Nerozhodnutelné problémy

Algoritmicky řešitelné problémy

Předpokládejme, že máme dán nějaký problém P.

Jestliže existuje nějaký algoritmus, který řeší problém P, pak říkáme, že problém P je **algoritmicky řešitelný**.

Jestliže P je rozhodovací problém a jestliže existuje nějaký algoritmus, který problém P řeší, pak říkáme, že problém P je **(algoritmicky) rozhodnutelný**.

Když chceme ukázat, že problém P je algoritmicky řešitelný, stačí ukázat nějaký algoritmus, který ho řeší (a případně ukázat, že daný algoritmus problém P skutečně řeší).

Algoritmicky neřešitelné problémy

Problém, který není algoritmicky řešitelný, je **algoritmicky neřešitelný**.

Rozhodovací problém, který není rozhodnutelný, je **nerozhodnutelný**.

Kupodivu existuje řada algoritmických problémů (přesně definovaných), o kterých je dokázáno, že nejsou algoritmicky řešitelné.

Halting Problem

Vezměme si nějaký libovolný obecný programovací jazyk $\mathcal{L}$.

Navíc předpokládejme, že programy v jazyce $\mathcal{L}$ běží na nějakém idealizovaném stroji, kde mají k dispozici (potenciálně) neomezené množství paměti — tj. kde alokace paměti nikdy neselže kvůli nedostatku paměti.

Příklad: Následující problém zvaný **Problém zastavení (Halting problem)** je nerozhodnutelný:

Halting problem

Vstup: Zdrojový kód programu P v jazyce $\mathcal{L}$, vstupní data x.

Otázka: Zastaví se program P po nějakém konečném počtu kroků, pokud dostane jako vstup data x?

Halting Problem

Předpokládejme, že by existoval nějaký program, který by rozhodoval Halting problem.

Mohli bychom tedy vytvořit podprogram H, deklarovaný jako

Bool H(String kod, String vstup)

kde $H(P, x)$ vrátí:

- true pokud se program P zastaví pro vstup x,
- false pokud se program P nezastaví pro vstup x.

Poznámka: Řekněme, že podprogram $H(P, x)$ by vracel false v případě, že P není syntakticky správný kód programu.

Halting Problem

S použitím podprogramu H bychom vytvořili program D, který bude provádět následující kroky:

- Načte svůj vstup do proměnné x typu String.
- Zavolá podprogram $H(x, x)$.
- Pokud podprogram H vrátil true, skočí do nekonečné smyčky

```
loop: goto loop
```

V případě, že H vrátil false, program D se ukončí.

Co udělá program D, pokud mu předložíme jako vstup jeho vlastní kód?

Halting Problem

Pokud D dostane jako vstup svůj vlastní kód, tak se buď zastaví nebo nezastaví.

- Pokud se D zastaví, tak $H(D, D)$ vrátí true a D skočí do nekonečné smyčky. Spor!
- Pokud se D nezastaví, tak $H(D, D)$ vrátí false a D se zastaví. Spor!

V obou případech dospějeme ke sporu a další možnost není. Nemůže tedy platit předpoklad, že H řeší Halting problem.

Částečně rozhodnutelné problémy

Problém je **částečně rozhodnutelný**, jestliže existuje algoritmus, který:

- Pokud dostane jako vstup instanci, pro kterou je odpověď ANO, tak se po konečném počtu kroků zastaví a vypíše `"ANO"`.
- Pokud dostane jako vstup instanci, pro kterou je odpověď NE, tak se buď zastaví a vypíše `"NE"` nebo se nikdy nezastaví.

Je očividné, že například HP (Halting problem) je částečně rozhodnutelný.

Některé problémy však nejsou ani částečně rozhodnutelné.

Postova věta

Doplňkový problém k danému rozhodovacímu problému P je problém, kde vstupy jsou stejné jako u problému P a otázka je negací otázky z problému P.

Postova věta

Jestliže problém P i jeho doplňkový problém jsou částečně rozhodnutelné, pak je problém P rozhodnutelný.

Převody mezi problémy

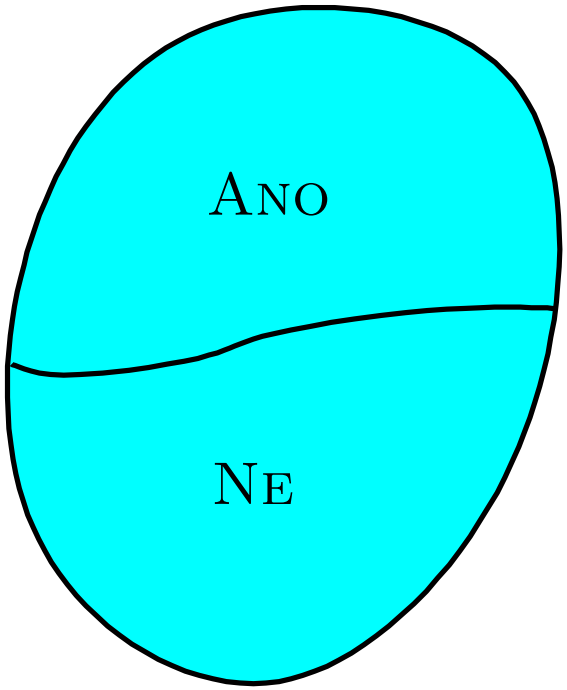
Pokud máme o nějakém (rozhodovacím) problému dokázáno, že je nerozhodnutelný, můžeme ukázat nerozhodnutelnost dalších problémů pomocí redukcí (převodů) mezi problémy.

Problém P_1 je **převeditelný** na problém P_2, jestliže existuje algoritmus Alg takový, že:

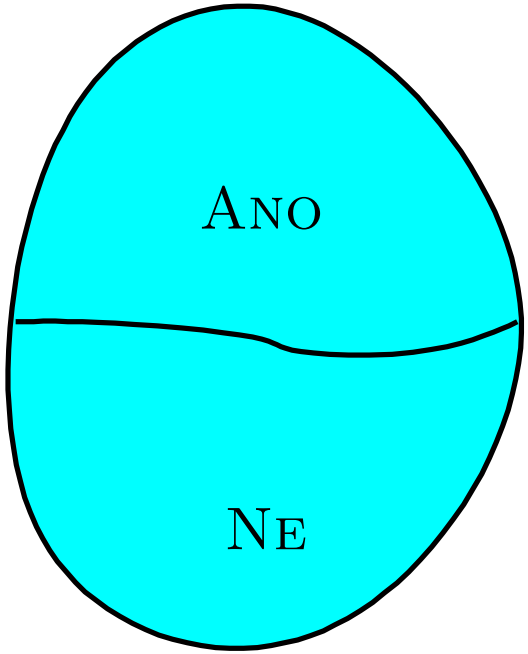
- Jako vstup může dostat libovolnou instanci problému P_1.
- K instanci problému P_1, kterou dostane jako vstup (označme ji w), vyprodukuje jako svůj výstup instanci problému P_2 (označme ji $Alg(w)$).
- Platí, že pro vstup w je v problému P_1 odpověď ANO právě tehdy, když pro vstup $Alg(w)$ je v problému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

Převody mezi problémy

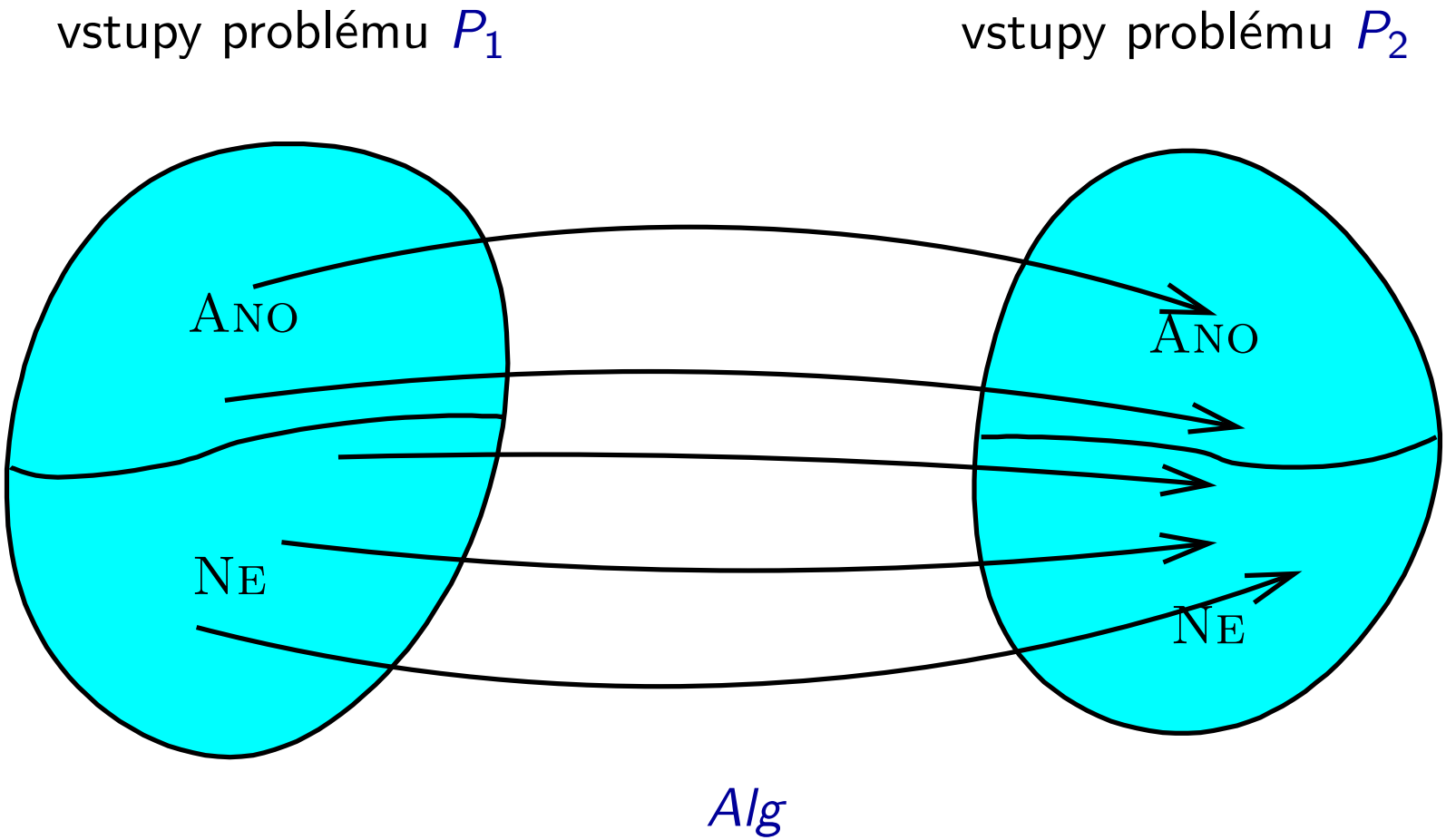
vstupy problému P_1



vstupy problému P_2



Převody mezi problémy



Převody mezi problémy

Řekněme, že existuje redukce Alg problému P_1 na problém P_2.

Pokud by problém P_2 byl rozhodnutelný, pak i problém P_1 je rozhodnutelný.

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.

Je zřejmé, že pokud P_1 je nerozhodnutelný, tak P_2 nemůže být rozhodnutelný.

Další nerozhodnutelné problémy

Redukcí z Halting problému se dá ukázat nerozhodnutelnost celé řady problémů, které se týkají ověřování chování programů:

- Vydá daný program pro nějaký vstup odpověď ANO?
- Zastaví se daný program pro libovolný vstup?
- Dávají dva dané programy pro stejné vstupy stejný výstup?
- ...

Halting problem

Pro účely důkazů se Halting problem nejčastěji používá v následující podobě:

Halting problem

Vstup: Popis Turingova stroje $\mathcal{M}$ a slovo w.

Otázka: Zastaví se stroj $\mathcal{M}$ po nějakém konečném počtu kroků, pokud dostane jako svůj vstup slovo w?

Další nerozhodnutelné problémy

S následujícím příkladem nerozhodnutelného problému už jsme se setkali:

Problém

Vstup: Bezkontextové gramatiky $\mathcal{G}_1$ a $\mathcal{G}_2$.

Otázka: Je $\mathcal{L}(\mathcal{G}_1) = \mathcal{L}(\mathcal{G}_2)$?

případně

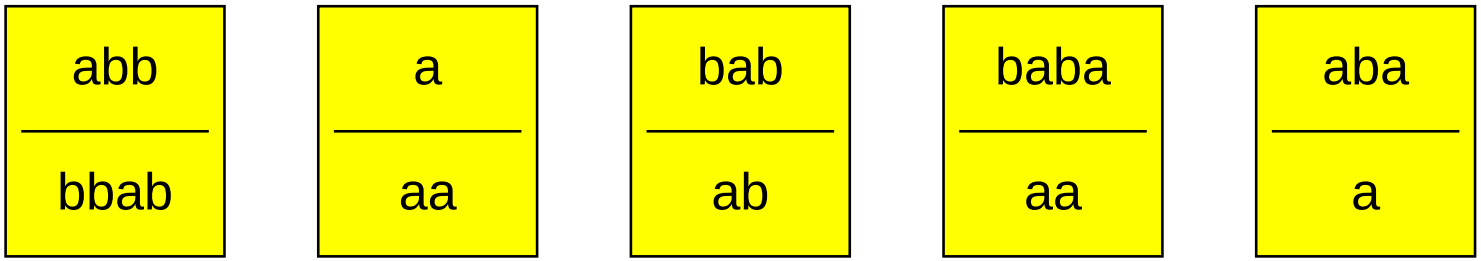
Problém

Vstup: Bezkontextová gramatika $\mathcal{G}$ generující jazyk nad abecedou Σ.

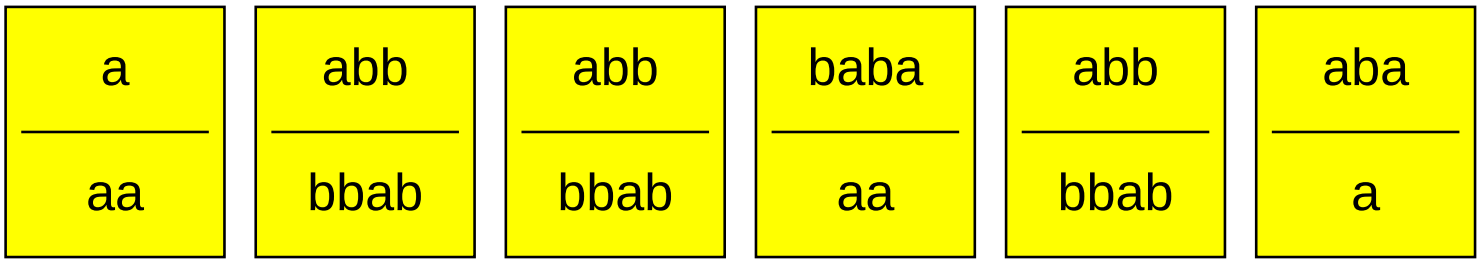
Otázka: Je $\mathcal{L}(\mathcal{G}) = \Sigma^*$?

Další nerozhodnutelné problémy

Vstupem je množina typů kartiček, jako třeba:



Otázka je, zda je možné z těchto typů kartiček vytvořit neprázdnou konečnou posloupnost, kde zřetězením slov nahoře i dole vznikne totéž slovo. Každý typ kartičky je možné používat opakovaně.



Nahoře i dole vznikne slovo `aabbabbbabaabbaba`.

Další nerozhodnutelné problémy

Redukcí z předchozího problému se dá snadno ukázat nerozhodnutelnost některých dalších problémů z oblasti bezkontextových gramatik:

Problém

Vstup: Bezkontextové gramatiky $\mathcal{G}_1$ a $\mathcal{G}_2$.

Otázka: Je $\mathcal{L}(\mathcal{G}_1) \cap \mathcal{L}(\mathcal{G}_2) = \varnothing$?

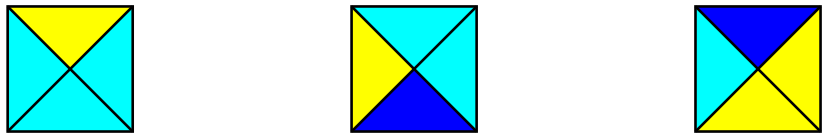
Problém

Vstup: Bezkontextová gramatika $\mathcal{G}$.

Otázka: Je $\mathcal{G}$ nejednoznačná?

Další nerozhodnutelné problémy

Vstupem je množina typů kachliček, jako třeba:

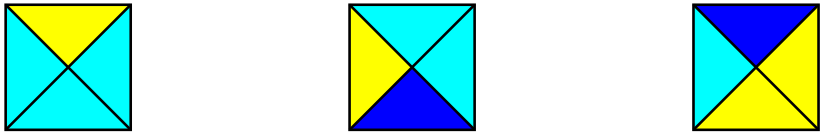


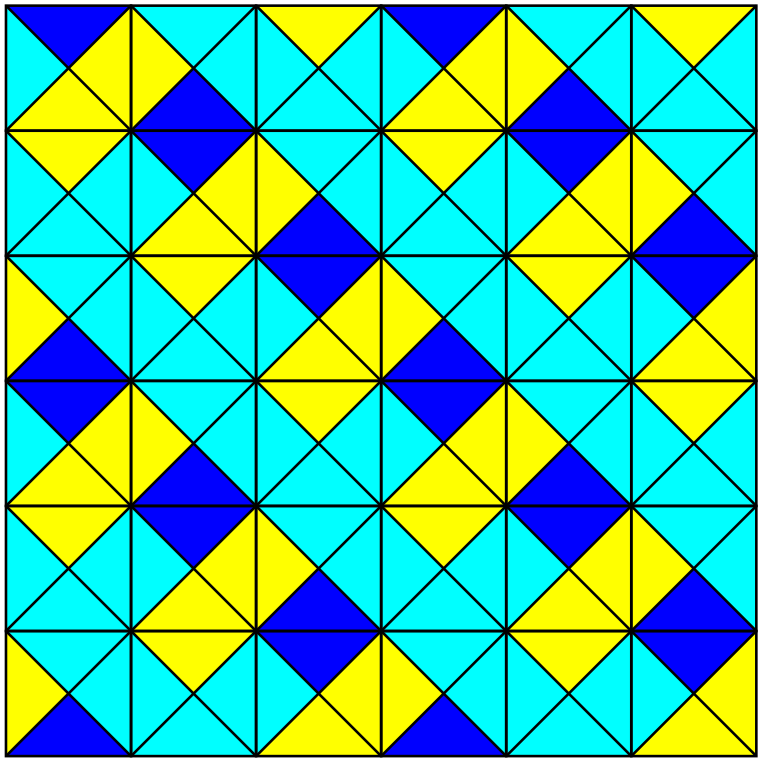
Otázka je, zda je možné použitím daných typů kachliček pokrýt každou libovolně velkou konečnou plochu tak, aby všechny kachličky spolu sousedily stejnými barvami.

Poznámka: Můžeme předpokládat, že máme v zásobě neomezené množství kachliček všech typů.

Kachličky není dovoleno otáčet.

Další nerozhodnutelné problémy





Další nerozhodnutelné problémy

Problém

Vstup: Uzavřená formule predikátové logiky (prvního řádu), ve které mohou být použity jako predikátové symboly pouze $=$ a $<$, jako funkční symboly pouze $+$ a $*$ a jako konstantní symboly pouze 0 a 1.

Otázka: Je daná formule pravdivá v oboru přirozených čísel (při přirozené interpretaci všech funkčních a predikátových symbolů)?

Příklad vstupu:

$$\forall x \exists y \forall z((x * y = z) \wedge (y + 1 = x))$$

Poznámka: Úzce souvisí s Gödelovou větou o neúplnosti.

Další nerozhodnutelné problémy

Je zajímavé, že analogický problém, kde ale místo přirozených čísel uvažujeme čísla reálná, je algoritmicky rozhodnutelný (i když popis daného algoritmu a důkaz jeho korektnosti jsou značně netriviální).

Rovněž pokud uvažujeme přirozená nebo celá čísla a stejné formule jako v předchozím případě, ale s tím, že v nich nesmí být použit funkční symbol $*$ (násobení), tak je problém algoritmicky rozhodnutelný.

Další nerozhodnutelné problémy

Pokud můžeme používat $*$, je ve skutečnosti nerozhodnutelný už velmi omezený případ:

Desátý Hilbertův problém

Vstup: Polynom $f(x_1, x_2, \ldots, x_n)$ vytvořený z proměnných $x_1, x_2, \ldots, x_n$ a celočíselných konstant.

Otázka: Existují přirozená čísla $x_1, x_2, \ldots, x_n$ taková, že $f(x_1, x_2, \ldots, x_n) = 0$?

Příklad vstupu: $5x^2y - 8yz + 3z^2 - 15$

Tj. ptáme se, zda

$$\exists x \exists y \exists z (5 * x * x * y + (-8) * y * z + 3 * z * z + (-15) = 0)$$

platí v oboru přirozených čísel.

Další nerozhodnutelné problémy

Také následující problém je algoritmicky nerozhodnutelný:

Problém

Vstup: Uzavřená formule φ predikátové logiky prvního řádu.

Otázka: Platí $\models \varphi$?

Poznámka: Zápis $\models \varphi$ znamená, že formule φ je logicky platná, tj. pravdivá v každé interpretaci.

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 $\mathcal{T}(f(n))$ jako třídu obsahující právě ty rozhodovací problémy, pro něž existuje algoritmus s časovou složitostí $O(f(n))$.

Příklad:

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

Třídy složitosti

Definice

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

Příklad:

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

Třídy složitosti

Poznámka:

Všimněte si, že u tříd $\mathcal{T}(f)$ a $\mathcal{S}(f)$ 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 $\mathcal{T}(f(n))$ a $\mathcal{S}(f(n))$ můžeme definovat třídy PTIME a PSPACE jako

$$\text{PTIME} = \bigcup_{k \geq 0} \mathcal{T}(n^k) \qquad \text{PSPACE} = \bigcup_{k \geq 0} \mathcal{S}(n^k)$$

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

Třídy složitosti

Poznámka: 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.

Třídy složitosti

Analogicky můžeme zavést další třídy:

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

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

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

Poznámka: Místo $2^{O(n^k)}$ bychom mohli psát také $O(c^{n^k})$, 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 $2^{2^{O(n^k)}}$, kde k je nějaká konstanta

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

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

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

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 $O(f(n))$, má tento algoritmus prostorovou složitost (nejvýše) $O(f(n))$.

Je tedy zřejmé, že platí následující vztah.

Pozorování

Pro libovolnou funkci $f : \mathbb{N} \to \mathbb{N}$ platí $\mathcal{T}(f(n)) \subseteq \mathcal{S}(f(n))$.

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

- Pro libovolná dvě reálná čísla ϵ_1 a ϵ_2 taková, že $0 \leq \epsilon_1 < \epsilon_2$, platí

$$\mathcal{S}(n^{\epsilon_1}) \subsetneq \mathcal{S}(n^{\epsilon_2})$$

- LOGSPACE $\subsetneq$ PSPACE
- PSPACE $\subsetneq$ EXPSPACE

- Pro libovolná dvě reálná čísla ϵ_1 a ϵ_2 taková, že $0 \leq \epsilon_1 < \epsilon_2$, platí

$$\mathcal{T}(n^{\epsilon_1}) \subsetneq \mathcal{T}(n^{\epsilon_2})$$

- PTIME $\subsetneq$ EXPTIME
- EXPTIME $\subsetneq$ 2-EXPTIME

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 $O(f(n))$, znamená to, že počet různých konfigurací dosažitelných během výpočtu je $2^{O(f(n))}$.

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

Pozorování

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

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

Vztahy mezi třídami složitosti

Z předchozího plynou následující důsledky:

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

Vztahy mezi třídami složitosti

Shrnutí:

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

- $\text{PTIME} \subsetneq \text{EXPTIME} \subsetneq \text{2-EXPTIME} \subsetneq \cdots$
- $\text{LOGSPACE} \subsetneq \text{PSPACE} \subsetneq \text{EXPSPACE} \subsetneq \text{2-EXPSPACE}, \subsetneq \cdots$

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 dosažitenosti 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 složist 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 nejmémě $cn \log n$ operací).

Nedeterministické algoritmy a třídy složitosti

Nedeterminismus

Nedeterministický stroj RAM:

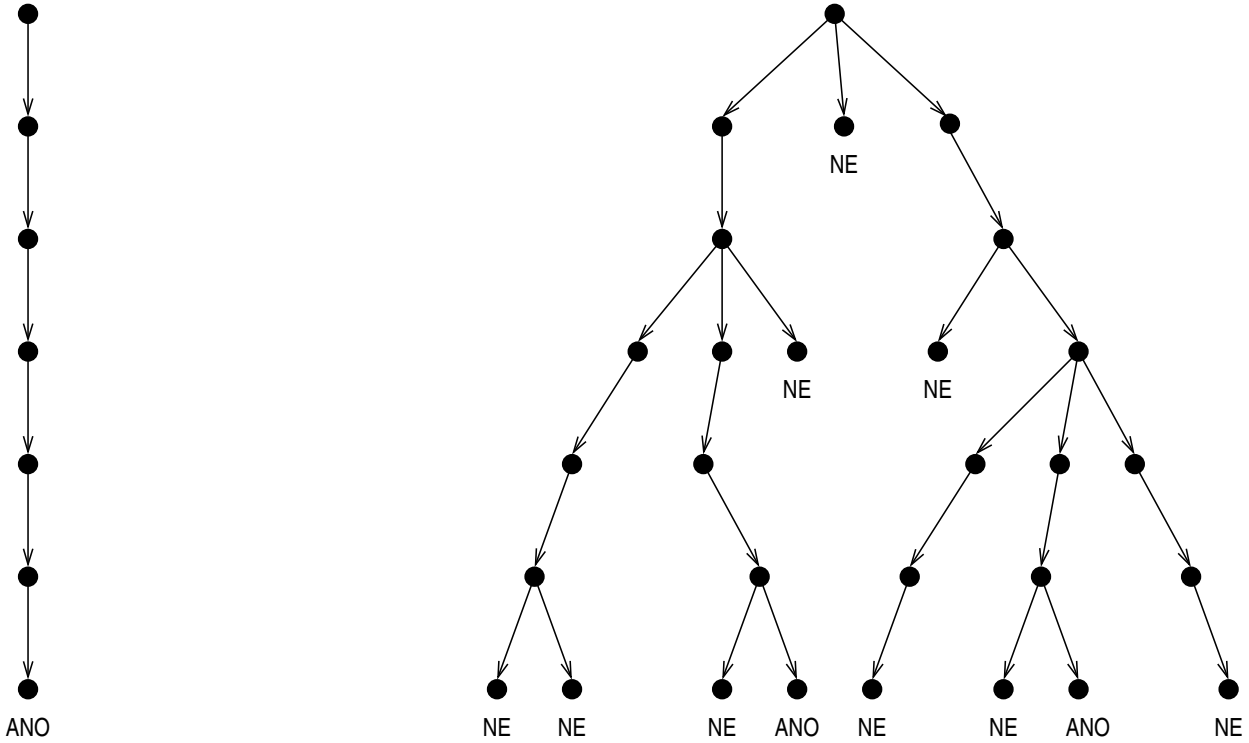
- Je definován velice podobně jako deterministický RAM.
- Navíc má instrukci

  nd_goto ℓ_1, ℓ_2

  která umožňuje stroji vybrat si jedno z možných pokračování.
- 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.

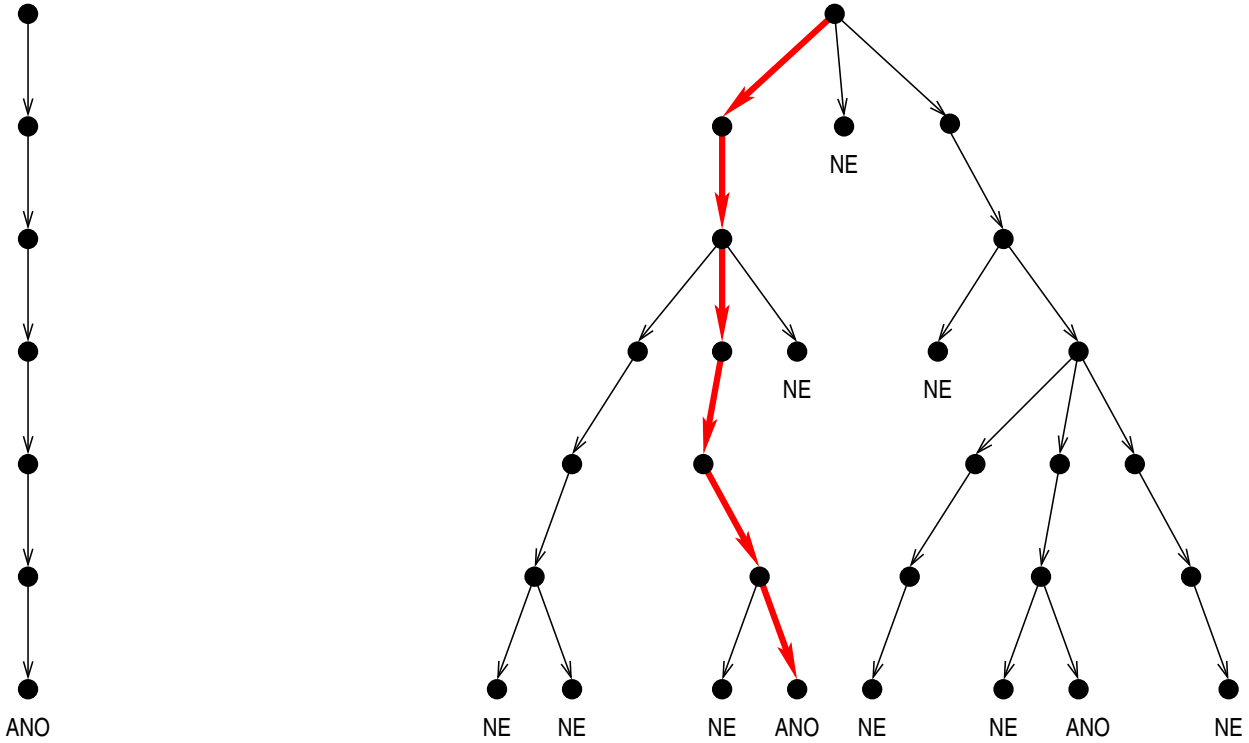
Podobně můžeme definovat nedeterministické verze jiných výpočetních modelů, např. nedeterministické Turingovy stroje.

Nedeterminismus



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

Nedeterminismus



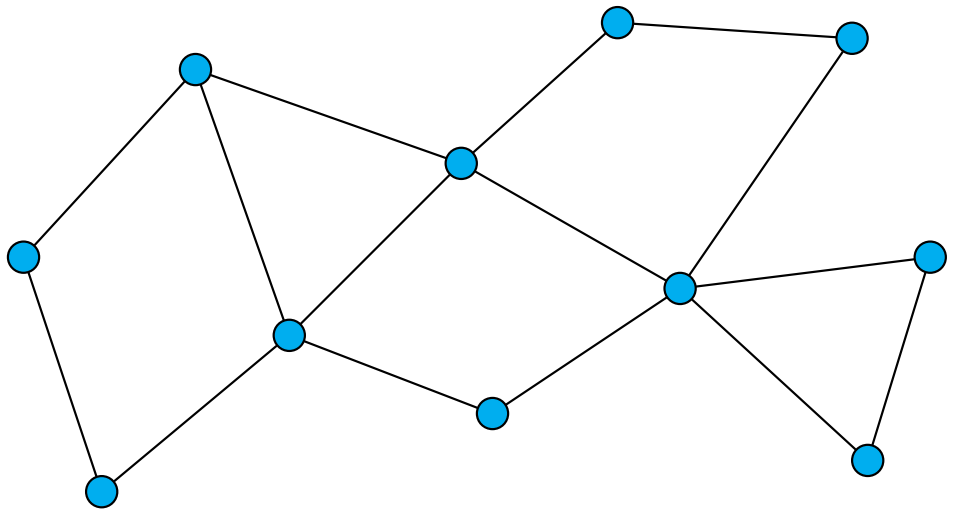
- Doba výpočtu nedeterministického stroje RAM (nebo jiného nedeterministického stroje) nad zadaným vstupem je definována jako délka nejdelšího možného výpočtu nad tímto vstupem.

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?



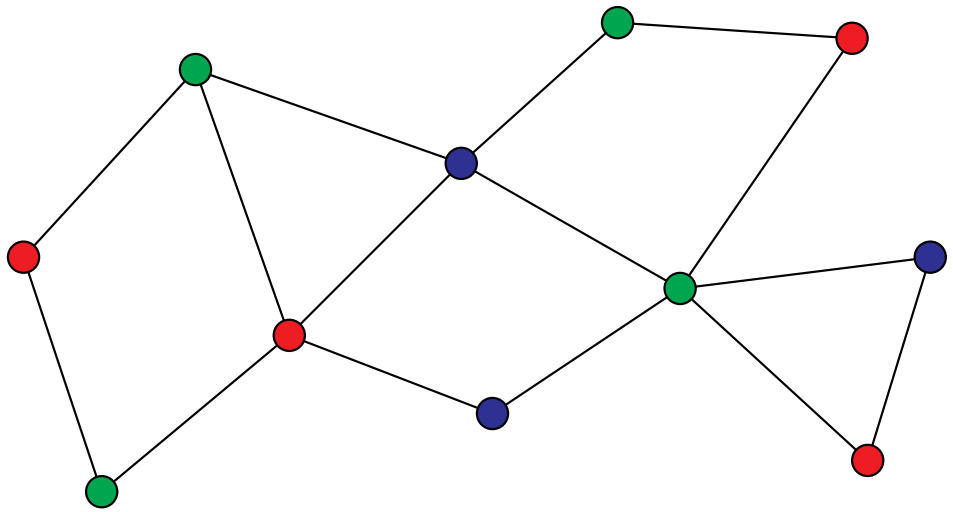
$k = 3$

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 zkotroluje, 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

- 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** $W(x)$, přičemž právě tehdy, když x má vlastnost P, tak existuje nějaký **skutečný svědek** $y \in W(x)$ toho, že x tuto vlastnost P skutečně má.

  Vezměme si **deterministický** algoritmus Alg, který jako vstup dostane dvojici (x, y) (kde $y \in W(x)$) 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 $c : V \to \{1, \ldots, k\}$.
- *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 $y \in W(x)$.
  - 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 deterministiky 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 poměrně komplikované 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 $f : \mathbb{N} \to \mathbb{N}$ rozumíme **třídou časové složitosti** $\mathcal{NT}(f)$ množinu těch rozhodovacích problémů, které jsou řešeny nedeterministickými RAMy s časovou složitostí v $O(f(n))$.

Definice

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

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

Třída NPTIME

Definice

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

- 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 $2^{O(n^k)}$, kde k je nějaká konstanta

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

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

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:

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

Vztahy mezi třídami složitosti

Vezměme si nějaký **nedeterministický** algoritmus s **časovou** složitostí $O(f(n))$.

Deteriministický algoritmus, který bude simulovat jeho činnost tím způsobem, že bude systematicky procházet všechny jeho výpočty (procházením stromu těchto výpočtů do hloubky), vystačí s následující pamětí:

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

- Vzhledem k tomu, že délka větví je $O(f(n))$, množství potřebné paměti pro zásobník je $O(f(n))$.
- Celkově tedy deterministický algoritmus při této simulaci vystačí s množstvím paměti, které je nejvýše $O(f(n))$.

Z výše uvedeného tedy vyplývá:

$$\begin{aligned} \text{NPTIME} &\subseteq \text{PSPACE} \\ \text{NEXPTIME} &\subseteq \text{EXPSPACE} \\ &\vdots \end{aligned}$$

Vztahy mezi třídami složitosti

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

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

  (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 $2^{2^{O(f(n))}}$.

Vztahy mezi třídami složitosti

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

- **vrcholy** — všechny konfigurace simulovaného stroje, jejichž velikost je nejvýše $O(f(n))$
  — těchto konfigurací je $2^{O(f(n))}$
- **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 $2^{O(f(n))}$

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.

Vztahy mezi třídami složitosti

Pro zjištění existence takové cesty je možné použít libovolný algoritmus na procházení grafu — procházení do šířky, 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. $2^{O(f(n))}$.

Vztahy mezi třídami složitosti

Dostáváme tedy následující:

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

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 $O(f(n))$. Teď nám ale pro změnu půjde o co nejmenší **prostorovou** složitost simulujícího deterministického algoritmu.

Věta (Savitch, 1970)

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

Myšlenka důkazu:

- Opět si představme výše popsaný graf konfigurací, který má $2^{O(f(n))}$ 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 $O(f(n))$
  - hodnotu čísla i, které je řádově $O(f(n))$ — proto na jeho uložení stačí zhruba $O(\log F(n))$ bitů
  - další pomocné proměnné, jejichž hodnoty jsou proti velikosti výše uvedených položek zanedbatelné
- Množství paměti potřebné v rámci jednoho rekurzivního volání je tedy $O(f(n))$.
- Hloubka zanoření rekurze je také $O(f(n))$.
- Celková prostorová složitost daného algoritmu je tedy $O(f(n)^2)$.

Vztahy mezi třídami složitosti

Z výše uvedené věty vyplývá:

$$\begin{aligned} \mathrm{NPSPACE} &\subseteq \mathrm{PSPACE} \\ \mathrm{NEXPSPACE} &\subseteq \mathrm{EXPSPACE} \\ &\vdots \end{aligned}$$

Spolu s triviálními fakty, že $\mathrm{PSPACE} \subseteq \mathrm{NPSPACE}$, $\mathrm{EXPSPACE} \subseteq \mathrm{NEXPSPACE}$, . . . nám to tedy dává:

$$\begin{aligned} \mathrm{PSPACE} &= \mathrm{NPSPACE} \\ \mathrm{EXPSPACE} &= \mathrm{NEXPSPACE} \\ &\vdots \end{aligned}$$

Poznámka: Všimněte si, že z výše uvedeného **nevyplývá**, že by muselo platit $\mathrm{LOGSPACE} = \mathrm{NLOGSPACE}$.

Vztahy mezi třídami složitosti

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

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