ZÁKLADY TEORETICKÉ INFORMATIKY

URČENO PRO VZDĚLÁVÁNÍ V AKREDITOVANÝCH STUDIJNÍCH PROGRAMECH

HASHIM HABIBALLA

ČÍSLO OPERAČNÍHO PROGRAMU: CZ.1.07
NÁZEV OPERAČNÍHO PROGRAMU:
VZDĚLÁVÁNÍ PRO KONKURENCESCHOPNOST
OPATŘENÍ: 7.2
ČÍSLO OBLASTI PODPORY: 7.2.2

INOVACE VÝUKY INFORMATICKÝCH PŘEDMĚTŮ VE STUDIJNÍCH PROGRAMECH OSTRAVSKÉ UNIVERZITY

REGISTRAČNÍ ČÍSLO PROJEKTU: CZ.1.07/2.2.00/28.0245

OSTRAVA 2021

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky

Recenzent: Doc. RNDr. PaedDr. Eva Volná, PhD.

Název: Základy teoretické informatiky
Autor: Doc. RNDr. PaedDr. Hashim Habiballa, Ph.D.
Vydání: třetí, 2021
Počet stran: 216

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

OBSAH

1. Teoretická informatika a její disciplíny

Cíl:

Získáte základní přehled o těchto otázkách:

- co je teoretická informatika a jaké jsou její disciplíny
- jaký je účel těchto disciplín

Vymezení toho, co je teoretická informatika může být složitý úkol. Pokud budeme studovat různou literaturu, je možné zařazovat do této oblasti i témata a teorie, které spíše spadají do technických disciplín jako je kybernetika. Například knihy o kybernetice, se kromě specificky kybernetických témat dotýkají i některých oblastí teorie automatů a teorie jazyků, které jsou typické pro teoretickou informatiku. Stejný fenomén se pak týká i umělé inteligence (UI), která je dnes samostatnou a rozsáhlou disciplínou. Jako hlavní disciplíny teoretické informatiky lze uvést:

1. *Teorii formálních jazyků a automatů*
2. *Teorii vyčíslitelnosti a složitosti* (souhrnně označovanou jako teorie algoritmů)
3. *Logiku* (její informatickou část zaměřenou na problematiku automatizovaného odvozování)

1.1. Teorie formálních jazyků a automatů

Tato disciplína zkoumá vlastnosti jazyků resp. jejich matematických modelů. Snaží se formalizovat intuitivní pojmy abecedy, slova, jazyka. Definuje model generátoru jazyka - gramatiky a jeho duálního pojmu akceptoru jazyka – automatu [Ce92].

Gramatika je souborem pravidel, které umožňují vytvářet správné věty (slova) jazyka a automat je algoritmem, který nám dává prostředek pro rozhodnutí o dané větě, zda patří do jazyka (je správně utvořenou). Na základě různě složitých (generativních) gramatik a automatů se pak

rozlišují třídy jazyků od nejjednodušších - regulárních po nejsložitější - typu 0. Mezi těmito pojmy v jednotlivých třídách jazyků je dokázáno logickými a pochopitelnými prostředky mnoho vztahů. Pro praktické využití v informatikově profesi však mají největší význam dvě třídy nejjednodušších jazyků - regulárních a bezkontextových.

Základy této disciplíny položil v roce 1956 americký matematik Noam Chomsky, který v souvislosti se studiem přirozených jazyků vytvořil matematický model gramatiky jazyka. Začala se však vyvíjet vlastní teorie jazyků, která nyní obsahuje bohaté výsledky v podobě matematicky dokázaných tvrzení teorémů. Tato teorie pracuje se dvěma duálními matematickými entitami, s gramatikou a s automatem, představující abstraktní matematický stroj. Zatímco gramatika umožňuje popsat strukturu vět formálního jazyka, automat dovede tuto strukturu rozpoznat. Poznatky teorie formálních jazyků mají význam pro mnohá odvětví kybernetiky.

Teorie formálních jazyků našla doposud největší uplatnění v oblasti programovacích jazyků. Krátce po Chomského definici gramatiky formálního jazyka a klasifikaci formálních jazyků (Chomského hierarchii formálních jazyků), Backus a Nauer použili základních objektů gramatiky pro definici syntaxe programovacího jazyka Algol 60 (ve tvaru formalismu, jež se nazývá Backus-Nauerova forma). Další vývoj pak přímočaře vedl k aplikacím teorie jazyků v oblasti překladačů programovacích jazyků. Stanovení principů syntaxí řízeného překladu a generátorů překladačů (programovacích systémů, které na základě

formálního popisu syntaxe a sémantiky programovacího jazyka vytvoří jeho překladač) představuje kvalitativní skok při konstrukci překladačů umožňující automatizovat náročnou programátorskou práci spojenou s implementací programovacích jazyků.

Aplikace TFJA

Jak bylo řečeno v předchozích odstavcích, mají poznatky TFJA význam pro kybernetiku i umělou inteligenci, ovšem mají samozřejmě význam pro obory informatiky, neboť s využitím jejich poznatků jsou vybudovány aplikované produkty informatiky jako jsou vývojové nástroje, databázové dotazovací jazyky, značkovací jazyky jako je XML, HTML apod. Zajímavější pro samotnou TI jsou však především teoretické poznatky, které TFJA objevila. Základním poznatkem je zmíněný duální koncept gramatiky-automatu a rozdělení tříd jazyků do Chomského hierarchie. Chomského hierarchie obsahuje 4 třídy jazyků, které lze generovat generativními gramatikami. Jazyky, tedy množiny slov v určité abecedě, lze podle typu generující gramatiky rozdělit. Samozřejmě, že s použitím generativních gramatik nelze vytvořit všechny jazyky – tyto jazyky jsou pak nad touto hierarchií. Pro teoretické výsledky teorie vyčíslitelnosti je důležitá třída jazyků typy 0 a kontextové jazyky (typu 1). Jazyky kontextové mají navíc význam pro umělou inteligenci, konkrétně analýzu přirozeného jazyka. Pro aplikované oblasti informatiky mají význam především jazyky bezkontextové (typu 2) a regulární (typu 3) a to při definování struktur programovacích a jiných jazyků používaných v praxi. Kromě gramatiky je důležitý zmíněný duální pojem automatu, který rozpoznává slova jazyka.

V Chomského hierarchii je možné dále rozlišovat podtřídy podle toho zda jazyky lze analyzovat pomocí deterministického nebo nedeterministického automatu. Zvláště důležité to je pro třídu bezkontextových jazyků, které korespondují s používanými programovacími jazyky. Deterministické jazyky (rozpoznatelné deterministickými zásobníkovými automaty) jsou ve svých speciálních formách jako LL nebo LR jazyky efektivně analyzovatelné.

Vlastnosti jazyků a automatů a jejich vzájemné vztahy jsou v TFJA předmětem zkoumání a existuje mnoho zajímavých výsledků ve formě teorémů, které lze ve většině případů dokazovat poměrně jednoduše s pomocí logických a algoritmizovatelných postupů.

Existují i alternativní hierarchie jazyků založené na odlišných přístupech ke generování jazyků, z nichž zřejmě nejznámější jsou Lindenmayerovy systémy využívané například v biologii pro simulaci chování živých organismů. Teorie jazyků je důležitou součástí informatiky a její poznatky se aplikují nejen v informatice samotné.

1.2. Teorie vyčíslitelnosti a složitosti

Teorie vyčíslitelnosti a složitosti zkoumá vlastnosti algoritmů a to v zásadě ze dvou hlavních hledisek. Vyčíslitelnost se zabývá algoritmickou řešitelností problémů a složitost náročností řešitelných problémů. V roce 1936 Alan Turing, který je pro teoretickou informatiku klíčovou postavou, formuloval svou ideu formalizace pojmu algoritmus ve formě Turingova stroje (TS). Tato formalizace má svůj velmi jednoduchý princip

mechanismu se vstupní potenciálně nekonečnou páskou s danou abecedou a čtecí hlavou, která může zapisovat i číst na pásce a pohybovat se po jednom políčku.

Tento velice jednoduchý formalismus s velkou výpočetní silou umožnil formulovat pro informatiku klíčové pojmy jako jsou rozhodnutelnost a částečná rozhodnutelnost problémů (příp. lze tyto pojmy aplikovat na funkce, množiny či jazyky). Podařilo se dokázat vlastnosti některých problémů (nejznámějším nerozhodnutelných problémem je problém zastavení). Myšlenky důkazů těchto faktů jsou poměrně jednoduché, i když netriviální a lze je najít v literatuře [Ja97a],[Ch84],[Pa02]. Dalšími důležitými výsledky jsou vztahy mezi jazyky typu 0 a rekurzivně spočetnými jazyky, které spadají také do TFJA.

Základní problémy vyčíslitelnosti

Pro formalizaci algoritmu existují další alternativní notace se stejnou výpočetní silou, například teorie rekurzivních funkcí, která je založena na několika základních funkcích a operátorech, pomocí kterých lze dospět ke všem rekurzivním funkcím [Az71]. I když jde o méně používanou formalizaci, lze ji ve výuce efektivně využít. Pro technicky orientované informatiky je vhodná formalizace pomocí RAM strojů (resp. RASP strojů) [Ch84], která je blízká funkčním prostředkům procesorů. Pracuje se střádači (registry) a instrukcemi, které manipulují s těmito registry – vkládání a vybírání hodnot, sčítání, podmíněných skoků v programu apod. Church-Turingova teze postuluje, že ke každému algoritmu lze sestrojit TS. Stejně tak jsou různé formalizace algoritmů mezi sebou rovnocenné (tj. navzájem simulovatelné).

Druhou stránkou je teorie složitosti, která zkoumá jakou náročnost mají řešitelné problémy. Může pracovat s různými notacemi algoritmu, ale základní myšlenka spočívá ve stanovení, za jaký čas dává algoritmus výsledek při stanovení jednotky času například jako jednoho přechodu TS (časová složitost) nebo kolik prostoru spotřebuje algoritmus (prostorová složitost) [Ja97]. Zkoumá se buď složitost konkrétních algoritmů (konkrétní složitost) nebo vlastnosti tříd (strukturální složitost). Důležité je rozlišovat složitost konkrétního výpočtu, složitost algoritmu (funkci velikosti vstupu na čas či prostor) a třídy složitosti (množiny problémů řešitelné s určitou složitostí). Jsou definovány třídy složitosti s určitým odhadem (zanedbáním nevýznamným faktorů) jako je třída zvládnutelných problémů (polynomiální složitosti) a velmi náročných problémů s exponenciální složitostí apod.
Jsou k dispozici důkazy o vlastnostech tříd a vztazích mezi časovou a prostorovou složitostí. TVS dále definuje pojem časově náročných problémů NP-úplných (těžkých), které v praxi představují především optimalizační úlohy jako je problém obchodního cestujícího nebo H-batohu. Pomocí polynomiální převeditelnosti problémů lze dokazovat NP-úplnost těchto problémů. Pro praxi je důležitá hypotéza o P-NP problému, která postuluje, že pro tyto NP-úplné problémy neexistuje zvládnutelný algoritmus (polynomiální). Toto tvrzení však zatím nebylo dokázáno.

Jelikož je objem znalostí, které tato teorie přináší příliš velký, nebudu Vám TVS v těchto textech předkládat. Vyhradím však jednu kapitolu, ve kterém Vám encyklopedickým zpusobem základní problematiku podám. Bylo by

zbytečné, abych se snažil pro tuto problematiku psát další studijní oporu, když taková opora pro distanční studium v elektronické podobě již existuje. Proto Vás odkáži na tento text [Pa02] a pouze Vám dám vodítko, které základní partie byste měli nastudovat.

1.3. Logika

Matematika používá logiku jako prostředek výstavby svých teorií. I když již v třicátých letech vznikly na pomezí logiky a teorie vyčíslitelnosti mnohé výsledky o rozhodnutelnosti teorií (zejména díky K. Gödelovi), význam axiomatických systémů [Lu97] pro informatiku rostl od šedesátých let 20. století díky umělé inteligenci. V roce 1965 Robinson formuloval rezoluční princip a otevřel tak cestu k automatizovanému dokazování vět, což dalo podnět k vytvoření mnoha systémů, které jsou založeny na matematické logice, resp. axiomatických systémech logiky [Lu95]. Bylo vytvořeno mnoho formalizací od rámců, sémantických sítí přes konekcionistické přístupy neuronových sítí až k prostředkům, které využívají logiku v největší míře, jako je klauzulární logika. V knihách z období Robinsonova objevu můžeme pozorovat velký optimismus k těmto metodám, předpovídá se automatické dokazování vědeckých teorií s využitím aparátu logiky. I přestože tomu prozatím tak není a zřejmě ani v dohledné době nebude, využití například v robotice vedlo k tvorbě a aplikaci mnoha funkčních systémů jako STRIPS, GPS nebo PLANNER.
Jelikož Vaše studium poskytuje velmi dobré základy matematické logiky i vyšších partií z oblasti automatizované dedukce nebudeme se těmito

otázkami zabývat. Pouze pro zájemce uvádím odkaz na existující literaturu a velmi kvalitní studijní opory Doc. Lukasové (jsou uvedeny v seznamu literatury).

2. Konečný automat

Cíl:

Po prostudování této kapitoly budete umět vysvětlit:

- intuitivní pojem jazyka a gramatiky, jejich význam v praxi a tím získáte motivaci pro studium
- intuitivní pojem automatu a vztah mezi jazykem, gramatikou a automatem

Získáte základní přehled o těchto otázkách:

- teorie formálních jazyků
- hierarchie jazyků, problém determinismu a nedeterminismu

Naučíte se používat:

- základní pojmy: abeceda, slovo, jazyk
- definice konečného automatu a budete umět podat jednoduché příklady

2.1. Intuitivní pojem jazyka a gramatiky

Jak jste již mohli číst v úvodu znáte pojem jazyka a gramatiky z vlastní zkušenosti již od dětství, i když si to třeba neuvědomujete. Uvědomte si následující schéma:

Řešený příklad 1:

<Jednoduchá česká věta> ::= <Podmět><Přísudek><Předmět>
<Podmět> ::= Jiří | Jan
<Přísudek> ::= má | řídí
<Předmět> ::= auto | firmu
Toto schéma obsahuje pravidla, která vždy položky v lomených závorkách přepisují na posloupnosti (řetězce, uspořádaný výčet) buď stejných položek v hranatých závorkách nebo slov (symbolů). Jde vlastně o gramatiku (soubor pravidel), jak položky v hranatých závorkách (může říci proměnné) můžeme přepisovat postupně až na slova (symboly). Můžeme tedy tímto postupem dostat například takovéto jednoduché české věty:

<Jednoduchá česká věta> -> <Podmět><Přísudek><Předmět> -> Jiří <Přísudek><Předmět>
-> Jiří řídí <Předmět> -> Jiří řídí firmu.

nebo

<Jednoduchá česká věta> -> <Podmět><Přísudek><Předmět> -> Jan <Přísudek><Předmět>
-> Jan má <Předmět> -> Jan má auto.

Tedy postupným dosazování za proměnné podle pravidel dojdeme až na posloupnosti, které už neobsahují žádné proměnné.

Průvodce studiem:

Odborně se neříká proměnné, ale neterminály (nejsou konečné – terminální) a slova, ale terminály (symboly, které se již nemohou dál rozvíjet). Pozor na zmatení v pojmech! Slovo je chápáno v TFJA jako věta, tedy již správně utvořená a symbol je chápán jako slovo. Např. „Jan“ je zde v tomto případě symbol a Jan řídí auto je slovo (věta)! Vidíte, jak důležitá je formalizace – jasná dohoda o používaných pojmech!

Gramatika *Gramatika je tedy laicky řečeno soubor výchozích symbolů, proměnných a pravidel pro vytváření vět.*

Jazyk *Jazykem jsou všechny věty, které jsou vytvořeny v souladu s gramatikou.*

Řešený příklad 2:

Budeme-li uvažovat o tom, co je to jazyk, zkusme vyjít z gramatiky z předchozího příkladu na jednoduchou českou větu.

Jazyk L je pak množina:

L = { Jiří má auto, Jiří má firmu, Jiří řídí auto, Jiří řídí firmu, Jan má auto, Jan má firmu, Jan řídí auto, Jan řídí firmu}

Jazyk má tedy 8 prvků (správně utvořených vět). Uvědomte si, že tento příklad je velice jednoduchý. Jednoduchá česká věta se rozvinula na posloupnost podmětu, přísudku a předmětu. Každá z těchto tří proměnných se mohla podle gramatiky rozvinout do dvou symbolů (slov). To tedy dává 2^3 možných vět. Jazyk je tedy v tomto případě konečný, neboť má konečně mnoho prvků. Například věta Auto řídí Jan by podle této gramatiky nepatřila do jazyka, protože není utvořena podle pravidel (nedodržuje slovosled daný prvním pravidlem)!

V dalším studiu se budeme zabývat ještě mnohem jednoduššími jazyky, které budou složeny jen z primitivních symbolů (např. 0, 1), avšak tyto jazyky budou často i nekonečné, tedy budou mít nekonečně mnoho různých vět. Například jazyk L složený z vět, které obsahují nejprve jakýkoliv (nenulový) počet nul a za nimi následuje stejný počet jedniček je nekonečný. Zapsáno symbolicky je to množina: L = { 01, 0011, 000111, ...}, kde ... značí dalších nekonečně mnoho možností.

Další pojem, který Vám sice zatím nemohu objasnit podrobně a přesně, neboť pro něj zatím nemáme vybudován aparát, je automat. Automat má zjišťovat, zda věta patří do určitého jazyka. Například automat, který zjišťuje zda věta je jednoduchá česká věta podle příkladu 1, by pracoval tak, že by si větu rozkouskoval do slov a pak zjistil zda první slovo je podmět, druhé přísudek a třetí předmět. Pokud by to tak bylo, pak by odpověděl, že slovo patří do jazyka.

Automat je postup (algoritmus, matematická struktura), který k danému jazyku rozlišuje, která slova do něj patří.

Pokuste se odpovědět na otázku: Je složitější napsat gramatiku (soubor pravidel) pro český jazyk nebo programovací jazyk Pascal?

2.2. Základní pojmy teorie jazyků

Abychom mohli ve studiu TFJA pokračovat, zavedeme nejprve základní definice, ze kterých budeme vycházet.

Základní pojmy a definice

Abeceda

Definice 1: **<u>Abeceda</u> Σ je konečná množina symbolů.** *Abeceda*

Řetězec (slovo)

Řetězec (slovo)

Definice 2: <u>Řetězec</u> (<u>slovo</u>) α prvků z konečné množiny Σ je libovolná konečná posloupnost prvků této množiny. Řetězce zpravidla označujeme řeckými písmeny. Počet prvků v řetězci udává jeho délku a označujeme ji |α|. Řetězec, který neobsahuje žádný prvek, nazýváme prázdný řetězec a označujeme ho ε nebo e (nedojde-li k záměně). Jeho délka je 0.

$\alpha = a_1a_2...a_{k-1}a_k$ $a_i \in \Sigma$, i = 1,2,...,k; $|\alpha| = k$

Pozn. Velmi dobře znáte řetězce z běžného života – např. Vaše jméno je řetězec složený z písmen české abecedy.

Řešený příklad 3:
Mějme množinu Σ={0,1}. Řetězce prvků této množiny jsou binární čísla bez znaménka. Například

α = 001011 |α| = 6

β = 0000 |β| = 4

Konkatenace (zřetězení), uzávěry množiny

Definice 3: <u>Zřetězení</u> (<u>konkatenace</u>) řetězců $\alpha=a_1a_2\ldots a_r$ a $\beta=b_1b_2\ldots b_s$ je řetězec $\alpha\beta=a_1a_2\ldots a_rb_1b_2\ldots b_s$.

Zřetězení (konkatenace)

Pozn. Pro délku konkatenace $\alpha\beta$ dvou řetězců α a β platí $|\alpha\beta| = |\alpha| + |\beta|$. Dále pro konkatenaci libovolného řetězce α s prázdným řetězcem ε platí $\alpha \cdot \varepsilon = \varepsilon \cdot \alpha = \alpha$

Konkatenace může být provedena také několikanásobně, potom používáme následující značení: α^n, přičemž označuje konkatenaci řetězce α provedenou n-krát. Například $(01)^3$ je řetězec 010101.

Pozn. Zřetězení opět není nic složitějšího než spojení dvou řetězců k sobě.

Uzávěry

Definice 4: <u>Pozitivním uzávěrem</u> Σ^+ konečné množiny prvků Σ nazveme množinu všech řetězců sestavených z prvků množiny Σ bez prázdného řetězce. (Uzávěr Σ^+ je spočetná množina.)

Definice 5: <u>Uzávěr</u> Σ^* množiny Σ navíc obsahuje prázdný řetězec ε, tj. je definován $\Sigma^* = \{ \varepsilon \} \cup \Sigma^+$

Pozn. Tyto uzávěry jsou tedy všechny možné kombinace, jak nějakou množinu můžeme prokombinovat spojením jejich řetězců libovolně-krát za sebou (viz příklad).

Řešený příklad 4:
Nechť množina prvků je $\Sigma = \{ a,b,c \}$ Její pozitivní uzávěr a uzávěr je (pouze některé prvky – uvědomte si, že je nekonečný!)

$\Sigma^+ = \{ a,b,c,aa,ab,ac,ba,bb,bc,ca,cb,cc,aaa,aab,\dots \}$

$\Sigma^* = \{ \varepsilon,a,b,c,aa,ab,\dots \}$

Jazyk

Formální jazyk

Definice 6: Nechť je dána abeceda Σ, pak libovolná podmnožina uzávěru Σ^* je formálním jazykem nad abecedou Σ. Tedy formální jazyk L je definován $L \subseteq \Sigma^*$ (pro jednoduchost budeme mluvit o jazyce).

Specifické případy:

- Je-li L prázdná množina ($L = \varnothing$), je to prázdný jazyk.
- Je-li L konečná podmnožina, je to konečný jazyk.
- Je-li L je nekonečná podmnožina, je to nekonečný jazyk.

Jazyk je tedy výběrem slov v určité abecedě ze všech možných kombinací symbolů. Může samozřejmě obsahovat všechny slova, žádné nebo některé. Je jasné, že jazyk může být různě složitý (jak jsme to již probíraly).

Řešený příklad 5:
Nechť abeceda je

$\Sigma = \{ +,-,0,1,2,\ldots,9 \}$

Pak množina celých čísel je jazykem nad Σ.

Uvedená definice formálního jazyka je pro praktické použití příliš obecná. Pouze definuje, co jazyk je, ale neposkytuje žádné prostředky pro popis struktury jazyka nebo zjištění, zda určitý řetězec do jazyka patří.

Pojmy, které jsme si právě zavedli, Vás budou provázet celým studiem. Pokuste se je nyní znovu projít a zkuste si pro každý z nich představit konkrétní příklad, podobně jako jste je viděli v mnou podaných příkladech. Uvidíte, že pokud si hned od začátku studia budete za každým matematizovaným pojmem představovat konkrétní příklady ze života, bude pro Vás mnohem jednodušší pracovat s nimi ve složitějších kapitolách.

2.3. Konečný automat

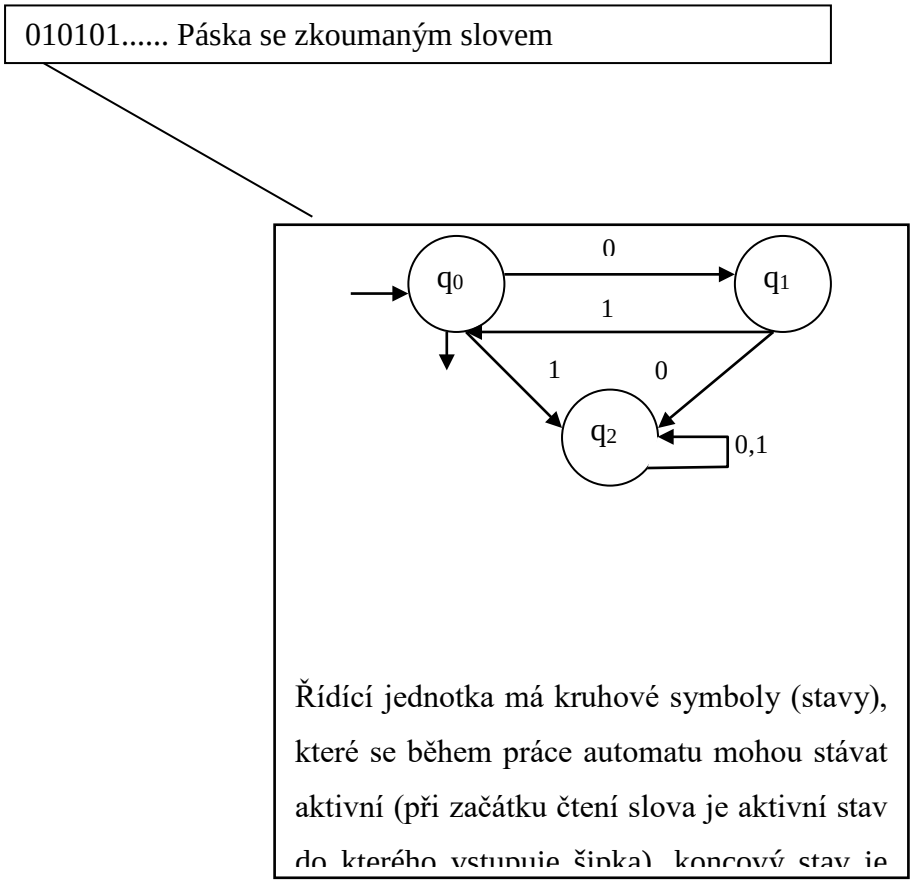
Mluvíme-li o jazyce jako množině slov a gramatice jako o souboru pravidel, která umožňují generovat slova z jazyka, pak automat je prostředkem, jak zjistit, která slova do jazyka patří a která ne.

Konečný automat

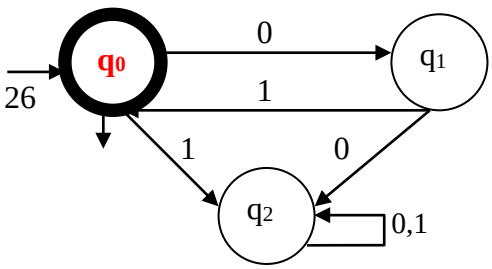
Pokuste se představit si následující stroj - automat. Má pásku, na kterou můžete zapisovat po symbolech slova, která chcete rozpoznat, zda patří do jazyka nebo ne. Dále má řídící jednotku se stavy, které si můžete představit jako žárovky, které se rozsvítí, pokud je automat právě v tom konkrétním stavu. Z pásky umí automat číst pouze po jednotlivých symbolech a nemůže se vracet zpět na již přečtené symboly. Je důležité, abyste si uvědomili, že automat si nemůže nic pamatovat. Vždy vidí jen jeden symbol ze zkoumaného slova. Dále automat ví, jak reagovat na situaci pokud je v nějakém stavu a na pásce vidí určitý symbol. Vždy je pak výsledkem takové akce přechod do nějakého stavu (může jít i o stejný stav). Nakonec má automat k dispozici informaci, ve kterém stavu má

začít a který stav je koncový (nebo více stavů) a pokud se dostane do takového stavu, pak je slovo z jazyka.

Podívejme se na příklad:

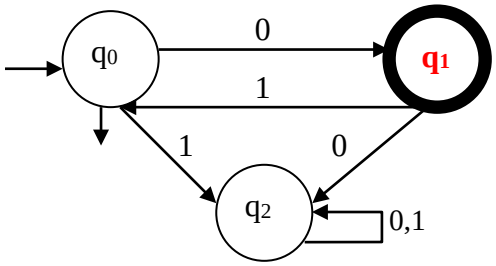


A nyní si znázorněme automat v činnosti. Postupně načteme slovo 0101 a vždy zvýrazníme ten stav, který je právě aktivní. Ve čteném slově pak

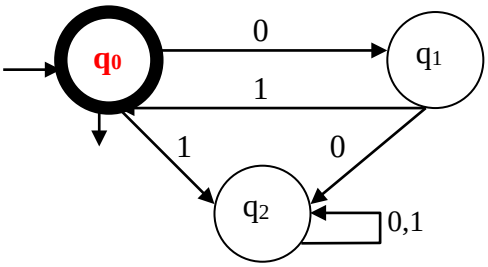


bude zvýrazněn ten symbol, který bude právě přečten.

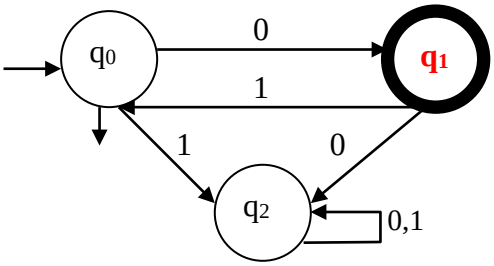
Počáteční situace – automat začíná svůj výpočet a je právě ve stavu q_0 a chystá se přečíst ze slova **0**101 první symbol.



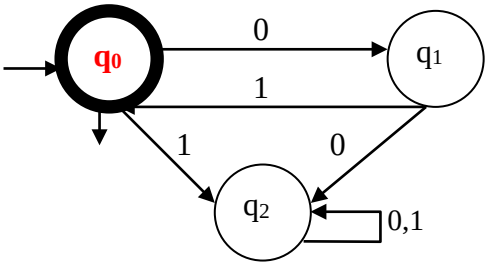
Automat přečetl první symbol 0 ze slova 0**1**01 a podle definovaného přechodu se stal aktivní stav q_1



Po přečtení symbolu 1 ze slova se automat dostal opět do stavu q_0. Ve slově nyní následuje další symbol 0 - 01**0**1



Automat je opět ve stavu q_1 a zbývá přečíst poslední symbol slova 0101



V poslední fázi výpočtu již není symbol, který by se dal z pásky přečíst. Proto zbývá zjistit, zda se automat dostal do stavu, který je koncový – má výstupní šipku. Je tomu tak, proto říkáme, že automat slovo 0101 rozpoznal – přijal.

Jak vidíte, princip tohoto typu automatu není nijak složitý. Doufám, že jste pochopili, jak se z počátečního (vstupního) stavu postupně dostává do jiných pomocí čtení jednotlivých symbolů na pásce. Uvědomte si, že automat přečte celé slovo a pak je otázka, zda skončil ve stavu koncovém nebo ne. Pokud ano, slovo patří do jazyka, který automat umí rozpoznávat. Tedy můžeme hovořit o slovech, které automat rozpoznává (patří do

jazyka rozpoznatelného tímto automatem) a které ne. Zkusme spolu uvažovat o jaký jazyk v tomto konkrétním případě jde.

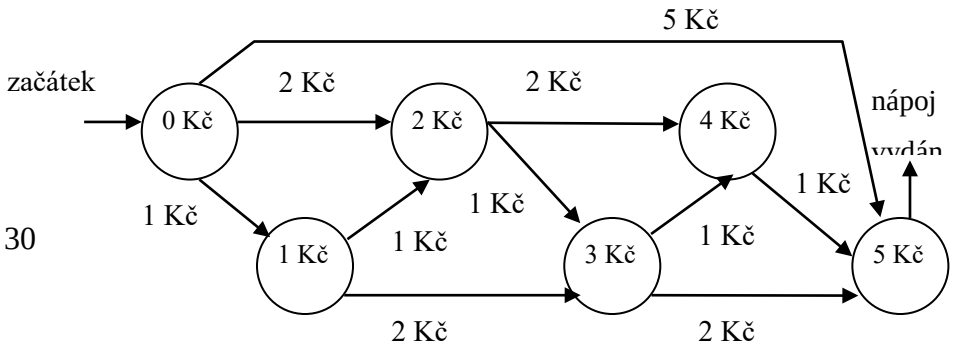
Pokud se blíže na automat podíváte, pak uvidíte, že pokud dostává na pásku slova, která obsahují přesně za sebou sled symbolů 01, pak se pohybuje mezi stavy q_0 a q_1. Pokud by však dostal po symbolu 0 další symbol nula, pak skončil ve stavu q_2, ze kterého by se už pak nemohl dostat jinam, protože přechod na 0 i 1 ze stavu q_2 vede opět do q_2. Stejná situace nastane, pokud se po symbolu 1 bude číst další symbol 1. Stav q_2 je v podstatě jakási „černá díra“, která se stará o situace, které nastanou ve slovech, která nechceme přijmout. Takže tento automat přijímá slova, která jsou libovolně krát zřetězeným slovem 01. A všimněte si, že rozpozná i slovo, které neobsahuje nic (tedy prázdné slovo), neboť počáteční stav q_0 je zároveň koncovým.
Zapišme tedy jazyk, který automat rozpoznává do matematického zápisu.

$L = \{ (01)^n, n \geq 0 \}$ (jinak řečeno, automat rozpoznává ta slova, která obsahují posloupnosti 01 v libovolném množství za sebou)

Uvědomme si, že automat také nemusí v určitém kroku mít definováno, do jakého stavu jít, nebo může mít více možností. Pak hovoříme o nedeterministickém automatu. S pojmem nedeterministický se budeme setkávat i u dalších typů automatů. Nedeterministický by se dalo volně přeložit jako takový, který nemůže v určitém okamžiku jednoznačně určit, co dělat.

Konečné automaty kolem Vás

Abychom nemluvili pouze o odtažitém příkladě se symboly nula a jedna, zkuste si představit například automat na kávu. Asi jste s ním již všichni přišli do kontaktu a vhazovali do něj mince, až jste se dočkali svého oblíbeného nápoje. Nedělali jste vlastně nic jiného než že jste konečnému automatu dávali na vstupní pásku symboly – mince o určité hodnotě. Stavy v tomto případě určují, kolik jste již do automatu vhodili peněz (počáteční stav je 0 Kč, koncový je cena nápoje). Pokud bychom to ilustrovali jako na předchozím (matematickém příkladě), pak můžete sledovat popis takového automatu na obrázku. Šipky mezi stavy reprezentují přechodovou funkci mezi nimi – tedy jakou minci jsme vhodili (můžete vhodit minci 1 Kč, 2 Kč nebo 5 Kč).



Ještě než ukončíme tuto kapitolu, zaveďme si definici konečného automatu, se kterým jsme se nyní seznámili intuitivně. Vím, že matematické definice pro Vás budou obtížné, ale pokuste se je prosím pochopit. Nestudujte text dál (další kapitoly), pokud si nebudete jisti, že se v matematicky vyjádřené definici vyznáte a chápete její smysl. Za vlastními definicemi najdete také pomůcku, jak docílit pochopení těchto formálních pojmů.

Konečný automat (deterministický) - KA

Deterministický konečný automat

Definice 7: ***(Deterministickým) konečným automatem*** **(DKA) nazýváme každou pětici A =(Q,Σ,δ, q_0, F), kde**

- **Q je konečná neprázdná množina (*množina stavů, stavový prostor*)**
- **Σ konečná neprázdná množina (*množina vstupních symbolů, vstupní abeceda*)**
- **δ je zobrazení Q ×Σ → Q (*přechodová funkce*)**
- **q_0 ∈ Q (*počáteční stav, iniciální stav*)**
- **F ⊆ Q (*množina koncových stavů, cílová množina*).**

Zobecněná přechodová fce

Přechodovou funkci δ:Q×Σ → Q konečného automatu A =(Q,Σ,δ, q_0, F) rozšíříme na *zobecněnou přechodovou funkci* δ^*:Q×Σ^* → Q následovně:

$\delta^*(q,e)=q, \forall q \in Q$

$\delta^*(q,wa)=\delta(\delta^*(q,w),a), \forall q \in Q, w \in \Sigma^*, a \in \Sigma.$

Pozn. Zobecněná přechodová funkce není opět nic složitého – jde o rozšíření, které umožňuje zkoumat, kam se dostaneme z určitého stavu na slovo (tedy několik symbolů místo jednoho). Definici si lépe přečtete, pokud si vzpomenete na funkci faktorial. Zřejmě jste v programování konstruovali algoritmus této funkce pomocí rekurze. Víte, že faktorial(n) = n * faktorial(n-1) a faktorial(0) = 1. Přesně toto říká definice pro tuto zobecněnou funkci. Tedy, že problém přechodu na slovo lze rozložit na problém přechodu na poslední symbol slova (což umíme podle obyčejné přechodové funkce) a pak jen stačí zbytek slova řešit stejným postupem, až dojdeme na prázdné slovo. Prázdné slovo odkudkoliv nemůže způsobit nic jiného, než že se zůstane ve stejném stavu.

Jazyky rozpoznávané a rozpoznatelné

Jazykem rozpoznávaným konečným automatem ***A*****, pak nazveme množinu L(*** ***A*** **)={ w|w ∈ Σ* ∧ δ*(q0,w) ∈ F }.**

Řekneme, že *jazyk* L (nad abecedou Σ) *je rozpoznatelný konečným automatem* , jestliže existuje konečný automat *A* takový, že L(*A*)=L.

Konečné automaty reprezentují výše uvedené množiny a zobrazení. Kromě přechodové funkce by mělo být pro Vás poměrně jednoduché pochopit, co jsou jednotlivé množiny. Přechodová funkce určuje, do jakého stavu se přechází na daný symbol a aktuální stav. Proto má strukturu danou kartézským součinem v definici (osvěžte si pojem kartézského součinu z teorie množin).

Rozlišujte prosím, co je jazyk rozpoznávaný a rozpoznatelný!

Rozpoznávaný je jazyk konkrétním automatem – jde o slova, která ho dostanou do koncového stavu. Tedy například u našeho automatu na kávu jde o všechny posloupnosti, jak lze do něj vhodit mince o celkové hodnotě 5 Kč (např. posloupnost 1Kč, 2 Kč, 2 Kč, a další)

Rozpoznatelný je jazyk tehdy, pokud k němu vůbec lze sestrojit KA. Říkali jsme, že jazyky jsou různě složité. Např. pro jazyk Pascal byste nedokázali sestrojit takový automat (jenž by skončil v koncovém stavu pro programy správně napsané), protože na to je jazyk příliš složitý. Jak uvidíme v pokročilých kapitolách, je konečný automat příliš „slabá“ formalizace na takový jazyk jako je Pascal.

Možnosti reprezentace konečného automatu (způsobu zápisu):

Reprezentace automatu

- zápis výčtu jednotlivých prvků pětice konečného automatu
- tabulka
- stavový diagram
- stavový strom

Řešený příklad 6:
Mějme konečný automat $A=(Q,\Sigma,\delta, q_0, F)$

$Q=\{q_0,q_1,q_2,q_3\}$, $\Sigma = \{0,1\}$

$\delta(q_0,0)=q_0$, $\delta(q_0,1)=q_2$

$\delta(q_1,0)=q_1$, $\delta(q_1,1)=q_3$

$\delta(q_2,0)=q_1$, $\delta(q_2,1)=q_3$

$\delta(q_3,0)=q_1$, $\delta(q_3,1)=q_3$

$F=\{q_1,q_2\}$

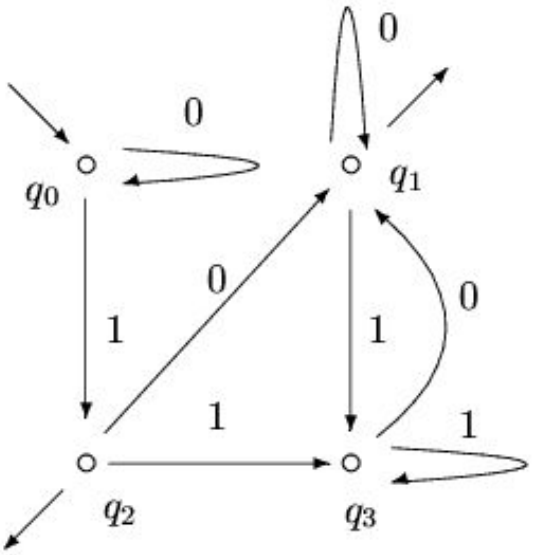
Tabulka

Reprezentace KA A tabulkou:

	$Q^{\setminus\Sigma}$	0	1
→	q_0	q_0	q_2
←	q_1	q_1	q_3
←	q_2	q_1	q_3
	q_3	q_1	q_3

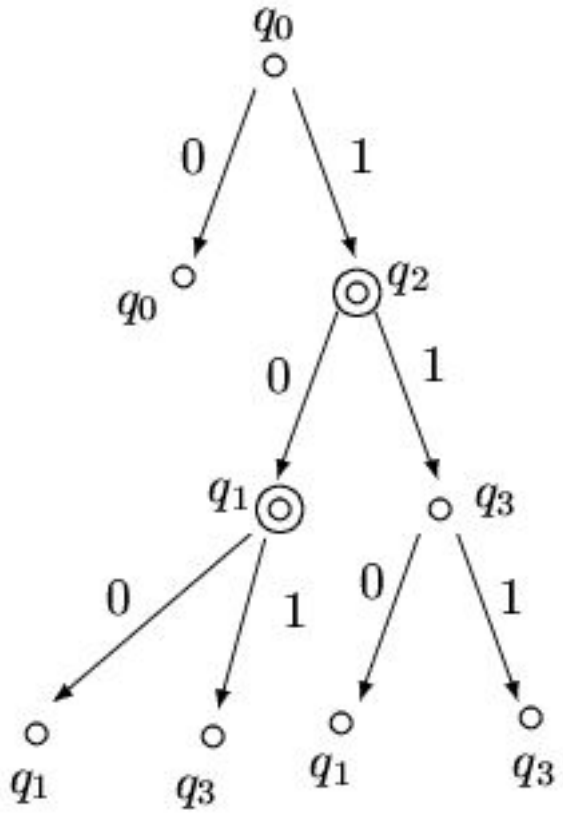
Stavový diaaram

Reprezentace KA A stavovým diagramem:



Reprezentace KA A stavovým stromem:

Stavový strom



Konečný automat (nedeterministický)

Nedeterministický automat

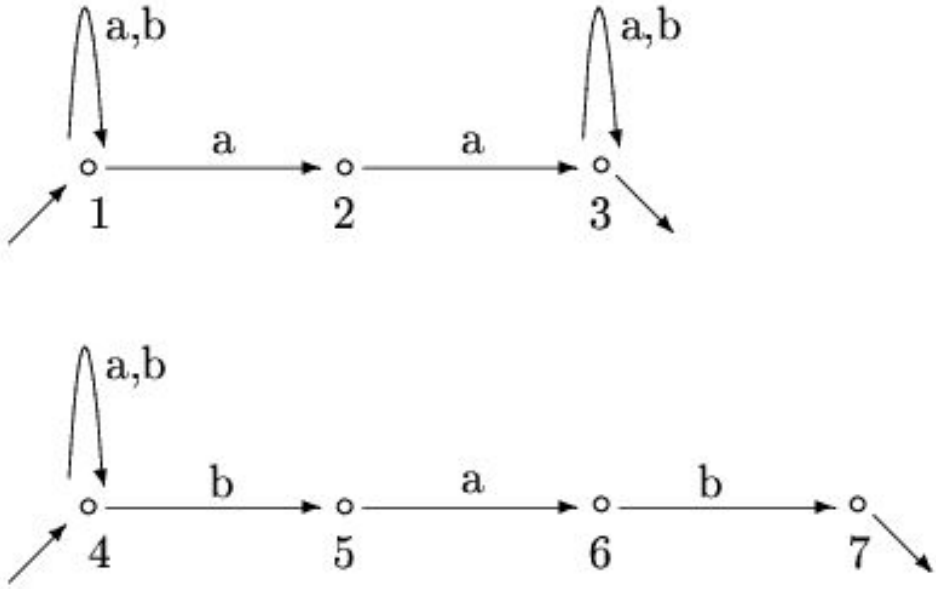
Definice 8: ***Nedeterministickým konečným automatem*** **(NKA) budeme nazývat pětici $A = (Q, \Sigma, \delta, I, F)$, kde**

- **Q a Σ jsou po řadě neprázdné množiny stavů a vstupních symbolů,**
- **$\delta: Q \times \Sigma \rightarrow P(Q)$ je přechodová funkce ($P(Q)$ je množina všech podmnožin množiny Q).**
- **$I \subseteq Q$ je množina počátečních stavů a $F \subseteq Q$ je množina koncových stavů.**

Poznámka: O dosavadním konečném automatu budeme hovořit jako o *deterministickém*. Všimněte si dvou základních rozdílů:

- NKA už nemusí začínat jen v jednom počátečním stavu, ale může jich mít několik (může si „vybrat“ kde začít)
- přechodová funkce je zobrazením do potenční množiny, tedy množiny všech podmnožin; ze stavu se na symbol může jít nejen do jednoho stavu, ale do libovolného počtu (podmnožiny) nebo i nikam (do prázdné podmnožiny)

Řešený příklad 7:
Příklad nedeterministického konečného automatu:



Nedeterministický automat reprezentovaný tabulkou:

	$Q \setminus \Sigma$	a	b
→	1	1,2	1
	2	3	–
←	3	3	3
→	4	4	4,5
	5	6	–
	6	–	7
←	7	–	–

Neformálně: NKA přijímá slovo w právě tehdy, když existuje cesta z nějakého z počátečních stavů do nějakého koncového stavu, jejíž ohodnocení je rovno w. V tom je potíž nedeterminismu – vy jako informatici, pro které je základním myšlenkovým principem řešení problému algoritmus, budete mít s pochopením této věty problém. Nedeterminismus totiž vyžaduje pouze, aby řešení existovalo. Cest, přes které se NKA může dostávat z počátečního stavu do koncového může být mnoho. Je to jako pro Vás opět známou hru šachy. Víte, že v ní je obrovské množství variant, jak hra může probíhat. V každém kroku máte několik možností, kterou figurkou a kam táhnout. Přesto nevíte, zda zrovna tato Vaše volba Vám nakonec zajistí vítězství. U automatu jde o něco podobného. „Nezajímá“ nás, jak si tuto hru automat rozehraje, ale pokud šance na vítězství existuje (slovo bude rozpoznáno), pak nám to stačí na tvrzení, že vyhrát lze.

Definice 9: Pro NKA *A* =(Q,Σ,δ, I, F) definujeme *zobecněnou přechodovou funkci* $\delta^*:P(Q)\times\Sigma^* \to P(Q)$ následující rekurzivní definicí:

1. $\delta^*(K,e)=K\ \forall K \in P(Q)$ **(tedy** $K \subseteq Q$**)**

2. $\delta^*(K,wa)=\cup_{q \in \delta^*(K,w)}\delta(q,a)\ \forall K \in P(Q), w \in \Sigma^*, a \in \Sigma.$

Slovo $w \in \Sigma^*$ ***je přijímáno*** **NKA *A*, jestliže** $\delta^*(I,w)\cap F \neq \emptyset$**.**

Pozn. Definice této funkce je v tomto případě složitější. Uvědomte si, že výsledkem klasické přechodové funkce je množina stavů, nikoliv jen jeden

stav. Proto se musí procházet všechny a je nutná operace sjednocení výsledků.

Jazyk **L(*A*)** ***rozpoznávaný*** **NKA *A* je množina všech slov přijímaných automatem *A*. (L(*A*)={ w ∈ Σ* | δ*(I,w) ∩ F ≠ ∅})**

Jazyk **L *je rozpoznatelný* NKA, právě když existuje NKA *A* takový, že L(*A*)=L.**

Poznámka: Slovo $w=a_1\ a_2 ...\ a_n$ ($a_i \in \Sigma$) je tedy přijímáno NKA *A* právě tehdy, když existuje posloupnost $q_1\ q_2 ...\ q_{n+1}$ stavů z Q taková, že $q_1 \in I$, $q_{n+1} \in F$, a pro všechna $i \in \{ 1,2,...n\}$ je $q_{i+1} \in \delta(q_i,a_i)$. Speciálně $e \in L(A)$ právě, když $I \cap F \neq \emptyset$.

Jazyk rozpoznávaný konečným automatem

Definice 10: Vstupní slovo $w=x_1x_2\ldots x_m \in \Sigma^+$ je rozpoznáváno nedeterministickým nebo deterministickým konečným automatem A, jestliže existuje posloupnost stavů $q_0,q_{i1},q_{i2},\ldots,q_{im}$ taková, že:

$q_{ik} \in \delta(q_{ik-1},x_k)$

v případě nedeterministického automatu nebo

$q_{ik} = \delta(q_{ik-1},x_k)$

v případě deterministického automatu a $q_{im} \in F$ (stav, ve kterém automat skončil činnost, je koncový stav).

Množina všech slov w $\in\Sigma^*$, jež jsou rozpoznávána (přijata) automatem A, tvoří jazyk rozpoznávaný automatem A. Označujeme ho L(A).

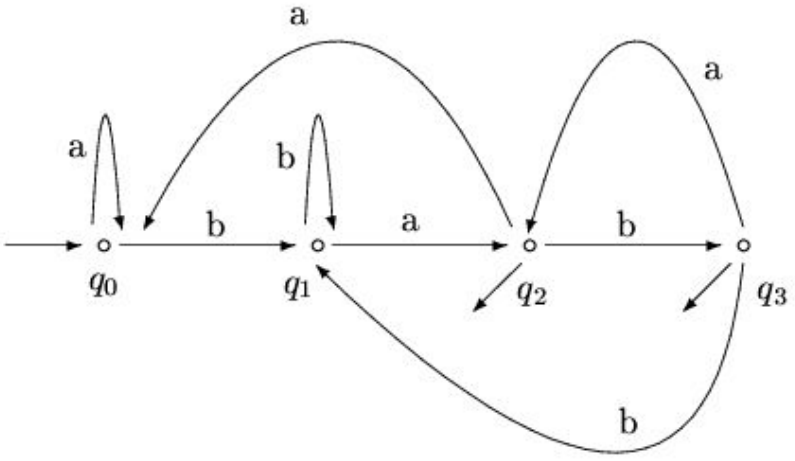
Jazyk rozpoznatelný konečným automatem

Definice 11: Řekneme, že *jazyk* L (nad abecedou Σ) *je rozpoznatelný konečným automatem* , jestliže existuje konečný automat *A* takový, že L(*A*)=L.

Řešený příklad 8:

L={w ∈ {a,b}*|w končí ba nebo bab}

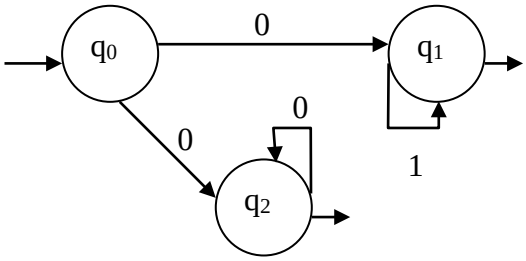
je jazyk rozpoznatelný konečným automatem ($\exists A_1$; L(A_1)=L):



Automat A_1:

V matematických definicích se nyní pokusíme udělat jasno. Vidíte, že konečný automat, který jsme poznali v jeho intuitivní podobě, je ve formalizované podobě matematickou strukturou – pětic, která ovšem reprezentuje popsané složky automatu, který jsme si představovali jako konkrétní stroj, který byste si třeba sami mohli sestavit doma v dílně. Ta pětice obsahuje stavy (žárovky), abecedu symbolů (písmena na vstupní pásce), počáteční stav (tedy označení toho, od kterého se začíná výpočet stroje, množinu koncových stavů (tedy takových, ve kterých pokud automat skončí, pak čtené slovo je přijato). Nejdůležitější je přechodová funkce, která má jako argument aktuální stav a četný symbol a k němu zobrazení dává nový stav (resp. celou množinu stavů u nedeterministického automatu). Kartézský součin tedy udává co se zobrazuje na co (stavy a symboly na stavy).

V další kapitole se budeme rozdílu mezi nedeterministickým a deterministickým automatem zabývat detailně. Pokuste se do další lekce promyslet, jak by vypadala a chovala se sada „žárovek“ (stavů) u nedeterministického automatu. Tedy například, co by se stalo, kdyby bylo možné se ze stavu q_0 do q_1 a q_2 zároveň (viz následující obrázek popisující přechody automatu).



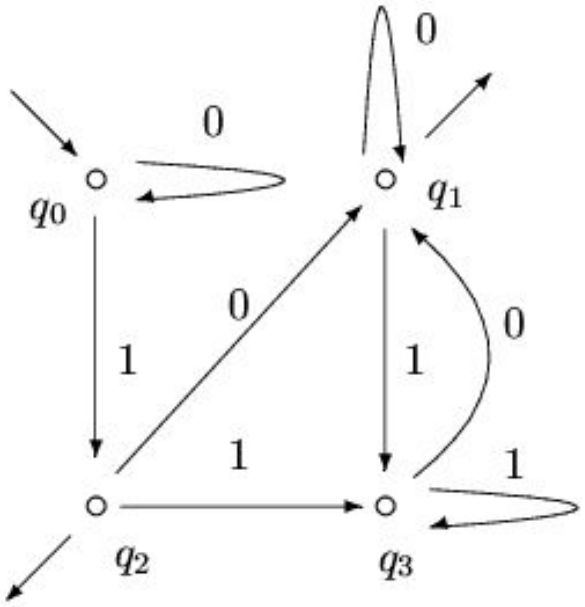
Pokud budete chtít s automaty pracovat, převádět je, upravovat, je vždy nutné mít přesnou matematickou formulaci. Přesto se snažte i za těmito formulacemi vidět konkrétní příklady. Vzpomeňte si na známou úlohu z matematiky, kdy máte dva proti sobě jedoucí vlaky, různými rychlostmi a vy máte určit, kde nebo kdy se střetnou. Jistě víte, jak jednoduché je tento problém vyřešit, pokud jej formulujete ve formě rovnic a řešíte naučeným způsobem tyto rovnice jako manipulaci se symboly. Přesně to je i případ této formalizace. Jde o to, abyste problémy z „reálného života“ uměli vyřešit dokazatelnými a přesně formulovanými (algoritmickými postupy). Zamyslete nad tím a pochopíte, že formalizace není tak samoúčelná, jak se může na první pohled zdát.

Kontrolní úkoly:

Úkol 1: Mějme slova u=0010=$0^2 1^1 0^1$, v=11000=$1^2 0^3$, u,v $\in \{0,1\}^*$. uv=?, vu=?, uvv=?, u^3=?, v^1=?, v^2=?, |v| = ?, |u| = ?, $|v^2|$ = ?, $|u^3|$ = ?

Úkol 2: Mějme abecedu $\Sigma = \{0,1\}$ a jazyky $L_1=\{0110,10\}$, $L_2=\{e\}$, $L_3=\emptyset$, $L_4=\{0^n1^n; n \geq 0\}$, $L_5=\{a^n01; n \geq 1\}$, $L_6=\{aa,a\}$, $L_7=\{b^{2k}a^kc; k \geq 0\}$. Určete, zda jazyky L_1, L_2, L_3, L_4, L_5, L_6, L_7 jsou jazyky nad abecedou Σ.

Úkol 3: Vezměme konečný automat:



Vyčíslete pomocí zobecněné přechodové funkce do jakého stavu se automat dostane v následujících případech:

$\delta^*(q_0,011001)=?$

$\delta^*(q_2,011001)=?$

Řešení 1:

uv=001011000, vu=110000010, uvv=00101100011000,

u^3=001000100010, v^1=11000, v^2=1100011000, |v| = 5, |u| = 4, $|v^2|$ = 10, $|u^3|$ = 12.

Řešení 2:

L_1={0110,10} je jazyk nad abecedou Σ (dvouprvkový).

L_2={e} je jazyk nad abecedou Σ obsahující pouze prázdné slovo.

L_3=∅ je prázdný jazyk nad abecedou Σ.

L_4={0^n1^n; n ≥ 0} je jazyk nad abecedou Σ obsahující slova sudé délky, skládající se ze dvou úseků stejné délky, z nichž první obsahuje pouze symboly 0 a druhý pouze symboly 1.

L_5={a^n01; n ≥ 1} není jazyk nad abecedou Σ (ale je jazykem nad abecedou {a,0,1}).

L_6={aa,a} není jazyk nad abecedou Σ (ale je jazykem nad abecedou {a}).

L_7={$b^{2k}a^kc$; k ≥ 0} není jazyk nad abecedou Σ (ale je jazykem nad abecedou {a,b,c}).

Řešení 3:

$\delta^*(q_0,011001) = \delta(\delta^*(q_0,01100),1) = \delta(\delta(\delta^*(q_0,0110),0),1)=$

$=\delta(\delta(\delta(\delta^*(q_0,011),0),0),1) = \delta(\delta(\delta(\delta(\delta^*(q_0,01),1),0),0),1)=$

$=\delta(\delta(\delta(\delta(\delta^*(q_0,0),1),1),0),0),1) = \delta(\delta(\delta(\delta(\delta(\delta^*(q_0,e),0),1),1),0),0),1)=$

$=\delta(\delta(\delta(\delta(\delta(\delta(q_0,0),1),1),0),0),1) = \delta(\delta(\delta(\delta(\delta(q_0,1),1),0),0),1)=$

$=\delta(\delta(\delta(\delta(q_2,1),0),0),1) = \delta(\delta(\delta(q_3,0),0),1)=$

$=\delta(\delta(q_1,0),1) = \delta(q_1,1)=q_3$

$\delta^*(q_2,011001) = \delta(\delta^*(q_2,01100),1) = \delta(\delta(\delta^*(q_2,0110),0),1)=$

$=\delta(\delta(\delta(\delta^*(q_2,011),0),0),1) = \delta(\delta(\delta(\delta(\delta^*(q_2,01),1),0),0),1)=$

$=\delta(\delta(\delta(\delta(\delta(\delta^*(q_2,0),1),1),0),0),1) = \delta(\delta(\delta(\delta(\delta(\delta(\delta^*(q_2,e),0),1),1),0),0),1)=$

$=\delta(\delta(\delta(\delta(\delta(q_2,0),1),1),0),0),1) = \delta(\delta(\delta(\delta(\delta(q_1,1),1),0),0),1)=$

$=\delta(\delta(\delta(\delta(q_3,1),0),0),1) = \delta(\delta(\delta(q_3,0),0),1)=$

$=\delta(\delta(q_1,0),1) = \delta(q_1,1)=q_3$

Nejdůležitější probrané pojmy:

- JAZYK, GRAMATIKA, AUTOMAT
- slovo, abeceda, zřetězení, uzávěry množiny, formální jazyk
- deterministický konečný automat
- nedeterministický konečný automat
- přechodová funkce, zobecněná přechodová funkce
- jazyk rozpoznávaný a jazyk rozpoznatelný konečným automatem
- reprezentace KA: výčet matematické struktury, tabulka, stavový diagram, strom

Úkoly a otázky k textu:

1. Je deterministický konečný automat speciálním případem nedeterministického nebo naopak?
2. Můžete pro konečný jazyk napsat vždy konečný automat?
3. Sestrojte a zapište všemi způsoby, které jste se naučili konečný automat reprezentující automat na jízdenky s následujícími vlastnostmi:
   - přijímá mince v hodnotě 1 Kč, 2 Kč, 5 Kč
   - vydává jízdenky, buď pro děti za 3 Kč nebo za 7 Kč pro dospělé

3. Deterministické a nedeterministické konečné automaty

Cíl:

Po prostudování této kapitoly si uvědomíte:

- rozdíl mezi DKA a NKA

Naučíte se:

- vytvářet automaty pro zadané jazyky
- zjišťovat jaké jazyky automaty rozpoznávají
- převádět NKA na DKA
- vlastnosti jazyků rozpoznatelných DKA a NKA a jejich dokazování

V předchozí kapitole jsme se seznámili s konečným automatem a jeho deterministickou a nedeterministickou verzí. Viděli jsme, že rozdíl mezi nimi spočívá především v možnosti přecházet z jednoho stavu do více stavů (nebo žádného) na symbol u NKA. Každý NKA lze ale pomocí postupu, který se naučíte, převést na deterministický. Pokud jste se zamýšleli nad úkolem na konci kapitoly, dospěli jste zřejmě k poznání, že „žárovky“ (jak jsme velmi zjednodušeně pojmenovali stavy) budou v případě automatu z příkladu svítit současně. Co z toho ale vyplývá? Asi

Vás také napadne, že bychom mohli nahradit několik svítících žárovek jednou, která by svítila za určitou rozsvícenou část žárovek – tím bychom mohli odstranit všechny možné kombinace rozsvícených žárovek v původním NKA a získat tím automat, který už nebude obsahovat více svítících žárovek najednou. Prostě pokud máme jako v příkladě aktivní stavy q_1 i q_2 , pak je nahradíme v novém automatu stavem $\{q_1,q_2\}$, který bude tuto situaci reprezentovat a přitom bude novým jedním stavem – tedy je to jakýsi makrostav, který může obsahovat více možných aktivních stavů z výchozího NKA.

3.1. Konstrukce automatů pro zadané jazyky

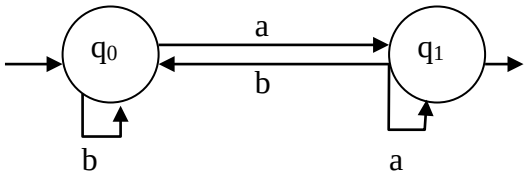
Základním úkolem je pro Vás sestrojení automatu, který rozpoznává jistý jazyk. Samozřejmě jsou i jazyky, pro které to nelze provést, ale nyní se soustřeďme na jednoduché jazyky. Uvidíte, že mnohdy je mnohem jednodušší sestrojit NKA pro zadaný jazyk než DKA. U DKA totiž musíte do všech detailů promyslet, na jaký stav se má přejít na všechny symboly. Zatímco u nedeterministického automatu se můžete soustředit jen na to „co má dělat“ a nemusíte promýšlet, co se má stát, když automat „dělá něco co dělat nemá“. Podívejme se na příklad, který to vysvětlí:

Řešený příklad 9:

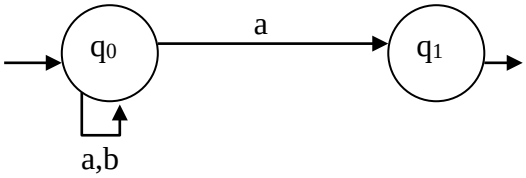
Mějme jazyk L={w ∈ {a,b}*|w končí symbolem a}. Navrhněte konečný automat, který rozpoznává jazyk L.

Výhody NKA

Pokusme se nejprve navrhnout deterministický automat:



A nyní nedeterministický automat:



Vidíte, že nedeterministický automat je mnohem jednodušší. Ve stavu q_0 načítá libovolný symbol a na konci slova „uhodne“, že má přejít do q_1, jen pokud je poslední symbol ‚a‘. V tom spočívá síla nedeterminismu, i když z algoritmického hlediska je tato situace nepřípustná. Naučíme se však převádět jakýkoliv nedeterministický automat na deterministický a díky tomu budete schopni navrhovat automaty i pro velmi složité problémy.

Naproti tomu deterministický automat je mnohem složitější. Ve stavu q_0 se cyklí symbol ‚b‘, neboť slovo má končit na ‚a‘. Pokud je nalezen symbol ‚a‘, přejde se do koncového stavu q_1. Jenže co když ještě nejde o poslední symbol? Pak je třeba se vrátit buď se vrátit do q_0, pokud přišel symbol ‚b‘ nebo setrvat v koncovém stavu, přišel-li symbol ‚a‘.

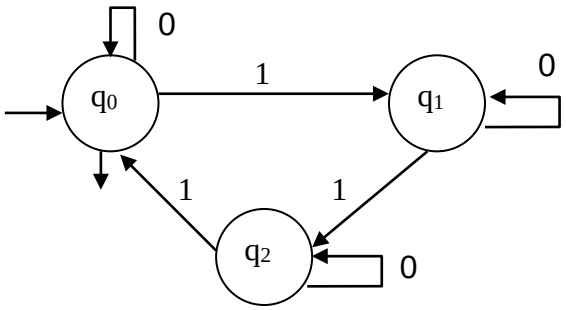
V této kapitole se pokusíme rozebrat některé vybrané úlohy, se kterými se můžete potkat při konstrukci automatu. Budeme je navrhovat jak deterministicky, tak nedeterministicky. V příští podkapitole si ukážeme, jak se dá každý NKA na DKA převést. Když na cvičení pracujeme se studenty prezenčního studia na těchto příkladech, stává se, že někteří studenti jdou „cestou nejmenšího odporu“ a navrhnou si nejprve NKA, který pak tímto postupem převedou. Někteří naopak nad problémem dlouho uvažují a navrhnou rovnou DKA (což je těžší). Někteří to zkoušejí a nejde-li jim to, vydají se jednodušší cestou. Nemohu Vám dát přesný recept, který způsob používat. Záleží to na Vašem způsobu myšlení, trpělivosti – je to spíše psychologická otázka. Mohu Vám říct, že já sám většinou jednoduché příklady napíši přímo jako DKA, ale ve složitějším případě se mi časově lépe osvědčuje navrhnou jednodušeji NKA a ten si převést. Opravdu záleží na konkrétním případě. Na druhou stranu pokud se pokouším přímo navrhnout DKA, je to velmi dobré mentální cvičení – rozvíjí schopnost prozkoumat možné situace, do kterých se dostane automat a ošetřovat je.

Řešený příklad 10:

Řešme problém, jak sestrojit automat, který rozpoznává jazyk z abecedy {0,1}, přičemž slova z jazyka obsahují počet symbolů 1, který je dělitelný

3 nebo 0. Tedy L = {ε, 0, 00, 000, ..., 010101, 111, 0111,001101, 111111 atd.....}

Takový automat by měl vyjít z počátečního stavu, který by měl být zároveň výstupní (slovo nemusí obsahovat žádný symbol), také nás však nezajímá kolik je ve slově symbolů 0, takže v tomto stavu na symbol 0 můžeme zůstávat. Pokud přijde symbol 1, pak takové slovo už neobsahuje počet dělitelný 3. Proto musíme přejít do stavu jiného (nekoncového). To samé platí, i pokud se vyskytne další 1. Přijde-li však třetí symbol 1, pak je toto slovo opět z jazyka a automat by měl přejít do stavu koncového. Jelikož je však zbytečné toto řešit novým stavem (je to stejná situace jako na začátku), vrátíme se do počátečního stavu a celý průběh se může opakovat do nekonečna. Během celé činnosti (výpočtu) automatu „ignorujeme“ symboly 0, protože jejich počet je nedůležitý. Ignorováním se myslí, že se na něj nemění stav. A nyní jak se tato úvaha prakticky realizuje:



3.2. Algoritmus převodu NKA na DKA – stromový algoritmus

Abychom mohli provádět převody automatů, které vytvoříme jako nedeterministické máme k dispozici přesný postup, jak kterýkoliv NKA převést na DKA. Spočívá přesně na principu, který jsme již zmiňovali. Tedy vytváříme z původního automatu podmnožiny stavů, do kterých se lze dostat na určitý symbol. To znamená, že máme li automat, který se dostane z q_0, jak do q_0 i q_1, pak vytvoříme na tento symbol podmnožinu $\{q_0, q_1\}$. Takto konstruujeme vlastně strom, jenž nám pak reprezentuje nový automat, který je již deterministický. Pozor! Tento stromový algoritmus budeme používat s mírnými obměnami i u dalších převodů, jako je sjednocení či průnik automatů. Věnujte mu proto velkou pozornost.

Algoritmus (stromový):

Na tento převod lze použít podmnožinovou stromovou konstrukci. Máme nedeterministický automat $A_1=(Q_1,\Sigma,\delta_1,I,F_1)$. Chceme sestrojit

Stromový algoritmus NKA -> DKA

deterministický automat $A_2=(Q_2,\Sigma,\delta_2,q_0,F_2)$, který bude pro každou dvojici stav-symbol obsahovat právě jeden přechod narozdíl od A_1.

Proces převodu:

1. Vytvoříme množinu, která bude kořenem stromu a bude obsahovat všechny stavy z I (všechny počáteční stavy automatu A_1).
2. Sestrojíme větev s označení symbolu a uzel A_i, pro každý symbol abecedy (tedy uzlů a větví bude tolik, kolik je symbolů abecedy). Obsah každého uzlu vytvoříme tak, že vezmeme všechny stavy z nadřazeného uzlu a zjistíme, kam mohou na daný symbol abecedy přecházet. Výsledný uzel tedy bude opět podmnožina vzniklá sjednocením všech těchto přechodů.
3. Postup z bodu 2. aplikujeme na všechny nově vzniklé uzly, které bude považovat za nadřazený uzel (podobně jako ten z bodu 1.) a vznikat tak bude vždy nový podstrom celého stromu, který vychází z kořene. Výjimkou jsou podmnožiny, které se již někde ve stromě vyskytují. Na tyto již se vyskytující podmnožiny se nebude znova aplikovat bod 2., ale tyto se stanou koncovými uzly (listy stromu).
   Pozn.: Pozor! Podmnožina je jednoznačně určena pouze svými prvky, nikoliv jejich pořadím. např. $\{q_1,q_3,q_5\}$ a $\{q_3,q_1,q_5\}$ jsou stejné množiny!
4. Celý postup vytváření stromu je konečný, protože množina má konečně mnoho prvků (automat je konečný!) a tedy i množina všech podmnožin je konečná. Vytváření uzlů podle bodu 2. skončí, když již nevznikne žádná nová podmnožina.
5. Po vytvoření stromu označíme každou podmnožinu symboly $q_{1'},...,q_{k'}$, které budou stavy automatu A_2. Označení musí být jednoznačné (tedy každá unikátní podmnožina má různé označení než všechny ostatní). Počátečním stavem A_2 bude podmnožina, která je kořenem stromu. Koncovými stavy A_2 budou ty podmnožiny, které obsahují některý z koncových stavů automatu A_1.
6. Sestavíme přechodovou funkci, takže ve stromu zjistíme všechny možné dvojice – nadřazená podmnožina $q_{i'}$, větev se symbolem a,

k ní přísluší podřízená podmnožina na dolním konci větve $q_{j'}$. Z těchto údajů sestrojíme $\delta_2(q_{i'},a) = q_{j'}$.

Pozn.: Pokud při tvorbě uzlu podle bodu 2. nenajdeme žádný přechod na žádný ze stavů v nadřazené podmnožině, pak vzniká prázdná množina. Tato prázdná množina je stavem, který ošetřuje chybovou situaci v automatu („zaseknutí“ nedeterministického automatu). Z prázdné množiny se logicky přechází na všechny symboly abecedy opět do prázdné množiny, která nemůže být výstupní.

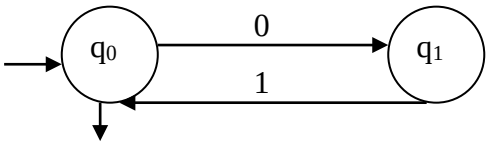
Řešený příklad 11:

Mějme NKA A_1, který rozpoznává jazyk $L = \{(01)^n , n \geq 0\}$.

$$A_1 = (\{q_1,q_2\}, \{0,1\}, \delta, q_0, \{q_0\}), \text{ kde}$$

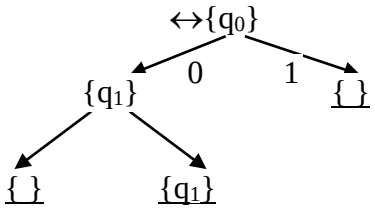
$$\delta(q_0, 0) = q_1, \delta(q_1, 1) = q_0$$

Stavový diagram NKA vypadá takto (nemá určeno kam jít na všechny



symboly a tudíž není deterministický):

Provedeme převod dle stromového algoritmu:

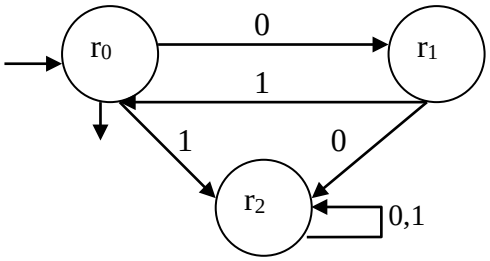


Označíme množiny takto: $\{q_0\} = r_0$, $\{q_1\} = r_1$, $\{\ \} = r_2$.

Pak $q_0 = r_0$, $F_2 = \{r_0\}$,

δ_2: $(r_0, 0) = r_1$, $(r_0, 1) = r_2$, $(r_1, 0) = r_2$, $(r_1, 1) = r_0$, $(r_2, 0) = r_2$, $(r_2, 1) = r_2$

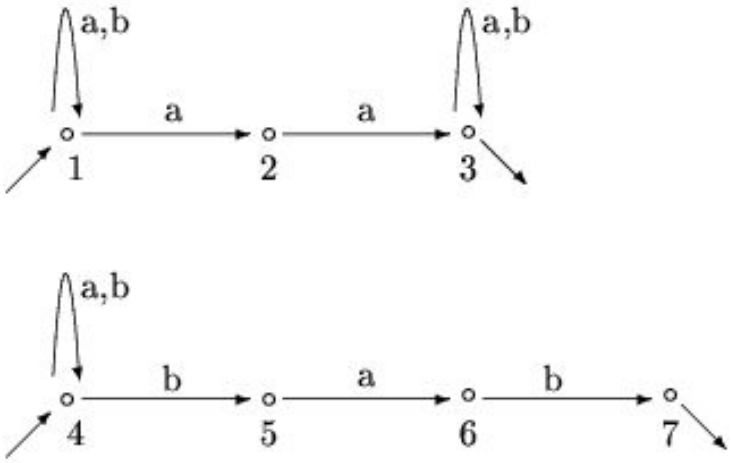
Výsledný deterministický automat zapsaný stavovým diagramem:



Řešený příklad 12:

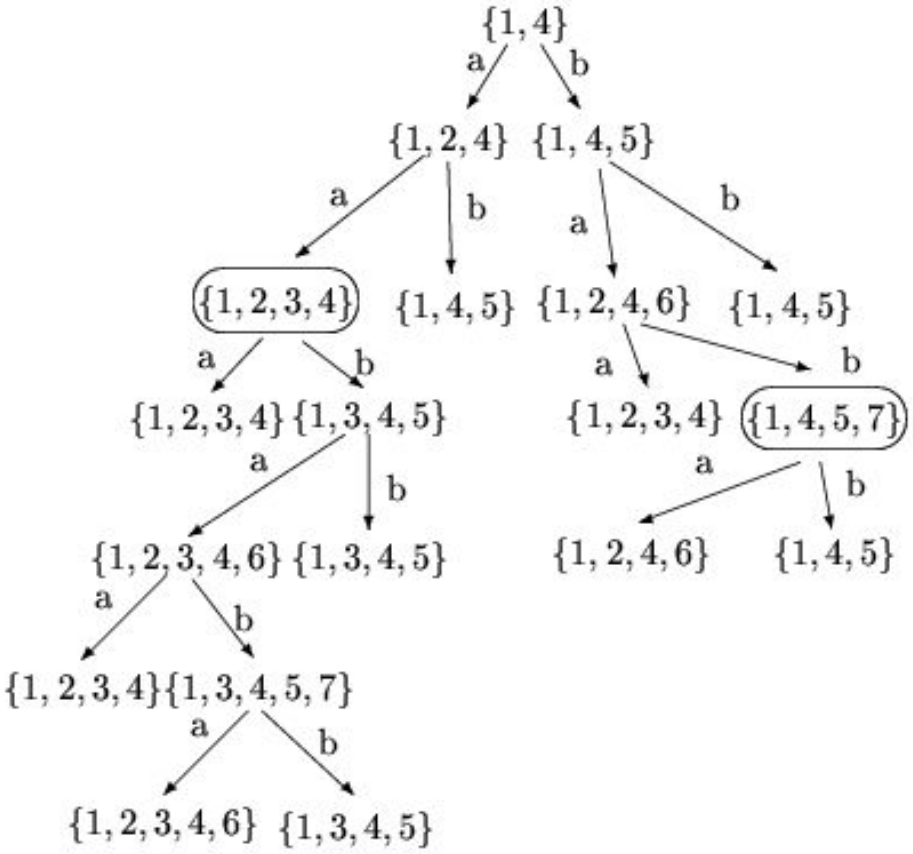
Nyní si proberme složitější příklad. Pokusme se navrhnout jednoduše nedeterministicky automat pro jazyk z abecedy {a,b}, který obsahuje slova s výskytem podslova aa nebo slova, která končí na bab. Pro jednoduchost

návrhu můžeme automat zapsat jakoby do dvou podautomatů, kde každý z nich rozpoznává slova s aa a slova končící na bab. Náš celkový automat potom má dva vstupní stavy – 1 a 4 a může si nedeterministicky „vybrat“ – „uhádnout“, kterým podautomatem se má vydat. Tento NKA je na následujícím obrázku.



Nyní s pomocí stromového algoritmu převodu na DKA vytvoříme deterministickou verzi.

Postupným procházením podmnožin jsme vytvořili strom DKA. Nyní



můžeme tento strom přepsat do libovolné reprezentace KA, např. do tabulky:

Označíme podmnožiny následovně:

A={1,4},B={1,2,4},C={1,4,5},D={1,2,3,4},E={1,2,4,6},F={1,3,4,5},

G={1,4,5,7},H={1,2,3,4,6},I={1,3,4,5,7}

Výstupní stavy jsou ty, které obsahují alespoň jeden výstupní stav z původního NKA: D,F,G,H,I

	$Q\backslash^{\Sigma}$	a	b
→	A	B	C
	B	D	C
	C	E	C
←	D	D	F
	E	D	G
←	F	H	F
←	G	E	C
←	H	D	I
←	I	H	F

Úkol k textu: Přepište tento DKA do formy stavového diagramu a proveďte výpočet podle přechodové funkce pro slova – baaaa, abba, baba, abab a zkontrolujte – zdůvodněte, že automat opravdu rozpoznává stejný jazyk jako NKA.

Nejdůležitější probrané pojmy:

- stromový algoritmus převodu NKA na DKA
- ekvivalence jazyků rozpoznatelných NKA a DKA + důkaz

Kontrolní otázka:

Může existovat jazyk, který rozpoznává nějaký nedeterministický konečný automat, ke kterému bychom nemohli sestrojit deterministický konečný automat?

Řešení:

Takový jazyk existovat nemůže, neboť známe postup jak každý NKA převést na DKA.

Úkol k textu:

Sestrojte DKA rozpoznávající jazyk $L=\{w \in \{a,b\}^{*} | w$ končí symbolem ‚a' nebo obsahuje ‚bab'$\}$.

Korespondenční úkol:

Část 1:

Mějme jazyk L1, který je tvořen aritmetickými výrazy s operacemi sčítání a násobení a jediným možným operandem, kterým je symbol X. Jazyk L1 nepřipouští používání závorek tedy do jazyka patří např. výraz X*X+X do jazyka patří a výraz (X+X)+X do jazyka nepatří.

Dále mějme jazyk L2, který je tvořen aritmetickými výrazy s operací sčítaní a jediným operandem X. Jazyk L2 připouští používání závorek. Např. výraz (X+X)+X patří do jazyka a výraz X+X*X nepatří.

a. popište slovně, co je průnikem jazyků L1 a L2 (L1 ∩ L2)
b. popište slovně, co je sjednocením jazyků L1 a L2 (L1 ∪ L2)
c. patří všechny následující výrazy do L1 ∩ L2: X+X+X, X+X+(X+X), (X*X)+X?
d. nepatří některý z následujících výrazů do L1 ∩ L2: X+X, X+(X+X)+X, X*X?
e. který z jazyků by se musel obohatit o jednu operaci a jakou, aby výraz X*X+X patřil do L1 ∩ L2?

Část 2:

Navrhněte (jakýmkoliv postupem) konečné automaty, které rozpoznávají následující jazyky:

a. L1 = {w; w $\in$ {0,1}, w obsahuje lichý počet symbolů 0 a zároveň sudý (i nulový) počet symbolů 1}
b. L2= {w; w $\in$ {0,1}, w obsahuje posloupnost 011}

4. Regulární jazyky, výrazy a aplikace

Cíl:

Po prostudování této kapitoly pochopíte:

- Co jsou regulární výrazy a jak se vztahují ke konečným automatům
- Co jsou regulární jazyky

Naučíte se:

- Vytvářet regulární výrazy k regulárním jazykům
- Převádět regulární výrazy na automaty

V kapitolách minulých jste se seznamovali s automaty – tedy nástroji, které rozpoznávají jazyky (umožňují Vám zjistit, zda slovo do jazyka patří). Jinak řečeno jsme analyzovali konkrétní jazyk pomocí automatu. Je však možné se na tento problém podívat z jiného – syntetického – hlediska. To znamená, že budeme chtít vytvářet (generovat) jazyk. K tomu nám slouží (kromě gramatik, které budeme studovat v druhé části) regulární výrazy, které generují regulární jazyky. Můžete si je představit jako jakýsi předpis (šablonu), podle které jsou slova z tohoto výrazu tvořena. Jelikož jsme v kapitolách 1 – 4 poměrně solidně pokročili s výkladem a mnohé jsme již naznačili pevně věřím, že Vám tato kapitola nebude dělat velké problémy. Pokusme se již nyní dát jednoduchý příklad takového výrazu:

Příklad:

Výraz (a + b)*b generuje jazyk L složený ze slov, která obsahují libovolnou kombinaci symbolů ‚a' a ‚b' a končí na symbol b. Vidíte, že operace, které se ve výrazu používají již znáte (kromě +). (a+b)* je zřetězeno s ‚b'. Operace * je iterací a + neznamená nic jiného než variantu – ‚a' nebo ‚b'.

L = {b,ab,bb,aab,abb,bab,bbb,...}

Regulární výrazy nejsou opět pouze teorií bez významu pro praxi. Uvědomte si, že v mnoha prostředcích, které používáte jsou implementovány. Ve vstupech tabulkových procesorů, databází se setkáte se vstupními filtry, které umožňují kontrolu, zda je například správně definováno datum v různých tvarech (DD/MM/RRRR, RRMMDD, atd.). Nebo například číslo s desetinnou čárkou lze definovat regulárním výrazem.

Příklad:

Jazyk L čísel s desetinnou tečkou lze intuitivně definovat takto:

(‚0' + ‚1' +...+ ‚9')(‚0' + ‚1' +...+ ‚9')*'.'(‚0' +...+ ‚9') (‚0' +...+ ‚9')*

tedy L={0.1, 0.23,123.456,.....}

Tedy tento výraz definuje číslo složené na začátku alespoň z jedné číslice (nebo více), pak následuje desetinná tečka a pak opět alespoň jedna číslice (nebo více).

4.1. Regulární jazyky a výrazy

Regulární jazyky

Regulární jazyk je takový, který je vytvořen ze základních symbolů abecedy pouze s pomocí operací sjednocení, zřetězení a iterace (postupnou aplikací v libovolném počtu a pořadí). Cítíte asi, že to přesně koresponduje s operacemi, které se používají ve výše zmíněných výrazech. Proto regulární výrazy generují právě regulární jazyky. Definujme je nyní exaktně:

Definice 12: ***Třída*** **RJ(Σ)** ***regulárních jazyků v konečné abecedě*** **Σ je nejmenší třída jazyků v abecedě Σ, která obsahuje jazyky ∅ a { a} pro všechna a ∈ Σ, a je uzavřena na tzv. regulární operace, tj. operace ∪, ·, * (∪ sjednocení,· zřetězení,* iterace). Tedy pro lib. L_1,L_2 platí**

$$L_1, L_2 \in RJ(\Sigma) \Rightarrow L_1 \cup L_2 \in RJ(\Sigma)$$

$$L_1, L_2 \in RJ(\Sigma) \Rightarrow L_1 \cdot L_2 \in RJ(\Sigma)$$

$L_1 \in RJ(\Sigma) \Rightarrow L_1^* \in RJ(\Sigma)$

Regulární výrazy slouží k přehlednějšímu zápisu regulárních jazyků.

Příklad:

$\{e\} \in RJ(\Sigma)$

Důkaz:

$\varnothing \in RJ(\Sigma)$

$L \in RJ(\Sigma) \Rightarrow L^* \in RJ(\Sigma)$

$\varnothing^* = \varnothing^+ \cup \{e\} = \{e\}$

Definice 13: ***Třídu*** **RV(Σ)** ***regulárních výrazů nad abecedou*** **$\Sigma = \{a_1, a_2, \ldots a_n\}$ definujeme jako nejmenší množinu slov v abecedě $\{a_1, a_2, \ldots a_n, \varnothing, e, +, \cdot, *, (,)\}$, $\varnothing, e, +, \cdot, *, (,) \notin \Sigma$ splňující následující podmínky:**

$\varnothing \in RV(\Sigma)$, $e \in RV(\Sigma)$, $a \in RV(\Sigma)$ pro všechna $a \in \Sigma$

Regulární výrazy

$\alpha, \beta \in RV(\Sigma) \Rightarrow (\alpha + \beta) \in RV(\Sigma)$

$\alpha, \beta \in RV(\Sigma) \Rightarrow (\alpha \cdot \beta) \in RV(\Sigma)$

$\alpha, \beta \in RV(\Sigma) \Rightarrow (\alpha^*) \in RV(\Sigma)$

Každý regulární výraz označuje (reprezentuje) konkrétní regulární jazyk.

$\varnothing$ označuje jazyk $\varnothing$

e označuje jazyk $\{e\}$

a označuje jazyk $\{a\}$ pro libovolné $a \in \Sigma$

Jestliže regulární výraz α označuje L_1, a regulární výraz β označuje L_2, pak

$(\alpha+ \beta)$ označuje jazyk $L_1 \cup L_2$

$(\alpha \cdot \beta)$ označuje jazyk $L_1 \cdot L_2$

α^* označuje jazyk $(L_1)^*$

Obecně budeme jazyk reprezentovaný regulárním výrazem α značit $[\alpha]$. Budeme vynechávat zbytečné závorky (např. vnější pár, zbytečné závorky vzhledem k asociativitě operací $\cup$, $\cdot$), tečky ($\cdot$), další závorky můžeme vynechávat na základě priorit operací. * má větší prioritu než $\cdot$ a ta má větší prioritu než +.

Procvičte si pochopení těchto pojmů nyní na konstrukci výrazů na těchto řešených příkladech:

Řešený příklad 13:
L={w $\in$ {0,1}*|w obsahuje podslovo 101 nebo končí podslovem 00 předcházeným libovolným počtem trojic 011}.
$\alpha = (0+1)^* 101(0+1)^* + (011)^* 00$
$L=[\alpha]$

Řešený příklad 14:
L={w $\in$ {0,1}*|w=uv, u obsahuje sudý počet symbolů 0, v obsahuje slovo 11}).

Regulární výraz popisující tento jazyk:

$(1^*01^*01^*)^* (1+0)^*11(0+1)^*$

Vidíte, že tento výraz rozděluje slova na dvě části – první řeší sudý počet nul $(1^*01^*01^*)^*$ a druhá podslovo 11 $(1+0)^*11(0+1)^*$.

Řešený příklad 15:

L={w ∈ {0,1}*|w=uv, u končí symbolem 1, v obsahuje slovo 11}).

Regulární výraz popisující tento jazyk:

$(1+0)^*1(1+0)^*11(0+1)^*$

Na příkladech jste viděli, že jsou velmi podobné konstrukci automatů. V podstatě regulární výraz je laicky řečeno jen jiným způsobem, jak popisovat stejnou třídu jazyků. Tento fakt je ale třeba přesně exaktně formulovat a dokázat. Uvidíte, že to není nijak složité. Formulujeme si větu, která říká, že ke každému výrazu lze sestrojit automat a naopak, že automat rozpoznává jazyk, který je regulární. Jde o velmi zásadní vlastnost v teorii formálních jazyků, proto prosím věnujte jak tvrzení (Kleeneho věta), tak důkazu patřičnou pozornost. Důkaz je opět konstruktivní a jeho postup nám umožní formulovat i algoritmus v další podkapitole, díky němuž budete schopni ke každému výrazu sestrojit konečný automat naprosto automaticky. Tvrzení se dá rozdělit do dvou vět.

Věta 1: Každý regulární jazyk je rozpoznatelný konečným automatem.

Věta 2: Každý jazyk rozpoznatelný konečným automatem je regulární.

Důkaz:

Obě tato tvrzení pak dohromady tvoří Kleeneho větu:

Věta 3: (*Kleene*) Libovolný jazyk je regulární, právě tehdy když je rozpoznatelný konečným automatem.

Kleeneho věta

4.2. Sestrojení automatu (ZNKA) k regulárnímu výrazu

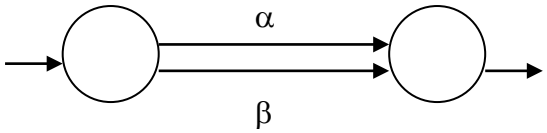
Na sestrojení automatu k regulárnímu výrazu můžete použít dva přístupy. První spočívá v postupné dekompozici výrazu na menší části podle regulárních operací a sestrojování schémat (připomínajících automaty), která nemusí mít ve svých přechodech pouze symboly, ale celé části výrazu. Až pak dojdeme na symboly, máme k dispozici zobecněný nedeterministický automat, který již můžeme převést na deterministický známými postupy.

Algoritmus konstrukce automatu k regulárního výrazu (rozklad):

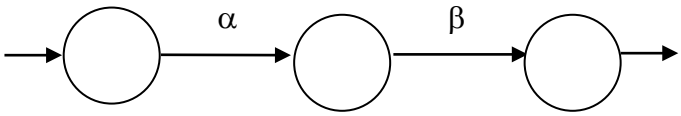
1. Mějme zadán výraz γ
2. K výrazu sestrojíme schéma podle jeho struktury:

- je-li $\gamma = (\alpha + \beta)$, pak

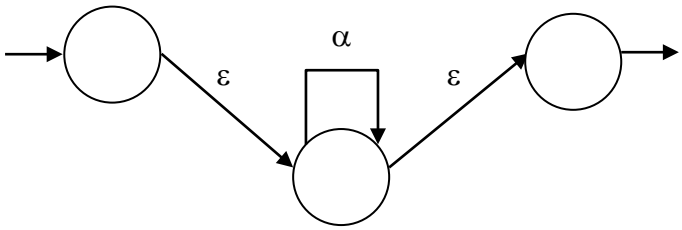
Konstrukce ZNKA k výrazu



- je-li $\gamma = (\alpha \cdot \beta)$, pak



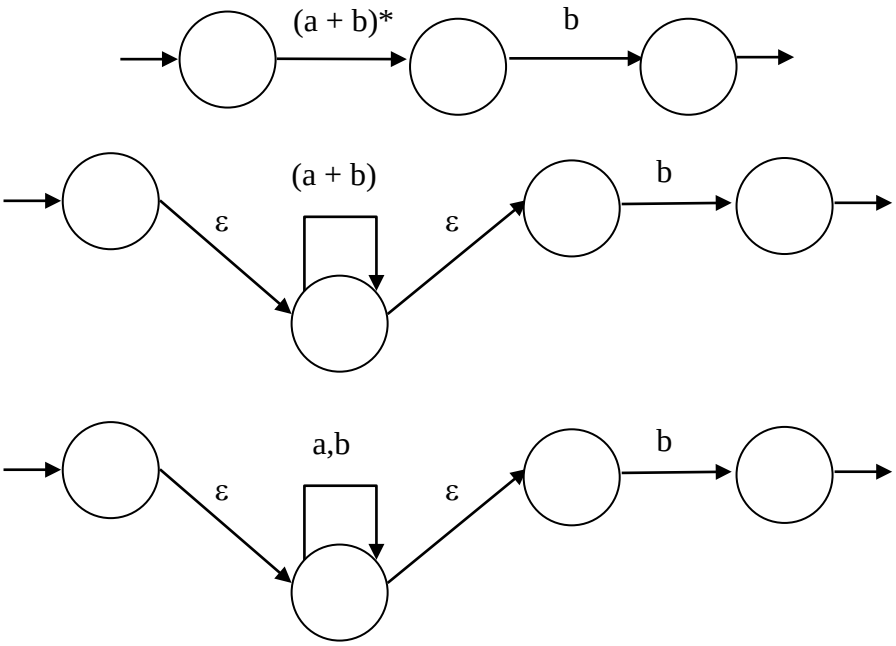
- je-li $\gamma = (\alpha)^*$, pak



3. Postup z bodu 2. aplikujeme na každou nově dekomponovanou část výrazu až jsou všechny přechody ve schématu pouze symboly abecedy

Řešený příklad 16:

Mějme výraz (a + b)*b. K němu postupnou dekompozicí sestrojíme



ZNKA:

Na základě důkazu tvrzení, že ke každému regulárnímu výrazu existuje automat můžeme sestavit algoritmus pro takovou konstrukci. Tento postup je opačný k dekompozici podle algoritmu uvedeného výše.

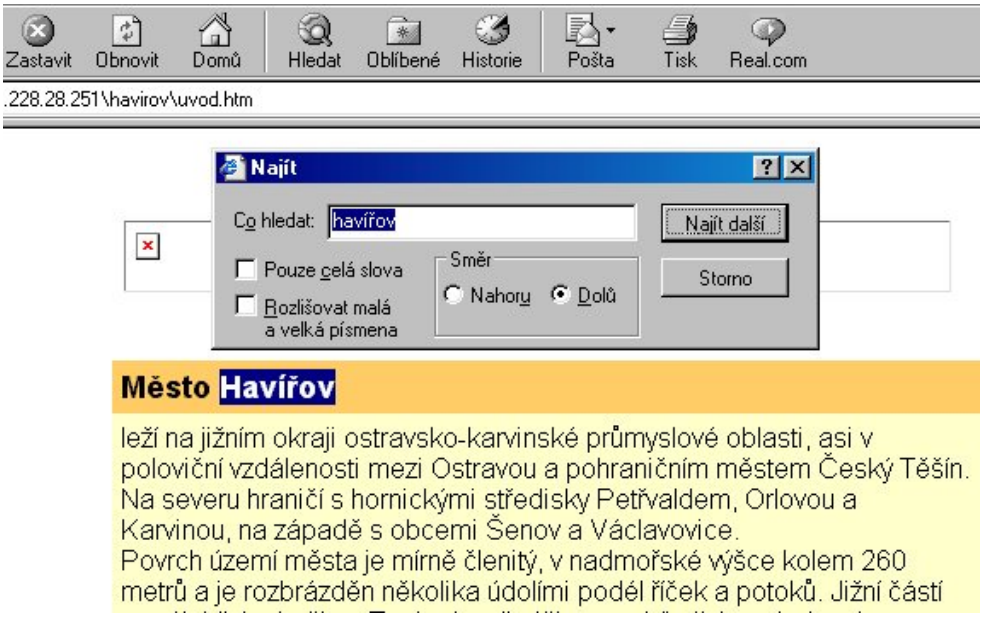
4.3. Regulární jazyky a konečné automaty v praxi

Nejjednoduššími jazyky, které zkoumá teorie formálních jazyků, jsou jazyky regulární (resp. jazyky rozpoznatelné konečnými automaty). I

když tyto pojmy zní odtažitě, podívejme se na jednu úlohu, kterou asi řešil každý čtenář tohoto článku a tou je vyhledávání v textu.

Intuitivní příklad:

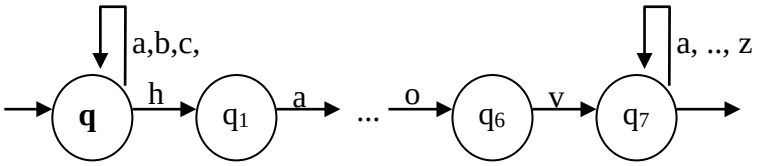
V Internetovém prohlížeči (např. MSIE) realizujeme vyhledávání



slov začínajících na „Havířov“ na webové stránce.

Algoritmy, které postupně načítají text a hledají výskyt slova, samozřejmě využívají knihoven funkcí. Tyto knihovny jistě obsahují již připravené algoritmy porovnání řetězců apod. Přesto půjdeme-li až na jádro způsobu nalezení slova bez použití těchto pomůcek, musí se číst postupně znaky textu a srovnávat – jde o první písmeno hledaného slova ‚h‘? Pokud ano, dále srovnávej zda souhlasí následující písmena... Z hlediska teorie formálních jazyků, jde o vyhledávání regulárního výrazu,

který můžeme realizovat pomocí konečného automatu. Podívejme se na



příklad:

Pro náš uvedený příklad bychom mohli sestrojit automat na obrázku. Takový automat by byl nedeterministický, což znamená, že by nebylo jednoznačně určeno, do kterého stavu se má jít z q_0. Automat totiž „neví“ zda při přečtení písmena ‚h‘ má jít do q_1 nebo má zůstat ve stejném stavu. Právě formalizace, kterou dává TFJA, však umožňuje používat algoritmy, které by uvedený nedeterminismus odstranily. Vznikl by automat deterministický – tedy takový, který má vždy jednoznačně určeno, do kterého stavu při čtení určitého znaku z textu by přešel. Deterministický automat lze simulovat v programovacím jazyce, což je právě ona praktická realizace teoretických výsledků TFJA.

Příklad, který jsme pravě prezentovali, je samozřejmě velice jednoduchý. Konečné automaty mají větší možnosti než jen rozpoznávání pevných řetězců. Regulární výrazy, které k nim přísluší, mohou nahrazovat části slov libovolnými kombinacemi apod.

Nejdůležitější probrané pojmy:

- regulární jazyk, regulární výraz
- ekvivalence regulárních jazyků a jazyků rozpoznatelných konečnými automaty (Kleeneho věta + důkaz)

- konstrukce ZNKA k regulárnímu výrazu

Úkoly a otázky k textu:

4. Sestrojte regulární výraz rozpoznávající jazyk
L={w ∈ {a,b}*|w končí symbolem ‚a' nebo obsahuje ‚bab'}.
5. Převeďte výraz z úkolu 1 na DKA.
6. Lze sestrojit regulární výraz ke každému konečnému automatu?

5. Bezkontextové gramatiky a jazyky, regulární gramatiky

Cíl:

Po prostudování této kapitoly pochopíte:

- co je bezkontextová gramatika
- jak pojem gramatiky souvisí s jazykem
- vztah regulárních gramatik k regulárním jazykům

Naučíte se:

- tvořit automaty pro jednoduché bezkontextové jazyky
- převádět regulární gramatiky na automaty a naopak

Nyní se v našem studiu dostáváme do druhé části. V předchozím díle jsme se zabývali třídou jazyků rozpoznatelných konečnými automaty resp. regulárními jazyky. Viděli jste, že existují i jazyky, které nejsou regulární. Konečné automaty jsou pro ně příliš „slabé“, nedokáží je rozpoznávat a regulární výrazy je nemohou generovat. Proto by bylo rozumné se ptát, zda neexistují nástroje, které nám umožní tyto jazyky generovat a analyzovat. Takové nástroje existují a postupně se s nimi i s jejich

vlastnostmi seznámíme a opět se naučíte vytvářet k jazykům jejich instance.

Složitější jazyky než regulární

Těmito nástroji jsou bezkontextová gramatika a zásobníkový automat. Pojem gramatiky jsme si již částečně objasnili na intuitivním příkladu. Je to prostředek, jak na základě pravidel lze generovat (terminální) slova v jisté (terminální) abecedě pomocí postupného dosazování do neterminálních symbolů (proměnných). Občerstvěte si tyto pojmy v paměti. Zásobníkový automat je konečný automat, který má však navíc možnost pracovat s jistým druhem paměťového zařízení – zásobníkem, na který si může ukládat symboly a vybírat je zpět. Nicméně může tak činit pouze přístupem – poslední dovnitř, první ven (to znamená může zapisovat jen na vrchol zásobníku – struktura LIFO, jak ji znáte z algoritmizace). Naučíte se tyto struktury používat a také si ukážeme jejich speciální tvary. Stejně jako u regulárních jazyků si pak ukážeme, že jejich výpočetní síla – třída jazyků rozpoznatelných zásobníkovými automaty a bezkontextových jazyků generovaných bezkontextovými gramatikami – je totožná.

Podívejme se nejprve na příklad, jak se s bezkontextovými gramatikami pracuje a pak si zavedeme jejich formální definice. Uváděli jsme si, že jazyk $L=\{0^n1^n;\ n \geq 0\}$ není rozpoznatelný konečným automatem. Existuje však velice jednoduchá bezkontextová gramatika, která tento jazyk generuje:

Tato gramatika má vstupní abecedu (terminálních symbolů) – {0,1}, abecedu proměnných (neterminálů) – {S}, neterminál, od kterého se generování (odvozování) vždy začíná určený jako S a tato dvě pravidla, určující možnosti „dosazení“ za proměnné:

$S \rightarrow 0S1$, $S \rightarrow \varepsilon$.

Tato pravidla umožňují buď dosadit za S řetězec 0S1 (obsahuje opět v sobě S), nebo ukončit toto generování slovem prázdným. Díky tomu, že vždy ke každé nule na levé straně slova dodáme jedničku na pravé straně slova, můžeme si takto „napumpovat“ kolik chceme nul a jedniček, ovšem jedině ve stejném počtu. Toto konečný automat ani regulární výraz neuměl. V gramatice pak můžeme provádět odvození terminálních slov (která budou všechna vytvářet jazyk), např. takto

$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$

(v posledním kroku jsme použili pravidlo na prázdné slovo, čímž jsme se zbavili S a dostali slovo složené jen z terminálů).

5.1. Bezkontextová gramatika a bezkontextový jazyk

Jak uvidíme, pojem gramatiky lze zobecnit. Tím dosáhneme i daleko větší síly než poskytuje bezkontextová gramatika. Obecný pojem gramatiky:

Gramatika G je určena konečnou množinou *neterminálů* (neterminálních symbolů - proměnných), konečnou množinou *terminálů*, která nemá společné prvky s množinou neterminálů, *počátečním neterminálem* a konečnou *soustavou* (množinou) *přepisovacích pravidel* typu $\alpha \rightarrow \beta$, (α přepiš na β), kde α,β jsou řetězece z neterminálů a terminálů, navíc $\alpha \neq \varepsilon$.

Gramatika, označme ji G, může být chápána jako čtveřice $G=(\Pi,\Sigma,S,P)$

Π je množina neterminálů,

Σ je množina terminálů, Π∩Σ = ∅

S ∈ Π je počáteční neterminál,

P je konečná množina přepisovacích pravidel.

Bezkontextová gramatika se od tohoto obecného pojmu odlišuje tím, že připouští, aby přepisovací pravidlo mělo pouze tvar X→α, to znamená že pouze můžeme přepisovat vždy jednu proměnnou na řetězec proměnných i terminálů.

Bezkontextová gramatika

Definice 14: ***Bezkontextová gramatika*** **(BKG) je určena konečnou množinou** ***neterminálů*** **(neterminálních symbolů - proměnných), konečnou množinou** ***terminálů*****, která nemá společné prvky s množinou neterminálů,** ***počátečním neterminálem*** **a konečnou** ***soustavou*** **(množinou)** ***přepisovacích pravidel*** **typu X→α, (X přepiš na α), kde X je neterminál a α je řetězec z neterminálů a terminálů.**

Bezkontextová gramatika, označme ji G, může být chápána jako čtveřice G=(Π,Σ,S,P)

Π je množina neterminálů,

Σ je množina terminálů, Π∩Σ = ∅

S ∈ Π je počáteční neterminál,

P je konečná množina přepisovacích pravidel.

V takto definované gramatice můžeme pak odvozovat slova, jak jsme viděli na příkladu.

Odvození v gramatice

Definice 15: Nechť G=(Π,Σ,S,P) je bezkontextová gramatika a nechť α,β ∈ (Π∪Σ)*.

Řekneme, že α ***se přímo přepíše na*** β podle pravidel gramatiky G, označíme $\alpha \Rightarrow_G \beta$ (nebo $\alpha \Rightarrow \beta$, pokud je zřejmé o jakou G se jedná), právě tehdy, když existuje $\gamma_1,\gamma_2,\delta \in (\Pi \cup \Sigma)^*$ a $X \in \Pi$ takové, že:

$\alpha = \gamma_1 X \gamma_2$; $\beta = \gamma_1 \delta \gamma_2$; $X \rightarrow \delta$ patří do P

Řekneme, že α ***se přepíše na*** β, značíme $\alpha \Rightarrow_G^* \beta$ (nebo $\alpha \Rightarrow^* \beta$), jestliže existuje posloupnost (*) $\gamma_0,\gamma_1,\gamma_2...\gamma_n$ prvků $(\Pi \cup \Sigma)^*$ (pro nějaké $n \geq 0$) taková, že:

$\alpha = \gamma_0 \Rightarrow_G \gamma_1 \Rightarrow_G \gamma_2 \Rightarrow_G \gamma_3 \Rightarrow_G ... \Rightarrow_G \gamma_{n-1} \Rightarrow_G \gamma_n = \beta$

Posloupnost (*) nazveme ***odvození*** (***derivace***) slova β ze slova α.

Jazyk generovaný gramatikou G, označme jej L(G), je definován následovně:

$L(G)=\{ w | w \in \Sigma^* \text{ a } S \Rightarrow_G^* w \}$

Ekvivalentní gramatika, Bezkontextový jazyk

Stejně jako u konečných automatů, můžeme definovat pojem ekvivalentních gramatik. Opět půjde o gramatiku, která generuje stejný jazyk. Příkladem budiž ekvivalentní gramatika ke gramatice v příkladu:

S→ ASB, S→ ε, A→ 0, B→ 1 (přidali jsme neterminály A,B)

Definice 16: Bezkontextové gramatiky G_1,G_2 nazveme *ekvivalentní*, právě když $L(G_1)=L(G_2)$.

Definice 17: *Bezkontextový jazyk (BKJ)* je jazyk (jazyk $L \subseteq \Sigma^*$ pro nějakou konečnou abecedu Σ) generovaný nějakou bezkontextovou

gramatikou (tedy L=L(G) pro nějakou bezkontextovou gramatiku G).

Bezkontextový jazyk tedy tvoří všechna terminální slova, která můžeme odvodit z počátečního neterminálu.

5.2. Regulární gramatiky, vztah k regulárním jazykům

Bezkontextové jazyky lze dále omezit až na přesně třídu regulárních jazyků (jinak řečeno za jistých podmínek BKG generují jazyky rozpoznatelné KA). Stačí omezíme-li pravidla BKG na taková, která přepisují neterminál na slovo terminálů a za ním následuje maximálně jeden neterminál. Taková formalizace odpovídá tomu, co rozpoznává KA. Uvidíte, jak se dá velmi jednoduše ke KA sestrojit regulární gramatika a naopak. Vychází se z toho, že můžeme stavy konečného automatu považovat za neterminály, symboly u přechodů za začátek přepisovaného slova a stav, do kterého se jde, za neterminál za terminálním slovem. Je-li stav výstupní, generování slova může skončit a proto se tento stav (neterminál) přepíše na ε.

Regulární gramatika

Definice 18: Bezkontextová gramatika G=(Π,Σ,S,P) se nazývá *regulární gramatika*, jestliže každé pravidlo v P je v jednom z tvarů $X \rightarrow wY$, $X \rightarrow w$, kde $X,Y \in \Pi$, $w \in \Sigma^*$.

***Jazyk* L nazveme *regulární*, jestliže je generován nějakou regulární gramatikou (tedy L=L(G) pro nějakou regulární gramatiku G).**

Ukážeme, že definice nekoliduje s dřívější definicí regulárního jazyka.

Věta 4: Každý jazyk rozpoznatelný konečným automatem je regulární (ve smyslu definice pomocí regulární gramatiky).

Způsob převodu KA na BKG

Pro názornost uvedeme příklad, kde A je zadán tabulkou:

	$Q\backslash\Sigma$	0	1
↔	1	1	2
	2	1	3
	3	1	3

Budeme konstruovat gramatiku G.

Nejprve označme všechny stavy automatu A např. písmeny $A_1,A_2,... A_n$, kde n je počet stavů.

V našem příkladu tedy:

	$Q\backslash\Sigma$	0	1
↔	A_1	A_1	A_2
	A_2	A_1	A_3
	A_3	A_1	A_3

Symboly $A_1,A_2,... A_n$ budou tvořit množinu neterminálů gramatiky G. Symbol označující počáteční stav, bude počátečním neterminálem gramatiky G, v našem příkladu tedy A_1. Množina terminálů gramatiky G bude totožná se vstupní abecedou automatu A ({ 0,1} v našem příkladu).

Množinu přepisovacích pravidel konstruujeme následovně.

Vezmeme symbol A_1. Dále vezmeme některý terminál, označme jej a a zjistíme stav, do kterého automat A přejde ze stavu A_1 přečtením terminálu a. Tento stav nechť je označen A_j. Pak mezi přepisovací pravidla zahrneme pravidlo $A_1 \rightarrow aA_j$.

V příkladu tedy takto vytvoříme pravidla:

$A_1 \rightarrow 0A_1$, $A_1 \rightarrow 1A_2$, $A_2 \rightarrow 0A_1$, $A_2 \rightarrow 1A_3$, $A_3 \rightarrow 0A_1$, $A_3 \rightarrow 1A_3$.

Nakonec přidáme pravidla, která umožňují smazat, neboli přepsat na prázdné slovo, všechny ty neterminály, které označují koncové stavy automatu A. V příkladu tedy přidáme jen jedno pravidlo, a to $A_1 \rightarrow e$.

Tím jsme vytváření přepisovacích pravidel, a tím i celé gramatiky G ukončili.

Všimněme si, že každému "výpočtu" automatu A znázorněnému

$A_{i0} \rightarrow^{a}{}_1 A_{i1} \rightarrow^{a}{}_2 A_{i2} \rightarrow^{a}{}_3 ... \rightarrow^{a}{}_m A_{im}$

odpovídá v gramatice G odvození

$A_{i0} \Rightarrow a_1 A_{i1} \Rightarrow a_1 a_2 A_{i2} \Rightarrow ... \Rightarrow a_1 a_2 a_3 ... a_m A_{im}$

V příkladu např.

$A_1 \rightarrow^{0} A_1 \rightarrow^{1} A_2 \rightarrow^{1} A_3$

$A_1 \Rightarrow 0 A_1 \Rightarrow 01 A_2 \Rightarrow 011 A_3$

Když si nyní uvědomíme, že počáteční neterminál gramatiky G odpovídá počátečnímu stavu automatu A, a dále, že smazat lze jedině neterminály odpovídající koncovým stavům, je zřejmé, že každý přijímající výpočet automatu A (pro nějaké slovo, označme ho w) odpovídá odvození slova w z počátečního neterminálu v gramatice G a naopak. Tím jsme se

přesvědčili, že gramatika G skutečně generuje jazyk L (rozpoznávaný automatem A).

Věta 5: Ke každé regulární gramatice existuje ekvivalentní regulární gramatika, která má pravidla pouze následující typů:
$X \rightarrow aY$, $X \rightarrow Y$, $X \rightarrow e$

Regulární gramatika -> KA

Věta 6: Každý regulární jazyk (ve smyslu definice podle regulární gramatiky) je rozpoznatelný konečným automatem.

Důkaz:

Nechť L=L(G) pro regulární gramatiku G=(Π,Σ,S,P). Podle předchozí věty, lze předpokládat, že pravidla G jsou pouze typů

$X \rightarrow aY$, $X \rightarrow Y$, $X \rightarrow e$

Sestrojme zobecněný nedeterministický konečný automat A=(Q,Σ,δ,I,F), kde Q=Π; I={S}, F={ q|(q→ e) ∈ P} a ∀q ∈ Q(=Π), ∀a ∈ Σ je δ(q,a)={ q′|(q → aq′) ∈ P} a δ(q,e)={ q′|(q → q′) ∈ P}.

Ověření L(G)=L(A) je podobné jako u důkazu věty 54.

Tento reverzibilní postup nyní demonstrujeme na tomto kontrolním úkolu:

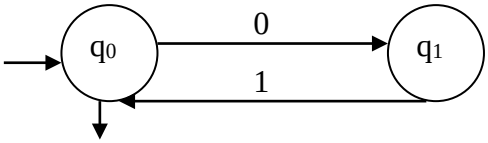
Kontrolní úkol:

Mějme regulární gramatiku:

$S \rightarrow 01S$, $S \rightarrow \varepsilon$, která rozpoznává jazyk $L = \{(01)^n , n \geq 0\}$.

Abychom mohli tuto gramatiku převést na automat, musíme nejprve postupem dle důkazu věty ji převést (na tvar $X \rightarrow aY$, $X \rightarrow Y$, $X \rightarrow e$).
$S \rightarrow 0A$, $A \rightarrow 1S$, $S \rightarrow \varepsilon$,

Automat pak vypadá takto:



Z tohoto automatu pak můžeme zpětně zase dojít k této gramatice.

Viděli jste, že BKG může mít jisté omezení pravidel, které ovlivňuje jazyky, které taková gramatiky generují. Například jazyk $L = \{ 0^n1^n , n \geq 0\}$ logicky nemůžeme generovat regulární gramatikou, neboť není regulární.

b) $G: S \rightarrow ABC$, $A \rightarrow bbA$, $A \rightarrow ccB$, $B \rightarrow bBb$, $B \rightarrow a$, $C \rightarrow aCa$, $C \rightarrow bb$.

Řešení:

$L(G)=\{w \in \{a,b,c\}^*; w=(bb)^i ccb^j ab^j b^k ab^k a^m bba^m; i,j,k,m \geq 0\}$

c) $G: S \rightarrow 001$, $S \rightarrow 01S$, $S \rightarrow 01S1$.

Řešení:

$L(G)=\{w \in \{0,1\}^*; w=(01)^j001(1)^i; i \geq 0; j \geq i\}$

Nejdůležitější probrané pojmy:

- bezkontextová gramatika, bezkontextový jazyk
- odvození
- regulární gramatika, lineární gramatika

Úkoly k textu:

1. Sestrojte DKA rozpoznávající jazyk $L=\{w \in \{a,b\}^* | w$ končí symbolem ‚a' nebo obsahuje ‚bab'$\}$ a převeďte ho na regulární gramatiku.
2. Lze každý ZNKA převést na regulární gramatiku? Zdůvodněte proč.

6. Zásobníkové automaty

Cíl:

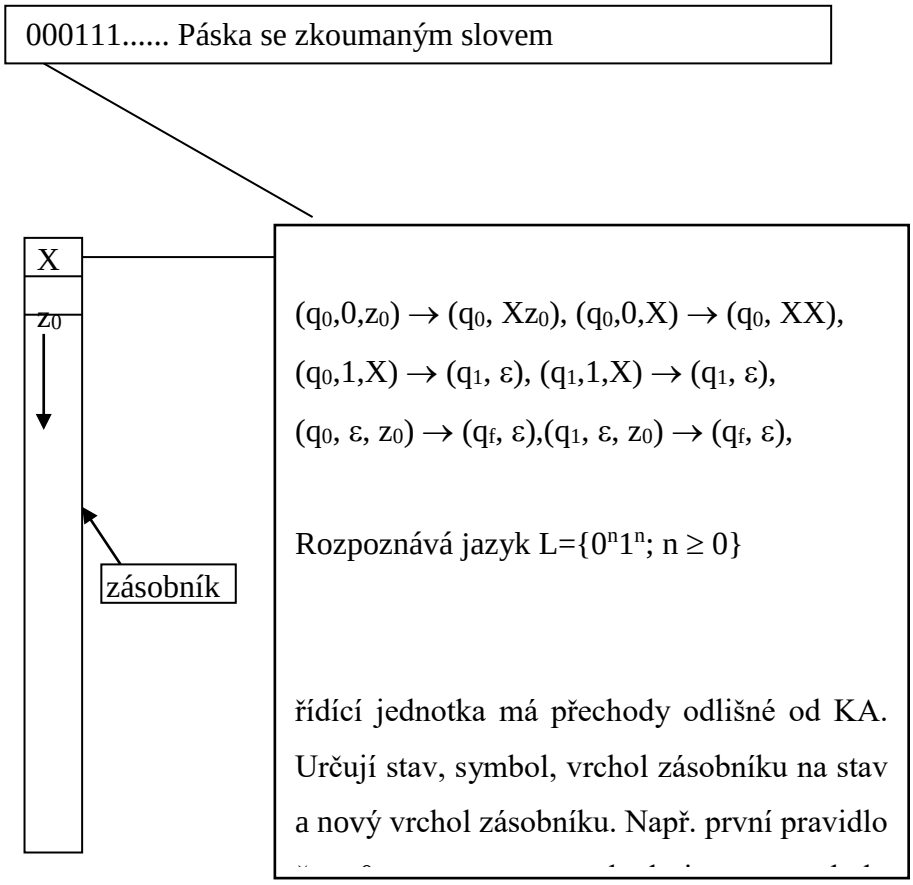
Po prostudování této kapitoly pochopíte:

- co je zásobníkový automat
- jaký jeho vztah k bezkontextovým jazykům
- co je pumping lemma (lemma o vkládání)

Naučíte se:

- vytvářet zásobníkové automaty pro zadané jazyky

Zásobníkový automat je stroj, který stejně jako konečný automat má nějakou řídící jednotku, která je vždy v nějakém ze svých stavů, a který čte ze vstupní pásky slovo (nad nějakou abecedou) a po jeho přečtení rozhodne, zda slovo patří či nepatří do jazyka, který zásobníkový automat rozpoznává. Avšak narozdíl od konečných automatů, zásobníkový automat využívá navíc zásobníku, neboli jakési paměti typu LIFO. Tedy může ukládat a vybírat symboly na vrchol zásobníku, který si lze představit jako naskládané talíře – nelze je brát odkudkoliv – pouze z vrcholu.



Idea zásobníkového automat se dá znázornit na následujícím obrázku.

6.1. Zásobníkový automat a vztah k BKJ

Zásobníkový automat

Definice 19: Zásobníkovým automatem nazveme sedmici (systém určený sedmi parametry)
***M* = (*Q*, Σ, Γ, δ, *q*0,*Z*0,*F*), kde Q je konečná neprázdná množina stavů, Σ je konečná neprázdná množina vstupních symbolů (abeceda), Γ je**

konečná neprázdná množina zásobníkových symbolů, $q_0 \in Q$ je počáteční stav, $Z_0 \in \Gamma$ je počáteční zásobníkový symbol, $F \subseteq Q$ je množina koncových stavů a δ je zobrazení množiny $Q\times(\Sigma\cup\{e\})\times\Gamma$ do množiny konečných podmnožin množiny $Q \times\Gamma^*$ (přechodová funkce). ($\delta: Q\times(\Sigma\cup\{e\})\times\Gamma \rightarrow P(Q\times\Gamma^*)$)

Z definice je patrné, že takto definovaný zásobníkový automat je nedeterministický.

Neformálně význam δ (tj. předpisu chování ZA *M*):
Je-li $\delta(q,a,X) = \{(q_1,\alpha_1),(q_2, \alpha_2), ...,(q_n, \alpha_n) \}$; $q \in Q$, $a \in (\Sigma\cup\{e\})$, $q_i \in Q, \alpha_i \in \Gamma^*$, $i \in \{1, 2, ..., n\}$,
$X \in \Gamma$, potom když *M* má čtecí hlavu na symbolu a, (konečná ŘJ) je ve stavu q a na vrcholu zásobníku je symbol X, může si *M* vybrat jedno i z {1, 2, ..., n } a posunout čtecí hlavu o jeden symbol vpravo, změnit stav řídící jednotky na q_i a symbol X v zásobníku nahradit řetězcem α_i. Speciálně je-li a=e, může *M* provést tzv. e-krok, při kterém nečte a hlava se tudíž neposunuje. Říkáme také, že *M* provedl instrukci (q,a,X) $[(\delta) \| (\rightarrow)]$ (q_i,α_i).
Důležitá je i skutečnost, že mohou existovat $q \in Q$, $a \in (\Sigma\cup\{e\})$, $X \in \Gamma$ tak, že $\delta(q,a,X)=\emptyset$ (v jistých situacích tedy nemůže automat pokračovat ve výpočtu). Při definici konkrétní přechodové funkce budeme definici obrazu pro takovéto vzory ((q,a,X)) vynechávat.

Konfigurace

Definice 20: Mějme ZA $M = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$. Situací (konfigurací) zásobníkového automatu M nazveme trojici (q, w, α), kde $q \in Q$, $w \in \Sigma^*$ a $\alpha \in \Gamma^*$. q je stav ŘJ, w je slovo (ta část slova) na vstupní pásce, která zbývá přečíst, α je obsah zásobníku. (Nejlevější symbol v α představuje *vrchol zásobníku*). Jestliže $(q', \alpha) \in \delta(q,a,X)$, pak pro lib. $w \in \Sigma^*$, $\beta \in \Gamma^*$ vede situace $(q, aw, X\beta)$ bezprostředně k situaci $(q',w,\alpha\beta)$, symbolicky značíme:

$(q,aw,X\beta) \Rightarrow (q',w,\alpha\beta)$

Nechť E a E′ jsou situace ZA M, pak řekneme, že E vede k situaci E′, značíme $E \Rightarrow^* E'$, jestliže existují situace $E_1, E_2,..., E_n$ tak, že $E=E_1 \Rightarrow E_2 \Rightarrow ... \Rightarrow E_n=E'$. Je-li potřeba, značíme o jaký ZA se jedná:

$\Rightarrow_M$ $\Rightarrow_M^*$

Narozdíl od konečného automatu může ZA rozpoznávat slova nejen tím, že skončí v koncovém stavu, ale také tím, že vyprázdní celý svůj zásobník. Například ilustrace na počátku kapitoly rozpoznává daný jazyk jak prázdným zásobníkem, tak i koncovým stavem.

Rozpoznávání jazyka zásobníkovým automatem budeme definovat dvěma způsoby:

1) přijímání koncovým stavem: slovo w je přijato ZA, jestliže existuje možnost, že po zpracování (přečtení) slova w se automat ocitne v koncovém stavu.

2) přijímání prázdným zásobníkem: slovo w je přijato ZA, jestliže existuje možnost, že po zpracování slova w se ZA ocitne v situaci s prázdným zásobníkem.

Rozpoznávání - koncovým stavem a prázdným

Definice 21: **Mějme ZA $M = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$. Definujme**
$L_{KS}(M)=\{w \in \Sigma^* | (q_0, w, Z_0) \Rightarrow_M^* (q,e,\alpha)$ pro nějaké $q \in F$ a $\alpha \in \Gamma^*\}$.

$L_{PZ}(M)=\{w \in \Sigma^* | (q_0, w, Z_0) \Rightarrow_M^* (q,e,e)$ pro libovolné $q \in Q\}$.

Deterministický ZA

Definice 22: **ZA $M = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$ nazveme deterministický (DZA), jestliže platí následující dvě podmínky:**

1. $\delta(q, a, X)$ je nejvýše jednoprvková množina pro lib. $q \in Q$, $a \in (\Sigma \cup \{e\})$, $X \in \Gamma$.

2. Jestliže $\delta(q,e,X) \neq \emptyset$ pro něj. $q \in Q$, $X \in \Gamma$, pak $\delta(q,a,X)=\emptyset$ pro lib. $a \in \Sigma$.

Definice 23: **Jazyky rozpoznatelné DZA koncovým stavem nazveme deterministické (třídu těchto jazyků označíme Det). Jazyky rozpoznatelné DZA prázdným zásobníkem nazveme bezprefixové deterministické (třídu těchto jazyků označíme BDet).**

Poznámka: Dá se ukázat, že Det je vlastní podtřída třídy bezkontextových jazyků.

(Např. jazyk $\{ww^R | w \in \{a,b\}^*\}$ není deterministický.)

(Srovnejte se situací u konečných automatů).

Řešený příklad 17:

Sestrojte zásobníkový automat, který rozpoznává jazyk $L=\{w(w)^R;\ w \in \{0,1\}^*\}$ (prázdným zásobníkem).

Hledaný automat $M=(\{p,q\},\{0,1\},\{A,B,C\},\delta,p,A,\varnothing)$ má přechodovou funkci δ definovánu takto:

$\delta(p,0,A)=\{(p,BA)\}$,

$\delta(p,1,A)=\{(p,CA)\}$,

$\delta(p,0,B)=\{(p,BB),(q,e)\}$,

$\delta(p,0,C)=\{(p,BC)\}$,

$\delta(p,1,B)=\{(p,CB)\}$,

$\delta(p,1,C)=\{(p,CC),(q,e)\}$,

$\delta(q,0,B)=\{(q,e)\}$,

$\delta(q,1,C)=\{(q,e)\}$,

$\delta(p,e,A)=\{(q,e)\}$,

$\delta(q,e,A)=\{(q,e)\}$.

Automat pracuje tak, že za každý symbol ze slova w přidá na zásobník zástupce, který pak porovná v zrcadlovém slově. Jelikož zásobník odebírá z vrcholu symboly rovněž zrcadlově, rozpozná právě slova z jazyk. Ale například jazyk $L=\{ww;\ w \in \{0,1\}^*\}$ (zdvojené slovo) už není možné rozpoznat ZA!

Řešený příklad 18:

Sestrojte zásobníkový automat, který rozpoznává jazyk $L=\{wc(w)^R; w \in \{0,1\}^*\}$ (prázdným zásobníkem).

Hledaný automat $M=(\{p,q\},\{0,1\},\{A,B,C\},\delta,p,A,\varnothing)$ má přechodovou funkci δ definovánu takto:

$\delta(p,0,A)=\{(p,BA)\}$,

$\delta(p,1,A)=\{(p,CA)\}$,

$\delta(p,0,B)=\{(p,BB)\}$,

$\delta(p,0,C)=\{(p,BC)\}$,

$\delta(p,1,B)=\{(p,CB)\}$,

$\delta(p,1,C)=\{(p,CC)\}$,

$\delta(p,c,A)=\{(q,e)\}$,

$\delta(p,c,B)=\{(q,B)\}$,

$\delta(p,c,C)=\{(q,C)\}$,

$\delta(q,0,B)=\{(q,e)\}$,

$\delta(q,1,C)=\{(q,e)\}$,

$\delta(q,e,A)=\{(q,e)\}$.

Poznámka: Tento automat je deterministický. Toto je příklad jazyka, ke kterému lze sestrojit DZA. Nicméně jsou i jazyky, ke kterým nelze DZA sestrojit (viz příklad předchozí).

Následující věta nám říká, že rozpoznávání KS a PZ jsou dvě ekvivalentní podmínky. Ke každému ZA KS lze sestrojit ekvivalentní ZA PZ a naopak. U PZ stačí doplnit instrukci, která při prázdném zásobníku automat dostane do koncového stavu. U KS je třeba doplnit více instrukcí, které v koncovém stavu (který změníme na nekoncový), vyprázdní postupně celý zásobník.

Věta 7: **Mějme libovolný jazyk L. Pak $L=L_{KS}(M_1)$ pro nějaký ZA M_1, právě když $L=L_{PZ}(M_2)$ pro nějaký ZA M_2.**

Ekvivalence rozpoznávání

Definice 24: **ZA $M = (Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$ nazveme deterministický (DZA), jestliže platí následující dvě podmínky:**

1. $\delta(q, a, X)$ je nejvýše jednoprvková množina pro lib. $q \in Q$, $a \in (\Sigma \cup \{e\})$, $X \in \Gamma$.

2. Jestliže $\delta(q,e,X) \neq \emptyset$ pro něj. $q \in Q$, $X \in \Gamma$, pak $\delta(q,a,X)=\emptyset$ pro lib. $a \in \Sigma$.

Deterministický ZA

Definice 25: **Jazyky rozpoznatelné DZA koncovým stavem nazveme deterministické (třídu těchto jazyků označíme Det). Jazyky rozpoznatelné DZA prázdným zásobníkem nazveme bezprefixové deterministické (třídu těchto jazyků označíme BDet).**

Poznámka: Dá se ukázat, že Det je vlastní podtřída třídy bezkontextových jazyků.

(Např. jazyk $\{ww^R|w \in \{a,b\}^*\}$ není deterministický.)

(Srovnejte se situací u konečných automatů).

Řešený příklad 19:
Sestrojte zásobníkový automat, který rozpoznává jazyk $L=\{w(w)^R;\ w \in \{0,1\}^*\}$ (prázdným zásobníkem).

Hledaný automat $M=(\{p,q\},\{0,1\},\{A,B,C\},\delta,p,A,\varnothing)$ má přechodovou funkci δ definovánu takto:

$\delta(p,0,A)=\{(p,BA)\}$,

$\delta(p,1,A)=\{(p,CA)\}$,

$\delta(p,0,B)=\{(p,BB),(q,e)\}$,

$\delta(p,0,C)=\{(p,BC)\}$,

$\delta(p,1,B)=\{(p,CB)\}$,

$\delta(p,1,C)=\{(p,CC),(q,e)\}$,

$\delta(q,0,B)=\{(q,e)\}$,

$\delta(q,1,C)=\{(q,e)\}$,

$\delta(p,e,A)=\{(q,e)\}$,

$\delta(q,e,A)=\{(q,e)\}$.

Automat pracuje tak, že za každý symbol ze slova w přidá na zásobník zástupce, který pak porovná v zrcadlovém slově. Jelikož zásobník odebírá z vrcholu symboly rovněž zrcadlově, rozpozná právě slova z jazyk. Ale například jazyk $L=\{ww;\ w \in \{0,1\}^*\}$ (zdvojené slovo) už není možné rozpoznat ZA!

Řešený příklad 20:

Sestrojte zásobníkový automat, který rozpoznává jazyk $L=\{wc(w)^R; w \in \{0,1\}^*\}$ (prázdným zásobníkem).

Hledaný automat $M=(\{p,q\},\{0,1\},\{A,B,C\},\delta,p,A,\varnothing)$ má přechodovou funkci δ definovánu takto:

$\delta(p,0,A)=\{(p,BA)\}$,

$\delta(p,1,A)=\{(p,CA)\}$,

$\delta(p,0,B)=\{(p,BB)\}$,

$\delta(p,0,C)=\{(p,BC)\}$,

$\delta(p,1,B)=\{(p,CB)\}$,

$\delta(p,1,C)=\{(p,CC)\}$,

$\delta(p,c,A)=\{(q,e)\}$,

$\delta(p,c,B)=\{(q,B)\}$,

$\delta(p,c,C)=\{(q,C)\}$,

$\delta(q,0,B)=\{(q,e)\}$,

$\delta(q,1,C)=\{(q,e)\}$,

$\delta(q,e,A)=\{(q,e)\}$.

Poznámka: Tento automat je deterministický. Toto je příklad jazyka, ke kterému lze sestrojit DZA. Nicméně jsou i jazyky, ke kterým nelze DZA sestrojit (viz příklad předchozí).

Následující věta nám říká, že rozpoznávání KS a PZ jsou dvě ekvivalentní podmínky. Ke každému ZA KS lze sestrojit ekvivalentní ZA PZ a naopak. U PZ stačí doplnit instrukci, která při prázdném zásobníku automat dostane do koncového stavu. U KS je třeba doplnit více instrukcí, které v koncovém stavu (který změníme na nekoncový), vyprázdní postupně celý zásobník.

Věta 8: Mějme libovolný jazyk L. Pak $L=L_{KS}(M_1)$ pro nějaký ZA M_1, právě když $L=L_{PZ}(M_2)$ pro nějaký ZA M_2.

Ekvivalence rozpoznávání

Nyní formulujeme velmi důležité tvrzení, že jazyky rozpoznávané ZA jsou právě jazyky bezkontextové. Jde o stejný typ tvrzení jako, když jazyky generované regulárními gramatikami byly právě rozpoznatelné konečnými automaty.

Lze také jednoduše sestrojit ke každé gramatice ZA, který bude rozpoznávat generovaný jazyk a to pomocí simulace odvození v gramatice na zásobníku. Zpětně lze každý automat reprezentovat pomocí BKG – složitěji pomocí postupné simulace přechodů mezi stavy ZA

Věta 9: Ke každému bezkontextovému jazyku L existuje ZA M takový, že $L=L_{PZ}(M)$. Navíc M má jediný stav.

Vztah jazyků rozpoznatelných ZA a BKJ

Věta 10: **K libovolnému ZA *M* s jedním stavem, lze zkonstruovat bezkontextovou gramatiku G tak, že $L_{PZ}(M)=L(G)$.**

Věta 11: **K libovolnému ZA *M* lze zkonstruovat ZA *M′* s jedním stavem takový, že $L_{PZ}(M)=L_{PZ}(M')$.**

Věta 12: **(důsledek) Pro libovolný jazyk L jsou následující podmínky ekvivalentní:**

Důsledky vztahů mezi

1) L je bezkontextový
2) L je rozpoznatelný ZA koncovým stavem
3) L je rozpoznatelný ZA prázdným zásobníkem
4) L je rozpoznatelný ZA s jedním stavem prázdným zásobníkem

Nejdůležitější probrané pojmy:

- zásobníkový automat
- jazyky rozpoznatelné ZA
- vztah k bezkontextovým jazykům
- důsledky těchto vztahů

Kontrolní otázka:

Existují jazyky rozpoznatelné ZA, které nejsou rozpoznatelné deterministickým ZA?

Řešení:

Takový jazyk existuje (uvedený v předchozím textu).

Korespondenční úkol:

Část 1:

Vyberte si dva bezkontextové jazyky, ke kterým nebyl v tomto textu sestrojen ZA a sestrojte jej. Pokud to jde, sestrojte DZA.

7. Chomského hierarchie

Cíl:

Po prostudování této kapitoly pochopíte:

- Hierarchizaci jazyků v jejich obecnosti
- Které jazyky jsou složitější než regulární a bezkontextové
- Jak pracují automaty pro složitější jazyky

Naučíte se:

- Klasifikovat jazyky z hlediska Chomského hierarchie

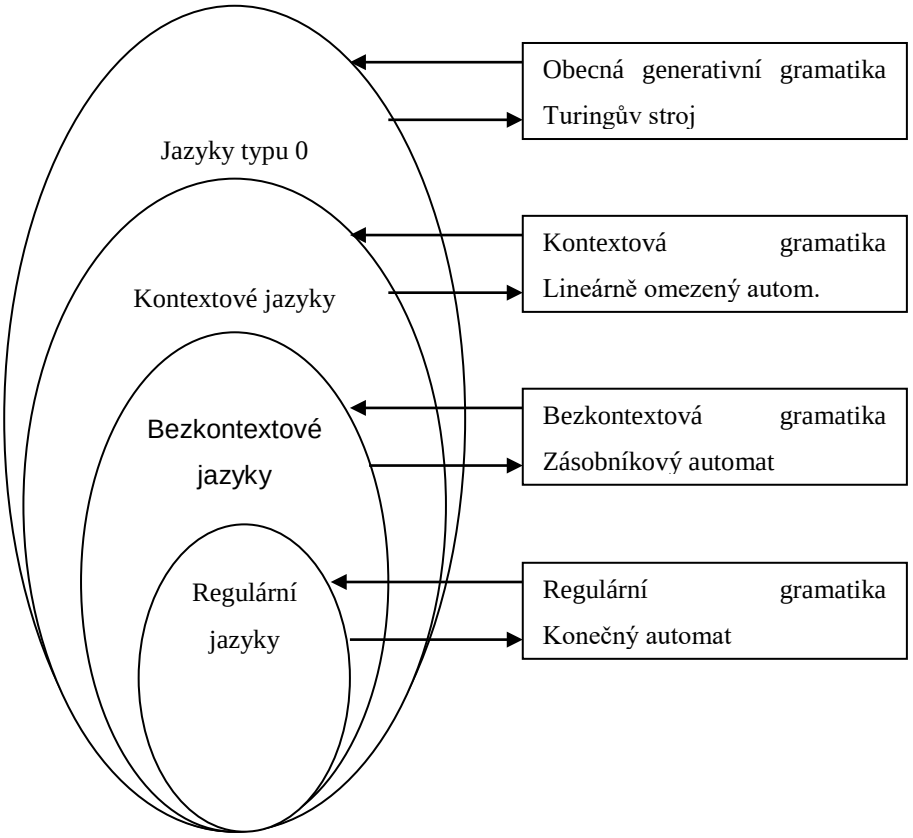
Během vašeho studia teorie formálních jazyků jste se seznámili především se dvěmi třídami jazyků – regulárními a bezkontextovými. Existují ale samozřejmě i vyšší třídy jazyků (složitější). Vzpomeňte si na obecný pojem gramatiky. Právě tyto obecné gramatiky generují nejvyšší třídu jazyků (tzv. jazyky typu 0) podle Chomského hierachie. Právě podle již zmiňovaného Noama Chomského se tato klasifikace jazyků, podle toho jaké typy gramatik je generují, nazývá.

Na obrázku můžete toto rozdělení vidět. Chomského hierarchie obsahuje 4 třídy jazyků, které lze generovat generativními gramatikami. Samozřejmě, že s použitím generativních gramatik nelze vytvořit všechny jazyky – tyto jazyky jsou pak nad touto hierarchií. Pro teoretické výsledky teorie vyčíslitelnosti je důležitá třída jazyků typy 0 a kontextové jazyky (typu 1).

Jazyky kontextové mají navíc význam pro umělou inteligenci, konkrétně analýzu přirozeného jazyka. Pro aplikované oblasti informatiky mají význam především jazyky bezkontextové (typu 2) a regulární (typu 3) a to při definování struktur programovacích a jiných jazyků používaných v praxi. Kromě gramatiky je důležitý zmíněný duální pojem automatu, který rozpoznává slova jazyka. Na obrázku jsou také ke každé třídě připojeny příslušné duální pojmy gramatiky - automatu.

V Chomského hierarchii je možné dále rozlišovat podtřídy podle toho zda jazyky lze analyzovat pomocí deterministického nebo nedeterministického automatu. Zvláště důležité to je pro třídu bezkontextových jazyků, které korespondují s používanými programovacími jazyky. Deterministické jazyky (rozpoznatelné deterministickými zásobníkovými automaty) jsou ve svých speciálních formách jako LL nebo LR jazyky efektivně analyzovatelné. Existují i alternativní hierarchie jazyků založené na odlišných přístupech ke generování jazyků, z nichž zřejmě nejznámější jsou Lindenmayerovy systémy využívané například v biologii pro simulaci chování živých organismů. Teorie jazyků je důležitou součástí informatiky a její poznatky se aplikují nejen v informatice samotné.

7.1. Obecná generativní gramatika a Chomského hierarchie



Chomského hierarchie

Definice 26: Generativní gramatika je čtveřice $G=(\Pi,\Sigma,S,P)$, kde všechny parametry mají tentýž význam jako u bezkontextových gramatik s tím, že přepisovací pravidla jsou obecně tvaru $\alpha\rightarrow\beta$, kde $\alpha,\beta \in (\Pi\cup\Sigma)^*$, přičemž α obsahuje alespoň jeden neterminál.

Generativní gramatika

Řekneme, že γ se přímo přepíše na δ a značíme $\gamma\Rightarrow\delta(\gamma,\delta \in (\Pi\cup\Sigma)^*)$, jestliže lze psát $\gamma = \gamma_1\alpha\gamma_2$, $\delta = \gamma_1\beta\gamma_2$, kde $(\alpha\rightarrow\beta) \in P$.

Relace $\Rightarrow^*$ je reflexivní a tranzitivní uzávěr relace $\Rightarrow$.

Jazyk generovaný gramatikou G je $L(G)=\{w \in \Sigma^* | S\Rightarrow^* w\}$.

Nejstarší a nejznámější hierarchie gramatik podle tvarů přepisovacích pravidel je tzv. Chomského hierarchie.

Definice 27: Generativní gramatika G=(Π,Σ,S,P) je
0) typu 0, jestliže na pravidla neklademe žádná omezení

1) typu 1, neboli kontextová gramatika, jestliže všechna pravidla jsou ve tvaru $\alpha X\beta \rightarrow \alpha\gamma\beta$, kde $|\gamma| \geq 1$, $\alpha,\beta,\gamma \in (\Pi \cup \Sigma)^*$, $X \in \Pi$. Jedinou vyjimkou je pravidlo typu $S \rightarrow e$, které se v gramatice objevit může, v tom případě se ale S nesmí objevit na pravé straně žádného pravidla.

2) typu 2, neboli bezkontextová gramatika (viz. dřívější definice)

3) typu 3, neboli regulární gramatika (viz. dřívější definice)

Věta 13: Nechť L_i označuje třídu jazyků typu i. Pak $L_3 \subset L_2 \subset L_1 \subset L_0$.

Důkaz:

$L_3 \subset L_2$, $L_1 \subset L_0$ triviálně platí.

$L_2 \subset L_1$ řeší se pomocí nevypouštějících bezkontextových gramatik.

Všechny inkluze jsou vlastní.

Např. $\{a^n b^n\} \in (L_2 - L_3)$.

Dále $\{a^n b^n c^n\} \in (L_1 - L_2)$. (viz následující příklad).

Inkluzi $L_1 \subset L_0$ nyní řešit nebudeme.

Třída kontextových jazyků, jak ji vidíte v definici obsahuje také jazyk, o kterém jsme dříve dokázali, že není bezkontextový. Kontextové gramatiky přepisují neterminály také v **kontextu** dalších slov. Nejlépe je to vidět na gramatice pro zmíněný jazyk.

Řešený příklad 21:

Příklad: Gramatika pro $L=\{a^n b^n c^n\}$

G:

$S \rightarrow aSBC$

$S \rightarrow e$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

Není těžké ověřit, že $L(G)=L=(\{a^n b^n c^n\})$.

G se dá převést na ekvivalentní kontextovou gramatiku G′:

pravidlo $CB \rightarrow BC$ se nahradí trojicí pravidel

$CB \rightarrow CB'$

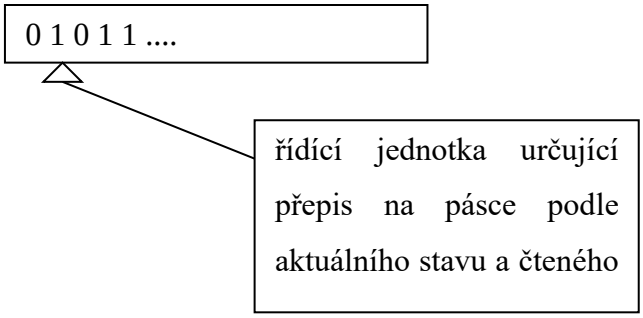
$CB' \rightarrow BB'$

$BB' \rightarrow BC$.

7.2. Turingův stroj

Turingův stroj

Na úrovni nejvyšší tedy u jazyků typu 0 je akceptorem takového jazyka Turingův stroj. Budete se jím detailně zabývat v teorii vyčíslitelnosti a složitosti. Nyní si ho ukažme pouze jako ideu. V roce 1936 Alan Turing, který je pro teoretickou informatiku klíčovou postavou, formuloval svou ideu formalizace pojmu algoritmus ve formě Turingova stroje (TS). Tato formalizace má svůj velmi jednoduchý princip mechanismu se vstupní potenciálně nekonečnou páskou s danou abecedou a čtecí hlavou, která může **zapisovat i číst** na pásce a pohybovat se po jednom políčku. Schéma tohoto stroje lze vidět na obrázku. Lineárně omezený automat se liší jen v tom, že páska pro něj není nekonečná, ale je omezena na k – násobek velikosti vstupního slova. Právě to pak způsobí, že není schopen rozpoznávat jazyky typu 0.



Tento velice jednoduchý formalismus s velkou výpočetní silou umožnil formulovat pro informatiku klíčové pojmy jako jsou rozhodnutelnost a částečná rozhodnutelnost problémů (příp. lze tyto pojmy aplikovat na funkce, množiny či jazyky). Podařilo se dokázat vlastnosti některých

problémů (nejznámějším nerozhodnutelných problémem je problém zastavení). Myšlenky důkazů těchto faktů jsou poměrně jednoduché, i když netriviální a lze je najít v literatuře [Ja97a] a [Ch84]. Dalšími důležitými výsledky jsou vztahy mezi jazyky typu 0 a rekurzivně spočetnými jazyky, které spadají také do TFJA. Pro zájemce lze doporučit distanční studijní oporu pro tento kurz [Pa02].

Nejdůležitější probrané pojmy:

Chomského hierarchie
Generativní gramatika
Turingův stroj

Úkol k textu:

Sestrojte ke každé třídě jazyků Chomského hierarchie pět jazyků, které do ní patří (s výjimkou typu 0).

8. Aplikace v programátorských úlohách

Cíl:

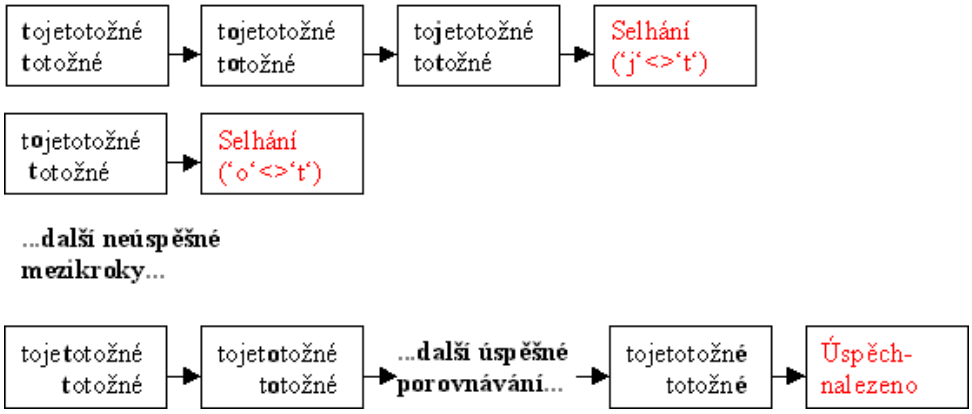
Po prostudování této kapitoly pochopíte:

- jednoduchý příklad, jak s pomocí KA vyložit netriviální algoritmus vyhledávání

8.1. Regulární jazyky a konečné automaty v praxi

Nejjednoduššími jazyky, které zkoumá teorie formálních jazyků, jsou jazyky regulární (resp. jazyky rozpoznatelné konečnými automaty). I když tyto pojmy zní odtažitě, podívejme se na jednu úlohu, kterou asi řešil každý čtenář tohoto článku a tou je vyhledávání v textu. Hned poté si ukážeme, jak možné takovou myšlenku ilustrativně vysvětlit pomocí pojmu konečného automatu. Pro tuto úlohu existují různě efektivní a složité algoritmy. Pokusme se rozebrat nejprve ten nejjednodušší, který nás zřejmě napadne (jde o algoritmus brute-force - brutální síla).

Intuitivní příklad: Vezměme si slovo „totožné“. Jak bychom realizovaly jeho vyhledávání například v textu „tojetotožné“.

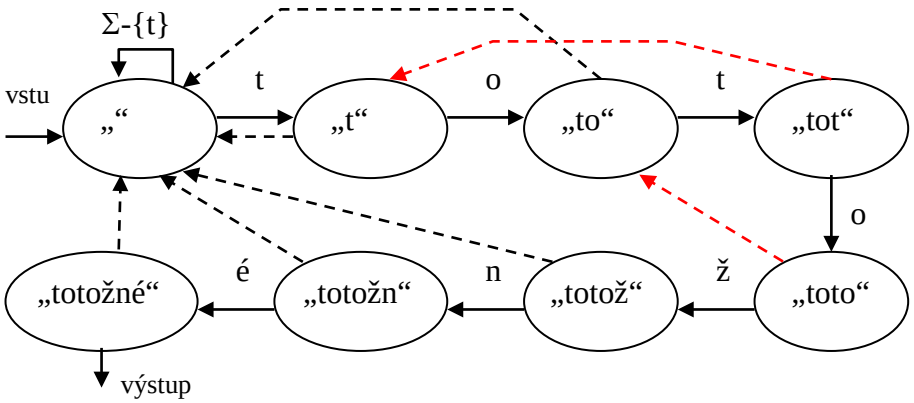


Algoritmy, které postupně načítají text a hledají výskyt slova, samozřejmě využívají knihoven funkcí. Tyto knihovny jistě obsahují již připravené algoritmy porovnání řetězců apod. Přesto půjdeme-li až na jádro způsobu nalezení slova bez použití těchto pomůcek, musí se číst postupně znaky textu a srovnávat – jde o první písmeno hledaného slova ‚t'? Pokud ano, dále srovnávej zda souhlasí následující písmena... Pokud projdeme celé slovo totožný, aniž by se v právě načítaném úseku textu něco lišilo, pak můžeme skončit a říct, že „totožný" se v textu vyskytuje a pokud ne, pak se vrátíme na další písmeno textu a celý postup opakujeme. Toto je zhruba řečeno algoritmus brutální síly. Jeho postup probíhá zkráceně takto (srovnávání textu a slova):

Vidíte, že uvedený algoritmus je skutečně „brutální". Nevyžaduje sice příliš mnoho uvažování, ale na druhou stranu je poměrně „hloupý", protože se vždy vrací v textu na následující symbol a vůbec nevyužívá informaci o tom, co již v hledaném slově úspěšně srovnal. Proto by bylo

rozumné zkusit navrhnout algoritmus, který by se již nemusel nikdy vracet v textu na symboly, které již srovnával. Pokusme se tuto myšlenku ilustrovat pomocí pojmů teorie formálních jazyků.

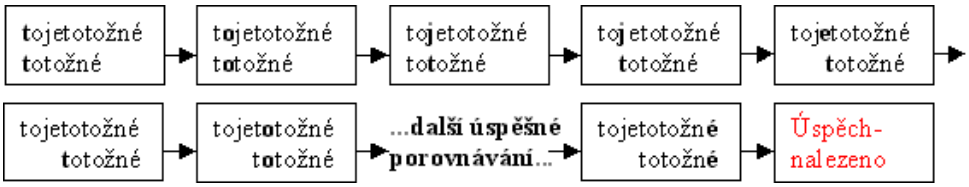
Z hlediska teorie formálních jazyků, jde o vyhledávání regulárního výrazu (mimochodem velice strukturálně jednoduchého – tyto výrazy mají obecně mnohem větší sílu než pro náš ilustrativní příklad), který můžeme realizovat pomocí konečného automatu. Pokusme se tento automat



řídící jednotka má kruhové symboly (stavy – určují zde, jaká část hledaného slova je již nalezena), které se během práce automatu mohou stávat aktivní (při

reprezentovat s pomocí stavového diagramu (jde o automat zobecněný s e-přechody – přerušovanými čarami, což nesnižuje obecnost):

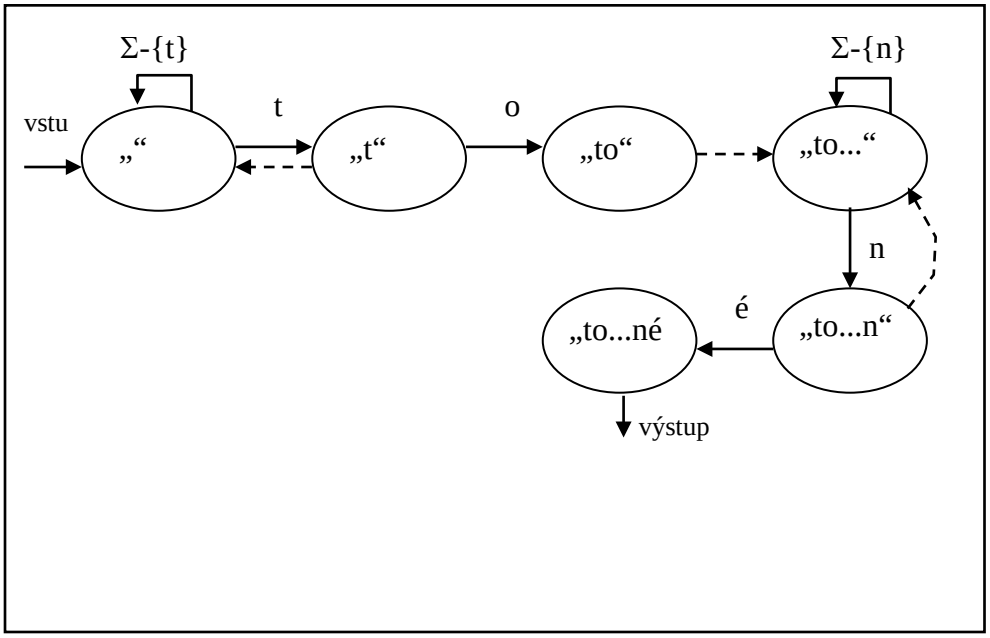
Co nám tento diagram říká? Stavy vyjadřují, jakou část hledaného slova jsme již úspěšně načetli. Na počátku po vstupu do automatu nejprve čekáme na symbol ‚t', kterým slovo začíná (to je ona smyčka Σ-{t}). Po



načtení ‚t' se dostáváme do příslušného stavu. Pokud přijde ‚o' pokračujeme v úspěšném porovnávání, ale pokud ne, pak se musíme rozhodnout, kam se vrátit, abychom sice nic z textu znovu zbytečně nečetli, ale zároveň abychom tak neopominuli již načtenou úspěšnou část slova. V tomto případě se můžeme vrátit až na počátek. Všechny situace s černou přerušovanou čarou jsou tyto „standardní návraty". Podívejme se ale, co se děje, jsme-li již ve stavu „tot". V tom případě se nemůžeme vrátit úplně na počátek, protože bychom tak ignorovali, že jsme již úspěšně načetli symbol ‚t'. Musíme se tedy do příslušného stavu nastavit, abychom tak neporušili potenciální možnost vyhledání slova v textu. Stejná „nestandardní" situace nastává ve stavu „toto". Musíme vzít v úvahu, že i přes neúspěch pro „totož" jsme se již dostali do stavu, kdy je načteno „to" a může potenciálně přijít „tot"! Naznačme schématicky, jak pracuje tento vylepšený algoritmus.

Tento automat nám vlastně poskytuje jakési „know-how“, díky němuž se zbavíme základního nedostatku (z hlediska efektivity algoritmu) a to je nutnost vracet se v textu vždy na další symbol. Při tomto postupu nikdy již čtený symbol v textu nemusíme znovu načítat. To jistě v rozsáhlých textech zrychlí hledání. Cena za to je nutnost zjistit si, kam se musím vrátit v automatu při selhání. Jelikož to však činíme jen jednou na počátku, u rozsáhlých textů se to vyplatí. Popsaný postup je vlastně teoreticky popsaným algoritmem známým jako Knuth-Morris-Prattův algoritmus. Lze jej naprogramovat a navíc poměrně jednoduše – je pouze třeba zkonstruovat postup vytváření „automatu“ – jinak řečeno tabulky, která popisuje kam se vrátit z jednotlivých stavů - jako algoritmus (není to složité – v podstatě jde o problém prohledávání slova v sobě samém a tím zjištění – kolik symbolů v jednotlivých částech slova se shoduje s jeho začátkem).

Příklad, který jsme pravě prezentovali, je samozřejmě velice jednoduchý. Konečné automaty mají větší možnosti než jen rozpoznávání pevných řetězců. Regulární výrazy, které k nim přísluší, mohou nahrazovat části slov libovolnými kombinacemi apod. Například lze sestrojit automat, který by vyhledával text se slovem, které začíná na „to“ a končí na „né“– tedy např. totožné, toporné, topné apod. Takový automat by vypadal následovně.



Dalším typickým příkladem využití konečných automatů je vyhledávání několika různých slov v textu najednou nebo naopak, slov která splňují více podmínek najednou. Právě zde se dostáváme opět již k diskutované formalizaci. Pokud pochopíte, jak automat pracuje a jak je sestrojovat, můžete po důkladnějším studiu teorie formálních jazyků a automatů používat její formální aparát – tedy například algoritmy na sestrojování sjednocení, průniku automatů apod. Pokud Vás zaujal tento ilustrativní příklad, poznali jste, jak může být pro studenta zajímavé používat tento formalismus například v algoritmizaci při výuce vyhledávacích algoritmů. Věříme, že tato teorie přímo vybízí k hledání dalších zajímavých úloh, na kterých se mohou studenti seznámit s pochopitelnými a efektivními programátorskými technikami a zároveň tak poznat svět teoretické informatiky. Lze k tomu využít již zmiňované učebnice, ať již v elektronické nebo fyzické podobě. Také Internet je velkým zdrojem inspirace pro další didaktickou práci s konečnými automaty a regulárními

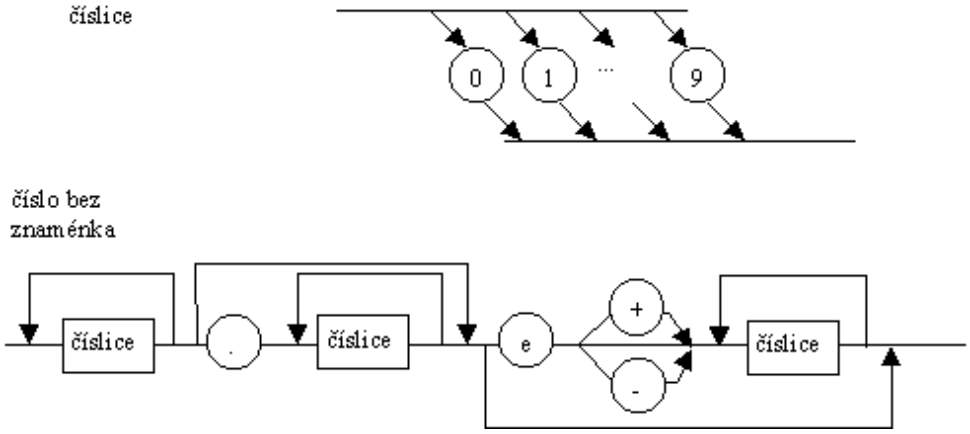
výrazy. V další kapitole pouze naznačíme, k čemu směřuje další důležitá formalizace. Bylo by nad rámec tohoto omezeného článku ji rozebírat blíže, považujte ji proto za inspiraci, jak ve výuce nastínit studentům téma bezkontextových gramatik a syntaktické analýzy.

8.2. Bezkontextové gramatiky a syntaktická analýza

Třídou jazyků bezprostředně následující v hierarchii jsou bezkontextové jazyky. Jejich význam v praxi leží pro informatika především v oblasti umělých jazyků – programovacích, dotazovacích apod. Studenti, kteří se seznamují se základy algoritmizace a programují v konkrétním programovacím jazyce, by měli hned od začátku pracovat s jasně definovanými (syntaktickými) konstrukcemi jazyka. U následujícího příkladu je pro jednoduchost zvolena konstrukce na úrovni regulárních jazyků. Nicméně syntaktické diagramy lze využít především u struktur na úrovni bezkontextových jazyků.

Intuitivní příklad:

V programovacím jazyce Pascal se využívají čísla bez znaménka. Můžeme je považovat za syntaktickou strukturu, kterou lze velmi ilustrativně zobrazit následujícím způsobem.



Tato je pro studenty velmi vhodná a názorná, umožňuje sledovat v tomto konkrétním případě, že číslice je jedna z možných deseti a číslo se pak skládá ze sledu číslic (alespoň jedné), dále může následovat desetinná tečka a za ní opět sled číslic (ovšem tento element je nepovinný – může být přeskočen). Následuje (opět nepovinně) znak ‚e' (exponent), který se skládá ze sledu číslic, které mohou a nemusí být uvozeny znaménkem.

Tyto konstrukce mohou být zapsány pro celý jazyk Pascal v přesné notaci bezkontextové gramatiky (nebo v bohatší Backus-Nauerově formě viz skripta [Ce92]). K takovému zápisu je pak možné sestrojit LL(1) gramatiku, ze které lze názorným postupem sestrojit a implementovat analyzátor (algoritmus, který zjišťuje zda program v Pascalu neobsahuje chyby v syntaxi). To lze udělat na základě různých formálních metod teorie formálních jazyků a automatů. Nejilustrativnější z nich je pak metoda rekurzivního sestupu [Dv92],[Le02] nebo [Ka02], která umožňuje přímočaře ke gramatice napsat kód v libovolném strukturovaném programovacím jazyce.

V kvalitní učebnici jazyka Pascal [Ji88], která je i dnes hojně využívaná nejen pro výuku Pascalu, ale i principů algoritmizace a programování, je možné nalézt pomocí syntaktických diagramů i Backus-Nauerovy formy celou gramatiku jazyka. Pro studenta je velice důležité, aby si při používání programovacího jazyka uvědomoval, že program není jen posloupně zapsanou sekvencí příkazů, ale že program má svá pevná syntaktická pravidla, na kterých musí být sestaven.

Následující kód v jazyce Pascal ukazuje zpracovaný fragment analyzátoru pro typ číslo, podle syntaktických diagramů.

```
procedure Number;
var point : boolean;
begin
  point := false;
  while ch in numeric do
  begin
    ident := ident+ch; GetCharI;
    if ch in ['e', 'E', DecimalSeparator] then
    begin
      point := true;
      ident := ident+ch; GetCharI;
      if ch in ['+', '-'] then
      begin
        ident := ident+ch; GetCharI;
      end;
    end;
```

```
  end;
  if ch in ignore then GetChar;
  if point then
     AssignTerms(TReal.Create(ident), 0)
  else AssignTerms(TInteger.Create(ident), 0);
  ident := '';
end;
```

Procedura GetChar (resp. GetCharI) zabezpečuje načítání jednotlivých znaků z čísla. Vidíme, že procedura odpovídá zápisu syntaktického diagramu. Situaci, kdy se nějaký element jazyka podle syntaktického diagramu opakuje přesně simulujeme pomocí příkazu cyklu „while“ a podmíněný výskyt lze algoritmicky vyjádřit pomocí podmíněného příkazu „if“. V návaznosti na strukturu celého jazyka Pascal by bylo možné vytvořit systém navzájem se rekurzivně volajících procedur, které by zjišťovaly správnost kódu (syntaktický analyzátor) a generovaly výstupní formu (zde například funkce AssignTerms vytvářejí objekty reprezentující buď celé nebo reálné číslo). Uvedený fragment je vyňat z programu, který realizuje automatizovanou dedukci a používá syntaktický analyzátor pro formule predikátové logiky [Ha99].

Nejdůležitější probrané pojmy:

algoritmy vyhledávání
syntaktické diagramy

Úkol k textu:

Vyhledejte (např. na Internetu) popis algoritmu vyhledávání Aho-Corasickové a pokuste se interpretovat jeho funkci na příkladu pomocí konečného automatu.

9. Vyčíslitelnost a složitost

Vyčíslitelnost a složitost je pro mnoho studentů informatiky nejtěžší základní partií teoretické informatiky. Zatímco teorie formálních jazyků pracuje s poměrně jednoduchými modely automatů, gramatik a jazyků u vyčíslitelnosti je mnoho těžko pochopitelných vlastností a zejména jejich důkazy a formální pojetí je náročné. Vyčíslitelnost také do značné míry souvisí s vyššími třídami jazyků než jsou regulární nebo bezkontextové. I přesto lze říci, že smysl i základní pojmy této teorie jsou opět velice blízké ”selskému rozumu” a informatice v praxi.

Algoritmus je dnes pojmem, který používají nejen informatici. S jistým zjednodušením bychom mohli říci, že algoritmy jsou jádrem informatiky. Čím by byla dnes informatika, kdyby se nesnažila najít postup řešení mnoha problémů od čistě matematických jako je řešení rovnic k ryze praktickým jako jsou algoritmy implementované v informačních systémech, které používáme každodenně (textové editory, tabulkové procesory, databázové prostředky a další).

Abychom však mohli prakticky implementovat, je nutné mít aparát pro jejich zápis, implementaci a používání automatizovanými prostředky (počítači). Samozřejmě, že algoritmem může být chápán i například postup pro přípravu jídla, ale takový vágní popis může někdy stěží zpracovat

člověk, natož stroj bez inteligence. Proto je snaha vytvářet umělé jazyky s pevně danou syntaxí a sémantikou, které by popis a implementaci algoritmu umožnily exaktně a jednoznačně. Takových prostředků existuje v informatice velké množství -nepřeberné množství programovacích jazyků vhodných pro různé účely, strojové jazyky či abstraktní a grafické prostředky, používané spíše v teoretickém návrhu nebo výuce.
Půjdeme-li ale ještě dále než k praktickému použití, začnou nás napadat otázky tohoto typu:
Jaké problémy mohu vlastně pomocí algoritmu vyřešit a jaké již ne?
Umím například pomocí jazyka Pascal zapsat řešení všech problémů jako strojovým jazykem procesoru počítače a naopak?
• Jak mohu rozpoznat, který algoritmus (program) napsaný pro řešení stejného pro
blému je lepší (např. rychlejší)?
Lze na takovéto otázky vůbec odpovědět exaktně nebo jen ’hypoteticky’.

Právě na tyto otázky hledá (a již na mnohé našla) odpověď teorie vyčíslitelnosti a složitosti. Aby měly tyto výsledky věrohodnost, je nutné přikročit opět k jisté míře formalizace a používání matematiky, ale v tomto článku se pokusíme ukázat, že základy lze vyložit i názorně.

9.1. Algoritmy, problémy a jejich řešitelnost

Pokud bychom chtěli najít specifickou činnost, kterou provádí informatik, pak by zřejmě šlo o hledání algoritmického řešení nějakého problému.

Toto řešení má tedy splňovat vlastnosti determinismu, obecnosti řešení a rezultativnosti. A právě ”problém” je základním teoretickým východiskem vyčíslitelnosti a složitosti. Problémem se zde v užším smyslu myslí otázka a instance (objekt, u kterého tuto otázku zkoumáme). Například můžeme uvažovat problém řešení kvadratické rovnice. Instancí je v tomto případě kvadratická rovnice zadaná kupříkladu ve tvaru $ax^2 + bx + c = 0$ a otázkou - jaké jsou kořeny této rovnice. Pro náš případ bychom našli velice jednoduchý algoritmus, který by kořeny vypočítal. Takovýto problém je tedy ALGORITMICKY ŘEŠITELNÝ. Právě algoritmická řešitelnost je klíčovým pojmem zkoumaným exaktně na bázi matematické formalizace. Aby bylo možné se řešitelností efektivně zabývat včetně důkazu vlastností jsou v zásadě třeba dva kroky:

1. Musíme se omezit na speciální případy problémů -tzv. PROBLÉMY TYPU ANO/NE. Jde o problémy, které mají otázku formulovanou tak, že na ni lze odpovědět buď ANO nebo NE. Např. výše zmíněný problém řešení kvadratické rovnice by nebyl tohoto typu, neboť odpověď na otázku je složitější objekt (dvě čísla). Pokud bychom chtěli dostat problém typu ANO/NE, museli bychom ho přeformulovat například na otázku, zda existuje reálný kořen. U našeho příkladu bychom pak mohli konstatovat, že jde o problém ROZHODNUTELNÝ -tedy takový, pro který máme algoritmus, jenž nám vždy dá správnou odpověď ANO nebo NE. Existují však i problémy, pro které existují pouze algoritmy dávající odpověď ANO avšak nemusejí spolehlivě dávat odpověď v případech NE. Takovéto problémy pak jsou NEROZHODNUTELNÉ, a zároveň jsou ČÁSTEČNĚ ROZHODNUTELNÉ. O těchto rozhodnutelných, nerozhodnutelných a

částečně rozhodnutelných problémech se pak dají dokazovat různé vlastnosti, nicméně je nutné podotknout, že tyto důkazy jsou již často poměrně těžké na pochopení a používají v mnoha případech principu nepřímého důkazu, nikoliv jasné a přímé konstrukce. Pro účely tohoto popularizujícího článku se zmíníme alespoň o jedné lehce pochopitelné a dokazatelné vlastnosti.

Jde o tvrzení známé jako Postova věta. Tvrdí, že pokud máme problém A a jeho doplňkový problém A (tedy takový, který má otázku formulovanou opačně -negovaně) a platí, že oba jsou částečně rozhodnutelné, pak problém A musí být rozhodnutelný. Představte si tuto situaci, která by nastala při praktickém programování. Měli byste tedy program A, který dává spolehlivě odpověď ANO pro instance, pro které má být odpověď ano, ale pro instance ostatní nemusí (může se třeba zacyklit v nekonečné smyčce při pokusu odpovědět na otázku). Dále máte program A, který odpovídá spolehlivě na opačnou otázku opět pouze pro ”ANO”. Jak můžeme tvrdit, že problém A je rozhodnutelný, tedy že pro něj existuje program, který dává spolehlivě ”ANO” i ”NE”? Velmi jednoduše, pokud si představíme program, který by po krocích paralelně prováděl instrukce vždy programu A a pak programu A (je důležité, aby se vždy provedla jen jedna instrukce). Jelikož možnosti odpovědi jsou jen dvě (ANO, NE), pak máme zaručeno, že buď nám po určitém počtu kroků skončí program A nebo A, pokud to bude první z nich, pak vrátíme odpověď ANO, protože na otázku A je tato odpověď správná. Pokud však skončí program pro opačnou otázku je odpověď na ni ANO a tedy odpověď na otázku A je NE a tu tedy vrátíme. Takovýto algoritmus vždy skončí, vrátí správnou

odpověď a je tedy rozhodnutelný. Kdyby však provádění původních programů nebylo paralelní, pak by tento postup nemusel fungovat jeden z programů by se mohl zacyklit a již bychom nedostali odpověď.

Vidíte, že otázky, které si klade teorie vyčíslitelnosti a složitosti, nejsou jen odtažitou matematickou látkou, jež studentům informatiky dělá studium náročnější. Tyto otázky totiž informatikovi pomáhají uvědomit si obecné vlastnosti algoritmů a pokud si dokáže promítnout jejich význam pro praktické programování, může to i velmi pozitivně ovlivnit jeho náhled na prakticky řešené problémy.

2. Druhým krokem pro exaktní zkoumání vlastností algoritmů je přesná definice pojmu algoritmus. Již jsme se dotkli problému, že pod pojmem algoritmus si každý čtenář může představit jiný způsob jeho zápisu. Na tomto místě se pokusíme jednoduše a na příkladu ukázat elegantní a přitom velmi jednoduchý způsob -tzv. Turingův stroj (TS). Jeho geniální a přesto prostá myšlenka má svůj původ již ve 30. letech 20. století, kdy jej formuloval Alan Turing (klíčová postava teoretické informatiky). Jde vlastně o automat (viz článek Formální jazyky a automaty uveřejněný v MFI dříve), který však má narozdíl od automatu konečného podstatnou schopnost čtení i zápisu na vstupní pásce, spolu s možností vracet se po potenciálně nekonečné pásce na libovolné místo na ní. Jde tedy opět stroj, který na vstup dostane slovo v určité abecedě, ale narozdíl od konečného automatu může nejen skončit v koncovém stavu z počátečního, ale také může slovo modifikovat a vydat tuto modifikaci jako výsledek. Důležitá je pro jeho funkci správně sestavená přechodová funkce, jenž je formálně zobrazením stavů a symbolů na nový stav, nový zapsaný symbol a příznak

posunu čtecí hlavy na pásce (hlava se může posouvat doleva a doprava, případně zůstat na místě). Takovýto Turingův stroj pak můžeme chápat jako prostředek, který realizuje zobrazení, stejně jako jej může realizovat program v Pascalu či strojový jazyk procesoru. Možná se to zdá jako neuvěřitelné, ale i takto jednoduchý formalismus má stejnou výpočetní sílu jako výše zmíněné způsoby zápisu algoritmu. Je sice jasné, že mnoho postupů je v těchto ”vyšších” prostředcích již zabudováno a můžeme s nimi pracovat, ale i přesto dokáže TS realizovat jakoukoliv funkci jako ”vyšší” prostředky. TS představuje tedy jakési programování na extrémně nízké úrovni. Výhodou však je, že tato jednoduchost je předurčena pro matematické dokazování vlastností algoritmů. Je totiž velice jednoduchou algebraickou strukturou, kde nejsložitější je přechodová funkce, představující příslušný ”program”.

Příklad 1:

Uvažujme poměrně jednoduchý problém, kterým je inkrementace čísel v binární soustavě (tedy zvýšení hodnoty čísla x o 1 na x + 1).

Např. 1011 (dekadicky 11) má být inkrementován na 1100 (dekadicky 12).

I v tak jednoduchém prostředků jako je strojový jazyk procesoru existuje prostředek, který nám přímo spočítá součet dvou čísel a tím pádem i dokáže jednou instrukcí inkrementovat hodnotu. V TS však musíme uvažovat na mnohem primitivnější úrovni. Binární číslo je pro nás slovem v abecedě složené ze symbolů 0 a 1. Umíme pouze definovat stavy TS a přechody (instrukce), které říkají jak máme změnit aktuální stav, jaký symbol máme na aktuální místo na pásce zapsat místo původního a kam se

máme přesunout. Musíme tedy uvažovat o jednotlivých akcích, které je třeba pro inkrementaci provést. Mělo by jít
o tyto činnosti:

Přesunout čtecí hlavu na konec slova, neboť tam budeme měnit číslice s nejnižším řádem.
Změnit hodnotu nejnižšího řádu z 0 na 1, ale pokud je nejnižší číslice 1, pak musíme provést změnu i ve vyšším řádu a číslici změnit na 0 (přetečení). To se ale může opakovat, takže vlastně musíme najít nejnižší řád, který má číslici 0, kterou pak můžeme změnit na 1 a tím skončit.

Podívejte se na sled těchto akcí na příkladu: 1011, 1011, 1010, 1100
Nyní nám zbývá zkonstruovat TS:
Stavy -q_1 -PŘESUN (počáteční stav), q_2 -INKREMENTACE, q_3 -KONEC (koncový stav)
Abeceda -0,1
Přechodová funkce (instrukce): 1.(q_1, 0) → (q_1, 0,P), 2.(q_1, 1) → (q_1, 1,P), 3.(q_1,ε) → (q_2, ε, L) -tyto instrukce představují přesun na konec slova -ε reprezentuje prázdné slovo, tedy symboly za a před slovem, P je přesun hlavy doprava, L přesun doleva a N když hlava zůstává na místě 4.(q_2, 0) → (q_3, 1,N), 5.(q_2, 1) → (q_2, 0,L), 6.(q_2,ε) → (q_3, 1,N) -postupná inkrementace až nedojde k přetečení

Máme-li sestrojený TS, můžeme jej aplikovat postupným výpočtem na slovo (např.

1011). Výpočet je posloupnost konfigurací TS od počáteční do koncové, kde konfigurace reprezentuje slovo na pásce, v němž je na místě čtecí hlavy symbol aktuálního stavu a znak h_i reprezentuje použití i-té instrukce: $q_1 1011\ h_2\ 1q_1 011\ h_1\ 10q_1 11\ h_2\ 101q_1 1\ h_2\ 1011q_1\ h_3\ 101q_2 1\ h_5\ 10q_2 10\ h_5\ 1q_2 000\ h_4\ 1q_3 100$

9.2. Formalizace pojmu algoritmu

Turingův stroj je nejen jednoduchou a přitom zcela exaktní formalizací pojmu algoritmus, ale na druhé straně je i lehce pochopitelný ”selským rozumem”. Je možné si jej opět jako konečný automat představit i jako fyzický stroj vykonávající instrukce (program). Lze pomocí něj i nahlédnout na zajímavé obecné vlastnosti programů. Jedním z nich je totiž existence tzv. UNIVERZÁLNÍHO TURINGOVA STROJE (UTS). Tento UTS dokáže simulovat libovolný jiný Turingův stroj (pokud se omezíme na jednoduchou abecedu, což ale nesnižuje obecnost). Pokud si opět místo TS představíme například program v Pascalu, pak nám to dává tvrzení, že existuje univerzální pascalovský program, který dokáže simulovat všechny napsané programy v Pascalu. Simulací se zde myslí, že takový univerzální program dostane na vstup kód simulovaného programu, provede ho přesně jako by byl program proveden sám a vrátí výstupy totožné očekávaným výstupům programu. Sestrojit takový univerzální pascalovský program je samozřejmě poměrně složité (i když je to jen otázka času a úsilí), ale právě jednoduchost formalizace TS umožňuje sestrojit takový UTS poměrně

rychle a snadno (dokonce to bývá úloha, kterou vysokoškolští studenti řeší jako samostatný domácí úkol!).

Pojem algoritmu je samozřejmě možno formalizovat i jinými prostředky. Jedním z nich je i více matematičtěji orientovaný formalismus, nazvaný jako PRIMITIVNĚ (OBECNĚ) REKURZIVNÍ FUNKCE (PRF/ORF). Tato formalizace bude mít pravděpodobně půvab pro ty čtenáře, kteří jsou více orientovaní na algebraické pojetí vyčíslitelnosti funkcí na množině přirozených čísel. Jejich idea se opírá o dvě základní definice (pro začátek budeme mluvit pouze o primitivně rekurzivních funkcích a později přidáme pojem obecně rekurzivní funkce):

1. Za základní považujeme tyto funkce:

• $o : N \rightarrow N, \forall x : o(x)=0$ (funkce, která vrací pro jakýkoliv argument 0 - identická nula)

• $s : N \rightarrow N, \forall x : s(x)= x +1$ (funkce, která vrací pro jakýkoliv argument jeho následující hodnotu -následník)

$I_i^{(n)(n)} : N^n \rightarrow N, \forall x_1,x_2, ..., x_n : I_i(x_1,x_2, ..., x_n)= x_i$ (funkce, která vrací i-tý argument -výběr -je nutná pro tvorbu funkcí více proměnných)

2. Další funkce můžeme skládat pomocí operátorů:

Operátory substituce S_n^m: $f = S_n^m(g, h_1, ..., h_m)$, kde platí $f(x_1,x_2, ..., x_n)= g(h_1(x_1, ..., x_n), ..., h_m(x_1, ..., x_n))$ (operátor, který umožňuje skládat funkce)

Operátory primitivní rekurze R^n: $f = R^n(g, h)$, kde platí $f(0,x_2, ..., x_n)= g(x_2, ..., x_n)$a $f(k+1,x_2, ..., x_n)= h(k, f(k, x_2, ..., x_n),x_2, ..., x_n)$ (operátor, který

definuje rekurzivní funkci na základě funkce g -zarážka rekurze pro k = 0, h -následující krok rekurze pro k + 1 definovaný pomocí k)

Z těchto základních funkcí můžeme pomocí postupné aplikace operátorů vytvářet složitější funkce.

Příklad 2:

Zkusme se podívat na příklad funkce $f(x_1,x_2)= x_1 + x_2$. Podobně jako Turingův stroj je tento formalismus poměrně primitivní, takže i takto jednoduchou funkci zde musíme sestavit. Myšlenka spočívá v tom, že použijeme rekurzi (která nám nahrazuje cyklus) a pomocí rekurze vždy vytvoříme následníka hodnoty až k nule, kdy dosadíme hodnotu druhého argumentu.

Sekvence vypadá následovně:

$f_1 = s$, $f_2 = I^3_2$, $f_3 = S^1_3(s, I^3_2)$, $f_4 = I^1_1$, $f = R(f_4,f_3)$

přičemž jednotlivé funkce vyjadřují:

f -je funkce, kterou jsme chtěli vytvořit (sčítání dvou argumentů), přičemž jsme museli aplikovat rekurzi následujícím způsobem; $f(0,x_2)= x_2$, $f(k+1,x_2)= f(k, x_2)+1$; což znamená, že x_1 -krát zvýšíme hodnotu x_2 o jedna a použijeme k tomu upravenou funkci následníka

f_4 -je funkcí výběru prvního argumentu z jednoho, kterou potřebujeme, abychom správně nadefinovali zarážku rekurze funkce f

f_3 -je upravená funkce následníka pro druhý argument ze tří (použije se v rekurzi dle formální definice operátoru); $f_3(x_1,x_2,x_3)= x_2 +1$

f_2 -je potřebná pro vytvoření následníka druhého argument v f_3; jde o výběr druhého argumentu ze tří $f_2(x_1,x_2,x_3)= x_2$

f_1 -je základní funkcí následník pro jednu proměnnou ($f_1(x_1)= x_1 + 1$)

Na tomto jednoduchém příkladě jste viděli, že notace primitivně rekurzivních funkcí umožňuje možná poněkud složitě, ale zejména exaktně symbolicky odvodit jakoukoliv funkci. I když je tato konstrukce značně odlišná od formalizace algoritmické vyčíslitelnosti pomocí TS, v konečném důsledku můžeme realizovat přesně Turingovsky vyčíslitelné funkce pouze pokud k definici primitivně rekurzivních funkcí přidáme jeden operátor tzv. operátor minimalizace, čímž dostaneme obecně rekurzivní funkce. Jelikož je přesná formální definice poněkud složitější, omezíme se na jeho vysvětlení laické. Jde o operátor, který umožňuje vytvořit funkci, která vrací nejmenší hodnotu prvního argumentu, kdy je funkční hodnota 0. Například, kdybychom potřebovali při výstavbě určité funkce znát nejmenší hodnotu funkce $f(x_1)=5 - x_1$ aplikovali bychom operátor minimalizace, který by vytvořil funkci vracející vždy 5. Samozřejmě by to asi v takto jednoduchém příkladě nedávalo smysl, ale tento operátor lze definovat pro libovolný počet proměnných a libovolnou složitost formule.

Další velice zajímavou formalizací pojmu algoritmu jsou takzvané PL-programy. Zmiňujeme se o nich ihned po rekurzivních funkcích nikoliv náhodou. Jak uvidíte, i když jde opět

o formalizaci z naprosto jiného pohledu, lze jednoduše ukázat, že jejich výpočetní síla je totožná. PL-programy (jak již název napovídá) jsou formalizací, kterou zřejmě ocení spíše programátorsky orientovaní čtenáři. PL-program je sekvence příkazů, které mohou obsahovat identifikátory

(proměnné). Jazyk je opět velmi jednoduchý, v zásadě máme pouze tyto příkazy a elementy:
Přiřazovací příkaz X := 0 (lze přiřadit nulu libovolnému identifikátoru)
Příkaz inkrementace INC X (X := X + 1)
Příkaz cyklu LOOP X [seznam příkazů] ENDLOOP (provede seznam příkazů X-krát)
Návěští L, které určuje bod programu
Příkaz skoku GOTO L (provede v programu skok do bodu L)
Specifikace vstupů a výstupu INPUT X_1,X_2, OUTPUT Y

Příklad 3:
Například funkci $f(x_1,x_2)= x_1 + x_2$ lze zapsat následujícím PL-programem:
{INPUT X1, X2} Y := 0; LOOP X1 INC Y; ENDLOOP LOOP X2 INC Y; ENDLOOP {OUTPUT Y}
Pravděpodobně se Vám zdá, že tento způsob zápisu má podobné prvky jako rekurzivní funkce. Zamyslíte-li se na jednotlivými elementy, zjistíte například že:
Přiřazovací příkaz je v podstatě identická nula
Příkaz inkrementace je vlastně funkcí následníka
Příkaz cyklu může beze zbytku nahradit rekurzi

Také lze ukázat, že bez příkazu GOTO budou PL-programy mít stejnou výpočetní sílu jako PRF, jinak jsou ekvivalentní ORF. Dále lze jednoduše simulovat libovolný PL-program pomocí Turingova stroje. Vlastně bychom jen na pásce TS vyhradili místo pro hodnoty proměnných a

postupně naprogramovali v TS všechny příkazy (viz ukázka TS - následník).

Všechny tyto vlastnosti nejsou jen matematickou teorií, která zastřešuje pojmy a metody, které využíváme v informatice. Studium těchto formalizací vám umožní pochopit, že způsobů zápisu algoritmu je mnoho a i přes jejich různorodost mohou mít stejnou vyjadřovací/výpočetní sílu. Je to podobné jako, když jeden programátor rád píše své programy v jazyce Pascal a jiný v jazyce C. I když techniky v jednotlivých jazycích se mohou lišit, oba programátoři mohou vytvořit plnohodnotné programy, které se budou chovat identicky. Také díky objevování těchto teoretických otázek můžeme hlouběji pochopit proč rekurze a cyklus jsou dva navzájem zaměnitelné nástroje algoritmického řešení problémů. Důležité je, že tyto notace jsou poměrně pochopitelné a lze si k nim vytvářet mnoho jednoduchých i složitějších příkladů, provádět převody mezi jednotlivými způsoby formalizace a tím vytvářet lepší úroveň ”informatického myšlení”. Pod tímto pojmem si představujeme schopnost navrhovat efektivní algoritmické řešení problémů, což je uvažování, které by mělo být specifickým rysem každého informatika.

9.3 Složitost řešení problému

Když už mluvíme o efektivním řešení problémů musíme se alespoň krátce zmínit o druhé stránce oboru, kterým se zabývá tento článek. Je jí takzvaná

složitost, čímž můžeme chápat především dvě odlišné vlastnosti algoritmů -buď časovou složitost nebo prostorovou složitost. Představte si, že několik studentů dostane za úkol samostatně vyřešit jistý algoritmický úkol. Budeme-li předpokládat, že všichni jej vyřeší správně a sestaví funkční algoritmy (programy) je pravděpodobné, že každý navrhne trochu jiný způsob řešení. U těchto programů pak lze položit otázku, který je vlastně nejlepší. A právě na tuto otázku lze (mimo jiné) nahlížet ze dvou hledisek:
který program dá v průměru nejrychleji výsledek?
který program spotřebuje ke svému správnému chodu nejméně paměti (prostoru)?

Jistě je odpověď na tyto otázky důležitá. Vždyť aby nám vůbec výpočetní technika v našem životě byla užitečná, musí pracovat rozumně dlouho a na strojích s kapacitou, které
máme v dané době k dispozici. Aby bylo možné zadefinovat exaktně tyto pojmy používá
se opět exaktní pojem algoritmu. Pro naše účely zvolme Turingův stroj. U něj můžeme rozlišovat tyto základní charakteristiky:
Složitost (časová) výpočtu TS na určitý vstup -jde o počet vykonaných instrukcí TS na tento vstup (je to tedy prosté přirozené číslo); např. v příkladu 1 v tomto textu byl uveden výpočet Turingova stroje na slovo 1101, počet vykonaných instrukcí než se stroj zastavil je 8.
Složitost (časová) TS (obecně) -zde jde již o funkci, kde argument je velikost slova a funkční hodnota je počet instrukcí na slovo o této velikosti

v nejhorším možném případě (je jasné, že různých slov o určité velikosti je mnoho -viz příklad 1); pro náš příklad 1 bychom opět mohli přemýšlet, jaká funkce to je. Stroj funguje tak, že vždy dojede na konec slova (což je n instrukcí), dále v nejhorším případě projde celé binární číslo (slovo) pozpátku až na začátek před první symbol a skončí v koncovém stavu(n + 2 kroků). Složitost toho TS je tedy f(n)= n + n +2=2n +2

Odhad (časové) složitosti TS je zjednodušením funkce složitosti TS. Většinou nás totiž nezajímá přesná funkce jako ve výše uvedeném případě, ale stačí nám jen ta nejdůležitější část funkce (kterou označíme O(f)), která roste nejrychleji, vzhledem k velikosti vstupního slova. Většinou se za tyto funkce berou f(n)= n, f(n)= n^2, ... atd. Tedy u našeho příkladu by odhad byl O(n).

Třída (časové) složitosti je množina těch problémů, které jsou vyčíslovány nějakým TS, se složitostí O(f). Většinou se obecně vymezují třídy lineární složitosti (T (n)), logaritmicko-lineární (T (n · log(n))), kvadratické (T (n^2)), kubické (T (n^3)), ..., polynomiální (T (n^k)) (PTIME), exponenciální (T (2^n)) (EXPTIME). Například do třídy (T (n)) patří náš problém inkrementace binárního čísla.

Druhé hledisko tedy prostorová složitost může být definována v podstatě totožně, pokud složitost výpočtu TS deklarujeme jako počet symbolů pásky, které během výpočtu projde TS než se zastaví. Třídy prostorové složitosti se pak označují podobně s příponou SPACE-PSPACE, EXPSPACE. Třídy časové složitosti a prostorové složitosti pak můžeme uspořádat a intuitivně dokázat některé vlastnosti. Například je jasné, že

když nějaký problém patří do určité třídy časové složitosti, pak určité patří do příslušné třídy prostorové složitosti (např. PTIME $\subseteq$ PSPACE). Proč tomu tak je? Vysvětlení je jednoduché. Turingův stroj nemůže za ”polynomiálně mnoho” času projít více než ”polynomiálně mnoho” symbolů, neboť se může při jedné instrukci s hlavou posunout o více než jeden symbol.

V informatice je na časové složitosti založen jeden významný problém, který je jen zdánlivě pouze teoretický. Ve skutečnosti je však více ”ze života” než si uvědomujeme. Jde

o takzvaný P-NP PROBLÉM. TS má podobně jako konečný automat také svou nedeterministickou verzi. Nedeterminismus spočívá ve více možnostech při přechodu z určité kombinace -stav, symbol, což je situace běžnému algoritmickému myšlení cizí, nicméně z hlediska složitosti umožňuje příslušnost do efektivnější třídy nedeterministické složitosti. Vezměme si příklad problému obchodního cestujícího. Spočívá v hledání nejefektivnější (nejkratší) cesty mezi několika městy, přičemž chceme navštívit všechna města právě jednou a vrátit se do výchozího. Jediné ”klasické deterministické” řešení, které je možné, spočívá v zásadě vždy ve zkoušení všech možných posloupností (existují i vylepšení, která částečně zlepšují složitost, nikoliv ale v principu). Tedy takové algoritmy patří do třídy (EXPTIME). Nicméně s pomocí nedeterministického TS můžeme tento problém řešit ve třídě NPTIME (nedeterministická polynomiální složitost). Teoretický trik (byť v praxi prozatím nic nepřinášející) je v tom, že takový nedeterministický stroj by prostě mohl z každého města zvolit jakoukoliv možnost a jelikož nás nezajímá, jak

dokáže TS tuto nejlepší možnost najít, je jeho složitost lineární (tu správnou možnost TS prostě uhodne).

Proč je to pro praxi důležité? Kdyby se podařilo dokázat, že třída PTIME = NPTIME (jinak řečeno by pro každý problém, který lze řešit nedeterministicky, existoval i deterministický polynomiální algoritmus), pak bychom našli polynomiální deterministický algoritmus pro řešení optimalizačních problémů. Tyto problémy se v každodenním životě řeší často (kupříkladu sestavení rozvrhu hodinu na škole). Jelikož je však umíme řešit pouze v exponenciální čase, jsou při velkém rozsahu nezvládnutelné. Sestavíme jinak řečeno algoritmus, který pro malý počet vstupních prvků (třeba měst u problému obchodního cestujícího) funguje, ale existuje jistá mez, kdy algoritmus bude trvat neúnosně dlouho a nepomůže nám ani zvýšení výkonu stroje, který algoritmus realizuje. Na vině je totiž ona exponenciální funkce složitosti, kdy růst této funkce je vždy od určité hodnoty příliš strmý.

Zkušenost nám říká, že tentýž úkol (problém) lze řešit různými metodami (algoritmy), které mají různou složitost. Navíc je intuitivně také zřejmé, že každý problém má určitou ‘vnitřní složitost’ (odpovídající, zhruba řečeno, složitosti toho nejoptimálnějšího algoritmu řešícího daný problém) a rovněž
problémy lze tedy určitým způsobem porovnávat podle jejich (vnitřní) složitosti. Rovněž už asi máme zkušenost s tím, že některé problémy jsou sice algoritmicky řešitelné, ale v praxi nezvládnutelné.

Z těchto intuitivních úvah lze již odvodit základní úkoly teorie složitosti:
precizovat pojmy složitost algoritmu, složitost problému a pokud možno vymezit třídu (prakticky) zvládnutelných problémů. To vše lze udělat různými způsoby, jde ovšem o to, aby zvolený způsob dával rozumné výsledky pro praxi a aby přitom zvolené pojmy byly dostatečně jednoduché a ‘průhledné’.

Začneme u pojmu složitosti algoritmu; roli algoritmů budou pro nás, ve světle předcházející části, hrát Turingovy stroje (místo Turingových strojů si můžete představovat programy ve vašem oblíbeném programovacím jazyce a vše níže uvedené si patřičně ‘překládat’); v této souvislosti je ovšem důležitá poznámka, která je uvedena na závěr textu.

Složitost Turingova stroje

Co si představovat pod pojmem složitost Turingova stroje (programu) stroje není jednoznačné. V jistém kontextu to může být např. počet instrukcí, hloubka ‘vnořených cyklů’ apod. Nám zde ovšem hlavně půjde o časovou (případně paměťovou) náročnost výpočtů daného stroje. Poznamenejme hned, že v případě nekonečných výpočtů složitost nedefinujeme – to v dalším textu nebudeme uvádět (implicitně budeme předpokládat, že relevantní výpočty jsou konečné, tj. že dojde k zastavení Turingova stroje).

Definice Časová složitost výpočtu Turingova stroje M nad slovem w se definuje jako počet elementárních kroků (instrukcí), které M nad w vykoná, než se zastaví.

Časová složitost stroje M by se teď dala chápat jako funkce typu $\Sigma^* \rightarrow N$ (kde Σ je abeceda stroje a N je množina přirozených čísel). Tento pojem je ale příliš detailní a navíc se explicitně odkazuje k abecedě daného stroje. Lépe se osvědčuje následující definice:

Definice: Časovou složitostí Turingova stroje M rozumíme funkci $TM : N \rightarrow N$, kde TM (n) znamená časovou složitost výpočtu M nad vstupem délky n v nejhorším případě (tj. TM (n) = max{k | k je časová složitost výpočtu M nad w, kde |w| = n}).

Odhady složitosti

Při analýze časové složitosti konkrétního Turingova stroje (či programu, chcete-li) M nám v praxi většinou nejde o přesný popis funkce TM , ale jen o její odhad. Navíc se většinou zanedbávají konstantní faktory, což vede k následujícímu značení:

Neostrý horní odhad

- Značením $f \in O(g)$, nebo $f(n) \in O(g(n))$, rozumíme, že ex. k a n_0 tž.

$\forall n \geq n_0 : \ f(n) \leq k \bullet g(n)$

Ostrý horní odhad

- $f \in o(g)$, nebo $f(n) \in o(g(n))$, znamená, že pro každé (reálné) $k > 0$ ex. n_0 tž. $\forall n \geq n0 : \ f(n) < k \bullet g(n)$.

Asymptotická rovnost

- $f \in \Theta(g)$, nebo $f(n) \in \Theta(g(n))$, znamená, že $f \in O(g)$ a zároveň $g \in O(f)$.

Dolní odhad

- $f \in \Omega_\infty(g)$, nebo $f(n) \in \Omega\infty(g(n))$, znamená, že ex. $k > 0$ a nekonečně dolní odhad mnoho n tž. $f(n) \geq k \bullet g(n)$.

Nejběžnější funkce vyskytující se v odhadech jsou funkce log n, n, n•log n, n^2, n^3, 2^n apod. (log se většinou chápe se základem 2; uvědomte si, že díky zanedbávání konst. faktoru na tom nezáleží).

Teď už je např. jasné, co to znamená, že časová složitost nějakého Turingova stroje je v $O(n^2)$, či v $O(n \bullet \log n)$ apod. Všimněme si, že O hraje roli (neostrého) horního odhadu, o roli ostrého horního odhadu, Ω_∞ představuje určitý dolní odhad a Θ je vlastně horní i dolní odhad zároveň (složitost je v tom případě 'přesně' určena – samozřejmě až na zanedbávané faktory).

Úkoly k zamyšlení:

Na rozdíl od složitosti algoritmu (Turingova stroje), je pojem složitosti problému hůře definovatelný (zamyslete se nad tím!).

Složitost problému

Definice: Třídou (časové) složitosti T (f) pro funkci f : N $\longrightarrow$ N rozumíme třídu těch problémů, které jsou rozhodovány (vyčíslovány) Turingovými stroji s časovou složitostí v O(f).

Všimněme si, že určitě platí např.

$T(n) \subseteq T(n \bullet \log n) \subseteq T(n^2) \subseteq T(n^3) \subseteq T(2^n)$

(Z hlubších výsledků teorie složitosti, které zde nebudeme zmiňovat, plyne, že každá z uvedených inkluzí je vlastní.)

Část teorie, která se někdy nazývá konkrétní složitost, studuje složitost konkrétních problémů (a algoritmů), resp. příslušné horní a dolní odhady. My se zde dotkneme spíše tzv. strukturální složitosti, jež má za úkol zkoumat strukturu tříd složitosti problémů. Podotkněme ovšem, že obě zmíněné partie se samozřejmě prolínají a ovlivňují. Jedním z nejdůležitějších cílů teorie (strukturální) složitosti je co možná nejlépe charakterizovat třídu zvládnutelných problémů (tj. třídu problémů, pro které existují ‘dostatečně rychlé’ algoritmy).

Jako nejrozumnější aproximace třídy zvládnutelných problémů se (zatím) ukázala třída označovaná P T IM E, nebo jen P (ze slova ‘Polynomial’), definovaná následovně

$$PTIME = \bigcup_{k=0}^{\infty} \mathcal{T}(n^k)$$

To znamená, že pojem ‘rychlý algoritmus’ je ztotožňován s pojem ‘polynomiální algoritmus’ (tj. algoritmus s polynomiální časovou složitostí). To není samozřejmě ideální (např. algoritmus s časovou složitostí zhruba $n^{1000000}$ těžko lze považovat za rychlý), zatím je však tato

charakterizace shledávána jako vyhovující (poznamenejme, že se ukazuje, že existuje-li pro problém ‘z praxe’ polynomiální algoritmus, pak exponent v polynomu je velmi malý – řekněme menší než 5).

Nepochybuji o tom, že jistě znáte spoustu zvládnutelných problémů (prvků P T IM E), později ukážeme (rozhodnutelné) problémy, o kterých je dokázáno, že zvládnutelné nejsou (jsou mimo P T IM E).

Podobnou roli jako algoritmická převeditelnost pro (ne)rozhodnutelnost problémů přináší tzv. polynomiální převeditelnost pro (ne)zvládnutelnost problémů (definice je v podstatě stejná, jen u algoritmu převodu je vyžadována
polynomiální časová složitost – tzn. složitost v $O(n^k)$ pro nějaké $k \in N$):

Složitost nedeterministických Turingových strojů

Definice: Daný problém (typu ANO/NE) je rozhodován nedeterministickým Turingovým strojem M , jestliže všechny výpočty M jsou konečné a vydávají ANO nebo NE, přičemž pro libovolnou instanci I daného problému existuje (alespoň jeden) výpočet M nad I vydávající ANO právě když (správná) odpověď na I je ANO.

Složitost takového nedeterministického Turingova stroje M pro slovo w je pak definována jako délka nejkratšího možného výpočtu nad w

vydávajícího ANO – pokud takový existuje; v opačném případě lze vzít délku nejkratšího výpočtu (vydávajícího NE).

Nedeterministická polynomiální časová složitost

Další definice lze již standardně doplnit, takže by mělo být jasné, co se myslí třídou (problémů typu ANO/NE) označovanou NPTIME, nebo někdy jen NP (N ze slova ‘nondeterministic’).

Dá se celkem snadno ukázat, že

PTIME ⊆ NPTIME ⊆ PSPACE.

Takto jsme se dostali k velmi známé dosud otevřené otázce, zda P T IM E = N P T IM E (dané otázce se často říká P − N P problém).

Poznamenejme, že podobně lze dodefinovat třídu NPSPACE apod. Savitch ukázal elegantní důkaz rovnosti PSPACE = NPSPACE.

O spoustě praktických problémů (jedním z nich je tzv. ‘problém obchodního cestujícího’, dále se ještě zmíníme o problému splnitelnosti booleovských formulí) se snadno ukáže, že jsou v NP , ale nikdo pro ně nezná (deterministický) polynomiální algoritmus. Tyto problémy jsou jistým způsobem nejtěžší ve třídě NP (jsou tzv. NP-úplné):

Definice: Mějme třídu složitosti C. O problému P řekneme, že je C-těžký, jestliže pro libovolný P´ ∈ C platí P´Á P . Je-li navíc P ∈ C, říkáme, že P je C-úplný.

Speciálně pro třídu NP dostaneme, že problém P je NP-úplný, pokud je ve třídě NP a pokud platí, že libovolný problém z této třídy je na něj převeditelný.

9.4. NP-úplné problémy

Podle definice lze (vcelku přímočaře, i když poměrně pracně) dokázat tzv. Cookovu větu:

Věta: (Cook) Problém splnitelnosti booleovských formulí je NP-úplný.

Problém SAT (Splnitelnost booleovských formulí)

Instance: Booleovská formule [obvykle předpokládáme v konjunktivní normální formě].

Otázka: Existuje ohodnocení proměnných, při němž je formule pravdivá?

Další NP-úplné problémy

Mezi další NP-úplné problémy se řadí například tyto následující

- Problém 3-SAT

Instance: Booleovská formule v konjunktivní normální formě, jejíž všechny klauzule mají právě tři literály.

Otázka: Existuje ohodnocení proměnných, při němž je formule pravdivá?

- **Problém CLI (Klika v grafu)**

Instance: Neorientovaný graf G, číslo k.

Otázka: Existuje k-klika v G (úplný podgraf velikosti k) ?

- **Problém IS (Nezávislá množina v grafu)**

Instance: Neorientovaný graf G, číslo k.

Otázka: Existuje nezávislá množina vrcholů v G velikosti k? (v ne- závislé množině nesmí existovat hrana mezi žádnými dvěmi vr- choly z této množiny)

- **Problém 3-COL (Barvení grafu)**

Instance: Neorientovaný graf G.

Otázka: Je možno obarvit vrcholy grafu G třemi barvami tak, aby žádné dva vrcholy, které jsou spojené hranou nebyly obarveny stejnou barvou?

- **Problém SALESMAN (Problém obchodního cestujícího)**

Instance: Neorientovaný ohodnocený graf G představující množinu měst propojených cestami zadané délky a povolená délka cesty d.

Otázka: Existuje způsob, jak navštívit právě jednou všechna města, jestliže má cesta skončit ve stejném městě jako začala a celková vzdálenost nemá převýšit d?

Jestliže prokážeme NP-úplnost nějakého problému, znamená to pro nás, že je prakticky nezvládnutelný (tzn. napíšeme-li program, který daný problém přesně rozhoduje, budeme ho moci skutečně použít jen na velmi malá vstupní data) – teoreticky ovšem pořád ještě možnost rychlého algoritmu existuje. Podobně to platí i pro PSPACE-úplné problémy (jako je třeba problém, zda dané dva regulární výrazy jsou ekvivalentní [tj. představují

tentýž jazyk]). Je-li ovšem nějaký problém např. EXPSPACE-úplný, je už určitě (dokazatelně) nezvládnutelný; příkladem je problém ekvivalence regulárních výrazů s mocněním (je možno psát $(a)^2$ místo a • a).

Existují samozřejmě i dokazatelně těžší (superexponenciální) problémy.

Mezi ně patří například problém rozhodování Presburgerovy aritmetiky (rozhodování pravdivosti formulí teorie sčítání).

Úkoly k zamyšlení:

Zamysleme se nyní nad robustností uvedených pojmů a výsledků – nejsou náhodou závislé na námi zvoleném modelu algoritmů, tj. na Turingových strojích? Úvahy tohoto typu jsou určitě oprávněné: ačkoli se nám např. nepodaří navrhnout stroj pro rozpoznávání slov typu ww^R se složitostí menší než řádově n^2, v případě modelu Turingova stroje s dvěma hlavami je složitost daného problému zřejmě lineární. Oba modely se ovšem vzájemně simulují s polynomiální ztrátou, takže definice tříd P, NP atd. jsou pro ně stejné. Zakončeme vše konstatováním, že všechny ‘rozumné’ (realistické) modely algoritmů jsou polynomiálně ekvivalentní Turingovým strojům (vzájemně se simulují s polynomiální ztrátou). Pro ně jsou tedy výše uvedené úvahy (týkající se např. NP-úplnosti apod.) totožné.

Shrnutí:

- Hlavní otázkou teorie složitosti je, jak náročné je vyčíslování řešitelných
problémů. Není naprosto jednoznačné určit kritéria, podle kterých se obtížnost konkrétních problémů má určovat. Pro praxi se však jeví jako nejpřínosnější třídění problémů podle jejich časových a prostorových nároků.

- V souvislosti se složitostí problému uvažujeme o jeho tzv. vnitřní složitosti, která by se dala charakterizovat jako složitost toho nejoptimálnějšího algoritmu, který daný problém řeší. Složitost konkrétního algoritmu málokdy potřebujeme znát přesně definovanou jako funkci v závislosti na vstupních hodnotách, ale mnohdy se bohatě spokojíme s tzv. odhady O, o, Θ, resp. Ω funkce určující složitost v závislosti na velikosti vstupu.

- Hlavním úkolem, který si teorie vyčíslitelnosti klade je najít třídu prakticky zvládnutelných problémů. V praxi se ukazuje, že onou třídou je právě třída P T IM E, tedy skupina obsahující problémy s polynomiální časovou složitostí.

- Podobně jako jsme měli zavedený pojem algoritmické převeditelnosti problému, zavádíme pojem polynomiální převeditelnosti s tím, že od algoritmu transformujícího instanci problému se požaduje, aby byl polynomiální. Když jsme schopni nějaký zaručeně těžký problém převést na jiný problém, tak máme zaručeno, že i ten bude přinejmenším tak náročný. Naopak, když nějaký problém dokážeme převést na něco jednoduchého, tak tím vlastně máme nalezeno jednoduché řešení pro daný problém.

- Často diskutovanou skupinou problémů je třída NPTIME rozhodovaná v polynomiálním čase pomocí nedeterministických Turingových strojů. Speciální podmnožinu problémů, které jsou jistým způsobem nejtěžší, v této třídě tvoří tzv. NP-úplné problémy, pro které známe rychlé (polynomiální) nedeterministické řešení, ale zatím se nikomu nepodařilo najít polynomiální algoritmus, který by alespoň jeden z nich rozhodoval deterministicky. Na druhou stranu se ještě nepodařilo dokázat, že takový algoritmus neexistuje. Je to stále otevřený a známý P − N P problém.

Kontrolní otázky:

1. Vysvětlete pojem vnitřní složitosti problému.

2. Objasněte základní úkoly teorie složitosti.

3. Vysvětlete vztah mezi konkrétní a strukturální složitostí.

4. Co to znamená, že je nějaký problém N P -úplný?

10. Složitost v praxi

Cíl:

Seznámíte se:

- s metodami pro řazení
- jejich časovou složitostí

Řadící algoritmy jsou algoritmy, které seřadí prvky v kontejneru, nebo jeho části. Řadícím algoritmům se často říká třídicí algoritmy. Není to ale přesné označení, protože pod pojmem třídicí algoritmus by jsme si měli představit spíše něco, co rozděluje objekty podle toho, jaké jsou třídy.

Na třídícím algoritmu může záležet **rychlost** celého výsledného programu. A právě proto musíme zvolit takový, který je dostatečně rychlý a stabilní.

Tříděním rozumíme uspořádání prvků vzestupně či sestupně.

Pro vzestupně seřazenou posloupnost musí platit:

i-tý prvek < i+1 prvek

Pro sestupně seřazenou posloupnost musí platit :

i-tý prvek > i+1 prvek

Klíč - položka záznamu, podle které se záznamy řadí

Třídění (sorting, sort) - rozdělování údajů na skupiny se stejnými vlastnostmi.

Vlastnosti řazení

Uspořádání podle klíčů (collating) - seřazení údajů podle prvků (klíčů) lineárně uspořádané množiny.

Řazení (sequencing) - uspořádání údajů podle relace lineárního uspořádání.

Slučování (coalescing) - vytváření souboru sjednocením několika souborů.

Setřídění (zakládání, merging) - vytváření souboru sjednocením několika souborů, jejichž údaje jsou seřazeny podle téže relace uspořádání se zachováním této relace.

Ve smyslu těchto definic budeme algoritmy tradičně nazývané "třídicí algoritmy" nazývat "řadicí algoritmy" a řazení budeme chápat jako zvláštní případ obecnějšího pojmu třídění.

Přirozenost

Přirozenost řazení je vlastnost algoritmu, která vyjadřuje, že doba potřebná k řazení již seřazené množiny údajů je menší než doba pro seřazení náhodně uspořádané množiny a ta je menší než doba pro seřazení opačně seřazené množiny údajů.

Stabilita

Stabilita řazení je vlastnost řazení, která vyjadřuje, že algoritmus zachovává vzájemné pořadí údajů se shodnými klíči.

Stabilita je nutná tam, kde se vyžaduje, aby se při řazení údajů podle klíče s vyšší prioritou neporušilo pořadí údajů se shodnými klíči vyšší priority, získané předcházejícím řazením množiny podle klíčů s nižší prioritou.

Rychlost

Rychlost je funkcí doby třídění v závislosti na počtu řazených prvků.

Požadavky na hardware

Tedy přesněji řečeno „požadavky na paměť.“

U kvalitního algoritmu nepřekročíme nároky na paměť více, než zabírá seznam nesetříděných údajů.

Podle typu paměti v níž je řazená struktura uložena

- **vnitřní** (interní) metody řazení (metody řazení polí) předpokládají uložení seřazované struktury v operační paměti a přímý (nesekvenční) přístup k položkám struktury.

- **vnější** (externí) řazení (metody řazení souborů) předpokládají sekvenční přístup k položkám seřazované struktury.

Složitost v praxi

Podle řádu složitosti

Od $O(n \log_2 n)$ do $O(n^2)$

Podle metody

- **přímé metody** - přirozené postupy řazení, složitost n^2
- **nepřímé metody** - založené na speciálních metodách.

Podle základního principu řazení

- metody založené na principu **výběru** (selection) - postupný přesun největšího (nejmenšího) prvku do seřazené výstupní struktury
- metody založené na principu **vkládání** (insertion) - postupné zařazování prvku na správné místo ve výstupní struktuře
- metody založené na principu **rozdělování** (partition) - rozdělování řazené množiny na dvě části tak, že prvky jedné jsou menší než prvky druhé
- metody založené na principu **setřídění** (merging) - sdružování seřazených podmnožin do větších celků
- metody **založené na jiných principech** - nesourodá skupina ostatních metod nebo kombinací metod

Nejznámější třídicí algoritmy

- **Bubble** Sort
- **Select** Sort

- **Insert** Sort
- **Binary** Insert Sort
- **Shell** Sort
- **Shaker** Sort
- **Heap** Sort
- **Radix** Sort
- **Merge** Sort
- **Quick** Sort

10.1. Bubble-Sort

Bubble-Sort

Je nejprimitivnějším, ale také ale také nejpomalejším třídícím algoritmem. Bublinkové řazení (angl. Bubblesort) se chová tak, že porovnává každý prvek se svým následníkem, a je-li tento větší, pak je zamění. Toto provede pro celou posloupnost n prvků. Celý postup (n porovnání a záměn) musíme aplikovat n-krát, abychom si byli jisti, že prvky jsou seřazeny. Největší prvky tak „probublají“ na konec seznamu, odtud název algoritmu.

Ale pokud necháme algoritmem projít již **seřazenou posloupnost, je tato metoda jedna z nejrychlejších !** Tohoto můžeme využít, pokud chceme např. zjistit, zda je pole již seřazené.

Analýza algoritmu Bubble-Sort

Algoritmus bublinové řazení nemá nejhorší a nejlepší případ. Počet porovnání a přesunů nezávisí na počátečních hodnotách prvků.

```
procedure BubbleSort(var pole:TPole; N:word);
var i,j, pom: integer;
begin
  for j:=1 to N-1 do {probublavani provadime pro n-1 prvku}
    for i:=1 to N-1 do {postupne pro vsechny prvky pred poslednim}
      if pole[i]>pole[i+1] then {pokud je prvek mensi nez nasledujici}
      begin {prehod prvky => probublavani vetsiho prvku polem}
        pom:=pole[i+1]; {prvky snadno prohodime pomoci pomocne promenne pom}
        pole[i+1]:=pole[i];
        pole[i]:=pom
      end
end;
```

Princip algoritmu Bubble-Sort

Cyklicky se prochází pole a porovnávají se sousední prvky. Je-li prvek vpravo menší než vlevo, pak se vzájemně prohodí. Řazení končí průchodem, ve kterém se nic neprohodilo.

První průchod:

```
for j:=1 to N-1 do
  for i:=1 to N-1 do
     if pole[1]>pole[2] then
{dále nic nedělám podmínka není splněna}
       begin
          pom:=pole[i+1];
          pole[i+1]:=pole[i];
         pole[i]:=pom;
    end;
end;
```

Řešený příklad 22:

	1	**2**	3	4	5	6	7	8	9	10
0.	2	4	8	6	5	7	9	1	3	10

```
for j:=1 to N-1 do
  for i:=1 to N-1 do
     if pole[3]>pole[4] then {podmínka splněna}
       begin                {dochází k přehození prvků}
```

```
            pom:=pole[4];     {do pom se uloží hodnota 6}
            pole[4]:=pole[3];
{do pole[4] se uloží hodnota 8 z pole[3]}
            pole[3]:=pom;
{do pole[3] se uloží pom což je 6}
     end;
end;
```

	1	2	**3**	**4**	5	6	7	8	9	10
1.	2	4	6	8	5	7	9	1	3	10

```
for j:=1 to N-1 do
  for i:=1 to N-1 do
     if pole[4]>pole[5] then {podmínka splněna}
       begin
{dochází k přehození prvků}
            pom:=pole[5];
{do pom se uloží hodnota 5}
            pole[5]:=pole[4];
{do pole[5] se uloží hodnota 8 z pole[4]}
            pole[4]:=pom;
{do pole[4] se uloží pom což je 5}
     end;
end;
```

	1	2	3	**4**	**5**	6	7	8	9	10
2.	2	4	6	5	8	7	9	1	3	10

```
for j:=1 to N-1 do
  for i:=1 to N-1 do
     if pole[5]>pole[6] then {podmínka splněna}
       begin
{dochází k přehození prvků}
          pom:=pole[6];
{do pom se uloží hodnota 7}
          pole[6]:=pole[5];
{do pole[6] se uloží hodnota 8 z pole[5]}
          pole[5]:=pom;
{do pole[5] se uloží pom což je 7}
    end;
end;
```

	1	2	3	4	**5**	**6**	7	8	9	10
3.	2	4	6	5	7	8	9	1	3	10

```
for j:=1 to N-1 do
  for i:=1 to N-1 do
     if pole[6]>pole[7] then
{dále nic nedělám podmínka není splněna}
       begin
```

```
            pom:=pole[i+1];
            pole[i+1]:=pole[i];
           pole[i]:=pom;
    end;
end;
```

4.

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

```
for j:=1 to N-1 do
  for i:=1 to N-1 do
     if pole[7]>pole[8] then {podmínka splněna}
       begin
{dochází k přehození prvků}
           pom:=pole[8];
{do pom se uloží hodnota 1}
           pole[8]:=pole[7];
{do pole[8] se uloží hodnota 9 z pole[7]}
           pole[7]:=pom;
{do pole[7] se uloží pom což je 1}
    end;
end;
```

5.

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

```
for j:=1 to N-1 do
  for i:=1 to N-1 do
     if pole[8]>pole[9] then {podmínka splněna}
       begin
{dochází k přehození prvků}
          pom:=pole[9];
{do pom se uloží hodnota 3}
          pole[9]:=pole[8];
{do pole[9] se uloží hodnota 9 z pole[8]}
          pole[8]:=pom;
{do pole[8] se uloží pom což je 3}
    end;
end;
```

	1	2	3	4	5	6	7	**8**	**9**	10
5.	2	4	6	5	7	8	1	3	9	10

Druhý průchod:

- začíná se procházet od začátku pole stejným způsobem jako v prvním
- průchodu

Atd.

Varianty Bubble-Sort

- Ripple-Sort - pamatuje si, kde došlo v minulém průchodu k první výměně a začíná až z tohoto místa

- Shaker-Sort - prochází oběma směry a "vysouvá" nejmenší na začátek a největší na konec

- Shuttle-Sort - dojde-li k výměně, algoritmus se vrací s prvkem zpět, pokud dochází k výměnám. Pak se vrátí do místa, odkud se vracel a pokračuje

Upravený algoritmus Bubble-Sort

a) Zjišťujeme, zda při průchodu pole došlo alespoň k jedné záměně prvků. Algoritmus končí, když projdeme n-prvkové pole n-1 krát nebo když při průchodu nedošlo k záměně.

Takto upravený algoritmus dokáže rozpoznat seřazené pole.

```
procedure BubbleSort(var pole:TPole; N:word);
var i,j, pom: integer;
    konec: boolean;
begin
  for j:=1 to N-1 do
  begin {probublavani provadime pro n-1 prvku}
    konec:=true;
{pred zacatkem vnitrniho cyklu nastavime konec na true}
    for i:=1 to N-1 do
```

```
{postupne pro vsechny prvky pred poslednim}
       if pole[i]>pole[i+1] then
{pokud je prvek mensi nez nasledujici}
       begin {prehod prvky => probublavani vetsiho prvku polem}
         pom:=pole[i+1];
         pole[i+1]:=pole[i];
         pole[i]:=pom;
         konec:=false {s prvkem se provedla vymena}
       end;
     if konec then Break
{pokud nebyl ani jeden prvek v cyklu vymenen,

tj. vsechny prvky byly uz na svem miste,

ukoncime trideni (bylo by zbytecne setridene prvky dale prochazet}
   end
end;
```

b) Při průchodu polem si pamatujeme místo, kdy došlo k poslední záměně prvků. Dále doprava je již pole seřazeno. Při dalším průchodu porovnáváme pouze prvky před poslední záměnou.

Takto upravený algoritmus dokáže rozpoznat seřazené pole.

```
Procedure BubbleSort(var a:pole;n:integer);
var
   i,j,r,pom:integer;
begin
   j:=0;
   r:=n-1;
   repeat
```

```
    j:=1;
    for i:=1 to r do
      if a[i]>a[i+1] then begin
                           pom:=a[i];
                           a[i]:=a[i+1];
                           a[i+1]:=pom;
                           j:=i;
                          end;
     r:=j-1;
until r=0;
end;
```

10.2. Select-Sort

Tato metoda pracuje tak, že vyhledá nejmenší prvek v nesetříděné části a zařadí ho na konec již setříděné části. Tzn. že musíme projít celé pole, najít nejmenší prvek a zařadit ho na první místo. Poté znova musí projít pole od druhého prvku pole (protože první prvek má již svou konečnou pozici) a vyhledá opět nejmenší prvek. Ten zařadí na druhou pozici. Tato činnost se opakuje tak dlouho, dokud neprojde celou posloupnost a nesetřídí ji.

Select-Sort

Tato metoda také pracuje tak, že vyhledává největší prvek v nesetříděné části a zařadí ho na konec nesetříděné části. Tzn. že musíme projít celé pole, najít největší prvek a zařadit ho na poslední místo. Poté znova musí projít pole od předposledního prvku pole (protože poslední prvek má již svou konečnou pozici) a vyhledá opět největší prvek. Ten zařadí na

předposlední pozici. Tato činnost se opakuje tak dlouho, dokud neprojde celou posloupnost a nesetřídí ji.

Procedura **hledání maxima**

```
procedure SelectSort(var a:pole;n:integer);
var   i,j,k,max:integer;
begin
   for i:=n downto 2 do
   begin
     max:=a[i];
     k:=i;
     for j:=1 to i do
     if a[j] > max then begin
                          max:=a[j];
                          k:=j;
                        end;
     a[k]:=a[i];
     a[i]:=max;
   end;
end;
```

Procedura **hledání minima**

```
procedure SelectSort(var  a:pole;n:integer);
var i,j,k,min : integer; { poslední minimum }
begin
  for i:=1 to n-1 do
     begin
       k:=i;
       min :=a[k];
       for j:=i+1 to n do     { hledání minima }
          if a[j] < min then begin
                               min:=a[j];
                               k:=j;
```

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

```
                                end;
      a[i]:=a[k];          { výměna prvků }
    end; end;
```

Řešený příklad 23:

Princip algoritmu Select Sort

- hledání maxima

1.

```
for i:=n downto 2 do {zleva do prava i je 10}
   begin
     max:=a[i]; {do max ulozim poslední hodnotu 10}
     k:=i;      {do k ulozim i coz je 10}
     for j:=1 to 10 do  {od zacatku pole do konce}
     if a[j] > max then begin  {jestlize hodnota je
vetsi nez max –
                                    podmínka není splněna}
                            max:=a[j];
                            k:=j;
                          end;
     a[k]:=a[i]; {a[10]:=a[10]}
     a[i]:=max;  {max je stale 10}
   end;
end;
```

2.

```
for i:=n downto 2 do  {zmensuji i na 9}
   begin
     max:=a[i];
{do max ulozim predposlední hodnotu 8}
     k:=i;           {do k ulozim i coz je 9}
     for j:=1 to 9 do  {prohledavam od zacatku}
     if a[j] > max then begin
{jestlize hodnota je vetsi nez max
                                        - podmínka splněna}
                              max:=a[j];
{do max ulozim a[8]=9}
                              k:=j;       {k=8}
                            end;
     a[k]:=a[i]; {přehodím prvky a[8]:=a[9]}
     a[i]:=max;  {max je 9}
   end;
end;
```

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

3.

```
for i:=n downto 2 do  {zmensuji i na 8}
   begin
     max:=a[i];
{do max ulozim predposlední hodnotu 8}
     k:=i;           {do k ulozim i coz je 8}
     for j:=1 to 8 do  {prohledavam od zacatku}
     if a[j] > max then begin    {jestlize hodnota je
vetsi nez max
                                        - neni splněna}
                              max:=a[j];
```

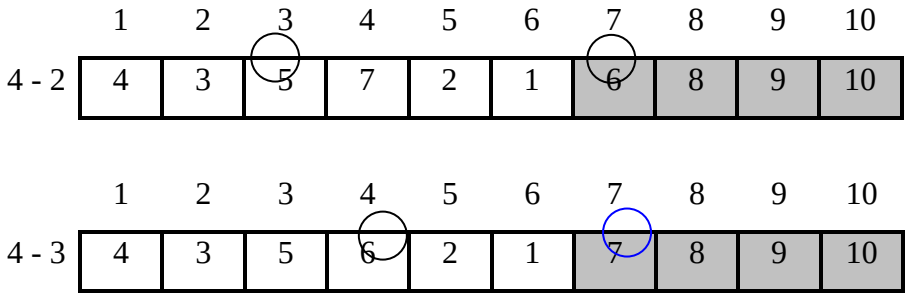
```
                        k:=j;
                      end;
    a[k]:=a[i]; {nechávám stejne}
    a[i]:=max;  {max je 8}
  end;
end;
```

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

4.

```
for i:=n downto 2 do  {zmensuji i na 7}
  begin
    max:=a[i];    {do max ulozim hodnotu 4}
    k:=i;         {do k ulozim i coz je 7}
    for j:=1 to 7 do  {prohledavam od zacatku,
postupne procházím}
    if a[j] > max then begin    {jestlize hodnota je
vetsi nez max
                                 - neni splněna}
                      max:=a[j];
                      k:=j;
                    end;
    a[k]:=a[i]; {prehozeni prvku}
    a[i]:=max;  {pri poslednim pruchodu je max 7}
  end;
end;
```

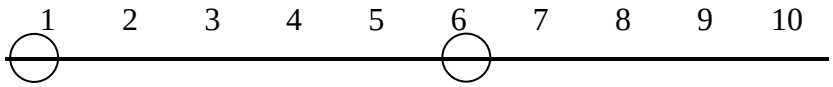
	1	2	3	4	5	6	7	8	9	10
4 - 1	4	3	6	7	2	1	5	8	9	10

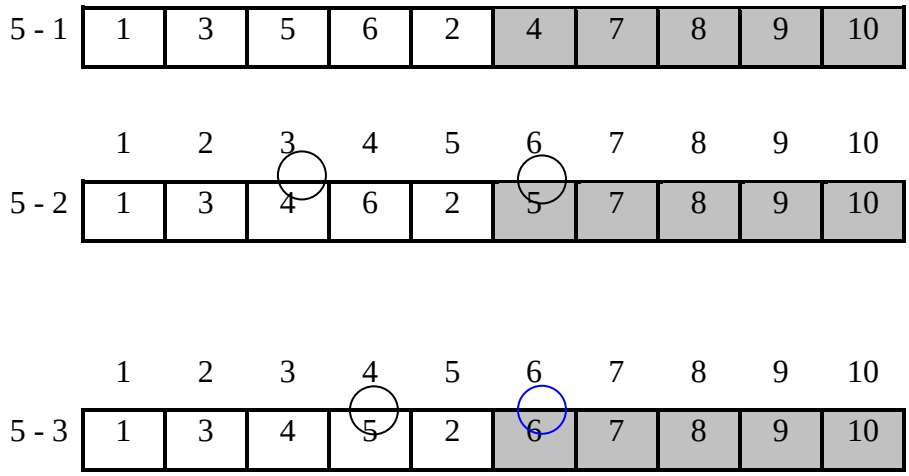


Zde jsme našli další maximum což je 7 a pokračujeme cyklem od začátku!

5.

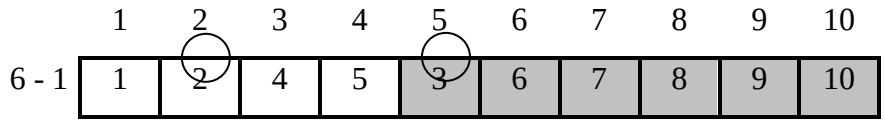
```
for i:=n downto 2 do  {zmensuji i na 6}
   begin
     max:=a[i];     {do max ulozim hodnotu 1}
     k:=i;          {do k ulozim i coz je 6}
     for j:=1 to 6 do  {prohledavam od zacatku,
postupne procházím}
     if a[j] > max then begin    {jestlize hodnota je
vetsi nez max
                                    - neni splněna}
                          max:=a[j];
                          k:=j;
                        end;
     a[k]:=a[i]; {prehozeni prvku}
     a[i]:=max;  {pri poslednim pruchodu je max 6}

   end;
end;
```

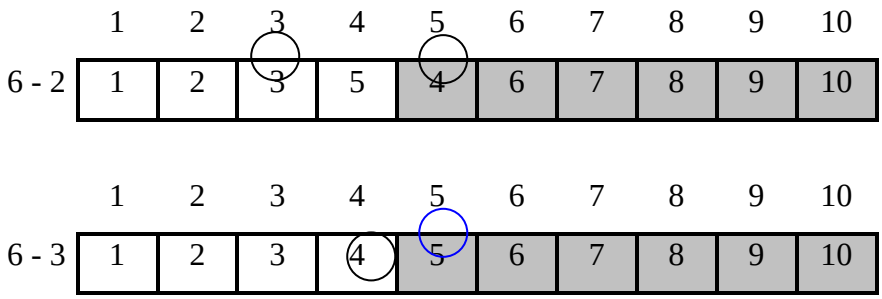




6.

```
for i:=n downto 2 do  {zmensuji i na 5}
   begin
     max:=a[i];     {do max ulozim hodnotu 2}
     k:=i;          {do k ulozim i coz je 5}
     for j:=1 to 5 do  {prohledavam od zacatku,
postupne procházím}
     if a[j] > max then begin    {jestlize hodnota je
vetsi nez max
                                    - neni splněna}
                          max:=a[j];
                          k:=j;
                        end;
     a[k]:=a[i]; {prehozeni prvku}
     a[i]:=max;  {pri poslednim pruchodu je max 5}

   end;
end;
```





Tím jsme získali seřazenou posloupnost!

10.3. Insert-Sort

Algoritmus Insert Sort pracuje tak, že vyhledává index, kam se má prvek zařadit a zároveň zbývající prvky posune o jednu pozici směrem ke konci pole.

Insert-Sort

```
procedure Insert_Sort(var a:pole; n:integer);
var  i,j:integer;
pom:integer;

begin  for i:=2 to n do
  begin
    pom:=a[i];
    j:=i-1;
    a[0]:=pom;            ... nastavení zarážky
    while (pom<a[j]) do
    begin
      a[j+1]:=a[j];       ... nalezení a uvolnění
místa
      j:=j-1;
```

```
    end;
    a[j+1]:=pom;          ... vložení prvku
  end;
end;
```

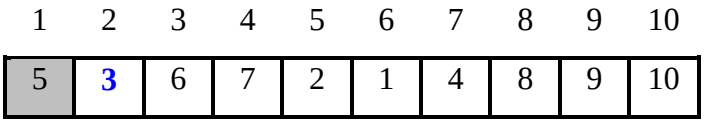
Princip algoritmu Insert Sort

Pole je rozděleno na seřazenou a neseřazenou část. V seřazené části se nalezne pozice, na kterou přijde vložit první prvek z neseřazené části a od této pozice až do konce seřazené části se prvky odsunou. Na začátku má seřazená část délku jedna.

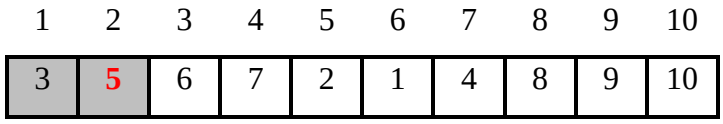
Řešený příklad 24:
1.

```
begin
for i:=2 to n do
  begin
    pom:=a[i];
{do pom se přiřadí hodnota a[2] čili 3}
    j:=i-1;          {j se přiřadí 1}
    a[0]:=pom;
... nastavení zarážky {a[0]se uloží číslo 3}
    while (pom<a[j]) do
    begin
      a[j+1]:=a[j];
... nalezení a uvolnění místa

{do a[2] se uloží hodnota a[1] čili 5}
      j:=j-1;        {snížení indexu j čili 0}
    end;
    a[j+1]:=pom;          ... vložení prvku do a[1]
  end;
```

```
end;
```



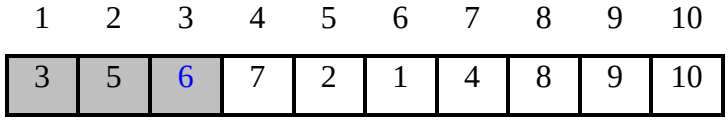
1.



2.

```
begin
for i:=3 to n do
  begin
    pom:=a[i];
{do pom se přiřadí hodnota a[3] čili 6}
    j:=i-1;          {j se přiřadí 2}
    a[0]:=pom;              ... nastavení zarážky
    while (pom<a[j]) do       {podmínka neplatí}
    begin
      a[j+1]:=a[j];
      j:=j-1;
    end;
    a[j+1]:=pom;            ... vložení prvku do a[3]
  end;
end;
```

2.



3.

```
begin
for i:=4 to n do
  begin
    pom:=a[i];      {do pom se přiřadí hodnota a[4] čili
7}
    j:=i-1;         {j se přiřadí 3}
    a[0]:=pom;            ... nastavení zarážky
    while (pom<a[j]) do       {podmínka neplatí}
    begin
      a[j+1]:=a[j];
      j:=j-1;
    end;
    a[j+1]:=pom;          ... vložení prvku do a[4]
  end;
end;
```

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

4.

```
begin
for i:=5 to n do
  begin
    pom:=a[i];
{do pom se přiřadí hodnota a[5] čili 2}
    j:=i-1;         {j se přiřadí 4}
    a[0]:=pom;            ... nastavení zarážky
```

```
    while (pom<a[j]) do
    begin
      a[j+1]:=a[j];       ... nalezení a uvolnění místa

{do a[5] se uloží hodnota a[4] čili 7}
      j:=j-1;        {snížení indexu j čili 3}
    end;
    a[j+1]:=pom;          ... vložení prvku do a[4]
  end;
end;
```

4.

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

Dochází k postupnému posunutí prvků v pořadí jak jdou za sebou!

```
while (pom<a[j]) do
    begin
      a[j+1]:=a[j];
      j:=j-1;
end;
```

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

5.

```
begin
for i:=6 to n do
  begin
    pom:=a[i];
{do pom se přiřadí hodnota a[6] čili 1}
    j:=i-1;          {j se přiřadí 5}
    a[0]:=pom;              ... nastavení zarážky
    while (pom<a[j]) do
    begin
      a[j+1]:=a[j];       ... nalezení a uvolnění místa

{do a[6] se uloží hodnota a[5] čili 7}
      j:=j-1;        {snížení indexu j čili 4}
    end;
    a[j+1]:=pom;          ... vložení prvku do a[5]
  end;
end;
```

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

Dochází k postupnému posunutí prvků v pořadí jak jdou za sebou!

```
while (pom<a[j]) do
    begin
      a[j+1]:=a[j];
      j:=j-1;
end;
```

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

6.

```
begin
for i:=7 to n do
  begin
    pom:=a[i];
{do pom se přiřadí hodnota a[7] čili 4}
    j:=i-1;          {j se přiřadí 6}
    a[0]:=pom;              ... nastavení zarážky
    while (pom<a[j]) do
    begin
      a[j+1]:=a[j];       ... nalezení a uvolnění místa

{do a[7] se uloží hodnota a[6] čili 7}
      j:=j-1;        {snížení indexu j čili 5}
    end;
    a[j+1]:=pom;           ... vložení prvku do a[6]
  end;
end;
```

	1	2	3	4	5	6	7	8	9	10
6.	1	2	3	**4**	5	6	**7**	8	9	10

Dochází k postupnému posunutí prvků v pořadí jak jdou za sebou!

```
while (pom<a[j]) do
    begin
      a[j+1]:=a[j];
      j:=j-1;
end;
```

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

Vznikla již setříděná posloupnost.

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

10.4. Heap-Sort

Jedná se o nejuniverzálnější třídící algoritmus. Je to nejrychlejší nerekurzivní algoritmus z dosud vyčtených. Pracuje na principu kontroly platnosti binárního stromu. Binární strom, je struktura prvků, kde každý uzel může mít maximálně dva potomky.

Princip haldy

Vezmeme-li náš nesetříděný seznam prvků jako binární strom, můžeme prvek na první pozici pokládat jako jeho kořen, prvek na druhé pozici bude levým potomkem kořene, prvek na třetí pozici bude pravým potomkem kořene, prvek na čtvrté pozici bude levým potomkem levého potomka kořene, prvek na páté pozici bude pravým potomkem levého potomka kořene atd. Obecně platí, že pro prvek na i-té pozici najdeme levého potomka na pozici 2*i a pravého potomka na pozici 2*i +1.

Podmínka platnosti binárního stromu zní: potomek musí být vždy nižší než rodičovský prvek. Při kontrole podmínky vlastně zjišťujeme, zda je větší potomek levý nebo pravý, a ten pak zaměníme s rodičovským. Pro první prvek najdeme oba potomky, a zjistíme platnost podmínky. To opakujeme pro druhý, třetí atd. Strom bude mít platnou podmínku (tedy že seznam bude seřízený) tehdy, jestliže zkontrolujeme podmínky pro první polovinu seznamu. Vezmeme-li totiž prostřední prvek seznamu, víme, že jsme vybrali zároveň poslední prvek stromu, který má ještě alespoň jednoho potomka. Dále už jsou pouze koncové prvky stromu (ty bez potomků) jímž se říká listy, a pro něž nemusí již žádná podmínka platit.

Postup přidání nové hodnoty do haldy (vytvoření haldy):
Vytvoříme jeden nový uzel, ten připojíme do poslední hladiny co nejvíce vlevo, aby byl zachován tvar haldy. Do nového uzlu vložíme hodnotu a zajistíme správné uspořádání hodnot v haldě. Zkontrolujeme, zda nová hodnota je větší než hodnota předchůdce. Pokud ano, je halda v pořádku a nic nemusíme dělat. V opačném případě je nutné vyměnit data uložená v těchto dvou uzlech. Stejným způsobem postupujeme v haldě výše. Formování haldy končí v okamžiku, když se nové přidaná hodnota dostane postupnými výměnami buď do uzlu, jehož předchůdce již obsahuje hodnotu menší, nebo až do kořene.
Postup vypuštění minimální hodnoty z haldy:
Minimální hodnotu uloženou v kořeni smažeme (odebereme). Haldu nyní chceme zmenšit o jeden uzel. Musí to být uzel umístěný v poslední hladině co nejvíce vpravo, aby zůstal správný tvar haldy. Hodnotu z tohoto uzlu

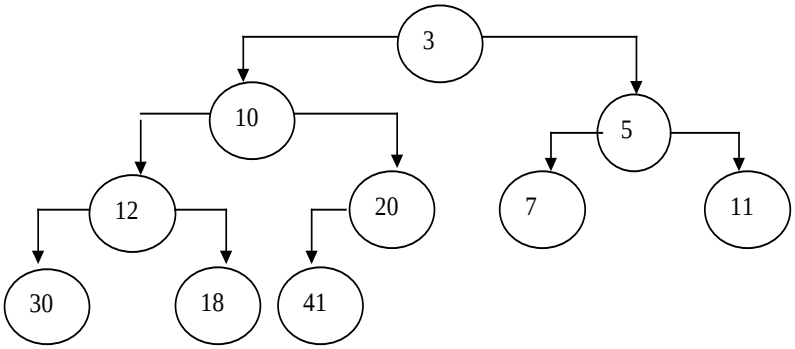
umístíme do kořene haldy. Tím jsme získali strom o správném počtu uzlů, se správnou množinou uložených údajů a se správným tvarem haldy. Zbývá pouze zajistit správné uspořádání hodnot(regulérnost haldy) vzájemnými výměnami. Začneme od kořene stromu a postupujeme směrem k listům. Hodnotu H v kořeni porovnáme s hodnotami obou jeho následníků. Je-li menší než oba následníci, halda je v pořádku a jsme hotovi. Jinak zaměníme hodnotu H z kořene s menší z hodnot následníků. Tím jsou nerovnosti mezi uzly první a druhé hladiny napraveny a pokračujeme stejným způsobem o hladinu níže od uzlu, do kterého se právě přesunula hodnota H. Postupné výměny hodnot uzlů končí ve chvíli, kdy se hodnota H dostane do místa, kde je menší než hodnoty následníků, nebo až do listu.

Postupným rozebíráním haldy dostaneme seřazenou posloupnost hodnot.

Realizace v programu:
V programech ukládáme haldu do pole, což není pro binární stromy obvyklé, ale v případě třídění je to efektivní co do rychlosti i paměťových nároků.
Uzly haldy si očíslujeme po hladinách zleva doprava čísly od 1 do N. Kořen má číslo 1, jeho následníci 2 a 3, atd. Tato čísla slouží jako indexy pro uložení uzlů v poli. Zvolené očíslování má jednu velmi důležitou vlastnost: následníci uzlu s číslem **i** mají čísla **2*i** a **2*i+1**

Řešený příklad 25:

Např.



Reprezentace stromu v paměti:

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.
3	10	5	12	20	7	11	30	18	41

První prvek pole obsahuje hodnotu z kořene stromu.

Následníci uzlu, který leží na **i_té** pozici, se v poli nacházejí na pozici **2*i** (levý) a **2*i + 1** (pravý).

10.5. Quick-Sort

Je asi nejrychlejším algoritmem ze všech, ale má obrovskou nevýhodu – je rekurzivní. Rekurzivní algoritmy jsou takové, které při svém průběhu volají samy sebe. Problematika rekurze je náplní mnohých knih a je nad rámec tohoto článku ji přiblížit.

Zjednodušeně se dá říct, že najdeme v seznamu medián (prvek na prostřední pozici) a všechny menší prvky zařadíme vlevo a větší prvky vpravo. Pro každý tento rozpůlený interval se znovu zavolá quicksort, až do okamžiku, kdy bude medián jediným členem v intervalu.

Nevýhoda je zřejmá – pro velká pole se může stát, že se algoritmus tak vnoří do sebe, že přeteče systémový zásobník, do kterého se zaznamenává počet volání, a třídění se zhroutí.

Analýza algoritmu Quick Sort

Základní myšlenka:

Zvolíme číslo X a prvky pole přerovnáme v poli tak, aby v levé části pole byly pouze prvky menší nebo rovné X a v pravé části prvky větší nebo rovné X. Po tomto rozdělení platí, že prvky ležící v levé části pole tam zůstanou i po úplném setřídění celého pole. Totéž platí i pravou část pole. Pak stačí samostatně setřídit levý a pravý úsek pole, jedná se dvě dílčí menší úlohy naprosto stejného typu, jako byla úloha původní - řešíme naprosto stejným způsobem. Pole postupně dělíme tak dlouho, dokud

nezískáme úseky délky 1 (ty jsou samy o sobě již seřazené a nemusíme s nimi nic dělat).

Volba hodnoty X:

Na vhodné volbě X závisí rychlost algoritmu. Pokud bychom za X zvolili pokaždé nejmenší (největší) prvek zpracovávaného úseku, rozdělí se pole v prvním kroku na úseky délky 1 a N-1, ve druhém kroku by se větší z nich opět dělil na úseky délek 1 a N-2 atd. Časová složitost by v tomto případě byla $O(N^2)$.

Nejlepší by bylo zvolit za číslo X pokaždé tzv. **medián** právě zpracovávaného úseku. Medián je číslo s prostřední hodnotou ze všech čísel úseku, např. medián z dvaceti čísel je desáté největší číslo z nich. Přesné vyhledávání mediánu je dosti pracné a v quicksortu se nepoužívá.

Nejjednodušší je vzít za X vždy první prvek úseku, nebo za X zvolit prostřední prvek zpracovávaného úseku nebo X vypočítat jako aritmetický průměr několika málo čísel (např. prvního, posledního a prostředního), aby se snížila pravděpodobnost té nejméně příznivé volby. Časová složitost při těchto volbách je O(N*logN).

Quick-Sort a rekurze

V quick sortu je X stanoveno jako **prostřední prvek úseku**: (L + R) div 2.

Přesuny prvků v poli do dvou úseků v závislosti na X:

K přerovnání pole potřebujeme dva pomocné indexy ukazující do právě pracovávaného úseku pole. Index I začíná na levém okraji úseku a zvětšuje se tak dlouho, dokud v pli nenajdeme první prvek větší než X (ten nepatří do levé části a má se přesunout někam vpravo. Index J postupuje od

pravého okraje úseku smětem doleva tak dlouho, až narazí na prvek menší než X (ten se má přesunout do levé části). Oba nalezené prvky spolu vyměníme. Pak pokračujeme stejným způsobem v pohybu indexů I a J směrem ke středu a s výměnami prvků tak dlouho, dokud se oba indexy nesetkají. V tom okamžiku je celý úsek rozdělen na levou část s menšími prvky a pravou část s většími prvky.

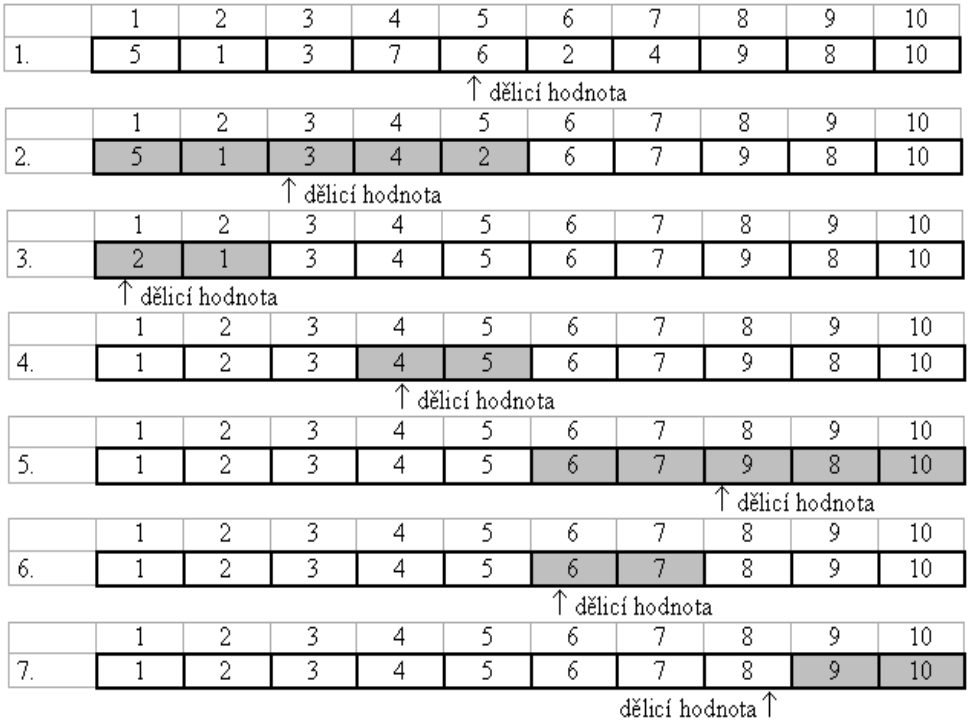
```
procedure QuickSort(var pole:TPole; Zac,Kon:integer);
{procedura setridi v poli usek od indexu Zac do indexu Kon}
var P: integer; {hodnota pro rozdeleni pole na useky - pivot}
    pom: integer; {pomocna promenna pro vymenu prvku}
    i,j: integer; {pracovni indexy pro vymezeni casti pole}
begin
  i:=Zac; {na zacatku zabiraji mezni indexy cele pole}
  j:=Kon;
  P:=pole[(Zac+Kon) div 2]; {urcime si pivotni prvek - vezmeme prostredni prvek pole}{idealni pivot by byl median - prostredni z hodnot v poli}

  repeat {nalevo od pivota budeme radit mensi prvky, napravo vetsi prvky nez pivot}
    while pole[i]<P do inc(i); {posouvame levý index, dokud neni na prvku vetsim nez pivot}

    while pole[j]>P do dec(j); {posouvame pravy index, dokud neni na prvku mensim nez pivot}
```

```
    if i<j then {pokud se indexy neprekrizily a nejsou shodne}
    begin
      pom:=pole[i]; {vymenime nalezene prvky}
      pole[i]:=pole[j];
      pole[j]:=pom;
      inc(i); dec(j); {a posuneme indexy na dalsi prvky}
    end
    else if i=j then {pokud se indexy sesly (ukazuji na pivota)}
    begin
      inc(i); {posunume je jeste dale, aby vymezovaly roztridene
poloviny pole}
      dec(j); {a doslo k prekrizeni, coz vede k ukonceni cyklu}
    end
  until i>j; {opakujeme, dokud nejsou obeti casti pole roztrideny
podle pivota}
  {Pole [Zac,Kon] je rozdeleno na 2 useky [Zac,j] a [i,Kon],
  ktere zpracujeme rekurzivne (opet touto procedurou)}
  if Zac<j then QuickSort(pole,Zac,j); {ma smysl tridit
nejmene dva prvky}
  if i<Kon then QuickSort(pole,i,Kon);
end;
```

Princip algoritmu

Základem je rozdělení pole do dvou částí a přeskládání prvků tak, aby všechny prvky v levé byly menší než v pravé. Tento algoritmus se potom aplikuje rekurzivně na levou a pravou polovinu pole.

Řešený příklad 26:



Základní problém - nalezení dělící hodnoty:

Hodnotu není možné určit algoritmicky - hledání by znehodnotilo celou metodu. Proto se "uhodne" jako hodnota prvku ležícího uprostřed řazené části. V nejhorším případě se pole rozdělí na jeden prvek a zbytek.

10.6. Výhody a nevýhody

Je zřejmé, že algoritmy probrané v prvních třech podkapitolách (Bubble-Sort, Select-Sort, Insert-Sort) patří k méně efektivním a pro profesionální

využití nevhodným algoritmům. Jejich časová složitost principiálně kvadratická – $o(n^2)$. To znamená, že v průměrném případě stoupá čas potřebný k seřazení pole kvadraticky vzhledem k počtu prvků. Jde tedy sice o postupy poměrně efektivní (v informatice je mnoho problémů, které neumíme řešit v polynomiálním čase, např. optimalizační problémy), ale přesto není tato kvadratická složitost tím nejlepším možným. Přesto existují i případy, kdy tyto postupy mohou být relativně zajímavé pro využití v praxi. Shrňme je nyní pro každý algoritmus zvlášť.

Bubble-Sort:

- velmi dobrá efektivita pro téměř seřazené vstupní posloupnosti
- díky přesunům mezi sousedními prvky je velmi vhodný pro použití u dynamických datových struktur jako jsou spojové seznamy
- jednoduchá implementace a možnost přidat dodatečné modifikace pro speciální typy vstupních polí

Insert-Sort:

- vhodný pro dynamické datové struktury (vkládání mezi prvky), odpadá časově náročné přesouvání celého zbytku pole – u statických struktur je to náročná operace

Select-Sort:

- dává nejstabilnější časové výsledky – provádí se vždy (u libovolného pole stejné velikosti) přesně stejný postup

Samozřejmě pro profesionální využití jsou použitelné dvě poslední metody (Quick-Sort a Heap-Sort). Více používanou metodou je Quick-Sort, který v průměrném případě má o něco lepší složitost. Přesto je složitost obou o((log n)*n) – tedy logaritmicko-lineární. Pokud si dokážete představit graf takovéto funkce, je jasné, že složitost této metody je výrazně lepší než u třech metod triviálních. I v řádech tisíců řazených prvků bude metoda hotova velmi rychle narozdíl od kvadraticky rostoucí funkce. Problém QuickSortu je možná jeho rekurzivní založení a pak také, že pronejhorší případ může degradovat až na kvadratickou složitost. Tuto nevýhodu nemá právě Heap-Sort, který pro všechny případy logaritmicko-lineární, avšak v průměrném případě má o něco horší koeficienty dané funkce.

Nejdůležitější probrané pojmy:

- triviální algoritmy třídění – Bubble-Sort, Insert-Sort, Select-Sort
- efektivní algoritmy třídění – Quick-Sort, Heap-Sort

Úkoly a otázky k textu:

1. Vyčíslete přesný počet operací, které vám nad vámi zvoleným polem prvků provedou postupně všechny uvedené algoritmy řazení.

11. Logika

Logika je oproti jiným disciplínám teoretické informatiky vědou velmi starou a její kořeny lze najít již ve starověku. Tehdy byla spjata spíše s filozofickými otázkami než s rodící se matematikou. Přesto otázky, které byly v té době aktuální, jsou v mnohém stejné jako je uplatnění logiky v informatice. Na logiku je možné se dívat v kontextu mnoha věd a výsledný pohled může být velmi odlišný. Touto vědou se zabývají filozofové, právníci, matematici i informatici a jejich zájem a cíl bývá velmi odlišný. Čtenář může sám posoudit, jak odlišné mohou tyto pohledy být -stačí si pročíst například filozoficky a informaticky orientovanou knihu o logice. Přesto lze najít společnou snahu o modelování lidského úsudku. I když matematik logiku spíše používá k formulaci a dokazování vlastností matematických objektů, pro informatika je logika nejen nástrojem, ale i plnoprávnou disciplínou zkoumanou v rámci teoretické informatiky.

Přístupy při modelování úsudku založené na logice patří do rodiny symbolických přístupů, existují pak také tzv. konekcionistické přístupy (např. umělé neuronové sítě), ale těmito přístupy se nebudeme zabývat.

Tyto přístupy se často inspirují biologickými procesy a snaží se je modelovat matematicky. Často se objevují také velmi úspěšné kombinace obou přístupů.
Díky své bohaté historii je samozřejmě logika velmi širokým oborem a v tomto článku bychom se chtěli zabývat několika důležitými otázkami z informatického pohledu.
Reprezentace znalostí klasickou logikou -výroková a predikátová logika.
Automatizovaná dedukce -formální systémy pro dokazování vět.
Vícehodnotová logika -možnosti modelování neurčitosti pomocí fuzzy logiky.

Chceme opět jako v předchozích článcích zdůraznit, že nám nejde o vytvoření rigorózní monografie, ale o text, který čtenáři přinese chuť do dalšího studia i do výuky logiky.

11.1. Reprezentace znalostí klasickou logikou

Klasickou logikou myslíme formalismus, který je v různých podobách znám již stovky let. Lidé jsou schopni komunikovat a formulovat své myšlenky v přirozeném jazyce a dále je zpracovávat a dedukovat závěry. Právě přirozený jazyk je však poměrně složitý pro stroje, které by měly simulovat tento způsob reprezentace znalostí. Proto logika nabízí různé úrovně zjednodušení formulace znalostí do co nejmenšího množství exaktně definovaných symbolů. Druhou stránkou je motivace, abychom z těchto statických znalostí reprezentovaných symboly dokázali také

odvozovat závěry. K tomu slouží různé symbolické metody, z nichž některé si popíšeme v tomto článku. Proveďme analogii s právními předpisy. Samotné zapsané zákony jsou sice velice užitečné, ale k čemu by nám byly, pokud by nebylo možné je interpretovat a zjišťovat, zda se například některý subjekt nedopustil porušení zákona. To můžeme provést jedině tak, že dedukujeme závěr (zda někdo jedná protizákonně nebo ne) z předpokladů (fakta popisující situaci a dané zákony).

Nejjednodušším formalismem vhodným pro reprezentaci znalostí je výroková logika. U každé logiky si musíme uvědomit, že má svou syntaxi a sémantiku. Zjednodušeně můžeme říci, že syntaxe je definicí symbolů a jejich používání (bez ohledu na význam těchto symbolů). Sémantika je pak chováním těchto symbolů s ohledem na smysl daných znalostí (např. pravdivost daných tvrzení). Právě možnost oddělit syntaxi a sémantiku ,po formulaci a dokázání potřebných vztahů mezi nimi, dává logice význam v informatice a pro automatizaci úsudku.

Syntaxe výrokové logiky pracuje se symboly zastupujícími elementární výroky (např. a,b,c,...), logické konstanty (pravda a nepravda -$\top, \bot$) a dále je umožňuje spojovat do složitějších formulí pomocí symbolů logických spojek a pomocných symbolů (např. $\wedge$ konjunkce, $\rightarrow$ implikace). Sémantika pak popisuje smysl těchto použitých symbol a pojmy, které souvisejí s popisem tohoto smyslu (tzv. interpretací formulí -v případě klasické logiky to může být buď formule pravdivá nebo nepravdivá). Jednotlivým výrokům můžeme přiřadit logickou hodnotu podle toho, jak se tyto syntaktické elementy chovají v našem modelovaném případě z reality, příp. to můžeme udělat také zcela nesmyslně. V obou případech

výrok A	výrok B	vylučovací nebo	ekvivalence	negace ekvivalence
0	0	0	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

pak můžeme uvažovat jaká bude interpretace celých složených formulí. Zaleží to na použitých unárních a binárních spojkách. Čtenář se zřejmě setkal se spojkami jako je negace ¬(opak ve smyslu pravdivosti), konjunkce ∧ (současná platnost výroků), disjunkce ∨ (platnost alespoň jednoho z výroků), implikace → (podmínka), ekvivalence ↔ (identická pravdivost výroků). Binárních spojek existuje samozřejmě více -v elektrotechnice se kupříkladu používají spojky jako je negace konjunkce (nand). Formule tedy v tomto případě platí (je interpretována jako pravdivá), pokud neplatí dva výroky současně (jinak řečeno pokud alespoň jeden neplatí). Druhým případem, který se zase vyskytuje často v reálných situacích je tzv. vylučovací nebo. V přirozené řeči často používáme výrok typu ”buď ... anebo ...”, který spíše odpovídá situaci, že musí platit alespoň jeden z výroků, ale zároveň nesmí platit oba současně. Nejde tedy o disjunkci, ale spojku, kterou někdy také označujeme jako negaci ekvivalence. Interpretaci logických spojek můžeme přehledně realizovat pomocí tabulky, ve které jsou vypsány všechny možné kombinace ohodnocení pravdivosti elementárních výroků. Interpretaci pravda označíme 1 a nepravda 0. Podívejme se na příklad identického chování negace ekvivalence a vylučovacího nebo.

V klasické logice existuje pouze konečný (a navíc velmi omezený) počet spojek. Stačí se zamyslet nad všemi možnostmi, které mohou nastat a

dojdeme k tomu, že pro 2 výroky ohodnocené 4 možnostmi pravda/nepravda existuje 2^4 možností dvouhodnotové interpretace tedy 16 možných spojek.

Sémantika jazyka výrokové logiky umožňuje na základě interpretace formule definovat další důležité pojmy.

Pokusme se nyní na příkladu namodelovat pomocí výrokové logiky jednoduchou situaci. Příklad 1:

K soudu byli předvedeni tři podezřelí z loupeže -A, B a C. Při výslechu se zjistily tyto skutečnosti:

Do případu nebyl zapleten nikdo jiný než A, B a C.

A pracuje vždycky alespoň s jedním společníkem.

C je nevinen.

Nejprve si musíme stanovit, co jsou elementární výroky. Jelikož se vyjadřujeme k vině a nevině podezřelých, bude rozumné pomocí výroků A, B a C symbolizovat, že daný podezřelý je vinen nebo nevinen. Pak můžeme zapsat větu 1. jako složený výrok vyjadřující pomocí disjunkce, že alespoň jeden z podezřelých je vinen. Druhá věta může být popsána formulí pomocí implikace (podmínky), která říká, že je-li vinen A, pak musí být vinen také alespoň jeden z podezřelých B, C. Výsledné formule výrokové logiky jsou tedy následující:

1. $A \vee B \vee C$, 2. $A \rightarrow (B \vee C)$, 3. $\neg C$

Samotná reprezentace znalostí je pouze polovičním úspěchem při pokusu o modelování úsudku. Dalším nevyhnutelným krokem je možnost ověřovat, zda nějaké tvrzení vyplývá z daných předpokladů. K tomu si nejprve

musíme definovat další pojmy sémantiky. Jde o tzv. splnitelnost (nesplnitelnost) a platnost formulí. Splnitelná formule je laicky řečeno taková, která má vůbec smysl pro určité ohodnocení výroků. Tedy taková formule musí alespoň pro nějakou kombinaci ohodnocení výroků pomocí pravda/nepravda dávat interpretaci pravda. Jinak je taková formule nesplnitelná resp. v logice se takové formuli říká kontradikce. Pokud bychom se obrátili zpátky na interpretační tabulku, pak by tabulka ve sloupci interpretace splnitelné formule musela mít alespoň v jednom řádku symbol 1.

Některé ze splnitelných formulí pak mohou být ještě na vyšším sémantickém stupni a mohou být pravdivé pro všechna možná ohodnocení. Takovým formulím se pak říká platná formule resp. v logice je zažitý pojem tautologie. Podívejme se na příklad.

Příklad 2: Mějme platné tvrzení: ”Pokud prší, beru si deštník.” Jestliže namodelujeme tvrzení, že ”prší” pomocí P a ”beru si deštník” pomocí D, pak výsledná formule je následující:

$P \rightarrow D$

Uvažujme nyní o výroku: ”Pokud si neberu deštník, neprší.” Je takové tvrzení ekvivalentní prvnímu? Platí-li první tvrzení, mohu ho brát jako postulát, kterému se každý musí podřídit. Zdá se tedy logické, že když si deštník neberu a dodržuji přitom postulát, pak nemůže pršet. Mnohem formálněji bychom tento vztah mohli dokázat právě pomocí pojmu tautologie. Druhá forma tvrzení se dá zapsat jako $\neg D \rightarrow \neg P$. Pak využijeme spojky ekvivalence, která interpretuje jako pravdivé dvě formule se stejnou pravdivostní hodnotou a tedy vlastně popisuje

ekvivalentní chování dvou formulí z hlediska pravdivosti. Sestrojíme tedy takovou formuli a prokážeme, že je to tautologie pomocí interpretační tabulky. Implikace je nepravdivá pouze pro případ, kdy první formule je pravdivá a druhá nepravdivá to je v souladu s naším chápáním podmínky. Pokud je splněn předpoklad podmínky, pak závěr musí platit -jinak by nebyla formule pravdivá. V případech kdy podmínka splněna není, může i nemusí závěr platit a v obou případech je to v pořádku. Jelikož sestrojená ekvivalence obou formulí je pravdivá ve všech interpretacích, můžeme konstatovat, že jde opravdu o tautologii.

P	D	$P \rightarrow D$	$\neg D$	$\neg P$	$\neg D \rightarrow \neg P$	$(P \rightarrow D) \leftrightarrow (\neg D \rightarrow \neg P)$
0	0	1	1	1	1	1
0	1	1	0	1	1	1
1	0	0	1	0	0	1
1	1	1	0	0	1	1

Nyní se již konečně dostáváme ke klíčovému pojmu tzv. logického důsledku množiny předpokladů. Máme-li množinu předpokladů a závěr, můžeme se ptát, zda tento závěr je logickým důsledkem (vyplývá z) předpokladů. To platí v případě, že pro každé ohodnocení výroků s interpretací pravda pro všechny formule z množiny předpokladů je pravdivá také formule reprezentující závěr. Jde tedy o chování podobné implikaci. Závěr nebude logicky vyplývat z předpokladů, jen pokud se

najde takové ohodnocení výroků, kdy všechny předpoklady jsou pravdivé a závěr není pravdivý. Tato činnost, kdy odvozujeme závěry, se nazývá dedukce. Důležitým faktorem pak ještě zůstává tzv. konsistence předpokladů. Jde o to, že formulovaná vlastnost vyplývání degraduje na triviální situaci, pokud předpoklady nemají žádné ohodnocení, ve kterém by byly současně pravdivé. Takovéto množiny předpokladů jsou sporné (nekonsistentní) a z hlediska definice důsledku, je důsledkem této množiny jakákoliv formule. To samozřejmě v realitě není zcela odpovídající. Jednalo by se například o situaci, kdy určitý balík zákonů obsahuje dvě navzájem protichůdná nařízení a podle této množiny by pak jakékoliv jednání bylo v souladu s tímto zákonem.

Příklad 3: Pokusme se nyní jednoduše pomocí tabulky prověřit, zda z množiny předpokladů z příkladu

1. vyplývá vina či nevina některého z podezřelých. Můžeme vyloučit podezřelého C, protože jeho nevina je přímo obsažena v předpokladech a tudíž musí vyplývat z množiny předpokladů a zároveň nemůže vyplývat jeho vina, pokud není množina předpokladů sporná. Postačí tedy sestrojit tabulku pro jednotlivé předpoklady a pro formule $A,\neg A,B,\neg B$ ověříme, zda nesplňují vlastnost důsledku. V tabulce se vyskytuje také sloupec s hvězdičkami, který zvýrazňuje ohodnocení, kde jsou všechny předpoklady platné a tedy v těchto řádcích musíme kontrolovat interpretaci potenciálního důsledku. U závěrů pak vyznačujeme pomocí hvězdičky resp. lomítkem, zda skutečně splňuje resp. nesplňuje formule podmínku důsledku. Pokud ji splňují všechna zvýrazněná ohodnocení jde skutečně o vyvoditelný logický důsledek.

Z tabulky je patrné, že jediný logický důsledek z prověřovaných závěrů je, že B je vinen. O podezřelém A nemůžeme konstatovat na základě předložených předpokladů ani jeho vinu ani nevinu. Vezmeme-li v úvahu možnost, že A je vinen, pak by musel být vinen i B. Pokud A vinen není, zároveň C vinen není dle předpokladu a někdo vinen být musí, pak padá

A	B	C	A ∨ B ∨ C	A → (B ∨ C)	¬C		A		¬A		B		¬B	
0	0	0	0	1	1		0		1		0		1	
0	0	1	1	1	0		0		1		0		1	
0	1	0	1	1	1	*	0	/	1	*	1	*	0	/
0	1	1	1	1	0		0		1		1		0	
1	0	0	1	0	1		1		0		0		1	
1	0	1	1	1	0		1		0		0		1	
1	1	0	1	1	1	*	1	*	0	/	1	*	0	/
1	1	1	1	1	0		1		0		1		0	

vina na B. B je tedy vinen v každém případě a o A nás ale nic neopravňuje to tvrdit.

Na uvedeném příkladu dedukce může ilustrovat hned několik klíčových otázek a problémů, které s logikou a naším článkem souvisí.

1. Pomocí přesně definovaných postupů jsme schopni dedukovat důsledky zcela automatizovaně a to zvládnou nejen lidé, ale zejména je to vhodné pro stroje (počítače). Už námi řešený příklad nemusí být pro každého člověka jednoduchý a složitost dedukce roste jednak s počtem výroků a jednak s počtem předpokladů. Už z těchto důvodů nejsou prostředky logiky zbytečné.

2. Problém automatizované dedukce lze řešit různě efektivní metodami. V tomto článku jsme se setkali prozatím jen s tabulkovou metodou, která je sice velmi triviální a dokonce si asi čtenář dokáže představit, že by takovýto algoritmus dokázal naprogramovat, nicméně její jednoduchost je také její slabinou. Už na příkladu 2 a 3 můžeme vidět, že rozsah tabulky roste exponenciálně vzhledem k počtu elementárních výroků. S ohledem na poznatky teorie vyčíslitelnosti a složitosti víme, že exponenciální časová a prostorová složitost je pro klasické deterministické počítače prakticky nepoužitelná. Už pro 10 výrokových pro
měnných je možností ohodnocení přes 1000, pro 20 přes milion atd... Proto bylo a stále je vyvíjeno mnoho různorodých metod a celých formálních systémů, které jsou schopny automatizovat dedukci mnohem efektivněji.

2. Klasická výroková logika je jednou z nejjednodušších logik, což má své výhody i nevýhody. Výhodou je její transparentnost, pochopitelnost a především vlastnost rozhodnutelnosti. Rozhodnutelnost ve smyslu schopnosti o dané formuli říci jednoznačně, zda je splnitelná nebo ne na základě algoritmu, je důležitou vlastností pro automatizaci dedukce. Složitější logiky tuto vlastnost mít nemusejí. Nevýhodou výrokové logiky je naopak neschopnost rozlišit objekty a vztahy mezi nimi, nemožnost kvantifikovat vztahy, pojmout proměnlivost pravdivosti v čase a podobně. V neposlední řadě je u klasických logik nevýhodou jejich ”černobílé vidění světa”. Myslíme tím neschopnost zachytit jinou než absolutní pravdivost nebo nepravdivost. V reálném životě je mnoho situací, které nelze popsat jednoznačně. Například to zda je člověk spokojený, lze stěží popsat jen výrokem pravdivým nebo nepravdivým. Člověk může být spokojený v určitém stupni spokojenosti. Proto se v tomto článku chceme dotknout i tématu vícehodnotových logik, které jsou nyní velmi intenzivně zkoumány.

11.2. Predikátová logika

Ještě než se budeme blíže zabývat otázkou automatizace dedukce, chtěli bychom také stručně popsat logiku predikátovou. Predikátová logika je v podstatě zobecněním logiky výrokové a dává ji schopnost pracovat nejen s elementárními výroky, ale také rozlišit objekty a jejich vztahy. Na syntaktické úrovni se zavádí pojem termu a formule. Termem může být buď symbol zastupující objekt (konstanta) nebo funkci (funktor) nebo proměnná, za kterou lze dosadit libovolný term. Klíčovým pojmem je predikát, který jako argumenty může mít termy a tím vlastně umožňuje vytvářet vazby mezi termy. Například bychom chtěli prostřednictvím predikátu zachytit vazby mezi rodiči a jejich dětmi. Zavedli bychom predikát s názvem dítě, který má dva argumenty -dítě a jeho rodiče. Samozřejmě pracujeme se symboly, takže skutečnými argumenty jsou pouze konstanty reprezentující objekty -konkrétní děti, rodiče atd. Konstanty jsou nejjednoduššími termy. Termy tedy nevyjadřují narozdíl od predikátů vazby, ale zastupují symbolicky objekty modelované reality. Takovými konstantami by mohla být například jména dětí a jejich rodičů. Tedy bychom mohli sestavit velmi jednoduchou formuli dítě(johana, lucie), která by měla pomocí predikátu vyjadřovat znalost, že mezi symbolem reprezentujícím objekty johana a lucie je vztah dítěte a rodiče. Termy mohou být však složitější a to funktory a proměnné. Funk-tory jsou symbolickými ekvivalenty pro nám dobře známé funkce. Funkce může mít několik argumentů (opět termů) a její interpretací je pak požadovaná hodnota. Například goniometrická funkce cos by pro argument -symbol 0

(intepretovaný číslem 0) -vracela číslo 1 ($\cos(0) = 1$). Proměnné pak jsou symbolickým prvkem, do kterých mohou být dosazeny jiné termy. Jelikož interpretace termů a predikátů zaleží (narozdíl od logických spojek) na uživateli predikátové logiky, mohl by si sémantiku uživatel uzpůsobit dle svých požadavků. Mohl by tedy vytvořit absurdní intepretace, kde by například funkce s názvem cos byla interpretována pro argument -číslo 0 - třeba číslem 333 nebo zcela nečíselným objektem. Chceme tím říci, že predikátová logika je velmi flexibilní a je potřeba mít stále na paměti rozdíl mezi syntaxí a sémantikou. Symboly mají své interpretace (sémantiku), kterou v případě logiky predikátové značně ovlivňuje ten, kdo ji používá. A i pod zcela totožnými
symboly se tak může skrývat (nekonečně) mnoho různých významů.

Predikát může ze sémantického hlediska nabývat opět hodnotu pravda nebo nepravda. Jeho sémantickým operátorem je relace. Od úrovně predikátů výše se pak formule chovají jako ve výrokové logice a můžeme je tedy spojovat pomocí logických spojek. Jediným rozdílem je, že můžeme formule kvantifikovat a to buď univerzálním ($\forall x$) resp. existenčním ($\exists x$) kvantifikátorem. Ty pro zvolenou proměnnou pak zaručují, že kvantifikovaná formule bude pravdivá pro všechny resp. alespoň jeden objekt dosaditelný za proměnnou x.

Příklad 4: Chtěli bychom pomocí predikátové logiky namodelovat situaci, kdy libovolné dítě je šťastné, pokud má otce i matku. Zavedli bychom si predikátové symboly pro vlastnosti objektů stastny, muz a zena a dále pro vztah býti dítětem někoho -dite. Pomocí formule 1. pak můžeme vyjádřit, že pro každé dítě, ke kterému existuje objekt, jenž je ženou a zároveň dítě

je dítětem tohoto objektu a zároveň platí totéž pro objekt typu muž, pak platí, že toto dítě je šťastné. Mnohem jednodušší je pak vyjádřit, které objekty jsou dítě, žena atd.. (např. johana je dítě lucie -formule 2. -5.).

$\forall X[\exists Y\ [dite(X, Y) \wedge zena(Y)] \wedge \exists Y\ [dite(X, Y) \wedge muz(Y)] \rightarrow stastny(X)]$.

dite(johana, hashim).

dite(johana, lucie).

muz(hashim).

zena(lucie).

Můžeme pozorovat, že predikátová logika má mnohem vyšší expresivitu (vyjadřovací schopnost) než logika výroková. Zásadní je především možnost modelovat vazby mezi objekty pomocí predikátů. Silným prvkem je rovněž schopnost modelovat existenci. Formule 2.-5. z příkladu mohou připomínat řádky relační databáze. Relační databáze jsou založeny na prezentaci znalostí v relacích, které modelují rovněž vztahy mezi objekty. Skutečně bychom našli jistou analogii mezi tabulkou databáze (např. tabulka dětí) a jejich atributy (kdo je dítětem koho). Zkuste si představit databázovou tabulku studentů s atributy jméno, příjmení, bydliště, ročník. Taková tabulka relační databáze se dá vyjádřit predikátem student. Databáze by pak obsahovala třeba následující řádky (a jejich ekvivalentí vyjádření v predikátové logice): jan, novák, ostrava, 3 formule: student(jan, novák, ostrava, 3) jan, veselý, ostrava, 4 formule: student(jan, veselý, ostrava, 4) jiří, novák, ostrava, 1 formule:

student(jiří, novák, ostrava, 1) ... petr, novotný, praha, 2 formule: student(petr, novotný, praha, 2)

Klasické relační databáze jsou ale velmi úzkou podmnožinou způsobu reprezentace pomocí predikátové logiky. V databázích nemáme možnost používat proměnné a zejména logické spojky. Ty dokáží ušetřit mnoho prostoru -co by se muselo v databázi vyjádřit explicitně (třeba tisíci relacemi), lze elegantně zapsat jedním pravidlem a aplikovat dedukci. Samozřejmě, že již dávno začaly snahy o přibližování databázové technologie a logiky a to

v tzv. deduktivních databázích (např. systém Datalog). Jde o inteligentní databáze, které

kromě klasického explicitního vyjmenování vztahů umožňují také používání omezených logických prostředků a dedukce.

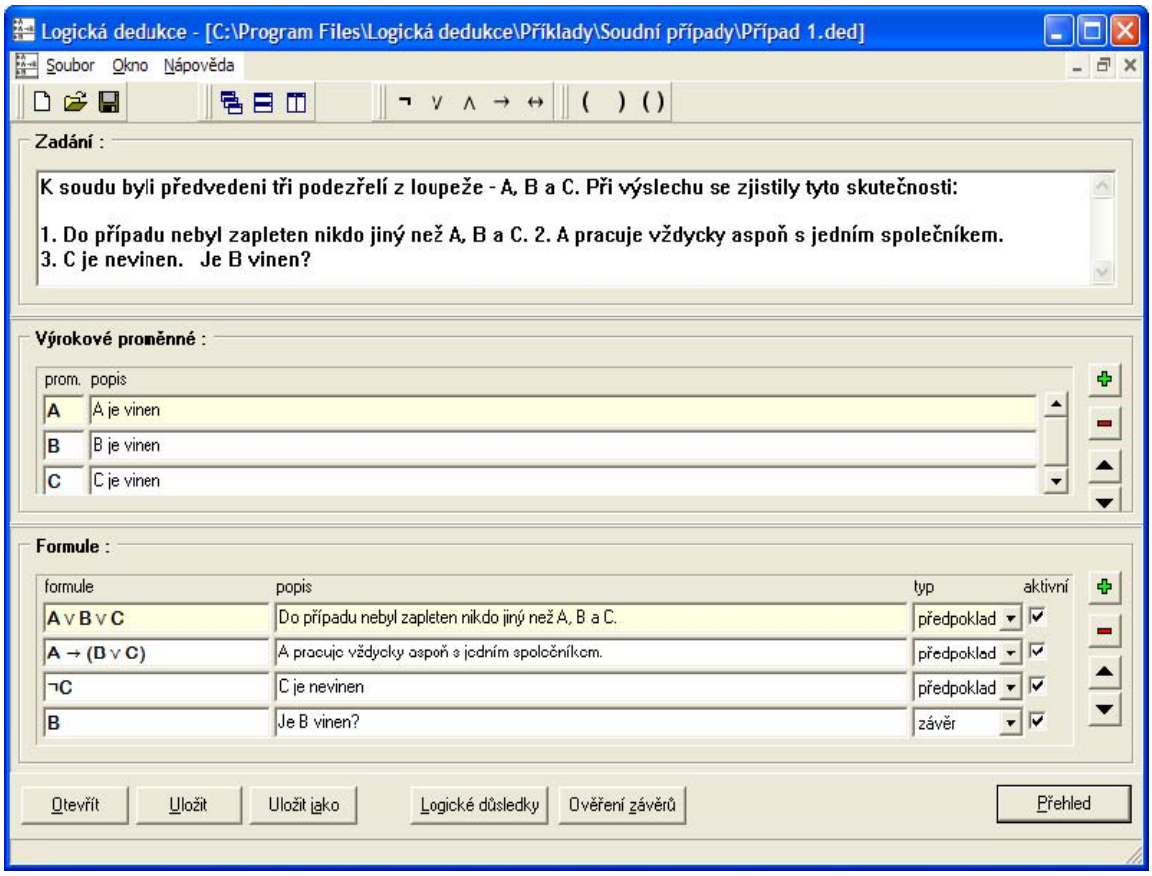
Jak už to ale bývá téměř všude v reálném životě, za výhody se platí určitými nevýhodami. Zatímco u výrokové logiky lze vždy rozhodnout (různě efektivně) o splnitelnosti dané formule, v predikátové logice je tento problém pouze částečně rozhodnutelný. Hlavním viníkem je potenciální možnost pracovat s nekonečnými doménami objektů (např. množina přirozených čísel -lze dosazovat za proměnné jakékoliv číslo). Potenciálně existuje možnost vytvořit také o něco složitější interpretační tabulku jako u logiky výrokové, ovšem takováto tabulka může být nekonečná. V případě univerzální kvantifikace formule musí tato formule platit pro všechny možné dosaditelné konstanty. Těch ale může být ve vztahu k nekonečným doménám také nekonečně mnoho a tudíž bych dostali tabulku s nekonečným počtem řádků. Proto existují snahy

predikátové logice ”něco vzít”, tj. odebrat jí některé vyjadřovací schopnosti při zachování některých výhod tak, aby se stala rozhodnutelná a zároveň existovaly efektivní algoritmy pro dedukci.

11.3. Automatizovaná dedukce

Automatizace dedukce vyžaduje najít efektivní algoritmy pro generování důsledku nebo pro prověřování konsistence množin formulí. V praktických situací jde o automatizované zjišťování, zda z logických axiómů (vybrané tautologie) a speciálních axiomů (formalizované předpoklady) vyplývá závěr. V zásadě existují metody sémantické a formální. Tabulky z předchozí kapitoly jsou typickou sémantickou metodou. U sémantických metod musíme provádět interpretaci formulí, což je s ohledem na již zmiňované obrovské množství ohodnocení elementárních výroků velmi neefektivní přístup. I když některé sémantické metody vylepšují tuto nevýhodu, v zásadě je sémantický přístup pro automatizaci zcela nevhodný. Druhý formální (syntaktický) přístup se snaží se zcela oprostit od interpretace (smyslu) a používat prověřená pravidla pro práci se symboly (formulemi) bez ohledu na to, co znamenají. To je v principu velice efektivní, protože časová složitost pak nezávisí na možných interpretacích, ale na velikosti předpokladů jako symbolických formulí. Tyto metody vyžadují tedy jisté ”know-how”, je složitější je pochopit, ale v konečném důsledku jsou zásadně lepší. Lze je rovněž naprogramovat a pak již mohou sloužit v aplikacích na ”inteligentní” usuzování. Právě pro složitost těchto metod (jsou dnes zcela v režii vysokoškolské výuky) je

nebudeme všechny ani naznačovat. Existuje však velmi dobře použitelná a precizně zpracovaná práce autorky Libuše Pavliskové (diplomová práce na Ostravské Univerzitě). Tato práce jednak obsahuje populární výklad problematiky dedukce a zejména je její součástí aplikace pro dedukci. Tato aplikace pracuje s výrokovou logikou (tudíž je dostatečně jednoduchá i pro středoškolskou výuku) a zároveň umožňuje zapsat libovolnou množinu předpokladů a buď generovat všechny možné důsledky nebo o konkrétním závěru zjistit, zda je to důsledek. Možná ještě významnější je pro výuku na středních školách existence velkého množství připravených příkladů s atraktivním zadáním jako jsou soudní případy podobné tomu z kapitoly 1. a další (samozřejmě i mnohem složitější). Didaktický text, popis aplikace i samotnou aplikaci pro operační systém Windows lze získat na adrese:
http://www1.osu.cz/home/habibal/dedukce/



Obrázek 1: Grafické rozhraní aplikace pro dedukci

Formální metody dedukce lze dále v principu provádět dvěma způsoby - přímo a ne-přímo. Zjednodušeně řečeno, při přímé metodě z předpokladů generujeme určitý důsledek pomocí odvozovacích (nebo jiných) pravidel. Při tomto způsobu tedy vždy hledáme potenciálně jiný důsledek podle toho, který závěr chceme dokázat. To v sobě samozřejmě skrývá jednu nevýhodu, jde především o možnost vygenerovat obrovské množství (potenciálně také nekonečné) různých důsledků a ten náš konkrétní se skrývá někde mezi nimi. To vede k neefektivitě a zejména se těžko hledají

různé pomocné postupy, jak dedukci urychlit. Proto se používají spíše postupy nepřímé. Jsou podobné principu známého nepřímého důkazu. K předpokladům přidáme náš zkoumaný závěr, ovšem opačný (tedy se spojkou negace). Jelikož závěr vyplývá pokud je jeho interpretace pravda ve všech případech, kdy jsou pravdivé předpoklady, pak při opačném (negovaném) závěru taková množina nemůže být splnitelná. Při nepřímé metodě tedy s negovaným závěrem chceme dokázat nesplnitelnost. Tím se vyhneme základnímu problému přímé metody, protože náš cíl při dokazování je vždy stejný. Je jím prokázání nesplnitelnosti bez ohledu na dokazovaný závěr. To činí dedukci efektivnější. I přestože je stále obrovské množství možností, jak můžeme pomocí pravidel nakládat s předpoklady, můžeme již najít obecné postupy, jak urychlit (v určitých případech) proces hledání důkazu. Například v nepřímé rezoluční metodě (kterou si krátce vysvětlíme níže) jde o to, najít prázdnou formuli. Dá se tedy použít heuristika (metoda, která nefunguje zcela dokonale, ale v mnoha případech urychluje řešení), že je výhodnější pro následující krok hledání použít formule s co nejmenším počtem unikátních atomů (atomem se rozumí ve výrokové logice elementární výrok bez spojek). To samozřejmě není vždy pravda, ale intuitivně to je docela rozumné, pokud chceme dospět k formuli prázdné.

V tomto článku s omezeným rozsahem bychom se ještě chtěli zmínit o dvou formálních metodách. První trochu méně známá, ale i přesto velmi účinná je metoda tablová. Je založena na rozkladu formule ve stromu podle logických spojek a hledání větví, které obsahují navzájem negativní atomy (stejný atom s negací a bez negace). To ji činí velice účinnou,

protože její časová náročnost je závislá jen na složitosti formule (počtu spojek). Bohužel v predikátové logice musí být obohacena o některé kroky, které její využití v praxi poněkud snižují. Druhou mnohem známější a široce používanou metodou je takzvaný rezoluční princip (rezoluce) resp. metody odvozené od něj. Myšlenku rezolučního principu formuloval v roce 1965 A. Robinson a od té doby byla velmi rozvinuta a zejména aplikována v různých systémech pro automatizované usuzování. I když existuje samozřejmě i její varianta pro logiku predikátovou, omezíme se zde pro jednoduchost pouze na logiku výrokovou. V klasickém pojetí rezoluce funguje pouze na formulích převedených do formy tzv. klauzulí (existují i zobecnění pro libovolné formule, ale zatím nejsou používány ani příliš prozkoumány). Klauzule je taková formule, kde se vyskytuje pouze binární spojka disjunkce, která spojuje jednotlivé atomy s negací nebo bez negace. Každou formuli lze pomocí logických pravidel (ekvivalencí -viz např. příklad 2) převést na ekvivalentní množinu klauzulí (může jich být i velmi mnoho). Rezoluční pravidlo pak umožňuje generovat ze dvou klauzulí obsahujících stejný atom (symbol elementárního výroku), který je v jednom případě s negací a v druhém bez negace, novou klauzuli spojenou disjunkcí obou výchozích klauzulí a zároveň vypustíme onen pár atomů, na kterých jsme prováděli rezoluci. Schématicky to lze znázornit následovně (atomy a jejich negace lze v klauzuli libovolně přesunovat bez změny interpretace dané formule). Rezoluční pravidlo

$$\frac{C_1 \vee x \qquad C_2 \vee \neg x}{C_1 \vee C_2}$$

C_1 a C_2 jsou zbývající části klauzulí a x je atom, na kterém rezoluci provádíme.

Máme-li pravidlo, jsme schopni z výchozích předpokladů (speciálních axiomů) a logických axiomů pomocí tohoto pravidla konstruovat sekvenci formulí, které říkáme důkaz. U každé syntaktické metody nebo formálního systému je důležité mít ověřeny dvě vlastnosti. Jde o to, aby metoda nebo systém, který obchází problematickou sémantiku tím, že pracuje pouze ”slepě” se symboly bez ohledu na jejich interpretaci, byl s touto sémantikou v souladu. První vlastnost, které se říká korektnost systému, zaručuje, že používaná pravidla generují pouze logické důsledky axiomů. Kdyby tomu tak nebylo, systém by z nekorektních pravidel generoval s předpoklady zcela nesouvisející formule (nesmyslné). Druhou vlastností je tzv. úplnost systému. Ta zaručuje, abychom pouze s danou množinou pravidel a axiomů byli schopni vygenerovat veškeré možné logické důsledky. Kdyby tomu tak nebylo, měli bychom sice korektní systém, který však neumí některé důsledky generovat/ověřovat, což je jako univerzální algoritmus opět nefunkční. Pokud je systém korektní a úplný, pak se chová zcela ekvivalentně sémantice a přesto ji nijak během dedukce nemusíme uvažovat.

Nyní rezoluci aplikujeme na již sémanticky řešený soudní případ.

Příklad 5: Nejprve je nutné formule 1. $A \vee B \vee C$, 2. $A \rightarrow (B \vee C)$, 3. $\neg C$ převést do klauzulí. Formule 1. a 3. jsou klauzulemi bez převodu. Formule 2. vyžaduje převod. Využít můžeme pravidla pro přepis implikace na disjunkci. Toto pravidlo lze schématicky zapsat jako $A \rightarrow B$

$\Leftrightarrow \neg A \vee B$ (čtenář si může lehce ověřit pomocí tabulky, že jde opravdu o formule s totožnou interpretací). Tímto pravidlem pak formuli 2. převedeme na formuli (disjunkce je navíc komutativní a asociativní): 2. $\neg A \vee B \vee C$. Nyní již můžeme buď přímo nebo ne-přímo ověřovat závěry. Nejprve se pokusíme pomocí rezolučního pravdila (rezoluce) přímo vygenerovat, že B je vinen (B). Navíc přitom použijeme pomocná pravidla (například vyskytuje-li v klauzuli atom vícenásobně, je klauzule s jedním výskytem ekvivalentní; tyto kroky označíme pomocí $\Rightarrow$). Dalším platným pravidlem je, že formule $\bot \vee F$ je ekvivalentní s F . Toto využijeme v případě, že jedna z premis obsahuje pouze jeden atom a tím pádem je po rezoluci ekvivaletní s $\bot$.

1. $A \vee B \vee C$ (axiom), 2. $\neg A \vee B \vee C$ (axiom), 3. $\neg C$ (axiom),

(rezoluce na A v 1. a 2.): $(B \vee C) \vee (B \vee C) \Rightarrow B \vee C$,

(rezoluce na C v 4. a 3. -z formule 3. nezbylo nic): B Tím jsme provedli důkaz toho, že B je vinen (jelikož systém založený na rezoluci je korektní a úplný).

Nyní se stejný závěr pokusíme dokázat nepřímo. Přitom přidáme k předpokladům negaci závěru a budeme se snažit prokázat nesplnitelnost (to znamená vygenerovat prázdnou klauzuli, která je nesplnitelná).

1. $A \vee B \vee C$ (axiom), 2. $\neg A \vee B \vee C$ (axiom), 3. $\neg C$ (axiom), 4. $\neg B$ (negace závěru),

(rezoluce na B v 4. a 2. -z formule 4. nezbylo nic): $\neg A \vee C$,

(rezoluce na B v 4. a 1. -z formule 4. nezbylo nic): $A \vee C$,

(rezoluce na A v 5. a 6.): $C \vee C \Rightarrow C$,

(rezoluce na C v 7. a 3. -z formule 7. ani 3. nezbylo nic): $\perp$ Jelikož jsme dospěli k prázdné formuli, prokázali jsme nesplnitelnost množiny formulí 1. 4., čímž je také prokázáno nepřímo, že B je logickým důsledkem množiny formulí 1. -3.

Nepřímý důkaz v předchozím příkladě jsme samozřejmě mohli realizovat různými způsoby. Univerzálním přístupem je konstrukce nepřímého důkazu pomocí přímého přidáním jediného kroku, kdy provedeme rezoluci na vygenerovaný přímý důsledek a jeho negaci
(pokud jde jen o atom). Na tom je vidět, že zřejmě existuje vždy mnoho způsobů, jak
důkaz provést. Z hlediska časové složitosti jde principiálně o úlohu s exponenciální složitostí, o kterých jsme se již zmínili v předchozím článku v MFI. Nicméně existují přístupy, jak důkaz urychlovat a těm se říká rezoluční strategie. Některé pomáhají málo, ale jsou v principu úplné (tedy zachovávají úplnost systému) a některé jsou velmi efektivní za cenu neúplnosti (ale ve většině praktických úloh to nevadí). V některých formálních systémech je rezoluce nazývána jinak, resp. je její obecná myšlenka skryta v jiné symbolice. Například se můžete setkat s tzv. klauzulární logikou, kde se rezoluční pravidlo skrývá v tzv. pravidle řezu. Řez je výstižným pojmenováním, neboť jsme viděli, že rezoluce vlastně ”vyřezává” atomy z původních formulí.

V praxi se rezoluční metoda uplatňuje především v logickém programování. Logické programování není tak rozšířeno jako procedurální

programování (např. algoritmizace v jazyce Pascal). Při procedurálním přístupu programátor musí vymyslet způsob, jak dosáhnout řešení cíle a to pomocí řízení výpočtu (musí správně použít proměnné, přiřazování, podmínky, cykly atd.). Logické programování vychází z myšlenky automatizace dedukce. Programátor nedefinuje postup řešení, ale pouze zadává formule (pravidla a fakta), která specifikují ”logiku” řešení úlohy. Na takto zapsaný program se pak může dotazovat, podobně jako při prověřování závěrů dedukce a systém sám odpoví na dotaz. Způsob řešení je univerzální a programátor se o něj nemusí starat. Jako jednoduchý příklad může sloužit výpočet faktoriálů. V procedurálním programování musí programátor vytvořit řízený výpočet, tj. musí naprogramovat cyklus nebo rekurzivní proceduru. V logickém programování stačí naprogramovat dvě pravidla (resp. jedno pravidlo a jeden fakt). Pravidlo udává hodnotu faktoriálu pro argument předchozího přirozeného čísla pomocí vazby na term s proměnnou (v logickém programování se nemá používat klasické přiřazování, měly by se používat výlučně prostředky logiky). Fakt udává hodnotu faktoriálu pro argument

0. V praxi by používání pouze logických prostředků způsobovalo neefektivní řešení, proto implementace logického programování často umožňují použití také některých omezených procedurálních prvků (např. řez). Asi nejznámějším prostředkem logického programování je jazyk PROLOG (PROgramming in LOGic). Vzhledem k omezenému rozsahu tohoto článku jej nebudeme rozebírat, ale čtenář má možnost využít elektronický učební text s mnohými příklady [2]. Pro vlastní pokusy doporučujeme získat z Internetu některou z freewarových implementací.

Logické programování je výhodné především u úloh, kdy hledáme řešení v rozsáhlém stavovém prostoru a v úlohách typických v umělé inteligenci.

11.4. Vícehodnotová logika

Pro modelování situací v reálném světě je klasická logika poměrně chudá. Přes všechny její přednosti je zásadním problémem především dvouhodnotová logická interpretace. Již dlouhou dobu existují formalismy zavádějící například logiku trojhodnotovou, kde třetí logická hodnota kromě pravda/nepravda je ”nevím”. Různé pokusy o zobecnění například pomocí pravděpodobnosti byly zastíněny v šedesátých letech 20.století tzv. fuzzy matematikou. Fuzzy matematika vychází z principu fuzzy množiny, která narozdíl od klasické množiny, která buď obsahuje nebo neobsahuje prvek, může prvek obsahovat na určitém stupni příslušnosti. Prvky tedy mohou být v množině buď zcela nebo vůbec, ale také ”jen trochu”. Čtenář již měl možnost získat základní informace o fuzzy množinách v článku [7]. Fuzzy logika pak tento princip využívá tím, že rovněž interpretace formule nemusí být jen pravda nebo nepravda, ale může to být pravda v určitém stupni.

I když myšlenka fuzzy logiky je velmi prostá, lehce pochopitelná a tudíž implementovatelná do různých automatizovaných systémů, je potřeba se při jejím používání držet některých vlastností. I v běžném životě se můžete setkat s aplikacemi fuzzy logiky. Například elektrospotřebiče -pračky - mají dnes často na sobě nápis ”Fuzzy-logic”. Tím se může myslet například schopnost dávkovat automaticky prací prostředky nikoliv v

přesně vymezených mantinelech podle váhy prádla (např. pro 1 kg prádla přidej přesně 10 g pracího prostředku), ale pouze vágními pravidly (např. je-li prádla málo, přidej pracího prostředku málo). Termínem fuzzy logika se označuje schopnost pracovat s neurčitou (vágní) informací, ale samozřejmě pojem fuzzy logika ve smyslu matematicko-informatickém je mnohem složitější. Základním problémem často bývá živelné používání množin stupňů příslušnosti bez ohledu na to, že musí existovat přímá souvislost také s používanými logickými spojkami. Proto je nezbytné, aby množina stupňů příslušnosti (pravdivosti) formule byla některou ze zavedených algeber. Množinou bývá v praktických aplikacích většinou interval [0, 1], i když teorie fuzzy množin a fuzzy logika má teoretické výsledky i pro mnohem složitější struktury. Tyto algebry mají vždy své výhody a nevýhody podle toho, jaké interpretace spojek na intervalu [0, 1] použijeme. Tyto spojky musí být ještě navzájem v souvislosti tj. použijeme-li určitou konjunkci předurčuje to již použití ostatních spojek. Nevýhodou algeber pak může být například nespojitost interpretačních funkcí spojek, neplatnost některých zásadních logických pravidel (zákonů) tak, jak je známe z klasické logiky. Pravděpodobně nejlepším kompromisem je tzv. Lukasiewiczova algebra pojmenovaná po významném polském logikovi. Tato algebra je následující struktura:

$$\mathbf{L}_{\mathbf{L}} = ([0, 1], \wedge, \vee, \neg, \rightarrow, 0, 1)$$

kde [0, 1] je interval reálných čísel mezi 0 a 1, což jsou nejmenší a největší hodnota (nepravda, pravda). $\wedge$ a $\vee$ jsou operace infima a suprema (resp.

na uvedené množině je lze ztotožnit s minimem a maximem). Definice standardních a odvozených operátorů je následující:

$a \otimes b = 0 \vee (a + b - 1)$ $a \rightarrow b = 1 \wedge (1 - a + b)$ $a \quad b = 1 \wedge (a + b)$ $\neg a = 1 - a$

Na základě této algebry bychom pak mohli vytvořit příslušnou fuzzy výrokovou nebo predikátovou logiku. Zavést bychom museli kromě standardních logických spojek konjunkce a disjunkce $\wedge$, $\vee$ (jejichž interpretační funkce by byly totožné s operacemi infima a suprema) také nové LLukasiewiczovy spojky a to LLukasiewiczova konjunkce (&) a disjunkce (\). Jejich interpretační funkce jsou operace a . Implikace a negace se připouští interpretovat pouze na základě operátorů této algebry.

Příklad 5: Pomocí fuzzy logik můžeme lehce vyřešit paradoxy, se kterými si klasická logika neumí poradit. Můžete se setkat s různými formulacemi, ale v zásadě jsou všechny totožné. Známý je tzv. paradox hromady. Jde o to, že bychom v klasické dvouhodnotové logice chtěli namodelovat situaci hromady, ke které přidáváme kameny. Víme, že hromada bez kamenů je rozhodně malá. Dále je rozumný předpoklad, pokud máme malou hromadu kamenů, pak hromada s přidaným jedním kamenem bude stále malá. V klasické logice, kde jsou formule pravdivé nebo nepravdivé, by pak každá hromada byla paradoxně malá. Můžeme totiž potenciálně nekonečnou sekvencí implikací vždy dokázat, že hromada s počtem kamenů o x větším je stále malá. Ve fuzzy logice můžeme díky pravdivosti s určitým stupněm příslušnosti namodelovat tuto situaci dvěmi formulemi (použijeme predikát malahromada(t), kde t je počet kamenů v hromadě):

$\text{malahromada}(x) \rightarrow \text{malahromada}(x + 1)$ -pravdivá ve stupni 0.999

malahromada(0) -pravdivá ve stupni 1 Formule 1. není narozdíl od klasické logiky pravdivá zcela a díky tomu nedojde k paradoxní dedukci. Díky omezené pravdivosti bude při dedukci s každou aplikací formule 1. při navyšení počtu kamenů o 1 klesat pravdivost vyvozeného predikátu malahromada o 0.001. Chtěli bychom například ověřit stupeň pravdivosti důsledku malahromada(1) z předpokladů 1. a 2. Platí-li formule 1. na 0.999 a formule 2. na 1, pak můžeme na základě definice operátoru → interpretace (I) implikace a platného vztahu:

I(malahromada(0))= 1 sestavit rovnici:

0.999 =1 ∧ (1 − 1+ I(malahromada(1))) ⇒ I(malahromada(1)) =0.999

Pro malahromada(2):

0.999 =1 ∧ (1 − 0.999 + I(malahromada(2))) = 0.001 + I(malahromada(2))

⇒ I(malahromada(2)) = 0.998 A tak dále pro zvětšující se počet kamenů, až pro malahromada(1000) bychom došli ke stupni pravdivosti 0.

Fuzzy logika umožňuje pracovat s neurčitou informací a je zobecněním klasické logiky. Klasická logika je vlastně speciální případ fuzzy logiky. Stejně jako mnoho jiných zobecnění, přináší i fuzzy logika řadu problémů, které klasická logika nemá. Například používání rezolučního pravidla je zde velmi omezeno, protože neexistuje univerzální postup převodu do formy klauzulí. Jde o problém aktuálně řešený například i na Ostravské Univerzitě (viz [1]). Čtenář s hlubším zájmem se může na počáteční výzkumy v této oblasti informovat v elektronicky dostupném článku (doporučujeme zejména příklady).

Zajímavé jsou také aplikace teorie fuzzy množin a fuzzy logiky, například existuje přístup s využitím tzv. lingvistických proměnných (viz [5]). Tyto proměnné mohou místo klasických číselných hodnot nabývat hodnot blízkých přirozenému jazyku jako jsou například lingvistické výrazy typu: ”velmi malý”, ”zhruba střední” atd. Pomocí těchto proměnných pak lze modelovat velmi srozumitelně a podobně jako člověk reálné situace. Například lze popisovat řízení auta, kde dvě z pravidel by mohla znít: KDYŽ na semaforu svítí jen žluté světlo A ZÁROVEŇ vzdálenost od křižovatky je malá A ZÁROVEŇ rychlost auta je malá PAK sešlápni brzdu střední silou KDYŽ na semaforu svítí jen žluté světlo A ZÁROVEŇ vzdálenost od křižovatky je velmi malá A ZÁROVEŇ rychlost auta je střední PAK sešlápni plyn velkou silou

Tato pravidla jsou pak interpretována pomocí fuzzy logiky a můžeme díky nim velmi lehce modelovat a zejména automatizovat postupy z mnoha oblastí života. Existují systémy, které se v praxi skutečně používaly a používají jako je systém LFLC (Linguistic Fuzzy Logic Controller) vyvinutý rovněž na Ostravské Univerzitě. Pomocí něj se modelovaly například technologické procesy v hutích a oproti klasických prostředkům jsou velkým vylepšením. Lze totiž na rozdíl od klasických regulačních technik, které vyžadují složitou matematiku jako jsou diferenciální rovnice a se kterými může pracovat jen velmi úzká skupina odborníků, tyto systémy svěřit i poučenému laikovi. Ten dobře zná například svůj technologický proces, který ručně obsluhoval dlouhou dobu a je schopen formulovat slovně své akční zásahy. Ty pak stačí naformulovat a

odzkoušet a máme ve velmi krátkém čase funkční automatizaci daného procesu, založenou na fuzzy logice.

Logika, problém dedukce a její automatizace se vyskytuje v mnoha oblastech života. Je nedílnou součástí matematiky a informatiky a proto by se výuka logiky měla odrazit nejen ve výuce matematiky na středních školách, ale právě díky problému automatizace dedukce by měla být alespoň v omezeném rozsahu i součástí informatické výuky. Výroková logika a systémy dedukce pro ni nejsou pro středoškolské studenty nedostupné, jak jsme se snažili ukázat na teorii i příkladech. S pomocí počítačových programů si student navíc může mnohem rychleji osvojit principy dokazování důsledků a to na populárních příkladech.

Logika, a to nejen vícehodnotová, ale i klasická, není v žádném případě ustrnulá disciplína. Pravý opak je pravdou a i jejich teorie se dynamicky vyvíjí právě v této době. Aplikace založené na vícehodnotové logice se dostávají již do civilního života. V současné době je velmi aktuální problém, jak lépe reprezentovat znalosti na Internetu. V současnosti zavedené možnosti reprezentace a vyhledávání informací jsou spíše syntaktického charakteru a postrádají tedy svůj smysl. Takzvaný projekt sémantického webu má za cíl dát jazyk a metody pro dedukci, které dokáží dát webovským stránkám i smysl -logiku. Jeden z nadějných pokusů je založen právě na tzv. Deskripční logice, která je v principu ”odlehčenou” verzí predikátové logiky. I proto tuto aktuálnost a perspektivitu si logika a dedukce zaslouží své místo ve výuce.

Literatura

[Ce92] ČEŠKA, Milan, RÁBOVÁ, Zdena. Gramatiky a jazyky. Brno, VUT 1992.

[Ch84] CHYTIL, Milan. Automaty a gramatiky. Praha, SNTL 1984.

[Ho79] HOPCROFT, J.E., ULLMAN, J. D. Introduction to Automata theory, Languages and Computation. Addison-Wesley, Reading (Mass.), 1979

[Ja97] JANČAR, Petr. Teorie jazyků a automatů. VŠB TU Ostrava, Dokument dostupný na URL: http://www.cs.vsb.cz/jancar/

[Ja97a] JANČAR, Petr. Vyčíslitelnost a složitost. VŠB TU Ostrava, Dokument dostupný na URL: http://www.cs.vsb.cz/jancar/

[Lu95] LUKASOVÁ, Alena. Logické základy umělé inteligence I. - výroková a predikátová logika. Ostravská univerzita, 1995 (také jako distanční opora 2002)

[Lu97] LUKASOVÁ, Alena. Logické základy umělé inteligence II. – formalizace a automatizace dedukce. Ostravská univerzita, 1997 (také jako distanční opora 2002)

[Pa02] PAVLISKA, Viktor: Vyčíslitelnost a složitost I. distanční studijní text OU, 2002

Další vhodná doplňková literatura:

U. Manber. Introduction to Algorithms, Addison-Wesley, 1989.

Aho, A., Hopcroft, J., Ullman, J. The design and analysis of computer algorithms, Addison-Wesley, 1974.

Jančar P. Teoretická informatika, VŠB Ostrava, 2010.

Černá, I. Úvod do teorie složitosti, FI MUNI, 1993.

O. Demuth, R. Kryl, and A. Kučera. Teorie algoritmů I, II. SPN, 1984.

M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, San Francisco, 1979.

Oded Goldreich. Computational Complexity: A Conceptual Perspective. Cambridge University Press, New York, NY, USA, 2008.