Poznámky k přednášce NTIN090 Úvod do složitosti a vyčíslitelnosti

Petr Kučera

24. ledna 2023

Obsah

Část I.

Úvod

1. Motivace

Tento text obsahuje poznámky, které jsem si zpočátku psal jako přednášející pro sebe, ale snažil jsem se je rovnou psát i tak, aby pomohly studentům při studiu tohoto předmětu.

Přednáška by se mimo jiné měla pokusit zodpovědět následující otázky:

(I) Co je to algoritmus?

(II) Co všechno lze pomocí algoritmů spočítat?

(III) Dokáží algoritmy vyřešit všechny úlohy a problémy?

(IV) Jak poznat, že pro řešení zadané úlohy nelze sestrojit žádným algoritmus?

(V) Jaké algoritmy jsou „rychlé“ a jaké problémy jimi můžeme řešit?

(VI) Jaký je rozdíl mezi časem a prostorem?

(VII) Které problémy jsou lehké a které těžké? A jak je poznat?

(VIII) Je lépe zkoušet nebo být zkoušený?

(IX) Jak řešit problémy, pro které neznáme žádný „rychlý“ algoritmus?

Jde o otázky, které se věnují mezím toho, co je možné vyřešit pomocí algoritmů, tedy mezím nástrojů, které používají ti, kdo se zabývají informatikou a tedy i programováním. Výklad začneme tou nejzákladnější otázkou, tedy co je to algoritmus.

2. Lehký úvod do teorie algoritmů

V textu se věnujeme algoritmům, což je pojem poněkud těžko uchopitelný. Než se budeme bavit o algoritmech obecně, zkusme se proto chvíli zabývat něčím méně abstraktním, a to programy v jazyce C[1].

2.1. Začínáme pozdravem

Protože se budeme dále věnovat algoritmům a programům, které je implementují, sluší se začít programem, který vypíše na obrazovku tradiční pozdrav, tedy řetězec „`Hello, world`“.

Výpis programu 2.1: Program `helloworld.c`

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello, world\n");
    return 0;
}
```

Příklad takového programu vidíme ve výpisu 2.1, nazvěme ho `helloworld.c`. Podíváme-li se na tento program, vidíme, že tento program s jakýmkoli vstupem předaným mu na standardní vstup vypíše jako prvních dvanáct znaků svého výstupu řetězec „`Hello, world`“ a ihned skončí. Toto však zdaleka není jediný způsob, jak napsat program s touto funkcionalitou, uvažme program `helloworld2.c` zobrazený na výpisu 2.2.

Program `helloworld2.c` po spuštění načte celé číslo n ze standardního vstupu a poté hledá trojici čísel $x, y, z \in \mathbb{N}$, pro kterou by platilo $x^n + y^n = z^n$. Pokud je taková trojice nalezena, je vypsán řetězec „`Hello, world`“. K tomu ovšem dojde jen v případě, že $n = 1$ nebo 2. Tento je ekvivalentní velké Fermatově větě, což je tvrzení, které čekalo téměř 400 let na svůj důkaz od chvíle, kdy Pierre de Fermat tuto větu vyslovil. Zamysleme se však obecně nad otázkou, zda je možné sestrojit program H, který by za nás ověřil, zda

[1] Volba jazyka C pro použití v tomto motivačním příkladu je víceméně náhodná a bylo by možné zvolit jakýkoli jiný vyšší programovací jazyk. Od jazyka C jsou odvozeny další jazyky, které jsou dnes používané, proto by syntaxe použitá v této sekci neměla být zcela cizí ani těm, kdo se jinak s jazykem C nesetkali.

Výpis programu 2.2: Program `helloworld2.c`

```
#include <stdio.h>

int exp(int i, int n)
/* Vrátí n-tou mocninu i */
{
   int moc, j;
   moc=1;
   for (j=1; j<=n; ++j) moc *= i;
   return moc;
}

int main(int argc, char *argv[]) {
   int n, total, x, y, z;
   scanf("%d", &n);
   total=3;
   while (1) {
      for (x=1; x<=total-2; ++x) {
         for (y=1; y<=total-x-1; ++y) {
            z=total-x-y;
            if (exp(x,n)+exp(y,n)==exp(z,n)) {
               printf("Hello, world\n");
               return 0;
            }
         }
      }
      ++total;
   }
}
```

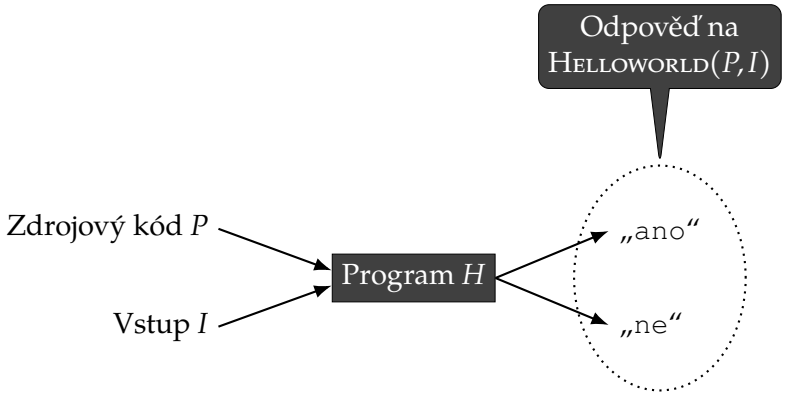
daný program P se vstupem I vypíše jako prvních dvanáct znaků svého výstupu právě „`Hello, world`“. Zaveďme si proto problém Helloworld.

Problém 2.1.1: Helloworld

Instance: Zdrojový kód programu P v jazyce C a jeho vstup I.

Otázka: Je pravda, že prvních 12 znaků, které daný program vypíše, je „`Hello, world`“? (Nevyžadujeme zastavení.)

Ukážeme si, že žádný program v jazyce C za nás problém Helloworld nevyřeší. Uvažujme pro chvíli (a pro spor), že máme takový program a nazvěme jej třeba H. Program H tedy očekává dva vstupy, zdrojový kód programu P v jazyce C a vstup tohoto programu I. V obou případech jde o textové soubory. Situace je znázorněna na obrázku 2.1.



Obrázek 2.1.: Vstup a výstup programu H.

Ve zbytku této podkapitoly přivedeme existenci programu H ke sporu. Základní myšlenkou je předložit programu H jeho vlastní zdrojový kód a ukázat, že program H nedokáže nic říci ani sám o sobě, natož aby to uměl obecně o všech ostatních programech. Ještě předtím však musíme program H upravit, protože tento program nevypisuje řetězec „`Hello, world`“ za žádné situace a navíc očekává na vstupu dva soubory, zatímco v problému Helloworld uvažujeme pouze programy s jedním vstupním souborem. Pro zjednodušení situace budeme uvažovat pouze programy, které splňují následující dvě omezení:

(i) Předpokládáme, že vstupní soubory jsou předávány všem uvažovaným programům pouze na standardní vstup, který čtou programy výhradně funkcí `scanf`[2].

[2]Jde o standardní funkci, jež se v jazyce C používá pro formátovaný vstup pro čtení ze standardního vstupu. Například volání `scanf("%d", &n)` použité v programu 2.2 načte ze vstupu řetězec kódující celé číslo, které dosadí do proměnné `n`.

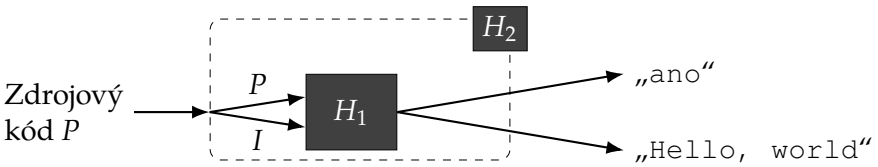
Pokud program očekává více vstupních souborů (což je případ programu H), pak načte ze standardního vstupu nejprve jeden vstupní soubor a poté druhý vstupní soubor. Konec prvního vstupního souboru pozná program při načtení vyhrazeného znaku „`EOF`“. Vzhledem k tomu, že se jedná o textové soubory a že programy jsou psané v jazyce C, může roli znaku „`EOF`“ hrát například znak s kódem 0, tj. znak „`\0`“.

(ii) Předpokládáme, že výstup všech programů je textový a že je zapisován na standardní výstup, a to výhradně funkcí `printf`[3].

První úpravou programu H bude náhrada vypisovaného řetězce „`ne`“ na „`Hello, world`“. Program H_1 s touto funkcí získáme následující úpravou: Vypíše-li H jako první znak „`n`“, víme, že nakonec vypíše „`ne`“. Změníme tedy funkčnost funkce `printf` takovým způsobem, že v okamžiku, kdy je tato funkce poprvé zavolána a je-li prvním znakem požadovaného výpisu „`n`“, pak místo svého výpisu funkce `printf` vypíše řetězec „`Hello, world`“ a zajistí (pomocí booleovské proměnné), že od této chvíle žádné volání funkce `printf` nevykoná žádný výpis. Zkonstruujeme takto program H_1, jehož funkčnost je naznačena na obrázku 2.2.

Obrázek 2.2.: Funkce programu H_1.

Naším cílem je předložit programu H_1 jeho zdrojový kód a podívat se, co takto o sobě program H_1 bude schopen říci. K tomu stačí postavit zdrojový kód programu H_1 do role vstupního zdrojového kódu P, musíme však předložit programu H_1 také vstupní soubor I. Položíme tedy $I = P$ a na základě této myšlenky vytvoříme program H_2, který bude očekávat jen jeden vstup, a to program P, který předloží programu H_1 jako oba požadované vstupy, tedy P i I. Program H_2 tak jinak pracovat týmž způsobem jako H_1. Situace programu H_2 je naznačena na obrázku 2.3.



Obrázek 2.3.: Funkce programu H_2.

Program H_2 získáme z programu H_1 následujícím postupem:

[3]Jde o standardní funkci jazyka C pro formátovaný výstup. Volání `printf("Hello, world")` prostě vypíše na standardní výstup řetězec „`Hello, world`“.

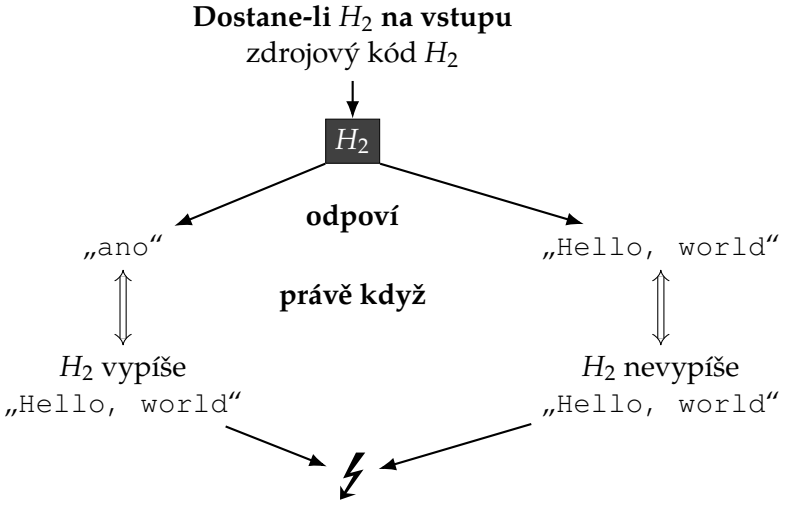
1: Program H_2 nejprve načte celý vstup a uloží jej v poli A, které pro tento účel alokuje v paměti (např. pomocí `malloc`).

2: Poté program H_2 simuluje práci H_1, přičemž:

   a: Ve chvíli, kdy H_1 čte vstup (pomocí `scanf`), H_2 místo čtení přistoupí do pole A. To znamená, že změníme implementaci `scanf` tak, aby místo čtení vstupu četla z pole A.

   b: Pomocí dvou ukazatelů do pole A si H_2 pamatuje, kolik z P a I program H_1 přečetl (`scanf` čte popořadě, takže ukazatele začínají inicializované nulou a posléze je vždy při přečtení znaky ze vstupu ve funkci `scanf` program H_2 inkrementuje).

Je zřejmé, že takto zkonstruovaný program H_2 bude mít požadovanou funkci. To znamená, že pokud je mu předložen zdrojový kód programu P, pak H_2 simulací programu H_1 (potažmo H) zjistí, zda P jako prvních dvanáct znaků svého výstupu vypíše řetězec „`Hello, world`“. Pokud tomu tak je, vypíše program H_2 na svůj výstup řetězec „`ano`“, v opačném případě vypíše H_2 na svůj výstup řetězec „`Hello, world`“.



Obrázek 2.4.: Pokud zavoláme program H_2 předloživše mu na vstup jeho vlastní zdrojový kód, nutně dojdeme k závěru, že ani jedna z možných odpovědí H_2 není správná, a tedy program H_2 nemůže existovat.

Položme si nyní otázku, co se stane, pokud předložíme programu H_2 na vstup jeho vlastní zdrojový kód. V úvahu připadají dvě možnosti (jež jsou naznačeny na obrázku 2.4). Buď H_2 (se vstupem $P = H_2$) odpoví „`ano`“, k tomu může ovšem dojít jen v situaci, kdy H_2 (v roli P) ve skutečnosti jako prvních dvanáct znaků svého výstupu vypíše řetězec „`Hello, world`“, což ovšem není možné, buď vypíše „`ano`“, nebo „`Hello, world`“, ale nikoli obojí. Tato možnost tedy vede ke sporu. Druhou možností je, že H_2

(se vstupem $P = H_2$) odpoví „`Hello, world`“. To znamená, podle toho, jak jsme program H_2 konstruovali, že program H_2 (nyní v roli P, jemuž je předložen P na vstupu) ve skutečnosti nevypíše jako prvních dvanáct znaků svého výstupu „`Hello, world`“, a tedy vypíše „`ano`“. Opět tak dostáváme spor.

Jediným možným závěrem je, že program H_2 nemůžeme takto zkonstruovat, což znamená, že program H_1 a zejména ani program H nemohou existovat. Problém Helloworld není řešitelný žádným programem v jazyce C. Tento fakt jsme ukázali sporem s použitím *diagonalizace*, později se dostaneme k tomu, v čem diagonalizace spočívá a z čeho pochází tento název.

2.2. Volá či nevolá program funkci `foo`?

Uvažme nyní jiný, snad i praktičtější problém, v němž se ptáme, zda daný program volá funkci s daným jménem, tomuto problému budeme říkat Volání funkce **`foo`**.

Problém 2.2.1: Volání funkce `foo`

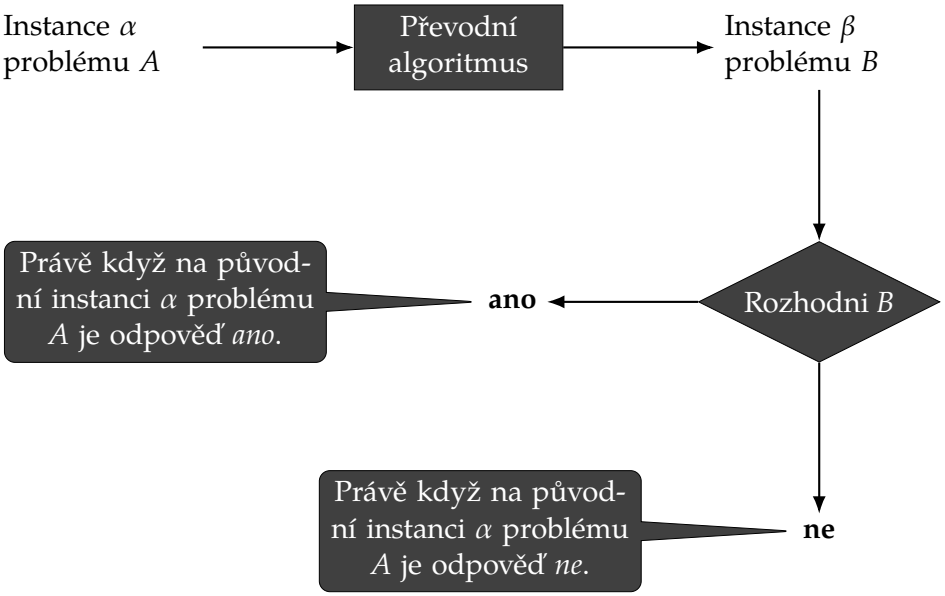
Instance: Zdrojový kód programu Q v jazyce C a jeho vstup V.

Otázka: Zavolá program Q při běhu nad vstupem V funkci jménem **`foo`**?

Ukážeme si, že ani tento problém není možné vyřešit žádným programem v jazyce C. Nyní však využijeme toho, že už máme jeden nerozhodnutelný problém, a to problém Helloworld. Ukážeme, že kdybychom byli schopni rozhodnout problém Volání funkce **`foo`** programem H_3 v jazyce C, mohli bychom rozhodnout i problém Helloworld (jiným) programem H v jazyce C. Protože víme, že takový program H neexistuje, ukážeme tím, že ani program H_3 rozhodující Volání funkce **`foo`** nemůže existovat.

Přesněji, ukážeme, jak *převést problém Helloworld na problém Volání funkce* **`foo`**. Použijeme k tomu způsob převodu jednoho problému na druhý, který je naznačen na obrázku 2.5. Uvažme problém A a problém B (každý z nich je daný popisem toho, jak vypadá *instance* problému a jakou *otázku* si klademe o takové instanci). Při převodu problému A na problém B musíme popsat *převodní algoritmus*, který přetvoří instanci α problému A na instanci β problému B. Přičemž musíme zajistit, aby platilo, že *kladná instance* problému A (tedy instance, na niž je odpověď kladná) je převedena na kladnou instanci problému B a *záporná instance* problému A (tedy instance, na niž je odpověď záporná) je převedena na zápornou instanci problému B. Potom můžeme říci, že máme-li program H_B, který rozhoduje problém B, jsme schopni zkonstruovat program H_A, který rozhoduje problém A: Program H_A se vstupem α prostě použije převodní algoritmus pro vytvoření instance β problému B, na níž pustí program H_B a vrátí touž odpověď jako tento program.

Ukážeme si nyní, jak v tomto smyslu převést problém Helloworld na problém Volání funkce **`foo`**. Již víme, že nemůže existovat program v jazyce C (a ani v jiném programovacím jazyce), který by byl schopen rozhodnout problém Helloworld, dostaneme tak,



Obrázek 2.5.: Princip převodu problému A na problém B.

že ani problém Volání funkce **foo** nelze rozhodnout programem v jazyce C.

Vyjdeme z instance problému Helloworld a popíšeme postup, jak ji převést na instanci problému Volání funkce **foo**. Instance obou problémů vypadají podobně — jedná se vždy o dvojici programu a vstupního souboru. Uvažme tedy dvojici P, I, která tvoří instanci problému Helloworld, kde P je zdrojový kód programu a I je vstupní soubor. Popíšeme postup, který tuto dvojici přetvoří do jiné dvojice Q, V, kde Q je zdrojový kód programu a V je vstupní soubor, přitom tento postup zabezpečí, že P, I je kladnou instancí problému Helloworld, právě když Q, V je kladnou instancí problému Volání funkce **foo**. Přesněji, že:

$$P \text{ se vstupem } I \text{ vypíše jako prvních dvanáct znaků svého výstupu řetězec „}\texttt{Hello, world}\text{“.} \iff Q \text{ se vstupem } V \text{ zavolá funkci } \textbf{foo}. \tag{2.1}$$

Při převodu P, I na Q, V budeme postupovat následujícím způsobem:

1: Je-li v P funkce **foo**, přejmenujeme ji i všechna její volání na dosud nepoužité jméno (*refaktoring*, výsledný program nazveme P_1).

2: K programu P_1 přidáme funkci **foo**, funkce nic nedělá a není volána ($\rightarrow P_2$).

3: Upravíme program P_2 tak, aby si pamatoval prvních dvanáct znaků, které vypíše a uložil je v poli A. Jde prostě o úpravu implementace funkce `printf`, A je nové pole znaků o dvanácti prvcích, jež se v programu jinak nevyskytuje a je lokální nové implementaci funkce `printf`. ($\rightarrow P_3$).

4: Upravíme program P_3 tak, že pokud použije příkaz pro výstup, zkontroluje pole A, je-li v něm alespoň dvanáct znaků a na začátku obsahuje „`Hello, world`“. Pokud ano, zavolá funkci **`foo`** (tím dostaneme výsledný program Q, vstup V položíme roven I, tedy $V = I$).

Nahlédněme nyní, že skutečně platí ekvivalence (2.1).

$\Rightarrow$ Předpokládejme nejprve, že program P se vstupem I vypíše jako prvních dvanáct znaků svého výstupu řetězec „`Hello, world`“. Potom to program Q se vstupem $V = I$ ve své implementaci printf zjistí a zavolá funkci **`foo`**.

$\Leftarrow$ Na druhou stranu předpokládejme, že program Q se vstupem $V = I$ zavolá funkci **`foo`**. Z konstrukce programu Q vyplývá, že k tomuto volání může dojít jen tehdy, pokud Q ve své implementaci funkce `printf` zjistí, že pole A obsahuje řetězec „`Hello, world`“. To je ovšem možné jen pokud původní program P se vstupem I vypsal jako prvních dvanáct znaků svého výstupu řetězec „`Hello, world`“.

Z toho vyplývá, že problém Volání funkce **`foo`** není řešitelný programem v jazyce C.

2.3. Nevýhody jazyka C pro budování teorie algoritmů

Ve zbytku tohoto textu je naším cílem popsat prostředky pro popis algoritmů. Pro tento účel bychom jistě mohli využít jazyk C s tím, pokud bychom připustili, že každý algoritmus lze implementovat v jazyce C (což nyní poznamenáváme s vědomím toho, že jsme dosud nespecifikovali formálně, co myslíme pojmem algoritmu). Jazyk C (jakož i jiný vyšší programovací jazyk) má však pro tyto účely řadu nevýhod.

(I) Jazyk C je příliš komplikovaný. Jinými slovy, pokud bychom skutečně použít jazyk C pro teoretický popis algoritmů, museli bychom věnovat značné úsilí tomu, abychom si popsali, co myslíme jazykem C, tedy popsat jeho specifikaci. Ta je dosti složitá a nejspíš bychom narazili i na řadu nejasností a nejednoznačností.

(II) Museli bychom definovat výpočetní model (tj. zobecněný počítač), který bude programy v jazyce C interpretovat. To proto, že k tomu, abychom skutečně byli schopni algoritmem zapsaným jako program v jazyce C něco spočítat, musíme jej přeložit a spustit na počítači. Způsob překladu a interpretace přeloženého kódu je pochopitelně podstatná pro zodpovězení otázek souvisejícím s tím, co a jak rychle je možno programy v jazyce C spočítat.

(III) V době vzniku teorie nebyly procedurální jazyky k dispozici, proto je teorie v literatuře obvykle popisovaná tradičnějšími prostředky. Tato teorie je již dobře vybudovaná a nedává proto smysl přepisovat ji pomocí složitějších prostředků.

Místo jazyka C nebo jiného vyššího programovacího jazyka potřebujeme tedy zvolit výpočetní model, který bude dostatečně jednoduchý, aby bylo možno jej jednoduše popsat, ale i dostatečně silný, aby zachycoval to, co si představujeme pod pojmem algoritmu.

2.4. Bibliografické poznámky

Motivační příklady popsané, tj. nerozhodnutelnost problémů Helloworld a Volání funkce `foo`, byly převzaty z [4].

3. Definice a konvence

V této kapitole zavedeme základní pojmy a konvence použité v textu.

3.1. Logika a logické formule

Pro zápis logických formulí budeme používat běžně užívané logické spojky. Občas budeme pracovat s formulemi v konjunktivní normální formě a formulemi v disjunktivní normální formě.

Literál je buď proměnná nebo její negace, proměnnou x nazveme *pozitivním literálem* a její negaci $\neg x$ nazveme *negativním literálem*. *Klauzule* je disjunkcí literálů, která neobsahuje dva literály s touž proměnnou, například $(x \vee \overline{y} \vee \overline{z})$. Formule φ je v *konjunktivní normální formě* (KNF), pokud je konjunkcí klauzulí, například $(x \vee \overline{y} \vee \overline{z}) \wedge (\overline{x} \vee y) \wedge (y \vee z) \wedge \overline{x}$. *Term* je konjunkcí literálů, která neobsahuje dva literály s touž proměnnou, například $(x \wedge \overline{y} \wedge \overline{z})$. Formule φ je v *disjunktivně normální formě* (DNF), pokud je disjunkcí termů, například $(x \wedge \overline{y} \wedge \overline{z}) \vee (\overline{x} \wedge y) \vee (y \wedge z) \vee \overline{x}$.

Je-li φ formulí na n proměnných a $t \in \{0,1\}^n$, pak pomocí $\varphi(t)$ označíme hodnotu, na kterou se φ vyhodnotí, přiřadíme-li proměnným hodnoty dané vektorem t. Hodnota 0 označuje lež, či *false*, zatímco hodnota 1 označuje pravdu, či *true*.

3.2. Množiny, relace a funkce

V této části zavedeme pojmy související s množinami a přirozenými čísly.

3.2.1. Množiny

Jsou-li A a B množiny, pak

$A \cup B$ označuje sjednocení množin A a B.

$A \cap B$ označuje průnik množin A a B.

$A \setminus B$ označuje rozdíl množin A a B.

$A \times B$ označuje kartézský součin množin A a B.

A^n pro $n \geq 0$ je n-tá kartézská mocnina množiny A a je definovaná jako $A = A^1$ a $A^n = A^{n-1} \times A$, pro $n > 1$.

$|A|$ označuje počet prvků množiny A, samozřejmě jen pokud je A konečná.

Jsou-li $A_1, \ldots, A_n$ množiny (a $n \geq 1$) a $a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n$ prvky těchto množin, pak uspořádanou n-tici tvořenou těmito prvkymi označujeme pomocí $(a_1, \ldots, a_n)$. Zřejmě platí $(a_1, \ldots, a_n) \in A_1 \times A_2 \times \cdots \times A_n$. Pokud navíc $A = A_1 = A_2 = \cdots = A_n$, pak $(a_1, \ldots, a_n) \in A^n$.

Potenční množinou množiny A je množina jejích podmnožin, kterou definujeme jako

$$\wp(A) = \{B \mid B \subseteq A\} \tag{3.1}$$

V případě konečné množiny A platí, že $|\wp(A)| = 2^{|A|}$. Speciálně $|\wp(A)|$ je vždy neprázdná, neboť vždy obsahuje přinejmenším prázdnou množinu.

3.2.2. Relace

Pro $n \geq 1$ definujeme *n-ární relaci R mezi množinami $A_1, \ldots, A_n$* jako množinu uspořádaných n-tic $R \subseteq A_1 \times \cdots \times A_n$, *$n$-ární relaci R na množině A* pak definujeme jako množinu uspořádaných n-tic $R \subseteq A^n$. Relaci budeme též říkat *predikát*.

Zvláštní roli hrají *binární relace*, tedy relace arity 2. O relaci $R \subseteq A^2$ řekneme, že je

Reflexivní pokud pro každé $x \in A$ je $(x, x) \in R$.

Antireflexivní pokud pro každé $x \in A$ je $(x, x) \notin R$.

Symetrická pokud pro každé $x, y \in A$ platí, že je-li $(x, y) \in R$, pak i $(y, x) \in R$.

Antisymetrická pokud pro každé $x, y \in A$ platí, že je-li $(x, y) \in R$ a $(y, x) \in R$, pak $y = x$.

Tranzitivní pokud pro každé $x, y, z \in A$ platí, že je-li $(x, y) \in R$ a $(y, z) \in R$, pak $(x, z) \in R$.

Zvláštní typy relací, které uvažujeme, jsou například.

Ekvivalence tedy reflexivní, symetrická a tranzitivní relace.

Částečné uspořádání tedy reflexivní, antisymetrická a tranzitivní relace.

Kvaziuspořádání tedy reflexivní a tranzitivní relace.

Unární relace či predikát $R \subseteq A$ má aritu 1 a tedy jde pouze o podmnožinu množiny A.

3.2.3. Zobrazení a funkce

Zobrazením či *funkcí* z množiny A do množiny B míníme relaci $f \subseteq A \times B$ pro kterou platí, že pro každý prvek $x \in A$ existuje nejvýš jeden prvek $y \in B$, pro který platí, že $(x, y) \in f$, tuto skutečnost častěji píšeme jako $f(x) = y$. Fakt, že f je funkce z množiny A do množiny B zapíšeme pomocí

$$f : A \to B.$$

Pokud je funkce f zobrazením z množiny $A_1 \times A_2 \times \cdots \times A_n$ pro nějaké $n \geq 1$ do množiny B, hovoříme o n-ární funkci (přičemž pokud je $n = 1$, jde o *unární* funkci a pokud je $n = 2$,

jde o *binární* funkci). V takovém případě píšeme hodnotu funkce f pro n-tici $a_1, \ldots, a_n$ jako $f(a_1, \ldots, a_n)$.

Definičním oborem (také *doménou*) funkce $f \subseteq A \times B$ je množina

$$\operatorname{dom} f = \{x \mid (\exists y)[f(x) = y]\},$$

přičemž pro $x \in \operatorname{dom} f$ je hodnota funkce f *definovaná*, což zapíšeme pomocí $f(x)\downarrow$, zatímco pro $x \notin \operatorname{dom} f$ je hodnota funkce f *nedefinovaná*, což zapíšeme pomocí $f(x)\uparrow$. Fakt, že hodnota funkce $f(x)$ je definovaná pro $x \in \operatorname{dom} f$ a současně je tato hodnota rovna $y \in B$, budeme také psát pomocí $f(x)\downarrow = y$. *Oborem hodnot* funkce f je množina

$$\operatorname{rng} f = \{y \mid (\exists x)\,[f(x)\downarrow \wedge f(x) = y]\}.$$

Pokud je $f : A \to B$ pro množiny A a B a pokud platí, že $\operatorname{dom} f = A$, pak jde o *totální funkci*, v opačném hovoříme o *částečné* funkci.

O funkci $f : A \to B$ řekneme, že je

Prostá pokud pro každé dva prvky $x_1, x_2 \in \operatorname{dom} f$, platí, že je-li $f(x_1) = f(x_2)$, pak $x_1 = x_2$.

Na pokud $\operatorname{rng} f = B$.

Vzájemně jednoznačné zobrazení (též *bijekce*) pokud je totální, prostá i na.

Jsou-li $f : B \to C$ a $g : A \to B$ funkce, pak složením těchto funkcí dostaneme funkci $(f \circ g) : A \to C$, která je definovaná právě pro ty prvky $x \in A$ pro něž je $x \in \operatorname{dom} g$ a $g(x) \in \operatorname{dom} f$, tj.

$$\operatorname{dom}(f \circ g) = \{x \in \operatorname{dom} g \mid g(x) \in \operatorname{dom} f\}.$$

Pro každý prvek $x \in \operatorname{dom}(f \circ g)$ definujeme $(f \circ g)(x) = f(g(x))$. Podobně bychom mohli zavést i složení funkcí více parametrů, přičemž pokud je f funkce arity n, pak uvažujeme n vnitřních funkcí $g_1, \ldots, g_n$.

Budeme občas používat anonymní funkce, tj. nepojmenované funkce definované výrazem. K jejich zápisu budeme používat *Churchovo λ-značení*: Je-li V výraz, pak pomocí $\lambda x_1 x_2 \ldots x_n V$ označíme funkci na proměnných $x_1, x_2, \ldots, x_n$, která je daná výrazem V. Například $\lambda abc[a + c]$ je funkce, která očekává tři (číselné) parametry, a, b a c a hodnotou této funkce je součet prvního a třetího parametru. Všimněme si, že samotný zápis $a + c$ není dostatečný, protože nespecifikuje přesně, kolik parametrů funkce očekává ani jejich pořadí.

Je-li A množina prvků z univerza U (tedy $A \subseteq U$), pak pomocí χ_A budeme značit *charakteristickou funkci množiny A*, jde o funkci $\chi_A : U \to \{0,1\}$, kde

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

Speciálně, pokud A je n-ární relace (či predikát), pak můžeme také hovořit o charakteristické funkci této relace (či predikátu). Charakteristická funkce je vždy totální.

3.3. Čísla a spočetné množiny

V této části zavedeme pojmy související zejména s přirozenými čísly, připomeneme si také, co myslíme pojmem spočetné množiny.

3.3.1. Celá a přirozená čísla

Množinu celých čísel označujeme pomocí $\mathbb{Z}$, množinu přirozených čísel pak pomocí $\mathbb{N}$, přičemž nulu považujeme za přirozené číslo (tj. $0 \in \mathbb{N}$). Doplňkem množiny $A \subseteq \mathbb{N}$ míníme množinu $\overline{A} = \mathbb{N} \setminus A$. Množinu reálných čísel budeme označovat pomocí $\mathbb{R}$ a množinu racionálních čísel pomocí $\mathbb{Q}$.

3.3.2. Mohutnost množiny, spočetné a nespočetné množiny

Množiny můžeme mezi sebou porovnávat s pomocí *mohutnosti* následujícím způsobem. Jsou-li A a B množiny, pak řekneme, že

1. množina A má *shodnou nebo menší mohutnost* než množina B, pokud existuje prosté zobrazení $f : A \to B$,
2. množina A má *shodnou mohutnost* s množinou B, pokud existuje bijekce $f : A \to B$ a
3. množina A má *menší mohutnost* než množina B pokud platí bod 1, ale nikoli bod 2.

Například je-li $A \subseteq B$, pak je mohutnost množiny A shodná nebo menší než mohutnost množiny B. Řekneme, že množina A je *spočetná*, pokud je její mohutnost shodná nebo menší než mohutnost množiny přirozených čísel $\mathbb{N}$. To znamená, že každému prvku A můžeme přiřadit nějaké přirozené číslo takovým způsobem, že dvěma různým prvkům vždy přiřadíme dvě různá přirozená čísla. Například každá konečná množina je zřejmě spočetná.

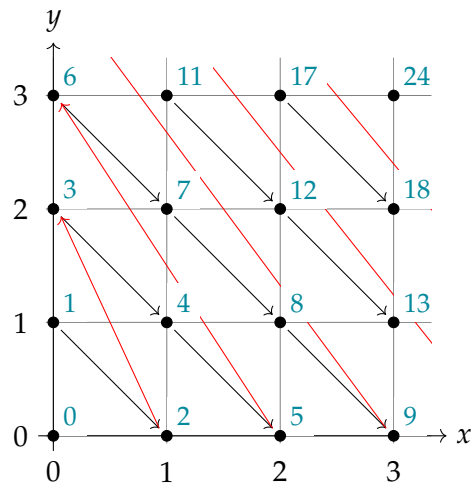
Platí, že sjednocení, průnik, rozdíl i kartézský součin spočetných množin jsou opět spočetné množiny, speciálně je-li množina A spočetná, pak i množina n-tic A^n je spočetná pro každé n.

Číslování n-tic přirozených čísel

Pokud chceme například ukázat, že $\mathbb{N}^2$ je spočetná množina, můžeme uvážit Cantorovu funkci dvojice, která je definovaná následujícím předpisem:

$$\pi_2(x, y) = \frac{(x + y)(x + y + 1)}{2} + x \tag{3.2}$$

Tato funkce je dokonce bijekcí mezi $\mathbb{N}^2$ a $\mathbb{N}$, způsob, jakým čísluje tato funkce dvojice čísel, je naznačen na obrázku 3.1.



Obrázek 3.1.: Pořadí, v jakém Cantorova párovací funkce čísluje dvojice přirozených čísel.

Z toho přímo plyne, že množina racionálních čísel $\mathbb{Q}$ a tedy i množina celých čísel $\mathbb{Z}$ jsou spočetné, protože racionální číslo odpovídá dvojici čísel přirozených — dělenci a děliteli.

Číslování dvojic přirozených čísel můžeme rozšířit na n-tice pro $n \geq 2$ induktivním způsobem:

$$\pi_n(x_1, \ldots, x_n) = \begin{cases} \pi_2(x_1, x_2) & n = 2 \\ \pi_2(x_1, \pi_{n-1}(x_2, \ldots, x_n)) & n > 2 \end{cases} \tag{3.3}$$

Funkce $\pi_n(x_1, \ldots, x_n)$ je bijekcí mezi $\mathbb{N}^n$ a $\mathbb{N}$. Dává proto smysl zavést inverzní funkci projekce. Pomocí $\pi_{n,i}^{-1}(:)\mathbb{N} \to \mathbb{N}$, kde $n, i \in \mathbb{N}$ a $i \leq n$ označíme funkci jedné proměnné definovanou následujícím předpisem:

$$\pi_{n,i}^{-1}(\pi_n(x_1, \ldots, x_n)) = x_i \tag{3.4}$$

Funkce $\pi_{n,i}^{-1}(z)$ si vyloží parametr z jako kód n-tice $x_1, \ldots, x_n$ a vrátí její i-tou složku, tedy x_i.

Nespočetné množiny

Je potřeba také zmínit, že například množina reálných čísel $\mathbb{R}$ spočetná není a spočetná není ani množina $\wp(\mathbb{N})$. Podle Cantorovy věty navíc platí pro každou množinu A, že její potenční množina $\wp(A)$ má větší mohutnost než A.

3.4. Řetězce a jazyky

V této části zavedeme pojmy související s řetězci a jazyky.

3.4.1. Řetězce

Abeceda je konečná množina znaků Σ. *Řetězec* nebo také *slovo* nad abecedou Σ je posloupnost znaků, psaná obvykle za sebou bez mezer. Například je-li $\Sigma = \{a, b, c, d\}$, pak $abc, cd, ddadb$ jsou slova nad abecedou Σ. Množinu všech řetězců nad abecedou Σ označujeme Σ^*, přičemž pokud je abeceda Σ jednoprvková, tedy například $\Sigma = \{1\}$, pak místo $\{1\}$ píšeme též 1^*. *Prázdný řetězec* označujeme ε. *Délkou slova* $w \in \Sigma^*$ rozumíme počet znaků v tomto slově a označujeme ji $|w|$. Například $|\varepsilon| = 0$, $|abc| = 3$. Je-li $w \in \Sigma^*$ řetězec a $i \in \{1, \ldots, |w|\}$ index, pak $w[i]$ označuje i-tý znak řetězce w.

Množinu všech slov délky i pro $i \in \mathbb{N}$ nad abecedou Σ označíme pomocí Σ^i, platí zřejmě, že

$$\Sigma^* = \bigcup_{i \in \mathbb{N}} \Sigma^i.$$

Jsou-li $u, v \in \Sigma^*$ slova nad abecedou Σ, pak jejich *konkatenací* je slovo $w = uv$ vzniklé tak, že slovo v připojíme za slovo v. Například konkatenací slov abc a ba je slovo $abcba$. Zřejmě platí, že $w = w\varepsilon = \varepsilon w$ pro každé slovo Σ^*. Je-li $w \in \Sigma^*$ slovo a $i \in \mathbb{N}$, pak w^i označuje slovo w zapsané i-krát za sebou, přitom pro $i = 0$ dostáváme $w^0 = \varepsilon$. Například $a^4 = aaaa$, $a^2b^3 = aabbb$, $(ab)^2 = abab$. Pomocí w^R označujeme *zrcadlový obraz* slova $w \in \Sigma^*$, tedy slovo, které vznikne otočením pořadí znaků ve slově w. Například $(abcd)^R = dcba$.

3.4.2. Uspořádání řetězců

Pro porovnávání řetězců budeme používat variantu lexikografického uspořádání, kde prvním kritériem pro porovnání je délka řetězce, přičemž řetězce s touž délkou jsou porovnány jako ve slovníku. Této variantě budeme říkat *shortlex uspořádání*.

Definice 3.4.1 (Shortlex uspořádání) Nechť Σ je abeceda, předpokládejme navíc, že máme k dispozici relaci lineárního uspořádání na znacích abecedy (které označíme pomocí $<$). Nechť $x, y \in \Sigma^*$ jsou dva různé řetězce. Řekneme, že řetězec x je *menší v shortlex uspořádání* než y, pokud buď

- $|x| < |y|$, nebo
- $|x| = |y|$ a je-li $i = \min\{j \mid x[j] \neq y[i]\}$, pak $x[i] < y[j]$.

Tento fakt označíme pomocí $x \prec y$. Obvyklým způsobem definujeme další varianty tohoto značení ($\preceq$ — menší nebo rovno, $\succ$ — větší a $\succeq$ — větší nebo rovno). ◂

Například pokud $\Sigma = \{a, b\}$, kde $a < b$, pak řetězce z Σ^* v shortlex uspořádání mají pořadí: $\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots$.

Je potřeba si uvědomit, že shortlex uspořádání se liší od slovníkového uspořádání (tj. způsobu třídění řetězců běžně užívaného ve slovnících). Ve slovníkovém uspořádání

se obvykle kratší z řetězců x a y před porovnáním doplní mezerami zprava na délku delšího řetězce (kde mezera je znak menší než všechny ostatní). Potom teprve proběhne porovnání podle prvního lišícího se znaku. Slovníkové uspořádání není ovšem vhodné v situaci, kdy máme nekonečnou množinu řetězců. Je-li například $\Sigma = \{a, b\}$, kde $a < b$, pak v Σ^* je jen pět řetězců, jež jsou menší než ba v shortlex uspořádání. Na druhou stranu je ovšem v Σ^* nekonečně mnoho řetězců, které jsou menší než ba ve slovníkovém uspořádání (jde o prázdný řetězec a dále všechny řetězce začínající znakem a).

3.4.3. Jazyky

Množině slov $L \subseteq \Sigma^*$ nad abecedou Σ říkáme *jazyk* (nad abecedou Σ). Například $L = \{ab, cd\}$ je jazyk se dvěma řetězci, jazyk $L = \varnothing$ je prázdný a jazyk $L = \{a^i b^i \mid i \in \mathbb{N}\}$ je jazyk slov tvaru $a^i b^i$ pro $i \in \mathbb{N}$. *Doplňkem* jazyka L nad abecedou Σ myslíme jazyk $\overline{L} = \Sigma^* \setminus L$. *Konkatenací* dvou jazyků L_1 a L_2 nad abecedou Σ vznikne jazyk $L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. *Kleeneho uzávěrem* jazyka L je jazyk

$$L^* = \{w \mid (\exists k \in \mathbb{N})(\exists w_1, \ldots, w_k \in L)[w = w_1 w_2 \ldots w_k]\}.$$

3.4.4. Číslování řetězců

Máme-li definované uspořádání řetězců, můžeme tyto řetězce očíslovat. Toho budeme často využívat zejména u binárních řetězců, číslování řetězců však můžeme definovat obecně. Uvažme pevnou konečnou abecedou Σ. Ke každému řetězci $w \in \Sigma^*$ definujeme jeho index

$$\text{index}(w) = |\{u \mid u \prec w\}|$$

Hodnota $\text{index}(w)$ je tedy definována jako počet řetězců, které jsou menší než w v shortlex uspořádání. Například $\text{index}(\varepsilon) = 0$. Z definice shortlex uspořádání plyne, že $\text{index}(w)$ je dobře definované přirozené číslo (tj. jeho hodnota je konečná). Platí dokonce, že index je bijekcí mezi Σ^* a $\mathbb{N}$.

Množina Σ^* je spočetná, neboť máme bijekci Σ^* na $\mathbb{N}$. Z toho plyne, že i libovolný jazyk $L \subseteq \Sigma^*$ je spočetný. Současně tím dostáváme vzájemně jednoznačnou korespondenci mezi množinami přirozených čísel a jazyky nad abecedou Σ. Množině přirozených čísel $A \subseteq \mathbb{N}$ odpovídá jazyk $L = \{w \mid \text{index}(w) \in A\}$. Na druhou stranu jazyku $L \subseteq \Sigma^*$ odpovídá množina kladných přirozených čísel $A = \{\text{index}(w) \mid w \in L\}$. Z toho plyne, že množina všech jazyků nad binární abecedou $\wp(\Sigma)$ už spočetná není (protože není spočetná množina $\wp(\mathbb{N})$). Toto platí i pro jednoprvkovou abecedu $\Sigma = \{\mathtt{1}\}$.

Číslování binárních řetězců

Často budeme uvažovat řetězce nad binární abecedou $\{\mathtt{0}, \mathtt{1}\}$, kde má číslování řetězců zajímavou interpretaci. Uvažme binární řetězec $w \in \{\mathtt{0}, \mathtt{1}\}^*$ a jeho index $i = \text{index}(w)$. Potom platí, že

$$(i+1)_B = \mathtt{1}w,$$

w	$1w$	$\mathrm{index}(w)+1$	$\mathrm{index}(w)$
ε	1	1	0
0	10	2	1
1	11	3	2
00	100	4	3
⋮	⋮	⋮	
001011	1001011	75	74
⋮	⋮	⋮	

Tabulka 3.1.: Příklady binárních řetězců a jejich indexů.

kde $(i+1)_B$ označuje binární zápis čísla $i+1$ bez zbytečných úvodních 0 a $1w$ označuje řetězec w, na jehož začátek přidáme znak 1. V tomto případě je tedy opravdu snadné přepočítávat navzájem řetězce a jejich indexy. Příklady binárních řetězců a jejich indexů ukazuje tabulka 3.4.4.

Definice 3.4.2 Uvažme přirozené číslo n. Pomocí $\mathbf{w}_n$ označíme n-tý binární řetězec, tedy binární řetězec, pro který platí $\mathrm{index}(\mathbf{w}_n) = n$. ◂

Jelikož index je bijekcí mezi $\{0,1\}^*$ a $\mathbb{N}$, platí, že v posloupnosti $\{\mathbf{w}_n\}_{n\in\mathbb{N}}$ se každý binární řetězec vyskytuje právě jednou.

3.5. Matice a vektory

Matice budeme označovat tučnými velkými písmeny anglické abecedy (např. $\mathbf{A}, \mathbf{B}$), vektory pomocí tučných malých písmen anglické abecedy (např. $\mathbf{a}, \mathbf{b}$). Matice $\mathbf{A}$ je *typu* $m \times n$, má-li m řádků a n sloupců, je-li $m = n$, hovoříme o *čtvercové matici* $\mathbf{A}$. *Délkou vektoru* $\mathbf{b}$ rozumíme počet jeho složek.

Je-li matice $\mathbf{A}$ typu $m \times n$ a jsou-li $i \in \{1,\ldots,m\}$ a $j \in \{1,\ldots,n\}$, pak pomocí $\mathbf{A}_{i,j}$ označíme prvek matice $\mathbf{A}$ na i-tém řádku a j-tém sloupci. Je-li $\mathbf{b}$ vektor délky n a je-li $i \in \{1,\ldots,n\}$, pak pomocí $\mathbf{b}_i$ označíme prvek vektoru $\mathbf{b}$ na pozici i.

Část II.

Vyčíslitelnost

4. Výpočetní modely

V této kapitole si představíme několik různých výpočetních modelů a ukážeme si jejich ekvivalenci.

4.1. Turingovy stroje

Model Turingova stroje byl poprvé navržen Alanem Turingem v roce 1936 v článku [10]. Turingovy stroje budou naším hlavním modelem, ke kterému se uchýlíme vždy, budeme-li potřebovat napsat něco formálně. Budeme jej tedy používat zejména v definicích základních pojmů v části věnující se teorii vyčíslitelnosti (např. pojmů rozhodnutelnosti či částečné rozhodnutelnosti, převoditelnosti atd.), tak v části věnující se teorii složitosti (například v definici základních tříd složitosti, tříd P, NP a dalších). Je proto přirozené začít popisem právě tohoto modelu.

4.1.1. Definice Turingova stroje

V této kapitole zavedeme model Turingova stroje spolu se souvisejícími pojmy, jež budeme dále potřebovat.

Definice 4.1.1 (Turingův stroj) (*Jednopáskový deterministický*) *Turingův stroj* M je pětice

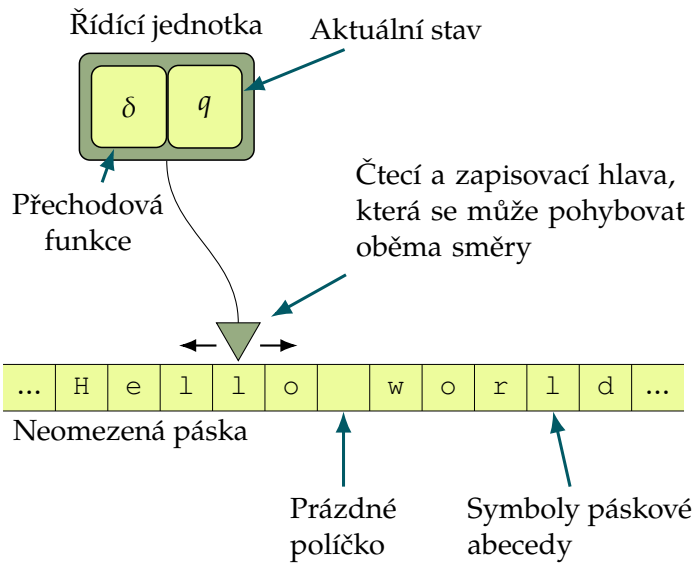
$$M = (Q, \Sigma, \delta, q_0, F),$$

kde

- Q je konečná množina stavů,
- Σ je konečná pásková abeceda, která obsahuje znak λ pro prázdné políčko,
- $\delta : Q \times \Sigma \to Q \times \Sigma \times \{R, N, L\} \cup \{\perp\}$ je přechodová funkce, kde $\perp$ označuje nedefinovaný přechod,
- $q_0 \in Q$ je počáteční stav a
- $F \subseteq Q$ je množina přijímajících stavů.

Turingův stroj (TS) sestává z *řídící jednotky*, *obousměrně potenciálně neomezené pásky* a *hlavy* pro čtení a zápis, která se může pohybovat po pásce oběma směry. ◂

Struktura Turingova stroje je zobrazena na obrázku 4.1.



Obrázek 4.1.: Struktura Turingova stroje.

Definice 4.1.2 (Displej, konfigurace a výpočet Turingova stroje)
Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je Turingův stroj. Dvojici (q, a) stavu řídící jednotky $q \in Q$ a čteného znaku na pásce $a \in \Sigma$, na základě níž se Turingův stroj rozhoduje, kterou instrukci použít pro další krok, říkáme *displej*. *Konfigurace* zachycuje aktuální informaci o stavu výpočtu Turingova stroje. Skládá se ze *stavu řídící jednotky*, *slova na pásce* (od nejlevějšího do nejpravějšího neprázdného políčka) a *pozice hlavy na pásce* (v rámci slova na této pásce).

Výpočet zahajuje TS M v *počáteční konfiguraci*, tedy v počátečním stavu se vstupním slovem zapsaným na pásce a hlavou nad nejlevějším symbolem vstupního slova. Pokud se M nachází ve stavu $q \in Q$ a pod hlavou je symbol $a \in \Sigma$ (tedy (q, a) je aktuální displej Turingova stroje M), pak *krok výpočtu* probíhá následovně:

1. Je-li $\delta(q, a) = \perp$, výpočet M končí,

2. Je-li $\delta(q, a) = (q', a', Z)$, kde $q' \in Q, a' \in \Sigma$ a $Z \in \{L, N, R\}$, přejde M do stavu q', zapíše na pozici hlavy symbol a' a pohne hlavou doleva (pokud $Z = L$), doprava (pokud $Z = R$), nebo hlava zůstane stát (pokud $Z = N$). ◂

Z popisu výpočtu Turingova stroje vidíme, že tabulka přechodové funkce popisuje, co se má stát v každé konfiguraci, nebo přesněji řečeno při daném displeji. Způsob programování Turingova stroje má tedy blízko k deklarativním jazykům.

Použití přechodové funkce δ rozšíříme i na konfigurace. Je-li K konfigurace Turingova stroje M, pak $\delta(K)$ označuje konfiguraci, která vznikne aplikací přechodové funkce δ na konfiguraci K, na základě displeje, který je v konfiguraci K obsažen. Hodnota $\delta(K) = \perp$,

pokud je hodnota δ nedefinovaná pro displej obsažený v konfiguraci K. O konfiguraci K, pro kterou platí, že $\delta(K) = \perp$, řekneme, že je *koncová*.

Na výpočet Turingova stroje lze také nahlížet jako na posloupnost konfigurací $K_0, \ldots, K_t$, kde K_0 je počáteční konfigurace, pro každou konfiguraci K_i, $i \in \{0, \ldots, t-1\}$ platí, že $K_{i+1} = \delta(K_i)$, a $\delta(K_t) = \perp$, tedy K_t je koncová konfigurace.

Definice 4.1.3 (Jazyk přijímaný Turingovým strojem) TS M *přijímá slovo* w, pokud výpočet M se vstupem w skončí a M se po ukončení výpočtu nachází v přijímajícím stavu. TS M *odmítá slovo* w, pokud výpočet M nad vstupem w skončí a M se po ukončení výpočtu nenachází v přijímajícím stavu. Fakt, že výpočet M nad vstupním slovem w skončí, označíme pomocí $M(w)\downarrow$ a řekneme, že výpočet *konverguje*. Fakt, že výpočet M nad vstupním slovem w nikdy neskončí, označíme pomocí $M(w)\uparrow$ a řekneme, že výpočet *diverguje*.

Množinu slov, která přijímá Turingův stroj M označíme pomocí $L(M)$, zveme ji také *jazykem přijímaným Turingovým strojem* M. Řekneme, že Turingův stroj M *rozhoduje* jazyk L, pokud $L = L(M)$ a M se zastaví nad každým vstupem. ◂

Příklad 4.1.4

Popišme si například Turingův stroj M, který rozhoduje jazyk

$$\mathrm{PAL} = \left\{ w = w^R \mid w \in \{\mathtt{a},\mathtt{b}\}^* \right\}, \tag{4.1}$$

tj. jazyk palindromů.[a] Při konstrukci M vyjdeme z toho, že slovo w délky $k = |w|$ je palindrom, pokud $w[1] = w[k]$ a slovo $w[2]\ldots w[k-1]$ je rovněž palindrom. Prázdný řetězec je též palindromem. Turingův stroj, který zkonstruujeme, bude implementovat následující algoritmus:

1: Je-li vstup prázdný, přijmi.
2: Zapamatuj si ve stavu první znak a jdi na konec vstupu.
3: Liší-li se poslední znak od toho, který je uložen ve stavu, odmítni.
4: Smaž poslední znak a vrať se na počátek vstupu.
5: Smaž první znak (pokud nějaký zbyl) a přejdi na začátek zbylého vstupu.
6: **goto** 1

Při implementaci algoritmu na Turingovu stroji je nejpodstatnějším krokem sestrojení přechodové funkce. Z ní potom můžeme vyčíst, které stavy a znaky páskové abecedy potřebujeme. Počátečním stavem bude q_0. Turingův stroj bude mít jediný přijímající stav q_1. Z tohoto stavu nepovedou již žádné instrukce, a tak změna stavu na q_1 odpovídá přímo přijetí. Tabulka 4.1 popisuje přechodovou funkci konstruovaného Turingova stroje.

Tabulka 4.1.: Přechodová funkce stroje M.

	q, c	$\rightarrow$	q', c', Z
1.	q_0, λ	$\rightarrow$	q_1, λ, N
2.	$q_0, \mathtt{a}$	$\rightarrow$	$q_2, \mathtt{a}, R$
3.	$q_0, \mathtt{b}$	$\rightarrow$	$q_3, \mathtt{b}, R$
4.	$q_2, \mathtt{a}$	$\rightarrow$	$q_2, \mathtt{a}, R$
5.	$q_2, \mathtt{b}$	$\rightarrow$	$q_2, \mathtt{b}, R$
6.	q_2, λ	$\rightarrow$	q_4, λ, L
7.	$q_3, \mathtt{a}$	$\rightarrow$	$q_3, \mathtt{a}, R$
8.	$q_3, \mathtt{b}$	$\rightarrow$	$q_3, \mathtt{b}, R$
9.	q_3, λ	$\rightarrow$	q_5, λ, L
10.	$q_4, \mathtt{a}$	$\rightarrow$	q_6, λ, L
11.	$q_5, \mathtt{b}$	$\rightarrow$	q_6, λ, L
12.	$q_6, \mathtt{a}$	$\rightarrow$	$q_6, \mathtt{a}, L$
13.	$q_6, \mathtt{b}$	$\rightarrow$	$q_6, \mathtt{b}, L$
14.	q_6, λ	$\rightarrow$	q_7, λ, R
15.	$q_7, \mathtt{a}$	$\rightarrow$	q_0, λ, R
16.	$q_7, \mathtt{b}$	$\rightarrow$	q_0, λ, R
17.	q_7, λ	$\rightarrow$	q_1, λ, N

Nyní položíme $M = (Q, \Sigma, \delta, q_0, F)$, kde

- $Q = \{q_0, q_1, \ldots, q_7\}$,
- $\Sigma = \{\mathtt{a}, \mathtt{b}, \lambda\}$,
- δ je určená tabulkou 4.1.
- $F = \{q_1\}$.

Práci Turingova stroje M nad vstupem w lze slovy popsat takto: Pokud je vstup prázdný, je vstup přijat. V opačném případě si M zapamatuje první symbol, ale zatím jej nechá beze změny. Přechodem do stavu q_2 si M pamatuje, že prvním znakem je a, přechodem do q_3 si M pamatuje, že prvním znakem je b. Následuje přechod na konec vstupu, tedy až k prvnímu prázdnému políčku. Na prvním prázdném políčku učiní M krok hlavou vlevo buď do stavu q_4, pokud zapamatovaný znak je a, nebo do stavu q_5, pokud zapamatovaný znak je b. Zde dojde ke kontrole posledního znaku (který může být současně i prvním v případě vstupu délky 1). Všimněme si, že k odmítnutí dojde ve stavu q_4 při čtení znaku b, neboť pro tento případ není hodnota přechodové funkce definovaná. Podobně dojde k odmítnutí ve stavu q_5 při čtení znaku a, neboť pro tento případ není hodnota přechodové funkce definovaná. Poslední znak je smazán a pomocí stavu q_6 dojde k pohybu hlavy na začátek pásky. Zde je ve stavu q_7 ověřeno, zda ještě řetězec obsahuje první znak. Pokud ano, je smazán a hlava se pohne vpravo na předpokládaný počátek zbylého, v tomto případě dojde k zahájení další smyčky, přechodem do stavu q_0. Pokud již je řetězec prázdný, přejde M ze stavu q_7 do přijímajícího stavu q_1.

[a] Poznamenejme, že PAL je bezkontextový jazyk a je pro něj možné zkonstruovat nedeterministický zásobníkový automat. Všimněme si, že zatímco zásobníkovému automatu stačí přečíst vstup jen jednou, jednopáskový Turingův stroj potřebuje projít vstupem w délky $2n$ až n-krát.

V příkladu 4.1.4 jsme tiše předpokládali, že vstupní slovo se skládá pouze ze znaků 0 a 1 a neobsahuje prázdná políčka. To je nutné proto, abychom poznali konec vstupu. Kdybychom povolili neomezený výskyt prázdných políček, tak by námi zkonstruovaný Turingův stroj ve skutečnosti selhal na slovech typu $\mathtt{aa}\lambda^k\mathtt{a}$ pro libovolné $k > 0$, tato slova by byla přijata, i když do jazyka palindromů nepatří. Ve skutečnosti by Turingův stroj nebyl schopen podobné případy rozpoznat, pokud by neznal dopředu horní odhad na hodnotu k. Pokud by totiž přečetl několik znaků λ, nemohl by vědět, zda je už na konci vstupu, nebo jen ještě nepřeskočil dostatek vložených mezer. Mohli bychom sice používat prázdných políček například k oddělení jednotlivých částí vstupu, ale k tomuto účelu můžeme klidně použít i jiný znak. Při definici Turingova stroje se často rozlišují vstupní, pracovní a výstupní abecedy. V našem případě by tedy vstupní abeceda obsahovala pouze znaky 0 a 1, pracovní by navíc obsahovala znak prázdného políčka λ. My jsme v definici vstupní abecedu nezaváděli, z kontextu bude vždy zřejmé, jaké znaky jsou očekávány na vstupu. Navíc přijmeme omezení dané následující úmluvou.

Úmluva 4.1.5 *Vždy předpokládáme, že vstupní řetězce neobsahují znak λ prázdného políčka.*[1]

Zápis přechodové funkce si trochu zjednodušíme a zejména zestručníme na základě následující poznámky:

Poznámka 4.1.6 *Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je Turingův stroj. Abychom zápis přechodové funkce δ zestručnili, použijeme proměnné v zápisech instrukcí pro symboly abecedy Σ. Je-li v zápisu instrukce použitá proměnná α, pak to znamená, že příslušná instrukce má kopie pro všechny možné hodnoty α, které jsou uvedeny v poznámce u daného řádku tabulky. Pokud je v instrukci použito více proměnných, pak daná instrukce má kopie pro všechny kombinace hodnot těchto proměnných.*

Příklad 4.1.7

Tabulka 4.2 obsahuje stručnější popis přechodové funkce Turingova stroje M zkonstruovaného v příkladu 4.1.4 s pomocí proměnných zavedených v poznámce 4.1.6.

Tabulka 4.2.: Stručný zápis přechodové funkce stroje M.

	q, c	$\rightarrow$	q', c', Z	Hodnoty proměnných
1.	q_0, λ	$\rightarrow$	q_1, λ, N	
2.	$q_0, \mathtt{a}$	$\rightarrow$	$q_2, \mathtt{a}, R$	
3.	$q_0, \mathtt{b}$	$\rightarrow$	$q_3, \mathtt{b}, R$	
4.	q_2, α	$\rightarrow$	q_2, α, R	$\alpha \in \{\mathtt{a}, \mathtt{b}\}$
5.	q_2, λ	$\rightarrow$	q_4, λ, L	
6.	q_3, α	$\rightarrow$	q_3, α, R	$\alpha \in \{\mathtt{a}, \mathtt{b}\}$
7.	q_3, λ	$\rightarrow$	q_5, λ, L	
8.	$q_4, \mathtt{a}$	$\rightarrow$	q_6, λ, L	
9.	$q_5, \mathtt{b}$	$\rightarrow$	q_6, λ, L	
10.	q_6, α	$\rightarrow$	q_6, α, L	$\alpha \in \{\mathtt{a}, \mathtt{b}\}$
11.	q_6, λ	$\rightarrow$	q_7, λ, R	
12.	q_7, α	$\rightarrow$	q_0, λ, R	$\alpha \in \{\mathtt{a}, \mathtt{b}\}$
13.	q_7, λ	$\rightarrow$	q_1, λ, N	

V případě jazyka přijímaného Turingovým strojem M nás zajímá stav, ve kterém Turingův stroj skončí. Každý Turingův stroj však rovněž počítá funkci z množiny řetězců do množiny řetězců.

[1] Turingovy stroje v této úmluvě nezmiňujeme úmyslně, budeme toto omezení předpokládat i v případě dalších výpočetních modelů.

Definice 4.1.8 (Turingovsky vyčíslitelné (řetězcové) funkce)
Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je Turingův stroj. Pomocí f_M označíme funkci, kterou M počítá, tedy funkci $f_M : \Sigma^* \to \Sigma^*$, jejíž doména je tvořena řetězci $w \in \Sigma^*$, pro které je $M(w)\downarrow$, přičemž pro každý řetězec w z domény f_M platí, že výpočet $M(w)$ skončí s tím, že na výstupní pásce[2] je napsáno slovo $f_M(w)$. O funkci f, která je počítaná nějakým Turingovým strojem řekneme, že je *turingovsky vyčíslitelná*. ◂

Turingovu stroji, který je použit pro přijímání slov z daného jazyka, se také někdy říká *akceptor* a Turingovu stroji, který je použit pro počítání řetězcové funkce se někdy říká *transducer*. V obou případech se však jedná o týž model Turingova stroje, záleží jen na tom, zajímá-li nás stav na konci výpočtu, nebo obsah pásky na konci výpočtu.

4.1.2. Varianty Turingových strojů

Model Turingova stroje má řadu variant, které jsou ekvivalentní co do výpočetní síly. Pro nás bude základní variantou jednopáskový deterministický Turingův stroj s oboustraně potenciálně nekonečnou páskou tak, jak byl popsán v kapitole 4.1.1. Některé varianty si popíšeme v následujících podkapitolách. Zaměříme se na varianty, které budou užitečné ve zbytku tohoto textu. Nejpodrobněji se budeme věnovat vícepáskovým Turingovým strojům a nedeterministickým Turingovým strojům. K oběma modelům se později vrátíme ještě v části věnované teorii složitosti, kde zejména nedeterministické Turingovy stroje hrají důležitou roli.

Turingovy stroje s páskou neomezenou jen doprava

V první variantě, kterou si popíšeme, uvažujeme Turingovy stroje, jejichž páska je neomezená pouze doprava. To znamená, že takto omezený Turingův stroj nesmí pohnout hlavou nalevo od její počáteční pozice.

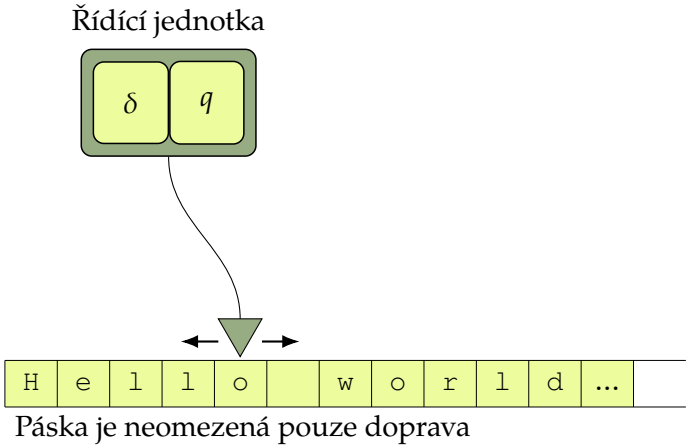
Libovolný Turingův stroj lze převést na Turingův stroj, který má pásku neomezenou pouze doprava, zformulujeme si toto tvrzení jako větu, jejíž důkaz ovšem ponecháváme čtenáři jako jednoduché cvičení.

Věta 4.1.9 *Ke každému Turingovu stroji M (jehož páska je neomezená v obou směrech) existuje Turingův stroj M', jehož páska je neomezená pouze doprava a který simuluje práci M, přijímá týž jazyk jako M a počítá touž funkci jako M.*

Vícepáskové Turingovy stroje

V případě více pásek je obvykle jedna páska určena jako vstupní a jedna jako výstupní, přičemž může jít i o jednu pásku, která plní obě role. Výstupní páska je navíc zajímavá jen v případě Turingova stroje, který počítá nějakou funkci. Ostatní pásky jsou pracovní. Často navíc platí omezení, že na vstupní pásku nelze zapisovat, ale lze z ní jen číst, podobně na výstupní pásku lze často pouze zapisovat a lze se na ní pohybovat jen jedním

[2] Zatím uvažujeme jen Turingovy stroje s jednou páskou, která je současně vstupní i výstupní, nicméně definice již počítá s možností mít pásek více.



Obrázek 4.2.: Struktura Turingova stroje s páskou neomezenou jen doprava.

směrem ve směru zápisu, tedy doprava. Tato omezení mají význam zejména v případě, kdy se začneme zabývat omezeným prostorem a zvláště tím, kolik pracovního prostoru potřebujeme ke zpracování vstupu. K tomu se vrátíme v kapitolách věnujícím se prostorové složitosti algoritmů. Počet pásek je pro daný vícepáskový Turingův stroj pevný a nezávislý na vstupu.

My si zde ukážeme, že každý vícepáskový Turingův stroj lze simulovat na jednopáskovém Turingovu stroji. To nám přinese zjednodušení v řadě situacích, kdy bude snazší popsat vícepáskový Turingův stroj pro daný účel, než jednopáskový. Zjednoduší to například konstrukci univerzálního Turingova stroje.

Zadefinujme si nyní model vícepáskového stroje formálně.

Definice 4.1.10 *k-páskový deterministický Turingův stroj M*, kde $k > 0$ je celočíselná konstanta, je pětice

$$M = (Q, \Sigma, \delta, q_0, F),$$

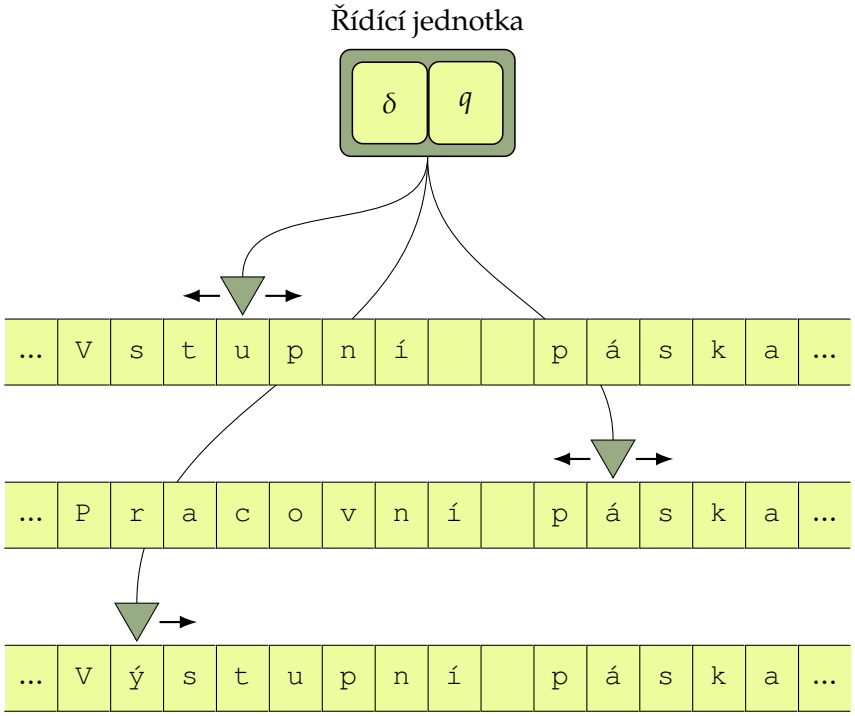
kde

- Q je konečná *množina stavů*,
- Σ je konečná *pásková abeceda*, která obsahuje znak λ pro prázdné políčko,
- $\delta : Q \times \Sigma^k \to Q \times \Sigma^k \times \{R, N, L\}^k \cup \{\perp\}$ je přechodová funkce, kde $\perp$ označuje nedefinovaný přechod,
- $q_0 \in Q$ je počáteční stav a
- $F \subseteq Q$ je množina přijímajících stavů.

Výpočet k-páskového TS probíhá podobně jako výpočet jednopáskového TS s těmito rozdíly:

- Na počátku je vstup napsán na jedné pásce, která je zvolena jako *vstupní*.
- Na konci je výstup uložen na jedné pásce, která je zvolena jako *výstupní*.
- Během kroku TS přečte symboly pod hlavami na všech páskách a postupuje podle příslušné instrukce (pokud taková existuje), v tom případě zapíše všemi hlavami současně nové symboly a vykoná pohyby hlav dané přechodovou funkcí, hlavy se pohybují nezávisle na sobě různými směry. ◂

Na obrázku 4.3 vidíme strukturu vícepáskového Turingova stroje.



Obrázek 4.3.: Struktura 3-páskového Turingova stroje s jednou vstupní, jednou pracovní a jednou výstupní páskou. Výstupní páska je určena jen pro zápis a hlava se může pohybovat jen vpravo.

Příklad 4.1.11

Uvažme opět jazyk palindromů, který je nám již známý z příkladu 4.1.4.

$$\mathrm{PAL} = \left\{ w = w^R \mid w \in \{\mathtt{a},\mathtt{b}\}^* \right\},$$

Vzpomeňme si, že jednopáskový Turingův stroj M přijímající jazyk palindromů, který jsme popsali v příkladu 4.1.4 měl tu nevýhodu, že musel vstupem délky $2n$ projít až n-krát. Máme-li k dispozici dvoupáskový Turingův stroj, stačí projít vstupní slovo jen čtyřikrát. Zkonstruujeme dvoupáskový Turingův stroj M', který bude rozhodovat jazyk PAL. První páska stroje M' bude vstupní, druhá pracovní. Stroj bude při své práci postupovat podle následujícího algoritmu:

1: Jdi na konec vstupu.
2: Vracej se na vstupní pásce hlavou zpět na začátek vstupu a současně kopíruj vstupní řetězec na pracovní pásku, psaný pozpátku.
3: Ve chvíli, kdy dojdeš na začátek vstupu, vrať se na pracovní pásce zpět na začátek vstupu.
4: Pohybuj hlavami na vstupní a pracovní pásce současně a přitom kontroluj, že řetězce na obou páskách jsou shodné.
5: Pokud jsou oba řetězce shodné, přijmi, jinak odmítni.

Nyní popíšeme přechodovou funkci Turingova stroje M'. Počátečním stavem bude q_0 a jediným přijímajícím stavem bude q_1. Přechodová funkce je zobrazena v tabulce 4.3. Používáme stručný zápis s proměnnými zavedený v poznámce 4.1.6.

Tabulka 4.3.: Přechodová funkce stroje M'.

	q, c_1, c_2	$\to$	q', c_1', c_2', Z_1, Z_2	Hodnoty proměnných
1.	q_0, α, λ	$\to$	$q_0, \alpha, \lambda, R, N$	$\alpha \in \{\mathtt{a},\mathtt{b}\}$
2.	q_0, λ, λ	$\to$	$q_2, \lambda, \lambda, L, N$	
3.	q_2, α, λ	$\to$	$q_2, \alpha, \alpha, L, R$	$\alpha \in \{\mathtt{a},\mathtt{b}\}$
4.	q_2, λ, λ	$\to$	$q_3, \lambda, \lambda, N, L$	
5.	q_3, λ, α	$\to$	$q_3, \lambda, \alpha, N, L$	$\alpha \in \{\mathtt{a},\mathtt{b}\}$
6.	q_3, λ, λ	$\to$	$q_4, \lambda, \lambda, R, R$	
7.	q_4, α, α	$\to$	$q_4, \alpha, \alpha, R, R$	$\alpha \in \{\mathtt{a},\mathtt{b}\}$
8.	q_4, λ, λ	$\to$	$q_1, \lambda, \lambda, N, N$	

Nyní položíme $M' = (Q, \Sigma, \delta, q_0, F)$, kde

- $Q = \{q_0, q_1, \ldots, q_4\}$,
- $\Sigma = \{0, 1, \lambda\}$,
- δ je určená tabulkou 4.3.
- $F = \{q_1\}$.

Popišme si práci Turingova stroje M' nad vstupem w slovy. Instrukce 1 a 2 přesunou hlavu na poslední symbol vstupu. Instrukce dané řádkou 3 překopírují (ve stavu q_2) vstupní slovo na pracovní pásku, ovšem psané opačně. Hlava na vstupní pásce se pohybuje zpět k začátku, na pracovní pásce se pohybuje hlava doprava. Na pracovní pásku je tak zapsáno slovo w^R. Kopírování je ukončeno instrukcí 4, kdy se přejde do stavu q_3, určeného pro návrat hlavy na pracovní pásce na začátek. V tuto chvíli je hlava na vstupní pásce na začátku a na pracovní pásce na konci řetězce, je potřeba se hlavou na pracovní pásce vrátit zpět na začátek, k čemuž slouží instrukce dané řádkem 5. Instrukce 6 ukončuje návrat hlavy a pohne oběma hlavami na první znaky řetězců na obou páskách. Instrukce dané řádkem 7 pak provádějí kontrolu shody, v úspěšném případě se hlavy dostanou na prázdné symboly za řetězci w a w^R na obou páskách a v instrukci 8 dojde k přijetí.

Naším dalším cílem je ukázat, že každý k-páskový Turingův stroj M má svůj ekvivalentní jednopáskový Turingův stroj M'. V tomto smyslu budeme hovořit o tom, že M' *simuluje práci Turingova stroje* M. Pojem simulace nebudeme zavádět příliš formálně, nicméně budeme jej používat v následujícím smyslu. Je-li M_1 Turingův stroj, který simuluje TS M_2, pak předpokládáme, že na některých páskách M_1 se nějakým způsobem postupně objevuje obsah pásek stroje M_2 a že z práce M_1 lze vyčíst výpočet stroje M_2, přičemž simulace jednoho kroku stroje M_2 může vyžadovat provedení několika kroků stroje M_1.

Někdy budeme uvažovat situaci, kdy k jednomu TS M_3 s nějakou vlastností (například k-páskovému) zkonstruujeme jiný TS M_4 s jinou vlastností (například jednopáskový), který přijímá týž jazyk nebo počítá touž funkci, v tom případě nepožadujeme, aby výpočet M_3 probíhal týmž způsobem jako výpočet M_4. Jde o obecnější princip, než simulace, ačkoli simulace stroje M_4 strojem M_3 může být jedním způsobem, pomocí nějž dosáhneme toho, aby oba stroje přijímaly týž jazyk.

Věta 4.1.12 *Nechť $k \geq 2$ je přirozené číslo. Ke každému k-páskovému Turingovu stroji M existuje jednopáskový Turingův stroj M', který simuluje práci M, přijímá týž jazyk jako M a počítá touž funkci jako M.*

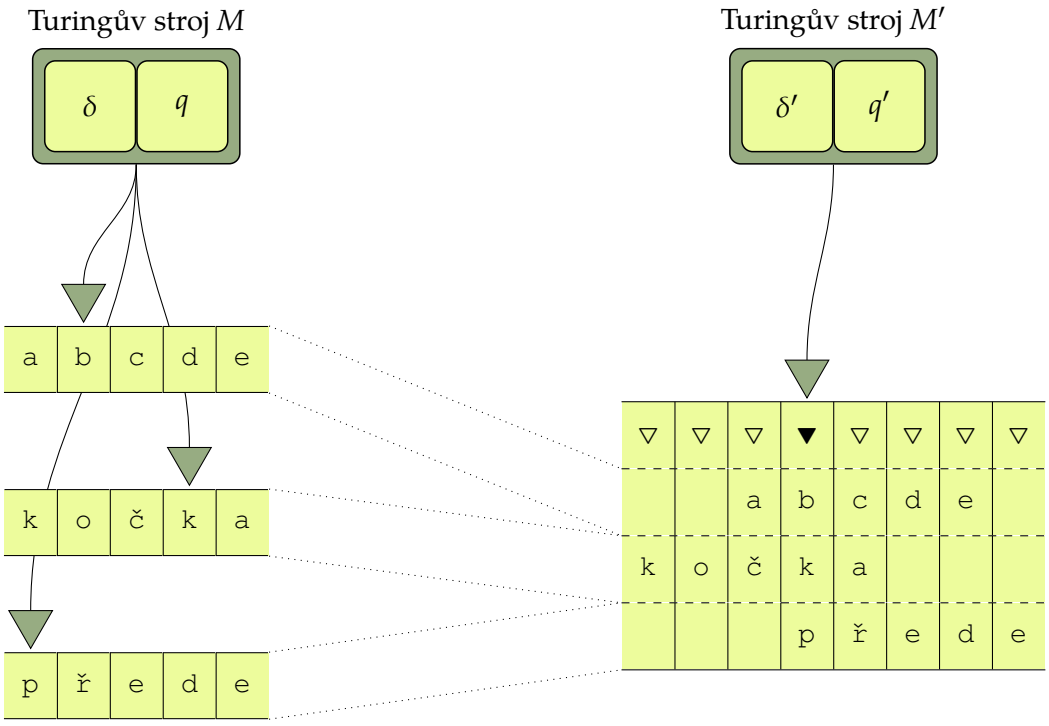
Důkaz: Podáme zde pouze ideu důkazu bez technických detailů. Předpokládejme, že $M = (Q, \Sigma, \delta, q_0, F)$ a že konstruovaný stroj $M' = (Q', \Sigma', \delta', q_0', F')$. Při konstrukci jednopáskového Turingova stroje M' musíme vyřešit zejména následující dva problémy:

1. Jak reprezentovat k pásek na jedné pásce a
2. jak si poradit s tím, že hlavy na k páskách stroje M se pohybují nezávisle na sobě.

Reprezentaci více pásek na jedné pásce vyřešíme tím, že budeme k políček, která jsou na páskách nad sebou, reprezentovat v jednom políčku pásky stroje M'. Můžeme si to představit tak, že páska stroje M' bude mít k *stop*. Stopa je v tomto smyslu podobná pásce, ovšem s tím rozdílem, že máme k dispozici jen jednu hlavu pro všechny stopy. Musíme si tedy poradit s tím, že ve stroji M máme k hlav a v stroji M' jen jednu. Při simulaci budeme zachovávat následující invariant:

Invariant I: Na začátku simulace kroku stroje M jsou pásky stroje M na pásce M' vždy zarovnané tak, aby čtené symboly byly pod sebou v jedné buňce více stopé pásky.

K tomu bude nutné posouvat obsahy stop při simulaci kroku stroje M tak, abychom dosáhli kýženého stavu. Pro posouvání budeme potřebovat značku ▼ na buňku s hlavou v místě, nad kterým mají být hlavy stroje M, aby M' po posunutí stopy věděl, na jakou buňku se má s hlavou vrátit. Struktura stroje M' je naznačena na obrázku 4.4. Popišme



Obrázek 4.4.: Reprezentace 3 pásek stroje M pomocí 4 stop na jediné pásce M'. Čárkovanou čarou jsou od sebe oddělené jednotlivé stopy na pásce M'. Obsahy pásek jsou na stopách zarovnané tak, aby hlavy stroje M byly pod sebou. Značka ▼ reprezentuje místo, nad kterým jsou hlavy stroje M.

si nyní konstrukci stroje M' podrobněji. Páskovou abecedu stroje M' položíme

$$\Sigma' = \Sigma \cup \left(\{ \triangledown, \blacktriangledown \} \times \Sigma^k \right).$$

Jedna buňka stroje M' tedy reprezentuje k buněk stroje M, která jsou na k páskách nad sebou. Navíc abeceda umožňuje rozlišit pomocí značek $\blacktriangledown$ a $\triangledown$ zda je nebo není nad danou buňkou hlava[3]. Stroj M' má vždy zformátovanou tu oblast pásky, která zachycuje obsahy pásek M. Na začátku má tato oblast délku vstupu, v průběhu simulace se může zformátovaná oblast zvětšovat podle toho, jak M využívá další prostor na páskách. Zformátovaná oblast pásky je ohraničena prázdnými políčky a je tedy vždy poznat, kde končí a začíná (uvažujeme případ $k > 1$, pro $k = 1$ je možné rovnou nechat stroj M beze změny). Stroj M' bude při své práci postupovat následujícím způsobem (lépe řečeno přechodová funkce δ' stroje M' je zkonstruována tak, aby implementovala následující kroky):

1. V rámci inicializace proběhne přeformátování vstupu do stop. Vstup je na vstupu předán pomocí symbolů v abecedě Σ. Stroj M' si přepíše vstup tak, že nejlevější buňku vstupu obsahující $a \in \Sigma$ přepíše na $\blacktriangledown a \lambda^{k-1}$ a další buňky vstupu, které obsahují symbol $a \in \Sigma$ přepíše na $\triangledown a \lambda^{k-1}$. To odpovídá tomu, že na počátku jsou pracovní pásky M prázdné, jsou tedy prázdné i odpovídající stopy. Hlava je nad nejlevější buňkou vstupu, proto je na ní zapsán symbol $\blacktriangledown$, zatímco nad dalšími buňkami je symbol $\triangledown$. Přechodová funkce δ' Turingova stroje M' bude obsahovat instrukce, které provedou tuto inicializaci.

2. Předpokládáme, že jednotlivé instrukce z přechodové funkce δ Turingova stroje M jsou očíslované. Místo i-té instrukce $\delta(q, a_1, \ldots, a_k) = (r, b_1, \ldots, b_k)$, kde $q, r \in Q$ a $a_1, \ldots, a_k, b_1, \ldots, b_k \in \Sigma$ přidáme posloupnost instrukcí do přechodové funkce δ', který provede simulaci této instrukce v stroji M'. Rozlišení toho, kterou posloupnost instrukcí potom M' zvolí, bude provedeno na základě displeje, δ' bude definovat přechod $\delta'(q, \blacktriangledown a_1 \ldots a_k) = (q^i, \blacktriangledown a_1 \ldots a_k, N)$, přičemž stavem q^i začíná posloupnost instrukcí simulující provedení i-té instrukce stroje M. Tato posloupnost se skládá z k fází. V j-té fázi pro $j = 1, \ldots, k$ je odsimulována instrukce na j-té stopě (odpovídající j-té pásce stroje M). Pokud i-tá instrukce na j-té pásce stroje M přikazuje pohnout hlavou vlevo, pak se celý obsah j-té stopy posune o jedno políčko vpravo. Pokud se má pohnout hlavou na j-té pásce M vpravo, pak se celý obsah j-té stopy posune o jedno políčko vlevo. Pokud má hlava zůstat na místě, ani stopa stroje M' se neposouvá. Při posouvání stopy o políčko vpravo nebo vlevo je využita značka $\blacktriangledown$. Má-li například dojít k posunu obsahu stopy vpravo, pak Turingův stroj M' dojde na začátek pásky, a cestou doprava posouvá políčko po políčku na této stopě o jedno vpravo, zatímco ostatní stopy ponechává beze změny. Pokud poslední buňka na stopě nebyla prázdná, je nutné přiformátovat jedno políčko na

[3] Je potřeba zmínit, že zde využíváme toho, že abeceda Turingova stroje může být libovolně velká a můžeme si ji tedy libovolně zvětšit, pokud zachováme to, že její velikost je pro daný Turingův stroj konstantní a není závislá na velikosti vstupu. Pokud bychom chtěli zachovat abecedu beze změny, mohli bychom reprezentovat k buněk stroje M pomocí $k + 1$ po sobě jdoucích políček.

pravém okraji. Posun vlevo probíhá obdobně. Po posunutí pásky M' díky značce ▼ pozná, kam se má vrátit. Po provedení poslední, tedy k-té fáze přejde M' do stavu r, v němž dojde podle čtené k-tice znaků k rozhodnutí, jaká posloupnost instrukcí se použije pro odsimulování další instrukce stroje M.

3. Za každou kombinaci stavu q a k-tice znaků $a_1, \ldots, a_k \in \Sigma$, pro které je $\delta(q, a_1, \ldots, a_k) = \perp$, přidáme do δ' přechod do počátečního stavu posloupnosti instrukcí provádějící překódávání obsahu pásky tak, aby na ní zůstal jen obsah výstupní pásky stroje M. □

Všimněme si, že jednopáskový Turingův stroj M' potřebuje k odsimulování provedení jedné instrukce stroje M až $\Theta(nk)$ instrukcí, kde n je délka vstupu.

Nedeterministické Turingovy stroje

Nedeterministický Turingův stroj rozšiřuje deterministický Turingův stroj o možnost činit několik rozhodnutí najednou. Základním modelem nedeterministického Turingova stroje pro nás bude stroj jednopáskový, i když budeme uvažovat i vícepáskové nedeterministické Turingovy stroje.

Definice 4.1.13 (Nedeterministický Turingův stroj) (*Jednopáskový*) *nedeterministický Turingův stroj* (*NTS*) je pětice $M = (Q, \Sigma, \delta, q_0, F)$, jejíž prvky mají týž význam jako v případě deterministického Turingova stroje s tím rozdílem, že přechodová funkce má tvar

$$\delta : Q \times \Sigma \to \wp(Q \times \Sigma \times \{R, N, L\}).$$

Jinými slovy přechodová funkce danému stavu a symbolu na pásce přiřazuje několik přechodů do jiných stavů se zápisem různých symbolů a s různými pohyby. V případě, že množina možných přechodů je prázdná, není přechod definován.

- Podobně jako v případě deterministického Turingova stroje rozšíříme použití přechodové funkce i na konfigurace, tedy $\delta(K)$ označuje množinu konfigurací, která vznikne aplikací přechodové funkce na konfiguraci K na základě displeje, který je v konfiguraci K obsažen.
- *Výpočet nedeterministického Turingova stroje nad vstupem* $x \in \Sigma^*$ je posloupnost konfigurací, která začíná počáteční konfigurací K_0 a pro každé dvě následující konfigurace K, K' v posloupnosti platí, že $K' \in \delta(K)$. V každém kroku si tedy nedeterministický Turingův stroj vybere, kterou z možností zvolí a tu aplikuje.
- Výpočet nedeterministického Turingova stroje je *konečný*, jde-li o konečnou posloupnost konfigurací $K_0, \ldots, K_t$ a pro poslední konfiguraci K_t platí, že $\delta(K_t) = \varnothing$.
- *Výpočet nedeterministického Turingova stroje* je přijímající, pokud je konečný a jeho poslední konfigurace je přijímající, tj. M je na konci v přijímajícím stavu.
- Řekneme, že nedeterministický Turingův stroj M *přijme slovo* x, pokud existuje výpočet nedeterministického Turingova stroje M nad x, který je přijímající.

- Jazyk slov přijímaných nedeterministickým Turingovým strojem M označíme pomocí $L(M)$. ◂

Nedeterministický Turingův stroj se od deterministického liší jen tím, že v dané chvíli umožňuje použít víc přechodů. Náš způsob definice výpočtu nedeterministického Turingova stroje předpokládá, že nedeterministický Turingův stroj se nachází vždy v jedné konfiguraci, pokud má z této konfigurace na výběr několik možných přechodů, jeden z nich si vybere, tedy „uhodne" jej. A pokud nějaká posloupnost těchto výběrů či „uhodnutí" vede k přijetí, řekneme, že nedeterministický Turingův stroj daný vstup přijal. Při definici výpočtu nedeterministického Turingova stroje M však narozdíl od deterministického Turingova stroje nepožadujeme, aby poslední konfigurace K_t v posloupnosti byla koncová, tedy aby již nebylo možno dále pokračovat aplikací přechodové funkce na konfiguraci K_t. Ponecháme na nedeterministické volbě stroje M, zda v dané konfiguraci zastaví výpočet či bude dál pokračovat. Obvykle budeme předpokládat, že přijímající konfigurace (tj. konfigurace, v nichž je M v přijímajícím stavu) koncové jsou.

Výpočet nedeterministického Turingova stroje si však lze představit i tak, že v každém kroku nedeterministický Turingův stroj použije všechny možné přechody a nachází se tak vždy ve všech dostupných konfiguracích najednou, podobně si obvykle představujeme i práci nedeterministického konečného automatu. V obou případech si také můžeme výpočet nedeterministického Turingova stroje reprezentovat pomocí stromu výpočtu.

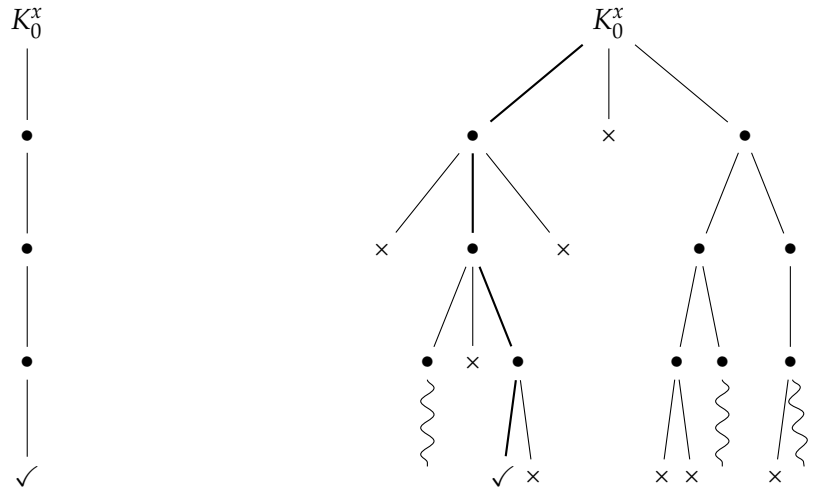
Definice 4.1.14 (Strom výpočtu nedeterministického Turingova stroje)
Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je nedeterministický Turingův stroj. Strom výpočtu stroje M nad vstupem $x \in \Sigma^*$ je dvojice $T, \mathcal{L}$, kde

- $T = (V, E)$ je orientovaný zakořeněný strom, přičemž V a E jsou (potenciálně i nekonečné) množiny vrcholů a hran. Hrany jsou orientované od kořene směrem k listům.
- $\mathcal{L}$ je funkce, která každému vrcholu $v \in V$ přiřazuje konfiguraci stroje M a která splňuje následující podmínky:
  - Kořeni $r \in V$ stromu T přiřazuje funkce $\mathcal{L}$ počáteční konfiguraci výpočtu (tj. na pásce je zapsán vstup x, stroj M je v počátečním stavu q_0 a hlava je nad nejlevějším symbolem vstupu).
  - Pokud $(u, v) \in E$ je hrana, pak $\mathcal{L}(v) \in \delta(\mathcal{L}(u))$.
  - Pokud $u \in V$ je uzel stromu, pak pro každou konfiguraci $K \in \delta(\mathcal{L}(u))$ existuje vrchol $v \in V$, pro který platí, že $\mathcal{L}(v) = K$ a $(u, v) \in E$. ◂

Podle naší definice je výpočet nedeterministického stroje M nad vstupem x posloupností konfigurací, což odpovídá jedné větvi stromu výpočtu. V případě, že M je deterministický Turingův stroj, strom výpočtu tvoří cestu.

Příklad 4.1.15

Vzpomeňme si, že jsme již zkonstruovali dva deterministické stroje, které testují,



Strom deterministického výpočtu

Strom nedeterministického výpočtu

Obrázek 4.5.: Nalevo je naznačen strom výpočtu deterministického Turingova stroje, tedy cesta. Napravo je naznačen strom výpočtu nedeterministického stroje. Tučně je vyznačena větev ukončená symbolem ✓ reprezentujícím přijímající konfigurací, tedy přijímající výpočet. Větve ukončené symboly × jsou ukončené, ale nikoli přijímající. Vlnovky na koncích dalších větví naznačují, že větve dále pokračují a jsou neukončené. K_0^x v kořeni obou stromů označuje počáteční konfiguraci při výpočtu nad vstupem x.

zda slovo patří do jazyka palindromů

$$\mathrm{PAL} = \left\{ w = w^R \mid w \in \{\mathtt{a},\mathtt{b}^*\} \right.,$$

jednopáskový stroj v příkladu 4.1.4 a dvoupáskový stroj v příkladu 4.1.11. Uvažme nyní jazyk

$$\mathrm{PAL}^2 = \mathrm{PAL} \cdot \mathrm{PAL} = \left\{ uv \mid u, v \in \{\mathtt{a},\mathtt{b}\}^* \wedge u = u^R \wedge v = v^R \right\}$$

Jazyk PAL^2 tedy obsahuje řetězce, jež vzniknou konkatenací dvou palindromů za sebe.

Rozmysleme si nejprve, jak by mohl vypadat deterministický Turingův stroj M přijímající jazyk PAL^2. Pro dané vstupní slovo $x \in \{\mathtt{a},\mathtt{b}\}^*$ délky $n = |x|$ by M musel uvážit všechna místa dělení, tedy pro každé $j = 0, \ldots, n$ by musel M uvážit dvojici slov $u = x[1] \ldots x[j]$, $v = x[j+1] \ldots x[n]$ (přičemž $u = \varepsilon$ pro $j = 0$ a $v = \varepsilon$ pro $j = n$) a otestovat, zda jde o palindromy. Test toho, zda je slovo palindrom by bylo nutné spustit až $2(n+1)$-krát.

Popišme nyní nedeterministický Turingův stroj $N = (Q, \Sigma, \delta, q_0, F)$, který přijímá právě jazyk PAL^2. Nedeterminismus nám umožňuje „uhodnout“ správné rozdělení vstupního slova x na dvě části u a v a poté jen otestovat, zda obě části jsou palindromy. Předpokládejme pro jednoduchost, že N je třípáskový Turingův stroj. Vstup je na začátku zapsán na první pásce, která slouží jen ke čtení vstupu a N na ni nezapisuje. Stroj N postupuje následujícím způsobem:

1. Kopíruj vstupní slovo na druhou pásku.
2. V nějakou chvíli se (nedeterministicky) kopírování zastaví.
3. Otestuj, zda slovo na druhé pásce je palindrom, pokud ne, výpočet končí odmítnutím. K tomuto testu jsou využity obě pracovní pásky a je použit postup popsaný v příkladu 4.1.11.
4. Vymaž obě pracovní pásky.
5. Okopíruj zbytek vstupního slova na druhou pásku.
6. Otestuj, zda slovo na druhé pásce je palindrom, pokud ne, výpočet končí odmítnutím.
7. Přijmi.

Stroj N tedy při jednom výpočtu nad vstupem x vyzkouší jednu možnost rozdělení slova x na podslova u a v. Slovo x patří do slova PAL^2, právě když jeden z těchto výpočtů uspěje. Nedeterministismus je zde obsažen pouze v kroku 2, tedy v tom,

kdy N ukončí první podslovo u. Tabulka 4.4 popisuje přechodovou funkci stroje N, používáme stručný způsob zápisu s proměnnými, zavedený v poznámce 4.1.6.

Tabulka 4.4.: Přechodová funkce nedeterministického Turingova stroje N.

	q, c_1, c_2, c_3	$\rightarrow$	$q', c_1', c_2', c_3', Z_1, Z_2, Z_3$	Hodnoty proměnných
	Kopírování vstupu s nedeterministickým ukončením			
1.	$q_0, \alpha, \lambda, \lambda$	$\rightarrow$	$q_0, \alpha, \alpha, \lambda, R, R, N$	$\alpha \in \{\mathtt{a}, \mathtt{b}\}$
2.	$q_0, \alpha, \lambda, \lambda$	$\rightarrow$	$q_2, \alpha, \lambda, \lambda, N, L, N$	$\alpha \in \{\mathtt{a}, \mathtt{b}\}$
3.	$q_0, \lambda, \lambda, \lambda$	$\rightarrow$	$q_2, \lambda, \lambda, \lambda, N, L, N$	
	Zde začíná kontrola palindromu na druhé pásce			
4.	$q_2, \alpha, \beta, \lambda$	$\rightarrow$	$q_2, \alpha, \beta, \beta, N, L, R$	
5.	$q_2, \alpha, \lambda, \lambda$	$\rightarrow$	$q_3, \alpha, \lambda, \lambda, N, N, L$	$\alpha \in \{\mathtt{a}, \mathtt{b}, \lambda\}$
6.	$q_3, \alpha, \lambda, \beta$	$\rightarrow$	$q_3, \alpha, \lambda, \beta, N, N, L$	$\alpha \in \{\mathtt{a}, \mathtt{b}, \lambda\}, \beta \in \{\mathtt{a}, \mathtt{b}\}$
7.	$q_3, \alpha, \lambda, \lambda$	$\rightarrow$	$q_4, \alpha, \lambda, \lambda, N, R, R$	$\alpha \in \{\mathtt{a}, \mathtt{b}, \lambda\}$
8.	$q_4, \alpha, \beta, \beta$	$\rightarrow$	$q_4, \alpha, \lambda, \lambda, N, R, R$	$\alpha \in \{\mathtt{a}, \mathtt{b}, \lambda\}, \beta \in \{\mathtt{a}, \mathtt{b}\}$
9.	$q_4, \alpha, \lambda, \lambda$	$\rightarrow$	$q_5, \alpha, \lambda, \lambda, N, N, N$	$\alpha \in \{\mathtt{a}, \mathtt{b}, \lambda\}$
	Zde začíná kopírování zbytku vstupu na druhou pásku			
10.	$q_5, \alpha, \lambda, \lambda$	$\rightarrow$	$q_5, \alpha, \alpha, \lambda, R, R, N$	$\alpha \in \{\mathtt{a}, \mathtt{b}\}$
11.	$q_5, \lambda, \lambda, \lambda$	$\rightarrow$	$q_6, \lambda, \lambda, \lambda, N, L, N$	
	Zde začíná kontrola palindromu na druhé pásce			
12.	$q_6, \alpha, \beta, \lambda$	$\rightarrow$	$q_6, \alpha, \beta, \beta, N, L, R$	
13.	$q_6, \alpha, \lambda, \lambda$	$\rightarrow$	$q_7, \alpha, \lambda, \lambda, N, N, L$	$\alpha \in \{\mathtt{a}, \mathtt{b}, \lambda\}$
14.	$q_7, \alpha, \lambda, \beta$	$\rightarrow$	$q_7, \alpha, \lambda, \beta, N, N, L$	$\alpha \in \{\mathtt{a}, \mathtt{b}, \lambda\}, \beta \in \{\mathtt{a}, \mathtt{b}\}$
15.	$q_7, \alpha, \lambda, \lambda$	$\rightarrow$	$q_8, \alpha, \lambda, \lambda, N, R, R$	$\alpha \in \{\mathtt{a}, \mathtt{b}, \lambda\}$
16.	$q_8, \alpha, \beta, \beta$	$\rightarrow$	$q_8, \alpha, \lambda, \lambda, N, R, R$	$\alpha \in \{\mathtt{a}, \mathtt{b}, \lambda\}, \beta \in \{\mathtt{a}, \mathtt{b}\}$
17.	$q_8, \alpha, \lambda, \lambda$	$\rightarrow$	$q_1, \alpha, \lambda, \lambda, N, N, N$	$\alpha \in \{\mathtt{a}, \mathtt{b}, \lambda\}$

Nedeterministická volba je skryta pouze v instrukcích 1 a 2. Stroj N čte prochází vstupem instrukcí 1 a v každém místě má možnost nedeterministicky se rozhodnout ukončit první slovo a začít test toho, zda jde o palindrom. Ve chvíli, kdy dojde N na konec vstupu, instrukce 3 ukončí první slovo a druhé podslovo bude pak prázdné. Bloky instrukcí 4 až 9 a 12 až 17 jsou zkopírovány z přechodové funkce Turingova stroje z příkladu 4.1.11, jsou upraveny jen v tom smyslu, že používají druhou a třetí pásku, jež po sobě nechávají prázdné. Instrukce 10 a 11 kopírují druhé slovo na druhou řádku.

Zde si ukážeme, že práci nedeterministického Turingova stroje M lze simulovat na deterministickém Turingovu stroji M'. Z tohoto hlediska nepřináší nedeterminismus nic pro teorii rozhodnutelnosti. Zajímavější je však situace v teorii složitosti, kde nedeter-

ministické Turingovy stroje hrají podstatnou roli.

Věta 4.1.16 *Ke každému nedeterministickému Turingovu stroji M existuje jednopáskový deterministický Turingův stroj M', který simuluje práci M a přijímá týž jazyk jako M.*

Důkaz: Jedním způsobem, jak převést nedeterministický Turingův stroj $M = (Q, \Sigma, \delta, q_0, F)$ na deterministický Turingův stroj $M' = (Q', \Sigma', \delta, q_0', F')$ by bylo implementovat v stroji M' průchod stromem výpočtu M do šířky. To by jistě šlo udělat, pokud bychom popisovali M' jako vícepáskový Turingův stroj, tak na jedné pásce by si udržoval frontu konfigurací, z níž by v cyklu vyzvedával konfiguraci, aplikoval na ni všechny možné přechody podle δ a výsledné konfigurace by M' přidával do fronty. Ve chvíli, kdy M' narazí na přijímající konfiguraci, přijme. Pokud zjistí M', že je fronta prázdná, odmítne.[4]

Popíšeme si podrobněji jiný postup, který se nám bude později hodit v části věnované složitosti. Označme pomocí r maximální faktor větvení stroje M, tedy

$$r = \max_{\substack{q \in Q \\ a \in \Sigma}} |\delta(q,a)| .$$

Předpokládejme dále, že pro každou dvojici $q \in Q, a \in \Sigma$ je množina možných přechodů $\delta(q,a)$ očíslovaná a pro $j \in \{1,\dots,r\}$ označme j-tý přechod pomocí $\delta(q,a,j)$ (pokud $|\delta(q,a)| < j \leq r$, definujeme $\delta(q,a,j) = \perp$, tedy pokud $\delta(q,a) = \varnothing$, pak definujeme $\delta(q,a,j) = \perp$ pro $j = 1,\dots,r$). To odpovídá tomu, že pokud máme k dispozici dodatečnou informaci o tom, který přechod z možných má M v danou chvíli použít, stane se výpočet M deterministickým.

Položme abecedu $B = \{b_1,\dots,b_r\}$, kde $b_1,\dots,b_r$ jsou nějaké symboly, jež se mohou, ale nemusí vyskytovat v abecedě Σ. Uvažme nyní výpočet $K_0^x,\dots,K_t$ stroje M nad vstupem x, tedy K_0^x je počáteční konfigurace a $K_i \in \delta(K_{i-1})$ pro $i = 1,\dots,t$. Tento výpočet lze jednoznačně popsat řetězcem $w \in B^t$. Znak $w[i] = b_j$ pro $i \in \{1,\dots,t\}$ a $j \in \{1,\dots,r\}$, pokud $K_i = \delta(K_{i-1}, j)$, kde $\delta(K_{i-1}, j)$ označuje konfiguraci vzniklou aplikací přechodu $\delta(q,a)$ na konfiguraci K_{i-1}, kde dvojice $q \in Q, a \in \Sigma$ tvoří displej v konfiguraci K_{i-1}. Na druhou stranu každý řetězec B^* určuje nějaký výpočet stroje M nad vstupem x.

Zkonstruujeme nejprve deterministický 3-páskový Turingův stroj M'', který bude simulovat práci M. Jednopáskový Turingův stroj M' pak můžeme z M'' zkonstruovat na základě věty 4.1.12. Význam pásek stroje M'' je následující:

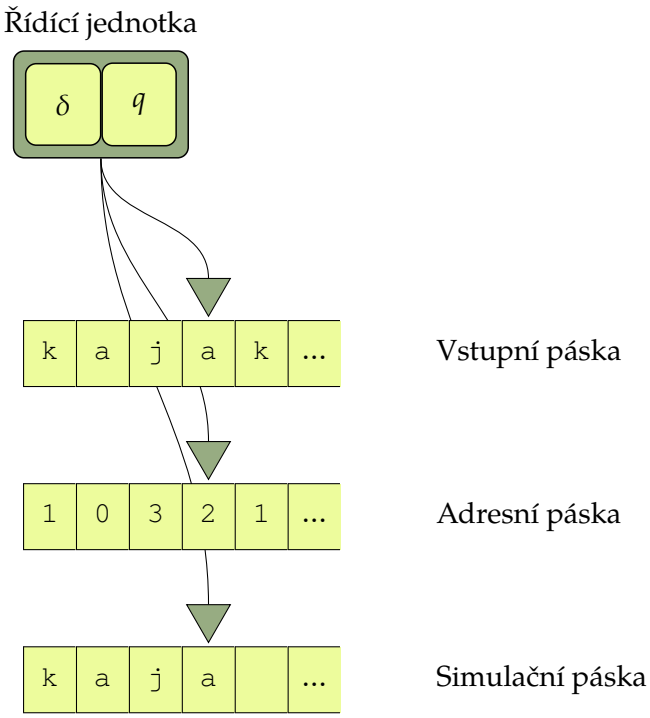
Vstupní páska uchovává vstup x, tato páska slouží jen ke čtení.

Adresní páska slouží ke generování řetězců $w \in B^*$, které specifikují adresy procházených uzlů stromu výpočtu M nad vstupem x.

Simulační páska slouží k simulaci stroje M na vstupu x při volbách přechodové funkce dle řetězce na adresní pásce.

Struktura stroje M'' je naznačena na obrázku 4.6.

[4] Nespecifikovali jsme přesně, co u nedeterministického Turingova stroje M znamená „odmítnutí“ vstupu. V případě nedeterministického Turingova stroje nás obvykle ani odmítnutí nezajímá a zajímáme se pouze o to, zda stroj přijme. Dá se ale říci, že M odmítne svůj vstup, pokud je jeho strom výpočtu konečný (tj. všechny větve výpočtu jsou konečné) a žádná z nich není přijímající.



Obrázek 4.6.: Struktura třípáskového deterministického Turingova stroje M'', který simuluje nedeterministický Turingův stroj M.

Stroj M'' postupně generuje řetězce $w \in B^*$ na adresní pásce. Při tom generuje nejprve všechny řetězce délky $1, 2, 3, \ldots$. S každým řetězcem $w \in B^*$ provádí M'' simulaci stroje M na vstupu x s volbami přechodové funkce podle řetězce w. K simulaci je využita simulační páska, která zde slouží přímo jako pracovní páska M, přičemž před každou simulací je vždy vyčištěna a je na ni okopírován vstup ze vstupní pásky. Pokud M'' vygeneruje při simulaci přijímající konfiguraci, pak přijme. Pokud pro nějakou délku $t \in \mathbb{N}$ na všech řetězcích $w \in B^t$ skončí simulace po méně než t krocích, pak M'' odmítne. □

Použitý způsob konstrukce stroj M'' v důkazu věty 4.1.16 provádí průchod stromu výpočtu stroje M nad vstupem x pomocí iterativního prohlubování a v principu se od průchodu do šířky příliš neliší.

4.2. Random Access Machine

V této kapitole zavedeme další výpočetní model, a to RAM — *random access machine*, tedy stroj s náhodným přístupem do paměti. Jde o model, který se často (ačkoli se to ne vždy zmiňuje) používá jako základní výpočetní model při měření časové a prostorové složitosti algoritmů. Motivací k zavedení tohoto modelu byla právě snaha přiblížit se více k reálným počítačům a umožnit tak studium algoritmů a jejich implementací pro klasické počítače. Při definici a popisu dalších vlastností budeme vycházet z článku [2], v němž byl model RAM zaveden. V literatuře je ovšem možno najít řadu jiných ekvivalentních definic, které se liší zejména v použitých instrukcích a způsobu zápisu programů.

4.2.1. Definice

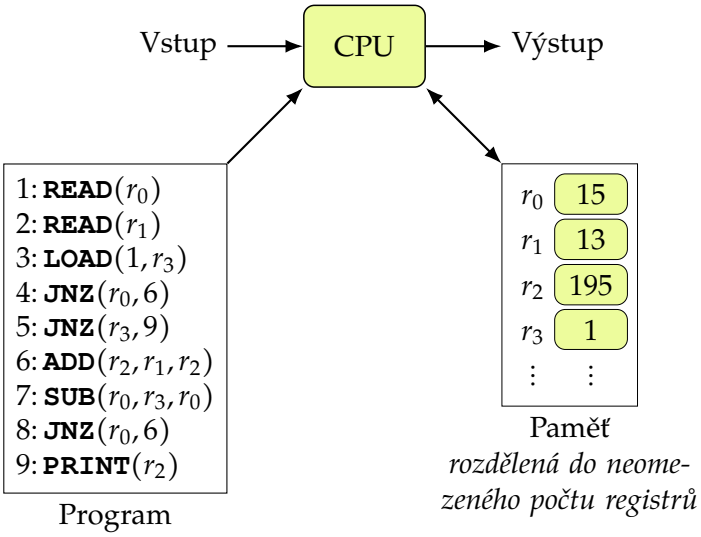
Podobně jako Turingovy stroje, i RAM je strojem s oddělenou pamětí pro data a pro instrukce. Struktura RAM je zobrazena na obrázku 4.7. RAM se skládá ze tří podstatných částí:

Program je konečnou posloupností instrukcí $I_0, I_1, I_2, \ldots I_\ell$, které jsou očíslované přirozenými čísly, očíslování určuje pořadí provádění těchto instrukcí a současně návěští pro příkaz podmíněného skoku. Jednotlivým prvkům posloupnosti instrukcí říkáme řádky programu.

Paměť pro data se skládá z neomezené posloupnosti registrů $r_0, r_1, r_2, \ldots$, které jsou očíslované přirozenými čísly počínaje nulou, těmto číslům budeme říkat *adresy*. Obsahem registru může být libovolně velké přirozené číslo[5].

Řídící jednotka (*central processing unit*, *CPU*) vykonává program. Instrukce jsou postupně vykonávány v pořadí, v jakém jsou zapsané v programu, i když některé

[5] Fakt, že se omezujeme na přirozená čísla, není nijak podstatný, klidně bychom mohli povolit libovolná celá čísla. Použití přirozených čísel má tu výhodu, že číslo uložené v registru je možné použít i jako adresu registru. V případě potřeby není velký problém reprezentovat i záporná čísla s pomocí přirozených čísel.



Obrázek 4.7.: Struktura RAM. Program na obrázku implementuje RAM, který počítá funkci násobení.

instrukce (např. instrukce skoku) mohou toto pořadí měnit. Řídící jednotka se při své práci rovněž stará o čtení hodnot ze vstupu a zápis hodnot na výstup.

Program pro RAM se skládá z instrukcí, které jsou popsány v tabulce 4.5. Při jejich popisu používáme následující konvence:

- Obsah registru r_i označujeme pomocí $[r_i]$.
- V popisu instrukcí může být adresa registru určena nepřímo, tedy obsahem jiného registru, pomocí $[\![r_i]\!]$ tak označujeme obsah registru s adresou, která je určena obsahem registru r_i, tj. je-li $j = [r_i]$ obsahem registru r_i, pak $[\![r_i]\!] = [r_{[r_i]}] = [r_j]$.
- Při popisu instrukcí dále používáme symbol $\leftarrow$ ve významu přiřazení, levá strana přitom určuje registr, do kterého má být přiřazena hodnota výrazu na pravé straně. Například výraz $r_i \leftarrow C$ popisuje efekt instrukce **LOAD**(C, r_i), která do registru r_i uloží hodnotu C.

Tabulka 4.5.: Seznam instrukcí RAM

Instrukce	Efekt	Popis
LOAD(C, r_i)	$r_i \leftarrow C$	Přiřadí do registru r_i konstantu $C \in \mathbb{N}$.

Tabulka 4.5.: Seznam instrukcí RAM

Instrukce	Efekt	Popis
ADD(r_i, r_j, r_k)	$r_k \leftarrow [r_i] + [r_j]$	Sečte obsahy registrů r_i a r_j a výsledek uloží do registru r_k. Adresy registrů r_i, r_j, r_k nemusí být nutně různé.
SUB(r_i, r_j, r_k)	$r_k \leftarrow \max([r_i] - [r_j], 0)$	Od obsahu registru r_i odečte obsah registru r_j a výsledek uloží do registru r_k. Adresy registrů r_i, r_j, r_k nemusí být nutně různé.
COPY$([r_p], r_d)$	$r_d \leftarrow [\![r_p]\!]$	Do registru r_d zkopíruje obsah registru s adresou určenou obsahem registru r_p.
COPY$(r_s, [r_d])$	$r_{[r_d]} \leftarrow [r_s]$	Do registru, jehož adresa je určena obsahem registru r_d, zkopíruje obsah registru r_s.
JNZ(r_i, I_z)	if $([r_i] > 0)$ goto z	Podmíněný skok (*jump if not zero*). Pokud je v registru r_i kladné číslo, pak program dále pokračuje instrukcí I_z, v opačném případě (tj. $[r_i] = 0$) program pokračuje instrukcí následující po právě vykonávané instrukci **JNZ**.
READ(r_i)	$r_i \leftarrow$ input	Do registru r_i přečte další číslo na vstupu. Pokud není na vstupu další číslo, načte hodnotu 0.
PRINT(r_i)	output $\leftarrow [r_i]$	Hodnotu uloženou v registru r_i zapíše na výstup.

Sada instrukcí popsaná v tabulce 4.5 zdaleka není jediná možná. Některé varianty neumožňují obecné sčítání a odčítání, ale pouze přičtení jedničky (*increment*) a odečtení jedničky (*decrement*). Další varianty připouštějí aritmetické operace nebo nepřímou adresaci pouze s použitím zvlášť pro to vyhrazeného registru, kterému se říká akumulátor. Na druhou stranu v některých případech se připouští obecná aritmetika, tedy ne jen sčítání a odčítání, ale například i násobení či celočíselné dělení. Prozatím se nezabýváme časovou a prostorovou složitostí, poznamenejme však již nyní, že vzhledem k tomu, že jednotlivé registry mohou obsahovat neomezeně velká čísla, je potřeba při počítání časové a prostorové složitosti algoritmu vzít do úvahy velikost reprezentace čísel uložených v použitých registrech. Tomu se budeme podrobněji věnovat v části III.

Příklad 4.2.1

Model RAM si ukážeme na jednoduchém příkladu stroje, který počítá funkci násobení. Program očekává na vstupu dvě čísla x_1 a x_2, na výstup zapíše jejich součin

MULT$(x_1, x_2) = x_1 \cdot x_2$. RAM R, který bude tuto funkci implementovat, bude postupovat podle velmi jednoduchého algoritmu: K hodnotě 0 postupně x_1-krát přičte hodnotu x_2.

```
1:  function MULT(x_1, x_2)
2:      READ(r_0)                                          ▷ r_0 ← x_1
3:      READ(r_1)                                          ▷ r_1 ← x_2
4:      LOAD(1, r_3)                          ▷ r_3 ← 1, budeme potřebovat odčítat 1.
5:      JNZ(r_0, 7)                                ▷ Pokud [r_0] > 0 skoč na řádek 7.
6:      JNZ(r_3, 10)                        ▷ [r_3] = 1, nepodmíněný skok na řádek 10.
7:      ADD(r_2, r_1, r_2)                  ▷ r_2 ← [r_2] + [r_1], přičtení r_1 k výsledku.
8:      SUB(r_0, r_3, r_0)     ▷ r_0 ← [r_0] − [r_3], protože [r_3] = 1, jde o odečtení jedničky.
9:      JNZ(r_0, 7)   ▷ Pokud je [r_0] > 0, skoč na řádek 7, tedy pokračuj v přičítání.
10:     PRINT(r_2)                                         ▷ Vypiš výsledek, který je v r_2.
11: end function
```

V programu jsou použity čtyři registry, registry r_0 a r_1 slouží k uložení vstupních hodnot, do registru r_2 se postupně x_1-krát přičte hodnota x_2, zde je tedy na závěr výsledek, který je vypsán instrukcí **PRINT**. Pomocný registr r_3 je použit jen k tomu, abychom postupně mohli odčítat jedničku od hodnoty v registru r_0 (instrukce odčítání **SUB** totiž neumožňuje přímo odečíst konstantu od registru, viz dále úmluvu 4.2.4).

RAM R vykonává program krok po kroku, tedy tak, jak jsme zvyklí z imperativních programovacích jazyků, přičemž význam jednotlivých kroků je uveden v komentáři u každého řádku programu. Krok 7 je proveden přesně x_1-krát (kde x_1 je první načtená hodnota uložená v r_0), pokud je $x_1 = 0$, pak v kroku 5 neprovede skok, ale RAM bude pokračovat krokem 6, což je nepodmíněný skok na konec programu. Tento nepodmíněný skok je implementován podmíněným skokem s použitím registru r_3, kde je uložena hodnota 1. Pokud je $x_1 > 0$ pokračuje se přičtením x_2 (v registru r_1) k 7. Poté je odečtena hodnota 1 v kroku 8. Pokud zůstane hodnota v r_0 pozitivní, dojde k opakování smyčky skokem na řádek 7. To znamená, že se krok 7 se opakuje dokud se postupným odčítáním jedničky nesníží hodnota x_1 na 0, tedy x_1-krát.

Povšimněme si dále instrukce **COPY**, která je popsaná ve dvou verzích. Tato instrukce umožňuje nepřímé adresování, bez kterého se neobejdeme, chceme-li, aby byl program schopen přistoupit k registrům s libovolně velkými adresami. Počet potřebných registrů může být pochopitelně závislý na velikosti vstupu a bez nepřímé adresace by počet registrů, k nimž může program přistoupit, byl pouze konstantní. Použití instrukce **COPY** si později ukážeme v příkladu 4.2.6.

Předpokládáme, že vstup je stroji RAM předán jako posloupnost čísel $x_1, x_2, \ldots, x_n$. Ke svému vstupu přistupuje RAM voláním instrukce **READ**, která načte hodnotu ze vstupu (tím se posune na vstup další hodnota). Pokud tato instrukce dojde ke konci vstupu, vrací již jen hodnoty 0. RAM buď ví, kolik parametrů má načíst (například počítá-li funkci

s pevným počtem parametrů), nebo očekává řetězec složený z čísel, který je ukončený 0, jsou možné samozřejmě i jiné možnosti, například první hodnotou předanou RAM může být počet parametrů, které následují. My se přidržíme prvních dvou přístupů, které budeme specifikovat za okamžik.

Výstup zapisuje RAM pomocí instrukce **PRINT**, opět se jedná o posloupnost čísel $y_1, y_2, \ldots, y_m$. Význam jednotlivých čísel na výstupu je pochopitelně dán programem, který RAM zpracovává.

Definice 4.2.2 (Výpočet RAM) Na začátku výpočtu dle programu $I_0, \ldots, I_\ell$ mají všechny registry hodnotu 0, výpočet začíná první instrukcí I_0 a pokračuje dalšími instrukcemi v pořadí. Pro zjednodušení dalších úvah budeme předpokládat, že číslo aktuální instrukce si řídící jednotka udržuje ve zvláštním registru (*čítač instrukcí*, *program counter*, `PC`). Hodnota tohoto registru je na začátku 0 a po provedení každé instrukce se zvyšuje o jedna. V případě použití instrukce **JNZ** může dojít ke skoku na jinou instrukci, než je další v pořadí, tím dojde i k příslušné změně hodnoty `PC`. Výpočet končí v případě, kdy je hodnota čítače instrukcí `PC` vyšší, než ℓ. K tomu může dojít ve dvou případech, buď je vykonána instrukce na poslední řádce programu I_ℓ (a její součástí není skok na řádek programu s nižším číslem), nebo dojde ke skoku (příkazem **JNZ**) na řádku programu $k > \ell$. Fakt, že se výpočet RAM R nad daným vstupem $x_1, \ldots, x_n$ zastaví, budeme označovat pomocí $R(x_1, \ldots, x_n)\downarrow$ a budeme říkat, že výpočet *konverguje*. To, že se výpočet nezastaví, tedy že *diverguje*, budeme označovat pomocí $R(x_1, \ldots, x_n)\uparrow$. ◂

Podobně jako v případě Turingových strojů, můžeme i RAM použít jednak k přijímání slov z daného jazyka, nebo k počítání hodnoty funkce. Tyto dva způsoby budeme rozlišovat zejména kvůli způsobu interpretace vstupu.

Definice 4.2.3 (RAM vyčíslitelné funkce) Nechť $f : \mathbb{N}^n \to \mathbb{N}$ je částečná funkce (nemusí tedy být definovaná pro všechny vstupy). Řekneme, že RAM R počítá funkci f, pokud platí:

- Pokud $f(x_1, \ldots, x_n)\uparrow$, pak výpočet R se vstupem $(x_1, \ldots, x_n)$ neskončí, nebo skončí s tím, že není vypsán žádný výstup pomocí **PRINT**.
- Pokud $f(x_1, \ldots, x_n)\downarrow$, pak výpočet R skončí, dojde k zápisu alespoň jedné hodnoty na výstup a první hodnotou, kterou R na výstup zapíše, je hodnota $f(x_1, \ldots, x_n)$

O funkci f, pro kterou existuje RAM, který ji počítá, budeme říkat, že je *RAM vyčíslitelná*. ◂

Všimněme si, že v definic 4.2.3 nijak neomezujeme počet provedených instrukcí **READ** ani **PRINT**. Z toho plyne, že s touto definicí každý RAM R počítá pro každé $n \in \mathbb{N}$ nějakou funkci n proměnných.

Všimněme si, že v případě Turingových strojů jsme v definici 4.1.8 zavedli turingovsky vyčíslitelnou funkci jako funkci řetězcovou, tedy typu $f : \Sigma^* \to \Sigma^*$. Naopak v případě RAM v definici 4.2.3 zavádíme RAM vyčíslitelnou funkci jako funkci nad přirozenými čísly, tedy funkci typu $f : \mathbb{N}^n \to \mathbb{N}$. Pro Turingovy stroje je přirozené pracovat s řetězci,

zatímco RAM přirozeně pracuje s čísly. Na druhou stranu je potřeba si uvědomit, že tento rozdíl není příliš podstatný: Posloupnost čísel na vstupu RAM může reprezentovat i řetězec, jak si ukážeme dále v definici 4.2.5 a podobně i posloupnost čísel na výstupu můžeme interpretovat jako řetězec. Na druhou stranu Turingovu stroji můžeme předat na vstup posloupnost čísel zakódovaných v jednom řetězci a podobně i řetězec na výstupu může reprezentovat číslo. Tomu se budeme dále věnovat v kapitole 5.4.

Úmluva 4.2.4 (Zjednodušení použití některých instrukcí) *Na příkladu 4.2.1 si můžeme všimnout, že instrukční sada, kterou jsme zvolili, je v lecčems velmi omezující, například nemůžeme rovnou napsat* **SUB**$(r_0, 1, r_0)$ *pro snížení hodnoty v registru* r_0 *o* 1*. Místo toho je potřeba využít pomocného registru* r_3*, kam si nejprve uložíme konstantu* 1*, abychom mohli zavolat* **SUB**(r_0, r_3, r_0) *a tím teprve odečíst od obsahu registru* r_0 *hodnotu* 1*. Toto omezení má svůj smysl, který se ukáže teprve ve chvíli, kdy začneme jednotlivým instrukcím přiřazovat čas provedení a tím počítat časovou složitost algoritmu. Zatímco instrukce* **LOAD***, tedy načtení konstanty do registru bude mít konstantní složitost, složitost operace* **SUB** *bude závislá na hodnotách menšence a menšitele, tedy na obsahu odpovídajících registrů. Podobně omezující je fakt, že nemáme k dispozici nepodmíněný skok, opět to lze obejít s pomocným registrem a uložením nenulové hodnoty do tohoto registru. Pro přehlednost programu v RAM by se nám navíc hodilo použití symbolických návěští.*

Na základě těchto úvah si proto zavedeme následující konvenci:

(*I*) *Registr* r_0 *vyhradíme v zápisech programů pro účely následujících zjednodušení instrukcí a nebudeme jej výslovně používat.*

(*II*) *V operacích* **ADD** *a* **SUB** *připustíme použití konstanty jako jednoho ze sčítanců, menšitele či menšence (číselnou konstantu od registru vždy odlišíme, neboť registry označujeme pomocí* r_i*, navíc není potřeba povolovat více konstant než jednu). Například* **ADD**$(r_i, 5, r_k)$ *je zkratkou za posloupnost* **LOAD**$(5, r_0)$, **ADD**(r_i, r_0, r_k).

(*III*) *Zavedeme si dodatečnou instrukci* **JUMP**(I_z)*, která provede nepodmíněný skok na instrukci* I_z*. Jde o zkratku za posloupnost* **LOAD**$(1, r_0)$, **JNZ**(r_0, I_z).

(*IV*) *V operaci* **PRINT** *povolíme zápis konstanty. Například* **PRINT**(5) *je zkratkou za posloupnost* **LOAD**$(5, r_0)$, **PRINT**(r_0).

(*V*) *V zápisech programů pro RAM také umožníme používat symbolických návěští, jež budeme deklarovat pomocí* `návěští:`*. Použito v instrukci skoku pak návěští odkazuje na následující řádek.*

(*VI*) *V zápisech programů pro RAM také umožníme používat symbolické názvy registrů. V kterémkoli místě programu může být přítomna deklarace proměnné s tím, ve kterém registru bude uložena. Dále je možno používat název této proměnné místo registru. To nám umožní dát registrům význam, který bude z jejich názvu snáze pochopitelný.*

Můžeme se samozřejmě ptát, proč již v tabulce 4.5 nemáme například k dispozici příkaz nepodmíněného skoku. Důvodem je snaha o minimalismus, snažíme se zvolit množinu instrukcí tak, aby byla co nejjednodušší a nebyly v ní nadbytečné instrukce.

Popíšeme si ještě, co znamená, že RAM R přijímá jazyk L.

Definice 4.2.5 Budeme uvažovat řetězce nad konečnou abecedou $\Sigma = \{\sigma_1, \ldots, \sigma_p\}$. Mohli bychom sice uvažovat i nekonečnou abecedu, protože neomezujeme čísla na vstupu, nebudeme to však potřebovat. Podstatné je, že symboly abecedy indexujeme od 1. Pokud je $w = \sigma_{i_1}\sigma_{i_2}\sigma_{i_3}\ldots\sigma_{i_n} \in \Sigma^*$ řetězec (který může být i prázdný), pak jej RAMu R předáme na vstup jako posloupnost čísel $i_1, \ldots, i_n, 0$. Protože indexy symbolů jsou nenulové, první nula přečtená instrukcí **`READ`** ze vstupu označuje konec řetězce. Výpočet RAM s takto předaným vstupem budeme označovat jako $R(w)$.

- Řekneme, že RAM R přijímá slovo $w \in \Sigma^*$, pokud $R(w)\downarrow$, R použije během výpočtu alespoň jednu instrukci **`PRINT`** a první hodnotou zapsanou na výstup je 1.
- Řekneme, že RAM R odmítá slovo $w \in \Sigma^*$, pokud $R(w)\downarrow$ a během výpočtu buď nedojde k zápisu na výstup instrukcí **`PRINT`**, nebo první hodnotou zapsanou na výstup není 1.

Podobně jako v případě Turingových strojů řekneme, že RAM R přijímá (resp. odmítá) jazyk $L \subseteq \Sigma^*$, pokud R přijímá (resp. odmítá) právě slova z jazyka L. Jazyk přijímaný RAMem R označíme pomocí $L(R)$. Řekneme, že RAM R rozhoduje jazyk L, pokud přijímá L a odmítá $\overline{L}$ (tj. na všech vstupech se zastaví a buď je přijme nebo odmítne). ◂

Příklad 4.2.6 ukazuje RAM rozhodující jazyk palindromů. Pro srovnání si můžeme připomenout, že jsme již zkonstruovali jednopáskový deterministický Turingův stroj v příkladu 4.1.4 a dvoupáskový deterministický Turingův stroj v příkladu 4.1.11. Navíc jsme zkonstruovali dvoupáskový nedeterministický Turingův stroj přijímající jazyk PAL^2.

Příklad 4.2.6

Ukážeme si například RAM R, který rozhoduje jazyk

$$\text{PAL} = \left\{w = w^R \mid w \in \{\mathtt{a},\mathtt{b}\}^*\right\}.$$

Pro účely předání řetězce $w \in \{\mathtt{a},\mathtt{b}\}^*$ RAMu, musíme očíslovat znaky symboly abecedy. Označíme proto $\sigma_1 = \mathtt{a}, \sigma_2 = \mathtt{b}$ a budeme pracovat nad abecedou $\Sigma = \{\sigma_1, \sigma_2\} = \{\mathtt{a},\mathtt{b}\}$. Číslo 1 předané na vstupu tedy odpovídá znaku `a` a číslo 2 předané na vstupu odpovídá znaku `b`. Program nejprve načte celý vstupní řetězec w délky n do registrů $r_8, \ldots, r_{n+7}$, během načítání vstupu si navíc program spočítá hodnotu $m = \lfloor \frac{n}{2} \rfloor$. Poté pro každý index $i = 0, \ldots, m-1$ program zkontroluje, zda $[r_{4+i}] = [r_{n+3-i}]$, tedy zda $x[i+1] = x[n-i]$. Popis jednotlivých kroků programu je uveden v komentářích.

1: **function** PALINDROM(w)

Alokace proměnných

2: Proměnná n označuje registr r_1, je v ní uložena délka vstupu.

3: Proměnná m označuje registr r_2, průběžně je v ní počítána hodnota $\lfloor \frac{n}{2} \rfloor$.
4: Proměnná a označuje registr r_3, slouží k načtení dalšího znaku ze vstupu. Dále je tato proměnná použita jako pomocná při porovnávání $x[i+1]$ s $x[n-i]$.
5: Proměnná *addr* označuje registr r_4, slouží k počítání adresy pro nepřímou adresaci.
6: Proměnná *odd* označuje registr r_5, slouží k identifikaci toho, zda je aktuální hodnota n lichá.
7: Proměnná b označuje registr r_6 a proměnná c označuje registr r_7. Tyto dvě proměnné jsou použity pro kontrolu rovnosti dvou znaků.

Načtení vstupu

`načti_znak:`
8: **READ**(a) ▷ $a \leftarrow$ znak ze vstupu.
9: **JNZ**$(a,$ `další_znak`$)$ ▷ Pokud nebyla načtena 0, jde o další znak.
10: **JUMP**(`kontrola`) ▷ Pokud byla načtena 0, jde o konec vstupu, přejdi na kontrolu toho, je-li načtený řetězec palindrom.

`další_znak:`
11: **LOAD**$(8, addr)$ ▷ $addr \leftarrow 8$
12: **ADD**$(addr, n, addr)$ ▷ $addr \leftarrow n + addr = n + 8$
13: **COPY**$(a, [addr])$ ▷ Načtený znak je uložen do registru $r_{addr} = r_{n+8}$.
14: **ADD**$(n, 1, n)$ ▷ $n \leftarrow n + 1$
15: **JNZ**$(odd,$ `lichá`$)$ ▷ Pokud je $odd > 0$, načtený znak je na liché pozici.

`sudá:`
16: **LOAD**$(1, odd)$ ▷ Byl načten znak na sudé pozici, příští bude na liché.
17: **ADD**$(m, 1, m)$ ▷ Zvyš hodnotu $m = \lfloor \frac{n}{2} \rfloor$ o 1.
18: **JUMP**(`načti_znak`) ▷ Vstup ještě neskončil, načti další znak.

`lichá:`
19: **LOAD**$(0, odd)$ ▷ Byl načten znak na liché pozici, příští bude na sudé.
20: **JUMP**(`načti_znak`) ▷ Vstup ještě neskončil, načti další znak.

Test rovnosti $x[i+1] = x[n-i]$ postupně pro $i = m-1, \ldots, 0$.

`kontrola:`
21: **JNZ**$(m,$ `cyklus`$)$ ▷ Pokud je počet znaků $n > 1$, pokračuj kontrolním cyklem.
22: **JUMP**(`přijmi`) ▷ Pokud je $m = 0$, je $n \leq 1$ a řetězec je automaticky přijat jako palindrom.

`cyklus:`
23: **SUB**$(m, 1, m)$ ▷ $m \leftarrow m - 1$

Načtení $x[m+1]$ do b

24: **LOAD**$(8, addr)$ ▷ $addr \leftarrow 8$
25: **ADD**$(addr, m, addr)$ ▷ $addr \leftarrow m + addr = m + 8$

```
26:     COPY([addr], b)                          ▷ b ← [[addr]] = [r_{m+8}] = x[m+1]
        Načtení x[n − m] do c
27:     LOAD(7, addr)                            ▷ addr ← 7
28:     ADD(addr, n, addr)                       ▷ addr ← addr + n = n + 7
29:     SUB(addr, m, addr)                       ▷ addr ← addr − m = n + 7 − m
30:     COPY([addr], c)                          ▷ c ← [[addr]] = [r_{n+7−m}] = x[n − m]
        Test x[m + 1] ≤ x[n − m]
31:     SUB(b, c, a)                             ▷ a ← max(b − c, 0)
32:     JNZ(a, odmítni)          ▷ a > 0 ⇔ b − c > 0 ⇔ b > c, tedy b ≠ c a vstup bude
        odmítnut.
        Test x[m + 1] ≥ x[n − m]
33:     SUB(c, b, a)                             ▷ a ← max(c − b, 0)
34:     JNZ(a, odmítni)          ▷ a > 0 ⇔ c − b > 0 ⇔ c > b, tedy b ≠ c a vstup bude
        odmítnut.
        Do další smyčky
35:     JNZ(m, cyklus)  ▷ Platí b = c a tedy x[m + 1] = x[n − m], navíc ještě zbývají
        dvojice k porovnání, pokračuj v cyklu.
        Přijetí
        přijmi:
36:     PRINT(1)        ▷ Všechna porovnání uspěla, přijmi zapsáním 1 na výstup.
37:     JUMP(konec)                  ▷ Přeskočíme zapsání 0 na výstup (není nutné).
        odmítni:
38:     PRINT(0)                                 ▷ Odmítni tím, že zapíšeš 0 na výstup.
        konec:
39: end function
```

4.2.2. Programování RAM

Abychom si dále zjednodušili situaci, uvědomme si, že programovací jazyk RAM odpovídá jednoduchému procedurálnímu jazyku podobného například Pascalu. Hlavní rozdíly proti běžnému jazyku jsou následující:

1. K dispozici jsou jen přirozená čísla.
2. Jediné přístupné aritmetické operace jsou sčítání a odčítání.
3. Procedury a funkce nelze volat rekurzivně.
4. Pole jsou jednorozměrná a s neomezenou velikostí.

Funkce jsou v programu RAM použity tím, že jejich kód je přímo vepsán do místa použití (případně s přečíslováním registrů). Funkce a procedury jsou tedy vždy *inline*. Uvažme například funkci Mult, jejíž program je uveden v příkladu 4.2.1. Pokud v jiném programu pro RAM použijeme funkci násobení tím, že napíšeme jako řádek volání

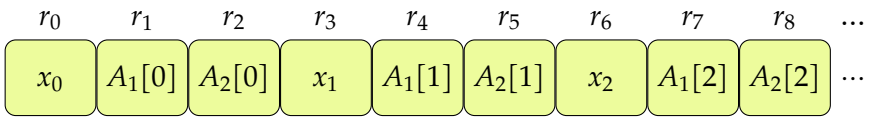
$$r_1 \leftarrow \text{Mult}(r_2, r_3),$$

je tento řádek jen zkratkou za vložení celého kódu programu pro funkci Mult, kde jsou vstupy vzaty z registrů r_2 a r_3 místo načtení ze vstupu a kde je výsledná hodnota uložena do registru r_1 místo zapsání na výstup. Absence rekurze není příliš omezující, neboť ji vždy můžeme nahradit pomocí cyklu **`while`**, který je možné v RAM zapsat pomocí podmíněného skoku.

Zastavme se chvíli u proměnných a polí. Paměť RAM je tvořena jedním neomezeným polem. Předpokládejme, že program stroje RAM R používá skalární proměnné (tedy jednoduché proměnné pro čísla) $x_0, \ldots, x_s$ a pole $A_1, \ldots, A_p$, která jsou indexována přirozenými čísly, tedy počínaje nulou, a jejichž délka není omezena. Potom všechna tato data reprezentujeme v jednorozměrné paměti RAM, následujícím způsobem:

- Prvek $A_i[j]$, kde $i \in \{1, \ldots, p\}$, $j \in \mathbb{N}$, umístíme do registru $r_{i+j*(p+1)}$.
- Prvky pole A_i, $i = 1, \ldots, p$ jsou tedy v registrech $r_i, r_{i+p+1}, r_{i+2(p+1)}, \ldots$.
- Proměnnou x_i, kde $i \in \{0, \ldots, s\}$ umístíme do registru $r_{i*(p+1)}$.
- Skalární proměnné jsou tedy postupně v registrech $r_0, r_{p+1}, r_{2(p+1)}, r_{3(p+1)}, \ldots, r_{s(p+1)}$.

Připomeňme si, že registr r_0 má zvláštní význam pro zjednodušení programu RAM dané úmluvou 4.2.4. Tento zvláštní význam má tedy při použití zjednodušeného zápisu instrukcí i proměnná x_0. Pole a skalární proměnné jsou tedy vzájemně propleteny jako zip. V paměti je nejprve uložena první skalární proměnná, poté první prvky polí, druhá skalární proměnná, druhé prvky polí, třetí skalární proměnná, třetí prvky polí atd. Tuto situaci ilustruje pro případ dvou polí obrázek 4.8. Uvědomme si, že není možné prostě zapsat do paměti pole za sebe, neboť jejich délka není omezena.



Obrázek 4.8.: Způsob alokace proměnných a polí v programu RAM, kde jsou použity skalární proměnné $x_0, \ldots, x_s$ a dvě pole A_1 a A_2.

4.2.3. Varianty RAM

V této podkapitole zmíníme dvě varianty modelu RAM, které je možné najít v literatuře — RASP a PRAM.

RASP

Stroj RAM ani Turingovy stroje nejsou stroji Von Neumannovy architektury, neboť mají oddělený paměťový prostor pro program a data. Tím se i RAM odlišuje od běžného počítače, byť se mu jinak snaží blížit[6]. Stroje RASP odstraňují tento nedostatek. Zkratka RASP znamená *random access stored program*, název tedy naznačuje, že program je uložen v hlavní paměti spolu s dalšími daty. Program má navíc možnost zasahovat i do paměti se svým kódem, protože ten je součástí přístupného paměťového prostoru.

Popíšeme si stručně verzi RASP, která byla popsána v článku [2]. Model RASP, který v tomto článku autoři popsali, má k dispozici dva zvláštní registry — *akumulátor* (*accumulator*, AC) a *čítač instrukcí* (*instruction counter*, IC). Na počátku jsou hodnoty v obou těchto registrech nulové. Kromě toho má RASP, stejně jako stroj RAM, neomezený počet registrů $r_0, r_1, r_2, \ldots$. Každá instrukce RASP je uložena v paměti ve dvou po sobě následujících registrech, kde v prvním registru je uveden *operační kód* instrukce a ve druhém registru je uvedena hodnota parametru instrukce, každá instrukce má tedy právě jeden parametr. Na začátku je program zapsán v po sobě jdoucích registrech, přičemž první instrukce zabírá první dva registry r_0, r_1. Program je ukončen instrukcí **`HALT`**. Veškerá aritmetika a další operace probíhají s pomocí akumulátoru. Seznam instrukcí RASP i s jejich operačními kódy je uveden v tabulce 4.6.

Tabulka 4.6.: Seznam instrukcí RASP

Instrukce	**Op. kód**	**Efekt**	**Popis**
`LOAD`(j)	1	AC $\leftarrow j$ IC $\leftarrow$ [IC] + 2	Přiřadí do akumulátoru konstantu $j \in \mathbb{N}$.
`ADD`(j)	2	AC $\leftarrow$ [AC] + $[r_j]$ IC $\leftarrow$ [IC] + 2	Přičte k akumulátoru hodnotu z registru r_j.
`SUB`(j)	3	AC $\leftarrow \max([\text{AC}] - [r_j], 0)$ IC $\leftarrow$ [IC] + 2	Od akumulátoru odečte obsah registru r_j.
`STORE`(j)	4	$r_j \leftarrow$ [AC]	Do registru r_j vloží obsah akumulátoru.
`JNZ`(j)	5	if $([AC] > 0)$ then IC $\leftarrow j$ else IC $\leftarrow$ [IC] + 2	Pokud je hodnota v akumulátoru kladná, pokračuje se instrukcí v registru r_j, jinak se pokračuje následující instrukcí.

[6] Z praktického hlediska není ovšem tento rozdíl příliš podstatný, neboť běžně nepíšeme programy, jež by potřebovaly upravovat svůj kód.

Tabulka 4.6.: Seznam instrukcí RASP

Instrukce	Op. kód	Efekt	Popis
READ(j)	6	$r_j \leftarrow$ input	Do registru r_j přečte další číslo na vstupu. Pokud není na vstupu další číslo, načte hodnotu 0.
PRINT(j)	7	output $\leftarrow [r_j]$	Hodnotu uloženou v registru r_j zapíše na výstup.
HALT	0, 8 až ∞	stop	Ukončení programu, operačním kódem je libovolné číslo, které není kódem jiné instrukce (tedy 0 nebo alespoň 8.)

Všimněme si, že instrukční sada v tabulce 4.6 neobsahuje instrukce **COPY** pro nepřímou adresaci. Nepřímou adresaci je možné provést úpravou programu za běhu tak, že program upraví hodnotu parametru příslušné instrukce (například **STORE** nebo **ADD**). Jinak jsou ovšem programy pro RASP podobné těm pro RAM a je možno celkem bez obtíží program pro RASP upravit na program pro RAM a naopak.

PRAM

Paralelní RAM (PRAM) je obvyklým modelem použitým při studiu paralelních algoritmů. Od běžného stroje RAM se liší tím, že má neomezený počet procesorů, které vykonávají týž program. Paměť je tvořena jednou neomezenou posloupností registrů a je sdílená pro všechny procesory. Při přístupu do paměti je tedy potřeba nějakým způsobem řešit kolize. Zde si vystačíme s tímto jednoduchým popisem, neboť paralelnímu programování se tento text nevěnuje.

4.3. Ekvivalence Turingových strojů a RAM

V této kapitole si ukážeme, že Turingovy stroje a RAM mají shodnou výpočetní sílu. To znamená, že ke každému Turingovu stroji M lze sestrojit RAM R, který „dělá totéž“, a naopak. Každý směr převodu popíšeme zvlášť. Zejména nás přitom zajímají jazyky přijímané Turingovým strojem a RAMem a funkce vyčíslitelné Turingovým strojem a RAMem.

Věta 4.3.1 (Ekvivalence Turingových strojů a RAM) *Nechť $L \subseteq \Sigma^*$ je jazyk a $f : \Sigma^* \to \Sigma^*$ je řetězcová funkce. Potom platí:*

Viz definice 4.1.3 a 4.2.5.

1. *Jazyk L je přijímaný nějakým Turingovým strojem M, právě když je přijímaný nějakým RAM R.*

Viz definice 4.1.8 a 4.2.3.

2. *Funkce f je turingovsky vyčíslitelná, právě když je RAM vyčíslitelná.*

Důkaz obou ekvivalencí provedeme ve dvou krocích, nejprve si ukážeme, jak převést Turingův stroj na RAM, který přijímá týž jazyk a počítá touž funkci. Poté si ukážeme druhý směr, tedy převod RAM na Turingův stroj, který přijímá týž jazyk a počítá touž funkci.

4.3.1. Převod Turingova stroje na RAM

Uvažme Turingův stroj $M = (Q, \Sigma, \delta, q_0, F)$. Ukážeme, jak zkonstruovat RAM R, který přijímá jazyk $L(M)$ a počítá funkci f_M.

Pro jednoduchost budeme předpokládat, že M má pásku neomezenou jen doprava, z věty 4.1.9 víme, že tento předpoklad není nijak omezující. Budeme dále předpokládat, že stavy i symboly páskové abecedy jsou očíslované, přesněji,

- $Q = \{q_0, q_1, \ldots, q_r\}$ pro nějaké $r \geq 0$, kde q_0 je počáteční stav a
- $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_s\}$ pro nějaké $s \geq 1$, kde $\sigma_0 = \lambda$ je znak prázdného políčka.

Připomeňme si, že hodnota 0 na vstupu RAMu označuje konec vstupního řetězce, což je konzistentní s tím, že σ_0 označuje znak prázdného políčka. Předpokládáme dále, že buňky na pásce M jsou očíslované přirozenými čísly, přičemž nejlevější políčko má index 0.

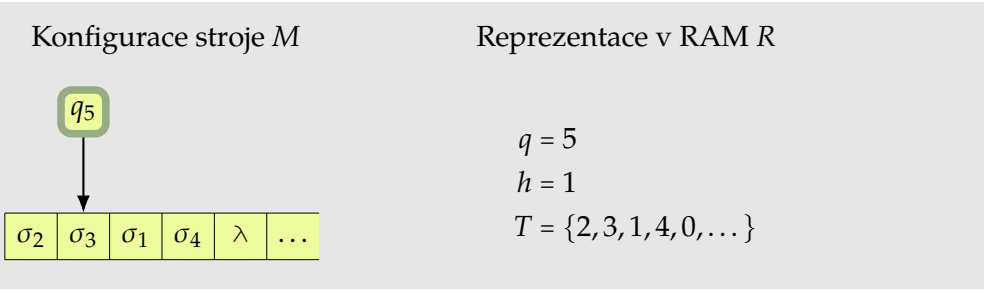
Viz definice 4.1.2

Konstruovaný RAM R bude v paměti reprezentovat aktuální konfiguraci, a tedy i displej stroje M. K tomu R použije následující proměnné a datové struktury (připomeňme si, že způsob uložení polí a proměnných v paměti RAM jsme popsali v kapitole 4.2.2).

- Proměnná q označuje index aktuálního stavu stroje M, tj. hodnota $q = i$ reprezentuje fakt, že M je ve stavu q_i.
- Proměnná h označuje index buňky, nad kterou je hlava M.
- Pole T reprezentuje obsah pásky M. Symbol pod hlavou M je tedy na pozici $T[h]$.

Příklad 4.3.2

Níže je ukázán příklad konfigurace a její reprezentace v RAM M.



Práci RAM R můžeme rozdělit do tří fází. V první fázi načte vstup do pole T, ve druhé fázi provádí samotnou simulaci stroje M a ve třetí fázi zpracuje výsledek této simulace.

Načtení vstupu: Zatímco Turingův stroj M očekává, že má vstup zapsán na pásce, RAM R oproti tomu čte vstup pomocí instrukcí **`READ`**. RAM R tedy nejprve načte celý vstupní řetězec a uloží jej do pole T na pozice $T[0], \dots, T[n-1]$, kde n označuje počet načtených nenulových indexů znaků před hodnotou 0, která označuje konec vstupního řetězce. Hodnoty proměnných h a q jsou na počátku 0, což je dáno počátečním obsahem registrů R.

Provedení kroku: Provedení jednoho kroku je provedeno pomocí sady podmíněných příkazů, z nichž každý implementuje jeden řádek tabulky přechodové funkce δ Turingova stroje M. Instrukce $\delta(q_i, \sigma_j) = (q_k, \sigma_l, R)$ je v programu R implementována podmíněným příkazem

```
1: if q = i and T[h] = j then
2:     q ← k
3:     T[h] ← l
4:     h ← h + 1
5: end if
```

V případě pohybu N zůstává hodnota h beze změny a v případě pohybu L je na řádku 4 použit rozdíl $h \leftarrow h - 1$, přičemž není nutné před odečtením jedničky kontrolovat, zda $h = 0$, protože předpokládáme, že M nepohne hlavou nalevo od nejlevější buňky pásky.

Pokud některý z těchto podmíněných příkazů uspěje, pokračuje R simulací dalšího kroku. V opačném případě cyklus vykonávání kroků končí a pokračuje se další fází, tedy zapsáním výstupu.

Příklad 4.3.3

Uvažme Turingův stroj M s následující přechodovou funkcí.

	q, c	$\rightarrow$	q', c', Z
1.	q_0, σ_2	$\rightarrow$	q_3, σ_1, R
2.	q_3, σ_1	$\rightarrow$	q_2, σ_0, L
3.	q_2, σ_0	$\rightarrow$	q_0, σ_2, N

RAM R tuto přechodovou funkci implementuje následující posloupností podmíněných příkazů.

```
1: if q = 0 and T[h] = 2 then
2:     q ← 3
3:     T[h] ← 1
4:     h ← h + 1
5: else if q = 3 and T[h] = 1 then
6:     q ← 2
7:     T[h] ← 0
8:     h ← h − 1
9: else if q = 2 and T[h] = 0 then
10:    q ← 0
11:    T[h] ← 2
12: else
13:     Konec simulace
14: end if
```

Zpracování výsledku simulace: Pokud nás zajímá, zda Turingův stroj M přijal svůj vstup, tedy zda vstupní slovo patří do jazyka $L(M)$, pak je na výstup zapsána hodnota 1 za podmínky, že stav uložený v proměnné q je přijímající. Pokud nás zajímá hodnota funkce f_M vyčíslované Turingovým strojem M, je na výstup zapsáno slovo na pásce aktuálně uložené v poli T. Buď R zapíše na výstup obsah T až do první prázdné buňky, nebo zapíše ten kus pole T, k němuž se M během výpočtu dostal, to je dáno maximální hodnotou h během výpočtu.

4.3.2. Převod RAM na Turingův stroj

Popišme si nyní, jak převést RAM R na Turingův stroj M, který počítá touž funkci a přijímá týž jazyk. Turingův stroj M zkonstruujeme jako 4-páskový.

Význam pásek je následující:

1. **Vstupní páska.** Posloupnost čísel, která má dostat R na vstup. Jsou zakódovaná binárně a oddělená znakem #. Z této pásky M jen čte.

2. **Výstupní páska.** Sem zapisuje M čísla, která R zapisuje na výstup. Jsou zakódovaná binárně a oddělená znakem #. Na tuto pásku M jen zapisuje.

3. **Paměť RAM.** Obsah paměti stroje R. Formát této pásky popíšeme níže.

4. **Pomocná páska.** Pro výpočty součtu, rozdílu, nepřímých adres, posunu části paměťové pásky a podobně.

Předpokládáme, že vstupem R mohou být libovolná přirozená čísla, proto předpokládáme, že Turingův stroj M dostane na vstup tato čísla zapsaná binárně. To je ovšem technická záležitost, bylo by samozřejmě možné Turingovu stroji rovnou dát seznam znaků, které jsou těmito čísly reprezentovány, pokud nás zajímá náležení řetězce do jazyka. Podobně přistupujeme i k výstupu Turingova stroje M, který má být opět v podobě binárních zápisů čísel, jež by vypsal RAM R.

Je třeba si podrobněji popsat reprezentaci obsahu paměti RAM na 3. pásce. Stačí si pamatovat obsahy využitých registrů, tedy těch, do nichž byla uložena nějaká hodnota v průběhu práce R. Ostatní registry obsahují 0. Obsahy registrů si zapíšeme za sebe v rostoucím pořadí podle čísel registrů[7] Za každý registr r_i přidáme dvojici tvořenou indexem i a číslem $[r_i]$ uloženým v registru r_i. Obě čísla jsou zapsaná binárně a oddělená znakem |. Jednotlivé dvojice jsou pak odděleny znakem #. To znamená, že jsou-li aktuálně využité registry $r_{i_1}, r_{i_2}, \ldots, r_{i_m}$, kde $i_1 < i_2 < \cdots < i_m$, pak je na pásce reprezentující paměť RAM R řetězec

$$(i_1)_B \mid ([r_{i_1}])_B \# (i_2)_B \mid ([r_{i_2}])_B \# \ldots \# (i_m)_B \mid ([r_{i_m}])_B.$$

Předpokládejme, že RAM R se řídí programem složeným z instrukcí $I_1, \ldots, I_\ell$, kde $\ell \in \mathbb{N}$. Přechodová funkce M zabezpečí vykonání těchto instrukcí v daném pořadí. Za tím účelem bude součástí stavu M hodnota čítače instrukcí (*program counter*, `PC`), což je číslo z rozmezí $1, \ldots, \ell$. Uvědomme si, že ℓ je konstanta nezávislá na vstupu a je tedy možné, aby hodnota `PC` byla uložená ve stavu. Každá instrukce bude nahrazena posloupností instrukcí Turingova stroje. Přičemž posloupnost instrukcí implementující instrukci I_j pro $j \in \{1, \ldots, \ell\}$ bude končit ve stavu, který bude počátečním pro posloupnost implementující následující instrukci v programu R. Obvykle po I_j následuje I_{j+1} kromě případu instrukce **`JNZ`**, která může změnit hodnotu `PC` na hodnotu danou parametrem. Posloupnost s číslem $\ell + 1$, tedy za koncem programu, pak bude provádět zakončení práce programu.

Implementace každé jednotlivé instrukce RAM na Turingovu stroji je při dané reprezentaci paměti RAM celkem přímočará a daná přímo významem instrukcí v tabulce 4.5. Při implementaci těchto instrukcí je podstatné zejména to, že Turingův stroj může manipulovat se seznamem dvojic reprezentujících data v registrech na 3. pásce. To znamená, že pro daný index i může najít blok odpovídající registru r_i, přečíst a upravit hodnotu v něm uloženou. Navíc Turingův stroj dokáže poznat, zda daný index i nemá dosud svůj záznam v paměti a je tedy potřeba jej přidat. Aritmetické operace nad binárními čísly lze opět snadno implementovat na Turingovu stroji (zejména je-li k dispozici pomocná 4. páska).

[7] Není sice nutné, aby čísla registrů byla uspořádaná, ale trochu to zjednodušuje hledání registru s daným indexem v seznamu.

4.4. Částečně rekurzivní funkce *

Dalším výpočetním modelem, který si popíšeme, jsou částečně rekurzivní funkce. Ty budou také reprezentovat další paradigma programovacích jazyků, totiž funkcionální přístup k programování.

4.4.1. Definice

Třídy primitivně a částečně rekurzivních funkcí budeme odvozovat ze základních funkcí pomocí odvozovacích pravidel neboli operátorů, obojí hned zavedeme. Všechny funkce uvažované v této části jsou funkce typu

$$f : \mathbb{N}^n \to \mathbb{N}$$

pro $n \geq 1$.

Primitivně rekurzivní funkce

Začneme popisem primitivně rekurzivních funkcí.

Definice 4.4.1 (Primitivně rekurzivní funkce) *Základní funkce*, z nichž bude začínat odvozování každé funkce mají následující tři formy.

(I) *Konstantní nulová funkce* $o(x) = 0$. Jde tedy o funkci jedné proměnné, která pro každý vstup nabývá hodnoty 0.

(II) *Funkce následníka* $s(x) = x+1$. Jde tedy opět o funkci jedné proměnné, která nabývá hodnoty o jedna vyšší, než je číslo na vstupu.

(III) *Projekce* $I_n^j(x_1, \ldots, x_n) = x_j$, kde $1 \leq j \leq n$ jsou libovolná přirozená čísla. Jde o funkci n proměnných, která vrací hodnotu j-tého parametru.

Ze základních funkcí budeme ostatní funkce odvozovat pomocí následujících *operátorů*.

(IV) *Substituce*. Je-li f funkce m proměnných a $g_1, g_2, \ldots, g_m$ jsou funkce n proměnných, pak operátor S_n^m přiřadí funkci f a funkcím $g_1, g_2, \ldots, g_m$ funkci n proměnných h, pro kterou platí:

$$h(x_1, \ldots, x_n) \simeq f\big(g_1(x_1, x_2, \ldots, x_n), \ldots, g_m(x_1, x_2, \ldots, x_n)\big)$$

(V) *Primitivní rekurze*. Nechť $n \geq 2$, pak funkci $n-1$ proměnných f a funkci $n+1$ proměnných g přiřadí operátor primitivní rekurze $R_n(f, g)$ funkci n proměnných h, pro niž platí:

$$h(x_1, x_2, x_3, \ldots, x_n) \simeq \begin{cases} f(x_2, x_3, \ldots, x_n) & x_1 = 0 \\ g(x_1 - 1, h(x_1 - 1, x_2, \ldots, x_n), x_2, x_3, \ldots, x_n) & x_1 > 0 \end{cases}$$

Odvození funkce f je pak konečná posloupnost funkcí $f_1, f_2, \ldots, f_k = f$, kde funkce f_i, $1 \leq i \leq k$ je buď základní funkce, nebo je odvozena pomocí některého operátoru z funkcí $\{f_j \mid 1 \leq j \leq i\}$. Součástí odvození je i informace o tom, které operátory a na které funkce byly použity.

Funkce je *primitivně rekurzivní* (*PRF*), pokud existuje její odvození ze základních funkcí pomocí operátorů substituce a primitivní rekurze. ◂

Všimněme si, že již základních funkcí je nekonečně mnoho, a to díky funkcím projekce I_n^j. Dá se také říci, že třída primitivně rekurzivních funkcí je nejmenší třídou funkcí, jež obsahuje všechny základní funkce a je uzavřena na skládání funkcí (substituci) a použití primitivní rekurze.

Poznámka 4.4.2 *Zatímco význam operátoru substituce je zřejmý, neboť tento umožňuje skládání funkcí, tedy použití již odvozené funkce, význam operátoru primitivní rekurze je třeba si trochu objasnit. Na první pohled to vypadá, že implementace tohoto operátoru v procedurálním jazyku by vyžadovala použití rekurzivního volání, ale ve skutečnosti lze nahlédnout, že primitivní rekurze odpovídá pascalovskému cyklu* **`for`** *(tedy omezenému cyklu s předem danými pevnými hraničními hodnotami a krokem) a naopak i pascalovský cyklus* for *lze přepsat pomocí primitivní rekurze. Pro jednoduchost budeme uvažovat* $n = 2$*, což je nejmenší hodnota* n*, pro kterou má smysl použít operátor primitivní rekurze* $R_n(f, g)$*. Toto omezení plyne z požadavku, že funkce* f *má mít* $n - 1$ *proměnných, přičemž každá funkce musí mít alespoň jednu proměnnou, proto pomocí primitivní rekurze není možné odvodit funkci jedné proměnné. Pokud přece potřebujeme odvodit funkci jedné proměnné, není problém přidat pomocí substituce jednu umělou proměnnou, která nebude využita, to uvidíme buď v rámci cvičení nebo dále v některých důkazech.*

Uvažme tedy funkce f *s jedním parametrem a* g *s třemi parametry a nechť* $h = R_2(f, g)$*, tedy funkce* h *je odvozena z funkcí* f *a* g *primitivní rekurzí. Můžeme tedy psát:*

$$\begin{aligned}
h(0, x_2) &= f(x_2) \\
h(1, x_2) &= g(0, h(0, x_2), x_2) \\
h(2, x_2) &= g(1, h(1, x_2), x_2) \\
h(3, x_2) &= g(2, h(2, x_2), x_2) \\
&\vdots \quad\quad \vdots \\
h(x_1 - 1, x_2) &= g(x_1 - 2, h(x_1 - 2, x_2), x_2) \\
h(x_1, x_2) &= g(x_1 - 1, h(x_1 - 1, x_2), x_2)
\end{aligned}$$

Je tedy vidět, že hodnotu funkce $h(x_1, x_2)$ *lze spočítat následujícím cyklem:*

```
1: function h(x_1, x_2)
2:     z ← f(x_1).
3:     for y ← 0 to x_1 − 1 do
4:         z ← g(y, z, x_2)
5:     end for
6:     return z
7: end function
```

Není ani těžké nahlédnout, že naopak cyklus tohoto typu lze přepsat pomocí primitivní rekurze. To nám nadále velmi ušetří naše úvahy, protože nám to dovolí uvažovat pomocí prostředků, které jsou dnes běžnému programátorovi blízké.

Většina běžných funkcí je primitivně rekurzivní, některé si shrneme v následujícím tvrzení.

Lemma 4.4.3 *Následující funkce jsou primitivně rekurzivní: Konstanta $c \in \mathbb{N}$, $x + y$, $x \cdot y$, x^y, $x!$, $|x - y|$, $\min\{x, y\}$, $\max\{x, y\}$, $x \dot{-} y = \max(x - y, 0)$, $\operatorname{sign}(x)$ (0 pokud je $x = 0$, 1, pokud je $x > 0$), $x \operatorname{div} y$ (celočíselné dělení), $x \bmod y$ (zbytek po celočíselném dělení), $\lfloor \log_x(y) \rfloor$ (pro $x > 1$, $y > 0$, případy pro $x \leq 1$ nebo $y = 0$ můžeme dodefinovat třeba 0), $\operatorname{parita}(x)$ (vrací 1, je-li x liché číslo, a 0, je-li x sudé číslo).*

Důkaz: Důkazy jsou většinou jednoduché a odvození ponecháme čtenáři jako cvičení.□

My si ukážeme na příklad odvození funkce sčítání.

Příklad 4.4.4

Ukážeme si odvození funkce $\textsc{Add}(a, b) = a + b$. Máme-li k dispozici jen přičítání jedničky, můžeme k výpočtu použít následující for cyklus, v němž b-krát přičteme jedničku k a:

```
function Add(a, b)
    z ← b
    for y ← 0 to a − 1 do
        z ← z + 1
    end for
    return z
end function
```

Z poznámky 4.4.2 již víme, jak takový cyklus přepsat pomocí primitivní rekurze. Pokud bychom měli funkce $f(b) = b$ a $g(y, z, b) = z + 1$, mohli bychom rovnou použít $\textsc{Add} = R_2(f, g)$. Funkce f je projekce I_1^1, neboť $I_1^1(b) = b$ je funkcí identity. Funkci g lze napsat pomocí základních funkcí jako $g(y, z, b) = s(I_3^2(y, z, b))$, tedy následník druhé ze tří vstupních proměnných.

Takové odvození nám obvykle bude stačit ve chvíli, kdy budeme potřebovat odvodit primitivně či částečně rekurzivní funkci. Ve skutečnosti nebudeme často ani zmiňovat použití projekce, neboť to je obvykle zcela přímočaré, podobně místo použití funkce následníka s budeme prostě používat přičítání 1. Od této chvíle navíc víme, že sčítání je primitivně rekurzivní, a tak bychom již při odvozování dalších funkcí, např. násobení, využili přímo sčítání a nemuseli je odvozovat znovu.

Pro úplnost si zde uvedeme i formální odvození (ve smyslu definice 4.4.1), to

můžeme ve zkratce zapsat jako:

$$\textsc{Add} = R_2(I_1^1, S_3^1(s, I_3^2)) \tag{4.2}$$

Z toho je již zřejmé, jak vypadá posloupnost funkcí odvozující Add:

$f_1 = s(x)$	funkce následníka (II)
$f_2 = I_1^1(x)$	projekce (III)
$f_3 = I_3^2(x_1, x_2, x_3)$	projekce (III)
$f_4 = S_3^1(f_1, f_3) = \lambda x_1 x_2 x_3[x_2 + 1]$	substituce (IV)
$\textsc{Add} = f_5 = R_2(f_2, f_4)$	primitivní rekurze (V)

Poslední řádek odvození s použitím primitivní rekurze má následující význam:

$$\begin{aligned} \textsc{Add}(0, x_2) &= f_2(x_2) = x_2 \\ \textsc{Add}(x_1 + 1, x_2) &= f_4(x_1, f_5(x_1, x_2), x_2) = \textsc{Add}(x_1, x_2) + 1 \end{aligned}$$

V dalším textu si vystačíme s daleko méně formálním přístupem a zastavíme se obvykle na vyšší úrovni ve chvíli, kdy bude již jasné, jak bychom se tohoto formálního odvození dobrali.

Naším cílem je popsat výpočetní model, který by nám umožnil zformalizovat pojem intuitivního algoritmu. Primitivní funkce tento požadavek nemohou splňovat. Je-li f primitivně rekurzivní funkce, pak je definovaná pro všechny vstupy (platí to pro základní funkce a není těžké nahlédnout, že operátory substituce a primitivní rekurze vždy z totálních funkcí odvodí totální funkci). Ovšem v jiných výpočetních modelech (zatím jsme si zmiňovali jen Turingův stroj a RAM) je možné implementovat algoritmus, který se nezastaví pro všechny vstupy. Dalším důvodem je, že existuje řada (i totálních) funkcí, které považujeme za vyčíslitelné intuitivním algoritmem, ale nejsou přitom primitivně rekurzivní. O tom nás přesvědčí jednoduchá úvaha. Odvození primitivně rekurzivní funkce f lze popsat řetězcem v nějaké vhodné abecedě (například v podobě, v jaké jsme popsali odvození funkce Add, viz (4.2)). Tento řetězec lze vždy přepsat do binární abecedy $\{0, 1\}$ a každému takovému řetězci můžeme přiřadit přirozené číslo, kterému budeme říkat Gödelovo číslo.[8] Primitivně rekurzivní funkci s jedním parametrem, která má v tomto očíslování číslo x si pro tuto chvíli označme jako f_x. Uvažme nyní funkci φ dvou parametrů, která je definovaná jako

$$\psi(x, y) \simeq f_x(y).$$

Takto definovaná funkce je *univerzální pro třídu primitivně rekurzivních funkcí*. Ukažme si, že funkce ψ nemůže být primitivně rekurzivní.

[8] Podrobněji to provedeme pro případ Turingových strojů v kapitole 5.2, pro účely této úvahy nejsou technické detaily kódování podstatné.

Věta 4.4.5 *Univerzální primitivně rekurzivní funkce $\psi(x,y) \simeq f_x(y)$ není primitivně rekurzivní.*

Důkaz: Předpokládejme sporem, že ψ je primitivně rekurzivní, potom je primitivně rekurzivní i funkce $g(x) \simeq \psi(x,x) + 1 \simeq f_x(x) + 1$, kterou odvodíme z funkce ψ a funkce následníka s pomocí substituce. V tom případě této funkci přináleží Gödelovo číslo z a tedy $f_z(x) \simeq g(x)$. Ovšem platí, že $f_z(z) \simeq g(z) \simeq f_z(z) + 1$, tedy $f_z(z) \simeq f_z(z) + 1$. To ovšem není možné uvážíme-li, že funkce ψ, g i f_z jsou definované pro všechny vstupy. Dostáváme tedy spor a funkce ψ, která je univerzální pro PRF nemůže existovat.[9] To znamená, že primitivně rekurzivní funkce nejsou dost silné na to, abychom v nich mohli naprogramovat univerzální funkci pro tuto třídu. Přitom je ovšem algoritmus výpočtu primitivně rekurzivní funkce velmi jednoduchý, a tedy očekáváme, že výpočetní model, který by měl zachycovat pojem intuitivního algoritmu, umožní implementovat výpočet primitivně rekurzivní funkce. □

Co nám navíc oproti intuitivnímu pojmu algoritmu chybí, je obecný **`while`** cyklus řízený logickou podmínkou, a tedy potenciálně neukončený. Primitivní rekurze odpovídá pouze *for* cyklu, u kterého dopředu víme, kolik smyček nejvýš provede.

Částečně rekurzivní funkce

Operátor, který přidáme k interpretaci potenciálně neomezeného cyklu **`while`**, je operátor efektivní minimalizace.

Definice 4.4.6 (Efektivní minimalizace) (VII) (*Efektivní*) *minimalizace*. Je-li f funkce $n+1$ proměnných, pak $M_n(f)$ určuje funkci n proměnných h, pro kterou platí

$$h(x_1, x_2, \ldots, x_n) \simeq \min\{y \mid f(x_1, x_2, \ldots, x_n, y)\downarrow = 0 \tag{4.3}$$

$$\text{a } (\forall z \leq y)[f(x_1, x_2, \ldots, x_n, z)\downarrow]\}. \tag{4.4}$$

Tedy h nabývá nejmenší hodnoty y, pro niž je hodnota funkce $f(x_1, \ldots, x_n, y)$ definovaná a rovna 0. Navíc požadujeme, aby pro všechny hodnoty z nižší než y byla hodnota funkce $f(x_1, \ldots, x_n, y)$ definována. Pro operátor minimalizace budeme používat následující značení:

$$h(x_1, x_2, \ldots, x_n) \simeq \lambda x_1 x_2 \ldots x_n \Big[\mu y[f(x_1, x_2, \ldots, x_n, y) \simeq 0]\Big] \quad \blacktriangleleft$$

Definice 4.4.7 (Částečně rekurzivní funkce) Funkce f je *částečně rekurzivní* (*ČRF*), pokud ji lze odvodit ze základních funkcí pomocí substituce, primitivní rekurze a minimalizace. Funkce f je *obecně rekurzivní* (*ORF*), pokud je f ČRF, která je definovaná pro všechny vstupy. ◂

[9] V důkazu jsme použili techniku diagonalizace.

Uvědomme si, že výraz $\mu y[f(x_1, x_2, \ldots, x_n, y) \simeq 0]$ obecně neoznačuje „nejmenší číslo y, pro které je $f(x_1, x_2, \ldots, x_n, y) \simeq 0$“, a to díky podmínce (4.4) v definici minimalizačního operátoru. Podmínka (4.4) požadující, aby byla hodnota $f(x_1, x_2, \ldots, x_n, z)$ definovaná pro všechny hodnoty $z \leq y$, je navíc velmi důležitá, protože odpovídá tomu, co intuitivně považujeme za efektivní minimalizaci. Rozmysleme si, jak by program v běžném imperativním jazyku počítal hodnotu funkce h: Postupně by počítal hodnoty $f(x_1, \ldots, x_n, 0)$, $f(x_1, \ldots, x_n, 1)$, ..., až by nalezl první y, pro něž by platilo $f(x_1, \ldots, x_n, y) = 0$. Tento postup odpovídá následujícímu **`while`** cyklu:

```
1: function h(x_1, ..., x_n)
2:     y ← 0
3:     while f(x_1, x_2, ..., x_n, y) ≠ 0 do
4:         y ← y + 1
5:     end while
6:     return y
7: end function
```

V tomto algoritmu jistě platí, že pokud jedna hodnota $f(x_1, \ldots, x_n, z)$ pro nějaké $z \leq y$ není definována, což odpovídá tomu, že výpočet $f(x_1, \ldots, x_n, z)$ neskončí, pak hodnota funkce $h(x_1, \ldots, x_n)$ není definovaná. Podobně i v případě, kdy $f(x_1, \ldots, x_n, z)$ je sice definovaná pro každou hodnotu $z \leq y$, ale pro žádnou z nich není nulová, není hodnota funkce $h(x_1, \ldots, x_n)$ definovaná.

Navíc platí, že pokud bychom podmínku (4.4) vynechali a definovali bychom funkci h odvozenou minimalizačním operátorem jako

$$h(x_1, x_2, \ldots, x_n) = \min\{y \mid f(x_1, x_2, \ldots, x_n, y)\downarrow = 0\},$$

nebyly by částečně rekurzivní funkce uzavřené na operaci minimalizace. Všimněme si také, že pokud je funkce f definovaná pro všechny vstupy (je tedy obecně rekurzivní), pak je podmínka (4.4) splněna automaticky a funkce $h = M_n(f)$ v tomto případě skutečně hledá nejmenší hodnotu y, pro kterou $f(x_1, \ldots, x_n, y) = 0$.

Příkladem funkce, která je částečně rekurzivní, ale není obecně rekurzivní může být funkce, která není definovaná pro žádný vstup:

$$\lambda x\Big[\mu y[s(y) \simeq 0]\Big].$$

Příkladem funkce, která je obecně rekurzivní, ale není primitivně rekurzivní je univerzální funkce pro PRF, jak jsme již ukázali ve větě 4.4.5. Dalším příkladem je Ackermannova funkce, která je sice definovaná pro všechny vstupy, ale roste rychleji, než jakákoli primitivně rekurzivní funkce. Fakt, že je daná funkce všude definovaná tedy ještě neznamená, že se při jejím výpočtu obejdeme bez **`while`** cyklu.

Na závěr této podkapitoly zmiňme, že odvozování částečně rekurzivní funkce má blízko k funkcionálnímu programování. I když díky tomu, že primitivní rekurze odpovídá **`for`** cyklu a minimalizace **`while`** cyklu, můžeme popisovat částečně rekurzivní funkce i imperativním (či procedurálním) způsobem.

4.4.2. Základní vlastnosti PRF, ORF a ČRF

Při popisu vlastností rekurzivních funkcí se nám bude hodit pojem rekurzivního a rekurzivně spočetného predikátu. Připomeňme, že predikáty a relace byly definovány v kapitole 3.2.2.

Definice 4.4.8 Predikát (nebo relace) $R \subseteq \mathbb{N}^n, n \geq 1$ je *primitivně* (resp. *obecně*) *rekurzivní predikát* (*PRP*, *ORP*), pokud je jeho charakteristická funkce primitivně (resp. obecně) rekurzivní. Obecně rekurzivním predikátům a relacím budeme též říkat *rekurzivní*.

Predikát (nebo relace) $R \subseteq \mathbb{N}^n, n \geq 1$ je *rekurzivně spočetný* (*RSP*), pokud existuje funkce n proměnných $f_R : \mathbb{N}^n \to \mathbb{N}$, pro kterou platí, že

$$R = \operatorname{dom} f.$$

Hodnota funkce f_R je tedy pro danou n-tici $x_1, \ldots, x_n$ definovaná právě když $R(x_1, \ldots, x_n)$. Hodnota funkce f_R není v tomto případě důležitá.

Unární rekurzivní (resp. rekurzivně spočetný) predikát $A \subseteq \mathbb{N}$ budeme též nazývat *rekurzivní množinou* (resp. rekurzivně spočetnou množinou). ◂

Zřejmě každý primitivně rekurzivní predikát je současně obecně rekurzivní, a tedy rekurzivní, naopak to však platit nemusí. Podobně každý rekurzivní predikát je i rekurzivně spočetný, ale naopak to neplatí. Totéž lze přirozeně říci o množinách. Většina obvyklých relací a predikátů je primitivně rekurzivní, ať již jde o běžné relace jako $<, >, =$ nebo testování prvočíselnosti predikátem $prime(x)$.

Příklad 4.4.9

Například relace $x_1 \geq x_2$ je primitivně rekurzivní, neboť $\chi_{\geq}(x_1, x_2) \simeq \operatorname{sign}(x_1 \dot{-} x_2)$.

Poznamenejme, že jako podmínku jsme v definici 4.4.6 operátoru minimalizace povolili pouze test toho, zda je hodnota vložené funkce $f(x_1, \ldots, x_n)$ definována a rovna 0. Ve skutečnosti je však možno připustit libovolný rekurzivní predikát, protože v tom případě bychom mohli v operátoru minimalizace použít jeho charakteristickou funkci. My tohoto faktu budeme používat a v minimalizaci používat libovolné podmínky. Zvláště pro libovolný obecně rekurzivní predikát $R(x_1, \ldots, x_n, y)$ budeme používat zápis

$$\mu y[R(x_1, \ldots, x_n, y)] \simeq \mu y[(1 \dot{-} \chi_R(x_1, \ldots, x_n, y)) \simeq 0]$$

Všimněme si, že je-li R obecně rekurzivní predikát a funkce χ_R je tedy obecně rekurzivní, je podmínka (4.4) v definici 4.4.6 automaticky splněna a tento zápis tedy skutečně znamená „nejmenší y, pro něž je predikát $R(x_1, \ldots, x_n, y)$ splněn“.

Zmíníme dále pár jednoduchých vlastností PRF, ORF, ČRF, rekurzivních a rekurzivně spočetných predikátů. Všechna následující tvrzení lze pochopitelně jednoduše zobecnit pro libovolný počet proměnných, my budeme pro jednoduchost uvažovat funkce a predikáty co nejmenšího počtu proměnných. V následujících tvrzeních lze navíc vždy nahradit „PRF“ pomocí „ORF“, tedy co lze ukázat pro primitivně rekurzivní funkce,

platí i pro obecně rekurzivní funkce. Navíc řadu důkazů lze zopakovat i pro částečně rekurzivní funkce a rekurzivně spočené predikáty.

Lemma 4.4.10 (Konečný součet a součin) *Je-li f PRF dvou proměnných (pro jednoduchost), potom i funkce $g(z,x) = \sum_{y<z} f(y,x)$ (přičemž $g(0,x) = 0$) a $h(z,x) = \prod_{y<z} f(y,x)$ (přičemž $h(0,x) = 1$) jsou PRF.*

Důkaz: Předpokládáme, že sčítání i násobení jsou primitivně rekurzivní operace. Potom $g(z,x)$ můžeme spočítat následujícím cyklem.

```
1: function g(z,x)
2:     s ← 0
3:     for y ← 0 to z − 1 do
4:         s ← s + f(y,x)
5:     end for
6:     return s
7: end function
```

Inicializace nulovou funkcí odpovídá situaci, kdy bychom chtěli sčítat 0 sčítanců, proto určíme tuto hodnotu podle definice jako 0. Přepíšeme-li tento cyklus pomocí primitivní rekurze ve smyslu poznámky 4.4.2, můžeme $g(z,x)$ odvodit pomocí primitivní rekurze jako $g = R_2(o, \lambda abc[b + f(a,c)])$.

Odvození funkce h je zcela shodné se záměnou sčítání za násobení a konstanty 0 za konstantu 1. □

Kdybychom chtěli spočítat $g(z) = \sum_{y<z} f(y)$, kde f je PRF jedné proměnné, nemůžeme použít přímo primitivní rekurzi, neboť pomocí ní nelze odvodit funkci jedné proměnné. Můžeme však zavést umělou proměnnou, jejíž hodnotu nikde nepoužijeme, přesněji, podle lemmatu 4.4.10 bychom odvodili funkci

$$g'(z,x) = \sum_{y<z} f'(y,x),$$

kde $f'(y,x) \simeq f(y)$. Poté bychom položili $g(z) \simeq g'(z,z)$. Dostaneme tak jednoduchý důsledek.

Důsledek 4.4.11 *Je-li f PRF jedné proměnné, potom i funkce $g(z) = \sum_{y<z} f(y)$ (přičemž $g(0) = 0$) a $h(z) = \prod_{y<z} f(y)$ (přičemž $h(0) = 1$) jsou PRF.*

Další užitečné tvrzení nám umožní použití podmíněného příkazu.

Lemma 4.4.12 (Podmíněný příkaz) *Ať $g_1(x), \ldots, g_n(x)$, $n > 0$ jsou PRF jedné proměnné (opět pro jednoduchost) a nechť $R_1(x), \ldots, R_n(x)$ jsou primitivně rekurzivní predikáty jedné proměnné, pro něž platí, že pro každé $x \in \mathbb{N}$ je splněn právě jeden z nich. Potom funkce f defino-*

vaná následujícím předpisem je PRF:

$$\begin{aligned} f(x) = g_1(x) &\Leftrightarrow R_1(x) \\ f(x) = g_2(x) &\Leftrightarrow R_2(x) \\ &\vdots \\ f(x) = g_n(x) &\Leftrightarrow R_n(x) \end{aligned}$$

Důkaz: Funkci f můžeme zapsat následujícím způsobem

$$f(x) \simeq g_1(x) \cdot \chi_{R_1}(x) + g_2(x) \cdot \chi_{R_2}(x) + \cdots + g_n(x) \cdot \chi_{R_n}(x),$$

kde $\chi_{R_1}, \chi_{R_2}, \ldots, \chi_{R_n}$ jsou primitivně rekurzivní charakteristické funkce primitivně rekurzivních predikátů $R_1(x), R_2(x), \ldots, R_n(x)$. Protože součet i součin primitivně rekurzivních funkcí vede podle lemmatu 4.4.3 opět k primitivně rekurzivní funkci, je f primitivně rekurzivní funkce. □

Lemma 4.4.12 nabízí analogii dvou důležitých struktur z vyšších programovacích jazyků, jednak **`if-then-else`** v případě $n = 2$, jednak **`case`** (případně **`switch`**) pro obecné $n > 0$. K tomu uvažme, že pokud splňují predikáty $R_1, \ldots, R_n$ předpoklady lemmatu 4.4.12, pak musí platit, že

$$R_n(x) \Leftrightarrow \neg R_1(x) \wedge \neg R_2(x) \wedge \cdots \wedge \neg R_{n-1}(x),$$

dá se proto říci, že $R_n(x)$ určuje větev **`else`** příkazu **`if`**, či implicitní (**`default`**) větev příkazu **`case`** či **`switch`**.

Lemma 4.4.13 (Omezená kvantifikace) *Mějme primitivně rekurzivní predikát P (pro jednoduchost binární), potom $V_P(z,x) = (\forall y < z)[P(y,x)]$ a $E_P(z,x) = (\exists y < z)[P(y,x)]$ jsou primitivně rekurzivní predikáty.*

Důkaz: Označme pomocí χ_P charakteristickou funkci P, pomocí χ_V charakteristickou funkci V_P a pomocí χ_E charakteristickou funkci E_P. Potom platí:

$$\begin{aligned} \chi_V(z,x) &= \prod_{y<z} \chi_P(y,x) \\ \chi_E(z,x) &= \operatorname{sign}\Big(\sum_{y<z} \chi_P(y,x) \Big) \end{aligned}$$

Tvrzení proto plyne přímo z lemmatu 4.4.10 a lemmatu 4.4.3. □

Situace s neomezenými kvantifikátory je poněkud komplikovanější a budeme se jí ještě dále věnovat v případě rozhodnutelných a částečně rozhodnutelných problémů. Pokud je P primitivně rekurzivní predikát, pak $(\exists y)[P(y)]$ je rekurzivně spočetný predikát, který však není nutně obecně rekurzivní, $(\forall y)[P(y)]$ nemusí být ani rekurzivně spočetný, ale jde obecně o doplněk rekurzivně spočetného predikátu $(\exists y)[\neg P(y)]$.

Lemma 4.4.14 (Logické spojky) *Jsou-li P a R primitivně rekurzivní predikáty (pro jednoduchost unární), pak i $R \wedge P$, $R \vee P$ a $\neg P$ jsou primitivně rekurzivní predikáty.*

Důkaz: Platí:

$$\begin{aligned}\chi_{P\wedge R}(x) &= \chi_P(x)\cdot\chi_R(x)\\ \chi_{P\vee R}(x) &= \operatorname{sign}\big(\chi_P(x)+\chi_R(x)\big)\\ \chi_{\neg P}(x) &= 1\dot{-}\chi_P(x)\end{aligned}$$

Proto jsou příslušné charakteristické funkce primitivně rekurzivní. □

Zatímco konjunkci a disjunkci bychom mohli přeříkat i pro rekurzivně spočetné predikáty, negací rekurzivně spočetného predikátu je rekurzivně spočetný predikát právě když jsou oba rekurzivní. K tomu se ještě vrátíme později v části věnované rozhodnutelným a částečně rozhodnutelným problémům.

Lemma 4.4.15 (Konečná konjunkce a disjunkce) *Je-li P primitivně rekurzivní predikát dvou proměnných, pak i predikáty $A(x,z) = \bigwedge_{y<z} P(x,y)$ a $B(x,z) = \bigvee_{y<z} P(x,y)$ jsou primitivně rekurzivní.*

Důkaz: Jistě platí $A(x,z) = (\forall y < z)[P(x,y)]$ a $B(x,z) = (\exists y < z)[P(x,y)]$, jde tedy o důsledek lemmatu 4.4.13. □

Narozdíl od neomezené efektivní minimalizace definované v definici 4.4.6 je omezená minimalizace primitivně rekurzivní.

Lemma 4.4.16 (Omezená minimalizace) *Je-li P primitivně rekurzivní predikát (pro jednoduchost binární), potom (binární) funkce f definovaná následujícím způsobem je primitivně rekurzivní.*

$$f(x,z) = \begin{cases}\min\{y < z \mid P(x,y)\} & \text{pokud takové } y \text{ existuje}\\ z & \text{jinak.}\end{cases}$$

Funkci f budeme také označovat pomocí $f(x,z) = \mu y < z[P(x,y)]$.

Důkaz: Označme si pomocí χ_P charakteristickou funkci predikátu P, stejně jako predikát P, je χ_P primitivně rekurzivní. Uvažme, jak bychom hodnotu funkce f spočítali, nemáme-li k dispozici cyklus **`while`**, mohli bychom použít třeba následující cyklus **`for`**:

```
1:  function f(x,z)
2:      u ← 0                    ▷ V u si budeme pamatovat návratovou hodnotu funkce.
3:      for y ← 0 to z − 1 do
4:          if P(x,y) then
5:              break
6:          end if
7:          u ← u + 1
8:      end for
9:      return u
10: end function
```

Cyklus **`for`** odpovídá primitivní rekurzi a podmíněný příkaz **`if`** můžeme použít díky lemmatu 4.4.12. Co však nemáme k dispozici, je příkaz **`break`** pro vyskočení z cyklu

ven. Místo ukončení cyklu předčasně jej tedy necháme doběhnout až do konce, přičemž od chvíle, kdy algoritmus nalezne hodnotu y, pro kterou je predikát $P(x, y)$ splněn, nebude již dále tuto hodnu měnit. To můžeme zabezpečit následujícím cyklem:

```
1: function f((x,z))
2:     u := 0                          ▷ V u si budeme pamatovat návratovou hodnotu.
3:     for y ← 0 to z − 1 do
4:         if ¬P(x,u) then
5:             u ← u + 1               ▷ Ještě jsme nenašli vhodnou hodnotu.
6:         end if
   ▷ Pokud byl splněn predikát P(x,u), necháme u beze změny.
7:     end for
8:     return u
9: end function
```

V cyklu zvyšujeme hodnotu u do chvíle, kdy není predikát $P(x, u)$ splněn. Ve chvíli, kdy je poprvé $P(x, u)$ splněn, ponecháme u beze změny a protože od té doby bude $P(x, u)$ platit stále. Na konci zůstane v u nejmenší hodnota, pro kterou byl predikát $P(x, u)$ splněn. Nyní už má funkce reprezentující tělo cyklu správný tvar, který odpovídá následující funkci:

$$g(y, u, x) \simeq \begin{cases} u & \text{pokud } P(x, u) \\ u + 1 & \text{jinak} \end{cases}$$

Tato funkce je primitivně rekurzivní[10] podle lemmatu 4.4.12 a toho, že $P(x, u)$ i $\neg P(x, u)$ jsou primitivně rekurzivní predikáty, první jmenovaný dle předpokladu, druhý dle lemmatu 4.4.14. Výslednou funkci f odvodíme primitivní rekurzí z nulové funkce a funkce g, tedy $f = R_2(o, g)$. □

Lemmata 4.4.10, 4.4.12, 4.4.16, 4.4.13, 4.4.14 a 4.4.15 lze přeformulovat i pro ORF a ORP, pro ČRF a RSP platí rovněž téměř všechna. Výjimku tvoří negace predikátu v lemmatu 4.4.14, jak jsme již zmínili v komentáři za tímto tvrzením.

4.4.3. Cvičení

Definice a odvozování primitivně a částečně rekurzivních funkcí

1. Ukažte, že funkce násobení je primitivně rekurzivní, předpokládejte při tom, že sčítání primitivně rekurzivní je (viz příklad 4.4.4) a nemusíte je odvozovat.

2. Ukažte, že následující funkce jsou primitivně rekurzivní, můžete při tom použít již odvozené funkce. (Nyní máme sčítání a násobení.)

   a) $c_k(x) = k$ (obecná konstantní funkce pro konstantu k, k je zde součást jména, nikoli parametr, víme, že $c_0(x) = o(x)$ je základní funkce)

[10] Ve skutečnosti je to zřejmé, protože bychom mohli přímo psát $g(y, u, x) \simeq u + (1 \dot{-} \chi_P(x, u))$.

b) $sign(x)$ (0 pro $x = 0$, 1 jinak)

c) $x \dot{-} 1$ ($x - 1$ pro $x > 0$, 0 pro $x = 0$)

d) $x \dot{-} y$ ($x - y$ pro $x \geq y$, 0 jinak)

e) $|x - y|$

f) $x \operatorname{div} y$ (celočíselné dělení)

g) $x \bmod y$ (zbytek po celočíselném dělení)

h) $\min(x, y)$, $\max(x, y)$

i) x^y

j) $x!$

3. Ukažte, že následující relace a predikáty jsou primitivně rekurzivní.
   a) Porovnávání: $<$, $\leq$, $=$, $\geq$, $>$.
   b) $prime(x)$ (splněný pokud x je prvočíslo)
4. Ukažte, že následující funkce jsou částečně rekurzivní.
   a) Funkce $f(x)$, pro kterou platí, že není definovaná pro žádný vstup, tj. $(\forall x)[f(x)\uparrow]$.
   b) Funkce $f(x, y)$, pro kterou platí, že $f(x, y)\downarrow \Leftrightarrow y \leq x$.
   c) Funkce $f(x, y)$, pro kterou platí, že $f(x, y)\downarrow \Leftrightarrow (\exists k)[y = kx]$.

S-m-n věta

V některých cvičení se využívá následující číslování rekurzivně spočetných množin.

$$W_e = \mathrm{dom}\varphi_e = \{x \mid \varphi_e(x)\downarrow\}.$$

kde φ_e označuje částečně rekurzivní funkci s Gödelovým číslem e. Jde o funkci počítanou Turingovým strojem M_e s Gödelovým číslem e. Viz též sekci 4.5.

5. S použitím s-m-n věty ukažte, že existuje prostá PRF $f(x, y)$, pro kterou platí, že

$$\varphi_{f(x,y)}(u) \simeq \varphi_x(u) + \varphi_y(u).$$

   (*Připomeňte si příklad 4.5.12. Toto cvičení zde slouží jen jako vzor.*)
6. S použitím s-m-n věty ukažte, že existuje prostá PRF $f(x)$, pro kterou platí, že

$$\varphi_{f(x)}(y) \simeq x^y.$$

7. S použitím s-m-n věty ukažte, že existuje prostá PRF $f(x)$, pro kterou platí, že

$$W_{f(x)} = \{0, \ldots, x\} = \{y \mid y \leq x\}.$$

8. S použitím s-m-n věty ukažte, že existuje prostá PRF $f(x)$, pro kterou platí, že

$$W_{f(x)} = \{kx \mid k \in \mathbb{N}\}.$$

4.5. Ekvivalence Turingových strojů a ČRF $*$

V této sekci probereme jednak ekvivalenci ČRF s Turingovsky vyčíslitelnými funkcemi, dále ekvivalenci pojmu rekurzivní množiny či predikátu s rekurzivním jazykem a rekurzivně spočetné množiny či predikátu s rekurzivně spočetným jazykem. Dospějeme také k univerzální ČRF a Kleeneho větě o normální formě.

Věta 4.5.1 *1. Je-li h ČRF n proměnných, pak h je Turingovsky vyčíslitelná. Přesněji, existuje Turingův stroj M_h takový, že pro každou n-tici přirozených čísel $x_1, x_2, \ldots, x_n$ platí*

$$M_h((x_1)_B\#(x_2)_B\#\ldots\#(x_n)_B)\downarrow \Leftrightarrow h(x_1, x_2, \ldots, x_n)\downarrow$$

a platí-li $h(x_1, x_2, \ldots, x_n)\downarrow = y$, potom výpočet Turingova stroje M_h vydá na výstupu řetězec $(y)_B$.

2. Převod je navíc možno učinit efektivně. Jinými slovy, existuje Turingův stroj CRF2TS, *který pokud na vstupu dostane kód ČRF h, spočítá Gödelovo číslo e stroje M_e, který počítá funkci h.*

Důkaz: Protože jsme přesněji nespecifikovali kódování ČRF, je zřejmé, že nemůžeme detailně dokázat druhý bod znění věty. Tento důkaz by byl zbytečně technický, a tak nám bude stačit, zůstaneme-li na intuitivní úrovni. Přesněji, vzhledem k tomu, že důkaz prvního bodu bude konstruktivní, ponecháme již čtenáři k uvážení, že popsané konstrukce by šlo implementovat na Turingově stroji. Ani důkaz prvního bodu však nebudeme provádět příliš detailně a zůstaneme na vyšší intuitivnější úrovni popisu konstruovaného Turingova stroje.

Tvrzení ukážeme indukcí podle délky (nebo struktury) odvození funkce h. Předpokládejme nejprve, že h je jedna ze základních funkcí, tento případ je sice triviální, ale my jej přece jen alespoň stručně provedeme.

(I) *h je konstantní nulová funkce $o(x)$.* TS M_h prostě smaže obsah pásky, zapíše 0 a skončí.

(II) *h je funkce následníka $s(x)$.* TS M_h implementuje přičtení jedničky k binárnímu číslu.

(III) *h je projekce $I_n^j(x_1, \ldots, x_n)$.* TS M_h přeskočí $j-1$ bloků 0 a 1 oddělených # spolu s jejich smazáním. Poté přeskočí j-tý blok, ale nechá jej být. Následně M_h smaže následujících $n - j + 1$ bloků a vrátí se na začátek toho jediného bloku, který zbyl. Čísla j a n jsou součástí stroje M_h, proto je možné je využívat i ve stavu, což usnadňuje počítání toho, kolik bloků je třeba ještě smazat a přeskočit.

Zřejmě všechny základní funkce jsou reprezentovány pomocí konkrétních TS, které mají konkrétní kódy a odpovídající čísla, ta je možno efektivně najít i v případě projekce pro zadané j a n. Nyní předpokládejme, že h bylo odvozeno některým odvozovacím pravidlem z již dříve odvozených funkcí. Podle indukčního přepokladu můžeme vždy předpokládat, že pro tyto dříve odvozené funkce máme již zkonstruované Turingovy stroje.

(IV) *Funkce h byla odvozena substitucí z funkce f na m proměnných a funkcí $g_1, g_2, \ldots, g_m$ na n proměnných.* Předpokládejme, že již máme TS $M_f, M_{g_1}, M_{g_2}, \ldots, M_{g_m}$, které počítají funkce $f, g_1, g_2, \ldots, g_m$. Popíšeme práci stroje M_h, který bude počítat funkci h. Stroj popíšeme jako třípáskový, z věty 4.1.12 už víme, že z něj lze zkonstruovat jednopáskový TS, který dělá totéž. Na první pásce si necháme vstup a nebudeme na ni zapisovat, na druhé pásce budeme postupně simulovat práci strojů $M_{g_1}, \ldots, M_{g_m}$ a na třetí pásce budeme na závěr simulovat práci stroje M_f. Přesněji, práce M_h bude vypadat následovně.

a) Pro $i := 1, \ldots, m$ provede M_h postupně následující kroky.

i. Okopíruje vstup na druhou pásku a vrátí se na její začátek.

ii. Provede simulaci M_{g_i} na druhé pásce, M_{g_i} je jednopáskový stroj, proto si s druhou páskou vystačí.

iii. Pokud se M_{g_i} zastaví a zapíše na výstup číslo, pak jej M_h připíše na konec třetí pásky za oddělující znak #. Pokud se výpočet M_{g_i} nezastaví, není hodnota $g_i(x_1, \ldots, x_n)$ definována, a tedy není definována ani hodnota $h(x_1, \ldots, x_n)$, v tom případě se přirozeně nemá zastavit ani M_h.

iv. Stroj M_h smaže obsah druhé pásky.

b) M_h se vrátí na začátek třetí pásky.

c) Na třetí pásce provede M_h simulaci stroje M_f, vstupem je obsah třetí pásky, což jsou hodnoty $g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)$.

d) Je-li výpočet M_f konečný, je po jeho ukončení na třetí pásce uložená hodnota funkce $h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$. To, kde na konec tato hodnota skončí, závisí na tom, jak je stroj M_h převeden na jednopáskový.

(V) *Funkce h byla odvozena primitivní rekurzí z funkcí f na $n-1$ proměnných a funkce g na $n+1$ proměnných.* Opět předpokládejme, že máme již sestrojené stroje M_f a M_g, které počítají funkce f a g, a popišme práci stroje M_h. Bylo by možné popsat stroj M_h s použitím rekurze, kdy by si stroj M_h udržoval zásobník aktivačních záznamů. Jednodušší však bude uvážit, že primitivní rekurze je totéž, co cyklus **`for`**. Přesněji, ve vyšším programovacím jazyce bychom hodnotu funkce $h(x_1, x_2, \ldots, x_n)$ mohli spočítat tímto cyklem:

Tento cyklus již jednoduše implementujeme i na TS. Pro úplnost si zde takový stroj M_h popíšeme, bude mít tři pásky, opět již standardním způsobem bychom jej převedli na jednopáskový. První páska obsahuje vstup, na druhé pásce si budeme pamatovat hodnotu čítače y zapsanou binárně jako $(y)_B$, na třetí pásce budeme simulovat stroje M_f a M_g.

a) M_h nejprve okopíruje na třetí pásku vstup s výjimkou hodnoty x_1.

b) M_h simuluje práci M_f na třetí pásce.

c) M_h zapíše na druhou pásku řetězec 0 (tj. $y = 0$).

d) Dokud řetězec na druhé pásce kóduje číslo menší než první parametr na první pásce, opakuje M_h následující kroky:

   i. Před slovo na třetí pásce (tedy hodnotu z předchozí smyčky z) okopíruje slovo z druhé pásky (tedy hodnotu čítače y) a oddělí je pomocí '#'.

   ii. Na konec třetí pásky přikopíruje obsah první pásky, vyjma první hodnoty x_1, tj. přikopíruje parametry $x_2, \ldots, x_n$.

   iii. Na třetí pásce simuluje práci M_g, který nechá na třetí pásce svůj výstup, tedy hodnotu $g(y, z, x_2, x_3, \ldots, x_n)$.

   iv. Ke čítači y na druhé pásce přičte jedničku.

e) Na závěr třetí páska obsahuje hodnotu $h(x_1, \ldots, x_n)$.

(VI) *Funkce h byla odvozena minimalizací z funkce f na $n+1$ proměnných.* Opět předpokládáme, že máme již sestrojený stroj M_f počítající funkci f. Jak jsme si již řekli, minimalizace odpovídá následujícímu **`while`** cyklu:

Není těžké si představit, jak bude vypadat stroj M_h, který bude provádět tento while cyklus. My si jej opět pro úplnost popíšeme, vystačí si dokonce se dvěma páskami a jeho práce bude vypadat následovně.

a) Na začátku připíše M_h za vstup #0, tedy hodnotu $y = 0$.

b) Dále M_h opakuje následující kroky do té doby, než simulovaný stroj M_f nevrátí 0.

   i. Okopíruje obsah první pásky na druhou pásku.

   ii. Na druhé pásce simuluje stroj M_f.

   iii. Pokud na závěr práce M_f bude na druhé pásce jen slovo reprezentující nulovou hodnotu, pak M_h ukončí opakování cyklu.

   iv. V opačném případě M_h smaže obsah druhé pásky a k hodnotě y uložené za posledním oddělovačem # přičte 1.

c) Nalezená hodnota y je nyní uvedena na první pásce jako poslední parametr.

Konstrukce popsané v důkazu by šlo implementovat na Turingově stroji, ale tomu se my věnovat nebudeme. □

Nyní se zaměříme na opačný směr, tedy fakt, že každá turingovsky vyčíslitelná funkce je ČRF. Na výpočet Turingova stroje můžeme pohlížet jako na posloupnost konfigurací, přičemž výpočet začíná v počáteční konfiguraci a přechod z jedné konfigurace do další je jednoznačně určený přechodovou funkcí (v případě deterministických TS, s nimiž nyní pracujeme). Připomeňme si, že konfigurace TS popisuje kompletní stav výpočtu, skládá se ze stavu, polohy hlavy na pásce a slova na pásce, přičemž z celé pásky v každém okamžiku stačí uvažovat jen její konečnou část od nejlevějšího k nejpravějšímu prázdnému políčku. Výpočet začíná v pevně dané počáteční konfiguraci. Z jedné konfigurace přejde Turingův stroj do další na základě přechodové funkce, jde o lokální a velmi elementární

změnu spočívající v přechodu do nového stavu, změny jednoho znaku ve slově na pásce a pohybu hlavou o nejvýš jedno políčko jedním směrem. Díky tomu i slovo na pásce se v každém kroku prodlouží o nejvýš jeden znak. Tato lokalita úpravy a omezenost délky konfigurace dává tušit, že k manipulaci s konfiguracemi a přechodu pomocí přechodové funkce by měly stačit omezené cykly, tedy primitivní rekurze.

Začneme popisem zakódování konfigurace do čísla tak, aby se s ním dobře pracovalo prostředky primitivně rekurzivních funkcí. Možností, jak takové kódování provést, je celá řada, my navážeme na to, jakým způsobem jsme zakódovali přechodovou funkci Turingova stroje v kapitole 5.2.1. Nejprve zakódujeme konfiguraci řetězcem v abecedě $\Gamma = \{0, 1, L, N, R, |, \#, ;\}$. Nechť se Turingův stroj M nachází ve stavu q_i, na pásce obsahuje slovo $X_{j_1} X_{j_2} \dots X_{j_\ell}$ a hlava čte symbol X_{j_k}, tuto konfiguraci zakódujeme řetězcem:

$$(i)_B \# (j_1)_B \# (j_2)_B \# \dots \# (j_{k-1})_B | (j_k)_B \# \dots \# (j_\ell)_B$$

tj. za binární zápis čísla stavu umístíme symboly slova na pásce oddělené #, přičemž před čtený symbol místo # umístíme |. Nyní převedeme řetězec v abecedě Γ na binární stejně jako u přechodové funkce. S každým binárním řetězcem w máme již jednoznačně asociované číslo, jehož binární zápis je $1w$. Obojí je provedeno stejně jako v kapitole 5.2.1.

Dále můžeme postupovat dvěma způsoby, buď popíšeme primitivně rekurzivní funkci simulující jeden krok TS M, která na vstupu obdrží číslo odpovídající kódu konfigurace K_1 a vrátí číslo odpovídající kódu konfigurace K_2, která z K_1 vznikne použitím přechodové funkce. Funkce simulující jeden krok TS je skutečně primitivně rekurzivní, neboť jak jsme již zmínili, jedná se o lokální a elementární změnu řetězce (k manipulaci s řetězci se však ještě vrátíme). Protože dopředu nevíme, jaký počet kroků musí TS M učinit, než se na daném vstupu zastaví, dojde-li k tomu vůbec, najdeme tento počet kroků pomocí while cyklu, čili minimalizace.

My však budeme postupovat jinak, zakódujeme posloupnost konfigurací tj. výpočet TS do binárního řetězce a popíšeme predikát, který otestuje, zda daný řetězec kóduje konvergující výpočet daného stroje. Tato kontrola bude primitivně rekurzivní, while cyklus, čili minimalizaci, využijeme k nalezení vhodného řetězce kódujícího výpočet daného TS.

Je-li výpočet tohoto TS M daný konfiguracemi $K_0, K_1, \dots, K_t$, potom kódem tohoto výpočtu bude

$$K_0; K_1; K_2; \dots; K_t.$$

Jde tedy o jednotlivé konfigurace umístěné za sebou a oddělené středníkem. Jak zmíněno, převod tohoto řetězce na binární a posléze na číslo lze najít v kapitole 5.2.1.

Uvědomme si, že běžné operace s řetězci reprezentovanými jim odpovídajícími čísly jsou primitivně rekurzivní.

Lemma 4.5.2 *Funkce pro běžné operace s binárními řetězci jsou primitivně rekurzivní, přičemž uvažujeme, že binární řetězec w_y je předán v parametru svým číslem y. Jde například o určení délky řetězce, přístup k znaku na zadané pozici, porovnání řetězců, výběr podřetězce, nalezení podřetězce, náhrada podřetězce za jiný, konkatenace a další.*

Důkaz: Důkaz pouze naznačíme, odvození všech operací zde provádět nebudeme, čtenář si je může vyzkoušet jako cvičení. Složitost obvyklých řetězcových operací je omezena délkou řetězců a hodnotou dalších parametrů na vstupu. Není tedy nijak překvapivé, že k jejich implementaci není třeba obecného cyklu a vystačíme si s **`for`** cyklem, podmíněným příkazem, voláním podprogramu a základními funkcemi, které jsou primitivně rekurzivní podle lemmatu 4.4.3. Například délku řetězce w_y spočítáme pomocí funkce

$$len(y) = \lceil \log_2 y \rceil.$$

Přístup ke znaku $(0/1)$ řetězce w_y na pozici i můžeme zrealizovat funkcí

$$znak(y,i) = (y \text{ div } 2^{len(y) \dot{-} (i+1)}) \bmod 2,$$

zde je potřeba si uvědomit, že binární řetězce čteme obvykle zleva do prava, tedy znak na pozici 0 řetězce w_y je na druhém nejvýznamnějším bitu čísla y hned za úvodní jedničkou. Pokud bychom raději pracovali přímo s abecedou Γ, můžeme přistupovat ke číslům znaků z ní pomocí funkce

$$znak_\Gamma(y,i) = 4 \cdot znak(y,3i) + 2 \cdot znak(y,3i+1) + znak(y,3i+2).$$

Připojení znaku $b \in \{0,1\}$ k řetězci w_y bychom zrealizovali pomocí

$$pridejznak(y,b) = 2 * y + b.$$

Obecnou konkatenaci řetězců w_a a w_b bychom mohli provést třeba pomocí

$$konkatenace(a,b) = a \cdot 2^{len(b)} + (b \dot{-} 2^{len(b)}).$$

Pomocí těchto funkcí bychom již jednoduše naprogramovali zbylé. □

Nyní již můžeme popsat predikát, který ověří, zda daný řetězec kóduje výpočet daného Turingova stroje.

Věta 4.5.3 *Definujme predikát $T_n(e,x_1,\ldots,x_n,y)$, který je splněn, právě když binární řetězec w_y je kódem výpočtu TS M_e nad řetězcem $(x_1)_B\#(x_2)_B\#\ldots\#(x_n)_B$, který na závěr vydá binárně zapsané číslo. Potom $T_n(e,x_1,\ldots,x_n,y)$ je primitivně rekurzivní predikát.*

Důkaz: Jde opět jen o náznak důkazu. Formální důkaz by byl zbytečně technicky komplikovaný, přičemž by nepřinesl myšlenkově nic zajímavého, proto vynecháme všechny technické detaily a budeme se věnovat pouze myšlence důkazu. Budeme uvažovat pouze případ $n = 1$ a budeme pro jednoduchost používat značení $T(e,x,y) = T_1(e,x_1,y)$, zobecnění pro libovolné $n > 1$ je pouze technickou záležitostí.

Predikát $T(e,x,y)$ si rozepíšeme jako konjunkci predikátů následujícím způsobem:

$$\begin{aligned} T(e,x,y) \quad &= \quad kodTS(e) \wedge \\ &\wedge \quad pocatecni(konfigurace(y,0),e,x) \wedge \\ &\wedge \quad \bigwedge_{i=0}^{pocet(y)-2} nasledujici(konfigurace(y,i),konfigurace(y,i+1),e) \wedge \\ &\wedge \quad koncova(e,konfigurace(y,pocet(y)-1)) \wedge \\ &\wedge \quad cislo(konfigurace(y,pocet(y)-1)) \end{aligned}$$

- Predikát $kodTS(e)$ provede syntaktickou kontrolu toho, jestli w_e je platným kódem TS. Pokud ne, znamená to, že nemá smysl pokračovat v dalších testech. V čem spočívá tento test, jsme popsali již při konstrukci univerzálního TS, zde bychom pouze museli naprogramovat syntaktickou analýzu v jazyku PRF.
- Funkce $konfigurace(y,i)$ vyextrahuje z řetězce w_y kód i-té konfigurace a vrátí jeho číslo, tedy číslo a, pro něž je w_a kódem i-té konfigurace. Pokud tato funkce zjistí, že na daném místě není platný kód konfigurace, tj. narazí na syntaktickou chybu, vrátí číslo prázdného řetězce, tedy 1.
- Predikát $pocatecni(a,e,x)$ otestuje, jestli řetězec w_a kóduje konfiguraci a jestli jde o počáteční konfiguraci stroje M_e při výpočtu nad vstupem $(x)_B$.
- Funkce $pocet(y)$ zjistí, kolik je v w_y zakódováno konfigurací.
- Predikát $nasledujici(a,b,e)$ zjistí, jestli řetězce a a b kódují konfigurace a jestli konfigurace zakódovaná řetězcem w_b následuje po konfiguraci zakódované řetězcem w_a ve výpočtu stroje M_e. Tj. jestli existuje odpovídající instrukce v programu w_e, která z w_a udělá w_b.
- Predikát $koncova(e,a)$ otestuje, jestli w_a kóduje koncovou konfiguraci stroje M_e, musí jít o platnou konfiguraci, z níž už nelze dále podle přechodové funkce stroje M_e pokračovat.
- Predikát $cislo(a)$ zkontroluje, jestli konfigurace kódovaná pomocí w_a má na pásce binární zápis čísla.

Fakt, že tyto funkce a predikáty jsou primitivně rekurzivní, nebudeme dále zkoumat, protože jde o technickou záležitost. Vše plyne z toho, že všechny běžné funkce pro manipulaci s řetězci jsou primitivně rekurzivní dle lemmatu 4.5.2, s těmito funkcemi už dokážeme implementovat výše uvedené primitivně rekurzivní funkce a predikáty. Konečná konjunkce je primitivně rekurzivní dle lemmatu 4.4.15. □

Když jsme schopni ověřit, jestli daný řetězec kóduje výpočet Turingova stroje, můžeme již k zadanému Turingovu stroji najít odpovídající ČRF.

Věta 4.5.4 *Je-li funkce f turingovsky vyčíslitelná, je f ČRF.*

Důkaz: Pro jednoduchost předpokládejme, že f je funkcí jedné proměnné. Je-li f turingovsky vyčíslitelná, pak podle definice 4.1.8 existuje Turingův stroj, který ji počítá. Předpokládejme, že jde o stroj M_e s Gödelovým číslem e. Můžeme tedy psát

$$f(x) = \mathcal{U}(\mu y[T(e,x,y)]),$$

kde $\mathcal{U}$ je funkce, která z y vytáhne poslední konfiguraci a z ní číslo, které tato konfigurace reprezentuje. $\mathcal{U}$ je zřejmě primitivně rekurzivní funkce, technickými detaily se opět nebudeme zabývat. Číslo programu e je konstanta, kterou můžeme do T vložit pomocí substituce. □

Nyní můžeme použít následující úvahu. Mějme ČRF f, protože jde o ČRF, existuje podle věty 4.5.1 TS M, který f počítá. TS M má přiřazeno Gödelovo číslo, označme jej e, které tedy určuje kromě stroje $M = M_e$ i funkci f. Očíslování Turingových strojů můžeme tedy využít i k očíslování ČRF. Toto očíslování ČRF se nám bude hodit víc, než to, které bychom dostali zakódováním odvození ČRF do řetězce a potažmo čísla, protože tímto způsobem číslo odpovídá skutečně algoritmu a je již jedno, jaký jazyk či výpočetní model (v našem případě TS nebo ČRF) bychom použili k jeho implementaci. Vzhledem k tomu, jak jsme definovali funkci, kterou M počítá, je zřejmé, že pro libovolný počet proměnných $n > 0$ počítá TS M nějakou funkci n proměnných, následující definice tedy dává dobrý smysl.

Definice 4.5.5 Pomocí $\varphi_e^{(n)}$, kde $e, n \in \mathbb{N}$, označíme ČRF n proměnných, kterou počítá stroj M_e. Pokud $n = 1$, budeme též psát φ_e. ◂

To, co jsme vlastně dosud ukázali, je Kleeneho věta o normální formě.

Věta 4.5.6 (Kleeneho o normální formě) *Existuje primitivně rekurzivní funkce $\mathcal{U}(y)$ a primitivně rekurzivní predikát $T_n(e, x_1, \ldots, x_n, y)$, pro které platí:*

$$(\forall n \in \mathbb{N})\ (\forall e \in \mathbb{N})\left[\varphi_e^{(n)}(x_1,\ldots,x_n) \simeq \mathcal{U}\Big(\mu y[T_n(e,x_1,\ldots,x_n,y)]\Big)\right]$$

Důkaz: Důkaz už vlastně máme hotový, viz věta 4.5.4. □

Zajímavým důsledkem Kleeneho věty o normální formě je, že libovolnou ČRF můžeme odvodit za pomoci jediného minimalizačního operátoru, tedy jediného kroku, který není primitivně rekurzivní. Dá se to také říci tak, že každý algoritmus lze přepsat za pomoci nejvýš jednoho cyklu **`while`**. Přičemž řada běžných funkcí a algoritmů je již primitivně rekurzivních, takže se obejdeme i bez toho while cyklu, nicméně Kleeneho věta nám dává uniformní pohled na všechny ČRF (a potažmo všechny algoritmy, připouštíme-li Churchovu-Turingovu tezi).

Dalším důsledkem Kleeneho věty o normální formě je fakt, že každý rekurzivně spočetný predikát lze napsat za použití primitivně rekurzivního predikátu a existenčního kvantifikátoru. Přesněji:

Lemma 4.5.7 *Predikát $R \subseteq \mathbb{N}^n$ je rekurzivně spočetný, právě když existuje primitivně rekurzivní predikát $P \subseteq \mathbb{N}^{n+1}$, pro nějž platí, že $R(x_1,\ldots,x_n) \Leftrightarrow (\exists y)[P(x_1,\ldots,x_n,y)]$.*

Důkaz: Předpokládejme nejprve, že R je rekurzivně spočetný predikát. Podle definice to znamená, že existuje ČRF $\varphi_e^{(n)}$ taková, že $R(x_1,\ldots,x_n) \Leftrightarrow \varphi_e^{(n)}(x_1,\ldots,x_n)\downarrow$. Podle Kleeneho věty o normální formě dostaneme, že

$$\chi_R(x_1,\ldots,x_n) \simeq \varphi_e^{(n)} \simeq \mathcal{U}\big(\mu y[T_n(e,x_1,\ldots,x_n,y)]\big).$$

Z toho plyne, že pro každý vstup $x_1,\ldots,x_n$ platí

$$\chi_R(x_1,\ldots,x_n)\downarrow \Leftrightarrow \mu y[T_n(e,x_1,\ldots,x_n,y)]\downarrow .$$

Jelikož T_n je primitivně rekurzivní predikát a je tedy definovaný pro všechny vstupy, znamená to, že minimalizační operátor najde vhodné y právě když nějaké existuje. Jinými slovy

$$\chi_R(x_1,\ldots,x_n)\downarrow \Leftrightarrow (\exists y)[T_n(e,x_1,\ldots,x_n,y)],$$

stačí tedy definovat

$$P(x_1,\ldots,x_n,y) = T_n(e,x_1,\ldots,x_n,y).$$

Nyní předpokládejme, že predikát $R(x_1,\ldots,x_n) = (\exists y)[P(x_1,\ldots,x_n,y)]$, kde P je primitivně rekurzivní predikát. Z toho plyne, že charakteristická funkce χ_P predikátu P je primitivně rekurzivní a funkci f, jejímž definičním oborem je R, můžeme definovat takto

$$f(x_1,\ldots,x_n) = \mu y[\chi_P(x_1,\ldots,x_n,y) \simeq 1],$$

přičemž opět využíváme toho, že v případě, kdy P je PRP, a tedy χ_P je PRF, je druhá podmínka minimalizačního operátoru automaticky splněna, a tak zde minimalizace odpovídá existenčnímu kvantifikátoru. □

Rozdíl mezi rekurzivním a rekurzivně spočetným predikátem je tedy možné zformulovat i takto: Rekurzivní predikáty jsou ty, které jsou algoritmicky rozhodnutelné, rekurzivně spočetné predikáty jsou ty, které jsou algoritmicky ověřitelné, podá-li nám někdo certifikát (tedy y) dokazující jejich platnost. Tedy u rekurzivně spočetného predikátu nejsme sice obecně schopni efektivně rozhodnout, jestli platí, ale pokud platí a někdo nám dá svědka y stvrzující tento fakt, jsme schopni efektivně ověřit, že jde skutečně o certifikát platnosti. Jak vidíme, mezi tím, co jsme schopni efektivně rozhodnout a ověřit je rozdíl, protože například predikát problému zastavení je rekurzivně spočetný, ale není rekurzivní. V části o složitosti zjistíme, že nahradíme-li slůvko *efektivní* souslovím *v polynomiálním čase*, není tento rozdíl tak snadno vidět a nikdo jej zatím neumí dokázat. Nejen to, ale pokud místo *efektivní* použijeme *v polynomiálním prostoru*, pak mezi ověřením a rozhodnutím (v polynomiálním prostoru) rozdíl není.

Jako další důsledek dostaneme existenci univerzální ČRF.

Věta 4.5.8 (O univerzální funkci) *Pro každé přirozené číslo $n > 0$ existuje ČRF $n+1$ proměnných $\varphi_z^{(n+1)}(e,x_1,\ldots,x_n)$ taková, že $\varphi_z^{(n+1)}(e,x_1,\ldots,x_n) = \varphi_e^{(n)}(x_1,\ldots,x_n)$.*

Důkaz: Podle Kleeneho věty o normální formě stačí položit

$$\varphi_z^{(n+1)}(e, x_1, \ldots, x_n) \simeq \mathcal{U}\Big(\mu y[T_n(e, x_1, \ldots, x_n, y)]\Big).$$

Nicméně vzhledem k ekvivalenci ČRF a TS bychom také mohli vzít již zkonstruovaný univerzální TS a jemu odpovídající ČRF, s případnou úpravou vstupu do vhodného tvaru. □

Jako tomu bylo u Turingových strojů, i univerzální funkce pro třídu ČRF je sama ČRF. V komentáři za definicí PRF jsme již zdůvodnili, že to neplatí pro třídu PRF, protože funkce univerzální pro třídu PRF sama nemůže být PRF, ale protože by šlo o funkci všude definovanou, byla by funkce univerzální pro třídu PRF obecně rekurzivní. Všimněme si dalšího rozdílu, v případě primitivně rekurzivních funkcí můžeme zakódovat jejich odvození a přiřadit jim přirozená čísla, protože jde o odvození bez použití minimalizace. Pokud však použijeme minimalizaci, pak otázka, zda odvozená funkce je obecně nebo primitivně rekurzivní, je algoritmicky nerozhodnutelná. Například rozhodnutí, zda daná ČRF f je obecně rekurzivní, tedy totální, odpovídá otázce, zda se daný TS počítající M_f, zastaví na všech vstupech, což je na první pohled složitější, než jenom zodpovědět, zda se zastaví na daném vstupu. Už to je přitom nerozhodnutelné, neboť se jedná o problém zastavení. Nerozhodnutelnost problému, kde se ptáme, zda funkce počítaná daným Turingovým strojem je primitivně rekurzivní plyne například z Riceovy věty 7.4.2, k níž se dostaneme později. Z toho plyne, že uvažovat cosi jako univerzální funkci pro třídu ORF nemá moc smysl, protože nejsme ani schopni poznat, zda dané odvození odvodí obecně rekurzivní funkci.

Poznámka 4.5.9 *Univerzální ČRF budeme využívat poměrně často, přestože ji nebudeme zmiňovat přímo. Půjde o situace, kdy jako Gödelovo číslo funkce (tj. zdrojový kód odpovídajícího programu) použijeme parametr. Například definujme funkci $\varphi_e(x, y, u) \simeq \varphi_x(u) + \varphi_y(u)$, formálně při této definici používáme univerzální funkci, která nám umožní zavolat x-tou a y-tou funkci, správně bychom tedy měli psát $\varphi_e(x, y, u) \simeq \varphi_z^{(2)}(x, u) + \varphi_z^{(2)}(y, u)$. My však budeme používat prvního zápisu, protože je to jednodušší, ale budeme mít na paměti, že nám tento zápis umožňuje právě existence univerzální funkce.*

Následující věta ukazuje, že je možné efektivně provést částečné dosazení do funkce a že kód nově vzniklé funkce lze efektivně spočítat.

Věta 4.5.10 (s-m-n) *Pro každá dvě přirozená čísla $m, n \geq 1$ existuje prostá PRF s_n^m, jež je funkcí $m + 1$ proměnných a pro všechna $x, y_1, y_2, \ldots, y_m$ platí:*

$$\varphi^{(n)}_{s_n^m(x, y_1, y_2, \ldots, y_m)} = \lambda z_1 z_2 \ldots z_n [\varphi_x^{(m+n)}(y_1, \ldots, y_m, z_1, \ldots, z_n)]$$

Důkaz: Neformálně popíšeme, co bude dělat program $s = s_n^m(x, y_1, y_2, \ldots, y_m)$. Na vstupu dostane čísla $z_1, \ldots, z_n$, poté spustí stroj M_x na vstup $y_1, y_2, \ldots, y_m, z_1, z_2, \ldots, z_n$, uvědomme si, že všechna tato čísla zná, jelikož je buď dostane na vstupu M_s (v případě $z_1, \ldots, z_n$), nebo je má zakódované do své přechodové funkce jako parametry (v případě $x, y_1, \ldots, y_m$). Stroj M_s tedy prostě napíše před parametry $z_1, \ldots, z_n$ parametry $y_1, \ldots, y_m$

a spustí M_x. Protože tento program jsme schopni popsat efektivně, jsme schopni popsat i TS, který pro $x, y_1, \ldots, y_m$ spočítá program $s = s_n^m(x, y_1, \ldots, y_m)$. Přesněji, výpočet $s_n^m(x, y_1, \ldots, y_m)$ spočívá v tom, že k instrukcím stroje M_x se přidají instrukce, které připíší $y_1, \ldots, y_m$ před vstup, tj. instrukce, které budou hýbat hlavou vlevo a postupně psát $y_m, \ldots, y_1$ zprava do leva, na konci tohoto zápisu bude instrukce, která přejde do počátečního stavu vlastního stroje M_x. Všimněme si, že pro tuto úpravu není vůbec rozhodující, jak vypadají instrukce M_x, ani to, jak bude probíhat jeho výpočet, jediné, co je v M_x třeba změnit jsou čísla stavů tak, aby nekolidovala s nově přidanými stavy. Úprava přechodové funkce M_x je tedy opravdu jednoduchá a vystačí si jistě s primitivně rekurzivními prostředky. Proto je i funkce s_n^m primitivně rekurzivní.

To, že lze funkci s_n^m implementovat tak, aby byla prostá, je snadno vidět, kód nového stroje totiž obsahuje nějakým způsobem i hodnoty parametrů, které funkce s_n^m dostane, pro různé hodnoty těchto parametrů budou i různé výstupní hodnoty. □

S-m-n věta opět není ničím překvapivým ani složitým, jde spíš o technickou záležitost, kterou však budeme často používat při manipulaci s částečně rekurzivními funkcemi. Už při přečtení znění s-m-n věty si dokážeme představit algoritmus v intuitivním smyslu, který počítá funkci s_n^m, podle Churchovy-Turingovy teze jsme tento algoritmus schopni implementovat na Turingově stroji a potažmo pomocí částečně rekurzivní funkce. Protože v něm nepotřebujeme cyklus *while*, stačí nám dokonce primitivně rekurzivní funkce. Tato věta navíc ukazuje velmi užitečný princip částečného dosazení, který je zvlášť ve funkcionálním programování velmi obvyklý.

Příklad 4.5.11

Uvažme funkci mocniny $pow(x, y) \simeq y^x$ s jejíž pomocí chceme odvodit funkci druhé mocniny $square(y) \simeq y^2$. Máme-li již funkci *pow* a její Gödelovo číslo e (tj. její zdrojový kód), tak můžeme psát $square(y) \simeq pow(2, y) \simeq \varphi_e^{(2)}(2, y)$. Aniž bychom se zajímali o to, jak vypadá zdrojový kód e, můžeme jej předhodit funkci s_1^1 a nechat si spočítat zdrojový kód funkce $square(y)$, protože $square(y) \simeq \varphi_e^{(2)}(2, y) \simeq \varphi_{s_1^1(e,2)}(y)$. Týmž způsobem bychom mohli odvodit $cube(y) \simeq \varphi_{s_1^1(e,3)}(y)$ a podobně i pro další libovolnou hodnotu mocniny k. V tomto případě je tedy funkce $f(k) \simeq s_1^1(e, k)$ PRF, která pro dané k spočítá zdrojový kód funkce y^k. Podle s-m-n věty totiž platí:

$$y^k \simeq pow(k, y) \simeq \varphi_e(k, y) \simeq \varphi_{s_1^1(e,k)}(y) \simeq \varphi_{f(k)}(y)$$

Zde můžeme poznamenat, že s-m-n věta je jakýmsi opakem věty o univerzální funkci, uvažme v předchozím příkladu funkci

$$square(y) \simeq \varphi_e^{(2)}(2, y) \simeq \varphi_{s_1^1(e,2)}(y).$$

Je-li $\varphi_z^{(2)}$ univerzální ČRF pro funkce jedné proměnné, pak můžeme pokračovat a psát

$$\varphi_{s_1^1(e,2)}(y) \simeq \varphi_z^{(2)}(s_1^1(e,2),y).$$

Nebo obecně

$$\varphi_e^{(2)}(x,y) \simeq \varphi_{s_1^1(e,x)}(y) \simeq \varphi_z^{(2)}(s_1^1(e,x),y).$$

Všimněme si, že na obou stranách máme funkci dvou parametrů x a y, na každé straně máme však jiný program, který tuto funkci počítá. Zatímco na levé straně běží přímo program e, na druhé straně běží program e interpretovaný univerzální funkcí (asi jako by běžel na virtuálním stroji).

Příklad 4.5.12

Uvažme funkci $\varphi_e(x,y,u) \simeq \varphi_x(u) + \varphi_y(u)$, kterou jsme již použili v poznámce 4.5.9 k univerzální ČRF. Podle s-m-n věty dostaneme, že

$$\varphi_e(x,y,u) \simeq \varphi_{s_1^2(e,x,y)}(u).$$

Definujeme-li $f(x,y) \simeq s_1^2(e,x,y)$, pak funkce f je prostou primitivně rekurzivní funkcí, pro kterou platí, že

$$\varphi_{f(x,y)}(u) \simeq \varphi_x(u) + \varphi_y(u).$$

Je-li například e Gödelovo číslo funkce I_1^1, pak $f(e,e)$ určuje Gödelovo číslo funkce násobení dvěma, neboť

$$\varphi_{f(e,e)}(u) \simeq \varphi_e(u) + \varphi_e(u) \simeq I_1^1(u) + I_1^1(u) = u + u = 2u.$$

Poznamenejme, že to, které parametry si vybereme k zafixování a předložení funkci s_n^m není příliš podstatné, pořadí parametrů si totiž můžeme vždy změnit s pomocí substituce a projekce.

4.6. Bibliografické poznámky

4.7. Cvičení

1. Ukažte větu 4.1.9.

5. Algoritmy

V této kapitole nejprve vyslovíme Churchovu-Turingovu tezi, která spojuje intuitivní pojem algoritmu s matematickým pojmem Turingova stroje. Zavedeme také číslování Turingových strojů a popíšeme univerzální Turingův stroj. Ten se nám bude dále hodit v našich úvahách o algoritmicky řešitelných problémech a algoritmicky vyčíslitelných funkcí, což jsou pojmy, které zavedeme v závěru kapitoly.

5.1. Churchova-Turingova teze

Pod pojmem algoritmu si obvykle intuitivně představujeme konečnou posloupnost jednoduchých instrukcí (příkazů, pokynů), která vede k řešení zadané úlohy. V tomto smyslu se dá za algoritmus považovat vlastně jakýkoli postup či návod, od Euklidova algoritmu přes popis cesty ke kamarádovi domů po návod na použití fotoaparátu či kuchařský recept. Kromě posloupnosti instrukcí potřebujeme však pochopitelně i prostředek, na kterém budeme tuto posloupnost vykonávat, ať už je to naše hlava, nohy, fotoaparát a ruce či my a kuchyňské vybavení.

Nás budou zajímat algoritmy, které pracují s matematickými objekty (čísly, řetězci, funkcemi apod.). Chceme-li formalizovat pojem algoritmu v matematice, potřebujeme model výpočetního prostředku, který by jednak byl dostatečně obecný, aby obsáhl naši intuitivní představu algoritmu, a jednak dostatečně jednoduchý, aby se s ním dobře manipulovalo. Dosud jsme si popsali tři takové modely — Turingovy stroje, RAM a částečně rekurzivní funkce. Ukázali jsme si také, že všechny tři tyto modely jsou stejně silné, je tedy možno v nich implementovat výpočet týchž funkcí, přijímání téže množiny jazyků. Potažmo je v těchto modelech tedy možno vyřešit touž třídu problémů a úloh. Kromě toho však existuje i celá řada dalších stejně silných výpočetních modelů — programy ve vyšších programovacích jazycích jako je C, Pascal, Basic, Java (i když zde je třeba mít vždy na paměti i počítač, na kterém jsou tyto programy interpretovány), programy ve funkcionálních jazycích jako je λ-kalkulus (který pochází od Alonza Churche, 1936 a který tvoří teoretický základ funkcionálních jazyků), Haskell, Lisp, mezi dalšími můžeme zmínit třeba skript pro sed.

Kterýkoli z těchto prostředků bychom si mohli vybrat a vybudovat tutéž teorii, protože všechny jsou co do výpočetní síly ekvivalentní. V roce 1936 právě Church, Turing a Kleene ukázali, že Turingovy stroje a λ-kalkulus jsou stejně silné prostředky jako o něco starší částečně rekurzivní funkce. Zformulovali současně tezi, které se říká Churchova-Turingova, a podle níž právě Turingovy stroje a jim ekvivalentní prostředky zachycují intuitivní pojem algoritmu.

Teze 5.1.1: Churchova-Turingova (1936)

Ke každému algoritmu v intuitivním smyslu existuje Turingův stroj, který jej implementuje.

Churchova-Turingova teze se snaží spojit intuitivní představu pojmu algoritmu s přesnou matematickou definicí Turingova stroje. Spojuje tak dva velmi rozdílné světy a ze své podstaty nejde o matematické tvrzení, jež by bylo možno dokázat, tezi pouze přijmout nebo odmítnout. My tuto tezi pro další výklad přijmeme, neboť je podpořena řadou ekvivalentních modelů a dá se v tomto smyslu říci, že jako reálný výpočetní prostředek nic lepšího než Turingovy stroje nemáme.

Churchovu-Turingovu tezi budeme dále často využívat v obou směrech. Často budeme popisovat algoritmy v intuitivním smyslu, přičemž na základě Churchovy-Turingovy teze budeme předpokládat, že bychom tento algoritmus mohli implementovat na Turingovu stroji (RAMu, ČRF, ...). Na druhou stranu ve chvílích, kdy budeme potřebovat říci něco skutečně formálně, budeme hovořit o Turingových strojích, při tom ovšem budeme mít na paměti, že současně můžeme dané výsledky vztáhnout i na jiné výpočetní modely, potažmo na algoritmy obecně.

5.2. Kódování objektů a univerzální Turingův stroj

Abychom mohli k Turingovům strojům (a tedy i k algoritmům) přistupovat uniformním způsobem bude se nám hodit to, že každý Turingův stroj lze zakódovat pomocí řetězce a přiřadit mu přirozené číslo. Toto kódování se nám bude hodit i při konstrukci univerzálního Turingova stroje, který se nám bude též při dalších úvahách o Turingových strojích velmi hodit. Například si tímto způsobem můžeme zodpovědět otázku, kolik je vlastně Turingových strojů.

5.2.1. Kódování Turingových strojů a Gödelovo číslo

V této kapitole zavedeme číslování Turingových strojů. Je třeba předeslat, že způsob kódování Turingových strojů a způsob přiřazení čísla, které zde popíšeme, je jen jedním z mnoha možných. Konkrétní způsob kódování není pro další úvahy o algoritmech a Turingových strojích nijak podstatný. Potřebujeme si nějaký způsob zvolit, abychom jej mohli dále využívat, ale jakýkoli jiný způsob kódování s podobnými vlastnostmi by bylo možno využít stejně dobře. Podstatnou vlastností kódu, který popíšeme, je jeho efektivita. To znamená, že kód je možno dobře zpracovávat na Turingovu stroji (potažmo na jiném výpočetním modelu).

Naším dalším cílem je také každému Turingovu stroji přiřadit jeho Gödelovo číslo. Postup tohoto přiřazení je rozložen do tří kroků.

1. Turingův stroj nejprve reprezentujeme řetězcem v abecedě

$$\Gamma = \{\texttt{0},\texttt{1},\texttt{L},\texttt{N},\texttt{R},\texttt{|},\texttt{\#},\texttt{;}\}. \tag{5.1}$$

2. Poté každý znak abecedy Γ přepíšeme pomocí tří bitů, tedy znaků `0` a `1`.

3. Nakonec využijeme toho, že binární řetězce lze očíslovat přirozenými čísly, jak jsme si ukázali v kapitole 3.4.4.

Omezení kladená na kódované Turingovy stroje

Pro účely popisu kódování budeme uvažovat jen Turingovy stroje splňující jisté omezující předpoklady, jež jim však neubírají na výpočetní síle. Přesněji, budeme uvažovat, že Turingův stroj $M = (Q, \Sigma, \delta, q_0, F)$, který kódujeme, splňuje následující omezení:

(i) Vstupní abeceda M je binární abeceda $\{\texttt{0}, \texttt{1}\}$. To znamená, že stroji M jsou na vstupu předávány pouze binární řetězce.

(ii) $F = \{q_1\}$, tedy M má jediný přijímající stav, který označíme pomocí $q_1 \in Q$.

Zdůrazněme ihned, že páskovou abecedu stroje M nijak neomezujeme, jediné, co vyžadujeme, aby spolu se symboly `0` a `1` vstupní abecedy obsahovala i symbol λ prázdného políčka.

Vstupní abecedu M omezujeme proto, že vstupní abeceda univerzálního Turingova stroje (který budeme dále konstruovat) musí obsahovat symboly vstupní abecedy simulovaného stroje. Mohli bychom zvolit libovolně velkou abecedu. Binární abeceda se pro naše účely hodí proto, že je sice dostatečně malá, aby se s ní dobře pracovalo, ale je současně dostatečně velká, abychom pomocí ní mohli zakódovat řetězec v libovolné abecedě (jak uvidíme dále). Mohli bychom ve skutečnosti uvažovat i jednoprvkovou vstupní abecedu, ale pak by popis kódování byl zbytečně komplikovaný, protože všechno bychom museli rovnou zakódovat do jediného přirozeného čísla, což učiníme až ve třetím kroku. Na druhou stranu tříprvková abeceda nepřináší proti binární nic podstatného.

Předpoklad, že kódovaný stroj má pouze jeden přijímající stav, nemá na výpočetní sílu Turingova stroje žádný vliv. Libovolný TS M lze totiž triviálně převést na ekvivalentní TS M', který má jediný přijímající stav, z nějž nevedou již žádné instrukce. Pokud má totiž M více přijímajících stavů, nebo z jeho přijímajících stavů vedou nějaké instrukce, přidáme k M nový stav q_1 a instrukce, které ve chvíli ukončeného výpočtu v některém z přijímajících stavů zabezpečí přechod do stavu q_1. Pokud naopak TS M nemá žádný přijímající stav, pak ničemu nevadí, pokud k němu přidáme nový stav, který bude přijímající, ale nebude do něj možné přejít pomocí žádné instrukce.

Kódování Turingových strojů řetězci v abecedě Γ

V této podkapitole popíšeme kódování Turingových strojů pomocí řetězců v abecedě $\Gamma = \{\texttt{0}, \texttt{1}, \texttt{L}, \texttt{N}, \texttt{R}, \texttt{|}, \texttt{\#}, \texttt{;}\}$. Za podmínek, které klademe na Turingův stroj, stačí k jeho popisu vhodným způsobem zakódovat přechodovou funkci, neboť veškeré další informace o stroji lze již ze zápisu přechodové funkce vyčíst. Dosud jsme uvažovali přechodovou

funkci zapsanou v tabulce. Kód Turingova stroje dostaneme prostě tak, že zapíšeme jednotlivé řádky tabulky přechodové funkce za sebe. Pro zápis jedné instrukce, tedy jednoho řádku tabulky, využijeme první šestici znaků z abecedy Γ, znak `#` posléze použijeme k oddělení jednotlivých instrukcí. Znak `;` nebude v kódu použit a je možné jej využít například k oddělení kódu stroje od jeho vstupu na vstupní pásce univerzálního TS.

Uvažme tedy TS $M = (Q, \Sigma, \delta, q_0, F)$, kde $F = \{q_1\}$. Předpokládáme, že stavy a symboly páskové abecedy jsou očíslované, tedy $Q = \{q_0, q_1, \ldots, q_r\}$ pro nějaké $r \geq 1$ a podobně $\Sigma = \{X_0, X_1, X_2, \ldots, X_s\}$ pro nějaké $s \geq 2$. Dále předpokládáme, že q_0 označuje počáteční stav a q_1 jediný přijímající stav. A konečně předpokládáme, že X_0 označuje symbol `0`, X_1 označuje symbol `1` a X_2 označuje symbol prázdného políčka λ. Pokud $s > 2$, pak $X_3, \ldots, X_s$ označují další symboly páskové abecedy stroje M. Instrukci $\delta(q_i, X_j) = (q_k, X_l, Z)$, kde $Z \in \{L, N, R\}$ označuje pohyb hlavy, zakódujeme řetězcem:

$$(i)_B \,|\, (j)_B \,|\, (k)_B \,|\, (l)_B \,|\, \texttt{Z} \tag{5.2}$$

Pomocí $(n)_B$ označujeme binární zápis čísla n. V kódu Turingova stroje přitom připouštíme libovolný počet nul na začátku tohoto zápisu. Nechť C_1 až C_n označují kódy instrukcí M v abecedě Γ, potom kód stroje M vznikne jejich konkatenací s použitím oddělovače `#`. Tedy kódem M v abecedě Γ je řetězec:

$$C_1 \# C_2 \# \ldots \# C_{n-1} \# C_n$$

Příklad 5.2.1

Uvažme například Turingův stroj M, který přijímá jazyk palindromů PAL a který jsme zkonstruovali v příkladu 4.1.4. Pro tento stroj platí, že q_1 je jediným přijímajícím stavem. Vstupní abeceda M je dvouprvková, byť symboly použité v příkladu 4.1.4 jsou `a` a `b`. My místo toho využijeme `0` (místo `a`) a `1` (místo `b`). To znamená, že budeme uvažovat stroj, který přijímá jazyk palindromů nad abecedou $\{\texttt{0}, \texttt{1}\}$.

$$\mathrm{PAL}_b = \left\{ w = w^R \mid w \in \{\texttt{0}, \texttt{1}\}^* \right\}, \tag{5.3}$$

Kódování jednotlivých řádků tabulky přechodové funkce je uvedeno v tabulce 5.1.

Tabulka 5.1.: Kódy řádků přechodové tabulky stroje M.

	q, c	$\to$	q', c', Z	Kód instrukce
1.	q_0, λ	$\to$	q_1, λ, N	`0|10|1|10|N`
2.	$q_0, \texttt{0}$	$\to$	$q_2, \texttt{0}, R$	`0|0|10|0|R`
3.	$q_0, \texttt{1}$	$\to$	$q_3, \texttt{0}, R$	`0|1|11|0|R`
4.	$q_2, \texttt{0}$	$\to$	$q_2, \texttt{0}, R$	`10|0|10|0|R`
5.	$q_2, \texttt{1}$	$\to$	$q_2, \texttt{1}, R$	`10|1|10|1|R`

6.	q_2,λ	$\rightarrow$	q_3,λ,L	`10|10|11|10|L`
7.	$q_3,0$	$\rightarrow$	$q_3,0,R$	`11|0|11|0|R`
8.	$q_3,1$	$\rightarrow$	$q_3,1,R$	`11|1|11|1|R`
9.	q_3,λ	$\rightarrow$	q_5,λ,L	`11|10|101|10|L`
10.	$q_4,0$	$\rightarrow$	q_6,λ,L	`100|0|110|10|L`
11.	$q_5,1$	$\rightarrow$	q_6,λ,L	`101|1|110|10|L`
12.	$q_6,0$	$\rightarrow$	$q_6,0,L$	`110|0|110|0|L`
13.	$q_6,1$	$\rightarrow$	$q_6,1,L$	`110|1|110|1|L`
14.	q_6,λ	$\rightarrow$	q_7,λ,R	`110|10|111|10|R`
15.	$q_7,0$	$\rightarrow$	q_0,λ,R	`111|0|0|10|R`
16.	$q_7,1$	$\rightarrow$	q_0,λ,R	`111|1|0|10|R`
17.	q_7,λ	$\rightarrow$	q_1,λ,N	`111|10|1|10|N`

Spojením kódů jednotlivých řádků s použitím oddělovacího znaku `#` dostaneme kód Turingova stroje M:

```
0|10|1|10|N#0|0|10|0|R#0|1|11|0|R#10|0|10|0|R#10|1|10|1|R#
10|10|11|10|L#11|0|11|0|R#11|1|11|1|R#11|10|101|10|L#
100|0|110|10|L#101|1|110|10|L#110|0|110|0|L#110|1|110|1|L#
110|10|111|10|R#111|0|0|10|R#111|1|0|10|R#111|10|1|10|N
```

Převod řetězce v abecedě Γ do binárního řetězce

Dalším krokem kódování Turingova stroje bude převod řetězce v abecedě Γ do binárního řetězce. Každý znak abecedy Γ zapíšeme v binární abecedě pomocí tří bitů podle tabulky 5.2.

Γ	kód
0	000
1	001
L	010
N	011

Γ	kód
R	100
|	101
#	110
;	111

Tabulka 5.2.: Převod znaků abecedy Γ do binární abecedy $\{0,1\}$.

Náhradou znaků v řetězci v abecedě Γ dostaneme jeho přepis pomocí binárního řetězce.

Příklad 5.2.2

Navažme na příklad 5.2.1, v němž jsme sestrojili řetězec v abecedě Γ, který kóduje Turingův stroj M rozhodující jazyk palindromů PAL_b. Přepisovat zde celý kód do binárního řetězce by asi nemělo příliš smysl, ale podívejme se alespoň na jednu instrukci, například

$$q_3, \lambda \to q_5, \lambda, L.$$

Této instrukci odpovídá kód v abecedě Γ

$$\texttt{11|10|101|10|L},$$

který je podle tabulky 5.2 převeden do binárního řetězce následujícím způsobem (na horním řádku je řetězec v abecedě Γ, na spodním řádku je řetězec v binární abecedě, který mu odpovídá, „zlomky“ naznačují, jaký znak daná trojice bitů kóduje):

1	1	|	1	0	|	1	0	1	|	1	0	|	L
001	001	101	001	000	101	001	000	001	101	001	000	101	010

Všimněme si, že na pořadí, v jakém očíslujeme stavy a znaky páskové abecedy či v jakém zapíšeme instrukce, nijak nezáleží. Z toho plyne, že každý TS M může mít mnoho různých kódů, které jsou zcela ekvivalentní. Ve skutečnosti pokud si uvědomíme, že stavům (kromě q_0 a q_1) či symbolům páskové abecedy (kromě $0, 1, \lambda$) můžeme přiřadit libovolná přirozená čísla a ne jen čísla z množiny $\{0, 1, \ldots, |Q| - 1\}$ v případě stavů či $\{0, 1, 2, \ldots, |\Sigma| - 1\}$ v případě páskové abecedy, získáme dokonce nekonečně mnoho řetězců, které kódují týž TS. To je naprosto v pořádku, podobně když v programu v jazyce *C* použijeme různé názvy proměnných či funkcí, dostaneme různé zdrojové kódy, třebaže na programu samotném jsme nic nezměnili.

Zřejmě ne každý binární řetězec kóduje nějaký Turingův stroj (počet bitů v syntakticky správném řetězci musí například být dělitelný třemi). Abychom se vyhnuli technickým obtížím s tím spojeným, přijmeme následující úmluvu.

Úmluva 5.2.3 *Binární řetězce, které nejsou syntakticky správnými kódy Turingova stroje, reprezentují Turingův stroj s prázdnou přechodovou funkcí.*

Turingův stroj s prázdnou přechodovou funkcí nutně zamítne každý vstup hned v prvním kroku, neboť počáteční stav je různý od přijímajícího. Poznat, zda daný řetězec je syntakticky správným kódem Turingova stroje, lze přitom celkem snadno, neboť tvar kódování je dobře popsaný.

Pro označení řetězce kódujícího Turingův stroj si zavedeme následující značení.

Definice 5.2.4 Nechť M je Turingův stroj, pomocí $\langle M \rangle$ označíme binární řetězec, který kóduje M. ◂

Vzhledem k tomu, že kód Turingova stroje není jednoznačný, označuje $\langle M \rangle$ nějaký jeho kód, zkonstruovaný výše uvedeným způsobem pro konkrétní očíslování stavů, znaků páskové abecedy, pořadí řádků v přechodové tabulce, atd.

Gödelovo číslo Turingova stroje

V kapitole 3.4.4 jsme zavedli číslování řetězců. Toho můžeme nyní využít k tomu, abychom přiřadili čísla Turingovým strojům.

Definice 5.2.5 (Gödelovo číslo Turingova stroje) Nechť M je Turingův stroj a $\langle M \rangle$ je jeho kód, pak index $\langle M \rangle$ je Gödelovým číslem Turingova stroje M. ◂

Vzhledem k tomu, že kód Turingova stroje M není jednoznačný, ani Gödelovo číslo není jednoznačné a dokonce M má nekonečně mnoho Gödelových čísel.

Protože jsme přiřadili prázdný stroj i řetězcům, které nejsou syntakticky správným kódem žádného TS, dostali jsme tak očíslování všech Turingových strojů a naopak každému přirozenému číslu jsme přiřadili nějaký Turingův stroj (těm číslům, jimž odpovídající řetězec neodpovídá syntakticky správnému kódu TS, jsme přiřadili prázdný TS). Jako důsledek očíslování Turingových strojů dostáváme jejich podstatnou vlastnost.

Důsledek 5.2.6 *Všech Turingových strojů je spočetně mnoho, neboť jsme schopni je očíslovat přirozenými čísly. Na základě teze 5.1.1 můžeme tedy říci, že i algoritmů je spočetně mnoho.*

5.2.2. Kódování dalších objektů

Podobně jako jsme popsali kódování Turingových strojů pomocí binárních řetězců, lze takto kódovat i další objekty, například čísla, RAM, logickou formuli, grafy apod. A navíc lze takto kódovat i n-tice těchto objektů. Pro nás nebude dále konkrétní způsob kódování podstatný. Podstatné jen je, aby zvolené kódování bylo vždy efektivní, to znamená, že je s ním možno algoritmicky pracovat, tedy že je s ním schopen Turingův stroj pracovat. To znamená, že je možné vždy z kódu efektivně (ve smyslu algoritmicky) získat všechny potřebné informace o daném objektu. Na základě těchto úvah si zavedeme následující značení.

Definice 5.2.7 Pomocí $\langle X \rangle$ budeme označovat binární řetězec kódující objekt X. Pomocí $\langle X_1, \ldots, X_n \rangle$ označíme binární řetězec kódující n-tici objektů $X_1, \ldots, X_n$. ◂

Například pomocí $\langle M \rangle$ označujeme kód Turingova stroje M (takové kódování jsme zavedli v kapitole 5.2.1), pomocí $\langle M, x \rangle$ budeme označovat kód dvojice, kde M je Turingův stroj a x je řetězec (ne nutně binární).

5.2.3. Univerzální Turingův stroj

V této kapitole si popíšeme univerzální Turingův stroj, který si označíme jako $\mathcal{U}$. Vstupem univerzálního Turingova stroje je řetězec $\langle M, x \rangle$, kde M je Turingův stroj a x je binární řetězec. Univerzální Turingův stroj simuluje práci Turingova stroje M nad vstupem x. Výsledek této simulace je určen výsledkem výpočtu M nad x. Přesněji řečeno, $\mathcal{U}(\langle M, x \rangle)\downarrow$, právě když $M(x)\downarrow$ a $\mathcal{U}(\langle M, x \rangle)$ přijme, právě když $M(x)$ přijme. Po ukončení výpočtu $\mathcal{U}(\langle M, x \rangle)$ navíc jeho páska v nějaké podobě reprezentuje obsah pásky po ukončení výpočtu $M(x)$.

Ve výsledku tedy chceme, aby univerzální Turingův stroj $\mathcal{U}$ přijímal **univerzální jazyk** definovaný následujícím způsobem:

$$L_u = \{\langle M, x\rangle \mid x \in L(M)\} \tag{5.4}$$

Současně ovšem univerzální Turingův stroj bude implementovat univerzální funkci.

Je dobré si uvědomit, že univerzální jazyk $L_u = L(\mathcal{U}) = \{\langle M, x\rangle \mid x \in L(M)\}$ je formalizací problému Přijetí vstupu, který definujeme následujícím způsobem.

Problém 5.2.8: Přijetí vstupu

Instance: Turingův stroj M a řetězec x.

Otázka: Je výpočet Turingova stroje M se vstupem x přijímající?

Univerzální Turingův stroj popíšeme jako třípáskový, neboť je to technicky jednodušší, než popisovat jednopáskový univerzální Turingův stroj. Na základě věty 4.1.12 víme, že třípáskový stroj lze vždy převést na jednopáskový. Tak dostaneme jednopáskový univerzální Turingův stroj, který je potom v případě potřeby schopen simulovat i sám sebe.

Jako pracovní abecedu $\mathcal{U}$ použijeme abecedu Γ definovanou v (5.1). Pro zjednodušení popisu budeme navíc přistupovat ke kódu $\langle M\rangle$ (tedy první složce dvojice $\langle M, x\rangle$) jako k řetězci v abecedě Γ, i když je třeba mít na paměti, že přečtení jednoho znaku z abecedy Γ potom znamená přečtení tří po sobě jdoucích bitů v binárním řetězci $\langle M\rangle$.

1. páska obsahuje vstup $\mathcal{U}$, tedy kód $\langle M, x\rangle$.

$\langle M, x\rangle$

Na **2. pásce** je uložen obsah pracovní pásky M. Symboly X_i jsou zapsány jako $(i)_B$ v blocích téže délky oddělených |.

...|010|001|100|000|010|011|...

3. páska obsahuje číslo aktuálního stavu q_i stroje M.

10011 (= $(i)_B$)

Obrázek 5.1.: Rozdělení funkcí pásek univerzálního Turingova stroje.

Rozdělení funkcí pásek UTS je následující (viz též obrázek 5.1):

1. **Vstupní páska.** Na vstupní pásce je na počátku uveden vstup tvořený kódem dvojice $\langle M, x\rangle$, kde M je simulovaný stroj a x je jeho vstup. Pro jednoduchost můžeme

předpokládat, že na pásce je nejprve zapsán kód $\langle M \rangle$ následovaný oddělovačem, za nímž následuje přímo zapsaný řetězec x. Jako oddělovač můžeme použít například znak „;“ z abecedy Γ zapsaný trojici bitů `111`. Podstatné je jen to, že $\mathcal{U}$ může číst přímo jak kód $\langle M \rangle$, tak jeho vstup x. Za tohoto předpokladu můžeme předpokládat, že vstupní páska je jen pro čtení a $\mathcal{U}$ ji během výpočtu neupravuje.

2. **Pracovní páska** M**.** Tato páska je v průběhu výpočtu využita k reprezentaci obsahu (jediné) pásky M. Připomeňme si, že páskovou abecedu stroje M jsme nijak neomezovali, její znaky jsou proto kódovány na této pásce týmž způsobem, jakým jsou zapsány v kódu $\langle M \rangle$, tedy znak X_j je zapsán jako $(j)_B$. Navíc jsou však kódy znaků zleva zarovnány nulami tak, aby všechny bloky reprezentující znaky páskové abecedy M měly touž délku, přičemž jednotlivé bloky jsou mezi sebou odděleny znakem `|`. Polohu hlavy stroje M si UTS pamatuje polohou hlavy na této pracovní pásce.

3. **Stavová páska** M**.** Na této pásce je uložen stav, v němž se aktuálně stroj M nachází. Stav q_i s číslem i je zakódován jako $(i)_B$, tedy jako číslo i zapsané binárně. Jde o totéž číslo, které lze vyčíst z přechodové funkce zakódované v $\langle M \rangle$.

Postup výpočtu $\mathcal{U}(\langle M, x \rangle)$ je naznačen v algoritmu 5.2.1. Některé kroky tohoto algoritmu si nyní popíšeme podrobněji.

Krok 1 Zde se kontroluje, zda má $\langle M \rangle$ správný tvar popsaný v kapitole 5.2.1. To znamená, že se skládá ze správně zapsaných kódů jednotlivých instrukcí oddělených znakem `#`, kde každá instrukce má tvar popsaný v (5.2). Na to stačí jeden průchod kódem $\langle M \rangle$ a není třeba další páska. Součástí syntaktické kontroly by mohl být i test toho, zda se v kódu nenacházejí dvě instrukce s týmž displejem, tj. test toho, zda je stroj na vstupu deterministický. Tento test však pro práci $\mathcal{U}$ není podstatný, protože pokud jsou v kódu $\langle M \rangle$ dvě instrukce s týmž displejem, použije se ta, kterou najde $\mathcal{U}$ jako první.

Krok 2 Pokud syntaktická kontrola selhala, je M podle úmluvy 5.2.3 považován za Turingův stroj s prázdnou přechodovou funkcí a odmítnutí je zde tedy na místě.

Krok 4 Délka bloku obsahující binární zápis čísla znaku na 2. pásce musí být dostatečně velká, aby do něj bylo možno zapsat kterýkoli znak, jenž se vyskytuje v kódu $\langle M \rangle$. Délku tohoto bloku si označíme pomocí b, její určení provede $\mathcal{U}$ následujícím způsobem: Před čtením bloku v kódu instrukce, který kóduje znak páskové abecedy (tj. 2. a 4. blok v pětici popisující instrukci, viz (5.2)) se $\mathcal{U}$ vrátí na začátek 2. pásky. Poté se čtením bloku píše na 2. pásku znaky `0`. Na konci po přečtení celého kódu $\langle M \rangle$ zůstane na 2. pásce nejdelší posloupnost znaků `0`. Tím bude dána délka bloku na pracovní pásce. Mezi každé dva bloky je vždy vložen znak oddělovače `|`. Vždy, když během simulace potřebuje UTS přiformátovat další blok délky b na 2. pásku, formátuje ho na délku totožnou se sousedícím blokem. To může $\mathcal{U}$ učinit tak, že si bude označovat naposledy přepsané políčko v předchozím bloku. Přitom si může $\mathcal{U}$ pamatovat označenou číslici ve stavu a psát místo ní na chvíli například

Algoritmus 5.2.1 Výpočet univerzálního Turingova stroje $\mathcal{U}$.

Vstup: Kód simulovaného stroje $\langle M \rangle$, kde $M = (Q, \Sigma, \delta, q_0, \{q_1\})$ a vstupní řetězec x, obojí zapsané na vstupní pásce. Druhá a třetí páska jsou prázdné.

Výstup: Výpočet $\mathcal{U}$ se zastaví, právě když výpočet $M(x)$ je konečný. Výpočet $\mathcal{U}$ je přijímající, právě když je přijímající výpočet $M(x)$. Po ukončení výpočtu obsahuje 2. páska slovo, které zůstalo na pásce M po ukončení výpočtu (zakódované způsobem popsaným při popisu 2. pásky $\mathcal{U}$).

Inicializace

if $\langle M \rangle$ nekóduje syntakticky správně Turingův stroj **then**
    reject
end if
Urči délku b nejdelšího bloku pro zápis znaku na druhé pásce.
Překóduj vstup x na 2. pásku, tedy pracovní pásku M.
Na 3. zapiš znak `0`, tedy binární zápis čísla počátečního stavu q_0 stroje M.
Vrať hlavy na začátky všech tří pásek.

Hlavní cyklus simulace

while $\langle M \rangle$ obsahuje instrukci $\delta(q_i, X_j) = (q_k, X_l, Z)$, kde (q_i, X_j) je displej M **do**
    Přepiš 3. pásku řetězcem $(k)_B$, kódujícím nový stav q_k.
    Přepiš číslo znaku X_j v bloku pod hlavou na 2. pásce číslem znaku X_l.
    if $Z \in \{L, R\}$ **then**
        Přesuň hlavu na 2. pásce na sousední blok (v odpovídajícím směru).
        if v odpovídajícím směru není žádný sousední blok (páska je prázdná) **then**
            Přidej blok kódující $X_2 = \lambda$, tedy $\mathtt{0}^{b-2}\mathtt{10}$.
        end if
    end if
end while

Zakončení a úklid

if na 3. pásce je číslo stavu q_1 **then**
    accept
else
    reject
end if

znak #. Zapsanou značku si pak $\mathcal{U}$ posouvá spolu s připisováním znaku 0 do nově vytvářeného bloku.

Krok 5 Znak 0 je zapsán jako 0^b a znak 1 jako $0^{b-1}1$, kde b označuje délku bloku na 2. pásce (viz bližší popis kroku 4). Připomeňme si, že v kódování abecedy popsaném v sekci 5.2.1 je $X_0 = 0$ a $X_1 = 1$, proto jsou kódy těchto dvou znaků při dané velikosti bloku b už pevně dané. Pokud je vstup x prázdný, pak je na 2. pásku zapsán jediný blok $0^{b-2}10$ kódující znak prázdného políčka $X_2 = \lambda$.

Krok 8 V tomto kroku je v kódu $\langle M \rangle$ hledána instrukce, jejíž displej je shodný s aktuálním stavem a čteným znakem M. Tedy hledá se instrukce $\delta(q_i, X_j) = (q_k, X_l, Z)$, kde q_i je stav s číslem zapsaným na 3. pásce $\mathcal{U}$ a X_j je znak kódovaný v bloku 2. pásky, který se nachází pod hlavou $\mathcal{U}$. K tomu stačí jednou projít řetězec $\langle M \rangle$.

Krok 10 Binární řetězec $(l)_B$ kódující znak X_l zapisuje $\mathcal{U}$ odprava od konce příslušného bloku 2. pásky, přičemž jej zleva doplní nulami do délky b (tedy až k předchozímu znaku |).

Krok 14 Pokud by se při pohybu přesunul M nad prázdné nenaformátované políčko λ, je třeba je nahradit kódem prázdného políčka. Protože je pevně určeno $X_2 = \lambda$, je kódem prázdného políčka řetězec $0^{b-2}10$.

Z konstrukce UTS je zřejmé, že splňuje požadavky na něj kladené, zejména tedy platí, že $L(\mathcal{U}) = L_u = \{\langle M, x \rangle \mid x \in L(M)\}$. Z toho vyplývá, že univerzální jazyk je částečně rozhodnutelný.

Věta 5.2.9 *Univerzální jazyk $L(\mathcal{U}) = L_u = \{\langle M, x \rangle \mid x \in L(M)\}$ je částečně rozhodnutelný.*

5.3. Algoritmicky rozhodnutelné problémy

Nyní jsme připraveni upřesnit to, co míníme algoritmicky řešitelným problémem. Rozhodovací problém P je pro nás daný popisem instance I problému a otázkou, kterou si o dané instanci klademe. My si rozhodovací problém formalizujeme jako jazyk L_P, který obsahuje slova kódující kladné instance tohoto problému, tedy instance, u nichž je na otázku kladenou v daném problému kladná odpověď. Otázka kladená v problému P je tedy převedena na otázku, zda dané slovo patří do jazyka L_P, reprezentuje tedy instanci problému P takovou, že otázka problému P má pro tuto instanci kladnou odpověď.

Příklad 5.3.1: Jazyk problému Helloworld

Vzpomeňme si například na problém Helloworld. Instancí tohoto problému je dvojice souborů — soubor P se zdrojovým kódem v jazyce C a vstupní soubor I. V problému Helloworld se ptáme, zda prvních dvanáct znaků, jež program P se vstupem I vypíše, tvoří řetězec „Hello, world“. Jazyk L_{HW}, který tomuto problému

Problém 2.1.1

odpovídá, je potom

$$L_{HW} = \left\{ \langle P, I \rangle \,\middle|\, \begin{array}{l} P \text{ je program v jazyce C a } I \text{ je vstupní soubor, pro které platí,} \\ \text{že prvních dvanáct znaků, jež } P \text{ se vstupem } I \text{ vypíše, tvoří} \\ \text{řetězec „}\texttt{Hello, world}\text{“.} \end{array} \right\}$$

Teze 5.1.1

Na základě Churchovy-Turingovy teze můžeme nyní definovat pojem rozhodnutelného (a také částečně rozhodnutelného) jazyka (potažmo problému) následujícím způsobem.

Definice 5.3.2 (Rozhodnutelné a částečně rozhodnutelné jazyky) Nechť L je jazyk nad abecedou Σ. Řekneme, že jazyk L je

- *částečně rozhodnutelný* (též *rekurzivně spočetný*), pokud existuje Turingův stroj M, který jej přijímá, tedy $L = L(M)$ a že je
- *rozhodnutelný* (též *rekurzivní*), pokud existuje Turingův stroj M, který jej přijímá, tedy $L = L(M)$ a navíc se výpočet M zastaví pro každé vstupní slovo $x \in \Sigma^*$.

Třídu rozhodnutelných jazyků budeme označovat DEC (z *decidable*), třídu částečně rozhodnutelných jazyků budeme označovat PD (z *partially decidable*) a třídu doplňků částečně rozhodnutelných jazyků budeme označovat co-PD $= \{L \mid \overline{L} \in \text{PD}\}$. ◂

Viz též kapitoly 3.3.2 a 3.4.4

Očíslování Turingových strojů, jež jsme zavedli v kapitole 5.2.1, můžeme nyní použít i k očíslování částečně rozhodnutelných jazyků. Z toho plyne, třída PD je spočetná množina. Na druhou stranu třída všech jazyků nad abecedou Σ, tedy $\wp(\Sigma^*)$ je nespočetná dle Cantorovy věty, což platí už pro jednoprvkovou abecedu $\Sigma = \{1\}$. Z toho plyne, že ve skutečnosti většina jazyků není částečně rozhodnutelná, tím spíše to platí o jazycích rozhodnutelných.

5.4. Algoritmicky vyčíslitelné funkce

Teze 5.1.1

Dalším důležitým pojmem pro nás budou funkce, jejichž hodnotu lze vyčíslit algoritmicky. Budeme převážně uvažovat řetězcové funkce. Na základě Churchovy-Turingovy teze můžeme pojem algoritmicky vyčíslitelné (řetězcové) funkce definovat následujícím způsobem.

Definice 5.4.1 (Algoritmicky vyčíslitelná funkce) Nechť Σ je konečná abeceda a $f : \Sigma^* \to \Sigma^*$ je (částečná) funkce. Řekneme, že f je *algoritmicky vyčíslitelná*, pokud je tato funkce turingovsky vyčíslitelná (ve smyslu definice 4.1.8). ◂

Je potřeba zmínit, že z principu nejsou všechny algoritmicky vyčíslitelné funkce totální, tedy definované pro všechny vstupy. To proto, že Turingovy stroje se nemusí zastavit pro všechny vstupy. Přesněji, je-li $f : \Sigma^* \to \Sigma^*$ algoritmicky vyčíslitelná funkce, jež je vyčíslovaná Turingovým strojem M, pak

$$\operatorname{dom} f = \{x \in \Sigma^* \mid M(x)\downarrow\}. \tag{5.5}$$

Funkce f je tedy totální (čili definovaná pro všechny vstupy), má-li M tu vlastnost, že jeho výpočet se zastaví pro každý vstup x. Totální algoritmicky vyčíslitelné funkce pro nás budou dále podstatné například při definici převoditelnosti.

Funkce, které zobrazují řetězce na řetězce, jsou velmi obecné. Uvážíme-li kódování objektů zavedené v definici 5.2.7, dají se i funkce jiných typů převést na řetězcové funkce. Dále budeme uvažovat zejména funkce více parametrů a aritmetické funkce s číselnými parametry.

Úmluva 5.4.2 (Funkce více parametrů a různých typů) *Aritmetické funkci $f : \mathbb{N}^n \to \mathbb{N}$ takto odpovídá funkce $f' : \Sigma^* \to \Sigma^*$, pro niž platí, že pro libovolnou n-tici vstupních hodnot $x_1, \ldots, x_n \in \mathbb{N}$ je $f'(\langle x_1, \ldots, x_n \rangle) \simeq \langle f(x_1, \ldots, x_n) \rangle$. Například funkci $f(x_1, x_2) = x^2 + y^2$ odpovídá řetězcová funkce $f'(\langle x_1, x_2 \rangle) = \langle x^2 + y^2 \rangle$. Dále budeme volně používat jako parametrů i hodnot funkcí řetězce či čísla podle potřeby.*

Podobně můžeme použít kódování objektů i při použití parametrů a hodnot funkcí jiných typů.

Poznamenejme, že na základě úmluvy 5.4.2 a Churchovy-Turingovy teze splývá pojem algoritmicky vyčíslitelné funkce s pojmy RAM-vyčíslitelné funkce a částečně rekurzivní funkce.

Teze 5.1.1

Definice 4.2.3 a 4.4.7

Očíslování Turingových strojů, jež jsme zavedli v kapitole 5.2.1, můžeme nyní použít i k očíslování algoritmicky vyčíslitelných funkcí. Na základě úmluvy 5.4.2 můžeme rozšířit číslování i na funkce více parametrů.

5.5. Univerzální funkce

Podobně jako máme k dispozici univerzální jazyk, který v sobě kóduje všechny částečně rozhodnutelné jazyky, máme na základě univerzálního Turingova stroje k dispozici i univerzální funkci, která reprezentuje ostatní algoritmicky vyčíslitelné funkce.

Věta 5.5.1 (O univerzální funkci) *Definujme* univerzální funkci *pro algoritmicky vyčíslitelné funkce jako funkci Ψ, která pro každou dvojici $\langle M, x \rangle$, kde M je Turingův stroj vyčíslující funkci f_M a x je řetězec, splňuje*

$$\Psi(\langle M \rangle, x) \simeq f_M(x). \tag{5.6}$$

Potom funkce Ψ je algoritmicky vyčíslitelná.

Důkaz: Univerzální Turingův stroj $\mathcal{U}$ je schopen simulovat jiné Turingovy stroje se zadaným vstupem a je jistě možné jej upravit tak, aby funkce jím vyčíslovaná byla právě univerzální funkce Ψ. □

Viz sekce 5.2.3

5.6. Bibliografické poznámky

5.7. Cvičení

Konstrukce Turingových strojů

V následujících cvičeních můžete pro konstrukci Turingových strojů použít model k-páskového stroje, pokud se vám to hodí. Pod „Turingovým strojem, který rozhoduje jazyk L“, míníme Turingův stroj M, který se vždy zastaví a $L = L(M)$. „Sestrojte“ znamená včetně instrukcí. „Popište“ znamená popište práci odpovídajícího Turingova stroje bez rozepisování instrukcí.

1. Sestrojte Turingův stroj, který rozhoduje jazyk palindromů (pomocí w^R označujeme zrcadlově obrácené slovo w, tj. napsané pozpátku):

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

2. Sestrojte Turingův stroj, který rozhoduje jazyk

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

3. Sestrojte Turingův stroj, který počítá funkci sčítání $f(x,y) = x + y$.

4. Popište, jak by pracovaly Turingovy stroje počítající funkce $f(x,y) = x \cdot y$ (násobení), $f(x,y) = x \operatorname{div} y$ (celočíselné dělení), $f(x,y) = x \bmod y$ (zbytek po celočíselném dělení).

5. Popište, jak by pracoval Turingův stroj rozhodující jazyk

$$L = \{a^i b^j c^k \mid i = j \text{ nebo } i = k\}.$$

6. Popište, jak by pracoval Turingův stroj rozhodující jazyk

$$L = \{a^i b^j c^k \mid i = j \text{ nebo } i = k \text{ nebo } j = k\}.$$

7. Popište, jak by pracoval Turingův stroj, který převádí číslo x zakódované unárně, tj. řetězcem 1^x, na číslo x zakódované binárně, tj. $(x)_B$. Rozmyslete si i převod opačným směrem.

8. Popište Turingův stroj, který přijímá jazyk

$$L = \{x \in \{0,1\}^* \mid x \text{ je binární reprezentací prvočísla}\}.$$

9. Popište, jak by pracoval Turingův stroj M, který ignoruje svůj vstup a během své práce postupně na výstupní pásku zapisuje seznam prvočísel (zakódovaných binárně, oddělených například pomocí #), tj. na výstupní pásce se postupně objevuje seznam 2, 3, 5, 7, 11, 13, ...

Varianty a omezení modelu Turingova stroje

10. Ukažte, jak lze libovolný Turingův stroj M převést na stroj M', který pracuje stejně a má pouze binární abecedu, tj. vstupní abecedu $\{0,1\}$, v pracovní je navíc jen znak λ prázdného políčka.

11. Ukažte, jak lze libovolný Turingův stroj M rozšířit o práci se zarážkami. Modifikovaný stroj M' by měl mít v abecedě dva nové znaky, znak pro levou zarážku $\triangleright$ a znak pro pravou zarážku $\triangleleft$, při své práci potom udržuje invariant, že levá zarážka $\triangleright$ je na nejlevější pozici, kam se dostala hlava stroje M', zatímco pravá zarážka $\triangleleft$ je na nejpravější pozici, na kterou se dostala během výpočtu hlava stroje M', jinak stroj M' pracuje stejně jako M. (*tj. zarážky v M' ohraničují prostor na pásce skutečně využitý strojem M' při výpočtu nad daným vstupem.*)

12. Ukažte, jak lze libovolný Turingův stroj M s jednou obousměrně potenciálně nekonečnou páskou převést na Turingův stroj M', který má pouze jednu jednosměrně (doprava) potenciálně nekonečnou pásku. V tomto modelu předpokládáme, že na začátku pásky je znak levé zarážky $\triangleright$, za ním následuje vstup. Hlava M' je na začátku na nejlevějším symbolu vstupu. Zarážka během výpočtu pomáhá M' poznat, kde je začátek pásky a z kterého políčka již nesmí učinit krok vlevo.

13. Ukažte, jak lze libovolný Turingův stroj M s $k \geq 1$ páskami převést na jednopáskový Turingův stroj M'. Stačí rozmyslet si princip úpravy M na M'. *Jde tedy o náznak důkazu věty 4.1.12.*

14. Ukažte, jak lze libovolný (jednopáskový) Turingův stroj M převést na Turingův stroj M', který v každém kroku provádí jen dvě ze tří možných akcí (tj. každá instrukce buď změní stav a pozici hlavy, změní stav a písmeno na pásce, nebo změní písmeno na pásce a pozici hlavy, ale neučiní všechny tyto akce najednou).

15. Ukažte, jak lze jednopáskový Turingův stroj M převést na Turingův stroj M', který ve svých instrukcích vždy pohne hlavou, tj. v žádné instrukci nezůstane hlava stát na místě.

16. Rozmyslete si, jakou třídu jazyků přijímají jednopáskové Turingovy stroje, které nesmí pohybovat hlavou vpravo (tj. hlava může zůstat stát nebo se pohnout vlevo).

17. Rozmyslete si, jakou třídu jazyků přijímají jednopáskové Turingovy stroje, které nesmí pohybovat hlavou vlevo (tj. hlava může zůstat stát nebo se pohnout vpravo).

18. Uvažte Turingův stroj s jednou páskou, která je potenciálně nekonečná jen v jednom směru (doprava). Uvažme, že Turingovu stroji povolíme jen dva typy pohybu hlavou: R a RESET. Při pohybu R se hlava pohne o jedno políčko doprava, při RESET se hlava přesune na začátek pásky. Ukažte, že takto omezený Turingův stroj je ekvivalentní s jednopáskovým Turingovým strojem.

19. Ukažte, že Turingův stroj, který může na každé políčko pásky zapsat nejvýš jednou (číst ho může vícekrát), je ekvivalentní s jednopáskovým Turingovým strojem.

20. Ukažte, že pokud jednopáskovému Turingovu stroji zakážeme přepisovat políčka vstupu, pak takto omezené stroje přijímají právě regulární jazyky.

21. Stroj s více zásobníky definujeme podobně jako Turingův stroj, s tím rozdílem, že k páskám přistupuje jako k zásobníkům, tj. jedná se o rozšíření zásobníkového automatu o více zásobníků. V jednom kroku může stroj přidat symboly na zásobníky, odebrat jej ze zásobníky, nebo je nechat netknuté, na každém zásobníku pracuje nezávisle. V každém kroku se rozhoduje na základě svého stavu a znaků na vrcholech svých zásobníků. Instrukce v případě tří zásobníků může vypadat například takto

$$\delta(q, a, b, c) = (q', PUSH\ a', POP, NOP),$$

tato instrukce v situaci, kdy řídící jednotka je ve stavu q a na vrcholech zásobníků jsou symboly a, b a c, přikazuje změnit stav řídící jednotky na q', na první zásobník připsat a', z druhého zásobníku odstranit vrchol zásobníku a třetí zásobník nechat netknutý. Symbol prázdného zásobníku $\triangleleft$ na vrcholu zásobníku znamená, že je prázdný a operace POP jej nechá netknutý, tento symbol nesmí být použit v operaci $PUSH$. Na začátku je vstup v prvním zásobníku.

Ukažte, že tento model je ekvivalentní s modelem Turingova stroje (tj. mezi oběma modely lze převádět oběma směry), připustíme-li více než jeden zásobník.

Rozhodnutelné a částečně rozhodnutelné jazyky

22. Ukažte, že rozhodnutelné jazyky jsou uzavřeny na operace sjednocení, průnik, doplněk, konkatenace (*jsou-li L_1 a L_2 jazyky, pak jejich konkatenací je jazyk $L = L_1 \circ L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$*) a hvězdička (*je-li L jazyk, pak jeho uzávěrem na hvězdičku je jazyk $L^* = \{x_1 x_2 \ldots x_k \mid (k \geq 0) \wedge \bigwedge_{i=1}^{k} (x_i \in L)\}$*).

23. Ukažte, že částečně rozhodnutelné jazyky jsou uzavřeny na operace sjednocení, průnik, konkatenace a hvězdička, zkuste si rozmyslet, jak je to s uzavřeností na doplněk.

6. Částečně rozhodnutelné jazyky a jejich vlastnosti

V této kapitole probereme některé vlastnosti částečně rozhodnutelných jazyků (a tedy i problémů). Podíváme se na některé ekvivalentní charakterizace částečně rozhodnutelných a rozhodnutelných jazyků, na uzávěrové vlastnosti těchto tříd. Ukážeme si navíc, že univerzální jazyk není algoritmicky rozhodnutelný.

6.1. Základní vlastnosti

V této kapitole prozkoumáme některé základní vlastnosti částečně rozhodnutelných jazyků.

6.1.1. Jednoduché ekvivalentní definice

Částečně rozhodnutelné jazyky jsme definovali jako jazyky přijímané Turingovými strojem. Mohli bychom je však ekvivalentně definovat pomocí zastavení Turingova stroje, jako domény algoritmicky vyčíslitelných funkcí, nebo pomocí existenční kvantifikace a rozhodnutelného jazyka.

Definice 5.3.2

Věta 6.1.1 (Ekvivalentní definice částečně rozhodnutelných jazyků) *Nechť L je jazyk nad abecedou Σ. Pak následující tvrzení jsou ekvivalentní:*

(*i*) *L je částečně rozhodnutelný.*

(*ii*) *Existuje Turingův stroj M splňující*

$$L = \{x \in \Sigma^* \mid M(x)\downarrow\}. \tag{6.1}$$

(*iii*) *Existuje algoritmicky vyčíslitelná funkce f splňující*

$$L = \operatorname{dom} f. \tag{6.2}$$

(*iv*) *Existuje rozhodnutelný jazyk B splňující*

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)[\langle x, y\rangle \in B]\}. \tag{6.3}$$

Důkaz: Důkaz si rozdělíme na jednotlivé implikace.

(i) ⇒ *(ii)* Je-li L částečně rozhodnutelný jazyk, pak $L = L(M)$ pro nějaký Turingův stroj M. Na základě tohoto stroje zkonstruujeme Turingův stroj M', který se vstupem $x \in \Sigma^*$ pracuje následujícím způsobem:

Výpočet M' se vstupem x:

1: Pusť $M(x)$.
2: **if** M přijal vstup x **then**
3: **accept**
4: **else**
5: pokračuj neukončeným výpočtem (zacykli se).
6: **end if**

Je zřejmé, že $x \in L = L(M)$, právě když $M'(x)\downarrow$, M' tedy splňuje (6.1).

(ii) ⇒ *(i)* Nechť M je Turingův stroj, který splňuje (6.1). Sestrojíme Turingův stroj M', který se vstupem $x \in \Sigma^*$ pracuje následujícím způsobem:

Výpočet M' se vstupem x:

1: Pusť $M(x)$.
2: **accept**

Je zřejmé, že potom M' přijímá právě řetězce z L, a tedy $L = L(M')$.

Definice 5.4.1

(ii) ⇔ *(iii)* Tato ekvivalence plyne přímo z definice algoritmicky vyčíslitelné funkce a z (5.5).

(i) ⇒ *(iv)* Předpokládejme, že L je částečně rozhodnutelný. Nechť M je Turingův stroj, který přijímá L, tedy $L = L(M)$. Definujme jazyk B následujícím způsobem:

$$B = \{\langle x, y\rangle \mid (x \in \Sigma^*), (y \in \mathbb{N}) \text{ a } M \text{ přijme } x \text{ v nejvýše } y \text{ krocích}\}$$

Jazyk B je rozhodnutelný, neboť k rozhodnutí toho, zda daná dvojice $\langle x, y\rangle$ patří do B stačí simulovat běh $M(x)$ po y kroků. Pokud do té doby M přijme, pak $\langle x, y\rangle \in B$. Pokud se do té doby M nezastaví, nebo pokud odmítne, pak $\langle x, y\rangle \notin B$. Přitom platí, že $x \in L$, právě když $M(x)$ přijme. A pokud y je počet kroků tohoto přijímajícího výpočtu, pak $\langle x, y\rangle \in B$. Jazyk B tedy splňuje (6.3).

(iv) ⇒ *(i)* Předpokládejme, že B je jazyk, který splňuje (6.3). Zkonstruujme Turingův stroj M, který se vstupem x pracuje podle následujícího algoritmu:

Výpočet M se vstupem x:

1: **for all** $y \in \Sigma^*$ **do**

2: **if** $\langle x, y \rangle \in B$ **then**
3: **accept**
4: **end if**
5: **end for**

Algoritmus generuje v cyklu všechny řetězce y ze Σ^* v shortlex uspořádání. Pro každý takový řetězec y algoritmus otestuje v kroku 2, zda právě on není svědkem toho, že $x \in L$. □

Důkaz toho, že (iv) implikuje (i) ve větě 6.1.1 lze snadno upravit i pro situaci, kdy jazyk B je jen částečně rozhodnutelný. To znamená, že nezáleží na tom, kolik existenčně kvantifikovaných proměnných použijeme v (6.3). Zformulujme toto tvrzení přesněji.

Věta 6.1.2 *Nechť B je částečně rozhodnutelný jazyk, potom i jazyk*

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)[\langle x, y \rangle \in B]\} \tag{6.4}$$

je částečně rozhodnutelný.

Důkaz: Je-li B částečně rozhodnutelný jazyk, pak podle bodu (iv) věty 6.1.1 existuje rozhodnutelný jazyk A, který splňuje, že

$$B = \{u \in \Sigma^* \mid (\exists v \in \Sigma^*)[\langle u, v \rangle \in A]\}.$$

Dohromady s (6.4) dostáváme, že

$$L = \{x \in \Sigma^* \mid (\exists u \in \Sigma^*)(\exists v \in \Sigma^*)[\langle \langle x, u \rangle, v \rangle \in A]\}.$$

Položme jazyk $A' = \{\langle x, u, v \rangle \mid \langle \langle x, u \rangle, v \rangle \in A\}$. Z rozhodnutelnosti jazyka A plyne, že i jazyk A' je rozhodnutelný. Navíc platí, že

$$L = \left\{x \in \Sigma^* \mid (\exists u \in \Sigma^*)(\exists v \in \Sigma^*)[\langle x, u, v \rangle \in A']\right\}.$$

Pokud nyní budeme uvažovat $y = \langle u, v \rangle$, pak

$$L = \left\{x \in \Sigma^* \mid (\exists y \in \Sigma^*)[y = \langle u, v \rangle \text{ a } \langle x, u, v \rangle \in A']\right\}. \tag{6.5}$$

Podmínka v (6.5) je rozhodnutelná a tedy podle bodu (iv) věty 6.1.1 je jazyk L částečně rozhodnutelný. □

Dá se tedy říci, že částečně rozhodnutelné jazyky jsou uzavřené na existenční kvantifikaci v podmínce, která daný jazyk definuje.

Příklad 6.1.3: Částečná rozhodnutelnost neprázdnosti jazyka

Uvažme problém, kde se ptáme o daném Turingovu stroji M, zda přijímá alespoň jeden vstup, tedy zda jazyk $L(M)$ přijímaný strojem M je neprázdný. Tento problém je formalizován pomocí jazyka následujícím způsobem.

$$\text{NONEMPTY} = \{\langle M \rangle \mid L(M) \neq \varnothing\}.$$

Tuto definici můžeme zapsat i jako

$$\text{NONEMPTY} = \{\langle M \rangle \mid (\exists x \in \Sigma^*)[x \in L(M)]\}\,,$$

S pomocí univerzálního jazyka $L_u = \{\langle M, x \rangle \mid x \in L(M)\}$ můžeme dále psát

$$\text{NONEMPTY} = \{\langle M \rangle \mid (\exists x \in \Sigma^*)[\langle M, x \rangle \in L_u]\}\,.$$

Jazyk L_u je částečně rozhodnutelný, jsa přijímaný univerzálním Turingovým strojem. Na základě věty 6.1.2 je tedy i jazyk NONEMPTY částečně rozhodnutelný. Později v příkladu 7.2.8 si ukážeme, že tento problém není rozhodnutelný.

6.1.2. Uzávěrové vlastnosti a Postova věta

V této kapitole se budeme věnovat jednoduchým uzávěrovým vlastnostem rozhodnutelných a částečně rozhodnutelných jazyků. Jde zejména o uzavřenost na běžné množinové operace.

Věta 6.1.4 *Uvažme jazyky L_1, L_2 nad abecedou Σ.*

- *Jsou-li L_1 a L_2 rozhodnutelné jazyky, potom $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 \cdot L_2$ a L_1^* jsou rozhodnutelné jazyky.*
- *Jsou-li L_1 a L_2 částečně rozhodnutelné jazyky, potom $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 \cdot L_2$ a L_1^* jsou částečně rozhodnutelné jazyky.*

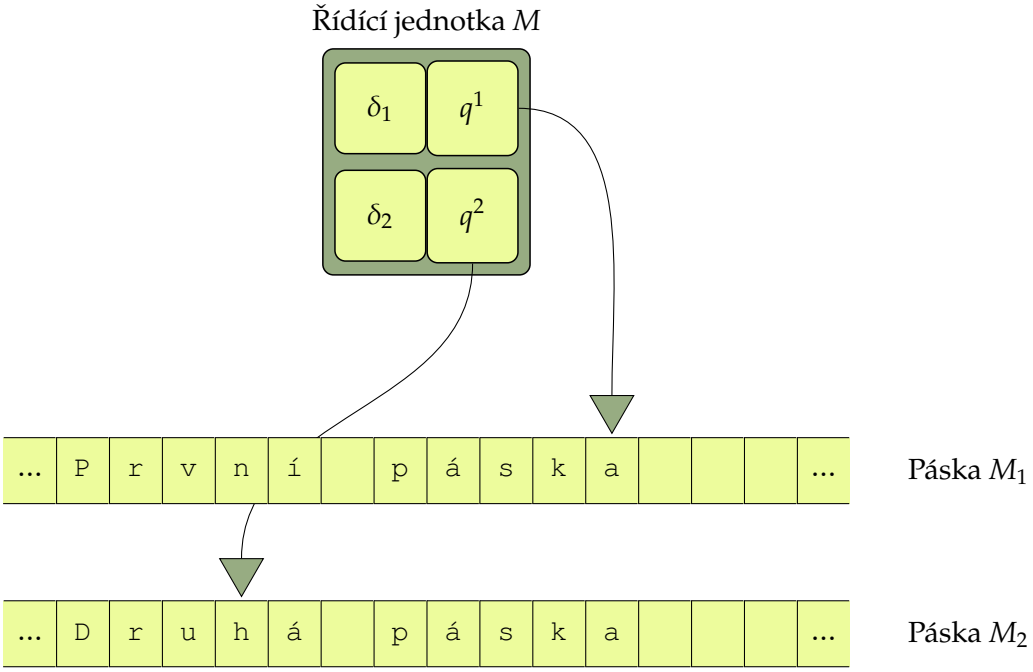
Důkaz: Tvrzení ukážeme pro sjednocení, přičemž průnik, konkatenaci $L_1 \cdot L_2$ a Kleeneho uzávěr L_1^* ponecháme jako cvičení. Předpokládejme, že $L_1 = L(M_1)$ a $L_2 = L(M_2)$ pro dva Turingovy stroje M_1 a M_2. Popíšeme konstrukci Turingova stroje M, který přijímá jazyk $L_1 \cup L_2$. Idea práce Turingova stroje M je velmi jednoduchá, se vstupem x bude postupovat podle následujícího algoritmu.

Výpočet Turingova stroje M se vstupem x, kde $L(M) = L_1 \cup L_2$

1: Pusť $M_1(x)$ a $M_2(x)$ paralelně.
2: **if** jeden z výpočtů $M_1(x)$ a $M_2(x)$ skončí přijetím **then**
3: **accept**
4: **end if**
5: **if** oba výpočty $M_1(x)$ a $M_2(x)$ skončí odmítnutím **then**
6: **reject**
7: **end if**

Z existence Turingova stroje M přímo plyne, že $L_1 \cup L_2$ je částečně rozhodnutelný jazyk. Je zřejmé, že pokud M_1 a M_2 jsou Turingovy stroje, jejichž výpočet je pro každý vstup konečný, pak má tuto vlastnost i stroj M. To znamená, že pokud L_1 a L_2 jsou rozhodnutelné jazyky, pak je jazyk $L_1 \cup L_2$ rovněž rozhodnutelný.

Zbývá popsat, jakým způsobem bude Turingův stroj M simulovat paralelní běh dvou Turingových strojů M_1 a M_2. Podle předpokladu jsou M_1 i M_2 jednopáskové Turingovy stroje. Stroj M popíšeme jako dvoupáskový s tím, že na základě věty 4.1.12 jej můžeme převést na jednopáskový. Na každé pásce stroje M bude probíhat výpočet jednoho z strojů M_1 a M_2. Přitom využíváme toho, že hlavy na páskách se pohybují nezávisle. Stav řídící jednotky M bude složen ze stavů strojů M_1 a M_2 a přechodová funkce M bude složena z přechodových funkcí M_1 a M_2. Řídící jednotka M tak v sobě vlastně obsahuje řídící jednotky obou strojů M_1 i M_2.



Obrázek 6.1.: Struktura Turingova stroje M, v němž paralelně běží výpočty Turingových strojů M_1 a M_2. Stav řídící jednotky M se skládá ze stavu q^1 Turingova stroje M_1 a q^2 Turingova stroje M_2. Přechodová funkce Turingova stroje M vznikne kombinací přechodové funkce δ_1 Turingova stroje M_1 a přechodové funkce δ_2 Turingova stroje M_2. První páska odpovídá pásce stroje M_1 a hlava na ní je řízena stavem q^1 a přechodovou funkcí δ_1. Podobně druhá páska odpovídá pásce stroje M_2 a hlava na ní je řízena stavem q^2 a přechodovou funkcí δ_2.

Pro úplnost si popíšeme konstrukci M i detailně, čtenář, který si dovede konstrukci M představit již na základě předešlého popisu, může zbytek důkazu směle přeskočit. Předpokládejme, že $M_1 = (Q_1, \Sigma_1, \delta_1, q_0^1, F_1)$ a $M_2 = (Q_2, \Sigma_2, \delta_2, q_0^2, F_2)$, na základě nich zkonstruujeme $M = (Q, \Sigma, \delta, q_0, F)$, kde

- $Q = (Q_1 \times Q_2) \cup \{q_0, q_{back}\}$,

- $\Sigma = \Sigma_1 \cup \Sigma_2$ a
- $F = (Q_1 \times F_2) \cup (F_1 \times Q_2)$.

Přechodová funkce δ je definovaná následujícím způsobem. Nejprve přidáme přechody, které okopírují vstup z první pásky i na druhou pásku.

$$\begin{aligned}
(\forall a \in \Sigma \setminus \{\lambda\}) \quad & \delta(q_0, a, \lambda) = (q_0, a, a, R, R) \\
& \delta(q_0, \lambda, \lambda) = (q_{back}, \lambda, \lambda, L, L) \\
(\forall a \in \Sigma \setminus \{\lambda\}) \quad & \delta(q_{back}, a, a) = (q_{back}, a, a, L, L) \\
& \delta(q_{back}, \lambda, \lambda) = ([q_0^1, q_0^2], \lambda, \lambda, R, R)
\end{aligned}$$

Dále definujeme hodnoty přechodové funkce, které odpovídají přechodovým funkcím strojů M_1 a M_2. Pro každou dvojici stavů $q^1 \in Q$ a $q^2 \in Q_2$ a dvojici znaků $a, b \in \Sigma$ definujeme hodnotu $\delta([q^1, q^2], a, b)$ jedním z následujících způsobů:

1. Je-li $\delta_1(q^1, a) = (r^1, c, Z_1)$ a $\delta_2(q^2, b) = (r^2, d, Z_2)$, kde $r^1 \in Q_1$, $r^2 \in Q_2$, $c, d \in \Sigma$ a $Z_1, Z_2 \in \{L, N, R\}$, pak definujeme

   $$\delta([q^1, q^2], a, b) = ([r^1, r^2], c, d, Z_1, Z_2).$$

2. Je-li $\delta_1(q^1, a) = (r^1, c, Z_1)$ a $\delta_2(q^2, b) = \perp$, kde $r^1 \in Q_1$, $c \in \Sigma$ a $Z_1 \in \{L, N, R\}$, a pokud navíc platí, že $q^2 \notin F_2$, pak definujeme

   $$\delta([q^1, q^2], a, b) = ([r^1, q^2], c, b, Z_1, N).$$

   Pokud platí, že $q^2 \in F_2$ (je to přijímající stav), pak ponecháme $\delta([q^1, q^2], a, b) = \perp$ (výpočet končí přijetím).

3. Je-li $\delta_1(q^1, a) = \perp$ a $\delta_2(q^2, b) = (r^2, d, Z_2)$, kde $r^2 \in Q_2$, $d \in \Sigma$ a $Z_2 \in \{L, N, R\}$, a pokud navíc platí, že $q^1 \notin F_1$, pak definujeme

   $$\delta([q^1, q^2], a, b) = ([q^1, r^2], a, d, N, Z_2).$$

   Pokud platí, že $q^1 \in F_1$ (je to přijímající stav), pak ponecháme $\delta([q^1, q^2], a, b) = \perp$ (výpočet končí přijetím).

4. V ostatních případech je $\delta([q^1, q^2], a, b) = \perp$, výpočet tedy končí. □

Způsob důkazu, kterým jsme ukazovali ve větě 6.1.4, že sjednocením dvou částečně rozhodnutelných jazyků vznikne opět částečně rozhodnutelný jazyk, využijeme i při důkazu Postovy věty. Tato věta dává do souvislosti rozhodnutelné jazyky, částečně rozhodnutelné jazyky a jejich doplňky.

Věta 6.1.5 (Postova věta) *Jazyk $L \subseteq \Sigma^*$ je rozhodnutelný, právě když jazyky L i $\overline{L}$ jsou částečně rozhodnutelné.*

Důkaz: Předpokládejme nejprve, že jazyk L je rozhodnutelný. Není těžké nahlédnout, že doplněk jazyka L, tedy jazyk $\overline{L}$, je také rozhodnutelný. Stačí uvážit, že je-li M Turingův stroj, který rozhoduje L (tj. $L = L(M)$ a $M(x)\downarrow$ pro každý vstup $x \in \Sigma^*$), pak Turingův stroj M', který vznikne z M tak, že zaměníme význam přijímajícího a nepřijímajícího stavu, tj. znegujeme jeho odpověď, bude rozhodovat jazyk $\overline{L}$. Z definice rozhodnutelného jazyka přímo plyne, že oba jazyky L a $\overline{L}$, jsouce rozhodnutelné, jsou i částečně rozhodnutelnými jazyky. Definice 5.3.2

Předpokládejme nyní, že jazyky L a $\overline{L}$ jsou oba částečně rozhodnutelné a ukažme, že za tohoto předpokladu je jazyk L ve skutečnosti rozhodnutelný. Podle předpokladu existují Turingovy stroje M a M', pro které platí, že $L = L(M)$ a $\overline{L} = L(M')$. Turingův stroj N, který bude rozhodovat L bude pracovat podle následujícího algoritmu.

Výpočet N se vstupem x:

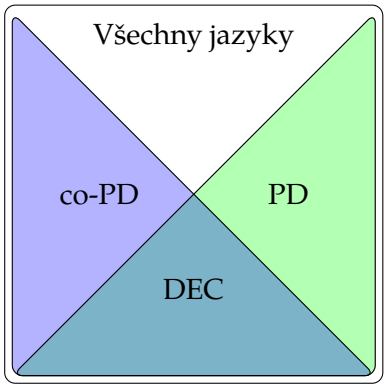
1: Pusť $M(x)$ a $M'(x)$ paralelně, dokud jeden z nich nepřijme.
2: **if** výpočet $M(x)$ skončil přijetím **then**
3: **accept**
4: **else if** výpočet $M'(x)$ skončil přijetím **then**
5: **reject**
6: **end if**

Uvědomme si, že pro každý vstup $x \in \Sigma^*$ platí, že $x \in L(M)$ nebo $x \in L(M')$. To proto, že $L(M) \cup L(M') = L \cup \overline{L} = \Sigma^*$. Z toho důvodu simulace v kroku 1 skončí s tím, že jeden ze strojů M nebo M' přijme vstup x. Pokud přijal Turingův stroj M, pak jistě $x \in L$, pokud naopak přijal Turingův stroj M', pak $x \in \overline{L}$ a tedy $x \notin L$. Platí tedy, že $L = L(N)$ a navíc $N(x)\downarrow$ pro každý vstup x. Paralelní běh strojů M a M' lze zabezpečit týmž způsobem jako v důkazu věty 6.1.4. □

Připomeňme si, že částečně rozhodnutelné jazyky jsou dále uzavřeny na existenční kvantifikaci ve smyslu věty 6.1.2. Toto však nelze říci o všeobecném kvantifikátoru, pomocí nějž lze naopak charakterizovat doplňky částečně rozhodnutelných jazyků. Uvidíme dále, že univerzální jazyk L_u definovaný v (5.4) je sice částečně rozhodnutelný, ale nikoli rozhodnutelný. Připomeňme si značení zavedené v definici 5.3.2, kde jsme pomocí DEC označili třídu rozhodnutelných jazyků, pomocí PD třídu částečně rozhodnutelných jazyků a pomocí co-PD třídu jejich doplňků. Platí tedy, že DEC $\subsetneq$ PD, DEC $\subsetneq$ co-PD a podle Postovy věty navíc DEC = PD $\cap$ co-PD. Tato situace je naznačena na obrázku 6.2.

6.2. Proč nemohou všechny jazyky být částečně rozhodnutelné

Zamysleme se nyní nad tím, zda mohou být všechny jazyky nad nějakou abecedou Σ alespoň částečně rozhodnutelné. Ukážeme si, že nikoli, důvod je přitom jednoduchý, všech jazyků již nad jednoprvkovou abecedou není spočetně mnoho, zatímco těch částečně rozhodnutelných je jen spočetně mnoho. K popisu konkrétního jazyka, který není



Obrázek 6.2.: Vztahy mezi třídami rozhodnutelných jazyků DEC, částečně rozhodnutelných jazyků PD a doplňků částečně rozhodnutelných jazyků co-PD.

částečně rozhodnutelný, použijeme důkaz diagonalizací.

Diagonalizační argument si ukážeme při důkazu Cantorovy věty, která říká, že potenční množina $\wp(A)$ množiny A nemůže mít shodnou mohutnost s množinou A.

Věta 6.2.1 *Nechť A je množina a $\wp(A)$ označuje její potenční množinu. Potom $\wp(A)$ má větší mohutnost než množina A.*

Důkaz: Jistě platí, že mohutnost množiny $\wp(A)$ je shodná nebo větší než mohutnost množiny A, neboť zobrazení $g : A \to \wp(A)$ definované pro $a \in A$ jako $g(a) = \{a\}$ je jistě prosté. Ukážeme, že mohutnost množiny $\wp(A)$ nemůže být shodná s mohutností množiny A. K tomu je třeba ukázat, že neexistuje bijekce mezi množinami $\wp(A)$ a A. Nechť $f : A \to \wp(A)$ je libovolné prosté zobrazení množiny A do $\wp(A)$. Ukážeme, že existuje množina $B \subseteq A$ taková, že pro žádný prvek a neplatí $f(a) = B$ a tedy že f nemůže být bijekcí. Definujme množinu B jako

$$B = \{a \in A \; a \notin f(a)\}. \tag{6.6}$$

Kdyby pro nějaký prvek a platilo, že $f(a) = B$, pak dostáváme, že

$$a \in f(a) \Leftrightarrow a \in B \Leftrightarrow a \notin f(a), \tag{6.7}$$

kde první ekvivalence platí díky tomu, že $f(a) = B$ a druhá ekvivalence platí dle definice B v (6.6). Platilo by tedy, že a patří do $f(a)$, právě když tam nepatří, což není pochopitelně možné, a takový prvek tedy neexistuje.

Dostáváme tedy, že pro každé prosté zobrazení $f : A \to \wp(A)$ platí, že existuje prvek $B \in \wp(A)$, který není obrazem žádného prvku $a \in A$, a tedy f nemůže být bijekce. Protože neexistuje bijekce mezi množinou A a množinou $\wp(A)$, platí tedy, že mohutnost $\wp(A)$ je větší než mohutnost A. □

V důkazu věty 6.2.1 jsme použili diagonalizačního argumentu, který spočíval v tom, že jsme při definici jazyka B použili ve výrazu $a \notin f(a)$ prvek a na obou stranách relace.

Díváme se tak na diagonální prvky relace $b \in f(a)$. Jako jednoduchý důsledek tohoto tvrzení dostáváme, že množina jazyků nad konečnou abecedou není spočetná.

Důsledek 6.2.2 *Nechť Σ je neprázdná konečná abeceda a nechť $\mathcal{L} = \wp(\Sigma^*)$ je množina jazyků nad abecedou Σ. Potom $\mathcal{L}$ není spočetná množina.*

Důkaz: Protože Σ je neprázdná množina, obsahuje nějaký znak, označme si jej třeba `1`. Potom $f(n) = \texttt{1}^n$ je prostým zobrazením $\mathbb{N}$ do Σ^*, tedy mohutnost Σ^* je shodná nebo větší než mohutnost $\mathbb{N}$[1]. Podle věty 6.2.1 tedy platí, že mohutnost $\mathcal{L} = \wp(\Sigma^*)$ je větší než mohutnost $\mathbb{N}$ a množina $\mathcal{L}$ tedy není spočetná. □

Uvědomme si, že třída částečně rozhodnutelných jazyků PD je spočetná, neboť každému částečně rozhodnutelnému jazyku můžeme přiřadit Gödelovo číslo Turingova stroje, který jej přijímá. Můžeme tedy psát $\text{PD} = \{L(M_e) \mid e \in \mathbb{N}\}$. Z toho plyne, že musí existovat jazyky, které nejsou částečně rozhodnutelné. Ve skutečnosti se dá říci, že většina jazyků nemůže být z principu ani částečně rozhodnutelná, natož rozhodnutelná.

6.3. Nerozhodnutelnost univerzálního jazyka

Nyní máme již dost nástrojů k tomu ukázat si nerozhodnutelnost univerzálního jazyka

$$L_u = \{\langle M, x\rangle \mid x \in L(M)\},$$

který je formalizací problému Přijetí vstupu. K důkazu tohoto faktu použijeme diagonalizační argument. Omezme se pro jednoduchost na Turingovy stroje, jejichž vstupní abeceda je binární a na jazyky, které jsou definované nad binární abecedou $\{\texttt{0},\texttt{1}\}$. Víme, že binární řetězce můžeme očíslovat přirozenými čísly pomocí číslování zavedeného v podkapitole 3.4.4, přesněji $\mathbf{w}_i$ označuje binární řetězec s číslem $i \in \mathbb{N}$. V podkapitole 5.2.1 jsme zavedli číslování Turingových strojů pomocí Gödelových čísel. Pomocí M_i označíme označili Turingův stroj s Gödelovým číslem $i \in \mathbb{N}$. Uvědomme si, že podle definice částečně rozhodnutelného jazyka jde právě o jazyky přijímané Turingovými stroji, číslování Turingových strojů lze tedy přenést i na částečně rozhodnutelné jazyky — i-tý částečně rozhodnutelný jazyk je ten, který je přijímán Turingovým strojem M_i, tedy $L(M_i)$.

Univerzální jazyk L_u byl definován vzorcem (5.4) v podkapitole 5.2.3, kde byl zaveden i univerzální Turingův stroj $\mathcal{U}$.

Definice 5.3.2

Na základě těchto úvah si můžeme charakteristickou funkci univerzálního jazyka reprezentovat jako matici $\mathbf{A}$, v níž řádky i sloupce jsou indexovány přirozenými čísly. Řádek s indexem $i \in \mathbb{N}$ odpovídá jazyku $L(M_i)$ přijímanému Turingovu stroji s Gödelovým číslem i. Sloupec s indexem $j \in \mathbb{N}$ pak odpovídá j-tému binárnímu slovu $\mathbf{w}_j$ (omezíme-li se na Turingovy stroje s binární vstupní abecedou). Hodnota prvku $\mathbf{A}_{i,j}$ je pak určena podle toho, zda $\mathbf{w}_j \in L(M_i)$, přesněji

$$\mathbf{A}_{i,j} = \begin{cases} 1 & \mathbf{w}_j \in L(M_i) \\ 0 & \mathbf{w}_j \notin L(M_i). \end{cases} \tag{6.8}$$

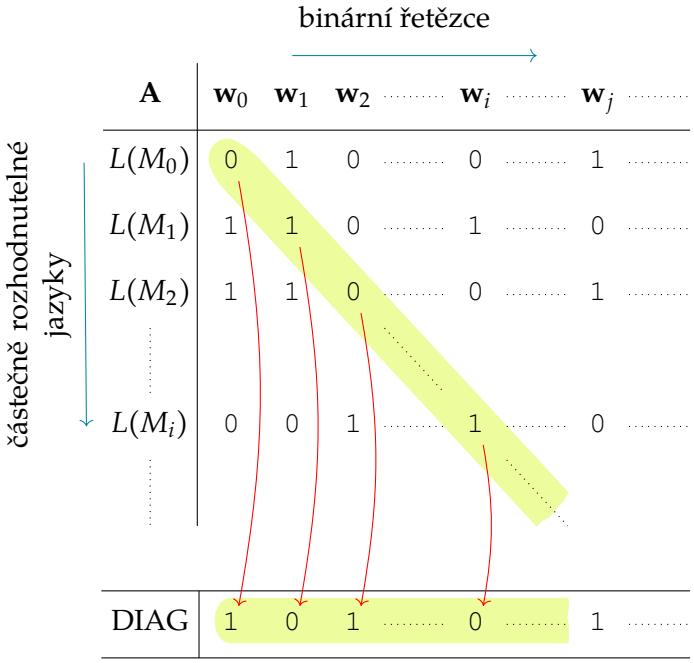
[1] Ve skutečnosti není těžké nahlédnout, že Σ^* je spočetná množina řetězců, ale to zde není třeba.

Řádek s indexem i tedy určuje charakteristickou funkci jazyka $L(M_i)$, přičemž každý částečně rozhodnutelný jazyk má v matici $\mathbf{A}$ svůj řádek (dokonce nekonečně mnoho řádků).

Chceme-li zkonstruovat jazyk, který není částečně rozhodnutelný, stačí zkonstruovat nekonečný vektor $\mathbf{b}$, který není roven žádnému řádku v matici $\mathbf{A}$. Uvážíme-li tento vektor jako charakteristickou funkci jazyka, půjde o jazyk, který není částečně rozhodnutelný (jinak by měl svůj řádek v matici $\mathbf{A}$). Diagonalizace nám nabízí jednoduchý způsob, jak takovýto vektor najít. Hodnotu $\mathbf{b}_i$ pro index $i = 1, \ldots, n$ určíme tak, aby se lišila od prvku $\mathbf{A}_{i,i}$, přesněji pro dané $i \in \mathbb{N}$ definujeme hodnotu vektoru $\mathbf{b}$ na indexu i jako

$$\mathbf{b}_i = 1 - \mathbf{A}_{i,i}.$$

Pro takto definovaný vektor $\mathbf{b}$ jistě platí, že se od každého řádku i sloupce matice $\mathbf{A}$ liší v diagonálním prvku. Tato situace je naznačena na obrázku 6.3.



Obrázek 6.3.: Matice $\mathbf{A}$ použitá při definici diagonálního jazyka DIAG.

Připomeňme si, že řetězec $\mathbf{w}_i$ je kódem Turingova stroje s Gödelovým číslem i, tedy pro každé $i \in \mathbb{N}$ platí $\mathbf{w}_i = \langle M_i \rangle$. Vektoru $\mathbf{b}$ tak odpovídá *Diagonální jazyk* DIAG daný doplňkem diagonály matice $\mathbf{A}$:

$$\mathrm{DIAG} = \{\langle M \rangle \mid \langle M \rangle \notin L(M)\}$$

Konvence 5.2.3

Diagonále matice $\mathbf{A}$ pak odpovídá doplněk diagonálního jazyka. Připomeňme si, že podle naší konvence každý řetězec kóduje nějaký Turingův stroj, tedy doplněk diago-

nálního jazyka je dán následujícím předpisem:

$$\overline{\text{DIAG}} = \{\langle M\rangle \mid \langle M\rangle \in L(M)\}. \tag{6.9}$$

Do DIAG tedy patří ty kódy Turingových strojů, jež nepřijímají vlastní kód. Vektor $\mathbf{b}$ určuje charakteristickou funkci jazyka DIAG, neboť pro $i \in \mathbb{N}$ platí, že

$$\mathbf{w}_i = \langle M_i\rangle \in \text{DIAG} \Leftrightarrow \mathbf{w}_i \notin L(M_i) \Leftrightarrow \mathbf{A}_{i,i} = 0 \Leftrightarrow \mathbf{b}_i = 1. \tag{6.10}$$

Z definice $\mathbf{b}$ plyne, že tento vektor není shodný s žádným řádkem matice $\mathbf{A}$, a tedy jazyk DIAG není částečně rozhodnutelný. Formální důkaz tohoto faktu lze zapsat velmi jednoduše.

Věta 6.3.1 *Jazyk* DIAG *není částečně rozhodnutelný. Jazyk* $\overline{\text{DIAG}}$ *není rozhodnutelný, je však částečně rozhodnutelný.*

Důkaz: Předpokládejme sporem, že DIAG je částečně rozhodnutelný jazyk. Existuje tedy Turingův stroj M, který tento jazyk přijímá, čili $L(M) = \text{DIAG}$. Potom ovšem

$$\langle M\rangle \in L(M) \Leftrightarrow \langle M\rangle \in \text{DIAG} \Leftrightarrow \langle M\rangle \notin L(M), \tag{6.11}$$

kde první ekvivalence plyne z faktu $L(M) = \text{DIAG}$ a druhá ekvivalence plyne z definice diagonálního jazyka v (6.3). Tím dostáváme $\langle M\rangle \in L(M) \Leftrightarrow \langle M\rangle \notin L(M)$, což je pochopitelně spor. Jazyk DIAG tedy nemůže být částečně rozhodnutelný. Nerozhodnutelnost jazyka $\overline{\text{DIAG}}$ plyne z Postovy věty, jeho částečná rozhodnutelnost pak z existence univerzálního Turingova stroje, neboť $\langle M\rangle \in \overline{\text{DIAG}}$, právě když $\langle M\rangle \in L(M)$, tedy právě když $\mathcal{U}(\langle M\rangle, \langle M\rangle)$ přijme. Věta 6.1.5 □

S pomocí diagonálního jazyka již snadno ukážeme, že univerzální jazyk je nerozhodnutelný.

Věta 6.3.2 *Univerzální jazyk* L_u *je algoritmicky nerozhodnutelný.*

Důkaz: Pro každý Turingův stroj M platí, že

$$\langle M\rangle \in \overline{\text{DIAG}} \Leftrightarrow \langle M\rangle \in L(M) \Leftrightarrow \langle M, \langle M\rangle\rangle \in L_u, \tag{6.12}$$

kde první ekvivalence plyne z definice diagonálního jazyka v (6.9) a druhá ekvivalence z definice univerzálního jazyka v (5.4). Z toho plyne, že kdyby byl jazyk L_u rozhodnutelný, byl by rozhodnutelný i jazyk $\overline{\text{DIAG}}$, neboť k rozhodnutí $\overline{\text{DIAG}}$ by stačilo zavolat algoritmus, který rozhoduje L_u se vstupem $\langle M, \langle M\rangle\rangle$. Podle věty 6.3.1 však $\overline{\text{DIAG}}$ rozhodnutelný není, a tedy ani jazyk L_u není rozhodnutelný. □

Uvědomme si, že z Postovy věty (Věta 6.1.5) a toho, že L_u je částečně rozhodnutelný jazyk (Věta 5.2.9), který není rozhodnutelný (Věta 6.3.2), plyne, že doplněk univerzálního jazyka $\overline{L_u}$ není částečně rozhodnutelný jazyk.

6.4. Výčet slov jazyka

V této kapitole se podíváme na částečně rozhodnutelné jazyky z pohledu enumerace. Enumerací jazyka L zde myslíme to, že jsme algoritmicky schopni vypsat všechna slova z tohoto jazyka. Tuto vlastnost jazyka budeme formalizovat pomocí zvláštního Turingova stroje, kterému budeme říkat enumerátor.

Definice 6.4.1 (Enumerátor) Nechť $L \subseteq \Sigma^*$ je jazyk nad abecedou Σ. Turingův stroj E nazveme *enumerátorem* pro jazyk L, pokud splňuje následující vlastnosti:

1. Turingův stroj E ignoruje svůj vstup.
2. Během své práce vypisuje E na zvláštní výstupní pásku řetězce z jazyka L,
3. Každý řetězec $w \in L$ je někdy vypsán Turingovým strojem E. ◂

Pokud je L nekonečný jazyk, pak z principu své činnosti enumerátor E nikdy svou činnost neukončí. Ani pokud je L konečný jazyk, nemusí se E zastavit — buď dokola vypisuje již vypsaná slova z jazyka L, nebo po vypsání posledního slova z L vstoupí do nekonečného cyklu bez vypisování dalších slov. Enumerátor E může vypsat některá slova z jazyka L vícekrát, není ovšem těžké změnit jakýkoli enumerátor takovým způsobem, aby každé slovo vypsal nejvýš jednou. Stačí E změnit tak, aby se vždy před vypsáním slova w z L podíval, zda již bylo slovo w vypsáno či nikoli. Pokud již slovo w vypsáno bylo, E jej již podruhé vypisovat nemusí.

Příklad 6.4.2

Viz též (4.1) v příkladu 4.1.4.

Popišme například enumerátor pro jazyk palindromů

$$\mathrm{PAL} = \left\{ w = w^R \mid w \in \{\mathtt{a},\mathtt{b}\}^* \right\}, \tag{6.13}$$

Připomeňme si, že v příkladu 4.1.4 jsme zkonstruovali Turingův stroj M, který rozhoduje jazyk PAL. Nyní zkonstruujeme s pomocí tohoto Turingova stroje jednoduchý enumerátor pro tento jazyk.

Algoritmus enumerátoru E pro jazyk PAL

```
1: for all w ∈ Σ* do                          ▷ V shortlex uspořádání.
2:     if w = w^R then
3:         print w
4:     end if
5: end for
```

Charakterizaci částečně rozhodnutelných jazyků pomocí enumerátorů zachycuje následující věta.

Věta 6.4.3 (Vyčíslitelnost částečně rozhodnutelných jazyků) *Jazyk L je částečně rozhodnutelný, právě když existuje enumerátor E pro L.*

Důkaz: Předpokládejme nejprve, že L je částečně rozhodnutelný jazyk. Dle bodu (iv) ve větě 6.1.1 existuje tedy rozhodnutelný jazyk B splňující

$$L = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)[\langle x, y\rangle \in B]\}\,.$$

Na základě toho můžeme sestavit pro jazyk L enumerátor E, který bude pracovat dle následujícího algoritmu:

Algoritmus enumerátoru E pro jazyk L

Vstup: Vstup je ignorován
Výstup: Vypisuje postupně prvky jazyka L
1: **for all** $w \in \Sigma^*$ **do** ▷ V shortlex uspořádání.
2: **if** w kóduje dvojici $\langle x, y\rangle \in B$ **then**
3: **print** x
4: **end if**
5: **end for**

Předpokládejme na druhou stranu, že E je enumerátor pro jazyk L a popišme Turingův stroj M, který bude přijímat L, tedy $L = L(M)$. Tento stroj bude pracovat dle následujícího algoritmu:

Algoritmus Turingova stroje M, přijímajícího L

Vstup: $x \in \Sigma^*$
Výstup: Algoritmus přijme, právě když $x \in L$.
1: Pusť enumerátor E a průběžně čti jeho výstupní pásku.
2: **if** Enumerátor E vypsal x **then**
3: **accept** ▷ Současně je pochopitelně zastavena činnost E.
4: **end if**

Dle definice enumerátoru $M(x)$ přijme, právě když řetězec x je někdy enumerátorem vypsán, tedy právě když $x \in L$. Jinými slovy $L(M) = L$. □

V případě rozhodnutelného jazyka L jsme schopni navíc sestrojit enumerátor, který vypisuje slova tohoto jazyka systematicky.

Věta 6.4.4 (Vyčíslitelnost rozhodnutelných jazyků) *Jazyk $L \subseteq \Sigma^*$ je rozhodnutelný, právě když existuje enumerátor E, který vypisuje slova L v pořadí daném shortlex uspořádáním.* *Definice 3.4.1*

Důkaz: Předpokládejme nejprve, že jazyk L je rozhodnutelný. Potom následující algoritmus popisuje práci enumerátoru E, který vypisuje slova L v pořadí daném shortlex uspořádáním:

Algoritmus enumerátoru E pro jazyk L

Vstup: Žádný.
Výstup: Slova z L v pořadí daném shortlex uspořádáním.

1: **for all** $w \in \Sigma^*$ **do** ▷ V pořadí daném shortlex uspořádáním.
2: **if** $w \in L$ **then**
3: **print** x
4: **end if**
5: **end for**

Předpokládejme na druhou stranu, že E je enumerátor, který vypisuje slova v pořadí daném shortlex uspořádáním. Pokud je jazyk L konečný, pak je jistě rozhodnutelný (dokonce konečným automatem), nadále budeme tedy předpokládat, že L je nekonečný jazyk. Nyní můžeme popsat algoritmus Turingova stroje M, který rozhoduje L, tj. přijímá L a navíc se jeho výpočet zastaví s každým vstupem.

Algoritmus Turingova stroje M rozhodujícího L

Vstup: Řetězec $x \in \Sigma^*$
Výstup: M přijme, pokud $x \in L$, jinak odmítne.
1: Pusť enumerátor E a průběžně čti slova, která vypisuje na svou výstupní pásku.
2: **if** E vypsal x **then**
3: **accept**
4: **else if** E vypsal slovo $y > x$ **then**
5: **reject**
6: **end if**

Z vlastností shortlex uspořádání vyplývá, že všechny řetězce y, které jsou delší než x, jsou větší než y v shortlex uspořádání (tedy $y > x$). Vzhledem k tomu, že abeceda Σ je konečná, L obsahuje jen konečně mnoho řetězců, které jsou v shortlex uspořádání menší než x. Protože jazyk L je sám nekonečný, musí obsahovat řetězec $y > x$. Řetězec y je někdy vypsán enumerátorem E. Připomeňme si, že E vypisuje řetězce z L v pořadí daném shortlex uspořádáním, mohou nastat tedy následující dva případy.

(i) Buď $x \in L$, v tom případě musel E vypsat řetězec x před řetězcem y a řetězec x je přijat v kroku 3 dřív, než je vůbec vypsán jakýkoli řetězec $y > x$.

(ii) Nebo $x \notin L$, v tom případě E řetězec x vůbec nevypíše. Vypíše ovšem někdy řetězec $y \in L$, který je větší než x v shortlex uspořádání. V tom okamžiku je už ovšem jasné, že enumerátor E nikdy x nevypíše a v kroku 5 může tedy Turingův stroj M odmítnout.

Turingův stroj M tedy skutečně rozhoduje jazyk L. □

6.5. Cvičení

Diagonalizace a nerozhodnutelné problémy

1. Pomocí diagonalizačního argumentu ukažte, že následující jazyky nejsou rekurzivně spočetné:

$$\begin{aligned} L_1 &= \{w_i \mid w_{2i} \notin L(M_i)\} \\ L_2 &= \{w_i \mid w_i \notin L(M_{2i})\} \end{aligned}$$

   Nápověda: *Při důkazu si nevystačíte s diagonálou matice, pomocí níž je definován jazyk* DIAG, *hledejte jinou nekonečnou množinu prvků této matice, která by mohla posloužit podobně jako diagonála v případě* DIAG. *U jazyka L_2 je třeba využít vlastností kódování, které jsme použili pro Turingovy stroje, rozmyslete si, že ke každému TS M existuje (v našem kódování) ekvivalentní TS, který má sudé Gödelovo číslo*

2. Definujme problém použití stavu následovně:

Problém 6.5.1: Použití stavu TS

Instance:	Kód Turingova stroje (zapsaný binárně) x, vstupní řetězec $y \in \{0,1\}^*$ a číslo stavu q.
Otázka:	Použije Turingův stroj M_x při výpočtu nad y stav q?

   Ukažte, že problém Použití stavu TS je algoritmicky nerozhodnutelný (tj. odpovídající jazyk není rekurzivní).

7. Převoditelnost a úplnost

Z kapitoly 6.3 víme, že jazyk univerzálního stroje, čili univerzální jazyk L_u je algoritmicky nerozhodnutelný. Důkaz tohoto faktu jsme provedli diagonalizací. Naším cílem je nyní využít nerozhodnutelnosti univerzálního jazyka k důkazům nerozhodnutelnosti dalších jazyků a problémů. Postup důkazu pomocí převoditelnosti si nejprve ukážeme na problému zastavení.

7.1. Problém zastavení

Uvažme následující problém.

Problém 7.1.1: Zastavení

Instance: Turingův stroj M a řetězec $x \in \Sigma^*$.

Otázka: Platí $M(x)\downarrow$, čili zastaví se výpočet Turingova stroje M nad vstupem x?

Naším cílem je ukázat, že tento problém je algoritmicky nerozhodnutelný. Mohli bychom postupovat obdobně jako při důkazu nerozhodnutelnosti univerzálního jazyka, a tedy problému Přijetí vstupu, v kapitole 6.3. Upravit diagonalizační argument použitý v důkazu věty 6.3.2 by nebylo příliš složité. My však budeme postupovat jinak. Využijeme toho, že problém Zastavení se od problému Přijetí vstupu příliš neliší. Skutečně jediný rozdíl je, zda nás zajímá přijetí či zastavení, ovšem kdybychom definovali přijetí vstupu Turingovým strojem s pomocí zastavení a nikoli přijímajícího stavu, tento rozdíl by zmizel. Úvaha, jež nás vede k důkazu nerozhodnutelnosti problému Zastavení je následující: Kdyby byl problém Zastavení algoritmicky rozhodnutelný, znamenalo by to, že problém Přijetí vstupu by rovněž musel být rozhodnutelný. Víme ovšem, že tak tomu není. Problém Zastavení formalizujeme jako jazyk

Problém 5.2.8

$$\text{HALT} = \{\langle M, x\rangle \mid M(x)\downarrow\}.$$

Věta 7.1.2 *Jazyk* HALT *je algoritmicky nerozhodnutelný.*

Důkaz: Předpokládejme pro spor, že HALT je algoritmicky rozhodnutelný jazyk, máme tedy Turingův H, který se vždy zastaví a HALT $= L(H)$. Na základě tohoto Turingova stroje nyní sestavíme Turingův stroj V, který bude rozhodovat univerzální jazyk L_u:

Výpočet V se vstupem $\langle M, x\rangle$

1: Uprav M na Turingův stroj M', který se místo odmítnutí vždy zacyklí.
2: Pusť $H(\langle M', x\rangle)$.

Turingův stroj M' se od M liší jen velmi málo. Je-li $M = (Q, \Sigma, \delta, q_0, F)$, pak $M' = (Q, \Sigma, \delta', q_0, F)$, kde pro $q \in Q$ a $a \in \Sigma$ definujeme následujícím způsobem:

$$\delta'(q,a) = \begin{cases} \delta(q,a) & \delta(q,a) \neq \perp \text{ nebo } q \in F \\ (q,a,N) & \text{jinak, tj. } \delta(q,a) = \perp \text{ a } q \notin F \end{cases}$$

Jde tedy opravdu jen o to, že v situaci, kdy by M odmítl tedy skončil výpočet ve stavu, který není přijímající, přejde M' místo toho do nekonečné smyčky. Uvažme nyní, že

$$\langle M, x\rangle \in L_u \Leftrightarrow x \in L(M) \Leftrightarrow x \in L(M') \Leftrightarrow M'(x)\downarrow \Leftrightarrow \langle M', x\rangle \in \text{HALT}. \tag{7.1}$$

Vskutku, pokud $\langle M, x\rangle \in L_u$, potom $M(x)$ přijme, tedy i $M'(x)$ přijme a tím i $M'(x)\downarrow$. Pokud na druhou stranu $\langle M, x\rangle \notin L_u$, potom buď $M(x)\uparrow$ nebo $M(x)$ odmítne, v obou případech ovšem $M'(x)\uparrow$. Volání $H(\langle M', x\rangle)$ přijme, právě když $\langle M', x\rangle \in \text{HALT} = L(H)$, a tedy právě když $\langle M, x\rangle \in L_u$ (dle (7.1)). Turingův stroj V tedy přijímá jazyk L_u a vždy se zastaví. To je ovšem v rozporu s nerozhodnutelností L_u, již jsme ukázali ve větě 6.3.2.□

7.2. Definice převoditelnosti

Převoditelnost nám nabízí obecný nástroj, který umožňuje s pomocí jednoho nerozhodnutelného problému ukázat nerozhodnutelnost jiného problému.

Definice 7.2.1 (*m*-převoditelnost) Nechť A a B jsou dva jazyky nad abecedou Σ. Řekneme, že jazyk A je m-převoditelný na jazyk B, pokud existuje totální algoritmicky vyčíslitelná funkce $f : \Sigma^* \to \Sigma^*$, pro kterou platí, že

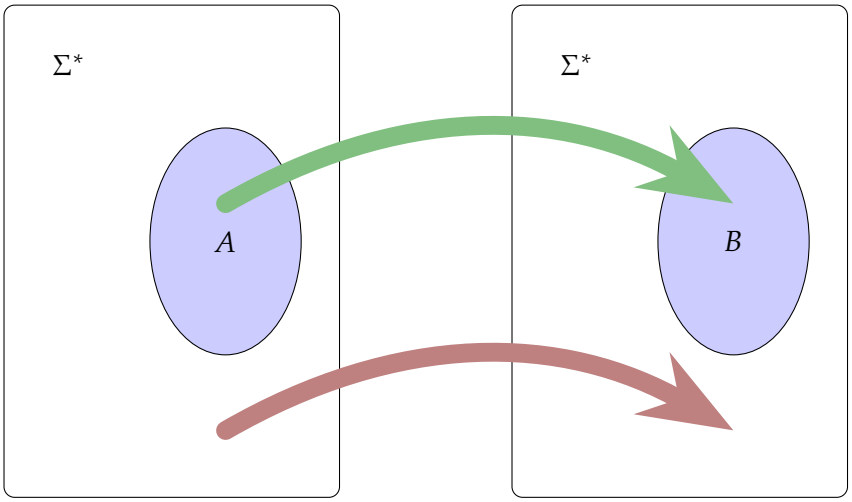
$$(\forall x \in \Sigma^*)[x \in A \Leftrightarrow f(x) \in B]. \tag{7.2}$$

Tento fakt označíme pomocí $A \leq_m B$. ◂

Funkce f, která převádí jazyk A na jazyk B tedy zobrazuje prvky z A na prvky B a prvky mimo A na prvky mimo B. Toto je naznačeno na obrázku 7.1.

Poznámka 7.2.2 (1-převoditelnost) *Znak m v pojmu m-převoditelnosti se dá považovat za zkratku* many to one reducibility, *nebo* mapping reducibility. *První z těchto výkladů pochází z toho, že funkce f není nutně prostá, může tedy dojít k tomu, že se několik řetězců zobrazí funkcí f na jeden řetězec. Poznamenejme, že kromě m-převoditelnosti se často zavádí i* 1*-převoditelnost, jež se od m-převoditelnosti liší tím, že navíc vyžaduje, aby funkce f byla prostá* (*tedy* one to one). *Pro naše účely však* 1*-převoditelnost nepřináší nic zajímavého a vystačíme si s m-převoditelností.*

Poznámka 7.2.3 (Turingovská převoditelnost) *Pomocí převoditelnosti se mimo jiné snažíme zachytit následující intuici:*



Obrázek 7.1.: Princip převodu problému A na problém B pomocí funkce f. Prvky z A jsou mapovány na prvky z B, zatímco prvky mimo A jsou mapovány na prvky mimo B.

Je-li problém A převoditelný na problém B a B je algoritmicky řešitelný, pak je algoritmicky řešitelný i problém A.

Uvědomme si ovšem, že m-převoditelnost je mnohem restriktivnější. Z faktu, že $A \leq_m B$ víme, že pokud máme k dispozici algoritmus rozhodující B, pak rozhodnutí, zda $x \in A$ dokážeme učinit výpočtem hodnoty $f(x)$ (kde f je funkce převádějící A na B) a dotazem, zda $f(x) \in B$, přičemž odpověď na tento dotaz dává přímo odpověď na původní otázku, zda $x \in A$. Naopak, pokud chceme ukázat $A \leq_m B$, nestačí popsat algoritmus rozhodující A, který může volat rozhodovací algoritmus pro B libovolně. Jsme opět omezeni tím, že je třeba popsat algoritmicky vyčíslitelnou funkci f tak, aby pro rozhodnutí, zda $x \in A$, stačilo rozhodnout, zda $f(x) \in B$.

Výše zmíněné intuici odpovídá turingovská převoditelnost, kterou si (ne zcela formálně) zavedeme. Řekneme, že jazyk A je turingovsky převoditelný na jazyk B, *pokud lze popsat algoritmus rozhodující A s tím, že může během výpočtu pokládat dotazy na to, zda $y \in B$ pro libovolný počet řetězců y. Tento fakt označíme pomocí $A \leq_T B$.*

Zřejmě platí, že pokud je $A \leq_m B$, pak i $A \leq_T B$, ale naopak to platit nemusí. Například $\overline{\text{HALT}} \leq_T \text{HALT}$, ovšem $\overline{\text{HALT}} \not\leq_m \text{HALT}$, neboť HALT je částečně rozhodnutelný jazyk, zatímco $\overline{\text{HALT}}$ nikoli. Pomocí m-převoditelnosti dokážeme tedy rozlišit mezi jazykem HALT a jeho doplňkem $\overline{\text{HALT}}$, pomocí turingovské převoditelnosti nikoli. I proto dále budeme využívat m-převoditelnost.

Připomeňme si, že v kapitole 2.2 jsme si ukázali převod problému Helloworld na problém Volání funkce `foo`, přičemž konstrukce splňovala požadavky definice 7.2.1, ukázali jsme tedy, že problém Helloworld je m-převoditelný na problém Volání funkce

foo.

V důkazu věty 7.1.2 jsme ve skutečnosti ukázali, že jazyk L_u je převoditelný na jazyk HALT. Zformulujme si zde nyní toto tvrzení jako důsledek.

Důsledek 7.2.4 *Platí, že* $L_u \leq_m$ HALT.

Důkaz: Připomeňme si, že v důkazu věty 7.1.2 jsme popsali konstrukci Turingova stroje M' z Turingova stroje M, kde M' se od M lišil jen v tom, že na každém vstupu x, jejž by M odmítl, výpočet M' vůbec neskončil. Jinak byl M' shodný s M. Protože úprava M na M' je velmi jednoduchá a bylo by lze ji provést algoritmicky, je funkce $f : \Sigma^* \to \Sigma^*$ definovaná jako $f(\langle M, x\rangle) = \langle M', x\rangle$ jistě algoritmicky vyčíslitelná. Dle ekvivalence (7.1) navíc dostáváme, že

$$\langle M, x\rangle \in L_u \Leftrightarrow f(\langle M, x\rangle) = \langle M', x\rangle \in \text{HALT}\,. \tag{7.3}$$

Funkce f je tedy převodní funkcí, jež ukazuje, že $L_u \leq_m$ HALT. □

I při důkazu nerozhodnutelnosti univerzálního jazyka ve větě 6.3.2 jsme využili převoditelnosti. Při důkazu této věty jsme totiž vlastně ukázali, že $\overline{\text{DIAG}} \leq_m L_u$. Platí však i opačný směr, tedy $L_u \leq_m \overline{\text{DIAG}}$.

Diagonální jazyk DIAG byl zaveden v (6.3) a jeho doplněk $\overline{\text{DIAG}}$ byl zaveden v (6.9).

Věta 7.2.5 *Platí, že*

(*i*) $\overline{\text{DIAG}} \leq_m L_u$ *a současně*

(*ii*) $L_u \leq_m \overline{\text{DIAG}}$.

Důkaz: Připomeňme, že dle (6.9) je $\overline{\text{DIAG}} = \{\langle M\rangle \mid \langle M\rangle \in L(M)\}$.

(i) Pro každý Turingův stroj M platí, že

$$\langle M\rangle \in \text{DIAG} \Leftrightarrow \langle M\rangle \in L(M) \Leftrightarrow \langle M, \langle M\rangle\rangle \in L_u,$$

kde první ekvivalence platí dle (6.9) a druhá dle definice univerzálního jazyka v (5.4). Funkce f definovaná jako $f(\langle M\rangle) = \langle M, \langle M\rangle\rangle$ je jistě totální algoritmicky vyčíslitelná a ukazuje, že $\overline{\text{DIAG}} \leq_m L_u$.

(ii) Tento směr je o něco obtížnější, musíme popsat totální algoritmicky vyčíslitelnou funkci f, jež splňuje, že $f(\langle M, x\rangle) = \langle M'\rangle$, kde

$$x \in L(M) \Leftrightarrow \langle M'\rangle \in L(M'). \tag{7.4}$$

Popíšeme, jak bude probíhat výpočet M' nad daným vstupem.

Algoritmus výpočtu Turingova stroje M' nad vstupem y

1: Smaž vstup y
2: Zapiš na pásku slovo x
3: Proveď výpočet Turingova stroje M nad vstupem x.

4: **if** M přijal **then**
5: **accept**
6: **else**
7: **reject**
8: **end if**

Všimněme si, že Turingův stroj M' ignoruje zcela svůj vstup y. Místo toho je výpočet $M'(y)$ ekvivalentní výpočtu $M(x)$. Funkce f zkonstruuje kód stroje M' na základě kódu stroje M a řetězce x. Tento postup je celkem přímočarý, jedná se vlastně jen o úpravu přechodové funkce M tak, aby před samotným výpočtem M došlo k přepsání vstupu řetězcem x. Funkce f je tedy jistě algoritmicky vyčíslitelná a také je jistě totální. Všimněme si, že

$$L(M') = \begin{cases} \Sigma^* & \text{pokud } x \in L(M) \\ \varnothing & \text{jinak, tedy pokud } x \notin L(M). \end{cases}$$

Platí tedy, že

$$\begin{array}{llllll} \langle M,x\rangle \in L_u \Rightarrow & x \in L(M) \Rightarrow & L(M') = \Sigma^* \Rightarrow & \langle M'\rangle \in L(M') \Rightarrow & \langle M'\rangle \in \overline{\mathrm{DIAG}} \\ \langle M,x\rangle \notin L_u \Rightarrow & x \notin L(M) \Rightarrow & L(M') = \varnothing \Rightarrow & \langle M'\rangle \notin L(M') \Rightarrow & \langle M'\rangle \notin \overline{\mathrm{DIAG}} \end{array}$$

Dohromady dostáváme, že

$$\langle M,x\rangle \in L_u \Leftrightarrow f(\langle M,x\rangle) = \langle M'\rangle \in \overline{\mathrm{DIAG}},$$

a tedy $L_u \leq_m \overline{\mathrm{DIAG}}$. □

Podívejme se nyní na některé základní vlastnosti m-převoditelnosti. Ukážeme si, že jako relace je m-převoditelnost reflexivní a tranzitivní, jde tedy o kvaziuspořádání.

Lemma 7.2.6 *Relace $\leq_m$ je reflexivní a tranzitivní.*

Důkaz: Reflexivita vyplývá z toho, že funkce identity, tj. funkce $f(x) = x$ je totální algoritmicky vyčíslitelná funkce. Díky tomu platí $A \leq_m A$ pro každý jazyk A.

Pro důkaz tranzitivity uvažme tři jazyky A, B, C a předpokládejme, že $A \leq_m B$ a $B \leq_m C$. Ukážeme, že potom $A \leq_m C$. Je-li $A \leq_m B$ na základě funkce f a $B \leq_m C$ na základě funkce g, pak platí pro každý řetězec $x \in \Sigma^*$

$$x \in A \Leftrightarrow f(x) \in B \Leftrightarrow g(f(x)) \in C.$$

To znamená, že definujeme-li funkci h jako složení funkce g s funkcí f, tedy $h(x) = g(f(x))$ pro každý řetězec x, pak tato funkce je jistě totální algoritmicky vyčíslitelná funkce, neboť f a g jsou obě totální algoritmicky vyčíslitelné funkce. Platí tedy, že $A \leq_m C$ na základě funkce h. □

Nás zajímá převoditelnost zejména jako nástroj pro důkazy nerozhodnutelnosti problémů a jazyků. Intuitivně pokud $A \leq_m B$, znamená to, že jsme schopni rozhodnout jazyk A pomocí B. Tuto intuici formalizujeme v následujícím tvrzení.A

Věta 7.2.7 *Předpokládejme, že A a B jsou jazyky, pro něž platí, že $A \leq_m B$. Potom platí, že*

(*i*) *je-li B rozhodnutelný jazyk, potom je rozhodnutelný jazyk i A a*

(*ii*) *je-li B částečně rozhodnutelný jazyk, potom je částečně rozhodnutelný jazyk i A.*

Důkaz: Označme převodní funkci ukazující, že $A \leq_m B$, jako f a uvažme následující algoritmus:

Algoritmus rozhodující A pomocí B

Vstup: Řetězec x
1: Spočti $y = f(x)$
2: **if** $y \in B$ **then**
3: **accept**
4: **else**
5: **reject**
6: **end if**

Vzhledem k tomu, že f je totální algoritmicky vyčíslitelná funkce, první krok lze vždy provést algoritmicky. Proveditelnost algoritmu tedy závisí na kroku 2.

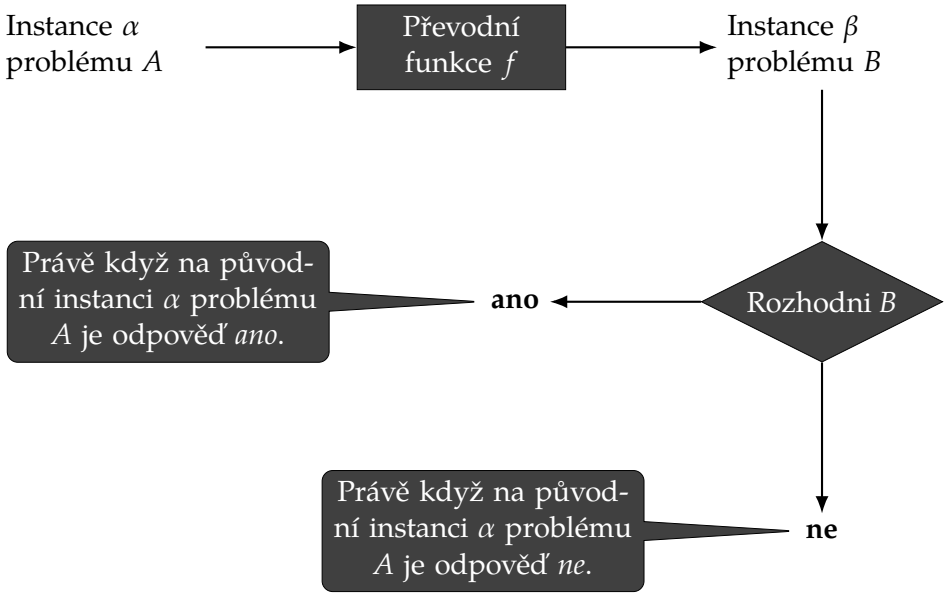
(i) Pokud je jazyk B rozhodnutelný, existuje Turingův stroj, který se vždy zastaví a přijímá jazyk B. To znamená, že uvedený algoritmus lze implementovat takovým způsobem, aby se vždy zastavil a rozhodoval A. Jazyk A je tedy rozhodnutelný.

(ii) Pokud je jazyk B částečně rozhodnutelný, existuje Turingův stroj, který přijímá jazyk B. To znamená, že uvedený algoritmus lze implementovat takovým způsobem, aby přijímal A, ale nemusí se zastavit na vstupech x, pro které platí $x \notin A$, a tedy $y = f(x) \notin B$. Jazyk A je tedy částečně rozhodnutelný. □

Algoritmus v důkazu věty 7.2.7 tedy pracuje dle schématu, jež je naznačeno na obrázku 7.2.

My budeme převoditelnost používat v opačném směru. Uvažme jazyky A a B, pro něž platí, že $A \leq_m B$. Z věty 7.2.7 vyplývá, že není-li A (částečně) rozhodnutelný jazyk, pak ani B není (částečně) rozhodnutelný. Víme například, že $\overline{\text{DIAG}}$, L_u ani HALT nejsou rozhodnutelné jazyky, byť jde o jazyky částečně rozhodnutelné. To znamená, že pokud pro nějaký jazyk A platí například $L_u \leq_m A$, pak A není rozhodnutelný a pokud o jiném jazyku B platí, že DIAG $\leq_m B$, pak B není částečně rozhodnutelný (dle věty 6.3.1 není totiž DIAG částečně rozhodnutelný). To je také způsob, který dále zvolíme pro dokazování toho, že nějaký problém není rozhodnutelný.

Věta 6.3.1, věta 6.3.2 a věta 7.1.2



Obrázek 7.2.: Princip převodu problému A na problém B pomocí funkce f.

Příklad 7.2.8: Nerozhodnutelnost neprázdnosti jazyka

Připomeňme si jazyk

$$\text{NONEMPTY} = \{\langle M \rangle \mid L(M) \neq \varnothing\}$$

zavedený v příkladu 6.1.3. Ukážeme, že $L_u \leq_m$ NONEMPTY. Z toho tedy plyne, že NONEMPTY není rozhodnutelný jazyk. Musíme popsat totální částečně rekurzivní funkci $f : \Sigma^* \to \Sigma^*$, pro kterou platí ekvivalence (7.2), tj. pro každou dvojici Turingova stroje M a řetězce x platí, že

$$\langle M, x \rangle \in L_u \Leftrightarrow f(\langle M, x \rangle) \in \text{NONEMPTY}. \tag{7.5}$$

Popíšeme, jak bude probíhat výpočet Turingova stroje $M' = f(\langle M, x \rangle)$ se vstupem $y \in \Sigma^*$.

Výpočet stroje M' se vstupem y

1: Vymaž pásku (tj. y)
2: Zapiš na pásku řetězec x
3: Pokračuj výpočtem $M(x)$ ▷ Zastavení, přijetí, či odmítnutí se řídí výsledkem $M(x)$.

Přechodová funkce M' se tedy od přechodové instrukce M liší jen tím, že jsou v ní navíc instrukce pro provedení prvních dvou kroků algoritmu, tedy nahrazení vstupu

y řetězcem x. Přidání těchto instrukcí je úkolem funkce f, která z dvojice $\langle M, x\rangle$ vytvoří kód $\langle M'\rangle$. Je potřeba si uvědomit, že funkce f sama nesimuluje výpočet $M(x)$, a její výpočet není tedy závislý na tom, zda se výpočet $M(x)$ zastaví. Podívejme se nyní na jazyk přijímaný strojem M'. Všimněme si, že výpočet $M'(y)$ na vstupu y vůbec nezávisí. Pokud je $x \in L(M)$, pak $M'(y)$ přijme každý řetězec y, pokud naopak $x \notin L(M)$, pak $M'(y)$ žádný řetězec y nepřijme. Toto lze shrnout následujícím výrazem:

$$L(M') = \begin{cases} \Sigma^* & x \in L(M) \\ \varnothing & x \notin L(M) \end{cases}$$

Z toho již snadno dostaneme platnost ekvivalence (7.5), neboť

$$\begin{aligned} \langle M, x\rangle \in L_u \Rightarrow \quad x \in L(M) \Rightarrow \quad L(M') = \Sigma^* \Rightarrow \quad f(\langle M, x\rangle) = \langle M'\rangle \in \text{NONEMPTY} \\ \langle M, x\rangle \notin L_u \Rightarrow \quad x \notin L(M) \Rightarrow \quad L(M') = \varnothing \Rightarrow \quad f(\langle M, x\rangle) = \langle M'\rangle \notin \text{NONEMPTY} \end{aligned}$$

Z toho tedy vyplývá, že $L_u \leq_m$ NONEMPTY, a jazyk NONEMPTY je tedy nerozhodnutelný. V příkladu 6.1.3 jsme viděli, že NONEMPTY je částečně rozhodnutelný jazyk a z Postovy věty tedy plyne, že $\overline{\text{NONEMPTY}}$ není částečně rozhodnutelný jazyk. Věta 6.1.5

7.3. Úplné problémy

Jak jsme již zmínili, dle lemmatu 7.2.6 je m-převoditelnost kvaziuspořádáním (tedy tranzitivní a reflexivní relací). Zřejmě ne všechny jazyky jsou na sebe vzájemně převoditelné (například $\overline{\text{HALT}}$ není převoditelný na HALT). Proto se může zdát i trochu překvapivé, že existují jazyky, na které jsou převoditelné všechny ostatní částečně rozhodnutelné jazyky. Takovým jazykem ja například univerzální jazyk L_u nebo problém zastavení HALT. Budeme říkat, že takové problémy jsou m-úplné.

Definice 7.3.1 (m-úplnost) Řekneme, že jazyk $A \subseteq \Sigma^*$ je *m-úplný*, pokud je

1. A částečně rozhodnutelný jazyk a
2. pro každý částečně rozhodnutelný jazyk B platí, že $B \leq_m A$. ◂

Řada z nerozhodnutelných problémů, s nimiž jsme se setkali, je m-úplná.

Věta 7.3.2 *Jazyky L_u,* HALT *a* $\overline{\text{DIAG}}$ *jsou m-úplné.*

Důkaz: Víme již, že tyto jazyky jsou částečně rozhodnutelné díky existenci univerzálního Turingova stroje. Nechť A je libovolný částečně rozhodnutelný jazyk, který je přijímaný Turingovým strojem M_A. Pak platí, že $x \in A$, právě když $\langle M_A, x\rangle \in L_u$. Funkce f definovaná jako $f(x) = \langle M_A, x\rangle$ je jistě algoritmicky vyčíslitelná a ukazuje tedy, že $A \leq_m L_u$. Jazyk L_u je tedy m-úplný. Z věty 7.2.5 plyne, že $\overline{\text{DIAG}}$ je m-úplný, podobně m-úplnost HALT vyplývá z důsledku 7.2.4. □

Jako další příklad m-úplného problému nám poslouží otázka neprázdnosti jazyka přijímaného daným Turingovým strojem:

Příklad 7.3.3: Úplnost neprázdnosti jazyka

Vzpomeňme si, že v příkladu 7.2.8 jsme ukázali, že $L_u \leq_m$ NONEMPTY a v příkladu 6.1.3 jsme ukázali, že je jazyk NONEMPTY částečně rozhodnutelný. Dohromady dostáváme, že je tento jazyk m-úplný.

7.4. Riceova věta

V této kapitole zformulujeme a ukážeme Riceovu větu. Tato věta říká, že otázka, zda jazyk přijímaný daným Turingovým strojem (algoritmem) splňuje danou vlastnost, je rozhodnutelná, jen pokud odpověď na tuto otázku je buď triviálně ano (je tedy splněna každým jazykem), nebo triviálně ne (není tedy splněna žádným jazykem). V tomto kontextu můžeme mluvit stejně dobře o jazyku přijímaném Turingovým strojem (algoritmem) jako o funkci vyčíslované Turingovým strojem. Pro úplnost si postupně zformulujeme Riceovu větu jak ve verzi pro jazyky, tak ve verzi pro funkce.

Viz definice Turingovsky vyčíslitelné funkce v definici 4.1.8

Začněme verzí pro jazyky.

Věta 7.4.1 (Riceova věta pro jazyky) *Nechť $\mathcal{C}$ je třída částečně rozhodnutelných jazyků. Potom jazyk*

$$L_{\mathcal{C}} = \{\langle M \rangle \mid L(M) \in \mathcal{C}\}$$

je rozhodnutelný, právě když je třída $\mathcal{C}$ triviální, tj. buď je prázdná, nebo obsahuje všechny částečně rozhodnutelné jazyky.

Důkaz: Budeme přepokládat, že třída $\mathcal{C}$ obsahuje jazyky nad abecedou Σ, kde Σ je současně abeceda použitá pro kódování Turingových strojů a jiných objektů. Je-li třída $\mathcal{C}$ prázdná, pak $L_{\mathcal{C}} = \emptyset$, pokud naopak třída $\mathcal{C}$ obsahuje všechny částečně rozhodnutelné jazyky, pak $L_{\mathcal{C}} = \Sigma^*$. V obou případech je jistě $L_{\mathcal{C}}$ rozhodnutelný jazyk.

Například $\Sigma = \{0,1\}$ nebo $\Sigma = \Gamma$, použijeme-li kódování popsané v kapitole 5.2.1. Připomeňme si též, že dle konvence 5.2.3 každý řetězec odpovídá nějakému Turingovu stroji.

Předpokládejme nyní, že $\mathcal{C}$ je netriviální třída částečně rozhodnutelných jazyků, tedy je neprázdná, ale neobsahuje ani všechny částečně rozhodnutelné jazyky. Ukážeme, že v tom případě platí

$$L_u \leq_m L_{\mathcal{C}} \quad \text{nebo} \tag{7.6}$$
$$L_u \leq_m \overline{L_{\mathcal{C}}}. \tag{7.7}$$

V obou případech dostaneme, že $L_{\mathcal{C}}$ není rozhodnutelný jazyk. To, zda ukážeme převod (7.6) nebo (7.7), závisí na tom, kam patří prázdný jazyk. Předpokládejme nejprve, že prázdný jazyk do třídy $\mathcal{C}$ nepatří, tedy $\emptyset \notin \mathcal{C}$. V tom případě ukážeme, že platí (7.6), tedy $L_u \leq_m L_{\mathcal{C}}$. Nechť L je navíc libovolný částečně rozhodnutelný jazyk z třídy $\mathcal{C}$. Speciálně $L \neq \emptyset$. Předpokládejme navíc, že N je Turingův stroj, který přijímá L, tj. $L = L(N)$. Převodní funkce f se vstupem $\langle M, x \rangle$ vrátí kód Turingova stroje M', který se vstupem y pracuje následujícím způsobem:

Výpočet stroje M', kde $\langle M' \rangle = f(\langle M, x \rangle)$ se vstupem y

Pusť $M(x)$.
if $M(x)$ odmítl **then**
    reject
end if
Pusť $N(y)$.
if $N(y)$ přijal **then**
    accept
else
    reject
end if

Kód Turingova stroje M' vznikne složením kódů Turingových strojů M a N a vstupu x. Funkce f je tedy jistě algoritmicky vyčíslitelná a navíc je i definovaná pro každý vstup. Stačí ukázat, že $\langle M, x \rangle \in L_u \Leftrightarrow \langle M' \rangle \in L_{\mathcal{C}}$, tím dostaneme, že $L_u \leq_m L_{\mathcal{C}}$.

Předpokládejme nejprve, že $\langle M, x \rangle \in L_u$, tedy že $x \in L(M)$. V tom případě výpočet $M(x)$ skončí a přijme, tedy dojde ke spuštění $N(y)$ pro každý vstup y. To znamená, že $L(M') = L(N) = L \in \mathcal{C}$.

Předpokládejme nyní, že $\langle M, x \rangle \notin L_u$, tedy $x \notin L(M)$. V tomto případě buď výpočet $M(x)$ vůbec neskončí, nebo nezávisle na vstupu y odmítne. Turingův stroj M' tedy rozhodně žádný vstup y nepřijme, a proto $L(M') = \emptyset$, podle předpokladu tedy $L(M') \notin \mathcal{C}$.

Případ, kdy prázdný jazyk patří do třídy $\mathcal{C}$, je symetrický. Stačí zaměnit role $\mathcal{C}$ doplňku $\mathcal{C}$, dostaneme tak převod $L_u \leq_m \overline{L_{\mathcal{C}}}$. □

Podobně můžeme zformulovat Riceovu větu i pro funkce.

Věta 7.4.2 (Riceova věta pro funkce) *Nechť $\mathcal{C}$ je třída algoritmicky vyčíslitelných funkcí. Potom jazyk*

$$L_{\mathcal{C}} = \{\langle M \rangle \mid f_M \in \mathcal{C}\}$$

je rozhodnutelný, právě když je třída $\mathcal{C}$ triviální, tj. buď je prázdná, nebo obsahuje všechny algoritmicky vyčíslitelné funkce.

Důkaz: Důkaz je zcela analogický důkazu verze Riceovy věty pro jazyky, tedy věty 7.4.1. Budeme předpokládat, že funkce v třídě $\mathcal{C}$ jsou typu $\Sigma^* \to \Sigma^*$, kde Σ je současně abeceda použitá pro kódování Turingových strojů a jiných objektů.

Pokud je třída $\mathcal{C}$ triviální, pak zřejmě buď $L_{\mathcal{C}} = \emptyset$, nebo $L_{\mathcal{C}} = \Sigma^*$. V obou případech jde o rozhodnutelný jazyk. Nechť g_0 označuje funkci, která není definovaná pro žádný vstup, tedy pro každý řetězec $y \in \Sigma^*$ platí $g_0(y)\uparrow$. Předpokládejme bez újmy na obecnosti, že $g_0 \notin \mathcal{C}$ (v opačném případě zaměníme role $\mathcal{C}$ a doplňku $\mathcal{C}$). Ukážeme, že potom $L_u \leq_m L_{\mathcal{C}}$, z čehož plyne, že $L_{\mathcal{C}}$ je nerozhodnutelný jazyk. Nechť g_1 je libovolná funkce, která naopak patří do $\mathcal{C}$ Funkce f, která převádí L_u na $L_{\mathcal{C}}$ se vstupem $\langle M, x \rangle$ vrátí kód

Turingova stroje M', který vyčísluje funkci $f_{M'}$, jež pro vstup $y \in \Sigma^*$ splňuje

$$f_{M'}(y) = \begin{cases} g_1(y) & x \in L(M) \\ \uparrow & x \notin L(M) \end{cases}$$

Uvědomme si, že kód Turingova stroje M' lze zkonstruovat jen se znalostí kódu M, vstupu x a kódu Turingova stroje, který vyčísluje funkci g_1. Zejména pak platí, že f je algoritmicky vyčíslitelná funkce.

Nechť nyní $\langle M, x\rangle \in L_u$, tedy $x \in L(M)$. Potom $f(\langle M, x\rangle) = \langle M'\rangle$, kde $f_{M'} \simeq g_1 \in \mathcal{C}$. Na druhou stranu pokud $\langle M, x\rangle \notin L_u$, tedy $x \notin L(M)$, pak funkce $f_{M'}$ není definovaná pro žádný vstup $y \in \Sigma^*$, a tedy $f_{M'} \simeq g_0 \notin \mathcal{C}$. Dohromady dostáváme, že $\langle M, x\rangle \in L_u$, právě když $f(\langle M, x\rangle) \in L_{\mathcal{C}}$. Z toho plyne, že $L_u \leq_m L_{\mathcal{C}}$ a $L_{\mathcal{C}}$ tedy není algoritmicky rozhodnutelný jazyk. □

Jak jsme zmínili v úvodu, Riceova věta ukazuje o určitém typu problémů, že nemohou být algoritmicky rozhodnutelné. Jde o problémy, kde vstupem je kód Turingova stroje (nebo obecněji program v jakémkoli programovacím jazyku, který je s Turingovými stroji ekvivalentní) a kde se ptáme, zda funkce nebo jazyk určená tímto Turingovým strojem splňuje danou netriviální vlastnost. Nejde tedy o otázky, odpověď na něž je závislá na konkrétním kódu (programu), ale naopak závisí na funkci daného kódu (programu). Například nerozhodnutelnost diagonálního jazyka DIAG = $\{\langle M\rangle \mid \langle M\rangle \notin L(M)\}$ zavedeného předpisem (6.3) nevyplývá z Riceovy věty, protože to, zda $\langle M\rangle \in L(M)$ podstatně závisí na konkrétním kódu $\langle M\rangle$. Nerozhodnutelnost řady jiných jazyků však z Riceovy věty plyne.

Důsledek 7.4.3 *Následující jazyky jsou nerozhodnutelné*

$$\begin{aligned} \text{NONEMPTY} &= \{\langle M\rangle \mid L(M) \neq \varnothing\} \\ \text{ALL} &= \{\langle M\rangle \mid L(M) = \Sigma^*\} \\ \text{Tot} &= \{\langle M\rangle \mid (\forall y \in \Sigma^*)[f_M(y)\downarrow]\} \\ \text{Fin} &= \{\langle M\rangle \mid L(M) \textit{ je konečný jazyk}\} \\ \text{Inf} &= \{\langle M\rangle \mid L(M) \textit{ je nekonečný jazyk}\} \\ \text{Cof} &= \{\langle M\rangle \mid \overline{L(M)} \textit{ je konečný jazyk}\} \\ \text{Regular} &= \{\langle M\rangle \mid L(M) \textit{ je regulární jazyk}\} \end{aligned}$$

Důkaz: Plyne přímo z věty 7.4.1. □

U všech těchto jazyků je vlastnost, která nás u daného Turingova stroje zajímá, vlastností jím vyčíslované funkce či jím přijímaného jazyka, jde tedy o funkční vlastnost. Všimněme si také, že Riceova věta nevylučuje částečnou rozhodnutelnost, například jazyk NONEMPTY je částečně rozhodnutelný, jak jsme si ukázali v příkladu 6.1.3.

7.5. Postův korespondenční problém

Problémy, o nichž jsme si dosud ukázali, že jsou nerozhodnutelné, nějakým způsobem souvisely s Turingovými stroji (potažmo algoritmy a programy obecně) a jejich výpočty. V této kapitole si ukážeme nerozhodnutelnost problému, jehož definice se na Turingův stroj nijak neodkazuje a pracuje pouze s řetězci.

Instancí Postova korespondenčního problému je množina dvojic řetězců, které si můžeme představit jako dominové kostky

$$\left[\begin{array}{c} t_i \\ \hdashline b_i \end{array}\right],$$

kde $t_i, b_i \in \Sigma^*$ pro $i = 1, \ldots, k$ a pevnou abecedu Σ. Od každého typu je k dispozici neomezené množství kostek, otázkou je, zda je možno vybrat párovací posloupnost kostek. *Párovací posloupností* je zde konečná neprázdná posloupnost $i_1, \ldots, i_l \in \{1, \ldots, k\}$, pro kterou platí, že srovnáme-li vybrané kostky vedle sebe, nahoře i dole dostaneme týž řetězec. Jinými slovy platí $t_{i_1} t_{i_2} \ldots t_{i_l} = b_{i_1} b_{i_2} \ldots b_{i_l}$. Formálně definujeme Postův korespondenční problém následujícím způsobem.

Problém 7.5.1: Postův korespondenční problém (PKP)

Instance: Množina „dominových kostek“ P:

$$P = \left\{ \left[\begin{array}{c} t_1 \\ \hdashline b_1 \end{array}\right], \left[\begin{array}{c} t_2 \\ \hdashline b_2 \end{array}\right], \ldots, \left[\begin{array}{c} t_k \\ \hdashline b_k \end{array}\right] \right\}$$

kde $t_1, \ldots, t_k, b_1, \ldots, b_k \in \Sigma^*$ jsou řetězce.

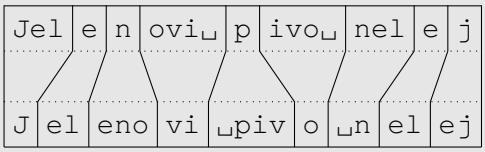
Otázka: Existuje *párovací posloupnost* indexů $i_1, i_2, \ldots, i_l \in \{1, \ldots, k\}$, kde $l \geq 1$ a $t_{i_1} t_{i_2} \ldots t_{i_l} = b_{i_1} b_{i_2} \ldots b_{i_l}$?

Příklad 7.5.2

Uvažme například množinu P, která obsahuje kostky

$$\left[\begin{array}{c} \texttt{Jel} \\ \hdashline \texttt{J} \end{array}\right] \left[\begin{array}{c} \texttt{e} \\ \hdashline \texttt{el} \end{array}\right] \left[\begin{array}{c} \texttt{n} \\ \hdashline \texttt{eno} \end{array}\right] \left[\begin{array}{c} \texttt{ovi␣} \\ \hdashline \texttt{vi} \end{array}\right] \left[\begin{array}{c} \texttt{p} \\ \hdashline \texttt{␣piv} \end{array}\right] \left[\begin{array}{c} \texttt{ivo␣} \\ \hdashline \texttt{o} \end{array}\right] \left[\begin{array}{c} \texttt{nel} \\ \hdashline \texttt{␣n} \end{array}\right] \left[\begin{array}{c} \texttt{j} \\ \hdashline \texttt{ej} \end{array}\right]$$

Z této množiny není těžké vybrat následující párovací posloupnost:



Všimněme si, že kostka $\left[\begin{smallmatrix} \texttt{e} \\ \texttt{el} \end{smallmatrix}\right]$ je v této posloupnosti použita dvakrát.

Naším cílem je nyní ukázat, že Postův korespondenční problém je algoritmicky neřešitelný. To ukážeme tím, že popíšeme převod z problému Přijetí vstupu (čili z univerzálního jazyka L_u) na PKP. Budeme postupovat ve dvou krocích, jako mezikrok použijeme trochu zjednodušenou verzi problému PKP, kde máme pevně danou první kostkou, kterou musí začínat hledaná párovací posloupnost.

Problém 7.5.3: Modifikovaný Postův korespondenční problém (MPKP)

Instance: Shodná s instancí Postova korespondenčního problému.

Otázka: Existuje párovací posloupnost, která začíná kostkou $\left[\begin{smallmatrix} t_1 \\ b_1 \end{smallmatrix}\right]$?

Problém 5.2.8

Nejprve ukážeme, že problém Přijetí vstupu (univerzální jazyk L_u) je převoditelný na problém MPKP, poté ukážeme, jak převést problém MPKP na problém PKP.

7.5.1. Převod problému Přijetí vstupu na problém MPKP

Připomeňme si, že instancí problému Přijetí vstupu je dvojice tvořená kódem Turingova stroje $\langle M \rangle$ a jeho vstupu $x \in \Sigma^*$, kde Σ je vstupní abeceda Turingova stroje M. Musíme popsat algoritmus, který převede tuto instanci na instanci problému MPKP, tedy množinu dominových kostek

$$P = \left\{ \left[\begin{smallmatrix} t_1 \\ b_1 \end{smallmatrix}\right], \left[\begin{smallmatrix} t_2 \\ b_2 \end{smallmatrix}\right], \ldots, \left[\begin{smallmatrix} t_k \\ b_k \end{smallmatrix}\right] \right\}$$

pro kterou platí následující ekvivalence:

$$x \in L(M) \Leftrightarrow \text{z } P \text{ lze vybrat párovací posloupnost, jež začíná kostkou } \left[\begin{smallmatrix} t_1 \\ b_1 \end{smallmatrix}\right] \tag{7.8}$$

Označme si Turingův stroj $M = (Q, \Sigma, \delta, q_0, F)$ a předpokládejme vstup $x = x_1 x_2 \dots x_n$ délky n. Zjednodušíme si poněkud situaci tím, že budeme o Turingovu stroji M předpokládat, že má jediný přijímající stav q_1, tedy $F = \{q_1\}$. Navíc předpokládáme, že přejde-li M do stavu q_1, jeho výpočet skončí, tedy předpokládáme, že $\delta(q_1, a) = \perp$ pro každý znak $a \in \Sigma$.

Podle definice výpočtu Turingova stroje platí, že $x \in L(M)$ právě když existuje posloupnost konfigurací $K_0, \dots, K_t$ Turingova stroje M, kde Definice 4.1.2

- K_0 je počáteční konfigurací Turingova stroje M při výpočtu nad vstupem x,
- K_t je přijímající konfigurací Turingova stroje M a
- pro každý index $i = 1, \dots, t$ platí, že konfigurace K_i vznikne z konfigurace K_{i-1} aplikací přechodové funkce δ.

Naším cílem bude vytvořit k Turingovu stroji M a vstupu x takovou sadu kostek, pro kterou bude platit, že párovací posloupnost odpovídá právě posloupnosti konfigurací určující přijímající výpočet. Konfiguraci stroje M zapíšeme jako posloupnost symbolů na pásce, přičemž před čtený symbol zapíšeme symbol reprezentující stav, v němž se Turingův stroj nachází. Pro zápis řetězců v instanci MPKP tedy budeme používat abecedu, která odpovídá sjednocení Q a Σ. Kromě toho budeme používat symbol $\#$ pro ohraničení zápisu jedné konfigurace. Například řetězec

$$\#\lambda\mathtt{a}\mathtt{a}q_7\mathtt{b}\mathtt{b}\#$$

popisuje konfiguraci, kde na pásce je zapsáno slovo `aabb`, Turingův stroj se nachází ve stavu q_7 a čte první ze dvou symbolů `b`. Součástí zápisu je v tomto příkladu i jedno prázdné políčko před prvním políčkem řetězce. Konfigurace vždy zachycuje konečný kus pásky, který je dost velký, aby zahrnul všechny neprázdné znaky, může ovšem zahrnovat i prázdná políčka v okolí.

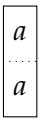
Myšlenka konstrukce je následující. Spodní řada bude o jednu konfiguraci napřed. První kostka bude dole obsahovat počáteční konfiguraci a nahoře jen oddělovač $\#$. Další kostky pak budou implementovat přechodovou funkci δ, kde horní řetězec bude odpovídat situaci před aplikací přechodové funkce a spodní řetězec situaci po aplikaci přechodové funkce. Po ukončení výpočtu dojde v případě přijetí k zarovnání horní a spodní řady řetězců tak, aby ve výsledku byly řetězce nahoře i dole v párovací posloupnosti shodné. Popíšeme nyní dominové kostky, které budou zkonstruovány pro Turingův stroj $M = (Q, \Sigma, \delta, q_0, F = \{q_1\})$ a vstup $x = x_1 x_2 \dots x_n$. Tyto kostky budou tvořit výslednou množinu P.

(I) První kostka vynutí to, že první konfigurací v párovací posloupnosti je počáteční konfigurace K_0 pro vstup $x = x_1 x_2 \dots x_n$.

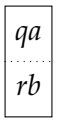
$$\left[\begin{array}{c} t_1 \\ \hdotsfor{1} \\ b_1 \end{array}\right] = \left[\begin{array}{l} \# \\ \hdotsfor{1} \\ \#\lambda\lambda q_0 x_1 x_2 \dots x_n \# \end{array}\right]$$

Pokud by byl vstup x prázdný, pak vložíme za q_0 znak prázdného políčka. Všimněme si dvou znaků prázdných políček vložených před symbol stavu q_0. Ty jsou zde proto, aby bylo možno hned v prvním možno pohnout ze stavu q_0 hlavou doleva. Dva jsou proto, aby i po pohybu hlavou doleva v prvním kroku, před hlavou stále ještě jeden znak prázdného políčka zbýval.

(II) Pro každý znak $a \in \Sigma$ vložíme do P kostku, která provede okopírování znaku do následující konfigurace v místech, která nemají být změněna. Jde o znaky na pásce, které nejsou pod hlavou Turingova stroje.

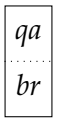


(III) Pro každou dvojici znaků $a, b \in \Sigma$ a dvojici stavů $q, r \in Q$, pro které platí $\delta(q, a) = (r, b, N)$, vložíme do P kostku



.

(IV) Pro každou dvojici znaků $a, b \in \Sigma$ a dvojici stavů $q, r \in Q$, pro které platí $\delta(q, a) = (r, b, R)$, vložíme do P kostku

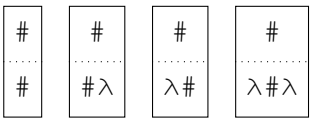


.

(V) Pro každou trojici znaků $a, b, c \in \Sigma$ a dvojici stavů $q, r \in Q$, pro které platí $\delta(q, a) = (r, b, L)$, vložíme do P kostku

.

(VI) Dále přidáme do P čtyři kostky, které umožní zakončit konfiguraci.



První kostka pouze zakončuje konfiguraci nahoře i dole. Druhá a třetí kostka umožňují rozšířit konfiguraci o prázdné políčko nalevo nebo napravo. Čtvrtá kostka pak umožňuje přidat prázdné políčko nalevo i napravo.

(VII) V případě, že Turingův stroj M přejde do přijímajícího stavu q_1, můžeme použít následujících kostek k postupnému vymazání znaků z pásky. Pro každý znak $a \in \Sigma$ vložíme do P kostky

$$\left[\begin{array}{c} aq_1 \\ \hdashline q_1 \end{array}\right] \quad \left[\begin{array}{c} q_1 a \\ \hdashline q_1 \end{array}\right].$$

(VIII) Na úplný závěr je potřeba odstranit i stav q_1 z konfigurace a provést zarovnání znaků # kostkou

$$\left[\begin{array}{c} q_1 \# \# \\ \hdashline \# \end{array}\right].$$

Ukážeme si nyní na příkladu, jakým způsobem odpovídá přijímající výpočet Turingova stroje M nad vstupem x párovací posloupnosti pro množinu P, která se skládá z kostek typů (I) až (VIII).

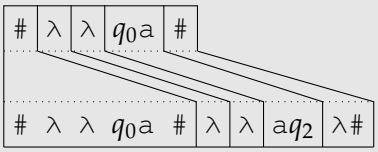
Příklad 7.5.4

Uvažme Turingův stroj $M = (Q, \Sigma, \delta, q_0, \{q_1\})$, který jsme popsali v příkladu 4.1.4. Připomeňme si, že se jednalo o Turingův stroj, který rozhoduje jazyk palindromů $\mathrm{PAL} = \{w = w^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$. Rozmysleme si, jak by vypadala párovací sekvence v sadě kostek P vytvořené k dvojici Turingova stroje M a vstupu $x =$ „$\mathtt{a}$“. První kostkou je nutně kostka typu (I), která obsahuje počáteční konfiguraci výpočtu:

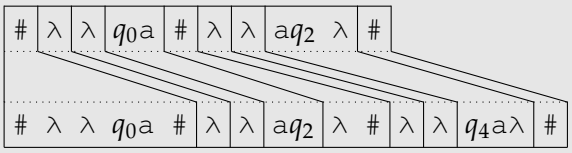
$$\left[\begin{array}{l} \# \\ \hdashline \#\lambda\lambda q_0 \mathtt{a} \# \end{array}\right]$$

Vzhledem k tomu, že dle tabulky 4.1 je $\delta(q_0, \mathtt{a}) = (q_2, \mathtt{a}, R)$, jediná kostka, která přichází v úvahu pro párování znaku odpovídajícího stavu q_0, je kostka $\left[\begin{array}{c} q_0 \mathtt{a} \\ \hdashline \mathtt{a} q_2 \end{array}\right]$ typu (IV). Před jejím použitím je nutné spárovat znaky prázdných políček dvěma kostkami $\left[\begin{array}{c} \lambda \\ \hdashline \lambda \end{array}\right]$ typu (II). Potřebujeme dále přidat prázdné políčko na konec kon-

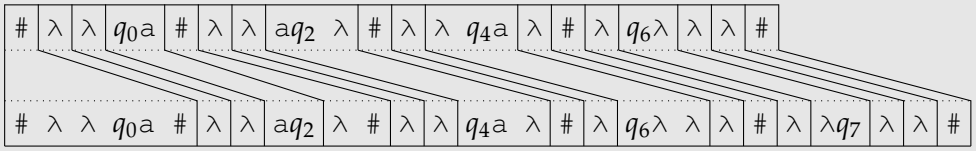
figurace a uzavřít konfiguraci kostkou $\begin{array}{|c|}\hline \# \\ \hdashline \lambda\# \\ \hline \end{array}$ typu (VI). Volíme zde kostku s přidáním prázdného políčka doprava, abychom měli za znakem stavu q_2 ještě čtené prázdné políčko. Situace nyní vypadá takto:



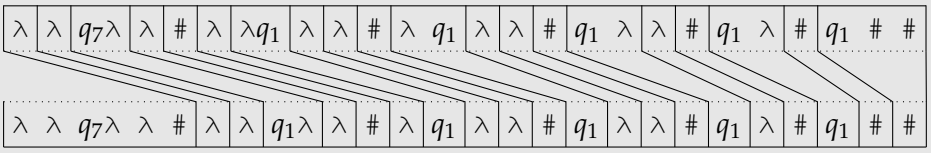
Potřebujeme tedy zarovnat řetězec $\lambda a q_2 \lambda \#$. Dle přechodové funkce M je $\delta(q_2, \lambda) = (q_4, \lambda, L)$. Musíme tedy použít kostku $\begin{array}{|c|}\hline a q_2 \lambda \\ \hdashline q_4 a \lambda \\ \hline \end{array}$ typu (V). Tuto kostku doplníme kostkami $\begin{array}{|c|}\hline \lambda \\ \hdashline \lambda \\ \hline \end{array}$ typu (II) a $\begin{array}{|c|}\hline \# \\ \hdashline \# \\ \hline \end{array}$ typu (VI). Dostaneme následující situaci:



Takto pokračujeme dál použitím instrukcí $\delta(q_4, a) = (q_6, \lambda, L)$ a $\delta(q_6, \lambda) = (q_7, \lambda, R)$, s použitím příslušných kostek se dostaneme do následující situace:



Nyní přidáme kostky odpovídající provedení instrukce $\delta(q_7, \lambda) = (q_1, \lambda, N)$. Ve spodní řadě nyní přebývá řetězec $\lambda\lambda q_1 \lambda\lambda\#$. K jeho spárování použijeme čtyři kostky typu (VII) spolu s kostkami typu (II) a (VI). Párovací posloupnost ukončíme kostkou (VIII):



Tím jsme dokončili konstrukci párovací posloupnosti.

Na příkladu 7.5.4 si můžeme všimnout, že volba kostek, které simulují výpočet je daná přechodovou funkcí, jež nám nedává na výběr. Můžeme sice v konfiguraci přidávat prázdná políčka na okrajích, ale jinak je následující konfigurace v řadě daná přechodovou funkcí. Navíc si můžeme všimnout, že od začátku je řetězec na spodní straně delší, než řetězec na horní straně kostek. Jediné kostky, které mohou zkrátit horní řetězec jsou přitom typu (VII) a (VIII), tedy kostky, které vyžadují přítomnost přijímajícího stavu. Spárování je tedy skutečně možné jen v případě, kdy je výpočet přijímající. Z těchto úvah plyne i korektnost celého převodu. Formální důkaz platnosti ekvivalence (7.8) zde však popisovat nebudeme, protože by byl technicky náročný a nepřinesl by již žádné nové myšlenky.

7.5.2. Převod problému MPKP na problém PKP

Zbývá ukázat, že problém MPKP je m-převoditelný na problém PKP. Instance obou problémů vypadají podobně, avšak v MPKP máme danou první kostku, kterou musí začínat každá párovací posloupnost. Mějme tedy množinu kostek P, musíme popsat, jak zkonstruovat množinu kostek P', pro kterou by platila následující ekvivalence:

$$\text{Z } P \text{ lze vybrat párovací posloupnost, jež začíná kostkou } \left[\begin{smallmatrix} t_1 \\ b_1 \end{smallmatrix}\right] \text{ právě když z } P' \text{ lze vybrat párovací posloupnost.} \tag{7.9}$$

Trik je v tom, že vhodnou úpravou kostek v P vynutíme, že každá párovací posloupnost v P' musí začínat variantou kostky $\left[\begin{smallmatrix} t_1 \\ b_1 \end{smallmatrix}\right]$. Abecedu Σ rozšíříme o dva nové znaky $*$ a $\diamond$.

Pro účely konstrukce si zavedeme následující značení. Je-li $u = u_1 u_2 \dots u_n \in \Sigma^*$ řetězec, označíme

$$\begin{aligned} \star u &= *u_1 * u_2 * \cdots * u_n \\ u\star &= u_1 * u_2 * \cdots * u_n * \\ \star u\star &= *u_1 * u_2 * \cdots * u_n * . \end{aligned}$$

Nyní definujeme P' na základě P následujícím způsobem.

$$P' = \left\{ \left[\begin{array}{c} \star t_i \star \\ \hline b_i \star \end{array}\right] \;\middle|\; \left[\begin{array}{c} t_i \\ \hline b_i \end{array}\right] \in P \right\} \cup \left\{ \left[\begin{array}{c} \star t_1 \\ \hline \star b_1 \star \end{array}\right], \left[\begin{array}{c} *\diamond \\ \hline \diamond \end{array}\right] \right\} \tag{7.10}$$

Máme-li párovací posloupnost $i_1, i_2, \ldots, i_l$ v instanci P, kde $i_1 = 1$, jednoduše z ní vytvoříme posloupnost pro P' — tvoří ji kostky $\left[\begin{array}{c} \star t_1 \\ \hline \star b_1 \star \end{array}\right]$, $\left[\begin{array}{c} \star t_{i_2} \\ \hline b_{i_2} \star \end{array}\right]$, ..., $\left[\begin{array}{c} \star t_{i_l} \\ \hline b_{i_l} \star \end{array}\right]$, $\left[\begin{array}{c} *\diamond \\ \hline \diamond \end{array}\right]$.

Na druhou stranu předpokládejme, že $i_1, i_2, \ldots, i_l$ je párovací posloupnost v instanci P'. První kostkou v této posloupnosti musí být $\left[\begin{array}{c} \star t_1 \\ \hline \star b_1 \star \end{array}\right]$, to je totiž jediná kostka, kde spodní řetězec začíná hvězdičkou. Přitom horní řetězec vždy začíná hvězdičkou, tedy horní i spodní řetězec v párovací posloupnosti musí hvězdičkou začínat. Naopak posledním znakem je jistě $\left[\begin{array}{c} *\diamond \\ \hline \diamond \end{array}\right]$, který dorovná přebývající hvězdičku na spodní straně.

Není těžké nahlédnout, že odstraníme-li z posloupnosti výskyty $\left[\begin{array}{c} *\diamond \\ \hline \diamond \end{array}\right]$ a nahradíme-li ostatní kostky v párovací posloupnosti jejich obrazy v P (tedy odstraníme vložené hvězdičky z obou řetězců v každé kostce), dostaneme párovací posloupnost pro P, která začíná kostkou $\left[\begin{array}{c} t_1 \\ \hline b_1 \end{array}\right]$.

Dohromady dostáváme, že ekvivalence (7.9) je splněna a tím jsme dokončili převod Přijetí vstupu $\leq_m$ MPKP $\leq_m$ PKP. Problém PKP je tedy algoritmicky nerozhodnutelný.

7.6. Jazyky za hranicí částečné rozhodnutelnosti

V této kapitole si ukážeme několik příkladů jazyků, pro které platí, že ani ony ani jejich doplňky nejsou částečně rozhodnutelné.

7.6.1. Ekvivalence programů

Uvažme nejprve problém, v němž se ptáme, zda dva dané Turingovy stroje přijímají týž jazyk.

Problém 7.6.1: Ekvivalence programů

Instance: Dva Turingovy stroje M_1 a M_2 dané svými kódy $\langle M_1 \rangle$ a $\langle M_2 \rangle$.

Otázka: Platí $L(M_1) = L(M_2)$?

Tento problém můžeme zapsat ve formě jazyka EQ:

$$\text{EQ} = \{\langle M_1, M_2 \rangle \mid L(M_1) = L(M_2)\} \tag{7.11}$$

Ukážeme si, že EQ ani jeho doplněk $\overline{\text{EQ}}$ nejsou částečně rozhodnutelné jazyky. Rozmysleme si nejprve, že tento jazyk není rozhodnutelný. Uvažme jazyk

$$\text{NONEMPTY} = \{\langle M \rangle \mid L(M) \neq \varnothing\},$$

který jsme zavedli v příkladu 6.1.3, kde jsme ukázali, že tento jazyk je částečně rozhodnutelný. V příkladu 7.2.8 jsme dále ukázali, že tento jazyk není rozhodnutelný, což přímo vyplývá i z Riceovy věty, jak jsme již zmínili v důsledku 7.4.3. Z Postovy věty dále vyplývá, že doplněk tohoto jazyka

Věty 7.4.1 a 6.1.5

$$\text{EMPTY} = \overline{\text{NONEMPTY}} = \{\langle M \rangle \mid L(M) = \varnothing\}$$

není částečně rozhodnutelný. Není těžké nahlédnout, že $\text{EMPTY} \leq_m \text{EQ}$. Převod je velmi jednoduchý: Uvažme Turingův stroj N, pro který platí, že $L(N) = \varnothing$ — může se například jednat o Turingův stroj s přechodovou funkcí, která není definovaná pro žádný displej, takový stroj všechny vstupy odmítne. Potom pro každý Turingův stroj M platí

$$\langle M \rangle \in \text{EMPTY} \Leftrightarrow L(M) = \varnothing \Leftrightarrow L(M) = L(N) \Leftrightarrow \langle M, N \rangle \in \text{EQ}.$$

Z toho plyne, že funkce $f(\langle M \rangle) = \langle M, N \rangle$ je algoritmicky vyčíslitelnou funkcí, jež převádí EMPTY na EQ. Dle bodu (ii) věty 7.2.7 tedy nejenže EQ není rozhodnutelný jazyk, ale není navíc ani částečně rozhodnutelný.

Podobným způsobem ukážeme, že ani $\overline{\text{EQ}}$ není částečně rozhodnutelný jazyk. Ukážeme, že univerzální jazyk L_u (viz (5.4)) je převoditelný na EQ. Z toho dostaneme, že $\overline{L_u} \leq_m \overline{\text{EQ}}$, a tedy $\overline{\text{EQ}}$ není částečně rozhodnutelný. Připomeňme si, že

$$L_u = \{\langle M, x \rangle \mid x \in L(M)\}$$

Zavedeno v (5.4)

K danému Turingovu stroji M a vstupu x sestrojíme Turingův stroj N, který nad vstupem y pracuje dle následujícího algoritmu.

Výpočet Turingova stroje N nad vstupem y

1: **if** $y = x$ **then**
2: **accept**
3: **else**

```
    Pusť M(y)
    if výpočet M(y) skončil přijetím then
        accept
    else
        reject
    end if
end if
```

Jediný vstup, na kterém se může jazyk $L(N)$ lišit od $L(M)$ je x. Přesněji, platí $L(N) = L(M) \cup \{x\}$. Z toho plyne, že $L(N) = L(M)$ právě když $x \in L(M)$. Pokud definujeme funkci $f(\langle M, x\rangle) = \langle M, N\rangle$, pak f je algoritmicky vyčíslitelná funkce, která převádí univerzální jazyk L_u na EQ, a tedy i $\overline{L_u}$ na $\overline{EQ}$. Jazyk $\overline{EQ}$ tedy není částečně rozhodnutelný.

7.6.2. Konečnost jazyka

Uvažme nyní problém, v němž se ptáme, zda je jazyk přijímaný daným Turingovým strojem konečný.

Problém 7.6.2: Konečnost jazyka

Instance: Turingův stroj M daný svým kódem $\langle M\rangle$
Otázka: Je $L(M)$ konečný jazyk?

Tento problém a jeho doplněk formalizujeme jazyky

$$\text{Fin} = \{\langle M\rangle \mid L(M) \text{ je konečný jazyk}\} \tag{7.12}$$
$$\text{Inf} = \overline{\text{Fin}} = \{\langle M\rangle \mid L(M) \text{ je nekonečný jazyk}\} \tag{7.13}$$

V důsledku 7.4.3 jsme již nahlédli, že nerozhodnutelnost obou těchto jazyků plyne z Riceovy věty. Definujeme-li třídy jazyků

Věta 7.4.1

$$\mathcal{FIN} = \{L \mid L \text{ je konečný jazyk}\}$$
$$\mathcal{INF} = \overline{\mathcal{FIN}} = \{L \mid L \text{ je nekonečný jazyk}\},$$

pak (ve smyslu znění Riceovy věty) $\text{Fin} = L_{\mathcal{FIN}}$ a $\text{Inf} = L_{\mathcal{INF}}$. Vzhledem k tomu, že prázdný jazyk $\varnothing \in \mathcal{FIN}$, převod v důkazu Riceovy věty v sekci 7.4 ukazuje, že $L_u \leq_m \text{Inf}$. Ukážme, že platí i $L_u \leq_m \text{Fin}$, z toho plyne, že ani Inf ani Fin nejsou částečně rozhodnutelné jazyky.

Nechť M je Turingův stroj a x je jeho vstup. Definujme na základě M a x Turingův stroj N, jehož práce nad vstupem y se řídí následujícím algoritmem.

Práce Turingova stroje N nad vstupem y

```
1: Simuluj práci M nad vstupem x po |y| kroků.
```

2: **if** $M(x)$ v daném limitu přijal **then**
3: **reject**
4: **else**
5: **accept**
6: **end if**

Pokud $x \in L(M)$ a délka výpočtu $M(x)$ je t, pak $L(N) = \{y \in \Sigma^* \mid |y| < t\}$, proto $L(N) \in \mathcal{FIN}$ a $\langle N \rangle \in$ Fin. Na druhou stranu pokud $x \notin L(M)$, pak $L(N) = \Sigma^* \in \mathcal{INF}$, a tedy $\langle N \rangle \in$ Inf. Dohromady dostáváme, že

$$\langle M, x \rangle \in L_u \Leftrightarrow x \in L(M) \Leftrightarrow \langle N \rangle \in \text{Fin}.$$

Funkce $f(\langle M, x \rangle) = \langle N \rangle$ je tedy algoritmicky vyčíslitelnou funkcí, jež převádí L_u na Fin.

8. Věta o rekurzi a její aplikace *

8.1. Věta o rekurzi

Na závěr části o vyčíslitelnosti probereme větu o rekurzi, čili větu o pevném bodě. Tato věta má mnoho důsledků, za jeden z nich se dá považovat i Riceova věta, což si také ukážeme. Začneme zněním věty o rekurzi.

Věta 8.1.1 (Kleene, Věta o rekurzi) *Pro libovolnou ORF jedné proměnné f existuje n (jež zveme* pevným bodem *f), pro které platí, že $\varphi_n \simeq \varphi_{f(n)}$.*

Zamysleme se nejprve nad významem věty, funkci f si můžeme představit jako transformaci algoritmu, vstupem funkce f je Gödelovo číslo, tedy program, a jejím výstupem je nový program. To, co věta 8.1.1 říká, tedy znamená, že pro jakoukoli transformaci programů f existuje nějaký program n, který i po provedení transformace $f(n)$ počítá touž funkci, tedy $\varphi_n \simeq \varphi_{f(n)}$. Tím, že f je obecně rekurzivní, má i zápis $\varphi_{f(n)}$ vždy smysl. Všimněme si, že funkce φ_n ani funkce $\varphi_{f(n)}$ nemusí být definované pro žádný vstup, potom tvrzení $\varphi_n \simeq \varphi_{f(n)}$ není příliš zajímavé, nicméně ve chvíli, kdy tyto funkce jsou definované třeba pro všechny vstupy, můžeme dostat pomocí věty o rekurzi zajímavá tvrzení.

Ukážeme si postupně dva důkazy, oba z nich jsou velmi krátké, pokusíme se ale u obou i zdůvodnit, proč fungují, protože to není tak jednoduché pochopit.

Důkaz: První důkaz věty o rekurzi 8.1.1 Tento důkaz byl převzat z [9], ale je obsažen i v [7].

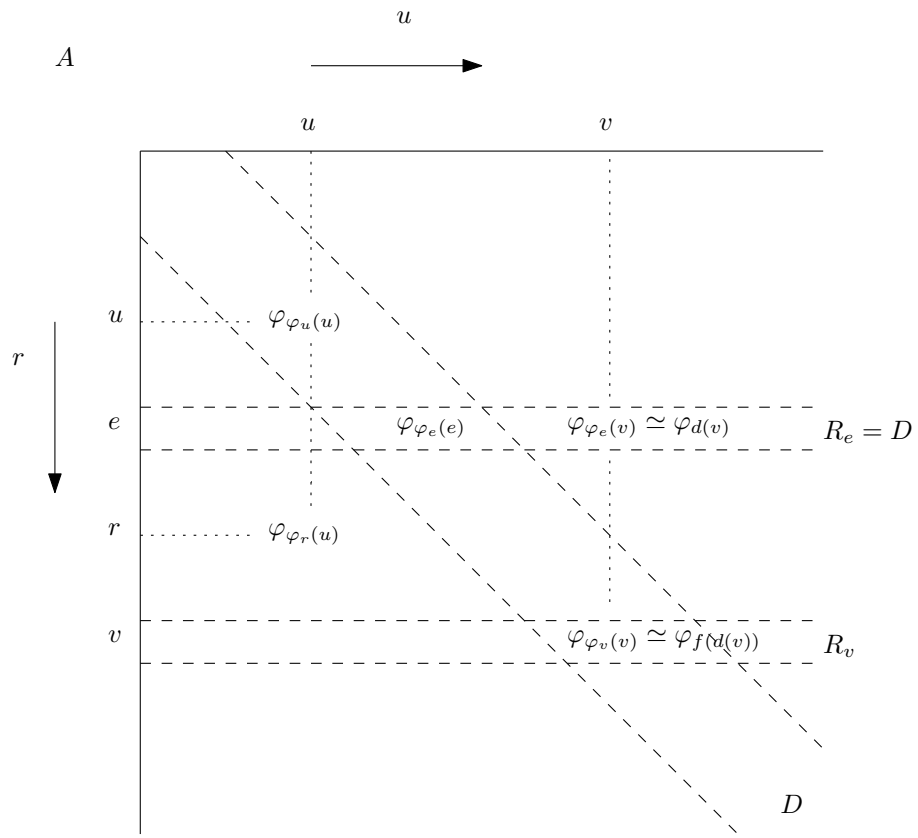
Základem prvního důkazu je diagonalizace, tentokrát však použitá jiným způsobem než jak jsme viděli dosud. Používáme-li v důkazu diagonalizaci, postupujeme obvykle podle následujícího schematu. Mějme matici $A = \{\alpha_{i,j}\}_{i,j\in\mathbb{N}}$ a uvažme prvky na diagonále, tj. $\{\alpha_{i,i}\}_{i\in\mathbb{N}}$. Označme si tuto posloupnost pomocí d, tedy

$$d_i = \alpha_{i,i}. \tag{8.1}$$

Nyní vytvoříme novou posloupnost d', pro kterou platí, že $d'_i \neq d_i$, o takové posloupnosti můžeme nyní prohlásit, že se nevyskytuje jako řádek ani jako sloupec matice A, neboť se s každým řádkem i sloupcem v jednom prvku liší (s i-tým řádkem/sloupcem se liší na pozici $d'_i = \alpha'_{i,i} \neq \alpha_{i,i}$).

Nyní však uvažme trochu jiné použití diagonalizace, uvažme situaci, kdy se diagonální posloupnost $\{d_i\}_{i\in\mathbb{N}}$ v matici A vyskytuje jako jeden z jejích řádků, řekněme e-tý, tedy pro každé $i \in \mathbb{N}$ platí

$$d_i = \alpha_{e,i} = \alpha_{i,i}. \tag{8.2}$$



Obrázek 8.1.: Ilustrativní obrázek k prvnímu důkazu věty o rekurzi.

Proveďme nyní opět úpravu posloupnosti $\{d_i\}_{i\in\mathbb{N}}$ na novou posloupnost $\{d'_i\}_{i\in\mathbb{N}}$, ale nyní předpokládejme, že i takto upravená posloupnost se vyskytuje v matici A jako jeden z jejích řádků, například jako v-tý, tj. pro každé $i \in \mathbb{N}$ platí

$$d'_i = \alpha_{v,i}. \tag{8.3}$$

V této situaci dostaneme, že prvek na diagonále na v-tém řádku musel touto transformací projít netknutý, neboť $d_v = \alpha_{v,v} = d'_v$, kde první rovnost plyne z definice d v (8.1) a druhá rovnost plyne z (8.3). Také z toho podle (8.2) plyne, že pro v platí, že $\alpha_{e,v} = \alpha_{v,v}$, tedy hodnoty ve v-tém sloupci jsou na e-tém a v-tém řádku shodné.

Využijeme nyní tohoto postupu k důkazu věty o rekurzi. Prvním krokem je definice vhodné matice A (viz též ilustrativní obrázek 8.1), v níž položíme $\alpha_{r,u} \simeq \varphi_{\varphi_r(u)}$, přičemž předpokládáme, že pokud $\varphi_r(u)\uparrow$, potom $\alpha_{r,u}$ představuje funkci, jež není nikde definována[1]. Všimněme si, že v takto definované matici A nelze popsat algoritmický postup, který by k funkci $\alpha_{r,u}$ našel jinou, která se od ní liší, protože o dvou částečně rekurzivních funkcích nejsme ani schopni rozhodnout, zda jsou si podmíněně rovny.

[1] To odpovídá tomu, že $\alpha_{r,u}(x) \simeq \varphi_z^{(2)}(\varphi_z^{(2)}(r,u),x)$, kde $\varphi_z^{(2)}$ označuje univerzální ČRF pro funkce jedné proměnné.

Podívejme se nyní na posloupnost diagonálních prvků

$$D = \{\alpha_{u,u}\}_{u\in\mathbb{N}} = \{\varphi_{\varphi_u(u)}\}_{u\in\mathbb{N}}.$$

Nechť e_1 je Gödelovým číslem funkce, pro kterou platí, že[2]

$$\varphi^{(2)}_{e_1}(u,x) \simeq \alpha_{u,u} \simeq \varphi_{\varphi_u(u)}(x).$$

Podle s-m-n věty 4.5.10 platí, že $\varphi^{(2)}_{e_1}(u,x) \simeq \varphi_{s^1_1(e_1,u)}(x)$, označme si $d(u) \simeq s^1_1(e_1,u)$, víme přitom, že jde o prostou PRF. Gödelovo číslo funkce d si označme pomocí e, dostáváme tedy, že

$$\alpha_{e,u} \simeq \varphi_{\varphi_e(u)}(x) \simeq \varphi_{d(u)}(x) \simeq \varphi_{s^1_1(e_1,u)}(x) \simeq \varphi^{(2)}_{e_1}(u,x) \simeq \varphi_{\varphi_u(u)}(x) \simeq \alpha_{u,u} = D(u).$$

Diagonála matice A se tedy rovná jejímu e-tému řádku, který označíme jako $R_e = \{\alpha_{e,u}\}_{u\in\mathbb{N}}$, a tedy $D = R_e$. ORF f provádí transformaci matice A, řádek

$$R_e = \{\alpha_{e,u} \simeq \varphi_{\varphi_e(u)} \simeq \varphi_{d(u)}\}_{u\in\mathbb{N}}$$

přemapuje na řádek R_v, kde v je Gödelovým číslem funkce $f \circ d$, tedy

$$R_v = \{\alpha_{v,x} \simeq \varphi_{\varphi_v(x)} \simeq \varphi_{f(d(x))}\}_{x\in\mathbb{N}}$$

Řádek R_e však obsahoval diagonálu, jeden z prvků musí tedy zůstat beze změny, a to ten, který se na řádku R_v promítne na diagonálu, což je $\alpha_{e,v}$, které se promítne do $\alpha_{v,v}$. Musí tedy platit $\alpha_{e,v} \simeq \alpha_{v,v}$. Pokud rozvineme tuto úvahu, tak dostaneme

$$\varphi_{d(v)} \simeq \alpha_{e,v} \simeq \alpha_{v,v} \simeq \varphi_{f(d(v))},$$

nebo také

$$\varphi_{d(v)} \simeq \varphi_{\varphi_e(v)} \simeq \varphi_{\varphi_v(v)} \simeq \varphi_{f(d(v))}$$

kde první rovnost platí proto, že Gödelovým číslem funkce d je e, tedy $d \simeq \varphi_e$, druhá rovnost platí proto, že diagonála se rovná e-tému řádku, tj. $D = R_e$, a konečně třetí rovnost platí díky tomu, že v je Gödelovým číslem funkce $f \circ d$, tj. $\varphi_v(v) \simeq f(d(v))$. Položíme-li tedy $n = d(v)$, získáme pevný bod funkce f. Připomeňme si, že funkci d jsme odvodili s pomocí s-m-n věty a je to tedy dokonce prostá PRF. Všimněme si, že i číslo v můžeme efektivně spočítat z Gödelových čísel funkcí f a d, protože jde o složení dvou funkcí. □

I další důkaz, který si předvedeme, je založen na diagonalizaci.

[2] Gödelovo číslo e_1 bychom opět mohli určit s pomocí univerzální funkce $\varphi^{(2)}_z$:

$$\varphi^{(2)}_{e_1}(u,x) \simeq \varphi_{\varphi_u(u)}(x) \simeq \varphi^{(2)}_z(\varphi^{(2)}_z(u,u),x).$$

Důkaz: Druhý důkaz věty o rekurzi 8.1.1 Tento důkaz byl převzat z [7].

Nechť e_1 je číslem funkce, pro kterou platí

$$\varphi^{(2)}_{e_1}(e,x) \simeq \varphi_{f(\varphi_e(e))}(x),$$

tuto funkci bychom snadno odvodili pomocí univerzální ČRF[3]. Nechť b je Gödelovým číslem funkce $s^1_1(e_1,e)$, podle s-m-n věty (4.5.10) tedy platí, že

$$\varphi_{\varphi_b(e)}(x) \simeq \varphi_{s^1_1(e_1,e)}(x) \simeq \varphi^{(2)}_{e_1}(e,x) \simeq \varphi_{f(\varphi_e(e))}(x).$$

Protože φ_b je PRF, je $\varphi_b(b)\downarrow$ a platí, že

$$\varphi_{\varphi_b(b)} \simeq \varphi_{f(\varphi_b(b))},$$

$\varphi_b(b)$ je tedy hledaným pevným bodem f.

Zkusme si rozebrat, jakými úvahami lze k podobnému důkazu dospět. Jak je vidět již z uvedeného formálního zápisu, použijeme hned dvě diagonalizace.

Hledaným pevným bodem funkce f bude číslo n následujícího programu:
Program n: „Uprav program n podle f a výsledek aplikuj na vstup x.“
Pak podle definice platí $\varphi_n \simeq \varphi_{f(n)}$. Takové číslo n bychom tedy chtěli najít. Pro dané n můžeme spočítat číslo takového programu jednoduchou úpravou funkce f, tedy existuje dokonce PRF $h(n)$, která spočítá číslo výše zmíněného programu pro dané n a funkci f danou svým Gödelovým číslem. Je-li $h \simeq \varphi_a$, pak $h(n) \simeq \varphi_a(n)$, přičemž naším cílem je najít kombinaci a a n tak, abychom dostali následující program:
Program $\varphi_a(n)$: „Uprav program s číslem $\varphi_a(n)$ podle f a výsledek aplikuj na x.“
Tento program závisí na a a n a takhle bychom mohli přidávat parametry do nekonečna, abychom se tomu vyhnuli, začneme hledat program s číslem ve tvaru $\varphi_e(e)$, což bude první použitá diagonalizace. Toto číslo závisí jen na jednom parametru a navíc má správný tvar. Číslo tohoto programu můžeme spočítat s pomocí nějaké primitivně rekurzivní funkce φ_b na základě e se znalostí Gödelova čísla funkce f, tedy:
Program $\varphi_b(e)$: „Uprav program s číslem $\varphi_e(e)$ podle f a výsledek aplikuj na x.“
Nyní stačí použít diagonalizaci podruhé a vzít $e = b$, protože $\varphi_b(b)$ je číslo programu:
Program $\varphi_b(b)$: „Uprav program s číslem $\varphi_b(b)$ podle f a výsledek aplikuj na x.“
Tento program zřejmě dělá totéž, co program $f(\varphi_b(b))$ a $\varphi_b(b)$ je tedy hledaný pevný bod n. □

Z toho, jak jsme určili pevný bod funkce f, plyne, že je možno jej určit efektivně z Gödelova čísla funkce f.

Důsledek 8.1.2 *Existuje prostá PRF g, která ke Gödelovu číslu funkce f určí její pevný bod.*

Důkaz: (*Uvažujeme první důkaz věty o rekurzi 8.1.1.*) Nechť $v(x)$ označuje funkci, pro níž platí $\varphi_{v(x)} \simeq \varphi_x \circ d$, funkci v dostaneme ze s-m-n věty, potom $g(x) \simeq d(v(x))$. Protože funkce d i v jsme obdrželi ze s-m-n věty, jsou obě funkce prosté PRF, to tedy platí i pro g. □

[3] Označíme-li si univerzální ČRF pro funkce jedné proměnné pomocí $\varphi^{(2)}_z$, pak $\varphi^{(2)}_{e_1}(e,x) \simeq \varphi^{(2)}_z(f(\varphi^{(2)}_z(e,e)),x)$.

Důsledek 8.1.3 *Každá ORF f má nekonečně mnoho pevných bodů.*

Důkaz: (*Uvažujeme první důkaz věty o rekurzi 8.1.1.*) Funkce $f \circ d$ má, stejně jako všechny ČRF, nekonečně mnoho Gödelových čísel, z každého z nich dosazením do d dostaneme pevný bod funkce f, protože d je prostá funkce, dostaneme tedy nekonečně mnoho pevných bodů funkce f. □

Ukažme si alespoň jednoduché použití věty o rekurzi.

Důsledek 8.1.4 *1. Existuje $n \in \mathbb{N}$, pro nějž $W_n = \{n\}$.*

2. Existuje $n \in \mathbb{N}$, pro nějž $\varphi_n \simeq \lambda x[n]$.

Důkaz: 1. Nechť e je Gödelovo číslo funkce definované jako

$$\varphi_e^{(2)}(x, y) \simeq \mu z[x = y].$$

Pro tuto funkci tedy platí, že $\varphi_e^{(2)}(x, y) \downarrow$ právě když $(x = y)$. Použijeme-li větu 4.5.10 (s-m-n) a definujeme-li $f(x) \simeq s_1^1(e, x)$, dostaneme, že $\varphi_{f(x)} \simeq \varphi_{s_1^1(e,x)} \simeq \varphi_e^{(2)}(x, y)$ a tedy $W_{f(x)} = \{x\}$. Funkce f je podle s-m-n věty obecně rekurzivní, a tak na ni můžeme použít větu o rekurzi 8.1.1, podle které existuje n, pro nějž $\varphi_n \simeq \varphi_{f(n)}$, a tak

$$W_n = W_{f(n)} = \{n\}.$$

2. Podobně jako v předchozím bodu, nechť e je Gödelovo číslo funkce definované jako

$$\varphi_e^{(2)}(x, y) \simeq x.$$

Pomocí s-m-n věty definujeme-li $f(x) = s_1^1(e, x)$, dostaneme $\varphi_{f(x)}(y) \simeq x$ a s použitím věty o rekurzi nalezneme n, pro něž je

$$\varphi_n(y) \simeq \varphi_{f(n)}(y) \simeq n.$$

S pomocí prvního bodu důsledku 8.1.4 lze mimo jiné ukázat, že neexistuje třída ČRF $\mathcal{C}$ taková, pro kterou by platilo, že $K = \{e \mid \varphi_e \in \mathcal{C}\}$, tento fakt ponecháme čtenáři jako jednoduché cvičení.

Podle druhého bodu důsledku 8.1.4 dostáváme, že existuje ČRF, jejímž výstupem je její vlastní Gödelovo číslo, tedy její kód. Protože například i programy v jazyce C (nebo jakémkoli jiném) tvoří stejně silný prostředek jako jsou Turingovy stroje a ČRF, z věty o rekurzi plyne i to, že existuje například program v C, který vypíše svůj zdrojový kód (a navíc ignoruje parametry a nečte ani žádný soubor, protože vypsat svůj zdrojový soubor, když jej může číst, je triviální). Ve skutečnosti takový program nemusí být ani dlouhý. Takovému programu, který vypíše svůj zdrojový kód, se říká quinovský podle logika a filozofa Willarda Van Ormana Quinea.

Tvrzení věty o rekurzi lze rozšířit o parametry, což bude jediná z řady variant věty o rekurzi, kterou si ukážeme.

Věta 8.1.5 (Kleene, Věta o rekurzi s parametry) *Nechť $f(x,y)$ je ORF, potom existuje prostá ORF $n(y)$ taková, že $\varphi_{n(y)} \simeq \varphi_{f(n(y),y)}$ pro každé $y \in \mathbb{N}$.*

Důkaz: Důkaz je analogický důkazu věty 8.1.1 (*uvažujeme první důkaz*). Pomocí s-m-n věty definujeme funkci d, která splňuje

$$\varphi_{d(x,y)}(z) = \begin{cases} \varphi_{\varphi_x(x,y)}(z) & \text{pokud } \varphi_x(x,y)\downarrow, \\ \uparrow & \text{jinak.} \end{cases}$$

Zvolme v tak, že $\varphi_v(x,y) \simeq f(d(x,y),y)$, potom $n(y) \simeq d(v,y)$ je hledaným pevným bodem f, protože $\varphi_{d(v,y)} \simeq \varphi_{\varphi_v(v,y)} \simeq \varphi_{f(d(v,y),y)}$. První rovnost plyne z definice d, druhá z definice v. Protože funkci $n(y)$ jsme dostali ze s-m-n věty, jedná se dokonce o prostou PRF. □

8.2. Důkaz Riceovy věty pomocí věty o rekurzi

Jako důsledek věty o rekurzi můžeme uvažovat i Riceovu větu (uvažujeme verzi pro funkce, tedy 7.4.2, využívajíce navíc toho, že částečně rekurzivní funkce jsou právě algoritmicky vyčíslitelné funkce).

Důsledek 8.2.1 (Riceova věta) *Nechť $\mathcal{C}$ je libovolná třída částečně rekurzivních funkcí, potom je množina $A_\mathcal{C} = \{e \mid \varphi_e \in \mathcal{C}\}$ rekurzivní, právě když $\mathcal{C} = \varnothing$ nebo $\mathcal{C}$ obsahuje všechny ČRF.*

Důkaz: Je-li $\mathcal{C}$ prázdná nebo obsahuje všechny ČRF, pak $A_\mathcal{C}$ je zřejmě rekurzivní, což jsme ukázali už v přímém důkazu Riceovy věty (věta 7.4.1, ačkoli v důkazu to bylo formulováno pro jazyky, pro funkce je důkaz obdobný). Předpokládejme tedy, že $\mathcal{C}$ není prázdná, ale neobsahuje ani všechny ČRF. Předpokládejme sporem, že $A_\mathcal{C}$ je rekurzivní. Protože $A_\mathcal{C}$ je neprázdná, existuje číslo $a \in A_\mathcal{C}$, na druhou stranu $A_\mathcal{C}$ neobsahuje Gödelova čísla všech funkcí, existuje také číslo $b \notin A_\mathcal{C}$[4]. Nyní definujme funkci f následujícím způsobem:

$$f(x) = \begin{cases} a & x \notin A_\mathcal{C} \\ b & x \in A_\mathcal{C} \end{cases}$$

Funkce f je ORF, protože a, b jsou konkrétní čísla a charakteristická funkce $\chi_{A_\mathcal{C}}$ množiny $A_\mathcal{C}$ je obecně rekurzivní z předpokladu, že $A_\mathcal{C}$ je rekurzivní množina. Ať n označuje pevný bod funkce f, který dostaneme z věty o rekurzi, tedy $\varphi_n \simeq \varphi_{f(n)}$. Ptejme se, jestli $\varphi_n \in \mathcal{C}$ nebo ne.

$$\begin{array}{lllllllll} \varphi_n \in \mathcal{C} & \Rightarrow & n \in A_\mathcal{C} & \Rightarrow & f(n) = b & \Rightarrow & f(n) \notin A_\mathcal{C} & \Rightarrow & \varphi_{f(n)} \notin \mathcal{C} \\ \varphi_n \notin \mathcal{C} & \Rightarrow & n \notin A_\mathcal{C} & \Rightarrow & f(n) = a & \Rightarrow & f(n) \in A_\mathcal{C} & \Rightarrow & \varphi_{f(n)} \in \mathcal{C} \end{array}$$

[4] Poznamenejme, že z předpokládané rekurzivity $A_\mathcal{C}$ plyne, že její charakteristická funkce $\chi_{A_\mathcal{C}}$ je obecně rekurzivní. S pomocí této charakteristické funkce bychom mohli efektivně najít $a \in A_\mathcal{C}$ a $b \notin A_\mathcal{C}$. Například pomocí funkcí

$$\begin{array}{rcl} a(x) & \simeq & \lambda x[\mu(y)[\chi_{A_\mathcal{C}}(y) \simeq 1]] \text{ a} \\ b(x) & \simeq & \lambda x[\mu(y)[\chi_{A_\mathcal{C}}(y) \simeq 0]], \end{array}$$

zde $a(x)$ vrátí nejmenší číslo patřící do $A_\mathcal{C}$ a $b(x)$ vrátí nejmenší číslo nepatřící do $A_\mathcal{C}$.

Z toho plyne, že $\varphi_n \in \mathcal{C}$ právě když $\varphi_{f(n)} \notin \mathcal{C}$, což je ale ve sporu s tím, že φ_n a $\varphi_{f(n)}$ označují tutéž funkci a $\mathcal{C}$ je třídou funkcí, tedy buď tato funkce do $\mathcal{C}$ patří nebo ne bez ohledu na to, jaké její Gödelovo číslo uvažujeme. Množina $A_{\mathcal{C}}$ tedy nemůže být rekurzivní. □

8.3. Cvičení

1. Ukažte, že existuje přirozené číslo n, pro které platí, že $W_n = \{0, \ldots, n\}$.
2. Ukažte, že existuje přirozené číslo n, pro které platí, že $W_n = \{kn \mid k \in \mathbb{N}\}$.
3. Ukažte, že K není indexová množina, tj. neexistuje žádná třída částečně rekurzivních funkcí $\mathcal{C}$, pro kterou by platilo, že $K = \{e \mid \varphi_e \in \mathcal{C}\}$.

Část III.

Složitost

9. Základní třídy problémů ve složitosti

V této kapitole zavedeme základní třídy složitosti a podíváme se na jejich základní vlastnosti.

9.1. Problémy a úlohy

Budeme obvykle rozlišovat rozhodovací problém a úlohu.

Rozhodovací problém V rozhodovacím problému se ptáme, zda daný vstup má danou vlastnost. V problému souvislosti grafu se například ptáme, zda daný graf G je souvislý. Na otázku kladenou v definici takového problému tedy očekáváme odpověď typu ano/ne. Vstupu budeme říkat *instance problému*, přičemž *pozitivní instance* jsou ty, pro které je odpověď „ano“, zatímco *negativní instance* jsou ty, pro které je odpověď „ne“. Například instancí problému souvislosti grafu je neorientovaný graf, pozitivními instancemi jsou souvislé grafy a negativními instancemi jsou grafy, jež mají alespoň dvě komponenty. Rozhodovací problém formalizujeme jako jazyk slov, která kódují pozitivních instancí. Například jazyk

$$\text{CONN} = \{\langle G\rangle \mid G \text{ je souvislý graf}\},$$

formalizuje problém souvislosti grafu. Rozhodnutí, zda daný graf G je souvislý je potom ekvivalentní tomu, zda $\langle G\rangle \in \text{CONN}$. Poznamenejme, že instancí problému $w \in \text{CONN}$ je libovolný řetězec $w \in \Sigma^*$, tedy i řetězec, který nekóduje graf. Předpokládáme ovšem, že poznat, zda daný řetězec kóduje graf, je jednoduché. Můžeme si proto dovolit předpokládat, že instancí problému souvislosti grafu formalizovaného jazykem CONN je skutečně graf. Tohoto zjednodušení se budeme dopouštět velmi často.

Úloha V případě úlohy hledáme k danému vstupu x vhodné y, které splňuje danou vlastnost. Například v úloze hledání cesty v orientovaném grafu předpokládáme na vstupu orientovaný graf G a dva vrcholy s a t, tato trojice tedy tvoří *instanci* této úlohy. Očekávaným výstupem je potom seznam vrcholů na cestě z s do t. Pokud žádná taková cesta neexistuje, pak výstupem může být například prázdný seznam vrcholů. Na tomto příkladu vidíme, že výstup y nemusí být určen jednoznačně (cest z s do t může být více) a nemusí ani existovat. Formálně definujeme úlohu jako binární relaci $R \subseteq \{0,1\}^* \times \{0,1\}^*$. Úlohou je potom najít k danému řetězci x řetězec y tak, aby $(x, y) \in R$, nebo oznámit, že takový řetězec y neexistuje.

Například formalizací úlohy nalezení cesty v orientovaném grafu je relace

$$\text{PATH} = \{(\langle G, s, t\rangle, \langle y_1, \ldots, y_k\rangle) \mid$$
$$y_1, \ldots, y_k \text{ tvoří cestu v orientovaném grafu } G \text{ z } s \text{ do } t\}.$$

Úloze se též říká vyhledávací problém (*search problem*).

Optimalizační úloha je úloha, kde navíc požadujeme, aby nalezený výstup y byl optimální vzhledem k nějaké míře. Může jít například o hledání maximálního toku v síti. Tento typ úloh zavedeme později v kapitole věnující se aproximačním algoritmům.

Většinou se budeme zabývat rozhodovacími problémy, které budeme identifikovat s jazyky (tj. pojmy problém a jazyk budeme používat jako synonyma). V některých místech budeme však též používat úlohy, pro které budeme používat i pojem relace. Pro účely aproximace pak budeme uvažovat i optimalizační úlohy.

9.2. Deterministické třídy složitosti

Nyní již můžeme zavést základní třídy rozhodovacích problémů, které jsou rozhodnutelné v daném čase a prostoru. Pro účel definice použijeme výpočetní model Turingova stroje. Řekněme si nejprve, co znamená, že Turingův stroj M pracuje v omezeném čase nebo prostoru.

Definice 9.2.1 Nechť $f : \mathbb{N} \to \mathbb{N}$ je totální funkce a M je Turingův stroj.

- Řekneme, že *M pracuje v čase $f(n)$*, pokud výpočet M nad libovolným vstupem $x \in \Sigma^*$ délky $|x| = n$ skončí po provedení nejvýš $f(n)$ kroků.
- Řekneme, že *M pracuje v prostoru $f(n)$*, pokud výpočet M nad libovolným vstupem $x \in \Sigma^*$ délky $|x| = n$ skončí a během výpočtu využije M nejvýše $f(n)$ buněk pracovní pásky. ◂

S pomocí těchto pojmů nyní definujeme dvě základní třídy složitosti.

Definice 9.2.2 Nechť $f : \mathbb{N} \to \mathbb{N}$ je totální funkce, pak definujeme následující třídy problémů:

TIME$(f(n))$ je třídou problémů rozhodnutelných v čase $O(f(n))$. Přesněji, jazyk $L \subseteq \Sigma^*$ patří do třídy TIME$(f(n))$, právě když existuje Turingův stroj M, který rozhoduje jazyk L a který pracuje v čase $O(f(n))$.

SPACE$(f(n))$ je třídou problémů rozhodnutelných v prostoru $O(f(n))$. Přesněji, jazyk $L \subseteq \Sigma^*$ patří do třídy SPACE$(f(n))$, právě když existuje Turingův stroj M, který rozhoduje jazyk L a který pracuje v prostoru $O(f(n))$. ◂

Poznámka 9.2.3 *Poznamenejme, že v literatuře je možné nalézt drobné odlišnosti v definicích tříd* TIME($f(n)$) *a* SPACE($f(n)$).

- *V případě* TIME($f(n)$) *se často vyžaduje, aby Turingův stroj M pracoval v čase $f(n)$, nepovoluje se tedy násobek konstantou jako v naší definici, kde připouštíme čas $O(f(n))$. V případě* SPACE($f(n)$) *je častá podobná úprava definice. Je potřeba zmínit, že v případě Turingova stroje toto není podstatný rozdíl, neboť za určitých ne příliš omezujících předpokladů je možné každý Turingův stroj M zrychlit konstantním faktorem, či provést kompresi prostoru na pásce konstantním faktorem.*[1] *Je potřeba ovšem zdůraznit, že tato možnost je specifická pro Turingův stroj, kde připouštíme libovolně velkou abecedu. Je proto jednodušší přímo povolit čas nebo prostor $O(f(n))$ v definici tříd* TIME($f(n)$) *a* SPACE($f(n)$).

- *V případě třídy* SPACE($f(n)$) *se často nevyžaduje zastavení Turingova stroje M pro všechny vstupy. Opět platí, že pokud funkce $f(n)$ je v jistém smyslu hezká* (*přesněji je prostorově konstruovatelná, což je pojem, který budeme definovat později*), *pak je možné detekovat zacyklení M nad daným vstupem bez nárůstu požadavků na prostor. Dává proto smysl přímo předpokládat, že Turingův stroj M se zastaví nad každým vstupem. Této vlastnosti budeme dále využívat, protože zjednodušuje argumentaci o daném Turingovu stroji.*

Zaveďme si některé podstatné třídy problémů:

$$\mathrm{P} = \bigcup_{k=0}^{\infty} \mathrm{TIME}(n^k) \tag{9.1}$$

$$\mathrm{EXPTIME} = \bigcup_{k=0}^{\infty} \mathrm{TIME}(2^{n^k}) \tag{9.2}$$

$$\mathrm{PSPACE} = \bigcup_{k=0}^{\infty} \mathrm{SPACE}(n^k) \tag{9.3}$$

$$\mathrm{L} = \mathrm{SPACE}(\log_2 n) \tag{9.4}$$

Poznámka 9.2.4 *Povšimněme si třídy* L, *kde omezení na využitý prostor je menší než samotná velikost vstupu. Abychom mohli vůbec měřit prostor menší než lineární, je třeba oddělit vstupní a pracovní pásku. V případech, kdy Turingův stroj počítá funkci, je navíc je vhodné uvažovat i oddělenou výstupní pásku. Obecně tedy uvažujeme k-páskový Turingův stroj $M = (Q, \Sigma, \delta, q_0, F)$, kde*

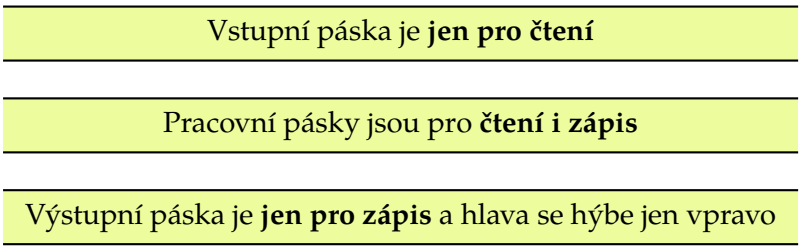
- *První páska je vstupní, tedy je na ní na začátku napsán vstup a Turingův stroj na této pásce nepřepisuje symboly. Hlavou však může pohybovat libovolně. Tato páska se nepočítá do využitého prostoru.*

- *Druhá až $k-1$-ní pásky jsou pracovní, tyto jsou na začátku výpočtu prázdné a Turingův stroj je může používat libovolným způsobem. Čtení, zápis a pohyb po této pásce není nijak omezen.*

[1] https://en.wikipedia.org/wiki/Linear_speedup_theorem

- *k-tá páska je výstupní, sem Turingův stroj zapisuje svůj výstup. Po této pásce může Turingův stroj pohybovat hlavou jen doprava a může na ni jen zapisovat, číst z ní nemůže. Obsah této pásky se nepočítá do využitého prostoru.*

Do prostoru počítáme jen počet buněk, které Turingův stroj využije na pracovních páskách, tedy počítáme prostor využití navíc ke vstupu a výstupu. To odpovídá představě programu, který čte svůj vstup ze standardního vstupu, výstup zapisuje na standardní výstup. Do prostoru počítáme jen to, kolik paměti využije tento program, nikoli to, kolik znaků musí přečíst ze vstupu nebo zapsat na výstup.



Obrázek 9.1.: Struktura pásek Turingova stroje, u nějž měříme prostor jen na pracovních páskách. Naznačen je případ s jednou pracovní páskou.

9.3. Polynomiálně rozhodnutelné problémy

Zastavme se chvíli u třídy P, kterou jsme zavedli v (9.1). Rádi bychom řekli, že jde o třídu rozhodovacích problémů, pro které existuje polynomiální algoritmus, který daný problém rozhoduje. Ovšem definice říká, že jde o třídu problémů rozhodnutelných nějakým deterministickým Turingovým strojem v polynomiálním čase. Přijmeme-li Churchovu-Turingovu tezi, pak pojmy „problém rozhodnutelný Turingovým strojem“ a „algoritmicky rozhodnutelný problém“ splývají. To ale ještě nutně neznamená, že splynou i pojmy „problém rozhodnutelný Turingovým strojem v polynomiálním čase“ a „problém rozhodnutelný nějakým algoritmem v polynomiálním čase“. Na druhou stranu v sekci 4.2 jsme zavedli výpočetní model RAM a ukázali jsme si, že tento je ekvivalentní s Turingovým strojem v tom smyslu, že problémy rozhodnutelné Turingovými stroji jsou právě ty, které jsou rozhodnutelné na RAMu. Navíc projdeme-li důkaz této ekvivalence popsaný v sekci 4.3, zjistíme, že jsme ukázali ve skutečnosti něco silnějšího. K Turingovu stroji M můžeme sestrojit RAM R, který přijímá týž jazyk a pracuje až na konstantní faktor stejně rychle jako M. Naopak k RAMu R můžeme sestrojit Turingův stroj M, který přijímá týž jazyk a navíc pracuje jen s polynomiálním zpomalením oproti R[2]. To znamená, že pokud bychom třídu P zavedli pomocí RAMu místo Turingových strojů, dostali bychom touž třídu jazyků.

Teze 5.1.1

[2]Je ovšem pravda, že jsme nijak formálně nezavedli měření času na RAMu, zde prozatím ponecháváme prostor čtenářově intuici.

Při zavádění třídy P tedy vycházíme z předpokladu, který autoři [8] formulují jako tezi o invarianci (*Invariance Thesis*).

Teze 9.3.1: Teze o invarianci (1984)

Existuje standardní třída výpočetních modelů, která mimo jiné obsahuje všechny varianty Turingových strojů, všechny varianty strojů RAM a RASP, kde počítáme aritmetické operace s logaritmickou cenou, stejně jako všechny varianty RAM a RASP, kde je cena všech instrukcí shodná a využíváme jen standardní aritmetické instrukce (sčítání, odčítání, přičtení jedničky, odečtení jedničky a test na nulu). Stroje v této standardní třídě mohou simulovat sebe navzájem s polynomiálním zpomalením a s konstantním nárůstem prostoru.

Teze 9.3.1 tedy říká, že za rozumných předpokladů na uvažované výpočetní modely můžeme brát definici třídy P jako nezávislou na použitém výpočetním modelu. Dává proto smysl hovořit o polynomiálním algoritmu bez bližší specifikace modelu.

S definicí třídy P souvisí i další teze, pojmenovaná po Alanu Cobhamovi a Jacku Edmondsovi.

Teze 9.3.2: Cobhamova-Edmondsova teze (1965)

Třída P odpovídá třídě problémů, které lze prakticky řešit na počítači.

Tato teze má však své limity, pokud složitost algoritmu bude například n^{500}, nelze o něm říci, že by byl prakticky použitelný, byť jde o polynomiální algoritmus. Na druhou stranu je třeba zmínit, že obvyklé praktické problémy, které jsou řešitelné v polynomiálním čase, mají složitost danou polynomem s malým stupněm, takže pokud už problém do třídy P patří, obvykle je i prakticky řešitelný.

9.4. Polynomiálně ověřitelné problémy

Bohužel zdaleka ne všechny problémy jsou rozhodnutelné v polynomiálním čase a nelze to říci ani o problémech praktických. O řadě problémů, které jsou celkem přirozené, však můžeme říci, že jsou polynomiálně ověřitelné. Uvažme následující problém.

Problém 9.4.1: Problém obchodního cestujícího

Instance: Je dáno n měst $c_1, \ldots, c_n$, pro každou dvojici měst c_i, c_j, $i \neq j$, $i, j \in \{1, \ldots, n\}$ je určena její vzdálenost $d(c_i, c_j) \in \mathbb{N}$ a je zadáno číslo $D \in \mathbb{N}$.

Otázka: Existuje pořadí měst, v němž je má obchodní cestující projet, aby najetá vzdálenost byla nejvýš D? Přesněji, existuje permutace $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, pro kterou platí, že

$$\sum_{i=1,\ldots,n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi}(n), c_{\pi(1)}) \leq D?$$

Problém obchodního cestujícího je přirozeně optimalizační problém, protože ve skutečnosti je přirozené ptát se po co nejkratší cestě přes všechna města, uvedená verze je rozhodovací verzí a vypadá o něco jednodušeji, stačí jen najít pořadí měst, při kterém obchodní cestující najezdí nejvýš vzdálenost D. Nejen, že neumíme tento problém rozhodnout v polynomiálním čase, ale dokonce se řada vědců domnívá, že to ani nelze (byť ani to zatím neumíme ukázat). Na druhou stranu, pokud dostane obchodní cestující kandidáta na řešení, tedy dodá-li mu někdo konkrétní pořadí měst, není pro něj problém ověřit, zda dané pořadí splňuje požadovanou podmínku. K tomu stačí posčítat vzdálenosti hran v daném pořadí a porovnat součet s hodnotou D. Jde tedy o polynomiálně ověřitelný problém. Formálně definujeme tyto problémy s využitím pojmu polynomiálního verifikátoru.

Definice 9.4.2 Řekneme, že rozhodovací problém (čili jazyk) $A \subseteq \Sigma^*$ je *polynomiálně ověřitelný*, pokud existuje algoritmus $V(x, y)$, který pracuje v polynomiálním čase vzhledem k $|x|$ a pro který platí, že

$$A = \{x \mid (\exists y \in \Sigma^*)[V(x, y) \text{ přijme}]\}.$$

Algoritmu V budeme říkat *polynomiální verifikátor* (*polynomial verifier*). Řetězci y, pro který platí, že $V(x, y)$ přijme, říkáme *certifikát* pro řetězec x. ◂

Povšimněme si několika vlastností pojmů definovaných v definici 9.4.2.

- Vstup polynomiálního verifikátoru V je tvořen dvojicí řetězců x a y, přitom ale časovou složitost V měříme jen vzhledem k délce řetězce x, nikoli také vzhledem k délce řetězce y. Toto je trochu neobvyklé, protože obvykle měříme složitost algoritmu vzhledem k délce celého vstupu.
- Certifikát y pro řetězec x dosvědčuje, že $x \in A$. Uvážíme-li, že algoritmus $V(x, y)$ pracuje v čase polynomiálním vzhledem k $|x|$, nutně musí platit, že počet znaků y, které stihne při své práci $V(x, y)$ přečíst, je opět jen polynomiální v $|x|$. To znamená, že pro účely certifikátu nemá smysl uvažovat delší než polynomiálně dlouhé certifikáty. Hovoříme také o *polynomiálním certifikátu*.

- Konečně si povšimněme podobnosti s částečně rozhodnutelnými jazyky, zvláště pak s bodem (iv) věty 6.1.1. Podle tohoto bodu můžeme říci, že jazyk A je částečně rozhodnutelný, právě když existuje rozhodnutelný jazyk B, pro který platí, že

  $$A = \{x \mid (\exists y \in \Sigma^*)[\langle x, y \rangle \in B]\}.$$

  Můžeme tedy říci, že algoritmus rozhodující B zde hraje roli verifikátoru, který však nemusí pracovat v polynomiálním čase a řetězec y zde pak hraje roli certifikátu, který dokazuje, že řetězec x patří do jazyka A.

Nyní již můžeme definovat třídu NP.

Definice 9.4.3 (Třída NP) NP definujeme jako třídu polynomiálně ověřitelných jazyků. ◂

Třídu NP lze definovat i jiným způsobem, jako třídu jazyků, jež jsou přijímány nějakým nedeterministickým Turingovým strojem v polynomiálním čase. Odtud též pochází zkratka NP, tedy *nondeterministically polynomial*. Tuto definici si ukážeme v sekci 9.5.

9.5. Nedeterministické třídy složitosti

V sekci 4.1.2 jsme zavedli pojem nedeterministického Turingova stroje (NTS). Ve větě 4.1.16 jsme si ukázali, že každý nedeterministický Turingův stroj M lze převést na jednopáskový deterministický Turingův stroj M', který přijímá týž jazyk jako M. Podíváme se nyní na nedeterministický Turingův stroj z pohledu složitosti. Na rozdíl od deterministického Turingova stroje, přechodová funkce nedeterministického Turingova stroje nabízí pro každý displej hned množinu možných přechodů. Výpočet NTS $M = (Q, \Sigma, \delta, q_0, F)$ jsme definovali jako posloupnost konfigurací, která začíná v počáteční konfiguraci a každá další v posloupnosti pak odpovídá nějakému přechodu dle přechodové funkce. Výpočet je přijímající, pokud končí přijímající konfigurací (tj. konfigurací v níž je M v přijímajícím stavu) a navíc z poslední konfigurace není již možný žádný další přechod (tj. množina přechodů daných přechodovou funkcí δ je prázdná, výpočet v této konfiguraci končí). Vstup $x \in \Sigma^*$ je přijat strojem M, pokud existuje přijímající výpočet počínající v konfiguraci, kde na pásce je zapsán vstup x.

Připomenuvše si základní pojmy, můžeme přistoupit k definici základních nedeterministických tříd složitosti. Začneme popisem toho, co znamená když řekneme, že nedeterministický Turingův stroj M pracuje v omezeném čase nebo prostoru.

Definice 9.5.1 Nechť $f : \mathbb{N} \to \mathbb{N}$ je totální funkce a M je nedeterministický Turingův stroj.

- Řekneme, že M *pracuje v čase* $f(n)$, pokud každý výpočet M nad libovolným vstupem $x \in \Sigma^*$ délky $|x| = n$ skončí po provedení nejvýš $f(n)$ kroků.
- Řekneme, že M *pracuje v prostoru* $f(n)$, pokud každý výpočet M nad libovolným vstupem $x \in \Sigma^*$ délky $|x| = n$ skončí a během výpočtu využije nejvýše $f(n)$ buněk pracovní pásky. ◂

S pomocí těchto pojmů nyní definujeme dvě základní nedeterministické třídy složitosti.

Definice 9.5.2 Nechť $f : \mathbb{N} \to \mathbb{N}$ je totální funkce, pak definujeme následující třídy problémů:

NTIME($f(n)$) je třídou problémů přijímaných nedeterministickými Turingovými stroji v čase $O(f(n))$. Přesněji, jazyk $L \subseteq \Sigma^*$ patří do třídy NTIME($f(n)$), právě když existuje nedeterministický Turingův stroj M, který přijímá jazyk L a který pracuje v čase $O(f(n))$.

NSPACE($f(n)$) je třídou problémů přijímaných nedeterministickými Turingovými stroji v prostoru $O(f(n))$. Přesněji, jazyk $L \subseteq \Sigma^*$ patří do třídy NSPACE($f(n)$), právě když existuje nedeterministický Turingův stroj M, který přijímá jazyk L a který pracuje v prostoru $O(f(n))$. ◂

Poznámka 9.5.3 *Poznamenejme, že v literatuře se lze setkat i s definicemi, které se od té naší mírně liší. Často se podmínka počtu kroků a využitého prostoru vztahuje jen na přijímající výpočty. O těch nepřijímajících se potom nehovoří. Což dává dobrý smysl, neboť je potřeba si uvědomit, že v případě NTS nás zajímají pouze přijímající výpočty, nepřijímající nás nezajímají. Nám se bude hodit, že omezení kladená na čas nebo prostor se vztahují i na nepřijímající výpočty, jde však pouze o technický předpoklad, který může v některých místech naše úvahy zjednodušit.*

Nedeterminismus nám dovoluje zachytit existenční kvantifikaci. Třídu NP lze alternativně definovat následujícím způsobem.

Věta 9.5.4 *Platí, že*

$$\mathrm{NP} = \bigcup_{k=0}^{\infty} \mathrm{NTIME}(n^k). \tag{9.5}$$

Důkaz: Uvažme nejprve jazyk $L \in$ NP. Dle definice 9.4.3 existuje polynomiální verifikátor $V(x, y)$ pro jazyk L. Předpokládejme, že V pracuje v čase $p(|x|)$ pro nějaký polynom p. Popišme nedeterministický Turingův stroj M, který přijímá L v polynomiálním čase. Se vstupem x pracuje M ve dvou krocích.

1: Nedeterministicky zapiš na pásku řetězec y, jehož délka je nejvýš $p(|x|)$.
2: **if** $V(x, y)$ přijme **then**
3: **accept**
4: **else**
5: **reject**
6: **end if**

Uvažme naopak jazyk $L \in \bigcup_{k=0}^{\infty} \mathrm{TIME}(n^k)$. Z toho plyne, že existuje NTS M, který přijímá L v polynomiálním čase $p(|x|)$. Pro daný řetězec x platí, že je přijat strojem M, právě když existuje přijímající výpočet $M(x)$. Jde o posloupnost konfigurací $K_0^x, \ldots, K_t$, kde $t \leq p(|x|)$ a k zápisu každé konfigurace stačí řetězec délky $p(|x|)$. Jako certifikát y dosvědčující, že $x \in L$ můžeme tedy použít přímo tento výpočet. Verifikátor $V(x, y)$ pro jazyk L

pak bude jen ověřovat, jestli y kóduje přijímající výpočet $M(x)$ délky nejvýš $p(|x|)$. Toto ověření již lze provést deterministicky v polynomiálním čase vzhledem k $|x|$. □

Na závěr této sekce zaveďme dvě další nedeterministické prostorové třídy.

$$\text{NPSPACE} = \bigcup_{k=0}^{\infty} \text{NSPACE}(n^k) \tag{9.6}$$

$$\text{NL} = \text{NSPACE}(\log_2 n) \tag{9.7}$$

10. Vztahy mezi třídami složitosti

V této kapitole si ukážeme některé známé vztahy mezi třídami složitosti, jež jsme zavedli v kapitole 9. Začneme těmi nejzákladnějšími vztahy, dále si ukážeme Savičovu větu a věty o deterministické časové a prostorové hierarchii.

10.1. Vztahy mezi třídami

V této sekci si ukážeme některá základní tvrzení o vztahu mezi třídami složitosti, které jsme zavedli v předchozích kapitolách. Začneme těmi nejzákladnějšími vztahy, které povětšinou plynou z definic.

Věta 10.1.1 *Nechť $f : \mathbb{N} \to \mathbb{N}$ je totální funkce, pak*

$$\mathrm{TIME}(f(n)) \subseteq \mathrm{NTIME}(f(n)) \subseteq \mathrm{SPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n)).$$

Důkaz: První a třetí inkluze vyplývají triviálně z definic, neboť deterministický Turingův stroj je zvláštním případem stroje nedeterministického. Ukažme prostřední inkluzi $\mathrm{NTIME}(f(n)) \subseteq \mathrm{SPACE}(f(n))$. Uvažme jazyk $L \in \mathrm{NTIME}(f(n))$ a nedeterministický Turingův stroj M, který přijímá jazyk L a pracuje v čase $O(f(n))$. NTS M můžeme převést na deterministický Turingův stroj M' postupem z důkazu věty 4.1.16. Není těžké nahlédnout, že M' pracuje v prostoru $O(f(n))$ a tedy $L \in \mathrm{SPACE}(f(n))$.

Konstrukce M' postupuje v důkazu věty 4.1.16 ve dvou krocích, nejprve je zkonstruován třípáskový TS M'' a poté je na základě věty 4.1.12 převeden tento stroj na M'. Převod pomocí věty 4.1.12 zachovává využitý prostor, případně jej zvětšuje s konstantním násobkem (kde konstanta závisí na počtu pásek, ale nikoli na velikosti vstupu). Prostor využitý třípáskovým strojem M'' je (nepočítáme-li vstupní pásku) daný délkou výpočtu M, tedy omezený $O(f(n))$. □

Věta 10.1.1 ukazuje, že můžeme přejít od nedeterministického času k deterministickému prostoru. Uvažme nyní NTS M, který pracuje v prostoru $f(n)$. Budeme chtít zkonstruovat DTS M', který simuluje M. Při simulaci nad vstupem x nemůžeme vyloučit situaci, kdy je nutné projít všechny konfigurace, do kterých se může M při výpočtu nad x dostat. Odhadněme tedy nejprve, kolik takových konfigurací může být.

Lemma 10.1.2 *Nechť M je nedeterministický TS, který pracuje v prostoru $f(n)$, kde $f(n) \geq \log_2 n$. Potom existuje konstanta c_M, pro kterou platí, že pro libovolný vstup x délky n je počet různých konfigurací M v nichž se může NTS M nacházet při výpočtu nad vstupem x nejvýš $2^{c_M f(n)}$.*

Důkaz: Nechť $M = (Q, \Sigma, \delta, q_0, F)$. Budeme předpokládat, že M má strukturu popsanou v poznámce 9.2.4 tak, abychom mohli připustit prostor menší než lineární. M má tedy tři pásky — vstupní jen pro čtení, pracovní pásku pro čtení i zápis a výstupní pásku jen pro zápis, na níž se hlava pohybuje jen doprava. Mohli bychom uvažovat i obecnější případ, kdy M by měl k pracovních pásek, ale je to zbytečné. Redukce počtu pásek popsaná ve větě 4.1.12 zvětší prostor jen na násobek daný konstantou k, což bychom mohli zahrnout do konstanty c_M.

Konfigurace se skládá ze slova na pásce, poloh hlavy na vstupní a pracovní pásce a stavu, v němž se stroj M nachází. Délka vstupu je n a délka slova na pásce je nejvýš $f(n)$. Počet různých poloh hlavy v rámci vstupu je n (uvažujeme-li pro jednoduchost, že první a poslední znak vstupu je označen nějakou značkou). Počet různých poloh hlavy na pracovní pásce je nejvýš $f(n)$ a počet stavů je $|Q|$. Označíme-li C počet různých konfigurací, v nichž se M může nacházet při výpočtu nad vstupem x délky n, pak dostaneme

$$\begin{aligned} C &\leq |\Sigma|^{f(n)} \cdot n \cdot f(n) \cdot |Q| = 2^{f(n)\cdot\log_2|\Sigma|} \cdot 2^{\log_2 n} \cdot 2^{\log_2 f(n)} \cdot 2^{\log_2|Q|} = \\ &= 2^{f(n)\log_2|\Sigma|+\log_2 n+\log_2 f(n)+\log_2|Q|} \leq 2^{f(n)\log_2|\Sigma|+f(n)+f(n)+\log_2|Q|} \leq \\ &\leq 2^{f(n)\cdot(\log_2|\Sigma|+2+\log_2|Q|)}, \end{aligned}$$

kde druhá nerovnost platí díky tomu, že $f(n) \geq \log_2 n$ a $f(n) \geq \log_2 f(n)$, poslední nerovnost pak platí díky tomu, že $f(n) \geq \log_2 n \geq 1$ pro $n \geq 2$. Stačí tedy zvolit hodnotu konstanty $c_M = \log_2|\Sigma| + \log_2|Q| + 2$. □

Uvážíme-li tedy Turingův stroj M, který pracuje v prostoru $f(n) \geq \log_2 n$ a vstup x délky n, pak provede-li v nějakém výpočtu M více než $2^{c_M f(n)}$ kroků, nutně se musela zopakovat konfigurace. Je-li M deterministický, pak to znamená, že $M(x)$ se zacyklí. I v případě, kdy M je nedeterministický TS, nemá smysl uvažovat situaci, kdy se ve výpočtu zopakuje konfigurace (celou část výpočtu mezi dvěma výskyty téže konfigurace lze jistě zapomenout). Dá se tedy předpokládat, že M by mělo být lze nahradit deterministickým strojem M', který pracuje v čase $2^{c\cdot f(n)}$ pro nějakou konstantu c, která závisí na M. V kapitole 4.1.2 jsme zavedli pojem stromu výpočtu. Deterministický stroj M' by tedy mohl procházet stromem výpočtu M nad vstupem x do hloubky nebo do šířky. To by skutečně šlo provést, stačilo by vždy průchod větve ukončit v konfiguraci, která již byla zpracována. My však stroj M' popíšeme s pomocí grafu výpočtu. Graf výpočtu M nad vstupem x se od stromu výpočtu liší tím, že vrcholy odpovídající téže konfiguraci sloučíme do jednoho vrcholu. Tento pojem se nám bude hodit i dále při důkazu Savičovy věty.

Definice 4.1.14

Definice 10.1.3 (Graf konfigurací nedeterministického Turingova stroje) Nechť $M = (Q, \Sigma, \delta, q_0, F)$ je nedeterministický Turingův stroj. *Graf konfigurací $G_{M,x}$ stroje M nad vstupem x* (též *konfigurační graf*) definujeme jako orientovaný graf, jehož vrcholy $G_{M,x}$ jsou tvořeny možnými konfiguracemi výpočtu M nad vstupem x. Jsou-li K_1 a K_2 dvě konfigurace a tedy vrcholy $G_{M,x}$, pak (K_1, K_2) je hranou $G_{M,x}$, právě když z K_1 lze do K_2 přejít s pomocí přechodové funkce δ. ◂

Předpokládejme, že nedeterministický TS M pracuje v prostoru $f(n)$, kde $f(n) \geq \log_2 n$. Z lemmatu 10.1.2 plyne, že je-li $G_{M,x} = (V, E)$ grafem výpočtu nedeterministického stroje M nad vstupem x délky n, pak $|V| \leq 2^{c_M f(n)}$ a $|E| \leq 2^{2c_M f(n)}$ pro nějakou konstantu c_M, která závisí na Turingovu stroji M, ale nikoli na vstupu x. Výpočet stroje M nad vstupem x je posloupností konfigurací, která odpovídá tahu v grafu $G_{M,x}$, který začíná v počáteční konfiguraci. Pokud se ve výpočtu neopakuje konfigurace, pak tento odpovídá cestě v grafu $G_{M,x}$. Můžeme tedy říci, že $M(x)$ přijme, právě když v $G_{M,x}$ existuje cesta z počáteční konfigurace $M(x)$ do nějaké přijímající konfigurace. S tímto pozorováním již není těžké popsat deterministický stroj M', který bude simulovat M. Toho využijeme k důkazu následujícího tvrzení.

Věta 10.1.4 *Nechť $f(n)$ je funkce, pro kterou platí $f(n) \geq \log_2 n$. Pro každý jazyk L platí, že*

$$L \in \mathrm{NSPACE}(f(n)) \Rightarrow (\exists c_L \in \mathbb{N}) \left[L \in \mathrm{TIME}(2^{c_L f(n)}) \right].$$

Důkaz: Uvažme $L \in \mathrm{NSPACE}(f(n))$, potom existuje NTS $M = (Q, \Sigma, \delta, q_0, F)$, který přijímá L, tedy $L(M) = L$, a který pracuje v prostoru $O(f(n))$. Uvažme deterministický TS M', který průchodem grafu $G_{M,x} = (V, E)$ (do hloubky nebo do šířky) hledá cestu z počáteční konfigurace K_0^x se slovem x na vstupní pásce do nějaké přijímající konfigurace. Pokud takovou cestu najde, $M'(x)$ přijme, jinak odmítne. Platí, že $L(M') = L(M) = L$, navíc M' pracuje v polynomiálním čase vzhledem k $|V| \leq 2^{c_M f(n)}$ (dle lemmatu 10.1.2), tedy v čase $2^{c_L f(n)}$ pro nějakou konstantu c_L, která závisí na stroji M a implementaci průchodu do hloubky nebo do šířky.

Zbývá rozmyslet jeden technický detail. Stroj M' nezná funkci $f(n)$ a nemůže si spočítat její hodnotu, protože o funkci $f(n)$ nepředpokládáme, že je algoritmicky vyčíslitelná. To znamená, že M' nemůže nejprve zkonstruovat graf $G_{M,x}$, neboť neví, jak dlouhý může být řetězec kódující konfiguraci. Pro danou konfiguraci K je ovšem dle přechodové funkce δ dáno, které konfigurace jsou sousední, dokonce těch sousedních konfigurací je jen konstantně mnoho. Stroj M' si tedy bude graf $G_{M,x}$ generovat za běhu a bude si ukládat objevené konfigurace na pásku. Ve chvíli, kdy vygeneruje konfigurace, jež sousedí s aktuálně navštěvovanou konfigurací dle δ, porovná ji se seznamem objevených konfigurací. Z předpokladu je zaručeno, že seznam objevených konfigurací nemůže být delší než $2^{c_M f(n)}$. Práce M' končí ve chvíli, kdy objeví přijímající konfiguraci (pak přijme), nebo kdy již není možné vygenerovat další konfiguraci (všichni sousedi konfigurací v seznamu jsou již v seznamu). □

Je dobré si uvědomit, že z věty 10.1.4 nevyplývá, že $\mathrm{NSPACE}(f(n)) \subseteq \mathrm{TIME}(2^{cf(n)})$ pro nějakou konstantu c, protože konstanta c_L v tvrzení věty závisí na konkrétním jazyku L a pro každý jazyk může být jiná. Na druhou stranu můžeme zformulovat následující důsledek.

Důsledek 10.1.5 *Je-li $f(n)$ funkce, pro kterou platí $f(n) \geq \log_2 n$ a je-li $g(n)$ funkce, pro kterou platí $f(n) = o(g(n))$, pak*

$$\mathrm{NSPACE}(f(n)) \subseteq \mathrm{TIME}(2^{g(n)}).$$

Důkaz: Uvažme jazyk $L \in \text{NSPACE}(f(n))$. Dle věty 10.1.4 existuje konstanta $c_L \in \mathbb{N}$, pro kterou platí, že $L \in \text{TIME}(2^{c_L f(n)})$. Z předpokladu $f(n) = o(g(n))$ existuje $n_0 \in \mathbb{N}$ takové, že pro $n \geq n_0$ platí $c_L f(n) \leq g(n)$. Z toho plyne, že $\text{TIME}(2^{c_L f(n)}) \subseteq \text{TIME}(2^{g(n)})$ a tedy $L \in \text{TIME}(2^{g(n)})$. □

Věty 10.1.1 a 10.1.4 nám umožňují ukázat některé vztahy mezi třídami jež jsme si dříve zavedli.

Věta 10.1.6 *Platí následující inkluze*

$$\text{L} \subseteq \text{NL} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{NPSPACE} \subseteq \text{EXPTIME}.$$

Důkaz: Inkluze $\text{L} \subseteq \text{NL}$ a $\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{NPSPACE}$ plynou přímo z věty 10.1.1. Uvažme jazyk $L \in \text{NL} = \text{NSPACE}(\log_2 n)$, dle věty 10.1.4 existuje konstanta c_L, pro kterou platí $L \in \text{TIME}(2^{c_L \log_2 n}) = \text{TIME}(n^{c_L}) \subseteq P$, tedy $\text{NL} \subseteq P$. Uvažme jazyk $L \in \text{NPSPACE}$, tedy $L \in \text{NSPACE}(n^k)$ pro nějakou konstantu $k \in \mathbb{N}$. Dle důsledku 10.1.5 platí, že $\text{NSPACE}(n^k) \subseteq \text{TIME}(2^{n^{k+1}}) \subseteq \text{EXPTIME}$. □

Z věty 10.2.1, kterou si ukážeme v kapitole 10.2 plyne, že PSPACE = NPSPACE, proto se také obvykle v literatuře setkáme s třídou PSPACE a nikoli s třídou NPSPACE. U zbývajících inkluzí není známo, zda jsou ostré, ovšem některé z nich ostré jsou, neboť z vět o hierarchii, které si ukážeme v kapitole 10.3 plyne, že $\text{L} \subsetneq \text{PSPACE}$ a $\text{P} \subsetneq \text{EXPTIME}$.

10.2. Savičova věta

V této kapitole si ukážeme Savičovu větu, která ukazuje, že nedeterministický Turingův stroj M lze převést na deterministický Turingův stroj M', který přijímá týž jazyk a využívá jen kvadraticky více prostoru.

Věta 10.2.1 (Savičova věta) *Pro každou funkci $f(n) \geq \log_2 n$ platí, že*

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

Důkaz: K důkazu Savičovy věty využijeme graf konfigurací zavedený v definici 10.1.3 podobně jako v důkazu věty 10.1.4. Uvažme jazyk $L \in \text{NSPACE}(f(n))$, existuje tedy nedeterministický Turingův stroj $M = (Q, \Sigma, \delta, q_0, F)$, který přijímá jazyk $L = L(M)$ a který pracuje v prostoru $O(f(n))$. Popíšeme deterministický Turingův stroj M', který přijímá jazyk L v prostoru $O(f^2(n))$, tedy $L \in \text{SPACE}(f^2(n))$.

Uvažme vstup $x \in \Sigma^*$. Víme, že $x \in L$, právě když existuje cesta v grafu konfigurací $G_{M,x}$ z počáteční konfigurace K_0^x se vstupem x do nějaké přijímající konfigurace. Pro jednoduchost budeme předpokládat, že M má jedinou přijímající konfiguraci K_F (tj. před přechodem do jediného přijímajícího stavu q_1 stroj M vymaže obsah pásky a vrátí se na nejlevější pozici využitého prostoru). Potom stačí ověřit, zda existuje v grafu $G_{M,x} = (V, E)$ cesta z konfigurace K_0^x do konfigurace K_F. V důkazu věty 10.1.4 nám stačilo použít průchod grafem do šířky či do hloubky k nalezení této cesty. S tím si zde ovšem

nevystačíme, neboť průchod do šířky (BFS) i do hloubky (DFS) vyžadují ke své práci prostor lineární ve velikosti grafu $G_{M,x}$. Dle lemmatu 10.1.2 platí, že $|V| \leq 2^{c_M f(n)}$, kde $n = |x|$, pro vhodnou konstantu c_M závislou jen na stroji M a nikoli na vstupu x. Ovšem z důkazu tohoto odhadu také plyne, že byť jde o horní odhad, existují Turingovy stroje, kde tento odhad je těsný. Jinými slovy graf $G_{M,x}$ je příliš velký na to, aby si jej stroj M' mohl zkonstruovat a uložit na pásce, navíc není možné použít průchod do šířky ani do hloubky tímto grafem k ověření existence cesty z K_0^x do K_F.

Budeme tedy postupovat jinak. Předpokládejme na chvíli, že známe hodnotu $t = c_M f(n)$ (byť o funkci f nepředpokládáme ani to, že by byla algoritmicky vyčíslitelná, s čímž si budeme muset v nějakou chvíli poradit). To znamená, že $|V| \leq 2^t$, a tedy délka nejdelší cesty mezi libovolnými dvěma vrcholy $G_{M,x}$ je nejvýš 2^t. To nabízí následující postup, který nám umožní ušetřit paměť: Cesta z K_0^x do K_F délky nejvýš 2^t existuje právě tehdy, když existuje „prostřední" konfigurace K, pro kterou platí, že v $G_{M,x}$ existují cesty z K_0^x do K a z K do K_F poloviční délky, tedy nejvýš 2^{t-1}. Tyto dvě podmínky můžeme ověřit rekurzivně, přičemž můžeme využít toho, že pro oba testy (cesty z K_0^x do K a cesty z K do K_F) můžeme použít touž paměť. Rekurze se zastaví ve chvíli, kdy $t = 0$, neboť potom je již ověření existence cesty snadné, máme-li k dispozici přechodovou funkci δ nedeterministického Turingova stroje M. Stroj M' samozřejmě nemůže dopředu vědět, kterou konfiguraci K zvolit jako prostřední, bude tedy postupně zkoušet všechny vrcholy $G_{M,x}$. Pokud známe hodnotu t, stačí zkoušet všechny řetězce dané délky, jež mohou kódovat konfiguraci. Takto dostaneme rekurzivní algoritmus, kde hloubka rekurze je $O(f(n))$ a navíc v každé instanci rekurzivně volané funkce je využitá paměť $O(f(n))$, tedy dohromady bude využitá paměť $O(f^2(n))$. Na závěr si musíme však poradit s tím, že M' nezná hodnotu $t = c_M f(n)$ a nemůžeme ani předpokládat, že si ji může spočítat, protože o funkci $f(n)$ nepředpokládáme, že by byla algoritmicky vyčíslitelná.

Celý postup si nyní popíšeme podrobněji. Začneme tím, že si popíšeme funkci testující dosažitelnost v grafu, která tvoří jádro celého algoritmu, tato funkce je popsána v Algoritmu 10.2.1.

Korektnost algoritmu 10.2.1 byla zdůvodněna v předchozím textu. Je potřeba si uvědomit, že M' zná přechodovou funkci M, a tedy může ověřit, zda $(K_1, K_2) \in E$. Stačí tedy, aby M' zavolal Dosažitelná(K_0^x, K_F, t_0) pro vhodnou počáteční hodnotu t_0. Cyklus na řádku 8 pak implementujeme postupným generováním binárních řetězců délky t_0. Ideální by jistě bylo rovnou použít hodnotu $t_0 = c_M f(n)$, ovšem hodnotu $f(n)$ stroj M' nezná a nemůže ji ani určit. Stroj M' tedy bude zkoušet postupně různé počáteční hodnoty $t_0 = 1, 2, 3, \ldots$. Pokud pro danou hodnotu t_0 volání Dosažitelná(K_0^x, K_F, t_0) uspěje, bylo ověřeno, že cesta z K_0^x do K_F v grafu $G_{M,x}$ existuje. Pokud toto volání selže, ověří M' ještě, zda existuje konfigurace K, jejíž délka je větší než t_0 a která je dosažitelná z K_0^x. Pokud ano, pokračuje hledání s vyšší hodnotou t_0, v opačném případě test dosažitelnosti končí. Tento postup je popsán v algoritmu 10.2.2.

Maximální hodnotou t_0, jež musí algoritmus 10.2.2 vzít do úvahy, je hodnota $t_0 = c_M f(n)$. Prostorové nároky jedné instance funkce Dosažitelná jsou tedy $O(f(n))$. Hloubka rekurze každého volání je daná třetím parametrem a je tedy nejvýš $O(f(n))$. Dohromady dostáváme, že prostorové nároky algoritmu 10.2.2 jsou $O(f^2(n))$. Jelikož se jedná o deterministický algoritmus a platí $L = L(M')$, dostáváme, že $L \in \mathrm{SPACE}(f^2(n))$. □

Algoritmus 10.2.1 Funkce Dosažitelná(K_1, K_2, t)

Vstup: Konfigurace K_1 a K_2, limit t
Výstup: **true**, pokud v $G_{M,x}$ existuje cesta z K_1 do K_2 délky nejvýš 2^t, jinak **false**

```
1: if t = 0 then
2:     if K_1 = K_2 or (K_1, K_2) ∈ E then
3:         return true
4:     else
5:         return false
6:     end if
7: end if
8: for all K ∈ V do
9:     if Dosažitelná(K_1, K, t − 1) and Dosažitelná(K, K_2, t − 1) then
10:        return true
11:    end if
12: end for
13: return false
```

Algoritmus 10.2.2 Práce M' se vstupem x

```
1: for all t_0 = 1, 2, 3, ... do
2:     if Dosažitelná(K_0^x, K_F, t_0) then
3:         accept
4:     end if
5:     k ← maximální počet políček na pracovní pásce v nějaké konfiguraci reprezentované pomocí t_0 bitů
6:     for all konfigurace K využívající k + 1 políček na pracovní pásce do
7:         if Dosažitelná(K_0^x, K, t_0 + 1) then
8:             continue
9:         end if
10:    end for
11:    reject
12: end for
```

Ze Savičovy věty plyne následující důsledek, který jsme již zmiňovali.

Důsledek 10.2.2 NPSPACE = PSPACE.

V literatuře se ve znění Savičovy věty často předpokládá, že funkce f je prostorově konstruovatelná, případně vyčíslitelná v prostoru $f(n)$. Za tohoto předpokladu je možné, aby si M' na začátku spočítal hodnotu $f(n)$, protože konstanta c_M je závislá na M, můžeme předpokládat, že ji M' zná a tedy může rovnou určit ten správný limit t_0 pro volání funkce Dosažitelná, což zjednoduší algoritmus 10.2.2. Pojem prostorové konstruovatelnosti si zavedeme v sekci 10.3, neboť jej budeme potřebovat ve větě o prostorové hierarchii.

10.3. Věty o hierarchii

Intuitivně bychom řekli, že pokud nějakému výpočetnímu stroji umožníme využít více prostředků pro výpočet, měl by být schopen spočítat více. V případě Turingových strojů pracujeme se dvěma prostředky, velikostí paměti a časem, po který necháme stroj běžet, od toho se odvíjí třídy TIME($f(n)$) a SPACE($f(n)$). V této kapitole si ukážeme, že zmíněná intuice je platná, tedy alespoň za určitých předpokladů, které jsou však velmi realistické.

10.3.1. Deterministická prostorová hierarchie

Nejprve se budeme věnovat prostoru. Uvažme funkci $f(n)$ a funkci $g(n) = o(f(n))$. Naším cílem je ukázat, že SPACE($g(n)$) $\subsetneq$ SPACE($f(n)$). K tomu potřebujeme ukázat, že existuje jazyk L, pro který existuje Turingův stroj M, který přijímá L, tedy $L = L(M)$ v prostoru $O(f(n))$, ale neexistuje žádný Turingův stroj M', který by přijímal L v prostoru $O(g(n))$. Abychom však byli schopni uvedené tvrzení dokázat, budeme se muset omezit na funkce $f(n)$, které splňují následující předpoklad: K výpočtu hodnoty $f(n)$ postačuje prostor, který je úměrný velikosti výstupu. Přesněji tento pojem definujeme následujícím způsobem.

Definice 10.3.1 Funkci $f : \mathbb{N} \to \mathbb{N}$, kde $f(n) \geq \log_2 n$, nazveme **prostorově konstruovatelnou**, je-li funkce, která zobrazuje 1^n na binární reprezentaci $f(n)$ vyčíslitelná v prostoru $O(f(n))$. ◂

Všechny běžné funkce, které používáme k měření složitosti jsou prostorově konstruovatelné. Jde například o polynomy, exponenciály, funkce využívající logaritmy jako $\log_2 n$, $n \log_2 n$ a další. Omezíme-li se na prostorově konstruovatelné funkce, neztrácíme tedy příliš. Je-li funkce prostorově konstruovatelná, znamená to, že při výpočtu můžeme funkci využít k vymezení nebo alokaci prostoru. Je-li dán vstup x s $|x| = n$, můžeme na začátku výpočtu určit hodnotu $f(n)$ a následně vyznačit na pásce prostor této velikosti. Pro tento způsob využití je také pojem prostorové konstruovatelnosti navržen. Je-li totiž hodnota n dána délkou vstupního řetězce, dává smysl počítat hodnotu funkce $f(n)$ se vstupem v tomto tvaru, definice vyžaduje řetězec délky n na vstupu složený ze samých

jedniček, který z x vytvoříme jednoduše náhradou každého znaku znakem 1. Zmiňme, že v literatuře se definice prostorové konstruovatelnosti vyskytuje v různých podobách, někdy se vyžaduje, aby existoval Turingův stroj, který při výpočtu nad vstupem x využije přesně $f(n)$ buněk pracovní pásky, ale existují i jiné varianty. V případě Turingových strojů lze obvykle ukázat ekvivalenci mezi různými definicemi, definice 10.3.1 nabízí vcelku jednoduché použití, je podle ní například vcelku přímočaré ukázat, že polynomy a další funkce jsou prostorově konstruovatelné.

Nyní již můžeme zformulovat větu o deterministické prostorové hierarchii.

Věta 10.3.2 (Věta o deterministické prostorové hierarchii) *Pro každou prostorově konstruovatelnou funkci $f : \mathbb{N} \to \mathbb{N}$ existuje jazyk A, který je rozhodnutelný v prostoru $O(f(n))$, nikoli však v prostoru $o(f(n))$.*

Důkaz: Popíšeme Turingův stroj N, pro který bude platit, že pracuje v prostoru $O(f(n))$, ale neexistuje žádný deterministický Turingův stroj, který by přijímal týž jazyk $A = L(N)$ v prostoru $o(f(n))$. Při popisu algoritmu stroje N využijeme techniky diagonalizace. Pro každý deterministický Turingův stroj M, který pracuje v prostoru $o(f(n))$ musí platit, že A se od $L(M)$ liší alespoň na jednom vstupu. Přirozeným kandidátem na vstup, v němž se mají tyto dva jazyky lišit, je kód stroje M, tedy $\langle M \rangle$, tento krok využívá zmíněnou diagonalizaci. Nestačí však uvažovat jediný řetězec, na kterém se budou A a $L(M)$ lišit, protože nám jde o asymptotickou složitost daného algoritmu. Pokud by platilo že, A se od $L(M)$ liší právě jen na vstupu $\langle M \rangle$, pak jednoduchou úpravou, která by ošetřila tuto výjimku, bychom dostali Turingův stroj, který přijímá A a pracuje v témž prostoru jako $\langle M \rangle$. Je proto nutné uvažovat řetězce různých délek. My budeme uvažovat řetězce tvaru $\langle M \rangle 10^*$. Aby mohl stroj N rozhodnout, má-li přijmout vstup tvaru $\langle M \rangle 10^*$, potřebuje na tomto vstupu odsimulovat stroj M a rozhodnout opačně. Jsou-li prostorové nároky stroje M určeny funkcí $g(n) = o(f(n))$, pak k simulaci je potřeba $c_M g(n)$ buněk pracovní pásky, kde c_M je konstanta závislá na stroji M. Vzhledem k tomu, že N může využít prostor $O(f(n))$, má N dost prostoru k dokončení simulace na dost dlouhých vstupech, tedy i na nějakém vstupu tvaru $\langle M \rangle 10^*$, ovšem pouze za předpokladu, že výpočet M nad vstupem x skončí, tedy $M(x)\downarrow$. Pokud $M(x)\uparrow$, může stroj N využít toho, že počet konfigurací, v nichž se výpočet $M(x)$ může nacházet je dle lemmatu 10.1.2 omezený a N tedy podle počtu provedených kroků může poznat, zda se výpočet $M(x)$ zacyklil.

Algoritmus 10.3.1 popisuje práci stroje N se vstupem x. Položme $A = L(N)$. Není těžké nahlédnout, že N pracuje v prostoru $O(f(n))$, výpočet hodnoty $f(n)$, kde $n = |x|$, v kroku 5 lze provést v prostoru $O(f(n))$, neboť funkce $f(n)$ je dle předpokladu prostorově konstruovatelná. Všechny další kroky probíhají ve vyznačeném prostoru $f(|x|)$ buněk. Počítadlo kroků se do daného prostoru vejde také, uvážíme-li, že abeceda N může být větší než binární (počítadlo může být na jiné stopě pásky, než vstup a simulace). Celkově tedy dostáváme, že A je rozhodnutelný v prostoru $O(f(n))$.

Předpokládejme nyní sporem, že A je rozhodnutelný v prostoru $o(f(n))$. Existuje tedy deterministický Turingův stroj M, který rozhoduje jazyk A v prostoru $g(n)$, kde $g(n) = o(f(n))$. Simulace výpočtu $M(x)$ probíhá podobně jako simulace univerzálním Turingovým strojem. K simulaci tedy N potřebuje $c_M g(n)$ buněk pracovní pásky, kde c_M je konstanta závislá na stroji M. Přesněji c_M určíme na základě lemmatu 10.1.2. Stroji N

Algoritmus 5.2.1

Algoritmus 10.3.1 Práce stroje N se vstupem x

Vstup: Vstupní řetězec $x \in \Sigma^*$.

1: **if** $x \neq \langle M \rangle 10^*$ pro nějaký TS M **then**
2: **reject**
3: **end if**
4: Na základě prostorové konstruovatelnosti urči hodnotu funkce $f(|x|)$
5: Vyznač $f(|x|)$ buněk na pracovní pásce, pokud v kterémkoli z dalších kroků hlava N opustí vymezený prostor, **reject**.
6: Simuluj výpočet stroje $M(x)$ a počítej kroky simulace, pokud je odsimulováno více než $2^{f(n)}$ kroků, **reject**.
7: **if** M přijal **then**
8: **reject**
9: **else**
10: **accept**
11: **end if**

stačí c_M buněk k zakódování jednoho políčka pásky M. Protože $g(n) = o(f(n))$, existuje přirozené číslo n_0, pro které platí, že $c_M g(n_0) \leq f(n)$. Uvažme nyní vstup $x = \langle M \rangle 10^{n_0}$. Simulace $M(x)$ strojem N doběhne až do konce, neboť na ni stačí vyznačený prostor a počet kroků je též menší než $2^{f(n)}$, neboť dle předpokladu je výpočet $M(x)$ konečný. Na základě podmíněného příkazu na řádce 7 tak dostáváme, že $x \in L(M)$ právě když $x \notin L(N)$. Z toho plyne, že $L(M) \neq A$, a to je spor s předpokladem. Platí tedy, že neexistuje Turingův stroj, který by A rozhodl v prostoru $o(f(n))$. □

Z věty 10.3.2 vyplývají následující důsledky.

Důsledek 10.3.3

(*i*) *Jsou-li* $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ *funkce, pro které platí, že* $f_1(n) \in o(f_2(n))$ *a* f_2 *je prostorově konstruovatelná, potom*

$$\mathrm{SPACE}(f_1(n)) \subsetneq \mathrm{SPACE}(f_2(n)).$$

(*ii*) *Pro každá dvě reálná čísla* $0 \leq \epsilon_1 < \epsilon_2$ *platí, že*

$$\mathrm{SPACE}(n^{\epsilon_1}) \subsetneq \mathrm{SPACE}(n^{\epsilon_2}).$$

(*iii*) *Platí* $\mathrm{NL} \subsetneq \mathrm{PSPACE} \subsetneq \mathrm{EXPSPACE} = \bigcup_{k \in \mathbb{N}} \mathrm{SPACE}(2^{n^k})$.

Důkaz: Tvrzení (i) vyplývá přímo z věty 10.3.2. Uvažme nyní dvě reálná čísla $0 \leq \epsilon_1 < \epsilon_2$. Platí jistě $n^{\epsilon_1} = o(n^{\epsilon_2})$, musíme však ověřit podmínku prostorové konstruovatelnosti. Není těžké nahlédnout, že je-li ϵ_2 ve skutečnosti celé číslo, pak n^{ϵ_2} je prostorově konstruovatelná funkce. Trochu obtížnější, byť stále možné, je ukázat, že n^{ϵ_2} je prostorově konstruovatelná funkce i v případě, kdy ϵ_2 je racionální číslo. Je-li ϵ_2 iracionální číslo, pak z hustoty racionálních čísel existuje racionální číslo ϵ_2', pro které platí $\epsilon_1 < \epsilon_2' < \epsilon_2$.

Protože ϵ_2' je číslo racionální, plyne z předchozích úvah, že $\mathrm{SPACE}(n^{\epsilon_1}) \subsetneq \mathrm{SPACE}(n^{\epsilon_2'}) \subseteq \mathrm{SPACE}(n^{\epsilon_2})$. Dostáváme tedy tvrzení (ii).

Věta 10.2.1

Připomeňme si, že $\mathrm{NL} = \mathrm{NSPACE}(\log_2 n)$, dle Savičovy věty platí $\mathrm{NSPACE}(\log n) \subseteq \mathrm{SPACE}((\log_2 n)^2)$, dle věty 10.3.2 dostáváme tedy $\mathrm{NL} \subseteq \mathrm{SPACE}((\log n)^2) \subsetneq \mathrm{SPACE}(n) \subsetneq \mathrm{PSPACE}$. Z definice třídy PSPACE dostáváme, že $\mathrm{PSPACE} \subseteq \mathrm{SPACE}(2^n)$, a tedy z věty 10.3.2 dostáváme $\mathrm{PSPACE} \subsetneq \mathrm{SPACE}(2^{n^2})$. Dohromady tedy dostáváme tvrzení (iii).□

Větu 10.3.2 jsme ukazovali pro prostorové třídy, jež jsme zavedli pro výpočetní model jednopáskového deterministického Turingova stroje. Je dobré si však uvědomit, že volba modelu v tomto případě nehraje podstatnou roli. Větu 10.3.2 by bylo možné ukázat i v případě, kdy bychom uvažovali vícepáskové Turingovy stroje nebo RAM. Větu 10.3.2 je možné ukázat i pro nedeterministické třídy prostorové složitosti, v tomto případě je situace složitější, neboť nedeterministické Turingovy stroje nenabízejí jednoduchý způsob pro negaci odpovědi.

10.3.2. Deterministická časová hierarchie

Analogii věty 10.3.2 můžeme ukázat i v případě časových tříd složitosti, i když tvrzení, které v tomto případě ukážeme, je o něco slabší. Budeme postupovat velmi podobně jako v kapitole 10.3.1. Omezíme se na funkce, které jsou časově konstruovatelné, což je pojem analogický prostorové konstruovatelnosti, avšak nyní omezujeme čas výpočtu, nikoli prostor.

Definice 10.3.4 Funkci $f : \mathbb{N} \to \mathbb{N}$, kde $f(n) = \Omega(n \log_2 n)$, nazveme **časově konstruovatelnou**, je-li funkce, která zobrazuje 1^n na binární reprezentaci $f(n)$ vyčíslitelná v čase $O(f(n))$. ◂

I v tomto případě platí, že všechny běžné funkce, které používáme k měření složitosti, jsou časově konstruovatelné. Jde například o polynomy (vyššího stupně než 1), exponenciály, funkce využívající logaritmy jako $n \log_2 n$, ale i funkce s odmocninami jako $\lceil n\sqrt{n} \rceil$ a další. Na rozdíl od prostorové konstruovatelnosti, v případě časové konstruovatelnosti vyžadujeme, aby platilo, že $f(n) = \Omega(n \log_2 n)$. Tento silnější požadavek je zde proto, že výstupem je binární zápis hodnoty funkce, zatímco vstupní hodnota n je zapsána v podobě 1^n. Na jednopáskovém Turingovu stroji potřebujeme k převodu řetězce 1^n do binárního zápisu čísla n čas $\Omega(n \log_2 n)$. Proto nepovažujeme funkci $f(n) = n$ za časově konstruovatelnou. Jak uvidíme, není tento předpoklad omezující. Požadovat binární zápis hodnoty funkce přitom dává dobrý smysl, který plyne ze způsobu použití časové konstruovatelnosti. Je-li funkce $f(n)$ časově konstruovatelná, můžeme totiž na začátku algoritmu určit hodnotu $f(n)$ a inicializovat binární čítač kroků. Poté můžeme počítat kroky a výpočet zastavit při dosažení $f(n)$ kroků. Pro operace s čítačem (například přesun, zvýšení či snížení hodnoty o 1) nám stačí $O(\log_2 f(n))$ kroků právě díky tomu, že hodnota $f(n)$ je zapsaná binárně. Tento způsob použití využijeme při důkazu věty o deterministické časové hierarchii.

Je třeba zmínit, že definice 10.3.4 není jedinou možností, jak pojem časové konstruovatelnosti zavést. V literatuře najdeme různé způsoby, které také uvažují různá omezení

na funkci $f(n)$. Definice 10.3.4 má výhodu v tom, že je jednoduchá a slouží přesně účelu, ke kterému chceme pojem časové konstruovatelnosti využít.

Nyní již můžeme zformulovat větu o deterministické časové hierarchii.

Věta 10.3.5 (Věta o deterministické časové hierarchii) *Pro každou časově konstruovatelnou funkci $f : \mathbb{N} \to \mathbb{N}$ existuje jazyk A, který je rozhodnutelný v čase $O(f(n))$, nikoli však v čase $o(f(n)/\log_2 f(n))$.*

Důkaz: Postup důkazu je analogický důkazu věty 10.3.2 o deterministické prostorové hierarchii. Sestrojíme Turingův stroj N, pro který bude platit, že pracuje v čase $O(f(n))$, ale neexistuje žádný deterministický Turingův stroj, který by přijímal týž jazyk $A = L(N)$ v čase $o(f(n)/\log_2 f(n))$. Struktura stroje N je podobná stroji popsanému v rámci důkazu věty 10.3.2. Stroj N opět předpokládá vstupní řetězec x ve tvaru $x = \langle M \rangle 10^*$ pro nějaký Turingův stroj M. Jádro práce stroje N pak spočívá v simulaci výpočtu $M(x)$ s tím, že pokud výpočet doběhne v čase $f(n)/\log_2 f(n)$, zneguje N jeho odpověď. Stroj N tedy provádí simulaci a navíc počítá kroky, toto je v případě omezeného času složitější, než v případě omezeného prostoru. V případě času si stroj N nevystačí s konstantním počtem kroků na simulaci jednoho kroku výpočtu $M(x)$ a současně aktualizaci počítadla kroků, proto je nutné požadovat logaritmický odstup uvažovaných funkcí.

Algoritmus 10.3.2 Práce stroje N se vstupem x

Vstup: Vstupní řetězec $x \in \Sigma^*$.

1: **if** $x \neq \langle M \rangle 10^*$ pro nějaký TS M **then**
2: **reject**
3: **end if**
4: Na základě časové konstruovatelnosti urči hodnotu funkce $f(|x|)$ a inicializuj binární čítač hodnotou $f(|x|)/\log_2 f(|x|)$. Před každým krokem ve zbytku výpočtu proveď snížení hodnoty čítače o 1, pokud hodnota čítače dosáhne hodnoty 0, **reject**.
5: Simuluj výpočet $M(x)$.
6: **if** M přijal **then**
7: **reject**
8: **else**
9: **accept**
10: **end if**

Algoritmus 10.3.2 popisuje práci stroje N se vstupem x. Položme $A = L(N)$. Kroky 1 až 4 lze provést v čase $O(f(n))$, kde $n = |x|$. Rozmysleme si nejprve, jak bude probíhat krok 5, tedy simulace výpočtu $M(x)$ spolu s dekrementací čítače po každém kroku. Pro simulaci kroku výpočtu M, musí N znát aktuální stav stroje M, symbol pod hlavou stroje M a musí mít přístup k přechodové funkci stroje M zapsané v kódu $\langle M \rangle$. Tyto tři údaje potřebuje mít N na pásce blízko sebe, protože pokud by byl například stav zapsaný na druhém konci pásky než je $\langle M \rangle$, znamenalo by to, že pro určení následující instrukce potřebuje N projít celou svou páskou mnohokrát.

Připomeňme si, že při konstrukci univerzálního Turingova stroje v kapitole 5.2.3 jsme každé části výše uvedených dat vyhradili zvláštní pásku. Nyní ovšem popisujeme N

jako jednopáskový Turingův stroj, není možné tedy mít stav zapsaný na jiné pásce než $\langle M \rangle$ a než obsah pracovní pásky M. Toto jsme při popisu univerzálního Turingova stroje vyřešili použitím generického převodu k-páskového Turingova stroje na 1-páskový popsaného v důkazu věty 4.1.12. Nicméně tento postup není pro nás dostačující, protože použitím postupu z důkazu věty 4.1.12 by provedení jednoho kroku simulace M zabralo čas úměrný velikosti využitého prostoru stroje M, tedy až $f(n)/\log_2 f(n)$. Tím bychom ovšem nedosáhli požadovaného výsledku.

Musíme proto postupovat jinak. Pásku N si rozdělíme na tři stopy (podobně jako v důkazu věty 4.1.12). Na první stopě bude zakódována pracovní páska M. Na druhé stopě bude zapsán aktuální stav výpočtu $M(x)$ a přechodová funkce zakódovaná pomocí $\langle M \rangle$. Na třetí stopě bude zapsána aktuální hodnota čítače kroků. Podstatné je, že obsahy stop budou zarovnané. Poloha hlavy na pásce odpovídá poloze hlavy na pásce M, tedy na první stopě. Dále je zachován následující invariant: Na začátku simulace jednoho kroku M obsah druhé stopy vždy začíná pod hlavou.

Odmysleme si zatím práci s čítačem a popišme kroky simulace. Během inicializace dojde k zakódování vstupu x ve vhodném tvaru na první stopě, na druhou stopu je překopírován kód $\langle M \rangle$ a stav číslo 0 (tedy číslo počátečního stavu). Na inicializaci stačí čas $O(n)$. Dále v cyklu probíhá simulace kroků výpočtu $M(x)$. Při simulaci jednoho kroku nejprve N najde v kódu $\langle M \rangle$ vhodnou instrukci a poté ji provede. Pokud instrukce zahrnuje pohyb hlavou, dojde k posunu kódu $\langle M \rangle$ tak, aby začínal pod hlavou na začátku simulace dalšího kroku. Dohromady lze simulaci jednoho kroku výpočtu M provést pomocí d kroků N, kde d je konstanta závislá na stroji M ($\langle M \rangle$ je sice součástí vstupu, ale v další argumentaci budeme uvažovat fixní M, tedy d je možné považovat za konstantu).

Zbývá rozmyslet si, jak bude probíhat práce s čítačem. Hodnota čítače bude zapsána na třetí stopě pásky s tím, že bude začínat vždy pod hlavou. To znamená, že po provedení každého kroku stroje N bude v případě nutnosti posunut obsah čítače. Při tom bude provedena i jeho dekrementace. K posunu a dekrementaci čítače postačuje $O(\log_2 f(n))$ kroků, neboť tolik bitů čítač zabírá. Dle kroku 4 dostáváme, že N vykoná v následujících krocích $f(n)/\log_2 f(n)$ instrukcí, po provedení každé z nich provede dekrementaci a posun čítače, tedy dohromady provede $O(f(n))$ instrukcí. Kroky 1 až 4 lze provést v čase $O(f(n))$, celkově tedy N pracuje v čase $O(f(n))$.

Předpokládejme nyní sporem, že M je Turingův stroj, který přijímá jazyk A v čase $g(n) = o(f(n)/\log_2 f(n))$. Z popisu simulace plyne, že simulaci výpočtu M se vstupem x provede N v čase $dg(n)\log_2 f(n)$, kde d je konstanta závislá na stroji M, přesněji na jeho kódu $\langle M \rangle$. Protože $g(n) = o(f(n)/\log_2 f(n))$, existuje $n_0 \in \mathbb{N}$, takové, že $dg(n) \leq f(n)/\log_2 f(n)$ pro každé $n \geq n_0$. Uvažme nyní vstup $x = \langle M \rangle 10^{n_0}$, pro který platí, že $n = |x| > n_0$. Stroj N tedy simulaci $M(x)$ dokončí v rámci $dg(n)$ simulovaných kroků než hodnota čítače dosáhne 0. Tedy výpočet N se dostane k provedení kroku 6. Z toho plyne, že $x \in L(M)$ právě když $x \notin L(N) = A$. Tedy $A \neq L(M)$, což je spor s předpokladem. Žádný takový stroj M tedy neexistuje. □

Z věty 10.3.5 vyplývají následující důsledky.

Důsledek 10.3.6

(*i*) *Jsou-li* $f_1, f_2 : \mathbb{N} \to \mathbb{N}$ *funkce, pro které platí, že* $f_1(n) \in o(f_2(n)/\log_2 f_2(n))$ *a* f_2 *je časově konstruovatelná, potom*

$$\mathrm{TIME}(f_1(n)) \subsetneq \mathrm{TIME}(f_2(n)).$$

(*ii*) *Pro každá dvě reálná čísla* $1 \leq \epsilon_1 < \epsilon_2$,

$$\mathrm{TIME}(n^{\epsilon_1}) \subsetneq \mathrm{TIME}(n^{\epsilon_2}).$$

(*iii*) $P \subsetneq EXPTIME$.

Důkaz: Tvrzení (i) plyne přímo z věty 10.3.5. Podobně jako v případě důsledku 10.3.3 není těžké nahlédnout, že je-li ϵ_2 přirozené číslo větší než 1, pak n^{ϵ_2} je časově konstruovatelná funkce. Trochu obtížněji je možné ukázat, že n^{ϵ_2} je časově konstruovatelná funkce i pokud $\epsilon_2 > 1$ je racionální číslo. Dále můžeme využít hustoty racionálních čísel a toho, že $n^{\epsilon_1} = o(n^{\epsilon_2}/\log_2 n)$ k důkazu (ii) (podobně jako při důkazu (ii) v důkazu důsledku 10.3.3). Z definice třídy P dostáváme, že $\mathrm{P} \subseteq \mathrm{TIME}(2^n)$. Zřejmě platí, že $2^n = o(2^{n^2}/n)$, a tedy z věty 10.3.5 dostáváme, že $\mathrm{P} \subsetneq \mathrm{EXPTIME}$, tedy (iii). □

Podobně jako v případě prostoru, větu 10.3.2 jsme ukazovali pro časové třídy, jež jsme zavedli pro výpočetní model jednopáskového deterministického Turingova stroje. Analogickou větu lze ukázat i pro k-páskové Turingovy stroje. Pro nedeterministické časové třídy dokonce stačí slabší předpoklad $f_1(n+1) = o(f_2(n))$, aby platila ostrá inkluze $\mathrm{NTIME}(f_1(n)) \subsetneq \mathrm{NTIME}(f_2(n))$. V případě RAM stačí předpoklad $f_1(n) = o(f_2(n))$.

11. Polynomiální převoditelnost a úplnost

11.1. Polynomiální převoditelnost a její vlastnosti

S principem převoditelnosti stejně jako s úplností jsme se již setkali v kapitole 7. Tehdy nás zajímala převoditelnost, která zachovávala algoritmickou řešitelnost a k tomu nám stačilo, že převádějící funkce byla algoritmicky vyčíslitelná. Nyní chceme, aby převádějící funkce zachovávala polynomiální řešitelnost, a proto od ní budeme požadovat, aby byla vyčíslitelná v polynomiálním čase.

Definice 11.1.1 Nechť A a B jsou dva jazyky nad abecedou Σ. Řekneme, že jazyk A je *polynomiálně převoditelný* na jazyk B, pokud existuje funkce $f : \Sigma^* \to \Sigma^*$, která je vyčíslitelná v polynomiálním čase[1] a pro kterou platí, že

$$(\forall x \in \Sigma^*)[x \in A \Leftrightarrow f(x) \in B].$$

Tento fakt označíme pomocí $A \leq_m^p B$, ◂

V literatuře se kromě značení $A \leq_m^p B$ též objevuje značení $A \propto B$, námi zvolené značení však lépe naznačuje, že polynomiální převoditelnost je m-převoditelnost s přidaným požadavkem vyčíslitelnosti převádějící funkce v polynomiálním čase. Pokud je $A \leq_m^p B$, znamená to, že rozhodnutí, zda $x \in A$ lze provést tak, že nejprve určíme $y = f(x)$ a poté položíme dotaz zda $y \in B$. Pokud je tedy jazyk B rozhodnutelný v polynomiálním čase, pak totéž platí i o A. Na druhou stranu pokud neznáme, žádný polynomiální algoritmus rozhodující A, pak neznáme ani polynomiální algoritmus rozhodující B. Pro tyto úvahy je samozřejmě potřebné, že hodnotu převádějící funkce $f(x)$ lze určit v polynomiálním čase. Polynomiální převoditelnost má řadu vlastností společných s m-převoditelností.

Lemma 11.1.2 (Vlastnosti polynomiální převoditelnosti)

1. *Relace $\leq_m^p$ je reflexivní a tranzitivní.*
2. *Nechť A a B jsou jazyky, pro něž platí $A \leq_m^p B$. Pokud je $B \in \mathrm{P}$, pak i $A \in \mathrm{P}$.*
3. *Nechť A a B jsou jazyky, pro něž platí $A \leq_m^p B$. Pokud je $B \in \mathrm{NP}$, pak i $A \in \mathrm{NP}$.*

Důkaz: 1. Reflexivita plyne z toho, že identita je funkce vyčíslitelná v polynomiálním čase. Tranzitivita plyne z toho, že složením dvou polynomů vznikne opět polynom, přesněji, je-li f funkce převádějící jazyk A na jazyk B, tedy $A \leq_m^p B$ s použitím f, a g je funkce převádějící jazyk B na jazyk C, tedy $B \leq_m^p C$ s použitím g,

[1] To znamená, že existuje Turingův stroj M, který pracuje v polynomiálním čase a počítá funkci f. Viz též definice 5.4.1 a 4.1.8.

pak $g \circ f$ převádí A na C, jsou-li f i g vyčíslitelné v polynomiálním čase, lze totéž říci i o $g \circ f$, tedy $A \leq_m^p C$ s použitím $g \circ f$.

2. Je-li $B \in \mathrm{P}$, pak existuje Turingův stroj M, který přijímá B v polynomiálním čase. Je-li f funkce, která převádí A na B, a je-li f vyčíslitelná v polynomiálním čase, pak TS M', který pro vstup x spočítá $f(x)$ a poté pustí M k rozhodnutí, zda $f(x) \in B$, přijímá A v polynomiálním čase.

3. Platí z téhož důvodu jako předchozí bod, protože tytéž argumenty lze použít i pro nedeterministický TS. □

Poznámka 11.1.3 *V poznámce 7.2.3 jsme navíc k m-převoditelnosti neformálně zavedli turingovskou převoditelnost. I tu můžeme omezit polynomiálním časem a takovému převodu se říká Cookův či cookovský, zatímco relaci $\leq_m^p$ se říká Karpův či karpovský převod. Přesněji tedy řekneme, že problém A je* cookovsky převoditelný *na problém B, což označíme pomocí $A \leq_T^p B$, pokud existuje algoritmus, který rozhoduje problém A, přičemž může pokládat dotazy černé skříňce, či orákulu, která umí řešit problém B, a tento algoritmus pracuje v polynomiálním čase, počítáme-li dobu zodpovězení dotazů černou skříňkou (orákulem) jako konstantní. Toto tedy odpovídá následující intuici: „Pokud existuje polynomiální algoritmus rozhodující problém B, pak existuje i polynomiální algoritmus rozhodující problém A.“ Podobně jako v případě m-převoditelnosti a turingovské převoditelnosti i zde platí, že je-li $A \leq_m^p B$, platí také $A \leq_T^p B$, ale opačná implikace platit nemusí. Triviálním příkladem je opět jazyk* DIAG *neboť zřejmě platí, že* DIAG $\leq_T^p \overline{\text{DIAG}}$, *ale jistě neplatí* DIAG $\leq_m^p \overline{\text{DIAG}}$, *protože to by muselo platit i* DIAG $\leq_m \overline{\text{DIAG}}$, *víme přitom, že to není možné, protože* DIAG *není částečně rozhodnutelný jazyk, zatímco* $\overline{\text{DIAG}}$ *ano. Až se dostaneme ke třídě* co-NP, *všimneme si, že pokud se* co-NP *nerovná* NP, *pak mají tuto vlastnost všechny* NP*-úplné úlohy.*

Stejně jako v případě m-převoditelnosti, i zde budou předmětem našeho zájmu ty nejtěžší problémy, přičemž nás nejvíce zajímají problémy z třídy NP.

Definice 11.1.4 Jazyk A je NP-*těžký*, pokud pro každý jazyk $B \in \mathrm{NP}$ platí, že $B \leq_m^p A$. O jazyku A řekneme, že je NP-*úplný*, pokud je NP-těžký a navíc patří do třídy NP. ◂

Pojem úplnosti můžeme definovat pro libovolnou třídu, nejen pro třídu NP, v literatuře se mnoho prostoru věnuje například PSPACE-úplným problémům. Jen v případě třídy P bychom zjistili, že vzhledem k polynomiální převoditelnosti by byly P-úplné všechny netriviální problémy v P. Za triviální považujeme problém bez pozitivních instancí, který odpovídá prázdnému jazyku, a problém bez negativních instancí, který odpovídá jazyku všech řetězců. Všechny netriviální problémy v P jsou na sebe totiž vzájemně polynomiálně převoditelné. Kdybychom tedy chtěli studovat strukturu třídy P a zabývat se P-úplností, nezbylo by nám, než dále omezit polynomiální převod. Obvykle se P-úplnost definuje za pomoci převodu s logaritmickým meziprostorem.

Pokud by se ukázalo, že nějaký NP-úplný problém patří do P, platilo by P = NP. Protože se domníváme, že tato rovnost neplatí, předpokládáme také, že je-li problém NP-úplný, pak nejspíš nepatří do P a zjistíme-li tedy o nějakém problému, že je NP-úplný, jsou naše šance na nalezení polynomiálního algoritmu řešící tento problém velmi

malé, přinejmenším se to dosud nikomu nepodařilo. Na druhou stranu se dá ukázat, že je-li P $\subsetneq$ NP, pak existují i jazyky, které sice nejsou NP-úplné, ale patří do NP $\setminus$ P, a skutečně existují i praktické problémy, u kterých byly zatím veškeré snahy o nalezení polynomiálního algoritmu marné, ale na druhou stranu nejsme zatím schopni dokázat či vyvrátit jejich NP-úplnost[2].

Chceme-li o nějakém problému A ukázat, že je NP-úplný, můžeme samozřejmě postupovat podle definice a popsat, jak libovolný problém z NP polynomiálně převést na A. Takový důkaz je sice poněkud obtížný, díky tranzitivitě nám však stačí jej provést jen pro jeden problém, neboť máme-li už nějaký NP-úplný problém, můžeme dále využít tranzitivity polynomiální převoditelnosti a následujícího lemmatu.

Lemma 11.1.5 *Nechť A a B jsou jazyky patřící do třídy* NP. *Platí-li $A \leq_m^p B$ a je-li A* NP*-úplný, pak i B je* NP*-úplný.*

Důkaz: Tvrzení plyne přímo z tranzitivity relace $\leq_m^p$ zaručené lemmatem 11.1.2 a definice NP-úplného problému. □

Jak jsme již zmínili, abychom mohli tohoto postupu využít, musíme nejprve dokázat těžkost nějakého problému přímo podle definice. V případě 1-úplnosti či m-úplnosti byl nejpřirozenějším úplným problémem univerzální jazyk L_u a problém zastavení L_{HALT}, v případě třídy NP a polynomiální převoditelnosti můžeme podobně uvážit analogii univerzálního jazyka s omezeným časem.

Problém 11.1.6: Existence certifikátu (CERT)

Instance: Řetězec w kódující DTS M, řetězec x a řetězec 1^t pro nějaké $t \in \mathbb{N}$.
Otázka: Existuje řetězec y s $|y| \leq t$, pro který platí, že $M(x, y)$ přijme v t krocích?

Všimněme si, že t je v instanci problému *CERT* zakódováno unárně, to proto, že kdybychom použili binární kódování, pak čas, který dovolíme stroji M by byl exponenciální ve velikosti vstupu t, protože při binárním kódování by byl kód t dlouhý jen $\log_2 t$. My však chceme omezovat počet kroků stroje M pomocí t, a proto jeho hodnotu předáváme zakódovanou unárně.

Věta 11.1.7 *Problém CERT je* NP*-úplný.*

Důkaz: Nejprve ukážeme, že problém *CERT* patří do třídy NP. Nechť U označuje univerzální Turingův stroj, který na vstup dostane w, x, y, 1^t a simuluje nad x a y práci stroje M zakódovaného ve w. Pokud M přijme v t krocích, U přijme, v opačném případě U odmítne. Z toho, jak jsme si popisovali univerzální Turingův stroj, lze ukázat, že tento stroj nestráví při práci nad vstupem o mnoho víc času, než M, měl by mu stačit čas $O(t^2 + |w|t)$, každopádně U pracuje v polynomiálním čase vzhledem k velikosti vstupu.

[2] Jde např. o rozhodnutí, zda je daná formule φ v konjunktivně normální formě ekvivalentní dané formuli ψ v disjunktivně normální formě.

Nyní $(w, x, 1^t) \in CERT$, právě když existuje y, pro něž platí, že $|y| \leq t$ a $U(w, x, y, 1^t)$ přijme. Podle definice je tedy problém $CERT \in$ NP.

Zbývá ukázat, že $CERT$ je NP-úplný. Nechť A je libovolný jazyk z třídy NP. To znamená, že existuje jazyk $B \in$ P a polynom $p(n)$, pro které platí, že $x \in A$ právě když existuje y, $|y| \leq p(|x|)$ takové, že $(x, y) \in B$. Předpokládejme, že jazyk B je přijímán TS M, jehož čas je též omezen polynomem $p(n)$, jako p prostě zvolíme takový polynom, který omezuje jak délku y, tak délku výpočtu M. Nechť w je řetězec kódující TS M. Definujme funkci $f(x) = (w, x, 1^{p(|x|)})$. Tuto funkci jistě zvládneme spočítat v polynomiálním čase, délka kódu w je konstantní a nezávislá na délce řetězce x, a známe-li hodnotu polynomu $p(|x|)$, můžeme vygenerovat řetězec $1^{p(|x|)}$ v čase $p(|x|)$. Zbývá ukázat, že $x \in A$, právě když $f(x) \in CERT$.

Předpokládejme nejprve, že $x \in A$, podle definice to znamená, že existuje y, jehož délka je omezena pomocí $p(|x|)$, pro které platí, že $(x, y) \in B$, tedy že $M(x, y)$ přijme. $M(x, y)$ se zastaví v čase $t = p(|x|)$, protože polynom p omezuje i délku výpočtu M nad x a y. Podle definice problému $CERT$ tedy $(w, x, 1^{p(|x|)}) \in CERT$.

Nyní předpokládejme, že $f(x) = (w, x, 1^{p(|x|)}) \in CERT$, podle definice problému $CERT$ to znamená, že existuje y, $|y| \leq p(|x|)$, pro něž $M(x, y)$ přijme v $p(|x|)$ krocích, tedy $(x, y) \in B$, potažmo $x \in A$. □

Problém $CERT$ ve skutečnosti není nic jiného, než přeformulování definice problému ze třídy NP, podobně jako v sobě univerzální jazyk kódoval všechny částečně rozhodnutelné jazyky, $CERT$ v sobě kóduje všechny jazyky z třídy NP, pokud se omezíme na hodnoty t, které jsou polynomiální v délce $|x|$. Díky větě 11.1.7 tedy víme, že existuje nějaký NP-úplný problém, což je sice zajímavé, ale problém $CERT$ je velmi umělý a navíc ukazovat NP-těžkost nějakého problému převodem z $CERT$ je asi stejně obtížné, jako bychom to ukazovali přímo z definice. Zaměříme se proto na nalezení nějakého praktického problému, o kterém bychom mohli ukázat, že je NP-úplný.

11.2. Cookova-Levinova věta

Za nejznámější NP-úplným problém lze považovat Splnitelnost, tedy otázka, zda daná formule v KNF je splnitelná.

Problém 11.2.1: Splnitelnost formule v KNF (SAT)

Instance:	Formule φ na n proměnných v konjunktivně normální formě.
Otázka:	Existuje ohodnocení proměnných $t \in \{0, 1\}^n$, pro které platí $\varphi(t) = 1$? Jinými slovy, existuje ohodnocení, pro které je φ splněná?

Větě, která ukazuje NP-úplnost problému Splnitelnosti přímo z definice třídy NP, se říká Cookova-Levinova podle dvou vědců, kteří nezávisle na sobě položili základ

teorie NP-úplnosti, Stephen A. Cook v [1] a Leonid A. Levin v [6]. Cook však ve skutečnosti používal Cookovu převoditelnost (viz poznámka 11.1.3), pojem polynomiální převoditelnosti, který používáme my, pochází od Richarda M. Karpa z [5]. My ukážeme NP-úplnost problému Splnitelnosti ve dvou krocích. Nejprve ukážeme NP-úplnost problému Kachlíkování, poté převodem z Kachlíkování na Splnitelnost teprve ukážeme Cookovu-Levinovu větu.

Problém 11.2.2: Kachlíkování (KACHL, *anglicky* Tiling)

Instance: Množina barev B, přirozené číslo s, čtvercová síť S velikosti $s \times s$, hrany jejíchž krajních políček jsou obarveny barvami z B. Dále je součástí instance množina $K \subseteq B \times B \times B \times B$ s typy kachlíků, které odpovídají čtverci, jehož hrany jsou obarveny barvami z B. Tyto kachlíky mají přesně definovaný horní, dolní, levý i pravý okraj a není možné je otáčet.

Otázka: Existuje přípustné vykachlíkování čtvercové sítě S kachlíky, jejichž typy jsou v množině K? Přípustné vykachlíkování je takové přiřazení typů kachlíků jednotlivým polím čtvercové sítě S, v němž kachlíky, které spolu sousedí mají touž barvu na vzájemně dotýkajících se hranách a kachlíky, které se dotýkají strany S, mají shodnou barvu s okrajem. Jednotlivé typy kachlíků lze použít víckrát.

Ačkoli obvykle se pod Cookovou-Levinovou větou rozumí důkaz NP-úplnosti splnitelnosti, zůstaneme u tohoto názvu i v případě Kachlíkování. Hned následující problém, jehož NP-úplnost si ukážeme, bude již splnitelnost logické formule v konjunktivně normální formě, a můžeme to tedy brát i tak, že teprve potom budeme mít ukázanou původní Cookovu-Levinovu větu.

Věta 11.2.3 *KACHL je* NP*-úplný problém.*

Důkaz: Všimněme si nejprve, že $KACHL \in$ NP. To plyne z toho, že dostaneme-li vykachlíkování sítě S, tedy přiřazení typů kachlíků jednotlivým políčkům, dokážeme ověřit v polynomiálním čase, jde-li o přípustné vykachlíkování. Ve skutečnosti úloha nalezení správného vykachlíkování je polynomiálně ověřitelná a patří tedy do NPF, proto její rozhodovací verze patří do NP.

Nechť $A \subseteq \{0,1\}^*$ je libovolný problém, který patří do NP, ukážeme, že $A \leq_m^p KACHL$. Podle definice NP-úplného problému to bude znamenat, že KACHL je NP-těžký, a protože patří do NP, tak i NP-úplný problém. To, že $A \in$ NP, podle věty 9.5.4 znamená, že existuje NTS M, který přijímá A (tj. $A = L(M)$), a počet kroků každého přijímajícího výpočtu je omezen polynomem $p(n)$, bez újmy na obecnosti můžeme předpokládat, že $p(n) \geq n$, v opačném případě by M nepřečetl ani celý svůj vstup a pokud by platilo, že $p(n) \leq n$, stačí vzít $\max\{p(n), n\}$ místo $p(n)$. Připomeňme si, že podle definice $x \in A$, právě když existuje přijímající výpočet NTS M nad vstupem x, který má délku nejvýš

$p(|x|)$. Nechť $M = (Q, \Sigma, \delta, q_0, F)$, kde Q obsahuje stavy q_0 a q_1 a $\{0, 1, \lambda\} \subseteq \Sigma$. Abychom si zjednodušili situaci, budeme předpokládat, že M splňuje následující předpoklady:

1. $F = \{q_1\}$, tj. M má jediný přijímající stav q_1 různý od q_0.
2. Pro každé $a \in \Sigma$ je $\delta(q_1, a) = \varnothing$, tj. z přijímajícího stavu neexistuje definovaný přechod.
3. Počáteční konfigurace vypadá tak, že hlava stojí na nejlevějším symbolu vstupního slova x, které je zapsáno počínaje od levého okraje vymezeného prostoru délky $p(|x|)$. Zbytek pásky je prázdný.
4. Během výpočtu se hlava M nepohne nalevo od místa, kde byla v počáteční konfiguraci, tj. mimo vymezený prostor.
5. Přijímající konfigurace je daná jednoznačně a vypadá tak, že páska je prázdná a hlava stojí na nejlevější pozici vymezeného prostoru. To odpovídá tomu, že než se M rozhodne přijmout, smaže nejprve obsah pásky a přesune hlavu k levému okraji vymezeného prostoru.

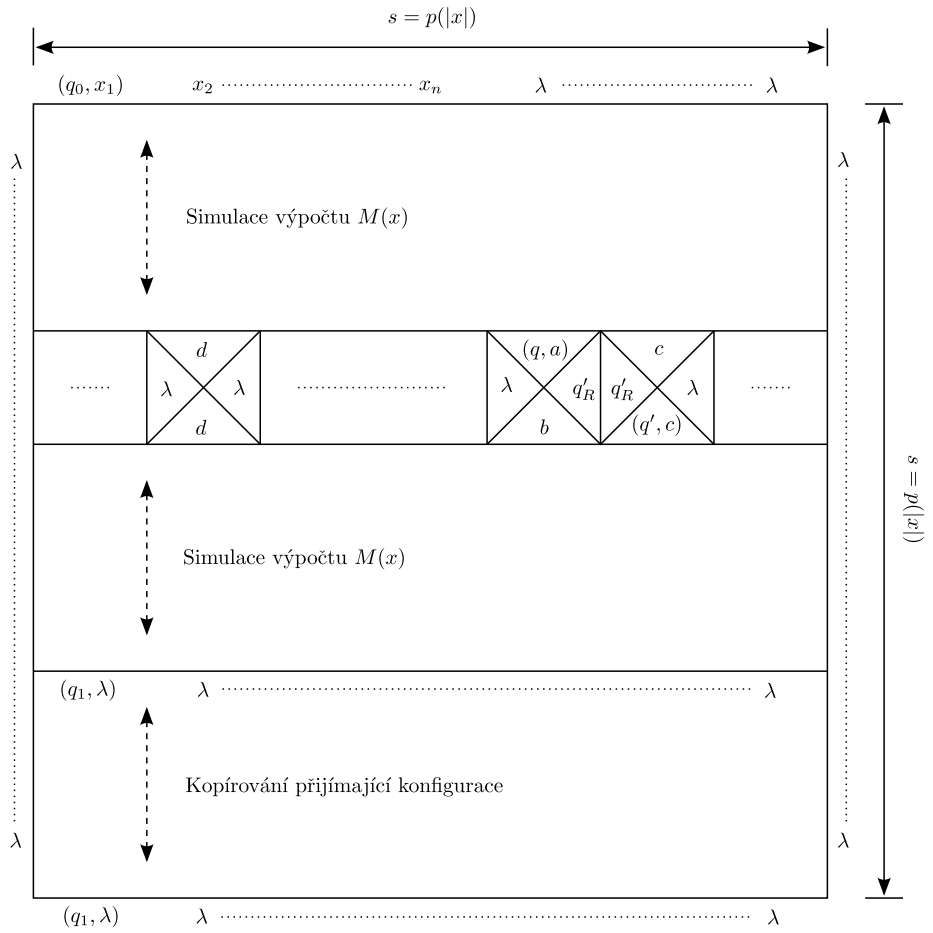
Není těžké ukázat, že ke každému NTS M_1 lze zkonstruovat NTS M_2, který přijímá týž jazyk jako M_1, „dělá totéž“ a splňuje uvedené podmínky. Většinu zmíněných předpokladů jsme již dříve použili, například při konstrukci univerzálního TS. Bez újmy na obecnosti tedy můžeme předpokládat, že M splňuje uvedené podmínky.

Nechť x je instance problému A, popíšeme, jak z M, polynomu p a instance x vytvořit instanci Kachlíkování, pro kterou bude platit, že v ní existuje přípustné vykachlíkování, právě když existuje přijímající výpočet $M(x)$, tj. $M(x)$ přijme.

Idea důkazu je taková, že hrany barev mezi dvěma řádky kachlíků budou kódovat konfigurace výpočtu NTS M nad vstupem x. Vhodným výběrem kachlíků zabezpečíme, že v přípustném vykachlíkování bude řada kachlíků simulovat změnu konfigurace na následující pomocí přechodové funkce. Horní a dolní okraje sítě S obarvíme tak, aby barvy určovaly počáteční a přijímající konfiguraci, obě jsou dané jednoznačně, přijímající zcela jednoznačně, počáteční je sice závislá na x, ale pro dané x je již jednoznačná. Konstrukce je ilustrovaná na obrázku 11.1. Barvy kachlíků tedy budou odpovídat symbolům, které potřebujeme pro zakódování konfigurace, ale budeme potřebovat i pomocné barvy pro přenos informace o stavu o kachlík vlevo nebo vpravo. Položíme tedy

$$B = \Sigma \cup Q \times \Sigma \cup \{q_L, q_R \mid q \in Q\}$$

Význam těchto barev se ozřejmí při konstrukci jednotlivých typů kachlíků. Zatímco vstupní abecedou stroje M je jen $\{0, 1, \lambda\}$, neboť x musí být binární řetězec, pracovní abecedu stroje M nijak neomezujeme, a proto zde používáme obecnou abecedu Σ, přičemž předpokládáme, že $\lambda \in \Sigma$. V dalším textu budeme označovat políčko sítě S na i-tém řádku a v j-tém sloupci pomocí $S[i, j]$, kde $i, j \in \{1, \ldots, s\}$. Jak jsme zmínili, cílem je, aby řada barev mezi dvěma řadami kachlíků odpovídala konfiguraci. Barva (q, a) v této konfiguraci bude kódovat políčko na pásce, nad kterým se vyskytuje hlava, přičemž



Obrázek 11.1.: Přehled převodu obecného problému $A \in$ NP na problém KACHL. Horní hrana čtvercové sítě je obarvena počáteční konfigurací, spodní hrana přijímající konfigurací, boky jsou obarveny symbolem λ. Pro ukázku je zde řádek kachlíků s provedením instrukce $(q', b, R) \in \delta(q, a)$, dva kachlíky slouží k provedení instrukce, na ostatních místech je jen kopírovací kachlík pro okopírování barev na další řádek. Po ukončení výpočtu je dokopírována přijímající konfigurace až ke spodní hraně čtvercové sítě.

q označuje stav, ve kterém se M nachází, ostatní políčka konfigurace budou obarvena barvou odpovídající symbolu na daném místě.

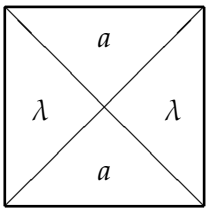
Velikost jedné strany čtvercové sítě položíme $s = p(|x|)$, všimněme si, že $p(|x|)$ omezuje nejen počet konfigurací v přijímajícím výpočtu, ale i počet buněk, které stihne M při výpočtu nad x popsat, tedy i délku slova na pásce v konfiguraci.

Popišme nyní, jakými barvami budou obarveny okraje čtvercové sítě S. Boční strany budou obarveny barvou λ, horní strana bude obarvena počáteční konfigurací a dolní strana konfigurací přijímající. Přesněji, nechť $x = x_1 \dots x_n$, kde $x_i \in \{0,1\}$ pro $i = 1, \dots, n$. Pak horní hrana nejlevější horní buňky $S[1,1]$ bude obarvena barvou (q_0, x_1), přičemž je-li vstup x prázdný, tj. $n = |x| = 0$, pak je obarvena barvou (q_0, λ). Horní strany políček $S[1,2], \dots, S[1,n]$ jsou obarveny popořadě symboly $x_2, \dots, x_n$. Horní hrany zbylých políček prvního řádku $S[1,n+1], \dots, S[1,s]$ budou obarveny barvou λ, odpovídající prázdnému políčku. Levé hrany políček v prvním sloupci budou obarveny barvou λ, stejně jako pravé hrany políček v s-tém sloupci. Spodní hrana sítě S bude obarvena jednoznačnou přijímající konfigurací, tedy spodní hrana levého dolního políčka $S[s,1]$ bude mít barvu (q_1, λ), další políčka dolního řádku $S[s,2], \dots, S[s,s]$ budou mít spodní hrany obarveny barvou λ.

Do typů kachlíků zakódujeme přechodovou funkci Turingova stroje M, čímž dosáhneme toho, že správné vykachlíkování řádků 2 až s bude odpovídat výpočtu M nad x. Navíc přidáme možnost kopírování přijímající konfigurace tak, abychom ošetřili i případ, kdy výpočet M nad x skončí po méně než $p(|x|)$ krocích.

Pro každý symbol $a \in \Sigma$ nejprve přidáme typ kachlíku:

(I)



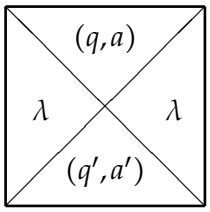
Tento kachlík bude odpovídat těm místům konfigurace, která přechodová funkce nemění, protože se nad nimi nevyskytuje hlava v předchozí ani v následující konfiguraci.

Pro každý stav $q \in Q$ a každý znak $a \in \Sigma$ nyní popíšeme kachlíky, které je nutno přidat, abychom zabezpečili provedení možných přechodů daných funkcí $\delta(q,a)$. Připomeňme, že místo na pásce, kde je hlava, označujeme dvojicí (q,a), kde q je stav, v němž se M nachází a a je čtený symbol. Volbou kachlíků musíme zabezpečit jednak správné provedení instrukce, jednak to, aby ve správném vykachlíkování byla v každé konfiguraci právě jedna dvojice (q,a), musíme si tedy dát pozor, aby se kachlíky odpovídající jednotlivým instrukcím nepomíchaly mezi sebou.

Pokud $\delta(q,a) = \varnothing$, nepřidáme přirozeně nic.

Pro každý stav $q' \in Q$ a znak $a' \in \Sigma$, pro které platí, že $(q', a', N) \in \delta(q,a)$, tj. z (q,a) lze přejít do stavu q' s přepsáním symbolu a na a' a s tím, že hlava zůstane na místě, přidáme kachlík:

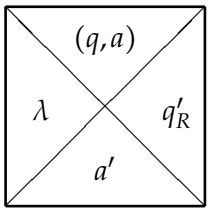
(II)

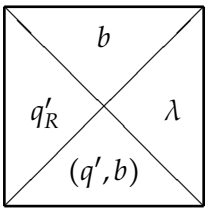


To odpovídá tomu, že pokud nevykonáváme žádný pohyb, pouze změníme stav a přepíšeme symbol.

Pro každý stav $q' \in Q$ a znak $a' \in \Sigma$, pro které platí, že $(q', a', R) \in \delta(q, a)$, tj. z (q, a) lze přejít do stavu q' s přepsáním symbolu a na a' a s tím, že se hlava pohne vpravo, přidáme pro každé $b \in \Sigma$ kachlíky:

(III) (IV)

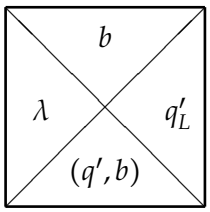


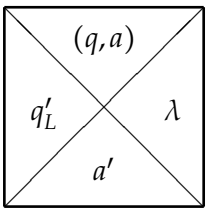


To odpovídá tomu, že při pohybu doprava přesuneme hlavu o jednu buňku vpravo, stav si zapamatujeme v boční hraně. Pro přenesení stavu přitom využijeme barvu q'_R s indexem R, zapamatujeme si tedy i směr pohybu, a to proto, aby se nám nepomíchaly dvojice kachlíků pro pohyb doleva a pohyb doprava.

Podobně pro každý stav $q' \in Q$ a znak $a' \in \Sigma$, pro které platí, že $(q', a', L) \in \delta(q, a)$, tj. z (q, a) lze přejít do stavu q' s přepsáním symbolu a na a' a s tím, že se hlava pohne vlevo, přidáme pro každé $b \in \Sigma$ kachlíky:

(V) (VI)

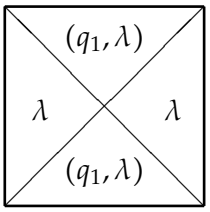




Tím, že nyní používáme verzi q'_L stavu q', nepomíchají se nám přechody doprava a doleva mezi sebou.

Výpočet stroje M nemusí skončit přesně po $s = p(n)$ krocích a mohlo by se tedy stát, že bychom s dosud uvedenými typy kachlíků nemohli dokachlíkovat zbylé řady čtvercové sítě S po ukončení výpočtu. Za tím účelem přidáme kachlík umožňující spolu s (I) pro $a = \lambda$ okopírování přijímající konfigurace. V této konfiguraci je páska prázdná a i čteným symbolem je prázdný znak λ, stačí nám tedy přidat následující kopírovací kachlík:

(VII)



Protože z přijímajícího stavu q_1 nevede v M žádný přechod, v okamžiku, kdy začneme tento stav kopírovat tímto typem kachlíku, musíme s tím vytrvat až do posledního řádku.

Funkce f, která bude převádět instanci problému A na instanci problému KACHL provede právě popsanou konstrukci, tj. z popisu M a instance x vytvoří instanci (B, K, s, S), kde množina K obsahuje popsané typy kachlíků a pod S míníme obarvení okrajů sítě. Tuto konstrukci je zřejmě možné provést v polynomiálním čase. Všimněme si, že velikost reprezentace M je nezávislá na velikosti instance A a jde tedy o konstantu, jediné co je v konstrukci závislé na velikosti vstupu je tedy rozměr sítě $s = p(|x|)$ a obarvení horního okraje S počáteční konfigurací.

Zbývá ukázat, že $x \in A$, právě když má takto zkonstruovaná instance prefprob:kachl-KACHL přípustné vykachlíkování.

Předpokládejme nejprve, že $x \in A$, to znamená podle definice, že existuje přijímající výpočet $M(x)$ daný posloupností konfigurací $K_0^x = K_0, \ldots, K_t = K_F$, kde K_0^x je počáteční konfigurace M při výpočtu nad x a K_F je jednoznačně daná přijímající konfigurace. Podle předpokladu platí, že $t \leq p(|x|)$ a délka slova na pásce v každé konfiguraci je rovněž nejvýš $p(|x|)$. Popíšeme, jak s pomocí konfigurací $K_0, \ldots, K_t$ obarvit síť S. Intuitivně je jasné, jak toto vykachlíkování bude vypadat, máme-li vykachlíkovaných i řad, kde $i \in \{0, \ldots, t-1\}$ s tím, že barvy na spodní hraně řady i odpovídá konfiguraci K_i (je-li $i = 0$, pak jde o barvy horního okraje S, víme, že ty odpovídají $K_0^x = K_0$, základní krok je tedy splněn), vykachlíkujeme $(i+1)$-ní řadu s odsimulováním příslušné instrukce, která vedla k přechodu z K_i do K_{i+1}. Takto se dostaneme k $K_t = K_F$ a poté dokopírujeme K_F až na poslední řádek čtvercové sítě S. Inspekcí jednotlivých kachlíků bychom ověřili, že instrukci lze vždy odsimulovat, většina kachlíků by byla typu (I), tedy kopírovacích na místech beze změny, v místě provedení instrukce bychom použili odpovídající kachlík (II), dvojici (III) a (IV), nebo dvojici (V) a (VI). Horní barva kachlíku (III) nebo (V) je daná tím, co se v K_i na příslušném místě nachází za znak. Kopírování K_F do konce (je-li třeba), pak zabezpečí kachlík (VII) ve spolupráci s (I) pro $a = \lambda$. Detailní rozbor případů ponecháme na čtenáři.

Nyní předpokládejme, že existuje přípustné vykachlíkování čtvercové sítě S. Potřebujeme ukázat, že řádky barev mezi jednotlivými řádky kachlíků určují posloupnost konfigurací v přijímajícím výpočtu M nad x. Postupujme opět indukcí podle $i = 0, \ldots, s$, nechť $b_{i,1}, \ldots, b_{i,s} \in B$ je posloupnost barev mezi i-tým a $(i+1)$ ním řádkem vykachlíkování čtvercové sítě S, přičemž je-li $i = 0$, označují barvy $b_{i,1}, \ldots, b_{i,s}$ horní barvy S a je-li $i = s$, označují tyto barvy spodní barvy S. Musíme ukázat tyto dvě vlastnosti pro každé $i = 0, \ldots, s$:

1. V posloupnosti $b_{i,1}, \ldots, b_{i,s}$ je právě jedna barva typu $(q,a) \in Q \times \Sigma$, ostatní jsou typu $a \in \Sigma$, tj. posloupnost $b_{i,1}, \ldots, b_{i,s}$ určuje konfiguraci K_i.

2. Pro takto určené konfigurace platí, že K_i lze z K_{i-1} vytvořit přechodem pomocí přechodové funkce δ pro $i > 0$ a $K_0 = K_0^x$.

Obě vlastnosti jsou jistě splněné pro $i = 0$. Předpokládejme nyní, že jsou splněny pro $i \in \{0, \ldots, s\}$ a ukažme, že platí pro $i + 1$. Ve skutečnosti obě vlastnosti opět jednoduše plynou z toho, jaké kachlíky máme k dispozici. Podle indukčního předpokladu se na řádku barev $b_{i,1}, \ldots, b_{i,s}$ vyskytuje právě jedna pozice, řekněme k, pro kterou platí, že $b_{i,k} = (q,a)$ pro nějaký stav $q \in Q$ a znak $a \in \Sigma$. To znamená, že v $(i+1)$-ním řádku kachlíků je právě jeden kachlík s horní barvou (q,a), a to na pozici $S[i+1,k]$, ostatní mají horní barvu typu $c \in \Sigma$. Rozborem případů ověříme, že obě požadované vlastnosti jsou splněné i pro řádek barev $b_{i+1,1}, \ldots, b_{i+1,s}$. Předně si všimněme, že kachlíky nám neumožňují vytvořit barvu (q,a) z ničeho, ale musí jít o barvu tohoto typu převedenou z předchozího řádku, kachlíky se spodní barvou (q',b), které nahoře nemají barvu (q,a) (tj. kachlíky typu (IV) nebo (V)) totiž vyžadují aby nalevo nebo napravo od nich byl odpovídající kachlík s barvou (q,a) nahoře a barvou a' dole (tj. kachlíky typu (III) a (VI)). To znamená, že pokud se má někde barva (q,a) objevit, musela vedle zmizet, díky tomu zůstane zachovaná první vlastnost. Tento přenos však vždy odpovídá provedení instrukce díky tomu, jak jsme kachlíky definovali, i druhá vlastnost zůstane tedy zachovaná. Tím, že spodní hrana S je obarvena přijímající konfigurací, je vynuceno, že poslední řada barev kachlíků bude odpovídat přijímající konfiguraci, ta se může kopírovat nahoru, ale v každém případě musí platit, že poslední konfigurace v posloupnosti konfigurací odpovídajícím barvám mezi řadami kachlíků musí být přijímající. Formální rozbor případů ponecháme na čtenáři.

Dostáváme tedy, že posloupnost konfigurací daných barvami na hranách mezi řádky kachlíků v čtvercové síti S odpovídají přijímajícímu výpočtu M nad vstupem x a jde dokonce o vzájemnou korespondenci, a tedy máme-li přípustné vykachlíkování, znamená to, že $M(x)$ přijme. Z toho plyne, že $x \in A$.

Tímto jsme tedy dokončili převod libovolného problému $A \in$ NP na Kachlíkování a protože Kachlíkování patří do NP, znamená to, že Kachlíkování je NP-úplný problém.□

Nyní můžeme přistoupit k důkazu toho, že problém splnitelnosti je NP-úplný.

Věta 11.2.4 (Cookova-Levinova) *Problém SAT je* NP*-úplný.*

Důkaz: To, že SAT patří do NP, plyne z toho, že pokud dostaneme vektor $t \in \{0,1\}^n$, můžeme spočítat hodnotu $\varphi(t)$ v polynomiálním čase. Certifikátem kladné odpovědi je tedy splňující ohodnocení. Toto ohodnocení je ve skutečnosti řešením úlohy splnitelnosti, v níž chceme splňující ohodnocení nalézt a ne jen zjistit, zda existuje, tato úloha tedy patří do NPF.

NP-těžkost splnitelnosti ukážeme převodem z problému Kachlíkování. Uvažme instanci Kachlíkování danou množinou barev B, přirozeným číslem s, čtvercovou sítí S velikosti $s \times s$, jejíž okraje jsou obarveny barvami z B, a množinou typů kachlíků $K =$

$\{T_1, \ldots, T_{|K|}\}$. Popíšeme, jak zkonstruovat formuli φ v KNF, která bude splnitelná právě tehdy, když tato instance má přípustné vykachlíkování.

Formule φ bude využívat $s^2 \cdot |K|$ proměnných $x_{i,j,k}$ pro $i = 1, \ldots, s$, $j = 1, \ldots, s$ a $k = 1, \ldots, |K|$. Konstrukcí formule φ zajistíme, že ve splňujícím ohodnocení bude hodnota $x_{i,j,k} = 1$ odpovídat tomu, že na políčko $S[i, j]$ umístíme kachlík T_k. V konstrukci použijeme následující dvě množiny určující nekompatibilní dvojice kachlíků:

$$V = \{(p, q) \mid \text{spodní barva } T_p \text{ se neshoduje s horní barvou } T_q\}$$
$$H = \{(p, q) \mid \text{pravá barva } T_p \text{ se neshoduje s levou barvou } T_q\}$$

tj. množina V obsahuje dvojice typů, jež nemohou být umístěny nad sebe, tedy vertikálně. Množina H obsahuje dvojice typů, jež nemohou být umístěny vedle sebe, tedy horizontálně. Podobné množiny si definujeme i pro barvy na okrajích, pro $i = 1, \ldots, s$ definujeme množiny U_i, B_i, L_i a R_i.

$$U_j = \{k \mid \text{horní barva } T_k \text{ se shoduje s barvou horního okraje } S \text{ v } j\text{-tém sloupci}\}$$
$$B_j = \{k \mid \text{spodní barva } T_k \text{ se shoduje s barvou spodního okraje } S \text{ v } j\text{-tém sloupci}\}$$
$$L_j = \{k \mid \text{levá barva } T_k \text{ se shoduje s barvou levého okraje } S \text{ na } j\text{-tém řádku}\}$$
$$R_j = \{k \mid \text{pravá barva } T_k \text{ se shoduje s barvou pravého okraje } S \text{ na } j\text{-tém řádku}\}$$

Množina U_i tedy obsahuje typy kachlíků, jež mohou být umístěny na políčko $S[1, i]$, množina B_i obsahuje typy kachlíků, jež mohou být umístěny na políčko $S[s, i]$, množina L_i obsahuje typy kachlíků, jež mohou být umístěny na políčko $S[i, 1]$, a konečně R_i obsahuje typy kachlíků, jež mohou být umístěny na políčko $S[i, s]$.

Konstrukci formule φ popíšeme po částech.

(I) Pro $i, j = 1, \ldots, s$ definujeme klauzuli

$$C_{i,j} = \bigvee_{k=1}^{|K|} x_{i,j,k}.$$

Klauzule $C_{i,j}$ je splněna, pokud je na políčku $S[i, j]$ přiřazen alespoň jeden typ kachlíku, čili když existuje alespoň jedno $k \in \{1, \ldots, |K|\}$, pro něž je $x_{i,j,k} = 1$.

(II) Pro $i, j = 1, \ldots, s$ definujeme

$$\alpha_{i,j} = \bigwedge_{k=1}^{|K|} \bigwedge_{l=k+1}^{|K|} (\neg x_{i,j,k} \vee \neg x_{i,j,l}).$$

Formule $\alpha_{i,j}$ je splněna, je-li políčku $S[i, j]$ přiřazen nejvýš jeden typ kachlíku, čili pokud nejvýš pro jedno $k \in \{1, \ldots, |K|\}$ platí $x_{i,j,k} = 1$. Pokud by totiž pro dva různé indexy $k, l \in \{1, \ldots, |K|\}$ platilo $x_{i,j,k} = x_{i,j,l} = 1$, nebyla by klauzule $(\neg x_{i,j,k} \vee \neg x_{i,j,l})$ splněná.

(III) Pro $i, j = 1, \ldots, s$ definujeme

$$\beta_{i,j} = C_{i,j} \wedge \alpha_{i,j}.$$

Formule $\beta_{i,j}$ je tedy splněna, pokud políčku $S[i, j]$ přiřadíme právě jeden typ kachlíku, čili pokud existuje právě jeden index $k \in \{1, \ldots, |K|\}$, pro který platí $x_{i,j,k} = 1$.

(IV) Pro $1 \leq i \leq s - 1$, $1 \leq j \leq s$ definujeme formuli

$$\gamma_{i,j} = \bigwedge_{(p,q) \in V} (\neg x_{i,j,p} \vee \neg x_{i+1,j,q}).$$

Formule $\gamma_{i,j}$ je splněna, pokud nad sebou umístěným políčkům $S[i, j]$ a $S[i + 1, j]$ nejsou přiřazeny nekompatibilní typy kachlíků.

(V) Pro $1 \leq i \leq s$, $1 \leq j \leq s - 1$ definujeme formuli

$$\delta_{i,j} = \bigwedge_{(p,q) \in H} (\neg x_{i,j,p} \vee \neg x_{i,j+1,q}).$$

Formule $\delta_{i,j}$ je splněna, pokud vedle sebe umístěným políčkům $S[i, j]$ a $S[i, j + 1]$ nejsou přiřazeny nekompatibilní typy kachlíků.

(VI) Definujeme formuli

$$\varepsilon_u = \bigwedge_{j=1}^{s} \left(\bigvee_{k \in U_j} x_{1,j,k} \right).$$

Formule ε_u je splněna, pokud je na každém políčku v prvním řádku kachlík s barvou shodnou s příslušným horním okrajem.

(VII) Definujeme formuli

$$\varepsilon_b = \bigwedge_{j=1}^{s} \left(\bigvee_{k \in B_j} x_{s,j,k} \right).$$

Formule ε_b je splněna, pokud je na každém políčku v nejspodnějším řádku kachlík s barvou shodnou s příslušným spodním okrajem.

(VIII) Definujeme formuli

$$\varepsilon_l = \bigwedge_{j=1}^{s} \left(\bigvee_{k \in L_j} x_{j,1,k} \right).$$

Formule ε_l je splněna, pokud je na každém políčku v nejlevějším sloupci kachlík s barvou shodnou s příslušným levým okrajem.

(IX) Definujme formuli

$$\varepsilon_r = \bigwedge_{j=1}^{s} \left(\bigvee_{k \in R_j} x_{j,s,k} \right).$$

Formule ε_r je splněna, pokud je na každém políčku v nejpravějším sloupci kachlík s barvou shodnou s příslušným pravým okrajem.

S pomocí výše definovaných formulí nyní definujeme formuli φ jako

$$\varphi = \bigwedge_{i=1}^{s} \bigwedge_{j=1}^{s} \beta_{i,j} \wedge \bigwedge_{i=1}^{s-1} \bigwedge_{j=1}^{s} \gamma_{i,j} \wedge \bigwedge_{i=1}^{s} \bigwedge_{j=1}^{s-1} \delta_{i,j} \wedge \varepsilon_u \wedge \varepsilon_b \wedge \varepsilon_l \wedge \varepsilon_r.$$

Z konstrukce okamžitě vyplývá, že φ je formule v KNF a že velikost φ je polynomiální v s a $|K|$, a v polynomiálním čase lze φ i zkonstruovat. Zbývá ukázat, že pro výchozí instanci problému Kachlíkování existuje přípustné vykachlíčkování, právě když φ je splnitelná formule. Ve skutečnosti ukážeme, že ze splňujícího ohodnocení formule φ lze vyčíst přípustné vykachlíkování čtvercové sítě S a naopak z přípustného vykachlíkování lze určit splňující ohodnocení formule φ.

Předpokládejme nejprve, že existuje přípustné vykachlíčkování S kachlíky z K. Označme pomocí $S[i,j]$ index typu kachlíku, který je na i-tém řádku a j-tém sloupci. Definujme ohodnocení, které pro každé $i, j = 1, \ldots, s$ a $k = 1, \ldots, |K|$ přiřadí proměnné $x_{i,j,k}$ hodnotu

$$x_{i,j,k} = \begin{cases} 1 & \text{pokud je na políčku } S[i,j] \text{ kachlík typu } T_k \\ 0 & \text{jinak} \end{cases}$$

Není těžké ověřit, že každá z podformulí $\beta_{i,j}$, $\gamma_{i,j}$, $\delta_{i,j}$, ε_u, ε_b, ε_l a ε_r je tímto ohodnocením splněna, a tedy že i celá formule φ je splněna.

Předpokládejme na druhou stranu, že existuje splňující ohodnocení formule φ a pro $i, j = 1, \ldots, s$ a $k = 1, \ldots, |K|$pomocí $v[i,j,k]$ označme hodnotu přiřazenou proměnné $x_{i,j,k}$ ve splňujícím ohodnocení. Díky tomu, že ohodnocení splňuje i formuli $\beta_{i,j}$, existuje pro každé $i, j = 1, \ldots, s$ právě jedna hodnota $k \in \{1, \ldots, |K|\}$, pro kterou je $v[i,j,k] = 1$. Na pozici $S[i,j]$ tedy dáme kachlík toho typu T_k, pro který je $v[i,j,k] = 1$. Lze snadno ověřit, že podmínky zakódované do formulí $\gamma_{i,j}$, $\delta_{i,j}$, ε_u, ε_b, ε_l a ε_r zaručují, že tímto způsobem obdržíme přípustné vykachlíkování S. □

11.3. Další NP-úplné problémy

Nyní, když máme k dispozici praktický NP-úplný problém, můžeme jej použít k důkazu NP-těžkosti řady dalších problémů. My si ukážeme několik základních problémů, jmenovitě půjde o splnitelnost logické formule v KNF (SAT), splnitelnost logické formule složené z klauzulí délky 3 (3-SAT), vrcholové pokrytí grafu (VP), hamiltonovskou kružnici v grafu (HK), trojrozměrné párování (3DM) a loupežníky (*LOUP*), důkazy těžkosti všech těchto problémů se objevilo již v Karpově článku [5]. Zmíníme i několik variant těchto problémů, důkazy jejich těžkosti si ukážeme v rámci cvičení. Účelem této části a ukázky těchto převodů je sžít se s konceptem převodu. Současně je naším cílem seznámit se s různorodými převody a problémy v naději, že znalost obojího může pomoci čtenáři k případnému důkazu těžkosti problému, jenž se může objevit v praxi. Pokud narazíme na problém, u nějž máme podezření, že by mohl být NP-těžký, měla by naše cesta směřovat k nějakému seznamu NP-úplných úloh, v němž bychom se měli snažit nalézt problém tomu našemu podobný. V takovém případě se nám může stát, že nalezneme přímo nějakou formu problému, který nás zajímá. Jako zdroj řady známých NP-úplných problémů může sloužit seznam v [3].

11.3.1. Splnitelnost formulí v 3-KNF (3-SAT)

V případě, že se nějaký problém, například splnitelnost, ukáže být NP-úplným, můžeme se ptát, jak bychom museli omezit zadání, abychom dostali jednodušší úlohu. Už splnitelnost formulí v KNF je vlastně jen zvláštním případem obecné splnitelnosti, v níž se zajímáme o splnitelnost obecné formule zadané v libovolném tvaru. Zůstaneme-li u KNF, můžeme tuto formu dále omezovat například tak, že budeme připouštět jen krátké klauzule. Řekneme, že formule φ je v *3KNF*, je-li φ v KNF a každá klauzule φ obsahuje právě tři literály.

Problém 11.3.1: Splnitelnost formule v 3KNF (3-SAT)

Instance:	Formule φ na n proměnných v 3KNF, tj. konjunktivně normální formě, v níž každá klauzule obsahuje právě tři literály.
Otázka:	Existuje ohodnocení proměnných $t \in \{0,1\}^n$, pro které platí $\varphi(t) = 1$? (tj. ohodnocení, pro které je φ splněná).

Často se v definici problému 3-SAT vyžaduje, aby v každé klauzuli byly nejvýše tři literály, to, že my vyžadujeme, aby byla každá klauzule složena z právě tří literálů, se nám bude hodit v dalších převodech. Ukážeme si, že i tento problém je stále NP-úplný. Použití trojky není náhodné, protože analogicky definovaný problém 2*SAT* je už polynomiálně řešitelný.

Věta 11.3.2 *Problém 3-SAT je* NP*-úplný.*

Důkaz: Problém 3-SAT je jen zvláštním případem problému SAT a z téhož důvodu jako v případě problému SAT, patří i 3-SAT do třídy NP. Těžkost problému 3-SAT ukážeme převodem z problému SAT. Nechť ψ je formule v KNF, vytvoříme z ní formuli φ, v níž každá klauzule bude obsahovat tři literály a pro níž bude platit, že ψ je splnitelná tehdy a jen tehdy, když φ je splnitelná.

Předpokládejme, že $\psi = C_1 \wedge \cdots \wedge C_m$. Uvažme libovolnou klauzuli $C_j = (e_1 \vee \cdots \vee e_k)$, kde $e_1, \ldots, e_k$ jsou literály (pozitivní či negativní). Tuto klauzuli nahradíme formulí v KNF α_j složenou z klauzulí o třech literálech, která bude obsahovat literály z C_j a k nim přidané nové literály, které se budou vyskytovat jen v α_j. Tato formule bude mít následující vlastnost. Je-li t ohodnocení splňující C_j, je možné ohodnotit přidané proměnné tak, aby i α_j byla splněna. Je-li naopak t' ohodnocení splňující α_j, pak t' splňuje i C_j. Podle délky klauzule, tedy hodnoty k, rozlišíme tyto případy:

- Pokud $k = 1$, pak nechť y_1^j, y_2^j jsou nové proměnné, které se nevyskytují v ψ, s nimiž definujeme

$$\alpha_j = (e_1 \vee y_1^j \vee y_2^j)(e_1 \vee y_1^j \vee \neg y_2^j)(e_1 \vee \neg y_1^j \vee y_2^j)(e_1 \vee \neg y_1^j \vee \neg y_2^j).$$

- Pokud $k = 2$, pak nechť y^j je nově přidaná proměnná, jež se nevyskytuje v ψ a

definujme

$$\alpha_j = (e_1 \vee e_2 \vee y^j)(e_1 \vee e_2 \vee \neg y^j).$$

- Pokud $k = 3$, pak definujeme

$$\alpha_j = C_j.$$

- Pokud $k > 3$, pak nechť $y_1^j, \ldots, y_{k-3}^j$ jsou nově přidané proměnné, které se nevyskytují v ψ, s nimiž definujeme

$$\alpha_j = (e_1 \vee e_2 \vee y_1^j)(\neg y_1^j \vee e_3 \vee y_2^j)(\neg y_2^j \vee e_4 \vee y_3^j)\cdots(\neg y_{k-3}^j \vee e_{k-1} \vee e_k).$$

Formuli φ nyní definujeme jako

$$\varphi = \bigwedge_{j=1}^{m} \alpha_j.$$

Z konstrukce okamžitě plyne, že formule φ je v KNF a navíc každá její klauzule obsahuje právě tři literály, navíc má φ polynomiální velikost vzhledem k velikosti ψ a lze ji i v polynomiálním čase zkonstruovat. Zbývá ukázat, že φ je splnitelná, právě když ψ je splnitelná.

Předpokládejme nejprve, že φ je splnitelná, nechť v je její splňující ohodnocení a nechť v' označuje ohodnocení proměnných ψ, jež jim přiřazuje touž hodnotu jako v, tj. pro každou proměnnou x původní formule ψ položíme $v'(x) := v(x)$. Nechť C_j je libovolná klauzule ψ, jí odpovídající formule α_j je ohodnocením v splněná. Nechť $C_j = (e_1 \vee \cdots \vee e_k)$, ukážeme, že C_j je ohodnocením v' splněna, dohromady tedy dostaneme, že $\psi(v) = 1$. Podle délky klauzule C_j, tedy hodnoty k, rozlišíme tyto případy:

- Pokud $k = 1$, pak jedna z klauzulí formule α_j musí být splněna díky e_1, tedy $v(e_1) = 1$. To proto, že dohromady obsahují klauzule α_j všechny možné kombinace literálů obsahujících y_1^j a y_2^j, v jedné z těchto kombinací se musí stát, že ohodnocení v vyhodnotí oba literály s y_1^j a y_2^j jako 0. Protože i klauzule, jež obsahuje tuto kombinaci, musí být splněna, platí, že $v(e_1) = 1$ (pokud je e_1 negativní, tak ohodnocení příslušné proměnné je ve skutečnosti 0).

- Pokud $k = 2$, pak opět jedna z klauzulí formule α_j musí vynutit, že $v(e_1) = 1$ nebo $v(e_2) = 1$, protože dohromady obsahují y^j i $\neg y^j$, přičemž tyto dva literály nemohou být současně splněny.

- Pokud $k = 3$, pak $\alpha_j = C_j$ a C_j je tedy splněna.

- Pokud $k > 3$, pak je každá klauzule z α_j splněna ohodnocením v. Pokud je $v[y_1^j] = 0$, pak v a tedy i v' ohodnocuje jeden z literálů e_1 a e_2 na 1. Podobně pokud je $v[y_{k-3}^j] = 1$, pak jeden z literálů e_{k-1} a e_k ohodnocuje v, tedy i v' na 1. Pokud je $v[y_1^j] = 1$ a $v[y_{k-3}^j] = 0$, pak existuje $1 \leq i \leq k-4$, pro něž je $v[y_i^j] = 1$ a $v[y_{i+1}^j] = 0$. Klauzule $(\neg y_i^j \vee e_{i+2} \vee y_{i+1}^j)$ patří do $\alpha(C)$ a je tedy splněna ohodnocením v. Proto musí být e_{i+2} ohodnocením v a tedy i v' ohodnoceno na 1. V každém případě jeden literál z $e_1, \ldots, e_k$ je ohodnocen v i v' na 1 a tedy klauzule C_j je splněna.

Nyní předpokládejme, že ψ je splnitelná, tedy existuje ohodnocení v, které ohodnocuje proměnné ψ a tato formule je jím splněna. Rozšíříme v o ohodnocení přidaných proměnných tak, aby nově vzniklé ohodnocení v' splnilo φ. Nechť $C_j = (e_1 \vee \cdots \vee e_k)$ je libovolná klauzule formule ψ. Pokud je $k \leq 3$, pak všechny klauzule α_j jsou splněny přímo ohodnocením v a na ohodnocení příslušných y-ových proměnných v těchto případech nezáleží. Pokud $k > 3$, pak nechť i je index literálu e_i, kterému ohodnocení v přiřadí 1. Položme $v'[y_r^j] = 1$ pro $r \leq i-2$ a $v'[y_r^j] = 0$ pro $r \geq i-1$. Pokud $i < 3$, pak v' přiřadí všem y-ovým proměnným 0 a pokud $i > k-2$, pak v' přiřadí všem y-ovým proměnným 1, v obou těchto případech jsou zřejmě všechny klauzule α_j splněny. Pokud je $3 \leq i \leq k-2$, pak klauzule α_j obsahující e_r pro $r < i$ jsou splněny díky pozitivnímu y-ovému literálu a klauzule obsahující e_r pro $r > i$ jsou splněny díky negativnímu y-ovému literálu, okrajové klauzule jsou splněny díky své jediné y-ové proměnné. Ta klauzule α_j, která obsahuje e_i, je splněna díky e_i. □

Jak jsme již zmínili, verze 2SAT, tj. splnitelnost formulí v KNF, kde každá klauzule obsahuje nejvýš 2 literály, je už polynomiálně rozhodnutelná. To platí i pro řadu dalších variant, za všechny zmiňme monotónní formule, tedy formule, které obsahují každou proměnnou jen pozitivně nebo jen negativně, tyto formule jsou vždy splnitelné. Další důležitou variantou je splnitelnost hornovských formulí, tedy problém HORN-SAT, jehož instance tvoří formule v KNF, kde každá klauzule obsahuje nejvýš jeden pozitivní literál.

11.3.2. Vrcholové pokrytí v grafu

V této části si ukážeme NP-úplnost problému vrcholového pokrytí v grafu.

Problém 11.3.3: Vrcholové pokrytí (VP)

Instance: Graf $G = (V, E)$ a přirozené číslo k

Otázka: Existuje množina $S \subseteq V$, která obsahuje nejvýš k vrcholů a z každé hrany obsahuje alespoň jeden koncový vrchol? (Tj. $|S| \leq k$ a $(\forall e \in E)[S \cap e \neq \varnothing]$.)

Věta 11.3.4 *Problém vrcholového pokrytí je* NP*-úplný.*

Důkaz: Fakt, že problém VP patří do třídy NP, plyne z toho, že pro danou množinu S dokážeme ověřit v polynomiálním čase, jde-li o vrcholové pokrytí správné velikosti. NP-těžkost dokážeme převodem z problému 3-SAT. Nechť formule $\varphi = C_1 \wedge \cdots \wedge C_m$ s proměnnými $U = \{u_1, \ldots, u_n\}$ je instancí problému 3-SAT, tedy každá klauzule $C_1, \ldots, C_m$ obsahuje právě tři literály. Popíšeme, jak na základě φ zkonstruovat graf $G = (V, E)$ a číslo k, pro něž bude platit, že v G existuje vrcholové pokrytí velikosti nejvýš k, právě když φ je splnitelná.

Pro každou proměnnou $u_i \in U$ definujme podgraf $T_i = (V_i, E_i)$ s $V_i = \{u_i, \neg u_i\}$ a $E_i = \{\{u_i, \neg u_i\}\}$, tj. T_i obsahuje jen dva vrcholy pro pozitivní a negativní literál obsahující u_i a

hranu mezi těmito dvěma vrcholy. Vrcholové pokrytí musí z této hrany využít alespoň jeden vrchol, má-li pokrýt hranu v E_i.

Pro každou klauzuli C_j, $j = 1, \ldots, m$ přidáme úplný podgraf na třech nových vrcholech $a_1[j]$, $a_2[j]$, $a_3[j]$. Jde tedy trojúhelník, $R_j = (V'_j, E'_j)$, kde

$$V'_j = \{a_1[j], a_2[j], a_3[j]\}$$
$$E'_j = \{\{a_1[j], a_2[j]\}, \{a_1[j], a_3[j]\}, \{a_2[j], a_3[j]\}\}$$

Všimněme si, že vrcholové pokrytí musí obsahovat alespoň dva vrcholy z V'_j, má-li pokrýt všechny hrany z E'_j.

Zbývá propojit tyto grafy mezi sebou hranami, jež spojují vrcholy pro literály s jejich výskyty v klauzulích. Předpokládejme, že pro každé $j = 1, \ldots, m$, je klauzule $C_j = (x_j \vee y_j \vee z_j)$, kde x_j, y_j, z_j jsou tři různé literály (pozitivní nebo negativní). Pak přidáme hrany z R_j do příslušných vrcholů literálů, tedy množinu hran:

$$E''_j = \{\{a_1[j], x_j\}, \{a_2[j], y_j\}, \{a_3[j], z_j\}\}$$

V naší konstrukci nakonec položíme $k = n + 2m$ a $G = (V, E)$, kde

$$V = \bigcup_{i=1}^{n} V_i \cup \bigcup_{i=1}^{m} V'_i$$
$$E = \bigcup_{i=1}^{n} E_i \cup \bigcup_{i=1}^{m} E'_i \cup \bigcup_{i=1}^{m} E''_i$$

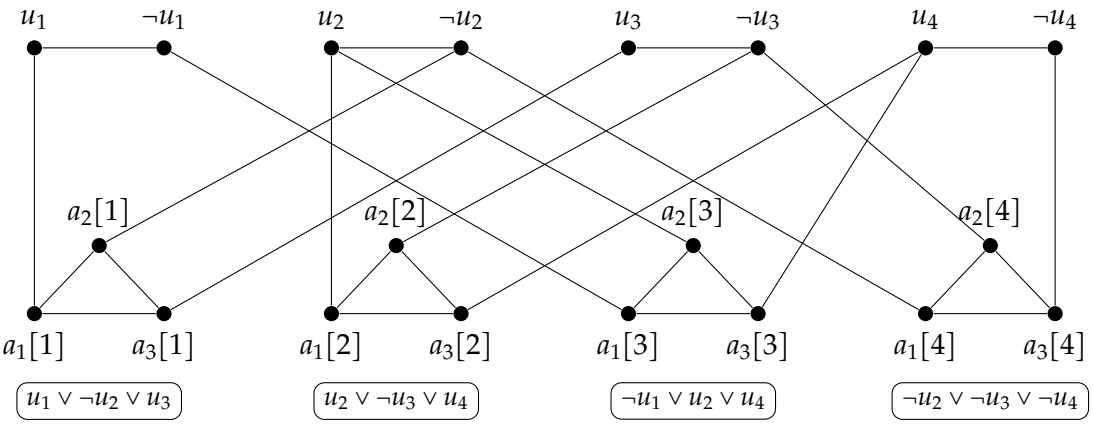
Příklad konstrukce můžete nahlédnout v obrázku 11.2.

Z popisu konstrukce je zřejmé, že ji lze provést v polynomiálním čase. Zbývá ukázat, že G má vrcholové pokrytí velikosti nejvýš k, právě když φ je splnitelná.

Předpokládejme nejprve, že G má vrcholové pokrytí $S \subseteq V$ velikosti $k = n + 2m$. Protože S musí pokrýt alespoň jeden vrchol z každého podgrafu T_i, $i = 1, \ldots, n$ a alespoň dva vrcholy z každého trojúhelníku R_j, $j = 1, \ldots, m$, musí platit, že $|S| \geq n + 2m$, a protože současně $|S| \leq n + 2m$, musí ve skutečnosti platit, že S pokrývá právě jeden vrchol z každého T_i a právě dva vrcholy z každého R_i. Definujme ohodnocení $t : U \to \{0, 1\}$, které dané proměnné $u_i \in U$ přiřadí

$$t(u_i) = \begin{cases} 1 & u_i \in S \\ 0 & \neg u_i \in S \end{cases}$$

Díky tomu, že z každého T_i je v pokrytí S právě jeden vrchol, je v definici $t(u_i)$ vždy splněna právě jedna z podmínek a jde tedy o dobře definované ohodnocení proměnných pravdivostními hodnotami. Nechť $C_j = (x_j \vee y_j \vee z_j)$ je libovolná klauzule formule φ. Množina S pokrývá právě dva vrcholy z trojúhelníku R_j, existuje v něm tedy jeden vrchol, který do S nepatří, nechť je to bez újmy na obecnosti $a_1[j]$, protože hrana $\{a_1[j], x_j\}$ musí být pokryta množinou S, musí platit, že $x_j \in S$ a tedy ohodnocení t přiřadí literálu



Obrázek 11.2.: Příklad převodu formule $\varphi = (u_1 \vee \neg u_2 \vee u_3) \wedge (u_2 \vee \neg u_3 \vee u_4) \wedge (\neg u_1 \vee u_2 \vee u_4) \wedge (\neg u_2 \vee u_3 \vee \neg u_4)$ na instanci vrcholového pokrytí, tedy graf na obrázku s $k = 2.4 + 4 = 12$.

x_j hodnotu 1, tj. pokud $x_j = u_i \in S$ pro nějakou proměnnou $u_i \in U$, platí $t(u_i) = 1$ a pokud $x_j = \neg u_i \in S$ pro nějakou proměnnou $u_i \in U$, platí $t(u_i) = 0$, v obou případech je literál x_j splněn ohodnocením t, a je tedy splněna i klauzule C_j. Protože to platí pro všechny klauzule, je t splňující ohodnocení φ.

Nyní předpokládejme, že φ je splněná ohodnocením $t : U \to \{0, 1\}$ a definujme množinu $S = \{u_i \mid t(u_i) = 1\} \cup \{\neg u_i \mid t(u_i) = 0\}$, množina obsahuje n vrcholů. Mezi hranami z E''_j vedoucími z každého trojúhelníku R_j musí být alespoň jedna pokrytá nějakým vrcholem z S, neboť t je splňující ohodnocení, které tedy splňuje některý literál z klauzule C_j. Z každého trojúhelníku stačí tedy vybrat dva vrcholy tak, aby byly pokryty jak hrany z E'_j, tak hrany z E''_j. Přidáním těchto vrcholů do S dostaneme vrcholové pokrytí celého grafu velikosti $k = n + 2m$. □

Problémy, které s vrcholovým pokrytím úzce souvisí, jsou problémy kliky a nezávislé množiny, jejich těžkost si ukážeme v rámci cvičení. Je zajímavé poznamenat, že hranová verze tohoto problému, tedy hranové pokrytí, kde hledáme co nejmenší množinu hran takovou, že každý vrchol patří do jedné z nich, je polynomiálně řešitelná pomocí algoritmu na hledání maximálního párování v grafu.

11.3.3. Hamiltonovská kružnice v grafu

I další problém, kterému se budeme věnovat, bude grafový.

Problém 11.3.5: Hamiltonovská kružnice (HK)

Instance: Graf $G = (V, E)$.
Otázka: Existuje v grafu G cyklus vedoucí přes všechny vrcholy?

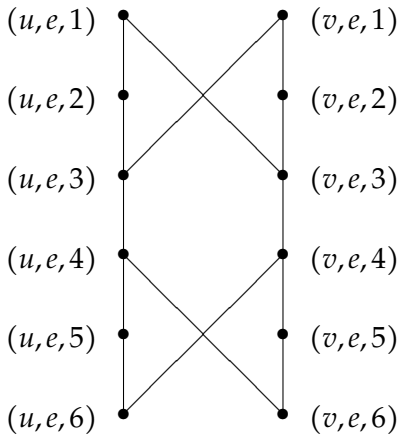
Věta 11.3.6 *Problém hamiltonovské kružnice je* NP*-úplný.*

Důkaz: Máme-li k dispozici pořadí vrcholů, snadno v polynomiálním čase ověříme, tvoří-li hamiltonovskou kružnici, proto patří problém hamiltonovské kružnice do třídy NP.

Těžkost tohoto problému ukážeme převodem z problému vrcholového pokrytí. Uvažme instanci vrcholového pokrytí, tedy graf $G = (V, E)$ a přirozené číslo k. Zkonstruujeme nový graf $G' = (V', E')$, pro který bude platit, že v G' existuje hamiltonovská kružnice, právě když v G existuje vrcholové pokrytí velikosti k. Pro každou hranu $e = \{u, v\} \in E$ definujeme podgraf $G'_e = (V'_e, E'_e)$, kde

$$\begin{aligned}
V'_e &= \{(u,e,i),(v,e,i) \mid 1 \le i \le 6\} \text{ a} \\
E'_e &= \Big\{\{(u,e,i),(u,e,i+1)\} \mid 1 \le i \le 5\Big\} \\
&\cup \Big\{\{(v,e,i),(v,e,i+1)\} \mid 1 \le i \le 5\Big\} \\
&\cup \Big\{\{(u,e,1),(v,e,3)\},\{(u,e,3),(v,e,1)\}, \\
&\quad \{(u,e,4),(v,e,6)\},\{(u,e,6),(v,e,4)\}\Big\}.
\end{aligned}$$

Graf G'_e je zobrazen na následujícím obrázku:



Jediné vrcholy, k nimž v další konstrukci připojíme další hrany budou $(u,e,1)$, $(u,e,6)$, $(v,e,1)$ a $(v,e,6)$, což zaručí, že hamiltonovská kružnice musí vstoupit i vystoupit z podgrafu G'_e jedním z těchto čtyř vrcholů. Jsou jen tři způsoby, jak může tato cesta vypadat, ty budou odpovídat tomu, jak pokrýt hranu e v grafu G.

(I) Cesta vstoupí do G'_e vrcholem $(u, e, 1)$, vystoupí $(u, e, 6)$ a mezitím projde všechny vrcholy (cesta je jednoznačně daná).

(II) Nebo cesta vstoupí do G'_e vrcholem $(v, e, 1)$, vystoupí $(v, e, 6)$ a mezitím projde všemi vrcholy komponenty (toto je symetrické s předchozím případem).

(III) Třetí možností je, že kružnice do této komponenty vstoupí dvakrát, jednou vrcholem $(u, e, 1)$, odkud jde přímo do $(u, e, 6)$ a ven, podruhé vstoupí do $(v, e, 1)$ a odejde pomocí $(v, e, 6)$.

Vzhledem k tomu, že graf je neorientovaný, nezáleží na tom, jestli například v prvním případě vstoupí cesta do G'_e vrcholem $(u, e, 1)$ a vystoupí $(u, e, 6)$ nebo naopak. Všimněme si také, že pokud vstoupí cesta do G'_e vrcholem $(v, e, 1)$, vystoupí ve všech případech vrcholem $(v, e, 6)$, podobně to platí pro $(u, e, 1)$.

Do množiny vrcholů V' nového grafu G' přidáme ještě k vrcholů $a_1, \ldots, a_k$, jež použijeme k výběru k vrcholů vrcholového pokrytí. Zbývá popsat, jak spojíme jednotlivé podgrafy G'_e a nově přidané vrcholy a_i.

Pro každý vrchol $v \in V$ zapojíme jednotlivé grafy G'_e s $v \in e$ za sebe do řetízku. Přesněji, označme si stupeň vrcholu v pomocí $\deg(v)$ a označme si hrany grafu G, v nichž se vyskytuje vrchol v pomocí $e_{v[1]}, \ldots, e_{v[\deg(v)]}$, přičemž na jejich pořadí nezáleží. Nyní definujme množinu hran

$$E'_v = \Big\{ \{(v, e_{v[i]}, 6), (v, e_{v[i+1]}, 1)\} \mid 1 \leq i < \deg(v) \Big\}.$$

Vrcholy na koncích této posloupnosti, tedy $(v, e_{v[1]}, 1)$ a $(v, e_{v[\deg(v)]}, 6)$ připojíme ke všem výběrovým vrcholům $a_1, \ldots, a_k$ hranami

$$E''_v = \Big\{ \{a_i, (v, e_{v[1]}, 1)\}, \{a_i, (v, e_{v[\deg(v)]}, 6)\} \mid 1 \leq i \leq k \Big\}.$$

Zkonstruovaný graf $G' = (V', E')$ vznikne spojením popsaných částí:

$$V' = \{a_1, \ldots, a_k\} \cup \bigcup_{e \in E} V'_e$$
$$E' = \bigcup_{e \in E} E'_e \cup \bigcup_{v \in V} E'_v \cup \bigcup_{v \in V} E''_v$$

Tuto konstrukci je zřejmě možné provést v polynomiálním čase, zbývá ukázat, že v grafu G' najdeme hamiltonovskou kružnici, právě když v grafu G existuje vrcholové pokrytí velikosti k.

Předpokládejme nejprve, že v G' existuje hamiltonovská kružnice, daná pořadím vrcholů $u_1, \ldots, u_n$, kde $n = |V'|$. Uvažme úsek této kružnice, který začíná a končí v některém z vrcholů $a_1, \ldots, a_k$, přičemž mezi tím žádným z nich neprochází. Nechť tento úsek začíná v a_i a končí v a_j pro dva různé indexy $i, j \in \{1, \ldots, k\}$. Vrchol následující po a_i v hamiltonovské kružnici musí být $(v, e_{v[1]}, 1)$ pro nějaký vrchol $v \in V$. Vejde-li hamiltonovská kružnice vrcholem $(v, e_{v[1]}, 1)$ do $G'_{e_{v[1]}}$, musí z něj vyjít vrcholem $(v, e_{v[1]}, 6)$,

odtud přejde do $(v, e_{v[2]}, 1)$, takto projde všechny grafy G'_e pro hrany e obsahující vrchol v až nakonec přejde z vrcholu $(v, e_{v[\deg(v)]}, 6)$ do a_j. Definujme nyní množinu vrcholů

$$S = \{v \mid (\exists i \in \{1, \ldots, k\})(\exists j \in \{1, \ldots, n\})\ [(u_j = a_i)\ \&\ (u_{(j+1 \bmod n)} = (v, e_{v[1]}, 1))]\},$$

tvrdíme, že jde o vrcholové pokrytí v grafu G, přičemž jeho velikost je zřejmě k. Nechť $e = \{u, v\}$ je libovolná hrana grafu G, protože $u_1, \ldots, u_n$ určuje pořadí vrcholů na hamiltonovské kružnici, musí projít i podgraf G'_e, vejde-li do tohoto podgrafu vrcholem $(u, e, 1)$, pak musí z předchozích úvah platit, že $u \in S$, podobně pokud vejde vrcholem $(v, e, 1)$, musí platit, že $v \in S$, tedy hrana e je pokryta.

Naopak uvažme vrcholové pokrytí S velikosti k, můžeme uvažovat, že jde o množinu velkou přesně k, pokud je menší, doplníme do ní libovolné vrcholy tak, aby její velikost byla k. Z popisu konstrukce plyne, jak pospojujeme jednotlivé podgrafy odpovídající hranám. Předpokládejme, že $S = \{v_1, \ldots, v_k\} \subseteq V$. Mezi vrcholy a_i a a_{i+1} (resp. a_1 pokud $i = k$) povedeme cestu přes podgrafy G'_e pro $e = e_{v_i[1]}, \ldots, e_{v_i[\deg(v_i)]}$ v tomto pořadí. Graf G'_e přitom projdeme způsobem (I) nebo (II) pokud $e \cap S = \{v_i\}$, pokud platí $e \subseteq S$, pak použijeme způsob (III). Vzhledem k tomu, že S tvoří vrcholové pokrytí, projdeme takto všechny vrcholy grafu G' a vznikne tedy hamiltonovská kružnice. □

Podotkněme, že problém hamiltonovské kružnice má několik variant. Předně se můžeme ptát na existenci hamiltonovské kružnice v orientovaném grafu. V problému hamiltonovské cesty se ptáme, existuje-li v grafu cesta mezi dvěma vrcholy, jež jde přes všechny vrcholy grafu, orientovaného či neorientovaného, přičemž počáteční a koncový vrchol hledané cesty mohou být předepsané na vstupu. Všechny tyto varianty jsou NP-úplné. Na druhou stranu rozhodnutí zda graf obsahuje eulerovský tah, tedy tah, který použije každou hranu právě jednou, lze učinit v polynomiálním čase, ať už se jedná o tah uzavřený či neuzavřený.

11.3.4. Trojrozměrné párování

Dalším problémem, jehož těžkost si ukážeme, bude trojrozměrné párování (3D Matching).

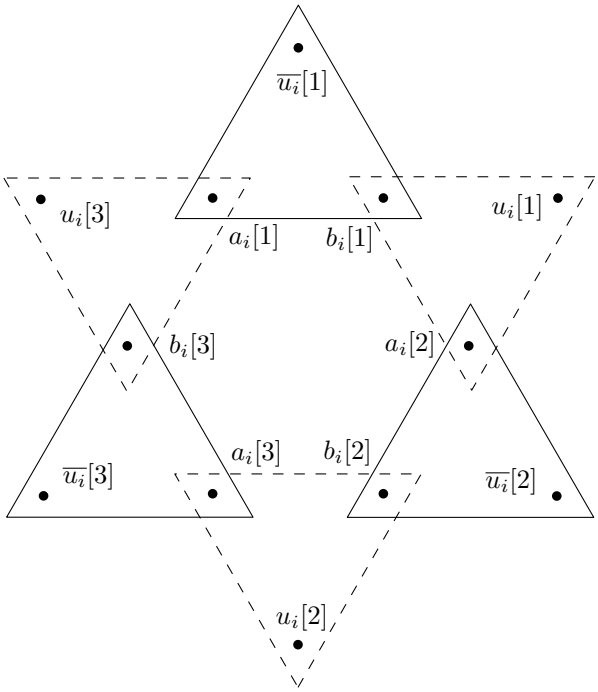
Problém 11.3.7: Trojrozměrné párování (3DM)

Instance: Množina $M \subseteq W \times X \times Y$, kde W, X, Y jsou po dvou disjunktní množiny, z nichž každá obsahuje právě q prvků.

Otázka: Obsahuje M perfektní párování? Jinými slovy, existuje množina $M' \subseteq M$, $|M'| = q$, trojice v níž obsažené jsou po dvou disjunktní?

Věta 11.3.8 *Problém 3DM je* NP*-úplný.*

Důkaz: Fakt, že tento problém patří do třídy NP, plyne z toho, že pokud máme k dispozici množinu $M' \subseteq M$, dokážeme ověřit v polynomiálním čase, jde-li o párování velikosti q.



Obrázek 11.3.: Ukázka množin T_i^t (plnou čarou) a T_i^f (čárkovaně) pro případ $m = 3$.

Těžkost tohoto problému si ukážeme převodem z problému SAT. Nechť $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ je formule v KNF na proměnných $U = \{u_1, \ldots, u_n\}$. Sestrojíme instanci problému 3DM, pro kterou bude platit, že v této instanci existuje perfektní párování, právě když je φ splnitelná. Konstrukci rozdělíme na tři části, nejprve vytvoříme komponentu, která bude určovat, jakou hodnotu která proměnná dostane, poté vytvoříme komponentu, která bude zajišťovat propojení této hodnoty s klauzulemi, v nichž se tato proměnná vyskytuje, nakonec doplníme trojice tak, abychom dostali k splňujícímu ohodnocení skutečně perfektní párování a naopak.

Za každou proměnnou u_i, $i = 1, \ldots, n$ přidáme nové vnitřní prvky $a_i[1], \ldots, a_i[m]$ do X a $b_i[1], \ldots, b_i[m]$ do Y. Do W přidáme prvky $u_i[1], \ldots, u_i[m]$ a $\overline{u}_i[1], \ldots, \overline{u}_i[m]$. Na těchto prvcích vytvoříme množiny trojic T_i^f a T_i^t takto (viz též příklad na obrázku 11.3):

$$T_i^t = \{(\overline{u}_i[j], a_i[j], b_i[j]) \mid 1 \leq j \leq m\}$$
$$T_i^f = \{(u_i[j], a_i[(j \bmod m) + 1], b_i[j]) \mid 1 \leq j \leq m\}$$
$$T_i = T_i^t \cup T_i^f$$

Protože žádný z prvků $a_i[j]$ ani $b_i[j]$ se nebude vyskytovat v jiných trojicích, je tímto vynuceno, že perfektní párování buď musí obsahovat všechny trojice z T_i^t, nebo všechny trojice z T_i^f. Pokud obsahuje trojice z T_i^t, znamená to, že žádná další vybraná trojice nesmí obsahovat literál $\overline{u}_i$, tedy vynucujeme hodnotu 1, true pro u_i, proto také T_i^t. Podobně pokud obsahuje perfektní párování trojice z T_i^f, vynucujeme hodnotu 0, false, odtud T_i^f.

Za klauzuli C_j přidáme nový prvek $s_1[j]$ do množiny X, nový prvek $s_2[j]$ do množiny Y a množinu trojic

$$S_j = \{(u_i[j], s_1[j], s_2[j]) \mid u_i \in C_j\} \cup \{(\overline{u}_i[j], s_1[j], s_2[j]) \mid \overline{u}_i \in C_j\}$$

Prvky $s_1[j]$ a $s_2[j]$ se opět nebudou vyskytovat v jiných trojicích, díky tomu v perfektním párování musí být právě jedna trojice z množiny S_j. Navíc pokud se trojice $(u_i[j], s_1[j], s_2[j])$ vyskytuje v perfektním párování, znamená to, že $u_i[j]$ se nemůže vyskytovat v jiné trojici a to znamená, že v tomto párování jsou všechny trojice z T_i^t a žádná z T_i^f. Podobně by to bylo, kdyby v perfektním párování byla trojice s negativním literálem.

Těmito trojicemi jsme ale schopni v perfektním párování pokrýt jen $mn + m$ prvků z $2mn$ prvků $u_i[j], \overline{u}_i[j]$, $i = 1, \ldots, n$, $j = 1, \ldots, m$. Z toho mn jich pokryjeme trojicemi z T_i^t nebo T_i^f, $i = 1, \ldots, n$. Trojicemi z S_j, $j = 1, \ldots, m$ pokryjeme dalších m prvků. Zbývá tedy $2mn - (mn + m) = mn - m = m(n-1)$ prvků, jež nejsme zatím schopni pokrýt, proto přidáme do množiny M trojice, které nám jejich pokrytí zabezpečí. Do X přidáme prvky $g_1[k]$ a do Y prvky $g_2[k]$ obojí pro $k = 1, \ldots, m(n-1)$ a do M přidáme množinu trojic

$$G = \{(u_i[j], g_1[k], g_2[k]), (\overline{u}_i[j], g_1[k], g_2[k]) \mid 1 \le k \le m(n-1), 1 \le i \le n, 1 \le j \le m\}$$

Tím je konstrukce dokončena, ještě si na závěr shrňme její výsledek:

$$\begin{aligned}
W &= \{u_i[j], \overline{u}_i[j] \mid 1 \le i \le n, 1 \le j \le m\} \\
X &= \{a_i[j] \mid 1 \le i \le n, 1 \le j \le m\} \\
&\quad \cup \{s_1[j] \mid 1 \le j \le m\} \\
&\quad \cup \{g_1[j] \mid 1 \le j \le m(n-1)\} \\
Y &= \{b_i[j] \mid 1 \le i \le n, 1 \le j \le m\} \\
&\quad \cup \{s_2[j] \mid 1 \le j \le m\} \\
&\quad \cup \{g_2[j] \mid 1 \le j \le m(n-1)\} \\
M &= \bigcup_{i=1}^{n} T_i \cup \bigcup_{j=1}^{m} S_j \cup G
\end{aligned}$$

Zřejmě platí, že $M \subseteq W \times X \times Y$, navíc $|M| = 2mn + 3m + 2m^2n(n-1)$ a $|W| = |X| = |Y| = 2mn$. Velikost takto vytvořené instance trojrozměrného párování je tedy polynomiálně velká a v polynomiálním čase ji zřejmě lze i vytvořit.

Předpokládejme, že v M existuje perfektní párování M', na jehož základě zkonstruujeme ohodnocení t, které bude splňovat formuli φ. Nechť i je libovolný index z $1, \ldots, n$, jak jsme již zdůvodnili, v M' jsou buď všechny trojice z T_i^t, nebo všechny trojice z T_i^f, pokud M' obsahuje trojice z T_i^t, definujeme $t(u_i) := 1$, pokud M' obsahuje trojice z T_i^f, definujeme $t(u_i) := 0$. Nechť C_j je libovolná klauzule formule φ, množina M' musí obsahovat právě jednu z trojic z S_j, neboť to je jediná možnost, jak mohou být pokryty prvky $s_1[j]$ a $s_2[j]$ a dvě obsahovat nemůže, protože každý z těchto prvků musí být pokryt právě jednou. Nechť tato trojice je $(u_i[j], s_1[j], s_2[j])$ pro nějaké $i = 1, \ldots, n$. To znamená, že u_i

je proměnná, vyskytující se jako pozitivní literál v klauzuli C_j, navíc musí platit, že $u_i[j]$ se nemůže vyskytovat v žádné jiné trojici v M', a proto M' obsahuje trojice z T_i^t a nikoli trojice z T_i^f, a tedy $t(u_i) = 1$, čímž je klauzule C_j splněna. Podobně bychom postupovali v případě, kdy trojicí v $S_j \cap M'$ by byla $(\neg u_i[j], s_1[j], s_2[j])$, tedy pokud by obsahovala negativní literál, jediný rozdíl by byl, že bychom dostali, že $t(u_i) = 0$ a že C_j je splněna díky negativnímu literálu $\neg u_i$.

Pokud je naopak φ splnitelná, zkonstruujeme perfektní párování následovně. Nechť $t : U \to \{0,1\}$ je ohodnocení splňující φ a nechť z_j označuje literál, který je v C_j tímto ohodnocením splněn pro $j = 1, \ldots, m$. Pokud je takových literálů víc, vybereme prostě jeden z nich. Pak položíme

$$M' = \bigcup_{t(u_i)=1} T_i^t \cup \bigcup_{t(u_i)=0} T_i^f \cup \left\{(z_j[j], s_1[j], s_2[j]) \mid j = 1, \ldots, m\right\} \cup G',$$

kde G' je množina vhodně vybraných trojic z G, které doplňují párování o pokrytí zbylých literálů. Není těžké ověřit, že takto definovaná množina M' tvoří perfektní párování M. □

Je třeba připomenout, že dvojrozměrná verze, tedy hledání maximálního párování v bipartitním grafu, je úloha řešitelná v polynomiálním čase například pomocí toků v sítích. Podobně i hledání maximálního párování v obecném grafu je stále polynomiálně řešitelná úloha.

11.3.5. Loupežníci

Posledním problémem, jehož těžkost si ukážeme, je problém Loupežníci, anglicky Partition. Český název pochází z představy, že jde o dělení lupu dvou loupežníků na shodné díly.

Problém 11.3.9: Loupežníci (*LOUP*)

Instance: Množina prvků A a s každým prvkem $a \in A$ asociovaná cena (váha, velikost, ...) $s(a) \in \mathbb{N}$.

Otázka: Lze rozdělit prvky z A na dvě poloviny s toutéž celkovou cenou? Přesněji, existuje množina $A' \subseteq A$ taková, že

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a) \; ?$$

Věta 11.3.10 *Problém Loupežníci je* NP*-úplný.*

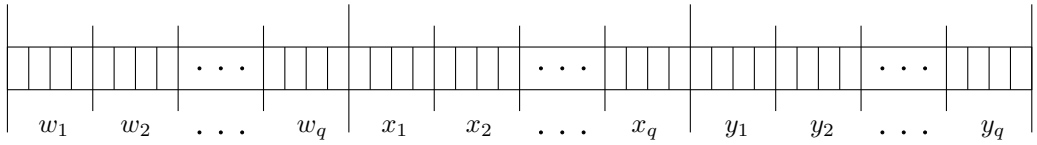
Důkaz: Fakt, že tento problém patří do třídy NP, plyne z toho, že pro zadanou množinu A' je jistě snadné ověřit, zda obsahuje prvky poloviční ceny. Těžkost tohoto problému ukážeme převodem z trojrozměrného párování. Uvažme instanci 3DM, tedy množinu

$M \subseteq W \times X \times Y$, kde $|W| = |X| = |Y| = q$. Vytvoříme instanci Loupežníků, tedy množinu A a cenovou funkci $s(a)$, pro něž bude platit, že M má perfektní párování, právě když prvky v A lze rozdělit na dvě části s touž celkovou cenou.

Předpokládejme, že $M = \{m_1, \ldots, m_k\}$, $W = \{w_1, \ldots, w_q\}$, $X = \{x_1, \ldots, x_q\}$ a $Y = \{y_1, \ldots, y_q\}$. Jednotlivé prvky trojice m_i si označíme jako $w_{f(i)}$, $x_{g(i)}$ a $y_{h(i)}$, tj. funkce f (resp. h, g) vrátí k zadanému indexu trojice i index toho prvku trojice m_i, který patří do množiny W (resp. X, Y). Množinu prvků A definujeme jako

$$A = \{a_1, \ldots, a_k, b_1, b_2\},$$

kde prvky $a_1, \ldots, a_k$ odpovídají trojicím $m_1, \ldots, m_k$ a prvky b_1 a b_2 jsou pomocné a vyrovnávací, jejich účel se ozřejmí později. Cenu $s(a_i)$ prvku a_i pro $i \in \{1, \ldots, k\}$ popíšeme její binární reprezentací, která bude rozdělena na $3q$ bloků, z nichž každý má $p = \lceil \log_2(k+1) \rceil$ bitů. Každý z těchto bloků bude odpovídat jednomu z elementů $W \cup X \cup Y$, přesněji viz obrázek 11.4.



Obrázek 11.4.: Označení zón bitů v definici ceny $s(a_i)$ prvku a_i.

Reprezentace $s(a_i)$ bude záviset jen na prvcích trojice m_i, tedy na $w_{f(i)}$, $x_{g(i)}$ a $y_{h(i)}$. Váha $s(a_i)$ bude mít nastaveny na 1 nejpravější (tj. nejméně významné) bity v blocích označených těmito třemi prvky, ostatní bity budou nulové. Formálně můžeme tento fakt zapsat jako

$$s(a_i) = 2^{p(3q-f(i))} + 2^{p(2q-g(i))} + 2^{p(q-h(i))}.$$

Protože počet bitů, který potřebujeme na reprezentaci $s(a_i)$ je $3pq$ a tedy polynomiální vzhledem k velikosti vstupu, předpokládáme-li standardní binární reprezentaci vstupních čísel, lze tyto ceny zkonstruovat v polynomiálním čase. O takto zkonstruovaných cenách lze vypozorovat jeden důležitý fakt. Pokud posčítáme ceny všech prvků v kterékoli zóně, nemůže dojít k přetečení do další zóny, neboť jde o sečtení nejvýš k jedniček, přičemž počet bitů v jedné zóně je $p = \lceil \log_2(k+1) \rceil$ a vejde se do ní tedy i $k+1$. Přesněji ani při počítání součtu $\sum_{i=1}^{k} s(a_i)$ nedojde k přenosu jedničky z méně významné zóny do významnější.

Pokud tedy položíme

$$B = \sum_{j=0}^{3q-1} 2^{pj},$$

což je číslo, kde v každé zóně nastavíme na 1 nejméně významný bit, množina $A' \subseteq \{a_i \mid 1 \leq i \leq k\}$ bude splňovat

$$\sum_{a \in A'} s(a) = B,$$

právě když $M' = \{m_i \mid a_i \in A'\}$ je perfektní párování M. V tuto chvíli jsme tedy učinili převod problému trojrozměrného párování na problém součtu podmnožiny, kde se ptáme, zda existuje výběr prvků s celkovou cenou rovnou zadané hodnotě.

Abychom dostali dělení na poloviny, doplníme do A dva prvky b_1 a b_2 s cenami:

$$s(b_1) = 2\Big(\sum_{i=1}^{k} s(a_i)\Big) - B$$

$$s(b_2) = \Big(\sum_{i=1}^{k} s(a_i)\Big) + B$$

Na reprezentaci obou těchto vah nám postačí $3pq+1$ bitů. Pokud nyní $A' \subseteq A$ splňuje, že

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a),$$

pak se oba součty musí rovnat $2\sum_{i=1}^{k} s(a_i)$, neboť to je polovina ze součtu cen všech prvků $\sum_{i=1}^{k} s(a_i)+s(b_1)+s(b_2)$. Přitom platí, že prvky b_1 a b_2 se nemohou oba vyskytovat na jedné straně, tj. nemohou být oba v A' nebo oba v $A \setminus A'$, protože $s(b_1) + s(b_2) = 3\sum_{i=1}^{k} s(a_i)$. Bez újmy na obecnosti předpokládejme, že $b_1 \in A'$ a $b_2 \in A \setminus A'$, z toho plyne, že

$$\sum_{a \in A' \setminus \{b_1\}} s(a) = B,$$

a prvky v A' bez b_1 tedy určují perfektní párování v M.

Nyní předpokládejme, že $M' \subseteq M$ je perfektní párování a definujme $A' = \{a_i \mid m_i \in M'\}$. Jak jsme již zdůvodnili, musí platit, že $\sum_{a \in A'} s(a) = B$, a tedy $\sum_{a \in A'} s(a) + s(b_1) = 2\sum_{i=1}^{k} s(a_i)$, což znamená, že množina $\{a_i \mid m_i \in M'\} \cup \{b_1\}$ obsahuje prvky právě poloviční ceny.

Dohromady jsme ukázali, že v množině trojic M existuje perfektní párování, právě když z množiny A lze vybrat prvky poloviční ceny. □

11.4. Cvičení

Důkazy NP-úplnosti

1. Ukažte, že problémy Kliky a Nezávislé množiny jsou stejně těžké a že oba patří do NP.

Problém 11.4.1: Klika

Instance: Graf $G = (V, E)$ a přirozené číslo k.
Otázka: Obsahuje G jako podgraf úplný podgraf (kliku) s alespoň k vrcholy?

Problém 11.4.2: Nezávislá množina

Instance: Graf $G = (V, E)$ a přirozené číslo k.

Otázka: Existuje množina $S \subseteq V$ vrcholů taková, že žádné dva vrcholy z S nejsou spojeny hranou a $|S| \geq k$?

2. Ukažte, že problém vrcholového pokrytí, který jsme probírali na přednášce, lze polynomiálně převést na problém nezávislé množiny a že jsou tedy problémy Kliky a Nezávislé množiny NP-úplné.

3. Ukažte, že problém hamiltonovské kružnice, který jsme probírali na přednášce, lze polynomiálně převést na problém orientované hamiltonovské kružnice a na následující dvě varianty problému hamiltonovské cesty a že jsou tedy oba tyto problémy NP-těžké. Ukažte, že jsou tyto problémy i ve třídě NP a jsou tedy NP--úplné.

Problém 11.4.3: Orientovaná hamiltonovská kružnice (OHK)

Instance: Orientovaný graf $G = (V, E)$.

Otázka: Existuje v G cyklus, který projde každý vrchol právě jednou?

Problém 11.4.4: Hamiltonovská cesta z s do t ($HC(s, t)$)

Instance: Neorientovaný graf $G = (V, E)$, vrcholy $s, t \in V$.

Otázka: Existuje v G cesta z s do t, která obsahuje každý vrchol G právě jednou?

Problém 11.4.5: Hamiltonovská cesta (HC)

Instance: Neorientovaný graf $G = (V, E)$.

Otázka: Existuje v G cesta, která obsahuje každý vrchol G právě jednou?

4. Ukažte, že problém obchodního cestujícího je NP-úplný. K důkazu těžkosti využijte problému hamiltonovské kružnice.

Problém 11.4.6: Obchodní cestující (OC)

Instance: Množina měst $C = \{c_1, \ldots, c_n\}$, vzdálenost $d(c_i, c_j) \in \mathbb{N}$ pro každá dvě města $c_i, c_j \in C$, přirozené číslo B

Otázka: Existuje cyklus, který navštíví každé město právě jednou a jehož délka nepřesahuje B? Jinými slovy, existuje permutace $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ taková, že

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}) \leq B?$$

5. Ukažte, že problémy existence přípustného řešení celočíselného programování a 0–1 celočíselného programování jsou NP-těžkosti. K důkazu použijte některý z následujících problémů: Kachlíkování, Splnitelnost, 3-Splnitelnost, Vrcholové pokrytí, Trojrozměrné párování, Hamiltonovská kružnice, Obchodní cestující nebo Loupežníci.

Problém 11.4.7: Existence přípustného řešení celočíselného programování (*IP*)

Instance: Celočíselná matice A typu $n \times m$, vektor $b \in \mathbb{Z}^n$.

Otázka: Existuje vektor $x \in \mathbb{Z}^m$, který splňuje, že $Ax \geq b$?

Problém 11.4.8: Existence přípustného řešení celočíselného programování (0–1*IP*)**

Instance: Celočíselná matice A typu $n \times m$, vektor $b \in \mathbb{Z}^n$.

Otázka: Existuje vektor $x \in \{0, 1\}^m$, který splňuje, že $Ax \geq b$?

Ukažte navíc, že problém 0–1IP patří do NP, proč není tak jednoduché ukázat, že i obecná verze IP patří do NP?

6. S použitím problému Loupežníci ukažte, že problémy Batoh a Rozvrhování jsou NP-úplné.

Problém 11.4.9: Batoh

Instance: Množina předmětů U, s každým prvkem $u \in U$ jeho velikost $s(u) \in \mathbb{N}$ a cena $v(u) \in \mathbb{N}$, přirozená čísla B a K.

Otázka: Existuje množina $U' \subseteq U$, pro kterou platí, že

$$\sum_{u \in U'} s(u) \leq B \quad \text{a} \quad \sum_{u \in U'} v(u) \geq K \ ?$$

tj. lze do batohu velikosti B zabalit předměty s cenou alespoň K?

Problém 11.4.10: Rozvrhování

Instance: Množina úloh U, s každou úlohou $u \in U$ asociovaná délka zpracování $d(u) \in \mathbb{N}$, počet procesorů $m \in \mathbb{N}$ a limit na délku zpracování $D \in \mathbb{N}$.

Otázka: Lze rozdělit prvky z množiny U do po dvou disjunktních množin $U_1, \ldots, U_m$ tak, aby

$$\max\{\sum_{u \in U_i} d(u) \mid 1 \leq i \leq m\} \leq D?$$

tj. lze přiřadit úlohy z množiny U na m procesorů tak, aby spočítání úloh na všech procesorech skončilo nejpozději v čase D?

Při rozvrhování ve skutečnosti minimalizujeme D při pevném počtu procesorů. Pokud minimalizujeme m při pevném D, říká se této úloze Bin Packing. Rozhodovací verze obou úloh jsou ale stejné a jde o popsaný problém Rozvrhování.

7. S pomocí některého z problémů Kachlíkování, Splnitelnost, 3-Splnitelnost, Vrcholové pokrytí, Trojrozměrné párování, Hamiltonovská kružnice, Obchodní cestující nebo Loupežníci ukažte, že následující problém je *NP*-úplný:

Problém 11.4.11: Kostra s omezeným stupněm

Instance: Graf $G = (V, E)$ a přirozené číslo $k \geq 0$.

Otázka: Obsahuje graf G kostru, v níž všechny vrcholy jsou stupně nejvýš k?

8. S pomocí některého z problémů Kachlíkování, Splnitelnost, 3-Splnitelnost, Vrcholové pokrytí, Trojrozměrné párování, Hamiltonovská kružnice, Obchodní cestující nebo Loupežníci ukažte, že následující problém je *NP*-úplný:

Problém 11.4.12: Přesné pokrytí 3-prvkovými množinami (X3C)

Instance: Množina X s $|X| = 3q$ a množina trojic $C \subseteq X^3$

Otázka: Existuje $C' \subseteq C$ taková, že každý prvek z X se vyskytuje v právě jedné trojici z C'?

9. S pomocí některého z problémů Kachlíkování, Splnitelnost, 3-Splnitelnost, Vrcholové pokrytí, Trojrozměrné párování, Hamiltonovská kružnice, Obchodní cestující nebo Loupežníci ukažte, že následující problém je *NP*-úplný:

Problém 11.4.13: Dominující množina

Instance: Graf $G = (V, E)$, kladné číslo $k > 0$

Otázka: Existuje množina vrcholů $V' \subseteq V$, $|V'| \leq k$ taková, že každý vrchol $v \in V \setminus V'$ je spojen hranou s alespoň jedním vrcholem z V'?

10. S pomocí některého z problémů Kachlíkování, Splnitelnost, 3-Splnitelnost, Vrcholové pokrytí, Trojrozměrné párování, Hamiltonovská kružnice, Obchodní cestující nebo Loupežníci ukažte, že následující problém je *NP*-úplný:

Problém 11.4.14: Hitting set

Instance: Množina S a množina C podmnožin množiny S, přirozené číslo $k > 0$.

Otázka: Existuje $S' \subseteq S$, $|S'| \leq k$ taková, že S' obsahuje nejméně jeden prvek z každé množiny v C?

11. S pomocí některého z problémů Kachlíkování, Splnitelnost, 3-Splnitelnost, Vrcholové pokrytí, Trojrozměrné párování, Hamiltonovská kružnice, Obchodní cestující nebo Loupežníci ukažte, že následující problém je *NP*-úplný:

Problém 11.4.15: Pokrytí orientovaných cyklů (Feedback vertex set)

Instance: Orientovaný graf $G = (V, E)$ a přirozené číslo $k \geq 0$.

Otázka: Existuje množina $S \subseteq V$, $|S| \leq k$ taková, že z každého orientovaného cyklu v G obsahuje S alespoň jeden vrchol?

12. S pomocí některého z problémů Kachlíkování, Splnitelnost, 3-Splnitelnost, Vrcholové pokrytí, Trojrozměrné párování, Hamiltonovská kružnice, Obchodní cestující nebo Loupežníci ukažte, že následující problém je *NP*-úplný:

Problém 11.4.16: Čtyřlistá kostra grafu

Instance: Neorientovaný graf $G = (V,E)$.

Otázka: Existuje kostra grafu G, která má právě čtyři listy?

13. S pomocí některého z problémů Kachlíkování, Splnitelnost, 3-Splnitelnost, Vrcholové pokrytí, Trojrozměrné párování, Hamiltonovská kružnice, Obchodní cestující nebo Loupežníci ukažte, že následující problém je *NP*-úplný:

Problém 11.4.17: Největší společný podgraf

Instance: Grafy $G_1 = (V_1, E_1)$ a $G_2 = (V_2, E_2)$, přirozené číslo $k \geq 0$.

Otázka: Existují množiny hran $E_1' \subseteq E_1$ a $E_2' \subseteq E_2$, $|E_1'| = |E_2'| \geq k$, pro něž jsou grafy $G_1' = (V_1, E_1')$ a $G_2' = (V_2, E_2')$ izomorfní (tj. stejné až na přejmenování vrcholů)?

14. S pomocí některého z problémů Kachlíkování, Splnitelnost, 3-Splnitelnost, Vrcholové pokrytí, Trojrozměrné párování, Hamiltonovská kružnice, Obchodní cestující nebo Loupežníci ukažte, že následující problém je *NP*-úplný:

Problém 11.4.18: Set Packing

Instance: Množina C konečných množin a přirozené číslo $k \geq 0$

Otázka: Obsahuje C alespoň k po dvou disjunktních množin?

15. S pomocí některého z problémů Kachlíkování, Splnitelnost, 3-Splnitelnost, Vrcholové pokrytí, Trojrozměrné párování, Hamiltonovská kružnice, Obchodní cestující nebo Loupežníci ukažte, že následující problém je NP-úplný:

Problém 11.4.19: Množinové pokrytí (Set Cover)

Instance: Systém množin $\mathcal{C}$ konečné množiny prvků S, tj. $\mathcal{C} \subseteq \wp(S)$, přirozené číslo k.

Otázka: Je možné vybrat $\mathcal{C}' \subseteq \mathcal{C}$, která obsahuje nejvýš k množin a přitom

$$\bigcup_{C \in \mathcal{C}'} C = S \, ?$$

Ostatní

16. Představte si, že máte černou skříňku, která dostane na vstupu formuli φ a rozhodne, zda je tato formule splnitelná, tedy odpoví ANO nebo NE. Popište algoritmus, který může volat tuto černou skříňku a pro danou formuli ψ nalezne splňující ohodnocení, je-li ψ splnitelná. Algoritmus by měl pracovat v polynomiálním čase, pokud volání černé skříňky budeme počítat jako konstantní operaci.

17. Předpokládejte, že máte černou skříňku, která umí zodpovědět, zda v daném grafu existuje hamiltonovská kružnice. Popište algoritmus, který najde v daném grafu hamiltonovskou kružnici, pokud v něm existuje, tj. vypíše seznam jejích vrcholů v pořadí na této kružnici. Algoritmus bude polynomiální, pokud bychom brali volání zmíněné černé skříňky jako konstantní.

18. Popište algoritmus, který v polynomiálním čase vyřeší úlohu HORN-SAT.

Problém 11.4.20: Splnitelnost hornovských formulí (HORN-SAT)

Instance: Formule φ v KNF, kde každá klauzule obsahuje nejvýš jeden pozitivní literál.

Cíl: Hledáme ohodnocení proměnných v, pro něž je $\varphi(v) = 1$, nebo očekáváme odpověď, že takové ohodnocení neexistuje.

19. Popište algoritmus, který v polynomiálním čase vyřeší úlohu 2SAT.

Problém 11.4.21: Splnitelnost kvadratických formulí (2SAT)

Instance: Formule φ v KNF, kde každá klauzule obsahuje nejvýš dva literály.

Cíl: Hledáme ohodnocení proměnných v, pro něž je $\varphi(v) = 1$, nebo očekáváme odpověď, že takové ohodnocení neexistuje.

12. Pseudopolynomiální algoritmy a silná NP-úplnost

I mezi různými NP-úplnými problémy a úlohami může být co do složitosti jejich řešitelnosti rozdíl. V této kapitole se budeme věnovat problémům, které je možné řešit pseudopolynomiálními algoritmy.

12.1. Pseudopolynomiální algoritmus pro batoh

Úloha Batohu je definovaná následovně.

Problém 12.1.1: Batoh (Knapsack)

Instance: Množina n předmětů $A = \{a_1, \ldots, a_n\}$, s každým předmětem asociovaná velikost $s(a_i) \in \mathbb{N}$ a cena či hodnota $v(a_i) \in \mathbb{N}$. Přirozené číslo $B \geq 0$ udávající velikost batohu.

Cíl: Nalézt množinu předmětů $A' \subseteq A$, která dosahuje maximální souhrnné hodnoty předmětů v ní obsažených, a přitom se předměty z A' vejdou do batohu velikosti B. Chceme tedy, aby platilo:

$$\begin{aligned} \sum_{a \in A'} s(a) &\leq B \\ \sum_{a \in A'} v(a) &= \max_{\substack{A'' \subseteq A \\ \sum_{a \in A''} s(a) \leq B}} \sum_{a \in A''} v(a) \end{aligned}$$

Pro tuto úlohu můžeme zkonstruovat algoritmus, který není sice obecně polynomiální, ale pro jistým způsobem omezené instance se polynomiálním může stát. Algoritmus 12.1.1 je založen na dynamickém programování. Na vstupu tento algoritmus očekává jen předměty, které se do batohu vejdou, tj. jen předměty s velikosti nejvýš B, což je celkem přirozené, protože nemá smysl uvažovat předměty, které se do batohu nevejdou ani samy o sobě. Navíc splnění této podmínky lze snadno ověřit. Podobně o všech předmětech můžeme předpokládat, že jejich velikost je nenulová, neboť předměty s nulovou velikostí můžeme přidat do batohu vždy a není nutné je tedy v algoritmu uvažovat.

Ukážeme si, že tento algoritmus najde při vhodné implementaci řešení úlohy Batoh v čase $O(nV)$. Zdá se, tedy, že jde o polynomiální algoritmus, ale musíme si uvědomit,

Algoritmus 12.1.1 Batoh(s, v, B)

Vstup: Pole s velikostí předmětů a pole v cen předmětů, obě délky n, velikost batohu B. Předpokládáme, že $(\forall i \in \{1, \ldots, n\})\ \big[0 < s[i] \leq B\big]$.

Výstup: Množina M předmětů, jejichž souhrnná velikost nepřesahuje B a jejichž souhrnná cena je maximální.

1: $V \leftarrow \sum_{i=1}^{n} v[i]$
2: Nechť T je tabulka typu $(n+1) \times (V+1)$, na pozici $T[j,c]$, $0 \leq j \leq n$, $0 \leq c \leq V$, bude podmnožina indexů $\{1, \ldots, j\}$ prvků celkové ceny přesně c s minimální velikostí.
3: Nechť S je tabulka typu $(n+1) \times (V+1)$, na pozici $S[j,c]$ je celková velikost předmětů v množině $T[j,c]$. Pokud neexistuje množina s předměty ceny přesně c, je na pozici $S[j,c]$ číslo $B+1$.
4: **for** $c \leftarrow 0$ **to** V **do**
5: $\quad T[0,c] \leftarrow \varnothing$
6: $\quad S[0,c] \leftarrow B+1$
7: **end for**
8: $S[0,0] \leftarrow 0$
9: **for** $j \leftarrow 1$ **to** n **do**
10: $\quad T[j,0] \leftarrow \varnothing$
11: $\quad S[j,0] \leftarrow 0$
12: $\quad$ **for** $c \leftarrow 1$ **to** V **do**
13: $\quad\quad T[j,c] \leftarrow T[j-1,c]$
14: $\quad\quad S[j,c] \leftarrow S[j-1,c]$
15: $\quad\quad$ **if** $v[j] \leq c$ **and** $S[j,c] > S[j-1, c-v[j]] + s[j]$ **then**
16: $\quad\quad\quad T[j,c] \leftarrow T[j-1, c-v[j]] \cup \{j\}$
17: $\quad\quad\quad S[j,c] \leftarrow S[j-1, c-v[j]] + s[j]$
18: $\quad\quad$ **end if**
19: $\quad$ **end for**
20: **end for**
21: $c \leftarrow \max\{c' \mid S[n,c'] \leq B\}$
22: **return** $T[n,c]$

že velikost vstupu je pouze $O(n \log_2(B+V))$ a V tedy může být exponenciálně větší než vstup.

Poznámka 12.1.2 *Náš odhad složitosti, tedy $O(nV)$ vychází z předpokladu, že aritmetické operace vyžadují jen konstantní čas. To pochopitelně není úplně korektní předpoklad, neboť ve chvíli, kdy nemáme žádný odhad na velikost čísel na vstupu, nemůžeme předpokládat, že například sčítání bude konstantní. Na druhou stranu všechny aritmetické operace vyžadují jen čas polynomiální v délce binární reprezentace čísel, tedy v $\log_2(B+V)$, což není faktor, který by něco změnil na polynomialitě algoritmu 12.1.1. Dovolíme si tedy zjednodušení, které je ve složitosti obvyklé, a budeme počítat složitost aritmetických operací jako konstantní.*

Věta 12.1.3 *Algoritmus 12.1.1 nalezne pro zadaný vstup množinu předmětů s nejvyšší cenou, jež se vejdou do batohu velikosti B. Algoritmus 12.1.1 pracuje v čase $O(nV)$.*

Důkaz: Fakt, že algoritmus pracuje v čase $O(nV)$ je v podstatě zřejmý. Kroky 1 až 8 zvládneme jistě v čase $O(nV)$, uvažujeme-li aritmetické operace jako konstantní. Následují dva vnořené cykly v rámci nichž jsou použity aritmetické operace, na něž stačí konstantní čas. I řádek 16 lze provést v konstantním čase při vhodné reprezentaci množin v $T[j,c]$. Můžeme například použít reprezentaci pomocí bitového pole délky n, nebo pomocí spojového seznamu, v obou případech je přidání prvku do množiny jednoduché.

Zbývá ukázat, že algoritmus pracuje korektně. Ukážeme, že na konci běhu algoritmu splňují tabulky T a S vlastnosti, jež jsme popsali na řádcích 2 a 3. Jmenovitě ukážeme, že:

(i) Na pozici $T[j,c]$, $0 \le j \le n$, $0 \le c \le V$ je po ukončení algoritmu podmnožina indexů $\{1,\ldots,j\}$ prvků celkové ceny přesně c s minimální velikostí mezi všemi takovými množinami. Pokud neexistuje množina prvků s cenou přesně c, pak na pozici $T[j,c]$ bude prázdná množina.

(ii) Na pozici $S[j,c]$, $0 \le j \le n$, $0 \le c \le V$ je po ukončení algoritmu součet velikostí prvků v množině na pozici $T[j,c]$. Pokud neexistuje množina prvků s cenou přesně c, pak na pozici $S[j,c]$ je $B+1$.

Tyto vlastnosti ukážeme indukcí podle j. Na začátku vyplníme nultý řádek v cyklu na řádcích 4 až 7, pro $j=0$ tedy vlastnosti (i) i (ii) platí. Nyní předpokládejme, že vlastnosti (i) a (ii) platí pro řádky $0,\ldots,j-1$ a ukažme, že platí i pro j-tý řádek. Na pozici $T[j,0]$ je vždy prázdná množina, neboť to je jistě nejmenší množina prvků s cenou 0, prvky prázdné množiny totiž mají přirozeně nulovou velikost. Takto provedeme nastavení $T[j,0]$ a $S[j,0]$ na řádcích 10 a 11. Pokud je $c>0$, pak na ně narazíme na vhodném místě v rámci vnitřního cyklu. Pro $j>0$ a $c>0$ máme dvě možnosti, buď v nejmenší množině prvků $\{1,\ldots,j\}$ s cenou c není prvek j, potom se jedná o množinu uloženou podle indukční hypotézy na pozici $T[j-1,c]$, nebo se v této množině prvek j vyskytuje. Pokud bychom z této nejmenší množiny prvek j odstranili, museli bychom dostat nejmenší množinu s cenou $c-v[j]$ z prvků $\{1,\ldots,j-1\}$, která je podle indukčního předpokladu uložena na pozici $T[j-1,c-v[j]]$. Z těchto dvou možností vybereme tu, která má menší velikost.

Díky tomu také na pozici $T[j,c]$ uložíme prázdnou množinu a na pozici $S[j,c]$ hodnotu $B+1$ jedině tehdy, pokud obě množiny na pozicích $T[j-1,c]$ a $T[j-1,c-v[j]]$ jsou prázdné a obě odpovídající velikosti jsou $B+1$. Zde využíváme toho, že porovnávání v kroku 15 je ostré, a tedy k přiřazení v kroku 13 dojde jedině tehdy, když je množina na pozici $T[j-1,c-v[j]]$ neprázdná, jinak by z prázdné množiny přidáním prvku j vznikla množina neprázdná.

Na závěr tedy při platnosti (i) a (ii) stačí v kroku 21 vybrat množinu s největší cenou. Množina cen, z nichž vybíráme, je vždy neprázdná, neboť přinejmenším $S[n,0]=0$. □

Poznamenejme ještě, že ve skutečnosti není potřeba pamatovat si celé matice T a S, stačí úplně jedno pole T' a S' s aktuálním řádkem matic T a S, přičemž ale musíme toto pole procházet v klesajícím pořadí podle cen. To proto, že pro určení množiny na pozici $T[j,c]$ potřebujeme hodnoty pouze z předchozího kroku, tedy pro $j-1$ a s cenou nejvýše rovnou c, pokud tedy pole procházíme podle klesajících cen, je vždy na pozici $T'[c']$ pro $c' \leq c$ množina z $T[j-1,c']$, totéž lze říci o velikostech v poli S'. Tento postup vede k výrazné úspoře paměti, nemění však nic na časové složitosti, o kterou se v tuto chvíli zajímáme především.

Protože velikost vstupu je $O(n\log_2(B+V))$ při standardním binárním kódování, může být ve skutečnosti čas $O(nV)$ exponenciální ve velikosti vstupu, a proto nejde o polynomiální algoritmus. Nicméně, pokud by existoval polynom p, pro který by platilo, že $(B+V) \leq p(n)$, pak by Algoritmus 12.1.1 byl polynomiální. Tuto podmínku samozřejmě nelze obecně zaručit, ale znamená to, že pro rozumně malá čísla na vstupu poběží algoritmus rozumně rychle. Druhý pohled na věc je, že jde o algoritmus polynomiální, pokud předáme vstup zakódovaný unárně, protože při unárně zadaném vstupu je jeho velikost $O(n(B+V))$. Algoritmům tohoto typu budeme říkat pseudopolynomiální.

12.2. Číselné problémy a pseudopolynomiální algoritmy

Upřesněme si tyto pojmy následující definicí.

Definice 12.2.1 (Číselný problém) Nechť A je libovolný rozhodovací problém a I nechť je instance tohoto problému. Pomocí $\text{len}(I)$ označíme délku zakódování instance I při standardním binárním kódování. Pomocí $\max(I)$ označíme hodnotu největšího číselného parametru v instanci I. Řekneme, že A je *číselný problém*, pokud pro každý polynom p existuje instance I problému A, pro kterou platí, že $\max(I) > p(\text{len}(I))$. ◂

V případě batohu je tedy $\text{len}(I) = O(n\log_2(B+V))$, zatímco $\max(I) = \max(\{B\} \cup \{v[i], s[i] \mid 1 \leq i \leq n\})$. Například Loupežníci nebo Batoh jsou tedy číselné problémy, zatímco *SAT* nebo *KLIKA* číselné problémy nejsou.

Definice 12.2.2 (Pseudopolynomiální algoritmus) Řekneme, že algoritmus řešící problém A je *pseudopolynomiální*, pokud je jeho časová složitost omezená polynomem dvou proměnných $\max(I)$ a $\text{len}(I)$. ◂

Algoritmus 12.1.1 je tedy zřejmě pseudopolynomiální. Je-li problém A řešitelný pseudopolynomiálním algoritmem a není-li A číselný problém, pak je zřejmě tento pseudo-

polynomiální algoritmus ve skutečnosti polynomiální. Pokud bychom tedy pro nějaký nečíselný NP-úplný problém, například kliku nebo splnitelnost, našli pseudopolynomiální algoritmus, znamenalo by to, že P = NP.

12.3. Silná NP-úplnost

Definice 12.3.1 (Silná NP-úplnost) Nechť p je polynom a A NP-úplný problém. Pomocí $A(p)$ označíme restrikci problému A na instance I s $\max(I) \leq p(\text{len}(I))$, tedy instance I je instancí $A(p)$, pokud $\max(I) \leq p(\text{len}(I))$. Řekneme, že problém A je *silně* NP-*úplný*, pokud existuje polynom p, pro nějž je problém $A(p)$ NP-úplný. ◂

Uvedená definice by pochopitelně ztratila smysl, kdyby platilo P = NP, protože potom by byly všechny netriviální problémy z P = NP silně NP-úplné. I pokud to nezmiňujeme, předpokládáme však obvykle opak, tedy, že platí P ≠ NP. Z našich předchozích úvah plyne, že každý nečíselný NP-úplný problém je automaticky silně NP-úplný. Pokud bychom pro nějaký silně NP-úplný problém našli pseudopolynomiální algoritmus, znamenalo by to, že P = NP. To proto, že na $A(p)$ by byl tento algoritmus polynomiální. Z toho plyne, že například problém Batoh není (pokud P ≠ NP) silně NP-úplný a totéž lze říci například o problému Loupežníci, pro který lze použít jen mírnou modifikaci algoritmu 12.1.1. Je také třeba zmínit, že pokud P ≠ NP, pak existují i NP-úplné problémy, jež nejsou ani silně NP-úplné ani řešitelné pseudopolynomiálním algoritmem. To je analogické tomu, že pokud NP ≠ P, existují v NP problémy, jež nejsou v P ani nejsou NP-úplné. Jak již bylo zmíněno, rozdíl mezi P a algoritmy řešitelnými pseudopolynomiálním algoritmem tkví ve způsobu kódování — binárního v případě P a unárního v případě pseudopolynomiálního algoritmu. Podobně tedy silně NP-úplné problémy jsou ty NP-úplné problémy, jež zůstanou NP-úplnými i v případě unárního kódování čísel na vstupu.

Jako příklad číselného problému, který je NP-úplný nám může posloužit již vícekrát zmiňovaný problém obchodního cestujícího.

Problém 12.3.2: Obchodní cestující (*OC*, Traveling Salesperson)

Instance: Množina měst $C = \{c_1, \ldots, c_n\}$, vzdálenost $d(c_i, c_j) \in \mathbb{N}$ pro každá dvě města $c_i, c_j \in C$, přirozené číslo D.

Otázka: Existuje cyklus, který navštíví každé město právě jednou a jehož délka nepřesahuje D? Jinými slovy, existuje permutace $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ taková, že

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}) \leq B?$$

Věta 12.3.3 *Problém Obchodního cestujícího je silně* NP-*úplný.*

Důkaz: Problém *OC* jistě patří do třídy NP, neboť jsme schopni ověřit v polynomiálním čase, zda daná permutace měst splňuje naše požadavky, a zřejmě i $OC(p)$ patří do NP pro každý polynom p. Těžkost problému $OC(p)$ pro vhodně zvolený polynom p si ukážeme převodem z problému Hamiltonovské kružnice v neorientovaném grafu. Uvažme neorientovaný graf $G = (V, E)$ a sestavme na jeho základě instanci obchodního cestujícího následujícím způsobem. Množinu měst C položíme rovnu množině vrcholů V, předpokládejme tedy, že $C = V = \{v_1, \ldots, v_n\}$. Vzdálenost mezi městy či vrcholy v_i a v_j pro $i \neq j$ a $i, j \in \{1, \ldots, n\}$ určíme následujícím předpisem.

$$d(v_i, v_j) = \begin{cases} 0 & \{v_i, v_j\} \in E \\ 1 & \text{jinak} \end{cases}$$

Hodnotu D položíme rovnu 0. Není těžké ukázat, že v grafu G existuje hamiltonovská kružnice, právě když v sestrojené instanci obchodního cestujícího existuje cyklus nulové délky, který obejde všechna města, přičemž navštíví každé z nich právě jednou. Z jedné strany hamiltonovská kružnice využívá jen hrany nulové délky, z druhé strany cesta nulové délky využívá jen hrany grafu G. Hodnota maximálního číselného parametru v sestrojené instanci je přitom rovna 1, tedy omezená verze problému $OC(1)$ je stále NP-úplná, což znamená, že problém *OC* je silně NP-úplný. □

Dalšími silně NP-úplnými číselnými problémy jsou například Rozvrhování či Bin Packing. Najdeme-li k nějakému problému pseudopolynomiální algoritmus, má to hned dva kladné důsledky, jednak lze takový algoritmus použít na instance, v nichž se nevyskytují příliš velká čísla. Na instancích, kde už by i pseudopolynomiální algoritmus počítal příliš dlouho, je často alespoň možné nalézt dobrou aproximaci optimálního řešení, jak si záhy ukážeme v případě Batohu.

12.4. Cvičení

1. Ukažte, že následující problém je polynomiálně řešitelný:

Problém 12.4.1: Omezený součet podmnožiny

Instance: Množina n prvků A, s každým prvkem $a \in A$ asociovaná velikost $v(a) \in \{0, \ldots, n\}$, číslo $B \geq 0$.

Otázka: Existuje množina prvků $A' \subseteq A$, pro níž platí, že

$$\sum_{a \in A'} v(a) = B?$$

13. Aproximační algoritmy a schémata

13.1. Aproximační algoritmy

Koncept aproximačních algoritmů je velmi užitečný pro řešení NP-úplných optimalizačních úloh. Pokud nejsme schopni rychle získat optimální řešení úlohy, můžeme slevit ze svých požadavků a pokusit se najít řešení, jež není od toho optimálního příliš vzdáleno. Nejprve si upřesníme pojem optimalizační úlohy.

Definice 13.1.1 *Optimalizační úloha A* je buď *maximalizační* nebo *minimalizační* a skládá se z těchto tří částí:

1. Množina instancí $D_A \subseteq \{0,1\}^*$.
2. Pro každou instanci $I \in D_A$ je dána množina přípustných řešení $S_A(I) \subseteq \{0,1\}^*$.
3. Funkce μ_A, která každé instanci $I \in D_A$ a každému přípustnému řešení $\sigma \in S_A(I)$ přiřadí kladné racionální číslo $\mu_A(I,\sigma)$, které nazveme *hodnotou řešení* σ.

Pokud je A maximalizační úlohou, pak *optimálním řešením* instance $I \in D_A$ je $\sigma \in S_A(I)$, pro něž je $\mu_A(I,\sigma)$ maximální (tj. $\mu_A(I,\sigma) = \max\{\mu_A(I,\sigma) \mid \sigma \in S_A(I)\}$). Pokud je A minimalizační úlohou, pak optimálním řešením instance $I \in D_A$ je $\sigma \in S_A(I)$, pro něž je $\mu_A(I,\sigma)$ minimální (tj. $\mu_A(I,\sigma) = \min\{\mu_A(I,\sigma) \mid \sigma \in S_A(I)\}$). Hodnotu optimálního řešení instance $I \in D_A$ budeme označovat pomocí $OPT_A(I)$, přičemž index A budeme často vynechávat, bude-li jasné, o jakou úlohu se jedná. Tedy je-li σ^* optimální řešení instance I, pak $OPT_A(I) = \mu_A(I,\sigma^*)$. ◂

Uvažme například minimalizační úlohu Vrcholového pokrytí, v tomto případě je množina instancí D_{VP} tvořena řetězci kódujícími graf. Pro danou instanci, tedy graf $G = (V,E)$, obsahuje množina přípustných řešení $S_{VP}(G)$ všechny množiny vrcholů $S \subseteq V$, které pokrývají všechny hrany. Mírou daného přípustného řešení S je pak počet jejích prvků, tedy $\mu_{VP}(G,S) = |S|$. Jde o minimalizační úlohu, neboť se snažíme najít co nejmenší množinu, která pokrývá všechny hrany. Všimněme si, že najít nějakou množinu S, která pokrývá všechny vrcholy, je snadné, stačí vzít třeba množinu všech vrcholů $S = V$, co činí úlohu obtížnou, je právě to, že chceme minimalizovat její velikost. To je obvyklé u řady úloh, pro něž hledáme aproximační algoritmy, protože pokud je těžké najít vůbec nějaké přípustné řešení pro danou instanci, pak je o to těžší najít optimální řešení.

Nyní můžeme definovat pojem aproximačního algoritmu.

Definice 13.1.2 Řekneme, že algoritmus ALG je *aproximačním algoritmem* pro optimalizační úlohu A, pokud pro každou instanci $I \in D_A$ vrátí ALG se vstupem I řešení $\sigma \in S_A(I)$, případně ohlásí, že žádné přípustné řešení neexistuje, pokud $S_A(I) = \varnothing$. Hodnotu řešení

vráceného algoritmem ALG na instanci I označíme jako $ALG(I)$, tj. $ALG(I) = \mu_A(I, \sigma)$, kde $\sigma \in S_A(I)$ je přípustné řešení vrácené algoritmem ALG. *Aproximační poměr* algoritmu ALG definujeme takto: Pokud je A maximalizační úloha, pak racionální číslo $\varepsilon \geq 1$ nazveme *aproximačním poměrem* algoritmu ALG, pokud pro každou instanci $I \in D_A$ platí, že

$$OPT(I) \leq \varepsilon \cdot ALG(I).$$

Je-li A minimalizační úlohou, pak racionální číslo $\varepsilon \geq 1$ nazveme aproximačním poměrem algoritmu ALG, pokud pro každou instanci $I \in D_A$ platí, že

$$ALG(I) \leq \varepsilon \cdot OPT(I).$$ ◂

Naše definice aproximačního poměru umožňuje přistupovat jednotně k minimalizačním i maximalizačním úlohám, často se též definuje aproximační poměr pro maximalizační úlohu jako obrácená hodnota námi definovaného aproximačního poměru, na tom, který způsob jsme si zvolili, však příliš nezáleží.

13.2. Příklad aproximačního algoritmu pro Bin Packing

Začneme jednoduchým aproximačním algoritmem pro úlohu Bin Packing.

Problém 13.2.1: Bin Packing (BP)

Instance: Konečná množina předmětů $U = \{u_1, .., u_n\}$, s každým předmětem asociovaná velikost $s(u)$, což je racionální číslo, pro které platí $0 \leq s(u) \leq 1$.

Cíl: Najít rozdělení všech předmětů do co nejmenšího počtu po dvou disjunktních množin $U_1, \ldots, U_m$ takové, že

$$(\forall i \in \{1, \ldots, m\}) \Big[\sum_{u \in U_i} s(u) \leq 1 \Big].$$

Naším cílem je minimalizovat m.

Formálně je tedy D_{BP} množinou řetězců kódujících instance BP, pro danou instanci $I = \langle U, s(u) \rangle$ je množina $S_{BP}(I)$ množinou všech možných rozdělení do dostatečného množství košů. Mírou řešení $\mu_{BP}(\sigma)$ pro $\sigma \in S_{BP}(I)$ je počet košů, které řešení využívá, tedy hodnota m. Rozhodovací verze tohoto problému je shodná s problémem Rozvrhování, jehož těžkost jsme si ukazovali v rámci cvičení, z toho plyne, že i úloha BP je NP-těžká. Šance na to, že bychom našli polynomiální algoritmus, řešící BP přesně, jsou tedy malé.

Uvažme jednoduchý hladový algoritmus, kterým bychom řešili tuto úlohu: *Ber jeden předmět po druhém, pro každý předmět u najdi první koš, do nějž se tento předmět ještě vejde, pokud takový koš neexistuje, přidej nový koš obsahující jen předmět u.* Tomuto algoritmu budeme říkat „First Fit“ (FF). Algoritmus FF je zřejmě polynomiální. Ukážeme si, že pro

každou instanci $I \in D_{BP}$ platí, že $FF(I) \leq 2 \cdot OPT(I)$, jinými slovy, že aproximační poměr algoritmu *FF* je 2.

Věta 13.2.2 *Pro každou instanci $I \in D_{BP}$ platí, že $FF(I) < 2 \cdot OPT(I)$.*

Důkaz: V řešení, které vrátí *FF* je nejvýš jeden koš, který je zaplněn nejvýš z poloviny. Kdyby totiž existovaly dva koše U_i, U_j pro $i < j$, které jsou zaplněny nejvýš z poloviny, tak by *FF* nepotřeboval zakládat nový koš pro předměty z U_j, všechny by se vešly do U_i. Pokud $FF(I) > 1$, pak z toho plyne, že

$$FF(I) < \left\lceil 2 \sum_{i=1}^{n} s(u_i) \right\rceil \leq 2 \left\lceil \sum_{i=1}^{n} s(u_i) \right\rceil,$$

kde první nerovnost plyne z toho, že po zdvojnásobení obsahu jsou všechny koše plné až na jeden, který může být zaplněn jen částečně. Rovnosti bychom přitom dosáhli jedině ve chvíli, kdy by byly všechny koše zaplněné právě z poloviny, což není podle našeho předpokladu možné. Druhá nerovnost plyne z vlastností zaokrouhlování.

Na druhou stranu musí platit, že

$$OPT(I) \geq \left\lceil \sum_{i=1}^{n} s(u_i) \right\rceil.$$

Dohromady tedy dostaneme, že $FF(I) < 2 \cdot OPT(I)$. Pokud $FF(I) = 1$, pak i $OPT(I) = 1$ a i v tomto případě platí ostrá nerovnost. □

Lze dokonce ukázat o něco lepší odhad $FF(I) \leq \frac{17}{10} OPT(I) + 2$. Na druhou stranu však existují instance I s libovolně velkou hodnotou $OPT(I)$, pro něž $FF(I) \geq \frac{17}{10}(OPT(I) - 1)$. My si ukážeme příklad instancí, pro něž je $FF(I) \geq \frac{5}{3} OPT(I)$, což není od 2 příliš vzdáleno.

Lemma 13.2.3 *Pro libovolnou hodnotu m existuje instance $I \in D_{BP}$, pro niž je $OPT(I) \geq m$ a $FF(I) \geq \frac{5}{3} OPT(I)$.*

Důkaz: Instance bude mít $U = \{u_1, u_2, \ldots, u_{18m}\}$, s těmito prvky asociujeme váhy takto:

$$s(u_i) = \begin{cases} \frac{1}{7} + \varepsilon & 1 \leq i \leq 6m, \\ \frac{1}{3} + \varepsilon & 6m < i \leq 12m, \\ \frac{1}{2} + \varepsilon & 12m < i \leq 18m, \end{cases}$$

kde $\varepsilon > 0$ je dostatečně malé kladné racionální číslo. Optimální rozdělení rozdělí prvky do $6m$ košů, do každého dá po jednom prvku s velikostmi $\frac{1}{7} + \varepsilon$, $\frac{1}{3} + \varepsilon$, $\frac{1}{2} + \varepsilon$. Hodnotu ε zvolíme dostatečně malou tak, aby se každá z těchto trojic vešla do koše velikosti 1. Algoritmus *FF* bude brát prvky jeden po druhém a vytvoří nejprve m košů, každý s šesti prvky velikosti $\frac{1}{7} + \varepsilon$, přičemž hodnotu ε zvolíme dostatečně malou na to, aby se tam

vešly, ale protože je kladná, nevejde se k nim nic dalšího. Poté vytvoří $3m$ košů, každý se dvěma prvky velikosti $\frac{1}{3}+\varepsilon$, k nimž se opět nic nevejde. Nakonec *FF* vytvoří $6m$ košů, každý s jedním prvkem velikosti $\frac{1}{2}+\varepsilon$. Dohromady je $FF(I) = 10m$, a tedy $\frac{FF(I)}{OPT(I)} = \frac{5}{3}$. □

Pochopitelně brát prvky v libovolném pořadí tak, jak to činí *FF* je nejhloupější možnou strategií. Na instanci popsanou v důkazu lemmatu 13.2.3 by přitom stačilo, kdybychom nejprve prvky U setřídili sestupně podle velikosti a poté je umísťovali do košů v pořadí od největšího k nejmenšímu. Algoritmus, který postupuje podle této strategie nazveme *FFD* z anglického „First Fit Decreasing". Lze ukázat, že pro libovolnou instanci $I \in D_{BP}$ platí

$$FFD(I) \leq \frac{11}{9}OPT(I) + 4.$$

My si pouze ukážeme příklad libovolně velké instance, pro níž je

$$FFD(I) \geq \frac{11}{9}OPT(I).$$

To ukazuje, že uvedený odhad nelze příliš zlepšit. Instance bude mít $U = \{u_1, \ldots, u_{30m}\}$,

$$s(u_i) = \begin{cases} \frac{1}{2}+\varepsilon & 1 \leq i \leq 6m, \\ \frac{1}{4}+2\varepsilon & 6m < i \leq 12m, \\ \frac{1}{4}+\varepsilon & 12m < i \leq 18m, \\ \frac{1}{4}-2\varepsilon & 18m < i \leq 30m, \end{cases}$$

kde $\varepsilon > 0$ je opět dostatečně malé racionální číslo. Optimální počet košů pro tuto instanci je $9m$, $6m$ košů, z nichž každý obsahuje $\{\frac{1}{4}-2\varepsilon, \frac{1}{4}+\varepsilon, \frac{1}{2}+\varepsilon\}$ a $3m$ košů, z nichž každý obsahuje $\{\frac{1}{4}-2\varepsilon, \frac{1}{4}-2\varepsilon, \frac{1}{4}+2\varepsilon, \frac{1}{4}+2\varepsilon\}$. Tento počet skutečně nelze zlepšit, protože všechny koše jsou zcela zaplněny. Jak se však zachová *FFD*? Tento algoritmus vytvoří dohromady $11m$ košů. $6m$ košů bude obsahovat $\{\frac{1}{2}+\varepsilon, \frac{1}{4}+2\varepsilon\}$, $2m$ košů bude obsahovat $\{\frac{1}{4}+\varepsilon, \frac{1}{4}+\varepsilon, \frac{1}{4}+\varepsilon\}$ a $3m$ košů bude obsahovat $\{\frac{1}{4}-2\varepsilon, \frac{1}{4}-2\varepsilon, \frac{1}{4}-2\varepsilon, \frac{1}{4}-2\varepsilon\}$.

13.3. Úplně polynomiální aproximační schéma pro Batoh

Vraťme se nyní k úloze Batohu, pro který jsme dříve zkonstruovali pseudopolynomiální algoritmus. Tohoto algoritmu využijeme při konstrukci aproximačního algoritmu, jehož aproximační poměr si můžeme předepsat v rámci vstupu. Vzpomeňme si, že algoritmus 12.1.1 pracuje dle věty 12.1.3 v čase $O(nV)$, kde n je počet předmětů a V je jejich celková cena. Řekli jsme si také, že tento čas by byl polynomiální, kdyby hodnota V byla omezena nějakým polynomem $p(n)$. Hodnota V se do odhadu složitosti algoritmu 12.1.1 dostala jako horní odhad potenciálních cen předmětů v batohu. Pokud provedeme zaokrouhlení cen předmětů a poté jejich přeškálování, můžeme zmenšit součet nových cen

předmětů a tím snížit časové nároky algoritmu 12.1.1. Zaokrouhlením však ztratíme optimalitu výsledku. Čím hrubější zaokrouhlení vstupních cen provedeme, tím rychlejší běh algoritmu dostaneme, ale tím horšího výsledku dosáhneme. Této myšlenky využívá algoritmus 13.3.1, kde parametr ε určuje míru zaokrouhlení vstupních cen, což se projeví jednak v časových nárocích algoritmu, jednak v jeho aproximačním poměru.

Algoritmus 13.3.1 Batoh-apx(s, v, B, ε)

Vstup: Pole s velikostí předmětů a pole v cen předmětů, obě délky n, velikost batohu B, racionální číslo $\varepsilon > 0$. Předpokládáme, že $(\forall i \in \{1, \ldots, n\})\ \left[0 < s[i] \leq B\right]$

Výstup: Množina M předmětů, jejichž souhrnná velikost nepřesahuje B.

1: Spočti index m, pro nějž je $v[m] = \max_{1 \leq i \leq n} v[i]$.
2: **if** $\varepsilon \geq n - 1$ **then**
3: **return** $\{m\}$
4: **end if**
5: $t \leftarrow \lfloor \log_2(\frac{\varepsilon \cdot v[m]}{n}) \rfloor - 1$
6: c je nové pole délky n.
7: **for** $i \leftarrow 1$ **to** n **do**
8: $c[i] \leftarrow \lfloor \frac{v[i]}{2^t} \rfloor$
9: **end for**
10: **return** Batoh(s, c, B) {*Viz algoritmus 12.1.1*}

Hodnota t, která určuje zaokrouhlení, je samozřejmě zvolena tak, aby správně vyšel aproximační poměr a časová složitost algoritmu. Pokud si odmyslíme celé části, můžeme nahlédnout, že

$$\frac{v[i]}{2^t} \sim \frac{v[i]}{\frac{\varepsilon \cdot v[m]}{n} \frac{1}{2}} = \frac{2 \cdot n \cdot v[i]}{\varepsilon \cdot v[m]}.$$

Poměr mezi $v[i]$ a $v[m]$ určuje, kolik procent zabírá $v[i]$ vzhledem k $v[m]$. Základním cílem přepočtu je pochopitelně zmenšit hodnotu $v[m]$, odpovídajícím způsobem se proporcionálně musí zmenšit i $v[i]$. Dalším důležitým faktorem je podíl ε/n. To vychází z toho, že chceme-li celkově dosáhnout chyby ε, můžeme si na jednom prvku dovolit chybu ε/n. Ve výpočtu se uvažuje opačná hodnota podílu, tedy n/ε, protože chybu chceme dosáhnout v přepočtené hodnotě $c[i]$ a ne v $v[i]$.

Ukažme si nyní, jakého aproximačního poměru a jakého času dosáhneme s algoritmem 13.3.1 při zadané hodnotě ε.

Věta 13.3.1 *Algoritmus 13.3.1 pracuje v čase* $O(\frac{1}{\varepsilon} n^3)$. *Pro libovolnou instanci* $I = \langle s, v, B \rangle$ *úlohy Batoh a libovolné* $\varepsilon > 0$ *platí, že*

$$OPT(I) \leq (1 + \varepsilon) Batoh\text{-}apx(I, \varepsilon)$$

Důkaz: Nejprve se zaměříme na aproximační poměr algoritmu 13.3.1. Pokud je $\varepsilon \geq n-1$, pak algoritmus vrátí množinu $\{m\}$, to je jistě přípustné řešení, neboť předpokládáme, že $s[m] \leq B$. Protože zřejmě $OPT(I) \leq n \cdot v[m]$, dostáváme:

$$\frac{OPT(I)}{\text{Batoh-apx}(I, \varepsilon)} = \frac{OPT(I)}{v[m]} \leq \frac{n \cdot v[m]}{v[m]} = n \leq 1 + \varepsilon$$

Nyní předpokládejme, že $\varepsilon < n - 1$. V dalším důkazu použijeme následujícího pozorování: Pro každé reálné číslo x platí

$$x - 1 \leq \lfloor x \rfloor \leq x \,. \tag{13.1}$$

Použijeme-li tento odhad na novou cenu $c[i]$, dostaneme

$$\frac{v[i]}{2^t} - 1 \leq \left\lfloor \frac{v[i]}{2^t} \right\rfloor \leq \frac{v[i]}{2^t} \,.$$

A tedy podle definice $c[i]$ a (13.1)

$$v[i] - 2^t \leq c[i] \cdot 2^t \leq v[i] \tag{13.2}$$

Využijeme-li pozorování (13.1) pro odhad 2^t zdola, obdržíme

$$\frac{1}{4} \frac{\varepsilon \cdot v[m]}{n} = 2^{\log_2(\frac{\varepsilon \cdot v[m]}{n}) - 2} \leq 2^t \tag{13.3}$$

a pro odhad shora dostaneme společně s faktem $\varepsilon < n - 1$, že

$$2^t \leq 2^{\log_2(\frac{\varepsilon \cdot v[m]}{n}) - 1} = \frac{1}{2} \cdot \frac{\varepsilon \cdot v[m]}{n} < \frac{1}{2} \cdot v[m] \,. \tag{13.4}$$

Množina M, kterou algoritmus vrátí v kroku 10, splňuje

$$\text{Batoh-apx}(I, \varepsilon) = \sum_{i \in M} v[i] \geq \sum_{i \in M} c[i] \cdot 2^t,$$

kde nerovnost plyne z 13.2. Nechť M^* je optimální řešení dané instance I, tj. množina prvků s maximální cenou v, jež se vejdou do batohu velikosti B. Protože M je dle věty 12.1.3 optimální řešení pro instanci $I' = \langle s, c, B \rangle$, zatímco M^* je přípustné řešení této upravené instance, můžeme pokračovat:

$$\begin{aligned}
\sum_{i \in M} c[i] \cdot 2^t \geq \sum_{i \in M^*} c[i] \cdot 2^t \geq & \qquad \text{dle (13.2)} \\
\geq \sum_{i \in M^*} (v[i] - 2^t) = & \\
= \sum_{i \in M^*} v[i] - \sum_{i \in M^*} 2^t \geq & \\
\geq OPT(I) - n \cdot 2^t &
\end{aligned}$$

Z toho dostaneme, že

$$\frac{OPT(I)}{\text{Batoh-apx}(I, \varepsilon)} \leq 1 + \frac{n \cdot 2^t}{\text{Batoh-apx}(I, \varepsilon)} \,.$$

Stačí tedy ukázat, že

$$\frac{n \cdot 2^t}{\text{Batoh-apx}(I, \varepsilon)} \leq \varepsilon.$$

Protože M je optimálním řešením I' a $\{m\}$ je přípustným řešením I', dostaneme, že

$$\begin{aligned}\text{Batoh-apx}(I,\varepsilon) &\geq \sum_{i\in M} c[i]\cdot 2^t \geq c[m]\cdot 2^t \geq && \text{(dle 13.2)}\\ &\geq v[m] - 2^t \geq && \text{(dle 13.4)}\\ &\geq v[m] - \frac{1}{2}v[m] = \frac{1}{2}v[m].\end{aligned}$$

Dohromady se 13.4 tedy dostaneme, že

$$\frac{n\cdot 2^t}{\text{Batoh-apx}(I,\varepsilon)} \leq \frac{n\cdot\frac{1}{2}\cdot\frac{\varepsilon\cdot v[m]}{n}}{\frac{1}{2}\cdot v[m]} = \varepsilon\ .$$

Nyní odhadneme časové nároky algoritmu 13.3.1. Vyjma kroku 10 lze zřejmě všechny kroky provést dokonce v čase $O(n)$, tedy v čase lineárním vzhledem k velikosti vstupu. Zde opět předpokládáme, že aritmetické operace lze provést v konstantním čase, což je zjednodušení, které jsme si dovolili v poznámce 12.1.2 bez újmy na obecnosti předpokládat. Z věty 12.1.3 víme, že krok 10 zabere čas $O(nC)$, kde $C = \sum_{i=1}^{n} c[i]$. Stačí tedy odhadnout hodnotu C.

$$C = \sum_{i=1}^{n} c[i] \leq n\cdot c[m] \leq \frac{n\cdot v[m]}{2^t} \leq \frac{n\cdot v[m]}{\frac{1}{4}\cdot\frac{\varepsilon\cdot v[m]}{n}} = \frac{4n^2}{\varepsilon},$$

kde předposlední nerovnost plyne z (13.3). Platí tedy, že $C = O(\frac{1}{\varepsilon}n^2)$. Volání algoritmu 12.1.1 tedy zabere čas $O(\frac{1}{\varepsilon}n^3)$, tím je dán i celkový čas algoritmu 13.3.1. □

13.4. Aproximační schémata

To, co jsme získali algoritmem 13.3.1, je postup, jak pro libovolné $\varepsilon > 0$ dostat aproximační algoritmus pro úlohu Batohu s aproximačním poměrem $1+\varepsilon$, který je polynomiální ve velikosti vstupu a $\frac{1}{\varepsilon}$. Takovému schématu, parametrizovanému hodnotou ε, budeme říkat úplně polynomiální aproximační schéma.

Definice 13.4.1 Nechť A je libovolná optimalizační úloha. Algoritmus ALG nazveme *aproximačním schématem* pro úlohu A, pokud na vstupu očekává instanci $I \in D_A$ a racionální číslo $\varepsilon > 0$ a na výstupu vydá řešení $\sigma \in S_A(I)$, jehož hodnota se od optimální liší s aproximačním poměrem $(1+\varepsilon)$. Pro maximalizační úlohu tedy platí, že

$$OPT(I) \leq (1+\varepsilon)ALG(I,\varepsilon)$$

a pro minimalizační úlohu platí, že

$$ALG(I,\varepsilon) \leq (1+\varepsilon)OPT(I)\ .$$

Předpokládejme, že algoritmus ALG je aproximační schéma a očekává na vstupu instanci $I \in D_A$ a racionální číslo ε. Pomocí ALG_ε označíme instanci algoritmu ALG, kde hodnota ε je zafixována, vstupem ALG_ε je tedy jen instance $I \in D_A$ a běh i výstup ALG_ε na vstupu I jsou totožné s algoritmem ALG se vstupem I a ε.

Řekneme, že ALG je *polynomiální aproximační schéma* (PAS), pokud je pro každé ε časová složitost algoritmu ALG_ε polynomiální v $len(I)$, kde ALG_ε označuje algoritmus vzniklý dosazením konstanty ε do ALG

Řekneme, že ALG je *úplně polynomiální aproximační schéma* ($ÚPAS$), pokud ALG pracuje v čase polynomiálním v $len(I)$ a $\frac{1}{\varepsilon}$. ◂

Rozdíl mezi polynomiálním aproximačním schématem a úplně polynomiálním aproximačním schématem je v tom, že v případě PAS se může ve funkci odhadující složitost algoritmu ALG_ε objevit hodnota $1/\varepsilon$ i v exponentu, což není možné v případě ÚPAS. Například složitost $O(n^{\frac{1}{\varepsilon}})$ by byla naprosto v pořádku pro PAS, ale nikoli pro ÚPAS.

Algoritmus 13.3.1 tedy tvoří úplně polynomiální aproximační schéma pro úlohu Batoh. Dá se říci, že úplně polynomiální aproximační schéma maximum, čeho lze s aproximačními algoritmy dokázat v případě, že nemáme přímo polynomiální algoritmus pro danou úlohu. Navíc technika, kterou jsme použili při konstrukci algoritmu 13.3.1 je obecnější a lze ji použít i v mnoha jiných případech, kdy máme k dispozici pseudopolynomiální algoritmus, před jehož použitím jde jen o to vhodně zaokrouhlit vstupní hodnoty.

Tento postup lze i obrátit, pokud se nám pro úlohu A, která splňuje jistá omezení, podaří popsat ÚPAS, můžeme z něj zpětně zkonstruovat i pseudopolynomiální algoritmus pro tuto úlohu.

Věta 13.4.2 *Nechť A je optimalizační úloha, jejíž přípustná řešení mají nezápornou celočíselnou hodnotu. Předpokládejme, že existuje polynom dvou proměnných q, který pro každou instanci $I \in D_A$ splňuje*

$$OPT(I) < q(\text{len}(I), \max(I)).$$

Pokud existuje úplně polynomiální aproximační schéma pro úlohu A, pak existuje i pseudopolynomiální algoritmus pro A.

Důkaz: Předpokládejme nejprve, že A je maximalizační úloha, případ, kdy A by byla minimalizační úloha, bychom probrali analogicky. Nechť ALG je úplně polynomiální aproximační schéma pro úlohu A, pseudopolynomiální algoritmus ALG' řešící úlohu A postupuje následovně: Pro danou instanci I položíme

$$\varepsilon = q(\text{len}(I), \max(I))^{-1}$$

a pustíme $ALG_\varepsilon(I)$. Čas tohoto kroku je polynomiální v $\text{len}(I)$ a $1/\varepsilon = q(\text{len}(I), \max(I))$, dohromady se tedy jedná o polynomiální algoritmus v $\text{len}(I)$ a $\max(I)$, tedy o pseudopolynomiální algoritmus. Protože ALG je úplně polynomiální aproximační schéma a protože předpokládáme, že A je maximalizační úloha, dostaneme, že

$$OPT(I) \leq (1 + \varepsilon) ALG_\varepsilon(I),$$

tedy

$$OPT(I) - ALG_\varepsilon(I) \leq \varepsilon ALG_\varepsilon(I) \leq \varepsilon OPT(I) < 1,$$

kde druhá nerovnost plyne z toho, že A je maximalizační úloha, a třetí nerovnost plyne z definice ε a toho, že q omezuje velikost $OPT(I)$. Protože hodnoty přípustných řešení úlohy A jsou nezáporná celá čísla, musí ve skutečnosti platit, že $OPT(I) = ALG_\varepsilon(I)$.

Pokud by A byla minimalizační úloha, postupovali bychom stejně, jen na závěr bychom dostali, že

$$ALG_\varepsilon(I) \leq (1 + \varepsilon)OPT(I),$$

a tedy

$$ALG_\varepsilon(I) - OPT(I) \leq \varepsilon OPT(I) < 1. \qquad \square$$

Omezení kladené na úlohu A není z praktického hlediska příliš vážné, protože reálné optimalizační úlohy nebudou mít obvykle s existencí polynomu omezujícího optimální hodnotu řešení problém. Jako okamžitý důsledek věty 13.4.2 dostáváme, že pro silně NP-úplné úlohy, které splňují předpoklady této věty, nemá asi smysl pokoušet se o konstrukci úplně polynomiálního aproximačního schématu.

Důsledek 13.4.3 *Nechť A je silně* NP*-úplná optimalizační úloha, která splňuje předpoklady věty 13.4.2. Pokud* P $\neq$ NP*, pak neexistuje úplně polynomiální aproximační schéma pro úlohu A.*

13.5. Neaproximovatelnost

V některých případech nemůžeme (pokud P $\neq$ NP) najít ani polynomiální aproximační algoritmus s konstantním poměrem. Například pro úlohu Obchodního cestujícího s trojúhelníkovou nerovností existuje $\frac{3}{2}$-aproximační algoritmus (a dosud není znám žádný lepší). Pro úlohu Obchodního cestujícího v euklidovské rovině existuje dokonce polynomiální aproximační schéma. Na druhou stranu obecná úloha Obchodního cestujícího bez omezení nepřipouští existenci aproximačního algoritmu s vůbec nějakým konstantním aproximačním algoritmem, pokud P $\neq$ NP.

Věta 13.5.1 *Pokud existuje polynomiální aproximační algoritmus ALG pro úlohu obchodního cestujícího s aproximačním poměrem ε, kde $\varepsilon > 1$ je konstanta, potom* P = NP*.*

Důkaz: Předpokládejme, že ALG je polynomiální aproximační algoritmus pro úlohu Obchodního cestujícího s konstantním aproximačním poměrem $\varepsilon > 1$. Ukážeme, jak tohoto algoritmu využít k vyřešení NP-úplného problému Hamiltonovské kružnice. Nechť $G = (V, E)$ je libovolný graf, v němž chceme najít hamiltonovskou kružnici. Zkonstruujeme instanci I obchodního cestujícího následujícím způsobem. Množina měst bude V, vzdálenost $d(u, v)$ mezi městy u a v určíme jako

$$d(u,v) = \begin{cases} 1 & \text{pokud } \{u,v\} \in E \\ \varepsilon \cdot |V| & \text{jinak} \end{cases}$$

Konstrukce je zřejmě polynomiální a polynomiální je i běh algoritmu ALG na instanci I. Pokud v G existuje hamiltonovská kružnice, potom $OPT(I) = |V|$, protože k tomu, abychom prošli všechna města si vystačíme s přechody délky 1. Pokud v G neexistuje hamiltonovská kružnice, pak i nejlepší řešení v I musí využít některý přechod délky $\varepsilon \cdot |V|$, a proto je v tomto případě $OPT(I) > \varepsilon \cdot |V|$. Tato nerovnost je ostrá proto, že kružnice má alespoň dvě hrany a všechny mají nenulovou délku. Platí tedy, že $OPT(I) = |V|$ právě když v G existuje hamiltonovská kružnice, přičemž pokud v G hamiltonovská kružnice neexistuje, platí $OPT(I) > \varepsilon|V|$. Vzhledem k tomu, že $OPT(I) \leq ALG(I) \leq \varepsilon \cdot OPT(I)$, znamená to, že $ALG(I) \leq \varepsilon \cdot |V|$, právě když v G existuje hamiltonovská kružnice. Protože ALG je polynomiální algoritmus, který takto řeší NP-úplný problém hamiltonovské kružnice, tak P = NP. □

Pokud se začneme zabývat aproximací, jsou tedy ve složitosti NP-úplných úloh značné rozdíly, ačkoli uvážíme-li jen polynomiální převoditelnost, vypadají všechny tyto úlohy jako stejně těžké. Jsou zde však úlohy, které jsou rychle libovolně dobře aproximovatelné (mají ÚPAS) jako například Batoh, další mají alespoň PAS. Může se stát, že pro úlohu A máme sice aproximační algoritmus s konstantním poměrem, ale ne už s libovolným konstantním poměrem, například Vrcholové pokrytí, nemají tedy ani PAS. Konečně nejtěžší jsou úlohy, pro něž dokonce nemůžeme najít ani aproximační algoritmus s konstantním poměrem, jako je třeba Klika nebo Obchodní cestující, tyto úlohy jsou tedy řešitelné aproximačními algoritmy jen velmi obtížně, pokud vůbec. Na příkladu Obchodního cestujícího navíc vidíme, že úloha se stává tím jednodušší, čím víc víme o jejích instancích (zda splňují trojúhelníkovou nerovnost, zda metrika vzdáleností je euklidovská), což však není nakonec nijak překvapivé. Samozřejmě všechny tyto úvahy a výsledky platí jen za předpokladu, že P ≠ NP.

13.6. Cvičení

1. Definujme optimalizační úlohu Vrcholového pokrytí následovně:

Problém 13.6.1: Vrcholové pokrytí

Instance:	Neorientovaný graf $G = (V,E)$
Cíl:	Nalézt co nejmenší vrcholové pokrytí grafu G. Cílem je tedy nalézt množinu $S \subseteq V$ s co nejmenším počtem vrcholů, pro kterou by platilo, že $(\forall \{u,v\} \in E)\ [\{u,v\} \cap S \neq \varnothing]$.

Uvažme jednoduchý aproximační pro úlohu Vrcholového pokrytí popsaný v algoritmu 13.6.1.

Ukažte, že množina S vrácená tímto algoritmem skutečně tvoří vrcholové pokrytí

Algoritmus 13.6.1 Hladový aproximační algoritmus pro Vrcholové pokrytí

Vstup: Neorientovaný graf $G = (V,E)$
Výstup: Vrcholové pokrytí $S \subseteq V$

1: $S \leftarrow \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: Vyber libovolnou hranu $e = \{u,v\} \in E$.
4: $S \leftarrow S \cup \{u,v\}$
5: Odstraň z E hrany incidentní s u nebo s v.
6: **end while**
7: **return** S

grafu G a že platí $|S| \leq 2OPT(G)$, kde $OPT(G)$ označuje velikost nejmenšího vrcholového pokrytí grafu G.

2. Popište, jak s využitím minimální kostry zkonstruovat 2-aproximační algoritmus pro úlohu obchodního cestujícího, předpokládáme-li, že funkce vzdálenosti splňuje trojúhelníkovou nerovnost.

3. Popište, jak algoritmus z předchozího bodu vylepšit na $\frac{3}{2}$-aproximační algoritmus za pomoci kombinace minimální kostry T a maximálního párování minimální ceny (vzdálenosti) na vrcholech lichého stupně kostry T.

14. Další zajímavé složitostní třídy

14.1. Doplňky jazyků z NP — třída co-NP

Víme již, že pokud P ≠ NP, je splnitelnost, tedy SAT těžký problém, ale jsme alespoň schopni ověřit, zda dané ohodnocení je splňující. Co kdybychom se u dané formule ale zeptali opačně, čili, co kdybychom se neptali, zda daná formule je splnitelná, ale zda daná formule je nesplnitelná?

Problém 14.1.1: Nesplnitelnost KNF (UNSAT)

Instance: Formule φ v KNF
Otázka: Platí, že pro každé ohodnocení v je $\varphi(v) = 0$?

Na první pohled se nezdá, že by tento problém mohl patřit do třídy NP, neboť není jasné, jak by mohl vypadat polynomiálně velký certifikát dosvědčující fakt, že daná formule φ není splněna při žádném ohodnocení jejích proměnných. Jednou z možností, jak by takový certifikát mohl vypadat, je například rezoluční zamítnutí dané formule. Existují však formule, které nemají polynomiálně dlouhé rezoluční zamítnutí. Nejznámějším příkladem takové formule je formule reprezentující princip holubníku (Dirichletův princip, *pigeon hole principle*). K podobné situaci dochází i v jiných důkazových systémech a převládá názor, že obecně neexistuje polynomiálně velký certifikát, který by dosvědčil, že daná formule φ není splnitelná.

Na druhou stranu máme k dispozici polynomiálně velký a ověřitelný certifikát negativní odpovědi na otázku kladenou v problému UNSAT, protože ohodnocení proměnných v, pro které $\varphi(v) = 1$ ukazuje, že φ je splnitelná, a tedy není nesplnitelná. Je vidět, že problémy SAT a UNSAT jsou si velmi podobné, záleží jen na tom, jak položíme otázku. Problém UNSAT je tedy negací nebo doplňkem problému SAT. Třídu jazyků, jejichž doplňky patří do třídy NP, nazveme co-NP.

Definice 14.1.2 (Třída co-NP) Jazyk (problém) $A \subseteq \{0,1\}^*$ patří do třídy co-NP, pokud jeho doplněk $\overline{A} = \{0,1\}^* \setminus A$ patří do třídy NP. ◂

Poznámka 14.1.3 *Vzpomeneme-li si na definici rozhodovacího problému, je formálně jazykem odpovídajícím problému SAT jazyk binárních řetězců, které kódují splnitelné formule, tedy* Kapitola 9.1

$$\text{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ je formule v KNF a } (\exists v)[\varphi(v) = 1]\}\,.$$

Formálně vzato je tedy doplněk jazyka SAT jazykem binárních řetězců, které buď nekódují formule, nebo kódují sice formule, ale nesplnitelné, tedy

$$\overline{\text{SAT}} = \text{UNSAT} \cup \{w \mid w \text{ není kódem formule v KNF}\}\,,$$

kde

$$\text{UNSAT} = \{\langle\varphi\rangle \mid \varphi \textit{ je formule v KNF a } (\forall v)[\varphi(v) = 0]\}$$

je jazykem nesplnitelných formulí. Při běžném kódování jsme však jistě schopni jednoduše poznat, zda daný řetězec kóduje formuli, či nikoli. Můžeme si tedy dovolit jisté zjednodušení a jazyk UNSAT*, tedy jazyk nesplnitelných formulí, opravdu za doplněk jazyka* SAT *považovat.*

Místo problému UNSAT se obvykle uvažuje problém Tautologie (TAUT), kde se pro danou formuli v disjunktivně normální formě (DNF) ptáme, zda je tautologií, tedy zda je splněna při každém ohodnocení proměnných. Není těžké si rozmyslet, že jde ve skutečnosti o týž problém.

Způsob převoditelnosti, pomocí nějž jsme definovali, co je to NP-úplný problém, využijeme i při definici co-NP-úplného problému.

Definice 14.1.4 Jazyk A je co-NP-úplný, patří-li do třídy co-NP a pro libovolný jiný jazyk $B \in$ co-NP platí, že $B \leq_m^p A$. ◂

Protože třídy NP a co-NP mají zjevně mnoho společného, nepřekvapí nás následující tvrzení:

Lemma 14.1.5 *Jazyk A je* co-NP*-úplný, právě když jeho doplněk $\overline{A}$ je* NP*-úplný.*

Důkaz: Plyne přímo z definice převoditelnosti, NP-úplnosti a co-NP-úplnosti. □

Připomeňme si, že třídu NP jsme definovali jako třídu polynomiálně ověřitelných jazyků. Přesněji, jazyk A patří do třídy NP právě když existuje polynomiální verifikátor $V(x, y)$ pro A. Tedy algoritmus $V(x, y)$ pracuje v polynomiálním čase vzhledem k $|x|$ a platí pro něj, že

Definice 9.4.3

$$A = \{x \mid (\exists y \in \Sigma^*)[V(x, y) \text{ přijme}]\}.$$

Z definice polynomiálního verifikátoru plyne, že stačí omezit se na řetězce y, pro něž platí, že $|y| \leq p(|x|)$ pro vhodný polynom p. Pak tedy můžeme psát

$$A = \{x \mid (\exists y \in \Sigma^*)[|y| \leq p(|x|) \text{ a } V(x, y) \text{ přijme}]\}$$
$$\overline{A} = \{x \mid (\forall y \in \Sigma^*)[\text{je-li } |y| \leq p(|x|), \text{ pak } V(x, y) \text{ odmítne}]\}.$$

Uvážíme-li, že je snadné odpověď verifikátoru znegovat, dostáváme alternativní charakterizaci třídy co-NP.

Lemma 14.1.6 *Jazyk A patří do třídy* co-NP*, právě když existuje polynom p a polynomiální verifikátor $V(x, y)$, pro který platí, že*

$$A = \{x \mid (\forall y \in \Sigma^*)[\textit{je-li } |y| \leq p(|x|), \textit{ pak } V(x, y) \textit{ přijme}]\}.$$

Protože třída polynomiálně rozhodnutelných problémů P je zřejmě uzavřená na doplňky, platí, že P $\subseteq$ NP $\cap$ co-NP. K tomu je potřeba dodat, že všeobecně se předpokládá, že NP $\neq$ co-NP a P $\subsetneq$ NP $\cap$ co-NP, přesto se stále může ukázat, že platí opak. Pokud by se ukázalo, že nějaký NP-úplný problém patří do co-NP, pak by bylo NP = co-NP.

Všimněme si na závěr analogie s třídami rozhodnutelných jazyků (DEC), částečně rozhodnutelných jazyků (PD) a třídou doplňků částečně rozhodnutelných jazyků (co-PD). Podle věty 6.1.1 dostáváme, že jazyk A je částečně rozhodnutelný, právě když existuje rozhodnutelný jazyk B, pro který platí, že

$$A = \{x \in \Sigma^* \mid (\exists y \in \Sigma^*)[\langle x, y\rangle \in B]\}.$$

Pro doplněk $\overline{A}$ tedy platí

$$\overline{A} = \{x \in \Sigma^* \mid (\forall y \in \Sigma^*)[\langle x, y\rangle \in B]\}.$$

Jde tedy o podobnou situaci jako v případě tříd P, NP a co-NP. V případě rozhodnutelných a částečně rozhodnutelných jazyků víme na základě věty 6.1.5, že DEC = PD ∩ co-PD. Navíc víme, že existují jazyky, jež jsou částečně rozhodnutelné, nikoli však rozhodnutelné, například univerzální jazyk L_u, a tedy PD ≠ co-PD. V případě složitosti víme jen, že P ⊆ NP ∩ co-NP a ostrá inkluze není vyloučena.

14.2. Početní problémy — třída #P

Je řada úloh, v nichž nás zajímá počet jejich řešení. Můžeme se například ptát, kolik splňujících ohodnocení má daná formule v KNF, nebo kolik perfektních párování má daný bipartitní graf. Úlohám tohoto typu budeme říkat početní úlohy.

Definice 14.2.1 Funkce $f : \Sigma^* \to \mathbb{N}$ patří do třídy #P, pokud existuje polynom p a polynomiální verifikátor V takové, že pro každé $x \in \Sigma^*$

$$f(x) = |\{y \mid |y| \leq p(|x|) \text{ a } V(x, y) \text{ přijme}\}|. \quad \blacktriangleleft$$

S každým problémem $A \in$ NP můžeme asociovat funkci #A v #P. Tato je asociovaná s *přirozeným polynomiálním verifikátorem* pro A, čímž myslíme verifikátor, který ověřuje, zda y je řešením odpovídající úlohy. Například přirozený verifikátor pro Splnitelnost přijme dvojici φ, v, pokud φ je KNF a v je splňující ohodnocení φ. Potom #SAT(φ) = $|\{v \mid \varphi(v) = 1\}|$. Funkce #SAT$(\varphi)$ tedy počítá počet modelů dané formule.

Je-li $f(x)$ funkce v #P, pak problém, v němž se ptáme, zda pro daný vstup $x \in \Sigma^*$ je hodnota $f(x)$ nenulová patří do NP.

Následující tvrzení ukazuje, že pokud nás zajímá polynomiální čas, není příliš velký rozdíl mezi vyřešením úlohy $f(x)$ nebo její rozhodovací verze, tedy dotazu $f(x) \geq N$ pro nějaké $N \in \mathbb{N}$.

Lemma 14.2.2 *Výpočet hodnoty funkce $f \in$ #P lze provést za pomoci polynomiálně mnoha dotazů na náležení prvku do množiny $\{(x, N) \mid f(x) \geq N\}$.*

Důkaz: Protože $f \in$ #P, existuje polynom p a polynomiální verifikátor V, pro který platí, že $f(x) = |\{y \mid |y| \leq p(|x|) \text{ a } V(x, y) \text{ přijme}\}|$. To znamená, že $f(x) \leq 2^{p(|x|)}$, pokud uvažujeme x i y jako binární řetězce. Binárním vyhledáváním najdeme hodnotu $f(x)$ dotazy na $N = 0, \ldots, 2^{p(|x|)}$, protože binární vyhledávání pracuje v logaritmickém čase vzhledem k velikosti prohledávaného intervalu, bude nám na toto vyhledání stačit $p(|x|)$ dotazů, je jich tedy polynomiálně mnoho. □

Lemma 14.2.3 *Nechť $f \in$ #P, pak lze hodnotu $f(x)$ vypočítat v polynomiálním prostoru vzhledem k $|x|$.*

Důkaz: Podobně jako v důkazu 14.2.2 můžeme odvodit, že $f(x) \leq 2^{p}(|x|)$ pro nějaký polynom p. K reprezentaci hodnoty $f(x)$ nám tedy postačí $p(|x|)$ bitů. Algoritmus počítající $f(x)$ bude postupně generovat všechny binární řetězce y délky nejvýš $p(|x|)$. Pro každý řetězec y algoritmus ověří, zda polynomiální verifikátor $V(x, y)$, jehož existenci zaručuje definice 14.2.1, přijme. Pokud ano, zvýší algoritmus aktuální hodnotu $f(x)$ o jedna. Protože verifikátor V pracuje v polynomiálním čase vzhledem k $|x|$, pracuje i v polynomiálním prostoru, a celkový prostor využitý algoritmem je tedy polynomiální. □

I u třídy #P se budeme zajímat o nejtěžší problémy, pokud však chceme převádět na sebe funkce, musíme si definovat takový typ převoditelnosti, který nejen převede odpověď typu ano/ne, ale i hodnotu funkce.

Definice 14.2.4 Řekneme, že funkce $f : \{0,1\}^* \to \mathbb{N}$ je *polynomiálně převoditelná* na funkci $g : \{0,1\}^* \to \mathbb{N}$ ($f \leq_P g$), pokud existují polynomiálně spočitatelné funkce $\alpha : \{0,1\}^* \times \mathbb{N} \to \mathbb{N}$ a $\beta : \{0,1\}^* \to \{0,1\}^*$, pro které platí, že

$$(\forall x \in \{0,1\}^*)\ [f(x) = \alpha(x, g(\beta(x)))].$$ ◂

Relací $f \leq_P g$ chceme zachytit fakt, že pokud dokážeme spočítat hodnotu funkce g, pak dokážeme spočítat jen s polynomiálním zpomalením a jedním výpočtem hodnoty funkce g i hodnotu funkce f, a to tak, že nejprve funkcí β upravíme vstup, spočítáme hodnotu funkce g na novém vstupu a funkcí α přepočítáme hodnotu na tu správnou. I v případě třídy #P se zajímáme o ty nejtěžší úlohy.

Definice 14.2.5 Funkce $f : \{0,1\}^*$ je #P-*těžká*, pokud je každá funkce $g \in$ #P polynomiálně převoditelná na f. Řekneme, že f je #P-*úplná*, je-li #P-těžká a platí-li současně, že $f \in$ #P. ◂

Stejně jako v případě NP-úplnosti, i zde jsme pomocí #P-úplné funkce f schopni spočítat všechny funkce z #P v čase je jen o polynomální hodnotu delší než čas výpočtu $f(x)$. Připomeňme si, že s každým problémem A ze třídy NP můžeme asociovat početní úlohu (tedy funkci) #A, která počítá počet certifikátů (často řešení úlohy spojené s A), jež dosvědčují pro daný vstup x, že je kladnou instancí do A. Uvažme nyní dva problémy $A, B \subseteq \Sigma^*$ a jim odpovídající početní úlohy #A a #B. Pokud platí, že $A \leq_m^p B$, přičemž tento převod zachová počet řešení úloh spojených s A a B, pak lze jednoduše převést i #A na #B.

Definice 14.2.6 Řekneme, že problém $A \in \Sigma^*$ je převoditelný na problém $B \in \Sigma^*$ *v polynomiálním čase se zachováním počtu řešení* ($A \leq_c^P B$), pokud existuje funkce $f : \Sigma^* \to \Sigma^*$ vyčíslitelná v polynomiálním čase, pro kterou platí, že

$$|\{y \mid V_A(x, y) \text{ přijme}\}| = |\{y \mid V_B(f(x), y) \text{ přijme}\}|,$$

kde V_A a V_B jsou přirozené verifikátory pro A a B. ◂

Z této definice i předchozích úvah plyne následující pozorování.

Lemma 14.2.7 *Nechť B je problém v* NP*, pro který platí, že každý jiný problém A v* NP *je převoditelný na B v polynomiálním čase se zachováním počtu řešení, tedy $A \leq_c^P B$. Pak #B je* #P*-úplná úloha.*

Převody, které jsme si ukazovali, lze udělat tak, aby zachovávaly počty řešení, proto lze ukázat, že úlohy odpovídající problémům, jejichž těžkost jsme si ukazovali, jsou #P-úplné. Například tedy #KACHL, #SAT, #HK, #LOUP a další jsou všechno #P-úplné úlohy.[1] Kromě toho však existují i úlohy, které jsou sice polynomiálně řešitelné, ale s nimi asociovaná početní úloha je #P-úplná. Jde například o případ splnitelnosti formule v disjuntivní normální formě. Problém DNF-SAT, tedy splnitelnost formule v DNF je polynomiálně řešitelný, což ponecháme čtenáři jako jednoduché cvičení. Jak je to však s určením počtu splňujících ohodnocení formule v DNF?

Věta 14.2.8 *Funkce #DNF-SAT je* #P*-úplná.*

Důkaz: Protože problém DNF-SAT zřejmě patří do NP (dokonce do do P), patří funkce #DNF-SAT do #P. Popíšeme, jak převést #SAT (tj. #KNF-SAT, chceme-li explicitně zdůraznit, že jde o počítání modelů formule v KNF) na #DNF-SAT. Protože #KNF-SAT je #P-úplná funkce, bude totéž platit i proo #DNF-SAT. Nechť φ je formule v KNF, pomocí de-Morganových pravidel převedeme její negaci $\neg\varphi$ na DNF, kterou si označíme pomocí $\psi \equiv \neg\varphi$, pak platí, že

$$\#\text{KNF-SAT}(\varphi) = 2^n - \#\text{DNF-SAT}(\psi),$$

kde n je počet proměnných formule φ. V převodu tedy funkce $\psi = \beta(\varphi)$ spočítá DNF negace zadané formule v KNF a funkce $\alpha(\varphi, c)$ odečte počet $c = \#\text{DNF-SAT}(\psi)$ od 2^n. To vše lze jistě provést v polynomiálním čase. □

Mírnou úpravou převodu popsaného v důkazu věty 14.2.8 bychom mohli ukázat #P-úplnost funkce #KNF-FALSE, tedy určení počtu ohodnocení, která nesplňují formuli φ. V určitém smyslu je tedy třída #P uzavřená na doplňky, a to proto, že jsme obvykle schopni spočítat počet možných kandidátů na řešení, který je v tomto případě 2^n. Z tohoto hlediska není nijak překvapivé, že #DNF-SAT je #P-úplná funkce.

Existují však relace, které nemají takto přímočarou souvislost s nějakým těžkým problémem, a přesto jsou s nimi asociované početní úlohy #P-úplné. Příkladem může být určení počtu perfektních párování v bipartitním grafu. Nalézt jedno perfektní párování v bipartitním grafu, pokud existuje, můžeme v polynomiálním čase například pomocí toků v sítích. Určit počet perfektních párování v bipartitním grafu je však #P-úplná úloha. Tato úloha je ekvivalentní výpočtu permanentu matice. Permanent čtvercové matice A typu $n \times n$ je definován jako

$$per(A) = \sum_{\pi \in S(n)} \prod_{i=1}^{n} A[i, \pi(i)], \tag{14.1}$$

[1] Není ovšem známo, zda početní úlohy asociované s NP-úplnými problémy jsou současně #P-úplné. Viz též http://cstheory.stackexchange.com/questions/25644/p-complete-problems-are-at-least-as-hard-as-np-complete-problems

kde prvky $S(n)$ jsou permutace množiny $\{1,\ldots,n\}$. Všimněme si, že vzorec (14.1) je téměř shodný se vzorcem pro výpočet determinantu matice. V případě permanentu však nebereme do úvahy znaménko permutace. Zatímco tedy spočítat determinant matice lze pomocí Gaussovy eliminační metody v polynomiálním čase, tak spočítat permanent je #P-úplné, a to i v případě, kdy prvky matice nabývají pouze hodnoty 0 nebo 1.

14.3. Cvičení

1. Rozmyslete si, jak obtížné je rozhodnutí následujících problémů v závislosti na tom, je-li φ v KNF, DNF, či jde-li o obecnou formuli výrokové logiky, přesněji, rozhodněte jsou-li v P, NP či co-NP a jsou-li NP-těžké či co-NP-těžké.
   a) $(\exists v)\varphi(v) = 1$ (tj. SAT ve verzi pro KNF, DNF, obecnou formuli).
   b) $(\exists v)\varphi(v) = 0$ (tj. FALS ve verzi pro KNF, DNF, obecnou formuli).
   c) $(\forall v)\varphi(v) = 1$ (tj. TAUT ve verzi pro KNF, DNF, obecnou formuli).
   d) $(\forall v)\varphi(v) = 0$ (tj. UNSAT ve verzi pro KNF, DNF, obecnou formuli).
2. Je-li φ v KNF uvažme následující algoritmus pro test splnitelnosti: Převeď φ do DNF ψ a otestuj splnitelnost splnitelnost ψ algoritmem pro DNF-SAT. Bude takový algoritmus polynomiální? Odpověď zdůvodněte.

Literatura

[1] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[2] Stephen A Cook and Robert A Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[3] 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.

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

[5] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[6] L. A. Levin. Universal'nye zadachi perebora. *Probl. peredachi inform.*, 9(3):115–116, 1973.

[7] P. Odifreddi. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers (Studies in Logic and the Foundations of Mathematics)*. North Holland, new edition, February 1992.

[8] Cees Slot and P Boas. On tape versus core an application of space efficient perfect hash functions to the invariance of space. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 391–400. ACM, 1984.

[9] Robert I. Soare. *Recursively enumerable sets and degrees*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[10] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, November 1936.