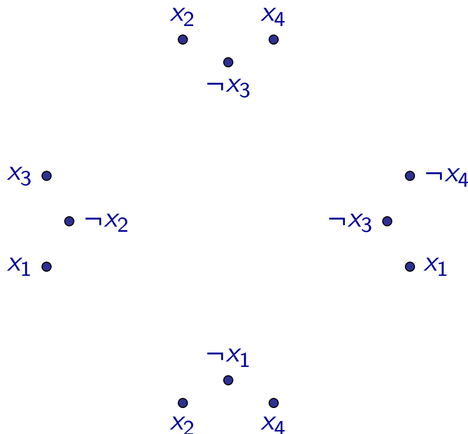
  V grafu G bude existovat nezávislá množina velikosti k právě tehdy, když formule φ bude splnitelná.

Převod 3-SAT na IS

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Převod 3-SAT na IS

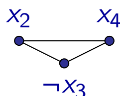
$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$

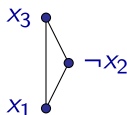


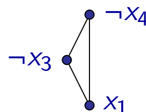
Pro každý výskyt literálu přidáme do grafu jeden vrchol.

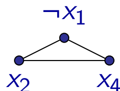
Převod 3-SAT na IS

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$





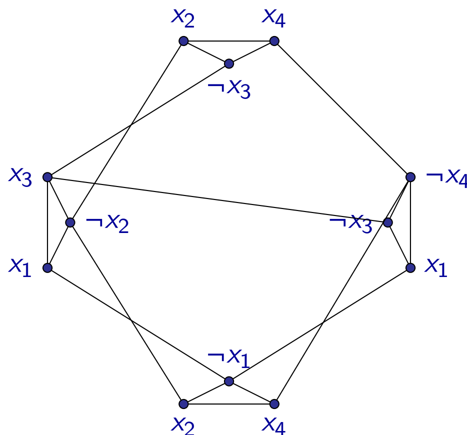




Vrcholy odpovídající výskytům literálů patřícím do stejné klauzule spojíme hranami.

Převod 3-SAT na IS

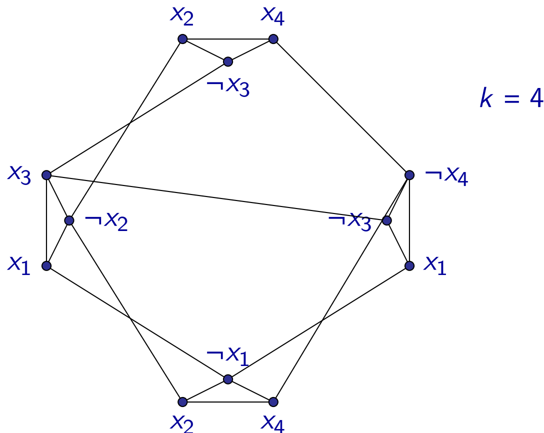
$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$



Dvojice vrcholů odpovídající literálům x_i a $\neg x_i$ spojíme hranami.

Převod 3-SAT na IS

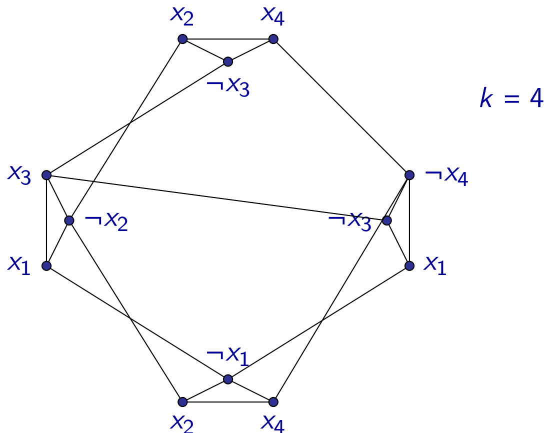
$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$



Číslo k položíme rovno počtu klauzulí.

Převod 3-SAT na IS

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$

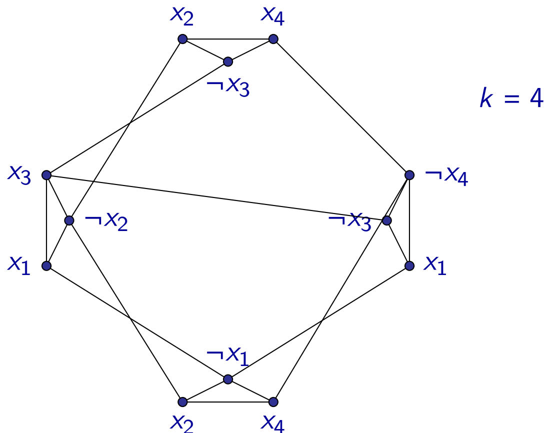


Vytvořený graf a číslo k vydá algoritmus jako výstup.

Převod 3-SAT na IS

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$

$\nu(x_1) = 1$
$\nu(x_2) = 1$
$\nu(x_3) = 0$
$\nu(x_4) = 1$

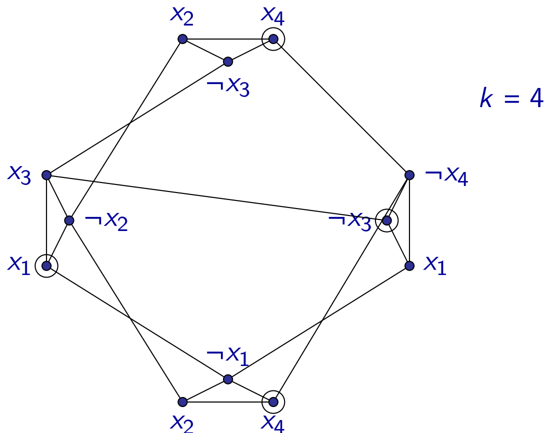


Jestliže je formule φ splnitelná, existuje ohodnocení ν, při kterém má v každé klauzuli alespoň jeden literál hodnotu 1.

Převod 3-SAT na IS

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$

$\nu(x_1) = 1$
$\nu(x_2) = 1$
$\nu(x_3) = 0$
$\nu(x_4) = 1$

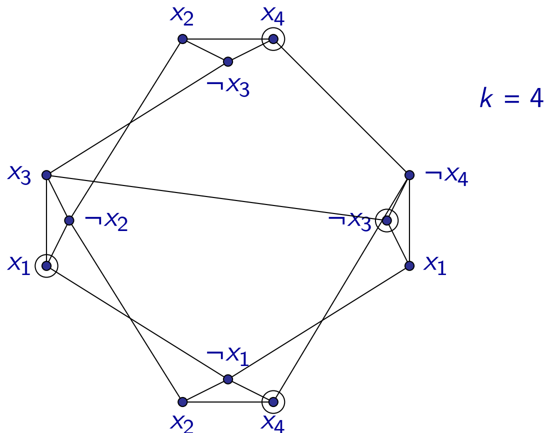


Z každé klauzule vybereme jeden literál, který má při ohodnocení ν hodnotu 1, a do nezávislé množiny přidáme odpovídající vrchol.

Převod 3-SAT na IS

$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$

$\nu(x_1) = 1$
$\nu(x_2) = 1$
$\nu(x_3) = 0$
$\nu(x_4) = 1$



$k = 4$

Lehce ověříme, že vybrané vrcholy tvoří nezávislou množinu.

Převod 3-SAT na IS

Vybrané vrcholy tvoří nezávislou množinu, protože:

- Z každé trojice vrcholů odpovídající jedné klauzuli byl vybrán jen jeden vrchol.
- Nemohly být současně vybrány vrcholy označené x_i a $\neg x_i$.
  (Při daném ohodnocení ν má hodnotu 1 jen jeden z nich.)

Převod 3-SAT na IS

Na druhou stranu, pokud v grafu G existuje nezávislá množina velikosti k, musí určitě splňovat následující vlastnosti:

- Z každé trojice vrcholů odpovídající jedné klauzuli musí být vybrán nejvýše jeden vrchol.

  Protože je ale klauzulí k a je vybráno k vrcholů, musí být z každé takové trojice vybrán právě jeden.

- Nemohly být současně vybrány vrcholy označené x_i a $\neg x_i$.

Ohodnocení tedy zvolíme podle vybraných vrcholů, protože z předchozího vyplývá, že nehrozí, že by neexistovalo.
(Zbylým proměnným přiřadíme libovolné hodnoty.)

Při daném ohodnocení má formule φ určitě hodnotu 1, neboť v každé klauzuli má hodnotu 1 alespoň jeden literál.

Převod 3-SAT na IS

Popsaný algoritmus je určitě polynomiální:

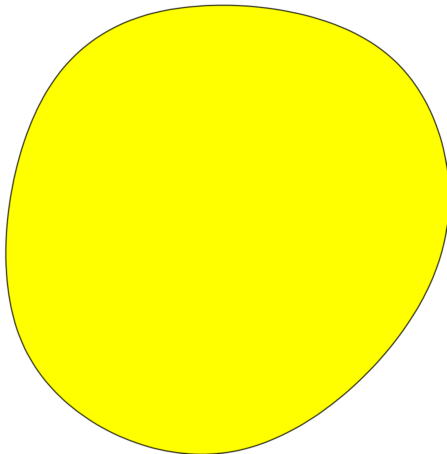
Graf G a číslo k je možné zkonstruovat k formuli φ v čase $\mathcal{O}(n^2)$, kde n je velikost formule φ.

Navíc jsme viděli, že ve zkonstruovaném grafu G existuje nezávislá množina velikosti k právě tehdy, když formule φ je splnitelná.

Popsaný algoritmus tedy ukazuje, že problém 3-SAT je polynomiálně převeditelný na problém IS.

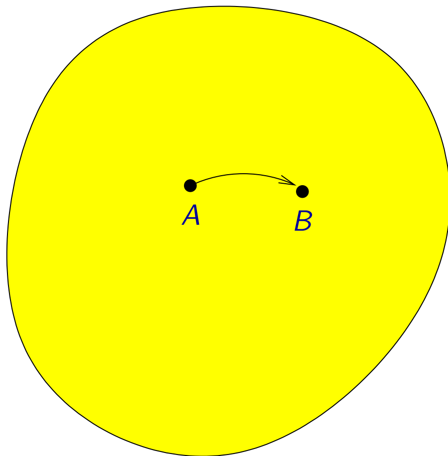
NP-úplné problémy

Vezměme si množinu všech možných rozhodovacích problémů.



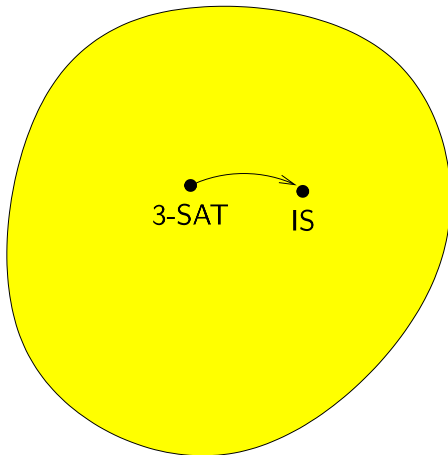
NP-úplné problémy

Šipkou si znázorníme, že problém A je polynomiálně převeditelný na problém B.



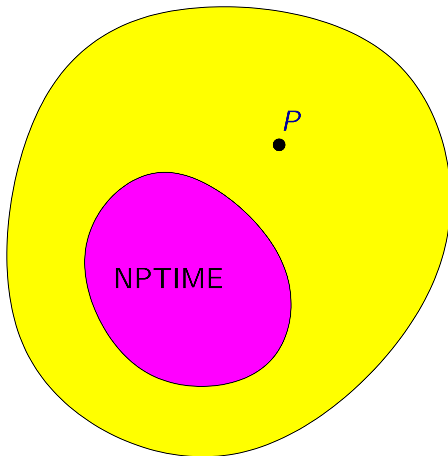
NP-úplné problémy

Například problém 3-SAT je polynomiálně převeditelný na problém IS.



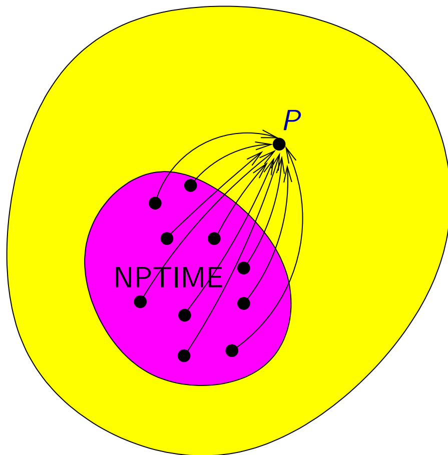
NP-úplné problémy

Vezměme si nyní třídu NPTIME a nějaký problém P.



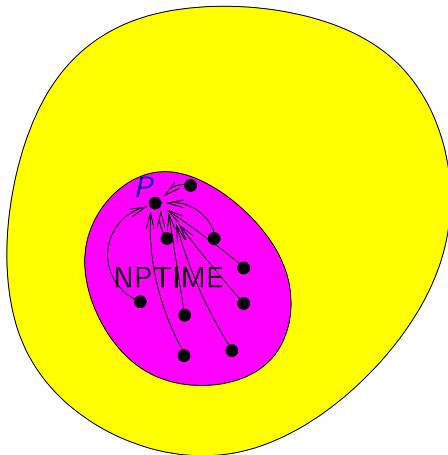
NP-úplné problémy

Problém *P* je NP-**těžký**, jestliže každý problém z NPTIME je polynomiálně převeditelný na *P*.



NP-úplné problémy

Problém P je **NP-úplný**, jestliže je NP-těžký a navíc sám patří do třídy NPTIME.



NP-úplné problémy

Pokud bychom pro nějaký NP-těžký problém P nalezli polynomiální algoritmus, získali bychom tím polynomiální algoritmus pro každý problém P' z NPTIME:

- Na vstup problému P' bychom nejprve aplikovali algoritmus realizující polynomiální převod z P' na P.
- Na vytvořenou instanci problému P bychom aplikovali polynomiální algoritmus řešící problém P a výsledek bychom vrátili jako odpověď pro danou instanci problému P'.

V takovém případě by tedy platilo PTIME = NPTIME, neboť pro každý problém z NPTIME by existoval polynomiální (deterministický) algoritmus.

NP-úplné problémy

Na druhou stranu, pokud existuje alespoň jeden problém z NPTIME, pro který neexistuje polynomiální algoritmus, tak z předchozího plyne, že pro žádný NP-těžký problém nemůže existovat polynomiální algoritmus.

Zda platí první nebo druhá možnost, je otevřený problém.

NP-úplné problémy

Není těžké si rozmyslet následující:

Pokud je problém A polynomiálně převeditelný na problém B a problém B je polynomiálně převeditelný na problém C, pak problém A je polynomiálně převeditelný na problém C.

Pokud tedy o nějakém problému P víme, že je NP-těžký a že P je polynomiálně převeditelný na problém P', pak víme, že i problém P' je NP-těžký.

NP-úplné problémy

Věta (Cook, 1971)

Problém SAT je NP-úplný.

Dá se ukázat, že SAT je polynomiálně převeditelný na 3-SAT a viděli jsme, že 3-SAT je polynomiálně převeditelný na IS.

Z toho plyne, že problémy 3-SAT a IS jsou NP-těžké.

Není také těžké ukázat, že 3-SAT i IS patří do třídy NPTIME.

Problémy 3-SAT i IS jsou NP-úplné.

Příklady některých NP-úplných problémů

Ze zatím uvedených problémů jsou NP-úplné následující tři problémy:

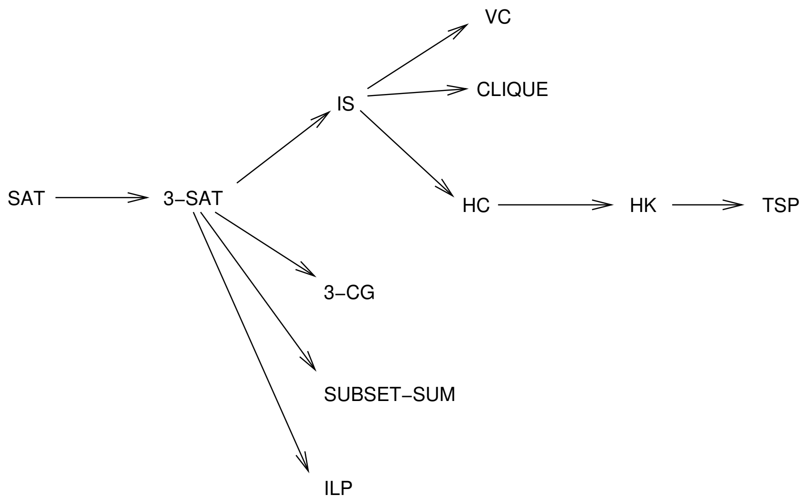
- SAT (splnitelnost booleovských formulí)
- 3-SAT
- IS — problém nezávislé množiny (independent set)

Na následujících slidech jsou uvedeny některé další NP-úplné problémy:

- CG — vrcholové barvení grafu (pozn.: je NP-úplný i ve speciálním případě, kdy máme právě 3 barvy)
- VC — vrcholové pokrytí grafu (vertex cover)
- CLIQUE — problém kliky
- HC — problém Hamiltonovského cyklu
- HK — problém Hamiltonovské kružnice
- TSP — problém obchodního cestujícího (traveling salesman problem)
- SUBSET-SUM
- ILP — celočíselné lineární programování (integer linear programming)

Příklady některých NP-úplných problémů

NP-obtížnost těchto problémů je možné ukázat tak, že ukážeme polynomiální převody z již známých NP-úplných problémů:



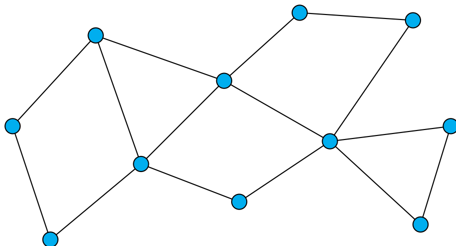
Barvení grafu

Barvení grafu

Vstup: Neorientovaný graf G, přirozené číslo k.

Otázka: Lze vrcholy grafu G obarvit k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

Příklad: $k = 3$



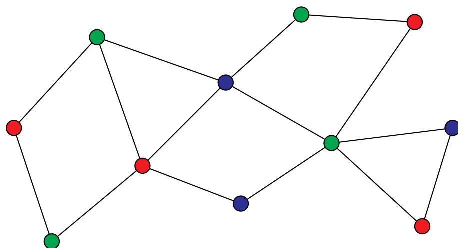
Barvení grafu

Barvení grafu

Vstup: Neorientovaný graf G, přirozené číslo k.

Otázka: Lze vrcholy grafu G obarvit k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

Příklad: $k = 3$



Odpověď: Ano

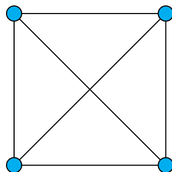
Barvení grafu

Barvení grafu

Vstup: Neorientovaný graf G, přirozené číslo k.

Otázka: Lze vrcholy grafu G obarvit k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

Příklad: $k = 3$



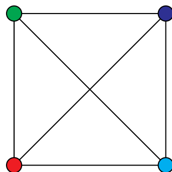
Barvení grafu

Barvení grafu

Vstup: Neorientovaný graf G, přirozené číslo k.

Otázka: Lze vrcholy grafu G obarvit k barvami tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu?

Příklad: $k = 3$



Odpověď: NE

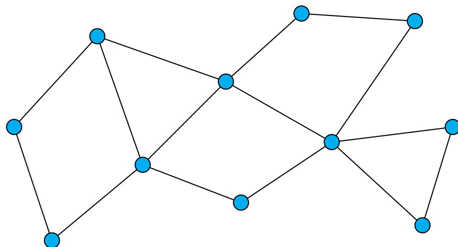
VC – Vrcholové pokrytí

VC – vrcholové pokrytí (vertex cover)

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Existuje v grafu G množina vrcholů velikosti k taková, že každá hrana má alespoň jeden svůj vrchol v této množině?

Příklad: $k = 6$



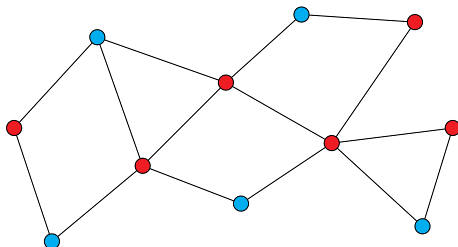
VC – Vrcholové pokrytí

VC – vrcholové pokrytí (vertex cover)

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Existuje v grafu G množina vrcholů velikosti k taková, že každá hrana má alespoň jeden svůj vrchol v této množině?

Příklad: $k = 6$



Odpověď: Ano

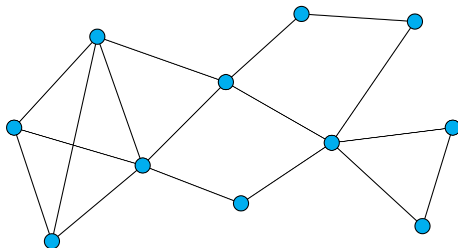
CLIQUE – problém kliky

CLIQUE – problém kliky

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Existuje v grafu G množina vrcholů velikosti k taková, že každé dva vrcholy této množiny jsou spojeny hranou?

Příklad: $k = 4$



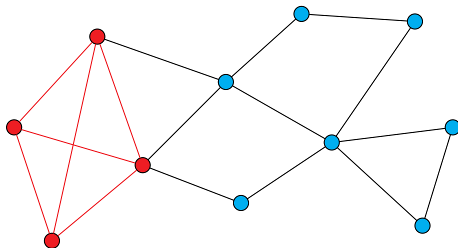
CLIQUE – problém kliky

CLIQUE – problém kliky

Vstup: Neorientovaný graf G a přirozené číslo k.

Otázka: Existuje v grafu G množina vrcholů velikosti k taková, že každé dva vrcholy této množiny jsou spojeny hranou?

Příklad: $k = 4$



Odpověď: Ano

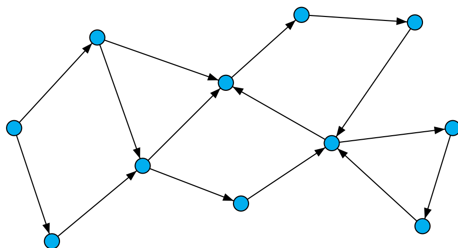
Hamiltonovský cyklus

HC – Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovský cyklus (orientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



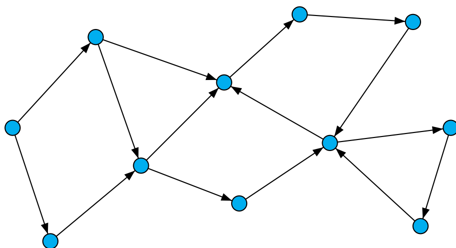
Hamiltonovský cyklus

HC – Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovský cyklus (orientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



Odpověď: NE

Hamiltonovský cyklus

HC – Problém „Hamiltonovský cyklus"

Vstup: Orientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovský cyklus (orientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:

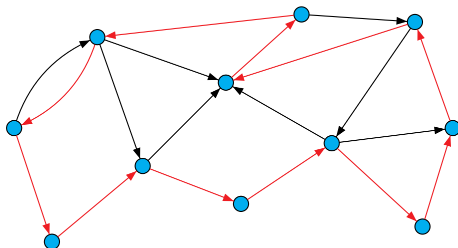
Hamiltonovský cyklus

HC – Problém „Hamiltonovský cyklus“

Vstup: Orientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovský cyklus (orientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



Odpověď: Ano

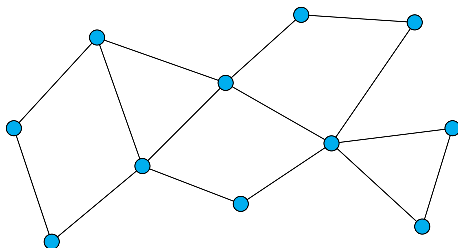
Hamiltonovská kružnice

HK – Problém „Hamiltonovská kružnice“

Vstup: Neorientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovská kružnice (neorientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



Odpověď: NE

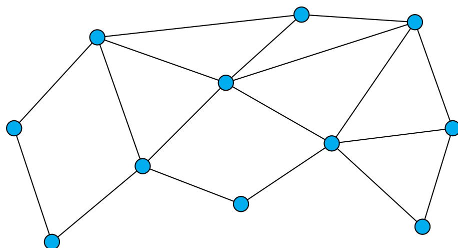
Hamiltonovská kružnice

HK – Problém „Hamiltonovská kružnice“

Vstup: Neorientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovská kružnice (neorientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:



Hamiltonovská kružnice

HK – Problém „Hamiltonovská kružnice“

Vstup: Neorientovaný graf G.

Otázka: Existuje v grafu G Hamiltonovská kružnice (neorientovaný cyklus procházející každým vrcholem právě jednou)?

Příklad:

Odpověď: Ano

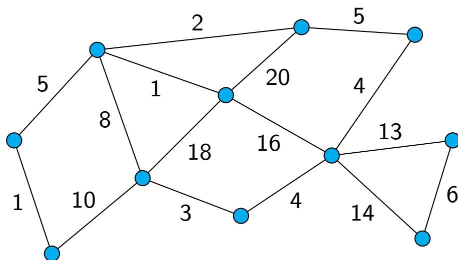
Problém obchodního cestujícího

TSP - Problém „obchodního cestujícího“

Vstup: Neorientovaný graf G s hranami ohodnocenými přirozenými čísly a číslo k.

Otázka: Existuje v grafu G uzavřená cesta procházející všemi vrcholy takový, že součet délek hran na této cestě (včetně opakovaných) je maximálně k?

Příklad: $k = 70$



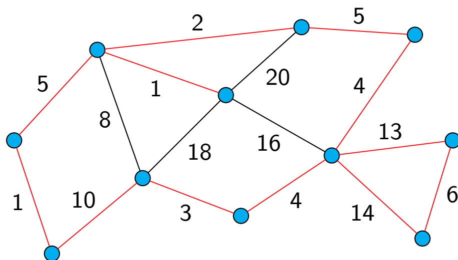
Problém obchodního cestujícího

TSP - Problém „obchodního cestujícího“

Vstup: Neorientovaný graf G s hranami ohodnocenými přirozenými čísly a číslo k.

Otázka: Existuje v grafu G uzavřená cesta procházející všemi vrcholy taková, že součet délek hran na této cestě (včetně opakovaných) je maximálně k?

Příklad: $k = 70$



Odpověď: Ano, protože byla nalezena cesta se součtem 69.

SUBSET-SUM

Problém SUBSET-SUM

Vstup: Sekvence přirozených čísel $a_1, a_2, \ldots, a_n$ a přirozené číslo s.

Otázka: Existuje množina $I \subseteq \{1, 2, \ldots, n\}$ taková, že $\sum_{i \in I} a_i = s$?

Jinak řečeno, ptáme se zda z dané (multi)množiny čísel je možné vybrat podmnožinu, jejíž součet je s.

Příklad: Pro vstup tvořený čísly $3, 5, 2, 3, 7$ a číslem $s = 15$ je odpověď ANO, neboť $3 + 5 + 7 = 15$.

Pro vstup tvořený čísly $3, 5, 2, 3, 7$ a číslem $s = 16$ je odpověď NE, neboť žádná podmnožina těchto čísel nedává součet 16.

SUBSET-SUM

Poznámka:
Pořadí čísel $a_1, a_2, \ldots, a_n$ na vstupu není důležité.

Všimněte si však určitého rozdílu oproti tomu, kdybychom problém formulovali tak, že vstupem je množina $\{a_1, a_2, \ldots, a_n\}$ a číslo s — v množině se čísla neopakují, zatímco v sekvenci se může totéž číslo vyskytnout vícekrát.

SUBSET-SUM

Problém SUBSET-SUM je speciálním případem **problému batohu** (knapsack problem):

Knapsack problem

Vstup: Sekvence dvojic přirozených čísel $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ a dvě přirozená čísla s a t.

Otázka: Existuje množina $I \subseteq \{1, 2, \ldots, n\}$ taková, že $\sum_{i \in I} a_i \leq s$ a $\sum_{i \in I} b_i \geq t$?

SUBSET-SUM

Neformálně můžeme problém batohu formulovat takto:

Máme n předmětů, kde i-tý předmět váží a_i gramů a má cenu b_i Kč. Do batohu se vejdou předměty o maximální celkové váze s gramů.

Otázka zní, zda můžeme z předmětů vybrat podmnožinu, která by vážila maximálně s gramů a měla celkovou cenu alespoň t Kč.

Poznámka:
Zde jsme problém batohu formulovali jako rozhodovací problém.

Běžnější je formulovat tento problém jako optimalizační problém, kde je cílem najít takovou množinu $I \subseteq \{1, 2, \ldots, n\}$, kde hodnota $\sum_{i \in I} b_i$ je maximální, přičemž ovšem musí být dodržena podmínka $\sum_{i \in I} a_i \leq s$, tj. vybrat předměty s maximální celkovou cenou tak, aby nebyla překročena kapacita batohu.

SUBSET-SUM

To, že SUBSET-SUM je speciálním případem problému batohu, vidíme z následující jednoduché konstrukce:

Řekněme, že $a_1, a_2, \ldots, a_n$, s_1 je instance problému SUBSET-SUM. Je očividné, že pro instanci problému batohu, kde máme sekvenci (a_1, a_1), $(a_2, a_2), \ldots,$ (a_n, a_n), $s = s_1$ a $t = s_1$, je odpověď stejná jako pro původní instanci SUBSET-SUM.

SUBSET-SUM

Pokud chceme studovat složitost problémů jako jsou SUBSET-SUM nebo problém batohu, je dobré si nejprve ujasnit, co považujeme za velikost vstupu.

Asi nejpřirozenější je definovat velikost vstupu jako celkový počet bitů, který potřebujeme k zápisu instance.

Musíme však určit, jakým způsobem jsou na vstupu zadána přirozená čísla – zda binárně (případně v jiné číselné soustavě o základu alespoň 2, např. desítkové nebo šestnáctkové) nebo unárně.

- Pokud počítáme velikost vstupu jako celkový počet bitů při použití **binárního** zápisu čísel, tak je problém SUBSET-SUM NP-úplný (a není tedy pro něj znám polynomiální algoritmus).
- Pokud počítáme velikost vstupu jako celkový počet bitů při použití **unárního** zápisu, tak existuje pro problém SUBSET-SUM algoritmus s polynomiální časovou složitostí.

ILP – celočíselné lineární programování

Problém ILP (celočíselné lineární programování)

Vstup: Celočíselná matice A a celočíselný vektor b.

Otázka: Existuje celočíselný vektor x, takový že $Ax \leq b$?

Příklad instance problému:

$$A = \begin{pmatrix} 3 & -2 & 5 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix} \qquad b = \begin{pmatrix} 8 \\ -3 \\ 5 \end{pmatrix}$$

Ptáme se tedy, zda existuje celočíselné řešení následující soustavy nerovnic:

$$\begin{aligned} 3x_1 - 2x_2 + 5x_3 &\leq 8 \\ x_1 + x_3 &\leq -3 \\ 2x_1 + x_2 &\leq 5 \end{aligned}$$

ILP – celočíselné lineární programování

Jedním z řešení soustavy

$$\begin{aligned} 3x_1 - 2x_2 + 5x_3 &\leq 8 \\ x_1 + x_3 &\leq -3 \\ 2x_1 + x_2 &\leq 5 \end{aligned}$$

je například $x_1 = -4$, $x_2 = 1$, $x_3 = 1$, tj.

$$x = \begin{pmatrix} -4 \\ 1 \\ 1 \end{pmatrix}$$

neboť

$$\begin{aligned} 3 \cdot (-4) - 2 \cdot 1 + 5 \cdot 1 &= -9 \leq 8 \\ -4 + 1 &= -3 \leq -3 \\ 2 \cdot (-4) + 1 &= -7 \leq 5 \end{aligned}$$

Pro tuto instanci je tedy odpověď Ano.

ILP – celočíselné lineární programování

Poznámka: Analogický problém, kdy se pro danou soustavu lineálních nerovnic ptáme, zda existuje její řešení v oboru **reálných** čísel, je možné řešit v polynomiálním čase.