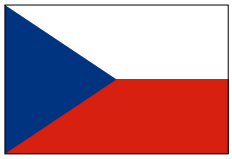


Vysoká škola báňská – Technická univerzita Ostrava

Úvod do teoretické informatiky

učební text

Petr Jančar
Martin Kot
Zdeněk Sawa

Ostrava 2007

Recenze: Doc. RNDr. Jaroslav Markl

Název: Úvod do teoretické informatiky – učební text
Autoři: Petr Jančar, Martin Kot, Zdeněk Sawa
Vydání: první, 2007
Počet stran: 261
Náklad: xx
Vydavatel a tisk: Ediční středisko VŠB-TUO

Studijní materiály pro studijní obor Informatika a výpočetní technika
fakulty elektrotechniky a informatiky
Jazyková korektura: nebyla provedena

Určeno pro projekt:
Operační program Rozvoj lidských zdrojů
Název: E-learningové prvky pro podporu výuky odborných a technických
předmětů
Číslo: CZ.04.01.3/3.2.15.2/0326
Realizace: VŠB – Technická univerzita Ostrava
Projekt je spolufinancován z prostředků ESF a státního rozpočtu ČR

ISBN xxxx

Obsah

Úvod

Poznámka autorů určená recenzentům. Tento učební text byl dopsán a zkompletován před začátkem běhu kursu v letním semestru 2006/2007. Jsme si vědomi, že v něm jsou ještě mnohé nedostatky (byť doufáme, že ne zásadního rázu). V průběhu semestru hodláme celý text postupně revidovat (souběžně s výukou příslušných partií); revidované části budou rovněž zpřístupněny studujícím, bude-li to třeba. Samozřejmě hodláme při revizi vzít v potaz i kritické připomínky recenzentů.

Předkládaný materiál slouží jako studijní text pro předměty teoretické informatiky, speciálně pro oblasti teorie jazyků a automatů a teorie algoritmů (tj. teorie vyčíslitelnosti a složitosti). Text existuje ve dvou verzích. Základní verze je určena pro kurs „Úvod do teoretické informatiky“, rozšířená verze pak pro kurs „Teoretická informatika“. Rozšířená verze obsahuje veškerý materiál verze základní a navíc má části označené jako pokročilé; tyto části se vyskytují v rámci jednotlivých kapitol či jako celé kapitoly.

Souhrnný název studijního textu by také mohl být *Základy teorie výpočtů* (Theory of Computation). Tato teorie 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. 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. 3.1, 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 Javě je správně „javovsky“.

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.

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 univerzálními modely počítačů (algoritmů) – konkrétně s *Turingovými stroji* a *stroji RAM*. Na těchto modelech postavíme vysvětlení pojmů *rozhodnutelnosti* a *nerozhodnutelnosti* problémů a podrobněji se budeme zabývat *výpočetní složitosti algoritmů a problémů*; speciálně pak třídami složitosti PTIME a NPTIME.

Rozšířenou verzi zakončíme úvodem do problematiky *aproximačních, pravděpodobnostních, paralelních a distribuovaných algoritmů*.

Poznamenejme, že účelem kursu není popis konkrétních větších reálných aplikací studovaných (teoretických) pojmů; cílem 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]).

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), [Kuč83], [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].

Pokyny ke studiu

Jak jsme již zmínili, tento text existuje ve dvou verzích – základní a rozšířené. Základní verze textu je primárně určena pro předmět „Úvod do teoretické informatiky“, zatímco rozšířená pro předmět „Teoretická informatika“ a studenty Úvodu do teoretické informatiky s hlubším zájmem o danou problematiku. Rozšířená verze se od základní verze odlišuje v následujících ohledech:

- Obsahuje oproti základní verzi několik dalších kapitol. U názvů těchto kapitol je uvedeno, že patří do pokročilé části.
- Některé kapitoly, které se nacházejí i v základní části jsou rozšířeny o pokročilou část. Začátek této části je označen nadpisem

Pokročilé partie

- Do textu, který je i v základní verzi, jsou na některých místech přidány rozšiřující poznámky. Tyto poznámky jsou označeny textem „*Pro pokročilé:*“.

Text je členěn do kapitol podle jednotlivých témat. Kapitola 1 shrnuje základní definice, kterým je třeba rozumět pro studium následujícího textu a které by čtenář již měl znát z dřívějšího studia. Tato kapitola slouží také k upřesnění a shrnutí matematické notace používané v textu.

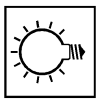
Každá kapitola začíná uvedením cílů dané kapitoly, které stručně zhrnují, jaké znalosti čtenář získá po prostudování dané kapitoly. Cíle jsou vždy uvedeny následující ikonkou a nadpisem:

Cíle kapitoly:

- Zde budou uvedeny cíle dané kapitoly.

V případě, že rozšiřující část kapitoly obsahuje témata, která nejsou obsažena v základní části, jsou cíle uvedeny rovněž na začátku rozšiřující části.

Součástí textu jsou řešené příklady, které podrobně ukazují, jak řešit vybrané typy příkladů. Tyto příklady jsou vždy uvedeny následující ikonkou a textem:



Řešený příklad: Zde je uvedeno zadání příkladu.

Řešení: Zde pak následuje ukázkové řešení.

Další součástí textu jsou otázky a cvičení, které by měly čtenáři sloužit k tomu, aby ověřil nabyté znalosti. Rozdíl mezi otázkami a cvičeními je ten, že na otázky by měl být čtenář být schopen odpovědět hned či po krátkém zamyšlení bez nutnosti něco řešit. Naproti tomu vyřešení cvičení bude většinou vyžadovat použití tužky a papíru.

Otázky, které jsou roztroušeny v textu a vztahují se přímo k právě diskutované problematice, označujeme jako „kontrolní otázky“ a jsou označeny takto:

Kontrolní otázka: Zde bude uveden text otázky.

Některé otázky jsou shrnuty do bloku otázek na konci příslušné kapitoly či sekce. Tento blok je označen stejnou ikonkou jako kontrolní otázka:

Otázky:

Otázka: Text první otázky.

Otázka: Text další otázky.

Cvičení jsou označena následujícím způsobem (pokud následuje více cvičení za sebou, je ikonka uvedena jen u prvního z nich):

Cvičení: Zde bude uveden text zadání.

Některé kapitoly obsahují samostatnou sekci nazvanou „Cvičení“. Tato sekce obsahuje další příklady k dokonalejšímu procvičení probírané látky.

Otázky a cvičení jsou číslovány. Na konci textu jsou pak v Příloze B uvedena řešení většiny z nich.

Kapitola 1

Základní definice

Cíle kapitoly:

- Připomenutí základních pojmů (množiny, relace, funkce, grafy, základy výrokové logiky, ...), které by měly být čtenáři známy už z předchozího studia.

1.1 Množiny

Množina je kolekce vzájemně odlišitelných objektů, kterým říkáme její *prvky*. Jestliže je objekt x prvkem množiny S, píšeme $x \in S$. Jestliže x není prvkem S, píšeme $x \notin S$.

Často používanou konvencí je, že množiny jsou označovány velkými písmeny $(A, B, X, Y, \ldots)$ a jejich prvky malými písmeny $(a, b, x, y, \ldots)$.

Jednou z možností, jak popsat množinu je explicitně vyjmenovat všechny její členy mezi složenými závorkami. Pokud například chceme definovat, že množina S obsahuje čísla 1, 2 a 3 (a neobsahuje žádné další prvky), můžeme napsat $S = \{1, 2, 3\}$.

Množina nemůže prvek obsahovat více než jednou a prvky množiny nejsou nijak seřazeny. Množiny A a B jsou *si rovny*, jestliže obsahují tytéž prvky. Pro označení toho, že si jsou množiny A a B rovny, používáme zápis $A = B$. Platí tedy například $\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$.

Poznámka: Kromě množin se také někdy používají *multimnožiny*. Na rozdíl od množiny může multimnožina obsahovat více výskytů jednoho prvku.

Množina, která neobsahuje žádné prvky, se nazývá *prázdná množina* a označuje se symbolem $\emptyset$.

Pro označení některých často používaných množin budeme v textu používat následující symboly:

- $\mathbb{N}$ pro označení množiny všech *přirozených* čísel, tj. $\mathbb{N} = \{0, 1, 2, \ldots\}$,
- $\mathbb{N}_+$ pro označení množiny všech *kladných přirozených* čísel, tj. $\mathbb{N}_+ = \{1, 2, 3, \ldots\}$.
- $\mathbb{Z}$ pro označení množiny všech *celých* čísel, tj. $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
- $\mathbb{Q}$ pro označení množiny všech *racionálních* čísel, tj. množiny všech čísel, která jsou vyjádřitelná jako zlomek, kde v čitateli i jmenovateli je celé číslo.
- $\mathbb{R}$ pro označení množiny všech *reálných* čísel.

Jestliže všechny prvky množiny A patří rovněž do množiny B (tj. pokud z $x \in A$ plyne $x \in B$), pak říkáme, že A je *podmnožinou* B, což zapisujeme výrazem $A \subseteq B$. Množina A je *vlastní podmnožinou* množiny B, jestliže $A \subseteq B$, ale $A \neq B$, tj. jestliže existuje prvek x takový, že $x \in B$, ale $x \notin A$. To, že A je vlastní podmnožinou B, zapisujeme výrazem $A \subset B$.

Poznámka: Někteří autoři používají zápis $A \subset B$ pro označení toho, že A je podmnožinou B (tj. připouští i možnost $A = B$), a pro označení toho, že A je vlastní podmnožinou B, pak používají zápis $A \subsetneq B$.

Pro libovolnou množinu A platí $A \subseteq A$. Pro libovolné množiny A a B platí, že $A = B$ právě když $A \subseteq B$ a $B \subseteq A$. Pro libovolné množiny A, B a C platí, že jestliže $A \subseteq B$ a $B \subseteq C$, pak $A \subseteq C$. Pro libovolnou množinu A platí $\emptyset \subseteq A$.

Pro danou množinu A můžeme definovat množinu $B \subseteq A$ tvořenou těmi prvky množiny A, které mají určitou vlastnost (splňují nějakou podmínku). Například můžeme definovat podmnožinu X množiny přirozených čísel $\mathbb{N}$ tvořenou těmi čísly, která dávají po dělení pěti zbytek 2 (tj. takovými čísly

$x \in \mathbb{N}$, která splňují podmínku $x \bmod 5 = 2$). Pro definici takové množiny používáme následující zápis:

$$X = \{x \in \mathbb{N} \mid x \bmod 5 = 2\}$$

Pokud je z kontextu zřejmé, z jaké množiny A prvky vybíráme, je možné tuto informaci vynechat a psát například $X = \{x \mid x \bmod 5 = 2\}$.

Poznámka: Někteří autoři používají místo symbolu '|' symbol ':', takže píší např. $X = \{x \in \mathbb{N} : x \bmod 5 = 2\}$.

Z již definovaných množin můžeme vytvářet nové množiny pomocí *množinových operací*:

- *Průnik* množin A a B je množina

$$A \cap B = \{x \mid x \in A \text{ a } x \in B\}$$

- *Sjednocení* množin A a B je množina

$$A \cup B = \{x \mid x \in A \text{ nebo } x \in B\}$$

- *Rozdíl* množin A a B je množina

$$A - B = \{x \mid x \in A \text{ a } x \notin B\}$$

Poznámka: Rozdíl množin A a B se též někdy označuje jako $A \setminus B$.

Příklad: Jestliže $A = \{a, b, c, d\}$ a $B = \{b, c, e, f\}$, pak $A \cup B = \{a, b, c, d, e, f\}$, $A \cap B = \{b, c\}$ a $A - B = \{a, d\}$.

Někdy jsou všechny množiny, které uvažujeme, podmnožinami nějaké jedné množiny U nazývané *universum*. Pokud se například bavíme o množinách přirozených čísel, pak je universem množina $\mathbb{N}$. Pro dané universum U definujeme *doplněk* množiny A jako $\overline{A} = U - A$.

Pro libovolné množiny $A, B \subseteq U$ platí DeMorganovy zákony:

$$\overline{A \cap B} = \overline{A} \cup \overline{B} \qquad \overline{A \cup B} = \overline{A} \cap \overline{B}$$

Množiny A a B jsou *disjunktní*, jestliže nemají žádný společný prvek, tj. jestliže $A \cap B = \emptyset$.

Počet prvků dané množiny S se nazývá její *kardinalita* (resp. *velikost*) a označuje se $|S|$. Dvě množiny mají stejnou kardinalitu, jestliže existuje bijekce (tj. vzájemně jednoznačné zobrazení) mezi jejich prvky. (Pozn.: Pojem bijekce je podrobněji definován v Sekci 1.3).

Kardinalita prázdné množiny je $|\emptyset| = 0$. Množina je *konečná*, jestliže její kardinalita je přirozené číslo. V opačném případě je množina *nekonečná*. Množina S je *spočetná*, jestliže je konečná nebo pokud existuje bijekce mezi S a $\mathbb{N}$ (tj. pokud je možné všechny její prvky očíslovat přirozenými čísly). Množina, která není spočetná, je *nespočetná*.

Příklad: Množiny $\mathbb{N}$, $\mathbb{Z}$ a $\mathbb{Q}$ jsou spočetné, množina $\mathbb{R}$ je nespočetná.

Množina všech podmnožin množiny S se nazývá *potenční množina* množiny S a označuje se zápisem $\mathcal{P}(S)$.

Pokud například $S = \{a, b, c\}$, pak

$$\mathcal{P}(S) = \{\emptyset,\, \{a\},\, \{b\},\, \{c\},\, \{a, b\},\, \{a, c\},\, \{b, c\},\, \{a, b, c\}\}\,.$$

Pokud je množina S konečná, pak $|\mathcal{P}(S)| = 2^{|S|}$.

Poznámka: Často se také používá pro označení potenční množiny místo $\mathcal{P}(S)$ výraz 2^S.

Uspořádaná dvojice prvků a a b, označovaná (a, b), je formálně definována jako množina $(a, b) = \{a, \{a, b\}\}$. Všimněte si, že na rozdíl od množiny u uspořádané dvojice záleží na pořadí prvků – (a, b) je něco jiného než (b, a). Analogicky můžeme definovat uspořádané trojice, čtveřice atd.

Kartézský součin množin A a B, označovaný $A \times B$, je množina všech uspořádaných dvojic, kde první prvek z dvojice patří do množiny A a druhý do množiny B:

$$A \times B = \{(a, b) \mid a \in A,\, b \in B\}$$

Příklad: $\{a, b\} \times \{a, b, c\} = \{(a, a),\, (a, b),\, (a, c),\, (b, a),\, (b, b),\, (b, c)\}$

Jestliže A a B jsou konečné množiny, pak $|A \times B| = |A| \cdot |B|$.

Kartézský součin n množin $A_1, A_2, \ldots, A_n$ je množina *n-tic*

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) \mid a_i \in A_i,\, i = 1, 2, \ldots, n\}$$

Jestliže všechna A_i jsou konečné množiny, platí

$$|A_1 \times A_2 \times \cdots \times A_n| = |A_1| \cdot |A_2| \cdots |A_n|$$

Místo kartézského součinu $A \times A \times \cdots \times A$, kde se množina A vyskytuje n krát, píšeme A^n. Pro konečnou množinu A platí $|A^n| = |A|^n$.

1.1.1 Potenciální a aktuální nekonečno

Pro lepší pochopení pojmu nekonečnosti množiny si připomeňme jeden ze základních *antických paradoxů* – závod Achilla se želvou: Achilles běží desetkrát rychleji než želva, ale želva má 100 m náskok. Než Achilles těchto 100 m proběhne, želva je dalších 10 m před ním, po uběhnutí dalších 10 m má želva stále 1 m náskok, atd... Vždy, když Achilles náskok želvy doběhne, želva se posune ještě o kousek dále. Předběhne Achilles vůbec někdy želvu?

Tento filozofický paradox vzniká z rozdílných pohledů na pojmy konečné a nekonečné velikosti, které stručně shrneme takto:

- *„Omezeně velké“* — uvažujeme množiny/objekty, jejichž velikost je omezená (jakoukoliv) předem danou konstantou.
- *„Potenciální nekonečno“* — uvažujeme stále ještě konečné množiny/objekty, ale jejich velikost nelze shora omezit univerzální konstantou. Jinými slovy, ke každému objektu nalezneme mezi uvažovanými ještě větší objekt. Také se říká, že velikost *r*oste nade všechny meze.
- *„Aktuální nekonečno“* — uvažujeme skutečně nekonečné množiny v celé jejich šíři.

Matematická odpověď na Achillův paradox je dána rozdílem mezi chápáním potenciálního a aktuálního nekonečna. Při uvedené paradoxní úvaze stále přidáváme další a další (zkracující se) úseky běhu, ale nikdy nevidíme dráhu běhu v celku (rozdělenou na nekonečně mnoho čím dál menších úseků, tj. jako aktuální nekonečno). Samozřejmě Achilles želvu předběhne ve vzdálenosti $111.\bar{1}$ m.

1.2 Relace

Relace na množinách $A_1, A_2, \ldots, A_n$ je libovolná podmnožina kartézského součinu $A_1 \times A_2 \times \cdots \times A_n$. Relace na n množinách se nazývá *n-ární* relace. Jestliže $n = 2$, jedná se o *binární* relaci. Jestliže $n = 3$, jedná se o *ternární* relaci.

V případě, že $A_1 = A_2 = \cdots = A_n$ hovoříme o *homogenní* relaci, v opačném případě o relaci *heterogenní*.

Když říkáme, že R je n-ární relace na množině A, máme tím na mysli, že $R \subseteq A^n$. Nejčastěji uvažované relace jsou binární relace. Proto, když řekneme, že R je relace na A, máme tím většinou na mysli, že R je binární relace na množině A, tj. $R \subseteq A \times A$. Ve zbytku této sekce se zaměříme na binární relace.

Jestliže $R \subseteq A \times B$ je binární relace, někdy místo $(a, b) \in R$ používáme infixový zápis a píšeme $a\ R\ b$.

Příklad: Relace „menší než“ na množině přirozených čísel je množina

$$\{(a, b) \in \mathbb{N} \times \mathbb{N} \mid a < b\}$$

Binární relace $R \subseteq A \times A$ je:

- *reflexivní*, jestliže pro všechna $a \in A$ platí $(a, a) \in R$,
- *ireflexivní*, jestliže pro všechna $a \in A$ platí $(a, a) \notin R$,
- *symetrická*, jestliže pro všechna $a, b \in A$ platí, že pokud $(a, b) \in R$, pak $(b, a) \in R$,
- *asymetrická*, jestliže pro všechna $a, b \in A$ platí, že pokud $(a, b) \in R$, pak $(b, a) \notin R$,
- *antisymetrická*, jestliže pro všechna $a, b \in A$ platí, že pokud $(a, b) \in R$ a $(b, a) \in R$, pak $a = b$,
- *tranzitivní*, jestliže pro všechna $a, b, c \in A$ platí, že pokud $(a, b) \in R$ a $(b, c) \in R$, pak $(a, c) \in R$.

Příklad:

- Relace "=" na $\mathbb{N}$ je reflexivní, symetrická, antisymetrická a tranzitivní, ale není ireflexivní ani asymetrická.
- Relace "$\leq$" na $\mathbb{N}$ je reflexivní, antisymetrická a tranzitivní, ale není ireflexivní, symetrická ani asymetrická.
- Relace "$<$" na $\mathbb{N}$ je ireflexivní, asymetrická, antisymetrická a tranzitivní, ale není reflexivní ani symetrická.

Reflexivní uzávěr relace $R \subseteq A \times A$ je nejmenší reflexivní relace $R' \subseteq A \times A$ taková, že $R \subseteq R'$. Pojmem „nejmenší" zde máme na mysli to, že neexistuje žádná reflexivní relace R'' taková, že $R \subseteq R'' \subset R'$.

Symetrický uzávěr relace $R \subseteq A \times A$ je nejmenší symetrická relace $R' \subseteq A \times A$ taková, že $R \subseteq R'$.

Tranzitivní uzávěr relace $R \subseteq A \times A$ je nejmenší tranzitivní relace $R' \subseteq A \times A$ taková, že $R \subseteq R'$.

Reflexivní a tranzitivní uzávěr relace $R \subseteq A \times A$ je nejmenší relace $R' \subseteq A \times A$ taková, že $R \subseteq R'$ a R' je současně reflexivní i tranzitivní.

1.2.1 Ekvivalence

Binární relace R na množině A je *ekvivalence* právě tehdy, když je reflexivní, symetrická a tranzitivní.

Jestliže R je ekvivalence na množině A, pak *třídou ekvivalence* prvku $a \in A$ je množina $[a]_R = \{b \in A \mid (a, b) \in R\}$, tj. množina všech prvků s ním ekvivalentních. Jestliže je ekvivalence R zřejmá z kontextu, píšeme místo $[a]_R$ jen $[a]$.

Mějme množinu A. Množina jejích podmnožin $\mathcal{A} = \{A_i\}$ tvoří *rozklad* na množině A jestliže:

- všechny množiny A_i jsou vzájemně disjunktní, tj. jestliže pro libovolné $A_i, A_j \in \mathcal{A}$ platí $A_i \cap A_j = \emptyset$ pokud $i \neq j$, a
- sjednocením množin z $\mathcal{A}$ je množina A, tj.

$$A = \bigcup_{A_i \in A} A_i$$

Ekvivalence $R \subseteq A \times A$ definuje na A rozklad $\{\, [a]_R \mid a \in A \,\}$.

Naopak rozklad $\mathcal{A} = \{A_i\}$ na množině A definuje ekvivalenci

$$R = \{(a, b) \subseteq A \times A \mid a, b \in A_i \text{ pro nějaké } A_i \in \mathcal{A}\}\,.$$

1.2.2 Uspořádání

Binární relace R na množině A je *(částečné a neostré) uspořádání*, jestliže je reflexivní, tranzitivní a antisymetrická.

Binární relace R na množině A je *(částečné) ostré uspořádání*, jestliže je asymetrická a tranzitivní. (Pozn.: Z toho, že je R asymetrická plyne, že je také ireflexivní a antisymetrická.)

Pro neostrá uspořádání se obvykle používají symboly jako $\leq$ a jemu podobné, pro ostrá uspořádání pak symboly jako $<$ a jemu podobné.

Ke každému neostrému uspořádání R na množině A existuje odpovídající ostré uspořádání $R' = R - \{(a, a) \mid a \in A\}$. Naopak ke každému ostrému uspořádání S na množině A existuje odpovídající neostré uspořádání $S' = S \cup \{(a, a) \mid a \in A\}$

Uspořádání (ať už neostré či ostré) $R \subseteq A \times A$ je *úplné* (nebo také *lineární*), jestliže pro všechna $a, b \in A$ platí buď $(a, b) \in R$, $(b, a) \in R$ nebo $a = b$ (tj. pokud neexistují vzájemně nesrovnatelné prvky).

Mějme libovolné neostré uspořádání $\leq$ na množině A.

- Prvek $a \in A$ je *minimální prvek* množiny A, jestliže v A neexistuje menší prvek než a (tj. z $x \leq a$ plyne $x = a$).
- Prvek $a \in A$ je *maximální prvek* množiny A, jestliže v A neexistuje větší prvek než a (tj. z $a \leq x$ plyne $a = x$).
- Prvek $a \in A$ je *nejmenší prvek* množiny A, jestliže je menší než všechny ostatní prvky v A (tj. pro každé $x \in A$ platí $a \leq x$).
- Prvek $a \in A$ je *největší prvek* množiny A, jestliže je větší než všechny ostatní prvky v A (tj. pro každé $x \in A$ platí $x \leq a$).

- Prvek $a \in A$ je *infimum* množiny B (píšeme $a = \inf B$), jestliže a je největší ze všech prvků, které jsou menší než všechny prvky z B, tj. platí

$$(\forall x \in B)(a \leq x) \ \wedge \ (\forall b)((\forall x \in B)(b \leq x) \Rightarrow b \leq a)$$

- Prvek $a \in A$ je *supremum* množiny B (píšeme $a = \sup B$), jestliže a je nejmenší ze všech prvků, které jsou větší než všechny prvky z B, tj. platí

$$(\forall x \in B)(x \leq a) \ \wedge \ (\forall b)((\forall x \in B)(x \leq b) \Rightarrow a \leq b)$$

1.3 Funkce

Funkce f z množiny A do množiny B je binární relace $f \subseteq A \times B$ taková, že pro každé $a \in A$ existuje právě jedno $b \in B$ takové, že $(a, b) \in f$. Množina A se nazývá *definiční obor* funkce f, množina B se nazývá *obor hodnot* funkce f.

To, že f je funkce z množiny A do množiny B obvykle zapisujeme jako $f : A \to B$. Místo $(a, b) \in f$ obvykle píšeme $b = f(a)$, neboť volbou prvku a je prvek b jednoznačně určen. Funkce $f : A \to B$ tedy každému prvku z A přiřazuje právě jeden prvek z B. Jestliže $b = f(a)$, říkáme, že a je *argumentem* funkce f a že b je *hodnotou* funkce f v bodě a.

Poznámka: Výše uvedená definice se týká tzv. *totální* funkce, tj. funkce, jejíž hodnota je definovaná pro každou hodnotu argumentu. Někdy má smysl uvažovat také tzv. *částečné (parciální) funkce*, tj. funkce, jejichž hodnota není pro některé hodnoty argumentu definována. Formálně je částečná funkce $f : A \to B$ definována jako relace $f \subseteq A \times B$ taková, že pro každé $a \in A$ existuje nejvýše jedno $b \in B$ takové, že $(a, b) \in f$. Pokud budeme v dalším textu mluvit o funkci a neuvedeme jinak, budeme mít vždy na mysli funkci totální.

Konečná posloupnost (sekvence) délky n je funkce, jejímž definičním oborem je množina $\{0, 1, \ldots, n-1\}$. Konečnou posloupnost obvykle zapisujeme tak, že vypíšeme její hodnoty: $f(0), f(1), \ldots, f(n-1)$. *Nekonečná posloupnost*

(sekvence) je funkce, jejímž definičním oborem je $\mathbb{N}$. Nekonečnou posloupnost někdy zapisujeme tak, že uvedeme několik prvních prvků, za kterými následují tři tečky: $f(0), f(1), \ldots$

Jestliže definičním oborem funkce f je kartézský součin, obvykle vynecháváme jeden pár závorek v zápise argumentu funkce f. Pokud například máme funkci $f : A_1 \times A_2 \times \cdots \times A_n \to B$, pak místo $b = f((a_1, a_2, \ldots, a_n))$ píšeme $b = f(a_1, a_2, \ldots, a_n)$.

Místo o funkci někdy též mluvíme o *operaci*. Speciálně, v případě funkce f typu $f : A_1 \times A_2 \times \cdots \times A_n \to B$ mluvíme o n-ární operaci. V případě, že $n = 2$, mluvíme o *binární* operaci. Jestliže říkáme, že f je n-ární operace na množině A, rozumíme tím, že f je funkce typu $f : A^n \to A$. Jestliže říkáme, že f je unární operace na množině A, rozumíme tím, že f je funkce typu $f : A \to A$. Jestliže říkáme, že f je binární operace na množině A, rozumíme tím, že f je funkce typu $f : A \times A \to A$.

Mějme funkci $f : A^n \to A$. Množina $B \subseteq A$ je *uzavřená na operaci* f, jestliže z $a_1, a_2, \ldots, a_n \in B$ plyne, že $f(a_1, a_2, \ldots, a_n) \in B$.

Jestliže $f : A \to B$ je funkce a $b = f(a)$, pak někdy také říkáme, že b je *obrazem* a. Obrazem množiny $A' \subseteq A$ je množina

$$f(A') = \{b \in B \mid b = f(a) \text{ pro nějaké } a \in A'\}\,.$$

Funkce $f : A \to B$ je:

- *surjektivní* (je surjekcí, je zobrazením na), jestliže $f(A) = B$,
- *injektivní* (je injekcí, je prostá), jestliže z $a \neq a'$ plyne $f(a) \neq f(a')$,
- *bijektivní* (je bijekcí, je vzájemně jednoznačným zobrazením), jestliže je současně surjektivní i injektivní.

Jestliže funkce f je bijekcí, pak funkce *inverzní* k funkci f, označovaná f^{-1}, je definována takto: $f^{-1}(b) = a$ právě když $f(a) = b$.

Předpokládejme nyní funkci $f : A \times A \to A$. Funkce f je *asociativní*, jestliže pro libovolné prvky $a, b, c \in A$ platí

$$f(f(a, b), c) = f(a, f(b, c))\,.$$

Funkce f je *komutativní*, jestliže pro libovolné prvky $a, b \in A$ platí

$$f(a, b) = f(b, a)\,.$$

Prvek $z \in A$ je *nulovým prvkem* vzhledem k funkci f, jestliže pro libovolné $a \in A$ platí $f(z, a) = f(a, z) = z$. Prvek $e \in A$ je *jednotkovým prvkem* vzhledem k funkci f, jestliže pro libovolné $a \in A$ platí $f(e, a) = f(a, e) = a$. Dá se ukázat, že ke každé funkci existuje nejvýše jeden nulový a nejvýše jeden jednotkový prvek.

Jestliže k funkci f existuje jednotkový prvek e, pak $b \in A$ je *inverzním prvkem* k prvku $a \in A$ právě tehdy, když $f(a, b) = f(b, a) = e$.

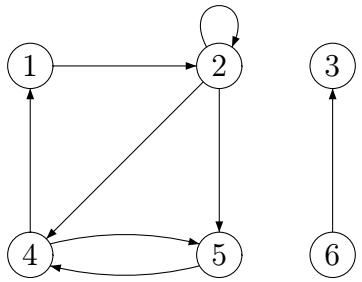
Poznámka: Pro funkce typu $f : A \times A \to A$ je často vhodnější používat infixovou notaci a používat jako název funkce nějaký speciální symbol. Mějme například funkci $\otimes : A \times A \to A$. Pak místo $\otimes(a, b)$ píšeme $a \otimes b$. Asociativita $\otimes$ pak znamená, že pro libovolné $a, b, c \in A$ platí $(a \otimes b) \otimes c = a \otimes (b \otimes c)$, a komutativita, že pro libovolné $a, b \in A$ platí $a \otimes b = b \otimes a$.

1.4 Grafy

Rozlišujeme dva základní typy grafů – orientované a neorientované.

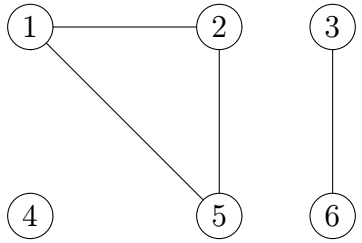
Orientovaný graf G je dvojice (V, E), kde V je konečná množina *vrcholů* a $E \subseteq V \times V$ je množina *hran*. Orientovaný graf můžeme znázornit obrázkem, kde vrcholy znázorníme jako kolečka a hrany jako šipky vedoucí mezi těmito kolečky. Příklad takového grafu je na Obrázku 1.1, kde je znázorněn graf $G = (V, E)$, kde $V = \{1, 2, 3, 4, 5, 6\}$ a $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. Všimněte si, že v orientovaném grafu mohou existovat *smyčky* – hrany vedoucí z vrcholu do něho samého.

Neorientovaný graf je dvojice $G = (V, E)$, kde význam V a E je podobný jako u orientovaného grafu, ale na rozdíl od orientovaného grafu je E množinou *neuspořádaných* dvojic vrcholů, tj. hrana v neorientovaném grafu je množina $\{u, v\}$, kde $u, v \in V$ a $u \neq v$. Podobně jako u orientovaného grafu se však obvykle při zápisu hrany používá notace (u, v) místo $\{u, v\}$, s tím, že v případě neorientovaného grafu se (u, v) a (v, u) považují za jednu a tutéž hranu. V případě neorientovaného grafu nejsou povoleny smyčky. Neorientovaný graf se znázorňuje podobně jako orientovaný, na konci čar, které



Obrázek 1.1: Příklad orientovaného grafu

představují hrany však nekreslíme šipky. Příklad neorientovaného grafu je uveden na Obrázku 1.2. Jedná se o graf $G = (V, E)$, kde $V = \{1, 2, 3, 4, 5, 6\}$ a $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$.



Obrázek 1.2: Příklad neorientovaného grafu

Poznámka: Obecně můžeme uvažovat i nekonečné grafy, kde je množina vrcholů nekonečná. Pokud však neuvedeme jinak, při použití pojmu „graf“ budeme mít vždy na mysli konečný graf.

U orientovaného grafu říkáme, že hrana (u, v) *vychází* z vrcholu u a *vstupuje* do vrcholu v. U neorientovaného grafu říkáme, že hrana (u, v) je *incidentní* s vrcholy u a v.

Stupeň vrcholu v neorientovaném grafu je počet hran, které jsou s tímto vrcholem incidentní. V orientovaném grafu je *vstupní stupeň vrcholu* počet hran, které do daného vrcholu vstupují, *výstupní stupeň vrcholu* počet hran,

které z daného vrcholy vystupují a *stupeň vrcholu* je součet jeho vstupního a výstupního stupně.

Grafy $G = (V, E)$ a $G' = (V', E')$ jsou *isomorfní*, jestliže existuje bijekce $f : V \to V'$ taková, že $(u, v) \in E$ právě tehdy, když $(f(u), f(v)) \in E'$.

Graf $G' = (V', E')$ je podgrafem grafu $G = (V, E)$, jestliže $V' \subseteq V$ a $E' \subseteq E$. Pro danou množinu $V' \subseteq V$ je podgraf grafu G indukovaný množinou V' definován jako graf $G' = (V', E')$, kde

$$E' = \{(u, v) \in E \mid u, v \in V'\}.$$

Mějme libovolný (ať už orientovaný či neorientovaný) graf $G = (V, E)$. *Sled* je libovolná posloupnost $v_0, v_1, \ldots, v_k$, kde $k \geq 0$ a (v_{i-1}, v_i) pro $i = 1, 2, \ldots, k$. Tento sled obsahuje vrcholy $v_0, v_1, \ldots, v_k$ a hrany (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k). *Tah* je sled, kde se neopakují hrany. *Cesta* je tah, kde se neopakují vrcholy. *Cyklus* je tah, kde $k > 0$, $v_0 = v_k$ a všechny vrcholy kromě v_0 a v_k jsou navzájem různé. V případě neorientovaného grafu se cyklu též říká *kružnice*. *Délka* sledu (tahu, cesty, cyklu, kružnice) $v_0, v_1, \ldots, v_k$ je k.

Poznámka: Ve výše uvedených definicích jsme připouštěli vždy nejvýše jednu hranu mezi každou dvojicí vrcholů. Někdy je užitečné povolit mezi dvěma vrcholy více než jednu hranu. Tomuto typu grafů se říká *multigrafy*. V případě multigrafu by bylo třeba poněkud upravit definici sledu a dalších z něho odvozených pojmů (tah, cesta, cyklus) tak, aby zahrnoval informaci o hranách. Sled bychom definovali jako posloupnost

$$v_0, e_1, v_1, e_2, v_3, \ldots, e_k, v_k$$

kde se střídají vrcholy a hrany (tj. $v_0, v_1, \ldots, v_k \in V$ a $e_1, e_2, \ldots, e_k \in E$, a hrana e_i vede z vrcholu v_{i-1} do v_i pro $i = 1, 2, \ldots, k$).

Vrchol v je *dosažitelný* z vrcholu u jestliže existuje cesta z vrcholu u do vrcholu v, tj. cesta $v_0, v_1, \ldots, v_k$, kde $u = v_0$ a $v = v_k$. Všimněte si, že vždy existuje cesta délky 0 z vrcholu u do vrcholu u.

Neorientovaný graf je *souvislý*, jestliže mezi každými dvěma jeho vrcholy existuje cesta. *Souvislá* komponenta neorientovaného grafu G je maximální indukovaný podgraf grafu G (maximální vzhledem k relaci inkluze na množinách vrcholů). Pro graf $G = (V, E)$ můžeme definovat relaci „dosažitelnosti“ $R_G \subseteq V \times V$ jako

$$R_G = \{(u, v) \in V \times V \mid \text{v } G \text{ existuje cesta z } u \text{ do } v\}$$

Všimněte si, že v případě neorientovaného grafu je relace R_G ekvivalencí, a množiny vrcholů jednotlivých souvislých komponent grafu G odpovídají třídám ekvivalence relace R_G.

Mějme orientovaný graf $G = (V, E)$ a definujme pro něj relaci dosažitelnosti R_G stejně jako v předchozím případě. Graf G je *silně souvislý*, jestliže pro každou dvojici vrcholů $u, v \in V$ platí $(u, v) \in R_G$ a $(v, u) \in R_G$. *Silně souvislá komponenta* grafu G je silně souvislý podgraf grafu G. Všimněte si, že množina maximálních silně souvislých komponent grafu G indukuje rozklad na množině vrcholů grafu G.

Graf je *acyklický* jestliže neobsahuje cyklus. Neorientovaný acyklický graf se nazývá *les*. Souvislý les se nazývá *strom*. Orientovaný acyklický graf se někdy nazývá *dag* (z anglického „directed acyclic graph“).

Úplný graf je neorientovaný graf, kde jsou každé dva vrcholy spojeny hranou. Mějme neorientovaný graf $G = (V, E)$ a množinu $V' \subseteq V$. Množina V' tvoří *kliku*, jestliže podgraf grafu G indukovaný množinou V' je úplný, tj. jestliže každé dva vrcholy z V' jsou v G spojeny hranou. Množina V' tvoří *nezávislou množinu*, jestliže žádné dva vrcholy z V' nejsou v G spojeny hranou. Nezávislé množině se též někdy říká *antiklika*.

Bipartitní graf je neorientovaný graf $G = (V, E)$, kde V můžeme rozdělit do dvou disjunktních množin V_1, V_2 takových, že pro libovolnou hranu $(u, v) \in E$ platí buď $u \in V_1$ a $v \in V_2$ nebo $u \in V_2$ a $v \in V_1$, tj. hrany vedou jen mezi V_1 a V_2.

1.5 Stromy

Připomeňme, že strom je souvislý acyklický neorientovaný graf.

Mějme neorientovaný graf $G = (V, E)$. Všechna následující tvrzení jsou ekvivalentní v tom smyslu, že z platnosti libovolného z nich plyne i platnost všech ostatních. Tato tvrzení zachycují některé důležité vlastnosti stromů:

- G je strom.
- Libovolné dva vrcholy grafu G jsou spojeny právě jednou cestou.
- G je souvislý, ale po odebrání libovolné hrany z E už výsledný graf souvislý není.

- G je souvislý a $|E| = |V| - 1$.
- G je acyklický a $|E| = |V| - 1$.
- G je acyklický, ale přidání libovolné hrany do E vznikne cyklus.

Kořenový strom je strom, kde je jeden z vrcholů označen jako *kořen* stromu.

Uvažujme kořenový strom T s kořenem r a jeden z jeho vrcholů x. Z vlastností (obecných) stromů plane, že existuje právě jedna cesta z r do x. Libovolný vrchol y ležící na této cestě je *předchůdcem* vrcholu x. Jestliže y je předchůdcem x, pak x je *následníkem* y. (Všimněte si, že vrchol je svým vlastním předchůdcem i následníkem.) Vrchol y je *rodičem* vrcholu x, jestliže y je posledním předchůdcem x na cestě z r do x (tj. hrana (y, x) je poslední hranou na této cestě). Vrchol x je *potomkem* vrcholu y, jestliže y je rodičem x.

Kořen r je jediným vrcholem, který nemá rodiče. Vrchol, který nemá žádné potomky se nazývá *list*. Vrchol, který není listem se nazývá *vnitřní vrchol*.

Délka cesty z kořene r do vrcholu x se označuje jako *hloubka* vrcholu x. (Kořen má tedy hloubku 0.) *Výška* vrcholu x je délka nejdelší cesty z x do některého z jeho následníků. *Výška stromu* je výška kořene stromu. Všimněte si, že výška stromu je totéž co maximální hloubka vrcholu ve stromě.

Podstrom stromu T s kořenem ve vrcholu x je podgraf grafu T indukovaný množinou všech následníků vrcholu x (včetně x), kde x je zvolen jako kořen.

Uspořádaný strom je kořenový strom, ve kterém jsou potomci každého vrcholu seřazeni.

Binární strom je kořenový strom, ve kterém má každý vrchol x nejvýše dva následníky, přičemž pokud má následníky dva, tak jeden z nich je označen jako jeho *levý* potomek a druhý jako jeho *pravý* potomek, a pokud má následníka jediného, tak ten je označen buď jako jeho levý nebo jako jeho pravý potomek. Přirozeně pak můžeme mluvit o levém a pravém podstromu.

Úplný k-ární strom je strom, kde všechny vnitřní vrcholy mají právě k potomků a všechny listy mají stejnou hloubku. *Úplný binární strom* je úplný 2-ární strom. Jestliže je výška úplného k-árního stromu h, pak tento strom obsahuje k^h listů a počet jeho vnitřních vrcholů je

$$1 + k + k^2 + \cdots + k^{h-1} = \sum_{i=0}^{h-1} k^i = \frac{k^h - 1}{k - 1}$$

Úplný binární strom výšky h tedy obsahuje 2^h listů a $2^h - 1$ vnitřních vrcholů.

1.6 Výroková logika

Předpokládejme, že máme dánu množinu *atomických výrokových symbolů* At (stručněji nazýváme prvky množiny At výrokové symboly nebo atomy), která neobsahuje žádný ze symbolů $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, (,)$. Symboly $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ představují *logické spojky* a nazývají se *negace* ($\neg$), *konjunkce* ($\wedge$), *disjunkce* ($\vee$), *implikace* ($\Rightarrow$) a *ekvivalence* ($\Leftrightarrow$).

Množina všech *výrokových formulí* je nejmenší množina všech výrazů sestavených z prvků množiny $At \cup \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, (,)\}$, splňující následující podmínky:

- každý výrokový symbol z množiny At je výroková formule,
- je-li φ výroková formule, pak i $\neg\varphi$ je výroková formule,
- jsou-li φ a ψ výrokové formule, pak i $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \Rightarrow \psi)$ a $(\varphi \Leftrightarrow \psi)$ jsou výrokové formule.

Abychom se vyhnuli zápisu velkého počtu závorek, používají se následující konvence, které umožňují část závorek vypustit: Není třeba psát vnější pár závorek. Je definována priorita logických spojek v následujícím pořadí $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ (tj. nejvyšší prioritu má $\neg$ a nejnižší $\Leftrightarrow$) – spojky s vyšší prioritou vážou silněji. Implikace $\Rightarrow$ je asociativní směrem doprava (tj. formuli $x_1 \Rightarrow x_2 \Rightarrow x_3 \Rightarrow y$ chápeme jako $x_1 \Rightarrow (x_2 \Rightarrow (x_3 \Rightarrow y))$). Logické spojky $\wedge$ a $\vee$ jsou asociativní, tj. $x \wedge y \wedge z$ je to samé jako $(x \wedge y) \wedge z$ nebo $x \wedge (y \wedge z)$, a $x \vee y \vee z$ je to samé jako $(x \vee y) \vee z$ nebo $x \vee (y \vee z)$.

Poznámka: Pro negaci se též někdy používá symbol $\sim$, pro konjunkci symbol $\&$, pro implikaci symboly $\rightarrow$ a $\supset$, a pro ekvivalenci symboly $\leftrightarrow$ a $\equiv$.

Pravdivostní (booleovské) ohodnocení je libovolná funkce $\nu : At \rightarrow \{0, 1\}$. Hodnoty 0 a 1 představují pravdivostní hodnoty – 0 nepravdu a 1 pravdu. Místo 0 a 1 také někdy používáme ve stejném významu hodnoty FALSE a TRUE. Pravdivostní hodnotu, která je přiřazena formuli φ při daném pravdivostním ohodnocení ν označujeme $[\varphi]_\nu$ a definujeme:

- $[x]_\nu = \nu(x)$ pro $x \in At$,
- $[\neg\varphi]_\nu = 1$, právě když $[\varphi]_\nu = 0$,
- $[\varphi \wedge \psi]_\nu = 1$, právě když $[\varphi]_\nu = 1$ a $[\psi]_\nu = 1$,

- $[\varphi \vee \psi]_\nu = 1$, právě když $[\varphi]_\nu = 1$ nebo $[\psi]_\nu = 1$,
- $[\varphi \Rightarrow \psi]_\nu = 1$, právě když $[\varphi]_\nu = 0$ nebo $[\psi]_\nu = 1$,
- $[\varphi \Leftrightarrow \psi]_\nu = 1$, právě když $[\varphi]_\nu = [\psi]_\nu$.

Výše popsaný význam logických spojek je možné znázornit pomocí následujících *pravdivostní tabulek* (poznamenejme, že u spojek, které mají dva operandy se řádky tabulky vztahují k levému operandu a sloupce k pravému):

$\neg$	
0	1
1	0

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

$\Rightarrow$	0	1
0	1	1
1	0	1

$\Leftrightarrow$	0	1
0	1	0
1	0	1

Poznámka: Ne všechny logické spojky je třeba definovat jako základní. Můžeme vybrat jen některé z nich a zbylé pak definovat pomocí nich. Například bychom mohli vzít jako základní logické spojky pouze $\neg$ a $\wedge$. Formule $\varphi \vee \psi$ je pak definována jako zkratka pro $\neg(\neg\varphi \wedge \neg\psi)$, formule $\varphi \Rightarrow \psi$ jako zkratka pro $\neg\varphi \vee \psi$ a formule $\varphi \Leftrightarrow \psi$ jako zkratka pro $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$.

Dále se někdy zavádí speciální logické konstanty $\top$ a $\bot$ označující pravdu a nepravdu, které se syntakticky používají stejně jako výrokové atomy, ovšem s tím, že při každém ohodnocení ν platí $[\top]_\nu = 1$ a $[\bot]_\nu = 0$. Konstanty $\top$ a $\bot$ můžeme považovat za „nulární“ logické spojky. Formálně je můžeme definovat jako zkratky zastupující formule $(x \vee \neg x)$ a $(x \wedge \neg x)$.

Výrokové formule φ a ψ jsou *logicky ekvivalentní*, jestliže pro každé ohodnocení ν platí $[\varphi]_\nu = [\psi]_\nu$.

Výroková formule φ je *splnitelná*, jestliže existuje ohodnocení ν takové, že $[\varphi]_\nu = 1$. Formule φ je *tautologií*, jestliže pro každé ohodnocení ν platí $[\varphi]_\nu = 1$. Formule φ je *kontradikcí*, jestliže pro každé ohodnocení ν platí $[\varphi]_\nu = 0$. Tautologie je tedy logicky ekvivalentní formuli $\top$ a kontradikce je logicky ekvivalentní formuli $\bot$.

Literál je formule tvaru x nebo $\neg x$, kde $x \in At$. Disjunkce několika literálů se nazývá *klauzule*. O formuli, která je konjunkcí klauzulí říkáme, že je v *konjunktivní normální formě*. Formule je v *disjunktivní normální formě*, jestliže je disjunkcí formulí, z nichž každá je konjunkcí literálů.

Každá výroková formule je ekvivalentní s nějakou formulí v konjunktivní normální formě a s nějakou formulí v disjunktivní normální formě.

1.7 Další značení

Pokud x je reálné číslo, $\lfloor x \rfloor$ označuje největší celé číslo y takové, že $y \leq x$. Podobně $\lceil x \rceil$ označuje nejmenší celé číslo y takové, že $y \geq x$. Jinými slovy, zápis $\lfloor x \rfloor$ označuje zaokrouhlení čísla x „dolů“ na nejbližší celé číslo a zápis $\lceil x \rceil$ zaokrouhlení čísla x „nahoru“.

Příklad: $\lfloor 3.14 \rfloor = 3$, $\lceil 3.14 \rceil = 4$, $\lfloor -3.14 \rfloor = -4$, $\lceil -3.14 \rceil = -3$

1.8 Cvičení

Otázky:

Otázka 1.1: Je množina všech čísel vyjádřitelných datovým typem `double` v jazyce C konečná?

Otázka 1.2: Je množina všech prvočísel konečná?

Otázka 1.3: Je množina všech prvočísel spočetná?

Otázka 1.4: Jak byste do jedné posloupnosti seřadili všechna celá čísla?

Otázka 1.5*: Jak byste do jedné posloupnosti seřadili všechna racionální čísla?

Cvičení 1.6: Pro každý z následujících formálních zápisů množin uveďte (svými slovy), jaké prvky daná množina obsahuje:

a) $\{1, 3, 5, 7, \ldots\}$

b) $\{\ldots, -4, -2, 0, 2, 4, \ldots\}$

c) $\{n \mid n = 2m \text{ pro nějaké } m \in \mathbb{N}\}$

d) $\{n \mid n = 2m \text{ pro nějaké } m \in \mathbb{N} \text{ a } n = 3k \text{ pro nějaké } k \in \mathbb{N}\}$

e) $\{n \in \mathbb{Z} \mid n = n + 1\}$

Cvičení 1.7: Popište vhodným formálním zápisem následující množiny:

a) Množina obsahující čísla 1, 10 a 100.

b) Množina obsahující všechna celá čísla větší než 5.

c) Množina obsahující všechna přirozená čísla menší než 5.

d) Množina neobsahující žádné prvky.

e) Množina všech podmnožin dané množiny X.

Cvičení 1.8: Uvažujme množiny $A = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ a $B = \{\mathtt{x}, \mathtt{y}\}$.

a) Je A podmnožinou B?

b) Je B podmnožinou A?

c) Co je $A \cup B$?

d) Co je $A \cap B$?

e) Co je $A \times B$?

f) Co je $\mathcal{P}(B)$?

Cvičení 1.9: Jestliže množina A má a prvků a množina B má b prvků, kolik prvků má množina $A \times B$?

Cvičení 1.10: Jestliže množina C má c prvků, kolik prvků má množina $\mathcal{P}(C)$?

Cvičení 1.11: Nechť $X = \{1, 2, 3, 4, 5\}$ a $Y = \{6, 7, 8, 9, 10\}$. Unární funkce $f : X \to Y$ a binární funkce $g : X \times Y \to Y$ jsou popsány následujícími tabulkami:

n	$f(n)$
1	6
2	7
3	6
4	7
5	6

g	6	7	8	9	10
1	10	10	10	10	10
2	7	8	9	10	6
3	7	7	8	8	9
4	9	8	7	6	10
5	6	6	6	6	6

a) Jaká je hodnota $f(2)$?

b) Co definičním oborem a oborem hodnot funkce f?

c) Jaká je hodnota $g(2, 10)$?

d) Co definičním oborem a oborem hodnot funkce g?

e) Jaká je hodnota $g(4, f(4))$?

Cvičení 1.12: Uveďte příklad relace, která je:

a) Reflexivní a symetrická, ale není tranzitivní.

b) Reflexivní a tranzitivní, ale není symetrická.

c) Symetrická a tranzitivní, ale není reflexivní.

Cvičení 1.13: Kde je chyba v následujícím důkazu toho, že všichni koně mají stejnou barvu?

Tvrzení: Pro libovolnou množinu n koňů platí, že všichni koně v této množině mají stejnou barvu.

Důkaz: Indukcí podle n.

- **Báze:** Pro $n = 1$ tvrzení zjevně platí, protože v množině obsahující právě jednoho koně mají všichni koně stejnou barvu.
- **Indukční krok:** Předpokládejme, že tvrzení platí pro $n = k$ pro nějaké $k \geq 1$. Ukážeme, že pak platí i pro $n = k+1$. Vezměme nějakou množinu K obsahující $k+1$ koňů. Z této množiny odebereme jednoho koně, čímž dostaneme nějakou množinu K_1 obsahující k koňů. Podle indukčního předpokladu mají všichni koně v K_1 stejnou barvu. Nyní vrátíme koně, kterého jsme předtím odebrali, a odebereme nějakého jiného koně, čímž dostaneme množinu K_2 obsahující k koňů. Stejně jako v předchozím případě mají podle indukčního předpokladu všichni koně v K_2 stejnou barvu. Z toho plyne, že všichni koně v množině K mají stejnou barvu.

Cvičení 1.14: Mějme neprázdnou konečnou množinu X, kde $|X| = n$. Uvažujme posloupnost ekvivalencí na této množině $\equiv_0, \equiv_1, \equiv_2, \ldots$, kde každá následující ekvivalence je zjemněním předchozí ekvivalence, tj. pro libovolné $i \geq 0$ platí, že z $x \equiv_{i+1} y$ plyne $x \equiv_i y$.

Označme $[x]_i$ třídu ekvivalence $\equiv_i$, do které patří prvek x, tj. $[x]_i = \{y \in X \mid y \equiv_i x\}$. Definuje množinu C jako množinu všech tříd všech těchto ekvivalencí, tj.

$$C = \bigcup_{i \geq 0} \{[x]_i \mid x \in X\}$$

a) Dokažte, že $|C| \leq 2n - 1$.

b) Ukažte, že pro libovolné $n \geq 1$ může nastat rovnost $|C| = 2n - 1$.

Cvičení 1.15: Co nejvíce zjednodušte dva následující výrazy:

$$\sum_{k=1}^{n} (2k - 1) \qquad \prod_{k=1}^{n} 2 \cdot 4^k$$

Cvičení 1.16: n-té harmonické číslo je definováno předpisem

$$H_n = \sum_{k=1}^{n} \frac{1}{k}$$

Ukažte, že pro všechna $n \geq 1$ platí

$$\ln(n + 1) \leq H_n \leq \ln n + 1$$

Cvičení 1.17:

a) Ukažte, že množina všech racionálních čísel $\mathbb{Q}$ je spočetná. (Ukažte, že existuje bijekce z množiny přirozených čísel $\mathbb{N}$ do množiny $\mathbb{Q}$.)

b) Předpokládejme, že množina X je spočetná. Ukažte, že množina všech konečných posloupností prvků z X je spočetná.

c) Ukažte, že množina všech reálných čísel $\mathbb{R}$ nespočetná.

d) Ukažte, že pro libovolnou nekonečnou spočetnou množinu X platí, že množina $\mathcal{P}(X)$ je nespočetná.

Cvičení 1.18: Uveďte příklad uspořádání na množině přirozených čísel, které není úplným uspořádáním.

Cvičení 1.19: Předpokládejme, že $n \geq 1$ je nějaké dané přirozené číslo. Ukažte, že relace

$$a \equiv b \pmod{n}$$

je ekvivalence.

Poznámka: Zápis $a \equiv b \pmod{n}$ znamená, že a i b dávají po dělení n stejný zbytek, tj. $(a \bmod n) = (b \bmod n)$.

Cvičení 1.20: Nechť S je konečná množina a $R \subseteq S \times S$ ekvivalence. Ukažte, že pokud relace R je současně také antisymetrická, pak její třídy ekvivalence jsou jednoprvkové množiny.

Cvičení 1.21: Je následující tvrzení pravdivé? Vyskytuje se v jeho důkazu nějaká chyba?

Tvrzení: Jestliže je relace R symetrická a tranzitivní, pak je i reflexivní.

Důkaz: Díky symetrii platí pro libovolné dva prvky a a b, že z aRb plyne bRa. Díky tranzitivitě z toho plyne aRa, čímž je důkaz hotov.

Cvičení 1.22:Předpokládejme, že A a B jsou konečné množiny a $f : A \to B$ je funkce. Ukažte, že:

- Jestliže f je injektivní, pak $|A| \leq |B|$.
- Jestliže f je surjektivní, pak $|A| \geq |B|$.

Cvičení 1.23:Je funkce $f(x) = x+1$ bijekcí na množině přirozených čísel $\mathbb{N}$? A na množině celých čísel $\mathbb{Z}$?

Cvičení 1.24:Navrhněte a podrobně popište algoritmus, který řeší následující problém:

Vstup: Orientovaný graf $G = (V, E)$, vrchol $s \in V$.

Výstup: Množina všech vrcholů dosažitelných z s.

Kolik operací váš algoritmus zhruba provede, jestliže dostane na vstupu graf, který má n vrcholů a m hran?

Cvičení 1.25:Uvažujme nějaký binární strom T, kde každý vrchol, který není listem, má dva potomky. Jakou minimální a maximální výšku může mít strom T, jestliže má celkem n vrcholů? (Výškou stromu myslíme vzdálenost od kořene k nejvzdálenějšímu vrcholu.)

Cvičení 1.26:Ukažte, že binární strom, který má n listů, má $n - 1$ vrcholů stupně 2 (tj. vrcholů, které mají 2 potomky).

Kapitola 2

Formální jazyky

Cíle kapitoly:

- Seznámení se s pojmy jako (formální) abeceda, slovo, formální jazyk a s operacemi na slovech a jazycích.

2.1 Formální abeceda a jazyk

Teoretická informatika poskytuje formální základy a nástroje pro praktické informatické aplikace (jako programování či softwarové inženýrství). Jedním z jejích důležitých úkolů je matematicky popsat různé typy algoritmických problémů a výpočtů. Pro matematický popis vstupů a výstupů problémů (výpočtů) je užitečné nejprve zavést pojmy jako jsou abeceda, slovo, (formální) jazyk.

Použitá symbolická abeceda pro vstupy a výstupy výpočtů závisí na dohodnuté formě zápisu. V počítačové praxi využíváme např. binární abecedu $\{0, 1\}$, hexadecimálí abecedu $\{0, 1, \ldots, 9, A, \ldots, F\}$ nebo „textovou“ abecedu typu ASCII či nověji UTF-8. Matematicky můžeme za abecedu považovat libovolnou (dohodnutou) konečnou množinu symbolů; převody zápisů mezi různými abecedami jsou přímočaré. (V konkrétním případě obvykle volíme abecedu, která se přirozeně hodí k danému problému.)

Důležitým pojmem je „slovo“, což znamená libovolný konečný řetězec symbolů nad danou abecedou; pokud je v abecedě mezera, nemá žádný zvláštní

význam. (Jakkoli vymezená) množina slov se nazývá (formálním) „jazykem“. Uvedené pojmy nyní přesně nadefinujeme a zároveň zavedeme důležité operace s formálními jazyky.

Definice 2.1
Abecedou myslíme libovolnou konečnou množinu, obvykle označovanou Σ . Prvky Σ nazýváme *symboly* (písmena, znaky) této abecedy. (Např. $\Sigma = \{0, 1\}$.)
Slovem nad abecedou Σ rozumíme libovolnou konečnou posloupnost prvků množiny Σ, například „a,b,b,a,b“, přičemž tuto posloupnost píšeme bez čárek jako „abbab“.
Prázdné slovo „ “ je také slovem a značí se ε.

Značení: Znaky abecedy obvykle značíme malými písmeny ze začátku abecedy $(a, b, c, \ldots)$ s případnými indexy apod. Slova obvykle pojmenováváme malými písmeny z konce abecedy $(u, v, w, \ldots)$. Výrazem Σ^* značíme *množinu všech slov* na abecedou Σ a Σ^+ *množinu všech neprázdných slov* v abecedě Σ. (Je tedy $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.)

Poznámka: Množina všech slov nad konečnou abecedou je vždy spočetná – stačí všechna slova uspořádat nejprve podle délky a pak podle abecedy mezi slovy stejné délky. Tím budou všechna slova napsána do jedné posloupnosti, ve které je lze po řadě očíslovat přirozenými čísly.

Např. pro $\Sigma = \{0, 1\}$ (s abecedním uspořádáním $0 < 1$) lze prvky Σ^* generovat v pořadí $\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$. Příslušné uspořádání budeme označovat $<_L$ (např. $11 <_L 001$).

Značení: Délku slova w, tj. počet písmen ve w, značíme $|w|$. Symbolem $|w|_a$ označujeme počet výskytů symbolu a ve slově w.

Značení: Pro zjednodušení zápisu slov někdy používáme „exponenty“ u znaků, kde „ x^k “ znamená k opakování znaku x za sebou. Například zápis „a^3bc^4a“ je zkratkou pro slovo *aaabcccca*.

Přirozenou operací se slovy je jejich spojení za sebou do jednoho výsledného slova.

Definice 2.2
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; tedy $u^0 = \varepsilon$, $u^1 = u$, $u^2 = uu$, $u^3 = uuu$ atd.

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.

Někdy potřebujeme mluvit jen o určitých částech slova. Úsek znaků, kterým nějaké slovo začíná, budeme nazývat předponou, neboli odborně prefixem. Obdobně se úsek znaků, kterým slovo končí, budeme nazývat příponou, odborně sufixem. Jakoukoliv část slova budeme nazývat podslovem.

Definice 2.3

- Slovo t je *prefixem* slova z, pokud lze psát $z = tu$ pro nějaké slovo u.
- Slovo u je *sufixem* slova z, pokud lze psát $z = tu$ pro nějaké slovo t.
- Slovo u je *podslovem* slova z, pokud lze psát $z = tuv$ pro nějaké slova t a v.

Příklad: Vezměme si například slovo „abcadbcdc“. Pak slovo „abc“ je jedním z jeho prefixů, kdežto „bc“ prefixem není. Naopak „bcdc“ je jedním z jeho sufixů. Všechna tři uvedená slova jsou jeho podslovy. Prázdné slovo ε je prefixem, sufixem a samozřejmě i podslovem každého slova.

Definice 2.4
Formální jazyk, stručně *jazyk* nad abecedou Σ je libovolná podmnožina slov v abecedě Σ.

Značení: Jazyky obvykle označujeme L (s indexy); pro jazyk L nad abecedou Σ je tedy $L \subseteq \Sigma^*$.

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

Cvičení 2.1:

a) Vypište všechna slova v abecedě $\{a, b\}$, která mají délku 3.

b) 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).

c) Vypište všechna slova délky 2, které jsou podslovy slova 00010 (v abecedě $\{0, 1\}$).

d) Vypište všechny prefixy slova 0010. (Je jich pět!)

e) Vypište všechny sufixy slova 0010. (Je jich pět!)

Poznámka: Na rozdíl od přirozeného jazyka (češtiny) budou naše jazyky obvykle obsahovat nekonečně mnoho slov, budou to tedy nekonečné jazyky. Jako příklad si lze představit množinu L všech možných vět, které lze gramaticky správně v češtině vytvořit (taková věta je pak slovem formálního jazyka L).

Poznámka: Byť v praktických případech má abeceda např. desítky prvků, v našich příkladech bude abeceda často (jen) dvouprvková (většinou $\{a, b\}$ či $\{0, 1\}$). Uvědomme si, že to není zásadní omezení, jelikož písmena víceprvkové abecedy lze přirozeně zakódovat řetězci dvouprvkové abecedy.

Kontrolní otázka: Jak dlouhé řetězce byste použili např. při kódování 256-ti prvkové abecedy?

Příklad: Příklady formálních jazyků nad abecedou $\{0, 1\}$ jsou:

- $L_1 = \{\varepsilon, 01, 0011, 1111, 000111\}$
- L_2 je množina všech (konečných) posloupností v abecedě $\{0, 1\}$ obsahujících stejně 0 jako 1, tedy $L_2 = \{w \in \{0, 1\}^* \mid |w|_0 = |w|_1\}$
- $L_3 = \{w \in \{0, 1\}^* \mid$ číslo s binárním zápisem w je dělitelné $3\}$

Jazyk L_1 je zde konečný, kdežto zbylé dva jsou nekonečné. Slovo 101100 patří do jazyka L_2, ale 10100 do L_2 nepatří, neboť obsahuje více nul než jedniček. Slovo 110 binárně vyjadřuje číslo 6, a proto patří do jazyka L_3, kdežto 1000 vyjadřující 8 do L_3 nepatří.

2.2 Některé operace s jazyky

Někdy může být výhodné definovat složitější jazyk prostřednictvím dvou jednodušších a nějaké operace, která je spojí. Protože jsou jazyky definovány jako množiny, můžeme používat běžné množinové operace (definované v Sekci 1.1). Tedy např. průnik dvou jazyků nám dá zase jazyk. Dále můžeme definovat nové operace speciálně pro práci s jazyky. Konkrétně tedy při práci s formálními jazyky používáme nejčastěji následující matematické operace:

- Běžné množinové operace sjednocení $K \cup L$, průnik $K \cap L$, rozdíl $K - L$ nebo doplněk $\overline{L}$ (rozumí se pro příslušnou abecedu Σ, tj. $\overline{L} = \Sigma^* - L$).
- *Zřetězení* dvou jazyků $K \cdot L = \{uv \mid u \in K, v \in L\}$, tj. jazyk všech slov, které začínají nějakým slovem z K a pokračují libovolným slovem z L.
- *Iterace* jazyka L, značená L^*, která je definovaná rekurentně pomocí zřetězení

  $L^0 = \{\varepsilon\}$, $L^1 = L$, $L^{n+1} = L^n \cdot L$, celkem $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$,

  tj. jazyk všech slov, která lze rozdělit na několik částí, přičemž každá z nich patří do L.

Poznámka: Všimněme si, že operace zřetězení dvou jazyků není komutativní, tj. obecně nelze zaměnit pořadí $K \cdot L \neq L \cdot K$.

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

Všimněte si, že značení pro iteraci L^* odpovídá značení množiny všech slov Σ^* nad abecedou Σ — abeceda samotná je množinou všech jednopísmenných slov, přitom jejich iterací vzniknou všechna konečná slova.

Definice iterace nám také říká, že prázdné slovo do ní patří vždy (vznikne „zřetězením nula slov z L“).

Dále si všimněme, že např. $\emptyset^* = \{\varepsilon\}$, $(L^*)^* = L^*$ apod.

Příklad: Uveďme si následující ukázky operací s jazyky nad abecedu $\{0, 1\}$:

a) Sjednocením jazyka L_0 všech slov obsahujících více 0 než 1 a jazyka L_1 všech slov obsahujících více 1 než 0 je jazyk všech slov majících počet 1 různý od počtu 0.

b) Co vznikne zřetězením $L_0 \cdot L_1$ jazyků z předchozí ukázky (a)? Patří sem všechna možná slova?

Všechna slova do tohoto jazyka nepatří, například snadno najdeme $10 \notin L_0 \cdot L_1$. Obecný popis celého zřetězení však není úplně jednoduchý. Dle definice do tohoto zřetězení patří všechna slova, která lze rozdělit na počáteční úsek mající více 0 než 1 a zbytek mající naopak více 1 než 0.

c) Je pravda, že $L_0 \cdot L_1 = L_1 \cdot L_0$ v předchozí ukázce?

Není, například, jak už bylo uvedeno, $10 \notin L_0 \cdot L_1$, ale snadno $10 \in L_1 \cdot L_0$.

d) Co vznikne iterací jazyka $L_2 = \{00, 01, 10, 11\}$?

Takto vznikne jazyk L_2^* všech slov sudé délky, včetně prázdného slova. Zdůvodnění je snadné, slova v L_2^* musí mít sudou délku, protože vznikají postupným zřetězením úseků délky 2. Naopak každé slovo sudé délky rozdělíme na úseky délky 2 a každý úsek bude mít zřejmě jeden z tvarů v L_2.

Další zajímavou operací definovanou pro jazyky je zrcadlový obraz.

Definice 2.5
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 $L^R = \{u \mid \exists v \in L : u = v^R\}$.

Příklad: Zrcadlovým obrazem jazyka $L_1 = \{\varepsilon, a, abb, baaba\}$ je jazyk $L_1^R = \{\varepsilon, a, bba, abaab\}$.

Zrcadlovým obrazem jazyka $L_2 = \{w \mid |w|_a \bmod 2 = 0\}$ je jazyk L_2, neboli $L_2^R = L_2$.

Otázky:

Otázka 2.2: Můžeme množinu všech přirozených čísel považovat za abecedu v našem smyslu?

Otázka 2.3: Můžeme množinu všech přirozených čísel (alespoň v nějaké reprezentaci) považovat za formální jazyk v našem smyslu?

Otázka 2.4: Lze operacemi sjednocení nebo zřetězení z konečných jazyků vytvořit nekonečný jazyk?

Otázka 2.5: Jaký je rozdíl mezi prázdným jazykem $\emptyset$ a prázdným slovem ε?

Otázka 2.6*: Kdy iterací jazyka L vzniká jen konečný jazyk?

Otázka 2.7*: Rozmyslete si, proč dvojí iterací jazyka nevznikají už nová slova, neboli proč $L^* = (L^*)^*$.

Cvičení 2.8: Která slova jsou zároveň prefixem i sufixem slova 101110110? (Najdete všechna tři taková?)

Cvičení 2.9: Vypište slova ve zřetězení jazyků $\{110, 0111\} \cdot \{01, 000\}$.

Cvičení 2.10: 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 $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, $\overline{L}$.

Cvičení 2.11: Najděte dva různé jazyky, které komutují v operaci zřetězení, tj. $L_1 \cdot L_2 = L_2 \cdot L_1$.

Cvičení 2.12*: Co vzniká iterací jazyka $\{00, 01, 1\}$? Patří tam všechna slova nad $\{0, 1\}$?

Cvičení 2.13**: Dokažte či vyvraťte obecnou platnost následujících vztahů:

- $L_1 \cdot L_2 = L_2 \cdot L_1$
- $L_1(L_2 \cup L_3) = L_1L_2 \cup L_1L_3$
- $(L_1 \cup L_2)^* = L_1^*(L_2 \cdot L_1^*)^*$
- $(L_1 \cap L_2)^* = L_1^* \cap L_2^*$

2.3 Cvičení

Cvičení 2.14:Uvažujme jazyky nad abecedou $\{0, 1\}$. Nechť L_1 je jazykem všech těch slov obsahujících nejvýše pět znaků 1 a L_2 je jazykem všech těch slov, která obsahují stejně 0 jako 1. Kolik je slov v průniku $L_1 \cap L_2$?

Cvičení 2.15:Uvažujme jazyky nad abecedou $\{a, b\}$. Vypište všechna slova ve zřetězení jazyků $L_1 = \{\varepsilon, abb, bba\}$ a $L_2 = \{a, b, abba\}$.

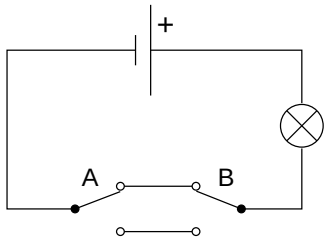
Cvičení 2.16:Uvažujme jazyky nad abecedou $\{c, d\}$. Nechť L_0 je jazyk všech těch slov, která obsahují různé počty výskytů symbolu c a výskytů symbolu d. Popište slovně zřetězení$L_0 \cdot L_0$.

Cvičení 2.17:Zjistěte, které z následujících dvou vztahů jsou platné pro všechny jazyky L_1, L_2:

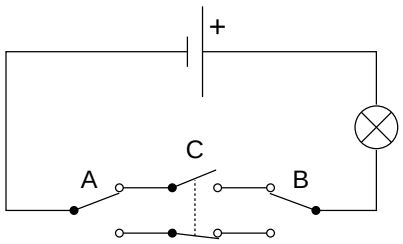
a) $(L_1 \cup L_2) \cdot L_3 = (L_1 \cdot L_3) \cup (L_2 \cdot L_3)$?

b) $(L_1 \cap L_2)^* = L_1^* \cap L_2^*$?

Cvičení 2.18:Představme si následující elektrický obvod s dvěma přepínači A a B. (Přepínače jsou provedeny jako aretační tlačítka, takže jejich polohu zvnějšku nevidíme, ale každý stisk je přehodí do druhé polohy.) Na počátku žárovka svítí. Pokusme se schematicky popsat, jaké posloupnosti stisků A, B vedou k opětovnému rozsvícení žárovky.



Cvičení 2.19:Obdobně jako v předchozím příkladě si vezměme následující obvod s přepínači A, B, C a jednou žárovkou. (Přepínač C má dva společně ovládané kontakty, z nichž je spojený vždy právě jeden.) Na počátku žárovka nesvítí. Jaké posloupnosti stisků A, B, C vedou k rozsvícení žárovky?



Cvičení 2.20:Uvažujme jazyky nad abecedou $\{0, 1\}$. Vypište všechna slova ve zřetězení

$$\{0, 001, 111\} \cdot \{\varepsilon, 01, 0101\}$$

Cvičení 2.21:Uvažujme jazyky nad abecedou $\{0, 1\}$. Popište (slovně) jazyk vzniklý iterací $\{00, 111\}^*$.

Cvičení 2.22:Uvažujme jazyky nad abecedou $\{0, 1\}$. Nechť L_1 je jazykem všech těch slov obsahujících nejvýše jeden znak 1 a L_2 je jazykem všech těch slov, která se čtou stejně zepředu jako zezadu (tzv. palindromů). Která všechna slova jsou v průniku $L_1 \cap L_2$?

Poznámka: Pozor, průnik obou jazyků je nekonečný.

Cvičení 2.23:Proč obecně neplatí $(L_1 \cap L_2) \cdot L_3 = (L_1 \cdot L_3) \cap (L_2 \cdot L_3)$?

Cvičení 2.24:Navrhněte obvod s třemi přepínači a žárovkou mající více než jeden vnitřní svítící i nesvítící stav.

Cvičení 2.25*:Uvažujme jazyky nad abecedou $\{a, b\}$. Nechť L_a je jazyk všech těch slov, která obsahují více a než b, a L_b je jazyk všech těch slov, která obsahují více b než a. Jaký jazyk vznikne zřetězením $L_a \cdot L_b$?

Cvičení 2.26*:Jazyk L_1 obsahuje 6 slov a jazyk L_2 obsahuje 7 slov. Kolik nejméně slov musí obsahovat zřetězení $L_1 \cdot L_2$?

Cvičení 2.27:Proč obvod s třemi (dvoupolohovými) přepínači a žárovkou nemůže mít více než 8 vnitřních stavů?

Kapitola 3

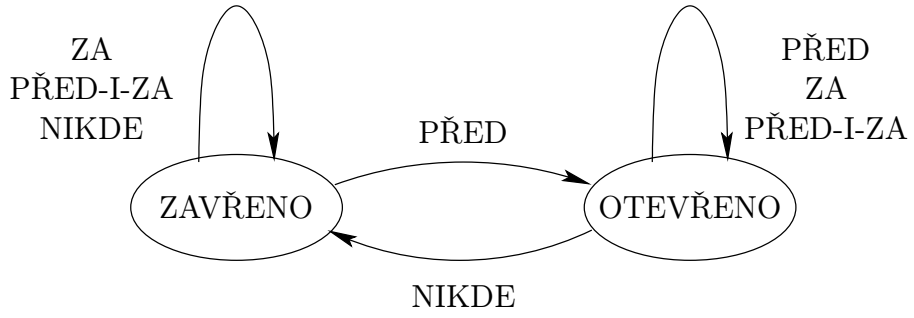
Konečné automaty

Cíle kapitoly:

- Seznámení se s pojmem konečného automatu, speciálně jako s prostředkem rozpoznávání posloupností symbolů splňujících určité (jednoduché) podmínky.
- Zvládnutí návrhu elementárních automatů a jednoduchých algoritmů zjišťujících vlastnosti automatů.
- Pochopení možností návrhu složitějších automatů modulárním postupem.

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 a pro složitější automaty může být i přehlednější.

Příklad: Diagram na Obrázku 3.1 je popisem jednoduchého automatu řídícího vstupní dveře do supermarketu. Dveře nejsou posuvné, ale otvírají se dovnitř. Proto se nemohou otvírat ani zavírat, když za nimi někdo stojí.



Obrázek 3.1:

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

Obrázek 3.2:

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 Obrázku 3.2.

Cvičení 3.1: 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 3.1
Konečný automat (zkráceně KA) je dán uspořádanou pěticí $A = (Q, \Sigma, \delta, q_0, F)$, 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.

Význam množin Q a Σ v definici je jasný. Všimněme si, že dále 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. Počáteční stav q_0 musí být určen jednoznačně, ale automat může mít více přijímajících stavů, třeba i všechny.

Přechodová funkce δ má dva argumenty $\delta(q, x)$, které mají následující význam: Pokud se automat právě nachází ve stavu q a čte ze vstupu znak x, musí přejít do stavu $\delta(q, x)$ (ten může být jiný i stejný jako q). Důležité je, že pro *každý stav a každý vstupní podnět* musí být definováno, kam automat přejde.

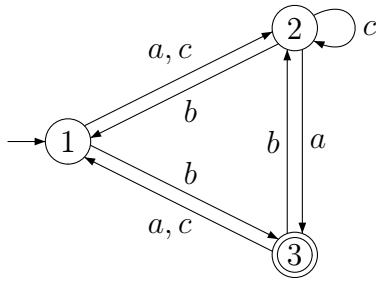
Značení: *Grafem automatu* (neboli stavovým diagramem) rozumíme orientovaný ohodnocený graf, ve kterém

- vrcholy jsou stavy automatu, tj. množina Q,
- počáteční stav (q_0) je vyznačen příchozí šipkou a koncové stavy (F) dvojitým kroužkem,
- hrana z u do v je označená výčtem všech písmen abecedy, které stav u převádějí na v, tj. $\{x \in \Sigma \mid \delta(u, x) = v\}$.

Hrany nekreslíme pro dvojice vrcholů mezi kterými není přechod žádným písmenem abecedy. Pokud se z vrcholu u přechází zpět do u, kreslí se smyčka.

Poznámka: Někdy se v literatuře můžete setkat i se značením koncových stavů šipkou vedoucí z nich.

Komentář: Zde vidíme ukázku grafu jednoduchého třístavového automatu:



V ukázce je $Q = \{1, 2, 3\}$ a $\Sigma = \{a, b, c\}$. Přechodová funkce například říká, že $\delta(1, a) = \delta(1, c) = 2$ nebo $\delta(2, c) = 2$, $\delta(2, b) = 1$, atd. Počáteční stav je 1 a přijímající stav je také jediný 3. Pokud na vstupu bude slovo „*accbb*“, stane se následující: Automat začne v 1, přejde čtením a do 2, pak čtením c dvakrát zůstává v 2, čtením b se vrátí do stavu 1 a dalším b přejde do stavu 3, kterým celé slovo přijme.

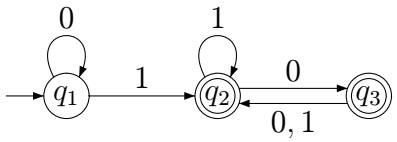
Značení: Přechodovou tabulkou automatu rozumíme tabulku s řádky označenými stavy automatu a sloupci označenými symboly abecedy, ve které políčko na řádku q a sloupci a udává stav $\delta(q, a)$. Počáteční stav je značený $\rightarrow$ a přijímající $\leftarrow$ nebo kroužkem kolem čísla stavu.

Komentář: Například výše zakreslený automat má přechodovou tabulku:

	a	b	c
$\rightarrow 1$	2	3	2
2	3	1	2
$\leftarrow 3$	1	2	1

Postup výpočtu konečného automatu si můžeme obecně popsat následovně:

- Automat začne v počátečním stavu q_0 na začátku slova s.
- Přečte aktuální písmeno x slova s a *přejde* do stavu určeného $\delta(q, x)$, kde q je současný stav automatu. Zároveň se jeho vstup přesune na následující písmeno slova s.
- Předchozí bod se opakuje, dokud nejsou přečtena všechna písmena v s.
- Pokud je poslední stav automatu přijímající ($q \in F$), pak je slovo s *přijato*, v opačném případě je s *odmítnuto*.

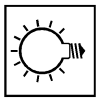


přijímá:	nepřijímá:
1101,010101,...	0110,0010,...

Obrázek 3.3:

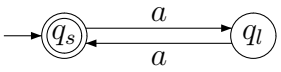
Říkáme také, že jsme dosáhli / nedosáhli přijímající stav.

Komentář: Výpočet automatu A na slově w si také můžeme představit jako sled v grafu A, který začíná v počátečním stavu q_0 a znaky jeho hran tvoří posloupnost písmen slova w. Tento sled se může libovolně cyklit a opakovat hrany i stavy, jen musí být konečný. Slovo w je přijato, pokud jeho sled výpočtu končí v množině F.



Řešený příklad 3.1: Navrhněme automat nad jednoznakovou abecedou $\{a\}$ přijímající právě ta slova mající sudou délku.

Řešení: To je velmi jednoduché – automat bude obsahovat jeden cyklus délky 2, který bude počítat paritu délky vstupního slova:



Neboli, vždy, když je automat ve stavu q_l, poslední přečtená je lichá pozice slova, a ve stavu q_s je to sudá pozice.

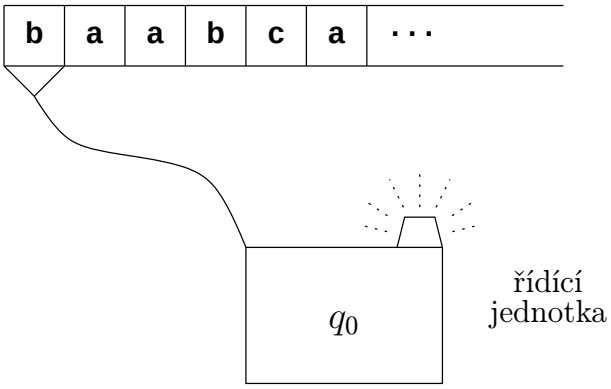
Komentář: Pro více (řešených) příkladů na jednoduché konečné automaty doporučujeme čtenáři se podívat do kapitoly 3.3.

Cvičení 3.2: Chápejme Obrázek 3.3 jako popis konečného automatu $A = (Q, \Sigma, \delta, q_0, F)$. Vypište *přímým výčtem* hodnoty všech členů pětice A. Pak porovnejte s Obrázkem 3.4. Na obrázku je počáteční stav q_1 rovnou zapsán jako čtvrtý člen pětice místo q_0; mohli bychom to samozřejmě také řešit zápisem $q_0 = q_1$.

$A = (Q, \Sigma, \delta, q_1, F)$, 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 3.4:



Obrázek 3.5:

Konečný automat si lze představovat různě. Pro naše účely je zřejmě nejvhodnější představa znázorněná na Obrázku 3.5. *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 formální definici přijímaných slov.

Definice 3.2
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.

Poznámka: Konfigurace je vlastně globální stav automatu zahrnující stav řídící jednotky a situaci na pásce.

Relace $\vdash_A$ popisuje, jak se tento globální stav změní přečtením jednoho symbolu a tedy provedením jednoho přechodu. Nepřečtená část slova se vždy o jeden znak zkrátí a podle přechodové funkce se může (ale nemusí pro $\delta(q, a) = q$) změnit stav.

Cvičení 3.3: Ověřte si, že uvedená definice jazyka rozpoznávaného KA skutečně zachycuje (formalizuje) předcházející neformální vysvětlení.

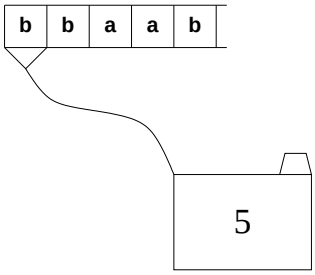
Cvičení 3.4: Pro automat A na Obrázku 3.6:

a) Simulujte krok po kroku výpočet na slově *bbaab* (zapište příslušnou posloupnost konfigurací; první, tedy $(5, bbaab)$, je zachycena na Obrázku 3.7).

b) Vypište všechna slova délky ≤ 3 v abecedě $\{a, b\}$ a zjistěte, která z nich automat A přijímá.

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

Obrázek 3.6:



Obrázek 3.7: Počáteční konfigurace automatu A

c) Zakreslete graf (stavový diagram) automatu A

d) Charakterizujte co nejjednodušeji vlastnosti slov, která automat přijímá.)

Někdy pro nás může být výhodná přechodová funkce rozšířená na celá slova. Definovat ji můžeme touto induktivní definicí:

Definice 3.3
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. $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$.

Značení: Dále bubudeme místo δ^* psát jen δ. Platí totiž $\delta^*(q,a) = \delta(q,a)$. (δ^* je tedy rozšířením δ na větší definiční obor; z kontextu bude vždy zřejmé, odkazujeme-li se na δ v užším nebo širším smyslu.)

3.1 Jazyk rozpoznávaný automatem

Jazyk rozpoznávaný (neboli přijímaný, akceptovaný) konečným automatem A je množinou všech těch slov, které automat přijímá, tj. těch slov, kterými automat A dosáhne některý z přijímajících stavů.

Definice 3.4
Jazykem rozpoznávaným (přijímaným) automatem A rozumíme jazyk $L(A) = \{w \mid$ slovo w je přijímáno $A\}$.

Definice 3.5
Jazyk $L \subseteq \Sigma^*$ je *regulární* právě když jej lze rozpoznat konečným automatem nad abecedou Σ, tj. existuje konečný automat A takový, že $L = L(A)$.

Poznámka: Nepleťme si zatím regulární jazyky s regulárními výrazy, které asi znáte u počítačů, třeba v příkazu `grep`. I když, brzy si už ukážeme, že regulární výrazy popisují právě regulární jazyky.

Základní poznatky o jazycích rozpoznávaných automaty jsou uvedeny zde.

Lemma 3.6
Pro konečný automat A s n stavy je jazyk $L(A)$ neprázdný právě tehdy, když existuje slovo $w \in L(A)$ délky menší než n (tj. $|w| < n$).

Důkaz: Jazyk je neprázdný právě když existuje orientovaný sled, tedy i cesta, v grafu automatu A z počátku do některého přijímajícího stavu. Nejkratší taková cesta má jistě méně než n hran. □

Lemma 3.7
Pro konečný automat A s n stavy je $L(A)$ nekonečný právě tehdy, když existuje $w \in L(A)$ splňující $n \leq |w| < 2n$.

Důkaz (náznak): Pokud existuje sled v grafu automatu A z počátečního stavu do některého přijímajícího stavu o délce aspoň n, pak je tento sled někde

„zacyklený“ (vrací se do stejného vrcholu) a tento cyklus můžeme libovolně krát zopakovat, tj. vygenerovat libovolné množství přijímaných slov.

Naopak v nekonečném jazyce existuje libovolně dlouhé přijímané slovo. Sled výpočtu takového slova (myslíme tím sled v grafu automatu A) může mít mnoho cyklů, ale alespoň jeden z těchto cyklů je délky menší než n, a proto lze postupným vypouštěním cyklů nakonec získat přijímající sled délky mezi n a $2n$. □

Definice 3.8
Dva konečné automaty A_1, A_2 přijímající shodné jazyky, tj. $L(A_1) = L(A_2)$, se také nazývají (jazykově) *ekvivalentní*.

3.1.1 Normovaný tvar automatu

Lze snadno nahlédnout, že pro jazyk $L(A)$ přijímaný automatem A můžeme navrhnout (nekonečně) mnoho dalších automatů, které jej také přijímají. Stačí přidávat nové stavy, do kterých se automat pro žádné slovo během výpočtu nedostane. Je ale zřejmé, že výhodnější budou pro nás ty automaty, které jsou co nejmenší a tudíž takové zbytečné, nedosažitelné stavy neobsahují.

Definice 3.9
Říkáme, že stav q automatu A je *dosažitelný* slovem w, pokud výpočet A se po přečtení (celého) slova w zastaví ve stavu q.

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

Tvrzení 3.10
Ke každému KA A lze zkonstruovat KA A', v němž každý stav je dosažitelný a $L(A') = L(A)$.

Z tvrzení 3.10 snadno nahlédneme, že

Věta 3.11
Existuje algoritmus, který pro zadaný KA A rozhodne, zda $L(A)$ je neprázdný.

Kontrolní otázka: Jak vypadá ten algoritmus?

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 $<_L$ (popsáno na straně 32) je zřejmé následující tvrzení.

Tvrzení 3.12
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.

Kontrolní otázka: Jak plyne spočetnost množiny racionálních čísel ze spočetnosti Σ^*?

I automat bez nedosažitelných stavů můžeme prezentovat mnoha různými způsoby. Proto nás také zajímá nějaká jeho jednoznačná – *normovaná prezentace*. 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 3.13
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 $u_i <_L u_j$.

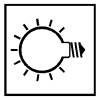
Komentář: Tedy automat A je v *normovaném tvaru*, jestliže jeho stavy jsou očíslované $1, 2, \ldots$ v abecedním pořadí nejmenších slov, kterými tyto stavy lze dosáhnout.

Poznámka: Uvedená definice je sice matematicky přesná, neposkytuje však žádný rozumný přímý postup pro nalezení normovaného tvaru. Když se však nad tímto problémem hlouběji zamyslíme, uvidíme, že je velmi podobný hledání nejkratší cesty v grafu a prosté prohledávání grafu do šířky jej dokáže vyřešit.

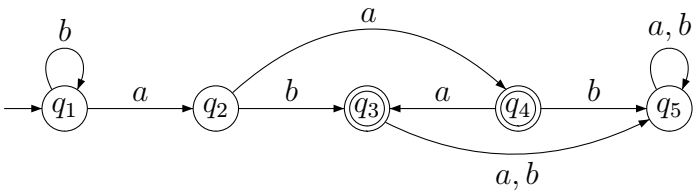
Metoda 3.14
Převod KA do normovaného tvaru (přečíslováním stavů) provede následující jednoduchý algoritmus.

- 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, 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. . . , dokud nezískáme všechny dosažitelné stavy.

Jedná se vlastně o *procházení grafu do šířky* při seřazení hran podle abecedy.

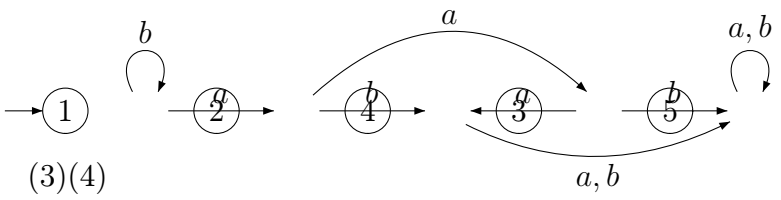


Řešený příklad 3.2: Stavy následujícího automatu seřaďte tak, jak mají být číslovány v normovaném tvaru.



Řešení: Podle Metody 3.14 očíslujeme stav q_1 číslem 1. Znakem a z q_1 se dostaneme do nového stavu q_2, kterému přiřadíme číslo 2. Znakem b z q_1 zůstaneme v již očíslovaném stavu q_1. Z druhého stavu q_2 přejdeme znakem a do q_4, kterému dáme číslo 3, a znakem b do q_3, kterému dáme číslo 4. Ze třetího stavu q_4 se přes a dostaneme do již očíslovaného q_3 a přes b do nového q_5, který dostane číslo 5.

Výsledný normovaný tvar tak vyjde



Otázky:

Otázka 3.5: Může být počáteční stav automatu zároveň přijímajícím? A co by to znamenalo pro přijímaná slova?

Otázka 3.6: Může se stát, že konečný automat nepřijímá žádné slovo?

Otázka 3.7: Je normovaný tvar automatu určený jednoznačně?

Cvičení 3.8: Převeďte do normovaného tvaru automat z Obrázku 3.6.

Cvičení 3.9: 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 zkonstruujte automat rozpoznávající jazyk $L_1 \cap L_2$; zkuste přitom vhodně využít automaty zkonstruované pro jazyky L_1, L_2.

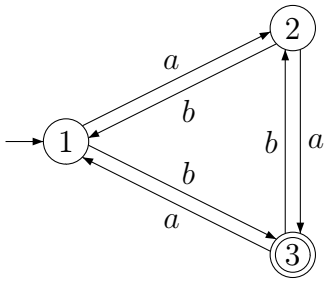
Cvičení 3.10: Navrhněte konečný automat přijímající všechna ta slova nad abecedou $\{a,b\}$, která obsahují lichý počet výskytů a.

Cvičení 3.11: Navrhněte konečný automat přijímající všechna ta slova nad abecedou $\{a\}$, jejichž délka dává zbytek 2 po dělení 3.

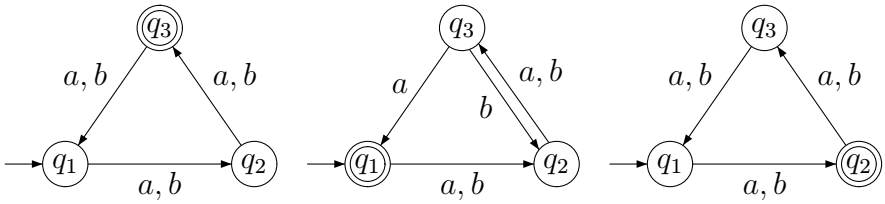
Cvičení 3.12: Jaká všechna slova přijímá automat na Obrázku 3.8?

Cvičení 3.13: Jaká všechna slova přijímá automat z Řešeného příkladu 3.2?

Cvičení 3.14: Které z těchto tří automatů nad abecedou $\{a,b\}$ přijímají nějaké slovo délky přesně 100?



Obrázek 3.8:



Cvičení 3.15: Nakreslete konečný automat přijímající právě všechna slova nad $\{a, b\}$, ve kterých je třetí znak stejný jako první.

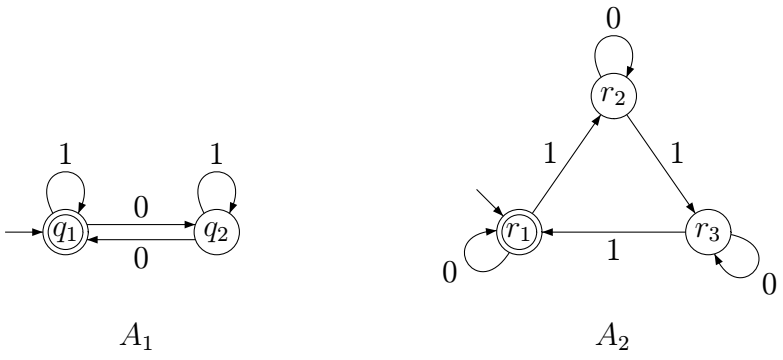
Cvičení 3.16: Nakreslete konečný automat přijímající právě všechna slova nad $\{a, b, c\}$, ve kterých se první znak ještě aspoň jednou zopakuje.

Cvičení 3.17: Mějme konečný automat A. Jak (jednoduše) sestrojíte automat A' přijímající právě všechna slova, která A nepřijímá? (Tzn. $L(A) = \overline{L(A')}$)

Cvičení 3.18: Na vybraných zkonstruovaných automatech si procvičte převod do normovaného tvaru.

3.2 Návrh složitějších konečných automatů

Návrh konečného automatu, který bude rozpoznávat zadaný jazyk, je tvůrčí činnost, vlastně jakési jednoduché programování, a proto nelze podat „me-



Obrázek 3.9:

chanický“ 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.

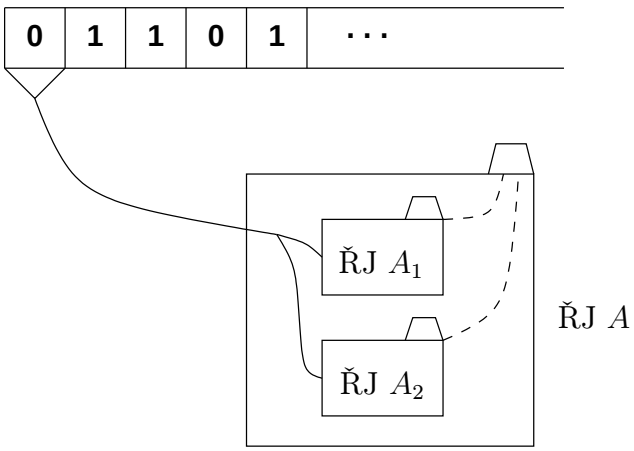
Mnoho věcí se přitom zmechanizovat (zalgoritmizovat) dá. Např. existují algoritmy, které z automatů pro „jednodušší“ jazyky vytvářejí automaty pro jazyky vzniklé operacemi nad oněmi (jednoduššími) jazyky.

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 $L_1 = \{w \in \{0,1\}^* \mid |w|_0 \text{ je dělitelné } 2\}$ a $L_2 = \{w \in \{0,1\}^* \mid |w|_1 \text{ je dělitelné } 3\}$. Na Obrázku 3.9 jsou znázorněny automaty rozpoznávající jazyky L_1 a L_2.

Poznámka: Připomeňme se, že označením $|w|_a$ značíme počet výskytů symbolu a ve slově w.

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 Obrázek 3.10. Rozmyslete si, co jsou stavy automatu A, co je to počáteční a koncový stav, a jak vypadá přechodová funkce.

Pak se podívejte na Obrázek 3.11, 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



Obrázek 3.10:

následující věty.

Věta 3.15
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 $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$.

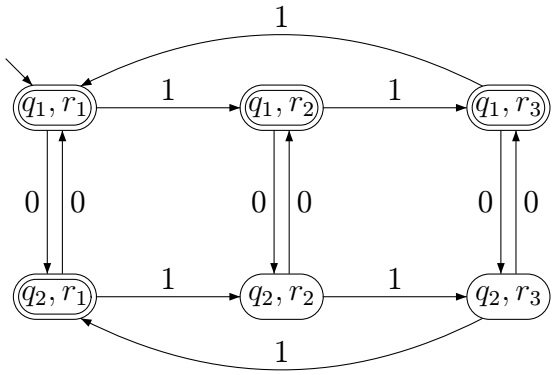
Definujme automat $A = (Q, \Sigma, \delta, q_0, F)$ tž.

- $Q = Q_1 \times Q_2$,
- $\delta(\,(q_1, q_2), a\,) = (\,\delta_1(q_1, a), \delta_2(q_2, a)\,)$ 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 $\delta(\,(q_1, q_2), w\,) = (\,\delta_1(q_1, w), \delta_2(q_2, w)\,)$.

Jinými slovy, každým vstupním slovem w automat A přejde do stavu (q_1, q_2), kde q_1 je stav automatu A_1 dosažený slovem w a obdobně q_2 je příslušný stav automatu A_2.

Z toho snadno plyne, že $L(A) = L_1 \cup L_2$. □



Obrázek 3.11:

Cvičení 3.19: Na základě (důkazu) věty 3.15 pro sjednocení teď ukažte analogickou větu pro průnik:

Věta 3.16
Jestliže jazyky $L_1, L_2 \subseteq \Sigma^*$ jsou regulární, pak také jazyk $L_1 \cap L_2$ je regulární.

(Nápověda: $F = (F_1 \times F_2)$)

Komentář: Konstrukci uvedenou ve Větě 3.15 si můžeme snadno vizuálně představit – automat A vypadá jako „mřížka“, jejíž sloupce představují automat A_1 a řádky představují automat A_2. Přechody se přitom dějí jak po sloupcích, tak po řádcích zároveň. Přijímající stavy jsou ty, které jsou přijímající aspoň v jednom z automatů A_1 a A_2. Promyslete si to podle Obrázku 3.11.

Poznámka: Je velmi vhodné si uvědomit *konstruktivnost* našich tvrzení. Např. věta 3.15 (věta 3.16) 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.

Otázky:

Otázka 3.20: Lze způsobem obdobným jako ve Větě 3.15 rozpoznávat rozdíl jazyků $L_1 - L_2$?

Otázka 3.21*: Dokážete si představit jazyk slov, který není přijímaný žádným konečným automatem?

Cvičení 3.22: Automat pro průnik požadovaný ve cvičení 3.9 sestrojte podle obecného návodu z (důkazu) předchozí věty. (Porovnejte se svou předchozí konstrukcí.)

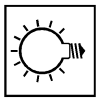
Cvičení 3.23: Lze konečným automatem rozpoznávat jazyk všech slov nad abecedou $\{a, b\}$, ve kterých je součin počtů výskytů znaků a a b sudý?

Cvičení 3.24: Lze konečným automatem rozpoznávat jazyk všech slov nad abecedou $\{a, b\}$, ve kterých je součet počtů výskytů znaků a a b sudý?

Cvičení 3.25: Lze konečným automatem rozpoznávat jazyk všech slov nad abecedou $\{a, b\}$, ve kterých je součet počtů výskytů znaků a a b větší než 100?

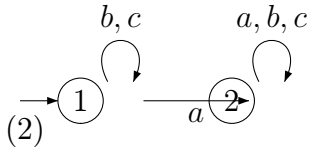
Cvičení 3.26: Navrhněte automat rozpoznávající všechna ta slova nad $\{a, b\}$, která začínají znakem a a končí znakem b.

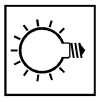
3.3 Cvičení



Řešený příklad 3.3: Existuje konečný automat se dvěma stavy rozpoznávající jazyk všech těch neprázdných slov nad abecedou $\{a, b, c\}$, která obsahují alespoň jeden znak a? Pokud ano, příslušný automat nakreslete.

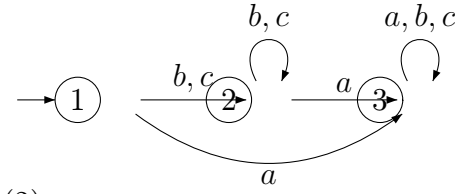
Řešení: Existuje, stačí se po prvním přečtení znaku a přesunout do přijímajícího stavu, ve kterém už zůstaneme.



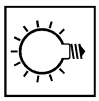


Řešený příklad 3.4: Existuje konečný automat se třemi stavy rozpoznávající jazyk všech těch neprázdných slov nad abecedou $\{a, b, c\}$, která neobsahují žádný znak a? Pokud ano, příslušný automat zde nakreslete. (Nezapomeňte, že přijímaná slova mají být neprázdná.)

Řešení: Na první pohled by se mohlo zdát, že stačí vzít automat z předchozího příkladu a přehodit přijímající stav. To však není tak jednoduché, neboť takový automat by přijímal i prázdné slovo. Náš automat má vypadat takto:



(2)



Řešený příklad 3.5: Jak poznáme, že dva konečné automaty A_1 a A_2 přijímají shodné jazyky, tj. zda $L(A_1) = L(A_2)$?

Řešení: Asi není schůdnou cestou kontrolovat všechna slova přijímaná jedním z těchto automatů, může jich být nekonečně mnoho. Přesto již znáte dost, abychom mohli popsat jednoduchý konečný postup pro rozhodnutí dané otázky. Nejprve ověříme, zda $L(A_1) \subseteq L(A_2)$:

Podle Úlohy 3.17 sestrojíme automat $\neg A_2$ přijímající opak (doplněk) jazyka $L(A_1)$. Poté sestrojíme podle Věty 3.15 automat B přijímající průnik jazyků $L(A_1) \cap L(\neg A_2)$. Elementární poznatky teorie množin nám říkají, že $L(A_1) \subseteq L(A_2)$ právě když $L(A_1) \cap L(\neg A_2) = \emptyset$. Takže nám stačí ověřit, že automat B nepřijímá žádné slovo, neboli že v grafu B nevede žádná orientovaná cesta z počátečního do přijímacího stavu. (Pokud by naopak B přijímal nějaké slovo w, pak by w rozlišovalo naše dva automaty.)

Symetricky si pak ověříme, zda $L(A_2) \subseteq L(A_1)$. Pokud to opět vyjde, celkově dojdeme k závěru, že $L(A_1) = L(A_2)$.

Cvičení 3.27:Je automat sestrojený v příkladě 4.3 nejmenší možný pro svůj jazyk?

Cvičení 3.28:Navrhněte konečný automat přijímající právě ta slova nad abecedou $\{a, b, c, d\}$, která nezačínají a, druhý znak nemají b, třetí znak nemají c a čtvrtý znak nemají d. (Včetně těch s délkou < 4.)

Cvičení 3.29:Navrhněte konečný automat přijímající právě ta slova nad abecedou $\{a, b, c, d\}$, která nezačínají a nebo druhý znak nemají b nebo třetí znak nemají c nebo čtvrtý znak nemají d.

Cvičení 3.30:Sestrojte konečný automat přijímající všechna ta slova délky aspoň 4 nad abecedou $\{a, b\}$:

a) ve kterých jsou druhý, třetí a čtvrtý znak stejné,

b) ve kterých jsou třetí a poslední znak stejné.

Cvičení 3.31:Sestrojte konečný automat přijímající všechna ta slova délky aspoň 2 nad abecedou $\{a, b\}$, ve kterých nejsou poslední dva znaky stejné.

Kapitola 4

Nedeterministické konečné automaty

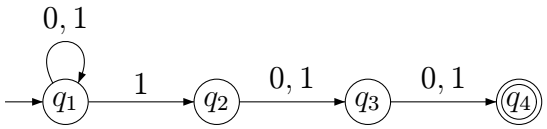
Cíle kapitoly:

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

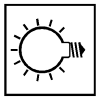
Uvažujme nyní obdobu věty 3.15 pro zřetězení jazyků. Jistě nás napadne přístup: „necháme na první část slova běžet první automat, v přijímajícím stavu pak předáme řízení druhému automatu a ten dočte slovo do konce“. Problém je, že obecně nepostačí prostě předat řízení při *prvním* příchodu do přijímajícího stavu, ale je nutno nechat to jako *možnost při jakémkoli* příchodu do přijímajícího stavu.

Kontrolní otázka: Proč nestačí předat řízení při prvním 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.



Myšlenku nedeterminismu využijeme i v následujícím motivačním příkladě.



ŘEŠENÝ PŘÍKLAD 4.1: Sestrojme automat přijímající všechna slova nad $\{0, 1\}$, ve kterých je třetí znak od konce 1.

Řešení: Sestrojit takový automat podle Definice 3.1 asi nebude lehké. Jak máme poznat dopředu, který znak bude třetí od konce? Asi nejjednodušším (třebaže zavánějícím podvodem) řešením je ponechat rozhodnutí na „vyšší moc“, která vidí slovo dopředu. Proto navrhneme následující automat, který setrvává při čtení 0 i 1 ve stavu q_1, až dosáhne třetí znak od konce slova. Pokud je ten 1, automat může přejít do stavu q_2. Z něj již jenom přečte následující (dle předpokladu poslední) dva znaky a slovo přijme. Takovou myšlenku znázorňuje automat na Obrázku 4.1.

Stavy q_3 a q_4 jsou v automatu proto, abychom si ověřili, že za vybraným znakem 1 skutečně následují další dva znaky a konec. Co se stane ve stavu q_4, pokud ještě konec slova není dosažen? Žádný další přechod z q_4 není definován, a proto zde výpočet selže a takové slovo není přijato.

Komentář: Nyní zbývá najít matematickou definici, která by způsob automatového výpočtu naznačeného v Příkladě 4.1 přesně formalizovala. Pochopitelně zde nemůžeme mluvit o žádné „vyšší moci“, ale to lze nahradit požadavkem přijetí všech těch slov, pro která *existuje alespoň jeden přijímající výpočet*. Jinak řečeno, místo vyšší moci si lze velmi dobře představit vševědoucí pomocnou „nápovědu“, která nám pomáhá vybrat přechody vedoucí k přijetí (pokud to vůbec je možné).

Definice 4.1
Nedeterministický konečný automat, zkráceně NKA, je dán uspořádanou pěticí $A = (Q, \Sigma, \delta, I, F)$, 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*
- $F \subseteq Q$ je neprázdná množina *přijímajících (koncových) stavů*.

Komentář: Rozdíl proti běžnému automatu z Definice 3.1 je v této definici na dvou místech:

- V daném stavu máme při čtení vstupního znaku možnost přechodu do více různých stavů, nebo také nemusíme mít žádnou možnost přechodu. Dokonce můžeme přecházet po hranách značených symbolem ε bez čtení ze vstupního slova – tzv. *ε-přechody*.
- Je povolen více než jeden počáteční stav.

Přímočaře lze opět nadefinovat *graf* (též nazývaný stavový diagram) NKA. Příklad takového grafu jsme již viděli na Obrázku 4.1 v motivačním příkladu. 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.

Pro definici *výpočtu* NKA lze takřka doslova použít definici 3.2. Musíme udělat jen drobné změny v definici přechodové relace $\vdash$ mezi konfiguracemi a v definici přijímajícího výpočtu. Pro úplnost uvedeme celé znění této upravené definice.

Definice 4.2
Konfigurací nedeterministického konečného automatu $A = (Q, \Sigma, \delta, I, F)$ rozumíme dvojici (q, w), kde $q \in Q$ a $w \in \Sigma^*$ (q představuje aktuální stav a w slovo, které zbývá přečíst).

Na množině všech konfigurací automatu A definujeme relaci $\vdash_A$ takto: $(q, aw) \vdash_A (q', w)$ právě když $\delta(q, a) \ni q'$ (rozumí se $a \in \Sigma$, $w \in \Sigma^*$).

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 $i = 0, 1, \ldots, n-1$.

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 $q_0 \in I$ a nějaký $q \in F$.

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

	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 4.1:

Jazyk přijímaný NKA A je množina všech slov přijímaných A.

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.

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).

Cvičení 4.1: Zkonstruujte deterministický KA, který přijímá tentýž jazyk jako automat na Obrázku 4.1.

Cvičení 4.2: Sestrojte co nejjednodušší *nedeterministický* konečný automat, který přijímá právě ta slova v abecedě $\{0, 1\}$, jež začínají 110 nebo končí 001 nebo obsahují 1111.

4.1 Převod na deterministické automaty

Čtenář si nejspíše teď klade přirozenou otázku, o kolik „silnější“ je nedeterministický automat oproti deterministickému. Odpověď je docela překvapivá – o nic! Jak nyní dokážeme, každý NKA lze jednoduchým postupem převést na ekvivalentní deterministický automat.

Věta 4.3
Pro každý nedeterministický konečný automat A existuje ekvivalentní (deterministický) konečný automat A', tj. rozpoznávající stejný jazyk $L(A) = L(A')$.

Důkaz: Nechť $A = (Q, \Sigma, \delta, I, F)$. Sestrojíme KA $A' = (Q', \Sigma, \delta', q_0, F')$, kde

- $Q' = \mathcal{P}(Q)$ je množina všech podmnožin stavů Q a $q_0 = I \in Q'$,
- $F' \subset Q'$ obsahuje všechny podmnožiny původních stavů Q, které obsahují některý stav z F.
- Přechodová funkce $\delta' : Q' \times \Sigma \to Q'$ každé podmnožině původních stavů $P \in Q'$ a písmenu $x \in \Sigma$ přiřadí podmnožinu $R \in Q'$ těch stavů automatu A, do kterých se lze v A dostat z některého stavu v P přechodem po jedné hraně označené x.

Není těžké nyní zdůvodnit, že nový automat A' přijímá stejná slova w jako původní A – po každém kroku výpočtu deterministického A' aktuální stav q představuje podmnožinu těch stavů A, které lze dosáhnout různými (nedeterministickými) větvemi výpočtu A na w. (I stav prázdná množina $\emptyset$ má svůj význam, neboť výpočet nedeterministického A nemusí mít definovaný žádný přechod nad určitým znakem.)

□

Chování DKA A' 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

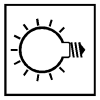
rozmístění knoflíků) je přijímající právě když alespoň jeden knoflík leží na uzlu odpovídajícím některému přijímajícímu stavu automatu A.

Všimněte si, že důkaz věty 4.3 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.

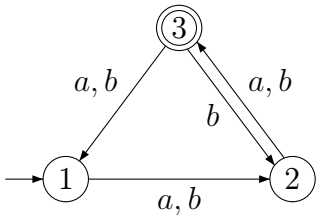
Na základě konstrukce v důkazu Věty 4.3, neformální představy knoflíkové hry a myšlenky omezit se jen na dosažitelné stavy si můžeme nyní popsat algoritmus převodu NKA na ekvivalentní DKA.

Metoda 4.4
Konstrukce determin. automatu z nedeterministického.

- Začneme se stavem reprezentujícím množinu I počátečních stavů nedeterministického automatu A.
- Dokud máme v sestrojovaném automatu A' stavy s nedefinovanými přechody, vybereme si jeden takový q a znak x. Pro všechny stavy reprezentované q najdeme všechny možnosti přechodu znakem x v A a shrneme je v nové množině stavů q' (ta již v našem automatu může být sestrojená).
- Když nový stav reprezentuje množinu, která má neprázdný průnik s F, označíme jej jako přijímající.

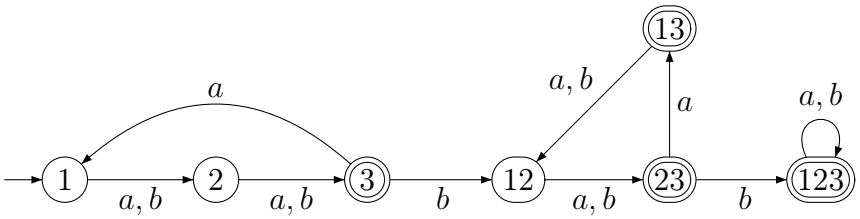


ŘEŠENÝ PŘÍKLAD 4.2: Sestrojte k tomuto nedeterministickému automatu ekvivalentní deterministický:



Řešení: Postupujeme přesně podle Metody 4.4. Tento automat má jediný nedeterministický přechod znakem b ze stavu 3, ale přesto budeme muset

použít všech 7 stavů odpovídajících neprázdným podmnožinám. Pro jednoduchost množiny stavů z $1, 2, 3$ vpisujeme do kroužků bez závorek a čárek, jako 123. Začneme v množině stavů $\{1\}$ a zleva doprava sestrojíme následující automat:



Cvičení 4.3: Převeďte na DKA automat z Obrázku 4.1.

Poznámka: Bohužel ne vždy deterministický automat sestrojovaný z nedeterministického n-stavového automatu má rozumnou velikost, jsou případy, kdy musí mít nejméně 2^n stavů, což už může být prakticky nezvládnutelné. 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.

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!)

Věta 4.5
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.

Věta 4.3 říká, že ke každému NKA lze sestrojit ekvivalentní DKA a tedy každý jazyk rozpoznávaný NKA je regulární. □

Poznámka: Závěrem se krátce zmíníme o tzv. „chybovém stavu“ automatu. V praktických příkladech konstrukce automatů se obvykle stává, že po přečtení některých nechtěných posloupností znaků automat přechází do stavu, který není přijímající a ve kterém už navždy zůstává. Takovému stavu se pak přirozeně říká *chybový*. Při konstrukci DKA převodem z NKA tento stav odpovídá prázdné množině stavů NKA (v pojmech knoflíkové hry to je situace, kdy nám zmizí všechny knoflíky).

Ve zjednodušených zobrazeních automatu se pak někdy takový chybový stav vynechává a přechody do něj nejsou definovány. To je zcela v souladu s definicí nedeterministického automatu, neboť nedefinované přechody znamenají nepřijetí slova. Takový automat však rozhodně *není deterministický* ve smyslu našich definic, přestože třeba všechny ostatní přechody deterministické jsou. Pokud máte za úkol sestrojit deterministický automat, pak ten musí obsahovat i případný chybový stav! A také musí i z něj být definovány přechody pro všechny symboly abecedy, i když vedou vždy zase zpět do tohoto chybového stavu. Pokud by zakreslení chybového stavu udělalo automat příliš nepřehledný. množstvím šipek, musíte aspoň jasně slovně poznamenat, že všechny zbylé šipky vedou do tohoto chybového stavu.

4.2 Zobecněný nedeterministický konečný automat

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 4.6
Zobecněný nedeterministický konečný automat (ZNKA) je dán uspořádanou pěticí $A = (Q, \Sigma, \delta, I, F)$, kde

- Q je konečná neprázdná množina *stavů*,
- Σ je konečná neprázdná *abeceda*,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ je *přechodová funkce*,
- $I \subseteq Q$ je množina *počátečních (iniciálních) stavů*
- $F \subseteq Q$ je neprázdná množina *přijímajících (koncových) stavů*.

Jediný rozdíl mezi ZNKA a NKA je tedy v definici přechodové funkce. Proto i definice konfigurace, výpočtu a přijímání slova automatem jsou stejné jako pro NKA v definici 4.2. Jediné, co musíme trochu pozměnit je definice přechodové relace mezi konfiguracemi.

Definice 4.7
Na množině všech konfigurací zobecněného nedeterministického konečného automatu A definujeme relaci $\vdash_A$ takto: $(q, aw) \vdash_A (q', w)$ právě když $\delta(q, a) \ni q'$ (rozumí se $a \in \Sigma \cup \{\varepsilon\}$, $w \in \Sigma^*$).

Cvičení 4.4: ZNKA z Obrázku 4.2 zadejte grafem.

Naformulujte pojem přijímání slova automatem ZNKA v řeči grafů automatů.

Charakterizujte jazyk, který je daným automatem přijímán.

Analogicky jako větu 4.3 lze ukázat:

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

Obrázek 4.2:

Věta 4.8
Pro každý zobecněný nedeterministický konečný automat A existuje ekvivalentní (deterministický) konečný automat A', tj. rozpoznávající stejný jazyk $L(A) = L(A')$.

Důkaz: Nechť $A = (Q, \Sigma, \delta, I, F)$. Sestrojíme KA $A' = (Q', \Sigma, \delta', q_0, F')$, kde

- $Q' = \mathcal{P}(Q)$ je množina všech podmnožin stavů Q
- $q_0 \in Q'$ je množina obsahující všechny stavy z I a všechny stavy z nich dosažitelné po ε-hranách.
- $F' \subset Q'$ obsahuje všechny podmnožiny původních stavů Q, které obsahují některý stav z F.
- Přechodová funkce $\delta' : Q' \times \Sigma \rightarrow Q'$ každé podmnožině původních stavů $P \in Q'$ a písmenu $x \in \Sigma$ přiřadí podmnožinu $R \in Q'$ těch stavů automatu A, do kterých se lze v A dostat z některého stavu v P přechodem po jedné hraně označené x a po libovolném počtu následujících hran označených ε.

Je snadné dokázat, že automaty A a A' přijímají stejný jazyk. □

Na základě konstrukce v důkazu věty 4.8 můžeme upravit metodu 4.4 tak, aby převáděla ZNKA na DKA.

Metoda 4.9
Konstrukce determin. automatu z nedeterministického.

- Začneme se stavem reprezentujícím množinu I počátečních stavů nedeterministického automatu A doplněnou o všechny stavy dosažitelné v A z počátečních stavů jen po hranách označených ε.

- Dokud máme v sestrojovaném automatu A' stavy s nedefinovanými přechody, vybereme si jeden takový q a znak x. Pro všechny stavy reprezentované q najdeme všechny možnosti přechodu znakem x v A a shrneme je v nové množině stavů q'. Do této množiny přidáme všechny stavy dosažitelné v A libovolně dlouhou sekvencí ε-přechodů ze stavů již se v q' nacházejících.
- Když nový stav reprezentuje množinu, která má neprázdný průnik s F, označíme jej jako přijímající.

Podobně jako 4.5 můžeme také ukázat:

Věta 4.10
ZNKA rozpoznávají právě regulární jazyky.

Otázky:

Otázka 4.5: Je výpočet ZNKA vždy konečný?

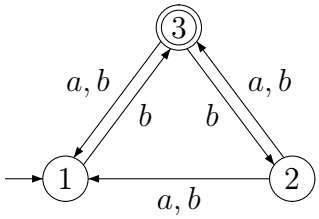
Otázka 4.6: Co se stane, pokud Metodu 4.4 aplikujeme na deterministický automat?

Otázka 4.7: Kdy při konstrukci podle Metody 4.4 vznikne stav reprezentovaný prázdnou množinou $\emptyset$?

Otázka 4.8: Může být stav $\emptyset$ (dle předchozí otázky) přijímajícím?

Cvičení 4.9: Kdy automat z následujícího Cvičení 4.10 přijímá slovo složené ze samých písmen a?

Cvičení 4.10: Sestrojte ekvivalentní deterministický automat k tomuto:



Cvičení 4.11: Kdy automat ze cvičení 4.10 přijímá slovo složené ze samých písmen b?

Cvičení 4.12*: Uměli byste slovně (a názorně) popsat jazyk přijímaný automatem ze Cvičení 4.10?

Cvičení 4.13: Navrhněte ZNKA přijímající jazyk všech těch slov nad $\{a, b\}$, které končí sufixem „abb“ nebo sufixem „aa“.

4.3 Uzávěrové vlastnosti třídy regulárních jazyků.

Množině všech možných jazyků určitého typu říkáme třída. Například do třídy regulárních jazyků patří všechny jazyky, pro které existuje nějaký konečný automat. Jazykové operace se provádějí s jazyky, čili prvky třídy (množiny jazyků). Můžeme tedy mluvit o uzavřenosti třídy vůči jazykové operaci (pojem uzavřenosti množiny vzhledem k operaci je definován v Sekci 1.3). Třída je například uzavřena na operaci zřetězení jazyků, jestliže zřetězením libovolných regulárních jazyků vznikne zase jazyk patřící do této třídy.

Je pro nás výhodné vědět, na které operace je uzavřena třída regulárních jazyků. Můžeme potom pomocí těchto operací a jednoduchých jazyků jednoznačně popsat složitější jazyky a máme jistotu, že takto vzniklé složitější jazyky jsou také regulární. Navíc si ukážeme postupy, jak pro takto vytvořené jazyky můžeme snadno zkonstruovat konečné automaty.

Zvláštní roli mezi operacemi, na které je uzavřena třída regulárních jazyků, hrají tzv. regulární operace. Proč jsou tak významné bude vysvětleno v Kapitole 5.

Definice 4.11
Regulárními operacemi s jazyky nazýváme operace

- *sjednocení* $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$,
- *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 $L^0 = \{\varepsilon\}$, $L^{n+1} = L \cdot L^n$.

Komentář: Příklady regulárních operací budiž třeba výrazy
$\{0, 11\}^* \cdot \{000, 11\}^*$,
$\{0, 11\}^* \cup \{010, 101\}^*$.
Co tyto zápisy znamenají?

V prvním případě nejprve iterujeme jazyk skládající se ze dvou slov `0` a `11` – vytvoříme jazyk všech slov skládajících se z nul nebo z dvojic jedniček. Poté iterujeme jazyk ze dvou slov `000` a `11` a vzniklé dva jazyky zřetězíme.

Ve druhém případě iterujeme obdobné dva jazyky, ale nakonec je sjednotíme (jako množiny).

Kontrolní otázka: Dokážete vlastními slovy popsat jazyky vzniklé použitím regulárních operací v předcházejícím komentáři?

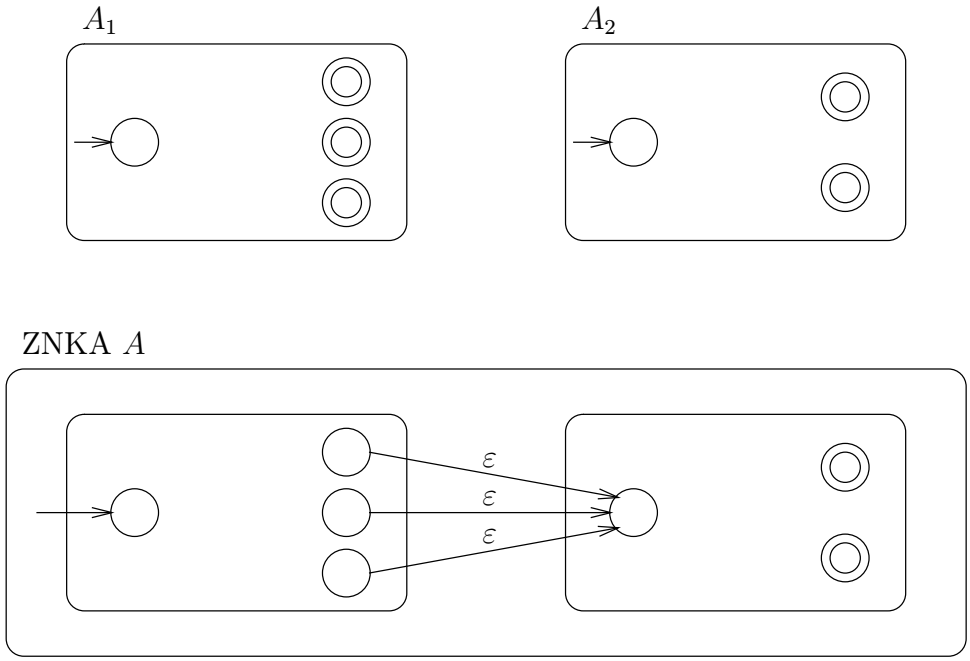
Uzavřenost na sjednocení jsme již dokázali ve Větě 3.15. Všimněme si, že pomocí NKA lze podat jiný, velmi přímočarý, důkaz této věty. (Nápověda: definujte vhodně pojem (disjunktního) *sjednocení* (tj. „vedle sebe položení“) *dvou NKA*.) Použijeme-li navíc ε-šipek (tedy ZNKA), docílíme navíc snadno toho, aby výsledný NKA měl jediný počáteční stav.

Uzavřenost třídy regulárních jazyků na zřetězení a iteraci je znázorněna na Obrázku 4.3 a 4.4, které by měly dostatečně naznačit myšlenku. Následuje formální důkaz.

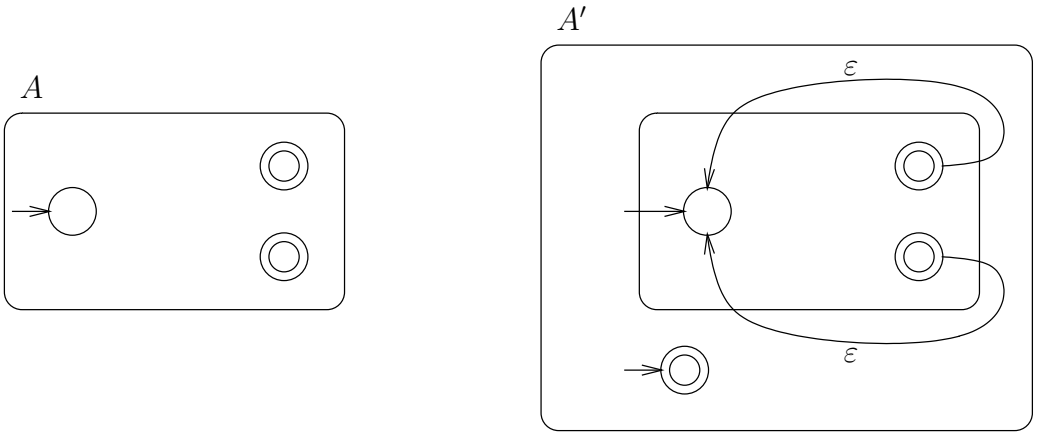
Věta 4.12
Jestliže jazyky $L_1, L_2 \subseteq \Sigma^*$ jsou regulární, pak také jazyk $L_1 \cdot L_2$ je regulární. Jestliže L je regulární, pak také L^* je regulární.

Vedle regulárních operací (sjednocení, zřetězení, iterace), je množina regulárních jazyků uzavřena i vůči dalším operacím. Snadno ukážeme také uzavřenost na doplněk a vyvodíme uzavřenost na (množinový) rozdíl:

Věta 4.13
Jestliže L je regulární, pak také jeho doplněk $\overline{L}$ je regulární. Jestliže L_1, L_2 jsou regulární jazyky, pak také rozdíl $L_1 - L_2$ je regulární.



Obrázek 4.3: $L(A) = L(A_1) \cdot L(A_2)$



Obrázek 4.4: $L(A') = L(A)^*$

Důkaz: Nechť $L = L(A)$, kde $A = (Q, \Sigma, \delta, q_0, F)$ je konečný automat. 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 $L_1 - L_2 = L_1 \cap \overline{L_2}$. □

Cvičení 4.14: 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).

Uzavřenost vůči průniku jsme již ukázali dříve (věta 3.16). Na rozdíl od sjednocení nám pro průnik elegance ε-šipek moc nepomůže. Ovšem uzavřenost třídy regulárních jazyků vůči průniku plyne také např. z uzavřenosti vůči sjednocení a doplňku (de Morganova pravidla).

Další operace, o které se zde zmíníme, je zrcadlový obraz.

Tvrzení 4.14
L je regulární právě když L^R je regulární.

Důkaz: Idea: (u NKA) zaměníme počáteční stavy s přijímajícími a obrátíme šipky.

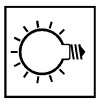
Formálně:

Nechť $L = L(A)$ pro NKA $A = (Q, \Sigma, \delta, I, F)$.

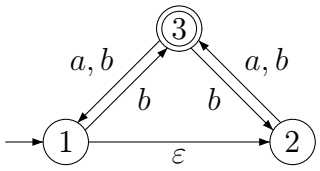
Definujme NKA $A' = (Q, \Sigma, \delta', F, I)$ tak, že pro vš. $q_1, q_2 \in Q$, $a \in \Sigma$: $q_2 \in \delta'(q_1, a) \Leftrightarrow q_1 \in \delta(q_2, a)$.

Pak lze snadno ukázat, že $L(A') = L^R$. □

4.4 Cvičení



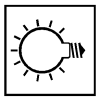
Řešený příklad 4.3: Sestrojte ekvivalentní deterministický automat k tomuto:



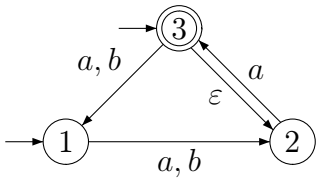
Řešení: Postupujeme přesně podle Metody met:nadet a automat tentokrát vyjde malý:

automaton60,10 (1)(10,5)1 (3)(30,5)3 (12)(50,5)12 (1)(3) [ELside=r](1,3)a, b [ELside=r,syo=-1,eyo=-1](3,12)a, b [ELside=r,syo=1,eyo=1](12,3)a, b automaton

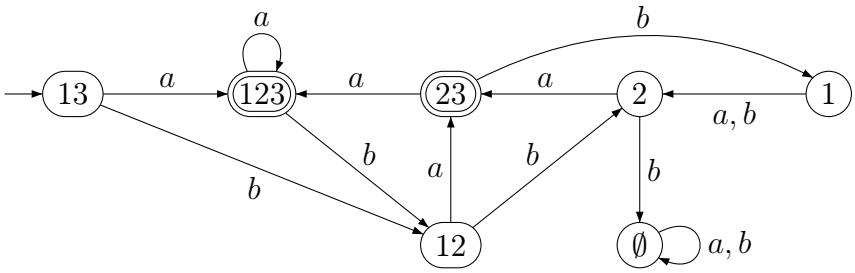
Všimněme si, že výsledný automat vlastně jenom počítá paritu délky vstupního slova a vůbec nezáleží na tom, který ze znaků a, b přijde na vstup.



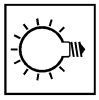
Řešený příklad 4.4: Následující zobecněný nedeterministický konečný automat převeďte na deterministický bez nedosažitelných stavů.



Řešení: Pozor, všimněme si nejprve, že daný automat má dva počáteční stavy, takže počáteční stav deterministického automatu bude tvořen množinou $\{1, 3\}$. Dalším bodem k zamyšlení je hned přechod znakem a z $\{1, 3\}$ – přímými přechody se lze dostat do stavů 2 a 1, ale navíc se můžeme dostat i do stavu 3 přechodem z 3 nejprve po ε a následně po a. Další přechody odvodíme obdobně.

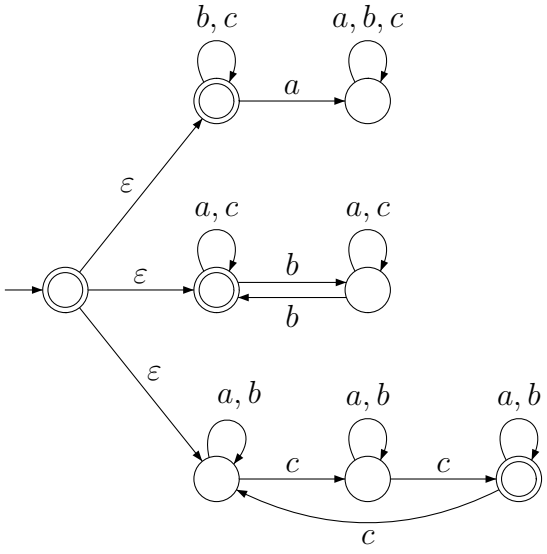


Na závěr si všimněme, že ze stavu 2 není přechod b definován, a proto v deterministickém automatu příslušný přechod povede do stavu $\emptyset$, ve kterém již automat zůstane navždy (to je někdy nazýváno „chybovým“ stavem).



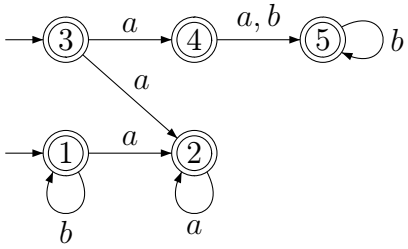
Řešený příklad 4.5: Sestrojme nedeterministický automat (ZNKA) rozpoznávající jazyk všech těch slov nad abecedou $\{a, b, c\}$, která neobsahují žádný znak a, nebo počet výskytů znaku b je sudý nebo počet výskytů znaku c dává zbytek 2 po dělení třemi.

Řešení: Požadovaný automat jednoduše poskládáme z opaku automatu v Řešeném příkladu 3.3 a z automatů v Řešeném příkladu 3.1 a ve Cvičení 3.11. Budeme mít jeden nový počáteční stav (který bude přijímající, neboť prázdné slovo je v našem jazyce) a z něj ε-přechody do počátků těchto tří vyjmenovaných automatů.



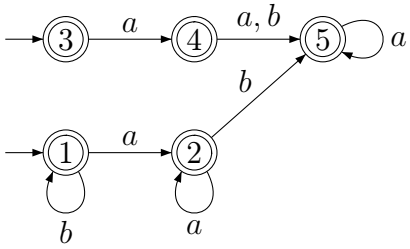
Dokážete tento automat převést na deterministický?

Cvičení 4.15:Najděte libovolné slovo nad abecedou $\{a, b\}$, které *nepatří* do jazyka přijímaného tímto nedeterministickým automatem se dvěma počátečními stavy:

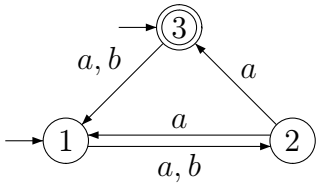


Poznámka: Pozor, přestože všechny stavy jsou přijímající, odpověď není tak triviální.

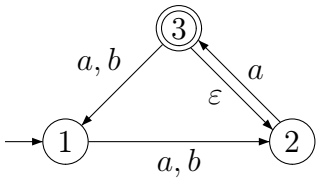
Cvičení 4.16:Najděte libovolné slovo nad abecedou $\{a, b\}$, které *nepatří* do jazyka přijímaného tímto nedeterministickým automatem se dvěma počátečními stavy:



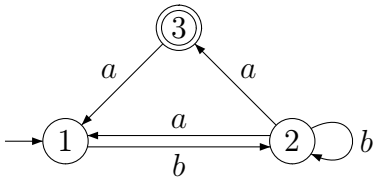
Cvičení 4.17:Následující zobecněný nedeterministický konečný automat převeďte na deterministický bez nedosažitelných stavů.



Cvičení 4.18:Následující zobecněný nedeterministický konečný automat převeďte na deterministický bez nedosažitelných stavů.



Cvičení 4.19*:Slovně popište jazyk přijímaný následujícím nedeterministickým automatem.



Kapitola 5

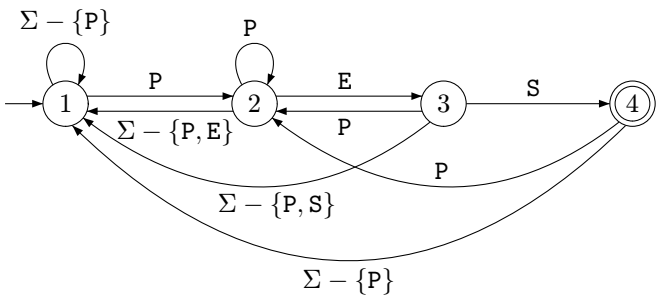
Regulární výrazy

Cíle kapitoly:

- Ukázat teoretický základ algoritmů pro vyhledávání vzorků v textu a také formálně zavést tzv. regulární výrazy pro symbolický zápis celých tříd slov (právě regulárních jazyků, jak uvidíme).
- Zvládnutí popisu regulárních jazyků pomocí regulárních výrazů.
- Pochopení algoritmů (oboustranného) převodu mezi regulárními výrazy a konečnými automaty.

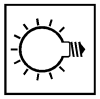
Významnou oblastí aplikací konečných automatů je vyhledávání vzorků (slov) v textu. Jistě bude čtenář souhlasit, že s takovou úlohou se při počítači potkává téměř každodenně. Na vyhledávání existují standardní softwarové nástroje, které jsou obvykle přímo zabudovány do systému nebo do textových editorů. Představme si však, že takový nástroj nemáme a chtěli bychom v rozsáhlém souboru nalézt slovo „PES“. Jak na to?

Naivní programátorský přístup by bylo z každé pozice v souboru zkontrolovat, zda se v následujících třech bytech nacházejí znaky P, E a S. Co je však nevýhodou takového přístupu? Ke znakům souboru zbytečně přistupujeme třikrát. (A bylo by to ještě horší, pokud bychom hledali dlouhá slova.) Copak by to nešlo rychleji? Pokud se nad problémem hlouběji zamyslíme, vidíme, že na každém místě souboru nám mimo aktuálního znaku stačí si pamatovat dva předchozí. To by přece měl zvládnout i konečný automat.



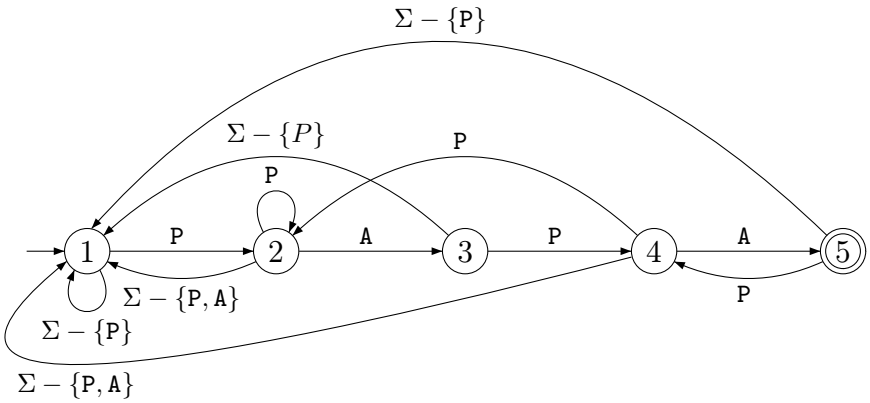
Komentář: Pro vysvětlení, náš automat hledá po sobě znaky P,E,S, přitom při výskytu jiných znaků se vrací zase na začátek.Na konci každého výskytu hledaného slova projde automat přijímajícím stavem. Formálně řečeno tento automat přijímá slova mající hledané slovo jako sufix.

Pro lepší představu si uvedeme ještě jeden příklad automatu hledajícího jedno konkrétní slovo.



Řešený příklad 5.1: Navrhněte konečný automat přijímající právě ta slova nad ASCII abecedou, která mají za sufix `PAPA`.

Řešení: Je docela zřejmé, že základem našeho automatu bude posloupnost přechodů přes symboly P, A, P, A vedoucí do přijímajícího stavu. Co však uděláme, pokud a vstupu nalezneme jiný znak? Většinou se vrátíme do počátečního stavu. Ne však vždy!



Například znak P na chybném místě automat pošle do stavu 2, což je nutné, aby tento znak byl také započítán jako první v možném sufixu „PAPA“.

Dokonce ze stavu 5 musí při dalším znaku P automat přejít rovnou do stavu 4, neboť za prvním „PAPA“ může hned následovat další (s překryvem výskytu) jako „PA*PAPA*“. Souhrnem těchto úvah získáme výše nakreslený výsledný automat.

Fakt 5.1
Simulací automatu z předchozího Příkladu 5.1 a sledováním průchodů jeho přijímajícími stavy získáme ten nejrychlejší algoritmus *pro vyhledávání vzorků v textu.*

Někdy potřebujeme vyhledávat obecnější vzorky než konkrétní slova. Např. vzorek může být 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“.

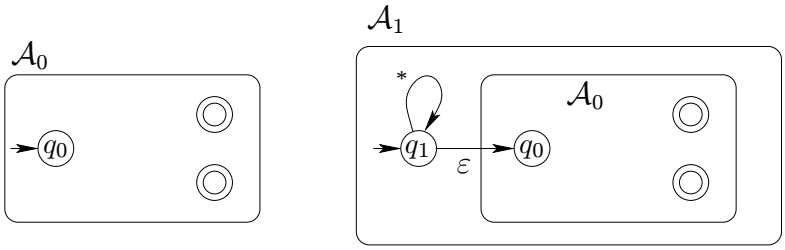
Na výrazy podobné uvedenému lze pohlížet jako na popis (reprezentaci) určitého jazyka – reprezentovány jsou ty posloupnosti písmen (v „reálu“ např. dokumenty, v našich pojmech jim říkáme prostě slova), které danému výrazu vyhovují; všimněte si, že takto reprezentovaný jazyk je pak obvykle nekonečný.

Pokud vzorek popíšeme regulárním jazykem, můžeme k jeho vyhledávání použít konečné automaty. To nám (dokonce konstruktivně) umožňuje následující tvrzení:

Věta 5.2
Pro každý regulární jazyk L_0 existuje konečný automat přijímající právě všechna ta slova mající za sufix některé slovo z L_0.

Důkaz: Nechť $\mathcal{A}_0 = (Q_0, \Sigma, \delta_0, q_0, F)$ je konečný automat přijímající jazyk L_0. Nadefinujeme zobecněný nedeterministický konečný automat $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, q_1, F)$ následovně:

- $Q_1 = Q_0 \cup \{q_1\}$, kde $q_1 \notin Q_0$ je nový počáteční stav,
- δ_1 vznikne z δ_0 přidáním smyčky na q_1 ohodnocené všemi znaky Σ a ε-hrany z q_1 do q_0.



Nakonec $\mathcal{A}_1$ (případně) převedeme dle Věty 4.3 na deterministický automat. □

Komentář: Pokud bychom konstrukci uvedenou v důkazu Věty 5.2 aplikovali na automat z Řešeného příkladu 5.1, vyšel by nám (po sloučení v podstatě zbytečného přidaného stavu q_1 s původním počátečním stavem) stejný deterministický automat. Zkuste si to sami.

Cvičení 5.1: Sestrojte deterministický konečný automat vyhledávající v textu slovo „TATAR“. Udělejte to jak heuristickým přístupem popsaným v Řešeném příkladě 5.1, tak i formálním postupem podle Věty 5.2. Vyšly vám oba automaty stejně?

Cvičení 5.2: Sestrojte deterministický konečný automat vyhledávající v textu nad abecedou $\{a, b\}$ místa, ve kterých se na konci nevyskytuje stejný znak více než dvakrát za sebou. (Prázdné slovo nechceme.)

Cvičení 5.3: Jaký jazyk vlastně máte vyhledávat v předchozí úloze?

Návod: Je to jazyk všech slov s jistými sufixy, ale navíc ještě několik speciálních krátkých slov. Najdete je?

5.1 Regulární operace a výrazy

Hlavní význam regulárních operací definovaných dříve (viz. definice 4.11) je v tom, že jimi můžeme zapisovat různé jazyky. K tomu si však nejprve musíme domluvit správnou „syntaxi“ takového zápisu – nazýváme ji regulárním výrazem. 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 5.3
Regulárními výrazy nad abecedou Σ rozumíme nejmenší množinu $RV(\Sigma)$ slov v abecedě $\Sigma \cup \{\, \emptyset, \varepsilon, +, \cdot, ^*, (,) \,\}$ (přitom předpokládáme, že $\emptyset, \varepsilon, +, \cdot, ^*, (,) \notin \Sigma$) splňující tyto podmínky:

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

Jinými slovy do $RV(\Sigma)$ patří právě všechny výrazy konstruované z $\emptyset, \varepsilon$ a písmen abecedy Σ výše uvedenými pravidly.

Komentář: Příklady jazyků zapsaných regulárními operacemi v komentáři k definici 4.11 se v zápisu regulárními výrazy vyjádří
$(0 + 11)^* \cdot (000 + 11)^*$,
$(0 + 11)^* + (010, 101)^*$.

Definice 5.4
Regulární výraz α *reprezentuje jazyk*, který označujeme $[\alpha]$, podle této rekurzivní definice

- $[\emptyset] = \emptyset$, $[\varepsilon] = \{\varepsilon\}$, $[a] = \{a\}$
- a dále $[(\alpha + \beta)] = [\alpha] \cup [\beta]$, $[(\alpha \cdot \beta)] = [\alpha] \cdot [\beta]$, $[(\alpha^*)] = [\alpha]^*$.

Komentář: Tato definice neformálně znamená, že regulární výrazy zkratkovitě zapisují regulární operace nad regulárními jazyky. Přitom atomickými výrazy jsou ε, $\emptyset$ a jednotlivé znaky abecedy. Operace se zapisují svými obvyklými symboly $\cdot$, $*$ a závorkami (), jenom sjednocení se zapisuje jako $+$.

Znovu si uvědomte, že v regulárních výrazech není operace průniku jazyků!

Značení: Při zápisu regulárních výrazů 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 $((((0\cdot1)^*\cdot1)\cdot(1\cdot1))+((0\cdot0)+1)^*)$ napíšeme $(01)^*111+(00+1)^*$.

Regulární výrazy, tak jak je formálně definujeme zde, se sice syntaxí liší od běžných „počítačových regulárních výrazů“, ale přesto zde můžeme vidět společný ideový základ. Nakonec nejdůležitějšími atributy regulárních výrazů jsou tři použité *regulární operace* – sjednocení, zřetězení a iterace. Ty se ve shodném významu objevují jak v naší definici, tak v počítačové praxi.

Poznámka: V této souvislosti je nutné upozornit, že tzv. *zpětné reference*, které se vyskytují třeba v nových verzích `regexp` knihovny, *nepatří do regulárních výrazů* v matematickém smyslu!

Otázky:

Otázka 5.4: Je zápis jazyka regulárním výrazem jednoznačný, nebo jinak, lze jeden jazyk zapsat různými výrazy?

Cvičení 5.5: Jak zapíšete regulárním výrazem jazyk všech slov, kde za počátečním úsekem znaků a se může jednou (ale nemusí vůbec) objevit znak c a pak následuje úsek znaků b?

Cvičení 5.6: A co když v předchozí úloze vyžadujeme, že úsek a i úsek b musí být neprázdný?

Cvičení 5.7: A co když v předchozí úloze ještě povolíme, že znak c se mezi a a b může vyskytnout 0-, 1- nebo 2-krát?

Cvičení 5.8: Zjistěte, zda jsou jazyky $[(011+(10)^*1+0)^*]a[011(011+(10)^*1+0)^*]$ stejné.

Cvičení 5.9: Zjistěte, zda jsou jazyky $[((1+0)^*100(1+0)^*)^*]a[((1+0)100(1+0)^*100)^*]$ stejné.

Cvičení 5.10*: Zadejte regulárním výrazem jazyk
$L = \{\, w \in \{0,1\}^* \mid$ ve w je sudý počet nul a každá jednička je bezprostředně následována nulou $\}$

Cvičení 5.11: Procvičujte si regulární výrazy tím, že jimi popíšete některé regulární jazyky, s nimiž se v našem textu (včetně úkolů) setkáváte.

5.2 Ekvivalence regulárních výrazů a jazyků

Hlavním teoretickým důvodem, proč jsme regulární výrazy zaváděli, je fakt, že popisují právě naše regulární jazyky. Dávají nám tedy alternativní (textový) způsob, jak popsat jazyk přijímaný konečným automatem.

Věta 5.5
Ke každému regulárnímu výrazu α lze sestrojit konečný automat přijímající jeho jazyk $[\alpha]$.

Důkaz (náznak): Pro jazyky $\emptyset$, $\{\varepsilon\}$, $\{a\}$ lze triviálně zkonstruovat rozpoznávající KA. Ke sjednocení dvou jazyků pak sestrojíme induktivně automat podle Věty 3.15, pro zřetězení nebo iteraci obdobně podle Věty 4.12. Takto indukcí podle délky regulárního výrazu α vždy sestrojíme příslušný automat. □

Cvičení 5.12: Myšlenky algoritmu převodu RV → ZNKA jsou načrtnuty na obrázku 5.1 – 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 5.6
Regulárními výrazy lze reprezentovat právě regulární jazyky.

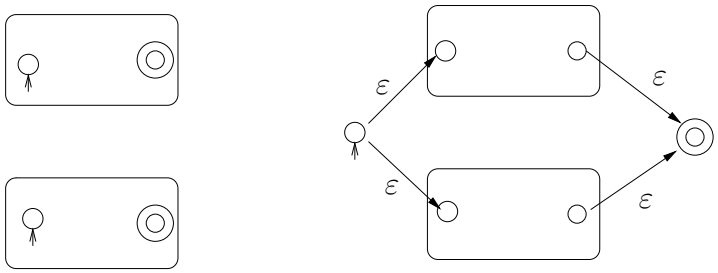
Důkaz: Jeden směr jsme ukázali větou 5.5. Zbývá tedy dokázat druhý směr ekvivalence.

(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.

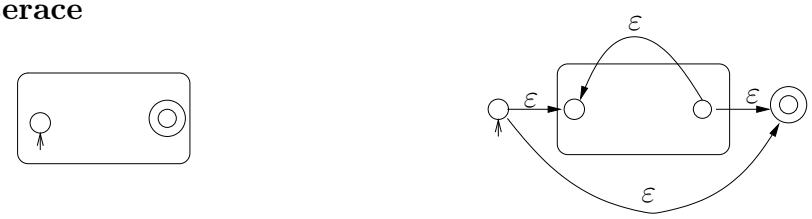
Když cesta w vede z q_0 do (pevně zvoleného přijímajícího) q_A, tak buď je to „přímá cesta“ – na níž se žádný stav neopakuje – nebo vznikne z přímé cesty

Sjednocení



Zřetězení

Iterace



Obrázek 5.1: Konstrukce ZNKA k regulárnímu výrazu

„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.

□

Otázkou je, jak pro daný konečný automat najdeme regulární výraz popisující jeho jazyk? Neformální popis v důkazu Věty 5.6 je sice svým způsobem konstruktivní, ale jen těžko si lze představit, že u větších automatů najdeme všechny možné cesty od počátečního do přijímajících stavů najednou. Jiný

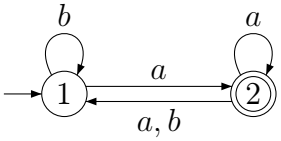
přístup ke hledání takových cest (přesněji řečeno sledů) poskytuje modifikace klasického algoritmu pro výpočet metriky grafu [Hli05, Část 8.2].

Metoda 5.7 (Výpočet regulárního výrazu z automatu)
Daný automat nemusí být deterministický. Jeho stavy libovolně označíme čísly $1, 2, \ldots, n$ a vytvoříme tabulku $n \times n$ číslovanou v řádcích i sloupcích stavy automatu.

- V nultém kroku na každé pole i, j tabulky napíšeme regulárním výrazem všechny přímé přechody ze stavu i do stavu j (šipky nebo smyčky pro $i = j$). Píšeme $\emptyset$ pokud přechod není možný.
- V kroku $t = 1, 2, \ldots, n$ k poli i, j předchozí tabulky přičteme ($+$ ve smyslu sjednocení jazyků) zřetězení předchozího přechodu z i do t, iterace přechodu z t do t a přechodu z t do j.
- Na konci sečteme (sjednotíme) všechny přechody z počátečních do přijímajících stavů.

Pro vysvětlení, v kroku $t \geq 0$ pole i, j tabulky obsahuje regulární výraz popisující všechna možná slova, která převedou automat ze stavu i do stavu j za použití vnitřních přechodů pouze přes stavy s čísly $\leq t$.

Příklad: Pro přiblížení Metody 5.7 následně uvádíme postupně vytvořené tabulky regulárních výrazů pro tento jednoduchý dvoustavový automat.



	1	2
1	$\varepsilon + b$	a
2	$a + b$	$\varepsilon + a$

	1	2
1	b^*	$a + b^*a$
2	$(a + b)b^*$	$\varepsilon + a + (a + b)b^*a$

	1	2
1	$b^* + b^*a(a + (a + b)b^*a)^*(a + b)b^*$	$(a + b^*a)(a + (a + b)b^*a)^*$
2	$(a + (a + b)b^*a)^*(a + b)b^*$	$(a + (a + b)b^*a)^*$

Například první tabulka nám říká, že ze stavu 1 přejdeme zpět či zůstaneme v 1 slovy b nebo ε. Z 1 do 2 přímo přejdeme jen znakem a. V druhé tabulce, kde jsou povoleny vnitřní přechody přes stav 1, již máme zajímavější výrazy. Například z 1 do 1 nyní kromě prázdného slova můžeme přejít libovolným řetězcem samých b, tedy slovy $[b^*]$. Ještě zajímavější je přechod z 2 zpět do 2, kde k původní možnosti $[\varepsilon + a]$ přibylo zřetězení přechodu $2 \to 1$ $[a + b]$, iterovaného přechodu uvnitř 1 $[b^*]$ a pak přechodu $1 \to 2$ $[a]$. Zbylé výrazy v tabulkách mají analogický význam a odvození...

Jak vidíme, v tabulkách vycházejí docela dlouhé výrazy (a to jsme přitom už automaticky udělali nějaká zjednodušení, jako třeba vypuštění explicitního ε u následné iterace). Nakonec nás z poslední tabulky zajímají jen přechody z počátečního stavu do přijímajících stavů, tedy zde pole $1, 2$. To je výraz $(a + b^*a)(a + (a + b)b^*a)^*$, který však dále dokážeme zjednodušit:

$$[\,(a + b^*a)(a + (a + b)b^*a)^*\,] = [\,b^*a((a + \varepsilon)b^*a)^*\,] = [\,(a + b)^*a\,]$$

Teď vidíme, že výsledek je už docela jednoduchý. Ano, náš automat skutečně přijímá všechna ta slova, která končí znakem a.

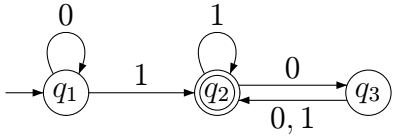
Otázky:

Otázka 5.13: Jak byste za použití automatů a zde uvedených postupů mohli mechanicky konstruovat regulární výraz popisující průnik jazyků dvou daných regulárních výrazů? Je to jednoduchý postup?

Cvičení 5.14: Sestrojte konečný automat (třeba nedeterministický) přijímající jazyk zapsaný výrazem $(0 + 11)^*01$.

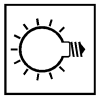
Cvičení 5.15: Upravte předchozí automat, aby přijímal jazyk zapsaný $(0 + 11)^*00^*1$.

Cvičení 5.16: Jak byste zapsali regulárním výrazem jazyk přijímaný automatem z následujícího obrázku.

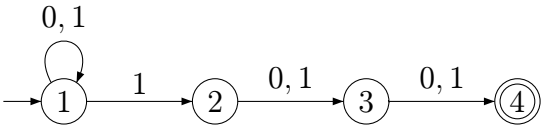


CVIČENÍ 5.17: Jak byste zapsali regulárním výrazem jazyk přijímaný automatem z Řešeného příkladu 4.2

5.3 Cvičení



ŘEŠENÝ PŘÍKLAD 5.2: Podle postupu popsaného v Metodě 5.7 sestrojte regulární výraz popisující jazyk tohoto automatu:



Řešení: Jedná se o automat z Řešeného příkladu 4.1, takže výsledný regulární výraz by měl vyjít ekvivalentní výrazu $(0+1)^*1(0+1)(0+1)$. Tabulkovým postupem nám vyjde:

	1	2	3	4
1	$\varepsilon+0+1$	1	$\emptyset$	$\emptyset$
2	$\emptyset$	ε	$0+1$	$\emptyset$
3	$\emptyset$	$\emptyset$	ε	$0+1$
4	$\emptyset$	$\emptyset$	$\emptyset$	ε

	1	2	3	4
1	$(0+1)^*$	$(0+1)^*1$	$\emptyset$	$\emptyset$
2	$\emptyset$	ε	$0+1$	$\emptyset$
3	$\emptyset$	$\emptyset$	ε	$0+1$
4	$\emptyset$	$\emptyset$	$\emptyset$	ε

	1	2	3	4
1	$(0+1)^*$	$(0+1)^*1$	$(0+1)^*1(0+1)$	$\emptyset$
2	$\emptyset$	ε	$0+1$	$\emptyset$
3	$\emptyset$	$\emptyset$	ε	$0+1$
4	$\emptyset$	$\emptyset$	$\emptyset$	ε

	1	2	3	4
1	$(0+1)^*$	$(0+1)^*1$	$(0+1)^*1(0+1)$	$(0+1)^*1(0+1)(0+1)$
2	$\emptyset$	ε	$0+1$	$(0+1)(0+1)$
3	$\emptyset$	$\emptyset$	ε	$0+1$
4	$\emptyset$	$\emptyset$	$\emptyset$	ε

Z toho již vyčteme přechodový výraz $(0+1)^*1(0+1)(0+1)$, jelikož žádné přechody s vnitřním stavem 4 zřejmě nejsou možné. (Ze 4 již nic dál nevede.) Takže nám výsledek vyšel správně, že?

Cvičení 5.18:Zapište regulárním výrazem jazyk všech slov nad abecedou $\{0,1\}$, která neobsahují tři stejné znaky za sebou.

Cvičení 5.19:Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a,b,c\}$, ve kterých se nikde nevyskytují znaky a, b hned za sebou (ani ab, ani ba).

Cvičení 5.20:Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a,b,c\}$, ve kterých se nikde nevyskytují dva znaky a hned za sebou.

Cvičení 5.21:Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a,b,c\}$, ve kterých je po a vždy b a po b vždy a.

Cvičení 5.22:Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a,b,c\}$, ve kterých je po a vždy b a po b nikdy není c.

Cvičení 5.23*:Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a,b,c\}$, ve kterých je podslovo aa a není podslovo cc.

Cvičení 5.24*:Mějme dva regulární jazyky K a L popsané regulárními výrazy

$$K = [\,0^*1^*0^*1^*0^*\,], \qquad L = [\,(01+10)^*\,].$$

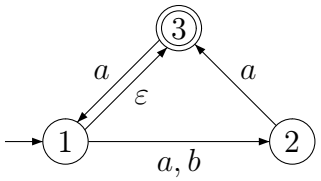
a) Jaké je nejkratší a nejdelší slovo v průniku $L \cap K$?

b) Proč žádný z těchto jazyků K a L není podmnožinou toho druhého?

c) Jaké je nejkratší slovo, které nepatří do sjednocení $K \cup L$? Je to jednoznačné?

Všechny vaše odpovědi dobře zdůvodněte!

Cvičení 5.25:Sestavte regulární výraz popisující jazyk přijímaný následujícím zobecněným nedeterministickým automatem. Použijte buď tabulkovou metodu, nebo vlastní (správnou) úvahu.



Cvičení 5.26: Sestavte konečný automat (třeba nedeterministický) přijímající jazyk zapsaný regulárním výrazem $(0 + 11)^*01$.

Cvičení 5.27: Upravte automat ze Cvičení 5.26 tak, aby přijímal jazyk zapsaný regulárním výrazem $(0 + 11)^*00^*1$.

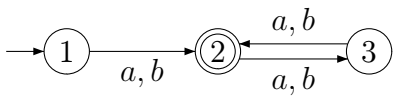
Kapitola 6

Minimalizace konečných automatů

Cíle kapitoly:

- Zvládnutí algoritmu redukce konečného automatu.

Vezmeme-li v úvahu praktické (implementační) hledisko, jistě není třeba sáhodlouze motivovat otázku možné minimalizace daného konečného automatu. Zdůrazňujeme, že zde máme na mysli minimalizaci výhradně deterministického automatu. Například v následujícím automatu



je stav 1 vlastně zbytečný, neboť lze stejně dobře začít výpočet ve stavu 3 a stav 1 vypustit. Jak ale lze takové „zmenšování“ automatu aplikovat mechanicky?

Ukážeme, že odpověď je v tomto případě potěšující – existuje algoritmus (který je dokonce velmi rychlý), který k zadanému KA sestrojí ekvivalentní automat s nejmenším možným počtem stavů.

Komentář: Jeden možný způsob zmenšení automatu jsme si již ukázali – stačí se omezit na dosažitelné stavy (do kterých vede nějaká orientovaná cesta z počátečního stavu). Je však ještě jiná možnost, daný automat totiž může

obsahovat stavy, které se „chovají stejně“ vzhledem k přijetí slov automatem. Pak přirozeně stačí všechny takto ekvivalentní stavy sloučit do jednoho a přijímaný jazyk se nezmění.

V praxi při minimalizaci automatu postupujeme opačným směrem, tedy rozkládáme množinu všech stavů automatu na neekvivalentní podmnožiny. To děláme v jednotlivých krocích tak dlouho, dokud ještě dochází k dalšímu rozlišení. Po ukončení procedury jsou podmnožiny nerozlišitelných stavů sloučeny do jednotlivých stavů nového, již minimálního, automatu.

Definice 6.1
Deterministický konečný automat nazveme *minimálním automatem*, jestliže neexistuje automat, který by s ním byl ekvivalentní a měl by menší počet stavů.

Poznámka: Pojem ekvivalence mezi automaty byl zaveden v Definici 3.8.

Značení: Uvažujme konečný deterministický automat $A = (Q, \Sigma, \delta, q_0, F)$ 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

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

($L_A(q)$ je tedy jazyk rozpoznávaný automatem, jenž vznikne z A prohlášením stavu q za počáteční; tedy množina těch slov x, pro která $\delta^*(q, x) \in F$.)

Idea konstrukce minimálního automatu je, že všechny stavy $q \in Q$ automatu A mající stejný jazyk $L_{\mathcal{A}}(q)$ „sloučíme“ vždy do jednoho stavu. Nyní můžeme vyslovit následující větu:

Věta 6.2
Existuje algoritmus, který k zadanému konečnému automatu A sestrojí minimální automat *ekvivalentní s A.*

Větu nebudeme nyní formálně dokazovat, jen si uvedeme ten algoritmu konstruující minimální automat.

Metoda 6.3
Minimalizace konečného automatu
Je dán deterministický automat $A = (Q, \Sigma, \delta, q_0, F)$ bez nedosažitelných stavů.

- Začneme rozkladem $R_0 = \{Q \setminus F,\ F\}$ množiny všech stavů A na ty nepřijímající a přijímající.

- Nechť v kroku $k \geq 0$ máme rozklad $R_k = \{P_1, P_2, \ldots, P_m\}$ množiny všech stavů. Pro všechna $i \in \{1, 2, \ldots, m\}$ a $a \in \Sigma$ uděláme následovné:

  – Rozložíme P_i na rozklad P_i^a podle toho, do kterých množin z R_k vedou ze stavů v P_i šipky se symbolem a.

- Uděláme sjednocení průniků těchto (mini-)rozkladů $R_{k+1} = \bigcup_i \bigcap_a \mathcal{P}_i^a$.

  Tím získáme všechna možná další rozlišení uvnitř tříd rozkladu R_k všemi znaky abecedy.

- Pokud $R_{k+1} \neq R_k$, tj. došlo k dalšímu rozdělení v rozkladu, vracíme se krokem $k+1$ na druhý bod postupu.

- Jinak nechť R je množina stavů po jednom vybraných z tříd rozkladu R_k (reprezentanti, přitom q_0 je také vybráno) a δ' je restrikce přechodové funkce δ na jednotlivých třídách rozkladu R_k. Pak minimální automat je $A_0 = (R, \Sigma, \delta', q_0, F \cap R)$.

Komentář: V algoritmu hledáme dvojice stavů q, q', které mají stejný jazyk $L_q = L_{q'}$. Jestliže najdeme jedno slovo, na kterém se tyto jazyky liší, již jistě stavy nemohou být ekvivalentní a nesloučíme je.

Algoritmus začíná rozdělením na přijímající a nepřijímající stavy. Jejich jazyky se od sebe liší již slovem ϵ (délky 0). Postupně v každém kroku rozlišíme stavy, které se liší delším slovem (v k-tém kroku máme rozlišeny stavy, které se od sebe liší nějakým slovem délky nejvýše k).

Jelikož automat A má jen n stavů, nejvýše $n-2$-krát v průběhu algoritmu může dojít ke zjemnění rozkladu R_k všech stavů. Proto algoritmus skončí po nejvýše $n-1$ iteracích.

Opět bez důkazu si uvedeme následující důležitou větu:

Věta 6.4
Pro každý regulární jazyk L platí, že všechny minimální konečné automaty přijímající tento jazyk jsou vzájemně izomorfní.

Protože izomorfní konečné automaty mají stejný normovaný tvar, snadno z této věty odvodíme:

Důsledek: Pro každý regulární jazyk L existuje právě jeden minimální konečný automat v normovaném tvaru.

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

Věta 6.5
Existuje algoritmus, který pro lib. zadané KA A_1, A_2 rozhodne, zda $L(A_1) = L(A_2)$.

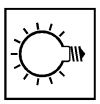
Důkaz: Stačí k oběma KA sestrojit ekvivalentní minimální automaty v normovaném tvaru a ty porovnat. Jiný důkaz plyne z partie o (konstruktivních) uzávěrových vlastnostech třídy regulárních jazyků, uvědomíme-li si, že $L(A_1) = L(A_2) \iff (L(A_1) - L(A_2)) \cup (L(A_2) - L(A_1)) = \emptyset$ a připomeneme-li si větu 3.11. □

Komentář: Při praktickém použití Metody 6.3 postupujeme tak, že si stavy automatu A v jednotlivých podmnožinách rozkladu R symbolicky označíme indexem jejich podmnožiny (třeba římskými číslicemi, aby se to nepletlo se stavy). Tímto symbolickým zápisem stavů pak vyplňujeme v každém kroku k symbolickou přechodovou tabulku. Poslední tabulka nám nakonec udává výsledný minimální automat. Blíže viz následující příklad.

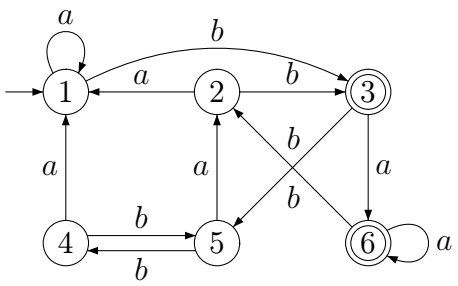
Cvičení 6.1: Zjistěte všechny dvojice stavů q, q' u následujících dvou automatů (tedy $q, q' \in \{0, 1, 2, \ldots, 9\}$), pro něž $L(q) = L(q')$.

	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



Řešený příklad 6.1: Minimalizujme následující automat:



Řešení: Podle komentáře k Metodě 6.3 vyplníme symbolickou přechodovou tabulku pro prvotní rozklad $\mathcal{R}_0 = \{\{1,2,4,5\},\{3,6\}\}$:

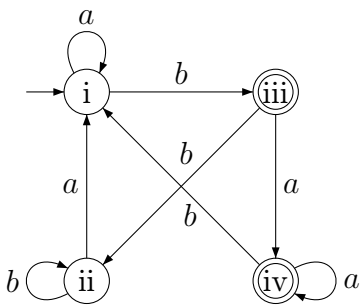
	a	b
1	i	ii
2	i	ii
4	i	i
5	i	i
3	ii	i
6	ii	i

První podmnožina rozkladu se tak dále rozpadá na $\{1,2\}$ a $\{4,5\}$, mezi kterými lze rozlišit přechodem při znaku b. V dalších dvou krocích tedy získáme obdobně přechodové tabulky:

	a	b
1	i	iii
2	i	iii
4	i	ii
5	i	ii
3	iii	ii
6	iii	i

	a	b
1	i	iii
2	i	iii
4	i	ii
5	i	ii
3	iv	ii
6	iv	i

Poslední tabulka nám udává rozklad $\mathcal{R}_2 = \{\{1,2\},\{4,5\},\{3\},\{6\}\}$, který již žádným ze znaků a, b nelze více rozlišit (zjemnit), a proto je automat udaný touto tabulkou minimální.



Otázky:

Otázka 6.2: Proč v Příkladě 6.1 vznikla u stavu ii smyčka?

Otázka 6.3: Pokud je zadaný automat již minimální, ukáže se to v postupu minimalizace hned?

Otázka 6.4*: Proč postup minimalizace automatu nefunguje na nedeterministických automatech?

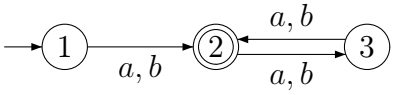
Cvičení 6.5: Sestrojte minimální (deterministický) konečný automat, který rozpoznává tentýž jazyk jako NKA zadaný následující tabulkou (a převeďte ho do normovaného tvaru):

	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	-	-

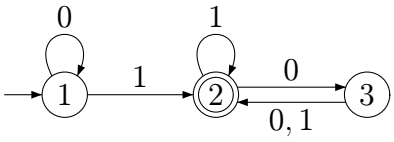
Cvičení 6.6: Sestrojte minimální (deterministické) konečné automaty, rozpoznávající jazyky reprezentované následujícími regulárními výrazy:

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

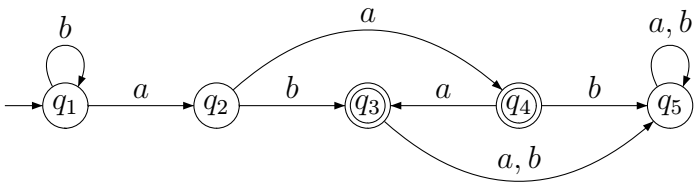
Cvičení 6.7: Minimalizujte podle uvedeného postupu automat:



Cvičení 6.8: Je tento automat minimální?



Cvičení 6.9: Je tento automat minimální?

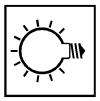


Cvičení 6.10: Najděte minimální automat ekvivalentní s následujícím automatem zadaným tabulkou.

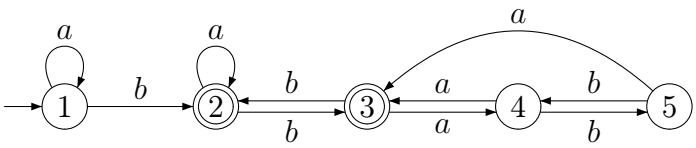
	a	b
→1	2	3
2	2	4
←3	3	5
4	2	7
←5	6	3
←6	6	6
7	7	4
8	2	3
9	9	4

Cvičení 6.11: Nechť L je jazyk všech těch slov nad abecedou $\{a, b\}$, která obsahují lichý počet výskytů znaku a a sudý počet výskytů znaku b. Jaký nejmenší možný počet stavů má konečný deterministický automat rozpoznávající jazyk L?

6.1 Cvičení



Řešený příklad 6.2: Minimalizujte následující automat:



Řešení: Začneme s rozkladem stavů $\{\{1, 4, 5\}, \{2, 3\}\}$ podle přijímajících. Opět budeme postupně vyplňovat symbolické přechodové tabulky odpovídající současnému rozkladu, přitom třídy rozkladu si budeme po řadě číslovat římskými číslicemi. Vyjde:

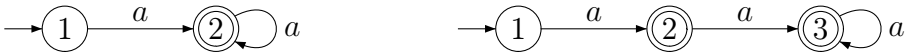
	a	b
1	i	ii
4	ii	i
5	ii	i
2	ii	ii
3	i	ii

	a	b
1	i	iii
4	iv	ii
5	iv	ii
2	iii	iv
3	ii	iii

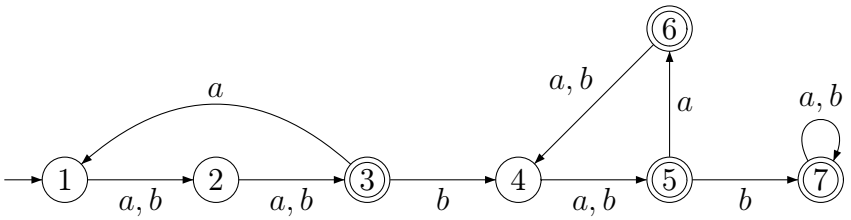
Po první tabulce se nám obě třídy rozpadnou na dvě podtřídy každá. Po druhé iteraci již k dalšímu rozlišení nedojde, a proto skončíme. Vidíme tedy, že v minimálním automatu dojde ke sloučení stavů 4 a 5 do jednoho.

Cvičení 6.12:Jsou tyto dva automaty nad abecedou $\{a\}$ ekvivalentní?

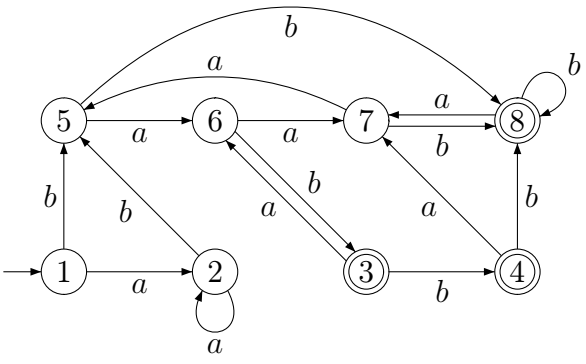
Cvičení 6.13:Jsou tyto dva automaty nad abecedou $\{a\}$ ekvivalentní?



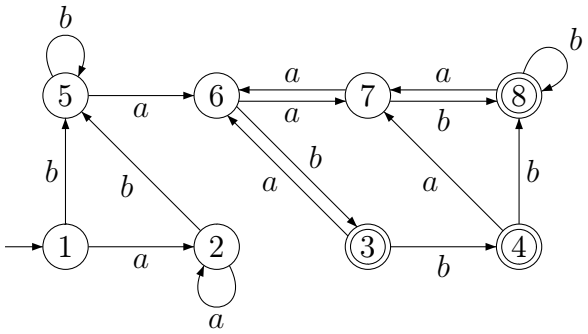
Cvičení 6.14:Zdůvodněte minimalitu tohoto automatu:



Cvičení 6.15:Minimalizujte následující automat:



Cvičení 6.16:Minimalizujte následující automat:



Cvičení 6.17:Nechť L je jazyk všech těch *neprázdných* slov nad abecedou $\{a, b\}$, která obsahují sudý počet výskytů znaku a nebo sudý počet výskytů znaku b. Jaký nejmenší možný počet stavů má konečný deterministický automat rozpoznávající jazyk L a proč?

Cvičení 6.18:Nechť L je jazyk všech těch slov nad abecedou $\{a, b, c\}$, která obsahují alespoň dva výskyty znaku a a méně než dva výskyty znaku b. Jaký nejmenší možný počet stavů má konečný deterministický automat rozpoznávající jazyk L a proč?

Cvičení 6.19:Nechť L je jazyk všech těch slov nad abecedou $\{a, b, c\}$, která obsahují alespoň dva výskyty znaku a nebo alespoň dva výskyty znaku b. Jaký nejmenší možný počet stavů má konečný deterministický automat rozpoznávající jazyk L a proč?

Cvičení 6.20:Nechť L je jazyk všech těch slov nad abecedou $\{a, b, c\}$, která obsahují alespoň dva výskyty znaku a a alespoň dva výskyty znaku b. Jaký nejmenší možný počet stavů má konečný deterministický automat rozpoznávající jazyk L?

Kapitola 7

Omezení konečných automatů

Cíle kapitoly:

- Seznámení se se způsobem prokázání neregularity konkrétního jazyka.

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?

Komentář: Neregulární jazyk musí mít nějakou vlastnost, která neumožňuje rozpoznání jeho slov, máme-li pouze omezenou paměť. Uvažujme například jazyk

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

(kde každé slovo 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!

Poznámka: 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. Kdyby 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.

7.1 Pumping lemma (pro regulární jazyky)

U $L = \{a^j b^j \mid j \geq 0\}$ 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, protože v deterministickém automatu druhé opakování a^{j-i} nutně vynutí stejné přechody vedoucí zase do stavu q_j a zbytek slova (tedy $a^{n-j} b^n$) dovede automat do přijímajícího stavu. Toto slovo 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:

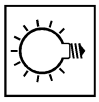
> V každém „dostatečně dlouhém“ slově regulárního jazyka L existuje „krátké“ neprázdné podslovo „blízko začátku“, jehož vynecháním či „pumpováním“ dostáváme vždy slova jazyka L.

Formálně (a přesně) to vyjadřuje následující věta.

Lemma 7.1 (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 $uv^i w \in L$.

Důkaz: Nechť L je přijímán deterministickým automatem A s n stavy. Pak přijímací sled slova z je nutně zacyklen po nejpozději n přechodech. Označme u prefix slova z, který je čten před prvním zacyklením, a v tu část z, která je čtena během prvního cyklu sledu. Pak jsou zřejmě tvrzení věty splněna – přijímací výpočet může cyklus v libovolně krát zopakovat. □

V následujícím příkladu si ukážeme použití této věty na stejném jazyku, pro který jsme již dokázali, že není regulární.



ŘEŠENÝ PŘÍKLAD 7.1: Ukažte, že jazyk $L = \{a^j b^j \mid j \geq 0\}$ není regulární.

Řešení: Pro spor předpokládejme, že L je regulární, a vezměme slovo $a^n b^n = uvw$. Pak podslovo v podle podmínky $|uv| \leq n$ Lemmatu 7.1 obsahuje jen písmena a. Takže slovo uv^2w (dvakrát zopakujeme střed v) má více a než b, a tudíž $uv^2w \notin L$, což je spor.

Poznámka: Čtenáře možná napadla otázka, zda Pumping lemma přesně charakterizuje regulární jazyky, tj. zda pro jakýkoli neregulární jazyk lze toto lemma použít pro získání sporu. Není tomu tak, jak dokládá např.jazyk

$$L = \{a, b\}^* \cup \{c\} \cdot \{c\}^* \cdot \{a^j b^j \mid j \geq 0\}\,,$$

který splňuje podmínku v Pumping lemmatu a přitom není regulární.

CVIČENÍ 7.1: Je regulární jazyk všech těch slov nad abecedou $\{a, b\}$, ve kterých je stejně znaků a jako znaků b?

Kapitola 8

Bezkontextové gramatiky a jazyky

Cíle kapitoly:

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

Po zvládnutí regulárních jazyků a s nimi asociovaných konečných automatů se nyní přesuneme do vyšších sfér tzv. *bezkontextových* jazyků. Jedná se o obecnější třídu jazyků (tj. máme v ní k dispozici silnější popisné prostředky – odvozovací pravidla) než byly regulární jazyky. S bezkontextovými jazyky a gramatikami se hojně setkáváme při syntaktické analýze textu, třeba zdrojových kódů programovacích jazyků.

Pro ilustraci uvažujme jazyk aritmetických výrazů vytvořených z prvků abecedy $\{ a, +, \times, (,) \}$ (číselné konstanty či proměnné teď nejsou podstatné, proto všechny reprezentujeme jedním atomickým symbolem a). Příklady takových výrazů jsou $a + a \times a$ nebo $(a + a) \times a$. Je zřejmé, že kvůli nutnému počítání levých a pravých závorek se nejedná o regulární jazyk, nemůžeme jej tedy zadat regulárním výrazem nebo konečným automatem. Na druhou stranu, pomocí v praxi zaužívaného způsobu zápisu, může být množina všech takových aritmetických výrazů popsaná těmito (přepisovacími) pravidly:

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

Viděli jste už někdy podobný způsob popisu v manuálech příkazů nebo funkcí? Formálně zde vlastně vidíme zápis *bezkontextové gramatiky* pomocí odvozovacích *pravidel*.

8.1 Odvození aritmetických výrazů

Podívejme se znova na výše uvedený příklad popisu aritmetických výrazů očima teorie.

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

Značení: Napíšeme-li místo $\langle EXPR\rangle$ jen E a pravé strany pravidel se stejnou levou stranou sloučíme do jednoho řádku, oddělené symbolem „|“, vznikne zkrácený zápis

$$E \longrightarrow E + E \mid E \times E \mid (E) \mid a\,.$$

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 Obrázku 8.1 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 Obrázku 8.1 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 ze stránky 110 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) a je přitom jednoznačná. Např. naše slovo $a+a\times a$ má v ní jedinou levou derivaci (jediný derivační strom):

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

Cvičení 8.1: Přidejte do aritmetických výrazů operaci umocnění a^2 nejvyšší priority a napište příslušná odvozovací pravidla.

Cvičení 8.2: Přidejte do aritmetických výrazů operátor porovnání $=$ (ne přiřazení), který už dále nesmí vystupovat jako člen v součtech či součinech.

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

Definice 8.1
Bezkontextová gramatika je čtveřice (tj. je dána uspořádanou čtveřicí) $G = (\Pi, \Sigma, S, P)$, 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 $\beta \in (\Pi \cup \Sigma)^*$.

Poznámka: Slovo „bezkontextová“ v názvu gramatiky znamená, že na levé straně každého pravidla stojí jeden neterminál bez sousedních symbolů (tedy není v „kontextu“).

Značení: Pro jednoduchost používáme konvenci, že neterminální symboly jsou značené velkými písmeny a terminální symboly malými písmeny.

Definice 8.2
Mějme gramatiku $G = (\Pi, \Sigma, S, P)$ a 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.

Definice 8.3
Jazyk generovaný gramatikou G, označme jej $L(G)$, je je množinou všech slov $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

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:

- Všimněte si dobře, že do jazyka generovaného gramatikou patří pouze ta slova odvozená z S, která jsou složená *jen z terminálů*.
- 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.
- 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 8.4
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. $\alpha = uX\delta$,

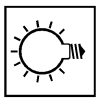
$\beta = u\gamma\delta$ pro nějaké $u \in \Sigma^*$, $\delta \in (\Pi \cup \Sigma)^*$. Odvození $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$ je levým odvozením (levou derivací), jestliže pro vš. $i = 0, 1, \ldots, n-1$ lze α_i přepsat na α_{i+1} levým přepsáním. (Pravé odvození 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 8.5
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.

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 $w = a_1 a_2 \ldots a_n$.

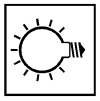
Poznámka: Všimněme si, že každému odvození slova w v gramatice G odpovídá (přirozeným způsobem) právě jeden derivační strom pro w; derivačnímu stromu pro w odpovídá obecně více odvození slova w, ovšem např. právě jedno levé odvození.



Řešený příklad 8.1: Navrhněte gramatiku generující jazyk palindromů $L_1 = \{ww^R \mid w \in \{a, b\}^*\}$, kde w^R značí slovo w zapsané pozpátku.

Řešení: Zde si vystačíme s jediným neterminálním symbolem S a úvahou, že takový palindrom se vytvoří z menšího palindromu přidáním stejného znaku na začátek jako na konec. Zapsáno pravidly

$$S \longrightarrow \varepsilon \mid aSa \mid bSb\,.$$

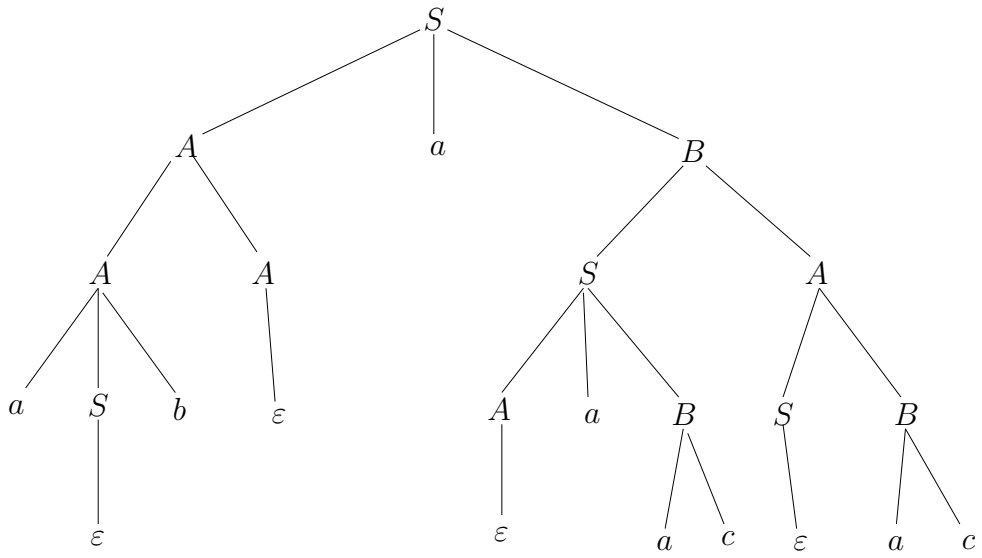


Řešený příklad 8.2: Co nejvýstižněji slovně popište jazyk generovaný gramatikou

$$S \longrightarrow SbS \mid a\,.$$

Řešení: První pravidlo $S \longrightarrow SbS$ vytváří jenom a pouze střídavé řetězce „$SbSb \ldots SbS$“. Jelikož druhé pravidlo $S \longrightarrow a$ končí jen terminálem, není na něj možné navázat dalšími pravidly, a proto si jeho výskyty můžeme nechat až na konec odvozování. Proto všechna možná odvozená slova jsou získaná z „$SbSb \ldots SbS$“ dosazením a, tj. generován je jazyk všech slov tvaru „$abab \ldots aba$“.

Cvičení 8.3: Na Obrázku 8.1 je derivační strom popisující odvození slova $w = abaaacac$ podle jisté bezkontextové gramatiky G.



Obrázek 8.1: Derivační strom pro slovo $w = abaaacac$

- Napište levé odvození slova w podle gramatiky G.
- Napište pravé odvození slova w podle gramatiky G.
- Najděte rozklad $w = w_1w_2w_3$ s $w_2 \neq \varepsilon$ tak, aby slovo $w_1w_2w_2w_3$ také patřilo do $L(G)$.
- Vypište pravidla gramatiky G.

Definice 8.6
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*).

Cvičení 8.4: Lze v úkolu 8.3 z dostupné informace zjistit něco ohledně víceznačnosti příslušné gramatiky?

Poznámka: Otázky konstrukce minimální gramatiky či ekvivalence dvou gramatik jsou obecně *algoritmicky neřešitelné*. To je velký rozdíl proti konečným automatům...

Otázky:

Otázka 8.5: Je možný postup odvození daného slova vždy jen omezeně dlouhý?

Otázka 8.6: Jaký jazyk generují pravidla $SaS \mid Sb$?

Otázka 8.7: Jak lze výpočet konečného automatu simulovat pravidly gramatiky?

Cvičení 8.8: Generuje gramatika z Příkladu 8.1 skutečně všechny palindromy nad $\{a, b\}$?

Cvičení 8.9: Sestrojte gramatiku generující jazyk $\{0^n 1^m 0^n \mid m, n \geq 0\}$.

Cvičení 8.10: Jaký jazyk generuje gramatika

$$S \longrightarrow aBC \mid aCa \mid bBCa$$
$$B \longrightarrow bBa \mid bab \mid SS$$
$$C \longrightarrow BS \mid aCaa \mid bSSc$$

Cvičení 8.11: Generuje gramatika

$$S \longrightarrow abSa \mid \varepsilon$$

stejný jazyk jako gramatika

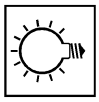
$$S \longrightarrow aSa \mid bS \mid \varepsilon \quad ?$$

Cvičení 8.12: Navrhněte bezkontextové gramatiky generující následující jazyky:

- $L_1 = \{\, w \in \{a,b\}^* \mid w$ obsahuje podslovo $baab \,\}$
- $L_2 = \{\, w \in \{a,b\}^* \mid |w|_b \bmod 3 = 0 \,\}$
- $L_3 = \{\, ww^R \mid w \in \{a,b\}^* \,\}$
- $L_4 = \{\, 0^n 1^m 0^n \mid m,n \geq 0 \,\}$
- $L_5 = \{\, 0^n 1^m \mid 1 \leq n \leq m \leq 2n \,\}$

Cvičení 8.13: Snažte se co nejvýstižněji charakterizovat jazyk generovaný gramatikou
$S \longrightarrow bSS \,|\, a$

8.2 Cvičení



Řešený příklad 8.3: Definují už známá pravidla aritmetických výrazů jednoznačnou gramatiku?

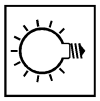
$$\begin{array}{lcl} E & \longrightarrow & E + E \mid F \\ F & \longrightarrow & (E) \mid F \times F \mid a \end{array}$$

Řešení: Ne. Přestože jsme pravidla navrhli tak, aby výsledek vyhodnocení byl aritmeticky jednoznačný, ona gramatika není jednoznačná ve smyslu definice! Pro dosažení jednoznačnosti gramatiky ještě musíme zavést pravidlo,

že stejné operace se vyhodnocují zleva doprava, což zajistíme opět prioritními neterminály

$$\begin{array}{lcl} E & \longrightarrow & E + F \mid F \\ F & \longrightarrow & (E) \mid F \times G \mid G \\ G & \longrightarrow & (F) \mid a \end{array}$$

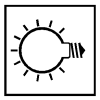
$G \longrightarrow (F) \mid a$ Všimněme si, že (dle čtení pravidel odvození „odzadu“) nám pravidlo $E \longrightarrow E + F$ říká, že $+$ vpravo se vyhodnotí až nakonec, pokud to možné závorky psané v F neurčí jinak.



ŘEŠENÝ PŘÍKLAD 8.4: Přidejte do aritmetických výrazů z příkladu 8.3 operaci rozdílu, opět s vlastností jednoznačného vyhodnocení vzhledem k aritmetickým pravidlům.

Řešení: Zde si musíme dávat pozor – odečítání není na rozdíl od sčítání asociativní, neboli $(a-b)-c$ je něco jiného než $a-(b-c)$. Pokud není výraz závorkovaný, odečítání se provádí zleva, takže $a-b-c$ odpovídá $(a-b)-c$a tak by to naše gramatika měla i odvozovat. K tomu využijeme již zavedeného prioritního neterminálu F, takže pravidla zní

$$\begin{array}{lcl} E & \longrightarrow & E + F \mid F \mid E - F \\ F & \longrightarrow & (E) \mid F \times G \mid G \\ G & \longrightarrow & (F) \mid a \end{array}$$



ŘEŠENÝ PŘÍKLAD 8.5: Sestrojte bezkontextovou gramatiku generující všechna slova nad abecedou $\{a, b\}$ mající stejně výskytů symbolů a jako b.

Řešení: Možná by čtenáře mohlo napadnou používat pravidla jako $S \longrightarrow abS \mid baS$nebo i složitější podobného typu, která samozřejmě zajistí stejný počet a jako b, ale nevygenerují slova s dlouhými úseky „$aa \ldots a$“. Správnějším přístupem je expandovat hlavně neterminál S na všechna možná místa mezi terminálními znaky. Například pravidly

$$S \longrightarrow SaSbS \mid SbSaS \mid \varepsilon$$

která vytvářejí všechna slova mající stejně a jako b a navíc mající symbol S mezi každými dvěmi písmeny a, b a na začátku i na konci. Převodem $S \longrightarrow \varepsilon$ se nakonec všech S snadno zbavíme.

Proč tedy popsaná gramatika generuje všechna taková slova? To snadno dokážeme indukcí podle délky slova. Prázdné slovo je vytvořeno. Je-li w neprázdné slovo obsahující stejně a jako b a mající symbol S mezi každými z písmen a, b, pak v něm nutně je někde úsek $\ldots SaSbS \ldots$ nebo $\ldots SbSaS \ldots$ a ten lze naší gramatikou vytvořit ze slova o 4 znaky kratšího aplikováním příslušného pravidla z $S \longrightarrow SaSbS \mid SbSaS$.

Řešený příklad 8.6: Je následující gramatika jednoznačná?

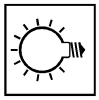
$$S \longrightarrow SaSbS \mid SbSaS \mid \varepsilon$$

Řešení: Není, už množství stejných neterminálů na pravých stranách pravidel by vám mělo napovědět, že asi bude možných více odvození. Není však tak jednoduché různá odvození nalézt, že? Třeba vezměme slovo $abab$ – to lze odvodit buď jako

$$S \to SaSbS \to abS \to abSaSbS \to abab\,,$$

nebo úplně jinak jako

$$S \to SaSbS \to aSb \to aSbSaSb \to abab\,.$$



Řešený příklad 8.7: Generují obě následující gramatiky tentýž jazyk?

$$S \longrightarrow aaSbb \mid ab \mid aabb$$
$$S \longrightarrow aSb \mid ab$$

Řešení: Druhá gramatika zřejmě generuje jazyk $\{a^i b^i \mid i \geq 1\}$. Zbývá tedy ověřit, zda první gramatika generuje tentýž jazyk. I první gramatika generuje jazyk, ve kterém jsou nejprve znaky a a až pak znaky b. Pravidlo $S \longrightarrow aaSbb$ vygeneruje všechna slova tvaru $a^j S b^j$, kde $j \geq 0$ je sudé. Takže pokud i z druhé gramatiky je liché, jsme hotovi užitím $S \longrightarrow ab$. Pokud máme generovat slovo $a^i b^i$ pro sudé $i \geq 2$, nakonec aplikujeme pravidlo $S \longrightarrow aabb$, čímž vznikne slovo $a^{j+2} b^{j+2}$ a $i = j+2$. Takže jsme dokázali, že obě gramatiky generují tutéž množinu slov nad $\{a, b\}$.

Cvičení 8.14:Mezi následujícími třemi jazyky nad abecedou $\{a, b\}$ najděte všechny, které jsou regulární, a další jazyk, který je bezkontextový a není regulární.

a) $[\,(ab)^*ba\,]$

b) $\{\,a^ib^ja \mid i,j \in N\,\}$

c) $\left\{\,a^ib^ja^k \mid i,j,k \in N,\, i+j=k\,\right\}$

Cvičení 8.15*:Mezi následujícími třemi jazyky nad abecedou $\{a,b\}$ najděte všechny, které jsou regulární, a další jazyk, který je bezkontextový a není regulární.

a) $[\,a^*b(a+b)\,]$

b) $\{\,a^ib^j \mid i,j \in N, i<j\,\}$

c) $\{\,a^i \mid i$ je prvočíslo$\,\}$

Cvičení 8.16:Jak byste napsali gramatiku k jazyku $\{\,a^ib^j \mid i,j \in N, i<j\,\}$?

Cvičení 8.17:Zapište odvozovacími pravidly bezkontextové gramatiky jazyk všech těch palindromů nad abecedou $\{a,b\}$, jejichž délka je násobkem čtyř.

Cvičení 8.18*:Zapište odvozovacími pravidly bezkontextové gramatiky jazyk všech těch palindromů nad abecedou $\{a,b\}$, jejichž délka je násobkem tří.

Cvičení 8.19:Generují obě následující gramatiky tentýž jazyk?

$$\begin{aligned} S &\longrightarrow aaSbb \mid ab \mid \varepsilon \\ S &\longrightarrow aSb \mid ab \end{aligned}$$

Cvičení 8.20:Generují obě následující gramatiky tentýž jazyk?

$$\begin{aligned} S &\longrightarrow aaSb \mid ab \mid \varepsilon \\ S &\longrightarrow aSb \mid aab \mid \varepsilon \end{aligned}$$

Cvičení 8.21:Rozhodněte, která z následujících dvou gramatik generuje regulární jazyk, tj. přijímaný také konečným automatem.

a) $S \longrightarrow aSb \mid bSa \mid \varepsilon$

b) $S \longrightarrow abS \mid baS \mid \varepsilon$

Cvičení 8.22:Rozhodněte, která z následujících dvou gramatik generuje regulární jazyk, tj. přijímaný také konečným automatem.

a) $S \longrightarrow ASa \mid \varepsilon$; $A \longrightarrow b$

b) $S \longrightarrow BSa \mid \varepsilon$; $B \longrightarrow a$

Cvičení 8.23:Rozhodněte, která z následujících dvou gramatik generuje regulární jazyk, tj. přijímaný také konečným automatem.

a) $S \longrightarrow aSb \mid bSa \mid bbS$

b) $S \longrightarrow ab \mid ba \mid bbS$

Cvičení 8.24**:Uměli byste nalézt jednoznačnou gramatiku pro řešení Příkladu 8.5??

Cvičení 8.25*:Napište gramatiku pro jazyk všech těch slov nad abecedou $\{a, b, c\}$, ve kterých za každým úsekem znaků a bezprostředně následuje dvakrát delší úsek znaků b.

Cvičení 8.26:Zredukujte následující bezkontextovou gramatiku

$$\begin{array}{lcl} S & \longrightarrow & aSb \mid aAbb \mid aDaS \mid \varepsilon \\ A & \longrightarrow & aAB \mid bB \\ B & \longrightarrow & aAb \mid BB \mid E \\ C & \longrightarrow & CC \mid cS \\ D & \longrightarrow & aSb \mid cD \mid aEE \\ E & \longrightarrow & EB \mid bD \end{array}$$

Cvičení 8.27:Vytvořte pro jazyk $L = \{a^i b^j \mid i, j > 0\}$ gramatiku v Greibachové normální formě.

Kapitola 9

Zásobníkové automaty

Cíle kapitoly:

- Pochopení pojmu zásobníkového automatu a zvládnutí jeho návrhu v jednoduchých případech.

Víme, že např. jazyk

$$\begin{array}{ll}\{ & \langle begin\rangle\langle end\rangle, \\ & \langle begin\rangle\langle begin\rangle\langle end\rangle\langle end\rangle, \\ & \langle begin\rangle\langle begin\rangle\langle begin\rangle\langle end\rangle\langle end\rangle\langle end\rangle, \\ & \ldots \quad \}\end{array}$$

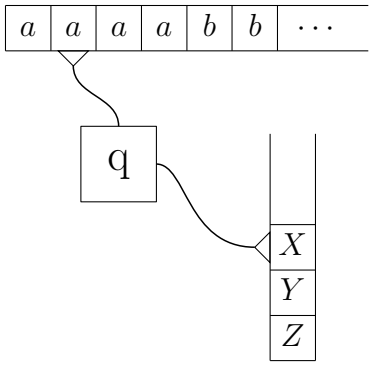
nebo „ekvivalentní“ jazyk

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

Snadno ovšem takový jazyk (tedy slova daného jazyka) rozpoznáme zařízením podobným konečnému automatu, které může navíc používat neomezenou paměť typu *zásobník*. Přečtené symboly a se ukládají do zásobníku a při čtení symbolů b se pak tyto zásobníkové symboly odebírají. Tímto 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 9.1.

Čtenář by si měl být schopen udělat představu, jak takové zařízení pracuje a jakým způsobem reprezentuje (rozpoznává) jazyk. Tuto představu je pak



Obrázek 9.1: Vnější pohled na zásobníkový automat

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 9.1
Zásobníkový automat, zkráceně ZA, M je šestice $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$, 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* a
- δ je zobrazení množiny $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ do množiny všech konečných podmnožin množiny $Q \times \Gamma^*$.

Značení: Přechodová funkce $\delta : (q, a, z) \to (q', w)$ znamená, že ve stavu q při čtení vstupního symbolu a a současném vyzvednutí zásobníkového symbolu

z přejde ZA do stavu q' a na vrch zásobníku zapíše slovo $w \in \Gamma^*$. Pokud místo z je v pravidle vlevo ε, znamená to, že se ze zásobníku nic nečte.

Poznámka: Důležitým rozdílem ZA od běžného automatu je, že v ZA nejsou žádné přijímací stavy, místo toho je slovo přijato, pokud výpočet (resp. některá jeho nedeterministická větev) dojde na konci slova ke stavu s prázdným zásobníkem.

Definice 9.2
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.

Výpočtem zásobníkového automatu M, začínajícím v konfiguraci K, rozumíme posloupnost konfigurací $K_0, K_1, K_2, \ldots, K_n$, kde $K_0 = K$ a $K_i \vdash_M K_{i+1}$ 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 $K_0 = (q_0, w, Z_0)$ a $K_n = (q, \varepsilon, \varepsilon)$ pro nějaký $q \in Q$.

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

Cvičení 9.1:

a) Sestrojte zásobníkový automat rozpoznávající jazyk $L = \{\, wc(w)^R \mid w \in \{a, b\}^* \,\}$.

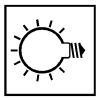
b) Navrhněte ZA rozpoznávající jazyk $\{ww^R \mid w \in \{0, 1\}^*\}$. Jistě přitom využijete nedeterminismus (o důvodu se zmíníme později).

c) Sestrojte zásobníkový automat rozpoznávající jazyk $L = \{\, u \in \{a, b, c\}^* \mid$ po vynechání všech výskytů symbolu c z u dostaneme slovo ve tvaru $w(w)^R \,\}$.

Věta 9.3
Nedeterministické zásobníkové automaty rozpoznávají právě bezkontextové jazyky (a jsou takto ekvivalentní bezkontextovým gramatikám).

Komentář: Pokud budeme chtít v praxi napsat program, který parsuje vstupní aritmetický výraz a vyhodnotí jej, naše řešení nejspíše bude přirozeně tíhnout k použití zásobníkové struktury pro ukládání dosud nevyhodnocených podvýrazů. Vysvětlení nám k tomu dává právě předchozí věta – jak už víme, aritmetické výrazy vyhodnocujeme podle pravidel vhodné bezkontextové gramatiky, a tato gramatika je emulovatelná na zásobníkovém automatu.

I zásobníkové automaty mají svá omezení, z nichž nejdůležitější uvidíme v následujícím příkladě.



Řešený příklad 9.1: Navrhněte zásobníkový automat pro jazyk $a^i b^i c^i$.

Řešení: Zhruba řečeno, zásobníkový automat „může počítat“ počet písmen a jen jednou, srovnávání s b v druhé části pak už uložený počet nutně „zničí“. Proto nelze zajistit ještě porovnání počtu třetího symbolu c. Uvedený jazyk není bezkontextový, tj. nelze jej rozpoznat zásobníkovým automatem.

9.1 Vlastnosti bezkontextových jazyků

Opět v tomto oddíle uvedeme jen *velmi stručný přehled* některých vlastností bezkontextových jazyků.

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.

Poznámka: Příslušnou reprezentací jsou samozřejmě bezkontextové gramatiky či zásobníkové automaty. Lze volit, co je vhodnější.

Věta 9.4
CFL je uzavřena vůči sjednocení, zřetězení, iteraci, zrcadlovému obrazu, substituci (tedy i homomorfismu).

Důkaz: K libovolným 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 $L(G_a)$. □

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

Věta 9.5
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. □

Poznámka: Je možné definovat i *deterministické ZA*. Ty jsou však striktně slabší než nedeterministické, takže se používají méně často. (Neboli žádný obecný převod nedeterministického ZA na deterministický na rozdíl od klasických automatů neexistuje.)

Otázky:

Otázka 9.2: Jakým způsobem může zásobníkový automat rozpoznat prázdné slovo?

Otázka 9.3: Jak můžeme zásobníkovým automatem simulovat obyčejný automat?

Cvičení 9.4: Proč není bezkontextový jazyk L_3 všech těch slov nad abecedou $\{a, b, c\}$ obsahujících stejně výskytů od každého znaku a, b, c?

9.2 Cvičení

Cvičení 9.5: Sestrojte zásobníkový automat pro jazyk $\{wc(w)^R \mid w \in \{a,b\}^*\}$

Cvičení 9.6: Jaký jazyk přijímá následující zásobníkový automat?

$Q = \{p, q\}, \Sigma = \{a, b, c\}, \Gamma = \{A, B, Z\}$, počáteční zásobníkový symbol je Z, počáteční stav p,
$\delta(p, a, Z) = \{(p, AZ)\}$
$\delta(p, b, Z) = \{(p, BZ)\}$
$\delta(p, c, Z) = \{(p, Z)\}$
$\delta(p, a, A) = \{(p, AA)\}$
$\delta(p, b, A) = \{(p, BA)\}$
$\delta(p, c, A) = \{(p, A)\}$
$\delta(p, a, B) = \{(p, AB)\}$
$\delta(p, b, B) = \{(p, BB)\}$
$\delta(p, c, B) = \{(p, B)\}$
$\delta(p, \varepsilon, Z) = \{(q, \varepsilon)\}$
$\delta(p, \varepsilon, A) = \{(q, A)\}$
$\delta(p, \varepsilon, B) = \{(q, B)\}$
$\delta(q, a, A) = \{(q, \varepsilon)\}$
$\delta(q, c, A) = \{(q, A)\}$
$\delta(q, b, B) = \{(q, \varepsilon)\}$
$\delta(q, c, B) = \{(q, B)\}$
$\delta(q, c, Z) = \{(q, Z)\}$
$\delta(q, \varepsilon, Z) = \{(q, \varepsilon)\}$

Cvičení 9.7: Jaký jazyk přijímá následující zásobníkový automat?

$Q = \{q\}, \Sigma = \{a, b, c\}, \Gamma = \{A, B, C, S\}$, počáteční zásobníkový symbol je S,
$\delta(q, \varepsilon, S) = \{(q, ASA), (q, BSB), (q, CS), (q, SC), (q, \varepsilon)\}$
$\delta(q, a, A) = \{(q, \varepsilon)\}$
$\delta(q, b, B) = \{(q, \varepsilon)\}$
$\delta(q, c, C) = \{(q, \varepsilon)\}$

Cvičení 9.8: Sestrojte zásobníkový automat přijímající jazyk generovaný

následující gramatikou

$$\begin{aligned} A &\longrightarrow A + B \mid B \\ B &\longrightarrow B * C \mid C \\ C &\longrightarrow (A) \mid a \end{aligned}$$

Kapitola 10

Chomského hierarchie

Cíle kapitoly:

- Pochopení Turingových strojů jako reprezentanta nejobecnějších algoritmických prostředků k popisu jazyků.
- Zvládnutí klasické klasifikace jazyků podle Chomského.
- Získání schopnosti zařadit běžné jazyky do uvedené hierarchie.

10.1 Turingovy stroje

Představme si konečný automat jako stroj, který čte zleva doprava slovo zapsané na vstupní pásce a přechází při tom mezi svými vnitřními stavy. Pro názornost říkáme, že ta část automatu, která čte symboly z pásky, se nazývá *hlava*, a mluvíme o jejím posunu (pohybu) po pásce.

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.

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 sekvenční přístup k „paměti“ (v jednotlivém kroku má přístup jen k jedné buňce, „posunout“ se v jednom kroku může vždy jen na buňku sousední).

Definice 10.1
Turingův stroj, zkráceně TS, M je 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*,
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ je přechodová funkce.

Předpokládáme, že v $\Gamma \setminus \Sigma$ je vždy obsažen speciální prvek $\Box$ označující *prázdný znak* (tomuto prvku se také říká *blank*).

Konfigurace Turingova stroje je dána aktuálním stavem řídící jednotky, obsahem pásky a aktuální pozicí hlavy.

Výpočet Turingova stroje začíná v konfiguraci, kde řídící jednotka je v počátečním stavu q_0, vstupní slovo je zapsáno na pásce, přičemž hlava Turingova stroje se nachází na jeho prvním symbolu, a kde všechny ostatní políčka pásky (kromě těch, která obsahují vstupní slovo) obsahují symbol $\Box$.

Formálně můžeme konfiguraci Turingova stroje M popisovat jako slovo tvaru uqv, kde $u, v \in \Gamma^*$ a $q \in Q$. Slovo uqv označuje konfiguraci, kde aktuální stav řídící jednotky je q, na políčkách pásky nalevo od aktuální pozice hlavy je zapsáno slovo u, na políčku kde se nachází hlava je zapsán první symbol slova v (resp. symbol $\Box$, pokud $v = \varepsilon$) a na políčkách napravo od hlavy jsou zapsány zbývající symboly slova v. Všechna zbývající políčka pásky jsou prázdná (tj. obsahují symbol $\Box$).

Všimněte si, že při tomto způsobu zápisu konfigurací ztotožňujeme konfigurace uqv a $\square^i uqv\square^j$ $(i, j \geq 0)$, speciálně tedy konfiguraci qv ztotožňujeme s konfigurací $\square qv$, a podobně ztotožňujeme uq s $uq\square$.

Pokud $w \in \Sigma^*$ je vstupem Turingova stroje M, pak *počáteční konfigurace*, ve které výpočet stroje M nad w začíná, je konfigurace $q_0 w$. Výpočet Turingova stroje M končí v některé *koncové konfiguraci*. Konfigurace uqv je koncovou konfigurací, jestliže $q \in F$.

Jeden krok Turingova stroje z aktuální konfigurace do další je určen přechodovou funkcí δ. Řekněme, že aktuální stav řídící jednotky je q, na políčku pásky na pozici, kde se nachází hlava, je zapsán symbol a, a že $\delta(q, a) = (q', b, d)$. Pak novým stavem řídící jednotky bude q', na aktuální pozici hlavy se zapíše symbol b (místo a), a poté se hlava posune o d políček doprava (tj. pokud $d = -1$, posune se o jedno políčko doleva, pokud $d = 0$, zůstane na místě, a pokud $d = 1$, posune se o jedno políčko doprava).

Formálně můžeme kroky Turingova stroje M popsat pomocí relace $\vdash_M$, kde $K \vdash_M K'$ znamená, že stroj M přejde jedním krokem z konfigurace K do konfigurace K', neboli, že K *vede v jednom kroku ke konfiguraci* K'. Vztah $K \vdash_M K'$ platí právě tehdy, když platí jedna z následujících možností (ve všech případech předpokládáme $K = uaqbv$, kde $u, v \in \Gamma^*$, $a, b \in \Gamma$):

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

Poznámka: Pokud je zřejmé z kontextu o jaký stroj M se jedná, často místo $\vdash_M$ píšeme jen $\vdash$.

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

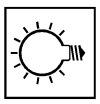
$$L(M) = \{w \in \Sigma^* \mid w \text{ je přijímáno } M\}.$$

Turingovy stroje neslouží jen k přijímání slov, ale můžeme je použít i k realizaci výpočtů, kde výsledkem je slovo z nějaké abecedy. Jednou možností,

jak toto definovat, je říci, že výstupem Turingova stroje je to, co „zbude“ na pásce po skončení výpočtu Turingova stroje (tj. obsah pásky, ze kterého odstraníme symboly $\Box$). Výpočet Turingova stroje nad zadaným vstupem se tedy zastaví v momentě dosažení koncového stavu (dojde-li k tomu vůbec). Výstupem (výpočtu) pak rozumíme řetězec z $(\Gamma - \{\Box\})^*$ zapsaný na pásce v příslušné koncové konfiguraci.

Poznámka: Pokud na začátek a konec slova přidáme speciální ukončovací symboly #, kde # $\in \Gamma - \Sigma$, a definujeme, že hlava se nesmí nikdy dostat mimo úsek vymezený těmito symboly (a nesmí tyto ukončovací znaky nikdy přepsat), dostaneme speciální variantu Turingova stroje nazývanou *lineárně omezený automat* (LBA – z anglického „linear bounded automaton“).

Značení: Turingovy stroje můžeme podobně jako konečné automaty zobrazovat jako grafy, kde vrcholy odpovídají stavům řídící jednotky a hrany reprezentují jednotlivé přechody. Pro označování přechodů používáme následující konvenci: Přechod $\delta(q, a) = (q', b, d)$ značíme zkratkou $a \to b; +$ nebo $a \to b; 0$ nebo $a \to b; -$ podle směru pohybu hlavy po přečtení, to vše zapsáno u šipky z q do q'. Pokud znak na pásce nechceme přepsat, zkráceně píšeme jen $a; +$ a můžeme uvádět i více čtených znaků najednou.



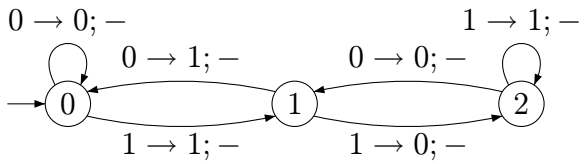
Řešený příklad 10.1: Sestrojte TS, který na začátku dostane na pásce napsané binární číslo, tj. slovo z $\{0, 1\}^*$ a nic víc (zbytek vyplněný $\Box$). Pro jednoduchost TS začíná práci na poslední číslici vpravo. Úkolem TS je vynásobit zadané číslo třemi.

Řešení: Zde si vzpomeneme na primitivní školní algoritmus násobení: Pokud je poslední číslice 0, v trojnásobku bude také 0 a žádný přenos. Pokud je poslední číslice 1, v trojnásobku bude také 1 a navíc přenos 1. Naopak s přenosem 1 se 0 změní na 1 a žádný přenos, 1 se změní 0 a přenos 2. Obdobně pokračujeme s přenosem 2, více už nebudeme potřebovat.

Vidíme tedy, že náš TS potřebuje 3 vnitřní stavy pro uchování hodnotu přenosu 0, 1 nebo 2. Výše popsaná pravidla pak již snadno převedeme do přechodů našeho TS, viz Obrázek 10.1.

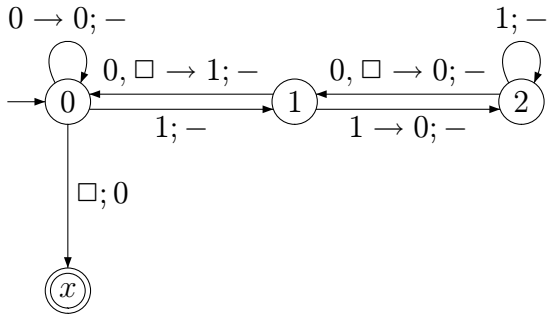
Stavy TS jsou přirozeně značeny hodnotou přenosu, který uchovávají. Všimněme si dobře, že všechny posuny hlavy jsou o -1, neboť zadané slovo čteme zprava doleva.

Je však toto všechno? Kde vlastně TS skončí svůj výpočet? Vidíme, že se

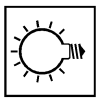


Obrázek 10.1: Turingův stroj realizující násobení třemi (začátek konstrukce)

TS stále bude pohybovat hlavou doleva, až přejde přes všechny číslice na mezery □. Kde však máme přechody stavů mezerou definovány? Nikde, takže je musíme doplnit. Možná by se zdálo, že čtení mezery by nás mělo hned převést do koncového stavu, ale uvědomme si, že ještě nejdřív před ukončením výpočtu musíme na pásku vypsat zapamatovaný přenos. Takže celý TS teď vypadá tak, jak je znázorněno na Obrázku 10.2.



Obrázek 10.2: Turingův stroj realizující násobení třemi

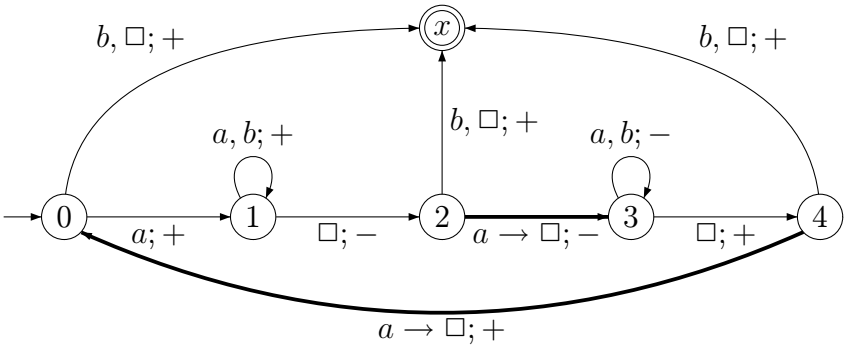


Řešený příklad 10.2: Navrhněte TS, který ze zadaného slova nad abecedou $\{a, b\}$ umaže od začátku i od konce nejdelší možné stejně dlouhé úseky znaků a. (Tj. ze slova *aaababaa* udělá *abab*, kdežto z *aaabab* neumaže nic. Ze slova *aaa* zbude ε.)

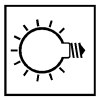
Řešení: Nejprve si problém rozebereme. Pokud slovo začíná b, můžeme hned skončit. Pokud je na začátku a, možná by se někomu chtělo jej hned umazat – přepsat na □, ale to není možné, protože jsme ještě nezkontrolovali, jestli je a i na konci slova. Proto se nejprve vždy musíme podívat i na konec, zda

tam jsou odpovídající a, od konce už ho pak můžeme umazat a od začátku a umažeme až po návratu zpět.

Další otázkou je, jak si spočítáme, kolik a je na začátku i na konci společných. Bohužel zde narazíme na podobné omezení jako u automatů – samotný TS (jeho řídící jednotka) si nemůže znaky a spočítat, protože má jen omezeně mnoho stavů. Proto budeme muset znaky a umazávat postupně a synchronizovaně.

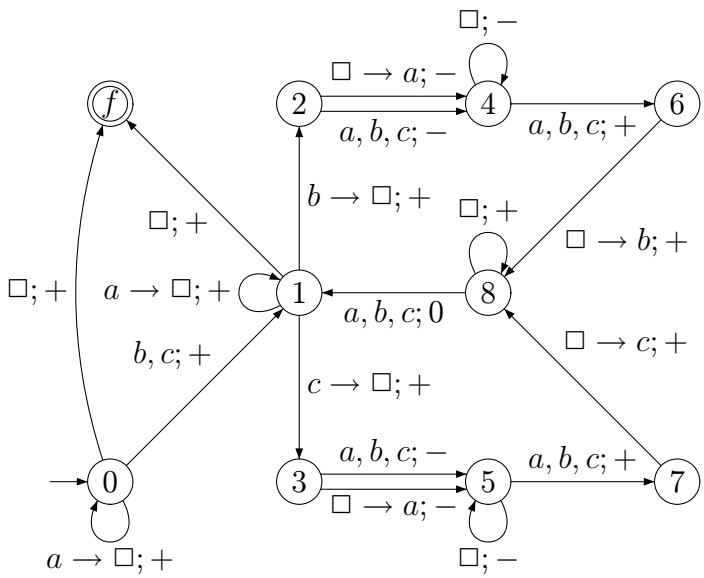


Jinými slovy, zkontrolujeme znak a na začátku, pak se přesuneme na konec, pokud tam a najdeme, umažeme jej, vrátíme se na začátek a odpovídající a také umažeme, a tak pořád dokola až do zastavení. Ty dva přechody, které umazávají znaky a, jsou v obrázku TS zvýrazněny.



Řešený příklad 10.3: Navrhněte Turingův stroj, který z daného slova nad abecedou $\{a, b, c\}$ vypustí všechny výskyty znaku a. Předpokládáme, že TS začíná výpočet na prvním znaku slova vlevo.

Řešení: Příklad se zdá jednoduchý – TS by mohl projít všechny znaky slova a znak a přepíše na □. Je to však korektní postup? Zadání přece říká, že se znaky a mají *vypustit*, ne nahradit mezerami, takže my místo pouhého přepisování znaku a musíme všechny znaky za ním posunout o jednu pozici dopředu. (Přepisování a na □ lze tedy použít jen na prvních a posledních znacích slova.) Navíc si musíme uvědomit, že po vypuštění dalších znaků a se už bude zbytek slova posouvat o více než jeden znak doleva.



Výsledný Turingův stroj je již dosti složitý, neboť musí řešit množství problematických okrajových situací. Zde uvádíme neformální slovní popis jeho činnosti:

Na začátku jsou ve stavu 0 umazávány všechny znaky a. Po prvním výskytu jiného znaku stroj přejde do stavu 1, který je vlastně centrálním stavem hlavního pracovního cyklu stroje. Při každém průchodu tímto pracovním cyklem z 1 zpět do 1 je přenesen jeden následující znak b (horní větev) či c (dolní větev) z původní pozice na novou (vlevo). Znak se přenáší přes střední úsek mezer (který může být libovolně dlouhý), což nám umožňuje znaky a prostě mazat. Přenos je konkrétně implementován tak, že znak je na původní pozici smazán, pak stroj přejde po mezerách doleva na upravený úsek slova, tam přenesený znak zpětně zapíše a po mezerách zase přejde doprava.

Všimněte si „podivných“ přechodů $2 \to 4$ a $3 \to 5$ po znaku $\square$. Proč je tam zapisován znak a? Čtení znaku $\square$ ve stavech $2, 4$ znamená, že jsme dosáhli konce slova, avšak skončit ještě nemůžeme, neboť nám zbývá zapsat předchozí smazaný znak b nebo c. Pracovní cyklus proto musíme dokončit, ale zároveň si nemůžeme dovolit nechat na pravém konci slova jen mezery, protože by pak stroj ve stavu 8 skončil v nekonečné smyčce. Proto si pomůžeme zapsáním na konec znaku a, který se pak stejně smaže.

Všimněte si, že koncové stavy Turingova stroje se chovají jinak než přijímající stavy konečného automatu. Pokud je dosažen koncový stav, výpočet skončí.

Vzhledem k tomu, že ve výše uvedené definici je Turingův stroj definován jako deterministický, pro dané vstupní slovo existuje vždy jen jen možný výpočet. Uvědomme si však dobře, že tento výpočet nemusí vždy skončit. Jestliže výpočet TS neskončí (nedojde do koncového stavu, nezastaví se), je jeho výsledek *nedefinovaný*. Podle naší definice tedy slovo není přijímáno strojem M právě tehdy, když je výpočet nad ním nekonečný.

Čtenáře asi napadne varianta definice, která by vyžadovala, aby každý výpočet TS vždy skončil, a sice buď ve speciálním *přijímajícím stavu* q_{ANO} (vstupní slovo přijato) nebo *zamítajícím stavu* q_{NE} (vstupní slovo zamítnuto). Jazyky, které je možné Turingovými stroji takto *rozhodovat* se nazývají *rekurzivní* nebo *rozhodnutelné jazyky* a tvoří vlastní podtřídu jazyků přijímaných Turingovými stroji (které jsou také nazývány *rekurzivně spočetné* či *částečně rozhodnutelné jazyky*). Důkaz bude proveden v kapitole 12, kde se touto problematikou budeme zabývat podrobněji. (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 kapitole 11. 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 následujících kapitolách, teď si jen stručně všimneme rozšíření vzniklého využitím nedeterminismu.

V základní definici je Turingův stroj deterministický, ale dá se ukázat, že i při povolení nedeterminismu získáme jen stejnou výpočetní sílu.

Cvičení 10.1: Využitím zkušeností s konečnými a zásobníkovými automaty nadefinujte pojem *nedeterministických Turingových strojů* a jazyků jimi přijímaných.

Věta 10.2
Třída jazyků přijímaných (deterministickými) Turingovými stroji se rovná třídě jazyků přijímaných nedeterministickými Turingovými stroji.

Důkaz (náznak): 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. □

10.2 Generativní gramatiky

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 10.3
Generativní gramatika G je čtveřice (Π, Σ, 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 libovolnou generativní gramatiku $G = (\Pi, \Sigma, S, P)$. Pro $\gamma, \delta \in (\Pi \cup \Sigma)^*$ 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 je G zřejmá

z kontextu), jestliže existují slova $\mu_1, \mu_2, \alpha, \beta$ taková, že $\gamma = \mu_1\alpha\mu_2$, $\delta = \mu_1\beta\mu_2$ a P obsahuje pravidlo $\alpha \to \beta$.

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 nějaké $n \geq 0$) taková, že

$$\gamma = \mu_0 \Rightarrow \mu_1 \Rightarrow \cdots \Rightarrow \mu_n = \delta\,.$$

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 $L(G_1) = L(G_2)$.

10.3 Chomského hierarchie

Obecné gramatiky můžeme rozdělit do několika typů, podle toho, jaká omezení klademe na pravidla, která se mohou v gramatice vyskytovat.

V tzv. *Chomského hierarchii* se rozlišují čtyři typy gramatik označované jako gramatiky typu 0 (obecné gramatiky), typu 1 (kontextové gramatiky), typu 2 (bezkontextové gramatiky) a typu 3 (regulární gramatiky). Jak konkrétně vypadají omezení na pravidla v jednotlivých typech gramatik je uvedeno v následující definici.

Definice 10.4
Obecná generativní gramatika $G = (\Pi, \Sigma, S, P)$ je:

- *Typu 0* neboli *obecná gramatika*, jestliže na její pravidla neklademe žádná další omezení než ta, která plynou z definice generativní gramatiky.
- *Typu 1*, neboli *kontextová gramatika*, jestliže každé pravidlo v P je tvaru
  $$\alpha X \beta \to \alpha\gamma\beta$$
  kde $\alpha, \beta, \gamma \in (\Pi \cup \Sigma)^*$, $X \in \Pi$ a $|\gamma| \geq 1$.

Takto definovaná gramatika by však neumožňovala odvodit slovo ε. Proto je jako speciální výjimka povoleno, že P může obsahovat pravidlo $S \to \varepsilon$. Pokud však P obsahuje toto pravidlo, nesmí se neterminál S vyskytovat na pravé straně žádného pravidla.

- *Typu 2*, neboli bezkontextová gramatika, jestliže každé pravidlo v P je tvaru
$$X \to \alpha$$
- *Typu 3*, neboli *regulární gramatika*, jestliže každé pravidlo v P je tvaru
$$X \to wY \qquad \text{nebo} \qquad X \to w$$
kde $w \in \Sigma^*$.

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 a gramatika typu 1 je speciálním 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). Je tedy zjevné, že platí následující tvrzení.

Tvrzení 10.5
Nechť $\mathcal{L}_i$ je třída jazyků typu i ($i = 0, 1, 2, 3$). Platí
$$\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_0$$

Dá se ukázat, že jednotlivé třídy jazyků v Chomského hierarchii přesně odpovídají třídám jazyků, které jsou rozpoznávány určitými typy automatů:

- Jazyky typu 0 jsou rozpoznávány *Turingovými stroji*.
- Jazyky typu 1 jsou rozpoznávány *lineárně omezenými automaty*.
- Jazyky typu 2 jsou rozpoznávány *nedeterministickými zásobníkovými automaty*.
- Jazyky typu 3 jsou rozpoznávány *konečnými automaty*.

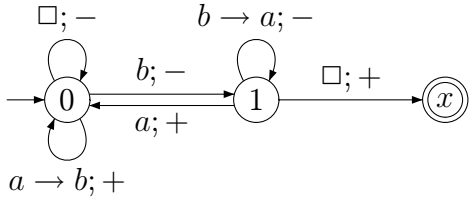
10.4 Cvičení

Otázky:

Otázka 10.2: Jak velký úsek pásky může TS při svém výpočtu použít?

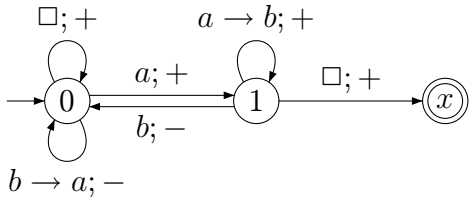
Cvičení 10.3:Navrhněte Turingův stroj, který rozpoznává palindromy, tj. stroj se zastaví právě tehdy, když se zadané slovo čte stejně od začátku jako od konce.

Cvičení 10.4: Popište slovně, na jakých slovech se zastaví výpočet následujícího Turingova stroje a co se stane s daným vstupem. Stroj začíná výpočet s hlavou na prvním znaku zleva.



Cvičení 10.5*:Popište slovně, na jakých slovech se zastaví výpočet Turingova stroje ze Cvičení 10.4 a co se stane s daným vstupem. Stroj nyní začíná výpočet s hlavou na prvním znaku zprava.

Cvičení 10.6*:Popište slovně, na jakých slovech se zastaví výpočet následujícího Turingova stroje a co se stane s daným vstupem. Stroj začíná výpočet s hlavou na prvním znaku zleva.



Cvičení 10.7*:Navrhněte jednopáskový Turingův stroj, který dané číslo zapsané v binární soustavě vydělí třemi. Začíná se na slově vlevo.

Návod: Vzpomeňte si na klasický školní algoritmus dělení čísel a postupujte přesně podle něj.

Cvičení 10.8:Navrhněte jednopáskový Turingův stroj, který pracuje s (páskovou) abecedou $\{a, b, c, \Box\}$ a který vykonává následující výpočet:

Na začátku je na pásce napsáno libovolné slovo $w \in \{a, b\}^*$ a zbytek pásky je vyplněn symboly $\Box$. Hlava stroje je na prvním znaku slova w. Váš Turingův stroj musí vždy skončit výpočet a po skončení musí mít někde na pásce napsáno slovo $\underbrace{c \dots c}_{k}$, kde k je počet přechodů mezi písmeny a, b (v obou směrech, tj. počítáte jak přechod $\dots ab \dots$, tak i $\dots ba \dots$) v původním slově w. Zbytek pásky musí být opět vyplněn symboly $\Box$.

Návod: Zhruba řečeno, výpočet vašeho stroje musí ve slově w spočítat všechny změny znaků z a na b i z b na a a výsledek "zapsat" počtem znaků c. Například pro aaa je výsledek ε, pro $aaab$ je výsledek c, pro $ababa$ je výsledek $cccc$, pro $aabbbbaabbbba$ je také $cccc$.

Cvičení 10.9: Navrhněte TS, který zadané slovo nad abecedou $\{0, 1\}$ invertuje, tj. nuly přepíše na jedničky a naopak.

Cvičení 10.10: Jak byste upravili TS z Řešeného příkladu 10.1, aby začínal výpočet na prvním znaku vstupu (zleva)?

Cvičení 10.11: Navrhněte TS, který číslo zadané ternárně nad abecedou $\{0, 1, 2\}$ vynásobí dvěma.

Kapitola 11

Problémy, algoritmy a výpočetní modely

Cíle kapitoly:

- Pochopení pojmu (algoritmický) problém.
- Seznámení se s možnými způsoby kódování vstupů a výstupů.
- Pochopení programování na výpočetním modelu RAM.
- Hlubší pochopení pojmu algoritmus (Churchova-Turingova teze).

V druhé polovině předmětu se budeme věnovat základům teorie algoritmické složitosti. Položme si nejprve otázku: Co je to vlastně „*algoritmus*“?

Pohledy na algoritmy se vyvíjejí už od dávné minulosti (vzpomeňme třeba netriviální Euklidův algoritmus pro nalezení největšího společného dělitele dvou čísel). Tradičně byl algoritmus vnímán jako posloupnost slovně popsaných jednoznačných kroků. Teprve s nástupem výpočetních strojů ve 20. století přišla potřeba přesně definovat pojem algoritmu, který se na nových strojích dá provádět.

Čtenář má jistě určitou intuitivní představu o tom, co to „algoritmus“. Pokud bychom ale chtěli tento pojem nějak podrobněji popsat, asi bychom používali slova a slovní spojení jako „postup“, „návod“, „posloupnost elementárních

kroků“ apod. a asi bychom požadovali, aby ho bylo možné provádět *mechanicky* (ať už to znamená cokoli), a aby jeho jednotlivé kroky byly konečné (finitní).

Kdybychom ale chtěli pojem „algoritmus“ definovat matematicky přesně pomocí nějakých jednodušších základnějších pojmů, asi se nám to nepodaří. Pojem *algoritmus* je podobně jako pojem *množina* základním pojmem, který není definován pomocí jednodušších pojmů.

Místo toho abychom se ptali, co je to algoritmus, zkusíme se zeptat trochu jinak: K čemu vlastně algoritmy slouží? Odpověď zní: K řešení problémů.

Slovo „problém“ má však v přirozeném jazyce hodně významů; jaký konkrétní druh problémů máme na mysli? Příklady problémů, kterými se budeme dále zabývat jsou například problémy typu sečíst dvě čísla, nalézt nejkratší cestu v grafu, zjistit, zda je dané číslo prvočíslem, vynásobit dvě matice apod., tj. problémy, které se dají přesně formulovat pomocí matematických pojmů, a u kterých má rozumný smysl uvažovat o tom, že k jejich řešení použijeme počítač.

Podrobnější definice pojmu „problém“ je uvedena v následující sekci.

11.1 Definice pojmu „problém“

Nyní přejdeme k formální definici algoritmického problému. Tato definice je nutná, abychom si sjednotili různé možné praktické pohledy na způsoby zadání vstupů a výpisu výsledků algoritmů.

Jak již bylo řečeno, algoritmy chápeme jako návody na řešení určitých problémů. Problémy, kterými se budeme zabývat, budeme většinou zadávat následujícím schématem:

NÁZEV: *XY*

VSTUP: Zde je popsáno, co je přípustným vstupem (zadáním, instancí) našeho problému.

VÝSTUP: Zde je popsáno, jaký výstup (výsledek) je očekáván pro zadaný vstup (je přiřazen zadanému vstupu).

Problém sečíst dvě přirozená čísla zadaný tímto schématem vypadá takto:

NÁZEV: *Součet*

VSTUP: Dvojice přirozených čísel x a y.

VÝSTUP: Přirozené číslo z takové, že $z = x + y$.

Jiným příkladem problému je problém nalezení nejkratší cesty v grafu:

NÁZEV: *Nejkratší cesta v grafu*

VSTUP: Orientovaný graf $G = (V, E)$ a dvojice vrcholů $u, v \in V$.

VÝSTUP: Nejkratší cesta z u do v, tj. nejkratší sekvence $v_0, v_1, \ldots, v_k$, kde $v_i \in V$, taková, že $v_0 = u$, $v_k = v$ a $(v_{i-1}, v_i) \in E$ pro $\forall i \in \{1, \ldots, k\}$, případně prázdná sekvence, pokud žádná cesta z u do v neexistuje.

Pokud chceme napsat formální definici pojmu problém, mohla by vypadat takto:

Definice 11.1
Problém je určen trojicí (IN, OUT, p), kde IN je množina (přípustných) *vstupů*, OUT je množina *výstupů* a $p : IN \to OUT$ je funkce přiřazující každému vstupu odpovídající výstup.

Takto definovaný problém se někdy též nazývá „výpočetní problém“ (computational problem). Algoritmus *řeší* daný problém (IN, OUT, p), jestliže pro *libovolný* vstup x z množiny IN vyprodukuje po konečném počtu kroků výstup y (z množiny OUT) takový, že $y = p(x)$.

Poznámka: Někdy má dobrý smysl uvažovat i problémy, kde pro jeden vstup může existovat více správných výstupů a po algoritmu, který by tento problém řešil, chceme, aby našel (alespoň) jeden z nich. Příkladem takového problému je třeba výše uvedený problém hledání nejkratší cesty v grafu (je zřejmé, že mezi dvojicí vrcholů může existovat více než jedna nejkratší cesta). Alternativně jsem mohli pojem „problém“ definovat definovat poněkud obecněji jako trojici (IN, OUT, P), kde význam IN a OUT je stejný jako v předchozím případě a $P \subseteq IN \times OUT$ je relace, která musí splňovat, že ke každému x z IN musí existovat alespoň jedno y z OUT takové, že $(x, y) \in P$. Intuitivně $(x, y) \in P$ znamená, že y je korektní výstup pro vstup x.

11.2 Kódování vstupů a výstupů

V předchozí definici pojmu „problém“ jsme nedefinovali, co jsou prvky množin IN a OUT. Vzhledem k tomu, že jednotlivé kroky algoritmu by měly být finitní operace a vzhledem k tomu, že algoritmus by měl při práci nad daným vstupem vykonat pouze konečný počet takovýchto kroků, je zřejmé, že i vstupy a výstupy by měly být konečné (finitní) objekty.

Neformálně bychom mohli říct, že se musí jednat o objekty, které jsou nějak reprezentovatelné konečným způsobem.

Konkrétní příklady toho, co mohou být prvky množin IN a OUT jsou:

- slova v nějaké dané abecedě Σ,
- slova v abecedě $\{0, 1\}$, tj. sekvence bitů,
- sekvence celých čísel,
- přirozená čísla.

Podle potřeby můžeme zvolit kteroukoliv z těchto možností. Ve skutečnosti příliš nezáleží na tom, kterou možnost zvolíme, protože tyto různé reprezentace můžeme snadno převádět jednu na druhou:

- Slova libovolné abecedy Σ je možné reprezentovat jako sekvence přirozených čísel:

  Stačí očíslovat symboly abecedy.

  Např. pro $\Sigma = \{a, b, c, d\}$ můžeme znaku a přiřadit číslo 0, znaku b číslo 1, znaku c číslo 2 a znaku d číslo 3. Slovo $bddaba$ pak bude reprezentováno jako posloupnost $1, 3, 3, 0, 1, 0$.

- Slova libovolné abecedy Σ je možné reprezentovat jako slova v abecedě $\{0, 1\}$:

  Slova převedeme na sekvence čísel jako v předchozím případě, přičemž čísla zapíšeme binárně jako k-bitová čísla, přičemž k musí být zvoleno dostatečně velké, aby bylo možné reprezentovat všechny symboly abecedy.

Např. pro $\Sigma = \{a, b, c, d\}$ můžeme znak a reprezentovat jako 00, znak b jako 01, znak c jako 10 a znak d jako 11. Slovo *bddaba* pak zapíšeme jako 011111000100.

- Přirozená čísla je možno zapisovat jako slova v abecedě $\{0, 1\}$:

  Stačí použít binární zápis čísla. Například číslo 22 je možné zapsat jako `10110`.

- Sekvence celých čísel je možné reprezentovat jako slova ve vhodně zvolené abecedě Σ:

  Například můžeme čísla zapisovat binárně, pro označení záporných čísel používat znak `-` a pro oddělení jednotlivých čísel používat znak `#`. V tomto případě je tedy $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{-}, \texttt{\#}\}$, a například sekvence

  $$4, -6, 3, 0, 13, -5$$

  je pak reprezentována slovem

  ```
  100#-110#11#0#1101#-101
  ```

- Slova v abecedě $\{0, 1\}$ je možné reprezentovat přirozenými čísly:

  Na začátek slova přidáme symbol 1 a výsledné slovo chápeme jako zápis čísla ve dvojkové soustavě. Například slovu 0110 přiřadíme číslo 22 (tj. 10110 binárně).

  Pozn.: Přidání symbolu 1 na začátek je třeba, abychom byli schopni rozlišit slova, která se liší jen počtem nul na začátku, např. slova 1, 01, 001 atd.

Podotkněme, že výše popsané transformace samozřejmě nejsou jediné možné.

Cvičení 11.1: Pro každý z výše uvedených převodů jedné reprezentace na druhou uveďte nějaký alternativní způsob, jak by bylo možné onen převod provést.

Další typy objektů pak můžeme reprezentovat pomocí výše popsaných. Pokud je například vstupem matice čísel, je možné ji reprezentovat jako slovo v nějaké abecedě, přičemž jednotlivé řádku budou zapsány za sebou, odděleny nějakým speciální oddělovacím znakem, a každý jednotlivý řádek bude

reprezentován podobným způsobem, jaký jsme použili pro reprezentaci sekvence čísel.

Například grafy můžeme reprezentovat jako sekvence tvořené seznamem vrcholů a seznamem hra, případně pomocí incidenční matice. Logické formule můžeme reprezentovat jako slova v nějaké vhodně zvolené abecedě apod.

11.3 Důležité typy problémů

Speciálním případem problémů jsou tzv.

rozhodovací problémy, neboli *ANO/NE problémy*.

U takového problému je množina OUT dvouprvková; standardně pak předpokládáme, že $OUT = \{\text{Ano}, \text{Ne}\}$ (či $OUT = \{1, 0\}$).

Příkladem takového problému je například problém prvočíselnosti:

Název: *Prvočíselnost*

Vstup: Přirozené číslo x.

Výstup: Ano pokud je x prvočíslo, Ne pokud x není prvočíslo.

U takových problému je při jejich definici pohodlnější definovat, co je výstupem, pomocí otázky, na kterou je odpověď buď Ano nebo Ne:

Název: *Prvočíselnost*

Vstup: Přirozené číslo x.

Otázka: Je x prvočíslo?

Obecně tedy budeme pro rozhodovací problémy používat následující schéma:

Název: *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.

Další důležitou třídou problémů jsou *optimalizační* problémy. U optimalizačního problému je pro každý vstup určena množina *přípustných řešení* a dále je definována určitá kriteriální funkce, která každému přípustnému řešení přiřazuje nějaké reálné číslo. Cílem je mezi všemi přípustnými řešeními pro daný vstup nelézt to, pro které je hodnota kriteriální funkce největší, nebo případně nejmenší, v závislosti na typu řešeného problému.

Příkladem optimalizačního problému je již dříve uvedený problém hledání nejkratší cesty v grafu. V tomto případě je množinou přípustných řešení množina všech cest mezi dvě danými vrcholy, kriteriální funkcí je délka dané cesty a cílem je hodnotu kriteriální funkce minimalizovat.

Jiným příkladem je problém nalezení minimální kostry v grafu:

NÁZEV: *Minimální kostra*

VSTUP: Neorientovaný souvislý graf $G = (V_G, E_G)$, ohodnocení hran $f : E \to \mathbb{N}_+$.

VÝSTUP: Souvislý graf $H = (V_H, E_H)$, kde $V_H = V_G$ a $E_H \subseteq E_G$, a kde hodnota $\sum_{e \in E_H} f(e)$ je minimální.

V tomto případě je množinou všech přípustných řešení množina všech souvislých podgrafů H grafu G takových, že $V_H = V_G$. Hodnota kriteriální funkce pro dané přípustné řešení H je dána výrazem $\sum_{e \in E_H} f(e)$. Opět je cílem minimalizovat tuto hodnotu.

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 existuje kostra s ohodnocením nejvýše rovným c, a NE v opačném případě:

NÁZEV: *Minimální kostra (ANO/NE verze)*

VSTUP: Neorientovaný souvislý graf $G = (V_G, E_G)$, ohodnocení hran $f : E \to \mathbb{N}_+$ a 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 $\sum_{e \in E_H} f(e) \leq c$?

Uveďme si ještě jeden ANO/NE problém, který bude hrát důležitou roli v dalším výkladu.

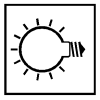
NÁZEV: SAT (*problém splnitelnosti booleovských formulí*)

VSTUP: Booleovská formule v konjunktivní normální formě.

OTÁZKA: Je daná formule splnitelná (tj. existuje pravdivostní ohodnocení proměnných, při kterém je formule pravdivá)?

Poznámka: Každý výpočetní problém lze rozdělit na konečnou (ale neomezenou) posloupnost rozhodovacích problémů: Představme si pro jednoduchost abecedu $\Sigma = \{0, 1\}$ a ptejme se rozhodovacím způsobem na jednotlivé bity výstupního slova a na ukončení výstupu.

V teorii výpočetní složitosti se povětšinou, pro svou jednodušší formu, uvažují rozhodovací problémy, ale jak vidíme z předchozí poznámky, neznamená to vážnou újmu na obecnosti zkoumání.



ŘEŠENÝ PŘÍKLAD 11.1: Ukažte si, jak problém výpočtu funkce $\sin x$ s přesností na k desetinných míst pro dané k lze rozložit na posloupnost rozhodovacích problémů.

Řešení: Nejprve se zeptáme na odpověď (rozhodovacího) problému ($\sin x > 0$). Například pokud ANO, zeptáme se na ($\sin x < \frac{1}{2}$). Řekněme, že NE, a zeptáme se na ($\sin x < \frac{3}{4}$). Takto dále postupujeme metodou půlení intervalu, až jsme spokojeni s přesností dosaženého odhadu výsledku. (Vždyť výsledek $\sin x$ stejně nelze obecně přesně vypočítat, jen přibližně. Pokud chceme výsledek na 9 desetinných míst, stačí zhruba 30 dotazů.)

11.4 Výpočetní modely

Pokud chceme nějak přesně definovat, co je to algoritmus, je vhodné nejprve zavést nějaký *výpočetní model*. Jako výpočetní model bychom mohli zvolit například některý existující programovací jazyk, případně si definovat nějaký další vlastní. Je poměrně překvapivé, že příliš nezáleží na tom, který programovací jazyk bychom zvolili, neboť se ukazuje, že jakýkoliv algoritmus, který

je možné naprogramovat v jednom programovacím jazyce je možné naprogramovat i v jiném programovacím jazyce (i když třeba ne stejně snadno) – stačí si uvědomit, že i když je program naprogramován v nějakém vyšším programovacím jazyce, stejně se při jeho běhu provádějí instrukce na úrovni strojového jazyka daného procesoru, takže v principu jsme mohli původní program místo toho zapsat na úrovni těchto instrukcí.

Programovací jazyky tedy nepochybně splňují první podmínku, avšak popsat přesně syntaxi a sémantiku nějakého programovacího jazyka není až tak triviální. Proto se v teoretické informatice často používají daleko jednodušší výpočetní modely, které ovšem kupodivu rovněž umožňují popsal libovolný algoritmus.

S jedním takovým výpočetním modelem už jsme se setkali. Je to *Turingův stroj*, který tradičně používán jako výpočetní model. V této kapitole se seznámíme s dalším často používaným výpočetním modelem, tzv. strojem RAM (Random Access Machine), který je svou podobou mnohem bližší skutečným počítačům. Ukážeme, že oba tyto modely jsou výpočetně ekvivalentní.

V souvislosti s tím popíšeme tzv. *Church–Turingovu tezi*, která je všeobecně přijímána jako axiomatická „definice“ algoritmu. V následující kapitole si pak ukážeme, že ne všechny problémy lze algoritmicky řešit.

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.

Definice 11.2
Turingův stroj M *počítá* (částečnou) funkci $P_M : L(M) \to \Gamma^*$, kde $P_M(w)$ je slovo bez mezer, které zůstane na pracovní pásce TS M po jeho zastavení na vstupním slově w.

Definice 11.3
Nechť $P : \Sigma^* \to \Sigma^*$ je problém. Říkáme, že Turingův stroj M *řeší problém* P, jestliže $L(M) = \Sigma^*$ (výpočet *vždy skončí*) a $P_M \equiv P$.

Další model počítače, který si nyní uvedeme, se již velmi blíží skutečné hardwarové konstrukci dnešních počítačů. Dá se říci, že se jedná o jednoduchou abstrakci reálného procesoru s jeho strojovým kódem, pracujícího nad lineární pamětí. (Jakožto v teoretickém modelu se zde vůbec nezabýváme periferiemi.) Tento model se nazývá *stroj RAM* (Random Access Machine), česky je někdy nazýván *„počítač s libovolným přístupem“*; název není zcela výstižný, znamená prostě to, že v jednom kroku je možný přístup k libovolné buňce paměti.

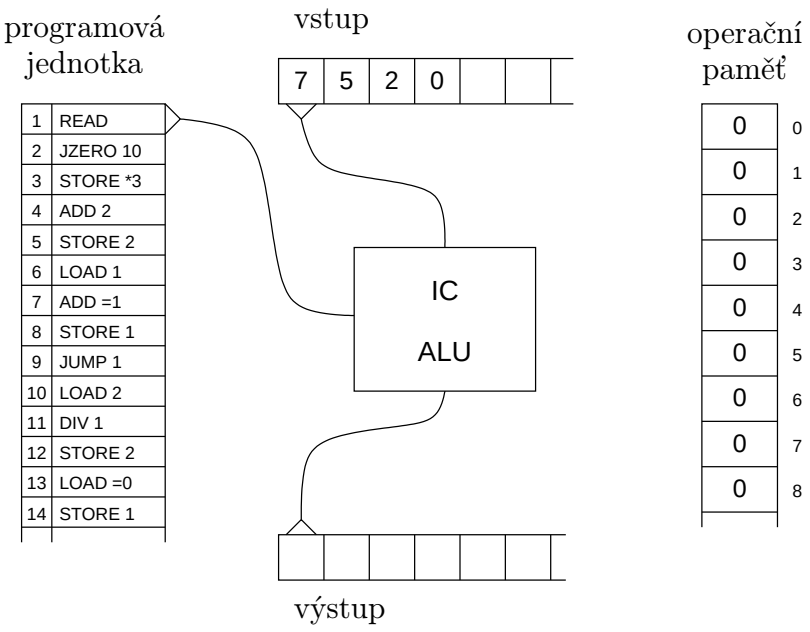
Na rozdíl od Turingova stroje nepracuje stroj RAM se slovy, ale jeho vstupy a výstupy jsou posloupnosti celých čísel, tj. v případě stroje RAM $IN, OUT \subseteq \mathbb{Z}^*$ (kde $\mathbb{Z}$ označuje množinu všech celých čísel).

RAM (Random Access Machine), neboli *počítač s libovolným přístupem*, se skládá z těchto částí (viz Obrázek 11.1):

- Programová jednotka, ve které se provádějí jednotlivé instrukce, a ve které je uložen program stroje RAM, 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).
- Neomezená pracovní paměť tvořená buňkami, kde každá buňka může obsahovat libovolné celé číslo. Buňky jsou očíslovány přirozenými čísly $0, 1, 2, \ldots;$. Číslo buňky se nazývá adresa buňky. Do buněk je možno zapisovat i z nich číst.
- Vstupní páska tvořena buňkami, kde každá buňka obsahuje jedno celé číslo. Z této pásky je možno pouze sekvenčně číst. Na aktuální políčko ukazuje hlava. Základní krok v činnosti hlavy spočívá v přečtení obsahu snímaného políčka a posunutí doprava o jedno políčko.
- Výstupní páska, do jejíchž buněk se zapisují celá čísla. Na tuto pásku je pouze možné sekvenčně zapisovat.

Buňky s adresou 0 a 1 mají zvláštní postavení a nazývají se *pracovní registr* (buňka 0) a *indexový registr* (buňka 1).

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).



Obrázek 11.1: Stroj RAM

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 na Obrázku 11.2.)

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 indexového registru

Instrukce vstupu a výstupu (jsou bez operandu):

zápis	význam
`READ`	do pracovního 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 pracovního registru a posune se o jedno políčko doprava

Instrukce přesunu v paměti:

zápis	význam
`LOAD` *op*	do pracovního registru se načte hodnota operandu
`STORE` *op*	hodnota operandu se přepíše obsahem pracovního registru (zde se nepřipouští operand tvaru =i)

Instrukce aritmetických operací:

zápis	význam
`ADD` *op*	číslo v pracovním registru se zvýší o hodnotu operandu (tedy přičte se k němu hodnota operandu)
`SUB` *op*	od čísla v pracovním registru se odečte hodnota operandu
`MUL` *op*	číslo v pracovním registru se vynásobí hodnotou operandu
`DIV` *op*	číslo v pracovním registru se „celočíselně“ vydělí hodnotou operandu (do pracovního registru se uloží výsledek příslušného celočíselného dělení)

Instrukce skoku:

zápis	význam
`JUMP` *návěští*	výpočet bude pokračovat instrukcí určenou návěštím
`JZERO` *návěští*	je-li obsahem pracovního registru číslo 0, bude výpočet pokračovat instrukcí určenou návěštím; v opačném případě bude pokračovat následující instrukcí
`JGTZ` *návěští*	je-li číslo v pracovním 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, programový čítač ukazuje „mimo program“, adresa při použití operandu $*i$ vyjde záporná), výpočet se („neregulérně“) zastaví.

RAM M řeší problém $P = (IN, OUT, p)$, kde $IN, OUT \subseteq \mathbb{Z}^*$, jestliže má tuto vlastnost:
začne-li výpočet v počáteční konfiguraci se vstupem $c_1c_2 \ldots c_n \in IN$, pak svůj výpočet (regulérně) skončí, přičemž na výstupu je $p(c_1c_2 \ldots c_n)$.

Na Obrázku 11.2 je příklad programu pro stroj RAM, který řeší následující problém:

VSTUP: Neprázdná posloupnost kladných celých čísel ukončená nulou.

VÝSTUP: Odchylky jednotlivých čísel od aritmetického průměru zadané posloupnosti zaokrouhleného dolů.

Další příklad programu pro stroj RAM najdete v kapitole 13.

Poznámka: Kromě výrazů „stroj RAM“ či „RAM-stroj“ budeme užívat jen zkratku „RAM“; budeme např. mluvit o sestrojení RAMu apod.

```
 1  READ
 2  JZERO 10
 3  STORE *3
 4  ADD 2
 5  STORE 2
 6  LOAD 1
 7  ADD =1
 8  STORE 1
 9  JUMP 1
10  LOAD 2
11  DIV 1
12  STORE 2
13  LOAD =0
14  STORE 1
15  LOAD *3
16  JZERO 23
17  SUB 2
18  WRITE
19  LOAD 1
20  ADD =1
21  STORE 1
22  JUMP 15
23  HALT
```

Obrázek 11.2: Příklad programu pro stroj RAM

Poznámka: Ve skutečných programovacích úlohách obvykle nebudeme rozepisovat programy až do jednotlivých instrukcí stroje RAM, ale vystačíme si se strukturovaným popisem algoritmu (jako ve vyšších programovacích jazycích). Musíme si však umět představit, jak by takovéto rozepsání instrukcí mělo vypadat.

Stroj RAM a Turingův stroj jsou vzájemně ekvivalentní v tom smyslu, že je možné jeden simulovat pomocí druhého.

Cvičení 11.2: Promyslete si, jak je možné Turingův stroj simulovat pomocí stroje RAM, a naopak, jak je možné stroj RAM simulovat pomocí Turingova stroje. Jak se při těchto simulacích změní celkový počet provedených

instrukcí?

11.5 Churchova-Turingova teze

Viděli jsme, že Turingovy stroje a stroje RAM jsou ekvivalentní v tom smyslu, že pokud je nějaký problém řešitelný jedním z těchto modelů, pak je řešitelný i druhým.

Podobně se ukázat ekvivalence těchto dvou modelů s celou řadou dalších modelů ať už teoretických (jako jsou například λ-kalkulus nebo tzv. rekurzivní funkce, kterými se zde ovšem nebudeme dále zabývat) či praktických (všechny obecné programovací jazyky).

Doposud u každého modelu, který byl navržen jako formalizace pojmu „algoritmus“, se vždy dá ukázat, že je ekvivalentní Turingovým strojům. Toto ospravedlňuje obecné přesvědčení, že Turingův stroj (či libovolný model, který je s ním ekvivalentní) je vhodnou formalizací pojmu algoritmus. Předpokládá se, že ani není možné navrhnout žádný model, který by odpovídal intuitivní představě pojmu algoritmus a přitom by jej nebylo možno simulovat Turingovým strojem.

Toto přesvědčení je formulováno jako tzv. *Churchova-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.*

Není to věta v matematickém slova smyslu, kterou by bylo možno dokázat či vyvrátit. K tomu by bylo třeba nejprve pojem „algoritmus“ nějak definovat. Místo toho je Churchova-Turingova teze přijímána jako definice pojmu „algoritmus“, jako axiom. Pojem „algoritmus“ bereme tedy jako pojem základní (podobně jako např. pojem „množina“) a nikoli odvozený (tj. definovaný pomocí základních a z nich odvozených pojmů).

Všimněte si ještě, že v obráceném směru je platnost teze zřejmá. Tyto úvahy jsou už spíše logicko-filozofické povahy a zde je nebudeme dále rozebírat.

11.6 Cvičení

Otázky:

Otázka 11.3: Jak byste na stroji RAM implementovali jednorozměrné pole o délce k?

Otázka 11.4: Jak byste na stroji RAM implementovali dvourozměrné pole $k \times \ell$?

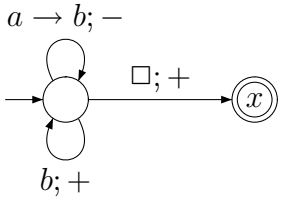
Otázka 11.5: Jakým způsobem může stroj RAM vykonávat rekurzivní programy, tj. takové, kde nějaká funkce rekurzivně mnohokrát volá sama sebe?

Cvičení 11.6: Předpokládejme, že máme dán nějaký vícepáskový Turingův stroj. Ukažte, jak činnost tohoto stroje simulovat pomocí Turingova stroje s jednou páskou.

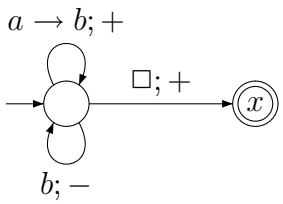
Cvičení 11.7:Předpokládejme, že máme dán nějaký konkrétní stroj RAM. Popište podrobně, jak vytvořit (vícepáskový) Turingův stroj, který by simuloval činnost tohoto stroje RAM.

Pokud víme, že stroj RAM vykoná pro nějaký konkrétní vstup t kroků, co můžeme říct o počtu kroků, které by pro tento vstup vykonal vámi navržený Turingův stroj?

Cvičení 11.8: Zjistěte, kolik přesně kroků provede níže zakreslený Turingův stroj v závislosti na daném slově w nad abecedou $\{a, b\}$. (Slovo w je na začátku napsáno na pásku a vše ostatní je vyplněno $\Box$. Hlava stroje začíná na prvním znaku w zleva.)



Cvičení 11.9: Zjistěte, kolik přesně kroků provede níže zakreslený Turingův stroj v závislosti na daném slově w nad abecedou $\{a, b\}$.


$a \to b; +$
$\square; +$
x
$b; -$

Kapitola 12

Rozhodnutelné a nerozhodnutelné problémy

Cíle kapitoly:

- Pochopení pojmu rozhodnutelného, nerozhodnutelného a částečně rozhodnutelného problému.

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 12.1
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 přirozených čísel, jazyky, či funkce:

Definice 12.2

- Množina $M \subseteq \mathbb{N}$ je *rozhodnutelná*, jestliže problém příslušnosti k M (Vstup: $n \in \mathbb{N}$; Otázka: platí $n \in M$?) 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 $f : \mathbb{N} \to \mathbb{N}$ je *algoritmicky* (někdy též *efektivně*) *vyčíslitelná*, jestliže „problém výpočtu jejích hodnot“ (Vstup: $n \in \mathbb{N}$; 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 12.3
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.

Při přijetí Church-Turingovy teze lze ztotožňovat pojmy *rekurzivní* a *rozhodnutelný* (v případě (totální) funkce pojmy *rekurzivní* a *algoritmicky vyčíslitelná*).

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 12.4
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.

Určitý vztah mezi rozhodnutelností a částečnou rozhodnutelností uvádí následující věta (zformulujte si ji i pro ANO/NE problémy):

Věta 12.5 (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“. □

12.1 Nerozhodnutelné problémy

Ne všechny (formální) problémy jsou algoritmicky řešitelné, jak lze snadno nahlédnout z toho, že algoritmů je jen spočetně mnoho, kdežto všech problémů je nespočetně mnoho. Jak však algoritmicky neřešitelné problémy vypadají konkrétně?

Komentář: Vzpomeňte si na klasickou logickou hádanku, kde v malém městečku působí holič, který holí právě všechny ty muže, kteří se neholí sami. Holí se náš holič nebo ne?

Místo holiče si představme stroj, který na vstupu dostává popisy algoritmů, a tento stroj se zastaví právě tehdy, když algoritmus daný na vstupu se nikdy nezastaví. Co náš stroj udělá se vstupem, který algoritmicky popisuje jeho sama?

Ukážeme si nerozhodnutelnost následujícího problému:

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 12.6
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 speciálním 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 a z Postovy věty:

Věta 12.7
Problém HP je částečně rozhodnutelný, jeho doplňkový problém není (ani) částečně rozhodnutelný.

CVIČENÍ 12.1: Jsou rekurzivně spočetné jazyky, tj. ty přijímané zastavením nějakého Turingova stroje, uzavřené na doplněk?

CVIČENÍ 12.2: Lze algoritmicky poznat, zda daný program dělá přesně to, co by měl? (Problém automatizovaného testování softwaru.)

Kapitola 13

Výpočetní složitost, analýza algoritmů

Cíle kapitoly:

- Pochopení pojmu složitosti algoritmu včetně role referenčního modelu počítače.
- Porozumění analýze složitosti (jednoduchých) programů pro RAM.
- Seznámení se s asymptotickou notací používanou pro odhady rychlosti růstu funkcí.

V předchozích kapitolách jsme se zabývali tím, co je to algoritmus, co je to problém, co to znamená, že algoritmus řeší daný problém apod. Jeden a tentýž problém může být řešen řadou různých algoritmů. Pokud by počítače pracovaly nekonečně rychle, příliš by nezáleželo na tom, jaký konkrétní algoritmus použijeme, stačilo by, že by korektně řešil daný problém. Tak tomu ovšem není. Počítače sice pracují rychle, ale ne nekonečně rychle, provedení každé instrukce trvá nějakou (i když velmi krátkou) dobu. (Pozn.: Současné běžné počítače provádějí řádově miliardy operací za sekundu.) Mezi možnými algoritmy, které řeší daný problém tedy chceme vybrat takový, který ho řeší nejrychleji. Přirozenou otázkou je, jak tedy máme algoritmy porovnávat a jak určit, jak „rychlý“ je daný algoritmus.

Tak jako v předchozí teorii byl historicky daným základním modelem výpočtu Turingův stroj, pro modelování složitosti algoritmu se používá stroj RAM, který je velmi blízký skutečným počítačům (procesorům).

Hodláme zde ukázat, jak se teoreticky měří časová složitost jednotlivých algoritmů (tj. kolik náš výpočet trvá) a také složitost problémů (tj. jak dlouho řešení problému musí trvat). Náš způsob měření složitosti algoritmů je postavený na asymptotických odhadech funkce času výpočtu, a je tudíž nezávislý na konkrétní implementaci algoritmu a rychlosti našich počítačů. Konečným výsledkem je pak zavedení tzv. třídy PTIME všech efektivně řešitelných problémů.

Čtenář už jistě má určitou představu o tom, že například 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: *Třídění čísel*

VSTUP: Konečná posloupnost přirozených čísel.

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 (pole A obsahuje členy vstupní posloupnosti, které označujeme $A[1], A[2], \ldots, A[n]$):

BUBBLESORT(A, n)
while *Nesetříděno*
do for $i \leftarrow 1$ **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
$n \leftarrow 0$
repeat
$n \leftarrow n + 1$; READ$(A[n])$
until $A[n] = 0$
$n \leftarrow n - 1$
▷ v n je uložen počet členů vstupní posloupnosti
for $j \leftarrow 1$ **to** $n - 1$
do for $i \leftarrow 1$ **to** $n - j$
do if $A[i] > A[i + 1]$
then $pom \leftarrow A[i]$; $A[i] \leftarrow A[i + 1]$; $A[i + 1] \leftarrow pom$
▷ výsledná seřazená posloupnost se vypíše
for $i \leftarrow 1$ **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

```
▷ pole H představuje haldu
▷ kon udává aktuální koncový index haldy
kon ← 0    ▷ halda je prázdná
READ(clen)
while clen ≠ 0
      do ZARAD-DO-HALDY(clen)
         READ(clen)
while kon > 0        ▷ halda není prázdná
      do clen ← VYDEJ-MIN-Z-HALDY()
         WRITE(clen)
```

ZARAD-DO-HALDY(k)

```
kon ← kon +1; H[kon] ← k; p ← kon
while p > 1 and H[⌊p/2⌋] > H[p]
      do prohoď H[⌊p/2⌋] a H[p];
         p ← ⌊p/2⌋
```

```
VYDEJ-MIN-Z-HALDY()
    min ← H[1]
    if kon > 1
       then H[1] ← H[kon]
    kon ← kon −1
    p ← 1
    while 2 ∗ p + 1 ≤ kon and (H[p] > H[2 ∗ p] or H[p] > H[2 ∗ p + 1])
          do if H[2 ∗ p] ≤ H[2 ∗ p + 1]
                then prohoď H[p] a H[2 ∗ p]; p ← 2 ∗ p
                else prohoď H[p] a H[2 ∗ p + 1]; p ← 2 ∗ p + 1
    if 2 ∗ p = kon and H[p] > H[2 ∗ p]
       then prohoď H[p] a H[2 ∗ p]
    return min
```

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“ al-

goritmu 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 vyjádření 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).

Co vlastně máme na mysli, pokud mluvíme o funkci času výpočtu? Je přirozené, že u obvyklých algoritmů doba výpočtu silně závisí na zadaném vstupu. Avšak informace o všech dobách výpočtu pro všechny možné vstupy by byla tak obsáhlá, že by vlastně na nic nebyla. Proto se při analýze rychlosti algoritmu obvykle soustřeďujeme na to, jak závisí doba výpočtu jen na délce vstupu (místo všech vstupů téže délky).

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). V kapitole 11 jsem se seznámili se strojem RAM.

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 13.1
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 $T_M : \mathbb{N} \to \mathbb{N}$, 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 13.2
Délkou výpočtu (neboli dobou) algoritmu $\mathcal{A}$ na vstupu x rozumíme počet kroků stroje RAM implementujícího algoritmus $\mathcal{A}$, které vykoná na vstupu x až do svého zastavení. Pokud se výpočet nezastaví, délka výpočtu není definovaná (∞).

Definice 13.3
Časová složitost algoritmu $\mathcal{A}$ je funkce $t_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$, kde $t_{\mathcal{A}}(n)$ udává maximální délku výpočtu $\mathcal{A}$ mezi všemi vstupy x délky n. Předpokládá se přitom, že $\mathcal{A}$ vždy svůj výpočet skončí a že vstupy x se berou nad konečnou abecedou $x \in \Sigma^*$, tedy délkou vstupu rozumíme počet znaků nutných k zapsání vstupu.

Takto definované časové složitosti se také říká složitost *nejhoršího případu*, neboť funkční hodnota $t_{\mathcal{A}}(n)$ je dána délkou nejhoršího možného výpočtu

mezi všemi vstupy délky n, přestože běžný výpočet může být mnohem rychlejší. Tento přístup vyjadřuje naši snahu zaručit ukončení výpočtu v rozumné době.

Poznámka: Pro zápis časové složitosti obvykle používáme dříve definované asymptotické značení $O(\cdot)$ nebo $\Theta(\cdot)$, neboť nás příliš nezajímají aditivní a multiplikativní konstanty závislé na hardwarové a softwarové implementaci. Navíc nejčastěji používáme jen *horní odhad* $O(\cdot)$ a dolním odhadem se pro jednoduchost nezabýváme.

(Když náš program bude počítat ještě rychleji, než tvrdíme, asi to nikomu nebude vadit...)

Definice 13.4
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 $S_M : \mathbb{N} \to \mathbb{N}$, 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). 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,
02  STORE  1               STORE  1           ; tj. 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)

$$T_1 = 2 + 7n + 2 + 3 = 7n + 7$$

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

$$T_3 = 2 + 9(n-1) + 5 + 1 = 9n - 1$$

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

$$T_2 = \left(\sum_{j=1}^{n-1} \left(10 + \left(\sum_{i=1}^{n-j} 34 \right) + 8 \right) \right) + 6$$

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

Výraz pro T_2 můžeme upravovat standardní manipulací se sumami například takto:

$$T_2 = 6 + \sum_{j=1}^{n-1} 18 + \sum_{j=1}^{n-1} \sum_{i=1}^{n-j} 34 = 6 + 18(n-1) + \sum_{j=1}^{n-1} 34(n-j) =$$

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

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

$$T_2 = 17n^2 + n - 12$$

Celkový čas potřebný pro zpracování vstupu velikosti n je tedy $T_1+T_2+T_3 = 17n^2 + 17n - 6$. Označíme-li náš RAM-stroj M a jeho časovou složitost T_M, ukázali jsme tak, že pro každé $n \geq 1$ je

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

13.1 Turingovy stroje

Časovou a prostorovou složitost konkrétního Turingova stroje definujeme samozřejmě obdobně jako u RAMu:

Definice 13.5
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 $T_M : \mathbb{N} \to \mathbb{N}$, 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 13.6
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 $S_M : \mathbb{N} \to \mathbb{N}$, 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 $n \,\}$.

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 $0, 1, \Box$ apod.

13.2 Asymptotická složitost

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.

Zajímá-li nás rychlost růstu funkce $f(n)$ v závislosti na n, zaměřujeme se především na tzv. *asymptotické chování* f při velkých hodnotách n. V popisu f nás tedy nezajímají ani různé přičtené „drobné členy“, které se významněji projevují jen pro malá n, ani konstanty, kterými je f násobena a které jen ovlivňují číselnou hodnotu $f(n)$, ale ne rychlost růstu.

Příklad: Tak například funkce $f(n) = n^2$ roste (zhruba) stejně rychle jako $f'(n) = 100000000n^2$ i jako $f''(n) = 0.00000001n^2 - 100000000n - 1000000$. Naopak $h(n) = 0.00000000001n^3$ roste mnohem rychleji než $f'(n) = 100000000n^2$.

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

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

13.3 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 13.7
Pro libovolné funkce $f, g : \mathbb{N} \to \mathbb{N}$ řekneme, že $f \in O(g)$, označujeme též $f(n) \in O(g(n))$, právě tehdy, když platí

$$(\exists k \in \mathbb{N})(\exists n_0 \in \mathbb{N})(\forall n \geq n_0) : \; f(n) \leq k \cdot g(n)\,.$$

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 $f, g : \mathbb{N} \to \mathbb{N}$; pro přehlednost zde opakujeme i definici O):

- $f \in O(g)$, označujeme též $f(n) \in O(g(n))$, právě když platí
  $(\exists k \in \mathbb{N})(\exists n_0 \in \mathbb{N})(\forall n \geq n_0) : \; f(n) \leq k \cdot g(n)$

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

Značení O, o hrají roli neostrého resp. ostrého horního odhadu, značení Ω, ω roli neostrého resp. ostrého dolního odhadu.

Kontrolní otázka: 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 konstantní 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.)

Poznámka: Výše uvedené definice nejsou jediné možné. Například $O(\cdot)$ je možné definovat tak, že $f \in O(g)$ právě tehdy, jestliže existují konstanty $A, B > 0$ takové, že

$$\forall n \in \mathbb{N} : f(n) \leq A \cdot g(n) + B\,.$$

Rozmyslete si, že obě definice jsou ekvivalentní.

V praxi se obvykle (i když matematicky méně přesně) píše místo $f \in O(g)$ výraz

$$f(n) = O(g(n))\,.$$

Znamená to, slovně řečeno, že funkce f *neroste rychleji* než funkce g. (I když pro malá n třeba může být $f(n)$ mnohem větší než $g(n)$.)

Poznámka: Kromě vlastnosti $f \in O(g)$ se někdy setkáte i s vlastností $f \in o(g)$, která znamená $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ (funkce f *roste striktně pomaleji* než g).

Definice 13.8
Píšeme $f \in \Omega(g)$, neboli $f(n) = \Omega(g(n))$, pokud $g \in O(f)$. Dále píšeme $f \in \Theta(g)$, neboli $f(n) = \Theta(g(n))$, pokud $f \in O(g)$ a zároveň $f \in \Omega(g)$, neboli $g \in O(f)$.

Výraz $f(n) = \Theta(g(n))$ pak čteme jako „funkce f *roste stejně rychle* jako funkce g“.

Značení: O funkci $f(n)$ říkáme:

- $f(n) = \Theta(n)$ je *lineární* funkce,
- $f(n) = \Theta(n^2)$ je *kvadratická* funkce,
- $f(n) = \Theta(\log n)$ je *logaritmická* funkce,
- $f(n) = O(n^c)$ pro nějaké $c > 0$ je *polynomiální* funkce,
- $f(n) = \Theta(c^n)$ pro nějaké $c > 1$ je *exponenciální* funkce.

Příklad: Příklady růstů různých funkcí.

Funkce $f(n) = \Theta(n)$: pokud n vzroste na dvojnásobek, tak hodnota $f(n)$ taktéž vzroste (zhruba) na dvojnásobek. To platí jak pro funkci $f(n) = n$, tak i pro $1000000000n$ nebo $n + \sqrt{n}$, atd.

Funkce $f(n) = \Theta(n^2)$: pokud n vzroste na dvojnásobek, tak hodnota $f(n)$ vzroste (zhruba) na čtyřnásobek. To platí jak pro funkci $f(n) = n^2$, tak i pro $1000n^2 + 1000n$ nebo $n^2 - 99999999n - 99999999$, atd.

Naopak pro funkci $f(n) = \Theta(2^n)$: pokud n vzroste byť jen o 1, tak hodnota $f(n)$ už vzroste (zhruba) na dvojnásobek. To je *obrovský rozdíl* exponenciálních proti polynomiálním funkcím.

Pokud vám třeba funkce $999999n^2$ připadá velká, jak stojí ve srovnání s 2^n? Zvolme třeba $n = 1000$, kdy $999999n^2 = 999999000000$ je ještě rozumně zapsatelné číslo, ale $2^{1000} \simeq 10^{300}$ byste už na řádek nenapsali. Pro $n = 10000$ je rozdíl ještě mnohem výraznější!

Otázky:

Otázka 13.1: Co znamená vztah $f(n) = O(g(n))$ pro porovnání hodnot $f(10)$ a $g(10)$?

Cvičení 13.2: Která z těchto funkcí roste nejrychleji?

a) $1000n$ b) $n \cdot \log n$ c) $n \cdot \sqrt{n}$

Cvičení 13.3: Udejte správný asymptotický vztah mezi funkcemi $\sqrt{n}$ a $\log n$.

Cvičení 13.4*: Roste rychleji funkce n^5 nebo $(\log n)^{\log n}$?

Cvičení 13.5*: Která z těchto funkcí roste nejrychleji?

a) $n!$ b) $(\log n)^n$ c) $(\sqrt{n})^n$

Cvičení 13.6*: Dokažte, že pro všechna $c > 0$ a přirozená k platí

$$\log^k n \in O(n^c)$$

13.4 Délka výpočtu

Komentář: Někdy je vhodné kromě časové složitosti nejhoršího případu také uvažovat o *průměrné složitosti* algoritmu, což je definováno jako střední hodnota délky výpočtu na daném pravděpodobnostním prostoru všech vstupů. Pěkným příkladem je třeba algoritmus *quicksort*, který v nejhorším případě trvá až $\Theta(n^2)$, ale v průměru jen $O(n \cdot \log n)$.

Ještě markantnější rozdíl mezi nejhorší a průměrnou složitostí představuje algoritmus tzv. *simplexové metody* v lineární optimalizaci, který v nejhorším případě vykoná exponenciálně mnoho kroků (v tom nejhorším případě je tedy zcela nepoužitelný), ale ve skoro všech ostatních případech běží velice rychle, a proto se ani jiné, v nejhorším případě mnohem lepší algoritmy lineární optimalizace skoro nepoužívají.

Poznámka: Kromě časové jsou i jiné míry složitosti algoritmů, které jen stručně zmíníme zde: Můžeme například sledovat množství paměti použité našim algoritmem (*paměťová složitost*), nebo množství dat vyměněných mezi různými počítači při distribuovaném výpočtu (*komunikační složitost*). Dále můžeme uvažovat třeba *paralelní výpočty* – jakého urychlení se dá dosáhnout rozdělením výpočtu na více procesorů, a mnoho jiných věcí ...

13.5 Cvičení

Otázky:

Otázka 13.7: Vezměme nějaký algoritmus počítající s permutacemi množiny $\{1, 2, \ldots, n\}$. Lze si rozumně představit, že každé z čísel $1, 2, \ldots, n$ zabírá jedno paměťové pole?

Otázka 13.8: Mějme algoritmus počítající hodnotu $n!$ v „dlouhé aritmetice“ postupným násobením. Bylo by rozumné předpokládat, že jednotlivá násobení lze provést najednou v registru stroje RAM?

Cvičení 13.9: Máme za úlohu sečíst dvě (dlouhá) k-místná čísla. Jaká je velikost vstupu v tomto problému? (V asymptotické notaci.)

Cvičení 13.10: Na vstupu algoritmu je dán obecný jednoduchý graf s n vrcholy. Jakou má tento vstup délku?

Cvičení 13.11*: Na vstupu algoritmu je dán rovinný graf s n vrcholy. Jakou má tento vstup délku? Je menší než u obecného grafu?

Cvičení 13.12:Průměr z daných $n > 1$ čísel spočítáme následující funkcí:

```
Průměr(X, n)
    z ← 0.0
    for i ← 1 to n
          do z ← z + X[i]
    return z/n
```

Určete, kolik tato funkce Prumer vykoná aritmetických operací v závislosti na n.

Cvičení 13.13:Určete, kolik průchodů vnitřním cyklem provede pro vstup n následující jednoduchý program.

```
Alg1(n)
    for i ← 1 to n * n
          do for j ← 1 to i
                do print "jeden průchod"
```

Je to v asymptotické notaci $\Theta(n^2)$ nebo $\Theta(n^3)$ nebo $\Theta(n^4)$?

Cvičení 13.14:Rozhodněte, které z následujících asymptotických vztahů mezi funkcemi proměnné n jsou platné. (Pozor, platné mohou být oba nebo i žádný.)

a) $\log n \in O(\sqrt{n})$

b) $\sqrt{n} \in O(n)$

Cvičení 13.15:Rozhodněte, které z následujících asymptotických vztahů mezi funkcemi proměnné n jsou platné. (Pozor, platné mohou být oba nebo i žádný.)

a) $2^n \in O(n^n)$

b) $2^n \in O(n^{1024})$

Cvičení 13.16:Rozhodněte, které z následujících asymptotických vztahů mezi funkcemi proměnné n jsou platné. (Pozor, platné mohou být oba nebo i žádný.)

a) $n! \in O(2^n)$

b) $n^{\log n} \in O(n^{1024})$

Cvičení 13.17:Seřaďte následující tři funkce podle asymptotické rychlosti jejich růstu od nejpomalejšího růstu.

a) $n + \sqrt{n} \cdot \log n$

b) $n \cdot \log n$

c) $\sqrt{n} \cdot \log^2 n$

Cvičení 13.18:Seřaďte následující tři funkce podle asymptotické rychlosti jejich růstu od nejpomalejšího růstu.

a) 2^n

b) $2^{\sqrt{n}}$

c) $n!$

Cvičení 13.19:Seřaďte následující tři funkce podle asymptotické rychlosti jejich růstu od nejpomalejšího růstu.

a) $n/2005$

b) $\sqrt{n} \cdot 3n$

c) $n + n \cdot \log n$

Cvičení 13.20:Seřaďte následující tři funkce podle asymptotické rychlosti jejich růstu od nejpomalejšího růstu.

a) $(\log n)^n$

b) n^n

c) $2^{\sqrt{n}}$

Kapitola 14

Efektivní algoritmy

Cíle kapitoly:

- Pochopení rozdílu mezi složitostí v nejhorším a průměrném případě.
- Pochopení způsobů řešení rekurentních vztahů vznikajících při analýze složitosti algoritmů.

14.1 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“ (například zapsat 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). Toto je obecná definice, která je aplikovatelná vždy, v případě konkrétních problémů však může být někdy vhodnější popisovat velikost vstupu jinak. Vezměme si následující problém:

NÁZEV: *Násobení (čtvercových) matic*

VSTUP: Čtvercové Matice A a B.

VÝSTUP: Matice C taková, že $C = A \cdot B$.

Jestliže jsou matice A a B velikosti $n \times n$, je přirozené brát jako velikost vstupu hodnotu n; přitom (linearizované) zadání čtvercové matice 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.

Někdy je také vhodné popisovat velikost vstupu více čísly a složitost je pak funkcí více parametrů. Typickým příkladem jsou grafové problémy. Jestliže je vstupem problému graf, který má n vrcholů a m hran, může být velikost vstupu popsána dvojicí čísel n a m. Velikost vstupu RAMu (tj. počet členů vstupní posloupnosti, resp. počet bitů nutný k jejich zápisu) pak samozřejmě závisí na tom, jakým konkrétním způsobem bude graf ve vstupu kódován.

Cvičení 14.1: Jaká bude velikost vstupní posloupnosti u problému, kde vstupem je graf, který má n vrcholů a m hran, který je ve vstupu reprezentován:

a) seznamem vrcholů a seznamem hran,

b) incidenční maticí (resp. linearizovaným zápisem této matice).

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:

```
Test-Prime(k)
  ▷ předp. vstup k ≥ 2
  hm ← ⌊√k⌋
  for i ← 2 to hm
        do if k mod i = 0
              then return Ne
  return Ano
```

Když chceme odhadnout složitost algoritmu, musí být samozřejmě 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).

14.2 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.

```
QUICKSORT(A, p, r)
    if p < r
       then q ← PARTITION(A, p, r)
            QUICKSORT(A, p, q)
            QUICKSORT(A, q+1, r)
```

```
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 return j
             prohoď A[i] a A[j]
```

14.3 Některé „rychlé“ algoritmy

V této sekci si uvedeme přehled (horních odhadů) časových složitostí některých běžných problémů. Předpokládáme, že ty jednoduché algoritmy již znáte a některé složitější si v případě potřeby dokážete sami vyhledat v literatuře. Jinými slovy, tato sekce uvádí stručný přehled o tom, jak rychle se umí některé běžné algoritmické problémy řešit, ale nezabývá se konkrétním popisem algoritmů.

Mějme dáno pole A s n prvky (například s čísly nebo řetězci):

- Nalezení konkrétního prvku v A lineárním prohledáváním trvá čas $O(n)$.
- Časová složitost setřídění A algoritmem bubblesort je $O(n^2)$.
- Mnohem rychleji lze pole A setřídit algoritmy heapsort nebo mergesort v čase $O(n \cdot \log n)$.
- V setříděném poli A lze nalézt prvek binárním vyhledáváním v čase $O(\log n)$.

Uvažujme datovou strukturu D s n záznamy, ve které chceme vyhledávat, vkládat nebo odebírat záznamy.

- Pokud je D implementovaná jen polem, každá operace trvá až $O(n)$.
- Pokud je D implementovaná některým vhodným druhem *uspořádaného stromu* (AVL, 2-3 nebo R-B strom), tyto operace trvají jen $O(\log n)$.
- Pokud redukujeme naše požadavky tak, že vybírat nám stačí pouze „nejmenší“ záznam, lze D implementovat tzv. línou haldou, ve které většina operací (v průměru) trvá jen konstantní čas $O(1)$.
- Další možností implementace datové struktury je tzv. hashovací tabulka, která v praktických aplikacích také dává velice krátké (až konstantní) průměrné časy operací, přestože v nejhorší případě složitost bývá až $O(n)$.

Vezměme aritmetiku dlouhých n-místných čísel:

- Sečíst dvě taková čísla lze v čase $O(n)$ a vynásobit v čase $O(n^2)$ běžnými „školními“ postupy.
- Sofistikovaný Strassenův algoritmus vynásobí dvě n-místná čísla v čase $O(n \log n \log \log n)$.

Poznámka: U číselných problémů je třeba dávat velký pozor na to, jaká je velikost vstupu (tj. počet číslic) problému!

Mějme daný graf G s n vrcholy a m hranami ($m = O(n^2)$):

- Komponenty souvislosti G najdeme v čase $O(n + m)$.
- Nejkratší cestu mezi dvěma vrcholy vypočteme v čase $O(n + m \log n)$.

- Minimální kostru nalezneme hladově v čase $O(n + m \log n)$.
- Rovinné nakreslení grafu lze nalézt (pokud existuje) v čase $O(n)$.
- Maximální párování v bipartitním grafu G nalezneme v $O(n^2\sqrt{n})$.

Poznámka: Výše zavedená konvence, že n značí délku vstupu v odhadech časové složitosti se ne vždy plně dodržuje, jak vidíme v těchto příkladech grafových problémů. I zde však n – počet vrcholů, úzce souvisí se skutečnou velikostí vstupu $n + m$.

Podívejme se znovu z druhé strany na problém třídění čísel a jeho časovou složitost. Ukážeme, že pokud vyloučíme „podvodné“ způsoby třídění jako třeba *přihrádkové třídění*, kde se tříděnými čísly indexují pole paměti, nelze získat lepší algoritmus než výše uvedené heapsort či mergesort.

Věta 14.1
Nechť je dáno n čísel. Pokud algoritmus $\mathcal{A}$ správně setřídí daná čísla za použití porovnávání dvojic čísel, pak časová složitost $\mathcal{A}$ je nejméně $\Omega(n \cdot \log n)$.

Důkaz: Pro jednoduchost si stačí představit daná čísla jako množinu $M = \{1, 2, \ldots, n\}$. Na vstupu tak můžeme dostat jednu z $n!$ permutací množiny M. Uvědomme si dobře, že pro dvě různé permutace M na vstupu musí být různé i průběhy algoritmu $\mathcal{A}$, aby byl výstup v obou případech dobře setříděný. Pokud $\mathcal{A}$ provede nejvýše t kroků, může se „rozvětvit“ do nejvýše $2^{O(t)}$ větví výpočtu – různých průběhů. Máme tedy nerovnost

$$2^{O(t)} \geq n!$$

$$O(t) \geq \log(n!) \doteq \Theta(n \cdot \log n)\,,$$

neboli nejhorší výpočet $\mathcal{A}$ musí mít aspoň $t \geq \Theta(n \cdot \log n)$ kroků. □

Ještě pro zajímavost dodáme, že tato věta je jedním z velmi mála tvrzení udávajících absolutní dolní odhady na složitost algoritmů. Problematika dolních odhadů složitosti je prostě velmi obtížná.

Otázky:

Otázka 14.2: Jaká je vlastně délka vstupu u problému násobení dvou matic velikostí $n \times n$?

Cvičení 14.3: Jak rychle byste dokázali vybrat medián z n různých čísel? (Medián je takové číslo, že mezi danými je stejně mnoho větších jako menších než on sám.)

Cvičení 14.4: Jak rychle byste dokázali zjistit, zda daný graf na n vrcholech má nějakou nezávislou množinu velikosti k?

Cvičení 14.5*: Jak rychle byste dokázali nalézt konvexní obal z n bodů v rovině?

Cvičení 14.6*: Je možné navrhnout uspořádanou datovou strukturu, do které by se daly nové prvky přidávat i staré odebírat v (průměrném) konstantním čase?

14.4 Rekurentní vztahy

V tomto oddíle si uvedeme krátký přehled některých rekurentních vzorců, se kterými se můžete setkat při řešení časové složitosti (převážně rekurzivních) algoritmů.

Lemma 14.2
Nechť $a_1, \ldots, a_k, c > 0$ jsou kladné konstanty takové, že $a_1 + \ldots + a_k < 1$, a funkce $T : \mathbb{N} \to \mathbb{N}$ splňuje nerovnost

$$T(n) \leq T(\lceil a_1 n \rceil) + T(\lceil a_2 n \rceil) + \ldots + T(\lceil a_k n \rceil) + cn\,.$$

Pak $T(n) = O(n)$.

Důkaz: Zvolme $\varepsilon > 0$ takové, že $a_1 + \ldots + a_k < 1 - 2\varepsilon$. Pak pro dostatečně velká n platí (i se zaokrouhlením nahoru) $\lceil a_1 n \rceil + \ldots + \lceil a_k n \rceil \leq (1 - \varepsilon)n$, řekněme pro všechna $n \geq n_0$. Dále zvolme dostatečně velké $d > 0$ tak, že $\varepsilon d > c$ a zároveň $d > \max\{\frac{1}{n}T(n) : n = 1, \ldots, n_0\}$.

Dále už snadno indukcí podle n dokážeme $T(n) \leq dn$ pro všechna $n \geq 1$:

- Pro $n \leq n_0$ je $T(n) \leq dn$ podle naší volby d.

- Předpokládejme, že $T(n) \leq dn$ platí pro všechna $n < n_1$, kde $n_1 > n_0$ je libovolné. Nyní dokážeme i pro n_1

$$T(n_1) \leq T(\lceil a_1 n_1 \rceil) + \ldots + T(\lceil a_k n_1 \rceil) + cn_1 \leq$$

$$\leq d \cdot \lceil a_1 n_1 \rceil + \ldots + d \cdot \lceil a_k n_1 \rceil + cn_1 \leq$$

$$\leq d \cdot (1 - \varepsilon) n_1 + cn_1 \leq dn_1 - (\varepsilon d - c) n_1 \leq dn_1 \,.$$

□

Lemma 14.3
Nechť $k \geq 2$ a $a_1, \ldots, a_k, c > 0$ jsou kladné konstanty takové, že $a_1 + \ldots + a_k = 1$, a funkce $T : \mathbb{N} \to \mathbb{N}$ splňuje nerovnost

$$T(n) \leq T(\lceil a_1 n \rceil) + T(\lceil a_2 n \rceil) + \ldots + T(\lceil a_k n \rceil) + cn \,. \tag{1}$$

Pak $T(n) = O(n \cdot \log n)$.

Důkaz (náznak): Bylo by možno postupovat obdobně jako v předchozím důkaze, ale výpočty by byly složitější. Místo formálního důkazu indukcí nyní předestřeme poměrně jednoduchou úvahu zdůvodňující řešení $T(n) = O(n \cdot \log n)$.

Představme si, že upravujeme pravou stranu výrazu (1) v následujících krocích: V každém kroku rozepíšeme každý člen $T(m)$ s dostatečně velkým argumentem m rekurzivní aplikací výrazu (1) (s $T(m)$ na levé straně). Jelikož $a_1 + \ldots + a_k = 1$, součet hodnot argumentů všech $T(\cdot)$ ve zpracovávaném výrazu bude stále zhruba n. Navíc po zhruba $t = \Theta(\log n)$ krocích už budou hodnoty argumentů všech $T(\cdot)$ „malé“ (nebude dále co rozepisovat), neboť $0 < a_i < 1$ a tudíž $a_i^t \cdot n < 1$ pro všechna i. Při každém z kroků našeho rozpisu se ve výrazu (1) přičte hodnota $cn = O(n)$, takže po t krocích bude výsledná hodnota

$$T(n) = t \cdot O(n) + O(n) = O(n \cdot \log n) \,.$$

Vyzkoušejte si tento postup sami na konkrétním příkladě $T'(n) \leq 2T'\left(\frac{n}{2}\right) + n$.
□

V obecnosti je známo:

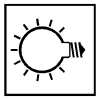
Lemma 14.4
Nechť $a \geq 1$, $b > 1$ jsou konstanty, $f : \mathbb{N} \to \mathbb{N}$ je funkce a pro funkci $T : \mathbb{N} \to \mathbb{N}$ platí rekurentní vztah

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + f(n).$$

Pak platí:

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

Důkaz tohoto obecného tvrzení přesahuje rozsah našeho předmětu. Všimněte si, že nikde ve výše uvedených řešeních nevystupují počáteční podmínky, tj. hodnoty $T(0), T(1), T(2), \ldots$ – ty jsou „skryté“ v naší $O()$-notaci. Dále v zápise pro zjednodušení zanedbáváme i necelé části argumentů, které mohou být zaokrouhlené.



Řešený příklad 14.1: Algoritmus mergesort pro třídění čísel pracuje zhruba následovně:

- Danou posloupnost n čísel rozdělí na dvě (skoro) poloviny.
- Každou polovinu setřídí zvlášť za použití rekurentní aplikace mergesort.
- Tyto dvě už setříděné poloviny „slije“ (anglicky merge) do jedné setříděné výsledné posloupnosti.

Jaký je celkový počet jeho kroků?

Řešení: Nechť na vstupu je n čísel. Při rozdělení na poloviny nám vzniknou podproblémy o velikostech $\lceil n/2 \rceil$ a $\lfloor n/2 \rfloor$ (pozor na necelé poloviny). Pokud počet kroků výpočtu označíme $T(n)$, pak rekurzivní volání trvají celkem

$$T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)\,.$$

Dále potřebujeme $c \cdot n$ kroků (kde c je vhodná konstanta) na slití obou částí do výsledného setříděného pole. Celkem tedy vyjde

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn \leq T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + cn$$

a to už je tvar řešený v Lemmatu 14.3 pro $a_1 = a_2 = \frac{1}{2}$. Výsledek tedy je $T(n) = O(n \cdot \log n)$.

(Stejný výsledek by bylo možno získat i z Lemmatu 14.4 pro $a = b = 2$.)

14.5 Cvičení

Otázky:

Otázka 14.7: Proč nám předchozí tvrzení dávají jen asymptotické odhady na funkci $T(n)$ a ne přesné vzorce?

Otázka 14.8: Proč požadujeme $b > 1$ v Lemmatu 14.4?

Cvičení 14.9: Odhadněte asymptoticky výsledek rekurence $T(n) \leq T(n/2) + T(n/5) + 2n$.

Cvičení 14.10: Odhadněte asymptoticky výsledek rekurence $T(n) \leq T(n/2) + T(n/3) + T(n/6) + 2n$.

Cvičení 14.11: Odhadněte asymptoticky výsledek rekurence $T(n) \leq 4T(n/3) + 2n^2$.

Cvičení 14.12: Odhadněte asymptoticky výsledek rekurence $T(n) \leq 4T(n/2) + 2n^2$.

Cvičení 14.13:Jednoduchá implementace Euklidova algoritmu největšího společného dělitele dvou přirozených čísel a, b počítá výsledek následovně:

```
EUCLID(a, b)
  if b = 0
    then return a
    else  if a > b
            then return EUCLID(b, a − b)
            else  return EUCLID(a, b − a)
```

Nechť k označuje počet bitů v binárním zápise čísel a, b. Jaká je nejhorší možná časová složitost zadaného algoritmu vzhledem ke k? (Uvažujte jednotkový čas na každou aritmetickou operaci.)

CVIČENÍ 14.14:Kolik kroků (elementárních výpočetních operací) provede následující efektivní implementace Euklidova algoritmu pro největšího společného dělitele na vstupech – číslech a, b, která mají v binárním zápise nejvýše ℓ bitů? (Předpokládejte, že aritmetické operace trvají jednotkový čas.)

```
EUCLID(a, b)
    while b ≠ 0
          do c ← a mod b
             a ← b
             b ← c
    return a
```

CVIČENÍ 14.15:Představme si, že z daných n čísel máme vybrat k-té v uspořádání podle velikosti. (Nejpřirozenějším postupem by bylo čísla setřídit a pak k-té z nich vybrat, ale to je spousta zbytečných výpočtů navíc, že?)

Pro rychlý výpočet použijeme následující rekurzivní algoritmus, svým způsobem podobný algoritmu quicksort.

Rekurzivní procedura VYBER(A, p, r, k) vracející k-tý nejmenší prvek z úseku pole $A[p\,..\,r]$ (kde $p \le k \le r$) pracuje následovně:

- Z úseku $A[p\,..\,r]$ zvolíme libovolný prvek x a tento úsek přeuspořádáme tak, že bude rozdělen na tři části:
  - $A[p\,..\,q-1]$ – obsahující prvky, které jsou menší než x,
  - $A[q\,..\,q']$ – obsahující prvky, které jsou rovny x,
  - $A[q'+1\,..\,r]$ – obsahující prvky, které jsou větší než x
- Pokud je $q \le k \le q'$, je výsledkem x.
- Pokud $k < q$, výsledek spočítáme rekurzivním výpočtem jako VYBER$(A, p, q-1, k)$.

- Pokud $k > q'$, výsledek spočítáme rekurzivním výpočtem jako VYBER$(A, q'+1, r, k)$.

Jaká je časová složitost popsaného algoritmu?

CVIČENÍ 14.16:Určete, kolik průchodů vnitřním cyklem provede pro vstup n následující jednoduchý program.

```
ALG2(n)
    for i ← 1 to n
        do for j ← 1 to i * i
               do print "jeden průchod"
```

Je to v asymptotické notaci $\Theta(n^2)$ nebo $\Theta(n^3)$ nebo $\Theta(n^4)$?

CVIČENÍ 14.17: Určete, kolik průchodů cyklem provede následující program pro vstup n. Zapište výsledek v asymptotické notaci $\Theta(\cdot)$.

```
ALG3(n)
    i ← 1
    while i < n
        do print "jeden průchod"
           i ← i + i
```

CVIČENÍ 14.18*:Určete, kolik průchodů cyklem provede následující program pro vstup n. Zapište výsledek v asymptotické notaci $\Theta(\cdot)$.

```
ALG4(n)
    i ← 1; j ← 1
    while i < n
        do print "jeden průchod"
           i ← i + j
           j ← j + 1
```

Cvičení 14.19*: Jak byste upravili algoritmus v Příkladě 14.15, aby počítal v (nejhorším) lineárním čase?

Návod: Je potřeba najít vhodný způsob výběru prvku x tak, aby bylo zaručeno, že rozdělení nebude velmi nevyvážené.

Cvičení 14.20:Rozeberte a zdůvodněte, jakou časovou složitost (v asymptotické notaci) má následující problém: Daná je posloupnost z čísel $1, 2, \ldots, n$ (s možným opakováním i chybějícími čísly). Úkolem je zjistit, zda tato posloupnost je permutací.

Cvičení 14.21:Odhadněte asymptoticky výsledek rekurence

$$T(n) = T(n/2) + T(n/3) + T(n/4) + 5n^2\,.$$

Cvičení 14.22:Nezáporná funkce $T(n)$ splňuje pro všechna přirozená n následující rekurentní vztah

$$T(n) \;\leq\; 3 \cdot T(n/5) + n\,.$$

Jaký je (nejlepší) asymptotický odhad růstu funkce $T(n)$ v závislosti na n?

Cvičení 14.23:Nezáporná funkce $T(n)$ splňuje pro všechna přirozená n následující rekurentní vztah

$$T(n) \;\leq\; 4 \cdot T(n/2) + n\,.$$

Jaký je (nejlepší) asymptotický odhad růstu funkce $T(n)$ v závislosti na n?

Cvičení 14.24:Nezáporná funkce $T(n)$ splňuje pro všechna přirozená n následující rekurentní vztah

$$T(n) \;\leq\; T(n/3) + T(n/4) + T(n/5) + n\,.$$

Jaký je (nejlepší) asymptotický odhad růstu funkce $T(n)$ v závislosti na n?

Cvičení 14.25:Nezáporná funkce $T(n)$ splňuje pro všechna přirozená n následující rekurentní vztah

$$T(n) \leq T(n/2) + T(n/3) + T(n/6) + n\,.$$

Jaký je (nejlepší) asymptotický odhad růstu funkce $T(n)$ v závislosti na n?

Cvičení 14.26:Uvažujme následující problém, kde vstupem je nějaká množina booleovských proměnných $V = \{x_1, x_2, x_n\}$ a výstupem booleovská formule φ (vytvořená z proměnných z množiny V a booleovských operátorů $\wedge$, $\vee$ a $\neg$) taková, že φ nabývá hodnoty TRUE právě tehdy, když právě jedna z proměnných z množiny V nabývá hodnoty TRUE.

Jedním z možných řešení je následující rekurzivní algoritmus:

- Nechť $n = |V|$. Pokud $n = 1$, vrať (jedinou) proměnnou z V.
- Pokud $n > 1$:
  - Rozděl V na množiny L a R takové, že $L \cup R = V$, $L \cap R = \emptyset$, $|L| = \lceil n/2 \rceil$ a $|R| = \lfloor n/2 \rfloor$.
  - Rekurzivně vytvoř formule φ_L a φ_R pro množiny L a R.
  - Vrať formuli

$$(\varphi_L \wedge \bigwedge_{x_i \in R} \neg x_i) \vee (\varphi_R \wedge \bigwedge_{x_i \in L} \neg x_i)$$

Co nejpřesněji odhadněte velikost výsledné formule. Pro jednoduchost počítejte, že velikost názvu proměnné je v $\Theta(1)$.

Poznámka: Zápis $\bigwedge_{x_i \in X} x_i$ označuje formuli $x_1 \wedge x_2 \wedge \cdots \wedge x_k$, kde $X = \{x_1, x_2, \ldots, x_k\}$.

Cvičení 14.27:Jak jistě víte, vynásobit dvě n-místná čísla školním postupem (každou číslici s každou) trvá čas $\Theta(n^2)$. My si slovně popíšeme rychlejší rekurzivní algoritmus. Předpokládejme, že $n = 2k$ je velmi velké číslo (pokud je n liché, doplníme nulu). Součin $a \cdot b$ dvou n-místných čísel a, b vypočítáme následovně:

- Rozdělíme obě čísla na poloviční úseky jejich dekadických zápisů, tj. $a = 10^k \cdot a_1 + a_2$ a $b = 10^k \cdot b_1 + b_2$. Jednoduše upravíme součin $a \cdot b = (10^k \cdot a_1 + a_2)(10^k \cdot b_1 + b_2) = 10^n(a_1 \cdot b_1) + 10^k(a_1 b_2 + a_2 b_1) + (a_2 \cdot b_2)$. Takže pro výpočet $a \cdot b$ nám stačí spočítat každý ze třech výrazů v posledních závorkách.
- Rekurzivní aplikací téhož algoritmu násobení spočítáme $z_1 = (a_1 \cdot b_1)$ i $z_3 = (a_2 \cdot b_2)$.
- Dále jednoduchým sečtením a následnou rekurzivní aplikací algoritmu násobení spočítáme $(a_1 + a_2) \cdot (b_1 + b_2)$. Z toho už jednoduchým odečtením vypočítáme hodnotu poslední požadované závorky $z_2 = (a_1 b_2 + a_2 b_1) = (a_1 + a_2) \cdot (b_1 + b_2) - z_1 - z_3$.
- Mezivýsledky sčítáním složíme do výsledného $a \cdot b = 10^n z_1 + 10^k z_2 + z_3$.

Jaká je časová složitost popsaného algoritmu?

Cvičení 14.28*: Chytrý Strassenův algoritmus pro násobení dvou matic velikosti $n \times n$ pracuje zhruba následovně: Každá matice A, B se rozdělí do čtyř podmatic polovičních velikostí a součin $A \times B$ se rozepíše pomocí známých pravidel násobení a chytrého triku do sedmi součinů a několika součtů mezi zmíněnými polovičními podmaticemi. Jaká je časová složitost takového algoritmu?

Cvičení 14.29*: Jak byste v Příkladě 14.27 odvodili výsledný asymptotický odhad na $T(n)$ bez zanedbání "+1" v $T(k+1)$?

Kapitola 15

Složitost problémů

Cíle kapitoly:

- Pochopení pojmu složitost problému.
- Seznámení se s třídami složitosti a speciálně pak se třídou PTIME.
- Seznámení se s polynomiálními převody mezi problémy.

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).

Neméně důležitou otázkou je, jak vlastně máme měřit výpočetní obtížnost, čili „*složitost*“ daného problému. Přirozenou mírou je zde čas, které na řešení tohoto problému musíme věnovat (čím obtížnější problém, tím déle nám to trvá). Pro objektivní posouzení obtížnosti však musíme eliminovat subjektivní vlivy jako chytrost řešitele či rychlost počítače.

V tomto ohledu se jako mnohem vhodnější míra obtížnosti problému ukazuje ne měření samotného času nutného k vyřešení problému, ale sledování tempa nárůstu času řešení při zvětšování vstupu.

15.1 Časová složitost problému

Vzpomeňme si, že problém můžeme formálně definovat jako zobrazení $P : \Sigma^* \to \Sigma^*$, kde $w \in \Sigma^*$ je vstup a $P(w)$ je příslušný výstup. Algoritmus $\mathcal{A}$ *řeší problém* P pokud implementace $\mathcal{A}$ (na stroji RAM) vždy skončí výpočet a pro každý vstup w odpoví na výstupu $P(w)$.

Všimněme si teď otázky *složitosti 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 či Turingových strojů. Víme-li např. o algoritmu (RAMu, Turingově stroji), 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“).

Na určení *horního odhadu* $f(n)$ složitosti problému (tedy ukázání, že složitost je nejvýše $f(n)$ pro každé n) stačí navrhnout algoritmus s časovou složitostí $f(n)$.

Na druhou stranu, pokud jsme schopni dokázat, že *každý* algoritmus, který daný problém řeší, má složitost alespoň $f(n)$ (tj. složitost každého takového algoritmu je v $\Omega(f(n))$), hovoříme o *dolním odhadu* složitosti problému.

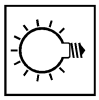
Poznámka: Ideální je, pokud se pro horní i dolní odhad složitosti problému rovnají. Pak to znamená, že pokud máme algoritmus, který řeší daný problém a má stejnou složitost jako je složitost tohoto problému, pak je tento algoritmus optimální.

V praxi je situace taková, že mezi horním a dolním odhadem může být značná „mezera“. Zejména co se týká dolních odhadů, u řady problémů známe většinou pouze triviální odhady (např. odhady typu, že algoritmus musí vstup velikosti n alespoň načíst, aby mohl určit odpověď, takže musí vykonat alepoň $\Theta(n)$ kroků).

Nějaký netriviální dolní odhad jen málokdy umíme odvodit. (Při odvozování dolního odhadu samozřejmě nemůžeme postupovat tak, že bychom prozkoumali všechny algoritmy, které daný problém řeší. Většinou se postupuje důkazem sporem. Předpokládáme, že by existoval algoritmus, který by měl složitost menší než $f(n)$ a ukážeme, že by tento předpoklad vedl k logickému

sporu.)

Jedním z mála netriviálních dolních odhadů je výsledek, že každý algoritmus řešící problém třídění (a založený na porovnávání dvojic prvků) má nutně složitost $\Omega(n \log n)$. Vzhledem k tomu, že pro tento jsou známy algoritmy se složitostí $O(n \log n)$ (např. Heapsort a Mergesort), složitost problému třídění je takto určena přesně (samozřejmě až na zanedbávané konstantní faktory) – horní odhad se rovná dolnímu.



Řešený příklad 15.1: Jaký je horní a dolní odhad složitosti problému spočítat součet prvků pole přirozených čísel velikosti n?

Řešení: Jelikož v zadání není řečen opak, předpokládáme, že zadaná čísla mají rozumnou velikost, tedy že každé je uloženo celé v jednom místě paměti. Snadno pak odvodíme dolní odhad potřebného času $\Omega(n)$, neboť každý správný algoritmus musí aspoň všech n čísel přečíst. Naopak snadno napíšeme (a případně podrobně rozepíšeme) algoritmus, který tento výsledek získá v čase $O(n)$:

```
SOUČET(a, n)
    s ← 0
    for i ← 1 to n
          do s ← s + a[i]
    return s
```

Časová složitost našeho problému tedy je $\Theta(n)$.

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 15.1
Pro funkci $f : \mathbb{N} \to \mathbb{N}$ rozumíme *třídou časové složitosti* $\mathcal{T}(f)$, též značenou $\mathcal{T}(f(n))$, množinu těch problémů, které jsou řešeny RAMy s časovou 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 při odkazu na složitost RAMu na mysli *logaritmickou míru*, pokud není uvedeno jinak. Jde o to, aby takto definované třídy složitosti rozumně odpovídaly realitě.

Poznámka: Podobně bychom mohli zavést definice tříd složitosti i pro jiné výpočetní modely (např. pro Turingovy stroje). V takovém případě však dostaneme *jiné* třídy složitosti, proto je třeba vždy uvést, k jakému výpočetnímu modelu se dané třídy složitosti vztahují.

Řekneme-li tedy například, 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

$$P \in \mathcal{T}(f) \ \wedge \ f \in O(g) \ \Rightarrow \ P \in \mathcal{T}(g)$$

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 například 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.

15.2 Třída P (neboli PTIME)

Jako nejrozumnější (horní) aproximace třídy zvládnutelných problémů se (zatím) ukázala třída označovaná PTIME, nebo jen P (ze slova „Polynomial“),

definovaná

$$\mathsf{PTIME} = \bigcup_{k=0}^{\infty} \mathcal{T}(n^k)$$

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 například 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 15.1, 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 15.1). V literatuře se ovšem často v uvedené definici „rozumnosti“ odkazuje k historicky prvnímu modelu počítače – k Turingůvě stroji.

Poznamenejme ještě, že uvažujeme *sekvenční modely*, k otázce *paralelních modelů* se letmo dostaneme později.

Poznámka: Problématikou vzájemné simulace stroje RAM a Turingova stroje jsme se již zabývali dříve. Není težké si rozmyslet, že jak při simulaci Turingova stroje strojem RAM, tak při simulaci stroje RAM pomocí Turingova stroje, vzroste počet kroků i množství použité paměti nanejvýš polynomiálně.

Možná by si čtenář myslel, že třída PTIME by měla být definována s omezeným exponentem c při časové složitosti $O(n^c)$, například jako všechny problémy řešitelné v čase $O(n^5)$ nebo podobně. Vždyť přece algoritmus pracující v čase $\Theta(n^{1000})$ už v žádném případě nelze považovat za „efektivní“! Exponent c se však v teorii neomezuje, hlavně proto, že by jinak různé modely nebyly polynomiálně ekvivalentní – všimněte si například, že Turingův stroj simulující výpočet stroje RAM s časovou složitostí $\Theta(n^c)$ pracuje v čase $\Theta(n^{c''})$, kde $c' > c$. (Také by při pevném omezení exponentu nebylo možno používat polynomiální převody, které definujeme níže.)

Přesto je vhodné považovat třídu PTIME za třídu rozumně zvládnutelných problémů, neboť se ukazuje, že pokud je pro nějaký praktický (rozumně definovaný) problém známo, že patří do třídy PTIME (tj. je znám polynomiální algoritmus řšící tento problém), pak lze tento problém řešit s časovou složitostí $O(n^c)$ s „rozumně malým“ exponentem c, obvykle třeba $c \leq 5$.

Poznámka: Dá se ovšem ale například ukázat, že pro libovolné c existuje problém, který se nedá řešit v čase $O(n^c)$, ale dá se řešit v čase $O(n^{c+1})$.

V tomto případě se ovšem jedná o problémy, které byly uměle zkonstruovány pro potřeby důkazu, nikoliv o praktické rozumně definované problémy.

15.3 Polynomiální převod

Zajisté se již čtenář setkal se situací, kdy zadaný problém místo přímého řešení raději převedeme na podobný, již vyřešený problém. Toto se hojně využívá i v informatice, neboť základní algoritmy obvykle jsou k dispozici

naprogramované v knihovnách a my často jen překládáme dané specifické problémy do tvarů oněch knihovních algoritmů, které voláme pro samotné vyřešení. V teorii se formalizuje pojem polynomiálního převodu.

Definice 15.2
Mějme dva problémy $P_1 : \Sigma^* \to \Sigma^*$ a $P_2 : \Sigma^* \to \Sigma^*$ nad stejnou abecedou. *Převodem* P_1 na P_2 rozumíme algoritmus počítající zobrazení $R : \Sigma^* \to \Sigma^*$ takové, že pro všechny vstupy $w \in \Sigma^*$ platí $P_1(w) = P_2(R(w))$.

Definice 15.3
Polynomiální převod (jinak také redukce) problému P_1 na P_2 je převod R počítaný algoritmem s polynomiální časovou složitostí.

Definice polynomiálního převodu nám tedy říká, že pokud umíme efektivně řešit problém P_2, pak problém P_1 také můžeme efektivně vyřešit tím, že jej (rychle) převedeme na známý problém P_2.

Poznámka: Tato definice převodu je silně restriktivní v tom, že požaduje výstup problému P_2 přímo ve tvaru výstupu P_1. Proto se uvedená definice převodu aplikuje především na *rozhodovací problémy*, u kterých je výstup ANO/NE. *Zobecněný převod* pro výpočetní problémy bychom definovali jako dvojici zobrazení R, R' takovou, že $P_1(w) = R'(P_2(R(w)))$, kde druhá převodové zobrazení R' převádí výstup z P_2 zpět do tvaru výstupu P_1.

Definice polynomiálního převodu bude také klíčová pro další partie našeho předmětu v následujícím smyslu:

Představme si, že se nám stále nedaří pro daný problém P nalézt efektivní algoritmus (pracující v polynomiálním čase). Už tušíme, že něco takového asi není ani možné, ale jak o tom přesvědčíme kolegu/šéfa? (Co kdyby si oni mysleli, že jsme jen líní efektivní algoritmus najít?) Stačí ukázat, že existuje polynomiální převod nějakého jiného (známého) těžkého problému Q na náš problém P!

Jinými slovy, pokud víme, že problém Q se už mnoho chytrých lidí pokoušelo efektivně vyřešit a neuspělo, pak náš problém P, na který jsme Q převedli, musí být alespoň tak těžký jako Q. Takovým polynomiálním převodem Q na P jsme si sice nepomohli v řešení P, ale ušetřili jsme si spoustu marných pokusů o nalezení efektivního algoritmu.

Cvičení 15.1: Jak zobecněně převedeme problém násobení dvou čísel $a \cdot b$

na problém sčítání $c + d$? Používal se tento převod v minulosti často?

15.4 Cvičení

Otázky:

Otázka 15.2: Proč je časová složitost problémů na vstupech délky n obvykle nejméně $\Omega(n)$?

Cvičení 15.3: Jakou časovou složitost má problém setřídění n daných čísel?

Cvičení 15.4: Je dána matice A (tj. dvourozměrné pole) o rozměrech $n \times n$ s hodnotami $0, 1$, která definuje graf G s vrcholy $\{1, 2, \ldots, n\}$ následovně: Vrcholy i, j jsou spojeny hranou v G právě když $A[i, j] == A[j, i] == 1$. Jaká je časová složitost zjištění největšího stupně vrcholu grafu G?

Cvičení 15.5: Co když, na rozdíl od předchozího příkladu, je graf dán seznamem sousedů každého vrcholu? Jakou časovou složitost má pak zjištění největšího stupně vrcholu grafu G?

Kapitola 16

NP-úplnost

Cíle kapitoly:

- Pochopení pojmu nedeterministického výpočtu a definice třídy NPTIME (obsahující problémy s kladnou polynomiální nápovědou).
- Seznámení se s tzv. NP-úplnými problémy.

Poznámka: Definice třídy NPTIME je poměrně obtížná, proto čtenářům doporučujeme si ji přečíst mnohokrát a pokusit se tak proniknout až k jejímu myšlenkovému jádru.

V minulé kapitole jsme uvedli třídu PTIME všech efektivně řešitelných algoritmických problémů. Bohužel však svět není tak jednoduchý a na řešení mnoha praktických problémů žádný efektivní algoritmus není znám. (Jinými slovy, takové problémy nejspíše nepatří do třídy PTIME.)

Na druhou strany mnoho, dá se říci většina, prakticky motivovaných algoritmických problémů je popsána ve stylu „nalezněte řešení splňující dané podmínky“; kde sice nalezení onoho vyhovujícího řešení není lehké, ale ověření, zda někým navržené či uhodnuté řešení podmínkám vyhovuje, bývá snadné. To ideově vede k následující definici širší třídy NPTIME, nazývané též zkráceně NP.

Zhruba řečeno, *problém patří do třídy* NPTIME, pokud kladnou odpověď na něj lze prokázat (ve smyslu „uhodnout a ověřit“) výpočtem, který běží v

polynomiálním čase. Definice třídy NPTIME se tak týká *výhradně rozhodovacích problémů*. To však není na velkou újmu obecnosti uvažování, neboť vlastně každý problém lze nahradit několika rozhodovacími verzemi.

Třída NPTIME je důležitá hlavně proto, že zahrnuje rozhodovací verze řady běžných praktických problémů. Navíc vlastnost, že správnost i nějakého magicky uhodnutého řešení umíme efektivně ověřit, je zajisté významná v praxi. Přesto většinu problémů v třídě NPTIME nejsme sami schopni efektivně vyřešit. Náplní této i příští přednášky tak bude i stručné pochopení důvodů, proč jsou ve třídě NPTIME obtížně řešitelné úlohy a jak je poznat.

16.1 Třída NPTIME

Pro definici třídy NPTIME je třeba nejprve zavést pojem nedeterministického algoritmu. My budeme konkrétně definovat nedeterministický Turingův stroj, ale podobně bychom mohli použít i jiný „rozumný“ výpočetní model.

Definice 16.1
Nedeterministický Turingův stroj je definován obdobně jako deterministický Turingův stroj, jen přechodová funkce dovoluje nedeterminismus, tedy možnost přechodu do více stavů stroje současně.

Definice 16.2
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.

Třída rozhodovacích problémů řešených nedeterministickými Turingovými stroji se shoduje s třídou řešenou deterministickými Turingovými stroji, neboť všechny nedeterministické výpočty jednoho stroje lze simulovat rekurzivním prohledáváním na deterministickém stroji. Počet kroků výpočtu v takovéto simulaci však vzroste exponenciálně, a proto nedeterministický stroj má svůj význam při sledování časové složitosti výpočtu.

Poznámka: Podobně bychom mohli definovat např. i nedeterministický stroj RAM, například rozšířením instrukce `JUMP` o možnost skoku na více různých adres (mezi kterými by se jedna nedeterministicky vybrala).

Složitost nedeterministického Turingova stroje a příslušné třídy složitosti lze definovat takto:

Definice 16.3
Časová složitost nedeterministického Turingova stroje M je zobrazení $T_M : \mathbb{N} \to \mathbb{N}$, kde $T_M(n)$ znamená maximální délku výpočtu pro vstup velikosti n.

Třídou časové složitosti $N\mathcal{T}(f)$ pro funkci $f : \mathbb{N} \to \mathbb{N}$ rozumíme třídu těch problémů, které jsou řešeny nedeterministickými Turingovými stroji s časovou složitostí v $O(f)$.

Čtenář si jistě snadno doplní definice pro prostorovou složitost.

Třídu NPTIME nyní můžeme definovat jako

$$\mathsf{NPTIME} = \bigcup_{k=0}^{\infty} N\mathcal{T}(n^k)$$

NPTIME je tedy třída těch problémů, které jsou řešitelné nedeterministickými Turingovými stroji v polynomiálním čase.

Poznámka: 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 NPTIME 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).

(To, že PTIME $\subseteq$ NPTIME je ovšem zřejmé, neboť deterministické algoritmy jsou speciálním případem nedeterministických.)

Uveďme si příklady některých problémů, které patří do třídy NPTIME.

Mnoho těchto problémů vypadá tak, že se ptáme na existenci nějakého objektu (např. množiny, přiřazení, čísla apod.), který splňuje nějaké dané pod-

mínky. Nedeterministické algoritmy (implementované například nedeterministickým Turingovým strojem) řešící tento typ problémů v polynomiálním čase pracují většinou tak, že nejprve nedeterministicky uhodnou, jak tento objekt na jehož existenci se ptáme, vypadá, a poté (už deterministicky) ověří, zda se skutečně jedná o tento hledaný objekt a podle toho vydají odpověď Ano nebo Ne.

Název: *Složenost čísla*

Vstup: Přirozené číslo ℓ.

Otázka: Je číslo ℓ složené?

U tohoto problému algoritmus nejprve nedeterministicky zvolí číslo x takové, že $1 < x < \ell$, a poté ověří, zda $\ell \bmod x = 0$. Tj. algoritmus nedeterministicky hádá netriviální dělitel čísla ℓ. (Pro tento problém je znám i deterministický polynomiální algoritmus.)

Název: CG (*Barvení grafu*)

Vstup: Neorientovaný graf G a číslo k.

Otázka: Je možné graf G obarvit k barvami (tj. existuje přiřazení barev vrcholům tak, aby žádné dva sousední vrcholy nebyly obarveny stejnou barvou)?

V tomto případě algoritmus nejprve nedeterministicky zvolí přiřazení barev jednotlivým vrcholům a poté ověří, že se jedná o korektní obarvení.

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)?

Algoritmus nejprve nedeterministicky zvolí k vrcholů z grafu G a poté ověří, že tyto vrcholy tvoří nezávislou množinu.

NÁZEV: HK (*problém hamiltonovské kružnice*)

VSTUP: Neorientovaný graf G.

OTÁZKA: Existuje v G hamiltonovská kružnice (tj. uzavřená cesta, procházející každým vrcholem právě jednou)?

Algoritmus nedeterministicky uhodne posloupnost hran tvořící tuto kružnici a ověří, že každý vrchol je navštíven právě jednou.

NÁZEV: *Isomorfismus grafů*

VSTUP: Dva neorientované grafy G a H.

OTÁZKA: Jsou grafy G a H isomorfní?

Algoritmus nedeterministicky uhodne isomorfismus mezi oběma grafy a ověří, že se skutečně jedná o isomorfimus.

NÁZEV: *Subset-Sum*

VSTUP: Multimnožina přirozených čísel $M = \{x_1, x_2, \ldots, x_n\}$ a přirozené číslo s.

OTÁZKA: Existuje podmnožina multimnožiny M, která dává součet s?

Algoritmus nedeterministicky zvolí podmnožinu multimnožiny M a ověří, že součet čísel v této množině je s.

VSTUP: Nevypouštějící bezkontextová gramatika G bez jednoduchých pravidel, slovo w.

OTÁZKA: Patří slovo w do jazyka $L(G)$?

Algoritmus nedeterministicky uhodne deterivaci slova w v gramatice G.

Existuje také alternativní definice třídy NPTIME, která se neodkazuje k nedeterminismu:

Definice 16.4
Třída NPTIME, zkráceně NP, je třídou všech rozhodovacích problémů $P : \Sigma^* \to \{0, 1\}$ (NE/ANO) nad konečnou abecedou Σ takových, že existuje

zobrazení $R : \Sigma^* \times \Sigma^* \to \{0,1\}$ počítané algoritmem s polynomiální časovou složitostí v délce prvního argumentu ($|x|$), pro které je $P(x) = 1$ (Ano) pro $x \in \Sigma^*$, právě když pro nějaké $y \in \Sigma^*$ platí $R(x,y) = 1$. Zkráceně, pro nějaké $R \in \mathsf{P}$ platí

$$\forall x \in \Sigma^* : \quad P(x) = 1 \iff \big(\exists y \in \Sigma^* : R(x,y) = 1\big) .$$

V této definici je x vstupem problému P a y hraje roli „nápovědy“ správného řešení pro $P(x)$, jehož přípustnost ověříme efektivním algoritmem R, jehož polynomiální čas výpočtu se měří jen vzhledem k délce x, ne y. (O efektivitě nalezení správného y se v této definici nehovoří! Nápovědu y je proto třeba „uhodnout“ a mimo jiné musí být jen polynomiálně velké vzhledem k $|x|$.) V rozhodovacím problému $P(x)$ pak je odpověď Ano, právě když pro vstup x existuje přípustná „nápověda“.

Věta 16.5
Obě definice třídy NPTIME *jsou ekvivalentní.*

Důkaz: . Nechť problém $P \in \mathsf{NPTIME}$, tj. dle definice existuje polynomiální R, pro které platí

$$\forall x \in \Sigma^* : \quad P(x) = 1 \iff \big(\exists y \in \Sigma^* : R(x,y) = 1\big) .$$

Nedeterministický Turingův stroj M implementuje polynomiální výpočet zobrazení $R(x,y)$ tak, že jednotlivé bity „nápovědy“ y nahrazuje nedeterministickými rozdvojeními výpočtu na možnosti 0/1. Pak M odpoví někdy Ano, pokud pro některé y je $R(x,y) = 1$, tedy právě pokud $P(x) = 1$. (Což je přesně to, co chceme.)

Naopak pro polynomiální nedeterministický Turingův stroj M řešící nějaký problém P odvodíme polynomiální deterministické zobrazení $R(x,y)$, které každé nedeterministické rozdvojení běhu M nahrazuje deterministickým větvením běhu programu podle nápovědných bitů y. Takže M někdy odpoví Ano, právě když $R(x,y) = 1$ pro některé y, tj. když $P(x) = 1$. □

Nakonec si ukažme, proč předpoklad uvažování pouze rozhodovacích problémů neubírá nijak na teoretické obecnosti našeho uvažování.

Komentář: Představme si například problém P_0, jehož výsledkem $P_0(x)$ má být číslo. Pak lze P_0 teoreticky nahradit posloupností rozhodovacích problémů, které postupně metodou půlení intervalů aproximují výsledek $P_0(x)$.

Fakt 16.6
Každý problém $P : \Sigma^* \to \Sigma^*$ lze nahradit posloupností rozhodovacích problémů $P_1, P_2, \ldots, P_{2k}$ (k závisí na vstupu x), kde $P_{2i-1}(x)$ odpovídá i-tý bit výsledku $P(x)$ a $P_{2i}(x)$ říká, zda i-tý bit výsledku byl poslední.

Fakt 16.7
Všechny rozhodovací verze problémů z P *patří do* NP.

Otázky:

Otázka 16.1: Proč se v definici třídy NP omezujeme jen na rozhodovací problémy?

Otázka 16.2: Hraje v definici třídy NP roli, zda se ptáme na odpověď ANO nebo na odpověď Ne?

Otázka 16.3: Co tedy dostaneme, pokud se v definici analogické třídě NP budeme ptát na nápovědu pro odpověď Ne?

Cvičení 16.4: Problém dominující množiny zjišťuje, zda v daném grafu G existuje podmnožina vybraných k vrcholů takových, že každý další vrchol je s aspoň jedním z vybraných spojený hranou. Proč tento problém patří do třídy NP?

Cvičení 16.5: Proč patří do třídy NP problém, zda daný graf má vrcholovou souvislost méně než k?

Cvičení 16.6*: Proč patří do třídy NP problém, zda daný graf má vrcholovou souvislost naopak alespoň k?

Cvičení 16.7*: Mějme následující problém porovnání barevnosti: Dány jsou dva grafy G, H a otázkou je, zda G má menší barevnost než H. Lze jednoduše tvrdit, že tento problém patří do třídy NP, když napovíme obarvení grafu G méně barvami než obarvení grafu H?

16.2 NP-úplné problémy

Je zřejmé, že PTIME $\subseteq$ NPTIME. Zda je tato inkluze vlastní, tj. zda PTIME $\subsetneq$ NPTIME nebo PTIME = NPTIME je jedním z největších problémů teoretické informatiky (i matematiky obecně).

Většina odborníků se přiklání k první možnosti, tj. že existují problémy, které je možné řešit v polynomiálním čase nedeterministickým algoritmem, ale ne deterministickým algoritmem, ale dosud se to nepodařilo nikomu dokázat.

Existuje celá řada důležitých praktických problémů, u kterých je zřejmé, že patří do třídy NPTIME, ale pro které není znám polynomiální algoritmus. Mezi těmito problémy hrají zvláště důležitou roli tzv. NP-úplné problémy. Jedná se o problémy, které jsou ve třídě NPTIME v určitém smyslu nejtěžší. Pokud platí PTIME $\subsetneq$ NPTIME, pak pro žádný z NP-úplných problémů nemůže existovat polynomiální algoritmus.

Definice 16.8
Problém Q nazveme NP-*těžkým*, pokud každý problém ve třídě NP lze na problém Q převést polynomiálním převodem (oddíl 15.3). Problém Q nazveme NP-*úplným*, pokud je NP-těžký a náleží do třídy NP.

Z této definice je zřejmé, že pokud bychom nalezli efektivní řešení některého (kteréhokoliv) NP-těžkého problému, dostali bychom tím i efektivní řešení všech problémů ve třídě NP.

Jak vlastně poznáme NP-těžké problémy? Pokud již známe nějaký NP-těžký problém, těžkost jiného problému zdůvodníme snadno:

Lemma 16.9
Nechť problém Q je NP-*těžký (*NP-*úplný). Pokud existuje polynomiální převod problému Q na nějaký problém P, pak také P je* NP-*těžký.*

Důkaz: . Označme $R_Q : \Sigma^* \to \Sigma^*$ polynomiální převod Q na P. Dle definice NP-těžkého problému pro každý problém $S \in$ NP existuje polynomiální převod $R_S : \Sigma^* \to \Sigma^*$ problému S na Q. Jelikož složení dvou polynomiálních převodů je opět polynomiálním převodem (tranzitivita), je $R'_S = R_Q \circ R_S$ polynomiálním převodem problému S na problém P. (Nejprve převedeme vstup w pro S na vstup $R_S(w)$ pro Q, pak na vstup $R_Q(R_S(w))$ pro P.) Proto dle definice i P je NP-těžký problém. □

Dávejte si dobrý pozor, v jakém *směru převodu* Lemma 16.9 funguje! Převádí se z problému Q, o kterém je známo, že je těžký, na neznámý problém P. Zjednodušeně řečeno, pokud každý možný vstup NP-těžkého problému Q jsme schopni „přeložit“ na vstup jiného problému P (se zachováním stejné odpovědi), pak P také musí být NP-těžký.

Existence problému, který je „těžší“ než všechny problémy v NP (NP-těžký) je poměrně intuitivní, ale proč by měl takový problém existovat přímo ve třídě NP? To již intuitivní není a objev *prvního* NP*-úplného problému* v 1971 přinesl velkou revoluci do oblasti složitosti algoritmů. Prvním problémem, pro který byla dokázána jeho NP-úplnost byl problém SAT.

Připomeňme si jeho definici:

NÁZEV: SAT (*splnitelnost booleovských formulí*)

VSTUP: Booleovká formule φ.

OTÁZKA: Je formule φ splnitelná?

Věta 16.10 (Cook)
Problém SAT je NP*-úplný.*

Komentář: Obecně řečeno, NP-těžké problémy jsou považovány za „*výpočetně nezvládnutelné*“, a proto pokud takový problém potkáte, ani se nepokoušejte jej přesně řešit (je to jen ztráta času).

Raději v takovém případě zkoušejte *hledat přibližná či částečná řešení*, která uspokojivě odpoví alespoň v některých (dokonce někdy v mnoha) praktických případech.

Všimněme si však jednoho zajímavého háčku – odkud víme, že NP-těžké problémy nelze efektivně řešit? Bohužel to nikdo matematicky zdůvodnit neumí, ale všeobecně se tomu věří, jelikož se již tolik chytrých lidí pokoušelo efektivní řešení NP-úplných problémů najít a *neuspělo*. (A na správné rozřešení je vypsána odměna $1000000.) Přitom nalezení polynomiálního řešení byť jen pro jeden NP-úplný problém by znamenalo (dle definice) polynomiální řešení všech ostatních problémů ve třídě NP.

O některých z výše uvedených problémů je známo, že jsou NP-úplné. Konkrétně se jedná o problémy barvení grafu (CG), nezávislé množiny (IS), Hamiltonovské kružnice (HK) a Subset-Sum.

Co se týká problému isomorfismu grafů, je to příklad problému, u kterého není znám polynomiální algoritmus, ale ani není známo, jestli se jedná o NP-úplný problém. Mezi takové problémy dlouho patřil dříve uvedený *problém prvočíselnosti* (v létě 2002 byl zveřejněn důkaz příslušnosti k PTIME).

Uveďme si příklady ještě několika dalších NP-úplných problémů:

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)?

Příslušnost těchto problémů do třídy NPTIME si čtenář jistě snadno odvodí.

Věta 16.11
(Cook, 1971) Problém SAT je NP-úplný.

Máme-li dokázánu NP-úplnost jednoho problému, je možné ji využít k důkazu NP-úplnosti (či NP-obtížnosti) problémů dalších:

Tvrzení 16.12
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 NP-úplnost několika již uvedených a dalších problémů. Například ukážeme:

- SAT $\triangleleft$ IS
- IS $\triangleleft$ HC
- 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 $\forall\{v_1, v_2\} \in E : c(v_1) \neq c(v_2)$?

Poznámka: Problém 3-SAT je speciálním případem obecnějšího problému SAT a podobně 3-CG je speciálním případem problému CG. Ukazuje se, že tyto problémy zůstávají NP-úplné, i když se omezíme jen na instance určitého konkrétního typu. To může být výhodné při hledání vhodných redukcí při dokazování NP-obtížnosti dalších problémů. Zejména problém 3-SAT je k tomuto účelu často využíván.

Uvažujme ještě následující problém:

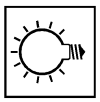
Obrázek 16.1: Příklad grafů, kde Hamiltonovská kružnice existuje, a kde neexistuje

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.

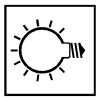


ŘEŠENÝ PŘÍKLAD 16.1: Hamiltonovská kružnice v grafu G je takový podgraf, který je isomorfní kružnici a přitom obsahuje všechny vrcholy G. (Jinak řečeno, kružnice procházející každým vrcholem jednou.) Proč patří do třídy NP problém poznat, zda daný graf G obsahuje Hamiltonovskou kružnici?

Řešení: Jak již bylo řečeno výše u definice, třída NP je vlastně třídou těch problémů, kde odpověď ANO lze ověřit efektivně s vhodnou nápovědou. Jestliže se ptáme na existenci Hamiltonovské kružnice v grafu G, přirozeně se jako nápověda nabízí právě ona kružnice. Pro ilustraci ukazujeme na Obrazku 16.1 příklady dvou grafů, kde v prvním je Hamiltonovská kružnice vyznačena tlustě, kdežto ve druhém neexistuje.

Jak ale Hamiltonovskou kružnici popíšeme a jak ověříme, že se skutečně jedná o Hamiltonovskou kružnici? Obojí musíme zvládnout v polynomiálním čase!

Jako popis Hamiltonovské kružnice se přirozeně nabízí zadat tu permutaci vrcholů grafu G, v jejímž pořadí dotyčná kružnice vrcholy prochází. (Tím neříkáme, že by nebyly jiné způsoby popisu, jen že tento se nám hodí.) Takže nápovědu zadáme jednoduše polem $k[\,]$ délky n, kde n je počet vrcholů G. Pro ověření, že se jedná o Hamiltonovskou kružnici, stačí zkontrolovat, že $k[i] \neq k[j]$ pro různá i, j a že vždy $\{k[i], k[i+1]\}$ je hranou v grafu G pro $i = 1, 2, \ldots, n-1$ a také $\{k[1], k[n]\}$ je hranou. Při vhodné implementaci maticí sousednosti grafu G to zvládneme vše zkontrolovat v lineárním čase, ale i při jiných implementacích nám stačí čas $n \cdot O(n) = O(n^2)$, což je skutečně polynomiální. Proto problém existence Hamiltonovské kružnice patří do třídy NP.



Řešený příklad 16.2: Patří do třídy NP problém poznat, zda daný graf G obsahuje nejvýše čtyři Hamiltonovské kružnice?

Řešení: Čtenář opět může navrhnout, že vhodnou nápovědou pro příslušnost do třídy NP jsou ony čtyři Hamiltonovské kružnice v grafu. To lze přece snadno ověřit stejně jako v předchozím příkladě. Skutečně tomu tak je?

Není! My sice dokážeme ověřit, že napověděné čtyři kružnice v grafu jsou Hamiltonovské, ale nijak tím neprokážeme, že více Hamiltonovských kružnic v grafu není. Takové ověření by nakonec bylo stejně obtížné, jako nalezení Hamiltonovské kružnice samotné.

Proto na základě současných znalostí teoretické informatiky *nelze tvrdit*, že by popsaný problém náležel do třídy NP. Avšak pokud bychom otázku negovali, tj. ptali se, zda graf G obsahuje více než čtyři Hamiltonovské kružnice, tak by už problém do třídy NP náležel. (Napověděli bychom některých pět Hamiltonovských kružnic.) Proto vidíte, jak je důležité správně se v zadání problému ptát.

16.3 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 za to, 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 polynomiálním čase. 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.

16.4 Cvičení

Otázky:

Otázka 16.8: Věta 16.10 dokazuje existenci NP-úplného problému za použití modelu RAM. Znamená to, že při použití jiného modelu algoritmu pro zobrazení R, třeba Turingova stroje, by nám vyšly jiné NP-úplné problémy?

Cvičení 16.9: Problémem 3-obarvení grafu je rozhodnutí, zda existuje korektní obarvení grafu pomocí tří barev. Najděte polynomiální převod problému 3-obarvení na analogický problém 4-obarvení grafu.

Cvičení 16.10: Považujme za známé, že problém 3-obarvení grafu je NP-úplný. Proč je pak NP-úplný problém 4-obarvení?

Cvičení 16.11*: Proč nelze stejně tvrdit, že i problém 2-obarvení je NP-úplný, když přece existuje stejný převod problému 2-obarvení na problém 3-obarvení grafu?

Cvičení 16.12: Rozhodněte, které z následujících problémů patří do třídy P všech efektivně řešitelných problémů.

a) Problém rozhodnout, zda daný graf obsahuje nezávislou množinu (tj. podmnožinu vrcholů nespojených hranami) velikosti 7.

b) Problém rozhodnout, zda daný graf obsahuje nezávislou množinu (tj. podmnožinu vrcholů nespojených hranami) velikosti nejméně 2005.

c) Problém rozhodnout, zda daný graf má barevnost nejméně tři.

d) Problém rozhodnout, zda daný graf má barevnost nejvýše tři.

e) Problém rozhodnout, zda daný graf má barevnost přesně tři.

f) Problém rozhodnout, zda daný graf má barevnost přesně dva.

Cvičení 16.13:Párováním v grafu rozumíme podmnožinu hran, které nesdílejí žádný svůj koncový vrchol. Jak byste polynomiálně převedli problém nalezení párování velikosti p v grafu G na problém nezávislé množiny?

Cvičení 16.14*:Dokážete najít převod problému dominující množiny na vrcholové pokrytí?

Cvičení 16.15:Patří do třídy NP problém zjistit, zda graf G obsahuje dvě Hamiltonovské kružnice, které nesdílí žádnou hranu?

Cvičení 16.16:Patří do třídy NP problém zjistit, jaká je barevnost grafu?

Cvičení 16.17:Patří do třídy NP problém zjistit, zda graf G je rovinný? A co třeba negace tohoto problému?

Cvičení 16.18*:Je známo, že do třídy NP patří problém k-obarvení (zda graf lze obarvit korektně k barvami, tj. zda barevnost je $\leq k$) pro všechna k. Patří ale do třídy NP problém zjistit, zda graf G má barevnost právě k? Pro která k?

Cvičení 16.19:Rozhodněte, které z následujících problémů patří do třídy NP:

a) Problém rozhodnout, zda daný graf má barevnost nejvýše čtyři.

b) Problém rozhodnout, zda daný graf má barevnost přesně čtyři.

c) Problém rozhodnout, zda daný graf má barevnost nejméně čtyři.

d) Problém rozhodnout, zda daný graf obsahuje nejméně tři Hamiltonovské kružnice.

e) Problém rozhodnout, zda daný graf obsahuje přesně tři Hamiltonovské kružnice.

Příloha A

Vyhledávání z pohledu programátora

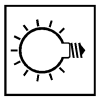
Komentář: Vyhledávání zadaných řetězců v textu (často v rozsáhlém, třeba v balíku celých zdrojových kódů systému) je častou a oblíbenou činností v programátorské praxi. Jak je ale takové vyhledávání implementováno, aby bylo rychlé i na velmi rozsáhlých textech?

Není ideální začínat na každé pozici souboru zvlášť hledat celé slovo, protože často tak budeme číst stejné znaky vícekrát po sobě. To by znamenalo velký problém pro datová média s fyzicky sekvenčním přístupem. (Zkušený čtenář jistě hned namítne, že většina takovým problémů se automaticky vyřeší kešováním vstupu, ale stále to znamená zbytečné paměťové operace navíc, přitom při rozsáhlém hledání je každá sekunda drahá.) Pomocí konečného automatu, jak jsme popsali v předchozí části, však lze vyhledávání implementovat se striktně sekvenčním přístupem ke vstupním znakům (jeden po druhém, každý jen jednou).

Jinou věcí je, že často chceme vyhledat ne jeden fixní řetězec textu, ale nějaký obecnější vzorek, který může nabývat „různých podob“. Jak lze takový proměnný vzorek vůbec symbolicky zadat? Toho se obvykle dosahuje použitím tzv. *regulárních výrazů*, které mají na různých výpočetních platformách a v různých programech různou podobu a syntaxi, ale jejich obecný smysl je (téměř) jednotný.

V této části si místo suché teorie prakticky ukážeme dva velmi mocné ná-

stroje na vyhledávání a zpracovávání v textu, známé především z unixových systémů.



Řešený příklad 1.1: Vyhledejme ze zdrojových kódů v jazyce C všechny řádky obsahující přiřazení celých čísel do proměnné `i`.

Řešení: Pro vyhledávání tohoto typu je přímo stvořený příkaz `grep`. Jednoduché řešení příkladu je třeba toto:

```
grep '\<i *= *[0-9]\+;' *.c
```

Co však tento příkaz znamená?

- Zjednodušená syntaxe příkazu je '`grep` regexp files', kde regexp je zkratka označující regulární výraz, zde

  '`\<i *= *[0-9]\+;`'.

- Část

  '`\<i`'

  říká, že požadujeme vyhledání znaku `i`, který je začátkem svého slova (tj. vyloučíme například proměnné `bi=6`).

- Poté ' `*=`' říká, že může následovat libovolné opakování mezer, třeba i žádná, následované znakem '='.

- Po dalším volitelném opakování mezer vidíme

  '`[0-9]\+`',

  což je výčet zde povolených znaků – číslic, opakovaných jednou nebo vícekrát.

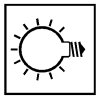
- Na konci `;` ukončuje příkaz přiřazení v C.

Pozorný čtenář nyní může namítnout několik nedostatků. Třeba jak najdeme čísla se znaménkem? Co když na konci přiřazení ještě budou další mezery? A co když přiřazení bude ukončeno čárkou místo středníkem? Pro takové případy můžeme náš vyhledávací příkaz ještě zobecnit o volitelné znaménko a volitelné zakončení:

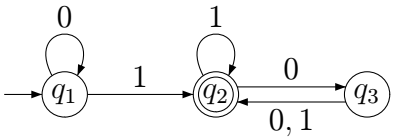
```
grep '\<i *= *[-+]\?[0-9]\+ *[;,]' *.c
```

Poznámka: Pokud bychom takto vybrané řádky z textu nejen rádi viděli, ale i chtěli zpracovávat, možným a vhodným nástrojem by byl třeba klasický skriptový jazyk `awk`.

Ještě obecněji si můžeme představit program filtrující a upravující vstupní text podle zadaných (i složitých) pravidel– zde již nejde o jednoduchý příkaz, ale o komplexní programátorský úkol. Pro jeho tvorbu je volně k dispozici metaprogramovací jazyk `flex`. (Slovem „metaprogramovací“ myslíme, že `flex` překládá daná pravidla do zdrojového kódu C programu, který poté můžeme zařadit do svých projektů.) Zhruba řečeno, `flex` pracuje jako velmi zobecněný automat, který podle vstupního textu přechází mezi svými vnitřními stavy a v nich tento text dále zpracovává.



Řešený příklad 1.2: Simulujte na počítači následující automat:



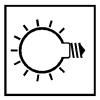
Řešení: Bez dlouhých řečí ukážeme, jak jsou přechody tohoto automatu zapsány přechodovými pravidly jazyka `flex`:

```
<INITIAL>0      ;
<INITIAL>1      BEGIN(Q2);
<Q2>1           ;
<Q2>0           BEGIN(Q3);
<Q3>0|1         BEGIN(Q2);

 <Q2><<EOF>>     printf("slovo přijato!"); return;
 .|\n            printf("neznámý znak %s na vstupu",yytext);
```

(Počáteční stav q_1 je zde nazýván „initial“. Plný text tohoto programu, tj. včetně nezbytných hlaviček a startovního kódu, je uveden ve cvičení – Příklad.)

Poznámka: V obecnosti `flex` umí ze vstupu načítat kromě jednotlivých znaků i celé řetězce popsané regulárními výrazy najednou. Při každém z nich kromě přechodu vnitřního stavu umožní načtený řetězec zpracovat libovolným kódem v C. Pro případné další (vyšší) zpracování textu čteného `flex`em poslouží třeba nástroj `yacc`.



Řešený příklad 1.3: Najděte v platném zdrojovém kódu jazyka C všechny `for` cykly používající řídící proměnnou `i`, tj. `i` je v hlavičce cyklu inkrementováno. (Pozor, nestačí hledat výskyty '`for (i=0`', neboť cyklus může vypadat třeba takto '`for (i=0,j=1;j<5;j++)`'.)

Řešení: Použijeme základní regulární výrazy knihovny `regexp` (ty se liší od rozšířených v zásadě jen syntaxí – použitím backslashu). Řídící proměnnou cyklu poznáme nejlépe podle toho, že je inkrementována ve třetí středníkem oddělené sekci hlavičky příkazu `for(;;)`. (Pro jednoduchost uvažujeme, že program je slušně napsán a celá hlavička cyklu je na jednom řádku.)

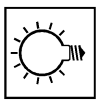
Začneme vyhledáním začátku třetí sekce v hlavičce `for`, tj.:

```
'\<for *([^;)]*;[^;)]*;'
```

Zde `[^;)]` znamená výčtový typ všech znaků, které nejsou `;` ani `)`. Všimněme si, že před `for` musíme kontrolovat, že je to začátek slova `\<` a ne nějaký identifikátor jako třeba `xxxfor`. Pak následuje vyhledání řetězce `i++` nebo `++i`. Opět si musíme dát pozor, že identifikátor `i` není začátkem nebo koncem delšího jména. Navíc ještě mějme na paměti, že před `i++` může být jiný výraz oddělený čárkou. (Znáte úplný význam čárky v syntaxi jazyka C?) Takže celkem řešení vyjde:

```
grep '\<for *([^;)]*;[^;)]*;[^)]*\(++i\>\|\<i++\)' *.c
```

Zde `\(..\)` neznamená výskyt skutečných závorek v textu, ale závorkuje příslušný regulární podvýraz uvnitř. Tam je pomocí znaku `\|` (nebo) řečeno, že nalézt se má `++i` nebo `i++`. To je vše (až na extrémně degenerované případy).



Řešený příklad 1.4: Najděte v daném textu všechny výskyty úplných (a pokud možno platných) e-mailových adres, oddělených mezerami či konci řádků.

Řešení: Jak jistě víte, e-mailové adresy se typicky vyznačují znakem `@` někde uvnitř. Co může být před ním a za ním? Běžně se tam vyskytují písmena, číslice a tečka. Pro větší obecnost ještě zahrneme znaky `-_`. Takže náš regulární výraz bude následovný:

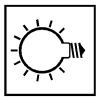
```
'[a-zA-Z0-9-_.]\+@[a-zA-Z0-9-_.]\+'
```

Je to ale přesně, co po nás úloha chtěla? Na jednu stranu tento regulární výraz rozezná všechny normální platné e-mail adresy, ale na druhou stranu vezme třeba i slovo `x@@y@z.t`. (To proto, že se rozpozná sufix `y@z.t`, ale nezkontroluje se, že před ním je více neoddělených znaků.)

Proto musíme před i za výraz rozpoznávající adresu dát mezery nebo speciální symboly `^ $` rozpoznávající začátek a konec řádku. Celý výraz pak vypadá:

```
'\( \|^\)[a-zA-Z0-9-_.]\+@[a-zA-Z0-9-_.]\+\( \|$\)'
```

Vyzkoušejte si to na počítači sami!



Řešený příklad 1.5: Navrhněte regulární výraz, který zkontroluje, jestli vstupní text (celý od začátku do konce řádku) vypadá jako platná e-mail adresa.

Řešení: Použijeme obdobné úvahy jako v předchozím příkladě, ale navíc se zamyslíme, jak by měla vypadat úplná a platná doména – mít alespoň dvě složky a poslední by měl být dvoupísmenný kód národní domény nebo některé speciální vrchní domény jako třeba „.edu“. Výsledek nyní bude (spojte si oba řádky dohromady):

```
'^[a-zA-Z0-9-_.]\+@[a-zA-Z0-9-_.]\+[.]

  \([a-zA-Z][a-zA-Z]\|com\|edu\|gov\|mil\|info\)$'
```

(Již neuvažujeme mezery před nebo za adresou, viz. zadání.)

Cvičení 1.1*:Zapište regulární výraz pro `grep` hledající všechny ty řádky zdrojového kódu C, na kterých je proměnná `xyz`, ale ne uvnitř řádkového komentáře `//...`.

Příloha B

Řešení příkladů

Otázka 1.1: Ano, přestože se jim říká „reálná“ čísla, mají omezený rozsah i přesnost, proto mohou nabývat jen konečně mnoha hodnot.

Otázka 1.2: Není, je jich nekonečně mnoho, jak byste měli znát z matematiky.

Otázka 1.3: Je, lze je přece seřadit podle velikosti do jedné posloupnosti.

Otázka 1.4: Třeba $0, 1, -1, 2, -2, 3, -3, \ldots$.

Otázka 1.5*: Čísla tvaru $\frac{p}{q}$ seřadíme nejprve do skupin podle součtu $|p|+|q|$ a pak uvnitř jednotlivých skupin podle hodnot q.

Cvičení 1.6:

a) Lichá přirozená čísla

b) Sudá celá čísla

c) Sudá přirozená čísla

d) Přirozená čísla dělitelná 6

e) Neobsahuje žádné prvky, jedná se o prázdnou množinu

Cvičení 1.7:

a) $\{1, 10, 100\}$

b) $\{n \in \mathbb{Z} \mid n > 5\}$

c) $\{n \in \mathbb{N} \mid n < 5\}$

d) $\emptyset$

e) $\{Y \mid Y \subseteq X\}$

Cvičení 1.8:

a) Ne.

b) Ano.

c) $\{x, y, z\}$ neboli množina A.

d) $\{x, y\}$ neboli množina B.

e) $\{(x, x), (x, y), (y, x), (y, y), (z, x), (z, y)\}$

f) $\{\emptyset, \{x\}, \{y\}, \{x, y\}\}$

Cvičení 2.1:

a) $aaa, aab, aba, abb, baa, bab, bba, bbb$

b) $ababbabbabba$

c) $00, 01, 10$

d) $\varepsilon, 1, 00, 001, 0010$

e) $\varepsilon, 0, 10, 010, 0010$

Otázka 2.2: Ne, není totiž konečná, což je důležitá podmínka.

Otázka 2.3: Ano, přeneseně, třeba jako jazyk všech slov nad abecedou $\{0, 1, \ldots, 9\}$, která nezačínají nulami.

Otázka 2.4: Nelze.

Otázka 2.5: Velký – prázdný jazyk nemá v sobě žádné slovo, je to prázdná množina, kdežto prázdné slovo je slovem jako každé jiné, jen má nulovou délku.

Otázka 2.6*: Skoro nikdy, jen pokud je L prázdný nebo obsahuje jen prázdné slovo. Jinak vždy vytvoříme opakováním nekonečně mnoho slov v L^*.

Otázka 2.7*: ...

Cvičení 2.8: Slova ε, 10 a celé slovo 101110110.

Cvičení 2.9: $\{11001, 110000, 011101, 0111000\}$

Cvičení 2.10:

- $L_1 \cup L_2 = \{\varepsilon, a, b, aa, bb, aaa\}$
- $L_1 \cap L_2 = \{\varepsilon, b, aa, aba, abba, baab\}$
- $L_1 - L_2 = \{aab, baa, aabb, abab, baba, bbaa\}$
- $\overline{L} = \{a, ab, ba, bb, aaa, abb\}$

Cvičení 2.11: Třeba $L_1 = \{\varepsilon\}$ a $L_2 = \{1\}$.

Cvičení 2.12*: Ne, jen všechna ta slova, co nekončí lichým počtem 0.

Cvičení 2.20: $\{0, 001, 00101, 0010101, 111, 11101, 1110101\}$

Cvičení 2.21: Je to jazyk všech těch slov, která mají úseky nul sudé délky a úseky jedniček délky dělitelné třemi.

Cvičení 2.22: Slova ze samých nul nebo ta slova, která mají jediný znak 1 právě uprostřed, tj. ε, 0, 00, 000, ..., 1, 010, 00100, ...

Cvičení 2.23: Třeba pro $L_1 = \{1\}$, $L_2 = \{11\}$ a $L_3 = \{1, 11\}$ vyjde $L_1 \cap L_2 = \emptyset$, ale $111 \in L_1 \cdot L_3$ i $111 \in L_2 \cdot L_3$.

Cvičení 2.25*: Je to jazyk všech těch slov w, ve kterých rozdíl počtů výskytů a a b počítaný na prefixech w dosáhne svého maxima uvnitř w, tj. ne na začátku ani ne na konci slova w.

Cvičení 2.26*: Nejméně 12, zdůvodněte si, proč ne méně! Pro které dva jednoduché jazyky se minima dosahuje?

Cvičení 2.27: Protože 3 dvoupolohové přepínače se mohou nacházet jen v $2^3 = 8$ celkem kombinacích poloh.

Cvičení 3.4:

- $(5, bbaab) \vdash (4, baab) \vdash (4, aab) \vdash (7, ab) \vdash (7, b) \vdash (4, \varepsilon)$
- Přijímá $a, b, aa, bb, aaa, aba, bab, bbb$
-
- Jedná se o slova, která začínají a končí stejným symbolem.

Otázka 3.5: Může, potom bude přijato i prázdné slovo, pro nějž výpočet skončí již v počátku.

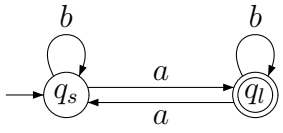
Otázka 3.6: Ano, automat nemusí mít žádný přijímající stav nebo přijímající stav nemusí být dosažitelný.

Otázka 3.7: Ano, je.

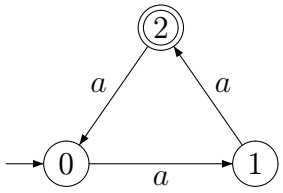
Cvičení 3.8:

	a	b
→1	2	3
←2	2	4
←3	5	3
4	2	4
5	5	3

Cvičení 3.10: Čtení znaku b nemění stav:



Cvičení 3.11: Počítání na cyklu délky 3:

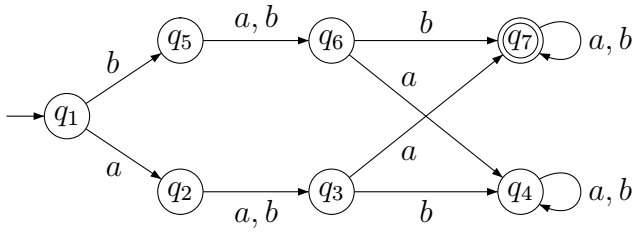


Cvičení 3.12: Právě taková, ve kterých počet výskytů znaku a mínus počet b dává zbytek 2 po dělení 3.

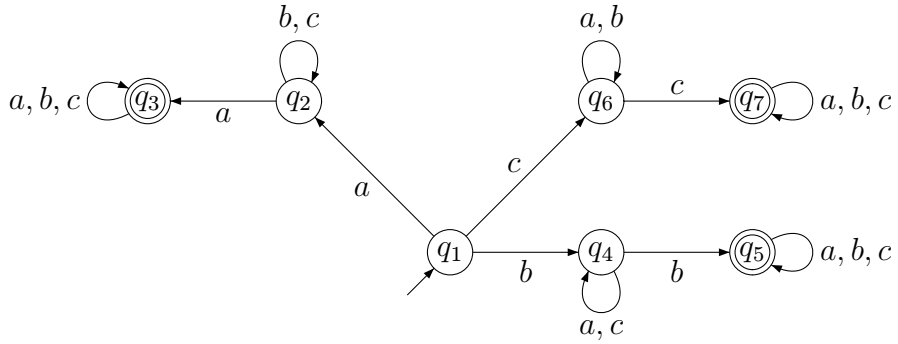
Cvičení 3.13: Všechna taková, ve kterých po prvním výskytu znaku a následuje sufix „a“, „aa“ nebo „b“ (a nic víc).

Cvičení 3.14: Prostřední a ten vpravo. Automat vlevo se dostává do přijímajícího stavu jen když délka dává zbytek 2 po dělení 3.

Cvičení 3.15:



Cvičení 3.16:



Cvičení 3.17: Velmi jednoduše – vyměníme přijímající stavy F za jejich doplněk $Q - F$.

Otázka 3.20: Ano, stačí jen vhodně pozměnit definici přijímajících stavů F.

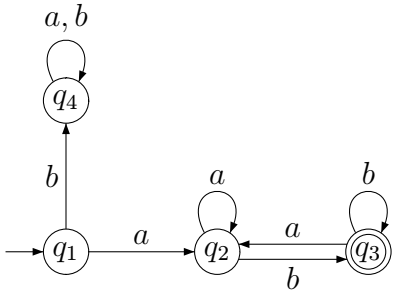
Otázka 3.21*: Například všechna slova z a, b, která mají stejně mnoho a jako b. (Konečný automat si je nemůže „spočítat“ kvůli omezenosti své „paměti“ ve stavech.)

Cvičení 3.23: Ano, počítáme paritu výskytů pro a i b zvlášť podle Příkladu 3.1 a uděláme sjednocení těchto dvou jazyků.

Cvičení 3.24: Ano, obdobně jako v předchozí úloze, jen pozměníme přijímající stavy. Nakreslete si to a ověřte!

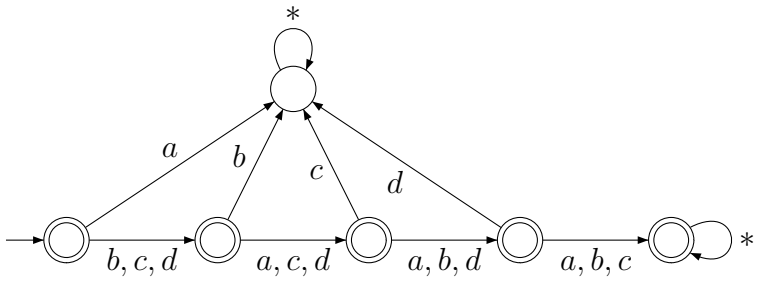
Cvičení 3.25: Ano, ale už se nám tento automat bude obtížně kreslit, protože má 101 stavů „počítajících“ prvních 100 znaků, pak jsou již všechna delší slova přijata.

Cvičení 3.26:

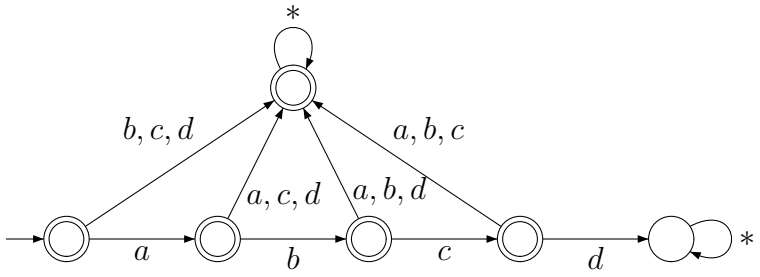


Cvičení 3.27: Není, snadno je vidět, že počáteční stav 1 lze sloučit se stavem 12. (Později si uvedeme více o tzv. minimalizaci automatu.)

Cvičení 3.28:



Cvičení 3.29:



Cvičení 3.30: Nejmenší takový KA má a) 8, b) 7 stavů.

Cvičení 3.31: Nejmenší takový KA má 5 stavů.

Otázka 4.5: Nemusí být – definice nám povoluje se neustále přesouvat po ε-přechodech a nečíst vstup. Takový výpočet však nikdy k přijetí slova nevede, takže pro nás nemá praktický význam.

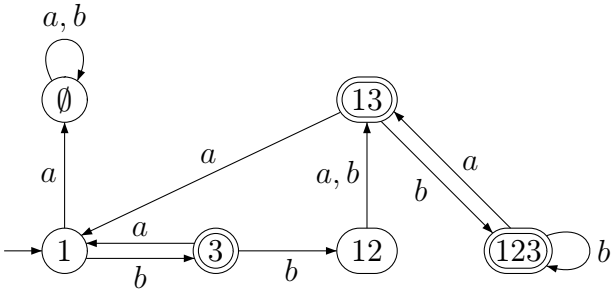
Otázka 4.6: V podstatě nic – všechny vzniklé stavy budou reprezentované jednoprvkovými podmnožinami původních stavů, takže jim budou přímo odpovídat, bude to stejný automat.

Otázka 4.7: Právě tehdy, pokud některá posloupnost vstupních znaků nemá definovány všechny své přechody v nedeterministickém automatu.

Otázka 4.8: Pochopitelně nemůže, výpočty s nedefinovanými přechody mají dle definice přijímaného jazyka selhat, tj. nepřijmout.

Cvičení 4.9: Nikdy, již první přechod znakem a není definován.

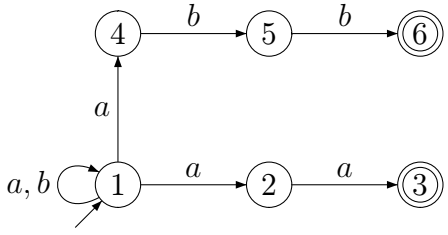
Cvičení 4.10: Zde:



Cvičení 4.11: Vždy kromě ε a „bb“, viz sestrojený deterministický automat.

Cvičení 4.12*: Není lehké, že? V prvé řadě ta slova začínají vždy b. Pak zbytek (po prvním znaku) těch přijímaných slov lze rozdělit na úseky, kde každý úsek je buď ab, nebo ba, nebo bb, nebo se za jistých okolností může b vyskytnout samotné. Krátký slovní popis asi není možný.

Cvičení 4.13: Přirozeně využijeme nedeterminismu:



Cvičení 4.15: Třeba bab.

Cvičení 4.16: Třeba abb.

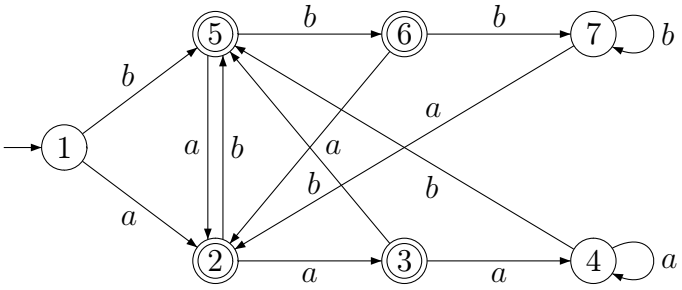
Cvičení 4.17: 5 stavů

Cvičení 4.18: 6 stavů

Cvičení 4.19*: Slova začínají b, končí ba, opakují se $\leq 2\times$ a za sebou.

Cvičení 5.1: V podstatě ano, jen počáteční stav u formálně sestrojeného automatu je navíc.

Cvičení 5.2: Toto:



Cvičení 5.3: Je to jazyk všech slov se sufixy „ba“, „ab“, „baa“ nebo „abb“ (ten první znak musíme explicitně uvést, aby bylo jasné, že více opakování předtím nebylo!), plus navíc krátká slova a, b, aa, bb.

Otázka 5.4: Nejednoznačný, třeba $(0+1)^*$ a $(0+00+1)^*$ označují stejný jazyk.

Cvičení 5.5: Třeba takto $a^*(c+\varepsilon)b^*$.

Cvičení 5.6: Třeba takto $aa^*(c+\varepsilon)b^*b$.

Cvičení 5.7: Třeba takto $aa^*(cc+c+\varepsilon)b^*b$.

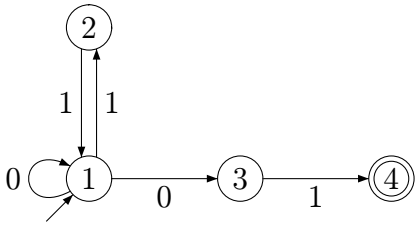
Cvičení 5.8: Nejsou, třeba první jazyk obsahuje prázdné slovo, kdežto druhý ne.

Cvičení 5.9: Nejsou, třeba slova v prvním jazyku mohou končit znakem 1, kdežto v druhém jazyku ne.

Cvičení 5.10*: Např. $(00 + 100 + 010 + 1010)^*$

Otázka 5.13: Není to jednoduché, ale je to aspoň možné. Z regulárních výrazů sestrojíme automaty, pro ně uděláme průnik a z výsledku zpět odvodíme regulární výraz.

Cvičení 5.14: Zde:



Cvičení 5.15: Jen přidáme smyčku s 0 nad stav 3.

Cvičení 5.16: Třeba $0^*1(1 + 00 + 01)^*$.

Cvičení 5.17: Třeba $(a + b)(a + b)(((a + b)(a + b) + b)(a + b))^*$. Už to není tak jednoduché, že?

Cvičení 5.18: $(\varepsilon + 1 + 11)(01 + 011 + 001 + 0011)^*(\varepsilon + 0 + 00)$

Cvičení 5.19: $(((b + c)^* + (a + c)^*)c)^*(b^* + a^*)$

Cvičení 5.20: $((b + c + a(b + c))^*(\varepsilon + a)$

Cvičení 5.21: c^*

Cvičení 5.22: $c^*(abb^* + b^*)^*$

Cvičení 5.23*: $((c+\varepsilon)(a+b)(a+b)^*)^*(c+\varepsilon)\,aa\,((c+\varepsilon)(a+b)(a+b)^*)^*(c+\varepsilon)$

Cvičení 5.24*:

a) Nejkratší je ε a nejdelší 01100110, neboť jazyk L nedovoluje opakovat stejný znak za sebou více než dvakrát.

b) Protože $1 \in K - L$ a $010101 \in L - K$.

c) 10101 jednoznačně.

Cvičení 5.25: $(\varepsilon + aa + ba)a^*(abaa^*)^*$

Otázka 6.2: Protože ii vznikl sloučením stavů $4, 5$ a mezi nimi vedly přechody znakem 1. Nyní se tyto dva přechody sloučí do jedné smyčky.

Otázka 6.3: Neukáže, naopak bude postup obvykle dosti dlouhý, protože bude muset rozložit množinu stavů až na jednotlivé jednoprvkové podmnožiny.

Otázka 6.4*: Protože bychom v prvé řadě do symbolické přechodové tabulky nemohli zapsat jednoznačné stavy. (Sice by se mohlo zobecňovat na množiny stavů ...)

Cvičení 6.7: Sloučí se stavy $1, 3$ do jednoho.

Cvičení 6.8: Ano, postupem minimalizace začneme s rozkladem $\{\{1, 3\}, \{2\}\}$ a hned v první iteraci rozlišíme stavy $1, 3$ přechodem znakem 0.

Cvičení 6.9: Ano, již po dvou iteracích dojde k rozlišení všech stavů.

Cvičení 6.11: 4 stavy, je nutno počítat paritu počtů jak a, tak b.

Cvičení 6.12: Samozřejmě ne, ten první nepřijímá prázdné slovo, kdežto druhý ano.

Cvičení 6.13: Ano, ten druhý se minimalizuje na stejný jako první.

Cvičení 6.14: Již po dvou iteracích postupu minimalizace dojde k rozložení na jednotlivé stavy zvlášť.

Cvičení 6.15: Spojit 12; 348; 567.

Cvičení 6.16: Spojit 12; 348; 5; 67.

Cvičení 6.17: 5 stavů, 2×2 jsou nutné k rozpoznávání parity počtů a a b jeden další pro odlišení prázdného slova.

Cvičení 6.18: Vyjdeme ze základního automatu na $3 \times 3 = 9$ stavech, který počítá výskyty a a b do dvou (tj. jako 0, 1, mnoho) a na vhodných místech má přijímající stavy. Tento automat pak minimalizujeme. Výsledek má 7 stavů.

Cvičení 6.19: Obdobně předchozímu 5 stavů.

Cvičení 6.20: Obdobně předchozímu 9 stavů.

Cvičení 7.1: Není, stačí se podle Pumping lemmatu podívat na slovo „$a^n b^n$“ jako zřetězení uvw, kde v může být složeno jen ze samých a, a proto už uv^2w nepatří do našeho jazyka.

Cvičení 8.1: Přidáme symbol G pro nejvyšší prioritu $E \longrightarrow E + E \mid F$, $F \longrightarrow F \times F \mid G$, $\quad G \longrightarrow (E) \mid G^2 \mid a$

Cvičení 8.2: Jako první odvození přidáme $P \longrightarrow E \mid E = E$, což znamená, že výraz může být buď bez porovnání nebo s jedním porovnáním dvou aritmetických podvýrazů.

Cvičení 8.3:

- $S \Rightarrow AaB \Rightarrow AAaB \Rightarrow aSbAaB \Rightarrow abAaB \Rightarrow abaB \Rightarrow abaSA \Rightarrow abaAaBA \Rightarrow abaaBA \Rightarrow abaaacA \Rightarrow abaaacSB \Rightarrow abaaacB \Rightarrow abaaacac$
- $S \Rightarrow AaB \Rightarrow AaSA \Rightarrow AaSSB \Rightarrow AaSSac \Rightarrow AaSac \Rightarrow AaAaBac \Rightarrow AaAaacac \Rightarrow Aaaacac \Rightarrow AAaaacac \Rightarrow Aaaacac \Rightarrow aSbaaacac \Rightarrow abaaacac$
- Např. $w_1 = \varepsilon, w_2 = ab, w_3 = aaacac$

- $$\begin{array}{lcl} S & \longrightarrow & AaB \mid \varepsilon \\ A & \longrightarrow & AA \mid AaB \mid \varepsilon \mid aSb \mid SB \\ B & \longrightarrow & SA \mid ac \end{array}$$

Otázka 8.5: Není, třeba pravidlo $S \longrightarrow SS \mid \varepsilon$ generuje jen prázdné slovo, ale lze jej odvozovat pře libovolně mnoho kroků prvního pravidla a následným dosazením prázdných slov všude.

Otázka 8.6: Prázdný, protože neterminálu S se nijak nezbavíme a generovaná slova musí být složená jen z terminálů.

Otázka 8.7: Prohlásíme stavy automatu za neterminály. Pro každý stav Q a stav $Q' = \delta(Q, x)$ přidáme pravidlo $Q \longrightarrow xQ'$.

Cvičení 8.8: Chybí v něm slova liché délky, takže obecněji všechna $S \longrightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$.

Cvičení 8.9: $S \longrightarrow \varepsilon \mid 0S0 \mid T, \qquad T \longrightarrow \varepsilon \mid 1T.$

Cvičení 8.10: Prázdný, protože se nikdy nezbavíme výskytu neterminálu S nebo C.

Cvičení 8.11: Ne, druhá generuje třeba slovo „$abba$“, které v první nelze.

Cvičení 8.14: Regulární A i B, bezkontextový C.

Cvičení 8.15*: Regulární A, bezkontextový B, ani jedno pro C.

Cvičení 8.16: Snadno $S \longrightarrow aSb \mid Sb \mid b$.

Cvičení 8.17: $S \to aaSaa \mid abSba \mid baSab \mid bbSbb \mid \varepsilon$

Cvičení 8.18*: $S \to aTa|bTb|\varepsilon$, $T \to aUa|bUb|a|b$, $U \to aSa|bSb$

Cvičení 8.19: Negeneruje, neboť z druhé gramatiky nezískáme slovo ε.

Cvičení 8.20: Negeneruje, neboť z druhé gramatiky nezískáme slovo $aaaabb$.

Cvičení 8.21: b)

Cvičení 8.22: b)

Cvičení 8.23: a),b)

Cvičení 8.25*: Třeba

$$\begin{array}{lcl} S & \longrightarrow & bS \mid cS \mid TFS \mid T \mid \varepsilon \\ T & \longrightarrow & aTbb \mid abb \\ F & \longrightarrow & c \mid T \end{array}$$

Cvičení 8.26: Po odstranění neterminálů, které není možno přepsat na terminální slovo dostaneme gramatiku:

$$\begin{array}{lcl} S & \longrightarrow & aSb \mid aDaS \mid \varepsilon \\ C & \longrightarrow & CC \mid cS \\ D & \longrightarrow & aSb \mid cD \mid aEE \\ E & \longrightarrow & bD \end{array}$$

Po odstranění nedosažitelných neterminálů:

$$\begin{array}{lcl} S & \longrightarrow & aSb \mid aDaS \mid \varepsilon \\ D & \longrightarrow & aSb \mid cD \end{array}$$

Cvičení 8.27:

$$\begin{array}{lcl} S & \longrightarrow & aSB \mid aB \\ B & \longrightarrow & b \end{array}$$

Otázka 9.2: Například tak, že přechodová funkce obsahuje pravidlo $\delta(q_0, Z, \varepsilon) \ni (q, \varepsilon)$ pro počáteční stav q_0, počáteční zásobníkový symbol Z_0 a nějaký stav Q.

Otázka 9.3: Jednoduše, všechny přechody obyčejného automatu převezmeme beze změny, jenom přidáme pravidla, že při vstupu do přijímajícího stavu se nedeterministicky může (jediný!) zásobníkový symbol odstranit.

Cvičení 9.4: Vezměme regulární jazyk L_0 daný výrazem $a^*b^*c^*$ a utvořme průnik $L = L_0 \cap L_3$. Pokud by byl L_3 bezkontextový, byl by takový i L podle Věty 9.4, ale L je přece jazyk z Příkladu 9.1.

Cvičení 9.5: $Q = \{p, q\}, \Sigma = \{a, b, c\}, \Gamma = \{A, B, D\}$, počáteční zásobníkový symbol je D,
$\delta(p, a, D) = \{(p, A)\}$
$\delta(p, b, D) = \{(p, B)\}$
$\delta(p, a, A) = \{(p, AA)\}$
$\delta(p, b, A) = \{(p, BA)\}$
$\delta(p, a, B) = \{(p, AB)\}$
$\delta(p, b, B) = \{(p, BB)\}$
$\delta(p, c, A) = \{(q, A)\}$
$\delta(p, c, B) = \{(q, B)\}$
$\delta(p, c, D) = \{(q, \varepsilon)\}$
$\delta(q, a, A) = \{(q, \varepsilon)\}$
$\delta(q, b, B) = \{(q, \varepsilon)\}$
$\delta(q, a, B) = \emptyset$
$\delta(q, b, A) = \emptyset$

Cvičení 9.6: Slova $w \in \{a, b, c\}^*$ taková, že po vynechání všech výskytů symbolu c z w dostaneme slovo ve tvaru $v(v)^R$.

Cvičení 9.7: Stejný jazyk jako v předchozím příkladě, tedy slova $w \in \{a, b, c\}^*$ taková, že po vynechání všech výskytů symbolu c z w dostaneme slovo ve tvaru $v(v)^R$.

Cvičení 9.8: $Q = \{q\}, \Sigma = \{+, *, (,), a\}, \Gamma = \{A, B, C, a, (,), +, *\}$, počáteční zásobníkový symbol je A,

$\delta(q, \varepsilon, A) = \{(q, A + B), (q, B)\}$
$\delta(q, \varepsilon, B) = \{(q, B * C), (q, C)\}$
$\delta(q, \varepsilon, C) = \{(q, (A)), (q, a)\}$
$\delta(q, a, a) = \{(q, \varepsilon)\}$
$\delta(q, (, () = \{(q, \varepsilon)\}$
$\delta(q,),)) = \{(q, \varepsilon)\}$
$\delta(q, +, +) = \{(q, \varepsilon)\}$
$\delta(q, *, *) = \{(q, \varepsilon)\}$

Otázka 10.2: Libovolně velký, může se přesunout, jak daleko chce.

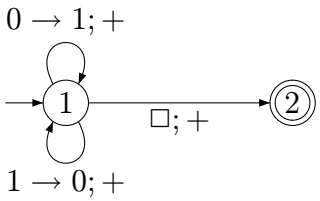
Cvičení 10.3: Využije se podobná implementace jako v Řešeném příkladu 10.2. Pokud slovo není palindrom, stroj se nezastaví, nýbrž skončí v nekonečné smyčce na jednom pomocném stavu.

Cvičení 10.4: Zastaví na všech slovech kromě ε. Poslední znak počátečního úseku samých a (pokud tam je) je změněn na b.

Cvičení 10.5*: ...?

Cvičení 10.6*: Zastaví jen na $(a + b)a^*$, přepisuje na ab^*.

Cvičení 10.9:



Cvičení 10.10: Přidáme jako počáteční nový stav, který se po znacích $0, 1$ posouvá doprava (bez přepisování) a na prvním prázdném vpravo se vrátí o jeden znak doleva, a pak už pokračujeme jako v Příkladě 10.1.

Otázka 11.3: Vyhradíme volný úsek paměti délky k a počátku p a potom budeme k prvku $a[j]$ přistupovat jako k paměťovému místu na adrese $p + j$.

Otázka 11.4: Vyhradíme volný úsek paměti délky $k\ell$ a počátku p. Přistupovat budeme k $a[i,j]$ na adrese $p + i\ell + j$.

Otázka 11.5: Musí si ve vymezeném úseku paměti implementovat zásobník pro lokální proměnné a návraty. (Však stejně tak to dělá běžný CPU.)

Cvičení 11.7: TS nejvíce času ztrácí na přístupu do paměti – pro přístup k proměnné na adrese ℓ musí vykonat až ℓ posunů hlavy, než se na toto místo dostane, kdežto RAM přistoupí na adresu ℓ přímo.

Cvičení 11.8: 2 kroky pro w zač. a, jinak počet b plus 3× počet a plus 1.

Cvičení 11.9: Délka plus 1 pro w bez b, jinak počet a na začátku krát 2 plus 2.

Cvičení 12.1: Nejsou, protože doplněk takového jazyka znamená, že by ten doplňkový Turingův stroj musel rozpoznat, kdy se ten první stroj (ne)zastaví, což víme, že je nemožné.

Cvičení 12.2: Nelze, neboť by to znamenalo přinejmenším vyřešit i problém zastavení daného programu. Proto lze správnost programu jen odhadovat.

Otázka 13.1: Vůbec nic, značka $O()$ se týká rychlosti růstu, ne srovnání konkrétních malých hodnot.

Cvičení 13.2: (c)

Cvičení 13.3: $\sqrt{n} = \Omega(\log n)$, přesněji roste striktně rychleji.

Cvičení 13.4*: Druhá, neboť $(\log n)^{\log n} = n^{\log\log n} \gg n^5$.

Cvičení 13.5*: (a)

Otázka 13.7: Ano, číslo n potřebuje ke svému zápisu zhruba $\log n$ bitů, což je zanedbatelně málo vzhledem k rozsahu problému. Proto (i v souladu s praxí) lze rozsah čísla n zanedbat.

Otázka 13.8: To už by rozumné nebylo, neboť výsledek $n!$ má zhruba $\Theta(n \log n)$ bitů, což už je velmi (skutečně velmi!) dlouhé číslo vzhledem k $\log n$ bitům původního čísla n. Proto musíme poctivě dlouhou aritmetiku rozepsat na jednotlivá elementární násobení.

Cvičení 13.9: Pozor, každé číslo potřebuje k znaků. Pak mohou být nějaké další znaky pro oddělení čísel atd., ale ty se asymptoticky zanedbají. Velikost vstupu tak je $\Theta(k)$.

Cvičení 13.10: Je potřeba nejen zadat n vrcholů, ale také až n^2 hran, takže délka vstupu je $O(n^2)$. (Nepoužíváme Θ, neboť nakonec těch hran může být méně než n^2.)

Cvičení 13.11*: Vzpomeňte si, že rovinný graf má nejvýše $3n - 6$ hran, takže délka vstupu stačí vždy $O(n)$.

Cvičení 13.12: Cyklus `for` se vykoná právě n-krát, takže n-krát se i přičte hodnota do z a n-krát se inkrementuje i. Dále mezi aritmetické operace počítáme $n+1$ porovnání $i < n$ a jedno závěrečné dělení. Celkem tedy máme $3n + 2$ aritmetických operací. Asymptoticky to je $\Theta(n)$.

Cvičení 13.13: Vnější cyklus se jasně iteruje n^2-krát. Počet iterací vnitřního cyklu však závisí na proměnné i vnějšího cyklu. Vnitřních iterací proto proběhne součtem $\sum_{i=1}^{n^2} i = 1+2+3+\ldots+n^2-1+n^2$. Někteří z vás možná ví, že součet této řady je $\frac{1}{2}n^2(n^2+1)$, ale my si výsledek umíme asymptoticky odvodit sami

$$1 + 2 + 3 + \ldots + n^2 \leq n^2 + n^2 + \ldots + n^2 = n^2 \cdot n^2 = n^4\,,$$

ale na druhou stranu

$$1+2+\ldots+\frac{1}{2}n^2+\ldots+n^2 \geq \frac{1}{2}n^2+\frac{1}{2}n^2+1+\ldots+n^2 \geq \frac{1}{2}n^2 \cdot \frac{1}{2}n^2 = \Theta(n^4)\,.$$

Počet průchodů vnitřním cyklem tedy je $\Theta(n^4)$.

Cvičení 13.14: a) i b).

Cvičení 13.15: Jen a).

Cvičení 13.16: Ani jeden.

Cvičení 13.17: $C \prec A \prec B$

Cvičení 13.18: $B \prec A \prec C$

Cvičení 13.19: $A \prec C \prec B$

Cvičení 13.20: $C \prec A \prec B$

Cvičení 14.1: a) $O(n+m)$, b) $O(n^2)$

Otázka 14.2: $\Theta(n^2)$

Cvičení 14.3: Například v čase $O(n \log n)$ – setřídíme čísla a vezmeme to prostřední, nebo průměr prostředních dvou. Existují však i algoritmy pracující v čase $O(n)$, našli byste je?

Cvičení 14.4: Třeba $O(n^{k+1})$ – probereme všechny k-tice z vrcholů a u každé se podíváme, zda je nezávislá. Předpokládá se, že výrazně lepší obecný algoritmus neexistuje.

Cvičení 14.5*: Snadno v čase $O(n \log n)$ – všechny body seřadíme podle souřadnice x a pak po řadě zleva doprava vybíráme ty, co jsou „nejníže“ a „nejvýše“. Chytřejší algoritmus by to však zvládl i v čase $O(n)$.

Cvičení 14.6*: Ne, protože potom bychom dokázali třídit v čase $O(n)$: Čísla jedno po druhém do struktury přidáme a pak je od nejmenšího zase odebíráme. To nelze podle Věty 14.1.

Otázka 14.7: Protože pro přesné vzorce bychom se museli zabývat i počátečními hodnotami $T(0), T(1), \ldots$ a bylo by to příliš složité. Asymptotické odhady budou pro klasifikaci rychlosti algoritmů stačit.

Otázka 14.8: Jednak $\log_b a$ je definovaný jen pro $b > 1$, za druhé pro b=1 by vztah zněl $T(n) = aT(n) + f(n)$, což je nesmyslné.

Cvičení 14.9: Podle Lemmatu 14.2 je $T(n) = O(n)$.

Cvičení 14.10: Podle Lemmatu 14.3 je $T(n) = O(n \log n)$.

Cvičení 14.11: Podle Lemmatu 14.4 je $T(n) = O(n^2)$.

Cvičení 14.12: Podle Lemmatu 14.4 je $T(n) = O(n^2 \log n)$.

Cvičení 14.13: S každým rekurzivním voláním se jedno z čísel a, b zmenší aspoň o 1. Nemůže tedy nastat více než $2 \cdot 2^k = 2^{k+1}$ rekurzivních volání. Každé volání trvá konstantní čas. Na druhou stranu si lze snadno představit zadání, při kterém bude 2^k iterací skutečně nutných: $a = 1$, $b = 2^k$. Zkuste si to projít sami! Celkem tedy je složitost našeho algoritmu $\Theta(2^k)$.

Cvičení 14.14: Pokud $a \geq b$, tak číslo (zbytek) $c = a \bmod b$ je vždy méně než polovinou hodnoty b. Proto se v každé iteraci našeho algoritmu (možná mimo první) jedno z čísel a, b zmenší na méně než polovinu, neboli z jeho binárního zápisu ubude aspoň jedna číslice. Pokud na začátku měli a, b jen ℓ bitů, algoritmus musí skončit po méně než 2ℓ iteracích. Každá tato iterace trvá konstantní čas, takže celkem máme horní odhad $O(\ell)$, což je mnohem lepší než v Příkladě 14.13.

Pro dolní odhad bychom měli najít dvojici čísel a, b, pro které trvá běh algoritmu co nejdéle. (Sice můžeme zjednodušeně říci, že potřebujeme aspoň přečíst ℓ bitů vstupu, ale to není úplně dostačující k rigoróznímu argumentu, neboť v zadání zanedbáváme délku zápisu vzhledem k aritmetickým operacím.) Není to nyní zase tak jednoduché, ale po pár pokusech asi přijdete na to, že nejlepší je volit dva po sobě jdoucí členy Fibonaciho posloupnosti ($a_0 = a_1 = 1$, $a_{n+1} = a_n + a_{n-1}$, znáte úlohu o množení králíků na ostrově?). Například $a = 13$ a $b = 8$ dá 6 iterací cyklu. Celkem v tomto případě počet iterací vyjde $\Theta(\ell)$, takže to je i nejhorší složitost našeho algoritmu.

Cvičení 14.15: Vidíme, že v algoritmu dochází jen k jednomu rekurzivnímu volání, ale bohužel, pokud zvolíme za špatného pivota $A[i]$ to největší z čísel, bude rekurzivní volání zpracovávat pole velikosti $n-1$. Takový případ žádný ze vzorců z kapitoly 14.4 neřeší.

Musíme si tedy pomoci prostou úvahou – v každém výpočetním kroku se velikost zpracovávaného pole zmenší aspoň o 1 (o pivota), takže bude jen

$O(n)$ vnoření rekurze a v každém provedeme $O(n)$ kroků, celkem $O(n^2)$. Na druhou stranu ve zmíněném nejhorším případě budeme v druhé úrovni rekurze zpracovávat pole velikosti $n-1$, ve třetí úrovni pole o velikosti $n-2$, atd.... Celkový čas na zpracování pak skutečně bude

$$\Theta(n) + \Theta(n-1) + \ldots + \Theta(1) = \Theta(n^2)\,.$$

Na závěr dodáváme, že sice tento algoritmus má pomalý běh v nejhorším případě, ale v průměrném případě bude velmi rychlý, neboť obvykle pivot rozdělí pole téměř „napůl“. (Je to stejný případ jako s algoritmem quicksort.)

Cvičení 14.16: $1^2 + 2^2 + 3^2 + \ldots + n^2 = \Theta(n^3)$

Cvičení 14.17: $\Theta(\log n)$

Cvičení 14.18*: $\Theta(\sqrt{n})$

Cvičení 14.19*: Rozdělíme pole A na pětice čísel, z každé z nich vybereme prostřední přímým porovnáváním, a pak z vybraných $n/5$ čísel vybereme to prostřední rekurzivní aplikací algoritmu Vyber(). Jej pak vezmeme za pivota, což zaručí, že každé z polí B, C bude obsahovat aspoň 30% z čísel. Vzorec pro dvě rekurzivní volání pak bude znít $T(n) \leq T(0.7n) + T(0.2n) + O(n)$, což je $O(n)$ podle Lemmatu 14.2.

Cvičení 14.20: Šikovně to lze lineárně $O(n)$.

Cvičení 14.21: Tento vztah sice na první pohled nepatří do žádného z uvedených lemat, ale lze jej velice snadno upravit:

$$T(n) \leq T(n/2) + T(n/3) + T(n/4) + 5n^2 \leq 3T(n/2) + 5n^2$$

Poslední vztah již podle Lemmatu 14.4má řešení $T(n) = O(n^2)$ neboť $\log_2 3 < 2$. Na druhou stranu hned ze zadaného vztahu vidíme, že $T(n) \geq 5n^2$, takže výsledné řešení skutečně je $T(n) = \Theta(n^2)$.

Cvičení 14.22: $O(n)$

Cvičení 14.23: $O(n^2)$

Cvičení 14.24: $O(n)$

Cvičení 14.25: $O(n \log n)$

Cvičení 14.26: $\Theta(n \log n)$

Cvičení 14.27: Na rozdíl od předchozích příkladů se zde zaměříme na aspekt rekurze v algoritmu. Nechť $T(n)$ je časová složitost našeho algoritmu. V první řadě si zjistíme, kolikrát a pro jak dlouhá čísla se volá rekurze. Dvakrát se násobí čísla délky $k = \frac{n}{2}$ pro výpočty $z_1 = (a_1 \cdot b_1)$ a $z_3 = (a_2 \cdot b_2)$. Pak se jednou násobí čísla délky (až) $k + 1$ pro výpočet $(a_1 + a_2) \cdot (b_1 + b_2)$. Takže máme začátek rekurentního vzorce $T(n) \leq 2T(k) + T(k + 1) + \ldots$.

V druhé řadě se podíváme, kolik času zaberou zbylé výpočty v algoritmu. Jedná se o rozdělení čísel a, b na poloviny jejich dekadických zápisů a o několik sčítání a odečítání n-místných čísel. Zde si již musíme přesně ujasnit, jaké použijeme datové struktury. Jako nejvhodnější se jeví použít pole pro uložení jednotlivých číslic dekadického zápisu čísel a, b. Potom jak rozdělení, tak i sčítání a odečítání lze snadno implementovat v čase $O(n)$. Celkem tak dostaneme odhad

$$T(n) \leq 2T(k) + T(k + 1) + O(n) = 2T\left(\frac{n}{2}\right) + T\left(\frac{n}{2} + 1\right) + O(n)$$

a po zanedbání "+1" v $T(k + 1)$ vyjde

$$T(n) \leq 3T\left(\frac{n}{2}\right) + O(n)\,.$$

Podle Lemmatu 14.4 je řešením tohoto rekurentního vztahu asymptoticky

$$T(n) = O(n^{\log_2 3}) \doteq O(n^{1.585})\,.$$

Cvičení 14.28*: Zde využijeme Lemma 14.4, kde je $b = 2$ (poloviční velikost rozdělených matic), $a = 7$ (7 násobení mezi nimi) a $f(n) = n^2$ (čas potřebný na manipulaci a sčítání matic). Výsledek pak podle vzorce je $T(n) = n^{\log_2 7} \doteq \Theta(n^{2.8})$.

Cvičení 14.29*: Rekurentní vzorec přepíšeme pro náhradní funkci $T'(n) = T(n + 2)$:

$$T'(n) \leq 2T(k) + T(k + 1) + O(n) \leq 3T(k + 1) + O(n) =$$

$$= 3T(\frac{n+2}{2}+1) + O(n) = 3T(\frac{n}{2}+2) + O(n) = 3T'(\frac{n}{2}) + O(n)$$

Cvičení 15.1: Použijeme vzorec $x \cdot y = e^{\ln x + \ln y}$, takže vstup (x, y) převedeme na vstup $(\ln x, \ln y)$, sečteme logaritmy a pro zpětný převod výsledku umocníme e^z. Takto se násobilo před objevením kalkulaček, pomocí logaritmických pravítek či tabulek.

Otázka 15.2: Protože obvykle každý algoritmus pro správnou odpověď musí nejprve celý vstup délky n přečíst.

Cvičení 15.3: Primitivní algoritmy třídění pracují sice va čase $O(n^2)$, ale ty chytřejší, jako třeba mergesort nebo heapsort už běží v čase $O(n \log n)$, takže to je hledaná časová složitost problému třídění.

Cvičení 15.4: $\Theta(n^2)$, bez ohledu na počet hran, neboť musíme projít všechna políčka matice.

Cvičení 15.5: Teď již jen $\Theta(n)$, neboť stačí projít n vrcholů a porovnat jejich stupně, tj. délky seznamů sousedů.

Otázka 16.1: Kdyby bylo více možných výsledků než jen Ano/Ne, mohly by být i falešné nápovědy pro různá řešení a jak bychom mezi nimi v definici NP vybrali? Je to prostě technický problém daný našim pohledem na třídu NP.

Otázka 16.2: Hraje, a velkou. Podívejte se například na problém, zda dva grafy jsou isomorfní, kde nápovědou odpovědi Ano je vyznačení příslušného isomorfismu, kdežto pro odpověď Ne žádná obecně efektivní nápověda známa není.

Otázka 16.3: Popíšeme tak „duální“ třídu, která se často značí coNP a patří do ní právě negace problémů z NP.

Cvičení 16.4: Protože jednoduše napovíme (uhodneme) onu podmnožinu k vybraných vrcholů a efektivně ji zkontrolujeme.

Cvičení 16.5: Protože napovíme, kterých $< k$ vrcholů vypustit, aby graf zbyl nesouvislý. Nesouvislost zbytku už pak snadno ověříme.

Cvičení 16.6*: To je obtížnější, neboť nelze kontrolovat všechny k-tice vrcholů k vypuštění (exponenciální čas pro velká k). Využijeme však Mengerovu větu a pro každou dvojici vrcholů v grafu napovíme (uhodneme) k disjunktních cest mezi nimi. To je hodně velká nápověda, ale stále polynomiální.

Cvičení 16.7*: Nelze, neboť takovou nápovědou sice ukážeme dobré obarvení grafu G, ale nijak neprokážeme, že H přece jenom nelze obarvit lépe (méně barvami, než v nápovědě).

Otázka 16.8:

Cvičení 16.9: K danému grafu G přidáme jeden nový vrchol w spojený se všemi vrcholy. Pak G lze obarvit 3 barvami právě když $G \oplus w$ lze obarvit 4 barvami (čtvrtou barvu výhradně na w).

Cvičení 16.10: Podle Lemmatu 16.9 a převodu z předchozí úlohy víme, že 4-obarvení je NP-těžké. Zároveň existence 4-obarvení snadno patří do třídy NP.

Cvičení 16.12: a, b, c, f

Cvičení 16.13: Definujeme pro graf G nový graf H, jehož vrcholy budou odpovídat hranám grafu G, a hranami H budou spojeny dvojice hran z G sdílející vrchol. Potom prostě stačí v grafu H hledat nezávislou množinu velikosti p, což dá párování v G.

Cvičení 16.14*: ...?

Cvičení 16.15: Ano, patří, tyto kružnice napovíme a snadno ověříme.

Cvičení 16.16: Ani nemůže patřit, neboť se nejedná o rozhodovací problém!

Cvičení 16.17: Obojí patří: Pro ověření toho, že graf je rovinný, stačí zkontrolovat napovědněné rovinné nakreslení. (Vzpomeňte si, že rovinný graf lze nakreslit tak, aby hrany byly úsečky.) Naopak pro ověření nerovinnosti stačí napovědět, kde je v G podrozdělení grafu $K_{3,3}$ nebo K_5.

Cvičení 16.18*: Pro $k = 0, 1, 2$ lze barevnost efektivně určit, takže tam otázka patří přímo do třídy P. Také pro $k = 3$ patří problém barevnosti k do NP, neboť napovědněné obarvení 3 barvami jsme schopni snadno ověřit, a zároveň dokážeme určit, že barevnost není 2 (kružnice liché délky). Ale pro $k > 3$ již problém (dle současných znalostí teoretické informatiky) do třídy NP nepatří, protože neumíme nijak efektivně prokázat, že graf G nelze obarvit méně než k barvami.

Cvičení 16.19: a, d

Literatura

[Chy84] Michal Chytil. *Automaty a gramatiky.* SNTL Praha, 1984.

[Gru97] Jozef Gruska. *Foundations of Computing.* Intern. Thomson Computer Press, 1997.

[Hli05] Petr Hlineny. Diskrétní matematika. VŠB-TUO, Ostrava, 2005. http://www.cs.vsb.cz/hlineny/vyuka/DIM-slides/.

[HU78] J. Hopcroft and J. Ullman. *Formálne jazyky a automaty.* Alfa Bratislava, 1978.

[Kuč83] Luděk Kučera. *Kombinatorické algoritmy.* SNTL Praha, 1983.

[MvM87] Molnár, Češka, and Melichar. *Gramatiky a jazyky.* Alfa-SNTL, 1987.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publish. Comp., 1997.