Poznámky k přednášce NTIN090 Úvod do složitosti a vyčíslitelnosti

Petr Kučera

12. února 2016

Obsah

Část I

Úvod

Kapitola 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.

Část II

Vyčíslitelnost

Kapitola 2

Algoritmy a výpočetní modely

2.1 Churchova-Turingova teze

Intuitivně je algoritmus konečná posloupnost jednoduchých instrukcí (příkazů, pokynů), která vede k řešení zadané úlohy. V tomto smyslu se dá vzít jako algoritmus 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.

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. Jeden z takových modelů jsou Turingovy stroje, které, připustíme-li platnost Churchovy-Turingovy teze, zachycují přesně intuitivní pojem algoritmu. Dalším model, které zde uvážíme bude model RAM, tedy random access machine, který se blíží reálnému počítači tím, že připouští náhodný přístup do paměti. Zmíníme (i když spíše jen okrajově), model částečně rekurzivních funkcí. Tím popíšeme reprezentanty tří paradigmat programování, zatímco Turingovy stroje se programují deklarativním způsobem, částečně rekurzivní funkce funkcionálně a RAM imperativně.

Všechny tyto výpočetní modely poskytují stejně dobré prostředky a dávají vzniknout týmž třídám jazyků a funkcí. Ve skutečnosti existuje bezpočet dalších modelů, které jsou s Turingovými stroji, RAMem a ČRF ekvivalentní, 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-kalkulu (což je teoretický základ všech funkcionálních jazyků, pochází od Churche, 1936), 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 2.1.1 (Churchova-Turingova (1936)) *Ke každému algoritmu v intuitivním smyslu existuje Turingův stroj, který jej implementuje.*

Všimněme si, že tato teze se snaží spojit intuici s matematickým pojmem, a proto ji nelze formálně dokázat, nejde proto o větu či podobné matematické tvrzení.

2.2 Turingovy stroje

Turingovy stroje budou naším hlavním modelem, ke kterému se uchýlíme vždy, když budeme potřebovat říci něco formálním způsobem, například v definicích základních pojmů jak ve vyčíslitelnosti (např. pojmů rozhodnutelnosti či částečné rozhodnutelnosti, převoditelnosti atd.) tak ve složitosti (například v definici základních tříd složitosti, tříd P, NP a dalších). Je proto přirozené začít popisem právě tohoto modelu.

2.2.1 Definice

Definice 2.2.1 (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,
- $\delta : Q \times \Sigma \mapsto Q \times \Sigma \times \{R, N, L\} \cup \{\perp\}$ 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 nekonečné pásky a hlavy pro čtení a zápis, která se může pohybovat po pásce oběma směry.

Konfigurace TS se skládá ze stavu řídící jednotky, slova na pásce (od nejlevějšího do nejpravějšího neprázdného políčka) a pozice hlavy na pásce (v rámci slova na této pásce). Konfigurace TS v sobě tedy zahrnuje vše, co určuje aktuální stav výpočtu TS.

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 $a \in \Sigma$, 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 toho, jak vypadá výpočet Turingova stroje, vidíme, že tabulka přechodové funkce je deklarativním popisem toho, co se má stát v každé konfiguraci. Způsob programování Turingova stroje má tedy blízko k deklarativním jazykům.

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**.

Příklad 2.2.2 *Ukážeme například TS, který přijímá jazyk* $\{0^n1^n \mid n \geq 0\}$. *Rozmysleme si nejprve algoritmus, který by tento jazyk mohl rozpoznávat a který potom budeme implementovat na TS.*

1: Je-li vstup prázdný, přijmi.
2: Není-li prvním znakem vstupu znak 0, odmítni.
3: Smaž počáteční znak 0 a odjeď hlavou na konec vstupu.
4: Není-li posledním znakem vstupu znak 1, odmítni.
5: Smaž poslední znak 1 a odjeď hlavou na začátek vstupu.
6: **goto** 1

Rozmysleme si nyní, jak tento algoritmus implementovat na TS. Nejprve popíšeme tabulkou přechodovou funkci, potom teprve z ní vyčteme, 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.*

q, a	$\rightarrow q', a', Z$
q_0, λ	$\rightarrow q_1, \lambda, N$
$q_0, 0$	$\rightarrow q_2, \lambda, R$
$q_2, 0$	$\rightarrow q_2, 0, R$
$q_2, 1$	$\rightarrow q_2, 1, R$
q_2, λ	$\rightarrow q_3, \lambda, L$
$q_3, 1$	$\rightarrow q_4, \lambda, L$
$q_4, 1$	$\rightarrow q_4, 1, L$
$q_4, 0$	$\rightarrow q_4, 0, L$
q_4, λ	$\rightarrow q_0, \lambda, R$

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

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{0, 1, \lambda\}$,
- δ *je určená výše uvedenou tabulkou.*
- $F = \{q_1\}$.

Uvědomme si, že odmítnutí není ani tak dáno tím, co je napsané v přechodové funkci, jako spíš tím, co v ní napsané není. Není-li například prvním znakem vstupu 0, dojde k odmítnutí proto, že v přechodové funkci $\delta(q_0,1) = \perp$, *tedy tento přechod není definován.*

V uvedeném příkladu jsme tiše předpokládali, že vstupní slovo se skládá pouze z 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 TS ve skutečnosti selhal na slovech typu $0^n1^n\lambda^k0$ pro libovolné $k > 0$. Ve skutečnosti by ale TS nebyl schopen podobné případy rozpoznat, pokud by neznal dopředu horní odhad na hodnotu k, protože pokud by přečetl několik znaků λ, nemohl by poznat, je-li už na konci vstupu, nebo jen ještě nepřeskočil dostatek vložených mezer. Z těchto důvodů se hodí přijmout omezení, podle nějž nejsou znaky λ součástí vstupu. Mohli bychom sice používat prázdných políček například k oddělení jednotlivých částí vstupu, ale k tomuto účelu můžeme klidně použít i jiný znak.

Ačkoli jsme do formální definice nezahrnuli vstupní abecedu, budeme ji přesto občas využívat, v našem případě půjde většinou o abecedu binární $\{0, 1\}$.

U Turingova stroje M nás může jednak zajímat, zda dané slovo w přijme, tedy zda jeho výpočet nad slovem w skončí v přijímajícím stavu. Může nás ale také zajímat, co po (případném)

ukončení výpočtu nad slovem w zanechá Turingův stroj M na své pásce. V tomto případě se zajímáme o to, jakou funkci M počítá. Každý Turingův stroj počítá přirozeným způsobem částečnou funkci $g : \Sigma^* \mapsto \Sigma^*$ mapující řetězce na jiné řetězce. Jde o částečnou funkci, neboť pro některé vstupy se výpočet M nemusí zastavit. Jako další stupeň abstrakce budeme uvažovat nejen jazyky, tedy množiny řetězců, ale i množiny přirozených čísel. Zavedeme si zde tedy i způsob, jakým mohou Turingovy stroje počítat právě funkce nad přirozenými čísly. Pro kódování čísel do řetězce, který můžeme předat Turingovu stroji jako vstup na jeho pásku, se zdá přirozené použít binárního kódování.

Definice 2.2.3 Řekneme, že TS M **počítá (částečnou) funkci** $f : \mathbb{N} \mapsto \mathbb{N}$, pokud pro $x \in \mathbb{N}$ stroj M s číslem x zapsaným na vstupu v binární soustavě[1] skončí výpočet v konfiguraci se slovem y zapsaným na pásce v binární soustavě, právě když $f(x)\downarrow = y$. Pokud hodnota $f(x)$ není definována, pak M buď svůj výpočet neskončí, nebo po ukončení výpočtu není slovo na pásce reprezentací čísla v binární soustavě. V případě funkce více proměnných předpokládáme, že čísla jsou na vstupu oddělená symboly „#“. O funkci f, pro níž existuje TS, který ji počítá, řekneme, že je *turingovsky vyčíslitelná*.

Podobně jako u TS, u částečné funkce $f : \mathbb{N} \mapsto \mathbb{N}$ označíme pomocí $f(x)\downarrow$ fakt, že hodnota $f(x)$ je pro x definována, pomocí $f(x)\uparrow$ označíme fakt, že hodnota $f(x)$ pro x definována není.

Uvědomme si, že podle definice 2.2.3 počítá každý TS pro každé n nějakou funkci n proměnných.

Příklad 2.2.4 *Popíšeme TS M, který bude počítat funkci $f(x) = x + 1$. Práce stroje je v tomto případě jednoduchá a řídí se běžným algoritmem na přičtení jedničky k binárnímu číslu. Pokud je vstup prázdný, zapíše M na vstup 1 a skončí, zde je třeba dodat, že v definici počítání funkce jsme nespecifikovali, jak se má počítající Turingův stroj zachovat ve chvíli, pokud vstup není syntakticky správně zapsaný. V tomto případě dává smysl uvážit prázdný vstup jako 0 a zapsat místo něj 1. Pokud by se však ve vstupu třeba vyskytoval oddělující znak „#“, nebude již výsledek specifikován, lépe řečeno, TS v tom případě nebude mít definovanou pokračující instrukci a neučiní nic. V případě, že je vstup neprázdný, dojede M s hlavou na jeho poslední znak. Poté se vrací s tím, že dokud čte 1, přepisuje je na 0 a ve chvíli, kdy narazí na 0 nebo λ, přepíše znak na 1, poté se vrátí na začátek, to jen v případě, kdy nečetl λ ale 0. Návrat na začátek není nutností, ale slušností, na konci tak bude hlava na prvním znaku výstupu. Zápis do instrukcí je obsažen v následující tabulce.*

	$q, a \quad \rightarrow q', a', Z$
1	$q_0, \lambda \rightarrow q_1, \lambda, N$
2	$q_0, 0 \rightarrow q_2, 0, R$
3	$q_0, 1 \rightarrow q_2, 1, R$
4	$q_2, 0 \rightarrow q_2, 0, R$
5	$q_2, 1 \rightarrow q_2, 1, R$
6	$q_2, \lambda \rightarrow q_3, \lambda, L$
7	$q_3, 1 \rightarrow q_3, 0, L$
8	$q_3, \lambda \rightarrow q_1, 1, N$
9	$q_3, 0 \rightarrow q_4, 1, L$
10	$q_4, 0 \rightarrow q_4, 0, L$

[1] Namísto binárního kódování čísla x bychom mohli využít i unárního kódování pomocí 1^x, stejně jako jiné než binární soustavy, pokud se zajímáme jen o řešitelnost, a nikoli o čas či potřebný prostor, není mezi tím rozdíl, neboť mezi různými zápisy čísel lze efektivně, tedy algoritmicky, převádět.

11	$q_4, 1 \rightarrow q_4, 1, L$
12	$q_4, \lambda \rightarrow q_1, \lambda, R$

V této tabulce předpokládáme, že q_0 je počáteční a q_1 přijímající stav, a tedy pokud vydá stroj správný výsledek a nedojde k chybě, skončí práci přijetím, což však není pro výpočet funkce nutné. Instrukce číslo 1 odpovídá zapsání „1“ místo prázdného slova a ukončení práce. Instrukce 2 až 6 odpovídají pohybu hlavy na konec vstupu a přechodem zpět zpoza konce na poslední znak vstupu. Instrukce 7 odpovídá přepisu 1 na 0 s návratem vlevo. Instrukce 8 a 9 ukončí přepis 1 na 0, buď s ukončením práce dojde-li na začátek vstupu v instrukci 8, kdy přepíše λ na 1 a skončí, nebo s návratem na začátek ve zbývajících instrukcích, pokud M přepíše 0 na 1 instrukcí 9.

My nadále nebudeme popisovat stroje takto detailně rozepsáním jejich přechodové funkce, zůstaneme obvykle u slovního popisu toho, jak bude daný Turingův stroj postupovat, později se budeme i tomuto přístupu vzdalovat a zůstávat na vyšších úrovních popisu algoritmu, to až se sžijeme s Turingovými stroji natolik, abychom přijali Churchovu-Turingovu tezi za vlastní. Ačkoli naším výpočetním modelem budou Turingovy stroje a RAM, naším zájmem jsou spíše algoritmy a nikoli konkrétní výpočetní modely. Často také budeme používat pseudokód odpovídající spíš vyššímu procedurálnímu jazyku, přičemž s odkazem na Churchovu-Turingovu tezi poté prohlásíme, že takto zapsaný algoritmus by bylo lze implementovat na Turingovu stroji.

Námi popsaný Turingův stroj má řadu variant, které jsou ekvivalentní co do výpočetní síly. Jde například o TS s jednosměrně nekonečnou páskou; vícepáskový TS; TS připouštějící více hlav na jedné pásce, ať už čtecích či zápisových. Některé z těchto variant si probereme v rámci cvičení. Zvlášť vícepáskový TS budeme v dalším textu občas využívat. V případě více pásek je obvykle jedna páska určena jako vstupní a jedna jako výstupní, přičemž může jít i o jednu pásku. 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 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, v tom případě se nezřídka hodí nepočítat do tohoto prostoru velikost vstupu a výstupu. Počet pásek je pochopitelně pro daný TS pevný a nezávislý na vstupu, stejně tak je pevná například velikost abecedy. V rámci cvičení si ukážeme, jak převést libovolný vícepáskový TS na jednopáskový TS, který pracuje stejně, toho, že lze tento převod provést využijeme v dalším textu v některých důkazech a zejména při konstrukci univerzálního TS.

Často budeme využívat zejména toho, že vícepáskové stroje jsou co do výpočetní síly ekvivalentní s jednopáskovými. Zkuste si třeba rozmyslet, jak by bylo komplikované popsat jednopáskový stroj pro sčítání dvou binárních čísel oproti vícepáskovému stroji.

Zadefinujme si model vícepáskového stroje formálněji.

Definice 2.2.5 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,
- $\delta : Q \times \Sigma^k \mapsto Q \times \Sigma^k \times \{R, N, L\}^k \cup \{\perp\}$ 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.

Poznámka 2.2.6 *Často budeme říkat, že jeden TS simuluje práci jiného TS, například univerzální TS může simulovat práci jiného TS, jehož kód dostane na vstupu. Tento pojem nebudeme zavádět příliš formálně, nicméně budeme jej používat v následujícím smyslu. Je-li M_1 TS, 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.*

Poněkud slabším požadavkem je, že 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 stejně jako výpočet M_4.

Následující tvrzení, které budeme často používat, si ukážeme v rámci cvičení.

Věta 2.2.7 *Ke každému k-páskovému TS M existuje jednopáskový TS M', který simuluje práci M, přijímá týž jazyk jako M a počítá touž funkci jako M.*

2.2.2 Kódování TS a Gödelovo číslo

Naším dalším cílem bude dospět k univerzálnímu Turingovu stroji. Univerzální Turingův stroj (UTS) očekává na vstupu kód čili program Turingova stroje M a vstupní slovo w, jeho výpočet potom simuluje práci stroje M nad slovem w ve smyslu, který jsme popsali v poznámce 2.2.6. UTS svůj výpočet skončí, pokud by i M skončil výpočet a UTS přijme, pokud by i M přijal. To, proč vůbec chceme dospět k univerzálnímu TS, má několik důvodů, za prvé náš popis výpočtu Turingova stroje v části 2.2 je jistě intuitivním algoritmem, který popisuje, jak simulovat práci TS zadaného jeho přechodovou funkcí. Pokud připustíme Churchovu-Turingovu tezi, měli bychom být schopni takový algoritmus přepsat do podoby univerzálního Turingova stroje, jeho konstrukci tedy můžeme brát jako jakýsi test Churchovy-Turingovy teze. Druhým důvodem je, že chceme k Turingovým strojům přistupovat určitým způsobem uniformě a v řadě důkazů a argumentaci o Turingových strojích se nám bude hodit, že každý Turingův stroj lze zakódovat do podoby čitelné univerzálním Turingovým strojem. V neposlední řadě je UTS sám Turingovým strojem a jeho konstrukce současně poslouží jako netriviální příklad TS, který nám pomůže sžít se s modelem Turingových strojů. Než však budeme moci popsat samotný UTS, musíme nějakým způsobem zakódovat program Turingova stroje tak, aby jej byl UTS schopen zpracovat. Naše kódování nám posléze umožní každému TS přiřadit přirozené číslo. Navíc, protože námi popsaný UTS bude vyhovovat těmto omezením, bude i UTS mít vlastní kód a bude tedy schopen simulovat i sám sebe.

Z technických důvodů se omezíme na stroje, jejichž vstupní i výstupní abeceda je pouze dvouprvková $\{0, 1\}$, to proto, že vstupní abeceda UTS musí obsahovat symboly vstupní abecedy simulovaného stroje. Můžeme se omezit na libovolně velkou abecedu, volba binární

abecedy je vhodná jednak v tom, že je dostatečně malá, aby se s ní dobře pracovalo, i dostatečně velká, aby do ní bylo lze jednoduše zakódovat cokoli. 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 (to sice nakonec stejně učiníme, ale až ve druhém kroku). Na druhou stranu tři prvky už jsou zbytečně moc. Všimněme si, 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.

Bez újmy na obecnosti budeme navíc předpokládat, že kódovaný stroj má pouze jeden přijímající stav, což není v podstatě žádné omezení, 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 přejdou 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.

Za podmínek, které klademe na Turingův stroj, stačí k jeho popisu vhodným způsobem zakódovat přechodovou funkci. Dosud jsme uvažovali přechodovou funkci zapsanou v tabulce, což je vlastně již jejím kódem ve vhodné abecedě, zapíšeme-li jednotlivé řádky tabulky za sebe. Jeden řádek tabulky přitom odpovídá jedné instrukci. Začneme tedy tím, že jednotlivé instrukce zakódujeme v abecedě $\Gamma = \{0, 1, L, N, R, |, \#, ;\}$ a zapíšeme je za sebe. Ve skutečnosti si pro jednu instrukci vystačíme s první šesticí znaků, znak # posléze použijeme k oddělení jednotlivých instrukcí a znak ; si zde rezervujeme, abychom jej později mohli použít pro oddělení kódu stroje od jeho vstupu na vstupní pásce univerzálního TS. Poté převedeme znaky abecedy Γ do binární abecedy $\{0, 1\}$, čímž vznikne řetězec v binární abecedě popisující přechodovou funkci daného Turingova stroje.

Uvažme tedy TS $M = (Q, \Sigma, \delta, q_0, F)$. Budeme předpokládat, že $Q = \{q_0, q_1, \ldots, q_r\}$ pro nějaké $r \geq 1$, kde q_0 označuje počáteční stav a q_1 jediný přijímající stav (tímto je dané očíslování stavů). Podobně budeme předpokládat, že $\Sigma = \{X_0, X_1, X_2, \ldots, X_s\}$ pro nějaké $s \geq 2$, kde X_0 označuje symbol 0, X_1 označuje symbol 1 a X_2 označuje symbol pro prázdné políčko, tedy λ, $X_3, \ldots, X_s$ (pokud $s > 2$) pak 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:

$$(i)_B|(j)_B|(k)_B|(l)_B|Z$$

Zde pomocí $(x)_B$ označujeme zápis čísla x v binární soustavě pomocí symbolů 0 a 1. Nechť C_1 až C_n označují kódy instrukcí M, potom kód stroje M vznikne jejich konkatenací s použitím oddělovače „#“, tedy kódem M je řetězec:

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

Znaky abecedy Γ zapíšeme v binární abecedě každý pomocí tří bitů podle následující tabulky:

Γ	kód
0	000
1	001
L	010
N	011

Γ	kód
R	100
|	101
#	110
;	111

Náhradou znaků v kódu jejich binárními zápisy dostaneme binární řetězec reprezentující přechodovou funkci M.

Příklad 2.2.8 *Uvažme například Turingův stroj počítající funkci $f(x) = x + 1$ z příkladu 2.2.4. Zakódujeme-li jeho instrukce do řetězce v abecedě Γ, dostaneme:*

	$q, a \rightarrow q', a', Z$	*kód*
1	$q_0, \lambda \rightarrow q_1, \lambda, N$	$0\|10\|1\|10\|N$
2	$q_0, 0 \rightarrow q_2, 0, R$	$0\|0\|10\|0\|R$
3	$q_0, 1 \rightarrow q_2, 1, R$	$0\|1\|10\|1\|R$
4	$q_2, 0 \rightarrow q_2, 0, R$	$10\|0\|10\|0\|R$
5	$q_2, 1 \rightarrow q_2, 1, R$	$10\|1\|10\|1\|R$
6	$q_2, \lambda \rightarrow q_3, \lambda, L$	$10\|10\|11\|10\|L$
7	$q_3, 1 \rightarrow q_3, 0, L$	$11\|1\|11\|0\|L$
8	$q_3, \lambda \rightarrow q_1, 1, N$	$11\|10\|1\|1\|N$
9	$q_3, 0 \rightarrow q_4, 1, L$	$11\|0\|100\|1\|L$
10	$q_4, 0 \rightarrow q_4, 0, L$	$100\|0\|100\|0\|L$
11	$q_4, 1 \rightarrow q_4, 1, L$	$100\|1\|100\|1\|L$
12	$q_4, \lambda \rightarrow q_1, \lambda, R$	$100\|10\|1\|10\|R$

Spojením těchto kódů pomocí oddělovače # získáme kód:

0 | 10 | 1 | 10 | N # 0 | 0 | 10 | 0 | R # 0 | 1 | 10 | 1 | R # 10 | 0 | 10 | 0 | R #
10 | 1 | 10 | 1 | R # 10 | 10 | 11 | 10 | L # 11 | 1 | 11 | 0 | L # 11 | 10 | 1 | 1 | N #
11 | 0 | 100 | 1 | L # 100 | 0 | 100 | 0 | L # 100 | 1 | 100 | 1 | L # 100 | 10 | 1 | 10 | R

Přepíšeme-li tento kód do binární abecedy podle výše uvedené tabulky, dostaneme binární řetězec kódující Turingův stroj M.

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, a navíc, že tato čísla můžeme s použitím uvozujících 0 zapsat mnoha způsoby, získáme 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 TS (počet bitů v syntakticky správném řetězci musí například být dělitelný třemi), abychom se vyhnuli technickým obtížím s tím spojeným, budeme předpokládat, že pokud binární řetězec w nekóduje žádný TS, odpovídá w „prázdnému“ TS, který okamžitě po zahájení práce skončí se zamítnutím vstupu. K tomu stačí, pokud neobsahuje žádné instrukce a počáteční stav není současně přijímajícím, což v našem případě ani nemůže být. To odpovídá tomu, že pokud univerzální stroj dostane na vstup špatně zapsaný program Turingova stroje, odmítne jej jako syntakticky chybný, aniž by odsimuloval jediný krok stroje na vstupu. Poznat, zda daný řetězec je syntakticky správným kódem Turingova stroje, lze přitom celkem snadno. Blíže si to popíšeme při konstrukce univerzálního Turingova stroje.

Mějme binární řetězec w a přiřaďme mu přirozené číslo jehož binární zápis je $1w$. Prázdnému řetězci ε tak odpovídá 1, řetězci 0 odpovídá číslo 2, řetězci 01 odpovídá číslo 5 a tak podobně. Toto přiřazení je zřejmě vzájemně jednoznačné, tj. kladnému číslu jednoznačně odpovídá řetězec a naopak, dostali jsme tedy očíslování binárních řetězců, i-tý řetězec budeme označovat pomocí w_i. Důvod, proč musíme použít číslo dané $1w$ a ne přímo w je ten, že například 000 je jiný řetězec, než 0, což nám 1 na začátku umožní rozlišit. Tímto jsme nepřiřadili žádný řetězec číslu 0, mohli bychom sice odečíst jedna od výsledného čísla a tím posunout číslování k 0, ale tím bychom ztratili přímočarost převodu. Místo toho budeme uvažovat, že číslu 0 odpovídá prázdný řetězec. Vzniklá nejednoznačnost, totiž že nyní prázdnému řetězci odpovídají dvě čísla 0 a 1, nám nebude činit obtíže.

Je-li w_e kódem TS M, pak e nazveme **Gödelovým číslem** stroje M. Protože jsme přiřadili prázdný stroj i řetězcům, které nekódují žádný 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). Stroj s číslem 0 budeme také uvažovat jako prázdný. Turingův stroj s Gödelovým číslem e, tedy s kódem w_e, budeme označovat pomocí M_e. Víme, že jeden Turingův stroj může mít nekonečně mnoho binárních řetězců, která jej kódují, a má tedy i nekonečně mnoho Gödelových čísel.

2.2.3 Rekurzivní a rekurzivně spočetné jazyky

Nyní jsme připraveni upřesnit to, co míníme algoritmicky řešitelným problémem. Mějme TS M, pomocí $L(M)$ označíme *jazyk slov přijímaných* M, tj. $L(M)$ obsahuje slovo w právě tehdy, když se výpočet M nad w zastaví a M skončí v přijímajícím stavu, tedy přijme slovo w. Slovo w do $L(M)$ nepatří, pokud jej M odmítne, nebo pokud se výpočet M nad w nezastaví.

Jazyk L je *rekurzivně spočetný* (také bychom mohli říci *částečně rozhodnutelný*), pokud existuje TS M, pro nějž $L = L(M)$. Jazyk L je *rekurzivní* (také *rozhodnutelný*), pokud existuje TS M, který se vždy zastaví a $L = L(M)$. Zřejmě platí, že každý rekurzivní jazyk je i rekurzivně spočetný. Jakýkoli rozhodovací problém se dá formulovat jako jazyk řetězců popisujících pozitivní instance a otázku, zda dané slovo – instance problému – patří do tohoto jazyka, tedy je kladnou instancí daného problému. Pojem rekurzivního jazyka se tedy podle Churchovy-Turingovy teze shoduje s tím, co intuitivně považujeme za algoritmicky řešitelný rozhodovací problém. Díky tomu, že jsme očíslovali Turingovy stroje, máme rovnou očíslované i rekurzivně spočetné jazyky, pomocí L_e označíme jazyk přijímaný strojem M_e, tj. $L_e = L(M_e)$.

Nyní se naskýtá přirozená otázka, zda jsou všechny jazyky rekurzivně spočetné. Odpověď je velmi jednoduchá, není to možné proto, že rekurzivně spočetných jazyků je jen spočetně mnoho, jelikož se nám je podařilo očíslovat přirozenými čísly, zatímco už všech jazyků nad jednoprvkovou abecedou, tedy podmnožin 1^*, je nespočetně mnoho. Každý jazyk nad jednoprvkovou abecedou $\{1\}$ lze totiž jednoznačně identifikovat s množinou přirozených čísel a těch je nespočetně mnoho. V tomto smyslu je jazyků nad binární (i jakoukoli větší) abecedou stejné množství, neboť každý jazyk nad binární abecedou si vzájemně jednoznačně odpovídá s jazykem nad jednoprvkovou abecedou díky očíslování řetězců, které jsme právě popsali. Z těchto úvah plyne, že ve skutečnosti většina jazyků rekurzivně spočetných není, tím spíše to platí o jazycích rekurzivních, a tedy algoritmicky rozhodnutelných problémech. To, že většina problémů není algoritmicky rozhodnutelných, nás ale nemusí většinou trápit, protože obvyklé praktické problémy rozhodnutelné jsou.

Nejjednodušší ukázku jazyka, který není rekurzivně spočetný, představuje asi diagonální jazyk L_{DIAG}, který definujeme následujícím způsobem.

$$L_{DIAG} = \{w_i \in \{0,1\}^* \mid w_i \notin L(M_i)\}$$

Důvod, proč se tomuto jazyku říká diagonalizační je následující, představíme-li si tabulku, v níž sloupce jsou indexovány čísly Turingových strojů a řádky čísly řetězců, dostaneme jazyk L_{DIAG} jako negaci diagonály této tabulky.

Věta 2.2.9 *Jazyk L_{DIAG} není rekurzivně spočetný.*

Důkaz : Sporem předpokládejme, že existuje Turingův stroj M, pro nějž $L_{DIAG} = L(M)$. TS M má jistě nějaký kód a odpovídající Gödelovo číslo e, tedy $L_{DIAG} = L(M_e)$. Protože $L_{DIAG} = L(M_e)$, platí, že $w_e \in L_{DIAG}$, právě když $w_e \in L(M_e)$. Na druhou stranu však podle definice L_{DIAG} je $w_e \in L_{DIAG}$ ekvivalentní tomu, že $w_e \notin L(M_e)$. Dohromady dostáváme, že $w_e \in L_{DIAG}$, právě když $w_e \notin L_{DIAG}$, což pochopitelně není možné, a došli jsme tedy ke sporu. Uvedené ekvivalence můžeme rozepsat následujícím způsobem:

$$w_e \in L(M_e) \Leftrightarrow w_e \in L_{DIAG} \Leftrightarrow w_e \notin L(M_e) \ ,$$

kde první ekvivalence vyplývá z toho, že $L_{DIAG} = L(M_e)$ a druhá ekvivalence z definice L_{DIAG}. Důkaz je nejjednodušší ukázkou diagonalizace a je založen na témž principu jako důkaz toho, že reálných čísel není spočetně mnoho. ■

Ačkoli definice jazyka L_{DIAG} vypadá poněkud uměle, již zanedlouho jej využijeme při zkoumání jazyka univerzálního Turingova stroje. K vlastnostem rekurzivních a rekurzivně spočetných jazyků a množin se dostaneme později, ale už nyní si můžeme ukázat dvě nejzákladnější, které budeme často používat. Obě tvrzení dohromady jsou jen části ekvivalence, které se říká Postova věta. Pomocí $\overline{L}$ budeme označovat doplněk jazyka L, tedy $\overline{L} = \{0,1\}^* \setminus L$.

Lemma 2.2.10 *Pokud je L rekurzivní jazyk, je rekurzivní i $\overline{L}$.*

Důkaz : Předpokládejme, že L je rekurzivní jazyk, což podle definice znamená, že existuje TS M, který se vždy zastaví a $L = L(M)$. Turingův stroj M', který přijímá jazyk $\overline{L}$ (tj. $\overline{L} = L(M')$) pracuje stejně jako M, jen na závěr neguje odpověď, tj. M' vypadá stejně jako M jen u něj prohodíme význam přijímajícího a nepřijímajícího stavu. Přesněji, je-li $M = (Q, \Sigma, \delta, q_0, F)$, pak můžeme položit $M' = (Q, \Sigma, \delta, q_0, Q \setminus F)$. ■

Trochu složitější je následující tvrzení.

Lemma 2.2.11 *Jsou-li L i $\overline{L}$ rekurzivně spočetné jazyky, pak L je rekurzivní jazyk.*

Důkaz : Mějme TS M_a a M_b takové, že $L = L(M_a)$ a $\overline{L} = L(M_b)$. Sestrojíme TS M, který se vždy zastaví a bude platit, že $L = L(M)$. Idea práce M je jednoduchá – pusť M_a i M_b paralelně a počkej až jeden z nich skončí přijetím, protože každé slovo patří buď do L nebo do $\overline{L}$, je zřejmé, že se M zastavit musí. Pokud skončí M_a a přijme, M přijme také. Pokud M_b skončí přijetím, M skončí odmítnutím. $L(M)$ je zřejmě rovno L a M se vždy zastaví. Nejjednodušší je uvažovat stroj M jako dvoupáskový, přičemž na každé pásce je simulován jeden stroj. To si můžeme dovolit proto, že libovolný k-páskový TS lze převést na ekvivalentní jednopáskový dle věty 2.2.7. Protože M je stroj, který se vždy zastaví a přijímá L, dostáváme, že L je rekurzivní jazyk. Všimněte si, že ze téhož důvodu je i $\overline{L}$ rekurzivní jazyk.

Formální popis stroje M je pouze technickou záležitostí, nicméně pro úplnost si jej uvedeme. Ukážeme si na tom, jak podobné úpravy Turingova stroje na jiný vypadají. Předpokládejme, že $M_a = (Q_a, \Sigma, \delta_a, q_0^a, F_a = \{q_1^a\})$ a $M_b = (Q_b, \Sigma, \delta_b, q_0^b, F_b = \{q_1^b\})$, bez újmy na obecnosti tedy předpokládáme, že každý z těchto strojů má právě jeden přijímající stav a navíc že oba stroje mají shodnou páskovou abecedu (v opačném případě by stačilo uvážit sjednocení jejich páskových abeced). Popíšeme $M = (Q, \Sigma, \delta, q_0, F)$, kde $Q = Q_a \times Q_b \cup \{q_0, q_1, q_2, q_3\}$ a

$F = \{q_1\}$. Předpokládejme, že M má dvě pásky, přičemž první z nich odpovídá M_a a druhá M_b, a popišme definici přechodové funkce δ.

Na počátku své práce musí M okopírovat vstup z první pásky, na které jej očekává, i na druhou pásku pro stroj M_b. K tomu poslouží instrukce, ty, které jsou parametrizované znakem $x \in \Sigma$, přidáme do instrukční sady ve všech kopiích pro každý znak x zvlášť.

$$\begin{aligned}
\delta(q_0, x, \lambda) &= (q_2, x, x, R, R) \\
\delta(q_0, \lambda, \lambda) &= ([q_0^a, q_0^b], \lambda, \lambda, N, N) \\
\delta(q_2, x, \lambda) &= (q_2, x, x, R, R) \\
\delta(q_2, \lambda, \lambda) &= (q_3, \lambda, \lambda, L, L) \\
\delta(q_3, x, x) &= (q_3, x, x, L, L) \\
\delta(q_3, \lambda, \lambda) &= ([q_0^a, q_0^b], \lambda, \lambda, R, R)
\end{aligned}$$

Další instrukce zabezpečují paralelní běh strojů M_a a M_b a jsou odvozené od jejich přechodových funkcí.

- Pro každou dvojici instrukcí $\delta_a(q_a, x) = (q_a', x', Z_a)$, $\delta_b(q_b, y) = (q_b', y', Z_b)$, kde $q_a, q_a' \in Q_a$, $q_b, q_b' \in Q_b$, $x, x', y, y' \in \Sigma$ a $Z_a, Z_b \in \{L, N, R\}$, přidáme instrukci:

  $$\delta([q_a, q_b], x, y) = ([q_a', q_b'], x', y', Z_a, Z_b)$$

- Pro každou dvojici instrukcí $\delta_a(q_a, x) = (q_a', x', Z_a)$, $\delta_b(q_b, y) = \perp$, kde $q_a, q_a' \in Q_a$, $q_b \in Q_b$, $x, x', y \in \Sigma$ a $Z_a \in \{L, N, R\}$, přidáme instrukci:

  $$\delta([q_a, q_b], x, y) = \begin{cases} \perp & q_b = q_1^b \\ (q_1, x, y, N, N) & \text{jinak} \end{cases}$$

  V případě, že stroj M_b ukončil svou činnost, pak jsou dvě možnosti, buď přijal, v tom případě vstupní slovo $w \in L(M_b) = \overline{L}$, a tedy $M(w)$ odmítne, pokud M_b slovo w odmítl, pak $w \in L$, a tedy $M(w)$ přijme.

- Podobně pro každou dvojici instrukcí $\delta_a(q_a, x) = \perp$, $\delta_b(q_b, y) = (q_b', y', Z_b)$, kde $q_a \in Q_a$, $q_b, q_b' \in Q_b$, $x, y, y' \in \Sigma$ a $Z_b \in \{L, N, R\}$, přidáme instrukci:

  $$\delta([q_a, q_b], x, y) = \begin{cases} (q_1, x, y, N, N) & q_a = q_1^a \\ \perp & \text{jinak} \end{cases}$$

  V případě, že stroj M_a ukončil svou činnost, pak jsou dvě možnosti, buď přijal, v tom případě vstupní slovo $w \in L(M_a) = L$, a tedy $M(w)$ přijme, pokud M_a slovo w odmítl, pak $w \notin L$, a tedy $M(w)$ odmítne.

- A nakonec vyřešíme případ, kdy by oba stroje skončily současně. Pro každou dvojici instrukcí $\delta_a(q_1^a, x) = \perp$, $\delta_b(q_b, y) = \perp$, kde $x, y \in \Sigma$ a $q_b \in Q_b$ přidáme instrukci:

  $$\delta([q_1^a, q_b], x, y) = (q_1, x, y, N, N)$$

■

Lemma 2.2.10 a 2.2.11 dohromady dávají následující ekvivalenci. K ní se dostaneme ještě jednou ve verzi pro rekurzivní a rekurzivně spočetné množiny.

Věta 2.2.12 (Postova) *Jazyk L je rekurzivní tehdy a jen tehdy, když L i $\overline{L}$ jsou rekurzivně spočetné jazyky.*

K dalším vlastnostem rekurzivních a rekurzivně spočetných jazyků a množin se dostaneme později.

2.2.4 Univerzální Turingův stroj

Vstupem univerzálního TS, který si označíme jako U, bude dvojice $w; x$, kde w je kódem TS M v binární abecedě a x je vstupní binární řetězec nad nímž chceme práci M simulovat. Znak „;“ z abecedy Γ očekává UTS též v binární abecedě, tedy jako „111“ (viz podsekce 2.2.2). UTS bude simulovat práci M ve smyslu, který jsme popsali v poznámce 2.2.6. Zejména tedy výpočet $U(w; x)\downarrow$, právě když $M(x)\downarrow$ a v tom případě $U(w; x)$ vydá i týž výsledek, tj. přijme právě když přijme $M(x)$ a případně na jedné z pásek ponechá konečný stav pásky M po ukončení výpočtu nad x.

UTS popíšeme jako třípáskový, protože je to technicky jednodušší, než popisovat jednopáskový UTS. To si můžeme bez újmy na obecnosti dovolit proto, že každý k-páskový Turingův stroj lze převést na jednopáskový dle věty 2.2.7. Při popisu UTS se budeme nadále odkazovat přímo na řetězce v abecedě Γ místo toho, abychom pracovali s jejich reprezentací v binárním abecedě, nutnost přečtení tří znaků, aby UTS rozpoznal jeden znak v abecedě Γ je snadno řešitelná záležitost, kterou se nebudeme zatěžovat.

Rozdělení funkcí pásek UTS je následující:

1. *Vstupní páska.* Na vstupní pásce je na počátku uveden kód simulovaného stroje M a jeho vstup. Jako oddělovače mezi těmito částmi použijeme znak ';' z abecedy Γ. Na vstupní pásku během výpočtu UTS nezapisuje, pouze ji čte a na závěr na ni přepíše obsah pásky stroje M po ukončení jeho výpočtu.

2. *Pracovní páska M.* Tato páska je v průběhu výpočtu využita k uložení slova na pracovní pásce M. Připomeňme si, že páskovou abecedu stroje M jsme nijak neomezovali, její znaky si na tuto zakódujeme v binární abecedě stejně, jako je tomu v kódu přechodové funkce M. Ať b označuje délku nejdelšího zápisu znaku páskové abecedy v kódu w Turingova stroje M (dá se tedy s jistou rezervou říci, že hodnota b odpovídá $\lceil \log_2 |\Sigma| \rceil$, kde Σ je pásková abeceda simulovaného stroje M. Pak tato páska bude rozdělena do bloků délky b oddělených symbolem „|“. Každý blok bude kódovat obsah políčka pracovní pásky stroje M týmž způsobem, jakým je daný znak zakódovaný v kódu w přechodové funkce stroje M, doplněný nulami na začátku na délku b (jde o binárně zapsané číslo symbolu, jehož hodnotu nuly přidané na začátek nemění). 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 M uložené v řetězci w na 1. pásce.

Nyní můžeme popsat, jak UTS na daném vstupu pracuje.

1. UTS nejprve zkontroluje syntaktickou správnost vstupu. Konkrétně:

   (a) Nalezne oddělující středník, pokud se ve vstupu nenachází, skončí UTS práci odmítnutím.

   (b) Současně s hledáním středníku kontroluje, zda se kód skládá ze správně zapsaných kódů jednotlivých instrukcí oddělených znakem „#“. Pokud tento test kdykoli selže, UTS se zastaví s odmítnutím vstupu. *Připomeňme, že kód instrukce musí mít tvar* $(i)_B|(j)_B|(k)_B|(l)_B|Z$, *kde* $(t)_B$ *označuje binární zápis čísla* t *a* $Z \in \{L, N, R\}$.

   (c) Současně stroj počítá délku nejdelšího bloku pro zápis znaku b, to může učinit tak, že před čtením znaku v kódu instrukce se vrátí na začátek 2. pásky, poté se čtením znaku píše na 2. pásku znaky 0. Na konci po projití všech znaků zůstane na 2. pásce nejdelší posloupnost 0.

(d) Součástí syntaktické kontroly by mohl být i test toho, zda se v kódu nenacházejí dvě instrukce s týmž displejem – tedy vstupní dvojicí (stav, znak), tj. test toho, zda je stroj na vstupu deterministický. My tento test vynecháme, i když by jej bylo lze jednoduše provést s pomocí některé z pracovních pásek a s více průchody kódu w. Chování UTS bude takové, že ve chvíli, kdy pro jednu dvojici (stav, znak) jsou v kódu w zapsány dvě instrukce, použije se ta, která je zapsaná více vlevo.

2. Všimněme si, že testy z předchozího kroku lze provést na jeden průchod (kromě případného testu determinismu) a UTS je skončí na prvním znaku vstupního slova x. Nyní UTS překóduje vstup x na 2. pásku, tedy pracovní pásku M. Přičemž znak 0 zapíše jako 0^b a znak 1 jako $0^{b-1}1$, mezi každé dva takovéto kódy znaků zapíše UTS oddělovač „|“. To může učinit proto, že v kódování abecedy popsaném v sekci 2.2.2 odpovídá znaku $X_0 = '0'$ číslo 0 a znaku $X_1 = '1'$ číslo 1. Počet znaků b na zapsání jednoho čísla kódujícího znak abecedy je na počátku uložen na 2. pásce ve formě řetězce 0^b. Vždy, když potřebuje UTS přiformátovat další blok délky b, formátuje ho na délku totožnou se sousedícím blokem. To může učinit například tak, že si bude označovat naposledy přepsané políčko v předchozím bloku, tj. můžeme přepisovat vždy 0 na $0'$ a 1 na $1'$ nebo si můžeme pamatovat označenou číslici a psát místo ní na chvíli třeba #, to pokud nechceme zvětšovat abecedu proti Γ. Zapsanou značku si posouváme spolu s připisováním 0 do nově vytvářeného bloku.

3. Po překódování vstupu se vrátí UTS na začátek 2. pásky.

4. Na 3. pásku zapíše UTS znak 0, tedy binárně zapsané číslo počátečního stavu q_0 stroje M.

5. Následuje simulace jednotlivého kroku stroje M, která se opakuje, dokud existuje instrukce, kterou lze použít na současnou konfiguraci. Na začátku kroku předpokládáme, že hlava na vstupní pásce je na prvním symbolu kódu stroje M, na 2. pásce, tedy pracovní pásce M, je hlava na začátku symbolu, který je čten strojem M a na 3. stavové pásce je hlava na začátku zakódovaného stavu.

   (a) Nejprve musí UTS najít instrukci, kterou je možné použít, to udělá průchodem kódu w stroje M s tím, že u každé instrukce se porovnává výchozí stav se slovem na stavové pásce a výchozí znak se čteným znakem na pracovní pásce.

   (b) Pokud UTS dojde na konec kódu, aniž by našel vhodnou instrukci, pak simulace M končí.

   (c) Pokud UTS najde instrukci, pak odsimuluje její provedení, to spočívá v následujících krocích.

      i. Nejprve UTS zapíše na stavovou pásku číslo nového stavu.

      ii. Dalším krokem je přepsání políčka na pracovní pásce, znak zapisuje UTS odprava od konce příslušného bloku 2. pásky, přičemž jej zleva doplní nulami do délky b.

      iii. Zbývá vykonat pohyb stroje M, což je už jednoduchá záležitost, pokud by se při pohybu přesunul M nad prázdné nenaformátované políčko $'\lambda'$, je třeba je nahradit kódem prázdného políčka, protože je pevně určen kód $X_2 = '\lambda'$, je kódem prázdného políčka řetězec „$0^{b-2}10$“. (Z toho plyne, že b musí být alespoň 2 i tehdy, když λ není v kódu w zmíněno.) To lze na začátku i na konci páskového slova učinit bez dalšího posunu pásky.

6. Na konci ještě musí UTS provést úklid, na vstupní pásku přepíše obsah pracovní pásky s překódováním zpět do abecedy $\{0, 1\}$, pokud na pracovní pásce jsou i jiné znaky než 0 a 1, pak UTS skončí, to odpovídá tomu, že M nesplňuje podmínku vyžadující, aby jeho výstupní abeceda obsahovala jen znaky 0 a 1. Tento krok je vhodný pro situaci, kdy M provádí nějakou transformaci nebo počítá funkci.

7. Na závěr přečte UTS stav uložený na stavové pásce, pokud je na ní číslo 1, což je číslo přijímajícího stavu q_1, přejde UTS do přijímajícího stavu, v opačném případě skončí v jiném než přijímajícím stavu.

Z konstrukce UTS je zřejmé, že splňuje požadavky na něj kladené. Nejde sice o zcela formální popis, protože jsme nevypisovali instrukce UTS, ale to není ani naším cílem, nám tato úroveň zcela postačí.

2.2.5 Univerzální jazyk a problém zastavení

Univerzálním jazykem nazveme $L_u = L(U)$, kde U označuje UTS.

Věta 2.2.13 *Univerzální jazyk L_u je rekurzivně spočetný, ale není rekurzivní.*

Důkaz : To, že L_u je rekurzivně spočetný jsme ukázali tím, že jsme popsali TS U, který jej přijímá. Zbývá tedy ukázat, že není rekurzivní. Intuitivně je tento fakt zřejmý, pokud je vůbec nějaký rekurzivně spočetný jazyk nerekurzivní, pak to jistě můžeme říci i o L_u, neboť tento jazyk má v sobě zakódované všechny rekurzivně spočetné jazyky. Podle věty 2.2.12 stačí ukázat, že $\overline{L_u}$ není rekurzivně spočetný. Budeme postupovat sporem, předpokládejme tedy, že existuje TS M, který přijímá jazyk $\overline{L_u}$, tedy $\overline{L_u} = L(M)$. Pomocí stroje M zkonstruujeme stroj M', který bude přijímat diagonalizační jazyk L_{DIAG}. Z věty 2.2.9 už víme, že L_{DIAG} není rekurzivně spočetný a dospějeme tím tedy ke sporu. M' se vstupem w bude pracovat následujícím způsobem:

1. Pokud w nekóduje Turingův stroj způsobem, jaký jsme popsali při popisu UTS, pak w odpovídá prázdnému Turingovu stroji, který vždy odmítne a $M'(w)$ tedy přijme.

2. Pokud je w syntakticky správným kódem Turingova stroje, pak M' zavolá $M(w; w)$, tedy Turingův stroj M, jemuž na vstup předá dvojici řetězců w a w. (Stroji M jsou předány na vstupu oddělené znakem „;“ v abecedě Γ, tedy 111 v binárním zápisu.) M' se zachová podle M, tedy přijme, odmítne, nebo se nezastaví podle toho, jak se zachová M.

Není těžké nahlédnout, že takto popsaný stroj M' přijímá jazyk L_{DIAG}, protože $M'(w)$ přijme právě když w je validním kódem TS N a $N(w)$ nepřijme, což je ekvivalentní tomu, že $U(w; w)$ nepřijme a $M(w; w)$ přijme. Fakt, že $L(M') = L_{DIAG}$, je však ve sporu s tím, že L_{DIAG} není rekurzivně spočetný. ■

Technika, kterou jsme zde použili, tedy převod problému L_{DIAG}, o kterém už něco víme, na jiný, tedy $\overline{L_u}$, o kterém chceme něco zjistit, je obvyklou technikou jak v teorii vyčíslitelnosti, tak v teorii složitosti. Později ji zformalizujeme a budeme hojně využívat.

Problém zastavení, tedy Halting problém, je asi nejznámějším algoritmicky nerozhodnutelným problémem. Instancí problému je dvojice (M, x), kde M je Turingův stroj a x je vstupní slovo. Ptáme se na to, jestli se výpočet M nad vstupem x zastaví, tj. jestli $M(x)\downarrow$. V našem kódování lze tento problém zachytit jako otázku náležení do následujícího jazyka:

$$L_{HALT} = \{w; x \mid w \text{ kóduje TS } M \text{ a } M(x)\downarrow\}$$

Přitom za zápisem $w;x$ můžeme vnímat jak zápis kódu v abecedě Γ, tak jeho binární přepis, jak byl popsán v sekci 2.2.2. Možností je však pochopitelně více a nezáleží příliš na tom, jak provedeme zakódování tohoto problému do jazyka. Ukážeme si, že problém zastavení, tedy jazyk L_{HALT}, je sice rekurzivně spočetný, ale není rekurzivní. Uvědomme si, že tuto práci jsme již ve skutečnosti učinili ve větě 2.2.13. Rozdíl mezi univerzálním jazykem a problémem zastavení je totiž jen v jednom detailu. My jsme definovali přijetí slova Turingovým strojem pomocí přijímajícího stavu. Pokud bychom definovali přijetí slova zastavením, tj. slovo x by bylo strojem M přijato, právě když $M(x)\downarrow$, pak by pojmy univerzálního jazyka a problému zastavení splynuly. Není tedy překvapivé, že důkaz věty 2.2.14 je analogický důkazu věty 2.2.13.

Věta 2.2.14 *Problém zastavení je rekurzivně spočetný, ale není rekurzivní. Přesněji, jazyk L_{HALT} je rekurzivně spočetný, ale není rekurzivní.*

Důkaz : Fakt, že L_{HALT} je rekurzivně spočetný je zřejmý z existence univerzálního Turingova stroje. Slovo $w;x$ patří do L_{HALT}, právě když $U(w;x)\downarrow$, Diagonalizací ukážeme, že doplněk jazyka L_{HALT}, tedy $\overline{L_{HALT}}$ není rekurzivně spočetný, čímž bude důkaz dokončen díky větě 2.2.12. Předpokládejme pro spor, že existuje stroj M, který přijímá jazyk $\overline{L_{HALT}}$, tedy $\overline{L_{HALT}} = L(M)$. Nyní definujme jazyk $L = \{w_i \in \{0,1\}^* \mid M_i(w_i)\uparrow\}$, jde vlastně o diagonálu $\overline{L_{HALT}}$. S pomocí stroje M sestrojíme nyní stroj M_e (půjde tedy o e-tý TS) přijímající jazyk L. Po obdržení řetězce w_i, zkontroluje M_e, zda w_i je syntakticky správným kódem Turingova stroje a pokud ano, spustí $M(w_i;w_i)$. Pokud $M(w_i;w_i)$ přijme, pak přijme i $M_e(w_i)$, v opačném případě se $M_e(w_i)$ zacyklí a nezastaví se, tj. $M_e(w_i)\uparrow$. To učiní i v případě, zjistí-li, že w_i nekóduje syntakticky správně Turingův stroj a odpovídá tedy prázdnému TS (to je zde nutné odlišit jen proto, že kdyby w_i obsahovalo oddělovač ';', pak by $w_i;w_i$ mohlo odpovídat výpočtu jiného stroje nad jiným vstupem). Takto zkonstruovaný stroj M_e má tu vlastnost, že přijme svůj vstup právě když se zastaví. Zřejmě platí, že $L(M_e) = L$. Nyní se podívejme, jestli w_e, tedy kód M_e patří do L, nebo ne.

1. Pokud $w_e \in L$, znamená to, že se $M_e(w_e)$ zastaví a přijme, protože $L = L(M_e)$, to ale současně znamená, že $M_e(w_e)\uparrow$ podle definice L, což je pochopitelně ve sporu.

2. Nyní předpokládejme, že $w_e \notin L$, ale podle definice L to znamená, že $M_e(w_e)\downarrow$. Podle toho, jak jsme si popsali M_e, znamená to, že přijme slovo w_e, z toho dostaneme $w_e \in L(M_e) = L$, což je však ve sporu s předpokladem.

Jazyk L použitý v důkazu není nic jiného než L_{DIAG}, pokud bychom předpokládali, že TS přijímají zastavením v jakémkoli stavu. ■

2.2.6 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 rozpoznává 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 rozpoznává 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 rozpoznává 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), $f(x, y) = x$ div y (celočíselné dělení), $f(x, y) = x$ mod y (zbytek po celočíselném dělení).

5. Popište, jak by pracoval Turingův stroj rozpoznávající jazyk

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

6. Popište, jak by pracoval Turingův stroj rozpoznávající 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 2.2.7.*

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 rozpoznávají jednopáskové Turingovy stroje, které nesmí pohybovat hlavou vpravo (tj. hlava může zůstat stát nebo se pohnout vlevo).

17. Rozmyslete si, jakou třídu jazyků rozpoznávají 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 rozpoznává 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. Tj. instrukce v případě tří zásobníků může vypadat třeba 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 $\lhd$ 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.

Rekurzivní a rekurzivně spočetné jazyky

22. Ukažte, že rekurzivní 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 $L^* = \{x_1x_2...x_k \mid (k \geq 0) \wedge \bigwedge_{i=1}^{k}(x_i \in L)\}$*).

23. Ukažte, že rekurzivně spočetné jazyky jsou uzavřeny na operace sjednocení, průnik, konkatenace a hvězdička, nejsou však uzavřeny na doplněk.

Diagonalizace a nerozhodnutelné problémy

24. 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 L_{DIAG}, hledejte jinou nekonečnou množinu prvků této matice, která by mohla posloužit podobně jako diagonála v případě L_{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*

25. Definujme problém použití stavu následovně:

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).

2.3 Částečně rekurzivní funkce

Dalším výpočetním modelem, který zde zmíníme, jsou částečně rekurzivní funkce. Jejich definice pochází už z třicátých let, primitivně rekurzivní funkce použil už Gödel pro důkaz vět o neúplnosti, později byly zavedeny obecně a částečně rekurzivní funkce pracemi Herbranda, Gödela a Kleeneho.

2.3.1 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 definované nad přirozenými čísly a jejich výstupem jsou opět přirozená čísla (včetně nuly).

Primitivně rekurzivní funkce

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

Okamžitě vidíme, že základních funkcí je nekonečně mnoho, a to díky funkcím I_n^j. 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 h na n proměnných, 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 h na n proměnných, pro niž platí:

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

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 substituce a primitivní rekurze.

Poznámka 2.3.1 *U substituce je okamžitě vidět, že odpovídá volání již odvozené funkce, tedy podprogramu. Zastavme se však u operátoru primitivní rekurze. 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 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:

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

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

function $h(x_1, x_2)$

```
1: begin
2:    z := f(x_2)
3:    for y := 0 to x_1 − 1
4:    do
5:       z := g(y, z, x_2)
6:    done
7:    return z
8: end
```

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 do následujícího tvrzení.

Lemma 2.3.2 *Následující funkce jsou primitivně rekurzivní: Konstanta $c \in \mathbb{N}$, $x + y$, $x \cdot y$, x^y, $x!$, $|x - y|$, $\min\{x, y\}$, $\max\{x, y\}$, $x \dot{-} y$, $\mathrm{sign}(x)$ (0 pokud je $x = 0$, 1, pokud je $x > 0$), x div y (celočíselné dělení), x mod 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), $\mathrm{parita}(x)$ (vrací 1, je-li x liché číslo, a 0, je-li x sudé číslo).*

Důkaz : Důkazy jsou většinou jednoduché a některá odvození si zkusíme probrat na cvičení. ■

My si ukážeme na příklad odvození funkce sčítání.

Příklad 2.3.3 *Ukážme si na odvození funkce $h(a, b) = a + b$. Máme-li k dispozici jen přičítání jedničky, můžeme k výpočtu použít následující for cyklus, v němž b-krát přičteme jedničku k a:*

function $h(a, b)$

```
1: begin
2:    z := b
3:    for y := 0 to a − 1
4:    do
5:       z := z + 1
6:    done
7:    return z
8: end
```

Z poznámky 2.3.1 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 $h = R_2(f, g)$. 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, to můžeme ve zkratce zapsat jako:

$$h := R_2(I_1^1, S_3^1(s, I_3^2))$$

Z toho je již jasné, jakým způsobem dostat posloupnost funkcí odvozující h, ta by tedy vypadala takto (zde $h = f_5$):

$f_1 = s(x)$ *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)*
$f_5 = R_2(f_2, f_4)$ *primitivní rekurze (V)*
tj. $f_5(0, x_2) = f_2(x_2) = x_2$
a $f_5(x_1 + 1, x_2) = f_4(x_1, f_5(x_1, x_2), x_2) = f_5(x_1, x_2) + 1$

V dalším textu si však 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 dospět k prostředku stejně silnému jako Turingův stroj, a to primitivně rekurzivní funkce nejsou. Pokusme se zamyslet nad tím, proč tomu tak je. Způsob, jakým jsme definovali odvození funkce nám umožňuje zapsat odvození řetězcem v pevné konečné abecedě. Tyto řetězce můžeme uspořádat a očíslovat podobně, jako jsme to již udělali s binárními řetězci, potažmo s řetězci v abecedě Γ v případě Turingových strojů. Naopak každé odvození jednoznačně určuje primitivně rekurzivní funkci, proto dostaneme očíslování primitivně rekurzivních funkcí $f_1, f_2, \ldots$. Vzhledem k tomu, že už základních funkcí je nekonečně mnoho, i PRF bude nekonečně (ale stále spočetně) mnoho. Nyní se podívejme na funkci danou následujícím předpisem:

$$f(x) = f_x(x) + 1$$

Zřejmě nemůže jít o PRF, protože s každou PRF se liší alespoň na jednom vstupu, jmenovitě s x-tou funkcí f_x se liší na vstupu x, jde o podobný diagonalizační argument jako při definici jazyka L_{DIAG} a důkazu toho, že tento jazyk není rekurzivně spočetný Pokud jsme však k číslu x schopni efektivně dostat odvození funkce f_x, což při vhodném kódování jistě můžeme a pokud jsme schopni navíc efektivně spočítat hodnotu primitivně rekurzivní funkce f_x, což opět není těžké nahlédnout, navíc tak definujeme v této chvíli náš výpočetní model (k převodu z PRF/ČRF na TS se ale ještě budeme věnovat), tak by bylo poněkud zvláštní, kdybychom nemohli určit hodnotu $f_x(x) + 1$. Algoritmus pro výpočet by byl prostý: Spočti $f_x(x)$ a přičti jedna. Navíc pokud by měly PRF odpovídat intuitivnímu pojmu algoritmu, tak proč by náš algoritmus pro výpočet $f_x(x) + 1$ neměl být intuitivní? Z argumentů v tomto odstavci také plyne, že univerzální funkce $g(x, y) \simeq f_x(y)$ nemůže být primitivně rekurzivní. To znamená, že model PRF je příliš slabý na to, aby splňoval Churchovu-Turingovu tezi, protože v něm nelze implementovat intuitivní algoritmus pro výpočet hodnoty PRF.

Co nám přitom v tuto chvíli svazuje ruce, je možná trochu paradoxně fakt, že PRF jsou všude definované funkce (základní funkce zřejmě jsou definované všude a ani substituce ani primitivní rekurze na tom nemohou nic změnit). Kdybychom měli dovoleno, aby f_x nebyla pro nějaké vstupy definovaná, nedospěli bychom nutně ke sporu. Například pokud by z bylo číslem funkce $f_z(x) = f_x(x)+1$, nemohla by být hodnote $f_z(z)$ definovaná, aby to dávalo smysl. Podobně jsme to viděli již dříve u diagonalizace a rozdílu mezi rekurzivními a rekurzivně spočetnými jazyky. Co nám navíc oproti intuitivnímu pojmu algoritmu chybí, je obecný *while* cyklus, 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:

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

$$h(x_1, x_2, \ldots, x_n) \simeq \min\{y \mid f(x_1, x_2, \ldots, x_n, y)\downarrow = 0 \\ \text{a } (\forall z \leq y)[f(x_1, x_2, \ldots, x_n, z)\downarrow]\}.$$

Tedy h nabývá nejmenší hodnoty y, pro níž je f definováno a rovno 0. Navíc požadujeme, aby pro všechny hodnoty nižší než y byla hodnota f definována. Pro operátor minimalizace budeme používat následující značení:

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

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.

Uvědomme si, že $\mu y[f(x_1, x_2, \ldots, x_n, y) \simeq 0]$ obecně neznamená „nejmenší číslo y, pro které je $f(x_1, x_2, \ldots, x_n, y) \simeq 0$“, a to díky druhé podmínce v definici minimalizačního operátoru. Podmínka požadující, aby byla hodnota $f(x_1, x_2, \ldots, x_n, z)$ definována pro všechna $z \leq y$, je navíc velmi důležitá, protože odpovídá tomu, co intuitivně považujeme za efektivní minimalizaci. Představme si, jak bychom počítali hodnotu funkce h, prostě bychom počítali hodnoty $f(x_1, \ldots, x_n, 0)$, $f(x_1, \ldots, x_n, 1)$, ..., až bychom nalezli první y, pro něž by bylo $f(x_1, \ldots, x_n, y) = 0$, tento postup můžeme zachytit pomocí následujícího *while* cyklu:

function $h(x_1, \ldots, x_n)$

```
1: begin
2:    y := 0
3:    while f(x_1, x_2, ..., x_n, y) ≠ 0
4:    do
5:       y := y + 1
6:    done
7:    return y
8: end
```

V tomto algoritmu jistě platí, že Pokud jedna z hodnot $z \leq y$ není definována nebo pokud f je sice definována všude, ale nikde nenabývá 0, hodnota funkce h pro daný vstup není definována. Druhý důvod, proč je druhá podmínka důležitá, je ten, že pokud bychom definovali funkci h danou 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 by funkce f v operátoru minimalizace byla obecně rekurzivní, a tedy všude definovaná, pak je požadavek na definovatelnost splněn automaticky a jde skutečně o hledání nejmenší hodnoty y, pro kterou $f(x_1, \ldots, x_n, y)$ nabude hodnoty 0.

Na druhou stranu později ukážeme, že omezená minimalizace je primitivně rekurzivní. Konkrétně mějme primitivně rekurzivní funkci f na 2 proměnných, potom funkce h definovaná jako

$$h(x,z) = \begin{cases} \min\{y < z \mid f(x,y) = 0\} & \text{pokud takové } y \text{ existuje} \\ z & \text{jinak} \end{cases}$$

je primitivně rekurzivní. Důvod je ten, že pro výpočet h nám stačí *for* cyklus, a tedy primitivní rekurze, nepotřebujeme *while* a minimalizaci. Toto tvrzení si ukážeme v o něco obecnější podobě jako lemma 2.3.12.

Zatímco PRF jsou všude definované, zavedením minimalizace jsme tuto vlastnost pro ČRF ztratili, proto jsme zavedli i ORF. Ne každá obecně rekurzivní funkce je však i primitivně rekurzivní, důvod jsme viděli u diskuze o $f_x(x) + 1$, univerzální funkce pro PRF je sice totální, tedy ORF, není však primitivně rekurzivní. Dalším známým příkladem funkce, která je sice obecně rekurzivní, ale není primitivně rekurzivní, je Ackermannova funkce. Fakt, že je daná funkce všude definovaná tedy ještě neznamená, že se při jejím výpočtu obejdeme bez *while* cyklu. Podobně každá ORF je i ČRF, ale ne každá ČRF je všude definovaná, tedy ORF, například funkce $\lambda x\Big[\mu y[s(y) \simeq 0]\Big]$ je jistě ČRF, která není definovaná pro žádné x.

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.

2.3.2 Rekurzivní a rekurzivně spočetné predikáty, vlastnosti PRF, ORF a ČRF

Upřesněme si nejprve samotný pojem predikátu či relace.

Definice 2.3.4 Množině n-tic $R \subseteq \mathbb{N}^n$, $n \geq 1$ budeme říkat *predikát* nebo *relace* a fakt, že $(x_1, \ldots, x_n) \in R$ budeme též označovat pomocí $R(x_1, \ldots, x_n)$. *Charakteristickou funkcí* predikátu R je funkce $\chi_R(x_1, \ldots, x_n)$, pro kterou platí, že

$$\chi_R(x_1, \ldots, x_n) \simeq \begin{cases} 1 & R(x_1, \ldots, x_n) \\ 0 & \text{jinak} \end{cases}$$

Nyní si můžeme říci, co míníme pojmem rekurzivní či rekurzivně spočetný predikát.

Definice 2.3.5 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, pro kterou platí, že

$$(\forall (x_1, \ldots, x_n) \in \mathbb{N}^n) \Big[f_R(x_1, \ldots, x_n)\downarrow \Leftrightarrow R(x_1, \ldots, x_n) \Big],$$

to jest, hodnota funkce f_R je pro danou n-tici definovaná právě když je na ní predikát R splněný. Hodnota funkce v tomto případě není důležitá. Takové funkci budeme také říkat částečná charakteristická funkce.

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 $prime(x)$.

Poznamenejme, že jako podmínku jsme v operátoru minimalizace, tedy *while* cyklu, povolili pouze test, zda je hodnota vložené funkce $f(x_1, \ldots, x_n)$ definována a rovna 0. Ve skutečnosti je však možno připustit libovolný 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 druhá podmínka v definici efektivní minimalizace automaticky splněna a tento zápis tedy skutečně znamená „nejmenší y, pro něž je predikát $R(x_1, \ldots, x_n, y)$ splněn“. Podobně lze minimalizaci rozšířit i na podmínky, které nejsou nutně obecně rekurzivní, například test, zda je daná funkce definovaná a její hodnota nabývá jakékoli hodnoty, ne nutně 0.

Rekurzivní a rekurzivně spočetné predikáty a množiny jsou ve skutečnosti jen nové názvy pro nám již známé rekurzivní a rekurzivně spočetné jazyky, vzhledem k tomu, že jsme popsali bijekci mezi binárními řetězci a kladnými čísly, odpovídají si vzájemně množiny čísel s jazyky jako množinami řetězců. K tomu, že pojmy rekurzivní jazyk a rekurzivní množina jsou ekvivalentní, však musíme samozřejmě ukázat, že ČRF a TS jsou stejně silné výpočetní modely. Než se do toho budeme moci pustit, zmíníme si však 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.

Lemma 2.3.6 (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.

function $g(z, x)$

1: **begin**
2: $soucet := 0$
3: **for** $y := 0$ **to** $z - 1$
4: **do**
5: $soucet := soucet + f(y, x)$
6: **done**
7: **return** *soucet*
8: **end**

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 2.3.1, 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, mohli bychom postupovat stejně, ale nemůžeme použít pouze primitivní rekurzi, neboť ta jak víme vytvoří funkci nejméně dvou proměnných a my chceme, aby g měla jednu proměnnou. Můžeme však zavést umělou proměnnou, jejíž hodnotu nikde nepoužijeme, přesněji, podle lemmatu 2.3.6 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 2.3.7 *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 dá k dispozici podmíněný příkaz.

Lemma 2.3.8 (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 definovaná následujícím předpisem je PRF:*

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

Důkaz: Funkci f můžeme napsat jako

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

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 2.3.2 opět k primitivně rekurzivní funkci, je f primitivně rekurzivní funkce. ∎

Lemma 2.3.8 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 2.3.8, 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 2.3.9 (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 2.3.6 a lemmatu 2.3.2. ■

Situace s neomezenými kvantifikátory je poněkud komplikovanější a budeme se jí ještě věnovat v části o rekurzivních a rekurzivně spočetných množinách. 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 2.3.10 (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 zřejmě 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 ještě vrátíme později v části věnované rekurzivním a rekurzivně spočetným množinám.

Lemma 2.3.11 (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 2.3.9. ■

Lemma 2.3.12 (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*:

function $f(x, z)$

```
 1: begin
 2:   u := 0 {V u si budeme pamatovat návratovou hodnotu}
 3:   for y := 0 to z − 1
 4:   do
 5:     if P(x, y)
 6:     then
 7:       break
 8:     endif
 9:     u := u + 1
10:   done
11:   return u
12: end
```

Cyklus *for* odpovídá primitivní rekurzi a podmíněný příkaz *if* můžeme použít díky lemmatu 2.3.8. Co však nemáme k dispozici, je příkaz *break* pro vyskočení z cyklu ven. Musíme tedy nechat cyklus doběhnout až do konce, přičemž když narazíme na hodnotu y, pro kterou je predikát $P(x, y)$ splněn, musíme si ji zapamatovat a dále ji neměnit. To můžeme zabezpečit následujícím cyklem:

function $f(x, z)$

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

V cyklu zvyšujeme hodnotu u do chvíle, kdy není splněno $P(x, u)$. Ve chvíli, kdy je poprvé $P(x, u)$ splněno, ponecháme u beze změny a protože od té doby bude $P(x, u)$ platit stále, na konci zůstane u nejmenší hodnota, pro kterou bylo $P(x, u)$ splněno. 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[2], podle lemmatu 2.3.8 a toho, že $P(x, u)$ i $\neg P(x, u)$ jsou primitivně rekurzivní predikáty, první jmenovaný dle předpokladu, druhý dle lemmatu 2.3.10. Výslednou funkci f odvodíme primitivní rekurzí z nulové funkce a funkce g, tedy $f = R_2(o, g)$. ■

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

Lemmata 2.3.6, 2.3.8, 2.3.12, 2.3.9, 2.3.10 a 2.3.11 lze přeformulovat i pro ORF a ORP, pro ČRF a RSP platí rovněž téměř všechna. Výjimku tvoří negace predikátu v lemmatu 2.3.10, jak jsme již zmínili v komentáři za tímto tvrzením.

Na závěr poznamenejme, že odvození ČRF lze zakódovat do binárního řetězce, kterému potom opět jednoznačně odpovídá číslo, i ČRF lze tedy očíslovat stejně jako Turingovy stroje. Kódováním ČRF se nebudeme hlouběji zabývat, je pouze technicky náročné a nepřináší nic myšlenkově nového ve chvíli, kdy jsme již viděli, jak zakódovat Turingův stroj. Ponecháme tedy na čtenáři rozmyslet si, jak by bylo lze kódování provést, aby s ním bylo možno efektivně pracovat. My budeme v dalším textu prostě předpokládat, že to lze a že máme nějaké vhodné a pevné kódování k dispozici. Prozatím nebudeme fixovat žádné očíslování ČRF, které by mohlo tímto kódováním vzniknout. Důvod je ten, že hned v další sekci dospějeme k tomu, že pojem ČRF splývá s pojmem Turingovsky vyčíslitelné funkce, posléze převezmeme očíslování Turingových strojů i pro ČRF, abychom mohli využívat toho, že e-tá ČRF je počítána e-tým Turingovým strojem M_e. Gödelovo číslo tak bude určovat algoritmus nezávisle na modelu, který budeme používat.

2.3.3 Ekvivalence ČRF a Turingových strojů

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 2.3.13

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, takový stroj byl popsán v příkladu 2.2.4.

(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 2.2.7 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:

function $h(x_1, x_2, \ldots, x_n)$

```
1: begin
2:    z := f(x_2, ..., x_n)
3:    for y := 0 to x_1 - 1
4:    do
5:       z := g(y, z, x_2, ..., x_n)
6:    done
7:    return z
8: end
```

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:

function $h(x_1, \ldots, x_n)$

```
1: begin
2:    y := 0
3:    while f(x_1, x_2, ..., x_n, y) ≠ 0
4:    do
5:       y := y + 1
6:    done
7:    return y
8: end
```

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 sekci 2.2.2. 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 sekci 2.2.2.

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 sekci 2.2.2.

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

Lemma 2.3.14 *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 2.3.2. 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 2.3.15 *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:

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

- 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 2.3.14, s těmito funkcemi už dokážeme implementovat výše uvedené primitivně rekurzivní funkce a predikáty. Konečná konjunkce je primitivně rekurzivní dle lemmatu 2.3.11. ■

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

Věta 2.3.16 *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 2.2.3 existuje stroj M_e, který ji počítá. 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 2.3.13 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 2.3.17 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 2.3.18 (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 2.3.16. ■

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 2.3.19 *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 halting problému 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 2.3.20 (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 rozhodnutí toho, zda je funkce primitivně rekurzivní funkcí vyplývá z Riceovy věty 3.4.6, 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 2.3.21 *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 2.3.22 (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 bud,e 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. Svým způsobem však tento princip použijeme i ve chvíli, kdy si napíšeme funkci, volající jinou s pevnými parametry. Například v jazyku C máme funkce getchar(void) a getc(FILE *). Překladač za nás spočítá kód nové funkce, tedy její Gödelovo číslo, to že je to možné, právě ukazuje uvedená s-m-n věta.

Příklad 2.3.23 *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. Tj. v tomto případě funkce $f(k) \simeq s_1^1(e, k)$ je 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 2.3.24 *Uvažme funkci $\varphi_e(x,y,u) \simeq \varphi_x(u)+\varphi_y(u)$, kterou jsme již použili v poznámce 2.3.21 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

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

Pokud bychom do f dosadili za oba parametry třeba Gödelovo číslo funkce I_1^1, dostaneme funkci pro vynásobení u dvěma.

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.

2.3.4 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 2.3.3) 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)

   (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 div y (celočíselné dělení)

   (g) x mod 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 číslování rekurzivně spočetných množin zavedené dále v definici 3.1.2:

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

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 2.3.24. 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

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

2.4 Random Access Machine

2.4.1 Definice

V této kapitole si 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 i prostorové složitosti algoritmů. Tento model byl zaveden v článku [3] právě pro potřeby měření časové a prostorové složitosti algoritmů, cílem bylo vytvořit model, který by se co nejvíce blížil reálným počítačům. Budeme zde sledovat (byť s jistými odchylkami) přístup z tohoto původního zdroje, i když je možno najít i celou řadu jiných definic, lišících se zejména v použitých instrukcích a jejich zápisu.

Podobně jako Turingovy stroje, i RAM je strojem s oddělenou pamětí pro data a pro instrukce, nejedná se tedy o stroje Von Neumannovy architektury[3]. Program pro RAM 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

[3] K RAMu však existuje i verze RASP (*random stored program*, v níž program i data sdílí týž paměťový prostor, instrukční sada dokonce počítá s tím, že program přepisuje sám sebe, aby se dostal k registrům, na které se odkazuje nepřímo.

čísly počínaje nulou, těmto číslům budeme říkat adresy. Obsahem registru může být libovolně velké přirozené číslo[4]. Při popisu instrukcí budeme dodržovat následující konvence:

- Obsah registru r_i budeme označovat 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 $LOAD(r_i, C)$, která do registru r_i uloží hodnotu C.

V programu pro RAM máme k dispozici instrukce zobrazené v tabulce 2.1.

Instrukce	**Efekt**	**Popis**
LOAD(C, r_i)	$r_i \leftarrow C$	Přiřadí do registru r_i konstantu $C \in \mathbb{N}$.
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)	$r_k \leftarrow [r_i] \dot{-} [r_j]$	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. Pomocí $\dot{-}$ označujeme operaci opatrného odčítání, což znamená, že v případě, kdy $[r_i] < [r_j]$, uloží se do registru r_k hodnota 0.
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 s číslem z, pokud je hodnota $r_i = 0$, pak program dále pokračuje následující instrukcí (po **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.

Tabulka 2.1: Seznam instrukcí RAM

Sada instrukcí popsaná v tabulce 2.1 zdaleka není jediná možná. Objevují se varianty, které neumožňují obecné sčítání a odčítání, ale pouze přičtení jedničky (*increment*) a odečtení

[4]Fakt, že se omezujeme na přirozená čísla, není nijak podstatný, klidně bychom mohli povolit libovolná celá čísla, protože se dále budeme zabývat množinami přirozených čísel, omezíme se zde však také pouze na přirozená čísla. Na druhou stranu v případě potřeby není velký problém reprezentovat i záporná čísla s pomocí přirozených čísel.

jpedničky (*decrement*). Další varianty připouštějí aritmetické operace nebo nepřímou adresaci pouze s použitím zvlášť pro to vyhrazeného registru, kterému se říká akumulátor. 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. K tomu se budeme podrobněji věnovat v části III.

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 bychom nebyli schopni přistoupit k registrům s vyššími adresami, než je třeba nejvyšší adresa zmíněná v programu.

Předpokládáme, že RAM je předán vstup 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.

Na začátku výpočtu mají všechny registry hodnotu 0, výpočet začíná první instrukcí I_0 a pokračuje dalšími instrukcemi v pořadí. V případě použití instrukce **JNZ** může dojít ke skoku na jinou instrukci, než je další v pořadí. Výpočet končí v případě, kdy měla být instrukce za koncem programu, 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.

Nechť $f : \mathbb{N}^n \mapsto \mathbb{N}$ 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 $f(x_1, \ldots, x_n)$

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

Příklad 2.4.1 *Ukažme si například RAM, který počítá funkci násobení $f(x_1, x_2) = x_1 \cdot x_2$. Program tohoto stroje je popsán v algoritmu 1. 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.*

Algoritmus 1 Program pro RAM, který počítá funkci násobení $f(x_1, x_2) = x_1 \cdot x_2$.

```
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_O] > 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
```

Poznámka 2.4.2 (Zjednodušení použití některých instrukcí) *Na příkladu 2.4.1 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 sčítanců, tedy na obsahu odpovídajících registrů. Prozatím si však zavedeme konvenci, že v operacích* **ADD** *a* **SUB** *připustíme použití konstant jako sčítanců, menšitelů a menšenců (číselnou konstantu od registru vždy odlišíme, neboť registry označujeme pomocí r_i). Podobně omezující je, že nemáme k dispozici nepodmíněný skok, opět to lze obejít s pomocným registrem a uložením nenulové hodnoty do tohoto registru, i zde si dovolíme jako parametru v* **JNZ** *místo registru použít i číselnou konstantu 1 jako nenulové číslo, tedy* **JNZ**$(1, 10)$ *je nepodmíněným skokem na řádku 10. Podobně v operaci* **PRINT** *povolíme zápis konstanty.*

Popíšeme si ještě, co znamená, že RAM R přijímá jazyk L. 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 RAM R předáme na vstup jako posloupnost $i_1, \ldots, i_n$, protože indexy symbolů jsou nenulové, první nula přečtená instrukcí **READ** ze vstupu označuje konec řetězce. Výpočet RAM s takto předaným vstupem budeme zapisovat prostě pomocí $R(w)$.

- Řekneme, že RAM R přijímá slovo $w \in \Sigma^*$, pokud výpočet $R(w)\downarrow$, 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).

Přidat příklad řetězcové funkce $f(w) = w$, tam bude vidět použití COPY

Příklad 2.4.3 *Algoritmus 2 popisuje program pro RAM, který rozhoduje jazyk $\{1^n 2^n \mid n \geq 0\}$. Vzhledem k přímému přístupu do paměti máme zde mnohem jednodušší úlohu, než v případě TS (viz příklad 2.2.2). Pro lepší čitelnost používáme symbolicky zapsaná návěští. Algoritmus pracuje na*

Algoritmus 2 Program pro RAM, který rozhoduje jazyk $\{1^n 2^n \mid n \geq 0\}$.

```
1:  function RozhodniRAM(w)
2:  start:
3:      READ(r_0)                          ▷ Načti další znak ze vstupu to registru r_0.
4:      JNZ(r_0, next_char)                ▷ Pokud ještě není konec vstupu, jdi na next_char.
5:                  ▷ V opačném případě zkontrolujeme, jestli se počet 0 shoduje s počtem 1.
6:      SUB(r_1, r_2, r_3)                 ▷ r_3 ← [r_1] ∸ [r_2], v r_3 je 0 pokud [r_2] ≥ [r_1].
7:      JNZ(r_3, reject)       ▷ [r_3] > 0, pokud [r_2] < [r_1], tedy dvojek je méně než jedniček.
8:      SUB(r_2, r_1, r_3)                 ▷ r_3 ← [r_2] ∸ [r_1], v r_3 je 0 pokud [r_1] ≥ [r_2].
9:      JNZ(r_3, reject)       ▷ [r_3] > 0, pokud [r_1] < [r_2], tedy jedniček je méně než dvojek.
10:     JNZ(1, accept)             ▷ Jedniček a dvojek je týž počet, skoč na příkaz přijetí.
11: next_char:                                 ▷ Kontrola, jde-li o 1 nebo 2.
12:     SUB(r_0, 1, r_0)                       ▷ r_0 ← [r_0] ∸ 1
13:     JNZ(r_0, two)          ▷ Pokud v r_0 zbývá nenulové číslo, vstupní znak nebyl 1.
14: one:                                       ▷ Přečtený znak byla 1.
15:     JNZ(r_2, reject)                       ▷ Pokud už byla přečtena 2, odmítni.
16:     ADD(r_1, 1, r_1)                       ▷ Zvyš počet načtených 1.
17:     JNZ(1, start)                          ▷ A skoč zpátky na začátek.
18: two:                          ▷ Zkontroluj, jestli číslo přečtené ze vstupu byla 2.
19:     SUB(r_0, 1, r_0)                       ▷ r_0 ← [r_0] ∸ 1
20:     JNZ(r_0, reject)       ▷ Pokud v [r_0] > 0, pak vstupní znak nebyl 1 ani 2, odmítni.
21:     ADD(r_2, 1, r_2)                       ▷ Zvyš počet načtených dvojek.
22:     JNZ(1, start)                          ▷ Zpět na začátek.
23: reject:
24:     PRINT(0)                   ▷ Odmítni zapsáním 0 jako prvního znaku.
25: accept:
26:     PRINT(1)                                                ▷ Přijmi.
27: end function
```

jednoduchém principu, dokud jsou na vstupu nenulová čísla, zpracovává jedno po druhém. Pokud je načtené číslo 1*, zkontroluje algoritmus, zda již byla načtena nějaká* 2*, pokud ano odmítne, jinak zvýší počet načtených jedniček, který si průběžně ukládá v registru* r_1*. Pokud je načtené číslo* 2*, zvýší počet načtených dvojek v registru* r_2*. V případě, že byl již načten celý vstup, dojde ke kontrole toho, zda počet* 1 *je shodný s počtem* 2*.*

Z popisu RAMu vidíme, že programy pro RAM jsou imperativní, jednotlivé instrukce popisují, jaký krok se má provést, a provádějí se sekvenčně po sobě nezávisle na dané konfiguraci RAMu. To je rozdíl oproti Turingovým strojům, kde je další instrukce k provedení vybrána podle toho, jaké symboly jsou aktuálně čteny hlavami na páskách a v jakém stavu se Turingův stroj nachází. Je to rozdíl i proti částečně rekurzivním funkcím, jejichž výpočet je dán odvozením dané funkce.

2.4.2 Programování RAM

2.4.3 Ekvivalence RAM a Turingových strojů

2.4.4 Varianty RAM

K modelu RAM byl v [3] zaveden i model RASP (*random access stored program*), který má v jediném paměťovém prostoru uložen jak program, tak i data, nad kterými tento program pracuje.

Paralelní RAM (PRAM) je velmi populární při studiu paralelních algoritmů.

2.5 Algoritmicky vyčíslitelné funkce

- Sjednocují turingovsky vyčíslitelné, částečně rekurzivní a RAM-vyčíslitelné funkce.
- Hlavně komentáře.
- Zavést číslování a značení.
- Přesunout sem s-m-n větu a ukázat bez pomoci ČRF.

Kapitola 3

(Ne)rozhodnutelnost

3.1 Definice a nástroje

V této kapitole se budeme věnovat vlastnostem rekurzivních a rekurzivně spočetných množin. Ačkoli tyto pojmy jsme již definovali, neboť jde o unární rekurzivní a rekurzivně spočetné predikáty, zopakujme si ještě jednou jejich definici.

Definice 3.1.1 Množina $A \subseteq \mathbb{N}$ je *rekurzivně spočetná*, je-li definičním oborem nějaké ČRF. Tj. existuje-li ČRF f, pro níž $A = \text{dom } f = \{x \mid f(x)\downarrow\}$. Množina $A \subseteq \mathbb{N}$ je *rekurzivní*, je-li její charakteristická funkce χ_A obecně rekurzivní, tedy ORF.

Pojmy „rekurzivní“ a „rekurzivně spočetné“ již jsme jednou použili v případě jazyků, uvědomíme-li si, že množina $A \subseteq \mathbb{N}$ odpovídá jazyku $L = \{w_i \mid i \in A\}$, zjistíme, že pojmy množina a jazyk jsou ve skutečnosti (efektivně) shodné, tedy to, co jsme si dosud řekli o rekurzivních a rekurzivně spočetných jazycích, platí i pro množiny a naopak to, co si nyní řekneme o rekurzivních a rekurzivně spočetných množinách, platí i pro jazyky, to vše pochopitelně díky tomu, že již víme, že TS jsou stejně silným výpočetním modelem jako ČRF. S tímto na paměti se nadále můžeme věnovat jen množinám. Stejně jako v případě jazyků i v případě množin můžeme každé rekurzivně spočetné množině přiřadit Gödelovo číslo funkce, jejímž je definičním oborem.

Definice 3.1.2 Pomocí W_e označíme e-tou rekurzivně spočetnou množinu, tj.

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

Při našich úvahách o rekurzivních a rekurzivně spočetných množinách se nám budou hodit další dva nástroje, jednak konečné aproximace funkcí a množin, jednak očíslování k-tic přirozených čísel. Konečnou aproximaci částečně rekurzivní funkce definujeme následujícím způsobem.

Definice 3.1.3 Pomocí $\varphi^{(n)}_{e,s}$, kde $e, n, s \in \mathbb{N}$ označíme *konečnou aproximaci* $\varphi^{(n)}_e$ funkce $\varphi^{(n)}_e$, kterou definujeme následujícím způsobem:

$$\varphi^{(n)}_{e,s}(x_1, \ldots, x_n) \simeq \begin{cases} \varphi^{(n)}_e(x_1, \ldots, x_n) & \text{pokud } (\exists y < s)[T_n(e, x_1, \ldots, x_n, y)] \\ \uparrow & \text{jinak} \end{cases}$$

Konečná aproximace funkce φ_e odpovídá intuitivně tomu, že stroj M_e necháme běžet s prostředky omezenými limitem s. Limitem s jsme omezili délku kódu výpočtu M_e nad vstupem, ale stejně tak bychom mohli omezovat přímo počet kroků, který dovolíme stroji M_e vykonat. Pokud si M_e nevystačí s omezenými protředky, je hodnota aproximace nedefinovaná.

Zatímco rozhodnutí, zda se stroj M_e zastaví, je dle věty 2.2.14 algoritmicky nerozhodnutelné, k rozhodnutí, zda se zastaví například nejpozději po s krocích, jej stačí vybavit počítadlem kroků. V implementovaném algoritmu navíc můžeme všechny cykly *while* nahradit cykly *for* s s jako horní mezí. Následující tvrzení by nás tedy nemělo překvapit.

Lemma 3.1.4 *Predikát $H^{(n)}(e, s, x_1, \ldots, x_n)$, definovaný jako*

$$H^{(n)}(e, s, x_1, \ldots, x_n) \Leftrightarrow \varphi^{(n)}_{e,s}(x_1, \ldots, x_n)\downarrow$$

je primitivně rekurzivní a platí, že

$$\varphi^{(n)}_e(x_1, \ldots, x_n)\downarrow \Leftrightarrow (\exists s)[\varphi^{(n)}_{e,s}(x_1, \ldots, x_n)\downarrow].$$

Důkaz : Plyne přímo z toho, že T_n je primitivně rekurzivní predikát dle věty 2.3.15 a omezená kvantifikace je primitivně rekurzivní dle lemmatu 2.3.9. To, že

$$\varphi^{(n)}_e(x_1, \ldots, x_n)\downarrow \Leftrightarrow (\exists s)[\varphi^{(n)}_{e,s}(x_1, \ldots, x_n)\downarrow],$$

je zřejmé z definice. ■

Konečnou aproximaci částečně rekurzivní funkce můžeme hned využít k definici konečné rekurzivní aproximace rekurzivně spočetné množiny.

Definice 3.1.5 Pomocí $W_{e,s}$ označíme odpovídající rekurzivní konečnou aproximaci. Tj.

$$W_{e,s} = \operatorname{dom} \varphi_{e,s} = \{x \mid \varphi_{e,s}(x)\downarrow\}.$$

Uvědomme si, že $W_{e,s}$ je nejen rekurzivní, ale i konečná množina.

Lemma 3.1.6 *Pro každé e, s je $W_{e,s}$ konečná rekurzivní množina, navíc $W_{e,s} \subseteq \{x \mid x < s\}$.*

Důkaz : Pro každé x platí, že $x \in W_{e,s}$, právě když $H(e, s, x)$, kde $H = H^{(1)}$ je primitivně rekurzivní predikát z lemmatu 3.1.4, z toho plyne, že $W_{e,s}$ je rekurzivní množina. Pokud platí, že $x \in W_{e,s}$, pak je to proto, že $\varphi_{e,s}(x)\downarrow$, což podle definice znamená, že existuje $y < s$, pro které $T(e, x, y)$, tedy y je kódem výpočtu Turingova stroje M_e nad vstupem $(x)_B$. Kód y v sobě obsahuje i tento vstup, a tedy musí platit, že $x < y < s$. ■

Intuitivně tedy $W_{e,s}$ obsahuje prvky množiny W_e menší než s, o kterých jsme schopni rozhodnout do s kroků. Ačkoli to, že pomocí s omezujeme délku kódu výpočtu je odlišný od situace, kdy bychom omezovali počet kroků stroje M_e, k definici $W_{e,s}$ bychom mohli použít tento druhý způsob, ten který jsme zvolili však vede k jednoduššímu použití omezené minimalizace a predikátu T. Další pomůckou, která se nám bude hodit, bude zakódování uspořádaných k-tic do přirozených čísel. Dvojice či k-tice čísel by šlo zakódovat řadou způsobů, které by všechny byly stejně dobré, my použijeme funkci, která se pro tyto účely často používá.

Definice 3.1.7 Pomocí $\langle x, y\rangle_2$ označíme kód uspořádané dvojice $[x, y]$ daný předpisem:

$$\langle x, y\rangle_2 = \frac{(x + y)(x + y + 1)}{2} + x$$

Je-li $n > 2$, pak kód n-tice $[x_1, \ldots, x_n]$ určíme pomocí $\langle x_1, \ldots, x_n\rangle_n = \langle x_1, \langle x_2, \ldots, x_n\rangle_{n-1}\rangle_2$. Pomocí $\pi_{n,i}(\langle x_1, \ldots, x_n\rangle) = x_i$ označíme odpovídající inverzní funkci.

Lemma 3.1.8 *Pro každé $n \geq 2$ a $i \leq n$ jsou funkce $\langle x_1, \ldots, x_n\rangle_n$ a $\pi_{n,i}$ primitivně rekurzivní a $\langle x_1, \ldots, x_n\rangle_n$ je bijekcí mezi $\mathbb{N}$ a $\mathbb{N}^{n+1}$.*

Důkaz : Všechny operace použité v definici jsou primitivně rekurzivní. Výběr prvku z dvojice je také primitivně rekurzivní, například

$$\pi_{2,1}(z) = \mu x \leq z\Big[(\exists y \leq z)[\langle x, y\rangle_2 = z]\Big].$$

Protože omezená minimalizace je primitivně rekurzivní dle lemmatu 2.3.12 a totéž platí pro omezený existenční kvantifikátor dle lemmatu 2.3.9, je tato funkce primitivně rekurzivní. Podobně by to šlo zobecnit na výběr libovolného prvku z n-tice. To, že jde o bijekci, plyne z použité funkce, důkaz tohoto faktu však vynecháme, protože by byl zbytečně technický a pro nás je pouze podstatné, že taková funkce existuje. ■

Protože přechod od jednotlivých složek ke kódu n-tice i opačný převod jsou primitivně rekurzivní, dovolíme si trochu zjednodušit značení a na místech, kde bychom použili proměnnou, kterou chceme interpretovat jako n-tici, použijeme často přímo n-tici s vyjmenovanými prvky, například pokud by byla f funkce dvou proměnných, budeme psát

$$\mu\langle x, y\rangle_2[f(x, y)],$$

místo formálně správného zápisu

$$\mu z[f(\pi_{2,1}(z), \pi_{2,2}(z))].$$

Podobně budeme toto značení používat i v jiných případech.

Díky efektivnímu zakódování dvojic se můžeme v jistém smyslu omezit na predikáty jedné proměnné, tedy množiny.

Lemma 3.1.9 *Predikát $P(x_1, \ldots, x_n)$ je primitivně rekurzivní (resp. obecně rekurzivní, resp. rekurzivně spočetný), právě když je primitivně rekurzivní (resp. obecně rekurzivní, resp. rekurzivně spočetný) i predikát $R(x) = \{\langle x_1, \ldots, x_n\rangle_n \mid P(x_1, \ldots, x_n)\}$.*

Důkaz : Zřejmé z toho, že kódování dvojic je efektivní. ■

3.2 Rekurzivní množiny

Nejprve se budeme věnovat rekurzivním množinám, které podle Churchovy-Turingovy teze odpovídají algoritmicky rozhodnutelným problémům. Není divu, že se tedy rekurzivní množiny chovají hezky k běžným množinovým operacím.

Lemma 3.2.1 *Rekurzivní množiny jsou uzavřeny na sjednocení, průnik a operaci doplňku, tj. jsou-li A a B rekurzivní množiny, pak i množiny $A \cup B$, $A \cap B$ a $\overline{A}$ jsou rekurzivní.*

Důkaz : Plyne z toho, že obecně rekurzivní predikáty jsou uzavřené na konjunkci, disjunkci a negaci dle lemmatu 2.3.10. V něm jsme to sice dokázali pro primitivně rekurzivní predikáty, tytéž důkazy však platí i pro obecně rekurzivní predikáty. ■

U množiny se jednak můžeme ptát, jak těžké je zjistit, zda do ní patří daný prvek, podobně se však můžeme zajímat o to, jak obtížné by bylo vypsat prvky množiny, pokud možno systematicky. Ukazuje se, že prvky rekurzivních množin lze vypsat v rostoucím pořadí, což platí i naopak.

Uvažme nyní pro jednoduchost konečnou množinu $A = \{1, 3, 4, 7, 9\}$, v paměti počítače bychom si tuto množinu mohli reprezentovat různými způsoby. Jednou možností je vytvořit bitové pole B, jehož rozsah je daný velikostí největšího čísla, které lze do množiny vložit, a

pro nějž platí, že $B[i] = 1$ právě když $i \in A$, jinak $B[i] = 0$. Pokud by například množina A obsahovala čísla z rozsahu $0, \ldots, 10$, pak bychom množinu A reprezentovali následujícím bitovým polem B:

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

Tento způsob reprezentace odpovídá charakteristické funkci χ_A, neboť zřejmě $B[i] = \chi_A(i)$, i když B v tomto případě obsahuje jen počáteční úsek nekonečného bitového pole, které odpovídá celé charakteristické funkci. Druhou možností je reprezentovat množinu A pomocí pole C, do kterého umístíme prvky množiny A uspořádané v rostoucím pořadí. Pole C, které by reprezentovalo množinu A by tedy vypadalo následovně:

0	1	2	3	4
1	3	4	7	9

Na pozici $C[i]$ je tedy i-tý nejmenší prvek množiny A počítáno od 0. Pokud by A byla nekonečná množina, je zřejmé, že i pole C by bylo nekonečné, je-li A konečná množina, pak hodnoty na pozicích $C[i]$ pro $i \geq |A|$ nejsou definované. Pole C tedy odpovídá funkci, která je definovaná jen na počátečním úseku přirozených čísel. Takové funkci budeme říkat úseková funkce.

Definice 3.2.2 Funkci f nazveme *úsekovou funkcí*, pokud její definiční obor tvoří počáteční úsek množiny přirozených čísel, tedy pokud platí, že

$$(\forall x)(\forall y)\Big[\Big(f(x)\downarrow \wedge (y < x)\Big) \Rightarrow f(y)\downarrow\Big].$$

Funkce je tedy úseková, pokud je buď definovaná pro všechny vstupy, nebo existuje nějaké číslo c takové, že pro každé $x < c$ je $f(x)\downarrow$ a pro každé $x \geq c$ je $f(x)\uparrow$.

Představíme-li si pole C jako takovou funkci f, je množina A rovna oboru hodnot f, protože obsahuje prvky, které jsou vráceny pro nějaký vstup. Není překvapivé, že obě reprezentace množiny A, tedy reprezentaci bitovým polem B a reprezentací uspořádaným seznamem prvků v poli C, jsme schopni vzájemně na sebe převést. Místo potenciálně nekonečného pole B však použijeme charakteristickou funkci a místo potenciálně nekonečného pole C rostoucí úsekovou funkci.

Věta 3.2.3 *Množina A je rekurzivní, právě když je oborem hodnot nějaké rostoucí úsekové ČRF (tj. existuje rostoucí úseková ČRF f, pro níž platí $A = rng\ f$).*

Důkaz : Předpokládejme nejprve, že A je rekurzivní množina. To znamená, že její charakteristická funkce $\chi_A(x)$ je obecně rekurzivní. Popíšeme funkci $f(i)$, která pro dané i vrátí i-tý nejmenší prvek množiny A (přičemž i počítáme od 0, tj. první prvek je vrácen pro $i = 0$). Je-li A konečná množina, není pro $i \geq |A|$ hodnota $f(i)$ definována. Takto popsaná funkce bude rostoucí i úseková. Intuitivní algoritmus, který bude počítat funkci $f(i)$ je jednoduchý: Najdi nejmenší prvek množiny A a poté i-krát hledej nejmenší větší prvek. Tento postup by odpovídal algoritmu 3.2.4[1].

[1]Poznamenejme, že přímočarý algoritmus pro výpočet funkce i implementovaný ve vyšším programovacím jazyku vypadal trochu jinak a byl by jistě jednodušší, stačilo by si počítat počet nalezených 1 vrácených charakteristickou funkcí a počítat, kdy dosáhneme počtu $i+1$. Zde popsaný algoritmus je již ušit na míru tomu, že jej chceme implementovat pomocí ČRF.

Algoritmus 3.2.4 Výpočet funkce $f(i)$

Vstup: Přirozené číslo i, implicitně též množina A reprezentovaná svou charakteristickou funkcí χ_A.

Výstup: i-tý nejmenší prvek množiny A, počítáno od 0, pokud je A nekonečná nebo $i < |A|$. Jinak není hodnota funkce f definována.

```
 1: y := 0
 2: while χ_A(y) = 0
 3: do
 4:    y := y + 1
 5: done
 6: u := y
 7: for k := 0 to i − 1
 8: do
 9:    z := 0
10:    while z ≤ u or χ_A(z) = 0
11:    do
12:       z := z + 1
13:    done
14:    u := z
15: done
16: return u
```

První cyklus *while* na řádcích 2 – 5 najde index nejmenšího (tedy 0-tého) prvku, který do množiny A patří. Poté algoritmus i-krát opakuje hledání následujícího prvku. Hledání následujícího prvku je obsaženo v druhém cyklu *while* na řádcích 10 – 13. Zbývá tento algoritmus přepsat do jazyka částečně rekurzivních funkcí, což je již celkem přímočaré.

První inicializační cyklus si definujeme jako ČRF

$$h(v) = \mu y[\chi_A(y) \simeq 1],$$

všimněte si, že tato funkce na svém parametru vůbec nezávisí, vždy najde nejmenší prvek množiny A. Funkci uvnitř cyklu *for* si definujeme jako ČRF

$$g(i, u, v) = \mu z[(z > u) \wedge \chi_A(z) \simeq 1].$$

Funkce g dostává jako druhou hodnotu (tedy z) hodnotu z rekurze, tedy aktuální hodnotu naposledy nalezeného prvku, následně hledá nejmenší prvek množiny A, který je větší než tento poslední nalezený prvek uložený v z. Pomocí primitivní rekurze odvodíme funkci $f' = R_2(h, g)$ a poté stačí položit $f(i) \simeq f'(i, i)$ (připomeňme si, že primitivní rekurze odvodí funkci alespoň dvou proměnných, proto funkci f odvodíme takovouto oklikou).

Nyní předpokládejme, že $A = \text{rng}\, f$, kde f je rostoucí úseková ČRF. Pokud f není ORF, znamená to podle definice úsekové funkce, že doména f je konečná, a tedy je konečná i samotná množina A a je tedy triviálně rekurzivní. Předpokládejme tedy, že f je obecně rekurzivní, tedy všude definovaná funkce, pro kterou platí, že $f(i)$ vrátí i-tý nejmenší prvek množiny A. Charakteristickou funkci můžeme nyní spočítat jednoduchým způsobem, popsaným v algoritmu 3.2.5.

Algoritmus 3.2.5 Výpočet charakteristické funkce $\chi_A(x)$ množiny A.

Vstup: Přirozené číslo x, implicitně též funkce f, pro kterou platí, že $f(i)$ vrátí i-tý nejmenší prvek množiny A.

Výstup: 1, pokud $x \in A$, 0 jinak.

```
 1: i := 0
 2: while f(i) < x
 3: do
 4:    i := i + 1
 5: done
 6: if f(i) = x
 7: then
 8:    return 1
 9: else
10:    return 0
11: endif
```

Algoritmus 3.2.5 vždy skončí, neboť množina A je nekonečná, a proto musí obsahovat prvek, který je větší nebo roven x, ve skutečnosti platí, že i-tý prvek musí být větší nebo roven i, jinými slovy platí, že $i \leq f(i)$. Proto ve skutečnosti stačí hledat $i \leq x$ a mohli bychom se tedy obejít bez obecného cyklu *while* a nahradit jej cyklem *for*, toho využijeme při odvození funkce χ_A. Definujme funkci

$$g(x) = \mu i \leq x[f(i) \geq x],$$

která odpovídá cyklu *while* v Algoritmu 3.2.5. Nyní platí, že $x \in A$, právě když $x = f(g(x))$, přičemž rovnost je primitivně rekurzivní predikát, a proto je charakteristická funkce χ_A obecně rekurzivní. Pokud je f dokonce primitivně rekurzivní funkce, dostaneme, že i χ_A je primitivně rekurzivní funkce, neboť omezená minimalizace použitá v odvození funkce g je primitivně rekurzivní podle lemmatu 2.3.12. V každém případě je-li f ORF, je ORF i χ_A a A je tedy rekurzivní množina. ■

Protože úsekové funkce s nekonečnou doménou musí být všude definované, můžeme tuto větu pro nekonečné množiny přeformulovat následovně.

Důsledek 3.2.6 *Nechť A je nekonečná množina, pak A je rekurzivní, právě když je oborem hodnot nějaké rostoucí ORF.*

3.3 Rekurzivně spočetné množiny a predikáty

Než se budeme věnovat rekurzivně spočetným množinám, vraťme se ještě na chvíli k rekurzivně spočetným predikátům. Podle lemmatu 2.3.19 lze každý rekurzivně spočetný predikát napsat ve formě primitivně rekurzivního predikátu, před který vložíme jeden existenční kvantifikátor. Jak ukazuje následující tvrzení, ve skutečnosti nezáleží na počtu těchto existenčních kvantifikátorů, neboť pokud přidáme existenční kvantifikátor před rekurzivně spočetný predikát, vznikne opět rekurzivně spočetný predikát.

Lemma 3.3.1 *Je-li $P(x_1, \ldots, x_n, y)$ rekurzivně spočetný predikát, pak i*

$$R(x_1, \ldots, x_n) = (\exists y)\ [P(x_1, \ldots, x_n, y)]$$

je rekurzivně spočetný predikát.

Důkaz : Pro jednoduchost budeme uvažovat případ, kdy $n = 1$, zobecnění pro $n > 1$ proměnných je triviální. Protože $P(x, y)$ je rekurzivně spočetný predikát, můžeme jej podle lemmatu 2.3.19 zapsat jako

$$P(x, y) = (\exists z)\ [Q(x, y, z)]$$

pro nějaký primitivně rekurzivní predikát Q. Z toho plyne, že

$$R(x) = (\exists y)(\exists z)\ [Q(x, y, z)].$$

Zakódujeme-li nyní y a z jako dvojici, dostaneme, že

$$R(x) = (\exists \langle y, z\rangle_2)\ [Q(x, y, z)]$$

a označíme-li si

$$D(x, u) = Q(x, \pi_{2,1}(u), \pi_{2,2}(u)),$$

pak D je primitivně rekurzivní predikát, pro který platí, že

$$R(x) = (\exists u)\ D(x, u).$$

Podle lemmatu 2.3.19 tedy dostáváme, že $R(x)$ je rekurzivně spočetný predikát. ■

Nyní si ukážeme, že i existenční kvantifikátor před rekurzivně spočetným predikátem je v jistém smyslu efektivní, neboť existuje funkce, která jej implementuje.

Věta 3.3.2 *Nechť $R(x, y)$ je rekurzivně spočetný predikát, potom existuje ČRF f (také výběrová funkce nebo selektor pro R), pro kterou platí, že $f(x)\downarrow \Leftrightarrow (\exists y)R(x, y)$ a pokud $f(x)\downarrow$, pak $R(x, f(x))$.*

Důkaz : Podle lemmatu 2.3.19 můžeme R zapsat jako

$$R(x, y) = (\exists z)\ [Q(x, y, z)]$$

pro nějaký primitivně rekurzivní predikát Q. A tedy

$$(\exists y)R(x, y) \Leftrightarrow (\exists y)(\exists z)Q(x, y, z).$$

Protože predikát Q je primitivně rekurzivní, dostaneme, že funkce

$$g(x) \simeq \mu\langle y, z\rangle_2 Q(x, y, z)$$

skutečně najde vhodnou dvojici y, z, pro kterou je $Q(x, y, z)$ splněn, pokud taková dvojice existuje. Nyní z nalezené hodnoty stačí vybrat y, tedy první prvek, celkově tedy definujeme funkci f jako

$$f(x) \simeq \pi_{2,1}(g(x)).$$

■

Fakt, že rekurzivně spočetné množiny jsou uzavřené na sjednocení a průnik, plyne už z toho, že rekurzivně spočetné predikáty jsou uzavřeny na konjunkci a disjunkci. Díky s-m-n větě 2.3.22 jsme navíc schopni provést toto sjednocení či průnik efektivně.

Věta 3.3.3 *Rekurzivně spočetné množiny jsou efektivně uzavřené na sjednocení a průnik. Tj. existují PRF f a g pro něž platí, že $W_{f(x,y)} = W_x \cup W_y$ a $W_{g(x,y)} = W_x \cap W_y$.*

Důkaz : Uvažme nejprve sjednocení a funkci $f(x, y)$. Nechť e-tý program počítá následovně: Pro daný vstup x, y, u hledej nejmenší limit s, který již stačí, aby u patřilo do $W_{x,s}$ nebo do $W_{y,s}$. Tedy

$$\varphi_e(x, y, u) \simeq (\mu s)[u \in W_{x,s} \vee u \in W_{y,s}].$$

Z vlastností konečných aproximací, jmenovitě z lemmatu 3.1.6 vyplývá, že predikát použitý jako podmínka minimalizace je rekurzivní. Platí tedy, že

$$\varphi_e(x, y, u)\downarrow \Leftrightarrow u \in W_x \cup W_y.$$

Program s číslem e je konkrétní program, který sestrojíme s pomocí univerzální ČRF. Funkci f dostaneme pomocí s-m-n věty 2.3.22 jako $f(x, y) = s_1^2(e, x, y)$, protože potom platí

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

Podobně můžeme postupovat i v případě průniku a funkce g, stačilo by místo disjunkce použít konjunkci. V tomto případě však můžeme použít i

$$g(x, y, u) \simeq \varphi_x(u) + \varphi_y(u),$$

neboť pro takto definovanou funkci g platí, že

$$g(x, y, u)\downarrow \Leftrightarrow \varphi_x(u)\downarrow \wedge \varphi_y(u)\downarrow \Leftrightarrow u \in W_x \wedge u \in W_y.$$

Další postup s použitím s-m-n věty 2.3.22 je shodný s postupem v případě sjednocení. ■

Postup důkazu věty 3.3.3 je velmi obecný a můžeme jej použít i v jiných případech. Řadu operací s množinami či funkcemi lze provést efektivně, což by nás však nemělo překvapit, protože jde přesně o ty operace, které jsme schopni popsat formou intuitivního algoritmu. Podle Churchovy-Turingovy teze jsme schopni tento intuitivní algoritmus realizovat na Turingově stroji a tedy i částečně rekurzivní funkcí, jde-li o algoritmus, který nevyužívá cyklus *while*, pak dokonce získáme primitivně rekurzivní funkci. S-m-n věta 2.3.22, jíž je tato věta důsledkem, tedy jen znovu potvrzuje Churchovu-Turingovu tezi. My si však často ulehčíme situaci a nebudeme vždy postupovat takto detailně s použitím s-m-n věty.

Stejně jako v případě jazyků, i v případě množin platí Postova věta, která vypovídá o vztahu mezi rekurzivními a rekurzivně spočetnými množinami.

Věta 3.3.4 (Postova) *Množina A je rekurzivní, právě když A i $\overline{A} = \mathbb{N} \setminus A$ jsou rekurzivně spočetné množiny.*

Důkaz : Plyne z vět 2.2.10, 2.2.11, ekvivalence TS s ČRF a korespondence mezi jazyky a množinami. My si ale ukážeme i přímý důkaz v kontextu množin. Je-li A rekurzivní, pak je definičním oborem funkce

$$f(x) \simeq \mu y[(\lambda xy[1 \dot{-} \chi_A(x)]) \simeq 0],$$

tj. funkce $f(x)$ odpovídá nekonečnému cyklu, pokud $x \notin A$, naopak pokud $x \in A$, skončí $f(x)$ okamžitě už ve chvíli, kdy $y = 0$. Podobně $\overline{A}$ je definičním oborem funkce

$$g(x) \simeq \mu y[(\lambda xy[\chi_A(x)]) \simeq 0].$$

Obě množiny A i $\overline{A}$ jsou tedy rekurzivně spočetné.

Předpokládejme nyní, že A i $\overline{A}$ jsou rekurzivně spočetné. Buďte i a j jejich Gödelova čísla, tedy $A = W_i$ a $\overline{A} = W_j$. Uvědomme si, že část práce jsme už učinili v případě sjednocení ve větě 3.3.3, i zde si nejprve určíme limit, který již stačí k rozhodnutí následující funkcí $g(x)$:

$$g(x) \simeq \mu s[x \in W_{i,s} \vee x \in W_{j,s}]$$

Narozdíl od sjednocení, v tomto případě je funkce g definovaná pro každé přirozené číslo x, je tedy obecně rekurzivní. To proto, že prvek x patří buď do A nebo $\overline{A}$, čili $W_i \cup W_j = \mathbb{N}$, a tedy musí existovat limit s, který stačí k tomu, aby buď $x \in W_{i,s}$, nebo $x \in W_{j,s}$. Nyní stačí tento limit použít k rozhodnutí, nyní totiž $x \in A$, právě když $x \in W_{i,g(x)}$. Protože $g(x)$ je obecně rekurzivní, je toto rozhodnutí rovněž obecně rekurzivní podle lemmatu 3.1.4, a tedy i množina A je rekurzivní. ■

Nyní se zaměříme na problém vypsání prvků rekurzivně spočetné množiny.

Věta 3.3.5 (Generování rekurzivně spočetných množin) *Nechť A je množina, potom jsou následující tvrzení ekvivalentní:*

(i) Množina A je rekurzivně spočetná.

(ii) A je prázdná, nebo je oborem hodnot nějaké ORF, tj. existuje ORF g, pro níž $A = \operatorname{rng} g = \{x \mid (\exists y)[g(y) = x]\}$.

(iii) A je oborem hodnot nějaké ČRF.

(iv) A je oborem hodnot nějaké rostoucí ČRF.

Důkaz :

- $(i) \Rightarrow (ii)$. Předpokládejme, že A je neprázdná rekurzivně spočetná množina, jejíž Gödelovo číslo si označíme pomocí e, tedy $A = W_e = \operatorname{dom} \varphi_e$. Naším cílem je zkonstruovat obecně rekurzivní funkci g tak, aby platilo $A = \operatorname{rng} g$. Náš postup navíc ukáže, jak z Gödelova čísla e určit Gödelovo číslo funkce g. Rozeberme si nejprve, proč nemůžeme postupovat stejně jako v případě rekurzivních množin. Už postup, kterým jsme u rekurzivní množiny hledali nejmenší prvek, v případě rekurzivně spočetné množiny selže. Pokud by totiž 0 nepatřila do A, tak se cyklus *while*, v němž bychom nejmenší prvek hledali, zarazí už při výpočtu $\varphi_e(0)$, neboť tato hodnota není definovaná. Podobně nemůžeme hledat nejmenší prvek větší než zadaný parametr.

  Musíme proto postupovat opatrněji. Za prvé, nemůžeme sice přímo najít nejmenší prvek, ale můžeme najít nějaký prvek množiny $A = W_e$, a to například následující funkcí:

  $$a(e) = \pi_{2,1}(\mu\langle x, s\rangle_2[x \in W_{e,s}])$$

  Nejedná se při tom o nic jiného, než o selektor rekurzivně spočetného predikátu $(\exists y)[y \in W_e]$ (viz věta 3.3.2), protože tento predikát je splněn díky neprázdnosti W_e, bude pro dané e hodnota $a(e)$ definovaná. Takto nalezený prvek použijeme jako jakéhosi žolíka, kterého vrátí konstruovaná funkce g ve chvíli, kdy si neví jinak rady. Sestrojíme nejprve funkci $\varphi^{(2)}_{e_1}(e, z)$, která si vyloží parametr z jako dvojici $z = \langle x, s\rangle_2$, ověří, zda $x \in W_{e,s}$ a pokud ano, vrátí hodnotu x, v opačném případě vrátí žolíka $a(e)$, tedy:

  $$\varphi^{(2)}_{e_1}(e, z = \langle x, s\rangle_2) \simeq \begin{cases} x & x \in W_{e,s} \\ a(e) & \text{jinak} \end{cases}$$

Protože $W_{e,s}$ je rekurzivní množina, a hodnota $a(e)$ je pro dané e vždy definovaná vzhledem k neprázdnosti W_e, bude hodnota $\varphi^{(2)}_{e_1}(e,z)$ definovaná pro každé z. Všimněme si, že definice funkce $\varphi^{(2)}_{e_1}$ nezávisí na hodnotě e, ta je jí předána jako parametr. Nyní už odvodíme $g(z) \simeq \varphi^{(2)}_{e_1}(e,z)$, a tedy podle s-m-n věty (2.3.22) platí

$$g(z) \simeq \varphi_{s^1_1(e_1,e)}(z) \simeq \varphi^{(2)}_{e_1}(e,z).$$

Takto jsme tedy spočítali i Gödelovo číslo funkce g z e. Zřejmě platí, že $g(z) \in W_e$ pro každé z, uvažme naopak, že pro každý prvek $b \in W_e$ existuje s, pro něž $b \in W_{e,s}$, a tedy $g(\langle b,s\rangle_2) = b$. Každý prvek W_e je tedy funkcí g pro nějaký vstup y vrácen, jenom nedokážeme odhadnout, pro jak velké y. Dohromady dostáváme, že $A = W_e = \operatorname{rng} g$.

- $(ii) \Rightarrow (iii)$. Plyne toho, že každá ORF je i ČRF. Prázdná množina je oborem hodnot ČRF, která není definovaná pro žádný vstup.

- $(iii) \Rightarrow (i)$. Předpokládejme, že A je oborem hodnot částečně rekurzivní funkce $g \simeq \varphi_e$. Uvědomme si, že rozhodnutí, zda $g(y)\downarrow = x$ je rekurzivně spočetný predikát (plyne například z toho, že $\varphi_{e,s}(y)\downarrow = x$ je primitivně rekurzivní predikát dle lemmatu 3.1.4, a tedy $g(y)\downarrow = x \Leftrightarrow (\exists s)[\varphi_{e,s}(y)\downarrow = x]$ je rekurzivně spočetný predikát dle lemmatu 2.3.19), podle lemmatu 3.3.1 je tedy i $(\exists y)[g(y)\downarrow = x]$ rekurzivně spočetný predikát, a tedy množina
$$A = \{x \mid (\exists y)[g(y)\downarrow = x]\}$$
je rekurzivně spočetná. Ze znalosti Gödelova čísla e funkce g bychom opět s pomocí s-m-n věty mohli spočítat Gödelovo číslo funkce, jejíž doménou množina A je. Pro úplnost totiž můžeme psát
$$A = \operatorname{dom} \lambda x\Big[\mu\langle y,s\rangle_2[\varphi_{e,s}(y)\downarrow = x]\Big].$$

- $(i) \Rightarrow (iv)$. Předpokládejme, že $A = W_e$, tedy $A = \operatorname{dom}\varphi_e$. Buď $g(e,x)$ funkce definovaná jako:
$$g(e,x) = \begin{cases} x & x \in W_e \\ \uparrow & \text{jinak} \end{cases}$$
Funkce $g(e,x)$ je zřejmě ČRF, například proto, že ji lze zapsat jako $g(e,x) = o(\varphi_e(x)) + x$ (tj. do nulové funkce vložíme $\varphi_e(x)$ jako argument, hodnota tedy na hodnote $\varphi_e(x)$ nezávisí, definovatelnost ano). Položme nyní $h(x) = g(e,x)$, potom je A jak oborem hodnot, tak definičním oborem funkce h, funkce h je rostoucí ČRF, jejíž Gödelovo číslo lze opět efektivně spočítat z e.

- $(iv) \Rightarrow (iii)$. Toto je zřejmé. Implikaci $(iii) \Rightarrow (i)$ už máme hotovou, a tedy jsme ukázali i $(iv) \Rightarrow (i)$.

■

Bod (ii) věty 3.3.5 naznačuje, odkud rekurzivně spočetné množiny (*recursively enumerable sets*) dostaly své jméno. Je-li A rekurzivně spočetná množina, nejsme sice obecně schopni algoritmicky rozhodnout, zda $x \in A$, ale jsme schopni efektivně vypsat (*enumerate*) všechny prvky A. O funkci g z tvrzení (ii) budeme též říkat, že *generuje* množinu A. Na rozdíl od rekurzivních množin však nejsme schopni vypsat prvky rekurzivně spočetné množiny systematicky, tedy v rostoucím pořadí, a nepodaří se nám ani vyhnout tomu, abychom při výpisu

nějaký prvek zopakovali[2]. Funkci, která by byla rostoucí a jejímž oborem hodnot je množina A, sice můžeme zkonstruovat, ale musíme v tom případě rezignovat na obecně rekurzivní funkce a na to, aby daná funkce byla všude definovaná, jak ukazuje bod (iv). Taková funkce přitom není pro generování množiny A použitelná.

3.4 Nerozhodnutelné problémy, převoditelnost a Riceova věta

V této části se vrátíme k nerozhodnutelným problémům a podíváme se na to, jak dokázat o daném problému, že není algoritmicky rozhodnutelný (tj. není rekurzivní). Podobně jako u Turingových strojů, i zde začneme s problémem zastavení K_0 a jeho diagonálou K:

$$\begin{aligned} K_0 &= \{\langle x,y\rangle_2 \mid \varphi_x(y)\downarrow\} = \{\langle x,y\rangle_2 \mid y \in W_x\} \\ K &= \{x \mid \varphi_x(x)\downarrow\} = \{x \mid x \in W_x\} \end{aligned}$$

I pro tyto množiny platí to, co již víme o problému zastavení z věty 2.2.14.

Věta 3.4.1 *Množiny K a K_0 jsou rekurzivně spočetné, ale nejsou rekurzivní.*

Důkaz : Obojí jsme vlastně už ukázali v kontextu Turingových strojů ve větě 2.2.14, zopakujme si však důkaz i v jazyce ČRF. Nechť $\varphi_z^{(2)}$ označuje univerzální ČRF. Potom

$$\begin{aligned} K &= \text{dom}\ \lambda x[\varphi_z^{(2)}(x,x)] \qquad \text{a} \\ K_0 &= \text{dom}\ \lambda\langle x,y\rangle_2[\varphi_z^{(2)}(x,y)]. \end{aligned}$$

Podle definice jsou tedy obě množiny rekurzivně spočetné.

Předpokládejme pro spor, že množina K je rekurzivní, podle věty 3.3.4 to znamená, že $\overline{K}$ je rekurzivně spočetná množina. Nechť φ_e je ČRF, pro níž $\overline{K} = \text{dom}\ \varphi_e$ a ptejme se, zda $e \in \overline{K}$ nebo ne. Podle definice K platí, že

$$e \in \overline{K} \Leftrightarrow \varphi_e(e)\uparrow .$$

Na druhou stranu však to, že $\overline{K} = \text{dom}\ \varphi_e$, implikuje

$$e \in \overline{K} \Leftrightarrow \varphi_e(e)\downarrow .$$

Dohromady dostáváme, že

$$\varphi_e(e)\downarrow \Leftrightarrow \varphi_e(e)\uparrow ,$$

což pochopitelně nemůže platit a náš předpoklad musí být mylný. Množina K tedy není rekurzivní.

Předpokládejme nyní pro spor, že množina K_0 je rekurzivní a nechť χ_{K_0} označuje její obecně rekurzivní charakteristickou funkci. Definujme $\chi_K(x) = \chi_{K_0}(\langle x,x\rangle_2)$, potom je χ_K obecně rekurzivní charakteristickou funkcí K. Tím jsme se však dostali do sporu s tím, že K rekurzivní není. ■

Všimněme si, jakým způsobem jsme dokazovali to, že K_0 není rekurzivní množina. Otázku $x \in K$ jsme převedli na otázku $\langle x,x\rangle_2 \in K_0$, tedy kdybychom uměli rozhodnout tuto druhou

[2] I když nepříliš komplikovanou úpravou důkazu implikace $(i) \Rightarrow (ii)$ věty 3.3.5, bychom mohli dosáhnout toho, že jediným prvkem, který by byl vypsán víckrát než jednou, by byl prvek-žolík nalezený funkcí $a(e)$. K tomu by například stačilo nahradit podmínku $x \in W_{e,s}$ podmínkou $x \in W_{e,s+1} \setminus W_{e,s}$. Není totiž těžké vypozorovat, že $|W_{e,s+1} \setminus W_{e,s}| \leq 1$ pro $s > 0$. Dalším pokračováním těchto úvah bychom mohli vytvořit prostou (ale nikoli současně rostoucí a totální!) funkci g, pro níž by platilo $W_e = \text{rng}\ g$. Stačilo by postupně zvyšovat limit s a vypisovat prvky v pořadí, v jakém přibývají do $W_{e,s}$, pro nás toto však již není podstatné.

otázku, uměli bychom rozhodnout i první. Zodpovězení otázky $z \in K_0$ je tedy alespoň tak těžké, jako zodpovězení otázky $x \in K$. Tento postup důkazu zobecníme pomocí pojmů 1-převoditelnosti a m-převoditelnosti.

Definice 3.4.2 Řekneme, že množina A je *m-převoditelná* na množinu B a označíme tento fakt pomocí $A \leq_m B$, pokud existuje ORF f, taková, že $(\forall x \in \mathbb{N})[x \in A \Leftrightarrow f(x) \in B]$. Je-li funkce f navíc prostá, řekneme, že A je 1-*převoditelná* na B a označíme tento fakt pomocí $A \leq_1 B$.

Řekneme, že množina A je *m-úplná* (resp. 1-*úplná*) *pro třídu rekurzivně spočetných množin*, pokud je A rekurzivně spočetná a každá jiná rekurzivně spočetná množina je na ni m-převoditelná (resp. 1-převoditelná). Obvykle budeme říkat pouze, že množina A je m-úplná nebo 1-úplná, a dodatek „pro třídu rekurzivně spočetných množin“ budeme vynechávat, protože jinými než rekurzivními a rekurzivně spočetnými množinami se nebudeme zabývat.

Následující lemma shrnuje základní vlastnosti převoditelnosti.

Lemma 3.4.3

1. *Je-li $A \leq_m B$ a B je rekurzivní množina, pak A je rekurzivní množina.*

2. *Je-li $A \leq_m B$ a B je rekurzivně spočetná množina, pak A je rekurzivně spočetná množina.*

3. *Relace $\leq_m$ a $\leq_1$ jsou reflexivní a tranzitivní.*

4. *Je-li A m-úplná množina, je-li B rekurzivně spočetná množina a platí-li $A \leq_m B$, potom i B je m-úplná množina. Totéž platí pro 1-převoditelnost a 1-úplnost.*

5. *Je-li $A \leq_m B$, pak platí i $\overline{A} \leq_m \overline{B}$. Totéž platí pro 1-převoditelnost.*

Důkaz :

1. Je-li B rekurzivní množina a f je ORF, která převádí A na B, pak $x \in A \Leftrightarrow f(x) \in B$, tedy charakteristickou funkci A můžeme zapsat jako $\chi_A(x) = \chi_B(f(x))$. Tato funkce je jistě ORF a tedy A je rekurzivní množina.

2. Je-li B rekurzivně spočetná množina a platí-li $B = \text{dom}\, \varphi_e$ a je-li f ORF, která převádí A na B, pak $A = \text{dom}\, \lambda x[\varphi_e(f(x))]$ a jde tedy o rekurzivně spočetnou množinu.

3. To, že jsou $\leq_m$ a $\leq_1$ reflexivní relace plyne z toho, že identita je prostá ORF. Předpokládejme, že $A \leq_1 B$ a $B \leq_1 C$, nechť f je prostá ORF převádějící A na B a g buď prostá ORF převádějící B na C. Pak $h(x) = g(f(x))$ je prostá ORF která převádí A na C, což ukazuje, že $\leq_1$ je tranzitivní. Pro $\leq_m$ platí týž argument s vynecháním prostoty.

4. Toto plyne okamžitě z tranzitivity $\leq_m$, $\leq_1$ a definice úplnosti.

5. Plyne přímo z definice převoditelnosti, převod $\overline{A}$ na $\overline{B}$ zajistí stejná funkce jako převod A na B.

■

Lemma 3.4.4 *Množiny K a K_0 jsou 1-úplné.*

Důkaz : Ukážeme nejprve, že K je 1-úplná množina. Již víme, že K je rekurzivně spočetná, stačí tedy ukázat, že libovolnou rekurzivně spočetnou množinu lze na K převést prostou obecně rekurzivní funkcí. Definujme funkci $\varphi^{(3)}_{e_1}(e,x,y) \simeq \varphi_e(x)$. Podle s-m-n věty (2.3.22) platí:

$$\varphi_{s^2_1(e_1,e,x)}(y) \simeq \varphi^{(3)}_{e_1}(e,x,y) \simeq \varphi_e(x)$$

Definujme tedy funkci $f(e,x) = s^2_1(e_1,e,x)$. Protože s^2_1 je prostá PRF, je i f prostá PRF. Navíc platí, že $\varphi_{f(e,x)}$ je buď všude definovaná konstantní funkce, pokud $x \in W_e$, nebo jde o funkci, která není definovaná pro žádný vstup v případě, že $x \notin W_e$, platí tedy, že:

$$\begin{array}{lllllllll} x \in W_e & \Rightarrow & \varphi_e(x)\downarrow & \Rightarrow & (\forall y)[\varphi^{(3)}_{e_1}(e,x,y)\downarrow] & \Rightarrow & \varphi_{f(e,x)}(f(e,x))\downarrow & \Rightarrow & f(e,x) \in K \\ x \notin W_e & \Rightarrow & \varphi_e(x)\uparrow & \Rightarrow & (\forall y)[\varphi^{(3)}_{e_1}(e,x,y)\uparrow] & \Rightarrow & \varphi_{f(e,x)}(f(e,x))\uparrow & \Rightarrow & f(e,x) \notin K \end{array}$$

Dohromady dostaneme požadovanou ekvivalenci $x \in W_e \Leftrightarrow f(e,x) \in K$, a z toho plyne, že funkce $g(x)$ definovaná jako $g(x) \simeq f(e,x)$, je prostá ORF, která převádí množinu W_e na K. Protože W_e byla libovolná množina, dostáváme, že $(\forall e)[W_e \leq_1 K]$, a K je tedy 1-úplná množina.

Podle věty 3.4.1 je K_0 rekurzivně spočetná množina. V téže větě jsme však ukázali i to, že $K \leq_1 K_0$ pomocí prosté ORF $h(x) = \langle x,x\rangle_2$. Těžkost množiny K_0 tedy plyne z tranzitivity 1-převoditelnosti. ■

Poznámka 3.4.5 *Z lemmatu 3.4.3 a faktu, že $\overline{K}$ není narozdíl od K rekurzivně spočetná množina, plyne, že nemůže platit $\overline{K} \leq_1 K$, a podle bodu 5 lemmatu 3.4.3 tedy neplatí ani $K \leq_1 \overline{K}$. Toto lze pochopitelně říci i o každé jiné rekurzivně spočetné množině, která není rekurzivní. Z toho plyne, že existují dvojice množin, jež nejsou pomocí $\leq_1$ porovnatelné, totéž zřejmě platí i pro m-převoditelnost. Kromě 1-převoditelnosti a m-převoditelnosti se často uvažuje turingovská převoditelnost. Řekneme, že množina A je turingovsky převoditelná na množinu B a označíme tento fakt pomocí $A \leq_T B$, pokud existuje Turingův stroj, který vždy skončí a přijímá jazyk A, přičemž během své práce může pokládat dotazy, zda $x \in B$ pro libovolné x. U těchto dotazů předpokládáme, že jsou pokládány černé skříňce, která vždy okamžitě a správně odpoví*[3]*. Tento způsob převoditelnosti odpovídá následující intuici: „Pokud existuje algoritmus rozhodující otázku $x \in B$, pak existuje i algoritmus rozhodující $x \in A$.“ Jde tedy o celkem přirozený koncept. Všimněme si však, že v tomto případě platí, že $K \leq_T \overline{K}$ i $\overline{K} \leq_T K$, protože algoritmus, který se může ptát, zda $x \in K$, může tuto odpověď znegovat a zodpovědět i otázku, zda $x \in \overline{K}$. Zřejmě neplatí, že pokud $A \leq_T B$, pak i $A \leq_m B$, opačná implikace přitom jednoduše platí. I proto používáme raději 1-převoditelnost či alespoň m-převoditelnost, neboť jde o silnější pojmy než turingovská převoditelnost a umožňuje rozlišit rekurzivně spočetné množiny od jejich doplňků.*

Riceova věta ukazuje, že většina otázek, jež si můžeme o algoritmech položit, není algoritmicky řešitelná.

Věta 3.4.6 (Riceova) *Nechť C je libovolná třída částečně rekurzivních funkcí, potom je množina $A_C = \{e \mid \varphi_e \in C\}$ rekurzivní, právě když $C = \emptyset$ nebo C obsahuje všechny ČRF.*

Důkaz : Pokud je $C = \emptyset$, potom i $A_C = \emptyset$ a jde tedy o rekurzivní množinu. Podobně pokud C obsahuje všechny ČRF, potom zřejmě $A_C = \mathbb{N}$, což je opět rekurzivní množina. Předpokládejme nyní, že C není ani prázdná, ani neobsahuje všechny ČRF. Ukážeme, že platí $K \leq_1 A_C$ nebo $K \leq_1 \overline{A_C}$, což podle lemmatu 3.4.3 a Postovy věty 3.3.4 implikuje, že A_C nemůže být rekurzivní, protože K není rekurzivní množina.

[3]Formálně bychom definovali pojem turingovské převoditelnosti s pomocí Turingova stroje s orákulem, ale pro naše účely bude uvedená definice postačující.

Nechť e_0 je Gödelovým číslem funkce, která není definovaná pro žádný vstup, platí pro ni tedy, že $(\forall x)[\varphi_{e_0}(x)\uparrow]$, takovou funkci můžeme definovat třeba jako $\varphi_{e_0}(x) \simeq \mu y[s(x) \simeq 0]$. Předpokládejme nejprve, že $\varphi_{e_0} \in \overline{C}$, tedy i $e_0 \in \overline{A_C}$, a ukažme, že v tom případě $K \leq_1 A_C$. Nechť e_1 je Gödelovým číslem funkce, která patří do C, a tedy $e_1 \in A_C$, C je neprázdná, proto takové e_1 určitě existuje. Protože $\varphi_{e_0} \notin C$, zatímco $\varphi_{e_1} \in C$, musí platit, že $\varphi_{e_0} \neq \varphi_{e_1}$. Nyní definujme funkci $\varphi^{(2)}_{e_2}(x,y)$ následujícím způsobem:

$$\varphi^{(2)}_{e_2}(x,y) \simeq \begin{cases} \varphi_{e_1}(y) & \text{je-li } x \in K \\ \uparrow & \text{jinak} \end{cases}$$

Funkci $\varphi^{(2)}_{e_2}$ můžeme sestrojit například takto:

$$\varphi^{(2)}_{e_2}(x,y) \simeq \varphi^{(2)}_{e_1}(y) + o(\varphi_x(x))$$

Definujeme-li funkci $f(x) \simeq s^1_1(e_2, x)$, tedy $\varphi_{f(x)}(y) \simeq \varphi^{(2)}_{e_2}(x,y)$, dostáváme, že

$$\begin{array}{lllll} x \in K & \Rightarrow & \varphi_{f(x)} \simeq \varphi_{e_1} & \Rightarrow & \varphi_{f(x)} \in C \quad \Rightarrow \quad f(x) \in A_C \text{ a} \\ x \notin K & \Rightarrow & \varphi_{f(x)} \simeq \varphi_{e_0} & \Rightarrow & \varphi_{f(x)} \notin C \quad \Rightarrow \quad f(x) \notin A_C \end{array}$$

Dohromady tedy dostáváme požadovanou ekvivalenci: $x \in K \Leftrightarrow f(x) \in A_C$. Funkce f převádí množinu K na množinu A_C a podle s-m-n věty 2.3.22 jde o prostou ORF, a proto $K \leq_1 A_C$.

Pokud $\varphi_{e_0} \in C$, potom týž převod vede k $K \leq_1 \overline{A_C}$, stačí vyměnit role A_C a $\overline{A_C}$. Tvrzení věty je tím dokázáno. ■

Riceovu větu lze vnímat i jako důsledek věty o rekurzi, a proto se k ní ještě vrátíme. Riceova věta tedy ukazuje, že na otázky, které si můžeme klást o algoritmech, je buď triviální odpověď, nebo je nelze algoritmicky zodpovědět. Jde například o otázky, zda dvě Gödelova čísla x a y určují touž funkci, tedy zda $\varphi_x \simeq \varphi_y$. Neznamená to, že pokud dostaneme dva konkrétní programy, tak nejsme schopni dokázat, zda počítají či nepočítají tutéž funkci, například o základních funkcích $s(x)$ a $o(x)$ triviálně víme, že touž funkci nepočítají. To, co nám říká Riceova věta je, že nemůže existovat algoritmus, který za nás pro dané programy x a y rozhodne, zda tyto programy počítají tutéž funkci či nikoli.

Všimněme si, že je-li C netriviální třída ČRF (tedy neprázdná, ale současně neobsahuje všechny ČRF), pak nejen, že A_C není rekurzivní množina, ale důkaz Riceovy věty i dává návod, jak převést množinu K na A_C (pokud $\varphi_{e_0} \notin C$, kde φ_{e_0} označuje funkci, která není nikde definovaná) nebo $\overline{A_C}$ (pokud $\varphi_{e_0} \in C$). Pokud je A_C rekurzivně spočetná množina a $\varphi_{e_0} \notin C$, plyne z důkazu Riceovy věty, že je A_C navíc 1-úplná, protože $K \leq_1 A_C$.

Další zajímavé třídy ČRF, které nejsou rekurzivní a mnohé ani rekurzivně spočetné jsou třeba následující (pro úplnost jsou uvedeny i množiny K a K_0, i když to, že nejsou rekurzivní neplyne z Riceovy věty):

$$\begin{array}{rcc} K & = & \{x \mid \varphi_x(x)\downarrow\} \\ K_0 & = & \{\langle x,y\rangle_2 \mid x \in W_y\} = \{\langle x,y\rangle_2 \mid \varphi_y(x)\downarrow\} \\ K_1 & = & \{x \mid W_x \neq \emptyset\} \\ Fin & = & \{x \mid W_x \text{ je konečná}\} \\ Cof & = & \{x \mid \overline{W_x} \text{ je konečná}\} \\ Inf & = & \overline{Fin} = \{x \mid W_x \text{ je nekonečná}\} \\ Rec & = & \{x \mid W_x \text{ je rekurzivní}\} \\ Tot & = & \{x \mid \varphi_x \text{ je ORF}\} = \{x \mid W_x = \mathbb{N}\} \\ Ext & = & \{x \mid \varphi_x \text{ lze rozšířit na ORF}\} \\ Con & = & \{x \mid \varphi_x \text{ je konstantní funkce}\} \end{array}$$

Množiny K, K_0 a K_1 jsou rekurzivně spočetné, všechny jsou tedy i 1-úplné podle důkazu Riceovy věty a toho, že K a K_0 jsou 1-úplné množiny, ostatní množiny nejsou ani rekurzivně spočetné.

3.5 Cvičení

3.5.1 Rekurzivní množiny

1. Ukažte, že funkce $p(i)$, která pro zadané i vypíše i-té nejmenší prvočíslo (tj. $p(0) = 2, p(1) = 3, p(2) = 5, \ldots$), je primitivně rekurzivní.

3.5.2 Rekurzivně spočetné množiny a predikáty

2. Dokažte následující tvrzení. Predikát $P(x)$ je obecně rekurzivní právě tehdy, když existují primitivně rekurzivní predikáty $Q_1(x, y)$ a $Q_2(x, y)$ a platí:

$$P(x) \Leftrightarrow (\exists y)\ [Q_1(x, y)] \Leftrightarrow (\forall y)\ [Q_2(x, y)]$$

   Uvažte Postovu větu.

3. Ukažte, že množina

$$K_1 = \{x \mid W_x \neq \emptyset\}$$

   je rekurzivně spočetná.

4. Uvažte množinu

$$S = \{e \mid (\exists c)(\forall x)\ [\varphi_e(x)\downarrow \Rightarrow \varphi_e(x) = c]\}.$$

   Množina S tedy obsahuje Gödelova čísla funkcí, které jsou konstantní na těch vstupech, na kterých jsou definovány. Ukažte, že $\overline{S}$ je rekurzivně spočetná množina.

5. Rozhodněte, zda množina

$$S = \{\langle x, y\rangle \mid W_x \cap W_y = \emptyset\}$$

   je rekurzivní, své rozhodnutí zdůvodněte. V případě, že S není rekurzivní, rozhodněte, zda S či $\overline{S}$ je rekurzivně spočetná množina.

6. Rozhodněte, zda množina

$$S = \{x \mid (\forall w \in \{0,1\}^*)[M_x(w)\downarrow \Leftrightarrow M_x(w^R)\downarrow]\}$$

   je rekurzivní. Pokud S není rekurzivní, rozhodněte, zda S je rekurzivně spočetná množina, doplněk rekurzivně spočetné množiny, nebo ani jedno z toho.

   Upřesnění: *Slovo w^R je slovo w napsané pozpátku, aby x patřilo do S, musí se tedy stroj M_x s Gödelovým číslem x nad w i otočeným w^R zastavit, nebo se nezastavit ani nad jedním z těchto slov.*

7. Ukažte, že množina

$$S = \{e \mid |W_e| \geq e + 1\}$$

   je rekurzivně spočetná. Algoritmus ověřující pro dané e, zda patří do S stačí popsat pouze neformálně.

8. Rozhodněte, zda množina
$$S = \{x \mid |W_x| \leq 1\}$$
je rekurzivní, rekurzivně spočetná, doplněk rekurzivně spočetné množiny, nebo nepatří do žádné z těchto kategorií. Své rozhodnutí zdůvodněte.

9. Rozhodněte, zda množina
$$S = \Big\{x \mid \varphi_x \text{ je rostoucí ČRF}\}$$
je rekurzivní, rekurzivně spočetná, doplněk rekurzivně spočetné množiny, nebo nepatří do žádné z těchto kategorií. Své rozhodnutí zdůvodněte. *ČRF f je rostoucí pokud platí, že je rostoucí na vstupech, pro něž je definována, tj.:*
$$(\forall y_1, y_2 \in \mathbb{N})\Big[\Big((y_1 < y_2) \wedge f(y_1)\downarrow \wedge f(y_2)\downarrow\Big) \Rightarrow \Big(f(y_1) < f(y_2)\Big)\Big]$$
Speciálně funkce, která není definovaná pro žádný vstup, podle této definice rostoucí je.

10. Uvažte množinu
$$S = \{\langle e, n\rangle_2 \mid W_e \subseteq \{0, \ldots, n\}\}.$$
Rozhodněte, zda S nebo $\overline{S}$ je rekurzivně spočetná množina.

3.5.3 Převoditelnost a Riceova věta

U převodů si zkuste rozmyslet převod i v případech, kdy by se šlo odkázat na důkaz Riceovy věty.

2. Ukažte, že jsou-li A a B dvě netriviální rekurzivní množiny (tj. $A, B \neq \emptyset, \mathbb{N}$), pak $A \leq_m B$.

3. Ukažte, že jsou-li A a B dvě konečné množiny, pak $A \leq_1 B$ tehdy a jen tehdy, když $|A| \leq |B|$.

4. Operaci disjunktního sjednocení $\oplus$ množin A a B definujeme jako
$$A \oplus B = \{2a \mid a \in A\} \cup \{2b + 1 \mid b \in B\}.$$
Ukažte, že
   (a) $A \leq_m A \oplus B$, $B \leq_m A \oplus B$ a
   (b) je-li C množina, pro kterou platí, že $A \leq_m C$ i $B \leq_m C$, pak rovněž $A \oplus B \leq_m C$.

5. Ukažte, že množina K_1 je 1-úplná.

6. Ukažte, že $K \leq_1 Tot$, kde $Tot = \{x \mid W_x = \mathbb{N}\}$.

7. Ukažte, že $K \leq_1 Even$, kde $Even = \Big\{x \mid W_x = \{2i \mid i \in \mathbb{N}\}\Big\}$.

8. Uvažte množinu $EQ = \{\langle x, y\rangle_2 \mid \varphi_x \neq \varphi_y\}$. Ukažte, že platí jak $K \leq_1 EQ$, $\overline{K} \leq_1 EQ$. Rozmyslete si, že množina EQ ani její doplněk nejsou rekurzivně spočetné množiny.

9. Ukažte, že $K \leq_1 Inf$ (kde $Inf = \{x \mid W_x$ je nekonečná$\}$ je množina Gödelových čísel nekonečných množin definovaná na konci sekce 3.4).

10. Ukažte, že $K \leq_1 Fin$ (kde $Fin = \overline{Inf} = \{x \mid W_x \text{ je konečná}\}$ je množina Gödelových čísel konečných množin definovaná na konci sekce 3.4).

11. Rozmyslete si, že z předchozích dvou cvičení vyplývá, že množiny Fin ani Inf nejsou rekurzivně spočetné.

12. Ukažte, že pro množinu
$$S = \{e \mid |W_e| \geq e + 1\}$$
platí, že $K \leq_m S$.

13. Ukažte, že množina
$$S = \{\langle e, n\rangle_2 \mid W_e \subseteq \{0, \ldots, n\}\}.$$
není rekurzivní (*např. převeďte K na S nebo* $\overline{S}$).

Kapitola 4

Věta o rekurzi a její aplikace

4.1 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 4.1.1 (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 4.1.1 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 (*1. důkaz věty o rekurzi 4.1.1*) **:** Tento důkaz byl převzat z [8], 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

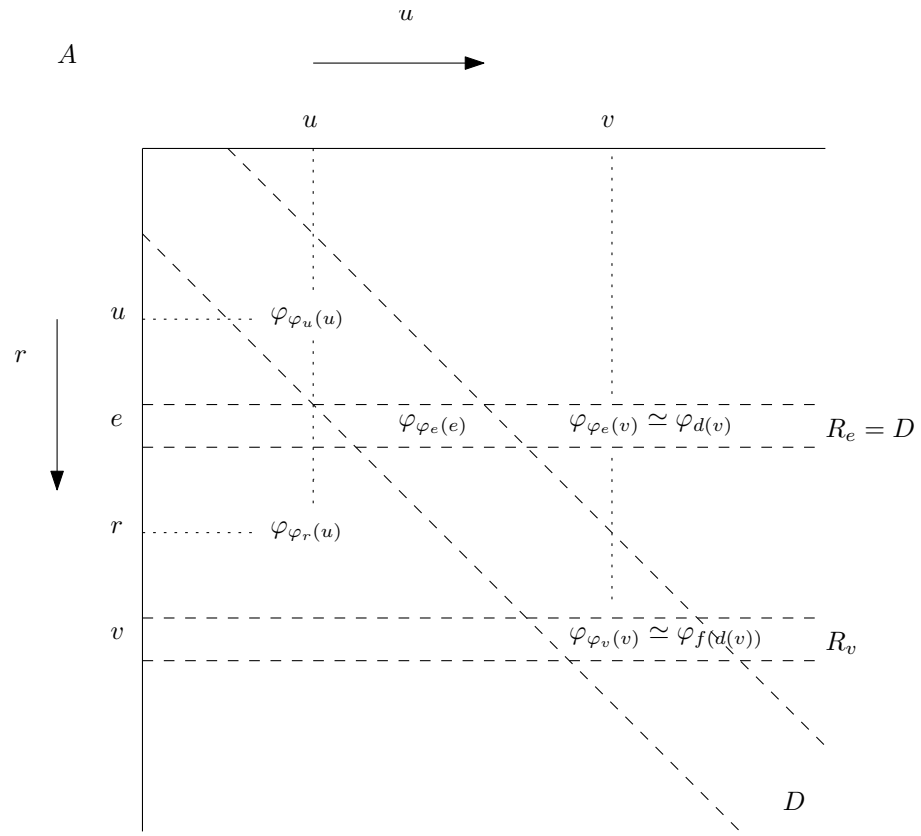
$$d_i = \alpha_{i,i}. \tag{4.1}$$

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

$$d_i = \alpha_{e,i} = \alpha_{i,i}. \tag{4.2}$$

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,



Obrázek 4.1: Ilustrativní obrázek k prvnímu důkazu věty o rekurzi.

například jako v-tý, tj. pro každé $i \in \mathbb{N}$ platí

$$d'_i = \alpha_{v,i}. \tag{4.3}$$

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 (4.1) a druhá rovnost plyne z (4.3). Také z toho podle (4.2) 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 4.1), 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.

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]

$$\varphi^{(2)}_{e_1}(u,x) \simeq \alpha_{u,u} \simeq \varphi_{\varphi_u(u)}(x).$$

[1]To odpovídá tomu, že $\alpha_{r,u}(x) \simeq \varphi^{(2)}_z(\varphi^{(2)}_z(r,u),x)$, kde $\varphi^{(2)}_z$ označuje univerzální ČRF pro funkce jedné proměnné.

[2]Gödelovo číslo e_1 bychom opět mohli určit s pomocí univerzální funkce $\varphi^{(2)}_z$:

$$\varphi^{(2)}_{e_1}(u,x) \simeq \varphi_{\varphi_u(u)}(x) \simeq \varphi^{(2)}_z(\varphi^{(2)}_z(u,u),x)$$

.

Podle s-m-n věty 2.3.22 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.

Důkaz (*2. důkaz věty o rekurzi 4.1.1*) **:** 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 (2.3.22) 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.“

[3]Označíme-li si univerzální ČRF pro funkce jedné proměnné pomocí $\varphi^{(2)}_z$, pak $\varphi^{(2)}_{e_1}(e,x) \simeq \varphi^{(2)}_z(f(\varphi^{(2)}_z(e,e)),x)$.

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 4.1.2 *Existuje prostá PRF g, která ke Gödelovu číslu funkce f určí její pevný bod.*

Důkaz : (*Uvažujeme první důkaz věty o rekurzi 4.1.1.*) 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. ■

Důsledek 4.1.3 *Každá ORF f má nekonečně mnoho pevných bodů.*

Důkaz : (*Uvažujeme první důkaz věty o rekurzi 4.1.1.*) 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 4.1.4

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 2.3.22 (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 4.1.1, 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 4.1.4 lze mimo jiné ukázat, že neexistuje třída ČRF C taková, pro kterou by platilo, že $K = \{e \mid \varphi_e \in C\}$, tento fakt ponecháme čtenáři jako jednoduché cvičení.

Podle druhého bodu důsledku 4.1.4 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 4.1.5 (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 4.1.1 (*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. ■

4.2 Důkaz Riceovy věty pomocí věty o rekurzi

Jako důsledek věty o rekurzi můžeme uvažovat i Riceovu větu (věta 3.4.6).

Důsledek 4.2.1 (Riceova věta) *Nechť C je libovolná třída částečně rekurzivních funkcí, potom je množina $A_C = \{e \mid \varphi_e \in C\}$ rekurzivní, právě když $C = \emptyset$ nebo C obsahuje všechny ČRF.*

Důkaz : Je-li C prázdná nebo obsahuje všechny ČRF, pak A_C je zřejmě rekurzivní, což jsme ukázali už v přímém důkazu Riceovy věty (věta 3.4.6). Předpokládejme tedy, že C není prázdná, ale neobsahuje ani všechny ČRF. Předpokládejme sporem, že A_C je rekurzivní. Protože A_C je neprázdná, existuje číslo $a \in A_C$, na druhou stranu A_C neobsahuje Gödelova čísla

všech funkcí, existuje také číslo $b \notin A_C$[4]. Nyní definujme funkci f následujícím způsobem:

$$f(x) = \begin{cases} a & x \notin A_C \\ b & x \in A_C \end{cases}$$

Funkce f je ORF, protože a, b jsou konkrétní čísla a charakteristická funkce χ_{A_C} množiny A_C je obecně rekurzivní z předpokladu, že A_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 $\varphi_n \in C$ nebo ne.

$$\begin{array}{lllllllll} \varphi_n \in C & \Rightarrow & n \in A_C & \Rightarrow & f(n) = b & \Rightarrow & f(n) \notin A_C & \Rightarrow & \varphi_{f(n)} \notin C \\ \varphi_n \notin C & \Rightarrow & n \notin A_C & \Rightarrow & f(n) = a & \Rightarrow & f(n) \in A_C & \Rightarrow & \varphi_{f(n)} \in C \end{array}$$

Z toho plyne, že $\varphi_n \in C$ právě když $\varphi_{f(n)} \notin C$, což je ale ve sporu s tím, že φ_n a $\varphi_{f(n)}$ označují tutéž funkci a C je třídou funkcí, tedy buď tato funkce do C patří nebo ne bez ohledu na to, jaké její Gödelovo číslo uvažujeme. Množina A_C tedy nemůže být rekurzivní. ■

4.3 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 C, pro kterou by platilo, že $K = \{e \mid \varphi_e \in C\}$.

[4]Poznamenejme, že z předpokládané rekurzivity A_C plyne, že její charakteristická funkce χ_{A_C} je obecně rekurzivní. S pomocí této charakteristické funkce bychom mohli efektivně najít $a \in A_C$ a $b \notin A_C$. Například pomocí funkcí

$$\begin{array}{rcl} a(x) & \simeq & \lambda x[\mu(y)[\chi_{A_C}(y) \simeq 1]] \text{ a} \\ b(x) & \simeq & \lambda x[\mu(y)[\chi_{A_C}(y) \simeq 0]], \end{array}$$

zde $a(x)$ vrátí nejmenší číslo patřící do A_C a $b(x)$ vrátí nejmenší číslo nepatřící do A_C.

Kapitola 5

Shrnutí části o vyčíslitelnosti a závěrečné poznámky

5.1 Shrnutí

Větou o rekurzi jsme ukončili část o vyčíslitelnosti, tedy té části, v níž jsme se zamýšleli nad tím, co algoritmy dokáží bez ohledu na to, kolik jim k tomu dáme času a prostoru. Zamysleme se nad tím, co jsme se v této části dozvěděli.

Dozvěděli jsme se, co je to algoritmus, pro nás je to program TS nebo odvození ČRF, víme, že tyto pojmy jsou stejně silné a připustíme-li Churchovu-Turingovu tezi, zachycují intuitivní pojem algoritmu.

Dozvěděli jsme se, že ne všechny úlohy je možné efektivně vyřešit, ty řešitelné jsme nazvali rekurzivní. Navíc jsme si ukázali způsob, jak o řadě problémů poznat, že nejsou efektivně rozhodnutelné - buď na ně převedeme problém zastavení nebo můžeme použít Riceovu větu.

5.2 Rozdíly mezi rekurzivními a rekurzivně spočetnými množinami

Jaké jsou rozdíly mezi rekurzivními a rekurzivně spočetnými množinami?

Rekurzivní množiny jsou ty, u nichž jsme schopni efektivně zodpovědět otázku náležení. U rekurzivně spočetných množin jsme schopni efektivně ověřit, že daný prvek do množiny patří, pokud nám někdo dá důkaz tohoto faktu.

Rekurzivní množiny jsme schopni efektivně a systematicky generovat, tj. můžeme vypsat jejich prvky v rostoucím pořadí. Rekurzivně spočetné množiny jsme schopni efektivně generovat, ale obecně nikoli systematicky, tj. můžeme vypsat prvky rekurzivně spočetné množiny, ale pouze bez ladu a skladu, na přeskáčku a může se stát, že některé prvky vypíšeme víckrát.

5.3 Rozdíly mezi ČRF, ORF a PRF

U částečně rekurzivní funkce f nejsme schopni říct, jestli se na vstupu x zastaví. U obecně rekurzivní funkce f víme, že se na vstupu x zastaví, ale nevíme, za jak dlouho, navíc nejsme ani schopni poznat, jestli je f obecně rekurzivní. U primitivně rekurzivní funkce f nejen víme, že se zastaví na vstupu x, ale také dokážeme předem odhadnout, jak dlouho jí to bude trvat, pokud dostaneme její primitivně rekurzivní odvození. To proto, že PRF nepotřebuje obecný

while cyklus a pokud máme program jen s *for* cykly, tak dokážeme spočítat, jak dlouho bude pracovat v závislosti na velikosti vstupu. Navíc pokud dostaneme primitivně rekurzivní odvození f, pak víme, že f je primitivně rekurzivní.

V dalším textu se budeme věnovat už tomu, jak dlouho trvá, než daný algoritmus spočítá zadanou úlohu a co jsme schopni spočítat, omezíme-li počet kroků TS nebo prostor pásky, který může popsat. Přesněji, pro danou (neklesající) funkci $f : \mathbb{N} \mapsto \mathbb{N}$ budeme studovat následující třídy problémů:

- $DTIME(f)$, jazyk L patří do $DTIME(f)$, pokud existuje TS M_e, pro který $L = L(M_e)$ a pro každý vstup délky n se M_e zastaví nejvýše po $f(n)$ krocích.
- $DSPACE(f)$, jazyk L patří do $DSPACE(f)$, pokud existuje TS M_e, pro který $L = L(M_e)$ a pro každý vstup délky n použije M_e nejvýše $f(n)$ políček pracovní pásky.

Později budeme definovat nedeterministický TS a odpovídající třídy NTIME a NSPACE, pro ně platí stejné tvrzení, které uvedeme nyní.

Lemma 5.3.1 *Je-li f PRF a $L \in DSPACE(f)$, pak predikát $P_L(x) = (w_x \in L)$ je PRP, tj. otázka náležení do L je primitivně rekurzivní. Totéž platí pro DTIME.*

Důkaz : Nechť M_e je TS, který popíše při práci nad vstupem délky n nejvýše $f(n)$ políček. Délka kódu jeho konfigurace je tedy omezená pomocí $h(n) = |Q| + 2 + (|\Sigma| + 1) \cdot f(n)$, kde Q je množina stavů M_e a Σ jeho pásková abeceda. Počet konfigurací je tedy omezen $g(n) = 2^{h(n)}$, kde g je zřejmě PRF. Kód výpočtu y stroje M_e je tedy omezen pomocí $y \leq v(n) = 2^{g(n)\cdot(h(n)+3)}$. Opět $v(n)$ je zřejmě PRF. Podle Kleeneho věty o normální formě můžeme psát

$$w_x \in L \Leftrightarrow R(\mu y[T(e, x, y)]),$$

kde $T(e, x, y)$ je primitivně rekurzivní predikát, který je splněn právě když y kóduje výpočet M_e nad x, a kde R je primitivně rekurzivní predikát, který jen z y vytáhne poslední konfiguraci a zjistí, jestli je přijímající. Protože délka y je omezená, můžeme psát ve skutečnosti

$$w_x \in L \Leftrightarrow R(\mu y < v(n)[T(e, x, y)]).$$

Protože omezená minimalizace je PR (stačí na ni for cyklus) a $v(n)$ je PRF, je predikát na pravé straně primitivně rekurzivní.

Pro čas je situace dokonce o trochu jednodušší, nemusíme odhadovat délku výpočtu, ale máme ji danou. ■

To znamená, že od této chvíle se již budeme pohybovat jen v rámci rozhodnutelných problémů, a to dokonce problémů rozhodnutelných pomocí primitivní rekurze. Též to zdůvodňuje uvedený rozdíl mezi ORF a PRF, kde ORF je funkce, u níž víme, že dopočítá, ale nevíme kdy, zatímco PRF je funkce, u níž víme, že dopočítá, a navíc víme i kdy.

Z Riceovy věty plyne, že otázka, jestli M_e je TS, jehož jazyk patří do $DTIME(f)$, je algoritmicky nerozhodnutelná (totéž platí pro *DSPACE* a *NTIME*/*NSPACE* definované později).

Část III

Složitost

Kapitola 6

Základní třídy problémů ve složitosti

6.1 Nedeterminismus a definice složitostních tříd

Od této chvíle začneme omezovat čas a prostor a zabývat se tím, co to znamená „rychlý“ nebo „prostorově nenáročný“ algoritmus. Vrátíme se opět k Turingovým strojům, neboť v jejich případě lze omezit čas či prostor celkem přímočaře a jednoduše, narozdíl od částečně rekurzivních funkcí. Od množin se tedy vrátíme opět k jazykům, které budeme obvykle uvažovat nad binární abecedou. V případě omezeného času a prostoru už velikost abecedy může hrát roli, protože vstup zakódovaný unárně je exponenciálně větší než tentýž vstup zakódovaný binárně. Na druhou stranu víme, že změníme-li základ logaritmu, tedy velikost abecedy, změní se už velikost vstupu jen konstantněkrát, protože $\log_a(n) = O(\log_b(n))$ pro každé $a, b > 1$, proto nám binární abeceda bude stačit, přičemž časem se budeme více věnovat jejímu rozdílu od unární abecedy. Na druhou stranu se abecedou budeme ve skutečnosti zabývat jen v případě formálních definic a většinou na ní nebude záležet.

V teorii složitosti obvykle odlišujeme rozhodovací problém od úlohy, v níž nás zajímá výstup méně triviální, než rozhodnutí ano/ne.

Definice 6.1.1

- Pokud u daného vstupu chceme pouze rozhodnout, má-li nějakou vlastnost, tedy očekáváme na naši otázku odpověď typu ano/ne, jedná se o *rozhodovací problém*, vstupu budeme říkat *instance problému*. Formálně je rozhodovací problém otázkou, zda x patří do jazyka $L \subseteq \{0,1\}^*$, kde L je množina řetězců kódujících instance s požadovanou vlastností.

- Pokud chceme pro daný vstup x nalézt y, které splňuje požadovanou vlastnost, jedná se o *úlohu*. Pokud má být toto y minimální nebo maximální (vzhledem k nějakému uspořádání) mezi řetězci, které splňují danou vlastnost, jedná se o *optimalizační úlohu* (tento pojem si budeme později definovat ještě trochu formálněji). Vstupu opět budeme říkat *instance úlohy*. Formálně je úloha binární relací $R \subseteq \{0,1\}^* \times \{0,1\}^*$. Jako vstup předáme x a hledáme y, pro nějž $(x, y) \in R$, nebo chceme dostat informaci o tom, že takové y neexistuje.

Například rozhodnutí o tom, je-li daný graf souvislý, je rozhodovacím problémem. Instancí tohoto problému je graf. Formálně bychom měli jako instanci tohoto problému uvážit libovolný binární řetězec, ale protože při vhodném kódování jednoduše poznáme, kóduje-li řetězec na vstupu graf, či nikoli, budeme za instanci považovat jen řetězce kódující graf. Obecně tedy budeme považovat za instanci problému nebo úlohy jen řetězce, kódující nějaký objekt našeho zájmu, čili syntakticky správný vstup. Úlohou může být hledání silně souvislých komponent orientovaného grafu. Instancí této úlohy je orientovaný graf, výstupem, tedy

hledaným řetězcem y, je seznam silně souvislých komponent grafu zakódovaného na vstupu řetězcem x. Optimalizační úlohou je třeba hledání maximálního toku v síti. Zde je instancí síť, tj. orientovaný graf, kapacity hran, zdroj a spotřebič.

Nyní můžeme definovat základní třídy problémů a úloh, tedy jazyků a relací, rozpoznatelných nebo řešitelných v omezeném čase či prostoru.

Definice 6.1.2 Nechť $f : \mathbb{N} \mapsto \mathbb{N}$ je libovolná funkce, pak definujeme následující třídy jazyků a relací (tj. problémů a úloh):

- $DTIME(f(n))$, jazyk $L \subseteq \{0,1\}^*$ patří do třídy $DTIME(f(n))$, pokud existuje TS M takový, že $L = L(M)$ a pro každé slovo x délky n skončí výpočet $M(x)$ nejvýše po $O(f(n))$ krocích.

- $DSPACE(f(n))$, jazyk $L \subseteq \{0,1\}^*$ patří do třídy $DSPACE(f(n))$, pokud existuje TS M takový, že $L = L(M)$ a pro každé slovo x délky n použije výpočet $M(x)$ během své práce nejvýše $O(f(n))$ buněk pracovní pásky.

- $DTIMEF(f(n))$, relace $R \subseteq \{0,1\}^* \times \{0,1\}^*$ patří do třídy $DTIMEF(f(n))$, pokud existuje TS M, jehož výpočet nad vstupem x délky n se zastaví po nejvýše $O(f(n))$ krocích, $M(x)$ přijme, právě když existuje y, pro nějž $(x, y) \in R$, a v tom případě obsahuje po ukončení výpočtu jeho páska toto slovo y[1].

Písmenko „D“ v názvu je zkratkou za „deterministický“, našemu modelu TS budeme nadále též říkat *deterministický Turingův stroj* (*DTS*), abychom jej odlišili od nedeterministického TS, který si zanedlouho zadefinujeme. Tradiční definice tříd *DTIME*, *DSPACE* a *DTIMEF* vyžaduje počet kroků či buněk nejvýš $f(n)$ a nikoli $O(f(n))$, dá se ukázat, že v případě Turingova stroje tak, jak jsme si jej definovali my, na multiplikativní konstantě nezáleží. Existují však i modely jako RAM, kde to neplatí, proto přímo v definici používáme $O(f(n))$. Při měření asymptotické složitosti algoritmů se totiž o konkrétní hodnoty multiplikativních konstant většinou nezajímáme, přestože mohou hrát podstatnou roli při praktické využitelnosti algoritmu.

Lemma 6.1.3 *Pro každou funkci $f : \mathbb{N} \mapsto \mathbb{N}$ platí, že $DTIME(f(n)) \subseteq DSPACE(f(n))$.*

Důkaz : Během své práce nestihne TS M nad vstupem x popsat víc buněk, než kolik na to má času, pracuje-li tedy M v čase $f(n)$, nestihne popsat víc než $f(n)$ buněk. Tvrzení platí i v případě, kdy $f(n) < n$, i když v tom případě musíme uvažovat jiný model TS, který umožňuje nepočítat do prostoru velikost vstupu. ■

Uvědomme si ještě jednou, že nám již nadále stačí jen primitivně rekurzivní prostředky, je-li funkce f omezující čas či prostor primitivně rekurzivní.

Lemma 6.1.4 *Je-li f ORF, pak třídy $DTIME(f(n))$ a $DSPACE(f(n))$ obsahují rekurzivní jazyky a třída $DTIMEF(f(n))$ obsahuje rekurzivní relace. Je-li f PRF, pak každý jazyk $L \in DSPACE(f(n))$ i každá relace $R \in DTIMEF(f(n))$ mají primitivně rekurzivní charakteristické funkce.*

Důkaz : Viz závěrečný komentář k části s vyčíslitelností. Dále už to rozvádět nebudeme, toto tvrzení je zde jen na dokreslení toho, v jakém světě se budeme nadále pohybovat. ■

Zbývá si vhodně zvolit funkce, které nám budou definovat pojmy „rychlého“ či „paměťově úsporného“ algoritmu. Z mnoha důvodů se k tomuto účelu hodí polynomy.

[1] Lze to také říci tak, že selektor pro predikát R je funkce spočitatelná v polynomiálním čase.

- Polynomy nerostou příliš rychle, pokud nějaký algoritmus pracuje v čase $O(n^3)$, pak i pro velká n může dopočítat ve snesitelném čase, případně si vystačí se snesitelným množstvím paměti.

- Jsou-li f a g polynomy, pak i jejich složení $f \circ g$ je polynom, což je pěkná vlastnost, která znamená, že použijeme-li v polynomiálním algoritmu podprogram, který je polynomiální, ve výsledku rovněž dostaneme polynomiální algoritmus.

- Silnější verze Churchovy-Turingovy teze, tvrdí, že každý „rozumný a obecný" výpočetní model lze na Turingově stroji simulovat s polynomiálním zpomalením či polynomiálním zvětšením potřebného prostoru a naopak že každý Turingův stroj lze simulovat s polynomiálním zpomalením či polynomiálním zvětšením potřebného prostoru v daném „rozumném a obecném" modelu. Pod pojmem rozumný si představíme takový model, který lze opravdu fyzicky zkonstruovat a reálně použít k výpočtu. Pod pojmem obecný si představíme takový model, který je schopen interpretovat jakýkoli algoritmus, tedy dle Churchovy-Turingovy teze jde o model, který je schopen simulovat jakýkoli Turingův stroj. Pro zatím známé modely tohoto typu silnější Churchova-Turingova teze platí, což říká, že pojmy algoritmu pracujícího v polynomiálním čase či v polynomiálním prostoru jsou nezávislé na použitém (dosud známém) „reálném" výpočetním modelu. Rozhodně jsou nezávislé na tom, v jakém běžném programovacím jazyce algoritmus implementujeme. Definice třídy P a myšlenka, že polynomy tvoří tu správnou třídu funkcí pochází od Cobhama [1] a tezi, podle které je třída P nezávislá na výpočetním modelu se říká Cobhamova-Edmondsova teze.

S pomocí polynomů si tedy můžeme definovat třídy P, PSPACE a PF následovně.

Definice 6.1.5

$$\begin{aligned} \mathrm{P} &= \bigcup_{i\in\mathbb{N}} DTIME(n^i) \\ \mathrm{PSPACE} &= \bigcup_{i\in\mathbb{N}} DSPACE(n^i) \\ \mathrm{PF} &= \bigcup_{i\in\mathbb{N}} DTIMEF(n^i) \end{aligned}$$

Třída P tedy obsahuje problémy rozhodnutelné v polynomiálním čase, třída PSPACE obsahuje problémy rozhodnutelné v polynomiálním prostoru a konečně třída PF obsahuje úlohy řešitelné v polynomiálním čase.

I k polynomům je však potřeba přistupovat opatrně, řekněme, že pro daný jazyk L sestrojíme dva algoritmy, A_1, který pracuje v čase $2^{2^{2^2}} \cdot n^{2^{10}}$ a A_2, který pracuje v čase $2^{0.0005n}$, který z nich byste si vybrali? Jistě A_2, protože ten bude pro všechny vstupy normální velikosti rychlejší, přitom ale A_1 ukazuje, že $L \in \mathrm{P}$ a podle toho by měl být A_1 „rychlý". Už bychom ale měli vědět, že O-notace je v tomto velmi zrádná, protože se zabývá nekonečnem, kterého však reálně nemůžeme dosáhnout. Pro nereálně velké vstupy by tedy skutečně byl algoritmus A_2 pomalejší než algoritmus A_1, ale reálně k tomu asi nedojde. Naštěstí pro většinu praktických problémů, které jsou řešitelné v polynomiálním čase, existují algoritmy s časem omezeným polynomy menších řádů, a tímto nedostatkem tedy netrpí. Fakt, že do třídy P patří i problémy, které jsou řešitelné jen v časech omezených polynomy s příliš velkými stupni by nás mohl inspirovat k tomu, abychom definovali třídu „rychlých" algoritmů jako například třídu problémů řešitelných v čase $O(n^{10})$, ale taková definice by se nám jen těžko odůvodňovala, například proč by už $O(n^{11})$ bylo pomalé? Proč když v „rychlém" algoritmu zavoláme „rychlý" algoritmus jako podprogram, dostaneme algoritmus „pomalý"? Navíc taková definice by byla závislá na výpočetním modelu, protože například existují problémy řešitelné v lineárním čase na dvoupáskovém TS, které vyžadují kvadratický čas na jednopáskovém TS. Současně platí, že neexistuje konstanta c, pro kterou by platilo $\mathrm{P} \subseteq DTIME(n^c)$, tj. žádný polynom není dost velký, aby se do něj schovaly všechny ostatní.

Třída PF obsahuje úlohy, které jsme schopni rychle, tedy v polynomiálním čase, vyřešit. Přesněji, je-li R binární relace patřící do PF, umíme pro dané x nalézt v polynomiálním čase y, pro které $(x, y) \in R$, pokud existuje, nebo rozhodnout, že neexistuje. Existuje však řada praktických úloh, které sice neumíme v polynomiálním čase vyřešit, ale pokud nám někdo dá řešení, uhodneme jej, získáme jej pomocí heuristiky či vnuknutím, umíme alespoň ověřit, zda jde o správné řešení. Takovým úlohám budeme říkat, že jsou *polynomiálně ověřitelné*. Třídu polynomiálně ověřitelných úloh nazveme NPF, kde NP znamená nedeterministicky polynomiální a F znamená funkce, název tak volíme proto, že jde o úlohy řešitelné nedeterministickým Turingovým strojem v polynomiálním čase, i když k tomu se dostaneme později.

Definice 6.1.6 Binární relace $R \subseteq \{0,1\}^* \times \{0,1\}^*$ patří do třídy NPF, pokud platí:

1. Existuje polynom $p(n)$, pro který platí, že v každé dvojici $(x, y) \in R$ je $|y| \leq p(|x|)$ a
2. ověření toho, zda $(x, y) \in R$ lze provést v polynomiálním čase.

První podmínka v definici 6.1.6 vlastně říká, že řešení k danému zadání nejsou moc dlouhá. Tato podmínka je podstatná proto, že složitost ověřování $(x, y) \in R$ je měřena v délce vstupu pro R, tedy v délce x a y. Kdybychom připustili, že y může mít délku exponenciální vzhledem k délce x, pak by polynomiální algoritmus neměl šanci y ani vypsat a ověření relace $(x, y) \in R$ by sice bylo polynomiálnív délce y, ale exponenciální v délce x, předpokládáme-li, že si ověřovač vstup y má alespoň přečíst. Proto se omezíme jen na úlohy, kde možná řešení nejsou moc dlouhá.

Poznamenejme, že neplatí přímo PF $\subseteq$ NPF, neboť lze najít úlohy, které jsou sice polynomiálně řešitelné, ale nejsou polynomiálně ověřitelné. Na druhou stranu pro běžné a praktické úlohy platí, že pokud jsou polynomiálně řešitelné, jsou i polynomiálně ověřitelné a z tohoto hlediska můžeme brát PF jako podtřídu NPF. Existuje však spousta úloh, o kterých víme, že jsou polynomiálně ověřitelné, ale domníváme se, že nejsou polynomiálně řešitelné. Příkladem takové úlohy může být úloha obchodního cestujícího.

Úloha obchodního cestujícího
Instance : Je dáno n měst, vzdálenosti mezi nimi a číslo d.
Cíl : Najít pořadí měst, při kterém může obchodní cestující projet všechna města a projezdit při tom vzdálenost nejvýše d.

Pokud dostaneme pořadí měst, tedy kandidáta na řešení této úlohy, jsme schopni rychle spočítat vzdálenost, kterou při tomto pořadí obchodní cestující najezdí, tedy jestli splňuje náš cíl. Z toho plyne, že tato úloha patří do NPF. Na druhou stranu neznáme způsob, jak vhodné pořadí najít v polynomiálním čase. A nejen to, kdyby takový způsob existoval, byli bychom schopni vyřešit všechny úlohy z NPF v polynomiálním čase a znamenalo by to tedy, že pojmy polynomiální řešitelnosti a polynomiální ověřitelnosti z praktického hlediska splynuly.. Je to samozřejmě možné, ale většina badatelů se domnívá, že tomu tak není, vede nás k tomu i každodenní zkušenost. Obvykle je lehčí ověřit, zda dané řešení je správné, než řešení nějakého problému vymyslet. Podobnou zkušenost zažívají studenti ve zkoušovkém období pravidelně, neboť i zde je snazší zkoušet než být zkoušený, protože je snazší ověřit, zda to, co poslouchám jako zkoušející, dává smysl, než je pro zkoušeného říkat něco, co má hlavu a patu.

Tyto úvahy lze přenést i na problémy. Existuje řada praktických problémů, které sice neumíme vyřešit v polynomiálním čase, ale dá-li nám někdo důkaz kladné odpovědi, umíme

jej opět ověřit, k tomu samozřejmě potřebujeme, aby důkaz sám nebyl příliš dlouhý. Uvažme opět obchodního cestujícího, nyní jako problém.

Problém obchodního cestujícího
Instance : Je dáno n měst, vzdálenosti mezi nimi a číslo d.
Otázka : Existuje pořadí měst, v němž je má obchodní cestující projet, aby najetá vzdálenost byla nejvýš d?

Nikdo nezná polynomiální algoritmus, který by byl schopen zodpovědět tuto otázku, ale pokud nám někdo dá pořadí měst, tedy důkaz či certifikát kladné odpovědi na otázku, dokážeme tento certifikát ověřit v polynomiálním čase. U problémů, které jsou jen rozhodovací verzí odpovídající úlohy, jako je tomu v tomto případě, je obvykle certifikátem řešení této úlohy. To ale neznamená, že certifikát vždy musí mít tuto vlastnost, může jít o jakýkoli řetězec, který nějakým způsobem dosvědčuje kladnou odpověď. Příklad takového certifikátu, který není současně řešením uvidíme i zanedlouho, až budeme ukazovat, že problémy řešitelné nedeterministickým Turingovým strojem v polynomiálním čase mají polynomiálně ověřitelné důkazy.

Třídu problémů s polynomiálně ověřitelným důkazem nazveme rovnou NP, což znamená *nedeterministicky polynomiální*, název pochází z druhého způsobu definice, který si ukážeme vzápětí. Protože si také ukážeme, že oba způsoby definují tutéž třídu problémů, přidržíme se od začátku tradičního názvu NP, i když k následující definici by se lépe hodil pojem třídy jazyků s *polynomiálně ověřitelným důkazem*.

Definice 6.1.7 Jazyk $L \subseteq \{0,1\}^*$ patří do třídy NP, pokud existuje polynom p a jazyk $B \in \mathrm{P}$ a pokud pro každé $x \in \{0,1\}^*$ platí, že

$$x \in L \Leftrightarrow (\exists y) \left[|y| \leq p(|x|) \text{ a } (x,y) \in B \right].$$

Neformálně řečeno jazyk L patří do třídy NP, pokud existuje polynomiální algoritmus (daný jazykem B), který je schopen ověřit, že polynomiálně dlouhý certifikát y dosvědčuje fakt $x \in L$. Je opět nutné uvažovat pouze polynomiálně dlouhé certifikáty, neboť složitost ověřování se měří vzhledem k délce x i certifikátu y.

Zřejmě platí, že $\mathrm{P} \subseteq \mathrm{NP}$, protože pokud jsme problém schopni rozhodnout v polynomiálním čase, nepotřebujeme se na certifikát vůbec podívat. Ještě poznamenejme, že našeho obchodního cestujícího ve skutečnosti zajímá pořadí měst, při němž je nacestovaná vzdálenost nejkratší. K tomu, abychom o nějakém pořadí měst mohli rozhodnout, že jím daná vzdálenost je nejkratší, bychom museli umět rozhodnout, že neexistuje žádné lepší pořadí, což odpovídá rozhodnutí, že neexistuje pořadí dávající vzdálenost menší než daná mez d, jde tedy o negaci našeho problému obchodního cestujícího. Tyto úvahy naznačují, že nalézt nejkratší pořadí může být mnohem těžší, než nalézt pořadí, které vede ke vzdálenosti menší než daná mez d. Proto často u přirozeně optimalizačních úloh uvažujeme jejich takto upravené verze.

Všimněme si dále analogie rozdílu mezi třídami P a NP s rozdílem mezi rekurzivními a rekurzivně spočetnými jazyky či množinami. Podle věty 3.3.5 je množina A rekurzivně spočetná, pokud existuje rekurzivní predikát $P(x,y)$, pro který platí, že

$$A = \{x \mid (\exists y) P(x,y)\}.$$

Podle naší definice třídy NP patří jazyk L do třídy NP pokud existuje predikát $R(x,y)$ rozhodnutelný v polynomiálním čase a polynom p, pro které platí, že

$$L = \{x \mid (\exists y)[|y| \leq p(|x|) \text{ a } R(x,y)]\}.$$

Víme, že existuje problém, který je rekurzivně spočetný, ale není rekurzivní, například problém zastavení, ale nevíme, jestli existuje také problém, který patří do NP, ale nepatří do P.

Třída NPF tedy obsahuje úlohy, u kterých nejsme vždy schopni (za současného stavu znalostí) najít v polynomiálním čase řešení, ale pokud dostaneme kandidáta na řešení, jsme alespoň schopni v polynomiálním čase ověřit, jde-li skutečně o řešení úlohy, či nikoli. To znamená, že pokud použijeme heuristiku, nebo řešení jinak uhodneme, můžeme aspoň poznat, jestli jsme dospěli ke správnému výsledku. Podobně třída NP obsahuje problémy, které sice neumíme všechny za současného stavu znalostí v polynomiálním čase zodpovědět (ty, co jsou v P, tak pochopitelně ano), ale pokud nám někdo dá polynomiálně velký důkaz kladné odpovědi, dovedeme jej v polynomiálním čase ověřit.

Možnost hádání řešení nebo certifikátu, čili existenční kvantifikaci, je možné na Turingově stroji zachytit pomocí nedeterminismu.

Definice 6.1.8 *Nedeterministický Turingův stroj* je pětice $M = (Q, \Sigma, \delta, q_0, F)$, jejíž prvky mají týž význam jako v případě deterministického TS s tím rozdílem, že

$$\delta : Q \times \Sigma \mapsto \mathscr{P}(Q \times \Sigma \times \{R, N, L\}).$$

Tj. 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.

- *Výpočet NTS* je posloupnost konfigurací $K_0, \ldots, K_t$, kde K_0 je počáteční konfigurace a pro každou konfigurace K_i, $i \in \{0, \ldots, t-1\}$ platí, že K_{i+1} lze vytvořit z K_i pomocí přechodové funkce vhodnou volbou přechodu. Tj. v každém kroku si NTS vybere, kterou z možností zvolí a tu aplikuje.

- *Výpočet NTS* je přijímající, pokud končí v přijímajícím stavu.

- Řekneme, že NTS M *přijme slovo* x, pokud existuje výpočet NTS M nad x, který je přijímající.

- Jazyk slov přijímaných NTS M označíme pomocí $L(M)$.

- Nechť $f : \mathbb{N} \mapsto \mathbb{N}$ je libovolná funkce. Řekneme, že NTS M pracuje v čase $f(n)$, pokud každý výpočet nad slovem x délky n buď skončí po nejvýš $f(n)$ krocích, nebo je nekonečný.

- Řekneme, že NTS M pracuje v prostoru $f(n)$, pokud každý výpočet nad slovem x délky n buď použije nejvýš $f(n)$ buněk pásky, nebo je nekonečný.

Nedeterministický Turingův stroj se tedy od deterministického liší jen tím, že v dané chvíli umožňuje použít víc přechodů. Náš způsob definice výpočtu NTS předpokládá, že NTS 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 NTS daný vstup přijal. Posloupnosti výběrů či „uhodnutí“, které nevedou k přijetí, nás nezajímají a můžeme pro jednoduchost předpokládat, že se nezastaví. Výpočet NTS si však lze představit i tak, že v každém kroku NTS použije všechny možné přechody a nachází se tak vždy ve všech dostupných konfiguracích najednou, podobně pracuje i nedeterministický konečný automat. V tomto případě nás zajímá, jestli do určitého počtu kroků je jedna z těchto konfigurací přijímající. V obou případech si také můžeme výpočet NTS představit jako strom, jehož hrany jsou orientované

směrem od kořene k listům. Kořen odpovídá počáteční konfiguraci a z každé konfigurace K vedou hrany do konfigurací, do nichž lze z K přejít pomocí přechodové funkce. V prvním případě odpovídá výpočet jedné větvi tohoto stromu, ve druhém případě odpovídá celému stromu.

Poznamenejme, že nedeterministický Turingův stroj není reálný výpočetní model, nelze jej skutečně zkonstruovat, a také se nepředpokládá, že by se na něj vztahovala silnější verze Churchovy-Turingovy teze. Na druhou stranu je ovšem pravda, že libovolný nedeterministický Turingův stroj lze simulovat na deterministckém a tedy i nedeterministické Turingovy stroje jsou výpočetním modelem ekvivalentním Turingovým strojům. Pro nás je však nedeterministický Turingův stroj zejména abstrakcí, která zachycuje možnost hádání v TS. Podobně jako v deterministickém případě můžeme nyní definovat třídy jazyků přijímaných nedeterministickým TS v omezeném čase či prostoru.

Definice 6.1.9 Nechť $f : \mathbb{N} \mapsto \mathbb{N}$ je libovolná funkce, pak definujeme následující třídy jazyků a relací (tj. problémů a úloh):

- $NTIME(f(n))$, jazyk $L \subseteq \{0,1\}^*$ patří do třídy $NTIME(f(n))$, pokud existuje NTS M, který pracuje v čase $O(f(n))$ a $L = L(M)$.

- $NSPACE(f(n))$, jazyk $L \subseteq \{0,1\}^*$ patří do třídy $NSPACE(f(n))$, pokud existuje NTS M, který pracuje v prostoru $O(f(n))$ a $L = L(M)$.

Podobně jako polynomiální DTS odpovídají třídě P, polynomiální NTS odpovídají třídě NP, proto jsme ji nazvali nedeterministicky polynomiální.

Lemma 6.1.10 *Platí, že:*

$$\mathrm{NP} = \bigcup_{i \in \mathbb{N}} NTIME(n^i)$$

Důkaz : Předpokládejme nejprve, že $L \in$ NP, to znamená, že existuje polynom p a jazyk $B \in$ P, pro které platí, že $x \in L$ právě když existuje y, $|y| \leq p(|x|)$, pro které $(x, y) \in B$. Nedeterministický TS M', který bude přijímat L, bude pracovat ve dvou fázích. V první fázi zapíše na vstupní pásku za slovo x slovo y, tato fáze je nedeterministická a pro každé slovo y, $|y| \leq p(|x|)$ existuje výpočet M', který jej napíše. Na zápis y stačí čas $p(|x|)$. Ve druhé fázi bude M' simulovat práci Turingova stroje M, který rozpoznává jazyk B, na vstupu (x, y), přičemž přijme, pokud $(x, y) \in B$. Zřejmě $L = L(M')$ a M' pracuje v polynomiálním čase.

Nyní předpokládejme, že $L \in \bigcup_{i \in \mathbb{N}} NTIME(n^i)$. To znamená, že $x \in L$ právě když existuje polynomiálně dlouhý výpočet nedeterministického Turingova stroje M, kde $L = L(M)$, jenž x přijme. V každém kroku tohoto výpočtu vybírá M z několika možných instrukcí, nechť řetězec y kóduje právě to, které instrukce byly v každém kroku vybrány. Řetězec y má délku nejvýš $p(|x|)$ pro nějaký polynom p, protože M pracuje v polynomiálním čase a možností jak pokračovat z dané konfigurace podle přechodové funkce je jen konstantně mnoho. Simulací M s použitím instrukcí daných dle y můžeme deterministicky ověřit, zda y kóduje přijímající výpočet. Řetězec y tedy může sloužit jako polynomiálně dlouhý certifikát kladné odpovědi. ■

Podívejme se ještě, jak spolu souvisí polynomiálně ověřitelné úlohy a problémy s polynomiálně ověřitelným důkazem.

Lemma 6.1.11 *Nechť R je relace, která patří do* NPF *a definujme jazyk $L = \{x \mid (\exists y) R(x, y)\}$, potom $L \in$ NP.*

Důkaz : Důkaz je zřejmý, protože y, pro které $R(x, y)$, je polynomiálně velkým certifikátem faktu, že $x \in L$. ■

Toto tvrzení lze v jistém smyslu i otočit.

Lemma 6.1.12 *Nechť $L \in$ NP, pak existuje relace $R \in$ NPF, pro kterou platí, že*

$$L = \{x \mid (\exists y)\ R(x, y)\}.$$

Důkaz : Dvojice (x, y) bude patřit do relace R, pokud y je certifikátem toho, že $x \in L$. ■

Lemma 6.1.13 *Je-li f ORF, pak třídy $NTIME(f(n))$ a $NSPACE(f(n))$ obsahují rekurzivní jazyky. Je-li f PRF, pak každý jazyk $L \in NSPACE(f(n))$ má primitivně rekurzivní charakteristickou funkci.*

Důkaz : Důkaz nebudeme rozvádět, platí zde tytéž argumenty jako v případě deterministického prostoru a času. ■

6.2 Savičova věta: PSPACE = NPSPACE

V této části si ukážeme, že v případě prostoru má analogie otázky, zda P = NP, kladnou odpověď. Přesněji, definujeme-li si třídu

$$\text{NPSPACE} = \bigcup_{i \in \mathbb{N}} NSPACE(n^i),$$

ukážeme si, že NPSPACE = PSPACE. Tento fakt je důsledkem Savičovy věty, která ve skutečnosti říká, že pro „rozumné“ funkce $f(n)$ platí $NSPACE(f(n)) \subseteq DSPACE((f(n))^2)$. Pro začátek si budeme pod rozumnou funkcí představovat polynomy a během důkazu se dostaneme k tomu, jaké vlastnosti musí funkce mít, aby na ni bylo lze použít Savičovu větu.

U definice nedeterministického Turingova stroje jsme zmínili, že jeho výpočet si lze představit jako strom konfigurací, jehož kořen odpovídá počáteční konfiguraci a od vrcholu vedou hrany ke konfiguracím následujícím podle přechodové funkce. Chceme-li simulovat nedeterministický Turingův stroj na deterministickém, mohli bychom projít tento strom buď průchodem do šířky nebo do hloubky a hledat dosažitelnou přijímající konfiguraci. Větve, které použijí více prostoru, než je povoleno, nebo v nichž se zopakuje konfigurace a tedy se zacyklí, není třeba dále procházet. Nejhlubší větev v tomto stromě, do které má tedy smysl se dívat, má hloubku rovnou počtu různých konfigurací, které používají vymezený prostor.

Lemma 6.2.1 *Nechť M je libovolný TS (ať už deterministický či nedeterministický), který pracuje v prostoru $f(n)$, pak existuje konstanta c_M (jejíž hodnota je závislá na stroji M), pro kterou platí, že počet konfigurací M je nejvýš $2^{c_M f(n)}$.*

Důkaz : Nechť $M = (Q, \Sigma, \delta, q_0, F)$. Konfigurace se skládá ze slova na pásce, polohy hlavy v rámci tohoto slova a stavu, v němž se stroj M nachází. Délka slova na pásce je omezená $f(n)$, počet různých poloh hlavy v rámci tohoto slova je $f(n)$ a počet stavů je $|Q|$. Počet konfigurací je tedy shora omezen pomocí

$$|\Sigma|^{f(n)} f(n) |Q| = 2^{f(n)\log_2|\Sigma|} 2^{\log_2 f(n)} 2^{\log_2|Q|} = 2^{f(n)\log_2|\Sigma| + \log_2 f(n) + \log_2|Q|} \leq 2^{f(n)\cdot(\log_2|\Sigma| + 1 + \log_2|Q|)}.$$

Stačí tedy zvolit $c_M = (\log_2|\Sigma| + \log_2|Q| + 1)$. ■

Lemma 6.2.1 tedy ukazuje, že průchod do hloubky ani do šířky stromem výpočtu si nevystačí s polynomiálním prostorem, pokud už funkce $f(n)$ je polynomem. V důkazu Savičovy

věty proto musíme postupovat jinak. Čeho si můžeme na stromu výpočtu všimnout je fakt, že se tam řada konfigurací může opakovat v různých větvích, tímto nedostatkem netrpí *konfigurační graf*, který také zachycuje možné výpočty Turingova stroje M nad vstupem x a který si definujeme následujícím způsobem.

Definice 6.2.2 Je-li dán NTS M a jeho vstup x, definujeme orientovaný graf $G_{M,x} = (V, E)$ následovně:

- V obsahuje všechny konfigurace stroje M při výpočtu nad vstupem x a
- jsou-li K_1 a K_2 dvě konfigurace M, pak $(K_1, K_2) \in E$, právě když existuje přechod podle přechodové funkce M, který lze použít na K_1 a který z ní vytvoří K_2.

Pokud je M ve skutečnosti DTS, pak pro libovolný vstup x je graf $G_{M,x}$ orientovaná cesta[2], ale pokud M využívá nedeterminismu, může být graf $G_{M,x}$ komplikovanější. Z lemmatu 6.2.1 plyne, že pokud M pracuje v prostoru $f(n)$, pak $|V| \leq 2^{c_M f(|x|)}$ a $|E| \leq 2^{2c_M f(|x|)}$ pro vhodnou konstantu c_M.

Označme si nyní počáteční konfiguraci M při výpočtu nad x pomocí K_0^x. Předpokládejme, že M má jedinou přijímající konfiguraci a označme ji jako K_F. Tento předpoklad není nijak omezující, zřejmě můžeme každý NTS upravit tak, že jeho přijímající konfigurace bude jednoznačná a už jsme toho několikrát využili. Můžeme totiž předpokládat, že pokaždé, než M přijme, vymaže obsah pásky, poté se vrátí na levý okraj vymezeného pracovního prostoru a přejde do jediného přijímajícího stavu. NTS M přijme x, právě když v grafu $G_{M,x}$ existuje orientovaná cesta z K_0^x do K_F. Stačí nám tedy popsat algoritmus, který existenci takové cesty ověří a vystačí si přitom s prostorem, který bude jen $O((f(n))^2)$. Uvažme také, že nemáme prostor na konstrukci a uložení konfiguračního grafu $G_{M,x}$ do paměti, tento graf máme zadaný jen implicitně tím, že pro dané dvě konfigurace K_1, K_2 jsme schopni s pomocí přechodové M funkce rozhodnout, zda v $G_{M,x}$ vede hrana z K_1 do K_2, či nikoli.

Odhlédneme-li od toho, jak jsme přišli ke grafu $G_{M,x}$, zajímá nás ve skutečnosti řešení následující grafové úlohy. Na vstupu očekáváme orientovaný graf $G = (V, E)$ zadaný nikoli seznamem sousedů, ale maticí sousednosti zadanou implicitně pomocí funkce $hrana(i, j)$, která je schopna rozhodnout, zda $(i, j) \in E$ v polynomiálním čase a jen s konstantním prostorem (nepočítáme-li do prostoru velikost vstupu i, j). Dále jsou zadané vrcholy s a t. Ptáme se, zda je možné s pracovním prostorem $O((\log_2(|V|))^2)$ rozhodnout, zda v G existuje cesta z s do t.

Pro jednoduchost uvážíme nejprve případ, kdy $f(n)$ je polynom a poté se teprve budeme zamýšlet nad tím, nejde-li důkaz Savičovy věty zobecnit i pro jiné funkce.

Věta 6.2.3 (Savičova - formulace pro polynomy) NPSPACE = PSPACE

Důkaz : Zřejmě platí, že PSPACE $\subseteq$ NPSPACE, protože každý DTS je jen zvláštním případem NTS. Ukažme nyní, že NPSPACE $\subseteq$ PSPACE. Předpokládejme, že $L \in$ NPSPACE, tedy existuje k, pro něž $L \in NSPACE(n^k)$. Nechť M je NTS přijímající L v prostoru n^k, který má navíc jedinou přijímající konfiguraci. Ukážeme, že existuje DTS M', který přijímá L v prostoru $O(n^{2k})$. Nechť $G_{M,x} = (V, E)$ je konfigurační graf výpočtů M nad x, popíšeme algoritmus, který bude testovat to, jestli v $G_{M,x}$ existuje cesta z počáteční konfigurace K_0^x do jediné přijímající konfigurace K_F, a vystačí si přitom s prostorem $O((\log_2(|V|))^2)$. Algoritmus popíšeme pomocí funkce *Dosažitelná(i, K_1, K_2)*, která zjistí, zda z konfigurace K_1 je konfigurace K_2 dosažitelná v nejvýše 2^i krocích Turingova stroje M.

[2]Toto není úplně přesné, o cestu jde jen v případě, že výpočet $M(x)$ je konečný, v případě nekonvergujícího výpočtu jde o laso, tedy cestu s cyklem na konci. Nás však zajímají jen konečné a navíc jen přijímající větve.

Algoritmus 6.2.4 Dosažitelná(i, K_1, K_2)

Vstup: Celé číslo i, konfigurace K_1 a K_2
Výstup: **True**, je-li K_2 dosažitelná v grafu $G_{M,x}$ z konfigurace K_1 pomocí nejvýš 2^i kroků. **False** jinak.

```
 1: if i = 0
 2: then
 3:    if (K_1, K_2) ∈ E or K_1 = K_2
 4:    then
 5:       return true
 6:    else
 7:       return false
 8:    endif
 9: endif
10: for each K ∈ V
11: do
12:    if Dosažitelná(i − 1, K_1, K) and Dosažitelná(i − 1, K, K_2)
13:    then
14:       return true
15:    endif
16: done
17: return false
```

Protože TS M přijímá jazyk L v prostoru n^i, existuje podle lemmatu 6.2.1 konstanta c_M, pro kterou je počet konfigurací M nejvýš $2^{c_M n^k}$, tedy voláním Dosažitelná($G_{M,x}$, $c_M n^k$, K_0^x, K_F) zjistíme, je-li v $G_{M,x}$ dosažitelná v $2^{c_M n^k}$ krocích, tedy, je-li vůbec dosažitelná. Korektnost tohoto algoritmu je zřejmá, proto ji probereme jen ve stručnosti: Pokud je $i = 0$, znamená to, že K_2 musí být z K_1 dosažitelná v nejvýš jednom $(= 2^0)$ kroku, jinými slovy, buď (K_1, K_2) je hranou v $G_{M,x}$, nebo $K_1 = K_2$. Pokud je $i > 0$, je K_2 dosažitelná z K_1 v nejvýš 2^i krocích, pokud existuje konfigurace K taková, že K je z K_1 dosažitelná v nejvýš 2^{i-1} krocích a K_2 je z K dosažitelná v nejvýš 2^{i-1} krocích, což ověříme dvojicí rekurzivních volání na řádku 12.

Zbývá odhadnout, jak velký prostor požaduje volání funkce Dosažitelná. Prostor $O(n^{2k})$, kterého chceme dosáhnout nám neumožňuje uložit si v paměti celý graf $G_{M,x}$, protože už jen počet jeho vrcholů je exponenciální v n. Nám ale ve skutečnosti stačí umět otestovat pro dvě dané konfigurace K_1 a K_2, zda $(K_1, K_2) \in E$, tedy zda lze z konfigurace K_1 přejít do konfigurace K_2 pomocí přechodové funkce stroje M. K tomuto testu nám tedy stačí přechodová funkce, jejíž velikost je konstantní a nezávislá na $n = |x|$. V cyklu *for* na řádcích 10 až 16 potřebujeme postupně generovat všechny konfigurace. Na uložení jedné konfigurace nám stačí $c_M n^k$ bitů, stačí tedy generovat všechny binární řetězce této délky a vybírat si ty, které kódují konfigurace. K zakódování konfigurace v podobě, s níž by se Turingovu stroji dobře pracovalo, můžeme ve skutečnosti potřebovat více bitů, než jen $c_M n^k$, kde c_M je konstanta z tvrzení lemmatu 6.2.1. Použijeme-li například totéž kódování jako v důkazu věty 2.3.15, postačí nám však stále $O(n^k)$ bitů (kde v „O“ notaci se zde schovává i konstanta c_M). Můžeme tedy předpokládat, že do konstanty c_M jsou již nároky na uložení kódu konfigurace započítány.

Jedna instance funkce Dosažitelná tedy vyžaduje prostor pro uložení K_1, K_2, K a i, na všechny tyto proměnné stačí prostor $c_M n^k$. Navíc potřebujeme jen bity pro uložení odpovědí z rekurzivních volání, které už prostorovou složitost nenaruší. Abychom mohli v cyklu generovat konfigurace K, musíme však vědět, jak velký prostor rezervovat pro jednu konfiguraci, potřebujeme tedy umět označit $c_M n^k$ buněk, máme-li vstup x délky n, a při tom si musíme vy-

stačit s prostorem $O(n^k)$, tj. nemůžeme použít příliš prostoru navíc oproti tomu, který máme vyznačit. Necháme čtenáři na rozmyšlení, že to můžeme v případě polynomů učinit.

Hloubka rekurze je omezena pomocí $c_M n^k = O(n^k)$, protože to je počáteční hodnota i, které předáváme funkci jako parametr a v každém dalším voláním toto i snížíme o jedna. Dohromady tedy dostaneme, že celkový prostor, který k vykonání Dosažitelná($G_{M,x}$, $c_M n^i$, K_0^x, K_F) potřebujeme, je velký $O(n^k \cdot n^k) = O(n^{2k})$. Protože funkce Dosažitelná je deterministická, vyžaduje její volání deterministický prostor $O(n^{2k})$. Nebylo by také obtížné na základě této funkce vytvořit DTS M', který by vyžadoval týž prostor a přijímal jazyk L. Z toho plyne, že $L \in DSPACE(O(n^{2k})) \subseteq$ PSPACE. ■

Důkaz věty 6.2.3 nikde nevyužíval žádných zvláštních vlastností polynomů, můžeme jej tedy zobecnit a dostat tak obecnou Savičovu větu. Ukázali jsme vlastně, že k otestování toho, jestli v obecném orientovaném grafu $G = (V, E)$ vede cesta z daného vrcholu s do vrcholu t, stačí prostor $O((\log_2 |V|)^2)$, pokud do tohoto prostoru nepočítáme prostor nutný k uložení grafu. Pokud tedy místo polynomu n^k použijeme obecnou funkci $f(n)$, dostaneme tvrzení, že $NSPACE(f(n)) \subseteq DSPACE(f(n)^2)$.

Funkce $f(n)$ ale přece jen nemůže být úplně libovolná, protože musíme být schopni odhadnout, kolik místa potřebujeme pro uložení jedné konfigurace, musíme tedy umět spočítat hodnotu funkce $f(n)$ v prostoru $O(f(n))$ (v našem případě hodnotu $c_M n^k$ v prostoru $O(c_M n^k)$), je-li vstup Turingova stroje počítajícího hodnotu f zakódovaný unárně, tj. na vstupu je řetězec x délky n, ne přímo hodnota n zakódovaná binárně. Například, je-li $x = 010010001$, pak $n = 9$, ale na vstupu stroje počítajícího hodnotu $f(|x|)$ bude přímo $x = 010010001$ a ne hodnota 9 zakódovaná binárně pomocí čtyř bitů. I výsledek výpočtu hodnoty funkce $f(|x|)$ očekáváme zapsaný unárně, tj. očekáváme na výstupu přímo řetězec délky $f(|x|)$, tedy označený prostor $f(|x|)$, protože potom přímo vidíme, kolik buněk je potřeba alokovat pro zápis konfigurace. Kdyby byl výstup zakódovaný binárně, byla by alokace komplikovanější. Je-li však vstup i výstup zapsán unárně, odpovídá to přirozeně způsobu použití.

Definice 6.2.5 Funkce je *f vyčíslitelná v prostoru*[3] $O(f(n))$, pokud k ní existuje Turingův stroj M_f, který pracuje v prostoru $O(f(n))$ a při výpočtu nad vstupem $x = 1^n$ je po ukončení jeho činnosti na výstupu zapsán řetězec $y = 1^{f(n)}$.

Ne každá funkce f je vyčíslitelná v prostoru $O(f(n))$, nicméně pro běžné funkce to platí.

Dosud jsme předpokládali, že do prostorové složitosti se počítá i vstup, tedy $f(n) \geq n$. Chceme-li se zabývat prostorem menším než lineárním, musíme náš model Turingova stroje upravit tak, aby umožnil velikost vstupu do použitého prostoru nepočítat. V tom případě obvykle uvažujeme tři pásky - vstupní, pracovní a výstupní. Ze vstupní je možno jen číst, ale nelze na ni zapisovat. Na pracovní pásce je možné číst i zapisovat. Na výstupní pásku lze pouze zapisovat a hlava se může pohybovat jen jedním směrem, tj. po zapsání symbolu se pohne doprava a nemůže se vrátit k tomu, co už zapsala. Do prostorové složitosti se pak počítá jen obsah pracovní pásky. Použijeme-li tento model, můžeme tvrzení zobecnit pro všechny funkce $f(n) \geq \log_2 n$. Toto omezení je nutné proto, že v rámci konfigurace kódujeme polohu hlavy v rámci vstupu, k čemuž potřebujeme $\log_2 n$ bitů. Na druhou stranu platí, že i funkce $O(\log_2 n)$ je vyčíslitelná v prostoru $O(\log_2 n)$.

Nyní si tedy můžeme zformulovat obecnou Savičovu větu:

Věta 6.2.6 (Savičova - obecné znění) *Nechť $f : \mathbb{N} \mapsto \mathbb{N}$ je funkce vyčíslitelná v prostoru $O(f(n))$,*

[3] Poznamenejme, že obvykle se uvažuje poněkud striktnější pojem prostorové konstruovatelnosti, který vyžaduje aby existoval TS M, který při výpočtu nad řetězcem $x = 1^n$ využije přesně $f(n)$ buněk. V případě Turingových strojů však mezi prostorovou konstruovatelností a vyčíslitelností v prostoru $O(f(n))$ není rozdíl a nás multiplikativní konstanty příliš nezajímají.

pro kterou platí, že $f(n) \geq \log_2 n$ *pro všechna* $n \in \mathbb{N}$. *Potom platí, že*

$$NSPACE(f(n)) \subseteq DSPACE(f(n)^2).$$

To, co nám umožňuje ukázat tuto větu, je samozřejmě fakt, že prostor je možné používat opakovaně. Jednotlivé větve rekurzivních volání funkce Dosažitelná tedy používají týž prostor. V případě času je situace velmi odlišná, protože jednou využitý čas nám nikdo nevrátí. I proto je otázka, zda P = NP mnohem komplikovanější a nejenže dosud nikdo nebyl schopen najít na ni odpověď, ale převládá názor, že P ≠ NP, a tedy že zmíněný rozdíl mezi časem a prostorem je opravdu zásadní.

6.3 Cvičení

1. Odhadněte časové nároky funkce Dosažitelná z důkazu Savičovy věty 6.2.3. Pracuje v polynomiálním čase vzhledem k velikosti grafu $G_{M,x}$?

2. Ukažte, že platí $DSPACE(\log_2 n) \subseteq P$.
   Nápověda: Kolik různých konfigurací může mít TS pracující v prostoru $O(\log_2 n)$*?*

3. Nechť $L \in NSPACE(n)$ je libovolný jazyk, ukažte, že existuje konstanta c_L (závislá na konkrétním jazyku L), pro kterou platí, že $L \in DTIME(2^{c_L n})$. Lze toto tvrzení zobecnit i pro jiné funkce f, než $f(n) = n$? (Uvažte funkce f vyčíslitelné v prostoru $O(f(n))$.)

Kapitola 7

Polynomiální převoditelnost a úplnost

7.1 Polynomiální převoditelnost a existence NP-úplného problému

S principem převoditelnosti stejně jako s úplností jsme se setkali už v případě 1-převoditelnosti a m-převoditelnosti. Tehdy nás zajímala převoditelnost, která zachovávala algoritmickou řešitelnost a k tomu nám stačilo, že převádějící funkce byla obecně rekurzivní. Nyní chceme, aby převádějící funkce zachovávala polynomiální řešitelnost, a proto je přirozené požadovat od ní samé, aby byla spočitatelná v polynomiálním čase.

Definice 7.1.1 Řekneme, že jazyk A je *polynomiálně převoditelný* na jazyk B, což zapíšeme pomocí $A \leq_m^p B$, pokud existuje funkce $f : \{0,1\}^* \mapsto \{0,1\}^*$, která je spočitatelná v polynomiálním čase (tj. relace $\{(x, f(x)) \mid x \in A\} \in \mathrm{PF}$) a pro kterou platí, že

$$x \in A \Leftrightarrow f(x) \in B.$$

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 spočitatelnosti převádějící funkce v polynomiálním čase. Pokud je $A \leq_m^p B$, znamená to, že rozhodnutí, zda $y \in B$, je alespoň tak těžké, jako rozhodnutí, zda $x \in A$, nebereme-li v úvahu polynomiální zpomalení způsobené výpočtem převádějící funkce f. Polynomiální převoditelnost má řadu vlastností společných s m-převoditelností.

Lemma 7.1.2 (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 spočitatelná 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, pak $g \circ f$ převádí A na C, jsou-li f i g spočitatelné 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 spočitatelná 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 7.1.3 *V poznámce 3.4.5 jsme navíc k 1-převoditelnosti a 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 množina K, tedy diagonála problému zastavení, neboť zřejmě platí, že $\overline{K} \leq_T^p K$, ale jistě neplatí $\overline{K} \leq_m^p K$, protože to by muselo platit i $\overline{K} \leq_m K$, víme přitom, že to není možné, protože $\overline{K}$ není rekurzivně spočetná množina. 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 1-převoditelnosti, i zde budou předmětem našeho zájmu ty nejtěžší problémy, přičemž nás nejvíce zajímají problémy z třídy NP.

Definice 7.1.4 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 $\mathrm{P} = \mathrm{NP}$. 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 $\mathrm{P} \subsetneq \mathrm{NP}$, pak existují i jazyky, které sice nejsou NP-úplné, ale patří do $\mathrm{NP} \setminus \mathrm{P}$, 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[1].

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

[1] Jde např. o rozhodnutí, zda je daná formule φ v konjunktivně normální formě ekvivalentní dané formuli ψ v disjunktivně normální formě.

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 následujícho lemmatu.

Lemma 7.1.5 *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 7.1.2 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.

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 7.1.6 *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. 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 rekurzivně spočetné jazyky, *CERT* v sobě kóduje všechny jazyky z třídy NP, pokud se omezíme na hodnoty t, které jsou polynomiální v délce $|x|$. Díky větě 7.1.6 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.

Větě, která ukazuje NP-úplnost praktického problému 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 [2] a Leonid A. Levin v [6]. Cook však ve skutečnosti používal Cookovu převoditelnost (viz poznámka 7.1.3), pojem polynomiální převoditelnosti, který používáme my, pochází od Richarda M. Karpa z [5]. Jako první praktický NP-úplný problém se obvykle uvažuje splnitelnost formule v konjunktivně normální formě. U nás se však jako výchozí problém vžilo Kachlíkování, proto i my jím začneme.

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 7.1.7 (Cookova-Levinova) *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 lemmatu 6.1.10 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 $\delta(q_1, a) = \emptyset$, 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[2]. Většinu zmíněných předpokladů jsme již dříve použili, například při konstrukci univerzálního TS. Bez újmy na obecnosti tedy můžeme předpokládat, že M splňuje uvedené podmínky.

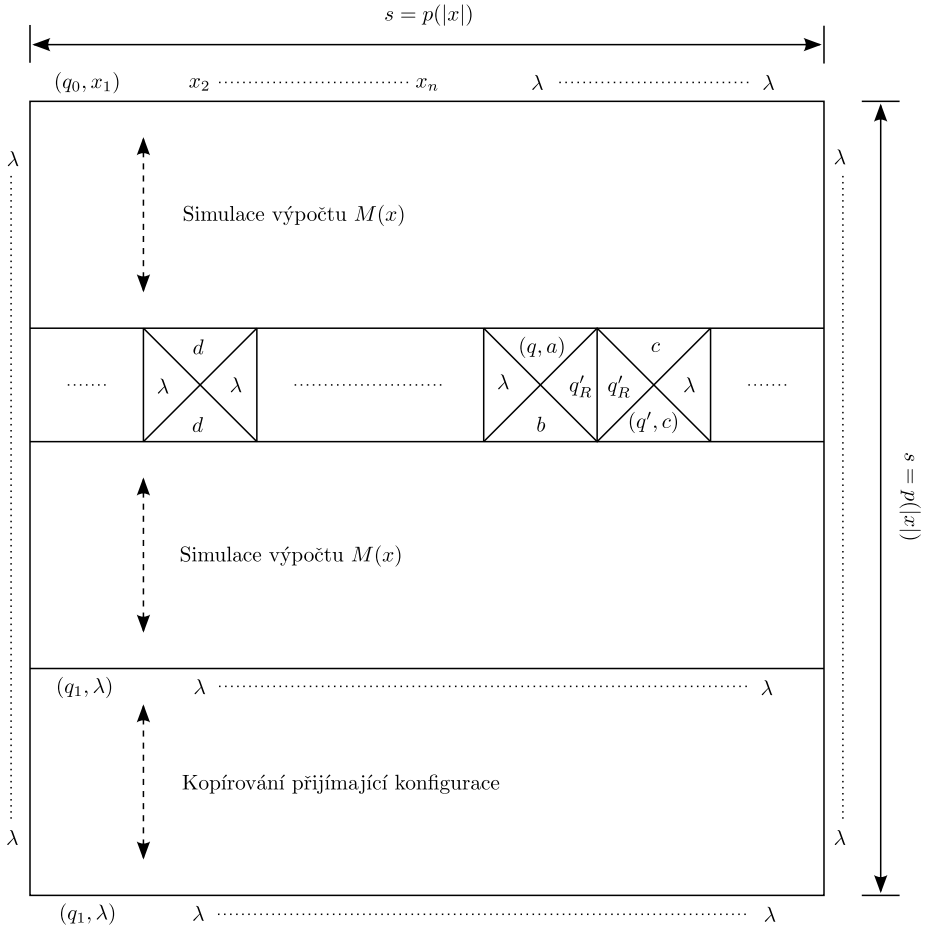
Nechť x je instance problému A, popíšeme, jak z M, polynomu p a instance x vytvořit instanci Kachlíkování, pro kterou bude platit, že v ní existuje přípustné vykachlíkování, právě když existuje přijímající výpočet $M(x)$, tj. $M(x)$ přijme.

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 7.1. 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

[2]Ve skutečnosti by se hodilo mít vstup od 2. políčka, aby mohl TS M snadno poznat začátek vstupu dle prázdného 1. políčka, ale můžeme také předpokládat, že si TS M 1. znak označí v první instrukci. Každopádně jde o technický detail, kterým se nemusíme příliš zabývat.



Obrázek 7.1: 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.

na pásce, nad kterým se vyskytuje hlava, přičemž q označuje stav, ve kterém se M nachází, ostatní políčka konfigurace budou obarvena barvou odpovídající symbolu na daném místě.

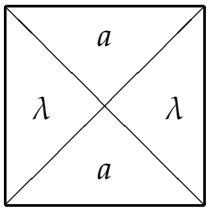
Velikost jedné strany čtvercové sítě položíme $s = p(|x|)$, všimněme si, že $p(|x|)$ omezuje nejen počet konfigurací v přijímajícím výpočtu, ale i počet buněk, které stihne M při výpočtu nad x popsat, tedy i délku slova na pásce v konfiguraci.

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)



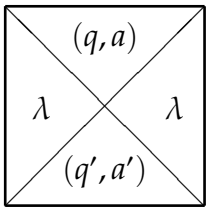
Tento kachlík bude odpovídat těm místům konfigurace, která přechodová funkce nemění, protože se nad nimi nevyskytuje hlava v předchozí ani v následující konfiguraci.

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 $\delta(q, a) = \emptyset$, 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)

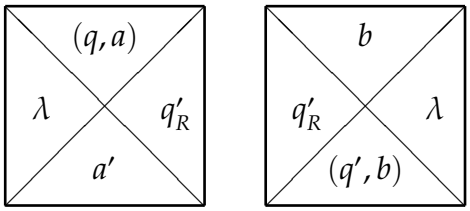


To odpovídá tomu, že pokud nevykonáváme žádný pohyb, pouze změníme stav a přepíšeme

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:

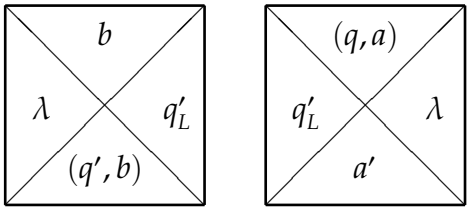
(III) (IV)



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:

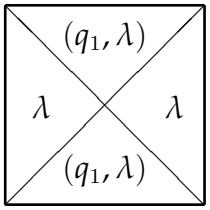
(V) (VI)



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 *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)). Tj. 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. ■

Poznamenejme ještě, že při důkazu NP-úplnosti Kachlíkování bychom mohli použít i přímo certifikátovou definici 6.1.7, důkaz by však byl složitější, protože bychom museli pomocí kachlíků zajistit „hádání" vhodného certifikátu místo toho, abychom to nechali na příslušném nedeterministickém Turingově stroji. V tomto případě je tedy vhodnější použít model nedeterministického Turingova stroje, ačkoli pro důkaz toho, že nějaký problém (např. i Kachlíkování) patří do NP se hodí spíš certifikátová definice 6.1.7 než ekvivalentní definice pomocí nedeterministických Turingových strojů na základě lemmatu 6.1.10. Ve chvíli, kdy totiž chceme o nějakém problému ukázat, že patří do třídy NP, nezačneme obvykle konstruovat nedeterministický Turingův stroj, ale popíšeme, jak vypadá snadno ověřitelný certifikát. Často je navíc certifikát shodný s řešením úlohy, která přirozeně odpovídá danému rozhodovacímu problému.

7.2 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 (3*DM*) 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 [4].

7.2.1 Splnitelnost formulí v KNF

Než si řekneme, v čem spočívá problém splnitelnosti, zadefinujeme si pojem konjunktivně normální formy.

Definice 7.2.1 *Literál* je buď proměnná nebo její negace, proměnnou x nazveme *pozitivním literálem* a její negaci $\overline{x}$ 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 *konjunktivně normální formě* (*KNF*), pokud je konjunkcí klauzulí, například $(x \vee \overline{y} \vee \overline{z}) \wedge (\overline{x} \vee y) \wedge (y \vee z) \wedge \overline{x}$.

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*.

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?

Nyní můžeme přistoupit k důkazu toho, že problém splnitelnosti je NP-úplný.

Věta 7.2.2 *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:

$$\begin{aligned} V &= \{(p,q) \mid \text{ spodní barva } T_p \text{ se neshoduje s horní barvou } T_q\} \\ H &= \{(p,q) \mid \text{ pravá barva } T_p \text{ se neshoduje s levou barvou } T_q\} \end{aligned}$$

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.

$$\begin{aligned} U_j &= \{k \mid \text{ horní barva } T_k \text{ se shoduje s barvou horního okraje } S \text{ v } j\text{-tém sloupci}\} \\ B_j &= \{k \mid \text{ spodní barva } T_k \text{ se shoduje s barvou spodního okraje } S \text{ v } j\text{-tém sloupci}\} \\ L_j &= \{k \mid \text{ levá barva } T_k \text{ se shoduje s barvou levého okraje } S \text{ na } j\text{-tém řádku}\} \\ R_j &= \{k \mid \text{ pravá barva } T_k \text{ se shoduje s barvou pravého okraje } S \text{ na } j\text{-tém řádku}\} \end{aligned}$$

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

$$\alpha_{i,j} = \bigwedge_{k=1}^{|K|} \bigwedge_{l=k+1}^{|K|} (\overline{x_{i,j,k}} \vee \overline{x_{i,j,l}}).$$

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 $(\overline{x_{i,j,k}} \vee \overline{x_{i,j,l}})$ 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

$$\gamma_{i,j} = \bigwedge_{(p,q) \in V} (\overline{x_{i,j,p}} \vee \overline{x_{i+1,j,q}}).$$

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

$$\delta_{i,j} = \bigwedge_{(p,q) \in H} (\overline{x_{i,j,p}} \vee \overline{x_{i,j+1,q}}).$$

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

$$\varepsilon_u = \bigwedge_{j=1}^{s} (\bigvee_{k \in U_j} x_{1,j,k}).$$

Formule ε_u je splněna, pokud je na každém políčku v prvním řádku kachlík s barvou shodnou s příslušným horním okrajem.

(VII) Definujeme formuli

$$\varepsilon_b = \bigwedge_{j=1}^{s} (\bigvee_{k \in B_j} x_{s,j,k}).$$

Formule ε_b je splněna, pokud je na každém políčku v nejspodnějším řádku kachlík s barvou shodnou s příslušným spodním okrajem.

(VIII) Definujeme formuli

$$\varepsilon_l = \bigwedge_{j=1}^{s} (\bigvee_{k \in L_j} x_{j,1,k}).$$

Formule ε_l je splněna, pokud je na každém políčku v nejlevějším sloupci kachlík s barvou shodnou s příslušným levým okrajem.

(IX) Definujme formuli

$$\varepsilon_r = \bigwedge_{j=1}^{s} \Big(\bigvee_{k \in R_j} x_{j,s,k} \Big).$$

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. ■

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.

Splnitelnost formule v 3KNF ($3SAT$)
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 $3SAT$ 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 $2SAT$ je už polynomiálně řešitelný.

Věta 7.2.3 *Problém* $3SAT$ *je* NP*-úplný.*

Důkaz : Problém $3SAT$ je jen zvláštním případem problému SAT a z téhož důvodu jako v případě problému SAT, patří i $3SAT$ do třídy NP. Těžkost problému $3SAT$ 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

$$\alpha_j = (e_1 \vee y_1^j \vee y_2^j)(e_1 \vee y_1^j \vee \overline{y_2^j})(e_1 \vee \overline{y_1^j} \vee y_2^j)(e_1 \vee \overline{y_1^j} \vee \overline{y_2^j}).$$

- Pokud $k = 2$, pak nechť y^j je nově přidaná proměnná, jež se nevyskytuje v ψ a definujme

$$\alpha_j = (e_1 \vee e_2 \vee y^j)(e_1 \vee e_2 \vee \overline{y^j}).$$

- 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

$$\alpha_j = (e_1 \vee e_2 \vee y_1^j)(\overline{y_1^j} \vee e_3 \vee y_2^j)(\overline{y_2^j} \vee e_4 \vee y_3^j)...(\overline{y_{k-3}^j} \vee e_{k-1} \vee e_k).$$

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 $\overline{y^j}$, 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 $(\overline{y_i^j} \vee e_{i+2} \vee y_{i+1}^j)$ 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 $C_j = (e_1 \vee ... \vee e_k)$ 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.

7.2.2 Vrcholové pokrytí v grafu

V této části si ukážeme NP-úplnost problému vrcholového pokrytí v grafu.

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. $\lvert S\rvert \leq k$ a $(\forall e \in E)[S \cap e \neq \emptyset]$.)

Věta 7.2.4 *Problém vrcholového pokrytí je* NP*-úplný.*

Důkaz : Fakt, že problém $VP \in$ 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 $3SAT$. Nechť formule $\varphi = C_1 \wedge \cdots \wedge C_m$ s proměnnými $U = \{u_1, \ldots, u_n\}$ je instancí problému $3SAT$, 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 $V_i = \{u_i, \overline{u_i}\}$ a $E_i = \{\{u_i, \overline{u_i}\}\}$, 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, tedy trojúhelník, $R_j = (V'_j, E'_j)$.

$$\begin{aligned} V'_j &= \{a_1[j], a_2[j], a_3[j]\} \\ E'_j &= \{\{a_1[j], a_2[j]\}, \{a_1[j], a_3[j]\}, \{a_2[j], a_3[j]\}\} \end{aligned}$$

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

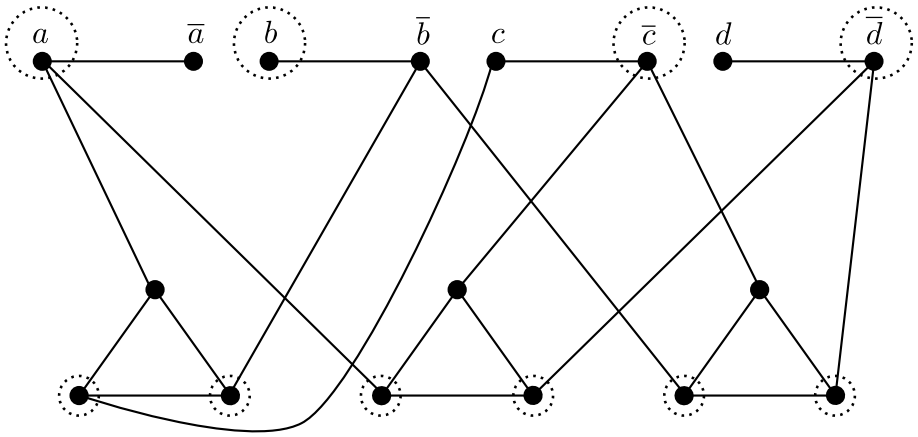
$$\begin{aligned} V &= (\bigcup_{i=1}^{n} V_i) \cup (\bigcup_{i=1}^{m} V'_i) \\ E &= (\bigcup_{i=1}^{n} E_i) \cup (\bigcup_{i=1}^{m} E'_i) \cup (\bigcup_{i=1}^{m} E''_i) \end{aligned}$$

Příklad konstrukce můžete nahlédnout v obrázku 7.2.

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 $\lvert S\rvert \geq n + 2m$, a protože současně $\lvert S\rvert \leq n + 2m$, musí ve skutečnosti platit, že S pokrývá právě jeden vrchol z každého T_i a právě dva vrcholy z každého R_i. Definujme ohodnocení $t : U \mapsto \{0, 1\}$, které dané proměnné $u_i \in U$ přiřadí

$$t(u_i) = \begin{cases} 1 & u_i \in S \\ 0 & \overline{u_i} \in S \end{cases}$$



Obrázek 7.2: Příklad převodu formule $\varphi = (a \vee \overline{b} \vee c)(a \vee \overline{c} \vee \overline{d})(\overline{b} \vee \overline{c} \vee \overline{d})$ na instanci vrcholového pokrytí, tedy graf na obrázku s $k = 2.3 + 4 = 10$. Jednotlivé trojúhelníky dole odpovídají zmíněným klauzulím, zakroužkované vrcholy určují vrcholové pokrytí velikosti k, které odpovídá splňujícímu ohodnocení $a = b = 1$, $c = d = 0$.

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 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 $x_j = \overline{u_i} \in S$ 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 $t : U \mapsto \{0, 1\}$ a definujme množinu $S = \{u_i \mid t(u_i) = 1\} \cup \{\overline{u_i} \mid t(u_i) = 0\}$, 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.

7.2.3 Hamiltonovská kružnice v grafu

I další problém, kterému se budeme věnovat, bude grafový.

Hamiltonovská kružnice (*HK*)
Instance : Graf $G = (V, E)$.
Otázka : Existuje v grafu G cyklus vedoucí přes všechny vrcholy?

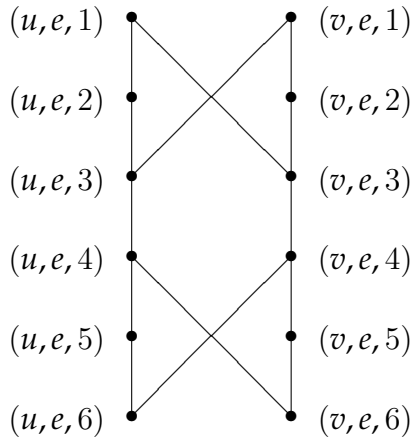
Věta 7.2.5 *Problém hamiltonovské kružnice je* NP*-úplný.*

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:

$$\begin{aligned} V' &= \{a_1, \ldots, a_k\} \cup \Big(\bigcup_{e \in E} V'_e\Big) \\ E' &= \Big(\bigcup_{e \in E} E'_e\Big) \cup \Big(\bigcup_{v \in V} E'_v\Big) \cup \Big(\bigcup_{v \in V} E''_v\Big) \end{aligned}$$

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.

7.2.4 Trojrozměrné párování

Dalším problémem, jehož těžkost si ukážeme, bude trojrozměrné párování (3D Matching).

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 7.2.6 *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.

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 7.3):

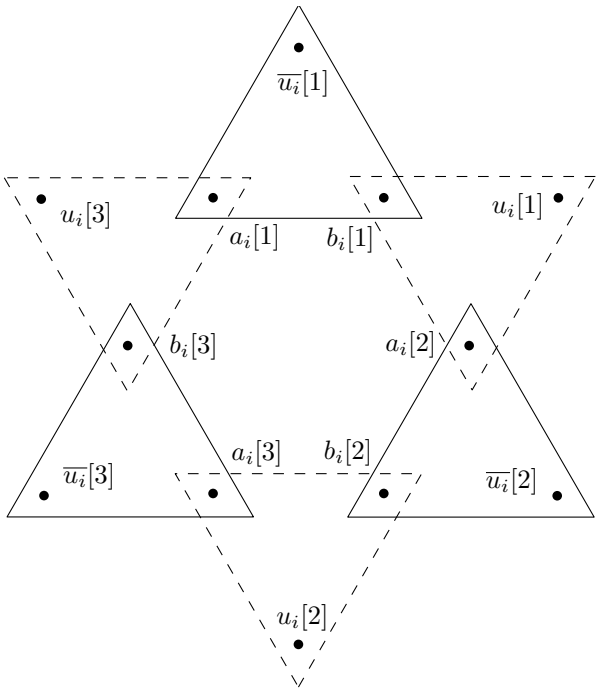
$$\begin{aligned} T_i^t &= \{(\overline{u}_i[j], a_i[j], b_i[j]) \mid 1 \leq j \leq m\} \\ T_i^f &= \{(u_i[j], a_i[(j+1) \bmod m], b_i[j]) \mid 1 \leq j \leq m\} \\ T_i &= T_i^t \cup T_i^f \end{aligned}$$

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 $(u_i[j], s_1[j], s_2[j])$



Obrázek 7.3: Ukázka množin T_i^t (plnou čarou) a T_i^f (čárkovaně) pro případ $m = 3$.

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

$$G = \{(u_i[j], g_1[k], g_2[k]), (\overline{u}_i[j], g_1[k], g_2[k]) \mid 1 \leq k \leq m(n-1), 1 \leq i \leq n, 1 \leq j \leq m\}$$

Tím je konstrukce dokončena, ještě si na závěr shrňme její výsledek:

$$\begin{aligned}
W &= \{u_i[j], \overline{u}_i[j] \mid 1 \leq i \leq n, 1 \leq j \leq m\} \\
X &= \{a_i[j] \mid 1 \leq i \leq n, 1 \leq j \leq m\} \cup \\
&\quad \cup \{s_1[j] \mid 1 \leq j \leq m\} \\
&\quad \cup \{g_1[j] \mid 1 \leq j \leq m(n-1)\} \\
Y &= \{b_i[j] \mid 1 \leq i \leq n, 1 \leq j \leq m\} \cup \\
&\quad \cup \{s_2[j] \mid 1 \leq j \leq m\} \\
&\quad \cup \{g_2[j] \mid 1 \leq j \leq m(n-1)\} \\
M &= \left(\bigcup_{i=1}^{n} T_i\right) \cup \left(\bigcup_{j=1}^{m} S_j\right) \cup G
\end{aligned}$$

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 $(\overline{u_i}[j], s_1[j], s_2[j])$, 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 $\overline{u_i}$.

Pokud je naopak φ splnitelná, zkonstruujeme perfektní párování následovně. Nechť $t : U \mapsto \{0, 1\}$ 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

$$M' = \Big(\bigcup_{t(u_i)=1} T_i^t \Big) \cup \Big(\bigcup_{t(u_i)=0} T_i^f \Big) \cup \Big(\bigcup_{j=1}^{m} \{(z_j[j], s_1[j], s_2[j])\} \Big) \cup G',$$

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.

7.2.5 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.

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 7.2.7 *Problém Loupežníci je* NP*-úplný.*

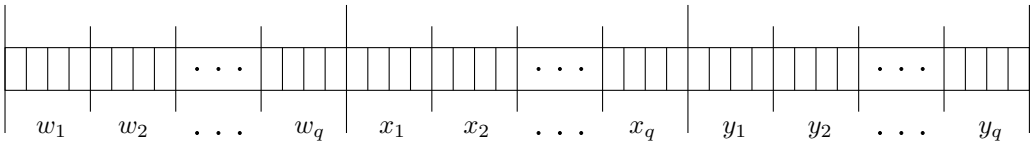
Důkaz : Fakt, že tento problém patří do třídy NP, plyne z toho, že pro zadanou množinu A' je jistě snadné ověřit, zda obsahuje prvky poloviční ceny. Těžkost tohoto problému ukážeme převodem z trojrozměrného párování. Uvažme instanci $3DM$, tedy množinu $M \subseteq W \times X \times Y$, kde $|W| = |X| = |Y| = q$. 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 7.4.



Obrázek 7.4: 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:

$$\begin{aligned} s(b_1) &= 2\left(\textstyle\sum_{i=1}^{k} s(a_i)\right) - B \\ s(b_2) &= \left(\textstyle\sum_{i=1}^{k} s(a_i)\right) + B \end{aligned}$$

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. ■

7.3 Cvičení

Důkazy NP-úplnosti

1. Ukažte, že problémy Kliky a Nezávislé množiny jsou stejně těžké a že oba patří do NP.

Klika
Instance : Graf $G = (V, E)$ a přirozené číslo k.
Otázka : Obsahuje G jako podgraf úplný podgraf (kliku) s alespoň k vrcholy?

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 $\lvert S \rvert \geq k$?

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.

Orientovaná hamiltonovská kružnice (OHK)
Instance : Orientovaný graf $G = (V, E)$.
Otázka : Existuje v G cyklus, který projde každý vrchol právě jednou?

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?

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.

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? Tj. existuje permutace $\pi : \{1, \ldots, n\} \mapsto \{1, \ldots, n\}$ taková, že $\left(\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right) + d(c_{\pi(n)}, c_{\pi(1)}) \leq B?$

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.

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 $Ax \geq b$?

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.

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?

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:

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:

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:

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:

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:

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:

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:

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:

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:

Množinové pokrytí (Set Cover)
Instance : Systém množin C konečné množiny prvků S, tj. $C \subseteq \mathscr{P}(S)$, přirozené číslo k.
Otázka : Je možné vybrat $C' \subseteq C$, která obsahuje nejvýš k množin a přitom $\bigcup_{C \in C'} C = S$?

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.

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.

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.

Kapitola 8

Pseudopolynomiální a aproximační algoritmy

V této kapitole se budeme věnovat tomu, co pozitivního lze říci o řešitelnosti NP-úplných problémů a úloh.

8.1 Pseudopolynomiální algoritmy a silná NP-úplnost

I mezi různými NP-úplnými problémy a úlohami může být co do složitosti jejich řešitelnosti rozdíl. Podívejme se například na úlohu Batohu definovanou následovně.

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. Tj. chceme, aby platilo: $$\sum_{a \in A'} s(a) \leq B$$ $$\sum_{a \in A'} v(a) = \max_{\substack{A'' \subseteq A \\ \sum_{a \in A''} s(a) \leq B}} \sum_{a \in A''} v(a)$$

Pro tuto úlohu můžeme zkonstruovat algoritmus, který není sice obecně polynomiální, ale pro jistým způsobem omezené instance se polynomiálním může stát. Algoritmus 8.1.1 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[1].

[1] Navíc se zdá na první pohled poněkud nerealistické uvažovat předměty s nulovou velikostí. Snad jedině pokud bychom vzali vážně doporučení: „Přibalte si do batohu dobrou náladu.“

Algoritmus 8.1.1 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: $V := \sum_{i=1}^{n} v[i]$
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** $c := 0$ **to** V
5: **do**
6: $\quad T[0,c] := \emptyset$
7: $\quad S[0,c] := B+1$
8: **done**
9: $S[0,0] := 0$
10: **for** $j := 1$ **to** n
11: **do**
12: $\quad T[j,0] := \emptyset$
13: $\quad S[j,0] := 0$
14: $\quad$ **for** $c := 1$ **to** V
15: $\quad$ **do**
16: $\quad\quad T[j,c] := T[j-1,c]$
17: $\quad\quad S[j,c] := S[j-1,c]$
18: $\quad\quad$ **if** $v[j] \leq c$ **and** $S[j,c] > S[j-1,c-v[j]] + s[j]$
19: $\quad\quad$ **then**
20: $\quad\quad\quad T[j,c] := T[j-1,c-v[j]] \cup \{j\}$
21: $\quad\quad\quad S[j,c] := S[j-1,c-v[j]] + s[j]$
22: $\quad\quad$ **endif**
23: $\quad$ **done**
24: **done**
25: $c := \max\{c' \mid S[n,c'] \leq B\}$
26: **return** $T[n,c]$

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, že velikost vstupu je pouze $O(n\log_2(B+V))$ a V tedy může být exponenciálně větší než vstup.

Poznámka 8.1.2 *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 8.1.1. Dovolíme si tedy zjednodušení, které je ve složitosti obvyklé, a budeme počítat složitost aritmetických operací jako konstantní.*

Věta 8.1.3 *Algoritmus 8.1.1 nalezne pro zadaný vstup množinu předmětů s nejvyšší cenou, jež se vejdou do batohu velikosti B. Algoritmus 8.1.1 pracuje v čase $O(nV)$.*

Důkaz : Fakt, že algoritmus pracuje v čase $O(nV)$ je v podstatě zřejmý. Kroky 1 až 9 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 20 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 $T[j, c]$, $0 \leq j \leq n$, $0 \leq c \leq V$ 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 $S[j, c]$, $0 \leq j \leq n$, $0 \leq c \leq V$ 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 8, 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 12 a 13. 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 18 je ostré, a tedy k přiřazení v kroku 16 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 25 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 8.1.1 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 a NP-úplným problémům, pro něž existuje pseudopolynomiální algoritmus, budeme říkat slabě NP-úplné. Upřesněme si tyto pojmy následující definicí.

Definice 8.1.4 Nechť A je libovolný rozhodovací problém a I nechť je instance tohoto problému. Pomocí $len(I)$ 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 $max(I) > p(len(I))$.

V případě batohu je tedy $len(I) = O(n \log_2(B+V))$, 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 8.1.5 Řekneme, že algoritmus řešící problém A je *pseudopolynomiální*, pokud je jeho časová složitost omezená polynomem dvou proměnných $max(I)$ a $len(I)$.

Algoritmus 8.1.1 je tedy zřejmě pseudopolynomiální. Je-li problém A řešitelný pseudopolynomiálním algoritmem a není-li A číselný problém, pak je zřejmě tento pseudopolynomiální 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.

Definice 8.1.6 Nechť p je polynom a A NP-úplný problém. Pomocí $A(p)$ označíme restrikci problému A na instance I s $max(I) \leq p(len(I))$, tedy instance I je instancí $A(p)$, pokud $max(I) \leq p(len(I))$.

- Řekneme, že problém A je *silně* NP-*úplný*, pokud existuje polynom p, pro nějž je problém $A(p)$ NP-úplný.
- Řekneme, že problém A je *slabě* NP-*úplný*, pokud pro něj existuje pseudopolynomiální algoritmus.

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 je slabě NP-úplný a totéž lze říci například o problému Loupežníci, pro který lze použít jen mírnou modifikaci algoritmu 8.1.1. Je také třeba zmínit, že pokud NP ≠ P, pak existují i NP-úplné problémy, jež nejsou ani silně ani slabě NP-úplné. To je analogické tomu, že pokud NP ≠ P, existují v NP problémy, jež nejsou v Pani nejsou NP-úplné. Jak již bylo zmíněno, rozdíl mezi Pa algoritmy řešitelnými pseudopolynomiálním algoritmem tkví ve způsobu kódování – binárního v případě Pa 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.

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? Tj. existuje permutace $\pi : \{1, \ldots, n\} \mapsto \{1, \ldots, n\}$ taková, že $$\left(\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right) + d(c_{\pi(n)}, c_{\pi(1)}) \leq D?$$

Věta 8.1.7 *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.

8.2 Aproximační algoritmy a schémata

8.2.1 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 8.2.1 *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. Tj. 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 8.2.2 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 $S_A(I) = \emptyset$. 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.

8.2.2 Příklad aproximačního algoritmu pro Bin Packing

Začneme jednoduchým aproximačním algoritmem pro úlohu Bin Packing.

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 $$(\forall i \in \{1, \ldots, m\}) \left[\sum_{u \in U_i} s(u) \leq 1 \right].$$ 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 8.2.3 *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 8.2.4 *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 8.2.4 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 $\{\frac{1}{4} - 2\varepsilon, \frac{1}{4} - 2\varepsilon, \frac{1}{4} - 2\varepsilon, \frac{1}{4} - 2\varepsilon\}$.

8.2.3 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 8.1.1 pracuje dle věty 8.1.3 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 8.1.1 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 8.1.1. 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 8.2.5, kde parametr ε určuje míru zaokrouhlení vstupních cen, což se projeví jednak v časových nárocích algoritmu, jednak v jeho aproximačním poměru.

Algoritmus 8.2.5 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** $\varepsilon \geq n - 1$
3: **then**
4: **return** $\{m\}$
5: **endif**
6: $t := \lfloor \log_2(\frac{\varepsilon \cdot v[m]}{n}) \rfloor - 1$
7: c je nové pole délky n.
8: **for** $i := 1$ **to** n
9: **do**
10: $c[i] := \lfloor \frac{v[i]}{2^t} \rfloor$
11: **done**
12: **return** Batoh(s, c, B) {*Viz algoritmus 8.1.1*}

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 8.2.5 při zadané hodnotě ε.

Věta 8.2.6 *Algoritmus 8.2.5 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*

$$OPT(I) \leq (1+\varepsilon)\textit{Batoh-apx}(I,\varepsilon)$$

Důkaz: Nejprve se zaměříme na aproximační poměr algoritmu 8.2.5. 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

$$x - 1 \leq \lfloor x \rfloor \leq x\ . \tag{8.1}$$

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 (8.1)

$$v[i] - 2^t \leq c[i] \cdot 2^t \leq v[i] \tag{8.2}$$

Využijeme-li pozorování (8.1) pro odhad 2^t zdola, obdržíme

$$\frac{1}{4}\frac{\varepsilon \cdot v[m]}{n} = 2^{\log_2(\frac{\varepsilon \cdot v[m]}{n})-2} \leq 2^t \tag{8.3}$$

a pro odhad shora dostaneme společně s faktem $\varepsilon < n-1$, že

$$2^t \leq 2^{\log_2(\frac{\varepsilon \cdot v[m]}{n})-1} = \frac{1}{2} \cdot \frac{\varepsilon \cdot v[m]}{n} < \frac{1}{2} \cdot v[m]\ . \tag{8.4}$$

Množina M, kterou algoritmus vrátí v kroku 12, 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 8.2. 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 8.1.3 optimální řešení pro instanci $I' = \langle s, c, B\rangle$, zatímco M^* je přípustné řešení této upravené instance, můžeme pokračovat:

$$\begin{aligned}
\textstyle\sum_{i\in M} c[i]\cdot 2^t &\geq \textstyle\sum_{i\in M^*} c[i]\cdot 2^t \geq && (\text{ dle (8.2)})\\
&\geq && \textstyle\sum_{i\in M^*}(v[i]-2^t) =\\
&= && \textstyle\sum_{i\in M^*} v[i] - \sum_{i\in M^*} 2^t \geq\\
&\geq && OPT(I) - n\cdot 2^t
\end{aligned}$$

.

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

$$\begin{aligned} \text{Batoh-apx}(I, \varepsilon) &= \textstyle\sum_{i \in M} c[i] \cdot 2^t \geq c[m] \cdot 2^t \geq \quad \text{(dle 8.2)} \\ &\geq \qquad v[m] - 2^t \geq \qquad \text{(dle 8.4)} \\ &\geq \quad v[m] - \tfrac{1}{2} v[m] = \tfrac{1}{2} v[m] \ . \end{aligned}$$

Dohromady se 8.4 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 8.2.5. Vyjma kroku 12 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 8.1.2 bez újmy na obecnosti předpokládat. Z věty 8.1.3 víme, že krok 12 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 (8.3). Platí tedy, že $C = O(\frac{1}{\varepsilon} n^2)$. Volání algoritmu 8.1.1 tedy zabere čas $O(\frac{1}{\varepsilon} n^3)$, tím je dán i celkový čas algoritmu 8.2.5. ■

8.2.4 Aproximační schémata

To, co jsme získali algoritmem 8.2.5, 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 8.2.7 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 $(1 + \varepsilon)$. Tj. pro maximalizační úlohu 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 8.2.5 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 8.2.5 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 8.2.8 *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*

$$OPT(I) < q(len(I), max(I)).$$

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

$$\varepsilon = q(len(I), max(I))^{-1}$$

a pustíme $ALG_\varepsilon(I)$. Čas tohoto kroku je polynomiální v $len(I)$ a $1/\varepsilon = q(len(I), max(I))$, dohromady se tedy jedná o polynomiální algoritmus v $len(I)$ 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

$$ALG_\varepsilon(I) - OPT(I) \leq \varepsilon OPT(I) < 1.$$

■

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 8.2.8 dostáváme, že pro silně NP-úplné úlohy, které splňují předpoklady této věty, nemá asi smysl pokoušet se o konstrukci úplně polynomiálního aproximačního schématu.

Důsledek 8.2.9 *Nechť A je silně* NP*-úplná optimalizační úloha, která splňuje předpoklady věty 8.2.8. Pokud* P ≠ NP*, pak neexistuje úplně polynomiální aproximační schéma pro úlohu A.*

8.2.5 Neaproximovatelnost

V některých případech nemůžeme (pokud P ≠ 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 ≠ NP.

Věta 8.2.10 *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.

8.3 Cvičení

1. Ukažte, že následující problém je polynomiálně řešitelný:

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 $$\sum_{a \in A'} v(a) = B?$$

2. Definujme optimalizační úlohu Vrcholového pokrytí následovně:

Vrcholové pokrytí
Instance : Neorientovaný graf $G = (V,E)$
Cíl : Nalézt co nejmenší vrcholové pokrytí grafu G. Tj. cílem je nalézt množinu $S \subseteq V$ s co nejmenším počtem vrcholů, pro kterou by platilo, že $(\forall \{u,v\} \in E)\ [\{u,v\} \cap S \neq \emptyset]$.

Uvažme následující jednoduchý aproximační pro úlohu Vrcholového pokrytí:

Algoritmus 8.3.1 Hladový aproximační algoritmus pro Vrcholové pokrytí

Vstup: Neorientovaný graf $G = (V,E)$
Výstup: Vrcholové pokrytí $S \subseteq V$

1: $S := \emptyset$
2: **while** $E \neq \emptyset$
3: **do**
4: Vyber libovolnou hranu $e = \{u,v\} \in E$.
5: $S := S \cup \{u,v\}$
6: Odstraň z E hrany incidentní s u nebo s v.
7: **done**
8: **return** S

Ukažte, že množina S vrácená tímto algoritmem skutečně tvoří vrcholové pokrytí grafu G a že platí $|S| \leq 2OPT(G)$, kde $OPT(G)$ označuje velikost nejmenšího vrcholového pokrytí grafu G.

3. Popište, jak s využitím minimální kostry zkonstruovat 2-aproximační algoritmus pro úlohu obchodního cestujícího, předpokládáme-li, že funkce vzdálenosti splňuje trojúhelníkovou nerovnost.

4. 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.

Kapitola 9

Další zajímavé složitostní třídy

9.1 Doplňky jazyků z NP - třída co-NP

Víme již, že pokud P $\neq$ 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?

Nesplnitelnost KNF (*UNSAT*)
Instance : Formule φ v KNF
Otázka : Platí, že pro každé ohodnocení v je $\varphi(v) = 0$?

U tohoto problému není vidět, zda patří do NP, protože neznáme polynomiálně malý certifikát stvrzující, že formule φ není splněna při žádném ohodnocení jejích proměnných, ve skutečnosti převládá názor, že takový certifikát ani neexistuje. Na druhou stranu si dovedeme představit polynomiálně velký a ověřitelný certifikát negativní odpovědi, protože ohodnocení proměnných v, pro které $\varphi(v) = 1$ ukazuje, že φ je splnitelná. Je vidět, že *SAT* a *UNSAT* jsou v podstatě stejné problémy, záleží jenom 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 9.1.1 Jazyk (problém) $A \subseteq \{0,1\}^*$ patří do třídy co-NP, pokud jeho doplněk $\overline{A} = \{0,1\}^* \setminus A$ patří do třídy NP.

Vzpomeneme-li si na definici rozhodovacího problému 6.1.1, je formálně jazykem SAT jazyk binárních řetězců, které kódují splnitelné formule. Formálně vzato je tedy doplněk jazyka SAT jazykem binárních řetězců, které buď nekódují formule, nebo kódují sice formule, ale nesplnitelné. 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 9.1.2 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 9.1.3 *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. ■

Z toho lze jednoduše odvodit i to, že pokud by se ukázalo, že nějaký NP-úplný problém patří do co-NP, pak by bylo NP = co-NP. Protože třída polynomiálně rozhodnutelných problémů P je zřejmě uzavřená na doplňky, platí také, že $\mathrm{P} \subseteq \mathrm{NP} \cap \text{co-NP}$. K tomu je potřeba dodat, že všeobecně se předpokládá, že NP $\neq$ co-NP a $\mathrm{P} \subsetneq \mathrm{NP} \cap \text{co-NP}$, přesto se stále může ukázat, že platí opak.

Třídu co-NP je také možné si představit jako polynomiálně rozhodnutelný predikát, před nímž použijeme všeobecný kvantifikátor. Přesněji, je-li $A \in \mathrm{NP}$, pak můžeme napsat

$$A = \{x \mid (\exists y) P(x, y)\} ,$$

kde P je predikát rozhodnutelný v polynomiálním čase v délce x, pro nějž platí, že pokud $P(x, y)$, pak y má délku polynomiální v délce x. To plyne z definice třídy NP. Podobně lze jazyk $B \in$ co-NP zapsat jako

$$B = \{x \mid (\forall y) P(x, y)\} ,$$

kde P splňuje výše zmíněné předpoklady na polynomialitu. Opět si všimněme analogie s třídami rekurzivních jazyků (REK), rekurzivně spočetných jazyků (RS) a třídy doplňků rekurzivně spočetných jazyků (co-RS), kde platí podobné vztahy jako v tomto případě, pouze nepožadujeme polynomialitu, ale rekurzivitu. Navíc v případě rekurzivních a rekurzivně spočetných množin víme, že $\mathrm{REK} = \mathrm{RS} \cap \text{co-RS}$ a RS $\neq$ co-RS, situace je tedy přece jen trochu odlišná. (A může to být naopak argument proč si myslet, že $\mathrm{P} = \mathrm{NP} \cap \text{co-NP}$, ačkoli myslet si můžeme cokoli, nezbývá nám však přesto než počkat na důkaz, protože do té doby nevíme nic.)

9.2 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 9.2.1 Funkce f patří do třídy #P, pokud existuje binární relace R v NPF a pro každé x platí, že $f(x) = |\{y \mid (x, y) \in R\}|$. V tom případě nazveme f *početní úlohou asociovanou* s R a budeme ji též označovat pomocí $\#R = f$.

Následující tvrzení je jednoduchým důsledkem definic.

Lemma 9.2.2 *Pro každý problém $A \in$ NP existuje relace R_A v* NPF *pro kterou platí, že $x \in A \Leftrightarrow \#R_A(x) > 0$.*

Důkaz : Relace R_A bude obsahovat dvojice (x, y), kde y je polynomiálně velký certifikát dosvědčující to, že $x \in A$. ■

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 9.2.3 *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 relace $R \in$ NPF, pro kterou platí, že $f = \#R$. Z definice 6.1.6 plyne, že existuje polynom p, pro který platí, že pokud $(x, y) \in R$, pak $|y| \leq p(|x|)$. 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, bude nám na toto vyhledání stačit $p(|x|)$ dotazů, je jich tedy polynomiálně mnoho. ■

Lemma 9.2.4 *Nechť $f \in$ #P, pak lze hodnotu $f(x)$ vypočítat v polynomiálním prostoru vzhledem $|x|$.*

Důkaz : Nechť $f \in$ #P, podle definice to znamená, že existuje relace $R \in$ NPF, pro kterou platí, že $f = \#R$. Z definice 6.1.6 plyne, existuje polynom p, pro který platí, že pokud $(x, y) \in R$, pak $|y| \leq p(|x|)$, což znamená, že $f(x) \leq 2^{p(|x|)}$, a 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 z nich ověříme, zda $(x, y) \in R$ a pokud ano, zvýšíme hodnotu f o jedna. K ověření $(x, y) \in R$ nám podle definice NPF stačí polynomiální čas, a tedy i prostor, pro uložení y i hodnoty $f(x)$ nám také stačí polynomiální prostor, a celkový prostor 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 9.2.5 Řekneme, že funkce $f : \{0,1\}^* \mapsto \mathbb{N}$ je *polynomiálně převoditelná* na funkci $g : \{0,1\}^* \mapsto \mathbb{N}$, pokud existují polynomiálně spočitatelné funkce $\alpha : \{0,1\}^* \times \mathbb{N} \mapsto \mathbb{N}$ a $\beta : \{0,1\}^* \mapsto \{0,1\}^*$, pro které platí, že

$$(\forall x \in \{0,1\}^*) \left[f(x) = \alpha(x, g(\beta(x))) \right] .$$

Intuice za touto převoditelnosti je ta, 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 9.2.6 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 je-li funkce f #P-úplná, znamená to, že jsme schopni spočítat pomocí ní všechny funkce z #P. Máme-li funkce f a g z třídy #P, víme, že každá z nich je asociovaná s nějakou úlohou z třídy NPF. Je-li například $f = \#A$ a $g = \#B$ pro relace $A, B \in$ NPF a víme-li, že $A \leq_m^p B$, přičemž tento převod zachová počet řešení úloh, pak i funkci f lze polynomiálně převést na funkci g.

Definice 9.2.7 Řekneme, že relace $A \in$ NPF je *polynomiálně převoditelná na relaci $B \in$ NPF se zachováním počtu řešení*, pokud existuje polynomiálně spočitatelná funkce $\beta : \{0,1\} \mapsto \{0,1\}^*$ a pro každý řetězec $x \in \{0,1\}^*$ platí, že

$$|\{y \mid (x, y) \in A\}| = |\{y \mid (\beta(x), y) \in B\}| .$$

Z této definice i předchozích úvah plyne následující pozorování.

Lemma 9.2.8 *Nechť $A \in$* NPF *je taková, že libovolnou relaci z* NPF *lze na A polynomiálně převést se zachováním počtu řešení, potom #A 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. 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 disjuntivně normální formě.

Definice 9.2.9 *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 $(x \wedge \overline{y} \wedge \overline{z}) \vee (\overline{x} \wedge y) \vee (y \wedge z) \vee \overline{x}$.

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 9.2.10 *Funkce #DNF-SAT je* #P*-úplná.*

Důkaz : Protože úloha *DNF-SAT* zřejmě patří do NPF (dokonce do do PF), patří *#DNF-SAT* do #P. Popíšeme, jak převést *#SAT* (tj. *#KNF-SAT*, chceme-li explicitně zdůraznit, že jde o splnitelnost formule v KNF) na *#DNF-SAT*. Protože *#KNF-SAT* je #P-úplná funkce, bude to platit i o *#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

$$\#\textit{KNF-SAT}(\varphi) = 2^n - \#\textit{DNF-SAT}(\psi),$$

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 $c = \#\textit{DNF-SAT}(\psi)$ od 2^n. To vše lze jistě provést v polynomiálním čase. ■

Kdybychom neprovedli převod negované funkce na KNF, můžeme stejně 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, ale určit, kolik takových perfektních párování v bipartitním grafu je, je #P-úplná úloha. Tato úloha je ekvivalentní výpočtu permanentu matice. Permanent čtvercové matice A typu $n \times n$ je definován jako

$$per(A) = \sum_{\pi \in S(n)} \prod_{i=1}^{n} A[i, \pi(i)] \ ,$$

kde prvky $S(n)$ jsou permutace množiny $\{1, \ldots, n\}$. Všimněte si, že jde o skoro stejný předpis jako v případě výpočtu determinantu, ale v případě permanentu 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.

9.3 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 (tj. jsou-li v P, NPči co-NPa 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] Alan Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Logic, Methodology and Philosophy of Science, proceedings of the second International Congress, held in Jerusalem, 1964*, Amsterdam, 1965. North-Holland.

[2] 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.

[3] Stephen A Cook and Robert A Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[4] 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.

[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] Robert I. Soare. *Recursively enumerable sets and degrees*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.