VYČÍSLITELNOST A SLOŽITOST

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 2020

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

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

Název: Vyčíslitelnost a složitost
Autor: Doc. RNDr. PaedDr. Hashim Habiballa, Ph.D.
Vydání: druhé, 2020
Počet stran: 159

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

OBSAH

1 Úvod

V této kapitole se dozvíte:

- Čím se zabývá vyčíslitelnost a složitost (teorie algoritmů).
- Jaké matematické formalismy se používají při práci s pojmy teorie.

Po jejím prostudování byste měli být schopni:

- Používat běžné algebraické prostředky formalizace.

Klíčová slova této kapitoly:

Vyčíslitelnost a složitost, teorie algoritmů.

Průvodce studiem

Studium této kapitoly je spíše opakováním vašich znalostí ze základních kurzů algebry. Abychom mohli studovat formální pojmy teoretické informatiky, je nutné, abyste s jistotou používali běžné definice jako je množina, relace, funkce apod.

Vyčíslitelnost a složitost problémů je základní znalostí informatika. Pro efektivní návrh a realizaci algoritmů je potřebné znát meze algoritmizace a také omezení vyplývající především jejich časové složitosti. Klíčovým je pak také problém P-NP, který limituje naši schopnost řešit efektivně algoritmicky mnoho zásadních problémů optimalizace pro praxi důležitých.

1.1 Množiny

Množinou rozumíme souhrn (konečný či nekonečný) jistých objektů. Objekt x z množiny M se nazývá prvkem množiny M, značíme $x \in M$. Opačný případ, tj. x není prvkem M, značíme $x \notin M$.

Speciální množinou je množina neobsahující žádný prvek, tzv. prázdná množina, označovaná $\emptyset$.

Jestliže každý prvek množiny M je zároveň prvkem množiny N, říkáme, že M je podmnožinou množiny N a značíme $M \subseteq N$. Pro libovolnou množinu M platí $M \subseteq M$ a rovněž $\emptyset \subseteq M$. Množiny M a N jsou shodné (M = N), jestliže $M \subseteq N$ a $N \subseteq M$; opačný vztah značíme $M \neq N$. O M říkáme, že je vlastní podmnožinou N, jestliže $M \subseteq N$ a $M \neq N$, značíme $M \subset N$.

Množiny zadáváme buď výčtem jejich prvků,

$\{x_1, x_2, \ldots, x_n\}$,

či pomocí podmínky (charakteristické funkce, predikátu), která vyčleňuje prvky definované množiny. Používaná notace je

{x | podmínka},

což čteme „množina všech x, pro které platí podmínka". Je-li potřeba vymezit možný obor hodnot pro proměnnou x v takové specifikaci na prvky nějaké množiny N, píšeme

$\{x \in N \mid \text{podmínka}\}$.

Základními operacemi s množinami jsou sjednocení $M \cup N$ definované jako

$M \cup N = \{x \mid x \in M \text{ nebo } x \in N\}$,

a průnik

$M \cap N = \{x \mid x \in M \text{ a zároveň } x \in N\}$.

Obě tyto operace jsou komutativní, tj. platí $M \cup N = N \cup M$ a $M \cap N = N \cap M$, a asociativní, tj. $(M \cup N) \cup P = M \cup (N \cup P)$ a také $(M \cap N) \cap P = M \cap (N \cap P)$. Tyto operace jsou také navzájem distributivní, tj. $M \cap (N \cup P) = (M \cap N) \cup (M \cap P)$ a rovněž $M \cup (N \cap P) = (M \cup N) \cap (M \cup P)$. Množiny M, N, pro které $M \cap N = \emptyset$, se nazývají disjunktní množiny.

Potenční množina

Potenční množina množiny M je množina obsahující všechny podmnožiny množiny M. Budeme ji značit: P(M).

Příklad:
Například pro konečnou množinu M = {a, b} dostaneme

$P(M) = \{\{\emptyset\}, \{a\}, \{b\}, \{a, b\}\}$

Z objektů $a_1, \ldots, a_n$ můžeme utvořit uspořádanou n-tici (zkráceně jen n-tici)

$(a_1, \ldots, a_n)$

jako objekt sestávající z $a_1, \ldots, a_n$ přesně ve stanoveném pořadí (s možným vícenásobným výskytem kteréhokoli z nich).

Kartézským součinem množin $M_1, \ldots, M_n$ rozumíme množinu

$M_1 \times M_2 \times \bullet\bullet\bullet \times M_n = \{(x_1, \ldots, x_n) \mid x_1 \in M_1, \ldots, x_n \in M_n\}$.

Pro kartézský součin množiny samé se sebou, např. $M \times M$, používáme notace $M^2 = M \times M$, $M^3 = M \times M \times M$, atd., obecně $M^n = M \times M^{n-1}$.

1.2 Abeceda, slovo, jazyk

Jelikož budeme často pracovat s funkcemi, které budou mít jako parametry množiny slov, bude dobré, když si předem ujasníme, co budeme pod těmito pojmy myslet.

Definice: (Abeceda, slovo, jazyk)

1. Abecedou budeme nazývat libovolnou konečnou množinu symbolů. Nejčastěji budeme pro označení abecedy používat symbol Σ.

2. Konečnou posloupnost symbolů nad abecedou Σ budeme nazývat slovem. Budeme uvažovat i prázdné slovo, které budeme značit symbolem ε.

3. Délku slova w budeme psát jako |w| a znamená počet symbolů, ze kterých je dané slovo utvořeno. Délka prázdného slova |ε| = 0.

4. Symbol Σ^n budeme používat pro označení množiny všech slov délky n nad abecedou Σ. Množinu všech slov nad abecedou Σ označíme Σ^*. Poznamenejme ještě, že do Σ^* patří rovněž ε!

5. Jazykem L nad abecedou Σ budeme rozumět libovolnou množinu slov vytvořených ze symbolů z Σ, jinými slovy libovolnou podmnožinu Σ^*. Budeme tedy psát $L \subseteq \Sigma^*$ Vzhledem k zavedenému pojmu potenční množina, můžeme rovněž psát, že každý jazyk $L \in P(\Sigma^*)$.

Příklad:

Jako abecedu si v tomto příkladu můžeme zvolit množinu: Σ = {a, b, c}. Z této abecedy můžeme skládat různá slova.

Například aab, bc, c jsou tři slova vytvořená z dané abecedy. Σ^2 představuje množinu: {aa, ab, ac, ba, bb, bc, ca, cb, cc}

Pro jazyk $L \subseteq \Sigma^*$ nad abecedou Σ tvořený slovy délky 2 a 3 můžeme použít

zápis: $L = \Sigma^2 \cup \Sigma^3$

1.3 Uspořádání a kódování slov

Na množině slov můžeme definovat uspořádání. Předpokládejme, že je dáno uspořádání prvků abecedy Σ.

Definice: Lexikografické uspořádání

V lexikografickém uspořádání prvků ze Σ^* slovo α předchází slovo β, když buď α je prefixem (částečným úsekem) β, anebo první písmeno zleva, které
je rozdílné v těchto slovech, je menší v α.

Někdy je výhodnější používat jiný typ uspořádání, tzv. rostoucí uspořádání.

Definice: Rostoucí uspořádání

Při tomto uspořádání slovo kratší délky předchází slovo větší délky a stejně dlouhá slova jsou uspořádána lexikograficky. Další důležitou vlastnost, kterou má rostoucí uspořádání je, že pro libovolnou abecedu Σ určuje bijekci mezi množinami N a $\Sigma*$ (N označuje množinu všech celých nezáporných čísel). Jinými slovy nám umožňuje ke každému slovu přiřadit číslo, něco jako index slova vzhledem k danému uspořádání, respektive jeho číselný kód. A na druhou stranu k danému přirozenému číslu umíme tímto způsobem najít odpovídající slovo.

Příklad:
Vezměme jako abecedu množinu cifer, tedy $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ s tím, že $0 < 1 < \bullet \bullet \bullet < 9$. Podle právě definovaného lexikografického uspořá-
dání například platí, že 379313 předchází 387 a to předchází 38782.

1.4 Relace

Relací nad množinami $M_1, \ldots, M_n$ (n-ární relací pro n zúčastněných množin) rozumíme libovolnou podmnožinu R kartézského součinu $M_1 \times \bullet \bullet \bullet \times M_n$, tj.

$R \subseteq M_1 \times \bullet \bullet \bullet \times M_n$.

Pro n-tici $(x_1, \ldots, x_n) \in R$ říkáme, že relace R platí (je splněna) pro
$x_1, \ldots, x_n$.

Pro binární relace často používáme infixové notace xRy, např. $x < y$, namísto $(x, y) \in R$, tj. $(x, y) \in <$.

Je-li R binární relace nad množinami A, B, pak inverzní relací k R rozumíme relaci

$R^{-1} = \{(y, x) \mid (x, y) \in R\}$.

Vlastnosti relací:

Relace $R \subseteq M \times M$ se nazývá

- **reflexivní**, jestliže xRx pro každé $x \in R$
- **symetrická**, jestliže kdykoli xRy, je i yRx
- **tranzitivní**, jestliže kdykoli xRy a yRz, je i xRz

- **antireflexivní**, jestliže pro žádné $x \in M$ neplatí xRx
- **antisymetrická**, jestliže pro žádné $x, y \in M$, $x \neq y$, neplatí zároveň xRy i yRx

Relace ekvivalence

Reflexivita, symetrie a tranzitivita jsou navzájem nezávislé; rovněž tak anti- reflexivita s antisymetrií.
Binární relace $R \subseteq M \times M$, která je reflexivní, symetrická a tranzitivní, se nazývá relace ekvivalence na M. Pro libovolnou relaci ekvivalence R na M definujeme třídu ekvivalence prvku $x \in M$ jako množinu $[x]_R$ definovanou jako:

$$[x]_R = \{y \mid xRy\}.$$

Pro libovolnou třídu ekvivalence platí buď $[x]_R = [y]_R$ (a to pro xRy) nebo $[x]_R \cap [y]_R = \emptyset$. Říkáme, že relace ekvivalence na dané množině M definuje
rozklad na (vzájemně disjunktní) podmnožiny množiny M (jejichž sjednocení je rovno celé M) – a naopak každému rozkladu odpovídá jednoznačně jistá
relace ekvivalence. Množinu všech tříd ekvivalence značíme M/R, tj. třídy ekvivalence $M/R = \{[x]_R \mid x \in M\}$.

1.5 Funkce

Definice: Funkce

Funkce f (či zobrazení f) s definičním oborem D a oborem hodnot H, značíme f : D → H, je relace f ⊆ D × H taková, že pro libovolné d ∈ D existuje právě jedno y ∈ H takové, že x f y. Tento prvek y značíme f (x) či f x. Parciální
funkce f : D → H je relace na D × H, pro kterou ke každému x ∈ D existuje nejvýše jedna hodnota f (x) ∈ H. (Funkce všude definované někdy explicitně
označujeme jako totální funkce.) Je-li M ⊆ D podmnožina definičního oboru
funkce f : D → H, pak f (M) označuje obraz této množiny vzhledem k funkci
f , tj. množinu f (M) = {f (x) | x ∈ M }.

Funkce f : D → H, pro kterou pro libovolné x, y ∈ D platí f (x) = f (y), právě když x = y, se nazývá jedno-jednoznačnou, zapisujeme též 1 : 1.
Totální funkce f : D → H, která je 1 : 1 a pro kterou platí f (D) = H, se nazývá bijekcí (mezi množinami D a H). Je-li f : D → H bijekce, pak inverzní bijekce relace f^{-1} je rovněž funkcí, tj. f^{-1} : H → D

Jsou-li f : D → G a g : G → H funkce, pak kompozicí g ◦ f rozumíme funkci z D do H takovou, že pro libovolné x ∈ D platí

$g \circ f(x) = g(f(x))$.

Je-li $f : D \to H$ bijekce, pak $f^{-1} \circ f = idD$ (kde $idD : D \to D$ je identická funkce (identita) na D – tj. funkce, pro kterou pro libovolné $x \in D$ platí $idD(x) = x$) a $f \circ f - 1 = idH$.

1.6 Induktivní definice

Častým případem definic v dalším textu budou induktivní definice množin objektů. Typická induktivní definice množiny $M \subseteq U$ je dána množinou $A \subseteq U$, funkcí $f : U \to U$ a schématem:

Množina M je definována jako množina splňující:

1. $A \subseteq M$
2. jestliže $x \in M$, pak $f(x) \in M$
3. nic jiného než to, co vyplývá z bodů 1 a 2 není prvkem M .

Na takovou definici lze pohlížet jako na definici posloupnosti množin
$M_0 = A$,
$M_1 = M_0 \cup f(x) \mid x \in M_0$, $M_2 = M_1 \cup f(x) \mid x \in M_1$,

.

$M_{n+1} = M_n \cup f(x) \mid x \in M_n$.

tj. sjednocení všech takto definovaných množin M_0, M_1, ... Snadnou indukcí lze dokázat, že pro danou množinu $A \subseteq U$ a funkci $f : U \rightarrow U$ je tato M

právě množinou, pro kterou platí:

1. $A \subseteq M$
2. M je uzavřená vzhledem k f , tj. kdykoli $x \in M$, platí i $f(x) \in M$
3. pro libovolnou množinu $S \subseteq U$ splňující body 1 a 2 platí $M \subseteq S$ – tj.

M je nejmenší množinou obsahující A a uzavřenou vzhledem k f .

Nejdůležitější probrané pojmy:

- množinové a jazykové operace
- potenční množina
- relace, funkce, ekvivalence a rozklad
- relace rozkladu
- induktivní definice

2 Vyčíslitelnost a problém

Cíl:

After reading this section and its parts, you should be able to:
- explain what computability theories are doing,
- characterize different types of problems,
- to distinguish between different types of problems,
- to list some undecidable problems,

Průvodce studiem

Nejdříve se budeme věnovat oblasti vyčíslitelnosti. Jedná se o rozsáhlý celek, a proto bude rozdělen do několika části, které na sebe plynule navazují a úzce spolu souvisejí.

Základní otázka, na kterou budeme v této části hledat odpověď, zní:

Co všechno je a co není algoritmicky vyčíslitelné (řešitelné)?

Ke kterým problémům existují algoritmy, jež je řeší a ke kterým problémům takové algoritmy neexistují ?

Vyčíslitelnost a problém

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 problému je lepší (např. rychlejší)?
Lze na takovéto otázky vůbec odpovědět exaktně nebo jen 'hypoteticky'?

2.1 Problémy

V předchozím textu jsme již zmínili, co je a co není algoritmicky řešitelné. Abychom takovou otázku mohli rozumně zodpovědět, je mj. třeba podrobněji specifikovat, co rozumíme pod slovem problém. Význam tohoto slova v přirozeném jazyce je velmi široký (může se jednat jak o problém řešení problémů kvadratické rovnice, tak např. o problém „Co je největší otázka života“ apod.). Zde se pochopitelně soustředíme na problémy, jejichž podstata umožňuje vůbec uvažovat o potenciálním nasazení algoritmických metod, potažmo počítačových programů. (Proto je nám zde bližší spíše problém kvadratické rovnice než druhý zmíněný problém.)
Je rozumné se shodnout na tom, že každý uvažovaný problém lze popsat následujícím schématem:

Problém XY (označení, či výstižný název problému)

Instance: zde je popsáno, co může být zadáním (vstupem) u našeho problému (např. libovolná kvadratická rovnice)

Výsledek: zde je popsáno, jak vypadají výsledky (výstupy) a jak se žádaný výsledek má vztahovat k zadané instanci (např. to mají být kořeny zadané rovnice)

Budeme předpokládat, že výsledek je jednoznačně určen zadanou instancí (nebudeme tedy připouštět výsledek typu ‘nějaký z kořenů dané rovnice’ apod.).

Dále budeme (celkem přirozeně) předpokládat, že jakoukoli instanci problému, stejně jako jakýkoli výsledek, lze zadat řetězcem (posloupností symbolů) v nějaké konečné abecedě (obsahující např. písmena, číslice, interpunkční znaménka a případné další symboly).

Všimněme si, že v tomto smyslu není uvedený problém kvadratické rovnice ještě jednoznačně specifikován (jak zadáme konečným řetězcem libovolné reálné číslo?). Pokud se např. omezíme na racionální koeficienty a kořeny budeme uvažovat zaokrouhlené na určitý počet desetinných míst, nabízí se už běžný popis koeficientů a kořenů v desítkové soustavě.

Pozastavme se ještě nad problémem abecedy – množiny symbolů, pomocí nichž jsou popsány instance a výsledky. Je možné např. ke každému problému uvažovat jeho specifickou abecedu. Je ale také možné vzít jednu (univerzální) abecedu a v ní popisovat instance a výsledky kteréhokoli problému. Např. je možné se omezit na znaky ASCII. Od této představy je už jen krůček k uvě-domnění si, že jako ona univerzální může vlastně stačit dvouprvková abeceda
{0, 1}! Zápisy v ní sice pro nás asi nebudou příjemně čitelné, nicméně jistě tu-
šíte, jak zkonstruovat jednoduché algoritmy (prográmky) převádějící řetězce (texty) z naší oblíbené abecedy do abecedy {0, 1} a zpět – pro studované otázky algoritmické rozhodnutelnosti tedy nic neztratíme, omezíme-li se na onu dvouprvkovou abecedu.

Na problém ve výše uvedeném smyslu se lze dívat jako na zobrazení typu $\Sigma^* \to \Sigma^*$ pro danou abecedu Σ (ve světle předešlého odstavce si zde vždy lze za Σ dosadit abecedu {0, 1}), které každé (povolené) instanci problému (resp. jejímu popisu v dané abecedě) přiřazuje žádaný výsledek (resp. jeho popis v dané abecedě). K řetězci ze Σ^*, který nepředstavuje (povolenou) instanci problému, je možno přiřadit nějaký speciální výsledek (znamenající ‘Nekorektní zadání’). Je možné ale tohle neudělat a pro nepovolené instance prostě výsledek nedefinovat. Příslušné zobrazení $\Sigma^* \to \Sigma^*$ se pak chápe jako
částečné (tj. nedefinované pro všechny řetězce ze Σ^* – na rozdíl od obvyklého
totálního zobrazení definovaného všude).

Jako příklad problému, který realizuje částečné zobrazení, může posloužit:

Problém SQRT (Druhá odmocnina)

Instance: Celé číslo zadané ve dvojkové soustavě, přičemž záporná čísla budeme zapisovat tak, že jako první znak dáme 0 a pak binární tvar absolutní hodnoty čísla. Tedy např. 0101 představuje číslo −5

Výsledek: Celá část z druhé odmocniny zadaného čísla

Speciálními problémy budou problémy typu ANO/NE, kde žádaný výsledek (příslušný k povolené instanci) je prvek dvouprvkové množiny (např.
{ANO, NE}, či {1, 0}). Takový problém se nejčastěji zadává následujícím schématem.

Problém XY (označení, či výstižný název problému)

Instance: zde je popsáno, co může být zadáním (vstupem) u našeho problému (např. libovolná kvadratická rovnice)

Otázka: zde je otázka (vztahující se k instanci), na níž je vždy odpověď ANO nebo NE (např. existuje alespoň jeden reálný kořen zadané rovnice?)

Problém tohoto typu se často ztotožňuje s množinou právě těch řetězců v dané abecedě, které popisují ty instance problému, jimž je přiřazena odpověď ANO. Jedná se tedy vlastně o určitý jazyk v abecedě Σ. Na druhou stranu lze u každého jazyka uvažovat o problému příslušnosti zadaného slova k jazyku. Tedy o problému, zda slovo do jazyka patří, či nepatří.

2.2 Vlastnosti problémů

Nyní, po upřesnění toho, co rozumíme pod pojmem ‘problémy’, vymezíme jejich vlastnosti, které nás zde zajímají.

Definice (Algoritmická řešitelnost)

O daném problému řekneme, že je algoritmicky řešitelný (neboli odpovídající (částečné) zobrazení $\Sigma^* \rightarrow \Sigma^*$ je algoritmicky vyčíslitelné), jestliže existuje
algoritmus, který je schopen jako vstup přijmout libovolnou instanci daného problému a jeho výpočet pro libovolný takový vstup vždy skončí, přičemž výstupem bude požadovaný výsledek.

Je-li problém chápán jako výše zmíněné částečné zobrazení $\Sigma^* \rightarrow \Sigma^*$, pak se z technických důvodů obvykle požaduje, že příslušný algoritmus se pro

nekorektní zadání (tj. pro řetězec, jenž nepopisuje instanci problému) nezastaví (a jeho výstup není v tomto případě definován).
My se budeme hlavně zajímat o problémy typu ANO/NE, se kterými se snadněji pracuje a na něž se úvahy o obecných problémech dají převést. Pojem algoritmické řešitelnosti pro ně vyjadřujeme následovně:

Definice (Algoritmická rozhodnutelnost)

O daném problému typu ANO/NE řekneme, že je algoritmicky rozhodnutelný, pro stručnost rozhodnutelný, jestliže existuje algoritmus, který je schopen jako vstup přijmout libovolnou instanci daného problému a jeho výpočet pro libovolný takový vstup vždy skončí, přičemž výstupem bude požadovaná odpověď ANO či NE.

Celkem přirozeně lze tuto definice přenést na jazyky, tedy množiny řetězců v dané abecedě:

Definice: Jazyk L v abecedě Σ je rozhodnutelný, tj. množina $L \subseteq \Sigma^*$ je rozhodnutelná, jestliže problém příslušnosti k L je rozhodnutelný (Instance:
$w \in \Sigma^*$; Otázka: Platí $w \in L$?).

Budeme předpokládat, že pro libovolný uvažovaný problém P s ‘popisnou’ abecedou Σ je následující ANO/NE problém daný schématem:

Instance:

$w \in \Sigma*$;

Otázka: Je w (korektním) popisem instance problému P?

Pak je zřejmé, že ANO/NE problém je rozhodnutelný právě tehdy, když jemu příslušný jazyk (sestávající ze všech instancí na které je odpověď ANO) je rozhodnutelný (tzn. v tomto smyslu nedefinovanost problému pro případné nekorektní instance nehraje roli).

Bude se nám hodit i související pojem částečné rozhodnutelnosti:

Definice 1.4 (Částečná rozhodnutelnost)

O daném problému typu ANO/NE řekneme, že je částečně rozhodnutelný (neboli příslušný jazyk v abecedě Σ je částečně rozhodnutelný, tj. daná podmnožina množiny $\Sigma*$ je částečně rozhodnutelná), jestliže existuje algoritmus, jehož výpočet skončí právě pro takové vstupy, které odpovídají instancím daného problému s odpovědí ANO.

Úkoly k zamyšlení:

Každé tvrzení týkající se (částečné) rozhodnutelnosti problémů lze přeformu- lovat pro jazyky, tedy množiny a naopak. Tuto důležitou věc je potřeba mít neustále na paměti (důkladně uvědomit si to můžete např. na následujících tvrzeních, která jsou explicitně formulována jen pro množiny).

Všimněme si následujících vztahů mezi uvedenými pojmy.

Postova věta

Tvrzení: Je-li množina $A \subseteq \Sigma*$ rozhodnutelná, pak je i částečně rozhodnutelná.

Tvrzení: Množina $A \subseteq \Sigma*$ je rozhodnutelná právě když $\neg A$ (doplněk A, tj. $\Sigma^* \setminus A$) je rozhodnutelná.

Věta (Post): Množina $A \subseteq \Sigma*$ je rozhodnutelná právě když A i $\neg A$ jsou částečně rozhodnutelné.

Důkaz obou tvrzení a tím i důkaz jednoho směru Postovy věty, by měl být triviální, neboť rozhodnutelnost automaticky znamená i částečnou rozhodnutelnost. Pro opačný směr Postovy věty si stačí uvědomit, že potřebný algoritmus (rozhodující A) prostě ‘paralelně spustí’ (tj. např. střídavě krok po kroku simuluje) dva příslušné algoritmy (částečně rozhodující A a $\neg A$), ‘počká’, až jeden z nich skončí, a poté vydá příslušnou odpověď.

Nyní už má pro nás základní formulovaná otázka podstatně konkrétnější smysl. Opírá se ovšem stále o intuitivní pojem ‘algoritmus’, který je pro naše úvahy nutné formalizovat, zpřesnit.

Všimněte si, že Postovu větu jsme prokázali bez dalšího zpřesnění pojmu ‘algoritmus’. Opírali jsme se zde konkrétně například o to, že ke každým dvěma algoritmům lze navrhnout třetí algoritmus, který ty dva ‘paralelně’ (či ‘střídavě sekvenčně’) provádí – to jsme vzali jako intuitivně zřejmý fakt. Ve skutečnosti se už dostáváme na trochu nebezpečnou půdu a měli bychom si začít uvědomovat, proč asi je zpřesnění pojmu ‘algoritmus’ potřebné.

Shrnutí:

- V této kapitole jsme si ukázali, že existuje celá řada problémů. Některé z nich má smysl řešit pomocí počítačů, u některých je algoritmické řešení zjevně nemyslitelné. Všechny problémy, o kterých budeme v následujících kapitolách uvažovat se dají formulovat v jednotném tvaru (schématu). Stačí se dohodnout na formulaci zadání a specifikovat tvar výstupů. Jako abecedu, ve které budeme zapisovat instance problémů, si můžeme zvolit univerzální abecedu tvořenou pouze symboly $\{0, 1\}$.
- Na všechny problémy lze nahlížet jako na zobrazení tvaru $\Sigma* \rightarrow \Sigma*$,

které ke každé instanci problému přiřazuje její řešení. Toto zobrazení může být buď totální anebo pouze částečné, pokud existují instance daného problému, pro nějž není výstup definován.

- Speciálními pro nás budou problémy typu ANO/NE, u kterých je otázka formulována tak, aby se na ni dalo jednoznačně odpovědět ANO či NE. Ke každému problému takovéhoto typu se váže jazyk, který se- stává z těch instancí problému, na něž je odpověď ANO. Naopak lze na každý jazyk nahlížet ve světle problému příslušnosti slov do daného jazyka.
- Problém je algoritmicky řešitelný, pokud existuje algoritmus, který ke každému korektnímu zadání problému vždy v konečném čase jednoznačně určí požadovaný výsledek. Podobně je pro problémy typu ANO/NE zaveden pojem algoritmické rozhodnutelnosti. Rozhodnutelné jazyky a množiny jsou právě ty, pro které je problém příslušnosti k danému jazyku či množině rozhodnutelný.
- Problém je částečně rozhodnutelný, pokud známe algoritmus, který se zastaví a vydá výsledek, pouze pokud je odpověď na danou otázku ANO. Pokud je správná odpověď NE, tak se ji nikdy nedozvíme, protože příslušný algoritmus se nikdy nezastaví.

Kontrolní otázky:

1. Charakterizujte typy problémů, o kterých má smysl hovořit v souvislosti s algoritmickým nebo případně počítačovým nasazením.

2. Popište schéma zápisu obecného problému a uveďte alespoň tři konkrétní příklady z praxe.
3. Vysvětlete, co rozumíme pod pojmy formát vstupu a výstupu.

4. Objasněte, jak lze na problém nahlížet z pohledu funkčního zobrazení

5. V čem jsou specifické problémy typu ANO/NE?

6. Jak spolu souvisí jazyky a problémy?

7. Uveďte dva možné přístupy k vypořádání se s nekorektním zadáním.

8. Vysvětlete, v čem jsou odlišné pojmy ‘algoritmická řešitelnost’ a ‘algo- ritmická rozhodnutelnost’.

9. Uveďte příklad rozhodnutelného jazyka.

10. Kdy je jazyk částečně rozhodnutelný?

11. Přeformulujte Postovu větu pro jazyky.

Pojmy k zapamatování:

- abeceda vstupu a výstupu
- univerzální abeceda
- částečné a totální zobrazení

- problém příslušnosti slova k jazyku
- algoritmická řešitelnost
- algoritmická rozhodnutelnost
- rozhodnutelné jazyky a množiny
- částečná rozhodnutelnost

3 Turingovy stroje

Cíl:

V této kapitole se budeme zabývat formalizací pojmu algoritmus, kterou pro nás bude představovat Turingův stroj. Po úspěšném absolvování této kapitoly budete:

- znát základní vlastnosti Turingových strojů,
- vědět v čem spočívá hlavní rozdíl Turingových strojů oproti konečným a zásobníkovým automatům,
- umět formálně definovat Turingův stroj,
- vědět, jak Turingovy stroje realizují zobrazení a přijímají jazyky,
- vědět, co jsou to rekurzivní a rekurzivně spočetné jazyky,
- umět navrhovat své vlastní Turingovy stroje,
- schopni zakódovat libovolný Turingův stroj do abecedy {0, 1},
- umět navrhovat a používat některé modifikace Turingova stroje,
- schopni vymezit výpočetní sílu Turingových strojů spolu s jejich modifikacemi,
- chápat souvislost mezi Turingovy stroji a algoritmickými postupy,
- umět navrhnout univerzální Turingův stroj,
- schopni určit třídu jazyků rozpoznatelných Turingovými stroji a správně ji zařadit do Chomského hierarchie generativních jazyků.

Po prostudování tohoto oddílu a jeho částí byste měli být schopni:

- vysvětlit čím se zabývá teorie vyčíslitelnosti,

- charakterizovat různé typy problémů,
- vysvětlit co to znamená, že daný problém je rozhodnutelný, částečně rozhodnutelný nebo nerozhodnutelný.
- rozlišovat mezi různými typy problémů,
- jak formálně zapisovat problémy, jimiž se budeme ve zbývající části textu zabývat,
- rozlišovat mezi problémy vyčíslitelnými a nevyčíslitelnými,
- vysvětlit pojem algoritmické nerozhodnutelnosti,
- vyjmenovat některé nerozhodnutelné problémy,
- objasnit a dokázat proč některé problémy nelze algoritmicky řešit.

Průvodce studiem

Nyní se zaměříme na nejznámější a nejjednodušší výpočetní model – Turingův stroj. Měli byste jednak dbát na pochopení principů a jednak na formální vyjádření probíraných pojmů.

Dále byste měli věnovat pozornost také konstrukci konkrétních příkladů.

O formalizaci poměrně vágního pojmu algoritmu se pokusil v 30. letech mj. anglický matematik Alan M. Turing, pro jehož formalizaci pojmu algoritmus se brzy vžil název ‘Turingův stroj’. Turingovy stroje jsou podobné konečným automatům v tom, že jsou tvořeny řídícím mechanismem a vstupním tokem (informacemi), který si představujeme

jako pásku nekonečné délky. Rozdíl je v tom, že Turingovy stroje mohou pohybovat svými čtecími/zápisovými hlavami vpřed i vzad a mohou na pásku zapisovat i číst z ní. Ukážeme, že tyto vlastnosti výrazně zvýší mocnost strojů.

3.1 Základní vlastnosti Turingových strojů

Řídící mechanismus může být v daném okamžiku v libovolném stavu z konečného počtu stavů. Jeden z těchto stavů se nazývá počáteční a reprezentuje stav, v němž výpočet začíná. Jiný stav je označen jako koncový; jakmile stroj přejde do tohoto stavu, výpočetní proces končí. (Touto vlastností se liší Turingovy stroje od konečných a zásobníkových automatů, které mohou pokračovat v činnosti i po dosažení koncového stavu). Počáteční stav tedy ne- může být zároveň koncovým stavem a každý Turingův stroj musí mít alespoň dva stavy.

Důležitějším rozdílem mezi Turingovým strojem a automaty je v tom, že Turingovy stroje mohou ze vstupní pásky číst i zapisovat na ni. Páska je zleva ukončena, na pravé straně však pokračuje do nekonečna. Turingův stroj ji může využívat jako paměť stejným způsobem, jakým používá zásobníkový automat zásobník, ale není omezen operacemi pop a push při přístupu k uloženým datům. Může přeskočit část dat na své pásce, přičemž některá z nich modifikuje a jiná nechává nezměněna.

Používá-li Turingův stroj pásku jako paměť, je vhodné, aby používal speciální značky k označení různých částí pásky. Proto se zavádějí znaky, které nepatří do vstupní abecedy. Budeme tedy rozlišovat mezi konečnou množinou symbolů, zvanou vstupní abeceda, kterými jsou zapisována vstupní data a potenciálně větší (konečnou) množinou symbolů, zvanou pásková abeceda, kterou stroj umí číst a zapisovat.

Zvláštním symbolem této abecedy je blank, který představuje prázdné políčko na pásce. Dále v textu bude blank označován symbolem #. Předpokládáme, že symboly # jsou uloženy na pásce, pokud na daném místě nebylo zapsáno něco jiného. Např. pokud bude Turingův stroj pokračovat ve čtení pásky za vstupním řetězcem, bude dále rozpoznávat na pásce symboly #. Mazání pásky se provádí zápisem symbolu # na pásku. Často bývá symbol # prvkem vstupní abecedy; my ho do ní však zahrnovat nebudeme.

Jednotlivé akce prováděné Turingovým strojem spočívají v operacích zápisu nebo posuvu. Operace zápisu je dána nahrazením symbolu na pásce jiným symbolem a přechodem do nového stavu (který může být stejný jako původní stav). Operace posuvu je dána přesunutím hlavy o jedno místo doprava nebo doleva a přechodem do nového stavu (který opět může zůstat stejný jako předešlý stav). Volba akce, která bude provedena, závisí na aktuálním symbolu, který je na místě právě pod čtecí hlavou a na současném stavu řídícího mechanismu stroje. Akce zápisu i posuvu lze

provádět zároveň v jednom kroku stroje. Tedy např. lze přepsat aktuální čtený symbol a přitom posunout hlavu doprava.

Způsob, kterým se má změnit stav, přepsat obsah políček a posunout hlava, udává přechodová funkce δ. Turingův stroj pracuje tak, že opakovaně provádí přechody, dokud nedosáhne koncového stavu. Za určitých podmínek výpočet nikdy neskončí, protože program ‘uvázl’ v nekonečném cyklu.

3.2 Formální definice Turingova stroje

Definice (Turingův stroj)

Formálně lze psát, že Turingův stroj je šestice $(Q, \Sigma, \Gamma, \delta, q_0, F)$, kde

Q je konečná množina stavů,

Σ je konečná množina symbolů, která neobsahuje #, zvaná vstupní abeceda stroje,

Γ je konečná množina symbolů obsahující množinu Σ, zvaná množina páskových symbolů,

δ je přechodová funkce tvaru $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, N\}$, je to tedy zobrazení, které určuje chování stroje. V závislosti na stavu, ve kterém se stroj nachází a na čteném symbolu toto zobrazení určuje, do kterého nového stavu má stroj přejít, jaký symbol má zapsat na místo původního a kam má posunout čtecí hlavu – L znamená vlevo, R vpravo a N je pro určení toho, že má zůstat na místě.

q_0 je počáteční stav a

F je množina koncových stavů.

Definice (Konfigurace Turingova stroje)

Konfigurace Turingova stroje je libovolné slovo uqv, kde $q \in Q$, $u, v \in \Sigma*$. Počáteční konfigurace bude q_0w, kde $w \in (\Sigma \setminus \{\#\})*$

Pojem konfigurace Turingova stroje se nám bude velmi často hodit, proto je nezbytně nutné správně pochopit, co tento pojem znamená. Jedná se v pod- statě o zakódování aktuálního nastavení stroje během výpočtu, které lze plně popsat obsahem pásky, pozicí hlavy na pásce a vnitřním stavem stroje.

Příklad

Například konfigurace stroje abq_0 caab znamená, že stroj se nachází ve vnitřním stavu q_0, na pásce má uloženo slovo abcaab a hlavu má umístěnu nad třetím symbolem zleva, což v tomto případě je symbol c.

Definice (Bezprostřední následování)

Pro konfigurace lze zavést relaci bezprostředního následování `.

Konfigurace K_2 bezprostředně následuje konfiguraci K_1, což zapisujeme K_1 ` K_2, jestliže pro nějaké $q, q_l \in Q$, $x, y, z \in \Sigma$, $u, v \in \Sigma^*$ platí:

$K_1 = uxqyv$ a dále platí jedna z následujících možností:

1. $(K_2 = uq_lxzv) \wedge (\delta(q, y) = (q_l, z, L))$
2. $(K_2 = uxq_lzv) \wedge (\delta(q, y) = (q_l, z, N))$
3. $(K_2 = uxzq_lv) \wedge (\delta(q, y) = (q_l, z, R))$

Definice (Následování)

Relace $\vdash^*$ označuje reflexivní a tranzitivní uzávěr relace $\vdash$ ($K_1 \vdash^* K_2$ znamená, že K_2 lze dostat konečně mnoha kroky z K_1, když napíšeme $K_1 \vdash^n K_2$, budeme tím mít na mysli, že se z konfigurace K_1 lze dostat do K_2 přesně po n-krocích).

Definice (Koncová konfigurace)

Koncová konfigurace je libovolná konfigurace uq w, kde $q \in F$. Z technických
důvodů budeme předpokládat, že v koncové konfiguraci tvoří neprázdné znaky na pásce vždy souvislý úsek.

Definice (Zastavení Turingova stroje)

Řekneme, že Turingův stroj M se zastaví na $w \in \Sigma*$, značíme !M (w), jestliže

$q_0w \vdash^* K$, kde K je koncová konfigurace. Jinými slovy pokud existuje výpočet Turingova stroje nad slovem w, který po konečném počtu kroků dospěje z počáteční konfigurace do nějaké koncové konfigurace.

Uveďme následující přirozené definice (využívající předchozí úmluvu).

Definice

Turingův stroj M s abecedou Σ realizuje (částečné) zobrazení $M : \Sigma^* \to \Sigma^*$ definované následovně pro libovolné $w \in \Sigma^*$: zastaví-li se M pro vstup w (tedy skončí-li jeho výpočet, začne-li se slovem w na pásce), pak M(w) je definováno a rovná se řetězci (souvislému úseku neprázdných znaků), který je obsahem pásky v příslušné koncové konfiguraci; nezastaví-li se M na w, pak M(w) definováno není.

Turingův stroj M přijímá (akceptuje, částečně rozhoduje) jazyk (množinu) $L \subseteq \Sigma^*$, jestliže pro libovolné slovo $w \in L$ se zastaví a pro libovolné slovo $w \notin L$ se nezastaví.

Turingův stroj M rozhoduje jazyk (množinu) $L \subseteq \Sigma^*$, jestliže pro libovolné slovo $w \in \Sigma^*$ se zastaví a rozhodne (například koncovým stavy q_{ano}, q_{ne}), zda
$w \in L$ či nikoliv.

Úkoly k zamyšlení:

Někdy je výhodnější pracovat s Turingovými stroji, které mají pouze jeden koncový stav. V takovém případě pak stroj dává najevo svou odpověď o přijetí/zamítnutí slova stavem pásky po skončení výpočtu. Zamyslete se nad tím, že tento fakt nic nemění na výpočetní síle takovýchto strojů.

Je vhodné říci, že jako synonymum k pojmu ‘turingovsky vyčíslitelné (zobrazení)’ se užívá pojem částečně rekurzivní (funkce), synonymem k ‘turingovsky rozhodnutelný jazyk (množina)’ je rekurzivní jazyk (množina) a místo ‘turingovsky částečně rozhodnutelný jazyk (množina)’ se užívá rekurzivně spočetný jazyk (množina).

Příklad

Navrhneme Turingův stroj M přijímající jazyk $L = \{a^n b^n c^n \mid n \geq 0\}$. Budeme pracovat s modifikací stroje, u kterého je levý konec pásky označen symbolem Ã. Stroj nejdříve posouvá svoji hlavu až na konec vstupu a kontroluje, zda řetězec zapsaný na pásce je ve tvaru $\{a^* b^* c^*\}$. Přitom vůbec nemění obsah pásky. Když narazí na první prázdné políčko, začne se posouvat doleva až na levý konec pásky. Následuje cyklus, ve kterém hlava přepíše symbolem X vždy jeden symbol a, jeden b, jeden c a vrátí se na levý konec pásky. Když vstupní řetězec patří do jazyka L, tak stroj nakonec přepíše všechny symboly a, b, c symbolem X a přijme slovo. V opačném případě vstup zamítne.

Nechť M = (Q, Σ, Γ, δ, q_0, F), kde

Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_{ano}, q_{ne}}

Σ = {a, b, c}

Γ = Σ ∪ { Ã , #, X}

F = {q_{ano}, q_{ne}}

Přechodová funkce je určena následující tabulkou:

	$\triangleright$	a	b	c	$\#$	X
q_0	$(q_0, \triangleright, R)$	(q_0, a, R)	(q_1, b, R)	(q_2, c, R)	(q_3, X, L)	$-$
q_1	$-$	$(q_{ne}, -, -)$	(q_1, b, R)	(q_2, c, R)	(q_3, X, L)	$-$
q_2	$-$	$(q_{ne}, -, -)$	$(q_{ne}, -, -)$	(q_2, c, R)	(q_3, X, L)	$-$
q_3	$(q_4, \triangleright, R)$	(q_3, a, L)	(q_3, b, L)	(q_3, c, L)	(q_3, X, L)	(q_3, X, L)
q_4	$-$	(q_5, X, R)	$(q_{ne}, -, -)$	$(q_{ne}, -, -)$	$(q_{ano}, -, -)$	(q_4, X, R)
q_5	$-$	(q_5, a, R)	(q_6, X, R)	$(q_{ne}, -, -)$	$(q_{ne}, -, -)$	(q_5, X, R)
q_6	$-$	$-$	(q_6, b, R)	(q_3, X, L)	$(q_{ne}, -, -)$	(q_6, X, R)

Symbol − v tabulce vyjadřuje, že není podstatné, co se zapíše na uvedené místo (můžeme tam doplnit cokoli vyhovující definici). Výpočet stroje M pro vstup $a^2b^2c^2$ je

$$
\begin{array}{llllll}
(q_0 \rhd aabbcc\#) & \vdash & (\rhd q_0 aabbcc\#) & \vdash & (\rhd a q_0 abbcc\#) & \vdash \\
(\rhd aaq_0 bbcc\#) & \vdash & (\rhd aabq_1 bcc\#) & \vdash^3 & (\rhd aabbccq_2\#) & \vdash \\
(\rhd aabbcq_3 c\#) & \vdash^7 & (\rhd q_4 aabbcc\#) & \vdash & (\rhd X q_5 abbcc\#) & \vdash^2 \\
(\rhd XaXq_6 bcc\#) & \vdash^2 & (\rhd XaXq_3 bXc\#) & \vdash^5 & (\rhd q_4 XaXbXc\#) & \vdash^2 \\
(\rhd XXq_5 XbXc\#) & \vdash^2 & (\rhd XXXXq_6 Xc\#) & \vdash^2 & (\rhd XXXXq_3 XX\#) & \vdash^6 \\
(\rhd q_4 XXXXXXX\#) & \vdash^7 & (\rhd XXXXXXXq_4\#) & \vdash & (\rhd XXXXXXq_{ano} XX\#) &
\end{array}
$$

3.3 Kódování Turingových strojů

Existuje mnoho různých způsobů, jak kódovat Turingovy stroje. Ukážeme si jednu z možných variant. Pro jednoduchost se omezíme na stroje se vstupní
abecedou {0, 1} a jedním koncovým stavem. Náš uvedený tvar kódu bude mít
tu vlastnost, že nám k zakódování postačí opět pouze dva symboly {0, 1}. Mějme tedy Turingův stroj

$M = (Q, \{0, 1\}, \{0, 1, \#\}, \delta, q_1, \{q_2\})$

dále bez újmy na obecnosti předpokládáme, že $Q = \{q_1, q_2, \ldots, q_n\}$ je množina stavů a že q_2 je jediný koncový stav.

Je vhodné pojmenovat symboly 0, 1 a # synonymy X_1, X_2, X_3. Obdobně také hodnoty pro pohyb hlavou L, R, a N pojmenujeme D_1, D_2, D_3. Pak obecný tvar pro přechod $\delta(q_i, X_j) = (q_k, X_l, D_m)$ zakódujeme binárním řetězcem

$0^i10^j\ 10^k10^l10^m.$ (1)

Binární kód Turingova stroje M je

$111\ kod_1\ 11\ kod_2\ 11 \bullet\bullet\bullet 11\ kod_r\ 111, \qquad (2)$

kde každý kod_i je řetězec ve tvaru (1) a každý přechod z M je zakódován jedním kod_i. Přechody nemusí být v žádném speciálním pořadí, takže každý Turingův stroj má ve skutečnosti vícero různých kódů. Každý z těchto kódů
stroje M bude značen $\langle M \rangle$.

Každý binární řetězec může být interpretován jako kód pro nejvýše jeden Turingův stroj; mnoho binárních řetězců není kódem žádného Turingova stroje.

Dvojice M a w je reprezentována řetězcem ve tvaru (2) následovaným slovem w. Každý z takovýchto řetězců bude označen $\langle M, w \rangle$.

Příklad

Mějme Turingův stroj $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, \#\}, \delta, q_1, \{q_2\})$, který má následující přechody:

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

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

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

$\delta(q_3, \#) = (q_3, 1, L)$.

Pro dvojici $\langle M, 1011 \rangle$ dostaneme následující kód

11101001000101001100010101001011

000100100101001100010001000100101111011

Právě uvedený typ kódování měl tu vlastnost, že používal pouze symboly {0, 1}, což je vhodné zejména v některých situacích, při práci s Turingovými
stroji. Ukážeme si ještě jeden typ kódování, který je o něco ‘přívětivější’, alespoň co se týče přehlednosti výsledného kódu. Při zpracovávání korespondenčního úkolu Vám doporučuji použít právě tento druhý způsob kódování, případně si můžete navrhnout svůj vlastní formát kódu Turingova stroje. Měli byste si být vědomi, že různé druhy kódování jsou ve své podstatě ekvivalentní v tom smyslu, že musí existovat jednoduché algoritmické pře- vody mezi nimi. Tedy, že když máme libovolné dva různé formáty kódování Turingových strojů, tak zjevně existuje algoritmický postup, který jako vstup dostane kód Turingova stroje v jednom z dohodnutých formátů a jako výstup vydá transformaci tohoto kódu do druhého formátu, přičemž musí zůstat zachována stejná funkčnost stroje.

Budeme kódovat stroj M, který splňuje stejné požadavky jako v předchozím případě. Pro již avízovanou přehlednost použijeme k zakódování stroje rozšířenou abecedu sestávající z následujících symbolů: {#, 0, 1, L, R, N, ¢ }.

Využijeme toho, že máme stavy Turingova stroje seřazeny, a že z každého stavu existují maximálně tři možné přechody v závislosti na tom, který ze tří

možných symbolů z množiny {0, 1, #} je zrovna umístěn pod čtecí hlavou. Každý stav stroje budeme tedy kódovat trojicí po sobě jdoucích zakódovaných přechodů pro jednotlivé symboly oddělených znakem ¢ . Celý kód stroje

bude tedy sestaven z takovýchto tříčlenných skupin. Pro technické usnadnění práce s kódem stroje obklopíme tento ještě z každé strany dvěmi symboly ¢ . Na začátku i konci kódu budeme mít sekvenci tří symbolů ¢ a uprostřed kódu se vyskytují dva ¢ po sobě pro oddělení již zmíněných trojic. Jednotlivé přechody můžeme kódovat například následovně: $(q_1, 1) \rightarrow (q_3, 0, R)$ zakódujeme na druhé pozici (protože se čte symbol 1) v první skupině (protože se kóduje přechod z prvního stavu) jako řetězec 111R0, kde první tři jedničky určují, že se půjde do třetího stavu, symbol R přímo značí posun hlavy doprava a poslední symbol určuje, co se má zapsat na pásku. Pokud přechod není definován, tak jej zakódujeme jako symbol #.

Příklad

Vezměme stejný Turingův stroj, jako v předchozím příkladu $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, \#\}, \delta, q_1, \{q_2\})$, který má následující přechody:

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

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

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

$\delta(q_3, \#) = (q_3, 1, L)$.

Pro dvojici $\langle M, 1011 \rangle$ dostaneme následující kód

¢ ¢ ¢ #¢ 111R0¢ #¢ ¢ #¢ #¢ #¢ ¢ 1R1¢ 11R0¢ 111L1¢ ¢ ¢ 1011

3.4 Ekvivalenty a modifikace Turingova stroje

Různých jiných formalizací pojmu algoritmus se v 30. letech a později objevilo více (např. Postovy systémy, Markovovy algoritmy, kalkul částečně rekurzivních funkcí atd.; nám je dnes asi nejbližší ztotožnění

pojmu algoritmus s pojmem počítačový program – např. program v jazyce Pascal). Všechny tyto formalizace se ukázaly být ekvivalentní (umějí realizovat tatáž zobrazení

$\Sigma^* \rightarrow \Sigma^*$, resp. přijímat (rozhodovat) tytéž jazyky – podmnožiny Σ^*).

Tato ekvivalence se většinou dá ukázat předvedením schopnosti simulovat jeden prostředek jiným. V tomto kursu si to ilustrujeme prokázáním ekvivalence různých modifikací základního modelu Turingova stroje.

V tomto odstavci se budeme zabývat Turingovými stroji, které mají více než jednu pásku. Takové stroje se označují k-páskové Turingovy stroje, k je přirozené číslo, které udává počet pásek v případech, kdy je počet pásek důležitý.
Každá páska má levý konec, zprava je nekonečná a přísluší jí vlastní hlava, která může číst i zapisovat. Volba přechodu, který se v daném okamžiku provede, závisí na aktuálních symbolech všech pásek a na aktuálním stavu stroje.

Akce, které se při přechodu provedou, jsou obdobné jako u základního Turingova stroje. V jednom kroku může stroj přejít do jiného stavu, přepsat symboly, nad kterými jsou umístěny jednotlivé čtecí hlavy stroje a posunout tyto hlavy doleva, doprava, či nechat je na místě (není nutné, aby se všechny hlavy posunovaly ve stejném směru). Přechodová funkce takto modifikovaného stroje bude mít následující tvar (pro jednoduchost

můžeme předpokládat, že na všech páskách používáme stejnou abecedu Γ):

Přechodová funkce $\delta : (Q \setminus F) \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, N\}^k$

Testování, zda vícepáskový Turingův stroj přijme daný řetězec symbolů, začíná z počátečního stavu, kdy je vstupní řetězec zapsán na první pásce (ve stejném tvaru jako u klasického jednopáskového stroje). Ostatní pásky jsou prázdné hlavy jsou umístěny nad nejlevějším místem. Řetězec je přijat, jestliže stroj přejde z počáteční konfigurace do koncového stavu.

Dalo by se očekávat, že vícepáskový Turingův stroj bude mít větší sílu, než jeho jednopáskový protějšek. Následující věta však ukáže, že to není pravda. Užití vícepáskového stroje může být v některých případech vhodnější, ale množina jazyků přijímaných Turingovým strojem se tímto způsobem nezvětší.

Věta 2.1 Ke každému vícepáskovému Turingovu stroji M existuje jednopáskový Turingův stroj M_1 tak, že $L(M) = L(M_1)$.

Důkaz: Pouze naznačíme princip provedení důkazu. Předpokládejme, že M je k-páskový Turingův stroj, který přijímá jazyk L. Budeme postupovat tak, že ukážeme, že obsah k-pásek lze zobrazit na jednu pásku tak, aby akce stroje

M byly simulovány jednopáskovým strojem M_1. Představme si pásky stroje

M paralelně pod sebou, zarovnané podle jejich levého konce. Dostaneme tím v podstatě tabulku o k-řádcích a nekonečným počtem sloupců, které pokračují směrem doprava. Na každý sloupec se nyní můžeme dívat jako na uspořádanou k-tici, přičemž je zřejmé, že všech možných takovýchto k-tic je

konečně mnoho - konkrétně jich je $|\Gamma|^k$ (když do Γ počítáme i symbol #).

Pro zakódování konfigurace k-páskového stroje M je ještě zapotřebí uložit informaci o tom, kde se nacházejí jeho čtecí hlavy. To můžeme udělat rozšířením páskové abecedy stroje o množinu nových symbolů $\Gamma' = \{x' \mid \text{pro } x \in \Gamma\}$, přičemž nový symbol x´ bude použit pro znázornění toho, že čtecí hlava je zrovna na pozici symbolu x. Páskova abeceda stroje M_1 bude tedy obsahovat uspořádané k-tice ve tvaru $(x_1, x_2, \ldots, x_k)$ pro $x_i \in (\Gamma \cup \Gamma')$ (přičemž každá k-tice reprezentuje jeden samostatný symbol!) a navíc ještě prázdný symbol #.

Například pro k = 3 a $\Gamma = \{0, 1, \#\}$ dostaneme jako jednu z možných konfigurací ve tvaru:

1	1	0	$\overline{0}$	1	0	1	1	0	1	1	1	0	#		
0	0	1	0	1	1	$\overline{1}$	0	1	0	1	0	1	#	#	. . .
1	$\overline{0}$	1	1	1	0	1	1	0	1	1	0	0	#		

V tomto případě je pásková abeceda stroje M_1 tvořena uspořádanými trojicemi tvaru (x_1, x_2, x_3), kde $x_i \in \{0, 1, \#, 0', 1', \#'\}$ a prázdným symbolem
#.

Výpočet jednopáskového stroje M_1 nad slovem w bude probíhat tak, že nejprve toto slovo 'přeloží' do vícepáskového formátu, nastaví pozice všech hlav na nejlevější pozici a započne samotnou simulaci k-páskového stroje. Jeden krok k-páskového stroje bude muset simulovat posloupností několika dílčích kroků. Nejprve zjistí, které symboly jsou uloženy pod čtecími hlavami, a pak aktualizuje symboly na pásce podle příslušné přechodové funkce. Ještě se musejí dořešit 'technické' detaily jako je třeba, že když stroj narazí na
symbol #, tak jej vždy převede na symbol (#, #, #) atd.

3.5 Nedeterministické Turingovy stroje

Nedeterministický Turingův stroj se liší od tradičního Turingova stroje tím, že jeho činnost není úplně determinovaná, neboli v jednom okamžiku může existovat více než jeden proveditelný přechod pro aktuální dvojici stav/symbol. Jestliže nedeterministický Turingův stroj přejde do stavu, v němž pro aktuální symbol není proveditelný žádný přechod, stroj zruší výpočet. Jestliže nedeterministický Turingův stroj přejde do stavu, v němž je pro daný aktuální symbol proveditelných více přechodů než jeden,

vybere stroj nedeterministicky jeden z nich a pokračuje ve výpočtu zvoleným přechodem.

Formálně je nedeterministický Turingův stroj definován jako šestice (Q, Σ, Γ, δ, q_0, F) stejně jako deterministický Turingův stroj, s tím rozdílem, že δ je podmnožina kartézského součinu ((Q \ F) × Γ) × (Q × Γ × {L, R, N }) místo funkce z (Q \ F) × Γ do Q × Γ{L, R, N }. Z toho vyplývá, že nedeterministické Turingovy stroje tvoří třídu obsahující deterministické Turingovy stroje, které jsme doposud zkoumali. Říkáme, že řetězec w je přijat nedeterministickým Turingovým strojem M, jestliže je možné, aby stroj M přešel do koncového stavu, začne-li výpočet se vstupem w. Slovní obrat 'je možné' chápeme v tom smyslu, že pokud stroj nedosáhne koncového stavu, může to být výsledek špatného rozhodnutí při volbě přechodu a nikoli odezva na vstupní řetězec. Množinu všech řetězců přijímaných nedeterministickým Tu- ringovým strojem M definujeme jako jazyk L(M). Řetězec w patří do L(M) tehdy a jenom tehdy, jestliže existuje posloupnost přechodů stroje M, která vede při vstupu w k dosažení koncového stavu.

Nedeterministický Turingův stroj je zobecněním tradičního Turingova stroje a proto každý jazyk přijímaný tradičním Turingovým strojem, je přijímán také nedeterministickým Turingovým strojem. Důležitějším poznatkem je však to, že nedeterministické Turingovy stroje nejsou schopny přijímat více jazyků než deterministické.

Věta: Ke každému nedeterministickému Turingovu stroji M existuje deterministický Turingův stroj D tak, že L(M) = L(D).

Důkaz: Předpokládejme, že M je nedeterministický Turingův stroj, který přijímá jazyk L(M). Musíme dokázat existenci deterministického Turingova stroje, který přijímá stejný jazyk. Navržený stroj bude fungovat tak, že na
začátku výpočtu 'ozávorkuje' vstupní slovo speciálními symboly (např. ¢, který není součástí páskové abecedy původního stroje). Vzhledem k tomu, že páska je zleva konečná, provede to tak, že vstupní slovo w posune o jeden symbol doprava a teprve pak zapíše značku na první pozici pásky zleva a za poslední znak slova w. Z technických důvodů takto upravené slovo ozávorkujeme ještě dalším novým symbolem nevyskytujícím se v původní abecedě (např. $). Dostaneme tedy slovo ve tvaru $¢ w¢ $. Dále rozšíříme páskovou abecedu stroje o množinu Γ′. Pomocí symbolů z této množiny budeme opět zaznamenávat aktuální pozici hlavy v simulovaných konfiguracích původního nedeterministického stroje M.

Na začátku vlastního výpočtu označíme první symbol slova w symbolem pro umístění čtecí hlavy a simulace může začít. Simulace bude 'sledovat' přechodovou funkci nedeterministického stroje s tím, že když narazí při výpočtu na situaci, ve které má více možností, tak provede 'zduplikování' dané konfigurace tolikrát kolik možností má (resp. vytvoří o jednu kopii méně, protože bude dále pracovat také s originálem dané konfigurace). Duplikaci provede jednoduše tak, že zkopíruje danou konfiguraci, (kterou

má ozávorkovanou mezi symboly ¢) mezi poslední ¢ a $, který je umístěn až na samém konci pásky vpravo. Na pásce bude tedy postupně během výpočtu narůstat počet možných konfigurací, které budou simulovány zároveň! Tato paralelní simulace může být například prováděna tak, že se budou procházet všechny vytvořené konfigurace na pásce zprava do doleva a v každé se provede jeden 'elementární' krok. Pokud se během výpočtu v některé z konfigurací dojde do koncového stavu původního nedeterministického stroje, tak se všechny ostatní konfigurace smažou a tato se přesune na začátek pásky. Ještě je za- potřebí pamatovat na 'technický' detail, když se během simulace dojde do stavu, že pozice hlavy je na konci slova představujícího konfiguraci (tedy před
pravým symbolem ¢ aktuální konfigurace), tak se musí zbývající část pásky
za hlavou posunout doprava a na místo původního symbolu ¢ , který se také
posunul doprava zapsat #.
Prokázání zmíněných ekvivalencí a další velmi dobré důvody nás vedou k přijímání tzv. Churchovy (či Church–Turingovy) teze, že třída jazyků přijímaných Turingovými stroji představuje vrchol hierarchie strojově rozpoznatelných jazyků.

Teze (Church–Turing) Ke každému algoritmu je možné zkonstruovat s ním ekvivalentní Turingův stroj (při vhodném vyjádření vstupů a výstupů jako řetězců v určité abecedě). Ekvivalencí zde rozumíme podmínku, že algoritmus i Turingův stroj se zastaví (tj. jejich běh, výpočet po konečném

počtu kroků skončí) právě pro tytéž vstupy, přičemž pro tyto vstupy budou příslušné výstupy totožné.

Průvodce studiem:

Pokud tedy známe nějaký postup k vyřešení zadaného problému, můžeme vždy sestrojit Turingův stroj, který tento postup bude realizovat.

Jinými slovy: každý rozhodnutelný problém (případně jazyk, množina) je turingovsky rozhodnutelný (tj. rekurzivní). Podobně každý částečně rozhod- nutelný problém (jazyk, množina) je turingovsky částečně rozhodnutelný (tj. rekurzivně spočetný).

Úkoly k zamyšlení:

Všimněte si, že jelikož každý Turingův stroj je příkladem algoritmu, je obrácené tvrzení zřejmé.

Churchovu tezi není možné dokázat jako matematickou větu (právě kvůli ‘intuitivnímu charakteru’ pojmu algoritmus); jelikož však máme velmi dobré důvody k jejímu přijetí, často pojmy ‘rekurzivní’ a ‘rozhodnutelný’ ztotožňujeme. Níže zmíníme důkazy algoritmické nerozhodnutelnosti

konkrétních problémů; ve skutečnosti exaktně dokážeme jen to, že nejsou rekurzivní (tj. turingovsky rozhodnutelné) – v tvrzení se tedy implicitně odvoláváme na Churchovu tezi, čehož bychom si měli být stále vědomi.

Úkoly k zamyšlení:

V textu se budeme držet ‘všeobecnějších’ pojmů jako je ‘rozhodnutelný’, ‘částečně rozhodnutelný’ i tam, kde by z hlediska matematické exaktnosti mělo být raději ‘rekurzivní’, ‘rekurzivně spočetný’. Bude dobré, když si toto aspoň na několika případech důkladně promyslíte. Dobrým cvičením může být příklad Postovy věty. Promyslete si její znění ve vztahu ke znění ‘Množina A je rekurzivní právě když A i $\neg$A jsou rekurzivně spočetné’. (Čím se liší důkazy věty v jednotlivých zněních? Proč jde vlastně o jednu a tutéž větu?)

Ve světle předchozích úvah bychom nyní měli chápat, že budeme-li dále otázky teorie vyčíslitelnosti studovat na Turingových strojích, dostaneme výsledky, které jsou robustní – nezávislé na volbě tohoto konkrétního modelu (ke stejným výsledkům bychom např. dospěli, pokud bychom používali např. pascalské programy – technicky by ovšem vše bylo náročnější [důvod je zřejmý již z porovnání definic Turingových strojů a Pascalu]).

O jaké (robustní) výsledky jde? Především samozřejmě půjde o prokázání nerozhodnutelnosti konkrétních problémů. Ještě předtím si ale ukážeme výsledek týkající se existence určitého, tzv. univerzálního algoritmu.

3.6 Univerzální Turingův stroj

Úkoly k zamyšlení:

Představte si, že Vám někdo předloží zadání Turingova stroje M (v dohodnutém formátu. De facto se určitě jedná o řetězec v určité dohodnuté abecedě, byť napsaný např. ve více řádcích na papíře) a nějaké jeho vstupní slovo w. Dotyčný Vás vyzve, ať předvedete (simulujete) výpočet stroje M na slově w. To, co pak budete provádět, bude zajisté naprosto mechanický (algoritmický) postup, který jste schopni aplikovat na libovolný Turingův stroj a libovolné jeho vstupní slovo. Realizujete tedy určitý konkrétní algoritmus!

Úkoly k zamyšlení:

Všimněte si, že zastaví-li se M na w, což budeme značit !M (w), vaše činnost po nějakém čase skončí – odhlížíme teď od času a prostoru, který byste k dané simulaci potřebovali – a jste schopni vydat ‘výstup’ totožný s (neprázdným) obsahem pásky v koncové konfiguraci stroje M dosažené při výpočtu nad w.

Tento obsah pásky (konečný řetězec) značíme M(w). Pokud se M na w nezastaví, tj. ¬!M (w), vaše simulace je potenciálně nekonečná (a o nějakém ‘výstupu’ pak nebudeme hovořit).

Tento algoritmus, který vykonáváte při zmíněné simulaci Turingových strojů, musí ovšem podle Churchovy teze být rovněž realizován nějakým konkrétním Turingovým strojem! Jistě Vás proto nepřekvapí následující věta. Zde Kod(M) bude představovat dohodnutou formu zápisu Turingova stroje M . Mělo by být zřejmé, že také zde můžeme předpokládat, že užijeme jen
abecedu {0, 1}, tj. Kod(M) ∈ {0, 1}*.

Věta 2.4 (O univerzálním Turingovu stroji)
Lze sestrojit Turingův stroj U takový, že pro libovolný Turingův stroj M s abecedou {0, 1} a libovolné slovo w ∈ {0, 1}* platí:
1) !M (w) ⇔ !U (Kod(M) • w) a
2) jestliže !M (w), pak M (w) = U (Kod(M) • w)

Průvodce studiem:

Co vlastně říká tato věta? Jednoduše řečeno se v ní tvrdí, že lze sestrojit takový univerzální Turingův stroj U , který je schopen jako vstup přijmout kód jiného stroje M spolu s jeho vstupem w a že pak tento univerzální stroj

dokáže simulovat chod zadaného stroje M . Přičemž se požaduje, aby po skončení simulace (samozřejmě pouze pokud bude výpočet konečný) byl stav pásky stroje U totožný s výstupem, jaký bychom dostali, kdybychom přímo spustili zadaný stroj M na slovo w.

Korespondenční úkol:

Sestrojte takový konkrétní stroj U . Vyžaduje to samozřejmě určitý čas, ale de facto se jedná o naprogramování jednoduchého algoritmu, který dobře znáte
– musíte ovšem programovat v jazyce hodně ‘nízké úrovně’. Váš univerzální

stroj může samozřejmě používat pro jeho naprogramování co nejvýhodnější formu kódu neomezující se na abecedu {0, 1}.

Úkoly k zamyšlení:

Uvědomte si ale, že byste měli být schopni rutinně upravit váš U pro případ
Kod(M) ∈ {0, 1}* a navíc by šlo požadovat, aby U měl také jen abecedu {0, 1}. Mimo jiné to znamená, že jej lze aplikovat i na svůj vlastní kód – zamyslete se nad tím!

Úkoly k zamyšlení:

Předchozí věta mluví o univerzálním Turingovu stroji (schopném ‘provádět práci’ libovolného Turingového stroje). Z hlediska dříve zmíněné robustnosti našich výsledků existuje takový univerzální ‘program’ v libovolné formalizaci algoritmů. Obecně tedy můžeme hovořit o univerzálním algoritmu. Promyslete si, proč se na počítač lze dívat jako na zařízení realizující onen univerzální algoritmus.

3.7 Jazyky přijímané Turingovými stroji a neomezené jazyky

Třída neomezených jazyků (jazyky typu 0) je generovaná gramatikami, jejichž přepisovací pravidla nemusí mít žádný speciální tvar. Levá i pravá strana pravidla může být tvořena libovolným konečným řetězcem

terminálů a neterminálů, pokud je v levém řetězci alespoň jeden neterminál. Někdy se takovéto gramatice říká Obecná generativní gramatika.

Třída neomezených jazyků může být generována ‘gramaticky’, tj. jejich řetězce mohou být analyzovány cestou vyhledávání frází větné formy. V této sekci budeme charakterizovat neomezené jazyky jako jazyky přijímané Turingovými stroji. Přesněji řečeno, třída neomezených jazyků odpovídá právě třídě jazyků, přijímaných Turingovými stroji. Důkaz tohoto tvrzení provedeme ve dvou krocích. Nejdříve ukážeme, že každý jazyk přijímaný Turingovými stroji, je neomezeným jazykem. Pak dokážeme, že každý neomezený jazyk je přijímán Turingovými stroji.

Každý jazyk přijímaný Turingovými stroji patří do třídy neomezených jazyků.

Důkaz: Pro následující úvahy budeme používat Turingův stroj, který bude mít jediný koncový stav q_K a po skončení výpočtu nad zadaným slovem w bude jeho stav pásky Y´ ### . . . při přijetí slova w a N´ ### . . . při nepřijmutí slova w.

Důkaz věty spočívá v konstrukci gramatiky, která generuje stejný jazyk, jaký přijímá daný Turingův stroj. Při této konstrukci využijeme notace pro

konfiguraci Turingova stroje, tak jak jsme ji uvedli v definici TS. S použitím této notace budeme obsah pásky stroje reprezentovat řetězcem páskových symbolů uzavřeným v závorkách. Levý konec pásky znázorníme symbolem [, potom bude následovat řetězec symbolů na pásce počínaje nejlevějším místem, obsahující alespoň jeden prázdný symbol za posledním neprázdným a nakonec uzavřeme řetězec závorkou]. Páska s obsahem xxyx### . . . bude reprezentována jako [xxyx#] nebo např. [xxyx#####] a pásku obsahující [###xx#yx#x###] lze reprezentovat posloupností [###xx#yx#x##].
Aby byla reprezentace konfigurace stroje úplná, vložíme symbol označující aktuální stav stroje právě vlevo od aktuálního symbolu v naší reprezentaci pásky stroje. Je-li p jedním ze stavů stroje, potom [xpxyxx#] bude reprezentovat konfiguraci stroje, který je právě ve stavu p a stav jeho pásky je xxyxx##. (Můžeme předpokládat, že symboly použité pro reprezentaci stavů stroje jsou různé od páskových symbolů stroje.)

Naším úkolem bude ukázat, že ke každému jazyku L přijímanému Turingovými stroji můžeme sestrojit neomezenou gramatiku, která generuje jazyk L. Zvolíme Turingův stroj M , který přijímá řetězce zastavením s konfigurací pásky Y´ ### . . . a pro nějž L(M) = L.

Pro takový stroj definujeme gramatiku G tímto způsobem: Neterminály v G budou S (počáteční symbol gramatiky), [,], symboly reprezentující stavy stroje M a páskové symboly M včetně # a Y. Terminály gramatiky G budou symboly vstupní abecedy stroje M .

Jediným přepisovacím pravidlem, které obsahuje startovací symbol bude

$S \rightarrow [q_K\ Y\ \#]$

kde q_K je koncový stav stroje. Toto pravidlo zaručuje, že všechny derivace v této gramatice začnou od konce nějaké posloupnosti konfigurací stroje. Zavedeme také pravidlo

$\#] \rightarrow \#\#],$

které nám umožní rozšířit derivací řetězec $[q_K\ Y\ \#]$ na libovolnou délku. Dále zavedeme pravidla simulující přechody v obráceném pořadí. Pro každý pře- chod tvaru $\delta(p, x) = (q, y, N)$ vytvoříme přepisovací pravidlo

$qy \rightarrow px.$

(Např. [z q y#] lze přepsat na [z px#], což odráží skutečnost, že stroj přejde z konfigurace [z px#] aplikací přechodové funkce $\delta(p, x) = (q, y, N)$ do konfigurace [z q y#].) Pro každý přechod tvaru $\delta(p, x) = (q, y, R)$ zavedeme pravidlo

$yq \rightarrow px.$

(Např. [y q yz#]) lze přepsat na [pxyz#].) Pro každý přechod tvaru δ(p, x) = (q, y, L) a každý páskový symbol z stoje M , zavedeme pravidlo

q zy → z px.

(Např. [q zy##] lze přepsat na [z px##]) Seznam přepisovacích pravidel doplníme třemi pravidly, která umožní za určitých podmínek odstranit neterminály [, q0, # a]. Jsou to pravidla:

$[q_0\# \rightarrow \varepsilon$
$\#\#] \rightarrow \#]$
$\#] \rightarrow \varepsilon$

(Jestliže derivacemi získáme počáteční konfiguraci [q0xyx###] můžeme od- stranit neterminály tak, že dostaneme řetězec xyx.)

Nakonec ukážeme, že L(M) = L(G). Je-li w řetězec patřící do L(M), musí existovat posloupnost konfigurací stroje M počínající [q0w#] a končící [qK Y #]. Můžeme proto vytvořit derivaci řetězce w ve tvaru

$S \Rightarrow [qK\ Y\ \#] \Rightarrow \bullet\bullet\bullet \Rightarrow [q0w\#] \Rightarrow w\#] \Rightarrow w$

Začneme jednoduše aplikací pravidla S → [qK Y #] a potom budeme opakovaně aplikovat pravidlo #] ⇒ ##] dokud řetězec [qK Y ## . . . #] nebude tak dlouhý jako jedna z konfigurací v posloupnosti, která reprezentuje výpočet stroje M . Dále aplikujeme v opačném pořadí přepisovací pravidla odpovídající přechodům v původní posloupnosti konfigurací. Tímto získáme větnou formu [q0w## . . . #], kterou můžeme redukovat na w pomocí pravidel

##] → #]
#] → ε
[q0# → ε

Z toho vyplývá, že w patří do L(G). Obráceně, je-li dán řetězec z L(G), jeho derivace představuje posloupnost konfigurací, která naopak ukazuje, jak je řetězec přijímán strojem M . Proto každý řetězec z L(G) je také v L(M).

Věta: Každý neomezený jazyk je přijímán Turingovými stroji.

Důkaz: Začneme tím, že si všimneme, že je-li důkaz věty o existenci jednopáskového stroje k vícepáskovému aplikován na nedeterministický vícepáskový Turingův stroj, získáme nedeterministický jednopáskový stroj M′, který bude přijímat stejný jazyk jako vícepáskový stroj. Toto zjištění nám umožní provést důkaz tím, že ukážeme, že ke každé gramatice G

existuje nedeterministický dvoupáskový Turingův stroj N tak, že L(G) = L(N). Stroj N může být potom simulován nedeterministickým jednopáskovým Turingovým strojem, který lze dále simulovat tradičním Tu- ringovým strojem.

Nyní ukážeme, jak lze každé přepisovací pravidlo gramatiky implementovat Turingovým strojem. Přesněji řečeno, jestliže se na pásce stroje někde objeví řetězec symbolů v a gramatika obsahuje pravidlo v → w, kde w reprezentuje (potenciálně prázdný) řetězec terminálů a neterminálů, potom může stroj pomocí levých a pravých posunů spolu s operacemi zápisu nahradit řetězec v řetězcem w. Nyní vytvoříme nedeterministický dvoupáskový stroj, který pracuje takto: Pásku 1 využije k uložení vstupního řetězce, který se bude testovat. Zapíše startovací symbol gramatiky na pásku 2. Potom opakovaně nedeterministickým způsobem aplikuje přepisovací pravidla na řetězec, který je na pásce 2. (Říkáme nedeterministickým způsobem, protože v daném okamžiku může existovat více než jedno použitelné pravidlo.) Když se na pásce 2 objeví řetězec pouze z terminálů, srovná tento řetězec se vstupním řetězcem uloženým na pásce 1. Jsou-li řetězce identické, zastaví; když budou řetězce různé, vstoupí do nekonečné smyčky.

Při tomto výpočtu se využívá páska 2 k tomu, aby se na ní postupně vytvářely derivace podle přepisovacích pravidel gramatiky. Jestliže lze vstupní řetězec získat derivacemi podle dané gramatiky, je možné, že bude vygenerován na pásce 2. V tomto případě stroj přijme vstup tak, že zastaví.

Jestliže vstupní řetězec nelze derivovat podle dané gramatiky, řetězec generovaný na pásce 2 nebude nikdy odpovídat vstupnímu a stroj nebude moci řetězec přijmout. Z toho plyne, že jazyk přijímaný strojem je právě jazyk generovaný gramatikou.

Shrnutí:

- Nejen proto, abychom mohli prokázat nerozhodnutelnost některých konkrétních problémů, musíme nejprve formalizovat a přesně definovat pojem algoritmu. My budeme pojem algoritmu ztotožňovat s Turingovým strojem, který navrhl ve 30-tých letech Alan M. Turing.
- Turingovy stroje jsou jistým způsobem podobné konečným automatům s tím, že mají jistým způsobem rozšířenu instrukční sadu akcí, které mohou provádět. Každý krok stroje je určen tzv. přechodovou funkcí, která v závislosti na aktuálním čteném symbolu a vnitřním stavu stroje určuje chování stroje, do kterého nového vnitřního stavu má přejít, co zapsat na pásku a kam pohnout hlavou. Výpočet stroje vždy končí přechodem do některého z jeho koncových stavů.

- Turingovy stroje se dají používat buď k realizaci zobrazení nebo k rozhodování a přijímání jazyků. V této souvislosti pak hovoříme o rekurzivních a rekurzivně spočetných jazycích.
- Každý Turingův stroj je možno zakódovat v předem domluveném formátu nad zvolenou abecedou. V tomto kódu je uloženo celé chování stroje včetně počtu stavů a přechodové funkce, tak aby bylo možno při zadání tohoto kódu spolu se vstupem rekonstruovat resp. simulovat

výpočet tohoto stroje nad zadaným vstupním slovem. Přesně tohle je úloha tzv. Univerzálního Turingova stroje. Jedná se v jistém smyslu o univerzální algoritmus, který je schopen realizovat libovolné zobrazení, které dostane zadáno ve formě kódu Turingova stroje.

- I když povolíme Turingovu stroji používat více pásek, respektive zavedeme nedeterministické chování tím, že povolíme stroji, aby si v jednom okamžiku sám zvolil jednu z možných akcí jeho výpočetní síla se nezvětší. Tyto i jiné další důvody nás vedou k přesvědčení o platnosti Church–Turingovy teze, která tvrdí, že každý algoritmický postup je realizovatelný Turingovým strojem.
- Turingovy stroje jsou schopny přijímat jazyky, které jsou v Chomského hierarchii generativních jazyků reprezentovány jazyky typu 0, tyto jsou generovány gramatikami, na které nejsou kladeny žádná omezení. Proto se těmto jazykům také někdy říká neomezené jazyky.

Kontrolní otázky:

1. Popište slovně co to je Turingův stroj a porovnejte ho s konečnými automaty.

2. Vysvětlete v jakém smyslu může Turingův stroj používat pásku jako paměť.

3. Objasněte, jak jsou Turingovy stroje schopny realizovat zobrazení.

4. Vysvětlete rozdíl mezi pojmy ‘přijímání jazyka’ a ‘rozhodování jazyka’.

5. Jaký je rozdíl mezi deterministickým a nedeterministickým Turingovým strojem?

6. Jaký je praktický přínos Curch–Turingova teze?

7. Objasněte činnost Univerzálního Turingova stroje.

8. Charakterizujte třídu jazyků rozpoznatelných Turingovými stroji.

Cvičení:

1. Navrhněte Turingův stroj, který realizuje zobrazení

$w \rightarrow ww$ pro $w \in \{a, b\}*$

2. Navrhněte Turingův stroj, který rozhoduje jazyk

$L = \{ww^R \mid w \in \{a, b\}*\}$

3. Zakódujte stroj z předchozího cvičení pomocí Vámi zvoleného kódování.

4. Navrhněte Nedeterministický Turingův stroj, který rozhoduje jazyk

$L = \{ww \mid w \in \{a, b\}*\}$

Pojmy k zapamatování:

- Turingův stroj
- přechodová funkce
- konfigurace Turingova stroje
- relace následování
- počáteční a koncová konfigurace
- zastavení Turingova stroje

- realizace zobrazeni Turingovým strojem
- přijímání jazyka
- rozhodování jazyka
- rekurzivní jazyk
- rekurzivně spočetný jazyk
- nedeterministický Turingův stroj
- Univerzální Turingův stroj

4 Nerozhodnutelné problémy

Cíl:

V této sekci se budeme věnovat problémům, o kterých si ukážeme, že jsou algoritmicky nerozhodnutelné. Po jejím prostudování byste měli:

- být schopni ukázat, že existují problémy, které nejsou rozhodnutelné,
- znát postup, pomocí kterého se ukáže nerozhodnutelnost dalších problémů,
- umět vyjmenovat několik základních nerozhodnutelných problémů a

být schopni jejich nerozhodnutelnost dokázat.

Průvodce studiem

Nyní se zaměříme na poměrně abstraktní problematiku problémů, které nejsou rozhodnutelné. Zejména pochopení důkazu vyžaduje zvláštní úsilí, neboť se používá principu nepřímého důkazu, který není pro informatika příliš přirozený.

4.1 Problém zastavení

Je načase uvést příklad problému, který algoritmicky rozhodnutelný není.

Problém HP (Halting Problem) - Problém zastavení

Instance: Turingův stroj M – resp. jeho kód Kod(M) v abecedě {0, 1}
a slovo w ∈ {0, 1}∗.

Otázka: Zastaví se M na w (platí !M (w)) ?

Věta: Problém HP není rozhodnutelný.

Důkaz: Důkaz je vedený sporem. Předpokládejme, že existuje Turingův stroj H, který se pro libovolný vstup u ∈ {0, 1}∗ tvaru u = Kod(M) • w pro nějaký stroj M a slovo w zastaví a rozhodne, zda !M (w) či nikoliv (skončí např. buď ve spec. stavu qano nebo v qne). U stroje H je možné předpokládat
abecedu {0, 1}. Sestrojme nyní stroj H´ s abecedou {0, 1}, který se chová následovně: vstupní slovo v ∈ {0, 1}∗ nejprve zdvojí (vytvoří slovo vv) a na to ‘spustí’ (jako podprogram) stroj H. Jestliže (podprogram) H skončí ve stavu q_{ano}, stroj Hl přejde do nekonečného cyklu (a tedy se nezastaví); jestliže H skončí ve stavu q_{ne}, stroj H´ se zastaví (stav q_{ne} bude také jeho koncovým stavem). Když ovšem nyní prozkoumáme, zda se H´

při spuštění na svůj kód Kod(H´) zastaví či nezastaví, dospějeme při obou možnostech k logickému sporu (zastaví se a nezastaví se zároveň).

Jelikož HP je zřejmě částečně rozhodnutelný (částečně ho rozhoduje např. univerzální Turingův stroj), dostáváme: HP je částečně rozhodnutelný.

Důsledek: Problém HP je příkladem problému (či jazyka), který je částečně rozhodnutelný, ale není rozhodnutelný.

Průvodce studiem:

Je užitečné si všimnout, že při důkazu nerozhodnutelnosti problému HP jsme vlastně ukázali nerozhodnutelnost jeho podproblému, který označíme DHP.

Problém DHP (Diagonal Halting Problem)

Instance: Turingův stroj M daný svým kódem Kod(M)

Otázka: Zastaví se M na svůj kód (tj. na slovo Kod(M))?

Věta: Problém DHP je částečně rozhodnutelný, ale není rozhodnutelný.

4.2 Převody problémů

Máme-li dokázánu nerozhodnutelnost jednoho problému, je možno ji využít k prokázání nerozhodnutelnosti dalších problémů. Např. z nerozhodnutelnosti problému P ihned plyne nerozhodnutelnost jeho doplňkového problému (ANO, NE přehozeny).

Definice: Problém P 1 je (algoritmicky) převeditelný na problém P 2,

převeditelnost označme P 1 ;P 2, jestliže existuje algoritmus A, který pro libovolnou instanci I problému P 1 (chápanou jako jeho vstup) sestrojí (tzn. skončí svůj výpočet a jako výstup vydá) instanci problému P 2, označme ji (I), přičemž platí, že odpověď na otázku problému P 1 pro instanci I je ANO právě tehdy, když odpověď na otázku problému P 2 pro instanci A(I) je ANO.

Průvodce studiem:

Je důležité, aby se Vám právě uvedená definice dostala takříkajíc pod kůži, protože v podstatě celá teorie vyčíslitelnosti a posléze i teorie složitosti je založená na převodech jednoho problému na druhý. K převodům mezi

problémy se váže pěkná historka o jednom informatikovi. Byl s ním dělán rozhovor a jen tak mimochodem se ho zeptali, jak si vaří kávu. Jednoduše odpověděl, že napustí vodu do konvice, postaví na vařič a až voda začne vřít, tak si kafe zaleje. Jednomu z přihlížejících to nedalo a zeptal se, jak by postupoval, když by už nějaká voda v konvici byla. On mu na to s naprosto vážnou tváří odpověděl: „To je jednoduché, vodu bych vylil a pak bych postupoval stejně jako v předchozím případě. Vidíte, kde všude se dá znalost algoritmické převeditelnosti použít!

Úkoly k zamyšlení:

Nemělo by Vás překvapit, že pojem rekurzivní převeditelnosti se definuje obdobně s tím, že pojem algoritmus se nahradí pojmem Turingův stroj. Připomeňme, že při přijetí Churchovy teze jsou pojmy rekurzivní a algoritmické převeditelnosti totožné.

Jelikož problému typu ANO/NE koresponduje dříve uvedeným přirozeným způsobem určitý jazyk (sestávající ze všech řetězců popisujících instance s odpovědí ANO), dostáváme takto rovněž pojem (algoritmické) převeditelnosti jazyků.

Užitečnost uvedeného pojmu algoritmické převeditelnosti pro naše účely vyslovuje následující tvrzení, jehož důkaz by měl být zřejmý.

Tvrzení 3.4 Je-li P 1 ;P 2 a problém P 1 je nerozhodnutelný, je i problém P 2 nerozhodnutelný.

Průvodce studiem:

Právě uvedené tvrzení je jedna z nejdůležitějších vět teorie vyčíslitelnosti vůbec, proto je nezbytně nutné ji do důsledku promyslet a pochopit. Je důležité dávat pozor na pořadí problémů, který převádíme na který. Nejčastější chyba při používání této věty spočívá v tom, že když někdo chce prokázat nerozhodnutelnost nějakého problému P , tak se ho snaží převést na problém, o kterém již víme, že je nerozhodnutelný. Pozor!!! Dělá se to přesně naopak! A je to i logické. Postup je tedy takový, že vezmu nějaký problém P 1, o kterém vím, že je nerozhodnutelný a tento problém převedu na problém P 2, jehož nerozhodnutelnost se snažím prokázat. Hlavní myšlenka spočívá v tom, že kdyby problém P 2 byl rozhodnutelný, tak bych vlastně měl návod, resp. postup, jak řešit problém P 1, o kterém vím, že je nerozhodnutelný a tedy řešit nelze, což samozřejmě dává spor.

Příklad:

Takto se např. prokáže nerozhodnutelnost problému UHP - Uniform Halting

Problem.

Problém UHP (Uniform Halting Problem)

Instance: Turingův stroj M daný svým kódem Kod(M)

Otázka: Zastaví se M na každý vstup (tj. platí ∀w: !M (w)) ?

Budeme tedy postupovat tak, že převedeme HP na UHP. Při prokázání HP╝UHP stačí navrhnout algoritmus, který k zadanému (M, w) sestrojí stroj M´, jenž nejdříve otestuje vstup a v případě, že jde o w, 'spustí' (podprogram) M a v opačném případě se ihned zastaví. Je vidět, že když nově vytvořený stroj M´ dostane jako vstup slovo odlišné od w, tak se vždy zastaví, čili jedinou možnost se nezastavit má pouze tehdy, když na vstup dostane slovo w. V tomto případě se zastaví právě tehdy, když se na slovo w zastaví i původní stroj.

4.3 Postův korespondenční problém

Představme si dvojici seznamů neprázdných řetězců v nějaké abecedě. **Postův korespondenční problém** (dále jen PKP) řeší, zda existuje alespoň jedna posloupnost těchto dvojic, která utvoří stejný řetězec. Tento problém je obecně nerozhodnutelný, protože jeho rozhodnost je

podmíněna konkrétním vstupem, kdežto rozhodnost problému samotného musí nutně platit pro jakýkoliv vyhovující vstup.

Problém PKP (Postův korespondenční problém)

Instance: Dvojice seznamů $u_1, u_2, \ldots, u_n$ a $v_1, v_2, \ldots, v_n$ (pro něj. $n \geq 1$) neprázdných řetězců (slov) v nějaké abecedě.

Otázka: Má PKP pro danou instanci řešení? Tj. existují indexy $i_1, i_2, \ldots, i_r$, $r > 0$, tak, že $u_{i1}\, u_{i2} \ldots u_{ir} = v_{i1}\, v_{i2} \ldots v_{ir}$? (Jestliže $i_1 = 1$, hovoříme o iniciálním řešení.)

Příklad

Mějme ∑ = {a,b,c} a dále dva seznamy řetězců X a Y. Každý z nich obsahuje pět řetězců: **X** = <x1, x2, x3, x4, x5> a **Y** = <y1, y2, y3, y4, y5>, přičemž platí x1 = *abc*, x2 = *abba*, x3 = *c*, x4 = *bbba*, x5 = *abcc* a y1 = *ab*, y2 = *c*, y3 = *ccc*, y4 = *cbbb*, y5 = *aab*. Tento příklad si pro přehlednost převedeme do následující tabulky:

i	X	Y

1.	abc	ab
2.	abba	c
3.	c	ccc
4.	bbba	cbbb
5.	abcc	aab

Nalézt první dvojici je jednoduché, protože stačí nalézt takový pár, kde jeden řetězec je podřetězcem toho druhého z levé strany. Pokud by existoval příklad, kde se v jednom páru oba řetězce rovnají (např. *ab* a *ab*), pak by řešením byla posloupnost tohoto páru libovolného počtu.

V našem případě máme dvě možnosti, a to 1.pár (*abc* a *ab*) nebo 3.pár (*c* a *ccc*). Nejdříve si ukážeme cestu, kdy počáteční dvojice bude 3.pár. Skládané řetězce budeme zobrazovat následovně:

I : **3**

X: c

Y: ccc

"I" označuje posloupnost iterací zvolených párů, "X" a "Y" reprezentuje skládané řetězce pro dané seznamy. Je zřejmé, že následující pár budeme

volit podle seznamu X, kde nám chybí dva znaky k dorovnání, avšak jediný vyhovující pár je opět 3.pár, takže další krok by vypadal takto:

I : **3** $\overline{\mathbf{3}}$

X: cc

Y: cccccc

Tato cesta nemá řešení, jelikož skládaný řetězec ze seznamu Y tvoří trojnásobně delší řetězec než ten ze seznamu X, tzn. řetězce se nikdy nemohou rovnat. Musíme zvolit jiný počáteční pár, tedy 1.pár.

I : **1**

X: abc

Y: ab

Nyní je menší řetězec Y, proto budeme hledat v jeho seznamu takové řetězce,které začínají na chybějící podřetězec, tedy "*c*". Takto začínají řetězce 2, 3 a 4.

Při zvolení 2.páru skončíme s posloupností "121", kdy v seznamu Y není řetězec, který by obsáhnul přesahující podřetězec "*baabc*".

I : **121**

X: abcabbaabc

Y: abcab

Zvolením 3.páru se rychle dostaneme do stejného problému jako na začátku, a to k zacyklení řetězců "*c*" a "*ccc*".

I : **13$\overline{3}$**

X: abcc

Y: abccccccc

Správnou možností pro nás je 4.pár, kde nám v řetězci X přebývá symbol "*a*".

I : **14**

X: abcbbba

Y: abcbbb

Když použijeme stejný postup pro další hledání správného páru, dojdeme ke zdárnému výsledku v podobě posloupnosti **I**: 1453.

I : **1453**

X: abcbbbaabccc

Y: abcbbbaabccc

Příklad

Mějme $\Sigma = \{0, 1\}$ a dále dva seznamy řetězců U a V. Každý z nich obsahuje tři řetězce: $U = (u_1, u_2, u_3)$ a $V = (v_1, v_2, v_3)$, přičemž $u_1 = 1$, $u_2 = 10111$,

$u_3 = 10$ a $v_1 = 111$, $v_2 = 10$, $v_3 = 0$.

	Seznam U	Seznam V
i	u_i	v_i
1	1	111
2	10111	10
3	10	0

V tomto případě P KP má následující řešení: $r = 4$, $i_1 = 2$, $i_2 = 1$, $i_3 = 1$, $i_4 = 3$. Pak

$u_2u_1u_1u_3 = v_2v_1v_1v_3 = 101111110$

Příklad

Mějme $\Sigma = \{0, 1\}$ a dále dva seznamy řetězců U a V. Každý z nich obsahuje tři řetězce: $U = (u_1, u_2, u_3)$ a $V = (v_1, v_2, v_3)$, přičemž $u_1 = 10$, $u_2 = 011$,

$u_3 = 101$ a $v_1 = 101$, $v_2 = 11$, $v_3 = 011$.

	Seznam U	Seznam V
i	u_i	v_i
1	10	101

2	011	11
3	101	011

Předpokládejme, že tato instance P KP má řešení $i_1, i_2, \ldots, i_m$. Je zřejmé, že $i_1 = 1$, protože žádný řetězec začínající na $u_2 = 011$ nemůže být shodný s řetězcem začínajícím na $v_2 = 11$; obdobně pro $u_3 = 101$ a $v_3 = 011$. Napíšeme řetězec ze seznamu U nad odpovídající řetězec ze seznamu V. Takže máme:

10

101

Další výběr z U musí začínat symbolem 1. Tedy $i_2 = 1$ nebo $i_2 = 3$. Ale $i_2 = 1$ nebude fungovat, protože žádný řetězec začínající na $u_1u_1 = 1010$ se nemůže rovnat řetězci začínajícím na $v_1v_1 = 101101$. Pro $i_2 = 3$ máme

10101

101011

Vzhledem k tomu, že řetězec ze seznamu V opět přesahuje řetězec ze seznamu

88

U o jeden symbol 1, musí z obdobných důvodů $i_3 = i_4 = \bullet \bullet \bullet = 3$. Vidíme tedy, že existuje jen jedna možná posloupnost výběrů indexů, která generuje přípustné řetězce. Pro tuto posloupnost bude řetězec vytvořený ze seznamu V vždy o jeden symbol delší. Proto tento P KP nemá řešení.

Průvodce studiem:

Důkaz nerozhodnutelnosti problému PKP lze provést například prokázáním převeditelností HP ;IPKP ;PKP , kde problém IPKP je zadán obdobně jako PKP, jen otázka se ptá, zda existuje iniciální řešení pro danou instanci. Hlavní myšlenka převeditelnosti HP ; IPKP spočívá v následujícím: k danému M, w se sestrojí první členy seznamů $u_1 = \$$, $v_1 = \$q_0w\$$ (kde q_0 je počáteční stav M). Další dvojice se volí tak, aby jediná možná cesta k získání iniciálního řešení spočívala v určité simulaci výpočtu M na w s tím, že řešení existuje, právě když M se zastaví na w.

Převod IPKP

Důkaz: (IPKP ;PKP)

Pro provedení důkazu je zapotřebí zkonstruovat algoritmus převodu instance IPKP na instanci PKP , tak aby platilo, že IPKP má iniciální řešení právě tehdy, když PKP má libovolné řešení. Musíme tedy navrhnout postup, kterým ke každé instanci IPKP budeme schopni zkonstruovat instanci PKP . Nechť

$U = u_1, u_2, \ldots, u_k$ a $V = v_1, v_2, \ldots, v_k$

je zadání IPKP . Dále nechť Σ je abeceda obsahující všechny symboly vyskytující se v řetězcích seznamů U, V a nechť ¢ a $ nejsou obsaženy v Σ. Vytvořme x_i z u_i vložením symbolu ¢ za každý znak ve slově u_i, podobně vytvořme y_i z v_i vložením symbolu ¢ před každý znak ve slově v_i. Dále vytvořme nová slova

$x_0 = ¢\, x_1$, $y_0 = y_1$

$x_{k+1} = \$$, $y_{k+1} = ¢\, \$$

Nové seznamy

Nyní vytvoříme nové seznamy $X = x_0, x_1, \ldots, x_{k+1}$ a $Y = y_0, y_1, \ldots, y_{k+1}$, které budou vstupem pro PKP. Například pro seznamy U, V z předchozího příkladu dostaneme následující seznamy X, Y :

IP KP

	Seznam U	Seznam V
i	u_i	v_i
1	1	111
2	10111	10
3	10	0

P KP

	Seznam X	Seznam Y
i	x_i	y_i
0	¢1¢	¢1¢1¢1
1	1¢	¢1¢1¢1
2	1¢0¢1¢1¢1¢	¢1¢0
3	1¢0¢	¢0
4	$	¢$

Je zřejmé, že pokud má vytvořený PKP řešení, tak musí začínat dvojicí slov, s indexem 0, protože pouze tato jediná dvojice má první společný symbol ¢ .

Tato dvojice jistým způsobem koresponduje s prvními slovy z původního IPKP. Dále je vidět, že když najdeme řešení vytvořeného PKP se seznamy X, Y, tak budeme umět najít i řešení původního IPKP prostým vynecháním speciálních symbolů ¢ a $. Podařilo se nám tedy ukázat, že když existuje algoritmus rozhodující PKP, dokážeme vytvořit algoritmus pro rozhodování IPKP tím, že převedeme libovolnou instanci IPKP na PKP právě uvedeným způsobem.

Věta: Postův korespondenční problém je nerozhodnutelný.

Důkaz provedeme již dříve naznačeným způsobem tj. převodem HPĮPKPĮPKP . Vzhledem k tomu, že převod IPKPĮPKP jsme
již ukázali, stačí nyní prokázat převod HPĮPKP. Musíme tedy zkonstruovat algoritmus, který ke každé instanci HP (tedy Turingovu stroji M respektive jeho kódu Kod(M) a slovu w) sestrojí dva seznamy slov U, V , které budou vstupem pro IPKP , tak aby platilo, že vytvořená instance IPKP má řešení právě tehdy, když Turingův stroj M přijímá slovo w. Pro daný Turingův stroj a slovo zkonstruujeme instanci IPKP takovou, že když bude mítřešení, tak bude ve tvaru:

$\#q_0w\#\alpha_1q_1\beta_1\# \bullet \bullet \bullet \#\alpha_k\, q_k\, \beta_k\, \#,$

kde řetězce mezi po sobě jdoucími symboly # jsou po sobě jdoucí konfigurace Turingova stroje M se vstupem w a koncovým stavem q_k . Dostaneme tak vlastně zakódovanou celou posloupnost výpočtu Turingova stroje M nad slovem w od počáteční konfigurace q_0w až po některou koncovou konfiguraci $\alpha_k\, q_k\beta_k$ (pokud ovšem nějaký takový konečný výpočet existuje, pokud se daný Turingův M stroj na slovo w zacyklí, tak zjevně takto konstruovaný IP KP nebude mít řešení, což je přesně to, co jsme potřebovali). Formálně jsou dvojice řetězců vytvářejících seznamy U, V uvedeny níže. Kromě prvního páru, který musí být použit první, je pořadí ostatních párů nepodstatné a nijak neovlivňuje existenci řešení. Páry proto uvedeme bez indexů.

První vytvořený pár je:

Seznam U	Seznam V
#	#q_0w#

Zbývající páry můžeme následovně seskupit do skupin:

Skupina I

Seznam U	Seznam V	
X	X	
#	#	pro každé X ∈ Γ

Skupina II.

Pro všechny q ∈ Q \ F, p ∈ Q a X, Y, Z ∈ Γ:

Seznam U	Seznam V	
qX	Y p	jestliže $\delta(q, X) = (p, Y, R)$
ZqX	pZY	jestliže $\delta(q, X) = (p, Y, L)$
q#	Y p#	
Zq#	pZY#	jestliže $\delta(q, \#) = (p, Y, R)$

Skupina III. Pro všechna q ∈ F , a X, Y ∈ Γ:

Seznam U	Seznam V
XqY	q
Xq	q
qY	q

Skupina IV

Seznam U	Seznam V
q##	# pro každé q ∈ F

Úkoly k zamyšlení:

Promyslete si, že takto zkonstruovaná instance IP KP má řešení právě tehdy, když existuje konečný výpočet stroje M na slově w.
Celý postup si předvedeme na následujícím příkladu.

Příklad

Vezměme M = ({q_1, q_2, q_3}, {0, 1}, {0, 1}, δ, q_1, {q_3}), přičemž přechodová funkce δ je definována následovně:

q_i	$\delta(q_i, 0)$	$\delta(q_i, 1)$	$\delta(q_i, \#)$
q_1	$(q_2, 1, R)$	$(q_2, 0, L)$	$(q_2, 1, L)$
q_2	$(q_3, 0, L)$	$(q_1, 0, R)$	$(q_2, 0, R)$
q_3	—	—	—

Jako vstupní slovo použijeme w = 01. Zkonstruujeme instanci IPKP se seznamy U, V. První pár je # pro seznam U a #$q_1$01# pro seznam V. Ostatní páry jsou:

Skupina I

Seznam U	Seznam V
0	0
1	1
$\#$	$\#$

Skupina II

Seznam U	Seznam V	
$q_1 0$	$1 q_2$	z $\delta(q_1, 0) = (q_2, 1, R)$
$0 q_1 1$	$q_2 00$	z $\delta(q_1, 1) = (q_2, 0, L)$
$1 q_1 1$	$q_2 10$	
$0 q_1 \#$	$q_2 01 \#$	z $\delta(q_1, \#) = (q_2, 1, L)$
$1 q_1 \#$	$q_2 11 \#$	
$0 q_2 0$	$q_3 00$	z $\delta(q_2, 0) = (q_3, 0, L)$
$1 q_2 0$	$q_3 10$	
$q_2 1$	$0 q_1$	z $\delta(q_2, 1) = (q_1, 0, R)$
$q_2 \#$	$0 q_2 \#$	z $\delta(q_2, \#) = (q_2, 0, R)$

Skupina III

Seznam U	Seznam V
$0q_30$	q_3
$0q_31$	q_3
$1q_30$	q_3
$1q_31$	q_3
$0q_3$	q_3
$1q_3$	q_3
q_30	q_3
q_31	q_3

Skupina IV

Seznam U	Seznam V
$q_3\#\#$	$\#$

Poznamenejme, že M přijímá vstupní slovo w = 01 posloupností konfigurací:

$$q_101, 1q_21, 10q_1, 1q_201, q_3101$$

Podívejme se, jestli existuje řešení IPKP, který jsme právě zkonstruovali. První pár dává částečné řešení (#, #$q_1$01#). Po bližším prozkoumání dvojic řetězců vidíme, že jediná cesta, jak získat delší částečné řešení, je použít jako další pár ($q_1$0, 1q_2), což odpovídá přechodu Turingova stroje ze stavu q_1 do stavu q_2. Výsledné částečné řešení je (#$q_1$0, #$q_1$01#1q_2). Část, která nyní nadbývá v druhém řetězci je 1#1q_2. Další tři použité páry musí být (1, 1), (#, #), (1, 1). Částečné řešení pak bude (#$q_1$01#1, #$q_1$01#1$q_2$1#1).

Přebytek je nyní $q_2 1\#1$. Dalším postupem dojdeme až k částečnému řešení (y, y1#$q_3$10), kde
y = #$q_1$01#1$q_2$1#10q_1#1$q_2$0

Protože q_3 je koncový stav, můžeme nyní použít páry ze skupin I, III a IV k nalezení řešení instance IPKP . Výběr párů je

(1, 1), (#, #), ($q_3$1, q_3), (0, 0), (1, 1), (#, #), ($q_3$0, q_3)
(1, 1), (#, #), ($q_3$1, q_3), (#, #), (q_3##, #).

Tedy nejkratší slovo, které může být vytvořeno odpovídajícími řetězci ze seznamů U, V začínající prvním párem je

#$q_1$01#1$q_2$1#10q_1#1$q_2$01#$q_3$101#$q_3$01#$q_3$1#q_3##

4.4 Další nerozhodnutelné problémy

Problém IBKJ (Neprázdného průniku dvou BKJ)

Instance: Dvě bezkontextové gramatiky G_1, G_2

Otázka: Platí $L(G_1) \cap L(G_2) \neq \emptyset$? (Tzn. ‘Lze nějaké slovo vygenerovat oběma gramatikami?’)

Věta: Problém neprázdného průniku dvou bezkontextových jazyků je nerozhodnutelný.

Důkaz provedeme převodem PKPĮBKJ . K zadané instanci PKP

$U = u_1, u_2, \ldots, u_n$ a $V = v_1, v_2, \ldots, v_n$

sestrojíme instanci problému IBKJ , tj. dvě bezkontextové gramatiky G_1 a G_2. Budeme požadovat, aby tyto nově zkonstruované gramatiky generovaly jazyky s neprázdným průnikem právě tehdy, když PKP bude mít řešení. Předpokládejme, že nově zavedené symboly $a_1, a_2, \ldots, a_n$ se nevyskytují v žádném z řetězců seznamů PKP . Potom bezkontextové gramatiky G_1 a G_2 můžeme navrhnout následovně:

G_1 : $S \rightarrow u_1Sa_1| \bullet\bullet\bullet |u_nSa_n|\varepsilon$ G_2 : $S \rightarrow v_1Sa_1| \bullet\bullet\bullet |v_nSa_n|\varepsilon$

Nově zavedené symboly $a_1, a_2, \ldots, a_n$ zajišťují, aby pro potenciální shodná slova generovaná oběma gramatikami musela existovat v obou gramatikách odvození složená právě ze stejných posloupností výběru pravidel. Tedy aby byla zajištěna korespondence odpovídajících párů slov ze seznamů instance P KP .

Velmi podobně se dá ukázat nerozhodnutelnost následujícího problému:

Problém ABKG (Nejednoznačnosti BKG)

Instance: Bezkontextová gramatika G

Otázka: Je zadaná gramatika nejednoznačná? (Tzn. ‘Lze nějaké slovo vygenerovat dvěmi různými odvozeními (derivacemi) ?’)

Další nerozhodutelné problémy týkající se bezkontextových jazyků, které vyplývají přímo z předchozích jsou například otázky, zda ‘$L(G) = \Sigma^*$?’ nebo
zda ‘$L(G_1) = L(G_2)$?’ apod.

Shrnutí:

- Některé problémy, ač je jejich zadání a formulace poměrně jednoduchá
jsou nerozhodnutelné. V praxi to znamená, že ať bychom se snažili sebevíc, tak se nám nikdy nemůže podařit zkonstruovat algoritmus nebo nějaký počítačový program, který by dokázal na všechny přípustné instance daného problému dát správnou odpověď.

- Základním takovým nerozhodnutelným problémem je otázka, zda se zadaný Turingův stroj M zastaví, či nezastaví na určité vstupní slovo w. Tento problém není rozhodnutelný, ale je částečně rozhodnutelný. Nejpříhodněji nám v této situaci poslouží Univerzální Turingův stroj zmíněný v předchozí kapitole, který pomocí simulace chodu zadaného stroje M je schopen říct, že se výpočet stroje M na slově w zastavil a dokonce nám i sdělí výsledek výpočtu. Na druhou stranu, když se M na w nezastaví, tak i samotná simulace nebude mít konce a tudíž se kýžené odpovědi nikdy nedočkáme.

- Nějaký problém je převeditelný na jiný problém, jestliže jsme schopni navrhnout obecný algoritmus převodu instance prvního problému na instanci druhého problému, tak aby se zpětně dalo usuzovat na řešení převáděného problému, jinými slovy, aby odpovědi na otázky kladené u jednotlivých problémů byly shodné.

- Když máme jeden zaručeně nerozhodnutelný problém, můžeme pomocí něj prokázat nerozhodnutelnost celé řady problémů, na které budeme schopni tento problém převést. Kdyby totiž tyto problémy byly řešitelné, měli bychom v podstatě postup, jak řešit i původní neřešitelný problém.

- Další problém, o kterém můžeme s jistotou tvrdit, že je nerozhodnutelný, je tzv. Postův korespondenční problém, u kterého jde o

nalezení posloupnosti párů slov ze dvou seznamů takové, aby po zřetězení těchto slov vznikly shodné řetězce.

- Z oblasti bezkontextových jazyků pak máme například nerozhodnutelný problém neprázdného průniku dvou bezkontextových gramatik, případně problém víceznačnosti bezkontextové gramatiky, jejichž ne- rozhodnutelnost se snadno prokáže převedením PKP na odpovídající problém.

Kontrolní otázky:

1. Ukažte, že problém zastavení je částečně rozhodnutelný.

Cvičení:

1. Navrhněte Turingův stroj, který částečně rozhoduje jazyk

$L = \{a^i \mid i = 2k, k \in N\}$ a na tomto stroji demonstrujte důkaz převodu HP na IPKP .

Pojmy k zapamatování:

- problém zastavení
- algoritmická převeditelnost problémů
- Postův korespondenční problém
- iniciální Postův korespondenční problém

5 Enumerace Turingových strojů

Cíl:

Nyní se zaměříme na jazyky, které nejsou ani částečně rozhodnutelné, k čemuž nám dopomůže jisté seřazení resp. enumerace Turingových strojů. Po prostudování této kapitoly budete:

- znát postup, pomocí kterého je možno enumerovat všechny Turingovy stroje,
- umět nad libovolnou abecedou zkonstruovat jazyk, který není ani částečně rozhodnutelný,
- schopni pomocí Riceovy věty o celé řadě problémů schopni prohlásit, zda jsou nerozhodnutelné.

Průvodce studiem

Další otázka, která je pro nás důležitá z abstraktního pohledu na rozhodnutelnost je tzv. enumerace Turongových strojů. V podstatě tím stroje "indexujeme" a můžeme tak za každý strojem vidět jeho konkrétní „číslo". Formulujeme také důležitou větu tzv. Riceovu, která je aplikovatelná nejen na TS, ale lze ji přeformulovat pro libovolný jiný ekvivaletní výpočetní model.

Enumerace Turingových strojů

Všimněme si nyní, že všechny Turingovy stroje (s abecedou {0, 1}) lze přirozeně seřadit (očíslovat, enumerovat). Už jsme si uvědomili, že Turingovy
stroje (s abecedou {0, 1}) lze kódovat řetězci nul a jedniček. Když si uvědomíme, že pro libovolnou konečnou abecedu Σ (tedy i pro Σ = {0, 1}) je množina Σ^* nekonečná spočetná (řetězce lze např. uspořádat pomocí rostoucího
uspořádání, které bylo zmíněno v definici 4), je ihned jasné, že i Turingových strojů je nejvýše spočetně – samozřejmě ovšem nekonečně spočetně.
Navíc nám zmíněné uspořádání řetězců z $\{0, 1\}^*$ automaticky dává přirozené uspořádání Turingových strojů (podle jejich kódů v abecedě {0, 1}):

je tedy možné hovořit o enumeraci $M_0, M_1, M_2, \ldots$ Turingových strojů (či jejich kódů). Navíc příslušné zobrazení $N \to \{w \in \{0, 1\}^* \mid w = Kod(M)$ pro nějaký Turingův stroj $M\}$ je bijekce, která je (obousměrně) algoritmicky vyčíslitelná.

Úkoly k zamyšlení:

Zamysleme se na chvíli nad otázkou, zda existují jazyky (v abecedě {0, 1}), které nejsou částečně rozhodnutelné. Dříve uvedená Postova věta

spolu s faktem, že (jazyk) HP (nebo DHP) je částečně rozhodnutelný a není rozhodnutelný, už poskytuje odpověď: doplněk jazyka HP (či DHP) není částečně rozhodnutelný.

5.1 Aplikace Cantorovy věty

Zmíněný fakt, že existují i problémy (jazyky), které nejsou ani částečně rozhodnutelné je možné ukázat i jinou cestou. Nejprve však budeme potřebovat následující větu:

Věta: (Cantorova)
Pro libovolnou konečnou i nekonečnou množinu M platí:

$|M| < |P(M)|$

kde P(M) značí tzv. potenční množinu - množinu všech podmnožin M. Jinými slovy lze Cantorovu větu formulovat tak, že počet prvků množiny M je ostře menší než počet všech jejích podmnožin.

Důkaz Cantorovy věty můžete najít téměř v každé knize zabývající se teorií množin.

Jelikož Turingových strojů je spočetně mnoho, je také částečně rozhodnutelných jazyků (nejvýše, ovšem samozřejmě právě) spočetně mnoho. Díky právě uvedené obecné Cantorově větě, je zřejmé, že množina všech jazyků
$\{L \mid L \subseteq \{0, 1\}^*\}$ je nespočetná. Z toho vyplývá, že lze vytvořit více jazyků,

Turingových strojů než kolik existuje Turingových strojů. Proto musí existovat jazyky, které nepatří do třídy jazyků typu 0 .

Je ilustrativní neodvolávat se na obecnou větu, ale provést přímý důkaz tzv. Cantorovou diagonalizační metodou; tato metoda je totiž v oblasti vyčíslitelnosti a složitosti velmi užitečná.

Věta: Pro libovolnou neprázdnou abecedu Σ existují jazyky $L \subseteq \Sigma^*$, které nejsou částečně rozhodnutelné.

Důkaz: (Ilustrace Cantorovy diagonalizační metody).

Pro libovolnou (např. výše uvedenou) enumeraci (všech) Turingových strojů $M_0, M_1, M_2, \ldots$ a pro libovolné uspořádání (všech) slov $w_0, w_1, w_2, \ldots$ v abecedě Σ (např. rostoucí uspořádání.) definujme jazyk L následovně:

pro každé i slovo w_i patří do jazyka L právě tehdy, když stroj Mi slovo w_i nepřijímá (nezastaví se na něj). Tedy $L = \{w_i \mid \neg!M_i(w_i)\}$. Je zřejmé, že L není přijímán žádným ze strojů M_i.

5.2 Riceova věta

Nyní si uvedeme důležitou, tzv. Riceovu, větu, která zahrnuje celou třídu nerozhodnutelných problémů. Vyslovíme ji nejdříve v ‘obšírnějším’ znění:

Jakákoli netriviální vlastnost Turingových strojů týkající se výhradně jejich vstupně/výstupního chování (tzn. každé dva Turingovy stroje, které realizují totéž zobrazení, buď oba vlastnost mají nebo oba vlastnost nemají; netrivialita spočívá v tom, že existuje Turingův stroj, jenž vlastnost má a existuje Turingův stroj, jenž ji nemá) je nerozhodnutelná (tj. množina všech kódů (indexů) Turingových strojů s danou vlastností) je nerozhodnutelná.

Níže vyslovíme totéž v elegantnější podobě. Připomeňme si nejprve naši enumeraci M_0, M_1, M_2, . . .; o čísle i budeme hovořit jako o indexu Turingova stroje M_i. Dále připomeňme, že každý M_i realizuje (částečné) zobrazení
$\{0, 1\}^* \rightarrow \{0, 1\}^*$. Dále si ještě uvědomme, že pojem rozhodnutelnosti (jako

i částečné rozhodnutelnosti apod.) máme vlastně definován i pro množiny přirozených čísel – množinu N totiž můžeme např. velmi přirozeně ztotožnit
s množinou řetězců nad jednoprvkovou abecedou.

Věta: (Rice)
Nechť A je nějaká množina algoritmicky vyčíslitelných (částečných) zobrazení typu $\{0, 1\}^* \rightarrow \{0, 1\}^*$. Potom množina $B = \{i \in N \mid M_i \in A$ (tzn. M_i realizuje zobrazení patřící do A)$\}$ je rozhodnutelná právě když $B = \emptyset$ nebo $B = N$.

Důkaz: Vezměme nějakou takovou A, která není prázdná ani nezahrnuje všechny algoritmicky vyčíslitelné funkce. Nechť nikde nedefinované zobrazení (tzn. zobrazení, které má pro každý vstup nedefinovaný výstup a tudíž Turingův stroj, který jej realizuje se pokaždé zacyklí a nezastaví se) $\perp : \{0, 1\}^* \rightarrow \{0, 1\}^*$ nepatří do A (opačný případ se řeší podobně). Nechť
M_{i0} realizuje $\perp$, tedy $_{i0} \notin B$ a nechť M_{j0} realizuje zobrazení z A (nutně takový existuje); tedy $j_0 \in B$.

Ukážeme, že problém DHP je převeditelný na B (tj. na problém příslušnosti k B), čímž prokážeme nerozhodnutelnost B. Algoritmus PREV převodu DHPƀ pracuje následovně:

K danému stroji (kódu stroje) M (tj. k instanci problému DHP) algoritmus P REV nejdříve sestaví stroj M´, který je ‘naprogramován’ tak, že jeho činnost je následovná:
M´ nejprve vpravo vedle svého vstupu (na kterém v této chvíli nezáleží) zapíše slovo Kod(M) a na něj spustí (podprogram) M. Pokud tento (pod)výpočet skončí, smaže M´ případný zbytek po tomto výpočtu, najede na původně daný vstup a spustí na něj M_{j0}. Po sestavení tohoto M´ spočte PREV jeho index a ten vydá.

Je zřejmé, že když M se zastaví na Kod(M) (tj. odpověď na onu instanci DHP je ANO), realizuje M´ totéž zobrazení jako M_{j0} a jeho index tedy patří do B. Když se M nezastaví na Kod(M) (odpověď v DHP je NE), realizuje
M´ zobrazení $\perp$, tedy totéž jako M_{i0} a jeho index tedy do B nepatří.

Shrnutí:

- Jakmile jsme schopni zakódovat libovolný Turingův stroj do předem domluveného formátu řetězce symbolů, můžeme hovořit o jisté enumeraci neboli seřazení všech Turingových strojů. Tímto jsme schopni zjistit počet Turingových strojů, který je zjevně shodný s počtem přirozených čísel. Jinými slovy množina všech Turingových strojů je spočetná.

- Na druhou stranu množina všech jazyků je podle Cantorovy věty nespočetná, což nám dává jednoduchý výsledek, že jazyků je více než Turingových strojů. Musí tedy existovat jazyky, které nejsou přijímány žádnými Turingovy stroji. Vzhledem k tomu, že Turingovy stroje přijímají nejobecnější jazyky generované neomezenými gramatikami (jazyky typu 0), bude se jednat o jazyky bez gramatického základu.

- Jeden z příkladů jazyků tohoto typu je jazyk příslušný k doplňku DHP , tedy jazyk tvořený kódy Turingových strojů, které se nezastaví na svůj vlastní kód. Další pěkný příklad jazyka nepřijímaného žádným Turingovým strojem dostaneme použitím Cantorovy diagonalizační metody, k čemuž využijeme výše zmíněnou enumeraci Turingových strojů a slov nad jejich vstupní abecedou.

- Riceova věta nám určuje celou třídu nerozhodnutelných problémů v zá- visloSti na čistě vstupně/výstupním charakteru chování Turingových strojů. Například z ní přímo plyne, že je nerozhodnutelné určit, zda daný stroj realizuje konkrétní zobrazení.

Kontrolní otázky:

1. Objasněte princip enumerace Turingových strojů.

2. Pomocí Cantorovy věty ukažte, že existují jazyky, které nejsou částečně rozhodnutelné.

3. Ukažte princip Riceovy věty na praktickém příkladu.

Pojmy k zapamatování:

- enumerace Turingových strojů
- Cantorova diagonalizační metoda
- Riceova věta

6 Model RAM (Random Access Machine)

Cíl:

Po prostudování této kapitoly pochopíte:

- Princip modelu RAM (Random Access Machine)

Naučíte se:

- vytvářet RAM stroje a analyzovat jejich složitost

Klíčová slova této kapitoly:

RAM stroj, Random Access Machine, registr, program.

Průvodce studiem

Turingův stroj je sice geniálně jednoduchou a silnou formalizací, přesto může na techničtěji orientované jedince působit příliš abstraktně či těžkopádně. Zvláště programátor je zvyklý pracovat na vyšší úrovni než nabízí model TS a to nejméně na principu procesorů (dnes zdaleka už neplatí ani to – vyšší programovací jazyky). Ty neobsahují paměťové buňky se znakem abecedy, ale s čísly a mezi přímo použitelné instrukce patří řada aritmetických funkcí, které by se museli TS složitě implementovat (stačí si vzpomenout na příklad s inkrementací čísla).

Model RAM (Random Access Machine)

Dalším univerzálním výpočetním modelem je RAM (random access memory) stroj, tedy stroj s náhodným přístupem k paměti, který se svou podstatou snaží napodobit práci reálného procesoru.

Ačkoliv Turingův stroj a RAM stroj odděluje časově pár desítek let, jejich výpočetní síla je ekvivalentní a dá se to díky algebraickým prostředkům i dokázat. Za podobné rysy můžeme považovat např. práci s vstupními/výstupními páskami pomocí čtecích/zapisovacích hlavách. V čem se však tyto stroje liší? To nám osvětlí následující tabulka:

	TURINGŮV STROJ	RAM STROJ
Zpracovávaný vstup	symboly vstupní a páskové abecedy	celá čísla
Pásky	libovolný počet a užití	jedna vstupní a jedna výstupní
Provádění výpočtu	procházení stavového prostoru pomocí přechodové funkce	provádění instrukcí uložených v programové jednotce
Druh paměti	pomocí pásek; sekvenčně	pracovní paměť; libovolný přístup
Aritmeticko-logické funkce	není implementováno	integrovaná aritmeticko-logická jednotka
Pohyb hlav	doprava i doleva (na základě přechodové funkce)	sekvenčně směrem doprava (při použití instrukce pro čtení nebo zápis)

6.1 Prvky RAM stroje

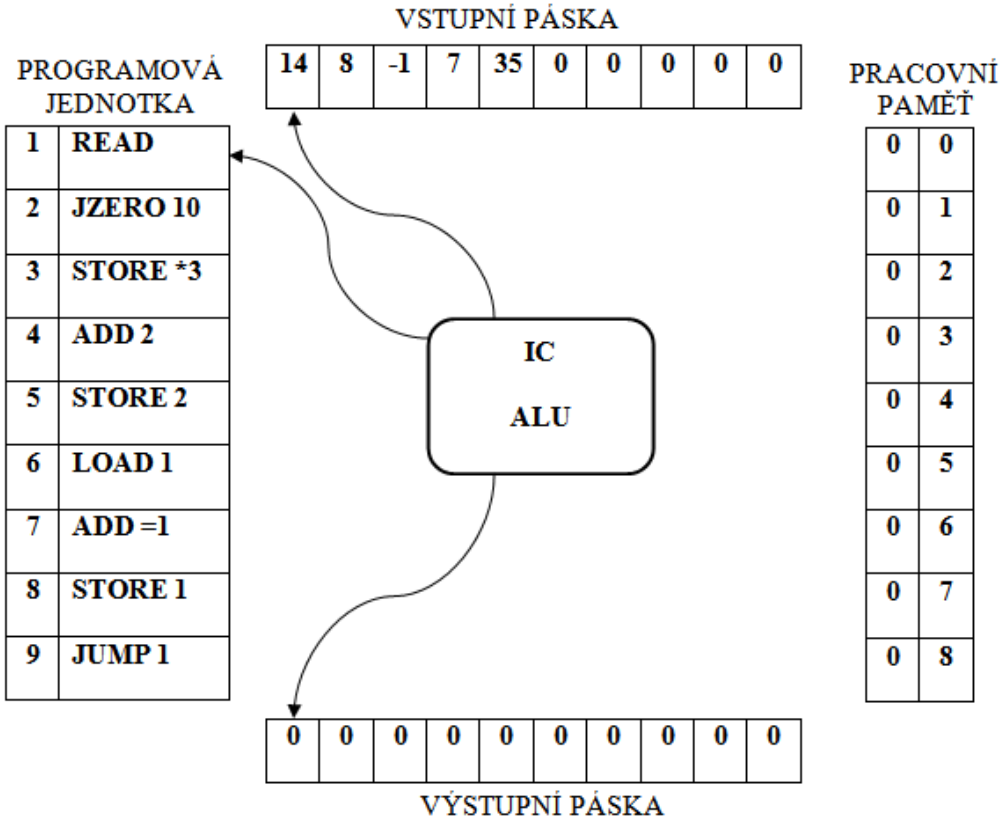
Celkově se RAM stroj skládá z několika části, které si nyní popíšeme:

- **programová jednotka** - zde je uložen program, tvořený konečnou posloupnosti instrukci (příkazů). Každý konkrétní příklad je reprezentovat jako tento program.

Random Access Machine

- **neomezená pracovní paměť** - je tvořena buňkami, kde každá buňka může obsahovat libovolné celé číslo. Buňky jsou očíslovány přirozenými čísly 0, 1,..., n. Číslo buňky se nazývá adresa buňky. Do buněk je možno zapisovat i z nich číst.
- **vstupní páska** - tvořena buňkami (políčky), kde každá buňka obsahuje jedno celé číslo. Z této pásky je možno pouze sekvenčně číst. Na aktuálním políčku stojí (čtecí) hlava. Základní krok v činnosti hlavy spočívá v přečtení obsahu snímaného políčka a posunutí doprava o jedno políčko.
- **výstupní páska** - do jejich buněk se zapisuji celá čísla. Na tuto pásku je pouze možné sekvenčně zapisovat (pomoci zapisovací hlavy).
- **centrální jednotka** - obsahuje programový registr (instruction counter, IC) ukazující, která instrukce má být v daném okamžiku prováděna (programový registr prostě obsahuje pořadové číslo příslušné instrukce). Tato instrukce se provede a programový registr se příslušně změní (např. se zvýší o 1 či se změní jinak v případě skoku). Podrobný popis jednotlivých instrukcí bude popsán níže. Další součástí centrální jednotky je aritmeticko-logická jednotka (Arithmetic Logic Unit, ALU), umožňující některé aritmetické a logické operace.

Na obrázku níže můžeme vidět jednotlivé části RAM stroje a jejich vazby.



Počáteční nastavení

- **programová jednotka** - obsahuje posloupnost instrukcí, které mají své pořadové číslo. Výpočet začíná první instrukcí a končí instrukcí HALT (viz níže).

- **vstupní páska** - prvních *n* čísel obsahuje vstupní data určena ke zpracování, následující nekonečnou sekvencí znaku nula jako výchozí znak. Pokud vstupní data také obsahují nulu, program nepozná, zda se jedná o součást vstupních dat nebo první znak za těmito daty.
- **výstupní páska, pracovní paměť** - všechny buňky mají výchozí hodnotu nulu

Nejdřív si definujeme, jakým způsobem mohou být zapsány operandy.

tvar	**hodnota operandu**
= i	číslo přímo udané zápisem *i*
i	číslo, které se nalézá v buňce s adresou *i*
***i**	číslo v buňce s adresou *i+j*, kde *j* je aktuální obsah indexového registru

Instrukce vstupu a výstupu

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

Instrukce přesunu v paměti

zápis	význam
LOAD operand	do pracovního registru se načte hodnota operandu
STORE operand	hodnota operandu se přepíše obsahem pracovního registru (zde se nepřipouští operand tvaru =i, který představuje konstantu)

Instrukce aritmetických operaci

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

Instrukce skoku

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

Instrukce zastavení

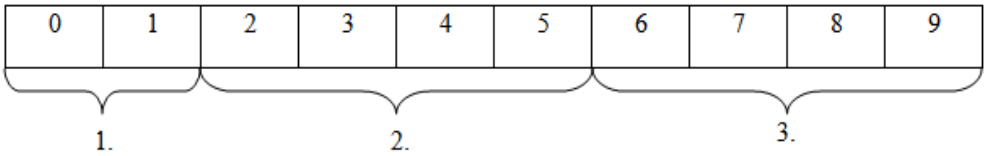
zápis	význam
HALT	výpočet je regulérně ukončen

U RAM stroje je nutné si uvědomit, že postup není řešen pomocí stavů jako u Turingova stroje, ale sekvencí instrukcí, které se zpracovávají buď sekvenčně nebo pomocí skoků. Skoky nám nejvíce pomáhají s problematikou cyklů, kdy se potřebujeme vrátit na konkrétní místo počátku cyklu po jeho průchodu.

Než začneme psát samotné instrukce, je třeba si rozvrhnout jednotlivé oblasti programu.

Vývojový diagram nám ukáže jednotlivé oblasti, cykly a skoky.

Druhým nejdůležitějším úkolem analýzy je zjištění, jaké hodnoty budeme potřebovat uchovat za běhu programu. Na obrázku jsou znázorněny paměťové buňky s jejich indexy.



Paměť RAM stroje si můžeme rozdělit na 3 části:

1. funkčnost nultého a první indexu je pro každý program stejná. Buňka s nultým indexem slouží jako pracovní registr, přes který se provádí veškeré operace. Buňka s indexem 1 slouží jako indexový registr, díky kterému se můžeme pohybovat v datech. Tento pohyb je sekvenční, a tak více než pole (s indexy) nám tato konstrukce slouží jako jednostranný seznam.
2. druhá část je specifická každému programu. Do této části si ukládáme důležité proměnné, např. pomocná proměnná, součet, minimum/maximum, atd.. Díky libovolnému přístupu si můžeme na tyto hodnoty sáhnout kdykoliv potřebujeme.
3. do třetí části se ukládají samotná data, které máme na vstupu. První index této části si můžeme představit jako odkaz na počátek konstrukce seznam (viz výše).

- z důvodu sekvenčního přístupu na vstupní a výstupní pásce jsou READ/WRITE operace zařazeny téměř vždy na začátek (READ) a konec (WRITE),
- jediný způsob, jak změnit hodnotu kterékoliv buňky v paměti (vyjma nultého pracovního registru), je pomocí instrukce STORE. Takto víme, že pokud načteme nějakou hodnotu, můžeme s ní libovolně manipulovat v pracovním registru a přesto původní hodnota zůstává uložena na původním místě až do vykonání instrukce STORE. Stejně tak je jasné, že pokud chceme nějakou hodnotu změnit, budeme potřebovat minimálně 3 instrukce, a to LOAD (načtení hodnoty), libovolná aritmetická funkce (např. ADD), a nakonec STORE (přepsání hodnoty),
- použití operandu *i indikuje odkazování do třetí části pracovní paměti, která má proměnlivou délku na základě vstupních hodnot, a proto jsme schopni se trvale odkazovat pouze na první buňku za druhou částí pracovní paměti

6.2 Aplikace RAM

Bubble sort (bublinkové řazení) si nyní ukážeme detailně zpracovaný pomocí RAM. Algoritmus obsahuje vnější a vnitřní cyklus. Ten vnitřní prochází vždy z jedné strany na druhou a porovnává každé dva sousedící prvky, zda jsou ve správném pořadí a pokud ne, prohodí je. V případě vzestupného řazení se kontroluje, zda je hodnota pravého prvku větší než hodnota levého prvku. Pokud tedy procházíme posloupnost zleva, tak nám postupně "probublává" největší hodnota až k pravému konci. Tento (vnitřní) cyklus opakujeme pomocí vnějšího cyklu, pokaždé však máme o jeden prvek méně, protože na pravé straně se nám postupně staví seřazené hodnoty. Pokud máme n hodnot, pak první průchod vnějším cyklem prochází n prvků, druhý průchod n-1 prvků, atd.

Mějme neseřazenou posloupnost čísel 2, 7, 5, 3, 1. Počet prvků je tedy n=5, seřazujeme vzestupně. Porovnávané dvojice jsou označeny červeně a již seřazené hodnoty modře. Nejdříve si ukážeme první průchod vnějším cyklem:

1.průchod: 27531→27531→25731→25371→25317

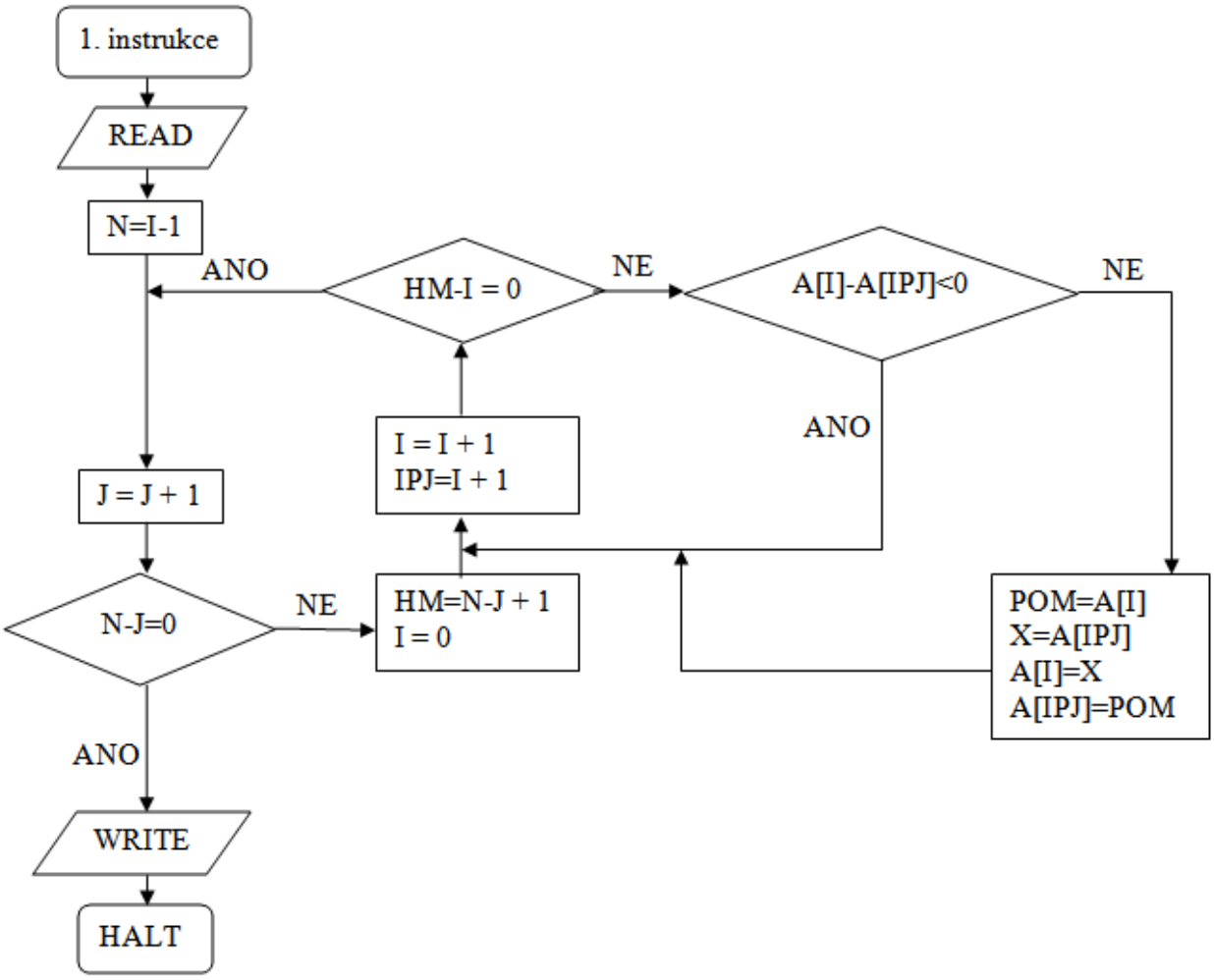
Vidíme, že až na první porovnání jsme vždy museli hodnoty přehodit a doprava se postupně přesunula hodnota 7. V dalším průchodu porovnáváme n-1 p
n-1, stejně tak p
průchodu.

2.průchod: 25317→25317→23517→23157

3.průchod: 23157→23157→21357

4.průchod: 21357→12357

Model RAM (Random Access Machine)



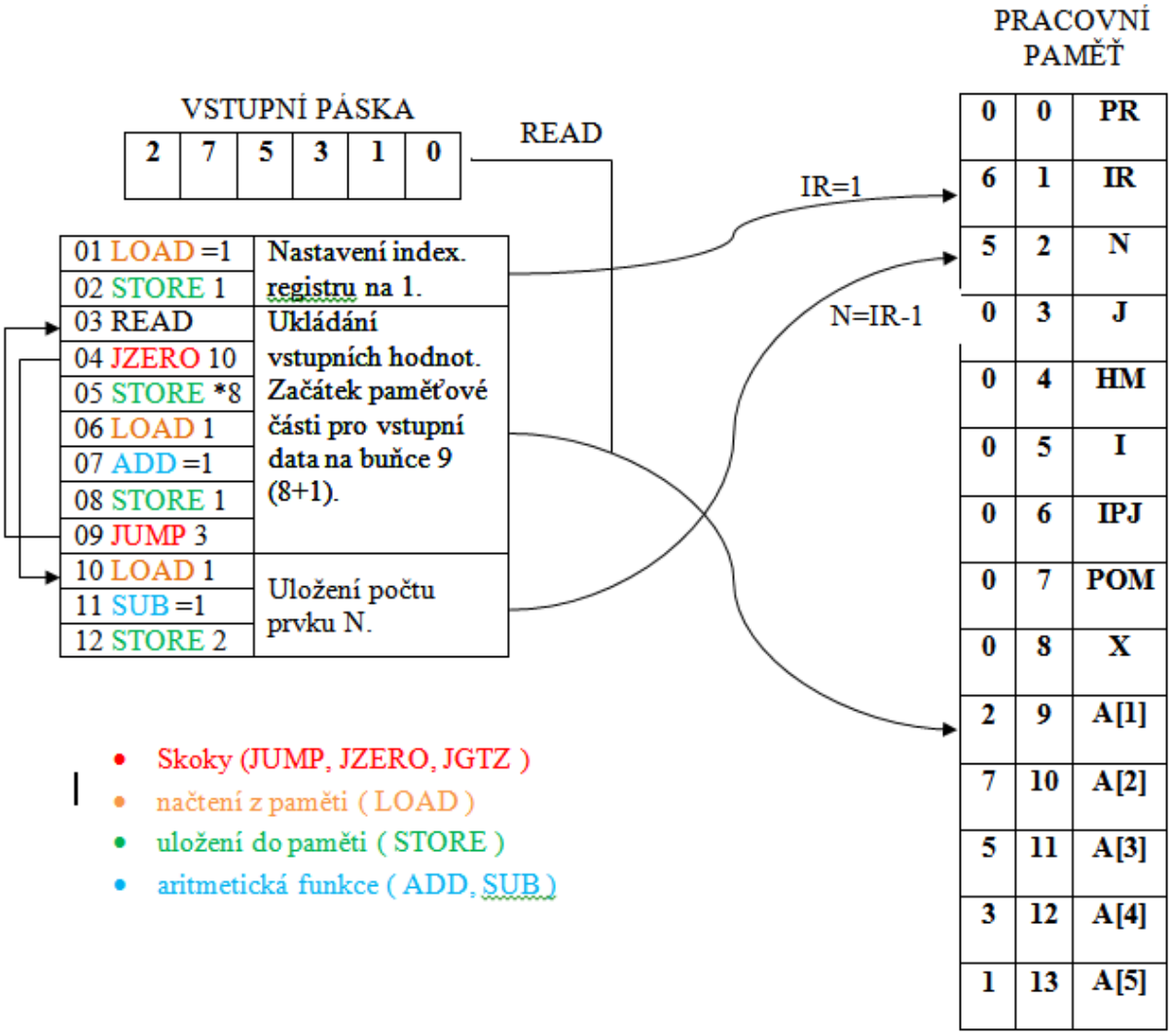
Obrázek obsahuje názvy proměnných, jejichž význam si nyní vysvětlíme a přiřadíme jim konkrétní paměťové buňky v paměti:

Paměťové buňky

2) N - počet prvků
3) J - indexování vnějšího cyklu
4) HM - Horní mez (počet neseřazených prvků uvnitř vnitřního cyklu)
5) I - indexování vnitřního cyklu

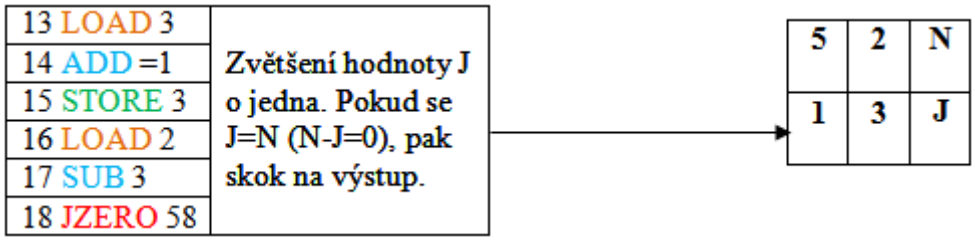
6) IPJ(I+1) - index následujícího prvku
7) POM - pomocná proměnná při prohození
8) X - proměnná pro uložení hodnoty
9) Pole A s uloženými daty

Nyní si rozebereme celý algoritmus po menších částech. První část bude pro názornost obsahovat celý výpis pracovní paměti, další části budou už mít jen podstatný zlomek.

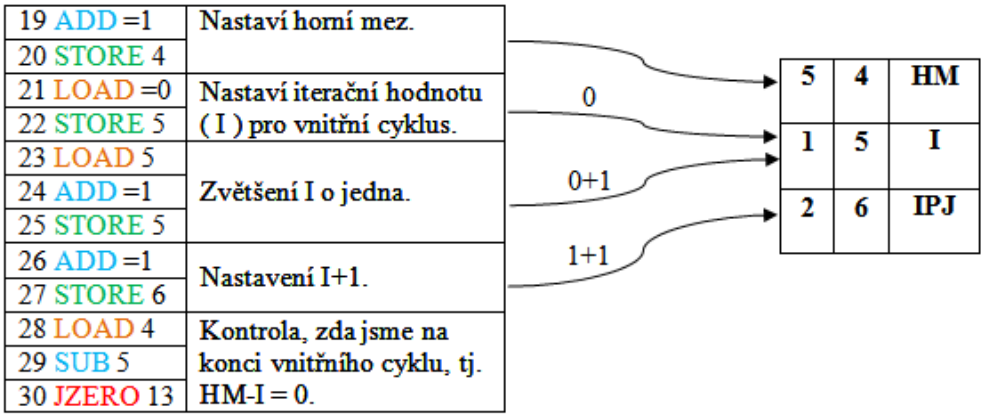


První část je jednoduchá. Nejdříve si připravíme indexový registr pro cyklus načítání vstupu. Pak postupně načítáme hodnoty. Pokud je na vstupu hodnota 0, provedeme skok. Nakonec využijeme hodnoty v index.

registru k uložení hodnoty počtu prvků. Tuto hodnotu musíme snížit o jedna, neboť při posledním průchodu cyklem se hodnota zvýšila na 6.

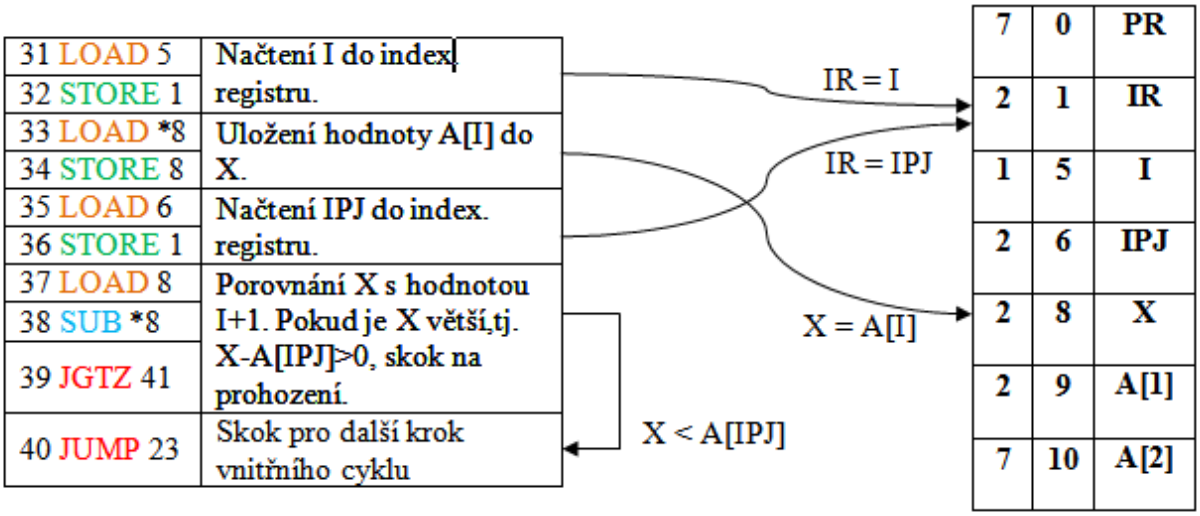


V této části začíná a zároveň končí samotný řadící algoritmus. Jedná se o nastavení a kontrolu iterační hodnoty vnějšího cyklu značeného jako J. Pokud je J stejné jako počet prvků N, pak víme, že se provedlo n-1 průchodů vnějšího cyklu, a proto je jisté, že posloupnost je již seřazena.



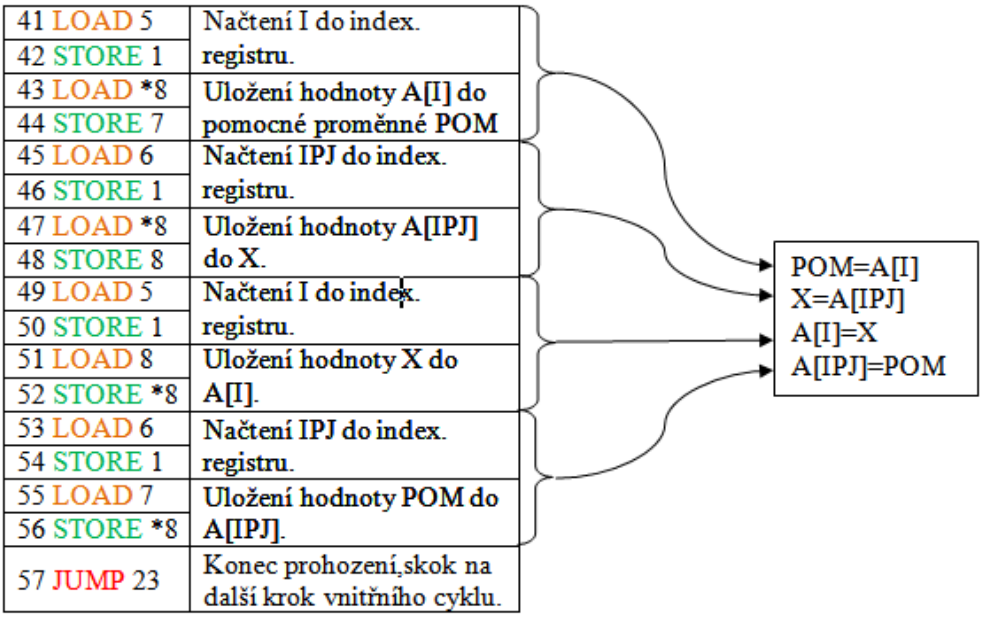
Z předchozí části nám v pracovním registru zůstal rozdíl N-J, při prvním průchodu 5-1=4, což značí 4 potřebné průchody k dokončení. Přičtením 1 dostaneme hodnotu počet prvků pro následující průchod nazvanou horní mez. Jelikož začínáme nový vnitřní cyklus, je třeba nastavit a uložit si jeho iterační hodnotu. Od instrukce č.23 už pokračujeme ve vnitřním cyklu,

navýšením hodnoty I a IPJ o jedna. Hodnota I tedy začíná jedničkou a při kontrole na instrukci č.30 nám opět vyjde výsledek 4, což je tentokrát počet porovnání sousedících dvojic. Po dokončení těchto porovnání se vracíme k instrukci č.23 (viz níže).

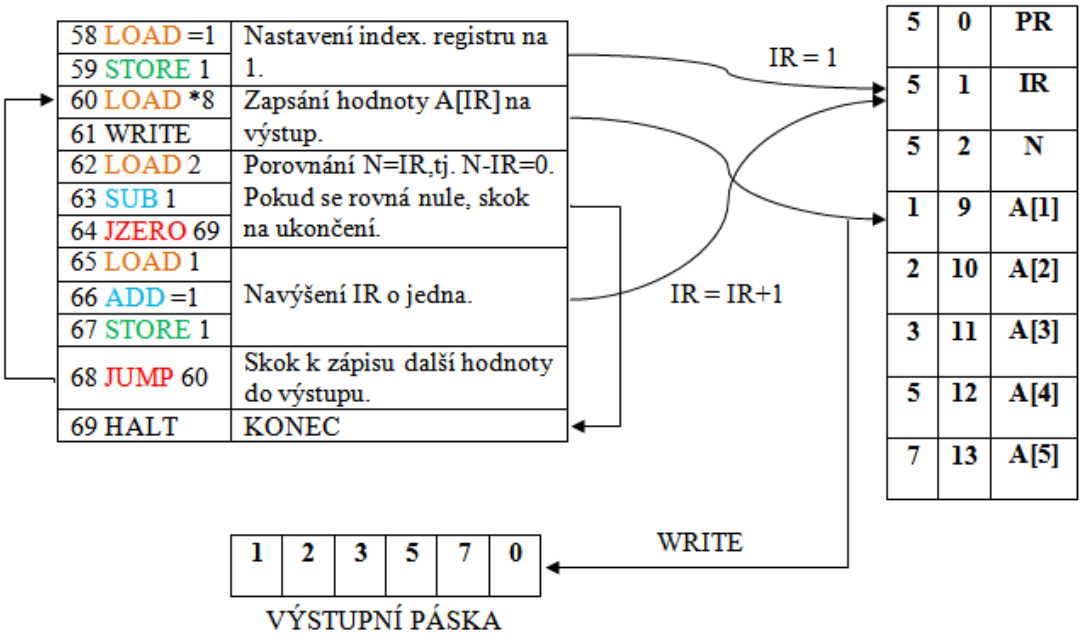


Nyní jsme už v samém srdci algoritmu, kde se porovnávají sousedící dvojice hodnot. Nejsme schopni se odkazovat do dvou míst třetí části pracovní paměti zároveň, protože indexový registr je schopný pojmout pouze jednu iterační hodnotu, proto si načteme první hodnotu do hodnoty X. Poté stačí načít iteraci následujícího prvku (IPJ) a teď už máme možnost porovnat oba prvky. V prvním případě porovnání nám vychází, že 2 a 7 jsou ve správném pořadí, proto přecházíme k další dvojici. V

opačném případě bychom provedli skok instrukce č. 39 na prohazovací část algoritmu, která je popsána na obrázku.



Proces prohození je složitější pro naše klasické zobrazení, proto jsem využil popis tohoto procesu z vývojového diagramu, který tento postup dostatečně vysvětluje. Opět musíme pracovat s dvěma hodnotami zároveň, proto se musí náležitě prohazovat hodnota iterací mezi prvním a druhým prvkem dané dvojice. Na konci této části si můžeme povšimnout skoku, který směřuje na stejné místo jako skok instrukce č.40 v předchozí části.



Poslední část zajišťuje zápis seřazených hodnot na výstupní pásku po ukončení řadícího algoritmu. Nejdříve nastavujeme iterační hodnotu pro cyklus zápisu. Poté zapisujeme hodnoty na výstupní pásku. Pokud jsme zapsali všechny hodnoty, podmínka na instrukci se vyhodnotí kladně a provedeme skok na poslední instrukci (HALT), která zastaví program.

Nejdůležitější probrané pojmy:

- RAM
- Vstupní a výstupní páska

Model RAM (Random Access Machine)

- Paměťová jednotka
- ALU a IC

7 Rekurzivní funkce

Cíl: Po prostudování této kapitoly pochopíte:

- Princip modelu rekurzivních funkcí

Naučíte se:

- vytvářet rekurzivní funkce

Klíčová slova této kapitoly:

Rekurzivní funkce, identická nula, následník, substituce, rekurze.

Průvodce studiem

Výpočetní modely mohou být nejen algoritmicky orientované – což je spíše bližší praktickému informatikovi, ale také zcela založené na funkcích jako matematickém pojmu. Uvidíte, že vše co lze naprogramovat RAM nebo TS lze popsat jako složení základních funkcí pomocí jednoduchých operatorů.

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.

7.1 Základní funkce a operátory

Rekurzivní funkce

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)

(n)(n)

$I_i : N^n \rightarrow N$, $\forall x1,x2, ..., xn : I_i(x1,x2, ..., xn) = xi$ (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: $f = S_n^m(g, h1, ..., hm)$, kde platí $f(x1,x2, ..., xn) = g(h1(x1, ..., xn), ..., hm(x1, ..., xn))$ (operátor, který umožňuje skládat funkce)

Operátory primitivní rekurze R^n: $f = R^n(g, h)$, kde platí $f(0,x2, ..., xn) = g(x2, ..., xn)$a $f(k+1,x2, ..., xn) = h(k, f(k, x2, ..., xn),x2, ..., xn)$ (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:

Zkusme se podívat na příklad funkce $f(x1,x2) = x1 + x2$. 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:

f1 = s, f2 = I_2^3,f3 = S_3^1(s, I_2^3),f4 = I_1^1,f = R(f4,f3)

přičemž jednotlivé funkce vyjadřují:

f -je funkce, kterou jsme chtěli vytvořit (sčítání dvou argumentů), přičemž jsme museli aplikovat rekurzi následujícím způsobem; f(0,x2)= x2,f(k+1,x2)= f(k, x2)+1; což znamená, že x1 -krát zvýšíme hodnotu x2 o jedna a použijeme k tomu upravenou funkci následníka

f4 -je funkcí výběru prvního argumentu z jednoho, kterou potřebujeme, abychom správně nadefinovali zarážku rekurze funkce f

f3 -je upravená funkce následníka pro druhý argument ze tří (použije se v rekurzi dle formální definice operátoru); f3(x1,x2,x3)= x2 +1

f2 -je potřebná pro vytvoření následníka druhého argument v f3; jde o výběr druhého argumentu ze tří f2(x1,x2,x3)= x2

f1 -je základní funkcí následník pro jednu proměnnou (f1(x1)= x1 + 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(x1)=5 − x1 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.

7.2 PL-programy

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:

PL-programy

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 X1,X2, OUTPUT Y

Příklad:

Například funkci f(x1,x2)= x1 + x2 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.

Nejdůležitější probrané pojmy:

- rekurzivní funkce
- základní funkce – identická nula, následník, projekce
- operátory – substituce, rekurze, minimalizace

Korespondenční úkol:

1. Sestrojte TS nebo rekurzivní funkci pro realizaci funkce f(x,y) = x+y. (pozn. TS může pracovat s čísly libovolné soustavy, nejjednodušší bude binární a výsledná páska bude obsahovat pouze hodnotu funkce)
2. Sestrojte RAM stroj pro výpočet odchylek od průměru pole čísel.

8 Složitost

Cíl:

Zatímco v předchozí části jsme se zabývali otázkou co lze a co nelze řešit, nyní se zaměříme na to, jak je řešení rozhodnutelných problémů složité. Po prostudování této části dokážete:

- vysvětlit základní úkoly teorie složitosti,
- rozlišovat mezi různými typy složitostí,
- určit složitost algoritmů implementovaných Turingovými stroji,
- používat různé typy odhadů složitostí,
- odhadnout složitosti některých základních problémů,
- zařadit problémy do odpovídajících tříd složitostí,
- charakterizovat třídu prakticky zvládnutelných problémů,
- specifikovat tzv. N P -úplné problémy,
- identifikovat zaručeně nezvládnutelné problémy

Průvodce studiem

Druhou částí teorie algoritmů (Vyčíslitelnosti a složitosti) je otázka složitosti. Poté, co jsme uspokojivým způsobem zodpověděli otázku 'Co všechno je algoritmicky řešitelné (vyčíslitelné)?', zauvažujme o základní otázce složitosti: Jak je řešení (vyčíslování) algoritmicky řešitelných problémů složité?

Složitost

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.

8.1 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 → 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}).

8.2 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 • 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 pro- blému hůře definovatelný (zamyslete se nad tím!).

8.3 Složitost problému

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

Všimněme si, že určitě platí např.

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

(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 $k \in N$):

8.4 Polynomiální převeditelnost

Definice: Problém P1 je polynomiálně převeditelný na problém P2, označme P1 ÁP2, jestliže existuje Turingův stroj M s polynomiální časovou složitostí, který pro libovolnou instanci I problému P1 sestrojí instanci problému P2, označme ji M (I), přičemž platí, že odpověď na otázku

problému P1 pro instanci I je ANO právě tehdy, když odpověď na otázku problému P2 pro instanci M (I) je ANO.

Úkoly k zamyšlení:

Je zřejmé, že jestliže označme P1 ÁP2 a P2 je v PTIME, pak i P1 je v PTIME. Naopak, když P1 není v PTIME, ani P2 není v PTIME. Zamyslete se nad tím!

Exponenciální časová složitost

Jednou ze silných motivací pro rozvoj strukturální složitosti je fakt, že u mnoha konkrétních praktických problémů nejsme (zatím) schopni prokázat, zda jsou či nejsou v PTIME. O těchto problémech většinou víme, že jsou v třídě EXPTIME, kde

$$EXPTIME = \bigcup_{k=0}^{\infty} \mathcal{T}(2^{n^k}).$$

Jsou známy konkrétní problémy, které jsou v EXPTIME, ale ne v PTIME, o spoustě z nich ale nepříslušnost k PTIME prokázána není.

Když si např. celkem přímočaře zavedeme třídy PSPACE, EXPSPACE založené na prostorové (paměťové) složitosti, lze snadno ukázat, že

PTIME ⊆ PSPACE ⊆ EXPTIME ⊆ EXPSPACE

neví se ovšem, které inkluze jsou vlastní. Např. je jasné, že jedna z inkluzí PTIME ⊆ PSPACE, PSPACE ⊆ EXPTIME musí být vlastní.
Zdá se sice, že vlastní jsou obě, nicméně stále není vyloučena možnost PT IME = PSPACE! Přitom o spoustě praktických problémů se ví, že jsou v PSPACE, ale neumí se prokázat nepříslušnost k PTIME. Mnoho těchto problémů je speciálního charakteru: jsou rozhodnutelné v polynomiálním čase nedeterministickým Turingovým strojem.

8.5 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´ÁP . 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.

8.6 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?

Když už máme k dispozici jeden NP-úplný problém, dá se NP-úplnost dalších problémů prokázat využitím následujícího tvrzení.

Tvrzení: Jestliže P1 ÁP2 a P1 je NP-těžký, pak P2 je rovněž NP-těžký. Speciálně, když P2 je v NP (P1 je pak rovněž v NP), pak P2 je NP-úplný.

Úkoly k zamyšlení:

Tohoto tvrzení lze využít při dokazování NP-úplnosti nějakého problému podobným způsobem, jako jsme využili větu pro dokazování nerozhodnutelnosti některých problémů.

8.7 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?

8.8 Nezvládnutelné problémy

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?

5. Vyjmenujte a vysvětlete některé nejznámější N P -úplné problémy.

Cvičení:

1. Určete časovou složitost Turingova stroje rozpoznávajícího jazyk

$L = \{a^n b^n c^n \mid n \geq 0\}$ uvedeného v příkladu.

2. Upravte algoritmus uvedený v již zmíněném příkladu tak, aby jeho složitost byla v $O(n \bullet \log n)$ a navrhněte k němu odpovídající Turingův stroj.

Pojmy k zapamatování:

- složitost algoritmu
- vnitřní složitost problému

- časová složitost
- prostorová složitost
- třída časové a prostorové složitosti
- P T IM E, N P T IM E, EXP T IM E
- P SP ACE, N P SP ACE, EXP SP ACE
- N P -úplné problémy

Literatura

[1] Habiballa, H. Vyčíslitelnost a složitost, OU Ostrava, 2013.

[2] Habiballa, H. Úvod do informatiky, OU Ostrava, 2013.

[3] U. Manber. Introduction to Algorithms, Addison-Wesley, 1989.

[4] Aho, A., Hopcroft, J., Ullman, J. The design and analysis of computer algorithms, Addison-Wesley, 1974.

[5] Jančar P. Teoretická informatika, VŠB Ostrava, 2010.

[6] Černá, I. Úvod do teorie složitosti, FI MUNI, 1993.

[7] O. Demuth, R. Kryl, and A. Kučera. Teorie algoritmů I, II. SPN, 1984.

[8] 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.

[9] Oded Goldreich. Computational Complexity: A Conceptual Perspective. Cambridge University Press, New York, NY, USA, 2008.

[10] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation (2nd Edition). Addison Wesley, November 2000.