Úvod do Teoretické Informatiky (456-511 UTI)

Doc. RNDr. Petr Hliněný, Ph.D.

petr.hlineny@vsb.cz

25. ledna 2006

Verze 1.02.

Copyright © 2004–2006 Petr Hliněný.

(S využitím části materiálů © Petr Jančar.)

Obsah

0.1 Předmluva . . . iv

I Jazyky a Automaty 1

1 Úvodní pojmy, jazyky 1
1.1 Množiny a operace s nimi . . . 1
1.2 Orientované grafy . . . 2
1.3 Formální abeceda a jazyk . . . 3
1.4 Cvičení: Práce s formálními jazyky . . . 5

2 Konečné automaty 8
2.1 Definice konečného automatu . . . 9
2.2 Jazyk rozpoznávaný automatem . . . 13
2.3 Nedeterministické automaty . . . 15
2.4 Cvičení: Konstrukce a převody automatů . . . 18

3 Vyhledávání a regulární výrazy 21
3.1 Automaty a vyhledávání v textu . . . 22
3.2 Vyhledávání z pohledu programátora . . . 23
3.3 Regulární operace a výrazy . . . 25
3.4 Ekvivalence regulárních výrazů a jazyků . . . 26
3.5 Cvičení: Vyhledávání a regulární výrazy . . . 29

4 Minimalizace a omezení automatů 33
4.1 Minimalizace konečného automatu . . . 33
4.2 Neregulární jazyky . . . 37
4.3 Věta Myhilla–Neroda . . . 38
4.4 Cvičení: Minimalizace a neregulárnost . . . 39

5 Bezkontextové jazyky a ZA 41
5.1 Odvození aritmetických výrazů . . . 41
5.2 Bezkontextové gramatiky a jazyky . . . 43
5.3 Zásobníkové automaty . . . 45
5.4 Cvičení: Konstrukce bezkontextových gramatik . . . 47

II Složitost Algoritmů 50

6 Matematické základy složitosti 50
6.1 Asymptotické značení funkcí . . . 50
6.2 Rekurentní vztahy . . . 52
6.3 Kódování slov a problémů . . . 53
6.4 Cvičení: Odhady funkcí a rekurentních vztahů . . . 55

7 Co je to algoritmus 56
7.1 Turingův stroj . . . 56
7.2 RAM model počítače . . . 59
7.3 Univerzální algoritmus a neřešitelnost . . . 60
7.4 Cvičení: Výpočty na TS a RAM modelech . . . 62

8 Výpočetní složitost algoritmů a problémů **65**
8.1 Délka výpočtu 66
8.2 Časová složitost problému 67
8.3 Některé "rychlé" algoritmy 68
8.4 Třída $\mathcal{P}$ a polynomiální redukce 70
8.5 Cvičení: Určení časové složitosti algoritmu 71

9 Základy $\mathcal{NP}$-úplnosti **75**
9.1 Třída $\mathcal{NP}$ 75
9.2 Nedeterministický Turingův stroj 77
9.3 $\mathcal{NP}$-úplný problém 78
9.4 Cvičení: Polynomiální převody a příslušnost do $\mathcal{NP}$ 79

10 $\mathcal{NP}$-úplné problémy **83**
10.1 Splnitelnost formulí 83
10.2 Grafové problémy 85
10.3 Aritmetické problémy 88
10.4 Cvičení: Další $\mathcal{NP}$-úplné problémy a převody 89

III **92**

Klíč k řešení úloh **92**
Literatura 99

0.1 Předmluva

Vážení čtenáři,

dostává se vám do rukou výukový text pro Úvod do Teoretické Informatiky, který je primárně určený pro studenty informatických oborů (od bakalářského stupně) na technických vysokých školách.

Teoretická informatika nám dává formální a teoretické základy nezbytné pro přesné formulace a správná řešení problémů spojených s informatikou. Teoretická stránka informatiky se počala rozvíjet již s prvními (mechanickými) výpočetními stroji v první polovině 20. století, jak můžeme zmínit například na pracech Turinga či jiných. Skutečného rozmachu dosáhla pak v druhé polovině století s prudkým nástupem počítačů do všech oblastí života, což vyvolalo i naléhavou potřebu teoreticky definovat a řešit mnohé zcela nové vědecké otázky spojené s algoritmizací praktických problémů.

Náš text vás seznámí s úvodními partiemi dvou asi nejvýznamnějších oblastí teoretické informatiky – s teorií jazyků a konečných automatů a s teorií výpočetní složitosti algoritmů. Jde o seznámení jen velmi zběžné, neboť podání pouhých úplných základů každé jedné z těchto oblastí by vydalo alespoň na jeden semestr výuky. Přesto jsme se pokusili do jednoho celku z každé části shrnout několik základních pojmů a poznatků, které by studentům umožnily nahlédnout pod pokličku teoretické informatiky a poznat její užitečnost v různých informatických aplikacích.

Náš výukový text je koncipován jako soběstačný celek, který od čtenáře nevyžaduje o mnoho více než běžné středoškolské matematické znalosti, zkušenosti s počítačem a chuť do studia. Svým rozsahem text odpovídá jednomu semestru běžné výuky včetně cvičení. Vhodným doplňkem při jeho studiu je obdobný výukový text Diskrétní Matematika [4] stejného autora.

Při přípravě látky vycházíme zčásti ze stávajících výukových materiálů pro teoretickou informatiku na FEI VŠB–TUO od Doc. RNDr. Petra Jančara, Ph.D., z druhé části z autorových vlastních studijních zkušeností na MFF UK. Proti zaběhlým zvyklostem výuky teoretické informatiky uvádíme několik méně běžných motivů, z nichž za zmínku stojí například "nápovědný" pohled na nedeterminismus a na třídu $\mathcal{NP}$ ve výpočetní složitosti.

Ve stručnosti se zde zmíníme o struktuře našeho textu. Přednesený materiál je dělený do jednotlivých lekcí (kapitol), které zhruba odpovídají obsahu týdenních přednášek v semestru. Lekce jsou dále děleny tematicky na oddíly. Výklad je strukturován obvyklým matematickým stylem na definice, tvrzení, algoritmy, případné důkazy, poznámky a neformální komentáře. Navíc je proložen řadou vzorově řešených příkladů navazujících na vykládanou látku a doplněn dalšími otázkami a úlohami. Každá lekce je zakončena zvláštním oddílem určeným k dodatečnému procvičení látky. Správné odpovědi k otázkám a úlohám jsou shrnuty na konci textu.

Přejeme vám mnoho úspěchů při studiu a budeme potěšeni, pokud se vám náš výukový text bude líbit. Jelikož nikdo nejsme neomylní, i v této publikaci zajisté jsou nejasnosti či chyby, a proto se za ně předem omlouváme. Pokud chyby objevíte, dejte nám prosím vědět e-mailem `mailto:petr.hlineny@vsb.cz`.

Autor

Pár slov o vzniku

Tento učební text vznikal v průběhu let 2004 až 2005 podle autorových přednášek a cvičení předmětu Úvod do teoretické informatiky na FEI VŠB – TUO. Jeho základní verzi utvořil soubor autorových slidů pro přednášky předmětu. Většina přidaných řešených příkladů a úloh k procvičení pak vychází z náplně skutečných cvičení a písemných zápočtů a zkoušek v předmětu.

Významné poděkování za vznik tohoto textu patří doc. Petru Jančarovi, který nezištně poskytl části svých výukových materiálů týkajících se teoretické informatiky i pro náš text. Za přípravu některých příkladů a zvláště za kontrolu celého textu patří díky také cvičícím – bývalým i současným doktorandským studentům Martinu Kotovi, Přemyslu Teperovi, Ondřeji Kohutovi, Dušanu Fedorčákovi.

Část I
Jazyky a Automaty

1 Úvodní pojmy, jazyky

Úvod

Předmět Teoretická informatika poskytuje formální základy a nástroje pro praktické informatické aplikace (jako programování či softwarové inženýrství). Jeho asi nejdůležitějším úkolem je matematicky popsat různé typy algoritmických výpočtů, jejich proveditelnost a složitost. Přitom pro matematický popis zadání (vstupů a výstupů) výpočtu je potřebné nejprve zavést pojmy abecedy symbolů a formálních slov nad touto abecedou.

Použití symbolické abecedy pro vstupy a výstupy výpočtů závisí na dohodnuté formě zápisu, v praxi u počítačů se nejčastěji používají binární zápisy s abecedou $\{0,1\}$, občas hexadecimální s abecedou $\{0,1,\ldots,9,a,\ldots,f\}$, nebo nejčastěji "lidsky čitelný" zápis s abecedou ASCII či nověji UTF-8. Matematicky můžeme za abecedu považovat libovolnou (dohodnutou) konečnou množinu symbolů, přitom převody zápisů mezi různými abecedami jsou velmi jednoduché. (Volíme si proto obvykle abecedu, která se nejlépe hodí pro vyřešení daného problému.)

Důležitým teoretickým pojmem je "slovo", které znamená libovolný konečný řetězec symbolů nad abecedou, přitom mezera zde nemá žádný zvláštní význam. Určená množina slov se nazývá "jazykem". Tyto pojmy v úvodní lekci přesně definujeme a zároveň zavedeme důležité operace s formálními jazyky.

Cíle

Za hlavní cíl úvodní části považujeme jednak správné (víceméně filozofické) pochopení pojmů konečnosti a nekonečnosti množin; za druhé nastudování pojmů abecedy, formálního jazyka a operací s nimi.

1.1 Množiny a operace s nimi

Úvodem si připomeneme známé matematické pojmy týkající se převážně množin a jejich konečnosti či nekonečnosti. Zároveň si takto zavedeme (pro ujednocení) některé základní konvence a značení používané dále v textu.

Symbolika množin

- *Množiny* obvykle značíme velkými písmeny $A, B, X, Y, \ldots$
  a jejich *prvky* malými písmeny $a, b, x, y, \ldots$,
  symbol $\emptyset$ značí *prázdnou množinu*.
- $M = \{a, b, c, x\}$ je množina se čtyřmi prvky a, b, c, x a píšeme $a \in M$, $b \in M$, ale $d \notin M$. Počet prvků značíme $|M| = 4$.
  $N = \{a, b, x\}$ je *podmnožinou* M, což píšeme jako $N \subseteq M$.
- *Sjednocení* množin značíme $A \cup B$, jejich *průnik* $A \cap B$.
- *Rozdíl* množin je definován $A \setminus B = \{a \in A : a \notin B\}$
  a *symetrický rozdíl* je $A \Delta B = (A \setminus B) \cup (B \setminus A)$ (jako "XOR").
- Rozšířené značení sjednocení $\bigcup_{i=1}^{n} X_i$ a průniku $\bigcap_{i=1}^{n} X_i$ používáme jako u sumy.

`Komentář:` Pro názornou ukázku si zvolme například množiny

$$A = \{a, b, c, d\}, \quad B = \{b, c, e, f\}.$$

Pak sjednocením těchto dvou množin je $A \cup B = \{a, b, c, d, e, f\}$ a průnikem je $A \cap B = \{b, c\}$. Výsledkem rozdílu je $A \setminus B = \{a, d\}$ a symetrického rozdílu $A \Delta B = \{a, d, e, f\}$.

Konečné a nekonečné množiny

- Množina je *konečná* pokud počet jejich prvků je přirozené číslo.
- Množina je *spočetná* pokud všechny její prvky lze seřadit do jedné posloupnosti, neboli očíslovat přirozenými čísly. (Může být konečná i nekonečná.)
- Množina je *nespočetná* pokud není spočetná. (Například množina všech reálných čísel je nespočetná.)

Potenciální a aktuální nekonečno

Pro lepší pochopení pojmu nekonečnosti množiny si připomeňme jeden ze základních antických paradoxů – závod Achilla se želvou: Achilles běží 10× rychleji než želva, ale želva má 100m náskok. Než Achilles těchto 100m proběhne, želva je dalších 10m před ním, po uběhnutí dalších 10m má želva stále 1m 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 *roste nade všechny meze*.

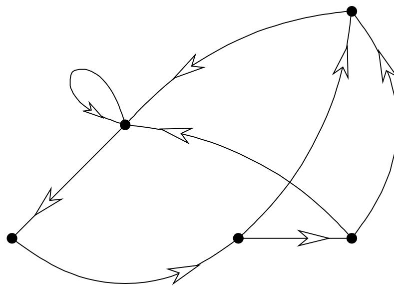
– *"Aktuální nekonečno"* — uvažujeme skutečně nekonečné množiny v celé jejich šíři.

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 $111.\overline{1}$ m.

1.2 Orientované grafy

S pojmem grafu jsme se dobře seznámili v diskrétní matematice – graf se skládá z množiny vrcholů a množiny hran spojujících dvojice vrcholů. V některých případech (jako třeba u automatů) potřebujeme u každé hrany grafu vyjádřit její směr. To vede na definici *orientovaného grafu*, ve kterém hrany jsou formálně uspořádané dvojice vrcholů. Orientované grafy odpovídají relacím, které nemusí být symetrické.

`Značení`: Hrana (u, v) v orientovaném grafu D *začíná* ve vrcholu u a *končí* ve (míří do) vrcholu v. Také připouštíme tzv. smyčku (u, u) mířící z u zpět do u. Opačná hrana (v, u) je různá od (u, v)! Příklad kreslení orientovaného grafu následuje.



1.3 Formální abeceda a jazyk

Pro zápis zadání a výsledků výpočtů je nezbytné se domluvit na použití jednotných znaků–symbolů abecedy. Sdělované informace pak mají formu řetězců nad těmito znaky, které nazýváme prostě "slovy".

`Definice`: *Abecedou* myslíme libovolnou konečnou množinu Σ. Prvky Σ nazýváme *symboly* (písmena) této abecedy. (Třeba $\Sigma = \{a,b\}$.)
Slovem nad abecedou Σ rozumíme libovolnou konečnou posloupnost prvků Σ, například 'abbbaaab' (jako počítačový *řetězec*).
Prázdné slovo ' ' je také slovem a značí se ε.

`Značení`: Výrazem Σ^* značíme *množinu všech slov* na abecedou Σ.

`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.

`Značení`: Délku slova w, tj. počet písmen ve w, značíme $|w|$.

`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`: Říkáme, že slovo z je *zřetězením* slov x a y, pokud z vzniká zapsáním x a y za sebou bez mezer. Značíme $z = x \cdot y$, nebo zkráceně $z = xy$.

Naopak někdy potřebujeme slovo zpět rozdělit na jeho počáteční a koncovou část. Úsek znaků, kterým nějaké slovo začíná, se nazývá předponou, neboli odborně matematicky prefixem. Obdobně se úsek znaků, kterým slovo končí, nazývá příponou, odborně sufixem.

`Definice`: Slovo t je *prefixem* slova z, pokud lze psát $z = tu$ pro nějaké slovo u, přitom u je také nazýváno *sufixem*.

> `Komentář:` Vezměme si například slovo 'abcadbcdc'. Pak slovo 'abc' je jeho prefixem, kdežto 'bc' prefixem není. Naopak 'bcdc' je sufixem. Prázdné slovo ε je prefixem i sufixem každého slova.

Nakonec přicházíme k důležité formální definici jazyka, který je tvořený jakoukoliv vybranou množinou slov.

Definice 1.1. ***Jazykem*** nad abecedou Σ
rozumíme libovolnou podmnožinu $L \subseteq \Sigma^*$ množiny Σ^* všech slov.

`Komentář:` Na rozdíl od přirozeného jazyka (češtiny) budou naše jazyky obvykle obsahovat nekonečně mnoho slov, řečeno slovy *nekonečný jazyk*. V analogii s přirozenou mluvou si formální jazyk můžeme představit třeba jako množinu všech knih, které kdy byly na světe napsány, nebo třeba jako množinu všech možných knih, které lze gramaticky správně v češtině napsat.

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

$$L_1 = \{\varepsilon, 01, 0011, 1111, 000111\},$$

$$L_2 = \big\{\text{všechny posloupnosti z } \{0,1\}^* \text{ mající stejně '0' jako '1'}\big\},$$

$$L_3 = \big\{\text{všechny posl. z } \{0,1\}^* \text{ zapisující binárně číslo dělitelné třemi}\big\}.$$

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.

Některé operace s jazyky

Při práci s formálními jazyky používáme nejčastěji následující matematické operace:

- Běžné množinové operace sjednocení $K \cup L$, průniku $K \cap L$, nebo rozdílu $K \setminus L$.
- *Zřetězení* dvou jazyků $K \cdot L = \{uv : u \in K, v \in L\}$, tj. jazyk všech slov, které začínají slovem z K a pokračují 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 vzniklých libovolným opakováním slov z jazyka L za sebou.

`Komentář:` Dejte si pozor, že operace zřetězení dvou jazyků není komutativní, tj. obecně nelze zaměnit pořadí $K \cdot L \neq L \cdot K$.

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.

Příklad 1.2. *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 $`10' \not\in L_0 \cdot L_1$. 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, $`10' \not\in L_0 \cdot L_1$, 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. □

Úlohy k řešení

(1.3.1) *Která slova jsou zároveň prefixem i sufixem slova '101110110'? (Najdete všechna tři taková?)*

(1.3.2) *Vypište slova ve zřetězení jazyků* $\{110, 0111\} \cdot \{01, 000\}$.

(1.3.3) *Najděte dva různé jazyky, které komutují v operaci zřetězení, tj.* $L_1 \cdot L_2 = L_2 \cdot L_1$.

****(1.3.4)** *Co vzniká iterací jazyka* $\{00, 01, 1\}$*? Patří tam všechna slova nad* $\{0, 1\}$*?*

Rozšiřující studium

Pro teoretickou informatiku existuje množství učebních textů a knih. (Obvykle bývají oddělené zvlášť pro automaty a jazyky, zvlášť pro výpočetní složitost, tak jako je dělen i náš učební text.) Jak již bylo uvedeno, náš text vychází částečně z učebního textu P. Jančara [8], ale je zjednodušen a lépe zpřístupněn studentům bakalářského studia bez předchozích znalostí informatické teorie.

Mimo zmíněného textu [8] pro rozšiřující studium problematiky automatů doporučujeme například [7] a [6], či nepřeberné množství anglických textů. Dobře zpracovaný úvod do problematiky složitosti a $\mathcal{NP}$*-úplnosti především kombinatorických algoritmů čtenář najde v knize [9].*

1.4 Cvičení: Práce s formálními jazyky

Příklad 1.3. *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* $L_1 \cap L_2$*?*

V prvé řadě si uvědomme, že oba jazyky jsou nekonečné. Přesto je průnik konečný – slovo patří do obou jazyků jen pokud má nejvýše pět výskytů každého ze znaků 0,1, tj. celkem nejvýše délku 10. (Podle definice L_1 má slovo $w \in L_1 \cap L_2$ nejvýše pět znaků '1' a podle L_2 má stejně znaků '0'.) Navíc slovo v průniku $L_1 \cap L_2$ má sudou délku 2ℓ, kde $\ell \leq 5$ je počet výskytů znaku 0 i 1.

Pro $\ell = 0$ máme jediné slovo ε, pro $\ell = 1$ jsou dvě slova '01' a '10' a obecně je v průniku $\binom{2\ell}{\ell}$ slov pro každé $\ell = 0, 1, 2, 3, 4, 5$. (Vybíráme pozice znaků 1 z 2ℓ písmen ve slově.) V součtu pak je

$$|L_1 \cap L_2| = 1 + 2 + \binom{4}{2} + \binom{6}{3} + \binom{8}{4} + \binom{10}{5} = 351\,.$$

□

Příklad 1.4. *Uvažujme jazyky nad abecedou* $\{a, b\}$*. Vypište všechna slova ve zřetězení jazyků* $L_1 = \{\varepsilon, abb, bba\}$ *a* $L_2 = \{a, b, abba\}$*.*

Postupujeme jednoduše podle definice – bereme jedno za druhým slova z L_1 a zřetězujeme za ně jedno po druhém slova z L_2. Celkem vyjde

$$L_1 \cdot L_2 = \{a, b, abba, abbb, abbabba, bbaa, bbab, bbaabba\}\,.$$

(Všimněte si dobře, že výsledek obsahuje jen $8 < 3 \cdot 3$ slov. Jak je to možné? To proto, že slovo '*abba*' se ve zřetězeních objeví dvakrát, jako $\varepsilon \cdot abba$ a jako $abb \cdot a$.) □

Příklad 1.5. *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* $L_0 \cdot L_0$*.*

Za prvé si všimněme, že prázdné slovo $\varepsilon \notin L_0 \cdot L_0$, neboť $\varepsilon \notin L_0$. Ze stejného důvodu ani jednopísmenná slova c a d nejsou v $L_0 \cdot L_0$ – pokud by byly složeny ze dvou slov, jedno z nich by bylo ε. Naopak každé slovo liché délky musí mít různý počet c a d, protože dva stejné počty by vynutily sudou délku slova, a tudíž náleží do L_0. Každé neprázdné slovo w sudé délky lze napsat jako zřetězení jeho prvního písmene w_1 a zbylého sufixu

liché délky '$w_2 \dots w_k$', a proto je $w \in L_0 \cdot L_0$. Co však se slovy w liché délky větší než 1? Pokud v nich najdeme sudý prefix nebo sufix, který neobsahuje stejně c jako d, máme opět $w \in L_0 \cdot L_0$.

Na základě výše uvedených úvah odhadneme, že slovo w nepatří do $L_0 \cdot L_0$ pouze tehdy, když $w = \varepsilon$ nebo w je liché délky ve "střídavém" tvaru '*cdcdc*...*dc*' nebo '*dcd*...*cd*'. (Je poměrně zřejmé, že taková slova do $L_0 \cdot L_0$ patřit nemohou.) Nyní zbývá naše tvrzení korektně dokázat: Pokud slovo w není ve výše uvedeném vyjímečném tvaru, má buď w sudou délku a potom $w \in L_0 \cdot L_0$, jak jsme již uvedli výše, nebo w má lichou délku a obsahuje dva výskyty c za sebou nebo dva výskyty d za sebou. Potom buď za prvním nebo za druhým z těchto opakovaných znaků w rozdělíme na prefix a sufix z L_0. □

Příklad 1.6. *Zjistěte, který z následujících dvou vztahů jsou platné pro všechny jazyky* L_1, L_2*:*

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

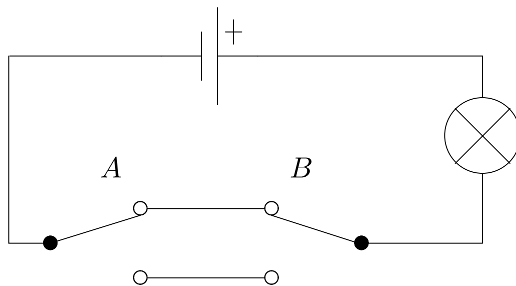
b) $(L_1 \cap L_2)^* = L_1^* \cap L_2^*$ *?*

Univerzální platnost (a) plyne přímo z definice zřetězení, jak si sami snadno ověříte.

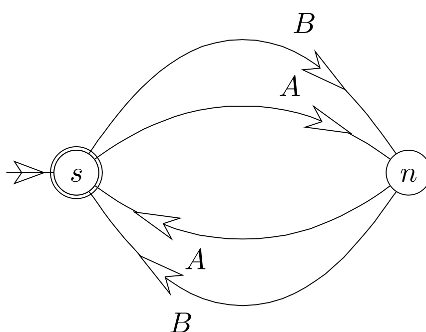
Naopak vztah (b) je obecně chybný, neboť snadno najdeme dvojici jazyků nemající žádné slovo ve svém průniku, ale zároveň mající společné iterace slov. Například pro disjunktní $L_1 = \{0\}$ a $L_2 = \{00\}$ obě iterace L_1^*, L_2^* obsahují slovo '00' nebo '0000', atd. □

V další části si neformálně přiblížíme praktické motivační příklady vedoucí k (pozdější) definici konečného automatu.

Příklad 1.7. *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.*

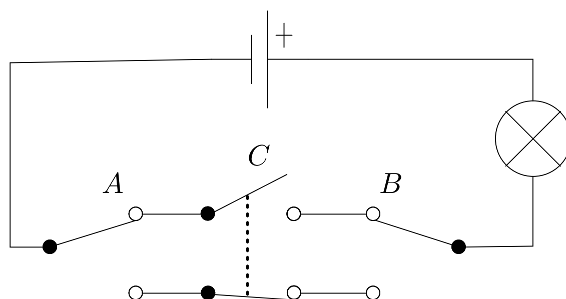


Posloupnosti stisků přepínačů A, B vedoucí k rozsvícení žárovky si můžeme popsat formálně jako množinu (tj. jazyk) R jistých slov nad abecedou $\{A, B\}$. (Každý znak A nebo B znamená přepnutí příslušného přepínače.) Za zadání je zřejmé, že $\varepsilon \in R$. Naopak třeba 'A'$\notin R$. Když se na znázorněný obvod blíže podíváme, zjistíme, že každý stisk kteréhokoliv tlačítka změní současný stav svícení. Schematicky tak pozorovatelné změny v obvodu můžeme znázornit následujícím diagramem, ve kterém levý stav s je počáteční a znamená, že žárovka svítí, kdežto v pravém stavu n žárovka nesvítí. (Tento diagram–model si v následující lekci matematicky popíšeme jako tzv. "konečný automat".)



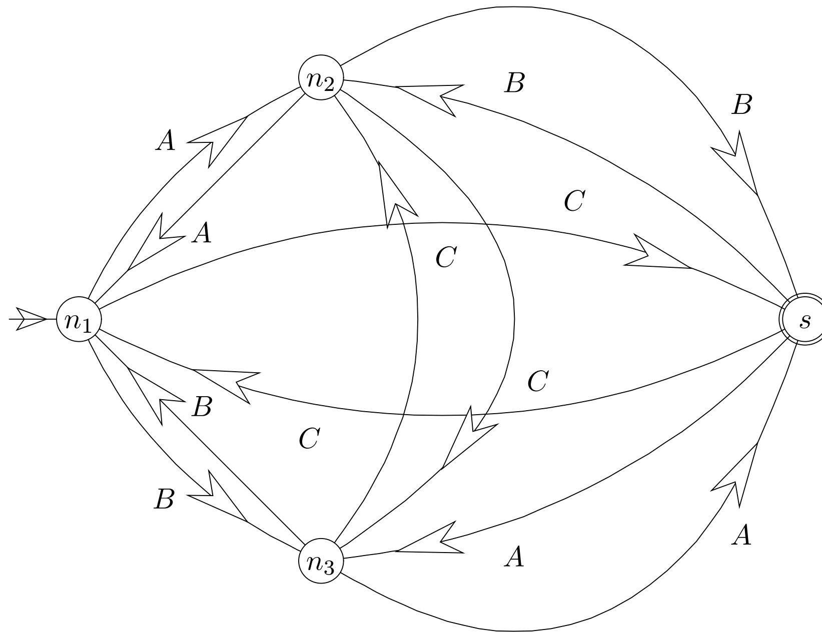
Všimněme si ještě jedné zajímavosti – každý přechod v našem diagramu je symetrický, tj. dvojím stiskem téhož tlačítka se dostaneme zase zpět. To je dáno fyzickou stavbou našeho obvodu, ale vůbec to nemusí být vlastností jiných systémů (představte si třeba obvod s cyklickými třístavovými přepínači). □

Příklad 1.8. *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?*



Označme nyní R' množinu (jazyk) všech těch slov nad abecedou $\{A, B, C\}$, která rozsvítí žárovku. Nyní $\varepsilon \notin R'$, ale třeba 'C'$\in R'$ a 'AB'$\in R'$. Zamysleme se však dobře nad tím, zda dokážeme chování obvodu plně popsat jen dvěma stavy "svítí–nesvítí". Například z počátečního stavu lze rozsvítit žárovku jediným stiskem C, ale ze stavu po přepnutí A již žárovku žádným stiskem C nelze zapnout. Analýzou této komplikace přijdeme na to, že asi bude třeba popsat více různých vnitřních stavů, které se navenek projevují jako "nesvítí".

Máme tedy počáteční stav obvodu n_1, jiný nesvítící stav n_2, do kterého se dostaneme přepnutím A, ještě jiný nesvítící stav n_3, do kterého se dostaneme přepnutím B (na rozdíl od n_2 z n_3 nelze rozsvítit žárovku stiskem B), a nakonec svítící stav s. (V obecnosti i svítících stavů může být více než jeden, ale to nenastává v našem obvodě.) Když pochopíme toto rozdělení vnitřních stavů, už nám nebude činit potíže doplnit jednotlivé přechody:



□

Úlohy k řešení

(1.4.1) *Uvažujme jazyky nad abecedou* $\{0,1\}$. *Vypište všechna slova ve zřetězení* $\{0,001,111\}\cdot\{\varepsilon,01,0101\}$.

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

(1.4.3) *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 (tj. palindromů). Která všechna slova jsou v průniku* $L_1 \cap L_2$*?*

Návod: Pozor, průnik obou jazyků je nekonečný.

(1.4.4) *Proč obecně neplatí* $(L_1 \cap L_2)\cdot L_3 = (L_1\cdot L_3)\cap(L_2\cdot L_3)$ *?*

(1.4.5) *Navrhněte obvod s třemi přepínači a žárovkou mající více než jeden vnitřní svítící i nesvítící stav.*

***(1.4.6)** *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* $L_a \cdot L_b$*?*

***(1.4.7)** *Jazyk* L_1 *obsahuje 6 slov a jazyk* L_2 *obsahuje 7 slov. Kolik nejméně slov musí obsahovat zřetězení* $L_1\cdot L_2$*?*

***(1.4.8)** *Proč obvod s třemi (dvoupolohovými) přepínači a žárovkou nemůže mít více než 8 vnitřních stavů?*

2 Konečné automaty

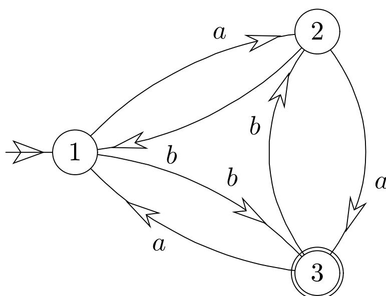
Úvod

První část našeho kurzu se věnuje především nejjednoduššímu klasickému výpočetnímu modelu – konečně-stavovému automatu, který popisuje procesy s předem omezeným počtem možných stavů.

Stručně řečeno, konečný automat je model systému, který může nabývat konečně mnoho vnitřních stavů. Tento stav se mění na základě jednoduchého vnějšího podnětu s tím, že pro daný stav a daný vstupní podnět je jednoznačně určeno, jaký (vnitřní) stav bude následující.

Přitom pro vnějšího pozorovatele je viditelné pouze, zda aktuální vnitřní stav automatu je tzv. "přijímací" nebo není.

Konkrétní konečný automat se často zadává diagramem – ohodnoceným orientovaným grafem, který také nazýváme stavovým diagramem či grafem automatu (kde šipky přechodů jsou označeny vstupními podněty). Pro příklady nemusíme chodit daleko, již jsme si je ukazovali v předechozím cvičení při popisu chování obvodů s přepínači a žárovkou. Jinou ukázkou je toto:



Cíle

Úkolem této lekce je přesně zavést pojem konečného automatu, jeho deterministickou a nedeterministickou variantu. Jsou zde uvedeny základní poznatky o automatech a definován pojem regulárního jazyka. Ukázán je převod obecného automatu na deterministický automat.

2.1 Definice konečného automatu

Konečný automat je model systému, který může nabývat konečně mnoho (obvykle ne "příliš mnoho") vnitřních stavů. Tento stav se mění na základě jednoduchého vnějšího podnětu (možných podnětů také není "příliš mnoho") s tím, že pro daný stav a daný vstupní podnět je jednoznačně určeno, jaký vnitřní stav automatu bude následující (tj. do jakého stavu systém přejde). Vnější pozorovatel přitom vnitřní stavy automatu nevidí, vnímá z automatu pouze jednoduchou dvoustavovou informaci – zda se automat právě nachází v tzv. *přijímajícím* stavu nebo v *nepřijímajícím*.

Přirozeným způsobem vizualizace konečného automatu je použití orientovaného (multi)grafu, kde vrcholy reprezentují jednotlivé vnitřní stavy a ohodnocené orientované hrany (šipky) ukazují přechody mezi stavy pro jednotlivé vstupní podněty. Stavy automatu obvykle číslujeme a začínáme v předepsaném *počátečním* stavu. Vstupní podněty obvykle reprezentujeme symboly zvolené abecedy a celé posloupnosti podnětů pak zapisujeme jako slova nad touto abecedou.

Matematicky konečný automat popíšeme následovně:

Definice 2.1. ***Konečný automat*** (zkráceně KA)
je uspořádaná pětice (tzn. je dán pěticí parametrů) $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, 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 neprázdná množina přijímajících (koncových) stavů.

`Komentář:` Význam množin Q a Σ v definici je jasný. 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.

Počáteční stav q_0 musí být určen jednoznačně, ale automat může mít více přijímajících stavů, třeba i všechny. Přijímajícím stavům se také často říká koncové, ale vůbec to neznamená, že by v těchto stavech měl výpočet vždy končit. (Na rozdíl od pozdější definice Turingova stroje.)

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

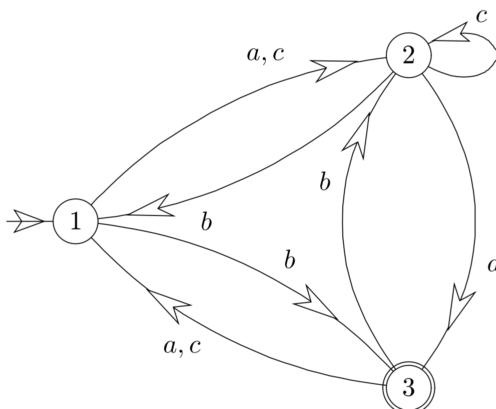
– vrcholy jsou stavy automatu, tj. množina Q,

– počáteční stav (q_0) je vyznačený příchozí šipečkou a koncové stavy (F) dvojitým kroužkem,

– hrana z u do v je označená výčtem všech písmen abecedy, které stav u převádějí na v, tj. $\{x \in \Sigma : \delta(u, x) = v\}$.

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

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.

Poznámka: Mnozí autoři dávají přednost zápisu konečného automatu pomocí tabulky přechodové funkce δ. (Jedním z důvodů zajisté je i to, že tabulku zapíšeme na počítači snadněji než nakreslíme obrázek grafu automatu.) Graf automatu je však mnohem názornější a snadněji uchopitelný. Proto v našem textu budeme důsledně dávat přednost zakreslení automatu jeho grafem.

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 $\leftarrow$.

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

Formálně si postup výpočtu automatu definujeme takto:

`Definice`: *Výpočet konečného automatu* na vstupním slově s probíha následovně.

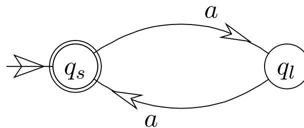
- Začne v počátečním stavu q_0 na začátku slova s.
- Přečte aktuální písmeno x slova s a *přejde* do stavu určeného $\delta(q, x)$, kde q je současný stav automatu. Zároveň se jeho vstup přesune na následující písmeno slova s.
- Předchozí bod se opakuje, dokud nejsou přečtena všechna písmena v s.
- Pokud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

Říkáme také, že jsme dosáhli / nedosáhli přijímající stav.

> `Komentář:` Výpočet automatu $\mathcal{A}$ na slově w si také můžeme představit jako sled v grafu $\mathcal{A}$, 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.

Příklad 2.2. *Navrhněme automat nad jednoznakovou abecedou $\{a\}$ přijímající právě ta slova mající sudou délku.*

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 Cvičení 2.3.

`Definice`: Říkáme, že stav q automatu $\mathcal{A}$ je *dosažitelný* slovem w, pokud výpočet $\mathcal{A}$ se po přečtení (celého) slova w zastaví ve stavu q.

> `Komentář:` Uvědomme si, že v automatu mohou být stavy, do kterých nevede žádný sled z počátku q_0. Takové stavy jsou očividně zbytečné a nazývají se *nedosažitelné*. Je naší přirozenou snahou se takových stavů zbavit.

Normovaný tvar automatu

Jeden automat lze prezentovat mnoha různými způsoby, a proto nás také zajímá nějaká jeho jednoznačná – *normovaná prezentace*.

`Definice`: Automat $\mathcal{A}$ je v *normovaném tvaru* jestliže jeho stavy jsou očíslované $1, 2, \ldots$ v abecedním pořadí nejmenších slov, kterými tyto stavy lze dosáhnout.

`Poznámka`: Touto definicí také automaticky vyloučíme nedosažitelné stavy – stavy, do kterých nevede žádná orientovaná cesta z počátečního stavu.

> `Komentář:` 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. (Viz. přednášky Diskrétní matematiky [4, Lekce 6,8].)

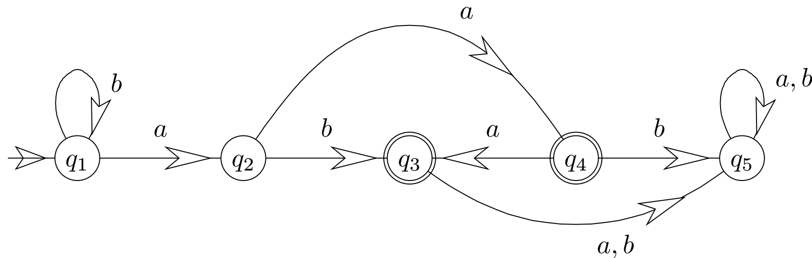
Metoda 2.3. *Převod KA do normovaného tvaru*
(přečíslováním stavů) provede následující jednoduchý algoritmus.

- 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.

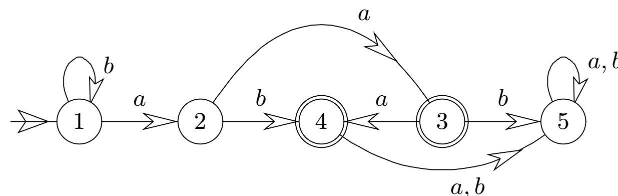
Jedná se vlastně o *procházení grafu do šířky* při seřazení hran podle abecedy.

Příklad 2.4. *Stavy následujícího automatu seřaďte tak, jak mají být číslovány v normovaném tvaru.*



Podle Metody 2.3 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



.

□

Úlohy k řešení

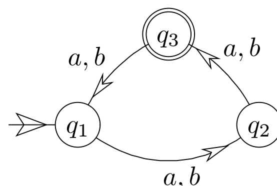
(2.1.1) *Navrhněte konečný automat přijímající všechna ta slova nad abecedou $\{a, b\}$, která obsahují lichý počet výskytů a.*

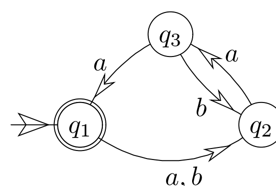
(2.1.2) *Navrhněte konečný automat přijímající všechna ta slova nad abecedou $\{a\}$, jejichž délka dává zbytek 2 po dělení 3.*

(2.1.3) *Jaká všechna slova přijímá automat z úvodu lekce na straně 9?*

(2.1.4) *Jaká všechna slova přijímá automat z Příkladu 2.4?*

(2.1.5) *Které z těchto dvou automatů nad abecedou $\{a, b\}$ přijímají nějaké slovo délky přesně 100?*





(2.1.6) *Nakreslete konečný automat přijímající právě všechna slova nad $\{a, b\}$, ve kterých je třetí znak stejný jak první.*

(2.1.7) *Nakreslete konečný automat přijímající právě všechna slova nad $\{a, b, c\}$, ve kterých se první znak ještě aspoň jednou zopakuje.*

(2.1.8) *Mějme konečný automat $\mathcal{A}$. Jak (jednoduše) sestrojíte automat $\neg\mathcal{A}$ přijímající právě všechna slova, která $\mathcal{A}$ nepřijímá? (Tj. opak $\mathcal{A}$.)*

2.2 Jazyk rozpoznávaný automatem

Jazyk rozpoznávaný (neboli přijímaný, akceptovaný) konečným automatem $\mathcal{A}$ je množinou všech těch slov, které automat přijímá, tj. těch slov, kterými automat $\mathcal{A}$ dosáhne některý z přijímajících stavů. Mnohem formálněji (ale také nepřehledněji) lze definovat tento pojem následovně:

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

1. $\delta^*(q, \varepsilon) = q$,

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

Pak *jazykem $L(\mathcal{A})$ rozpoznávaným* (přijímaným) automatem $\mathcal{A}$ je množina slov

$$L(\mathcal{A}) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}.$$

`Komentář:` Zobecněná přechodová funkce $\delta^*(q, w)$ nám vlastně udává, kam konečný automat přejde ze stavu q přečtením slova w.

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

`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.

Lema 2.5. *Pro konečný automat $\mathcal{A}$ s n stavy je jazyk $L(\mathcal{A})$ neprázdný právě tehdy, když existuje slovo $w \in L(\mathcal{A})$ délky menší než n (tj. $|w| < n$).*

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

Lema 2.6. *Pro konečný automat $\mathcal{A}$ s n stavy je $L(\mathcal{A})$ nekonečný právě tehdy, když existuje $w \in L(\mathcal{A})$ splňující $n \leq |w| < 2n$.*

`Důkaz` (náznak): Pokud existuje sled v grafu automatu $\mathcal{A}$ z počátku 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 $\mathcal{A}$) může mít mnoho cyklů, ale každý z jeden těchto cyklů je délky menší než n, a proto lze postupným vypouštěním cyklů nakonec získat přijímající sled délky mezi n a $2n$. □

`Definice`: Dva konečné automaty $\mathcal{A}_1$, $\mathcal{A}_2$ přijímající shodné jazyky, tj. $L(\mathcal{A}_1) = L(\mathcal{A}_2)$, se také nazývají (jazykově) *ekvivalentní* .

Sjednocení a průnik jazyků

Pro ukázku, které jazyky lze rozpoznávat konečnými automaty, si ukážeme konstrukci automatů pro sjednocení a průnik regulárních jazyků. (Později si ukážeme i jiné zajímavé a užitečné operace, které zachovávají regularitu jazyka.)

Věta 2.7. *Jestliže jazyky $L_1, L_2 \subseteq \Sigma^*$ jsou regulární, pak také jazyky $L_1 \cup L_2$ a $L_1 \cap L_2$ jsou regulární.*

`Důkaz` nejprve pro sjednocení: Nechť $L_1 = L(\mathcal{A}_1)$, $L_2 = L(\mathcal{A}_2)$ pro konečné automaty $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$.

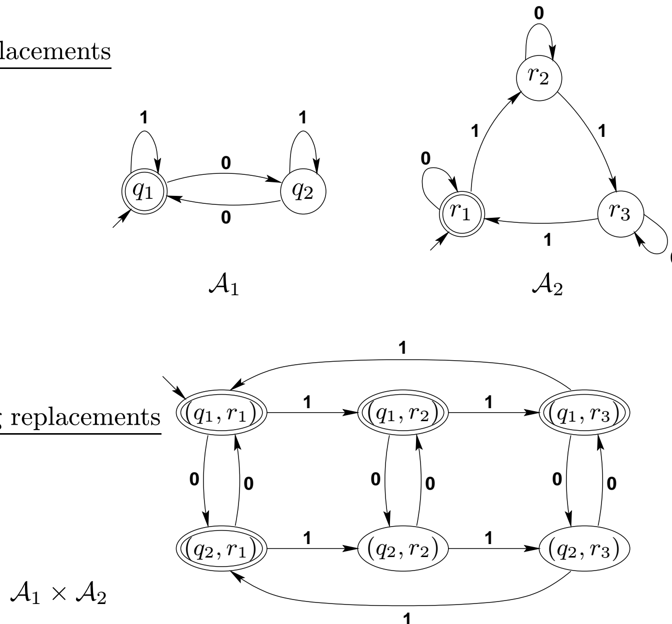
Definujme automat $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ tž.

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

Je očividné (exaktně lze ukázat např. indukcí podle délky $|w|$), že pro libovolné $q_1 \in Q_1$, $q_2 \in Q_2$ 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 $\mathcal{A}$ přejde do stavu (q_1, q_2), kde q_1 je stav automatu $\mathcal{A}_1$ dosažený slovem w a obdobně q_2 je příslušný stav automatu $\mathcal{A}_2$. Z toho snadno plyne, že $L(\mathcal{A}) = L_1 \cup L_2$.

Pro průnik je důkaz téměř stejný, jen výsledná množina přijímajících stavů je $F = F_1 \times F_2$. □

> `Komentář:` Konstrukci uvedenou ve Větě 2.7 si můžeme snadno vizuálně představit – automat $\mathcal{A}$ vypadá jako "mřížka", jejíž sloupce představují automat $\mathcal{A}_1$ a řádky představují automat $\mathcal{A}_2$. Přechody se přitom dějí jak po sloupcích, tak po řádcích zároveň. Přijímající stavy jsou ty, které jsou přijímající aspoň v jednom z automatů $\mathcal{A}_1$ a $\mathcal{A}_2$. Podívejme se na obrázek:



Úlohy k řešení

(2.2.1) *Lze konečným automatem rozpoznávat jazyk všech slov nad abecedou $\{a, b\}$, ve kterých je součin počtů výskytů znaků a a b sudý?*

(2.2.2) *Lze konečným automatem rozpoznávat jazyk všech slov nad abecedou $\{a, b\}$, ve kterých je součet počtů výskytů znaků a a b sudý?*

(2.2.3) *Lze konečným automatem rozpoznávat jazyk všech slov nad abecedou $\{a, b\}$, ve kterých je součet počtů výskytů znaků a a b větší než 100?*

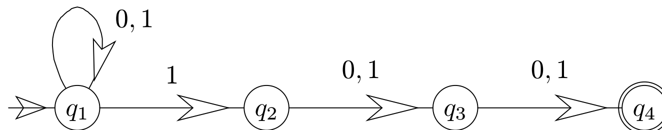
(2.2.4) *Navrhněte automat rozpoznávající všechna ta slova nad $\{a, b\}$, která začínají znakem a a končí znakem b.*

2.3 Nedeterministické automaty

Při konstrukci konečných automatů je v mnoha případech velmi výhodné ponechat automatu možnost se "rozhodovat mezi více přechody", tj. povolit *nedeterminismus*. Co však je tou autoritou, která mezi více přechody rozhodne? Jak pak jednoznačně poznáme, která slova jsou přijata?

Příklad 2.8. *Sestrojme automat přijímající všechna slova nad $\{0, 1\}$, ve kterých je třetí znak od konce 1.*

Sestrojit takový automat podle Definice 2.1 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.



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 2.8 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 2.9. ***(Zobecněný) Nedeterministický konečný automat*** (ZNKA) je uspořádaná pětice $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, kde

– Q je konečná neprázdná množina stavů,

– Σ je konečná neprázdná množina zvaná vstupní abeceda ,

– $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ je (nedeterministická) *přechodová funkce* ,

– $I \subseteq Q$ je neprázdná množina počátečních stavů

– a $F \subseteq Q$ je množina přijímajících (koncových) stavů .

`Komentář:` Rozdíl proti běžnému automatu z Definice 2.1 je v této definici na dvou místech:

- V daném stavu máme při čtení vstupního znaku možnost přechodu do více stavů, nebo také prázdnou 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.

`Poznámka`: Pro dané vstupní slovo tak díky nedeterminismu může existovat mnoho (i nekonečně) různých výpočtů toho samého ZNKA.

`Definice`: Slovo $s \in \Sigma^*$ je *přijímáno nedeterministickým automatem* $\mathcal{A}$, pokud alespoň některý z možných výpočtů $\mathcal{A}$ nad s vede do přijímajícího stavu, tj. při vhodné volbě z nedeterminovaných možností. (Možné výpočty nevedoucí do přijímajícího stavu nás přitom nezajímají.)

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 ZNKA lze jednoduchým postupem převést na ekvivalentní deterministický automat.

Věta 2.10. *Pro každý zobecněný nedeterministický konečný automat $\mathcal{A}$ existuje ekvivalentní (deterministický) konečný automat $\mathcal{A}'$, tj. rozpoznávající stejný jazyk $L(\mathcal{A}) = L(\mathcal{A}')$.*

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

- $Q' = 2^Q$ je množina všech podmnožin stavů Q a $q_0 = I \in Q'$,
- $F' \subseteq Q'$ obsahuje všechny podmnožiny původních stavů Q, které obsahují některý stav z F nebo se z nich dá jen po ε-hranách přejít do 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 $\mathcal{A}$, do kterých se lze v $\mathcal{A}$ dostat z některého stavu v P přechodem po jedné hraně označené x a po libovolném počtu hran (před i po) označených ε.

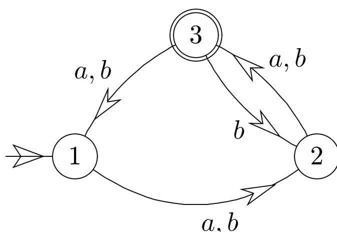
Není těžké nyní zdůvodnit, že nový automat $\mathcal{A}'$ přijímá stejná slova w jako původní $\mathcal{A}$ – po každém kroku výpočtu deterministického $\mathcal{A}'$ aktuální stav q představuje podmnožinu těch stavů $\mathcal{A}$, které lze dosáhnout různými (nedeterministickými) větvemi výpočtu $\mathcal{A}$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

Metoda 2.11. ***Konstrukce determin. automatu z nedeterministického.***
Postup konstrukce přímo vyplývá z důkazu Věty 2.10. Uvědomme si, že sestrojovaný automat $\mathcal{A}'$ má až exponenciální velikost vzhledem k $\mathcal{A}$, ale v praxi nám stačí sestrojit pouze dosažitelné stavy $\mathcal{A}'$, kterých obvykle nebývá tak mnoho. Neboli následovně:

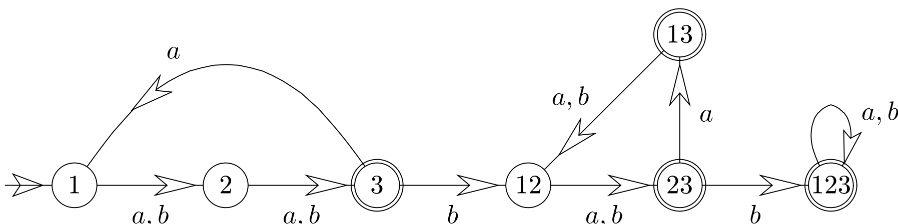
- Začneme se stavem reprezentujícím množinu I počátečních stavů nedeterministického automatu $\mathcal{A}$.
- Dokud máme v sestrojovaném automatu $\mathcal{A}'$ stavy s nedefinovanými přechody, vybereme si jeden takový q a znak x. Pro všechny stavy reprezentované q najdeme všechny možnosti přechodu znakem x v $\mathcal{A}$ a shrneme je v nové množině stavů q' (ta již v našem automatu může být sestrojená).
- Když nový stav reprezentuje množinu, která protíná nebo ze které se v $\mathcal{A}$ dá dostat ε-přechody do F, označíme jej jako přijímající.

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

Příklad 2.12. *Sestrojte k tomuto nedeterministickému automatu ekvivalentní deterministický:*



Postupujeme přesně podle Metody 2.11. 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 $1, 2, 3$ vepisujeme do kroužků bez závorek a čárek, jako 123. Začneme v množině stavů $\{1\}$ a zleva doprava sestrojíme následující automat:



□

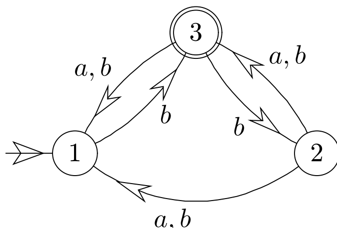
`Poznámka`: Závěrem se krátce zmíníme o tzv. "chybovém stavu" automatu. V praktických příkladech konstrukce automatů se obvykle stává, že po přečtení některých nechtěných posloupností znaků automat přechází do stavu, který není přijímající a ve kterém už navždy zůstává. Takovému stavu se pak přirozeně říká *chybový*.

Ve zjednodušených zobrazeních automatu se pak 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! (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.)

Úlohy k řešení

(2.3.1) *Kdy automat z následující Úlohy 2.3.2 přijímá slovo složené ze samých písmen a?*

(2.3.2) *Sestrojte ekvivalentní deterministický automat k tomuto:*



(2.3.3) *Kdy automat z Úlohy 2.3.2 přijímá slovo složené ze samých písmen b?*

***(2.3.4)** *Uměli byste slovně (a názorně) popsat jazyk přijímaný automatem z Úlohy 2.3.2?*

(2.3.5) *Navrhněte ZNKA přijímající jazyk všech těch slov nad $\{a, b\}$, které končí sufixem 'abb' nebo sufixem 'aa'.*

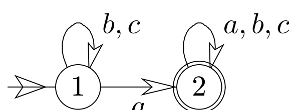
Rozšiřující studium

Čtenářům, kteří chtějí lépe "vidět", jak vlastně konečný automat pracuje, vřele doporučujeme shlédnutí počítačových animací, které jsou přílohami skript [8]. Tyto animace jsou pro vás extrahované na web stránkách výuky [5].

2.4 Cvičení: Konstrukce a převody automatů

Příklad 2.13. *Existuje konečný deterministický 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 zde nakreslete.*

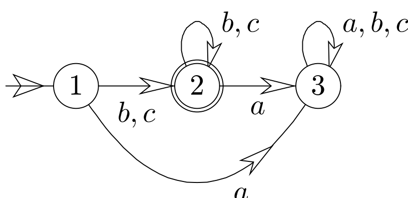
Existuje, stačí se po prvním přečtení znaku a přesunout do přijímajícího stavu, ve kterém už zůstaneme.



□

Příklad 2.14. *Existuje konečný deterministický 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.)*

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:



□

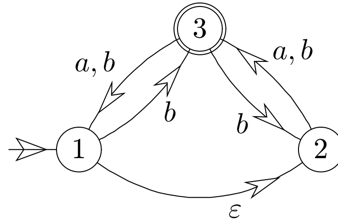
Příklad 2.15. *Jak poznáme, že dva konečné automaty $\mathcal{A}_1$ a $\mathcal{A}_2$ přijímají shodné jazyky, tj. zda $L(\mathcal{A}_1) = L(\mathcal{A}_2)$?*

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 $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$:

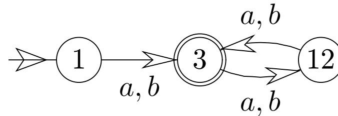
Podle Úlohy 2.1.8 sestrojíme automat $\neg\mathcal{A}_2$ přijímající opak (doplněk) jazyka $L(\mathcal{A}_1)$. Poté sestrojíme podle Věty 2.7 automat $\mathcal{B}$ přijímající průnik jazyků $L(\mathcal{A}_1) \cap L(\neg\mathcal{A}_2)$. Elementární poznatky teorie množin nám říkají, že $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ právě když $L(\mathcal{A}_1) \cap L(\neg\mathcal{A}_2) = \emptyset$. Takže nám stačí ověřit, že automat $\mathcal{B}$ nepřijímá žádné slovo, neboli že v grafu $\mathcal{B}$ nevede žádná orientovaná cesta z počátečního do přijímacího stavu. (Pokud by naopak $\mathcal{B}$ přijímal nějaké slovo w, pak by w rozlišovalo naše dva automaty.)

Symetricky si pak ověříme, zda $L(\mathcal{A}_2) \subseteq L(\mathcal{A}_1)$. Pokud to opět vyjde, celkově dojdeme k závěru, že $L(\mathcal{A}_1) = L(\mathcal{A}_2)$. □

Příklad 2.16. *Sestrojte ekvivalentní deterministický automat k tomuto:*



Postupujeme přesně podle Metody 2.11 a automat tentokrát vyjde malý:

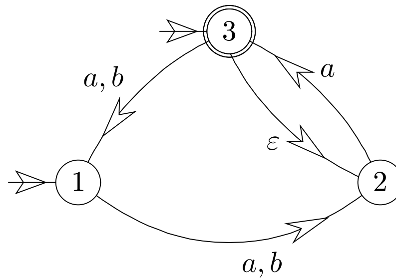


Všimněme si, že výsledný automat vlastně jenom počítá paritu délky vstupního slova a vůbec nezáleží na tom, který ze znaků a, b přijde na vstup. □

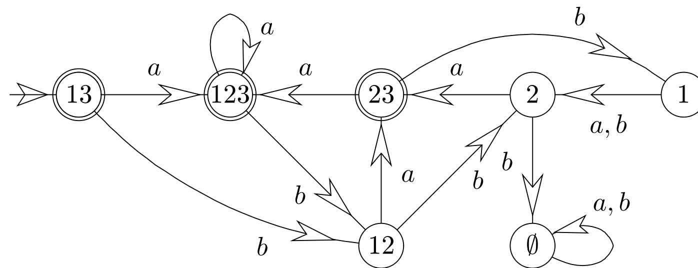
Příklad 2.17. *Je deterministický automat sestrojený v Příkladě 2.16 nejmenší možný pro svůj jazyk?*

Není, snadno je vidět, že počáteční stav 1 lze sloučit se stavem 12. (Později si uvedeme více o tzv. minimalizaci automatu.) □

Příklad 2.18. *Následující zobecněný nedeterministický konečný automat převeďte na deterministický bez nedosažitelných stavů.*



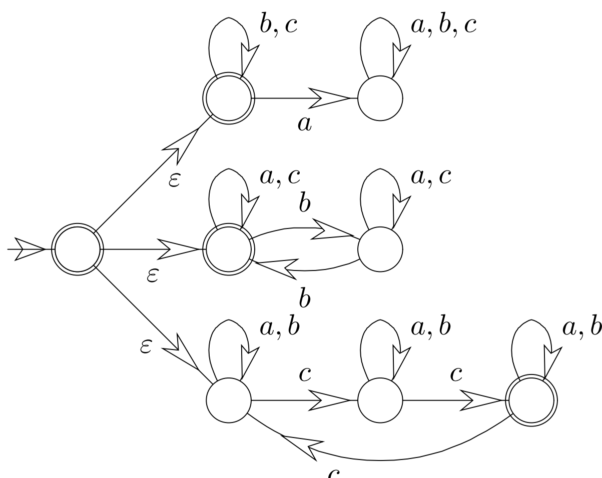
Pozor, všimněme si nejprve, že daný automat má dva počáteční stavy, takže počáteční stav deterministického automatu bude tvořen množinou $\{1, 3\}$. Dalším bodem k zamyšlení je hned přechod znakem a z $\{1, 3\}$ – přímými přechody se lze dostat do stavů 2 a 1, ale navíc se můžeme dostat i do stavu 3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

Příklad 2.19. *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.*

Požadovaný automat jednoduše poskládáme z opaku automatu v Příkladě 2.13 a z automatů v Příkladě 2.2 a v Úloze 2.1.2. 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? □

Úlohy k řešení

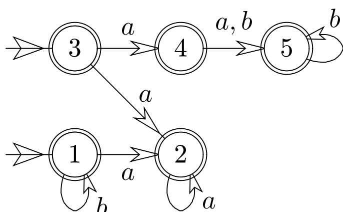
(2.4.1) *Navrhněte konečný deterministický automat přijímající právě ta slova nad abecedou $\{a, b, c, d\}$, které nezačínají a, druhý znak nemají b, třetí znak nemají c a čtvrtý znak nemají d. (Včetně těch s délkou < 4.)*

(2.4.2) *Navrhněte konečný deterministický automat přijímající právě ta slova nad abecedou $\{a, b, c, d\}$, které nezačínají a nebo druhý znak nemají b nebo třetí znak nemají c nebo čtvrtý znak nemají d.*

(2.4.3) *Sestrojte deterministický 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.

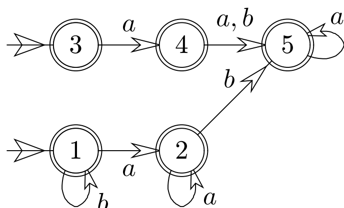
(2.4.4) *Sestrojte deterministický konečný automat přijímající všechna ta slova délky aspoň 2 nad abecedou $\{a, b\}$, ve kterých nejsou poslední dva znaky stejné.*

(2.4.5) *Najděte libovolné slovo nad abecedou $\{a, b\}$, které* **ne***patří do jazyka přijímaného tímto nedeterministickým automatem se dvěma počátečními stavy:*

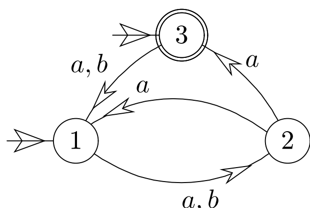


Návod: Pozor, přestože všechny stavy jsou přijímající, odpověď není tak triviální.

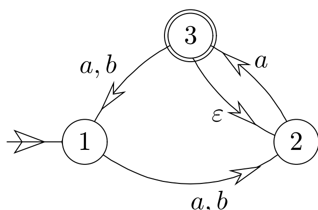
(2.4.6) *Najděte libovolné slovo nad abecedou $\{a, b\}$, které* **ne***patří do jazyka přijímaného tímto nedeterministickým automatem se dvěma počátečními stavy:*



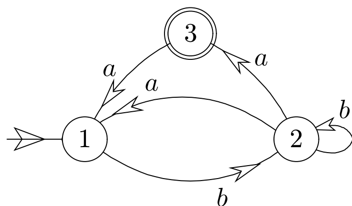
(2.4.7) *Následující zobecněný nedeterministický konečný automat převeďte na deterministický bez nedosažitelných stavů.*



(2.4.8) *Následující zobecněný nedeterministický konečný automat převeďte na deterministický bez nedosažitelných stavů.*



***(2.4.9)** *Slovně popište jazyk přijímaný následujícím nedeterministickým automatem.*



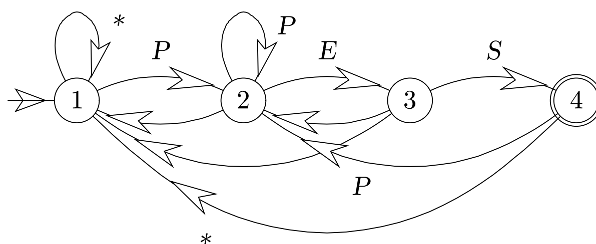
3 Vyhledávání a regulární výrazy

Úvod

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.



(Pro vysvětlení, náš automat hledá po sobě znaky P,E,S, přitom při výskytu jiných znaků se vrací zase na začátek.)

Co je důležité, automaty umí v textu vyhledávat nejen fixní slova, ale i proměnlivé vzorky textu podle popsané struktury. Přesněji řečeno, konečným automatem lze vyhledávat v daném textu všechny výskyty slov z nějakého regulárního jazyka. A tato aplikace nás pak přímo přivádí k otázce, jak bychom dokázali symbolicky (textově) zapsat nějaký regulární jazyk. K tomu účelu si popíšeme tzv. regulární výrazy, se kterými jste se již mohli dříve setkat na vašem počítači (třeba u příkazu `grep`*).*

Cíle

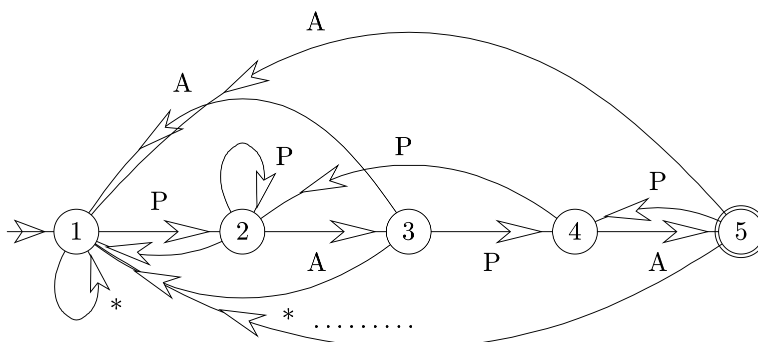
Naším cílem je ukázat teoretický základ algoritmů pro vyhledávání vzorků v textu a také formálně zavést tzv. regulární výrazy pro symbolický zápis celým tříd slov (právě regulárních jazyků, jak uvidíme). Čtenář by měl pochopit sílu regulárních výrazů a naučit se je používat v praxi.

3.1 Automaty a vyhledávání v textu

Navrhnout automat přijímající právě jedno dané slovo s je snadné. Uvědomte si však, že vyhledání slova v textu je *úplně jiný úkol* — představme si jej jako automat, který čte vstupní text a projde na konci každého výskytu hledaného slova přijímajícím stavem. Formálně řečeno, tento automat má přijímat právě všechna slova mající hledané slovo s jako sufix (na konci).

Příklad 3.1. *Navrhněte konečný automat přijímající právě ta slova nad ASCII abecedou, která mají za sufix 'PAPA'.*

Je docela zřejmé, že základem našeho automatu bude posloupnost přechodů $\to P, \to A, \to P, \to A$ 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 'PAPAPA'. Souhrnem těchto úvah získáme výše nakreslený výsledný automat. □

`Fakt`: Simulací automatu z předchozího Příkladu 3.1 a sledováním průchodů jeho přijímajícími stavy získáme ten nejrychlejší algoritmus pro vyhledávání vzorků v textu.

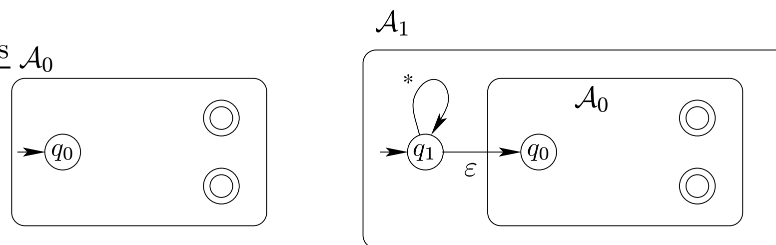
Obecně můžeme vyhledávat v textu nejen jednotlivá slova, ale také vzorky patřící do libovolného regulárního jazyka. To nám (dokonce konstruktivně) umožňuje následující tvrzení:

Věta 3.2. *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.

PSfrag replacements



Nakonec $\mathcal{A}_1$ (případně) převedeme dle Věty 2.10 na deterministický automat. □

`Komentář:` Pokud bychom konstrukci uvedenou v důkaze Věty 3.2 aplikovali na automat z Příkladu 3.1, vyšel by nám (po sloučení v podstatě zbytečného přidaného stavu q_1 s původním počátečním stavem) stejný deterministický automat. Zkuste si to sami.

Úlohy k řešení

(3.1.1) *Sestrojte deterministický konečný automat vyhledávající v textu slovo 'TATAR'. Udělejte to jak heuristickým přístupem popsaným v Příkladě 3.1, tak i formálním postupem podle Věty 3.2. Vyšly vám oba automaty stejně?*

(3.1.2) *Sestrojte deterministický konečný automat vyhledávající v textu nad abecedou $\{a, b\}$ místa, ve kterých se poslední znak opakuje nejvýše dvakrát. (Prázdné slovo nechceme.)*

(3.1.3) *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?

3.2 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 nástroje na vyhledávání a zpracovávání v textu, známé především z unixových systémů. (Avšak dostupné volně pro všechny POSIX systémy. Zájemce z řad obdivovatelů M$ Windows odkazujeme na projekt CygWin [1].)

Příklad 3.3. *Vyhledejme ze zdrojových kódů v jazyce C všechny řádky obsahující přiřazení celých čísel do proměnné* `i`*.*

Pro vyhledávání tohoto typu je přímo stvořený příkaz `grep` [3]. 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:

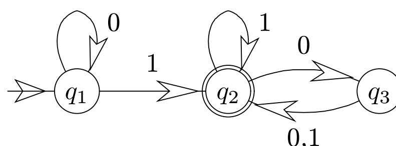
```
grep '\<i *= *[-+]\?[0-9]\+ *[;,]' *.c
```

□

`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` [2]. (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.

Příklad 3.4. *Simulujte na počítači následující automat:*



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 3.15.) □

`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`.

3.3 Regulární operace a výrazy

Až nyní přistoupíme k formální definici *regulárních výrazů*, které byly již výše používány při praktickém vyhledávání v textu. Čtenáře nechť nezaskočí, že v teorii se používá poněkud jiná syntaxe zápisu regulárních výrazů, než jaká je zvykem v unixových systémech. Význam regulárních výrazů je však stále stejný.

Definice 3.5. ***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).

`Poznámka`: Všimněte si, že regulární operace nezahrnují průnik dvou jazyků nebo doplněk (přitom regulární jazyky jsou uzavřené také na tyto operace). To ale neznamená, že bychom třeba průnik dvou jazyků nebyli vůbec schopni pomocí regulárních operací vyjádřit. Možné to obecně je, jen se na to musí jít "oklikou".

Hlavní význam regulárních operací je v tom, že jimi můžeme zapisovat různé jazyky. K tomu si však nejprve musíme domluvit správnou "syntaxi" takového zápisu – nazýváme ji regulárním výrazem.

Definice 3.6. ***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:` Výše uvedené příklady jazyků zapsaných regulárními operacemi se v zápisu regulárními výrazy vyjádří
$(0+11)^* \cdot (000+11)^*$,
$(0+11)^* + (010, 101)^*$.

`Definice`: 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 $((((0 \cdot 1)^* \cdot 1) \cdot (1 \cdot 1)) + ((0 \cdot 0) + 1)^*)$ napíšeme $(01)^*111 + (00+1)^*$.

Regulární výrazy, tak jak je formálně definujeme zde, se syntaxí velmi 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 (podívejte se na `man regexp` [11]).

`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!

Úlohy k řešení

(3.3.1) *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?*

(3.3.2) *A co když v předchozí úloze vyžadujeme, že úsek a i úsek b musí být neprázdný?*

(3.3.3) *A co když v předchozí úloze ještě povolíme, že znak c se mezi a a b může vyskytnout 0-, 1- nebo 2-krát?*

(3.3.4) *Zjistěte, zda platí* $\left[(011+(10)^*1+0)^*\right] \;=?\; \left[011(011+(10)^*1+0)^*\right]$.

(3.3.5) *Zjistěte, zda platí* $\left[((1+0)^*100(1+0)^*)^*\right] \;=?\; \left[((1+0)100(1+0)^*100)^*\right]$.

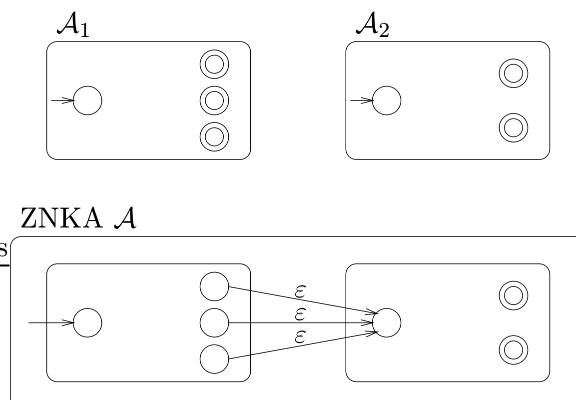
3.4 Ekvivalence regulárních výrazů a jazyků

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

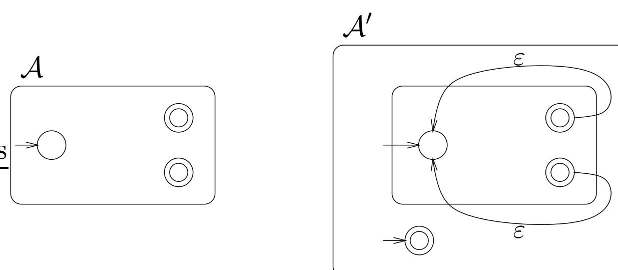
Důkaz: Nechť $\mathcal{A}_1, \mathcal{A}_2$ jsou automaty přijímající jazyky L_1, L_2. Jejich následujícím složením sestrojíme ZNKA přijímající zřetězení $L_1 \cdot L_2$:

PSfrag replacements



Pro druhou část důkazu vyjdeme z automatu $\mathcal{A}$ přijímajícího jazyk L a sestrojíme odvozený ZNKA $\mathcal{A}'$ přijímající iteraci L^*:

PSfrag replacements



□

Následující tvrzení se již implicitně vyskytlo dříve, nyní si jej znovu uvedeme explicitně pro úplnost.

Tvrzení 3.8. *Jestliže L je regulární, pak také jeho doplněk $\overline{L}$ je regulární. Jestliže L_1, L_2 jsou regulární jazyky, pak také rozdíl $L_1 \setminus L_2$ je regulární.*

Důkaz: Nechť $L = L(\mathcal{A})$, kde $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$. Pak $\overline{L}$ je zřejmě rozpoznáván KA $(Q, \Sigma, \delta, q_0, Q{-}F)$. (Přijímající a nepřijímající stavy byly prohozeny.)

Druhá část tvrzení pak již plyne z toho, že $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$. □

Převod mezi automatem a výrazem

Věta 3.9. *Regulárními výrazy lze reprezentovat právě regulární jazyky.*

Důkaz (náznak): Nejprve ukážeme, že ke každému regulárnímu výrazu α lze sestrojit konečný automat přijímající jeho jazyk $[\alpha]$. Atomické výrazy snadno přijmeme automatem. Ke sjednocení dvou jazyků pak sestrojíme induktivně automat podle Věty 2.7, pro zřetězení nebo iteraci obdobně podle Věty 3.7. Takto indukcí podle délky regulárního výrazu α vždy sestrojíme příslušný automat.

V opačném směru – jak popsat jazyk konečného automatu regulárním výrazem, uvedeme jen velmi hrubý nástin:
Slovo w je přijímáno automatem $\mathcal{A}$ právě když v grafu automatu existuje sled (ohodnocený) 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. Pak již stačí pomocí zřetězení a iterací popsat jednotlivé podcesty a cykly na

těchto přijímacích sledech a vše dát dohromady...
Celý důkaz by však byl velmi obtížný. □

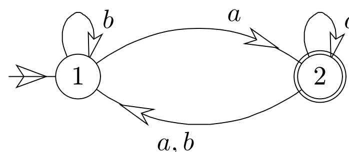
Otázkou je, jak pro daný konečný automat najdeme regulární výraz popisující jeho jazyk? Důkaz Věty 3.4 je sice svým způsobem konstruktivní, ale jen těžko si lze představit, že u větších automatů najdeme všechny možné cesty od počátečního do přijímajících stavů najednou. Jiný přístup ke hledání takových cest (přesněji řečeno sledů) poskytuje modifikace klasického algoritmu pro výpočet metriky grafu [4, Část 8.2].

Metoda 3.10. ***Výpočet regulárního výrazu z automatu***
Daný automat nemusí být deterministický. Jeho stavy libovolně očíslujeme $1, 2, \dots, n$ 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 $t = 1, 2, \dots, n$ 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 3.11. *Pro přiblížení Metody 3.10 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. □

Úlohy k řešení

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

(3.4.2) *Upravte předchozí automat, aby přijímal jazyk zapsaný* $(0+11)^*00^*1$

(3.4.3) *Jak byste zapsali regulárním výrazem jazyk přijímaný automatem z Příkladu 3.4?*

*(3.4.4)** *Jak byste zapsali regulárním výrazem jazyk přijímaný automatem z Příkladu 2.12?*

Rozšiřující studium

Pro další, hlavně prakticky orientované, studium problematiky vyhledávání v textu a regulárních výrazů je nejlépe čtenáře odkázat na dokumentaci knihovny `regexp` *[11] a programu* `grep` *[3]. Dokumentaci lze standardně nalézt také v Linuxových distribucích.*

3.5 Cvičení: Vyhledávání a regulární výrazy

Příklad 3.12. *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++)`*'.)*

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). □

Příklad 3.13. *Najděte v daném textu všechny výskyty úplných (a pokud možno platných) e-mailových adres, oddělených mezerami či konci řádků.*

Jak jistě znáte, 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! □

Příklad 3.14. *Navrhněte regulární výraz, který zkontroluje, jestli vstupní text (celý od začátku do konce řádku) vypadá jako platná e-mail adresa.*

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.) □

Příklad 3.15. *Simulace automatu uvedená v Příkladu 3.4 má kompletní zdrojový kód (flexu) uvedený zde. Zkuste si jej na nějakém Linuxovém stroji přeložit a spustit!*

```
/*
compile with "flex flex.l; gcc lex.yy.c",
run as "echo -n '01001001' | ./a.out".
*/

/*%option debug*/
%option noyywrap
%s Q2 Q3
%{
#include "stdio.h"
%}

%%
/*
flex rules start here (with meaning like in an automaton):
*/
<INITIAL>0      ;
<INITIAL>1      BEGIN(Q2);
<Q2>1           ;
<Q2>0           BEGIN(Q3);
<Q3>0|1         BEGIN(Q2);
<Q2><<EOF>>     printf("slovo přijato!\n");  return;
.|\n            printf("neznámý znak %s na vstupu\n",yytext);

%%
/*
supplementary C code for running the program:
*/
void main(void) {
yylex();
}
```

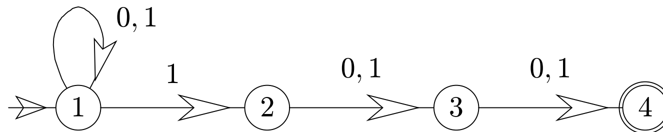
□

Příklad 3.16. *Další příklad filtru ve flexu následuje zde. Jedná se o jednoduchý program, který ze vstupu čte text a dívá se, zda jde o celé nebo desetinné číslo nebo jiný text.*

```
/* nepouziva na konci souboru fci yywrap() => neni potreba linkovat s knihovnou fl */
%option noyywrap

/* deklarace vsech vnitrnich stavu*/
%s q1 q2 q3 q4 q5 q6

%{ /* deklarace globalnich promennych */
char m[255];
char *m_ptr=m;
%}

%%

<INITIAL>-{strcat(m,yytext);BEGIN(q2);
/* vstupni stav, jestlize bude zacinat cislo - tak se presune do */
/* stavu q2, do promenne m postupne kopiruje symboly ze vstupu*/}

<INITIAL>[0-9] {strcat(m,yytext);BEGIN(q3);}
<INITIAL>"." {strcat(m,yytext);BEGIN(q4);}
<q2>[0-9] {strcat(m,yytext);BEGIN(q3);}
<q2>"." {strcat(m,yytext);BEGIN(q4);}
<q3>[0-9] {strcat(m,yytext);BEGIN(q3);}
<q3><<EOF>>{printf("cele cislo %s\n",m);yyterminate();
/* jestlize bude automat ve stavu q3 a zaroven na konci vstupniho */
/* souboru, pak vytiskne cislo a fci yyterminate() ukonci cinnost */}

<q3>"." {strcat(m,yytext);BEGIN(q4);}
<q4>[0-9] {strcat(m,yytext);BEGIN(q5);}
<q5>[0-9] {strcat(m,yytext);BEGIN(q5);}
<q5><<EOF>>{printf("desetinne cislo %s\n",m);yyterminate();
/* jestlize bude automat ve stavu q5 a zaroven na konci vstupniho */
/* souboru, pak vytiskne cislo a fci yyterminate() ukonci cinnost */}

<q6><<EOF>>{printf("%s neni cislo\n",m);yyterminate();
/* jestlize bude automat ve stavu q6 a zaroven na konci vstupniho */
/* souboru, pak vytiskne cislo a fci yyterminate() ukonci cinnost */}

. {strcat(m,yytext);BEGIN(q6);
/* jestlize neni rozpoznan symbol libovolnym pravidlem, pak prechazi */
/* automat do stavu q6, kterym jsou rozpoznana slova, ktera nejsou cisly */}

%%
/* vlastni kod programu */

int main()
{
yylex(); /* funkce, ktera spousti automat*/
return 0;
}
```

Jedná se hlavně o výukový kód demonstrující možnosti flexu, který by však bylo možno na druhou stranu zjednodušit a zkrátit. (Například čtením celých sekvencí znaků–číslic regulárními podvýrazy.) Spusťte si jej a pak se pokuste kód vylepšit třeba o čtení čísel v exponenciální notaci. □

Příklad 3.17. *Podle postupu Metody 3.10 sestrojte regulární výraz popisující jazyk tohoto automatu:*



Jedná se o automat z Příkladu 2.8, 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? □

Úlohy k řešení

(3.5.1) *Zapište regulárním výrazem jazyk všech slov nad abecedou $\{0,1\}$ neobsahující tři stejné znaky za sebou.*

(3.5.2) *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).*

(3.5.3) *Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a,b,c\}$, ve kterých se nikde nevyskytují dva znaky a hned za sebou.*

(3.5.4) *Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a,b,c\}$, ve kterých je po a vždy b a po b vždy a.*

(3.5.5) *Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a,b,c\}$, ve kterých je po a vždy b a po b nikdy není c.*

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

***(3.5.7)** *Zapište regulární výraz pro* `grep` *hledající všechny ty řádky zdrojového kódu C, na kterých je proměnná* `xyz`*, ale ne uvnitř řádkového komentáře* `//...`.

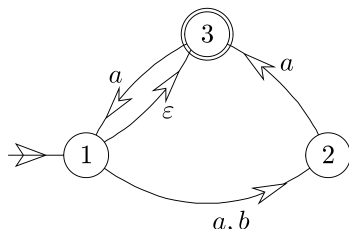
***(3.5.8)** *Mějme dva regulární jazyky K a L popsané regulárními výrazy*

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

a) Jaké je nejkratší a nejdelší slovo v průniku $L \cap K$?
b) Proč žádný z těchto jazyků K, 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!

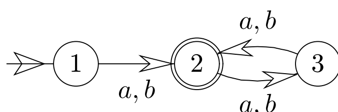
(3.5.9) *Sestavte regulární výraz popisující jazyk přijímaný následujícím zobecněným nedeterministickým automatem. Použijte buď tabulkovou metodu, nebo vlastní (správnou) úvahu.*



4 Minimalizace a omezení automatů

Úvod

Vezmeme-li v úvahu praktické (implementační) hledisko, jistě není třeba sáhlodlouze motivovat otázku možné minimalizace daného konečného automatu. Zdůrazňujeme, že zde máme na mysli minimalizaci výhradně deterministického automatu. Jak jsme si již uvedli dříve, 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.

Další zajímavou otázkou je, jak vůbec poznat, zda pro daný jazyk existuje automat jej rozpoznávající. Čtenář asi tuší, že některé jazyky automaty nemohou být rozpoznávány. Například jazyk všech správně vyvážených závorkových výrazů (jako '(()(())()())') automat nerozpozná, protože si v konečně mnoha svých stavech neumí "spočítat" levé a pravé závorky. Co lze na takové neformální zdůvodnění říci matematicky? To bude náplní druhé části.

Cíle

V této lekci si především ukážeme, jak lze daný automat mechanickým postupem jednoznačně minimalizovat. Důležitým teoretickým přínosem lekce je uvedení kritérií, která umožňují rozhodnout, zda daný jazyk je vůbec rozpoznatelný konečným automatem.

4.1 Minimalizace konečného automatu

`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.

Pro úplnost si zopakujeme definici ekvivalentních automatů z Lekce 2. Na ni navážeme přirozenou definicí minimálního automatu (vzhledem k počtu jeho stavů).

`Definice`: Dva konečné automaty $\mathcal{A}_1$, $\mathcal{A}_2$ nazveme *(jazykově) ekvivalentní*, jestliže přijímají týž jazyk, tj. jestliže $L(\mathcal{A}_1) = L(\mathcal{A}_2)$.

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

Před uvedením samotného postupu minimalizace je vhodné si osvěžit znalosti o rozšířené přechodové funkci $\delta^*(q,w)$ automatu (strana 13) udávající, do kterého stavu je náš automat převeden z libovolného stavu q přečtením (pod)slova w. Myšlenka sledovat množiny všech podslov (sufixů), která nějaký stav q převedou do přijímajících stavů, je zde tou hlavní ideou.

Věta 4.1. *Existuje rychlý algoritmus, který k zadanému konečnému automatu $\mathcal{A}$ sestrojí minimální (deterministický) automat ekvivalentní s $\mathcal{A}$.*

Tuto větu postupně dokážeme několika jednoduchými tvrzeními a následným popisem tohoto minimalizačního algoritmu. Zavedeme nejdříve hlavní ideový pojem.

`Značení`: Uvažujme konečný deterministický automat $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ bez nedosažitelných stavů. Pro každý stav $q \in Q$ definujme

$$L_{\mathcal{A}}(q) = L(\mathcal{A}_q), \text{ kde } \mathcal{A}_q = (Q, \Sigma, \delta, q, F)\,.$$

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

Nyní dokážeme, že různé jazyky $L_{\mathcal{A}}(q) \neq L_{\mathcal{A}}(q')$ na stavech q, q' automatu nutně rozliší tyto stavy.

Lema 4.2. *Mějme v automatu $\mathcal{A}$ dva stavy $q_1, q_2 \in Q$ takové, že $L_{\mathcal{A}}(q_1) \neq L_{\mathcal{A}}(q_2)$. Nechť stav q_i, $i = 1,2$ je dosažitelný v $\mathcal{A}$ z počátku q_0 slovem z_i. Pak každý automat $\mathcal{A}'$ ekvivalentní $\mathcal{A}$ musí být slovem z_1 převeden do jiného stavu než slovem z_2.*

`Důkaz`: Nechť $y \in L_{\mathcal{A}}(q_1) \setminus L_{\mathcal{A}}(q_2)$ (nebo symetricky naopak). Podle definice tedy je $z_1 y \in L(\mathcal{A})$, ale $z_2 y \notin L(\mathcal{A})$. Pokud by nějaký (jakýkoliv) automat $\mathcal{A}'$ nad Σ byl slovem z_1 převeden do stejného stavu q_z jako slovem z_2, pak by $\mathcal{A}'$ zároveň přijal nebo zároveň nepřijal obě slova $z_1 y$ i $z_2 y$. (Uvědomte si dobře, že od stavu q_z již v obou případech automat $\mathcal{A}'$ čte tentýž sufix y a přitom je deterministický.) To by však $\mathcal{A}'$ nemohl být ekvivalentní našemu automatu $\mathcal{A}$, spor. □

Z předchozího lematu již velmi snadno vyvodíme následující důležité důsledky.

Důsledek 4.3. *Automat, ve kterém každý stav je dosažitelný a pro každé dva jeho stavy $q \neq q'$ platí $L_{\mathcal{A}}(q) \neq L_{\mathcal{A}}(q')$, je minimální.*

Důsledek 4.4. *Každé dva minimální automaty pro stejný jazyk jsou si isomorfní.*

> `Komentář`: Zde se již jasně rýsuje *idea konstrukce minimálního automatu.* Všechny stavy $q \in Q$ automatu A mající stejný jazyk $L_{\mathcal{A}}(q)$ "sloučíme" vždy do jednoho stavu a výsledný automat $\mathcal{A}_0$ pak bude dle Důsledku 4.3 minimální. Jak však budeme algoritmicky porovnávat jazyky $L_{\mathcal{A}}(q)$ pro $q \in Q$ mezi sebou? A bude naše "sloučení" stavů konzistentní, tj. budou šipky ze dvou sloučených stavů mířit do také stejně sloučených stavů? Odpověď na obě otázky nám dá přímo následující algoritmus.

`Značení`: Mějme množinu M, která je rozložená do sjednocení k podmnožin $M = N_1 \cup N_2 \cup \ldots \cup N_k$, které jsou vzájemně disjunktní, tj. $N_i \cap N_j = \emptyset$ pro všechny indexy $i \neq j$. Pak podmnožiny $N_1, \ldots, N_k$ nazýváme *rozkladem* množiny M a značíme jej jako systém (množinu) množin $\mathcal{R} = \{N_1, N_2, \ldots, N_k\}$.

Následuje metoda odvození minimálního automatu, která bude postupně rozkládat množinu stavů daného automatu, až už nebude možné mezi stavy každé jedné podmnožiny rozkladu nijak rozlišit.

Metoda 4.5. ***Minimalizace konečného automatu***
Daný je deterministický automat $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ bez nedosažitelných stavů.

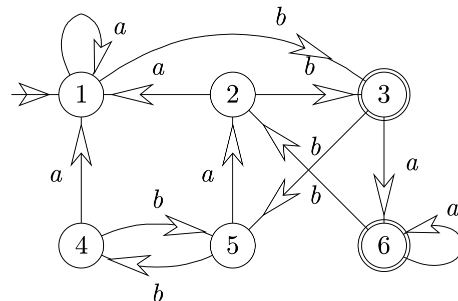
- Začneme rozkladem $\mathcal{R}_0 = \{Q \setminus F,\ F\}$ množiny všech stavů $\mathcal{A}$ na ty nepřijímající a přijímající.
- Nechť v kroku $k \geq 0$ máme rozklad $\mathcal{R}_k = \{P_1, P_2, \ldots, P_m\}$ množiny všech stavů. Pro všechna $i \in \{1, 2, \ldots, m\}$ a $a \in \Sigma$ uděláme následovné:
  – Rozložíme P_i na rozklad $\mathcal{P}_i^a$ podle toho, do kterých množin z $\mathcal{R}_k$ vedou ze stavů v P_i šipky se symbolem a.
- Uděláme sjednocení průniků těchto (mini-)rozkladů $\mathcal{R}_{k+1} = \bigcup_i \bigcap_a \mathcal{P}_i^a$. Tím získáme všechna možná další rozlišení uvnitř tříd rozkladu $\mathcal{R}_k$ všemi znaky abecedy.
- Pokud $\mathcal{R}_{k+1} \neq \mathcal{R}_k$, 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 $\mathcal{R}_k$ (reprezentanti, přitom q_0 je také vybráno) a δ' je restrikce přechodové funkce δ na jednotlivých třídách rozkladu R_k. Pak minimální automat je $\mathcal{A}_0 = (R, \Sigma, \delta', q_0, F \cap R)$.

Důkaz (náznak): Jelikož automat A má jen n stavů, nejvýše $n - 2$-krát v průběhu algoritmu může dojít ke zjemnění rozkladu $\mathcal{R}_k$ všech stavů. Proto algoritmus skončí po nejvýše $n - 1$ iteracích.

Pro zdůvodnění správnosti si všimněme, že v kroku $k \geq 0$ třídy rozkladu $\mathcal{R}_k$ vyjadřují rozlišitelnost stavů automatu $\mathcal{A}$ pomocí slov délky nejvýše k (snadno lze dokázat indukcí podle k). Pokud v některém kroku k náš postup už dalšího rozlišení nedosáhne, jsou již stavy $\mathcal{A}$ v jednotlivých podmnožinách rozkladu $\mathcal{R}_k$ (napořád) nerozlišitelné. Navíc, vzhledem k nemožnosti dalšího rozlišení stavů v rozkladu $\mathcal{R}_k$ je jasné, že výsledný automat $\mathcal{A}_0$ nebude záviset na tom, které reprezentanty stavů R v posledním bodu vybereme. □

Komentář: Při praktickém použití Metody 4.5 postupujeme tak, že si stavy automatu $\mathcal{A}$ v jednotlivých podmnožinách rozkladu $\mathcal{R}$ 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.

Příklad 4.6. *Minimalizujme následující automat:*



Podle komentáře k Metodě 4.5 vyplníme symbolickou přechodovou tabulku pro prvotní rozklad $\mathcal{R}_0 = \{\{1,2,4,5\},\{3,6\}\}$:

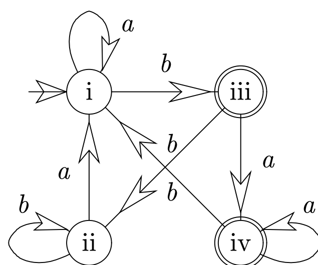
	a	b
1	i	ii
2	i	ii
4	i	i
5	i	i
3	ii	i
6	ii	i

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.



□

Porovnání automatů

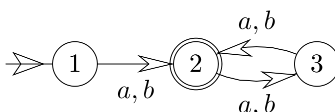
Věta 4.7. *Existuje algoritmus, který pro zadané KA* $\mathcal{A}_1$, $\mathcal{A}_2$ *rozhodne, zda jsou ekvivalentní, tj. zda* $L(\mathcal{A}_1) = L(\mathcal{A}_2)$.

`Důkaz`: Stačí k oběma KA sestrojit minimální automaty v normovaném tvaru a ty přímo porovnat (Důsledek 4.4). Jiná možnost porovnání dvou automatů již byla prezentovaná v Příkladě 2.15. □

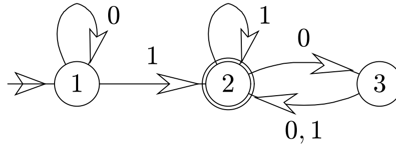
`Poznámka`: Všimněte si, že jsme ukázali u algoritmu minimalizace, že pokud dva stavy automatu, který má n stavů ($n \geq 2$), nelze rozlišit slovem délky nejvýše $n-2$, pak je nelze rozlišit vůbec. Tento fakt ukazuje, že pro rozhodování otázky, zda $L(q) = L(q')$, stačí probrat všechna slova do délky $n-2$ (jichž je samozřejmě konečně mnoho). Ovšem algoritmus založený na této myšlence by byl pro praktické použití velmi nevhodný, protože čas jeho výpočtu roste exponenciálně s velikostí daných automatů.

Úlohy k řešení

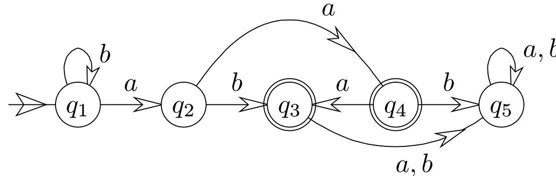
(4.1.1) *Minimalizujte podle uvedeného postupu automat:*



(4.1.2) *Je tento automat minimální?*



(4.1.3) *Je tento automat minimální?*



(4.1.4) *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?*

4.2 Neregulární jazyky

Je poměrně zřejmé, že konečné automaty jsou natolik primitivní stroje, že nemohou rozpoznávat všechny možné jazyky. Jak ale vypadá nějaký konkrétní neregulární jazyk?

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

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

(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!

Náplní této sekce je podání rigorózního argumentu pro neregulárnost nějakého jazyka. Tento argument bude založený na (již poznaném) faktu, že přijímací výpočet dostatečně dlouhého slova se musí v automatu někde "zacyklit", přitom takový cyklus pak můžeme opakovat libovolně krát. Precizním vyjádřením těchto myšlenek je následující tzv. "*Pumping lemma*".

Lema 4.8. *Nechť L je regulární jazyk. Pak nutně existuje n takové, že každé slovo $z \in L$, $|z| \geq n$ lze psát jako zřetězení $z = uvw$, kde $|uv| \leq n$, $|v| \geq 1$ a pro všechna $i \geq 0$ je $uv^i w \in L$.*

`Důkaz`: Nechť L je přijímán deterministickým automatem $\mathcal{A}$ 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. □

Příklad 4.9. *Jazyk $L = \{a^j b^j : j \geq 0\}$ není regulární.*

Pro spor předpokládejme, že L je regulární, a vezměme slovo $a^n b^n = uvw$. Pak podslovo v podle podmínky $|uv| \leq n$ Lematu 4.8 obsahuje jen písmena a. Takže slovo $uv^2 w$ (dvakrát zopakujeme střed v) má více a než b, a tudíž $uv^2 w \notin L$, což je spor. □

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

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

který splňuje podmínku v Pumping lematu a přitom není regulární. (Ověřte si oboje jiným postupem.)

Úlohy k řešení

(4.2.1) *Je regulární jazyk všech těch slov nad abecedou $\{a, b\}$, ve kterých je stejně znaků a jako znaků b?*

4.3 Věta Myhilla–Neroda

Je tedy možné přesně popsat neregulární jazyky nějakým "testem" podobným Lematu 4.8? Kupodivu to možné je a takový popis byl publikován zhruba ve stejné době jako pumping lemma, přesto je méně známý a méně používaný.

`Komentář`: Princip takového rozpoznání neregulárnosti jazyka je postavený na stejné myšlence jako Lema 4.2. Pro demonstraci si vezměme stejný jazyk $L = \{a^j b^j : j \geq 0\}$ a předpokládejme, že je přijímaný automatem $\mathcal{A}$. Slova a^j a $a^{j'}$ pro $j \neq j'$ musí převést $\mathcal{A}$ do různých stavů q, q', neboť $b^j \in L_{\mathcal{A}}(q)$, ale $b^j \notin L_{\mathcal{A}}(q')$. Jelikož však přípustných prefixů a^j je nekonečně mnoho, nemůže mít automat $\mathcal{A}$ jen konečně mnoho stavů.

Takovéto "zdůvodnění" má však v sobě značnou dávku schizofrenie – neexistenci automatu $\mathcal{A}$ dokazujeme použitím jazyků $L_{\mathcal{A}}(q)$ definovaných právě pomocí stavů tohoto neexistujícího automatu. (Toto nelze jen tak obejít formulací důkazu sporem...) Proto pro jazyky "jako" $L_{\mathcal{A}}(q)$ musíme nejprve najít vhodnou definici nezávislou na (neexistujícím) automatu $\mathcal{A}$. Vlastně zde podáme "popis stavů q automatu" daný pouze vlastnostmi jazyka L.

`Definice`: Nechť L je jazyk na abecedou Σ. *Pravou kongruencí indukovanou L* nazýváme relaci ekvivalence $\simeq_L$ definovanou na všech slovech Σ^* předpisem

$$x \simeq_L y \text{ právě když } \forall z \in \Sigma^* : xz \in L \Leftrightarrow yz \in L\,.$$

Intuitivně, pravá kongruence indukovaná jazykem L vyjadřuje "příbuznost" prefixů slov ve smyslu, že se chovají stejně v L vzhledem ke všem možným sufixům.

Věta 4.10. (Myhill, Nerode) *Jazyk L je regulární právě když pravá kongruence indukovaná jazykem L vytváří konečně mnoho tříd ekvivalence.*

Větu zde dokazovat nebudeme, ale všimněme si, ve spojení s Lematem 4.2, že třídy ekvivalence $\simeq_L$ vlastně odpovídají stavům minimálního konečného automatu rozpoznávajícího L. Takže Větu 4.10 lze využít nejen k dokazování neregulárnosti jazyka, ale i při zdůvodňování minimality automatu přímo ze zadání jazyka.

Příklad 4.11. *Jazyk $L = \{a,b\}^* \cup \{c\} \cdot \{c\}^* \cdot \{a^j b^j : j \geq 0\}$ není regulární.*

Zde okamžitě vidíme, že prefixy ca^j pro $j = 1, 2, \ldots$ musí patřit do různých tříd pravé kongruence $\simeq_L$: Platí $ca^j \cdot b^j \in L$, ale $ca^j \cdot b^k \notin L$ pro $k \neq j$. Proto L není regulární z Věty 4.10. □

Úlohy k řešení

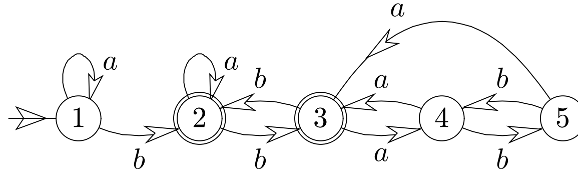
(4.3.1) *Je regulární jazyk všech těch slov nad abecedou $\{a, b\}$, ve kterých je stejně znaků a jako znaků b?*

(4.3.2) *Je regulární jazyk všech těch slov nad abecedou $\{a, b\}$, ve kterých je rozdíl počtů znaků a a znaků b nezáporný? Jak odpověď souvisí s předchozí úlohou?*

(4.3.3) *Je regulární jazyk všech těch slov nad abecedou $\{a, b\}$, ve kterých je rozdíl počtů znaků a a znaků b větší než 100?*

4.4 Cvičení: Minimalizace a neregulárnost

Příklad 4.12. *Minimalizujme následující automat:*



Začneme s rozdkladem 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

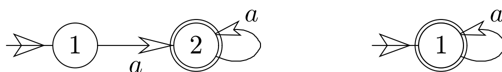
Po první tabulce se nám obě třídy rozpadnou na dvě podtřídy každá. Po druhé iteraci již k dalšímu rozlišení nedojde, a proto skončíme. Vidíme tedy, že v minimálním automatu dojde ke sloučení stavů 4 a 5 do jednoho. □

Příklad 4.13. *Je regulární jazyk všech těch slov nad $\{a, b, c\}$, ve kterých je rozdíl počtů znaků a a znaků b rovný počtu znaků c?*

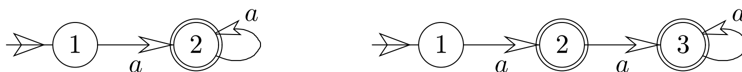
Intuice nám napoví, že i v tomto jazyce L se po nás žádá "spočítat" počty všech jednotlivých znaků, takže jazyk asi regulární nebude. Podívejme se proto na pravou kongruenci $\simeq_L$ podle Věty 4.10: Všechny prefixy $a^i b$ patří do ruzných tříd $\simeq_L$ pro různá $i > 0$, protože jsou navzájem rozlišeny různými sufixy c^{i-1}, kde $a^i b \cdot c^{i-1} \in L$, ale $a^i b \cdot c^j \notin L$ pro $j \neq i - 1$. Proto $\simeq_L$ vytváří nekonečně mnoho tříd ekvivalence. □

Úlohy k řešení

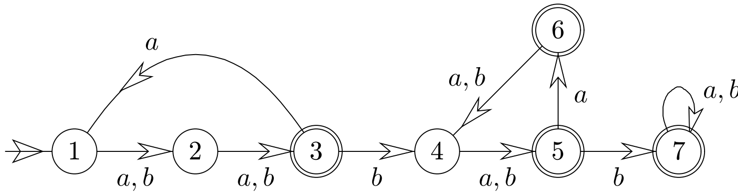
(4.4.1) *Jsou tyto dva automaty nad abecedou $\{a\}$ ekvivalentní?*



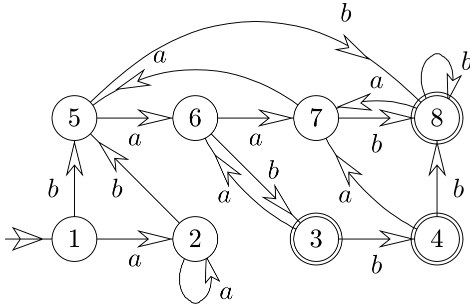
(4.4.2) *Jsou tyto dva automaty nad abecedou $\{a\}$ ekvivalentní?*



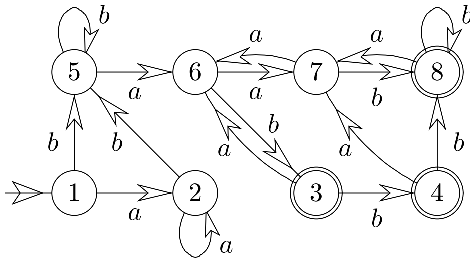
(4.4.3) *Zdůvodněte minimalitu tohoto automatu:*



(4.4.4) *Minimalizujte následující automat:*



(4.4.5) *Minimalizujte následující automat:*



(4.4.6) *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?*

(4.4.7) *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?*

(4.4.8) *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?*

(4.4.9) *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?*

(4.4.10) *Je regulární jazyk všech těch slov nad abecedou $\{a, b\}$, ve kterých je součet počtů znaků a a znaků b větší než 100?*

(4.4.11) *Je regulární jazyk všech těch slov nad abecedou $\{a, b\}$, ve kterých je součin počtů znaků a a znaků b větší než 100?*

***(4.4.12)** *Je regulární jazyk všech těch slov nad abecedou $\{a, b\}$, ve kterých je podíl počtů znaků a a znaků b větší než 100?*

***(4.4.13)** *Je regulární jazyk všech těch slov nad abecedou $\{a\}$, ve kterých je počet znaků a prvočíselný?*

5 Bezkontextové jazyky a ZA

Úvod

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. Na druhou stranu, pomocí v praxi zaužívaného způsobu zápisu, může být množina všech takových aritmetických výrazů popsaná těmito (přepisovacími) pravidly:*

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

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

Cíle

Tato lekce zavádí bezkontextové gramatiky jako vhodný způsob zápisů jazyků, které jsou obecnější než regulární jazyky. Ukáže se příklad odvozování aritmetických výrazů, který pěkně ilustruje použití bezkontextových gramatik. Definují se jednoznačné gramatiky a Chomského normální forma. Stručně se nastíní ekvivalence bezkontextových gramatik se zásobníkovými automaty.

5.1 Odvození aritmetických výrazů

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

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

`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

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

Před čtením dalšího textu je potřebné si uvědomit, že význam odvozovacích pravidel je nejen ve formálním popisu bezkontextového jazyka, ale hlavně v popisu postupu odvození každého konkrétního slova jazyka.

Příklad 5.1. *Co je problematického s našimi pravidly?*

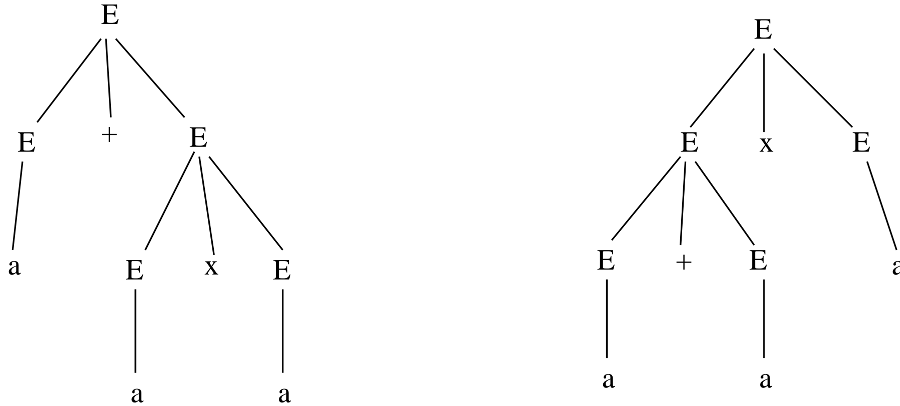
$$E \longrightarrow E + E \mid E \times E \mid (E) \mid a$$

Problematičnost této gramatiky tkví v její *nejednoznačnosti*: Tentýž výraz $a + a \times a$ lze odvodit dvěma zcela rozdílnými způsoby (odlišnými nejen pořadím přepisování symbolů, ale i syntaktickým významem)

$$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E \times E \Rightarrow a + a \times E \Rightarrow a + a \times a\,, \text{ nebo}$$

$$E \Rightarrow E \times E \Rightarrow E + E \times E \Rightarrow a + E \times E \Rightarrow a + a \times E \Rightarrow a + a \times a\,.$$

Schematicky jsou tato odvození znázorněná takto:



Představme si, že bychom podle takového schematického odvození chtěli konkrétní výraz aritmeticky vyhodnotit. Přitom pouze první odvození $a + a \times a$ je v souladu s běžnými aritmetickými pravidly, konkrétně s prioritou násobení před sčítáním. Druhým způsobem podle aritmetických pravidel správně vznikne závorkovaný výraz $(a + a) \times a$.

Proto se musíme (chybného) druhého odvození nějak zbavit – udělat odvození *jednoznačným* vzhledem k aritmetickým konvencím. □

Příklad 5.2. *Zapišme výše uvažované aritmetické výrazy pravidly, které povedou k jednoznačnému aritmetickému vyhodnocení.*

Jednou z možností, jak dosáhnout jednoznačnosti odvození výrazu v souladu s běžnými aritmetickými pravidly, je použití "vynuceného závorkováni" u součinu

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

Takto však na druhou stranu vygenerujeme zbytečně mnoho závorek, jako například $(a + a) \times (a)$. Jinými slovy, generovaný jazyk je menší než v předchozím případě a ne všechny platné aritmetické výrazy jsou jim popsány.

Správného řešení dosáhneme použitím dalšího symbolu F vyšší priority, tj. budeme mít tzv. otevřené E-výrazy a uzavřené (uzávorkované v případě součtu) F-výrazy. Pravidla naší aritmetické gramatiky pak budou znít

$$E \longrightarrow E + E \mid F\,,$$

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

Zkontrolujte si je sami!

Co by se zde ještě dalo vylepšit? Viz Příklad 5.14. □

`Komentář:` Všimněme si ještě blíže dvojice pravidel $E \longrightarrow F$ a $F \longrightarrow (E)$ – zdá se, jakoby první z nich říkalo, že otevřený výraz můžeme prohlásit za uzavřený, a druhé, že uzavřený výraz se uzávorkováním stane otevřeným (???). To je samozřejmě nesmysl. Je třeba si dobře uvědomit, že odvozovací pravidla se mají vnímat "pozpátku", takže $E \longrightarrow F$ říká, že uzavřený F-výraz lze použít na místě otevřeného E, a $F \longrightarrow (E)$ říká, že uzávorkováním otevřeného E-výrazu vznikne uzavřený F-výraz.

Úlohy k řešení

(5.1.1) *Přidejte do aritmetických výrazů operaci umocnění a^2 nejvyšší priority a napište příslušná odvozovací pravidla.*

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

5.2 Bezkontextové gramatiky a jazyky

V předchozím oddíle jsme na příkladech uvedli v podstatě všechny důležité koncepty bezkontextových gramatik, které nyní budeme formalizovat.

Definice 5.3. ***Bezkontextová gramatika*** je čtveřice $G = (\Pi, \Sigma, S, \mathcal{P})$, 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
- $\mathcal{P}$ je konečná množina *pravidel* tvaru $A \to \beta$, kde
  - A je neterminál, tedy $A \in \Pi$,
  - β je řetězec složený z terminálů a neterminálů, tedy $\beta \in (\Pi \cup \Sigma)^*$.

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

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

`Definice`: *Přímé odvození* slova δ ze slova γ podle pravidel gramatiky G, tj. nahrazení jednoho neterminálu v γ podle vhodného pravidla z G, značíme $\gamma \Rightarrow_G \delta$, nebo zkráceně jen $\gamma \Rightarrow \delta$. Říkáme, že γ *lze přepsat* na δ (neboli δ je *odvozeno* z γ), jestliže existuje posloupnost přímých odvození

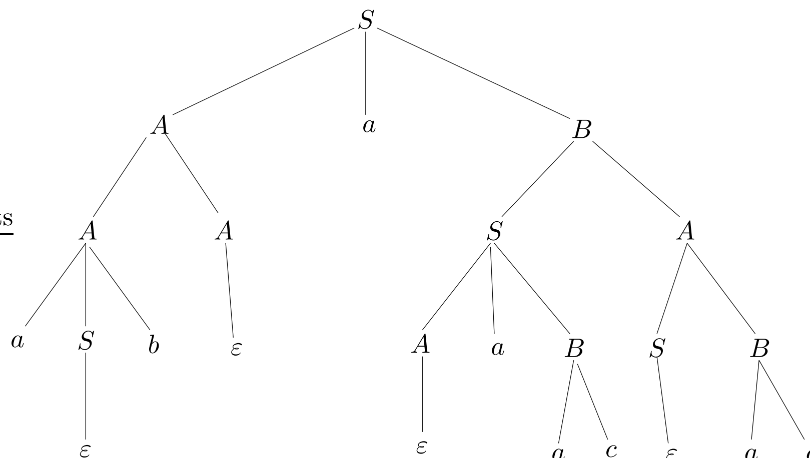
$$\gamma \Rightarrow^* \delta \ : \quad \gamma \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \delta\,.$$

> `Komentář`: Již dříve jsme uváděli postupné odvození aritmetického výrazu $E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E \times E \Rightarrow a + a \times E \Rightarrow a + a \times a$.

`Definice`: *Derivačním stromem* slova w v bezkontextové gramatice G rozumíme uspořádaný ohodnocený kořenový strom, ve kterém je kořen označen S a listy jsou zleva doprava označené jednotlivými písmeny slova w, případně ε. Přitom každý ne-list stromu je označen neterminálem X a jeho synové jsou po řadě značeny písmeny slova β příslušného (vhodného) pravidla $X \to \beta$ gramatiky G.

> `Komentář`: Příklad derivačního stromu používajícího níže zapsaná pravidla.

PSfrag replacements



$$S \longrightarrow AaB \mid \varepsilon,$$

$$A \longrightarrow AA \mid AaB \mid \varepsilon \mid aSb \mid SB, \qquad B \longrightarrow SA \mid ac$$

`Definice`: Gramatika je *jednoznačná*, jestliže pro každé slovo v této gramatice je jediný derivační strom.

`Poznámka`: *Levým odvozením* (derivací) slova v bezkontextové gramatice nazýváme to odvození, ve kterém se v každém kroku přepisuje první neterminál zleva. Obdobně pravým odvozením je to, kdy se přepisuje vždy poslední neterminál. Jednoznačnost gramatiky pak znamená, že každé slovo má jediné levé odvození.

> `Komentář:` Podívejme se opět na odvození aritmetických výrazů – Příklad 5.2...
> Přestože jsme jej navrhli tak, že výsledek vyhodnocení byl aritmeticky jednoznačný, ona gramatika není jednoznačná ve smyslu naší definice! Pro dosažení jednoznačnosti gramatiky ještě musíme zavést pravidlo, že stejné operace se vyhodnocují zleva doprava.

Definice 5.4. ***Jazyk generovaný gramatikou*** G, značený $L(G)$,
je množinou všech slov $L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$. Všimněte si dobře, že do jazyka patří pouze ta slova odvozená z S, která jsou složená jen z terminálů.

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

> `Komentář:` Použití bezkontextových jazyků a gramatik je důležité v překladačích.

Příklad 5.5. *Navrhněte gramatiku generující jazyk palindromů $L_1 = \{ww^R : w \in \{a,b\}^*\}$, kde w^R značí slovo w zapsané pozpátku.*

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

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

□

Příklad 5.6. *Co nejvýstižněji slovně popište jazyk generovaný gramatikou*

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

První pravidlo $S \longrightarrow SbS$ vytváří jenom a pouze střídavé řetězce '$SbSb\dots SbS$'. 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 '$SbSb\dots SbS$' dosazením a, tj. generován je jazyk všech slov tvaru '$abab\dots aba$'. □

Speciální formy gramatik

Dále si uvedeme jen velmi stručný přehled některých forem gramatik a jejich převodů. Blíže viz doplňková literatura...

`Definice`: Gramatiky jsou *ekvivalentní*, pokud generují stejný jazyk.

`Definice`: Bezkontextová gramatika je *redukovaná*, jestliže každý neterminál generuje nějaké slovo a každý neterminál je dosažitelný z počátečního neterminálu.

Věta 5.7. *Ke každé bezkontextové gramatice G tž. $L(G) \neq \emptyset$ lze sestrojit ekvivalentní redukovanou gramatiku.*

`Definice`: Bezkontextová gramatika se nazývá nevypouštějící, jestliže neobsahuje žádné pravidlo tvaru $X \longrightarrow \varepsilon$. Bezkontextová gramatika je v Chomského normální formě, zkráceně v CHNF, jestliže každé její pravidlo je tvaru $X \to YZ$ nebo $X \to a$, kde X, Y, Z označují neterminální symboly a a je terminální symbol.

Věta 5.8. *Ke každé bezkontextové gramatice G lze sestrojit (téměř ekvivalentní) gramatiku G' v Chomského normální formě tž. $L(G') = L(G) - \{\varepsilon\}$.*

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

Úlohy k řešení

(5.2.1) *Generuje gramatika z Příkladu 5.5 skutečně všechny palindromy nad $\{a, b\}$?*

(5.2.2) *Sestrojte gramatiku generující jazyk $\{0^n 1^m 0^n : m, n \geq 0\}$.*

(5.2.3) *Jaký jazyk generuje gramatika?*

$$\begin{aligned} S &\longrightarrow aBC \mid aCa \mid bBCa \\ B &\longrightarrow bBa \mid bab \mid SS \\ C &\longrightarrow BS \mid aCaa \mid bSSc \end{aligned}$$

(5.2.4) *Generuje gramatika*

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

stejný jazyk jako gramatika

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

5.3 Zásobníkové automaty

S bezkontextovými gramatikami se přirozeně pojí následující zobecnění automatu.

> `Komentář`: Víme, že např. jazyk $L = \{a^n b^n : n \geq 1\}$ 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 prostě 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.

Definice 5.9. ***Zásobníkový automat*** (zkráceně ZA) $\mathcal{M}$
je dán parametry $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$, kde

- Q je konečná neprázdná množina *stavů*,
- Σ je konečná neprázdná množina *vstupních symbolů*,
- Γ je konečná neprázdná množina *zásobníkových symbolů*,
- $q_0 \in Q$ 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 $Q \times \Gamma^*$.

Všimněte si, že zásobníkový automat je definován jako nedeterministický.

`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`: Slovo w je přijímáno (nedeterministickým) zásobníkovým automatem $\mathcal{M}$, pokud existuje větev výpočtu $\mathcal{M}$ nad slovem w, která skončí na konci slova w s prázdným zásobníkem.

Věta 5.10. *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.

Příklad 5.11. *Jazyk $a^i b^i c^i$ není bezkontextový, tj. nelze jej rozpoznat zásobníkovým automatem.*

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. Přesněji viz [8]. □

Vlastnosti bezkontextových jazyků

Opět v tomto oddíle uvedeme jen velmi stručný přehled některých vlastností bezkontextových jazyků. Zájemce o hlubší studium odkazujeme například na [8].

Věta 5.12. *Bezkontextové jazyky jsou uzavřené vůči sjednocení, zřetězení, iteraci a zrcadlovému obrazu. Navíc jsou uzavřeny vůči průniku s regulárním jazykem.*

Tvrzení 5.13. *Bezkontextové jazyky NEjsou uzavřené vůči průniku ani doplňku.*

`Důkaz`: Snadno najdeme bezkontextové odvození pro jazyky tvaru $a^i b^i c^k$ i $a^i b^k c^k$. Jejich průnikem však je jazyk tvaru $a^i b^i c^i$, který není bezkontextový podle Příkladu 5.11. Zároveň je tak dokázaná neuzavřenost vůči doplňku, neboť jinak bychom pomocí doplňku a sjednocení určili i průnik jako bezkontextový. □

`Poznámka`: V literatuře lze najít i *deterministické ZA*, které jsou však striktně slabší než nedeterministické. (Neboli žádný obecný převod nedeterministického ZA na deterministický na rozdíl od klasických automatů neexistuje.)

Úlohy k řešení

(5.3.1) *Proč není bezkontextový jazyk L_3 všech těch slov nad abecedou $\{a, b, c\}$ obsahujících stejně výskytů od každého znaku a, b, c?*

Rozšiřující studium

Problematika gramatik je v našem textu probrána jen velice velice stručně, a to kvůli nedostatku času. Čtenář najde mnohem obsáhlejší teoretické informace například v textu [8] nebo v knize [6]. S problematikou bezkontextových gramatik a jejich praktického použití se také zajisté podrobně setká ve studijních předmětech o překladačích.

5.4 Cvičení: Konstrukce bezkontextových gramatik

Příklad 5.14. *Definují už známá pravidla aritmetických výrazů jednoznačnou gramatiku?*

$$E \longrightarrow E+E \mid F$$
$$F \longrightarrow (E) \mid F\times F \mid a$$

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

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

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. □

Příklad 5.15. *Přidejte do aritmetických výrazů z Příkladu 5.14 operaci rozdílu, opět s vlastností jednoznačného vyhodnocení vzhledem k aritmetickým pravidlům.*

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

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

□

Příklad 5.16. *Sestrojte bezkontextovou gramatiku generující všechna slova nad abecedou $\{a,b\}$ mající stejně výskytů symbolů a jako b.*

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 vyvářejí 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 $S \longrightarrow SaSbS \mid SbSaS$. □

`Poznámka`Přechozí příklad má také jiná řešení, zkuste některé další najít sami…

Příklad 5.17. *Je gramatika* $S \longrightarrow SaSbS \mid SbSaS \mid \varepsilon$ *jednoznačná?*

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

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

□

Příklad 5.18. *Generuje gramatika*

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

stejný jazyk jako gramatika

$$S \longrightarrow aSb \mid ab \ \ ?$$

Druhá gramatika zřejmě generuje jazyk $\{a^ib^i : i \geq 1\}$. 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^jSb^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^ib^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 $\{a, b\}$. □

Úlohy k řešení

(5.4.1) *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 : $\{\, a^ib^ja : i, j \in N \,\}$

C : $\{\, a^ib^ja^k : i, j, k \in N,\ i + j = k \,\}$

*__(5.4.2)__ *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 : $\{\, a^ib^j : i, j \in N, i < j \,\}$

C : $\{\, a^i : i$ *je prvočíslo* $\}$

(5.4.3) *Jak byste napsali gramatiku k jazyku* $\{\, a^ib^j : i, j \in N, i < j \,\}$*?*

(5.4.4) *Zapište odvozovacími pravidly bezkontextové gramatiky jazyk všech těch palindromů nad abecedou* $\{a, b\}$*, jejichž délka je násobkem čtyř.*

*__(5.4.5)__ *Zapište odvozovacími pravidly bezkontextové gramatiky jazyk všech těch palindromů nad abecedou* $\{a, b\}$*, jejichž délka je násobkem tří.*

(5.4.6) *Generuje gramatika*

$$S \longrightarrow aaSbb \mid ab \mid \varepsilon$$

stejný jazyk jako gramatika

$$S \longrightarrow aSb \mid ab \ \ ?$$

(5.4.7) *Generuje gramatika*

$$S \longrightarrow aaSb \mid ab \mid \varepsilon$$

stejný jazyk jako gramatika

$$S \longrightarrow aSb \mid aab \mid \varepsilon \ \ ?$$

(5.4.8) *Rozhodněte, která z následujících dvou gramatik generuje regulární jazyk, tj. přijímaný také konečným automatem.*

A : $\quad S \longrightarrow aSb \mid bSa \mid \varepsilon$

B : $\quad S \longrightarrow abS \mid baS \mid \varepsilon$

(5.4.9) *Rozhodněte, která z následujících dvou gramatik generuje regulární jazyk, tj. přijímaný také konečným automatem.*

A : $\quad S \longrightarrow ASa \mid \varepsilon \,; \quad A \longrightarrow b$

B : $\quad S \longrightarrow BSa \mid \varepsilon \,; \quad B \longrightarrow a$

(5.4.10) *Rozhodněte, která z následujících dvou gramatik generuje regulární jazyk, tj. přijímaný také konečným automatem.*

A : $\quad S \longrightarrow aSb \mid bSa \mid bbS$

B : $\quad S \longrightarrow ab \mid ba \mid bbS$

****(5.4.11)** *Uměli byste nalézt jednoznačnou gramatiku pro řešení Příkladu 5.16?*

***(5.4.12)** *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.*

Část II
Složitost Algoritmů

6 Matematické základy složitosti

Úvod

V druhé polovině předmětu se budeme věnovat základům teorie algoritmické složitosti. Položme si nejprve otázku: Co je to vlastně "algoritmus"?
Jistě jste již algoritmy řešící různé problémy tvořili i programovali na počítači. Zamýšleli jste se však nad tím, co ono slovo "algoritmus" představuje? A co to vůbec je ten "problém"? Tyto základní pojmy budou formálně vysvětleny v této a následující lekci.

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. Proto náš výklad začneme pohledem na rychlost nárůstu běžných funkcí.

Cíle

V této lekci nejprve zavedeme symboliku a terminologii pro srovnávání rychlosti růstu funkcí a pak si uvedeme vzorce pro řešení některých rekurentních vztahů, které se často objevují při analýze výpočetního času algoritmů. Nakonec si formálně definujeme, co je to (algoritmický) problém a jeho řešení.

6.1 Asymptotické značení funkcí

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.

`Komentář`: 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 porovnávání rychlostí růstů funkcí nám slouží následující terminologie.

`Definice`: Nechť $g : \mathbb{N} \to \mathbb{N}$ je daná funkce. Pro funkci $f : \mathbb{N} \to \mathbb{N}$ píšeme

$$f \in O(g)$$

pokud existují konstanty $A, B > 0$ takové, že

$$\forall n \in \mathbb{N} : f(n) \leq A \cdot g(n) + B\,.$$

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: 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 6.1. *Příklady růstů různých funkcí.*

Funkce $\boldsymbol{f(n) = \Theta(n)}$: 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 $\boldsymbol{f(n) = \Theta(n^2)}$: 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 $\boldsymbol{f(n) = \Theta(2^n)}$: 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 $\boldsymbol{999999n^2}$ 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! □

Úlohy k řešení

(6.1.1) *Která z těchto funkcí roste nejrychleji ?*

a) $1000n$ *b)* $n \cdot \log n$ *c)* $n \cdot \sqrt{n}$

(6.1.2) *Udejte správný asymptotický vztah mezi funkcemi* $\sqrt{n}$ a $\log n$.

***(6.1.3)** *Roste rychleji funkce* n^5 *nebo* $(\log n)^{\log n}$*?*

***(6.1.4)** *Která z těchto funkcí roste nejrychleji ?*

a) $n!$ *b)* $(\log n)^n$ *c)* $(\sqrt{n})^n$

***(6.1.5)** *Dokažte, že pro všechna* $c > 0$ *a přirozená* k *platí:*

$$\log^k n \in O(n^c)$$

6.2 Rekurentní vztahy

V tomto oddíle si uvedeme krátký přehled některých rekurentních vzorců, se kterými se můžete setkat při řešení časové složitosti (převážně rekurzivních) algoritmů.

Lema 6.2. *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

$$T(n_1) \leq T(\lceil a_1 n_1 \rceil) + \ldots + T(\lceil a_k n_1 \rceil) + cn_1 \leq$$
$$\leq d \cdot \lceil a_1 n_1 \rceil + \ldots + d \cdot \lceil a_k n_1 \rceil + cn_1 \leq$$
$$\leq d \cdot (1 - \varepsilon)n_1 + cn_1 \leq dn_1 - (\varepsilon d - c)n_1 \leq dn_1\,.$$

□

Lema 6.3. *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:

Lema 6.4. *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.

Příklad 6.5. *Algoritmus merge-sort pro třídění čísel pracuje zhruba následovně:*

- *Danou posloupnost* n *čísel rozdělí na dvě (skoro) poloviny.*
- *Každou polovinu setřídí zvlášť za použití rekurentní aplikace merge-sort.*
- *Tyto dvě už setříděné poloviny “slije” (anglicky merge) do jedné setříděné výsledné posloupnosti.*

Jaký je celkový počet jeho kroků?

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 Lematu 6.3 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 Lematu 6.4 pro $a = b = 2$.) □

Úlohy k řešení

(6.2.1) *Odhadněte asymptoticky výsledek rekurence* $T(n) \leq T(n/2) + T(n/5) + 2n$.

(6.2.2) *Odhadněte asymptoticky výsledek rekurence* $T(n) \leq T(n/2) + T(n/3) + T(n/6) + 2n$.

(6.2.3) *Odhadněte asymptoticky výsledek rekurence* $T(n) \leq 4T(n/3) + 2n^2$.

(6.2.4) *Odhadněte asymptoticky výsledek rekurence* $T(n) \leq 4T(n/2) + 2n^2$.

6.3 Kódování slov a problémů

Abychom mohli jednotně formálně zavést pojmy problému a později i jeho řešení algoritmem, potřebujeme nejprve ukázat, že vlastně vůbec nezáleží na tom, jakou (konečnou) abecedu používáme pro zápisy.

Lema 6.6. *Pro každou konečnou abecedu Σ existuje jednoznačné kódování všech slov z Σ^* přirozenými čísly, neboli dvojice zobrazení*

$$k : \Sigma^* \to \mathbb{N} \quad a \quad d : \mathbb{N} \to \Sigma^*$$

(jako "kódování" a "dekódování") takových, že složené zobrazení $d \circ k(w) = d(k(w))$ je identitou na množině slov Σ^* (tj. slova lze zpětně dekódovat).
Funkce k, d navíc lze velmi snadno a rychle vypočítat.

Důkaz: Nechť $z = |\Sigma| + 1$. Znaky abecedy Σ jednoznačně "očíslujeme" přirozenými čísly $c : \Sigma \leftrightarrow \{1, 2, \ldots, z - 1\}$. Slovo $w = x_1x_2 \ldots x_n$, kde $x_i \in \Sigma$ jsou jednotlivé znaky, zakódujeme číslem $k(w) = (c(x_1)c(x_2) \ldots c(x_n))_z$ zapsaným v soustavě o základu z.

Naopak číslo $a = (a_1a_2 \ldots a_n)_z$ zapsané v soustavě o základu z převedeme na slovo $d(a)$ tak, že číslice $a_i \neq 0$ po řadě přepíšeme na příslušné znaky $c^{-1}(a_i)$ a číslice $a_j = 0$ vypustíme (jako ε prázdné slova). □

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.

Komentář: Obvykle je zadání problému dáno nějakým řetězcem znaků nebo bytů, takže dle předchozího lematu jej lze zakódovat jednoduše přirozeným číslem (obvykle obrovským, ale to nás nezajímá, neboť je to jen teoretická abstrakce). Výstup algoritmu také bývá obvykle ve formě řetězce znaků nebo bytů a může být obdobně zakódován. Z pohledu teorie mají však speciální důležitost takové formy problémů, ve kterých je odpověď jednobitová (rozhodovací) – buď ANO nebo NE. Jak uvidíme, každý obecný problém lze rozložit na rozhodovací.

Definice: *Problém* je libovolné zobrazení (funkce) na množině všech slov

$$P : \Sigma^* \to \Sigma^*,$$

kde Σ je konečná (předem dohodnutá) abeceda.
Slovo $w \in \Sigma^*$ nazýváme *vstupem* problému a výslednou hodnotu $P(w)$ nazýváme *výstupem*.

Pro odlišení od příští definice se takovému problému říká také "výpočetní problém". Ve spojení s číselným kódováním slov v Lematu 6.6 tak lze říci:

Fakt: Výpočetní problém je určený libovolnou funkcí $p : \mathbb{N} \to \mathbb{N}$.

Definice: *Rozhodovací problém* je problém ve tvaru $P : \Sigma^* \to \{0, 1\}$, neboli $P : \Sigma^* \to \{NE, ANO\}$, kde Σ je konečná (předem dohodnutá) abeceda.

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. (Viz také Oddíl 9.1.)

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.

Příklad 6.7. *Ukažme si, jak problém výpočtu funkce* $\sin x$ *lze rozložit na posloupnost rozhodovacích problémů.*

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.) □

6.4 Cvičení: Odhady funkcí a rekurentních vztahů

Příklad 6.8. *Odhadněte asymptoticky výsledek rekurence*

$$T(n) = T(n/2) + T(n/3) + T(n/4) + 5n^2\,.$$

Tento vztah sice na první pohled nepatří do žádného z uvedených lemat, ale lze jej velice snadno upravit:

$$T(n) \le T(n/2) + T(n/3) + T(n/4) + 5n^2 \le 3T(n/2) + 5n^2$$

Poslední vztah již podle Lematu 6.4 má řešení $T(n) = O(n^2)$ neboť $\log_2 3 < 2$. Na druhou stranu hned ze zadaného vztahu vidíme, že $T(n) \ge 5n^2$, takže výsledné řešení skutečně je $T(n) = \Theta(n^2)$. □

Úlohy k řešení

(6.4.1) *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 : $\sqrt{n} \in O(n)$

(6.4.2) *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 : $2^n \in O(n^{1024})$

(6.4.3) *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 : $n^{\log n} \in O(n^{1024})$

(6.4.4) *Seřaďte následující tři funkce podle asymtpotické rychlosti jejich růstu od nejpomalejšího růstu.*

A : $n + \sqrt{n} \cdot \log n$

B : $n \cdot \log n$

C : $\sqrt{n} \cdot \log^2 n$

(6.4.5) *Seřaďte následující tři funkce podle asymtpotické rychlosti jejich růstu od nejpomalejšího růstu.*

A : 2^n

B : $2^{\sqrt{n}}$

C : $n!$

(6.4.6) *Seřaďte následující tři funkce podle asymtpotické rychlosti jejich růstu od nejpomalejšího růstu.*

A : $n/2005$

B : $\sqrt{n} \cdot 3n$

C : $n + n \cdot \log n$

(6.4.7) *Seřaďte následující tři funkce podle asymtpotické rychlosti jejich růstu od nejpomalejšího růstu.*

$A:$ $(\log n)^n$

$B:$ n^n

$C:$ $2^{\sqrt{n}}$

(6.4.8) *Nezáporná funkce $p(n)$ splňuje pro všechna přirozená n následující rekurentní vztah*

$$p(n) \le 3 \cdot p(n/5) + n\,.$$

Jaký je (nejlepší) asymptotický odhad růstu funkce $p(n)$ v závislosti na n?

(6.4.9) *Nezáporná funkce $p(n)$ splňuje pro všechna přirozená n následující rekurentní vztah*

$$p(n) \le 4 \cdot p(n/2) + n\,.$$

Jaký je (nejlepší) asymptotický odhad růstu funkce $p(n)$ v závislosti na n?

(6.4.10) *Nezáporná funkce $p(n)$ splňuje pro všechna přirozená n následující rekurentní vztah*

$$p(n) \le p(n/3) + p(n/4) + p(n/5) + n\,.$$

Jaký je (nejlepší) asymptotický odhad růstu funkce $p(n)$ v závislosti na n?

(6.4.11) *Nezáporná funkce $p(n)$ splňuje pro všechna přirozená n následující rekurentní vztah*

$$p(n) \le p(n/2) + p(n/3) + p(n/6) + n\,.$$

Jaký je (nejlepší) asymptotický odhad růstu funkce $p(n)$ v závislosti na n?

7 Co je to algoritmus

Úvod

Nyní se dostáváme k oné fundamentální teoretické otázce; co je vlastně "algoritmus"? Pohledy na algoritmy se vyvíjejí už od dávné minulosti (vzpomeňme třeba netriviální Euklidův algoritmus největšího společného dělitele). 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.

Nejprve si zavedeme tradiční teoretický model výpočtu (algoritmu), Turingův stroj, který je podobný konečnému automatu. Rozdíl je v tom, že páska, na níž je na začátku zapsáno vstupní slovo, 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.

Poté uvedeme výpočetní model RAM, který je svou podobou mnohem bližší skutečným počítačům. Ukážeme, že tyto dva modely jsou výpočetně ekvivalentní. V souvislosti s tím popíšeme tzv. Church–Turingovu tezi, která je všeobecně přijímána jako axiomatická "definice" algoritmu. Nakonec si ukážeme, že ne všechny problémy lze algoritmicky řešit.

Cíle

V této lekci definujeme Turingův stroj a počítač RAM jako základní teoretické modely algoritmických výpočtů. Zároveň si pomocí Church–Turingovy teze zavedeme přesně pojem algoritmu. Na závěr si ukážeme, že existují problémy, které nelze řešit žádným algoritmem.

7.1 Turingův stroj

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 velmi jednoduchý teoretický model počítače, který vychází z konečného automatu, ale rozšiřuje jeho možnosti o zápis nových symbolů na neomezenou pásku a o možnost pohybu hlavy po této pásce v obou směrech.

Definice 7.1. ***Turingův stroj*** $\mathcal{M}$**,** zkráceně TS,
je určen šesticí parametrů $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, F)$, 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.

`Poznámka`: Předpokládáme, že v $\Gamma \setminus \Sigma$ je vždy obsažen speciální prvek $\Box$ – *prázdný znak*, neboli také mezera.

`Komentář:` Někdy se vstupní slovo dané TS na počátku výpočtu ještě dodatečně ohraničuje zleva i zprava dalšími speciálními znaky, třeba '#*slovo*#', ale my budeme obvykle předpokládat, že vstupní slovo je od levého prázdného znaku po první pravý prázdný znak.

Definice 7.2. ***Výpočet TS*** $\mathcal{M}$ probíhá v následujících krocích:

- Na počátku je vstupní slovo zapsáno na pásce a zbytek je vyplněn $\Box$. TS začíná s hlavou na dohodnutém znaku slova (obvykle na prvním zleva).
- Hodnota přechodové funkce $\delta(q, x) = (q', y, m)$ znamená, že při čtení symbolu x z pásky ve stavu q dojde k přechodu do stavu q', zápisu symbolu y na pásku (místo x) a k posunu hlavy o $m = \pm 1, 0$ políček.
- Při přechodu TS do koncového stavu výpočet končí.

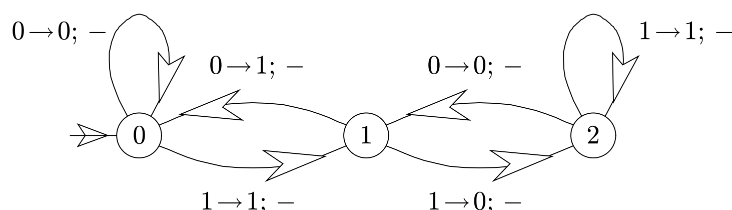
`Komentář:` V základní definici je Turingův stroj deterministický, ale i při povolení nedeterminismu získáme jen stejnou výpočetní sílu. Význam posunu hlavy o ± 1 nebo 0 políček je tento; při $+1$ se hlava posune o políčko doprava, při -1 o políčko doleva a při 0 zůstane na místě.

Uvědomme si dobře, že výpočet Turingova stroje nemusí vždy skončit, ale i uváznout v nekonečné smyčce. Proto pro něj musí být zavedeny koncové stavy, ve kterých se výpočet vždy zastaví. (Takže koncové stavy se chovají jinak než přijímající stavy automatu!)

Příklad 7.3. *Sestrojme 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.*

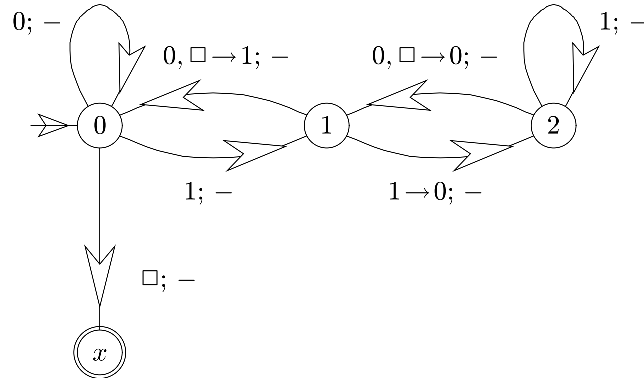
Zde si vzpomeneme na primitivní školní algoritmus násobení: Pokud je poslední číslice 0, v trojnásobku bude také 0 a žádný přenos. Pokud je poslední číslice 1, v trojnásobku bude také 1 a navíc přenos 1. Naopak s přenosem 1 se 0 změní na 1 a žádný přenos, 1 se změní 0 a přenos 2. Obdobně pokračujeme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



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 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:



□

`Značení`: Jak jsme viděli v předchozích obrázcích, stavové přechody TS jako $\delta(q,x) = \delta(q',y,m)$ značíme zkratkou $x \to y;\, +$ nebo $x \to y;\, 0$ nebo $x \to y;\, -$ podle směru pohybu hlavy po přečtení, to vše zapsáno u šipky z q do q'. Pokud znak na pásce nechceme přepsat, zkráceně píšeme jen $x;\, +$ a můžeme uvádět i více čtených znaků najednou.

`Definice`: Jazyk *přijímaný Turingovým strojem* $\mathcal{M}$, značený $L(\mathcal{M})$, je množinou všech slov nad Σ, pro něž výpočet M nakonec dojde do koncového stavu. Jazyk $L \subseteq \Sigma^*$ je *rekurzivně spočetný*, pokud je přijímaný nějakým Turingovým strojem.

> `Komentář`: Jestliže výpočet TS neskončí (nedojde do koncového stavu, nezastaví se), je jeho výsledek nedefinovaný. Pokud bychom však vyžadovali, aby platné Turingovy stroje vždy ukončili svůj výpočet (a měli třeba koncové stavy rozdělené na přijímací a odmítací), třída takto rozpoznávaných jazyků (jsou nazývány *rekurzivní*) je striktně menší. To je částečně podchyceno následující definicí – rekurzivní jazyky jsou ty, které jsou jako rozhodovací problém řešeny Turingovými stroji.

`Definice`: Turingův stroj $\mathcal{M}$ *počítá* (částečnou) funkci $P_{\mathcal{M}} : L(\mathcal{M}) \to \Gamma^*$, kde $P_{\mathcal{M}}(w)$ je slovo bez mezer, které zůstane na pracovní pásce TS $\mathcal{M}$ po jeho zastavení na vstupním slově w.

`Definice`: Nechť $P : \Sigma^* \to \Sigma^*$ je problém. Říkáme, že Turingův stroj $\mathcal{M}$ *řeší problém* P, jestliže $L(\mathcal{M}) = \Sigma^*$ (výpočet vždy skončí) a $P_{\mathcal{M}} \equiv P$.

Někdy je výhodné využívat následující zobecnění TS: *Vícepáskovým Turingovým strojem* míníme model, který je definován obdobně jako Turingův stroj, ale má pevně daný počet pásek (větší než 1) se samostatně řízenými hlavami. Přechodová funkce pak bere ohled na symboly čtené hlavami ze všech pásek a určuje pohyb každé hlavy (na její pásce) zvlášť.

Lema 7.4. *Pro každé $k > 1$ lze výpočet k-páskového Turingova stroje emulovat jednopáskovým Turingovým strojem, který vykoná nejvýše kvadratický počet kroků vzhledem k emulovanému stroji.*

`Důkaz`: k pásek původního stroje $\mathcal{M}_k$ emulujeme na jedné pásce $\mathcal{M}_1$ tak, že rozdělíme pásku na $2k$-tice políček, kde vždy v i-té skupině se budou nacházet symboly z i-tých políček všech emulovaných pásek a speciální znaky značící, zda se emulované

hlavy nacházejí na i-té pozici. Jeden krok $\mathcal{M}_k$ pak bude emulován následovně:
Stroj $\mathcal{M}_1$ projde celou obsazenou část $2k$-tic své pásky a zapamatuje si při tom symboly všech k emulovaných pásek. Pak emuluje přechod stroje $\mathcal{M}_k$ a opět projde celou obsazenou část své pásky, přičemž zapíše všech k symbolů emulovaných pásek a zaznačí posuny emulovaných hlav. Jelikož obsazená část paměti emulovaného stroje $\mathcal{M}_k$ je velikosti $O(t)$, kde t je počet kroků $\mathcal{M}_k$, jeden krok $\mathcal{M}_k$ zabere $O(t)$ kroků $\mathcal{M}_1$. Celkem tak $\mathcal{M}_1$ vykoná $O(t^2)$ kroků (pokud vůbec skončí). □

Úlohy k řešení

(7.1.1) *Navrhněte TS, který zadané slovo nad abecedou $\{0, 1\}$ invertuje, tj. nuly přepíše na jedničky a naopak.*

(7.1.2) *Jak byste upravili TS z Příkladu 7.3, aby začínal výpočet na prvním znaku vstupu (zleva)?*

(7.1.3) *Navrhněte TS, který číslo zadané ternárně nad abecedou $\{0, 1, 2\}$ vynásobí dvěma.*

7.2 RAM model počítače

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.)

Definice 7.5. ***RAM (Random Access Machine),***
neboli *počítač s libovolným přístupem*, se skládá z těchto částí:

- omezená řídící jednotka s (několika) registry,
- R/O programová paměť s ukazatelem instrukce,
- neomezená pracovní R/W lineární paměť s libovolným přístupem,
- oddělené R/O vstupní paměť a W/O výstupní paměť.

Přitom řídící jednotka může vykonávat následující *elementární operace dané instrukcemi* v programové paměti (dle ukazatele):

- Přesunovat data mezi registrem a libovolným (indexovaným) polem pracovní paměti, ze vstupní nebo do výstupní paměti.
- Zpracovat základní logické operace a podmínky, vykonat skok v programu.
- Vykonat základní aritmetické operace $(+ - \times /)$ v registrech.
- Zastavit běh programu.

Všimněme si, že v definici není žádné explicitní omezení na velikost čísel uložených v paměti, ale předpokládá se, že všechna použitá čísla jsou “rozumně velká” vzhledem k charakteru a rozsahu řešeného problému.

`Komentář:` Z formálních důvodů je celá paměť modelu RAM rozdělena na vstupní, ze které lze pouze číst, na výstupní, do které lze pouze zapisovat, a nakonec na tu pracovní, ve které stroj může dělat, co chce. Jinak si vstupní a výstupní paměti lze představit jako streamy, tj. se sekvenčním přístupem. Ať už se na ně díváme jakkoliv, rozdíly jsou pouze formální a neovlivňují výpočetní schopnosti stroje RAM. Dále je zde programová paměť, do které vůbec nelze zapisovat a která je uvažovaná zcela odděleně od předchozích pamětí.

`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.

Ukážeme si, jak vlastně stroj RAM řeší zadané problémy.

`Definice`: Nechť $P : \Sigma^* \to \Sigma^*$ je problém. Říkáme, že počítač RAM s programem $\mathcal{A}$ *řeší problém* P, jestliže se výpočet $\mathcal{A}$ na všech vstupech vždy zastaví a přitom při zadání slova $w \in \Sigma^*$ ve vstupní paměti RAM (w zakódovaného přirozenými čísly po jednotlivých znacích) bude po zastavení ve výstupní paměti uvedeno slovo $P(w)$.

> `Komentář:` Všimněme si dobře, že vstup problému P se stroji RAM zadává po jednotlivých znacích (jeden v každém poli paměti). Tato konvence nabude veliké důležitosti v dalších lekcích, které pojednávají o délce výpočtu algoritmu pro daný vstup.

Věta 7.6. *Model počítače RAM je výpočetně ekvivalentní Turingovu stroji.*
Konkrétně každý výpočet Turingova stroje lze simulovat ve zhruba stejném počtu kroků vhodným programem na RAM. Naopak každý výpočet RAM s programem $\mathcal{A}$ lze simulovat výpočtem jistého Turingova stroje $\mathcal{M}$, který získá (vhodně zapsaný) program $\mathcal{A}$ na vstupu pásky. Počet kroků vykonaných $\mathcal{M}$ při simulaci je polynomiální oproti počtu kroků RAM a $\mathcal{A}$ (pokud výpočet skončí).

`Důkaz` (náznak): Simulace výpočtu Turingova stroje na RAM je poměrně triviální. V opačném směru naznačíme, jak lze simulovat RAM na čtyřpáskovém Turingově stroji, a poté se odvoláme na Lema 7.4. Nechť výpočet RAM trvá t kroků.

- Nejprve program $\mathcal{A}$ upravíme tak, aby nezapisoval na adresy paměti větší než $O(t)$: Místo zápisu x do buňky $i \gg t$ zapíšeme do vyhrazené tabulky v paměti dvojici $\langle i, x \rangle$. Místo čtení z těchto buněk čteme z tabulky.
- Poté program $\mathcal{A}$ přepíšeme, aby používal jen omezeně velké hodnoty čísel v každé buňce – emulujeme "aritmetiku dlouhých čísel". (Mimo jiné i pro indexaci paměti budeme používat "dlouhá čísla".)
- První páska $\mathcal{M}$ bude emulovat vstupní paměť RAM, druhá páska výstupní paměť, třetí páska bude obsahovat zakódovaný program s ukazatelem instrukcí a registry RAM a čtvrtá páska obsah pracovní paměti. (Jelikož rozsah čísel v RAM je nyní omezený, stačí jeden symbol na jedno pole paměti.)
- Vzhledem k omezenosti rozsahu použitých čísel lze podmínky i aritmetické operace vyhodnocovat přímo přechodovou funkcí $\mathcal{M}$.
- Jediným technickým problémem je indexace přístupu do pracovní paměti – na čtvrtou pásku, neboť indexy jsou v obecnosti neomezeně velké. (Řešení leží v použití jakoby "stromového" přístupu k indexaci.) Přenos mezi pamětí a registry je již pak snadný.

Simulace jednoho kroku výpočtu je úměrná velikosti užité paměti, tj. $O(t)$. $\square$

7.3 Univerzální algoritmus a neřešitelnost

Nejprve si řekneme, co je to algoritmus. Každý informatik asi dobře vnímá význam slova algoritmus jako posloupnost elementárních kroků výpočtu, ale co jsou to elementární kroky?

Následující axiom je obecně přijímán jako "definice" algoritmu:

Church-Turingova teze.

Ke každému algoritmu je možné zkonstruovat s ním ekvivalentní Turingův stroj (při vhodném vyjádření vstupů a výstupů jako řetězců v určité abecedě). Ekvivalencí zde rozumíme podmínku, že algoritmus i Turingův stroj se zastaví právě pro tytéž vstupy, přičemž pro tytéž vstupy budou příslušné výstupy totožné.

Jinými slovy, za elementární kroky považujeme kroky vykonané jedním přechodem Turingova stroje.

Důsledek 7.7. *Existuje tzv. univerzální algoritmus $\mathcal{U}(k,x)$, při vhodném očíslování všech algoritmů přirozenými čísly: Běh $\mathcal{U}$ na vstupech přirozeném k a slově x simuluje průběh k-tého algoritmu na vstupu x.*

`Důkaz` (náznak): Tento důsledek vyplývá přímo z Church-Turingovy teze za použití simulace z důkazu Věty 7.6 (kde číslo k kóduje podle Lematu 6.6 zápis k-tého algoritmu v modelu RAM). □

Algoritmická neřešitelnost

Jiným důsledkem Church-Turingovy teze je následující definice.

`Definice`: Problém $P : \Sigma^* \to \Sigma^*$ nazveme *algoritmicky řešitelným* pokud existuje Turingův stroj řešící P. (Ekvivalentně, pokud existuje program RAM řešící P.)

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. Hezké, že? 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?

`Definice`: *Problémem zastavení* nazveme problém určit, zda se daný Turingův stroj na daném vstupu zastaví nebo ne.

Věta 7.8. *Problém zastavení je algoritmicky neřešitelný.*

`Důkaz`: Tato věta je jen dalším důsledkem Důsledku 7.7. Pro spor předpokládejme, že nějaký algoritmus $\mathcal{A}$ řeší problém zastavení ($\mathcal{A}$ vždy skončí svůj výpočet). Formálně uvažujeme vstup $\mathcal{A}(k,x)$ tak, že $\mathcal{A}$ rozhoduje problém zastavení k-tého algoritmu $\mathcal{U}(k,x)$ na vstupu x. Nechť $\bar{\mathcal{A}}$ značí "opak" algoritmu $\mathcal{A}$, tj. $\bar{\mathcal{A}}$ zavolá $\mathcal{A}$ na daný vstup a po odpovědi NE se zastaví, kdežto po odpovědi ANO upadne do nekonečné smyčky.

Nechť slovo $\langle k,y\rangle$ kóduje dvojici parametrů k a y. Podle Důsledku 7.7 existuje přirozené ℓ takové, že $\bar{\mathcal{A}}(k,\langle k,x\rangle) \equiv \mathcal{U}(\ell,\langle k,x\rangle)$. Položme si otázku, zda se zastaví výpočet algoritmu $\bar{\mathcal{A}}(\ell,\langle \ell,x\rangle)$ pro jakékoliv x. Pokud ano, znamená to, že $\mathcal{A}(\ell,\langle \ell,x\rangle)$ odpověděl NE, tedy že ℓ-tý algoritmus se na vstupu $\langle \ell,x\rangle$ nezastaví. Ale ℓ-tý algoritmus je právě náš algoritmus $\mathcal{U}(\ell,\cdot) \equiv \bar{\mathcal{A}}(\cdot)$, který se podle předpokladu zastavil na vstupu $\langle \ell,x\rangle$. Pokud se naopak výpočet $\bar{\mathcal{A}}(\ell,\langle \ell,x\rangle)$ nezastaví, znamená to obdobně, že ℓ-tý algoritmus se na vstupu $\langle \ell,x\rangle$ zastaví, což je opět ve sporu s předpokladem. Proto v žádném případě algoritmus $\mathcal{A}$ nerozhodl správně o zastavení algoritmu $\bar{\mathcal{A}}$, a tudíž $\mathcal{A}$ neřeší problém zastavení. Problém zastavení algoritmické řešení nemá. □

Úlohy k řešení

(7.3.1) *Lze algoritmicky poznat, zda daný program dělá přesně to, co by měl? (Problém automatizovaného testování softwaru.)*

****(7.3.2)** *Lze automaticky detekovat zacyklení výpočtů v tabulkovém programu? (spreadsheet, neomezeně velká tabulka)*

Rozšiřující studium

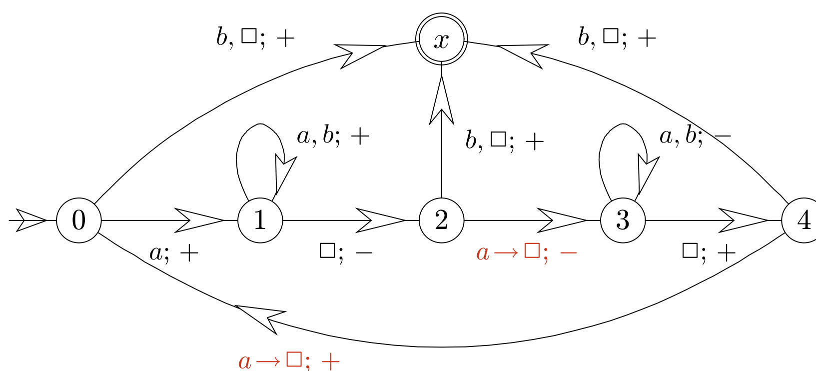
I zde jsme se z důvodu nedostatku času dopustili značného zjednodušení výpočetních modelů TS a RAM na úplně základní poznatky potřebné pro pochopení dalšího výkladu. Pro lepší pochopení Turingova stroje doporučujeme shlédnutí počítačových animací, které jsou přílohami skript [8]. Tyto animace jsou pro vás extrahované na web stránkách výuky [5]. Podrobný (místy až příliš detailní) popis modelu RAM, včetně příkladů rozepsání některých algoritmů až na jednotlivé instrukce, je uveden v textu [8].

7.4 Cvičení: Výpočty na TS a RAM modelech

Příklad 7.9. *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' zbyde ε.)*

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 $\square$, 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 pak můžeme umazat a od začátku a umažeme až po návratu zpět.

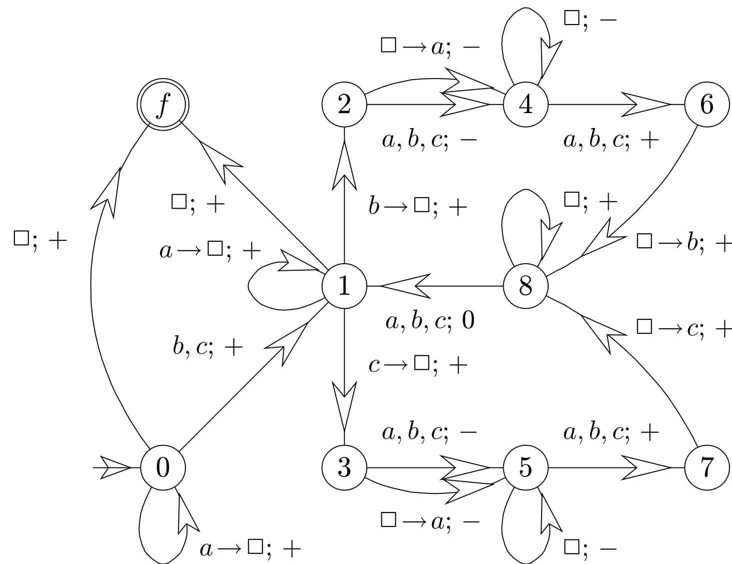
Další otázkou je, jak si spočítáme, kolik a je na začátku i na konci společných. Bohužel zde narazíme na podobné omezení jako u automatů – samotný TS (jeho řídící jednotka) si nemůže znaky a spočítat, protože má jen omezeně mnoho stavů. Proto budeme muset znaky a umazávat postupně a synchronizovaně.



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. □

Příklad 7.10. *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.*

Příklad se zdá jednoduchý – TS by mohl projít všechny znaky slova a znak a přepíše na $\Box$. 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 1 dopředu. (Přepisování a na $\Box$ 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 $\Box$. Proč je tam zapisován znak a? Čtení znaku $\Box$ 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 stejně pak smaže. $\Box$

Příklad 7.11. *Popišme slovně, jaké instrukce stroje RAM by vykonaly následující programový kód. Kolik instrukcí se přesně vykoná?*

```
for (n=i=1; i<10; i++)  n = n*i;
```

Nechť I a N jsou adresy, na kterých jsou uloženy proměnné i a n.

- Na adresy I a N uložíme konstanty 1.
- Do registru r_1 načteme hodnotu z I.
- Otestujeme, zda hodnota r_1 je menší než 10. Pokud ne, skončíme.

– Do registru r_2 načteme hodnotu z N.

– Vynásobíme r_2 hodnotou r_1.

– Uložíme r_2 na adresu N.

– Zvětšíme hodnotu r_1 o 1.

– Uložíme r_1 na adresu I.

– Skočíme zpět na druhý bod.

Zkušený programátor může teď namítnout, proč vůbec hodnoty i a n ukládáme do paměti, když je rovnou můžeme držet v registrech. Je však třeba si uvědomit, že program může mít mnoho vnořených cyklů a jiných proměnných, kdežto počet registrů je podle definice omezený (neříkáme explicitní číslo, ale měli bychom registry používat jen na výpočty, ne na držení proměnných!).

Nyní již můžeme přesně spočítat počet vykonaných elementárních instrukcí: Na začátku jsou vykonány dva kroky inicializace. Pak následuje 8 instrukcí těla cyklu, každá z nich trvá jeden krok. Celkem se tento cyklus vykoná devětkrát (pro hodnoty $i = 1, 2, \ldots, 9$). Po konci cyklu ještě musíme dopočítat jednu instrukci zastavení. Celkem tedy kroků je:

$$2 + 8 \cdot 9 + 1 = 75$$

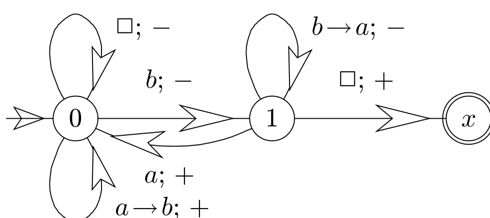
□

Úlohy k řešení

(7.4.1) *Srovnejte si počty kroků, které vykonává stroj RAM a ekvivalentní Turingův stroj. Na jakých operacích Turingův stroj ztrácí nejvíce času?*

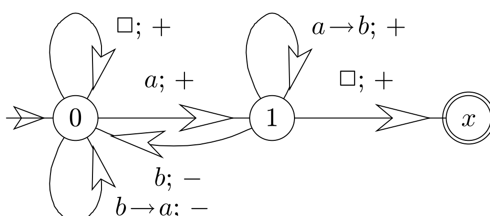
(7.4.2) *Navrhněte Turingův stroj, který rozpoznává palindromy, tj. stroj se zastaví právě tehdy, když se zadané slovo čte stejně od začátku jako od konce.*

(7.4.3) *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.*



***(7.4.4)** *Popište slovně, na jakých slovech se zastaví výpočet předchozího Turingova stroje a co se stane s daným vstupem. Stroj nyní začíná výpočet s hlavou na prvním znaku zprava.*

***(7.4.5)** *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.*



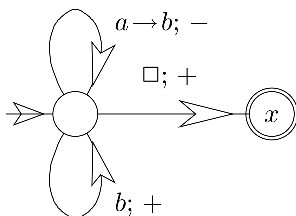
*(**7.4.6**) *Navrhněte jednopáskový Turingův stroj, který dané číslo zapsané v binární soustavě vydělí třemi. Začíná se na slově vlevo.*

Návod: Vzpomeňte si na klasický školní algoritmus dělení čísel a postupujte přesně podle něj.

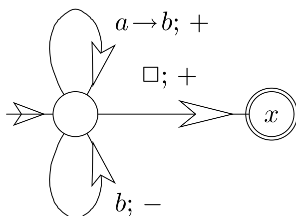
(**7.4.7**) *Navrhněte jednopáskový Turingův stroj, který pracuje s (páskovou) abecedou $\{a, b, c, \Box\}$ a který vykonává následující výpočet:*
Na začátku je na pásce napsáno libovolné slovo $w \in \{a, b\}^*$ a zbytek pásky je vyplněn $\Box$. Hlava stroje je na prvním znaku slova w. Váš Turingův stroj musí vždy skončit výpočet a po skončení musí mít někde na pásce napsáno slovo $\underbrace{c \ldots c}_{k}$, kde k je počet přechodů mezi písmeny a, b (v obou směrech, tj. počítáte jak přechod $\ldots ab \ldots$, tak i $\ldots ba \ldots$) v původním slově w. Zbytek pásky musí být opět vyplněn $\Box$.

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 $cccc$.

(**7.4.8**) *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.)*



(**7.4.9**) *Zjistěte, kolik přesně kroků provede níže zakreslený Turingův stroj v závislosti na daném slově w nad abecedou $\{a, b\}$.*



8 Výpočetní složitost algoritmů a problémů

Úvod

Náročné uživatele softwaru jistě velmi zajímá, jak "rychlé" vlastně jejich používané algoritmy a programy jsou. Asi se shodneme, že při dnešní rychlosti hardwaru počítačů už příliš nezáleží, jak rychlý je třeba textový editor, ale u náročných výpočtů (třeba ve vědeckých výpočtech nebo v 3D grafice) bývá rychlost použitých algoritmů kritická i při seberychlejším hardwaru. Co ale znamenají slova "rychlý algoritmus"? V této přednášce proto přejdeme od obecných teoretických úvah o algoritmech k praktičtěji zaměřeným odhadům délky výpočtu algoritmů.

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 $\mathcal{P}$ všech efektivně řešitelných problémů.

Cíle

V této lekci zavedeme způsob(y) měření rychlosti algoritmu a časové složitosti (náročnosti) problémů. Ukážeme si dále stručný přehled časových složitostí některých základních programátorských problémů (jako třídění, prohledávání, aritmetika, atd). Definujeme třídu $\mathcal{P}$ všech problémů řešitelných v polynomiálním čase.

8.1 Délka výpočtu

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).

`Definice`: *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 8.1. ***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. . .)

`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 *quick-sort*, 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. . .

Úlohy k řešení

(8.1.1) *Máme za úlohu sečíst dvě (dlouhá) k-místná čísla. Jaká je velikost vstupu v tomto problému? (V asymptotické notaci.)*

(8.1.2) *Na vstupu algoritmu je dán obecný jednoduchý graf s n vrcholy. Jakou má tento vstup délku?*

***(8.1.3)** *Na vstupu algoritmu je dán rovinný graf s n vrcholy. Jakou má tento vstup délku? Je menší než u obecného grafu?*

8.2 Časová složitost problému

Vzpomeňme si, že problém je formálně definován 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 $P(w)$.

Definice 8.2. ***Časová složitost problému*** $P : \Sigma^* \to \Sigma^*$
je infimum funkcí časových složitostí všech algoritmů $\mathcal{A}$ řešících problém P. Přitom funkce f, g porovnáváme předpisem $f \leq g$ pokud $f(n) \leq g(n)$ až na konečně mnoho výjimek n.

`Komentář:` Pojem infimum je definovaný jako *největší dolní mez*, a navíc díky použitému způsobu porovnání funkcí (až na konečně mnoho hodnot) vyjde jako infimum ne jedna konkrétní funkce, nýbrž celá třída funkcí, chovajících se asymptoticky stejně. Proto pro naše účely, zvláště s ohledem na použití asymptotického zápisu, si lépe můžeme infimum představit jako "nejmenší asymptotickou funkci" časové složitosti řešící náš problém.

Definice časové složitosti problému s sebou nese jeden podstatný problém: 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 $f(n)$. Jak však ukážeme, že neexistuje algoritmus s menší složitostí než $f(n)$? Všechny algoritmy přece reálně prozkoumat nemůžeme.

V praxi je situace taková, že jen málokdy umíme odvodit *dolní odhad* na časovou složitost problému. (Však i samotná matematická definice dolního odhadu časové složitosti je dosti komplikovaná a problematická, jak vidíme výše.) Proto se používá pro časovou složitost právě zápis jen *horního odhadu* $O(f(n))$. Často také zjednodušeně říkáme, že složitost problému je lineární $O(n)$, kvadratická $O(n^2)$, kubická $O(n^3)$, atd...

Příklad 8.3. *Jakou má časovou složitost problém vypočítat průměr z daného pole n čísel?*

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 $O(n)$:

```
for (i=s=0; i<n; i++)  s += a[i];
p = s/n;
```

Časová složitost našeho problémy tedy je $\Theta(n)$. □

Příklad 8.4. *Jakou časovou složitost má problém rozložit dané (velké) číslo ℓ na prvočinitele?*

Zajisté znáte jednoduchý školní postup – zkoušet ℓ dělit všemi čísly postupně od 2 do $\lfloor\sqrt{\ell}\rfloor$. (Po nalezení dělitele pokračujeme obdobně s podílem, jinak máme prvočíslo.) Jestliže zanedbáme pro zjednodušení jednu operaci dělení jako jeden krok, vyjde nám tak doba výpočtu $O(\sqrt{\ell})$.

Je to však skutečně tak, jak jsme napsali? Co je ℓ? To přece není délka vstupu, kterým je "velké" číslo! Uvědomte si, jak bude takové číslo zadané – zapsané v desítkové či binární soustavě (může být i velmi dlouhé). Délkou n vstupu je počet číslic zápisu ℓ, tedy zhruba

$$n = \Theta(\log \ell)\,.$$

Potom však

$$\sqrt{\ell} \simeq \sqrt{2^n} = 2^{n/2}\,,$$

ale to je už exponenciální funkce složitosti $O(2^{n/2})$, která roste strašně rychle. Pokud například dokážete rozložit 10-místné číslo na prvočísla, pak obdobně rozložit 100-místné číslo, které naneštěstí nemá malé prvočíselné dělitele, je prakticky nemožné!
(Na této nemožnosti rozkladu velkých čísel je založena šifra RSA.) □

Úlohy k řešení

(8.2.1) *Jakou časovou složitost má problém setřídění n daných čísel?*

(8.2.2) *Je dána matice* `A` *(tj. dvourozměrné pole) o rozměrech $n \times n$ s hodnotami $0, 1$, která definuje graf G s vrcholy $\{1, 2, \dots, n\}$ následovně: Vrcholy i, j jsou spojeny hranou v G právě když* `A[i,j]==A[j,i]==1`. *Kolik elementárních kroků (časová složitost) je potřeba ke zjištění největšího stupně vrcholu grafu G v závislosti na n?*

(8.2.3) *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?*

***(8.2.4)** *Jakou časovou složitost má problém násobení dvou matic $n \times n$?*

8.3 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.

Tvrzení 8.5. *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 bubble-sort je $O(n^2)$.*
- *Mnohem rychleji lze pole* `A` *setřídit algoritmy heap-sort nebo merge-sort v čase $O(n \cdot \log n)$.*
- *V setříděném poli* `A` *lze nalézt prvek binárním vyhledáváním v čase $O(\log n)$.*

Tvrzení 8.6. *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. hašovací tabulka, která v praktických aplikacích také dává velice krátké (až konstantní) průměrné časy operací, přestože nejhorší složitost bývá až $O(n)$.*

Tvrzení 8.7. *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! Podívejte se znovu na Příklad 8.4.

Tvrzení 8.8. *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 heap-sort či merge-sort.

Věta 8.9. *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 $M = \{1, 2, \dots, n\}$. 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.

Úlohy k řešení

(8.3.1) *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.)*

(8.3.2) *Jak rychle byste dokázali zjistit, zda daný graf na n vrcholech má nějakou nezávislou množinu velikosti k?*

***(8.3.3)** *Jak rychle byste dokázali nalézt konvexní obal z n bodů v rovině?*

***(8.3.4)** *Je možné navrhnout uspořádanou datovou strukturu, do které by se daly nové prvky přidávat i staré odebírat v (průměrném) konstantním čase?*

8.4 Třída $\mathcal{P}$ a polynomiální redukce

Jak jsme již uvedli v dřívější přednášce, mezi rychlostí růstu polynomiálních a exponenciálních funkcí je propastný rozdíl. To je ilustrováno následujícím příměrem:

> `Komentář:` Pokud velikost vstupu problému vzroste třeba dvojnásobně, tak polynomiální funkce časové složitosti (tj. čas výpočtu algoritmu) vzroste konstantně krát, přesněji 2^c-krát, kde c je exponent onoho polynomu. Praktickými slovy, pokud chceme zvětšit několikanásobně rozsah zpracovaných dat, stačí nám koupit několikrát výkonnější počítač nebo případně sestavit paralelní cluster počítačů. Na druhou stranu při exponenciální funkci časové složitosti (tvaru d^n) už při zvětšení velikosti vstupu o pouhých několik jednotek vzroste čas výpočtu několikrát, a proto i nákup mnoha výkonnějších počítačů by nám umožnil záběr takového algoritmu zvětšit jen nepatrně.
>
> V informatice se proto problémy řešené s polynomiální časovou složitostí považují za "rozumné", kdežto ty vyžadující exponenciální časovou složitost se považují za "nezvládnutelné". (Přestože existují funkce rostoucí rychleji než polynomy, ale pomaleji než exponenciála, v odhadech složitosti algoritmů se takové příklady téměř nevyskytují.)

Definice 8.10. ***Třída PTIME,*** zkráceně třída $\mathcal{P}$,
je třídou všech problémů $P : \Sigma^* \to \Sigma^*$, které lze řešit s časovou složitostí $O(n^c)$ pro nějakou konstantu $c > 0$, tj. v *polynomiálním čase*. Problémům v třídě $\mathcal{P}$ se jinak říká *efektivně řešitelné*.

Věta 8.11. *Definice třídy $\mathcal{P}$ je "robustní" – nezávisí na použitém modelu výpočtu (stroje RAM či Turingova stroje).*

`Důkaz:` Dle Věty 7.6 lze vzájemně simulovat výpočty RAM a TS v polynomiálním počtu kroků. (Dle Church-Turingovy teze se totéž týká i jiných možných modelů výpočtu.) Jelikož složením dvou polynomů je opět polynom, lze každý algoritmus pracující na RAM v polynomiálním čase simulovat na Turingově stroji taktéž v polynomiálním čase a naopak. □

> `Komentář:` Možná by si čtenář myslel, že třída $\mathcal{P}$ 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 formálně neplatila předchozí věta – všimněte si, že TS v jejím důkaze simuluje výpočet RAM v polynomiálním čase s výrazně vyšším exponentem c'. (Také by při pevném omezení exponentu nebylo možno používat polynomiální převody, které definujeme níže.)
>
> Přesto je vhodné považovat třídu $\mathcal{P}$ za třídu rozumně zvládnutelných problémů, neboť je dobře známo následující:

`Fakt:` Je empiricky mnohokrát ověřeno, že pokud praktický (rozumně definovaný) problém patří do třídy $\mathcal{P}$, pak tento problém lze řešit s časovou složitostí $O(n^c)$ s "rozumně malým" exponentem c, obvykle třeba $c \leq 5$.

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:` 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`: *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.

`Komentář`: 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... (Tato úvaha je podstatou následující *teorie $\mathcal{NP}$-úplnosti*.)

Úlohy k řešení

(8.4.1) *Jak převedeme problém odečítání dvou čísel $a - b$ na problém sčítání $c + d$?*

(8.4.2) *Jak zobecněně převedeme problém násobení dvou čísel $a \cdot b$ na problém sčítání $c + d$? Používal se tento převod v minulosti často?*

(8.4.3) *Více příkladů na polynomiální převody a úloh k řešení bude uvedeno v příští lekci v souvislosti s třídou $\mathcal{NP}$.*

Rozšiřující studium

Hezky zpracovaný přehled efektivních algoritmů z různých oblastí, včetně odvození jejich časových složitostí, může čtenář najít v knize [9].

8.5 Cvičení: Určení časové složitosti algoritmu

Nejprve začneme elementárními příklady počítání počtu kroků v algoritmech.

Příklad 8.12. *Průměr z daných $n > 1$ čísel spočítáme následující funkcí:*

```
double prumer(double x[], int n) {
    int i;  double z = 0.0;
    for (i=0; i<n; i++)
        z = z + x[i];
    return z/n;
}
```

Určete, kolik tato funkce `prumer` *vykoná aritmetických operací v závislosti na n.*

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 $\Theta(n)$. □

Příklad 8.13. *Určete, kolik průchodů vnitřním cyklem provede pro vstup n následující jednoduchý program zapsaný v jazyce C.*

```
scanf("%d",&n); // vstup cisla n
for (i=0; i<n*n; i++)
    for (j=0; j<i; j++)
        printf("jeden pruchod\n");
```

Je to v asymptotické notaci $\Theta(n^2)$ nebo $\Theta(n^3)$ nebo $\Theta(n^4)$?

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 $\Theta(n^4)$. □

Příklad 8.14. *Jednoduchá implementace Euklidova algoritmu největšího společného dělitele dvou přirozených čísel a, b počítá výsledek následovně:*

```
while (a>0 && b>0) {
    while (a<=b)   b = b-a;
    x = a; a = b; b = x;
}
printf("NSD je %d",a+b);
```

Nechť k označuje počet míst (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.)

Jelikož se v každé iteraci vnitřního cyklu jedno z čísel a, b aspoň o 1 zmenší, nemůže nastat více než $2 \cdot 2^k = 2^{k+1}$ jeho iterací celkem. (Může sice nastat, že se vnitřní cyklus nevykoná vůbec, ale to jen na počátku iterace vnějšího cyklu.) Každá tato iterace 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 $\Theta(2^k)$. □

Příklad 8.15. *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.)*

```
while (a>0 && b>0) {
    b = b % a;  // % je "modulo"
    x = a; a = b; b = x;
}
printf("NSD je %d",a+b);
```

Pokud $b \geq a$, tak číslo (zbytek) $z = b\%a$ 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 8.14.

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. □

V druhé části cvičení se na rekurzivní algoritmy. Jejich časovou složitost budeme určovat za použití vzorců z Oddílu 6.2.

Příklad 8.16. *Jak jistě znáte, 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?

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 $T(n) \le 2T(k) + T(k+1) + \ldots$.

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

$$T(n) \le 2T(k) + T(k+1) + O(n) = 2T\left(\frac{n}{2}\right) + T\left(\frac{n}{2} + 1\right) + O(n)$$

a po zanedbání "+1" v $T(k+1)$ vyjde

$$T(n) \le 3T\left(\frac{n}{2}\right) + O(n)\,.$$

Podle Lematu 6.4 je řešením tohoto rekurentního vztahu asymptoticky

$$T(n) = O(n^{\log_2 3}) \doteq O(n^{1.585})\,.$$

□

Příklad 8.17. *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?) Nechť daná čísla jsou (neuspořádaně) uložena v poli A a výsledek – k-té z nich, je počítán funkcí* `vyber`(A,n,k)*. Indexy počítejme od 0. Pro rychlý výpočet použijeme následující rekurzivní algoritmus, svým způsobem podobný algoritmu quick-sort:*

- *V poli A zvolíme libovolně pivota $A[i]$. Do nového pole B vložíme všechny prvky A menší než $A[i]$, nechť jejich počet je b. Do nového pole C vložíme všechny prvky A větší než $A[i]$, nechť jejich počet je c.*
- *Pokud je $b \le k < n - c$, je výsledkem* `vyber`$(A,n,k) = A[i]$.
- *Pokud $k < b$, výsledek spočítáme rekurzivním výpočtem* `vyber`$(A,n,k) =$ `vyber`(B,b,k)*. Pokud $k \ge n - c$, výsledek spočítáme rekurzivním výpočtem* `vyber`$(A,n,k) =$ `vyber`$(C,c,k-n+c)$.

Jaká je časová složitost popsaného algoritmu?

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 B velikosti $n-1$. Takový případ žádný ze vzorců z Oddílu 6.2 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 B 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 B, C téměř "napůl". (Je to stejný případ jak s algoritmem quick-sort.) □

Úlohy k řešení

(8.5.1) *Určete, kolik průchodů vnitřním cyklem provede pro vstup n následující jednoduchý program zapsaný v jazyce C.*

```
scanf("%d",&n); // vstup cisla n
for (i=0; i<n; i++)
    for (j=0; j<i*i; j++)
        printf("jeden pruchod\n");
```

Je to v asymptotické notaci $\Theta(n^2)$ nebo $\Theta(n^3)$ nebo $\Theta(n^4)$?

(8.5.2) *Určete, kolik průchodů* `for` *cyklem provede následující C-program pro vstupní číslo n. Zapište výsledek v asymptotické notaci $\Theta(\cdot)$.*

```
scanf("%d",&n); // vstup cisla n
for (i=1; i<n; i=i+i)
    printf("jeden pruchod\n");
```

***(8.5.3)** *Určete, kolik průchodů* `for` *cyklem provede následující C-program pro vstupní číslo n. Zapište výsledek v asymptotické notaci $\Theta(\cdot)$.*

```
scanf("%d",&n); // vstup cisla n
for (i=j=1; i<n; i+=j++)
    printf("jeden pruchod\n");
```

(Příkaz `i+=j++` *znamená, že do* `i` *se přičte hodnota* `j` *a pak se* `j` *inkrementuje.)*

(8.5.4) *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). Za úkol je zjistit, zda tato posloupnost je permutací.*

***(8.5.5)** *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?*

***(8.5.6)** *Jak byste v Příkladě 8.16 odvodili výsledný asymptotický odhad na $T(n)$ bez zanedbání "+1" v $T(k+1)$?*

***(8.5.7)** *Jak byste upravili algoritmus v Příkladě 8.17, aby počítal v (nejhorším) lineárním čase?*

Návod: Je potřeba najít vhodný způsob výběru pivota $A[i]$ tak, aby bylo zaručeno, že rozdělení *nebude velmi nevyvážené.*

9 Základy $\mathcal{NP}$-úplnosti

Úvod

V minulé lekci jsme uvedli třídu $\mathcal{P}$ 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 $\mathcal{P}$.) 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 $\mathcal{NP}$.

Zhruba řečeno, problém patří do třídy $\mathcal{NP}$, pokud kladnou odpověď na něj lze prokázat (ve smyslu "uhodnout a ověřit") výpočtem, který běží v polynomiálním čase. Definice třídy $\mathcal{NP}$ se tak týká výhradně rozhodovacích problémů. To však není na velkou újmu obecnosti uvažování, neboť vlastně každý problém lze nahradit několika rozhodovacími verzemi.

Třída $\mathcal{NP}$ je důležitá hlavně proto, že zahrnuje rozhodovací verze skoro všech 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 $\mathcal{NP}$ nejsme sami schopni efektivně vyřešit. Náplní této i příští přednášky tak bude i stručné pochopení důvodů, proč jsou ve třídě $\mathcal{NP}$ obtížně řešitelné úlohy a jak je poznat.

Cíle

Cílem je definovat třídu $\mathcal{NP}$ všech problémů s "kladnou polynomiální nápovědou" a ukázat její vztah k nedeterministickým výpočtům. (Definice třídy $\mathcal{NP}$ je poměrně obtížná, proto čtenářům doporučujeme si ji přečíst mnohokrát a pokusit se tak proniknout až k jejímu myšlenkovému jádru.) Našim záměrem je ukázat tuto třídu jako přirozené a prakticky motivované rozšíření třídy $\mathcal{P}$ všech efektivně řešitelných problémů. Dále na závěr ukážeme existenci tzv. $\mathcal{NP}$-úplného problému.

9.1 Třída $\mathcal{NP}$

Nyní se budeme zabývat výhradně rozhodovacími problémy, tj. takovými, ve kterých je odpověď typu ANO/NE. To je běžný předpoklad v oblasti teoretické informatiky, neboť rozhodovací verze problémů se mnohem snadněji formálně popisují. Následující definice

přímo navazuje na myšlenku problémů, jejichž kladnou odpověď lze prokázat (s vhodnou nápovědou) efektivně. Vzpomeňme si, že $\mathcal{P}$ označuje třídu všech výpočetních problémů řešitelných algoritmy v polynomiálním čase.

Definice 9.1. ***Třída NPTIME,*** zkráceně $\mathcal{NP}$,
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 $R \in \mathcal{P}$ platí

$$\forall x \in \Sigma^* : \quad P(x) = 1 \iff \left(\exists y \in \Sigma^* : R(x,y) = 1\right) .$$

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! 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".

`Komentář:` Příklady problémů ve třídě $\mathcal{NP}$.

- Všechny rozhodovací problémy z třídy $\mathcal{P}$ patří zároveň do $\mathcal{NP}$: V definici stačí algoritmus počítající $R(x,y)$ nahradit algoritmem efektivně řešícím náš problém, který vstup y zcela ignoruje.
- Otázka, zda dané číslo k je složené, patří do $\mathcal{NP}$. Nápovědou y je některý netriviální dělitel k a funkce R ověřuje dělitelnost.
- Dále sem patří otázka, zda daný graf lze korektně obarvit třemi barvami (aby žádné dva vrcholy spojené hranou nedostaly stejnou barvu). Nápovědou y je takové obarvení.
- Otázka, zda v daném grafu existuje nezávislá množina velikosti k. Nápovědou y je tato množina.
- Otázka, zda v daném grafu existuje Hamiltonovská kružnice (procházející všemi vrcholy). Nápovědou y je tato kružnice.
- Otázka, zda dva grafy jsou isomorfní.
- Problém "loupežníka", zda danou množinu přirozených čísel lze rozdělit na dvě části tak, aby se jejich součty rovnaly.
- Problém, zda pevně daná nevypouštějící gramatika generuje slovo x. Nápovědou y zde je odvození slova x.

Aby čtenář lépe chápal význam zmiňované nápovědy v definici problémů třídy $\mathcal{NP}$, uvedeme ještě další neformální přiblížení:
Při sledování výpočetní složitosti problémů si představujeme, že zadání x problémů nám dává "*nepřítel*", který nás chce co nejvíce potrápit, a proto vymýšlí co nejtěžší případy (proti našemu postupu řešení). Naopak ona *nápověda* y z definice $\mathcal{NP}$ je představována našim "vševědoucím *přítelem*", který nám chce co nejvíce pomoci při řešení daného problému. Proto můžeme očekávat, že nápověda odpovědi ANO přijde tak, jak ji optimisticky očekáváme. Na nás už pak jen zbývá správnost nápovědy v polynomiálním čase ověřit.

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`: Každý problém $P : \Sigma^* \to \Sigma^*$ lze nahradit posloupností rozhodovacích problémů $P_1, P_2, \dots, P_{2k}$ (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`: Všechny rozhodovací verze problémů z $\mathcal{P}$ patří do $\mathcal{NP}$.

Úlohy k řešení

(9.1.1) *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 $\mathcal{NP}$?*

(9.1.2) *Proč patří do třídy $\mathcal{NP}$ problém, zda daný graf má vrcholovou souvislost méně než k?*

***(9.1.3)** *Proč patří do třídy $\mathcal{NP}$ problém, zda daný graf má vrcholovou souvislost naopak alespoň k?*

***(9.1.4)** *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 $\mathcal{NP}$, když napovíme obarvení grafu G méně barvami než obarvení grafu H?*

9.2 Nedeterministický Turingův stroj

V této sekci si formálně rozšíříme možnosti Turingova stroje o nedeterminismus. (Takové teoretické rozšíření nemá žádný reálný význam při modelování počítačů, ale poskytne nám jiný, tradiční způsob popisu třídy $\mathcal{NP}$.)

`Definice`: *Nedeterministický Turingův stroj* je definován obdobně jako deterministický TS, jen přechodová funkce dovoluje nedeterminismus, tedy možnost přechodu do více stavů stroje současně.

`Definice`: Daný rozhodovací problém P je řešen nedeterministickým Turingovým strojem $\mathcal{M}$, jestliže všechny výpočty $\mathcal{M}$ jsou konečné a vydávají výsledek ANO nebo NE, a navíc platí:

- Jestliže odpověď problému $P(w)$ pro vstup w je ANO, pak existuje (alespoň jeden) výpočet $\mathcal{M}$ se vstupem w dávající ANO.
- Jestliže odpověď pro w je NE, pak všechny výpočty $\mathcal{M}$ se vstupem w dávají 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.

Věta 9.2. *Třída $\mathcal{NP}$ je právě třídou všech rozhodovacích problémů, které lze řešit nedeterministickým Turingovým strojem s (nejhorší) polynomiální časovou složitostí.*

`Důkaz`. Nechť problém $P \in \mathcal{NP}$, tj. dle definice existuje polynomiální R, pro které platí
$$\forall x \in \Sigma^* : \quad P(x) = 1 \quad \Longleftrightarrow \quad \left(\exists y \in \Sigma^* : R(x,y) = 1\right) .$$
Nedeterministický Turingův stroj $\mathcal{M}$ implementuje polynomiální výpočet zobrazení $R(x,y)$ tak, že jednotlivé bity "nápovědy" y nahrazuje nedeterministickými rozdvojeními výpočtu na možnosti 0/1. Pak $\mathcal{M}$ 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 $\mathcal{M}$ řešící nějaký problém P odvodíme polynomiální deterministické zobrazení $R(x, y)$, které každé nedeterministické rozdvojení běhu $\mathcal{M}$ nahrazuje deterministickým větvením běhu programu podle nápovědných bitů y. Takže $\mathcal{M}$ někdy odpoví ANO, právě když $R(x, y) = 1$ pro některé y, tj. když $P(x) = 1$. □

9.3 $\mathcal{NP}$-úplný problém

V poslední sekci se zde podíváme na problémy, které jsou ve třídě $\mathcal{NP}$ nejtěžší, tedy těžší než všechny ostatní.

`Definice`: Problém Q nazveme *$\mathcal{NP}$-těžkým,* pokud každý problém ve třídě $\mathcal{NP}$ lze na problém Q převést polynomiálním převodem (Oddíl 8.4). Problém Q nazveme *$\mathcal{NP}$-úplným,* pokud je $\mathcal{NP}$-těžký a náleží do třídy $\mathcal{NP}$.

> `Komentář:` Neformálně řečeno, efektivní řešení některého (kteréhokoliv) $\mathcal{NP}$-těžkého problému by nám poskytlo efektivní řešení všech problémů ve třídě $\mathcal{NP}$, tj. skoro všech běžných problémů.

Jak vlastně poznáme $\mathcal{NP}$-těžké problémy? Pokud již známe nějaký $\mathcal{NP}$-těžký problém, těžkost jiného problému zdůvodníme snadno:

Lema 9.3. *Nechť problém Q je $\mathcal{NP}$-těžký ($\mathcal{NP}$-úplný). Pokud existuje polynomiální převod problému Q na nějaký problém P, pak také P je $\mathcal{NP}$-těžký.*

`Důkaz`. Označme $R_Q : \Sigma^* \to \Sigma^*$ polynomiální převod Q na P. Dle definice $\mathcal{NP}$-těžkého problému pro každý problém $S \in \mathcal{NP}$ 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 $\mathcal{NP}$-těžký problém. □

> `Komentář:` Dávejte si dobrý pozor, v jakém směru převodu Lema 9.3 funguje! Převádí se z problému Q, o kterém je známo, že je těžký, na neznámý problém P. Zjednodušeně řečeno, pokud každý možný vstup $\mathcal{NP}$-těžkého problému Q jsme schopni "přeložit" na vstup jiného problému P (se zachováním stejné odpovědi), pak P také musí být $\mathcal{NP}$-těžký.

`Poznámka`: Existence problému, který je "těžší" než všechny problémy v $\mathcal{NP}$ ($\mathcal{NP}$-těžký) je poměrně intuitivní, ale proč by měl takový problém existovat přímo ve třídě $\mathcal{NP}$? To již intuitivní není a objev *prvního $\mathcal{NP}$-úplného problému* v 1971 přinesl velkou revoluci do oblasti složitosti algoritmů.

Věta 9.4. (Cook) *Existuje $\mathcal{NP}$-úplný problém.*

`Důkaz` této věty je velmi komplikovaný, protože se musí ošetřit mnoho drobností, ale idea je vcelku jednoduchá: Nechť problém $P \in \mathcal{NP}$, tj. dle definice existuje polynomiální R, pro které platí

$$\forall x \in \Sigma^* : \quad P(x) = 1 \quad \Longleftrightarrow \quad (\exists y \in \Sigma^* : R(x, y) = 1) \ .$$

Vyjděme z implementace asociovaného zobrazení R na stroji RAM. Každý krok RAM lze snadno popsat logickou formulí, a proto i celý výpočet RAM pro $R(x, y)$ lze zakódovat logickou formulí, jež roste úměrně počtu kroků, které implementace R vykoná na RAM. Nechť je to formule $\Phi_x(y)$, ve které je vstup x "zadrátovaný" a nápověda y vstupuje do formule ve formě logických proměnných popisujících jednotlivé bity y. Hodnotou $\Phi_x(y)$ je pravda, právě když odpovědí výpočtu R je 1 (ANO).

Vezměme nyní problém *splnitelnosti logických formulí* SAT (v konjunktivním normálním tvaru, což bude blíže popsáno příště): Vstupem je logická formule Φ s volnými proměnnými a otázkou je, zda pro některé y je $\Phi(y)$ pravda (v logickém smyslu), tj. zda je Φ splnitelná některým y. Pokud na vstupu problému dostaneme výše popsanou formuli Φ_x, splnitelnost Φ_x bude znamenat odpověď ANO původnímu problému $P(x)$. Proto každý problém $P \in \mathcal{NP}$ dokážeme (polynomiálně) převést na problém splnitelnosti logických formulí SAT, a tudíž je SAT $\mathcal{NP}$-úplný. □

`Komentář:` Obecně řečeno, $\mathcal{NP}$-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 $\mathcal{NP}$-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 $\mathcal{NP}$-úplných problémů najít a neuspělo. (A na správné rozřešení je vypsána odměna \$1000000.) Přitom nalezení polynomiálního řešení byť jen pro jeden $\mathcal{NP}$-úplný problém by znamenalo (dle definice) polynomiální řešení všech ostatních problémů ve třídě $\mathcal{NP}$. So there is likely no such solution.

Úlohy k řešení

(9.3.1) *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.*

(9.3.2) *Považujme za známé, že problém* 3*-obarvení grafu je $\mathcal{NP}$-úplný. Proč je pak $\mathcal{NP}$-úplný problém* 4*-obarvení?*

***(9.3.3)** *Proč nelze stejně tvrdit, že i problém* 2*-obarvení je $\mathcal{NP}$-úplný, když přece existuje stejný převod problému* 2*-obarvení na problém* 3*-obarvení grafu?*

Rozšiřující studium

Problematika $\mathcal{NP}$-úplnosti je uváděna ve většině knih o teoretické informatice. Na rozdíl od našeho zavedení však obvykle třídu $\mathcal{NP}$ definují pomocí nedeterminismu Turingova stroje. My jsme zde dali přednost ekvivalentní "nápovědné" definici, která nic nemění na podstatě věci, ale mění náš styl prezentace. Pro jiné, obšírnější a přitom poměrně snadno pochopitelné zavedení $\mathcal{NP}$-úplnosti bychom doporučili třeba knihu [9].

9.4 Cvičení: Polynomiální převody a příslušnost do $\mathcal{NP}$

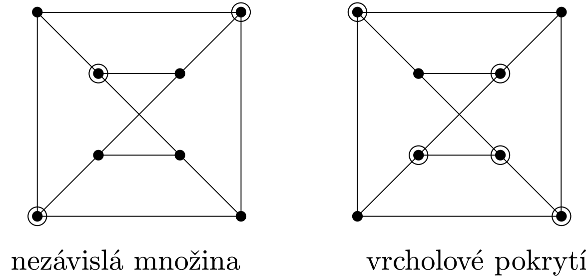
Úvodem poznamenejme, že se budeme zabývat převody mezi problémy, které jsou $\mathcal{NP}$-úplné, ale tomuto aspektu se ještě v této lekci nebudeme přímo věnovat. Ještě si připomeňme, že uvažujeme pouze rozhodovací verze problémů, takže pokud se ptáme na nějaký objekt (třeba v grafu), zajímá nás jen fakt jeho existence, ne jeho konkrétní podoba. (Tu lze obvykle snadno dodatečně odvodit, pokud umíme rozhodovat samotnou existenci.)

Příklad 9.5. *Problém nezávislé množiny se ptá, zda v grafu existuje podmnožina k vrcholů nespojených žádnými hranami. Problém vrcholového pokrytí se ptá, zda v grafu existuje podmnožina m vrcholů dotýkajících se všech hran. (Vrchol se dotýká hrany, pokud je jejím koncem.) Ukažme si podrobně, jak se problém nezávislé množiny polynomiálně převede na problém vrcholového pokrytí.*

Nejprve si ujasněme, co je vstupem a výstupem kterého problému. Pro nezávislou množinu je vstupem dvojice G, k, kde G je graf a k přirozené číslo. Pro vrcholové pokrytí

je obdobně vstupem dvojice G, m. (V případě nezávislé množiny se přirozeně snažíme dostat co největší vyhovující k, kdežto u vrcholového pokrytí co nejmenší m.) V obou případech se jedná o rozhodovací problémy, takže odpovědí je ANO/NE.

Všimněme si následujícího jednoduchého faktu: Pokud $I \subseteq V(G)$ je nezávislá množina v grafu G, pak žádná hrana G nemá oba konce v I. To ale znamená, že doplněk množiny $J = V(G) \setminus I$ se dotýká všech hran grafu G, a tudíž J je vrcholovým pokrytím. Pokud $|I| = k$, pak $|J| = |V(G)| - k = m$. Naopak doplňkem vrcholového pokrytí J je ze stejného důvodu nezávislá množina $I = V(G) \setminus J$. Příklad je na následujícím obrázku. (Zakreslete si to také do některého vlastního grafu.)

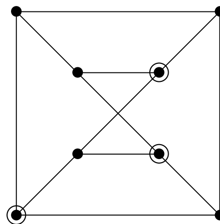


nezávislá množina vrcholové pokrytí

Takže stačí vstup G, k problému nezávislé množiny převést na vstup G, m, $m = |V(G)| - k$ problému vrcholového pokrytí, ze kterého už získáme správnou odpověď i na původní problém. Tento převod dokonce spočítáme v konstantním čase, jen provedeme jedno odečtení. (Všimněme si ještě jedné zajímavosti – při našem převodu se vůbec nezměnil graf G, jen číslo k na m, ale to je pouze specifickou vlastností tohoto jednoduchého převodu. Ve složitějších případech však dochází ke změně celého vstupu, včetně grafu.) □

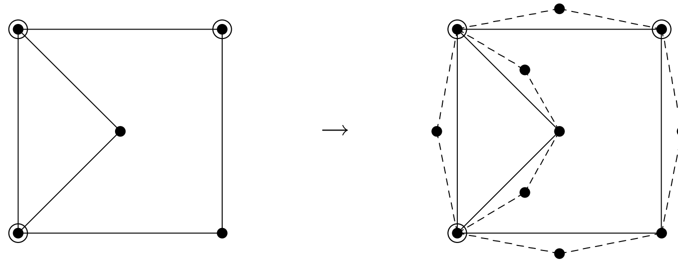
Příklad 9.6. *Problém dominující množiny se ptá, zda v grafu existuje podmnožina m vrcholů taková, že každý jiný vrchol je s některým z nich spojený hranou. Ukažme si podrobně, jak se problém vrcholového pokrytí polynomiálně převede na problém dominující množiny na souvislých grafech.*

Definice dominující množiny je velmi podobná vrcholovému pokrytí, že? Vlastně by mělo stačit jaksi ke každé hraně daného grafu G (ve kterém hledáme vrcholové pokrytí) přiřadit nějaký nový vrchol, který bude třeba po převodu dominovat. Abychom se vyhnuli patologickým případům, předpokládáme souvislé grafy s více než jedním vrcholem. Začneme jednoduchou ukázkou, jak třeba dominující množina může vypadat.



(Dokážete odpovědět, proč graf z obrázku nemá dominující množinu velikosti 2?)

Nyní přejdeme k popisu naznačeného převodu ze vstupu G, m problému vrcholového pokrytí na vstup H, m' problému dominující množiny.



Graf H vytvoříme z grafu G přidáním, pro každou hranu $e \in E(G)$, nového vrcholu v_e spojeného hranami do obou koncových vrcholů hrany e. (Tak se vlastně z každé hrany stane trojúhelník s třetím novým vrcholem, viz naznačený obrázek.) Číselný parametr $m' = m$ zůstane tentokrát nezměněn.

Abychom zdůvodnili, že se skutečně jedná o převod problémů, musíme dokázat implikace v obou směrech: Že vrcholové pokrytí $C \subseteq V(G)$ je zároveň dominující množinou v novém grafu H a že dominující množina $D \subseteq V(H)$ vytváří i stejně velké vrcholové pokrytí v původním grafu G. První část je snadná, podle definice vrcholového pokrytí každá hrana e grafu G má některý konec u v množině C, takže jak druhý konec hrany e, tak i nově přidaný vrchol v_e v grafu H jsou dominovány z vrcholu $u \in C$. Navíc z předpokladu souvislosti G plyne, že žádné další izolované vrcholy v H nejsou, takže C je zároveň dominující množinou v H.

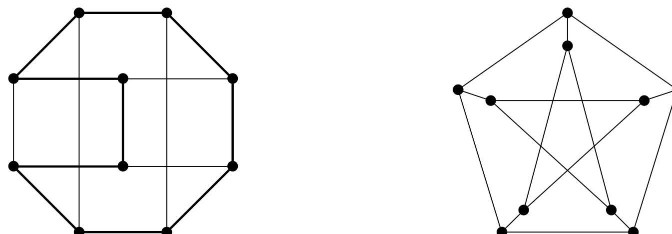
Naopak vezměme dominující množinu $D \subseteq V(H)$ v novém grafu H. (Pozor, nelze hned říci, že by D byla vrcholovým pokrytím v G, neboť D může obsahovat přidané vrcholy, které v G nebyly.) Definujeme novou množinu $D' \subseteq V(G)$ takto: Pokud $w \in D \cap V(G)$, pak $w \in D'$. Jinak pro $w \in D$, kde $w = v_e$ byl přidán pro hranu $e \in E(G)$, dáme do D' libovolný z konců hrany e. Potom $|D'| \leq |D|$ a D' je vrcholovým pokrytím v původním grafu G, neboť pro každou hrany $e \in E(G)$ je přidaný vrchol $v_e \in V(H)$ dominován v grafu H množinou D, a tudíž hrana e bude mít některý konec v D'.

Dokázali jsme tedy, že se jedná o převod problému, a zbývá zdůvodnit, že tento převod je spočítán v polynomiálním čase. Pokud G má n vrcholů, má nejvýše $O(n^2)$ hran, a proto nový graf H má velikost $O(n^2)$ a v takovém čase jsme snadno schopni jej sestrojit. Je to polynom v n. □

V druhé části cvičení se budeme věnovat otázkám, proč některé problémy náleží či nenáleží do třídy $\mathcal{NP}$. Stále zůstáváme u rozhodovacích verzí problémů.

Příklad 9.7. *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 $\mathcal{NP}$ problém poznat, zda daný graf G obsahuje Hamiltonovskou kružnici?*

Jak již bylo řečeno výše u definice, třída $\mathcal{NP}$ 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 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 $\mathcal{NP}$. □

Příklad 9.8. *Patří do třídy $\mathcal{NP}$ problém poznat, zda daný graf G obsahuje nejvýše čtyři Hamiltonovské kružnice?*

Čtenář opět může navrhnout, že vhodnou nápovědou pro příslušnost do třídy $\mathcal{NP}$ 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 $\mathcal{NP}$. Avšak pokud bychom otázku negovali, tj. ptali se, zda graf G obsahuje více než čtyři Hamiltonovské kružnice, tak by už problém do třídy $\mathcal{NP}$ náležel. (Napověděli bychom některých pět Hamiltonovských kružnic.) Proto vidíte, jak je důležité správně se v zadání problému ptát. □

Úlohy k řešení

(9.4.1) *Rozhodněte, které z následujících problémů patří do třídy $\mathcal{P}$ 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.

(9.4.2) *Analogicky k Příkladu 9.5 ukažte převod problému vrcholového pokrytí na nezávislou množinu. (Tj. opačný směr.)*

(9.4.3) *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?*

***(9.4.4)** *Dokážete najít převod problému dominující množiny na vrcholové pokrytí? (Tj. opačný směr k Příkladu 9.6.)*

(9.4.5) *Patří do třídy $\mathcal{NP}$ problém zjistit, zda graf G obsahuje dvě Hamiltonovské kružnice, které nesdílí žádnou hranu?*

(9.4.6) *Patří do třídy $\mathcal{NP}$ problém zjistit, jaká je barevnost grafu?*

(9.4.7) *Patří do třídy $\mathcal{NP}$ problém zjistit, zda graf G je rovinný? A co třeba negace tohoto problému?*

***(9.4.8)** *Patří do třídy $\mathcal{NP}$ problém zjistit, zda graf G obsahuje právě jedinou Hamiltonovskou kružnici? A co třeba negace tohoto problému?*

*(**9.4.9**) *Je známo, že do třídy $\mathcal{NP}$ patří problém k-obarvení (zda graf lze obarvit korektně k barvami, tj. zda barevnost je $\leq k$) pro všechna k. Patří ale do třídy $\mathcal{NP}$ problém zjistit, zda graf G má barevnost právě k? Pro která k?*

(**9.4.10**) *Rozhodněte, které z následujících problémů patří do třídy $\mathcal{NP}$.*
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.

10 $\mathcal{NP}$-úplné problémy

Úvod

Jak už bylo řečeno, problém patří do třídy $\mathcal{NP}$, pokud jeho odpověď ANO lze prokázat (ve smyslu "uhodnout a ověřit") výpočtem, který běží v polynomiálním čase. Již jsme si také neformálně ukázali, že ve třídě $\mathcal{NP}$ existuje problém, který je "nejobtížnější" ze všech z nich.

Aby nebylo špatným zprávám konec, ukážeme si vhodnými převody, že oněch nejobtížnějších (přesněji $\mathcal{NP}$-úplných) problémů je mnohem více, bohužel by se dalo říci většina. To ostatně ukazuje, proč jsme zatím v praxi tak málo úspěšní při počítačovém řešení mnohých praktických problémů – přesné a efektivní řešení $\mathcal{NP}$-úplných úloh se totiž všeobecně považuje za nemožné. Nejsme zatím schopni ani naopak matematicky zdůvodnit, proč by efektivní řešení těch nejobtížnějších problémů v $\mathcal{NP}$ nemělo existovat, třebaže tomu všichni odborníci věří.

Náplní této lekce bude ukázka základních abstraktních $\mathcal{NP}$-úplných problémů, s jejichž variacemi se při řešení praktických problémů setkáte. Praktická rada pak zní: Vidíme-li $\mathcal{NP}$-úplný problém, nebudeme marnit čas snahami o jeho rychlé a přesné obecné řešení, ale raději se soustředíme na řešení přibližná či částečná. (Tj. ta, která uspokojivě odpoví alespoň v některých, dokonce někdy v mnoha, praktických případech.)

Cíle

Našim cílem je jednak ukázat čtenáři, kolik je všude kolem snadno popsaných $\mathcal{NP}$-úplných úloh, za druhé jej naučit odvozovat polynomiální převody, kterými se $\mathcal{NP}$-těžkost úloh zdůvodňuje. Tím by se měl čtenář naučit i správně odhadovat, zda nové úlohy, se kterými se v programátorské praxi setká, jsou efektivně řešitelné nebo beznadějně těžké.

Není nezbytné, aby čtenář porozuměl všem zde předneseným důkazům, neboť ty jsou často trikové a tudíž obtížné k pochopení. Mnohem důležitější je získat základní encyklopedický přehled o tom, které běžné problémy patří mezi $\mathcal{NP}$-úplné.

10.1 Splnitelnost formulí

V předchozí Lekci 9 jsme si na závěr uvedli existenci $\mathcal{NP}$-úplného problému, neboli problému ve třídě $\mathcal{NP}$, který je ze všech takových "nejtěžší možný". Uvedení bylo velmi neformální, ale nyní si alespoň upřesníme nezbytné definice logických formulí a problému jejich splnitelnosti.

`Značení`: *Logická formule* se skládá z boolovských proměnných (které nabývají hodnoty F/T), logických spojek $\wedge$ (a), $\vee$ (nebo), $\neg$ (negace) a závorek.

`Definice`: Logická formule Φ s proměnnými $x_1, x_2, \ldots, x_n$ je v *konjunktivním normálním tvaru* pokud je zapsaná jako

$$\Phi = c_1 \wedge c_2 \wedge \ldots \wedge c_k \,,$$

kde každé c_i se nazývá *klauzule* a představuje zkratku pro

$$c_i = (\ell_{i1} \vee \ell_{i2} \vee \ldots \vee \ell_{im_i})$$

a kde ℓ_{ij} je *literál* mající jednu ze dvou podob pro některé $p \in \{1, \ldots, n\}$

$$\ell_{ij} \equiv x_p \quad \text{nebo} \quad \ell_{ij} \equiv \neg x_p \,.$$

`Komentář:` Příklady logických formulí v konjunktivním normálním tvaru jsou:

$$\Phi_1 = (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4)$$

$$\Phi_2 = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_4 \vee \neg x_5) \wedge (x_2 \vee x_5)$$

U takovýchto formulí si budeme všímat, zda některé logické ohodnocení vstupních proměnných má za výsledek hodnotu T (pravda) celé formule. Pro Φ_1 je takové pravdivé ohodnocení například $x_1 = x_3 = x_4 = T$, $x_2 = F$, kdežto pro Φ_2 stačí třeba $x_2 = x_4 = T$.

Náš očekávaný problém splnitelnosti logických formulí je přesně definován takto:

Problém 10.1. ***SAT (splnitelnost logických formulí)***
Následující problém je $\mathcal{NP}$-úplný:

Vstup: Logická formule Φ v konjunktivním normálním tvaru.

Výstup: Existuje logické ohodnocení proměnných Φ tak, aby výsledná hodnota Φ byla T (pravda)?

`Důkaz:` Je snadno vidět, že problém SAT náleží do třídy $\mathcal{NP}$ – pro splnitelnou formuli Φ napovíme logické hodnoty proměnných a pak snadno Φ vyhodnotíme (jako T). Jeho $\mathcal{NP}$-úplnost vyplývá z důkazu Věty 9.4. □

`Komentář:` Zde je příklad splnitelné formule

$$\Phi_1 = (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4)$$

a příklad nesplnitelné formule

$$\Phi_2 = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge \ldots \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge \ldots \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

(v této formuli se objevují všechny možné kombinace rozmístění negací v klauzulích).

Všimněte si dobře, jak důležitou roli v popisu problému splnitelnosti SAT hraje předpoklad, že formule Φ je v konjunktivním normálním tvaru. Například pokud by Φ byla zapsána tak, že konjunkce by byly uvnitř závorek a disjunkce vně závorek (naopak), pak by řešení problému splnitelnosti Φ vyšlo zcela jednoduše.

Mimo problému SAT je $\mathcal{NP}$-úplná i jeho následující restriktivní verze, která se nám bude mnohem lépe hodit k dalším převodům.

Problém 10.2. ***3-SAT (splnitelnost logických formulí ve spec. verzi)***
Následující problém je také $\mathcal{NP}$-úplný:

Vstup: Logická formule Φ v konjunktivním normálním tvaru taková, že každá klauzule obsahuje nejvýše 3 literály.

Výstup: Existuje logické ohodnocení proměnných Φ tak, aby výsledná hodnota Φ byla T (pravda)?

`Důkaz:` Ukážeme polynomiální převod vstupu Φ' problému SAT na ekvivalentní vstup Φ problému 3-SAT: Každou klauzuli $c_i = (\ell_{i1} \vee \ell_{i2} \vee \ldots \vee \ell_{im_i})$ s m_i více než třemi literály v Φ' nahradíme novou proměnnou y_i a dvojicí klauzulí

$$\gamma_i = (\ell_{i1} \vee \ell_{i2} \vee y_i) \wedge (\ell_{i3} \vee \ldots \vee \ell_{im_i} \vee \neg y_i) \,.$$

Zřejmě je $c_i \equiv \gamma_i$ ve smyslu logické ekvivalence (s volnou proměnnou y_i), a proto se výsledek problému splnitelnosti s novou formulí nezmění. Navíc mají nové klauzule jen 3 a $m_i - 1$ literálů.

Popsanou náhradu ve Φ' provádíme opakovaně, dokud nemají všechny klauzule nejvýše 3 literály. Vstup Φ' problému SAT takto převedeme v lineárním počtu kroků na ekvivalentní vstup Φ problému 3-SAT. Podle Lematu 9.3 je proto i problém 3-SAT $\mathcal{NP}$-úplný. □

Úlohy k řešení

(10.1.1) *Vraťme se k poznámce, jak důležitou roli v popisu problému splnitelnosti SAT hraje předpoklad, že formule Φ je v konjunktivním normálním tvaru. Vezměme si, že by Φ' byla zapsána tak, že konjunkce by byly uvnitř závorek a disjunkce vně závorek (naopak než zde). S jakou časovou složitostí vzhledem k délce formule byste pak rozhodli, zda Φ' je splnitelná?*

***(10.1.2)** *Jistě víte, že mezi konjunkcí a disjunkcí lze "roznásobit" jako u běžných čísel: $a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)$. Podívejte se na řešení Úlohy 10.1.1 – pokud ve formuli Φ takto "roznásobíme" všechny disjunkce v závorkách, dostaneme formuli $\Phi' \equiv \Phi$ právě ve tvaru Úlohy 10.1.1. Proč tedy nelze takto efektivně řešit SAT "roznásobením"?*

10.2 Grafové problémy

Teorie grafů nám dává nepřeberné množství snadno popsatelných, ale obtížně řešitelných problémů. (Vzpomínáte na některé z výuky diskrétní matematiky?) Mnohé z nich přímo jsou $\mathcal{NP}$-úplné a my si jich několik ukážeme.

`Značení`: Nejprve si připomeneme některé grafové termíny (viz DIM [4]).

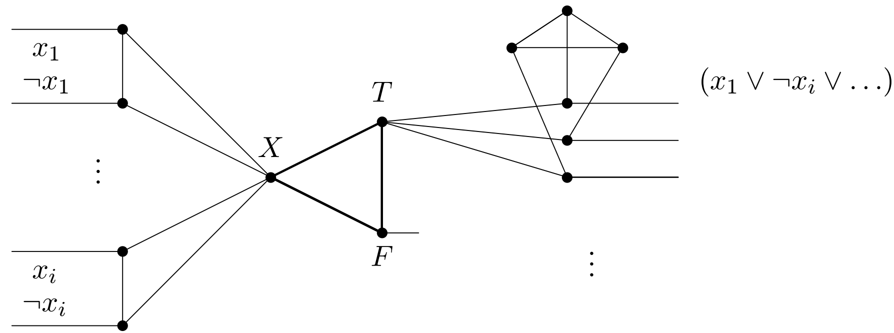
- Korektním obarvením grafu G k barvami rozumíme přiřazení jedné z k daných barev každému z vrcholů G tak, aby žádné dva vrcholy spojené hranou nedostali stejnou barvu.
- Nezávislou množinou I v grafu G rozumíme podmnožinu vrcholů $I \subseteq V(G)$ takovou, že žádné dva vrcholy z I nejsou v grafu G spojeny hranou.
- Hamiltonovským cyklem či kružnicí v grafu G nazýváme orientovanou či normální kružnici procházející všemi vrcholy v G.
- Vrcholovým pokrytím C v grafu G rozumíme podmnožinu vrcholů $C \subseteq V(G)$ takovou, že každá hrana grafu G má aspoň jeden konec v C.
- Dominující množinou D v grafu G rozumíme podmnožinu vrcholů $D \subseteq V(G)$ takovou, že každý jiný vrchol grafu G je sousedem některého vrcholu v D.

Pokud budou čtenáři následující důkazy připadat příliš obtížné, doporučujeme nejprve prostudovat jednodušší Cvičení 10.3 a pak se k následujícím důkazům vrátit.

Problém 10.3. ***3-COL (3-obarvení grafu)***
Následující problém je $\mathcal{NP}$-úplný:

Vstup: Graf G.
Výstup: Lze vrcholy G korektně obarvit 3 barvami?

`Důkaz` (náznak): Ukážeme si polynomiální převod z problému 3-SAT. Sestrojíme graf G pro danou formuli Φ. Základem grafu je trojúhelník, jehož vrcholy označíme X, T, F (přitom barvy přiřazené vrcholům T, F budou reprezentovat logické hodnoty). Každé proměnné x_i ve Φ přiřadíme dvojici vrcholů spojených s X. Každé klauzuli ve Φ přiřadíme podgraf na 6 vrcholech (z nichž tři jsou spojené s T), jako na obrázku. Nakonec volné "půlhrany" z obrázku pospojujeme dle toho, jaké literály vystupují v klauzulích.



Například první půlhrana naznačené klauzule na obrázku je napojena na půlhranu značenou x_1 vlevo, druhá půlhrana na půlhranu značenou $\neg x_i$ vlevo, atd. Pokud v klauzuli chybí třetí literál, je jeho půlhrana napojena přímo na F vpravo.

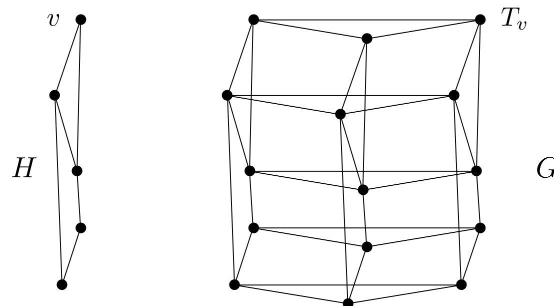
Pak G má 3-obarvení právě když je Φ splnitelná, jak si lze ověřit na obrázku. Tím jsme sestrojili polynomiální převod formule Φ z problému 3-SAT na graf G v problému 3-obarvení. $\mathcal{NP}$-úplnost nyní vyplývá z Lemat 9.3 a 10.2. □

Kromě barevnosti grafu je $\mathcal{NP}$-úplných mnoho problémů ptajících se na výběry vrcholů v grafu s jistými vlastnostmi.

Problém 10.4. ***IS (nezávislá množina)***
Následující problém je $\mathcal{NP}$-úplný:

Vstup: Graf G a přirozené číslo k.
Výstup: Lze v G najít nezávislou podmnožinu velikosti (aspoň) k?

`Důkaz`: Ukážeme polynomiální převod z problému 3-COL. Nechť H je graf na n vrcholech, který je za úkol obarvit třemi barvami. Položíme $k = n$ a graf G sestrojíme ze tří disjunktních kopií grafu H tak, že vždy tři kopie každého jednoho vrcholu $v \in V(H)$ spojíme hranami do trojúhelníku T_v v G.



Pokud $c : V(H) \to \{1, 2, 3\}$ je obarvení H třemi barvami, v grafu G lze vybrat $k = n$ nezávislých vrcholů tak, že pro každý $v \in V(H)$ vezmeme $c(v)$-tou kopii vrcholu v v grafu G. (Nakreslete si v obrázku!) Naopak pokud I je nezávislá množina v grafu G o velikosti $k = n$, pak z každého trojúhelníku T_v, $v \in V(H)$ náleží do I právě jeden vrchol. Podle toho již určíme jednu ze tří barev pro vrchol v v H. □

Problém 10.5. ***VC (vrcholové pokrytí)***
Následující problém je $\mathcal{NP}$-úplný:

Vstup: Graf G a přirozené číslo k.
Výstup: Lze v G najít vrcholové pokrytí velikosti nejvýše k?

Důkaz: Problém vrcholového pokrytí jasně patří do $\mathcal{NP}$ a jeho úplnost je dokázána polynomiálním převodem z Příkladu 9.5. □

Problém 10.6. ***DOM (dominující množina)***
Následující problém je $\mathcal{NP}$-úplný:

Vstup: Graf G a přirozené číslo k.
Výstup: Lze v G najít dominující množinu velikosti nejvýše k??

Důkaz: Problém dominující množiny jasně patří do $\mathcal{NP}$ a jeho úplnost je dokázána polynomiálním převodem z Příkladu 9.6. □

Další předvedené problému se týkají procházení grafem (pokud možno) bez opakování vrcholů.

Problém 10.7. ***HC (Hamiltonovský cyklus)***
Následující problém je $\mathcal{NP}$-úplný:

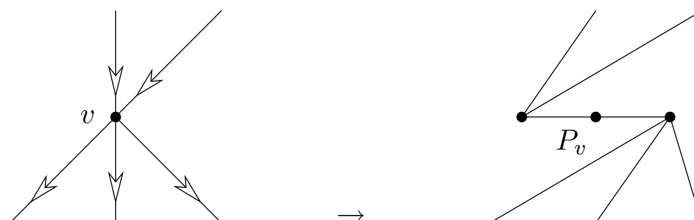
Vstup: Orientovaný graf G.
Výstup: Lze v G najít orientovanou kružnici (cyklus) procházející všemi vrcholy?

Důkaz (náznak): Tento převod je docela obtížný. Převádí se problém vrcholového pokrytí grafu G tak, že vrcholy i hrany v G se nahrazují speciálními malými orientovanými podgrafy, které zaručí následovné: Je přidáno k dodatečných vrcholů umožňujících "přeskočit" do libovolného vrcholového podgrafu (to budou ony vrcholy v pokrytí v G), z každého však lze skočit jen jednou podle definice Hamiltonovského cyklu. Pokud přeskočí kružnice do vrcholového podgrafu, může dále projít i všechny připojené hranové podgrafy (tím se všechny připojené hrany G "pokryjí") a "přeskočit" zase jinam. Takto Hamiltonovský cyklus v novém grafu přesně odpovídá vrcholovému pokrytí velikosti $\leq k$ v původním grafu G. □

Problém 10.8. ***HK (Hamiltonovská kružnice)***
Následující problém je $\mathcal{NP}$-úplný:

Vstup: Graf G.
Výstup: Lze v G najít kružnici procházející všemi vrcholy?

Důkaz:



Použijeme snadný převod z předchozího problému HC. Každý vrchol v orientovaného grafu H nahradíme třemi vrcholy tvořícími cestu P_v délky 2 v grafu G. Orientované hrany grafu H přicházející do v pak přivedeme do prvního vrcholu cesty P_v, hrany odcházející z v naopak vedeme z posledního vrcholu cesty P_v. □

Problém 10.9. ***TSP (problém obchodního cestujícího)***
Následující problém je $\mathcal{NP}$-úplný:

Vstup: Souvislý graf G s nezáporným ohodnocením hran ("délkou") a číslo r.

Výstup: Lze v G najít uzavřený sled procházející všemi vrcholy a mající součet délek hran (včetně opakovaných) nejvýše roven r?

Důkaz: Použijeme snadný převod z předchozího problému HK. Každou hranu ohodnotíme délkou 1 a položíme $r = n$, kde n je počet vrcholů našeho grafu. Pak uzavřený sled délky n se nesmí opakovat v žádném vrcholu ani hraně, aby prošel všemi vrcholy, a proto musí být kružnicí. □

Úlohy k řešení

(10.2.1) *Zdůvodněte, proč je následující problém ("orientovaná dominující množina") $\mathcal{NP}$-úplný: Vstupem je orientovaný graf G a přirozené číslo k. Lze v grafu G najít podmnožinu $D \subseteq V(G)$ s nejvýše k vrcholy takovou, že každý vrchol w grafu G náleží do D nebo do w vede orientovaná hrana (šipka) z některého vrcholu v D?*

Návod: Použijte třeba polynomiální převod z problému vrcholového pokrytí.

10.3 Aritmetické problémy

Také mnohé aritmetické problémy se dají jen velmi obtížně řešit. (Vzpomeňme si, že délka vstupu obvyklých číselných problémů se měří počtem bitů potřebných k zápisu vstupních čísel, která tak mohou být obrovská.)

Problém 10.10. ***PART (problém loupežníka či batohu)***
Následující problém je $\mathcal{NP}$-úplný:

Vstup: Množina M přirozených čísel.

Výstup: Lze rozložit M na dvě podmnožiny $M = M_1 \cup M_2$, $M_1 \cap M_2 = \emptyset$ tak, aby součet čísel v M_1 byl rovný součtu v M_2?

Důkaz (náznak): Problém zřejmě patří do třídy $\mathcal{NP}$, vhodnou nápovědou je hledané rozložení na dvě podmnožiny, přičemž jejich součty lze snadno porovnat. Naopak existuje polynomiální převod z problému 3-SAT, který modeluje jednotlivé proměnné a klauzule pomocí oddělených úseků bitů v zápise dlouhých binárních čísel.
Každá proměnná formule Φ má dvě čísla, která musí patřit do různých částí, jejich přehození mění logickou hodnotu proměnné. Klauzulím pak odpovídají jen společné úseky v bitových zápisech čísel, kde bity 1 indikují výskyty proměnných jako literálů.

			c_1	c_2	...
x_1 :	1000	...	0001	0000	
$\neg x_1$:	1000	...	0000	0001	
x_2 :	0100	...	0000	0001	
$\neg x_2$:	0100	...	0001	0000	
...					
F :	0000	0000	0001	0001	0001
$* \times 2$ adj :	0000	0000	0001	$*$	$*$

Navíc je přidáno jedno speciální číslo F s bity 1 na místech všech klauzulí naší formule, které pak v rozdělení na dvě podmnožiny označuje tu část odpovídající logické hodnotě False proměnných (tj. literálů). Pro každou klauzuli jsou nakonec přidána dvě "úpravová" čísla schematicky zaznačená *adj* (jen jedno *adj* pro klauzule se dvěma litarály), která mají obě hodnotu bitu 1 jedině na místě jeho klauzule.

Promyslete si sami, proč rozdělení na dvě části se stejným součtem nutně znamená rozdělení hodnot proměnných dané formule, při kterém každá klauzule má alespoň jeden pravdivý literál, tedy ale □

Problém 10.11. ***IP (celočíselná lineární optimalizace v rozhod. verzi)***
Následující problém je $\mathcal{NP}$-úplný:

Vstup: Matice $\boldsymbol{A}$ a vektor $\vec{b}$ určující soustavu lineárních nerovnic s proměnnými z_i (ve vektorovém zápise)

$$\boldsymbol{A} \cdot \vec{z} \leq \vec{b}\,.$$

Výstup: Existuje vektor $\vec{z} \in \{0,1\}^*$, tj. ohodnocení proměnných $z_i \in \{0,1\}$ takové, že soustava nerovnic $\boldsymbol{A} \cdot \vec{z} \leq \vec{b}$ je splněná?

Důkaz: Problém opět patří do $\mathcal{NP}$. Vezměme množinu $M = \{m_1, m_2, \ldots, m_k\}$ přirozených čísel, která je vstupem problému "loupežníka". Nechť $m = m_1 + m_2 + \ldots + m_k$ je celkový součet. Sestavíme soustavu dvou nerovnic:

$$m_1 z_1 + m_2 z_2 + \ldots + m_k z_k \leq m/2$$

$$m_1(1 - z_1) + m_2(1 - z_2) + \ldots + m_k(1 - z_k) \leq m/2$$

Hodnota $z_i = 1$ nám říká, že číslo m_i dáme do první množiny hledaného rozkladu, kdežto $1 - z_i = 1$ znamená, že m_i dáme do druhé množiny. Proto platné řešení výše uvedené soustavy poskytne rozdělení čísel z M na dvě podmnožiny tak, že v žádné není více než polovina celkového součtu. To znamená, že čísla z M jsou rozdělena "přesně napůl". Problém loupežníka jsme tímto převedli na problém IP. □

Úlohy k řešení

(10.3.1) *Proč je $\mathcal{NP}$-úplný problém MIP – smíšené celočíselné optimalizace, který je definovaný obdobně jako IP 10.11, ale některé vybrané proměnné mohou nabývat libovolných reálných hodnot?*

10.4 Cvičení: Další $\mathcal{NP}$-úplné problémy a převody

Komentář: Nejprve si uveďme stručný neformální návod, jak by vlastně běžný polynomiální převod pro zdůvodnění $\mathcal{NP}$-úplnosti problému měl vypadat...
Předpokládejme, že o problému P již víme, že je $\mathcal{NP}$-úplný, a o problému Q to chceme dokázat. Neboli chceme ukázat, že každý vstup problému P bychom uměli rozřešit převodem na případ problému Q, a tudíž i problém Q musí být tak těžký jako P.

Takže v jednoduchých případech stačí vzít obecný (libovolný) vstup $\mathcal{V}$ známého těžkého problému P a "přeložit" jej na vstup $\mathcal{U}$ pro problém Q tak, že Q odpoví ANO na nový vstup $\mathcal{U}$ právě tehdy, když P odpověděl ANO na $\mathcal{V}$. (Všimněte si dobře, že musíte dokázat logickou ekvivalenci, tedy že z odpovědi ANO v Q vyplývá ANO v P i naopak!)

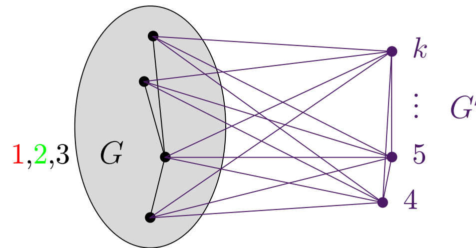
Příklad 10.12. *Problém k-obarvení grafu se ptá, zda daný graf lze obarvit k barvami tak, aby žádná hrana nespojovala dva vrcholy stejné barvy. (Skutečná barevnost našeho grafu může být i menší než k, o to v problému nejde.) Dokažme, že problém k-obarvení je $\mathcal{NP}$-úplný pro každé (i fixní) $k \geq 3$.*

Nejprve musíme zdůvodnit, že problém patří do třídy $\mathcal{NP}$. To je snadné, neboť nápovědou nám je ono obarvení grafu G pomocí k barev. Napověděné obarvení snadno zkontrolujeme v počtu kroků úměrném počtu hran grafu G, tedy polynomiálně v počtu vrcholů bez ohledu na hodnotu k.

Na druhou stranu už víme z Problému 10.3, že 3-obarvení grafu je $\mathcal{NP}$-úplné. Stačí nám tedy nalézt polynomiální převod z problému 3-obarvení na problém k-obarvení

grafu. Předpokládejme tedy, že $k > 3$ a že je dán graf G, o kterém se ptáme, zda jej lze obarvit 3 barvami. My sestrojíme graf G' přidáním $k-3$ nových vrcholů ke grafu G spojených každý se všemi ostatními vrcholy. Proč to děláme?

To je zřejmé, v grafu G' všechny nově přidané vrcholy musí mít různých barvu od všech ostatních, takže pokud původní graf G šel obarvit 3 barvami, půjde tak i graf G' obarvit k barvami. Naopak pokud G' je obarven k barvami, všechny barvy nových $k-3$ vrcholů jsou jiné a různé od barev všech původních vrcholů, takže na původní vrcholy G nám zbudou jen $k-(k-3) = 3$ různé barvy, což je přesně problém 3-obarvení na G. (Neboli 3-obarvení G jednoznačně odpovídají k-obarvením G'.) Schematickým obrázkem:



Vidíme tedy, že jsme našli polynomiální převod ze 3-obarvení grafu G na k-obarvení grafu G', a proto je problém k-obarvení $\mathcal{NP}$-těžký. Celkem dostáváme, že k-obarvení je $\mathcal{NP}$-úplné. □

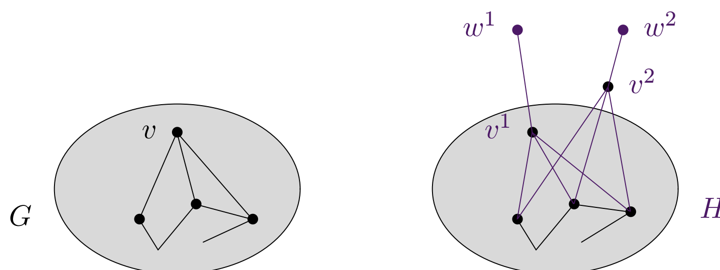
Příklad 10.13. *Hamiltonovská cesta v grafu je takový podgraf, který je isomorfní cestě a prochází všemi vrcholy grafu. (Obdoba Hamiltonovské kružnice.) Dokažme, že problém zjištění existence Hamiltonovské cesty v daném grafu G je $\mathcal{NP}$-úplný.*

Již víme z Problému 10.8, zjištění existence Hamiltonovské kružnice je $\mathcal{NP}$-úplné. Problém Hamiltonovské cesty taktéž náleží do $\mathcal{NP}$, snadnou nápovědou existence je ukázat onu Hamiltonovskou cestu. Pro důkaz $\mathcal{NP}$-úplnosti využijeme polynomiální převod Hamiltonovské kružnice na Hamiltonovskou cestu.

Názorně řečeno, potřebujeme převést daný graf G na jiný graf H tak, že Hamiltonovská kružnice v G se stane Hamiltonovskou cestou v H a naopak. Na první pohled by se toto zdálo jako snadný cíl – přece z každé Hamiltonovské kružnice uděláme cestu odebráním hrany či vrcholu. Velký problém je však v onom slůvku "naopak", my také musíme zajistit, že každá Hamiltonovská cesta v H vytvoří Hamiltonovskou kružnici v G! Proto nějakým způsobem musíme fixovat počátek a konec Hamiltonovské cesty v H tak, aby se daly spojit do kružnice v G.

Jednou z možností je vybrat jakýkoliv vrchol v v G, "zdvojit" jej (tj. přidat další vrchol se stejnými sousedy jako v) a navíc ke každé v^i ze zdvojených kopií v přidat novou hranu vedoucí do nového vrcholu w^i stupně 1. Tyto přidané vrcholy w^1, w^2 pak nutně musí být konci Hamiltonovské cesty, pokud ona existuje (jinak se do vrcholů stupně 1 přece dostat nedá).

Schematickým obrázkem:



□

Příklad 10.14. *Pro jaké k je $\mathcal{NP}$-úplný problém zjistit, zda v daném grafu existuje nezávislá množina velikosti k? (Nezávislá množina je taková podmnožina vrcholů grafu, z níž žádné dva její vrcholy nejsou spojené hranou.) Je tento problém $\mathcal{NP}$-úplný pro fixní k nebo pro proměnné hodnoty k?*

Tento příklad uvádíme proto, aby si čtenář dobře uvědomil roli *parametrů problému*, které jsou fixní, a těch, které jsou proměnlivé, neboli dané na vstupu.

Problém existence nezávislé množiny velikosti k totiž snadno rozřešíme v čase $O(n^{k+1})$ – prostě projdeme hrubou silou všechny k-tice vrcholů grafu a pokaždé se podíváme, zda náhodou netvoří nezávislou množinu. Čas $O(n^{k+1})$ je pochopitelně polynomiální pro každé fixní k, takže pak tento problém těžko může být $\mathcal{NP}$-úplný. Naopak pro k na vstupu problému se jedná o $\mathcal{NP}$-úplný problém podle našeho Tvrzení 10.4. □

Úlohy k řešení

(10.4.1) *Ukažte, že také následující problém tzv. "kubické kostry" je $\mathcal{NP}$-úplný: Vstupem je jednoduchý souvislý neorientovaný graf G. Otázkou je, zde lze v grafu G najít takovou kostru, jejíž všechny vrcholy jsou stupně nejvýše 3? (Stupně se samozřejmě myslí v té kostře, ne v G.)*
Jedná se vlastně o obdobu Hamiltonovské cesty, která je kostrou, jejíž všechny vrcholy jsou stupně nejvýše 2.

Návod: Použijte (třeba) převod z problému Hamiltonovské cesty.

***(10.4.2)** *Ukažte, proč je následující problém $\mathcal{NP}$-úplný: Vstupem je jednoduchý neorientovaný graf G. Ptáme se, zda lze v grafu G najít uzavřený tah procházející všemi vrcholy takový, že nejvýše jednou projdeme znovu vrcholem, kterým jsme již prošli dříve?*
Pro osvětlení – u Hamiltonovské kružnice jde o uzavřený tah, který žádný vrchol nezopakuje, kdežto v našem případě je dovoleno tahem zopakovat nejvýše jednou jeden vrchol.

Návod: Použijte třeba převod z problému Hamiltonovské kružnice.

(10.4.3) *Sice už víme, že jak Hamiltonovská kružnice, tak i Hamiltonovská cesta jsou $\mathcal{NP}$-úplné, ale zkuste cvičně najít polynomiální převod Hamiltonovské cesty na Hamiltonovskou kružnici (naopak než v Příkladu 10.13).*

Část III

Klíč k řešení úloh

(1.3.1) Slova ε, ‘10’ a celé slovo ‘101110110’.

(1.3.2) $\{11001, 110000, 011101, 0111000\}$

(1.3.3) Třeba $L_1 = \{\varepsilon\}$ a $L_2 = \{1\}$.

(1.3.4)* Ne, jen všechna ta slova, co nekončí lichým počtem ‘0’.

(1.4.1) $\{0, 001, 00101, 0010101, 111, 11101, 1110101\}$

(1.4.2) Je to jazyk všech těch slov, která mají úseky nul sudé délky a úseky jedniček délky dělitelné třemi.

(1.4.3) Slova ze samých nul nebo ta slova, která mají jediný znak ‘1’ právě uprostřed, tj. ε,‘0’,‘00’,‘000’,...,‘1’,‘010’,‘00100’,...

(1.4.4) 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 ‘$111' \in L_2 \cdot L_3$.

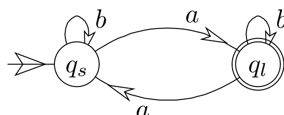
(1.4.5) ...

(1.4.6)* 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.

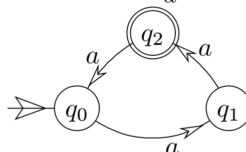
(1.4.7)* Nejméně 12, zdůvodněte si, proč ne méně! Pro které dva jednoduché jazyky se minima dosahuje?

(1.4.8)* Protože 3 dvoupolohové přepínače se mohou nacházet jen v $2^3 = 8$ celkem kombinacích poloh.

(2.1.1) Čtení znaku b nemění stav:



(2.1.2) Počítání na cyklu délky 3:

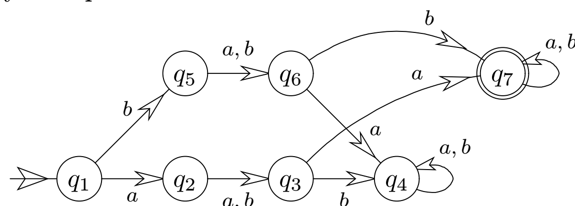


(2.1.3) Právě taková, ve kterých počet výskytů znaku a mínus počet b dává zbytek 2 po dělení 3.

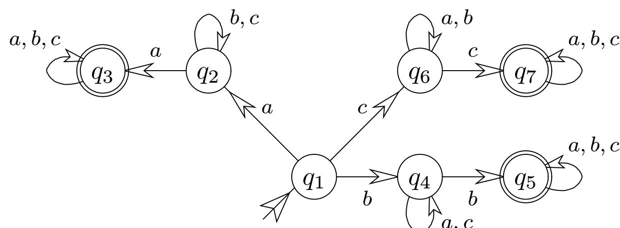
(2.1.4) Všechna taková, ve kterých po prvním výskytu znaku a následuje sufix ‘a’, ‘aa’ nebo ‘b’ (a nic víc).

(2.1.5) Jen ten vpravo, neboť automat vlevo se dostává do přijímajícího stavu jen když délka dává zbytek 2 po dělení 3.

(2.1.6)



(2.1.7)



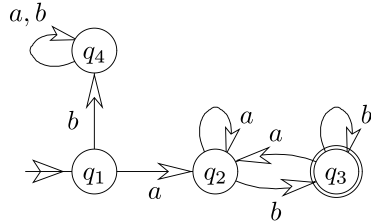
(2.1.8) Velmi jednoduše – vyměníme přijímající stavy F za jejich doplněk $Q \setminus F$.

(2.2.1) Ano, počítáme paritu výskytů pro a i b zvlášť podle Příkladu 2.2 a uděláme sjednocení

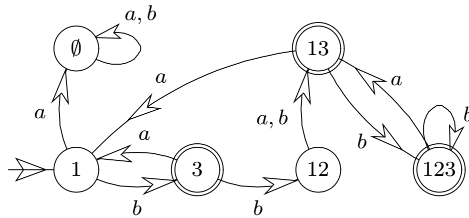
těchto dvou jazyků.

(2.2.2) Ano, obdobně jako v předchozí úloze, jen pozměníme přijímající stavy. Nakreslete si to a ověřte!

(2.2.3) 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.

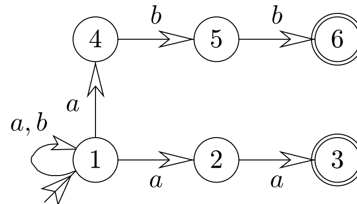
(2.2.4) 

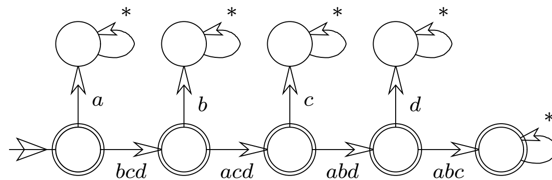
(2.3.1) Nikdy, již první přechod znakem a není definován.

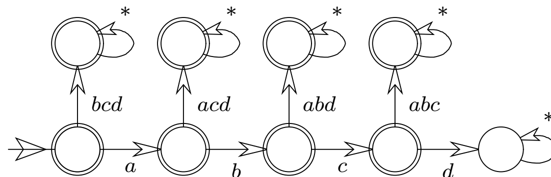
(2.3.2) Zde: 

(2.3.3) Vždy kromě ε a 'bb', viz sestrojený deterministický automat.

(2.3.4)* 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.

(2.3.5) Přirozeně využijeme nedeterminismu: 

(2.4.1) Zde: 

(2.4.2) Podobně zde: 

(2.4.3) Nejmenší takový KA má a) 8, b) 7 stavů.

(2.4.4) Nejmenší takový KA má 5 stavů.

(2.4.5) Třeba bab

(2.4.6) Třeba abb

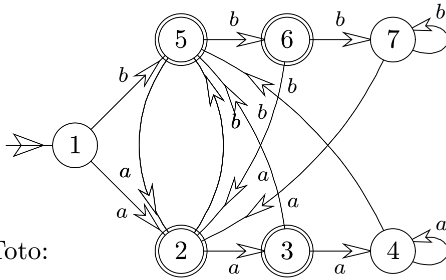
(2.4.7) 5 stavů

(2.4.8) 6 stavů

(2.4.9)* Slova začínají b, končí ba, opakují se $\leq 2\times$ a za sebou.

(3.1.1) V podstatě ano, jen počáteční stav u formálně sestrojeného automatu je navíc.

(3.1.2) Toto:



(3.1.3) 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.

(3.3.1) Třeba takto $a^*(c+\varepsilon)b^*$.

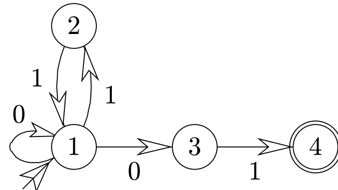
(3.3.2) Třeba takto $aa^*(c+\varepsilon)b^*b$.

(3.3.3) Třeba takto $aa^*(cc+c+\varepsilon)b^*b$.

(3.3.4) Neplatí, třeba levý jazyk obsahuje prázdné slovo, kdežto pravý ne.

(3.3.5) Neplatí, třeba slova v levém jazyku mohou končit znakem 1, kdežto v pravém jazyku ne.

(3.4.1) Zde:



(3.4.2) Jen přidáme smyčku s 0 nad stav 3.

(3.4.3) Třeba $0^*1(1+00+01)^*$.

(3.4.4)* Třeba $(a+b)(a+b)\big(((a+b)(a+b)+b)(a+b)\big)^*$. Už to není tak jednoduché, že?

(3.5.1) $(\varepsilon+1+11)(01+011+001+0011)^*(\varepsilon+0+00)$

(3.5.2) $\big(((b+c)^*+(a+c)^*)c\big)^*(b^*+a^*)$

(3.5.3) $\big((b+c+a(b+c)\big)^*(\varepsilon+a)$

(3.5.4) c^*

(3.5.5) $c^*(abb^*+b^*)^*$

(3.5.6)* $\big((c+\varepsilon)(a+b)(a+b)^*\big)^*(c+\varepsilon)\,aa\,\big((c+\varepsilon)(a+b)(a+b)^*\big)^*(c+\varepsilon)$

(3.5.7)* …

(3.5.8)* a) Nejkratší je ε a nejdelší 01100110, neboť jazyk L nedovoluje opakovat stejný znak za sebou více než dvakrát.
b) Protože $1 \in K \setminus L$ a $010101 \in L \setminus K$.
c) 10101 jednoznačně.

(3.5.9) $(\varepsilon+aa+ba)a^*(abaa^*)^*$

(4.1.1) Sloučí se stavy $1, 3$ do jednoho.

(4.1.2) Ano, postupem minimalizace začneme s rozkladem $\{\{1,3\},\{2\}\}$ a hned v první iteraci rozlišíme stavy $1, 3$ přechodem znakem 0.

(4.1.3) Ano, již po dvou iteracích dojde k rozlišení všech stavů.

(4.1.4) 4 stavy, je nutno počítat paritu počtů jak a, tak b.

(4.2.1) Není, stačí se podle Pumping lematu podívat na slovo 'a^nb^n' jako zřetězení uvw, kde v může být složeno jen ze samých a, a proto už uv^2w nepatří do našeho jazyka.

(4.3.1) Není, neboť prefixy a^i pro různá i patří jasně do různých tříd pravé kongruence jazyka, takže těch je nekonečně mnoho (Věta 4.10).

(4.3.2) Není, neboť pokud by byl, byl by regulární i jazyk těch slov, kde je rozdíl počtů b a a nezáporný. Tudíž by byl regulární i průnik těchto jazyků (Věta 2.7), což je jazyk se stejně a jako b, který není regulární podle předchozí úlohy.

(4.3.3) Není, odpověď je obdobně zdůvodněná jako v předchozích příkladech.

(4.4.1) Samozřejmě ne, ten první nepřijímá prázdné slovo, kdežto druhý ano.

(4.4.2) Ano, ten druhý se minimalizuje na stejný jako první.

(4.4.3) Již po dvou iteracích postupu minimaliace dojde k rozložení na jednotlivé stavy zvlášť.

(4.4.4) Spojit 12; 348; 567.

(4.4.5) Spojit 12; 348; 5; 67.

(4.4.6) 5 stavů, 2×2 jsou nutné k rozpoznávání parity počtů a a b, jeden další pro odlišení prázdného slova.

(4.4.7) 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 7 stavů.

(4.4.8) Obdobně předchozímu 5 stavů.

(4.4.9) Obdobně předchozímu 9 stavů.

(4.4.10) Je, stačí automatem počítat prvních 100 znaků slova a po nich už je všechno přijato.

(4.4.11) Opět je, ale automat už vyjde dosti veliký.

(4.4.12)* Není, opět použijeme verzi argumentu s "počítáním" znaků.

(4.4.13)* Není, zde je nejlepší pumping lemma: slovo $a^p \in L$ pro p prvočíslo rozepíšeme jako $a^p = a^k a^m a^n$ a potom má být $a^k a^{(k+n)m} a^n = a^{(k+n)(m+1)} \in L$, ale ten exponent není prvočíslo.

(5.1.1) Přidáme symbol G pro nejvyšší prioritu:
$E \longrightarrow E + E \mid F$, $\quad F \longrightarrow F \times F \mid G$, $\quad G \longrightarrow (E) \mid G^2 \mid a$

(5.1.2) 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.

(5.2.1) Chybí v něm slova liché délky, takže obecněji všechna $S \longrightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb$.

(5.2.2) $S \longrightarrow \varepsilon \mid 0S0 \mid T$, $\quad T \longrightarrow \varepsilon \mid 1T$.

(5.2.3) Prázdný, protože se nikdy nezbavíme výskytu neterminálu S nebo C.

(5.2.4) Ne, druhá generuje třeba slovo '*abba*', které v první nelze.

(5.3.1) 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 5.12, ale L je přece jazyk z Příkladu 5.11.

(5.4.1) Regulární A i B, bezkontextový C.

(5.4.2)* Regulární A, bezkontextový B, ani jedno pro C.

(5.4.3) Snadno $S \longrightarrow aSb \mid Sb \mid b$.

(5.4.4) $S \rightarrow aaSaa \mid abSba \mid baSab \mid bbSbb \mid \varepsilon$

(5.4.5)* $S \rightarrow aTa|bTb|\varepsilon$, $T \rightarrow aUa|bUb|a|b$, $U \rightarrow aSa|bSb$

(5.4.6) Negeneruje, neboť z druhé gramatiky nezískáme slovo ε.

(5.4.7) Negeneruje, neboť z druhé gramatiky nezískáme slovo *aaaabb*.

(5.4.8) B

(5.4.9) B

(5.4.10) A,B

(5.4.11)** ???

(5.4.12)* Třeba $S \longrightarrow bS \mid cS \mid TFS \mid T \mid \varepsilon$; $T \longrightarrow aTbb \mid abb$; $F \longrightarrow c \mid T$.

(6.1.1) (c)

(6.1.2) $\sqrt{n} = \Omega(\log n)$, přesněji roste striktně rychleji.

(6.1.3)* Druhá, neboť $(\log n)^{\log n} = n^{\log \log n} \gg n^5$.

(6.1.4)* (a)

(6.1.5)* ...

(6.2.1) Podle Lematu 6.2 je $T(n) = O(n)$.

(6.2.2) Podle Lematu 6.3 je $T(n) = O(n \log n)$.

(6.2.3) Podle Lematu 6.4 je $T(n) = O(n^2)$.

(6.2.4) Podle Lematu 6.4 je $T(n) = O(n^2 \log n)$.

(6.4.1) A i B.

(6.4.2) Jen A.

(6.4.3) Ani jeden.

(6.4.4) $C \prec A \prec B$

(6.4.5) $B \prec A \prec C$

(6.4.6) $A \prec C \prec B$

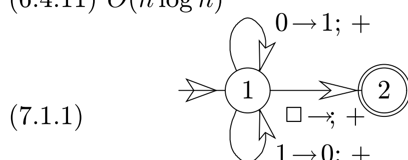
(6.4.7) $C \prec A \prec B$

(6.4.8) $O(n)$

(6.4.9) $O(n^2)$

(6.4.10) $O(n)$

(6.4.11) $O(n \log n)$

(7.1.1) 

(7.1.2) Přidáme jako počáteční nový stav, který se po znacích $0, 1$ posouvá doprava (bez přepisování) a na prvním prázdném vpravo se vrátí o 1 doleva a už pokračuje jako v příkladě.

(7.1.3) ...

(7.3.1) Nelze, neboť by to znamenalo přinejmenším vyřešit i problém zastavení daného programu. Proto lze správnost programu jen odhadovat.

(7.3.2)* Opět nelze, neboť běh výpočtu TS lze v tabulkovém programu takovými výpočty simulovat a jeho zastavení odpovídá přesně nezacyklení téhle simulace.

(7.4.1) 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.

(7.4.2) Využije se podobná implementace jako v Příkladu 7.9. Pokud slovo není palindrom, stroj se nezastaví, nýbrž skončí v nekonečné smyčce na jednom pomocném stavu.

(7.4.3) Zastaví na všech slovech kromě ε. Poslední znak počátečního úseku samých a (pokud tam je) je změněn na b.

(7.4.4)* ...?

(7.4.5)* Zastaví jen na $(a + b)a^*$, přepisuje na ab^*.

(7.4.6)* ...

(7.4.7) ...

(7.4.8) 2 kroky pro w zač. a, jinak počet b plus $3\times$ počet a plus 1.

(7.4.9) Délka plus 1 pro w bez b, jinak počet a na začátku krát 2 plus 2.

(8.1.1) 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 $\Theta(k)$.

(8.1.2) 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.)

(8.1.3)* Vzpomeňte si, že rovinný graf má nejvýše $3n-6$ hran, takže délka vstupu stačí vždy $O(n)$.

(8.2.1) Primitivní algoritmy třídění pracují sice va čase $O(n^2)$, ale ty chytřejší, jako třeba merge-sort nebo heap-sort už běží v čase $O(n \log n)$, takže to je hledaná časová složitost problému třídění. (Později si ukážeme, že ani rychleji třídit nelze.)

(8.2.2) $\Theta(n^2)$, bez ohledu na počet hran, neboť musíme projít všechna políčka matice.

(8.2.3) Teď již jen $\Theta(n)$, neboť stačí projít n vrcholů a porovnat jejich stupně, tj. délky seznamů sousedů.

(8.2.4)* Běžný algoritmus násobení každého řádku každým sloupcem trvá $\Theta(n^3)$, ale jsou i rychlejší algoritmy, třeba Strassenův pracující v čase zhruba $O(n^{2.8})$. Optimální algoritmus se nezná.

(8.3.1) 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?

(8.3.2) 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.

(8.3.3)* Snadno v čase $O(n \log n)$ – všechny body seřadíme podle x-souřadnice a pak po řadě zleva doprava vybíráme ty, co jsou “nejníže” a “nejvýše”. Chytřejší algoritmus by to však zvládl i v čase $O(n)$.

(8.3.4)* 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 8.9.

(8.4.1) (x, y) převedeme na vstup $(x, -y)$ pro sčítání.

(8.4.2) 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.

(8.4.3) ...

(8.5.1) $1^2 + 2^2 + 3^2 + \ldots + n^2 = \Theta(n^3)$

(8.5.2) $\Theta(\log n)$

(8.5.3)* $\Theta(\sqrt{n})$

(8.5.4) Šikovně to lze lineárně $O(n)$.

(8.5.5)* Zde využijeme Lema 6.4, 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 $T(n) = n^{\log_2 7} \doteq \Theta(n^{2.8})$.

(8.5.6)* Rekurentní vzorec přepíšeme pro náhradní funkci $T'(n) = T(n + 2)$:

$$T'(n) \leq 2T(k) + T(k + 1) + O(n) \leq 3T(k + 1) + O(n) =$$

$$= 3T(\frac{n+2}{2} + 1) + O(n) = 3T(\frac{n}{2} + 2) + O(n) = 3T'(\frac{n}{2}) + O(n)$$

(8.5.7)* 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 Lematu 6.2.

(9.1.1) Protože jednoduše napovíme (uhodneme) onu podmnožinu k vybraných vrcholů a efektivně ji zkontrolujeme.

(9.1.2) Protože napovíme, kterých $< k$ vrcholů vypustit, aby graf zbyl nesouvislý. Nesouvislost zbytku už pak snadno ověříme.

(9.1.3)* 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.

(9.1.4)* 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).

(9.3.1) 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).

(9.3.2) Podle Lematu 9.3 a převodu z předchozí úlohy víme, že 4-obarvení je $\mathcal{NP}$-těžké. Zároveň existence 4-obarvení snadno patří do třídy $\mathcal{NP}$.

(9.3.3)* Jednak už víme z přednášky DIM, že 2-obarvení grafu je efektivně řešitelné. Za druhé ta popsaná redukce 2-obarvení na 3-obarvení jde špatným směrem – pro důkaz $\mathcal{NP}$-úplnosti by musel být převod z 3-obarvení na 2-obarvení!

(9.4.1) A,B,C,F

(9.4.2) Je to stejný převod, neboť to funguje v obou směrech.

(9.4.3) 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.

(9.4.4)* ...?

(9.4.5) Ano, patří, tyto kružnice napovíme a snadno ověříme.

(9.4.6) Ani nemůže patřit, neboť se nejedná o rozhodovací problém!

(9.4.7) Obojí patří: Pro ověření toho, že graf je rovinný, stačí zkontrolovat napověděné 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.

(9.4.8)* Nepatří, neboť nijak polynomiálně neověříme, že graf G neobsahuje více než jednu Hamiltonovskou kružnici. Ani v případě negace problému nejsme schopni ověřit případ, kdy G neobsahuje žádnou Hamiltonovskou kružnici.

(9.4.9)* Pro $k = 0, 1, 2$ lze barevnost efektivně určit, takže tam otázka patří přímo do třídy $\mathcal{P}$. Také pro $k = 3$ patří problém barevnosti k do $\mathcal{NP}$, neboť napověděné 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 $\mathcal{NP}$ nepatří, protože neumíme nijak efektivně prokázat, že graf G nelze obarvit méně než k barvami.

(9.4.10) A,D

(10.1.1) S lineární složitostí – stačilo by najít závorku, ve které konjunkce neobsahuje zároveň x_i i $\neg x_i$, tou by pak v celé disjunkci byla splněná celá formule Φ'. Například $(x_1 \wedge x_2) \vee (\neg x_1 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \neg x_2)$ je splnitelná, kdežto $(x_1 \wedge \neg x_1) \vee (\neg x_2 \wedge x_2)$ není.

(10.1.2)* Zkuste si nějakou SAT formuli Φ skutečně roznásobit – vyjde vám exponenciálně mnoho členů ve výsledku, takže ten čas výpočtu nakonec stejně vyjde exponenciální!

(10.2.1) Je to kupodivu ještě jednodušší než v Tvrzení 10.6. Prostě v převodu z vrcholového pokrytí na grafu G vytvoříme orientovaný graf, jehož vrcholy jsou vrcholy G a také středy hran G. Šipky vedou vždy od vrcholů do středů přilehlých hran.

(10.3.1) Jelikož IP je speciální verzí MIP, bude MIP aspoň tak těžké, tj. NP-těžké. Naopak MIP náleží do NP, neboť lze napovědět a zkontrolovat přípustné řešení. (Je ale třeba zdůvodňovat, proč stačí reálné hodnoty proměnných napovědět s rozumnou přesností.)

(10.4.1) V převodu připojte ke každému vrcholu nový vrchol stupně 1.

(10.4.2)* Stačí G vytvořit tak, že k danému grafu připojíme jedním vrcholem w jeden nový trojúhelník. Právě vrchol w se pak v tahu bude muset zopakovat.

(10.4.3) Přidejte do grafu nový vrchol x spojený se vším – pak H. kružnice v novém musí procházet x a po odebrání x z ní zbude H. cesta.

Reference

[1] Cygwin, Linux-like development environment for Windows,
http://www.cygwin.com/.

[2] Flex, a fast lexical analyser generator,
http://www.gnu.org/software/flex/.
Český úvod http://atrey.karlin.mff.cuni.cz/~0rfelyus/bifle/flex.html.

[3] Grep, searching for lines containing a match to a specified pattern,
http://www.gnu.org/software/grep/.
Česky psané http://unix.felk.cvut.cz/unix/download/predn_05.pdf.

[4] P. Hliněný, Diskrétní Matematika, výukový text FEI VŠB (2004),
http://www.cs.vsb.cz/hlineny/vyuka/DIM-slides/.

[5] P. Hliněný, Web stránky předmětu UTI (2005),
http://www.cs.vsb.cz/hlineny/vyuka/UTI.html.

[6] J. Hopcroft, J. Ullman, Formálne jazyky a automaty, Alfa Bratislava, 1978.

[7] M. Chytil, Automaty a Gramatiky, SNTL, Praha, 1984.

[8] Petr Jančar, Teoretická informatika, výukový text FEI VŠB (2004),
http://www.cs.vsb.cz/jancar/TEORET-INF/teoret-inf.htm.

[9] L. Kučera, Kombinatorické Algoritmy, SNTL, Praha, 1983.

[10] L. Kučera, Algovize,
http://kam.mff.cuni.cz/~ludek/Algovision/Algovision.html.

[11] Regular expressions tutorial,
http://www.regular-expressions.info/.
Česky psané cvičení http://www.regexp.cz/.