Složitost

Upozornění: tento text doplňuje slidy k přednášce a neobsahuje vše, co bylo odpřednášeno.

Luboš Brim
Katedra teorie programování
Fakulta Informatiky
Masarykova universita Brno

Prosinec 2005

Časová složitost

Přednáška č.10

1.1 Měření časové složitosti

Uvažujme libovolný rozhodnutelný problém a odpovídající rozhodovací algoritmus. Počet kroků, které algoritmus použije pro daný vstup může záviset na několika parametrech. Je-li například vstupem graf, pak počet kroků může záviset na počtu vrcholů, na počtu hran, na stupni grafu či na kombinaci těchto a jiných faktorů. Pro jednoduchost budeme čas, který algoritmus spotřebuje pro řešení zadané úlohy, chápat jako funkci délky řetězce reprezentujícího vstup a nebudeme uvažovat žádné další parametry. V případě *analýzy nejhoršího případu* uvažujeme nejdelší čas výpočtu potřebný pro všechny vstupy určité délky. V případě *analýzy průměrného případu* uvažujeme průměr přes všechny výpočty na vstupech určité délky.

Definice 1.1 *Nechť M je daný deterministický TS (DTS), který zastaví pro každý vstup.* Časová složitost *stroje M je funkce $f : \mathbb{N} \to \mathbb{N}$ definovaná takto:*

$$f(n) = \max\{Steps_M(x) \mid |x| = n\}$$

kde $Steps_M(x)$ je počet kroků, který provede stroj M pro vstup x a $|x|$ je velikost vstupu.

Protože vyjádření přesné časové složitosti TS je poměrně komplikované, často se spokojíme s jistým odhadem. To je navíc i užitečné pro porovnávání Turingových strojů z hlediska jejich časové složitosti. Jedním z nejčastěji používaných odhadů je *asymptotická analýza*, při které zkoumáme časovou složitost TS pro velké vstupy.

Definice 1.2 *Nechť $f, g : \mathbb{N} \to \mathbb{R}^+$. Řekneme, že f* je řádu *g (neroste asymptoticky rychleji než), píšeme $f \in \mathcal{O}(g)$ nebo $f = \mathcal{O}(g)$, právě když existují konstanty $c, n_0 \in \mathbb{N}^+$ takové, že*

$$\forall n \geq n_0 : f(n) \leq cg(n)$$

$f \equiv g$ (jsou stejného řádu) *právě když $g \in \mathcal{O}(f)$ a $f \in \mathcal{O}(g)$.*

Příklad 1.3 *Nechť $f_1(n) = 5n^3 + 2n^2 + 22n + 6$. Zvolme $c = 6$ a $n_0 = 10$. Snadno se ukáže, že pro všechna $n \geq 10$ platí $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ a tedy $f_1 = \mathcal{O}(n^3)$. Rovněž platí, že $f_1 = \mathcal{O}(n^4)$ neboť $n^4 \geq n^3$ pro všechna $n \in \mathbb{N}$. Na druhé straně však neplatí, že $f_1 = \mathcal{O}(n^2)$.*

Výraz $f \in \mathcal{O}(g)$ znamená, že f není asymptoticky větší než g. K vyjádření toho, že f je asymptoticky *menší* než g používáme označení $f \in o(g)$

Definice 1.4 *Nechť $f, g : \mathbb{N} \to \mathbb{R}^+$. Jestliže*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

pak řekneme, že g roste asymptoticky rychleji než *f a píšeme $f(n) = o(g(n))$.*

1.2 Třídy složitosti

Nyní zavedeme základní notaci pro klasifikaci problémů z hlediska jejich časové složitosti. Jedná se o analogii tříd nerozhodnutelnosti. Zatímco pro klasifikaci problémů z hlediska jejich rozhodnutelnosti byla podstatná existence algoritmu a bylo jedno, který konkrétní algoritmus z nekonečně mnoha možností máme na mysli, je situace u složitostní klasifikace komplikovanější.

Uvažujme následující algoritmus (TS) pro rozhodování jazyka $A = \{0^k 1^k \mid k \geq 0\}$.

M_1 = Pro vstupní řetězec w:
1. Prohlédni pásku a jestliže nějaká 0 leží napravo od 1, pak *zamítni*,
2. Jestliže páska obsahuje 0 i 1, pak opakovaně:
3. Projdi pásku a označ jednu 0 a jednu 1.
4. Jestliže na pásce zůstane neoznačená 0 nebo 1, pak *zamítni*, jinak *akceptuj*.

Analyzujme počet kroků, které provede M_1 pro vstup w délky n. Instrukce 1 vyžaduje průchod přes celou pásku a tedy n kroků. Pro návrat hlavy na začátek pásky je potřeba rovněž n kroků. Tato instrukce tedy vyžaduje $\mathcal{O}(n)$ kroků. Všimněme si, že jsme v popisu algoritmu neuvedli požadavek na přesun hlavy na počátek pásky. Asymptotická analýza právě umožňuje takovéto detaily vynechat, neboť ovlivňují délku výpočtu nejvýše o konstantní faktor. V části 2 a 3 prochází stroj opakovaně přes pásku a označí vždy jednu 0 a jednu 1. Každý průchod vyžaduje $\mathcal{O}(n)$ kroků. Protože při každém průchodu jsou označeny dva symboly, je potřeba provést nejvý3e $n/2$ těchto průchodů. Celkový čas potřebný k provedení instrukcí 2 a 3 je tedy $(n/2)\mathcal{O}(n) = \mathcal{O}(n^2)$. V poslední instrukci provede stroj pouze jeden průchod přes pásku, což vyžaduje $\mathcal{O}(n)$ kroků. Celková časová složitost stroje M_1 je $\mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$.

Definice 1.5 *Nechť* $t : \mathbb{N} \to \mathbb{N}$. *Množina*

$$\mathsf{TIME}(t(n)) = \{L \mid \textit{existuje DTS } M, \textit{jehož časová složitost je } \mathcal{O}(t(n)) \textit{ a } L = \mathcal{L}(M)\}$$

se nazývá třídou časové složitosti.

Víme, že $A = \{0^k 1^k \mid k \geq 0\} \in \mathsf{TIME}(n^2)$. Je otázkou, zda jazyk A lze rozhodovat asymptoticky rychleji. Řečeno jinak, zda $A \in \mathsf{TIME}(t(n))$ pro $t(n) \in o(n^2)$? Výpočet stroje M_1 lze zrychlit tak, že v jednom průchodu budeme označovat dvojice nul a jedniček namísto jedné dvojice. Tím zmenšíme počet průchodů na polovinu, což je ale pouze konstantní faktor a asymtptická složitost zůstane stejná. Následující algoritmus používá jinou metodu pro rozhodování jazyka A s asymtpoticky lepší složitostí.

M_2 = Pro vstupní řetězec w:
1. Prohlédni pásku a jestliže nějaká 0 leží napravo od 1, pak *zamítni*,
2. Jestliže páska obsahuje 0 i 1, pak opakovaně:
3. Projdi pásku a zjisti, zda celkový počet neoznačených 0 a 1 je sudý nebo lichý. Je-li lichý, pak *zamítni*.
4. Projdi opět pásku a označ každou další 0 počínaje první 0 a poté označ každou další 1 počínaje první 1.
5. Jestliže na pásce nezůstane neoznačená 0 nebo 1, pak *akceptuj*. Jinak *zamítni*.

Předložený algoritmus skutečně rozpoznává jazyk A. Při každém průchodu páskou v instrukci 4 je počet neoznačených 0 zmenšen na polovinu a zbytek odstraněn. Jestliže tedy

začneme s 13 nulami, pak po prvním průchodu jich zbude 6, po dalším 3, pak 1 a nakonec žádná. Totéž platí i pro jedničky.

Nyní prozkoumejme chování instrukce 3. Uvažujme situaci, kdy začínáme se 13 nulami a 13 jedničkami. První provedení instrukce 3 zjistí lichý počet 0 a lichý počet jedniček. V následující iteraci se objeví sudý počet (6), poté lichý (3) a poté opět lichý (1). Pro nulový počet 0 a 1 se iterace neprovádí. Jestliže získanou posloupnost parit *lichý, sudý, lichý, lichý* zapíšeme obráceně a výraz *lichý* zapíšeme jako "1" a *sudý* jako "0", dostaneme řetězec 1101, který je binární reprezentací čísla 13 (počtu nul či jedniček na počátku). V instrukci 3 je test na sudost celkového počtu nul a jedniček ve skutečnosti realizován jako test na shodu parit nul a parit jedniček. Jestliže se všechny parity shodují, je binární reprezentace počtu nul a jedniček stejná a tedy jsou tato čísla shodná.

Nyní přistupme k analýze časové složitosti stroje M_2. Nejprve si všimněme, že každá instrukce vyžaduje čas $\mathcal{O}(n)$. Stačí tedy určit počet provedení instrukcí. Instrukce 1 a 5 jsou provedeny pouze jednou. V instrukci 4 je označena alespoň polovina 0 a 1 při každém jejím provedení. Je tedy potřeba nejvýše $1 + \log_2 n$ iterací pro označení všech symbolů. Celkový čas potřebný pro provedení instrukcí 2,3 a 4 je proto $(1 + \log_2 n)\mathcal{O}(n)$, což je $\mathcal{O}(n \log n)$.

Dříve jsme ukázali, že $A \in \mathsf{TIME}(n^2)$. Nyní jsme tento výsledek zlepšili na $A \in \mathsf{TIME}(n \log n)$. Tento výsledek nelze dále zlepšit na jednopáskovém TS. Ve skutečnosti každý jazyk, který lze rozhodovat v čase $o(n \log n)$ na jednopáskovém TS je regulární.

Jazyk A lze rozhodovat v čase $\mathcal{O}(n)$ (v *lineárním* čase), jestliže použijeme druhou pásku.

M_3 = Pro vstupní řetězec w:

1. Prohlédni pásku a jestliže nějaká 0 leží napravo od 1, pak *zamítni*,
2. Procházej pásku 1 až do prvního výskytu sumbolu 1 a současně kopíruj 0 na pásku 2.
3. Procházej přes symboly 1 na pásce 1 až do konce vstupu. Pro každou jedničku označ jednu 0 na pásce 2. Jsou-li na pásce 2 označeny všechny 0 dříve, než byly přečteny všechny jedničky, pak *zamítni*.
4. Jestliže zůstane neoznačená 0, pak *zamítni*. Jinak *akceptuj*.

Předchozí analýza ukázala, že časová složitost problému A na jednopáskovém TS je $\mathcal{O}(n \log n)$ zatímco na vícepáskovém TS je $\mathcal{O}(n)$ a že tedy závisí na zvoleném výpočetním modelu. To je zásadní rozdíl oproti teorii vyčíslitelnosti. Church-Turingova teze totiž říká, že všechny rozumné výpočetní modely jsou ekvivalentní. V teorii složitosti tomu tak tedy není a volba modelu ovlivňuje výpočetní složitost problému. Jestliže chceme klasifikovat problémy v závislosti na jejich výpočetní náročnosti, pak se zdá být nemožné zvolit jediný model. Naštěstí není situace tak zcela beznadějná. Ukazuje se totiž, že např. časová složitost problému se neliší příliš pro typické deterministické výpočetní modely. Pokud tedy naše klasifikace nebude příliš citlivá k těmto rozdílům, nebude volba konkrétního determinstckého modelu hrát zásadní roli.

1.3 Vliv modelu na výpočetní složitost

V této části se budeme zabývat tím, jak volba výpočetního modelu ovlivňuje časovou složitost problémů. Budeme uvažovat tři modely: jednopáskový deterministický TS, vícepáskový deterministický TS a nedeterministický TS.

Věta 1.6 *Nechť $t(n)$ je funkce taková, že $t(n) \geq n$. Pak pro každý vícepaskový TS časové složitosti $t(n)$ existuje ekvivalentní jednopáskový TS časové složitosti $\mathcal{O}(t^2(n))$.*

DŮKAZ: Nechť M je k-páskový TS složitosti $t(n)$. Sestrojíme jednopáskový TS S časové složitosti $\mathcal{O}(t^2(n))$, který akceptuje stejný jazyk jako M. Stroj S simuluje činnost stroje M. Stroj S na své pásce uchovává obsah všech k pásek stroje M a pozice hlav. Pásky jsou zapsány za sebou, odděleny symbolem # a pozice příslušných hlav je vyznačena tečkou nad čteným symbolem.

Stroj S nejprve na pásku zapíše počáteční obsahy pásek stroje M a poté začne simulovat jednotlivé kroky stroje M. Simulace jednoho kroku vyžaduje průchod páskou k určení symbolů, které jsou pod hlavami stroje M. Pak S provede další průchod k provedení odpovídajících změn v symbolech a pozicích hlav. Jestliže některá z hlav se má posunout doprava za poslední obsazené pole pásky, musí S přidělit této pásce více prostoru. To provede tak, že posune celý zbývající obsah své pásky o jeden symbol doprava.

Nyní analyzujme časovou složitost simulace. Pro každý krok stroje M udělá S dva průchody nad aktivním obsahem pásky. Aktivní páska stroje S je určena aktivními páskami stroje M. Každá aktivní část pásky stroje M může zabírat nejvíve $t(n)$ políček po provedení $t(n)$ výpočetních kroků stroje M. Stroj S tedy potřebuje k průchodu aktivní částí pásky $\mathcal{O}(t(n))$ kroků.

K simulaci každého kroku stroje M potřebuje stroj S provést dva průchody a případně k posunů obsahu pásky doprava o jedno políčko. K simulaci jednoho kroku je tedy potřeba $\mathcal{O}(t(n))$ kroků.

Počáteční nastavení obashu pásky stroje S vyžaduje $\mathcal{O}(t(n))$ kroků. Poté S simuluje každý z $t(n)$ kroků stroje M pomocí $\mathcal{O}(t(n))$ kroků. Tato část tedy vyžaduje $t(n) \times \mathcal{O}(t(n)) = \mathcal{O}(t^2(n))$ kroků. Celkově je potřeba $\mathcal{O}(t(n)) + \mathcal{O}(t^2(n)) = \mathcal{O}(t^2(n))$ kroků.

Předpokládali jsme že $t(n) \geq n$ (je nutné přečíst alespoň celý vstup) a proto je časová složitost stroje S rovna $\mathcal{O}(t^2(n))$. ∎

Definice 1.7 *Nechť M je daný* nedeterministický *TS (NTS) takový, že pro každý vstup má výpočetní strom všechny výpočetní cesty konečné.* Čas výpočtu *($RunTime_M(x)$) stroje M pro vstup $x \in \Sigma^*$ je roven* maximální *délce výpočetní cesty (t.j. maximálnímu počtu kroků, které provede stroj M pro vstup x).* Časová složitost *stroje M je funkce $Time_M : \mathbb{N} \to \mathbb{N}$ definovaná takto:*

$$Time_M(n) = \max\{RunTime_M(x) \mid |x| = n\}$$

Definice času výpočtu nedeterministickéh TS není myšlena jako míra pro měření složitosti výpočtu na skutečném zařízení. Jedná se o matematickou definici, která umožní klasifikaci poměrně důležité skupiny problémů, jak to uvidíme později.

Věta 1.8 *Nechť $t(n)$ je funkce taková, že $t(n) \geq n$. Pak pro každý nedeterministický jednopaskový TS časové složitosti $t(n)$ existuje ekvivalentní deterministický jednopáskový TS časové složitosti $2^{\mathcal{O}(t(n))}$.*

DŮKAZ: Nechť N je nedeterministický TS časové složitosti $t(n)$. Sestrojíme deterministický TS D, který simuluje N tak, že prohledává výpočetní strom stroje N. Analyzujme složitost simulace.

Pro každý vstup délky n má každá větev výpočetního stromu délku nejvýše $t(n)$. Každý uzel ve stromu může mít nejvýše b potomků, kde b je maximální počet korektních voleb daných přechodovou funkcí stroje N. Celkový počet listů ve stromu je tedy nejvýše $b^{t(n)}$.

Prohledávání stromu probíhá do šířky po úrovních. To znamená, že všechny uzly na úrovni d (vzdálené od kořene d) jsou navštíveny dříve než některý uzel na úrovni $d+1$. Celkový počet uzlů je menší než dvojnásobek maximálního počtu listů a můžeme ho omezit hodnotou $\mathcal{O}(b^{t(n)})$. Čas potřebný k průchodu od kořene k některému uzlu je $\mathcal{O}(t(n))$. Proto je celkový čas výpočtu stroje D roven $\mathcal{O}(t(n)b^{t(n)}) = 2^{\mathcal{O}(t(n))}$.

Poznamenejme ještě, že D m6e k simulaci použít více pásek. Podle Věty 1.6 však převod na jednopaskový stroj způsobí nejvýše kvadratickou ztrátu a $(2^{\mathcal{O}(t(n))})^2 = 2^{\mathcal{O}(2t(n))} = 2^{\mathcal{O}(t(n))}$. ∎

1.4 Třída P

Věty 1.6 a 1.8 ilustrují jeden významný rozdíl. Ukázali jsme, že je nejvýše *polynomický* rozdíl mezi časovou složitostí problémů měřenou na deterministických strojích s jednou páskou a s více páskami. Ukázali jsme ale také, že tento rozdíl je *exponenciální*, pokud uvažujeme deterministické a nedeterministické stroje.

Pro naše účely budeme polynomální rozdíl považovat za malý, zatímco exponenciální za velký. Jaké pro to máme důvody? Především "typická" exponenciální funkce roste podstatně více než "typická" polynomická funkce. Pro vstup délky $n = 1000$ (což není přehnaná velikost vstupu) potřebuje algoritmus o složitosti n^3 provést miliardu kroků a to je reálné. Na druhé straně by algoritmus o složitosti 2^n potřeboval provést více kroků než je atomů ve vesmíru – a to není reálné. Polynomické algoritmy jsou dostatečně rychlé pro mnohé úlohy, zatímco exponenciální algoritmy jsou jenom zřídka užitečné.

Exponenciální algoritmus se typicky objeví při řešení úlohy prohledáváním prostoru všech možných řešení, tzv. hlední *hrubou silou*. Například jeden ze způsobů, jak rozložit číslo na součin prvočinitelů je prozkoumávat všechny potencionální dělitele. Těch je ale exponenciálně mnoho a proto takový algoritmus bude exponenciální. Často je naštěstí možné se hrubé síle vyhnout tím, že lépe porozumíme řešenému problému, objevíme nějaké užitečné vlastnosti a na nich postavíme polynomický algoritmus.

Všechny rozumné deterministické výpočetní modely jsou *polynomicky ekvivalentní*. Každý z nich dokáže simulovat jiný pouze s polynomickým nárůstem časové složitosti. Nepokoušíme se ovšem definovat pojem *rozumný model* formálně. To, co máme na mysli, je dostatečně široký pojem, zahrnující všechny modely, které těsně aproximují výpočetní čas skutečných počítačů.

Od této chvíle se soustředíme na ty aspekty teorie časové složitosti, které nejsou ovlivněny polynomickým rozdílem v časové složitosti. To dovolí vybudovat poměrně robustní klasifikaci problémů, která nebude ovlivněna volbou konkrétního výpočetního modelu. To ovšem neznamená, že při hledání konkrétních algoritmů pro konkrétní problémy je polynomický rozdíl zanedbatelný. Zrychlení výpočtu programu na dvojnásobek, může mít velký význam. Naším cílem však není hledání konkrétních algoritmů, ale klasifikace problémů.

Definice 1.9 P *je třída jazyků (problémů), které jsou rozhodnutelné v polynomickém čase na deterministických jednopáskových Turingových strojích. T.j.*

$$\mathsf{P} = \bigcup_k \mathsf{TIME}(n^k).$$

Třída P sehraje v našem zkoumání významnou úlohu z těchto důvodů:

1. P je invariantem pro všechny výpočetní modely, které jsou polynomicky ekvivalentní deterministickým jednopáskovým TS a

2. P zhruba odpovídá třídě problémů, které jsou prakticky řešitelné na reálných počítačích.

První bod zdůrazňuje, že zařazení problému do třídy P není ovlivněno podrobnostmi výpočetního modelu, použitého pro klasifikaci problému. Druhý bod pak říká, že pro daný problém existuje algoritmus, jehož čas výpočtu je n^k pro nějakou konstantu k. Je-li tento algoritmus skutečně praktický závisí na k a na dané aplikaci. Pochopitelně lze algoritmus se složitostí n^{100} těžko považovat za praktický. Nicméně se ukázalo, že použití polynomické složitosti jako hranice mezi praktickou řešitelností a neřešitelností se jeví jako užitečné. Jestliže by se například podařilo pro některý problém, který dříve vyžadoval exponenciální algoritmus, ukázat, že patří do P, pak to bylo zpravidla odrazem zjištění nějaké klíčové vlastnosti a lepšího pochopení problému. Důsledkem pak zpravidla byla i postupná redukce složitosti, často až do úrovně praktické použitelnosti.

1.4.1 Příklady problémů z P

Uveďme nyní několik příkladů problémů, které patří do třídy P . Pro popis odpovídajících algoritmů použijeme modifikaci jazyka **while**-programů. To umožní abstrahovat od podrobností popisu detailů výpočtu, např. obsahy pásky či pohyb hlavy u TS. Při analýze však samozřejmě musíme zdůvodnit, že instrukce algoritmu jsou implementovatelné v polynomickém čase na rozumném deterministickém modelu.

Zvláštní pozornost zasluhuje otázka kódování. Výraz $\langle\alpha\rangle$ označuje zakódování objektu (či posloupnosti objektů) α do řetězce symbolů. Nebudeme však zpravidla specifikovat podrobnosti použitého kódování. Rozumný způsob pro kódování je takový, který kódování a dekodování do rozumného kódovacího systému realizuje v polynomickém čase. Běžně používané metody pro kódování grafů, automatů ap. jsou v tomto smyslu rozumné. Všimněme si však, že kódování přirozených čísel do unární notace (např. 13 je zakódováno jako 11111111111111) není rozumné neboť je exponenciálně větší než např. binární kódování.

Mnohé problémy, se kterými se v této přednášce setkáme, požadují kódování grafu. Jedno možné kódování je seznam vrcholů a hran. Jiné kódování je *matice sousednosti*, ve které prvek (i, j) je 1 když z vrcholu i je hrana do vrcholu j, jinak je hodnota 0. Při analýze času výpočtu algoritmu nad grafem je možné provést analýzu vzhledem k počtu vrcholů a ne vzhledem k velikosti reprezentace grafu. U rozumných kódování grafu je velikost reprezentace polynomická vzhledem k počtu vrcholů. Je-li tedy výsledkem analýzy algoritmu polynomický (exponenciální) čas vzhledem k počtu vrcholů, pak je takový i vzhledem k velikosti vstupu.

První problém se týká orientovaných grafů. Problém *PATH* spočívá v rozhodnutí, zda v grafu existuje orientovaná cesta mezi dvěma danými vrcholy. Přesněji

$$PATH = \{\langle G, s, t\rangle \mid \ G \text{ je orientovaný graf, ve které je orientovaná cesta z } s \text{ do } t\}.$$

Věta 1.10 $PATH \in \mathbf{P}$

Důkaz: Větu dokážeme tak, že předložíme polynomický algoritmus pro rozhodování *PATH*. Hlavní myšlenka algoritmu spočívá v postupném označování vrcholů, které jsou dosažitelné z s orientovanými cestami délek 1, 2, ... m.

```
1 proc M; vstup : ⟨G, s, t⟩
2   Označkuj vrchol s;
3   while byl označen další vrchol  do
4         Prohledej všechny hrany grafu G. Jestliže existuje hrana (a, b) z označe-
5         ného vrcholu a do neoznačeného vrcholu b, pak označ vrchol b.
6   od;
7   if t je označen  then akceptuj else zamítni fi
```

Nechť m je počet vrcholů grafu G. Instrukce 2 a 7 jsou provedeny pouze jednou. Instrukce 4-5 se provede nejvýše m krát neboť při každé iteraci (kromě poslední) je označen alespoň jeden další (neoznačený) vrchol. Celkem je tedy potřeba provést nejvýše $1+1+m$ výpočetních kroků a algoritmus je polynomický ve velikosti grafu G. ∎

K předchozímu příkladu ještě poznamenejme, že algoritmus založený na hrubé síle, t.j. takový, který by prozkoumal všechny potencionální cesty v grafu vede k exponenciální složitosti. Všech cest je totiž m^m.

Druhým příkladem je numerický problém. Řekneme, že dvě čísla jsou *relativní prvočísla* (jsou nesoudělná), jestliže 1 je největší celé číslo, které dělí obě čísla. Například 10 a 21 jsou relativní prvočísla, i když ani jedno z nich není prvočíslo. Na druhé straně 10 a 22 nejsou relativní prvočísla. Problem *RELPRIME* spočívá v rozhodnutí, zda dvě daná čísla jsou relativní prvočísla, t.j.

$$RELPRIME = \{\langle x, y\rangle \mid x \text{ a } y \text{ jsou relativní prvočísla}\}.$$

Věta 1.11 $RELPRIME \in \mathsf{P}$

Důkaz: Čísla x a y jsou relativní prvočísla právě když jejich největší společný dělitel je roven 1. K výpočtu největšího společného dělitele použijeme Euklidův algoritmus.

```
1 proc R; vstup: ⟨x, y⟩, kde x a y jsou přirozená čísla zapsaná binárně
2   repeat
3         x := x  mod y;
4         swap(x, y)
5   until y = 0;
6   if x = 1 then akceptuj else zamítni fi
```

Každý výpočet těla cyklu (instrukce 3-4) s případnou výjimkou první iterace zmenší hodnotu proměnné x alespoň na polovinu. Po provedení příkazu 3 je $x < y$ a po provedení příkazu 4 je $x > y$. Před opětovným provedením příkazu 3 je tedy $x > y$. Jestliže je $x/2 \geq y$, pak $x \mod y < y \leq x/2$ a x je zmenšeno alespoň na polovinu. Je-li $x/2 < y$, pak $x \mod y = x - y < x/2$ a opět je x zmenšeno alespoň na polovinu.

Protože jsou hodnoty proměnných vyměněny každým provedení příkazu 4, jsou obě hodnoty zmenšeny alespoň na polovinu při každé iteraci cyklu. Maximální počet iterací cyklu je roven minimu z hodnot $\log_2 x$ a $\log_2 y$. Tyto logaritmy jsou proporcionální k velikosti vstupu (čísla jsou reprezentována binárně) a tedy počet iterací je $\mathcal{O}(n)$. Každou intrukci lze realizovat polynomicky a tedy čas výpočtu algoritmu je polynomický. ∎

Připomeňme opět, že algoritmus, který by zkoumal všechny možné dělitele čísel x a y by měl exponenciální složitost.

Jako poslední příklad ukážeme, že každý bezkontextový jazyk lze rozhodovat v polynomickém čase.

Věta 1.12 *Každý bezkontextový jazyk patří do P.*

DŮKAZ: Důkaz využívá techniky *dynamického programování*. Tato technika je založena na akumulaci informace o menších podproblémech k řešení většího problému. Zaznamenáváme řešení každého podproblému a tím ho řešíme pouze jednou. V našem případě budou podproblémy spočívat ve zjištění, zda každá proměnná z gramatiky G generuje každý podřetězec řetězce w. Tyto výsledky budeme ukládat do matice $n \times n$. Pro $i \leq j$ obsahuje prvek (i, j) matice proměnné, které generují podřetězec $w_i, w_{i+1}, \ldots, w_j$. Pro $i > j$ je prvek nevyužit.

Algoritmus plní postupně matici hodnotami pro každý podřetězec řetězce w. Nejprve jsou přidány hodnoty pro podřetězce délky 1, pak délky 2, atd. Pro výpočet nové hodnoty jsou využity již vypočítané hodnoty pro kratší podřetězce. Předpokládejme například, že algoritmus již určil, které proměnné generují všechny podřetězce do délky k včetně. K určení toho, zda proměnná A generuje určitý podřetězec délky $k + 1$, rozdělí algoritmus podřetězec do dvou neprázdných částí všemi z možných k způsobů. Pro každé takové dělení, zkoumá algoritmus každé pravidlo $A \to BC$ ke zjištění, zda B generuje první část a C druhou a při tom využívá informace uložené v matici. Jestliže B i C generují odpovídající části, pak A generuje podřetěz a algoritmus uloží do matice zjištěnou informaci. Algoritmus začíná podřetězci délky 1 a iniciálním naplněním matice pomocí pravidel $A \to b$.

Nyní uvedeme podrobný algoritmus. Předpokládejme, že G je bezkontextová gramatika zadaná v Chomského normálním tvaru, která generuje jazyk L. Nechť S je počáteční symbol.

```
1  proc D; vstup: w = w_1 ... w_n
2    if w = ε ∧ S → ε ∈ P then akceptuj fi
3    for i = 1 to n do
4        Pro každou proměnnou A
5        if A → w_i ∈ P then put A in table(i,i) fi
6    od;
7    for l = 2 to n do
8        for i = 1 to n − l + 1 do
9            j := i + l − 1
10           for k = i to j − 1 do
11               Pro každé pravidlo A → BC
12               if table(i,k) obsahuje B ∧ table(k + 1, l) obsahuje C
13                  then put A in table(i,j) fi
14           od
15       od
16   od
17   if S ∈ table(1,n) then akceptuj else zamítni fi.
```

Každou instrukci algoritmu D je možné implementovat polynomicky. Příkaz 5 se provede nejvýše nv krát, kde v je počet proměnných v G. Hodnota v je konstanta, která nezávisí na n a tedy příkaz 5 je proveden $\mathcal{O}(n)$ krát. Cyklus 7 se provede nejvýše n krát. Pro každou iteraci cyklu 7 se nejvýš n krát provede cyklus 8 a pro každou iteraci cyklu 8 se nejvýše n krát provede cyklus 10. V každé iteraci cyklu 10 se nejvýše r krát provede příkaz 11, kde r je počet pravidel v G, což je další konstanta nezávislá na n. Příkaz 13, který je nejvíce vnořeným příkazem algoritmu, se tedy provede nejvýše $\mathcal{O}(n^3)$ krát a celková časová složitost algoritmu je $\mathcal{O}(n^3)$. ∎

1.5 Třída NP

Jak jsme viděli na příkladech v části 1.4.1, je možné v mnoha případech nahradit metodu hrubého násilí a získat polynomický algoritmus. To však asi neplatí obecně. V mnoha jiných případech, které zahrnují velice zajímavé a užitečné problémy, se totéž nepodařilo a nejsou známé polynomické algoritmy pro řešení těchto problémů.

Proč tomu tak je? Na tuto důležitou otázku neumíme odpovědět. Možná pro tyto problémy existují polynomické algoritmy, ale ty dosud nebyly objeveny. Je ale také možné, že takové algoritmy neexistují a problémy *nelze* řešit v polynomickém čase. Tyto problémy mohou být zkrátka vnitřně obtížné.

Jednou z důležitých skutečností je fakt, že časová složitost mnoha problémů je vzájemně svázána. Nalezení polynomického řešení pro jeden z těchto problémů dá polynomické řešení pro celou třídu takto svázaných problémů. Pro porozumnění tomuto fenoménu začněme následujícím příkladem.

Hamiltonovská cesta v orientovaném grafu G je cesta, která prochází každým vrcholem grafu přesně jednou. Uvažujme problém, který spočívá v testování toho, zda daný graf obsahuje Hamiltonovskou cestu spojující dva dané vrcholy.

$$HAMPATH = \{\langle G, s, t\rangle \mid \ G \text{ je orientovaný graf s Hamiltonovskou cestou z } s \text{ do } t\}.$$

Snadno lze navrhnout exponenciální algoritmus pro *HAMPATH* modifikací algoritmu hrubého násilí pro řešení problému *PATH*. Je jenom nutné přidat test, který ověří, zda potenciální cesta z s do t je Hamiltonovská. Nikdo však zatím neví, zda lze tento problém řešit polynomickým algoritmem.

Problém *HAMPATH* má vlastnost, kterou nazýváme *polynomická verifikovatelnost*. I když nevíme o rychlém (např. polynomickém) způsobu, jak zjistit, zda G obsahuje Hamiltonovskou cestu, pokud však takovou cestu objevíme (např. použitím exponencionálního algoritmu), pak umíme snadno někoho přesvědčit o tom, že cesta je Hamiltonovská. Řečeno jinak: verifikovat, že daná cesta je Hamiltonovská je mnohem snadnější než rozhodnout, zda Hamiltonovská cesta existuje.

Jiným polynomicky verifikovatelným problémem je problém, zda přirozené číslo je složené.

$$COMPOSITES = \{x \mid x = pq \text{ pro přirozená čísla } p, q > 1 \ \}.$$

I když opět není znám polynomický algoritmus pro rozhodování tohoto problému, je snadné verifikovat, že číslo je složené – jediné, co je potřeba je dělitel tohoto čísla.

Některé problémy nejsou polynomicky verifikovatelné. Například problém $\overline{HAMPATH}$, který je doplňkem problému *HAMPATH*, není polynomicky verifikovatelný. I když by se nám nějak podařilo zjistit, že graf neobsahuje Hamiltonovskou cestu, neznáme způsob, jak tento fakt verifikovat, aniž bychom byli nuceni použít stejný exponenciální algoritmus, kterým bylo provedeno původní zjištění.

Definice 1.13 Verifikátorem *pro jazyk A je algoritmus V takový, že*

$$A = \{w \mid V \text{ akceptuje } \langle w, c\rangle \text{ pro nějaký řetězec } c\}$$

Časovou složitost verifikátoru měříme pouze v závislosti na w a tedy polynomický verifikátor *má polynomickou časovou složitost vzhledem k velikosti vstupu w. Jazyk A je* polynimicky verifikovatelný*, jestliže má polynomický verifikátor.*

Verifikátor používá doplňkovou informaci, reprezentovanou symbolem c, k verifikaci toho, že $w \in A$. Tato informace se nazývá *svědek* nebo také *důkaz* příslušnosti do A.

Všimněme si, že pro polynomický verifikátor má svědek polynomickou velikost (vzhledem k velikosti w) protože to je maximální velikost, kterou může verifikátor přečíst v polynomickém čase. Ukažme si uvedenou definici na příkladech problémů *HAMPATH* a *COMPOSITES*.

Pro problém *HAMPATH* je svědkem toho, že $\langle G, s, t\rangle \in G$, jednoduše Hamiltonovská cesta z s do t. Pro problém *COMPOSITES* je svědkem složenosti čísla x jeden z jeho dělitelů. V obou případech může verifikátor pomocí svědka v polynomickém čase ověřit, že vstup patří do jazyka.

Definice 1.14 NP *je třída jazyků, které mají polynomický verifikátor.*

Třída NP je důležitou složitostní třídou, neboť obsahuje mnoho prakticky zajímavých problémů. Z předchozího víme, že *HAMPATH* a *COMPOSITES* patří do NP . Označení NP je odvozeno od *nedeterministický polynomický čas* a ten je odvozen od alternativní charakterizace třídy NP pomocí nedeterministických polynomických TS.

Následující nedeterministický TS rozhoduje problém *HAMPATH* v (nedeterministickém) polynomickém čase.

N_1 = Pro vstup $\langle G, s, t\rangle$:
1. Zapiš seznam m čísel, $p_1, \ldots, p_m$, kde m je počet vrcholů grafu G. Každé číslo v seznamu je nedeterministicky zvoleno tak, aby leželo v rozsahu 1 až m.
2. Ověř opakování v seznamu a pokud se čísla opakují, pak *zamítni*.
3. Ověř, zda $s = p_1$ a $t = p_m$. Jestliže něco neplatí, pak *zamítni*.
4. Pro každé i mezi 1 a $m - 1$ ověř, zda (p_i, p_{i+1}) je hrana v G. Jestliže něco neplatí, pak *zamítni*. Jinak *akceptuj*.

Je snadno vidět, že uvedený algoritmus je polynomický.

Věta 1.15 *Jazyk patří do* NP *právě když existuje nedeterministický TS polynomické časové složitosti, který tento jazyk rozhoduje.*

DŮKAZ: Ukážeme, jak převést polynomický verifikátor na ekvivalentní polynomický nedeterministický TS a obrácně. NTS simuluje verifikátor uhádnutím svědka, verifikátor simuluje NTS tak, že použije akceptující větev jako svědka.

Nechť nejprve $A \in$ NP . Nechť V je polynomický verifikátor pro A a předpokládejme, že V má časovou složitost n^k. Potřebný NTS je

N = Pro vstup w velikosti n:
1. Nedeterministicky vyber řetězec c délky n^k.
2. Proveď V nad vstupem $\langle w, c\rangle$.
3. Jestliže V akceptuje, pak *akceptuj*. Jinak *zamítni*.

Nyní dokážeme obrácené tvrzení. Předpokládejme, že A je rozhodnutelný pomocí polynomického NTS N. Polynomický verifikátor pro A je

N = Pro vstup $\langle w, c\rangle$, kde w, c jsou řetězce:
1. Simuluj N pro vstup w a při tom používej každý symbol v c jako popis nedeterministické volby provedené v každém kroku výpočtu stroje N. ∎
2. Jestliže je takto získaná větev výpočtu stroje N akceptující, pak *akceptuj*. Jinak *zamítni*.

Analogicky třídě TIME($t(n)$) definujeme nedeterministickou třídu NTIME($t(n)$)

Definice 1.16 *Nechť* $t : \mathbb{N} \to \mathbb{N}$. *Množina*

$$\mathsf{NTIME}(t(n)) = \{L \mid \textit{existuje NTS } N, \textit{jehož časová složitost je } \mathcal{O}(t(n)) \textit{ a } L = \mathcal{L}(N)\}$$

se nazývá nedeterministickou třídou časové složitosti.

Důsledek 1.17

$$\mathsf{NP} = \bigcup_k \mathsf{NTIME}(n^k)$$

Třída NP rovněž není citlivá k volbě rozumného nedeterministického výpočetního modelu, protože všechny takové modely jsou polynomicky ekvivalentní. Při analýze nedeterministického TS je nutné demonstrovat, ža každá větev výpočetního stromu má nejvýše polynomickou délku.

1.5.1 Příklady problémů z NP

Klika v neorientovaném grafu G je takový jeho podgraf, ve kterém jsou všechny vrcholy propojeny hranou. k-klika je klika s k vrcholy. Problém *CLIQUE* spočívá v rozhodnutí, zda graf obsahuje kliku určité velikosti.

$$\textit{CLIQUE} = \{\langle G, k\rangle \mid \ G \text{ je neorientovaný graf, který má } k\text{-kliku}\}.$$

Věta 1.18 $\textit{CLIQUE} \in \mathsf{NP}$

Důkaz: Svědkem je klika. Následující algoritmus je verifikátor.

N = Pro vstup $\langle\langle G, k\rangle, c\rangle$:
1. Testuj, zda c je množinou k vrcholů z G
2. Testuj, zda G obsahuje všechny hrany spojující vrcholy z c
3. Jsou-li oba testy úspěšné, pak *akceptuj*, jinak *zamítni*

Větu lze rovněž dokázat předložením polynomického NTS pro řešení problému *CLIQUE*.

N = Pro vstup $\langle G, k\rangle$, kde G je graf:
1. Nedeterministicky vyber podmnožinu c tvořenou k vrcholy grafu G.
2. Testuj, zda G obsahuje všechny hrany spojující vrcholy z c.
3. Jestliže ano, pak *akceptuj*, jinak *zamítni*.

∎

Jako další uvažme problém *SUBSET-SUM*. Je dána multimnožina celých čísel $x_1, \ldots, x_k$ a celé číslo t. Problém spočívá rozhodnutí, zda tato množina obsahuje podmnožinu (opět se v ní mohou prvky opakovat), jejíž součet je t.

$$\textit{SUBSET-SUM} = \{\langle S, t\rangle \mid \ S = \{x_1, \ldots, x_k\} \text{ a pro nějakou množinu } \{y_1, \ldots, y_l\} \subseteq S \text{ platí } \textstyle\sum y_i = t.$$

Například $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in \textit{SUBSET-SUM}$, protože 4+21=25.

Věta 1.19 $\textit{SUBSET-SUM} \in \mathsf{NP}$

DŮKAZ: Podmnožina je svědek. Následující algoritmus je verifikátor.

V = Pro vstup $\langle\langle S, t\rangle, c\rangle$:
1. Testuj, zda c je multimnožina čísel, jejíchž součet je t.
2. Testuj, zda S obsahuje všechna čísla z c.
3. Jsou-li oba testy úspěšné, pak *akceptuj*, jinak *zamítni*

Pro úplnost uveďme opět alternativní důkaz přes NTS. Následující algoritmus je polynomický NTS, který rozhoduje problém.

N = Pro vstup $\langle S, t\rangle$:
1. Nedeterministicky vyber podmnožinu c množiny S.
2. Testuj, zda c je multimnožinou čísel, jejichž součet je t.
3. Jestliže ano, pak *akceptuj*, jinak *zamítni*.

∎

Poznamenejme, že problémy $\overline{SUBSET\text{-}SUM}$ a $\overline{CLIQUE}$ nepatří do NP . Verifikace toho, že něco "není" přítomno se zdá být obtížnější než verifikace toho, že "je" to přítomno. Budeme uvažovat speciální složitostní třídu co-NP, která bude tvořena doplňky jazyků z NP . Nevíme, zda co-NP a NP se liší.

1.5.2 P versus NP

Vztah mezi třídami P a NP je velice zajímavý. Řekli jsme, že třída NP obsahuje problémy, které jsou rozhodnutelné v polynomickém čase na nedeterministických výpočetních modelech a nebo ekvivalentně jsou to problémy, pro něž lze pravdivou odpověď verifikovat v polynomickém čase na deterministických výpočetních modelech. Třída P obsahuje problémy, které je možné rozhodnout v polynomickém čase na deterministických výpočetních modelech.

P = třída jazyků, pro které lze příslušnost *rozhodovat* rychle.
NP = třída jazyků, pro které lze příslušnost *verifikovat* rychle.

Uvedli jsme příklady problémů (*HAMPATH*, *CLIQUE*), které patří do NP , ale neví se, zda patří do P. Síla polynomické verifikovatelnosti se zdá být mnohem větší než síla polynomické rozhodnutelnosti. Nicméně, i když si to lze těžko představit, je možné, že obě třídy jsou shodné. Nejsme schopni *dokázat* existenci problému, kter patří do NP, ale nepatří do P.

Otázka, zda P=NP je jednou z velkých otevřených otázek současné informatiky. Jestliže by platilo, že P=NP, pak by všechny polynomicky verifikovatelné problémy byly i polynomicky rozhodnutelné.

Nejlepší dosud možný způsob, jak řešit problémy z NP na deterministických strojích, je použít exponenciální algoritmus. Umíme dokázat, že

$$\mathsf{NP} \subseteq \mathsf{EXPTIME} = \bigcup_k \mathsf{TIME}(2^{n^k}),$$

ale nevíme, zda NP je obsaženo v striktně menší deterministické složitostní třídě.

1.5.3 NP-úplnost

V třídě NP existují problémy, jejichž výpočetní složitost je v jistém smyslu svázána se složitostí celé třídy. Jestliže by byl objeven polynomický deterministický algoritmus pro

takový problém, pak všechny problémy z NP by byly polynomicky řešitelné. Takové problémy nazýváme NP-úplné. Jsou důležité nejen z teoretického hlediska, ale mají i praktické dopady. Je zřejmé, že při důkazu otázky P=NP se stačí soustředit na NP-úplné problémy. Z praktického hlediska je naopak dobré si uvědomit, že je (zatím) marné se pokoušet hledat polynomický algoritmus pro NP-úplný problém. V současné době se obecně věří, že P≠NP. Pokud tedy o nějakém problému dokážeme, že je NP-úplný, pak tím současně podáme i silnou evidenci pro jeho nepolynomičnost.

První NP-úplný problém, se kterým se seznámíme, je problém splnitelnosti výrokových formulí.

$$SAT = \{\langle\Phi\rangle \mid \Phi \text{ je splnitelná výroková formule}\}.$$

Věta 1.20 (Cook-Levin) *$SAT \in \mathsf{P}$ právě když $\mathsf{P} = \mathsf{NP}$*

Základem pro důkaz Cook-Levinovy věty je metoda redukce. Vlastní důkaz uvedeme později.

Definice 1.21 *Funkce $f : \Sigma^* \to \Sigma^*$ je* vyčíslitelná v polynomickém čase, *jestliže existuje deterministický TS polynomické časové složitosti, který počítá f.*

Definice 1.22 *Jazyk A se* polynomicky redukuje *na jazyk B, píšeme $A \leq_P B$, jestliže existuje funkce $f : \Sigma^* \to \Sigma^*$, která je vyčíslitelná v polynomickém čase a taková, že pro každé $w \in \Sigma^*$ platí*

$$w \in A \Leftrightarrow f(w) \in B$$

Funkce f se nazývá polynomická redukce z A do B.

Polynomická redukce je "praktickou" analogií redukce $\leq_m$ z Definice **??**. Existují i další typy "praktické" redukce, ale polynomická redukce je jednoduchá a plně postačuje pro řešení otázek, které nás v této přednášce zajímají.

Lema 1.23 *Relace $\leq_P$ je reflexivní a tranzitivní, t.j. je* kvaziuspořádání.

Redukce dává způsob, jak převést rozhodování jednoho problému na rozhodování jiného problému. Je-li pro druhý problém znám polynomický algoritmus, pak i původní problém má polynomické řešení.

Věta 1.24 *Nechť $A \leq_P B$ a $B \in \mathsf{P}$. Pak $A \in \mathsf{P}$.*

Důkaz: Nechť M je polynomický algoritmus pro rozhodování B a nechť f je polynomická redukce z A do B. Jazyk A je rozhodován tímto polynomickým algoritmem.

N = Pro vstup w:
1. Vypočítej $f(w)$.
2. Spusť M pro vstup $f(w)$ a předej na výstup výsledek výpočtu algortimu M.

Je-li $w \in A$, pak $f(w) \in B$ neboť f je redukce z A do B. Proto M akceptuje $f(w)$ kdykoliv $w \in A$. Algoritmus N je polynomický, protože oba jeho příkazy vyžadují polynomický čas. ∎

Použití polynomické redukce demonstrujeme na problému *3SAT*. Jedná se o speciální případ problému *SAT*. *Literál* je booleovská proměnná nebo negovaná booleovska proměnná. *Klauzule* je několik literálů spojených disjunkcí $\vee$, např. $x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4$. Formule

je v *normální konjuktivní formě*, nayváme ji *cnf-formule*, jestliže je konjunkcí konečného počtu klauzulí. Např.

$$(x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6)$$

je cnf-formule. Formule je *3cnf-formule*, jestliže všechny její klauzule mají tři literály, např.

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6 \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$

Nechť $3SAT = \{\langle\Phi\rangle \mid \Phi$ je splnitelná 3cnf-formule$\}$. Ve splnitelné 3cnf-formuli musí každá klauzule obsahoval alespoň jeden literál, který bude mít přiřazenu hodnotu 1.

Věta 1.25 3SAT *se polynomicky redukuje na* CLIQUE.

Důkaz: Redukce z *3SAT* na *CLIQUE* je funkcí z formulí do grafů. K formuli zkonstruujeme kliku specifické velikosti, která odpovídá splňujícímu přiřazení proměnným ve formuli. Struktura grafu bude navržena tak, aby napodobovala chování proměnných a klauzulí.

Nechť Φ je formule s k klauzulemi

$$\Phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots (a_k \vee b_k \vee c_k)$$

Redukční funkce generuje řetězec $\langle G, k\rangle$, kde G je neorientovaný graf definovaný následovně.

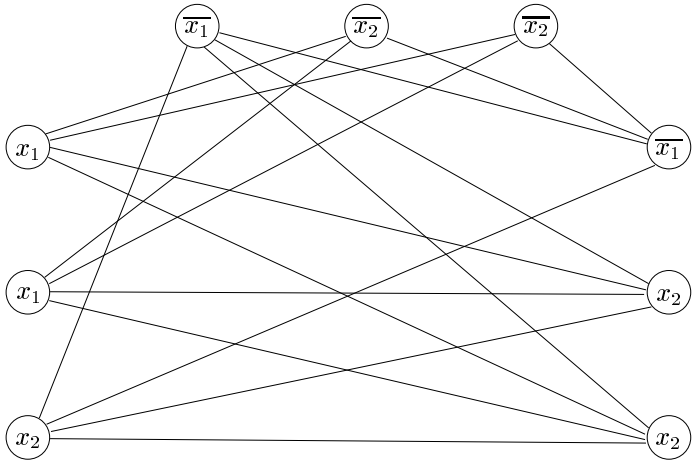
Vrcholy jsou organizovány to k skupin $t_1, \ldots, t_k$ po třech vrcholech - *trojicích*. Každá trojice odpovídá jedné klauzuli v dané formuli, každý vrchol ve trojici odpovídá literálu v příslušné klauzuli. Označme každý vrchol grafu G tímto literálem.

Hrany grafu G spojují, až na dvě výjimky, všechny dvojice vrcholů z G. Hrana není mezi vrcholy, které patří ke stejné trojici a hrana není rovněž mezi dvěma vrcholy s negovanými návěštími, např. není hrana mezi x_2 a $\neg x_2$

Konstrukce grafu pro formuli

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

je ilustrována na obrázku Obrázku 1.1. Nyní ukážeme, že Φ je splnitelná právě když G



Obrázek 1.1: Konstrukce grafu pro $(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

má k-kliku. Předpokládejme nejprve, že Φ je splnitelná. Splňující přiřazení musí přiřadit

1 alespoň jednomu literálu v každé klauzuli. V každé trojici grafu G zvolíme literál, kterému je přiřazena 1 (je-li jich více, pak libovolný z nich). Takto vybrané vrcholy tvoří k-kliku. Počet vrcholů v klice je k, protože formule má k klauzulí a tedy je k trojic v G. Z každé trojice jsme vybrali jeden vrchol. Každá dvojice vybraných vrcholů musí být spojena hranou. Vrchnoly nemohou být ze stejné trojice a nemohou být negovány, neboť proměnná nemůže mít současně přiřazenu 1 a 0. G tedy pbsahuje k-kliku.

Předpokládejme nyní, že G má k-kliku. Žádné dva vrcholy kliky se nemohou vyskytovat ve stejné trojici, protože žádné dva vrcholy ze stejné trojice nejsou spojeny hranou. Tedy každá trojice obsahuje přesně jeden vrchol k-kliky. Proměnným ve Φ přiřadíme hodnotu 1 tak, aby každý literál spojený s vrcholem kliky nabyl hodnotu 1. To je vždy možné, protože vrcholy označené negovanými proměnnými nejsou nikdy spojeny hranou a tedy nemohou být v klice. Takto vytvořené přiřazení je splňujícím pro Φ, protože každá trojice obsahuje vrchol z kliky a tedy každá klauzule obsahuje literál, kterému je přiřazena hodnota 1. Φ je splnitelná. ∎

Důledkem Věty 1.25 je, že pokud by problém *CLICQUE* byl řešitelný v deterministickém polynomickém čase, pak by takový byl i problém *3SAT*. I když oba problémy jsou zcela rozdílné, polynomická redukce umožnila formulovat těsnou souvislost mezi jejich složitostí. Nyní definujeme pojem, který dá do podobného vztahu ihned celou třídu problémů.

Definice 1.26 *Jazyk B je* NP-úplný*, jestliže:*

1. $B \in \mathsf{NP}$ *a*
2. *pro každý jazyk $A \in \mathsf{NP}$ platí $A \leq_P B$.*

Pokud B splňuje pouze druhou podmínku, pak se nazývá NP-těžký*.*

Věta 1.27 *Jestliže B je NP-úplný a $B \in \mathsf{P}$, pak P=NP.*

Důkaz: Přímo z definice polynomické redukce. ∎

Věta 1.28 *Jesltiže B je NP-úplný a $B \leq_P C$ pro každý jazyk $C \in \mathsf{NP}$, pak C je NP-úplný.*

Důkaz: Stačí ukázat, že pro každé $A \in \mathsf{NP}$ je $A \leq_P C$. Protože B je NP-úplný, každé $A \in \mathsf{NP}$ je $A \leq_P B$. Z předpokladu víme, že $B \leq_P C$ a tedy pro každé $A \in \mathsf{NP}$ je $A \leq_P C$. ∎

Polynomická redukce dává obecnou metodu pro získávání NP-úplný problémů. Její využití však předpokládá existenci alespoň jednoho NP-úplného problému. Nyní ukážeme, že problém splnitelnosti výrokových formulí je NP-úplný. Tvrzení, které uvedeme je jinou formulací Věty 1.20.

Věta 1.29 (Cook-Levin) *SAT je NP-úplný.*

Důkaz: Nechť M je *nedeterministický* TS takový, že $L(M) = A$. Nechť w je vstupní slovo pro M. Sestrojíme výrokovou formuli Φ takovou, že M akceptuje w právě když Φ je splnitelná. Konstrukce formule Φ bude polynomická.

Předpokládejme, že M má s stavů označených $q_1, \ldots, q_s$ a používá n páskových symbolů $X_1, \ldots, X_n$. Nechť $p(n)$ je časová složitost stroje M. Položme $w = \|n\|$. Jestliže stroj M akceptuje w, pak to neudělá za více než $p(n)$ kroků.

Jestliže M akceptuje w, pak existuje posloupnost konfigurací (slov tvaru xqy) $Q_0 \ldots, Q_q$ taková, že Q_0 je počáteční konfigurace, $Q_{i-1} \vdash Q_i, 1 \leq i \leq q$, Q_q je koncová konfigurace a navíc $q \leq p(n)$ a žádná konfigurace nezabírá více než $p(n)$ políček na pásce.

Sestrojíme formuli Φ, která *modeluje* posloupnosti $Q_0, \ldots, Q_q$ odpovídající výpočtům stroje M. Každé přiřazení hodnot *pravda* (1) a *lež* (0) proměnným ve formuli Φ odpovídá právě jedné posloupnosti konfigurací (třeba i nemožné).

Formule Φ nabývá hodnotu 1 právě když odpovídající přiřazení hodnot proměnným odpovídá posloupnosti konfigurací, vedoucí k akceptování vstupu a tedy Φ je splnitelná právě když M akceptuje w.

Nyní popišme konstrukci formule Φ. Za výrokové proměnné ve Φ vezmeme:

1. $C\langle i, j, t\rangle$ $(= 1 \Leftrightarrow i$-té políčko pásky obsahuje po provedení t kroků symbol $X_j)$
   $1 \leq i \leq p(n), 1 \leq j \leq m, 0 \leq t \leq p(n)$
2. $S\langle k, t\rangle$ $(= 1 \Leftrightarrow M$ je po provedení t kroků ve stavu $q_k)$
   $1 \leq k \leq s, 0 \leq t \leq p(n)$
3. $H\langle i, t\rangle$ $(= 1 \Leftrightarrow$ hlava je po provedení t kroků nad i-tým políčkem)
   $1 \leq i \leq p(n), 0 \leq t \leq p(n)$

Proměnných je $\mathcal{O}(p^2(n))$. Proměnné je možné zakódovat do binární abecedy řetězci délky $c \log n$, kde c závisí na p. V dalším si budeme výrokovou proměnnou představovat jako jeden symbol a ne jako $c \log n$ symbolů. Ztráta faktoru $c \log n$ se nemůže promítnout do polynomické časové složitosti (kódování a dekódování).

Při konstrukci formule Φ použijeme predikát $U(X_1, \ldots, X_r)$, který nabývá hodnotu 1 (true) právě když *přesně* jeden z argumentů má hodnotu 1, t.j.

$$U(X_1, \ldots, X_r) = (X_1 \vee \ldots \vee X_r) \wedge \bigwedge_{\substack{i,j \\ i \neq j}} (\neg X_i \vee \neg X_j)$$

Velikost U je $\mathcal{O}(r^2)$.

Jestliže M akceptuje w, pak existuje posloupnost konfigurací $Q_0, Q_1, \ldots, Q_q$, která končí akceptující konfigurací. Předpokládejme, že M dělá *přesně* $p(n)$ kroků - jinak přidáme po akceptování "prázdné" kroky a že všechny konfigurace zabírají $p(n)$ políček.

Formuli Φ budeme konstruovat jako konjunkci formulí $A, B, \ldots, G$, která tvrdí "přípustnost" posloupnosti $Q_0, Q_1, \ldots, Q_q$, přičemž každé Q_i má délku $p(n)$ a $q = p(n)$. Tvrzení, že $Q_0, Q_1, \ldots, Q_q$ je přípustná, znamená:

1. v každém Q_i je hlava přesně nad jedním symbolem,
2. v každém Q_i je v každém poli přesně jeden sybmol,
3. každé Q_i obsahuje přesně jeden stav,
4. při přechodu od Q_i k Q_{i+1} se změní nejvýše pole pod hlavou,
5. změna stavu odpovídá přechodové funkci stroje M,
6. Q_0 je počáteční konfigurace a
7. stav v Q_q je koncový (Q_q je koncová konfigurace).

Sestrojíme formule $A, B, \ldots, G$, které odpovídají postupně výrokům $1, \ldots, 7$.

1. **Formule A** (v každém Q_i je hlava přesně nad jedním symbolem)

   Nechť A_t říká, že po provedení t kroků je hlava právě nad jedním políčkem pásky. Pak

   $$A = A_0 \wedge A_1 \wedge \ldots \wedge A_{p(n)}$$

   kde

   $$A_t = U(H\langle 1,t\rangle, H\langle 2,t\rangle, \ldots, H\langle p(n),t\rangle)$$

   A má délku $\mathcal{O}(p^3(n))$.

2. **Formule B** (v každém Q_i je v každém poli přesně jeden sybmol)

   Nechť B_{it} říká, že i-té políčko obsahuje po provedení t kroků přesně jeden symbol. Pak

   $$B = \bigwedge_{i,t} B_{it}$$

   $$B_{it} = U(C\langle i,1,t\rangle, \ldots C\langle i,m,t\rangle)$$

   Délka B_{it} nezávisí na n neboť velikost páskové abecedy závisí jen na M.

   B má délku $\mathcal{O}(p^2(n))$.

3. **Formule C** (každé Q_i obsahuje přesně jeden stav)

   $$C = \bigwedge_{0 \le t \le p(n)} U(S\langle 1,t\rangle, \ldots, S\langle s,t\rangle)$$

   Počet stavů je konstatní (nezávisí na n).

   C má délku $\mathcal{O}(p(n))$.

4. **Formule D** (při přechodu od Q_i k Q_{i+1} se změní nejvýše pole pod hlavou)

   $$D \quad = \quad \bigwedge_{i,j,t} \quad [(C\langle i,j,t\rangle \equiv C\langle i,j,t+1\rangle) \vee H\langle i,t\rangle]$$

   $$\Uparrow$$

   (a) po provedení t kroků je hlava nad i-tým políčkem *nebo*

   (b) j-tý symbol je zapsán v i-tém políčku po provedení $t+1$ kroků $\Leftrightarrow$ tam byl po provedení t kroků

   D má délku $\mathcal{O}(p^2(n))$.

5. **Formule E** (změna stavu odpovídá přechodové funkci stroje M)

   Nechť E_{ijkt} platí, jestliže je pravdivé jedno z tvrzení:

   (a) v čase t neobsahuje pole i symbol j

   (b) v čase t není hlava nad i-tým polem

   (c) v čase t není stroj M ve stavu k

   (d) následující konfigurace vzniká z předchozí podle přechodové funkce

Pak

$$E = \bigwedge_{i,j,k,t} E_{ijkt}$$

kde

$$\begin{aligned} E_{ijkt} \quad = \quad & \neg C\langle i,j,t\rangle \vee \neg H\langle i,t\rangle \vee \neg S\langle k,t\rangle \\ & \bigvee_l [C\langle i,j_l,t+1\rangle \wedge S\langle k_l,t+1\rangle \wedge H\langle i_l,t+1\rangle] \end{aligned}$$

l jde přes všechny kroky stroje M možné v situaci, kdy M čte symbol x_j a je ve stavu q_k.

Pro každou trojici $\langle q,x,d\rangle \in \delta(q_k,x_j)$ existuje jedna hodnota l, pro kterou $x_{j_l} = x, q_{k_l} = q$ a $i_l \in \{i-1,i,i+1\}$.

M je nedeterministický, tedy trojic $\langle q,x,d\rangle$ může být více, ale vždy *konečně mnoho.*

E_{ijkt} má ohraničenou velikost nezávislou na n.

Velikost E je $\mathcal{O}(p^2(n))$.

6. **Formule F** (Q_0 je počáteční konfigurace)

$$F = S\langle 1,0\rangle \wedge H\langle 1,0\rangle \wedge \bigwedge_{1\le i\le n} C\langle i,j_i,0\rangle \wedge \bigwedge_{n<i\le p(n)} C\langle i,1,0\rangle$$

Velikost F je $\mathcal{O}(p(n))$.

7. **Formule G** (Q_q je koncová konfigurace)

$$G = S\langle s,p(n)\rangle$$

$$\Phi = A \wedge \ldots \wedge G$$

Φ neobsahuje více než $\mathcal{O}(p^3(n))$ symbolů. Délka Φ závisí na délce slova w polynomicky a lze ji sestrojit v polynomickém čase v závislosti na n (délce slova w).

∎

Prostorová složitost

Přednáška č.11

2.1 Měření prostorové složitosti

V této části se soustředíme na složitost výpočetních problémů z hlediska jejich požadavku na prostor (paměť). Prostorová složitost má mnoho společných rysů s časovou složitostí, zejména pak slouží ke další klasifikaci problémů z hlediska jejich výpočetní obtížnosti. Pro měření prostorové složitosti budeme používat opět Turingovy stroje.

Definice 2.1 (deterministický TS) *Nechť M je daný deterministický TS, který zastaví pro každý vstup. Řekneme, že M má* prostorovou složitost $Space_M : \mathbb{N} \to \mathbb{N}$, *jestliže $Space_M(n)$ je maximální počet políček pásky čtených strojem M, kde maximum se bere přes všechny vstupy délky n.*

Definice 2.2 (nedeterministický TS) *Nechť M je daný nedeterministický TS, jehož výpočetní strom má pro každý vstup všechny výpočetní cesty konečné.* Prostorová složitost *stroje M je funkce $Space_M : \mathbb{N} \to \mathbb{N}$ taková, že $Space_M(n)$ je maximální počet políček pásky čtených strojem M, kde maximum se bere přes všechny vstupy délky n a pro každý vstup délky n přes všechny výpočetní cesty.*

Podobně jako u časové složitosti, budeme i pro prostorovou složitost TS používat asymptotické odhady. Nyní definujme základní prostorové složitostní třídy.

Definice 2.3 *Nechť $s : \mathbb{N} \to \mathbb{N}$. Množina*

$$\mathsf{SPACE}(s(n)) = \{L \mid \textit{existuje TS } M, \textit{jehož prostorová složitost je } \mathcal{O}(s(n)) \textit{ a } L = \mathcal{L}(M)\}$$

se nazývá (deterministickou) třídou prostorové složitosti. *Množina*

$$\mathsf{NSPACE}(s(n)) = \{L \mid \textit{existuje NTS } N, \textit{jehož prostorová složitost je } \mathcal{O}(s(n)) \textit{ a } L = \mathcal{L}(N)\}$$

se nazývá nedeterministickou třídou prostorové složitosti.

Příklad 2.4 *Ukážeme, že problém SAT lze řešit v deterministickém lineárním prostoru. Věříme, že SAT nelze vyřešit v polynomickém čase, tím méně v lineárním. Zdá se, že prostor je silnější výpočetní zdroj než čas. Lze ho totiž znovu použít, zatímco čas nikoliv. Uvažujme algoritmus*

M_1 = *Pro vstup $\langle \Phi \rangle$, kde Φ je výroková formule:*
1. *Pro každé přiřazení pravdivostních hodnot proměnným $x_1, \ldots, x_m$ ve formuli Φ:*
2. *Vyhodnoť Φ pro toto splňující přiřazení.*
3. *Jestliže Φ je vyhodnoceno na* pravdu, *pak* akceptuj, *jinak* zamítni.

Stroj M_1 potřebuje pouze lineární prostor, neboť každá iterace cyklu může znovu použít stejnou část pásky. Stroj potřebuje pouze uložit aktuální pravdivostní přiřazení a k tomu je potřeba prostor $\mathcal{O}(m)$, kde m je počet proměnných ve formuli. Počet proměnných není větší než délka formule a tedy M_1 pracuje v prostoru $\mathcal{O}(n)$.

Příklad 2.5 *Nyní demonstrujeme nedeterministickou prostorovou složitost problému. Uvažujme problém*

$$ALL_{NKA} = \{\langle A\rangle \mid A \text{ je NKA (nedeterministický konečný automat) a } \mathcal{L}(A) = \Sigma^*\}.$$

Uvedeme nedeterministický TS, který pracuje v lineárním prostoru a rozhoduje doplněk $\overline{ALL_{NKA}}$. Myšlenka algoritmu je použít nedeterminismus k uhádnutí slova, které je automatem zamítnuto a použít lineární prostor k zapamatování si těch stavů, do kterých se NKA může dostat v určitém čase. Poznamenejme, že o tomto jazyku není známo, zda patří do NP *ani zda patří do* co-NP.

N = *Pro vstup $\langle M\rangle$, kde M je NKA:*
1. *Umísti značku na počáteční stav automatu M.*
2. *Opakuj 2^q-krát, kde q je počet stavů automatu M:*
3. *Nedeterministicky vyber vstupní symbol a změn pozice značek na stavech automatu M tak, aby simuloval čtení tohoto symbolu.*
4. *Jestliže byla někdy značka umístěna na koncový stav, pak* zamítni, *jinak* akceptuj.

Jestliže M akceptuje nějaké slovo, pak musí akceptovat i slovo délky nejvýše 2^q, neboť pro každé delší slovo, které je akceptováno, by se umístění značek muselo opakovat. Úsek slova mezi těmito opakováními lze vyjmout a získat tak kratší akceptované slovo. Tedy N rozhoduje ALL_{NKA}.

Jediný prostor potřebný k výpočtu stroje N je nutný k uchování umístění značek a to vyžaduje pouze nedeterministický lineární prostor.

2.2 Savitchova věta

Savitchova věta říká, že deterministické stroje mohou simulovat nedeterministické s použitím překvapivě malého prostoru. Pro časovou složitost se zdá, že takováto simulace vyžaduje exponnciální nárůst času.

Věta 2.6 (Savitchova věta) *Pro každou funkci $f : \mathbb{N} \to \mathbb{N}$, kde $f(n) \geq n$, platí:*

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f^2(n)).$$

Důkaz: Nechť N je NTS, který rozhoduje jazyk A v prostoru $f(n)$. Sestrojíme DTS M pro rozhodování jazyka A. Strij M bude používat proceduru *REACH*, která testuje, zda z jedné konfigurace stroje N lze dosáhnout jinou v daném počtu kroků. Nechť w je vstup pro N. Pro konfigurace c_1 a c_2 a celé číslo t vrátí procedura *REACH* hodnotu *akceptuj*, jestliže existuje posloupnost nedeterministických voleb stroje N taková, že z konfigurace c_1 přejde do konfigurace c_2 v t krocích. V opačném případě vrátí hodnotu *zamítni*.

REACH = Pro vstup c_1, c_2, t:
1. Je-li $t = 1$, pak přímo ověř, zda $c_1 = c_2$ nebo zda c_1 přejde do c_2 v jednom kroku. Pokud alespoň jedna možnost platí, pak *akceptuj*, jinak *zamítni*.
2. Je-li $t > 1$, pak pro každou konfiguraci c_m stroje N pro vstup w, která používá prostor $f(n)$:
3. Proveď $REACH(c_1, c_m, \lceil \frac{t}{2} \rceil)$
4. Proveď $REACH(c_m, c_2, \lceil \frac{t}{2} \rceil)$
5. Jestliže v obou krocích 3 a 4 je výsledkem *akceptuj*, pak *akcceptuj*.
6. Jestliže nebylo akcpetováno, pak *zamítni*.

Nyní definujeme stroj M, který simuluje N. Nejprve modifikujeme N tak, aby po akceptování vymazal pásku, přesunul hlavu na začátek pásky a přešel do akceptujícího stavu c_{accept}. Nechť c_{start} je počáteční konfigurace stroje N pro vstup w. Nechť d je takové, že $2^{df(n)}$ je maximální délka větve ve výpočetním stromu stroje N, kde n je délka slova w.

M = Pro vstup w:
1. Předej na výstup hodnotu $REACH(c_{start}, c_{accept}, 2^{df(n)})$.

Je zřejmé, že M simuluje N. Analyzujme jeho prostorovou složitost. Kdykoli procedura *REACH* volá rekurzívně sama sebe, uloží hodnoty c_1, c_2 a t na zásobník. Každá úroveň rekurze tak používá dodatečný prostor $f(n)$. Na každé úrovni je hodnota t zmenšena na polovinu. Na začátku je $t = 2^{df(n)}$ a tedy hloubka rekurze je $\mathcal{O}(\log 2^{df(n)})$ neboli $\mathcal{O}(f(n))$. Celkový potřebný prostor je $\mathcal{O}(f^2(n))$

∎

2.3 Třída PSPACE

Třídu PSPACE definujem podobně jako třídu P .

Definice 2.7 PSPACE *je třída jazyků (problémů), které jsou rozhodnutelné v polynomickém prostoru na deterministických jednopáskových Turingových strojích. T.j.*

$$\mathsf{P} = \bigcup_k \mathsf{SPACE}(n^k).$$

Nedeterministickou třídu NPSPACE lze definovat analogicky. Nicméně, ze Savitchovy věta vyplývá, že PSPACE = NPSPACE .

Jaký je vzájemný vztah mezi třídami PSPACE a P či NP ? Každý stroj, který počítá rychle, nemůže spotřebovat příliš mnoho prostoru. Je-li, totiž $t(n) \geq n$ časové omezení na TS M, pak M nemůže použít více políček pásky než $t(n)$, neboť v každém kroku výpočtu může použít nejvýše jedno nové políčko. Je tedy P ⊆ PSPACE . Podobně NP ⊆ NPSPACE a tedy NP ⊆ PSPACE .

Spotřebovaný prostor omezuje i možný čas výpočtu. Je-li $s(n) \geq n$, pak TS M, který používá prostor $s(n)$, může mít nejvýše $s(n).2^{\mathcal{O}(s(n))}$ vzájemně různých konfigurací. TS, který zastaví, nemusí opakovat konfigurace. Proto TS, který používá prostor $s(n)$, musí pracovat v čase $s(n).2^{\mathcal{O}(s(n))}$ a tedy PSPACE ⊆ EXPTIME .

Vzájemné vztahy mezi doposud zkoumanými třídami jsou:

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} \subseteq \mathsf{EXPTIME}$$

Není známo, zda některé z výše uvedených inkluzí je rovnost. Nicméně je známo, že $\mathsf{P} \neq \mathsf{EXPTIME}$. Proto alespoň jedna z těchto inkluzí musí být vlastní.

Podobně jako v případě NP-úplných problémů, hrají významnou roli i PSPACE-úplné problémy.

Definice 2.8 *Jazyk B je* PSPACE-těžký, *jestliže pro každý jazyk $A \in \mathsf{PSPACE}$ platí $A \leq_P B$. Jestliže navíc platí i $B \in \mathsf{PSPACE}$, pak B se nazývá* PSPACE-úplný.

Všimněme si, že PSPACE-úplnost jsme definovali pomocí polynomické *časové* redukce. Proč jsme nepoužili polynomickou prostorovou redukci? Úplné problémy v nějaké třídě představují ty nejtěžší problémy neboť každý jiný problém v této třídě se na něj dá snadno redukovat. Jestliže tedy najdeme snadný způsob, jak vyřešit úplný problém, pak máme i snadný způsob řešení pro všechny problémy z této třídy. Redukce tedy musí být *snadná* relativně k uvažované třídě. Jesltiže by redukci bylo těžké spočítat, pak snadné řešení pro úplný problém by nemuselo dát snadné řešení pro ostatní problémy. Kdykoli tedy definujeme úplné problémy pro nějakou složitostní třídu, pak použitá redukce musí být výrazněji snadnější než model použitý pro definici této třídy.

Nyní uvedeme příklad PSPACE-úplného problému. Jedná se o problém rozhodnout, zda kvantifikovaná výroková formule je pravdivá. Výrokové formule, ve kterých je povoleno používat všobecný ($\forall$) a existenční ($\exists$) kvantifikátor pro kvantifikaci výrokových proměnných, nazýváme *kvantifikované výrokové formule*. Příkladem takové formule je

$$\Phi_1 = \forall x \exists y[(x \vee y) \wedge (\neg x \vee \neg y)]$$

Formule Φ je pravdivá. Formule

$$\Phi_2 = \exists x \forall y[(x \vee y) \wedge (\neg x \vee \neg y)]$$

je naproti tomu nepravdivá. Konečně, formule

$$\Phi_3 = \exists y[(x \vee y) \wedge (\neg x \vee \neg y)]$$

může být někdy pravdivá a někdy nepravdivá. To záleží na přiřazení pravdivostní hodnoty (0 nebo 1) volné proměnné x.

Uzavřená kvantifikovaná výroková formule je taková, že každá proměnná je v rozsahu nějakého kvantifikátoru (formule neobsahuje volné výrokové proměnné). Uzavřené formule jsou vždy pradvivé nebo vždy nepravdivé.

Položme

$$TQBF = \{\langle \Phi \rangle \mid \Phi \text{ je pravdivá uzavřná kvantifikovaná výroková formule}\}.$$

Věta 2.9 *TQBF je PSPACE-úplný.*

Důkaz:

1. Nejprve dokážeme, že $TQBF \in \mathsf{PSPACE}$. Uvažujme následující algoritmus pro rozhodování $TQBF$.

T = Pro vstup $\langle\Phi\rangle$, kde Φ je výrok:
1. Pokud Φ neobsahuje kvantifikátory, vyhodnoť Φ a *akceptuj* právě když hodnota je pravda.
2. Jestliže $\Phi = \exists\Psi$, pak rekurzívně zavolej T na ψ nejprve s hodnotou $x = 0$ a pak s $x = 1$. Je-li alespoň jeden výsledek akceptující, pak akceptuj, jinak zamítni.
3. Jestliže $\Phi = \forall\Psi$, pak rekurzívně zavolej T na ψ nejprve s hodnotou $x = 0$ a pak s $x = 1$. Jsou-li oba výsledky akceptující, pak akceptuj, jinak zamítni.

Hloubka rekurze je nejvýše rovna počtu proměnných m. Na každé úrovni je nutné uložit hodnotu jedné proměnné. Celkový prostor je tedy $\mathcal{O}(m)$ a algoritmus pracuje v lineárním prostoru.

2. Nyní dokážeme, že *TQBF* je PSPACE-těžký. Nechť A je jazyk rozhodovaný DTS M v prostoru n^k pro nějaké k. Popíšeme polynomickou redukci z A do *TQBF*.

Redukce přiřadí slovu w uzavřenou kvantifikovanou výrokovou formuli Φ tak, že Φ je pravdivá právě když M akceptuje w.

Konfigurace TS lze reprezentovat jako výrokové formule, ve kterých proměnné reprezentují stavy a obsahy políček na pásce a pozice hlavy je zakódována uspořádáním formulí (viz důkaz Věty 1.29). Každá konfigurace má n^k políček a tedy je zakódována pomocí $\mathcal{O}(n^k)$ proměnných.

Uvažme obecnější problém. Pro dvě konfigurace c_1 a c_2 a celé číslo $t > 0$ sestrojíme formuli $\Phi_{c_1,c_2,t}$ takovou, že $\Phi_{c_1,c_2,t}$ je pravdivá, právě když M může přejít z konfigurace c_1 do c_2 v nejvýše t krocích.

Požadované tvrzení pak dostaneme, jestliže položíme $c_1 = c_{start}$, $c_2 = c_{accept}$ a $t = 2^{df(n)}$, kde c_{start}je počáteční konfigurace, c_{accept} je akceptující konfigurace, d je maximální délka větve ve výpočetním stromu stroje M a n je délka slova w.

Je-li $t = 1$, pak formule vyjadřuje, že $c_1 = c_2$ nebo c_2 lze odvodit z c_1 v jednom kroku.

Pro $t > 1$ setrojíme $\Phi_{c_1,c_2,t}$ rekurzívně:

$$\Phi_{c_1,c_2,t} = \exists m_1[\Phi_{c_1,m_1,\lceil\frac{t}{2}\rceil} \wedge \Phi_{m_1,c_2,\lceil\frac{t}{2}\rceil}]$$

kde m_1 je konfigurace stroje M. Výraz $\exists m_1$ je zkratkou pro $\exists x_1, \ldots, x_l, l \in \mathcal{O}(n^k)$ a $x_1, \ldots, x_l$ jsou proměnné, které kódují m_1.

Při každém rekurzívním volání se velikost formule zdvojnásobí. Výpočet formule tedy vyžaduje exponenciální čas, což je přpíliš. Pro zmenšení velikosti formule použijeme všeoecný kvantifikátor. Nechť

$$\Phi_{c_1,c_2,t} = \exists m_1 \forall (c_3, c_4) \in \{(c_1, m_1), (m_1, c_2)\}[\Phi_{c_3,c_4,\lceil\frac{t}{2}\rceil}].$$

Zavedením nových proměnných, které reprezentují konfigurace c_3 a c_4, získáváme možnost "zabalit" dvě rekurzivní podformule do jedné při zachování původního významu. Zápis $\forall(c_3, c_4) \in \{(c_1, m_1), (m_1, c_4)\}$ znamená, že proměnné reprezentující konfigurace c_3 a c_4 mohou nabýt hodnoty proměnných c_1 a m_1 nebo m_1 a c_4 a že výsledná formule $\Phi_{c_3,c_4,\lceil\frac{t}{2}\rceil}]$ je v obou případech pravdivá. Výraz $\forall x \in \{y, z\}[...]$ lze zapsat jako $\forall x[(x = y \vee x = z) \rightarrow ...]$ a získat tak syntakticky korektní uzavřenou kvantifikovanou výrokovou formuli.

Každé rekurzívní volání přidá část formule, které je lineární ve velikosti konfigurace, tedy $\mathcal{O}(f(n))$. Počet úrovní rekurze je $\log(2^{df(n)})$ neboli $\mathcal{O}(f(n))$. Velikost výsledné formule $\Phi_{c_{start},c_{accept},2^{df(n)}}$ je $\mathcal{O}(f^2(n))$. ∎