Předmluva

I když počítač byl vytvořen především jako stroj na provádění výpočtů, postupně se do programů vkládaly části, které pracovaly s textovými řetězci. Nejdříve šlo jen o přidávání textových informací do výstupních dat, které sloužily pro jejich lepší čitelnost. Dalším krokem byla možnost čtení textových řetězců, které byly do výstupních dat ze stejných důvodů přidávány buď beze změny nebo jen s malými změnami. Později bylo možno vstupní text uchovat jako hodnotu proměnné, což přímo vedlo k možnosti zpracování textu. Tento vývoj je možno pozorovat na změnách v možnostech programovacích jazyků. Nejlépe je tato skutečnost patrná z vývoje jazyka FORTRAN.

Typickým jevem pro prostředky výpočetní techniky je pevný formát dat, a proto se v začátcích zpracování textu používaly především řetězce pevné délky, tabulky, formuláře. V řadě případů se textové informace pro zpracování na počítači kódovaly pomocí čísel. Tomuto jevu odpovídal i vývoj databázových systémů. Databázové systémy se začaly používat především pro práci s číselnými údaji a s formátovanými textovými údaji. Všechny základní modely databázových systémů (relační, síťový, hierarchický) jsou orientovány na práci s formátovanými daty.

Zpracování neformátovaných textových informací na počítačích se vyvíjelo poměrně pomalu, i když je zřejmé, že většina textů, se kterými se v životě setkáváme, má volný neformátovaný tvar. Tento tvar vyhovuje čtenářům. Jedná se o knihy, časopisy, noviny, úřední spisy, zákony, příkazy a podobné dokumenty.

Současná doba je charakterizována jako období informační exploze, což především znamená prudký nárůst vytváření textových materiálů. Výpočetní technika se v této oblasti uplatňuje v mnoha směrech:

- při přípravě textu se používají textové editory a systémy pro kontrolu jazykové správnosti textu,
- při uchování textu se používají nejrůznější způsoby organizace textu tak, aby potřebný prostor pro uchování textu byl co nejmenší a (nebo) aby bylo možno v textu co nejrychleji nalézt potřebnou informaci,
- při přenosu textu se používají metody zabezpečující text proti chybám a metody komprese textu,
- při vyhledávání informací v textu se používají postupy umožňující rychlé nalezení požadované informace,
- při výstupu textu pro prezentaci se používají formátovací systémy umožňující připravit přehledný a estetický výstupní dokument.

Tento učební text je určen jako učební pomůcka pro studium předmětu Textové informační systémy. Vzhledem k tomu, že pojem informační systém je velice široký, bylo nutno zaměření tohoto textu zúžit. V textu jsou podrobně rozebrány z algoritmického hlediska zejména tyto tři problémy:

1. vyhledávání v textu,
2. komprese textu,
3. kontrola správnosti textu.

Kromě těchto základních skupin problémů textových systémů jsou dále zmíněny otázky základních principů organizace a použití informačních systémů.

Na druhé straně je třeba uvést, že v textu nejsou uvedeny kapitoly pojednávající o přibližném vyhledávání a o hypertextových systémech. Obě tyto oblasti se rychle rozvíjejí a příprava příslušných kapitol si vyžádá delší dobu.

Podkladem pro tento učební text byl především učební text pro postgraduální kurz Inženýrská informatika (Melichar,B.: Informační systémy, ES ČVUT Praha, 1991) a přehledová zpráva vydaná na katedře počítačů (Melichar,B., Pokorný,J.: Data Compression, Survey Report DC-92-04, April 1992).

Na přípravě předlohy pro tisk v systému LATEX se podíleli studenti v rámci cvičení z předmětu Textové informační systémy ve školním roce 1993/94. Jim děkuji za přípravu první verze učebního textu. Dále děkuji RNDr. Aleně Balvínové a Ing. Martinu Blochovi, CSc. za pečlivé přečtení pracovní verze textu a za jejich cenné náměty, které byly využity při přípravě finální verze textu. Můj dík patří také RNDr. Aleně Balvínové, Olze Vrtiškové a Janě Hanzlíkové za přípravu konečné verze předlohy pro tisk.

Mníšek pod Brdy, červenec 1994 Bořivoj Melichar

Za první rok po vydání tohoto učebního textu byl celý náklad rozebrán. Proto byl připraven dotisk, ve kterém jsou opraveny chyby, které byly nalezeny. Děkuji všem studentům, kteří mi při tom vydatně pomáhali.

Mníšek pod Brdy, květen 1996 Bořivoj Melichar

Obsah

1 Základní pojmy informačních systémů **7**

2 Klasifikace informačních systémů **8**

3 Vyhledávací systémy **10**

4 Vyhledávací algoritmy **11**

4.1 Elementární algoritmus 12

4.2 Vyhledávací metody s předzpracováním vzorků 13

4.2.1 Vyhledávací metody s předzpracováním vzorků se sousměrným vyhledáváním 13

4.2.1.1 Vyhledávací stroj 13

4.2.1.2 Knuth-Morris-Prattův algoritmus 14

4.2.1.3 Algoritmus Aho-Corasickové 16

4.2.1.4 Vyhledávací konečné automaty 19

4.2.1.5 Vyhledávání nekonečné množiny vzorků 23

4.2.2 Vyhledávací metody s předzpracováním vzorků s protisměrným vyhledáváním 34

4.2.2.1 Boyer-Mooreův algoritmus 34

4.2.2.2 Algoritmus Commentz-Walterové 36

4.2.2.3 Protisměrné vyhledávání nekonečné množiny vzorků 39

4.2.2.4 Dvoucestný automat se skokem 41

4.2.3 Shrnutí 43

4.3 Vyhledávací metody s předzpracováním textu – indexové metody 44

4.3.1 Implementace indexových systémů 45

4.3.1.1 Použití invertovaného souboru 45

4.3.1.2 Použití seznamu dokumentů 45

4.3.1.3 Souřadnicový systém s ukazateli 45

4.3.2 Metody indexování 47

4.3.2.1 Analýza textu 47

4.3.2.2 Jednoduchá metoda automatického indexování 49

4.3.2.3 Řízené indexování 49

4.3.2.4 Konstrukce tezauru 50

4.3.2.5 Vyhledávání vzorků pomocí fragmentových indexů 51

4.4 Vyhledávací metody s předzpracováním textu a vzorků – signaturové metody . 51

4.4.1 Řetězené a vrstvené kódování 52

4.4.2 Metody tvorby signatur 53

4.4.3 Vyhledávání vzorku pomocí signatury a indexu 54

4.5 Vyhledávání vzorků pomocí dvojitého slovníku 54

5 Jazyky pro vyhledávání **56**

6 Komprese dat **60**

6.1 Základní pojmy komprese dat . . . 60
6.1.1 Kódování a dekódování . . . 60
6.1.2 Entropie a redundance . . . 62
6.2 Predikce a modelování . . . 62
6.3 Reprezentace celých čísel . . . 64
6.3.1 Fibonacciho kódy . . . 64
6.3.2 Eliasovy kódy . . . 65
6.4 Statistické metody komprese dat . . . 67
6.4.1 Shannon-Fanovo kódování . . . 67
6.4.2 Huffmanovo kódování . . . 68
6.4.2.1 Statické Huffmanovo kódování . . . 69
6.4.2.2 Vlastnosti Huffmanových stromů . . . 70
6.4.2.3 Adaptivní Huffmanovo kódování . . . 72
6.4.2.4 Implementační poznámky . . . 74
6.4.3 Aritmetické kódování . . . 76
6.5 Slovníkové metody komprese dat . . . 78
6.5.1 Statické slovníkové metody . . . 79
6.5.2 Semiadaptivní slovníkové metody . . . 79
6.5.3 Adaptivní slovníkové metody . . . 80
6.5.3.1 Metoda posuvného okna . . . 80
6.5.3.2 Metody s rostoucím slovníkem . . . 82
6.5.3.3 Slovníkové metody s restrukturalizací slovníku . . . 87
6.6 Syntaktické metody . . . 89
6.6.1 Derivační kódování . . . 89
6.6.2 Analytické kódování . . . 90
6.7 Kontextové modelování . . . 91
6.7.1 Konečné kontextové modely . . . 91
6.7.2 Modely založené na konečných automatech . . . 92
6.7.2.1 Automaty s konečným kontextem . . . 92
6.7.2.2 Dynamické Markovovo modelování . . . 93

7 Kontrola správnosti textu **95**

7.1 Kontrola textu pomocí frekvenčního slovníku . . . 96
7.2 Kontrola textu pomocí dvojitého slovníku . . . 96
7.3 Interaktivní kontrola textu . . . 97
7.4 Kontrola textu založená na pravidelnosti slov . . . 98

Literatura **99**

1 Základní pojmy informačních systémů

Informační systém (informační soustava) je systém, který umožňuje účelné uspořádání sběru, uchování, zpracování a poskytování informací.

Informace je odraz poznaného nebo předpokládaného obsahu skutečností. Informaci lze definovat vždy jen ve vztahu k systému, pro který je určena. Informace je možno posuzovat z těchto tří hledisek:

1. Kvantitativního – jako měřitelnou veličinu, která číselně vyjadřuje množství informace obsažené v dané zprávě. Toto hledisko studuje teorie informace.

1. Kvalitativního – (významového, sématického) jako zprávu, která zmenšuje u příjemce neznalost jistých skutečností, tj. zvětšuje jeho informovanost.

2. Pragmatického – jako zprávu, která má pro příjemce určitou hodnotu. Při určení pragmatické (hodnotové) stránky informací vycházíme z těchto vlastností:

   a) význam pro řešení určitého problému,
   b) užitečnost – vyjadřuje podíl účasti na řešení problému,
   c) upotřebitelnost – vyjadřuje možnost jednorázového nebo vícenásobného použití,
   d) periodicita – vyjadřuje četnost použití,
   e) okruh použití pro řešení různých problémů,
   f) aktuálnost podle rychlosti stárnutí informace,
   g) hodnověrnost.

Kromě uvedených vlastností se dále u informace posuzuje pohotovost, podrobnost, úplnost, jednoznačnost, dostupnost a náklady na její získání.

Údaj (datum) je konkrétní vyjádření zprávy ve formě posloupnosti symbolů nějaké abecedy.

Z uvedených definic pojmů údaj a informace je zřejmé, že tyto pojmy nelze zaměňovat. Jistě si můžeme představit údaj, který nenese žádnou informaci.

Informačním procesem rozumíme proces vzniku informací, jejich zobrazení ve formě údajů, uchování, zpracování, poskytování a využití. Tomuto procesu odpovídají určité operace nad informacemi.

2 Klasifikace informačních systémů

Automatizované informační systémy můžeme klasifikovat podle převládající funkce, kterou plní. Nejdůležitější typy informačních systémů jsou:

1. databázové systémy (data base management systems),
2. dokumentografické vyhledávací systémy (information retrieval systems),
3. faktografické systémy pro řízení (management information systems),
4. systémy pro podporu rozhodování (decision support systems),
5. expertní systémy (expert systems, question-answering systems).

Databázové systémy

Jakýkoliv automatizovaný informační systém je založen na databázi, tj. souboru položek uložených v nějaké paměti, ke kterým má přístup. V databázových systémech jsou jednotlivé položky uloženy jako záznamy, které se skládají ze složek. Každá složka obsahuje hodnotu určitého typu. Při vyhledávání informací je nutno stanovit specifické hodnoty složek záznamů, které je třeba vyhledat. Výsledkem vyhledávání je množina těch záznamů, jejichž příslušné složky mají hodnoty, které přesně odpovídají zadanému požadavku. Dá se tedy říci, že informace uložená v databázi databázového systému je přísně strukturovaná a vyhledávání je na této struktuře založeno.

Dokumentografické vyhledávácí systémy

V dokumentografickém vyhledávácím systému jsou jednotlivé položky uloženy ve formě záznamů (dokumentů) obsahujících textové informace obvykle v přirozeném jazyce. Při vyhledávání informací je obvykle zadán vyhledávací výraz obsahující slova, slovní spojení nebo části slov. Výsledkem vyhledávání je množina záznamů, které přibližně odpovídají vyhledávacímu výrazu. Na rozdíl od databázových systémů, je informace uložena v databázi vyhledávacího systému nestrukturovaná nebo jen málo strukturovaná a při vyhledávání se není tedy možno opřít o strukturu záznamů.

Faktografické systémy pro řízení

Informační systém pro řízení je databázový systém přizpůsobený potřebám řídícího pracovníka (manažera). Je to systém, který navíc má některé funkce, které nejsou běžné u databázových systémů. Jedná se např. o možnost provádění procesů s informacemi vybranými z databáze.

Systémy pro podporu rozhodování

Systém pro podporu rozhodování je informační systém, ve kterém jsou integrovány funkce databázového systému, vyhledávacího systému a systému pro řízení. Obvykle je v tomto systému možno použít grafiky a prezentovat výsledky výběru informací nebo různých výpočtů v grafické podobě.

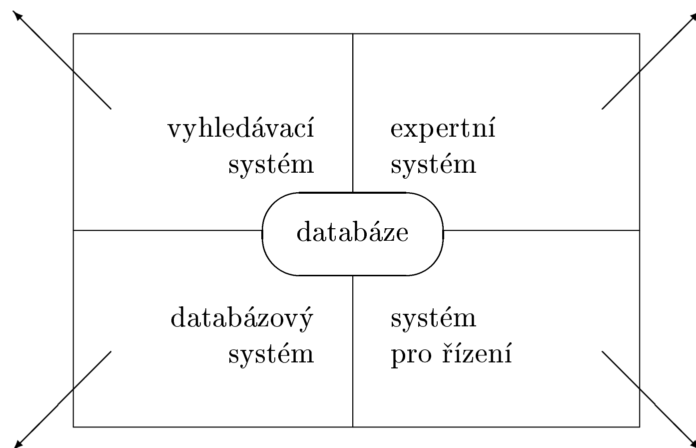
Expertní systémy

Expertní systém se skládá z databáze a odvozovacího mechanismu. Uživatel zadává dotazy v jazyce, který se podobá přirozenému jazyku. Systém dotazy analyzuje a na základě znalostí uložených v databázi vytváří odpověď.

Obr. 2.1 shrnuje vztahy mezi různými typy informačních systémů.

Vyhledává textové informace
Uchovává text v přirozeném jazyce
Zpracovává přibližné dotazy

Vyhledává specifické informace
Uchovává fakta z určité oblasti
a obecné znalosti
Zpracovává dotazy z dané oblasti



Vyhledává uložené informace
Uchovává strukturované informace
Zpracovává přesné dotazy

Přidává k databázovému
systému zpracování
vyhledané informace

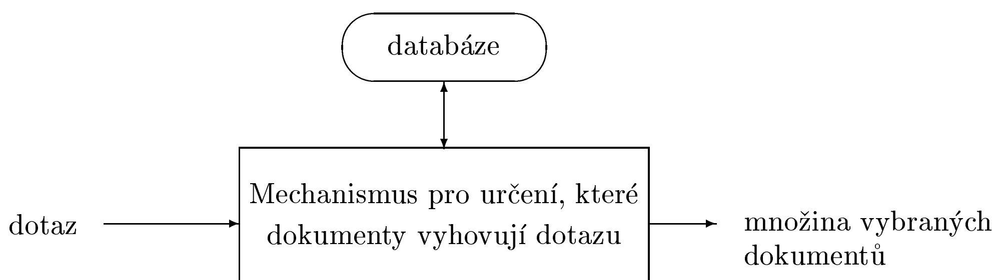
Obr. 2.1: Vztahy různých typů informačních systémů

Databázové systémy a systémy pro řízení mají mnoho společného. To se už nedá říci o ostatních typech informačních systémů. Databázové systémy a systémy pro řízení zpracovávají strukturované údaje často ve formě tabulek číselných údajů. Vyhledávací systémy a expertní systémy zpracovávají údaje v přirozeném jazyce. Vyhledávací systémy vyhledávají dokumenty a expertní systémy vyhledávají fakta potřebná k odpovědi na zadaný dotaz. Společným znakem všech typů automatizovaných informačních systémů je existence databáze pro uložení informací.

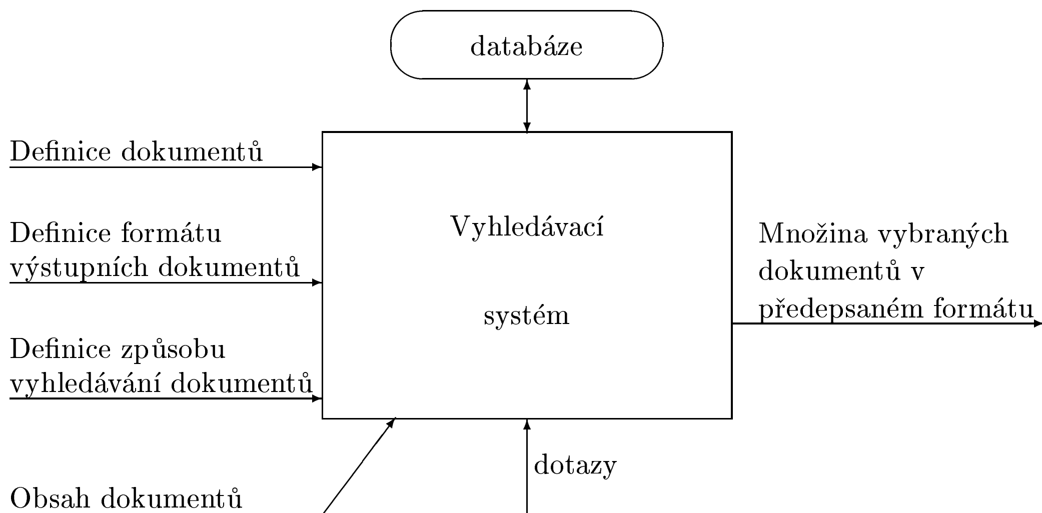
3 Vyhledávací systémy

Existuje mnoho různých typů vyhledávacích systémů. Abychom mohli posoudit výhody a nevýhody různých systémů, podíváme se na vyhledávací systém z hlediska jeho funkce. Databáze vyhledávacího systému obsahuje množinu záznamů (dokumentů). Uživatelé systému mohou zadávat požadavky (dotazy) na výběr dokumentů. Vyhledávací systém na základě dotazu určí, které dokumenty odpovídají zadanému dotazu. Tuto situaci znázorňuje obr. 3.1.

Některé vyhledávací systémy jsou dodávány s databází a umožňují jen vyhledávání dokumentů. Existují však tzv. prázdné vyhledávací systémy, které umožňují doplňování databáze s tím, že je možno definovat tvar a strukturu dokumentů, způsob přípravy vstupních dokumentů, způsob výběru dokumentů a způsob prezentace vybraných dokumentů. Princip takového systému je znázorněn na obr. 3.2.



Obr. 3.1: Princip vyhledávání dokumentu



Obr. 3.2: Vyhledávací systém s možností definice a doplňování databáze

V dalších kapitolách se budeme podrobněji zabývat metodami uložení dokumentů, možnostmi definice jejich struktury, algoritmy pro výběr dokumentů, možnostmi organizace databáze z hlediska rychlosti vyhledávání, jazyky pro zápis dotazů a metodami popisu formátu vystupujících dokumentů.

4 Vyhledávací algoritmy

Vyhledávání je operace, při které se zjišťuje, zda zadaný text obsahuje hledaná slova – vzorky. V případě, že zadaný text obsahuje hledané vzorky, zajímá nás i informace o tom, kde se v zadaném textu vzorky vyskytují.

Budeme předpokládat, že úloha na vyhledávání vzorků v textu je formulována jedním z těchto způsobů:

a) Vyhledej vzorek v v textu.
Výsledkem je informace o tom, kde se v textu vzorek v vyskytuje.

b) Vyhledej konečnou množinu vzorků $p = \{v_1, v_2, \ldots, v_k\}$.
Výsledkem je informace o tom, kde se v textu vyskytuje některý ze zadaných vzorků.

c) Vyhledej nekonečnou množinu vzorků zadanou regulárním výrazem R. Regulární výraz R definuje množinu $L(R)$, která může být nekonečná.
Výsledkem je informace o tom, kde se v textu vyskytuje některý ze vzorků z množiny $L(R)$.

Ve všech případech můžeme úlohu na vyhledávání vzorků formulovat tak, že vyhledáváme buď první výskyt nebo všechny výskyty nějakého vzorku. Ve druhém případě nás zajímá i případ, kdy se vyhledávané vzorky překrývají. Například při vyhledávání vzorku *abab* v textu *aaababababa* můžeme zjistit, že vzorek se objevuje dvakrát a to od pozice třetí a páté.

Metody vyhledávání můžeme rozdělit do dvou skupin:

a) Do první skupiny patří postupy, při kterých se prohlíží samotný text.

b) Do druhé skupiny patří postupy, při kterých je nejdříve prohlédnuta předem připravená „náhražka" textu, která charakterizuje původní text. Taková náhražka může být

   – index, tj. uspořádaný seznam významných prvků s odkazy do původního textu,
   – signatura, tj. řetězec příznaků indikující přítomnost významných prvků v textu.

Co do kvality hledání je nejlepší metoda, která prohlíží samotný text. Tento přímý přístup může však být značně časově náročný. Metoda používající „náhražky" umožňuje hrubé vyhledání rychleji, avšak výsledek nemusí být kvalitní. Oba přístupy lze spolu kombinovat při dvoustupňovém vyhledávání. Nejdříve se použije rychlá metoda náhražek, čímž se vyloučí většina textu, ve které se zadané vzorky vůbec nevyskytují. Na zbylé menšině textu se pak použije přímé prohledání, které vede k přesným výsledkům. Metoda náhražek vyžaduje předchozí přípravu textu, která může být dosti nákladná. Aby se tato investice vyplatila, musí být vyhledávaných vzorků dostatečný počet. Na druhé straně některé metody vyžadují předzpracování hledaných vzorků.

Metody vyhledávání můžeme klasifikovat podle toho, zda vyžadují předzpracování textů nebo předzpracování vzorků nebo obojí, do čtyř kategorií podle tabulky 4.1.

Do skupiny **I** patří *elementární algoritmus*, který nevyžaduje ani předzpracování textu ani předzpracování vzorku.

Metoda vyžaduje		Předzpracování textu	
		ne	ano
Předzpracování vzorků	ne	I	III
	ano	II	IV

Tabulka 4.1: Klasifikace vyhledávacích metod

Do skupiny **II** patří metody, které pro daný vzorek nebo množinu vzorků vytvoří nejdříve *vyhledávací stroj*, který potom provádí vyhledávání.

Do skupiny **III** patří *indexové metody*, které pro text, ve kterém se má vyhledávat, vytvoří index. Index je uspořádaný seznam slov s odkazy na jejich umístění v textu.

Do skupiny **IV** patří *signaturové metody*, které vytvoří pro vzorek nebo množinu vzorků a prohledávaný text řetězce bitů (signatury), který jak vzorky tak text charakterizují, a vyhledávání se provádí porovnáváním signatur.

Druhým kritériem pro klasifikaci vyhledávacích metod je odpověď na otázku, zda metoda umožňuje vyhledání pouze jediného vzorku nebo současné vyhledávání konečné nebo nekonečné množiny vzorků.

Třetím kriteriem pro klasifikaci vyhledávacích metod je směr porovnávání vzorků a směr průchodu textem. Text se obvykle prohledává směrem zleva doprava. Sousměrné metody porovnávají vzorky také ve směru zleva doprava. Protisměrné metody provádějí porovnání vzorků v opačném směru, t.j. zprava doleva.

4.1 Elementární algoritmus

Elementární algoritmus postupně přikládá a porovnává jediný vzorek ve všech pozicích textu. Prohledávaný text a zadaný vzorek se nijak nepředzpracovává. Princip tohoto algoritmu můžeme znázornit fragmentem programu v Pascalu takto:

```
var TEXT : array[1..T] of char;
    VZOREK : array[1..V] of char;
    I,J :  integer;
    NALEZEN, SHODA : boolean;
    ...

    NALEZEN := false;
    I := 0;
    while not NALEZEN and (I <= T - V) do
      begin
        I := I + 1
        J := 0;
        SHODA := true;
        while SHODA and (J < V) do
          if TEXT[I+J] = VZOREK[J+1] then J:=J + 1
          else SHODA := false;
        NALEZEN:= J = V;
      end
    ...
```

Poznámka: Proměnná I ukazuje v případě nalezení vzorku na první znak prvního výskytu vzorku v textu.

Budeme-li považovat za míru časové složitosti algoritmu počet porovnání jednotlivých znaků vzorku a textu (viz výraz $TEXT[I+J] = VZOREK[J+1]$), pak počet těchto porovnání je přinejhorším $V*(T-V+1)$, tedy asymptoticky lepší než $\mathcal{O}(V*T)$. Tento případ nastane například, když vzorek má tvar $a^{V-1}b$ a text má tvar $a^{T-1}b$. U přirozených jazyků však dochází k neshodě znaků textu a vzorku většinou velmi brzy (u prvního či druhého porovnávaného znaku). Počet porovnání se tedy sníží na $C_E*(T-V+1)$, kde C_E je empiricky zjištěná konstanta. Pro angličtinu má hodnotu 1.07, pro češtinu zjištěna není. Je tedy praktická asymptotická složitost elementárního algoritmu rovna $\mathcal{O}(C_E*T)$, tj. algoritmus se chová prakticky jako lineární. Implementace elementárního algoritmu je velmi jednoduchá.

Elementární algoritmus umožňuje sousměrné vyhledávání jednoho vzorku. V případě konečné množiny vzorků je nutno jej použít opakovaně pro každý vzorek. Pro vyhledávání nekonečné množiny vzorků je nepoužitelný.

4.2 Vyhledávací metody s předzpracováním vzorků

V tomto odstavci se budeme zabývat metodami vyhledávání, které provádí předzpracování vzorků před vlastním vyhledáváním. Výsledkem tohoto předzpracování vzorků je vyhledávací stroj, který je potom použit pro vyhledávání. Popisované metody rozdělíme na metody pro sousměrné vyhledávání a protisměrné vyhledávání.

4.2.1 Vyhledávací metody s předzpracováním vzorků se sousměrným vyhledáváním

Do skupiny metod, které vytváří vyhledávací stroj, který provádí sousměrné vyhledávání patří:

1. Knuth-Morris-Prattův algoritmus pro vyhledávání jednoho vzorku,
2. algoritmus Aho-Corasickové pro vyhledávání konečné množiny vzorků,
3. konečný automat pro vyhledávání nekonečné množiny vzorků.

4.2.1.1 Vyhledávací stroj

Vyhledávací stroj pro sousměrné vyhledávání je definován takto:

Vyhledávací stroj $A = (Q, T, g, h, q_0, F)$, kde

- Q je konečná množina stavů,
- T je konečná vstupní abeceda,
- g $Q \times T \rightarrow Q \cup \{fail\}$ je dopředná přechodová funkce,
- h $(Q - q_0) \rightarrow Q$ je zpětná přechodová funkce,
- q_0 je počáteční stav,
- F je množina koncových stavů.

V dalších odstavcích uvedeme konstrukci vyhledávacích strojů pro jednotlivé metody. Konfigurace vyhledávacího stroje je dvojice (q, w), kde $q \in Q$ je stav, ve kterém se stroj nachází a $w \in T^*$ je dosud neprohledaná část textu. Počáteční konfigurace je dvojice (q_0, w), kde w je celý prohledávaný text. Konfigurace, ve které došlo k nalezení vzorku, je dvojice (q, w), kde

$q \in F$ je koncový stav a w neprohledaná část textu. Nalezený vzorek je bezprostředně před textem w.

Vyhledávací stroj pro sousměrné vyhledávání provádí dopředné a zpětné přechody. Přechod vyhledávacího stroje je relace $\vdash \subset (Q \times T)^* \times (Q \times T)^*$ definovaná takto:

a) je-li $g(q, a) = p$, pak $(q, aw) \vdash (p, w)$ je dopředný přechod pro všechna $w \in T^*$,

b) je-li $h(q) = p$, pak $(q, w) \vdash (p, w)$ je zpětný přechod pro všechna $w \in T^*$.

Při dopředném přechodu $(q, aw) \vdash (p, w)$ pro $g(q, a) = p$ je přečten jeden vstupní symbol a stroj přechází do následujícího stavu p.

Je-li $g(q, a) = fail$, provede se zpětný přechod s použitím zpětné přechodové funkce h. Pro $g(q, a) = fail$ a $h(q) = p$, pak $(q, w) \vdash (p, w)$. Při tomto přechodu se nečte žádný vstupní symbol.

Funkce g a h mají tyto vlastnosti:

1. $g(q_0, a) \neq fail$ pro všechna a v T.
2. Jestliže $h(q) = p$, pak hloubka p je menší než hloubka q, kde hloubkou stavu q je míněna délka nejkratší dopředné posloupnosti přechodů ze stavu q_0 do stavu q.

První podmínka zajišťuje, že se neprovádí žádný zpětný přechod v počátečním stavu. Druhá podmínka zajišťuje, že celkový počet zpětných přechodů při zpracování vstupního řetězce bude menší než celkový počet dopředných přechodů. Protože pro každý vstupní symbol se provádí jeden dopředný přechod, bude celkový počet přechodů menší než dvojnásobek délky textu. Vyhledávání má tedy složitost $\mathcal{O}(T)$.

4.2.1.2 Knuth-Morris-Prattův algoritmus

Knuth, Morris a Pratt navrhli vyhledávací algoritmus, který má asymptotickou časovou složitost $\mathcal{O}(T + V)$. Tento algoritmus konstruuje nejdříve pro zadaný vzorek zpětnou přechodovou funkci h, což znamená předzpracování vzorku, a potom použije následující postup pro vyhledání vzorku v textu:

```
var TEXT:array[1..T] of char;
    VZOREK:array[1..V] of char;
    I,J :  integer;
    NALEZEN : boolean;
    ...


    I:=1; J:=1;
    while (I <= T) and (J <= V) do
      begin
        while (TEXT[I] <> VZOREK[J]) and (J > 0) do J:=h[J];
        J := J + 1;
        I := I + 1
      end;
    NALEZEN := J > V;
    ...
```

Vyhledávací algoritmus porovnává postupně znaky vzorku se znaky textu. Jeho základní myšlenka je založena na pozorování, že v okamžiku, kdy je porovnána předpona vzorku

$v_1 v_2 \dots v_{j-1}$ s podřetězcem textu $t_{i-j+1} t_{i-j+2} \dots t_{i-1}$ a $v_j <> t_i$, není třeba znovu porovnávat znaky $t_{i-j+1}, t_{i-j+2}, \dots, t_{i-1}$, protože je známo, jaká část porovnávaného textu je rovna předponě vzorku. Místo toho se ve vnitřním cyklu posouvá vzorek o $j - h(j)$ pozic doprava a do j se ukládá $h(j)$ tak dlouho, dokud j není nula (v tomto případě předpona vzorku není v prohledávaném textu obsažena) nebo $t_i = v_j$ (v tomto případě $v_1 v_2 \dots v_{j-1} v_j$ souhlasí s podřetězcem textu $t_{i-j+1} t_{i-j+2} \dots t_{i-1} t_i$). Aby mohl uvedený algoritmus správně pracovat, musí $h(j)$ mít hodnotu rovnou největšímu $k < j$ takovou, že $v_1 v_2 \dots v_{k-1}$ je příponou $v_1 v_2 \dots v_{j-1}$ (tj. $v_1 v_2 \dots v_{k-1} = v_{j-k+1} v_{j-k+2} \dots v_{j-1}$) a $v_j <> v_k$. Jestliže takové k neexistuje, pak $h(j) = 0$. Hodnoty funkce h počítá následující algoritmus, který se nápadně podobá výše uvedenému vyhledávacímu algoritmu.

```
I := 1; J := 0; h[1] := 0;
while I < V do
     begin
       while (J>0) and (VZOREK[I]<>VZOREK[J]) do J := h[J];
       I := I+1;
       J := J+1;
       if VZOREK[I]=VZOREK[J] then h[I] := J
                              else h[I] := h[J]
     end;
```

Funkce h je zpětná přechodová funkce vyhledávacího stroje pro Knuth-Morris-Prattův algoritmus (KMP vyhledávacího stroje). KMP vyhledávací stroj zkonstruujeme pomocí tohoto postupu:

Algoritmus 4.1:

Konstrukce KMP vyhledávacího stroje.
Vstup: Vzorek $v_1 v_2 \dots v_V$.
Výstup: KMP vyhledávací stroj.
Metoda:

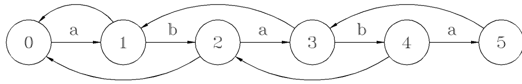
1. Počáteční stav bude q_0.
2. Každý stav q vyhledávacího stroje odpovídá předponě $v_1 v_2 \dots v_j$ zadaného vzorku. Definujme $g(q, v_{j+1}) = q'$, kde q' odpovídá předponě $v_1 v_2 \dots v_j v_{j+1}$.
3. Pro stav q_0 definujme $g(q_0, a) = q_0$ pro všechna a, pro která $g(q_0, a)$ nebylo definováno v kroku 2.
4. $g(q, a) = fail$ pro všechna q a a, pro která $g(q, a)$ nebylo definováno v kroku 2 nebo 3.
5. Stav, který odpovídá úplnému vzorku, je koncový stav.
6. Funkce h je zpětná přechodová funkce.

Příklad 4.1:

Konstrukci KMP vyhledávacího stroje provedeme pro vzorek *ababa* nad abecedou $T = \{a, b\}$. Dopředná přechodová funkce g a zpětná přechodová funkce h mají hodnoty podle následující tabulky:

g	**a**	**b**	**h**
0	1	0	nedefinováno
1	fail	2	0
2	3	fail	0
3	fail	4	1
4	5	fail	2
5			

Přechodový diagram KMP vyhledávacího stroje je na obr. 4.1. Dopředné přechody jsou označeny symboly, pro které se provádí, zpětné přechody označeny nejsou.



Obr. 4.1: KMP vyhledávací stroj pro vzorek *ababa*

Použijme KMP vyhledávací stroj pro vzorek *ababa* na vyhledávání v textu *abaababab*:

$$
\begin{array}{lllr l}
(0,\text{abaababab}) & \vdash & (1, & \text{baababab}) & \\
 & \vdash & (2, & \text{aababab}) & \\
 & \vdash & (3, & \text{ababab}) & \text{fail} \\
 & \vdash & (1, & \text{ababab}) & \text{fail} \\
 & \vdash & (0, & \text{ababab}) & \\
 & \vdash & (1, & \text{babab}) & \\
 & \vdash & (2, & \text{abab}) & \\
 & \vdash & (3, & \text{bab}) & \\
 & \vdash & (4, & \text{ab}) & \\
 & \vdash & (5, & \text{b}) & \text{vzorek nalezen}
\end{array}
$$

Postup zde uvedený je speciálním případem dále uvedeného AC vyhledávacího stroje.

4.2.1.3 Algoritmus Aho-Corasickové

Algoritmus Aho-Corasickové vede na AC vyhledávací stroj, který je schopen vyhledávat současně konečnou množinu vzorků. Je zadána množina vzorků $p = \{v_1, v_2, \ldots, v_k\}$. Na základě této množiny je sestrojen AC vyhledávací stroj a pak vyhledávání probíhá podle následujícího algoritmu:

```
var TEXT:array[1..T] of char;
    I :  integer;
    NALEZEN : boolean;
    STATE: TSTATE;
    g: array[1..MAXSTATE,1..MAXSYMBOL] of TSTATE;
    h: array[1..MAXSTATE] of TSTATE;
    F: set of TSTATE;
    ...
begin
     NALEZEN:=false;
     STATE:=q0;
     I:=0;
     while (I<=T) and not NALEZEN do
         begin
             I:=I+1;
             while g[STATE,TEXT[I]]=fail do STATE:=h[STATE];
```

```
            STATE:=g[STATE,TEXT[I]];
            NALEZEN:=STATE in F
        end;
end
```

Vyhledávací stroj postupně prochází text a v případě, že $g[STATE, TEXT[I]]$ má hodnotu $fail$, provádí zpětné přechody tak dlouho, dokud g má hodnotu $fail$. Potom provede dopředný přechod. Stroj končí vyhledávání v okamžiku, kdy dojde do některého koncového stavu.

Nyní uveďme postup konstrukce funkcí g a h. Dopředná přechodová funkce bude konstruována pomocí Algoritmu 4.2:

Algoritmus 4.2:

Konstrukce dopředné přechodové funkce AC vyhledávacího stroje.
Vstup: Množina vzorků $p = \{v_1, v_2, \ldots, v_k\}$.
Výstup: Dopředná přechodová funkce g AC vyhledávacího stroje.
Metoda:

1. Počáteční stav bude q_0.
2. Každý stav q vyhledávacího stroje odpovídá předponě $b_1 b_2 \cdots b_j$ nějakého vzorku v_i v množině p. Definujme $g(q, b_{j+1}) = q'$, kde q' odpovídá předponě $b_1 b_2 \cdots b_j b_{j+1}$ vzorku v_i.
3. Pro stav q_0 definujme $g(q_0, a) = q_0$ pro všechna a pro která $g(q_0, a)$ nebylo definováno v kroku 2.
4. $g(q, a) = fail$ pro všechna q a a, pro která $g(q, a)$ nebylo definováno v kroku 2 nebo 3.
5. Každý stav odpovídající úplnému vzorku bude koncový stav.

Dříve než budeme konstruovat zpětnou přechodovou funkci h, definujeme chybovou funkci f takto:

1. Pro všechny stavy q hloubky 1 bude $f(q) = q_0$.
2. Předpokládejme , že funkce f byla definována pro všechny stavy hloubky d a menší. Nechť q_d je stav hloubky d a $g(q_d, a) = q'$, pak $f(q')$ vypočítáme takto:

```
q:=f[qd];
while g[q,a]=fail do q:=f[q];
f[q']:=g[q,a];
```

Protože $g(q_0, a) \neq fail$, cyklus vždy skončí. Na obrázku 4.2 je schématicky znázorněn výpočet funkce $f(q')$. Jednotlivé hrany jsou ohodnoceny symboly, pro které se provádějí dopředné přechody.

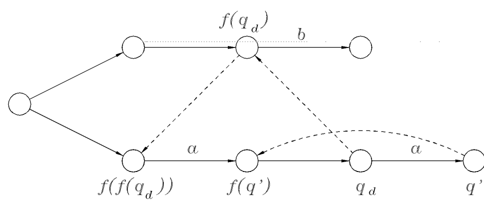
Chybová funkce f má tuto vlastnost:

Jestliže stavy q a r reprezentují předpony u a v nějakých vzorků z p, pak $f(q) = r$ právě tehdy, jestliže v je nejdelší vlastní předpona u.

Chybová funkce může být přímo použita jako zpětná přechodová funkce, avšak může způsobovat zbytečné zpětné přechody. Efektivnější zpětná přechodová funkce h vylučující tyto zbytečné zpětné přechody může být zkonstruována pomocí Algoritmu 4.3:

Algoritmus 4.3:

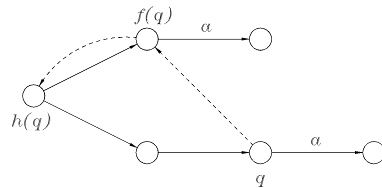
Konstrukce zpětné přechodové funkce AC vyhledávacího stroje.
Vstup: Dopředná přechodová funkce g AC vyhledávacího stroje.
Výstup: Zpětná přechodová funkce h AC vyhledávacího stroje.
Metoda:



Obr. 4.2: Výpočet $f(q')$

1. $h(q) = q_0$ pro všechny stavy hloubky jedna.
2. Předpokládejme, že funkce h byla definována pro všechny stavy hloubky d a menší. Nechť q je stav hloubky $d+1$. Jestliže množina znaků, pro které je ve stavu $f(q)$ hodnota funkce g jiná hodnota než $fail$, je podmnožinou množiny znaků, pro které je hodnota funkce g ve stavu q jiná než fail, pak $h(q) := h(f(q))$, jinak $h(q) := f(q)$.

Na obr. 4.3 je schematicky znázorněn výpočet funkce $h(q)$.

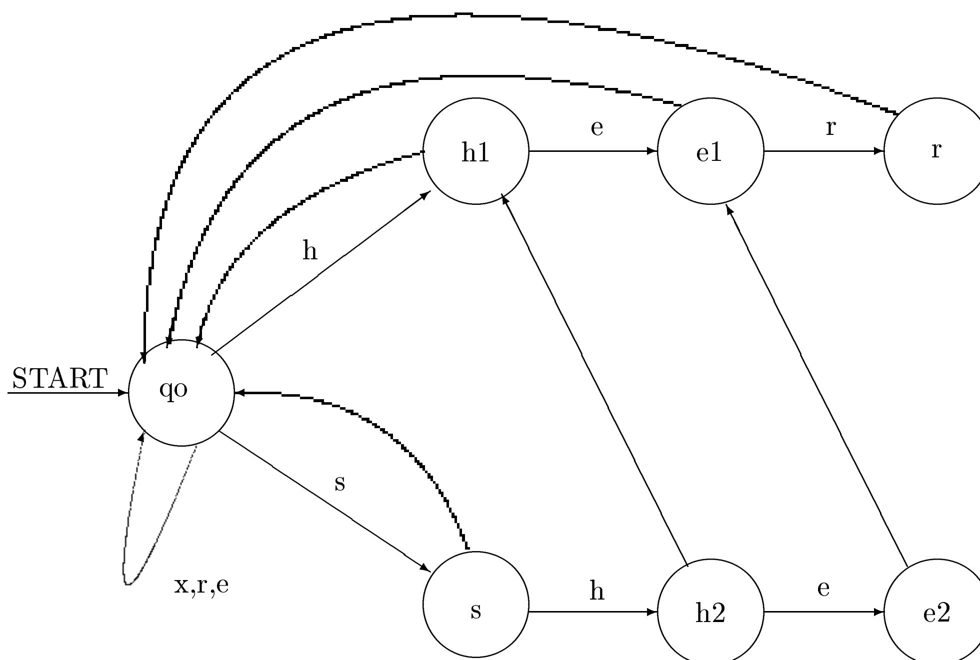


Obr. 4.3: Výpočet $h(q)$

Příklad 4.2:

Je zadána množina vzorků $p = \{he, she, her\}$ nad abecedou $T = \{h, e, r, s, x\}$. Symbol x zastupuje všechny znaky kromě h, e, r, s. Sestrojíme AC vyhledávací stroj. Funkce g, f a h jsou uvedeny v následující tabulce.

	g					f	h
	h	e	r	s	x		
q_0	h_1	q_0	q_0	s	q_0	–	–
h_1		e_1				q_0	q_0
e_1			r			q_0	q_0
r						q_0	q_0
s	h_2					q_0	q_0
h_2		e_2				h_1	q_0
e_2						e_1	e_1



Obr. 4.4: AC vyhledávací stroj pro $p = \{he, she, her\}$

Na obr. 4.4 je uveden přechodový diagram AC vyhledávacího stroje. Ohodnocené hrany patří dopředné přechodové funkci g. Neohodnocené hrany patří chybové funkci f.

Funkce h se liší od funkce f jen ve stavu h_2. Ve stavu h_2 je možný dopředný přechod jen pro symbol e stejně jako ve stavu h_1. To znamená, že po zpětném přechodu ze stavu h_2 do stavu h_1, který se provede jen pro jiné symboly než e, je nutno vždy provést další zpětný přechod do stavu q_0. Je proto možné ve stavu h_2 provést zpětný přechod přímo do stavu q_0.

4.2.1.4 Vyhledávací konečné automaty

V tomto odstavci uvedeme pojmy deterministický a nedeterministický konečný automat a ukážeme jejich použití při vyhledávání.

Deterministický konečný automat M je pětice $M = (K, T, \delta, q_0, F)$, kde

K je konečná množina vnitřních stavů,
T je konečná vstupní abeceda,
δ je zobrazení z $K \times T$ do K,
$q_0 \in K$ je počáteční stav,
$F \subset K$ je množina koncových stavů.

Deterministický konečný automat pracuje tak, že provádí posloupnost přechodů. Přechod je určen stavem, ve kterém se automat nachází, a symbolem, který je čten ze vstupního řetězce. Při přechodu přejde automat do nového stavu a přečte jeden vstupní symbol. Čtení vstupních symbolů se provádí zleva doprava počínaje nejlevějším symbolem vstupního řetězce. Pokud je zobrazení definováno pro každou dvojici (q, a) z $K \times T$, říkáme, že automat je úplně určený. V opačném případě se jedná o neúplně určený automat.

Abychom mohli určit budoucí chování deterministického konečného automatu, potřebujeme znát:

a) stav, ve kterém se automat právě nachází,

b) zbylou část vstupního řetězce, kterou dosud automat nepřečetl.

Tuto dvojici $(q, w) \in K \times T^*$ nazveme *konfigurací* konečného automatu M. Konfiguraci (q_0, w) nazveme počáteční konfigurací konečného automatu M, konfiguraci (q, ε), kde $q \in F$, nazveme koncovou konfigurací konečného automatu.

Relaci $\vdash \subset (K \times T^*) \times (K \times T^*)$ nazveme *přechodem* automatu M. Jestliže $\delta(q, a) = p$, pak $(q, aw) \vdash (p, w)$ pro všechna $w \in T^*$. Symbolem $\vdash k$ označíme k-tou mocninu relace $\vdash$. Symboly $\vdash +$ a $\vdash^*$ budou označovat tranzitivní a tranzitivně reflexivní uzávěr relace $\vdash$.

Řetězec w je přijat konečným deterministickým automatem $M = (K, T, \delta, q_0, F)$, jestliže $(q_0, w) \vdash^* (q, \varepsilon)$ pro nějaké $q \in F$. $L(M) = \{w : w \in T^*, (q_0, w) \ \vdash^* (q, \varepsilon)$ pro nějaké $q \in F\}$ je jazyk přijímaný konečným automatem M. Řetěz $w \in L(M)$, jestliže se skládá pouze ze symbolů vstupní abecedy a existuje posloupnost přechodů taková, která z počáteční konfigurace (q_0, w) vede do koncové konfigurace (q, ε).

Dále uvedeme vyhledávací algoritmus založený na principu konečného automatu, který používá tabulku přechodů g, která odpovídá zobrazení δ.

```
var TEXT:array[1..T] of char;
    STATE: TSTATE;
    g:array[1..MAXSTATE,1..MAXSYMB] of TSTATE;
    I:  integer;
    NALEZEN : boolean;
    F: set of TSTATE;
...
begin
     NALEZEN:=FALSE;
     STATE:=q0;
     I:=0;
     while (I<=T) and not NALEZEN do
         begin
             I:=I+1;
             STATE:=g[STATE,TEXT[I]];
             NALEZEN:=STATE in F
         end
end;
...
```

Při konstrukci vyhledávacího konečného automatu je výhodné sestrojit nejdříve nedeterministický konečný automat a ten transformovat na deterministický.

Nedeterministický konečný automat M je pětice $M = (K, T, \delta, q_0, F)$, kde

K je konečná množina vnitřních stavů,
T je konečná vstupní abeceda,
δ je zobrazení z $K \times T$ do množiny podmnožin K,
$q_0 \in K$ je počáteční stav,
$F \subset K$ je množina koncových stavů.

Z porovnání definic plyne, že podstatný rodíl nedeterministického konečného automatu oproti deterministickému konečnému automatu spočívá v tom, že $\delta(q, a)$ je u nedeterministického automatu množina stavů, zatímco $\delta(q, a)$ je u deterministického automatu jeden stav.

Pojmy konfigurace, počáteční a koncové konfigurace podle definice můžeme použít i pro nedeterministický konečný automat. Definici přechodu musíme rozšířit.

Relaci $\vdash \subset (K \times T^*) \times (K \times T^*)$ nazveme přechodem v automatu M. Jestliže $p \in \delta(q, a)$, pak $(q, aw) \vdash (p, w)$ pro všechna $w \in T^*$.

Řetězec w je přijat konečným nedeterministickým automatem $M = (K, T, \delta, q_0, F)$, jestliže existuje posloupnost přechodů $(q_0, w) \vdash^* (q, \varepsilon)$ pro nějaké $q \in F$. Potom $L(M) = \{w : w \in T^*, (q_0, w) \vdash^* (q, \varepsilon) \text{ pro nějaké } q \in F\}$ je jazyk přijímaný nedeterministickým konečným automatem M.

Pro každý nedeterministický konečný automat M můžeme sestrojit deterministický konečný automat M', pro který platí, že $L(M) = L(M')$. Převod nedeterministického konečného automatu na ekvivalentní deterministický provedeme takto:

Algoritmus 4.4:

Vstup: Nedeterministický konečný automat $M = (K, T, \delta, q_0, F)$
Výstup: Deterministický konečný automat $M' = (K', T, \delta', q_0', F')$, pro který platí, že $L(M) = L(M')$.
Metoda:

1. Definujeme $K' = \{\{q_0\}\}$, stav $\{q_0\}$ budeme považovat za neoznačený.
2. Jestliže v K' jsou všechny stavy označeny, pokračuj krokem (4).
3. Vybereme z K' neoznačený stav q' a provedeme tyto operace:
   (a) Určíme $\delta'(q', a) = \bigcup \delta(p, a)$ pro všechna $p \in q'$ a pro všechna $a \in T$,
   (b) $K' = K' \cup \delta'(q', a)$ pro všechna $a \in T$,
   (c) stav $q' \in K'$ označíme,
   (d) pokračujeme krokem (2)
4. $q_0 = q_0'$.
5. $F' = \{q' : q' \in K', q' \cap F \neq \emptyset\}$.

Poznámka: Z důvodu přehlednosti a čitelnosti budeme pro označení stavů v K' používat hranatých místo složených závorek.

Nyní uvedeme postup konstrukce přechodové tabulky g pro zadanou množinu vzorků $p = \{v_1, v_2, \ldots, v_k\}$. Nejdříve sestrojíme přechodovou tabulku g' pro nedeterministický konečný automat M', který přijímá všechny řetězce s příponami z množiny p takto:

1. Počáteční stav bude q_0 a $g'(q_0, a) = q_0$ pro všechna a ze vstupní abecedy automatu.
2. Každý stav q automatu odpovídá předponě $b_1 b_2 \ldots b_j$ nějakého vzorku v_i v množině p. Definujeme $g'(q, b_{j+1}) = q'$, kde q' odpovídá předponě $b_1 b_2 \ldots b_j b_{j+1}$ vzorku v_i.
3. Každý stav, který odpovídá úplnému vzorku bude koncový stav.

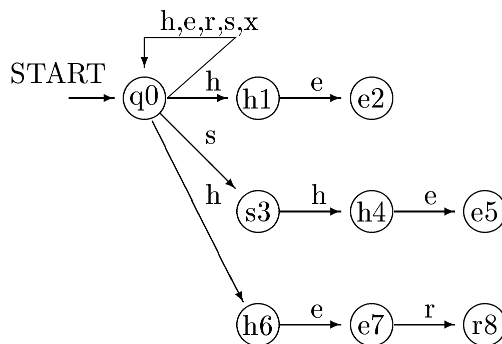
Dále sestrojíme deterministický automat M s přechodovou tabulkou g.

Příklad 4.3:

Celý postup návrhu automatu ukážeme na příkladě. Mějme zadánu úlohu takto: Nalezněte, zda v daném řetězci nad abecedou $T = \{h, e, r, s, x\}$ se vyskytují podřetězce z množiny $p = \{he, she, her\}$. Symbol x zastupuje všechna písmena kromě h, e, r, s. Celý problém budeme řešit takto:

1. Sestrojíme nedeterministický konečný automat M', který přijímá všechny řetězce nad abecedou $\{h, e, r, s, x\}$ s příponami z množiny p. Přechodový diagram automatu M' je na obr. 4.5. Počáteční stav je stav q_0, koncové stavy jsou stavy $e2, e5, e7$ a $r8$ a znamenají situace, kdy byly přečteny přípony he, he nebo she, he a her.

2. Získaný nedeterministický konečný automat M' převedeme na deterministický. Přechodová tabulka nedeterministického automatu má tvar:



Obr. 4.5: Nedeterministický automat M' pro vyhledávání množiny vzorků $p = \{he, she, her\}$

g'	h	e	s	r	x
q0	q0,h1,h6	q0	q0,s3	q0	q0
h1		e2			
e2					
s3	h4				
h4	e5				
e5					
h6		e7			
e7				r8	
r8					

Po převodu automatu M' na deterministický automat získáme automat M, jehož přechodová tabulka má tvar:

g	h	e	s	r	x
[q0]	[q0,h1,h6]	[q0]	[q0,s3]	[q0]	[q0]
[q0,h1,h6]	[q0,h1,h6]	[q0,e2,e7]	[q0,s3]	[q0]	[q0]
[q0,s3]	[q0,h1,h4,h6]	[q0]	[q0,s3]	[q0]	[q0]
[q0,e2,e7]	[q0,h1,h6]	[q0]	[q0,s3]	[q0,r8]	[q0]
[q0,h1,h4,h6]	[q0,h1,h6]	[q0,e2,e5,e7]	[q0,s3]	[q0]	[q0]
[q0,r8]	[q0,h1,h6]	[q0]	[q0,s3]	[q0]	[q0]
[q0,e2,e5,e7]	[q0,h1,h6]	[q0]	[q0,s3]	[q0,r8]	[q0]

3. Získaný deterministický automat M' implementujeme jako program. Vzhledem k tomu, že automat M' je úplně určený, což znamená, že zobrazení g' je definováno vždy, je vhodné jej implementovat tak, že tabulku přechodů budeme reprezentovat jako celočíselnou matici, kterou používá univerzální algoritmus.

Přítomnost hledaných řetězců se zjistí tak, že při přechodu do koncového stavu můžeme identifikovat konec jistého vzorku podle této tabulky:

hledaný vzorek	koncové stavy
he	[q0,e2,e7],[q0,e2,e5,e7]
she	[q0,e2,e5,e7]
her	[q0,r8]

4.2.1.5 Vyhledávání nekonečné množiny vzorků

Vhodným prostředkem pro popis nekonečné množiny vzorků je regulární výraz.

Definice:

Regulární výraz V nad abecedou A je definován takto:

1. $\emptyset, \varepsilon, a$ jsou regulární výrazy pro všechna $a \in A$.
2. Jsou-li x, y regulární výrazy nad A, pak:

a)	$(x + y)$	(sjednocení)
b)	$(x \;.\; y)$	(zřetězení)
c)	$(x)^*$	(iterace)

jsou regulární výrazy nad A.

Hodnota $h(x)$ regulárního výrazu x je definována takto:

1. $h(\emptyset) = \emptyset,\ h(\varepsilon) = \{\varepsilon\},\ h(a) = \{a\},$
2. $h(x + y) = h(x) \cup h(y),$
   $h(x \;.\; y) = h(x) \;.\; h(y),$
   $h(x^*) = (h(x))^*.$

Hodnotu $h(x^*)$ můžeme vyjádřit také takto:

$$h(x^*) = \varepsilon + x + x.x + x.x.x + \ldots,$$

což znamená, že $h(x^*)$ obsahuje prázdný řetěz a dále všechny řetězy, které vzniknou zřetězením libovolného počtu řetězů x.

Hodnotou libovolného regulárního výrazu je regulární jazyk. Naopak každý regulární jazyk lze reprezentovat nějakým regulárním výrazem.

Pro zápis regulárních výrazů se obvykle zavádí konvence o prioritě regulárních operací a pak je možno vynechávat přebytečné závorky. Nejvyšší prioritu má operace iterace, nejnižší pak operace sjednocení.

Pro regulární výrazy jsou definovány tyto axiomy (Salomaa 1966):

A1:	$x + (y + z) = (x + y) + z$	(asociativnost sjednocení)
A2:	$x.(y.z) = (x.y).z$	(asociativnost zřetězení)
A3:	$x + y = y + x$	(komutativnost sjednocení)
A4:	$(x + y).z = x.z + y.z$	(distributivnost zprava)
A5:	$x.(y + z) = x.y + x.z$	(distributivnost zleva)
A6:	$x + x = x$	(idenpotence sjednocení)
A7:	$\varepsilon \;.\; x = x$	(ε je jednotkový prvek pro zřetězení)
A8:	$\emptyset \;.\; x = \emptyset$	($\emptyset$ je nulový prvek pro zřetězení)
A9:	$x + \emptyset = x$	($\emptyset$ je nulový prvek pro sjednocení)
A10:	$x^* = \varepsilon + x^*x$	
A11:	$x^* = (\varepsilon + x)^*$	

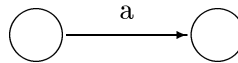
Je dokázáno, že všechny ostatní rovnosti mezi regulárními výrazy lze odvodit z těchto axiomů.

Pro každý regulární výraz V je možno sestrojit konečný automat M takový, že $h(V) = L(M)$.

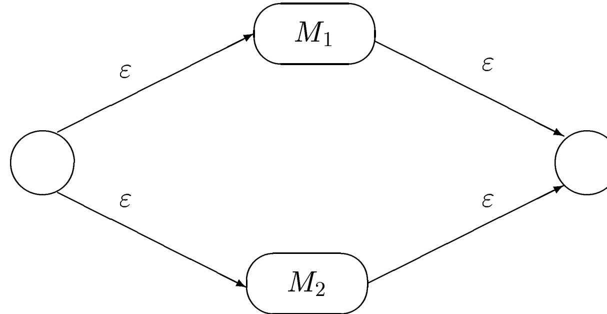
Nejdříve uvedeme klasický způsob konstrukce zobecněného nedeterministického konečného automatu pro daný regulární výraz. Zobecněný konečný automat dovoluje ε-přechody, tj. přechody bez čtení vstupního symbolu.

Následující rekurzivní procedura sestrojí pro daný regulární výraz V automat M.

1. Jestliže výraz V je tvořen jediným symbolem a, sestrojíme automat mající dva stavy a přechod z počátečního do koncového stavu pro symbol a.



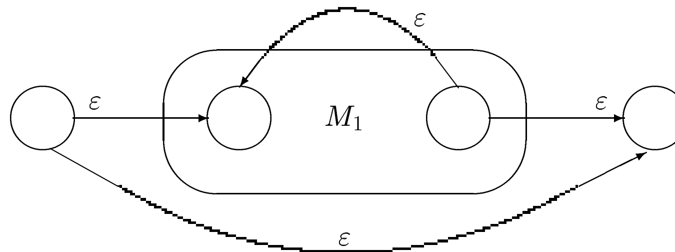
2. Jestliže výraz $V = V_1 + V_2$, M_1 a M_2 jsou automaty pro výrazy V_1 a V_2, pak sestrojíme pro výraz V automat:



Vytvořili jsme nový počáteční a nový koncový stav. Z počátečního stavu vedou ε-přechody do počátečních stavů automatů M_1 a M_2. Dále jsou zde ε-přechody z koncových stavů automatů M_1 a M_2 do nově vytvořeného koncového stavu.

3. Jestliže výraz $V = V_1 \cdot V_2$, M_1 a M_2 jsou automaty pro výrazy V_1 a V_2, pak sestrojíme pro výraz V automat M tak, že ztotožníme koncový stav automatu M_1 s počátečním stavem automatu M_2. Počáteční stav automatu M_1, je počátečním stavem automatu M a koncový stav automatu M_2 je koncovým stavem automatu M.

4. Jestliže výraz $V = V_1^*$ a M_1 je automat pro výraz V_1, pak sestrojíme automat:



Vytvořili jsme nový počáteční a nový koncový stav. V automatu M jsou čtyři nové ε-přechody:

- z vytvořeného počátečního stavu do počátečního stavu automatu M_1,
- z vytvořeného počátečního stavu do vytvořeného koncového stavu,
- z koncového stavu automatu M_1 do jeho počátečního stavu,
- z koncového stavu automatu M_1 do nově vytvořeného koncového stavu.

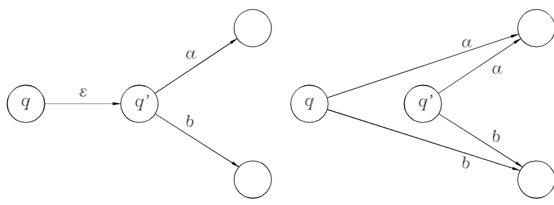
Takto vytvořený nedeterministický konečný automat má tyto zajímavé a užitečné vlastnosti:

a) počet stavů automatu M je nejvýše roven dvojnásobku délky výrazu V,

b) ze žádného stavu nevychází více jak dvě hrany do dalších stavů,

c) z koncového stavu nevychází žádná hrana.

Délka d regulárního výrazu V je definována takto:



Obr. 4.6: Odstranění ε-přechodů

$d(V) = 1$, jestliže V je tvořen jediným symbolem,

$d(V_1 + V_2) = d(V_1) + d(V_2) + 1$,

$d(V_1 \cdot V_2) = d(V_1) + d(V_2) + 1$,

$d(V^*) = d(V) + 1$,

$d((V)) = d(V) + 2$.

Tyto vlastnosti umožňují efektivně simulovat chování automatu. Nechť d je délka výrazu V a T délka textu. Při simulaci se používá množina stavů automatu M, do které automat může přejít po přečtení řetězce $a_1 a_2 \ldots a_{i-1}$. Z této množiny můžeme určit v čase $\mathcal{O}(d)$ množinu stavů, do kterých se automat může dostat po přečtení symbolu a_i, protože z každého stavu jsou možné nejvýše dva přechody. Tímto způsobem je možno simulovat chování automatu M v čase $\mathcal{O}(dT)$ a v paměti $\mathcal{O}(d)$. Proto se nyní budeme zabývat metodami implementace nedeterministických konečných automatů. Uvedeme dvě metody. První metoda je založena na použití tabulky přechodů. Druhá metoda se opírá o implementaci automatu ve formě programu.

V obou případech jsou stavy automatu reprezentovány stavovým vektorem. Každému stavu nedeterministického konečného automatu odpovídá jeden prvek tohoto vektoru. Jestliže se automat nachází v určitém stavu, bude mít odpovídající prvek stavového vektoru hodnotu $true$, jinak bude mít hodnotu $false$. Nedeterministický konečný automat budeme implementovat tak, že současně budeme procházet všemi možnými cestami. To znamená, že ve stavovém vektoru může mít více prvků hodnotu true, což odpovídá tomu, že automat se může nacházet ve více stavech.

Pro následující metody simulace nedeterministických konečných automatů je třeba odstranit ε-přechody. Tuto transformaci je možno provést tímto postupem:

1. Jestliže ε-přechod $q' \in \delta(q, \varepsilon)$ vede do stavu, ze kterého existují další přechody, pak tento ε-přechod nahradíme novými přechody tak, aby $\delta(q, a) = \delta(q', a)$ pro všechna $a \in T$. Tato transformace je znázorněna na obr. 4.6.

2. Jestliže ε-přechod vede ze stavu q do koncového stavu q_f, do kterého žádný další přechod nevede, vynecháme jej a q bude další koncový stav.

Nejdříve uvedeme program simulující nedeterministický konečný automat tak, že stav je reprezentován Booleovským vektorem.

```
program   NKA(INPUT,OUTPUT);
type      TYPSTAV = (Q0,Q1,...,QN);
          TYPSYMBOL = 1 .. |T|;

var   SYMBOL:TYPSYMBOL;
      STAV,NSTAV: array [Q0..QN] of BOOLEAN;
      TABULKAPŘECHODŮ: array [Q0..QN,TYPSYMBOL] of set of TYPSTAV;
      PŘIJAT, DEFINOVÁN: BOOLEAN;
      PŘECHODY, F : set of TYPSTAV;
      I,J: TYPSTAV;

procedure DALŠÍSYMBOL(var X:TYPSYMBOL); external;
procedure ČTENÍTABULKYPŘECHODŮ; external;

begin
     ČTENÍTABULKYPŘECHODŮ;
     F := [QF1,QF2,...,QFK];
     STAV[Q0] := TRUE;
     for I := Q1 to QN do STAV[I] := FALSE;
     DEFINOVÁN :=TRUE;
while not EOF(INPUT) and DEFINOVÁN do
begin
     DALŠÍSYMBOL(SYMBOL);
     for I := Q0 to QN do NSTAV[I] := FALSE;
     for I := Q0 to QN do
       if STAV[I] then
         begin
            PŘECHODY := TABULKAPŘECHODŮ[I,SYMBOL];
            for J := Q0 to QN do
              if J in PŘECHODY then NSTAV[J] := TRUE;
         end;
     DEFINOVÁN := FALSE;
     for I := Q0 to QN do DEFINOVÁN := DEFINOVÁN or NSTAV[I];
     STAV := NSTAV;
end;
PŘIJAT := FALSE;
for I := Q0 to QN do PŘIJAT := PŘIJAT or (I in F);
if PŘIJAT then  WRITE(OUTPUT,'VSTUP PŘIJAT')
          else  WRITE(OUTPUT,'VSTUP NEPŘIJAT');
end.
```

V uvedeném programu jsou použity dva stavové vektory STAV a NSTAV. Důvodem k tomu je skutečnost, že při výpočtu nové hodnoty stavového vektoru je po celou dobu nutno uchovat hodnotu starého stavového vektoru. Činnost automatu může skončit ze dvou důvodů. Ukončení vstupního řetězu je důvod k normálnímu ukončení práce automatu. Ukončení při nedefinovaném stavu znamená, že automat neumožňuje přechod do žádného stavu, t.j. stavový vektor má všechny hodnoty $false$.

Uvedený program představuje univerzální implementaci pro libovolný konečný automat.

Druhá metoda implementace nedeterministického konečného automatu, při které je automat implementován jako program bez použití tabulky přechodů vede samozřejmě na různé programy pro různé automaty. Proto uvedeme implementaci následujícího automatu:

NKA= $(\{1,2,3,4,5\},\{a,b\},\delta,1,\{1\})$, kde zobrazení δ je definováno tabulkou:

δ	a	b
1	{2}	$\emptyset$
2	{3}	{1,2}
3	{4}	{1,3}
4	{5}	{1,4}
5	{1}	{1,5}

```
program  NKA(INPUT,OUTPUT);
type  TYPSTAV = (JEDNA,DVĚ,TŘI,ČTYŘI,PĚT);
      TYPSYMBOL = (A,B);

var   SYMBOL: TYPSYMBOL;
      STAV, NSTAV: array TYPSTAV of BOOLEAN;
      I: TYPSTAV;
      DEFINOVÁN: BOOLEAN;

procedure DALŠÍSYMBOL(var X: TYPSYMBOL); external;

begin
     STAV[JEDNA] := TRUE;
     for I := DVĚ to PĚT do STAV[I] := FALSE;
     DEFINOVÁN := TRUE;
while not EOF(INPUT) and DEFINOVÁN do
begin
    DALŠÍSYMBOL(SYMBOL);
    for I := JEDNA to PĚT do NSTAV := FALSE;

if STAV[JEDNA] then
        case SYMBOL of
            A : NSTAV[DVĚ] := TRUE;
            B :
        end;

if STAV[DVĚ] then
        case SYMBOL of
            A : NSTAV[TŘI] := TRUE;
            B : begin
                  NSTAV[JEDNA] := TRUE;
                  NSTAV[DVĚ]  := TRUE;                                  (*)
                end;
        end;

if STAV[TŘI] then
        case SYMBOL of
            A : NSTAV[ČTYŘI] := TRUE;
            B : begin
                  NSTAV[JEDNA] := TRUE;
                  NSTAV[TŘI] := TRUE                                    (*)
                end;
        end;
```

```
if STAV[ČTYŘI] then
        case SYMBOL of
            A : NSTAV[PĚT] := TRUE;
            B : begin
                  NSTAV[JEDNA] := TRUE;
                  NSTAV[ČTYŘI] := TRUE;                                (*)
                end;
        end;

if STAV[PĚT] then
        case SYMBOL of
            A : NSTAV[JEDNA] := TRUE;
            B : begin
                  NSTAV[JEDNA] := TRUE;
                  NSTAV[PĚT] := TRUE                                   (*)
                end;
        end;

DEFINOVÁN := NSTAV[JEDNA] or NSTAV[DVĚ] or NSTAV[TŘI] or NSTAV[ČTYŘI]
              or NSTAV[PĚT];

STAV := NSTAV
end;

if STAV[JEDNA] then WRITE (OUTPUT,'ŘETĚZ PŘIJAT')
               else WRITE (OUTPUT,'ŘETĚZ NEPŘIJAT')
end.
```

Poznámka: Příkazy na řádcích označených $(*)$ je možno vynechat.

Jiný klasický způsob vyhledávání vzorku generovaného regulárním výrazem je použití deterministického konečného automatu. Pro regulární výraz je možno sestrojit deterministický konečný automat několika způsoby. Jeden z nich je sestrojení deterministického automatu ekvivalentního nedeterministickému, jehož konstrukce byla popsána výše. Jednodušší a přímá cesta spočívá v přímém sestrojení deterministického konečného automatu. Uveďme dva způsoby.

Nejdříve uvedeme postup, který je založen na pojmu sousedních symbolů.

Algoritmus 4.5:

Konstrukce ekvivalentního konečného automatu pro daný regulární výraz.
Vstup: Regulární výraz V.
Výstup: Konečný automat $M = (K, T, \delta, q_0, F)$ takový, že $h(V) = L(M)$.
Metoda: Nechť abeceda, nad kterou je definován výraz V je T.

1. Očíslujeme čísly 1,2,…,n všechny výskyty symbolů z T ve výrazu V tak, aby každé dva výskyty téhož symbolu byly očíslovány různými čísly. Vzniklý regulární výraz označíme V'.
2. Sestrojíme množinu začátečních symbolů
   $Z = \{x_i : x \in T$, symbolem x_i může začínat nějaký řetězec z $h(V')\}$.
3. Sestrojíme množinu sousedů P takto:
   $P = \{x_i y_j$: symboly x_i a y_j mohou být vedle sebe v nějakém řetězci z $h(V')\}$.

4. Sestrojíme množinu koncových symbolů F takto:
   $F = \{x_i$: symbolem x_i může končit nějaké slovo z $h(V)\}$.
5. Množina stavů konečného automatu $K = \{q_0\} \cup \{x_i : x \in T,\ i \in \langle 1, n\rangle\}$.
6. Zobrazení δ sestrojíme takto:

   a) $\delta(q_0, x)$ obsahuje x_i pro všechna $x_i \in Z$ vzniklá očíslováním x.

   b) $\delta(x_i, y)$ obsahuje y_j pro všechny dvojice $x_i y_j \in P$ takové, že y_j vzniklo očíslováním y.

7. Množina F je množinou koncových stavů.

Příklad 4.4:

Sestrojíme konečný automat pro výraz $V = ab^*a + ac + b^*ab^*$.

$T = \{a, b, c\}$

Po očíslování symbolů má výraz V tvar:

$V = a_1 b_2^* a_3 + a_4 c_5 + b_6^* a_7 b_8^*$

Množina začátečních symbolů:

$Z = \{a_1, a_4, b_6, a_7\}$

Množina sousedů:

$P = \{a_1 b_2, a_1 a_3, b_2 b_2, b_2 a_3, a_4 c_5, b_6 b_6, b_6 a_7, a_7 b_8, b_8 b_8\}$

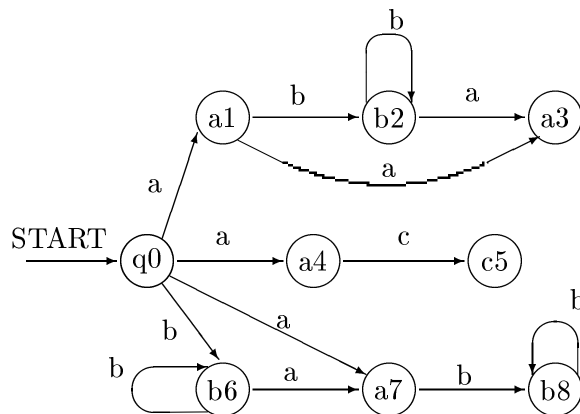
Množina koncových symbolů:

$F = \{a_3, c_5, a_7, b_8\}$

Konečný automat

$M = (\{q_0, a_1, b_2, a_3, a_4, c_5, b_6, a_7, b_8\},\ \{a, b, c\},\ \delta,\ q_0,\ \{a_3, c_5, a_7, b_8\})$,

a zobrazení δ je znázorněno přechodovým diagramem na obr. 4.7.
Z uvedeného příkladu je vidět, že výsledný automat může být nedeterministický.



Obr. 4.7: Přechodový diagram konečného automatu z příkladu 4.4

Další postup, který umožňuje vždy sestrojení deterministického konečného automatu pro zadaný regulární výraz, je založen na pojmu derivace regulárních výrazů. Pro regulární výrazy je definován pojem derivace regulárního výrazu takto:

Derivace $\frac{dV}{dx}$ regulárního výrazu V podle řetězce $x \in T^*$ je definována takto:

1. $\frac{dV}{d\varepsilon} = V$.

2. Pro $a \in T$ platí:

$$\frac{d\varepsilon}{da} = \emptyset, \quad \frac{d\emptyset}{da} = \emptyset$$

$$\frac{db}{da} = \begin{cases} \emptyset & \text{jestliže } a \neq b \\ \varepsilon & \text{jestliže } a = b \end{cases}$$

$$\frac{d(U+V)}{da} = \frac{dU}{da} + \frac{dV}{da}$$

$$\frac{d(U.V)}{da} = \frac{dU}{da}.V + \{\frac{dV}{da} : \varepsilon \in h(U)\}$$

$$\frac{d(V^*)}{da} = \frac{dV}{da}.V^*$$

3. Pro $x = a_1 a_2 \dots a_n,\ a_i \in T$ platí

$$\frac{dV}{dx} = \frac{dV}{da_n}(\frac{dV}{da_{n-1}}(\dots \frac{dV}{da_2}(\frac{dV}{da_1})\dots))$$

Příklad 4.5:

Je dán regulární výraz $y = (0+1)^*.1$. Určíme několik derivací výrazu y.

$$\frac{dy}{d\varepsilon} = (0+1)^*.1$$

$$\frac{dy}{d1} = \frac{d(0+1)^*}{d1}.1 + \frac{d1}{d1} = \frac{d(0+1)}{d1}.(0+1)^*.1 + \varepsilon$$

$$= (\frac{d0}{d1} + \frac{d1}{d1})(0+1)^*.1 + \varepsilon = (\emptyset + \varepsilon).(0+1)^*.1 + \varepsilon$$

$$\frac{dy}{d0} = \frac{d(0+1)^*}{d0}.1 + \frac{d1}{d0} = \frac{d(0+1)}{d0}.(0+1)^*.1 + \emptyset = (\varepsilon + \emptyset).(0+1)^*.1 + \emptyset.$$

Z pravidel pro derivaci regulárního výrazu V podle řetězce x se dá lehce odvodit, že platí:

$$\frac{dV}{dx} = \{y : xy \in h(V)\}.$$

To znamená, že můžeme říci, že derivací výrazu V podle x je výraz U takový, že $h(U)$ obsahuje řetězce, které vzniknou odtržením předpony x v řetězcích z $h(V)$.

Dále uvedeme způsob konstrukce konečného automatu pro zadaný regulární výraz. Tato metoda je založena na tom, že regulární výraz má konečný počet derivací, které nejsou podobné.

Regulární výrazy x, y jsou podobné, když jeden z nich může být transformován na druhý pomocí těchto rovností:

$$x + x = x$$
$$x + y = y + x$$
$$(x + y) + z = x + (y + z)$$
$$x + \emptyset = x$$
$$x.\emptyset = \emptyset.x = \emptyset$$
$$x.\varepsilon = \varepsilon.x = x$$

Algoritmus 4.6:

Konstrukce ekvivalentního konečného automatu pro daný regulární výraz.
Vstup: Regulární výraz V nad abecedou T.
Výstup: Konečný automat $M = (K, T, \delta, q_0, F)$ takový, že $h(V) = L(M)$.
Metoda:

1. Položíme $Q = \{V\},\ Q_0 = \{V\},\ i := 1$.
2. Vytvoříme derivace všech výrazů z Q_{i-1} podle všech symbolů z abecedy T. Do množiny Q_i vložíme všechny výrazy vzniklé derivací výrazů z Q_{i-1}, které nejsou podobné výrazům z Q.
3. Jestliže $Q_i \neq \emptyset$, přidáme Q_i do Q, položíme $i := i + 1$ a přejdeme na krok 2. V případě, že $Q_i = \emptyset$, pak vytvoříme automat $M = (Q, T, \delta, V, F)$, kde zobrazení je vytvořeno takto: $\delta(\frac{dU}{dx}, a) = \frac{dU}{dx'}$ v případě, že výraz $\frac{dU}{dx'}$ je podobný výrazu $\frac{dU}{d(xa)}$.
   Množina $F = \{\frac{dU}{dx} : \varepsilon \in h(\frac{dU}{dx})\}$.

Příklad 4.6:

Sestrojíme pomocí Algoritmu 4.2 konečný automat, který přijímá jazyk definovaný regulárním výrazem $(0+1)^*1$.

1. Položíme $\quad Q = \{(0+1)^*1\}, \quad Q_0 = \{(0+1)^*1\}$.

2. Vypočteme Q_1:

$$\frac{d}{d1}((0+1)^*1) = (\emptyset + \varepsilon)(0+1)^*1 + \varepsilon = (0+1)^*1 + \varepsilon$$

$$\frac{d}{d0}((0+1)^*1) = (\varepsilon + \emptyset)(0+1)^*1 + \emptyset = (0+1)^*1$$

   Protože

$$\frac{d}{d0}((0+1)^*1) = (0+1)^*1,$$

   bude

$$Q_1 = \{(0+1)^*1 + \varepsilon\} \quad \text{a} \quad Q = \{(0+1)^*1, (0+1)^*1 + \varepsilon\}$$

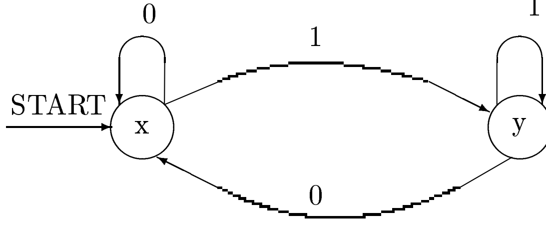
3. Vypočteme Q_2:

$$\frac{d}{d1}((0+1)^*1 + \varepsilon) = (\emptyset + \varepsilon)(0+1)^*1 + \varepsilon + \emptyset = (0+1)^*1 + \varepsilon$$

$$\frac{d}{d0}((0+1)^*1 + \varepsilon) = (\varepsilon + \emptyset)(0+1)^*1 + \emptyset = (0+1)^*1.$$

   Protože oba výrazy již jsou v množině Q, je množina Q_2 prázdná.

4. Výsledný automat bude mít dva stavy: stav x odpovídající výrazu $(0+1)^*1$ a stav y odpovídající výrazu $(0+1)^*1 + \varepsilon$. Počáteční stav bude x, koncový stav bude y, protože $\varepsilon \in h((0+1)^*1 + \varepsilon)$.



Obr. 4.8: Přechodový diagram konečného automatu z příkladu 4.6

Protože platí:

$$\frac{dx}{d1} = y, \quad \frac{dx}{d0} = x, \quad \frac{dy}{d1} = y, \quad \frac{dy}{d0} = x,$$

bude výsledný automat mít přechodový diagram podle obr. 4.8.

Dále uvedeme konstrukci derivace regulárního výrazu pomocí pozičního vektoru. Poziční vektor je množina čísel, které odpovídají pozicím takových symbolů abecedy, které se mohou vyskytnout na začátku zbytku řetězu, který je součástí hodnoty daného regulárního výrazu.

Činnost při vytváření nového pozičního vektoru můžeme shrnout do těchto bodů:

1. Ke každé syntaktické konstrukci se vytvoří seznam počátečních pozic na začátcích členů.
2. Je-li symbol v konstrukci roven symbolu podle kterého se provádí derivace a je-li na označené pozici, pak se označení posouvá před následující pozici.
3. Je-li za konstrukcí operátor iterace a označení je na konci konstrukce, pak se do výsledného seznamu připojí také seznam začátečních pozic náležející této konstrukci.
4. Je-li označení před nějakou konstrukcí, pak se do výsledného seznamu připojí seznam počátečních pozic této konstrukce.
5. Je-li označení před konstrukcí, která generuje také prázdný řetěz, pak se do výsledného seznamu připojí také seznam počátečních pozic konstrukce následující.
6. Má-li se označit konstrukce v závorce, pak je třeba označit začátky všech členů v závorce.

Příklad 4.7:

Postup při derivaci pomocí pozičního vektoru objasníme na jednoduchém příkladě.
Mějme regulární výraz: (1) a . b^* . c
K označení pozic budeme používat šipky. Na začátku bude tedy výraz (1) označen takto:

$$\underset{\wedge}{a} \; . \; b^* \; . \; c \qquad (2)$$

Derivací označeného regulárního výrazu dostaneme nově označený regulární výraz. Základní pravidlo pro derivaci je toto:

1. Je-li označen operand, podle kterého se derivuje, pak se označí místa následující za tímto operandem. Jeho označení se ruší. To znamená, že derivací výrazu (2) podle operandu a dostaneme:

$$a \; . \; \underset{\wedge}{b^*} \; . \; c \qquad (3a)$$

2. Protože je označena konstrukce, která generuje také prázdný řetězec, označíme také konstrukci následující:

$$a \; . \; \underset{\wedge}{b^*} \; . \; \underset{\wedge}{c} \qquad (3b)$$

Nyní derivací podle operandu b výrazu (3b) dostaneme:

$$a \; . \; b^* \; . \; \underset{\wedge}{c} \qquad (4a)$$

3. Protože je označena konstrukce následující za konstrukcí v iteraci musí se označit i předchozí konstrukce.

$$\mathrm{a}\ .\ \underset{\hat{}}{b^*}\ .\ \underset{\hat{}}{\mathrm{c}} \qquad (4b)$$

   Derivací výrazu (4b) podle operandu c dostaneme:

$$\mathrm{a}\ .\ b^*\ .\ \mathrm{c}\ \underset{\hat{}}{} \qquad (5)$$

   Takto označený regulární výraz odpovídá prázdnému regulárnímu výrazu ε.

Časová složitost vyhledávání pomocí deterministického konečného automatu je $\mathcal{O}(T)$ a pomocí nedeterministického konečného automatu je $\mathcal{O}(ST)$, kde T je délka textu a S je počet stavů nedeterministického konečného automatu. Paměťová složitost nedeterministického konečného automatu je $\mathcal{O}(S)$ a ekvivalentního deterministického konečného automatu je $\mathcal{O}(2^S)$.

Pokud pro daný nedeterministický konečný automat konstruujeme jemu ekvivalentní deterministický konečný automat, pak časová složitost této konstrukce a vyhledávání je $\mathcal{O}(2^S + T)$ a paměťová složitost deterministického konečného automatu je $\mathcal{O}(2^S)$.

Lineární paměťová složitost nedeterministického konečného automatu a lineární časová složitost deterministického konečného automatu nabízí možnost použití hybridního deterministicky-nedeterministického přístupu. V takovém případě sestrojíme nedeterministický konečný automat pro daný regulární výraz. Potom nejčastěji navštěvované stavy tohoto automatu převedeme na deterministické. Tyto stavy implementujeme pomocí celočíselné matice, což vede ke konstantní časové složitosti provedení přechodu. Tento přístup zatím nebyl teoreticky prozkoumán a není známa jeho průměrná časová a paměťová složitost.

4.2.2 Vyhledávací metody s předzpracováním vzorků s protisměrným vyhledáváním

Do skupiny metod, které vytváří vyhledávací stroj provádějící protisměrné vyhledávání patří:

1. Boyer-Mooreův algoritmus pro vyhledávání jednoho vzorku,
2. algoritmus Commentz-Walterové pro vyhledávání konečné množiny vzorků,
3. konečný automat sestrojený pro reverzovaný regulární výraz.

4.2.2.1 Boyer-Mooreův algoritmus

Boyer a Moore navrhli důmyslný a zajímavý algoritmus pro vyhledání vzorku v textu, který přeskakuje úseky prohledávaného textu, ve kterých nemůže být vzorek obsažen. Pro abecedu s větší mohutností má algoritmus průměrnou časovou složitost $\mathcal{O}(T/V)$. Základní myšlenka algoritmu spočívá v posouvání vzorku po textu a srovnávání znaků vzorku a textu zprava doleva. Na začátku se srovnává znak `VZOREK[V]` se znakem `TEXT[V]`. Jestliže se znak `TEXT[V]` nevyskytuje ve vzorku, pak je možno posunout vzorek o `V` znaků doprava a srovnávat znak `VZOREK[V]` se znakem `TEXT[2*V]`.

Jestliže se srovnávané znaky rovnají, pokračuje se ve srovnávání znaku vzorku se znaky textu zprava doleva dokud není celý vzorek srovnán nebo dojde k neshodě. V případě neshody je možno použít různé postupy na určení, jak daleko posunout vzorek doprava. Vyhledávací Boyer-Mooreův algoritmus má tvar:

```
var TEXT:array[1..T] of char;
    VZOREK:array[1..V] of char;
    I,J :  integer;
    NALEZEN : boolean;
    ...
begin
    NALEZEN:=false;
    I:=V;
    while(I<=T) and not NALEZEN do
          begin
              J:=0;
              while (J<V) and (VZOREK[V-J]=TEXT[I-J]) do J:=J+1;
              NALEZEN:=J=V;
              if not NALEZEN then I:=I+SHIFT(TEXT[I-J]],J)
           end
end
```

Tento algoritmus používá funkci $SHIFT$, která určuje, jak daleko posunout vzorek doprava, když `VZOREK[V-J] <> TEXT[I-J]`. Existuje celá řada způsobů, jak definovat funkci SHIFT. Uvedeme jeden z nich.

Jestliže znak textu `TEXT[I-J]` není ve vzorku obsažen, pak se vzorek posouvá o celou dosud nepoužitou část vzorku, t.j. o `V-J`, doprava. V případě, že k neshodě došlo pro poslední znak vzorku, provede se posun o celou délku vzorku, protože v tom okamžiku `J=0`.

V případě, že znak textu `TEXT[I-J]` je ve vzorku obsažen v jeho dosud nepoužité části, t.j. na pozici `J+K` (počítáno odzadu), posune se vzorek o `K` pozic doprava tak, aby `J+K` -tý prvek vzorku odpovídal znaku `TEXT[I-J]`.

Můžeme tedy napsat:

SHIFT(A,J) = **if** A se nevyskytuje ve vzorku **then** V-J
else nejmenší K takové, že VZOREK[V-(J+K)]=A
= min{K:K=V, $0 \leq$ K<V a VZOREK[V-(J+K)]=A}

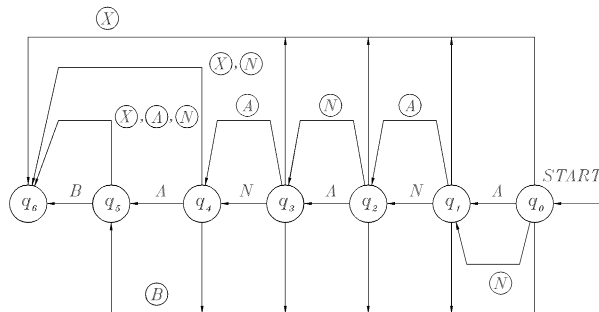
Příklad 4.8:

Ukažme hledání vzorku BANANA v textu I-WANT-TO-FLAVOR-NATURAL-BANANAS. V následující tabulce jsou uvedeny hodnoty funkce SHIFT(ZNAK,J) pro vzorek BANANA. Symbol X zastupuje všechny znaky, které se ve vzorku nevyskytují.

	A	B	N	X
0		5	1	6
1	1	4		5
2		3	1	4
3	1	2		3
4		1	2	2
5	1		1	1

Hodnoty funkce SHIFT pro vzorek BANANA.

Na obr. 4.9 je znázorněn Boyer-Mooreův (BM) vyhledávací stroj. Hrany označené znaky v kroužku ukazují, jak se má vzorek posunout pro jednotlivé případy.



Obr. 4.9: BM vyhledávací stroj pro vzorek BANANA

Vyhledávání probíhá takto (značka označuje srovnávaný znak v textu):

```
BANANA
I-WANT-TO-FLAVOR-NATURAL-BANANAS
     ^
```

(1) T <> A a T se nevyskytuje ve vzorku, posun o délku vzorku (V=6, SHIFT=6).

```
      BANANA
I-WANT-TO-FLAVOR-NATURAL-BANANAS
           ^
```

(2) L <> A a L se nevyskytuje ve vzorku, posun o délku vzorku (SHIFT=6).

```
            BANANA
I-WANT-TO-FLAVOR-NATURAL-BANANAS
                 ^
```

(3) N <> A, ale N se vyskytuje ve vzorku, posun vzorku doprava je SHIFT=1, aby se srovnaly symboly N.

```
             BANANA
I-WANT-TO-FLAVOR-NATURAL-BANANAS
                ^..
```

(4) Protože – <> A, – se ve vzorku nevyskytuje, je možno posunout vzorek za –, tj. SHIFT=4.

```
                 BANANA
I-WANT-TO-FLAVOR-NATURAL-BANANAS
                     ^.
```

(5) A = A, ale R <> N, SHIFT=5 vzhledem k tomu, že R se ve vzorku nevyskytuje.

```
                      BANANA
I-WANT-TO-FLAVOR-NATURAL-BANANAS
                           ^
```

(6) N <> A, SHIFT=1, protože N se vyskytuje ve vzorku

```
                       BANANA
I-WANT-TO-FLAVOR-NATURAL-BANANAS
                         ^...
```

(7) ANA = ANA, ale B <> N; SHIFT=2, protože B se vyskytuje ve vzorku o 2 znaky vlevo.

```
                         BANANA
I-WANT-TO-FLAVOR-NATURAL-BANANAS
                         ......
```

(8) Vzorek nalezen.

4.2.2.2 Algoritmus Commentz-Walterové

Commentz-Walterová navrhla algoritmus, který je založen na kombinaci Boyer-Mooreova algoritmu a algoritmu Aho-Corasickové. Tento algoritmus je určen pro vyhledávání konečné množiny vzorků protisměrným vyhledáváním. Předzpracováním konečné množiny vzorků $p = \{v_1, v_2, \ldots, v_k\}$ vznikne CW vyhledávací stroj a pak vyhledávání probíhá podle následujícího algoritmu:

```
const LMIN = /délka nejkratšího vzorku/
var TEXT : array [1..T] of char;
    I,J : integer;
    NALEZEN : boolean;
```

```
    STATE : TSTATE;
    g : array[1..MAXSTATE,1..MAXSYMBOL] of TSTATE;
    F : set of TSTATE;
...
begin
     NALEZEN:=FALSE;
     STATE:=q0;
     I:=LMIN;
     J:=0;
     while (I<=T) and not NALEZEN do
         begin
             if g[STATE,TEXT[I-J]]=fail
                 then
                     begin
                         I:=I+SHIFT[STATE,TEXT[I-J]];
                         STATE:=q0;
                         J:=0;
                     end
                 else
                     begin
                         STATE:=g[STATE,TEXT[I-J]];
                         J:=J+1;
                     end:
             NALEZEN:=STATE in F
         end;
end
```

CW vyhledávací stroj postupně porovnává text a vzorek zprava doleva a v případě, že `g[STATE, TEXT[I-J]]` má hodnotu $fail$, posune se celý vyhledávací stroj doprava o hodnotu $SHIFT$, která závisí na stavu a právě srovnávaném znaku v textu. Pak opět začíná vyhledávání a CW vyhledávací stroj vychází z počátečního stavu.

CW vyhledávací stroj zkonstruujeme pomocí algoritmu 4.7. Sestrojíme funkci g a zavedeme ohodnocení w jednotlivých stavů:

Algoritmus 4.7:

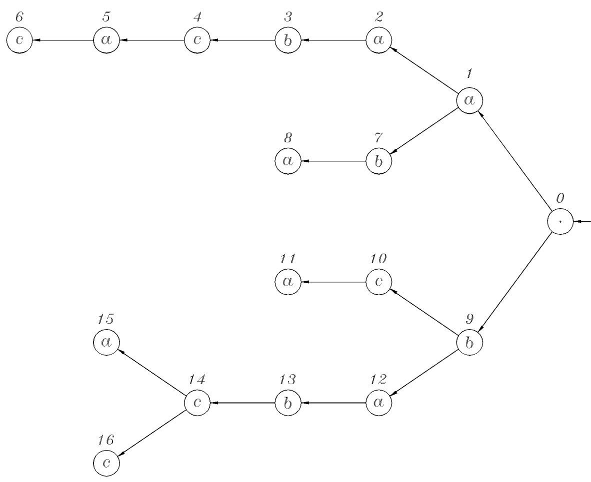
Konstrukce CW vyhledávacího stroje.
Vstup: Množina vzorků p=$\{v_1, v_2, \ldots, v_k\}$
Výstup: CW vyhledávací stroj
Metoda:

1. Počáteční stav bude q_0; $w(q_0) = \varepsilon$.
2. Každý stav vyhledávacího stroje odpovídá příponě $b_m b_{m+1} \ldots b_n$ nějakého vzorku v_i z množiny p. Definujeme $g(q, b_{m-1}) = q'$, kde q' odpovídá příponě $b_{m-1} b_m b_{m+1} \ldots b_n$ vzorku v_i. $w(q) = b_n \ldots b_{m+1} b_m$, $w(q') = w(q) b_{m-1}$
3. $g(q, a) = fail$ pro všechna q a a, pro která $g(q, a)$ nebylo definováno v kroku 2.
4. Každý stav, který odpovídá úplnému vzorku bude koncový stav.

Příklad 4.9:

Sestrojíme CW vyhledávací stroj pro množinu vzorků $p = \{cacbaa, aba, acb, acbab, ccbab\}$. Výsledek je uveden na obr. 4.10. Jednotlivé stavy CW stroje jsou ohodnoceny symboly, při kterých se do nich přechází. Počáteční stav je označen šipkou.

Poslední otázkou, kterou je třeba vyřešit, je výpočet funkce $SHIFT$. Je opět několik možností, jak tuto funkci definovat. Zde uvedeme návrh autorky:



Obr. 4.10: CW vyhledávací stroj pro množinu vzorků $p = \{cacbaa, aba, acb, acbab, ccbab\}$

$SHIFT[STATE, TEXT[I-J]] =$

$\min(\max(shift1(STATE), char(TEXT[I-J]) - J - 1), shift2(STATE)).$

Jednotlivé funkce jsou definovány takto:

1. $char(a)$ je definována pro všechny symboly z abecedy T jako nejmenší hloubka stavu, do kterého se v CW vyhledávacím stroji přechází při symbolu a. Pokud symbol a není v žádném vzorku, je $char(a) = LMIN + 1$, kde LMIN je délka nejkratšího vzorku. Formálně:

   $char(a) = min(\{d(q) : w(q) = xa\}, LMIN + 1).$

2. Funkce $shift1(q)$ má hodnotu $shift1(q_0) = 1$. Pro ostatní stavy má hodnotu

   $shift1(q) = min(k : \{k = d(q') - d(q), w(q') = w(q)u$

   pro nějaké neprázdné slovo $u\}, LMIN).$

   Stav q' má větší hloubku než q a $w(q)$ je vlastní příponou $w(q')$.

3. Funkce $shift2$ má hodnotu $shift2(q_0) = LMIN$. Pro ostatní stavy má hodnotu

   $shift2(q) = min(k : \{k = d(q') - d(q), w(q') = w(q)u$

   pro nějaké neprázdné slovo u, q' je koncový stav$\},$

   $\{shift2(q') : q'$ je předchůdce $q\}).$

Příklad 4.10:

CW vyhledávací stroj pro množinu vzorků $p = \{cacbaa, aba, acb, acbab, ccbab\}$ z příkladu 4.9 bude mít jednotlivé funkce definovány takto:

	a	b	c	x
$char$	1	1	2	4

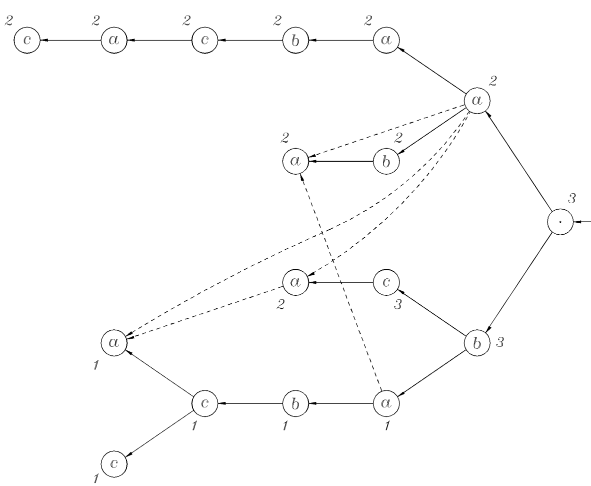
Poznámka: x zastupuje ostatní symboly abecedy.

Funkce $shift1$ je naznačena na obr. 4.11 jako ohodnocení jednotlivých uzlů a čárkovaně jsou pro příslušné stavy q uvedeny jim odpovídající stavy q', které byly při výpočtu funkce $shift1$ vzaty v úvahu. Na obr. 4.12 je totéž provedeno pro funkci $shift2$.

Obr. 4.11: Výpočet funkce $shift1$

4.2.2.3 Protisměrné vyhledávání nekonečné množiny vzorků

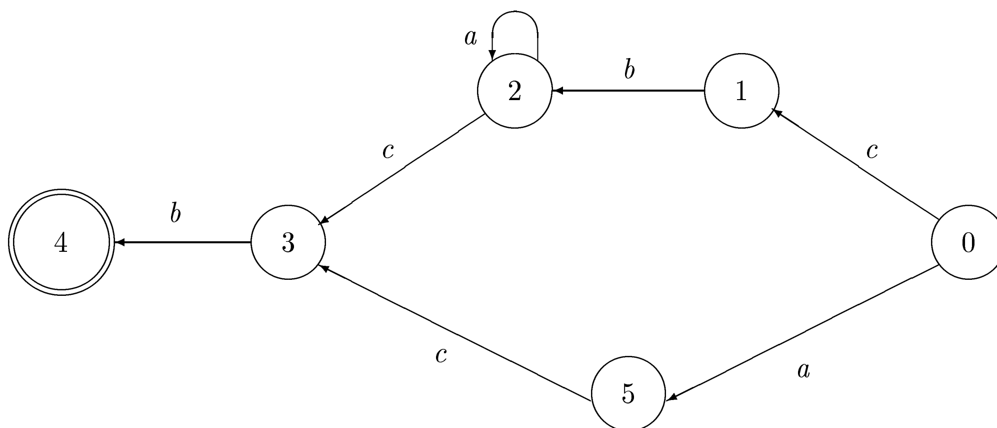
Podobně jako při sousměrném vyhledávání je možno použít konečný automat i při protisměrném vyhledávání. V tomto případě sestrojíme konečný automat pro reverzovaný regulární výraz popisující množinu vzorků (konečnou či nekonečnou). Potom pro jednotlivé stavy a vstupní symboly určíme velikosti posuvů. Tento postup navrhl Buczilowski.



Obr. 4.12: Výpočet funkce $shift2$

Příklad 4.11:

Je dán regulární výraz $RV = bc(a + a^*bc)$. Reverzovaný výraz má tvar $RV^R = (a + cba^*)cb$. Pro tento výraz sestrojíme konečný automat, jehož přechodový diagram je na obr. 4.13.



Obr. 4.13: Konečný automat pro reverzovaný regulární výraz $RV^R = (a + cba^*)cb$

V následující tabulce jsou uvedeny velikosti posuvů ($shift[STAV, SYMBOL]$) pro případ, kdy v automatu není možný správný přechod. Symbol x zastupuje ostatní symboly abecedy.

		symboly			
		a	b	c	x
stavy	0		1		3
	1	1		1	2
	2		1		1
	3	1		1	1
	4	1	1	1	1
	5	2	2		2

Hodnoty funkce $shift$ z příkladu 4.11

V tabulce jsou důležité zejména případy, kdy posuv je větší než jedna. Rozebereme tyto případy podrobněji:

- $shift[0, x] = 3$
  Symbol x není součástí žádného ze vzorků a proto posuv se provede o délku nejkratšího vzorku, která je 3.
- $shift[1, x] = 2$
  Platí totéž co v předchozím případě s tím, že předchozí symbol byl c a je tedy možno provést posuv o 2, protože symbol c se vyskytuje ještě při přechodu ze stavu 2.
- $shift[5, a] = 2$
  Symbol a se nevyskytuje jako přípona žádného vzorku začínající ve stavu 5. Posuv se provede o 2, protože symbol a se vyskytuje při přechodu ze stavu 2.
- $shift[5, b] = 2$
  $shift[5, x] = 2$
  Před symbolem b nebo x byl nalezen symbol a. Posuv se provede o 2, protože symbol a se vyskytuje při přechodu ze stavu 2.

4.2.2.4 Dvoucestný automat se skokem

Kromě použití konečného automatu pro protisměrné vyhledávání navrhl Buczilowski také dvoucestný konečný automat se skokem jako zobecnění všech vyhledávácích strojů pro sousměrné a protisměrné vyhledávání.

Dvoucestný deterministický konečný automat se skokem (2DFJA) je $M = (Q, T, \delta, q_0, k, \uparrow, F)$, kde

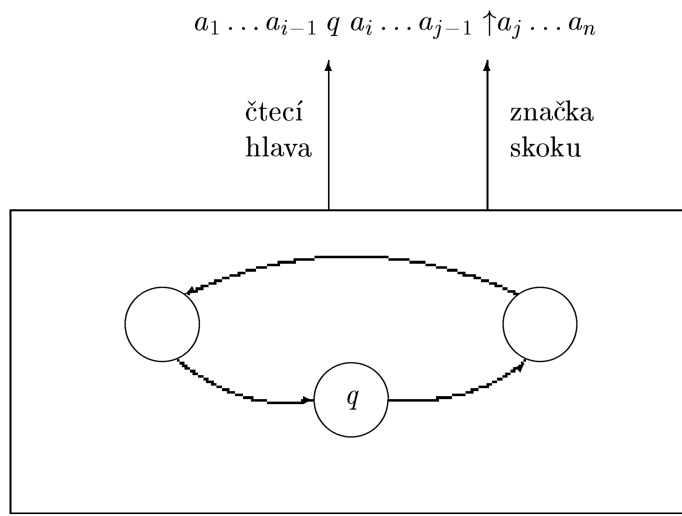
Q	je množina stavů,
T	je vstupní abeceda,
$q_0 \in Q$	je počáteční stav,
δ	je zobrazení z $Q \times T$ do $Q \times \{-1, 1, \dots, k\}$,
$k \in N$	je maximální délka skoku,
$\uparrow \notin Q \cup T$	je značka skoku,
$F \subset Q$	je množina koncových stavů.

Konfigurace dvoucestného deterministického konečného automatu M je řetězec z množiny $T^*QT^* \uparrow T^*$. Například řetězec

$a_1 a_2 \dots a_{i-1} q a_i \dots a_{j-1} \uparrow a_j \dots a_n$

je konfigurace. Význam této konfigurace je znázorněn na obr. 4.14. Množinu konfigurací automatu M označíme $K(M)$.

Přechod je relace, kterou označíme $\vdash$, a je podmnožinou $K(M) \times K(M)$ definovanou takto:



Obr. 4.14: Význam konfigurace 2DFJA

- $a_1 \dots a_{i-1} q a_i \dots a_{j-1} \uparrow a_j \dots a_n \vdash a_1 \dots a_{i-2} q' a_{i-1} a_i \dots a_{j-1} \uparrow a_j \dots a_n$
  pro $i > 1,\ \delta(q, a_i) = (q', -1)$,
- $a_1 \dots a_{i-1} q a_i \dots a_{j-1} \uparrow a_j \dots a_n \vdash a_1 \dots a_{i-1} a_i \dots a_{t-1} q' \uparrow a_t \dots a_n$
  pro $\delta(q, a_i) = (q', m),\ m \geq 1, t = min(j + m, n + 1)$.

Symboly $\overset{k}{\vdash}, \overset{+}{\vdash}, \overset{*}{\vdash}$ použijeme pro označení k-té mocniny, transitivního, transitivního a reflexivního uzávěru relace $\vdash$.

Jazyk přijímaný automatem $M = (Q, T, \delta, q_0, k, \uparrow, F)$ je množina

$L(M) = \{w \in T^* : q_0 \uparrow w \vdash^* wf \uparrow, f \in F\}$.

Je dokázáno, že dvoucestný deterministický konečný automat se skokem je formální systém pro specifikaci regulárních jazyků. To znamená, že pro každý 2DFJA existuje ekvivaletní klasický konečný automat. Pro sousměrné vyhledávání můžeme příslušné vyhledávací stroje simulovat pomocí 2DFJA tak, že v každé konfiguraci bude označení stavu a značka skoku bezprostředně za sebou.

Příklad 4.12:

Sestrojíme 2DFJA pro BM vyhledávácí stroj z příkladu 4.8.

$M = (\{start, q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \{A, B, N, X\}, \delta, start, 6, \uparrow, \{q_6\})$

Přechodová tabulka má tvar:

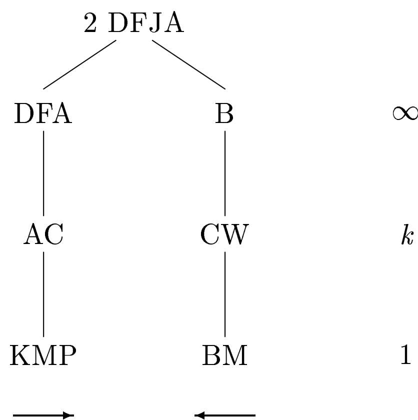
δ	A	B	N	X
start	$(q_0, 5)$	$(q_0, 5)$	$(q_0, 5)$	$(q_0, 5)$
q_0	$(q_1, -1)$	$(q_0, 5)$	$(q_0, 1)$	$(q_0, 6)$
q_1	$(q_0, 1)$	$(q_0, 4)$	$(q_2, -1)$	$(q_0, 5)$
q_2	$(q_3, -1)$	$(q_0, 3)$	$(q_0, 1)$	$(q_0, 4)$
q_3	$(q_0, 1)$	$(q_0, 2)$	$(q_4, -1)$	$(q_0, 3)$
q_4	$(q_5, -1)$	$(q_0, 1)$	$(q_0, 2)$	$(q_0, 2)$
q_5	$(q_0, 1)$	$(q_6, -1)$	$(q_0, 1)$	$(q_0, 1)$
q_6				

Pro text z příkladu 4.8 provede automat M tuto posloupnost přechodů:

start $\uparrow$ I-WANT-TO-FLAVOR-NATURAL-BANANAS
$\vdash$ I-WANq_0 $\uparrow$T-TO-FLAVOR-NATURAL-BANANAS
$\vdash$ I-WANT-TO-Fq_0 $\uparrow$LAVOR-NATURAL-BANANAS
$\vdash$ I-WANT-TO-FLAVOR-q_0 $\uparrow$NATURAL-BANANAS
$\vdash$ I-WANT-TO-FLAVOR-Nq_0 $\uparrow$ATURAL-BANANAS
$\vdash$ I-WANT-TO-FLAVOR-q_1N $\uparrow$ATURAL-BANANAS
$\vdash$ I-WANT-TO-FLAVORq_2-N $\uparrow$ATURAL-BANANAS
$\vdash$ I-WANT-TO-FLAVOR-NATURq_0 $\uparrow$AL-BANANAS
$\vdash$ I-WANT-TO-FLAVOR-NATUq_1R $\uparrow$AL-BANANAS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-BAq_0 $\uparrow$NANAS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-BANq_0 $\uparrow$ANAS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-BAq_1N $\uparrow$ANAS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-Bq_2AN $\uparrow$ANAS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-q_3BAN $\uparrow$ANAS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-BANANq_0 $\uparrow$AS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-BANAq_1N $\uparrow$AS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-BANq_2AN $\uparrow$AS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-BAq_3NAN $\uparrow$AS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-Bq_4ANAN $\uparrow$AS
$\vdash$ I-WANT-TO-FLAVOR-NATURAL-q_5BANAN $\uparrow$AS
$\vdash$ I-WANT-TO-FLAVOR-NATURALq_6-BANAN $\uparrow$AS

4.2.3 Shrnutí

Metody sousměrného a protisměrného vyhledávání je možno uspořádat podle toho, jak velkou množinu vzorků vyhledávají. Toto uspořádání je na obr. 4.15.



Obr. 4.15: Shrnutí metod sousměrného a protisměrného vyhledávání v textu

Jednotlivé zkratky mají tento význam:

KMP	– KNUTH-MORRIS-PRATT
AC	– AHO-CORASICKOVÁ
DFA	– DETERMINISTICKÝ KONEČNÝ AUTOMAT
BM	– BOYER-MOORE
CW	– COMMENTZ-WALTEROVÁ
B	– BUCZILOWSKI
2DFJA	– DVOUCESTNÝ DETERMINISTICKÝ KONEČNÝ AUTOMAT SE SKOKEM

4.3 Vyhledávací metody s předzpracováním textu – indexové metody

Metoda, kterou se budeme zabývat v této kapitole, patří do skupiny III, při které se provádí předzpracování textu a neprovádí se předzpracování vzorků.

Základními pojmy této metody jsou pojmy index a indexový soubor. Pojem index je velmi široký, ale ve vyhledávacích systémech je to obvykle uspořádaná množina slov, která jsou obsažena v textu. U každého slova v indexu jsou uvedeny odkazy do textu, které ukazují, kde se příslušné slovo vyskytuje. Je to obdobný způsob, jaký se používá v kvalitnějších odborných knihách při konstrukci rejstříku. Indexový soubor pak obsahuje slova tvořící index s identifikací jejich výskytu v textu. Také se používá pojmu indexsekvenční soubor, čímž se míní indexový soubor a sekvenční soubor obsahující text.

Index je možno implementovat mnoha způsoby. Nejnázornější je implementace indexu ve formě invertovaného souboru. Tento princip ukážeme na příkladě.

Rozsáhlý text bývá zpravidla hierarchicky rozčleněn na úrovně. Například text – dokument – odstavec – věta – slovo. Odkazy v indexu lokalizují výskyty daného slova v textu. Předpokládejme, že text je rozčleněn na dokumenty, které jsou průběžně očíslovány. Následující tabulka popisuje výskyty jednotlivých slov v dokumentech uložených v sekvenčním souboru.

	slovo1	slovo2	slovo3	slovo4
dokument1	1	1	0	1
dokument2	0	1	1	1
dokument3	1	0	1	1

Jednička znamená, že slovo se vyskytuje v dokumentu, nula znamená, že slovo se nevyskytuje v dokumentu. Invertovaný soubor vznikne z této tabulky operací transpozice matice.

	dokument1	dokument2	dokument3
slovo1	1	0	1
slovo2	1	1	0
slovo3	0	1	1
slovo4	1	1	1

Hledáme-li např. dokument, který obsahuje slovo3, je zřejmé, že je obsaženo v dokumentu2 a dokumentu3. Přitom vyhledávání vzorku v indexu je možno provést binárním půlením, protože slova mohou být v indexu uspořádána.

Časová složitost nalezení dokumentů pro jeden vzorek při binárním půlení je $\mathcal{O}(V * \log(i))$, kde V je délka vzorku a i je délka indexu. Pro množinu vzorků $p = \{v_1,\ v_2,\ \cdots,\ v_k\}$ můžeme použít dvě metody. Jestliže počet vzorků je mnohem menší než délka indexu $(k << i)$, pak je výhodné použít opakovaně vyhledávání binárním půlením. V tomto případě je časová složitost nalezení dokumentů, ve kterých se vyskytuje množina vzorků $p = \{v_1,\ v_2,\ \cdots,\ v_k\}$, rovna $\mathcal{O}(s * k * log(i))$, kde i je délka indexu, s je průměrná délka vzorku. V opačném případě je výhodné použít vyhledání vzorků metodou dvojitého slovníku.

Metoda vyhledávání vzorků pomocí dvojitého slovníku je běžně používaná metoda při porovnávání dvou uspořádaných množin. V našem případě se jedná o množiny slov. Nejdříve je třeba uspořádat množinu vzorků a vytvořit uspořádaný index. Podstata vyhledávacího algoritmu pak spočívá v porovnávání dvou uspořádaných množin slov. Počet porovnání není větší než součet počtu slov v obou seznamech. Vyhledávání má tedy složitost $\mathcal{O}((k+i) * s)$, kde i je počet slov v indexu, k je počet vzorků a s je průměrná délka vzorku.

Problémem, který se nevyskytl v metodách nepoužívajících předzpracování textu, je přidání dokumentu do souboru. Při přidání dokumentu do souboru je nutno upravit invertovaný soubor,

což znamená přidání sloupce pro nový dokument a případně přidání nových slov. Přitom je nutno stále udržovat index uspořádaný.

4.3.1 Implementace indexových systémů

Indexový systém se skládá z indexového souboru, ve kterém jsou uloženy informace o přístupu k jednotlivým dokumentům, a sekvenčního souboru, ve kterém jsou uloženy vlastní dokumenty. Z mnoha implementací vybereme tři:

1. přímé použití invertovaného souboru,
2. použití seznamu dokumentů ke každému klíčovému slovu,
3. souřadnicový systém s ukazateli.

4.3.1.1 Použití invertovaného souboru

Nejjednodušší způsob implementace indexového systému je přiřazení bitového vektoru ke každému klíčovému slovu. Pozice jedniček v tomto vektoru udávají čísla dokumentů, ve kterých se klíčové slovo vyskytuje. Dokumenty jsou tedy očíslovány. Tento princip je naznačen na obr. 4.16.

slovo1	1	0	1
slovo2	1	1	0
slovo3	0	1	1
slovo4	1	1	1

Obr. 4.16: Indexový soubor s bitovým vektorem

Z tohoto obrázku je zřejmé, že například slovo3 se vyskytuje v dokumentech číslo 2 a 3.

4.3.1.2 Použití seznamu dokumentů

Stejně jako v předchozím případě jsou jednotlivé dokumenty očíslovány a ke každému klíčovému slovu je přiřazen seznam čísel dokumentů, ve kterých se toto slovo vyskytuje. Indexový soubor je uveden na obr. 4.17.

slovo1	1, 3
slovo2	1, 2
slovo3	2, 3
slovo4	1, 2, 3

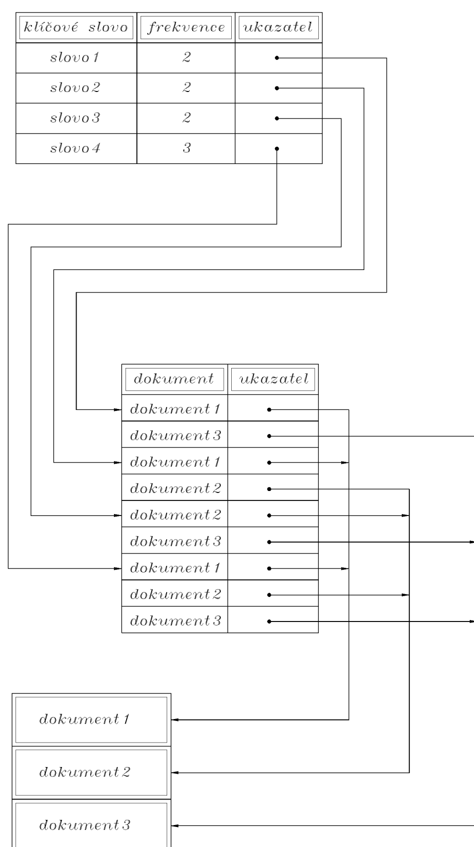
Obr. 4.17: Indexový soubor se seznamem čísel dokumentů

4.3.1.3 Souřadnicový systém s ukazateli

V tomto případě se nepracuje s čísly dokumentů, ale indexový soubor je rozdělen na dvě části:

- slovník s ukazateli do seznamu dokumentů,
- seznam dokumentů s ukazateli na dokumenty.

Celé uspořádání je znázorněno na obr. 4.18.



Obr. 4.18: Souřadnicový systém s ukazateli

4.3.2 Metody indexování

Ze všech operací v oblasti indexových metod je nejsložitější řešení problému nalezení vhodných slov, která mají být součástí indexu. Slova, která jsou součástí indexu, musí být vybrána tak, aby dostatečně přesně reprezentovala obsah jednotlivých dokumentů. Je zřejmé, že slovo, které se vyskytuje ve všech dokumentech nemá z hlediska vyhledávání žádný význam. Malý význam má i slovo, které se vyskytne jen v jednom dokumentu.

Úloha nalezení vhodných slov do indexu se nazývá indexování. Způsoby, jak se indexování provádí, můžeme rozdělit na

- ruční,
- automatické.

Ruční indexování provádí zkušení experti zejména při zpracování bibliografických dokumentů. Je to práce náročná, a proto je snaha tento úkol algoritmizovat. Uvedeme dále jednoduchou metodu automatického indexování založenou na frekvenci výskytu slov v jednotlivých dokumentech.

Při všech metodách se obvykle definuje určitá množina slov, která se při indexování používat nebudou. Jsou to slova, která mají v jazykových textech jen gramatický význam. Patří sem slovní druhy jako předložky, spojky, zájmena, dále pak pomocná a způsobová slovesa, členy a podobně. Seznam těchto slov (stop seznam, angl. stop list) se obvykle vytváří ručně. Při tvorbě tohoto seznamu je možné využít výsledků analýzy textu, protože slova ve stop seznamu jsou hodně frekventovaná a krátká.

Dalším kriteriem pro klasifikaci indexovacích metod je otázka, zda je indexování

- neřízené nebo
- řízené.

Při řízeném indexování je používán speciální slovník slov, která se používají pro indexování. Existují dva přístupy k tvorbě takového slovníku. První přístup je založen na seznamu slov, který není nijak vnitřně strukturován (angl. pass list). Druhý přístup je založen na slovníku, který je speciálním způsobem strukturován. Tento slovník se nazývá tezaurus. V dalším textu se budeme zejména zabývat otázkou automatického indexování a otázkou struktury a tvorby tezauru.

Výběr slov do indexu se provádí tak, aby byly splněny tyto požadavky:

1. Nalézt dokumenty, které se týkají oblasti zájmů uživatele.
2. Dát do souvislosti dokumenty, které se týkají podobných a souvisejících oblastí.
3. Užít slova s dobře definovaným významem tak, aby dobře charakterizovala jednotlivé dokumenty.

Dříve než se budeme zabývat metodami automatického indexování, shrneme stručně některé principy používané při ručním indexování.

Při neřízeném ručním indexování je třeba vzít v úvahu pro každé slovo jeho synonyma a příbuzná slova a pokud možno všechna použít pro indexování. Při řízeném ručním indexování je tato operace usnadněna tím, že v tezauru jsou seznamy synonym a příbuzných slov uvedeny.

4.3.2.1 Analýza textu

Úkolem indexování je nalezení slov, která charakterizují obsah určitého dokumentu a určení váhy nebo hodnoty těchto slov vzhledem k jejich důležitosti k identifikaci dokumentu. Lze očekávat, že taková slova jsou především součástí uvažovaného dokumentu. Proto se budeme zabývat otázkou analýzy textu vedoucí k výběru slov do indexu.

Většina automatických indexovacích metod vychází ze skutečnosti, že frekvence výskytu jednotlivých slov má přímou souvislost s jejich významností při identifikaci dokumentu.

Pokud by se jednotlivá slova vyskytovala ve všech dokumentech se stejnou frekvencí, nebylo by možné vybrat žádná z nich jako vhodná pro identifikaci jednotlivých dokumentů. Skutečností je, že jednotlivá slova se vyskytují v textu s různou frekvencí. Jestliže vytvoříme seznam slov a setřídíme jej podle klesající frekvence jejich výskytů, dostaneme frekvenční slovník. Začátek takového frekvenčního slovníku pro angličtinu je uveden v tab. 4.2. Tento slovník byl vytvořen ze souboru dokumentů obsahujících 1 000 000 slov.

pořadí	slovo	frekvence	pořadí ∗ frekvence
1	the	69971	0.070
2	of	36411	0.073
3	and	28852	0.086
4	to	26149	0.104
5	a	23237	0.116
6	in	21341	0.128
7	that	10595	0.074
8	is	10099	0.081
9	was	9816	0.088
10	he	9543	0.095

Tabulka 4.2: Začátek frekvenčního slovníku

Frekvenční slovník lze charakterizovat empirickým Zipfovým zákonem:

$$\textit{pořadí} * \textit{frekvence} \cong \textit{konstanta}$$

Tento zákon je vysvětlován pomocí obecného „principu nejmenšího odporu“. V tomto případě to znamená, že mluvčí či pisatel raději některá slova častěji opakuje místo používání nových a různých slov. Dále je vidět, že slova s vysokou frekvencí jsou obvykle velmi krátká.

Obr. 4.19: Kumulativní podíl používaných slov

Další charakteristikou frekvenčního slovníku je graf ukazující závislost kumulativního podílu používaných slov (KPS) na procentu slov z frekvenčního slovníku.

$$KPS = \frac{\sum_{pořadí=1}^{N} frekvence_{pořadí}}{počet\ slov\ textu}$$

Z grafu na obr. 4.19 je vidět, že 20% nejfrekventovanějších slov tvoří 70% textu.

4.3.2.2 Jednoduchá metoda automatického indexování

Frekvenční slovník, Zipfův zákon a jeho důsledky můžeme použít jako východisko pro jednoduchou metodu odvození významnosti slov pro identifikaci dokumentů.

Nechť je dán soubor n dokumentů. Pak provedeme tyto operace:

1. Vypočteme frekvenci $FREQ_{ik}$ pro každý dokument $i \in \langle 1, n \rangle$ a každé slovo $k \in \langle 1, K \rangle$, kde K je počet různých slov použitých v celém souboru dokumentů.
2. Vypočteme frekvenci $TOTFREQ_k$ pro každé slovo ve všech dokumentech dohromady:

$$TOTFREQ_k = \sum_{i=1}^{n} FREQ_{ik}$$

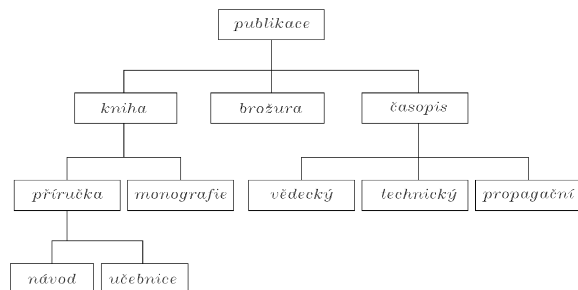
3. Vytvoříme frekvenční slovník pro všechna slova $k \in \langle 1, K \rangle$. Stanovíme práh pro vyloučení velmi frekventovaných slov ze začátku frekvenčního slovníku. Tím vyloučíme velmi frekventovaná slova, která mají buď žádný nebo malý význam pro identifikaci dokumentů.
4. Podobným způsobem vyloučíme slova s velmi nízkou frekvencí.
5. Zbývající slova se střední frekvencí jsou vhodná pro zařazení do indexu.

Tato metoda se opírá o empiricky zjištěnou skutečnost, že slova s nízkou a vysokou frekvencí mají malý význam pro vyhledávání. Na obr. 4.20 je uveden graf ukazující význam slova pro vyhledávání v závislosti na frekvenci jeho výskytu.

Obr. 4.20: Význam slov pro vyhledávání v závislosti na jejich frekvenci

4.3.2.3 Řízené indexování

Základním prostředkem pro řízení procesu indexování je stanovení indexovacího jazyka, který obsahuje omezený výběr termínů (slov) pro indexování. Takový slovník se nazývá *tezaurus*. Tezaurus je slovník, který obsahuje hierarchické a asociativní vztahy a vztahy ekvivalence mezi jednotlivými termíny. Tyto vztahy můžeme rozdělit do čtyř skupin:



Obr. 4.21: Příklad hierarchické struktury tezauru

1. Vazba na standardní termín (See ...). Pro každou množinu ekvivalentních termínů (synonym) je stanoven standardní (preferovaný) termín. Při indexování je vždy vybrán standardní termín bez ohledu na to, jaký termín je použit v dokumentu nebo v dotazu. Jen standardní termín je použit v indexu.
2. Vazba na příbuzné termíny (See also ..., related terms, RT). Mezi jednotlivými příbuznými termíny jsou uvedeny vzájemné vazby. Tyto termíny pak tvoří skupiny příbuzných termínů.
3. Vazba na obecnější (širší) termíny (broader terms, BT).
4. Vazba na užší termín (narrower terms, NT).

Tezaurus je používán nejen pro řízení indexovacího procesu, ale také při vlastním vyhledávání. Při zpracování dotazu se použité termíny upraví na standardní termíny a dále je možno na základě požadavků uživatele přidat pro vyhledávání příbuzné, obecnější i užší termíny. Tezaurus můžeme znázornit ve formě stromu. Tento strom znázorňuje vazby mezi užšími a širšími termíny a mezi příbuznými termíny. Na obr. 4.21 je příklad hierarchické struktury tezauru pro vyhledávání v oblasti publikací.

Z obrázku je vidět, že *publikace* je obecnější termín pro termíny *kniha*, *brožura* a *časopis*. *Časopis* má užší termíny *vědecký*, *technický* a *propagační*. *Kniha*, *brožura* a *časopis* jsou příbuzné termíny. Vazba mezi ekvivalentními termíny zde není vyjádřena, protože v tezauru jsou použity jen standardní termíny.

4.3.2.4 Konstrukce tezauru

Tezaurus může být konstruován ručně nebo automatizovaně. Bez ohledu na způsob konstrukce je třeba provést:

- rozhodnutí o tom, které termíny mají být do tezauru vloženy,
- vytvořit skupiny podobných termínů.

Rozhodnutí o tom, jaké termíny do tezauru vložit, můžeme odvodit z úvahy o vhodnosti určitého termínu při indexování. Z toho, co bylo uvedeno výše, plyne, že termíny se střední frekvencí jsou nejlepšími kandidáty na vložení do tezauru. Také termíny s nižší frekvencí je

možno použít, ale ty je třeba sdružit do skupin tak, aby součet frekvencí jednotlivých termínů ve skupině byl dostatečně vysoký.

Tezaurus se obvykle vytváří pro řízené indexování v určitém oboru. Z toho vyplývají experimentálně potvrzená pravidla o jeho tvorbě:

1. Tezaurus má obsahovat jen termíny, které patří do daného oboru.
2. V řadě případů se jako odborné termíny používají termíny z jiných oborů. V tomto případě je nutno jim v tezauru přisoudit sémantiku obvyklou v daném oboru. Příkladem mohou být termíny POLE a STROM v oblasti informatiky.
3. Při tvorbě skupin příbuzných termínů je dobře sdružovat termíny s podobnou frekvencí. Součet frekvencí termínů by měl být pro všechny skupiny přibližně stejný.
4. Je třeba z tezauru vyloučit termíny s vysokou frekvencí, které mají malý nebo dokonce nulový význam pro rozlišení jednotlivých dokumentů.

4.3.2.5 Vyhledávání vzorků pomocí fragmentových indexů

Metoda, která používá fragmentových indexů, patří mezi metody, které vyžadují jak předzpracování vzorků, tak předzpracování textu. Fragmentový index je založen na myšlence, že význam slova je soustředěn jen v jeho části. Hledáme-li např. molybden, stačí najít podřetězec (fragment) "ybd". Pro určitou úzkou tématickou oblast se dá sestrojit množina fragmentů, kterých je řádově tisíc. Slovník fragmentů může být pevný, čímž odpadají problémy s jeho aktualizací při doplňování souboru dokumentů.

Nevýhodou této metody je závislost na jazyce a tématické oblasti a dále snížená přesnost vyhledávání vzhledem k vyhledávání fragmentů místo celých slov.

4.4 Vyhledávací metody s předzpracováním textu a vzorků – signaturové metody

Signatura S_x je bitový řetězec délky m, který je přiřazen textovému řetězci x zobrazením $h : x \to S_x, S_x \in \{0,1\}^m$. Každý bit signatury S_x vyjadřuje nějakou vlastnost textového řetězce x. Zvolíme-li vhodně zobrazení h, pak je možno porovnáním signatury textu S_t a signatury vzorku S_v zjistit, zda text může či nemůže vzorek obsahovat. Může-li ho obsahovat, pak je třeba nějakým přesným algoritmem ověřit, zda jej skutečně obsahuje. Zřejmě, když

$S_t \textit{ and } S_v \equiv S_v$,

pak text splňuje všechny vlastnosti požadované vzorkem a je nutno prověřit, zda text vzorek skutečně obsahuje. Pro praktickou implementaci je možno použít vztahu:

$(S_t \textit{ and } S_v \not\equiv S_v) \equiv (\textit{not } S_t \textit{ and } S_v)$,

který vede na jednodušší výraz.

Zbývá otázka volby h. V této souvislosti se budeme zabývat těmito otázkami:

- jak vytvořit signatury jednotlivých termínů,
- jak ze signatur jednotlivých termínů vytvořit signaturu dokumentu,
- jak rozdělit rozsáhlé dokumenty na bloky vzhledem k vlastnostem signatur.

Signatury jednotlivých termínů můžeme vytvořit těmito způsoby:

1. Každému termínu přiřadíme jeden bit v signatuře.
2. Signaturu termínu vytvoříme pomocí nějaké hašovací funkce.

Ze signatur jednotlivých termínů vytvoříme signaturu dokumentů

- řetězeným kódováním,
- vrstveným kódováním.

4.4.1 Řetězené a vrstvené kódování

Při řetězeném kódování jsou signatury jednotlivých termínů zřetězeny. Tento přístup se hodí jen pro případy, kdy text je nějakým způsobem strukturován a jednotlivé složky jsou uspořádany. Mějme záznam, který má tvar

$$z = (a_1, a_2, \ldots, a_n).$$

Signatura záznamu $h(z) = h(a_1) \cdot h(a_2) \ldots h(a_n)$. Při vyhledávání je třeba určit hodnotu příslušných prvků záznamu.

Příklad 4.13:

Záznam má tvar $z =$ *(ISBN, AUTOR, TITUL)*. Konkrétní záznam pro jednu knihu bude například $z =$ *(72345, KAREL ČAPEK, KRAKATIT)*.
Signatury jsou zvoleny takto:
h(72345) = 10001001,
h*(KAREL ČAPEK)* = 01011000,
h*(KRAKATIT)* = 10010001.
Signatura záznamu vznikne zřetězením signatur jednotlivých prvků:

$h(z)$ = 10001001 01011000 10010001

Při vyhledávání knih KARLA ČAPKA bude mít signatura dotazu tvar:

???????? 01011000 ????????

Otazníky v signatuře dotazu znamenají, že v tomto místě může mít signatura libovolnou hodnotu.

Při vrstveném kódování jsou signatury jednotlivých termínů použitých v dokumentu logicky sčítány bit po bitu. Jestliže v dokumentu d jsou použita slova $a_1, a_2, \ldots, a_n$, pak

$h(d) = h(a_1)$ *or* $h(a_2)$ *or* ... *or* $h(a_n)$.

Příklad 4.14:

Při použití vrstveného kódování pro dokumenty o knihách z předchozího případu bude signatura určena takto:

hodnota	signatura
7345	10001001
KAREL ČAPEK	01011000
KRAKATIT	10010001
signatura dokumentu	11011001

Při vrstveném kódování dochází k tomu, že počet jedniček v signatuře se zvětšuje s počtem termínů použitých v dokumentu. Pokud počet jedniček vzroste natolik, že téměř všechny bity signatury jsou jedničky, má to negativní důsledek na vyhledávání. Takové dokumenty pak vyhovují téměř všem dotazům a při přesném prohledávání je nutno prohledávat velmi mnoho dokumentů. Experimentálně bylo zjištěno, že z hlediska vyhledávání je nejvhodnější signatura, ve které je 50% jedniček.

Z tohoto důvodu je třeba rozsáhlé dokumenty rozdělit na bloky tak, aby počet jedniček v jejich signaturách nebyl vysoký. Jsou dvě možnosti:

1. rozdělit dokument na bloky stejné délky (FSB – Fixed Size Blocks),
2. rozdělit dokument na bloky o stejné váze (FWB – Fixed Weight Blocks).

Při metodě FWB se signatura vytváří postupně přidáváním jednotlivých termínů. Blok se ukončí tehdy, když počet jedniček je přibližně 50%.

4.4.2 Metody tvorby signatur

Vhodnou kombinací uvedených principů vznikají jednotlivé metody tvorby signatur. Nejjednodušší je metoda založená na *vektoru výskytu slov*. Tato metoda přiřazuje každému termínu jeden bit v signatuře a signatura dokumentu vzniká vrstveným kódováním. To znamená, že pro každý dokument je vytvořen binární řetězec, kde každý bit odpovídá jednomu z možných termínů. Bity odpovídající termínům, které se v dokumentu vyskytují jsou nastaveny na jedničku, ostatní jsou nastaveny na nulu. V následující tabulce je uveden příklad přiřazení signatur jednotlivým termínům.

termín	signatura
data	000001
disc	000010
file	000100
information	001000
storage	010000
tape	100000

Dokument obsahující termíny *data file storage* bude mít signaturu 010101. Tato metoda vede na přesné vyhledávání, což znamená, že pro každý vzorek jsou vybrány právě ty dokumenty, které jej obsahují. Tato výhoda ovšem přináší dva základní problémy:

- signatura je velice dlouhá, protože její délka je dána počtem možných termínů,
- metoda vyžaduje pevný seznam termínů, protože při změně počtu termínů se mění délka signatury.

Tuto nevýhodu odstraňuje použití hašovacích funkcí pro výpočet signatur jednotlivých termínů. V tomto případě je signatura termínu bitový řetězec s konstantní délkou a vrsveným kódováním získáme signaturu dokumentu.

Příklad 4.15:

V následující tabulce je uveden příklad signatur několika termínů.

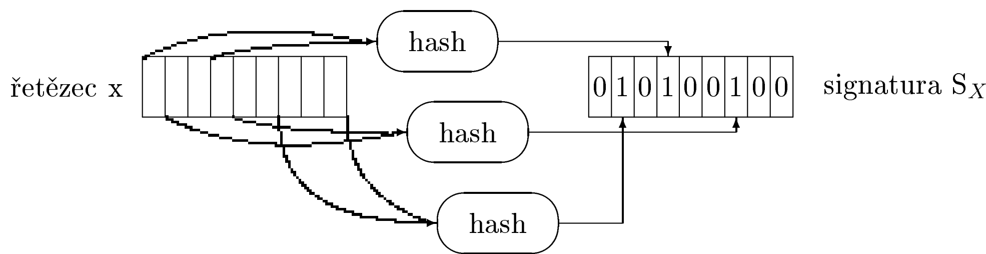
termín	signatura
data	0001
disc	0101
file	1001
information	0110
storage	0011
tape	1010

Signatura dokumentu obsahujícího "data file storage" bude 1011. Jestliže položíme dotaz na dokumenty obsahující termín "file", bude jeho signatura 1001.

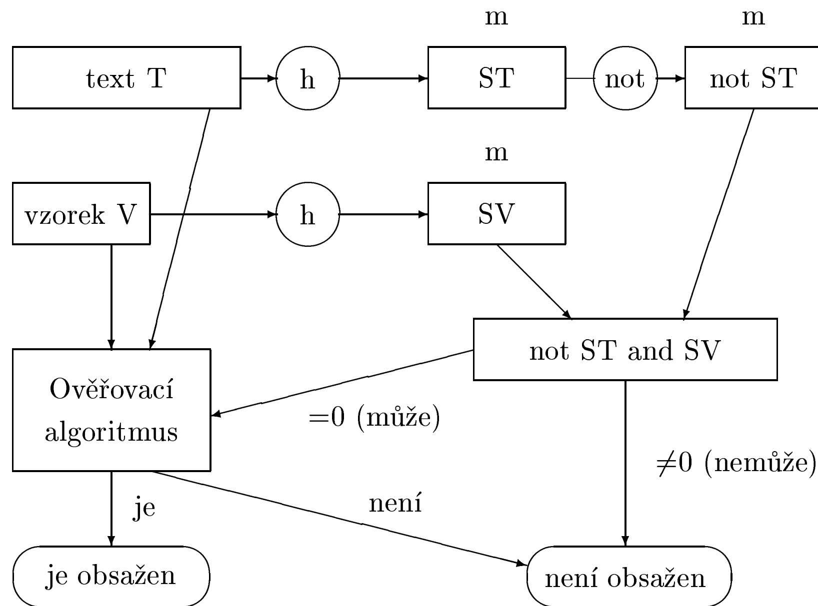
Příkladem, jak volit hašovací funkce, je použití hašovaných *k-signatur*. Všechny podřetězce délky k $(k - gramy)$ se použijí jako argumenty hašovací funkce *hash*. Její hodnotou je číslo v intervalu $< 1, m >$, které udává pořadí příznaku v signatuře. Tento příznak (bit) je v signatuře nastaven na jedničku. Názorně je tento výpočet naznačen na obrázku 4.22.

V případě použití hašovacích funkcí pro výpočet signatur a vrstveného kódování je signatura dokumentu vlastně jeho komprimovaná verze, ale *se ztrátou informace*. To znamená, že při výběru se stává, že nastane tzv. falešný výběr (false drop), tj. výběr dokumentu, který neodpovídá dotazu. Přitom signatura dokumentu dotazu odpovídá signatuře dotazu. V předchozím příkladě dokument obsahující "data file storage" (1011) odpovídá dotazu "tape" (1010). Toto je důvod k tomu, aby vybrané dokumenty byly dále prozkoumány přesným ověřovacím algoritmem.

Na obr. 4.23 je naznačen způsob použití metody vyhledávání vzorku pomocí signatury.



Obr. 4.22: Výpočet signatury



Obr. 4.23: Vyhledávání vzorku pomocí signatury

4.4.3 Vyhledávání vzorku pomocí signatury a indexu

Výsledkem porovnání signatur je zjištění, zda text může vzorek obsahovat, ale není určeno kde. Je tedy možné zkombinovat signaturu s indexem, tj. připojit k signatuře seznamy ukazatelů do textového řetězce. Příznakem v signatuře už pak není jediný bit, ale ukazatel na začátek řetězce ukazatelů nebo nic, když tento řetězec neexistuje. Nevýhodou této metody je značná paměťová složitost a přitom i tato metoda má časovou složitost $\mathcal{O}(p*t)$, kde p je počet vzorků a t je počet textů.

4.5 Vyhledávání vzorků pomocí dvojitého slovníku

Metoda vyhledávání vzorků pomocí dvojitého slovníku je běžně používaná metoda při porovnávání dvou uspořádaných množin. V našem případě se jedná o množiny slov. Nejdříve je třeba uspořádat množinu vzorků a vytvořit uspořádaný index slov, která se vyskytují v textu. Podstata vyhledávacího algoritmu pak spočívá v porovnávání dvou uspořádaných množin slov. Počet porovnání není větší než součet počtu slov v obou seznamech. Vyhledávání je tedy lineární se složitostí $0(n+m)$, kde n je počet slov v indexu a m je počet vzorků. Jestliže $n << m$, pak je výhodnější hledat v indexu metodou binárního půlení pro každý vzorek zvlášť a celková složitost je $0(m * log\ n)$, čímž dostaneme vlastně metodu vyhledávání v indexu. Z toho plyne,

že metoda vyhledávání vzorků pomocí dvojitého slovníku je výhodná především tehdy, když množina vzorků má mohutnost srovnatelnou s indexem.

Následující tabulka shrnuje základní vlastnosti probraných algoritmů na vyhledávání v textu.

předzpracování textu	předzpracování vzorků	počet vzorků	algoritmus
ne	ne	1	elementární
	ano	1	Knuth-Morris-Pratt
		1	Boyer-Moore
		konečný	Aho-Corasicková
		konečný	Commentz-Walterová
		nekonečný	konečný automat
ano	ne	konečný	slovní index
	ano	1	signatura
			signatura s indexem
		konečný	fragmentový index
			dvojitý slovník

Přehled metod vyhledávání vzorků v textu

5 Jazyky pro vyhledávání

Jen zřídka se vyskytuje situace, kdy vyhledáváme dokumenty, které obsahují jedno klíčové slovo. Výjimečně může jedno klíčové slovo stačit jako dotaz pro vyhledávací systém, ale jen v případě, že toto slovo je nové nebo zvlášť jedinečné. Příkladem takových slov mohou být specializované odborné termíny jako například „tezaurus".

Pokud například v informačním systému o počítačích se dotážeme na „computation", dostaneme jako odpověď téměř celou databázi. Tato odpověď má přibližně stejnou cenu jako žádná odpověď. Obvykle se uživatel při přípravě dotazu pro vyhledávání snaží omezit počet vyhledávaných dokumentů tak, aby jich nebylo příliš mnoho a aby získal všechny potřebné informace. Toho se dosahuje tím, že dotaz bývá složitější a kromě klíčových slov obsahuje operátory. Operátory v dotazech můžeme rozdělit na booleovské a poziční.

Booleovské operátory běžně používané jsou **and**, **or**, **xor** a **not**. Jejich interpretace je ovšem poněkud jiná, než v logice. Význam těchto operátorů je uveden v následující tabulce.

výraz	význam
A **and** B	vyber dokumenty, ve kterých se vyskytuje jak slovo A, tak i slovo B
A **or** B	vyber dokumenty, ve kterých se vyskytuje slovo A nebo slovo B nebo obě.
A **xor** B	vyber dokumenty, ve kterých se vyskytuje slovo A nebo B, ale ne obě.
A **not** B	vyber dokumenty, ve kterých se vyskytuje slovo A a nevyskytuje slovo B.

Z této tabulky je zřejmé, že operátory **and**, **or** a **not** budou implementovány pomocí množinových operací průnik, sjednocení a množinového rozdílu. Všimněme si, že operátor **not** je binární. Ve složitějších výrazech je nutno definovat prioritu operátorů. V tomto směru se jednotlivé systémy velmi liší. Ve většině systémů je možno používat závorky. Pokud nejsou závorky uvedeny, jsou používány tyto definice pořadí prováděných operací:

a) zleva doprava,

b) zprava doleva,

c) **and**, (**or**, **xor**), **not**,

d) (**or**, **xor**), **and**, **not**.

Priorita operátorů **or** a **xor** bývá stejná.

Protože operátor **not** není komutativní, je třeba dále definovat pořadí, v jakém se provádí stejné operátory. Obvykle bývá použita konvence levé asociativity, tj. operace se provádí zleva doprava.

Druhou skupinou operátorů jsou poziční operátory, které předepisují, v jakém pořadí musí být klíčová slova v hledaném dokumentu. Základní poziční operátory jsou **adj** (adjacent), **(n)words**, **with**, **same**, **syn**. Význam těchto operátorů je uveden v následující tabulce. Operátory **sentence** a **paragraf** jsou ekvivalentní s operátory **with** a **same**.

výraz	význam
A **adj** B	vyber dokumenty, ve kterých se vyskytuje slovo A následované slovem B
A **(n)words** B	vyber dokumenty, ve kterých se vyskytuje slovo A a nejdále o n slov za ním je slovo B
A **with** B A **sentence** B	vyber dokumenty, ve kterých se vyskytuje slovo A a B ve stejné větě
A **same** B A **paragraph** B	vyber dokumenty, ve kterých se vyskytuje slovo A a B ve stejném odstavci
A **syn** B	definuje, že slovo A a B jsou synonyma

Pokud se týče priority pozičních operátorů, platí o nich podobně jako o booleovských operátorech, že v různých systémech jsou používány různé typy priorit. Ve většině případů však mají poziční operátory vyšší prioritu než booleovské operátory. Mezi uvedenými pozičními operátory se používají priority v tomto pořadí: **adj**, **(n)words**, **syn**, **with**, **same**.

Dalším rysem vyhledávacích jazyků je možnost popisu množiny klíčových slov zkráceným způsobem. Například zápisem PSYCH$*$ jsme popsali množinu slov, která obsahuje např. slova:

PSYCHIATRIST

PSYCHIATRY

PSYCHOLOGICAL

PSYCHOLOGY

PSYCHOLOGIST

Hvězdička se používá jako zástupce libovolného řetězce znaků. Dále se používá otazník jako zástupce jednoho znaku, který může, nebo nemusí být přítomen. Např. DOCUMENT?? popisuje množinu, která obsahuje slova:

DOCUMENT

DOCUMENTS

DOCUMENTED

ale neobsahuje např. slovo DOCUMENTATION. Symboly $*$ a **?** mohou být použity na začátku, uvnitř a na konci slova. To je významné zejména u ohebných jazyků, kde je možná změna kmenových hlásek při skloňování a časování, což jsou například v češtině dosti časté jevy.

U systémů, které pracují s tezaurem, je možno použít při vyhledávání tezauru a tím se dostáváme k možnosti vyhledávání s použitím tématu (concept, topic).

V těchto systémech se používají operace uvedené v následující tabulce.

BT (A) BROADER TERM	vybere z tezauru širší termín k termínu A
NT (A) NARROWER TERMS	vybere z tezauru užší termíny k termínu A
PT (A) PREFERRED TERM	vybere z tezauru preferovaný termín k termínu A
SYN (A) SYNONYMS	vybere z tezauru všechna synonyma k termínu A
RT (A) RELATED TERMS	vybere z tezauru všechny příbuzné termíny k termínu A
TT (A) TOP TERM	vybere z tezauru nejširší termín

U operací BT, NT je možno ještě specifikovat, o kolik úrovní výše nebo níže se vybírají termíny z tezauru.

Jako příklad použití uvedených vlastností jazyků pro vyhledávání v textových systémech uveďme rozšíření jazyka SQL pro vyhledávání v textu tak, jak je definován v systému ORACLE SQL*TextRetrieval.

Toto rozšíření se týká především příkazu SELECT. Zde uvedeme rozšíření příkazu SELECT ve zjednodušené podobě. Podrobný popis je uveden ve firemní dokumentaci. Zjednodušený příkaz SELECT pro vyhledávání v textu má tvar:

```
SELECT specifikace položek
  FROM specifikace tabulek
    WHERE položka
      CONTAINS textový výraz
```

Například:

```
SELECT TITLE
  FROM BOOK
    WHERE ABSTRACT
      CONTAINS 'text retrieval'
```

Textový výraz může obsahovat operátory AND, OR a NOT a tím, že NOT je obvyklý unární logický operátor. Dále může obsahovat kulaté závorky v obvyklém významu. Operandy, které může obsahovat textový výraz jsou shrnuty v následující tabulce.

operand	zápis v textovém výrazu
jednoduchý řetěz	'řetěz'
řetěz s pravým rozšířením	'řetěz'*
řetěz s levým rozšířením	*'řetěz'
řetěz s oboustranným rozšířením	*'řetěz'*
řetěz s libovolným znakem místo ?	'ře?ěz'
řetěz s libovolným podřetězem místo %	'ře%ěz'
poziční výraz, 'řetězb' může být m slov za 'řetěza' nebo 'řetěza' může být n slov za 'řetězb'	'řetěza' (m,n) 'řetězb'
fráze	'hledám tuto frázi'
širší termín o jednu úroveň	BT ('řetěz')
širší termíny o n úrovní	BT ('řetěz', n)
šírší termíny všech úrovní	BT ('řetěz', *)
užší termíny o jednu úroveň	NT ('řetěz')
užší termíny o n úrovní	NT ('řetěz', n)
užší termíny všech úrovní	NT ('řetěz', *)
synonyma	SYN ('řetěz')
preferovaný termín	PT ('řetěz')
příbuzné termíny	RT ('řetěz')
nejširší termín	TT ('řetěz')

Hodnotou správného textového výrazu je vždy neprázdná konečná množina slov.

Příklad 5.1:

```
SELECT JMÉNO
  FROM ZAMĚSTNANEC
  WHERE VZDĚLÁNÍ
    CONTAINS RT(UNIVERSITA)
  AND JAZYKY
    CONSTAINS 'ANGLIČTINA' AND 'NĚMČINA'
  AND PUBLIKACE
    CONTAINS 'KNIHA' OR NT('KNIHA',*)
```

Tento příkaz způsobí výběr jmen zaměstnanců, kteří mají vysokoškolské vzdělání, ovládají němčinu a angličtinu a napsali knihu, příručku, monografii, návod nebo učebnici, jestliže byl použit tezaurus z obr.4.21 a příbuzné termíny k termínu *universita* jsou označení vysokých škol.

6 Komprese dat

Při uchovávání značného množství dat se dříve nebo později narazí na fyzické omezení pamětového prostoru. Jednou z možností, jak tento problém řešit, je zvětšení pamětového prostoru. Zde se budeme zabývat druhou možností, kterou je lepší využití existujícího pamětového prostoru. Uvedeme metody, které jsou zařazovány do oblasti nazývané komprese dat.

A. Shannon již v padesátých letech poukázal na to, že entropie, tj. informace obsažená ve znaku v rámci textu, je podstatně menší, než by se očekávalo. Obecně řečeno, veškerá data obsahují nějakou nadbytečnou informaci. Cílem algoritmů pro kompresi dat je transformace dat do tvaru, který bude menší, ale ponese stále stejné informace.

V literatuře je tzv. Shannon-Fano metoda považována za nejstarší. Shanonn s Weaverem a Fano ji vyvinuli v roce 1949 nezávisle na sobě. Po třech letech publikoval Huffmann svou nejznámější práci, týkající se metody konstrukce minimálně redundantního kódu. Musíme si uvědomit, že cílem všech těchto snah nebyla ani tak komprese dat, ale spíše kódování informace pro komunikační účely. Není to ovšem nic nového. V roce 1838 Morse navrhl svůj velmi dobře známý kód pro vysílání zpráv elektrickým telegrafem. Podobně jako ve výše zmíněných kompresních metodách, znaky s velkou pravděpodobností výskytu mají kratší kódová slova, než ty, který se vyskytují méně často.

Na druhé straně problémy, které přináší komprese při zpracování dat se v jedné důležité věci liší od tradičních studií v teorii komunikací: není zde jednoznačně definovaný statistický informační zdroj, na jehož základě by se dal kód doladit. Mnohé dříve vyvinuté metody tedy pouze velmi těžko umožňují podrobnou analýzu, která by teprve zajistila skutečně dobrou kompresi. Moderní kompresní metody používají model dat, který obsahuje:

1. strukturu, která je vlastně sadou jednotek, které máme komprimovat, a kontextu, ve kterém se vyskytují,
2. parametry, které vyjadřují pravděpodobnosti výskytu jednotlivých jednotek.

Každý algoritmus, který komprimuje data, můžeme chápat tak, jako by pracoval ve dvou krocích: vytvoření modelu dat a vlastní kódování. Tyto dva aspekty komprese dat otevírají cestu k pochopení modelů, odrážející lépe strukturu dat, která se mají komprimovat.

6.1 Základní pojmy komprese dat

6.1.1 Kódování a dekódování

Abeceda A je konečná neprázdná množina symbolů. Nějaká posloupnost symbolů z A je *řetězec* (*slovo*, *zpráva*) nad A. Prázdná posloupnost se nazývá *prázdný řetězec* a budeme ji značit ε. Množina všech řetězců nad abecedou A bez prázdného řetězce se značí A^+.

Kód K je trojice $K = (S, C, f)$, kde S je konečná množina *zdrojových jednotek*, C je konečná množina *kódových jednotek*, f je zobrazení z S do C^+.

Zobrazení f přiřazuje každé zdrojové jednotce z S právě jedno *kódové slovo* (sekvenci kódových jednotek) z C^+. Zobrazení f musí být injektivní. To znamená, že nikdy nezobrazuje dvě různé zdrojové jednotky na stejná kódová slova. Zobrazení f může být rozšířeno pro konečnou posloupnost zdrojových jednotek (*zdrojovou zprávu*) takto:

$f(S_1 S_2 \cdots S_k) = f(S_1) f(S_2) \cdots f(S_k)$.

Pojem kód se někdy používá pro obor hodnot (kódových slov) zobrazení f. Řetěz $x \in C^+$ je *jednoznačně dekódovatelný* vzhledem k f, jestliže existuje maximálně jedna posloupnost $y \in S^+$ taková, že $f(y) = x$. Říkáme, že kód $K = (S, C, f)$ je jednoznačně dekódovatelný, jestliže jsou vzhledem k f jednoznačně dekódovatelné všechny řetězy v C^+. Kód K se nazývá *prefixový kód*, jestliže žádné kódové slovo není prefixem jiného. Prefixové kódy se používají velmi často, protože právě ty jednoznačně dekódují zprávu během jejího čtení zleva doprava. Blokový kód délky n je takový kód, při kterém všechna kódová slova mají délku n. Každý blokový kód je jednoznačně dekódovatelný. Každý blokový kód je prefixový. Prefixový kód nemusí být blokový.

Příklad 6.1:

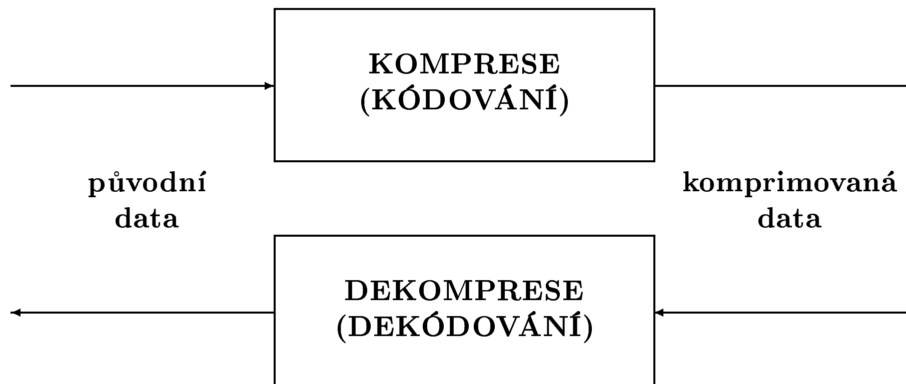
Chceme sestrojit binární prefixový kód pro desítkové číslice

$(A = \{1, 2, \ldots, 9\}, \; C = \{0, 1\})$

vhodný pro slova, kde se často vyskytují číslice 3 a 4, ale zřídka se vyskytují číslice 5 a 6. To znamená, že číslicím 3 a 4 chceme přiřadit kód co nejkratší. Pro jejich zakódování ale nemůžeme použít kódy 0 a 1, protože pro další číslice bychom nenalezli kódová slova tak, aby kód byl prefixový, protože každé binární slovo má prefix 0 nebo 1. Zvolíme tedy $f(3) = 00$, $f(4) = 01$. Potom žádné další kódové slovo nemůže začínat nulou. Pro další číslice musíme zvolit kódová slova začínající jedničkou. Pro číslice 5 a 6 zvolíme nejdelší kódová slova. Počet slov délky 3, která začínají jedničkou, je pouze 4. Nemůžeme tedy zakódovat číslice 0, 1, 2, 7, 8, 9 kódovými slovy délky 3. Zvolíme proto kódová slova o délce 4 a potom můžeme nalézt například tento kód:

a	0	1	2	3	4	5	6	7	8	9
$f(a)$	1000	1001	1010	00	01	1101	1110	1111	1011	1100

Jak je vidět na obr. 6.1, původní data jsou transformována na data koprimovaná (zakódovaná).



Obr. 6.1: Komprese a dekomprese dat

Tento proces označujeme jako *kódování*. Opačný proces, *dekódování*, je transformace, kdy komprimovaná data jsou dekódována zpět na původní data. Odpovídající algoritmy jsou nazývány *kodér* a *dekodér*. Stupeň redukce dat získaných jako výsledek kódování označujeme *kompresní poměr*. Tento poměr měří velikost komprimovaných dat vzhledem k velikosti původních dat.

$$\textit{Kompresní poměr} = \frac{\textit{Délka komprimovaných dat}}{\textit{Délka původních dat}}$$

Například kompresní technika, kde jeden znak komprimovaných dat odpovídá třem znakům původních dat, má kompresní poměr 0,33.

6.1.2 Entropie a redundance

Entropie je míra množství informace. Mějme zdrojové jednotky $S = \{x_1, x_2, \cdots, x_n\}$. Jestliže pravděpodobnost výskytu zdrojové jednotky x_i je p_i, $1 \leq i \leq n$, pak *entropie informačního obsahu jednotky* x_i je

$E_i = -log_2 p_i$ bitů.

To znamená, že zdrojové jednotky s větší pravděpodobností obsahují méně informace než jednotky s menší pravděpodobností. *Průměrná entropie zdrojové jednotky* z S je

$AE = \sum_{i=1}^{n} p_i E_i = -\sum_{i=1}^{n} p_i log_2 p_i$ bitů.

Entropie zdrojové zprávy $X = x_{i_1} x_{i_2} \cdots x_{i_k}$ z S^+ je pak

$E(X) = -\sum_{j=1}^{k} p_{i_j} log_2 p_{i_j}$ bitů.

Použijeme-li pro zakódování zdrojové jednotky $x_i, 1 \leq i \leq n$, kódové slovo $f(x_i)$ délky d_i bitů, pak *délka zakódované zprávy* X je $l(X) = \sum_{j=1}^{k} d_{i_j}$ bitů.

Z teorie informace plyne, že $l(X) \geq E(X)$. Rozdíl

$R(X) = l(X) - E(X) = \sum_{j=1}^{k} (d_{i_j} + p_{i_j} log_2 p_{i_j})$ bitů

je *redundance kódu* K pro *zprávu* X. *Průměrná délka kódového slova* K je

$AL(K) = \sum_{i=1}^{n} d_i p_i$ bitů.

Průměrná redundance kódu K je

$AR(K) = AL(K) - AE(S) = \sum_{i=1}^{n} p_i (d_i + log_2 p_i)$ bitů.

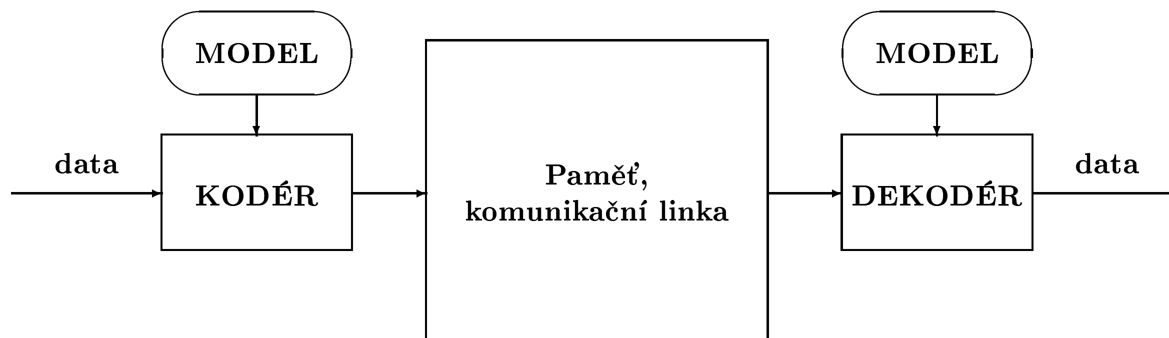
Kód K je *optimální*, když má minimální redundanci. Kód K je *asymptoticky optimální*, když platí, že pro dané rozložení pravděpodobností se poměr $AL(K)/AE(S)$ blíží k jedničce, když se entropie blíží k nekonečnu.

6.2 Predikce a modelování

Komprese dat je možná proto, že v datech je velké množství redundace způsobené:

- opakováním symbolů, skupin symbolů a slov,
- nestejnoměrnou pravděpodobností výskytu symbolů,
- závislostmi mezi sousedními symboly.

Použití této redundance znamená předpovědět, jaký bude následující symbol a použít tuto znalost pro kompresi. Predikce následujícího symbolu znamená odhadnutí pravděpodobnosti výskytu následujícího symbolu. Z tohoto důvodu je použit model dat obsahující rozložení pravděpodobností. Obr. 6.2 ukazuje, jak je model použit pro kódování a dekódování.



Obr. 6.2: Použití modelu pro kompresi dat

Kodér i dekodér musí mít stejný model. Existují tři způsoby, jak kodér a dekodér mohou pracovat se stejným modelem: statické, semiadaptivní a adaptivní modelování.

Statické modelování znamená, že kodér a dekodér používají týž model připravený předem, bez ohledu na data, která jsou komprimována. Protože statický model nebere v úvahu vstupní data, může být kompresní poměr daleko horší než optimální.

Semiadaptivní modelování je postup, který umožňuje vytvoření modelu, odpovídajícího charakteru komprimovaných dat. Vstupní data jsou nejdříve přečtěna a z nich je odvozen model. Potom je model uložen společně se zakódovanými daty. Nevýhodou tohoto postupu je skutečnost, že model musí být uchován s komprimovanými daty a může se stát, že celkové množství informace je větší než bez komprese. Navíc musí kodér mít všechna komprimovaná data k dispozici a musí provést dva průchody: vytvoření modelu a zakódování.

Adaptivní modelování znamená, že kodér odvozuje model ze vstupních dat a dekodér odvozuje týž model z komprimovaných dat. Takto vytvořený model dobře odpovídá komprimovaným datům, model není součástí komprimovaných dat a kodér provádí pouze jeden průchod.

Existuje několik typů modelů. Nejjednodušší model přiřazuje každé zdrojové jednotce určitou pravděpodobnost jejího výskytu bez ohledu na její pozici v datech. V tomto modelu pravděpodobnost *P(a)* je pravděpodobnost izolované zdrojové jednotky a. Tento model použil Morse pro známou Morseovu abecedu. V této abecedě jsou nejčastější písmena v angličtině *(e,t)* zakódována pomocí nejkratších kódových slov.

Lepší modely jsou založeny na myšlence, že pravděpodobnost zdrojové jednotky závisí na několika předchozích zdrojových jednotkách. Takové modely *s konečným kontextem* se nazývají Markovovy modely. Jestliže právě n předchozích zdrojových jednotek je použito pro určení pravděpodobnosti následující zdrojové jednotky, pak se jedná o model řádu n. Nejjednodušší model je řádu *0*, protože nepoužívá žádnou předchozí zdrojovou jednotku při určení pravděpodobnosti. Je to model zmíněný výše.

Modely založené na konečných automatech jsou mnohem obecnější než modely s konečným kontextem. Tyto modely předpokládají, že komprimovaná data jsou prvky regulárního jazyka. Speciálním případem konečných automatů jsou *automaty s konečným kontextem.* Vysvětleme, jakou vlastnost mají automaty s konečným kontextem.

Definice 6.1

Je dán konečný automat $M = (K, T, \delta, q_0, F)$. Množinu synchronizačních řetězů automatu M definujeme takto:

$$S(M) = \{x \in T^* : (q_i, x) \vdash^* (q_k, \varepsilon), (q_j, x) \vdash^* (q_l, \varepsilon), q_k = q_l \text{ pro všechny stavy } q_i, q_j \in K\}$$

Synchronizační řetězy mají tu vlastnost, že každý synchronizační řetěz donutí přejít automat z libovolného stavu do určeného stavu. Množina nesynchronizačních řetězů je pak definována takto:

$$N(M) = \{x \in T^* : (q_i, x) \vdash^* (q_k, \varepsilon), (q_j, x) \vdash^* (q_l, \varepsilon), q_k \neq q_l \text{ pro nějaká } q_i, q_j \in K\}.$$

Definice 6.2

Automat s konečným kontextem je konečný automat M, který má množinu nesynchronizačních řetězů $N(M)$ konečnou.

Pravděpodobnost zdrojové jednotky a v modelu s konečným kontextem řádu n je podmíněná pravděpodobnost $P(a|x_1x_2\dots x_n)$, která závisí na řetězu n předcházejících zdrojových jednotek $x_1, x_2, \dots, x_n$. Pravděpodobnost zdrojové jednotky v modelu založeném na konečném automatu je podmíněná pravděpodobnost $P(a|q_i)$, která závisí na stavu q_i, ve kterém se automat nachází.

V případě, že modelované řetězy patří do regulárního jazyka, jsou modely s konečným kontextem a modely používající konečné automaty velmi užitečné. V případě, že modelované řetězy patří do bezkontextového jazyka, nejsou tyto modely vhodné a je lépe použít bezkontextových

gramatik. Jedná se zejména o případy, kdy se v modelových řetězech vyskytují symetrické závorkové struktury.

Bezkontextová gramatika je čtveřice $G = (N, T, P, S)$, kde N je konečná množina neterminálních symbolů, T je konečná množina terminálních symbolů, P je konečná množina pravidel tvaru:

$$A \to \alpha,\ A\ \in N, \alpha \in (N \cup T)^*,$$

a $S \in N$ je startovací symbol. Derivace v gramatice G, označená $\Rightarrow$, je relace na řetězech definovaná takto:

$$x_1 A x_2 \Rightarrow x_1 \alpha x_2, \qquad \text{když } A\ \to \alpha \in P, x_1, x_2 \in (N \cup T)^*.$$

Tranzitivní a reflexivní uzávěr relace $\Rightarrow$ označíme $\Rightarrow^*$. Řetěz $x \in T^*$ je generován gramatikou G když $S \Rightarrow^* x$. Syntaktická analýza řetězu x generovaného gramatikou G je konstrukce derivace $x \in L(G)$.

6.3 Reprezentace celých čísel

Kód je univerzální, jestliže zdrojové jednotky zobrazí na kódová slova tak, že průměrná délka kódového slova je omezena hodnotou $c_1 * AE + c_2$, kde c_1 a c_2 jsou konstanty a AE je průměrná entropie zdrojové jednotky. Univerzální kód je asymptoticky optimální, když $c_1 = 1$. Nyní uvedeme některé univerzální kódy pro kódování celých čísel. Tyto kódy umožňují kódovat celé číslo x na posloupnost bitů S_x a přitom délka S_x je úměrná hodnotě čísla x. Tyto kódy jsou používány pro zakódování čísel vznikajících při kompresi pomocí slovníkových a syntaktických metod.

6.3.1 Fibonacciho kódy

Posloupnost Fibonacciho kódů je založena na Fibonacciho číslech řádu $m \geq 2$. *Fibonacciho čísla řádu* m jsou definována rekurzívně:

$$F_0 = F_{-1} = \ldots = F_{-m+1} = 1,$$
$$F_n = F_{n-m} + F_{n-m+1} + \ldots + F_{n-1} \quad \text{pro } n \geq 1.$$

Například Fibanocciho čísla řádu 2 jsou: 1,1,2,3,5,8,13,21,.... Každé nezáporné celé číslo N má binární reprezentaci tvaru:

$R(N) = \sum_{i=0}^{k} d_i F_i,$

kde $d_i \in \{0, 1\}, k\ \leq N$, a $F_i (0 \leq i \leq k)$ jsou Fibonacciho čísla řádu 2.
Zajímavou vlastností reprezentace *R(N)* je skutečnost, že neobsahuje dvě jedničky bezprostředně vedle sebe. Fibonacciho kód řádu 2 pro N je definován jako

$F(N) = d_0 d_1 d_2 \cdots d_k 1,$
kde $d_i, 0 \leq i \leq k$ je definováno výše.

To znamená, že Fibonacciho reprezentace je reverzovaná a je přidána jednička. Výsledná binární kódová slova tvoří prefixový kód, protože každé kódové slovo končí dvěma po sobě následujícími jedničkami, které se neobjeví nikde jinde v kódovém slově. Fibonacciho reprezentace $R(N)$ pro některá celá čísla jsou uvedeny na obr. 6.3. Dolní řádek obsahuje hodnotu F_i bitu na $i - té$ pozici v obvyklém uspořádání od vyšších řádů k nižším. Pravý sloupec obsahuje odpovídající Fibonacciho kódy $F(N)$.

Je dokázáno, že Fibonacciho kód řádu 2 je univerzální pro $c_1 = 2$ a $c_2 = 3$. Není asymptoticky optimální, protože $c_1 > 1$. Fibonacciho kódy vyšších řádů komprimují lépe než kód řádu 2, ale

N	$R(N)$	$F(N)$
1	1	1 1
2	1 0	0 1 1
3	1 0 0	0 0 1 1
4	1 0 1	1 0 1 1
5	1 0 0 0	0 0 0 1 1
6	1 0 0 1	1 0 0 1 1
7	1 0 1 0	0 1 0 1 1
8	1 0 0 0 0	0 0 0 0 1 1
16	1 0 0 1 0 0	0 0 1 0 0 1 1
32	1 0 1 0 1 0 0	0 0 1 0 1 0 1 1
	21 13 8 5 3 2 1	

Obr. 6.3: Příklad Fibonacciho reprezentace celých čísel

žádný Fibonacciho kód není asymptoticky optimální. Dále uvedeme Eliasovy kódy, které jsou asymptoticky optimální. Avšak Fibonacciho kódová slova jsou kratší ve velkém počátečním intervalu (až do N=514228). Tedy Fibonacciho kódy dávají lepší kompresi než Eliasovy kódy uvedené dále, dokud hodnota celých čísel není velká.

6.3.2 Eliasovy kódy

Elias definoval posloupnost kódů, které zobrazují celá čísla na binární kódová slova. Některé z nich jsou universální.

Nejjednodušší binární reprezentace celých čísel α se nazývá unární kód a je definován takto:

$$\alpha(N) = \underbrace{00\ldots0}_{N-1}\ 1$$

Standardní binární reprezentace β je definována induktivně takto

$$\begin{aligned} &\beta(1) = 1, \\ &\beta(N) = \beta(M)j, \text{ kde } N = 2M + j. \end{aligned}$$

Naneštěstí kód β není jednoznačně dekódovatelný, protože není prefixový. Například $\beta(5) = 101 = \beta(2)\beta(1)$ ukazuje tento problém. Abychom vytvořili kód β jednoznačně dekódovatelný, je třeba přidat znak *konec slova* (#). Pak dostaneme reprezentaci:

$$\tau(N) = \beta(N)\#.$$

Avšak reprezentace τ je ternární a ne binární kód. Existuje modifikace reprezentace τ označovaná τ', která vznikne následujícím postupem. Protože binární kód $\beta(N)$ vždy začíná jedničkou, můžeme ji vynechat a dostaneme $\beta^{'}$:

$$\begin{aligned} &\beta^{'}(1) = \varepsilon \text{ (prázdná posloupnost)} \\ &\beta^{'}(2N) = \beta^{'}(N)0, \\ &\beta^{'}(2N+1) = \beta^{'}(N)1. \end{aligned}$$

Pak $\tau^{'}(N) = \beta^{'}(N)\#$.

Eliasův kód γ je kompozice unárního a binárního kódu. Každý bit binárního kódu $\beta(N)$ je následován bitem unárního kódu délky $\beta(N)$. Protože první bit $\beta(N)$ je vždy jednička, je možno jej vynechat. Výsledkem je kód γ, ve kterém každý bit $\beta^{'}(N)$ je vložen mezi dvojici bitů z $\alpha(|\beta(N)|)$.

$$\begin{array}{lll} \gamma(1) = 1 & \gamma(3) = 0\bar{1}1 & \gamma(5) = 0\bar{0}0\bar{1}1 \\ \gamma(2) = 0\bar{0}1 & \gamma(4) = 0\bar{0}0\bar{0}1 & \gamma(6) = 0\bar{1}0\bar{0}1. \end{array}$$

Bity s pruhem jsou z $\beta^{'}(N)$ a ostatní bity jsou z $\alpha(|\beta(N)|)$. Množina $C_\gamma = \{\gamma(N) : N > 0\} = (0\{0,1\})^*1$ je regulární a proto může být dekódována konečným automatem.

Permutace bitů v $\gamma(N)$ dává kód $\gamma^{'}(N) = \alpha(|\beta(N)|)\beta^{'}(N)$ stejné délky, ale čitelnější:

$$\begin{array}{lll} \gamma^{'}(1) = 1 & \gamma^{'}(3) = 01\bar{1} & \gamma^{'}(5) = 001\overline{01} \\ \gamma^{'}(2) = 01\bar{0} & \gamma^{'}(4) = 001\overline{00} & \gamma^{'}(6) = 001\overline{10}. \end{array}$$

Protože poslední bit $\alpha(|\beta(N)|)$ je vždy jednička, může být použit jako první bit posloupnosti $\beta(N) = 1\beta^{'}(N)$. Kód γ zobrazuje tedy celé číslo N na binární kód, před kterým předchází $\lfloor log_2(N) \rfloor$ nul. Tento kód je jednoznačně dekódovatelný, protože celková délka kódového slova je o jednu větší než dvojnásobek počtu vedoucích nul. Množina

$C_{\gamma'} = \{0^k 1\{0,1\}^k : k \geq 0\}$

není regulární a dekodér potřebuje čítač.

Kódová slova pro kódy γ a γ' jsou dlouhá pro velká N. Délku $\beta(N)$ je možné reprezentovat jako $\gamma(|\beta(N)|)$ místo $\alpha(|\beta(N)|)$. Tento kód je kratší pro větší N. Nechť $\delta(N) = \gamma(|\beta(N)|)\beta^{'}(N)$.

$$\begin{array}{ll} \delta(1) = \gamma(1) = \bar{1} & \delta(3) = \gamma(2)1 = 00\overline{11} \\ \delta(2) = \gamma(2)0 = 00\overline{10} & \delta(4) = \gamma(3)00 = 01\overline{100}. \end{array}$$

Symboly s pruhem jsou $\beta(N)$. Dekódovací algoritmus pro $\delta(N)$ používá dekodér pro $\gamma(|\beta(N)|)$, aby vypočetl $|\beta(N)|$ a pak dává na výstup jedničku následovanou posledními $|\beta(N)| - 1$ symboly, t.j. $\beta(N)$.

Jiná representace je kód ω. Kódovací procedura má tvar:

$K := 0$;
while $\lfloor log_2(N) \rfloor > 0$ *do*
    begin $K := \beta(N)K$;
        $N := \lfloor log_2(N) \rfloor$
    end.

Příklady výsledků této procedury:

$$\begin{array}{lll} \omega(1) = 0 & \omega(4) = \overline{10}\ \overline{100}\ 0 & \omega(15) = \overline{11}\ \overline{1111}\ 0 \\ \omega(2) = \overline{10}\ 0 & \omega(7) = \overline{10}\ \overline{111}\ 0 & \omega(16) = \overline{10}\ \overline{100}\ \overline{10000}\ 0 \\ \omega(3) = \overline{11}\ 0 & \omega(8) = \overline{11}\ \overline{1000}\ 0 & \omega(32) = \overline{10}\ \overline{101}\ \overline{100000}\ 0. \end{array}$$

Pravá skupina číslic s pruhem je $\beta(N)$ s výjimkou případu, kdy $N = 1$. Každá předcházející skupina je binární kód délky následující skupiny zmenšené o jedničku. První skupina má délku 2. Dekodér je podobný jako pro kód δ. Kód ω' je variantou kódu ω s tím, že první skupina má délku 3.

6.4 Statistické metody komprese dat

Nejstarší statická metoda komprese dat byla navržena v roce 1949 Shannomem, Weaverem a Fanem. Vychází se zadaného statistického rozložení pravděpodobností zdrojových jednotek založeném na modelu řádu 0. Huffman v roce 1952 vyšel ze stejného předpokladu a jeho kód je nejznámější metodou prefixového kódování, která je minimální v tom smyslu, že dosahuje nejlepší kompresní poměr mezi prefixovými kódy při pevně zadaných pravděpodobnostech.

V tomto odstavci se budeme zabývat oběma metodami a ukážeme, jak Huffmanovou statickou metodu lze převést na adaptivní verzi (algoritmy FGK a V).

6.4.1 Shannon-Fanovo kódování

Algoritmus je založen na rekurzivní proceduře přidávající ke kódovému slovu v každém kroku jeden bit.

Algoritmus 6.1:

Shannon-Fanovo kódování
Vstup: Posloupnost n zdrojových jednotek $S[i]$, $1 \leq i \leq n$.
Výstup: n binárních kodových slov.

begin uspořádej zdrojové jednotky v pořadí neklesající pravděpodobnosti;
        přiřaď všem kódovým slovům prázdný řetěz;
        SF-SPLIT(S)
end
procedure SF-SPLIT(S);
begin if $|S| \geq 2$ **then**
        begin rozděl S do posloupností $S1$ a $S2$ tak, že obě posloupnosti
                mají přibližně stejnou celkovou pravděpodobnost;
                přidej ke všem kódovým slovům z $S1$ nulu;
                přidej ke všem kódovým slovům z $S2$ jedničku;
                SF-SPLIT($S1$);
                SF-SPLIT($S2$)
        end
end

Algoritmus končí a vytváří prefixový kód.

Příklad 6.2:

Nechť (1,1,1,1,3,3,3,3,7) je posloupnost vyjadřující frekvence zdrojových jednotek v textu nad abecedou o velikosti $n = 9$. Přiřazené pravděpodobnosti jsou

$(1/23, 1/23, 1/23, 1/23, 3/23, 3/23, 3/23, 3/23, 7/23)$.

Obrázek 6.4 ilustruje postupnou aplikaci procedury SF-SPLIT Algoritmu 6.1 a výsledné zobrazení f.

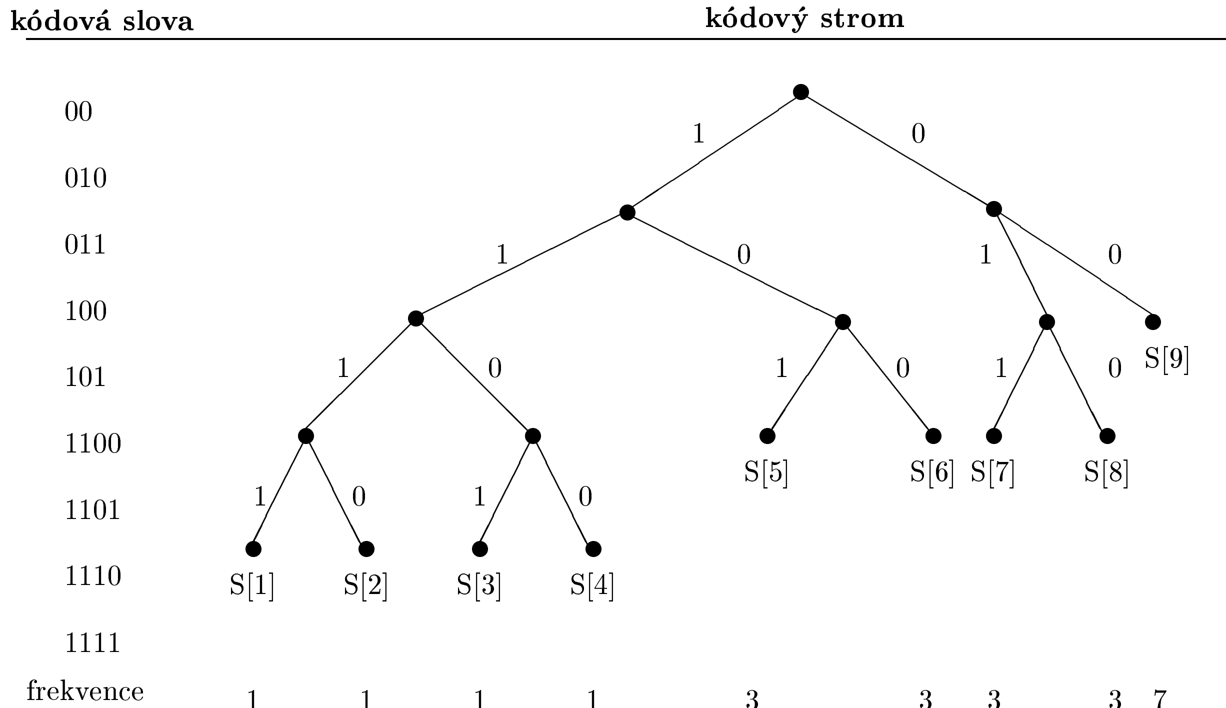
Každý prefixový kód můžeme reprezentovat binárním stromem. Tyto stromy budeme nazývat kódové stromy. Kódový strom odpovídající kódu z příkladu 6.2 je na obr. 6.5. Délka každého kódového slova v Shannon-Fanově kódování je buď $-log_2 p_i$ nebo $-log_2 p_i + 1$, t.j. Algoritmus 6.1 vytváří kódová slova splňující podmínku:

$AE \leq AL \leq AE + 1$.

Obecně Shannon-Fanovo kódování nezaručuje, že výsledný kód je optimální. Další nevýhodou je skutečnost, že algoritmus vyžaduje rozložení pravděpodobností zdrojových jednotek a proto kodér musí číst kódovaná data dvakrát.

zdrojová jednotka	pravděpodobnost	kódové slovo	krok
$S[1]$	1/23	1111	8
$S[2]$	1/23	1110	6
$S[3]$	1/23	1101	7
$S[4]$	1/23	1100	4
$S[5]$	3/23	101	5
$S[6]$	3/23	100	1
$S[7]$	3/23	011	3
$S[8]$	3/23	010	2
$S[9]$	7/23	00	

Obr. 6.4: Shannon-Fanovo kódování



Obr. 6.5: Shannon-Fanův kód a kódový strom

6.4.2 Huffmanovo kódování

Huffman navrhl v roce 1952 optimální kódování pomocí prefixového kódu. Navržený algoritmus je statický a vyžaduje pevně dané statistické charakteristiky výskytu jednotlivých zdrojových jednotek. Originální algoritmus vytváří kódový strom, nazývaný *Huffmanův strom.* Podobně jako Shannon-Fanův algoritmus vyžaduje i Huffmanův algoritmus dva průchody.

Nové varianty Huffmanova algoritmu jsou adaptivní, což umožňuje, aby se pravděpodobnosti výskytu zdrojových jednotek měnily při průchodu daty a odpovídajícím způsobem se měnil i počáteční Huffmanův strom. To znamená, že se provádí kódování současně s rekonstrukcí kódového stromu. Okamžitě je zřejmé, jak tato varianta zlepšuje původní algoritmus. Není třeba uchovávat kódový strom pro účely dekódování. Dekodér se „učí“ charakteristiky zdrojových dat při jejich dekódování. Je zřejmé, že jak kodér, tak dekodér musí začít se stejným počátečním stromem. Jednoprůchodová metoda může být efektivní, protože počet bitů nutných pro zakódování může být menší. Dále uvedeme dvě verze adaptivních Huffmanových algoritmů.

6.4.2.1 Statické Huffmanovo kódování

Nejdříve uvedeme Huffmanův originální algoritmus založený na konstrukci Huffmanova stromu.

Algoritmus 6.2:

Statické Huffmanovo kódování
Vstup: n zdrojových jednotek a posloupnost jejich pravděpodobností $p[i]$, $1 \leq i \leq n$,
Výstup: posloupnost n binárních kódových slov.

```
begin /* konstrukce Huffmanova stromu */
    vytvoř list o(p[i]) binárního stromu pro každou zdrojovou jednotku i;
    /* uzel o jednotky i je ohodnocen pravděpodobností p[i] */
    k := n;
    while k ≥ 2 do begin
        najdi dvě nejmenší nenulové pravděpodobnosti p[r] a p[s], kde r ≠ s;
        q := p[r] + p[s];
        vytvoř uzel o ohodnocený q a hrany [o(q), o(p[r])] a [o(q), o(p[s])]
        ohodnocené 0 a 1;
        p[r] := 0;
        p[s] := 0;
        k := k  − 1
    end
    /* konstrukce kódových slov */
    ohodnocení hran nulami a jedničkami na cestě z kořene do listu
    přiřaď zdrojové jednotce i, 1 ≤ i ≤ n.
end
```

Průměrná délka kódových slov AL získaných Huffmanovým algoritmem je stejná jako vážená délka cesty

$AEPL = \sum_{i=1}^{n} d[i] * p[i]$,

kde $d[i]$ je délka cesty z kořene do listu i. Je dokázáno, že $AEPL$ je minimální pro všechny možné binární stromy se zadanými listy. Huffmanův algoritmus tedy vytváří prefixový kód s minimální průměrnou délkou. Existují však optimální prefixové kódy, které nejsou Huffmanovy (viz příklad 6.4).

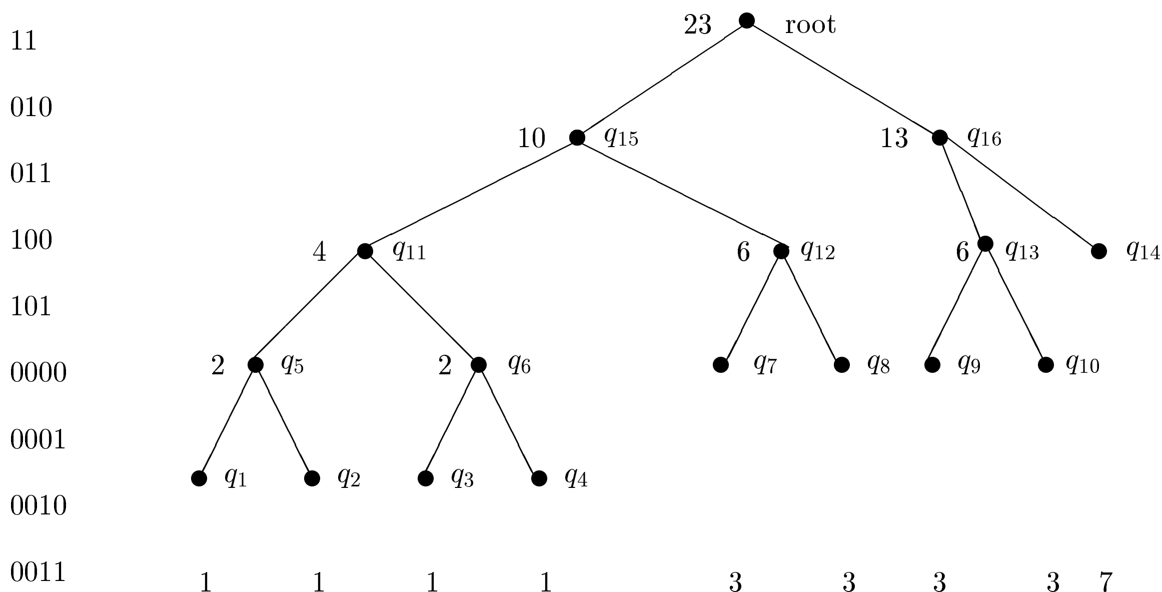
Uveďme některé poznámky:

- Není nutné normalizovat frekvence na pravděpodobnosti, tj. čísla z intervalu $< 0,1 >$. Stejné výsledky dostaneme, použijeme-li frekvence zdrojových jednotek.
- Ohodnocení hran nulami a jedničkami v pořadí uvedeném v algoritmu je jen jedno z možných. Obrácené pořadí vede též na optimální kód. Pouze délka kódových slov je podstatná.
- Jak Huffmanův, tak Shannon-Fanův kód může být generován v čase $\mathcal{O}(n)$ za předpokladu, že posloupnost zdrojových jednotek je uspořádaná podle jejich pravděpodobností.
- Počet uzlů v Huffmanově stromu je $2n - 1$.

Příklad 6.3:

Nechť posloupnost frekvencí (1,1,1,1,3,3,3,3,7) je stejná jako v příkladu 6.2. Algoritmus 6.2 vytváří Huffmanův strom a Huffmanův kód, který je uveden na obr. 6.6. Průměrná entropie zdrojové jednotky je 2,869 bitů a průměrná délka kódového slova je 2,840 bitů.

Hlavní problém statického Huffmanova kódování spočívá v tom, že je nutno znát rozložení frekvencí (pravděpodobností) zdrojových jednotek ve vstupních datech. Toto rozložení je často



Obr. 6.6: Huffmanův kód a kódový strom

známo a je pozoruhodně stabilní (např. v češtině nebo v angličtině). Jestliže jsme schopni získat takové rozložení pro dostatečně velké třídy dat, můžeme vytvořit tak zvané kódové tabulky s několika zobrazeními. V takovém případě je možné vytvořit Huffmanovy kódy předem a ušetřit čas na jejich konstrukci.

6.4.2.2 Vlastnosti Huffmanových stromů

Huffmanovy stromy mají některé zajímavé vlastnosti. Existuje-li více možností pro výběr $p[r]$ a $p[s]$, je možno vybrat libovolnou z nich. To znamená, že Huffmanův kód může být reprezentován několika binárními stromy. Maximální počet takových stromů je $2n-1$. Přitom však jsou některé kódy jen permutacemi jiných kódů a proto je maximální počet různých Huffmanových stromů menší.

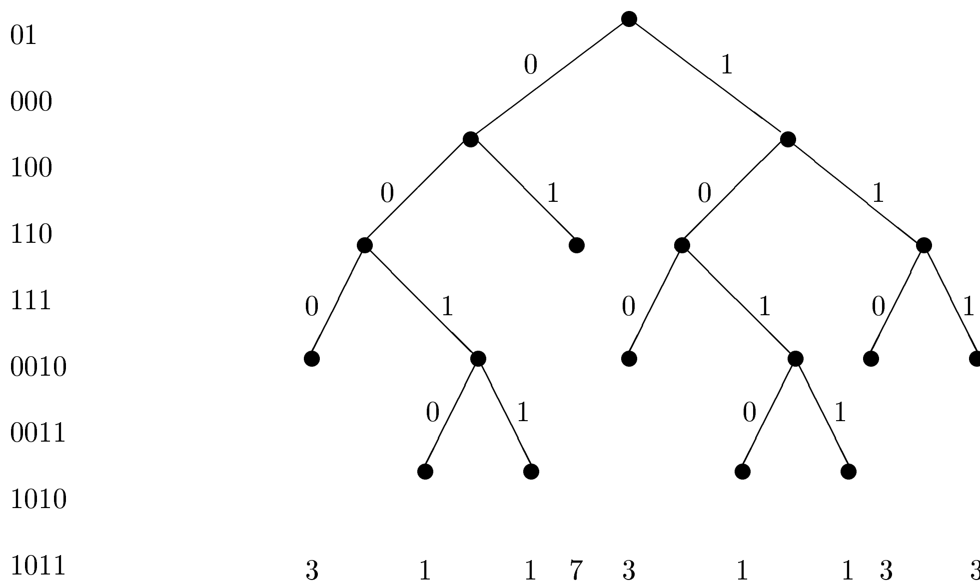
Významnou vlastností Huffmanových stromů je tzv. *sourozenecká vlastnost*. Binární strom má sourozeneckou vlastnost, když

1. každý uzel kromě kořene má sourozence,
2. uzly mohou být seřazeny v pořadí neklesající pravděpodobnosti tak, že každý uzel sousedící v seznamu s nějakým uzlem je jeho sourozenec (leví synové budou na lichých místech v seznamu a praví synové na sudých místech).

Například strom na obr. 6.6 má uspořádání dané posloupností uzlů $q_1, q_2, \ldots, q_{16}$. Je dokázáno, že binární prefixový kód je Huffmanův kód právě tehdy, když jeho kódový strom má sourozeneckou vlastnost. Uvedené uspořádání použijeme dále v algoritmu pro adaptivní Huffmanovo kódování.

Huffmanův algoritmus není jedinou možností konstrukce optimálního kódu. Libovolný prefixový kód, který má stejné délky kódových slov, je stejně dobrý jako Huffmanův kód. Pro daný seznam frekvencí posloupnost $<n_1, \ldots, n_l>$, kde n_i je počet kódových slov délky i, označuje třídu kódů se stejnou délkou jako Huffmanovy kódy. Uveďme optimální kód, který není Huffmanův.

kódová slova **kódový strom**



Obr. 6.7: Prefixový kód, který není Huffmanův

Příklad 6.4:

Uvažujme opět seznam frekvencí (1,1,1,1,3,3,3,3,7) z příkladu 6.2. Jeden z možných prefixových kódů a jemu odpovídající kódový strom je na obr. 6.7. Tento kód není Huffmanův kód, protože listy kódového stromu s frekvencí 1 nejsou ve společném podstromu. Tato skutečnost odporuje sourozenecké vlastnosti. Kód je však optimální a patří do třídy $< 0, 1, 4, 4 >$, což je táž třída, do které patří kód z příkladu 6.2. Shannon-Fanův kód na obr. 6.5 je také Huffmanův kód.

Další zajímavou skutečností je odhad horní meze redundance Huffmanova kódu. Nechť X je zpráva a p_n je nejvyšší pravděpodobnost zdrojové jednotky ve zprávě X. Pak pro redundanci $R(X)$ Huffmanova kódu platí

$R(X) \leq p_n + 0,086.$

Někdy je účelné vyžadovat další vlastnosti Huffmanových kódů. Jednou z nich je vlastnost býti *sufixovým kódem*, což dovoluje dekódování zprávy opačným směrem. Prefixové kódy mající tuto vlastnost se nazývají *afixové kódy.*

Je zřejmé, že každý kód s pevnou délkou kódového slova je afixový. Tyto kódy jsou triviální afixové kódy. Netriviální afixový kód je uveden v příkladu 6.4. Je dokázáno, že existuje neomezený počet netriviálních afixových kódů. Například, když rozšíříme strom na obr. 6.7. tak, že přidáme dva nové syny k listu a ohodnotíme hrany nulou a jedničkou, dostaneme opět kódový strom afixového kódu. Třídy kódů obsahující kódová slova o délce 1 neobsahují afixové kódy.

Afixové kódy mají řadu praktických aplikací

- KWIC index.
  Indexování textu v textovém vyhledávacím systému se provádí pro každé slovo K. Nadto je žádoucí, aby levý a pravý kontext slova K mohl být vyhledáván společně se slovem K. V případě komprese textu je požadováno, aby se zpracovával zakódovaný text od slova K pozpátku.
- vyhledávání vzorku $*X*$.
  Zpracování vzorků tohoto typu se při vyhledávání často vyžaduje.

Další vlastnost kódu je jeho *úplnost.* Kód je úplný, když po přidání libovolného dalšího kódového slova vznikne kód, který není jednoznačně dekódovatelný. Huffmanovy kódy jsou

úplné. Úplné kódy jsou nevýhodné v situaci, kdy dojde při přenosu k chybě. Když se jeden bit ztratí, pak nejsme schopni v mnoha případech se zotavit z chyby při omezeném zpoždění. Při použití úplného afixového kódu je možné zpracovat zakódovanou zprávu ve zpětném směru a získat alespoň částečně uspokojivý výsledek.

6.4.2.3 Adaptivní Huffmanovo kódování

Nevýhodou statického Huffmanova kódování je nutnost ukládání kódového stromu společně s komprimovanými daty. Různé modifikace vedly na dvě významné adaptivní verze Huffmanova kódování nazývané FGK a V algoritmy. Čas na kódování a dekódování zdrojové jednotky je u těchto algoritmů úměrný délce kódového slova. Proto může kódování a dekódování probíhat v reálném čase. V obou uvedených algoritmech se předpokládá, že není na začátku kódovacího procesu nic známo o pravděpodobnostech zdrojových jednotek. Základní myšlenka je založena na možnosti adaptace kódového stromu podle změn pravděpodobností během kódování. To znamená, že *k*-tá zdrojová jednotka je zakódována podle informací o *k-1* předchozích zdrojových jednotkách.

Algoritmus FGK byl navržen Fallerem, Gallagerem a Knuthem. Odhadování pravděpodobností je nahrazeno počítáním frekvencí zdrojových jednotek. Algoritmus používá čítače frekvencí a inkrementuje je o hodnotu INCR (obvykle INCR = 1). Tyto hodnoty se používají pro reorganizaci Huffmanova stromu.

Změny ve stromu způsobené zvětšením frekvence v listu se dějí ve dvou fázích:

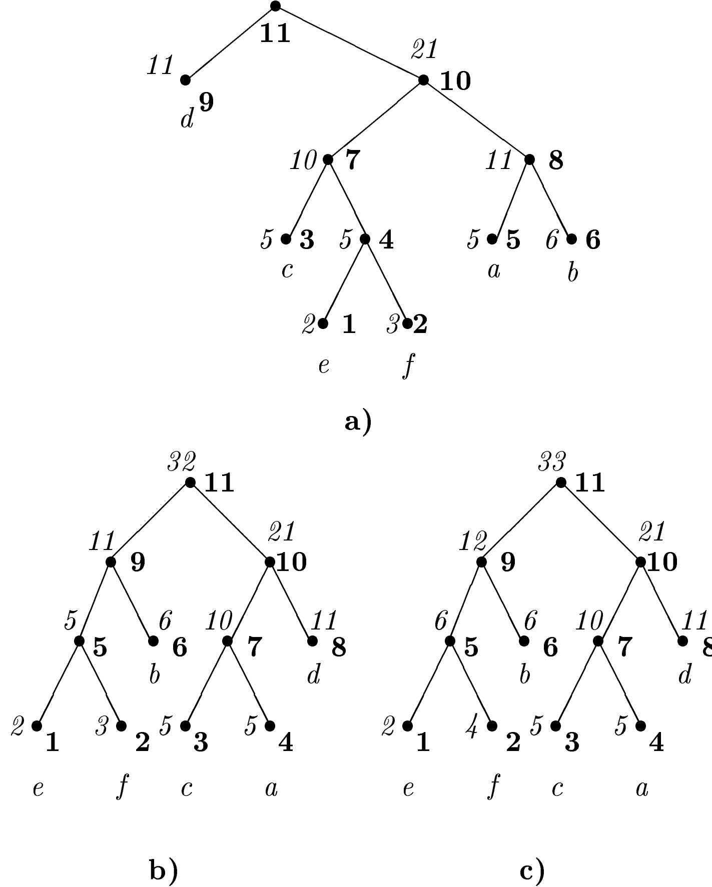
1. Nechť T_1 je daný Huffmanův strom. Postupně, počínaje od listu, který odpovídá právě zakódované zdrojové jednotce, až po kořen stromu, vyměníme podstromy tak, aby každý uzel o, kterým procházíme, byl v uspořádání uzlů se stejnou frekvencí na posledním místě. Jestliže takové uzly jsou $o_k, o_{k+1}, \ldots, o_{k+m}$, pak $o = o_{k+m}$. Výsledný strom označíme T_2.
2. Zvětšíme frekvence v jednotlivých uzlech od zpracovaného listu ke kořenu o hodnotu INCR.

Vzhledem ke způsobu konstrukce stromu T_2 je zachována sourozenecká vlastnost stromu na konci 2. fáze. Složitější 1. fázi zapíšeme ve formě algoritmu (*predcessor(x)* je rodič uzlu x).

Algoritmus 6.3:

Adaptivní Huffmanovo kódování (jádro FGK)
Vstup: Daný Huffmanův strom $T1$.
Výstup: Huffmanův strom $T2$.

```
begin /* právě zpracovávaný uzel je v proměnné current */
      current := o;
      while current ≠ root do
            begin
                  vyměň uzel v current (včetně odpovídajícího podstromu)
                  s uzlem o', který má nejvyšší pořadí mezi uzly se
                  stejnou frekvencí; /* current obsahuje o' */
                  current := predcessor(current)
            end
end
```



Obr. 6.8: Adaptování Huffmanova stromu

Příklad 6.5:

Je dána množina zdrojových jednotek $\{a,b,c,d,e,f\}$. Huffmanův strom T_1 je na obrázku 6.8 a) a byl vytvořen FGK algoritmem (čísla označují frekvence, tučná čísla označují uspořádání).

Následující kódovaná jednotka je f. f bude zakódováno pomocí stromu T_1 jako 1011. Po provedení 1. fáze bude výsledný strom T_2 podle obr. 6.8 b). 2. fáze způsobí zvětšení frekvencí podle obr. 6.8 c) (INCR = 1). Při dalším výskytu bude jednotka f zakódovaná jako 001.

Poznamenejme, že na začátku kódování dokonce není známo, jaké zdrojové jednotky budeme kódovat. Tento problém je možno řešit tak, že zavedeme zvláštní uzel s frekvencí *0* nazývaný *0-uzel* pro reprezentaci dosud nepoužitých jednotek. Pokud se objeví nová jednotka, *0-uzel* se rozdělí na nový *0-uzel* a list pro novou jednotku. Starý *0-uzel* se stane rodičem obou nově vzniklých uzlů a jeho frekvence bude 1. Ostatní uzly v T_2 jsou přečíslovány.

Při kompresi „nehomogenních“ dat není toto schema příliš vhodné, protože při kolísající frekvenci zdrojových jednotek se příliš využívá „starých“ informací o jejich frekvenci. Tento problém má však jednoduché řešení: Vynásobit všechny čítače koeficientem $r < 1$ po zpracování N jednotek. Tento přístup dovoluje využít lokálních vlastností kódovaných dat.

Příklad 6.6:

Jsou dány zdrojové jednotky $\{a,b,c\}$ a hodnoty odpovídajících čítačů 10, 20, 30. Nechť $N = 30$ a po zpracování N jednotek budou mít čítače hodnoty $10, 40, 40$. Jestliže vynásobíme tyto hodnoty čítačů číslem $r = 0.2$, pak budou mít čítače hodnoty $2, 8, 8$. Po dalších N krocích budou mít v prvním případě hodnoty $30, 50, 40$, ve druhém případě $22, 18, 8$ (tj. inkrementy jsou $20, 10, 0$). Odpovídající Huffmanovy kódy jsou:

	1.případ	2.případ
a	10	0
b	0	11
c	11	10

Je vidět, že ve druhém případě při větším počtu výskytu jednotek a pro tuto jednotku dostaneme kratší kódové slovo než při použití globálních frekvencí.

Metoda násobení frekvencí koeficientem má jednu nevýhodu. Vzhledem k nutnosti zaokrouhlování po násobení je třeba provést reorganizaci kódového stromu. Tuto nevýhodu je možno obejít tím, že hodnota INCR se bude postupně zvětšovat podle posloupnosti $1, r, r^2, \ldots r^n$, kde $r > 1$. V okamžiku, kdy nastane přeplnění některého z čítačů, provede se dělení hodnotou r^n.

Je dokázáno, že FGK algoritmus není nikdy horší než dvakrát optimální. Když AL_{HS} a AL_{HD} jsou po řadě průměrné délky kódových slov pro statické a dynamické Huffmanovo kódování, pak

$$AL_{HD} \leq 2AL_{HS}.$$

Knuth dokázal, že čas potřebný jak pro kódování, tak pro dekódování, je $\mathcal{O}(d)$, kde d je délka kódového slova.

Vitter navrhl úpravu FGK algoritmu, která je známa jako V algoritmus. Tento algoritmus má tyto vlastnosti:

(1) Počet výměn v 1. fázi algoritmu je nejvýše 1.

(2) Minimalizuje nejen

$$\sum_{i=1}^{n} d[i] * p[i], \text{ ale i } \sum_{i=1}^{n} d[i] \text{ a } max_i d[i].$$

(3) $AL_{HD} \leq AL_{HS} + 1$, a to je optimum mezi všemi dynamickými Huffmanovými metodami.

6.4.2.4 Implementační poznámky

Reorganizace stromu vyžaduje vhodnou datovou strukturu pro jednoduchou implementaci. Jedna z možností je zavedení ukazatelů na rodiče ze všech uzlů kromě kořene. Uspořádání dané sourozeneckou vlastností vyžaduje obousměrně zřetězený seznam, kde každý uzel obsahuje dopředný a zpětný ukazatel. Přístup k uzlům je možný například pomocí indexové tabulky.

Tato přímá reprezentace Huffmanova stromu vede na velmi složitou strukturu. Existuje daleko jednodušší reprezentace pomocí tabulky, která dovoluje přímé dekódování kódového slova bit po bitu jedním průchodem v čase $\mathcal{O}(n)$.

Příklad 6.7:

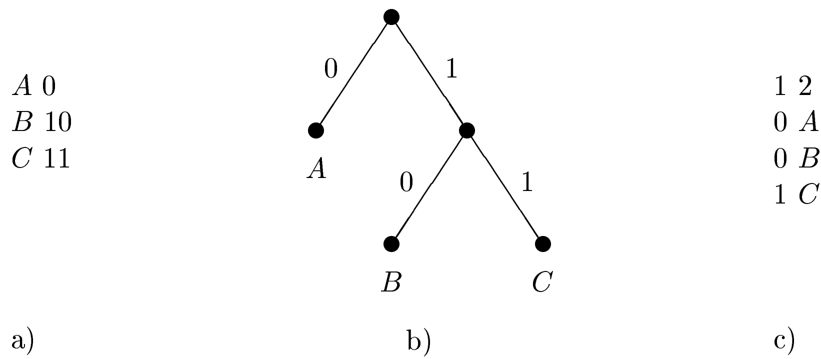
Předpokládejme kódovou tabulku na obr. 6.9 a) a kódový strom na obr. 6.9 b). Odpovídající dekódovací tabulka je na obr. 6.9 c).

Pro každý uzel Huffmanova stromu kromě kořene je v tabulce jeden řádek. Tabulka má proto $2(n-1)$ řádků. První sloupec každého řádku obsahuje bit kódového slova. Ve druhém sloupci je pro koncové uzly stromu uvedena zdrojová jednotka. Pro vnitřní uzly je ve druhém sloupci uvedeno číslo, které udává, o kolik řádků níže je řádek pro následující uzel.

Huffmanovo kódování je možno použít také pro zdrojové jednotky, kterými jsou slova nad nějakou abecedou. Uvedeme metodu, která je rozšířením výše uvedeného postupu. Kódový strom je reprezentován prefixovým zápisem ve tvaru:

(kořen, levý podstrom v prefixovém zápisu, pravý podstrom v prefixovém zápisu).

Každý uzel stromu obsahuje jedničku označující uzel jako list nebo nulu pro vnitřní uzel.



Obr. 6.9: Stromová reprezentace kódové tabulky

Každý vnitřní uzel obsahuje adresu pravého podstromu. Celkem potřebujeme

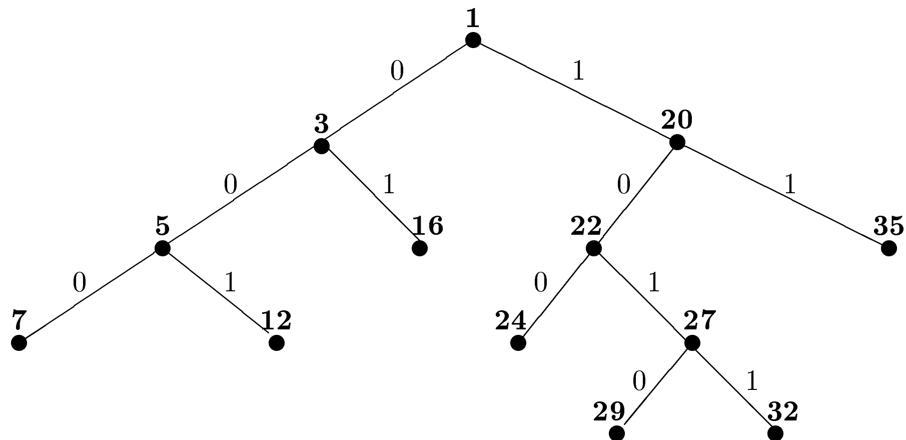
$nC + (n-1)A$

bytů a paměť pro n kódovaných slov. Typicky C=1 byte na délku slova a A=2 byty na adresu.

Příklad 6.8:

Je dán slovník podle obr. 6.10.

kódový strom:



zdroj. jednotka:	near	big	the	on	at	in	over
kódové slovo:	000	001	01	100	1010	1011	11

Obr. 6.10: Huffmanovo kódování slov

Strom v prefixovém zápisu má tvar:

adresa:	1	3	5	7	12	16	
obsah:	(0,20)	(0,16)	(0,12)	(1,4,near)	(1,3,big)	(1,3,the)	
adresa:	20	22	24	27	29	32	35
obsah:	(0,35)	(0,27)	(1,2,on)	(0,32)	(1,2,at)	(1,2,in)	(1,4,over)

Kódová slova nejsou v této reprezentaci obsažena. Algoritmus 6.4 představuje dekodér pro prefixovou reprezentaci.

Algoritmus 6.4:

Dekodér pro prefixovou reprezentaci dekódovací tabulky
Vstup: Zakódovaná data po předchozím dekódovaném kódovém slovu.
Výstup: Dekódované slovo.

/* předpokládejme, že každý uzel je reprezentován záznamem $P[i]$ s položkami *flag* a *address* a i je jeho adresa, A je počet bytů potřebných pro uložení adresy*/

```
begin k := 1;
    repeat receive bit;
        if bit = 0 then k := k + A else k := P[k] . address;
        flag . value := P[k] . flag;
    until flag . value = 1;
    Přidej obsah adres k, ..., k + P[k] . length – 1 k dekódovanému výstupu
end
```

6.4.3 Aritmetické kódování

Huffmanovo kódování je speciálním případem aritmetického kódování. Huffmanovo kódování dává výsledky rovné entropii $E(X)$, jestliže pravděpodobnosti zdrojových jednotek jsou záporné mocniny dvou. V opačném případě Huffmanovo kódování je daleko od optima a obsahuje určitou redundanci. Když rozdělení pravděpodobností je velmi nepříznivé, pak průměrná délka kódu generovaného Huffmanovým kódováním může být mnohem větší než optimum dané entropií $E(X)$.

Při aritmetickém kódování jsou kódovaná data reprezentována intervalem $< 0,1)$ na ose reálných čísel. Každá zdrojová jednotka zužuje tento interval. Jak se interval zmenšuje, tak se počet bitů potřebných pro jeho určení zvětšuje.

Vysoce pravděpodobné jednotky zmenšují tento interval méně než málo pravděpodobné jednotky. To znamená, že vysoce pravděpodobné jednotky přispívají méně bity do kódového řetězu. Tato metoda používá seznam zdrojových jednotek a jejich pravděpodobností. Číselná osa je rozdělena na subintervaly na základě kumulativních pravděpodobností. Kumulativní pravděpodobnost je definována tímto způsobem:

Definice 6.3

Nechť $a_1, a_2, \dots a_n$ je uspořádaná posloupnost zdrojových jednotek s pravděpodobností $p_1, p_2, \dots p_n$. Kumulativní pravděpodobnost cp_i zdrojové jednotky a_i je součet:

$$cp_i = \sum_{j=1}^{i-1} p_j.$$

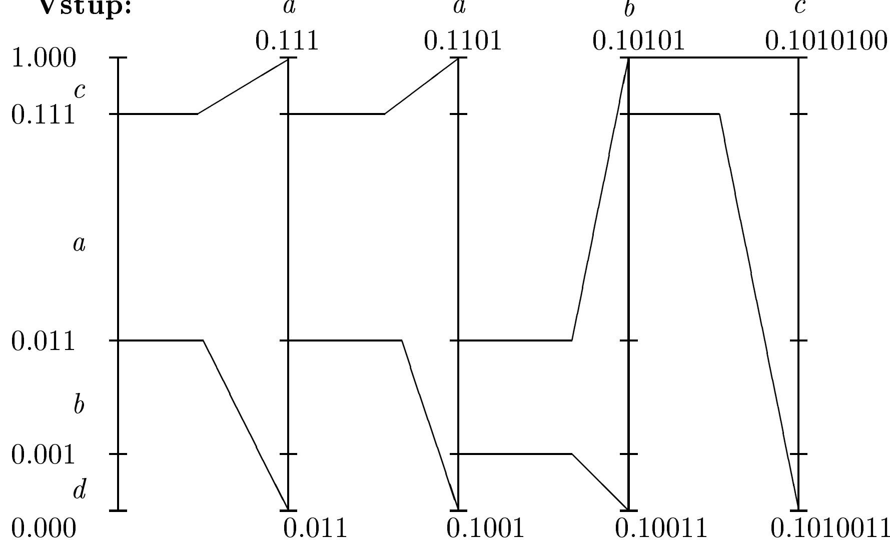
Příklad 6.9:

Na obr. 6.11 jsou uvedeny pravděpodobnosti zdrojových jednotek, jejich kumulativní pravděpodobnosti a subintervaly pro dané zdrojové jednotky.

zdrojová jednotka	pravděpodobnost p_i	kumulativní pravděpodobnost cp_i	subinterval
d	0,001	0,000	<0.000,0.001)
b	0,010	0,001	<0.001,0.011)
a	0,100	0,011	<0.011,0.111)
c	0,001	0,111	<0.111,1.000)

Obr. 6.11: Pravděpodobnosti pro aritmetické kódování

Zmenšování intervalu při aritmetickém kódování je graficky znázorněno na obr. 6.12 pro poslopnost *aabc*. Tato posloupnost je kódována takto:



Obr. 6.12: Aritmetické kódování

Na začátku máme interval $< 0, 1)$. Tento interval je rozdělen na subintervaly podle kumulativních pravděpodobností zdrojových jednotek. Pro první vstupní symbol a je vybrán subinterval $< 0.011, 0.111)$. Tento interval se dále rozdělí na subintervaly stejným způsobem. Pro druhý vstupní symbol a je vybrán podinterval $< 0.1001, 0.1101)$, jako nový interval. Pokračujeme stejným způsobem, pro třetí symbol b se vybere podinterval $< 0.10011, 0.10101)$, a pro poslední symbol c vstupní posloupnosti se vybere podinterval $< 0.1010011, 0.1010100)$. Posloupnost $aabc$ je zakódována tímto intervalem a jakékoliv číslo v tomto intervalu ji může reprezentovat. Číslo 0.1010011 je vhodné, protože je nejkratším reprezentantem uvedeného intervalu. Vstupní řetěz $aabc$ je zakódován pomocí bitového řetězce 1010011.

Interval můžeme reprezentovat jako dvojici (L, S), kde L je dolní mez intervalu a S je jeho délka. Počáteční hodnoty těchto proměnných jsou $L = 0, S = 1$. Během kódování vstupní jednotky se počítají nové hodnoty proměnných L a S takto:

$$L := L + (S * cp_i),$$
$$S := S * p_i,$$

kde cp_i je kumulativní pravděpodobnost $i-$té zdrojové jednotky a p_i je pravděpodobnost $i-$té zdrojové jednotky.

Jedním z potenciálních problémů je skutečnost, že kód může být po chvíli příliš dlouhý na to, aby se s ním mohlo dále pracovat. Naštěstí je možné provést výstup a zrušit část vytvořeného kódu tak, jak kódování pokračuje. Například po zakódování druhého symbolu (a) z příkladu 6.9 je vytvořen interval $< 0.1001, 0.1101)$. Bez ohledu na další zužování tohoto intervalu, musí výstup začínat jedničkou a proto tato část může být předána na výstup a zapomenuta. Totéž platí pro druhý bit, kterým je nula, po zakódování třetího symbolu b. Během dekódování se provádí opačné operace než při kódování. Dekódování probíhá ve třech krocích:

1. Zjištění, do kterého intervalu patří číslo N reprezentující zakódovaný řetěz.
2. Odečtění dolní meze nalezeného intervalu od čísla N.
3. Dělení výsledku 2. kroku délkou nalezeného intervalu.

Příklad 6.10:

Nyní ukážeme dekódování kódu 1010011 v příkladu 6.9. Protože toto číslo patří do intervalu $< 0.011, 0.111)$, první znak musí být a. Pak odečteme dolní mez intervalu

$$0.1010011 - 0.011 = 0.0100011$$

a vydělíme výsledek délkou intervalu pro a:

$$0.0100011/0.1 = 0.100011.$$

Toto číslo patří opět do intervalu $< 0.011, 0.111)$. Druhý symbol je opět a a proces pokračuje:

$$0.100011 - 0.011 = 0.001011,$$

$$0.001011/0.1 = 0.01011.$$

Toto číslo patří do intervalu $< 0.001, 0.011)$, a další symbol je tedy b.

$$0.01011 - 0.001 = 0.00111,$$

$$0.00111/0.01 = 0.111.$$

Toto číslo patří do intervalu $< 0.111, 1.0)$ a proto další symbol musí být c.

$$0.111 - 0.111 = 0.0,$$

$$0.0/0.001 = 0.0.$$

V tomto okamžiku je kód dekódován, ale dekodér může pokračovat, protože nula je kód pro libovolně dlouhou posloupnost symbolů d. Proto je nutné použít speciální koncový symbol, který určí konec vstupních dat.

Aritmetické kódování má tyto hlavní vlastnosti:

- Zakóduje daná data tak, že kód je libovolně blízký entropii

  $AE = -\sum p(x_i) log_2 p(x_i)$,

  kde $x_1, x_2, \ldots, x_n$ jsou zdrojové jednotky.

- Rozložení pravděpodobností se může měnit při kódování každé zdrojové jednotky a tedy adaptivní aritmetické kódování je možné.

- Je rychlé, protože je nutno použít pouze několik operací celočíselné aritmetiky pro zakódování zdrojové jednotky a je k tomu potřeba pouze několik registrů.

- Celý proces komprese je jasně rozdělen na dvě části: model použitý pro rozložení pravděpodobností a aritmetický kodér.

6.5 Slovníkové metody komprese dat

Slovníkové metody komprese dat jsou založeny na pozorování, že se zejména v textu některá slova opakují. Například v této kapitole se často opakují slova jako *kódování, komprese, data.* Slovníkové metody kódují slova (podřetězy, fráze) pomocí ukazatelů do slovníku.

Definice 6.4

Slovník je dvojice $D = (M, C)$, kde M je konečná množina slov, C je zobrazení, které zobrazuje M na množinu kódových slov. $L(m)$ bude označovat délku kódového slova $C(m)$, $m \in M$, v bitech.

Řetězy v určitém slovníku budou zdrojovými jednotkami podle naší dosavadní terminologie. Výběr řetězů do slovníku je možno provádět statickým, semiadaptivním a adaptivním způsobem.

6.5.1 Statické slovníkové metody

Statická slovníková metoda používá slovník, který je předem připraven. Pro přípravu takových slovníků je možno použít dvě základní metody:

První metoda používá slovník obsahující řetězy pevné délky. Tyto řetězy délky n jsme nazývali $n - gramy$. Nejčastější je použití digramů, tj. dvojic sousedních symbolů. V každém kroku je ve slovníku digramů hledána dvojice následujících symbolů. Jestliže je nalezena, je zakódována. V opačném případě je první symbol zakódován samostatně.

Druhý přístup k vytváření slovníku používá slova proměnné délky a to taková, která mají největší frekvenci. Toto je založeno na zjištění, že asi 50% anglického textu je tvořeno asi 150 nejfrekventovanějšími slovy. (Tuto skutečnost využijeme dále pro implementaci interaktivní metody kontroly správnosti textu).

Statické slovníkové metody mají nevýhody všech statických metod. Nejsou schopny reagovat přesně na rozdělení pravděpodobností výskytu slov v komprimovaných datech a proto dosažený kompresní poměr nemusí být příliš dobrý.

6.5.2 Semiadaptivní slovníkové metody

Hlavní nevýhodu statických slovníkových metod je možno překonat vytvořením slovníku, který odpovídá komprimovaným datům. Nevýhodou tohoto přístupu je skutečnost, že slovník musí být součástí komprimovaných dat.

Dále budeme předpokládat, že ukazatele do slovníku jsou dvojice (n, d), kde n je pozice prvního symbolu slova ve slovníku a d je délka slova.

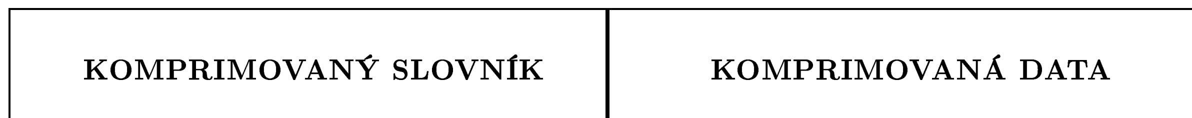
Existují dvě možnosti implementace semiadaptivní slovníkové komprese dat:

1. Slovník je první částí komprimovaných dat a sám není komprimován. Tuto situaci ukazuje obr. 6.13. Komprimovaná data jsou tvořena ukazateli do slovníku. Původní data získáme tak, že v komprimovaných datech nahradíme každý ukazatel příslušnou položkou ze slovníku.

Obr. 6.13: Semiadaptivní slovníková komprese (1. metoda)

2. Slovník je první částí komprimovaných dat a je sám komprimován. Tuto situaci znázorňuje obr. 6.14. Komprimovaná data jsou opět tvořena ukazateli do původního (nekomprimovaného) slovníku. Původní data získáme ve dvou krocích:
   - dekomprese slovníku,
   - nahrazením každého ukazatele z komprimovaných dat položkou ze slovníku.

Semiadaptivní slovníkové metody jsou výhodné zejména pro rozsáhlá data, protože v takovém případě tvoří slovník jen malou část celých komprimovaných dat. Kompresní poměr může



Obr. 6.14: Semiadaptivní slovníková komprese (2. metoda)

být lepší než pro statické a adaptivní slovníkové metody. To ovšem záleží na metodě tvorby slovníku.

Typický postup pro vytvoření téměř optimálního slovníku je tento:

Nejdříve se určí frekvence jednotlivých symbolů, dále pak digramů, trigramů, $\dots, n$-gramů pro určité n. Potom se slovník inicializuje vložením jednotlivých symbolů. Dále jsou do slovníku postupně vkládány digramy, trigramy, $\dots, n$-gramy tak, že se vždy začíná nejfrekventovanějšími k-gramy ($k = 2, 3, \dots, n$). Při vložení k-gramu do slovníku se snižuje frekvence jeho složek (dvou(k-1)-gramů, tří (k-2)-gramů, ..., n samostatných symbolů).

Jestliže díky snižování frekvencí se frekvence nějaké položky ve slovníku „velmi sníží“, pak je tato položka ze slovníku vyloučena. Tento proces pokračuje, dokud není slovník plný.

6.5.3 Adaptivní slovníkové metody

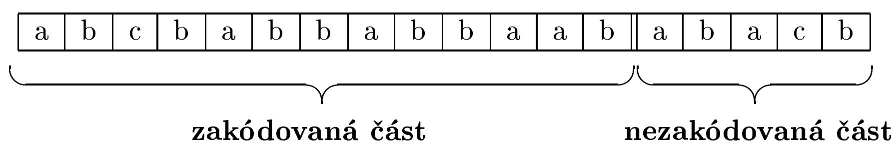
Velká část adaptivních slovníkových metod komprese dat je založena na dvou základních principech, které popsali Lempel a Ziv ve dvou článcích v letech 1977 a 1978. Tyto metody jsou označovány jako LZ77 a LZ78. Metoda LZ77 je označována dále jako metoda posuvného okna, metoda LZ78 používá rostoucí slovník.

6.5.3.1 Metoda posuvného okna

Metoda LZ77 používá ukazatele do okna pevné délky, které předchází místu, které se právě kóduje. Existuje řada variant této metody a dále jsou uvedeny principy nejdůležitějších z nich. Nejdříve popíšeme originální verzi metody LZ77.

Metoda LZ77

Na obr. 6.15 je uvedeno posuvné okno, které je rozděleno na část již zakódovaného řetězu a na část dosud nezakódovanou. Typická velikost okna je $N \leq 8192$ bytů a velikost části okna



Obr. 6.15: Posuvné okno

pro nezakódovanou část B se pohybuje v rozmezí 10 až 20 bytů. Na začátku je prvních $N - B$ pozic okna obsazeno mezerami a prvních B znaků textu uloženo v místě pro nazakódovanou část.

Jeden krok kódování probíhá takto:

V okně je vyhledána nejdelší předpona p řetězu, který je v dosud nezakódované části. Pokud je takový řetěz s nalezen s tím, že jeho začátek je v zakódované části, pak předpona p je zakódována pomocí trojice (i, j, a), kde i je vzdálenost prvního znaku podřetězu s od hranice mezi zakódovanou a nezakódovanou částí okna, j je délka řetězu s a a je první znak za předponou p. Okno je potom v textu posunuto o $j + 1$ znaků doprava. Jestliže žádný podřetěz odpovídající předponě nazakódované části nebyl nalezen, pak je vytvořena trojice $(0, 0, a)$, kde a je první znak nezakódované části. Řetěz s může přesahovat i do dosud nezakódované části.

Příklad 6.11:

Na obr. 6.15 je nalezený řetěz $s = aba$ a ten začíná dva znaky před hranicí obou částí okna.

To znamená, že nejdelší předpona je zakódována trojicí $(2, 3, c)$. Okno je posunuto o $3 + 1 = 4$ znaky.

Dekódování se provádí podobným způsobem jako kódování. Dekodér má stejné okno jako kodér rozdělené na dekódovanou a nedekódovanou část, ale místo hledání nejdelší předpony kopíruje část textu odpovídající příslušné trojici z dekódované části do nedekódované části.

Za slovník lze v tomto případě považovat okamžitý obsah okna. Počet položek slovníku je $|M| = (N - B) * B * t$, kde t je velikost abecedy T. Počet bitů pro zakódování ukazatele na položku ve slovníku je

$L(m) = \lceil log_2(N - B) \rceil + \lceil log_2 B \rceil + \lceil log_2 t \rceil.$

Hlavní nevýhodou metody LZ77 je, že během každého kódovacího kroku musí být vyhledávána nejdelší předpona dosud nezakódovaného řetězu. Pro tuto operaci je možno použít KMP algoritmus (viz odst. 4.2.1.2).

Metoda LZR

Metodu LZR navrhl Rodeh. Tato metoda je velmi podobná metodě LZ77 až na tyto rozdíly:

1. LZR používá strom obsahující všechny předpony v dosud zakódované části.
2. Je použita celá dosud zakódovaná část jako slovník.
3. Protože hodnoty i v trojici (i, j, a) mohou být libovolně velké, je použit kód ω' pro zakódování celých čísel (viz odst. 6.3.2).

Nevýhodou metody LZR je skutečnost, že velikost použitého stromu může růst bez omezení. Obvykle je po vyčerpání vymezené paměti strom vymazán a jeho konstrukce začíná od začátku.

Metoda LZSS

Metodu LZSS navrhl Bell na základě myšlenky Storera a Szymanskiho. Výstupem je při této metodě posloupnost ukazatelů a znaků. Ukazatelé jsou dvojice (i, j), kde i a j mají stejný význam jako při metodě LZ77. Předpokládejme, že ukazatel potřebuje stejnou paměť jako p znaků. To znamená, že má smysl vytvářet ukazatele jen na podřetězy delší než p znaků, jinak jsou do výstupu kopírovány jednotlivé znaky.

Výstup je tedy směs ukazatelů a nazakódovaných znaků, což znamená, že je nutno použít zvláštní bit na rozlišení, zda se jedná o ukazatel nebo znak.

Počet položek ve slovníku je $|M| = t * (N - B) * (B - p)$, protože se uvažují pouze podřetězy delší než p. Počet bitů pro zakódování je

$$L(m) = \begin{cases} 1 + \lceil log_2 t \rceil & \text{když } m \in T, \\ 1 + \lceil log_2 N \rceil + \lceil log_2(B - p) \rceil & \text{jinak,} \end{cases}$$

protože délku d podřetězu (druhý prvek ukazatele) můžeme reprezentovat jako $B - p$.

Metoda LZB

Metodu LZB navrhl rovněž Bell a je podobná metodě LZSS. Rozdíl je v kódování ukazatelů. Ukazatele jsou dvojice (i, j), kde i je ukazatel do okna na začátek podřetězce, který se rovná předponě nezakódované části. V následujících dvou situacích je plýtváním použití $\log_2 N$ bytů pro zakódování i:

1. Okno není plné. Tato situace nastává na začátku komprese dlouhého řetězu.
2. Komprimovaný text je kratší než N.

V obou případech je možno použít kratší kód pro i.

Metoda LZB používá fázování při binárním kódování. To znamená, že je použit prefixový kód s rostoucím počtem bitů pro rostoucí hodnoty čísel. Příklad tohoto kódování pro čísla od 0 do 8 je uveden v následující tabulce:

číslo	kód
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	1110
8	1111

Použití standardní binární reprezentace čísel v intervalu $< 0, 8 >$ vede na čtyřbitový kód. V tomto případě jsou kódy čísel od 0 do 6 tříbitové. Pro druhou komponentu ukazatele používá metoda LZB kód γ.

Metoda LZH

Metoda LZH byla navržena Brentem jako varianta metody LZSS, ve které je pro kódování ukazatelů použito Huffmanovo kódování. Zatímco metoda LZB používá jednoduché kódování pro reprezentaci obou komponent ukazatele, metoda LZH používá kódování ukazatelů odvozené od rozložení jejich pravděpodobností. Tento postup vede na lepší kompresi, ale přináší určité komplikace.

Kódování probíhá ve dvou průchodech. Při prvním průchodu je použita metoda LZSS, ke které je přidán výpočet statistických charakteristik. Druhý průchod probíhá jako statické Huffmanovo kódování. Huffmanův kód musí být částí komprimované zprávy.

6.5.3.2 Metody s rostoucím slovníkem

Tyto metody založené opět na myšlence Lempela a Ziva vytváří slovník tak, že do něho vkládají fráze z textu, který je komprimován. Opět existuje několik variant a nejdříve uvedeme metodu LZ78.

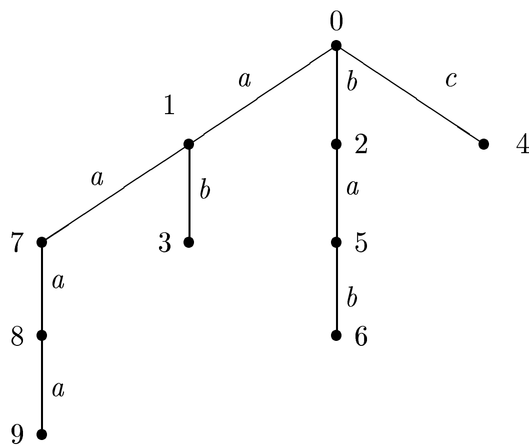
Metoda LZ78

Metoda LZ78 rozděluje text na fráze. Každá nová fráze přidávaná do slovníku je vytvořena tak, že již ve slovníku existující fráze je rozšířena o jeden symbol. Každá fráze je zakódována indexem její předpony a přidaným symbolem. Tato fráze je přidána do slovníku a je možno se na ni odvolávat pomocí příslušného indexu.

Příklad 6.12:

vstupní řetěz	a	b	$a\ b$	c	$b\ a$	$b\ a\ b$	$a\ a$	$a\ a\ a$	$a\ a\ a\ a$
index fráze	1	2	3	4	5	6	7	8	9
výstup	$(0, a)$	$(0, b)$	$(1, b)$	$(0, c)$	$(2, a)$	$(5, b)$	$(1, a)$	$(7, a)$	$(8, a)$

Během komprese je vytvářen následující slovník.



Obr. 6.16: Stromová struktura pro metodu LZ78

fráze	index	zakódovaná fráze
a	1	a
b	2	b
ab	3	$1b$
c	4	c
ba	5	$2a$
bab	6	$5b$
aa	7	$1a$
aaa	8	$7a$
$aaaa$	9	$8a$

Výstup se skládá ze dvou částí (index fráze, symbol). Fráze obsahující pouze jeden symbol jsou kódovány tak, že první část je rovna nule. Časově nejnáročnější část této metody je vyhledávání ve slovníku. Toto vyhledávání je možno realizovat s použitím stromové struktury. Strom pro úlohu z příkladu 6.12 je uveden na obr. 6.16. Pro každou frázi je zde uzel obsahující index. Fráze odpovídá cestě z kořene do odpovídajícího uzlu. Dekodér vytváří stejnou tabulku jako kodér. Když je paměťový prostor pro slovník vyčerpán, je slovník vymazán a proces kódování a dekódování začíná od začátku.

Metoda LZW

Metoda LZW navržena Welchem je jedna z variant metody LZ78. Jejím výstupem jsou pouze indexy. Vynechání symbolů z dvojic v LZ78 je možné díky těmto dvěma principům:

- Slovník je inicializován položkami pro všechny vstupní symboly.
- Poslední symbol každé fráze je prvním symbolem následující fráze.

Příklad 6.13:

vstupní řetěz	a	b	a	b	c	b	a	b	a	b	a	a	a	a	a	a
index fráze		4	5		6	7		8			9	10		11		
výstup	1	2		4	3		5			8	1		10			11

Během komprese je vytvořen tento slovník:

fráze	index	zakódovaná fráze
a	1	a
b	2	b
c	3	c
ab	4	$1b$
ba	5	$2a$
abc	6	$4c$
cb	7	$3b$
bab	8	$5b$
$baba$	9	$8a$
aa	10	$1a$
aaa	11	$10a$

Dekodér nahrazuje indexy frázemi ze slovníku a vytváří stejný slovník. Problém, který se může vyskytnout, je případ, kdy vstupní řetěz obsahuje posloupnost $axaxa$, kde a je jeden symbol, x je nějaký řetěz symbolů a fráze ax je již ve slovníku. Kodér najde frázi ax, přidá její index do výstupu a frázi axa vloží do slovníku, následující fráze vložená do výstupu bude axa. Dekodér při dekódování tohoto kódu nebude mít tento kód ještě ve slovníku, protože nezná další symbol pro předchozí frázi. Tento problém může být vyřešen, protože první a poslední symbol takové „neznámé“ fráze musí být stejný a fráze je rozšířením poslední dekódované fráze.

Popsaná situace se vyskytuje v příkladu 6.13, když je generován index 8. V tomto okamžiku dekodér má ve slovníku pouze index 7. Problém je vyřešen vytvořením fráze 5b=bab=8. Situace „přeplněný slovník“ se řeší statickým způsobem. To znamená, že v okamžiku přeplnění slovníku žádná další fráze není přidávána a kódování pokračuje staticky.

LZC metoda

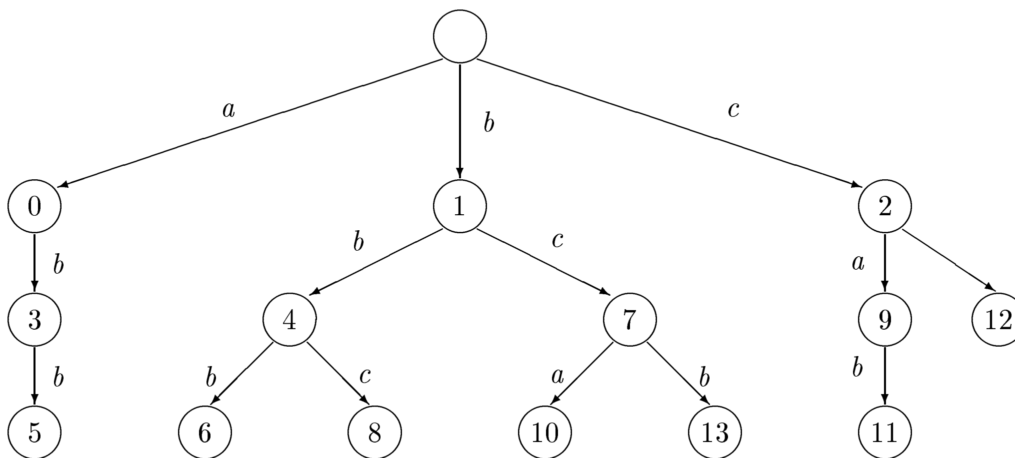
Metoda LZC je použita v programu „compress“, který je součástí UNIXových systémů. Je založena na metodě LZW, která je rozšířena o dva nové principy. První spočívá v tom, že ukazatele jsou kódovány pomocí kódů s prodlužující se délkou podle počtu frází ve slovníku. Druhý princip se uplatní v případě přeplnění slovníku. LZC monitoruje kompresní poměr a jakmile se ten začne snižovat, slovník se vymaže a začíná se znovu od počátečního nastavení.

LZT metoda

Metoda LZT byla navržena Tischerem a je založena na metodě LZC. Metoda LZT se liší od LZC řešením problému přeplnění slovníku. Jakmile je slovník plný, metoda LZT vylučuje ze slovníku fráze, které byly v nedávné minulosti nejméně použity. Slovník je organizován jako samoorganizující se seznam. Když se některá fráze do slovníku vkládá při kódování, vkládá se na začátek seznamu a tím se stává nejposledněji použitou frází. Během vkládání nové fráze se poslední fráze ze seznamu vylučuje. Navíc metoda LZT používá fázování při binárním kódování pro kódování indexů frází.

LZMW metoda

Metoda LZMW navržená Millerem a Wegmanem je založena na metodě LZT. Hlavní rozdíl oproti LZT metodě spočívá v určení nové fráze. Zatímco ostatní metody vytváří novou frázi přidáním jednoho symbolu k existující frázi, metoda LZMW konstruuje novou frázi zřetězením dvou posledně zakódovaných frází.



Obr. 6.17: Strom pro řetěz *abbbcabbcb*.

LZJ metoda

Metoda LZJ navržená Jakobssonem se liší od všech ostatních metod z rodiny LZ78 principem konstrukce slovníku. Tento slovník obsahuje všechny podřetězy již zpracovaného řetězu do nějaké maximální délky h. Na začátku jsou do slovníku vloženy jednotlivé symboly stejně jako v metodě LZW.

Metoda LZJ používá strom pro uložení slovníku. Počáteční strom se skládá z kořene pro prázdný řetěz a následníky pro všechny symboly abecedy. Na obr. 6.17 je uveden tento strom pro abecedu $T = \{a, b, c\}$, $h = 3$ a řetěz $S = abbbcabbcb$.

Následující tabulka ukazuje, jak probíhá komprese a jak roste slovník během komprese řetězu $S = abbbcabbcb$.

Vstup $a\ b\ b\ b\ c\ a\ b\ b\ c\ b$
Výstup 0 1 1 1 2 5 2 1

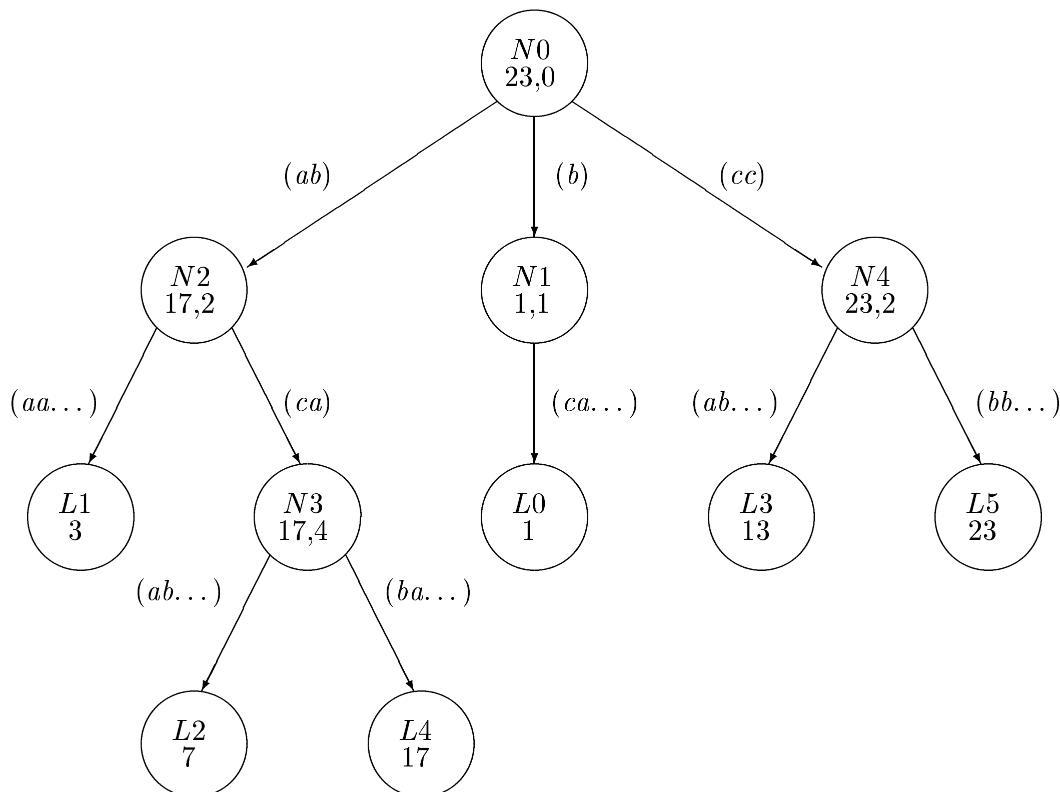
číslo fráze	fráze
0	*a*
1	*b*
2	*c*
3	*ab*
4	*bb*
5	*abb*
6	*bbb*
7	*bc*
8	*bbc*
9	*ca*
10	*bca*
11	*cab*
12	*cb*
13	*bcb*

Jakmile je slovník plný, mohou být použity dvě alternativy. První spočívá v pokračování komprese statickým způsobem s vytvořeným slovníkem. Druhá alternativa spočívá ve vynechávání uzlů s frekvencí použití menší než zadaná konstanta. V tomto případě musí uzly obsahovat čítač frekvence použití, který je inkrementován při každé návštěvě uzlu během vyhledávání fráze.

LZFG metoda

Metoda LZFG navržená Fialou a Greenem používá také stromovou strukturu pro uložení slovníku. V tomto stromu jsou hrany ohodnoceny řetězy tvořené jedním nebo více symboly. Tyto řetězy jsou uloženy v okně a každý uzel stromu obsahuje ukazatel ukazující do okna a identifikující symboly na cestě z kořene do tohoto uzlu. Uzly jsou rozděleny do dvou typů: vnitřní uzly a koncové uzly. Obr.6.18 ukazuje takový strom a okno pro 26 symbolů, které byly zakódovány touto posloupností:

(1,L0) (2,L1) (2,L2) (2,L3) (2,L4) (2,L5)



okno	*b*	*c*	*a*	*b*	*a*	*a*	*a*	*b*	*c*	*a*	*a*	*b*	*c*	*c*
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

okno	*a*	*b*	*a*	*b*	c	*a*	*b*	*a*	*c*	*c*	*b*	*b*
	15	16	17	18	19	20	21	22	23	24	25	26

Obr. 6.18: Strom a okno v LZFG metodě

Každá fráze je zakódována pomocí dvojice *(d,n)*, kde d je počet symbolů shodných s řetězem z ohodnocení hrany vedoucí do uzlu n a n je identifikace uzlu. Jestliže následující část vstupu je $abcb\ldots$, pak fráze abc bude zakódována pomocí dvojice $(N3,1)$, protože fráze abc je shodná až do symbolu c s ohodnocením hran na cestě vedoucí z kořene ke hraně mezi vnitřními uzly $N2$ a $N3$. V tomto případě je fráze zakódována dvojicí (n,d), kde n je identifikace uzlu na konci hrany, na které došlo k neshodě, a d je počet shodných symbolů na téže hraně. Jestliže následující část vstupu je $ccabac$, pak fráze bude $ccaba$ a bude zakódována pomocí dvojice

$(3, L3)$, protože tato fráze se shoduje se symboly na hraně $(N0, N4)$ a s řetězem abc na hraně $(N4, L3)$ – řetězec ohodnocující tuto hranu je neomezený řetězec začínající v okně na pozici 13.

Jestliže další část vstupního řetězu je $cacbca$, pak první tři symboly cac tohoto řetězu se neshodují se žádným řetězcem delším než jeden symbol. V tomto případě bude řetěz cac zakódován jako literál ve tvaru $(3, cac)$, kde první číslo je délka literálu.

Jakmile je fráze zakódována, následuje úprava stromu. Každá zakódovaná fráze se ukládá do okna. Literál se vloží do stromu tak, že se vytvoří nový list a hrana z kořene do tohoto uzlu ohodnocená vkládaným literálem. Pokud byla fráze zakódována pomocí dvojice (d, n), je hrana, na které došlo k neshodě rozdělena na dvě v místě neshody a zde je vytvořen nový vnitřní uzel a jeden nový list jako jeho následník. Hrana mezi novým vnitřním uzlem a listem není ohodnocena, ale ohodnocení bude vytvořeno a přidáno do okna před tím, než bude jeho ukazatel použit. Jestliže v okně není místo pro novou frázi, pak jsou z okna vyloučeny nejstarší fráze a odpovídající listy jsou ve stromě zrušeny.

6.5.3.3 Slovníkové metody s restrukturalizací slovníku

Ve všech výše uvedených adaptivních slovníkových metodách byl slovník rozšířen přidáním nového slova na jeho konec. Nyní se podíváme na metody, ve kterých se slovník bere jako seznam a položky slovníku jsou tedy uspořádáné. Toto uspořádání je zvlášť užitečné, když nejvíce pravděpodobná slova jsou blízko začátku seznamu. V tomto případě je možné zakódovat jejich indexy (jsou to malá čísla) krátkou posloupností bitů. Pro dosažení hlavního cíle, zejména umístění nejvíce pravděpodobných slov blízko začátku seznamu, se používá několik heuristických metod pro organizaci seznamů. Uveďme několik z nich:

- **Přesun na začátek**. Když nalezneme slovo, přesuneme jej na začátek seznamu.
- **Vyměňování**. Při nalezení slova jej přesuneme o jedno blíže k začátku seznamu tak, že jej vyměníme s předcházejícím.
- **Přesun vpřed o k**. Toto je kompromis mezi přesunem na začátek a vyměňováním. Při nalezení slova jej přesuneme dopředu o k pozic.
- **Četnost**. Při nalezení slova je jeho četnost zvýšena a je přesunuto na odpovídající pozici ve slovníku, který je sestupně setříděn podle této četnosti.

Metoda BSTW

Metoda BSTW je založena na heuristice *přesun na začátek*, která je použita pro restrukturalizaci slovníku, a na proměnné délce zakódování čísel. Problém hledání slova ve vstupu není popsán a můžeme tedy použít mnoho různých přístupů. Slovník je na začátku prázdný. Pro zakódování slova jej kodér hledá ve slovníku. Je-li nalezeno na pozici i, je zakódováno jako i. Protože dekodér má ten samý slovník je schopen je dekódovat. Kodér i dekodér přesunou dané slovo na první pozici ve slovníku a odsunou slova na pozicích $1, 2, \ldots, i-1$ na pozice $2, 3, \ldots, i$. Jestliže slovo není dosud ve slovníku je přidáno na konec slovníku a je zakódováno jako číslo $n + 1$ následované vlastním slovem (n je aktualní počet slov ve slovníku). Potom se provede *přesun na začátek* jako v předchozím případě.

Příklad 6.14:

Předpokládejme, že chceme zakódovat následující text:

KAŽDÝ SOFSEMISTA MUSÍ KAŽDÝ DEN BDÍT CELÝ DEN

Slovník se bude měnit takto:

KAŽDÝ
SOFSEMISTA KAŽDÝ
MUSÍ SOFSEMISTA KAŽDÝ
KAŽDÝ MUSÍ SOFSEMISTA
DEN KAŽDÝ MUSÍ SOFSEMISTA
BDÍT DEN KAŽDÝ MUSÍ SOFSEMISTA
CELÝ BDÍT DEN KAŽDÝ MUSÍ SOFSEMISTA
DEN CELÝ BDÍT KAŽDÝ MUSÍ SOFSEMISTA

Celá zpráva bude zakódována takto:
1 KAŽDÝ 2 SOFSEMISTA 3 MUSÍ 3 4 DEN 5 BDÍT 6 CELÝ 3

Je zřejmé, že se každé slovo v původní podobě objeví v zakódované formě textu pouze jednou. Čísla jsou kódována použitím kódu proměnné délky γ. Dále je možno použít Fibonacciho kódy pro zlepšení komprese.

Tato metoda je zejména výhodná při kompresi zpráv s větším počtem stejných slov v lokálním kontextu. Jinak řečeno, algoritmus odráží lokalitu výskytů slov. Nedávno použitá slova jsou umístěna blízko začátku seznamu a jsou tedy kódována kratšími kódy. Je zřejmé, že není jednoduché rozhodnout jak rozdělit text do slov. Například řetězce jako jsou mnohoslovné výrazy mohou být brány jako slova ve slovníku.

BSTW metoda může být mnohom lepší než statické Huffmanovo kódování. Uvažujme slova $\{1, 2, \ldots, n\}$ a zprávu $1^n 2^n \ldots n^n$, tj. každé z n slov má n za sebou jdoucích výskytů. Zatímco Huffmanovo kódování vede ke kódům pevné délky, tj. zpráva vyžaduje $n^2 log_2 n$ bitů, zde nám stačí pouze malé konstantní množství bitů na jedno slovo.

Koeficient nedávnosti a intervalové kódování

Počet různých slov mezi jejich dvěmi následujícími výskyty se nazývá koeficient nedávnosti. Kódování pomocí koeficientu nedávnosti je ekvivalentní BSTW metodě. Heuristika „přesun na začátek“ je jednoduchou implementancí výpočtu koeficientu nedávnosti. Další metodou založenou na přeuspořádávání slovníku je intervalové kódování. Intervalové kódování reprezentuje slovo celkovým počtem slov, které se vyskytly od jeho posledního výskytu. Předpokládejme, že slovník obsahuje slova $a_1, a_2, \ldots, a_n$. Ke každému slovu a_i, je přiřazeno číslo $LAST(a_i)$, které obsahuje „čas“ posledního výskytu a_i, $1 \leq i \leq m$. Tato hodnota je inicializována na nulu.

Vstupní posloupnost je rozšířena doleva přidáním všech n frází v opačném pořadí. Nechť vstupní posloupnost je $x_1, x_2, \ldots, x_t, \ldots, x_m$. Potom intervalové kódování můžeme popsat následujícím fragmentem programu:

```
for t := 1 to m do
begin {x_t = a_i}
    if LAST(x_t) = 0 then y(t) = t + i − 1 else y(t) = t−LAST(x_t);
    LAST(x_t) := t
end.
```

Poloupnost $y_1, y_2, \ldots, y_m$ je výstupem kodéru a může být dále zakódována některým kódem proměnné délky.

Délka intervalu použitého v intervalovém kódování není menší než koeficient nedávnosti zakódovaného slova. To znamená, že kódování použije méně bitů na jedno slovo než intervalové kódování. Slovům objevujícím se v krátkém úseku zprávy bude opět přiřazen kratší kód. Navíc není nutné udržovat slovník slov, stačí seznam indexů jejich posledních výskytů.

6.6 Syntaktické metody

Syntaktický přístup ke kompresi dat je výhodný, když zpráva náleží k nějakému formálnímu jazyku a jestliže je možné vytvořit gramatiku popisující tento jazyk. Příkladem takových jazyků jsou programovací jazyky. Proto může být syntaktický přístup výhodný pro kompresi programů. Program může být reprezentován derivačním stromem a množinou identifikátorů a konstant. Zaměříme se na kompresi derivačních stromů. Pro kompresi identifikátorů a konstant je možné použít například slovníkové metody. Existují dvě metody zakódování derivačních stromů:

1. derivační kódování,
2. analytické kódování.

6.6.1 Derivační kódování

Derivace řetězu v dané gramatice se nazývá *levou* (*pravou*), jestliže v každém kroku derivace je neterminál stojící nejvíce vlevo (vpravo) ve větné formě nahrazen pravou stranou gramatického pravidla.

Derivační strom řetězce w je možné reprezentovat v lineární formě *levého rozkladu*, což je posloupnost čísel pravidel gramatiky použitých při levé derivaci řetězce w. Podobně *pravý rozklad* w je převrácená posloupnost čísel gramatických pravidel použitých při pravé derivaci w. Pro dané w je pravý rozklad permutací levého rozkladu a nikoliv pouhým převrácením posloupnosti čísel.

Standardní přístup k číslování pravidel je použití globálních čísel pravidel, což znamená, že každé pravidlo gramatiky má své unikátní číslo. Avšak není nutné použít těchto globálních čísel pravidel v levém nebo pravém rozkladu. Protože neterminální symbol, který je přepisován v levé nebo pravé derivaci, je znám předem, stačí identifikovat pouze pravidlo použité pro tento neterminál. Toto číslování se nazývá *lokální číslování*. Jestliže existuje pouze jedno pravidlo pro daný neterminál, není třeba ho kódovat vůbec.

Příklad 6.15:

Nechť $G_1 = (\{S, A\}, \{a, b\}, P_1, S)$ je bezkontextová gramatika obsahující následující pravidla s globálními čísly v hranatých závorkách a lokálními čísly v kulatých závorkách:

[(0)] (0) $S \rightarrow AS$

[(1)] (1) $S \rightarrow \epsilon$

[(2)] (0) $A \rightarrow a$

[(3)] (1) $A \rightarrow b$

Levá derivace řetězce $abab$ je

$$S \Rightarrow AS \Rightarrow aS \Rightarrow aAS \Rightarrow abS \Rightarrow abAS \Rightarrow abaS \Rightarrow abaAS \Rightarrow ababS \Rightarrow abab.$$

Levý rozklad vyjádřený pomocí globálních čísel je 020302031. Tentýž levý rozklad vyjádřený pomocí lokálních čísel je 000100011.

Při použití globálního číslování jedno číslo může být zakódováno pomocí 2 bitů a celkový počet bitů je 18. Při použití lokálního číslování stačí celkem 9 bitů. Pro bezkontextovou gramatiku můžeme nalézt mnoho ekvivalentních gramatik. Ty mohou mít odlišné vlastnosti vzhledem k zakódovávání derivačních stromů.

Příklad 6.16:

Gramatika $G_2 = (\{S\}, \{a, b\}, P_2, S)$ s pravidly

$[(0)]\ S \rightarrow aS$ $[(1)]\ S \rightarrow bS$ $[(2)]\ S \rightarrow \epsilon$

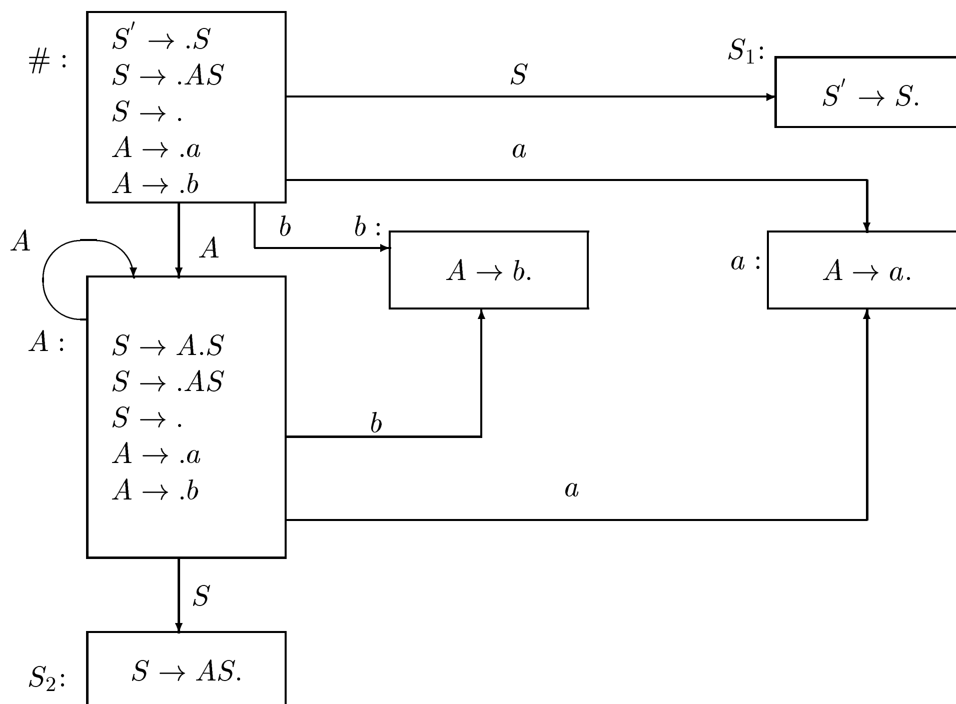
je ekvivalentní gramatice G_1. Řetězec *abab* může být zakódován pomocí levého rozkladu 01012 v gramatice G_2.

6.6.2 Analytické kódování

Analytické kódování znamená zaznamenání posloupnosti rozhodnutí provedených syntaktickým analyzátorem. Předpokládáme, že čtenář je seznámem s hlavními principy LR analýzy.

Příklad 6.17:

LR automat pro gramatiku G_1 z příkladu 6.15 je na obr. 6.19.



Obr. 6.19: LR automat pro gramatiku G_1

Analyzátor je následující:

stav	operace analyzátoru	výstupní kód
#	pro a:*přesun* a	0
	pro b:*přesun* b	1
	jinak :*redukce* pomocí $S \rightarrow \epsilon$	2
S_1	*přijetí*	
A	pro a:*přesun* a	0
	pro b:*přesun* b	1
	jinak:*redukce* pomocí $S \rightarrow \epsilon$	2
a	*redukce* pomocí $A \rightarrow a$	
b	*redukce* pomocí $A \rightarrow b$	
S_2	*redukce* pomocí $S \rightarrow AS$	

Pro řetězec *abab* LR analyzátor provede tuto posloupnost kroků:

zásobník	vstup	operace	výstupní kód
$\#$	$abab$	*přesun* a	0
$\#a$	bab	*redukce* $A \to a$	
$\#A$	bab	*přesun* b	1
$\#Ab$	ab	*redukce* $A \to b$	
$\#AA$	ab	*přesun* a	0
$\#AAa$	b	*redukce* $A \to a$	
$\#AAA$	b	*přesun* b	1
$\#AAAb$		*redukce* $A \to b$	
$\#AAAA$		*redukce* $S \to \epsilon$	2
$\#AAAAS_2$		*redukce* $S \to AS$	
$\#AAAS_2$		*redukce* $S \to AS$	
$\#AAS_2$		*redukce* $S \to AS$	
$\#AS_2$		*redukce* $S \to AS$	
$\#S_1$		*přijetí*	

Tato posloupnost kroků může být zakódována posloupností stavů navštívených během rozkladu, což je $\#aA\ bA\ aA\ bA\ S_2S_2S_2S_2S_1$. Stavy a, b, S_1, S_2 mohou být vynechány, protože v nich není žádné rozhodování. Jen ve stavech $\#$ a A se provádí nějaké rozhodování a proto je výsledné zakódování 01012.

6.7 Kontextové modelování

Komprese dat je tak dobrá, jak dobrý je použitý model dat. Model dat založený na pravděpodobnostech izolovaných zdrojových jednotek je velmi jednoduchý, ale není příliš vhodný, pokud požadujeme dobrou kompresi. Proto se pro lepší modelování dat používají kontextové modely.

6.7.1 Konečné kontextové modely

Konečné kontextové modely, uvedené v odstavci 6.2.1, určují pravděpodobnost zdrojové jednotky s použitím několika předchozích zdrojových jednotek. Jestliže je použito právě n předchozích zdrojových jednotek, potom se jedná o model řádu n. Typická délka kontextu je v rozsahu od 1 do 10 zdrojových jednotek. Adaptivní modelování s konečným kontextem spočívá v určení pravděpodobností zdrojových jednotek dosud přečtených ze vstupu. Pro všechny zdrojové jednotky musí být definovány nějaké počáteční pravděpodobnosti.

Konstrukce modelu začíná s počátečními pravděpodobnostmi, které nejsou vázány na zdrojovou zprávu. Během čtení zprávy je vytvářen model tak, že odráží strukturu zdrojové zprávy. Existují dva typy konečných kontextových modelů. První používá kontext pevné délky. Pokud použijeme krátký kontext, potom komprese nebude dobrá, protože k odhadnutí pravděpodobnosti bude použita pouze malá část ze struktury zdrojové zprávy. Pokud použijeme dlouhý kontext, komprese bude lepší, ale bude nutné zpracovat a uložit značné množství informací. Tento rozpor může být řešen použitím tzv. *kombinovaného přístupu*, kde jsou kombinovány kontexty různých délek k předpovězení pravděpodobnosti. Obecný mechanismus této kombinace je založen na přiřazení váhy každému modelu řádu n. Potom celková pravděpodobnost jednotky je určena jako vážená pravděpodobnost.

Nechť x je následující zdrojová jednotka, $p_n(x)$ pravděpodobnost x v konečném kontextovém modelu řádu n, w_n je váha modelu n, m je řád nejvyššího modelu. Váha musí být normalizována takto:

$$\sum_{n=0}^{m} w_n = 1.$$

Pravděpodobnost zdrojové jednotky x v kombinovaném modelu je

$$p(x) = \sum_{n=0}^{m} w_n p_n(x).$$

Jsou zde dva možné přístupy, jak definovat váhu. První možnost je přiřadit pevnou množinu vah modelům různých řádů. Druhou možností je počítat váhy během určování adaptivního modelu.

Existuje několik metod, jak přidělit pravděpodobnosti ke zdrojovým jednotkám, které se poprvé objeví ve zdrojové zprávě. Nejjednodušší procedura je založena na myšlence, že pravděpodobnosti všech dosud nepoužitých jednotek v konečném kontextu jsou stejné. Počáteční pravděpodobnost dosud nepoužité jednotky může být určena následujícím vztahem:

$$e = \frac{1}{C_n + 1},$$

kde n je řád modelu a C_n určuje, kolikrát byl konečný kontext použit.

Smíšené modely jsou velmi náročné na čas a paměť.

6.7.2 Modely založené na konečných automatech

Konečné kontextové modely jsou speciálním případem modelů založených na konečných automatech. Podobně jako jiné typy modelů, model založený na konečných automatech může být optimalizován nebo vytvářen během zpracování zdrojových dat. Takovéto modely jsou vytvářeny adaptivním způsobem a mohou mít výhody oproti ostatním adaptivním modelům.

6.7.2.1 Automaty s konečným kontextem

Nejdříve uvedeme postup vytváření konečného automatu z k-tic. Nechť máme množinu k-tic

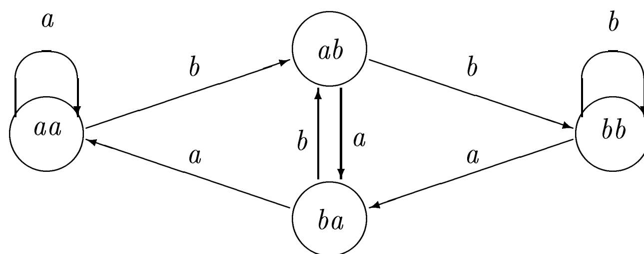
$$L = \{p_1, p_2, \dots, p_m\}.$$

Pro každou k-tici vytvoříme stav automatu. Potom následujícím způsobem vytvoříme přechody. Když $p_i = ay_i$ a $p_j = y_ib$, kde a a b jsou zdrojové jednotky, potom $p_j \in \delta(p_i, b)$. Potom je ke každému přechodu přidána pravděpodobnost.

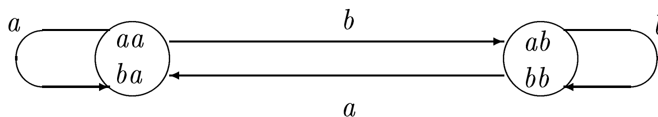
Příklad 6.18:

Mějme množinu $L = \{aa, ab, ba, bb\}$ a $k = 2$. Konečný stavový automat je na obr. 6.20. Výsledný automat může být velmi složitý (může mít velké množství stavů) v případě, že množina L má mnoho prvků. Nicméně redukce počtu stavů může být provedena sloučením některých stavů. Dva stavy mohou být sloučeny, pokud mají přechody pro stejné symboly do stejných stavů. V našem příkladu mohou být sloučeny páry stavů (aa, ba) a (ab, bb). Redukovaný automat je na obr. 6.21.

Konečný automat vytvořený touto metodou může být aktualizován inkrementálně. Předpokládejme, že se na vstupu objeví nová zdrojová jednotka. Ta vytvoří novou k-tici s novou jednotkou na jeho konci. Pokud je k-tice nová, potom je vytvořen nový stav s příslušným přechodem do něho. Přechod z tohoto stavu je vytvořen během zpracování následující jednotky. Potom může být automat redukován použitím stejného principu popsaného výše.



Obr. 6.20: Konečný automat pro množinu párů $L = \{aa, ab, ba, bb\}$



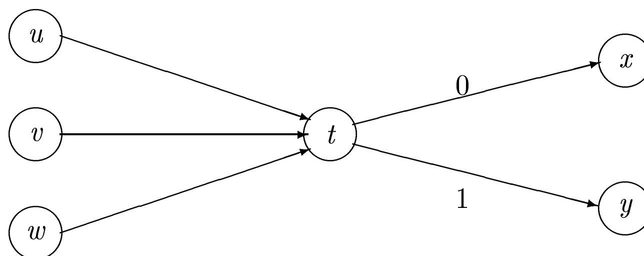
Obr. 6.21: Redukovaný konečný automat

6.7.2.2 Dynamické Markovovo modelování

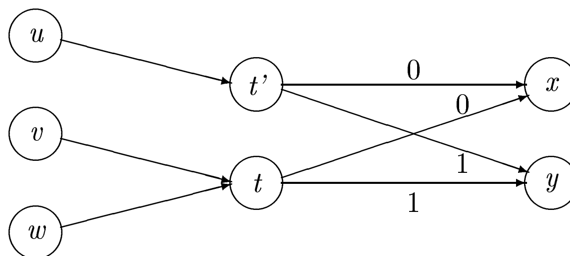
Dynamické Markovovo modelování navrhl Horspool a Cormack a je kombinací Markovova modelu s aritmetickým kódováním. Začíná s jednoduchým počátečním konečným automatem a přidává do něho nové stavy pomocí vytváření kopií stavů. K přechodům v konečném automatu jsou přidány čítače frekvencí. Tyto čítače ukazují, kolikrát byl příslušný přechod použit. Ukažme vytváření kopií stavů na příkladu.

Příklad 6.19:

Na obr. 6.22 je uveden fragment konečného automatu. Předpokládejme, že přechod ze stavu u do stavu t má vysokou hodnotu čítače frekvencí. Pro stav t se vytvoří jeho kopie t'. Přechod ze stavu u do stavu t je přesměrován do stavu t' přičemž ostatní přechody do stavu t zůstanou stejné.

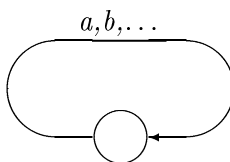


Obr. 6.22: Fragment konečného automatu



Obr. 6.23: Konečný automat pro vytvoření kopie stavu

Dále se vytvoří přechody ze stavu t' do stavů x a y. Výsledný automat je na obr. 6.23.



Obr. 6.24: Počáteční jednostavový model

Poslední otázka se týká úpravy čítačů frekvencí pro nově vytvořené přechody. Čítač frekvence přechodu ze stavu u do t' zůstane stejný jako dříve. Čítače frekvencí přechodů ze stavu t' do stavů x a y jsou proporciálně rozděleny podle čítačů přechodů ze stavu u do t' a ze stavů v a w do stavu t.

Jako iniciální modely mohou být použity různé konečné automaty. Nejjednodušší z nich je jednostavový automat podle obr. 6.24. Existují však jiné modely jako řady, stromy, pletence. Avšak je dokázáno, že bez ohledu na použitý iniciální model jsou všechny dynamické Markovovy modely modely s konečným kontextem.

7 Kontrola správnosti textu

Při přípravě textů v živých jazycích pomocí počítače s použitím nejrůznějších textových systémů se velice brzy objevila otázka možnosti automatické kontroly správnosti textu. Začaly proto vznikat různé systémy na takovou kontrolu. Problém, který kontrolní systém řeší, je velmi jednoduchý. Je dán textový soubor a je třeba určit slova, která jsou nesprávná.

Každé slovo je možno považovat za bod ve vícerozměrném prostoru posloupností písmen. Ne každá posloupnost písmen je správné slovo a kontrolní systém musí určit, zda posloupnost písmen je nebo není správné slovo. Přitom je třeba minimalizovat tyto dva druhy chyb:

1. indikace, že správné slovo je nesprávné,
2. indikace, že nesprávné slovo je správné.

Po nalezení nesprávného slova je možno v prostoru slov hledat nejbližšího souseda, nejbližší správné slovo.

Při implementaci tohoto systému je třeba vyřešit otázku: Co je slovo? Odpověď se zdá jednoduchá: Je to posloupnost písmen oddělená omezovači. Omezovače jsou: mezery, čárky, tečky, dvojtečky, středníky, vykřičníky, otazníky atd. Většinou je zřejmé, co lze považovat za písmeno a co za omezovač. Musíme však uvážit, kam zařadit číslice, pomlčky a apostrofy.

Posloupnosti znaků, které obsahují číslice, se obvykle neuvažují. To se týká jak čísel, tak posloupností, ve kterých se vyskytují číslice. Rozhodnutí, zda takové posloupnosti jsou slova nebo ne, je nejlépe ponechat uživateli.

Pomlčky se obvykle považují za omezovače. V mnoha jazycích se totiž používají k vytváření složených slov (např. v angličtině context-free) nebo k oddělení částic (v češtině -li) apod. Dále se pomlčky používají na rozdělování slov na konci řádku. Tento druh použití pomlček je vhodné zakázat, protože jej není obecně možné rozlišit od jiných případů použití pomlček.

Zařazení apostrofů mezi písmena nebo omezovače je závislé na použitém jazyku. V angličtině se apostrofy používají jako součásti slov (don't, I'd, King's, kids'). V češtině se apostrofy v této funkci nepoužívají. Z toho plyne, že pro angličtinu je apostrof písmeno, pro češtinu omezovač.

Další problém je otázka malých a velkých písmen. Většina systémů automaticky převádí písmena na jeden typ (malá nebo velká). Tento postup může ale narazit na některé problémy. Jedním z problémových případů jsou jména. Jestliže např. jméno Richta převedeme na malá písmena, dostaneme slovo richta, které bude považováno za chybné, a nejbližší správné slovo bude rychta. Dalším problémem jsou zejména v počítačové literatuře, slova, která se vždy píší velkými písmeny (např. FORTRAN), a zkratky (např. IBM). Jednou z možností řešení tohoto problému je vynechat tato slova při kontrole, protože jsou to jména nebo obecně užívané zkratky.

Výzkum v této oblasti byl zahájen přibližně v roce 1957, ale první systém na kontrolu textu byl systém SPELL pro počítač DEC-10 vytvořený v roce 1971. Tento systém a jeho modifikace jsou používány dodnes.

7.1 Kontrola textu pomocí frekvenčního slovníku

Nejjednodušší princip kontroly správnosti textu je založen na vytvoření jeho frekvenčního slovníku. Postup je velice jednoduchý:

1. Vytvoříme frekvenční slovník, což je posloupnost dvojic (slovo, počet výskytů).
2. Frekvenční slovník seřadíme podle počtu výskytů v pořadí od nejméně častých slov k nejčastějším.
3. Na začátku takto setříděného frekvenčního slovníku najdeme slova:
   a) která jsou nesprávná,
   b) která jsou velice řídká.

Počáteční část frekvenčního slovníku prohlédneme a určíme nesprávná slova.

Metoda kontroly textu pomocí frekvenčního slovníku má dvě základní nevýhody:

1. vyžaduje prohlížení frekvenčního slovníku,
2. vede k problému při odhalování systematických chyb.

Přesto se v literatuře uvádí, že tento typ kontroly byl použit při přípravě knihy Computer Organization and Assembly Language Programming (Academic Press, 1978). V této knize bylo použito 156134 slov (5649 různých slov) a výsledkem bylo, že v ní není žádná (známá) chyba.

7.2 Kontrola textu pomocí dvojitého slovníku

Použití slovníku správných slov je další stupeň při konstrukci kontrolního systému. Je zřejmé, že takový slovník je značně rozsáhlý a proto není možné postupovat tak, že by se každé slovo kontrolovaného textu hledalo ve slovníku. Proto se postupuje takto:

1. Vytvoříme seznam všech různých slov v kontrolovaném textu (slovník textu).
2. Seřadíme tento seznam podle abecedy (předpokládáme, že tak je seřazen i slovník).
3. Každé slovo v setříděném seznamu hledáme ve slovníku:
   a) jestliže je slovo nalezeno, je správné,
   b) jestliže není slovo nalezeno, pak je vloženo do výstupního seznamu.
4. Vytiskneme seznam slov, která nebyla nalezena ve slovníku.

Tento postup vyžaduje pouze jeden průchod slovníkem a proto je podstatně rychlejší než hledání každého kontrolovaného slova ve slovníku. Největší část tohoto postupu je slovník. Je zřejmé, že ruční vytvoření slovníku je náročná a rozsáhlá práce. Slovník je možno vytvořit automaticky takto:

1. Začneme provádět kontrolu s prázdným slovníkem.
2. Správná slova se objeví jako součást výstupu (viz bod 4. kontrolního algoritmu).
3. Z výstupního seznamu vynecháme chybná slova a vlastní jména.
4. Přidáme správná slova do slovníku.

Tímto postupem je možno vytvořit slovník, aniž bychom napsali jediné slovo.

V této souvislosti je třeba uvážit rozsah slovníku. Pro jednoho uživatele nebo pro malou skupinu uživatelů stačí slovník obsahující 10 000 slov. Velký slovník, řekněme 100 000 slov, jednak zpomaluje kontrolu a pak také svádí ke vkládání archaických, nepříliš frekventovaných a také nesprávných slov. Pro větší skupinu uživatelů je nutno zřídit funkci administrátora slovníku, který smí vkládat nová slova a vynechávat slova užívaná velice zřídka. Jedním z přístupů, jak udržet velikost slovníku v přijatelných mezích i pro větší skupinu uživatelů je vytvoření několika dílčích slovníků. Jeden z nich tvoří společný základ a ostatní speciální slovníky pokrývají slovní zásobu v určitém oboru. Při kontrole textového souboru se vytvoří pracovní slovník spojením základního slovníku a jednoho či několika speciálních slovníků. Speciální slovníky mohou být udržovány administrátorem slovníku nebo přímo uživateli.

Právě popsaný postup kontroly textu pomocí dvojitého slovníku má dvě nevýhody. Předně se jedná o operaci, která je časově náročná a nehodí se pro interaktivní použití. Druhou nevýhodou je ztráta kontextu. Výsledkem práce je seznam slov, která nejsou ve slovníku, bez jakékoliv informace o jejich umístění v kontrolovaném souboru. Proto je nutno chybná slova hledat v souboru pomocí editoru a přitom není možno použít operaci nahraď řetězec řetězcem, protože v textovém souboru jsou slova zapsána s použitím malých i velkých písmen a dále chybné slovo může být částí správných slov.

7.3 Interaktivní kontrola textu

Pro interaktivní způsob práce je nutno použít tento postup:

1. Zadání speciálních slovníků, které budou používány.
2. Každé slovo souboru je hledáno ve slovníku. Když není nalezeno, uživatel dostane dotaz, co udělat.

Tento systém může provádět například tyto operace:

1. **Nahraď:** Neznámé slovo je vynecháno ze souboru a uživatel je dotazován na správné slovo.
2. **Nahraď a zapamatuj:** Neznámé slovo je nahrazeno správným, které zadal uživatel, a všechna další použití tohoto slova jsou také nahrazena novým slovem.
3. **Ponech:** Neznámé slovo je považováno za správné (v daném kontextu) a je ponecháno.
4. **Ponech a zapamatuj:** Neznámé slovo je ponecháno v textu a všechna jeho další použití jsou považována za správná.
5. **Editace:** Přechod do stavu, kdy je možno soubor editovat.

Je zřejmé, že tento systém umožňuje ukázat kontext a místo v souboru, kde se neznámé slovo vyskytuje, a dovoluje provést okamžitě případnou opravu. Problém, který není řešen, je časová náročnost operace vyhledávání každého slova ve slovníku. V tomto případě jsou požadavky na systém ještě větší oproti principu založeném na dvojitém slovníku, protože každé slovo kontrolovaného textu je hledáno ve slovníku. Proto je základním požadavkem taková organizace slovníku, která dovoluje velice rychlé vyhledávání. Uveďme některé možné organizace slovníku.

- **Rozptýlení:** Program SPELL používá rozptylovou funkci

  $$h(L_1 L_2 \dots L_N) = (L_1 * 26 + L_2) * 10 + min(N-2,\ 9)$$

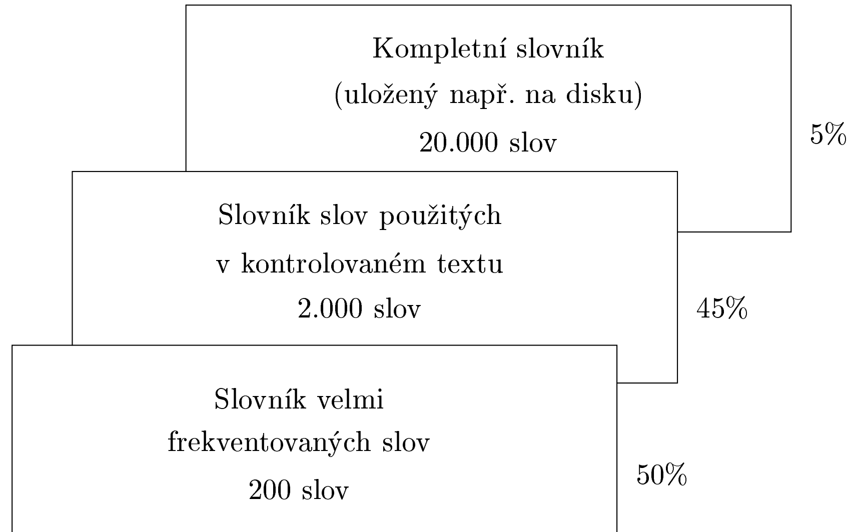
  Tato funkce má stejnou hodnotu pro všechna slova, která mají stejná první dvě písmena a stejnou délku.

- **Stromy:** Z kořene stromu vychází 26 hran odpovídajících jednotlivým písmenům do 26 uzlů. Z každého uzlu pak vychází hrany podle možných druhých písmen správných slov. Z dalších uzlů vychází hrany podle třetích písmen atd. Zvláštním bitem v uzlu je označen možný konec správného slova. Tento způsob vyžaduje pro slovo o délce N průchod stromem o N úrovní a test bitu "konec slova".

- **Použití hierarchického slovníku:** Tato metoda je založena na zjištění, že malý počet slov je v textu používán s velmi vysokou frekvencí. Pro angličtinu například platí, že slovník o 134 slovech stačí ke kontrole 50% slov v textu. Dále bylo zjištěno, že v určitém textovém souboru není použita celá slovní zásoba, ale stačí poměrně malý slovník. Z těchto úvah plyne výhodnost tříúrovňového systému podle obr. 7.1, kde jsou uvedeny rozsahy slovníků a frekvence slov vyhledávaných v jednotlivých slovnících. Hledání v tomto tříúrovňovém slovníku se provádí takto:

  1. Prohledávání malého slovníku velmi frekventovaných slov. Jestliže slovo není nalezeno, přejdi na 2., jinak konec.

2. Prohledávání slovníku slov, jiř dříve v textu použitých. Jestliže slovo není nalezeno, přejdi na 3., jinak konec.
3. Prohledání kompletního slovníku. Jestliže je slovo nalezeno, přidej ho do slovníku slov použitých v textu a konec. Jestliže slovo není nalezeno, jedná se o chybné slovo.



Obr. 7.1: Hierarchické uspořádání slovníku

Vzhledem k různé frekvenci prohledávání jednotlivých slovníků je možno volit vhodné datové struktury umožňující rychlé vyhledávání při přiměřených nárocích na paměť.

7.4 Kontrola textu založená na pravidelnosti slov

Metoda založená na pravidelnosti (regularitě) slov vychází z výzkumu frekvence výskytu dvojic písmen (digramů) a trojic písmen (trigramů) v jazykovém textu. Jestliže máme abecedu o 28 znacích (písmena, mezera, pomlčka), pak existuje 28^2(=784) digramů a 28^3(=21952) trigramů. Frekvence těchto skupin písmen je ovšem menší. V anglickém textu se používá jen asi 70%(=550) digramů a 25%(=5000) trigramů, s velice různou frekvencí. Jestliže slovo obsahuje několik velice řídkých digramů nebo trigramů, je pravděpodobně chybné. Na této skutečnosti je založena celá řada metod kontroly textu.

Zde uvedeme metodu založenou na „koeficientu podivnosti". Koeficient podivnosti trigramu xyz závisí na relativní frekvenci digramů $f(xy)$ a $f(yz)$ a trigramu $f(xyz)$ takto:

$$KPT = [log(f(xy) - 1) + log(f(yz) - 1)]/2 - log(f(xyz) - 1)$$

Poznámka: log(0) je definován pro tuto funkci jako -10.

Pro slova se koeficient podivnosti spočítá jako

$$KPS = \sqrt{\sum_{i=1}^{n} (KPT_i - SKPT)^2} \ ,$$

kde KPT_i je koeficient podivnosti i-tého trigramu a $STPT$ je střední hodnota koeficientu podivnosti všech trigramů obsažených ve slově. Pro slova, která jsou chybná, vychází tento koeficient vysoký a proto stačí setřídit slova podle koeficientu podivnosti a na začátku budou slova s velkou pravděpodobností chyb.

Literatura

[Adam89] Adámek, J.: *Kódování.* SNTL – Státní nakladatelství technické literatury, Praha, 1989.

[Aho74] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[Aho75] Aho, A.V., Corasick, M.J.: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, Vol.18, No.6., June 1975, pp.333-340.

[Aho79] Aho, A.: *Pattern matching in strings.* Proceeding of the Symposium on Formal Language Theory, University of Santa Barbara, December 1979, pp.325-347.

[Bell91] Bell, T.C., Cleary, J.G., Witten, I.H.: *Text compression.* Prentice Hall, Englewood Cliffs, N.J., 1991.

[Bent86] Bentley, J.L., Sleator, D.D., Tarjan, R.E., Wei,W.K.: *A locally adaptive data compression scheme.* Communications of the ACM, Vol.29, No.4, April 1986.

[Blo86] Bloch, M., Scheber, A.: *Textové bázy dát.* Sborník referátů semináře SOFSEM'86, Liptovský Ján, ÚVT UJEP Brno a JČMF, 1986, pp.47-80.

[Book90] Bookstein, A., Klein, S.T.: *Compression, Information Theory, and Grammars: A Unified Approach.* ACM Transactions on Information Systems, Vol.8, No.1, Jan. 1990, pp.27 – 49.

[Boye77] Boyer, R.S., Moore, J.S.: *A fast string searching algorithm.* Communications of the ACM, Vol.20, No.10, Oct.1977, pp.262-272.

[Comm79] Commentz-Walter, B.: *A string matching algorithm fast on the average.* Proceedings of the 6th International Colloquium on Automata, Languages and Programming, Springer-Verlag, 1979, pp.118-132.

[Corm84] Cormack, G.V., Horspool, R.N.S.: *Algorithms for adaptive Huffman codes.* Information Processing Letters, Vol.18, No.3, March 1984, pp.159 – 165.

[Corm87] Cormack, G.V., Horspool, R.N.S.: *Data Compression using dynamic Markov modelling.* The Computer Journal, Vol.30, No.6, 1987, pp.541 – 550.

[Elia75] Elias, P.: *Universal codeword sets and representations of the integers.* IEEE Transactions on Information Theory, Vol. IT–21, No.2, March 1975, pp.194 – 203.

[Elia87] Elias, P.: *Interval and recency rank source coding: two on-line adaptive variable-length schemes.* IEEE Trans. Inf. Theory IT–33, 1, 1987, pp.3 – 10.

[Even78] Even, S., Rodeh, M.: *Economical encoding of commas between strings.* Communications of the ACM, Vol.21, No.4, April 1978, pp.315 – 317.

[Fano49] Fano, R.M.: *The transmission of information.* Tech. Rep. 65, Research Laboratory of Electronics,MIT, Cambridge, MA. 1949.

[Fial89] Fiala, E.R., Greene, D.H.: *Data compression with finite windows.* Communications of the ACM, Vol.32, No.4, Apr.1989, pp.490 – 505.

[Gall78] Gallager, R.G.: *Variation on a theme by Huffman.* IEEE Transactions on Information Theory, Vol.IT-24, No.6, Nov.1978, pp.668 – 674.

[Held83] Held, G.: *Data compression.* John Wiley & Sons, Chichester, 1983.

[Hopk92] Hopkins, L.: *SQL∗Text-Retrieval.* Administrator's Guide, Version 2.0, Oracle Corporation, 1992.

[Huff52] Huffman, D.A.: *A method for the construction of minimum-redundancy codes.* Proceedings of IRE, Vol.40, No.9, Sept.1952, pp.1098 – 1101. Also in: Davisson, L.D., Gray, R.M. ed.: *Data compression.* Dowden, Hutchinson, Strondsburg, Pennsylvania, USA, 1976.

[Jak82] Jakobsson, M.: *Evalutation of a hierarchical bit-vector compression technique.* Information Processing Letters, Vol.14, No.4, June 1982, pp.147 – 149.

[Jak85] Jakobsson, M.: *Compression of character strings by an adaptive dictionary.* BIT, Vol.25, 4, pp.593 – 603.

[Knut85] Knuth, D.E.: *Dynamic Huffman coding.* Journal of Algorithms, Vol.6, 1985, pp.163 – 180.

[Knut77] Knuth, D.E., Morris, J.H., Pratt, V.R.: *Fast pattern matching in strings.* SIAM Journal of Computing, Vol.6, No.2, 1977, pp.323 – 350.

[Lang84] Langdon, G.G.: *An introduction to arithmetic coding.* IBM Journal of Research and Developement, Vol.28, No.2, March 1984, pp.135 – 149.

[Lel87] Lelewer, D.A., Hirschberg, D.S.: *Data compression.* ACM Computing Surveys, Vol.19, No.3, Sept.1987, pp. 261 – 296.

[Lemp76] Lempel, A., Ziv, J.: *On the complexity of finite sequences.* IEEE Transactions on Information Theory, Vol. IT–22, No.1, Jan.1976, pp.75 – 81.

[Mel91] Melichar, B.: *Informační systémy.* Ediční středisko ČVUT, Praha, 1991, 57 str.

[Mel92] Melichar, B., Pokorný, J.: *Data Compression.* Survey Report DC–92–04, Czech Technical University, Department of Computers, Praha, 1992, 55 str.

[Pelt91] Peltola, H., Tarhio, L.: *On syntactical data compression.* In: Symp. on Progr. Lang. and Software Tools, Pirkkala, 1991.

[Pok87] Pokorný, J.: *Komprese dat.* Proc. of Sem. DATASEM'87, DT ČSVTS, Praha, 1987, pp.43 – 54.

[Riss79] Rissanen, J., Langdon, G.G.: *Arithmetic coding.* IBM Journal of Research and Development, Vol.23, No.2, March 1979, pp.149 – 162.

[Salt83] Salton, G., McGill, M.J.: *Introduction to modern information retrieval.* McGraw-Hill, Tokyo, 1983.

[Stor88] Storer ,J.: *Data Compression: Methods and Theory.* Computer Science Press, Rockwille, 1988.

[Vitt87] Vitter, J.S.: *Design and analysis of dynamic Huffman codes.* Journal of the ACM, Vol.34, No.4, Oct.1987, pp.825 – 845.

[Vitt89] Vitter, J.S.: *ALGORITHM 673: Dynamic Huffman coding.* ACM Transactions on Mathematical Software, Vol.15, No.2, June 1989, pp.158 – 167.

[Welc84] Welch, T.A.: *A technique for high-performance data compression.* COMPUTER, Vol.17, No.6, June 1984, pp.8 – 19.

[Witt87] Witten, I.H., Neal, R.M., Cleary, J.G.: *Arithmetic coding for data compression.* Communications of the ACM, Vol.30, No.6, June 1987, pp.520 – 540.

[Ziv77] Ziv, J., Lempel, A.: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, Vol.IT–23, No.3, May 1977, pp.337 – 343.

[Ziv78] Ziv, J., Lempel, A.: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, Vol.IT–24, No.5, Sept.1978, pp.530 – 536.