Studijní opora k předmětům teoretické informatiky

Petr Jančar

22. února 2005

Obsah

Předkládaný materiál slouží jako studijní opora pro předměty teoretické informatiky, speciálně pro oblasti teorie jazyků a automatů a teorie algoritmů (či teorie vyčíslitelnosti a složitosti). Způsob jeho využití v konkrétním kursu doporučí přednášející.

Materiál vznikl (v lednu 2004) sloučením pracovních textů k dřívějším samostatným kursům ‘Teorie jazyků a automatů’ a ‘Vyčíslitelnost a složitost’ a skládá se tudíž ze dvou relativně samostatných částí. Každá část má svůj vlastní úvod, ovšem odkazuje se do společného seznamu literatury (uvedeného v závěru).

Omlouvám se za ponechané specifické poznámky u jednotlivých kursů, které v obecném kontextu možná nejsou relevantní.

Část I

Úvod do teorie jazyků a automatů

Kapitola 1

Úvod, cíle kursu, literatura

Teorie jazyků a automatů patří ke klasickým partiím teoretické informatiky a je ve větší či menší míře součástí studijních plánů informatiky na všech vysokých školách.

Zde předkládaný materiál je zamýšlen jako studijní opora pro úvodní kurs v dané oblasti. Materiál vznikl přepracováním a doplněním pracovního textu, jenž původně sloužil jen jako doprovodný materiál k přednáškám. Nynější text by měl umožňovat (resp. více podporovat) i samostatné či distanční studium. Speciálně byly do textu doplněny úkoly, o jejichž vyřešení je studující na příslušně označených místech v textu žádán.

Poznámka. Na jaře 2003 převedli diplomanti L. Jamrozová a L. Koblovský text do interaktivnější elektronické podoby; využitím TeXovského balíku maker “Elearn” J. Dvorského a dodáním ilustrativních animací k vybraným pojmům a postupům. Vedle tištěné verze tohoto materiálu lze tedy pracovat i s interaktivnějším pdf-souborem.

Text je členěn logicky do kapitol, z nichž každá obsahuje stručně specifikované studijní cíle. Jelikož materiál je samozřejmě i nadále možno použít jako podporu k řádnému kursu s přednáškami (tedy nejen k distančnímu studiu), je v textu vyznačeno i orientační rozdělení na jednotlivé přednášky (v délce 2 x 45 minut). Počítá se, že každá přednáška je doplněna (rovněž dvouhodinovým) cvičením, na které se ovšem studenti předem samostatně připravují. Z těchto orientačních údajů je možné odhadnout časovou náročnost studia jednotlivých kapitol či jejich částí – jde ovšem čistě o orientační odhad, jelikož u každého jednotlivce bude čas potřebný k důkladnému zvládnutí látky velmi individuální.

Velmi přínosná by samozřejmě byla snaha studujícího prostudovat probírané partie také v některé z doporučených (či jiných) monografií. V češtině či slovenštině vyšly např. [Chy84], [HU78] (což je slovenský překlad angl. originálu z r. 1969), [MvM87]. Kromě uvedených knih existují jistě i další texty v češtině či slovenštině, které se zabývají podobnou problematikou. Nepoměrně bohatší je ovšem nabídka příslušné literatury v angličtině, což lze snadno zjistit např. ‘surfováním’ na Webu. Uveďme např. alespoň [Sip97] či knihu českého autora [Med00].

Další informace ke kursu najdete přes odkaz na webovské stránce

http://www.cs.vsb.cz/jancar
(či přímo http://www.cs.vsb.cz/jancar/TJAA/tjaa.htm)

Autor bude vděčný za jakékoli připomínky k předkládanému materiálu (stejně jako obecně ke kursu), sdělené např. e-mailem na adresu Petr.Jancar@vsb.cz

Teorie jazyků a automatů patří k základním (a dnes již klasickým) partiím teoretické informatiky, partiím, jejichž vznik a vývoj byl a je úzce svázán s potřebami praxe při vývoji software, hardware a obecně při modelování systémů.

Náš kurs tvoří určitý celek s (následným) kursem *Vyčíslitelnost a složitost*, dohromady by se mohly nazývat *Základy teorie výpočtů* (Theory of Computation), což je součást *teoretických základů informatiky*.

Motivovat teorii výpočtů lze přirozenými otázkami typu:
- jak srovnat kvalitu (rychlost) různých algoritmů řešících tentýž problém (úkol) ?
- jak lze porovnávat (klasifikovat) problémy podle jejich (vnitřní) složitosti ?
- jak charakterizovat problémy, které jsou a které nejsou algoritmicky řešitelné, tj. které lze a které nelze řešit algoritmy (speciálně “rychlými”, neboli prakticky použitelnými, algoritmy).

Při zpřesňování těchto a podobných otázek, a při hledání odpovědí, nutně potřebujeme (abstraktní) modely počítače (tj. toho, kdo provádí výpočty). Z více důvodů je vhodné při našem zkoumání začít velmi jednoduchým, ale fundamentálním modelem, a sice tzv. *konečnými (tj. konečně stavovými) automaty*. Konečné automaty (pojem byl formalizován ve 40. letech 20. století), lze chápat nejen jako nejzákladnější model v oblasti počítačů, ale ve všech oblastech, kde jde o modelování systémů, procesů, organismů apod., u nichž lze vyčlenit konečně mnoho stavů a popsat způsob, jak se aktuální stav mění prováděním určitých akcí (např. přijímáním vnějších impulsů). (Jako jednoduchý ilustrující příklad nám může posloužit model ovladače dveří v supermarketu znázorněný na obr. 2.1, o němž pojednáme níže.)

Velmi běžná “výpočetní” aplikace, u níž je v pozadí konečný automat, je *hledání vzorků v textu*. Takové hledání asi nejčastěji používáme v textových editorech a při vyhledávání na Internetu; speciální případ také představuje např. *lexikální analýza* v překladačích. Při vyhledávání informací v počítačových systémech jste již jistě narazili na nějakou variantu *regulárních výrazů*, umožňujících specifikovat celé třídy vzorků. Regulárními výrazy a jejich vztahem ke konečným automatům se rovněž budeme zabývat.

Po seznámení se s konečnými automaty a regulárními výrazy budeme pokračovat silnějším modelem – tzv. *zásobníkovými automaty*; o ty se opírají algoritmy *syntaktické analýzy* při překladu (programovacích) jazyků, tedy algoritmy, které např. určí, zda vámi napsaný program v Pascalu (C-čku) je správně pascalsky (c-čkovsky).

Zmínili jsme pojem *jazyk* – obecně budeme mít na mysli tzv. formální jazyk; jazyky přirozené (mluvené) či jazyky programovací jsou speciálními případy. Na naše modely se v prvé řadě budeme dívat jako na *rozpoznávače jazyků*, tj. zařízení, které zpracují vstupní posloupnost písmen (symbolů) a rozhodnou, zda tato posloupnost je (správně utvořenou) větou příslušného jazyka.

Poznámka. Ve skutečnosti je zřejmě důležitější (a přirozenější) pojem *funkce* realizované daným automatem – automat k zadanému vstupu (“spočítá” a) vydá jistý výstup, tedy realizuje určitou (vstupně-výstupní) funkci. Rozpoznávač jazyka je pouze speciálním případem, kdy výstupem je 1 či 0 (ANO či NE). (Později se ještě o tomto problému zmíníme.)

S pojmem *jazyk* se nám přirozeně pojí pojem *gramatika*. Speciálně se budeme věnovat tzv. *bezkontextovým gramatikám*, s nimiž jste se již přinejmenším implicitně setkali u definic syntaxe programovacích jazyků (tj. pravidel konstrukce programů); ukážeme mj., že bezkontextové gramatiky generují právě ty jazyky, jež jsou rozpoznávány zásobníkovými automaty.

Seznámíme se také s jedním z univerzálních (nejobecnějších) modelů počítačů (resp. rozpoznávačů jazyků) – s tzv. *Turingovými stroji*. K tomuto modelu se také odkazuje zmíněný kurs o vyčíslitelnosti a složitosti.

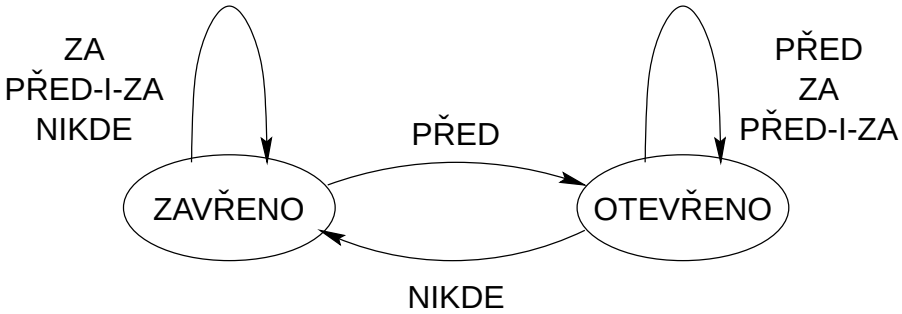
Ještě poznamenejme, že účelem kursu není popis konkrétních “reálných” (větších) aplikací studovaných (teoretických) pojmů; účelem je základní seznámení se s těmito pojmy a s příslušnými obecnými výsledky a metodami. Jejich zvládnutí je nezbytným základem pro porozumění i oněm reálným aplikacím a pro jejich návrh. Jako pěkný příklad relativně nedávné aplikace může sloužit použití konečných automatů s vahami pro reprezentaci složitých funkcí (na reálném oboru) a jejich využití při reprezentaci, transformaci a kompresi obrazové informace (viz např. kapitolu v [Gru97]).

Kapitola 2

Konečné automaty, regulární jazyky

Cíl kapitoly:
Seznámení se s pojmem konečného automatu, speciálně jako s prostředkem rozpoznávání posloupností symbolů splňujících určité (jednoduché) podmínky. Pochopení všech souvisejících pojmů a formálních (matematických) definic. Zvládnutí návrhu elementárních automatů a jednoduchých algoritmů zjišťujících vlastnosti automatů.

Studijní cíle: Seznámení se s pojmem konečného automatu, speciálně jako s prostředkem rozpoznávání posloupností symbolů splňujících určité (jednoduché) podmínky. Pochopení všech souvisejících pojmů a formálních (matematických) definic. Zvládnutí návrhu elementárních automatů a jednoduchých algoritmů zjišťujících vlastnosti automatů.



Obrázek 2.1:

	PŘED	ZA	PŘED-I-ZA	NIKDE
ZAVŘENO	OTEVŘENO	ZAVŘENO	ZAVŘENO	ZAVŘENO
OTEVŘENO	OTEVŘENO	OTEVŘENO	OTEVŘENO	ZAVŘENO

Obrázek 2.2:

Konečný automat je systém (či model systému), který může nabývat konečně mnoho (obvykle ne "příliš mnoho") stavů. Daný stav se mění na základě vnějšího podnětu (možných podnětů také není "příliš mnoho") s tím, že pro daný stav a daný podnět je jednoznačně určeno, jaký stav bude následující (tj. do jakého stavu systém přejde). Konkrétní konečný automat se často zadává diagramem, který také nazýváme *stavový diagram* či *graf automatu*. Jiná možnost zadání je *tabulkou*, která je sice poněkud suchopárnější, ale např. je vhodnější pro počítačové zpracování.

Příklad. Diagram na obr. 2.1 je popisem jednoduchého automatu řídícího vstupní dveře do supermarketu (dveře nejsou posuvné, ale otvírají se dovnitř). Automat může být ve dvou stavech (Zavřeno, Otevřeno) a podnětem (např. snímaným v pravidelných krátkých intervalech) je informace, na které z podložek (před dveřmi, za dveřmi) se někdo nachází. Kromě (pro naše oko přehledného) diagramu lze tutéž informaci sdělit tabulkou na obr. 2.2.

Úkol 1 *Popište slovně nějaký jednoduchý automat na mince, který je schopen vydat čaj nebo kávu dle volby, a pak jej modelujte stavovým diagramem (grafem automatu).*

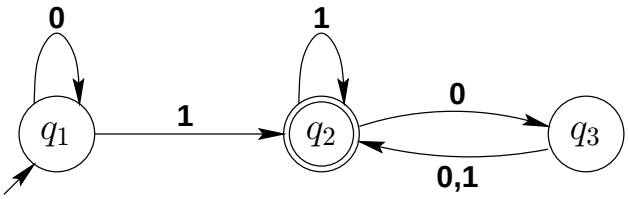
Na základě uvedeného jednoduchého příkladu máme již představu, co to konečný automat je. Formalizujme nyní tento pojem v jazyce matematiky. K čemu je to dobré ? Pro další zkoumání potřebujeme přesnou (a jednoznačnou) definici a také stručné a přehledné značení. (U takto jednoduchého pojmu bychom se bez formalizace snad ještě obešli, u složitějších pojmů později už těžko.)

Definice 2.0.1 ***Konečný automat**, zkráceně KA, A je pětice (tzn., že je dán pětic parametrů) $A = (Q, \Sigma, \delta, q_0, F)$, kde Q je konečná neprázdná množina **stavů**, Σ je konečná neprázdná množina zvaná (vstupní)* abeceda*, $\delta : Q \times \Sigma \to Q$ je* přechodová funkce*, $q_0 \in Q$ je **počáteční (iniciální) stav** a $F \subseteq Q$ je množina **přijímajících (koncových) stavů**.*

Všimněme si, že uvedená definice vyžaduje určení počátečního stavu a tzv. přijímajících (nebo též koncových) stavů – o těch dosud nebyla řeč. Vymezení těchto stavů je důležité v případě, o který se zvlášť budeme zajímat, tj. v případě, kdy konečný automat hraje roli rozpoznávače jazyka. Zatím řekneme jen neformálně: jazyk rozpoznávaný daným konečným automatem je množina posloupností symbolů (prvků abecedy), které automat přijímá (akceptuje), tj. těch posloupností, které automat převedou z počátečního do některého z přijímajících stavů.

Jak už víme, (malý) konečný automat lze často přehledně reprezentovat *grafem automatu*; dohodněme se, že počáteční stav budeme označovat malou vstupní šipkou, přijímající stavy pak dvojitým kolečkem.

Poznámka. V řeči teorie grafů bychom přesněji měli mluvit o ohodnoceném *multigrafu* – mezi dvěma vrcholy může vést více hran, ohodnocených prvky abecedy. Běžně ovšem takové hrany na obrázku znázorňujeme hranou jedinou, na níž uvedeme všechna příslušná ohodnocení; na obr. 2.3 to ilustruje hrana vedoucí ze stavu q_3 do stavu q_2 (zastupuje hranu s ohodnocením 0 i hranu s ohodnocením 1).



přijímá:
1101, 010101, . . .

nepřijímá:
0110, 0010, . . .

Obrázek 2.3:

Úkol 2 *Chápejme obr. 2.3 jako popis konečného automatu $A = (Q, \Sigma, \delta, q_0, F)$. Vypište* přímým výčtem *hodnoty všech parametrů automatu A. Pak porovnejte s obr. 2.7. Na obrázku je počáteční stav q_1 rovnou zapsán jako čtvrtý parametr místo q_0; mohli bychom to samozřejmě také řešit zápisem $q_0 = q_1$.*

Obr. 2.3 již čtenáři jistě dal představu o tom, co to znamená, že daný konečný automat přijímá danou posloupnost symbolů. Je ovšem opět užitečné a žádoucí pojmy zformalizovat. Začneme pojmem jazyk a souvisejícími definicemi; zároveň zavedeme i značení používané později.

Definice 2.0.2 ***Abeceda*** *je konečná neprázdná množina* symbolů *(písmen, znaků, signálů apod.). Často ji označujeme znakem Σ, její prvky pak znaky a, b, c (s případnými indexy apod.).*

Slovo *v abecedě Σ je libovolná konečná posloupnost $a_1, a_2, \ldots, a_n$ prvků Σ. Píšeme ji obvykle bez čárek – $a_1a_2 \ldots a_n$. Slova označujeme symboly u, v, w apod. Přirozeným způsobem definujeme délku slova, označujeme $|u|$; pro $u = a_1a_2 \ldots a_n$ je $|u| = n$. Symbolem $|u|_a$ označujeme počet výskytů symbolu a ve slově u.*

Speciální roli hraje ***prázdné slovo*** *(s délkou 0); označujeme jej ε.* Zřetězení slov *$u = a_1a_2 \ldots a_n$, $v = b_1b_2 \ldots b_m$ označujeme $u \cdot v$, stručněji uv, a definujeme $uv = a_1a_2 \ldots a_nb_1b_2 \ldots b_m$. Výrazem u^n označujeme n-násobné zřetězení slova u; je tedy $u^0 = \varepsilon$, $u^1 = u$, $u^2 = uu$, $u^3 = uuu$ atd.*

Slovo u je ***podslovem*** *slova v, jestliže v lze psát $v = w_1uw_2$ pro nějaká w_1, w_2 (jinak řečeno: jestliže existují slova w_1, w_2 taková, že $v = w_1uw_2$). Slovo u je* ***prefixem*** *(****sufixem****) slova v, jestliže $v = uw$ ($v = wu$) pro nějaké w.*

Σ^* označuje množinu všech slov a Σ^+ množinu všech neprázdných slov v abecedě Σ. (Je tedy $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.)

Formální jazyk*, stručně* ***jazyk****, nad abecedou Σ je libovolná množina slov v abecedě Σ. Jazyky obvykle označujeme L (s indexy); pro jazyk L nad abecedou Σ je tedy $L \subseteq \Sigma^*$.*

Příklad. Slova v abecedě $\Sigma = \{a, b\}$ jsou např. $\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba$.

Je užitečné si uvědomit, že slova v každé abecedě lze přirozeně uspořádat (a systematicky generovat) jak jsme naznačili v příkladu: v prvé řadě podle délky (kratší předchází delšímu) a v rámci stejné délky abecedně (lexikograficky) – což předpokládá, že máme uspořádány prvky abecedy. Např. pro $\Sigma = \{0, 1\}$ (s abecedním uspořádaním $0 < 1$) lze prvky Σ^* generovat v pořadí $\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$. Příslušné uspořádání budeme označovat $<_L$ (např. 11$<_L$ 001).

Vsuvka. Připomeňme i obecnou definici uspořádání: relace ρ na množině A je *(částečným) uspořádáním*, jestliže je reflexivní ($\forall x \in A : x\rho x$), tranzitivní ($\forall x, y, z \in A : (x\rho y \wedge y\rho z) \Longrightarrow x\rho z$)) a antisymetrická ($\forall x, y \in A : (x\rho y \wedge y\rho x) \Longrightarrow x = y$)). Hovoříme-li o *uspořádání*, obvykle myslíme úplné (neboli lineární) uspořádání, což znamená navíc $\forall x, y \in A : (x\rho y \vee y\rho x)$ (tedy neexistují vzájemně nesrovnatelné prvky).
Pro relace uspořádání se obvykle používají symboly jako $\leq$ a jemu podobné. Použijeme-li pak symbolu $<$, myslíme obvykle podrelaci "ostrého" (tj. irreflexivního) uspořádání definovanou takto: $x < y \iff_{df} x \leq y \wedge x \neq y$.

Říkáme-li pouze "jazyk", rozumíme tím, že příslušná abeceda je buď zřejmá z kontextu nebo na ní nezáleží.

Poznámka. Byť v praktických případech má abeceda např. desítky prvků, v našich příkladech bude abeceda často (jen) dvouprvková (většinou $\{a, b\}$, $\{0, 1\}$). Uvědomme si, že to není zásadní omezení, jelikož písmena víceprvkové abecedy lze přirozeně zakódovat řetězci dvouprvkové abecedy. (Jak dlouhé řetězce byste použili např. při kódování 256-ti prvkové abecedy ?)

Poznámka. Uvědomme si, že operace zřetězení slov je asociativní (tzn. $(u \cdot v) \cdot w = u \cdot (v \cdot w)$); proto je např. zápis $u \cdot v \cdot w$ jednoznačný i bez uvedení závorek.

Úkol 3 *Vypište všechna slova v abecedě $\{a, b\}$, která mají délku 3.*
Napište explicitně slovo u (posloupnost písmen), které je určeno výrazem $(ab)^3 \cdot ba \cdot (bba)^2$ (slovo u je tedy výsledkem provedení operací uvedených ve výrazu).
Vypište všechna slova délky 2, které jsou podslovy slova 00010 (v abecedě $\{0, 1\}$).
Vypište všechny prefixy (sufixy) slova 0010. (Je jich pět !)

Všimněme si ještě, že s jazyky—jakožto s množinami—je možné provádět množinové operace; např. výsledky operací $L_1 \cup L_2$ (sjednocení), $L_1 \cap L_2$ (průnik), $L_1 - L_2$ (množinový rozdíl), $\overline{L}$ (doplněk – rozumí se pro příslušnou abecedu Σ, tj. $\overline{L} = \Sigma^* - L$) jsou opět jazyky. Později zavedeme i speciální jazykové operace.

Úkol 4 *Uvažujme jazyky*
$L_1 = \{w \in \{a, b\} \mid w$ *obsahuje sudý počet výskytů symbolu* $a\}$,
$L_2 = \{w \in \{a, b\} \mid w$ *začíná a končí stejným symbolem* $\}$.
Vypište prvních šest slov (rozumí se v uspořádání $<_L$) postupně pro jazyky $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, $\overline{L}$.

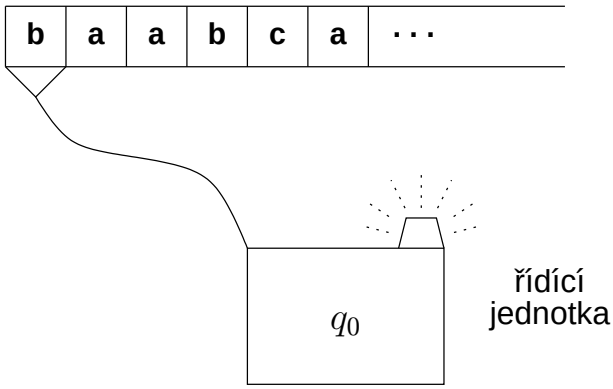
Pokračujeme přesnou definicí jazyka rozpoznávaného konečným automatem; příslušné jazyky budeme nazývat regulární.

Definice 2.0.3 *Přechodovou funkci automatu $A = (Q, \Sigma, \delta, q_0, F)$ zobecníme na funkci*
$\delta^* : Q \times \Sigma^* \to Q$ *touto induktivní definicí:*

1. $\delta^*(q, \varepsilon) = q$,
2. $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$.

	a	b
1	2	1
←2	2	1
3	7	5
←4	7	4
→5	2	4
←6	6	3
7	7	4

Obrázek 2.4:



Obrázek 2.5:

Úkol 5 *Automat A na obrázku 2.4 nejprve zadejte výčtem parametrů, $A = (Q, \Sigma, \delta, q_0, F)$.*
Z induktivní definice δ^* pak detailně odvoďte, čemu se rovná $\delta^*(2, \textbf{\textit{bab}})$.

Důkladně si formu a obsah induktivní definice promyslete. Zároveň se přesvědčte, že nemůže dojít k nedorozumění, když v dalším budeme místo δ^* psát jen δ, je totiž $\delta^*(q, a) = \delta(q, a)$. ($\delta^*$ je tedy rozšířením δ na větší definiční obor; z kontextu bude vždy zřejmé, odkazujeme-li se na δ v užším nebo širším smyslu.)

Definice 2.0.4 *Slovo $w \in \Sigma^*$ je* přijímáno *automatem A, jestliže $\delta(q_0, w) \in F$.* Jazykem rozpoznávaným (přijímaným) automatem A *rozumíme jazyk $L(A) = \{w \mid$ slovo w je přijímáno $A\}$.*

Jazyk L *je* regulární *(tj. rozpoznatelný konečným automatem), jestliže existuje KA A tž. $L(A) = L$.*

Úkol 6 *Ověřte si, že uvedená definice jazyka rozpoznávaného KA skutečně zachycuje (formalizuje) předcházející neformální vysvětlení. Dále zkuste formálně nadefinovat graf G_A automatu A a definnujte jazyk $L(A)$ pomocí (pojmů teorie grafů na) grafu G_A.*

Konečný automat si lze představovat různě. Pro naše účely je zřejmě nejvhodnější představa znázorněná na obr. 2.5. *Řídicí jednotka*, což je "skříňka" nabývající konečně mnoha (vnitřních) stavů (řekněme několika-bitová paměť), je *čtecí hlavou* spojena se (vstupní) *páskou*, na níž je zapsáno slovo – zleva doprava jsou v jednotlivých buňkách pásky uložena písmena daného slova.

Na začátku je řídicí jednotka v počátečním stavu a hlava je připojena k nejlevějšímu políčku pásky. Činnost automatu, zvaná *výpočet*, pak probíhá v *krocích*: v každém kroku je přečten symbol (hlava se po přečtení posune o jedno políčko doprava) a řídicí jednotka se nastaví do stavu určeného aktuálním stavem a přečteným symbolem (přechodová funkce je v řídicí jednotce "zadrátovaná").

Je možné si také představit, že na řídicí jednotce je "světélko" signalizující navenek, zda aktuální stav je či není přijímající (čili "zvenku" rozlišujeme u řídicí jednotky jen dva stavy – přijímá/nepřijímá).

Uvedená prezentace konečného automatu vede k další variantě definice přijímaných slov. Uvedeme ji teď hlavně proto, že pozdější definice pro další modely budou pak jen přímočarým zobecněním. Všimněte si také, že přitom budeme explicitně definovat pojem *výpočtu* automatu.

Vsuvka. Připomeňme nejdříve některé elementární pojmy teorie množin a algebry. *Kartézským součinem* $M \times N$ množin M, N rozumíme množinu dvojic $M \times N = \{\,(a,b) \mid a \in M, b \in N\,\}$. *(Kartézské) mocniny* množiny M lze definovat takto: $M^1 = M, M^2 = M \times M, M^3 = M \times (M \times M), \ldots, M^{i+1} = M \times M^i, \ldots$. *$n$-ární relace na množině* M je podmnožina M^n. Nejčastěji uvažované relace jsou 2-ární (neboli binární), proto pojmem *relace ρ na množině* M rozumíme *binární relaci*, tedy $\rho \subseteq M \times M$; přitom často píšeme $x\rho y$ místo $(x,y) \in \rho$. Relace ρ na množině M je *reflexivní* právě tehdy, když $\forall x \in M : x\rho x$; ρ je *tranzitivní* právě tedy, když $\forall x, y, z \in M : (x\rho y \wedge y\rho z) \Longrightarrow x\rho z$. *Reflexivním a tranzitivním uzávěrem* relace ρ rozumíme nejmenší relaci ρ' (nejmenší ve smyslu množinové inkluze), která obsahuje ρ a je přitom reflexivní i tranzitivní.

Definice 2.0.5 ***Konfigurací konečného automatu*** $A = (Q, \Sigma, \delta, q_0, F)$ *rozumíme dvojici* (q, w)*, kde* $q \in Q$ *a* $w \in \Sigma^*$ *(q představuje aktuální stav a w slovo, které zbývá přečíst – připomeňme, že čtecí hlava se pohybuje jen doprava).*

Na množině všech konfigurací automatu A definujeme relaci $\vdash_A$ takto: $(q, aw) \vdash_A (q', w)$ *právě když* $\delta(q, a) = q'$ *(rozumí se $a \in \Sigma$, $w \in \Sigma^*$). Zápis $K_1 \vdash_A K_2$ čteme např. "konfigurace K_1 bezprostředně (tj. v jednom kroku) vede ke konfiguraci K_2", "konfigurace K_2 bezprostředně následuje za K_1" apod.*

Výpočtem *automatu A, začínajícím v konfiguraci K, rozumíme posloupnost konfigurací* $K_0, K_1, K_2, \ldots, K_n$*, kde* $K_0 = K$ *a* $K_i \vdash_A K_{i+1}$ *pro* $i = 0, 1, \ldots, n-1$ *(takový výpočet má délku n, tj. sestává z n kroků).*

Výpočet $K_0, K_1, K_2, \ldots, K_n$ *je* přijímajícím výpočtem *pro slovo w, jestliže* $K_0 = (q_0, w)$ *a* $K_n = (q, \varepsilon)$ *pro nějaký* $q \in F$*.*

Slovo $w \in \Sigma^*$ je přijímáno *KA A, jestliže existuje přijímající výpočet pro slovo w.*

Alternativně lze také definovat:

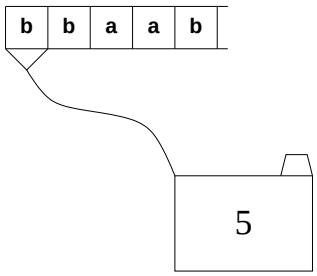
Relace $\vdash_A^*$ *je reflexivním a tranzitivním uzávěrem relace* $\vdash_A$*.* $K_1 \vdash_A^* K_2$ *pak čteme např. takto: "konfigurace K_1 vede ke K_2", "K_1 odvodí K_2", "K_2 je následníkem K_1" apod.*

Slovo $w \in \Sigma^*$ je přijímáno *KA A, jestliže* $(q_0, w) \vdash^* (q, \varepsilon)$ *pro nějaký přijímající stav* $q \in F$*. (Místo* $\vdash_A^*$ *píšeme jen* $\vdash^*$*, jestliže automat A, k němuž se daná relace vztahuje, je zřejmý z kontextu.)*

Poznámka. Všimněme si, že zápisy $\delta(q,w) = q'$ a $(q,w) \vdash^* (q',\varepsilon)$ mají stejný význam. Šlo by se také např. dohodnout, že $(q,w) \vdash^* (q',\varepsilon)$ budeme zapisovat elegantněji $q \xrightarrow{w} q'$ apod. Čtenář je vyzýván, ať si další definice a tvrzení používající notaci s δ přeformuluje i dalšími uvedenými způsoby. Tím mj. vyzdvihujeme obecný fakt, že totéž sdělení (tutéž sémantiku, tj. tentýž význam pojmů, tvrzení apod.) lze zachytit různými syntaktickými způsoby (značeními), z nichž každý může mít své výhody a nevýhody.

Úkol 7 *Pro automat A na obrázku 2.4*

*- simulujte krok po kroku výpočet na slově **bbaab** (zapište příslušnou posloupnost konfigurací; první, tedy $(5, bbaab)$, je zachycena na obrázku 2.6).*



Obrázek 2.6: Počáteční konfigurace automatu A

- vypište všechna slova délky ≤ 3 v abecedě $\{\mathbf{a},\mathbf{b}\}$ a zjistěte, která z nich patří do jazyka $L(A)$.

- zakreslete graf (stavový diagram) automatu A

- charakterizujte co nejjednodušeji jazyk $L(A)$ (vlastností slov do něj náležejících)

Lze snadno nahlédnout, že pro jazyk $L(A)$ hrají roli jen ty stavy automatu A, které jsou dosažitelné (z počátečního stavu):

Definice 2.0.6 Stav q *automatu* (Q,Σ,δ,q_0,F) *je* dosažitelný*, jestliže existuje* $w \in \Sigma^*$ *tž.* $\delta(q_0,w) = q$.

Jako procvičení formy induktivní definice můžeme dosažitelnost uvést také takto: Množina *dosažitelných stavů* automatu (Q,Σ,δ,q_0,F) je *nejmenší* množina $K \subseteq Q$ splňující tyto dvě podmínky: 1/ $q_0 \in K$, 2/ jestliže $q \in K$ a $q' = \delta(q,a)$ pro nějaké $a \in \Sigma$, potom $q' \in K$.

Tato definice je de facto přímým návodem k sestavení algoritmu, který zjistí všechny dosažitelné stavy. Připomeňme, že konečný automat můžeme zadat přímým výčtem všech parametrů (např. automat z obr. 2.3 je takto popsán na obr. 2.7), ale také grafem (stavovým diagramem) či tabulkou (do té je rovněž nutno přidat označení počátečního a přijímacích stavů).

$A = (Q,\Sigma,\delta,q_1,F)$, kde

$$\begin{array}{rcll} Q & = & \{q_1,q_2,q_3\} & \delta(q_1,0) = q_1, \quad \delta(q_1,1) = q_2, \\ \Sigma & = & \{0,1\} & \delta(q_2,0) = q_3, \quad \delta(q_2,1) = q_2, \\ F & = & \{q_2\} & \delta(q_3,0) = q_2, \quad \delta(q_3,1) = q_2 \end{array}$$

Obrázek 2.7:

Úkol 8 *Zformulujte algoritmus, který pro daný KA, reprezentovaný grafem, označí všechny dosažitelné stavy; algoritmus pak zformulujte pro případ reprezentace tabulkou.*
Aplikujte na automat z obrázku 2.4.

Jak jsme už zmínili, odstraníme-li nedosažitelné stavy (a příslušně tedy omezíme definiční obor přechodové funkce), rozpoznávaný jazyk se nemění. Máme tedy

Tvrzení 2.0.7 *Ke každému KA A lze zkonstruvat KA A', v němž každý stav je dosažitelný a $L(A') = L(A)$.*

Z tvrzení 2.0.7 snadno nahlédneme, že

Věta 2.0.8 *Existuje algoritmus, který pro zadaný KA A rozhodne, zda $L(A)$ je neprázdný.*

(Jak vypadá ten algoritmus ?)
Pro čtenáře teď bude jistě snadné ukázat i dva následující fakty:

Tvrzení 2.0.9 *Pro konečný automat A s n stavy je $L(A)$ neprázdný právě tehdy, když existuje $w \in L(A)$ délky menší než n ($|w| < n$).*

Tvrzení 2.0.10 *Pro konečný automat A s n stavy je $L(A)$ nekonečný právě tehdy, když existuje $w \in L(A)$ splňující $n \leq |w| < 2n$.*

Úkol 9 *Načrtněte* rychlý *algoritmus, rozhodující pro daný A, zda $L(A)$ je nekonečný.*

Doplnění:

Víme, že i nekonečné množiny mohou mít různou mohutnost. (Množinu přirozených čísel nazýváme spočetnou, množinu reálných čísel nespočetnou.) V této souvislosti je užitečné si uvědomit následující fakty.

Z uspořádání $<_L$ je zřejmé, že

Tvrzení 2.0.11 *Pro (konečnou neprázdnou) abecedu Σ je Σ^* nekonečná spočetná množina (tj. existuje bijekce, neboli vzájemně jednoznačné zobrazení, mezi Σ^* a množinou $\mathbb{N}$ všech přirozených čísel).*

Poznámka. Všimněme si, že ze spočetnosti Σ^* (pro lib. abecedu Σ) plyne např. spočetnost množiny všech racionálních čísel. (Proč ?)

Z obecné Cantorovy věty, která říká, že mohutnost množiny M je ostře menší než mohutnost množiny všech jejích podmnožin $\mathcal{P}(M)$, plyne

Tvrzení 2.0.12 *Jazyků nad (neprázdnou) abecedou Σ je nespočetně mnoho.*

Např. už z těchto úvah o mohutnostech množin plyne, že ne každý jazyk je regulární – konečných automatů je totiž spočetně mnoho ! (Proč ?)

Normovaný tvar konečného automatu

V dalším občas využijeme (speciálně pak u zkušebních testů !) jistý přirozeně definovaný normovaný tvar konečného automatu, v němž jsou všechny stavy dosažitelné. Lze si to představit tak, že každému stavu q přiřadíme nejkratší slovo (přesněji: nejmenší slovo vzhledem k uspořádání $<_L$, pomocí něhož je q dosažitelný (z počátečního stavu), a stavy pak uspořádáme podle těchto přiřazených slov, přičemž je označujeme čísly $1, 2, \ldots, n$. Přesněji můžeme definovat takto:

Definice 2.0.13 *Máme-li dán konečný automat $A = (Q, \Sigma, \delta, q_0, F)$ bez nedosažitelných stavů a uspořádání (prvků) abecedy Σ (které indukuje uspořádání $<_L$ na Σ^*), pak řekneme, že A je v **normovaném tvaru**, jestliže*

- $Q = \{1, 2, \ldots, n\}$ *(pro nějaké $n \geq 1$),*
- 1 *je počáteční stav,*
- *označíme-li pro každý stav $i \in \{1, 2, \ldots, n\}$ symbolem u_i nejmenší slovo v uspořádání $<_L$, pro něž $\delta(1, u_i) = i$, pak pro $i < j$ platí $u_i <_L u_j$.*

Úkol 10 *Rozmyslete si jednoduchý algoritmus, který každý KA převede (přejmenováním stavů) do normovaného tvaru.*
Návod: počáteční stav označíme 1. Dále, např. v případě abecedy $\{a, b\}$, zjistíme stav q, do něhož automat přejde ze stavu 1 symbolem a; když q není označen (jako 1), označíme jej 2. Pak zjistíme stav q, do něhož automat přejde ze stavu 1 symbolem b; když q není dosud označen, označíme jej nejmenším dosud nepoužitým číslem. Takto jsme "vyřídili" stav 1, pokračujeme "vyřízením" 2 atd.
Aplikujte na automat z obrázku 2.4; všimněte si přitom, že se takto automaticky vymezíme (označíme) právě všechny dosažitelné stavy

Úkol 11 *Navrhněte konečný automat s abecedou $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \langle\text{RESET}\rangle\}$, který přijímá ta slova, kde součet symbolů **0**, **1**, **2** (braných numericky) za posledním symbolem $\langle\text{RESET}\rangle$ (či od začátku, není-li $\langle\text{RESET}\rangle$) dává při dělení 3 zbytek 1.*

Úkol 12 *Navrhněte konečný automat s abecedou $\{\mathbf{0}, \mathbf{1}\}$ přijímající právě ta slova, v nichž je sudý počet (výskytů) symbolů **0** a každý symbol **1** je bezprostředně následován symbolem **0**.*
Dokažte správnost vašeho návrhu, tedy vysvětlete, proč vámi navržený automat splňuje daný požadavek.

Úkol 13 *Navrhněte konečný automat přijímající právě ta slova v abecedě $\{a, b\}$, u nichž prefix délky 2 se rovná sufixu délky 2 (slova mají délku alespoň 2).*
Snažte se, aby tento automat měl minimální počet stavů – můžete se přitom také pokusit vysvětlit, proč vámi navržený počet stavů nelze zmenšit.

Úkol 14 *Pro konečný automat $A = (Q, \Sigma, \delta, q_0, F)$, $\Sigma = \{a, b\}$, uveďte induktivní definici funkce $R_a : \mathcal{P}(Q) \to \mathcal{P}(Q)$, pro niž, neformálně řečeno, platí toto:*

stav q patří do $R_a(K)$ právě tehdy, když se do q lze dostat z nějakého $q' \in K$ jen pomocí a-šipek (tedy nějakým slovem tvaru a^i).

Úkol 15 *Navrhněte konečný automat rozpoznávající jazyk $L_1 = \{\, w \in \{a, b\}^* \mid w$ obsahuje podslovo $aba \,\}$ a konečný automat rozpoznávající jazyk $L_2 = \{\, w \in \{a, b\}^* \mid |w|_b \bmod 2 = 0 \,\}$ (v L_2 jsou tedy právě slova obsahující sudý počet b-ček).*
Pak zkonstuujte automat rozpoznávající jazyk $L_1 \cap L_2$; zkuste přitom vhodně využít automaty zkonstruované pro jazyky L_1, L_2.

Úkol 16 *Na vybraných zkonstruovaných automatech si procvičte převod do normovaného tvaru.*

Kapitola 3

Návrh konečných automatů. Regulární operace.

Cíl kapitoly:
Procvičení návrhu složitějších automatů modulárním postupem. Pochopení jazykových operací zřetězení a iterace.

Studijní cíle: Procvičení návrhu složitějších automatů modulárním postupem. Pochopení jazykových operací zřetězení a iterace.

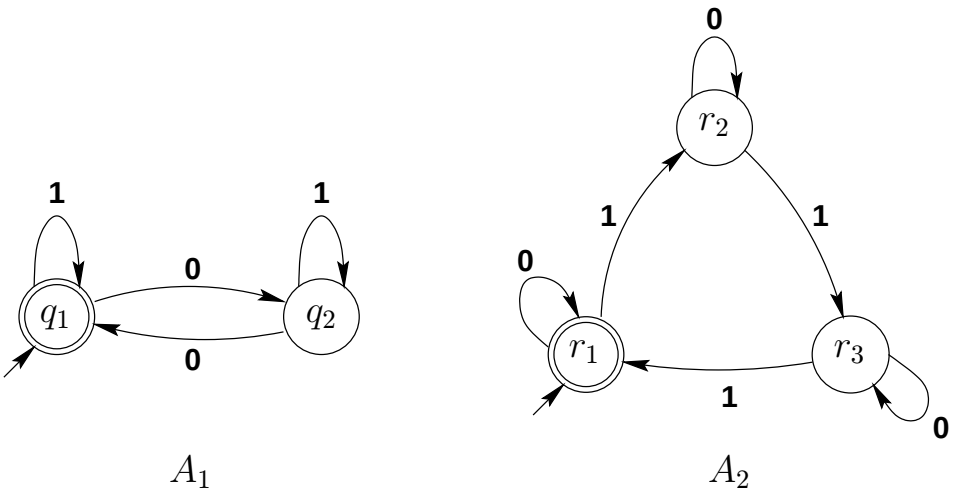
Návrh konečného automatu, který bude rozpoznávat zadaný jazyk, je tvůrčí činnost, vlastně jakési jednoduché programování, a proto nelze podat "mechanický" návod, jak automaty vytvářet. Stačí ovšem důkladně promyslet pár příkladů, aby člověk s programátorskou zkušeností (a tedy schopný "vžít se do role konečného automatu") tuto činnost zvládl.

Mnoho věcí se přitom zmechanizovat (zalgoritmizovat) dá. Např. existují algoritmy, které z automatů pro "jednodušší" jazyky vytvářejí automaty pro jazyky vzniklé operacemi nad oněmi (jednodušími) jazyky.

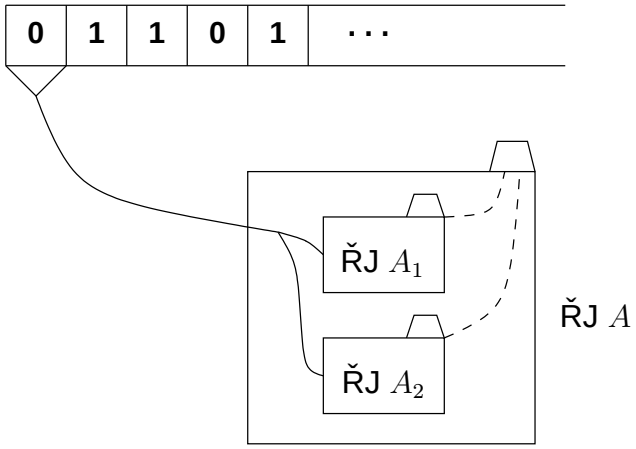
Představme si např., že chceme sestrojit automat, který má rozpoznávat jazyk, jenž je sjednocením dvou regulárních jazyků. Pro konkrétnost např. jazyk sestávající ze slov v abecedě $\{0,1\}$, v nichž počet nul je dělitelný dvěma nebo počet jedniček je dělitelný třemi. Tedy jazyk $L = L_1 \cup L_2$, kde $L_1 = \{w \in \{0,1\}^* \mid |w|_0$ je dělitelné $2\}$ a $L_2 = \{w \in \{0,1\}^* \mid |w|_1$ je dělitelné $3\}$. (Označením $|w|_a$ značíme počet výskytů symbolu a ve slově w.) Na obr. 3.1 jsou znázorněny automaty rozpoznávající jazyky L_1 a L_2.

Jak rozpoznáme, zda dané slovo patří do $L(A_1) \cup L(A_2)$? Prostě je necháme zpracovat oběma automatům A_1, A_2 a podíváme se, zda alespoň jeden z nich skončil v přijímajícím stavu. Ovšem tuto naši činnost může očividně provádět i jistý konečný automat A – viz obr. 3.2. Rozmyslete si, co jsou stavy automatu A, co je to počáteční a koncový stav, a jak vypadá přechodová funkce.

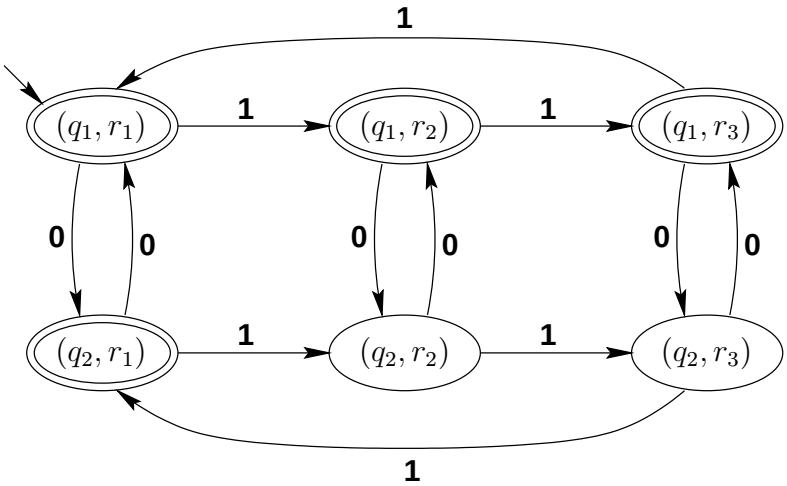
Pak se podívejte na obr. 3.3, znázorňující A pro náš konkrétní případ, a přesvědčte se, že to, co jste pochopili a co jste si odvodili, je přesně to, co je díky matematické notaci přesně a velmi stručně zachyceno v důkazu následující věty.



Obrázek 3.1:



Obrázek 3.2:



Obrázek 3.3:

Věta 3.0.14 *Jestliže jazyky $L_1, L_2 \subseteq \Sigma^*$ jsou regulární, pak také jazyk $L_1 \cup L_2$ je regulární.*

Důkaz: Nechť $L_1 = L(A_1)$, $L_2 = L(A_2)$ pro konečné automaty $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$.
Definujme automat $A = (Q, \Sigma, \delta, q_0, F)$ tž.

- $Q = Q_1 \times Q_2$,
- $\delta(\,(q_1, q_2), a\,) = (\,\delta_1(q_1, a), \delta_2(q_2, a)\,)$ pro vš. $q_1 \in Q_1$, $q_2 \in Q_2$, $a \in \Sigma$,
- $q_0 = (q_{01}, q_{02})$,
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.

Je očividné (exaktně lze ukázat např. indukcí podle délky w), že pro lib. $q_1 \in Q_1$, $q_2 \in Q_2$ a $w \in \Sigma^*$ je $\delta(\,(q_1, q_2), w\,) = (\,\delta_1(q_1, w), \delta_2(q_2, w)\,)$.
Z toho snadno plyne, že $L(A) = L_1 \cup L_2$. □

Úkol 17 *Mějme konečné automaty bez specifikace počátečních a přijímajících stavů $A_1 = (Q_1, \Sigma, \delta_1)$, $A_2 = (Q_2, \Sigma, \delta_2)$.*
Definujme $A = (Q, \Sigma, \delta)$ tž. $Q = Q_1 \times Q_2$ a $\delta(\,(q_1, q_2), a\,) = (\,\delta_1(q_1, a), \delta_2(q_2, a)\,)$ pro vš. $q_1 \in Q_1$, $q_2 \in Q_2$, $a \in \Sigma$.
Ukažte, že pro lib. $q_1 \in Q_1$, $q_2 \in Q_2$ a $w \in \Sigma^*$ platí

$$\delta(\,(q_1, q_2), w\,) = (\,\delta_1(q_1, w), \delta_2(q_2, w)\,).$$

Použijte indukci podle délky slova w.

Na základě (důkazu) věty 3.0.14 pro sjednocení teď ukažte analogickou větu pro průnik:

Věta 3.0.15 *Jestliže jazyky $L_1, L_2 \subseteq \Sigma^*$ jsou regulární, pak také jazyk $L_1 \cap L_2$ je regulární.*

(Nápověda: $F = (F_1 \times F_2)$)

Poznámka. Je velmi vhodné si uvědomit *konstruktivnost* našich tvrzení. Např. věta 3.0.14 (věta 3.0.15) říká jen to, že sjednocení (průnik) dvou regulárních jazyků je také regulární jazyk. Dokázali jsme však více: existuje *algoritmus*, který ke konečným automatům rozpoznávajícím L_1, L_2 zkonstruuje automat rozpoznávající $L_1 \cup L_2$ ($L_1 \cap L_2$). Podobně tomu bude u dalších vět v tomto kursu, i když se o tom třeba nebudeme explicitně zmiňovat.

Úkol 18 *Automat pro průnik požadovaný v úkolu 15 sestrojte podle obecného návodu z (důkazu) předchozí věty. (Porovnejte se svou předchozí konstrukcí.)*

Zvláštní roli (zmíněnou ještě později) hrají tzv. regulární operace. Vedle (množinové operace) *sjednocení* k nim patří další dvě (jazykové) operace – *zřetězení* a *iterace*. (Průnik mezi regulární operace nezařazujeme. Důvody vysvitnou později.)

Definice 3.0.16 ***Regulárními operacemi*** *s jazyky nazýváme operaci* ***sjednocení****,* $L_1 \cup L_2 = \{w \mid w \in L_1$ *nebo* $w \in L_2\}$*,* ***zřetězení****,* $L_1 \cdot L_2 = \{uv \mid u \in L_1, v \in L_2\}$*, a* ***iterace****,* $L^* = \bigcup_{n=0}^{\infty} L^n$*, kde* L^n *je definováno induktivně:* $L^0 = \{\varepsilon\}$*,* $L^{n+1} = L \cdot L^n$*.*

Předpokládáme, že čtenáři už nebude dělat potíže pochopit zaváděné pojmy pouze z jejich matematické definice. Odvodí si tak např., že do *zřetězení* dvou jazyků patří právě každé takové slovo, které vznikne, když za nějaké slovo z prvního jazyka postavíme nějaké slovo z druhého jazyka (zřetězíme je). Jinými slovy: takové slovo se dá rozdělit na dvě části, z nichž první patří do prvního a druhá do druhého jazyka. Mj. si všimneme $\emptyset \cdot L = L \cdot \emptyset = \emptyset$, $\{\varepsilon\} \cdot L = L \cdot \{\varepsilon\} = L$.

Rovněž vidíme, že do *iterace* jazyka L patří právě každé takové slovo, jež lze rozdělit na několik částí, přičemž každá z nich patří do L – tj. vznikne zřetězením několika (konečně mnoha) slov z L. Definice nám také říká, že prázdné slovo patří do iterace vždy (vznikne "zřetězením nula slov z L").

Poznámka. Formálně lze L^* definovat také např. takto: $L^* = \{\, w \mid$ ex. $n \geq 0$ a slova $u_1, u_2, \ldots, u_n \in L$ tak, že $w = u_1 u_2 \ldots u_n \,\}$.

Všimněme si např., že $\emptyset^* = \{\varepsilon\}$, $(L^*)^* = L^*$ apod. Rovněž si uvědomme, že naše dříve zavedené značení Σ^* je v souladu s nyní uvedenou definicí iterace.

Úkol 19 *Procvičte si definici zřetězení a iterace jazyků na malých příkladech a pak dokažte či vyvraťte obecnou platnost následujících vztahů:*

- $L_1 \cdot L_2 = L_2 \cdot L_1$
- $L_1(L_2 \cup L_3) = L_1 L_2 \cup L_1 L_3$
- $(L_1 \cup L_2)^* = L_1^*(L_2 \cdot L_1^*)^*$
- $(L_1 \cap L_2)^* = L_1^* \cap L_2^*$

Kapitola 4

Nedeterministické konečné automaty a jejich vztah k deterministickým.

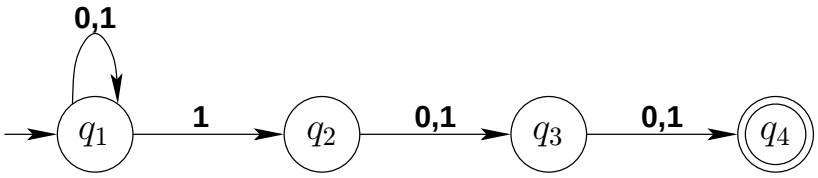
Cíl kapitoly:
Pochopení pojmu nedeterministického výpočtu a role nedeterminismu v usnadnění návrhu konkrétních automatů. Zvládnutí algoritmu převodu nedeterministického automatu na deterministický.

Studijní cíle: Pochopení pojmu nedeterministického výpočtu a role nedeterminismu v usnadnění návrhu konkrétních automatů. Zvládnutí algoritmu převodu nedeterministického automatu na deterministický.

Uvažujme nyní obdobu věty 3.0.14 pro zřetězení jazyků. Jistě nás napadne přístup: “necháme na první část slova běžet první automat, v přijímajícím stavu pak předáme řízení druhému automatu a ten dočte slovo do konce”. Problém je, že obecně nepostačí prostě předat řízení při *prvním* příchodu do přijímajícího stavu (proč ne ?), ale je nutno nechat to jako *možnost při jakémkoli* příchodu do přijímajícího stavu.

Toto chování snadno realizujeme automatem, v němž připustíme *nedeterminismus* – v daném stavu a při daném vstupním symbolu je obecně více možností, do kterého stavu přejít. V našem případě by se nám ještě více hodilo, aby šlo změnit stav, aniž se čte vstupní symbol (při onom “předání řízení”); tato možnost se objeví u ZNKA níže. Nejprve začneme s následující definicí nedeterministického konečného automatu ($\mathcal{P}(Q)$ označuje množinu všech podmnožin množinu Q):

Definice 4.0.17 ***Nedeterministický konečný automat****, zkráceně NKA, A je dán pěticí parametrů $A = (Q, \Sigma, \delta, I, F)$, kde Q je konečná neprázdná množina* stavů*, Σ je (konečná neprázdná)* abeceda*, $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ je* přechodová funkce*, $I \subseteq Q$ je množina* počátečních (iniciálních) stavů *a $F \subseteq Q$ je množina* přijímajících (koncových) stavů*.*



přijímá např.:
01101

nepřijímá např.:
011010

Obrázek 4.1:

Přímočaře lze opět nadefinovat *graf* (též nazývaný stavový diagram) NKA. Příklad takového grafu je na obrázku 4.1 – nadefinujte takový graf obecně pro NKA A ! (Mělo by být zřejmé, že pro NKA $A = (Q, \Sigma, \delta, I, F)$ znázorněný grafem na obrázku 4.1 je např. $\delta(q_1, 1) = \{q_1, q_2\}$, $\delta(q_2, 0) = \{q_3\}$, $\delta(q_4, 1) = \emptyset$ apod.)

Definice tedy připouští, že z daného stavu q pro daný symbol a vychází více "a-šipek" (nebo někdy žádná a-šipka), a také připouští více počátečních stavů.

Obr. 4.1 ukazuje k danému NKA také příklad přijímaného a nepřijímaného slova. Zkuste z toho vyvodit, jak je definováno přijímání slova nedeterministickým konečným automatem, a pak teprve svůj závěr konfrontujte s dále uvedenou definicí.

Pro definici *výpočtu* NKA lze takřka doslova použít definici 2.0.5 – jen místo $\delta(q, a) = q'$ napíšeme $\delta(q, a) \ni q'$ a místo q_0 použijeme např. $r, r \in I$. (Promyslete si podrobně tuto modifikovanou definici, speciálně si promyslete definici přijímání slova v tomto novém kontextu !)

Vidíme tedy, že na rozdíl od KA (užíváme též DKA, když chceme zdůraznit, že máme na mysli standardní, tj. deterministický automat), kde k danému slovu existuje jediný úplný (tj. dokončený, neprodloužitelný) výpočet, u NKA existuje k danému slovu obecně více úplných výpočtů. Rozhodující pro příslušnost slova w k jazyku $L(A)$ (rozpoznávanému NKA A) ovšem je, zda existuje *alespoň jeden* přijímající výpočet pro w.

Přijímání slova a jazyk $L(A)$ lze nadefinovat také takto: ¨

Definice 4.0.18 Slovo $w \in \Sigma^*$, *tvaru* $w = a_1 a_2 \dots a_n$, *je* přijímáno *NKA* $A = (Q, \Sigma, \delta, I, F)$, *jestliže existují stavy* $q_0, q_1, \dots, q_n$ *takové, že 1/* $q_0 \in I$, *2/* $q_n \in F$, *3/* $\delta(q_{i-1}, a_i) \ni q_i$ *pro vš.* $i = 1, 2, \dots, n$. Jazykem rozpoznávaným (přijímaným) automatem A *rozumíme jazyk* $L(A) = \{w \mid$ *slovo* w *je přijímáno* $A\}$. Jazyk L *je* rozpoznatelný nedeterministickým konečným automatem *jestliže existuje NKA* A *tž.* $L(A) = L$.

Poznámka. Velmi názorně lze definovat přijímání slova rovněž v termínech grafu NKA. (Jak ?) Všimněme si také explicitně, že dříve uvedený KA lze chápat jako speciální případ NKA (kde $\delta(q, a)$ je vždy jednoprvková množina a také I je jednoprvková množina).
Jak už jsme zmínili dříve„ místo konečný automat (KA) někdy pro zdůraznění říkáme *deterministický* konečný automat (DKA).

Úkol 20 *Charakterizujte (co nejjednodušeji slovně) jazyk, který je přijímán automatem z obrázku 4.1.*
Zkonstruujte deterministický KA, který přijímá tentýž jazyk.

	0	1
→1	1,2	1
2	3	-
3	-	4
←4	-	-
→5	5	5,6
6	-	7
7	-	8
8	-	9
←9	9	9

Obrázek 4.2:

Úkol 21 *Sestrojte co nejjednodušší* nedeterministický *konečný automat, který přijímá právě ta slova v abecedě* $\{0,1\}$*, jež začínají* 110 *nebo končí* 001 *nebo obsahují* 1111*.*

Bude nám rovněž užitečná následující definice:

Definice 4.0.19 *Definiční obor přechodové funkce v NKA* $A = (Q, \Sigma, \delta, I, F)$ *můžeme opět přirozeně zobecnit na* $\delta : Q \times \Sigma^* \longrightarrow \mathcal{P}(Q)$*, resp. ještě obecněji na* $\delta : \mathcal{P}(Q) \times \Sigma^* \longrightarrow \mathcal{P}(Q)$*; a sice induktivní definicí: 1/* $\delta(K, \varepsilon) = K$*, 2/* $\delta(K, wa) = \bigcup_{q \in \delta(K,w)} \delta(q, a)$*.*

Definici si opět řádně promyslete; speciálně si objasněte, co znamená $\delta(K_1, w) = K_2$. Také si všimněme, že $\delta(I, w)$ je množina právě těch stavů, do kterých se NKA může dostat zpracováním (přečtením) slova w (začne-li v některém z počátečních stavů). Můžeme tedy také psát $L(A) = \{w \in \Sigma^* \mid \delta(I, w) \cap F \neq \emptyset\}$.

Úkol 22 *NKA zadaný tabulkou na obrázku 4.2 zadejte grafem.*
Pak si na něm ilustrujte funkci $\delta : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ *pro zvolené argumenty.*

Rozšíření funkce δ u NKA (viz definici 4.0.19) v podstatě demonstruje, že nedeterministický konečný automat lze nahradit deterministickým – byť (obvykle) s podstatně větším počtem stavů:

Věta 4.0.20 *NKA rozpoznávají právě regulární jazyky (a jsou v tomto smyslu ekvivalentní DKA).*

Důkaz: Jelikož každý DKA je de facto speciálním případem NKA, je zřejmé, že každý regulární jazyk je rozpoznáván nějakým NKA.

Pro opačný směr stačí ukázat konstrukci (algoritmus), která pro zadaný NKA $A = (Q, \Sigma, \delta, I, F)$ vydá DKA A_1 tž. $L(A) = L(A_1)$. Stačí definovat $A_1 = (\mathcal{P}(Q), \Sigma, \delta_1, I, F_1)$, kde pro lib. $K \subseteq Q$ (tj. $K \in \mathcal{P}(Q)$) položíme $\delta_1(K, a) = \delta(K, a)$ (zde se odkazujeme k oné rozšířené definici δ) a dále $F_1 = \{K \subseteq Q \mid K \cap F \neq \emptyset\}$.

Důkaz $L(A) = L(A_1)$ je zřejmý:

Pro lib. slovo w platí: $w \in L(A) \iff \delta(I, w) \cap F \neq \emptyset \iff \delta_1(I, w) \in F_1 \iff w \in L(A_1)$. □

Chování DKA A_1 je užitečné si představit ve formě "knoflíkové hry" na grafu NKA A: Na začátku leží knoflíky právě na uzlech odpovídajících I. Přijde-li nyní (přečte-li se) symbol a, každý knoflík se posune podle všech možných přechodů (šipek) a – přitom se knoflík může "rozmnožit" či "zmizet". Potom na každém uzlu, na kterém je více než jeden knoflík, ponecháme knoflík jediný – a jsme připraveni přijmout další vstupní symbol. Daný stav (dané rozmístění knoflíků) je přijímající právě když alespoň jeden knoflík leží na uzlu odpovídajícím některému přijímajícímu stavu automatu A.

Všimněte si, že důkaz věty 4.0.20 ukazuje pro NKA s n stavy konstrukci DKA s 2^n stavů (!) Některé stavy u tohoto DKA mohou být ovšem nedosažitelné (tedy některých rozmístění knoflíků nelze v předchozí hře docílit) a omezení se jen na konstrukci dosažitelných stavů může někdy znamenat obrovskou úsporu.

Úkol 23 *Rozmyslete si, jak lze přímočaře konstruovat např. tabulku, která bude obsahovat jen dosažitelné stavy onoho DKA.*
Aplikujte tuto metodu na NKA z obrázku 4.2.

Bohužel jsou případy, kdy všechny stavy DKA dosažitelné jsou a navíc nelze tyto stavy "uspořit" ani jiným způsobem (jedná se o tzv. minimální, nebo též redukovaný, automat, jak o tom budeme hovořit později).

Úkol 24 *Např. k NKA s 5 stavy zadanému následující tabulkou*

	a	*b*
↔*1*	*2*	-
2	*3*	*1,2*
3	*4*	*1,3*
4	*5*	*1,4*
5	*1*	*1,5*

se vám nepodaří nalézt ekvivalentní DKA (tj. DKA rozpoznávající tentýž jazyk) s méně než $2^5 = 32$ stavy. (Zkuste zjistit, proč !)
Konstrukci přitom snadno zobecníte pro NKA s n stavy, pro nějž nejmenší ekvivalentní DKA má 2^n stavů.

Poznámka. Uvědomme si ovšem, že ke každému NKA s n stavy (např. $n = 1000$) lze příslušný DKA realizovat (např. simulovat na počítači) za použití (pouze) n bitů, byť má daný DKA 2^n stavů. (Jak ?) Čili tento úkol je zvládnutelný, byť by explicitně zkonstruovaný stavový prostor DKA vůbec nebyl uložitelný do paměti počítače ! (Řešit ovšem např. problém dosažitelnosti stavu v DKA při zmíněné n-bitové reprezentaci je pak úplně jiná otázka !)

Vraťme se nyní k úvahám o (nedeterministickém) automatu pro zřetězení dvou regulárních jazyků a připomeňme si, že se nám hodí více jiný druh nedeterminismu – tzv. ε-přechody, díky nimž automat může změnit stav, aniž čte vstupní symbol:

Definice 4.0.21 ***Zobecněný nedeterministický konečný automat*** *(ZNKA) A je dán pěticí parametrů $A = (Q, \Sigma, \delta, I, F)$. Jediný rozdíl proti NKA spočívá v tom, že přechodová funkce je typu $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ a přijímání slova je definováno takto:*

Slovo $w \in \Sigma^*$, *tvaru* $w = a_1 a_2 \dots a_n$, *je* přijímáno *ZNKA A, jestliže existují stavy* $q_1^0, q_2^0, \dots, q_{m_0}^0, q_1^1, q_2^1, \dots, q_{m_1}^1, \dots, q_1^n, q_2^n, \dots, q_{m_n}^n$ *($m_i \geq 1$) takové, že 1/* $q_1^0 \in I$, *2/* $q_{m_n}^n \in F$, *3/* $\delta(q_{m_{i-1}}^{i-1}, a_i) \ni q_1^i$ *pro vš.* $i = 1, 2, \dots, n$, *4/* $\delta(q_{j-1}^i, \varepsilon) \ni q_j^i$ *pro vš.* $i = 0, 1, 2, \dots, n$, $j = 2, 3, \dots, m_i$.

	a	b	c	ε
$\rightarrow$1	2	-	-	3
2	1	-	-	-
3	-	4	-	5
4	-	3	-	-
$\leftarrow$5	-	-	6	-
6	-	-	5	-

Obrázek 4.3:

Jazykem rozpoznávaným (přijímaným) automatem A *rozumíme jazyk* $L(A) = \{w \mid$ *slovo* w *je přijímáno* $A\}$.

Uvedená definice přijímání slova nevypadá zrovna elegantně, že ? Mělo by být nicméně jasné, co vyjadřuje.

Úkol 25 *ZNKA z obrázku 4.3 zadejte grafem.*
Naformulujte pojem přijímání slova automatem ZNKA v řeči grafů automatů.
Charakterizujte jazyk, který je daným automatem přijímán.

Úkol 26 *Upravte definici 2.0.5 pro případ ZNKA !*

Poznámka. Opět tedy máme příklad toho, že jeden a tentýž pojem lze formalizovat více způsoby (které jsou více či méně vhodné).

Uvedeme ještě jeden způsob, hodící se k zobecnění věty 4.0.20:

Mějme dán ZNKA $A = (Q, \Sigma, \delta, I, F)$. Nejprve pro $K \subseteq Q$ definujme $E(K)$ jako množinu stavů dosažitelných z K jen pomocí ε-šipek.

Úkol 27 *Zkuste podat induktivní definici* $E(K)$ *a pak se teprve podívejte na dále uvedené řešení.*

$E(K)$ je nejmenší množina splňující 1/ $K \subseteq E(K)$, 2/ jestliže $q \in E(K)$ a $q' \in \delta(q, \varepsilon)$, potom $q' \in E(K)$.

Nyní můžeme přechodovou funkci ZNKA rozšířit na $\delta : \mathcal{P}(Q) \times \Sigma^* \longrightarrow \mathcal{P}(Q)$ takto:

1. $\delta(K, \varepsilon) = E(K)$,
2. $\delta(K, wa) = \bigcup_{q \in \delta(K,w)} E(\delta(q, a))$.

Podobně jako u NKA takto dosáhneme, že $\delta(I, w)$ je množina právě těch stavů, do kterých se ZNKA může dostat zpracováním (přečtením) slova w (a tedy $L(A) = \{w \in \Sigma^* \mid \delta(I, w) \cap F \neq \emptyset\}$), a analogicky (”knoflíkovou hrou”) jako větu 4.0.20 lze ukázat

Věta 4.0.22 *ZNKA rozpoznávají právě regulární jazyky.*

Všimněme si ještě, že pomocí NKA lze podat jiný, velmi přímočarý, důkaz věty 3.0.14. (Nápověda: definujte vhodně pojem (disjunktního) *sjednocení* (tj. “vedle sebe položení”) *dvou NKA*.) Použijeme-li navíc ε-šipek (tedy ZNKA), docílíme navíc snadno toho, aby výsledný NKA měl jediný počáteční stav. (Proč ?)

Poznámka. Pro průnik nám ani elegance ε-šipek moc nepomůže. Ovšem uzavřenost třídy regulárních jazyků vůči průniku plyne také např. z uzavřenosti vůči sjednocení a doplňku, o které se zmíníme za chvíli (de Morganova pravidla).

Úkol 28 *Zkonstruujte ZNKA rozpoznávající jazyk* $L = \{\, uv \mid uav \in L(A) \vee ubv \in L(A) \,\}$*, kde* A *je KA zadaný uvedenou tabulkou. (Slova jazyka* L *vzniknou ze slov jazyka* $L(A)$ *vypadnutím jednoho písmene.)*

	a	b
↔1	2	1
2	2	3
3	4	3
4	5	5
5	1	5

(Nápověda: ZNKA bude "obsahovat dvě kopie výchozího KA".)
Nakonec alespoň započněte konstrukci DKA pro jazyk L*.*

Úkol 29 *Navrhněte rámcově realizaci konečného automatu, který vždy pro zadané slovo v abecedě* $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ *zjistí, zda v něm nějaké podslovo délky tři má alespoň dva (nepřekrývající se) výskyty.*

Kapitola 5

Uzávěrové vlastnosti třídy regulárních jazyků.

Cíl kapitoly:
Prohloubení schopnosti modulárního návrhu složitějších konečných automatů. Pochopení algoritmů prokazujících uzavřenost třídy jazyků rozpoznatelných konečnými automaty vůči různým operacím.

Studijní cíle: Prohloubení schopnosti modulárního návrhu složitějších konečných automatů. Pochopení algoritmů prokazujících uzavřenost třídy jazyků rozpoznatelných konečnými automaty vůči různým operacím.

Uzavřenost třídy regulárních jazyků na zřetězení a iteraci je znázorněna na obr. 5.1 a 5.2, které by měly dostatečně naznačit myšlenku. Následuje formální důkaz.

Věta 5.0.23 *Jestliže jazyky $L_1, L_2 \subseteq \Sigma^*$ jsou regulární, pak také jazyk $L_1 \cdot L_2$ je regulární. Jestliže L je regulární, pak také L^* je regulární.*

Důkaz: Nechť $L_1 = L(A_1)$, $L_2 = L(A_2)$ pro konečné automaty $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$; můžeme předpokládat $Q_1 \cap Q_2 = \emptyset$.

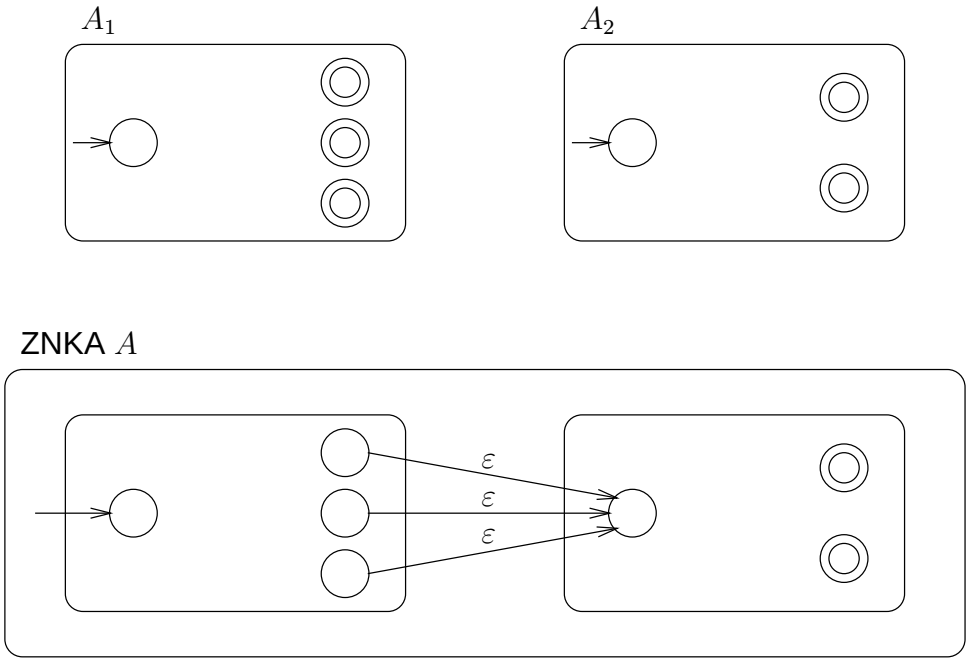
Definujme nyní ZNKA $A = (Q_1 \cup Q_2, \Sigma, \delta, \{q_{01}\}, F_2)$ tak, že $\delta(q,a) = \{\delta_1(q,a)\}$ je-li $q \in Q_1$ a $\delta(q,a) = \{\delta_2(q,a)\}$ je-li $q \in Q_2$; navíc pro každý stav $q \in F_1$ je $\delta(q,\varepsilon) = \{q_{02}\}$ a pro $q \notin F_1$ je $\delta(q,\varepsilon) = \emptyset$.

Je snadné ověřit, že $L(A) = L_1 \cdot L_2$. (Ověřte !)

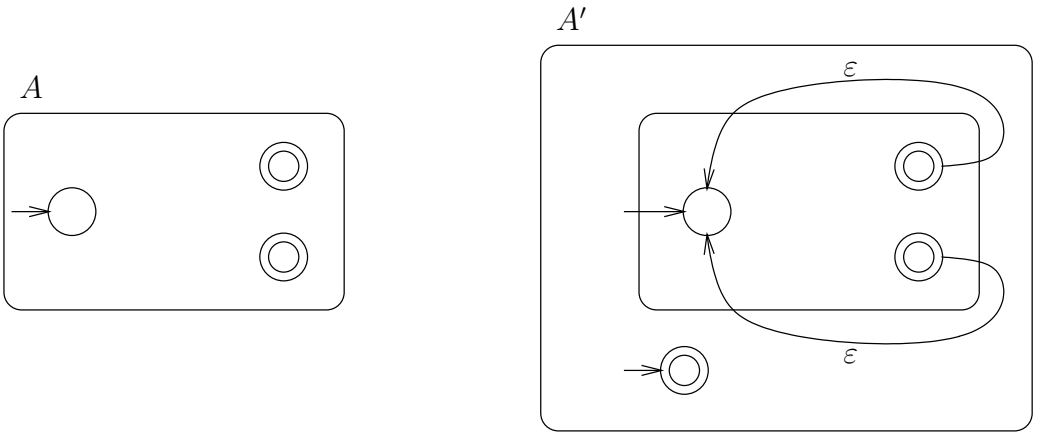
Nechť nyní $L = L(A)$ pro KA $A = (Q, \Sigma, \delta, q_0, F)$.

Definujme ZNKA $A' = (Q \cup \{p\}, \Sigma, \delta', \{q_0, p\}, F \cup \{p\})$, kde $p \notin Q$, $\delta'(q,a) = \{\delta(q,a)\}$ ($a \in \Sigma$) a pro $q \in F$ je $\delta'(q,\varepsilon) = \{q_0\}$; pro $q \notin F$ je $\delta'(q,\varepsilon) = \emptyset$ a navíc $\delta'(p,a) = \emptyset$ také pro vš. $a \in \Sigma$.

Je snadné ověřit, že $L(A') = L^*$. (Ověřte !) □



Obrázek 5.1: $L(A) = L(A_1) \cdot L(A_2)$



Obrázek 5.2: $L(A') = L(A)^*$

Úkol 30 *Uvědomili jste si roli přidaného (izolovaného) počátečního a přijímajícího stavu p v důkazu uzavřenosti na iteraci ?*
Uvažujme pro KA $A = (Q, \Sigma, \delta, q_0, F)$ konstrukci ZNKA $A' = (Q, \Sigma, \delta', \{q_0\}, F \cup \{q_0\})$, kde $\delta'(q, a) = \{\delta(q, a)\}$ ($a \in \Sigma$) a pro $q \in F$ je $\delta'(q, \varepsilon) = \{q_0\}$; pro $q \notin F$ je $\delta'(q, \varepsilon) = \emptyset$. Ukažte, že obecně neplatí $L(A') = L(A)^*$.

Vedle regulárních operací (sjednocení, zřetězení, iterace), je množina regulárních jazyků uzavřena i vůči dalším operacím. Uzavřenost vůči průniku jsme již ukázali dříve (věta 3.0.15), snadno ukážeme také uzavřenost na doplněk a vyvodíme uzavřenost na (množinový) rozdíl:

Věta 5.0.24 *Jestliže L je regulární, pak také $\overline{L}$ je regulární. Jestliže L_1, L_2 jsou regulární jazyky, pak také $L_1 - L_2$ je regulární.*

Důkaz: Nechť $L = L(A)$, kde A je KA $(Q, \Sigma, \delta, q_0, F)$. Pak $\overline{L}$ je zřejmě rozpoznáván KA $(Q, \Sigma, \delta, q_0, Q-F)$. (Přijímající a nepřijímající stavy byly prohozeny.)
Druhá část tvrzení pak již plyne z toho, že $L_1 - L_2 = L_1 \cap \overline{L_2}$. □

Úkol 31 *Uvažujme automaty A_1, A_2 zadané tabulkami:*

A_1

	a	b
$\rightarrow q_1$	q_2	q_3
$\leftarrow q_2$	q_2	q_4
$\leftarrow q_3$	q_5	q_3
q_4	q_2	q_4
q_5	q_5	q_3

A_2

	a	b
$\rightarrow r_1$	r_2	r_1
r_2	r_2	r_3
$\leftarrow r_3$	r_2	r_1

Zkonstruujte obecně použitelným algoritmem KA A rozpoznávající jazyk $L(A) = L(A_1) - L(A_2)$. Poté se snažte jazyk $L(A)$ co nejjednodušeji charakterizovat (podmínkou, kterou splňují slova do něj patřící).

Všimněme si ještě dalších jazykových operací (pojem kvocientu vyžaduje hlubší promyšlení !):

Definice 5.0.25 Zrcadlový obraz slova $u = a_1 a_2 \dots a_n$ *je* $u^R = a_n a_{n-1} \dots a_1$, zrcadlový obraz jazyka L *je* $L^R = \{u \mid \exists v \in L : u = v^R\}$.
Levý kvocient jazyka L_1 *podle jazyka* L_2 *je jazyk* $L_2 \backslash L_1 = \{\, u \mid \exists v \in L_2 : vu \in L_1 \,\}$.
Pravý kvocient jazyka L_1 *podle jazyka* L_2 *je jazyk* $L_1 / L_2 = \{\, u \mid \exists v \in L_2 : uv \in L_1 \,\}$.

Poznámka. Je užitečné si uvědomit, že kvocient typu $\{a\} \backslash L$ de facto implicitně používáme při konstrukci konečného automatu k danému jazyku. (Rozmyslete si proč.)

Úkol 32 *Rozmyslete si, proč obecně platí:*

- $(L^R)^R = L$
- $(L_1 \cdot L_2)^R = L_2^R \cdot L_1^R$
- $L / \emptyset = \emptyset$
- $L / \{\varepsilon\} = L$

- $L/(L_1 \cup L_2) = L/L_1 \cup L/L_2$
- $L_1/L_2 = (L_2^R \backslash L_1^R)^R$

Úkol 33 *Zjistěte, zda platí $L/(L_1 \cap L_2) = L/L_1 \cap L/L_2$.*
Dále zjistěte, zda operace "/" je asociativní.

Tvrzení 5.0.26 *L je regulární právě když L^R je regulární.*

Důkaz: Idea: (u NKA) zaměníme počáteční stavy s přijímajícími a obrátíme šipky.
Formálně:
Nechť $L = L(A)$ pro NKA $A = (Q, \Sigma, \delta, I, F)$.
Definujme NKA $A' = (Q, \Sigma, \delta', F, I)$ tak, že pro vš. $q_1, q_2 \in Q$, $a \in \Sigma$: $q_2 \in \delta'(q_1, a) \Leftrightarrow q_1 \in \delta(q_2, a)$.
Pak lze snadno ukázat, že $L(A') = L^R$. □

Zkuste dále sestavit důkaz uzavřenosti třídy regulárních jazyků vůči kvocientům:

Tvrzení 5.0.27 *Jestliže L_1, L_2 jsou regulární, pak také $L_2 \backslash L_1$ a L_1/L_2 jsou regulární.*

Návod:
Nechť $L_1 = L(A_1)$, kde $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, a $L_2 = L(A_2)$, kde $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$. Pro $q \in Q_1$ označme B_q automat $B_q = (Q_1, \Sigma, \delta_1, q, F_1)$ a C_q automat $C_q = (Q_1, \Sigma, \delta_1, q_{01}, \{q\})$. Definujme dále $U = \{\, q \in Q_1 \mid \exists w \in \Sigma^* : w \in L(A_2) \wedge \delta_1(q_{01}, w) = q \,\}$; jinak řečeno: $q \in U \iff L(A_2) \cap L(C_q) \neq \emptyset$. (Zdůvodněte, proč existuje algoritmus rozhodující příslušnost k množině U.) Ukažte nyní, že $L_2 \backslash L_1 = \bigcup_{q \in U} L(B_q)$.

Úkol 34 *Uvažujme automaty A_1, A_2 zadané tabulkami:*

A_1

	a	b
$\rightarrow q_1$	q_2	q_3
$\leftarrow q_2$	q_2	q_4
$\leftarrow q_3$	q_5	q_3
q_4	q_2	q_4
q_5	q_5	q_3

A_2

	a	b
$\rightarrow r_1$	r_2	r_1
r_2	r_2	r_3
$\leftarrow r_3$	r_2	r_1

Zkonstruujte obecně použitelným algoritmem KA A rozpoznávající jazyk $L(A) = L(A_1)/L(A_2)$ (pravý kvocient). Poté se snažte jazyk $L(A)$ co nejjednodušeji charakterizovat (podmínkou, kterou splňují slova do něj patřící).

Kapitola 6

Regulární výrazy a jejich ekvivalence s konečnými automaty.

Cíl kapitoly:
Zvládnutí popisu regulárních jazyků pomocí regulárních výrazů. Pochopení algoritmů (oboustranného) převodu mezi regulárními výrazy a konečnými automaty.

Studijní cíle: Zvládnutí popisu regulárních jazyků pomocí regulárních výrazů. Pochopení algoritmů (oboustranného) převodu mezi regulárními výrazy a konečnými automaty.

Již dříve jsme zmínili, že konečný automat "stojí v pozadí" mj. při vyhledávání vzorku v textu (např. při vyhledávání webovských stránek, které daný vzorek obsahují v záhlaví). Máme teď na mysli obecnější případ než je např. hledání konkrétního slova, a sice případ, kdy vzorek je specifikován (booleovskou) kombinací jednoduchých podmínek. Např. si lze představit, že výrazem "(česk$*$ & sloven$*$) $\vee$ (česk$*$ & němec$*$)" zadáváme (v nějakém systému) přání nalézt všechny dokumenty, které zároveň obsahují slovo začínající na "česk" a slovo začínající na "sloven" nebo zároveň obsahují slovo začínající na "česk" a slovo začínající na "němec".

Na výrazy podobné uvedenému lze pohlížet jako na popis (reprezentaci) určitého jazyka – reprezentovány jsou ty posloupnosti písmen (v "reálu" např. dokumenty, v našich pojmech jim říkáme prostě slova), které danému výrazu vyhovují; všimněte si, že takto reprezentovaný jazyk je pak obvykle nekonečný.

Teď si uvedeme definici tzv. regulárních výrazů a způsobu, jakým reprezentují jazyky (jak jste uhodli podle názvu, ukáže se, že regulárními výrazy lze reprezentovat právě regulární jazyky). Poznamenejme, že v různých softwarových systémech se setkáme s různými modifikacemi, nazvanými třeba také "regulární výrazy". V našem kursu (tak jako obecně v teorii jazyků a automatů) myslíme regulárními výrazy pouze pojem vymezený následující definicí.

Definice 6.0.28 *Množinu* $RV(\Sigma)$ ***regulárních výrazů*** *nad abecedou* Σ *definujeme jako nejmenší množinu slov v abecedě* $\Sigma \cup \{\emptyset, \varepsilon, +, \cdot, ^*, (,)\}$ *(předpokládáme* $\emptyset, \varepsilon, +, \cdot, ^*, (,) \notin \Sigma$*) splňující tyto podmínky:*

- $\emptyset \in RV(\Sigma)$, $\varepsilon \in RV(\Sigma)$, *pro každé* $a \in \Sigma$ *je* $a \in RV(\Sigma)$,
- *jestliže* $\alpha, \beta \in RV(\Sigma)$, *pak také* $(\alpha + \beta) \in RV(\Sigma), (\alpha \cdot \beta) \in RV(\Sigma), (\alpha^*) \in RV(\Sigma)$.

Definice 6.0.29 *Regulární výraz* α *reprezentuje jazyk – který označujeme* $[\alpha]$ *– tímto způsobem:* $[\emptyset] = \emptyset$, $[\varepsilon] = \{\varepsilon\}$, $[a] = \{a\}$ *a dále* $[(\alpha + \beta)] = [\alpha] \cup [\beta]$, $[(\alpha \cdot \beta)] = [\alpha] \cdot [\beta]$, $[(\alpha^*)] = [\alpha]^*$.

Poznámka. Při zápisu regulárních výrazů obvykle vynecháváme zbytečné závorky (asociativita operací, vnější pár závorek) a tečky pro zřetězení; další závorky lze vynechat díky dohodnuté prioritě operací: * váže silněji než $\cdot$, která váže silněji než $+$. Např. místo $((((0{\cdot}1)^*{\cdot}1){\cdot}(1{\cdot}1)){+}((0{\cdot}0){+}1)^*)$ obvykle napíšeme $(01)^*111{+}(00{+}1)^*$.

Úkol 35 *Zadejte regulárním výrazem jazyk*
$L = \{ w \in \{0,1\}^* \mid$ *ve* w *je sudý počet nul a každá jednička je bezprostředně následována nulou* $\}$

Úkol 36 *Zjistěte, zda platí*

$[(011 + (10)^*1 + 0)^*] = [011(011 + (10)^*1 + 0)^*]$

$[((1 + 0)^*100(1 + 0)^*)^*] = [((1 + 0)100(1 + 0)^*100)^*]$

Úkol 37 *Procvičujte si regulární výrazy tím, že jimi popíšete některé regulární jazyky, s nimiž se v našem textu (včetně úkolů) setkáváte.*

Již jsme avizovali, že vzhledem k popisu jazyků jsou regulární výrazy stejně silný prostředek jako konečné automaty. Jeden směr je zřejmý:

Pro jazyky $\emptyset$, $\{\varepsilon\}$, $\{a\}$ lze triviálně zkonstruovat rozpoznávající KA. Na základě příslušných konstrukcí v důkazech uzavřenosti třídy regulárních jazyků na regulární operace pak snadno sestavíme algoritmus, který k libovolnému regulárnímu výrazu α sestrojí (zobecněný) nedeterministický konečný automat A rozpoznávající jazyk $[\alpha]$. Všimněte si, že počet stavů A bude přitom v zásadě odpovídat délce α.

Úkol 38 *Myšlenky algoritmu převodu RV* $\rightarrow$ *ZNKA jsou načrtnuty na obrázku 6.1 – algoritmus k danému regulárnímu výrazu sestrojí ZNKA s jediným počátečním a jediným přijímajícím stavem.*
Aplikujte tento algoritmus na regulární výraz $((01^*0 + 101)^*100 + (11)^*0)^*01$.

Věta 6.0.30 *Regulárními výrazy lze reprezentovat právě regulární jazyky.*

Důkaz: Jelikož pro jazyky $\emptyset$, $\{\varepsilon\}$, $\{a\}$ lze triviálně zkonstruovat rozpoznávající KA, a třída regulárních jazyků je uzavřena na regulární operace (sjednocení, zřetězení, iterace), je zřejmý fakt, že ke každému regulárnímu výrazu α existuje (lze zkonstruovat) KA A tž. $L(A) = [\alpha]$ (tím jsme se zabývali v úkolu 38).

Technicky složitější je důkaz opačného směru, že totiž ke každému KA A existuje (opět lze zkonstruovat) regulární výraz α tž. $[\alpha] = L(A)$.

(Velmi) hrubá idea:

Slovo w je přijímáno automatem A právě když v grafu automatu existuje cesta (ohodnocená) w začínající v počátečním stavu q_0 a končící v "prvním" přijímajícím stavu nebo v "druhém" přijímajícím stavu atd. – v onom "nebo" lze snadno rozpoznat sjednocení jazyků.

Když cesta w vede z q_0 do (pevně zvoleného přijímajícího) q_A, tak buď je to "přímá cesta" – na níž se žádný stav neopakuje – nebo vznikne z přímé cesty "vložením cyklů". Přímých cest z q_0 do q_A je samozřejmě konečně mnoho (každá je nutně kratší než je počet stavů automatu), rozmístění cyklů je také konečně mnoho, a cykly lze iterovat. Stačí tedy "regulárně" popsat cykly a budeme hotovi. Elementárních cyklů (těch, které neobsahují kratší cyklus) je sice konečně mnoho, ale lze je různě kombinovat; to způsobuje, že ono regulární popsání vůbec není nabíledni a je velmi žádoucí podat přesvědčivý důkaz. Vhodný je např. induktivní důkaz, jenž je stručně načrtnut níže.

Induktivní důkaz:

Nechť $L = L(A)$ pro KA $A = (Q, \Sigma, \delta, q_1, F)$, kde $Q = \{q_1, q_2, \ldots, q_n\}$. Pro vš. $i, j \in \{1, 2, \ldots, n\}$ definujme $R_{ij} = \{w \in \Sigma^* \mid \delta(q_i, w) = q_j\}$ (tj. jako množinu slov, které převedou A ze stavu q_i do stavu q_j).

Dokážeme-li, že každá množina R_{ij} ($i, j \in \{1, 2, \ldots, n\}$) je reprezentovatelná regulárním výrazem, jsme hotovi – je totiž $L = \bigcup_{q_i \in F} R_{1i}$.

Zvolme nyní pevně i, j a uvažujme množinu R_{ij}. Pro $k \in \{0, 1, 2, \ldots, n\}$ definujme R^k_{ij} jako množinu slov, které převedou A ze stavu q_i do stavu q_j, přičemž všechny průběžné stavy mají index nejvýše rovný k. ($R^k_{ij} = \{w \in R_{ij} \mid \forall u, v : (u \neq \varepsilon \wedge v \neq \varepsilon \wedge w = uv \wedge \delta(q_i, u) = q_m) \Longrightarrow m \leq k\}$.)

Ukážeme-li, že pro každé k je množina R^k_{ij} reprezentovatelná regulárním výrazem, budeme hotovi – je totiž $R_{ij} = R^n_{ij}$. To ovšem lze ukázat indukcí podle k. Základem indukce je triviální fakt

$R^0_{ij} \subseteq \Sigma \cup \{\varepsilon\}$. Indukční krok je zřejmý ze vztahu

$$R^{k+1}_{ij} = R^k_{ij} \ \cup \ (\ R^k_{i,k+1}\ (R^k_{k+1,k+1})^*\ R^k_{k+1,j}\ )$$

□

Úkol 39 *Sestrojte regulární výraz reprezentující jazyk rozpoznávaný automatem zadaným uvedenou tabulkou.*

	a	*b*
→*1*	*2*	*1*
2	*2*	*3*
←*3*	*2*	*1*

Aplikujte přitom postup z předchozího důkazu, při němž se postupně konstruují výrazy reprezentující množiny $R^k_{i,j}$.

Dříve jsme rovnou zavedli pojem *regulární jazyk* jako synonymum pro *jazyk rozpoznatelný konečným automatem*. Pro úplnost dodejme, že v literatuře lze najít následující definici regulárních jazyků; ve světle výsledků uvedených výše je zřejmé, že obsah obou definic je totožný.

Třída $RJ(\Sigma)$ regulárních jazyků nad abecedou Σ je nejmenší třída jazyků nad abecedou Σ, která obsahuje tzv. *elementární jazyky* a je uzavřena na regulární operace, tzn.:

- elementární jazyky, tj. $\emptyset$ a $\{a\}$ (pro každé $a \in \Sigma$), patří do $RJ(\Sigma)$,
- jestliže $L_1, L_2 \in RJ(\Sigma)$, pak také $L_1 \cup L_2 \in RJ(\Sigma)$,

- jestliže $L_1, L_2 \in RJ(\Sigma)$, pak také $L_1 \cdot L_2 \in RJ(\Sigma)$,
- jestliže $L \in RJ(\Sigma)$, pak také $L^* \in RJ(\Sigma)$.

Poznámka. Jak plyne z uzávěrových vlastností třídy regulárních jazyků, mohli bychom regulární výrazy obohatit např. symboly pro průnik a doplněk (třeba $\&$, $\neg$, přičemž $[(\alpha\&\beta)] = [\alpha] \cap [\beta]$, $[\neg(\alpha)] = \Sigma^* - [\alpha]$ – abeceda Σ musí být zřejmá z kontextu), aniž se zvětší jejich vyjadřovací síla. Zápis jazyka se tak někdy zkrátí (např. místo $(0+1)^*1(0+1)^*$ lze psát $\neg(0^*)$; ztrácí se ale např. vlastnost přímočarého převodu RV $\rightarrow$ ZNKA zmíněného výše. To je jeden z důvodů proč průnik ani doplněk neřadíme k (standardním) regulárním operacím.

Poznámka. Důkazy matematickou indukcí (např. u věty 3.0.14 a dalších) jsme dosud v textu podrobně neprováděli. Příslušná tvrzení byla očividná a důkaz indukcí je u nich v podstatě jen rutinní cvičení (ovšem velmi užitečné !). Induktivní důkaz jsme skutečně provedli až nyní u věty 6.0.30, protože zde se bez něj opravdu těžko lze obejít (jak byste měli potvrdit, pokud jste se poctivě snažili větu "nahlédnout" a dokonale se přesvědčit o její platnosti).

Regulární výrazy nám poskytují další možnost reprezentace regulárních jazyků. Toho lze mj. také využít k elegantním důkazům některých dalších uzávěrových vlastností třídy regulárních jazyků; zde to ilustrujeme na příkladu tzv. regulární substituce.

Definice 6.0.31 *Nechť Σ je abeceda a pro každé $a \in \Sigma$ je dán jazyk $\sigma(a)$ v abecedě Δ_a. Položme $\sigma(\varepsilon) = \{\varepsilon\}$ a $\sigma(uv) = \sigma(u)\sigma(v)$ pro vš. $u, v \in \Sigma^*$. Potom zobrazení $\sigma : \Sigma^* \rightarrow \mathcal{P}(\Delta^*)$, kde $\Delta = \bigcup_{a\in\Sigma} \Delta_a$, se nazývá* substituce.

Pro každý jazyk $L \subseteq \Sigma^*$ pak definujeme $\sigma(L) = \bigcup_{w\in L} \sigma(w)$ a říkáme, že jazyk $\sigma(L)$ vznikl z jazyka L substitucí σ.

Substituce σ, u níž pro každé $a \in \Sigma$ obsahuje jazyk $\sigma(a)$ právě jedno slovo, se nazývá homomorfismus. *Homomorfismus h lze pak tedy považovat za zobrazení typu $h : \Sigma^* \rightarrow \Delta^*$ (které splňuje $h(\varepsilon) = \varepsilon, h(uv) = h(u)h(v)$).*

Tvrzení 6.0.32 *Nechť Σ je abeceda a σ* regulární substituce, *tzn. $\sigma(a)$ je regulární jazyk pro každé $a \in \Sigma$. Potom pro lib. regulární jazyk $L \subseteq \Sigma^*$ platí, že $\sigma(L)$ je rovněž regulárním jazykem.*

Důkaz: Nechť reg. výraz α reprezentuje L a nechť α_a reprezentuje $\sigma(a)$, pro každé $a \in \Sigma$. Dá se snadno ověřit, že dosadíme-li do α za každý výskyt symbolu a reg. výraz α_a, dostaneme regulární výraz reprezentující $\sigma(L)$. □

Úkol 40 *Uvedené tři tabulky nedeterministických automatů zadávají po řadě regulární jazyky L_1, L_2, L_3.*

	a	b
→A	A,B,C	-
→B	-	A,B,C
←C	B	C

	0	1
↔D	E	F
←E	E,F	D
F	D	F

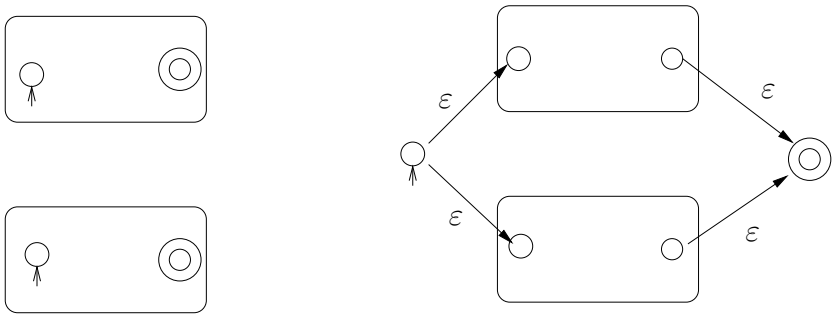
	0	1
→G	G	H
←H	H	G

Definujte regulární substituci σ předpisem $\sigma(a) = L_2$, $\sigma(b) = L_3$. Sestrojte regulární výraz popisující jazyk $\sigma(L_1)$.

Úkol 41 *K hlubšímu pochopení některých pojmů můžete zkusit zjistit, zda platí následující obecné vztahy;* h *označuje homomorfismus,* $\backslash$ *levý kvocient.*

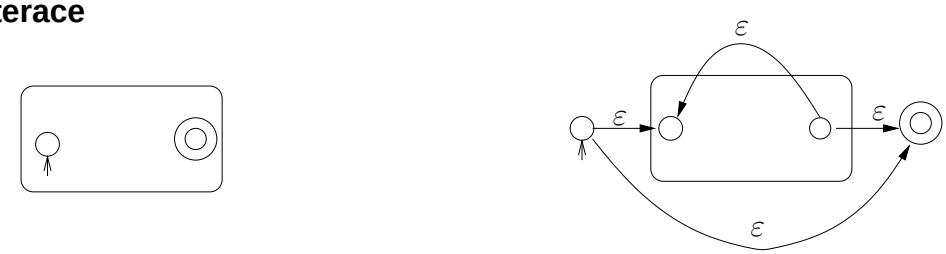
- $h(L_1 \cup L_2) = h(L_1) \cup h(L_2)$
- $h(L_1 \cap L_2) = h(L_1) \cap h(L_2)$
- $\{w\}\backslash(L_1 \cup L_2) = \{w\}\backslash L_1 \cup \{w\}\backslash L_2$
- $\{w\}\backslash(\Sigma^* - L) = \Sigma^* - \{w\}\backslash L$
- $L_2(L_2\backslash L_1) = L_1$

Sjednocení



Zřetězení

Iterace



Obrázek 6.1: Konstrukce ZNKA k regulárnímu výrazu

Kapitola 7

Minimální konečné automaty a algoritmus minimalizace.

Cíl kapitoly:
Zvládnutí algoritmu redukce konečného automatu a pochopení důkazu toho, že vede k tzv. minimálnímu automatu.

Studijní cíle: Zvládnutí algoritmu redukce konečného automatu a pochopení důkazu toho, že vede k tzv. minimálnímu automatu.

Vezmeme-li v úvahu praktické (implementační) hledisko, jistě není třeba sáhodlouze motivovat otázku možné minimalizace daného konečného automatu. Ukážeme, že odpověď je v našem případě velmi potěšující: existuje algoritmus (dokonce "rychlý" algoritmus), který k zadanému KA sestrojí ekvivalentní automat s nejmenším možným počtem stavů.

Definice 7.0.33 *Dva konečné* automaty A_1, A_2 *nazveme* jazykově ekvivalentní, *stručněji* ekvivalentní, *jestliže rozpoznávají týž jazyk, tj. jestliže* $L(A_1) = L(A_2)$.

Konečný automat nazveme ***minimálním automatem*** *jestliže neexistuje automat, který by s ním byl ekvivalentní a měl by menší počet stavů.*

Naším cílem nyní bude dokázat tuto větu:

Věta 7.0.34 *Existuje algoritmus, který k zadanému konečnému automatu* A *sestrojí minimální automat ekvivalentní s* A.

Úkol 42 *Velmi užitečné by bylo, kdybyste nejdříve sami zkusili (nějak) najít minimální automat ekvivalentní zde zadanému.*
V každém případě je vhodné si třeba na tomto příkladu ilustrovat dále zaváděné pojmy; konkrétně začněte konstrukcí rozkladů množiny stavů podle ekvivalencí $\overset{0}{\sim}, \overset{1}{\sim}, \overset{2}{\sim}, \ldots$, které jsou definovány až trochu dálee v textu.

	a	*b*
→*1*	*2*	*3*
2	*2*	*4*
←*3*	*3*	*5*
4	*2*	*7*
←*5*	*6*	*3*
←*6*	*6*	*6*
7	*7*	*4*
8	*2*	*3*
9	*9*	*4*

Ukážeme nejdříve hlavní ideu algoritmu. Uvažujme konečný automat $A = (Q, \Sigma, \delta, q_0, F)$, rovnou bez nedosažitelných stavů – ty bychom jinak prostě odstranili, aniž změníme rozpoznávaný jazyk (vzpomeňte si, že touto otázkou jsme se již zabývali dříve). Pro každý stav $q \in Q$ definujme

$$L(q) = L(A_q), \text{ kde } A_q = (Q, \Sigma, \delta, q, F)$$

($L(q)$ je tedy jazyk rozpoznávaný automatem, jenž vznikne z A prohlášením stavu q za počáteční, tedy množina těch slov, které převedou automat A ze stavu q do některého stavu v F).

Nechť nyní pro dva různé stavy q_1, q_2 platí $L(q_1) = L(q_2)$. Teď přijde ta idea: jeden z těchto stavů, např. q_2, můžeme z automatu A vypustit, přičemž šipky do něj směřující (v terminologii grafu automatu) přesměrujeme do q_1. Musíme dodat, že pokud q_2 byl počátečním (v automatu A), prohlásíme za nový počáteční stav q_1. Snadno se lze přesvědčit, že takto vzniklý automat je ekvivalentní s (původním) automatem A, přičemž má méně stavů – a nemá nedosažitelné stavy, když A je neměl. (Přesvědčte se !)

Vedle odstranění nedosažitelných stavů tak máme k dispozici další metodu, jak zmenšovat KA při zachování rozpoznávaného jazyka. Opakované použití této metody nás očividně přivede k automatu, který je ekvivalentní s původním a je to tzv. ***redukovaný automat***, což znamená, že nemá nedosažitelné stavy a pro každé jeho dva různé stavy q_1, q_2 platí $L(q_1) \neq L(q_2)$.

Tento výsledný automat lze elegantně popsat jako tzv. podílový automat podle ekvivalence $\sim$, kde relace $\sim$ je zavedena na množině stavů KA takto:

$$q \sim q' \Longleftrightarrow_{df} L(q) = L(q')$$

(je to samozřejmě jiná relace než ekvivalence mezi konečnými automaty zavedená výše).

Vsuvka. Připomeňme, že relace ρ na množině M je *ekvivalence* právě tehdy, když je reflexivní ($\forall x \in M : x \rho x$), symetrická ($\forall x, y \in M : x \rho y \Rightarrow y \rho x$) a tranzitivní ($\forall x, y, z \in M : (x \rho y \wedge y \rho z) \Longrightarrow x \rho z$). Ekvivalence ρ definuje na M *rozklad* $\{ [x]_\rho \mid x \in M \}$, tj. systém vzájemně disjunktních množin (neboli tříd ekvivalence), jejichž sjednocením je M, přičemž s každým $x \in M$ jsou v příslušné třídě obsaženy právě prvky s ním ekvivalentní ($[x]_\rho = \{y \mid x \rho y\}$).

Formální popis konstrukce podílového automatu:

Lemma 7.0.35 *K libovolnému konečnému automatu existuje ekvivalentní redukovaný automat.*

Důkaz: Mějme automat $A = (Q, \Sigma, \delta, q_0, F)$ bez nedosažitelných stavů. Značením $[q]$ ($q \in Q$) označujeme třídy ekvivalence $\sim$ obsahující q (tj. $[q] = \{p \mid p \sim q\}$).

Nyní k A definujme tzv. podílový automat podle ekvivalence $\sim$, označený $A_\sim$, takto: $A_\sim = (Q_\sim, \Sigma, \delta_\sim, [q_0], F_\sim)$, kde $Q_\sim = \{\ [q] \mid q \in Q\ \}$, $F_\sim = \{\ [q] \mid q \in F\ \}$ a $\delta_\sim([q], a) = [\delta(q, a)]$ (pro vš. $q \in Q$, $a \in \Sigma$).

Korektnost definice plyne z faktu $p \sim q \Rightarrow (p \in F \Leftrightarrow q \in F)$ a z faktu $p \sim q \Rightarrow \delta(p, a) \sim \delta(q, a)$.

Snadno ověříme, že $\delta(q, w) = q' \Leftrightarrow \delta_\sim([q], w) = [q']$ (např. provedením důkazu indukcí podle délky w), z čehož ihned vyvodíme ekvivalenci automatů A a $A_\sim$ (tj. $L(A) = L(A_\sim)$), i to, že $A_\sim$ nemá nedosažitelné stavy. □

K algoritmickému použití naší metody ovšem potřebujeme ukázat, že umíme pro libovolné stavy q_1, q_2 rozhodovat otázku, zda $L(q_1) = L(q_2)$. Pro tyto účely je vhodné definovat $L^{\leq i}(q)$ jako množinu všech slov z $L(q)$, které mají délku nejvýše i, a pak zavést na stavové množině automatu $A = (Q, \Sigma, \delta, q_0, F)$ relace $\overset{0}{\sim}, \overset{1}{\sim}, \overset{2}{\sim}, \ldots$ takto:

$$q_1 \overset{i}{\sim} q_2 \iff_{df} L^{\leq i}(q_1) = L^{\leq i}(q_2)$$

Jinak řečeno: pro stavy q_1, q_2 platí $q_1 \overset{i}{\sim} q_2$ právě když je nelze rozlišit žádným slovem délky nejvýše i (tj. pro každé slovo $w \in \Sigma^*$, $|w| \leq i$, je buď $\delta(q_1, w) \in F$, $\delta(q_2, w) \in F$ nebo $\delta(q_1, w) \notin F$, $\delta(q_2, w) \notin F$). Je očividné, že relace $\overset{i}{\sim}$ jsou ekvivalence a platí $\overset{0}{\sim} \supseteq \overset{1}{\sim} \supseteq \overset{2}{\sim} \supseteq \ldots$ (tedy $q_1 \overset{i+1}{\sim} q_2 \Longrightarrow q_1 \overset{i}{\sim} q_2$, neboli $\overset{i+1}{\sim}$ je zjemněním relace $\overset{i}{\sim}$).

Samozřejmě vidíme, že $q_1 \overset{0}{\sim} q_2$ právě když buď $q_1 \in F$, $q_2 \in F$ nebo $q_1 \notin F$, $q_2 \notin F$.

Dále je zřejmé, že dva stavy jsou rozlišitelné slovem délky nejvýše $i+1$ právě když jsou rozlišitelné slovem délky nejvýše i nebo existuje $a \in \Sigma$, které je převede do dvojice stavů rozlišitelných slovem délky nejvýše i; v obměněné podobě to můžeme formálně vyjádřit takto:

$$q_1 \overset{i+1}{\sim} q_2 \iff q_1 \overset{i}{\sim} q_2 \wedge (\forall a \in \Sigma :\ \delta(q_1, a) \overset{i}{\sim} \delta(q_2, a)) \tag{7.1}$$

Všimněme si teď, že ekvivalence $\overset{0}{\sim}$ rozloží množinu stavů Q na dvě třídy F, $Q - F$ (když jsou obě neprázdné). Také víme, že $\overset{i+1}{\sim}$ rozloží Q na stejně nebo více tříd než $\overset{i}{\sim}$, a ze vztahu (7.1) je zřejmé, že pokud $\overset{i}{\sim} = \overset{i+1}{\sim}$, pak $\overset{i}{\sim} = \overset{i+1}{\sim} = \overset{i+2}{\sim} = \ldots = \sim$.

Jelikož při počtu stavů n ($|Q| = n$) nemůže existovat rozklad na více než n tříd, je dokonce zřejmé, že rovnost $\overset{i}{\sim} = \overset{i+1}{\sim}$ musí určitě nastat pro nějaké $i \leq n-2$. Stačí tedy postupně konstruovat relace, resp. rozklady podle relací, $\overset{0}{\sim}, \overset{1}{\sim}, \overset{2}{\sim}, \ldots$, až zjistíme $\overset{i}{\sim} = \overset{i+1}{\sim}$ – víme, že pak $\overset{i}{\sim} = \sim$, a můžeme tak pro libovolné stavy q_1, q_2 rozhodovat, zda $L(q_1) = L(q_2)$ (zjištěním, zda jsou ve stejné třídě zkonstruovaného rozkladu podle $\sim$).

Máme tedy algoritmický postup, jak k danému automatu A zkonstruovat ekvivalentní automat B, který je redukovaný (lemma 7.0.35 tedy platí konstruktivně) – přitom když A je redukovaný, pak B je totožný s A, v opačném případě má B méně stavů než A. Je ale B ten nejmenší možný automat mezi automaty ekvivalentními s A ? Kladnou odpověď jednoduše vyvodíme z následujícího jednoduchého lemmatu (na něj se de facto odkazujeme i v důkazu 7.0.35).

Lemma 7.0.36 *Mějme dva KA $A = (Q, \Sigma, \delta, q_0, F)$ a $A' = (Q', \Sigma, \delta', q_0', F')$ Jestliže pro stav q automatu A a stav q' automatu A' platí $L(q) = L(q')$, pak pro každé $a \in \Sigma$ je $L(\delta(q, a)) = L(\delta'(q', a))$.*

Důkaz: Ukážeme, že když $L(\delta(q,a)) \neq L(\delta'(q',a))$, pak $L(q) \neq L(q')$. Když např. $u \in L(\delta(q,a))$ a $u \notin L(\delta'(q',a))$, pak nutně $au \in L(q)$ a $au \notin L(q')$. □

Uvědomte si, že z toho snadno plyne, že dva *redukované* automaty $A = (Q, \Sigma, \delta, q_0, F)$ a $A' = (Q', \Sigma, \delta', q_0', F')$, které jsou ekvivalentní (tj. $L(q_0) = L(q_0')$), mají jednak stejný počet stavů a dokonce jsou vlastně totožné (lépe řečeno: jsou izomorfní, tj. jeden dostaneme z druhého pouhým přejmenováním stavů). Teď už je zřejmé, že

minimální automat znamená totéž co redukovaný automat.

Dokázali jsme tak vlastně nejen větu 7.0.34, ale i

Věta 7.0.37 *Minimální automat ekvivalentní s daným KA A je určen jednoznačně.*

Víme, že tím "jednoznačně" míníme "jednoznačně, až na pojmenování stavů". Libovůli v onom pojmenování ovšem lze odstranit požadavkem převodu do normovaného tvaru, který jsme již definovali dříve.

Úkol 43 *Zjistěte všechny dvojice stavů q, q' u následujících dvou automatů (tedy $q, q' \in \{0, 1, 2, \ldots, 9\}$), pro něž $L(q) = L(q')$.*

	a	b
→0	0	1
←1	1	2
←2	3	1
3	2	4
4	2	3

	a	b
→5	5	6
6	7	5
←7	7	9
8	9	8
←9	8	7

Úkol 44 *Sestrojte minimální (deterministický) konečný automat, který rozpoznává tentýž jazyk jako NKA zadaný následující tabulkou (a převeďte ho do normovaného tvaru):*

	a	b
$\rightarrow q_0$	q_1, q_3	q_5
q_1	-	q_2
q_2	q_F	-
q_3	-	q_4
q_4	-	q_F
q_5	-	q_6
q_6	-	q_N
$\leftarrow q_F$	q_F	q_F
q_N	-	-

Úkol 45 *Sestrojte minimální (deterministické) konečné automaty, rozpoznávající jazyky reprezentované následujícími regulárními výrazy:*

- $(ab^*b + ab^*ab^*b + ab^*ab^*a)^*$
- $(a+bb)^* + (\,(b+c)^* \cdot (d+e)^*\,)^+$ *(kde pro jazyk L definujeme $L^+ = L + L^2 + L^3 + \ldots$ a pro reg. výraz α definujeme $[\alpha^+] = [\alpha]^+$)*

Všimněme si, že nyní (nejméně) dvěma různými způsoby umíme dokázat tuto důležitou větu:

Věta 7.0.38 *Existuje algoritmus, který pro lib. zadané KA A_1, A_2 rozhodne, zda $L(A_1) = L(A_2)$.*

Důkaz: Stačí k oběma KA sestrojit redukované automaty v normovaném tvaru a ty porovnat. Jiný důkaz plyne z partie o (konstruktivních) uzávěrových vlastnostech třídy regulárních jazyků, uvědomíme-li si, že $L(A_1) = L(A_2) \iff (L(A_1) - L(A_2)) \cup (L(A_2) - L(A_1)) = \emptyset$ a připomeneme-li si větu 2.0.8. □

Poznámka. Všimněte si, že jsme dokázali, že pokud dva stavy automatu, který má n stavů ($n \geq 2$), nelze rozlišit slovem délky nejvýše $n-2$, pak je nelze rozlišit vůbec (jestliže $L^{\leq n-2}(q) = L^{\leq n-2}(q')$, pak $L(q) = L(q')$). Tento fakt ukazuje, že pro rozhodování otázky, zda $L(q) = L(q')$, stačí probrat všechna slova do délky $n-2$ (jichž je samozřejmě konečně mnoho). Ovšem algoritmus založený na této myšlence by byl pro praktické použití velmi nevhodný. (Proč ?)

Kapitola 8

Neregulární jazyky. Pumping lemma pro regulární jazyky.

Cíl kapitoly:
Získání schopnosti v běžných případech poznat a prokázat vlastnosti, které činí konkrétní jazyk neregulárním.

Studijní cíle: Získání schopnosti v běžných případech poznat a prokázat vlastnosti, které činí konkrétní jazyk neregulárním.

Čtenáři by mělo být jasné, že ne každý jazyk je regulární (už jsme to dokonce dokázali při úvahách o mohutnostech množin: konečných automatů je spočetně mnoho, zatímco jazyků – už nad jednoprvkovou abecedou – je nespočetně mnoho).

Jak ale vypadá konkrétní neregulární jazyk ? Musí mít nějakou vlastnost, která neumožňuje rozpoznání libovolného jeho slova (tedy také libovolně dlouhého slova), máme-li pouze omezenou paměť a můžeme slovo pouze číst zleva doprava.

Uvažujme např. jazyk

$$L = \{a^j b^j \mid j \geq 0\}$$

(každé slovo tedy začíná úsekem a-ček, za nímž následuje stejně dlouhý úsek b-ček). Intuitivně je vidět, že při čtení slova zleva doprava nám "nezbývá nic jiného" než a-čka počítat a pak porovnat s počtem b-ček; k tomu nám ovšem předem omezená paměť nestačí, protože úsek a-ček může být libovolně dlouhý !

Ale pozor ! Tyto naše úvahy se nedají považovat za důkaz toho, že L je neregulární. Naše intuice nás může klamat, a třeba je to jen naší omezeností, že nevidíme způsob, jak se bez počítání a-ček můžeme obejít. Když by nám např. někdo tvrdil, že jazyk $\{\ a^m \mid m$ je dělitelné třemi $\}$ není regulární, protože nezbývá nic jiného než a-čka spočítat a výsledek dělit třemi, vyvrátili bychom mu to prostě předvedením konečného automatu, který tento jazyk rozpoznává – a tudíž se zde bez počítání a-ček lze obejít. Jak ale dokázat, že něco nelze ? Obvykle je klíčem vyvození logického sporu z předpokladu, že to lze.

U L bychom mohli postupovat takto: Předpokládejme, že L je rozpoznáván konečným automatem A; ten má nějaký (konečný) počet stavů, označme tento počet n. Automat A musí samozřejmě přijmout i slovo $a^n b^n$. Při čtení úseku a^n prochází postupně určitými stavy $q_0, q_1, q_2, \ldots, q_n$. Jelikož A má pouze n stavů, nutně se nějaký stav zopakuje, tedy $q_i = q_j$ pro nějaké i, j, kde $0 \leq i < j \leq n$. Pak ovšem slovo vzniklé zopakováním úseku mezi q_i a q_j, tedy slovo $a^i a^{j-i} a^{j-i} a^{n-j} b^n$, je automatem A přijímáno (proč ?); to ovšem nepatří do L a přivedli jsme tak ke sporu předpoklad, že A rozpoznává L. Všimněte si také, že A by musel rozpoznávat nejen slovo vzniklé jedním zopakováním příslušného úseku, ale také slova vzniklá libovolným "napumpováním" tohoto úseku, tedy slova tvaru $a^i a^{j-i} a^{j-i} \ldots a^{j-i} a^{n-j} b^n$; speciálním případem je pak vypuštění úseku, u nás slovo $a^i a^{n-j} b^n$.

Uvedené úvahy se snadno dají zobecnit. Velmi zhruba řečeno:

> v každém "dostatečně dlouhém" slově regulárního jazyka L existuje "krátké" neprázdné podslovo "blízko začátku", jehož vynecháním či "pumpováním" dostáváme vždy slova jazyka L

Formálně (a přesně) to vyjadřuje následující věta, již si čtenář teď už jistě sám snadno dokáže ("nápověda": za n, jehož existenci věta deklaruje, si pro konkrétnost můžete dosadit např. počet stavů minimálního automatu rozpoznávajícího L).

Věta 8.0.39 (Pumping lemma.) *Nechť L je regulární jazyk. Pak nutně existuje n tž. každé slovo $z \in L$, $|z| \geq n$, lze psát $z = uvw$, kde $|uv| \leq n$, $|v| \geq 1$ a pro vš. $i \geq 0$ je $uv^i w \in L$.*

Všimněte si, jak se střídají kvantifikátory: Je-li L regulární, pak (neboli $(\forall L$ tž. L je regulární $)$:)

$(\exists n)$ $(\forall z$ tž. $z \in L, |z| \geq n)$
$(\exists\, u, v, w$ tž. $z = uvw, |uv| \leq n, |v| \geq 1)$ $(\forall i \geq 0) : uv^i w \in L$

Poznámka. Více formálně bychom místo $(\forall x$ tž. $A)\, B$ psali $(\forall x : A \Rightarrow B)$ a místo $(\exists x$ tž $A)\, B$ bychom psali $(\exists x : A \wedge B)$.

Je užitečné představit si hru dvou hráčů A a B, kteří mají zadán (nějaký) jazyk L a hrají takto:

1. A zvolí $n \in \mathcal{N}$
2. B zvolí slovo z tž. $z \in L$ a $|z| \geq n$ (neexistuje-li takové slovo, A vyhrál)
3. A zvolí u, v, w tž. $z = uvw$, $|uv| \leq n$ a $|v| \geq 1$
4. B zvolí $i \geq 0$
5. *Výsledek:* je-li $uv^i w \in L$, pak vyhrál A, v případě $uv^i w \notin L$ vyhrál B.

Je zřejmé, že je-li L regulární, pak A má vítěznou strategii v uvedené hře. Jinak řečeno:

> *Má-li B vítěznou strategii, pak L není regulární.*

A právě navržení vítězné strategie hráče B je častým prostředkem k důkazu toho, že uvažovaný jazyk je neregulární.

Pro výše zkoumaný $L = \{a^j b^j \mid j \geq 0\}$ můžeme vítěznou strategii hráče B formulovat takto.

1. A zvolí (libovolné) $n \in \mathcal{N}$
2. B zvolí $z = a^n b^n$
3. A zvolí libovolné u, v, w tž. $z = uvw$, $|uv| \leq n$ a $|v| \geq 1$, tedy $u = a^j$, $v = a^k$ pro nějaké j, k tž. $j + k \leq n$, $k \geq 1$
4. B: zvolí $i = 0$ (lze také kterékoli $i \geq 2$)
5. Jelikož $a^j a^{n-(j+k)} b^n \notin L$, B vyhrává.

Poznámka. Všimněme si, že uvedená vítězná strategie hráče B pro jazyk $L_1 = \{a^j b^j \mid j \geq 0\}$ je rovněž vítěznou strategií pro jazyk $L_2 = \{\, w \in \{a, b\}^* \mid |w|_a = |w|_b \,\}$ (připomeňme, že $|w|_a$ označuje počet výskytů symbolu a ve slově w); konečný automat tedy nemůže rozpoznávat slova, v nichž jsou počty a-ček a b-ček stejné.
Z faktu, že L_1 není regulární, lze ovšem neregularitu jazyka L_2 elegantně prokázat také takto:
Kdyby L_2 byl regulární, byl by regulární i jazyk $L_2 \cap a^*b^*$, jelikož třída regulárních jazyků je uzavřena na průnik (a a^*b^* je očividně regulární). Ovšem je zřejmé, že $L_2 \cap a^*b^* = L_1$, a L_1 regulární není. Předpoklad, že L_2 je regulární je takto přiveden ke sporu, což znamená, že L_2 regulární není.

Poznámka. Správně bychom měli psát $[a^*b^*]$ místo a^*b^*, jelikož se odkazujeme k jazyku reprezentovanému daným regulárním výrazem. Protože ale nemůže dojít k nedorozumění, značení si takto zjednodušujeme.

Samozřejmě musíme ale být opatrní při posuzování (ne)regularity jazyka na základě jeho specifikace. Např. podobnost specifikace $L_3 = \{\, w \in \{a, b\}^* \mid$ počty podslov ab a ba ve w jsou stejné $\}$ s výše uvedenou specifikací jazyka L_2 může nabuzovat dojem, že L_3 je rovněž neregulární.

Úkol 46 *Zjistěte, zda jazyk L_3 je či není regulární. (Své zjištění dokažte.)*

Doplnění:

Čtenáře možná napadla otázka, zda pumping lemma přesně charakterizuje regulární jazyky, tj. zda pro jakýkoli neregulární jazyk má B vítěznou strategii. Není tomu tak, jak dokládá např.jazyk $L = \{a, b\}^* \cup \{c\}^+\{a^j b^j \mid j \geq 0\}$, který splňuje podmínku v pumping lemmatu (A má pro něj vítěznou strategii) a přitom není regulární. (Pro připomenutí: $\{c\}^+ = \{\, c, cc, ccc, \ldots \,\}$.)
To, že uvedený L není regulární, jsme samozřejmě schopni dokázat, použijeme-li i jiné prostředky. Z předpokladu, že L je regulární, můžeme např. využitím uzavřenosti třídy regulárních jazyků vůči kvocientům a průniku vyvodit, že i

$$(\,\{c\}^+ \backslash L\,) \cap \{a, b\}^* = \{\, a^j b^j \mid j \geq 0 \,\}$$

je regulární – o tom jsme už ovšem ukázali, že regulární není, a dospíváme takto ke sporu.

Úkol 47 *Dokažte, že následující jazyky nejsou regulární (využijte pumping lemma a rozmyslete si formulace důkazů ve formě hry dvou hráčů)*

- $L_1 = \{ 0^m 1^n 0^m \mid m, n \geq 0 \}$
- $L_2 = \{ ww \mid w \in \{0,1\}^* \}$
- $L_3 = \{ w(w)^R \mid w \in \{0,1\}^* \}$
- L_4 *je množina všech zápisů programů v Pascalu.*

Můžete zkusit též pro:

- $L_5 = \{ 0^p \mid p$ *je prvočíslo* $\}$
- $L_6 = \{ 0^n \mid n = k^2$ *pro nějaké* $k \geq 0 \}$

Úkol 48 *Uvažujte pumping lemma v tomto znění:*

> *Nechť* L *je regulární jazyk. Pak nutně existuje* n *tž. v každém slově* $z \in L$ *lze každé jeho podslovo* x *délky* n *(*$|x| = n$*) psát* $x = uvw$*, kde* $|v| \geq 1$*, přičemž pro* $z = y_1 x y_2 = y_1 uvw y_2$ *platí, že* $y_1 u v^i w y_2 \in L$ *pro vš.* $i \geq 0$*.*

Dokažte, že v tomto znění tvrzení také platí a vysvětlete, zda je obecnějším anebo speciálním případem dříve uvedené verze.

Kapitola 9

Další poznámky ke konečným automatům.

Cíl kapitoly:
Alespoň přehledově pochopit některé oblasti využití konečných automatů.

Studijní cíle: Alespoň přehledově pochopit některé oblasti využití konečných automatů.

Dvoucestné konečné automaty

Ukazovali jsme, že nemůžeme rozpoznávat právě slova tvaru $a^j b^j$, máme-li omezenou paměť a čteme-li slova zleva doprava. Ve skutečnosti by nám nepomohla ani možnost vracet se k již přečtenému úseku slova – pokud symboly nemůžeme nijak měnit, tj. nemůžeme si ve slově nic poznačit. Dá se ukázat, že odpovídající model, tzv. ***dvoucestné konečné automaty***, charakterizuje opět jen třídu regulárních jazyků, a to i v nedeterministické verzi; důkaz je ovšem technicky složitější.

Hledání vzorku v textu

Hledání všech výskytů vzorku v textu—tj. řetězce p (pattern) v (obvykle podstatně delším) řetězci t (text)—je velmi častá úloha každodenně probíhající v mnoha aplikacích. Proto je velmi podstatné, jakou časovou náročnost má použitý algoritmus. Naivní přístup, který nás asi napadne nejdříve (posouvající se "okénko" délky $|p|$ a kontrola, zda v okénku je p) vede k době úměrné $|p| \cdot |t|$. Podstatně lepší je tzv. "Knuth-Morris-Pratt algorithm", jehož doba běhu je úměrná $|p| + |t|$. Klíčovou myšlenkou tohoto algoritmu je tato:

Představme si přímočarou konstrukci nedeterministického konečného automatu, přijímajícího právě ta slova (v zadané abecedě), která mají sufix p. (Takový NKA by měl $|p| + 1$ stavů.) K tomuto NKA standardně zkonstruovaný DKA (konstruujeme jen dosažitelné stavy) má také jen $|p| + 1$ stavů. (Proč ?)

KMP-algoritmus pak obsahuje ještě další důležitou myšlenku (k použití onoho DKA není třeba konstruovat přechodovou funkci $\delta : Q \times \Sigma \to Q$, ale stačí použít jen jistou "failure" funkci $f : Q \to Q$), ale to v tomto textu dále nerozebíráme.

Úkol 49 *Sestrojte (deterministický) konečný automat, který je vhodný k použití pro vyhledávání řetězce* $ababaca$ *ve slovech (textech) v abecedě* $\{a, b, c\}$.

Konečně stavový překladač

Již dříve jsme zmínili, že konečný automat jako rozpoznávač jazyka je speciálním případem konečně stavového zařízení, realizujícího jistou vstupně-výstupní funkci.

Jedním z těchto obecnějších modelů je tzv. zobecněný sekvenční stroj (generalized sequential machine):

Zobecněný sekvenční stroj M je dán parametry $M = (Q, \Sigma, \Delta, q_0, \delta, \rho)$, kde Q je konečná neprázdná množina *stavů*, Σ je konečná neprázdná množina zvaná ***vstupní abeceda***, Δ je konečná neprázdná množina zvaná ***výstupní abeceda***, $q_0 \in Q$ je *počáteční (iniciální) stav*, $\delta : Q \times \Sigma \to Q$ je *přechodová funkce* a $\rho : Q \times \Sigma \to \Delta^*$ je *výstupní* funkce.

Takový stroj M jistým způsobem definuje zobrazení ("překlad") $f_M : \Sigma^* \to \Delta^*$. Zkuste odhadnout jakým. (K modelu konečného automatu si přidejte výstupní pásku s tím, že v každém kroku daném přechodovou funkcí δ se na výstup připíše řetězec daný funkcí ρ.) Pro jazyk L v abecedě Σ lze pak přirozeně definovat jeho obraz $f_M(L)$ (v abecedě Δ); podobně pro jazyk L v abecedě Δ lze definovat jeho vzor (inverzní obraz) $f_M^{-1}(L)$ (v abecedě Σ). Dá se pak např. ukázat, že f_M i f_M^{-1} zachovávají regulární jazyky (tj. když L je regulární, tak $f_M(L)$ i $f_M^{-1}(L)$ jsou regulární.

Další zobecnění dostaneme, povolíme-li (mj.) nedeterminismus. Příslušné zařízení se pak nazývá konečně-stavový překladač (převaděč; v angličtině "finite-state transducer"), který pak nedefinuje funkci, ale obecněji relaci (podmnožinu $\Sigma^* \times \Delta^*$).

Kapitola 10

Bezkontextové gramatiky a jazyky

Cíl kapitoly:
Seznámení se s pojmem bezkontextové gramatiky (neformálně i formálně), jakožto užitečného prostředku k (částečnému) popisu syntaxe programovacích jazyků. Zvládnutí návrhu elementárních bezkontextových gramatik.

Studijní cíle: Seznámení se s pojmem bezkontextové gramatiky (neformálně i formálně), jakožto užitečného prostředku k (částečnému) popisu syntaxe programovacích jazyků. Zvládnutí návrhu elementárních bezkontextových gramatik.

Uvažujme množinu všech aritmetických výrazů vytvořených z prvků abecedy $a, +, \times, (,)$ (číselné konstanty či proměnné teď nejsou podstatné, proto všechny reprezentujeme symbolem a). Příkladem takového výrazu je $a+a\times a$ nebo $(a+a)\times a$.

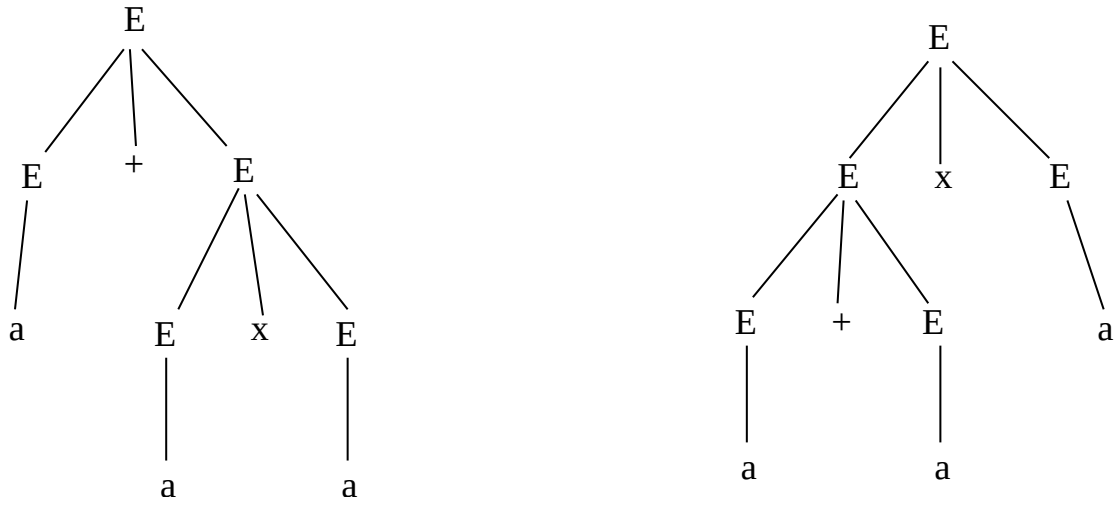
Čtenář jistě snadno dokáže, že se nejedná o regulární jazyk, nemůžeme jej tedy zadat regulárním výrazem či konečným automatem. V rámci v praxi užívaného popisu programovacích jazyků může být množina uvedených aritmetických výrazů zadaná těmito (přepisovacími) pravidly:

$$\langle EXPR\rangle \longrightarrow \langle EXPR\rangle + \langle EXPR\rangle$$
$$\langle EXPR\rangle \longrightarrow \langle EXPR\rangle \times \langle EXPR\rangle$$
$$\langle EXPR\rangle \longrightarrow (\langle EXPR\rangle)$$
$$\langle EXPR\rangle \longrightarrow a$$

Píšeme-li místo $\langle EXPR\rangle$ jen E a pravé strany pravidel se stejnou levou stranou soustředíme vedle sebe, přičemž je vzájemně oddělíme symbolem "|", vznikne:

$$E \longrightarrow E + E \mid E \times E \mid (E) \mid a \tag{10.1}$$

Jak vidno, v pravidlech vedle symbolů abecedy popisovaného jazyka, tak zvaných *terminálních symbolů* či stručněji *terminálů*, používáme i "proměnné" neboli *neterminály* (v našem případě E).



Možné *odvození*, neboli *derivace*, slova $a + a \times a$ pak může vypadat takto:

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E \times E \Rightarrow a + a \times E \Rightarrow a + a \times a$$

Uvedli jsme příklad tzv. *levé derivace*, kdy jsme v každém kroku přepisovali nejlevější neterminál (či přesněji řečeno nejlevější výskyt neterminálního symbolu). Uveďme příklad pravé derivace pro totéž slovo:

$$E \Rightarrow E + E \Rightarrow E + E \times E \Rightarrow E + E \times a \Rightarrow E + a \times a \Rightarrow a + a \times a$$

A ještě příklad derivace, která není ani levá ani pravá:

$$E \Rightarrow E + E \Rightarrow E + E \times E \Rightarrow E + a \times E \Rightarrow a + a \times E \Rightarrow a + a \times a$$

Je zřejmé, že se vlastně ve všech třech případech jedná o jedno a totéž odvození – jen pořadí přepisování neterminálů je různé. Strukturu odvození nezávislou na pořadí přepisování neterminálů zachycuje tzv. *strom odvození*, neboli *derivační strom*. V našem případě je derivační strom odpovídající všem třem derivacím znázorněn na obr. 10 vlevo.

Slovo $a + a \times a$ má ovšem i jinou *levou* derivaci než tu uvedenou výše, a sice:

$$E \Rightarrow E \times E \Rightarrow E + E \times E \Rightarrow a + E \times E \Rightarrow a + a \times E \Rightarrow a + a \times a$$

Této derivaci odpovídá *jiný* derivační strom – je zachycen na obr. 10 vpravo.

Existence dvou různých derivačních stromů (neboli dvou různých levých derivací) pro jedno slovo jazyka, je nežádoucí vlastnost – příslušná gramatika (tj. soubor přepisovacích pravidel) je *nejednoznačná* (o tomto problému se ještě zmíníme později).

Naši gramatiku (10.1) lze ovšem nahradit gramatikou

$$\begin{aligned}
&\langle EXPR\rangle \longrightarrow \langle EXPR\rangle + \langle TERM\rangle \mid \langle TERM\rangle \\
&\langle TERM\rangle \longrightarrow \langle TERM\rangle \times \langle FACTOR\rangle \mid \langle FACTOR\rangle \\
&\langle FACTOR\rangle \longrightarrow (\langle EXPR\rangle) \mid a
\end{aligned}$$

či stručněji

$$\begin{aligned}
&E \longrightarrow E + T \mid T \\
&T \longrightarrow T \times F \mid F \\
&F \longrightarrow (E) \mid a
\end{aligned}$$

která je s původní gramatikou ekvivalentní (tj. popisuje tentýž jazyk; umíte to dokázat ?) a je přitom jednoznačná. Např. naše slovo $a + a \times a$ má v ní jedinou levou derivaci (jediný derivační strom):

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T\times F \Rightarrow a+F\times F \Rightarrow a+a\times F \Rightarrow a+a\times a$

Naše příklady uvedly tzv. *bezkontextové gramatiky*. Tyto gramatiky reprezentují (generují) tzv. *bezkontextové jazyky*; jejich definici a způsob reprezentace jazyka nyní zformalizujeme.

Definice 10.0.40 ***Bezkontextová gramatika*** *je čtveřice (tj. je dána čtveřicí parametrů)* $G = (\Pi, \Sigma, S, P)$*, kde*

- Π *je konečná množina* neterminálních symbolů *(neterminálů)*
- Σ *je konečná množina* terminálních symbolů *(terminálů), přičemž* $\Pi \cap \Sigma = \emptyset$
- $S \in \Pi$ *je* počáteční *(startovací)* neterminál
- P *je konečná množina* pravidel *typu* $A \to \beta$*, kde*
  - A *je neterminál, tedy* $A \in \Pi$
  - β *je řetězec složený z terminálů a neterminálů, tedy* $\beta \in (\Pi \cup \Sigma)^*$*.*

Uvažujme lib. $\gamma, \delta \in (\Pi \cup \Sigma)^*$*. Řekneme, že* γ lze přímo přepsat na *(či* přímo odvodí*)* δ *(podle pravidel gramatiky* G*), značíme* $\gamma \Rightarrow_G \delta$ *nebo jen* $\gamma \Rightarrow \delta$ *když* G *zřejmá z kontextu, jestliže existují slova* μ_1, μ_2 *tž.* $\gamma = \mu_1 A \mu_2$*,* $\delta = \mu_1 \beta \mu_2$*, kde* $A \to \beta$ *je pravidlo v* P*.*

Řekneme, že γ lze přepsat na *(*odvodí*)* δ*, značíme* $\gamma \Rightarrow^* \delta$*, jestliže existuje posloupnost* $\mu_0, \mu_1, \ldots, \mu_n$ *slov z* $(\Pi \cup \Sigma)^*$ *(pro nějaké* $n \geq 0$*) tž.* $\gamma = \mu_0 \Rightarrow \mu_1 \Rightarrow \ldots \Rightarrow \mu_n = \delta$*. Zmíněnou posloupnost pak nazveme* odvozením *(*derivací*) délky* n *slova* δ *ze slova* γ*.*

Jazyk generovaný gramatikou G*, označme jej* $L(G)$*, je definován takto:* $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$*.*

Dvě gramatiky G_1, G_2 *nazveme* ekvivalentní*, jestliže* $L(G_1) = L(G_2)$*.*

Jazyk L *je* bezkontextový*, jestliže existuje bezkontextová gramatika* G *taková, že* $L(G) = L$*.*

Poznámka.

- Relace $\Rightarrow^*$ je reflexivním a tranzitivním uzávěrem relace $\Rightarrow$.
- $\gamma \Rightarrow^* \delta$ čteme také 'δ dostaneme z γ', 'γ generuje δ' apod.
- Výše zmíněné *odvození* $\mu_0 \Rightarrow \mu_1 \Rightarrow \ldots \Rightarrow \mu_n$ nazveme *minimální*, jestliže $\mu_i \neq \mu_j$ pro $i \neq j$. Je zřejmé, že jestliže $\gamma \Rightarrow^* \delta$, pak δ lze z γ odvodit (i nějakým) minimálním odvozením. V dalším budeme odvozením automaticky myslet minimální odvození.
- Není-li řečeno jinak, značíme jednotlivé terminály symboly $a, b, c, \ldots$, terminální slova $u, v, w, \ldots$, neterminály $A, B, C, \ldots, X, Y, Z$, řetezce neterminálů a terminálů $\alpha, \beta, \gamma, \ldots$.
- Uvedené příklady již ilustrovaly častý způsob zápisu bezkontextových gramatik, kdy udáváme kompaktně všechny pravé strany pravidel, jež mají týž neterminál na levé straně – tyto pravé strany pak vzájemně oddělujeme svislou čarou "|".

Definice 10.0.41 *Mějme bezkontextovou gramatiku $G = (\Pi, \Sigma, S, P)$; řekneme, že α lze přepsat na β* levým přepsáním*, jestliže v P ex. pravidlo $X \to \gamma$ tž. $\alpha = uX\delta$, $\beta = u\gamma\delta$ pro nějaké $u \in \Sigma^*$, $\delta \in (\Pi \cup \Sigma)^*$. Odvození $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$ je **levým odvozením** (**levou derivací**), jestliže pro vš. $i = 0, 1, \ldots, n-1$ lze α_i přepsat na α_{i+1} levým přepsáním. (**Pravé odvození** lze definovat obdobně.)*

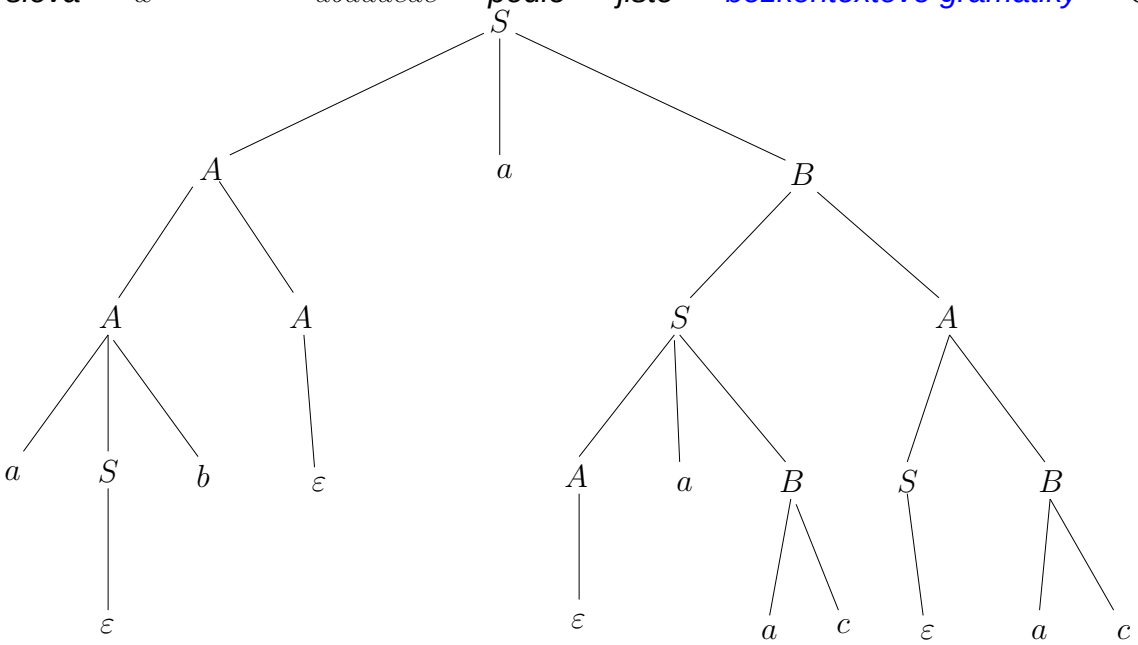
Poznámka. Lze snadno ukázat, že platí-li $X \Rightarrow_G^* w$, pak w lze z X odvodit (nějakým) levým odvozením i (nějakým) pravým odvozením.

Definice 10.0.42 ***Derivační strom***, *vztahující se k bezkontextové gramatice $G = (\Pi, \Sigma, S, P)$, je uspořádaný kořenový strom (tj. souvislý graf bez cyklů, s vyznačeným vrcholem-kořenem, následníci každého vrcholu jsou uspořádáni 'zleva doprava'), jehož vrcholy jsou ohodnoceny symboly z $\Pi \cup \Sigma$; přitom kořen je ohodnocen symbolem S a lib. vrchol s následníky je ohodnocen neterminálem $X \in \Pi$, přičemž tito následníci jsou ohodnoceni $Y_1, Y_2, \ldots, Y_n$ ($Y_i \in \Pi \cup \Sigma$), kde $X \to Y_1 Y_2 \ldots Y_n$ je pravidlo v P. V případě pravidla $X \to \varepsilon$ připouštíme následníka ohodnoceného ε. (Brzy uvidíme, že se bez těchto ε-pravidel můžeme obejít.)*

Jsou-li listy derivačního stromu zleva doprava ohodnoceny terminály $a_1, a_2, \ldots, a_n$, říkáme, že se jedná o derivační strom pro slovo $w = a_1 a_2 \ldots a_n$.

Poznámka. Všimněme si, že každému odvození slova w v gramatice G odpovídá (přirozeným způsobem) právě jeden derivační strom pro w; derivačnímu stromu pro w odpovídá obecně více odvození slova w, ovšem např. právě jedno levé odvození.

Úkol 50 *Na obrázku je derivační strom popisující odvození slova $w = abaaacac$ podle jisté bezkontextové gramatiky G.*



- *Napište levé odvození slova w podle gramatiky G.*
- *Napište pravé odvození slova w podle gramatiky G.*
- *Najděte rozklad $w = w_1 w_2 w_3$ s $w_2 \neq \varepsilon$ tak, aby slovo $w_1 w_2 w_2 w_3$ také patřilo do $L(G)$.*

Úkol 51 *Navrhněte bezkontextové gramatiky generující následující jazyky:*

- $L_1 = \{ w \in \{a, b\}^* \mid w$ *obsahuje podslovo* $baab \}$

- $L_2 = \{ w \in \{a, b\}^* \mid |w|_b \bmod 3 = 0 \}$
- $L_3 = \{ ww^R \mid w \in \{a, b\}^* \}$
- $L_4 = \{ 0^n 1^m 0^n \mid m, n \geq 0 \}$
- $L_5 = \{ 0^n 1^m \mid 1 \leq n \leq m \leq 2n \}$

Úkol 52 *Snažte se co nejvýstižněji charakterizovat jazyk generovaný gramatikou* $S \longrightarrow bSS \mid a$

Kapitola 11

Úloha syntaktické analýzy v překladačích

Cíl kapitoly:
Pochopení role syntaktické analýzy v překladačích.

Studijní cíle: Pochopení role syntaktické analýzy v překladačích.

Úlohou syntaktické analýzy v překladačích je rozpoznat, zda zadávaný program (tj. zadávaná posloupnost znaků) je skutečně programem, tj. zda je (syntakticky) správně utvořen. Nestačí ale jen odpověď Ano/Ne. V kladném případě je používaným výstupem např. derivační strom (resp. jeho vhodná reprezentace), jenž slouží jako vstup pro další fáze překladače.

Pro konkrétnější představu se podívejte na 'výsek' z jednoduchého překladače, který je uveden na konci této přednášky. Všimněte si, že derivační (pod)stromy na obr. 10 by vedly k sémanticky (tj. významově) různým cílovým programům !

Jednoznačné gramatiky a jazyky.

Uvedené úvahy mj. ilustrují, proč je důležitou vlastností gramatik tzv. jednoznačnost:

Definice 11.0.43 *Řekneme, že bezkont.* gramatika G *je* jednoznačná*, jestliže každé slovo z* $L(G)$ *má právě jedno levé odvození (tj. právě jeden derivační strom). V opačném případě je* G nejednoznačná *(či* víceznačná*).*

Úkol 53 *Lze v úkolu 50 z dostupné informace zjistit něco ohledně víceznačnosti příslušné gramatiky ?*

Viděli jsme, že např. víceznačnou gramatiku (10.1) je možné transformovat na ekvivalentní jednoznačnou gramatiku. Bohužel toto není možné vždy:

Definice 11.0.44 Bezkontextový jazyk L*, který lze generovat jednoznačnou gramatikou (tj.: ex. jednoznačná bezkontextová gramatika* G *tž.* $L(G) = L$*) se nazývá* jednoznačný*; v opačném případě se* L *nazývá (*vnitřně*)* nejednoznačný*.*

Např. jazyk $L_1 = \{ a^n b^n \mid n \geq 0 \}$ je generován jednoznačnou bezkontextovou gramatikou

$$S \longrightarrow aSb \mid \varepsilon$$

Jazyk $L_2 = \{ a^i b^j c^k \mid (i = j) \vee (j = k) \}$ generuje např. gramatika

$$S \longrightarrow S_1 C \mid A S_2$$
$$S_1 \longrightarrow a S_1 b \mid \varepsilon$$
$$C \longrightarrow c\, C \mid \varepsilon$$
$$S_2 \longrightarrow b S_2 c \mid \varepsilon$$
$$A \longrightarrow aA \mid \varepsilon$$

Tato gramatika jednoznačná není (proč ?) a dá se dokázat (ne zcela triviálně), že neexistuje jednoznačná bezkontextová gramatika generující L_2; L_2 je tedy nejednoznačný (bezkontextový) jazyk.

Ilustrace jednoduchého překladu

Omezme se na pascalské přiřazovací příkazy typu $V := E$, kde V je identifikátor proměnné typu real a E je aritmetický výraz vytvořený z identifikátorů proměnných a zápisů čísel typu real pomocí operátorů $+, *$ a pomocí závorek. Takovým příkazem je např.

$$Zisk := (Cena + Dan) * 0.12 \tag{11.1}$$

Všimněme si, že všechny takové příkazy lze generovat bezkontextovou gramatikou G ve tvaru

$$S \longrightarrow \langle id \rangle := E$$
$$E \longrightarrow E + E \mid E * E \mid (E) \mid \langle id \rangle$$

(kde $\{S, E\}$ je množina neterminálů, S počáteční neterminál a $\{:=, +, *, (,), \langle id \rangle\}$ množina terminálů) za předpokladu, že každý výskyt terminálu $\langle id \rangle$ bude nahrazen konkrétním identifikátorem proměnné nebo zápisem čísla typu real.

Chceme navrhnout algoritmus, který libovolný zmíněný pascalský příkaz přeloží do asembleru stroje s jediným pracovním registrem, zvaným akumulátor (ACC), s pamětí tvořenou posloupností (adresovaných) buněk a s následujícím instrukčním repertoárem:

Instrukce	Efekt
LOAD m	$c(m) \rightarrow ACC$
STORE m	$c(ACC) \rightarrow m$
ADD m	$c(ACC) + c(m) \rightarrow ACC$
MPY m	$c(ACC) * c(m) \rightarrow ACC$
LOAD $= m$	$m \rightarrow ACC$
ADD $= m$	$c(ACC) + m \rightarrow ACC$
MPY $= m$	$c(ACC) * m \rightarrow ACC$

K vysvětlení snad stačí následující poznámky:

- např. $c(m) \rightarrow ACC$ znamená, že obsah paměťové buňky m (tedy buňky s adresou m) se zkopíruje do akumulátoru
- výraz $= m$ znamená přímo numerickou hodnotu m
- předpokládáme, že ADD a MPY jsou "floating-point" operace

Práce překladače se dá rozdělit zhruba do následujících fází:

- ***lexikální analýza***, kdy se ve zpracovávaném zdrojovém textu (tj. ve vstupním řetězci) zjistí tzv. lexikální jednotky (např. identifikátory, zápisy čísel, znaky $+, *, :=$ apod.),
- ***syntaktická analýza***, kdy se zjistí syntaktická struktura řetězce předzpracovaného lexikální analýzou,
- ***generování kódu***, kdy se s využitím zjištěné syntaktické struktury vytváří cílový kód (tj. překlad vstupního řetězce).

(V reálném případě se tyto fáze různě prolínají, jsou doplněny o další fáze – např. o optimalizaci kódu, zotavení z chyb apod.; ale to není pro náš příklad podstatné).

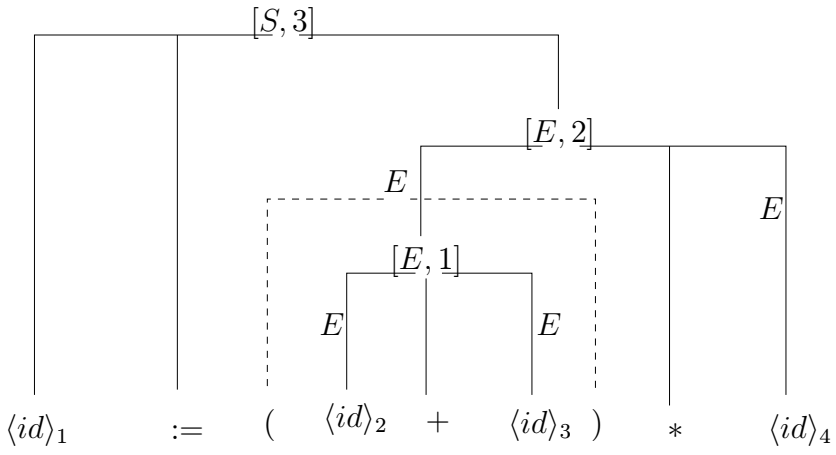
Výsledkem lexikální analýzy vstupního řetězce (11.1) by mohl být řetězec

$$\langle id \rangle_1 := (\langle id \rangle_2 + \langle id \rangle_3) * \langle id \rangle_4 \tag{11.2}$$

zároveň s tabulkou TAB:

Poř. číslo	Identifikátor	Informace
1	Zisk	prom. real
2	Cena	prom. real
3	Dan	prom. real
4	0.12	konst. real

Výsledkem syntaktické analýzy pro řetězec (11.2) by mohl být (derivační) strom na obrázku 11.1, ve kterém číslo v ohodnocení vnitřních uzlů udává maximální vzdálenost k listu.



Obrázek 11.1: Výstup syntaktické analýzy

(Derivační strom podle G by měl 7 vnitřních uzlů. Zde předpokládáme, že "zbytečné" uzly byly vyhozeny, takže výsledný strom má jen 3 vnitřní uzly; např. závorky jsou potřeba k správnému vytvoření stromu, ale pro další účely nejsou potřebné).

Výsledný kód lze ze sestrojeného stromu sestavit pomocí následujících pravidel, která každému vnitřnímu uzlu u přiřazují $Cod(u)$:

- uzel u je ohodnocen $\langle id \rangle_i$: jestliže i-tá položka v tabulce TAB je proměnná typu real, pak $Cod(u)$ je příslušný identifikátor (např. je-li u ohodnocen $\langle id \rangle_1$, je $Cod(u)$ řetězec 'Zisk'); jestliže i-tá položka v tabulce TAB je konstanta (typu real), pak $Cod(u)$ je příslušný zápis čísla předcházený znakem $=$ (např. je-li u ohodnocen $\langle id \rangle_4$, je $Cod(u)$ řetězec '$= 0.12$')
- je-li uzel u ohodnocen některým ze symbolů $:=, +, *$ pak $Cod(u)$ je prázdný řetězec
- uzel u je ohodnocen $[S, n]$ a má následníky u_1, u_2, u_3: Cod(u) je pak

  'LOAD' $Cod(u_3)$ '; STORE' $Cod(u_1)$
- uzel u je ohodnocen $[E, n]$ a má následníky u_1, u_2, u_3: pak
  - jestliže u_2 je ohodnocen $+$, $Cod(u)$ je

    $Cod(u_3)$ '; STORE $'$n$ '; LOAD' $Cod(u_1)$ '; ADD $'$n$
  - jestliže u_2 je ohodnocen $*$, $Cod(u)$ je

    $Cod(u_3)$ '; STORE $'$n$ '; LOAD' $Cod(u_1)$ '; MPY $'$n$

Např. kód příslušný k uzlu ohodnocenému $[E, 1]$ je

Dan ; STORE $1 ; LOAD $Cena$; ADD $1

Kód příslušný k uzlu ohodnocenému $[S, 3]$ (tedy kýžený program v asembleru) je

LOAD $= 0.12$;
STORE $2 ;
LOAD Dan ;
STORE $1 ;
LOAD $Cena$;
ADD $1 ;
MPY $2 ;
STORE $Zisk$

Kapitola 12

Speciální formy bezkontextových gramatik

Cíl kapitoly:
Zvládnutí algoritmů převádějících bezkontextové gramatiky do speciálních forem

Studijní cíle: Zvládnutí algoritmů převádějících bezkontextové gramatiky do speciálních forem

12.1 Redukované gramatiky

Vzpomeňme si na odstraňování nedosažitelných stavů u konečných automatů – takové stavy jsou "zbytečné". Teď se podíváme na odstraňování "zbytečných" neterminálů u bezkontextových gramatik. Neterminál je zbytečný, jestliže z něj nelze odvodit žádné terminální slovo (pak se tedy nemůže objevit v žádném odvození terminálního slova z počátečního neterminálu), nebo je nedosažitelný – nemůže se prostě vůbec objevit při jakémkoli přepisování začínajícím z počátečního neterminálu. Redukovaná gramatika neobsahuje takové zbytečné neterminály:

Definice 12.1.1 *Bezkontextová gramatika* $G = (\Pi, \Sigma, S, P)$ *se nazývá* ***redukovaná****, jestliže jsou pro každý* $X \in \Pi$ *splněny tyto dvě podmínky:*

1. *existuje aspoň jedno* $w \in \Sigma^*$ *tž.* $X \Rightarrow^* w$*,*
2. *existují slova* $\alpha, \beta \in (\Pi \cup \Sigma)^*$ *tž.* $S \Rightarrow^* \alpha X \beta$*.*

Uvažujme nejprve, jak pro danou gramatiku $G = (\Pi, \Sigma, S, P)$ zjistit neterminály splňující podmínku 1.

Chceme tedy zkonstruovat množinu $\mathcal{T} = \{X \in \Pi \mid \exists w \in \Sigma^* : X \Rightarrow^* w\}$. Konstruujeme postupně množiny $\mathcal{T}_1, \mathcal{T}_2, \ldots$, kde $\mathcal{T}_1 = \{X \in \Pi \mid \exists w \in \Sigma^* : (X \to w) \in P\}$ a $\mathcal{T}_{i+1} = \mathcal{T}_i \cup \{X \in \Pi \mid \exists \alpha \in \mathcal{T}_i^* : (X \to \alpha) \in P\}$, až k případu $\mathcal{T}_n = \mathcal{T}_{n+1}$. Na takový případ nutně narazíme pro $n \leq |\Pi|$ a očividně platí $\mathcal{T}_n = \mathcal{T}$.

Neterminály, slňující podmínku 2., tedy dosažitelné neterminály, lze zjistit takto:

Množinu $\mathcal{D} = \{X \in \Pi \mid \exists \alpha, \beta : S \Rightarrow_G^* \alpha X \beta\}$ sestrojíme zase postupnou konstrukcí $\mathcal{D}_1, \mathcal{D}_2, \ldots$, kde $\mathcal{D}_1 = \{S\}$ a $\mathcal{D}_{i+1} = \mathcal{D}_i \cup \{X \in \Pi \mid$ ex. $Y \in \mathcal{D}_i$ a α obsahující X tž. $(Y \to \alpha) \in P\}$, až k případu $\mathcal{D}_n = \mathcal{D}_{n+1}$.

Snadno teď ukážeme:

Věta 12.1.2 *Ke každé bezkontextové gramatice* G *tž.* $L(G) \neq \emptyset$ *lze sestrojit ekvivalentní redukovanou gramatiku.*

Důkaz: Mějme $G = (\Pi, \Sigma, S, P)$. Nejdříve zkonstruujeme množinu neterminálů splňujících podmínku 1. (z definice redukované gramatiky).

Pak v G vynecháme všechny neterminály nesplňující 1. a všechna pravidla, která takové neterminály obsahují. Dostaneme tak jistou gramatiku G' a je očividné, že $L(G) = L(G')$.

Pro gramatiku G' nyní zkonstruujeme množinu neterminálů splňujících podmínku 2. a dále vynecháme všechny neterminály nesplňující 2. a všechna pravidla, která takové neterminály obsahují. Dostaneme tak jistou gramatiku G'' a je opět očividné, že $L(G) = L(G') = L(G'')$.

Přesvědčte se, že G'' je skutečně redukovanou gramatikou. □

Úkol 54 *Zredukujte následující bezkontextové gramatiky:*

$S \longrightarrow aSb \,|\, aAbb \,|\, \varepsilon$
$A \longrightarrow aAB \,|\, bB$
$B \longrightarrow aAb \,|\, BB$
$C \longrightarrow CC \,|\, cS$

$S \longrightarrow aA \,|\, bB \,|\, aSa \,|\, bSb \,|\, \varepsilon$
$A \longrightarrow bCD \,|\, Dba$
$B \longrightarrow Bb \,|\, AC$
$C \longrightarrow aA \,|\, c$
$D \longrightarrow DE$
$E \longrightarrow \varepsilon$

Úkol 55 *Přehození uvedeného postupu (tj. nejprve odstranění neterminálů nesplňujících 2. a pak těch nesplňujících 1. nemusí vést k redukované gramatice.*

Snadno teď také můžeme ukázat tuto větu:

Věta 12.1.3 *Existuje algoritmus, který pro libovolnou bezkontextovou gramatiku G rozhodne, zda $L(G) = \emptyset$.*

Důkaz: Stačí ověřit, zda S patří do množiny neterminálů splňujících podmínku 1. □

Úkol 56 *Zjistěte, zda pro následující gramatiku G je $L(G) \neq \emptyset$*
$S \longrightarrow aS \,|\, AB \,|\, CD$
$A \longrightarrow aDb \,|\, AD \,|\, BC$
$B \longrightarrow bSb \,|\, BB$
$C \longrightarrow BA \,|\, ASb$
$D \longrightarrow ABCD \,|\, \varepsilon$

Poznámka. Dále poznamenejme, že na rozdíl od konečných automatů neexistuje algoritmus, který by k dané bezkontextové gramatice zkonstruoval nejmenší s ní ekvivalentní. Dá se to ukázat metodami teorie vyčíslitelnosti, z nichž rovněž plyne, že neexistuje algoritmus, který by rozhodoval ekvivalenci bezkontextových gramatik. (To bude demonstrováno v kursu o vyčíslitelnosti a složitosti.)

12.2 Nevypouštějící gramatiky

Již jsme dříve zmínili možnost zbavení se (z technických důvodů nepříjemných) tzv. ε-pravidel (pravidel typu $X \longrightarrow \varepsilon$):

Definice 12.2.1 *Bezkontextová gramatika se nazývá* ***nevypouštějící*** *, jestliže neobsahuje žádné pravidlo typu* $X \longrightarrow \varepsilon$.

Věta 12.2.2 *Ke každé bezkontextové gramatice* G *lze sestrojit ekvivalentní nevypouštějící gramatiku* G' *tž.* $L(G') = L(G) - \{\varepsilon\}$.

Důkaz: Konstrukce využívá obdoby výše uvedené konstrukce pro neterminály splňující podmínku 1. z definice redukované gramatiky.

Ke gramatice $G = (\Pi, \Sigma, S, P)$ totiž nejprve sestrojíme množinu $\mathcal{E} = \{X \in \Pi \mid X \Rightarrow^* \varepsilon\}$; zde opět konstruujeme množiny $\mathcal{E}_1, \mathcal{E}_2, \ldots$, kde $\mathcal{E}_1 = \{X \in \Pi \mid (X \to \varepsilon) \in P\}$ a $\mathcal{E}_{i+1} = \mathcal{E}_i \cup \{X \in \Pi \mid \exists \alpha \in \mathcal{E}_i^* : (X \to \alpha) \in P\}$. Skončíme v případě $\mathcal{E}_n = \mathcal{E}_{n+1}$ – je zřejmé, že pak $\mathcal{E}_n = \mathcal{E}$.

Na základě $\mathcal{E}$ sestrojíme množinu pravidel P' takto: pro každé pravidlo $(X \to \alpha) \in P$ zařadíme do P' všechna možná pravidla $X \to \beta$, kde β vznikne z α vypuštěním některých (třeba žádných) výskytů symbolů z $\mathcal{E}$; přitom ovšem vynecháme (nezařazujeme) případnou možnost $X \to \varepsilon$.

Položíme $G' = (\Pi, \Sigma, S, P')$; lze snadno ověřit, že skutečně $L(G') = L(G) - \{\varepsilon\}$ (formálně lze postupovat např. indukcí podle délky odvození). □

Důsledek 12.2.3 *Ke každé bezkontextové gramatice* $G = (\Pi, \Sigma, S, P)$ *existuje ekvivalentní bezkontextová gramatika* $G_1 = (\Pi_1, \Sigma, S_1, P_1)$*, kde* ε *může být pravou stranou pouze u pravidla* $S_1 \to \varepsilon$*; v takovém případě se pak* S_1 *nevyskytuje na pravé straně žádného z pravidel z* P_1.

Důkaz: Ke G lze sestrojit nevypouštějící gramatiku $G' = (\Pi, \Sigma, S, P')$. Platí-li $\varepsilon \notin L(G)$ (S nepatří do výše zmíněné $\mathcal{E}$), položíme $G_1 = G'$. Je-li $\varepsilon \in L(G)$, vznikne G_1 z G' přidáním nového neterminálu S_1, který bude počátečním, a pravidel $S_1 \to \varepsilon$, $S_1 \to S$. □

Úkol 57 *K bezkontextové gramatice* G *dané uvedenými pravidly sestrojte nevypouštějící gramatiku* G' *takovou, že* $L(G') = L(G) - \{\varepsilon\}$.
$S \longrightarrow AB \,|\, \varepsilon$
$A \longrightarrow aAAb \,|\, BS \,|\, CA$
$B \longrightarrow BbA \,|\, CaC \,|\, \varepsilon$
$C \longrightarrow aBB \,|\, bS$

12.3 Chomského normální forma

Z technických důvodů je užitečné, že bezkontextové gramatiky lze transformovat do různých *normálních forem*, u nichž jsou kladena další syntaktická omezení na povolená pravidla. Příkladem je tzv. Chomského normální forma:

Definice 12.3.1 *Bezkontextová gramatika je v* ***Chomského normální formě****, zkráceně v CHNF, jestliže každé její pravidlo je tvaru* $X \to YZ$ *nebo* $X \to a$*, kde* X, Y, Z *označují neterminální symboly a* a *terminální symbol.*

Věta 12.3.2 *Ke každé bezkontextové gramatice* G *lze sestrojit gramatiku* G' *v CHNF tž.* $L(G') = L(G) - \{\varepsilon\}$.

Důkaz: Podle věty 12.2.2 můžeme rovnou předpokládat, že G je nevypouštějící (jinak ji do této formy převedeme). Ukážeme, jak postupnou transformací G zkonstruujeme ekvivalentní gramatiku G' v CHNF.

Nejprve z gramatiky G odstraníme pravidla typu $X \to Y$:

Pro každý neterminál A zkonstruujeme množinu $\mathcal{D}_A = \{B \mid A \Rightarrow^* B\}$ (jak ?); pak pro každé pravidlo $B \to \alpha$, kde $B \in \mathcal{D}_A$ a α není rovno jednomu neterminálu, přidáme pravidlo $A \to \alpha$. Nakonec odstraníme všechna pravidla typu $X \to Y$. Snadno lze ověřit, že generovaný jazyk zůstává zachován.

Dále pro každý terminál a zavedeme nový neterminál A_a a přidáme pravidlo $A_a \to a$. Pak na pravé straně každého pravidla $X \to \alpha$, kde $|\alpha| \geq 2$, nahradíme každý výskyt terminálu a neterminálem A_a.

Je zřejmé, že generovaný jazyk zůstává stále zachován; jediná pravidla porušující podmínku CHNF mohou být typu $X \to Y_1 Y_2 \ldots Y_n$, kde $n \geq 3$ (Y_i jsou samozřejmě neterminály).

Každé pravidlo uvedeného typu lze ovšem nahradit soustavou $X \to Y_1 Z_1$, $Z_1 \to Y_2 Z_2$, …, $Z_{n-3} \to Y_{n-2} Z_{n-2}$, $Z_{n-2} \to Y_{n-1} Y_n$, kde $Z_1, Z_2, \ldots, Z_{n-2}$ jsou nově přidané neterminály.

Opět je snadné se přesvědčit, že generovaný jazyk se nezmění a uvedenými změnami vzniklá gramatika G' je požadovanou gramatikou v CHNF. □

Úkol 58 *Následující gramatiky převeďte do Chomského normální formy:*

$S \longrightarrow A \mid 0SA \mid \varepsilon$	$S \longrightarrow (E)$	$S \longrightarrow abS \mid CaS \mid BaS \mid a$
$A \longrightarrow 1A \mid 1 \mid B1$	$E \longrightarrow F + F \mid F \times F$	$B \longrightarrow aCB \mid SC$
$B \longrightarrow 0B \mid 0 \mid \varepsilon$	$F \longrightarrow a \mid S$	$C \longrightarrow BCb \mid SB$

12.4 Greibachové normální forma

V některých situacích je užitečná tato normální forma:

Definice 12.4.1 *Bezkontextová gramatika je v* ***Greibachové normální formě****, zkráceně v GNF, jestliže každé její pravidlo je v tvaru* $X \to aY_1Y_2 \dots Y_n$ *(*$n \geq 0$*,* a *je terminál,* Y_i *jsou neterminály).*

Věta 12.4.2 *Ke každé bezkontextové gramatice* G *lze sestrojit gramatiku* G' *v GNF tž.* $L(G') = L(G) - \{\varepsilon\}$*.*

Podrobný důkaz zde neuvádíme; zmiňme ale, že základní ‘procedury’ při převodu gramatiky do GNF jsou zachyceny v následujících dvou lemmatech. První je očividné:

Lemma 12.4.3 *Mějme bezkont. gramatiku* $G = (\Pi, \Sigma, S, P)$*. Nechť* P *obsahuje pravidlo* $A \to \alpha B\gamma$ *a* $B \to \beta_1$*,* $B \to \beta_2$*, …,* $B \to \beta_n$ *jsou všechna pravidla s* B *na levé straně. Potom odstraníme-li z* P *pravidlo* $A \longrightarrow \alpha B\gamma$ *a naopak přidáme pravidla* $A \longrightarrow \alpha\beta_1\gamma$*,* $A \longrightarrow \alpha\beta_2\gamma$*, …,* $A \longrightarrow \alpha\beta_n\gamma$*, dostaneme gramatiku ekvivalentní s* G*.*

Další lemma je základem pro odstranění *levé rekurze*, tj. případu $X \Rightarrow^* X\alpha$; to je důležité pro syntaktickou analýzu v překladačích.

Lemma 12.4.4 *Mějme bezkont. gramatiku* $G = (\Pi, \Sigma, S, P)$*. Nechť* $A \to A\alpha_1$*,* $A \to A\alpha_2$*, …,* $A \to A\alpha_m$*,* $A \to \beta_1$*,* $A \to \beta_2$*, …,* $A \to \beta_n$ *jsou všechna pravidla s* A *na levé straně, přičemž řetězce* β_i *nezačínají* A*. Gramatika* $G' = (\Pi \cup \{Z\}, \Sigma, S, P')$ *vzniklá z* G *dodáním nového neterminálu* Z *a nahrazením všech uvedených pravidel soustavou* $A \to \beta_i$*,* $A \to \beta_i Z$ *(*$i = 1, 2, \dots, n$*),* $Z \to \alpha_i$*,* $Z \to \alpha_i Z$ *(*$i = 1, 2, \dots, m$*), je ekvivalentní gramatice* G*.*

Kapitola 13

Zásobníkové automaty; ekvivalence s bezkontextovými gramatikami

Cíl kapitoly:
Pochopení pojmu zásobníkového automatu a zvládnutí jeho návrhu v jednoduchých případech. Pochopení algoritmů převodu mezi bezkontextovými gramatikami a zásobníkovými automaty.

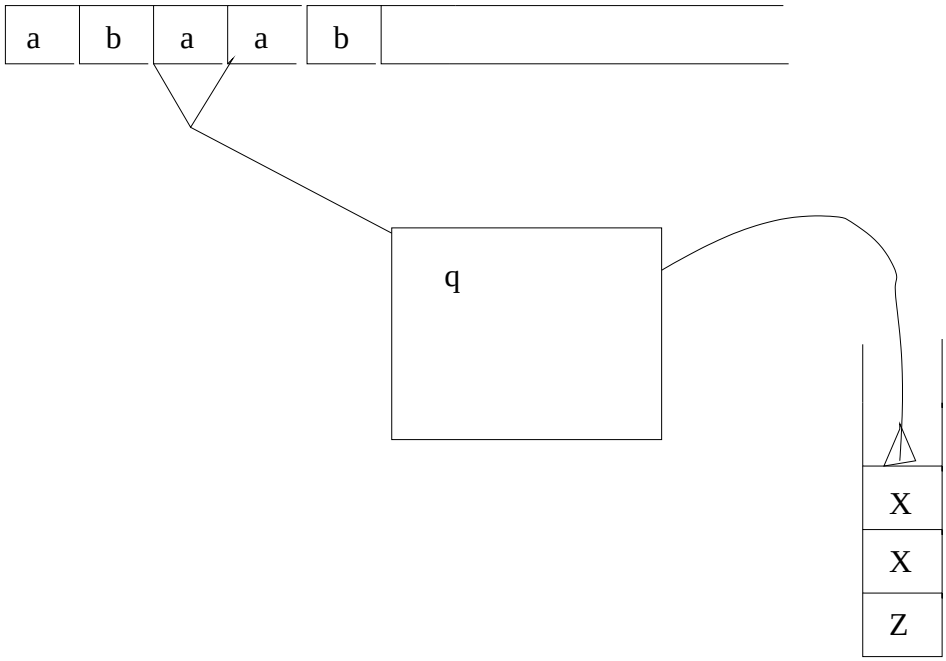
Studijní cíle: Pochopení pojmu zásobníkového automatu a zvládnutí jeho návrhu v jednoduchých případech. Pochopení algoritmů převodu mezi bezkontextovými gramatikami a zásobníkovými automaty.

Víme, že např. jazyk

$$\{\langle begin\rangle\langle end\rangle,\ \langle begin\rangle\langle begin\rangle\langle end\rangle\langle end\rangle,\ \langle begin\rangle\langle begin\rangle\langle begin\rangle\langle end\rangle\langle end\rangle\langle end\rangle,\ \ldots\}$$

nebo "ekvivalentní" jazyk $L = \{a^n b^n \mid n \geq 1\}$ nelze rozpoznávat konečným automatem.

Snadno ovšem takový jazyk (tedy slova daného jazyka) rozpoznáme zařízením podobným konečnému automatu, které může navíc používat neomezenou (tedy potenciálně nekonečnou) paměť typu *zásobník*: přečtené symboly a se prostě ukládají do zásobníku a při čtení symbolů b se pak tyto zásobníkové symboly odebírají – tím způsobem jsme schopni počet a-ček a b-ček porovnat. Zmíněnému zařízení budeme říkat zásobníkový automat, zkráceně ZA. "Vnější pohled" na ZA je ilustrován obrázkem 13.



Čtenář by si měl být schopen udělat představu, jak takové zařízení pracuje a jakým způsobem reprezentuje (rozpoznává) jazyk. Tuto představu je pak potřebné konfrontovat s níže uvedenou definicí (která odstraňuje všechny případné nejasnosti či nejednoznačnosti). Zdůrazněme hned, že obecným termínem "zásobníkový automat" se obvykle myslí "*nedeterministický* zásobníkový automat".

Definice 13.0.5 ***Zásobníkový automat****, zkráceně (ZA)* M *je dán parametry* $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$*, kde*

- Q *je konečná neprázdná množina* stavů
- Σ *je konečná neprázdná množina* vstupních symbolů *(* ***vstupní abeceda****)*
- Γ *je konečná neprázdná množina* zásobníkových symbolů *(* ***zásobníková abeceda****)*
- $q_0 \in Q$ *je* počáteční stav
- $Z_0 \in \Gamma$ *je* počáteční zásobníkový symbol
- δ *je zobrazení množiny* $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ *do množiny všech konečných podmnožin množiny* $Q \times \Gamma^*$*.*

Konfigurací zásobníkového automatu $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ *rozumíme trojici* (q, w, α)*, kde* $q \in Q$*,* $w \in \Sigma^*$*,* $\alpha \in \Gamma^*$*.*

Na množině všech konfigurací automatu M *definujeme relaci* $\vdash_M$*:*

$(q, aw, X\beta) \vdash_M (q', w, \alpha\beta) \iff_{df} \delta(q, a, X) \ni (q', \alpha)$

kde $a \in (\Sigma \cup \{\varepsilon\})$*,* $w \in \Sigma^*$*,* $\beta \in \Gamma^*$*. Říkáme pak, že konfigurace* $(q, aw, X\beta)$ bezprostředně vede ke *(resp. může bezprostředně vést ke) konfiguraci* $(q', w, \alpha\beta)$ *apod.*

Relace $\vdash_M^*$ *je reflexivním a tranzitivním uzávěrem relace* $\vdash_M$*.* $K_1 \vdash_M^* K_2$ *pak čteme: konfigurace* K_1 *vede k (resp. může vést k)* K_2 *apod.*

Slovo $w \in \Sigma^*$ je přijímáno *ZA* M*, jestliže* $(q_0, w, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon)$ *pro nějaké* $q \in Q$*.*

Jazykem rozpoznávaným *ZA* M *rozumíme jazyk* $L(M) = \{ w \in \Sigma^* \mid w$ *je přijímáno ZA* $M \}$*.*

Úkol 59 *Sestrojte zásobníkový automat rozpoznávající jazyk* $L = \{ wc(w)^R \mid w \in \{a,b\}^* \}$.
Pak navrhněte ZA rozpoznávající jazyk $\{ww^R \mid w \in \{0,1\}^*\}$. *Jistě přitom využijete nedeterminismus (o důvodu se zmíníme později).*
Sestrojte ještě zásobníkový automat rozpoznávající jazyk $L = \{ u \in \{a,b,c\}^* \mid$ *po vynechání všech výskytů symbolu* c *z* u *dostaneme slovo ve tvaru* $w(w)^R \}$.

Zásobníkové automaty tvoří "automatový protějšek" k bezkontextovým gramatikám, tj. rozpoznávají právě bezkontextové jazyky. Toto si teď ukážeme. Nejdříve ve směru od gramatice k automatu, což mj. ilustruje základní ideu syntaktické analýzy v překladačích.

Lemma 13.0.6 *Ke každé bezkontextové gramatice* G *lze sestrojit ZA* M *(s jedním stavem) tž.* $L(M) = L(G)$.

Důkaz: Mějme $G = (\Pi, \Sigma, S, P)$. K ní zkonstruujme ZA $M = (\{q_0\}, \Sigma, \Pi \cup \Sigma, \delta, q_0, S)$, kde pro každé $X \in \Pi$ je $\delta(q_0, \varepsilon, X) = \{(q_0, \alpha) \mid (X \to \alpha) \in P\}$ a pro každé $a \in \Sigma$ je $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$; jiným argumentům přiřazuje δ prázdnou množinu.

Dá se snadno ukázat $S \Rightarrow_G^* u\alpha \iff (q_0, u, S) \vdash_M^* (q_0, \varepsilon, \alpha)$. Zde $u \in \Sigma^*$, $\alpha \in \Pi(\Pi \cup \Sigma)^*$ nebo $\alpha = \varepsilon$; symbolem $\Rightarrow_G^*$ přitom zde označujeme relaci odpovídající *levému* odvození. □

ZA uvedený v důkazu provádí (nedeterministicky) tzv. analýzu "shora dolů": sledujíce jistou levou derivaci, snaží se de facto budovat derivační strom pro dané vstupní slovo od kořene k listům (průchodem zleva doprava).

Úkol 60 *Demonstrujte úspěšný běh (nedeterministického) zásobníkového automatu při syntaktické analýze shora dolů slova* $a * (a + a)$ *podle gramatiky*

1/ $A \longrightarrow A + B$
2/ $A \longrightarrow B$
3/ $B \longrightarrow B * C$
4/ $B \longrightarrow C$
5/ $C \longrightarrow (A)$
6/ $C \longrightarrow a$

Úkol 61 *Zkuste se alespoň zamyslet nad konstrukcí ZA ke gramatice (na uvedeném konkrétním příkladu i obecně) tak, aby prováděl analýzu "zdola nahoru", tj., aby sledoval jistou pravou derivaci pozpátku, budujíce derivační strom od listů ke kořeni.*

Poznámka. Na deterministických verzích takových zásobníkových automatů (pro speciální třídy gramatik), jsou založeny algoritmy používané u syntaktické analýzy v reálných překladačích. (Např. se jedná o tzv. LL- či LR-analyzátory.)

Ukázali jsme tedy, že k bezkontextové gramatice lze zkonstruovat ekvivalentní (dokonce jednostavový) zásobníkový automat. V případě jednostavového ZA lze snadno provést i opačnou transformaci, zachycenou následujícím lemmatem.

Lemma 13.0.7 *Ke každému ZA* M *s jedním stavem lze sestrojit bezkontextovou gramatiku* G *tž.* $L(G) = L(M)$.

Důkaz: Mějme $M = (\{q_0\}, \Sigma, \Gamma, \delta, q_0, Z_0)$; předpokládejme $\Sigma \cap \Gamma = \emptyset$ (toho docílíme případným přejmenováním zásobníkových symbolů). Ověřte (viz další Úkol), že následující gramatika je onou požadovanou: $G = (\Gamma, \Sigma, Z_0, P)$, kde $\delta(q_0, a, A) \ni (q_0, \alpha) \iff (A \to a\alpha) \in P$ ($a \in (\Sigma \cup \{\varepsilon\})$). □

Úkol 62 *Uvažujme ZA $M = (\{q_0\}, \Sigma, \Gamma, \delta, q_0, Z_0)$, kde $\Sigma \cap \Gamma = \emptyset$ a k němu sestrojenou BG $G = (\Gamma, \Sigma, Z_0, P)$ takovou, že $(A \to a\alpha) \in P \iff \delta(q_0, a, A) \ni (q_0, \alpha)$ ($a \in (\Sigma \cup \{\varepsilon\})$).*

Ukažte indukcí (podle počtu kroků odvození), že

$Z_0 \Rightarrow_G^* ua \iff (q_0, u, Z_0) \vdash_M^* (q_0, \varepsilon, \alpha)$ *(zde $u \in \Sigma^*$, $\alpha \in \Gamma^*$ a $\Rightarrow_G^*$ označuje* levé *odvození).*

Další lemma pak ukáže, že obecný ZA lze převést na jednostavový. To je technicky obtížnější, byť idea není nijak složitá – informaci o řídicím stavu původního ZA M musí mít nový jednostavový (tedy de facto "bezstavový", jakoby s pamětí 0 bitů) ZA M' při "simulaci" původního M vhodně uloženu na zásobníku. Jako obvykle je to "něco za něco": za zrušení řídicích stavů platíme rozšířením zásobníkové abecedy. (Při čtení důkazu je vhodné rovnou provádět úkol 63.)

Lemma 13.0.8 *Ke každému ZA M lze sestrojit ZA M' s jedním stavem tž. $L(M) = L(M')$.*

Důkaz: Idea: Jednostavový ZA M' (stav označíme s) bude mít zásobníkové symboly typu $\langle p, X, q\rangle$, kde p, q jsou stavy a X je zásobníkový symbol automatu M, přičemž bude platit:

$$\forall w : (s, w, \langle p, X, q\rangle) \vdash_{M'}^* (s, \varepsilon, \varepsilon) \iff (p, w, X) \vdash_M^* (q, \varepsilon, \varepsilon)$$

Konkrétně pro $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ konstruujeme $M' = (\{s\}, \Sigma, \Gamma', \delta', s, R)$, kde $\Gamma' = (Q \times \Gamma \times Q) \cup \{R\}$ a δ' je určena následovně:

- $\delta'(s, \varepsilon, R) = \{(s, \langle q_0, Z_0, q\rangle) \mid q \in Q\}$,
- pro $(q', \varepsilon) \in \delta(q, a, X)$ ($a \in (\Sigma \cup \{\varepsilon\})$) zařadíme do $\delta'(s, a, \langle q, X, q'\rangle)$ prvek (s, ε),
- pro $(q', A_1A_2 \ldots A_n) \in \delta(q, a, X)$ ($n \geq 1$) zařadíme do $\delta'(s, a, \langle q, X, \overline{q}\rangle)$ prvek $(s, \langle q', A_1, q_1\rangle\langle q_1, A_2, q_2\rangle \ldots \langle q_{n-1}, A_n, \overline{q}\rangle)$ pro každé $\overline{q}, q_1, q_2, \ldots, q_{n-1} \in Q$.

(Chápeme-li δ' jako množinu 'instrukcí', pak lze říci, že δ' je minimální množina instrukcí splňující výše uvedené podmínky.)

Dá se ověřit, že každému přijímajícímu výpočtu automatu M nad slovem w odpovídá přijímající výpočet automatu M' nad w a naopak. □

Úkol 63 *K zásobníkovému automatu M se vstupní abecedou $\{a, b\}$, zásobníkovou abecedou $\{A, B\}$, počátečním zásobníkovým symbolem A, množinou stavů $\{p, q, r\}$, počátečním stavem p a přechodovou funkcí δ definovanou následovně*

$\delta(p, a, A) = \{(q, AA), (p, B)\}$,
$\delta(q, b, A) = \{(q, AA)\}$,
$\delta(p, \varepsilon, B) = \{(q, A)\}$,
$\delta(q, \varepsilon, A) = \{(r, \varepsilon)\}$,
$\delta(r, a, A) = \{(r, A)\}$,
$\delta(r, b, A) = \{(r, \varepsilon)\}$

(pro ostatní prvky def. oboru je funkční hodnota rovna $\emptyset$) sestrojte nejdříve jednostavový *ZA rozpoznávající jazyk $L(M)$, a poté gramatiku generující tento jazyk. Použijte přitom konstrukce obsažené ve výše uvedených důkazech.*

Z uvedených lemmat ihned plyne

Věta 13.0.9 *(Nedeterministické) zásobníkové automaty rozpoznávají právě bezkontextové jazyky (a jsou takto ekvivalentní bezkontextovým gramatikám).*

Definovali jsme, že slovo je přijímáno ZA právě tehdy, když se po jeho přečtení ZA může ocitnout v situaci (konfiguraci) s prázdným zásobníkem. Obvykle se uvažuje také forma přijímání slova možným dosažením koncového stavu (řídicí jednotky). Obě alternativy jsou definovány níže a je ukázáno, že obě také mají tutéž rozpoznávací sílu (v případě *nedeterministických* ZA).

Definice 13.0.10 *Pro ZA* $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ *(kde jsme přidali parametr* F*, což je množina* koncových (přijímajících) *stavů –* $F \subseteq Q$*) definujeme* jazyk rozpoznávaný koncovým stavem $L_{KS}(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (q, \varepsilon, \alpha)$ *pro nějaké* $q \in F$ *a* $\alpha \in \Gamma^*\}$ *a* jazyk rozpoznávaný prázdným zásobníkem $L_{PZ}(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon)$ *pro nějaké* $q \in Q\}$.

Lemma 13.0.11 *K libovolnému ZA* M_1 *lze zkonstruovat ZA* M_2 *tž.* $L_{KS}(M_1) = L_{PZ}(M_2)$ *a také* M_2' *tž.* $L_{PZ}(M_1) = L_{KS}(M_2')$.

Důkaz: Neformální idea spočívá v následujícím: každý ZA lze jednoduše upravit tak, že dodáme nový počáteční zásobníkový symbol B (bottom=dno), který se bude stále vyskytovat na dně zásobníku (a pouze tam) – promyslete si technické podrobnosti ! Pak už je důkaz tvrzení přímočarý. □

Kapitola 14

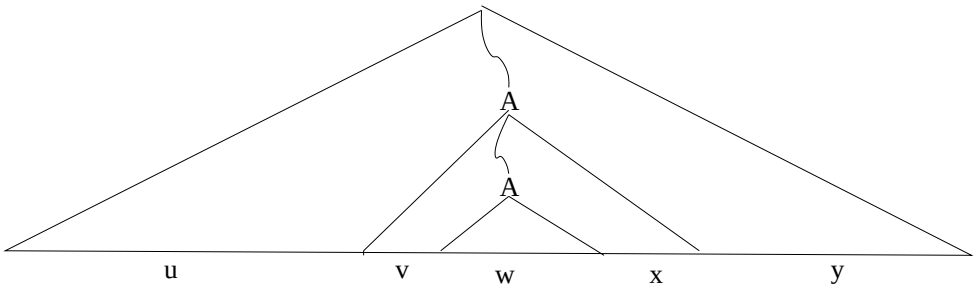
Pumping lemma pro bezkontextové jazyky

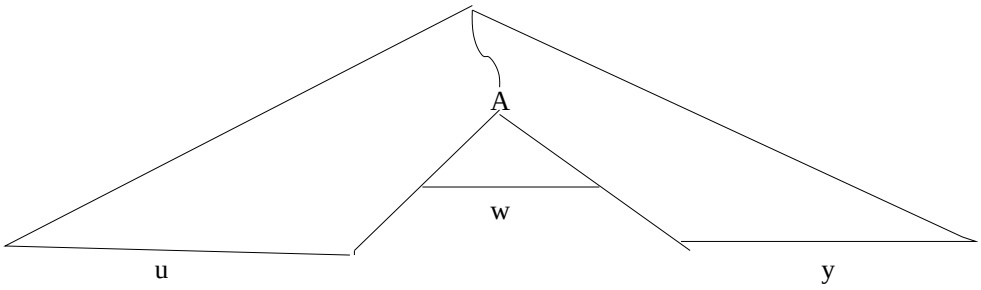
Cíl kapitoly:
Získání schopnosti v běžných případech poznat a prokázat vlastnosti, které činí konkrétní jazyk nebezkontextovým.

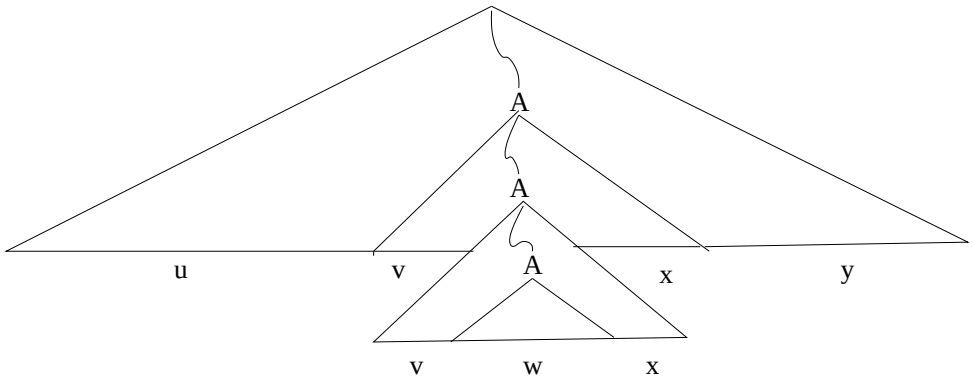
Studijní cíle: Získání schopnosti v běžných případech poznat a prokázat vlastnosti, které činí konkrétní jazyk nebezkontextovým.

Připomeňme si, jak jsme dokazovali, že jazyk $\{a^n b^n \mid n \geq 0\}$ není regulární, a jak jsme odvodili pumping lemma platné obecně pro regulární jazyky. Je jasné, že uvedený jazyk přijímá jednoduchý zásobníkový automat. Uvažujme nyní ale jazyk

$$L = \{a^n b^n c^n \mid n \geq 0\}$$


A
A
u
v
w
x
y


A
w
u
y


A
A
A
u
v
x
y
v
w
x

Jistě nás naše intuice brzy dovede k závěru, že na takový jazyk zásobníkový automat (byť nedeterministický) nestačí. Jak to ale jasně dokázat ? Opět přivedením předpokladu, že L je bezkontextový, k logickému sporu. Předpokládejme tedy, že L je bezkontextový a uvažujme bezkontextovou gramatiku G, která ho generuje (uvažovat gramatiku se ukáže pro naše účely vhodnější než uvažovat zásobníkový automat). Pro každé slovo $a^n b^n c^n$ tedy existuje derivační strom (podle gramatiky G). Vezmeme-li slovo "velmi dlouhé" (tj. n "velmi velké"), v příslušném derivačním stromu nutně dochází k opakování nějakého neterminálu na nějaké větvi (tj. cestě od kořene k listu) – viz obr. 14. Přesněji řečeno: derivačních stromů, ve kterých se takové opakování nevyskytuje, je konečně mnoho (proč ?); výraz "n je velmi velké" lze zpřesnit tak, že $3n$ (tj. délka slova $a^n b^n c^n$) je větší než délka nejdelšího slova odvoditelného derivačním stromem bez opakování.

Vezměme nyní tedy ono velmi dlouhé slovo $a^n b^n c^n$ a pro něj *nejmenší* možný derivační strom (i ten má opakování jako na obr. 14). Slovo $a^n b^n c^n$ se dá psát ve tvaru $uvwxy$ (jak znázorněno na obrázku), kde navíc aspoň jedno ze slov v, x je neprázdné (jinak bychom mohli oba uzly označené neterminálem A ztotožnit a získali bychom pro $a^n b^n c^n$ menší derivační strom !). Je jasné, že i pro uwy, a také uv^2wx^2y, uv^3wx^3y, $\ldots$ existují derivační stromy (viz obr. 14); tato slova tudíž také patří do L.

Snadno se ale můžeme přesvědčit, že ať rozdělíme slovo $a^n b^n c^n$ na 5 částí $uvwxy$ *jakkoliv*, přičemž alespoň jedno ze slov v, x je neprázdné, pak slovo uwy zaručeně nepatří do L (proč ?).

Opět jsme odvodili určité "pumping lemma" platné obecně pro bezkontextové jazyky (nikoli jen pro náš L) a demonstrovali jsme jeho použití pro důkaz, že L není bezkontextový. Zmíněné lemma následuje.

Věta 14.0.12 (Pumping lemma pro bezkontextové jazyky, neboli $uvwxy$-teorém.) *Nechť L je bezkontextový jazyk. Pak existují přirozená čísla p, q tž. každé slovo $z \in L$, $|z| > p$, lze psát ve tvaru $z = uvwxy$, přičemž platí $|vx| \geq 1$ (aspoň jedno ze slov v, x je neprázdné), $|vwx| \leq q$, a pro vš. $i \geq 0$ je $uv^i wx^i y \in L$.*

Důkaz: Čtenář jistě pochopil, že jako ono p můžeme vzít jakékoli číslo větší než délka nejdelšího slova, pro nějž existuje derivační strom bez opakování.

A odkud se vezme ono q zaručující, že lze dokonce omezit délku úseku vwx ? Podstrom na obr. 14 (s kořenem v 'horním' A) lze zvolit tak, že neobsahuje žádné jiné opakování neterminálů – a takových (pod)stromů je zřejmě jen konečně mnoho možných.

Uvedeme nyní podrobnější verzi důkazu s konkrétnějšími odhady čísel p, q.

Nechť $L = L(G)$ pro bezkontextovou gramatiku $G = (\Pi, \Sigma, S, P)$ v CHNF (náležení či nenáležení prázdného slova do L zde nehraje roli).

Předpokládejme, že pro nějaké slovo $z \in L$ existuje derivační strom, v němž se na jedné větvi (tj. cestě od kořene k listu) vyskytují alespoň dva vrcholy označené stejným neterminálem, řekněme A. Pak je zřejmé, že $S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvwxy = z$ pro nějaké $u, v, w, x, y \in \Sigma^*$.

Nechť $|\Pi| = k$ (k tedy označuje počet neterminálů). Všimněme si:

a/ na každé větvi derivačního stromu délky alespoň $k + 1$ jsou nejméně dva vrcholy označeny stejným neterminálem;

b/ máme-li derivační strom pro $z \in \Sigma^*$, v němž jsou všechny větve kratší než $k + 1$, pak nutně $|z| \leq 2^{k-1}$;

c/ vezmeme-li lib. $z \in L$ tž. $|z| > 2^{k-1}$ a derivační strom pro z, pak určitě na nejdelší větvi se vyskytují dva různé vrcholy v_1, v_2 (v_1 blíž ke kořeni) označené stejným neterminálem; přitom lze jistě v_1 zvolit tak, že jeho max. vzdálenost k listu je nejvýše $k+1$. To znamená, že podstrom s kořenem v_1 má nejvýše 2^k listů.

Stačí tedy jako hledaná p, q vzít čísla 2^{k-1} a 2^k. □

Pro naše další účely je vhodnější poněkud jednodušší verze věty 14.0.12 (n lze vzít jako $max(p,q)+1$):

Věta (jiná verze **14.0.12**). *Nechť L je bezkontextový jazyk. Pak existuje přirozené číslo n tž. každé slovo $z \in L$, $|z| \geq n$, lze psát ve tvaru $z = uvwxy$, přičemž platí $|vx| \geq 1$, $|vwx| \leq n$ a pro vš. $i \geq 0$ je $uv^iwx^iy \in L$.*

Všimněte si opět střídání kvantifikátorů:

$(\exists n)\ (\forall z \text{ tž. } z \in L, |z| \geq n)$
$(\exists\, u,v,w,x,y \text{ tž. } z = uvwxy, |vwx| \leq n, |vx| \geq 1)\ (\forall i \geq 0) : uv^iwx^iy \in L$

A opět se nabízí hra dvou hráčů:

1. A zvolí $n \in \mathcal{N}$
2. B zvolí slovo z tž. $z \in L$ a $|z| \geq n$
3. A zvolí u,v,w,x,y tž. $z = uvwxy$, $|vwx| \leq n$ a $|vx| \geq 1$
4. B zvolí $i \geq 0$
5. *Výsledek:* je-li $uv^iwx^iy \in L$, pak vyhrál A, v případě $uv^iwx^iy \notin L$ vyhrál B.

Je zřejmé, že je-li L bezkontextový, pak A má vítěznou strategii v uvedené hře. Jinak řečeno:

Má-li B vítěznou strategii, pak L není bezkontextový.

Navržení vítězné strategie hráče B je častým prostředkem k důkazu toho, že uvažovaný jazyk není bezkontextový.
Pro výše zkoumaný $L = \{a^nb^nc^n \mid n \geq 0\}$ můžeme vítěznou strategii hráče B formulovat takto.

1. A zvolí (libovolné) $n \in \mathcal{N}$
2. B zvolí $z = a^nb^nc^n$
3. A zvolí libovolné u,v,w,x,y tž. $z = uvwxy$, $|vwx| \leq n$ a $|vx| \geq 1$,
4. B zvolí $i = 0$ (lze také kterékoli $i \geq 2$)
5. Jelikož $|vwx| \leq n$, slova v, x neobsahují aspoň jeden ze symbolů a, b, c (a samozřejmě aspoň jeden obsahují). Proto ve slově uwy nemůže být stejný počet symbolů a, b i c a slovo tedy nepatří do L. B vyhrává.

Poznámka. Z úvodní analýzy (před Větou 14.0.12) víme, že B má vítěznou strategii i při ignorování podmínky $|vwx| \leq n$. Pak sice nemůžeme v posledním bodě jednoduše argumentovat, že v, x neobsahují aspoň jeden ze symbolů a, b, c, ale nepříslušnost slova uwy k L lze dokázat mírně složitějšími úvahami (které jste už, doufejme, provedli před větou 14.0.12). V následujícím příkladu je už podmínka $|vwx| \leq n$ skutečně nutná.

Ukážeme, že jazyk

$$L = \{ww \mid w \in \{0,1\}^*\}$$

není bezkontextový:

1. A zvolí (libovolné) $n \in \mathcal{N}$
2. B zvolí $z = 0^n 1^n 0^n 1^n$
3. A zvolí libovolné u, v, w, x, y tž. $z = uvwxy$, $|vwx| \leq n$ a $|vx| \geq 1$,
4. B zvolí $i = 0$ (lze také kterékoli $i \geq 2$)
5. Jelikož $|vwx| \leq n$, slova v, x zasahují nejvýš do jednoho úseku nul a nejvýš jednoho úseku jedniček v $z = 0^n 1^n 0^n 1^n$ (přičemž alespoň do jednoho úseku zasahují). Tedy $uwy = 0^{k_1} 1^{k_2} 0^{\ell_1} 1^{\ell_2}$, kde určitě $k_1 \neq \ell_1$ nebo $k_2 \neq \ell_2$. Tedy uwy nepatří do L a B vyhrává.

Úkol 64 *Dokažte, že následující jazyky nejsou bezkontextové:*

$L_1 = \{\, 0^m 1^n 0^m \mid 0 \leq n \leq m \,\}$

$L_2 = \{\, a^k \mid k = n^2$ *pro nějaké* $n \geq 1 \,\}$

Úkol 65 *Zjistěte, které z daných jazyků*

jsou regulární:

jsou bezkontextové, ale ne regulární:

nejsou bezkontextové:

$L_1 = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$ $L_2 = \{w \in \{a,b\}^* \mid |w|_a$ *je sudé* $\}$ $L_3 = \{w \in \{a,b\}^* \mid w$ *obsahuje podslovo* $abba\}$ $L_4 = \{w \in \{a,b,c\}^* \mid |w|_a = |w|_b = |w|_c\}$ $L_5 = \{w \in \{a,b\}^* \mid |w|_a$ *je prvočíslo*$\}$ $L_6 = \{\, 0^m 1^n \mid m \leq 2n \,\}$ $L_7 = \{\, 0^m 1^n 0^m \mid m = 2n \,\}$

Kapitola 15

Uzávěrové vlastnosti třídy bezkontextových jazyků

Cíl kapitoly:
Zvládnutí modulárního návrhu složitějších bezkontextových gramatik (a zásobníkových automatů). Pochopení algoritmů prokazujících uzavřenost třídy bezkontextových jazyků vůči různým operacím; získání náhledu na to, vůči kterým operacím třída uzavřená není.

Studijní cíle: Zvládnutí modulárního návrhu složitějších bezkontextových gramatik (a zásobníkových automatů). Pochopení algoritmů prokazujících uzavřenost třídy bezkontextových jazyků vůči různým operacím; získání náhledu na to, vůči kterým operacím třída uzavřená není.

Zkratkou CFL budeme označovat třídu bezkontextových jazyků. CFL není uzavřena na všechny operace, na které je uzavřena třída regulárních jazyků. Nejdříve si ukážeme případy operací, vůči nimž CFL uzavřena je. Důkazy jsou samozřejmě opět konstruktivní – ukazují algoritmy, které k zadané reprezentaci jazyků-operandů zkonstruují reprezentaci jazyka-výsledku operace. (Příslušnou reprezentací jsou samozřejmě bezkontextové gramatiky či zásobníkové automaty ; lze volit, co je vhodnější.)

Věta 15.0.13 *CFL je uzavřena vůči sjednocení, zřetězení, iteraci, zrcadlovému obrazu, substituci (tedy i homomorfismu).*

Důkaz: K bezkontextovým gramatikám $G_1 = (\Pi_1, \Sigma, S_1, P_1)$, $G_2 = (\Pi_1, \Sigma, S_1, P_1)$ lze zkonstruovat gramatiku $G = (\Pi, \Sigma, S, P)$ tž. $L(G) = L(G_1) \cup L(G_2)$ takto: Předpokládáme, že $\Pi_1 \cap \Pi_2 = \emptyset$ (docílíme toho případným přejmenováním neterminálů). Položíme $\Pi = \Pi_1 \cup \Pi_2 \cup \{S\}$, kde $S \notin \Pi_1 \cup \Pi_2$, a $P = P_1 \cup P_2 \cup \{S \longrightarrow S_1, S \longrightarrow S_2\}$.

Rovněž velmi přímočará je konstrukce gramatik generujících jazyky $L(G_1) \cdot L(G_2)$, $L(G_1)^*$, $L(G_1)^R$.

Podobně ke gramatice $G = (\Pi, \Sigma, S, P)$ a gramatikám G_a (pro vš. $a \in \Sigma$) lze snadno zkonstruovat gramatiku, která generuje jazyk vzniklý z $L(G)$ substituujeme-li za každé a jazyk $L(G_a)$. □

V důkazu další uzávěrové věty se více hodí reprezentace jazyků automaty.

Věta 15.0.14 *CFL je uzavřena vůči průniku s regulárním jazykem, i vůči kvocientu podle regulárního jazyka. (Tj. pro každý bezkontextový L a regulární R, jsou $L \cap R$, $R \backslash L$, L/R bezkontextové.)*

Důkaz: Idea pro průnik:
Lze podobně jako u dvou KA, zde lze příslušný KA "zabudovat" do řídicí jednotky ZA. (Stavová množina výsledného ZA je kartézským součinem stavových množin původního ZA a KA.)

Idea pro kvocient je:
Mějme ZA M a KA A. Připomeňme, že slovo u patří do $L(A) \backslash L(M)$ právě když ex. $v \in L(A)$ tak, že $vu \in L(M)$. Pro vytvoření ZA M' přijímající jazyk $L(A) \backslash L(M)$ opět použijeme myšlenku zabudování řídicí jednotky A do řídicí jednotky M. Nyní ale tak, že výsledný ZA M' dělá na začátku sérii ε-kroků, při nichž nedeterministicky "hádá" vhodné v.
(Zkuste dokončit promyšlení detailů konstrukce.) □

Neuzavřenost CFL vůči některým operacím se nejpříměji dokáže konstrukcí (jednoduchých) protipříkladů; je samozřejmě možné užít i další úvahy:

Věta 15.0.15 *CFL není uzavřena vůči průniku a doplňku.*

Důkaz: Jazyky $L_1 = \{a^i b^j c^k \mid i = j\}$, $L_2 = \{a^i b^j c^k \mid j = k\}$ jsou zřejmě bezkontextové. Přitom $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ bezkontextový není.

Z de Morganových pravidel plyne, že kdyby byla CFL uzavřena vůči doplňku, tak by díky uzavřenosti vůči sjednocení byla uzavřena i vůči průniku. □

Kapitola 16

Deterministické zásobníkové automaty

Cíl kapitoly:
Pochopení rozdílné situace u vztahu determinismu a nedeterminismu v případě konečných automatů a v případě zásobníkových automatů.

Studijní cíle: Pochopení rozdílné situace u vztahu determinismu a nedeterminismu v případě konečných automatů a v případě zásobníkových automatů.

Připomeňme si, že u zásobníkových automatů jsme jako základní vzali *nedeterministickou* verzi. Takto totiž ZA odpovídají bezkontextovým gramatikám. Vzniká přirozená otázka, jak to vypadá s deterministickou verzí zásobníkových automatů – determinismus je navíc potřebný, máme-li na takovém zařízení opravdu stavět (rychlý) *algoritmus* (např. již zmíněné syntaktické analýzy). Začneme s definicí *deterministického* zásobníkového automatu a deterministického bezkontextového jazyka:

Definice 16.0.16 ***Deterministický zásobníkový automat*** *(DZA) je ZA* $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$*, pro nějž platí:*

1. $\delta(q, a, X)$ *je vždy nejvýše jednoprvková množina (pro* $a \in \Sigma \cup \{\varepsilon\}$*) a*
2. *je-li* $\delta(q, \varepsilon, X) \neq \emptyset$*, pak* $\delta(q, a, X) = \emptyset$ *pro vš.* $a \in \Sigma$*.*

Jazyk L *je* deterministický bezkontextový jazyk*, jestliže* $L = L_{KS}(M)$ *pro nějaký DZA* M*.*

Smysl je jasný: pro každé vstupní slovo existuje jediný možný výpočet DZA M. Všimněme si, že v případě přijímání prázdným zásobníkem je přijímaný jazyk $L_{PZ}(M)$ nutně *bezprefixový* – pro slovo $u \in L_{PZ}(M)$ každý jeho vlastní prefix nepatří do $L_{PZ}(M)$. (Např. jazyk $\{\varepsilon, a\}$ bezprefixový není.) Proto jsou deterministické jazyky, tvořící třídu DCFL, definovány pomocí přijímání koncovým stavem.

Poznámka. Využitím "bottom-symbolu" lze opět snadno ukázat, že ke každému DZA M lze zkonstruovat DZA M' tž. $L_{PZ}(M) = L_{KS}(M')$. Na druhé straně lze ke každému DZA M zkonstruovat DZA M' tž. $L_{KS}(M) \cdot \{\$\} = L_{PZ}(M')$, kde $\$$ je přidaný koncový znak.

Na rozdíl od konečných automatů, deterministická verze zásobníkových automatů je skutečně slabší než nedeterministická, tedy DCFL je *vlastní* podtřídou CFL. Lze to vidět už díky jiným uzávěrovém vlastnostem třídy DCFL.

Věta 16.0.17 *Třída DCFL je uzavřena vůči doplňku. Na druhé straně není uzavřena vůči průniku ani vůči sjednocení.*

Uzavřenost vůči doplňku nelze sice demonstrovat prostým prohozením přijímajících a nepřijímajících stavů (proč ?), není ale těžké tuto myšlenku "dotáhnout". Neuzavřenost vůči průniku plyne např. z toho, že jazyky L_1, L_2 z důkazu věty 15.0.15 jsou deterministické. DCFL tedy nemůže být uzavřena ani vůči sjednocení (de Morganova pravidla).

Uzávěrové vlastnosti lze např. využít pro důkazy nepříslušnosti některých jazyků k DCFL. Např. jazyk $L = \{a^i b^j c^k \mid (i \neq j) \vee (j \neq k)\}$ není v DCFL (přitom zřejmě je v CFL): Kdyby byl, pak by i jeho doplněk $\overline{L}$ byl v DCFL, tedy i v CFL. Pak by ovšem i $\overline{L} \cap [a^*b^*c^*]$ byl v CFL (CFL je uzavřena vůči průniku s regulárním jazykem); ovšem $\overline{L} \cap [a^*b^*c^*] = \{\, a^n b^n c^n \mid n \geq 0\}$, a tedy bezkontextový není !

Využitím dalších uzávěrových vlastností se dá ukázat, že např. jazyky $\{ww^R \mid w \in \{a,b\}^*\}$, $\{a^i b^j c^k \mid (i = j) \vee (j = k)\}$ nejsou deterministické.

Poznámka. Vzpomeňme si, že existuje algoritmus, který o dvou zadaných konečných automatech rozhodne, zda jsou ekvivalentní (tj. zda přijímají tentýž jazyk). V kursu o vyčíslitelnosti a složitosti uvidíme, že podobný algoritmus pro (nedeterministické) zásobníkové automaty neexistuje. Od 60. let ale byla otevřena otázka, zda existuje algoritmus rozhodující ekvivalenci deterministických zásobníkových automatů . Pozitivní řešení prezentoval v r. 1997 G. Sénizergues, později důkaz zjednodušil C. Stirling. (Uvedení důkazu v našem kursu však pro jeho náročnost stále nepřipadá v úvahu.)

Kapitola 17

Chomského hierarchie

Cíl kapitoly:
Zvládnutí klasické klasifikace jazyků podle Chomského. Získání schopnosti zařadit běžné jazyky do uvedené hierarchie.

Studijní cíle: Zvládnutí klasické klasifikace jazyků podle Chomského. Získání schopnosti zařadit běžné jazyky do uvedené hierarchie.

Dříve uvedené bezkontextové gramatiky jsou speciálním případem obecných (generativních) gramatik. Od bezkontextových se liší jen tím, že na levé straně pravidel nestojí nutně jen jeden neterminál, ale obecně řetězec neterminálů a terminálů obsahující alespoň jeden neterminál.

Pro úplnost uvádíme úplnou obecnou definici:

Definice 17.0.18 ***Generativní gramatika*** *je dána čtveřicí parametrů* $G = (\Pi, \Sigma, S, P)$*, kde* Π *je konečná množina* neterminálních symbolů *(*neterminálů*),* Σ *je konečná množina* terminálních symbolů *(*terminálů*), přičemž* $\Pi \cap \Sigma = \emptyset$*,* $S \in \Pi$ *je* počáteční (startovací) neterminál *a* P *je konečná množina pravidel typu* $\alpha \to \beta$*, kde* $\alpha \in (\Pi \cup \Sigma)^* \Pi (\Pi \cup \Sigma)^*$ *a* $\beta \in (\Pi \cup \Sigma)^*$*.*

Uvažujme lib. $\gamma, \delta \in (\Pi \cup \Sigma)^*$*. Řekneme, že* γ se přímo přepíše *(*lze přímo přepsat*) na* δ *(podle pravidel gramatiky* G*), značíme* $\gamma \Rightarrow_G \delta$ *nebo jen* $\gamma \Rightarrow \delta$ *(když* G *zřejmá z kontextu), jestliže existují slova* $\mu_1, \mu_2, \alpha, \beta$ *tž.* $\gamma = \mu_1 \alpha \mu_2$*,* $\delta = \mu_1 \beta \mu_2$ *a* $\alpha \to \beta$ *je pravidlo v* P*.*

Řekneme, že γ se přepíše *na* δ*, značíme* $\gamma \Rightarrow^* \delta$*, jestliže existuje posloupnost* $\mu_0, \mu_1, \ldots, \mu_n$ *slov z* $(\Pi \cup \Sigma)^*$ *(pro něj.* $n \geq 0$*) tž.* $\gamma = \mu_0 \Rightarrow \mu_1 \Rightarrow \ldots \Rightarrow \mu_n = \delta$*. Zmíněnou posloupnost pak nazveme* odvozením *(*derivací*) délky* n *slova* δ *ze slova* γ*.*

Jazyk generovaný gramatikou G*, označme jej* $L(G)$*, je definován takto:* $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$*.*

Dvě gramatiky G_1, G_2 *nazveme* ekvivalentní*, jestliže* $L(G_1) = L(G_2)$*.*

Tzv. Chomského hierarchie vzniká omezením se na speciální typ pravidel:

Definice 17.0.19 *Obecná generativní gramatika $G = (\Pi, \Sigma, S, P)$ definovaná výše je* gramatika typu 0 *v tzv.* Chomského hierarchii*. G je* gramatika typu 1*, neboli* ***kontextová gramatika****, jestliže každé pravidlo v P je tvaru $\alpha X \beta \to \alpha \gamma \beta$, kde $|\gamma| \geq 1$ (připomeňme, že $\alpha, \beta, \gamma \in (\Pi \cup \Sigma)^*, X \in \Pi$); jedinou výjimkou může být pravidlo $S \to \varepsilon$, v kterémžto případě se pak S nesmí vyskytovat na pravé straně žádného pravidla. G je* gramatika typu 2*, neboli bezkontextová gramatika, jestliže každé pravidlo v P je tvaru $X \to \alpha$. G je* gramatika typu 3*, neboli* ***regulární gramatika****, jestliže každé pravidlo v P je tvaru $X \to wY$ nebo $X \to w$ ($w \in \Sigma^*$).*

Jazyk L *je* typu i *($i = 0, 1, 2, 3$) v Chomského hierarchii, jestliže jej generuje nějaká gramatika typu i. Speciálně řekneme, že* jazyk *je* kontextový *(*bezkontextový, regulární*), jestliže jej generuje nějaká kontextová (bezkontextová, regulární) gramatika.*

Všimněme si, že gramatika typu 3 je speciálním případem gramatiky typu 2, gramatika typu 1 je spec. případem gramatiky typu 0. Gramatika typu 2 (bezkontextová gramatika) nemusí být gramatikou typu 1 kvůli pravidlům s ε na pravé straně; je ji však možné do takové formy upravit (připomeňme si konstrukci nevypouštějící bezkontextové gramatiky). Označíme-li tedy $\mathcal{L}_i$ třídu jazyků typu i ($i = 0, 1, 2, 3$), pak je zřejmé

Lemma 17.0.20 $\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_0$

Ve skutečnosti jsou všechny inkluze vlastní, jak ještě zmíníme později.

Kapitola 18

Konečné automaty a regulární gramatiky

Teď si ukážeme, že nová definice regulárního jazyka nekoliduje s dřívější definicí; začneme technickým lemmatem.

Lemma 18.0.21 *Ke každé **regulární gramatice**, lze zkonstruovat ekvivalentní gramatiku, jejíž každé pravidlo je v jednom z tvarů $X \to aY$, $X \to Y$, $X \to \varepsilon$.*

Důkaz: Pravidlo typu $X \to a_1 a_2 \dots a_n Y$ ($n \geq 2$) nahradíme pravidly $X \to a_1 Z_1$, $Z_1 \to a_2 Z_2$, …, $Z_{n-1} \to a_n Y$, kde $Z_1, Z_2, \dots, Z_n$ jsou vždy nově přidané neterminály. □

Věta 18.0.22 *Jazyk je generován regulární gramatikou právě když je rozpoznáván konečným automatem.*

Důkaz: Nechť $A = (Q, \Sigma, \delta, q_0, F)$ je KA. Sestrojme $G = (Q, \Sigma, q_0, P)$, kde do P zařadíme $q \to aq'$ pro každé q, q', a tž. $\delta(q, a) = q'$ a navíc přidáme $q \to \varepsilon$ pro každé $q \in F$. Je snadné ověřit, že G je

regulární gramatika tž. $L(G) = L(A)$; indukcí je např. možné dokázat vztah $(\delta(q, w) = q') \Leftrightarrow (q \Rightarrow_G^* wq')$.

Naopak uvažujme gramatiku $G = (\Pi, \Sigma, S, P)$ s pravidly typu $X \to aY$, $X \to Y$, $X \to \varepsilon$. Sestrojme ZNKA $A = (\Pi, \Sigma, \delta, \{S\}, F)$, kde $Y \in \delta(X, a)$ ($a \in \Sigma \cup \{\varepsilon\}$) právě když $X \to aY$ patří do P. Navíc $F = \{X \mid (X \to \varepsilon) \in P\}$. Opět je snadné ověřit $L(A) = L(G)$. □

Úkol 66 *Rozšiřte konstrukci převodu KA na RG pro případ nedeterministického KA a aplikujte ji v případě NKA zadaného tabulkou.*

	a	b
↔1	-	4
→2	2,3	1
3	3	1
←4	3	3,4

Úkol 67 *K uvedené regulární gramatice sestrojte ekvivalentní nedeterministický konečný automat.*

$S \longrightarrow abS \,|\, bbaA \,|\, \varepsilon$
$A \longrightarrow abA \,|\, bB$
$B \longrightarrow acS \,|\, bC \,|\, \varepsilon$
$C \longrightarrow aC \,|\, bA$

Kapitola 19

Turingovy stroje

Cíl kapitoly:
Pochopení Turingových strojů jako reprezentanta nejobecnějších algoritmických prostředků k popisu jazyků.

Studijní cíle: Pochopení Turingových strojů jako reprezentanta nejobecnějších algoritmických prostředků k popisu jazyků.

Víme už, že jazyky třídy $\mathcal{L}_3$ jsou charakterizovány konečnými automaty, jazyky třídy $\mathcal{L}_2$

(nedeterministickými) zásobníkovými automaty. Nyní uvedeme výpočetní model, který se ukáže být ekvivalentní obecným gramatikám, a sice tzv. Turingovy stroje.

Turingův stroj je podobný konečnému automatu, rozdíl je v tom že páska, na níž je na začátku zapsáno vstupní slovo (ostatní buňky jsou prázdné, tj. je v nich zapsán speciální prázdný znak), je oboustranně nekonečná, hlava spojená s konečnou řídicí jednotkou se může pohybovat po pásce oběma směry a je nejen čtecí, ale i *zapisovací* – symboly v buňkách pásky je tedy možné přepisovat, a to i jinými než vstupními symboly. Formalizujme nyní pojem Turingova stroje, jeho výpočtu a jazyka jím přijímaného.

Definice 19.0.23 ***Turingův stroj****, zkráceně TS, M je určen následujícími parametry (přesněji řečeno, je to uspořádaná šestice)* $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$*, kde*

- Q *je konečná neprázdná množina* stavů
- Γ *je konečná neprázdná množina* (páskových) symbolů
- $\Sigma \subseteq \Gamma$, $\Sigma \neq \emptyset$ *je množina* vstupních symbolů *(tzv.* ***vstupní abeceda****)*
- $q_0 \in Q$ *je* počáteční stav*,*
- $F \subseteq Q$ *je množina* koncových stavů
- $\delta : (Q - F) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ *je přechodová funkce*

Předpokládáme, že v $\Gamma - \Sigma$ *je vždy obsažen speciální prvek* $\Box$ *označující* prázdný znak.

Konfigurací Turingova stroje M *rozumíme libovolné slovo tvaru* uqv*, kde* $u, v \in \Gamma^*$ *a* $q \in Q$. Konfigurace uqv *je* počáteční*, jestliže* $u = \varepsilon$*,* $q = q_0$ *a* $v \in \Sigma^*$ *;* uqv *je* koncová konfigurace*, jestliže* $q \in F$.

Konfigurace uqv *a* $\Box^i uqv \Box^j$ *(*$i, j \geq 0$*) ztotožňujeme, speciálně tedy konfiguraci* qv *ztotožňujeme s konfigurací* $\#qv$*, podobně* uq *s* $uq\#$.

Konfigurace $K = uaqbv$*, kde* $u, v \in \Gamma^*$*,* $a, b \in \Gamma$ vede v jednom kroku ke konfiguraci K'*, příslušnou relaci označujeme* $\vdash_M$ *nebo jen* $\vdash$ *(píšeme tedy* $K \vdash K'$*) právě když platí jedna z těchto možností:*

- $\delta(q, b) = (q', b', 0)$ *a* $K' = uaq'b'v$*,*
- $\delta(q, b) = (q', b', +1)$ *a* $K' = uab'q'v$*,*
- $\delta(q, b) = (q', b', -1)$ *a* $K' = uq'ab'v$.

Relace $\vdash^*$ *je reflexivním a tranzitivním uzávěrem relace* $\vdash$.

Slovo $u \in \Sigma^*$ je přijímáno *TS* M*, jestliže* $q_0 u \vdash^* K$ *pro nějakou koncovou konfiguraci* K.

Jazykem přijímaným *TS* M *rozumíme jazyk* $L(M) = \{w \in \Sigma^* \mid w$ *je přijímáno* $M\}$.

Všimněte si, že výpočet pro danou počáteční konfiguraci je jediný (stroj je deterministický), nemusí však skončit ! Podle naší definice slovo není přijímáno strojem M právě tehdy, když je výpočet nad ním nekonečný (na rozdíl od KA či ZA, koncový stav je u TS 'opravdu' koncový, výpočet dále nemůže pokračovat; pro nekoncový stav je další krok vždy definován).

Čtenáře asi napadne varianta definice, která by vyžadovala, aby každý výpočet TS vždy skončil, a sice buď ve speciálním *přijímajícím stavu* (vstupní slovo přijato) nebo *zamítajícím stavu* (vstupní slovo zamítnuto). Jazyky, které je možné Tur. stroji takto *rozhodovat* (nazývají se také *rekurzivní* nebo *rozhodnutelné jazyky*) jsou ovšem vlastní podtřídou jazyků přijímaných Tur. stroji (které jsou také nazývány *rekurzivně spočetné* či *částečně rozhodnutelné jazyky*). Důkaz bude proveden v kursu o vyčíslitelnosti a složitosti. (Tam se mj. také ukáže, že neexistuje algoritmus, který by pro zadaný Turingův stroj zjistil, zda (každý) jeho výpočet skončí.)

Turingovy stroje patří mezi tzv. *univerzální výpočetní modely*, tj. ty, které jsou schopny realizovat jakýkoli algoritmus (to je obsahem tzv. Church-Turingovy teze, o níž bude podrobněji pojednáno v kursu o vyčíslitelnosti a složitosti). Mj. to znamená, že obohacení uvedeného modelu např. o další pásky, další (čtecí a zapisovací) hlavy, nebo přidání programových konstrukcí jako např. *if ... then*, *while ... do* apod. vede sice k jednoduššímu zápisu algoritmů, ale nikoli k rozšíření třídy přijímaných jazyků (či obecněji třídy vyčíslitelných [realizovatelných] funkcí); standardní model Turingova stroje dokáže všechny tyto rozšířené modely simulovat. Podrobněji bude o této problematice pojednáno v kursu o vyčísltelnosti a složitosti, teď si jen stručně všimneme rozšíření vzniklého využitím nedeterminismu.

Úkol. Využitím zkušeností s konečnými a zásobníkovými automaty nadefinujte pojem *nedeterministických Turingových strojů* a jazyků jimi přijímaných.

Věta 19.0.24 *Třída jazyků přijímaných (determimistickými) Turingovými stroji se rovná třídě jazyků přijímaných nedetermimistickými Turingovými stroji.*

Důkaz: Idea:

Pro daný nedeterministický TS M lze snadno sestavit algoritmus, který pro zadané vstupní slovo w systematicky zkoumá všechny výpočty TS M délky 1, pak všechny výpočty délky 2, pak všechny výpočty délky 3 atd. (jinak řečeno: strom možných výpočtů M nad w je prohledáván ‘do šířky’). Pro zadané w uvedený algoritmus nutně objeví přijímající výpočet stroje M nad w, jestliže takový existuje; v takovém případě algoritmus skončí (a slovo w přijme), jinak běží donekonečna. Algoritmus pak stačí ‘naprogramovat’ jako deterministický Turingův stroj. □

Kapitola 20

Další poznámky ke vztahu automatů a gramatik

Cíl kapitoly:
Získání přehledu o dalších vztazích mezi automaty a gramatikami.

Studijní cíle: Získání přehledu o dalších vztazích mezi automaty a gramatikami.

Pojem nedeterministického Turingova stroje lze např. využít pro očividný důkaz jednoho směru následující věty:

Věta 20.0.25 *Jazyky přijímané Turingovými stroji jsou právě jazyky typu 0.*

Úkol 68 *Který směr je ten očividný ? Vysvětlete proč.*

Idea důkazu druhého směru: relace $\vdash_M$ je de facto relací přepisování mezi slovy jisté konečné abecedy, určené konečně mnoha pravidly. K M lze tedy sestavit obecnou gramatiku, která je schopna, zhruba řečeno, vygenerovat slovo wXw (pro lib. w), v 'pravé kopii' pak odsimulovat výpočet stroje M nad w a v případě, že tento skončí, smaže se symbol X se vším napravo.

———

Pro úplnost dodejme, že kontextové jazyky (jazyky typu 1) jsou charakterizovány tzv. ***lineárně omezenými automaty***, LBA (linear bounded automata), tj Turingovými stroji, které používají jen úsek pásky, v němž je zapsáno vstupní slovo. (Je možné si představit, že vstupní slovo je ohraničeno spec. symboly, levou a pravou zarážkou; ty nemohou být přepsány a z levé (pravé) zarážky je možný jen pohyb doprava (doleva). 'Základní' verze automatu je ovšem *nedeterministická*; problém, zda DLBA (deterministické LBA) přijímají tytéž jazyky jako LBA je dlouhodobě otevřený.

Věta 20.0.26 *Jazyk je kontextový (tj. typu 1) právě tehdy, když je přijímán nějakým LBA.*

Jeden směr je opět přímočarý (zjistíte, který ?), druhý je více techničtější.

———

Zmínili jsme již, že inkluze v Tvrzení 17.0.20 jsou vlastní. Víme např., že jazyk $\{a^n b^n \mid n \geq 0\}$ je typu 2 ale nikoli 3, a také, že $\{a^n b^n c^n \mid n \geq 0\}$ není typu 2 – je ovšem zřejmé, že je typu 1. Existenci jazyka v $\mathcal{L}_0 - \mathcal{L}_1$ lze ukázat např. diagonalizační metodou, o níž pojednáme v kursu o vyčíslitelnosti a složitosti.

———————

Všimněme si ještě, že u TS si lze pásku vlevo od hlavy a pásku vpravo od hlavy představit jako dva zásobníky. Vyvoďte z toho, že model "zásobníkový automat s dvěma zásobníky" (nadefinujte jej formálně !) přijímá tytéž jazyky jako Turingovy stroje.

———————

Část II

Úvod do teorie vyčíslitelnosti a složitosti

Úvod, cíle kursu, literatura

Kurs je určitým úvodem do partií teorie algoritmů, především partií, které se nazývají vyčíslitelnost (computability) a složitost (complexity). Základním úkolem teorie vyčíslitelnosti je charakterizovat ty problémy, jež jsou algoritmicky řešitelné (vyčíslitelné, vypočitatelné). Teorie složitosti se pak zabývá především otázkou, jak jsou příslušné výpočty, potažmo algoritmy, složité (z hlediska nároku na čas, paměť apod.).

Konkrétnější představu o náplni kursu si lze udělat z obsahu. Např. si tak lze všimnout, že kurs se dotkne mj. též obecných metod návrhu algoritmů, což je důležité téma, které do rámce vyčíslitelnosti a složitosti spadá jen nepřímo. (V obsahu je pro lepší orientaci rovněž uvedeno předpokládané rozdělení probírané látky do jednotlivých přednášek kursu.)

Cíle kursu

Absolvent má získat hlubší (teoretický) základ pro pojmy složitosti algoritmů a problémů, i pro jejich analýzu (odhady) u konkrétních případů – mj. u algoritmů navrhovaných obecnými metodami (prohledávání, “rozděl a panuj”, ‘greedy’ přístup, dynamické programování). Dále má zvládnout klasifikaci typických (praktických) problémů vzhledem k základním třídám složitosti (PTIME, NPTIME, PSPACE, ...), včetně prokázání případné nerozhodnutelnosti problému. Rovněž má získat představu, podloženou pochopením konkrétních příkladů, o problematice aproximačních, pravděpodobnostních, paralelních a distribuovaných algoritmů.

Poznámka. U každé kapitoly textu jsou vedle stručného obsahu uvedeny studijní cíle. Ty jsou uvedeny jen orientačně, nelze je chápat tak, že by přesně vyjadřovaly, co má (a co nemusí) studující zvládnout.

Literatura

Základní studijní pomůckou ke kursu je předkládaný pracovní text a je jej možné využít i jako základ samostudia. Text je ovšem na mnoha místech velmi stručný; jeho studium by mělo být významně ulehčeno *aktivní* účastí na přednáškách a cvičeních (kde jsou studované pojmy a metody názorněji ilustrovány na příkladech, myšlenky textu jsou dále rozvedeny a vysvětleny, apod.).

Poznámka. Program cvičení bude průběžně zvřejňován na Webu

http://www.cs.vsb.cz/jancar/VYCSLOZ/vycsloz.htm;

příklady a referáty na nich probrané jsou integrální součástí kursu !

Velmi vhodné by samozřejmě bylo, aby si posluchači znalost a pochopení probíraných témat upevnili studiem další odborné literaury. Uveďme alespoň některé knihy v češtině či slovenštině pokrývající části přednášené látky. Partie z teorie složitosti (včetně konkrétních algoritmů a problémů) najdeme např. v knihách [Kuč83], [Mor84] (zejména první z nich vřele doporučuji !). Další partie (jako Turingovy stroje, problém zastavení, zavedení výpočtové složitosti aj.) jsou pokryty např. v knihách [Chy84], [HU78]. Dále např. 1.kapitola knihy [Man81] je stručně věnována teorii vyčíslitelnosti.

Podstatně bohatší je literatura v angličtině. Z ní je možné doporučit např. [Sip97], [Har93], [CLR90], [HU79]. Samozřejmě spoustu relevantního materiálu (různé kvality) lze také získat “surfováním” na Webu.

Poznámka. Jako vždy, člověk se nejvíce naučí vlastními *aktivními* pokusy o řešení příslušných problémů – souběžně s “pasivním” studiem. Teprve (a jen) tehdy může pochopit, o co vlastně jde, a ocenit nastudované řešení.

Kapitola 21

Složitost algoritmu. Model RAM.

Stručný obsah: Pojem složitosti (časové či paměťové náročnosti) algoritmu – vztažen k referenčnímu abstraktnímu modelu počítače a definován jako funkce velikosti vstupu. Model (referenčního) stroje RAM (tj. počítače s okamžitým přístupem k libovolné buňce paměti); jednotková vs. logaritmická míra v definici složitosti. Jako příklad algoritmy *bubblesort* a *heapsort*; pro ilustraci návrh a analýza časové složitosti RAMu pro *bubblesort*.

Cíl kapitoly:
(Pochopit a) umět vysvětlit pojem složitosti algoritmu včetně role referenčního modelu počítače. Schopnost porozumění a analýzy složitosti (jednoduchých) programů pro RAM. Umět vysvětlit roli jednotkové a logaritmické míry v definici složitosti. Pochopení programování na úrovni RAMu, které dostačuje pro analýzu algoritmů zapsaných jazyky ‘vyšší úrovně’.

Studijní cíle: (Pochopit a) umět vysvětlit pojem složitosti algoritmu včetně role referenčního modelu počítače. Schopnost porozumění a analýzy složitosti (jednoduchých) programů pro RAM. Umět vysvětlit roli jednotkové a logaritmické míry v definici složitosti. Pochopení programování na úrovni RAMu, které dostačuje pro analýzu algoritmů zapsaných jazyky ‘vyšší úrovně’.

Čtenář už jistě má určitou představu o tom, že např. pro jeden a tentýž problém existují různé algoritmy, které ho řeší; takové algoritmy (a koneckonců nejen ty řešící stejný problém) je možné vzájemně srovnávat z různých hledisek. Pro konkrétnost si připomeňme problém třídění (sorting; v češtině by zde byl vhodnější termín ‘seřazování’):

> *Název (označení) problému*: *Třídění čísel*
>
> *(Očekávaný) vstup*: konečná posloupnost přirozených čísel
> *(Požadovaný) výstup*: posloupnost týchž čísel uspořádaná podle velikosti ve vzestupném pořadí.

V učebnicích se často mezi prvními algoritmy řešícími daný problém uvádí tzv. *bubblesort*, jehož základní myšlenka se dá vyjádřit takto:

Projdi posloupnost zleva doprava, přičemž prohazuješ sousední dvojice čísel, pokud v nich větší číslo předchází menšímu.

Tento postup procházení posloupnosti opakuj, dokud nedostaneš kompletně uspořádanou posloupnost.

Poznámka. Poznamenejme, že *bubblesort* se v učebnicích vyskytuje spíše jako odstrašující příklad (jelikož lze snadno navrhnout podstatně lepší algoritmy, jak o tom také budeme hovořit dále). My zde tento algoritmus také uvádíme jen pro jeho jednoduchost a ilustraci dále zkoumaných pojmů, nikoliv snad pro jeho 'hodnotu'.

Zpřesněné vyjádření algoritmu programátorským pseudokódem by mohlo vypadat takto:

{členy vstupní posloupnosti jsou označeny $A[1], A[2], \ldots A[n]$}
while Nesetříděno **do**
  for $i := 1$ **to** $n - 1$ **do**
    if $A[i] > A[i+1]$ **then** prohoď $A[i]$ a $A[i+1]$

(V této chvíli se náš návrh nezabývá tím, jak se přiřazuje do booleovské proměnné 'Nesetříděno'.)

Po přesvědčení se, že algoritmus je korektní – tj. vždy skončí a výsledná posloupnost je uspořádaná (jak byste to dokázali ?), je možné využít většího porozumění předepsaného procesu třídění a upravit (a zpřesnit) algoritmus následovně:

Bubblesort – progr. verze

{členy vstupní posloupnosti nejprve načteme do pole A}
{předp., že členy jsou nenulové a hodnota 0 označuje konec vstupu}
$n := 0$;
repeat
  $n := n + 1$; read($A[n]$)
until $A[n] = 0$;
$n := n - 1$;
{v n je uložen počet členů vstupní posloupnosti}
for $j := 1$ **to** $n - 1$ **do**
  for $i := 1$ **to** $n - j$ **do**
    if $A[i] > A[i+1]$ **then** ($pom := A[i]$; $A[i] := A[i+1]$; $A[i+1] := pom$);
{výsledná seřazená posloupnost se vypíše}
for $i := 1$ **to** n **do**
  write($A[i]$)

Že tento algoritmus (to je výpočetní proces jím předepsaný) pro každou (konečnou) vstupní posloupnost skončí, je zde zřejmé (proč ?); přesvědčte se, proč je výsledná posloupnost určitě uspořádaná.

Jak jsme už zmínili, *bubblesort* zdaleka není nejlepším algoritmem pro daný problém třídění. Připomeňme si teď metodu (čili algoritmus) *heapsort*. K tomu je potřebné si připomenout datovou strukturu *halda* (heap), tj. (speciální) binární strom: každý vrchol v je ohodnocen číslem $n(v)$ (prvkem tříděné posloupnosti), přičemž je-li v' následníkem v, pak $n(v) \leq n(v')$. Zařazení dalšího prvku do haldy i výběr nejmenšího prvku z haldy se dají snadno realizovat x kroky, kde x je hloubkou haldy (stromu); při počtu vrcholů n je tedy přibližně $x = \log n$.

Poznámka. V informatice při neuvedení základu $\log n$ většinou myslíme dvojkový logaritmus $\log_2 n$. Později vysvětlíme, proč je základ logaritmu pro účely analýzy algoritmů v zásadě nepodstatný.

Důležitou myšlenkou algoritmu *heapsort* je rovněž efektivní způsob reprezentace haldy jednorozměrným polem.

Vše se dá vyčíst z dále uvedeného pseudokódu; je ovšem velmi žádoucí, ať si čtenář běh algoritmu ilustruje (připomene) na rozumně zvoleném malém příkladu.

Heapsort – progr. verze

{proměnná H představuje haldu (var H: array[1..] of integer)}
{kon udává aktuální koncový index haldy}
kon:=0; {halda je prázdná}
read($clen$);
while $clen \neq 0$ **do**
  Zarad-do-haldy($clen$); read($clen$);
while $kon > 0$ {halda není prázdná} **do**
  Vydej-min-z-haldy($clen$); write($clen$)

procedure Zarad-do-haldy(k);
$kon := kon + 1; H[kon] := k;$
$p := kon;$
while ($p > 1$ and $H[p$ div $2] > H[p]$) **do**
  prohod $H[p$ div $2]$ a $H[p]$; $p := p$ div 2;

procedure Vydej-min-z-haldy(var min);
$min := H[1];$
if $kon > 1$ **then** $H[1] := H[kon];$
$kon := kon - 1;$
$p := 1;$
while ($2 * p + 1 \leq kon$ and ($H[p] > H[2 * p]$ or $H[p] > H[2 * p + 1]$)) **do**
  if $H[2 * p] \leq H[2 * p + 1]$
  then (prohod $H[p]$ a $H[2 * p]$; $p := 2 * p$)
  else (prohod $H[p]$ a $H[2 * p + 1]$; $p := 2 * p + 1$);
if ($2 * p = kon$ and $H[p] > H[2 * p]$)
then prohod $H[p]$ a $H[2 * p]$

Oba algoritmy (*bubblesort* a *heapsort*) řeší náš problém třídění, přičemž *heapsort* je očividně složitější z hlediska návrhu, zápisu i porozumění (ověření správnosti). V čem je tedy *heapsort* lepší ?

Zkušený čtenář asi odpoví, že *heapsort* má menší časovou složitost (náročnost) než *bubblesort*. Označujeme takto fakt, že (hodně neformálně řečeno) *heapsort* ‘běhá rychleji’ než *bubblesort*. Určitým způsobem se o tom můžeme přesvědčit, naprogramujeme-li obě metody v námi oblíbeném programovacím jazyku a srovnáme běh obou programů na počítači na sadě instancí (tj. povolených vstupů) problému třídění – pro každou instanci měříme čas, který na její zpracování jednotlivé programy spotřebují.

Doufejme, že čtenář není natolik ‘prakticky orientovaný’, že mu výše zmíněný test stačí, ale že by rád více porozuměl, proč tomu tak je, a své poznání opřel o solidnější základ. (Neměl by stačit argument ‘Protože jsem *bubblesort* a *heapsort* naprogramoval v Céčku a na mnou zvolených deseti příkladech běžel *heapsort* na PC-čku vždycky rychleji, je *heapsort* lepší’).

Chtělo by to definovat pro každý algoritmus nějakou kvantitativní charakteristiku, nazvěme ji časová složitost (či jen složitost, když se ‘časová’ rozumí samo sebou), podle které pak bude možné různé algoritmy srovnávat. Složitost ovšem musí zachycovat ‘dobu běhu’ globálně – tj. pro všechny přípustné vstupy, nejen pro vybranou sadu testovacích případů.

Nabízí se zmíněnou časovou složitost algoritmu prostě definovat jako funkci (zobrazení), která každému (přípustnému) vstupu přiřazuje ‘dobu běhu’ algoritmu na onen vstup. To má ovšem několik “vad na kráse” (např. pak není jasné, jak srovnávat rychlost algoritmů pracujících s různými vstupy). Jako vhodnější (jednodušší a přitom postačující) se ukazuje definovat

složitost jako *funkci velikosti vstupu*.

Poznámka. Složitost (jakožto funkce) má většinou nekonečný definiční obor (např. i v našem problému třídění délku vstupní posloupnosti nijak neomezujeme); nelze ji tedy zadat výčtem hodnot, ale je nutno hledat nějaký konečný popis (např. algebraické vyjaádření jako $3n^2 - 4n + 3$ apod.).

Vstupů se stejnou velikostí n ovšem může být hodně a výpočty pro tyto jednotlivé vstupy mohou trvat různou ‘dobu’. Co je pak hodnotou složitosti (tj. zmíněné funkce) pro n ? Pro praktické účely často stačí přístup

podle nejhoršího možného případu (worst-case),

kdy dané velikosti n přiřazuje ona funkce maximum z ‘dob běhu’ algoritmu na všech vstupech velikosti n.

Musí se samozřejmě vyjasnit několik věcí – např. co je to *velikost vstupu*. Později se k tomu ještě vrátíme, teď poznamenejme, že u našeho problému třídění je většinou postačující velikost vstupu definovat jako počet členů zadané posloupnosti (později dodáme: pokud je velikost čísel=členů omezená).

Poznámka. Obecně použitelné řešení je chápat velikost daného vstupu jako počet bitů, které vstup zabírá (při “přirozeném” zakódování). Často lze však dostatečné výsledky analýzy složitosti dosáhnout bez nutnosti uvažovat tuto “nízkou” úroveň (která může přidávat zbytečné technické komplikace).

Jistě jste si povšimli jiného velmi slabého místa v uvedených definicích: užívání pojmu ‘doba běhu’. Vždyť např. při různých implementacích (v různých programovacích jazycích, na různých počítačích apod.) budou ‘doby běhu’ jednoho a téhož algoritmu pro jeden a tentýž vstup různé ! Jako nejvhodnější exaktní definování pojmu ‘doba běhu’ se ukazuje volba nějakého (abstraktního) modelu počítače, ke kterému se pak budeme odkazovat jako k jakémusi *referenčnímu modelu*; dobu běhu, tj. ‘dobu trvání výpočtu’ (neboli délku výpočtu), pak budeme měřit počtem provedených (elementárních) instrukcí.

Modelů sloužících těmto účelům byla navržena celá řada (později se k tomu ještě dostaneme). My se teď seznámíme se strojem RAM (Random Access Machine), který je česky někdy zván *‘počítač s libovolným přístupem’*; název není zcela výstižný, znamená prostě to, že přístup k libovolné buňce paměti je okamžitý (či asi vhodněji: doba přístupu k libovolné buňce paměti je konstantní).

Abstraktní stroj RAM (Random Access Machine)

Stroj RAM se skládá z programové jednotky, (operační) paměti, vstupní jednotky a výstupní jednotky. (Pro ilustraci je vhodný obrázek, čtenář je vyzýván, ať si jej nakreslí.)

V programové jednotce je zapsán program, tvořený konečnou posloupností instrukcí (příkazů), které budou popsány dále. Dále je v ní programový registr ukazující, která instrukce má být v daném okamžiku prováděna (programový registr prostě obsahuje pořadové číslo příslušné instrukce).

Paměť je tvořena potenciálně nekonečným polem paměťových buněk očíslovaných (adresovaných) přirozenými čísly $0, 1, 2, \ldots$; každá buňka může obsahovat *libovolně velké celé číslo*. Buňky s adresou 0 a 1 mají zvláštní postavení a nazývají se **pracovní registr** (buňka 0) a **indexový registr** (buňka 1).

Vstupní jednotka se skládá z (potenciálně nekonečné) pásky rozdělené na políčka, z nichž každé může obsahovat *libovolně velké celé číslo*, a z hlavy, která v každém okamžiku snímá jedno z políček pásky. Základní krok v činnosti hlavy spočívá v přečtení obsahu snímaného políčka a posunutí doprava o jedno políčko.

Podobným způsobem je konstruována výstupní jednotka, jejíž hlava je však schopna pouze zapisovat do políček výstupní pásky a posouvat se zleva doprava o jedno políčko.

V počáteční konfiguraci (tj. na začátku výpočtu) je na určitém počátečním úseku vstupní pásky uložen vstup (prvních n políček, pro určité n, obsahuje (vstupní) čísla $c_1, c_2, \ldots, c_n$; vstupní hlava snímá první buňku s číslem c_1). Zbylá políčka vstupní pásky, všechna políčka výstupní pásky a všechny paměťové buňky obsahují číslo 0; programový registr ukazuje na první instrukci programu (tj. obsahuje číslo 1).

Konfigurace (tj. stav výpočtu) se mění krok za krokem prováděním předepsaných instrukcí. (Je možné si představit, že RAM má ještě jakési výkonné jednotky, jako např. aritmetickou jednotku, umožňující provádění příslušných operací).

Nyní uvedeme instrukce stroje RAM, z nichž lze sestavovat program (pro názornost se čtenář může podívat na konkrétní RAM-program – v prvním sloupci na ‘obrázku’ o několik stran dále). Tvary ‘operandů’ instrukcí a jejich příslušné hodnoty jsou patrny z následující tabulky (i je zápis přirozeného čísla). Za touto tabulkou pak již následuje přehled instrukcí, logicky rozdělených do několika skupin. (Označení ‘návěští’ zde představuje přirozené číslo, udávající pořadové číslo instrukce, která bude prováděna jako následující, dojde-li ke skoku.)

Tvary operandů

tvar	hodnota operandu
$=i$	přímo číslo udané zápisem i
i	číslo obsažené v buňce s adresou i
$*i$	číslo v buňce s adresou $i+j$, kde j je aktuální obsah index. registru

Instrukce vstupu a výstupu (jsou bez operandu):

zápis	význam
READ	do prac. registru se uloží číslo, které je v políčku snímaném vstupní hlavou, a vstupní hlava se posune o jedno políčko doprava
WRITE	výstupní hlava zapíše do snímaného políčka výstupní pásky obsah prac. registru a posune se o jedno políčko doprava

Instrukce přesunu v paměti:

zápis	význam
LOAD operand	hodnotou operandu se přepíše obsah prac. registru
STORE operand	hodnota operandu se přepíše obsahem prac. registru (zde se nepřipouští operand tvaru $=i$)

Instrukce aritmetických operací:

zápis	význam
ADD operand	číslo v prac. registru se zvýší o hodnotu operandu (tedy přičte se k němu hodnota operandu)
SUB operand	od čísla v prac. registru se odečte hodnota operandu
MUL operand	číslo v prac. registru se vynásobí hodnotou operandu
DIV operand	číslo v prac. registru se ‘celočíselně’ vydělí hodnotou operandu (do prac. registru se uloží výsledek příslušného celočíselného dělení

Instrukce skoku:

zápis	význam
JUMP návěští	výpočet bude pokračovat instrukcí určenou návěštím
JZERO návěští	je-li obsahem prac. registru číslo 0, bude výpočet pokračovat instrukcí určenou návěštím; v opačném případě bude pokračovat následující instrukcí
JGTZ návěští	je-li číslo v prac. registru kladné, bude výpočet pokračovat instrukcí určenou návěštím; v opačném případě bude pokračovat následující instrukcí

Instrukce zastavení

zápis	význam
HALT	výpočet je ukončen (‘regulérně’ zastaven)

Jak lze očekávat, provedení instrukce zpravidla také znamená zvýšení programového čítače o jedničku (výpočet pokračuje prováděním bezprostředně následující instrukce); výjimkou jsou případy, kdy dojde ke skoku (a také případ instrukce HALT).

Předpokládáme, že kdykoli by mělo při běhu dojít k nedefinované akci (dělení nulou, progr. čítač ukazuje 'mimo program', adresa při použití operandu $*i$ vyjde záporná), výpočet se ('neregulérně') zastaví.

Značení. $\mathcal{N}$ v celém textu označuje množinu všech přirozených čísel $\{0, 1, 2, \ldots\}$.

Nyní můžeme pro RAMy (RAM-programy, chcete-li) exaktně definovat časovou a paměťovou složitost (jakožto funkce velikosti vstupu). Přitom se omezujeme jen na RAMy, které se pro každý vstup zastaví (provedou HALT, případně skončí 'neregulérně'); nekonečnou časovou ani paměťovou složitost neuvažujeme.

Definice 21.0.27 Velikostí vstupu *stroje RAM rozumíme počet buněk (vstupní pásky), které daný vstup zabírá.*
Délka výpočtu *RAM-stroje* M *pro konkrétní vstup se definuje jako počet provedení instrukcí, které* M *pro daný vstup vykoná, než se zastaví.*
Časovou složitostí *RAM-stroje* M *rozumíme funkci* $T_M : \mathcal{N} \longrightarrow \mathcal{N}$*, kde* $T_M(n)$ *znamená délku výpočtu* M *nad vstupem velikosti* n *v nejhorším případě; tedy* $T_M(n) = max\{\, k \mid k$ *je délka výpočtu* M *nad (nějakým) vstupem velikosti* $n\,\}$*.*

Definice 21.0.28 Velikostí paměti *RAM-stroje* M *(potřebné při výpočtu) pro konkrétní vstup rozumíme číslo* $p+1$*, kde* p *je maximum z adres buněk, jež jsou během výpočtu (nad daným vstupem) navštíveny.*
Paměťovou složitostí *(nebo též* prostorovou složitostí*) RAM-stroje* M *rozumíme funkci* $S_M : \mathcal{N} \longrightarrow \mathcal{N}$*, kde* $S_M(n)$ *znamená velikost potřebné paměti při výpočtu* M *nad vstupem velikosti* n *v nejhorším případě; tedy* $S_M(n) = max\{\, k \mid k$ *je velikost paměti potřebné při výpočtu* M *nad (nějakým) vstupem velikosti* $n\,\}$*.*

Pozorný čtenář si už možná všiml podezřelého místa v uvedených definicích: nijak neomezujeme velikost čísla, které je možno uložit do jednotlivé buňky paměti ! Přitom velikost paměti zabrané danou buňkou (a čas elementární operace pracující s danou buňkou) chápeme jako jednotku, jinými slovy uvažujeme tzv.

jednotkovou (či uniformní) míru.

Takto však lze 'šikovně šetřit paměť' kódováním celé série čísel (např. matice čísel) číslem jediným. Podobnými 'triky' se dá šetřit i čas výpočtu (počet provedených instrukcí) a získané výsledky pak neodpovídají realitě.

Pokud podobný efekt hrozí, uvažuje se místo jednotkové míry

míra logaritmická:

je-li v buňce uloženo číslo z, počítá se, že je takto zabrána paměť velikosti $\lceil \log_2(|z| + 1) + 1 \rceil$ (počet bitů potřebných k zapsání z; znaky $\lceil\ \rceil$ znamenají zaokrouhlení nahoru). Podobně i cena (čas) provedení jedné instrukce není 1, ale je úměrná velikosti čísel, se kterými se při provádění instrukce operuje.

V následující analýze algoritmů pro problém třídění budeme užívat jednotkovou míru. V tom případě ovšem naše analýza může dávat realistické výsledky jen tehdy, když je velikost tříděných čísel (předem) omezená (čísla mají omezený počet cifer); přesněji řečeno, když je rozumné předpokládat, že operace jako načtení čísla, porovnání dvou čísel apod. trvají konstantní čas.

Zkusme nyní naprogramovat algoritmus *bubblesort* pro RAM a analyzovat jeho časovou složitost. Výsledkem vcelku přímočarého 'přeložení' dříve uvedeného pseudokódu (*Bubblesort – progr. verze*) do 'jazyka' RAM může být níže uvedený program (předpokládáme v něm, že vstupní posloupnost není prázdná).

Vlastní RAM-program, tvořený posloupností 69 instrukcí je uveden v prvním 'sloupci'. V druhém sloupci je tentýž program v poněkud srozumitelnější podobě – užívá symbolických návěští a symbolického adresování (N=2, J=3, HM=4, I=5, IPJ=6, POM=7, X=8, A=8; člen $A[1]$ bude uložen v buňce 9, $A[2]$ v buňce 10, $A[3]$ v buňce 11 atd.). Třetí sloupec obsahuje komentáře, ze kterých by mělo být patrno, že se skutečně jedná o 'překlad' uvedené verze *bubblesortu*. (Připomeňme, že všechny paměťové buňky mají na začátku hodnotu 0.)

Bubblesort, RAM-verze

```
01 LOAD   =1   |          LOAD   =1     {do indexreg. se vloží 1, tj.
02 STORE  1    |          STORE  1      {první volný index pole A
03 READ        |  Cykl-vst: READ        {načtení dalšího vstupu
04 JZERO  10   |          JZERO  Kon-vst{0 znamená konec vstupu
05 STORE  *8   |          STORE  *A     {A[indexreg]:=vstup    }
06 LOAD   1    |          LOAD   1      {                      }
07 ADD    =1   |          ADD    =1     {indexreg. se zvýší o 1}
08 STORE  1    |          STORE  1      {                      }
09 JUMP   3    |          JUMP   Cykl-vst{                      }
10 LOAD   1    |  Kon-vst: LOAD  1      {                      }
11 SUB    =1   |          SUB    =1     {                      }
12 STORE  2    |          STORE  N      {N obsah. počet vst. čísel
13 LOAD   3    |  Cykl-1:  LOAD  J      {                      }
14 ADD    =1   |          ADD    =1     {J:=J+1                }
15 STORE  3    |          STORE  J      {                      }
16 LOAD   2    |          LOAD   N      {                      }
17 SUB    3    |          SUB    J      {                      }
18 JZERO  58   |          JZERO  Vystup {skok při J=N          }
19 ADD    =1   |          ADD    =1     {                      }
20 STORE  4    |          STORE  HM     {HM (hor. mez) = N–J+1
21 LOAD   =0   |          LOAD   =0     {                      }
22 STORE  5    |          STORE  I      {I:=0                  }
23 LOAD   5    |  Cykl-2:  LOAD  I      {                      }
24 ADD    =1   |          ADD    =1     {I:=I+1                }
25 STORE  5    |          STORE  I      {                      }
26 ADD    =1   |          ADD    =1     {                      }
27 STORE  6    |          STORE  IPJ    {IPJ=I+1               }
28 LOAD   4    |          LOAD   HM     {                      }
29 SUB    5    |          SUB    I      {                      }
30 JZERO  13   |          JZERO  Cykl-1 {skok při I=HM         }
31 LOAD   5    |          LOAD   I      {                      }
32 STORE  1    |          STORE  1      {                      }
33 LOAD   *8   |          LOAD   *A     {                      }
```

```
34 STORE 8  |           STORE X    {X:=A[I]                }
35 LOAD  6  |           LOAD  IPJ  {                       }
36 STORE 1  |           STORE 1    {                       }
37 LOAD  8  |           LOAD  X    {                       }
38 SUB   *8 |           SUB   *A   {                       }
39 JGTZ  41 |           JGTZ  Prohod{skok při X>A[I+1]     }
40 JUMP  23 |           JUMP  Cykl-2{                      }
41 LOAD  5  |  Prohod:  LOAD  I    {                       }
42 STORE 1  |           STORE 1    {                       }
43 LOAD  *8 |           LOAD  *A   {                       }
44 STORE 7  |           STORE POM  {POM:=A[I]              }
45 LOAD  6  |           LOAD  IPJ  {                       }
46 STORE 1  |           STORE 1    {                       }
47 LOAD  *8 |           LOAD  *A   {                       }
48 STORE 8  |           STORE X    {X:=A[I+1]              }
49 LOAD  5  |           LOAD  I    {                       }
50 STORE 1  |           STORE 1    {                       }
51 LOAD  8  |           LOAD  X    {                       }
52 STORE *8 |           STORE *A   {A[I]:=X                }
53 LOAD  6  |           LOAD  IPJ  {                       }
54 STORE 1  |           STORE 1    {                       }
55 LOAD  7  |           LOAD  POM  {                       }
56 STORE *8 |           STORE *A   {A[I+1]:=POM            }
57 JUMP  23 |           JUMP  Cykl-2{                      }
58 LOAD  =2 |  Vystup:  LOAD  =1    {                       }
59 STORE 1  |           STORE 1    {indexreg.:=1           }
60 LOAD  *8 |  Cykl-vys:LOAD  *A   {                       }
61 WRITE    |           WRITE      {write(A[indexreg.])    }
62 LOAD  2  |           LOAD  N    {                       }
63 SUB   1  |           SUB   1    {                       }
64 JZERO 69 |           JZERO Konec{skok při indexreg.=N   }
65 LOAD  1  |           LOAD  1    {                       }
66 ADD   =1 |           ADD   =1   {                       }
67 STORE 1  |           STORE 1    {indexreg. se zvýší o 1 }
68 JUMP  60 |           JUMP  Cykl-vys                     }
69 HALT     |  Konec:   HALT       {                       }
```

Spočtěme nyní, kolik instrukcí bude provedeno při zpracování vstupu velikosti n (v nejhorším možném případě).

První (vstupní) fáze výpočtu, od začátku po první příchod na instrukci s návěštím Cykl-1, zřejmě zabere ‘čas’ (tj. počet provedení instrukcí)

$$T1 = 2 + 7n + 2 + 3 = 7n + 7$$

Podobně třetí (výstupní) fáze, od skoku na Vystup po Konec, zabere zřejmě čas

$$T3 = 2 + 9(n - 1) + 5 + 1 = 9n - 1$$

Více složitější je analyzovat zbylou (prostřední) fázi, od prvního skoku na Cykl-1 po skok na Vystup. Také s přihlédnutím k naší progr. verzi *bubblesortu* není ovšem zase tak obtížné odvodit, že ona prostřední fáze trvá

$$T2 = \left(\sum_{j=1}^{n-1} \left(10 + \left(\sum_{i=1}^{n-j} 34 \right) + 8 \right) \right) + 6$$

Poznamenejme, že vždy předpokládáme ten horší (tj. delší k zpracování) případ, kdy skutečně dojde k prohazování prvků (tj. provádí se ‘podprogram’ Prohod).

Výraz pro $T2$ můžeme upravovat standardní manipulací se sumami např. takto:

$$T2 = 6 + \sum_{j=1}^{n-1} 18 + \sum_{j=1}^{n-1} \sum_{i=1}^{n-j} 34 = 6 + 18(n-1) + \sum_{j=1}^{n-1} 34(n-j) =$$

$$= 18n - 12 + 34 \sum_{j=1}^{n-1} n - 34 \sum_{j=1}^{n-1} j$$

Dosadíme-li za první sumu $n(n-1)$ a za druhou sumu $(1 + 2 + \ldots + n - 1) = (n/2)(n-1)$, odvodíme pak již přímočarými úpravami vztah

$$T2 = 17n^2 + n - 12$$

Celkový čas potřebný pro zpracování vstupu velikosti n je tedy $T1 + T2 + T3 = 17n^2 + 17n - 6$. Označíme-li náš RAM-stroj M a jeho časovou složitost T_M, ukázali jsme tak, že pro každé $n \geq 1$ je

$$T_M(n) = 17n^2 + 17n - 6$$

Kapitola 22

Odhady složitosti; značení O aj.

Stručný obsah: Značení používané v odhadech složitosti (O, Θ, o, Ω, ω) a jeho význam. Ilustrace analýzy průměrného případu na příkladu quicksortu.

Cíl kapitoly:
Umět vysvětlit a používat značení O, Θ, o, Ω, ω užívané v odhadech složitosti. Umět vysvětlit význam a ilustrovat klady a zápory analýzy složitosti v nejhorším možném a průměrném případě.

Studijní cíle: Umět vysvětlit a používat značení O, Θ, o, Ω, ω užívané v odhadech složitosti. Umět vysvětlit význam a ilustrovat klady a zápory analýzy složitosti v nejhorším možném a průměrném případě.

Při analýze časové složitosti *bubblesortu* jsme viděli, že přesné (algebraické) vyjádření funkce T_M není technicky úplně triviální úkol již u velmi jednoduchého programu. U větších a komplikovanějších programů by už takový postup byl nesmírně náročný.

Pro naše účely (srovnávání algoritmů) naštěstí přesné vyjádření časové složitosti není nutné; většinou postačí ‘rozumný’ odhad příslušné funkce T_M. Např. v našem případě jsme T_M vyjádřili ve tvaru $T_M(n) = an^2+bn+c$, kde a, b, c jsou konstanty ($a = 17$, $b = 17$, $c = -6$). Když nám nezáleží na přesné hodnotě oněch konstant, mohli jsme analýzu urychlit (místo přesného počítání jsme konstanty mohli odhadovat shora – s určitou rozumnou rezervou) a dospět tak k (hornímu) odhadu např. $T_M(n) \leq 20n^2 + 50n + 100$.

Všimněme si, že pro získání podobného odhadu jsme *bubblesort* ani nemuseli fyzicky programovat pro RAM – pokud již máme určitou programátorskou zkušenost, dokážeme časovou složitost odhadnout např. již z pseudokódu (promyslete si to !).

Uvědomme si dále, že v našem vyjádření $T_M(n) = an^2 + bn + c$ má ‘největší váhu’ člen an^2. Byť by byla konstanta a ‘hodně menší než’ b (ale samozřejmě kladná), vždy od jistého n výše je hodnota an^2 (čím dál výrazněji) větší než hodnota ‘zbytku’ $bn + c$; exaktněji řečeno

$$\lim_{n\to\infty} \frac{bn + c}{an^2} = 0$$

Značení O, Θ, o, Ω, ω

Ono soustředění se na 'rozhodující člen' a zanedbávání přesných hodnot konstant nás vede k tzv. značení *velké-O*. O námi zjištěné funkci $T_M(n) = 17n^2 + 17n - 6$ pak prostě řekneme $T_M(n) \in O(n^2)$.

Přesněji uvedeme značení velké-O takto:

Definice 22.0.29 *Pro libovolné funkce $f, g : \mathcal{N} \to \mathcal{N}$ řekneme, že $f \in O(g)$, označujeme též $f(n) \in O(g(n))$, právě tehdy, když platí $(\exists k \in \mathcal{N})(\exists n_0 \in \mathcal{N})(\forall n \geq n_0) : f(n) \leq k \cdot g(n)$.*

Je-li $f(n) \in O(g(n))$, říkáme také, že $f(n)$ roste řádově nejvýše jako $g(n)$. $O(g(n))$ tedy slouží jako určitý horní odhad funkce $f(n)$.

Když tedy řekneme, že "bubblesort je v $O(n^2)$" (což rozumíme jako zkratku pro "časová složitost algoritmu *bubblesort* je v $O(n^2)$"), pak není vyloučeno, že jej lze (shora) odhadnout lépe. Ve skutečnosti je ovšem funkce n^2 pro *bubblesort* nejen horním, ale i spodním (řádovým) odhadem (proč ?). K vyjádření podobných faktů se vedle O hodí i další značení (f, g jsou libovolné funkce $f, g : \mathcal{N} \to \mathcal{N}$; pro přehlednost zde opakujeme i definici O):

- $f \in O(g)$, označujeme též $f(n) \in O(g(n))$, právě když platí $(\exists k \in \mathcal{N})(\exists n_0 \in \mathcal{N})(\forall n \geq n_0) : f(n) \leq k \cdot g(n)$
- $f \in \Theta(g)$, nebo $f(n) \in \Theta(g(n))$, znamená, že $f \in O(g)$ a $g \in O(f)$.
- $f \in o(g)$, označujeme též $f(n) \in o(g(n))$, právě když platí $(\forall k \in \mathcal{N})(\exists n_0 \in \mathcal{N})(\forall n \geq n_0) : k \cdot f(n) < g(n)$
- $f \in \Omega(g)$, označujeme též $f(n) \in \Omega(g(n))$, právě když platí $g \in O(f)$, tj. $(\exists k \in \mathcal{N})(\exists n_0 \in \mathcal{N})(\forall n \geq n_0) : k \cdot f(n) \geq g(n)$
- $f \in \omega(g)$, označujeme též $f(n) \in \omega(g(n))$, právě když platí $g \in o(f)$, tj. $(\forall k \in \mathcal{N})(\exists n_0 \in \mathcal{N})(\forall n \geq n_0) : f(n) > k \cdot g(n)$

Značení O, o hrají roli neostrého resp. ostrého horního odhadu, značení Ω, ω roli neostrého resp. ostrého dolního odhadu. (Jakou roli hraje značení Θ ?)

Jak už jsme se zmínili, uvedená značení se využívají např. pro možnost stručného vyjádření při analýze časové složitosti (podobně samozřejmě i u prostorové složitosti) konkrétního algoritmu, resp. příslušného (třeba ani fyzicky nesestrojeného) RAM-stroje M, kdy nám většinou postačí jen určitý odhad (růstu) funkce T_M, odhad, v němž se zanedbávají konstatní faktory.

Poznámka. Místo $f(n) \in O(g(n))$ (podobně pro další značení) se někdy (s vědomím záměrné nepřesnosti) píše $f(n) = O(g(n))$; tato notace je pak výhodnější např. v rovnicích, v nichž se značení O a další vyskytují.
Říkáme pak také např. 'časová složitost algoritmu XY je $O(n^2)$' místo přesnějšího '... je *v* $O(n^2)$'.

K *bubblesortu* můžeme dodat, že je v $\Theta(n^2)$. Potom fakt (který se dá přímočaře ukázat), že *heapsort* je v $O(n \log n)$ (je i v $\Theta(n \log n)$), skutečně prokazuje, že *heapsort* je lepší než *bubblesort* (pro dostatečně velké vstupy).

Poznámka. Čtenář je vyzýván, ať si provede srovnání růstu funkcí n, $n \log n$, n^2, n^3, 2^n, $n!$, n^n apod.
Je potřeba si také ujasnit, že např. (jenom) fakt, že algoritmus A má složitost $\Theta(n^2)$ a algoritmus B má složitost $O(n \log n)$, znamená, že A je zaručeně rychlejší než B jen *asymptoticky*, tj. pro 'dostatečně velké' hodnoty. (Záleží totiž na konstantách skrytých v značení Θ, O atd.)

Nejhorší vs. průměrný případ

Zamysleme se na chvíli nad zvoleným přístupem tzv. *nejhoršího možného případu*. Měli bychom si být alespoň vědomi, že tento přístup nemusí vždy poskytovat směrodatné výsledky pro praxi.

Běžně se tento fakt ilustruje na příkladu dalšího algoritmu třídění, tzv. *quicksortu*. Jeho časová složitost z hlediska nejhoršího možného případu je $\Theta(n^2)$, přitom v praxi je ale rychlejší než např. *heapsort* (který má složitost $\Theta(n \log n)$). Dá se totiž ukázat, že složitost *quicksortu* z hlediska *průměrného případu* je také $\Theta(n \log n)$, přičemž příslušná konstanta je menší než u *heapsortu*.

Poznámka. Jak čtenář očekává, u průměrného případu vyjadřuje $T(n)$ průměr z délek výpočtů pro všechny vstupy velikosti n. Předpokládá se tedy rovnoměrné rozložení pravděpodobnosti na množině vstupů. (Ve specifických případech může mít samozřejmě smysl uvažovat i jiné rozložení.)

Obvykle je ovšem analýza průměrného případu těžší než analýza nejhoršího možného případu. U námi dříve uvedené verze *bubblesortu* je sice vcelku zřejmé, že algoritmus má složitost $\Theta(n^2)$ i v průměrném případě (proč ?), u dále připomenutého *quicksortu* už analýza tak zřejmá není. Algoritmus *quicksort* (který pro parametry A, p, q seřadí v poli A prvky $A[p], A[p+1], \ldots, A[r]$ vzestupně) zde připomeneme jen pseudokódem.

```
procedure Quicksort(A, p, r) ;
if  p < r
then
   q := Partition(A, p, r)
   Quicksort(A, p, q)
   Quicksort(A, q+1, r)

procedure Partition(A, p, r) ;
x := A[p]; i := p − 1; j := r + 1;
while TRUE
do
   repeat j := j − 1 until A[j] ≤ x;
   repeat i := i + 1 until A[i] ≥ x;
   if  i < j then  exchange A[i], A[j] else  return j
```

Poznámka. Analýza složitosti průměrného případu bude předmětem referátu na cvičení.

Kapitola 23

Návrh a analýza konkrétních (rychlých) algoritmů

Stručný obsah: Obecné metody návrhu algoritmů. Konkrétní algoritmy a jejich časová složitost: algoritmy pro prohledávání (binární hledání, max. vzdálenost v polygonu), metoda ‘rozděl a panuj’ (merge sort, násobení matic), ‘greedy’ (tj. hltavé) algoritmy (výběr aktivit, minimální kostra), dynamické programování (nejdelší společná podsekvence, násobení řetězce matic).

Cíl kapitoly:
Umět vysvětlit podstatu probíraných metod. Umět ilustrovat vhodnost použití jednotlivých metod a porozumět souvisejícím otázkám analýzy složitosti.

Studijní cíle: Umět vysvětlit podstatu probíraných metod. Umět ilustrovat vhodnost použití jednotlivých metod a porozumět souvisejícím otázkám analýzy složitosti.

V této části si připomeneme základní obecné metody návrhu algoritmů. Budeme je ilustrovat na příkladech, které budeme analyzovat z hlediska časové složitosti.

23.1 Prohledávání

Součástí řešení mnohých problémů je nutnost prohledání jakéhosi prostoru (který obsahuje např. všechna ‘přípustná řešení’, z nichž je potřeba vybrat optimální).

Příkladem je hledání zadaného prvku v *utříděném* seznamu, např. jména v telefonním seznamu

‘Naivní’ algoritmus založený na myšlence

> projdi sekvenčně všechny prvky seznamu, přičemž každý z nich porovnáš se zadaným

má očividně složitost $\Theta(n)$ (jako velikost vstupu bereme počet prvků v seznamu). Čtenář ale zajisté řeší tento problém v praxi jinak a je schopen navrhnout algoritmus se složitostí $\Theta(\log n)$. (Zde je ovšem předpokládán přímý přístup k prvkům seznamu – u stroje RAM je tedy $\Theta(\log n)$ relevantní, pokud je již seznam uložen v paměti).

Uveďme další problém:

> *Název (označení) problému*: *Max. vzdálenost v polygonu*
>
> *(Očekávaný) vstup*: konvexní polygon v dvourozměrném prostoru – zadaný posloupností vrcholů $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$ (‘dokola’, např. ve směru hodinových ručiček)
> *(Požadovaný) výstup*: dvojice vrcholů s max. vzdáleností.

Je přirozené považovat n, tj. počet vrcholů, za velikost vstupu.

Velmi jednoduše lze navrhnout algoritmus, který při hledání maxima probírá všechny dvojice vrcholů. Ten pak má celkem očividně složitost $\Theta(n^2)$. Ovšem při určitém větším vhledu a zamyšlení se nad problémem není těžké navrhnout algoritmus se složitostí $\Theta(n)$. Klíčem je idea prohledávání jen “perspektivních” (konkrétněji: “protilehlých”) dvojic. (Zkuste domyslet pořebné detaily !)

23.2 Metoda ‘rozděl a panuj’

Často použitelnou metodou při řešení ‘velkého’ úkolu (tj. nalezení výstupu pro ‘velký’ vstup určitého problému) je rozdělení na podúkoly, jejich vyřešení a nalezení hledaného výstupu zkombinováním mezivýsledků získaných z podúkolů.

Jestliže zmíněné podúkoly spočívají v řešení téhož problému pro menší vstupy, nabízí se rekurzivní algoritmus.

Pěkným příkladem je utřídění (velké) posloupnosti čísel. Tu je možné rozdělit na dvě části, utřídit každou zvlášť a výsledky pak ‘slít’ dohromady. Rekurzivní algoritmus založený na této myšlence se nazývá *Mergesort*.

Označíme-li $T(n)$ čas, který Mergesort spotřebuje při setřídění posloupnosti délky n, je zřejmé, že platí

- $T(1) = \Theta(1)$ ($\Theta(1)$ znamená prostě nějakou konstantu)
- pro $n \geq 2$: $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + f(n)$

Zde $f(n)$ znamená čas potřebný k slévání a zřejmě je $f(n) = \Theta(n)$.

Pomineme-li zde nepodstatné komplikace se zaokrouhlováním (např. všechny zlomky si můžeme představovat zaokrouhlené nahoru), můžeme psát

$$T(n) = 2T(n/2) + \Theta(n).$$

Dá se ukázat (např. dosazením) že řešení uvedené rekurentní rovnice splňuje podmínku $T(n) = \Theta(n \log n)$ (stejně jako tomu bylo u *heapsortu*).

Dále uvedeme obecné tvrzení, které je užitečným nástrojem pro řešení podobných rekurentních rovnic.

Tvrzení 23.2.1 *Nechť $a \geq 1$, $b > 1$ jsou konstanty, f je funkce (typu $\mathcal{N} \to \mathcal{N}$) a pro funkci $T : \mathcal{N} \to \mathcal{N}$ platí rekurentní vztah*

$$T(n) = aT(n/b) + f(n).$$

Pak platí:

1. *Je-li $f(n) = O(n^c)$ a $c < \log_b a$, pak $T(n) = \Theta(n^{\log_b a})$.*
2. *Je-li $f(n) = \Theta(n^{\log_b a})$, pak $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.*
3. *Je-li $f(n) = \Theta(n^c)$ a $c > \log_b a$, pak $T(n) = \Theta(n^c)$.*

Důkaz alespoň nastíníme, a sice pro případ $T(n) = aT(n/b) + n^c$. (Problémy se zaokrouhlováním ignorujeme.)

Představa rekurzivního výpočtu hodnoty $T(n)$ vede přirozeně k a-árnímu stromu, jehož hloubka je $\log_b n$; počet listů je tedy $a^{\log_b n}$ neboli $n^{\log_b a}$ (je totiž $\log_b a^{\log_b n} = (\log_b n) \cdot (\log_b a) = \log_b n^{\log_b a}$).

Předpokládáme-li rovnou, že $T(1) = 1$, pak se $T(n)$ rovná následujícímu součtu:

$$n^c + a\left(\frac{n}{b}\right)^c + a^2\left(\frac{n}{b^2}\right)^c + \ldots + a^{\log_b n}\left(\frac{n}{b^{\log_b n}}\right)^c =$$

$$= n^c + n^c\left(\frac{a}{b^c}\right) + n^c\left(\frac{a}{b^c}\right)^2 + \ldots + n^c\left(\frac{a}{b^c}\right)^{\log_b n} =$$

$$= n^c\left(1 + \left(\frac{a}{b^c}\right)^1 + \left(\frac{a}{b^c}\right)^2 + \ldots + \left(\frac{a}{b^c}\right)^{\log_b n}\right)$$

Připomeňme si součet (začátku) geometrické řady

$$1 + q + q^2 + \ldots + q^k = \frac{q^{k+1} - 1}{q - 1}$$

V případě $0 < q < 1$ máme $\sum_{i=0}^{\infty} q^i = \frac{1}{1-q}$,

v případě $q = 1$ máme $\sum_{i=0}^{k} q^i = k + 1$ a

v případě $q > 1$ máme $\sum_{i=0}^{k} q^i = \frac{q^{k+1}}{q-1} - \frac{1}{q-1} = O(q^k)$.

Tedy v případě

- $\frac{a}{b^c} < 1$ (tj. $\log_b a < c$) máme

$$T(n) \leq n^c \frac{1}{1 - \frac{a}{b^c}} = \Theta(n^c)$$

- $\frac{a}{b^c} = 1$ (tj. $\log_b a = c$) máme

$$T(n) = n^c(1 + \log_b n) = \Theta(n^{\log_b a} \log_b n)$$

- $\frac{a}{b^c} > 1$ (tj. $\log_b a > c$) máme

$$T(n) = n^c \cdot \left(\frac{a}{b^c}\right)^{\log_b n + 1} \cdot \frac{1}{\frac{a}{b^c} - 1} - n^c \cdot \frac{1}{\frac{a}{b^c} - 1}$$

  Tedy $T(n) = \Theta(n^c \cdot n^{\log_b \frac{a}{b^c}})$. Jelikož $\log_b \frac{a}{b^c} = \log_b a - \log_b b^c = \log_b a - c$, dostáváme $T(n) = \Theta(n^{\log_b a})$.

Všimněme si, že u našeho příkladu Mergesort nastával 2. případ ($a = 2$, $b = 2$, $f(n) = \Theta(n) = \Theta(n^{log_2 2})$), což dává onen výsledek $T(n) = \Theta(n \log n)$.

Podívejme se teď na problém násobení matic

Název (označení) problému: *Násobení dvou matic*

(Očekávaný) vstup: dvě čtvercové matice A, B rozměrů $n \times n$
(Požadovaný) výstup: matice C tž. $C = AB$.

Předpokládáme, že prvky matice jsou celá čísla (omezené velikosti). Z hlediska našeho vnímání je zde vhodné jako velikost vstupu považovat číslo n, ačkoliv počet zadávaných čísel je $\Theta(n^2)$ (je pro nás totiž přirozenější tvrzení 'program za 1 minutu zvládne násobení matic 800×800' než '... matic s 640000 prvky').

Přesvědčte se, že běžný algoritmus (podle definice násobení) má složitost $\Theta(n^3)$, dá-li se omezit konstantou čas pro provedení jedné aritmetické operace. (Tento odhad se samozřejmě vztahuje k uvedenému chápání velikosti vstupu. Kdybychom jako velikost vstupu brali počet vstupních čísel, dostali bychom odhad $\Theta(n^{1.5})$!)

Podívejme se, jestli přístup ‘rozděl a panuj’ přinese zlepšení. Omezíme se na zkoumání případů, kdy n je mocninou dvojky (jinak bychom toho mohli docílit příslušným doplněním matic nulami, čímž by se velikost vstupu zvětšila méně než dvakrát).

Matice A vznikne ‘přirozeným poskládáním’ ze čtyř čtvercových matic rozměrů $n/2 \times n/2$, označme tyto matice A_{11}, A_{12}, A_{21}, A_{22}. Podobně označme příslušné podmatice pro B a C. Snadno ověříme, že

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Pro časovou složitost $T(n)$ odvodíme z tohoto postupu rekurentní vztah

$$T(n) = 8T(n/2) + \Theta(n^2),$$

což dá rovněž řešení $T(n) = \Theta(n^3)$.

Strassen ovšem vymyslel postup, při kterém se jedno násobení dá ušetřit (za cenu zvýšení počtu sčítání), a kde pak příslušná rovnice je ve tvaru

$$T(n) = 7T(n/2) + \Theta(n^2).$$

Výsledná složitost $T(n) = \Theta(n^{\log_2 7})$ (pozn.: $\log_2 7$ je zhruba 2.81) je asymptoticky lepší než u standardního algoritmu.

(Postup ve Strassenově algoritmu a další jeho aspekty budou diskutovány na cvičení.)

23.3 ‘Greedy’ algoritmy

Slovo ‘greedy’ zde budeme překládat jako ‘hltavý’ (autor si není vědom zaužívaného českého termínu).

Hltavý přístup se osvědčuje u některých optimalizačních problémů. Jsou to problémy, u nichž je třeba udělat řadu rozhodnutí – lokálních kroků. Hltavý přístup vždy volí z možných kroků ten (lokálně) nejnadějnější.

Příslušný algoritmus je pak většinou velmi jednoduchý, ovšem použitelný pro daný problém je jen tehdy, jestliže série oněch lokálně nejnadějnějších rozhodnutí skutečně vede ke globálně optimálnímu řešení. Tuto pěknou vlastnost samozřejmě nemají všechny optimalizační problémy a důkaz toho, že daný problém (resp. daný hltavý přístup) tuto vlastnost má, je často obtížný (obvykle mnohem obtížnější než návrh příslušného algoritmu).

Uvedeme příklad úspěšného použití hltavého algoritmu:

> *Název problému*: *Výběr aktivit*
>
> *Vstup*: množina konečně mnoha aktivit $\{1, 2, \ldots, n\}$ s pevně určenými časovými intervaly (s_1, f_1), (s_2, f_2), …, (s_n, f_n), kde $(\forall i, 1 \leq i \leq n) : s_i < f_i$
> *Výstup*: množina obsahující největší možný počet vzájemně kompatibilních aktivit (tj. aktivit s vzájemně se nepřekrývajícími intervaly)

Je možné si např. představit, že máme dán seznam, kde každý prvek seznamu je nějaká přednáška, cvičení, školení, či jiná aktivita s pevně přidělenou dobou konání (tj. s přiděleným počátkem s a koncem f). Jde o to, umístit maximum z těchto aktivit do jedné posluchárny. Poznamenejme nejdříve, že přístup *hrubou silou*, spočívající v prověření všech možných výběrů aktivit, je exponenciální složitosti (časová složitost příslušného algoritmu je $2^{\Theta(n)}$) a již při ‘malém’ n nepoužitelný.

Hltavým způsobem můžeme řešit problém např. takto:

> Dokud to jde, opakuj následující krok:
> vyber aktivitu, která je kompatibilní s dosud vybranými (na začátku nejsou vybrány žádné) a skončí co nejdříve (když je takových víc, tak kteroukoli z nich).

Hltavost, či maximální ‘lokální nadějnost’ spočívá v tom, že po každém takovém výběru zbyde maximum času pro další aktivity.

Poznámka. Všimněme si jisté ‘nedeterminističnosti’ našeho problému – k danému vstupu může uvedené podmínce odpovídat více výstupů. I uvedený postup není plně deterministický (proč ?). Řešíme-li takový problém (deterministickým) algoritmem, upřednostňujeme vlastně vždy jeden výstup mezi všemi možnými. Tak je tomu i v dále uvedeném pseudokódu.

Je zřejmé, že je užitečné aktivity nejdříve setřídit podle koncových časů; to je možné udělat nějakým známým algoritmem v čase $O(n \log n)$ (počet aktivit budeme považovat za velikost vstupu).

{předp., že poč. časy jsou již v poli s}
{a konc. časy v poli f}
{dále již předp. $f[1] \leq f[2] \leq \ldots f[n]$,}
{kde n představuje počet aktivit}
$A := \{1\}$; $j := 1$;
for $i := 2$ **to** n **do**
  if $s[i] \geq f[j]$ **then**
    ($A := A \cup \{i\}$; $j := i$)
{vybr. aktivity jsou v množ. A}

V uvedeném případě je důkaz faktu, že hltavý přístup skutečně vede k optimálnímu řešení, celkem snadný. (Indukcí podle počtu aktivit n.)

Jak jsme již uvedli, počet aktivit n je zde přirozenou mírou velikosti vstupu a snadno odvodíme, že uvedený algoritmus má složitost $\Theta(n)$, když předpokládáme, že aktivity jsou již utříděny podle koncových časů.

Typickým příkladem úspěšného použití hltavého přístupu je problém minimální kostry v grafu ($\mathcal{N}_+$ označuje množinu všech kladných celých čísel):

Název problému: *Minimální kostra*

Vstup: neorientovaný souvislý graf $G = (V_G, E_G)$, ohodnocení hran $f : E_G \to \mathcal{N}_+$
Výstup: graf $H = (V_H, E_H)$, kde $V_H = V_G$ a $E_H \subseteq E_G$, který je souvislý a přitom $\sum_{e \in E_H} f(e)$ je minimální.

Jednou možnou interpretací je problém stavitele železnic. Má dánu množinu měst (vrcholů grafu) a dále všechna možná spojení mezi dvojicemi měst, kudy je možné vést železnici. Každému takovému možnému spojení (hraně grafu) odpovídají určité náklady na postavení železnice (ohodnocení příslušné hrany). Úkolem stavitele je postavit železnici tak, aby každé město bylo spojeno s každým (případně přes další města) a aby náklady stavby byly co nejmenší.

Opět poznamenejme, že přístup hrubou silou (probrání všech možností postavení železnice) vede k exponenciálnímu algoritmu a nepřipadá pro větší vstupy v úvahu.

Všimněme si nyní, že řešení (tj. graf H) je určitě stromem; kdyby ne, obsahoval by cyklus a mohli bychom pak při zachování souvislosti jednu hranu odstranit a dostat tak lepší řešení. Jednou z 'hltavých' možností je následující přístup 'líného' stavitele:

Postav nádraží v libovolném městě.
Dokud to jde, opakuj:
k dosud postavené železnici připoj co nejlevnější úsek (hranu), ovšem tak, aby nevznikl cyklus.

Tento postup je zachycen následujícím pseudokódem. (Zde necháváme určitý nedeterminismus i v pseudokódu; samozřejmě jakýkoli deterministický výběr z příslušných možností vede k cíli – tj. k jedné z minimálních koster grafu).

{předp., že jsou dány V_G, E_G, f}
zvol lib. $u \in V_G$; $V_H := \{u\}$; $E_H := \emptyset$; $M := V_G - \{u\}$;
for each $v \in M$ **do**
  if $(u, v) \in E_G$

then ($otec(v) := u$; $d(v) := f((u,v))$)
else $d(v) := \infty$;
while $M \neq \emptyset$ **do**
najdi $v \in M$ tž. $d(v) = min\{d(w) \mid w \in M\}$;
$V_H := V_H \cup \{v\}$; $E_H := E_H \cup \{(v, otec(v))\}$; $M := M - \{v\}$;
for each $w \in M$ **do**
if $(v,w) \in E_G$ and $f((v,w)) < d(w)$
then ($otec(w) := v$; $d(w) := f((v,w))$)
return $H = (V_H, E_H)$

Není samozřejmé, že tento přístup vede k optimálnímu řešení, a je to potřeba dokázat.

Ukažme sporem. Uvažujme první situaci, kdy náš algoritmus k dosud vytvořenému stromu T, který je (dosud) podgrafem nějaké minimální kostry K, přidá hranu (u,v) způsobící, že takto vzniklý strom (již) není podgrafem žádné minimální kostry. V K ovšem vede cesta z u do v přes nějakou hranu (x,y), kde $x \in T$ a $y \notin T$. Odejmu-li ovšem z K hranu (x,y) a přidám (u,v), dostanu opět minimální kostru (promyslete si to !), což je spor.

Pokud za velikost vstupu považujeme počet vrcholů n (kolik hran pak graf může mít ?), je složitost uvedeného algoritmu $\Theta(n^2)$. Poznamenejme, že existují i jiné algoritmy; pro jejich srovnání je pak vhodné uvažovat složitost jako funkci dvou parametrů: počtu vrcholů n a počtu hran m (jeden z nich má např. složitost $O(m \log n)$).

23.4 Dynamické programování

Pojem ‘dynamické programování’ pochází z teorie řízení a nemá nic společného s běžným významem tohoto sousloví v oblasti programování počítačů. Zde se jedná o název metody, při které se problém řeší pomocí postupného (vyplňování ‘tabulky’) řešení podproblémů (od nejmenších po největší). (Někteří autoři hledají jiné názvy, např. dynamické plánování).

Metoda dynamického programování je jakýmsi limitním případem metody ‘rozděl a panuj’. Řešení (velké) instance (tj. vstupu) problému také spočívá v kombinaci výsledků podúkolů; jelikož ovšem při řešení různých podúkolů by mnohokrát docházelo k řešení společných podpodúkolů (atd.), je lepší rekurzivní řešení (přístup shora-dolů) nahradit přístupem zdola-nahoru: nejdříve se vyřeší všechny potřebné elementární (nejmenší) úkoly, z jejich řešení pak odvodíme řešení větších úkolů, z nich pak řešení ještě větších atd. až získáme řešení našeho (velkého) úkolu.

Tento přístup se někdy uplatní u optimalizačních úloh, u kterých selhávají hltavé algoritmy.

Jedním z typických příkladů je problém *nalezení nejdelší společné podsekvence*. Řekneme, že slovo (řetězec, posloupnost) u je podsekvencí slova v, jestliže u dostaneme z v vymazáním některých (třeba žádných) výskytů symbolů (členů posloupnosti). Instancí zmíněného problému jsou dvě slova v, w a úkolem je najít nejdelší slovo u, které je podsekvencí slova v i slova w.

Podrobněji budeme metodu dynamického programování ilustrovat na jiném příkladu:

> *Název problému*: *Násobení řetězce matic*
>
> *Vstup*: řetězec matic $A_1 A_2 \dots A_n$
> *Výstup*: plně uzávorkovaný součin $A_1 A_2 \dots A_n$ tž. při použití standardního algoritmu násobení matic se vykoná minimální počet skalárních násobení.

Připomeňme, že součin matic AB, kde A má rozměry $k \times \ell$ a B má rozměry $m \times n$, je definován jen v případě $\ell = m$ (výsledná matice má rozměry $k \times n$); počet potřebných skalárních násobení je zde $k\ell n$.

Budeme tedy předpokládat, že pro každé $i, 1 \leq i \leq n$ má matice A_i rozměry $p_{i-1} \times p_i$, kde $p_0, p_1, \dots, p_n$ jsou příslušná celá kladná čísla. Všimněme si, že vlastně tato čísla jsou podstatná v našem problému, nikoli hodnoty prvků jednotlivých matic.

Navíc připomeňme, že násobení matic je asociativní, takže nezáleží na pořadí násobení dvojic matic, které zvolíme při výpočtu součinu $A_1 A_2 \dots A_n$ (pořadí násobení dvojic matic jednoznačně určíme příslušným vložením závorek).

Čtenář se snadno přesvědčí (konstrukcí jednoduchého příkladu), že počty potřebných skalárních násobení se při různých uzávorkováních mohou výrazně lišit.

Dá se také ukázat, že počet možných uzávorkování roste s rostoucím n exponenciálně, takže probrání všech možností nepřipadá v úvahu (s výjimkou malých hodnot n).

Všimněme si, že optimální vynásobení řetězce $A_1 A_2 \dots A_n$ lze popsat tak, že nejdříve se optimálně vynásobí řetězec $A_1 A_2 \dots A_k$, potom řetězec $A_{k+1} A_{k+2} \dots A_n$ a na závěr se vynásobí získané dva mezivýsledky; k je ovšem (neznámý) 'optimální index' v rozmezí $1 \leq k \leq n-1$ (k určuje umístění 'nejvnějšnějších závorek').

Zmíněný optimální index označíme $s[1, n]$; obecně $s[i, j]$ ($1 \leq i < j \leq n$) označuje optimální index při násobení řetězce $A_i A_{i+1} \dots A_j$. Označme dále jako $m[i, j]$ počet skalárních násobení při optimálním vynásobení řetězce $A_i A_{i+1} \dots A_j$ (zde připouštíme i rovnost $i = j$; pochopitelně $m[i, i] = 0$). Všimněme si, že je-li $s[i, j] = k$, pak $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Následující procedura, parametrizovaná vektorem čísel (rozměrů matic) $p = (p_0, p_1, \dots, p_n)$ postupně vyplní 'tabulky' m a s:

```
Matrix-chain-order(p)
n := length(p) − 1;
for i := 1 to n do m[i, i] := 0;
for ℓ := 2 to n do
   for i := 1 to n − ℓ + 1 do
      j := i + ℓ − 1; m[i, j] := ∞;
      for k := i to j − 1 do
         q := m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j;
         if q < m[i, j] then (m[i, j] := q; s[i, j] := k);
return m, s
```

Je snadné vyvodit, že složitost Matrix-chain-order je $O(n^3)$ (a také $\Omega(n^3)$, tedy $\Theta(n^3)$). Vlastní program násobení řetězce $A_1 A_2 \dots A_n$, označme tento řetězec A, pak spočívá ve vytvoření tabulky s pomocí procedury Matrix-chain-order (volané pro příslušný vektor rozměrů) a v následném vyvolání Matrix-chain-multiply$(A, s, 1, n)$, kde procedura Matrix-chain-multiply je rekurzivně definována takto:

```
Matrix-chain-multiply(A, s, i, j)
if j > i
then
   X :=Matrix-chain-multiply(A, s, i, s[i, j])
   Y :=Matrix-chain-multiply(A, s, s[i, j] + 1, j)
   return X · Y
else
   return A_i
```

Kapitola 24

Problémy, třídy složitosti problémů, horní a dolní odhady

Stručný obsah: Další poznámky k analýze složitosti algoritmů. Vymezení pojmu ‘problém’ (nebo též ‘výpočetní problém’); speciální případ ANO/NE problému (tj. tzv. rozhodovacího problému). Složitost problému; třídy časové a prostorové složitosti. Algoritmy jakožto horní odhady. Dolní odhady, obtížnost jejich získávání. Mezery mezi známými horními a dolními odhady složitosti problémů.

Cíl kapitoly:
Umět definovat a ilustrovat pojem ‘problém’, umět vysvětlit způsob definice složitosti problémů, horních a dolních odhadů. Schopnost vše ilustrovat na příkladech.

Studijní cíle: Umět definovat a ilustrovat pojem ‘problém’, umět vysvětlit způsob definice složitosti problémů, horních a dolních odhadů. Schopnost vše ilustrovat na příkladech.

Poznámka. Kromě výrazů ‘stroj RAM’ či ‘RAM-stroj’ budu užívat jen zkratku ‘RAM’; budu např. mluvit o sestrojení RAMu apod.

Další poznámky k složitosti algoritmů

Vrátíme se k některým věcem týkajícím se (časové či paměťové) složitosti algoritmů, o kterých jsme zatím explicitně moc nemluvili, ale kterých bychom si měli být velmi dobře vědomi.

Zatím jsme přesně definovali pojem časové (a prostorové) složitosti RAMu. Většinou jsme ale uvedli algoritmus zapsaný pseudokódem a hovořili jsme o složitosti tohoto algoritmu. Striktně vzato bychom ale vždy místo o složitosti algoritmu A měli hovořit o složitosti RAMu M_A, který je možné k algoritmu A v podstatě mechanicky sestrojit (tj. který by byl z A vytvořen určitým ‘překladačem’).

Nedefinovali jsme ovšem, jaké příkazy se mohou objevovat v pseudokódu, ani jsme samozřejmě nedefinovali příslušný překladač. Mlčky jsme vlastně udělali určitou úmluvu: provedli jsme konstrukci příslušného RAMu jen pro několik málo algoritmů zapsaných pseudokódem a spoléháme se na to, že algoritmy popisujeme vždy dostatečně podrobně k tomu, abychom v případě potřeby mohli příslušný RAM přímočaře zkonstruovat. Přitom samozřejmě předpokládáme, že zmíněnou přímočarou konstrukcí bychom všichni dospěli v podstatě k jednomu a témuž RAMu. To 'jednomu a témuž' nelze brát úplně doslova, stačí samozřejmě, že příslušné RAMy by měly v podstatě stejnou složitost – to jest, mohly by se lišit v konkrétních počtech instrukcí realizujících ten či onen příkaz pseudokódu, což ovšem nevadí, nebazírujeme-li na přesných hodnotách příslušných konstant.

Právě to, že přesné hodnoty konstant pro nás nejsou podstatné a složitost vždy jen určitým způsobem odhadujeme (aproximujeme), nám umožňuje používat způsob analýzy složitosti algoritmů, při němž se přímo nezmiňujeme o RAMu v pozadí, natož abychom ho konstruovali.

Poznamenejme ještě další věc. Vstupem pro RAM je vlastně posloupnost čísel. Takže chceme-li mluvit o složitosti algoritmu, měl by vlastně jeho vstup (i výstup) být posloupností čísel – nebo by mělo být jasné, jaké kódování vstupu pomocí posloupnosti čísel máme na mysli. U námi dosud zkoumaných problémů to bylo v podstatě zřejmé: např. graf lze přirozeně zadat incidenční maticí, matici je možné přímočaře 'linearizovat' (zapsat např. po řádcích – s dohodnutými oddělovači) apod.

Připomeňme si ještě, že velikost vstupu u RAMu jsme definovali jako počet členů vstupní posloupnosti čísel (to je v případě uvažování jednotkové míry; v případě použití logaritmické míry je velikost vstupu v podstatě rovna počtu bitů, do kterých lze vstup zapsat). Ovšem když jsme např. analyzovali problém násobení matic, brali jsme za míru velikosti vstupu číslo udávající rozměr matic; přitom (ono přirozené linearizované) zadání čtvercové matice s rozměrem n (tedy typu $n \times n$) zabere n^2 vstupních buněk RAMu (případně další buňky jako oddělovače řádků).

Když tedy hovoříme o složitosti algoritmu v takovém případě, vztahujeme ji pořád k příslušnému RAMu, ale měníme standardní definici velikosti vstupu – to musíme vždy jasně říci. Jak jsme viděli, děláme to tehdy, když pro daný problém je pozměněná definice velikosti vstupu vhodnější při vyjadřování a umožňuje 'průhlednější' podání výsledků analýzy složitosti. Jak jsme také zmínili v případě minimální kostry grafu, je z takových důvodů někdy vhodné popisovat velikost vstupu více čísly (např. počet vrcholů n, počet hran m); složitost je pak funkcí více parametrů.

Všechny uvedené poznámky je vhodné si důkladně promyslet a u každého konkrétního případu je nutné si uvědomovat, co je (třeba mlčky) předpokládáno.

Některé aspekty si ještě osvětlíme na příkladu problému, který hraje důležitou roli např. v kryptografii.

Název: *Prvočíselnost*

Vstup: přirozené číslo k
Výstup: ANO, když k je prvočíslo, NE, když k je číslo složené.

Pravděpodobně nás rychle napadne následující algoritmus:

$\{$předp. vstup $k \geq 2\}$
$hm := \lfloor \sqrt{k} \rfloor$;
for $i := 2$ **to** hm **do**
  if k mod $i = 0$ **then return** NE

return ANO

Když chceme odhadnout složitost algoritmu, musí být samořejmě jasné, co se rozumí velikostí vstupu. Jelikož vstupem je jedno číslo, má být velikost vždy 1 ? U předchozích problémů (jako třeba násobení matic) jsme předpokládali omezenou velikost zadávaných čísel (a tedy konstantní čas při aritmetických operacích s nimi) – neomezovali jsme ovšem délku zadávané posloupnosti. Nyní ovšem neomezujeme velikost (jednoho) zadávaného čísla. Takový vstup si 'přímo říká' o použití logaritmické míry, kde tedy velikost vstupu pro vstupní číslo k je rovna $\log_2 k$ (přesněji $\lceil \log_2(k+1) + 1 \rceil$, což ale není podstatné).

Uvedený algoritmus má takto exponenciální časovou složitost; všimněte si, že v případě, že k je prvočíslo, provede se tělo cyklu zhruba $k^{0.5}$ krát, tj. $2^{0.5 \log_2 k}$, tedy $2^{\Theta(n)}$ krát, kde n je velikost vstupu. Z toho je vidět, že algoritmus je použitelný jen na čísla s malým počtem míst v dekadickém zápisu (a rozhodně ho nelze použít např. na 100-místná čísla vyskytující se v kryptografických problémech).

Všimněme si ještě, že kdybychom za velikost vstupu považovali přímo hodnotu čísla k (číslo bychom vlastně zadávali v unární soustavě – jako posloupnost k jedniček), byla by složitost algoritmu určitě v $O(n^2)$; 'najednou' by to bylo v praxi zvládnutelné pro vstupy délky 100 (problém je samozřejmě v tom, že např. vstup délky 100 v tomto případě odpovídá vstupu délky 3 v případě předchozím).

Definice pojmu 'problém'

Algoritmy jsme prezentovali jako návody k řešení určitých problémů. Slovo 'problém' má v přirozeném jazyce hodně významů; co ale znamená tento pojem v našem kontextu ?

Připomeňme, že problémy jsme zadávali většinou následujícím schématem:

Název (označení problému): *XY*

(Očekávaný) vstup: zde je popsáno, co je přípustným vstupem (zadáním, instancí) našeho problému
(Požadovaný) výstup: zde je popsáno, jaký výstup (výsledek) je očekáván pro zadaný vstup (je přiřazen zadanému vstupu)

Pokud chceme napsat formální definici pojmu problém, mohla by vypadat takto:

Definice 24.0.1 Problém *je určen trojicí* (IN, OUT, p)*, kde* IN *je množina (přípustných)* vstupů*,* OUT *je množina* výstupů *a* p *je zobrazení (tedy funkce)* $p : IN \to OUT$*.*

Takto definovaný problém se někdy též nazývá 'výpočetní problém' (computational problem).

Jak jsme už řekli, aby mělo smysl hovořit o tom, že určitý RAM řeší daný problém, musí být množiny IN i OUT jistými množinami posloupností celých čísel, tedy $IN, OUT \subseteq \mathcal{Z}^*$ (kde $\mathcal{Z}$ označuje množinu všech celých čísel):

Definice 24.0.2 RAM M řeší problém $P = (IN, OUT, p)$*, kde* $IN, OUT \subseteq \mathcal{Z}^*$*, jestliže má tuto vlastnost:*
začne-li výpočet v počáteční konfiguraci se vstupem $c_1 c_2 \ldots c_n \in IN$*, pak svůj výpočet (regulérně) skončí, přičemž na výstupu je* $p(c_1 c_2 \ldots c_n)$*.*

Jak jsme již poznamenali, u námi zkoumaných problémů byly přípustné vstupy de facto posloupnosti čísel. Ovšem např. v problému vyhledávání vzorku v textu

Název: *Vzorek v textu*

Vstup: text t ('dlouhá' posloupnost symbolů), vzorek p ('krátká' posloupnost symbolů)
Výstup: místo prvního výskytu p v t, či odpověď 'nevyskytuje se'.

čísla přímo nevystupují. Stačí ale kódovat užité symboly čísly (vzpomeňte např. na ASCII-kód) a máme vstupy a výstupy opět v žádaném tvaru.

Poznámka. Čísla zapisujeme většinou (v dekadické soustavě) jako řetězce symbolů z určité konečné abecedy. Když se na chvíli zamyslíme, uvědomíme si, že vlastně lze rozumně předpokládat, že jakýkoli vstup jakéhokoli problému lze vždy (více či méně elegantně) ztotožnit s řetězcem symbolů z pevně dané abecedy (např. '1-bytové' abecedy s 256 prvky).
Proto si lze pod množinou přípustných vstupů vždy představit množinu řetězců ze Σ^* (kde Σ je ona pevná abeceda), které splňují nějakou algoritmicky snadno verifikovatelnou podmínku (např. jsou dohodnutým zápisem řetězce matic apod.)
Podobně i množinu výstupů je možné ztotožnit s (pod)množinou (množiny) Σ^*.

Speciálním případem problémů jsou tzv.

rozhodovací problémy, neboli *ANO/NE problémy*.

U takového problému je množina OUT dvouprvková; standardně pak předpokládáme, že OUT$=\{$ANO,NE$\}$ (či OUT$=\{1,0\}$).

Příkladem ANO/NE problému je výše uvedený problém prvočíselnosti. Poznamenejme, že k některým (obecným) problémům (např. optimalizačním) lze přirozeně přiřadit tzv. rozhodovací (ANO/NE) verzi problému: např. u problému minimální kostry se vstup rozšíří o číslo c a požadovaný výstup pak bude ANO, jestliže ex. kostra s ohodnocením nejvýše rovným c, a NE v opačném případě. V této formě je pak přirozenější zadat problém schématem

Název: *Minimální kostra (ANO/NE verze)*

Vstup: neorientovaný souvislý graf $G=(V_G,E_G)$, ohodnocení hran $f: E\to\mathcal{N}_+$, číslo c
Otázka: existuje graf $H=(V_H,E_H)$, kde $V_H=V_G$ a $E_H\subseteq E_G$, který je souvislý a přitom $\sum_{e\in E_H} f(e)\leq c$?

Obecně běžné schéma pro zadávání ANO/NE problémů je tedy

Název (označení problému): *XY*

Vstup: zde je popsáno, co je přípustným vstupem (zadáním, instancí) našeho problému.
Otázka: zde je otázka týkající se (zadaného) vstupu, na niž je odpověď ANO nebo NE.

U problému prvočíselnosti by otázka samozřejmě zněla: 'je k prvočíslo ?'.

Uveďme si ještě jeden ANO/NE problém, který bude hrát důležitou roli v dalším.

Název: SAT (*problém splnitelnosti booleovských formulí*)

Vstup: booleovská formule v konjunktivní normální formě
Otázka: je daná formule splnitelná (tj. existuje pravdivostní ohodnocení proměnných, při kterém je formule pravdivá) ?

Složitost problémů

Intuitivně cítíme, že různé problémy mohou být různě 'složité'; co to ale je ona složitost problémů ? Zatím jsme hovořili jen o složitosti algoritmů, přesněji řečeno RAMů. Víme-li např. o algoritmu (RAMu), který daný problém řeší a má složitost $\Theta(n^3)$, je toto $\Theta(n^3)$ jen určitým horním odhadem 'skutečné' složitosti problému (můžeme říci 'složitost problému je nejvýše kubická'). Je např. možné, že nalezneme jiný algoritmus, který řeší náš problém a který má složitost $O(n^2)$; tím jsme náš dosud známý odhad zlepšili (víme už, že 'složitost problému je nejvýše kvadratická'). Tyto úvahy nás přivedou k závěru, že složitost problému lze ztotožnit se složitostí 'optimálního' algoritmu, který daný problém řeší. Při exaktní definici onoho pojmu 'optimální' ovšem vzniknou jisté komplikace. Ty obejdeme použitím následujícího přístupu:

Definice 24.0.3 *Pro funkci $f : \mathcal{N} \longrightarrow \mathcal{N}$ rozumíme* třídou časové složitosti $\mathcal{T}(f)$*, též značenou $\mathcal{T}(f(n))$, množinu těch problémů, které jsou řešeny RAMy s čas. složitostí v $O(f)$.*
Třídou prostorové složitosti $\mathcal{S}(f)$*, též $\mathcal{S}(f(n))$, rozumíme třídu těch problémů, které jsou řešeny RAMy s prostorovou složitostí v $O(f)$.*

Důležitá úmluva. V předchozí definici a všude, kde hovoříme o třídách složitosti vztahujících se k RAMu jako k referenčnímu modelu, máme vždy na mysli **logaritmickou míru** při odkazu na složitost RAMu. Jde o to, aby takto definované třídy složitosti rozumně odpovídaly realitě.

Řekneme-li tedy např., že problém P patří do $\mathcal{T}(n^2)$ (taky se v této souvislosti říká 'časová složitost P je v $O(n^2)$' či ještě stručněji 'P je v $O(n^2)$'), říkáme tím, že existuje algoritmus s nejvýš kvadratickou časovou složitostí, který řeší problém P (přesněji řečeno: existuje RAM s nejvýš kvadratickou časovou složitostí v logaritmické míře, který řeší problém P).

Poznámka. Připomeňme, že pro příslušnost problému k dané třídě složitosti je důležitý i způsob kódování vstupu (a ten je tedy nutno chápat jako součást definice problému).

Všimněme si, že platí

$$(P \in \mathcal{T}(f)) \wedge (f \in O(g))) \Longrightarrow (P \in \mathcal{T}(g)).$$

Takže např.

$$\mathcal{T}(n) \subseteq \mathcal{T}(n \cdot log\, n) \subseteq \mathcal{T}(n^2) \subseteq \mathcal{T}(n^3) \subseteq \mathcal{T}(2^n)$$

Jak jsme již řekli, algoritmy řešící daný problém poskytují *horní odhady složitosti problému*. Horší je to s *dolními odhady*. Chceme-li např. ukázat, že daný problém je v $\mathcal{T}(n^2)$, stačí ukázat algoritmus se složitostí $O(n^2)$, který jej řeší. Chceme-li ale ukázat, že problém není v $\mathcal{T}(n^2)$, musíme ukázat, že žádný algoritmus (RAM), který jej řeší, nemá složitost v $O(n^2)$. Získat takový dolní odhad je obvykle velmi těžké.

Na cvičení si ukážeme, že každý algoritmus řešící problém třídění (a založený na porovnávání dvojic) má nutně složitost $\Omega(n \log n)$. Složitost problému třídění je takto určena přesně (samozřejmě až na zanedbávané konstantní faktory) – horní odhad se rovná dolnímu.

Poznamenejme, že u mnohých problémů, se kterými jsme se setkali a setkáme, je velká 'propast' mezi (dosud známými) horními a dolními odhady. Např. pro výše uvedený problém SAT je známý horní odhad exponenciální, ovšem známý dolní odhad je pouze lineární (tj. $\Omega(n)$) ! (S dalšími problémy tohoto typu se setkáme v části o NP-úplných problémech).

Kapitola 25

Třída PTIME a její robustnost. Turingovy stroje.

Stručný obsah: P (=PTIME) jako aproximace třídy prakticky zvládnutelných problémů. Vysvětlení Její robustnosti vůči (rozumným) výpočetním modelům. Turingovy stroje; vzájemná ‘polynomiální’ simulace s RAMy. Zmínka o dalších výpočetních modelech a jejich vzájemné polynomiální simulaci.

Cíl kapitoly:
Umět vysvětlit význam a robustnost třídy PTIME. Umět definovat a ilustrovat Turingovy stroje, vysvětlit ideje polynomiální simulace s RAMy.

Studijní cíle: Umět vysvětlit význam a robustnost třídy PTIME. Umět definovat a ilustrovat Turingovy stroje, vysvětlit ideje polynomiální simulace s RAMy.

Část teorie, která se někdy nazývá *konkrétní složitost*, studuje složitost konkrétních problémů (a algoritmů), resp. příslušné horní a dolní odhady. Tzv. *strukturální složitost* má za úkol zkoumat strukturu tříd složitosti problémů. Podotkněme ovšem, že obě zmíněné partie se samozřejmě prolínají a ovlivňují. Jedním z nejdůležitějších cílů teorie (strukturální) složitosti je co možná nejlépe charakterizovat *třídu zvládnutelných problémů* (tj. třídu problémů, pro které existují ‘dostatečně rychlé’, tj. v praxi použitelné, algoritmy).

Třída P (neboli PTIME)

Jako nejrozumnější (horní) aproximace třídy zvládnutelných problémů se (zatím) ukázala třída označovaná PTIME, nebo jen P (ze slova ‘Polynomial’), definovaná

$$\text{PTIME} = \bigcup_{k=0}^{\infty} \mathcal{T}(n^k)$$

To znamená, že pojem ‘rychlý algoritmus’ je ztotožňován s pojmem ‘polynomiální algoritmus’ (tj. algoritmus s polynomiální časovou složitostí). To není samozřejmě ideální (např. algoritmus s časovou složitostí zhruba $n^{1000000}$ těžko lze považovat za rychlý), zatím však nebyla nalezena lepší charakterizace. V praxi se ukazuje, že když je pro nějaký problém nalezen polynomiální algoritmus, obvykle se podaří nalézt i algoritmus s nízkým stupněm polynomu (řekněme menším než 6).

Poznámka. Brzy se dostaneme k problémům, pro něž polynomiální algoritmy neexistují či nejsou známy. Klademe-li si (jen) otázku, zda daný problém patří do PTIME, pohybujeme se na na úrovni (podstatně) ‘hrubší’ analýzy než při pouhém zanedbávání konstant u značení O, Θ apod. Takto např. i metoda *bubblesort* prokazuje, že pro problém třídění existuje rychlý (rozuměj polynomiální) algoritmus (byť v detailnějším pohledu, tj. na jemnější úrovni analýzy, vnímáme *bubblesort* jako ‘pomalý’).

Uvědomme si, že třídy složitosti problémů, jak jsme je definovali v definici 24.0.3, i právě definovaná třída PTIME jsou závislé na našem zvoleném referenčním modelu – vztahují se tedy k RAMu (s logaritmickou mírou). Navrhneme-li jiný výpočetní model (do nějž budeme ‘překládat’ naše algoritmy), můžeme dostat jiné třídy složitosti !

Ovšem třída PTIME je (díky své ‘hrubosti’) *robustní*: PTIME je nezávislá na tom, zda zvolíme jako referenční model RAM nebo jiný ‘rozumný’ výpočetní model. Ukázalo se totiž, že všechny navržené rozumné modely počítače jsou ‘polynomiálně ekvivalentní’, tj. jsou schopny se vzájemně simulovat s ‘pouze’ polynomiální ztrátou; to znamená, že pro každé dva takové modely $\mathcal{M}_1$, $\mathcal{M}_2$ existuje konstanta c tak, že když problém P patří do $\mathcal{T}(f_1)$ pro referenční model $\mathcal{M}_1$, tak P patří do $\mathcal{T}(f_2)$, kde $f_2(n) = (f_1(n))^c$, pro referenční model $\mathcal{M}_2$. Všechny zmíněné rozumné modely tedy definují jednu a tutéž třídu PTIME.

Samozřejmě se naskýtá otázka, co to jsou *rozumné* výpočetní modely. Obecně řečeno se tím myslí ty, u nichž analýza algoritmů (v nich ‘naprogramovaných’) dává realistické výsledky pro praxi (alespoň na úrovni oné ‘hrubé’ analýzy diskutované výše). Technicky lze definovat jako rozumné ty modely, jež jsou polynomiálně ekvivalentní modelu RAM (myslí se samozřejmě s logaritmickou mírou, jak bylo dohodnuto v úmluvě za definicí 24.0.3). V literatuře se ovšem často v uvedené definici ‘rozumnosti’ odkazuje k historicky prvnímu modelu počítače (resp. ‘výpočtáře’), jenž známe pod názvem Turingův stroj a jenž si připomeneme níže.

Poznamenejme ještě, že uvažujeme *sekvenční modely*, k otázce *paralelních modelů* se letmo dostaneme později.

Turingovy stroje

Předpokládaným vstupem pro Turingův stroj je řetězec (vstupních) symbolů. Ten je rovnou uložen na (pracovní) pásce stroje (tj. oboustranně potenciálně nekonečné lineární pásce, rozdělené na buňky; každá buňka může obsahovat jeden symbol). Tímto vstupem je určena počáteční konfigurace stroje; tato konfigurace se mění krok za krokem podle předepsaných pravidel (daných přechodovou funkcí). Stroj má (na rozdíl od RAMu) sekvenční přístup k ‘paměti’ (v jednotlivém kroku má přístup jen k jedné buňce, ‘posunout’ se v jednom kroku může vždy jen na buňku sousední). Řetězec (neprázdných) symbolů na pásce po ukončení výpočtu (dosažení koncové konfigurace) je považován za výstup (příslušný původnímu vstupu).

Definice 25.0.4 Turingův stroj M *je určen následujícími parametry (formálně řečeno, je to uspořádaná šestice):* $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$*, kde* Q *je konečná neprázdná množina* stavů, Γ *je konečná neprázdná množina* (páskových) symbolů, $\Sigma \subseteq \Gamma$, $\Sigma \neq \emptyset$ *je množina* vstupních symbolů, $q_0 \in Q$ *je* počáteční stav, $F \subseteq Q$ *je množina* koncových stavů, $\delta : (Q - F) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ *je přechodová funkce. Předpokládáme, že v* $\Gamma - \Sigma$ *je vždy obsažen speciální prvek* $\Box$ *označující* prázdný znak.

Konfigurací Turingova stroje M *rozumíme libovolné slovo tvaru* uqv*, kde* $u, v \in \Gamma^*$ *a* $q \in Q$*.* Konfigurace uqv *je* počáteční*, jestliže* u *je prázdné slovo,* $q = q_0$ *a* $v \in \Sigma^*$ *;* uqv *je* koncová konfigurace*, jestliže* $q \in F$*.*

Konfiguraci qv *ztotožňujeme s konfigurací* $\Box qv$*, podobně* uq *s* $uq\Box$*. (Obecně přidání lib. řetězce z* $\{\Box\}^*$ *vlevo či vpravo nehraje roli – vede k ekvivalentní konfiguraci.)*

Konfigurace $K = uaqbv$*, kde* $u, v \in \Gamma^*$*,* $a, b \in \Gamma$ vede v jednom kroku ke konfiguraci K'*, příslušnou relaci označujeme* $\vdash_M$ *nebo jen* $\vdash$ *(píšeme tedy* $K \vdash K'$*) právě když platí jedna z těchto možností:*

- $\delta(q, b) = (q', b', 0)$ *a* $K' = uaq'b'v$*,*
- $\delta(q, b) = (q', b', +1)$ *a* $K' = uab'q'v$*,*
- $\delta(q, b) = (q', b', -1)$ *a* $K' = uq'ab'v$*.*

Výpočet Turingova stroje nad zadaným vstupem se tedy zastaví v momentě dosažení koncového stavu (dojde-li k tomu vůbec). Výstupem (výpočtu) pak obvykle rozumíme řetězec z $(\Gamma - \{\Box\})^*$ zapsaný na pásce v příslušné koncové konfiguraci.

Turingovy stroje představují výpočetní model, který je možné brát jako určitou alternativu k strojům RAM. Již jsme se zmínili o tom, že ačkoliv RAM pracuje s čísly a Turingův stroj se znaky (symboly abecedy), jedná se v zásadě o totéž, jelikož čísla běžně zapisujeme řetězcem symbolů a naopak symboly abecedy jsou běžně kódovány čísly.

Poznámka. Alan Turing navrhl 'svůj' model již v třicátých letech 20. století, dříve než byly vyvinuty samočinné počítače. RAM byl navržen o několik desetiletí později jako realističtější model počítače.

Časovou a prostorovou složitost konkrétního Turingova stroje definujeme samozřejmě obdobně jako u RAMu:

Definice 25.0.5 Velikostí vstupu *Turingova stroje* M *rozumíme počet buněk pásky, které daný vstup zabírá (tedy délku vstupního řetězce).*
Délka výpočtu *Turingova stroje* M *pro konkrétní vstup se definuje jako počet provedení (elementárních) kroků, které* M *pro daný vstup vykoná, než se zastaví.*
Časovou složitostí *Turingova stroje* M *rozumíme funkci* $T_M : \mathcal{N} \longrightarrow \mathcal{N}$*, kde* $T_M(n)$ *znamená délku výpočtu* M *nad vstupem velikosti* n *v nejhorším případě; tedy* $T_M(n) = max\{\, k \mid k$ *je délka výpočtu* M *nad (nějakým) vstupem velikosti* $n\,\}$*.*

Definice 25.0.6 Velikostí paměti *Turingova stroje* M *(potřebné při výpočtu) pro konkrétní vstup rozumíme počet buněk pásky, které jsou během výpočtu nad daným vstupem navštíveny.*
Paměťovou složitostí *(nebo též* prostorovou složitostí*) Turingova stroje* M *rozumíme funkci* $S_M : \mathcal{N} \longrightarrow \mathcal{N}$*, kde* $S_M(n)$ *znamená velikost potřebné paměti při výpočtu* M *nad vstupem velikosti* n *v nejhorším případě; tedy* $S_M(n) = max\{\, k \mid k$ *je velikost paměti potřebné při výpočtu* M *nad (nějakým) vstupem velikosti* $n\,\}$*.*

Připomeňme, že definice dříve zavedených tříd složitosti $\mathcal{T}(f)$, $\mathcal{S}(f)$ závisí na zvoleném referenčním modelu, kterým jsou v našem případě stroje RAM. Vzali-li bychom jako referenční model Turingovy stroje, dostaneme jiné třídy !

Poznámka. Intuitivně snadno vidíme, že Turingův stroj je 'pomalejší' než RAM už z důvodů jeho sekvenčního přístupu k jednotlivým buňkám pásky (na rozdíl od 'libovolného', tj. přímého přístupu v případě RAMu). Existují tedy např. problémy, které lze řešit RAMem s čas. složitostí $O(n)$ (a které tedy patří do $\mathcal{T}(n)$ definované vzhledem k RAMu), ale nelze je řešit Turingovým strojem s čas. složitostí $O(n)$ (a nepatřily by tedy do $\mathcal{T}(n)$, vzali-li bychom Turingovy stroje jako referenční model).

Cílem následujících úvah bude ovšem ilustrace faktu, že

> Turingovy stroje a RAMy jsou polynomiálně ekvivalentní.

Rozumíme tím, že

> Ke každému RAMu M existuje (dá se zkonstruovat) Turingův stroj M', který realizuje tutéž vstupně-výstupní funkci jako M (tj. řeší tentýž problém) a navíc pro příslušné časové a prostorové složitosti platí $T_{M'}(n) \leq (T_M(n))^{c_1}$, $S_{M'}(n) \leq (S_M(n))^{c_2}$ pro nějaké (malé) konstanty c_1, c_2.
> Totéž přitom platí i v opačném směru: ke každému Turingovu stroji M existuje RAM M' s příslušnými vlastnostmi.

Abychom to nahlédli, stačí si promyslet, že ke každému RAMu je možné (algoritmicky) zkonstruovat Turingův stroj, který jej *simuluje*; přitom dochází jen k 'malé ztrátě' z hlediska složitosti (odpovídající polynomu malého stupně) – zde znovu připomínáme úmluvu o uvažování logaritmické míry u RAMů. Naopak ke každému Turingovu stroji je možné (algoritmicky) zkonstruovat RAM, který jej *simuluje* s malou ztrátou z hlediska složitosti.

Uvedeným tvrzením ještě věnujeme několik odstavců; nicméně nepůjdeme do detailů – o nich předpokládáme, že by si je čtenář při svých programátorských zkušenostech snadno doplnil.

Především se zastavme u pojmu 'simulace'. Ten je jistě čtenáři intuitivně zřejmý. Podrobněji vysvětlit by se dal např. následovně.

Předpokládáme, že výpočet stroje nad zadaným vstupem lze chápat jako posloupnost konfigurací, kterými stroj (krok po kroku) prochází; začíná v počáteční konfiguraci určené zadaným vstupem a eventuálně skončí v jisté koncové konfiguraci s určitým výstupem – pokud jeho výpočet není nekonečný. $Con(M)$ nechť označuje množinu všech konfigurací stroje M.

Nyní lze vyjádření *'stroj M_1 je simulován strojem M_2'* vysvětlit takto:

> Existuje prostá funkce $cod : Con(M_1) \to Con(M_2)$ taková, že cod i cod^{-1} jsou (jednoduše) algoritmicky vyčíslitelné. Přitom pro libovolný výpočet $K_0, K_1, \ldots, K_m$ stroje M_1 prochází výpočet stroje M_2 začínající v $cod(K_0)$ postupně konfiguracemi $cod(K_0), cod(K_1), \ldots, cod(K_m)$ – s případnými 'mezikonfiguracemi'.

Např. lze snadno ukázat, jak lze (standardní) Turingův stroj simulovat Turingovým strojem s jen jednostranně nekonečnou páskou, jak lze vícepáskový Turingův stroj simulovat standardním Turingovým strojem, jak se lze omezit na případ, kde páskové symboly jsou pouze $0, 1, \square$ apod.

Vzájemnou (polynomiální) simulací RAMů a Turingových strojů se budeme podrobněji věnovat na cvičení.

Jen stručně zmíníme ještě jeden používaný výpočetní model – *stroje s čítači* ('counter machines'; podobné jsou 'register machines').

Stroj s čítači C má fixní počet (celočíselných nezáporných) čítačů $c_1, c_2, \ldots, c_m$ a jeho 'program' je posloupnost příkazů

$$1 : COMM_1;\ 2 : COMM_2;\ \ldots\ldots ;\ n : COMM_n$$

kde $COMM_n$ je instrukce $HALT$ a $COMM_i$ ($i = 1, 2, ..., n-1$) jsou příkazy následujících dvou typů (předp. $1 \leq k, k_1, k_2 \leq n$, $1 \leq j \leq m$)

1/ $c_j := c_j + 1;\ goto\ k$,

2/ $if\ c_j = 0\ then\ goto\ k_1\ else\ (c_j := c_j - 1;\ goto\ k_2)$.

Lze si představit, že jeden čítač je vyčleněn jako vstupní (vstupem je tedy jedno celé nezáporné číslo) a jeden je vyčleněn jako výstupní. Víc asi není k popisu modelu třeba dodávat.

Takto definovaný model *není* polynomiálně ekvivalentní Turingovým strojům; problém je v tom, že aritmetické operace s čísly (obsahy čítačů), jsou úměrné velikosti těchto čísel, nikoli velikosti jejich zápisu (což je, jak víme, 'exponenciální rozdíl'). Stačí ovšem přidat např. instrukce typu $c_j := c_j$ div 10 a $c_j := c_j * 10$; tyto modifikované stroje s čítači pak už *jsou* polynomiálně ekvivalentní Turingovým strojům.

Kapitola 26

Třída NPTIME. Polynomiální převeditelnost. NP-úplné problémy.

Stručný obsah: Příklady (běžných praktických) problémů, pro které jsou známy pouze exponenciální algoritmy. Nedeterministické Turingovy stroje. Třída NP (=NP-TIME); její robustnost vůči výpočetním modelům. Polynomiální převeditelnost problémů. NP-úplnost. NP-úplné problémy. Otevřená otázka, zda P=NP.

Cíl kapitoly:
Umět vysvětlit pojem NP-úplného problému a související pojmy (např. nedeterm. Turingův stroj) a dokazovat příslušnost k NP a NP-obtížnost u konkrétních problémů.

Studijní cíle: Umět vysvětlit pojem NP-úplného problému a související pojmy (např. nedeterm. Turingův stroj) a dokazovat příslušnost k NP a NP-obtížnost u konkrétních problémů.

Čtenář jistě zná spoustu algoritmů (některé jsme uvedli i v rámci tohoto kursu), které patří do třídy PTIME. Připomeňme, že tato je chápána jako (horní) aproximace třídy prakticky zvládnutelných problémů.

Nyní uvedeme příklady několika problémů, pro které jsou známy exponenciální algoritmy, ale není známo, zda patří či nepatří do PTIME, neboli zda pro ně existují či neexistují polynomiální algoritmy.

Mezi takové problémy dlouho patřil dříve uvedený *problém prvočíselnosti* (v létě 2002 byl zveřejněn důkaz příslušnosti k PTIME), nadále tam ovšem patří problém SAT (splnitelnost booleovských formulí) a následující problémy:

Název: TSP (*problém obchodního cestujícího*) (ANO/NE verze)

Vstup: množina 'měst' $\{1, 2, \ldots, n\}$, přir. čísla ('vzdálenosti') d_{ij} ($i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, n$); dále číslo ℓ ('limit').
Otázka: existuje 'okružní jízda' dlouhá nejvýše ℓ, tj. existuje permutace $\{i_1, i_2, \ldots, i_n\}$ množiny $\{1, 2, \ldots, n\}$ tž. $d(i_1, i_2)+d(i_2, i_3)+\ldots+d(i_{n-1}, i_n)+d(i_n, i_1) \leq \ell$?

Název: HC (*problém hamiltonovského cyklu*)

Vstup: orientovaný graf G.
Otázka: existuje v G hamiltonovský cyklus (tj. uzavřená orientovaná cesta, procházející každým vrcholem právě jednou) ?

Název: HK (*problém hamiltonovské kružnice*)

Vstup: neorientovaný graf G.
Otázka: existuje v G hamiltonovská kružnice (tj. uzavřená cesta, procházející každým vrcholem právě jednou) ?

Název: IS (*problém nezávislé množiny*)

Vstup: neorientovaný graf G (o n vrcholech); číslo k ($k \leq n$).
Otázka: existuje v G nezávislá množina velikosti k (tj. množina k vrcholů, z nichž žádné dva nejsou spojeny hranou) ?

Uvedené problémy jsou speciálního charakteru: jsou rozhodnutelné v polynomiálním čase *nedeterministickými* Turingovými stroji. (Jen stručně připomeňme, že u nedeterministického Turingova stroje může být v dané konfiguraci obecně možné provést několik vzájemně různých kroků – bezprostředně následující konfigurace tak není určena jednoznačně.)

Definice 26.0.7 *Daný problém P (typu ANO/NE) je rozhodován nedeterministickým Turingovým strojem M, jestliže všechny výpočty M jsou konečné a vydávají ANO nebo NE, a navíc platí:*

- *jestliže odpověď na otázku problému P pro vstup w je ANO, pak existuje (alespoň jeden) výpočet M nad w vydávající ANO,*
- *jestliže odpověď pro w je NE, pak všechny výpočty M nad w vydávají NE.*

Složitost nedeterministického Turingova stroje a příslušné třídy složitosti lze definovat takto:

Definice 26.0.8 Časová složitost *nedeterministického Turingova stroje M je zobrazení $T_M : \mathcal{N} \to \mathcal{N}$, kde $T_M(n)$ znamená maximální délku výpočtu pro vstup velikosti n.*
Třídou časové složitosti $N\mathcal{T}(f)$ *pro funkci $f : \mathcal{N} \longrightarrow \mathcal{N}$ rozumíme třídu těch problémů, které jsou řešeny nedeterministickými Turingovými stroji s čas. složitostí v $O(f)$.*

Čtenář si jistě snadno doplní definice pro prostorovou složitost.

Konzistentnější s předchozím textem by bylo, kdybychom při definování tříd $N\mathcal{T}(f(n))$ použili jako referenční model nedeterministické RAMy. Nám ovšem půjde především o třídu

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

tzv. problémů řešitelných v nedeterministickém polynomiálním čase. Její definice je podobně jako pro PTIME robustní (nezávislá na zvoleném ‘rozumném’ referenčním modelu).

Takto jsme se dostali k velmi známé dosud otevřené otázce, zda PTIME=NPTIME (dané otázce se často říká P-NP problém); přitom je ovšem zřejmé, že PTIME⊆NPTIME.

Uvedeme teď definici charakterizující 'nejtěžší' problémy v NPTIME. Pro tyto účely využijeme následující relaci $\triangleleft$ mezi problémy (v našem kontextu lze $P1 \triangleleft P2$ číst '$P1$ je nejvýš tak těžký jako $P2$'):

Definice 26.0.9 *Problém $P1$ je* polynomiálně převeditelný *na problém $P2$, označme $P1 \triangleleft P2$, jestliže existuje Tur. stroj M* s polynomiální časovou složitostí*, který pro libovolný vstup w problému $P1$ sestrojí vstup w' problému $P2$, přičemž platí, že odpověď na otázku problému $P1$ pro vstup w je stejná jako odpověď na otázku problému $P2$ pro vstup w'.*

Všimněme si následujících jednoduchých faktů:

- $((P1 \triangleleft P2) \wedge (P2 \triangleleft P3)) \Longrightarrow (P1 \triangleleft P3)$
- $((P1 \triangleleft P2) \wedge (P2 \in \text{PTIME})) \Longrightarrow (P1 \in \text{PTIME})$
- $((P1 \triangleleft P2) \wedge (P2 \in \text{NPTIME})) \Longrightarrow (P1 \in \text{NPTIME})$

Definice 26.0.10 *O problému P řekneme, že je* NP-těžký*, jestliže pro každý problém $P' \in$ NPTIME platí $P' \triangleleft P$. Je-li navíc $P \in$ NPTIME, říkáme, že P je* NP-úplný*.*

Definovali jsme pojem NP-úplných problémů (tj. 'nejtěžších' problémů v NPTIME), ale dosud jsme neukázali, že aspoň jeden takový problém vůbec existuje. Tuto existenci ukazuje následující věta.

Věta 26.0.11 *(Cook, 1971) Problém SAT je NP-úplný.*

Důkaz: Problém příslušnosti SAT k NPTIME je zřejmý (příslušný nedeterministický Turingův stroj prostě 'zvolí' nějaké pravdivostní ohodnocení proměnných v zadané formuli a ověří, zda při tomto ohodnocení je formule pravdivá; ke kladnému závěru má možnost dospět právě tehdy, když takové ohodnocení existuje).

Stačí tedy ukázat, že pro každý $P \in$ NPTIME platí $P \triangleleft$ SAT (připomeňme, že se omezujeme na ANO/NE problémy).

Uvažujme tedy libovolný, ale dále pevný, problém $P \in$ NPTIME. Ten je nutně rozhodován nedeterministickým Turingovým strojem M s časovou složitostí $T_M(n) \leq p(n)$ pro určitý polynom p. Je potřeba ukázat, že existuje polynomiální algoritmus (přesněji řečeno Turingův stroj s časovou složitostí omezenou polynomiální funkcí), který k libovolnému vstupu w problému P zkonstruuje booleovskou formuli $\mathcal{F}_w$ (v konjunktivní normální formě), která je splnitelná právě tehdy, když odpověď na otázku P pro w je ANO.

Máme-li ovšem dán vstup w velikosti n, pak k rozhodnutí o odpovědi na otázku P pro w stačí zjistit, zda existuje posloupnost konfigurací $C_0, C_1, C_2, \ldots, C_{p(n)}$, která představuje možný přijímající (tj. končící odpovědí ANO) výpočet stroje M na vstupu w; všimněme si, že velikost konfigurací je rovněž nutně omezena hodnotou $p(n)$.

Není těžké (i když je to technicky pracné) zkonstruovat formuli zachycující schéma takového výpočtu, která je splnitelná právě tehdy, když příslušná posloupnost konfigurací existuje. (Podrobněji bude konstrukce probrána na cvičení.)

□

Máme-li dokázánu NP-úplnost jednoho problému, je možné ji využít k důkazu NP-úplnosti (či NP-obtížnosti) problémů dalších:

Tvrzení 26.0.12 *Jestliže $P_1 \triangleleft P_2$ a P_1 je NP-těžký, pak P_2 je rovněž NP-těžký; když je navíc P_2 v NPTIME, je NP-úplný.*

Důkaz: Tvrzení plyne snadno z faktu, že složení dvou polynomiálních funkcí je opět polynomiální funkce—byť vyššího stupně; jinými slovy: relace $\triangleleft$ je tranzitivní.

□

Demonstrováním příslušných převeditelností ukážeme na přednášce NP-úplnost několika již uvedených a dalších problémů. Např. ukážeme:

- SAT $\triangleleft$ IS
- IS $\triangleleft$ HC (referováno na cvičení)
- HC $\triangleleft$ HK
- HK $\triangleleft$ TSP
- SAT $\triangleleft$ 3-SAT
- 3-SAT $\triangleleft$ 3-CG

kde dosud nezmíněné problémy jsou definovány takto:

> *Název*: 3-SAT (*problém SAT s omezením na 3 literály*)
>
> *Vstup*: booleovská formule v konjunktivní normální formě, kde v každé klauzuli (tj. v každém konjunktu) jsou právě 3 literály (literál je buď proměnná nebo její negace).
> *Otázka*: je daná formule splnitelná (tj. existuje pravdivostní ohodnocení proměnných, při kterém je formule pravdivá) ?

> *Název*: 3-CG (*problém barvení grafu třemi barvami*)
>
> *Vstup*: neorientovaný graf $G = (V, E)$.
> *Otázka*: lze G obarvit třemi barvami, tzn. existuje zobrazení $c : V \to \{col_1, col_2, col_3\}$ takové, že $\forall\{v_1, v_2\} \in E : c(v_1) \neq c(v_2)$?

Uvažujme ještě problém

> *Název*: ILP (*problém celočíselného lineárního programování*)
>
> *Vstup*: matice A typu $m \times n$ a sloupcový vektor b velikosti m, jejichž prvky jsou celá čísla.
> *Otázka*: existuje celočíselný sloupcový vektor x (velikosti n) tž. $Ax \geq b$?

Jedná se rovněž o NP-úplný problém. Snadno se ukáže, že je NP-těžký (např. převodem 3-SAT $\triangleleft$ ILP), ale na rozdíl od dříve uvedených problémů je obtížnější prokázat, že ILP $\in$ NPTIME. Zhruba řečeno, dá se ukázat, že pokud řešení nerovnosti $Ax \geq b$ existuje, existuje i řešení 'dostatečně malé'—jeho zápis je polynomiální vzhledem k zápisu A a b ; řešení se tedy dá v polynomiálním čase 'uhodnout' a ověřit.

Otázka P$\stackrel{?}{=}$NP

Když se hovoří o tzv. P-NP problému (slovo ‘problém’ zde není použito v našem technickém smyslu !), rozumí se tím otevřená otázka, zda PTIME je vlastní podtřídou NPTIME či zda jsou si tyto třídy rovny.

Obecně se má zato, že pro NP-úplné problémy neexistují polynomiální algoritmy; ovšem nikdo to zatím nedokázal. Všimněme si, že kdyby někdo objevil polynomiální algoritmus pro jeden NP-úplný problém, existovaly by polynomiální algoritmy pro všechny tyto problémy. Naopak když by někdo prokázal, že pro jeden konkrétní NP-úplný problém neexistuje polynomiální algoritmus, neexistoval by takový algoritmus pro žádný z NP-úplných problémů.

V praxi se tedy bere prokázání NP-úplnosti (či vlastně NP-obtížnosti) jako důkaz nezvládnutelnosti problému—trváme-li na zaručeném nalezení (nejlepšího možného) řešení. V úvahu pak přicházejí např. *aproximační algoritmy* (u optimalizační úlohy se např. může podařit sestavit rychlý algoritmus, který zaručeně nalezne řešení, jež je nejvýše dvakrát horší než optimální) či *pravděpodobnostní algoritmy* (využívají ‘házení kostkou’ a dávají rychle odpovědi, které však mohou být s určitou pravděpodobností nesprávné; jejich vícenásobným opakováním se ale dá docílit, že pravděpodobnost nesprávného výsledku je mizivá) – příklady takových algoritmů uvedeme v závěru kursu.

Kapitola 27

Další třídy časové i prostorové složitosti

Stručný obsah: Třídy PSPACE, EXPTIME, EXPSPACE. Idea rovnosti PSPACE=NPSPACE (Savitchova věta). Příklady PSPACE-úplných problémů. Dokazatelně nezvládnutelné problémy – příklady problémů s exponenciální či vyšší složitostí.

Cíl kapitoly:
Umět vysvětlit probrané třídy složitosti, ilustrovat je příklady, vysvětlit pojem dokazatelně nezvládnutelných problémů.

Studijní cíle: Umět vysvětlit probrané třídy složitosti, ilustrovat je příklady, vysvětlit pojem dokazatelně nezvládnutelných problémů.

Třída PSPACE

Předmětem našeho prvořadého zájmu je časová složitost algoritmů a problémů. Už v případě konkrétních algoritmů a problémů ovšem může mít dobrý smysl zkoumat také prostorovou (tj. paměťovou) složitost a speciálně vztah časové a prostorové složitosti (daný algoritmus může jít např. zrychlit jen za cenu zvýšení paměťové náročnosti a naopak).

Strukturální složitost samozřejmě zkoumá i třídy problémů vymezené prostorovou složitostí. Čtenář si jistě snadno doplní definice tříd PSPACE a NPSPACE a všimne si zřejmé inkluze PSPACE $\subseteq$ NPSPACE. Pro tyto třídy se ovšem ví, že platí i inkluze obrácená (a je tedy PSPACE $=$ NPSPACE); to ihned plyne z následující věty (jen poznamenejme, že věta platí obecněji—pro naše účely však postačuje uvedené znění):

Věta 27.0.13 *(Savitch, 1970) Je-li problém P rozhodován nedeterministickým Turingovým strojem s prostorovou složitostí $O(n^k)$, pak je také rozhodován deterministickým Turingovým strojem s prostorovou složitostí $O(n^{2k})$.*

Důkaz: Nechť problém P je rozhodován nedeterministickým Turingovým strojem M_1 s prostorovou složitostí nejvýše $c_1 n^k$ (pro nějaké konstanty c_1, k). Všimněme si, že pro vstup w velikosti n může stroj M_1 vydat odpověď ANO právě tehdy, když existuje posloupnost konfigurací $C_0, C_1, C_2, \ldots, C_m$ popisující příslušný výpočet; velikost každé konfigurace je nejvýše rovna $c_1 n^k$. Takových konfigurací je ovšem nejvýše $c^{c_1 n^k}$ pro vhodně zvolené c (jako c lze zvolit součet počtu symbolů abecedy a počtu stavů stroje M_1). Jelikož je zbytečné, aby se v uvedené posloupnosti konfigurací nějaká konfigurace opakovala, stačí uvažovat jen $m \leq c^{c_1 n^k}$.

Mělo by teď být jasné, jak lze M_1 simulovat deterministickým Turingovým strojem M_2 s prostorem $c_1 n^k \cdot c^{c_1 n^k}$ (ten prostě zkouší systematicky všechny možnosti a pro každou z nich zjišťuje, zda se jedná o zápis hledané posloupnosti $C_0, C_1, C_2, \ldots, C_m$).

Toto ovšem nestačí, neboť prostor používaný strojem M_2 je exponenciální. Základní idea ušetření prostoru spočívá v tom, že úkol ověření, zda z konfigurace C lze dosáhnout konfiguraci C' za 2^ℓ kroků se dá řešit systematickým generováním ('prostředních') konfigurací C'' a ověřováním, zda z C lze dosáhnout C'' za $2^{\ell-1}$ kroků a z C'' lze dosáhnout C' za $2^{\ell-1}$ kroků. K ověření zmíněných dvou podúkolů je ovšem možné použít tentýž prostor ! Uplatníme-li tuto myšlenku rekurzivně, není těžké vyvodit, že celkový potřebný prostor bude u upraveného M_2 nejvýše $c_1 n^k \cdot c_2 n^k$ pro vhodnou konstantu c_2 a tedy prostorová složitost M_2 je pak $O(n^{2k})$.

Detailněji bude důkaz probrán na cvičení.

□

Připomeňme, že pro lib. funkci f je $\mathcal{T}(f(n)) \subseteq \mathcal{S}(f(n))$ (např. Turingův stroj očividně navštíví při výpočtu nejvýše tolik políček, kolik udělá kroků). Měl by teď už být zřejmý vztah

$$PTIME \subseteq NPTIME \subseteq PSPACE = NPSPACE.$$

Přes velké úsilí vědecké komunity, nemůžeme dosud vyloučit nejen možnost $PTIME = NPTIME$, ale dokonce ani $PTIME = PSPACE$, byť se tyto možnosti jeví velmi 'nepravděpodobnými'.

Podobně jako u NP-úplnosti, lze definovat tzv. PSPACE-úplné problémy; definici napíšeme obecněji:

Definice 27.0.14 *O problému P řekneme, že je $\mathcal{C}$-těžký, kde $\mathcal{C}$ je nějaká třída problémů, jestliže pro každý $P' \in \mathcal{C}$ platí $P' \triangleleft P$. Je-li navíc $P \in \mathcal{C}$, říkáme, že P je $\mathcal{C}$-úplný.*

Známým příkladem PSPACE-úplného problému je problém

> *Název*: QBF (*problém pravdivosti kvantifikovaných booleovských formulí*)
>
> *Vstup*: formule $(\exists x_1)(\forall x_2)(\exists x_3)(\forall x_4)\ldots(\exists x_{2n-1})(\forall x_{2n})\mathcal{F}(x_1, x_2, \ldots, x_{2n})$, kde $\mathcal{F}(x_1, x_2, \ldots, x_{2n})$ je booleovská formule v konjunktivní normální formě.
> *Otázka*: je daná formule pravdivá ?

Převeditelností z tohoto problému lze např. ukázat PSPACE-obtížnost různých deskových her.

Dokazatelně nezvládnutelné problémy

Jestliže prokážeme NP-obtížnost či PSPACE-obtížnost nějakého problému, říkáme, že je *prakticky nezvládnutelný* (intractable). Je totiž jasné, že navrhneme-li algoritmus, který řeší (přesně ten) daný problém, a neobjevíme-li přitom geniální 'trik', na který dosud nikdo nepřišel, bude mít algoritmus zřejmě exponenciální (či ještě horší) složitost. Obecně používat algoritmus (resp. odpovídající program) budeme moci jen na velmi malá vstupní data. Teoreticky ovšem pořád ještě možnost rychlého algoritmu (založeného na 'geniálním triku') existuje.

Poznámka. Samozřejmě je možné, že exponenciální algoritmus ve skutečnosti chceme použít jen na malá data, anebo ho třeba používáme na data, při nichž se neprojeví ona (worst case) exponenciální složitost. V tomto smyslu praktická nezvládnutelnost (obecného) problému ještě neznamená, že ho v praxi nemůže počítačový program úspěšně řešit v pro nás zajímavých případech. (Existují i jiné možnosti, jak v praxi úspěšně řešit i 'nezvládnutelný' problém. Zmíníme se o tom v partiích o aproximačních a pravděpodobnostních algoritmech.)

Známe ovšem i tzv. *dokazatelně nezvládnutelné* problémy (provably intractable problems), tj. ty, u nichž máme *dokázáno*, že pro ně neexistují polynomiální algoritmy. Definujeme-li např. třídy

$$\text{EXPTIME} = \bigcup_{k=0}^{\infty} \mathcal{T}(2^{n^k}) \,, \qquad \text{EXPSPACE} = \bigcup_{k=0}^{\infty} \mathcal{S}(2^{n^k})$$

pak EXPTIME-těžký či EXPSPACE-těžký problém je takovým dokazatelně nezvládnutelným problémem.

Dá se totiž ukázat, že inkluze PTIME$\subset$EXPTIME, PSPACE$\subset$EXPSPACE jsou skutečně vlastní (tzn., že neplatí rovnost). (My to zde ale dokazovat nebudeme.)

Např. následující problém je EXPSPACE-úplný:

> *Název*: RE^2 (*ekvivalence regulárních výrazů s mocněním*)
>
> *Vstup*: dva regulární výrazy, v nichž je možné použít mocnění (tzn. je možno psát α^2 místo $\alpha \cdot \alpha$).
> *Otázka*: reprezentují zadané výrazy tentýž jazyk ?

Poznamenejme, že stejný problém pro standardní regulární výrazy (bez mocnění) je PSPACE-úplný. (Připomeňme si, že složitost je funkcí velikosti vstupu; mocnění v reg. výrazech umožňuje exponenciálně zkrátit některé výrazy bez mocnění.) Čtenář by si měl být schopen vyvodit, že problém ekvivalence dvou *nedeterministických* konečných automatů je tedy také PSPACE-úplný. (Pro deterministické konečné automaty ovšem samozřejmě známe polynomiální algoritmus; připomeňme si, že přechodem od nedeterministického automatu k deterministickému se obecně nemůžeme vyhnout exponenciálnímu nárustu počtu stavů.)

Existují samozřejmě i dokazatelně těžší (superexponenciální) problémy. Ilustrujme je příkladem problému Presburgerovy aritmetiky (rozhodování pravdivosti formulí teorie sčítání).

> *Název*: ThAdd (*problém pravdivosti teorie sčítání*)
>
> *Vstup*: formule jazyka 1. řádu užívající jediný 'nelogický' symbol – ternární (tj. 3-ární) predikátový symbol $PLUS$ ($PLUS(x,y,z) \Leftrightarrow_{df} x + y = z$).
> *Otázka*: je daná formule pravdivá pro množinu $\mathcal{N} = \{0, 1, 2, \ldots\}$, kde $PLUS(a,b,c)$ je interpretováno jako $a + b = c$?

Zde vůbec není zřejmé, že existuje algoritmus, který daný problém řeší. Presburger ukázal takový algoritmus ve 20. létech 20. století (jedním z cílů tzv. Hilbertova programu bylo ukázat podobný algoritmus i s připuštěním predikátu pro násobení—nemožnost řešení tohoto úkolu ukázal později Gödel). Mnohem později bylo ukázáno, že každý algoritmus, řešící problém ThAdd má složitost minimálně 2^{2^n}.

Presburger ukázal algoritmus využitím tzv. metody eliminace kvantifikátorů. Výsledky teorie konečných automatů umožňují podat důkaz následující věty velice elegantně:

Věta 27.0.15 *Existuje algoritmus rozhodující problém ThAdd.*

Důkaz: (Idea.) Představme si třístopou pásku, kde v každé stopě je řetězec nul a jedniček, tj. binární zápis čísla. Na pásku samozřejmě můžeme hledět jako na jednostopou s tím, že povolené *symboly abecedy* jsou uspořádáné *trojice* nul a jedniček. Snadno nahlédneme, že existuje konečný automat, který přijímá právě ta slova v 'abecedě trojic', která mají tu vlastnost, že součet čísla v první stopě s číslem v druhé stopě je roven číslu ve třetí stopě. Ihned je to jasné při čtení odzadu; pak si stačí připomenout, že regulární jazyky jsou uzavřeny na zrcadlový obraz.

Z dalších uzávěrových vlastností snadno vyvodíme, že pro lib. formuli $\mathcal{F}(x_1, x_2, \ldots, x_n)$ jazyka ThAdd která *neobsahuje kvantifikátory*, lze zkonstruovat kon. automat $A_{\mathcal{F}}$ přijímající právě binární zápisy těch n-tic čísel (na n-stopé pásce), pro které je $\mathcal{F}(x_1, x_2, \ldots, x_n)$ pravdivá.

Pro formuli $(\exists x_n)\mathcal{F}(x_1, x_2, \ldots, x_n)$ je možné zkonstruovat automat, který přijímá právě binární zápisy těch $(n-1)$-tic čísel (dosazených za $x_1, x_2, \ldots, x_{n-1}$), pro které je $(\exists x_n)\mathcal{F}(x_1, x_2, \ldots, x_n)$ pravdivá: automat pracuje na $(n-1)$-stopé pásce, ale simuluje $A_{\mathcal{F}}$ tak, že obsah n-té stopy nedeterministicky hádá ! (Pak ho samozřejmě lze převést na ekvivalentní deterministický automat.)

Jelikož $(\forall x_n)\mathcal{F}(x_1, x_2, \ldots, x_n)$ je ekvivalentní $\neg(\exists x_n)\neg\mathcal{F}(x_1, x_2, \ldots, x_n)$, načrtli jsme takto postup, který k formuli jazyka ThAdd v prenexní formě postupnou aplikací zmíněných konstrukcí sestrojí konečný automat, který přijme prázdné slovo právě tehdy, když výchozí formule je pravdivá.

□

Kapitola 28

Rozhodnutelnost a nerozhodnutelnost problémů

Stručný obsah: (Algoritmická) rozhodnutelnost, nerozhodnutelnost a částečná rozhodnutelnost problémů. Church-Turingova teze (algoritmus = Turingův stroj). Rekurzivní a rekurzivně spočetné množiny (jazyky). Rekurzivní a částečně rekurzivní funkce. Idea univerzálního algoritmu (Turingova stroje).

Cíl kapitoly:
Umět vysvětlit pojem rozhodnutelného, nerozhodnutelného a částečně rozhodnutelného problému a objasnit roli Church-Turingovy teze. Vše umět ilustrovat na příkladech problémů. Umět vysvětlit konstrukci univerzálního algoritmu.

Studijní cíle: Umět vysvětlit pojem rozhodnutelného, nerozhodnutelného a částečně rozhodnutelného problému a objasnit roli Church-Turingovy teze. Vše umět ilustrovat na příkladech problémů. Umět vysvětlit konstrukci univerzálního algoritmu.

Pro námi dosud uvažované problémy vždy existoval algoritmus, který příslušný problém řeší (rozhoduje). Chceme-li pojem algoritmické řešitelnosti (či rozhodnutelnosti) problémů uvést přesněji, lze např. podat tuto definici:

Definice 28.0.16 *Problém* $P = (IN, OUT, p)$ *(*$p : IN \rightarrow OUT$*) je* algoritmicky řešitelný*, jestliže existuje algoritmus, který pro libovolný vstup* $w \in IN$ *skončí a vydá jako výsledek* $p(w)$*. Jedná-li se o problém typu ANO/NE, říkáme, že je* algoritmicky rozhodnutelný*, nebo stručněji* rozhodnutelný*.*

Všimněme si, že se implicitně předpokládá, že vstupy a výstupy jsou ‘finitní objekty’ (či jsou takto kódovány); už jsme hovořili o tom, že stačí uvažovat kódování vstupů a výstupů řetězci symbolů z nějaké konečné abecedy.

Pojem algoritmické rozhodnutelnosti či algoritmické vyčíslitelnosti (řešitelnosti) lze přirozeně definovat např. pro množiny přir. čísel, jazyky, či funkce:

Definice 28.0.17 *Množina $\mathcal{M} \subseteq \mathcal{N}$ je* rozhodnutelná*, jestliže problém příslušnosti k $\mathcal{M}$ (Vstup: $n \in \mathcal{N}$; otázka: platí $n \in \mathcal{M}$?) je rozhodnutelný.*
Jazyk L v abecedě Σ (tedy $L \subseteq \Sigma^*$) je rozhodnutelný*, jestliže problém příslušnosti k L (Vstup: $w \in \Sigma^*$; otázka: platí $w \in L$?) je rozhodnutelný.*
Funkce $f : \mathcal{N} \to \mathcal{N}$ je algoritmicky *(někdy též* efektivně*)* vyčíslitelná*, jestliže 'problém výpočtu jejích hodnot' (Vstup: $n \in \mathcal{N}$; výstup $f(n)$) je algoritmicky řešitelný.*

Pojmy definované v předchozích definicích se odvolávaly k pojmu 'algoritmus'. Nahradíme-li v nich pojem 'algoritmus' pojmem 'Turingův stroj', uvádíme u definovaných pojmů výraz 'rekurzivní':

Definice 28.0.18 *Problém typu ANO/NE je* rekurzivní*, jestliže je rozhodován Turingovým strojem. Podobně zavádíme pojmy* rekurzivní jazyk*,* rekurzivní množina*,* rekurzivní funkce*.*

Poznámka. Pojem 'rekurzivní' je v této souvislosti ustálen z historických důvodů, které zde nebudeme rozebírat. Jen poznamenejme, že např. pojem 'rekurzivní funkce' nelze ztotožňovat s týmž pojmem v programovacích jazycích.

Jak ale čtenář zřejmě očekává, máme velmi dobré důvody přijímat následující tezi (axiom):

Church-Turingova teze

> *Ke každému algoritmu je možné zkonstruovat s ním ekvivalentní Turingův stroj (při vhodném vyjádření vstupů a výstupů jako řetězců v určité abecedě); ekvivalencí zde rozumíme podmínku, že algoritmus i Turingův stroj se zastaví (tj. jejich běh, výpočet, se zastaví) právě pro tytéž vstupy, přičemž pro tytéž vstupy budou příslušné výstupy totožné.*

Důsledek 28.0.19 *Při přijetí Church-Turingovy teze lze ztotožňovat pojmy* rekurzivní *a* rozhodnutelný *(v případě (totální) funkce pojmy* rekurzivní *a* algoritmicky vyčíslitelná*).*

Jelikož pojem 'algoritmus' bereme jako pojem základní (podobně jako např. pojem 'množina') a nikoli odvozený (tj. definovaný pomocí základních a z nich odvozených pojmů), nelze Church-Turingovu tezi dokázat jako matematickou větu. Všimněte si ovšem, že v obráceném směru je platnost teze zřejmá. Tyto úvahy jsou už spíše logicko-filozofické povahy a zde je nebudeme dále rozebírat.

Pro nás je zde podstatné, že Church-Turingova teze, stručně řečeno, ztotožňuje pojmy algoritmus a Turingův stroj. Přitom (zatím) naše zkušenost a mnohé výsledky teorie vyčíslitelnosti potvrzují, že přijímání teto teze je rozumné.

Brzy ukážeme důkaz algoritmické nerozhodnutelnosti určitého problému (problému zastavení). Ve skutečnosti ovšem dokážeme, že tento problém není turingovsky rozhodnutelný, tj. není rekurzivní. Že je v tom případě (algoritmicky) nerozhodnutelný, vyplývá z Church-Turingovy teze; to se v takové souvislosti většinou explicitně neuvádí, ale měli bychom to mít na paměti.

Nejprve ale uvedeme definici širší třídy než je třída rozhodnutelných problémů:

Definice 28.0.20 *Problém typu ANO/NE je* částečně rozhodnutelný*, jestliže existuje algoritmus, který skončí právě pro ty vstupy problému, na něž je odpověď ANO. (Pro vstupy s odpovědí NE je běh algoritmu nekonečný).*
Podobně definujeme pojmy částečně rozhodnutelný jazyk*,* částečně rozhodnutelná množina*.*

Je zřejmé, že každý rozhodnutelný problém je i částečně rozhodnutelný (proč ?). Za chvíli uvidíme, že naopak to neplatí.

Ještě uvedeme analogický pojem pro funkce:

Definice 28.0.21 *Částečná funkce $\phi : \mathcal{N} \to \mathcal{N}$ ($Dom(\phi) \subseteq \mathcal{N}$) je* algoritmicky *(někdy též* efektivně*)* vyčíslitelná*, jestliže existuje algoritmus, jenž přijme jako vstup libovolné $n \in \mathcal{N}$, zastaví se právě tehdy, když $n \in Dom(\phi)$, a v tom případě vydá $\phi(n)$.*

Nahradíme-li opět pojem 'algoritmus' pojmem 'Turingův stroj', dostaneme definice *rekurzivně spočetného jazyka*, *rekurzivně spočetné množiny*, *částečně rekurzivní funkce*. Za předpokladu Church-Turingovy teze můžeme ovšem opět provést příslušná ztotožnění pojmů.

Určitý vztah mezi rozhodnutelností a částečnou rozhodnutelností uvádí následující věta (zformulujte si ji i pro ANO/NE problémy):

Věta 28.0.22 *(Post) Množina $A \subseteq \Sigma^*$ je rozhodnutelná právě když A i $\overline{A}$ jsou částečně rozhodnutelné.*

Důkaz: Důkaz je vcelku přímočarý (promyslete jej); hlavní myšlenka spočívá v tom, že k dvěma algoritmům (Turingovým strojům) lze zkonstruovat algoritmus (Turingův stroj), který je provádí 'paralelně', tj. 'střídavě sekvenčně'.

□

Univerzální algoritmus

Uvažujme na chvíli pračku. Je to vlastně zařízení, které realizuje určitý algoritmus. Podobně třeba automobil atd. A co počítač ? Ten také realizuje určitý algoritmus. Hlavní činnost ('proceduru') tohoto algoritmu lze ovšem nazvat *Proveď zadaný algoritmus (neboli program)* pro zadaný vstup (neboli data). Proto ten algoritmus realizovaný počítačem je vlastně

univerzální algoritmus

(schopný realizovat jakýkoli zadaný algoritmus). Formálně existenci takového univerzálního algoritmu potvrzuje následující věta. (Důkaz, tj. konstrukci univ. stroje, podrobně probereme na cvičení.)

> Značení $!M(w)$ znamená "stroj M se na vstup w zastaví" (tzn. jeho výpočet skončí, je-li v počáteční konfiguraci na pásce slovo slovo w).

Věta 28.0.23 *(O* univerzálním Turingovu stroji*.) Lze sestrojit Turingův stroj U takový, že pro libovolný Turingův stroj M s abecedou $\{0, 1, \Box\}$ a libovolné $w \in \{0, 1\}^*$ platí:*
1/ $!M(w) \Leftrightarrow !U(Kod(M) \cdot w)$ *a*
2/ jestliže $!M(w)$*, pak* $M(w) = U(Kod(M) \cdot w)$*.*

Kapitola 29

Nerozhodnutelné problémy

Stručný obsah: Metoda diagonalizace a důkaz nerozhodnutelnosti problému zastavení (otázky, zda $!M(w)$). (Algoritmická) převeditelnost problémů; využití pro důkaz (ne)rozhodnutelnosti. Postův korespondenční problém a aplikace na problémy z teorie (bezkontextových) jazyků. Problémy z logiky. Riceova věta o nerozhodnutelnosti netriviálních vstupně-výstupních vlastností algoritmů (programů).

Cíl kapitoly:
Zvládnout prokazování nerozhodnutelnosti (a případně částečné rozhodnutelnosti) problémů v probraných a podobných příkladech. Umět aplikovat Riceovu větu.

Studijní cíle: Zvládnout prokazování nerozhodnutelnosti (a případně částečné rozhodnutelnosti) problémů v probraných a podobných příkladech. Umět aplikovat Riceovu větu.

Vzpomeňme si, že u NP-úplných problémů jsme nejprve dokázali NP-úplnost jednoho (konkrétně problému SAT) a pak jsme pomocí polynomiální převeditelnosti prokazovali NP-úplnost (resp. NP-obtížnost) dalších problémů.

Podobná situace je u nerozhodnutelných problémů. Když prokážeme nerozhodnutelnost jednoho, k prokázání nerozhodnutelnosti dalších už (většinou) stačí vhodně aplikovat tzv. algoritmickou převeditelnost.

Roli prvního (základního) nerozhodnutelného problému hraje problém zastavení:

Název: HP (*Halting problem*)

Vstup: Turingův stroj M (resp. jeho kód $Kod(M)$) s abecedou $\{0,1,\Box\}$ a slovo $w \in \{0,1\}^*$.
Otázka: zastaví se M na w (tj. platí $!M(w)$) ?

Věta 29.0.24 *Problém HP je nerozhodnutelný.*

Důkaz: Připomeňme nejprve, že Turingovy stroje lze přímočaře kódovat řetězci nul a jedniček, tedy $Kod(M) \in \{0,1\}^*$.

Důkaz je vedený sporem. Předpokládejme, že ex. Tur. stroj H, který se pro lib. vstup $u \in \{0,1\}^*$ tvaru $u = Kod(M) \cdot w$ (pro nějaký Tur. stroj M a slovo w) zastaví a rozhodne, zda $!M(w)$ či nikoliv (skončí např. buď ve spec. stavu q_{ANO} nebo v q_{NE}). U stroje H je samozřejmě možné také předpokládat pouze abecedu $\{0,1,\Box\}$.

Sestrojme nyní stroj D s abecedou $\{0,1,\Box\}$, který se chová následovně: vstupní slovo $v \in \{0,1\}^*$ nejprve zdvojí (vytvoří slovo vv) a na to 'spustí' (jako podprogram) stroj H. Jestliže (podprogram) H skončí ve stavu q_{ANO}, stroj D přejde do nekonečného cyklu (a tedy se nezastaví); jestliže H skončí ve stavu q_{NE}, stroj D se zastaví (stav q_{NE} bude také jeho koncovým stavem).

Když ovšem nyní prozkoumáme, zda se D při spuštění na svůj vlastní kód $Kod(D)$ zastaví či nezastaví, dospějeme při obou možnostech k logickému sporu (zastaví se a nezastaví se současně).

□

Další věta plyne snadno z předchozí věty, věty o univerzálním Tur. stroji a Postovy věty:

Věta 29.0.25 *Problém HP je částečně rozhodnutelný, jeho doplňkový problém není (ani) částečně rozhodnutelný.*

Všimněme si, že v důkazu nerozhodnutelnosti problému zastavení (HP) jsme vlastně dokázali nerozhodnutelnost speciálního podproblému, který označíme

DHP *(Diagonal Halting Problem)*

Vstup: Stroj M daný svým kódem $Kod(M)$;

Otázka: Zastaví se M na svůj kód (tj. na slovo $Kod(M)$) ?

Poznámka. Metoda důkazu je v principu aplikací tzv. Cantorovy diagonalizační metody, jež má v teorii vyčíslitelnosti časté použití (Cantor ji mj. použil na důkaz toho, že neexistuje bijekce mezi množinou přirozených a množinou reálných čísel).

Máme-li dokázánu nerozhodnutelnost jednoho problému, je možno ji využít k prokázání nerozhodnutelnosti dalších problémů. Např. z nerozhodnutelnosti problému P ihned plyne nerozhodnutelnost jeho doplňkového problému (ANO, NE přehozeny). Více užitečný je ovšem následující pojem:

Definice 29.0.26 *Problém $P1$ je* (algoritmicky) převeditelný *na problém $P2$, označme $P1 \to P2$, jestliže existuje algoritmus A, který pro libovolný vstup w problému $P1$ sestrojí (tzn. skončí svůj výpočet a jako výstup vydá) vstup $P2$, označme jej $A(w)$, přičemž platí, že odpověď na otázku problému $P1$ pro vstup w je ANO právě tehdy, když odpověď na otázku problému $P2$ pro instanci $A(w)$ je ANO.*

Poznámka. Čtenáře jistě nepřekvapí, že pojem rekurzivní převeditelnosti se definuje obdobně s tím, že pojem algoritmus se nahradí pojmem Turingův stroj. Připomeňme, že při přijetí Church-Turingovy teze jsou pojmy rekurzivní a algoritmické převeditelnosti totožné.

Užitečnost uvedeného pojmu pro naše účely vyslovuje následující tvrzení, jehož důkaz by měl být zřejmý (srovnejte se situací u NP-úplných, či NP-těžkých, problémů):

Tvrzení 29.0.27 *Je-li $P1 \to P2$ a problém $P1$ je nerozhodnutelný, je i problém $P2$ nerozhodnutelný.*

Příklad. Takto se např. prokáže nerozhodnutelnost problému UHP (Uniform Halting Problem. Vstup: Tur. stroj M; Otázka: Zastaví se M na každý vstup (tj. platí $\forall w$: $!M(w)$?.) Při prokázání $HP \rightarrow UHP$ stačí navrhnout algoritmus, který k zadanému M, w sestrojí stroj M', jenž nejdříve otestuje vstup a v případě, že jde o w, 'spustí' (podprogram) M a v opačném případě se ihned zastaví.

Uvedeme příklad jiného důležitého nerozhodnutelného problému:

> *Problém PKP* (Postův korespondenční problém)
>
> *Vstup:* dvojice seznamů $u_1, u_2, \ldots, u_n$ a $v_1, v_2, \ldots, v_n$ (pro něj. $n \geq 1$) neprázdných řetězců (slov) v nějaké abecedě.
>
> *Otázka:* má PKP pro danou instanci řešení , tj. existují indexy $i_1, i_2, \ldots, i_r$, $r > 0$, tak, že $u_{i_1} u_{i_2} \ldots u_{i_r} = v_{i_1} v_{i_2} \ldots v_{i_r}$? (Jestliže $i_1 = 1$, hovoříme o iniciálním řešení.)

Důkaz nerozhodnutelnosti problému PKP lze provést např. prokázáním převeditelností $HP \rightarrow IPKP \rightarrow PKP$, kde problém $IPKP$ je zadán obdobně jako PKP, jen otázka se ptá, zda existuje iniciální řešení pro daný vstup. Hlavní myšlenka převeditelnosti $HP \rightarrow IPKP$ spočívá v následujícím: k danému M, w se sestrojí první členy seznamů $u_1 = \$$, $v_1 = \$q_0 w\$$ (kde q_0 je poč. stav M). Další dvojice se volí tak, aby jediná možná cesta k získání iniciálního řešení spočívala v určité simulaci výpočtu M na w s tím, že řešení existuje právě tehdy, když M se zastaví na w.

(Podrobněji bude probráno na cvičení).

PKP se dá užít k důkazu nerozhodnutelnosti některých problémů v teorii formálních jazyků. Např. lze PKP snadno převést na následující problém:

> *Instance:* dvě bezkontextové gramatiky G_1, G_2
>
> *Otázka:* Platí $L(G_1) \cap L(G_2) \neq \emptyset$? (Tzn. 'Lze nějaké slovo vygenerovat oběma gramatikami ?')

Myšlenka převodu je jednoduchá:

K instanci $u_1, u_2, \ldots, u_n$, $v_1, v_2, \ldots, v_n$ problému PKP sestrojíme gramatiky

$$G_1 : S \rightarrow u_1 S a_1 \,|\, u_2 S a_2 \,|\, \ldots \,|\, u_n S a_n \,|\, u_1 a_1 \,|\, u_2 a_2 \,|\, \ldots \,|\, u_n a_n$$

$$G_2 : S \rightarrow v_1 S a_1 \,|\, v_2 S a_2 \,|\, \ldots \,|\, v_n S a_n \,|\, v_1 a_1 \,|\, v_2 a_2 \,|\, \ldots \,|\, v_n a_n$$

kde $a_1, a_2, \ldots, a_n$ jsou nově přidané symboly.

Podobně se dá dokázat, že pro bezkontextové gramatiky je nerozhodnutelným problémem otázka '$L(G) = \Sigma^*$?', tedy i otázka '$L(G_1) = L(G_2)$?' apod.

Poznámka. (Ne)rozhodnutelnost je také důležitou zkoumanou vlastností u logických teorií. Už jsme viděli příklad rozhodnutelné Presburgerovy aritmetiky (ThAdd). Zde jen poznamenejme, že přidáme-li vedle relačního symbolu PLUS ještě symbol MULT (s analogickou interpretací pro násobení), problém zjišťování pravdivosti formulí v množině přirozených čísel je pak nerozhodnutelný (což byl velmi významný a překvapivý výsledek dosažený Gödelem a Churchem ve třicátých létech dvacátého století).

Riceova věta

Na závěr této části uvedeme důležitou větu, jež ukazuje nerozhodnutelnost celé třídy problémů. Vyslovíme ji nejdříve v poněkud neformálním znění; slovo *algoritmus* zde 'pro praktičtější vyznění' nahrazujeme slovem *program* (což je algoritmus zapsaný v nějakém programovacím jazyku).

> *Jakákoli netriviální vlastnost programů týkající se výhradně jejich vstupně/výstupního chování je nerozhodnutelná (tj. množina všech programů s danou vlastností je nerozhodnutelná).*

Rozumí se, že vlastnost V je vstupně/výstupní právě tehdy, když každé dva programy, které realizují totéž vstupně/výstupní zobrazení (převádějí stejné vstupy na stejné výstupy) buď oba vlastnost V mají nebo ji oba nemají.
Vlastnost V je triviální, když ji mají buď všechny programy nebo ji nemá žádný program; jinak (tedy když ex. program, jenž V má, a ex. program, jenž V nemá) se V nazývá netriviální.

Vyslovíme uvedenou větu v přesnější formě (a pro Turingovy stroje). Omezíme se přitom (jak víme, bez velké újmy) na Turingovy stroje realizující (částečná) zobrazení typu $\{0,1\}^* \longrightarrow \{0,1\}^*$ (vstupem je řetězec nul a jedniček, při ukončení je neprázdný úsek pásky také řetězcem nul a jedniček). Označme množinu všech takových Turingových strojů jako ALL.

Věta 29.0.28 *(Rice) Nechť $\mathcal{A}$ je nějaká množina algoritmicky vyčíslitelných (částečných) zobrazení typu $\{0,1\}^* \longrightarrow \{0,1\}^*$. Pokud pro množinu*

$$\mathcal{M}_{\mathcal{A}} = \{M \mid M \text{ je kód Tur. stroje realizujícího zobrazení patřící do } \mathcal{A})\}$$

*platí $\mathcal{M}_{\mathcal{A}} \neq \emptyset$ a $\mathcal{M}_{\mathcal{A}} \neq$*ALL*, pak $\mathcal{M}_{\mathcal{A}}$ je nerozhodnutelná.*

Důkaz: Vezměme nějakou takovou $\mathcal{A}$, která není prázdná ani nezahrnuje všechny alg. vyčíslitelné funkce. Nechť nikde nedefinované zobrazení $\perp : \{0,1\}^* \longrightarrow \{0,1\}^*$ nepatří do $\mathcal{A}$ (opačný případ se řeší podobně). Nechť M_1 realizuje $\perp$, tedy $M_1 \notin \mathcal{M}_{\mathcal{A}}$, a nechť M_2 realizuje nějaké zobrazení z $\mathcal{A}$ (nutně takový existuje); tedy $M_2 \in \mathcal{M}_{\mathcal{A}}$.

Ukážeme, že problém DHP je převeditelný na $\mathcal{M}_{\mathcal{A}}$ (tj. na problém příslušnosti k $\mathcal{M}_{\mathcal{A}}$), čímž prokážeme nerozhodnutelnost $\mathcal{M}_{\mathcal{A}}$.

Algoritmus převodu $DHP \rightarrow \mathcal{M}_{\mathcal{A}}$ k danému stroji (kódu stroje) M (tj. ke vstupu problému DHP) sestaví stroj M', který je 'naprogramován' tak, že jeho činnost je následovná:

M' nejprve vpravo vedle svého vstupu (na kterém v této chvíli nezáleží) zapíše slovo $Kod(M)$ a na něj spustí (podprogram) M; pokud tento (pod)výpočet skončí, smaže M' případný zbytek po tomto výpočtu, najede na původně daný vstup a spustí na něj M_2.

Je zřejmé: když M se zastaví na $Kod(M)$ (tj. odpověď na daný vstup problému DHP je ANO), realizuje M' totéž zobrazení jako M_2 a tedy patří do $\mathcal{M}_{\mathcal{A}}$; když M se nezastaví na $Kod(M)$ (odpověď v DHP je NE), realizuje M' zobrazení $\perp$, tedy totéž jako M_1 a tedy do $\mathcal{M}_{\mathcal{A}}$ nepatří.

□

Kapitola 30

Další partie teorie a praxe algoritmů

Stručný obsah: Ilustrace problematiky v oblastech aproximačních, pravděpodobnostních, paralelních a distribuovaných algoritmů. Mj. též podstata šifrování s veřejným klíčem (metodou RSA). Zmínka o nových, netradičních výpočetních modelech (kvantové výpočty, DNA-výpočty).

Cíl kapitoly:
Umět vysvětlit základní principy v uvedených oblastech a ilustrovat je na příkladech.

Studijní cíle: Umět vysvětlit základní principy v uvedených oblastech a ilustrovat je na příkladech.

30.1 Aproximační algoritmy

Setkali jsme se s řadou optimalizačních úloh – v takové úloze je hledáno optimum v jisté množině přípustných řešení (např. se jedná o minimální kostru ohodnoceného grafu, maximální nezávislou množinu vrcholů grafu, nejkratší okružní jízdu obchodního cestujícího apod.). Pro některé úlohy jsou známy rychlé (tj. polynomiální) algoritmy (např. pro zmíněnou minimální kostru); pro mnohé ovšem polynomiální algoritmy (přesněji řečeno, algoritmy s polynomiální časovou složitostí), které by vždy nalezly optimum, nemáme – a podle mnoha indicií tyto algoritmy ani neexistují (je tomu tak u problému maximální nezávislé množiny i u problému nejkratší okružní cesty; připomeňte si otázku P$\stackrel{?}{=}$NP).

Co lze dělat, nemáme-li pro řešení úlohy použitelný algoritmus a přesto ji potřebujeme řešit v praxi ? Jednou z možností je použít *aproximační algoritmus*, čímž rozumíme rychlý (polynomiální) algoritmus, který alespoň aproximuje optimum – tj. vypočte nějaké přípustné řešení (např. okružní cestu), které není pokud možno moc horší než optimum (tedy např. vypočtená okružní jízda není “o moc delší” než ta nejkratší možná).

Pro sestavení aproximačního algoritmu můžeme např. použít některou obecnou metodu vedoucí k rychlým algoritmům; např. okružní jízdu je možné sestavit “hltavým” (greedy) přístupem – jeho podstatu lze vyjádřit slovy “došel-li jsi už do města A, pokračuj dále do jemu nejbližšího města, které jsi dosud nenavštívil, atd.”. Nebo lze např. použít nějakou heuristiku specifickou pro daný problém (tj. blíže nezdůvodněný postup, který dává v praxi uspokojivé výsledky nebo o kterém se alespoň domníváme, že takové výsledky může dávat).

Jak ale hodnotit kvalitu aproximačního algoritmu, tj. kvalitu jeho výstupů ? Určitou orientaci nám samozřejmě vždy může poskytnout experiment; např. srovnáním výstupů dvou různých aproximačních algoritmů pro vybrané (pro nás zajímavé) instance problému můžeme případně usoudit, který z algoritmů se pro naše účely jeví jako vhodnější. Byly a jsou ovšem vypracovávány i teoretické postupy, které umožňují vyjadřovat kvalitu aproximačních algoritmů na exaktnější bázi a navíc umožňují klasifikovat problémy podle míry jejich aproximovatelnosti. Zde si to budeme jen stručně ilustrovat na zmíněném problému obchodního cestujícího (Travelling SalesPerson); uvažujme jej v této formě:

> *Název*: TSP (*problém obchodního cestujícího*)
>
> *Vstup*: úplný neorientovaný graf s n vrcholy (“městy”) (úplný znamená, že mezi každými dvěma různými vrcholy je hrana); každé hraně (i, j) je přiřazeno celé kladné číslo $d(i, j)$ (vzdálenost mezi městy i a j)
>
> *Výstup*: permutace $\pi = \{i_1, i_2, \ldots, i_n\}$ množiny $\{1, 2, \ldots, n\}$ (tj. “okružní cesta”) tž. $D(\pi) = d(i_1, i_2) + d(i_2, i_3) + \ldots + d(i_{n-1}, i_n) + d(i_n, i_1)$ (tj. délka okružní cesty) je minimální možná.

Bude užitečné zvlášť uvažovat i podproblém problému TSP, který označíme Δ-TSP – pro jeho vstupy je vyžadováno splnění trojúhelníkové nerovnosti (tj. pro lib. vrcholy i, j, k je $d(i, j) \leq d(i, k) + d(k, j)$).

Poznámka. Připomeňme si standardní převod instance problému hamiltonovské kružnice (HK) na instanci (rozhodovacího problému příslušného k) TSP (při prokazování HK$\triangleleft$TSP); jestliže (u, v) je hrana ve výchozím grafu, položíme $d(u, v) = 1$, jinak položíme $d(u, v) = 2$. (Ve výchozím grafu G_1 s n vrcholy existuje hamiltonovská kružnice právě tehdy, když ve vytvořeném (úplném) grafu G_2 existuje okružní cesta délky n.) Jedná se vlastně o převod HK$\triangleleft$ Δ-TSP, což ukazuje, že už podproblém Δ-TSP je NP-těžký.

Cvičení. Zapište (pseudokódem) algoritmus $A1$, který k vstupu problému TSP sestrojí okružní cestu (tj. permutaci) hltavým přístupem, jak jsme se o tom zmínili výše.

Abychom se mohli vyjádřit o kvalitě algoritmu $A1$ (a dalších aproximačních algoritmů), zavedeme několik pojmů a označení. Pro vstup G problému TSP označme $m(G)$ délku nejkratší (tj. optimální) okružní cesty. Pro aproximační algoritmus A (pro problém TSP) označme $m_A(G)$ délku okružní cesty vydané algoritmem A pro vstup G. Je zřejmé, že absolutní chyba $m_A(G) - m(G)$ je nezáporná; ideální by bylo, aby byla co nejmenší. Všimněme si nejprve, že nemůžeme doufat, že by tato chyba byla omezená, a to ani v případě Δ-TSP:

Tvrzení 30.1.1 *Kdyby pro Δ-TSP existoval (polynomiální) aproximační algoritmus A s omezenou absolutní chybou (tj. existovalo by $k \in \mathbb{N}$ tak, že pro vš. instance G je $m_A(G) - m(G) \leq k$), pak P=NP.*

Důkaz: Předpokládejme, že takový algoritmus A a příslušné číslo k existují. Pak by následující polynomiální algoritmus vydával optimální řešení Δ-TSP (tj. nejkratší okružní cestu):

- v dané instanci G problému Δ-TSP vynásob každé $d(i, j)$ číslem $k + 1$; dostaneš tak instanci G' (trojúhelníková nerovnost zůstane zachována)

- na G' spusť algoritmus A; ten nutně vydá optimální řešení (permutaci vrcholů, odpovídající nejkratší cestě) (proč ?), které je ovšem i optimálním řešením pro G (proč ?)

□

Když už nemůžeme zajistit omezení absolutní chyby $m_A(G) - m(G)$, můžeme alespoň omezit (nějakou konstantou) *aproximační poměr* $m_A(G)/m(G)$? Algoritmus $A1$ nesplňuje ani to:

Cvičení. Ukažte, že pro každé $c > 1$ a každé $n > 3$ existuje instance G s n vrcholy problému TSP tak, že $m_{A1}(G)/m(G) > c$. Trošku náročnější je pak ukázat, že pro každé $c > 1$ existuje pro nekonečně mnoho n instance G s n vrcholy problému Δ-TSP tak, že $m_{A1}(G)/m(G) > c$.

Aproximační poměr se u algoritmu $A1$ dá alespoň omezit pomalu rostoucí funkcí velikosti vstupu - v případě problému Δ-TSP; konkrétně se dá ukázat (n označuje počet vrcholů grafu G)

$$\frac{m_{A1}(G)}{m(G)} \leq \frac{1}{2}\left(\lceil \log n \rceil + 1\right)$$

To ale neznamená, že problém Δ-TSP není aproximovatelný s omezeným aproximačním poměrem. Zkusme sofistikovanější (myšlenkově náročnější) algoritmus $A2$ (formulujeme ho obecně pro TSP):

ALGORITMUS $A2$

Pro daný vstup TSP, tj. ohodnocený graf s n vrcholy:

- sestroj minimální kostru grafu (podalgoritmem složitosti $O(n^2)$)

- zvol lib. vrchol a z něj realizuj prohledání sestrojené kostry (která je stromem) do hloubky (depth-first search)

- pořadí navštívení jednotlivých vrcholů při onom prohledávání udává permutaci, která je výstupem algoritmu

Pro $A2$ snadno ukážeme, že v případě problému Δ-TSP je aproximační poměr omezený konstantou, konkrétně

$$\frac{m_{A2}(G)}{m(G)} \leq 2$$

Cvičení. Dokažte předchozí vztah; speciálně dokažte $c \leq m(G)$ a $m_{A2}(G) \leq 2c$, kde c je součet ohodnocení hran v minimální kostře.

Algoritmus $A2$ je tedy lepší než $A1$; v případě problému Δ-TSP nám vždy zaručuje nalezení okružní cesty, která je nejvýše dvakrát tak dlouhá jako nejkratší cesta (optimum).

Poznámka. Christofides (1976) ukázal ještě lepší algoritmus $A3$ pro Δ-TSP, pro nějž $m_{A3}(G)/m(G) \leq 3/2$. (Sestavení algoritmu a jeho ověření je ovšem náročnější.) Aproximační algoritmus s lepším aproximačním poměrem není znám, ani není známo, zda by existence takového algoritmu implikovala P=NP.

U obecného problému TSP nelze ovšem žádný r-aproximační algorimus, tj. algoritmus s aproximačním poměrem omezeným konstantou r, očekávat:

Tvrzení 30.1.2 *Jestliže pro TSP existuje r-aproximační algoritmus (pro nějaké $r \in \mathbb{N}$), pak $P = NP$.*

Důkaz: Předpokládejme, že existuje r-aproximační algoritmus A pro TSP.

Nyní v převodu instance problému hamiltonovské kružnice (HK) na instanci TSP zmíněném výše položme $d(u, v) = 1$, je-li (u, v) hrana výchozího grafu, a jinak položme $d(u, v) = 1 + nr$, kde n je počet vrcholů výchozího grafu. Algoritmus A spuštěný na vytvořené instanci problému TSP by de facto rozhodl, zda ve výchozím grafu je hamiltonovská kružnice (proč ?).

□

K dalšímu studiu problematiky aproximačních algoritmů lze doporučit především přehledovou monografii [A+99]; některé aspekty jsou také rozebírány např. v [Hro99].

30.2 Pravděpodobnostní algoritmy

Existují rozhodovací (tedy ANO/NE) problémy, pro které neznáme rychlé (tj. polynomiální) algoritmy, a přesto je lze v praxi spolehlivě řešit nejen pro malé vstupy. Stačí slevit z absolutního nároku na řešící algoritmus, totiž z toho, aby algoritmus vždy vydal *zaručeně* (tedy stoprocentně) správnou odpověď. Přesněji řečeno, přestaneme vyžadovat, aby algoritmus nutně předepisoval *deterministický* proces, a připustíme náhodný prvek (házení kostkou). Opakované běhy algoritmu na tentýž vstup pak mohou vydávat různé výstupy – každý z nich s určitou pravděpodobností.

K formalizaci pojmu pravděpodobnostního algoritmu můžeme použít jednoduchou verzi *pravděpodobnostního Turingova stroje* (probabilistic Turing machine, PTM); takový stroj $PM = (Q, \Sigma, \Gamma, \delta, q_0, F)$ si můžeme představit jako standardní Turingův stroj s tím, že $\delta(q, a)$ může být nyní dvouprvková množina – v tom případě se při výpočtu v konfiguraci $uqav$ volí jedna ze dvou možných bezprostředně následujících konfigurací náhodně – tak, že každá má pravděpodobnost zvolení $1/2$ (můžeme si představit, že se zde hází mincí).

Podobně jako u nedeterministického Turingova stroje, odpovídá jednomu vstupu PTM strom výpočtů. Vrcholy stromu jsou ohodnoceny konfiguracemi; kořen je ohodnocen počáteční konfigurací a následníci vrcholu ohodnoceného konfigurací c jsou ohodnoceni právě bezprostředně následujícími konfiguracemi konfigurace c.

Každá hrana je přitom ohodnocena příslušnou pravděpodobností – hraně vycházející z vrcholu s jedním následníkem je přiřazena 1, každé ze dvou hran vycházejících z vrcholu s dvěma následníky je přiřazena $1/2$. Každému vrcholu v je přiřazena pravděpodobnost jeho dosažení – tj. součin ohodnocení hran na cestě od kořene k vrcholu v.

Pak lze např. přesně definovat pravděpodobnost jevu, že daný PTM vydá na daný vstup x odpověď ANO – tato pravděpodobnost je dána součtem pravděpodobností vrcholů ve stromu výpočtu pro vstup x, které jsou ohodnoceny koncovými konfiguracemi, znamenajícími odpověď ANO (tedy přijímajícími konfiguracemi).

Nepůjdeme zde do dalších formálních detailů, ale budeme si pravděpodobnostní algoritmy ilustrovat na problému prvočíselnosti:

Název: *Prvočíselnost*

Vstup: přirozené číslo k (v dekadickém zápise)
Výstup: ANO, když k je prvočíslo, NE, když k je číslo složené.

Zmiňovali jsme už dříve, že přímočaré testování prvočíselnosti podle definice vede k algoritmu s exponenciální složitostí. Žádný polynomiální) *deterministický* algoritmus nebyl až do léta 2002 znám. (Teď už je znám, ale přesto má smysl kvůli větší rychlosti nadále uvažovat pravděpodobnostní algoritmus.)

Poznámka. Všimněme si, že je zřejmé, že doplňkový problém – tj. problém složenosti čísla – je v NP (proč ?). Využitím jistých výsledků teorie čísel se dá ukázat, že i problém prvočíselnosti je v NP. Tento problém byl dlouho jedním z mála "přirozených" problémů v NP (resp. v NP∩co-NP), u kterých není znám polynomiální algoritmus a ani se neumí prokázat NP-úplnost. (Měli bychom upřesnit, že již dříve byla známa existence polynomiálního algoritmu, který by prvočíselnost rozhodoval za předpokladu, že platí tzv. Riemannova hypotéza – to se ovšem neví.)

S využitím poznatků teorie čísel lze sestavit rychlý pravděpodobnostní algoritmus A (existuje polynom p tak, že délka každé větve stromu výpočtu pro k je omezena $p(\log k)$; délka výpočtu je tedy omezena polynomiální funkcí velikosti vstupu), který má tyto vlastnosti

- jestliže n je prvočíslo, vydá A odpověď ANO s pravděpodobností 1 (NE vůbec nemůže vydat)

- jestliže n není prvočíslo, vydá A odpověď NE s pravděpodobností alespoň $1/2$.

Představme si nyní, že A zopakujeme pro dané n např. 50-krát. Když (aspoň) jednou vydá NE, víme, že n jistě není prvočíslo (a dále už A opakovat nemusíme). Když ve všech případech vydá ANO, tak n je prvočíslo s pravděpodobností větší než $1 - 2^{-50}$ – tedy "prakticky stoprocentně" (proč ?).

Naznačíme, jak algoritmus A vypadá. Mj. se opírá o tzv. *malou Fermatovu větu*:

Jestliže p je prvočíslo, tak pro každé $a, 0 < a < p,$ platí

$$a^{p-1} \equiv 1 \pmod{p}$$

Poznámka. Náčrt důkazu: Rozvineme-li $(a + b)^p$ podle binomické věty, ověříme snadno, že výsledek modulo p je roven $a^p + b^p$, je-li p prvočíslo (ověřte !). Tedy (počítáno modulo p) máme $1^p \equiv 1$, $2^p \equiv (1 + 1)^p \equiv 1^p + 1^p \equiv 1 + 1 \equiv 2$ a obecně $a^p \equiv a$, z čehož plyne $a^{p-1} \equiv 1$.

Jako důkaz složenosti (tj. neprvočíselnosti) daného čísla m nejlépe slouží nějaký netriviální dělitel a čísla m (všichni známe rychlý algoritmus, kterým se přesvědčíme, že dané a skutečně dělí m). Z Fermatovy věty ovšem plyne důležité pozorování: když nalezneme pro dané m číslo $a, 0 < a < m$ tak, že $a^{m-1} \not\equiv 1 \pmod{m}$, tak jsme našli *svědka složenosti* čísla m (tedy svědka toho, že m není prvočíslo), byť tím nezískáme netriviálního dělitele m (jen jsme tak dokázali jeho existenci).

Poznámka. Je důležité si uvědomit, že počítání mocnin modulo m umíme rovněž velmi rychle, a sice tzv. metodou "opakovaného umocňování": počítaním $x_0 = a$, $x_1 = (x_0)^2$ mod m, $x_2 = (x_1)^2$ mod m, $x_3 = (x_2)^2$ mod m, ..., $x_i = (x_{i-1})^2$ mod m, ... dostáváme postupně a, a^2 mod m, a^4 mod m, a^8 mod m, ..., a^{2^i} mod m, ... Chceme-li pak např. znát a^{560}, vynásobíme $a^{512} \cdot a^{32} \cdot a^{16}$ (tj.$x_9 \cdot x_4 \cdot x_5$; vše počítáno samozřejmě modulo m).

Představme si nyní *hypoteticky*, že by platilo:

Hypotéza:
jestliže m není prvočíslo, tak alespoň jedna polovina z čísel $1, 2, \ldots, m-1$ jsou svědkové složenosti čísla m

Pak by kýžený pravděpodobnostní algoritmus A pro testování prvočíselnosti (který se pro zadané složené číslo mohl splést s pravděpodobností nejvýš $1/2$) byl nabíledni. (Jak by vypadal ?)

Jak ukazují další výsledky teorie čísel, situace není takto jednoduchá – uvedená hypotéza neplatí pro všechna složená čísla; definice svědka složenosti se ale dá upravit tak, aby hypotéza (pro nově definovaného svědka) platila !

Poznamenejme, že ač problém testování prvočíselnosti je takto uspokojivě vyřešen, nejsou známy žádné rychlé algoritmy pro nalezení prvočíselného rozkladu složeného čísla. Tedy ač např. umíme rychle zjistit, že dané 200-místné číslo je složené, nezaručuje nám to rychlé nalezení netriviálního dělitele. Speciálně, když nám někdo dá 200-místné číslo, které vytvořil jako součin dvou 100-místných prvočísel, nemáme (zatím ?) praktickou šanci tato prvočísla najít. (To, že tato "praktická šance" skutečně neexistuje, neumíme dokázat. V této souvislosti si všimněte zmínky o Shorově algoritmu v kapitolce o kvantových výpočtech !). V následující podkapitolce si ukážeme, k čemu se to používá.

Šifrovací systém s veřejným klíčem; RSA-kryptosystém

Stručně nastíníme podstatu jednoho používaného způsobu elektronické komunikace, při kterém se zpráva šifruje veřejně známým klíčem adresáta a který rovněž umožňuje tzv. 'elektronické podpisy'. Přitom bude zřejmá souvislost bezpečnosti popsaného způsobu s otázkami teorie složitosti.

Mějme doménu D (přípustných) zpráv; např. si představme množinu všech (zápisů) 200-místných dekadických čísel. (Čtenář si snadno představí kódování textu pomocí řetězce dekadických číslic; delší zprávy pak sestávají z více bloků.)

Dvěma permutacím (vzájemně jednoznačným zobrazením) $enc : D \to D$ a $dec : D \to D$ řekneme oboustranně komutativní šifrovací pár, dále jen *šifrovací pár*, jestliže

$$\forall m \in D : dec(enc(m)) = enc(dec(m)) = m$$

Přitom enc a dec jsou efektivně vyčíslitelné (to zde znamená, že pro ně existují rychlé algoritmy) a nemáme způsob, jak z algoritmu pro enc odvodit algoritmus pro dec.

Princip šifrovacího systému s veřejným klíčem, v němž spolu uživatelé komunikují, spočívá v tom, že každý uživatel X si vyrobí svůj šifrovací pár (enc_X, dec_X), zveřejní algoritmus pro enc_X a drží v tajnosti algoritmus pro dec_X.

Když chce uživatel A zaslat zprávu m uživateli B, vyhledá ve veřejném seznamu (algoritmus pro) enc_B a zašle mu $enc_B(m)$. B po obdržení aplikuje dec_B a získá $dec_B(enc_B(m)) = m$.

Oboustranná komutativnost u šifrovacího páru umožňuje *elektronické podpisy*: A zašle dvojici $enc_B(m)$, $enc_B(dec_A(m))$; B po přečtení $dec_B(enc_B(m)) = m$ zjistí, že zpráva má být od A (A uvede v m své jméno), druhá část $dec_B(enc_B(dec_A(m))) = dec_A(m)$ je pro něj nesrozumitelná – ovšem po vyhledání a aplikaci enc_A musí dostat m, čímž ověří pravost (když se např. jedná o podepsaný šek, může ho B předložit bance k proplacení; všimněte si, že B není schopen podpis A zfalšovat).

V RSA-systému si uživatel X vyrobí šifrovací pár podle následujícího algoritmu:

1. Zvol dvě náhodná 100-místná prvočísla p, q.
2. Spočítej součiny $n = pq$ a $\Phi(n) = (p-1)(q-1)$.
3. Urči (malé) e tž. $GCD(e, \Phi(n)) = 1$ (GCD označuje největší společný dělitel).
4. Vypočti d tž. $de \equiv 1(\text{ mod } \Phi(n))$.
5. Zveřejni dvojici (e, n); šifrovací funkce je $enc_X(m) = m^e \text{ mod } n$.
6. Drž v tajnosti (d, n); dešifrovací funkce je $dec_X(m) = m^d \text{ mod } n$.

Bod 1 lze provést takto: Náhodně zvolíme 100-místné liché číslo, a otestujeme jeho prvočíselnost pravděpodobnostním algoritmem zmíněným výše. Když prvočíslem není, přičteme 2, znovu otestujeme, v negativním případě znovu přičteme 2 atd., dokud test prvočíselnosti neskončí pozitivně. Dá se ukázat, že prvočíslo bude "takřka jistě" nalezeno mezi prvními 200 testovanými čísly. (To lze odvodit z faktu, že funkce $\pi(n)$ udávající počet prvočísel mezi prvními n přirozenými čísly, je velmi dobře aproximována funkcí $n/\ln n$.)

Body 3. a 4. se dají řešit takto: postupně probíráme (kandidáty na číslo) e a Eukleidovým algoritmem ověřujeme zda, $GCD(e, \Phi(n)) = 1$; v kladném případě dostaneme (při určité variantě Eukleidova algoritmu) přímo lineární kombinaci $a \cdot e + b \cdot \Phi(n) = 1$ a tedy lze vzít $d = a$.

Z elementárních faktů teorie čísel se dá ukázat korektnost, tj. $\forall m : m^{ed} \equiv m \text{ (mod } n)$.

Připomeneme-li si metodu "opakovaného umocňování", je zřejmé, že (de)šifrovat se dá velmi rychle.

Bezpečnost systému spočívá v tom, že se neví, jak d zjistit jinak, než pro n nalézt rozklad $n = p \cdot q$; jak jsme už zmínili, nejsou ale známy žádné algoritmy, které by byly v rozumné době schopné nalézt prvočíselný rozklad 200-místných čísel.

Další informace o problematice pravděpodobnostních algoritmů, šifrování apod. najde čtenář např. v [CLR90], [Sip97].

30.3 Paralelní algoritmy

V této části se jen letmo dotkneme oblasti paralelních algoritmů a paralelní výpočtové složitosti.

S rozvojem paralelních architektur, speciálně počítačů s více procesory, se přirozeně vynořila otázka, zda a které problémy lze řešit rychleji, použijeme-li k jejich řešení paralelní přístup, tedy algoritmus, který (na rozdíl od standardního sekvenčního algoritmu) počítá s více vykonavateli (procesory). Např. k problému

> *Název*: *Vzorek v textu*
>
> *Vstup*: vzorek p ('krátká' posloupnost symbolů) a text t ('dlouhá' posloupnost symbolů)
>
> *Výstup*: místa všech výskytů p v t

můžeme nějaký standardní algoritmus A nahradit paralelním

> - rozděl vstup t na dvě (přibližně stejně dlouhé) části t_1, t_2
> - spusť paralelně A na p, t_1 a A na p, t_2
> - zpracuj ('slej') výsledky obou paralelně běžících '(pod)procesů' (přitom ošetři rozhraní t_1, t_2) a vydej na výstup

Máme-li skutečně možnost paralelní běh implementovat (máme dva vykonavatele), doba výpočtu se v zásadě dvakrát zmenší oproti algoritmu A – za předpokladu, že přidaná činnost spojená s rozdělením práce a zkombinováním výsledků práce jednotlivých (pod)procesů je v celkovém kontextu zanedbatelná.

Čtenář snadno navrhne úpravu algoritmu pro případ tří, čtyř, či obecně k procesorů. Na druhé straně asi tuší, že chceme-li nějak postihnout a studovat paralelní výpočty obecně, nestačí se omezit na model s fixním počtem procesorů. Proto budeme počítat s (potenciálně) neomezeným paralelismem, konkrétně s modely umožňujícími nasadit na vstup velikosti n až $f(n)$ procesorů, kde f je nějaká (např. i exponenciální) funkce.

V případě hledání výskytů vzorku p v textu t délky m si můžeme představit nasazení m procesorů $1, 2, \ldots, m$, přičemž procesor i zjišťuje, zda se vzorek p nachází na pozici i (v kladném případě vydá i na výstup). Pomineme-li etapu rozdělení práce mezi procesory a řešení případných konfliktů na výstupu, je délka 'jádra' výpočtu celého systému uměrná délce ℓ vzorku p.

Můžeme ovšem také paralelizovat činnost ověření, zda p se vyskytuje na pozici i: na tento úkol nasadíme tým – ℓ-tici procesorů $(i, 1), (i, 2), \ldots, (i, \ell)$, kde procesor (i, j) zjišťuje, zda $p(j) = t(i + j - 1)$ (zde $p(j)$ označuje j-tý znak v p); v záporném případě např. zapíše 1 do jisté paměťové buňky b_i vyhrazené i-tému týmu (její hodnota byla na začátku 0). Po ukončení práce všech členů týmu 'manažer týmu' (např. $(i, 1)$) vydá na výstup i v případě, že $b_i = 0$. Takto je délka trvání 'jádra' výpočtu nezávislá na velikosti vstupu a lze ji omezit nějakou konstantou.

Abychom mohli úvahy o paralelních algoritmech a jejich výpočtové složitosti zpřesnit, potřebujeme nějaký konkrétní ‘paralelní počítač’, resp. jeho abstraktní model. Podobně jako nám v případě sekvenčních algoritmů posloužil model RAM, může nám zde posloužit PRAM (parallel RAM), definovaný níže. Na rozdíl od ‘sekvenčního případu’, kde RAM je skutečně obecně přijímaným referenčním modelem, v ‘paralelním případě’ je otázka ‘toho pravého’ modelu daleko komplikovanější. Nejdříve se ale soustředíme na PRAM a pak se ke zmíněné otázce vrátíme.

Stroj PRAM sestává z potenciálně nekonečného pole strojů (procesorů) RAM očíslovaných $0, 1, 2, \ldots$ a ze (sdílené) globální paměti; přitom

- každému procesoru je jeho jedinečné identifikační číslo přístupné jako hodnota speciálního registru PID (processor identifier)
- každý procesor může kromě své lokální paměti přistupovat také ke společné globální paměti (která je organizována stejně jako paměti lokální)

Přitom (je možné si představit centrální řídicí počítač, který zajišťuje, že)

- všechny procesory se řídí týmž algoritmem
- procesory pracují paralelně a synchronizovaně, t.j. v každém kroku PRAMu se provede jedna instrukce na každém RAMu

Důležitou technickou otázkou jsou případné konflikty při paralelním čtení ze / zapisování do stejné buňky (globální) paměti. Zde předpokládáme

- ze stejné buňky sdílené paměti může v rámci každé instrukce libovolný počet procesorů číst (CR = Concurrent Read)
- do stejné buňky sdílené paměti může v rámci každé instrukce libovolný počet procesorů zapisovat za předpokladu, že všechny zúčastněné procesory zapisují tutéž hodnotu (CW-C Concurrent Write Common).

Poznamenejme jen, že používaných variant PRAMu je více; např. typ CREW umožňuje souběžné čtení, ale zakazuje souběžné zapisování (Exclusive Write).

Jinak ještě dodejme, že vstupní data jsou ukládána v počátečním úseku globální paměti (kam se také uloží výstup).

Čtenář si teď může rozmyslet naprogramování výše uvedeného algoritmu pro hledání vzorku v textu pro PRAM. Důležitým bodem k promyšlení je využití PID procesorů k rozdělení práce (každý procesor na základě svého PID zjistí, v jakém týmu a na jakém úkolu má pracovat). My to zde provádět nebudeme, ale ilustrujeme na příkladu násobení (booleovských) matic.

Uvažujme nejdříve standardní násobení dvou matic $n \times n$; provádíme přitom $\Theta(n^3)$ číselných operací (což odpovídá složitosti standardního sekvenčního algoritmu). Můžeme-li nasadit n^2 procesorů, z nichž každý (nezávisle) spočítá jeden prvek výsledné matice, potřebný čas se sníží na $O(n)$. Nebudeme teď zkoumat, jak se dá postup vylepšit nasazením týmu na součet n čísel, ale zjednodušíme si situaci uvažováním násobení *booleovských* matic: máme spočítat $C := A \times B$, kde prvky jsou 0 a 1, přičemž $1 + 1 = 1$. Nasaďme n^2 týmů o n členech, tedy celkově n^3 procesorů – označme je (i, j, p) pro $1 \leq i, j, p \leq n$. Pokud každý procesor (i, j, p) vykoná (paralelně s ostatními) příkaz

if $A(i,p)$ and $B(p,j)$ **then** $C(i,j) := true$

je úkol hotov !

Pseudokód celého programu, jímž se každý procesor řídí, je zachycen níže (čtenář samozřejmě vidí, jak by šel tento pseudokód přímočaře přepsat jako posloupnost skutečných instrukcí RAMu). Všimněte si, že pro konkrétní i, j provede příkaz $C(i,j) := true$ nula až n procesorů – jelikož všechny procesory ovšem přiřazují $C(i,j)$ tutéž hodnotu, jedná se o povolené souběžné zapisování (CW-C). Také je využito předpokladu, že matice C je na začátku vynulována (všechny prvky jsou $false$).

VSTUP: Ve sdílené paměti je uloženo číslo n a dvourozměrná pole A, B, C : array $[1 \ldots n, 1 \ldots n]$ of $boolean$; všechny prvky C jsou přitom 0 (tj. $false$).

VÝSTUP: Pole C obsahuje booleovský součin matic A, B

POSTUP:

begin { každý procesor začíná výpočtem (i,j,p) ze svého PID }

$i := 1 +$ PID $div\ n^2$;
$j := 1 +$ (PID $mod\ n^2$) $div\ n$;
$p := 1 +$ PID $mod\ n$;

if $A(i,p)$ and $B(p,j)$ **then** $C(i,j) := true$
{ píše 0 až n procesorů, ale vždy stejnou hodnotu }

end

Takto máme tedy (paralelní) algoritmus, který vynásobí booleovské matice A, B typu $n \times n$ v konstantním čase při použití n^3 procesorů.

Zmínili jsme již, že v paralelním případě je otázka všeobecně přijímaného referenčního modelu daleko složitější než v případě sekvenčním. Nebudeme teď diskutovat nakolik je PRAM blízký či vzdálený reálným paralelním architekturám, jen zmíníme, že patří do tzv. druhé třídy výpočetních modelů (viz např. [vEB90]). *První počítačovou třídu* zde tvoří všechny tzv. rozumné sekvenční modely, určené tezí

Rozumné sekvenční modely dokážou simulovat a být simulovány Turingovými stroji s nejvýš polynomiální časovou ztrátou a s nejvýš lineární prostorovou ztrátou.

Pozn.: Nebudeme se zde zabývat nuancí, zda se vyžaduje, aby obě podmínky platily u příslušné simulace zároveň.

Druhou počítačovou třídu pak tvoří tzv. rozumné paralelní modely, určené tzv. *paralelní tezí* (stručně: sekvenční prostor = paralelní čas):

Problémy řešitelné na rozumných *sekvenčních* modelech v polynomiálně omezeném *prostoru* se dají řešit na *rozumných paralelních modelech* v polynomiálně omezeném *čase*.

Pozn.: Někdy se teze formuluje obecněji: $\mathcal{M}$ je rozumný paralelní model, jestliže pro libovolnou funkci F je

$$\bigcup_k \mathcal{M}\text{-TIME}(F^k(n)) = \bigcup_k \text{SPACE}(F^k(n))$$

(Připomeňme, že $F^k(n)$ označuje $(F(n))^k$.)

Poznamenejme, že má rozumný smysl uvažovat i jiné třídy počítačů, např. jisté slabší ale 'fyzikálně realizovatelnější' paralelní modely (viz např. [Wie92]).

Praxi nejbližší je zřejmě navrhování efektivních paralelních algoritmů pro konkrétní problémy. U pojmu efektivního (a prakticky realizovatelného) paralelního algoritmu došlo k určité shodě: *efektivním paralelním algoritmem* se rozumí paralelní algoritmus (je možno dosadit PRAM), který pracuje v polylogaritmickém prostoru (tj. s celkovou prostorovou složitostí $(\log n)^k$ pro něj. k) a s nasazením polynomiálního počtu procesorů. (Nasazení většího než polynomiálního počtu procesorů lze těžko považovat za zvládnutelné.) Třída problémů, které jsou řešitelné takovýmito efektivními paralelními algoritmy, se označuje NC (Nick's Class; název je podle Nicka Pippengera); poznamenejme, že definice třídy NC je robustní vzhledem k mnoha variantám modelů paralelních algoritmů.

Je snadné vyvodit NC $\subseteq$ PTIME; opět se ale neumí dokázat, zda inkluze je vlastní (má se zato, že ano). Zhruba řečeno, NC obsahuje ty (sekvenčně zvládnutelné) problémy, které lze efektivně paralelizovat. Např. kniha [GR88] ukazuje řadu takových příkladů; problémy na grafech, třídění, analýza bezkontextových jazyků apod.

Pozn. Všimněme si, že nutnou podmínkou k existenci efektivního paralelního algoritmu pro daný problém je možnost jeho řešení v polylogaritmickém prostoru na sekvenčním stroji (viz paralelní tezi pro $F = \log$). Např. pro rozpoznávání bezkontextových jazyků bylo známo, že prostorová složitost je v $O(\log^2 n)$; netriviální efektivní paralelní algoritmus pro tento problém rozebírá také např. [CM90].

Jak jsme řekli, neumí se dokázat, že existují problémy v PTIME $-$ NC, tzv. *vnitřně sekvenční* (inherently sequential) problémy. Nejpravděpodobnějšími kandidáty na takové problémy jsou tzv. PTIME-úplné (stručně P-úplné) problémy; *problém je P-úplný* jestliže je v P a každý jiný problém v P je na něj převeditelný (sekvenčním) algoritmem pracujícím v logaritmickém prostoru (konkrétněji: Turingovým strojem s jednou vstupní páskou, jež je 'read-only' a je na ní napsán vstup velikosti n, jednou 'read-write' pracovní páskou, na níž je možno navštívit $\log n$ políček, a jednou 'write-only' výstupní páskou).

Poznámka. Není těžké ukázat, že takováto LOGSPACE redukce je realizovatelná v polylogaritmickém čase s použitím polynomiálního počtu procesorů; je to tedy instance tzv. NC-redukce. Někdy se P-úplnost definuje v širším smyslu – pomocí NC-redukcí.

Příkladem P-úplného (a tedy pravděpodobně 'vnitřně sekvenčního') problému je problém vyhodnocení booleovského obvodu, a to i v monotónní verzi (bez negace), kterou uvedeme:

> *Název*: (mono) CVP (monotone Circuit Value Problem)
>
> *Vstup*: konečný acyklický orientovaný graf (obvod), který má jediný výstupní vrchol (nevychází z něj hrana), všechny jeho vstupní vrcholy (nevchází hrana) jsou ohodnoceny 0 nebo 1 a všechny nevstupní vrcholy jsou označeny $\wedge$ (and) či $\vee$ (or).
>
> *Výstup*: hodnota na výstupním vrcholu (hradle)

(Z kontextu je snad zřejmé, že hodnota hradla $\wedge$ je 1 právě když hodnota všech jeho předchůdců je 1 a hodnota hradla $\vee$ je 1 právě když hodnota aspoň jednoho jeho předchůdce je 1.)

Poznamenejme, že podobně jako u NP-úplných problémů a v jiných analogických situacích je nejtěžším prokázat P-úplnost nějakého (základního) problému. P-úplnost (resp. P-obtížnost) dalších problémů lze pak ukazovat LOGSPACE redukcemi (resp. NC-redukcemi) z už známých P-úplných problémů. (Dá se totiž ukázat, že složení dvou LOGSPACE-redukcí je také realizovatelné jako LOGSPACE-redukce, byť to není tak triviální jako v případě PTIME-redukcí; zkuste si rozmyslet proč.)

30.4 Distribuované algoritmy

Distribuované algoritmy také představují jistý druh paralelismu. Typické ovšem je, že běží zároveň na třeba i hodně vzdálených samostatných počítačích, které nejsou svázány s nějakým centrálním počítačem, a pracují *asynchronně* (nikoli 'v taktu'). Vzájemně komunikují zasíláním zpráv po síti, kterou jsou vzájemně propojeny.

Stručně jen načrtneme jeden problém z dané oblasti a řešení necháme na čtenáři. Podrobné informace o distribuovaných algoritmech lze nalézt např v [Lyn96].

Volba koordinátora

Zadání je převzato z [CP91]. Představme si procesory spojené v *kruhové síti*. Předpokládáme:

1. Každý procesor je připojen na dva obousměrné komunikační kanály, jejichž prostřednictvím si vyměňuje zprávy se svými dvěma sousedy.
2. Komunikace je asynchronní; každý procesor může v libovolném okamžiku vyslat zprávu jednomu ze svých sousedů. Zprávy jsou doručovány bezpečně (neztrácejí se ani nekomolí) a v pořadí, ve kterém byly odeslány (předpokládá se, že kanály jsou vybaveny na příjmu i vysílání vyrovnávací pamětí, organizovanou jako fronta).
3. Všechny procesory se řídí týmž algoritmem. Každý procesor je opatřen svým (zaručeně jedinečným) identifikačním číslem PID (jehož hodnota je mu dostupná). Žádný z procesorů neví, jak je kruh dlouhý ani jaká mají čísla jiní účastníci.
4. Všechny procesory jsou trvale připraveny přijímat a zpracovávat zprávy.

Zadání úlohy:

> Prostřednictvím výměny zpráv zjistit maximální identifikační číslo na kruhu. Proces zjišťování maxima, zvaný též 'volba koordinátora', může zároveň spustit libovolný počet procesorů. Proces skončí v okamžiku, kdy všechny procesory znají maximum a je doručena poslední zpráva, která byla s tímto procesem spojená.

Mírou velikosti zadání je počet n procesorů na kruhu. Přirozenou mírou složitosti algoritmu je celkový počet $M(n)$ vyslaných zpráv (komunikační složitost). Zkuste navrhnout (distribuovaný) algoritmus s pokud možno co nejmenším $M(n)$.

30.5 Nové výpočetní modely

Zde v podstatě jen poznamenáme, že provádění výpočtů (computation) není nutně omezeno jen na elektronické počítače, jak je známe.

Kvantové výpočty (Quantum computing)

Zhruba v 80. létech 20. století se začala diskutovat možnost realizace výpočtů (počítačů) na bázi kvantové mechaniky. Z těchto (fyzikálních) úvah vznikl jistý teoretický model počítače, pro který byly v 90. létech navrženy zajímavé algoritmy. Patrně nejznámější je existence *Shorova algoritmu pro prvočíselný rozklad čísel*, který pracuje (na kvantovém počítači) v polynomiálním čase – v případě praktické realizace by tak umožnil rozbití šifrovacích algoritmů užívaných pro bezpečnou výměnu zpráv, elektronické podpisy apod. (viz kapitolku o pravděpodobnostních algoritmech a šifrování).

Kvantový počítač, resp. jeho model, je v něčem podobný pravděpodobnostní verzi Turingova stroje (jednotlivé možnosti nemají ovšem přiřazeny pravděpodobnosti (reálná čísla), ale komplexní ‘amplitudy’). Zdroj možných efektivních algoritmů je v tom, že kvantový stroj dokáže de facto najednou (a deterministicky) ověřovat exponenciálně mnoho možností, z nichž ovšem dostaneme k dispozici jen jednu v okamžiku ‘pozorování’ výpočtu; pointa je v tom, jak navrhnout algoritmus (a dobu pozorování), aby se pravděpodobnost pozorování ‘dobrých’ řešení blížila k jedničce, zatímco pravděpodobnost pozorování ‘špatných’ řešení šla k nule.

Zde téma nebudeme dále rozebírat a čtenáře odkazujeme např. na [Gru99].

DNA-výpočty (DNA-computing)

Experimentální výsledky (např. při řešení instancí některých NP-úplných problémů) byly dosaženy při výpočtech na ‘biologické bázi’; čtenáře můžeme odkázat např. na [PRS98].

Literatura

[A+99] Giorgio Ausiello et al. *Complexity and Approximation*. Springer Verlag, 1999.

[Chy84] Michal Chytil. *Automaty a gramatiky*. SNTL Praha, 1984.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CM90] Michal Chytil and František Mráz. Složitost paralelních výpočtů. In *Sborník SOFSEM'90*, pages 157–186, 1990.

[CP91] Michal Chytil and Jan Pavelka. Algoritmy. MFF UK, Praha, 1991.

[GR88] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[Gru97] Jozef Gruska. *Foundations of Computing*. Intern. Thomson Computer Press, 1997.

[Gru99] Jozef Gruska. *Quantum Computing*. McGraw-Hill, 1999.

[Har93] D. Harel. *Algorithmics (The Spirit of Computing)*. Addison Wesley, 1993.

[Hro99] Juraj Hromkovič. Stability of approximation algorithms for hard optimization problems. In *Proc. SOFSEM'99*, volume 1725 of *Lecture Notes in Computer Science*, pages 29–47. Springer Verlag, 1999.

[HU78] J. Hopcroft and J. Ullman. *Formálne jazyky a automaty*. Alfa Bratislava, 1978.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[Kuč83] Luděk Kučera. *Kombinatorické algoritmy*. SNTL Praha, 1983.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[Man81] Zohar Manna. *Matematická teorie programů*. SNTL Praha, 1981.

[Med00] Alexander Meduna. *Automata and Languages*. Springer, 2000.

[Mor84] J. Morávek. *Složitost výpočtů a optimální algoritmy*. Academia Praha, 1984.

[MvM87] Molnár, Češka, and Melichar. *Gramatiky a jazyky*. Alfa-SNTL, 1987.

[PRS98] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *DNA Computing: New Computing Paradigms*. Springer Verlag, 1998.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publish. Comp., 1997.

[vEB90] Peter van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A : Algorithms and Complexity, chapter 1. Elsevier, 1990.

[Wie92] Juraj Wiedermann. Weak parallel machines: a new class of physically feasible parallel machine models. In *Proc. MFCS'92*, volume 629 of *Lecture Notes in Computer Science*, pages 95–111. Springer Verlag, 1992.