Analýza algoritmů. Složitost.

Matematické algoritmy (11MAG)

Jan Přikryl

2. přednáška 11MAG
pondělí 13. října 2013

verze: 2014-11-10 10:37

Obsah

■ přidat následující text jako abstrakt ■

Tento text je ručním přepisem poslední verze přednášky, upravené těsně před Štedrým dnem roku 2010. Na Štěpána mi pak z auta nějaký dobrák ukradl notebook a disk se všemi zálohami, včetně zdrojových souborů pro LaTeX. Snažím se postupně doplňovat, co jsem tehdy již doplnil, omluvte prosím občasné nekonzistence textu a prezentace (oboje se generuje ze stejného zdroje).

1 Úvod

1.1 Analýza algoritmů

Algoritmus:

- myšlenka řešení nějakého problému
- konečný počet kroků řešení
- vyjadřujeme nejčastěji slovně nebo pseudokódem

Program:

- implementace algoritmu ve zvoleném programovacím jazyce

Bez algoritmu nelze napsat program.

Každý problém lze v matematice řešit mnoha různými postupy. Existuje tedy i mnoho různých algoritmů, které vedou ke stejnému cíli. Pokud programátor píše program, který bude použit jenom jednou, zvolí patrně řešení, které lze snadno a rychle implementovat. Jaký algoritmus bychom ale měli preferovat v případě, kdy bude zvolený kus kódu používán opakovaně? V takovém případě je třeba posuzovat různá subjektivní hlediska (srozumitelnost kódu, rozšiřitelnost algoritmu na různé typy vstupních dat, přenositelnost na jiné architektury počítačů a efektivitu výpočtu).

Efektivita algoritmu přímo závisí na tom, kolik zvolený algoritmus spotřebovává výpočetních zdrojů, tedy například jak dlouho bude trvat jeho vykonávání pro určitou množinu vstupních dat, kolik bude potřebovat operační paměti, do jaké míry bude využívat diskový subsystém či síťové rozhraní.

Primárním hlediskem posuzování efektivity algoritmu je ve většině případů *čas*.[1] Uvědomme si ale, že v mnoha případech musí při implementaci docházet k určitým kompromisům – může nastat situace, kdy lze stávající algoritmus zrychlit pouze za cenu neúměrného zvýšení spotřeby operační paměti, spotřeby energie či dalších (ve většině případů přeci jenom omezených) zdrojů. Mnohdy také platí, že implementace velmi efektivních algoritmů jsou mnohem hůře pochopitelné a jejich „údržba“ je proto časově náročnější.

Ideální kombinace: *efektivní algoritmus + efektivní implementace*

Analýza algoritmu:

- poskytne **předpověď výkonnosti** algoritmu
- je **vhodnější než experimenty**
- umožní výběr **vhodné varianty** řešení

Asymptotická složitost **paměťová** × složitost **časová**

[1] Pouze ve velmi ojedinělých případech se setkáváme s nutností omezit paměťový otisk výsledné aplikaci či její energetickou náročnost.

Rychlost vykonávání programu, jenž je implementací zvoleného algoritmu, závisí na zvoleném programovacím jazyce, výpočetním výkonu počítače, jeho momentální zátěži či na zvoleném kompilátoru. Přesto bychom rádi znali předem orientační odhad toho, jak efektivní je algoritmus, jenž jsme zvolili k implementaci.

Existují dva základní způsoby, jak si udělat představu o tom, jak se algoritmus chová:

- experimentální ověření chování implementace
- analýza na základě pseudokódu

Analýza efektivity: Identifikace efektivních a neefektivních částí algoritmu umožní soustředit se na ty části, jejichž úprava přinese nejvyšší nárůst výkonu.

Předpověď výkonnosti programu

Velké projekty potřebují apriorní odhad výkonnosti pro daný hardware – je třeba učinit odhad bez znalosti detailů programového kódu.

Identifikace úzkých míst a jejich vhodné ošetření ještě před vlastním naprogramováním.

Vhodnější než experimenty

Experimenty ověří chování pouze ve vybraných krizových případech – místo „dokázali jsme, že daný algoritmus funguje správně“ lze pouze tvrdit: *„nenalezli jsme způsob, jak prokázat, že algoritmus je špatně“*.

Záruky funkčnosti poskytuje jedině **formální analýza**.

Výběr vhodné varianty

Ne vždy je nejvhodnější varianta ta, jenž je v počtu instrukcí nejefektivnější a tedy nejrychlejší – *např. implementace pro jednočipový počítač × pracovní stanice.*

2 Vývojová stádia algoritmu

Poprvé popsána italským matematikem Leonardem z Pisy, známým také jako Fibonacci (1202).

Růst populace králíků za poněkud idealizovaných podmínek.

Číslo $F[n]$ popisuje velikost populace po n měsících, předpokládáme-li, že

- první měsíc se narodí jediný pár,
- nově narozené páry jsou produktivní od druhého měsíce svého života,
- každý měsíc zplodí každý produktivní pár jeden další pár,
- králíci nikdy neumírají, nemají predátory.

Fibonacciho číslo	Stav
$F[1] = 1$	začínáme s jedním párem
$F[2] = 1$	ještě jsou příliš mladí
$F[3] = 2$	tento měsíc již zplodí první potomky
$F[4] = 3$	druhý pár potomků
$F[5] = 5$	první potomci třetí generace

Obecně

$$F[n+2] = F[n+1] + F[n]$$

Problém 1 (Výpočet Fibonacciho posloupnosti)**.** *Jak efektivně zjistit $F[n]$ pro zvolené n?*

2.1 Algoritmus pomocí explicitního řešení

Explicitní nerekurzivní vztah pro n-tý člen Fibonacciho posloupnosti je

$$F[n] = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}},$$

kde ϕ je hodnota zlatého řezu,

$$\phi = \frac{1+\sqrt{5}}{2} = 1{,}61803398874989\cdots \approx 1{,}618.$$

Algoritmus 1 Přímá varianta výpočtu $F[n]$

Require: n
Ensure: $F[n]$
$F[n] \leftarrow \dfrac{\phi^n - (1-\phi)^n}{\sqrt{5}}$

Reprezentace čísel v plovoucí řádové čárce má svá omezení. V počítači budeme vztah reprezentovat jako

$$F[n] = \frac{1{,}61803^n - 0{,}61803^n}{2{,}23606}.$$

Jaké budou výsledky?

$F[2] = 1{,}00000$ je v pořádku, $F[3] = 1{,}78884$ zaokrouhlíme na 2, $F[20] = 6764{,}69$ ještě zaokrouhlíme na 6765, $F[21] = 10945{,}4$ už zaokrouhlíme na 10945 místo 10946, $F[25] = 75020{,}6$ by mělo být 75025.

Existuje přesnější varianta výpočtu?

2.2 Rekurzivní algoritmus

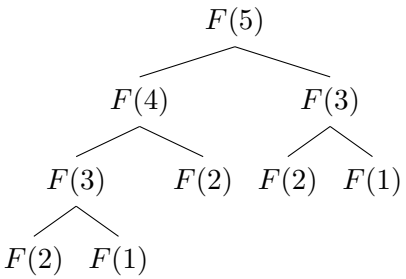
Hodnoty $F[0]$ až $F[2]$ předpočítáme, zbytek lze s jejich pomocí vyjádřit.

Algoritmus 2 Rekurzivní varianta výpočtu $F[n]$

Require: $n \geq 0$
Ensure: $y = F(n)$
1: **if** $n = 0$ **then**
2: $y \Leftarrow 0$
3: **else if** $n \leq 2$ **then**
4: $y \Leftarrow 1$
5: **else**
6: $y \Leftarrow F[n-1] + F[n-2]$
7: **end if**

Jak efektivní je daný algoritmus?

Posloupnost výpočtu $F(5)$:



Počet kroků pro $F(n)$ je $\tau(n) = 3 \cdot F(n) - 2$.

2.3 Dynamické programování

Technika matematické optimalizace.

Dekompozice problému na identické podproblémy.

Dva základní přístupy:

- **shora dolů** – řešíme podproblémy postupně a pamatujeme si řešení
- **zdola nahoru** – vyřešíme všechny potřebné podproblémy a skládáme je

Algoritmus 3

Require: $n \geq 0$
Ensure: $y = F[n]$
1: Alokuj $[f_1, f_2, \ldots, f_n]$
2: $f[2] \leftarrow f[1] \leftarrow 1$
3: **for** $i = 3$ **to** n **do**
4: $f_i \leftarrow f_{i-1} + f_{i-2}$
5: **end for**
6: $y \leftarrow f_n$

Algoritmus 3 potřebuje pole n prvků pro uchování minulých členů posloupnosti.

Jde to ale i bez něj.

Algoritmus 4

Require: $n \geq 0$
Ensure: $y = F[n]$
1: **if** $n = 0$ **then**
2: $y \leftarrow 0$
3: **else**
4: $a \leftarrow 1, b \leftarrow 1$
5: **for** $i = 3$ **to** n **do**
6: $c \leftarrow a + b$
7: $a \leftarrow b, b \leftarrow c$
8: **end for**
9: $y \leftarrow b$
10: **end if**

2.4 Maticová varianta pomocí opakovaného mocnění

Pro členy Fibonacciho posloupnosti platí také

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Použitím opakovaného mocnění snížíme počet kroků na $\mathcal{O}(\log n)$.

Důkaz: Indukcí pro $n \to n + 1$.

Začneme pro $n = 1$, kde platí

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix}$$

a vztah tedy pro $n = 1$ dává korektní výsledek. Za předpokladu, že vztah platí pro n, dostaneme pro $n + 1$

$$\begin{aligned} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n+1} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \\ &= \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \\ &= \begin{bmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{bmatrix} = \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix} \end{aligned}$$

Odvození celého maticového předpisu vychází z teorie dvouprvkových rekurencí a na žádost některých z vás zde uvádím stručný výtah. K dispozici máte volně stažitelnou plnou verzi článku [3], z níž jsem čerpal.

Věta 2. *Množina $\mathcal{R}(a, b)$ Mějme dvojici reálných čísel a a $b \neq 0$ a posloupnost A_n definovanou rekurentním vztahem*

$$A_{n+2} = a \cdot A_{n+1} + b \cdot A_n.$$

Potom pro pevné hodnoty parametrů a a b označíme $\mathcal{R}(a,b)$ množinu všech takto parametrizovaných posloupností.

Jedním z význačných prvků $\mathcal{R}(1,1)$ je Fibonacciho posloupnost F s počátečními členy $F_0 = 0$ a $F_1 = 1$.

Definice 3. Posloupnost A Všechny prvky posloupnosti $A_0, A_1, A_2, \ldots$ tvořící vektor v $\mathbb{R}^\infty$ označíme jako *posloupnost A*.

Věta 4. *Operátor* posunu doleva*, označovaný $\lhd$, odstraní z posloupnosti A její nejlevější prvek.*

Z původní posloupnosti $A = A_0, A_1, A_2, A_3, \ldots$ vznikne posunem doleva o jednu pozici posloupnost $\lhd A = A_1, A_2, A_3, A_4, \ldots$.

Studiem vlastností dvouprvkových rekurencí snadno dospějeme k závěru, že vlastnosti rekurencí $A \in \mathcal{R}(a,b)$ jsou zcela určeny volbou A_0 a A_1. Prostor $\mathcal{R}(a,b)$ je proto dvourozměrný a každé $A \in \mathcal{R}(a,b)$ musíme být schopni vyjádřit jako lineární kombinaci dvou bázových posloupností X a Y s prvky $X_0 = 1, X_1 = 0$ a $Y_0 = 0, Y_1 = 1$,

$$X = 1, 0, b, ab, a^2b + b^2, \ldots,$$
$$Y = 0, 1, a, a^2 + b, a^3 + 2ab, \ldots.$$

Pro všechna $A \in \mathcal{R}(a,b)$ je tedy

$$A = A_0 X + A_1 Y.$$

Všimněte si, že pro $a = 1$ a $b = 1$ je $Y = F$ (půjde o Fibonacciho posloupnost) a také, že

$$\lhd X = b \cdot Y,$$

takže můžeme jakoukoliv posloupnost $A \in \mathcal{R}(a,b)$ psát jako posloupnost členů

$$A_n = A_0 b Y_{n-1} + A_1 Y_n = A_0 b F_{n-1} + A_1 F_n.$$

Operátor posunu doleva lze v $\mathcal{R}(a,b)$ s bázovými prvky X a Y vyjádřit maticí $\mathbf{M}$ jako

$$\lhd \begin{bmatrix} X \\ Y \end{bmatrix} = \mathbf{M}^\mathsf{T} \begin{bmatrix} X \\ Y \end{bmatrix}.$$

Prvky matice $\mathbf{M}$ jsou dány vztahy

$$\lhd X = b \cdot Y,$$
$$\lhd Y = X + a \cdot Y,$$

a proto

$$\mathbf{M} = \begin{bmatrix} 0 & 1 \\ b & a \end{bmatrix}.$$

Posloupnost $A \in \mathcal{R}(a,b)$ je v bázi $[X, Y]$ možno zapsat jako

$$A = \begin{bmatrix} A_0 \\ A_1 \end{bmatrix}$$

přičemž pro n-krát posunutou posloupnost bude platit

$$\overset{n}{\lhd} A = \begin{bmatrix} A_n \\ A_{n+1} \end{bmatrix}.$$

Vzhledem k definici transformační matice $\mathbf{M}$ je ale také $\lhd^n A = \mathbf{M}^n A$, a proto

$$\begin{bmatrix} 0 & 1 \\ b & a \end{bmatrix}^n \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} = \begin{bmatrix} A_n \\ A_{n+1} \end{bmatrix}.$$

Pro n-násobné posuny bázové posloupnosti X a Y můžeme psát

$$\begin{bmatrix} 0 & 1 \\ b & a \end{bmatrix}^n \begin{bmatrix} X_0 \\ X_1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} m_{11} \\ m_{21} \end{bmatrix} = \begin{bmatrix} X_n \\ X_{n+1} \end{bmatrix},$$

respektive

$$\begin{bmatrix} 0 & 1 \\ b & a \end{bmatrix}^n \begin{bmatrix} Y_0 \\ Y_1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{12} \\ m_{22} \end{bmatrix} = \begin{bmatrix} Y_n \\ Y_{n+1} \end{bmatrix},$$

z čehož vyplývá

$$\begin{bmatrix} 0 & 1 \\ b & a \end{bmatrix}^n = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} X_n & Y_n \\ X_{n+1} & Y_{n+1} \end{bmatrix}.$$

Již dříve jsme si všimli, že $Y = F$ (a tedy $Y_n = F_n$) a že $\Lambda X = b \cdot F$ (a tedy $X_n = b \cdot F_{n-1}$). V $\mathcal{R}(a, b)$ bude proto platit

$$\begin{bmatrix} 0 & 1 \\ b & a \end{bmatrix}^n = \begin{bmatrix} b \cdot F_{n-1} & F_n \\ b \cdot F_n & F_{n+1} \end{bmatrix}$$

a v $\mathcal{R}(1, 1)$ ekvivalentně

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n = \begin{bmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{bmatrix}.$$

Algoritmus 5

Require: $n \geq 0$
Ensure: $y = F(n)$
1: **if** $n = 0$ **then**
2: $y \Leftarrow 0$
3: **else**
4: $\mathbf{M} \Leftarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
5: $\mathbf{M} \Leftarrow \mathrm{matpow}(\mathbf{M}, n-1)$
6: $y \Leftarrow M[0,0]$
7: **end if**

Algoritmus 5a

Require: $n \geq 0, \mathbf{A}[2 \times 2]$

Ensure: $\mathbf{B} = \text{matpow}(\mathbf{A}, n)$
1: **if** $n > 1$ **then**
2: $\quad \mathbf{B} \Leftarrow \text{matpow}(\mathbf{A}, n/2)$
3: $\quad \mathbf{B} \Leftarrow \mathbf{BA}$
4: **end if**
5: **if** n je liché **then**
6: $\quad \mathbf{B} \Leftarrow \mathbf{A} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
7: **end if**

Jakým způsobem se bude chování algoritmu měnit v závislosti na velikosti (počtu, objemu) vstupních dat?

Dva základní typy:

- **časová složitost** – vliv na dobu výpočtu
- **paměťová složitost** – nároky na operační paměť

Značíme:

- $\mathcal{O}(N)$ – lineární složitost,
- $\mathcal{O}(N^2)$ – kvadratická složitost,
- $\mathcal{O}(\log N)$ – logaritmická složitost.

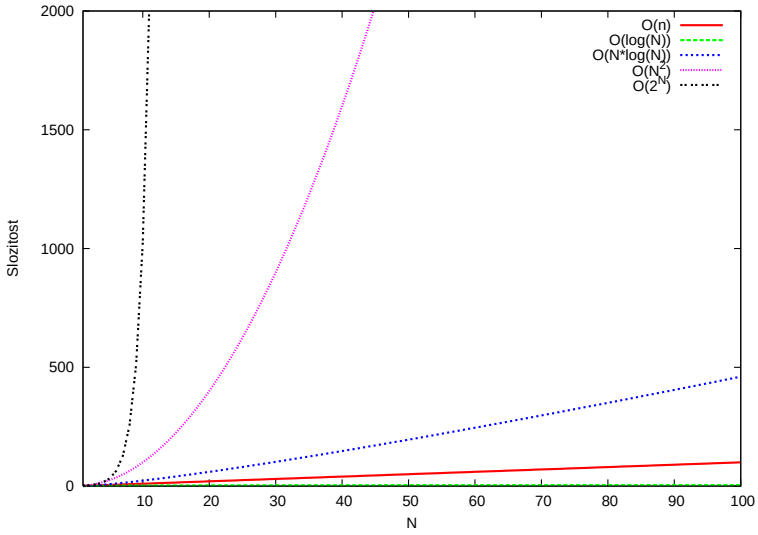
Složitost $\mathcal{O}(N)$ znamená lineárně rostoucí nároky, $\mathcal{O}(N) \sim k \cdot N + q$ pro nějaká $k, n \in \mathbb{N}$, pro $\mathcal{O}(1)$ jsou nároky konstantní, $\mathcal{O}(1) \sim q$.

Vliv asymptotické časové složitosti

Pro $\mathcal{O}(N^2)$ má zdvojnásobení objemu vstupních dat za následek čtyřnásobnou dobu vykonávání algoritmu. Pro $\mathcal{O}(\log N)$ může mít čtyřnásobný počet dat na vstupu za následek dvojnásobnou dobu vykonávání algoritmu. Pro $\mathcal{O}(1)$ je doba vykonávání algoritmu nezávislá na velikosti vstupu.

Vliv asymptotické paměťové složitosti

Pro $\mathcal{O}(N)$ má zdvojnásobení velikosti vstupu za následek dvojnásob vysoké nároky na operační paměť. Pro $\mathcal{O}(2^N)$ čtyřnásobná velikost vstupu zosmínásobí paměťové nároky.



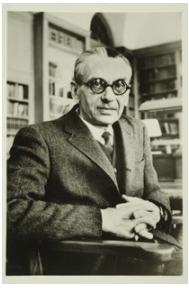
3 Porovnání variant

Algoritmus	Paměťové nároky	Časová složitost
1	$O(1)$	$O(\log N)$
2	$O(N)$	$O(F(N))$
3	$O(N)$	$O(N)$
4	$O(1)$	$O(N)$
5	$O(\log N)$	$O(\log N)$

4 Konečné automaty a Turingův stroj

4.1 Konečný automat

Již v roce 1936 se *Alan Turing*, *Alonso Church* a *Kurt Gödel* zabývali teoretickými základy, na nichž by se dala moderní počítačová věda definovat.



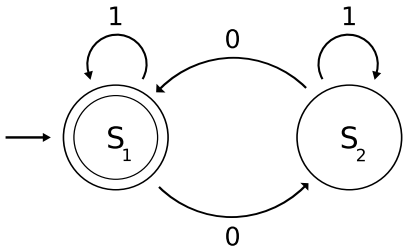
Nezávisle na sobě hledali univerzální model, který by z hlediska funkce bez ohledu na fyzikální vlastnosti popsal chování libovolného výpočtu – *algoritmu*.

Nejčastěji zmiňovaným řešením se stal model Alana Turinga, který byl založen na matematické abstrakci výpočetního stroje jako *konečného automatu*.

Primitivní automat, jenž čte symboly nějaké definované abecedy na vstupu a podle nich přechází

mezi různými vnitřními stavy. Nemá vnitřní paměť.

Může být deterministický nebo nedeterministický.



Nedeterministický konečný automat nemá pevnou tabulky přechodů, ale změna stavu nastane pouze s určitou pravděpodobností.

4.2 Turingův stroj

Motivace: Existuje mechanický proces, kterým je možno rozhodnout o pravdivosti libovolného matematického teorému nebo výroku.

Turingův stroj simuluje práci matematika při vytváření důkazu:

- nekonečná tabule (pro psaní i čtení)
- mozek (řídící jednotka)

Formalizace Turingova stroje

- místo tabule oboustranně nekonečná páska
- místo křídy čtecí a zapisovací hlava, kterou lze posouvat
- místo mozku konečná řídicí jednotka

Takový stroj měl několik základních vlastností:

- musel nahradit složitou symboliku matematických kroků. V takovém případě šlo každou konečnou množinu symbolů nahradit pouze dvěma symboly (jako je 0 a 1) a prázdnou mezerou, která by oba symboly oddělovala.
- podobně jako si matematik zapisuje poznámky na papír, má Turingův stroj k zápisu nekonečnou pásku skládající se z buněk, do/ze kterých se symbol zapisuje/čte.
- nad touto páskou je možné provádět za pomocí čtecí hlavy operace čtení, zápisu a posunu pásky (*read*, *write*, *shift left*, *shift right*).
- protože je možné symboly číst, zapisovat nebo se posunovat po pásce, je pro Turingův stroj důležitý vnitřní stav, ve kterém operaci čtení provádíme (čtený symbol a stav určují další akci a přechod do dalšího stavu)

Protože se chování tohoto stroje vyvíjí podle tabulky přechodů, můžeme říci, že každý následující stav lze jednoznačně určit na základě čteného symbolu a aktuálního stavu.

Chování Turingova stroje je proto *deterministické*.

Klasický Turingův stroj je konečný automat, jenž můžeme popsat sedmicí $T \equiv \{Q, \Gamma, b, \Sigma, \delta, q_0, F\}$, kde

- Q je konečná množina vnitřních stavů,
- Γ je konečná množina symbolů abecedy na pásce,
- $b \in \Gamma$ je symbol pro prázdné políčko,
- $\Sigma \subseteq \Gamma \setminus \{b\}$ je množina vstupních symbolů na pásce,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, P\}$ je přechodová funkce, popisující změnu stavu, zápis na pásku a posun hlavy,
- $q_0 \in Q$ je počáteční stav stroje,
- $F \subseteq Q$ je množina koncových (akceptovaných) stavů.

Cokoliv, co odpovídá tomuto formálnímu zápisu, je Turingův stroj.

5 NP-úplné problémy

Jistě jste už někde slyšeli či četli, že nějaký matematický problém je NP-úplný (angl. *NP-complete*). Podívejme se na závěr této přednášky krátce na to, co tento termín znamená a proč je z hlediska algoritmů tak důležitý. Detailnější pohled na celý problém lze nalézt například v poznámkách prof. Demlové na FEL [2] či v oblíbené učebnici pánů Cormena, Leisersona, Rivesta a Steina [1].

Jako **rozhodovací úlohu** označujeme úlohu, jejímž řešením jsou výroky „ANO“ respektive „NE“. Jako výstup v případě počítače můžeme uvažovat hodnoty true a false nebo 1 a 0.

Běžné úlohy v matematice lze snadno převést na rozhodovací úlohy. Například hledání nejkratší cesty v ohodnoceném grafu lze převést na rozhodovací úlohu „Lze v grafu najít cestu, jejíž délka je menší, než nějaké c?“

Definice 5 (Třída P)**.** Rozhodovací úloha L náleží do třídy P, pokud existuje *deterministický* Turingův stroj, který tuto úlohu rozhodne v polynomiálním čase.

Minimální kostra grafu – existuje kostra s ohodnocením menším, než c?

Nejkratší cesta v grafu – existuje cesta mezi dvěma uzly s ohodnocením menším, než c?

Lineární programování – existuje $\arg\max_{\mathbf{x}} \mathbf{w}^\mathsf{T}\mathbf{x} > c$ za daných omezujících podmínek?

Komprese dat (LZW) – přidá komprese řetězce s do slovníku slovo t?

Definice 6. Rozhodovací úloha L náleží do třídy NP, pokud existuje *nedeterministický* Turingův stroj, který tuto úlohu rozhodne v polynomiálním čase.

Nederministický Turingův stroj: vstupům může odpovídat více, než jedna jediná akce (sekvence instrukcí se převede na neustále se větvící strom instrukcí). Jeho chování si lze představit tak, že v každém přechodu může dojít k naklonování několika kopií stroje se stejnou historií, ale jiným cílovým stavem daného přechodu. Výsledkem provádění důkazu rozhodovací úlohy je potom neustále se rozšiřující strom možností, z nichž v určitém čase jedna povede k cíli.

Platí P⊆NP. Pokud použijeme nedeterministický Turingův stroj v takové konfiguraci, že v každém přechodu nedojde k naklonování žádné kopie, stroj bude vlastně deterministický a jsme v třídě P.

Všechny úlohy třídy P.

Izomorfismus grafu – lze dané dva grafy nakreslit stejně?

Faktorizace čísel – pro dané n a k, existuje $f : 1 < f < k, f \mid n$?

Všechny NP-úplné úlohy.

Třída NP-úplných problémů je třídou rozhodovacích úloh, pro něž platí následující definice:

Definice 7. Rozhodovací úloha L je NP-úplná, pokud náleží do třídy NP a zároveň jde o úlohu NP- těžkou.

Co to znamená:

- jakékoliv řešení L lze ověřit v polynomiálním čase,
- Jakýkoliv problém z třídy NP lze převést na L transformací vstupů opět v polynomiálním čase.

Tyto typy úloh umíme řešit pouze *přibližně*!

Základním úskalím NP-úplných úloh je to, že je sice možné rychle (tedy v polynomiálním čase) ověřit, zda zadané řešení úlohy platí, není ale známý žádný efektivní způsob, jak toto řešení pro dané vstupy najít. Tento paradox je jednou z nejvýznamnějších vlastností této třídy úloh: Pro NP-úplnou úlohu není znám algoritmus, jenž by dokázal dostatečně rychle najít její řešení. Čas, potřebný k vyřešení takové úlohy jakýmkoliv v současné době známým algoritmem roste velmi rychle spolu s tím, jak roste „velikost“ řešeného problému (například počet cifer faktorizovaného čísla). Doba řešení takových úloh v současné době známými algoritmy dosahuje i pro rozumně velké rozsahy vstupních data klidně miliónů let bez ohledu na to, jak roste výpočetní výkon současných počítačů.

Problém batohu – lze zabalit batoh tak, aby jeho hmotnost nepřesáhla m a cena věcí byla alespoň c?Vlastní problém lze možná lépe popsat takto: pokud mám množinu $\mathcal{A}$ obsahující N předmětů o hmotnosti μ a ceně γ, existuje $\mathcal{B} \subseteq \mathcal{A}$ taková, že

$$\sum_{j \in \mathcal{B}} \mu_j \leq m \text{ a zároveň } \sum_{j \in \mathcal{B}} \gamma_j \geq c?$$

Problém obchodního cestujícího – existuje v grafu hamiltonovská kružnice o délce nejvýše c? *Hamiltonovská kružnice* v grafu prochází právě jednou všemi jeho vrcholy, stejně tak jako si přejeme, aby náš obchodní cestující navštívil každé město právě jednou.

Obarvení grafu – lze uzly daného grafu obarvit nejvýše c barvami tak, aby sousedící uzly neměly stejnou barvu?

Problém čínského listonoše (pouze na smíšeném grafu) – existuje v grafu eulerovská kružnice o délce nejvýše c? *Eulerovská kružnice* v grafu prochází právě jednou všechny hrany, stejně tak, jako pošťák musí projet všechny ulice, z nichž některé jsou jednosměrné.

Reference

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, Cambridge, 2 edition, 2001.

[2] Marie Demlová. A4m01tal – teorie algoritmů, 2013.

[3] Dan Kalman and Robert Mena. The fibonacci numbers–exposed. *Mathematics Magazine*, 76(3):xx–xx, 2003.