Státnicové otázky – Matematická lingvistika

19. září 2010

© Studenti MFF

Obsah

1 Státnice – Matematická lingvistika **5**
1.1 Okruhy povinné pro obory I2 a I3 . . . 5
1.2 Matematická lingvistika . . . 5
1.3 PDF verze . . . 5

2 Tvorba algoritmů **6**
2.1 Popis složitosti algoritmů . . . 6
2.2 Rozděl a panuj . . . 6
2.3 Dynamické programování . . . 7
2.4 Hladové algoritmy . . . 8

3 Odhady složitosti **11**
3.1 Dolní odhady složitosti problémů . . . 11
3.2 Amortizovaná složitost . . . 11

4 NP-úplnost **13**
4.1 Třídy P a NP, polynomiální převody, NP-úplnost . . . 13
4.2 Příklady NP-úplných problémů a převody mezi nimi . . . 14
4.3 Silná NP-úplnost, pseudopolynomiální algoritmy . . . 16

5 Aproximační algoritmy **18**
5.1 Aproximační algoritmy . . . 18
5.2 Aproximační schémata . . . 19

6 Vyčíslitelné funkce **20**
6.1 Částečně rekurzivní funkce . . . 20
6.2 Univerzální funkce . . . 23

7 Rekurzivní množiny **26**
7.1 Rekurzivně spočetné množiny . . . 26
7.2 1-převeditelnost, m-převeditelnost . . . 26
7.3 Rekurzivně spočetné predikáty . . . 28
7.4 Generování rekurzivně spočetných množin . . . 30

8 Nerozhodnutelné problémy **32**
8.1 Turingovy stroje . . . 32
8.2 Univerzální Turingův stroj . . . 33

9 Věty o rekurzi **35**
9.1 Věty o rekurzi . . . 35
9.2 Riceova věta . . . 36

10 Stromy **37**
10.1 Binární vyhledávací stromy . . . 37
10.2 B-Stromy a jejich varianty . . . 41
10.3 Trie . . . 43
10.4 Haldy . . . 46

11 Hašování **50**
11.1 Hashování . . . 50
11.2 Univerzální hashování . . . 55
11.3 Perfektní hashování . . . 56

12 Dynamizace **58**
12.1 Úvod . . . 58
12.2 Semidynamizace . . . 58
12.3 Dynamizace . . . 59

13 Vnější paměť **61**
13.1 Základní pojmy . . . 61
13.2 Statické metody organizace souborů . . . 62
13.3 Statické hashovací metody . . . 63
13.4 Dynamické hashování . . . 64
13.5 Třídění na vnější paměti . . . 65

14 Třídění **66**
14.1 Třídění založené na porovnávání . . . 66
14.2 Přihrádkové třídění . . . 68
14.3 Pořadové statistiky (hledání mediánu) . . . 68

15 Závislostní syntax **70**
15.1 Úvod . . . 70
15.2 Závislostní strom . . . 70
15.3 Vztahy v závislostní syntaxi . . . 71
15.4 Projektivita . . . 72
15.5 Valence . . . 73

16 Frázové gramatiky **74**
16.1 Frázové gramatiky . . . 74
16.2 Začátky Transformační gramatiky . . . 75
16.3 Teorie Principles and Parameters / Government and Binding . . . 77
16.4 90. léta – Program Minimalismu . . . 79
16.5 Další teorie s frázovou gramatikou . . . 79
16.6 Poznámky . . . 81

17 Obecná lingvistika **82**
17.1 Typologie jazyků . . . 82
17.2 Strukturní lingvistika . . . 86
17.3 Poznámky . . . 88

18 FGD **89**
18.1 Úvod . . . 89
18.2 Roviny popisu . . . 89
18.3 Jazykový význam . . . 90
18.4 Valence . . . 91
18.5 Aktuální členění ve FGD . . . 93
18.6 Pražský závislostní korpus . . . 95

19 Formální sémantika **97**
19.1 Úvod . . . 97
19.2 Extenzionální model významu . . . 98
19.3 Intenzionální model významu . . . 100
19.4 “Hyperintenzionální” modely významu . . . 101
19.5 Dynamické modely významu . . . 102
19.6 Vybrané speciálnější problémy sémantiky . . . 102
19.7 Odkazy . . . 103

20 Korpusy **104**
20.1 Zdroje dat . . . 104
20.2 Anotace . . . 106
20.3 Datové formáty . . . 106
20.4 Typologie korpusů . . . 108
20.5 Počítačová lexikografie . . . 111
20.6 Wordnety . . . 112
20.7 Poznámky . . . 113

21 Strojové učení **114**
21.1 Úvod . . . 114
21.2 Učení založené na konceptu . . . 114
21.3 Rozhodovací stromy . . . 116
21.4 Neuronové sítě . . . 117
21.5 Učení založené na příkladech . . . 120
21.6 Vyhodnocování hypotéz . . . 121
21.7 Výpočetní aspekty strojového učení . . . 122

22 Stochastické metody **123**
22.1 Teorie informace . . . 123
22.2 Bayesovské učení . . . 125
22.3 Hidden Markov Models . . . 127
22.4 Algoritmy učení a zpracování, aplikace v lingvistice . . . 130

23 Experimenty **132**
23.1 Úvod . . . 132
23.2 Příprava dat . . . 132
23.3 Standardní evaluační metriky . . . 132
23.4 Typy evaluace podle úloh . . . 133

24 Analýza jazyka **134**
24.1 Morphology & Tagging . . . 134
24.2 Parsing . . . 138

25 Generování jazyka **142**
25.1 Úvod . . . 142
25.2 Struktura NLG systému . . . 142
25.3 Příklady NLG systémů . . . 143
25.4 Evaluace . . . 143

26 Analýza a syntéza řeči **144**
26.1 Speech Recognition . . . 144
26.2 Speech Synthesis . . . 151

27 Extrakce informací **156**
27.1 Informační systémy . . . 156
27.2 Vyhledávání v textu . . . 156
27.3 Boolské informační systémy . . . 156
27.4 Vektorové informační systémy . . . 156
27.5 Induktivní systémy . . . 157
27.6 Signaturové systémy . . . 157
27.7 Rozšířená boolská logika . . . 157
27.8 Rozlišovací hodnoty termů v indexu . . . 157
27.9 Přibližné hledání . . . 157

28 Strojový překlad **158**
28.1 Proč je strojový překlad těžký? . . . 158
28.2 Úkoly strojového překladu . . . 158
28.3 Překladový trojúhelník (Vauquois triangle) . . . 158
28.4 Metody Evaluace . . . 159
28.5 Rule-based strojový překlad . . . 159
28.6 Statistický strojový překlad — phrase-based . . . 160
28.7 Alignment . . . 163
28.8 Evaluace . . . 163

28.9 Historie 165
28.10Systémy podporující překlad 166
28.11České systémy 166

Kapitola 1

Státnice - Informatika - I3: Matematická lingvistika

Podle oficiálních stránek MFF sestávají státnicové otázky pro obor Matematická lingvistika z následujících okruhů:

1.1 Okruhy povinné pro obory I2 a I3

1.2 Matematická lingvistika

Základy formálního popisu přirozených jazyků

- Závislostní syntax (formální definice a vlastnosti závislostních stromů – závislosti, koordinace, projektivita)
- Syntax bezprostředních složek a frázové gramatiky (základní principy, vývoj Chomského školy)
- Základy obecné lingvistiky (zdroje a přínosy strukturní lingvistiky, typologie jazyků, pojem funkce)
- Funkční generativní popis (základní charakteristika, struktura rovin, valenční teorie, zachycení významu, aktuální členění)
- Formální sémantika

Jazykové korpusy, strojové učení a stochastické metody

- Jazykové korpusy a lingvistická anotace (zdroje dat, anotace, datové formáty, typologie korpusů, počítačová lexikografie, wordnety)
- Metody strojového učení (učení založené na konceptu, rozhodovací stromy, neuronové sítě, učení založené na příkladech, vyhodnocování hypotéz, výpočetní aspekty strojového učení)
- Stochastické metody a jejich aplikace v počítačové lingvistice (Teorie informace, Bayesovské učení, HMM, algoritmy učení a zpracování, aplikace v lingvistice)
- Návrh a vyhodnocování lingvistických experimentů (příprava dat, standardní evaluační metriky, typy evaluace podle úloh)

Automatické zpracování přirozeného jazyka

- Automatická analýza jazyka (morfologie, syntax povrchová a hloubková, aplikace)
- Generování přirozeného jazyka
- Analýza a syntéza mluvené řeči (jazykové modely, kombinace modelů)
- Vyhledávání a extrakce informací
- Strojový překlad (transfer, interlingua, metody překladu, systémy pro češtinu, počítačem podporovaný překlad)

1.3 PDF verze

K dispozici je PDF verze, která ale nemusí být aktuální. – Tuetschek 15:56, 1 Sep 2010 (CEST)

Kapitola 2

Státnice - Metody tvorby algoritmů

2.1 Popis složitosti algoritmů

Definice (Velikost dat, krok algoritmu)

Velikost dat je obvykle počet bitů, potřebných k jejich zapsání, např. data D jako pro čísla $a_1, \ldots, a_n$ je velikost dat $|D| = \sum_{i=1}^{n} \lceil \log a_i \rceil$.

Krok algoritmu je jedna operace daného abstraktního stroje (např. Turingův stroj, stroj RAM), zjednodušeně jde o nějakou operaci, proveditelnou v konstantním čase (např. aritmetické operace, porovnání hodnot, přiřazení – pro jednoduché číselné typy).

Definice (Časová složitost)

Časová složitost je funkce $f : \mathbb{N} \to \mathbb{N}$ taková, že $f(|D|)$ udává počet kroků daného algoritmu, pokud je spuštěn na datech D.

Definice (Asymptotická složitost)

Řekneme, že funkce $f(n)$ je *asymptoticky menší nebo rovna* než $g(n)$, značíme $f(n)$ je $O(g(n))$, právě tehdy, když

$$\exists c > 0 \exists n_0 \forall n > n_0 : 0 \leq f(n) \leq c \cdot g(n)$$

Funkce $f(n)$ je *asymptoticky větší nebo rovna* než $g(n)$, značíme $f(n)$ je $\Omega(g(n))$, právě tehdy, když

$$\exists c > 0 \exists n_0 \forall n > n_0 : 0 \leq c \cdot g(n) \leq f(n)$$

Funkce $f(n)$ je *asymptoticky stejná* jako $g(n)$, značíme $f(n)$ je $\Theta(g(n))$, právě tehdy, když

$$\exists c_1, c_2 > 0 \exists n_0 \forall n > n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Funkce $f(n)$ je *asymptoticky ostře menší* než $g(n)$ ($f(n)$ je $o(g(n))$, když

$$\forall c > 0 \exists n_0 \forall n > n_0 : 0 \leq f(n) < c \cdot g(n)$$

Funkce $f(n)$ je *asymptoticky ostře větší* než $g(n)$ ($f(n)$ je $w(g(n))$, když

$$\forall c > 0 \exists n_0 \forall n > n_0 : 0 \leq c \cdot g(n) < f(n)$$

Poznámka

Asymptotická složitost zkoumá chování algoritmů na velkých datech, zařazuje je podle toho do kategorií. Zanedbává multiplikativní a aditivní konstanty.

2.2 Rozděl a panuj

Definice (Metoda rozděl a panuj)

Rozděl a panuj je metoda návrhu algoritmů (ne strukturované programování), která má 3 kroky:

1. *rozděl* – rozdělí úlohu na několik podúloh stejného typu, ale menší velikosti
2. *vyřeš* – vyřeší podúlohy a to buď přímo pro dostatečně malé, nebo rekurzivně pro větší
3. *sjednoť* – sjednotí řešení podúloh do řešení původní úlohy

Aplikace

- QUICKSORT
- Hledání mediánu

Analýza složitosti algoritmů rozděl a panuj

Poznámka (Vytvoření rekurentní rovnice)

Pro časovou složitost algoritmů typu rozděl a panuj zpravidla dostávám nějakou rekurentní rovnici.

- $T(n)$ budiž doba zpracování úlohy velikosti n, za předpokladu, že $T(n) = \Theta(1)$ pro $n \leq n_0$.
- $D(n)$ budiž doba na rozdělení úlohy velikosti n na a podúloh stejné velikosti $\frac{n}{c}$.
- $S(n)$ budiž doba na sjednocení řešení podúloh velikosti $\frac{n}{c}$ na jednu úlohu velikosti n. Dostávám rovnici

$$T(n) = \begin{cases} D(n) + aT(\frac{n}{c}) + S(n) & n > n_0 \\ \Theta(1) & n \leq n_0 \end{cases}$$

Poznámka

Při řešení rekurentních rovnic:

- Zanedbávám celočíselnost ($\frac{n}{2}$ místo $\lceil \frac{n}{2} \rceil$ a $\lfloor \frac{n}{2} \rfloor$)
- Nehledím na konkrétní hodnoty aditivních a multiplikativních konstant, asymptotické notace používám i v zadání rekurentních rovnic, i v jejich řešení.

Věta (Substituční metoda)

1. Uhodnu asymptoticky správné řešení
2. Indukcí ověřím správnost (zvláště horní a dolní odhad)

Věta (Metoda "kuchařka" (Master Theorem))

Nechť $a \geq 1, c > 1, d \geq 0 \in \mathbb{R}$ a nechť $T : \mathbb{N} \to \mathbb{N}$ je neklesající funkce taková, že $\forall n$ tvaru c^k platí

$$T(n) = aT(\frac{n}{c}) + \Theta(n^d)$$

Potom

1. Je-li $\log_c a \neq d$, pak $T(n)$ je $\Theta(n^{\max\{\log_c a, d\}})$
2. Je-li $\log_c a = d$, pak $T(n)$ je $\Theta(n^d \log_c n)$

Věta (Master Theorem, varianta 2)

Nechť $0 < a_i < 1$, kde $i \in \{1, \ldots, k\}$ a $d \geq 0$ jsou reálná čísla a nechť $T : \mathbb{N} \to \mathbb{N}$ splňuje rekurenci

$$T(n) = \sum_{i=1}^{k} T(a_i \cdot n) + \Theta(n^d)$$

Nechť je číslo x řešením rovnice $\sum_{i=1}^{k} a_i^x = 1$. Potom

1. Je-li $x \neq d$ (tedy $\sum_{i=1}^{k} a_i^d \neq 1$), pak $T(n)$ je $\Theta(n^{\max\{x,d\}})$
2. Je-li $x = d$ (tedy $\sum_{i=1}^{k} a_i^d = 1$), pak $T(n)$ je $\Theta(n^d \log n)$

2.3 Dynamické programování

Dynamické programování je metoda řešení problemů, které v sobě obsahují překrývající se subproblémy.

Příklad

Typickým příkladem je výpočet fibonaciho čísla. Fib. posloupnost je definována jako:

$$f(0) = 1, f(1) = 1$$

$$f(n+2) = f(n+1) + f(n)$$

Výpočet třeba 4. čísla by pak byl $f(4) = f(3)+f(2) = f(2)+f(1)+f(1)+f(0) = f(1)+f(0)+f(1)+f(1)+f(0) = 5$. Vidíme, že jsme $f(2)$ počítali dvakrát, $f(1)$ třikrát a $f(0)$ dvakrát. Ve větším měřítku toto zbytečně počítání vede k exponenciální složitosti algoritmu. Lepší cestou je počítat "od spodu", kdy pomoci $f(0)$ a $f(1)$ spočteme $f(2)$, pak s jeho pomoci $f(3)$ nakonec $f(4)$ s lineární složitosti.

V dynamickém programování se například vytvoří mapa Fibonacciho čísel a jejich hodnot. Před tím, než začnu počítat hodnotu nějakého Fibonacciho čísla, se podívám do mapy.

Nutné podmínky

V dynamickém programování využíváme:

1. *Překrývání podproblému* — problém lze rozdělit na podproblémy, jejichž řešení se využívá opakovaně.
2. *Optimální podstruktury* — optimální řešení lze zkonstruovat z optimálních řešení podproblémů.

Postupy

Obvykle se používá jeden ze dvou přístupů k dynamickému programování:

- *Top-down* — problém se rekurzivně dělí na podproblémy a po jejich vyřešení se zapamatují výsledky, které se použijí při případném opětovném řešení daného podproblému.
- *Bottom-up* — nejdříve se spočítají všechny možné podproblémy (viz příklad s Fibonacciho posloupnosti), které se potom skládají do řešení větších problémů. Tento přístup je výhodnější z hlediska počtu volání funkci a místa na zásobníků, ale ne vždy musí být zřejmě, které všechny subproblémy je třeba předem spočítat.

Použití

Problém batohu – máme věci různé váhy a batoh určité nosnosti. Které věci máme dát do batohu, abychom jej co nejlépe zaplnili (jednorozměrná varianta problému batohu)? Speciální případ je součet podmnožiny (SP). Algoritmus je popsán v sekci o NP-úplnosti.

Uzávorkování součinu matic tak, aby počet skalárních součinů byl co nejmenší. Dělá se pomocí 2 čtvercových matic (M a K) řádu rovného počtu násobených matic, kde pracujeme jen nad diagonálou. Hodnota na M_{ij} udává minimální počet skalárních součinů při nejlepším uzávorkování matic i až j, hodnota K_{ij} určuje matici, která rozděluje závorkování na dvě podmnožiny matic. Hodnotu M_{ij} lze zkonstruovat ze všech M_{ik} a M_{kj} pro $i < k < j$.

2.4 Hladové algoritmy

Motivace

Problém 1

Dán souvislý neorientovaný graf $G = (V, E)$ a funkce $d : E \to \mathbb{R}^+$, udávající délky hran. Najděte minimální kostru grafu G, tj. kostru $G' = (V, E')$ tak, že $\sum_{e \in E'} d(e)$ je minimální.

Problém 2

Je dána množina $S = \{1, \ldots, n\}$ úkolů jednotkové délky. Ke každému úkolu je dána lhůta dokončení $d_i \in \mathbb{N}$ a pokud $w_i \in \mathbb{N}$, kterou je úkol i penalizován, není-li hotov do své lhůty. Najděte rozvrh (permutaci úkolů od času 0 do času n), který minimalizuje celkovou pokutu.

Problém 3

Je dána množina $S = \{1, \ldots, n\}$ úkolů. Ke každému úkolu je dán čas jeho zahájení s_i ukončení f_i, $s_i \leq f_i$. Úkoly i a j jsou kompatibilní, pokud se intervaly $[s_i, f_i)$ a $[s_j, f_j)$ nepřekrývají. Najděte co největší množinu (po dvou) kompatibilních úkolů.

Matroid

Definice (Matroid)

Matroid je uspořádaná dvojice $M = (S, I)$, splňující:

- S je konečná neprázdná množina (*prvky matroidu* M)
- I je neprázdná množina podmnožin S (*nezávislé podmnožiny*), která má

1. *dědičnou vlastnost*: $B \in I \wedge A \subseteq B \quad \Rightarrow \quad A \in I$,
2. *výměnnou vlastnost*: $A, B \in I \wedge |A| < |B| \quad \Rightarrow \quad \exists x \in B \backslash A : A \cup \{x\} \in I$.

Věta (O velikosti maximálních nezávislých podmnožin)

Všechny *maximální* (maximální vzhledem k inkluzi) nezávislé podmnožiny v matroidu mají stejnou velikost.

Důkaz

Z výměnné vlastnosti, nechť $A, B \in I$ maximální, $|A| < |B|$, pak $A \cup \{x\} \in I, x \notin A$, což je spor.

Definice (Vážený matroid)

Matroid $M = (S, I)$ je *vážený*, pokud je dána funkce $w : S \to \mathbb{R}^+$ a její rozšíření na podmnožiny množiny S je definováno předpisem:

$$A \in S \Rightarrow w(A) = \sum_{x \in A} w(x)$$

Problém 1 a 2 – zobecněný

Pro daný vážený matroid nalezněte nezávislou podmnožinu s co největší vahou (*optimální množinu*). Protože váhy jsou kladné, vždy se bude jednat o maximální nez. množinu.

Problém 1 je spec. případ tohoto, protože můžeme uvažovat *grafový matroid* (nad hranami) – $M_G = (S, I)$, kde $S = E$ a pro $A \subseteq E$ platí $A \in I$, pokud hrany z A netvoří cyklus (tj. indukovaný podgraf tvoří les).

Dědičná vlastnost této struktury je zřejmá, výměnnost se dá dokázat následovně: mějme $A, B \subseteq E, |A| < |B|$. Pak lesy z A, B mají $n - a > n - b$ stromů (vč. izolovaných vrcholů). V B musí být strom, který se dotýká ≥ 2 různých stromů z A. Ten obsahuje hranu, která není v žádném stromě A a netvoří cyklus a tak ji můžu k A přidat.

Váhovou funkci převedu na hledání maxim: $w(e) = c - d(e)$, kde c je dost velká konstanta, aby všechny váhy byly kladné. Algoritmus pak najde max. množinu, kde hrany netvoří cyklus. Implicitně bude mít $n - 1$ hran, takže půjde o kostru a její w bude maximální, tedy původní váha minimální.

Pro problém 2 zavedeme pojem *kanonického rozvrhu* – takového rozvrhu, kde jsou všechny *včasné* úkoly rozvrženy před všemi *zpožděnými* a uspořádány podle neklesající lhůty dokončení (tohle na celkové pokutě nic nezmění a máme bijekci mezi množinou včasných úkolů a kanonickými rozvrhy).

Optimální množinu pak lze hledat jen nad kanonickými rozvrhy – nezávislou množinou úkolů nazvu takovou, pro kterou existuje kanonický rozvrh tak, že žádný úkol v ní obsažený není zpožděný. Potom hledání max. nezávislé množiny při ohodnocení pokutami je hledání nejmenší pokuty (odeberu co nejvíc z možné celkové pokuty).

Pak zbývá dokázat, že takto vytvořená struktura je matroid. Dědičná vlastnost je triviální – vyhodím-li něco z nezávislé množiny, nechám v rozvrhu mezery a bude platit stále. Pro výměnnou vlastnost zavedu pomocnou funkci $N_t(C) = |\{i \in C | d_i \leq t\}|$, udávající počet úkolů z množiny C se lhůtou do času t. Pak množina C je nezávislá, právě když $\forall t \in \{1, \ldots, n\} : N_t(C) \leq t$.

Pak máme-li 2 nezávislé A, B, $|B| > |A|$, označíme k největší okamžik takový, že $N_k(B) \leq N_k(A)$, tj. od $k+1$ dál platí $N_t(A) < N_t(B)$. To skutečně nastane, protože $|B| = N_n(B) > N_n(A) = |A|$. Pak určitě v B je ostře víc úkolů s $d_i = k + 1$ než v A, tj. $\exists x \in B \backslash A$ se lhůtou $k + 1$. $A \cup \{x\}$ je nezávislá, protože $N_t(A \cup \{x\}) = N_t(A)$ pro $t \leq k$ a $N_t(A \cup \{x\}) \leq N_t(B)$ pro $t \geq k + 1$.

Hladový algoritmus na váženém matroidu

Algoritmus (Greedy)

Mám zadaný matroid $M = (S, I)$, kde $S = \{x_1, \ldots, x_n\}$ a $w : S \to \mathbb{R}^+$. Potom

1. $A := \emptyset$, setřiď a přeznač S sestupně podle vah
2. pro každé x_i zkoušej: je-li $A \cup \{x_i\} \in I$, tak $A := A \cup \{x_i\}$
3. vrať A jako maximální nezávislou množinu

Pokud přidám nějaké x_i, nikdy nezruším nezávislost množiny; s už přidanými prvky se nic nestane. Časová složitost je $\Theta(n \log n)$ na setřídění podle vah, testování nezávislosti množiny závisí na typu matriodu ($f(n)$), takže celkem něco jako $\Theta(n \log n + n \cdot f(n))$.

Důkaz

- Vyhození prvků, které samy o sobě nejsou nezávislé množiny, nic nezkazí (z dědičné vlastnosti).
- První vybrané x_i je "legální" (nezablokuje mi cestu), protože mezi max. nez. množinami určitě existuje taková, která jím mohla vzniknout (z výměnné vlastnosti – vezmeme první prvek, který je sám o sobě nez. mn. a s pomocí nějaké max. nez. množiny B ho doplníme, vzniklé $\{x_i\} \cup (B \backslash \{x_j\})$ musí být také maximální).
- Nalezení optimální množiny obsahující pevné (první vybrané) x_i v $M = (S, I)$ je ekvivalentní nalezení optimální množiny v $M' = (S', I')$, kde $S' = \{y \in S | \{x_i, y\} \in I\}$ a $I' = \{B \subset S \backslash \{x_i\} | B \cup \{x_i\} \in I\}$ (je-li S' neprázdná, je to matroid, vlastnosti se přenesou z původního; pokud je S' prázdná, tak algoritmus končí) a tedy když vyberu x_i, nezatarasím si cestu k optimu – stačí ukázat, že $A \cup \{x\}$ je optimální v M právě když A je optimální v M'.
- Takže když budu vybírat (indukcí opakovaně) x_i podle váhy, dojdu k optimální množině.

Algoritmus (Hladový algoritmus na problém 3)

Máme $S = \{1, \ldots, n\}$ množinu úkolů s časy startů s_i a konců f_i. Provedeme:

1. $A := \emptyset$, $f_0 := 0, j := 0$
2. setřiď úkoly podle f_i vzestupně a přeznač
3. pro každé i zkoušej: je-li $s_i \geq f_j$, pak $A := A \cup \{i\}$ a $j := i$
4. vrať A jako max. množinu nekryjících se úkolů

Složitost je $n \log n$ na setřídění, zbytek už lineární. Sice to funguje, ale tahle struktura NENÍ matroid – nesplňuje výměnnou vlastnost. Algoritmus má ale stejný důkaz jako předchozí na matroidech (legálnost hladového výběru – existuje max. množina obsahující prvek 1 – a existenci optimální množiny – převod na "menší" zadání).

Kapitola 3

Státnice - Odhady složitosti

3.1 Dolní odhady složitosti problémů

Definice (Složitost problému)

Složitost problému je složitost asymptoticky nejlepšího *možného* algoritmu, který řeší daný problém (ne nejlepšího známého).

Každý konkrétní algoritmus dává horní odhad složitosti. Dolní odhady (až na triviální – velikost vstupu, výstupu) jsou složitější.

Věta (Dolní odhad složitosti mediánu)

Pro výběr k-tého z n prvků je třeba alespoň $n-1$ porovnání, tj. problém je $\Omega(n)$.

Důkaz

Intuitivně potřebuju medián, i kdyby mi spadnul z nebe, porovnat se všemi ostatními prvky, abych vůbec zjistil, jestli to je medián ...

Věta (Dolní odhad složitosti třídění)

Pro každý třídící algoritmus, založený na porovnávání prvků, existuje vstupní posloupnost, pro kterou provede $\Omega(n \log n)$ porovnání.

Důkaz

Nakreslím si rozhodovací strom jako model algoritmu – všechny vnitřní uzly odpovídají nějakému porovnání, které algoritmus provedl, jejich synové jsou operace, které nasledovaly po různých výsledcích toho porovnání (BÚNO jsou-li prvky různé, bude strom binární). Listy odpovídají setříděným posloupnostem. Aby byl algoritmus korektní, musí mít strom listy se všemi $n!$ možnými pořadími prvků.

Pro plýtvající algoritmus mohou existovat listy, neodpovídající žádné permutaci, tj. porovnává stejnou dvojici prvků dvakrát (a jedna z možností už nemůže nastat). Pro rovnoměrné rozdělení je očekávaný čas průměrná délka cesty od kořene k listům, nejhorší čas je výška stromu.

Označím výšku jako h, pak počet listů je $\leq 2^h$ a tedy $n! \leq 2^h$, tj. $h \geq \log n!$, pro dolní odhad $\Omega(n \log n)$ stačí odhad faktoriálu $n! < n^{\frac{n}{2}}$, případně můžu použít Stirlingův vzorec $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$ a dostávám $\Theta(n \log n)$.

3.2 Amortizovaná složitost

Definice (Amortizovaná složitost)

Typicky se používá pro počítání časové složitosti operací nad datovými strukturami, počítá *průměrný čas na 1 operaci při provedení posloupnosti operací*. Dává realističtější odhad složitosti posloupnosti všech operací, než měření všech nejhorším případem.

Známe 3 metody amortizované analýzy:

- agregační
- účetní
- potenciálová

Problémy:

- *Inkrementace binárního čítače* – do binárního čítače délky k postupně přičteme n-krát jedničku. Počet bitových operací na 1 přičtení je v nejhorším případě $O(\log n)$, ale amortizovaně dojdeme k $O(1)$.
- *Vkládání do dynamického pole* – začnu s prázdným polem délky 1 a postupně vkládám n prvků. Pokud je stávající pole plné, alokujeme dvojnásobné a kopírujeme prvky. Počet kopírování prvků na jedno vložení je až $O(n)$, ale amortizovaně opět $O(1)$.

Algoritmus (Agregační metoda)

Spočítáme nejhorší čas pro celou posloupnost n operací – $T(n)$, amortizovaný čas na 1 operaci je pak $\frac{T(n)}{n}$.

- **Binární sčítání**: v průběhu n přičtení se i-tý bit překlopí $\lfloor \frac{n}{2^i} \rfloor$-krát, takže celková cena překlopení je $\leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$, tj. amortizovaně na jedno přičtení $\frac{2n}{2} = \Theta(1)$
- **Vkládání**: cena i-tého vložení do pole je $c_i = \begin{cases} i \text{ pokud } \exists k : i - 1 = 2^k \\ 1 \text{ jinak} \end{cases}$. Celkem dostávám $T(n) = \sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \leq n + 2n = 3n$, takže na jedno vložení vyjde $\frac{3n}{n} = \Theta(1)$.

Algoritmus (Účetní metoda)

Od každé operace vyberu urč. pevný "obnos", kterým onu operaci "zaplatím". Pokud něco zbyde, dám to na účet, pokud bude oprace naopak dražší než onen obnos, z účtu vybírám. Zůstatek na účtu musí být stále nezáporný – pokud uspěji, obnos je amortizovaná cena 1 operace.

- **Binární sčítání**: Při každém přičtení je *právě jeden* bit překlopen *z 0 na 1*. Proto každému bitu zavedeme účet a za přičtení budeme vybírat 2 jednotky. Jedna je použita na překlopení daného bitu z 0 na 1 a druhá uložena na právě jeho účet; překlápění z 1 na 0 jsou hrazeny z účtů (protože každý bit, který má nastavenu 1 má na účtu právě 1 jednotku, projde to). Amortizovaná cena tedy vyjde $2 = \Theta(1)$.
- **Vkládání**: Od každého vložení vyberu 3 jednotky – na vlastní vložení, na překopírování právě vloženého prvku při příštím vložení a na příští překopírování odpovídajícího prvku v levé polovině pole (na pozici $n - \frac{s}{2}$), který obnos ze svého vložení vyčerpal. Po expanzi je celkem na všech účtech 0, jindy víc, tj. amortizovaná cena operace je $3 = \Theta(1)$.

Algoritmus (Potenciálová metoda)

Je to podobné bankovní, roli účtu hraje nějaká funkce w, která popisuje vhodnost jednotlivých konfigurací $D_0, D_1, \ldots$. Potřebuji potom, aby $w(D_i) \geq 0 \ \forall i$. Amortizovaná složitost i -té operace o je potom:

$$am(o_i) = T(o_i) + w(D_{i+1}) - w(D_i)$$

Složitost nejhoršího případu celé posloupnosti operací může být mnohem "rychlejší" než posloupnost nejhorších případů jednotlivých operací:

$$\sum_{i=1}^{n} T(o_i) \leq \sum_{i=1}^{n} am(o_i) + w(D_0)$$

Kapitola 4

Státnice - NP-úplnost

4.1 Třídy P a NP, polynomiální převody, NP-úplnost

Definice (Úloha)

- *Úloha* je situace, kdy pro daný vstup (*instanci úlohy*) chceme získat výstup se zadanými vlastnostmi.
- *Optimalizační úloha* je úloha, kde cílem je získat optimální (zpravidla největší nebo nejmenší) výstup s danými vlastnostmi.
- *Rozhodovací problém* je úloha, jejímž výstupem je ANO/NE.

Definice (Kódování vstupů)

Každá instance problému Q je kódována jako posloupnost 0 a 1, tj. instance je slovo v abecedě $\{0,1\}^*$. Kódy všech instancí problému Q tvoří jazyk $L(Q)$ nad abecedou $\{0,1\}^*$, který se dělí na

- $L(Q)_Y$ – kódy instancí s odpovědí ANO (jazyk kladných instancí)
- $L(Q)_N$ – kódy instancí s odpovědí NE (jazyk záporných instancí)

Rozhodovací problém pak je rozhodnutí, zda $x \in L(Q)_Y$ nebo $x \in L(Q)_N$ (kde x je kód nějaké instance Q), když předpokládáme, že rozhodnutí $x \in L(Q)$ lze udělat v polynomiálním čase vzhledem k $|x|$.

Definice (Deterministický Turingův stroj)

DTS obsahuje řídící jednotku, čtecí a zápisovou hlavu a (nekonečnou) pásku. Program sestává z:

1. Konečné množiny Γ páskových symbolů, $\Sigma \subset \Gamma$ vstupních symbolů a $* \in \Gamma$ prázdného symbolu
2. Konečné množiny Q stavů řídící jednotky, která obsahuje startovní stav q_0 a 2 terminální stavy q_Y, q_N
3. Přechodové funkce $\delta : (Q \backslash \{q_Y, q_N\}) \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \bullet, \rightarrow\}$

DTS s programem M *přijímá* $x \in \Sigma^*$, právě když pro vstup x se M zastaví ve stavu q_Y. *Jazyk rozpoznávaný programem* M je $L(M) = \{x \in \Sigma^* | M \text{ prijima } x\}$.

DTS s programem M *řeší* problém Q, právě když výpočet M skončí pro každý vstup $x \in \Sigma^*$ a platí $L(M) = L(Q)_Y$.

Nechť M je program pro DTS, který skončí pro $\forall x \in \Sigma^*$. Časová složitost programu M je dána funkcí $T_M(n) = \max\{m | \exists x \in \Sigma^*, |x| = n$, výpočet na DTS s programem M a vstupem x skončí po m krocích stroje$\}$. Pokud existuje polynom p tak, že $T_M(n) \leq p(n) \forall n$, pak M je *polynomiální DTS program*.

Definice (Třída P)

Problém Q je ve třídě P, právě když existuje polynomiální DTS program M, který řeší Q.

Definice (Nedeterministický Turingův stroj)

Stejný jako DTS, ale místo přechodové funkce δ je zde zobrazení δ, které každé dvojici z $Q \times \Gamma$ přiřazuje množinu možných pokračování výpočtu, tj. trojic z $Q \times \Gamma \times \{\leftarrow, \bullet, \rightarrow\}$.

NTS s programem M *přijímá* $x \in \Sigma^*$, právě když existuje přijímající výpočet programu M (tj. běh M, kdy na vstupu je x a končí se ve stavu q_Y). *Jazyk rozpoznávaný programem* M je $L(M) = \{x \in \Sigma^* | M \text{ prijima } x\}$.

Čas, ve kterém M přijímá $x \in \Sigma^*$ definujeme jako počet kroků nejkratšího přijímajícího výpočtu nad daty x.

Časová složitost programu je dána funkcí:

$$T_M(n) = \begin{cases} 1 \text{ neexistuje } x \text{ delky } n, \text{ ktere je prijimano} \\ \max\{m | \exists x \in \Sigma^*, |x| = n, M \text{ prijima } x \text{ v case } m\} \end{cases}$$

Pokud existuje polynom p takový, že $T_M(n) \leq p(n)$, pak M je polynomiální NTS program.

Definice (Třída NP)

Problém Q je ve třídě P, právě když existuje polynomiální NTS program M, který řeší Q. Na rozdíl od deterministického případu netrváme na tom, že výpočet musí skončit i pro nepřijímané instance.

Poznámka (Jiný model NTS)

Přidáme další pásku (orákulum) a stroj pracuje ve 2 fázích:

1. Nedeterministicky hádá – zapíše problém do orákula.
2. Deterministicky ověřuje obsah orákula – práce DTS na původním vstupu plus obsahu orákula.

Je to ekvivalentní s původním – omezíme-li počet možných přechodů NTS na 2 (tím ho jen zpomalíme) a zapisujeme-li do orákula větve pokračování výpočtu (pak stačí na jednu jeden bit), převedeme veškerý nedeterminismus čistě na naplnění orákula.

Definice (Třída co-NP)

Problém Q je ve třídě co-NP, právě když existuje polynomiální NTS program M takový, že $L(M) = L(Q)_N$. O poměru množin co-NP a NP nevíme nic, jen to, že podmnožinou jejich průniku je P.

Převody a NP-úplnost

Definice (Polynomiálně vyčíslitelná funkce)

Funkce $f : \{0,1\}^* \to \{0,1\}^*$ je *polynomiálně vyčíslitelná*, právě když existuje polynom p a algoritmus A takový, že pro každý vstup $x \in \{0,1\}^*$ dává výstup $f(x)$ v čase nejvýše $p(|x|)$.

Definice (Polynomiální převoditelnost)

Jazyk L_1 je *polynomiálně převoditelný* na jazyk L_2 (píšeme $L_1 \propto L_2$), právě když existuje polynomiálně vyčíslitelná funkce f taková, že

$$\forall x \in \{0,1\}^* : x \in L_1 \equiv f(x) \in L_2$$

Definice (NP-těžký, NP-úplný problém)

- Problém Q je *NP-těžký*, právě když $\forall Q' \in \text{NP} : L(Q')_Y \propto L(Q)_Y$.
- Problém Q je *NP-úplný*, právě když je Q NP-těžký a $Q \in \text{NP}$.

Je-li nějaký NP-těžký problém převoditelný na jiný, pak ten musí být také NP-těžký.

4.2 Příklady NP-úplných problémů a převody mezi nimi

Cook-Levinova věta

Existuje NP-úplný problém.

Důkaz pro KACHL

Máme množinu barev B, čtvercová síť S s obvodem obarveným barvami z B a množinu K typů kachlíků, kde je každý typ definován svou horní, dolní, levou a pravou barvou.

Lze síť S vykachlíkovat pomocí kachlíků z množiny K (stejný typ lze použít libovolněkrát, kachlíky ale nelze otáčet) tak, aby:

- barvy kachlíků přilehlé k obvodu sítě souhlasily s barvami předepsanými tomto na obvodu sítě a
- každá dvojice barev na dotyku dvou kachlíků byla rovněž shodná?

NP-úplné problémy

Splnitelnost (SAT)

CNF (booleovská formule v konjunktivní normální formě, tj. konjunkce disjunkcí) F na n proměnných. Existuje pravdivostní ohodnocení proměnných, které splňuje formuli F?

Důkaz transformací **KACHL** $\propto$ **SAT**: pomocí proměnných x_{ijk}, kde $x_{ijk} = 1$, pokud na pozici $[i, j]$ se nachází kachlík typu k. Jednotlivé klauzule se vytvoří tak, aby zaručovaly, že na každé pozici je právě jeden kachlík, že kachlíky navazují horizontálně, vertikálně i na kraje stěny.

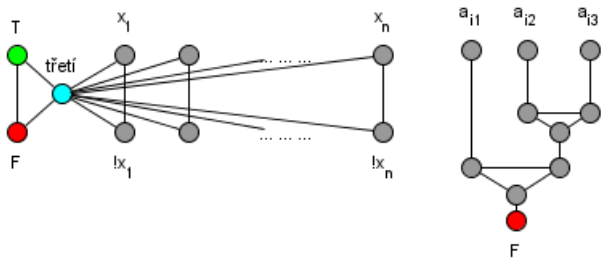
3-SAT

Kubická CNF (vždy jen 3 proměnné v jedné disjunkci) F na n Booleovských proměnných. Existuje pravdivostní ohodnocení proměnných, které splňuje formuli F?

Transformace **SAT** $\propto$ **3-SAT**: stačí každou klauzuli (disjunkci) rozložit s pomocí nových volných proměnných na několik kubických klauzulí: $(a_{i,1} \vee a_{i,2} \vee a_{i,3} \vee \cdots \vee a_{i,k_i})$ odpovídá $(a_{i,1} \vee a_{i,2} \vee y_{i,1}) \wedge (\neg y_{i,1} \vee a_{i,3} \vee y_{i,2}) \wedge (\neg y_{i,2} \ldots) \wedge \cdots \wedge (\neg y_{i,k_i-3} \vee a_{i,k_i-1} \vee a_{i,k_i})$

3-COLOR

Tříobarvení grafu: Mějme neorientovaný graf $G = (V, E)$. Lze obarvit vrcholy ve V třemi barvami tak, aby žádná hrana v E neměla na obou koncích vrcholy stejné barvy?



Obrázek 4.1: Transformace **3-SAT** $\propto$ **3-COLOR**

Transformace **3-SAT** $\propto$ **3-COLOR**: Vytvořím pro všechny proměnné a jejich negace vrcholy grafu a spojím se třemi body (z nichž každý musí být jinak barevný podle obrázku), aby proměnné musely mít barvu T nebo F. Proměnné a negace jsou taky spojené, aby bylo jednoznačně dána hodnota každé z nich. Pro každou klauzuli **3-SAT** přidám grafík podle obrázku (napojím na proměnné, které představují literály klauzule a na druhé straně na barvu F), aby proměnné v něm nešly obarvit FFF.

KLIKA

Mějme neorientovaný graf $G = (V, E)$ a přirozené číslo k. Existuje $V' \subseteq V$, $|V'| = k$, indukující úplný podgraf grafu G?

Transformace **SAT** $\propto$ **KLIKA** – pro každý literál vytvořím bod grafu, spojím všechny body odpovídající literálům různých klauzulí, pokud se nejedná o komplementární proměnné, tj. mezi x_i a $\neg x_i$ nevede hrana.

Nezávislá Množina (NM)

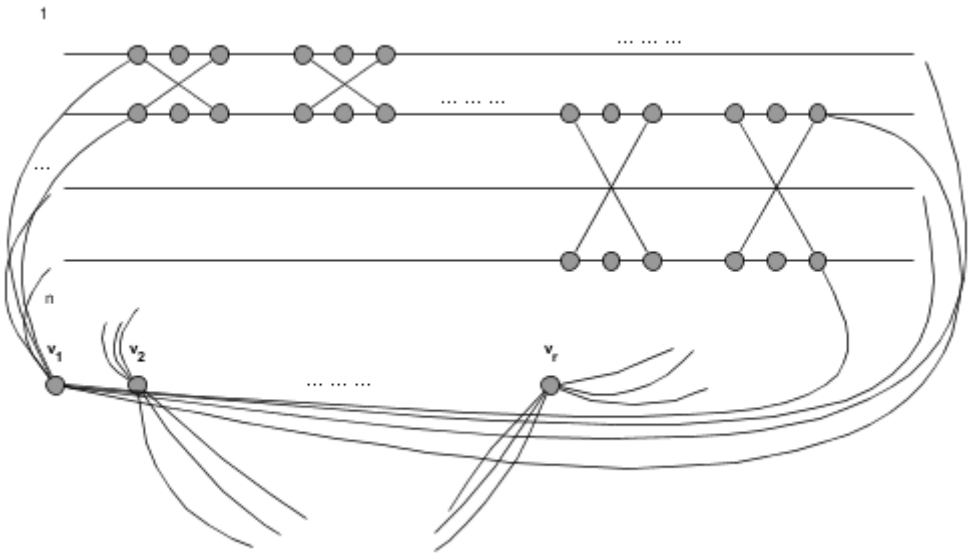
Mějme neorientovaný graf $G = (V, E)$ a přirozené číslo q. Existuje $V' \subseteq V$, $|V'| = q$, taková, že uvnitř V' nejsou žádné hrany?

Transformace **KLIKA** $\propto$ **NM**: stačí prohodit hrany a ne-hrany.

Vrcholové pokrytí (VP)

Máme neorientovaný graf $G = (V, E)$ a přirozené číslo r. Existuje $V' \subseteq V$, $|V'| = r$ taková, že každá hrana má ve V' alespoň jeden vrchol?

Transformace **NM** $\propto$ **VP**: **NM** je doplněk **VP** (vedou-li hrany do **VP**, už nemůžou vést mezi ostatními vrcholy).



Obrázek 4.2: Transformace **VP** $\propto$ **HK**

Hamiltonovská Kružnice (HK)

Máme neorientovaný graf $G = (V, E)$. Obsahuje G *hamiltonovskou kružnici*, tj. jednoduchou kružnici, která prochází každým vrcholem právě jednou?

Transformace **VP** $\propto$ **HK**: Na $|V|$ pomyslných linkách naskládám pro každou hranu původního grafu dvanáctici vrcholů spojených podle obrázku (widget). Krajní body všech linek spojím s vrcholy odpovídající původnímu **VP** $v_1, \ldots, v_r$. Protože widgety lze projít jen částečně (2x po linkách) nebo úplně (jednou všechny), bude **HK** vést částečným průchodem přes widgety, pokud oba vrcholy příslušné jejich hraně původního grafu patří do **VP** a úplným jinak.

Obchodní cestující (TSP)

Máme úplný neorientovaný graf $G = (V, E)$, váhy $w : E \to \mathbb{Z}_0^+$ a číslo $k \in \mathbb{Z}^+$. Existuje v G hamiltonovská kružnice s celkovou váhou nejvýše k? Někdy se počítá nad neúplným grafem a požaduje se hamiltonovský sled, tj. je možné opakovat vrcholy; to se ale na tuto definici snadno převede.

Transformace **HK** $\propto$ **TSP**: stačí nastavit váhy tak, že $w(e) = 1$, pokud e byla v původním grafu a $w(e) = 2$ jinak. Je-li chtěná váha rovna počtu hran původního grafu, řešení dává **HK** v něm.

Součet podmnožiny (SP)

Jsou daná čísla $a_1, \ldots, a_n, b \in \mathbb{Z}^+$. Existuje množina indexů $S \subseteq \{1, \ldots, n\}$ taková, že $\sum_{i \in S} a_i = b$?

Transformace **VP** $\propto$ **SP**: vyrobím incidenční matici grafu (řádky odp. vrcholům, sloupce hranám), kde budou jedničky na místech, kde daná hrana vede z daného vrcholu. Přidám k ní matici, jejíž řádky i sloupce odpovídají hranám a jedničky jsou pouze na diagonále (tj. každá hrana má jedničku ve "svém" řádku a sloupci). "Nalevo" od incidenční matice přidám sloupec plný jedniček. Řádky matice interpretuju jako čísla ve čtyřkové soustavě (v každém sloupci jsou tři jedničky, proto nedojde nikdy k přesunu řádů) a hledám součet podmnožiny jako číslo, které má na začátku velikost **VP** (sečte se ze sloupce jedniček) a následují samé dvojky (pro každou hranu).

4.3 Silná NP-úplnost, pseudopolynomiální algoritmy

Příklad

SP není exponenciální, ale polynomiální v počtu a velikosti čísel. Algoritmus (dynamické programování):

1. Nechť $a_1 \leq a_2 \leq \cdots \leq a_n$ a A je bitové pole délky b (kde 1 na pozici i bude indikovat možnost vytvoření podmnožiny se součtem i).
2. Všechny prvky pole A nastav na 0.
3. Pro i od 1 do n opakuj (hl. cyklus):
   (a) $A[a_i] := 1$

(b) Pro j od a_{i-1} do b zkoušej: když $A[j] = 1$ a $j + a_i \leq b$, nastav $A[j + a_i] := 1$

4. Je-li $A[b] = 1$, podmnožina se součtem rovným b existuje.

Po i-tém průchodu hlavním cyklem obsahuje A jedničky právě u všech součtů neprázdných podmnožin $\{a_1, \ldots, a_i\}$. Důkaz – indukcí. Celk. složitost je $O(n \cdot b)$, což je exponenciální vzhledem k binárně kódovanému vstupu, ale polynomiální, máme-li na vstupu čísla konstantní délky.

Definice (Pseudopolynomiální algoritmus)

Nechť je dán rozhodovací problém Π a jeho instance I. Pak definujeme:

- *kód(I)* – délka zápisu (počet bitů) instance I v binárním kódování (či jiném na něj polynomiálně převoditelném)
- *max(I)* – velikost největšího čísla, vyskytujícího se v I (NE délka jeho binárního zápisu!)

Algoritmus se nazývá *pseudopolynomiální*, pokud je jeho časová složitost omezena polynomem v proměnných $kód(I)$ a $max(I)$. Každý polynomiální algoritmus je tím pádem pseudopolynomiální.

Poznámka (O číselných problémech)

Pokud pro nějaký problém Π platí, že $\forall I : max(I) \leq p(kód(I))$ pro nějaký polynom p, pak všechny pseudopolynomiální algoritmy, řešící tento problém, jsou zároveň polynomiální.

Všechny problémy, kde tato rovnice neplatí(tj. neexistuje p, že by platila), nazýváme *číselné problémy.*

Věta (O pseudopolynomialitě a NP)

Nechť Π je NP-úplný problém a není číselný. Pak pokud P≠NP, nemůže být Π řešen pseudopolynomiálním algoritmem.

Poznámka

Ani ne každý číselný problém je řešitelný pseudopolynomiálním algoritmem.

Věta (O pseudopolynomialitě a podproblémech)

Nechť Π je rozhodovací problém a p polynom. Potom Π_p označme množinu instancí (podproblém) problému Π, pro které platí $max(I) \leq p(kód(I))$. Potom máme-li pseudopolynomiální algoritmus A, který řeší problém Π, určitě existuje polynomiální algoritmus, řešící Π_p. Toto platí pro libovolné p.

Důkaz

Algoritmus A', řešící Π_p v polynomiálním čase, otestuje x na přítomnost v Π_p (spočítá $kód(x)$ a $max(x)$) a pokud $x \in \Pi_p$, chová se stejně jako A, takže běží v čase $q(kód(x), max(x)) \leq q(kód(x), p(kód(x))) = q'(kód(x))$.

Definice (Silně NP-úplný problém)

Rozhodovací problém Π je *silně NP-úplný*, pokud $\Pi \in$NP a existuje polynom p takový, že podproblém Π_p je NP-úplný.

Věta (O silné NP-úplnosti)

Nechť problém Π je silně NP-úplný. Potom, pokud P≠NP, neexistuje pseudopolynomiální algoritmus, který by řešil Π.

Důkaz

Plyne z předchozí věty.

Příklady

TSP je silně NP-úplný. Je to číselný problém, protože váhy hran nejsou omezené. Když váhy na hranách omezím, dostanu NP-úplný podproblém (jde na něj převést **HK**).

3-ROZDĚLENÍ je silně NP-úplné. Problém: máme $a_1, \ldots a_{3m}, b \in \mathbb{N}$ takové, že $\forall j : \frac{1}{4}b \leq a_j \leq \frac{1}{2}b$ a navíc $\sum_{j=1}^{3m} a_j = mb$. Existuje $S_1, \ldots S_m$ disjunktní rozdělení množiny $\{1, \ldots, 3m\}$ takové, že $\forall i : \sum_{j \in S_i} a_j = b$?

Důkaz se provádí převodem z 3DM (třídimenzionální párování na tripartitních grafech), všechna čísla konstruovaná pro převod jsou polynomiálně velká vzhledem ke $|G|$ (v převodu $VP \propto SP$ byla exponenciálně velká).

Kapitola 5

Státnice - Aproximační algoritmy a schémata

5.1 Aproximační algoritmy

Definice (Aproximační algoritmus)

Aproximační algoritmus běží v polynomiálním čase a vrací řešení "blízká" optimu. Je nutné mít nějakou míru kvality řešení. Označme:

- C^* hodnotu optimálního řešení
- C hodnotu nalezenou aproximačním algoritmem

A předpokládejme nezáporné hodnoty řešení.

Definice (Poměrová chyba)

Řekneme, že algoritmus řeší problém s *poměrovou chybou* $\rho(n)$, pokud pro každé zadání velikosti n platí:

$$\max\{\frac{C^*}{C}, \frac{C}{C^*}\} \leq \rho(n)$$

Definice (Relativní chyba)

Řekneme, že algoritmus řeší problém s *relativní chybou* $\varepsilon(n)$, pokud pro každé zadání velikosti n platí:

$$\frac{|C - C^*|}{C^*} \leq \varepsilon(n)$$

Poznámka (O převodu chyb)

Z jedné chyby se dá vyjádřit druhá:

- V případě maximalizační úlohy: $\varepsilon(n) = \frac{C-C^*}{C^*} = \frac{C}{C^*} - 1 = \rho(n) - 1$
- V případě minimalizační úlohy: $\varepsilon(n) = \frac{C^*-C}{C^*} = 1 - \frac{1}{\rho(n)}$

Příklad (Aproximační algoritmy pro vrcholové pokrytí)

- "Brát vrcholy od nejvyššího stupně, dokud nemám celé pokrytí" nemá konstantní relativní chybu – ex. protipříklad, kdy $\rho(k) \leq a \cdot \ln k$
- "Vzít libovolnou hranu, dát do pokrytí její dva konce, odstranit její incidentní hrany a projít tak celé E" má relativní chybu 2 – žádné 2 hrany nemají společný vrchol, tj. mám pokrytí o velikosti $2 \times |\text{mn. disj. hran}|$. Každé vrcholové pokrytí je ale $\geq |\text{mn. disj. hran}|$.

Příklad (Aproximační algoritmy pro TSP)

Omezení na trojúhelníkovou nerovnost – pořád je NP (převod HK→TSP zachovával trojúhelníkovou nerovnost). Algoritmus:

1. Najdi minimální kostru T
2. Zvol lib. vrchol a pomocí DFS nad T v PRE-ORDERu očísluj vrcholy.

3. Cesta (s opakováním) po kostře T přes všechny vrcholy := X. Pak $w(X) = 2w(T)$ (každou hranou kostry jdu tam a zpět).

4. Výslednou HK H vyrobím zkrácením cesty (vypouštěním již navštívených vrcholů), z trojúhelníkové nerovnosti $w(H) \leq w(X)$. Celkem tedy dává $w(H) \leq w(X) \leq 2w(H^*)$, protože $w(T) \leq w(H^*)$ (H^* je kostra, bez jedné hrany).

Bez omezení – pro žádné konstantní ρ neexistuje polynomiální algoritmus, řešící obecný TSP s poměrovou chybou ρ. Mohu totiž mít HK v grafu $G = (V, E)$, pak zadání TSP zkonstruovat jako $K_{|V|}(V, V \times V)$, kde $w(e) = \begin{cases} 1 & e \in E \\ |V|\rho & e \notin E \end{cases}$. Pak by aproximační algoritmus s chybou ρ musel určitě vždy vrátit přesné řešení, takže by musel být NP-těžký.

5.2 Aproximační schémata

Definice (Aproximační schéma)

Aproximační schéma pro optimalizační úlohu je aproximační algoritmus, který má jako vstup instanci dané úlohy a číslo $\varepsilon > 0$, a který pro libovolné ε pracuje jako aproximační algoritmus s relativní chybou ε. Doba běhu může být exponenciální jak vzhledem k n – velikosti vstupní instance, tak vzhledem k $\frac{1}{\varepsilon}$.

Definice (Polynomiální aproximační schéma)

Polynomiální aproximační schéma je takové aproximační schéma, které pro každé pevné $\varepsilon > 0$ běží v polynomiálním čase vzhledem k n (ale stále může být exponenciální vzhledem k $\frac{1}{\varepsilon}$.

Definice (Úplně polynomiální aproximační schéma)

Úplně polynomiální aproximační schéma je polynomiální aprox. schéma, běžící také v polynomiálním čase vzhledem k $\frac{1}{\varepsilon}$ (tj. algoritmus s konstantně-krát menší relativní chybou běží v konstantně-krát delším čase).

Úplná polynomiálna aproximačná schéma pre problém batohu

Zadanie pre "dvojrozmernú variantu" problému batohu je n hmotností predmetov $w_1, w_2, \ldots w_n$, obmedzenie W na celkovú hmotnosť a hodnoty predmetov $v_1, v_2, \ldots, v_n$. Úlohou je nájsť takú množinu $S \subset \{1, 2, \ldots, n\}$, aby $\sum_{i \in S} w_i \leq W$ a $\sum_{i \in S} v_i$ bolo čo najväčšie.

Nech $V = \max\{v_1, v_2, \ldots, v_n\}$. Zadefinujme si $W(i, v)$ ako najmenšia hmotnosť takej podmnožiny predmetov $\{w_1, \ldots w_i\}$, ktorých celková hodnota je práve v. Potom:

$$W(i,v) = \begin{cases} 0 & v = 0 \\ \infty & i = 0, v > 0 \\ W(i-1, v) & v_i > v \\ \min\{W(i-1, v), w_i + W(i-1, v - v_i)\} & \text{inak} \end{cases}$$

V treťom prípade nepridáme vec i, lebo by sme prekročili cenu v (spomeňme si na definíciu $W(i, v)$), v štvrtom ju pridáme ak nám nepokazí hmotnosť.

Pomocou tohto môžeme napísať algoritmus využívajúci dynamické programovanie a ako výsledok vrátime $\max\{v | W(n, v) \leq W\}$. Tento algoritmus bude mať zložitosť n^2V, čo je iba pseudopolynomiálny algoritmus. Ten však môžeme upraviť.

Upravme si zadanie tak, že hodnoty všetkých predmetov orežeme o najnižšie bity. Ak chceme relatívnu chybu $\epsilon > 0$, tak orežeme $b = \lceil log \frac{\epsilon V}{n} \rceil$ bitov (nahradíme ich nulami) (prečo práve toľko bude vysvetlené nižšie). Tým sme získali novú inštanciu problému batohu, kde hmotnosti a limit sú rovnaké, ale pre každé i je hodnota predmetu $v'_i = 2^b \lfloor \frac{v_j}{2^b} \rfloor$. Náš algoritmus bude potrebovať čas $O(\frac{n^2V}{2^b})$ (pretože ignorujeme nulové bity na konci každej hodnoty predmetu). Riešenie S', ktoré dostaneme bude možno rôzne od optima S pôvodnej úlohy, ale bude platiť:

$$\sum_{i \in S} v_i \geq \sum_{i \in S'} v_i \geq \sum_{i \in S'} v'_i \geq \sum_{i \in S} v'_i \geq \sum_{i \in S} (v_i - 2^b) \geq \sum_{i \in S} v_i - n2^b$$

Prvá nerovnosť platí, lebo S je optimum v pôvodnej úlohe, druhá lebo $v'_i \leq v_i$, tretia lebo S' je optimum novej úlohy, štvrtá lebo $v'_i \geq v_i - 2^b$ a posledná lebo $|S| \leq n$. Naše riešenie je teda najviac $n2^b$ pod optimom. Dolný odhad optima je V (predpokladáme, že každý predmet sa samotný vmestí do batoha, ináč môžeme príliš ťažké predmety vyhodiť ako preprocessing). Relatívna chyba je teda $\frac{\text{approx}-\text{opt}}{\text{opt}} \leq \frac{\text{approx}-\text{opt}}{V} \leq \frac{n2^b}{V} = \epsilon$

Takže pre zadané ϵ orežeme $b = \lceil log \frac{\epsilon V}{n} \rceil$ bitov a dostaneme algoritmus, ktorého relatívna chyba je ϵ a jeho časová zložitosť je $O(\frac{n^2V}{2^b}) = O(\frac{n^3}{\epsilon})$, čo je polynomiálne aj v n aj v $\frac{1}{\epsilon}$, preto je to úplná polynomiálna aproximačná schéma.

Kapitola 6

Státnice - Algoritmicky vyčíslitelné funkce

6.1 Částečně rekurzivní funkce

K. Gödel v 30. letech vynalezl primitivní rekurzivní funkce, později společně s dalšími částečně rekurzivní funkce. Jde o funkcionální přístup k algoritmům. Lze se na ně dívat i jako na logiku 1. řádu: základní funkce jsou axiomy, máme operátory – odvozovací pravidla – a z toho vyrábíme formule – rekurzivní funkce.

Definice (Podmíněná rovnost, konvergence, divergence)

- $\simeq$ značí "podmíněnou rovnost", tj. v případě, že alespoň jedna strana má smysl, tak má smysl i druhá a rovnají se.
- $P_1(D)\downarrow$ značí, že predikát *je definován*, tj "konverguje" (občas se značí ! místo $\downarrow$)
- $P_1(D)\uparrow$ značí, že predikát *není definován*, tj. "diverguje"

Značky konvergence, divergence i podmíněné rovnosti se vztahují jak na predikáty, tak na funkce.

Definice (Základní funkce)

- $o(x) \simeq 0 \quad \forall x \in \mathbb{N}$ ("nula")
- $s(x) \simeq x + 1 \quad \forall x \in \mathbb{N}$ ("následník")
- $I_n^j(x_1, \ldots, x_n) \simeq x_j \quad 1 \leq j \leq n$ ("projekce", vybrání jedne ze složek)

Definice (Základní operátory)

- $R_n \quad (n \geq 1)$ – *primitivní rekurze*
  Funkcím f ($n-1$ proměnných) a g ($n+1$ proměnných) přiřadí $R_n(f,g) = h$, kde $h(0, x_2, \ldots, x_n) \simeq f(x_2, \ldots, x_n)$ a $h(y+1, x_2, \ldots, x_n) \simeq g(y, h(y, x_2, \ldots, x_n), x_2, \ldots, x_n)$ (analogické k for-cyklu).
- S_n^m – *substituce*
  Funkci f (m proměnných) a m funkcím g_i (všechny n proměnných) přiřadí funkci h (n proměnných) předpisem $h = S_n^m(f, g_1, \ldots, g_n) \equiv h(x_1, \ldots, x_n) \simeq f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ (analogické k podprogramu).
- M_n – *minimalizace*
  Funkci f ($n+1$ proměnných) přiřadí h (n proměnných) tak, že

  $$h(x_1, \ldots, x_n)\downarrow \wedge h(x_1, \ldots, x_n) \simeq y \equiv f(x_1, \ldots, x_n, y)\downarrow, \simeq 0 \wedge f(x_1, \ldots, x_n, j)\downarrow, \not\simeq 0 \ \forall j < y$$

  (analogické k while-cyklu).

Další značení:

- $\mu_y P(x,y)$ je funkce proměnné x, která vrátí nejmenší y takové, aby platil predikát $P(x,y)$. Lze jí sestrojit pomocí operátoru minimalizace.

Definice (Třída primitivně a částečně rekurzivních funkcí)

- *Třída primitivně rekurzivních funkcí* je nejmenší třída funkcí $f : \mathbb{N}^k \to \mathbb{N}$, která obsahuje základní funkce a je uzavřená na R_n a S_n^m.
- *Třída částečně rekurzivních funkcí* je nejmenší třída, která obsahuje zákl. funkce a je uzavřená na R_n, S_n^m a M_n.

Poznámka (Vlastnosti zákl. funkcí a operátorů)

- Všechny zákl. funkce jsou všude definované ("totální") a efektivně vyčíslitelné.
- Všechny zákl. operátory zachovávají efektivní vyčíslitelnost.
- R_n, S_n^m zachovávají totálnost.
- PRF jsou efektivně vyčíslitelné a totální.

Definice (Odvození funkce)

Odvození funkce je konečná posloupnost funkcí, z nichž každá je buď funkce základní, nebo vzniká z už odvozených funkcí pomocí nějakého operátoru. Ke každé funkci si pamatujeme, jak vznikla (toto v praxi hraje roli programu).

Definice (Obecně rekurzivní funkce)

Funkce je obecně rekurzivní (ORF), jestliže je ČRF a totální.

Operace s PRF, predikáty

Poznámka (Některé PRF)

Pomocí PRF lze popsat např.:

- součet
- součin
- mocninu, faktoriál
- operaci $x \dot{-} y$, kde $x \dot{-} y = x - y$ pro $x \geq y$, jinak 0
- operátory sg a $\overline{\text{sg}}$ (testy na nenulovost, resp. nulovost argumentu)
- minimum, maximum, absolutní hodnotu rozdílu

Definice (Charakteristická funkce)

Mějme predikát P (libovolné tvrzení) o n proměnných. Potom c_P je jeho *charakteristická funkce*, když je to všude definovaná funkce daná následovně:

$$c_P(x_1, \ldots, x_n) \simeq \begin{cases} 1 & \text{pokud } P(x_1, \ldots, x_n) \\ 0 & \text{jinak} \end{cases}$$

Částečná charakteristická funkce pro nějaký predikát P o n proměnných je funkce f o n proměnných taková, že $f(x_1, \ldots, x_n) \downarrow \Leftrightarrow P(x_1, \ldots, x_n)$ a $f(x_1, \ldots, x_n) \downarrow \Rightarrow f(x_1, \ldots, x_n) = 1$.

Definice (PR, OR, RS Predikáty)

Řekneme, že predikát je *primitivně (obecně) rekurzivní*, jestliže jeho charakteristická funkce je primitivně (obecně) rekurzivní. Predikát je *rekurzivně spočetný*, jestliže jeho částečná charakteristická funkce je částečně rekurzivní.

S funkcemi a predikáty se operuje docela nedůsledně, dají se v podstatě ztotožnit.

Poznámka (Jiná možnost nahlížení)

ČRF odpovídají funkcionální logice 1. řádu:

- termy číselné: $0, x, x+1, \ldots$
- termy funkční: $o, I_1^1, s, R_2(I_1^1, S_3^1(s, I_3^2)), \ldots$
- pravidlo aplikace: $Ap(f, x) = \cdots = f(x)$ (kde "..." je proces vyhodnocení termu, potenciálně nekonečný, dává z funkce číselný term)
- pravidlo zobecnění: $\lambda xy(x+y)$ dává z číselného termu $x+y$ funkci

Poznámka (Operace zachovávající PR)

PR jsou:

- Rozšíření počtu proměnných, konstantní funkce
- Permutace a ztotožnění proměnných
- Kódování $\mathbb{N}^k$ do $\mathbb{N}$ – iterace Cantorova diagonálního kódování dvojic ($\langle x,y\rangle_2 = \frac{(x+y)(x+y+1)}{2} + x$)
- Opačná operace – dekódování
- Funkce $p(i)$ – i-té prvočíslo
- Predikát rovnosti a $<, >$
- Logické spojky $\vee, \wedge, \neg$, omezené kvantifikátory (kvantifikace spočetně mnoha prvků)
- Gödelovo prvočíselné kódování: slovo $a_{i_0} \dots a_{i_k}$ do $p(0)^{i_0} \cdot \dots \cdot p(k)^{i_k}$

Ackermannova funkce

Definice (Ackermannova funkce)

Ackermannova funkce je funkce definovaná jako:

$$A(0,x) = \begin{cases} 1 & x = 0 \\ 2 & x = 1 \\ x+2 & x > 1 \end{cases}$$

$$A(y,0) = 1$$

$$A(y+1,x+1) = A(y, A(y+1,x))$$

Definice (Strukturální složitost)

Definujeme strukturální složitost – hloubku rekurze (intuitivně: počet vnořených for-cyklů – syntakticky, ne výpočtem) jako 0 pro základní funkce a

$$h(R_n(P,Q)) = \max(h(P), h(Q)+1),\ h(S_n^m(P,Q_0,\dots,Q_k)) = \max(h(P), h(Q_0), \dots, h(Q_k))$$

Pak $\mathcal{R}_i$ je třída PRF, které lze získat pomocí PR-termů hloubky $\leq i$ a PRF samo je $\cup_{i=1}^{\infty}\mathcal{R}_i$

Věta (O Ackermannově funkci)

Ackermannova funkce není PRF, ale je ORF.

Důkaz

- Určitě je ORF – důkaz se provádí transfinitní indukcí typu ω^2; pro výpočet každé hodnoty potřebuji jen konečně mnoho předchozích hodnot – stačí mi μ_z, kde z je nejmenší kus $\mathbb{N}^2$, který stačí k výpočtu $A(y,x)$ (dá práci dokázat, že je konečný, potřeba ordinálů, lexikografického uspořádání).
- A roste rychleji než každá PRF: $\forall\varphi$ PRF (jedné proměnné) $\exists x_0 : \forall x \geq x_0 : \varphi(x) < A(x,x)$.
- Uvažujme $A(y,x)$ jako matici funkcí $f_y(x)$. Potom určitě $f_i \in \mathcal{R}_i \backslash \mathcal{R}_{i-1}$ a $f_y(x)$ je (až na konečně mnoho x) rostoucí. Navíc pro libovolnou $\varphi \in \mathcal{R}_i$ existuje x_0 takové, že $\forall x \geq x_0 : \varphi(x) < f_{i+1}(x)$, tedy f_{i+1} majorizuje všechny funkce z $\mathcal{R}_i$
- Nechť pro spor má $A(x,x)$ hloubku i. Potom $A(x,x) = \varphi(x) < f_{i+1}(x)$ pro nějaké pevné i. Potom ale $A(x,x) < f_{i+1}(x) < f_x(x) = A(x,x)$, tj. pro $x > i+1$ máme spor.

Věta (O vztahu PRF, ORF a ČRF)

Platí PRF$\subset$ ORF $\subset$ ČRF a inkluze jsou ostré.

Důkaz

Pro ORF$\subset$ ČRF mám funkci $g(x,y) \simeq y+1$ a $h(x) \simeq \mu_y(g(x,y) \simeq 0)$, ta není nikde definovaná. Pro PRF$\subset$ ORF mám Ackermannovu funkci.

6.2 Univerzální funkce

Definice (Univerzální funkce)

Mějme $\mathcal{T}$ – spočetnou množinu ČRF jedné proměnné. Potom $\mathcal{U}(i,x)$ je *univerzální funkce* třídy $\mathcal{T}$, jestliže:

- $\forall$ přirozené $i : \lambda x\mathcal{U}(i,x) \in \mathcal{T}$
- $\forall \varphi \in \mathcal{T} : \exists i_0 : \varphi = \lambda x\mathcal{U}(i_0,x)$

A $\mathcal{U}$ tedy indexuje všechny funkce třídy $\mathcal{T}$. Podobně se definují i univerzální funkce pro ČRF více proměnných. Platí, že $\{\lambda x\mathcal{U}(i,x)\}_{y\geq 0}$ je posloupnost všech funkcí z $\mathcal{T}$, takže $\mathcal{U}$ *určuje numeraci* prvků $\mathcal{T}$

Věta (O univerzální funkci PRF)

Existuje ORF, která je univerzální pro třídu PRF (jedné proměnné). Taková funkce pak nemůže být PRF.

Důkaz

- Seřadím všechny PR-termy (PR-programy) do posloupnosti (máme 3 axiomy a 3 odvozovací pravidla, seřazení je možné).
- Potom $U(x,y) := h_x(y)$, kde h_x vyčísluje x-tý program.
- Sporem nechť $U(x,y)$ je PRF. Pak i $U(x,x)$ je PRF, $1\dot{-}U(x,x)$ je PRF, z toho $1\dot{-}U(x,x) = U(x_0,x)$. Dosadím $x = x_0$ a mám spor, neboť obě strany jsou definovány (toto je příklad použití Cantorovy diagonální metody).

Definice (Turingovsky vyčíslitelná funkce)

Vezmeme Turingovy stroje s vnější abecedou, jejíž prvním znakem je "|". Čísla $0, 1, \ldots$ zapisujme na pásku jako $|, ||, |||, \ldots$, n-tice oddělujme znakem λ. Potom:

- Řekneme, že stroj M je *n-aritmetický*, pokud pro každou n-tici přir. čísel $x_1, \ldots, x_n$ reprezentovanou počáteční konfigurací S platí: je-li M *použitelný* k S (zastaví-li se výpočet nad ní) a je-li výsledná konfigurace T, pak v T je na pásce nějaké jedno přirozené číslo a hlava stroje M stojí nad jeho posledním znakem |.
- Stroj je dále n-aritmetický *typu* 0/1, pokud má abecedu $\{|, \lambda\}$ a jediný koncový stav.
- Řekneme, že M *vyčísluje funkci* f o n proměnných, pokud M je n-aritmetický a pro každou n-tici přir. čísel $x_1, \ldots, x_n$ v poč. konfiguraci S platí: M je použitelný, právě když je f pro $x_1, \ldots, x_n$ definovaná a je-li f definovaná, pak ve výsledné konfiguraci T je na pásce stroje číslo $f(x_1, \ldots, x_n)$ a hlava stojí nad jeho posledním znakem.
- Řekneme, že funkce je *turingovsky vyčíslitelná*, pokud existuje nějaký n-aritmetický TS (typu 0/1), který ji vyčísluje.

Věta (O ekvivalenci TS a ČRF)

Funkce n proměnných je částečně rekurzivní, právě když existuje n-aritmetický Turingův stroj typu 0/1, který ji vyčísluje.

Důkaz

"⇐": Každá ČRF je T-vyčíslitelná.

Důkaz indukcí podle složitosti funkce – pro základní funkce to jistě platí, R_n, M_n, S_n^m toto zachovávají (S_n^m znamená použití více pásek, vyčíslení a složení, R_n znamená vyčíslení f a y-krát "otočení" g, M_n je vyčíslování f na vstupu a zvětšujícím se počítadle cyklů, dokud nedostanu 0 – pak vrátím hodnotu počítadla).

"⇒": Pro každý TS M existuje ČRF, která dává stejný výsledek

Je nutné zavést kódy konfigurací; TS navíc nemají žádné podtřídy, tedy nelze postupovat induktivně. Platí:

- $\text{step}_M(X)$ – jeden krok stroje je PR záležitost (pracuje se nad konfiguracemi TS $UqsV$, z obou stran obalenými spec. znakem h, pak slovo není nekonečné a lok. změna se dá spočítat). Existuje určitě PR funkce, která popisuje lokální změnu (jde vlastně o rozhodovací strom, který se staví pomocí R_n).
- $\text{comp}_M(X,i)$ – výsledek stroje po i krocích práce je stále PR (for-cyklus – R_n)
- $\mu_i(\text{comp}_M(X,i)$ obsahuje $q_0)$ – q_0 je koncový stav (pracuj, dokud neskončíš – while)
- Potom výsledná ČRF g je dána jako $g(\text{kód}(S)) \simeq \text{result}(\mu_i(\text{comp}_M(X,i)$ obsahuje $q_0))$, kde result je jednoduchá funkce smazání okrajů atp. BÚNO je takový stroj úplný a q_0 jeho jediný koncový stav. Operátor minimalizace se vyskytuje jen jednou, proto je vhodné ho vysunout co nejvíce "ven" v uzávorkování.

Pak také platí, že mám-li nějakou částečnou funkci (tj. nemusí být totální), která je turingovsky vyčíslitelná, pak je ČR.

Kleenova věta

Věta (Kleenova o normální formě)

Pro každé $k \geq 1$ existují

- ČRF Ψ_k $k+1$ proměnných
- PRP T_k $k+2$ proměnných (*Kleeneův predikát*)
- PRF U jedné proměnné
- PRF s_k $k+1$ proměnných

takové, že:

1. Ψ_k je univerzální funkcí pro třídu všech ČRF k proměnných. $\Psi_k(e, x_1, \ldots, x_k)$ vyčísluje e-tou ČRF k proměnných. Navíc z odvození ČRF lze efektivně získat e a naopak z e lze efektivně získat odvození příslušné ČRF.
2. $\Psi_k(e, x_1, \ldots, x_k) \simeq U(\mu_y T_k(e, x_1, \ldots, x_k, y))$, kde T_k odpovídá výpočtu Turingova stroje, $y = \langle y_0, y_1 \rangle$, y_0 je doba výpočtu, y_1 výsledek a U vydělí z $\langle y_0, y_1 \rangle$ druhou složku.
3. s_k je prostá funkce rostoucí ve všech proměnných, o které platí (tato část Věty o normální formě se nazývá *S-m-n věta*):
   $\Psi_{m+n}(e, z_1, \ldots, z_m, x_1, \ldots, x_n) \simeq \Psi_n(s_m(e, z_1, \ldots, z_m), x_1, \ldots, x_n)$
   $T_{m+n}(e, \vec{z}, \vec{x}) \equiv T_n(s_m(e, \vec{z}), \vec{x})$
4. $T_k(e, x_1, \ldots, x_k, y) \wedge T_k(e, x_1, \ldots, x_k, z) \Rightarrow y = z$

Díky tomu lze ČRF efektivně očíslovat. $\varphi_e(x_1, \ldots, x_k)$ pak značí e-tou funkci k proměnných. Indexu e se říká *Gödelovo číslo* funkce.

Důkaz

- Oklikou přes Univerzální Turingův stroj: ke každé ČRF máme TS a jeho kód e. Vezmeme si proto UTS, který s kódy umí počítat, a hledáme jeho ČRF.
- Páska univerzálního stroje vypadá v obecném případě následovně:

  $$Y \text{ blok1 } Y \text{ blok2 } \Delta \text{ blok3 } \times O_1 \times O_2 \ldots Y$$

  První blok je aktuální konfigurace, druhý číslo stavu a třetí aktuální políčko, zbytek je program. Čísla kódujeme unárně (x jako $x+1$ čar).
- Základní idea – bez proměnných $x_1, \ldots x_k$ páska UTS vypadá takto: $Y\ M\ Y$ blok2 Δ blok3 $\times O_1 \times O_2 \ldots Y$ (M je kód programu).
- Konstrukce $\Psi_m(e, x_1, \ldots, x_m)$:
  - Zkontrolujeme, zda e po rozkódování obsahuje nějaký kód programu M.
  - Jestliže ne, je výsledkem nulová funkce (syntax error).
  - Jestliže ano, nejlevější výskyt M nahradíme $||\ldots|\lambda||\ldots|\lambda\ldots\lambda||\ldots|M$ (kódování vstupních dat $x_1, \ldots, x_n$; substituce) a spustíme program e na UTS, podle toho získáme výsledek – Ψ_k
- $s_k(e, y_1, \ldots, y_k)$ odpovídá: čekej na $x_1, \ldots, x_j$, přidej k nim $y_1, \ldots, y_k$ a spusť program e.

Věta (Vlastnosti predikátu Ψ_k)

1. Predikát $\Psi_k(e, x_1, \ldots, x_k)\downarrow$ je rekurzivně spočetný, není rekurzivní.
2. jeho negace $\Psi_k(e, x_1, \ldots, x_k)\uparrow$ není rekurzivně spočetná.
3. Dále Ψ_k nelze rozšířit do ORF. Dokonce pokud α je ČRF, která je rozšířením Ψ_k, potom lze efektivně nalézt vstup $\vec{z}$ takový, že $\alpha(\vec{z})\uparrow$.

Univerzální funkce pro danou třídu funkcí tedy buď nemůže patřit do této třídy, nebo nemůže být totální.

Důkaz

- Z definice je zřejmé, že $\Psi_k(\ldots)\downarrow$ je rekurzivně spočetný predikát. Stačí ukázat, že $\Psi_k(\ldots)\uparrow$ není rekurzivně spočetný. Z toho přímo plyne, že $\Psi_k(\ldots)\downarrow$ není rekurzivní.
- Bez újmy na obecnosti uvažujme k=1. Použijeme Cantorovu diagonální metodu.
- Kdyby $\Psi_1(\ldots)\downarrow$ byl rekurzivní, potom by $\Psi_1(x,x)\uparrow$ byl také rekurzivní, tím spíše rekurzivně spočetný. Tedy pro nějakou ČRF φ by platilo $\Psi_1(x,x)\uparrow\Leftrightarrow \varphi(x)\downarrow$. Vezmeme-li index funkce φ (označme jej x_0), dostáváme $\Psi_1(x,x)\uparrow\Leftrightarrow \Psi_1(x_0,x)\downarrow$, po dosazení $x = x_0$ dostáváme $\Psi_1(x_0,x_0)\uparrow\Leftrightarrow \Psi_1(x_0,x_0)\downarrow$, což je spor.
- Pro důkaz zbytku tvrzení předpokládejme, že $h(e,x)$ je ORF rozšířením $\Psi_1(e,x)$. Potom $1\dot{-}h(x,x)$ je ORF g. Nechť g má index x_0, tj. $g(x) \simeq \Psi_1(x_0,x)$. Protože g je ORF, pro všechna x platí $\Psi_1(x_0,x)\downarrow$, tedy $\Psi_1(x_0,x_0)\downarrow$. Dostáváme $h(x_0,x_0) = \Psi_1(x_0,x_0)$, což ovšem vede ke sporu: $1\dot{-}\Psi_1(x_0,x_0) \simeq h(x_0,x_0) \simeq \Psi_1(x_0,x_0)$.
- Pokud nějaká ČRF β je rozšířením Ψ_1, umím pro β (podle předch. důkazu) najít e takové, že $\beta(e,e)\uparrow$.
- Myšlenka obsažená v předchozím důkazu je založená na Cantorově diagonální metodě. Spor na diagonále si vynutí divergenci, neboť rovnost funkcí je jenom podmíněná, tedy v případě divergence je vše v pořádku.

Kapitola 7

Státnice - Rekurzivní a rekurzivně spočetné množiny

7.1 Rekurzivně spočetné množiny

Definice (Rekurzivní a rekurzivně spočetná množina)

Charakteristická funkce množiny M označuje charakteristickou funkci predikátu náležení do množiny, tj. funkci $c_M(x)$, kde $c_M(x) =\downarrow 1$ pro $x \in M$ a $c_M(x) \downarrow= 0$ pro $x \notin M$.

Analogicky se definuje *částečná charakteristická funkce* množiny – $c_M(x) =\downarrow 1$ pro $x \in M$ a $c_M(x) \uparrow$ pro $x \notin M$.

Množina M je *rekurzivní*, je-li její charakteristická funkce obecně rekurzivní (každá char. fce je totální, takže ČRF by bylo totéž). Množina M je *rekurzivně spočetná*, jestliže je definičním oborem nějaké ČRF (neboli jestliže je její částečná char. funkce částečně rekurzivní).

Množina je rekurzivní, jestliže existuje program, který se na libovolném vstupu zastaví a rozhodne, zda do ní vstup patří. Množina je rekurzivně spočetná, jestliže existuje program, který se zastaví právě na jejích prvcích. Je-li množina rekurzivní, je i rekurzivně spočetná, opačně to neplatí.

Definice (dom, rng)

V následujícím *dom* značí *definiční obor*, *rng obor hodnot*.

Definice (x-tá rekurzivně spočetná množina)

$$W_x \ (x\text{-tá rekurzivně spočetná množina}) = dom(\varphi_x) = \{y : \varphi_x(y)\downarrow\}$$

Definice (K)

$$K = \{x : x \in W_x\} = \{x : \varphi_x(x)\downarrow\} = \{x : \Psi_1(x, x)\downarrow\}$$

Množina K vlastně odpovídá *halting problému*. Platí o ní následující tvrzení.

Věta (Rekurzivní spočetnost K)

Množina K je rekurzivně spočetná, není rekurzivní, $\overline{K}$ není rekurzivně spočetná.

Důkaz

K není rekurzivní, neboť $\overline{K}$ není rekurzivně spočetná. $\overline{K}$ není rekurzivně spočetná, neboť kdyby byla, měla by index x_0. Jednoduchou diagonalizací dostáváme $x_0 \in \overline{K} \Leftrightarrow x_0 \in W_{x_0} \Leftrightarrow x_0 \in K$. Spor.

7.2 1-převeditelnost, m-převeditelnost

Definice (1-převeditelnost, m-převeditelnost, 1-úplnost, m-úplnost)

- Množina A je *1-převedilná* na B (značíme $A \leq_1 B$), jestliže existuje prostá ORF f taková, že $x \in A \Leftrightarrow f(x) \in B$.
- Množina A je *m-převedilná* na B (značíme $A \leq_m B$), jestliže existuje ORF f (ne nutně prostá) taková, že $x \in A \Leftrightarrow f(x) \in B$.

- Množina M je *1-úplná*, jestliže M je rekurzivně spočetná a každá rekurzivně spočetná množina je na ni 1-převedilná.
- Množina M je *m-úplná*, jestliže M je rekurzivně spočetná a každá rekurzivně spočetná množina je na ni m-převedilná.

Věta (1-úplnost K)

K je 1-úplná. Tedy *halting problem* je vzhledem k 1 a m-převoditelnosti nejtěžší mezi rekurzivně spočetnými problémy.

Důkaz

Mějme libovolnou rekurzivně spočetnou množinu W_x.

Mějme ČRF $\alpha(y,x,w)$, popisující x-tou rekurzivně spočetnou množinu. Tedy $\alpha(y,x,w)\downarrow \Leftrightarrow y \in W_x \Leftrightarrow \Psi_1(x,y)\downarrow \Leftrightarrow \varphi_x(y)\downarrow$. w je tady fiktivní proměnná, funkce α na její hodnotě nezáleží. Z s-m-n věty dostáváme: $\alpha(y,x,w) \simeq \Psi_3(a,y,x,w) \simeq \Psi_1(s_2(a,y,x),w) \simeq \varphi_{s_2(a,y,x)}(w)$. Označme $h(y,x) = s_2(a,y,x)$ (s_2 je PRF, tím spíše ORF). $y \in W_x \Leftrightarrow \alpha(y,x,w)\downarrow \Leftrightarrow \varphi_{h(y,x)}(w)\downarrow \Leftrightarrow \varphi_{h(y,x)}(h(y,x))\downarrow \Leftrightarrow h(y,x) \in K$ Zde jsme mohli za w dosadit $h(y,x)$, neboť hodnota α na w nezáleží! Tedy $W_x \leq_1 K$ pomocí funkce $\lambda y : h(y,x)$.

Lemma (K_0 je 1-úplná)

$K_0 = \{\langle y,x\rangle : y \in W_x\}$ je 1-úplná.

Důkaz

Zřejmé. $K \leq_1 K_0$ a K je 1-úplná.

Lemma (Poznámky k 1-převeditelnosti)

1. Relace $\leq_1$ a $\leq_m$ jsou tranzitivní, reflexivní.
2. $A \leq_1 B \Rightarrow A \leq_m B$
3. B rekurzivní, $A \leq_m B \Rightarrow A$ rekurzivní.
4. B rekurzivně spočetná, $A \leq_m B \Rightarrow A$ rekurzivně spočetná.

Důkaz

1. Zřejmé.
2. Zřejmé.
3. Složením funkce dokazující $\leq_m$ s procedurou, která rozhoduje o $x \in B$, dostaneme proceduru rozhodující o $x \in A$. Dostáváme $c_A(x) = c_B(f(x))$.
4. Stejně.

Důsledek

K a $\overline{K}$ jsou m-nesrovnatelné.

Důkaz

Plyne z faktu, že K je rekurzivně spočetná, $\overline{K}$ není, a z bodu 4 předchozího lemma.

Definice (Rekurzivní permutace)

Permutace na $\mathbb{N}$, která je ORF, se nazývá *rekurzivní permutace*.

Definice (Rekurzivní isomorfismus)

Množiny A a B jsou *rekurzivně isomorfní*, jestliže existuje rekurzivní permutace p taková, že $p(A) = B$. Značíme $A \equiv B$.

Definice (1-ekvivalence a m-ekvivalence)

- $A \equiv_1 B$, jestliže $A \leq_1 B \wedge B \leq_1 A$.
- $A \equiv_m B$, jestliže $A \leq_m B \wedge B \leq_m A$.

Věta (Myhillova)

$$A \equiv B \Leftrightarrow A \equiv_1 B$$

Důkaz

Jedná se o vlastně o obdobu Cantor-Bernsteinovy věty.

$\Rightarrow$ Triviální.

$\Leftarrow$ Z předpokladů máme dvě prosté ORF f, g převádějící vzájemně A na B a opačně. Chceme sestrojit rekurzivní permutaci h takovou, že $h(A) = B$.

Plán: v krocích budeme generovat graf h tak, že v kroku n bude platit $\{0, \ldots, n\} \subseteq dom(h), \{0, \ldots, n\} \subseteq rng(h)$.
Z toho plyne, že h bude definovaná na celém $\mathbb{N}$ a bude na. Současně zajistíme, že h bude prostá.
Navíc budeme chtít, aby platilo $y \in A \Leftrightarrow h(y) \in B$, tedy aby h převáděla A na B.
Začneme v bodě 0 a položíme $h(0) = f(0)$. Rozlišíme následující případy:

1. $f(0) = 0$: vše je v pořádku, $h(0) = f(0) = 0$ a $0 \in A \Leftrightarrow 0 \in B$, pokračujeme dalším prvkem.
2. $f(0) \neq 0$: rozlišíme dva podpřípady
   (a) $g(0) \neq 0$: definujeme $h(g(0)) = 0$.
   Tedy $0 \in dom(h) \cap rng(h)$.
   (b) $g(0) = 0$: nemůžeme použít $h(g(0)) = 0$, protože v bodě 0 je již h definována. Najdeme tedy volný bod: definujme $h(g(f(0))) = 0$. Určitě $g(f(0)) \neq 0$, protože g je prostá a $f(0) \neq 0$. Tímto jsme opět dostali bod 0 do definičního oboru h i oboru hodnot. Zároveň funkci h definujeme podle f a g, tedy převádí vzájemně A na B.

Indukční krok: nechť v kroku k je z první volný prvek. Všechna čísla menší než z máme v $dom(h) \cap rng(h)$. Podíváme se, zda je $f(z)$ volný. Jestliže ano, položíme $h(z) = f(z)$. Jestliže $f(z)$ není volný, hledám "cik-cak" další volný (podobně jako pro 0, maximálně z prvků je blokovaných, tj. maximálně po z iteracích tohoto postupu dojdu k volnému prvku).

Důsledek

$K \equiv K_0$.

Důkaz

Zřejmé, neboť $K \equiv_1 K_0$ (obě množiny jsou 1-úplné).

7.3 Rekurzivně spočetné predikáty

Lemma (ORF $\rightarrow$ RSP)

Je-li Q obecně rekurzivní predikát, potom $\exists y : Q$ je rekurzivně spočetný predikát.

Důkaz

$\mu_y Q$ je ČRF, její definiční obor je $\{\exists y : Q\}$.

Věta (Univerzální RSP)

Predikát $\exists y T_k(e, x_1, \ldots, x_k, y)$ je univerzálním RSP pro třídu RSP k proměnných, tj. lze definovat index (*Gödelovo číslo*) rekurzivně spočetného predikátu.

Důkaz

Z věty o normální formě – numerace ČRF nám dává numeraci predikátů.

Věta (Log. spojky a rek. spočetnost)

Konjukce a disjunkce zachovávají rekurzivní spočetnost. Tedy průnik a sjednocení rekurzivně spočetných množin je rekurzivně spočetná množina. Stejně pro predikáty.

Důkaz

Pro průnik spustíme oba programy současně a čekáme, až se oba zastaví. Pro sjednocení čekáme, až se zastaví alespoň jeden.

Formálně pro průnik ($(w)_{2,1}$ znamená to, že w kóduje usp. dvojici a vybíráme z ní první prvek; to je PRF): $\exists s_1 T_k(a,\vec{x},w_1) \wedge \exists s_2 T_k(b,\vec{x},w_2) \Leftrightarrow \exists w(T_k(a,\vec{x},(w)_{2,1}) \wedge T_k(b,\vec{x},(w)_{2,2}))$. Uvedený predikát je rekurzivně spočetný, tedy má nějaký index, tj. ekvivalence pokračuje: $\exists w T_{k+2}(e,a,b,\vec{x},w) \Leftrightarrow \exists w T_k(s_2(e,a,b),\vec{x},w)$

Poznámka

Konjunkce a disjunkce tedy rek. spočetnost zachovávají, o negaci (tj. doplňku) to ale už samozřejmě neplatí.

Věta (Kvantifikace a rek. spočetnost)

Omezená kvantifikace $(\forall y)_{y\leq t}$ a existenční kvantifikace (pro $k \geq 2$) zachovávají rekurzivní spočetnost.

Důkaz

Neformálně: omezený kvantifikátor lze zkontrolovat for cyklem.

Formálně: $(\forall y)_{y\leq t}\ \exists s : T_k(e,x_1,\ldots,x_{k-1},y,s) \Leftrightarrow \exists$ kód $(t+1)$-tice w : $(\forall y)_{y\leq t}\ T_k(e,x_1,\ldots,x_{k-1},y,(w)_{t+1,y})$.

y můžeme zkoušet primitivní rekurzí, w minimalizací, dostáváme tedy rekurzivně spočetný predikát, který má nějaký index b, dále můžeme použít S-m-n větu. $\exists s : T_{k+1}(b,e,x_1,\ldots,x_{k-1},t,s) \Leftrightarrow \exists s : T_k(s_1(b,e),x_1,\ldots,x_{k-1},t,s)$.

Pro existenční kvantifikátor je situace ještě jednodušší. Kvantifikaci přes dvě proměnné převedeme na kvantifikaci přes jednu, kterou budeme považovat za kód dvojice a v predikátu potom vydělíme jednotlivé složky (a použijeme opět S-m-n větu). Dostáváme predikát $k-1$ proměnných, proto je ve větě požadavek na minimální velikost $k \geq 2$.

$$\exists y : \exists s : T_k(e,x_1,\ldots,x_{k-1},y,s) \Leftrightarrow \exists w : T_k(e,x_1,\ldots,x_{k-1},(w)_{2,1},(w)_{2,2})$$

$$\Leftrightarrow \exists s : T_k(b,e,x_1,\ldots,x_{k-1},s) \Leftrightarrow \exists s : T_{k-1}(s_1(b,e),x_1,\ldots,x_{k-1},s)$$

Poznámka

Neomezená obecná kvantifikace ($\forall$) rekurzivní spočetnost nezachovává.

Věta (O selektoru)

Nechť Q je RSP $k+1$ proměnných. Potom existuje ČRF φ k proměnných taková, že:

$$\varphi(x_1,\ldots,x_k)\downarrow \Leftrightarrow \exists y : Q(x_1,\ldots,x_k,y)$$

$$\varphi(x_1,\ldots,x_k)\downarrow \Rightarrow Q(x_1,\ldots,x_k,\varphi(x_1,\ldots,x_k))$$

Věta říká, že pro každý rekurzivně spočetný predikát existuje ČRF taková, že konverguje, právě když existuje y splňující predikát. Tato funkce navíc přímo vrací jedno takové y, pro které predikát platí. Tato φ je *selektor na grafu* Q.

Důkaz

Dáno $\vec{x}$, hledáme nejmenší dvojici (y,s) takovou, že za s kroků ověříme, že $Q(\vec{x},y)$ (tj. program pro Q konverguje za s kroků). Pak vydáme y.

Obecně: univerzální vyjádření RSP $\exists s : T_{k+1}(e,\vec{x},y,s)$, hledáme nejmenší w (kód dvojice) takové, že $\varphi(\vec{x}) \simeq (\mu_w T_{k+1}(e,\vec{x},(w)_{2,1},(w)_{2,2}))_{2,1}$. Funkce φ vrací první složku z první dvojice, kterou najde (v uspořádání daném naším kódováním dvojic).

Věta (Vztah ČRF a RS grafů)

Funkce je ČRF $\Leftrightarrow$ má rekurzivně spočetný graf.

Důkaz

Je-li φ ČRF, je její graf rekurzivně spočetný: $\langle x_1,\ldots,x_k,y\rangle \in$ Graf $\Leftrightarrow \exists s :$ za s kroků program konverguje.

Opačně, je-li graf funkce φ rekurzivně spočetný, je selektor na něm ČRF, ale selektor na grafu funkce je přímo ona funkce.

Věta (Postova)

Množina M je rekurzivní, právě když M i $\overline{M}$ jsou rekurzivně spočetné.

Predikát Q je ORP, právě když Q i $\neg Q$ jsou RSP.

Důkaz

"$\Rightarrow$": Triviální.

"$\Leftarrow$": Intuitivně: $M = dom(P_1)$, $\overline{M} = dom(P_2)$. Pustíme oba programy současně a čekáme, který se zastaví. Zastaví se právě jeden.

Formálně: $(x \in M \wedge y = 1) \vee (x \in \overline{M} \wedge y = 0)$ je rekurzivně spočetný predikát, selektor na něm je ORF, která je charakteristickou funkcí pro M.

7.4 Generování rekurzivně spočetných množin

Lemma (Rek. spočetná množina je obor hodnot ČRF)

Každá rekurzivně spočetná množina je oborem hodnot nějaké ČRF.

Důkaz

Pro každou množinu W_x vytvoříme množinu dvojic $R = \{\langle y, y\rangle : y \in W_x\}$. Množina R je rekurzivně spočetná, tedy má ČRF selektor φ, platí $dom(\varphi) = rng(\varphi) = W_x$.

Myšlenka toho důkazu je, že body, kde φ_x konverguje, vyneseme na diagonálu a vytvoříme selektor. Jeho definiční obor bude zároveň jeho oborem hodnot.

Věta (ČRF odpovídá Rek. spočetným množinám)

Každý obor hodnot ČRF je rekurzivně spočetná množina.

Důkaz

Máme ČRF g a její obor hodnot. Zkonstruujeme pseudoinverzní funkci h k ČRF g, tj. funkci takovou, že $dom(h) = rng(g)$ a to tak, že vyrobíme RS predikát $Q(x, y) \Leftrightarrow g(x) \simeq y$ a to má ČRF selektor, který hledáme – h.

Definice (Úseková funkce)

Funkce f je *úseková*, jestliže jejím definičním oborem je počáteční úsek $\mathbb{N}$ (nebo celé $\mathbb{N}$).

Věta (Rek. množiny a úsekové ČRF)

Rekurzivní množiny jsou právě obory hodnot rostoucích úsekových ČRF.

Důkaz

$\Rightarrow$: Definujeme ČRF f, která bude rostoucí a úseková.

- Začneme $f(0) \simeq \mu_x(x \in M)$.
- Dále $f(n+1) \simeq \mu_y(y > f(n) \wedge y \in M)$

$\Leftarrow$: Máme f rostoucí úsekovou ČRF.

1. V případě, že je f má konečné dom (tohle ale nejsme schopni efektivně rozpoznat!), víme jak, známe $D = dom(f)$ a tedy $rng(f)$ je rekurzivní.
2. V případě, že je f je všude definovaná (totální): $y \in M = rng(f) \Leftrightarrow \exists x : (f(x) = y)) \Leftrightarrow \exists x \leq y : (f(x) = y)$ Poslední ekvivalence platí, protože f je rostoucí a úseková. Tedy $y \in M \Leftrightarrow y \in \{f(0), \ldots, f(y)\}$.

Věta (O generování)

Mějme nekonečnou množinu M. Potom:

- Množina M je rekurzivní, právě když M lze generovat rostoucí ORF.
- Množina M je rekurzivně spočetná, právě když M lze generovat prostou ORF.

Důkaz

Důsledek předchozí, resp. následující věty.

Věta (Rek. spočetné množiny a prosté úsekové ČRF)

Rekurzivně spočetné množiny jsou právě obory hodnot prostých úsekových ČRF.

Důkaz

"$\Leftarrow$": Víme, obor hodnot ČRF je rekurzivně spočetná množina (z věty o tom, že ČRF odpovídají RSM).

"$\Rightarrow$": Mějme ČRF φ ($M = rng(\varphi)$ pro nějaké φ, z lemmatu o tom, že RSM je obor hodnot ČRF).

Důkaz provedeme pomocí rekurzivní množiny $B = \{\langle x, s\rangle : \varphi(x)\downarrow$ přesně za s kroků $\}$. Je vidět, že každé x bude pouze v jednom z párů $\langle x, s\rangle$.

Množinu B lze, protože je rekurzivní, generovat pomocí rostoucí úsekové ČRF h. Funkce h generuje dvojice, definujeme tedy $g(x) \simeq (h(x))_{2,1}$. Zřejmě g je prostá, úseková a ČRF (a generuje $rng(\varphi)$).

Důsledek

Každá nekonečná rekurzivně spočetná množina obsahuje nekonečnou rekurzivní podmnožinu.

Důkaz

Mějme f, která prostě generuje M. Vyber rostoucí podposloupnost. Ta je rekurzivní.

$$g(0) = f(0)$$

$$g(n+1) = f(\mu_j(f(j) > g(n)))$$

Obor hodnot g je nekonečná rekurzivní množina a je podmnožinou M.

Kapitola 8

Státnice - Algoritmicky nerozhodnutelné problémy

8.1 Turingovy stroje

Definice (Turingův stroj)

Deterministický Turingův stroj (DTS) M s k-páskami, kde k je konstanta, je pětice

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

- Q = konečná množina stavů řídící jednotky
- Σ = konečná pásková abeceda
- $\delta : (Q \backslash F) \times \Sigma^k \mapsto Q \times \Sigma^k \times \{L, N, R\}^k$ je přechodová funkce (částečná)
- $q_0 \in Q$ = počáteční stav
- $F \subseteq Q$ = množina přijímajících stavů

Definice (Konfigurace TS/Postovo slovo)

Konfigurace (jednopáskového) *TS*, neboli *Postovo slovo*, je obsah nejmenší souvislé části pásky, která obsahuje všechna neprázdná a čtené políčko; poloha hlavy a stav. Zapisujeme:

$$UqsV$$

kde U, V jsou části pásky nalevo a napravo od hlavy, q je stav a s čtené políčko.

Poznámka (Kombinování TS)

- Spojení dvou TS: napřed počítá $M1$, na výsledek pustím $M2$, tj. $M1 \wedge M2$
- Větvení (if-then-else): ve stroji $M1$ ze stavu q_0 přechod do (poč. stavu) $M2$, z q_1 do $M3$
- While-cyklus: složené spojení a větvení

Nutná opatrnost – stejná vnější abeceda, disjunktní vnitřní stavy atd.

Poznámka (Modifikace a omezení (stručně))

- Omezení pohybu na jeden směr – síla stroje klesne na úroveň konečných automatů
- Omezení na povinný pohyb (L/P) – OK
- Jen jedna činnost v taktu (buď zápis, nebo posuv) – OK
- Jednostranně omezená páska, více pásek, více hlav – stále stejná síla
- Okraje pásky z obou stran – páska není nekonečná, mám jen konfiguraci stroje – můžu mít potřebu pásku zvětšovat a zmenšovat (je možné)
- Omezení na 2 aktivní stavy – OK (jeden ale nestačí)
- Omezení na 2 symboly abecedy – OK (z toho jedno je "blank")
- K simulaci TS stačí 2 zásobníkové automaty – z jednoho zásobníku uděláme obsah pásky nalevo, z druhého napravo (vč. čteného znaku) a přehazujeme znaky.

8.2 Univerzální Turingův stroj

Věta (Univ. Turingův stroj)

Máme dánu abecedu A. Existuje univerzální TS $\mathcal{U}$ nad A, který pro každý TS nad A simuluje výpočet.

$$\mathcal{U}(\text{kód}(T) + \text{kód}(S)) \simeq T(S)$$

Důkaz

- Vezmeme $A = \{\lambda, |\}$, což stačí. BÚNO má každý TS jediný koncový stav q_f, počáteční stav buď q_s. Počet stavů – m – může být velký. Kód stavu q_i budiž blok znaků délky $m + 2$ ($|$ + i-krát $|$ + $m - i$-krát λ + λ).
- Pro $i \geq 1$ máme vždy dvě instrukce (jedna pro λ, druhá pro $|$). Ty se dají zakódovat do bloku $\times O_1 \times O_2 \times O_3 \cdots \times O_m \times$, kde $\times$ je pomocný symbol (v abecedě $\mathcal{U}$ být může) a O_i jsou kódy obou instrukcí pro stav r_i – kód zapisovaného písmene, pohybu a cílového stavu.
- Páska $\mathcal{U}$ pak vypadá následovně:

$$Y[\text{blok1}]Y[\text{blok2}]\Delta[\text{blok3}] \times O_1 \times O_2 \cdots \times O_m \times$$

  - "blok1" je konfigurace pův. stroje, jen obsah právě čteného pole je nahrazen pomocným symbolem M.
  - "blok2" kóduje aktuální stav pův. stroje.
  - "blok3" je jedna buňka, v níž je uložen obsah právě čteného pole.
- Univerzální stroj potom sestává z testu bloku2, zda obsahuje koncový stav, procedury vyčištění pásky a vydání výsledku a odsimulování jednoho kroku práce původního stroje
- Simulace:

1. najít relevantní blok O_i – stav i si nelze pamatovat přímo, proto musím z bloku 2 postupně umazávat $|$ a posouvat nějaký spec. symbol "zarážku" doprava
2. posunout zarážku na konkrétní instrukci podle bloku3 (čteného znaku)
3. provést instrukci (po kouskách přenést nový stav do bloku 2, pak 6 možností zapisování písmene a pohybu, při pohybu a mazání pozor na okraje pásky)

Důsledek

Díky tomu lze všechny TS očíslovat.

Věta (Halting problém)

Halting problém není algoritmicky rozhodnutelný.

Důkaz

Sporem nechť máme TS $H(T, K)$ rozhodující, zda se TS T zastaví nad daty K (a H se zastaví vždy a vydá buď 0 nebo 1). Potom lze vyrobit $Alg(K)$ takový, že $Alg(K)$ se zastaví, právě když $\mathcal{U}(K + K)$ se nezastaví (pomocí H). Pak $Alg(K)$ má nějaký kód, nazveme jej Q. Pak ale

$$Alg(Q) \text{ zastavi} \Leftrightarrow \mathcal{U}(Q + Q) \text{ nezastavi} \Leftrightarrow Alg(Q) \text{ nezastavi}$$

a to je spor.

Poznámka (Silně a slabě omezené mazání)

Omezíme mazání v TS:

- **slabě** – máme spec. symbol "kaňka" ($*$) a pravidla:
  - $\lambda \to$ cokoliv
  - cokoliv $\neq \lambda \to *$
- **silně** – máme abecedu jen $\{\lambda, *\}$ a povolený jen přepis $\lambda \to *$.

Oba dva případy mají stejnou sílu jako běžný TS (silné se slabým dá simulovat: kaňku kódovat jako blok samých kaněk, převést abecedu; normální TS se dá simulovat slabým postupným překreslováním konfigurací vedle sebe na pásku se současným měněním stavu).

Lze algoritmicky rozhodnout, zda TS T s konfigurací K někdy přepíše λ na něco jiného (existuje horní odhad počtu kroků v popsané části pásky). Nelze ale rozhodnout, zda TS T s konfigurací K někdy přepíše ne-λ na λ – to je ekvivalentní Halting problému (T simulujme silně omezeným T_1 a přidejme T_2, který smaže 1 kaňku. Pokud se T_1T_2 zastaví, musel se zastavit i T_1 a tím bychom rozhodli zastavení T).

Kapitola 9

Státnice - Věty o rekurzi

9.1 Věty o rekurzi

Věta (O rekurzi, o pevném bodě, self-reference)

Jestliže f je ORF jedné proměnné, potom existuje a takové, že $\varphi_{f(a)}(x) \simeq \varphi_a(x)$ pro všechna x (kde $\varphi_a(x)$ značí a-tou funkci, tedy odpovídá $\Psi_1(a,x)$).

Důkaz

Zjevně platí následující – první výraz je ČRF, má tedy své číslo e, druhá rovnost plyne ze S-m-n věty:

$$\lambda z,x(\varphi_{f(s_1(z,z))}(x)) \simeq \Psi_2(e,z,x) \simeq \varphi_{s_1(e,z)}(x)$$

Dosadíme $z = e$ a dostáváme hledané $a = s_1(e,e)$. Platí totiž $\varphi_{f(s_1(e,e))}(x) \simeq \Psi_2(e,e,x) \simeq \varphi_{s_1(e,e)}(x)$.

Vlastnosti programů a a $f(a)$

Funkce f zobrazuje program na program. Bod a je pevným bodem zobrazení f. Jak vypadají programy a a $f(a)$? Který z nich počítá déle? Uvidíme, že program a počítá déle než $f(a)$.

Co dělá program e na datech (z,x)? Počítá $\varphi_{f(s_1(z,z))}$, tj. vezme z a spočítá neprve $s_1(z,z)$, potom $f(s_1(z,z))$, který ale nemusí konvergovat. Jestliže $f(s_1(z,z))\downarrow$, spustí se na vstup x.

Co dělá program a? Program a vznikne jako $s_1(e,e)$. Mějme na vstupu x. Program a vezme e a přidá ho k x a spustí program e na (e,x). Co udělá program e na těchto datech? Spočítá $s_1(e,e)$ (tedy spočítá a), potom $f(s_1(e,e)) = f(a)$ a spustí program $f(a)$ na x.

Program a tedy neprve spočítá a, potom spočítá $f(a)$ (pokud konverguje) a ten simuluje na vstupu x. Program a je tedy složitější než $f(a)$ a počítá déle.

Poznámka z λkalkulu

V λkalkule sa ekvivalentné tvrdenie ukazuje trošku jednoduchšie. Pre každý λterm F (program F) existuje λterm X taký, že $X = FX$ (program F aplikovaný na X sa rovná X).

Dôkaz je nasledovný

- Majme F, pre ktoré chceme nájsť jeho pevný bod X.
- Nech $W = \lambda x.F(xx)$ (to je funkcia, ktorá x priradí $F(xx)$).
- $X = WW$ (to môžeme chápať ako program/funkciu W aplikovaný na W)
- $X = WW = (\lambda x.F(xx))W = F(WW) = F(X)$ (tretia rovnosť je βpravidlo λkalkulu. Ak si ale $(\lambda x.F(xx))W$ predstavíme ako funkciu, ktorá x priradí $F(xx)$ aplikovanú na W, rovnosť je (snáď) jasnejšia).

Věta (O generování pevných bodů)

Pro každou ORF f existuje prostá rostoucí PRF g taková, že platí:

$$\varphi_{f(g(j))}(x) \simeq \varphi_{g(j)}(x)$$

Tedy g rostoucím způsobem generuje nekonečně mnoho pevných bodů funkce f.

Důkaz

Postupujeme stejně jako v důkazu předchozí věty, jen máme o proměnnou (parametr j funkce g) navíc, tj. platí $\varphi_{f(s_2(z,z,j))}(x) \simeq \Psi_3(e,z,j,x) \simeq \varphi_{s_2(e,z,j)}(x)$. Zvolme $g(j) = s_2(e,e,j)$.

Věta (O rekurzi pro více proměnných)

Nechť f je ČRF $n+1$ proměnných. Potom existuje číslo a takové, že platí $\varphi_a(x_1,\ldots,x_n) \simeq f(a,x_1,\ldots,x_n)$ (tj. a je indexem funkce $\lambda x_1,\ldots,x_n f(a,x_1,\ldots,x_n)$).

Důkaz

$$f(y,x_1,\ldots,x_n) \simeq \Psi_{n+1}(e,y,x_1,\ldots,x_n) \simeq \varphi_{s_1(e,y)}(x_1,\ldots,x_n)$$

Následně aplikujeme větu o rekurzi na $s_1(e,y)$ v proměnné y a dostáváme hledané a (podle VR platí: $\exists a : \varphi_{s_1(e,a)} \simeq \varphi_a$).

Věta (O rekurzi v závislosti na parametrech)

Jestliže f je ČRF $n+1$ proměnných, potom existuje PRF g o n proměnných taková, že platí:

$$\varphi_{f(g(y_1,\ldots,y_n),y_1,\ldots,y_n)}(x) \simeq \varphi_{g(y_1,\ldots,y_n)}(x)$$

Důkaz

Pro $n = 0$ je to totéž jako verze bez parametrů. g nachází pevné body v závislosti na parametrech. Podobně jako v předchozích větách platí: $\varphi_{f(s_{n+1}(z,z,y_1,\ldots,y_n),y_1,\ldots,y_n)}(x) \simeq \Psi_{n+2}(e,z,y_1,\ldots,y_n,x) \simeq \varphi_{s_{n+1}(e,z,y_1,\ldots,y_n)}(x)$. Zvolme $g(y_1,\ldots,y_n) = s_{n+1}(e,e,y_1,\ldots,y_n)$.

9.2 Riceova věta

Věta (Rice)

Jestliže $\mathcal{A}$ je třída ČRF (jedné proměnné), která je netriviální (nejsou to všechny funkce a není prázdná), potom indexová množina $A_{\mathcal{A}} = \{x : \varphi_x \in \mathcal{A}\}$ (indexy programů, které vyčíslují funkce z $\mathcal{A}$) je nerekurzivní.

Důkaz

Sporem. Nechť $A_{\mathcal{A}}$ je rekurzivní. Potom lze vytvořit ORF f takovou, že všechny prvky z $A_{\mathcal{A}}$ zobrazí na nějaký prvek $b \notin A_{\mathcal{A}}$ a všechny prvky mimo $A_{\mathcal{A}}$ zobrazí na nějaký prvek $a \in A_{\mathcal{A}}$. Podle věty o rekurzi existuje pevný bod $f - u_0$, tedy platí:

$$\varphi_{u_0} = \varphi_{f(u_0)}$$

Takže:

$$u_0 \in A_{\mathcal{A}} \Rightarrow f(u_0) = b \notin A_{\mathcal{A}}$$
$$u_0 \notin A_{\mathcal{A}} \Rightarrow f(u_0) = a \in A_{\mathcal{A}}$$

To je ovšem spor, protože u_0 a $f(u_0)$ jsou indexy stejné funkce, a tedy buď obě čísla v $A_{\mathcal{A}}$ leží, nebo obě neleží.

Důsledky

Pozor, nejedná se o třídu programů, ale třídu funkcí. Tedy i pro jednoprvkovou $\mathcal{A}$ bude $A_{\mathcal{A}}$ nekonečná a nerekurzivní (každá funkce je vyčíslovaná nekonečně mnoha programy a rozhodnout o jejich ekvivalenci nelze efektivně).

Proto platí:

- Nechť $\mathcal{A} = \{\varphi_e\}$, potom $A_{\mathcal{A}} = \{x : \varphi_x = \varphi_e\}$ je nerekurzivní.
- Rozhodnout o rovnosti funkcí vyčíslovaných dvěma programy nelze algoritmicky.

Kapitola 10

Státnice - Stromové vyhledávací struktury

10.1 Binární vyhledávací stromy

Definice

Binární vyhledávací strom T reprezentující množinu prvků S z (uspořádaného) univerza U je úplný strom (tj. všechny vnitřní vrcholy mají 2 syny), ve kterém existuje bijekce mezi množinou S a *vnitřními* vrcholy taková, že pro v vnitřní vrchol stromu platí:

- všechny vrcholy podstromů levého syna jsou $\leq v$
- všechny vrcholy podstromů pravého syna jsou $> v$.

Listy reprezentují jednotlivé intervaly mezi vnitřními vrcholy. Můžeme je vynechat, ale s nimi je to (jak pro koho) logičtější.

Základní operace na stromech

- **MEMBER** – test, zda prvek x je obsažen ve stromě (vyhledání, zpravidla s využitím invariantu)
- **INSERT** – vložení prvku x do stromu
- **DELETE** – odebrání prvku x ze stromu
- **MIN**, **MAX**, **ORD** – nalezení prvního, posledního, k-tého největšího prvku
- **SPLIT** – rozdělení stromu podle x, které vyhodí, je-li ve stromě
- **JOIN** – spojení dvou stromů (jsou dvě verze, s přidáním prvku navíc, nebo bez něho)

Obecná (nevyvážená) implementace

- **INSERT**: najít list reprezentující interval, kam vkládám, udělat z něj normální uzel se vkládanou hodnotou a dát mu dva listy s podintervaly.
- **DELETE**: najdu vrchol, má-li jednoho syna-lista, pak druhý syn ho nahradí na jeho místě, jinak najdeme a dáme na jeho místo nejmenší větší vnitřní vrchol, jehož levý syn je list, a pravého syna tohoto vrcholu dáme na jeho místo.
- **SPLIT**: procházím stromem a hledám x, ost. prvky házím cestou do dvou stromů T_1, T_2, ve kterých si vždy uchovávám ukazatel na list, místo kterého vkládám (odkrojím syna, ve kterém hledám dál, místo něj vložím list, na který si pamatuju ukazatel).
- **JOIN**: s prvkem navíc, triviální – spojím stromy jako 2 syny nového prvku.

Tato struktura sama nepodporuje efektivní **ORD**, je nutné přidat navíc položky, které určují počet listů v podstromu každého vrcholu. ORD je pak jen jde do pravých synů a přičítá levé podstromy když může, jinak jde do levého syna (a nepřičte nic).

Analýza algoritmů

Definujme si pomocné hodnoty λ, π jako hodnoty nejbližšího menšího (levějšího), resp. většího (pravějšího) prvku na vyšší úrovni, nebo $-\infty$, resp. $+\infty$, pokud tyto prvky neexistují.

Korektnost vyhledávání: Je-li T' podstrom $t \in T$, pak T' reprezentuje $S \cap (\lambda(t), \pi(t))$ (a je to největší interval nezastoupený mimo T'). Pak pro vyhledání vrcholu x platí $\lambda(t) < x < \pi(t)$, vyšetřuji-li vrchol t.

Díky tomu je korektní **MEMBER** a **INSERT**. U **DELETE** musím dokázat korektnost případu s přehazováním vrcholů (dostávám bin. strom reprezentující $S \setminus \{x\}$).

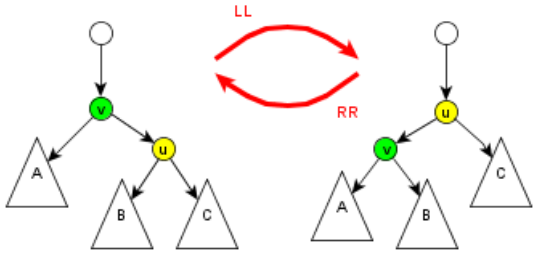
Korektnost **MIN**, **MAX**, **JOIN** je zřejmá, u **SPLIT** plyne z korektnosti hledání a toho, že moje označené listy jsou nejlevější, resp. nejpravější.

Korektnost **ORD** plyne z toho, že v každém kroku je k-tý prvek představován tolikátým v pořadí vrcholů akt. vyšetřovaného podstromu, kolik mi zbývá přičíst.

Složitost: Zpracování 1 vrcholu je vždy $O(1)$ a alg. se pohybuje po nějaké cestě od kořene k listu, která má $O(h)$, kde h je výška stromu.

Vyvažování

Chceme-li pro zachování efektivity operací zajistit, že výška bude $O(\log |S|)$, přidáme pro strom další podmínky, které bude muset splňovat a operace je zachovávat.

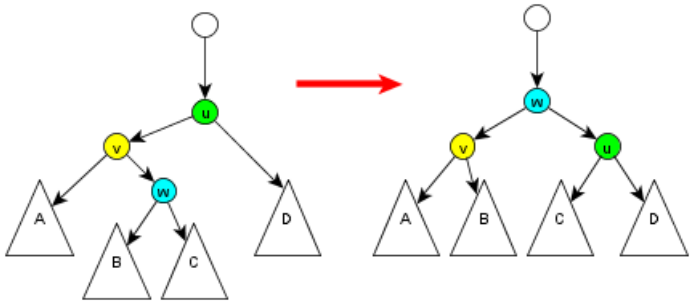


Obrázek 10.1: Rotace

Pro vyvažovací operace, které se snaží zachovat logaritmickou výšku, se používá pomocný algoritmus **ROTACE**(u, v):

1. Vezmu v, jeho pravého syna u a podstromy (zleva) A, B, C.
2. Přehodím v pod u, upravím ukazatel v otci a přeházím podstromy.

Existuje i symetrický případ, kdy se postupuje přesně opačným směrem. Někdy se této dvojici operací říká **LL-ROTACE** a **RR-ROTACE**.



Obrázek 10.2: Dvojrotace

Další potřebný algoritmus je **DVOJROTACE**(u, v, w):

1. Vezmu u, jeho levého syna v a pravého syna v – w
2. Seřadím je tak, že w je otec obou, u vpravo a v vlevo.
3. Přitom opět upravím ukazatele v nadřízeném uzlu a přepojím podstromy.

Taky existuje symetrický případ. Jiné označení je **LR-ROTACE** a **RL-ROTACE**.
U obou operací lze aktualizovat i počty listů v podstromě a obě pracují v $O(1)$.

Alternativy k vyvažování

Je velká pravděpodobnost, že i bez vyvažování strom zůstane $O(\log |S|)$ vysoký a operace na něm můžou tak (bez vyvažování) běhat i rychleji. Proto existují i *pravděpodobnostní postupy*, nahrazující vyvažování znáhodněním posloupností operací. Další možnost jsou *samoopravující struktury* – operace samy bez dalších uchovávaných dat obstarávají vyvažování, existuje strategie, která zajistí dobré chování bez ohledu na data. Nebo se *sleduje chování* struktury, a když začne být příliš pomalá, vytvoří se nová – vyvážená. Poslední možnost je *upravit dat. strukturu podle známého pravděpodobnostního rozdělení* dat.

AVL-stromy

AVL-stromy (Adel'son, Velskii, Landis) jsou nejstarší vyvážené stromy, dodnes oblíbené, jednoduše definované, ale detailně technicky složité.

Podmínka AVL pro vyvažování: *Výška pravého a levého podstromu lib. vrcholu se liší max. o 1.*

Definice: $\eta(v)$ – výška vrcholu (délka nejdelší cesty z vrcholu do listů), $\omega(v)$ – rozdíl výšek levého a pravého podstromu ($\in \{-1, 0, 1\}$). Uchovávat potřebuji jenom ω.

Logaritmická výška

Výška celého stromu (η(kořen)) vychází z toho, že podstrom AVL stromu je vždy AVL strom. Vezmeme rekurzivní vztahy pro největší a nejmenší množinu uzlů v AVL stromu výšky i:

Nejmenší: $mn(i) = mn(i-1) + mn(i-2) + 1$

Největší: $mx(i) = 2mx(i-1) + 1$

Indukcí dokážeme, že $mx(i) = 2^i - 1$ a $mn(i) = F_{i+2} - 1$, kde F_i je i-té Fibonacciho číslo (pro ty platí vzorec $F_{i+2} = F_{i+1} + F_i$). Víme, že $\lim_{i\to\infty} F_i = \sqrt{5}(\frac{1+\sqrt{5}}{2})^{-i}$ a z toho zlogaritmováním plyne pro AVL-strom o výšce i s n prvky:

$$\log\left(\frac{c_1}{\sqrt{5}}\right) + (i+2)\log\left(\frac{1+\sqrt{5}}{2}\right) < \log(n+1) < i$$

A tedy $0.69i < \log(n+1) < i$, takže $i = \Theta(\log n)$.

Operace na AVL stromech

Operace **MEMBER** je stejná jako pro nevyvážené.

INSERT se musí po běžném vložení zabývat vyvažováním. Jde zpět ke kořeni a hledá, který nejnižší vrchol x nemá po vložení vyvážené ω, přičemž cestou upravuje ω. Na vrcholu x se provede vhodná ROTACE nebo DVOJROTACE, což zajistí vyváženost (existuje několik podpřípadů).

Operace **DELETE** odstraní vrchol a pak vyvažuje podobně jako INSERT, ale potřebuje víc operací (až $O(\log |S|)$ rotací). Asymptotická složitost je ale stejná – logaritmická.

Červeno-černé stromy

Červeno-černý strom má tyto čtyři povinné vlastnosti:

1. Každý uzel má definovanou barvu, a to černou nebo červenou.
2. Každý list je černý.
3. Každý červený vrchol musí mít oba syny černé.
4. Každá cesta od libovolného vrcholu k listům v jeho podstromě musí obsahovat stejný počet černých uzlů. Pro červeno-černé stromy se definuje *černá výška uzlu* ($\mathbf{bh}(x)$) jako počet černých uzlů na nejdelší cestě od uzlu k listu.

Garantování výšky

Podstrom libovolného uzlu x obsahuje alespoň $2^{\mathbf{bh}(x)} - 1$ interních uzlů. Díky tomu má červeno-černý strom výšku vždy nejvýše $2\log(n+1)$ (kde n je počet uzlů). (Důkaz prvního tvrzení indukcí podle $\mathbf{h}(x)$, druhého z prvního a třetí vlastnosti červeno-černých stromů)

Algoritmy

U algoritmů **INSERT** a **DELETE** jde také o vložení a následné vyvažování. Bez porušení vlastností červeno-černých stromů lze kořen vždy přebarvit načerno, můžeme pro ně předpokládat, že *kořen stromu* je *vždy černý*.

INSERT vypadá následovně:

- Vložený prvek se přebarví načerveno.
- Pokud je jeho otec černý, můžeme skončit – vlastnosti stromů jsou splněné. Pokud je červený, musíme strom upravovat (předpokládejme, že otec přidávaného uzlu je levým synem, opačný připad je symetrický):
  - Je-li i strýc červený, přebarvit otce a strýce načerno a přenést chybu o patro výš (je-li děd černý, končím, jinak můžu pokračovat až do kořene, který už lze přebarvovat beztrestně).
  - Je-li strýc černý a přidaný uzel je levým synem, udělat pravou rotaci na dědovi a přebarvit uzly tak, aby odpovídaly vlastnostem stromů.
  - Je-li strýc černý a přidaný uzel je pravým synem, udělat levou rotaci na otci a převést tak na předchozí případ.

DELETE se provádí takto:

- Skutečně odstraněný uzel (z přepojování – viz obecné vyhledávácí stromy) má max. jednoho syna. Pokud odstraňovaný uzel byl červený, neporuším vlastnosti stromů, stejně tak pokud jeho syn byl červený – to řeším přebarvením toho syna načerno.
- V opačném případě (tj. syn odebíraného – x – je černý) musím udělat násl. úpravy (předp., že x je levým synem svého nového otce, v opačném případě postupuji symetricky):
  - x prohlásím za "dvojitě černý" ("porucha") a této vlastnosti se pokouším zbavit.
  - Pokud je (nový) bratr x (buď w) červený, pak má 2 černé syny – provedu levou rotaci na rodiči x, prohodím barvy rodiče x a uzlu w a převedu tak situaci na jeden z násl. případů:
    - Je-li w černý a má-li 2 černé syny, prohlásím x za černý a přebarvím w načerveno, rodiče přebarvím buď na černo (a končím) nebo na "dvojitě černou" a propaguji chybu (mohu dojít až do kořene, který lze přebarovat beztrestně).
    - Je-li w černý, jeho levý syn červený a pravý černý, vyměním barvy w s jeho levým synem a na w použiji pravou rotaci, čímž dostanu poslední případ:
    - Je-li w černý a jeho pravý syn červený, přebarvím pravého syna načerno, odstraním dvojitě černou z x, provedu levou rotaci na w a pokud měl původně w (a x) červeného otce, přebarvím w načerveno a tohoto (teď už levého syna w) přebarvím načerno.

MIN a **MAX** jsou stejné jako pro nevyvážené.

JOIN (s prvkem navíc): mám-li černou výšku u obou stejnou, není co řešit, pokud ne, projdu po tom s větší $\mathbf{bh}(x)$ do patra, kde se výšky rovnají, půjčím si přísl. podstrom a slepím s ním, vrátím celek zpátky a aplikuji vyvažování, jako kdybych vložil 1 prvek (poruším výšku max. o 1).

SPLIT: rozhazuji podstromy do zásobníků, odkud je pak slepuji operací **JOIN**.

Každý algoritmus pracuje jen s vrcholy na jedné cestě od kořene k listům a s každým dělá konstantně činností, takže všechny algoritmy mají logaritmickou složitost. DELETE volá max. 2 rotace nebo 1 rotaci a 1 dvojrotaci, INSERT zase max. 1 rotaci nebo dvojrotaci (i když přebarvovat můžou rekurzivně až do kořene).

Váhově vyvážené stromy (BB-α)

Dnes jsou už na ústupu, ale občas se ještě používají. Mějme $1/4 < \alpha < 1 - \sqrt{2}/2$, označme $p(T)$ počet listů ve stromě T. Pak strom je BB-α, když

$$\alpha \leq \frac{p(T_{\text{levý}(v)})}{p(T_v)} \leq 1 - \alpha$$

pro T_v jako podstrom určený (každým) vrcholem v. O BB-α stromech platí, že:

$$\text{výška}(T) \leq 1 + \frac{\log(n+1)-1}{\log \frac{1}{1-\alpha}}$$

Takže jsou také vyvážené a operace mají zaručenou logaritmickou hloubku, vyvažuje se na nich také rotacemi a dvojrotacemi. Vždy totiž existuje $\alpha \leq d \leq 1 - \alpha$ takové, že když mám strom, jehož oba podstromy splňují vlastnosti a navíc $p(T_l)/p(T) \leq \alpha$ a $p(T_l)/p(T) - 1$ nebo $p(T_l) + 1/p(T) + 1$ vyhovuje, vezmu $\rho = \frac{p(T')}{p(T_r)}$, kde T' je určen levým synem T_r, a pro $\rho \leq d$ provedu ROTACE(T, T_r), jinak DVOJROTACE(T, T_r, T') a dostanu BB-α strom (bez důkazu). Opačný případ je popsaný symetricky.

Mají pěknou vlastnost, kvůli které se používaly: pro $\forall\alpha$ existuje $c > 0$ takové, že každá posloupnost k operací INSERT a DELETE volá max. $c \cdot k$ rotací a dvojrotací.

10.2 B-Stromy a jejich varianty

(a,b)-stromy

(a, b)-strom pro $a \leq b$ přirozená je strom T , který splňuje následující podmínky:

- každý vnitřní vrchol v stromu T různý od kořene t má alespoň a a nejvíc b synů
- všechny cesty od kořene k listům mají stejnou délku

Tato definice je ale pro praktické účely příliš obecná – budeme chtít navíc podmínky:

- $a \geq 2$ a $b \geq 2a - 1$
- kořen je buď list nebo má alespoň 2 a nejvíc b synů

Takový (a, b)-strom existuje pro každý přirozený počet listů, jeho výška je mezi $\log_b n$ a $1 + \log_a(\frac{n}{2})$, tedy $O(\log n)$. Indukcí: strom o výšce h má $2a^{h-1} \leq n \leq b^h$ listů (přidáním h-té hladiny do stromu s k listy dostaneme strom s $ka \leq n \leq kb$ listy.

Reprezentace množiny

Strom *reprezentuje* nějakou množinu S (prvků z univerza U), když mám bijekci mezi uspořádáním S a lexikografickým uspořádáním listů.

Každý vnitřní vrchol v obsahuje informaci o počtu synů $\rho(v)$, pole ukazatelů na syny S_v a pole H_v prvků z U takových, že i-tý je největší v S reprezentovaný v podstromě i-tého syna. Listy mají jen svůj prvek.

Pro každý prvek kromě největšího existuje vnitřní vrchol, který obsahuje jeho klíč, proto lze listy i vynechat a ukládat data ve vnitřních vrcholech (což ale není moc přehledné). Vrcholy můžou mít i odkaz na otce, nebo si otce můžu pamatovat při průchodech dolů (na vrcholech, ke kterým jsem nedošel od kořene, otce nepotřebuju).

Algoritmy

Máme pomocnou funkci **VYHLEDEJ**, který do hloubky projde stromem a vrátí nejbližší větší k nějakému prvku (nebo prvek sám, je-li ve stromě). Základní operace:

- **MEMBER**: přímo použije onu pomocnou operaci a pak zjistí, jestli našel, co hledal
- **INSERT**: vyhledám místo kam, pokud tam prvek není, vytvořím nový list, připojím na správné místo do S_v a postupně nahoru štěpím, je-li potřeba, extrémně rozštěpím kořen.
- **DELETE**: najdu prvek, najdu, kam je pověšený a to jedno políčko v S_v a H_v zruším, opravím ρ, pokud dostanu méně než a synů v uzlu, spojím s bezprostředním bratrem (má-li ten právě a synů), nebo přesunu nějaký list z bratra do mého uzlu.

Oba algoritmy pracují v $O(\log |S|)$ v nejhorším případě.

JOIN (bez prvku navíc): Přepokládá $\max S_1 < \min S_2$. Je-li $h(T_1) \geq h(T_2)$, najde v T_1 hladinu o 1 nad připojení a v ní největší prvek / vytvoří nadkořen T_1 v případě rovnosti, slije do něj prvky obou kořenů a případně provede štěpení. Jinak hledá a připojuje v T_2. Potřebuje čas $O(|h(T_1) - h(T_2)|)$.

SPLIT: Prochází postupně dolů, rozděluje uzly (podstromy s prvky $< x$, resp. $\geq x$) a hází výsledky do 2 zásobníků. Pokud oddělí více než 1 krajní prvek, hodí na zásobník strom, jehož kořen je právě oddělená část uzlu, jinak na zásobník dává podstrom onoho krajního prvku. Tak pokračuje až k listům, pokud tam najde přímo x, tak ho vyhodí. Stromy ze zásobníků spojí postupným voláním **JOIN** – v 1 zásobníku jsou max. 2 stromy stejné výšky, celkem jich je $k \leq 2 \log_a |S|$ a jejich zpracování trvá $O(\sum_{i=1}^{k-1}(h(T_i) - h(T_{i+1}) + 1)) = O(h(T_1) + k)$.

ORD: S takovouto reprezentací efektivně nejde, proto musíme navíc $\forall$ uzel udržovat pole P_v s počty vrcholů v jeho jednotlivých podstromech a při vkládání a odebírání ho průběžně aktualizovat. Pak v ORD procházím do hloubky a postupně přičítám velikosti přeskočených podstromů (pokud bych se přičtením dalšího dostal k 0, jdu na nižší hladinu).

Implementace

- Pro vnitřní paměť se doporučuje $a = 2 - 3$ a $b = 2a$, pro vnější $a \approx 100$ a $b = 2a$ (tj. v obou případech vlastně B-stromy).
- Při přístupu více uživatelů ke struktuře je problém s aktualizačními operacemi – zamykání celého stromu není efektivní: použije se *vyvažování shora dolů*: algoritmus INSERT zamkne uzel, jeho otce a syny. Pak pokud je počet synů $= b$, rozštěpí ho (předem), nebude se pak už štěpit zpátky.
  - Aby tohle fungovalo (abych měl $\geq a$ synů všude), je nutné $b \geq 2a$.
  - Podobně funguje DELETE – najde-li uzel s a syny, provede "preventivně" slití nebo přesun.
  - Provádí se tak víc slití a štěpení než v původní variantě, ale asymptoticky je to furt stejné.
  - Pro takovéto struktury na externí paměti se doporučuje $a \approx 100$ a $b = 2a + 2$.

B-Stromy

B-strom řádu m je vlastně (a, b) strom pro $a = \frac{m}{2}$ a $b = m$. V určitých implementacích se ovšem data nacházejí už ve vnitřních vrcholech, potom má každý uzel vždy o 1 méně datových záznamů než potomků. Pokud jsou data uložena až v listech, jedná se o **Redundantní B-strom**.

Implementační detaily:

- Někdy jdou z uzlů na data jen pointery, listy můžou mít jinou (jednodušší) dat. strukturu než vnitřní uzly.
- Pro implementaci je vhodné pamatovat si celou aktuálně procházenou větev v nějakém bufferu.
- V redundantních stromech nemusím při odstranění dat odstraňovat klíč ve vnitrnich uzlech (lze podle toho hledat i kdyz to tam neni).
- Vylepšení – **vyvažování stránek**– při přetečení stránky se nejdřív dívám, jestli není volno v sousedních. Pokud ano, přerozdělím a upravím klíče – zaručuje lepší zaplnění, ale je pomalejší. Podobně je možné vyvažovat počty se sousedy v případě vyhazování (i když nemerguju).

Další varianty

B* stromy – Na základě vyvažování stránek zpřísníme podmínky na počet uzlů: Kořen ma min. 2 potomky, ost. uzly minimálně $\lceil (2m-1)/3 \rceil$ potomků, všechny větve jsou stejné dlouhé. Štěpení se odkládá, dokud nejsou sourozenci plní, potom se štěpí buď 2 do 3 (jen s jedním sourozencem), nebo 3 do 4 (s oběma) uzlů. Při odebírání se slévají 3 uzly do 2 (nebo 4 do 3). Štěpení a slévání jde zesložitit ještě na víc stránek.

Odložené štěpení – používá stránku přetečení, vkládá znova, až když se naplní. Stránka přetečení může být jedna pro jeden listový uzel, nebo ji může sdílet nějaká skupina listů – štěpí se, až když jsou všechny listy i přetečení zaplněné. Tedy pokud má strom víc než 1 úroveň, má všechny listy zaplněné (za předpokladu nepoužití **DELETE**). S odebíráním musím i slévat a štěpit skupiny – jejich velikost není pevná.

Prefixové stromy – pro redundantní B-stromy; klíče jsou co nejkratší řetězce nutné k odlišení listů, nikoliv celé hodnoty, které se nacházejí až v listech. Při vkládání a štěpení stránek se nějakou heuristikou hledá nejkratší prefix, který by dvě vznikající stránky oddělil. Mazaní a slévání – žádná změna. Další zkrácení – u potomků se neopakuje předpona klíče, kterou ma rodič – to ale hodně zvýší nároky na CPU.

B+ stromy – pro intervalové dotazy: zrychlení tím, že zřetězíme vždy uzly v jedné hladině (a nebo jenom listy), tj. přidáme do uzlů ukazatele na levého a pravého souseda.

Hladinově propojené (a,b)-stromy s prstem (Finger trees) – pro vyhledávání navíc ještě přidáme odkaz na otce do každého uzlu a pro celou strukturu jeden "prst" – odkaz na nějaký list. Vyhledávání začíná od prstu a postupuje nahoru, dokud nenajde podstrom, v němž by měl být hledaný prvek; potom se spustí dolů. Pokud je prvek poblíž prstu, je to rychlejší než klasická varianta. Typicky máme funkci nastevní prstu na nějaký prvek a pokud se motáme v jeho okolí, vyjde to lépe.

Proměnná délka zaznamů – modifikace pro záznamy různé délky: neštěpit podle počtu záznamů, ale na zhruba 1/2 podle velikosti. Podmínka existence uzlu: součet délek záznamů v něm je $\geq B/2$ kde B je délka uzlu(stránky) (pro B* stromy $2B/3$). Problémy: dlouhé klíče mají tendenci propadávat ke kořeni, tím se zmenšuje arita stromu; může se 1 stránka štěpit i na 3 (pokud vkládám záznam delší nez 1/2 stránky); vložením záznamu může dojít ke zmenšení stromu (jak, to se ve skriptech nepíše :( ) Nejde vyrobit nezávislé **INSERT** a **DELETE**, řešení: univerzální alg. nahrazování řetězce řetězcem, **INSERT** a **DELETE** jsou jeho spec. příp. Řešeni snižování arity stromu: minimalizace délky klíčů (nalezeni klíče min. délky, která navíc splňuje min. naplněni) — pro B* stromy docela složité.

Amortizované odhady počtů štěpení a slití a vyvážení

Obecně může **INSERT** volat až $\log(|S|)$-krát štěpení a **DELETE** $\log(|S|)$-krát slití a jedno vyvážení (přesun). Začínáme-li s prázdným stromem a měříme na nějaké posloupnosti n operací, zjistíme, že jde amortizovaně o $O(1)$.

Důkaz pro $(2, 4)$-stromy:

- Použijeme bankovní metodu, kdy **INSERT** bude stát dvě jednotky a **DELETE** jednu.
- Za štěpení a slití pak budeme platit vždy jednu jednotku, vyvážení nebude stát nic, protože je v každém **DELETE** voláno max. jednou a asymptoticky nic nezkazí.
- V jednotlivých uzlech stromu budeme udržovat následující počty jednotek podle stupně uzlů (přidáváme i stupně 1 a 5, které mají uzly těsně před štěpením a sléváním):

$\rho(v)$	1	2	3	4	5
jednotek	2	1	0	2	4

- Potom **INSERT** a **DELETE** bez štěpení díky své ceně udržují (občas i přeplácí) správné stavy jednotek v uzlech – snížení stupně stojí max. 1 a zvýšení max. 2 jednotky.

- Dojde-li ke štěpení uzlu stupně 5 do uzlů stupňů 3 a 2, čtyři jednotky se zaplatí: jedna za rozdělení, jedna na účet nového uzlu stupně 2 a (max.) dvě za zvýšení stupně rodiče.
- Dojde-li k vyvažování (přesunu), máme 2 jednotky, což nám stačí k vytvoření dvou uzlů stupně 2 ze stupňů 1 a 3 a přebývá nám při vyvážení uzlů stupňů 1 a 4.
- Sléváme-li uzly stupně 1 a 2, máme 3 jednotky celkem: jednou zaplatíme vlastní slití, jednou snížení stupně rodiče a jedna zbyde.

Protože všechny možnosti zachovávají invariant, je vidět, že celkem bude max. $2n$ operací slití a štěpení. Důkaz (prý) jde rozšířit i na libovolné a a $b \geq 2a$ (podle mě by mělo stačit zachovat ceny za operace a počty jednotek v krajních případech).

Pro $b = 2a - 1$ lze bohužel jednoduše nalézt takové posloupnosti operací, kde počet slití a štěpení je $O(n \log n)$, to samé, máme-li paralelní operace při $b = 2a + 1$. Proto se doporučuje $2a$, resp. $2a + 2$.

V hladinově propojeném stromě platí, že posloupnost n operací MEMBER, INSERT, DELETE a PRST vyžaduje $O(\log n +$ čas na vyhledání prvků).

10.3 Trie

Trie je vlastně stromová reprezentace slovníku. Její označení zřejmě pochází od slova "retrieval". Jejím úkolem je reprezentovat množinu $S \subseteq U$, kde U je tvořeno všemy slovy nad abecedou $\Sigma, |\Sigma| = k$ o délce l. Na této množině budeme provádět operace **MEMBER**, **INSERT** a **DELETE**.

Požadavek na délku se použije pouze u odhadů na složitost algoritmů. Ve skutečnosti ale není omezující – slova můžeme vždycky doplnit nějakými znaky mezery nebo lehce algoritmy upravit, aby s kratšími slovy počítaly.

Základní varianta

Definice

Trie je strom takový, že každý vnitřní vrchol má k synů, odpovídajících všem znakům abecedy. Každému vrcholu lze rekurzivně přiřadit slovo nad abecedou Σ následujícím způsobem:

- Kořeni patří prázdné slovo λ.
- a-tému synu patří slovo otce doplněné o a (kde a je libovolné písmeno).

Pro každý vnitřní vrchol trie musí platit, že tento vrchol je prefixem nějakého slova z reprezentované množiny S. Každý list obsahuje jeden bit, který udává přítomnost nebo nepřítomnost slova, které představuje, v množině S.

Je vidět, že taková struktura je dost paměťově náročná – každý vrchol potřebuje paměť $O(k)$ a celkem máme aspoň tolik vrcholů, co bodů množiny, násobeno délkou cesty k nim, tedy $O(kl|S|)$.

Operace

MEMBER je velice jednoduchý – postupně sestupuje stromem podél syna, který odpovídá i-tému písmenu hledaného slova v i-tém kroku. Pokud se dostane do listu dřív než dojde na konec slova, skončí neúspěchem. Jinak vrátí informaci o přítomnosti slova z listu, do kterého se dostal.

INSERT dojde do listu podobně jako **MEMBER**. Potom (je-li to potřeba) mění listy na vnitřní vrcholy a vkládá pokračování cesty až do dosažení délky slova. V posledním kroku upraví indikaci v listu.

DELETE vyhledá prvek a nastaví indikaci v jeho listu na FALSE. Pak se postupně vrací a dokud nalézá jen samé listy s FALSE, zruší celý vrchol a změní ho na list s FALSE.

Algoritmus **MEMBER** projde až l vrcholů a každý v konstantním čase (vrcholy se indexují přímo písmeny abecedy), tedy je $O(l)$. Algoritmy **INSERT** a **DELETE** vyžadují čas $O(kl)$, protože úprava jednoho vrcholu vyžaduje až k operací.

Komprimované Trie

Trie upravíme tak, že vyházíme vrcholy, jejichž slovo je prefixem stejné množiny slov jako slovo jejich otců (tj. vrcholy, kde nedochází k žádnému větvení a je jen jedna možnost pokračování). Ve vrcholech si teď ale místo toho musíme udržovat informaci o aktuální hloubce $\kappa(v)$ a navíc v listech musíme držet celé slovo, které reprezentují (abychom neztratili písmena z vrcholů, které jsme vyházeli).

Operace pak bude třeba upravit, ale protože takto upravené trie má jen $S-1$ vnitřních vrcholů, paměťová náročnost klesla na $O(k|S|)$

Operace

MEMBER pracuje podobně, ale ve vrcholu v vždy testuje $\kappa(v)+1$-ní písmeno hledaného slova. Nakonec (protože neotestoval všechna písmena a mohlo by tedy dojít ke kolizi) navíc porovná slovo uložené ve vrcholu se slovem, které jsme hledali.

INSERT pro nějaké slovo x opět vyhledá místo pro vložení a dojde do listu, kde najde jiné slovo y. Pak vezme největší společný prefix slov x, y a v jeho místě strom rozdělí – pokud ve správné hloubce už je vnitřní vrchol, pokračuje z něj, jinak nový dělící vrchol přidá. Potom upraví hodnoty v listech.

DELETE zruší informaci o mazaném slově stejně jako předtím. Navíc ale pokud zjistí, že otec "vyčištěného" listu v hierarchii stromu má jen jednoho dalšího potomka – vnitřní uzel, nacpe tohoto potomka na jeho místo.

Je vidět, že **INSERT** a **DELETE** změní maximálně jeden vnitřní vrchol a pracují tak v $O(k+l)$.

Očekávaná hloubka

Odhady časů pro operace **MEMBER**, **INSERT** a **DELETE** závisí na (maximální) délce slov l, ale takové hloubky komprimované trie dosáhne jen v nejhorším případě. Chceme proto odhad očekávané hloubky, za předpokladu, že reprezentovaná množina S je vzorkem dat z rovnoměrného rozdělení (což bývá často přibližně splněno).

Označíme q_d pravděpodobnost, že komprimovaný trie nad množinou $S, |S| = n$ má hloubku aspoň $d_n = d$. Pak je naše očekávaná (střední) hodnota hloubky:

$$E(d_n) = \sum_{i=1}^{\infty} i(q_i - q_{i+1}) = \sum_{i=1}^{\infty} q_i$$

Odhadneme proto velikost q_d. Víme, že hloubka trie je menší než d, když prefixy o délce d slov z naší množiny rozlišují tato slova jednoznačně. Pravděpodobnost, že toto nastane, je (počet jednoznačných prefixů a k nim počet libovolných doplnění do délky l, děleno počtem jednoznačných slov délky l, vše nad abecedou o velikosti k):

$$P(\text{jednoznačné rozlišení o délce } d) = \frac{\binom{k^d}{n} k^{n(l-d)}}{\binom{k^l}{n}}$$

Z toho:

$$q_d \leq 1 - P(\text{jednoznačné rozlišení o délce } d-1) =$$
$$1 - \frac{\binom{k^{(d-1)}}{n} k^{n(l-d+1)}}{\binom{k^l}{n}} \leq 1 - \frac{\left(\prod_{i=0}^{n-1}(k^{d-1}-i)\right)k^{n(l-d+1)}}{k^{nl}} = 1 - \prod_{i=0}^{n-1}\left(1 - \frac{i}{k^{d-1}}\right) \leq 1 - \exp\left(\frac{-n^2}{k^{d-1}}\right) \leq \frac{n^2}{k^{d-1}}$$

Poslední kroky jsme mohli udělat, protože platí (integrál s nerovností můžeme použít, protože daný logaritmus je klesající, při výpočtu integrálu použijeme substituci za $1 - \frac{x}{k^{d-1}}$):

$$\prod_{i=0}^{n-1}\left(1 - \frac{i}{k^{d-1}}\right) = \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{k^{d-1}}\right)\right) \geq$$
$$\geq \exp\left(\int_0^n \ln\left(1 - \frac{x}{k^{d-1}}\right) \mathrm{d}x\right) = \exp\left((n - k^{d-1})\ln\left(1 - \frac{n}{k^{d-i}}\right) - n\right) \leq \exp\left(\frac{-n^2}{k^{d-1}}\right)$$

Očekávaná výška stromu pak vyjde, položíme-li $c = 2\lceil \log_k n \rceil$:

$$\sum_{i=1}^{\infty} q_i = \sum_{i=1}^{c} q_i + \sum_{i=c+1}^{\infty} q_i \leq \sum_{i=1}^{c} 1 + \sum_{i=c+1}^{\infty} \frac{n^2}{k^{i-1}} = c + \frac{n^2}{k^c}\left(\sum_{i=0}^{\infty} \frac{1}{k^i}\right) \leq 2\lceil \log_k n \rceil + \frac{1}{1 - \frac{1}{k}}$$

Trie v tabulce

Pokud se vzdáme operací **INSERT** a **DELETE**, můžeme trie reprezentovat na extrémně zcvrklém prostoru. Nejdříve si ho představme jako matici M dimenze $r \times s$, kde každý vnitřní vrchol odpovídá jednomu řádku a sloupce jsou písmena abecedy. Potom na pozici $M(v, a)$ je a-tý syn vrcholu v. Pole matice může obsahovat buď odkazy na další vrcholy (identifikátor řádku), nebo přímo slova, která jsou obsažena v reprezentované množině, nebo prázdnou hodnotu **null**. Ve vedlejším poli si musíme uchovat i hloubky vrcholů odpovídající nekomprimovanému trie – $\kappa(v)$.

Komprese matic – uložení do pole

Hodnoty **null** ale nepřinášejí novou informaci – stačí při průchodu maticí na další vrcholy testovat, zda nám stoupá hodnota $\kappa(v)$ a nakonec provést test shody s nalezeným prvkem. Díky tomu můžeme na místa, kde je **null**, v klidu ukládat něco jiného.

Matici M tak můžeme reprezentovat dvěma poli

- VAL – v něm budou hodnoty z různých řádků matice
- RD ("row displacement", "posunutí řádku") – bude udávat, kde začíná který řádek původní matice M ve VAL

Jednotlivé datové řádky původní matice se v poli VAL v klidu můžou překrývat, pokud překryté hodnoty jsou jen **null**. Formálně musíme zachovat, že když $M(i,j)$ je definováno, pak $M(i,j) = VAL(RD(i)+j)$ a že když $M(i,j)$ a $M(i',j')$ jsou definovány pro $(i,j) \neq (i',j')$, pak $RD(i)+j \neq RD(i')+j'$.

Pro nalezení "dobrého" rozložení řádků původní matice do pole VAL se používá algoritmus **First-Fit Decreasing**:

- Pro každý řádek původní matice M spočteme, kolik míst je non-**null** a setřídíme řádky matice podle této hodnoty v klesajícím pořadí
- Bereme řádky podle setřídění a vkládáme je na první místo od začátku pole VAL tak, že neporušují výše uvedené podmínky.

Označíme počet všech non-**null** hodnot jako m a počet non-**null** hodnot v řádcích s alespoň l non-**null** hodnotami jako m_l. Pokud řádky matice M splňují pravidlo *harmonického rozpadu*, tj. $\forall l : m_l \leq \frac{m}{l+1}$ (tj. např. více než polovina řádků obsahuje jen jednu skutečnou hodnotu), pak pro každý řádek i platí $RD(i) < m$ a algoritmus stavby polí potřebuje $O(rs + m^2)$ času (důkaz je hnusný).

Posouvání sloupců

Protože podmínku *harmonického rozpadu* splňuje jen málo matic, upravíme si obecné matice tak, aby ji splňovaly taky. Využijeme toho, že matici trochu "natáhneme" do počtu řádků (tím se, pravda, zvětší pole RD) a jednotlivé sloupce v ní rozstrkáme tak, aby v jednom řádku nevyšlo moc zaplněných míst najednou. Kde začíná který sloupec si zapamatujeme v dalším pomocném poli CD ("column displacement", "posunutí sloupce").

"Dobré" posunutí sloupců nalezneme obyčejným přístupem **First-Fit**, když pro každý sloupec j nalezneme nejmenší číslo $CD(j)$ splňující:

$$m(j+1)_l \leq \frac{m}{f(l, m(j+1))} \; \forall l = 0, 1, \ldots$$

Hodnota $m(j)$ je počet všech zaplněných míst v prvních j sloupcích právě konstruované matice a $m(j)_l$ je počet zaplněných míst v řádcích s alespoň l zaplněnými místy.

Pozorování:

- Je vidět, že každá funkce f musí splňovat $f(0, m(j)) \leq \frac{m}{m(j)} \; \forall j$, protože jinak by v algoritmu nemohla být splněna testovaná podmínka pro $l = 0$ (protože $m_j = m(j)_0$).
- Dále musí funkce f splňovat nerovnost $f(l,m) \leq l+1 \; \forall l$, aby výsledná matice splňovala podmínku *harmonického rozpadu*.

Dá se ukázat (a je to hnusný důkaz), že vhodná funkce je třeba $f(x,y) = 2^{x(2-\frac{y}{m})}$, protože splňuje obě podmínky a navíc výsledný vektor CD má délku s, vektor RD má délku menší než $4m \log\log m + 15.3m + r$ a vektor VAL má délku menší než $m + s$. Protože hodnoty CD indexují RD a hodnoty RD indexují VAL, plynou z toho omezení i na hodnoty v nich uložené.

Čas celého algoritmu vytváření matic je $O(s(r + m \log\log m)^2)$.

Další komprese vektoru RD

Protože M má jen m definovaných mít, z algoritmu pro výpočet RD plyne, že jen max. m míst v tomto vektoru bude různých od nuly. Proto můžeme použít následující kompresi (řekněme, že nenulových míst je t):

1. Vektor RD rozdělíme na n bloků o délce d.
2. Vytvoříme nový vektor CRD o délce t, který obsahuje jen nenulové prvky původního vektoru. Označme jejich původní pozice $i_j, j = 0 \ldots t-1$ a jejich pozice ve vektoru CRD jako $v(i_j)$.
3. Vytvoříme vektor $BASE$ o délce n, kde $BASE(x) = \begin{cases} -1 & i_j \div d \neq x \; \forall j = 0, \ldots t-1 \\ \min\{l; i_l \div d = x\} & \text{jinak} \end{cases}$
4. Vytvoříme matici $OFFSET$ typu $n \times d$, kde $OFFSET(x,y) = \begin{cases} -1 & x \cdot d + y \neq i_j \; \forall j \\ j - BASE(x) & x \cdot d + y = i_j \end{cases}$
5. Uložíme matici $OFFSET$ do vektoru OFF dimenze n tak, že z každého řádku vytvoříme číslo v soustavě o základu $d+1$: $OFF(x) = \sum_{k=0}^{d-1} (OFFSET(x,k)+1)(d+1)^k$.

Potom platí, že:

- $v(h) = 0 \Leftrightarrow OFFSET(h \div d, h \mod d) = -1$
- $v(h) = 1 \Rightarrow h = BASE(h \div d) + OFFSET(h \div d, h \mod d)$
- $OFFSET(i,j) = ((OFF(i) \div (d+1)^j) \mod (d+1)) - 1$

Celá tahle legrace má smysl, jen pokud $d \ll n$, a $t < n$. Když $d \leq \lceil \log\log n \rceil$, pak lze celé trie uložit pomocí pěti vektorů dimenze n s hodnotami menšími než $4n \log\log n$.

10.4 Haldy

Haldy se používají pro měnící se uspořádané množiny. Nevyžaduje se efektivní operace **MEMBER** (často se předpokládá s argumentem operace informace o uložení prvku). Požadují se malé nároky na paměť a rychlost ostatních operací.

Definice, operace

Halda je stromová struktura nad množinou (dat) S, jejíž uspořádání je dáno funkcí $f : S \to \mathbb{R}$, splňující *lokální podmínku haldy*:

$$\forall v \in S : f(\text{otec}(v)) \leq f(v), \text{ případně v duální podobě.}$$

Množina *je reprezentovaná* haldou, když přiřazení prvků vrcholům haldy je bijekce, splňující podmínku haldy. Různé druhy hald se liší podle dalších podmínek, které musí splňovat stromové struktury.

- Krom běžných operací můžu měnit uspořádání: operace **INCREASE** a **DECREASE** změní velikost f na nějakém daném prvku s se známým uložením o $+a$, $-a$.
- Další operace: **DELETEMIN** – smazání prvku s nejmenší hodnotou f.
- Pro operaci **DELETE** budeme požadovat přímé zadání uložení prvku.
- Navíc definujeme operaci **MAKEHEAP** – vytvoření haldy při známé množině a f,
- a **MERGE** – slití dvou hald do jedné, reprezentující $S_1 \cup S_2$ a $f_1 \cup f_2$, aniž by se ověřovala disjunktnost.

Regulární haldy

Pro d-regulární strom ($d \in \mathbb{N}$) s kořenem r platí, že existuje pořadí synů vnitřních vrcholů takové, že očíslování prohledáváním z r do šířky splňuje:

1. každý vrchol má nejvýše d synů
2. když vrchol není list, tak všechny vrcholy s menším číslem mají právě d synů
3. má-li vrchol méně než d synů, pak všechny vrcholy s většími čísly jsou listy

Potom takový strom s n vrcholy má max. jeden ne-list, který nemá právě d synů, jeho výška je $\lceil \log_d(n(d-1)+1) \rceil$. Čísla synů vrcholu s číslem k jsou $(k-1)d+2, \ldots, kd+1$, číslo otce je $1 + \lfloor \frac{k-2}{d} \rfloor$. Takto vytvořená halda umožňuje i efektivní reprezentaci v poli.

Operace na regulárních haldách

- Není známa efektivní operace **MERGE**.
- Máme pomocné operace **UP**, **DOWN**, posunující prvek níž/výš ve struktuře, dokud není splněna podmínka haldy ("probublávání").
- **INSERT** jen vloží nový prvek za poslední a spustí **UP**
- **DELETE** nahradí odstraněný prvek posledním listem a volá **UP** nebo **DOWN** podle potřeby
- **DELETEMIN** odstraní kořen, nahradí ho posl. listem a volá **DOWN**
- **MIN** jen vrátí kořen
- **INCREASE** a **DECREASE** změní hodnotu f nějakého prvku a zavolají **DOWN**, resp. **UP** (pozor, je to naopak, než názvy napovídají).
- Operace **MAKEHEAP** vytvoří libovolný strom a pak postupně od posledního ne-listu ke kořeni volá na všechno **DOWN**.

U všech operací je korektnost zajištěna podmínkou haldy (a tím, že **UP** a **DOWN** zaručí její splnění).

Složitost operací

Běh **DOWN** vyžaduje $O(d)$ a **UP** $O(1)$ v každém cyklu, takže celkem jde o $O(d \log |S|)$ a $O(\log |S|)$.

Haldu lze vytvořit opakovaným **INSERT**em v čase $|S| \log |S|$, ale pro větší množiny je rychlejší **MAKEHEAP** – uvažujeme-li, že operace **DOWN** vyžaduje čas odpovídající výšce vrcholu. Ve výšce $k-i$ je d^i vrcholů. Tím dostávám celkový čas $O(\sum_{i=0}^{k-1} d^i(k-i)d)$, což se dá odhadnout jako $O(d^2|S|)$.

Aplikace

Heapsort – vytvoření haldy a postupné volání **MIN** a **DELETEMIN**. Lze ukázat, že pro $d = 3, d = 4$ je výhodnější než $d = 2$, empiricky je do cca 1000000 prvků $d = 6$ nebo $d = 7$ nejlepší. Pro delší posloupnosti je možné d zmenšit.

Dijkstra – normální Dijkstrův algoritmus, jen vrcholy grafu uchovávám v haldě, tříděné podle aktuálního d (horního odhadu vzdálenosti). Složitost $O((m+n)\log n)$, pro $d = \max\{2, \frac{m}{n}\}$ je to $O(m \log_d n)$ a pro husté grafy ($m > n^{1+\varepsilon}$) je lineární v m.

Leftist haldy

Leftist halda je binární strom (T, r). Označme $\mathrm{npl}(v)$ délku nejkratší cesty z v do vrcholu s max. 1 synem. Leftist halda musí splňovat následující podmínky:

1. má-li vrchol 1 syna, pak je vždy levý
2. má-li 2 syny l, p , pak $\mathrm{npl}(p) \leq \mathrm{npl}(l)$
3. podmínka haldy na klíče prvků (ex. přiřazení prvků vrcholům stromu)

Pro leftist haldu se definuje pravá cesta (posl. pravých synů) a pokud máme takovou cestu délky k z vrcholu v, víme, že podstrom v do hloubky k je úplný binární strom. Délka pravé cesty z každého vrcholu je tedy logaritmická ve velikosti podstromu.

Operace jsou založeny na algoritmech **MERGE** a **DECREASE**.

- **MERGE** testuje prázdnost jednoho ze stromů (a pokud je jeden prázdný, vrátí ten druhý jako výsledek). Pokud ne, volá se rekurzivně na podstrom pravého syna kořene s menším klíčem dohromady s celým druhým a výsledek připojí místo onoho pravého syna. Pokud neplatí podmínka na npl, syny vymění.
- **INSERT** je to samé co vytvoření jednoprvkové haldy a zavolání **MERGE**.
- **DELETEMIN** je z**MERGE**ování synů kořene (a jeho zahození).
- **MAKEHEAP** je vytvoření hald z jednotl. prvků. Nacpu je do fronty a potom v cyklu vyberu dva první, zmerguju a hodím výsledek na konec, dokud mám ve frontě víc než 1 haldu.

INCREASE a **DECREASE** se dělají jinak.

- Mám pomocnou operaci **OPRAV**, která odtrhne podstrom a dopočítá všem vrcholům správné npl. Po odtržení vrcholu a příp. přehození pravého syna doleva jde nahoru, dokud provádí změny npl (možno až do kořene), vztahuje npl odspoda a příp. prohazuje syny.
- **DECREASE** se pak udělá snížením hodnoty ve vrcholu, zavoláním **OPRAV**, tj. jeho odříznutím od zbytku haldy, a **MERGE** podstromu a zbytku.

INCREASE: zapamatuju si levý a pravý podstrom vrcholu s mým prvkem a provedu na něj **OPRAV** (vyhodím ho), potom vyrobím nový vrchol s mým prvkem se zvednutou hodnotou a jako samostatnou haldu ho z**MERGE**uju s levým podstromem. Pravý podstrom z**MERGE**uju se zbytkem haldy a nakonec s tím z**MERGE**uju výsledek **MERGE** levého podstromu a zvednutého prvku.

Složitost

1 běh **MERGE** bez rekurze je $O(1)$, hloubka rekurze je omezena pravými cestami, takže je to $O(\log(|S_1| + |S_2|))$. Z toho plyne logaritmovost **INSERT** a **DELETEMIN**.

Pro **MAKEHEAP** se uvažuje, kolikrát projdou haldy frontou: po k projití frontou mají velikost 2^{k-1} a tedy fronta obsahuje $\lceil \frac{|S|}{2^{k-1}} \rceil$ hald. Jeden **MERGE** je $O(k)$ a jedno projití frontou pro všechny haldy tedy trvá $O(k\lceil \frac{|S|}{2^{k-1}} \rceil)$. Celkem dostávám $O(|S| \sum_{k=1}^{\infty} \frac{k}{2^{k-1}}) = O(|S|)$ (součet řady je 4).

OPRAV chodí jen po pravé cestě, takže má logaritmickou složitost. **INSERT**, **INCREASE** a **DECREASE** se díky ní dostanou taky na $O(\log |S|)$, protože jejich části kromě **MERGE** a **OPRAV** mají konstantní složitost.

Binomiální haldy

Binomiální stromy se definují rekurentně jako H_i, kde H_0 je jednoprvkový a H_{i+1} vznikne z dvou H_i, kdy se kořen jednoho stane dalším (krajním) synem kořenu druhého. Pak strom H_i má 2^i prvků, jeho kořen má i synů, jeho výška je i a podstromy určené syny kořene jsou právě $H_{i-1}, \ldots, H_0$.

Binomiální halda reprezentující S je seznam stromů T_i takový, že celkový počet vrcholů v těchto stromech je $|S|$ a je dáno jednoznačné přiřazení prvků vrcholům, respektující podmínku haldy. Každý strom je přitom izomorfní s nějakým H_i a dva $T_i, T_j, i \neq j$ nejsou izomorfní.

Existence binomiální haldy pro každé přirozené $|S|$ plyne z existence dvojkového zápisu čísla.

Operace

Operace na binomiálních haldách jsou založené na MERGE.

- **MERGE** pracuje stejně jako binární sčítání – za pomoci operace **SPOJ** (slepení dvou stromů, přilepím jako syna toho, který má v kořeni vyšší klíč) slepí stromy stejného řádu, přenáší výsledky do dalšího spojování (přenos + obě haldy mající strom daného řádu = vyplivnutí 1 stromu na výsledek a spojení zbývajících dvou).
- **INSERT** je MERGE s jednoprvkovou haldou.
- **MIN** je projití kořenů a vypsání nejmenšího.
- **DELETEMIN** je **MIN**, odebrání stromu s nejmenším prvkem v kořeni a přidání (**MERGE**) podstromů jeho kořene do haldy.
- **INCREASE** a **DECREASE** se dělají úplně stejně jako u regulárních hald.
- Přímo není podporováno **DELETE**, jen jako **DECREASE** + **DELETEMIN**.
- **MAKEHEAP** se provádí opakováním **INSERT**.

Složitost **MERGE** je $O(\log|S_1| + \log|S_2|)$, protože 1 krok **SPOJ** je konstantní. Halda má nejvýše $\log|S|$ stromů, takže **MIN** a **DELETEMIN** mají tuto složitost. Výška všech stromů je $\leq \log|S|$, což dává složitost **INCREASE** $O(\log^2|S|)$ a **DECREASE** $O(\log|S|)$. Pro odhad složitosti **MAKEHEAP** se použije amortizovaná složitost přičítání jedničky k binárnímu číslu, což je $O(1)$, tedy celkem $O(|S|)$.

Líná implementace

Vynecháme předpoklad neexistence dvou izomorfních stromů v haldě a budeme "vyvažování" provádět jen u operací **MIN** a **DELETEMIN**, kdy se stejně musí projít všechny stromy. **MERGE** je pak prosté slepení seznamů hald. Vyvažování se provádí operací **VYVAZ**, která sloučí izomorfní stromy (podobně jako **MERGE** z pilné implementace).

Složitost **INSERT** a **MERGE** je $O(1)$, ale **DELETEMIN** a **MIN** v nejhorším případě $O(|S|)$.

Amortizovaná složitost vychází ale líp: použijeme potenciálovou metodu, když za hodnocení konfigurace $w(H)$ zvolíme počet stromů v haldě $|\mathcal{H}|$. **INSERT** a **MERGE** ho nemění, resp. mění o 1, takže jsou stále $O(1)$.

Operace **VYVAZ** potřebuje $O(|H|)$, protože slití dvou stromů trvá konstantně dlouho a nelze slévat víc stromů, než kolik jich je v haldě. Kromě operace **VYVAZ** potřebuje **MIN** $O(|\mathcal{H}|)$ a **DELETEMIN** $O(|\mathcal{H}| + \log|S|)$ (max. stupeň stromu je logaritmický).

Dohromady vychází amortizovaná složitost pro **MIN**: $am(o) = t(o) - w(H) + w(H') = O(|\mathcal{H}| - |\mathcal{H}| + \log|S|)$, protože výsledný počet stromů $|H'|$ odpovídá pilné implementaci. Pro **DELETEMIN** podobně dostanu $O(|\mathcal{H}| + \log|S| - |\mathcal{H}| + \log|S|) = O(\log|S|)$.

Fibonacciho haldy

Definují se jako množiny stromů, které splňují podmínku haldy a *musely vzniknout posloupností operací z prázdné haldy*. Všechny operace zachovávají podmínku, že jednomu vrcholu lze odříznout max. dva syny. Strom má *rank* i, má-li jeho kořen i synů (podobné jako izomorfismus s H_i u binomiálních hald).

Podmínka odříznutí max. dvou synů se zachovává pomocnou operací **VYVAZ2**. Když vrchol není kořen a byl mu předtím někdy odříznut syn, je speciálně označený. **VYVAZ2** prochází od daného vrcholu ke kořeni a dokud nalézá označené vrcholy, odtrhává je i s jejich podstromy, ruší jejich označení a vkládá do haldy jako zvláštní stromy. Když se vrchol stane kořenem, označení se zapomene.

Operace

MERGE, INSERT, MIN a **DELETEMIN** jsou stejné jako v líné implementaci binomiálních hald, jen požadavek na isomorfismus s H_i je nahrazen požadavkem na daný rank. Pomocné operace z binomiálních hald **VYVAZ** a **SPOJ** jsou také stejné.

DECREASE, INCREASE a **DELETE** vycházejí z leftist hald. Používají pomocnou operaci **VYVAZ2**

- **DECREASE** odtrhne podstrom určený snižovaným vrcholem (není-li to už kořen), zruší u něj případné označení a vloží ho zvlášť do haldy, na odtržené místo zavolá **VYVAZ2**.
- **INCREASE** provede to samé, jen ještě roztrhá podstrom zvedaného vrcholu (odtrhne všechny syny, zruší jejich příp. označení a vloží jako samostatné stromy do haldy) a vloží zvednutý vrchol do haldy zvlášť.
- **DELETE** je to samé co **INCREASE**, bez přidání vrcholu zpět do haldy.

Korektnost a složitost

Operace **SPOJ** podobně jako u binomiálních hald vyrobí ze dvou stromů ranku i jeden strom ranku $i+1$. Operace **VYVAZ2** zajistí, že od každého vrcholu kromě kořenů byl odtržen max. 1 syn – když odtrhnu dalšího, odtrhu i tento vrchol a propaguju operaci nahoru.

Složitost operací:

- **MERGE** a **INSERT** je $O(1)$ (stejně jako u binomiálních hald)
- **MIN** má $O(|\mathcal{H}|)$ (nemění označení vrcholů)
- **DELETEMIN** $O(|\mathcal{H}|+\text{maxrank}(\mathcal{H}))$, kde *maxrank* udává maximální rank stromu v haldě (může navíc odznačit některé vrcholy)
- **DECREASE** je $O(1+c)$, kde c je počet odznačených vrcholů (navíc označí max. 1 vrchol)
- **INCREASE** a **DELETE** jsou $O(1+c+d)$, kde navíc d je počet synů zvedaného nebo odstraňovaného vrcholu (také označí navíc max. 1 vrchol).

Pro výpočet amortizované složitosti použijeme potenciálovou metodu a zvolíme hodnotící funkci w jako počet stromů v haldě + 2× počet označených vrcholů. Můžeme říct, že amortizovaná složitost **MERGE**, **INSERT** a **DECREASE** je $O(1)$.

Označme max. rank stromů v lib. haldě reprezentující n-prvkovou množinu jako $\rho(n)$. Amortizovaná složitost **MIN**, **DELETEMIN**, **INCREASE** a **DELETE** pak je $O(\rho(n))$ (pro **MIN** a **DELETEMIN** je vzorec amortizované složitosti podobný jako u binomiálních hald, pro **INCREASE** a **DELETE** je to vidět přímo ze vzorců).

Pro odhad $\rho(n)$ je potřeba znát fakt, že i-tý nejstarší syn libovolného vrcholu má aspoň $i-2$ synů (plyne z toho, že se slévají jen stromy stejného řádu a odtrhnout lze max. jednoho syna).

Vezmeme tedy nejmenší strom T_j ranku j, který toto splňuje. Ten musí být složením T_{j-1} a T_{j-2} (vzniká tak, že se slijí dva T_{j-1} a potom se na tom, který je pověšený jako syn nového kořenu, provede **DECREASE** a tím se z něj stane T_{j-2}). Z minimálního počtu synů se dá odvodit i rekurence $|T_k| \geq 1+1+|T_0|+\cdots+|T_{k-2}|$, která dá indukcí to samé.

Potom $|T_{k+1}| = F_k$, kde F_k je k-té Fibonacciho číslo. Pro Fibonacciho čísla platí, že $\lim_{k\to\infty} F_k = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^k$. Proto je $\rho(n) = O(\log(n))$, což dává logaritmickou amortizovanou složitost pro **MIN**, **DELETEMIN**, **INCREASE** a **DELETE**. Z toho pochází i název *Fibonacciho* haldy.

Aplikace

Fibonacciho haldy se díky své rychlosti **INSERT**, **DECREASE** a **DELETEMIN** často používají v grafových algoritmech. Praktické porovnání rychlosti s jinými haldami však není dosud přesně prostudováno.

Motivací pro vývoj *Fibonacciho hald* byla možnost **aplikace v Dijkstrově algoritmu**. Dává totiž složitost celého algoritmu $O(m+n\log n)$, což by mělo být lepší pro velké, ale řídké grafy proti d-regulárním haldám. O prakticky zjištěném "zlomu" ale nevíme.

Kapitola 11

Státnice - Hašování

11.1 Hashování

Základní motivací pro hashování je *slovníkový problém*, kdy máme za úkol reprezentovat množinu S prvků z nějakého univerza U a provádět na ní následující operace:

- **MEMBER** (je třeba, aby tato operace probíhala velmi rychle)
- **INSERT**
- **DELETE**

Aby byl **MEMBER** rychlý, bylo by nejlepší mít v paměti pole bitů o velikosti U. V případě, že $|S| << |U|$ (a navíc U může být neúnosně velké), použiji *hashovací funkci* $h : U \to \{0, \ldots, m-1\}$ a množinu S reprezentuji polem s m políčky tak, že $x \in S$ je uložen na indexu $h(x)$. Předpokládejme, že funkce h se dá spočítat v čase $O(1)$ – jiné funkce vlastně nemají smysl, protože nepřináší dostatečné zrychlení.

Problém je, když nastane *kolize*: $x \neq y$, $h(x) = h(y)$. Jednotlivé druhy hashování, které následují, se liší strategiemi předcházení a řešení kolizí.

Pro následující analýzy si označíme:

- $|S| = n$
- $|U| = N$
- *Load factor* (faktor zaplnění) – $\alpha = \frac{n}{m}$.

Hashování se separovanými řetězci

V tomto typu hashování se kolize řeší *řetězením ve spojácích*: pro každé políčko založíme zvlášť spoják všech prvků, které se do něj hashují. Všechny algoritmy je musí projít. Předpokládejme, že řetězce jsou prosté – nic se v nich neopakuje. V nejhorším případě mají všechny prvky stejný hash a máme jen jeden seznam.

Paměťová náročnost je pro každý seznam $O(1 + l(i))$, kde $l(i)$ je délka toho seznamu.

Existují dvě varianty – neuspořádaná a s s uspořádanými prvky v řetězcích. Liší se jedině v očekávaném počtu testů pro neúspěšné hledání (když dojdu v řetězci za místo, kde by byl hledaný prvek, můžu skončit).

Pro odhad složitosti alogritmů předpokládáme, že:

- Hashovací funkce h rozděluje data rovnoměrně
- Sama reprezentovaná množina S je náhodný výběr z U s rovnoměrným rozdělením

Tyto předpoklady v praxi ale splněny být nemusí.

Očekávaná průměrná délka řetězců

Pro odhad složitosti se počítá **očekávaná délka řetězců**. Označme délku i-tého řetězce jako $\mathbf{l}(i)$. Potom pravděpodobnost, že tento řetězec má délku l, odpovídá binomickému rozdělení:

$$P(\mathbf{l}(i) = l) = p_{n,l} = \binom{n}{l} \left(\frac{1}{m}\right)^l \left(1 - \frac{1}{m}\right)^{n-l}$$

Toto je jen aproximace (pro nekonečnou velikost univerza i seznamů), pro případ, že $N >> n^2 m$, ale lze použít. Očekávaná délka řetězce pak vychází jako (rozepíšu faktoriál a vytknu $\frac{n}{m}$, pak změním rozsah sumace $1 \ldots n$ (protože násobení $l = 0$ mi nic nedá), pak můžu z $l - 1$ udělat l a sumovat $0 \ldots n - 1$):

$$E(l) = \sum_{l=0}^{n} lp_{n,l} = \frac{n}{m} \sum_{l=0}^{n-1} \binom{n-1}{l} (\frac{1}{m})^l (1-\frac{1}{m})^{n-1-l} = \frac{n}{m}(\frac{1}{m} + 1 - \frac{1}{m})^{n-1} = \frac{n}{m} = \alpha$$

Vlastně tu ale objevujeme Ameriku tím, že počítáme střední hodnotu binomicky rozdělené veličiny s parametrem $\frac{1}{m}$ – ze vzorce nám vyjde to samé. Stejně tak rozptyl ze vzorce vyjde $\frac{n}{m}\left(1-\frac{1}{m}\right)$.

Očekávaná délka nejdelšího řetězce

Tento údaj však sám o sobě nestačí, počítá se i očekávaný nejhorší případ (očekávaná **délka nejdelšího řetězce**). Ta se definuje následovně:

$$EMS = \sum_j jP(\max_j \mathbf{l}(i) = j) = \sum_j P(\max_i \mathbf{l}(i) \geq j)$$

Z toho (pravděpodobnost disjunkce jevů je $\leq$ součet jednotlivých pravděpodobností; vyčíslení: počet podmnožin správné velikosti a pravděpodobnost, že mají stejný hash):

$$P(\max_i \mathbf{l}(j) \geq j) \leq \sum_i P(\mathbf{l}(i) \geq j) \leq m\binom{n}{j}(\frac{1}{m})^j = \frac{\prod_{k=0}^{j-1}(n-k)}{j!}(\frac{1}{m})^{j-1} \leq n(\frac{n}{m})^{j-1}\frac{1}{j!}$$

Najdeme mezní hodnotu j_0, pro které $n(\frac{n}{m})^{j-1}\frac{1}{j!} \leq 1$. Označme $k_0 = \min\{k | n \leq k!\}$. Potom $j_0 \leq k_0$. Ze Stirlingovy formule plyne, že $\log x! = \Theta(x \log x)$. Z toho odvodíme (hodně neformálně, asymptoticky):

$$\log k_0! = k_0 \log k_0 = O(\log n)$$
$$\log k_0 + \log\log k_0 \approx \log k_0 = O(\log\log n)$$
$$k_0 = \frac{k_0 \log k_0}{\log k_0} = O(\frac{\log n}{\log\log n})$$

A $j_0 = O(k_0)$. Pro $\alpha \leq 1$ platí, že $EMS = O(j_0)$:

$$EMS = \sum_j P(\max_i \mathbf{l}(i) \geq j) \leq \sum_j \min\{1, n(\frac{n}{m})^{j-1}\frac{1}{j!}\} = \sum_{j=1}^{j_0} 1 + \sum_{j=j_0+1}^{\infty}(n(\frac{n}{m})^{j-1}\frac{1}{j!}) \leq j_0 + \sum_{j=j_0+1}^{\infty}\frac{n}{j!} = \cdots \leq j_0 + \frac{1}{j_0}$$

A tedy očekávaná délka nejdelšího řetězce je $O(\frac{\log n}{\log\log n})$.

Očekávaný počet testů

Testy jsou porovnání toho, co hledáme, s nějakým prvkem, nebo zjištění, že řetězec je prázdný. Jejich očekávaný počet je další odhad efektivity struktury. Rozlišujeme úspěšné a neúspěšné hledání.

Neúspěšné hledání (Je-li délka řetězce 0, jeden test stejně provedu, jinak otestuji celý řetězec):

$$E(T) = p_{n,0} + \sum_l lp_{n,l} = (1-\frac{1}{m})^n + \frac{n}{m} \approx e^{-\alpha} + \alpha$$

S uspořádanými řetězci končím dřív ($e^{-\alpha} + 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1-e^{-\alpha})$).

Počet testů pro úspěšné vyhledávání je roven průměru počtu testů provedených při vložení každého z prvků, tj. 1 + očekávaná délka řetězce při každém vkládání: $\frac{1}{n}\sum_{i=0}^{n-1}\left(1+\frac{i}{m}\right) = 1 + \frac{n-1}{2m} \approx 1 + \frac{\alpha}{2}$.

Hashování s přemísťováním

Nevýhodou separovaných řetězců je nutnost alokovat další paměť, to je neefektivní. Proto zavedeme do hashovací tabulky pomocné ukazatele a celé řetězce nacpeme přímo do ní (a zřetězené prvky prostě rozházíme na jiné adresy). Pro *hashování s přemísťováním* se v tabulce uchovává navíc jednoduše odkaz na *předchozí a následující prvek* řetězce. Pokud vkládáme na místo, kde už je nějaký prvek z jiného řetězce, přehodíme tento cizí prvek jinam.

Algoritmy jsou téměř stejné jako pro separované řetězce, jen při **DELETE** prvního prvku řetězce je nutné na jeho místo přesunout druhý (pokud existuje).

Očekáváný počet testů je stejný jako pro hashování se separovanými řetězci. Přemísťování v tabulce je ale náročnější než 1 test, proto jsou **INSERT** a **DELETE** pomalejší.

Hashování se dvěma ukazateli

Od předchozího se liší tím, že místo ukazatele na předchozí prvek používá *odkaz na začátek řetězce* BEGIN. Řetězec tak už nemusí začínat na indexu svého hashe.

Místo přesouvání prvků algoritmy mění BEGIN (ten je na j-tém políčku vyplněn, právě když existuje řetězec prvků s hashem j).

- **INSERT** všechno vkládá na konec řetězce, zakládá-li nový, do BEGIN (na místě určené hashem) píše, kde se ve skutečnosti nachází
- **DELETE** jen upravuje odkazy na následující, nebo BEGIN (pokud maže poslední prvek řetězce).

Kvůli tomu, že řetězce začínají jinde než na svém místě, je počet testů o něco větší:

- Úspěšné hedání: $1 + \frac{(n-1)(n-2)}{6m^2} + \frac{n-1}{2m} \approx 1 + \frac{\alpha^2}{6} + \frac{\alpha}{2}$
- Neúspěšné hledání přibližně $1 + \frac{\alpha^2}{2} + \alpha + e^{-\alpha}(2+\alpha) - 2$.

Srůstající (coalesced) hashování

Srůstající hashování používá jen jeden ukazatel v hashovací tabulce navíc – odkaz na další prvek NEXT. Řetězce tak obsahují hodnoty s různými hashi. Prvek s vkládáme vždy do řetězce, obsahujícího $h(s)$-té políčko v tabulce.

Existují různé varianty:

- Standardní (bez pomocné paměti, "late" a "early" insertion) – LISCH, EISCH
- Bezpřívlastkové (s pomocnou pamětí, "late", "early" a "varied" insertion) — LICH, VICH, EICH.

Bez pomocné paměti – LISCH a EISCH

LISCH je "late insertion", tedy přidává se za poslední prvek řetězce. *EISCH* ("early insertion") přidává za první prvek řetězce.

- Algoritmus **MEMBER** je stejný pro oba (jen projití řetězce po odkazech NEXT).
- Alg. **INSERT**:
  - U LISCH projití celého řetězce (v případě že není prázdný, jinak jednoduše vložím na správné políčko) s testy na přítomnost prvku, potom vložení na libovolné volné místo v tabulce a připojení na konec řetězce.
  - Pro EISCH vložení na nějaké volné místo v tabulce a jen přepojení ukazatelů NEXT – připojení do řetězce za první prvek (pokud je řetězec neprázdný).

Algoritmy **DELETE** nejsou známy, kromě primitivních. Problémem je u nich zachování náhodného uspořádání prvků v řetězcích, které se předpokládá pro dodržení očekávaných časů operací. Je ale možné také prvky jen označit jako odstraněné a jejich místa použít při vkládání dalších (to ale zpomaluje hledání).

EISCH je kupodivu o něco rychlejší na úspěšné vyhledání (je větší pravděpodobnost práce s novým prvkem), očekávaný počet testů je stejný.

Počet testů v neúspěšném případě: Spočteme průměr přes všechny posloupnosti délky $n+1$ (kde hledáme $n+1$. prvek v množině ostatních n). Označme $c_{n,l}$ počet řetězců délky l, které přispívají celkem $1+2+\cdots+l = l + \binom{l}{2}$ porovnáními k sumě:

$$\frac{c_{n,0} + \sum_{l=1}^{n} l c_{n,l} + \sum_{l=1}^{n} \binom{l}{2} c_{n,l}}{m^{n+1}}$$

Tady $c_{n,0}$ představuje počet prázdných řádků, tedy $c_{n,0} = (m-n)m^n$, $lc_{n,l}$ je součet délek všech řetězců v reprezentacích n-prvk. množin, tedy $\sum_{l=1}^{n} l c_{n,l} = nm^n$.

Poslední člen označíme S_n. Při **INSERT**u do $n-1$-prvkové množiny jsou 2 možnosti vzniku řetězce délky l: buď přidávám do řetězce (délky $l-1$), nebo řetězec (pův. délky l) nezměním. Z toho vyjádříme rekurentní vztah pro S_n (úpravy: rozepsání rozdílů, vykrácení, rozpis l^2 jako $l^2 - l + l = \binom{l}{2} + l$):

$$S_n = \sum_l \binom{l}{2}(m-l)c_{n-1,l} + \sum_l \binom{l}{2}(l-1)c_{n-1,l-1} = mS_{n-1} - \sum_l l^2 c_{n-1,l} = (m+2)S_{n-1} + (n-1)m^{n-1})$$

Pak pomocí vztahu $T_n^c = \sum_{i=1}^{n} ic^i = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2}$ spočítaného z $(c-1)T_n^c = nc^{n+1} + (\sum_{i=2}^{n} - c^i) - c$ získáme nerekurentní vztah ($S_0 = 0$, obrácení sumace a vytknutí $m + 2^{n-1}$):

$$S_n = (m+2)^{n-1}S_0 + \sum_{i=0}^{n-1}(m+2)^i(n-1-i)m^{n-1-i} = (m+2)^{n-1}\sum_{i=1}^{n-1} i\left(\frac{m}{m+2}\right)^i = \frac{1}{4}(m(m+2)^n - m^{n+1} - 2nm^n$$

A tedy očekávaný počet testů vyjde:

$$1 + \frac{1}{4}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) \approx 1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$$

Počet testů pro úspěšný případ spočteme pro LISCH jako počet testů při vkládání prvku. Metoda EISCH pro tento postup nesplňuje předpoklady. Porovnání klíčů při neúspěšném vyhledávání je stejně při přístupu na obsazené políčko, neporovnávám ale nic při přístupu na neobsazené políčko, takže dostávám:

$$\frac{n}{m} + \frac{1}{4}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) = \frac{1}{4}\left(\left(1+\frac{2}{m}\right)^n - 1 + \frac{2n}{m}\right)$$

Průměr pro postupné vkládání všech prvků pak dává:

$$1 + \sum_{i=0}^{n-1}\frac{1}{4}\left(\left(1+\frac{2}{m}\right)^i - 1 + \frac{2i}{m}\right) = 1 + \frac{m}{8n}\left(\left(1+\frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) + \frac{n-1}{4m} \approx 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$$

Pro metodu EISCH vychází (bez důkazu):

$$\frac{m}{n}\left(\left(1+\frac{1}{m}\right)n - 1\right) \approx \frac{1}{\alpha}(e^{\alpha} - 1)$$

Všechny odhady mají odchylku $O(\frac{1}{m})$.

S pomocnou pamětí – LICH, VICH, EICH

V této variantě rozdělíme paměť na dvě části:

- (hash-funkcí) přímo adresovatelná
- pomocná část (bez přístupu hash-funkcí)

Při kolizích nejdříve ukládáme do řádků z pomocné části, pak teprve do přímo adresovatelné, tedy oddalujeme srůstání řetězců. Chování se tak až do určitého okamžiku podobá separovaným.

Existují tři varianty podle chování algoritmu **INSERT**:

- LICH vždy přidává na konec řetězce
- EICH v případě neprázdného řetězce vždy za 1. prvek
- VICH vždy za poslední prvek v pomocné paměti nebo (pokud žádné v pomocné paměti nejsou) za 1. prvek řetězce (tj. chová se na pomocné paměti jako LICH a v přímo adresovatelné části jako EICH).

Algoritmy až na VICH se chovají stejně jako ve standardním srůstajícím hashování, rozhodující je výběr volného řádku pro vložení: např. "vždy vyber z nejvyšší adresy" může zaručit používání pomocné paměti. Také tu není přirozené efektivní **DELETE**.

Odhad složitosti: definujeme si následující hodnoty:

- n – počet uložených prvků
- m – velikost přímo adresovatelné paměti
- m' – celková velikost paměti
- $\alpha = \frac{n}{m'}$ – faktor zaplnění
- $\beta = \frac{m}{m'}$ – adresovací faktor
- λ – jediné nezáp. řešení rovnice $e^{-\lambda} + \lambda = \frac{1}{\beta}$

Pokud je $\alpha \leq \lambda\beta$, pak pro všechny verze vychází očekávaný počet testů $e^{-\frac{\alpha}{\beta}} + \frac{\alpha}{\beta}$ v neúspěšném případě a $1 + \frac{\alpha}{2\beta}$ v úspěšném (chyba: $O(\log \frac{m'}{\sqrt{m'}})$.

V případě, že $\alpha \geq \lambda\beta$ (začínají srůstat řetězce), se metody liší a vychází divnosti. V neúspěšném případě je VICH a LICH lepší než EICH, v úspěšném vede VICH před EICH a LICH (vždy o jednotky procent). Doporučená hodnota $\beta = 0.86$. Na hledání volného řádku se v praxi hodí např. spojový seznam volných řádků.

Lineární přidávání

Tato a následující metoda nepoužívá žádné dodatečné položky v hashovací tabulce a zároveň řetězce kolidujích prvků ukládá přímo do ní. Nalezení dalšího prvku z řetězce je přímo v algoritmu.

Lineární přidávání je nejjednoduším řešením takové situace: v případě kolize při **INSERT**u nalezne nejbližší vyšší volné políčko a vloží nový prvek tam. Předpokládáme "cyklickou" paměť, tj. když dojdeme na konec, vkládáme od začátku.

Problémem je tvoření *shluků* – při velkém zaplnění se operace dost zpomalují. Také nepodporuje efektivní DELETE (a ani primitivní způsoby nejsou moc richlé). V praxi je dobré uchovávát počet uložených prvků nebo mít zarážku (nikdy neobsazované pole), abychom věděli, kdy dojde k přeplnění.

Očekávaný počet testů je $\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$ v neúspěšném a $\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)\right)$ v úspěšném případě (bez důkazu).

Dvojité hashování

Dvojité hashování je vylepšení předchozí metody tak, aby nevznikaly shluky. Výběr následujícího řádku bude závislý na předchozím, ale s rovnoměrným rozložením. Na to použiji *druhou hashovací funkci* h_2.

Při operacích **INSERT** pak hledám nejmenší i od 0, že $(h_1(x)+i\cdot h_2(x)) \mod m$ je volné políčko, tj. postupně přičítám $h_2(x)$ a modulím. Stejný postup je i pro operaci **MEMBER**.

Je nutné, aby $h_2(x) \not| m$, tj. abych měl prosté posloupnosti (a z každého políčka tak mohl vést řetězec po celé tabulce). Idea, že iterace h_2 tvoří pro každé x náhodnou permutaci paměťových míst, není úplně přesná, ale v praxi stačí, aby z $h_1(x)=h_1(y)$ plynulo, že $h_2(x)$ a $h_2(y)$ budou odlišné.

Funkce navíc musíme volit "chytře" (i lineární přidávání je spec. příp. dvojitého hashování, kdy $h_2\equiv 1$). Pak tato metoda je znatelně rychlejší než lin. přidávání. Předpoklad náhodnosti použitý v teoretické analýze sice splnit nelze, ale přiblížit se mu ano.

Očekávaný počet testů

Předpokládáme, že iterování funkce h_2 tvoří náhodné permutace (což, jak bylo řečeno, není úplně přesné).

Pro neúspěšný případ: označme $q_i(n,m)$ pravděpodobnost, že při zaplnění $\frac{n}{m}$ je pro nějaké x prvních $i-1$ políček, kam bych ho mohl vložit, plných. Potom $q_i(n,m)=\frac{\prod_{j=0}^{i-1}(n-j)}{\prod_{j=0}^{i-1}(m-j)}$ a tedy $q_i(n,m)=\frac{n}{m}q_{i-1}(n-1,m-1)$.

Očekávaný počet testů je (předposlední rovnost plyne z rekurentního vztahu pro q_j, poslední krok dokázat indukcí):

$$C(n,m)=\sum_{j=0}^{n}(j+1)(q_j(n,m)-q_{j+1}(n,m))=\sum_{j=0}^{n}(q_j(n,m))=1+\frac{n}{m}C(n-1,m-1)=\frac{m+1}{m-n+1}$$

Počet testů v úspěšném případě – stejná metoda jako u dřívějších analýz, takže vychází:

$$\frac{1}{n}\sum_{i=0}^{n-1}C(i,m)=\frac{1}{n}\sum_{i=0}n-1\frac{m+1}{m-i+1}\approx\frac{1}{\alpha}\ln\left(\frac{m+1}{m-n+1}\right)\approx\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

Srovnání

Podle počtu testů:

	neúspěšné	úspěšné
1.	separované uspořádané řetězce	separované (usp. i neusp.) řetězce, přemísťování
2.	separované řetězce, přemísťování	dva ukazatele
3.	dva ukazatele	VICH
4.	VICH, LICH	LICH
5.	EICH	EICH
6.	LISCH, EISCH	EISCH
7.	dvojité hashování	LISCH
8.	lineární přidávání	dvojité hashování
9.		lineární přidávání

- VICH je při vhodném α lepší než hashování se dvěma ukazateli.
- Lineární přidávání se nedá použít pro $\alpha>0.7$, dvojité hashování pro $\alpha>0.9$.
- Separované řetězce a obecné srůstající hashování používají víc paměti, přemísťování a dvojité hashování zas víc času, tj. nelze říct, které je jednoznačně lepší.

Implementační dodatky

- Pro hledání volných řádků se většinou používá seznam (zásobník).
- Přeplnění se většinou řeší držením α v rozumném intervalu ($\langle 1/4, 1\rangle$) a přehashováním do jinak velké tabulky ($2^i \cdot m$) při pře- nebo podtečení
- V praxi se doporučuje přehashování odkládat (např. pomocnými tabulkami) a provádět při nečinnosti systému.

DELETE se ve strukturách, které ho nepodporují, řeší označením políčka jako smazaného s možností využití při vkládání. V případě, že polovina polí je blokovaná tímto způsobem, se vše přehashuje. Pro srůstající hashování se toto používat nemusí, máme metody na zachování náhodnosti rozdělení dat.

V praxi je výhodné, známe-li něco o rozdělení vstupních dat, aby ho hashovací funkce kopírovala (většinou to ale nejde), jinak musíme předpokládat rovnoměrnost, což zaručeno zdaleka není. Nutnost rovnoměrného rozdělení vstupních dat lze obejít (viz níže).

11.2 Univerzální hashování

Abychom nemuseli předpokládat rovnoměrné rozložení vstupních hashovaných dat (což zdaleka není vždy zaručeno), budeme mít místo pevné hash-funkce nějakou množinu H, z níž funkci náhodně rovnoměrně vyberu. Analýza složitostí se pak dělá přes všechny $h \in H$ a platí pro všechny $S \subset U$ – i nerovnoměrné (S je daná pevně a h se k ní volí; $|U| = N$).

Pro formalizaci analýz je nutné mít analytické zadání funkcí h a znát přesnou velikost množiny $|H|$. To obejdeme očíslováním funkcí $H = \{h_i; i \in I\}$ a počítáním s indexovou množinou (očekávaná hodnota je průměr přes I). Při použití skutečné velikosti H v odhadech bychom dostali horší výsledky, protože některé funkce s různými indexy se můžou ve skutečnosti ukázat jako identické, a to tu zanedbáváme.

Definice: Systém funkcí $H = \{h_i; i \in I\} : U \to \{0, \ldots, m-1\}$ je *c-univerzální*, pokud:

$$\forall x, y \in U, x \neq y : |\{i \in I; h_i(x) = h_i(y)\}| \leq \tfrac{c|I|}{m}.$$

Tj. zaručuje se, že pro každé dva různé prvky má maximálně c funkcí kolizi.

Existence c-univerzálních systémů

Předpokládejme, že universum má tvar $U = \{0, 1, \ldots, N-1\}$ kde N je nějaké prvočíslo a vezmeme funkce typu

$$h_{a,b}(x) = ((ax + b) \mod N) \mod m$$

Jsou dobře použitelné, protože se dají počítat rychle. Protože N je prvočíslo, můžeme pracovat v $\mathbb{Z}_N$, coz je těleso. Rovnice $h_{a,b}(x) = h_{a,b}(y)$ je ekvivalentní s:

$$\exists i \in \{0, \ldots, m-1\} \wedge \exists r, s \in \{0, \ldots, \lceil \frac{N}{m} \rceil - 1\} : (ax + b \equiv i + rm) \mod N \wedge (ay + b \equiv i + sm) \mod N$$

Z Frobeniovy věty o jednoznačnosti řešení lineárních rovnic plyne, že pro každé r, s, i existuje jen jedna dvojice a, b, které vyhovuje. Počet řešení soustavy je tedy omezený číslem $m \cdot \lceil \frac{N}{m} \rceil^2$ ($i - m$ hodnot, $r, s - \lceil \frac{N}{m} \rceil$ hodnot pro daná x, y).

Pak je systém c-univerzální pro $c = \left(\lceil \frac{N}{m} \rceil\right)^2 / \left(\frac{N}{m}\right)^2$ a jeho velikost opdovídá N^2.

Vlastnosti

Vyrobíme si pomocnou funkci

$$\delta_i(x, y) = \begin{cases} 1 \text{ pro } h_i(x) = h_i(y), x \neq y \\ 0 \text{ jinak} \end{cases}$$

Chceme potom spočítat součet $\delta_i(x, S) = \sum_{y \in S} \delta_i(x, y)$. Z výsledku vidíme očekávanou délku řetězce pro libovolnou (jednu) množinu dat. Tohle pak sečtu přes všechny mé hash-funkce a z c-univerzality dostanu

$$\sum_{i \in I} \delta_i(x, S) = \sum_{y \in S} \sum_{i \in I} \delta_i(x, y) \leq \sum_{y \in S, x \neq y} c\frac{|I|}{m} = \begin{cases} (|S| - 1)c\frac{|I|}{m} & \text{pro } x \in S \\ |S|c\frac{|I|}{m} & \text{jinak} \end{cases}$$

Z toho dopočítám (podělením $|I|$) horní odhad očekávaného $\delta_i(x, S)$.

Výsledek: očekávaný čas operací **MEMBER**, **INSERT** a **DELETE** v c-univerzálním hashování je $O(1 + c\alpha)$ (kde faktor naplnění $\alpha = \frac{|S|}{m}$). Čas n po sobě jdoucích operací na původně prázdné tabulce je $O(n(1 + \frac{c}{2}\alpha))$. To není lepší hodnota než mají separované řetězce ($O(1 + \alpha)$), ale u nich předpokládám rovnoměrné rozdělení dat.

Výběr vhodné funkce není úplně jednoduchý, protože funkcí může celkem být např až N^2, tj. nelze ho provést jednoduchým zavoláním generátoru náhodných čísel, nýbrž např. náhodným vybráním každého bitu indexu funkce. Proto je výhodné najít co nejmenší c-univerzální systémy (viz dále).

Dolní odhady velikosti univ. systémů

Očíslujme hash-funkce z I a induktivně definujme množiny U_i jako největší podmnožiny U_{i-1} takové, že $h_{i-1}(U_i)$ je jednoprvková. Platí $|U_i| \geq \lceil \frac{U_{i-1}}{m} \rceil$, tedy $|U_i| \geq \lceil \frac{N}{m^i} \rceil$ – velikost těchto množin klesá s logaritmem a $|I| \geq \frac{m}{c}(\lceil \log_m N \rceil - 1)$. Takže velikost univ. systému roste alespoň úměrně logaritmu velikosti univerza.

Dolní odhad c

5-univerzální systém: Zvolme $t \in \mathbb{N}$ a k němu vezměme t-té prvočíslo p_t tak, že $t \ln p_t \geq m \ln N$. Definujme systém funkcí $H = \{g_{c,d,l}(x) | t < l \leq 2t, c, d \in \{0, 1, \ldots, p_{2t}-1\}\}$ kde $((c(x \mod p_l)+d) \mod p_{2t}) \mod m$. Zřejmě $|H| = tp_{2t}^2$.

Odhadem $|G = \{(c,d,l); h_{c,d,l}(x) = h_{c,d,l}(y)\}|$, když si množinu rozdělíme na $G_1 = \{(c,d,l) \in G; x \mod p_l \neq y \mod p_l\}$ a $G_2 = \{(c,d,l) \in G; x \mod p_l = y \mod p_l\}$, se dá dokázat, že systém je 5-univerzální, za dalších podmínek i 3.25-univerzální.

Dolní odhad c: Platí: $c > 1 - \frac{m}{N}$. Spočítáme $\sum_{h \in H} \sum_{x,y \in U} \delta_h(x,y)$ – pro pevnou h máme (z Cauchy-Schwarzovy nerovnosti, $u_{i,h} = |\{x \in U, h(x) = i\}|$):

$$\sum_{x,y \in U} \delta_h(x,y) = \sum_{i=0}^{m-1} u_{i,h}(u_{i,h} - 1) \geq \frac{(\sum_{i=0}^{m-1} u_{i,h})^2}{m} - N = \frac{N^2}{m} - N$$

Tedy $\sum_{h \in H} \sum_{x,y \in U} \delta(x,y) \geq \frac{|H|N(N-m)}{m}$. Zároveň platí $\sum_{h \in H} \sum_{x,y \in U} \delta(x,y) \leq \sum_{x,y \in U} c\frac{|H|}{m} = N^2 c\frac{|H|}{m}$, což mi dává výsledek.

11.3 Perfektní hashování

Základní ideou *perfektního hashování* je nalézt hash-funkci, která pro danou množinu S nedělá žádné kolize, takže operace **MEMBER** bude velice rychlá. Potom je nevýhoda, že nelze přirozeným způsobem realizovat operaci **INSERT**, tj. v praxi se nesmí moc často vyskytovat.

Tabulka by neměla být o mnoho větší než množina S a funkce h rychle spočitatelná a její realizace nezabírat moc paměti (tedy žádná zadávání tabulkou).

Definice

- Hashovací funkce h je perfektní pro množinu S, pokud pro $\forall x, y \in S, x \neq y : h(x) \neq h(y)$
- Soubor funkcí $H : U \to \{0, \ldots, m-1\}$ je (N, m, n)-*perfektní*, pokud $\forall S \subseteq U$ takové, že $|S| = n$ existuje $h \in H$ perfektní pro S.

Odhady velikosti

Každá funkce h je perfektní pro $\sum\{\prod_{j=0}^{n-1} |h^{-1}(i_j)|; 0 \leq i_0 < \cdots < i_{n-1} < m\}$ množin (sčítáme přes všechny množiny hashů $h(S)$ a pro každou z nich uvažujeme všechny možnosti, jak mohla vzniknout). Z Cauchy-Schwarzovy nerovnosti plyne, že tento výraz nabývá maxima, když $\forall i : |h^{-1}(i)| = \frac{N}{m}$. Každá funkce je tedy perfektní pro max. $\binom{m}{n}(\frac{N}{m})^n$ množin. Z toho plyne:

$$|H| \geq \frac{\binom{N}{n}}{\binom{m}{n}\left(\frac{N}{m}\right)^n}$$

Jiný odhad lze provést jako u c-univ. systémů s očíslovanými funkcemi $|H| = \{h_1, \ldots, h_t\}$. Používám induktivně definované množiny U_i, kde $U_0 = U$ a U_i je největší podmnožina U_{i-1}, kde je zrovna funkce h_i konstantní. Dostáváme $|U_i| \geq \frac{|U_{i-1}|}{m}$, tj. $|U_t| \geq \frac{N}{m^t}$, ale z perfektnosti plyne $|U_t| \leq 1$. Dostáváme $t \geq \frac{logN}{logm}$.

Existence

Reprezentujme soubor funkcí $H = \{h_1, \ldots, h_t\}$ na univerzu velikosti N pomocí matice $M(H)$ typu $N \times t$, takže $M(H)_{x,i} = h_i(x)$, tj. v jednom sloupci jsou výsledky jedné hashovací funkce pro všechny prvky univerza.

Pak žádná funkce z H není perfektní pro množinu $S \subset U$, když podmatice $M(H)$ tvořená řádky příslušejícími prvkům S nemá prostý sloupec. Takových matic je maximálně (počet všech funkcí minus počet prostých, to celé krát libovolné doplnění na N řádek):

$$\left(m^n - \prod_{i=0}^{n-1}(m-i)\right)^t \cdot m^{(N-n)t}$$

Podmnožin U velikosti n je pak $\binom{N}{n}$, čímž vynásobeno mám počet matic neodpovídajících (N,m,n)-perfektnímu systému. Všech matic je m^{Nt}. Potom existuje (N,m,n)-perfektní systém, když:

$$\binom{N}{n}\left(m^n - \prod_{i=0}^{n-1}(m-i)\right)^t \cdot m^{(N-n)t} < m^{Nt}$$

Příšernými kejklemi dostaneme podmínku existence $t \geq n(\ln N)e^{\frac{n^2}{m}}$.

Konstrukce funkce

Chceme splnit rychlou spočitatelnost a paměťovou nenáročnost. Předpokládáme univerzum prvočíselné velikosti a funkce typu:

$$h_k(x) = (kx \mod N) \mod m$$

Označme $b_i^k = |\{x \in S; (kx \mod N) \mod m = i\}|$. Potom pokud $h_k(x)$ není perfektní, pak nějaké $b_i^k = 2$ a mám $\sum_{i=0}^{m-1}(b_i^k)^2 \geq n+2$.

Odhadnu výraz $\sum_{k=1}^{N-1}((\sum_{i=0}^{m-1}(b_i^k)^2) - n) = \sum_{x\neq y\in S}\{k; 1 \leq k < N, h_k(x) = h_k(y)\}$. Z vlastností modula mám takových k pro daná x,y nejvýše $2\lfloor\frac{N}{m}\rfloor = 2\lfloor\frac{N-1}{m}\rfloor$. Dostávám tedy odhad $2(N-1)\frac{n(n-1)}{m}$ a z něj vidím, že existuje takové k, že $\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{2n(n-1)}{m} + n$, tedy pro tabulku velikosti $m > n(n-1)$ mám perfektní funkci.

Dá se dokázat trochu slabší předpoklad, že $P(k; \sum_{i=0}^{m-1}(b_i^k)^2 < \frac{3n(n-1)}{m} + n) \geq 1/4$, který je základem pravděpodobnostního algoritmu.

Pak mám deterministický algoritmus, který pro $m = n(n-1)+1$ nalezne perfektní h_k v čase $O(nN)$ a pravděpodobnostní, který pro $m = 2n(n-1)$ najde perfektní h_k v čase $O(n)$. Mám tedy konstrukci perfektní hash-funkce, ta ale nesplňuje požadavek na malou tabulku ($m = \Theta(n^2)$).

Menší tabulka

Zmenšíme-li velikost tabulky na $m = n$, bude výše uvedený algoritmus schopný nalézt funkci, pro kterou platí $\sum(b_i^k)^2 < 3n$ ($\sum(b_i^k)^2 < 4n$ v pravděpodobnostní variantě). Každou kolizi pak můžeme "rozstrkat" perfektní funkcí nad miniaturní tabulkou a celková velikost všech tabulek bude mnohem menší:

- Vezmeme nalezenou funkci a najdeme všechny neprázdné množiny $S_i = \{s \in S; h_k(s) = i\}$
- Pro jim odpovídající $c_i = |S_i|(|S_i|-1)+1$ (dvojnásobek v pravděp. metodě) najdeme k_i takové, že h_{k_i} je perfektní funkce pro S_i do tabulky velikosti c_i.
- Definujme $d_i = \sum_{j=0}^{i-1} c_j$, potom pokud $h_k(x) = l$, pak $g(x) = d_l + h_{kl}(x)$ je perfektní, její hodnota spočitatelná v čase $O(1)$ a hashuje do tabulky velikosti $O(3n)$ ($O(6n)$ s pravděp. případě), je naleznutelná v čase $O(nN)$ ($O(n)$) a pro její uložení do paměti jsou potřeba hodnoty k a k_i, vyžadující $O(n \log N)$ paměti.

Pro výpočet $g(x)$ potřebuji 2 násobení, 2 modulo a 1 sčítání (pro d_i v paměti), tabulka má velikost $\sum c_i \leq \sum(b_i^k)^2 < 3n$. Taková funkce ale stále nesplňuje požadavek na málo paměti pro uložení.

"Malá" funkce

Víme, že pro $m \in \mathbb{N}$ je počet prvočísel, která ho dělí $O(\frac{\log m}{\log\log m})$. Z toho úvahou o dělitelích čísla $D = \prod_{1\leq i<j\leq n}(s_j - s_i) \leq N^{n^2}$ na n-prvkové S a vzorce hustoty prvočísel $p_t \leq 2t\ln t$ dostanu, že existuje p o velikosti $O(\ln D) = O(n^2 \ln N)$ takové, že $\phi_p(x) = x \mod p$ je perfektní pro S.

Deterministické nalezení trvá $O(n^3 \log n \log N)$ (test perfektnosti každého systému je $O(n\log n)$). Proto použijeme pravděpodobnostní algoritmus (mezi $4cn^2\ln N$ přir. čísly je aspoň 1/2 prvočísel, která vyhovují): nejdřív najde prvočíslo a pak testuje perfektnost. Očekávaný počet testů je $O(\ln(4cn^2\ln N))$, celková složitost algoritmu je pak $O(n\log n(\log n + \log\log N))$. Najde zhruba až 2× větší prvočíslo než deterministický.

Tuto funkci použijeme ke konstrukci výsledné hash-funkce:

1. Nalezneme prvočíslo q_0, aby $\phi_{q_0}(x) = x \mod q_0$ byla perfektní pro S
2. Položíme $S_1 = \{\phi_{q_0}(s) | s \in S\}$, pak najdeme prvočíslo $q_1 \in \langle n(n-1)+1, 2n(n-1)+2\rangle$
3. K němu existuje $l \in \langle 1, q_0 - 1\rangle$ takové, že $h_l(x) = ((lx \mod q_0) \mod q_1)$ je perfektní pro S_1
4. Položíme $S_2 = \{h_l(s) | s \in S_1\}$ a najdeme perfektní g pro S_2 do tabulky s méně než $3n$ řádky (viz výše, počítá se ale do univerza o velikosti q_1).
5. Pak $f(x) = g(h_l(\phi_{q_0}(x)))$ je perfektní.

Funkce q_0 je určena 1 číslem o velikosti $O(n^2 \log N)$, h_l 2 čísly o velikosti $O(n^2)$ a $O(q_0)$, g je určená $n+1$ čísly o $O(q_1)$, tj. celkem zadání vyžaduje $O(n\log n + \log n + \log\log N)$ paměti.

Kapitola 12

Státnice - Dynamizace datových struktur

12.1 Úvod

Mnoho datových struktur podporuje velice efektivní operaci **MEMBER**, ale nemá operace **INSERT** a **DELETE**. Úkolem *dynamizace* je právě tyto operace do jakékoliv obecné struktury co nejefektivněji doplnit.

Zobecněný vyhledávací problém

Zobecněný vyhledávací problém je libovolná funkce $f : U_1 \times 2^{U_2} \to U_3$ – pro prvek a nějakou množinu vrací nějakou hodnotu.

Struktura $\mathcal{S}$ je *statická struktura řešící vyhledávací problém* (neboli *S-struktura*) f, je-li dán algoritmus, který pro $A \subseteq U_2$ zkonstruuje datovou strukturu $\mathcal{S}(A)$ a algoritmus, který pro $x \in U_1$ a $\mathcal{S}(A)$ spočte $f(x, A)$.

Struktura je *semidynamická*, pokud navíc existuje algoritmus, který pro $x \in U_2$ a $\mathcal{S}(A)$ zkonstruuje S-strukturu $\mathcal{S}(A \cup \{x\})$.

Struktura je *dynamická*, pokud navíc existuje algoritmus, který pro $x \in A$ a $\mathcal{S}(A)$ zkonstruuje S-strukturu $\mathcal{S}(A \backslash \{x\})$.

Ve všech následujících operacích budeme uvažovat jen *rozložitelný* vyhledávací problém:

- Vyhledávací problém je *rozložitelný*, pokud existuje binární operace $\circ$ na univerzu U_3 taková, že pro disjunktní $A, B \subseteq U_2$ a $x \in U_1$ platí $f(x, A) \circ f(x, B) = f(x, A \cup B)$.

To platí pro situaci, která nás nejvíc zajímá – pokud funkce f provádí operaci **MEMBER** nad nějakou datovou strukturou. Jiné problémy (patří x do konvexního obalu množiny?) ale rozložitelné nejsou.

Navíc budeme předpokládat, že funkce $\circ$ lze spočítat v konstantním čase. Označme

- $Q_{\mathcal{S}}(n)$ čas na vyčíslení $f(x, A)$, pokud $|A| = n$
- $S_{\mathcal{S}}(n)$ paměťový prostor potřebný k reprezentaci n-prvkové množiny
- $P_{\mathcal{S}}(n)$ čas na vytvoření struktury $\mathcal{S}(A)$ pro $|A| = n$

Pro asymptotické odhady budeme předpokládat, že funkce $Q_{\mathcal{S}}(n)$, $\frac{S_{\mathcal{S}}(n)}{n}$ a $\frac{P_{\mathcal{S}}(n)}{n}$ jsou neklesající.

12.2 Semidynamizace

Vytvoříme semidynamickou strukturu, jejíž čas operace **INSERT** bude velmi rychlý a navíc se nám moc nepokazí doba pro provedení operace **MEMBER**. Základní idea je podobná jako v binomiálních haldách – roztrháme reprezentovanou množinu A na sadu *disjunktních* podmnožin A_i o velikosti 2^i (takové množiny jsou neprázdné pro i odpovídající jedničkám v binárním zápisu čísla $|A|$.

Formálně budeme mít spojový seznam S-struktur $\mathcal{S}(A_i)$ odpovídajících množinám A_i, potom seznam spojových seznamů $s(A_i)$, které obsahují prvky jednotlivých množin, a nakonec pomocnou dynamickou strukturu T (např. binární vyhledávací strom). To všechno dohromady zjevně vyžaduje jen $O(S_{\mathcal{S}}(n))$ paměti (celkem reprezentujeme pořád stejně prvků a vyhledávací stromy i spojáky jsou $O(n)$).

Struktura T se používá k "rychlému" testování před **INSERT**em, zda nechceme vkládat prvek, který už v množině máme, případně před **DELETEM**, jestli nechceme mazat neexistující prvek. Pro jednoduchost to v popisech operací vynecháme.

Operace MEMBER

Operace **MEMBER** potom projde všechny neprázdné množiny A_i a zavolá **MEMBER** na nich. Výsledek spojí za pomoci funkce $\circ$.

To celkem dává složitost **MEMBER**u $O(Q_{\mathcal{S}}(n)\log(n))$, protože hledáme v max. $\log_2$ dílčích množinách.

Protože jakékoliv mocniny rostou rychleji než logaritmus, platí, že pokud $Q_{\mathcal{S}} = n^\epsilon$ pro nějaké $\epsilon > 0$, pak je operace **MEMBER** $O(n^\epsilon)$.

Operace INSERT

Operace **INSERT** najde nejmenší i takové, že $A_i = \emptyset$. Potom vezme všechny $A_j, j < i$ (ty jsou neprázdné) a spolu s vkládaným prvkem je slije do jedné množiny A_i (zahodí S-struktury pro všechna $A_j, j < i$, zkonkatenuje spojáky $s(A_j)$ a na jejich základě zkonstruuje novou S-strukturu $\mathcal{S}(A_i)$). Nová množina má správný počet prvků, protože $\sum_{j=0}^{i-1} 2^j + 1 = 2^i$.

V odhadu amortizované složitosti využijeme fakt, že S-struktura pro A_i se tvoří jen tehdy, když existují S-struktury pro všechny $A_j, j < i$. Tj. S-struktura o velikosti 2^i prvků se konstruuje znovu až po dalších $2^{i+1} - 1$ **INSERT**ech. Amortizovaně tak vychází :

$$\sum_{i=0}^{\log n} \frac{P_{\mathcal{S}}(2^i)}{2^i} \leq \sum_{i=0}^{\log n} \frac{P_{\mathcal{S}}(n)}{n} = \frac{P_{\mathcal{S}}(n)}{n} \log n = O(\frac{P_{\mathcal{S}}(n)}{n} \log n).$$

Podobně jako pro **MEMBER** platí, že pokud $P_{\mathcal{S}} = n^\epsilon$ pro nějaké $\epsilon > 1$, pak je složitost operace **INSERT** $O(n^{\epsilon-1})$.

Operace INSERT se zaručeným nejhorším časem

Protože někdy nestačí mít **INSERT** rychlý amortizovaně, byla vytvořena jeho varianta, která zaručuje rozumnou rychlost i v nejhorším případě. Dělá se to prostým rozložením potřebných operací do všech volání. Musíme ale povolit i požadavky na naši strukturu – místo max. jedné množiny pro každou velikost odpovídající mocnině dvou budeme mít max. 4 takové množiny $A_{i,0}, \ldots A_{i,3}$, z nichž poslední navíc bývá zkonstruovaná jen částečně ("rozpracovaná").

Označme počet množin A_i jako k_i, přičemž $k_i = -1$, pokud neexistuje žádná množina velikosti 2^i. Algoritmus potom vypadá následovně:

1. Vytvoří se jednoprvková $A_{0,k_0+1} = \{x\}$ a zvýší k_0 (tj. i odpovídající S-struktura)
2. Pro první souvislý úsek, kde $k_i \geq 0$:
   (a) Provedeme dalších $\frac{P_{\mathcal{S}}(2^i)}{2^i}$ konstrukce S-struktury $\mathcal{S}(A_i, k_i)$
   (b) Pokud jsme tím tuto S-strukturu dotvořili, vezmeme první dvě struktury "o patro níž" ($A_{i-1,0}$ a $A_{i-1,1}$) a připravíme je prvním krokem k sloučení. Zároveň je odtud odebereme a přečíslujeme zbylé. Jejich započaté sloučení přidáme na úroveň i.
3. Pokud jsme v posledním kroku (i, t.ž. $k_{i+1} = 0$) vytvořili druhou strukturu stejné velikosti na nejvyšším dosaženém "patře", připravíme první krok sloučení těchto dvou největších struktur, přemístíme jejich započaté sloučení "o patro výš" a odebereme původní struktury.

Díky tomu, že počet kroků volíme tak šikovně, vždycky dotvoříme strukturu právě včas na to, abychom mohli vytvářet další stejné velikosti. Navíc, protože se provede jen $O(\frac{P_{\mathcal{S}}(2^i)}{2^i})$ kroků pro každé i, odpovídá složitost v nejhorším případě amortizované složitosti původní operace **INSERT**.

12.3 Dynamizace

Při úplné *dynamizaci* navíc požadujeme efektivní algoritmus **DELETE**. Přidáme další předpoklad, a to, že na naší původní S-struktuře existuje operace **FAKE-DELETE** (ta spočívá v označení nějakého prvku jako "smazaného", aniž by se smazal skutečně – dál tedy zabírá místo a prodlužuje operace).

Základní ideou oproti semidynamizaci navíc je, že se budeme snažit za každou cenu zajistit, aby všechny naše pomocné S-struktury stále obsahovaly alespoň osminu "nesmazaných" prvků a přestavovat je až poté, co počet "smazaných" překročí tuto mez. Formálně naše S-struktury reprezentují vlastně množiny B_i, kde $B_i = A_i \backslash \{x_i; i = 1, 2 \ldots k\}$ (a $2^{i-3} < |A_i| \leq 2^j$).

Asymptoticky tak zaručíme, že množin (a pomocných S-struktur) bude stále jen logaritmicky mnoho a paměťové nároky taky zůstávají asymptoticky stejné. Struktura je tedy stejná jako pro (základní) dynamizaci. Navíc jsou nyní spojové seznamy prvků přímo provázané s S-strukturami (protože už nepoznáme podle velikosti seznamu, ke které S-struktuře vlastně patří).

Operace **MEMBER** pak pracuje úplně stejně, jen nakonec zkontroluje, zda nalezený prvek nebyl označen jako "smazaný" (a případně ho pak nevrátí).

Operace INSERT

Algoritmus operace **INSERT**(x) je následující:

1. Najdi nejmenší j takové, že $|\cup_{i\leq j} A_i| < 2^j$.
2. Vytvoříme $A_j = \cup_{i\leq j} A_i \cup \{x\}$ (včetně S-struktury a spojáku). To splňuje požadavky na velikost $2^{j-1} < |A_j| \leq 2^j$, jinak by pro j nebylo nejmenší.
3. Položíme $A_i = \emptyset$ pro $i < j$.

Díky zachovávané minimální velikosti vytvářené množiny máme složitost amortizovaně stejnou jako v případě semidynamizace.

Operace DELETE

Odebrání prvku x z naší dynamické struktury vypadá následovně:

1. Odstraníme x ze spojáku
2. Vyřešíme čtyři různé případy podle velikosti množiny A_i, která obsahuje prvek x:
   (a) Je-li A_i jen jednoprvková, prostě zahodím struktury s ní spojené.
   (b) Je-li $|A_i| > 2^{i-3}$ (tj. ještě mám víc než osminu prvků "platných"), provedeme pouze **FAKE-DELETE**.
   (c) Je-li A_{i-1} buď prázdná, nebo dost velká ($|A_{i-1}| > 2^{i-2}$), můžeme ji s A_i prohodit. Pro "nové" A_{i-1} pak vytvoříme novou S-strukturu.
   (d) Je-li A_{i-1} neprázdná, ale moc malá na prohození s A_i, potom je určitě můžeme sloučit a získáme novou množinu, která splňuje velikostní omezení pro A_i. Pro ni vyrobíme novou S-strukturu.

Amortizovaná složitost celé operace **DELETE** je $O(D_{\mathcal{S}}(n) + \log n + \frac{P_{\mathcal{S}}(n)}{n})$, kde $D_{\mathcal{S}}(n)$ je čas potřebný na operaci **FAKE-DELETE** a $\log n$ je potřeba na vyhledání prvku. Odhadneme, kolikrát se dělá skutečné přestavění S-struktur. Pokud $x \in A_i$, pak **DELETE** může nově vytvořit jen $S(A_i)$ nebo $S(A_{i-1})$. Z jejich podmínek velikosti (v algoritmu) vidíme, že mezi dvěma **DELETE** pro stejné A_j (přičemž $|A_j| \leq 2^{i-2}$) se musí provést aspoň 2^{j-3}-krát **FAKE-DELETE** (a libovolný počet **DELETE** na jiných množinách). Amortizovaná složitost vytváření S-struktur tak vychází:

$$\frac{P_{\mathcal{S}}(2^{j-2})}{2^{j-3}} = O(\frac{P_{\mathcal{S}}(n)}{n})$$

I se současně prováděnými operacemi **DELETE** se amortizovaná složitost operací **INSERT** zachovává. Aby **INSERT** vytvořil podruhé strukturu pro A_i, musí opět nastat $1 + \sum_{j\leq i} |A_j| > 2^{j-1}$. Protože **DELETE** nikdy nevytvoří strukturu s víc než polovinou skutečně zaplněnou, musí se do té doby provést aspoň 2^{i-2} **INSERT**ů, čímž získáme stejnou situaci jako u semidynamizace.

Kapitola 13

Státnice - Datové struktury ve vnější paměti

13.1 Základní pojmy

- Logickou jednotkou dat je *záznam*, který má *atributy* se jmény a doménami (uvažujeme max. jednu hodnotu pro každý atribut).
- *(Homogenní) soubor* je kolekce (multimnožina) záznamů. Na souboru jsou definovány operace **INSERT**, **DELETE**, **UPDATE** a **FETCH**.
- Pro soubory s neopakujícími se záznamy je *klíč* množina atributů, které záznam jednoznačně identifikují v souboru. Jeden nich z klíčů se označuje jako *primární*.
- *Vyhledávací klíč* je něco jiného – k jeho jedné hodnotě se dá najít množina odpovídajících záznamů. Jsou tři druhy vyhledávacích klíčů: *hodnotový*, *hashovaný* a *relativní* (přímo pozice v souboru).
- Logickému záznamu odpovídá *fyzický záznam* délky R, BÚNO na magnetickém disku; ten může obsahovat další data — oddělovače, ukazatele, hlavičky. Záznamy mají buď pevnou, nebo proměnnou délku.
- Fyzické záznamy jsou organizovány do *bloků* délky B – hlavní jednotky přenosu mezi RAM a HDD.
- $\frac{B}{R}$ se nazývá *blokovací faktor* ($\lfloor b \rfloor$).
- *Schéma organizace souborů* (SOS) je popis logické paměťové struktury, ve které se soubor nachází. Může obsahovat více logických souborů, ten který nese uživatelská data je *primární*, jeho délka v počtu záznamů se značí N. Další operace nad SOS jsou **BUILD, REORGANIZATION, OPEN, CLOSE**.
- Dotaz je každá funkce, která pro zadaný argument vrátí "odpověď" – množinu záznamů. Dotazy mohou být buď na *úplnou*, nebo jen *částečnou shodu*, popř. *intervalovou shodu*.
- *Vyváženost struktury* je zajištění omezení délky cesty při vyhledávání nějakým výrazem ($O(\log N)$ atp.), popř. rovnoměrnosti naplnění bloků (popisuje ji *faktor naplnění stránek*). Splnění se dosahuje štěpením a sléváním bloku.
- SOS, které splňuje vyváženost, se nazývá *dynamické*, jinak *statické*.

Fyzická média

Soubory se fyzicky ukládají hlavně na magnetické pásky nebo HDD.

- *Pásky* umožňují jen sekvenční přístup, bloky jsou při vyhledávání čteny a kontrolovány sekvenčně (vhodné setřídění dát podle klíče). Pro kapacitu pásky je důležitá hustota, jsou nutné meziblokové mezery. Sekvenční čtení trvá $t' = R \cdot t/(R + W)$, kde W jsou meziblokové mezery, t je transfer rate.
- *HDD* umožňuje přímý přístup k datům. Uvažují se *hlavy*, *válce (cylindry)* a *stopy*. Vždy jen jedna hlava může číst. Většinou se HDD dělí na sektory. Pro rychlost přístupu jsou důležité náledující proměnné:
  - *seek* (přesun na jiný válec) — s
  - *rotate* (doba 1 půlotáčky) — r
  - *block transfer time* — propustnost sběrnice — btt

Nové disky mají různý počet sektorů na 1 stopu a tím pádem složitý výpočet cylindru a hlavy z čísla bloku, který dělá řídící elektronika. Udávaná adresa cylindr-hlava-sektor tak nemusí mít nic společného se skutečností.

Př. nechť 1 stopa = 512 K. Načtení 1 MB pak trvá minimálně $s + r$ pro nalezení prvního sektoru a $2 \times 2r$ za načtení stop. Při čtení z náhodně rozmístěných 4 K bloků ale vyjde $256 \times (s + r + btt)$ — tj. až 100x pomalejší.

Časový odhad doby přístupu k záznamu T_F je složitější:

- $s + r + btt$ obecně
- $r + btt$ pro další záznam ve stejném cylindru

Máme i operaci **REWRITE** — přepis (během 1 otáčky disků): $T_{RW} = 2r$. Další časy se značí T_I (**INSERT**), T_D (**DELETE**), T_U (**REWRITE**), T_Y (**REORG**), T_X (načtení celého souboru).

13.2 Statické metody organizace souborů

Sekvenční soubor

Jsou dvě varianty:

- *Neuspořádaný (hromada)* – jenom fyzicky za sebe naplácané záznamy. Složitost nalezení $O(N)$.
- *Uspořádaný* – záznamy řazeny podle klíče. Aktualizované záznamy se umisťují do zvláštního neuspořádaného souboru, **REORGANIZATION** celek znova setřídí. Nalezení je opět $O(N)$ nebo $O(N/\lfloor b \rfloor)$. U médií s přímým přístupem ale $O(\log_2 N)$.

Index-sekvenční soubor

Primární soubor je sekvenční soubor setříděný podle primárního klíče a nad ním (i víceúrovňový) index: číslo bloku a minimální hodnota klíče v něm; pro vyšší úroveň to samé, ale na blocích indexu nižší úrovně.

Nejvyšší úroveň indexu je obvykle jen jeden blok (*master*). Počet úrovní se dá spočítat: $x = \lceil \log_p N/\lfloor b \rfloor \rceil$, kde $p = \lfloor B/(V + P) \rfloor$ a V je velikost klíče a P velikost pointeru na blok nebo záznam.

Zařazení nového záznamu – problémy: přidávání do oblasti přetečení a v ní řetězení záznamu za sebe (každý záznam tam má pointer na další záznam v oblasti přetečení, nejenom blok). Pro oddálení nutnosti vkládat do oblasti přetečení lze bloky plnit na míň než 100 %.

Indexovaný soubor

Máme primární soubor a indexy pro různé vyhledávací klíče. Indexují se přímo záznamy, ne jen bloky. Primární soubor už nemusí být setříděný. Varianta: *clusterovaný index* – záznamy se společnou hodnotou 1 atributů jsou blízko sebe (jde jen pro jeden atribut).

Index může být podobný jako u index-sekvenčního souboru, ale lepší je, když pro záznamy se stejným klíčem je ve všech úrovních až na první (nejnižší) jen jeden klíč, který pak odkazuje na seznam záznamů. Pro aktualizace se nepředpokládá oblast přetečení, ale změny v indexu. $T_F = (1 + x)(s + r + btt)$.

Lze použít i dotazy na kombinovanou částečnou shodu, ale trvají dlouho. Alternativa: *kombinovaný index* pro více atributů; to ale vyžaduje analýzu, na co jsou dotazy nejčastější.

Bitové mapy

Bitové mapy jsou efektivní způsob indexace pro atributy s malou doménou (max. stovky). Pro každou hodnotu této domény vyrobíme vektor bitů délky N, kde jednička na i-té pozici indikuje, že i-tý záznam má právě tuto hodnotu atributů.

Booleovské dotazy potom fungují jako bitové operace nad těmito vektory. Vektory bitů lze navíc komprimovat — nejsou tak velké, vhodné pro databáze s hodně záznamy.

Indexy v DIS (document information system)

Jedná se o tzv. *invertované soubory* pro fulltextové hledání – pro každé slovo máme uložen počet souboru, kde se vyskytuje, a pointer na soubor souřadnic, tj. dvojic [dokument, pozice]. Dotazy na shodu pro více slov pak jsou množinové průniky (začínající od slova s nejmenším počtem výskytu v databázi).

Získání invertovaného souboru: rozparsování na slova, setřídění, odstranění duplicit, výpočet frekvencí slov. *Zipfův zákon* distribuce slov $f = k/r$, kde k je konstanta, r je pořadí četnosti, f je četnost. Taky lze využít kompresi a nebo zmenšit indexy vyhozením nevýznamných slov.

Soubor s přímým přístupem

Záznamy v primárním souboru ("adresový prostor") jsou rozptýleny pomocí *hashovací funkce*.

Implementace: buď hash = číslo stránky; nebo hash = číslo stránky i relativní pozice v ní. Pro kolize se snažím umístit záznamy pokud možno do stejné stránky.

13.3 Statické hashovací metody

Jako perfektní hashování se označuje nejen stav, kdy funkce nedává pro danou množinu žádné kolize, ale i takový systém, kde máme zaručený čas $O(1)$ pro přístup k záznamu.

Cormackovo perfektní hashování

Hashování odpovídá "velké funkci a malé tabulce" ze sekce o hashování. Máme soubor a adresář. Krom primární hashovací funkce h jsou potřeba další hashovací funkce $h_i(k,r) = (k \mod (2i + 100r + 1)) \mod r$, kde:

- r je velikost oboru hodnot (počet kolizí)
- i je nějaký vhodný parametr — číslo hash fce (hledá se postupným zkoušením, dokud není výsledek bez kolizí).

V adresáři uchováváme pro každý záznam odkaz do primárního souboru, r a i. V primárním souboru máme klíč a data. Při hledání:

1. Zahashujeme klíč pomocí h, dostaneme se do adresáře, pokud je tam $r = 0$, nenašli jsme.
2. Pokud $r \geq 1$, spočítám podle uloženého i hash-fci h_i, vezmu odkaz do primárního souboru, přičtu výsledek h_i a v primárním souboru zkontroluju, jestli jsem našel co jsem hledal.

Pro **INSERT** spočítám h a:

1. Pokud na daném místě v adresáři nic není, najdu volné místo v primárním souboru délky 1 a vložím
2. Pokud tam už něco je, najdu volné místo délky $r + 1$, zvětším r a najdu i, aby kolizní od sebe rozházela a přeházím je na nové místo.

r nemusím zvětšovat o 1, ale rychleji, takže najdu požadované i lépe (pokud mám blbý h_i, někdy r prostě musím zvětšit o víc).

Toto hashování lze použít i pro dynamický paměťový prostor, když z primárního souboru uděláme nesouvislý prostor, kam lze přidávat stránky.

Perfektní hashování Larson-Kalja

Uchováváme menší pomocnou tabulku než předchozím případě – pro každou stránku adresového prostoru máme jen jednu hodnotu – *separátor*. Máme:

- Množinu M hashovacích funkcí h_i, které vytvářejí pro každý záznam posloupnost stránek, do kterých ho můžeme zkoušet vkládat
- Navíc druhou sadu jim jednozn. odpovídajících funkcí s_i, které počítají *signaturu*.

Pokud je separátor stránky větší než signatura záznamu, bude záznam v této stránce, jinak ho tam nemůžu vložit. Po přeplnění vyhodím záznamy s největší signaturou a zmenším separátor. Prvek se nemusí povést vložit (když mi dojdou obě sady funkcí).

Pokud vkládám do plné stránky, nemůžu prvek vložit, i když má prvek menší signaturu než separátor — separátor pak musím zmenšit a ze stránky vyhodit i něco dalšího, čímž dojde ke kaskádování.

Toto hashování je vhodné pro málo vkládání a hodně čtení – mám zaručený jen jeden přístup na disk, protože malý adresář se do paměti vejde.

Dotazy na částečnou shodu

Máme-li záznamy s n atributy, použijeme pro každý z atributů zvláštní hashovací funkci, která generuje d_i-bitový výsledek (*signaturu atributu*). *Signatura* (hash) celého *záznamu* je pak konkatenací signatur atributů. Pro adresový prostor o velikosti $M = 2^d$ stránek volíme délky signatur atributů tak, aby $\sum d_i = d$.

Dotaz na částečnou shodu je potom dotaz, kde bity některých atributů jsou nahrazeny "?". Nazveme *s-bitovou signaturu dotazu*, má-li zadaných s bitů. Takový dotaz je 2^{d-s}-krát dražší než přímý konkrétní.

Je-li pravděpodobnost dotazů na i-tý atribut rovna P_i, vyjde průměrná cena dotazů:

$$\sum_{q \subseteq \{1,..n\}} (P_q \cdot \Pi_{i \notin q} 2^{d_i})$$

Ideální rozvržení d_i proto vychází (Lagrangovými multiplikátory):

$$d_i = (d - \textstyle\sum_{j=1}^{n} \log_2 P_j)/n + \log_2 P_i.$$

Je nutné upravit případné extrémy a přepočítat.
Pro urychlení dotazů můžeme použít *deskriptory stránek*:

- Máme *deskriptor záznamů*: $w = w_1 + .. + w_n$-bitový řetězec, kde pro každý atribut $A_1, ..A_n$ máme právě jednu jedničku (může být zadáno jinak, ex. vzorce pro ideální w_i i počet jedniček k nim).
- Deskriptor stránek je bitový OR deskriptorů všech záznamů ve stránce.
- Při hledání vyrobím *deskriptor dotazů*: místo "?" dám samé nuly. Pokud má dotaz někde v deskriptoru "1" a stránka "0", nemusím tam hledat.

Lze udělat i dvouúrovňově – s *deskriptory segmentů*.
Další možnost urychlení jsou *Grayovy kódy*, které se snaží řešit nesouvislost oblasti stránek se záznamy vyhovujícími dotazům (např. 10???1010). Jde o jiný způsob kódování binárních čísel — dvě po sobě jdoucí čísla se liší vždy jen v jednom bitu. Max. zisk — o 50% lepší než binární. Konverze čísla do Grayova kódování z pozičního vypadá následovně:

$$G(x) = B(x) \text{ xor } B(\lfloor \frac{x}{2} \rfloor)$$

Zpětně (g_i je i-tý bit Grayova kódu, n-tý bit je nejvyšší):

$$b_n = g_n,\ b_i = g_i \text{ xor } b_{i+1}$$

13.4 Dynamické hashování

Rozšiřitelné hashování – Fagin (Koubkovo "externí hashování")

Kromě primárního souboru (nebo souborů, protože jde o dynamickou strukturu) mám pomocnou strukturu – *adresář* s pointery na stránky o velikosti 2^d (přičemž některé stránky mohou být v adresáři uvedeny víckrát) a používám hashovací funkci s rovnoměrným rozdělením.

Vždy použiju *jen prvních d bitů hashe*, minimum kolik potřebuju. Podle nich určím záznam v adresáři a z něj najdu stránku. Pro stránku se může používat míň bitů, proto na ní může ukazovat víc záznamů v adresáři. Když se stránka naplní, rozštěpím ji na 2 podle dalšího bitu hash-funkce.

Pokud už stránka používá stejně bitů jako adresář, musím zvětšit o 1 bit i adresář (a zdvojnásobit jeho velikost), s ostatními stránkami nic nedělám. Při vypouštění záznamů lze stránky slévat, pokud si pamatuju jejich zaplněnost (můžu slévat jen stránky, které se liší jen v posledním bitu).

Adresář můžeme ukládat na jedné stránce externí paměti. Potom **FETCH** jsou max. 3 operace a **INSERT** nebo **DELETE** max. 6 operací s externí pamětí. Očekávaný počet použitých stránek je $\frac{n}{b \ln 2}$, kde b je počet prvků v jedné stránce, a velikost adresáře $\frac{e}{b \ln 2} n^{1+\frac{1}{b}}$ – to je víc než lineární růst, tj. nelze používat donekonečna (bez důkazů).

Lineární hashování – Litwin

V tomto případě nemám adresář, jen oblast přetečení, přístupnou pointry z primárních stránek. Data vkládám do primárních stránek, po každých L **INSERT**ech se vynuceně štěpí určená stránka (v pořadí podle čísel stránek $0, 1, 00, 01, 10, 11, 000$). Podle hashe (jeho konec musí odpovídá číslu stránky) se rozdělují při štěpení prvky.

FETCH: když mám aktuálně n stránek, vezmu posledních $\lfloor \log_2 n \rfloor + 1$ bitů hashe (pokud je to $> n$, zanedbám horní bit) a tím dostanu číslo stránky. Když v ní prvek není a stránka je přetečená, podívám se ještě do oblasti přetečení.

Při štěpení je nutné vrátit do rozštěpených stránek i záznamy z oblasti přetečení. Problémem jsou více zaplněné stránky na konci, tedy ještě nerozštěpené. Řešením je buď nerovnoměrné rozdělení hash-fce, nebo skupinové štěpení stránek.

Skupinové štěpení stránek

Rozšíření Litwina pro lepší využití stránek:

- Stránky jsou vždy organizovány vždy v s skupinách po g stránkách (začínáme s s_0 skupinami, postupně se s zvětšuje, g zůstává).
- Mám inicializační hash-funkci h do $\{0 \dots (g \cdot s_0) - 1\}$.
- Štěpím také po L vložení, vždy g stránek do $g + 1$, na to mám nezávislé hashovací funkce $h_0, ...h_\infty$ do $\{0 \dots g\}$.
- Až dostanu s skupin po $g + 1$ stránkách, přeorganizuju stránky zpátky do skupin po g (můžu přidat nějaké prázdné, aby to vycházelo).

Při hledání se počítají všechna prehashování úplně od začátku — je to docela HW náročné, ale proti rychlosti disků se vyplatí.

Spirálová paměť

Tohle je další rozšíření Litwinova hashování, tentokrát pro exponenciální rozdělení klíčů. Místo rozdělení jedné stránky na starou a novou tu starou zahodí a přidá dvě nové.

Klíče jsou nejprve rovnoměrně rozděleny hash-funkcí $G(c,k) \to \langle c, c+1\rangle$ $(c \in \mathbb{R}_0^+)$, pak přepočteny do $\langle b^c, b^{c+1}\rangle$ a zaokrouhleny na konkrétní čísla stránek. Při změně c se mění velikost prostoru.

Při expanzi se nové c volí tak, aby zničilo stránku, která se štěpí (nejnižší číslo):

$$c_{n+1} := log_b(f+1) \text{ (kde } f \text{ je číslo 1. stránky).}$$

Aby se mi neposouvaly všechny stránky pořád dál od 0, první zahozené stránce dám nové číslo a znova ji použiju – přepočítávání log. stránek na fyzické se pak dělá rekurzivně:

1. Je-li $\lfloor$logická stránka$/b\rfloor < \lfloor$(1+logická stránka)$/b\rfloor$, pak fyzická stránka := fyzická stránka($\lfloor$logická stránka$/b\rfloor$)
2. Jinak fyzická stránka = $\lfloor$log. stránka$/b\rfloor$.)

Hashovací funkce zachovávající uspořádání

Kvůli urychlení intervalových dotazů se objevily hashovaci metody, zachovávající uspořádání klíčů:

- *Lineární hashování pro intervalové dotazy* – vychází z Litwina, využívá nejvýznamnější bity k určení stránky.
- *Částečně lineární hashování zachovávající uspořádání* – vychází ze znalosti rozdělení klíčů, dělí doménu klíčů na nestejně dlouhé intervaly, aby vyvažovalo nerovnoměrnost rozdělení. Nelze ale pořád reorganizovat, takže se upravují jenom přilehlé intervaly při vkládání a odebírání. Nehodí se pro dynamicky rostoucí doménu klíčů. Může být provedeno víceúrovňově.

13.5 Třídění na vnější paměti

Pro třídění něčeho, co se nevejde do operační paměti, se používá **MERGESORT**:

1. Ze dvou souborů vezmu dva prvky
2. Vyberu menší z nich, zapíšu ho na výstup
3. Ze souboru, odkud ten menší pocházel, načtu další.
4. Konec setříděného úseku poznám tak, že další načtený prvek je menší než ten vypsaný. Pokud dojdu na konec setříděného úseku na jednom ze vstupů, dokopíruju zbytek setříděného úseku i z druhého vstupu na výstup.

Postupně dostávám delší setříděné kusy. Původní použití — setříděni kousků, které se vešly do paměti, a slévání na páskách. Dnes je použitelné i na HDD.

N-cestný **MERGESORT** na vnější paměti: Mám-li $n+1$ stránek v paměti:

1. Vytvořit setříděné běhy velikosti n stránek (použít **HEAPSORT** nebo **QUICKSORT**).
2. Pak v každém kroku slévat (maximálně) n nejkratších běhů, výsledek ukládat jako 1 běh.

Pro M stránek v celém souboru je složitost $O(2M\lceil \log_n M/n\rceil)$ — celé projde $\log_n(M/n)$-krát procesem slévání.
HEAPSORT může vytvořit i delší běhy, než co se vejde do paměti:

1. Průběžně odebírám a vypisuju minimální prvky a načítám další.
2. Pokud je načtený prvek větší než minimum, hodím ho na haldu.
3. Pokud je menší, dám ho do aktuálně vznikající haldy na druhém konci pole.

Až se vyčerpá halda, začnu nový běh. V nejhorším případě dopadne stejně jako třídění v paměti, průměrně je 2x lepší, ideálně setřídí všechno.

Kapitola 14

Státnice - Třídění

14.1 Třídění založené na porovnávání

Existuje mnoho algoritmů, známe i (za určitých podmínek) dolní odhad složitosti.

Heapsort

HEAPSORT je třídění pomocí (např.) d-regulárních hald, jen lokální podmínku haldy používáme v duální podobě a máme funkci **DELETEMAX** (která ale funguje stejně jako **DELETEMIN**). Postupně se odebírají maxima a setříděná posloupnost se staví na jejich místě od konce pole. Iterace končí, když v haldě zbývá jen jeden prvek. Celkem $O(n \log n)$ operací.

Mergesort

MERGESORT je snad nejstarší "chytrý" třídící algoritmus. Pracuje s frontou, do které na počátku nahází "předtříděné" rostoucí úseky, potom je v cyklu vybírá a jejich slití vrací zpátky, dokud nemá jen jednu posloupnost. Slití je vybrání vždy menšího na výstup a na konci dokopírování zbytků.

Jedna operace slití vyžaduje $O(n+m)$, jsou-li 2 posloupnosti dlouhé n a m prvků. První rozházení vyžaduje lineární čas, potom každé projití všech prvků slitím vyrábí posloupnosti délky $\geq 2^{i-1}$, tedy počet projití všech prvků je $\lceil \log n \rceil$ a celková složitost $O(n \log n)$. Je adaptivní na předtříděné posloupnosti a při omezeném počtu rostoucích úseků dosahuje lineární složitosti.

Quicksort

QUICKSORT je asi nejpoužívanější třídící algoritmus, v průměru má při rovnoměrném rozložení nejnižší očekávaný čas. Využívá techniky rozděl a panuj.

1. Vybere prvek a (pivot).
2. Vytvoří posloupnosti $a_1 \dots a_{k-1}$ prvků menších než a a $a_{k+1} \dots a_n$ větších než k.
3. Na ty sám sebe zavolá rekurzivně, do výsledku zapíše za sebe obě setříděné poloviny.

Procedura (bez rekurze) vyžaduje čas k nebo $n-k$ v jednom běhu, tj. pro a medián, kdyby $n-k=k$, by měl celý algoritmus složitost $O(n \log n)$. Medián lze nalézt v lineárním čase, ale pak by byly **MERGESORT** i **HEAPSORT** rychlejší, proto se jako a bere např. první prvek posloupnosti.

Potom má procedura očekávaný čas $O(n \log n)$, ale nejhorší případ $O(n^2)$, což je pro neznámé rozdělení dat nevhodné. Náhodná volba by celý běh také mohla dost zpomalit, proto se v praxi bere medián tří až pěti pevně zvolených prvků.

Očekávaný čas Quicksortu

Dva libovolné prvky a_i, a_j jsou porovnávány maximálně jednou, a to když a_i nebo a_j je pivot a předtím ani jeden z nich pivot nebyl. Vezmeme si náhodnou veličinu $X_{i,j}$, která má hodnotu 1, když **QUICKSORT** porovnal během výpočtu b_i a b_j z nějaké výsledné setříděné posloupnosti $(b_i)_{1 \leq i \leq n}$, a 0 jinak. Potom $\mathbf{E}X_{i,j} = p_{i,j}$, kde $p_{i,j}$ je pravděpodobnost porovnání. Potom celk. počet porovnání v celém běhu je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}X_{i,j} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j}$$

Pro výpočet $p_{i,j}$ uvažujeme "strom rekurze", kde každý vrchol odpovídá jednomu rekurzivnímu volání procedury a s tím i nějaké podposloupnosti $a_i, \ldots, a_j$ (do té už se sahá jen v jeho podstromě). V jeho levém podstromě jsou operace na úsecích prvků menších než pivot $a_i, \ldots a_{l-1}$ této posloupnosti a v pravém na větších $a_{l+1}, \ldots a_j$. Přiřadíme každému vrcholu jeho pivot a očíslujeme vrcholy prohledáváním do šířky. Pak $X_{i,j} = 1$, když b_j nebo b_i je první pivot z množiny $\{b_i, \ldots b_j\}$ v tomto očíslování (protože kdyby to byl jiný prvek z této množiny, rozdělím b_i a b_j, aniž bych je porovnal; naopak prvky mimo tuto množinu coby pivoty od sebe b_i a b_j neoddělí). Množina $\{b_i, \ldots, b_j\}$ má $j-i+1$ prvků, takže $p_{i,j} = \frac{2}{j-i+1}$. Počet porovnání pak vyjde

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2n \sum_{k=2}^{n} \frac{1}{k} \leq 2n \int_1^n \frac{1}{x} \mathrm{d}x = 2n \ln n$$

Jednoduché třídění

- **SELECTIONSORT** třídí vybíráním nejmenšího prvku a jeho prohozením s levým krajním v nesetříděném úseku.
- **INSERTSORT** vkládá do setříděného úseku další prvek a postupným vyměňováním ho řadí na správné místo.
- **SHELLSORT** je jeho vylepšení – postupně **INSERTSORT**em třídí sekvence složené z každého k-tého prvku pro klesající $k \to 1$ (sekvence klesajících k musí být zvolena "šikovně").
- **BUBBLESORT** – iterativně prochází posloupností a prohazuje inverze
- Jeho varianta **SHAKESORT**, která posloupnosti prochází tam a zpátky.

A-sort

Tento algoritmus je aplikací (a, b) stromů v třídících algoritmech, vhodnou pro částečně předtříděné posloupnosti. Jinak proti klasickým algoritmům nemá žádné výhody. Pro algoritmus je nutné znát list s nejmenším prvekm FIRST, cestu k němu od kořene a pro každý list ukazatele na následující v uspořádání NEXT.

Postup: Odzadu (od "předtříděně největšího") vkládat prvky do stromu modifikovaným A-INSERTem a pak přečíst posloupnost listů (jít po NEXT). A-INSERT pracuje tak, že místo pro vložení prvku hledá od FIRST (jde postupně nahoru po otcích a hledá, kde nejdřív může slézt zas k listům).

Složitost: Pomalejší než běžné třídění na libovolná data (asymptoticky stejné), ale rychlejší na částečně předtříděná. Vezmeme F – počet inverzí v posloupnosti. Celk. potřebuju $O(n)$ pro načtení prvků, $O(n)$ pro všechna štěpení dohromady ze všech běhů A-INSERTu a na každé vložení $O(h)$ pro nalezení místa, kde h je výška, kam se z FIRST dostanu, přeskočím tak $f_i \geq a^{h-2}$ vrcholů (menších než vkládaný) a $h \in O(\log f_i)$. Součet f_i za $\forall i$ dává:

$$\sum_{i=1}^{n} \log f_i = n \log(\prod_{i=1}^{n} f_i)^{\frac{1}{n}} \leq n \log \frac{\sum_{i=1}^{n} f_i}{n} = n \log \frac{F}{n}$$

Protože se nepoužívá DELETE, hodí se na toto $(2,3)$ stromy. Pro míru $F \leq n \log n$ má složitost $O(n \log \log n)$, v urč. případech i rychlejší než Quicksort.

Porovnání algoritmů

Algoritmus	Čas nejhůř	Čas ∅	Porov. nejhůř	Porov. ∅	PP	Paměť	AD
QUICKSORT	$\Theta(n^2)$	$9n \log n$	$n^2/2$	$1.44n \log n$	A	$n + \log n + c$	N
HEAPSORT	$20n \log n$	$\leq 20n \log n$	$2n \log n$	$2n \log n$	A	$n + c$	N
MERGESORT	$12n \log n$	$\leq 12n \log n$	$n \log n$	$n \log n$	N	$2n + c$	A
A-SORT	$O(n \log(F/n))$	$O(n \log(F/n))$	$O(n \log(F/n))$	$O(n \log(F/n))$	A	$5n + c$	A
SELECTIONSORT	$2n^2$	$2n^2$	$n^2/2$	$n^2/2$	A	$n + c$	N
INSERTSORT	$O(n^2)$	$O(n^2)$	$n^2/2$	$n^2/4$	N	$n + c$	A

c je nějaká konstanta, F značí počet inverzí v posloupnosti, PP – přímý přístup, AD – adaptivní na předtříděné.

Pro krátké posloupnosti je do délky 22 vhodný **SELECTIONSORT**, do 15 **INSERTSORT**, jinak **QUICKSORT**, což vede k hybridnímu algoritmu. Pro **A-SORT** jsou nejvhodnější $(2,3)$-stromy. Poměr časů **QUICKSORT, MERGESORT, HEAPSORT** je v průměru $1 : 1.33 : 2.22$, platilo to ale v roce 1984 :-).

Vylepšení Mergesortu

Nedosahuji optimálních výsledků, pokud slévané posloupnosti ve frontách nejsou přibližně stejně dlouhé. Proto provedu úvahu: mějme algoritmus, který slévá rostoucí posloupnosti a uvažujme jeho "slévací" strom T (kde posloupnost $P(v)$ odp. vrcholu v (délky $l(P(v))$) vznikne slitím posloupností z jeho synů). Součet časů pro MERGESORT je pak $O(\sum\{l(P(v))|v$ vnitřní vrchol $T\}) = O(\sum\{d(t)l(P(t))|t$ list $T\}$. Dále pracujeme jen s délkami posloupností, vytvoříme algoritmus **OPTIM**, který při slévání sumu minimalizuje – na začátku dá každé jednoprvkové posl. hodnotu c, která odpovídá hodnotě jejího prvku (?). Pro slévání vybírá posloupnosti (stromy) s nejmenším c a slitému stromu přiřadí $c_1 + c_2$. Nakonec zbyde v množině stromů jen jeden, a ten je optimální. Pro třídění fronty podle c se používá BUCKETSORT. Celkem pracuje v čase $O(\sum_{i=1}^{n} l(P_i))$ na posloupnosti rostoucích úseků $P_1, \ldots, P_n$.

14.2 Přihrádkové třídění

Bucketsort (Counting sort)

Algoritmus **BUCKETSORT** třídí jen přirozená čísla z intervalu $< 0, m >$ a to zavedením $m+1$ množin, do kterých je rozhází a nakonec tyto spojí do výsledku. Třídění je stabilní pro opakující se prvky, inicializace množin a projití při konkatenaci potřebují $O(m)$, rozházení prvků pak $O(n)$, takže celkem $O(n+m)$.

Varianta **RADIXSORT** umí třídit i ve větších intervalech, když používá BUCKETSORT na každou jednotlivou číslici. Protože BUCKETSORT je stabilní, bude to celé fungovat.

Hybrid Sort

Sofistikovanější verze **HYBRIDSORT** třídí reálná čísla z $\langle 0, 1\rangle$ (a obecně tedy jakékoliv klíče). Má dané α (počet přihrádek v poměru k n), rozhazuje do $k = \alpha n$ přihrádek a ty pak třídí haldou.

Nejhorší možný čas je $O(n \log n)$, protože nejhůře se může stát, že všechny prvky nacpu do 1 přihrádky. Očekávaný čas pro nezávislé rovnoměrně rozdělené prvky je $O(n)$ – pravděpodobnost velikostí množin se řídí binomickým rozdělením s parametrem $1/k$, střední hodnota pak je:

$$\mathbf{E}(\sum_{i=1}^{k}(1 + X_i \log X_i)) \leq \mathbf{E}(\sum_{i=1}^{k}(1 + X_i^2)) = k + k\left(\frac{n(n-1)}{k^2} + \frac{n}{k}\right) = O(n)$$

Jak je vidět z odhadu, i kdybychom použili algoritmus s kvadratickou složitostí ve třídění jednotlivých přihrádek, zůstane očekávaná složitost lineární.

Wordsort

WORDSORT je modifikace BUCKETSORTu pro třídění slov. Příprava:

1. Rozhodí slova do množin L_i podle jejich délky.
2. Vytvoří dvojice pozice-písmeno $\{(j, a_i[j])\}$, kterou setřídí podle druhé složky BUCKETSORTem, výsledek setřídí podle první složky (stejně), tj. má seznam setříděných dvojic pozice-písmeno, které se ale mohou opakovat.
3. Dvojice rozstrká do množin S_i pro každou pozici i a odstraní duplicity.

Pak v hlavním cyklu postupuje od největší možné délky a pro každé i pracuju jen s množinou slov délky $\geq i$:

1. Podle i-tého písmena rozhodím všechna aktuálně tříděná slova do množin T_x.
2. Potom podle množiny S_i vyberu všechna neprázdná T_x a sliju je za sebe.
3. Pro další krok přidávám kratší slova vždy na začátek množiny aktuálně tříděných.

Výpočet délek slov, inicializace L_i a zařazení slov do L_i vyžadují čas $O(L)$, kde L je součet délek všech řetězců. Vytvoření seznamu dvojic a jeho třídění vyžaduje také $O(L)$. Založení S_i a přeházení dvojic do nich také $O(L)$. Inicializace T_x je $O(|\Sigma|)$, kde $|\Sigma|$ je velikost abecedy. V hlavním cyklu v každém kroku potřebuji dvakrát čas $O(l_i)$ (součet délek slov dlouhých i nebo víc). Celkem tedy $O(L + |\Sigma|)$.

14.3 Pořadové statistiky (hledání mediánu)

Vstupem je (neuspořádáná) posloupnost n (navzájem různých) čísel. Výstup je $\lfloor \frac{n}{2} \rfloor$-té, nebo obecně k-té nejmenší číslo z nich. Složitost budeme měřit *počtem porovnání*.

Z dvou následujících **FIND** bývá rychlejší než **SELECT** pro většinu případů, ale nemá zaručenou asymptotickou složitost. Je známo, že medián lze nalézt na $3n$ porovnání a že dolní odhad počtu porovnání je $2n$.

Hledání mediánu technikou rozděl a panuj (algoritmus FIND)

Tento algoritmus používá techniku "rozděl a panuj". chová se podobně jako **QUICKSORT** a hledá obecně k-té nejmenší číslo:

1. Vybrat pivota a rozdělit posloupnost pomocí $n-1$ porovnání na 3 oddíly: čísla menší než pivot (a prvků), pivota samotného a čísla větší než pivot ($n-a-1$ prvků).
2. (a) Pokud je $a+1<k$, hledám rekurzivně $k-a+1$-tý prvek v $n-a-1$ prvcích
   (b) Pokud je $a+1=k$, vracím pivot
   (c) Pokud je $a+1>k$, hledám rekurzivně k-tý prvek v a prvcích

Rekurzivní vzorec $T(n)=T(n-1)+(n-1)$ v nejhorším případě, což dává $\Theta(n^2)$. Očekávaný čas odhadneme na $4n$ a dokážeme indukcí podle n z rekurzivního vzorce $T(n,i)=n+\frac{1}{n}\left(\sum_{k=1}^{i-1}T(n-k,i-k)+\sum_{k=i+1}^{n}T(k,i)\right)$.

Zaručeně lineární hledání mediánu (algoritmus SELECT)

Vylepšení, garantující dobrý výběr pivota a lineární složitost i v nejhorším čase:

1. Rozdělit posloupnost na pětice (poslední může být neúplná, mějme threshold např. $n=100$, pod kterým množinu přímo třídíme)
2. V každé pětici najít medián
3. Rekurzivně najít medián těchto mediánů
4. Použít ho jako pivot pro dělení celé posloupnosti, protože je větší než alespoň 3 prvky z alespoň 1/2 pětic (až na zakrouhlení), je větší než alespoň $1/2\cdot 3/5=3/10$ prvků (a také menší než alespoň 3/10 prvků)
5. Z toho mám vždy zaručeno, že zahodím alespoň 3/10 prvků pro následující krok.

Vzorec potom vychází: $T(n)=c\cdot\frac{n}{5}+T(\frac{n}{5})+(n-1)+T(\frac{7}{10}n)$ a podle Master Theoremu nebo substituční metodou se dá vyčíslit jako $T(n)=\Theta(n)$.

Kapitola 15

Státnice I3: Závislostní syntax

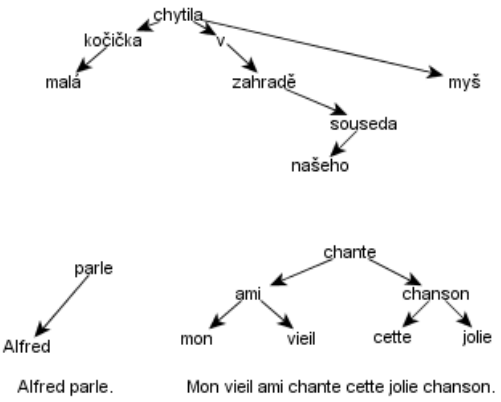
15.1 Úvod

Závislostní syntax je způsob popisu větné struktury, formálně zpracovaný L. Tesnièrem (dílo *Eléments de syntaxe structurale* vydáno 1959 posmrtně) v rámci tradice evropské strukturální lingvistiky (Tesnière spolupracoval i Pražským lingvistickým kroužkem). Na češtinu byla aplikována v knize *Novočeské skladba* (1947) V. Šmilauera a dále rozvíjena ve většině novějších mluvnic češtiny i v rámci *Funkčního generativního popisu* (FGD) P. Sgalla a dalších. Závislostní syntax se uplatňuje i v ruské teorii Meaning-Text.

Základní myšlenkou je vztah *závislosti* mezi jednotlivými slovy, přičemž v centru stojí hlavní sloveso a na něm závisí všechny ostatní členy (valence). Subjekt už tedy nemá stejné výsadní postavení jako v tradiční školské gramatice. Závislosti mezi jednotlivými slovy pak větu přirozeně uspořádají do stromové struktury.

Závislostní popis je vhodný i pro jazyky s volným slovosledem, na rozdíl od popisu pomocí bezprostředních složek, který na slovosledu do značné míry závisí.

15.2 Závislostní strom



Obrázek 15.1: Závislostní stromy uspořádané podle slovosledného pořadí (nahoře) nebo podle závislostí (dole)

Závislostní strom se formálně definuje jako pětice $T = \langle N, Q, E, WO, L \rangle$, kde:

- (N, E) je orientovaný graf (strom, tj. souvislý a bez kružnic, každý uzel kromě kořene má právě jednoho otce),
- Q je množina možných ohodnocení uzlů (gramatické kategorie),
- $WO \subset N \times N$ je silné úplné uspořádání (určující pořadí slov)
- $L : N \to Q$ je ohodnocovací funkce)

Takto vytvořený závislostní strom má přesně tolik vrcholů, kolik je slov ve větě (na rozdíl od složkového stromu, kde slova ve větě představují jen listy). Nedává ale žádnou informaci o tom, jak byla věta vytvořena – zaměřuje se čistě na vztahy mezi jednotlivými členy.

Topologické (lineární) uspořádání uzlů může být řešeno tak, jak naznačeno ve formální definici, ale může být provedeno jen čistě na základě struktury, jak to bylo v původním Tesnièrově popisu (viz obrázek).

15.3 Vztahy v závislostní syntaxi

Mezi slovy ve větě můžou existovat dva základní vztahy:

- *Závislost* (determinace) – jedno slovo nějakým způsobem (významově) určuje druhé.
- Slova jsou na stejné úrovni, pak se jedná o *koordinaci* (několikanásobný větný člen), *apozici* (přístavek) nebo *parentezi* (vsuvku).

Základní závislostní strom zachycuje pouze první z nich, pro ostatní je třeba vytvořit speciální formalismus.

Závislost se typicky vyjadřuje *hypotaxí* (podřadností) a koordinační a podobné vztahy *parataxí* (souřadností). Existují ale i příklady, kde je to naopak:

`Nechoď ven, nastydneš.` (determinace paratakticky), `Otec s matkou šli` (koordinace hypotakticky)

Závislost

Vztahy závislosti mezi slovy ve větě se formálně vyjadřují:

- *kongruencí* – shodou gramatických kategorií
- *rekcí* – určení gramatických kategorií nadřízeným členem (např. sloveso vyžaduje urč. pád svých doplnění)
- *juxtapozicí* – přimykáním (tj. závislý člen se prostě nachází poblíž nadřízeného ve slovosledu)

Závislosti mezi slovy ve větě je možné analyzovat na základě *principu redukce*: postupně se snažíme větu redukovat a vypouštíme slova, jejichž vynecháním zachováme gramatickou korektnost věty. Slova, která mohou být vypuštěna v libovolném pořadí, na sobě nezávisí. Rozlišujeme dva druhy závislostí:

- *endocentrická* – je jednoznačné, co závisí na čem, pořadí možného vypouštění je jasně dáno
- *exocentrická* – nelze jednoznačně určit, co závisí na čem (např. v rámci předložkových skupin, doplnění vyžadovaných slovesem)

U *exocentrických* závislostí se řídící a závislý člen určují na základě analogií s jinými případy (tj. sloveso je řídící člen, i když bez některých doplnění nemůže stát samostatně). Funkci a nutnost výskytu jednotlivých závislostí blíže popisuje teorie valence.

Některé závislosti se ve větě nedají analyzovat jednoznačně, věta má pak více interpretací. Jde zejména o tyto situace:

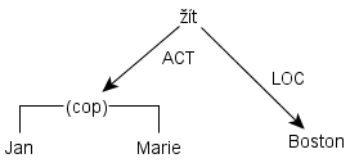
- tzv. "PP-attachment" (platí i pro složkové stromy, kde existuje více stromů odpovídajících stejné větě) a nejednoznačná doplnění adjektivem:

Př. `Ředitel banky roku`

- vyjádření užšího nebo volnějšího vztahu v některých konstrukcích (což nelze závislostním stromem, na rozdíl od složkového, popsat):

Př. `Profesor zjistil, že je jeho (hladový algoritmus) nefunkční, (Zítřejší noviny) ze včerejška`

Koordinace



Obrázek 15.2: Formalizace koordinace

Sémantický vztah *koordinace* (přiřaďování) je druh "zmnožení", obsazení jedné větné pozice více členy. Členy v tomto případě označují *různé* entity (větné členy nebo věty, i spojení větného členu s větou), které zastávají stejnou sémantickou roli. Jsou rovnocenné a samostatné, mají stejnou syntaktickou platnost. Do závislostí jdou jako celek, mají stejný řídící člen.

Podřadné a souřadné spojky se rozlišují podle pozice a přízvuku:

Př. `neboť` a `protože` se svojí souřadností / podřadností liší podle různých, spíš historicky daných kritérií (syntakticky jediný rozdíl: `neboť` nestojí nikdy na zač. souvětí).

Mezi členy koordinace nastává některý z následujících vztahů:

- *kopulativní* (CONJ – `a`)
- *adverzativní* (ADVS – `ale`)
- *disjunktivní* (DISJ – `nebo`)
- *gradační*: stoupá důležitost (GRAD – `nejen, ale i`) – v některých jiných lingv. tradicích se nerozlišuje
- *příčiný* (REAS – `neboť`), *důsledkový* (CSQ – `a tak`)
- oprava (`spíše, lépe`), zahrnutí (`a to i`)

Formalizace koordinace je složitá, je nutné přidat do závislostního stromu "další dimenzi". Petkevič (1995) navrhl pro FGD použití dvou různých typů stromových hran pro každý ze vztahů, popř. v lineárním zápisu dva typy závorek. Všechny hrany jedné koordinace jsou navázány na jeden závislostní uzel (viz obrázek).

Apozice

Apozice je také zmnožení, kdy více větných členů má tutéž syntaktickou platnost. V tomto případě ale všechny pojmenovávají jeden referent. Jsou navzájem zaměnitelné a gramaticky kongruentní. Na existenci apozice se shodne většina popisů, jsou ale různá pojetí – např. Šmilauer považuje za apozici i výrazy `Pan Novák`, `Prezident Klaus`, kdežto v PDT je to přívlastek. Někdy se rozlišuje i zda je výraz oddělený čárkou, nebo uvedený v závorkách:

Př. `"Obč. dem. strana (ODS) ..."`.

Apozici je taky občas problém odlišit od koordinace:

Př. `"Naši sousedé, Marie a Milan, ..."`

Formálně je možné ji zachytit stejně jako koordinaci.

Parenteze

Parenteze je vsuvka – věta nebo větný člen, jenž syntakticky nesouvisí s okolím, ale snaží se upřesnit, o čem se v okolní větě mluví. Typicky se zapisuje v závorkách, případně oddělená čárkou:

Př. `Mohl byste, prosím, přijít?`

Některé výrazy, jako např. `prosím, řekl bych`, se považují za *ustálenou parentezi* – de facto jde o částice nebo frazémy.

15.4 Projektivita

Uvažujeme závislostní stromy s uspořádanými vrcholy (např. podle slovosledu). Strom nad danou větou je *projektivní*, pokud neobsahuje žádnou *neprojektivní závislost*, tj. závislost mezi dvěma slovy oddělenými ve větě třetím slovem, které (ani nepřímo) nezávisí na žádném z nich. Pokud strom takovou závislost obsahuje, nazývá se *neprojektivní*.

Definice

Formálně definujeme *pokrytí uzlu* v závislostním stromě $Cov(u), u \in T$ jako množinu všech indexů vrcholů (na základě úplného uspořádání), do kterých z uzlu u vede (orientovaná) cesta, tedy jsou na uzlu u přímo či nepřímo závislé. Do této množiny se počítá i samotný uzel u. Pro kořen platí $Cov(r) = \{1, \ldots |N|\}$.

Řekneme, že *pokrytí uzlu* $Cov(u) = \{i_1, \ldots i_k\}; i_1 < \cdots < i_k; i_1, \ldots, i_k \in \{1, \ldots, |N|\}$ obsahuje *díru*, pokud existuje dvojice indexů vrcholů $(i_j, i_{j+1}) \in Cov(u), j \in \{1, \ldots k-1\}$ taková, že $i_{j+1} - i_j > 1$ (tj. pokrytí uzlu není souvislá řada indexů).

Potom pokud ve stromě existuje uzel, jehož pokrytí obsahuje díru, nazývá se strom *neprojektivní*.

Ekvivalentní definice projektivity říká, že jsou-li dva uzly u a v spojeny hranou a u leží nalevo od v (tj. má nižší pořadí ve slovosledném uspořádání), pak všechny uzly ležící nalevo od u a napravo od v jsou spojeny s kořenem cestou, která prochází jedním z vrcholů u, v.

Vlastnosti

Neprojektivní věty (tj. věty, jejichž syntaktickou analýzou je neprojektivní strom) jsou v některých jazycích spíše výjimkou (např. v angličtině), v češtině jsou naprosto běžné:

Př. `Karla jsme chtěli poslat do Francie. Soubor se nepodařilo otevřít.`

Př. `I saw a man with a dog yesterday which was a yorkshire terrier.`

Neprojektivní konstrukce nelze zobrazit souvislým složkovým stromem. V závislostním stromě to možné je. Ve FGD se s neprojektivitou počítá na úrovni povrchové syntaxe, ale už ne na tektogramatické rovině. Pořadí uzlů v tektogramatickém stromě totiž neodpovídá pořadí podle slovosledu.

15.5 Valence

Valence je vlastnost lexikálních jednotek (slov), která drží mnoho závislostních vztahů pohromadě. Jedná se o schopnost slov (prototypicky sloves, ale v mnoha teoriích se valence přiznává i jiným slovním druhům) vázat na sebe jiná slova, čímž vzniká větná struktura. Popisoval ji už Tesnière a přijímají ji v podstatě všechny závislostní teorie. Souvisí s principem redukce – jednotky, jejichž pozice se díky danému slovu otevírají, považujeme za závislé.

Valence se snaží popsat situaci, která se nám vybaví v souvislosti s daným slovem – zachytit, které *sémantické participanty* mají být přítomny (termín *sémantický participant* se nedá dobře definovat, ale všichni si v praxi představí to samé), tedy popsat počet a povahu *argumentů*, které na sebe slovo váže.

Valence pomáhá rozlišit nejednoznačnosti v:

- morfologii: `Ptala se jeho bratra` (Gen. nebo Acc.?)
- syntaxi: `Začala ho milovat. Nechala ho spát` (na čem závisí `ho`?)
- významech slova: `odpovídat na / za / čemu`
- sémantice větných členů: `sháněl se po ... / přišel po ...`

Je důležitá pro zpracování přirozeného jazyka, hlavně pravidlovými metodami (ve statistických je přítomna implicitně), učení se jazyku a lingvistický výzkum.

Pojetí valence se v jednotlivých teoriích liší. Jde už o rovinu lingvistického popisu, na které se uvažuje – pro FGD je to tektogramatická rovina, ale jiné teorie (německé, Meaning-Text) zahrnují valenci i do povrchové syntaxe. Týká se to i druhu vztahů, které jsou do valence zahrnuty – ve FGD valence souhlasí s hranami závislostních stromů, teorie Meaning-Text chápe jako valenci i např. to, že adjektivum vyžaduje substantivum, které by rozvíjelo ("pasivní valence").

Popisy valenčních rolí, tj. ohodnocení jednotlivých valenčních závislostí, se taky značně liší napříč teoriemi. Některé používají velice jemné sémantické rozdělení (FrameNet), jiné hrubší (německé teorie, Case Grammar a podobně i θ-role v Chomského teorii), jiné používají kombinace syntaktické a sémantické klasifikace (FGD, Meaning-Text). Některé teorie, jako např. původní Tesnièrova syntax, nebo z části PropBank/NomBank, dokonce doplnění pouze číslují.

Mnoho popisů valence se ale shodne na základním rozdělení dvou druhů závislostí, a to:

- *Aktanty* (*valenční argumenty*, *vnitřní doplnění*, *participanty*) – role, které mohou (nebo musí) být pro určitý řídící prvek obsazeny pouze jednou, navíc je jejich množina pro danou lexikální jednotku typická, tj. dají se vyjmenovat.
- *Volná doplnění* (*adverbiální modifikátory*, *circumstantials*) – role, které se mohou opakovat i několikrát a navíc určité doplnění se může vyskytovat s libovolným řídícím slovem.

Např. v Case Grammar se ale takovéto rozdělení neprovádí.

Typicky se taky u jednotlivých doplnění rozhoduje o jejich:

- sémantické *obligatornosti nebo fakultativnosti* – nějaké doplnění musí být buď vyjádřeno, nebo známo z kontextu (pozná se *dialogovým testem*)
- syntaktické *vypustitelnosti nebo nevypustitelnosti* – je-li nějaké nevypustitelné doplnění nevyjádřené, věta není gramaticky korektní

Typicky jsou aktanty obligatorní a volná doplnění fakultativní, ale není to jediný možný případ:

Př. `Jan se chová hezky.` (obligatorní, nevypustitelné volné doplnění), `vyrábět něco (z něčeho)` (fakultativní aktant)

Tato konkrétní terminologie pochází z FGD, ale podobné koncepty se vyskytují i jinde.

Valenci slov nelze zachytit pravidly, uchovává se *ve slovníku*. Takový slovník už zachycuje hodně syntaktických informací.

Kapitola 16

Státnice I3: Syntax bezprostředních složek a frázové gramatiky

16.1 Frázové gramatiky

Bezprostřední složky

V Americké lingvistice se od 30. let minulého století prosazoval hlavně deskriptivismus, jehož hlavním představitelem byl L. Bloomfield. V něm šlo hlavně o klasifikaci a statistický popis distribuce prvků ve větě a jejich vztahů. Na základě distribucí odlišovali syntaktické

- *endocentrické konstrukce*, kde výskyt jedné z částí samostatně má stejnou distribuci jako celek, a
- *exocentrické konstrukce*, kde výskyt pouze některých částí není možný.

V syntaxi se uplatňuje strukturální přístup, konkrétně analýza bezprostředních složek, kdy věta dostává stromovou strukturu: celá věta se postupně dělí na složky, přičemž vždy nadřazená složka dominuje své podřízené (viz níže). Pro dělení do složek ale Bloomfield spoléhal na intuici a nezaváděl žádná pravidla. Navíc převažovala binární kombinace a dekompozice, takže bylo trochu složité najít závislost a stromy měly hodně „úrovní".

Frázové stromy

V teorii bezprostředních složek, stejně jako v nejrůznějších verzích Chomského transformační gramatiky, je věta reprezentovaná frázovým stromem, někdy označovaným jako *frázový ukazatel*. Uzly tohoto stromu odpovídají složkám – frázím: každý uzel odpovídá určitému *souvislému* úseku věty. Vztah mezi uzly na vyšší a nižší úrovni je *dominance* – úsek věty příslušící složce na vyšší úrovni je složením úseků příslušících složkám na nižší úrovni. V generativní gramatice lze takový strom chápat tak, že uzel na nižší úrovni vznikl použitím přepisovacího pravidla z uzlu na vyšší úrovni.

Formálně: *frázový* neboli *složkový strom* je pětice $T = \langle N, Q, D, P, L \rangle$, kde:

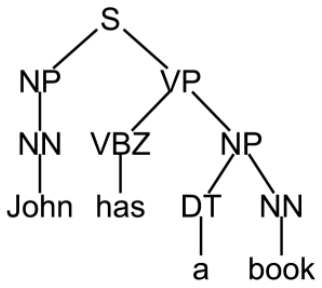
- N je množina uzlů,
- Q je množina ohodnocení uzlů (gramatických kategorií),
- $D \subset N \times N$ je relace dominance,
- $P \subset N \times N$ je relace precedence (ostré částečné uspořádání, zaručující slovosled) a
- $L : N \to Q$ je ohodnocovací funkce (přiřazení gram. kategorií uzlům)

A navíc platí následující podmínky:

- existuje jediný kořen stromu (daný relací dominance)
- relace D, P jsou antisymetrické
- strom splňuje podmínku *projektivity* (složky se nesmí částečně překrývat)

Složkový strom lze zapisovat buď jako stromové schéma (viz obrázek, možná je i varianta s kořenem nahoře), nebo pomocí "labeled bracketingu":

$[_S\ [_{NP}\ [_{NN}$ `John]]` $[_{VP}\ [_{VBZ}$ `has]` $[_{NP}\ [_{DT}$ `a]` $[_{NN}$ `book]]]]`



Obrázek 16.1: Frázový strom

Kvůli podmínce projektivity není možné zobrazit neprojektivní konstrukce, kde by se složky částečně překrývaly – docházelo by ke křížení frázových hran vzhledem ke slovosledu a nešlo by nalézt správné uzávorkování. Příklad:

```
I saw a man with a dog yesterday, which was a yorkshire terrier. Vánoční nadešel čas. Soubor se nepodařil
    otevřít.
```

zápis: strom (obrácený nebo normální) / „labelled bracketing“

16.2 Začátky Transformační gramatiky

Chomský vycházel z tradice amerického deskriptivismu a frázových gramatik, ale přidával myšlenku, že nestačí jen jazykový jev pozorovat a popsat, že je třeba ho vysvětlit. Navíc akceptuje staré teorie *univerzální gramatiky*, tj. vrozenosti části jazykové kompetence, která je společná pro všechny jazyky. Idea transformací, uvedená deskriptivistou Z. Harrisem, byla pro spojení všech myšlenek velice vhodná.

Znalostí jazyka je pro Chomského znalost pravidel, použitých ke generování gramaticky korektních vět. Gramatičnost považuje za rozhodnutelnou, i když přiznává, že jde o idealizaci. Ve svých výzkumech spoléha na vlastní intuici, nesoustředí se na korpusy.

Jako cíl výzkumu nestanovuje poznání mentalních procesů tvorby vět, ale to, jak větu tvoří *gramatika*. Gramatika je podle něj nástroj, který z konečné množiny slov generuje nekonečný jazyk – díky rekurzivitě (např. nepřímá řeč, “pes jitrničku sežral” a pod.), o které věřil, že je společná všem přirozeným jazykům.

Syntaktické struktury (1957)

V této knize Chomsky popsal první verzi své teorie a její aplikaci na angličtině. Představuje několik druhů generativních gramatik (které později shrnul do své slavné hierarchie) a na protipříkladech ukazuje, že ani regulární gramatiky, ani bezkontextové gramatiky nejsou schopné samy o sobě popsat přirozený jazyk. Bezkontextové gramatiky odpovídají bezprostředním složkám, ale generativní přístup (proti deskriptivnímu) je nový.

Chomsky zavádí *transformační generativní* gramatiku. Obsahuje bezkontextovou generativní gramatiku, ale navíc přidává ještě jeden krok tvorby vět – *transformace*. Spolu s tím se přidává distinkce dvou úrovní:

- *hloubková struktura* (vytvářená generativní gramatikou)
- *povrchová struktura* (tvořená transformacemi z hloubkové struktury)

Gramatika pak sestává z následujících komponent:

- *báze*, sestávající z:
  - *lexicon* (seznam slov podle druhů příslušných neterminálů, např. tranzitiva, intranzitiva ...)
  - *phrase structure rules* (soubor pravidel ke generování vět, včetně doplnění terminálních symbolů, S –> VP NP / nepovinné části, bezkontextová pravidla)
- *transformational rules* – pravidla např. pro aktiv/pasiv, oznamovací větu a otázku atd. Vychází z toho, že některé struktury jsou úzce provázané a generativní pravidla pro ně by se duplikovala. Obsahují dvě podčásti:
  - Strukturní analýza (na jakou část se aplikují)
  - Strukturní změna (jak se provádějí)
- *fonologický komponent* – regulární pravidla, přiřazující zápisům věty fonetické reprezentace

Při tvorbě vět se tedy nejprve použijí frázová (báze, vznik kernelové věty) a pak transformační pravidla (transformace jsou povinné nebo nepovinné, dochází i ke změnám významu).

Chomsky věřil, že je možné vyrobit pravidla, jejichž výstup budou korektní (anglické) věty. V této verzi teorie úplně vynechal sémantiku (zřejmě pod vlivem deskriptivismu), což kritizovali jeho studenti (J. J. Katz, P. M. Postal). To vede k dalšímu vylepšování.

Standardní teorie (1965) a její rozšíření v 60.–70. letech

Další verze, tzv. Standard Theory, shrnutá v knize *Aspects of the Theory of Syntax* obsahuje několik vylepšení. Nejdůležitější z nich je zavedení sémantiky, přičemž se tu (poprvé) objevuje odkaz na teorie univerzální gramatiky – hloubková struktura jako *sémanticky interpretovaný* základ pro transformace.

Gramatika tu sestává z následujících částí:

- Syntaktická komponenta (obsahuje *bázi* a *transformační komponentu*)
- Sémantická komponenta (*lexikon* obsahující sémantické informace a *projekční pravidla*, která přiřazují syntaktickým stromům interpretaci)
- Fonologická komponenta

Generování vět potom vypadá následovně:

1. Báze vygeneruje *hloubkovou strukturu* (používá už nejenom frázová, ale i transformační pravidla, slovník obsahuje *syntaktické rysy*, podle kterých lze transformacemi doplňovat slova za neterminální symboly)
2. Sémantický komponent jí dodá významovou interpretaci
3. Transformační komponenta z hloubkové struktury vytvoří *povrchovou strukturu* (transformace v této verzi nemohou měnit význam)
4. Fonologická komponenta z ní vytvoří fonetickou intepretaci

Synonymní věty mají stejnou hloubkovou, ale jinou povrchovou strukturu, homonymní naopak. To odpovídá vztahu asymetrického dualismu, popisovanému v Pražské škole.

I tato verze teorie měla problémy, např. s větami jako:

„Many men read few books“ a „Few books are read by many men“

Kdy obě nemají stejný význam. Stejný problém se v češtině ukazuje u aktuálního členění věty:

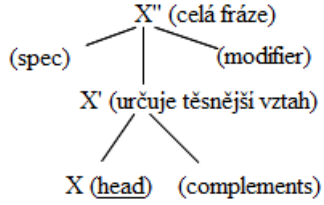
„Mnoho lidí čte málo knih“ / „Málo knih čte mnoho lidí“

Chomského studenti argumentovali, že vybíráme-li význam z hloubkové struktury, pak tato musí být „hlubší“ než je tu navržené. Rozmíšky okolo podobných problémů vedly k rozštěpení teorie. Vznikla tak Generativní sémantika (G. Lakoff, J. McCawley, J. R. Ross), která obsahuje hlubší hloubkovou strukturu a transformace v ní nemohou měnit význam.

Interpretativní sémantika (rozšířená standardní teorie)

Chomského reakce na Generativní sémantiku byla, že zachoval stejnou hloubkovou strukturu (dále i *d-struktura*), ale *sémantická interpretace* se provádí nejen z hloubkové, ale i z *povrchové struktury*. Transformace tak mohou měnit význam. Odůvodňoval to hlavně distinkcí mezi *presupozicí* a *fokusem*, tj. zachycením aktuálního členění.

X-bar



Obrázek 16.2: X-bar: základní struktura

X'' -> (spec) X' / X' (spec)
X' -> X (compl ...) / (compl ...) X

X' -> (modif) X' / X' (modif)
X'' -> (modif) X'' / X'' (modif)
X' -> X' (conj) X', X'' -> X'' (conj) X''

Obrázek 16.3: X-bar: přepisovací pravidla pro základní strukturu

Na základě výzkumu nominalizací byla jako rozšíření standardní teorie uvedena teorie X-bar (podle značení $\bar{X}$, které se ale často zjednodušovalo na X'), vycházející z tzv. *lexikalistické hypotézy*, která tvrdí, že nominalizace není transformací. Naopak identifikuje univerzální strukturu frází (libovolných typů – S, PP, NP, VP), díky níž nominalizace odpovídají větám. Ta má 2-3 patra: "bez pruhů" (nejnižší), "1 pruh", "2 pruhy" (nejvyšší), která obsahují následující prvky:

- *phrase* (*maximal projection*): celá fráze (X'' , XP)
- *head*: na nejnižší úrovni, charakteristický prvek fráze (později „governing element"), předp., že jenom 4 kategorie slov můžou být hlava (N, V, Prep, Adj)
- *specifier*: doplněk fráze na vyšší úrovni, pre- nebo post-order (např. pro NP je to v angličtině člen)
- *modifier (adjunct)*: volný doplněk X'' / X' , může jich být libovolný počet, v pre- nebo post-orderu – způsobuje rekurzi
- *complement (argument)*: doplněk hlavy na nejnižší úrovni (povinný, úzký vztah), může jich být lib. počet (i nula), mohou stát před nebo za X

Dvě různé úrovně pro *complement* a *modifier* jsou vlastně prostředek k popsání blízkosti vztahu (complement x modifier), což se dá chápat jako přiblížení se závislostnímu popisu. Navíc existují některé drobné odchylky pro úroveň věty a pro PP, celek ale funguje jako univerzální schéma pro celé věty. Kvůli členům, předložkám atp. ale vychází dost složitě.

Podmínky pro transformační pravidla a stopy

Pro omezení síly transformací, které se ukázaly být příliš obecné, byly přidány podmínky na jejich aplikaci – např. omezení jejiho rozsahu, později nazvané *bounding*, a zavedení *traces (stop)*, což jsou v případě přemístění prvků transformacemi prázdné kategorie na původních místech, svázané s přemístěným prvkem. Při přechodu do fonetické reprezentace se uplatňují další pravidla a povrchové filtry, které částečně nahrazují podmínky na transformační pravidla.

Sémantická interpretace nyní probíhá přímo z povrchové struktury a to přes *logickou strukturu*. Odvození významu ze samotného povrchu ale nestačí a je nutné uchovávat další informace, proto byla v první polovině 70. let zavedena místo povrchové struktury tzv. *decorated surface* / *shallow structure* (*s-struktura*). Může obsahovat např. údaje o tom, že „sloveso je tranzitivní", nebo stopy po transformacích.

16.3 Teorie Principles and Parameters / Government and Binding

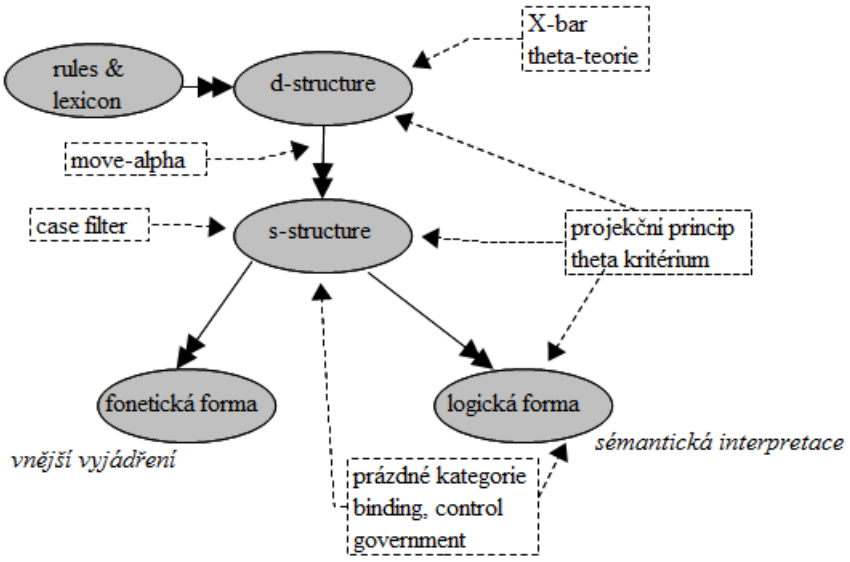
Tato teorie, popsaná poprvé v knize *Lectures on Government and Binding* (1981) se vyvinula ze Standardní teorie během 80. let. Je vedena snahou o lepší vysvětlení jazyka a rozvíjením myšlenek univerzální gramatiky. Chomsky zastával názor, že je vrozená nějaká jazyková schopnost, která je lidem společná (*principy*) a něco je naučené (*parametry*) – to dohromady dává schopnost mluvit mateřským jazykem.

Př. princip – „slova se dají přemísťovat", parametr – slovosled konkrétního jazyka (constraint)

Př. princip – "věta musí mít subjekt", parametr — "vyjadřuje se subjekt povinně vždy?"

Jedním z postulovaných principů je strukturálnost jazyka. Chomsky dále trvdí, že struktura jazyka je právě X-bar. Základní schéma generování vět zůstává stále stejné, s několika vylepšeními. Mezi základní principy patří:

- *Theta-role* a *cases*
- Vztahy *government*, *binding*, *bounding* a *control*

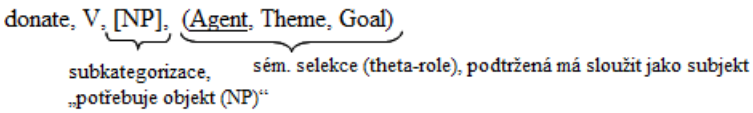


Obrázek 16.4: Schéma generování věty v teorii Government and Binding

Projection principle a theta-role

Tyto vlastnosti byly do gramatiky přidány, aby ve výsledku lépe odrážela lexikální vlastnosti jednotlivých slov.

Projection principle ve své podstatě tvrdí, že veškeré informace z lexikonu je třeba zachovávat během celého generování. Až do jeho zavedení byly lexikální informace při generování ignorovány. Formálně: reprezentace na obou rovinách splňují *subkategorizace* lexikálních jednotek, přičemž *subkategorizace* je omezení komplementů podle lexikální realizace hlavy fráze (např. kick vyžaduje NP, think povoluje celé S' (klauzi), discuss taky jen NP) . Pro zachování informací v povrchové struktuře se opět používá "decorated surface".



Obrázek 16.5: Theta-teorie – heslo v lexikonu

Subkategorizace se vztahují jak na gramatiku, tak na sémantiku, kde se nazývají *theta-role* (*θ-role*, *tématické role*). Každý argument (komplement) prvku (na každé úrovni) musí mít přiřazenu (právě jednu) theta-roli; každá theta-role je přiřazena právě jednomu argumentu (až na koordinaci). Řekneme, že A *theta-označuje* B , když subkategorizuje pozici, na které se nachází B (a to může být buď v rámci fráze, jíž je A hlavou, nebo mimo ni (např. sloveso pro subjekt). Theta-role jsou svým způsobem navázání na teorii Cases C. Fillmora, z níž vychází idea *pádového filtru (case filter)* (každé NP musí mít přiřazený pád), který Chomský s theta-teorií spojuje.

Heslo v lexikonu tedy obsahuje dva druhy informací – syntaktické (subkategorizace) a sémantické (theta-role), jak je vidět z obrázku.

Command, government a binding

Vztahy command, government a binding spojují fráze ještě dalšími vztahy, než je dominance původních bezprostředních složek, takže umožňují složitější závislosti. Jejich základem je vztah *command*, ostatní dva jsou na něm postavené. Tento vztah umožňuje spojit i fráze, které na sebe přímo nenavazují. Existují dvě varianty:

- *c-command* – A *c-command*uje B , když A nedominuje B a každý větvící se uzel nadřízený A je také nadřízený B
- *m-command* – zmírňuje podmínku na každou *maximální projekci* X'' .

Vztah *government* nastává, když:

- A je hlava fráze
- A *m-command*uje B
- mezi A a B není žádná *bariéra* (zpravidla *maximální projekce*)

*Govern*ující uzel potom určuje *case* (*theta-roli*) uzlů, které *govern*uje.

Př: Lze např. popsat, že sloveso „seem“ může být komplementováno slovy „as“, „to be“ – a to může procházet napříc větou.

Z transformační komponenty, tj. z přesouvání frází a *traces* byl v nové verzi teorie vytvořen obecný princip *move-α*. Traces zachovávají theta-role (*co-indexing*) a cíle přesunu jsou omezené (jen subjekt, pozice *adjunktu*). Navíc podle teorie *bounding (ohraničení)* nelze přesouvat přes víc než jeden *boundary node* (ty jsou různé pro různé jazyky, v Angličtině S, NP):

„the man who$_i$ [$_S$ I think [$_{S'}$ that [$_S$ you said [$_{S'}$ that [$_S$ you had seen e$_i$]]]]]“ – OK

„the man who$_i$ [$_S$ I identified [$_{\mathbf{NP}}$the dog [$_{S'}$ which$_j$ [$_S$ e$_j$ bit e$_i$]]]]“ – nepřejde přes NP, tedy nekorektní!

Navíc se aplikuje *pádový filtr*, který vyřadí taková NP, které nejsou prázdná a nemají přiřazenou theta-roli (pád). Vztah *binding* popisuje anaforicitu a reflexivitu, ale i např. subjekt infinitivu apod. A *bind*uje B , když:

- A "governuje B
- A, B jsou *co-index*ovány (tj. odkazují na to samé, jsou v anaforickém vztahu)

Existuje několik poddruhů tohoto vztahu, na které jsou další restrikce. Na vztahu *binding* je pak založená teorie *control*, která (slovníkově) klasifikuje slovesa, jež mají závislou infinitivní klauzi, podle toho, zda vyžadují, aby nějaké jiné jejich doplnění bylo koreferenční s podmětem infinitivu.

`decide, promise` – anaforická relace se subjektem („`John decided to go`“ = `John goes`)

`persuade` – anaforická relace s hl. objektem („`John persuaded Max to go`“ = `Max goes`)

16.4 90. léta – Program Minimalismu

V 90. letech si Chomsky uvědomil, že teorie by měla nejen vysvětlovat, ale zároveň být ekonomická. Všechny teoretické nároky by měly být zachovány, i když se na povrchu neprojevují. Přišel tedy s novou teorií, která uvažuje jen dvě roviny popisu (*interface levels*):

- myšlenková doména, proces, objekt (*logická forma*) – rozhraní mezi jazykem a kognitivní oblastí
- fyzická forma (*fonetická interpretace*) – rozhraní mezi jazykem a fyzikální akustickou skutečností

Snaží se popsat jejich vztahy. Generování by mělo mít stejný cíl, tj. schéma teorie by mělo být interpretovatelné stejně jako dřív:

- Jsou použitá generativní pravidla. Znovu jde o generování, soustava filtrů ztratila důležitost – věta se skládá operacemi *merge* a *move*.
- Potom se v tzv. bodě *spell-out* oddělí cesty logické a fonetické interpretace – operace provedené do tohoto bodu jsou viditelné v obou strukturách, pozdější jsou už na sobě nezávislé.

Nová teorie je dost rozdrobená, navíc nový typ pravidel ve skutečnosti výsledné struktury dost roztahuje. Nové práce vycházejí v časopise „Linguistic inquiry“.

16.5 Další teorie s frázovou gramatikou

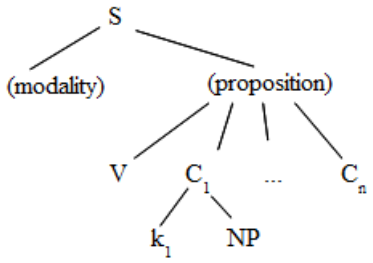
Cases

Tato teorie byla představena C. Fillmorem v roce 1968 jako reakce na to, že Chomsky zanedbává ve své původní teorii sémantiku. Vycházel z názoru, že hloubková struktura má být hlubší a obsahovat celý význam věty. Je ve své podstatě mnohem bližší evropskému závislostnímu přístupu, a ač používá frázovou gramatiku, uvažuje i o závislostním přístupu:

- *struktura* věty: bez přepisovacích pravidel, skládá se z *modality* a *propozice*
  - *case*: pád, ale jde hlavně o význam, sémantickou funkci (*deep case*)

Př. *agentiv, lokativ* (`Chicago is windy`), *instrumentál, objektiv, dativ* (`John believed ... = experienced a belief`)

- *propozice*: „jádro věty“ – obsahuje sloveso (přísudek) a case phrases (tolik, kolik je potřeba – není binární), není VP, tedy jde rovnou o sloveso a jeho komplementy, z nichž každý má přiřazen *case*
- *modalita*: negace, modální sloveso, vid, čas



Obrázek 16.6: Větná struktura v teorii C. Fillmora, naznačená je jedna z *case phrases* s *case feature* a nominální frází

- *case phrases*: jsou přímo vázané na sloveso, obsahují *case feature (pádový příznak)* a NP (které mají stejnou strukturu jako X-bar, to pro Fillmora nebylo důležité)
- *case frames*: sloty slovesa pro vázání cases (něco jako valence), některé jsou povinné, některé ne
  - existují závislosti – někt. cases musí být přítomny, když jsou přítomny jiné

Př. `break ((Ag) Instr) Obj)` – je-li agent, je nutný instrument („John rozbil okno“ – rozumí se „něčím“), je-li instrument, agent není nutný („Větev rozbila okno“)

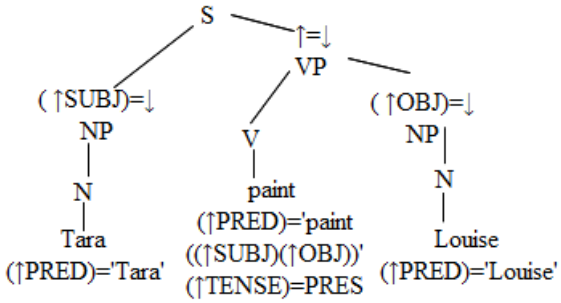
Lexikálně funkční gramatika

Tuto teorii vytvořili Joan Bresnanová a Ron Kaplan na přelomu 70. a 80. let. Zdůrazňuje funkci lexikonu, v němž každé slovo má udaný *slovní druh, potřebné fráze a tématické role* (explicitně zdůrazněno u sloves, ale ostatní sl. druhy to mohou používat taky). Není založena čistě na frázové struktuře, nemá transformace a projekční princip. Povrchová a hloubková struktura jsou různé:

- složková (povrchová) struktura (*c-structure*): typu X-bar
- funkční struktura (*f-structure*): jiná, z ní se odvozuje význam – předpokládá se tu podobnost mezi různými jazyky

Byla široce přijata i v Evropě, přinesla první detailní zpracování lexikonu. Předpokládá, že věty generuje mluvčí, ne gramatika, snaží se o realistický model – aby se přiblížila tomu, co opravdu lidé provádějí při tvorbě vět.

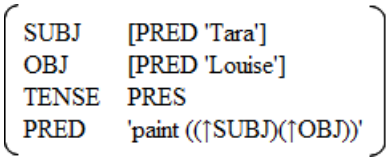
Složková (povrchová) struktura



Obrázek 16.7: LFG: Složková struktura

- přejímá X-bar, opět fráze a podobná pravidla jako u Chomského.
- *exocentrismus*: S' (klauze) nemá žádný řídící člen, jako řídící člen této složky mohou vystupovat různé „funkční řídící členy“ pro různé jazyky
- s každým frázovým pravidlem musí zůstat indikace, jak přejít na funkční strukturu (tohle bylo u Chomského dáno projekčním principem), každé pravidlo má proto *funkční anotaci (ohodnocení)*
- použití šipek: ↑ — odkazuje na „mateřský“ uzel (nadřízený) ↓- odkazuje na tento uzel
  př. pro NP: (↑SUBJ)=↓ „subjekt nadřízeného uzlu je tento“, VP: ↑=↓ „je funkčním řídícím uzlem (obsahuje (nějakou) přímou funkční informaci o nadřízeném uzlu)“ – toto mají všechny preterminály (tj. těsně nad konkr. slovem – N, P, V)
- *lexikon*: slouží k přenosu funkční informace z lex. jednotek do funkční struktury spojené se složkovou

Funkční struktura a její vlastnosti

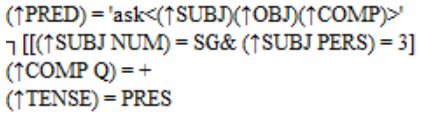


Obrázek 16.8: LFG: Funkční struktura

- formálně: množina dvojic rys/atribut, hodnota
- 3 druhy hodnot: symbol (PRES, SG ...), sémantická forma ('paint((↑SUBJ)(↑OBJ))'), vnořená f-struktura
- podmínky správnosti – každý atribut má mít max. jednu hodnotu, struktura musí obsahovat všechny funkce řízené predikátem, každá řiditelná funkce musí být řízena nějakým predikátem
- přechod od složkové struktury: každá část složk. struktury má svoji část f-struktury, skládání pomocí *unifikací*

Př. `[NUM SG] + [PERS 3] = [NUM SG, PERS 3]` — OK, `[NUM SG] + [NUM PL]` nelze

- *subkategorizace lexikonu*: podle gramatických funkcí (OBJ, SUBJ, ...), ne podle kategorií (NP, PP); *predikátově-argumentová struktura* (každý argument svázán s gramatickou funkcí, pro daný predikát realizován jen 1x)
- *gramatické funkce*: staví na Tesnièrově valenční teorii – aktanty a volná doplnění



Obrázek 16.9: LFG: Funkční struktura – gramatické funkce

- *lexikální pravidla*: úpravy lex. jednotek (výsledná lex. jednotka může mít i jinou synt. strukturu), často definováno pro celou třídu lex. jednotek

Př.: pasiv: `(SUBJ) -> 0 / (OBL`$_{AG}$`), (OBJ) -> (SUBJ)`, po aplikaci: `(PRED)='eat<(OBL`$_{AG}$`),(SUBJ)>'`

- Spousta dalších detailů, dost podrobné

16.6 Poznámky

- „generativní gramatika“ neznamená jen „transformační“, pojem se vztahuje i na Bresnanové teorii, která transformace odmítá
- další trendy: head phrase structure grammar, TAG theory (USA, Joshy, blízké závislostní, ale také frázová struktura) a další, lokálnější (něco viz PFL012-poznámky)

Kapitola 17

Státnice I3: Základy obecné lingvistiky

17.1 Typologie jazyků

- **typ jazyka** — *souhrn rysů, které se navzájem podmiňují a dávají mluvnici (gramatice) určitý ráz* (Skalička)
- současná strukturní typologie je založena převážně na morfologii
- deklinace je *výkladní skříňí typologie* (Sgall)
- pozn.: typy jazyků jsou umělé — každý jazyk se od svého typu odlišuje
  - žádný jazyk není čistý typ, jazyky jsou popsány pomocí souborů rysů — převažující rysy

Historie

- August Schleicher
  - **Stammbaumtheorie** (1861)
  - existuje **prajazyk**, který se štěpí na nové jazyky
- Wilhelm von Humboldt
  - jazyk má **vnitřní formu** (v hlavě člověka) a **vnější formu** (používání jayzkových prostředků)
  - jazyk je tvořivá síla — pozorujeme pouze výsledek
- Georg von der Gabelentz
  - **zakladatel typologie**
  - jazyky se vyvíjejí ve *šroubovici* — spirále
    - izolace → aglutinace → flexe → izolace (vyšší typ)
    - s každým vývojovým cyklem se dostanou o úroveň výš
- Vladimír Skalička
  - *strukturní (morfologická) typologie*
  - *morfém* — skládá se z elementárních jednotek: **séma**

```
matka:    matk - a
            /     \
         kořen    koncovka
           |        |  |  |
        1 séma      N  sg f - 3 sémata
```

Dělení jazyků

- starší dělení: *analytické* a *syntetické*

Skaličkovo dělění

 izolační (analytické)

- angličnina, francouzština, ...
- hlavní rys: obsahují *pomocná slova*
- další rysy: skloňování, časování, množství pomocných slov (nutnost podmětu ve větě — *Il pleut.* (fr.), *It is raining.* (angl.) — *Prší*

 aglutinační

- maďarština, turečtina, finština, gruzínština, baskitština
- hlavní rys: obsahují *afixy* (*prefix* nebo *sufix* — předponu nebo příponu)
  - každé séma je vyjádřeno vlastním morfémem **vs.** dom-ům v češtine — *ům* vyjadřuje 2 sémata najednou: plurál a dativ
- *vokalická harmonie* — *Praga-ban* (v Praze) vs *Becs-ben* (ve Vídni) — *ban* se mění v *ben* kvůli samohlásce v kořenu slova

```
turečtina dům:          Sg.         Pl.
                  Nom.  ev          ev-ler
                  Gen.  ev-in       ev-ler-in
                  Dat.  ev-e        ev-ler-e
                  Akuz. ev-i        ev-ler-i
                  Lok.  ev-de       ev-ler-de
                  Instr ev-den      ev-ler-den
```

 flexivní (ohýbací)

- latina, staré indoevropské jayzky, slovanské jayzky
- pravidlo jedné koncovky — kumulace funkcí (viz příklad *matk-a*: *a* kumuluje 3 sémata)

 polysyntetický

- čínština, vietnamština, některé rysy má i němčina
- skládání slov — např. *Haupt-bahn-hof* (něm.)
- gramatické významy jsou vyjadřovány lexikálními prostředky
  - např. zvláštní slova používaná při vyjadřování počtu věcí v čínštině: *tři knihy -> tři svazek kniha* (čín.)

 introflexivní

- semitské, čadské, berberské jazyky
- změna hlásek uvnitř slova, typicky kmen je určen souhláskami a gramatické informace samohláskami
- arabština: *k-t-b*: kitab(un) — kutub(un) (kniha — knihy)
  - proměňují se samohlásky, souhlásky beze změny
- němčina: trinken — tränken, čeština: hoch — hoši

Příklady jednotlivých rysů

- izolace
  - čeština, ruština: budoucí čas nedokonavých sloves
  - ruština: stupňování adjektiv — *bolee važnyj*, *najbolee važnyj*
- aglutinace
  - čeština: *nej-vážn-ějš-í*
- introflexe
  - němčina: Vater — Väter
  - angličtina: tooth — teeth
  - čeština: doktor — doktoři, nízký — nižší
- polysyntéze
  - němčina: složená slova

Další typy dělění jazyků

- *syntaktická typologie*

 nominativní vs *ergativní* jazyk

- *ergativní* jazyk
  - kavkazské jazyky, kamčatské jazyky, baskitština
  - existence zvláštního pádu pro podmět tranzitivního slovesa = *ergativ*

```
Muž postavil dům - ergativ
Muž přichází - nominativ
```

slovosled

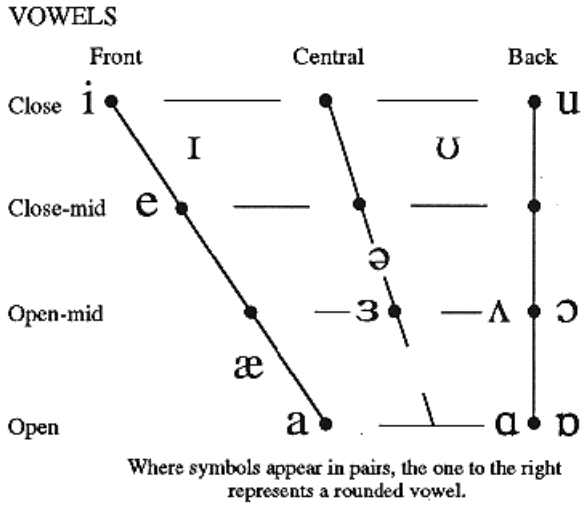
- SVO, SOV, VSO jazyky
  - pravděpodobnostní pozorování, které zkoumal již Jakobson ve 20. letech
  - SVO — germánské, románské, slovanské jazyky
  - SOV — turečtina, japonština, baskitština
  - VSO — hebrejština, welština
- *implikatury* (*implikační zákony*) — Jakobson
  - *má-li jazyk volný slovosled, má pádové koncovky*
  - *má-li jayzk pevný slovosled, má hodně pomocných slov*

Jazykové univerzálie

- významy termínu *typologie*:
  1. univerzálie — kde se bere univerzálnost jazyka?
  2. *clusters of properties* — svazky společných rysů
  3. klasifikace
  - bod 1 a 2 spolu úzce souvisí
- "absolutní" univerzálie
  - 2 třídy fonémů (hlásek) — vokály, konsonanty
- jazykové univerzálie
  - racionalismus (Descartes)
    - *karteziánská lingvistika — vrozené ideje*
    - vrozený mechanismus u dítěte pro produkování vět – navazuje generativní gramatika Chomského
  - sensualismus (Locke) — člověk je při zrození nepopsaný list papíru — *tabula rasa*
    - *nic není v rozumu, co dříve nebylo ve smyslech*

Hláskosloví

- lingvistika se do 50. let 20. století většinou zabývala hlavně *hláskoslovím*
- nové směry v lingvistice se začaly zabývat i syntaxí a sémantikou (lingvistickou či formální)



Obrázek 17.1: Hellwagův trojúhelník znázorňuje, jakým způsobem jsou samohlásky vyslovovány

Fonetika

- experimentální věda, která zkoumá, jak jsou tvořeny hlásky
- už ve staré Indii a klasickém Řecku (recitace, hudba); Jan Hus — výslovnost *y* a *i*
- přístroje na zkoumání zvuku: laryngoskop, rentgen, sonograf, magnetofon
- rozdělení hlásek
  - *samohlásky* (vokály)
    - přední, střední, zadní; vysoké, středové, nízké; zaokrouhlené (viz Hellwagův trojúhelník)
    - ústní/nosové, krátké/dlouhé
  - *souhlásky* (konsonanty)
    - mohou být charakterizovány pomocí šumů
    - metody zkoumání šumů:
      1. akustické — zkoumání šíření akustických vln (poslech, fonogram, sonogram, spektrogram);
      2. auditivní — sluchový dojem: explozívy, frikativy, sonory
      3. artikulační
         (a) způsob tvoření — podle existence *překážky*, která je v cestě výdechového proudu
         (b) místo tvoření

Fonologie

- zkoumá *funkci* jazykových hlásek
- **foném** — komplexní jednotka, která se skládá z *distinktivních rysů*
  - *ř*: *dři* (neznělé) vs. *tři* (znělé) — jeden foném s různými distinktivními rysy
  - čeština: 35 fonémů, z toho 10 samohláskových (Černý)
  - varianty 1 fonému v koncové pozici před rázem: snob x snop, Srb x srp
- *slabika*
  - základní (přirozená) jednotka
  - segmentace slov na slabiky: proud-it x pro-ud-it
- *suprasegmentální rysy*: melodie, intonace, přízvuk, mluvní akt
- *asimilace znělosti*: *progresivní* (shoda jako [schoda]), regresivní ([zhoda, zběr], v češtině typická)

Morfologie

- užší význam: teorie o tvoření tvarů
- širší význam: zabývá se odvozováním (nových) slov (ve flexivních a aglutinačních jazycích)
- morfologické kategorie jsou univerzální — jazyky se liší jejich repertoárem, počtem jejich členů
- **morfém**
  1. *lexikální* — kmen slova *matka*
  2. *gramatický* — realizace pomocí *morfu -ou* ve slově *matkou* — obsahuje 3 *sémata*
- **alomorf** — morfonologické podoby kmene: matk/matek/matc/matč
- slovní druhy — kritéria třídění: mofologická, syntaktická, sémantická
  - podstatné jméno (substantivum) — sémata: pád (různý počet pádů — arabština 3, maďarština 17, čeština 7), číslo (jednotné, množné, duál, triál, lexikální vyjádření čísla — čínština), rod
    - *determinace* — jazyky se členy (určité — neurčité)
  - sloveso (verbum)
    - gramatické kategorie:
      - osoba (1.,2.,3., inkluziv, exkluziv, plurál zdvořilostní: vykání, onikání, onkání), číslo, čas (absolutní, relativní), způsob (postoj mluvčího), rod slovesný (aktívum, pasívum, reflexívum), vid (perfektivní — dokonavý, imperfektivní — nedokonavý, iterativní)
    - nejasná hranice meyi tvaroslovím a slovotvorbou pro určité slovesné tvary: slovesná substantiva, gerundium, přechodník, participia, sloveská adjektiva
  - adjektiva — sémata: rod, číslo, pád
  - zájmena — deixe (*tento stůl*), anafora (odkaz na předcházející jméno), katafora (odkaz na následující jméno)
  - číslovky — druhy: základní, řadové, souborové, násobné, neurčité,...
  - příslovce — nejsou neohebná, protože je můžeme stupňovat
  - citoslovce
- **autosémantická** (významová) vs. **synsémantická** (pomocná — spojky, předložky, členy, částice) slova

17.2 Strukturní lingvistika

- heslo (strukturalismus) na Wikipedii
- navazuje na *mladogramatickou školu* (taky *Lipská škola*, mnoho historických gramatik indoevropských jazyků, asociativní psychologie, fyziologické změny mezi jazyky — *hláskový zákon*)
  - H. Paul, H. Osthoff, B. Delbrück, A Leskien, H. Bopp, J. Grimm, K. Brugmann
- *Ženevská škola* — Ferdinand de Saussure
  - předchůdci: H. Schuchard — historie *slov a věcí*, K. Vossler — *jazyk — nástroj ducha*, *kazaňská škola* — *foném* proti pojmu hláska, *moskevská škola* — zavedení pojmu *gramatická kategorie*
- hlavní rysy:
  1. vidět systém a systémové souvislosti
  2. synchronní pohled na jazyk
  3. formalizace popisu jazyka (metod popisu)
- roviny jazyka: langue (jazyk) — parole (mluva) — langage (řeč)
- sémiologie — relativní arbitrárnost znaku

```
označující - označované - hodnota
signifiant - signifié  - valeur
```

Strukturní školy

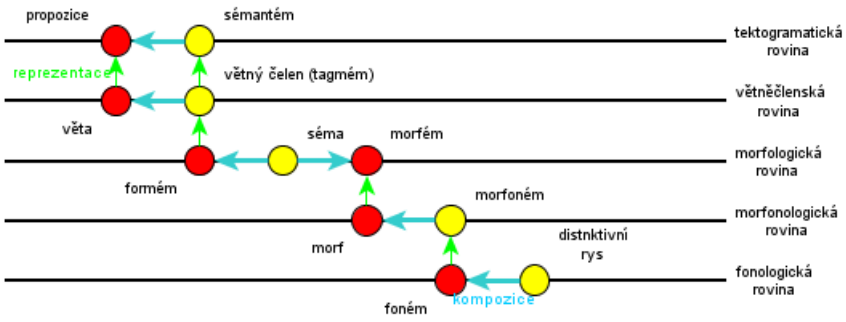
Americká strukturální škola

- **deskriptivní** — úkol fakta *registrovat, klasifikovat a popsat* — ne kauzální souvislosti
- L. Bloomfield, C. Hockett, Z. Harris
- jazyk (gramatika) má 3 hlavní systémy: fonologický, morfologický a jejich spojení mor(fo)fonologický
- zkoumání jevů pomocí *distribuce*, *komplementární distribuce* a *souvýskytu* (co-occurrence)
- *endocentrické konstrukce* vs. *exocentrické konstrukce*
- syntax — **bezprostřední složky**, později frázová gramatika (Chomsky)

Kodaňská strukturní škola

- Louis Hjelmslev
- **glosématika** — glossém: základní jednotka jazyka
- **jazyk** — síť abstraktních vztahů, deduktivní systém předpokladů a z nich vyvoditelných teorémů; lingvistická algebra
- sémiotika:
  1. vztah znaku k jiným znakům
  2. vztah znaku k tomu, co označuje
  3. reakce interpertátora (mluvčího, posluchače) na znaky
- **funkce** — je základem analýzy jazyka
  - *konstanta*: funktiv, jehož přítomnost je nutná, aby se realizovala funkce
  - *proměnná*: funktiv, jehož přítomnost není nutná
  - změna v **plánu výrazu** vyvolá změnu v **plánu obsahu**
  - základní vztahy: *determinace* (a → b — např. adjektivum předpokládá substantivum), *interdependance* (a ↔ b — V ↔ Obj), konstelace (a V b — vzájemně se nepředpokládají)
- místo pojmů *označující* a *označované* používá výrazy **plán výrazu** a **plán obsahu**

Pražská lingvistická škola



Obrázek 17.2: Roviny jazykového popisu FGD (Funkční generativní popis): základní jednotky rovin jsou značeny žlutě a složené červeně

- Pražský lingvistický kroužek (PLK)
  - Roman Jakobson, Nikolaj Trubeckoj, Sergej Karcevskij
- hlavní rysy PLK:

1. **funkcionalismus** — různé použití pojmu **funkce** (ve smyslu "účel", tj. teleologicky)
   (a) přechod od komunikativních (společných dorozumívacích) potřeb k jejich jazykovému vyjádření
   (b) externí funkce jazyka — pragmatika; triáda *Darstellung, Ausdruck, Appell* (K. Bühler)
   (c) funkce jednotlivých jazykových prostředků (funkce *skládání* a *reprezentace* — vztah formy a funkce – *asymetrický dualismus*) — interní jazyková funkce
   (d) funkční větná perspektiva — aktuální větné členění (členění na téma/základ/topic a téma/jádro/focus)
   (e) funkční výklad jazykového vývoje — analogie, rovnováha jazykového systému
2. funkční *onomatologie* a *syntax* (*pojmenování* a *usouvztažnění*) — Vilém Mathesius
3. pojem *funkční styl* (jazykový styl) — sdělovací, administrativní, publicistický, odborný, umělecký... (Havránek); synonymie mezi slovy různých stylů: kůň — oř, hrob — rov
4. roviny jazykového popisu — vztah **skládání** (přechod od menší jednotky k větší v rámci jedné roviny; zde značíme →), vztah **reprezentace** (vztah formy a funkce — jsou to dvě stránky znaku, přechod mezi rovinami; zde značíme ⇒)
   – fonologická: distinktivní rys → foném; foném ⇒ morfoném
   – morfofonologická: morfoném → morf; morf ⇒ morfém
   – morfologická: séma → morfém, séma → formém; formém ⇒ tagmém
   – rovina větných členů: větný člen (tagmém) → věta; tagmém ⇒ sémantém
   – tektogramatická rovina: sémantém → propozice;
5. fonologie, morfologie — systém binárních opozic (pojmy *příznakový*/*nepříznakový*, *primární*/*sekundární*)
6. pragmatika (dopad jazyka na mluvčího) — Jakobson: *obecný význam*, shiftery (já, dnes, zítra)
7. popis ruského pádového systému pomocí binárních opozic (Jakobson)

Přínos strukturalismu (vzhledem k předchozím školám)

1. vidí v jazyce systém, strukturu se svébytnými systémovými vztahy
2. důraz na synchronii; i diachronii vykládá strukturně a systémově
3. jazyk je znakový (sémiotický, sémiologický) systém — spojení *označujícího* a *označovaného* pomocí *hodnoty*
4. jazyk je čistá forma (arbitrárnost znaků)
5. rozlišení *jazyka* a *mluvy* — dohromady tvoří *řeč*
6. jazyk je systém v pohybu

17.3 Poznámky

Kapitola 18

Státnice I3: Funkční generativní popis

18.1 Úvod

Teorii začal vyvíjet P. Sgall na FF UK na zač. 60. let, inspirovaný Chomského teorií a motivovaný strojovým překladem. Vycházel ale přitom z tradic Pražského lingvistického kroužku a strukturalismu – centrem je tedy jazykový systém (*langue*), klade se důraz na explicitní formalizaci a celé je to založeno na syntaxi.

Zákl. koncepce

Základní rysy teorie jsou:

- Závislostní přístup, valence (sloves a i dalších slovních druhů) (J. Panevová)
- *Stratifikace* (rozložení popisu na jednotlivé roviny podle úrovně abstrakce)
- Vztah *formy* a *funkce* (jedna forma má více funkcí na vyšších rovinách, jedna funkce více forem na nižších (*asymetrický dualismus*))
- *Jazykový význam* (vyloučení kognitivního obsahu z popisu; rozlišení *víceznačnosti* a zachování *vágnosti*)
- *Aktuální členění* jako součást významu (P. Sgall, E. Hajičová)

Generování

Původní koncepce vychází z představy, že generování (tvorba vět na základě významového popisu) bude jednodušší než analýza. V původní verzi bylo generování rozdělené na dvě hlavní fáze:

- *Generativní složka* – vymezovala (v původní verzi pomocí přepisovacích pravidel frázové gramatiky a složkových stromů, které ale měly indikaci směru a typu závislostí) správné zápisy vět na tektogramatické rovině
- *Překladové složky* – přepis na nižší úrovně až do běžného textu (formálně 4 zásobníkové automaty a jeden regulární)

Kvůli frázovým stromům se generoval jen jeden druh slovosledu apod. V pozdějších verzích byly frázové stromy upraveny na závislostní. Skutečně existovala v 70. – 80. letech implementace, která generovala korektní české věty, ale dodnes se nedochovala.

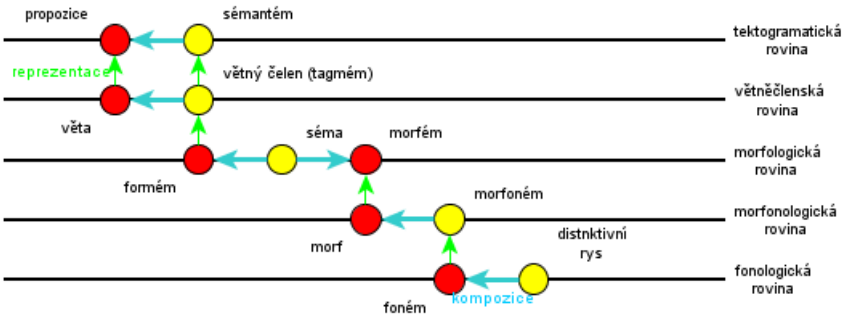
18.2 Roviny popisu

Popis jazyka se tu uskutečňuje na několika rovinách abstrakce, od lineárního proudu hlásek po samotný význam. Na každé rovině je ale reprezentovaná celá věta – v nižších lineární strukturou, na vyšších závislostním stromem.

- forma, funkce – nižší rovina je formou vyšší roviny (vztah *reprezentace*), základní jednotky na jedné rovině tvoří komplexní (*kompozice*)

Teorie FGD obsahuje tyto roviny popisu (od nejvyšší k nejnižší, hlavně v nižších úrovních není počet ustálený):

- *tektogramatická* (hloubková syntax, rovina jazykovéhyo významu)
- *povrchová syntax* (od 90. let Sgall zpochybnil její nutnost, v komputační lingvistice se z praktických důvodů stále používá)



Obrázek 18.1: Roviny popisu FGD: základní jednotky rovin jsou značeny žlutě a složené červeně

- *morfematická* (morfologická)
- *morfonologická*
- *fonologická* / *fonetická*

Platí, že nižší rovina je *formou* vyšší roviny a vyšší rovina *funkcí* nižší (vztah *reprezentace*). Na každé rovině existují základní jednotky popisu, které dávají dohromady složitější (vztah *kompozice*). Složitější jednotky pak zpravidla slouží jako základní na vyšší rovině.

Tektogramatická rovina musí obsahovat všechnu významovou informaci, během převodu na nižší roviny se žádná už nedodává. Základní jednotkou (uzly stromu) jsou *sémantémy*, celek se nazývá *propozice*. Ohodnocení sémantémů sestává z *komplexního symbolu*, lineární řazení jde podle "hloubkového slovosledu" – dynamiky výpovědi (aktuálního členění). Komplexní symbol sestává z následujících informací:

- *lexikální informace* – měla by obsahovat ne povrchový lexém, ale tektogramatický – synonyma by měla být ztotožněná, slovesná podstatná jména zahrnuta pod slovesa atd. (ale v PDT to tak úplně není)
- *morfologická informace* – jde taky o význam: mluvím o jednom, nebo více objektech? kdy se odehrává děj? (jen když si mluvčí vybírá, např. kongruence nás nezajímá)
- *syntaktická informace* – pomocí *funktoru* vyjadřuje vztah rodiče a dítěte ve stromu (ACT, PAT ... atd.)

Větněčlenská rovina pracuje s *tagmémy* (větnými členy), jejich kompozicí vzniká *věta*. Na morfologické rovině se z jednotlivých *sémat* (Sg., Nom., Fem. apod.) skládají jednak morfémy, odpovídající *morfům* na morfonologické rovině a jednak *formémy* (např. slova, předložkové vazby apod.), které odpovídají *tagmémům*. Morfémy se dělí na *lexikální* (kmeny, odvozovací předpony a přípony) a *gramatické* (ty vyjadřují zpravidla více *sémat*).

Fonému z fonetické roviny odpovídá *morfoném* (tj. všechny alofony v daném místě daného morfému). Kromě řetězů morfonémů – *morfů* – obsahuje morfonologická rovina i nástroje pro zachycení suprasegmentálních jevů (přízvukový takt, věta – intonace). Fonémy se skládají na fonologické/fonetické rovině z *distinktivních rysů*. Fonetická rovina se často z popisu vynechává (někdy zas se naopak ponechává a vynechává se morfonologická), lze taky nahradit fonologickou a fonetickou rovinu rovinou *grafématickou*.

18.3 Jazykový význam

Pro popis na tektogramatické rovině ve FGD se ostře odlišuje *jazykový význam* od *myšlenkového obsahu* (kognitivního obsahu, primárně nejazykového), tj. popisujeme jen to, co je obsaženo v jazyce – strukturu specifickou pro daný jazyk, včetně pragmatických rysů (indexy), ale zbavenou synonymie, homonymie a dalších nepravidelností. Už Saussure označoval význam za "formu obsahu". Rozlišuje se *víceznačnost*, naopak zachovává se *vágnost*.

Význam je neformálně to, co je viditelné přímo z formy vyjádření, obsah už jsou vyvozované výroky (v praxi je to často horší odlišit).

I pro rozlišení víceznačností je někdy třeba věcných znalostí:

Př. `Chytil tlouště na višni.` – musíme vědět, že neseděl na višni, ale že jde o návnadu

Pořád se ale jedná o víceznačnost, protože jde ale o jazykový fenomén (homonymie dvou různých doplnění).

Význam je vázaný na syntaktické i lexikální elementy:

Př. `wash = mýt / prát`, `go = jít / jet` – v angličtině to skutečně je jeden význam toho slova, není tam dvojznačné

Př. `fingers / toes = prsty` – totéž v češtině (prsty jsou to všechny, musí být blíže specifikovány rozvitím nebo kontextem)

Totéž platí např. o kategorii vidu, která se nekryje přesně s jinými vyjádřeními (časy v angličtině, lexikální prostředky v němčině apod). Jiné podobné fenomény jsou např. odlišení duálu nebo rozlišení osob "my včetně tebe" a "my kromě tebe" v některých jazycích.

Vágnost je naopak vlastní významovým jednotkám každého jazyka, vlastností významu je být vágní. Její rozlišení už není předmětem jazyka (a tedy popisu ve FGD), ale myšlenkového obsahu:

Př. `Francouzi nejedí polévku.` – že jde o "typické Francouze", věta neudává

Př. `Děti dostaly dárky.` – neříká se, kolik dárků dostalo které dítě

Vágní jsou i relační adjektiva – `švestkové / bramborové knedlíky` – nebo přechodníky. Vágnost je i v časové souslednosti vět v češtině:

Př. `Od té doby, co matka zemřela, bylo nám stále hůř.` – neříká se, jaký je vztah dvou vět, ale dá se pochopit, že následný

Většina vágních konstrukcí lze pochopit z kontextu nebo "vyrozumět" vyvozováním důsledků.

Z podobných důvodů, jako se omezujeme na význam, se u určování funktorů ACT a PAT omezujeme na syntaktické kritérium (viz dále) – často je jejich detailní sémantika totiž vágní a lze pouze "vyrozumět" z okolí.

Př. `Otec otevřel dveře. Klíč otevřel dveře. Vítr otevřel dveře.` – toto můžeme považovat za vágnost

18.4 Valence

Ve FGD je valence zkoumána už od počátku. Úzce se týká významu slov, proto se řadí na tektogramatickou rovinu. Dotýká se ale i nižších vrstev, protože valenční doplnění mohou vyžadovat konkrétní formu.

Každý autosémantický slovní druh je charakterizován valencí (*frame-bearing words*), primárně se jedná o *slovesa*, ale valenci lze nalézt i u *substantiv, adjektiv, a adverbií.*

Př: `zájem o co, bratr koho, předělaný z čeho na co, kolmý na co, blízko čeho`

Pro slovesa je ovšem teorie nejpropracovanější, nejpřesnější. V jiných teoriích se mluví i o valenci předložek, ale ve FGD to nemáme – to, že předložka dává pád substantivu, považujeme za morfologický jev (*rekci*).

Doplnění

Ve FGD se valenční doplnění dělí na *obligatorní* a *fakultativní* – obligatorní musí být (na tektogramatickérovině) vždy přítomna, abychom měli sémanticky úplný a srozumitelný zápis (nemusí být ale vyjádřena). Jejich přítomnost se prokazuje *dialogovým testem*:

Př. `A: Moji přátelé přijeli. B: Odkud? A: Nevím.` – OK, `B: Kam? A:*Nevím.` – nelze, proto určení kam je *obligatorní*

Některá doplnění jsou syntakticky *nevypustitelná* (tj. musí být vždy vyjádřená), jiná jsou *vypustitelná.*

Dále se doplnění dělí na *aktanty* a *volná doplnění*. Ve FGD se do valenčního rámce nějakého slova zapisují všechny aktanty (vč. fakultativních) a obligatorní volná doplnění (např. pro slovesa `přijít, chovat se`).

Pojetí aktantů u sloves ve FGD

Ve FGD Máme 5 aktantů, definovaných spíše syntakticky – ACT a PAT téměř výhradně, ostatní (EFF, ORIG, ADDR) částečně sémanticky. Kvůli svému spíše syntaktickému určení mají ACT a PAT hodně sémantických funkcí.

Jde o kompromis mezi hodně sémantickým přístupem, jako má např. FrameNet C. Fillmorea (doplnění jsou dnes pro každou typizovanou skupinu sloves jiná, hodně detailní), a hodně syntaktickým, jako obsahuje PropBank (aktanty jsou číslované a číslování specifické pro každé slovo). Syntaktický přístup k valenci prosazoval už Tesnière, z něj FGD vychází – např. akademická mluvnice češtiny (Daneš) razí naproti tomu sémantický přístup.

U aktantů se ve FGD uplatňuje princip *posouvání*:

- první aktant je vždy ACT
- druhý vždy PAT
- třetí je ADDR, ORIG nebo EFF
  - když nelze rozhodnout sémanticky, je to EFF

Př.: `Petr(ACT) vyrostl z chlapce(ORIG) v mladého muže(PAT!)`

Př.: `The janitor(ACT) opened the door(PAT) with a key(MEANS). A key(ACT) opened the door(PAT). The door(ACT) opened.`

EFF má primární význam "výsledek děje", nebo "vlastnost přiřazovaná patiensu" (`vyprávěl o nich, že...`). ADDR a ORIG jsou sémanticky homogenní, skoro jako volná doplnění. ADDR představuje příjemce informace nebo předmětu (i odebrání). ORIG značí látku původu, původce předmětu nebo informace při výměně:

Př.: `Dům je z kamene(PAT!). Vyrobil něco z něčeho(ORIG). Dozvědět se něco(PAT) od někoho(ORIG)`

ADDR a ORIG se jen málokdy dají dobře zkombinovat, ale existují i slovesa, kde je všech pět aktantů možných:

Př.: `Maminka(ACT) předělala dětem(ADDR) loutku(PAT) z kašpárka(ORIG) na čerta(EFF).`

Valenční informace ve slovníku

Valenční informace se uchovávají ve slovníku. Typicky patří valenční rámec k jednomu významu slova (*(základní) lexikální jednotce*), proto jeden *lexém* (soubor všech významů a forem příslušných jednomu základnímu tvaru – *lemmatu*) může obsahovat několik rámců.

Valenční slovník by se měl dělat z dat a ručně. Ukazuje se, že malý počet sloves pokryje velkou část korpusu, jen málo slov má větší počet lex. jednotek. Zároveň se různá slovesa chovají různě a mají různé rámce, i když popisují úplně stejnou sémantickou situaci; umožňují vyjádřit různé participanty.

V teorii FGD jsou zpracované slovníky PDT-VALLEX (pro Pražský závislostní korpus, jejž pokrývá) a VALLEX (ten se snaží o komplexní popis všech významů slov, slovesa jsou vybrána na základě frekvence v Českém národním korpusu).

Valence substantiv a adjektiv

Všechna valenční doplnění substantiv a adjektiv bývají vypustitelná. Liší se podle toho, zda se jedná o primární nebo deverbativní substantiva nebo adjektiva.

Primární substantiva

Rozlišují se následující doplnění:

- Partitiv/materiál (aktant) – množství/skupina (`dvojice, balení, sada`), kontejner (`sklenice, pytlík, tisíc`)
- Přináležitost (volné, u relačních substantiv (`otec, příbuzný, nadřízený`) aktant) – příbuzenský vztah, vztah části a celku (`střecha domu`), nositel vlastnosti (`míra čeho, délka čeho, čí upřímnost`), vlastnictví, přináležení (`klíč od`)
- Identita (volné) – metajazykové výrazy (`agentura Reuters, pojem času`), i další (`nápis Obětem války`)
- Autor (volné)
- Přívlastek restriktivní (volné)
- Přívlastek deskriptivní (volné)

Deverbativní substantiva

Pro valenční chování je důležitý typ derivace, jakým vznikly:

- *syntaktická derivace* – čistě syntaktický prostředek: `dělání, pokrytí`. Je tu vidět původní valence, ale často dochází k *abstrakci* (nevyjádření některých původně povinných aktantů)
- *lexikální derivace* – vznik ze sloves (základové slovo), ale sémanticky jde skutečně o substantiva: `letec, letiště`. Substantivum samo vyjadřuje jeden z participant děje – toto doplnění mizí (*zabudování pozice*), ostatní jsou přípustná, ale často taktéž nevyjádřená, dochází k uvolnění vazeb, často některá doplnění ani vyjádřit nelze (zní to divně).

Nejde o vyhraněné dělení, spíše škálu, přechod – je i spousta případů "mezi" (`dar, let` – "široce dějová jména").

Při konverzích ze sloves dochází ke změnám morfologického vyjádření, strukturní pády (Nom., Acc.) se primárně mění na genitiv:

Př. `vyrábět něco -> výroba čeho`

To ukazuje, že možnost vyjadřovat doplnění je u substantiv omezenější (genitivy se typicky nesmí opakovat). Nestrukturní pády (zejména Dat., Ins., ale i Gen., předložkové pády, infinitiv) většinou zůstávají, adverbia se mění typicky na adjektiva. Nepravidelnosti ale existují:

Př. `zájem o něco / na něčem`, `strachovat se čeho -> strach z čeho`

Primární adjektiva

Mají stejný repertoár možných doplnění jako slovesa, navíc komparativ má `než` a superlativ `z koho/čeho`. V teorii se zde už nepočítá s posouváním, ADDR, PAT se rozlišuje sémanticky. Většina adjektiv má jen jedno doplnění, jen výjimky více (`nápadný čím komu, vděčný komu za co`). Prototypicky se nevyskytuje ACT.

Deverbativní adjektiva

Sloveso se mění na adjektivum, které rozvíjí jedno z původních valenčních doplnění. Zachovávají rámec sloves až na jeden aktant, který je obsazený rozvíjeným substantivem. Povrchově jsou všechna doplnění vypustitelná.

Př.: např. `[kdo] omezí [co na co] -> *kdo* omezený [kým na co]`

Adverbia

Mají valenční chování, ale nikdo ho zatím podrobně nestudoval.

Př: `kolmo na co, vedle čeho, blízko čeho`

18.5 Aktuální členění ve FGD

Aktuální členění je ve FGD popisováno už od první verze. Navazuje tak na tradici strukturalismu a Pražského lingvistického kroužku, zejména práce Viléma Mathesia a Jana Firbase.

Definice

V různých teoriích najdeme různou terminologii, někdy se termíny kryjí přesně, někdy ne docela. *Informační struktura věty* je totéž co *aktuální členění věty* (původní termín od Mathesia), anglicky *topic-focus articulation* (podle ÚFALu, P. Sgalla a dalších), nebo *functional sentence perspective* (podle Brněnské školy, J. Firbase a dalších). Jde o dělení věty na:

- *základ*, *východisko*, *téma věty* nebo *topic*, tj. to, o čem se ve větě mluví (známá informace).
- *jádro*, *ohnisko*', *réma* nebo *focus*, tj. to, co se ve větě říká nového o známé informaci.

V pražském moderním přístupu se používá spíš anglických výrazů topic, focus a topic-focus articulation, protože původní české jsou zatíženy nepřesnostmi.

Vyjádření aktuálního členění

Informační strukturu lze vyjádřit různými prostředky, v češtině hlavně *slovosledem* a *intonací* – intonace je velmi důležitá, i když máme volný slovosled (a intonace má i další funkce). V angličtině např. je intonace kvůli pevnému slovosledu ještě důležitější.

Př.: `John gave me a letter. I met him [in a bookshop] [yesterday].` – jestli je focus `yesterday` nebo `in a bookshop`, poznáme jen podle intonace.

Př.: `Nejdražší je Audi. / Audi je nejdražší.` – při normální intonaci je focus na konci, proto první věta odpovídá situaci, kdy mluvím o *cenách* vozů, kdežto druhá připadá hovoru o *autech* a jejich vlastnostech.

V češtině můžeme topic-focus articulation rozlišit např. i použitím krátkého nebo dlouhého tvaru zájmene (ve focusu budou spíše dlouhé tvary, dlouhé tvary zájmen se ale využívají i pro vyjádření kontrastu v rámci topicu).

Př.: `Dej mi tu knížku. / Tu knížku dej mně.`

V angličtině se dá informační struktura vyjádřit i použitím určitého nebo neurčitého členu.

Př.: `A disabled man limped inside. / The disabled man limped inside.` – v prvním případě je invalida ve focusu, v druhém v topicu

Můžeme použít ale i různé částice (*focalizers*, *rematizátory*, přitahující větný přízvuk) nebo speciální syntaktickou konstrukci, tzv. *vytýkací* (to bývá častější v angličtině).

Př.: `Teprve Jeník dokázal draka porazit.` – *Jeník* je díky částici `teprve` ve focusu.

Př.: `Bill introduced John only to SUE. / Bill introduced only JOHN to Sue. / Bill only INTRODUCED John to Sue. / Only BILL introduced John to Sue.` – rematizátor mění focus

Př.: `Byla to vichřice, co ho zničilo.` – vytýkací konstrukce, ve focusu je `vichřice`.

Aktuální členění a význam

Aktuální členění úzce souvisí s *funkcí* sdělení, projevuje se ale různými *formami* (povrchovými strukturami věty); jedna forma může vyjadřovat naopak více různých aktuálních členění, ač to není tak časté:

Př.: `"Why do we dress boys in blue and girls in red?""Because they can't dress themselves."`

Aktuální členění patří do popisu významové stavby věty, v pražském popisu na tektogramatickou rovinu, protože jeho změna může změnit význam celé věty, když dojde ke změně *presupozice* – nutně předpokládané skutečnosti, aby měla věta smysl:

Př.: `The king of France didn't visit the exhibition. / The exhibition was not visited by the king of France.` – první varianta presuponuje existenci výstavy i krále, kdežto pro druhou nemusí francouzský král existovat.

Př.: `Aspoň dva jazyky zná v této místnosti každý. / Každý v této místnosti zná aspoň dva jazyky.` – První věta presuponuje dva stejné jazyky, ale druhá už ne.

Aktuálním členěním lze také měnit *dosah negace (scope of negation)* (negace může být buď v základu, nebo v jádře – potom se vztahuje na přísudek jen tehdy, je-li ten také v jádře):

Př. (1): `Moje sestra nehubovala bratra kvůli špatné známce` = `nehubovala vůbec / hubovala někoho jiného / hubovala bratra kvůli něčemu jinému.` – `moje sestra` nemůže být dotčeno negací, která je v jádru; stojí v základu

Př. (2): `Jirka nepřišel, protože mu došly peníze` – ve chvíli, kdy Jirka `nepřišel, ne` např. `protože byl nemocný, ale protože mu došly peníze`, se dostává negace do základu. Je to ale dvojznačné, můžu říct, že Jirka `přišel, protože chtěl vidět Marii` a potom je negace v jádru.

Různé druhy negace pak ovlivňují i presupozici:

Př.: `Jirka nezpůsobil naši porážku. / Naši porážku nezpůsobil Jirka.` – první věta je dvojznačná, kdežto v druhé je jasné, že jsme byli poraženi. Porážka se tak stává presupozicí.

Aktuální členění má navíc vliv i na *alegaci* věty (výrok, který vyplývá z kladné verze věty, ale ze záporné nevyplývá ani on, ani jeho negace). Může měnit presupozici v alegaci a naopak:

Př.: `Milanovou dceru včera viděl Jirkův bratr. / Včera Jirkův bratr viděl Milanovu dceru.` – v první větě se presuponuje existence Milanovy dcery a Jirkův bratr je jen alegován, kdežto v druhé větě tomu je přesně naopak.

Nejde jen o negace, ale i o kvantifikátory:

Př.: `Pražané většinou jezdí na Slapy. / Na Slapy jezdí většinou Pražané.` – v první větě neříkám, kdo všechno jezdí na Slapy, ale v druhé ano.

Pro nalezení změny ve významu při změně aktuálního členění nepotřebuju ale ani kvantifikátory:

Př.: `Na Moravě se mluví česky. Česky se mluví na Moravě.` – první případ je tzv. *exhaustive listing* – podávám úplnou informaci, protože na Moravě se jinak než česky nemluví; druhý ale ne, protože Česky se mluví i Čechách.

Př.: `Dogs must be CARRIED. / DOGS must be carried.` – první verze intonace říká, že mám-li psa, musím ho nést, druhá přikazuje nosit s sebou nějakého psa.

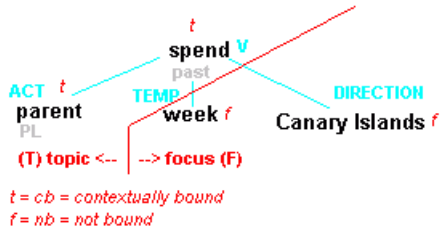
Začlenění do FGD

Ve FGD se na tektogramatické rovině jednotlivé uzly závislostního stromu řadí podle *dynamiky výpovědi* – míry kontextové zapojenosti ("hloubkový slovosled"). Uzly si nesou informaci, zda jsou:

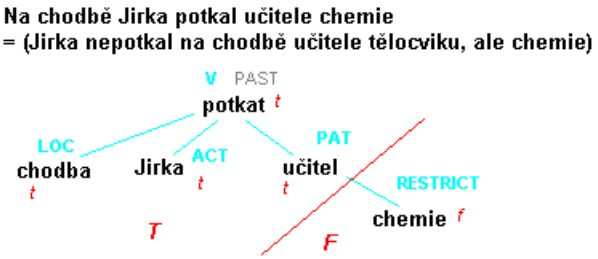
- *kontextově zapojené (contextually bound)* (značí malým písmenem t)
- *nezapojené (not bound)* (značí se malým f)

Na základě toho můžeme popsat, co je topic a co je focus.
Byla vypracovaná i pravidla pro oddělení topicu a focusu celé věty (značí se velkým T a F) podle těchto indikátorů:

1. Začne se od kořene (slovesa)
2. Přímé na slovese závislé složky vždy patří do T / F vcelku, se všemi svými členy dohromady (až na následující výjimku)



Obrázek 18.2: Ukázka topic-focus articulation podle funkčního generativního popisu



Obrázek 18.3: Pro oddělení topicu a focusu věty jsou nutná všechna 3 pravidla

3. Pokud jsou všechny přímé závislé složky kontextově zapojené, sleduje se podstrom poslední z nich (v pořadí členů ve větě), dokud se nenajde nezapojený element. Jeho podstrom je pak focus.

Mít jen první dvě pravidla nestačí (viz obrázek). Také to není jen výměna t a f za T a F, to platí jen na první závislé vrstvě, dá se to ukázat i na příkladu:

Př.: `Which schools do your children attend?` → `All (f) my children (t) attend (t) a private school (f) in London (f).` – v této větě je *all* sice kontextově nezapojené, ale patří do T.

V praxi byl tento algoritmus úspěšně zkoušen na větách z PDT.

Kontrastivní zapojení a souvětí v PDT

Pro popis v PDT musela být teorie trochu rozšířena. Byly tam zahrnuty:

1. *koordinace klauzí* – každá koordinovaná klauze má vlastní aktuální členění.
2. *subordinace* – závislé klauze jsou součástí aktuálního členění hlavní klauze, ač mají i svoje vlastní podřízené aktuální členění. Stojí-li podřízená klauze v topicu, většinou jí souvětí začíná, stojí-li ve focusu, souvětí jí zpravidla končí.
3. *kontrastivní zapojenost* – kromě t a f se přidává c pro *kontrastivně zapojené* větné členy. c i t patří do topicu (T).

Př.: `Kde jsi se setkal se svými spolužáky?` → `Jirku (c) jsem viděl v divadle (koordinace klauzí), Andulu (c) na koncertě.`

18.6 Pražský závislostní korpus

Pražský závislostní korpus (ve verzi 2.0) obsahuje necelé 2 milióny tokenů (novinových článků z Českého národního korpusu) anotované na základě FGD, z toho asi 800 000 až do úrovně tektogramatické roviny. Z časových, finančních a implementačních důvodů je tam ale několik rozdílů oproti původní FGD. Hodně z nich pramení z obráceného pohledu (analýza místo generování).

Máme tři roviny popisu + jednu "nultou" (tj. čtyři):

- *w-rovina* (slovní, nultá)
- *m-rovina* (morfologická)
- *a-rovina* (analytická, původní "větně členská")
- *t-rovina* (tektogramatická)

Nultá rovina obsahuje jen jednotlivá slova (slovní tvary = tokeny – včetně interpunkce), tak jak šla za sebou v novinách. Není to tedy přesně text z novin, ač s původními překlepy, prošel už tokenizací – tedy ani vytvoření nulté roviny není triviální.

Morfologická rovina zhruba odpovídá FGD, tektogramatická taky, ale analytická původní větněčlenské moc neodpovídá. *M-rovina* a *a-rovina* si odpovídají slovo od slova. U nulté roviny a a m-roviny nemusí být vztah 1:1 vždy, kvůli překlepům (např. zapomenutá mezera), ale děje se to jen zřídka, bývá buď 1:m, nebo n:1, když už (teoreticky vyloučit m:n nelze).

Anotační formát je jazyk PML založený na XML (nadefinované datové typy apod.), zpracovatelný editorem TrEd. Popis jednotlivých rovin je oddělený a roviny jsou provázány odkazy.

m-rovina

Sémata jsou sdružena ne do morfémů jako ve FGP, ale do slov. Text je rozdělen na jednotlivé věty; gramatické kategorie jsou reprezentovány tagy (morfolog. značkami). Překlepy jsou už opravené. Každý tvar má přiřazené lemma (slovníkový tvar), tam je taky rozlišená homonymie a přidané poznámky o vytvoření slova, např. "oživení" je z "oživit".

Morfologické tagy jsou poziční, s celkem 15 možnými údaji: slovní druh, jemněji určený slovní druh, rod, číslo, pád, rod vlastníka, číslo vlastníka, osoba, čas, stupeň, negace, slovesný rod + 3 rezervované (z nichž se používá poslední pro stylistické příznaky).

a-rovina

A-rovina už se skládá ze stromů. Ty mají drobné odlišnosti, např. kořen je speciální, technický, na něm teprve závisí hlavní sloveso (kvůli nevětným konstrukcím, parentezím apod.).

Uzly jsou ohodnocené *analytickou funkcí*, která popisuje syntaktické kategorie (Pred, Pnom (přísudek jmenný), AuxV, Sb, Obj, Atr, Adv, AuxP (předložka), AuxC (podřadící spojka) atd.). Členové koordinací, apozicí a parentezí jsou speciálně označeni a v TrEdu je možné určit *efektivního rodiče* nebo *efektivní děti* každého uzlu.

t-rovina

T-rovina docela dobře odpovídá tektogramatické rovině. Obsahuje tedy i uzly pro nevyjádřené větné členy, které jsou ve významu přítomné, neobsahuje uzly pro gramatická slova – jeden uzel tak může odkazovat k více uzlům analytické roviny. Uzlů je několik typů – pro běžná (vyjádřená) slova, pro nevyjádřená slova, koordinace a apozice, rematizátory apod.

Uzly mají určený *funktor* (tj. druh aktantu nebo volného doplnění, pro některé existuje další, užší klasifikace), udání *kontextové zapojenosti* a *hloubkového slovosledu* a nesou gramatické informace, jejichž skladba závisí na hloubkovém slovním druhu (vid, stupeň adjektiva, modalita, rod, negace, iterativnost děje, zdvořilost, ...). Uzel hlavního slovesa má určenou i větnou modalitu (oznamovací, tázací...). Navíc jsou určeny koreference – odkazy (např. zájmen) k objektům dříve zmíněným v textu.

Každý uzel má určené tektogramatické lemma, které ale někdy neodpovídá teorii z implementačních důvodů.

Kapitola 19

Státnice I3: Formální sémantika

Formální sémantika:

- interdisciplinární směr, ve kterém se spojili formálně orientovaní lingvisté, filosofové, logikové, i někteří odborníci z oblasti computer science
- souvisí s rozvojem *intenzionální sémantiky* a vznikem dalších formálně-sémantických systémů, které reagovaly na diskuse o jejích omezeních

19.1 Úvod

- Chomsky — inicioval interakci mezi lingvistikou a matematikou
- otázka, zda by bylo možné aplikovat matematické metody, které se ukázaly užitečné pro analýzu syntaxe, také na sémantiku
- **Syntax:** zabývá se výrazy a jejich skladbou *vs* **Sémantika:** studuje významy, u nichž je zásadně problematické se shodnout byť i jen na tom, čím vlastně jsou, natož pak na tom, jak je uchopit
- první pokusy o matematické uchopení sémantiky přirozeného jazyka
  - *generativní sémantika* Lakoffa a McCawleyho
  - teorie J.J. Katze a P. Postala
  - *logická forma* (Chomsky, Jackendoff)
  - v Čechách se v rámci Sgallovy varianty generativní gramatiky, *funkčního generativního popisu*, rozvinula koncepce *tektogramatiky*
  - kritika od filosofů a logiků, že sémantika nějak zásadně souvisí s pravdivostí a pravdivostními podmínkami, které výše jmenované přístupy nepostihují — jenom překlady z jednoho jazyka (toho přirozeného) do jazyka jiného, jakési *stromovštiny*
- *Teorie sémantiky formálních jazyků*
  - vycházely z propojení významu s pravdou
  - na sklonku devatenáctého století jeden ze zakladatelů moderní logiky, německý matematik a filosof Gottlob Frege nastínil, jakým způsobem by mohla logika význam matematicky zachytit. Alfred Tarski, Rudolf Carnap a další je potom v podstatě jenom zaintegrovali do rozvíjející se formální logiky.
  - z hlediska teorií přirozeného jazyka byl problém ve dvou věcech:
    1. jazyky, se kterými logici běžně pracovali a pro které budovali své formální teorie sémantiky, měly ve srovnání s přirozeným jazykem příliš jednoduchou syntaktickou strukturu
    2. sémantika, kterou byly opatřovány, nebylo možné považovat za realistickou explikaci významů, jaké mají výrazy v přirozeném jazyce
  - kolem roku 1970 se objevily formální jazyky, jejichž sémantika již nebyla z hlediska explikace sémantiky jazyka přirozeného tak nepoužitelná jako sémantika standardní logiky
    * *modální logiky* — logiky možnosti a nutnosti
    * *intenzionální logika* — opírá se o pojem *možného světa*
      · americký logik Richard Montague
      · český logik Pavel Tichý

- základní myšlenka: *matematizace významu*
- zapojením *možných světů* přechod od extenzionálního k intenzionálnímu modelu sémantiky přirozeného jazyka

19.2 Extenzionální model významu

Principy extenzionální sémantiky

- funkcionální koncepce sémantiky

Fregův manévr

- významy některých druhů výrazů by bylo možné explikovat jako určité *funkce* v matematickém slova smyslu
- podmět obvykle označuje nějakou věc: např. podmět „Gottlob Frege“ označuje Gottloba Frega
- (oznamovací) věta označuje svou pravdivostní hodnotu: věta „Gottlob Frege je dramatik“ označuje *nepravdu* (Frege *nebyl* dramatik)
- predikát P spolu s podmětem N_1 vytvoří nějakou větu V_1, s podmětem N_2 vytvoří V_2

$$P + N_1 = V_1 \text{ “být dramatikem” + “Frege” = “Frege je dramatikem”}$$

$$P + N_2 = V_2$$

“být dramatikem” + “prezident ČR” = “Prezident ČR je dramatikem” (za dob Václava Havla)

- predikát P můřeme vidět jako prostředek přiřazení věty V_1 podmětu N_1

$$P : N_1 \rightarrow V_1 \text{ “být dramatikem”: “Frege” } \rightarrow \text{ “Frege je dramatikem”}$$

$N_2 \rightarrow V_2$ “prezident ČR” $\rightarrow$ “Prezident ČR je dramatikem”

- *denotát* $\|Y\|$ — to, co je označováno výrazem Y

$$||P|| : ||N_1|| \rightarrow ||V_1||$$

$||N_2|| \rightarrow ||V_2||$

$\|$být dramatikem$\|$: $\|$Frege$\|$ $\rightarrow$ $\|$Frege je dramatikem$\|$

$\|$prezident ČR$\|$ $\rightarrow$ $\|$Prezident ČR je dramatikem$\|$

- denotáty podmětů jsou jimi pojmenovávané věci a denotáty výroků jejich pravdivostní hodnoty
  - $\|$být dramatikem$\|$ je funkce, která přiřazuje osobě Gottlobu Fregovi pravdivostní hodnotu *nepravda* (N) a Václavu Havlovi *pravda* (P)
- zobecnění *Fregova manévru* na spojky nebo příslovce
  - spojka zřejmě ‘vyrábí’ větu z dvojice vět — její denotát tedy můžeme ztotožnit s funkcí, přiřazující pravdivostní hodnoty dvojicím pravdivostních hodnot
    - $\|$a$\|$ bude přiřazovat P pouze dvojici $< P, P >$, zatímco $\|$nebo$\|$ bude přiřazovat P každé dvojici kromě $< N, N >$
  - příslovce se spojují s (unárními) predikáty v komplexní (unární) predikáty

Extenze a Intenze

POZOR: fregovské denotáty jistě nejsou přijatelnými explikáty významů v intuitivním slova smyslu

- významem věty by byla její pravdivostní hodnota a všechny pravdivé věty by tudíž měly stejný význam
- proto Frege zavedl pojmy **význam** (ve formě denotátů) *vs* **smysl** (intuitivní smysl slova)
- později byly tyto dva pojmy nahrazeny pojmy **extenze** (význam) a **intenze** (smysl)

Možný svět

- znát intenzi výrazu znamená být schopen určit jeho extenzi v každém *možném světě*
- intenzi výrazu tedy můžeme obecně ztotožnit s funkcí, přiřazující každému možnému světu extenzi tohoto výrazu v tomto možném světě
  - intenzí výroku je funkce, která každému možnému světu přiřadí pravdivostní hodnotu tohoto výroku v tomto možném světě
  - intenzí singulární fráze je funkce, která každému možnému světu přiřadí objekt označovaný touto frází v tomto možném světě
  - intenzí predikativní fráze je funkce, která každému možnému světu přiřadí třídu objektů, o kterých je tento predikát pravdivý v tomto možném světě

Model jazyka odpovídající standardnímu predikátovému počtu

Sémantický model jazyka je tvořen čtyřmi komponentami:

1. slovník
2. syntaktická pravidla
3. přiřazení denotátů slovům
4. pravidla pro to, jak „počítat“ denotáty složených výrazů z denotátů jejich složek

Všechna pravidla bodu 4 mají tvar ‘vezmi denotát jedné složky a aplikuj ho na denotáty těch ostatních‘

Kategoriální gramatika a teorie typů

Kategoriální gramatika

- zobecnění Fregova manévru (funkční aplikace)
- jazyk, jehož všechna pravidla fungují jako funkční aplikace
- máme-li gramatické pravidlo, které kombinuje výrazy kategorií K_1 ... K_n ve výraz kategorie K, pak denotáty výrazů jedné z kategorií $K_1 ... K_n$, řekněme K_i, musí být funkcemi aplikovatelnými na denotáty výrazů zbylých kategorií $K_1, ..., K_{i-1}, K_{i+1}, ..., K_n$
- kategorii K_i budeme označovat indexem $K/K_1, ..., K_{i-1}, K_{i+1}, ..., K_n$

Teorie typů

- kategorie výroků a termů budeme označovat **V** a **T**
- pak bude kategorie predikátů označena jako **V/T** (predikát potřebuje term, aby vytvořil výrok)
- spojky *a* či *nebo* můžeme nahlédnout výrazy kategorie **V/V,V** a *příslovce* jako výrazy kategorie **(V/T)/(V/T)**

Komponenty kategoriální gramatiky

1. Slovník
   - soubor KAT primitivních kategorií (např. kategorií **V** a **T**)
   - další gramatické kategorie dostaneme pomocí operace lomítka
     - kdykoli jsou K_1 a K_2 gramatickými kategoriemi, je gramatickou kategorií i K_1/K_2
   - každá kategorie pak obsahuje nejvýše konečný počet slov
2. Syntax
   - každé slovo kategorie K je výrazem kategorie K
   - pravidlo pro kombinaci výrazů:
     - je-li Y výraz kategorie K_1/K_2 a je-li Z výraz kategorie K_2, je Y(Z) výrazem kategorie K_1
3. Denotáty slov

- každé primitivní kategorii K je dána množina D_K, tzv. *doména* kategorie K
  - doménou D_T přiřazenou kategorii T může být nějaká daná množina *individuí* D
  - doménou D_V přiřazenou kategorii V množina {P,N} dvou pravdivostních hodnot
- přiřazení rozšíříme na všechny kategorie tak, že za $D_{B/A}$ vezmeme množinu všech funkcí z D_A do D_B, kterou budeme značit $[D_A \Rightarrow D_B]$

4. Denotáty složených výrazů
   - denotát ‖Y(Z)‖ složeného výrazu Y(Z) je dán jako hodnota ‖Y‖(‖Z‖) aplikace funkce ‖Y‖ na argument ‖Z‖

Lambda-abstrakce a lambda-kategoriální gramatika

- vezmeme nějaký složený výraz a „uděláme do něj díru“, tj. odstraníme z něj nějakou složku a nahradíme ji nějakým formálním symbolem, třeba písmenem x
  - z výroku *dramatik(Frege)* můžeme udělat **matrici** *dramatik(x)*
- matrice *dramatik(x)* může být chápána jako předpis funkce $\lambda x.dramatik(x)$, pro kterou platí $f(\| Z \|) = \| Y^{x \leftarrow Z} \|$, kde $Y^{x \leftarrow Z}$ značí variantu výrazu Y, ve které byl symbol x nahrazen výrazem Z
- pravidlo nahrazení složitějšího výrazu $(\lambda x.Y)(Z)$ jednodušším $Y^{x \leftarrow Z}$ nazýváme *pravidlem lambda-konverze*
- kategoriální gramatice obohacené o pravidlo lambda-konverze budeme říkat *lambda-kategoriální gramatika*

19.3 Intenzionální model významu

Meze extenzionální sémantiky, modální logika a pojem možného světa

- výroky *Praha je město* a *Havel je dramatik* jsou oba pravdivé a mají tedy tutéž *extenzi*, avšak jistě nemají tentýž význam
  - ačkoli mají stejnou pravdivostní hodnotu, je možné, aby ji stejnou neměli. Mohl by jistě existovat svět, ve kterém by Havel byl dramatik, ale Praha byla pouhou vesnicí, či svět, kde by Praha byla městem, ale Havel byl třeba hospodským
- intenze výrazu je funkce, která každému *možnému světu* přiřadí *extenzi* tohoto výrazu v tomto možném světě

Extenze vs. intenze

- nyní namísto extenzí, které jsme potřebovali v rámci extenzionální sémantiky, potřebujeme funkce, které mají za definiční obor množinu všech možných světů a za obory hodnot extenze
  - namísto objektů z D_T potřebujeme objekty $[MS \Rightarrow D_T]$ (kde MS je množina všech možných světů), namísto $[D_T \Rightarrow D_V]$ potřebujeme $[MS \Rightarrow [D_T \Rightarrow D_V]]$
- denotát jednoduché věty bude výsledkem poněkud komplikovanější operace s denotáty jejích částí (kde w je možný svět):

```
‖P(T)‖(w) = (‖P‖(w))(‖T‖(w))
```

- někdy je potřeba k výpočtu denotátu komplexního výrazu v daném možném světě potřeba nejenom extenze jeho komponent v tomto možném světě. Např. ve větě

```
Frege hledá prezidenta ČR
```

- Frege může jistě hledat prezidenta ČR i v možném světě, ve kterém žádný takový prezident neexistuje (představme si například, že Česko je v tomto světě monarchií, což ovšem Frege neví)
- na predikát *hledat* můžeme nahlédnout nikoli jako vztah mezi extenzemi (individui), ale jako vztah mezi extenzí a intenzí, takže ‖Hledat(N1,N2)‖(w) nebude (‖Hledat‖(w))(‖N_1‖(w),‖N_1‖(w)), ale (‖Hledat‖(w))(‖N_1‖(w),‖N_1‖)
- rozlišení dvou typů *postavení* (supozic) výroku:
  1. *de re*
     - výraz, který je součástí nějakého výroku a přispívá k pravdivostní hodnotě tohoto výroku v každém možném světě jen svou extenzí
  2. *de dicto*
     - v ostatních případech, např. výraz *prezident ČR* je jako předmět slovesa *hledat* v supozici *de dicto*

Montaguova intenzionální logika

- výrazy mají intenze navíc kromě svých standardních denotátů
- výraz kategorie K má přiřazen jednak prvek D_K (denotát či extenzi) a jednak prvek $[MS \Rightarrow D_K]$ (smysl či intenzi)

Tichého intenzionální logika a dvousortová teorie typů

- soubor základních kategorií extenzionálního lambda-kategoriálního jazyka, který je v typickém případě tvořen kategoriemi **V** a **T**, je obohacen o kvazikategorii **S**, které bude jako doména odpovídat množina MS možných světů
- výrazy, které byly při standardní analýze analyzovány jako kategorie K, jsou analyzovány jako kategorie **K/S**
- čili výrazy přímo denotují intenze

19.4 "Hyperintenzionální" modely významu

Domněnkové věty a intenzionální izomorfismus

- problém s doměnkovými větami jako

`Frege se domnívá, že jedna a jedna jsou dvě` (1)

- věta *jedna a jedna jsou dvě* je matematickou pravdou a protože matematické pravdy nezávisejí na stavu světa, bude extenzí této věty v každém možném světě pravdivostní hodnota P; a její intenzí tedy bude konstantní funkce přiřazující P každému možnému světu
- tatáž funkce ale zřejmě bude intenzí jakékoli matematické pravdy, např.

`Existuje nekonečně mnoho prvočísel`

- takže podle intenzionální analýzy by nemohlo nastat, že by byla například věta (1) pravdivá, zatímco věta (2) nepravdivá

`Frege se domnívá, že existuje nekonečně mnoho prvočísel` (2)

- to se zdá být v rozporu s intuicí: zdá se, že Frege může (v nějakém jiném možném světě, když ne v tom našem) docela dobře vědět, že jedna a jedna jsou dvě, a současně se mylně domnívat, že prvočísel je jenom konečně mnoho
- řešení pomocí *hyperintenzionální* sémantiky
  - jemnější sémantická analýza
  - myšlenka, že je-li intenze výrazu výsledkem nějaké kombinace intenzí jeho částí, pak bychom význam v intuitivním slova smyslu neměli explikovat jako výslednou intenzi, ale jako nějakou formu zachycení samotného procesu kombinace

Teorie strukturovaných významů

- nejjednodušší variantou realizace hyperintenzionální sémantiky
- ztotožnění denotátu složeného výrazu s uspořádanou n-ticí tvořenou denotáty jeho částí

Tichého konstrukce

- n-tice jsou jakési ‚konstrukce', které mají primárně co dělat nikoli s tím, jak se věci spojují v rámci světa, ale spíše s tím, jak uživatelé jazyka kombinují významy částí ve významy celků

Situační sémantika

19.5 Dynamické modely významu

Problémy anaforické reference

- význam zájmen, např. *on*
- individuum, které zájmeno pojmenovává, není určeno možným světem, ale spíše *kontextem*
- **dynamická sémantika** postavena na pojmu *kontextu* či *informačního stavu*
- výroky v jejím rámci jsou chápány jako denotující nikoli pravdivostní hodnoty či funkce z možných světů do pravdivostních hodnot, ale funkce z informačních stavů do informačních stavů, tzv. *přechody* (*updates*)

Teorie reprezentace diskurzu (DRT)

- opírá se o struktury podobné situacím, nazývané *struktury reprezentace diskurzu*, avšak soustředí se především na jejich ‚kinematiku'
- v rámci DRT jsou kontexty či informační stavy uchopeny jako ‚reprezentované situace' a věta je chápána jako prostředek přebudování takové reprezentace na nějakou reprezentaci bohatší

Dynamická logika

- van Benthem (1997)

19.6 Vybrané speciálnější problémy sémantiky

Různý počet doplnění slovesa

- obecně lze jednoduchou větu nahlédnout jako sloveso (které může být modifikováno různými ‚příslovečnými určeními') spojené s různými druhy jmenných doplnění (podmětem, 'předměty')
- počet jmenných doplnění slovesa se může větu od věty měnit (přičemž absence některých z nich může znamenat absenci příslušných argumentů na úrovni sémantiky, zatímco absence jiných jenom jejich nevyjádřenost)

```
Karel přednáší posluchačům báseň          (3)
Karel přednáší posluchačům                (4)
Karel přednáší báseň                      (5)
```

- několik možností reprezentace
  1. spojuje-li se totéž sloveso s různým počten argumentů, jde o případ homonynie a je tedy v pořádku, že pro jeho analýzu musíme v každém případě použít jiný predikát (s jinou aritou)
  2. můžeme pracovat s formálním jazykem, jehož predikáty mohou mít proměnný počet argumentů (nebo jejichž argumenty nejsou přímo označeními individuí, ale označeními třeba množin individuí)
  3. strom znázorňující strukturu věty převést na logickou formuli takovým způsobem, že by se uzly staly termy a označení hran by se stalo predikáty
     - z věty (5) by se stalo $Act(Prednaset, Karel) \wedge Obj(Prednaset, Basen)$

Východisko a jádro věty

- z hlediska dynamiky diskurzu je ve větě třeba rozlišit část, kterou se věta 'ukotvuje v kontextu' (východisko, to jest specifikace toho, o čem se hovoří) od části, která přináší skutečně novou informaci (jádro výpovědi, to jest vyjádření toho, co se o tom říká)
- pro *východisko* věty je charakteristický předpoklad *existence*
  - Český král je logik — spíše než nepravda je tato věta ne úplně smysluplný výrok, protože není jasné, o čem se vůbec mluví (Česko nemá krále)
- pro *jádro* je zase charakteristický předpoklad *reprezentativnosti*
  - pokud bude odpověd na otázku *Kde se mluví Německy?* třeba *V Hamburku*, bude to odpověď, která sice není nesprávná, ale není reprezentativní (německy se mluví i na jiných místech, např. v Mnichově)

19.7 Odkazy

- stručná skripta k předmětu *Úvod do teoretické sémantiky* jsou ve Studnici

Kapitola 20

Státnice I3: Jazykové korpusy a lingvistická anotace

20.1 Zdroje dat

- strojově čitelné soubory dat
  - noviny, sborníky, romány, technická dokumentace, dialogy, internet, ...
  - závisí na úkolu, který chci řešit
  - *čím větší, tím lepší*

Kódování znaků

- současné počítače jsou číslicové — všechno (program, data) je reprezentováno jako číslo
- pro práci s textem je potřeba zavést konvenci pro transformaci *číslo* ↔ *znak abecedy*

Základní pojmy:

- **znak** (character)
  - abstraktní pojem
  - nemá sám o sobě žádnou číselnou reprezentaci ani pevnou grafickou podobu — např. `velké písmeno A s čárkou`
- **repertoár znaků** (character repertoire)
  - množina znaků
  - otázka identity: stejně vypadající znaky mohou být považovány za logicky odlišné (A v latince a alfabetě)
- **kódování** (encoding)
  - algoritmus pro převod posloupnosti znaků na posloupnost oktetů
- **kódová pozice znaku** (code position)
  - číselná reprezentace znaku (nezáporné celé číslo)
- **kódování** (character code)
  - 1-1 relace mezi prvky repertoáru znaků a nezápornými celými čísly
- **glyf**
  - vizuální prezentace znaku
- **font**
  - repertoár glyfů pro množinu znaků

ASCII

- American Standard Code for Information Interchange (od 1950's)
- sedm bitů – hodnoty 0-127
- 0-31,127 — kontrolní znaky (Escape, Line Feed)
- 32-126 — mezera, speciální znaky, číslice, velká a malá písmena latinky:

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

- výhody:
  - velice jednoduché kódování: jeden znak – jedna kódová pozice
  - minimální objem: 1 znak – 1 oktet (jeden bit zbyde — oktety 128-255 zůstávají nevyužité)
- zásadní nevýhoda:
  - naprosto nedostačující pro repertoáry znaků jiných národních abeced

8-bitová kódování

- potřeba dalších znaků → vznikají nová kódování obsahující ASCII jako podmnožinu a navíc využívají oktety 128-255 (stále platí jeden znak – jeden oktet)
- International Standard Organisation vydává skupinu standardních kódování pro některé skupiny jazků — rodina ISO 8859 (1980's)
- ISO 8859-1 (ISO Latin 1) – západoevropské jazyky
- u češtiny a ostatních středo- a východoevropských jazyků vládne anarchie:
  - ISO 8859-2 (ISO Latin 2)
  - Windows-1250
  - Koi-8
  - Bratři Kameničtí
  - vlastní „standardy“ IBM, Apple

Unicode

- jediné řešení: víceoktetová kódování (neplatí ale mýtus, že Unicode je 16-bitové kódování!!!)
- 1991 — Unicode Consortium vytváří normu Unicode (resp. ISO 10646), která určuje repertoár znaků a jejich kódové pozice
- v současnosti 30 světových abeced užívaných v několika stovkách jazyků, cca 40000 znaků — arabština, sanskrt, čínština, japonština, korejština, ... (ambice: 250 abeced pro několik tisíc jazyků)
- znaky v Unicode: „LATIN CAPITAL LETTER A WITH ACUTE“
- různé typy fyzické reprezentace kódů:
  - UTF-8
    - proměnná délka: 1-6 oktetů na znak (použitých max. 4)
    - v prvním oktetu posloupnosti určuje počet bitů zleva po první nulu celkový počet oktetů
    - další byty zápisu stejného znaku vždy začínají `10xxxxxx`, jiný byte je začátek zápisu dalšího znaku
    - výhoda: znaky ASCII se v UTF-8 kódují stejně → kompatibilita
    - výhoda pro češtinu: všechny znaky české abecedy se kódují jedním nebo dvěma oktety
  - UTF-16 — každý znak v BMP (Basic Multilingual Plane) je reprezentován dvěma oktety; ostatní znaky jsou reprezentovány čtyřmi oktety
  - UTF-32 — každý znak jsou čtyři oktety
- problémy Unicode v porovnání s 8-bitovými kódováními:
  - řetězcová ekvivalence vizuálně totožných znaků různých abeced: např. *A* v latince, azbuce a alfabetě
  - abecední řazení: např. „LATIN CAPITAL LETTER A WITH ACUTE“ vs. „LATIN CAPITAL LETTER A WITH GRAVE“

Jiná řešení

- transliterace
- *escape notation* — znaky mimo kódování, které je k dispozici, mohou být nahrazeny dohodnutou posloupností znaků (různé pro různé systémy)

Ä: `\"{A} (TeX)` `Ä (HTML)`

20.2 Anotace

- přidávání vybrané lingvistické informace k již existujícímu korpusu
- **(semi)automatická** nebo **ruční**
- několik fází zpracování
  1. shromáždění materiálu — skenování + OCR, WWW jako korpus
  2. konverze + čištění — jednotný formát a kódování
  3. klasifikace dokumentů
  4. segmentace dokumentu — hranice vět, hranice slov (*tokenizace*; problém co je slovo — číselné výrazy, jazyky bez mezer)
- textová reprezentace: jednoduchá implementace, uložené přímo v nízkoúrovňovém datovém formátu
- grafická reprezentace: intuitivnější (minimálně pro lingvisty)
  - potřeba speciálních nástrojů pro anotaci, správu dat, dávkové zpracování příkazů
- tým lidí, kteří provádí anotaci
  - vedoucí (rozděluje úkoly, najímá lidi)
  - editor (člověk) pro tvorbu anotačních pravidel
  - lingvista pro řešení lingvistických jevů
  - anotátoři — nejpočetnější skupina
  - technická podpora/programátor
  - kontrola dat

20.3 Datové formáty

Rozmanitost datových formátů

- rozmanitost datových zdrojů vedla k rozmanitosti datových formátů
- postupně ale dochází ke konvergenci k XML

Ukázky různých formátů

- Negra Treebank – tabulka (odkazy na syntaktické rodiče)

```
%% word                tag     morph           edge parent
#BOS 1 1 985275570 1
Mögen                 VMFIN   3.Pl.Pres.Konj  HD      508
Puristen              NN      Masc.Nom.Pl.*   NK      505
aller                 PIDAT   *.Gen.Pl        NK      500
Musikbereiche         NN      Masc.Gen.Pl.*   NK      500
```

- Penn Treebank – závorková notace stromů

```
( (S (NP-SBJ The proposed changes)
    (ADVP also)
    (VP would
        (VP allow
            (S (NP-SBJ executives)
               (VP to
```

```
                    (VP report
                        (NP (NP exercises)
                            (PP of
                                (NP options)))
                        (ADVP-TMP (ADVP later)
                                  and
                                  (ADVP less often)))))))
.))
```

- WordNet – databáze s odkazy na ID příbuzných pojmů (ukázka: index a datový řádek)

```
entrance v 2 3 @ ~ + 2 0 01806505 00020926

00044113 04 n 05 entrance 0 entering 0 entry 0 ingress 0 incoming 0 012 @ 00043484 n 0000 + 01958650 v 030
+ 01958650 v 0201 + 01671620 v 0201 + 01958650 v 0101 ~ 00044454 n 0000 ~ 00044639 n 0000 ~ 00044745 n 000
~ 00044898 n 0000 ~ 00045146 n 0000 ~ 00046615 n 0000 ~ 01178182 n 0000 | the act of entering; "she made a
```

- Valenční slovník VALLEX

```
* HLÁSIT
 ~ ned: hlásit
 + ACT(1;obl) ADDR(3;opt) PAT(o+6;opt) EFF(4,jak,že;obl) LOC(;typ)
    -synon: oznámit
    -example:  hlásit někomu o něčem zprávu
    -reciprex:  hlásili si navzájem o sob? nové zprávy
    -reciprocity: ACT-ADDR-PAT
    -use: prim
    -class: communication
    -freq: 8;9
 ~ ned: hlásit
 + ACT(1;obl) ADDR(3;opt) PAT(na+4;opt) EFF(4,jak,?e;obl) LOC(;typ)
    -synon: udat
    -example:  hlásit na něj, že přechovává zakázané knihy
    -reciprocity: ACT-ADDR-PAT
    -use: posun
    -class: communication
    -freq: 3
```

PDT

- Pražský závislostní korpus (Prague Dependency Treebank)
- původně vlastní formáty
  - **CSTS** (Czech Sentence Tree Structure) – hlavní formát PDT 1.0, založený na SGML
    - původně pouze morfologická rovina, pozd. i analytická rovina
    - struktura zachycena pomocí odkazů

```
< s id="ln95048:053-p6s48">
  <f cap>Černý<l>černý-1_^(barva)
    <t>AAMS1----1A----<A>Sb<r>1<g>3
  <f>se<l>se_^(zvr._zájmeno/částice)
    <t>P7-X4----------<A>AuxT<r>2<g>3
  <f>vzdal<l>vzdát
    <t>VpYS---XR-AA---<A>Pred<r>3<g>0
  <d>.<l>.
    <t>Z:-------------<A>AuxK<r>4<g>0
```

- – **FS** (Feature Structures)
  - pro analytickou a morfologickou rovinu, dvojice atribut – hodnota
  - zachycení struktury pomocí závorek — nutná linearizace stromu
  - problémy při přesouvání uzlu z konce věty na začátek další věty (při opravě chyb v segmentaci) — nutno přepočítat celou strukturu

```
[#10,AuxS,#10,SENT,0,0](
  [vzdal,Pred,vzdát_se,PRED,3,3,0](
    [Černý,Sb,černý,ACT,1,1,3],
    [se,AuxT,se,2,2,3,hide]
  ),[.,AuxK,&Period;,4,4,0,hide]
)
```

- později přechod na formát **PML**
  - reprezentace pomocí XML
  - obecný jazyk poskytující nástroje k popisu omezení a vlastností svých aplikací pomocí **PML schématu** (popis toho, co se může v instancích vyskytovat, jaká je struktura, role)
  - PML instance
    - hlavička: odkaz na externí schéma nebo schéma samo
    - další odkazy na externí soubory
    - ostatní prvky podle příslušného schématu

```
<?xml version="1.0"?>
  <annotation xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head>
    <schema href="example1_schema.xml"/>
  </head>
  <meta>
    <annotator>Jan Novak</annotator>
    <datetime>Sun May 1 18:56:55 2005</datetime>
  </meta>
  <trees>
    <LM ord="2">
      <func>Pred</func>
      <form>loves</form>
      <governs>
        <LM ord="1">
          <func>Subj</func>
          <form>John</form>
        </LM>
        <LM ord="3">
          <func>Obj</func>
          <form>Mary</form>
        </LM>
      </governs>
    ...
```

XML

- GML (Generalized Markup Language), SGML (Standard GML), HTML, XML
- **well-formed** (odpovídá XML) vs **valid** (odpovídá druhu konkrétního dokumentu) dokument – validace pomocí DTD definic
- jmenné prostory, XPath, XSL + XSLT
- API pro XML: **SAX**, **DOM**

20.4 Typologie korpusů

- **korpus** — strukturovaný, unifikovaný a (často též označkovaný) rozsáhlý soubor jazykových dat
- **korpusová lingvistika**
  - zabývá se studiem jazyka a obsahuje všechny procesy týkající se zpracování, používání a analýzy psaných a mluvených (strojově čitelných) korpusů
  - relativně moderní pojem, který označuje přístup založený na příkladech použití jazyka v "reálném světě"
- kritéria klasifikace korpusu

  - **médium** — tištěný, elektronický text, digitalizovaná řeč, video
  - **metoda tvrby** — *vyvážený* (je vůbec možné vytvořit vyvážený korpus?) vs *speciální* (oborový)
  - **jazykové proměnné**
    * *jednojazyčný* vs. *mnohojazyčný*
    * původní text vs. překlady z cizího jazyka
    * rodilý mluvčí vs. nerodilý mluvčí
  - **jazykový vývoj** — *synchronní* vs. *diachronní*
  - *prostý* vs. *anotovaný*

Korpusy

Jednojazyčné korpusy

- Brown Corpus — 1 MW (1964)
  - děrné štítky, publikovaná statistika, později otagován; výběr amerických textů z r. 1961
- British National Corpus — 100 MW (1994)
  - 100 MW soubor psaného (90 procent) a mluveného (10 procent) jazyka obsahující současnou britskou angličtinu, morfologická anotace
- Deutsches Referenzkorpus/Cosmas IDS-Mannheim (2004)
  - >4 GW současné psané němčiny (bez anotace);
  - automatická lematizace, databanka kookurencí – předpřipravený seznam výskytových vzorů pro cca 220000 lemmat
- Český národní korpus
  - diachronní část 13-19.století — DIAKORP
  - Synchronní část — cca od 1900
  - psaný jazyk – 100MW v SYN 2000/2005 (“vyvážený”), 700MW SYN2009PUB (pouze publicistika)
  - mluvený jazyk – Pražský mluvený korpus (PMK), Brněnský mluvený korpus (BMK)
  - dialekty

Paralelní korpusy

- text a jeho překladové ekvivalenty v jednom nebo několika jazycích (hub language)
- přidaná hodnota — párování (alignment)
- InterCorp (ÚČNK)
  - čeština + cca 20 evropských jazyků, některé z nich s morfologickou anotací
- MULTEXT-EAST
  - Multilingual Text Tools and Corpora for Central and Eastern European Languages
  - překlad “1984” od George Orwella, kolem 100kW
  - bulharština, čeština, estonština, maďarština, rumunština, slovinština a angličtina (hub language) (a nedávno také chorvatština, litevština, rumunština, ruština)
  - manuálně ověřené zarovnání vět
- Europarl
  - texty extrahované ze sborníků Evropského parlamentu
  - 11 paralelních jazyků: Romanic (French, Italian, Spanish, Portuguese), Germanic (English, Dutch, German, Danish, Swedish), Greek and Finnish
  - až 50 MW na jazyk
- JRC-Acquis
  - Dokumenty evropská komise, 23 jazyků

  - Automatický alignment, často dost hrubý

- PCEDT
  - český překlad 21,600 anglických vět z části obsahující texty z *Wall Street Journal* z *Penn Treebank 3* korpusu
  - automatická morfologická anotace a parsování na analytickou a tektogramatickou úroveň

- CzEng
  - česko-anglický automaticky anotovaný korpus (8 MS, cca 80 MW)
  - cca polovina textu jsou filmové titulky (3 MS); zbytek pak legislativa EU (1,5 MS), technická dokumentace (1 MS), romány (1 MS), paralelní webové stránky (0,5 MS), novinové články

Treebanky

- **treebank**
  - databáze syntaktických stromů
  - korpus obsahující informaci o morfologické a syntaktické (a někdy i sémantické) struktuře

Penn Treebank

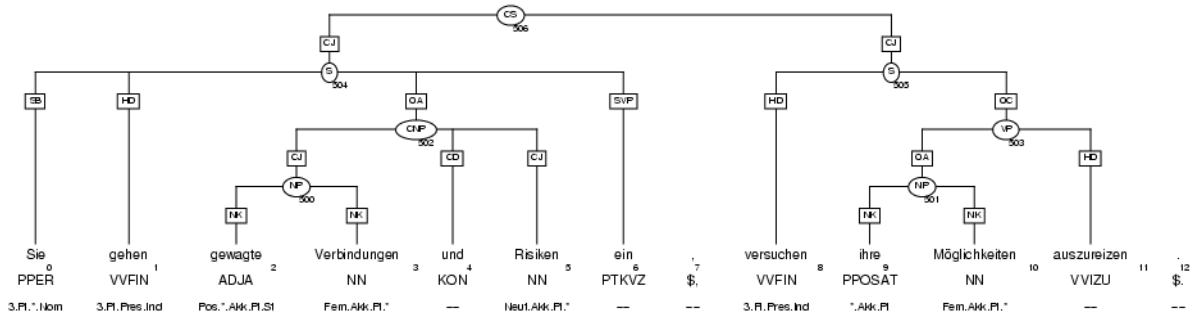
- obsahuje 1,5 MW anglických novinových článků
- syntax bezprostředních složek

PropBank / NomBank

- přidání sémantické vrstvy jako nadstavby nad Penn Treebank – propozice (predikát a argumenty) pro slovesa, resp. substantiva

```
Ukázka anotace: Mr.Bush met him privately, in the White House, on Thursday.
Rel: met
Arg0: Mr.Bush
Arg1: him
ArgM-MNR: privately
ArgM-LOC: in the White House
ArgM-TMP: on Thursday
```

NEGRA



Obrázek 20.1: Příklad stromu z NEGRA Treebank

- 350 kW novinových textů v němčině, syntaktická anotace podobná PTB

Tiger

- 700 kW novinových textů v němčině
- větší velikost než NEGRA treebank, jiná lematizace a morfologie, sekundární hrany (koordinace)

BulTreeBank

- bulharština (200 kW) anotovaná pomocí HPSG formalismu — složkové stromy

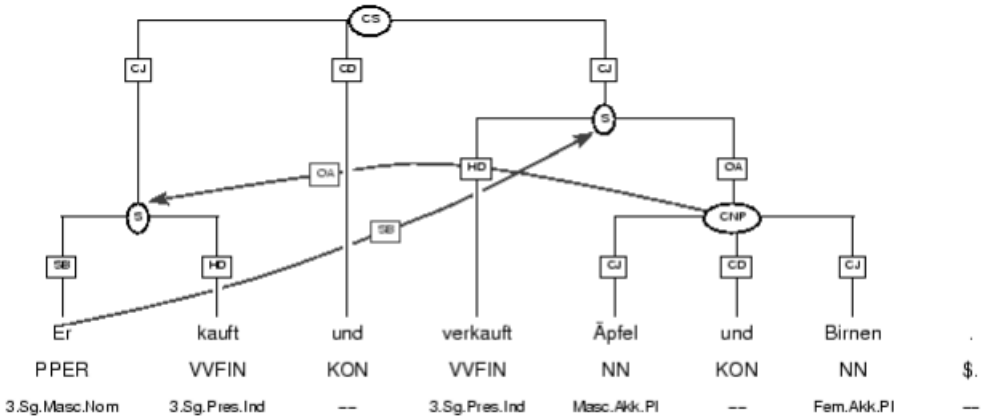


Abbildung 2: Annotation von Koordination durch sekundäre Kanten

Obrázek 20.2: Příklad stromu z Tiger Treebank se sekundárními hranami

- 4 typy prvků: lexikální (N, V, Prep, ...), frázové (VPAdjunct, NPComplement,...) , funkcionální (Conj, CongArg,...) , textové (vlastní řetězce)

Szeged Treebank

- syntaktické struktury vyvinuté pro maďarštinu
- 1,2 MW, ruční morfologická disambiguace a syntaktická anotace

Penn Chinese Treebank
Pražský závislostní korpus

- neboli Prague Dependency Treebank (PZK, PDT)
- obsahuje velké množství českých textů doplněných rozsáhlou a provázanou morfologickou (2 MW), syntaktickou (1,5 MW) a sémantickou (0,8 MW) anotací
- na sémantické rovině jsou navíc anotovány aktuální členění věty a koreferenční vztahy

20.5 Počítačová lexikografie

- vytváření strojově čitelných slovníků za účelem jejich využití v různých úlohách NLP
- ruční tvorba nebo automatická extrakce
- problémy při tvorbě slovníků:
  - neúplnost slovníku — opomenutí nebo přehlédnutí některého hesla
  - jaký zvolit přístup k vlastním jménům — vynechat nebo zahrnout?
  - *ghost words* – neexistující slova, ve slovníku omylem (př. *Dord* = density ('D or d'))
  - rozlišení významu hesel (P.Hanks: *A serious problem for computer applications is that dictionaries compiled for human users focus on giving lists of meanings for each entry, without saying much about how one meaning may be distinguished from another in text.*)
- rozlišování hesel
  - **homografie** — vlastnost dvou nebo více termínů, které mají stejnou grafickou formu, ale rozdílnou výslovnost — např. *pro-udit* vs. *proud-it*
  - **polysémie** — jednomu lexému odpovídají dva nebo více významů (v souvislosti s jejich *výskytem v různých kontextech*), např. *vyšel z místnosti* vs. *vyšel z předpokladu*
  - **homonymie** — *šetřit* (= *spořit*) vs. *šetřit* (= *zjišťovat*)
  - pozor — homonymie a polysémine nejsou to samé!
  - P.Hanks: *No generally agreed criteria exist for what counts as a sense, or for how to distinguish one sense from another*

Slovníky

- Longman Dictionary of Contemporary English (LDOCE) – three-level embedded structure for sense distinctions (homographs,senses,optional subsenses)
- Roget's Thesaurus (1852)
- Cambridge International Dictionary of English
- COBUILD English Language Dictionary
- WordNet
- VerbNet
  - největší on-line slovník anglických sloves
  - rysy: hierarchický, doménově nezávislý (vyvážený), široké pokrytí
  - propojení do PropBank (90% pokrytí), WordNetu, FrameNetu

FRAMENET ANNOTATION:

[Goods A car] was *bought* [Buyer by Chuck].

[Goods A car] was *sold* [Buyer to Chuck] [Seller by Jerry].

[Buyer Chuck] was *sold* [Goods a car] [Seller by Jerry].

PROPBANK ANNOTATION:

[Arg1 A car] was *bought* [Arg0 by Chuck].

[Arg1 A car] was *sold* [Arg2 to Chuck] [Arg0 by Jerry].

[Arg2 Chuck] was *sold* [Arg1 a car] [Arg0 by Jerry].

Obrázek 20.3: Rozdíly v anotaci mezi ProbBank a FrameNet.

- FrameNet
  - on-line zdroj anglických slov — slovesa, podstatná jména, přídavná jména, předložky
  - cílem je dokumentovat sémantické a syntaktické možnosti kombinace možných významů jednotlivých slov
  - **sémantické rámce**, **sémantické role**
- PDT-VALLEX
  - valenční rámce / významy pro slovesa, podstatná jména a přídavná jména
  - významy, které se vyskytly v textech v PDT (bottom-up přístup — důraz na pokrytí textu)
- VALLEX
  - valenční slovník jako zdroj syntakticko-sémantické informace
  - valenční rámce / významy pro slovesa — komplexní zpracování sloves (všechny významy daného slovesa — top-down přístup)

20.6 Wordnety

Princeton WordNet

- lexikální sémantická síť strukturovaná okolo pojmu *synset*
- **synset** — soubor literálů se stejným slovním druhem zaměnitelných v určitém kontextu (množina synonym)
- inspirován psycholingvistickou teorií lidské lexikální paměti
- příliš detailní pro běžné NLP úlohy
- vztahy mezi synsety: homonymie, hyperonymie, meronymie, ...

EuroWordNet

- vícejazyčná databáze obsahující několik jednojazyčných wordnetů strukturovaných podobným způsobem jako Princeton WordNet
- angličtina, holandština, němčina, španělština, francouzština, italština, čeština, estonština

Sense tagged corpora

- **sense tagging** — přiřazování významu z nějakého slovníku slovům v textu
  - je potřeba nějaký **sense-enumerative** slovník, který zachycuje všechny možné významy slova
- úloha **WSD** — word sense disambiguation
- *interest corpus*
  - 2 kS obsahující slovo *interest*
- SENSEVAL
  - organizace pořádající vyhodnocovací soutěže pro WSD (word sense disambiguation) systémy
- SEMCOR
  - více než 200 kW z anglického Brownova korpusu s označkovaným významem podle Princeton WordNetu 1.6

20.7 Poznámky

- kW, MW, GW — tisíce, miliony, miliardy slov
- kS, MS, GS — tisíce, miliony, miliardy vět

Kapitola 21

Státnice I3: Metody strojového učení

21.1 Úvod

Formálně se strojové učení definuje následovně: Počítač se *učí* řešit úlohu třídy T se zkušeností E a úspěšností měřenou kritériem P, jestliže se jeho úspěšnost se vzrůstající zkušeností zvyšuje.

Př.: T = parsing, E = treebank, P počet správně určených hran.

Je třeba určit druh znalostí, které se bude počítač učit, a jejich reprezentaci a navrhnout konkrétní algoritmus učení. Typicky se jedná o *aproximaci* nějaké *cílové funkce*, která realizuje úkol, jenž chceme počítač naučit (reálná funkce – *regrese*, diskrétní – *klasifikace*).

Zkušenost může být *přímá* (zápis pravidel) a *nepřímá* ("dobré" postupy na základě trénovacích dat). Učení může být *supervised* (s učitelem, kdy E = trénovací data), *semi-supervised* (systém generuje postupy a něco je známkuje) nebo *neřízené*.

21.2 Učení založené na konceptu

Učení se konceptů lze definovat jako získání vymezení nějakého konceptu na základě vzorku *pozitivních* a *negativních trénovacích příkladů (instancí) – trénovacích dat* $D = \langle x_i, y_i \rangle_{i=1}^{n}$, kde $X = x_1, x_2, \ldots x_n$ jsou *atributy* a $y_1 \ldots y_n \in Y = \{0, 1\}$ je klasifikace instancí. Snažíme se najít *cílovou funkci* $c : X \to Y, c(x_i) = y_i$. Hledáme ji nad *prostorem hypotéz* (modelem) H; tj. hledáme $h \in H : h(x_i) = c(x_i)\ \forall i \in 1, \ldots n$.

Učení se konceptů můžeme chápat *jako vyhledávání*. V množině hypotéz se nachází *prostor možností* (*version space*) – podmnožina hypotéz *konzistentních* s trénovacími daty (množina cílových funkcí, které klasifikují trénovací data správně) $VS_{H,D}$ a v této podmnožině se někde nachází *optimální* hypotéza. Pomocí strojového učení se jí přibližujeme, od *startovací* hypotézy se postupně dostaneme do něčeho, co je konzistentní s trénovacími daty, ale ne nutně k optimu.

Hledáme takovou hypotézu, která by odpovídala cílové funkci na všech trénovacích příkladech: $h \in H : \forall x \in X : h(x) = c(x)$. Hypotéza, která je "vhodně dobrou" aproximací cílové funkce nad trénovacími daty, bude i "vhodně dobrou" aproximací nad ostatními daty.

Uspořádání hypotéz

Zvolme si reprezentaci hypotéz jako n-tici "povolených" hodnot proměnných z trénovacích příkladů, přičemž pro hypotézu může být hodnotou:

- Specifická hodnota dané proměnné (tj. povolená je jedna hodnota)
- "?" (tj. na hodnotě této proměnné nezáleží, povolené je všechno)
- "$\emptyset$" (tj. nevyhovuje žádná hodnota této proměnné)

Pokud instance x vyhovuje všemi svými hodnotami proměnných povoleným hodnotám pro hypotézu h, potom $h(x) = 1$, tj. hypotéza h klasifikuje x jako *pozitivní příklad*.

Potom *nejobecnější hypotéza* je taková, která cokoliv klasifikuje jako pozitivní případ, tj. $(?, ?, ?, \ldots, ?)$ a *nejspecifičtější hypotéza* taková, která vše klasifikuje jako negativní, tj. $(\emptyset, \emptyset, \ldots, \emptyset)$.

Pro prohledávání prostoru hypotéz máme vodítko: *uspořádání podle specifičnosti*.

- Hypotéza $h \in H$ je *obecnější než* hypotéza $j \in H$ ($h \geq_g j$), právě když $\forall x \in X : j(x) = 1 \Rightarrow h(x) = 1$ (můžeme říct, že j je *specifičtější než* h). Relace $\geq_g$ je částečné uspořádání nad prostorem H.
- Hypotéza $h \in H$ je *striktně obecnější než* $j \in H$ ($h >_g j$), pokud h je obecnější než j, ale j není obecnější než h.

Ještě definujme *hranici obecnosti (general boundary)* G vzhledem k H a D jako množinu nejobecnějších hypotéz z H konzistentních s D. Proti tomu *Hranice specifičnosti (specific boundary)* S vzhledem k H a D je množina nejspecifičtějších hypotéz z H konzistentních s D.

Algoritmus Find-S

Nalezne nejspecifičtější hypotézu, která ještě vyhovuje trénovacím datům.

1. Vezmi nejspecifičtější hypotézu $h \in H$
2. $\forall x$ pozitivní trénovací příklad a $\forall a_i$ atribut reprezentace h: pokud hodnota a_i u x neodpovídá h, nahraď hodnotu a_i v h nejbližší obecnou hodnotou, která odpovídá x
3. Po projití všech pozitivních příkladů vrať upravené h

Find-S vrátí nejspecifičtější hypotézu konzistentní s pozitivními trénovacími příklady. Za předpokladů, že prohledávaný soubor hypotéz obsahuje cílovou funkci c a trénovací data neobsahují chyby, bude nalezená hypotéza konzistentní i s negativními trénovacími příklady, tj. s celými trénovacími daty.

Je-li víc správných hypotéz, nalezneme takto jen jednu z nich, navíc neodhalíme nekonzistenci trénovacích dat. Nemusí existovat ani jen jedna nejspecifičtější hypotéza.

Problém nalezení všech konzistentních hypotéz řeší algoritmus Candidate-Elimination.

Algoritmus Candidate-Elimination

Algoritmus pracuje s prostorem možností, určeným hranicí obecnosti a hranicí specifičnosti. To je možné proto, že platí následující věta:

Věta o reprezentaci prostoru možností: $VS_{H,D} = \{h \in H : \exists s \in S, \exists g \in G : (g \geq_g h \geq_g s)\}$.

Postup algoritmu:

1. Do G, S přiřaď množinu nejobecnějších, resp. nejspecifičtějších hypotéz z H
2. Postupuj přes trénovací příklady $d \in D$:
3. Je-li příklad $d \in D$ pozitivní:
   (a) Odstraň z G hypotézy nekonzistentní s d.
   (b) Každou s d nekonzistentní hypotézu z S odstraň a přidej do S její minimální zobecnění konzistentní s d, pro které existuje nějaká obecnější hypotéza v G. Odstraň pak z S hypotézy, které jsou obecnější než jiné hypotézy v S.
4. Je-li příklad $d \in D$ negativní:
   (a) Odstraň z S hypotézy nekonzistentní s d.
   (b) Každou s d nekonzistentní hypotézu z G odstraň a přidej do G její minimální specializaci konzistentní s d, pro kterou existuje nějaká specifičtější hypotéza v S. Odstraň pak z G hypotézy, které jsou specifičtější než jiné hypotézy v G.
5. Na konci vrať G, S jako reprezentaci prostoru možností.

Induktivní předpoklad

Různé algoritmy různým způsobem zobecňují údaje získané z trénovacích dat, aby byly schopny klasifikovat i dříve neviděné příklady. Pro zachycení této strategie definujme:

- Je-li algoritmus L natrénovaný na datech D_c schopný klasifikovat instanci x_i, je tato klasifikace (ozn. $L(x_i, D_c)$) *induktivně odvoditelná* z oné instance a trénovacích dat. Píšeme $(D_c \wedge x_i) \triangleright L(x_i, D_c)$. To ale ještě neznamená, že je z instance a dat dokazatelná – algoritmus v sobě může obsahovat navíc induktivní předpoklad.
- *Induktivní předpoklad (inductive bias)* algoritmu L je minimální množina tvrzení B taková, že pro libovolnou cílovou funkci c definovanou nad instancemi X a odp. trénovací data D_c platí: $\forall x_i \in X : (B \wedge D_c \wedge x_i) \vdash L(x_i, D_c)$, tj. klasifikace x_i algoritmem L natrénovaným na D_c je dokazatelná z instance, trénovacích dat a induktivního předpokladu.

Algoritmus *Candidate-Elimination* předpokládá, že cílovou funkci lze popsat pomocí naší zvolené reprezentace, tj. nachází se v našem prostoru hypotéz: $B = \{c \in H\}$. Bude-li tento předpoklad splněn, pak najde optimální hypotézu. Je jasné, že tento předpoklad splněn nebude, když např. dvě konkrétní hodnoty nějakého atributu budou určovat pozitivní případy a zbytek negativní. *Find-S* oproti *Candidate-Elimination* ještě navíc předpokládá, že všechny příklady jsou negativní, pokud jejich protějšek nepřináší novou znalost.

Kdybychom zvolili reprezentaci hypotéz disjunkcemi všech možných hodnot atributů, tj. nechali induktivní předpoklad prázdný, *Candidate-Elimination* nebude schopný zobecňovat za hranici trénovacích příkladů (nalezená hypotéza bude čistě disjunkcí všech trénovacích příkladů).

Pro algoritmy učení je tedy třeba uvažovat, jak silné mají indukční předpoklady. Ty jsou někdy velice striktní, jako v případě *Candidate-Elimination*, někdy znamenají jen preferenci určitého typu hypotéz (např. co nejkratších, podle kritéria *Occamovy břitvy*).

21.3 Rozhodovací stromy

Rozhodovací stromy jsou induktivní algoritmy, vytvořené ke klasifikaci instancí, které popisují pomocí *"disjunkce konjunkcí"*. Můžeme je použít, když instance jsou popsány atributy a hodnotami a cílová funkce je *diskrétní*. V trénovacích datech můžou být i chyby nebo nevyplněné atributy.

Cílová funkce je tu reprezentována právě *rozhodovacím stromem* (sekvencí rozhodnutí `if-then-else`), klasifikace se provádí průchodem stromu od kořene k listům, přičemž se v každém uzlu testuje hodnota jednoho atributu instance a na jejím základě se určí další směr (zvolí se hrana). Každý list určuje klasifikaci instance.

ID3 algoritmus

ID3 je algoritmus, který vytváří rozhodovací stromy na základě trénovacích dat. Postupuje *top-down* – konstruuje strom od kořene. V každém uzlu si klade otázku: *Který atribut je nejlepší pro tento uzel?* a řeší ji statistickým testem na základě maximální entropie. Počet větví vedoucích z uzlu je počet možných hodnot atributu, takže ho známe hned po vytvoření uzlu. Trénovací data jsou při vytváření dělena podle atributů v daném uzlu.

Pracuje rekurzivně:

1. Vyřeší singulární případy:
   (a) Všechna trénovací data jsou pozitivní případy / negativní případy: vytvoří jednouzlové stromy s *(+)* nebo *(-)*.
   (b) Nemáme (už) žádné atributy: vytvoří jednouzlový strom s *(nejčastější hodnota cíl. funkce v relevantní části trénovacích dat)*.
2. Vybere atribut A, který nejlépe klasifikuje trénovací data a:
   (a) Pro každou jeho možnou hodnotu udělá poduzel, rozdělí trénovací data podle těchto hodnot.
   (b) Máme-li pro danou hodnotu atributu trénovací data, volá se rekurzivně (na danou část trénovacích dat a atributy s vynechaným A).
   (c) Pokud nejsou trénovací data, vloží list s *(nejčastější hodnota cíl. funkce v celých trénovacích datech)*.
3. Vrátí výsledný (pod)strom.

Prochází se celý hypothesis space, ale hladovým způsobem bez backtrackingu. Výstupem je 1 rozhodovací strom, tj. jedna hypotéza.

Výběr nejlepšího atributu probíhá na základě *informačního zisku*. Který atribut má nejvyšší informační zisk (míra očekávaného snížení entropie, vlastně vzájemná informace cílové klasifikace a atributu), ten je vybrán. Platí: $G(D, A) = H(D) - \sum_{v \in \text{values}(A)} (\frac{|D_v|}{|D|} \cdot H(D_v))$, kde:

- $G(D, A)$ je informační zisk pro data D a atribut A,
- $H(D)$ je entropie dat D vzhledem k cílové funkci,
- D_v jsou data, která mají hodnotu atributu A rovnou v.

ID3 prochází úplný prostor hypotéz tím, že začíná od nejjednodušší a postupně ji zesložiťuje, netestuje ale zdaleka všechny možnosti. Na rozdíl od předchozích algoritmů v každém kroku uvažuje celá trénovací data. Induktivní předpoklad ID3 je (přibližně) preference kratších stromů.

Vylepšení ID3 (zhruba odpovídá algoritmu C4.5)

Jedním z možných problémů je *overfitting*, tj. přílišné přizpůsobení se trénovacím datům. Řekneme, že hypotéza $h \in H$ *overfituje* trénovací data, pokud existuje hypotéza $h' \in H$ taková, že h má menší chybu než h' na trénovacích datech, ale na celé distribuci instancí je to naopak.

Vyvarovat se overfittingu lze *prořezáváním (pruning)* – nahrazením některých podstromů listem s nejčastější hodnotou cílové funkce na trénovacích datech. Lze provádět buď rovnou zastavením tvorby stromu dřív, než dosáhne ideální klasifikace trénovacích dat, nebo zpětně strom zjednodušit za použití *heldout dat* pro validaci při trénování (*reduced error pruning*).

Složitější je strategie *rule-post-pruning* spočívá v následujícím postupu:

1. Vytvoření rozhodovacího stromu.
2. Jeho převedení na pravidla (počet pravidel = počet listů), kdy 1 pravidlo je 1 cesta od listu ke kořeni.
3. Pro každé pravidlo zahodit ty podmínky, jejichž vyhozením se přesnost pravidla zlepší.
4. Pravidla setřídit podle přesnosti na trénovacích datech, reálná data klasifikovat použitím pravidel v tomto pořadí.

Proti prořezávání nad stromem nemusíme vyhazovat jen z konce stromu, máme jemnější rozlišení.

Je-li některý atribut A *spojitý* (ale cílové ohodnocení stále diskrétní), lze ho převést na diskrétní nalezením jednoho předělu, pro který dostaneme nejvyšší *informační zisk*.

Má-li atribut více hodnot (např. nějaké datum), informační zisk ho bude preferovat, což často vede k vytvoření stromu hloubky 1, který funguje jen na trénovací data. Musíme pak změnit test výběru atributů, místo zisku informace používáme např. *ziskový poměr (gain ratio)*:

- *Rozštěpení informace (split information)*: $SI(D,A) = -\sum_{v \in \text{values}(A)} \frac{|D_v|}{|D|} \log_2(\frac{|D_v|}{|D|})$
- *Ziskový poměr*: $GR(D,A) = \frac{G(D,A)}{SI(D,A)}$, kde $G(D,A)$ je informační zisk pro trénovací data D a atribut A

Jsou i jiné alternativní způsoby výběru nejlepšího atributu, např. *distance-based measure*, kdy se vybírá atribut, který dělí data co nejrovnoměrněji podle nějaké metriky (směrodatná odchylka, Gini koeficient).

Neznáme-li hodnotu některého z atributů, můžeme mu přiřadit nejčastější hodnotu, nebo rozhodování rozdělit podle všech možných hodnot a na konci udělat průměr výsledků vážený podle jejich pravděpodobnosti výskytu.

Pokud jsou některé atributy důležitější než jiné, zavádí se *cena atributu (cost of the attribute)* a jiné kritérium výběru atributu.

21.4 Neuronové sítě

Vznik neuronových sítí je motivován biologicky, funkcí mozku jako spojení mnoha neuronů. *Umělé neuronové sítě* ale jsou poněkud jednodušší – jedná se o propojenou množinu jednotek (*neuronů*), z nichž každá z několika reálných hodnot na vstupu vydá jednu reálnou hodnotu na výsupu. Živé neurony proti tomu vydávají mnohem složitější elektrické signály.

Neuronová síť bývá typicky zapojená jako orientovaný acyklický graf, ale nemusí to být nutné. Tady se ale omezíme jen na takové sítě, protože na ně známe jednoduchý algoritmus trénování – *backpropagation*.

Neuronové sítě jsou vhodné pro složitá a potenciálně zašuměná vstupní data, jako např. vstupy z kamer a mikrofonů (rozpoznávání řeči, rozpoznávání obličejů apod.). Jsou ale použitélná i pro data se symbolickou reprezentací, kde dosahují výsledků srovnatelných s rozhodovacími stromy. Cílová funkce může být reálná i diskrétní a může jít i o vektor hodnot. Trénování zpravidla trvá dlouho, ale klasifikace nových instancí je velmi rychlá. Natrénované hodnoty se často nedají dobře interpretovat, ale fungují.

Perceptron

Perceptron je vlastně jeden druh neuronu často používaného v neuronových sítích. Jde o jednotku, která ze vstupu $x_1, \ldots, x_n$ spočítá hodnotu výstupu $o(x_1, \ldots, x_n)$. Podle druhu výpočtu rozlišujeme :

- Lineární perceptron, který počítá lineární kombinaci vstupů na základě natrénovaných vah $-w_0 + w_1 x_1 + \ldots w_n x_n$. Můžeme přidat $x_0 = 1$ a zapsat jeho výpočet jako $o(\vec{x}) = \vec{w} \cdot \vec{x}$.
- (Základní) prahový perceptron, který počítá funkci $o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$ a vrací diskrétní hodnoty $-1, 1$.
- Sigmoidovou jednotku, který používá sigmoidovou (logistickou) funkci $\sigma(y) = \frac{1}{1+e^{-y}}$ a počítá $o(\vec{x}) = \sigma(\vec{w} \cdot \vec{x})$. I když funguje na stejném principu jako dva předchozí a vlastně dává podobné výsledky jako prahový perceptron (její obor hodnot je interval $[-1, 1]$), podle jména se mezi perceptrony nepočítá.

Jak je vidět, prostor hypotéz perceptronu je množina všech možných reálných vah ($H = \{\vec{w} | \vec{w} \in \mathbb{R}^{n+1}\}$).

Uvažujme nyní prahový perceptron. Ten se vlastně dá považovat za reprezentaci "rozhodující nadroviny" v n-dimenzionálním prostoru instancí (to jsou vlastně body v n-rozměrném prostoru). Vrací 1 pro body, které leží na jedné straně nadroviny a -1 pro ty na druhé straně. To ale znamená, že správně umí klasifikovat jen problémy, kde opravdu pozitivní a negativní příklady jdou oddělit nadrovinou. Takovým problémům (a množině příkladů) se říká *lineárně separovatelné*.

Př. Perceptron se umí naučit booleovské funkce AND a OR, ale neumí XOR, protože není lineárně separovatelná.

Následují dvě možnosti, jak perceptron natrénovat. Další alternativou by bylo i lineární programování, ale to funguje jen v lineárně separovatelném případě a navíc se nedá použít jako základ pro složitější sítě.

Perceptron Training Rule

Nejjednodušším způsobem, jak natrénovat váhy (prahového) perceptronu, je použít *perceptron training rule* – začneme s náhodnými vahami a iterativně zkoušíme klasifikovat trénovací příklady a jakmile dojde ke špatné klasifikaci, váhy změníme. Formálně v každém kroku provedeme:

$$w_i := w_i + \Delta w_i, \text{ kde } \Delta w_i = \eta(t - o)x_i$$

Hodnota o je výstup perceptronu a t správná klasifikace. Hodnotě η říkáme *learning rate* – určuje, jak ostře nové příklady ovlivňují původní hodnoty vah. Většinou je nastavená na nějakou malou hodnotu, můžeme ji i snižovat s počtem iterací.

V případě, že trénovací příklady jsou lineárně separovatelné, takovýto algoritmus po konečném počtu iterací dojde k vektoru vah, se kterým perceptron správně klasifikuje všechny trénovací příklady.

Gradient Descent (Delta Rule)

Jiný způsob je *gradient descent* (klesání podle gradientu). Uvažujme nyní lineární perceptron, abychom mohli snadno derivovat. Stanovíme si funkci, která hodnotí *trénovací chybu* hypotézy (tj. vektoru vah) $E(\vec{w})$ a iterativně budeme hledat, kterým směrem jde gradient a vydáme se přesně opačným (tj. proti směru největšího růstu funkce). Tak dojdeme pro "hezké funkce" do místa, kde je gradient nulový a tedy je tam (bohužel jen lokální) minimum chybové funkce. Pro lineárně separovatelné trénovací instance bychom se měli dostat k nulové chybě, ale i pro neseparovatelné dostaneme (snad) to nejlepší, co můžeme. Velmi vhodnou funkcí trénovací chyby je:

$E(\vec{w}) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$, kde D jsou trénovací data a t_d, o_d skutečná hodnota a výstup perceptronu jako v předchozí sekci.

Potom metoda *gradient descent* postupuje stejně iterativně jako v předchozím případě:

$w_i := w_i + \Delta w_i$, kde $\Delta w_i = -\eta\nabla E(\vec{w})$ (bereme záporný gradient, takže jdeme opačným směrem)

Pokud si upravovací pravidlo rozložíme na jednotlivé složky, můžeme vypočítat:

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} \text{ a } \frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 = \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) = \sum_{d\in D}(t_d - o_d)(-x_{id})$$

x_{id} značí vstup x_i v trénovacím příkladu D. Dosazení do původní rovnice dává pravidlo:

$$\Delta w_i = \eta\sum_{d\in D}(t_d - o_d)x_{id}$$

Pro zrychlení výpočtu a snížení rizika "zamrznutí" v lokálním minimu se používá *stochastická aproximace* (*inkrementální* nebo *stochastický* gradient descent), kdy váhy upravuji ne podle gradientu na celých trénovacích datech, ale jen s přihlédnutím k jednomu trénovacímu příkladu :

$$\Delta w_i = \eta(t - o)x_i$$

Protože prahový perceptron čistě vrací znaménko výstupu lineárního perceptronu, je možné perceptron natrénovat touto metodou jako lineární a pak použít prahy (ale pozor, neminimalizujeme počet špatně klasifikovaných příkladů prahového perceptronu, ale sumu chyby lineárního perceptronu, což nemusí vést ke stejným výsledkům).

Neuronová síť

Perceptrony se dají spojovat do sítí a potom jsou schopny se naučit i lineárně neseparovatelné problémy (to ale neplatí o lineárních perceptronech – lineární kombinace lineárních kombinací je pořád lineární kombinace). Protože síťovat lineární perceptrony tedy nemá smysl a prahové perceptrony používají funkci, která se nedá dobře derivovat, použijeme sigmoidové jednotky. Derivace sigmoidové funkce je totiž $\sigma'(y) = \sigma(y)(1-\sigma(y))$. Někdy se používá i varianta sigmoidové funkce s $e^{-k\cdot y}$ nebo funkce tanh.

Takovéto jednotky můžeme tedy zasadit do sítě ve tvaru acyklického grafu. Trénování vah ale pak bude složitější, používá se na to algoritmus *backpropagation*.

Algoritmus backpropagation

Máme-li pevně dánu množinu jednotek a jejich spojení, která má libovolný počet vstupů a výstupů, můžeme natrénovat váhy každého vstupu pomocí algoritmu *backpropagation*. Používáme přitom stejných myšlenek jako u gradient descent, když se snažíme minimalizovat chybovou funkci:

$E(\vec{w}) = \frac{1}{2}\sum_{d\in D}\sum_{k\in \text{Out}}(t_{kd} - o_{kd})^2$ – jde o stejnou funkci jako předtím, jen upravenou pro množinu výstupů Out.

Stejně jako předtím prohlédáváme prostor hypotéz definovaný všemi možnými vektory vah a máme garantovanou konvergenci k lokálnímu minimu chybové funkce. Algoritmus vypadá (ve stochastické aproximaci) následovně (x_{ji} je vstup z i-té jednotky do j-té a w_{ji} odpovídající váha):

1. Inicializuje všechny váhy malými náhodnými čísly
2. Dokud není splněna nějaká podmínka konvergence, opakuje (pro jednotlivé trénovací příklady $\langle \vec{x}, \vec{t} \rangle$, může je procházet i víckrát):
   (a) Propaguje vstup dopředu sítí – vloží $\vec{x}$ na vstup a spočítá výstup o_u z každé jednotky u v síti
   (b) Propaguje chyby a úpravy vektorů vah zpátky:
      i. Pro každou výstupní jednotku k spočte chybu $\delta_k = o_k(1 - o_k)(t_k - o_k)$
      ii. Pro každou vnitřní (skrytou) jednotku h spočte chybu $\delta_h = o_h(1 - o_h) \sum_{j \in \text{Downstream}(h)} w_{jh}\delta_j$
      iii. Upraví všechny váhy v síti $w_{ji} = w_{ji} + \Delta w_{ji}$, kde $\Delta w_{ji} = \eta \delta_j x_{ji}$

Downstream(h) značí všechny jednotky, jejichž vstupy jsou přímo napojené na výstup jednotky u. Pokud jednotky nejsou spojené cyklicky, lze najít takové pořadí jednotek, ve kterém lze spočítat hodnotu δ_h pro všechny.

Pro upravovací pravidla vycházíme z gradientové metody, která by pro složitější síť měla ve stochastické variantě být $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$ pro naši chybovou funkci definovanou výstupem pro instanci d.

- Protože w_{ji} ovlivňuje zbytek sítě jen prostřednictvím uzlu j, můžeme použít řetízkové pravidlo a $\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial \text{In}(j)} \frac{\partial \text{In}(j)}{\partial w_{ji}} = \frac{\partial E_d}{\partial \text{In}(j)} x_{ji}$, kde $\text{In}(j) = \sum_{i \in \text{Upstream}(j)} w_{ji}x_{ji}$ a Upstream(j) jsou všechny jednotky, jejichž výstupy jsou napojené na vstup j-té jednotky. Definujeme $\delta_j = -\frac{\partial E_d}{\partial \text{In}(j)}$.
- Pro výstupní jednotky z podobné úvahy vyplyne, že $\frac{\partial E_d}{\partial \text{In}(j)} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial \text{In}(j)}$ a z toho – z definice chybové funkce (všechny části její sumy krom j-tého výstupu budou mít nulové derivace) a z definice sigmoidové funkce ($o_j = \sigma(\text{In}(j))$) – spočteme $\delta_j = (t_j - o_j)o_j(1 - o_j)$.
- Podobně i pro skryté jednotky $\frac{\partial E_d}{\partial \text{In}(j)} = \sum_{k \in \text{Downstream}(j)} \frac{\partial E_d}{\partial \text{In}(k)} \frac{\partial \text{In}(k)}{\partial o_j} \frac{\partial o_j}{\partial \text{In}(j)}$, protože skrytá jednotka ovlivňuje chybu jen přes své přímé výstupy; dostáváme rekurentní vzorec. Protože jediný vstup do k, který se mění v závislosti na j, je $w_{kj}x_{kj}$, ale vlastně $x_{kj} = o_j$, je $\frac{\partial \text{In}(k)}{\partial o_j} = w_{kj}$. Poslední člen řetízku spočteme stejně jako pro výstupní jednotky a z toho $\delta_j = o_j(1 - o_j) \sum_{k \in \text{Downstream}(j)} \delta_k w_{kj}$.

Vlastnosti neuronových sítí

Induktivní předpoklad algoritmu *backpropagation* se dá stanovit přesně jen těžko, intuitivně jde ale o cca lineární interpolaci mezi trénovacími příklady jako body n-rozměrného prostoru.

Garance pouze lokálního minima v praxi až tolik nevadí – díky tomu, že máme víc vah, je také méně pravděpodobné, že bude lokální minimum ve všech rozměrech vektoru. Inicializujeme-li malými hodnotami vah, bude výstup díky vlastnostem sigmoidové funkce přibližně lineární kombinace, až se zvýšením vah dostaneme výraznou nelinearitu.

Varianta algoritmu ještě přidává *moment* (α) a úprava vah částečně závisí na předchozí iteraci, čímž se snaží předejít zamrznutí v lokálním minimu (má pak určitou "setrvačnost"):

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n - 1)$$

Můžeme zkusit také natrénovat více sítí s lehce odlišnými počátečními vahami a vybrat si tu nejlepší.

Neuronová síť rozdělená do vrstev, v rámci nichž nejsou spoje, je schopná:

- S dvěma vrstvami přesně simulovat libovolnou boolovskou funkci (až na to, že počet potřebných jednotek může být až exponenciální v počtu vstupů)
- Spojité funkce lze aproximovat dvěma vrstvami do libovolné přesnosti, když první vrstva jsou sigmoidy a druhá lineární perceptrony.
- Libovolná funkce je aproximovatelná třema vrstvami – dvěma sigmoidovými a jednou lineární.

Konvergence a overfitting

Neuronová síť má velkou volnost v nastavování vah, proto může až příliš přesně aproximovat trénovací data, čímž se ztrácí použitelnost na data nová. Podmínku konvergence je třeba volit "rozumně":

- Samotné čekání, až změny vah budou dostatečně malé, většinou nestačí.
- Je vhodné např. penalizovat příliš velké váhy (po každé iteraci je trošku snižovat), čímž podporujeme "lineárnější" rozhodovací funkce.

- Velmi úspěšné je kromě trénování zároveň testovat síť na další datové množině a pamatovat si váhy, které na ní měly nejmenší chybu. Pokud se chyba na této množině výrazně zvýší oproti nejlepší hodnotě, trénování skončí a vezmeme váhy s nejmenší chybou na ladící množině.
- Předchozí metodu lze provádět i *křížovou validací* (rozdělím trénovací data na k množin a použiji v každém kroku $k-1$ z nich na trénování a zbylou na testování)

21.5 Učení založené na příkladech

Hlavní myšlenka *učení založeného na příkladech (instance-based learning)* zní: který trénovací příklad je nejbližší aktuální testova
V paměti vždy uchováváme všechny trénovací příklady, kdykoliv lze přidat další a pro každý testovací příklad v nich hledám ty nejpodobnější. Proto je nutné zavést *míru závislosti* příkladů. Lze např. použít euklidovskou míru $d(x,y) = \sqrt{\sum_{i=1}^{n}(a_i(x) - a_i(y))^2}$, ale není to nutné. Cílová funkce může být diskrétní (s def. oborem $V = \{v_1, \ldots, v_n\}$) nebo spojitá.

K-nearest neighbor

K-nearest neighbor nebo *k-nn* je algoritmus typu instance-based learning. Počítám s uloženými trénovacími daty E v paměti. Postup pro nějakou testovací instanci x_q:

- Je-li $x_q \in E$, potom mám přímo hodnocení.
- Jinak najdu k nejbližších $x_1, \ldots, x_k \in E$ a:
  - Pro diskrétní cílovou funkci $\hat{f}(x_q) = \text{argmax}_{v \in V} \sum_{i=1}^{k} \delta(v, f(x_i))$, kde $\delta = 1$ pro stejné hodnoty argumentů, a 0 pro různé (tj. přiřadím x_q takovou hodnotu, kterou má nejvíce z oněch k sousedů).
  - Pro spojitou cílovou funkci $\hat{f}(x_q) = \frac{\sum_{i=1}^{k} f(x_i)}{k}$, tj. udělám průměr hodnocení.

K-nn nikdy nepočítá explicitní obecnou hypotézu, pracuje jen v kontextu. Pro $k = 1$ de facto odpovídá Voronoi diagramu. Vylepšením může být *vážení podle vzdáleností (distance-weighted k-nn)*, kde:

- V diskrétním případě $\hat{f}(x_q) = \text{argmax}_{v \in V} \sum_{i=1}^{k} w_i \cdot \delta(v, f(x_i))$, kde navíc $w_i = \frac{1}{d(x_q, x_i)^2}$.
- Ve spojitém případě $\hat{f}(x_q) = \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$.

Díky vážení můžeme takhle jít dokonce přes celá trénovací data (*globální (Shepardova) metoda*).
Algoritmus je robustní vzhledem k šumu v trénovacích datech. Problémem je pomalý výpočet vzdáleností ke všem trénovacím datům, pro zrychlení se vytváří indexy, pomocí *k-d stromů*. Problém je taky, když mám hodně atributů a relevantních jen pár. Pak je metrika vzdáleností zavádějící – i když se shodují důležité atributy, mohou být od sebe instance v ostatních atributech "daleko". Řešením je váha atributů.

Lokálně vážená lineární regrese

Lokálně vážená lineární regrese (locally weighted linear regression) je konstrukce lokální aproximace cílové funkce na okolí kolem testovacího příkladu. Používá se často ve statistice. Používají se ale i jiné než lineární funkce. Definujeme:

- *Regrese* je aproximace reálné funkce.
- *Reziduum* je odchylka aproximace od reálné funkce, tj. $\hat{f}(x) - f(x)$.
- *Jádro (kernel function)* je funkce K vzdálenosti d za účelem získání váhy trénovacích příkladů $w_i = K(d(x_i, x_q))$. Často je to gaussovská funkce se střední hodnotou μ a rozptylem σ^2.

Aproximace hodnotící funkce se provádí ve tvaru: $\hat{f}(x) = w_0 + w_1 a_1(x) + \cdots + w_n a_n(x)$ (kde $a_i(x)$ je hodnota i-tého atributu příkladu). Úkolem je pak najít taková w_i, která minimalizují chybovou funkci. K tomu slouží metoda *gradient descent*. Chybová funkce se volí tak, aby byla jen *lokální*, máme tu tři možnosti (z nichž ta poslední je nejvýhodnější). Nechť $NN_k(x_q)$ je množina k nejbližších sousedů x_q:

1. $E_1(x_q) = \frac{1}{2} \sum_{x \in NN_k(x_q)} (\hat{f}(x) - f(x))^2$
2. $E_2(x_q) = \frac{1}{2} \sum_{x \in D} (\hat{f}(x) - f(x))^2 \cdot K(d(x_q, x))$ – nejpřesnější, ale výpočetní složitost závisí na velikosti trénovacích dat $|D|$.
3. $E_3(x_q) = \frac{1}{2} \sum_{x \in NN_k(x_q)} (\hat{f}(x) - f(x))^2 \cdot K(d(x_q, x))$ – toto je vhodná aproximace druhého přístupu, výpočetní složitost závisí jen na k.

21.6 Vyhodnocování hypotéz

Hypotéza, kterou nám vydá algoritmus učení, nemusí být vhodná. Je proto potřeba umět hypotézy ověřovat a případně porovnávat – ostatně i některé algoritmy učení (jako *rule-post-pruning* u rozhodovacích stromů) v sobě porovnávání hypotéz přímo obsahují.

Statistická formalizace pro diskrétní hypotézy

Mějme množinu všech možných datových instancí X, kterou budeme považovat za náhodnou veličinu s rozdělením $\mathcal{D}$. Nad instancemi X je definovaná cílová funkce f, kterou aproximujeme hypotézami.

Pro hypotézu h definujme:

- *Chybu na datech (sample error)* $\text{error}_S(h)$ jako část vzorku dat S, které h ohodnotí špatně, tj. $\text{error}_S(h) = r/n$ pro r špatně ohodnocených instací z n celkem.
- *Skutečnou chybu (true error)* $\text{error}_{\mathcal{D}}(h)$ jako pravděpodobnost p, že h ohodnotí špatně náhodně vybranou instanci z $\mathcal{D}$.

Chceme znát skutečnou chybu, ale můžeme ji jen odhadnout za pomoci chyby na datech. Skutečná chyba je vlastně náhodná veličina z alternativního rozdělení s parameterem p. Na základě náhodného výběru (nezávislých stejně rozdělených náhodných veličin) o velikosti n pak tento parametr odhadujeme. Náš *odhad*, funkce náhodného výběru, je další náhodná veličina. U takového odhadu je třeba dbát na:

- *Vychýlení (bias)* – je třeba, aby střední hodnota našeho odhadu se rovnala střední hodnotě původní náhodné veličiny (tj. abychom měli nulový bias).
- *Rozptyl* – čím menší rozptyl, tím lepší odhad, protože tak dostáváme menší očekávanou odchylku od skutečné hodnoty parametru.

Podmínka nezávislosti je důležitá, jinak nelze provést nevychýlený odhad (tj. netestovat na trénovacích datech, protože nejsou nezávislá!).

Vezmeme-li n nezávislých pokusů, které s pravděpodobností p dopadnou špatně, potom počet r pokusů, které selhaly, je binomicky rozdělená náhodná veličina. Odhad p jako r/n je nevychýlený, protože střední hodnota binomicky rozdělené veličiny je $E(r) = np$, tj. $E(r/n) = p$. Víme, že $\text{Var}(r) = np(1-p)$. Rozptyl odhadu $\text{error}_S(h) = r/n$ potom je:

$$\text{Var}(\text{error}_s(h)) = \frac{\text{Var}(r)}{n^2} = \frac{p(1-p)}{n} \approx \frac{\text{error}_S(h)(1-\text{error}_S(h))}{n}$$

(Oboustranný) interval spolehlivosti (pro danou pravděpodobnost) určuje interval, ve kterém se podle našeho odhadu bude s danou pravděpodobností N nacházet skutečná hodnota neznámého parametru – je to interval, ve kterém se na obě strany od našeho odhadu nachází dané procento pravděpodobnostní masy z rozdělení, které má náš odhad. Protože to je pro binomické rozdělení příliš namáhavé určit, pomůžeme si za předpokladu, že náš vzorek je dostatečně velký ($n > 30$), podle Centrální limitní věty aproximací normálním rozdělením a platí:

$$P(\text{error}_{\mathcal{D}}(h) \in \left(\text{error}_S(h) \pm z_N \sqrt{\tfrac{\text{error}_S(h)(1-\text{error}_S(h))}{n}}\right)) = N, \text{ kde } z_N \text{ jsou kvantily normálního rozdělení.}$$

Typicky se používá $N = 0.95$. Je možné používat i jednostranné intervaly spolehlivosti, např. pro ověření, jaká je pravděpodobnost, že chyba bude větší než určitá konstanta.

Rozdíl chyby dvou hypotéz

Pro porovnání dvou různých hypotéz (např. při prořezávání rozhodovacích stromů) se vlastně odhaduje rozdíl jejich chyb $d \equiv \text{error}_{\mathcal{D}}(h_1) - \text{error}_{\mathcal{D}}(h_2)$. To můžeme provést pomocí dvou nezávislých náhodných výběrů S_1, S_2 a $\hat{d} \equiv \text{error}_{S_1}(h_1) - \text{error}_{S_2}(h_2)$. $\hat{d}$ je nevychýlený odhad d.

Pokud uvažujeme dost velké náhodné výběry S_1, S_2, můžeme považovat odhady $\text{error}_{S_1}(h_1)$ a $\text{error}_{S_2}(h_2)$ za přibližně normálně rozdělené náhodné veličiny. Jejich rozdíl pak bude taky normálně rozdělený a roztpyl bude odpovídat součtu rozptylů, což nám dává i intervaly spolehlivosti. V typickém případě $S_1 = S_2$ a pak rozptyl typicky bude menší než tento odhad, ale pořád je možné s ním takto počítat.

Můžeme takto i spočítat, jaká je pravděpodobnost, že jedna z hypotéz dává větší chybu – je to $P(d > 0) = P(\hat{d} < E(\hat{d}) + \hat{d})$, což nám dává jednostranný interval a stačí určit jeho masu pravděpodobnosti.

Porovnávání algoritmů

Chceme-li porovnat nejen hypotézy, ale který ze dvou algoritmů učení L_A, L_B dosáhne v průměru lepší aproximace nějaké cílové funkce f, musíme odhadnout průměrný rozdíl výkonu d přes všechny n-prvkové trénovací množiny z distribuce $\mathcal{D}$. V praxi ale máme jen trénovací data S a testovací data T, takže náš odhad bude:

$$\hat{d} = \text{error}_T(L_A(S)) - \text{error}_T(L_B(S))$$

Pro zlepšení tohoto odhadu se často používá *křížová validace* – rozdělení dat D do k disjunktivních množin a provedení k testů, kdy se pokaždé použije jiná z množin jako testovací data všechny zbylé jako trénovací data. Odhad chyby $\bar{d}$ je průměr chyb $\hat{d}_i$ přes všech k množin. Rozptyl tohoto odhadu se stanoví jako výběrový rozptyl:

$$s_{\bar{d}}^2 = \frac{1}{k(k-1)} \sum_{i=1}^{k} (\hat{d}_i - \bar{d})^2$$

Interval spolehlivosti tohoto odhadu se pak stanoví jako $P(d \in (\bar{d} \pm t_{N,k-1} s_{\bar{d}})) = N$, kde $t_{N,k-1}$ jsou kvantily t-rozdělení o $k-1$ stupních volnosti.

Kromě křížové validace je také možné vybírat testovací podmnožinu z dat, která máme k dispozici, úplně náhodně a opakovat testy libovolněkrát, ale testovací množiny už nebudou nezávislé, protože se dřív nebo později stane, že některé prvky vyberu víckrát.

21.7 Výpočetní aspekty strojového učení

Kapitola 22

Státnice I3: Stochastické metody a jejich aplikace v počítačové lingvistice

22.1 Teorie informace

Pravděpodobnost

Jev, pravděpodobnost

Jev A je podmnožina libovolných výstupů (např. nějakého experimentu) – *sample space* Ω. *Prostor jevů* 2^Ω je množina všech možných jevů.

Pravděpodobnost je zobrazení $p : 2^\Omega \to [0, 1]$, pro kterou platí:

- $p(\Omega) = 1$,
- pro $A_i, i \in I$ takové, že $A_{i_j} \cap A_{i_k} = \emptyset$ platí $p(\cup A_i) = \sum_i p(A_i)$

Důsledky – $p(\emptyset) = 0$, $p(\bar{A}) = 1 - p(A)$, $A \subseteq B \Rightarrow p(A) \leq p(B)$, $\sum_{a\in\Omega} p(a) = 1$.

Spojená a podmíněná pravděpodobnost

Spojená (joint) pravděpodobnost dvou jevů A, B je $p(A, B) = p(A \cap B)$. *Podmíněná pravděpodobnost* (pravděpodobnost A, jestliže B už nastalo) je:

$$p(A|B) = \frac{p(A, B)}{p(B)}$$

Přímým důsledkem je *řetězové pravdilo*: $p(A_1, A_2, \ldots, A_n) = p(A_1|A_2, \ldots, A_n) \cdot p(A_2|A_3, \ldots, A_n) \cdot \cdots \cdot p(A_n)$

Bayesova věta

Bayesova věta vychází z toho, že $p(A|B) \cdot p(B) = p(B|A) \cdot p(A)$ což plyne z $p(A, B) = p(B, A)$. Proto platí:

$$p(A|B) = p(B|A) \cdot \frac{p(A)}{p(B)}$$

Pokud $p(A|B) = p(B)$ jsou A a B *nezávislé jevy.*

Zlaté pravidlo statistického zpracování jazyka se hodí pro zjištění, který jev A je nejpravděpodobnější za předpokladu, že nastalo B a nelze dobře odhadnout $p(A|B)$. Přes všechny jevy A:

$$\arg\max_A p(A|B) = \arg\max_A p(B|A) \cdot \frac{p(A)}{p(B)} = \arg\max_A p(B|A)p(A)$$

Náhodné veličiny, rozdělení

Náhodná veličina je funkce $X : \Omega \to Q$ (často $Q \equiv \mathbb{R}$, nebo jde o spočetnou množinu pro *diskrétní* náhodné veličiny). *Střední (očekávána) hodnota* je průměr náhodné veličiny – v diskrétním případě vychází $E(X) = \sum_{x\in X(\Omega)} x \cdot p_X(x)$.

Rozdělení (distribuce) pravděpodobnosti potom je funkce $p_X(x) = p(X = x) \stackrel{\text{def}}{=} p(A_X)$, kde $A_X = \{a \in \Omega | X(a) = x\}$.

Spojené rozdělení je $p_{X,Y}(x, y)$ a *podmíněné rozdělení* je $p_{X|Y}(x|y)$. Bayesova věta a řetězové pravidlo platí i pro rozdělení.

Častá rozdělení pravděpodobnosti:

- u diskrétních náhodných veličin je nejčastější *binomické* – $p(r|n) = \binom{n}{r}/2^n$
- u spojitých veličin *normální (Gaussovo)* – $p(x|\mu,\sigma) = (\sigma\sqrt{2\pi})^{-1} \cdot \exp(-\frac{(x-\mu)^2}{2\sigma^2})$.

Entropie

Entropie je míra nejistoty. Označuje nejmenší průměrný počet bitů potřebný k zakódování "zprávy" – výstupu nějaké náhodné veličiny. Mějme $p_X(x)$ distribuci náhodné veličiny X, jejíž hodnoty jsou v množině Ω. Potom se entropie spočítá následovně:

$$H(X) = -\sum_{x\in\Omega} p(x)\log_2 p(x)$$

Jednotky entropie jsou *bity* (používáme-li dvojkový logaritmus). Alternativní notace: $H(X) = H_p(X) = H(p) = H_X(p) = H(p_X)$. Platí, že $H(p) = 0$ pokud známe výsledek předem, tj. $\exists x \in \Omega : p(x) = 1 \wedge \forall y \in \Omega, y \neq x : p(y) = 0$. Horní hranice hodnot není, ale $|\Omega| = n : H(p) \leq \log_2 n$.

Alternativní definice: protože $E(X) = \sum_{x\in\Omega} p_X(x) \cdot x$, platí pro náhodnou veličinu $X \to \Omega$, že $H(p_X) = -\sum_{x\in\Omega} p_X(x)\log_2 p_X(x) = \sum_{x\in\Omega} p_X(x)\log_2(\frac{1}{p_X(x)}) = E(\log_2(\frac{1}{p_X(x)}))$.

Perplexity je jiné vyjádření míry nejistoty – $G(p) = 2^{H(p)}$.

Spojená a podmíněná entropie

Spojená entropie dvou náhodných veličin $X \to \Omega$, $Y \to \Psi$ se počítá tak, že považujeme (X,Y) za jeden jev. Potom $H(X,Y) = -\sum_{x\in\Omega}\sum_{y\in\Psi} p(x,y)\log_2 p(x,y)$.

Podmíněná entropie se definuje jako $H(Y|X) \stackrel{\text{def}}{=} \sum_{x\in\Omega} p(x) \cdot H(Y|X=x)$. Protože $p(x)\cdot p(y|x) = p(x,y)$, dá se to ekvivalentně zapsat následujícím způsobem:

$$-\sum_{x\in\Omega} p(x)\sum_{y\in\Psi} p(y|x)\log_2 p(y|x) = -\sum_{x\in\Omega,y\in\Psi} p(x,y)\log_2 p(y|x)$$

Vlastnosti entropie

- Je nezáporná ($p(x) \geq 0$, $\log_2 p(x) \leq 0$).
- *Řetězové pravidlo*: $H(X,Y) = H(Y|X) + H(X)$ (z definice a vlastností logaritmu)
- Podmíněná entropie je menší nebo rovna než nepodmíněná. Z toho $H(X,Y) \leq H(X) + H(Y)$, rovnost pro nezávislé jevy.
- $H(p)$ je konkávní funkce ($\forall x,y \in (a,b), \forall\lambda \in [0,1] : f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda)f(y)$).

Relativní entropie

Kullback-Leibler divergence (distance)/relativní entropie pro odhad pravděpodobnostního rozdělení q a skutečné rozdělení p nad stejným sample space Ω je:

$$D(p||q) = \sum_{x\in\Omega} p(x)\log_2\left(\frac{p(x)}{q(x)}\right) = E_p\log_2\left(\frac{p(x)}{q(x)}\right)$$

Tato funkce není symetrická a nesplňuje trojúhelníkovou nerovnost, ale je dobré jí uvažovat jako vzdálenost. Počet bitů pro zakódování dat z p, používáme-li q, lze vyjádřit jako $H(p) + D(p||q)$.

Vlastnosti relativní entropie

Informační nerovnost:

$$D(p||q) \geq 0$$

Důkaz plyne z Jensenovy nerovnosti – pro $p(x)$, náhodnou veličinu X on Ω a konvexní funkci f platí, že $f\left(\sum_{x\in\Omega} p(x)\cdot x\right) \leq \sum_{x\in\Omega} p(x)f(x)$. Jensenovu nerovnost lze ověřit indukcí nad $|\Omega|$ za použití definice konvexity funkce. Pak stačí vzít $0 = -\log 1 = -\log\sum_{x\in\Omega} q(x) = -\log\sum_{x\in\Omega} p(x)\frac{q(x)}{p(x)}$ a použít Jensenovu nerovnost.

Dále platí:

- Platí nerovnost sumy logaritmů pro $r_i, s_i \geq 0$: $\sum_{i=1}^n (r_i \log\left(\frac{r_i}{s_i}\right)) \leq (\sum_{i=1}^n r_i)\cdot\log\left(\frac{\sum_{i=1}^n r_i}{\sum_{i=1}^n s_i}\right)$. Z toho $D(p||q)$ je konvexní v p,q.
- Pro rozdělení p a rovnoměrné rozdělení u nad Ω platí: $H(p) \leq \log_2|\Omega|$, $H(X) = \log_2|\Omega| - D(p||u)$, $H(p)$ je konkávní.

Vzájmená informace

Vzájemná informace dvou náhodných veličin X, Y je:

$$I(X,Y) = D(p(x,y)||p(x)p(y))$$

Měří, kolik v průměru Y přispívá k zjednodušení predikce X (o kolik bitů se sníží entropie), nebo o kolik se $p(x,y)$ odchyluje od $p(x) \cdot p(y)$. Alternativní zápis je:

$$I(X,Y) = \sum_{x\in\Omega}\sum_{y\in\Psi} p(x,y)\log_2\left(\frac{p(x,y)}{p(x)\cdot p(y)}\right)$$

Platí:

$$I(X,Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$

Protože $p(x,y)/p(x)$ v předchozím vzorci se dá díky definici podmíněné pravděpodobnosti vyčlenit jako $-H(Y|X)$, zbývající $1/p(x)$ se dá přepsat na rozdíl dvou logartimů a $\sum_{y\in\Psi} p(x,y) = p(x)$ z něj dá $H(X)$.

Další vlastnosti:

- $I(X,Y) = H(X) + H(Y) - H(X,Y)$
- $I(X,X) = H(X)$ (protože $H(X|X) = 0$)
- $I(X,Y) = I(Y,X)$
- $I(X,Y) \geq 0$ (z Informační nerovnosti)

Křížová entropie

Odhadujeme skutečné rozdělení pravděpodobnosti r rozdělením p (na základě vzorku pozorování T) a potřebujeme znát přesnost aproximace. Nemáme skutečné p, takže ho simulujeme další množinou vzorků T' (s rozdělením p'). *Křížová entropie* se pak vypočítá jako:

$$H_{p'}(p) = -\sum_{x\in\Omega} p'(x)\log_2 p(x)$$

V praxi se více používá *podmíněná křížová entropie*. Testujeme rozdělení $p(y|x)$ proti $p'(x,y)$ (z nezávislých dat T'):

$$H_{p'}(p) = -\sum_{\substack{y\in\Psi\\x\in\Omega}} p'(x,y)\log_2 p(y|x) = -\frac{1}{|T'|}\sum_{i=1}^{|T'|}\log_2 p(y_i|x_i)$$

Křížová entropie $H_{p'}(p)$ může být nižší, vyšší než i rovna entropii odhadu rozdělení $H(p)$. Používá se k porovnávání odhadů – lepší odhad má nižší křížovou entropii na testovacích datech.

22.2 Bayesovské učení

Bayesovské učení je jiný přístup k odhadování pravděpodobnosti jevů než na základě frekvence výskytu: používáme navíc *apriorní předpoklad* (např. předpoklad známého rozdělení náhodné veličiny apod.), který na základě pozorovaných dat upravujeme. Hledaný jev (*hypotéza*) přitom může být např. klasifikační funkce strojového učení nebo nějaký neznámý parametr pravděpodobnostního modelu. Úplně přesně odpovídá *Zlatému pravidlu* NLP.

Definujeme:

- $p(h)$ – apriorní pravděpodobnost jevu h. Toto je náš apriorní předpoklad, nezávisí na datech.
- $p(D)$ – pravděpodobnost výskytu pozorovaných dat D bez znalosti pravděpodobnosti jevů (naštestí díky Zlatému pravidlu není moc potřeba ji znát).
- $p(D|h)$ – podmíněná pravděpodobnost výskytu dat D, nastal-li jev h.
- $p(h|D)$ – *aposteriorní pravděpodobnost*, pravděpodobnost jevu h za předpokladu, že máme pozorovaná data D.

Zajímá nás právě $p(h|D)$. Z Bayesovy věty plyne, že $p(h|D) = \frac{p(D|h)p(h)}{p(D)}$.

Hledáme-li jev s *maximální aposteriorní pravděpodobností (maximální aposteriorní hypotézu)*, pak můžeme podle Zlatého pravidla použít vzorec:

$$h_{MAP} = \arg\max_{h \in H} p(D|h)p(h)$$

Pokud jsou apriorní pravděpodobnosti všech jevů stejné (tj. apriorní předpoklad je rovnoměrné rozdělení), pak *maximálně vhodná hypotéza (maximum likelihood hypothesis)* je:

$$h_{ML} = \arg\max_{h \in H} p(D|h)$$

Bayesovské rozhodovací pravidlo (nebo *optimální bayesovské rozhodnutí*) velí vybrat právě h_{MAP} ze všech možných hypotéz.

Příklad

Mějme medicínský diagnostický test, který pro určité onemocnění vyjde pozitivně ($\bullet$) nebo negativně ($\circ$), a dva jevy (hypotézy) – pacient buď má (h_+) nebo nemá (h_-) danou nemoc. Apriorní předpoklad je, že 0.8% populace má tuto nemoc.

Testy mají omezenou spolehlivost:

	platí h_+	**platí** h_-
pozitivní test ($\circ$)	98%	3%
negativní test ($\bullet$)	2%	97%

Potom u pacienta s pozitivním výsledkem testu (tj. pozorovaných dat D) najdeme h_{MAP}:

1. pravděpodobnost, že tento pacient má danou nemoc je $p(\bullet|h_+)p(h_+) = 0.98 \cdot 0.008 = 0.0078$
2. pravděpodobnost, že tento pacient nemá tuto nemoc je $p(\bullet|h_-)p(h_-) = 0.03 \cdot 0.992 = 0.0298$

A tedy $h_{MAP} = h_-$. I když takto přesný test vyjde pozitivní, je u málo častého onemocnění velká šance, že jde o planý poplach. V praxi se proto testy opakují.

Vztah k učení založenému na konceptech

Mějme problém strojového učení, kdy hledané hypotézy $h \in H$ představují funkce klasifikující data D a my chceme z nich vybrat tu nejlepší. Použijme bayesovské učení tímto brutálně neefektivním způsobem:

1. Spočteme $p(h|D) = \frac{p(D|h)p(h)}{p(D)}$ pro každé $h \in H$.
2. Najdeme $h_{MAP} = \arg\max_{h \in H} p(h|D)$.

Předpokládejme, že žádná z hypotéz není pravděpodobnější než jiná (což typicky předpokládat musíme). Dále uvažujeme, že $p(D|h)$ bude 1, pokud klasifikace hypotézou h souhlasí s ohodnocením dat a 0 jinak.

Odhady velikostí pravděpodobností dojdeme k tomu, že jakákoliv hypotéza, která je konzistentní s trénovacími daty (tj. tato funkce by trénovací data klasifikovala správně), je h_{MAP}. Tedy mnoho algoritmů strojového učení, které takové hypotézy hledají, je efektivnější variantou bayesovského učení.

Souvislost s minimální délkou popisu

Bayesovské učení splňuje podmínku Occamovy břitvy, neboli *minimální délky popisu* (minimum descripton length), tedy nalezení co nejkratšího modelu h_{MDL} pro danou situaci. Platí totiž:

$$h_{MAP} = \text{argmax}_{h \in H} p(D|h)p(h) = \text{argmax}_{h \in H} \log_2 p(D|h) + \log_2 p(h) =$$

$$h_{MAP} = \text{argmin}_{h \in H} - \log_2 p(D|h) - \log_2 p(h)$$

A to se dá interpretovat jako preference nejkratší hypotézy s ohledem na specifický systém reprezentace hypotéz a dat. Pokud označíme $L_C(i)$ počet bitů potřebných k zakódování zprávy i v kódování C, pak se dá vzorec přepsat jako:

$$h_{MAP} = \text{argmin}_h L_{C_h}(h) + L_{C_{D|h}}(D|h)$$

Použijeme-li optimální kódování prostoru hypotéz i dat s ohledem na hypotézu h, pak $h_{MAP} = h_{MDL}$.

Optimální bayesovská klasifikace

Ve strojovém učení často není třeba vybrat nejlepší hypotézu h_{MAP}, ale postupně po jednom klasifikovat nové příklady z dat. Máme-li možné hodnoty $y_j \in Y$, pak je bayesovská pravděpodobnost těcthto hodnot dána vztahem:

$$p(y_j|D,x) = \sum_{h_i \in H} p(y_j|h_i)p(h_i|D,x)$$

Optimální bayesovská klasifikace (*optimální bayesovské učení*) je to y_j, pro kterou je $p(y_j|D,x)$ maximální, tedy argmax .

Tato metoda maximalizuje pravděpodobnost, že nová data budou klasifikována správně, jsou-li dána trénovací data, prostor hypotéz a apriorních pravděpodobnosti.

Bayesův optimální klasifikátor poskytuje sice nejlepší možné výsledky z trénovacích dat, ale je výpočetně náročný. To motivuje použití *Gibbsova algoritmu*:

1. Vybere se náhodně $h \in H$, ale vzhledem k distribuci aposteriorních pravděpodobností nad H.
2. h se použije k predikci klasifikace nové instance.

Dá se dokázat, že klasifikační chyba Gibbsova algoritmu je menší než dvojnásobek chyby optimálního bayesovského učení.

Naive Bayes Classifier

Naivní bayesovský klasifikátor je rychlá a relativně úspěšná metoda strojového učení. Mějme data, jejichž instance lze popsat n -ticí hodnot atributů $a_i, \dots a_n$ a jejich klasifikaci, jež nabývá hodnot $y_i \in Y$.

Naivní bayesovský klasifikátor předpokládá *nezávislost hodnot jednotlivých atributů*, tedy:

$$p(a_1, \dots, a_n|y) = \prod_{i=1}^{n} p(a_i|y)$$

Ve výsledku z běžného bayesovského přístupu dostaneme:

$$y_{NBC} = \text{argmax}_{y \in Y} p(y) \prod_{i=1}^{n} p(a_i|y) \text{ (a } p(y) \text{ odhadneme jako relativní frekvence na trénovacích datech)}$$

Je-li splněn předpoklad o podmíněné nezávislosti hodnot atributů, je $y_{NBC} = y_{MAP}$. To sice v praxi splněno nebývá, ale přesto lze dosáhnout dobrých výsledků.

22.3 Hidden Markov Models

Skrytý Markovův model je užitečný prostředek k predikci dat na základě omezené historie předchozích výstupů. Dá se aplikovat např. na jazykové modelování při statistickém překladu nebo na morfologické značkování.

Markovův proces

Mějme posloupnost náhodných veličin $\{X_i\}_{i=0}^{T}$ (obecně až do ∞) a stavový prostor $S = \{s_0, \dots, s_N\}$. Potom posloupnost $\{X_i\}$ nazveme *Markovův řetězec (proces)*, pokud splňuje *Markovovu vlastnost*:

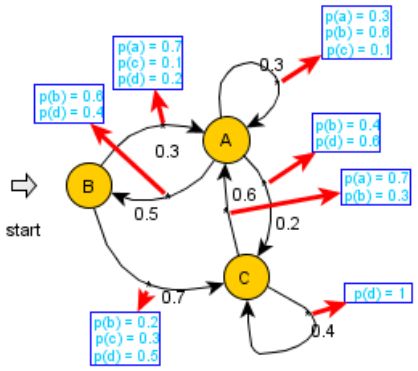
$$P(X_{i+1} = s_j | X_i = s_i, X_{i-1} = s_{i-1}, \dots X_0 = s_0) = P(X_{i+1} = s_j | X_i = s_i) \quad \forall i \in \{0, \dots T\}$$

Přitom musí pro každé $i \in \{0, \dots T\}$ platit $P(X_i = s_i, X_{i-1} = s_{i-1}, \dots, X_0 = i_0) > 0$. Mám tu tedy omezenou historii a rozdělení pravděpodobnosti přechodů z jednotlivých stavů se s časem nemění.

Formálně, abychom mohli mít závislost na více předchozích stavech, můžeme si nadefinovat nové stavy jako uspořádané n -tice stavů původních (a nastavit pravděpodobnosti u nenavazujících na 0). Takovému Markovovu procesu se říká *Markovův proces n -tého řádu*. Máme tu vlastně aproximaci řetězového pravidla:

$$P(s_1, \dots, s_T) = \prod_{i=1}^{T} p(s_i|s_1, \dots, s_{i-1}) \approx \prod_{i=1}^{T} p(s_i|s_{i-n+1}, \dots, s_{i-1})$$

Markovův proces lze zapsat buď přechodovou maticí, kde na pozici i, j se nachází pravděpodobnost přechodu ze stavu s_i do s_j , nebo stavovým diagramem – grafem. Odpovídá to vlastně nedeterministickému konečnému automatu.



Obrázek 22.1: Skrytý Markovův model – nejsložitější varianta

Skrytý Markovův Model

Nyní uvažujme složitější případ, kdy jednotlivé stavy Markovova procesu generují pozorovatelný výstup (znaky nějaké konečné abecedy), ale stavy samy a pravděpodobnosti přechodu nám jsou neznámé. Takové modely nazveme *skryté Markovovy modely* (HMM). Uvažuje se několik (postupně čím dál tím složitějších) variant:

1. Každý z výstupních symbolů je generován jen v jednom stavu
2. Různé stavy můžou generovat shodné symboly
3. Generování symbolů se provádí ne ve stavech, ale při přechodu (tj. závisí na dvojicích stavů)
4. Při každém přechodu mohou být s různou pravděpodobností vygenerovány různé symboly (tj. pro každou dvojici stavů mám rozdělení pravděpodobnosti výstupních symbolů)

Formálně se HMM definují jako pětice $\langle S, s_0, Y, P_S, P_Y \rangle$, kde:

- $S = \{s_0, \ldots, s_N\}$ je množina stavů, s_0 je počáteční stav
- $Y = \{y_0, \ldots, y_V\}$ je abeceda výstupních symbolů
- $P_S(s_j|s_i)$ je množina pravděpodobnostních rozdělení přechodů mezi stavy
- $P_Y(y_k|s_i, s_j)$ je množina rozdělení pravděpodobností výstupu jednotlivých symbolů abecedy pro každý přechod

HMM můžeme reprezentovat maticí přechodu a množinou $|Y|$ matic, které pro každý symbol udávají pravděpodobnost jeho vygenerování při všech stavových přechodech.

Hlavní použití HMM je:

- Spočtení pravděpodobnosti dané posloupnosti vygenerovaných symbolů
- Spočtení nejpravděpodobnější sekvence stavů, která vygenerovala danou posloupnost symbolů

Trellis a forward/backward algoritmus

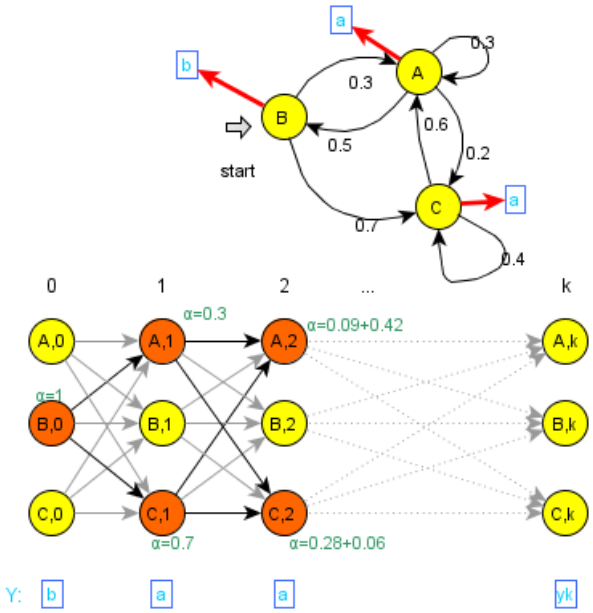
Uvažujme nyní HMM, kde se výstupní symboly generují ve stavech a každý stav generuje jeden symbol (ale více stavů může generovat shodné symboly) . Pro spočtení pravděpodobnosti posloupnosti výstupních znaků $Y, |Y| = k$ vytvoříme speciální graf – *trellis*, který modeluje chování původního HMM.

Stavy trellisu odpovídají stavům HMM v každém kroku generování, tj. místo stavů $s_0, s_1, \ldots, s_N$ máme stavy $s_{0,0}, s_{1,0}, \ldots, s_{N,0}, s_{0,1}, \ldots, s_{N,k-1}$. Pravděpodobnosti přechodů jsou vzaty z původního HMM. Uvažujeme ovšem jen stavy, které mohly vygenerovat sekvenci Y , ostatní nejsou užitečné.

Navíc si pro každý nový stav stanovíme hodnotu α , která udává pravděpodobnost, že se HMM v daný čas nacházel v daném stavu, je-li dána posloupnost Y . Ta se spočítá následovně:

$$\alpha(s_{\bullet,t+1}) = \sum_{s_i \Rightarrow y_i} P_S(s_\bullet|s_i)\alpha(s_{i,t}) \quad \text{(pro nějaký stav } s_\bullet \Rightarrow y_{i+1} \quad \text{a } t \in \{0, \ldots k-2\} \text{ , symbol "}\Rightarrow \text{ "zde značí "generuje")}$$

Potom pravděpodobnost posloupnosti Y je součet hodnot α ve stavech odpovídajících poslednímu kroku, které mohly vygenerovat poslední symbol. Při praktickém výpočtu pravděpodobnosti Y stačí, abychom drželi v paměti dva kroky výpočtu, tj. maximálně dvojnásobek stavů, než měl původní HMM. Máme tedy algoritmus s výpočetní složitostí $|S|^2|Y|$ a paměťovou složitostí $2|S|$, formálně nazývaný *forward algoritmus*.



Obrázek 22.2: HMM a jeho rozvití do trellisu. Stavy, které mohly vygenerovat výstupní sekvenci, jsou vyznačeny barevně a opatřeny hodnotou α.

Pro variantu HMM, kde se výstupní symboly emitují při přechodech, můžeme pracovat podobně, jen vzorec pro α bude následující:

$$\alpha(s_{\bullet,t+1}) = \sum_{(s_i,s_\bullet)\Rightarrow y_i} P_S(s_\bullet|s_i)P_Y(y_i|s_i,s_\bullet)\alpha(s_{i,t})$$

Je zřejmé, že takhle lze postupovat oběma směry (čímž dostáváme *backward algoritmus*). Při implementaci je třeba dát pozor na normalizaci, abychom nepočítali s příliš malými čísly a nedošlo k podtečení. Výhodné je např. počítat α v logaritmech.

Viterbi

Uvažujme teď další problém nad HMM. Pro danou posloupnost výstupních symbolů $Y, |Y| = k$ chceme najít nejpravděpodobnější sekvenci stavů S_{best}, která ji vygenerovala. Platí:

$$S_{\text{best}} = \arg\max_S P(S|Y) = \arg\max_S P(S,Y)$$

Protože $P(Y)$ je dané a tedy fixované. Potom z Markovovy vlastnosti plyne:

$$S_{\text{best}} = \arg\max_S P(S,Y) = \arg\max_S P(s_0,\ldots,s_k,y_0,\ldots,y_{k-1}) = \arg\max_S \prod_{i=0}^{k-1} P_S(s_{i+1}|s_i)P_Y(y_i|s_i,s_{i+1})$$

Uvažujme opět trellis, kde místo hodnot α budeme počítat hodnoty v, udávající pravděpodobnost nejpravděpodobnější sekvence stavů, která v zavedla HMM v daný čas do daného stavu, je-li dáno Y:

$$v(s_{\bullet,t+1}) = \max_{(s_i,s_\bullet)\Rightarrow y_i} P_S(s_\bullet|s_i)P_Y(y_i|s_i,s_\bullet)v(s_{i,t})$$

Viterbiho algoritmus na základě této rekurentní rovnice postupně počítá v (tedy jde o dynamické programování) a v každém stavu trellisu si pamatuje odkaz na nejpravděpodobnějšího předchůdce. Po projití celé posloupnosti výstupních symbolů pak může zpětně získat nejpravděpodobnější posloupnost skrytých stavů. V tomto případě je už nutné trellis uchovávat v paměti.

Existuje i varianta Viterbiho algoritmu, která hledá n nejpravděpodobnějších posloupností skrytých stavů. Pro tu je třeba uchovávat v každém stavu trellisu n nejpravděpodobnějších předchůdců. Při průchodu zpět stále uchováváme n nejlepších kandidátů.

Samozřejmě i tady je třeba pamatovat na normalizace a je možné použít prořezávání (s prahovou hodnotou buď pro v, nebo pro pravděpodobnost přechodu, nebo uchovávat jen daný počet stavů).

Baum-Welch

Dalším problémem na HMM je odhadnutí neznámých pravděpodobností P_S a P_Y na základě posloupnosti výstupních symbolů Y, tj. spočtení $\arg\max_{P_S,P_Y} P(Y|P_S,P_Y)$. Globální maximum efektivně nalézt neumíme, takže se musíme spokojit s lokálním.

K tomu se opět použije trellis a *Baum-Welchův algoritmus*, specifický typ EM-algoritmu (*expectation-maximization*). Ten postupuje následovně:

1. Inicializuje P_S a P_Y na nějaké počáteční hodnoty
2. *Expectation (estimation) step*: spočítá pravděpodobnost dat na základě původních odhadů P_S, P_Y
   (a) Spočítá forward pravděpodobnosti α (tj. aplikuje forward algoritmus, ale v paměti drží celý trellis)
   (b) Spočítá (obdobně) backward pravděpodobnosti β
3. *Maximization step*: spočítá nové odhady pravděpodobností P_S, P_Y :
   (a) Pro všechny dvojice stavů i generované symboly – $c(s_\bullet, s_\circ, y) = \sum_{i=0,(s_\bullet,s_\circ)\Rightarrow y}^{k-1} \alpha(s_{\bullet,i}) P_S(s_\circ|s_\bullet) P_Y(y|s_\bullet, s_\circ)\beta(s_{\circ,i+1})$
   (b) Pro všechny dvojice stavů – $c(s_\bullet, s_\circ) = \sum_{y\in Y} c(s_\bullet, s_\circ, y)$
   (c) Pro všechny stavy – $c(s) = \sum_{s'\in S} c(s, s')$
   (d) Nové pravděpodobnosti: $P'_S(s'|s) = \frac{c(s,s')}{c(s)}$, $P'_Y(y|s,s') = \frac{c(s,s',y)}{c(s,s')}$
4. Opakuje počítání odhadů, dokud nezačne konvergovat.

I tady je třeba normalizovat, navíc pokud máme nějaké aspoň hrubé odhady pravděpodobností (hlavně P_Y), výsledek bude mnohem lepší.

22.4 Algoritmy učení a zpracování, aplikace v lingvistice

Sem zřejmě patří něco z otázky o strojovém učení. Přidávám ještě dvě věci, které tam nejsou a v lingvistice se používají. Aplikace je možné vidět třeba v analýze jazyka.

EM-Algoritmus

EM (expectation-maximization) algoritmus je obecný postup iterativního odhadu neznámých parametrů nějakého modelu, který se používá např. v Baum-Welchově algoritmu z předchozí sekce.

Mějme pozorovaná data $X = \{x_1, \ldots, x_m\}$. Uvažujme ještě další, nepozorovaná data $Z = \{z_1, \ldots, z_m\}$ nad stejnými jevy, které popisuje X. Ta lze považovat za náhodnou veličinu, potom celková data $Y = X \cup Z$ jsou také náhodná veličina, jejíž rozdělení je určeno známými hodnotami X a rozdělením Z.

Cílem EM algoritmu je maximalizace pravděpodobnosti výskytu trénovacích dat za daných parametrů $E(\ln P(Y|\theta))$ – tím nalezneme parametry, které odpovídají pozorovaným datům. Používáme střední hodnotu, protože Y je náhodná veličina. Musíme tedy spočíst střední hodnotu přes její rozdělení, které je ale určeno parametry θ, jež neznáme. Proto definujeme rekurentní funkci, která počítá potřebnou střední hodnotu, má-li dány nějaké "předběžné" parametry θ:

$$Q(\theta'|\theta) = E(\ln P(Y|\theta')|\theta, X)$$

Celý algoritmus pak postupuje iterací následujících dvou kroků, za předpokladu, že nastavíme počáteční θ_0:

- *Expectation/Estimation* – spočítá se očekávaná hodnota $Q(\theta_{n+1}|\theta_n) = E(\ln P(Y|\theta_{n+1})|\theta_n, X)$
- *Maximization* – vybere se takové θ_{n+1}, které maximalizuje funkci $Q(\theta_{n+1}|\theta_n)$: $\theta_{n+1} = \arg\max_{\theta_{n+1}} Q(\theta_{n+1}|\theta_n)$

Je-li funkce Q spojitá, je zaručeno, že EM algoritmus v každém kroku zvýší její hodnotu a bude konvergovat ke stacionárnímu bodu věrohodnostní funkce $P(Y|\theta)$ (tj. k lokálnímu maximu).

Maximum Entropy

Modely *maximální entropie (maximum entropy, MaxEnt)* jsou pro zpracování jazyka jedny z nejpoužívanějších. Jejich základní ideou je najít podmíněné rozdělení pravděpodobnosti, které má, za daných podmínek, maximální entropii. To odpovídá principu Occamovy břitvy – najít co nejjednodušší popis na základě toho, co známe. Takový popis se co nejvíce blíží rovnoměrnému rozdělení a tedy má co nejvyšší entropii. Základní forma takového modelu, snažíme-li se předpovídat nějaký jev $y \in Y$ na základě kontextu $x \in X$, jako např. morfologický tag nějakého slova na základě vlastností okolního textu, vypadá následovně:

$$q_\theta(y|x) = \frac{\exp(\theta^T f(x,y))}{\sum_{y'\in Y} \exp(\theta^T f(x,y'))}$$

Funkce f představuje N-dimenzionální vektor rysů (jakýchkoli binárních klasifikací okolního textu) a θ odpovídající vektor vah. Jmenovatel tohoto zlomku, tj. suma přes všechna možná ohodnocení, má v praxi pouze normalizační funkci.

Váhy modelu se potom odhadují tak, aby splnily podmínku maximální entropie, což je ekvivalentní minimalizaci relativní entropie našeho modelu q_θ vzhledem k empirickému rozdělení pravděpodobnosti p pozorovanému na trénovacích datech (očekávaná frekvence výskytu jednotlivých vlastností), nebo maximalizaci jeho aposteriorní pravděpodobnosti (podmíněné pravděpodobnosti, jsou-li dána trénovací data), která se většinou popisuje pomocí logartimu věrohodnostní funkce:

$$\min_\theta D(p||q_\theta) = \min_\theta \sum_{y,x} p(y,x) \log \frac{p(y|x)}{q_\theta(y|x)} = \max_\theta L(\theta) = \max_\theta \sum_{y,x} p(y,x) \log q_\theta(y|x)$$

Hledání maxima logaritmické věrohodnostní funkce se většinou provádí gradientovou metodou.

Kapitola 23

Státnice I3: Návrh a vyhodnocování lingvistických experimentů

23.1 Úvod

- protože NLP používá ve velké míře stochastické metody, zaměříme se hlavně na experimenty testující účinnost těchto metod
- předtím, než je možné použít nějakou stochastickou metodu v praxi (a otesovat její účinnost), je nutné ji *natrénovat* na *trénovacích datech*
  - *trénování* závisí na dané metodě, ale většinou spočívá ve spočítání pravděpodobností použitých v metodě — často se odhadují pomocí relativních frekvencí získaných z trénovacích dat
- následně (pokud to daná metoda vyžaduje) je potřeba metodu přízpůsobit povaze dat (tzn. upravit její parametry, pokud existují), abychom maximalizovali její učinnost
  - parametry optimalizujeme na *development datech*
- následně je možné metodu otestovat pomocí vhodných metrik na *testovacích datech*

23.2 Příprava dat

- potřebujeme anotovaná data, u kterých ručně označíme správný výsledek experimentu — např. ručně přiřazené tagy slov pro tagging
- data je nutno rozdělit na 3 části
  - **training data** — největší, slouží k odhadnutí pravděpodobností; z velké části určují výsledek stochastické metody
  - **development data** — malá sada dat, která slouží k optimalizaci parametrů dané metody/modelu
  - **test data**
    - slouží pro ohodnocení kvality dané metody za použití vyhodnocovací metriky
    - nesmí být obsaženy v trénovacích a development datech, aby mohla být metoda objektivně ohodnocena
- pro nestochastické metody stačí pouze testovací data pro vyhodnocení

23.3 Standardní evaluační metriky

Evaluation

- test against evaluation test data – comparing the output of my parser to manually corrected data, done by someone else and in advance, independent of my algorithms
- rules:

1. should be automatic (if possible) – avoid subjective evaluation (but in e.g. SMT this is inevitable)
2. never tune the system using test data (use a small part of training data for this)
3. use standard metrics (if possible)

Hodnocení 1-1 metod

- pro každou vstupní jednotku vygeneruju jednu výstupní jednotku — např. tagging: každému slovu přiřadím tag
- **error rate**
- **accuracy**

Hodnocení 1-n metod

- délka vstupu a výstupu se může lišit — např. strojový překlad: výstupní věta může mít jinou délku než vstupní věta
- **precision**
- **recall**
- **f-measure**

Metriky strojového překladu

- BLEU
- NIST
- METEOR
  - upravená f-measure s důrazem na recall (precision:recall — 1:9)
  - párování slov na 3 úrovních: 1) slovní forma, 2) kořen slova, 3) WordNet synonymum
- PER (Position independent Error Rate), WER (Word Error Rate), TER (Translation Edit Rate), CDER

23.4 Typy evaluace podle úloh

Kapitola 24

Státnice I3: Automatická analýza jazyka

24.1 Morphology & Tagging

- Task, formally: $A^+ \rightarrow T$ (simplified), split morphology & tagging (disambiguation): $A^+ \rightarrow 2^{(L,C_1,C_2,...,C_n)} \rightarrow T$. Tagging must look at context.
- Tagset: influenced by linguistics as well as technical decisions.
  - Tag ~ n-tuple of grammar information, often thought of as a flat list.
  - Eng. tagsets ~ 45 (PTB), 87 (Brown), presentation: short names.
  - Other: bigger tagsets – only positional tags, size: up to 10k.
- Tagging inside morphology: first, find the right tag, then the morphological categories: $A^+ \rightarrow T \rightarrow (L, C_1, \ldots, C_n)$.
  - Doable for poor flection languages only.
  - Possibly only decrease the ambiguity for the purposes of tagging (i.e. morphology doesn't have to be so precise).
- Lemmatization: normally a part of morphology, sometimes (for searching) done separately.
  - *Stem* – simple code for Eng., no need of a dictionary, now out-dated.
- Possible methods for morphology:
  - Word lists: lists of possible tags for each word form in a language
    - Works well for Eng. (avg. ca. 2.5 tags/word), not so good for Cze. (avg. ca. 12-15 tags/word).
  - Direct coding: splitting into morphemes (problem: split and find possible combinations)
  - Finite State Machinery (FSM)
  - CFG, DATR, Unification: better for generation than analysis

Finite State Machinery

Finite-State-Automata

- Smarter word form lists: compression of a long word list into a compact automaton.
  - Trie + Grammar information, minimize the automaton (automaton reduction)
  - Need to minimize the automaton & not overgenerate

Two-Level-Morphology

- phonology + morphotactics, two-level rules, converted to FSA
- solves e.g. Eng. doubling (stop**p**ing), Ger. umlaut, Cze. "ský" –>pl. "čtí" etc.
- **Finite State Transducer**: an automaton, where the input symbols are tuples
  - run modes: *check* (for a sequence of pairs, gives out Y/N) + *analysis* (computes the "resolved" (upper) member of the pair for a sequence of "surface" symbols) + *synthesis* (the other way round)
  - used mostly for analysis

  - ususaly, one writes small independent rules (watch out for collisions), one FST for one rule – they're run in parallel and all must hold
  - zero-symbols, one side only, check for max. count for a language
  - we may eliminate zero-symbols using ordinary FSA on lexicon entries (upper layer alphabet; prefixes: according to them, the possible endings are treated specially)
- FSTs + FSAs may be combined, concatenated; the result is always an FST

Two-level morphology: analysis

1. initialize paths $P := \{\}$
2. read input surface symbols, one at a time, repeat *(this loop consumes one char of the input)*:
   (a) at each symbol, until max. consecutive zeroes for given language reached, repeat *(this loop just adds one possible zero)*:
      i. generate all lexical (upper-level) symbols possibly corresponding to zero (+all possible zeroes on surface)
      ii. prolong all paths in P by all such possible $(x : 0)$ pairs (big expansion)
      iii. check each new path extension against the phonological FST and lexical FSA (lexical symbols only), delete impossible path prefixes.
   (b) generate all possible lexical symbols (get from all FSTs) for the current input symbol, create pairs
   (c) extend all paths with the pairs
   (d) check all paths (next step in FST/FSA) and delete the impossible ones
3. collect lexical "glosses" from all surviving paths

Rule-Based Tagging

- Rules using: word forms/tags/tag sets for context and current position, combinations
  - If-then / regular expression style (blocking negative)
  - Implementation: FSA, for all rules – intersection
  - algorithm ~ similar to Viterbi (dynamic programming: if the FSA rejects a path, throw it away)
  - May even work, sometimes does not disambiguate enough
- Tagging by parsing: write rules for syntactic analysis and get morphological analysis as a by-product
  - in a way, this is the linguistically correct approach
  - better, cleaner rules
  - more difficult than tagging itself, nobody has ever done it right

HMM Tagging

- probabilistic methods, also applies to feature-based
- noisy channel: input (tags) -> output (words), goal: discover channel input given the output.
  - $p(T|W) = p(W|T) \cdot \frac{p(T)}{p(W)}$, $\arg\max_T p(T|W) = \arg\max_T p(W|T)p(T)$.
- two models – simplification:
  - tags depend on limited history (4-5 grams)
  - word depends on tag only (1 gram!)
- almost the general HMM
  - output words emitted by states (not arcs), states are $(n-1)$-tuples of tags for an n-gram tag model
  - (S, s_0, Y, P_S, P_Y) – set of states, initial state, output alphabet (words), set of prob. distributions of transitions, set of prob. distributions for emissions
- **supervised learning**: use MLE, smoothing:
  - p(w|t) –"add 1" for all possible tag+word pairs using a predefined dictionary (i.e. some 0's kept: p(word|impossible-tag) = 0)
  - p(t|context) – linear interpolation, up to uniform (as for language model)

- old and simple, but still accurate enough (only slower than e.g. neuron networks)
- may be trained even with **unsupervised** data: unambiguous words help get the disambiguation for the others (improvement depends on language and tagset)
  - Baum-Welch algorithm, minimizing the entropy; use heldout data
  - training always decreases the entropy, smoothing increases it again (in case of no bigger tagged corpus available, it's a good step to try; supervised is always better)
- **Out-of-Vocabulary**
  - no lists of possible tags
  - try all / open class tags (good for non-flective languages), or:
  - try to guess possible tags based on suffix/ending or both ends of the word (e.g. for Cze. – first 3 and last 8 letters) – train the classifier using *rare* words from the training data only!
- **Running the tagger**
  - Viterbi, remember to handle unknown words, or:
  - assign always the best tag at each word, but consider all possibilities for previous tags; introduces some errors, but might get better accuracy

Transformation-Based Tagging

- Not noisy channel, not probabilistic, but statistical – uses training data (combination of supervised/unsupervised), learning rules of type context + $\text{tag}_1 : \text{tag}_1 \rightarrow \text{tag}_2$
- criterion: accuracy – "objective function"
- **training**: stepwise greedy-select
  - iterate: pre-anotate using current rules (intermediate data), select the rule from the pool of possible ones (from templates) that contributes best to the improvement of the error rate
  - stopping criterion: no or too small improvement possible; prone to overtraining!
  - heldout possible (afterwards, remove rules that degrade performance on heldout data)
- rule types: context, lexical (looks at parts of the word)
  - application of the rules – left-to-right (a rule may be applied on part of its output) / delayed
- improved version: Fast-TBL(Transformation-Based Learning), there's no parallelized version
- old method (90's – was the best one), faster than HMM
  - tested for Cze. in the late 90's, not very good results, too many rules – uncomputable (no way to parallelize it, the rules are weird in the beggining)
  - may be used e.g. for named entity recognition (less rules, more effective)
- **tagger**: input = untagged data + rules from the learner
  - applies the rules one-by-one to all the data –> creates n iterations of intermediate data, the last one of which is the output
- n-best modification (criterion: accuracy + number of tags per word), unsupervised modification (use only unambiguous words for evaluation)

Maximum-Entropy Tagging

- using Max. Entropy model
  - **feature functions** – take the data and say Y/N (or give out a real number), weighted ($\sum \lambda = 1$)
  - maximize entropy ~ train the weights
  - respect the proportions of counts of features in the data (1's)
  - objective function optimization under certain constraints: Lagrange multiplicators
- the only freely adjustable part is the set of features (which is much more than what we may tune in an HMM – there's only n for n-grams)

  - features may use tags only at positions to the left (and dynamic programming ˜ Viterbi) + words on all positions, but don't have to do it at all (it's only very useful to do so)

- the model in general: $p(y|x) = \frac{1}{Z} e^{\sum_{i=1}^{N} \lambda_i f_i(y,x)}$, find λ_i satisfying the model and constraints ($E_p(f_i(y,x)) = d_i$ where $d_i = E'(f_i(y,x))$ – empirical expectation of the feature frequency)
  - for tagging: $p(t|x) = \frac{1}{Z} e^{\sum_{i=1}^{N} \lambda_i f_i(t,x)}$, where $t \in$tagset, x context (words and tags, e.g. up to 3 positions L-R (tags left only))
  - features may ask any information from this window
  - careful for too many details – it slows things down and there's an imminent danger of overtraining
  - best way: select features automatically – we then have a combination of HMM & Transformation-Based Learning
    * rules involve tags at previous places and are selected by machine
    * we still have to define the templates for the features
    * features are independent –> parallelizable; may be even interesting linguistic phenomena ˜ final punctuation -> mode of the verb etc.
- **feature selection**
  - manually: impossible –> greedy selection: iterative scaling algorithm ($O(n^2)$ – still too much)
  - limiting the number of features: be reasonable in creating the templates
    * use contexts which appear *in the training data*
    * use only rarer features (max. ca. 10 times in the training data) – too frequent features are probably too unspecific – get rid of them, e.g. if they have a distribution close to uniform (use relative entropy)
    * do not use all combinations of context
- feature types
  - any, even lexical features – words with most errors etc.
- the best way of tagging, the only problem is that we don't know what the optimal features are (the neural networks/perceptron are the best for Cze.)

Feature-Based Tagging

- idea: save on computing feature weights, select a few but good features
- training criterion: error rate
- model form: same as for maximum entropy $p(y|x) = \frac{1}{Z(x)} e^{\sum_{i=1}^{N} \lambda_i f_i(y,x)}$ – *exponential* or *loglinear* model

Tagger Evaluation

- A must: **Test data (S)**, previously unseen, change frequently if possible
- Formally: Out(w) / True(w) – for a given word
  - Errors(S) = $\sum_{i=1}^{|S|} \delta(\text{Out}(w_i) \neq \text{True}(w_i))$
  - Correct(S) = $\sum_{i=1}^{|S|} \delta(\text{True}(w_i) \in \text{Out}(w_i))$
  - Generated(S) = $\sum_{i=1}^{|S|} |\text{Out}(w_i)|$ – how many outputs the tagger produced (sum over all data)

Metrics

- *Error rate*: Err(S) = Errors(S) / |S|
- *Accuracy*: Acc(S) = 1 — Err(S)
- Multiple / no output:
  - *Recall*: R(S) = Correct(S) / |S| – must select the right one (possibly among others)
  - *Precision*: P(S) = Correct(S) / Generated(S) – against too much noise
  - no way to improve P + R at the same time, but also no way to tell what's better (depends on the application: Google – P, Medical test – R)
    * systems with a big difference in P/R are (empirically) worse
  - *F-Measure*: $F = \frac{1}{\frac{\alpha}{P} + \frac{1-\alpha}{R}}$, usually $F = \frac{2PR}{R+P}$ (for $\alpha = 0.5$)

24.2 Parsing

Chomsky Hierarchy

1. Type 0 Grammars/Languages – $\alpha \to \beta$, where α, β are any strings of nonterminals
2. Context-Sensitive Grammars/Languages – $\alpha X \beta \to \alpha \gamma \beta$, where X is a nonterminal, α, β, γ any string of terminals and nonterminals (and γ must not be empty)
3. *Context-Free Grammars/Languages* – $X \to \gamma$, where X is a nonterminal and γ is any string of terminals and nonterminals
4. Regular Grammmars/Languages – $X \to \alpha Y$, where X, Y are nonterminals and α a string of terminals, Y might be missing

Chomsky's Normal Form

- for CFG's – each CFG convertible to normal form
- rules only $A \to BC$ (two nonterminals), $A \to \gamma$ (one terminal), $S \to \varepsilon$ (empty string)

Parsing grammars

- Regular Grammars – FSA, constant space, linear time
- CFG – widely used for surface syntax description of natural languages, needed: stack space, $O(n^3)$ time – CKY/CYK algorithm

Shift-Reduce CFG Parsing

- CFG with no empty rules ($N \to \varepsilon$) – any CFG may be converted; recursion is OK
- Bottom-up, construction of a push-down automaton (non-deterministic); delay rule acceptance until all of it is parsed
- Asymptotically slower than CKY, but fast for usual grammars
- Builds upon a **state parsing table** ˜ graph, edges = transitions (defined by one terminal or nonterminal symbol)
  - each state: a special function telling if we output the rule number (even more rules – ambiguity) – this is what separates a shift-reduce parser from an FSA
- *lex/yacc* / *flex/bison* – shift-reduce parser generators

Table construction – dot mechanism

- dots = remember where we are in all the rules which possibly could go through this set, used only for table construction

1. take the *starting rule* and add it to the *1st state* (put all rules with the starting symbol on left-hand side into the 1st state, mark the dot at the beginning of the right-hand side of all of them)
2. *state expansion*: for all nonterminals right after the dot in any rule in this state, add the rewriting rules (in which the given nonterminal is on the left-hand side) into this state (and do this recursively until there are no more nonterminals that have not been expanded)
3. *construction of following states*: for each terminal / nonterminal after the dot, create a new state (if there are several rules for the same symbol, create only one state over the transition for this symbol)
4. *into the new state*: add all the rules with the transition symbol just after the dot and move the dot after it + perform state expansion
5. *reduction states*: if there is a rule with the dot at the end in the state, this state is a so-called *reduction-state* – in this state, the rule that caused the possibility of moving forward shall be printed (such rule has no expansions –> this leads to finish)
6. *merge identical states*: if the created state has the same rules (with dots at the same positions) as another state, merge the two (otherwise this would never finish for a recursive grammar; merge only after the whole state has been created!)

- problems:

  - *shift-reduce conflict* – a state is ambiguous (there is a rule with the dot at the end + some rules with dots in the middle ˜ state may be reductional, but doesn't have to) – this leads to backtracking, ambiguous parses
  - *reduce-reduce conflict* – another kind of ambiguity (more different rules with dot at the end)
  - the ambiguity does not occur for special kind of grammars – $LR(n)$: for bottom-up parsing, we only need to look n symbols ahead to prevent backtracking, $LR(0)$ never get a conflict in a table
    * there's no simple algorithm for obtaining the n for which a given grammar is $LR(n)$, but we may try for all n's
  - the algorithm complexity copies the complexity of the grammar – it's only expensive at the points of ambiguity

Parsing

- using *parsing stack* for states and *backtrack stack* for whole parser configurations at the points of ambiguity
  - backtracking may be implemented in such a way that only the position in the parsing stack and the input need to be stored on the backtrack stack

1. we have an empty backtrack stack, the 1st state on the parsing stack and the original string at the input
2. from a shift state, follow the transition using the input symbol:
   (a) if there is no such transition and there's nothing on the backtrack stack, FAIL; otherwise take something out of the backtrack stack – keep the stack saved if there still are more possibilities to follow!)
   (b) if we find the transition, eat one symbol from the input, follow it to the new state and put the state on the parsing stack
3. if we are in a reduce state:
   (a) remove as many elements from the *parsing stack* as there are on the right-hand side of the rule we're reducing over
   (b) put the nonterminal from the left-hand side of the rule on *the input*
4. for conflicts: follow the first path + save the current configuration to the backtrack stack
5. PASS condition: empty parsing stack and end of input (possibly continue looking for some other parses if there's something on the backtrack stack)

Probabilistic CFG

- relations among mother & daughter nodes in terms of derivation probability
- probability of a parse tree: $p(T) = \prod_{i=1}^{n} p(r(i))$, where $p(r(i))$ is a probability of a rule used to generate the sentence of which the tree T is a parse
  - probability of a string is not as trivial, as there may be many trees resulting from parsing the string: $p(W) = \sum_{j=1}^{n} p(T_j)$, where T_j's are all possible parses of the string W.
- assumptions (very strong!):
  - independence of context (neighboring subtrees)
  - independence of ancestors (upper levels)
  - place independence (regardless where the rule appears in the tree) ˜ similar to time invariance in HMM
- probability of a rule – distribution $r(i) : A \to \alpha$ ˜ $0 \leq p(r) \leq 1$; $\sum_{r \in \{A \to \dots\}} p(r) = 1$
  - may be estimated by MLE from a treebank following a CFG grammar: counting rules & counting non-terminals
- inside probability: $\beta_N(p,q) = p(N \to^{\star} w_{pq})$ (probability that the nonterminal N generates the part of the sentence $w_p \dots w_q$)
  - for CFG in Normal Form may be computed recursively: $\beta_N(p,q) = \sum_{A,B} \sum_{d=p}^{q-1} p(N \to A\ B) \beta_A(p,d) \beta_B(d+1,q)$

Computing string probability – Chart Parsing (CYK algorithm)

- create a table $n \times n$, where n is the length of the string

- initialize on the diagonal, using $N \to \alpha$ rules (tuples: nonterminal + probability), compute along the diagonal towards the upper right corner
  - fill the cell with a nonterminal + probability that the given part of the string, i.e. row = from, col = to, is generated by the particular rule
  - consider more probabilities that lead to the same results & sum them (here: for obtaining the probability of a string, not parsing)
- for parsing: need to store what was the best (most probable) tree – everything is computable from the table, but it's slow: usually, you want a list of cells / rules that produced this one
  - if the CFG is in Chomsky Normal Form, the reverse computation is much more effective

External links

- http://ufal.mff.cuni.cz/~novak/presentPraha/ — slides in Czech
- http://nlp.stanford.edu/fsnlp/pcfg/fsnlp-pcfg-slides.pdf — slides in English

Statistical Parsing

- parsing model: $p(s|W) = \frac{p(W,s)}{p(W)} = \frac{p(s)}{p(W)}$ since $p(W,s) = p(s)$ (where s is a parse and W the corresponding string – a parse defines a sentence!)
  - therefore: $\text{argmax}_s p(s|W) = \text{argmax}_s p(s)$ – we just select the most probable parse
  - similar to language model; we don't consider trees, but all the possible parses

Parser Creation

1. extract (all used) rules from a treebank
2. convert the grammar to the normal form
3. apply this back to the treebank (keep track of which rules were affected by the conversion)
4. get the counts of the rules –> probability
5. smoothen

Smoothing

- the extracted rules cover an infinite number of sentences, but certainly not the whole language
- add *poor* (missing) rules – get all possible combinations on the right side
  - but ensure their probability is really very low!
- there are many ways of smoothing
  - e.g. tie several probabilities to one: $B \to N\ V$ –> split to first and second nonterminal: $p(B \to N\ \star)$, then: $p(B \to N\ V) = p(B \to N\ \star) \cdot p(V|N)$ (only V following N on the right-hand side of any rule, regardless of the left-hand side)
  - or, use some linear combination of similar tied probabilities
  - may be combined with the original ones before smoothing
- the smoothing is often tuned on data, the best way is selected according to the performance
- if we don't use Chomsky's Normal Form, we may have much more sofisticated ways of smoothing, perhaps even reflecting the linguistic properties of the language, but it'll slow down the process

Lexicalization

- obtain more distinct nonterminals: use lexicalized parse tree (~ dependency tree + phrase labels; no lexicalization needed for dep. trees)
  - process of filling in words that were originally only on the terminal nodes – so that the word is taken from the head of the phrase

1. pre-terminals (right above the leaves): assign the word below

2. recursive step (up one level – bottom-up): select one node and copy it up (the "more important one", eg. the preposition for PP, the noun for NP; there are no clean rules, it's a linguistic problem)

- increases the number of rules (up to 100k), but helps – the smoothing must be very precise (e.g. using the non-lexicalized distribution)
  - particular words are important for the parsing of the sentence –> CFG development paradigm
- it's possible to use POS-tags with the words, or POS-tags only
- conditional probabilities: there are too many rules, the data are sparse –> we need to simplify – *assumptions*:
  - total independence ($p(\alpha B(head_B)\gamma \ldots | A(head_A)) = p(\alpha | A(head_A)) \cdot p(B(head_B)|A(head_A)) \ldots$) is too strong, too inaccurate
  - best known heuristics – *decomposition*: split the right side of the rule into head + left-of-head + right-of-head
    - technical terminal STOP at both sides of the head **(?)**
    - $p_H(H(head_A)|A(head_A))$, $p_L(L_i(l_i)|A(head_A), H)$, $p_R(R_i(r_i)|A(head_A), H)$
  - more conditioning – distance: absolute is non-zero ? path goes over verb ? over commas ?
  - other: complement/adjunct, subcategorization **(?)**

Remarks

- parsing is still not solved properly, the results are not sufficient

Dependency parsing

- until 2005, done via phrase-tree parsing, the trees were then converted
- McDonald's Parser
- result: a tree – each word has its parent (or is root)
- initialize: make a total graph, where each edge is rated with a probability (using a perceptron) + find the maximum spanning tree

Parsing Evaluation

Dependency parser metrics:

- dependency recall: $R_D = \text{Correct}(D)/|S|$, where Correct(D) is the number of correct dependencies (correct head / marked root), |S| is the size of the test data in words
- dependency precision: if output is not a tree – $P_D = \text{Correct}(D)/\text{Generated}(D)$, where Generated(D) is the number of output dependencies

Parse tree metrics:

- number of nonterminals may not be the same as in the "truth" –> more complicated
- crossing brackets measure: number of crossing brackets between the truth and the result
- labeled precision/recall – usual computation using bracket labels (phrase markers)
  - the bigger label coverage, the better – recall
  - the less brackets, the better – precision

Kapitola 25

Státnice I3: Generování přirozeného jazyka

25.1 Úvod

Problémy NLG (Natural Language Generation): Jak by měly počítače komunikovat s člověkem? Jaké chování od nich lidi očekávají? (Někdy je lepší vyplivnout graf nebo tabulku.) Co je "srozumitelný" jazyk v dané situaci? Jak převést reprezentaci znalostí (často hromada numerických dat) do "lidské podoby" (typicky malý počet abstraktních pojmů)?

Zahrnuje AI, lingvistické formální modely. Vlastně inverzní k analýze, k porozumění – nejde tu ale o zabývání se hypotézami, ale výběr vhodné strategie sdělení. "Dvojsměrný" systém se staví dost těžko – analýza musí počítat s nekorektním vstupem, ale neřeší srozumitelnost svého výstupu. Reprezentace znalostí v obou typech systémů je většinou odlišná.

Aplikace: většinou prezentace informací, které vznikají automaticky, případně (částečná) automatizace rutinní dokumentace (lékařské, programátorské atd.). Důležité, protože interní reprezentace databází nejsou člověku srozumitelné.

Důvody použití:

- Konzistence textů a dat
- Splnění standardů pro formát výstupu
- Rychlost produkce dokumentů
- Mnohojazyčnost
- Lidi to prostě nudí, je-li to monotónní úkon.

Historie

Od 50-60. let, v rámci MT systémů, první formální gramatiky pro náhodné generování korektních vět. V 70. letech první pokusy o NLG pro interpretaci dat. Skutečné nasazení systémů v 90. letech.

25.2 Struktura NLG systému

Není úplně ustálená, je mnoho možností.

- Vstup: zdroj znalostí, komunikativní cíl (konkrétního použití – např. "shrnout data o počasí za poslední měsíc"), uživatelský model ("charakterizace cílového publika"), historie diskurzu ("co už bylo řečeno")
- Výstup: text (formátování záleží na aplikaci — často např. HTML)

Typická základní architektura – pipeline:

- **Plánovač dokumentu (plánovač textu)** (určí obsah a strukturu výstupu)
  - Vytváří obsah ("zprávy"), určuje, které z nich je třeba vypsat pro splnění komunikativního cíle (*content determination* – výběr relevantních informací)
  - Strukturuje výstupní dokument, aby bylo možné vygenerovat srozumitelný a souvislý text (*document structuring*) – podle očekávání čtenáře (žánr), seskupování související informace (např. "vše o teplotě napsat za sebou")
  - Rozdělení do vět a jazykové otázky se tady zatím neřeší
- **Mikroplánování (plánovač vět)** (jaká slova, syntaktické struktury atp. použít)

  - Někdy je výstupem už text, někdy mezireprezentace (např. specifikace času věty apod.)
  - Lexikalizace (jaká slova a konstrukce použít – "pršelo od 11. do 14. / pršelo 11.,12.,13.,14."?), generování označení (*refering expression generation* – jak označovat entity? – první / následné zmínky)
  - Agregace (mapování struktury dokumentu na jazykovou strukturu – co vecpat do které věty/odstavce?) – např. "Minulý měsíc byl chladný" + "Minulý měsíc byl suchý" = "Minulý měsíc byl chladný a suchý"
- **Povrchová (lingvistická) realizace** (převod abstraktní reprezentace použité mikroplánovačem do skutečného textu)
  - Realizace lingvistická i strukturální (formátování textu)
  - Některé systémy (PEBA) mají dost jednoduchou jazykovou realizaci – jde vlastně o šablony doplňované údaji
  - Jiné (ModelExplainer) používají abstraktní synt. struktury, hodí se i pro mnohojazyčný výstup
  - Systemic Grammar, Functional Unification Grammar

Datové mezistupně:

- Plán dokumentu – typicky stromová struktura, vnitřní uzly – sturkturní informace, listy – obsah ("zprávy")
- Specifikace textu – opět stromy, vnitřní uzly – sturktura textu, listy – věty ("specifikace frází")
  - Specifikace fráze: ortografický řetězec (vše vyřešené) / *canned text* (nutné řešit velká písmena, interpunkci apod.), *abstraktní syntaktická sturktura* (lexémy a jejich rysy, závislostně uspořádané), *lexicalized case frame* (spíše lexikalizovaný sémantický strom).

25.3 Příklady NLG systémů

- KPML – obecný NLG systém, Systemic Functional Grammar (pro několik jazyků, vč. EN, CZ)
  - SFG – výběr z alternativ: povrchová realizace je důsledkem výběru funkčních rysů (z popisu/taxonomie celého jazyka – *systemic network*), znamená projití sítí (krok = výběr rysu) bez backtrackingu (každý přechod má určité podmínky, napojení = závislost jazyk. elementů)

Počítač jako pomůcka

- FoG – generování předpovědi počasí (v Kanadě – EN/FR)
- PlanDoc – dokumentace navrhovaných změn v telefonních sítích
- AlethGen – pomůcka při psaní odpovědí zákazníkům :-)
- Drafter – pomůcka pro psaní manuálů k softwaru

Počítač jako samostatný autor

- IDAS – poskytuje informace o používání přístroje na základě reprezentace znalostí
- ModelExplainer – vysvětluje strukturu objektově-orientovaných programů
- PEBA – popisy taxonomické báze znalostí
- Piglet – vysvětluje pacientům v nemocnici jejich lékařské zprávy
- STOP – generuje personalizovanou informaci o škodlivosti kouření :-)

25.4 Evaluace

- Založená na úkolech (jak systém pomáhá člověku zvládnout daný úkol)
- Lidská (lidi posuzují srozumitelnost)
- BLEU nebo něco podobného

Kapitola 26

Státnice I3: Analýza a syntéza mluvené řeči

26.1 Speech Recognition

- The task of transforming of acoustic signal into text.
- Based on statistical methods
- Constant increase of performance over last decade
- Parameters influencing the recognizers performance:
  - size of the vocabulary
    - yes/No recognition
    - digit recognition — 10 words
    - large vocabulary — 20,000 to 60,000 words
  - fluency of speech
    - isolated words recognition
    - continuous speech recognition
  - noise to signal ratio
- The following text will focus on **Large-Vocabulary Continuous Speech Recognition** (although the methods are applicable universally), the methods shown are **speaker independent** (The system was not trained on the person to be recognized)

Speech Recognition Architecture

- **Noisy-channel paradigm**
- Acoustic input $O = o_1, o_2, o_3, ..., o_t$
  - consist of individual "acoustic observations" o_i
    - o_i is represented as a feature vector
    - usually one acoustic observation for each 10ms
- Sentence treated as a string of words $W = w_1, w_2, w_3, ..., w_n$
- Probability model:

$$W^* = \arg\max_{W \in L} P(W|O) = \arg\max_{W \in L} \frac{P(O|W)P(W)}{P(O)} = \arg\max_{W \in L} P(O|W)P(W)$$

- $P(W)$ — the prior probability — computed by **language model**
- $P(O|W)$ — the observation likelihood — computed by **acoustic model**

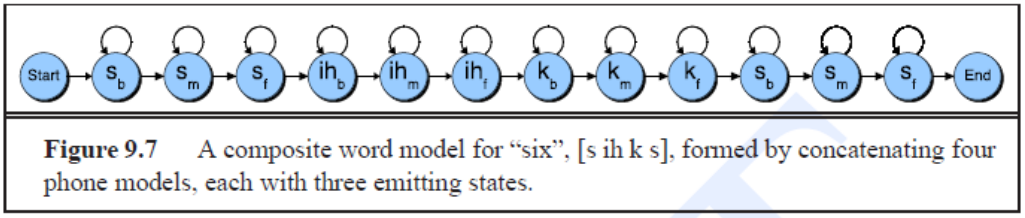


Figure 9.7 A composite word model for "six", [s ih k s], formed by concatenating four phone models, each with three emitting states.

Obrázek 26.1: (Speech and Language Processing(draft) — Jurafsky, Martin)

The Hidden Markov Model Applied To Speech

Vezmeme Markovův model, kde skryté stavy odpovídají hláskám (nebo jejich částem) a emitované znaky odpovídají akustickému pozorování podle akustického modelu. HMM pro analýzu řeči má ale speciální strukturu:

- Typical feature of ASR HMMs: **left-to-right** HMM structure
  - HMM don't allow transition from states to go to earlier states in the word
  - states can transition to themselves or to successive states
    - transitions from states to themselves are extremely important — durations of phones can vary significantly (for example: duration of [aa] phone varies from 7 to 387 miliseconds — 1 to 40 frames)

HMM representation of phone

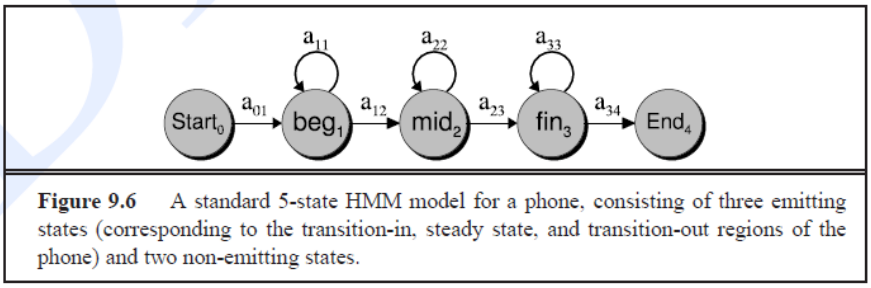


Figure 9.6 A standard 5-state HMM model for a phone, consisting of three emitting states (corresponding to the transition-in, steady state, and transition-out regions of the phone) and two non-emitting states.

Obrázek 26.2: (Speech and Language Processing(draft) — Jurafsky, Martin)

- *Figure 9.6*
  - Phones are non-homogeneous over time
    - Thus there are separate states modelling a beginning, middle, and end of each phone.

HMM representation of word

- pronounciation lexicon needed — tell us for each word what phones it consists of (sometimes multiple variants of pronounciation)
  - pronounciation lexicon of English: CMU dictionary (publicly available)
- Concatenation of HMM representations of phones

Speech recognition HMM

- *Figure 9.22*:
  - Combination of word HMMs
  - adde special state modelling silence
  - transition from *end* state to *start* state — sentence can be constructed out of arbitrary number of words
  - transitions from the *start* state are assigned unigram LM probabilities, higher n-grams also possible
- *Figure 9.29*:
  - *start* state, *end* state and silence states omitted here for a convinience reason
  - Bigram language model used here — probabilities of transitions from the ends to the beginnings of the words

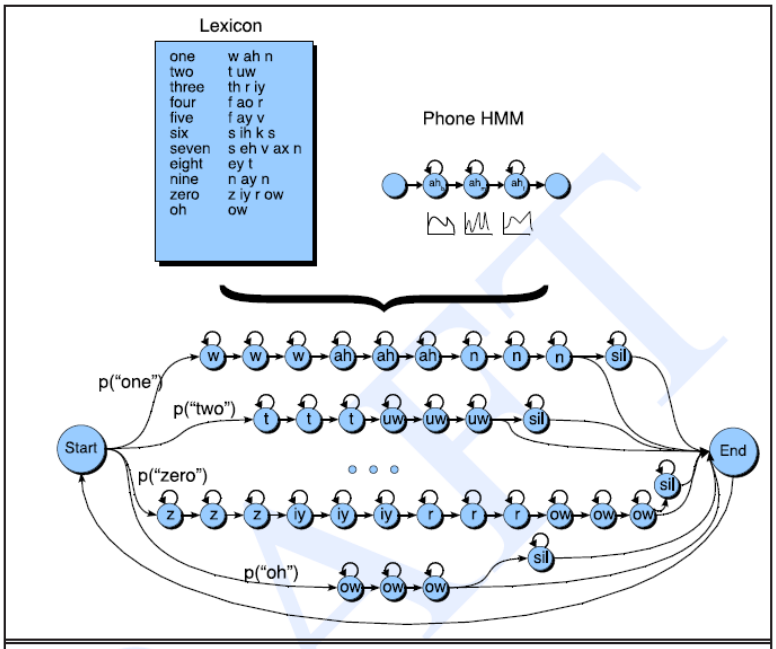


Figure 9.22 An HMM for the digit recognition task. A lexicon specifies the phone sequence, and each phone HMM is composed of three subphones each with a Gaussian emission likelihood model. Combining these and adding an optional silence at the end of each word, results in a single HMM for the whole task. Note the transition from the End state to the Start state to allow digit sequences of arbitrary length.

Obrázek 26.3: (Speech and Language Processing(draft) — Jurafsky, Martin)

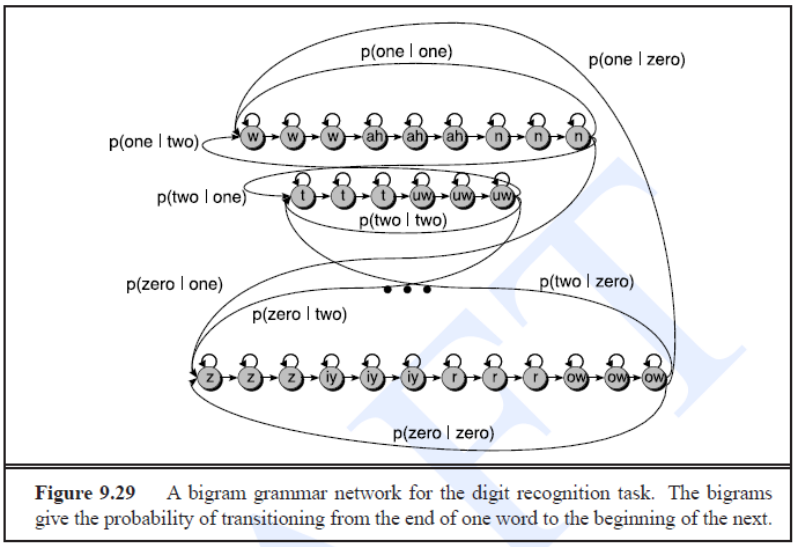


Figure 9.29 A bigram grammar network for the digit recognition task. The bigrams give the probability of transitioning from the end of one word to the beginning of the next.

Obrázek 26.4: (Speech and Language Processing(draft) — Jurafsky, Martin)

Feature Extraction: MFCC Vectors

- *MFCC — mel frequency cepstral coefficients* — most common feature representation (– spectral features)

Analog-to-Digital Conversion

- **sampling**
  - measuring of the the amplitude of the signal at a particular time
  - *sampling rate* — number of samples per second
  - maximum frequency that can be measured is half of the *sampling rate*
- **quantization**
  - Represenation of real-valued numbers (amplitude measurements) as integer
    - 8 bits => values from -128 to 127
    - 16 bits => values from -32768 to 32767

Preemphasis

- boosting of amount of energy in the high frequencies
- improves phone detection accuracy – vyšší harmonické frekvence jsou pak znatelnější

Windowing

- spectral features are to be extracted from a small segments of speech
  - asumption that the signal is stationary on small segment
- (rougly) stationary segments of speech extracted by using a **windowing** technique
- windowing characterized by
  - window's **width**
  - **offset** between succesiev windows
  - **shape** of the window
- segments of speech extracted by windowing are called **frames**
  - **frame size** — number of miliseconds in each frame
  - **frame shift** — miliseconds between the left edges of successive windows

Discrete Fourier Transform (DFT)

- extracts spectral information for the windowed signal
- how much energy each frame contains at different frequency bands

Mel Filter Bank and Log

- human ears less sensitive to higher frequencies (above 1000Hz)
- human hearing is "logaritmic"
  - (i.e. for human, distance between 440Hz and 880Hz equals the distance between 880Hz and 1760Hz — in both examples, the distance is one musical octave)
- these feature of hearing are utilized in speech recognition
- DFT outputs are warped onto the **mel scale** $\text{mel}(freq) = 1127 \ln(1 + \frac{freq}{700})$
  - Frequency scale divided into bands (frekvenční pásma)
    - 10 bands linearly spaced below 1000Hz
    - remaining bans spread logarithmically above 1000Hz
  - new **mel-scaled** vectors
    - energy collected from each frequency band
    - logarithms of energy values — the human response to signal level is logarithmic

The Cepstrum

briefly:

- inverse DFT of the MEL scaled signal
- result — (usually 12) cepstral coefficients for each frame
- motivation — these coefficients are uncorelated, "they model the vocal tract better"

Deltas and Energy

- So far — 12 cepstral features
- 13th feature — **Energy**: obtained by suming squares of signal energy for all samples in the given frame
- For each feature:
  - **delta** cepstral coefficient obtained as a derivation of feature values, computed as

    $$d(t) = \frac{c(t+1)-c(t-1)}{2}$$

    (for particular cepstral value *c(t)* at time t)
  - **double delta** cepstral coefficient obtained as a derivation of **delta** cepstral feature values

MFCC Summary

- 39 MFCC features

```
12 cepstral coefficients
12 delta cepstral coefficients
12 double delta cepstral coefficients
1 energy coefficient
1 delta energy coefficient
1 double delta energy coefficient
```

Acoustic Likelihood Computation

- Last chapter — how to obtain feature vectors
- This chapter — how to compute likelihood of these feature vectors given an HMM state
  - i.e. how to obtain the **emission probabilities**

$$B = b_i(o_t) = p(o_t|q_i)$$

Vector Quantization

- simple method, not used in state of the art systems
- idea: map feature vectors into a small number of classes
- **codebook** — list of possible classes
- **prototype vector** — feature vector representing the class
- codebook is created by **clustering** of all the feature vectors in the training set into the given number of classes
- prototype vector can be chosen as a central point of each cluster
- each incoming feature vector is compared to all prototype vectors, the closest prototype vector is selected, and the feature vector is replaced by the class label of the selected prototype vector.
- disadvantage of this method: loss of specific information about the given feature vector, significant impact on performance
- advantage: emissions probabilities can be stored for each pair of HMM state and output symbol, Baum-Welch training conceptually easier

Gaussian Probability Density Functions

- more adequate method for modelling of emission probabilities
- used in state of the art systems
- for each HMM state, emission probability distribution over the space of possible feature vectors is expressed by the **Gausian Mixture Model** (GMM) (tj. jde o složení gaussovských funkcí, každá odpovídá skrytému stavu a je charakterizovaná střední hodnotou a rozptylem, ty se získávají z korpusu):

$$b_j(o_t) = \sum_{m=1}^{M} c_{jm} \frac{1}{\sqrt{2\pi|\Sigma_{jm}|}} exp[(o_t - \mu_{jm})^T \Sigma_{jm}^{-1}(o_t - \mu_{jm})]$$

μ_{jm} - mean of the "m-th" Gaussian of the state "j"

$$\Sigma jm$$

— covariance matrix of the *m-th* Gaussian of the state j

- Estimation of the GMM parameters (mean and covariance matrix) — **Baum-Welch algorithm**

Search and Decoding

- Bayes probability formula:

$$W^* = \arg\max_{W \in L} P(O|W)P(W)$$

- typically, more complex formula used:

$$W_* = \arg\max_{W \in L} P(O|W)P(W)^{LMSF} WIP^N$$

  - *LMSF* — language model scaling factor (kvůli tomu, že jednotlivé framy v $P(O|W)$ nejsou nezávislé, musíme provést vyvážení obou bayesovských komponent)
  - *WIP* word insertion penalty (vážení jazykového modelu má vedlejší efekt – snižuje penalizaci za příliš mnoho slov ve větě, to je nutné kompenzovat)
  - N — number of words in sentence

- decoding — **Viterbi algorithm**
  - finds the most probably sequence of HMM states
  - the output sentence can be easily constructed out of this sequence of HMM states
  - **beam search pruning**
    - at each trellis stage, compute the probability of best state/path *D*. prune away any state that is less probable than $D \times \Theta$, where Θ is the beam width (value lower than 1)

function VITERBI(*observations* of len T,*state-graph* of len N) **returns** *best-path*

create a path probability matrix *viterbi*$[N+2,T]$
for each state s **from** 1 **to** N **do** ;initialization step
  $viterbi[s,1] \leftarrow a_{0,s} * b_s(o_1)$
  $backpointer[s,1] \leftarrow 0$
for each time step t **from** 2 **to** T **do** ;recursion step
  for each state s **from** 1 **to** N **do**
    $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
    $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s}$
$viterbi[q_F,T] \leftarrow \max_{s=1}^{N} viterbi[s,T] * a_{s,q_F}$; termination step
$backpointer[q_F,T] \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T] * a_{s,q_F}$; termination step
return the backtrace path by following backpointers to states back in time from $backpointer[q_F,T]$

Figure 9.26 Viterbi algorithm for finding optimal sequence of hidden states. Given an observation sequence of words and an HMM (as defined by the A and B matrices), the algorithm returns the state-path through the HMM which assigns maximum likelihood to the observation sequence. $a[s',s]$ is the transition probability from previous state s' to current state s, and $b_s(o_t)$ is the observation likelihood of s given o_t. Note that states 0 and F are non-emitting start and end states.

Obrázek 26.5: (Speech and Language Processing (draft) — Jurafsky & Martin)

Embedded Training

- Recall: A — HMM transition probabilities, B — emission probabilities (modeled by GMMs)
- Given: phoneset, pronunciation lexicon and the transcribed wavefiles:
  - Build a whole sentence HMM for each training sentence (concatenation of HMM for distinct words)
  - Initialize A probabilities to 0.5 (for loop-back or for the correct next subphone) or to zero (for all other transitions)
  - Initialize B probabilities by setting the mean and variance for each Gaussian to the global mean and variance for the entire training set
  - Run multiple iterations of the Baum-Welch algorithm
- This process will optimize the A and B probabilities
  - i.e. for each HMM state corresponding to a distinct subphone (each phone modelled by 3 subphones), the probability of loop-back transition and leaving transition is reestimated as well as the parameters of GMM for the given state.
  - phone is modelled by the same parameters independently from which word it occurs in (i.e. same HMM for phone [a] in every word containing [a] phone) — "(this is not written anywhere in Jurafsky & Martin, but it seems to be intuitive and it's hopefully true:))

Evaluation: Word Error Rate

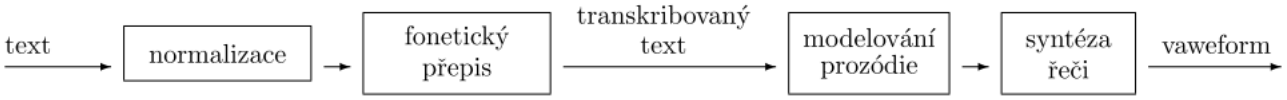
- **word error rate**: based on how much the word string returned by the recognizer differs from a reference transcription
- based on **minimum edit distance** in words between the hypothesized and correct string (number of **substitutions,** insertions **and** deletions"' needed to map between these sentences)

$$\text{Word Error Rate} = 100 \times \frac{\text{Insertions} + \text{Substitutions} + \text{Deletions}}{\text{Total Words in Correct Transcript}}$$

26.2 Speech Synthesis

Úvod

Základní schéma Text-To-Speech systému



- *Text* je většinou nějaký spec. případ, např. e-maily atp., text i transkripce většinou v nějaké formě obsahuje značky pro suprasegmentální fonémy.
- *Syntéza řeči* je jen jedna z částí TTS systému, to je nutné rozlišovat. Jde možná o nejsložitější a nejdůležitější součást, ale její vstup není text.
- *Normalizace* je někdy nutá provádět, někdy ne. Obsahuje např. vyházení hlaviček z e-mailů, příp. přidání spec. prozódie pro ně atd. Používá se, aby stejný TTS systém mohl být použit k různým věcem (předpřipravení textu podle jeho očekávaného typu).
- *Grafém, písmeno, letter* je nejmenší jednotka psané podoby jazyka. S některými jsou problémy, zda je považovat za jediný grafém (např. písmena s diakritikou), ale nám to většinou bude jedno.
- *Hláska, sound* je nejmenší jednotka zvukové podoby jazyka.
- *Foném* je nejmenší strukturální jednotka zvukové podoby jazyka, která rozlišuje význam.
- *Fonetický přepis* je přepis zvukové podoby textu, zaznamenávající hlásky, příp. suprasegmentální jevy. Může být postavená na pravidlech nebo na slovníku (ale vždy se používají oba komponenty, jeden slouží jako doplněk). Největší problém dělá přepis v jazycích, které např. neznačí samohlásky.

Fonetika a fonologie

Potřebujeme jednak popis výslovnosti, jednak popis akustiky (zvukových vln, jimiž se jednotlivé hlásky projevují). Můžu mít i popis vnímání (percepce) hlásek, ale ten pro naše účely není nezbytný.

Hlavní rozdíl mezi *fonetikou* a *fonologií* je ten, že fonetice jde o víceméně fyzikální akustický popis všech zvuků a hlásek, kdežto fonologii zajímá systém, struktura jazyka. Pro fonologický výzkum narozdíl od fonetiky potřebujeme alespoň nějaké základní informace o daném jazyce.

Akustika

Jednotlivé hlásky jsou složené zvuky (vlnění vzduchu), obsahující tónové (periodické) a šumové (neperiodické) složky. Rozlišujeme je právě podle složení jejich zvuku, např. konsonanty mají větší podíl šumových a vokály tónových složek, konsonanty se dále liší svou znělostí či neznělostí jako přítomností tónových složek.

Hlasové ústrojí se dá zjednodušeně představit jako zdroj zvuku (hlasivky) a rezonanční prostor (nadhrtanové dutiny). Hlasivky kmitají na nějaké základní frekvenci (F_0) a v nadhrtanových dutinách se zesilují některé harmonické frekvence.

Zvuk můžeme rozložit na frekvenční spektrum a zkoumat sílu zastoupení jednotlivých frekvencí v čase. Provádí se to běžně na počítači pomocí Fourierovy analýzy, výsledkem je spektrogram.

Výrazně zesílené frekvence číslujeme F_1, F_2 atd., a nazýváme *formanty*. Prvních několik je zastoupených v signálu "úmyslně", vlivem výslovnosti, příliš vysoké frekvence ale už člověk neovlivní. Proto při záznamu zvuku snímáme jen úzké pásmo (např. do 5 nebo 9 kHz). Frekvence, které jsou na spektru zvuku "úmyslně" potlačeny, nazýváme *antiformanty*.

U konsonantů (hlavně u neznělých) většinou nehledáme formanty, ale *transienty* – jde o "přechody" ve spektru, na místě, kde začínají nebo končí formanty okolních samohlásek. Místo, kam transienty ukazují (tedy hypotetický bod na spektru) pro danou souhlásku nazýváme *locus*. Bod locu je důležitý pro rozpoznání neznělých hlásek, považuje se za centrum jejich šumu.

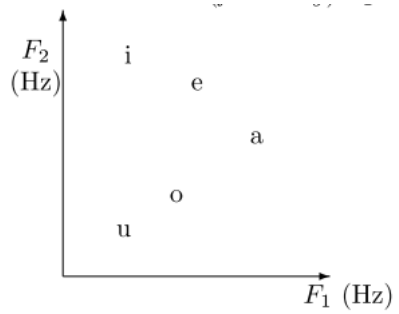
Samohlásky mají nejvýraznější formanty ($F_1, F_2, \ldots$). Pokud není u nějaké hlásky přítomna F_0 (a tedy ani výrazné vyšší formanty), jedná se o neznělou hlásku. Podle šumu (nekoncentrovaný signál po velké části spektra) se poznají frikativy (šumové hlásky). Explozivy (závěrové hlásky) se poznají okamžikem ticha a následným šumem exploze.

Fonetická abeceda IPA

Abeceda IPA slouží pro fonetickou transkripci, hlavně v jazykově nezávislém prostředí. Některé jazyky ji používají i pro zápis své výslovnosti, v některých se používají jiné abecedy, protože jsou pro ně šikovnější (např. v češtině). Hlásky, kterým je přiřazena jedna značka, se mohou napříč jazyky lišit – jde jen o aproximaci. Pro odlišení hlásek v rámci jednoho jazyka ale většinou plně dostačuje.

Vokalický systém

Extremální vokály jsou [i] (jazyk je nahoře vepředu), [u] (nahoře vzadu), [a] (dole uprostřed). Některé zvuky si v různých jazycích více či méně odpovídají, některé jazyky rozlišují více vokálů než jiné. Rozlišení může být podle několika různých vlastností najednou (a některé jejich kombinace nemusí být povolené).



Obrázek 26.6: První dva formanty českých samohlásek. Všimněte si, že zrcadlí místo jejich tvoření.

Pro akustický popis používáme tak krátké zvuky, že se nezmění pozice jazyka ("statický zvuk"), ale už se můžou analyzovat frekvence, ze kterých se zvuk skládá. Zajímáme se o lokální maxima frekvencí. Tím nalezneme nejnižší lokální maximum – základní frekvenci F_0 – a některé její vyšší harmonické frekvence. Podle prvních dvou formantů F_1 a F_2 lze samohlásky dobře odlišit (viz obrázek).

Konsonantický systém

Základní dělení konsonantů je na *znělé, voiced* a *neznělé, voiceless*. Ty se liší přítomností základního hlasového tónu (tj. kmitáním hlasivek při jejich tvorbě).

Podle místa artikulace rozlišujeme mj. *labiály (obouretné)* hlásky [p, b, m], *labiodentály (retozubné)* [f, v], *dentály (zubné)* [θ, ð], *prealveolární* [t, d, s, z, ts, dz], *postalveoláry* [š, ž], *palatály (tvrdopatrové)* [ď, ť, ň], *veláry (měkkopatrové)* [k, g, ch], *glotály (hlasivkové)* [h].

Podle způsobu tvoření rozlišujeme *plozivy, závěrové* [p, b, t, d, k, g, ť, ď], *nasální, nosové plozivy* [n, m, ŋ], *frikativy, šumové* [s, z, š, ž, ch, f, v], *afrikáty, polozávěrové* [ts, tš, dz, dž], *vibranty, trills* [r], *bokové hlásky, laterály* [l], *aproximanty* [j, w] (ty se vlastně fyzicky neliší od samohlásek, jde jen o popis).

Pro akustický popis konsonantů jsou určující transienty a bod locu. Locus se dá zhruba odhadnout podle místa tvoření – čím zadnější hláska (čím blíž je místo tvoření hlasivkám), tím vyšší je locus.

Pro nosovky je charakteristický nasální komponent na frekvenci cca 200-300 Hz (tedy pro vysoké hlasy nevýrazný) a potlačení formantu F_1 (vzniká antiformant).

Prozódie

Prozódie zahrnuje všechny vlastnosti, které se projevují nad hranicemi segmentů. Sestává z:

- F_0 *základní tón hlasu, voice pitch*
- *časování, timing*
- *intenzity* (není totéž, co hlasitost – je měřitelná, hlasitost je dojem; pro TTS ale není tak důležitá)

Vždy tu pracujeme jen s relativními hodnotami a prominencí (zvýrazněním) v některé z nich.

Hlásky ve skutečnosti existují až ve *slabikách*. Slabika je nejmenší část mluveného textu, která se dá zopakovat izolovaně – konsonant je vždy závislý na vokálu své slabiky a naopak. Vnímání slabik je jazykově závislé: když definuju slabiku jako "peak in sonority", bude slovo "lžu" sestávat ze 2 slabik.

Vyšší jednotka je *přízvukový takt, fonologické slovo, stress unit*. To je skupina slabik, z nichž na jedné je přízvuk. Přízvuk závisí na konkrétním jazyku a jedná se o kombinaci timingu, intonace i intenzity (prominence v některé z těchto hodnot).

Nad úrovní slov rozlišujeme *intonační jednotky, intonation contours*. Ty jsou relativně nezávislé, mezi nimi má člověk tendenci dělat pauzu v řeči. Jejich rozlišení ale není úplně přesné.

Nejvyšší jednotkou je celé *vyjádření, utterance*. Např. v dialogu odpovídá větě, ale může být i delší. Finální intonace vyjádření je terminální.

Melodie má v některých jazycích distinktivní funkci – stejné slabiky s jinou melodií mají jiný význam. Takové jazyky se nazývají *tónové*. Větná melodie ale může mít zároveň jinou funkci.

Mikroprozódie zahrnuje všechno, co se děje v rámci jedné hlásky, ale je ovlivňováno okolím. Má vliv na velkou prozódii. Je podvědomá, záleží i na konkrétních hláskách. Mikroprozodickým fenoménem je např. délka hlásky (záleží ale na tom, jestli délka hlásky rozlišuje význam). Další je např. změna tónu hlasu v rámci jedné hlásky. Běžný TTS systém se mikroprozodií nezabývá, protože ji má nahranou ve svém korpusu segmentů; prozódií se ale zabývat musí.

Stavba Text-To-Speech systému

Normalizace

Někdy se dohromady s normalizací textu dělá *chunking*, tj. rozdělení textu na dostatečně malé kousky pro zpracování, někdy je jako samostatný krok. Data se musí rozdělit někde, kde je to možné (ne např. uprostřed věty).

Fonetický přepis

Jde o přepis *letter-to-sound*, tedy přepisujeme grafémy na hlásky. Odlišují se dva přístupy:

- založený na *pravidlech, rule-based*
- založený na *slovníku, dictionary-based*

V dnešních systémech jsou v podstatě vždy přítomna i pravidla i slovník, ale jedna metoda je vždy primární, druhá doplňková.

Pro pravidla se de facto dají použít regulární výrazy, tj. přepisy se zapojením kontextu na obě strany. Většinou neoperují přímo nad textem, ale nad nějakými speciálně vytvořenými datovými strukturami.

Některými pravidly musí dojít k zjednoznačnění (disambiguization) textu, např. různou interpunkci apod. je nutné správně interpretovat – tečka např. může mít spoustu významů. Někdy se víc hodí (jazykově závislá) pravidla, někdy zas slovník, např. na interpretaci anglického členu "the" se hodí hlavně pravidla, ale bez slovníku to také nejde (srov. "the oak" proti "the one").

Pravidla můžu taky aplikovat buď jedním průchodem, nebo opakovaně. Typické druhy pravidel jsou následující:

1. *morfosyntaktická pravidla* – jedná se hlavně o určování slovních druhů apod. Používá se přitom hlavně slovník a statistické četnosti naměřené v nějakém korpusu. Příkladem takových pravidel může být i doplňování samohlásek v textech psaných souhláskovým písmem.
2. *kontextová pravidla* – Tato pravidla např. rozvíjejí zkratky, přibližují text čtené podobě.
3. *strukturální pravidla* – Výstup těchto pravidel se používá pro modelování prozódie – jde např. o identifikaci druhů vět, což umožní jejich správnou intonaci.
4. *pravidla fonetického přepisu (letter-to-sound)* – Tady se přímo převádí pravopis na výslovnost, mohou se používat různá pravidla pro výjimky (např. česky "diagram" změním na "dyagram", abych se vyhnul měkčení).

Modelování prozódie

Prozódie je vlastně ovlivňovaná "syntaxí", případně nějakými emocemi, jednotlivostmi mluvčího, ale ty se vystihnout nedají. Mělo by se dávat pozor i na mikroprozódii – tedy vystihnout prozodické fenomény, ale nenechat se zmást mikroprozodickými.

Intonace během řeči odpovídá změnám základní hlasové frekvence (F_0), na ostatních prozodických veličinách je víceméně nezávislá. Pro modelování intonace je nejznámější *Fudžisakiho model*. Ten sestává z *phrase commands* a *accent commands*. První typ pravidel je "trvanlivější", působí v podstatě na celou větu, vždy od daného času a s danou amplitudou (zvednutím nebo snížením F_0) a postupně doznívá. Druhý typ má kratší trvání, má definovaný čas začátku i konce a zase amplitudu. Výsledná F_0 v daném časovém bodě se (v logaritmické podobě) dá vyjádřit jako nějaká suma všech commandů, které působí, plus základní frekvence.

Prozodické modely je ale nutné nejprve natrénovat. Potřebuji tedy prozodický korpus, automatické nástroje na zpracování a prozodický model. Postup vytváření prozodického inventáře vypadá pak následovně:

$$\text{korpus} \rightarrow \text{detekce } F_0 \rightarrow \text{model} \rightarrow \underset{\text{rule extraction}}{\text{trénování /}} \rightarrow \text{pravidla (inventory)}$$

Krok *trénování, extrakce pravidel* probíhá buď automaticky za pomoci neuronové sítě (trénování), nebo ručně.

Výsledkem procesu by měl být *prosodic inventory*, tedy sada pravidel, jak upravovat prozodicky signál ve výstupu z TTS. Je to většinou malá množina nějakých hodnot – třeba informací o neuronové síti.

Syntéza řeči

Pro generování řeči ze zápisu hlásek se používá nějaký zjednodušený popis artikulace, podložený jistými předpoklady, tzv. *řečový model*. Pro syntézu existují dva hlavní druhy – buď *copy synthesis, konkatenační syntéza*, tedy syntéza na základě kopírování a slepování částí řečového inventáře, nebo *rule-based synthesis, formant synthesis*, syntéza založená na vytváření složeného zvuku za pomoci (frekvenčních) pravidel.

Pravidlová syntéza se používá většinou jenom v akademickém prostředí, až na pomůcky pro hyperrychlé čtení e-mailů. Projev většinou není příliš přirozený. Předpokládáme tu matematický model zjednodušeného artikulačního ústrojí a pravidla, popisující jeho změny. Ta pak zahrnují formanty samohlásek, transienty konsonantů, přítomnost základního tónu apod., všechno je v pravidlech relativně přímočaře. Model se vytváří ručně podle nějakého korpusu.

Kvalitu *syntézy založená na kopírování* určuje hlavně kvalita nahrávek v řečovém korpusu a také jejich reprezentativita. Korpus můžu získat dvěma způsoby – buď nahrávat televizní pořady, nebo výběrem vět, které někdo potom do korpusu přečte. Druhým způsobem můžu lépe pokrýt inventář cílového jazyka.

Chci mít výsledný korpus malý, aby ho mluvčí mohl přečíst najednou a bez změny podmínek (např. únavy hlasu). Postup je potom následující:

1. *identifikace hlásek* – Vyberu si, které hlásky potřebuji pro reprezentaci řeči v daném jazyce.
2. *identifikace fonotaktiky* – Zjistím, které kombinace vybraných hlásek se v jazyce vůbec můžou vyskytovat, tím zmenším počet kombinací.
3. *kompozice korpusu* – Ze všech možných kombinací hlásek, nalezených v předchozím kroku, složím psanou verzi korpusu.
4. *nahrávání korpusu* – Mluvčí přečte všechny věty, vložené do korpusu. Přitom by měl používat monotónní prozódii. Po nahrávání se vzorky normalizují na stejnou F_0.
5. *vytvoření řečového inventáře* – Protože pro každou kombinaci hlásek nepotřebuji více verzí, srovnám všechny dostupné a např. podle toho, jak moc se jejich F_0 blížila průměru, si vyberu tu nejlepší.

Pro výstup syntézy se nikdy nepoužívají samostatné hlásky, ale vždy kombinace dvou, tří nebo více hlásek, dvojhlásky apod. Projevuje se tu totiž důležitost koartikulace, navíc konsonanty jen "parazitují" na vokálech, samy stát nemohou, tedy samotné je extrahovat ani nemůžu. Pro konkatenaci zvuků potřebuji hlásky "stabilní", navíc vždy je potřeba nějaké vyhlazování zvuku. Tradičně se v konkatenační syntéze používají tzv. *diphones, dvojzvuky* – druhá polovina první, první polovina druhé hlásky, někde i delší úseky.

U složitějších systémů konkatenační syntézy nemám v řečovém inventáři pro každý diphone nebo triphone jen jednu zvukovou podobu, ale vybírám si z několika možností pomocí tzv. *unit selection algorithm* tu nejlepší pro dané místo v řeči. Přitom se zohledňuje prozódie, diskvalifikují se chyby výslovnosti apod., někdy se tak mohou použít i části slov úplně vcelku (na základě výběru). Pro výběr se používá upravený Viterbiho algoritmus (místo pravděpodobností přechodu uvažuju nějaké badness založené na podobné F_0, chybách výslovnosti, intenzitě a prozódi apod.).

Druhy Text-To-Speech Systémů

Time-Domain Pitch-Synchronous Overlap Add (TD-PSOLA)

Tento systém je příkladem konkatenativní syntézy, jde vlastně o velmi jednoduchý případ (dnes už relativně zastralý, používaný hlavně v 90. letech). Spočívá v tom, že každá hláska (jednotka řeči) je rozdělena na *framy*, krátké zvukové úseky během kterých se nemění F_0. V každém framu lze pozorovat *pitch-periody*, tedy jednotlivé kmity hlasu. Ty dávají možnost, jak měnit F_0 bez ohledu na kvalitu zvuku.

Mohu totiž jednotlivé framy skládat přes sebe a natahovat, pokud je upravím pomocí tzv. *windowing funkce* (funkce, která zesílí jen jednu pitch-periodu a postupně signál zeslabuje v jejím okolí až do ticha). Po použití windowing funkce na každou pitch-periodu pak výsledky můžu sečíst přes sebe i s nějakým posunutím. Tím dostanu signál, který může mít jinou F_0, ale jen neznatelně změněné vyšší frekvence (např. formanty). Někdy pitch-periody nesedí úplně přesně, ale díky windowing-funkci dojde k vyhlazení. Vyhazováním nebo duplikací pitch-period můžu (do určité míry) zvuk zrychlovat nebo zpomalovat.

Princip TD-PSOLA vypadá sice jednoduše, nutnou podmínkou jeho použití je ale spolehlivý detektor hlasové frekvence, jinak dochází k *chybě fáze, phase mismatch* hlasu (pitch-periody se netrefí přesně doprostřed kmitů). Na hranicích jednotlivých diphonů může dojít i k *chybě spektra, spectral mismatch*. Mám-li totiž dvě poloviny stejné samohlásky, které se trochu liší pod vlivem okolí, nedají se slepit úplně přesvědčivě. Poslední chybou, která se může v TD-PSOLA objevit, je *chyba výšky hlasu, pitch mismatch*. K té dojde, pokud dva přiléhající segmenty mají příliš odlišné F_0 (nesedí přesně na sebe).

Linear Prediction Coder (LPC) Speech Synthetizer

LPC syntéza vychází z modelu artikulačního ústrojí. Je to relativně stará technika, její výsledek ale nevypadá příliš přesvědčivě. Implementace v hardwaru ale není složitá, zvuk je srozumitelný i s minimálním inventářem.

Hlasové ústrojí si totiž lze představit jako na jedné straně otevřenou rezonační trubici (*tube*), ve které je na uzavřené straně zdroj zvuku (*buzzer*), který vytváří periodický signál. Když se nemění parametry tube ani buzzeru, pak vychýlení výsledné zvukové vlny v každém okamžiku (podle potřeb vzorkovací frekvence, kromě několika počátečních vzorků) se dá predikovat z určitého počtu předchozích vzorků.

Pro syntézu tedy vezmu řečový inventář a každou potřebnou jednotku rozdělím na *framy*, tedy časové úseky, kde jsou změny artikulace minimální. Pro každý frame potom odhadnu několik počátečních (např. 8) samplů, aby predikce vycházela s co nejmenší chybou. Tyto počáteční parametry se nazývají *LPC coefficients*. Odhad se typicky provádí metodou nejmenších čtverců.

Modelování pak provádím separátním ovládáním vlastností trubice i zdroje zvuku. Naměřené koeficienty nepoužívám pro generování zvuku přímo, protože mezi segmenty by vznikaly ostré předěly – zvuk se předem ještě vyhlazuje.

Problém je se simulací nosových hlásek, protože na to aproximace artikulačního ústrojí prostou trubicí nefunguje. Pokud bych chtěl trubici po části délky rozdělit, budu mít problém s nalezením počátečních LPC koeficientů.

Podobná technika (LPC komprese) se používá i v mobilních telefonech, protože aproximace parametrů je de facto druh ztrátové komprese.

Kapitola 27

Státnice I3: Vyhledávání a extrakce informací

27.1 Informační systémy

- Faktografické vs. dokumentografické
- Zpřístupnění vs. dodání dokumentu
- Indexace nutná – termy
  - řízená, neřízená
  - tezaury
- Kritérium predikce + maxima
- Precision, recall

27.2 Vyhledávání v textu

- Triviální algoritmus
- Knuth-Morris-Pratt
- Aho-Corrasicková

27.3 Boolské informační systémy

- Dokument reprezentován množinou termů, které ho vystihují
- Dotazy: AND, OR, NOT, wildcards, víceslovné, proximitní omezení, tezaurus, lemmatizace
- Invertovaný indexový soubor (org. po termech)
- Uspořádání výsledků (DNF, počet splněných konjunkcí)
- Zpětná vazba

27.4 Vektorové informační systémy

- Každý z n dokument reprezentování m-složkovým vektorem vah důležitostí termů ($\in [0,1]$)
- Indexový soubor je matice vah $m \times n$
- Dotaz je taky vektor, vyhodnocení a řazení pomocí:
  - základní $Sim(\vec{w}_i, \vec{q}) = \sum_{k=1}^{n} w_{i,k} q_k$
  - vylepšení na délku vektorů (počet nenulových w_k) – dělení $\sum w_i + \sum q$, $\sum w_i + \sum q - 2\sum wq$ nebo $\sqrt{\sum w_i^2 \cdot \sum q^2}$
  - jiné – normalizace na jednotkovou délku vektorů
- Nerozlišuje se disjunkce a konjunkce

- Negace = přidání záporných vah do dotazů
- Indexace podle *term frequency* – $TF_{i,j} = \frac{t_j}{\sum_{i=1}^{m} t_i}$ (podíl počtu výskytů daného termu v dokumentu z celk. počtu termů v něm)
  - Normalizovaná $NTF = \frac{1}{2} + \frac{TF}{2\max(TF)}$ (do $\{0\} \cup [1/2, 1]$).
  - Inverzní $ITF_j = \log(n/k)$, pokud se term j vyskytuje v k dokumentech z n.
- Výpočet vah $w = \frac{NTF \cdot ITF}{Z}$ (Z je normalizace)
- Matice podobnosti termů – závislost a zastupitelnost termů

27.5 Induktivní systémy

- Dvouvrstvá neuronová síť se zpětnou aplikací vah (1. vrstva — termy, 2. — dokumenty)
- Laterální inhibice – zabránění nárůstu vah

27.6 Signaturové systémy

- Uložení na pomalých médiích – předstupeň k lepší metodě
- Každý dokument i search term má signaturu, která funguje jako maska (pokud je bitový and signatury dokumentu a termu nenulový, je dokument možná relevantní a použije se k detailnímu hledání)
- Přiřazení signatury – každý term: jedna jednička na nějakém místě / hashovací funkce
  - Zabránění příliš mnoha jedničkám v signaturách dokumentů – rozdělení na bloky (pevné délky nebo pevného počtu jedniček v signatuře)
- Wildcardy obecně nejsou možné, jen s monotónními signaturami

27.7 Rozšířená boolská logika

- Reprezentace stejná jako vektorový model
- Dotazy stejné jako s boolskou logikou, ale s váhami (pokud nejsou uvedeny, bere se 1)
- OR – vzdálenost od nulového dokumentu $DF = (0, \ldots, 0)$ jako $\sqrt[p]{\frac{q_a^p w_{i,a}^p + q_b^p w_{i,b}^p}{q_a + q_b}}$ (kde q_a, q_b jsou váhy dotazu)
- AND – vzdál. od jednotkového dokumentu jako $1 - \sqrt[p]{\frac{q_a^p (1-w_{i,a})^p + q_b^p (1-w_{i,b})^p}{q_a + q_b}}$
- Pro $p = 1$ je to vlastně vektorový model, pro $p \to \infty$ se blíží k boolskému

27.8 Rozlišovací hodnoty termů v indexu

- Informace o tom, jak dobře termy rozlišují dokumenty – co se stane, když nějaký z nich vyhodíme
- Rozlišovací hodnota $DV_k = Q^{(k)} - Q$, kde $Q = \frac{\sum_{i=1}^{n} Sim(d_i, C)}{n}$ je průměrná podobnost dokumentů s *centroidem* ("průměrným dokumentem" $C = \frac{\sum_{i=1}^{n} d_i}{n}$) a $Q^{(k)}$ je totéž, odstraníme-li k-tý dokument.
- Je možné použít jako IFT, má lepší vlastnosti než ten logaritmus (viz výše)

27.9 Přibližné hledání

- Detekce chyb, nalezení blízkých termů ve slovníku:
  - Počet společných digramů
  - Hammingova míra (počet operací replace při doplnění slova znakem λ na stejnou délku)
  - Levenshteinova míra (počet operací replace, insert nebo delete)
- Lze použít konečné automaty

Kapitola 28

Státnice I3: Strojový překlad

28.1 Proč je strojový překlad těžký?

- Strukturální rozdíly mezi jazyky
  - Různé způsoby řazení podmětu, slovesa a předmětu — SVO, SOV, VSO languages
  - Head-marking

```
English: the man's house
Hungarian: az ember háza
           the man house-his
```

- – Pro-drop languages: Některé jazyky umožňují vynechávání zájmen
- Lexikální rozdíly
  - *Překlad homonym*: slovo ve zdrojovém jazyce může mít více významů, každý význam je potřeba přeložit jiným slovem cílového jazyka (anglické slovo **bass** může označovat jak hudební nástroj tak i druh ryby, tomu odpovídají dva různé překlady do španělštiny)
  - *polysémie*: *koruna* v češtině znamená jak část stromu, tak ozdobu hlavy – významy spolu souvisí. V angličtině ale tahle souvislost není vidět a překládá se *treetop* a *crown*.
  - *distinkce jazykového významu*: anglické slovo **know** označuje jak znalost faktu tak i znalost osoby či místa a je to jeden význam. Ve francouzštině odpovídají těmto významům 2 různá slovesa **connaitre** a **savoir**.
  - Slovo/fráze jednoho jazyka nemusí mít ekvivalent v druhém jazyce.

28.2 Úkoly strojového překladu

Zatím se nepodařilo vytvořit plně automatický systém, který by se mohl měřit s živými překladateli co se týče kvality překladu. Přesto je strojový překlad užitečný, používá se pro řešení následujících úkolů:

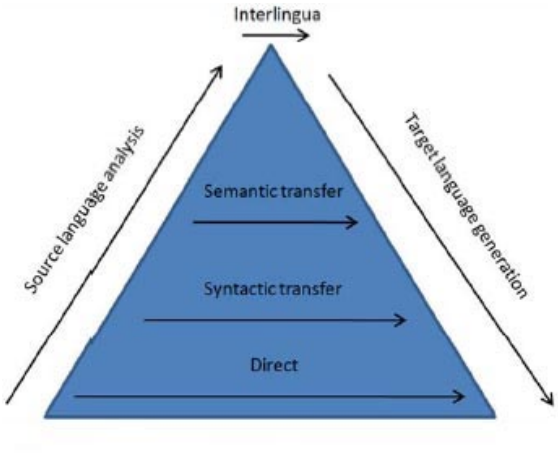
- **rough translation** — orientační překlad nízké kvality (Např. Google translate). Cílem je, aby uživatel neznalý daného jazyka mohl s určitým úsilým porozumět obsahu dokumentu/webové stránky.
- **computer-aided human translation*** — Výstup strojového překladu musí být natolik kvalitní, že je pro překladatele rychlejší provést drobné změny ve výstupu MT než psát vlastní překlad od nuly.
- **subdomain translation** — plně automatický kvalitní překlad na velmi omezené doméně: Např. překlady předpovědí počasí — velmi omezená slovní zásoba, omezená množina jazykových konstrukcí. (Další domény: software manuals, air travel queries, appoitment scheduling, restaurant recommendation).

28.3 Překladový trojúhelník (Vauquois triangle)

Toto schéma popisuje MT jako proces, který lze rozdělit do tří kroků:

- **Analysis** — ligvistická analýza zdrojové věty, může zahrnovat morfologickou, syntaktickou, sémantickou analýzu. Účelem je vytvořit lingvistickou reprezentaci zdrojové věty vhodnou pro překlad.
- **Transfer** — Převod lingvistické reprezentace zdrojové věty do lingvistické reprezentace cílové věty.
- **Generation** — Vytvoření povrchové reprezentace věty v cílovém jazyce.

Trojúhelníkový tvar naznačuje, že při hlubší analýze je transfer jednodušší.



Obrázek 28.1: Vauquois triangle

28.4 Metody Evaluace

28.5 Rule-based strojový překlad

Přímý překlad

- Analysis — pouze morfologická analýza
- Transfer — Překlad jednotlivých slov/frází za pomocí dvojjazyčného slovníku (pravidla ve formě Rozhodovacích stromů), local reordering (záměna pořadí slov/frází)
- Generation — morphological generation

```
//Příklad: Překlad slov much a many z angličtiny do ruštiny
//
if (preceding word is how)
  return skol'ko;
else if (preceding word is as)
  return skol'ko zhe;
else if (word is much)
{
  if (preceding word is very)
    return nil;
  else if (following word is a noun)
    return mnogo;
}
else
{
  if (preceding word is a preposition and following word is a noun)
    return mnogii;
  else
    return mnogo;
}
```

Rule-based překlad se zapojením syntaktické analýzy

V rámci analýzy je proveden syntaktický parsing. Strom zdrojové věty je převeden na strom cílové věty pomocí **contrastive knowledge** — znalosti rozdílů mezi jazyky (Příklad: Při překladu z SVO jazyka do SOV jazyka bude v syntaktickém stromě prohozeno pořadí uzlu odpovídajícího slovesu s uzlem odpovídajícím předmětu)

Combined Approach

- Analysis
  - Morphological analysis and part-of-speech tagging

  - Chunking of NPs, PPs, and larger phrases
  - Shallow dependency parsing (subject, passives, head modifiers)

- transfer
  - Translation of idioms
  - Word sense disambiguation
  - Assignment of prepositions according to governing verbs

- synthesis
  - Lexical translation with a rich bilingual dictionary
  - Reorderings
  - Morphological generation

Překlad pomocí sémantické analýzy — Interlingua

Interlingua označuje jazykově nezávislou reprezentaci významu. Překladové modely pracující na bázi *interlingvy* nejprve vytvoří jazykově nezávislou semantickou reprezentaci výchozí věty, na jejím základě pak zkonstruují cílovou větu (žádný transfer, pouze analýza a generování). Použití v systémech pro překlad textů z velmi omezené domény — model sémantiky lze vytvořit (předpověď počasí, rezervace letenek atd.)

28.6 Statistický strojový překlad — phrase-based

značení: z historických důvodů E resp. e označuje větu resp. frázi cílového jazyka, F resp. f větu resp. frázi výchozího jazyka (První překladové systémy: francouzština => angličtina).

- **základní model — Noisy-channel model**

$$E^* = \arg\max_E P(E|F) = \arg\max_E \frac{P(F|E)P(E)}{P(F)} = \arg\max_E P(F|E)P(E)$$

- State of the art systémy používají obecnější "'log-linear model'

$$E^* = \arg\max_E exp[\sum_{m=1}^{M} \lambda_m h_m(E,F)]$$

h_i označuje libovolnou **feature function**

Feature functions

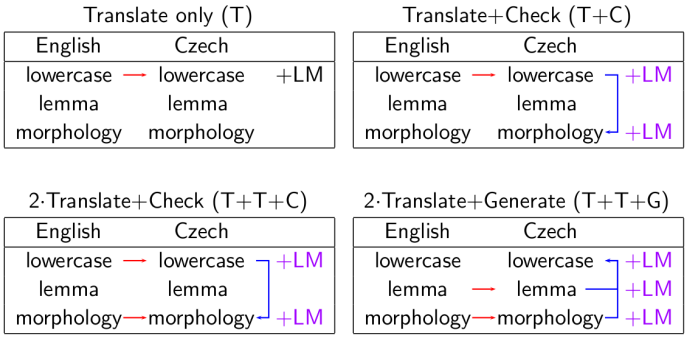
- **Jazykový model** *P(E)*: klasický n-gram language model. Překladové systémy běžně kombinují více jazykových modelů na různých faktorech (slovní formy, lemmata, morfologické tagy...)

$$h_L = \log(P(E))$$

- **Překladový model** *P(F|E)*

$$h_T = \log(P(F|E))$$

- **Word penalty** — penalizace hypotéz, které obsahují příliš málo/příliš hodně slov.
- **Unknown-word penalty**
- ...atd



Obrázek 28.2: Translation scenarios (picture by Ondřej Bojar)

Vícefaktorový překlad — více různých jazykových modelů

- Both input and output words can have more factors (a ty jsou pak použity jako featury loglineárního modelu).
- Arbitrary number and order of:
  - Mapping steps (red arrows): Translate (phrases of) source factors to target factors.
  - Generation steps (blue arrows)
    - Generate target factors from target factors.

`dvě` → `fem-nom; dva` → `masc-nom`

-
  - ∗ ⇒ Ensures "vertical" coherence.
  - Target-side language models (+LM)
    - Applicable to various target-side factors.
    - ⇒ Ensures "horizontal" coherence.

Překladový model P(F|E)

Úkolem překladového modelu je odhadnout pravděpodobnost toho, že E "generuje" F. Generování se skládá ze tří kroků

- Slova z E jsou rozdělena do frází $e_1, e_2 ... e_I$
- Každá fráze e_i je přeložena jako f_i
- reordering frází f_i

Odhad pravděpodobnosti je založen na

- **translation probability**: $p(f_i, e_i)$
  - pravděpodobnost, že e_i bude přeložena jako f_i. (Bayes tu obrátil směr překladu)
  - pravděpodobnost stanovena na základě alignovaného paralelního korpusu (alignment — mapování mezi frázemi zdrojových a cílových vět v paralelním korpusu): $p(f, e) = \frac{count(f,e)}{count(e)}$
- **distortion probability**: $d(a_i - b_{i-1}) = \alpha^{|a_i - b_{i-1} - 1|}$
  - a_i: počáteční pozice f_i
  - b_{i-1}: počáteční pozice e_{i-1}
  - Neformálně: ...Čím víc reorderingu, tím menší pravděpodobnost...

$$P(F|E) = \prod p(f_i, e_i) d(a_i - b_{i-1})$$

Hledání nejlepší překladové hypotézy

Dekodér

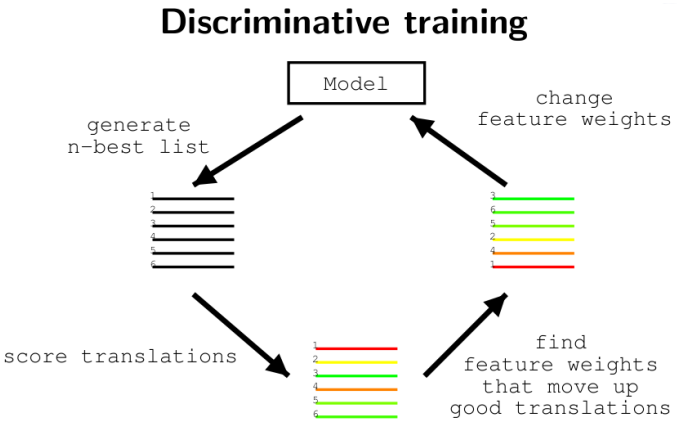
- Úkolem dekodéru je nalézt **N-best list** nejlepších překladových hypotéz na základě pravděpodobnostního modelu
  - Ve fázi dekódování bývá často použit jednodušší pravděpodobnostní model (pouze podmnožina všech *feature functions*) — tzv. generativní model

  modely na morfologických faktorech atd...)
- algoritmus: A* search
  - jeho varianty používané v MT a speech recognition běžně označované jako **stack decoding**
  - heuristika: best-first search
  - beam search pruning
- Popis dekodéru včetně názorných obrázku je možné nalézt zde.

Rescoring, Discriminative model

- Na výstupu dekodéru je N nejlepších překladových hypotéz podle **generativního modelu**.
- Tyto hypotézy jsou ohodnoceny pomocí podrobnějšího **Diskriminativního modelu** (všechny feature functions), je vybrána ta nejlepší
  - Motivace: Některé *feature functions* není možné vyhodnocovat pro částečné hypotézy nebo by to bylo moc drahé (časová náročnost)
  - Příklad (můj vlastní):
    * Generativní model vytvoří N-best list na základě jazykového modelu P(E) a překladového modelu P(F|E).
    * Na všech překladových hypotézách bude proveden HMM part-of-speech tagging, pravděpodobnost otagování $\prod_{i=1}^{n} P(w_i|t_i) \prod_{i=1}^{n} P(t_i|t_{i-2}, t_{i-1})$ bude použita jako další feature v diskriminativním modelu.

Minimum Error Rate Training (MERT)



Obrázek 28.3: MERT — discriminative training (picture by Phillip Koehn)

- Nalezení optimálních vah $\lambda_1 ... \lambda_M$ pro jednotlivé *feature functions*.
- Provádí se na *held-out* datech — část paralelního korpusu, která nebyla použita při trénování modelu.
- Och's minimum error rate training (MERT):

```
given: sentences with n-best list of translations
iterate n times
  randomize starting feature weights
    iterate until convergences
      for each feature
        find best feature weight
        update if different from current
return best feature weights found in any iteration
```

28.7 Alignment

Sentence alignment

- Classical algorithm: Gale and Church (1993).
  - Based on similar character length of aligned sentences, no words examined.
  - Dynamic-programming search for the best alignment.
  - Allows 0 to 2 sentences in a group: 0-1, 1-0, 1-1, 2-1, 1-2, 2-2

Word alignment

- Goal: Given a sentence in two languages, align words (tokens).
  - Lexical probabilities: IBM Model 1,
  - Expectation-Maximization Loop for IBM1.
  - Symmetrization techniques.
  - The "standard tool": GIZA++ (Och and Ney, 2000)
  - **Details here!**

28.8 Evaluace

- evaluace MT je subjektivní a netriviální úkol
- výzkum metodologie evaluace hrál ve vývoji MT vždy důležitou úlohu

"Ruční" evaluace

- Kvalitu překladů posuzují dobrovolníci
- Zvlášť se hodnotí:
  - **přesnost** (fidelity) — věrnost zachycení obsahu sdělení
  - **plynulost** (fluency) — zda je věta dobře srozumitelná, jasná; styl

Vyhodnocení *plynulosti*

- nejjednodušší metoda: dobrovolníci přiřazují skóre každé větě (např. od 1 do 5)
- **cloze** — některá slova na výstupu jsou zakryta, dobrovolník má za úkol uhádnout, jaké slovo je zakryto. Čím větší *plynulost*, tím je hádání jednodušší
- metoda měření času potřebného pro přečtení věty — čím větší plynulost, tím se věta čte snadněji

Vyhodnocení *přesnosti*

- nejjednodušší metoda: dobrovolníci přiřazují skóre každé větě (např. od 1 do 5)
- kladení otázek týkajících se obsahu textu — dobrovolník má odpovídat pouze na základě informací obsažených v překladu

Metody měření celkové kvality

- všechny aspekty (*plynulost*, *přesnost*) dohromady
- **edit cost of post-editing**
  - kolik slov je potřeba upravit, aby bylo dosaženo překladu rozumné kvality
  - kolik je potřeba stisknutí kláves, aby bylo dosaženo překladu rozumné kvality

Automatická evaluace

- horší kvalita, za to však výrazně levnější a rychlejší než manuální evaluace
- díky tomu, že se dá automatická evaluace provádět rychle a často, může být použita pro optimalizaci parametrů modelu (váhy jednotlivých features jsou nastaveny tak, aby bylo co největší **BLEU** score na *heldout* datech), nebo k posouzení zlepšení systému při změnách v implementaci.
- automatické metriky: **BLEU, NIST, TER, METEOR**
- všechny tyto metriky jsou založeny na porovnávání výstupu systému s **referenčnímy překlady** — lidmi vytvořené kvalitní překlady. Pro každou větu z testovací množiny je k dispozici více referenčních příkladů (věta se dá obvykle dobře přeložit více způsoby)
- jednotlivé metriky se od sebe navzájem liší tím, jak počítají podobnost mezi výstupem systému a referenčnímy překlady

BLEU

- nejpoužívanější metrika
- založená na **modified n-gram precision**
  - unigram precision: Počítá se poměr, kolik slov z výstupní věty je obsaženo v nějakém referenčním překladu.

```
Output: the the the the the the the
Reference 1: the cat is on the mat
Reference 2: there is a cat on the mat
```

-
  - V uvedeném příkladě je *unigram precision* $7/7 = 1$, protože všech 7 slov se vyskytlo v nějakém ref. překladu.
  - Uvedený příklad poukazuje na problém — vysoké ohodnocení (maximální hodnota precision), přestože je na výstupu velmi špatný překlad
  - problém odstraní **modified precission**
    - pro každé slovo výstupní věty se spočítá $Count_{clip}$ — upravený počet výskytů
      - Pokud je počet výskytů (*Count*) slova ve výstupní větě menší nebo roven počtu výskytů v některém z referenčních výskytů, tak $Count_{clip} = Count$
      - V opačném případě bude $Count_{clip}$ rovno maximálnímu počtu výskytů slova v referenčních větách
  - V uvedeném příkladě bude **modified unigram precision** rovna $2/7$, $Count_{clip}(the$ je totiž rovno 2. (Maximální počet výskytů slova *the* v některém z referenčních překladů je 2).
- *modified n-gram precission* pro n-gramy vyšších řádů se určuje stejným způsobem. Výpočet *modified precision* přes celou testovací množinu:

$$p_n = \frac{\sum_{O \in Output} \sum_{n\text{-}gram \in O} Count_{clip}(n\text{-}gram)}{\sum_{O' \in Output} \sum_{n\text{-}gram' \in O'} Count_{clip}(n\text{-}gram')}$$

Output: množina výstupních vět pro celá testovací data

- penalizace příliš krátkých překladů: **brevity penalty (BP)** (nelze použít *recall*, protože máme více referenčních překladů)

c: součet délek všech výstupních vět

r: součet délek nejpodobnějších referenčních překladů k výstupním větám

$$BP = \begin{cases} 1 & if\ c > r \\ e^{(1-r/c)} & if\ c <= r \end{cases}$$

- BLEU score je definováno jako harmonický průměr *modified ungram precision* pro n-gramy do řádu N normalizovaného pomocí *brevity penalty*

$$BLEU = BP \times exp(\frac{1}{N} \sum_{n=1}^{N} \log(p_n))$$

28.9 Historie

- První patenty – už 1930's: G. Artsrouni – automatický slovník na děrné pásce, P. Troyanskii – lidský editor, vyjadřující log. formy a synt. funkce, automatický překlad a editor, který přepíše log. formu cílového jazyka do textu.
- 1946 – A. D. Booth: automatický slovník, překlad slovo od slova, 1949 W. Weaver – informační teorie, desambiguace na zákl. kontextu, kryptografické metody, univerzálie
- 1948 – R.M.Richens – slovník s kořeny, předponami a příponami zvlášť
- 1950 – E. Reifler – **preediting** a **postediting** (zjednodušení textu pro účely překladu, oprava chyb, které udělal stroj)
- 1952 – 1. konference na MIT, L. Dostert – pivotní jazyk pro překlad do více jazyků
- 1954 – Georgetownský experiment: Rusko-anglický text o 250 slovech, 6 synt. pravidel, bez negací, slovesa ve 3. osobě, málo předložek; byl vidět úspěch, zkouší to další
- 1955 – Anglicko-ruský překlad v Moskvě
- 1956 – První mezinárodní konference
- 1957 – Chomsky: Syntactic Structures
- 1960 – Yehoshua Bar Hillel: "Fully automatic high quality machine translation is not feasible.", 1966 ALPAC (Amer. Lang. Processing Advisory committee) – zpráva, která způsobila útlum, mimo USA výzkum pokračoval

Projekty po ALPACu:

- SYSTRAN, Grenoble (GETA), SUSY (Saarbrücken), LOGOS (Texas), TAUM (Montreal), ETAP (Moskva)

TAUM METEO (1976)

- Montreal, překlad meteorologických zpráv z angličtiny do francouzštiny (wen:TAUM system)
- dobře definovaná a správně omezená podmnožina syntaxe a sémantiky
- vhodná implementace (Q-systémy), systém sám rozpozná, že text neumí přeložit
- praktická implementace METEO System fungovala až do 2001 (wen:METEO System)

SYSTRAN

- překlad dokumentů EU, přímý (každý pár zvlášť, cca 20, uspokojivě jen AJ, FJ, NJ, wen:SYSTRAN)
- data oddělena od programu
- řešeno ad-hoc

EUROTRA

- oficiální projekt EU v 80. letech, pokus nahradit Systran (72 jazykových párů, v každé zemi jedno centrum, wen:Eurotra)
- nezvládnutá modularita (každý si měl analyzovat sám, domlouvat se na rozhraní)
- negativní efekt

VERBMOBIL

- Německý nástupce Eurotry, víc jak 30 univerzit; překlad mluvené řeči: domluva obchodníků na příští schůzce
- Patent, prezentace na EXPO 2000, pak ticho

28.10 Systémy podporující překlad

- Využití dříve přeložených textů, princip **překladové paměti** (wen: Computer-assisted translation)
- IBM Translation Manager, Déja Vu, SDL TRADOS – prodává se sám systém, paměť si překladatel zajistí sám
- Hledání shodných úseků, oprava odlišností
- Zejména pro překlady dokumentace k systémům různých verzí
- Dnes se kombinují se statistickým překladem

28.11 České systémy

(převzano z wiki poznámek Úvod do počítačového zpracování přirozeného jazyka)

První překlad 1957 – jedna věta na SAmočinném POčítači: *"The consonants have not by far been investigated to the same extent as the vowels."* — *„Souhlásky zdaleka nebyly prozkoumány do stejné míry jako samohlásky."* Později se tu objevily Q-Systémy, takže se začaly psát gramatiky.

APAČ (80. léta)

- Z. Kirschner, slovník pokrýval oblast vodních pump (dokumentace), cca 1500 slov; Q-systémy
- **Transdukční slovník** pro latinské výrazy: *-zation* -
  textgreater *-zace*, *-ic* -
  textgreater *ický* atd., seznam výjimek

Ruslan (1985-1990)

- Překlad manuálů sálových počítačů z češtiny do ruštiny
- Slovník: cca 8500 slov, transdukční slovník (ale příbuznosti jazyků se u něj využít nedalo), Q-systémy
- Použití synt. transferu, očekával se minimální, ale ten stále rostl
- Tehdy na PC 286 trval překlad 1 věty asi 4 minuty, dnes 4 vteřiny
- Spec. kódování: háček = "3" za písmenem, čárka = "2" za písmenem, kroužek = "7" (*-1- $n(5) + $gs(1) + textless + Z3LUT3OUC3KY2 + KU7N3 + U2PE3L + D3A2BELSKE2 + O2DY + . + textgreater -2-*).
- Před operačními zkouškami vývoj ukončen

Česílko (od 1998)

- Překlad příbuzných jazyků, kvůli překladům dokumentací: lidský překlad z angličtiny do češtiny a odtud automaticky do slovenštiny a polštiny. Následně se výsledek opravuje.
- Morfologické slovníky, statistická analýza češtiny
- Využívá (většinou) shodné syntaxe, jsou tu ale odlišné slovníky (ač jistá pravidelnost) a úplně odlišné tvarosloví

PC Translator

- Komerční systém, založený na pravidlech, vyvíjený už hodně dlouho.

Příloha A

Document Information & History

A.1 PDF Information

This PDF was compiled from LaTeX on 19. září 2010, based on the 30 August 2010.

A.2 Authors

Michalisek, Rajjo, Stevko, Tuetschek, and anonymous contributors.