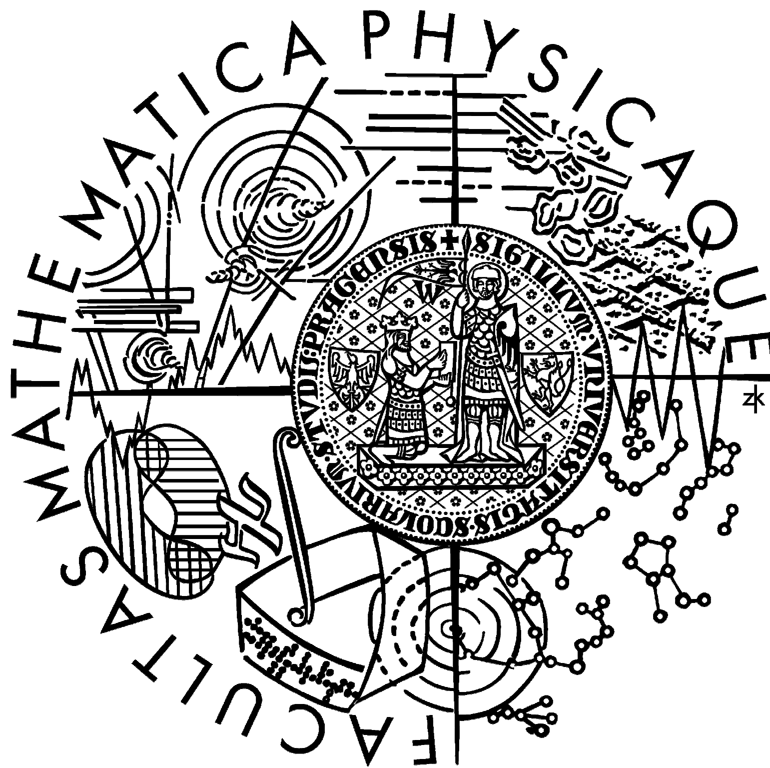
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Stanislav Kovalčin

Slovníkové metody jako druhá fáze BWT

Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Jan Lánsky

Studijní program: Informatika, Softwarové systémy

Na tomto mieste by som chcel poďakovať najmä svojmu vedúcemu diplomovej práce Mgr. Janovi Lánskemu za podnetné rady a námety pri tvorbe a zdokonaľovaní tejto diplomovej práce.

Chcel by som takisto poďakovať RNDr. Michalovi Zemličkovi, Ph.D. za uvedenie do problematiky kompresie na ním prednášaných predmetoch a za kvalitný základ, ktorý som tak získal.

Ďalej by som chcel poďakovať RNDr. Tomášovi Dvořákovi, CSc. za podnetné prednášky na predmete Algoritmy a komprese dat.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 8.8.2007 Stanislav Kovalčin

Název práce: Slovníkové metody jako druhá fáze BWT
Autor: Stanislav Kovalčin
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Jan Lánsky
e-mail vedoucího: jan.lansky@mff.cuni.cz
Abstrakt: Burrows-Wheelerova transformace je jeden z nejoblíbenějších algoritmů používaných při bezztrátové kompresi dat. Druhá fáze obvykle pozůstává z kombinace algoritmů Move-to-front, Run-length encoding a bývá zapsaná Huffmanovým nebo aritmetickým kódováním. Jiná skupina algoritmů pro bezztrátovou kompresi dat používá slovníkové metody prostřednictvím algoritmů rodiny LZ. Tato diplomová práce experimentálně testuje vhodnost zapojení vybraných slovníkových metod (LZC, LZSS) do druhé fáze Burrows-Wheelerove transformace, nejen nad abecedou znaků a slov, ale i slabik. Tato vhodnost je testovaná i na velkých XML souborech. Je proto vhodné navrhnout modifikaci algoritmů druhé fáze Burrows-Wheelerove transformace pro velké abecedy. Je uvedené porovnání kompresního poměru s programy, které využívají Burrows-Wheelerove transformace nejen nad velkými XML soubory ale i nad Calgary korpusem.
Klíčová slova: komprese textu, slovníkové metody, Burrows-Wheelerova transformace, dělení na slabiky, XML

Title: Dictionary methods as second phase of BWT
Author: Stanislav Kovalčin
Department: Department of software engineering
Supervisor: Mgr. Jan Lánsky
Supervisor’s e-mail address: jan.lansky@mff.cuni.cz
Abstract: Burrows-Wheeler transform is one of the most favorite lossless data compression algorithm. Second phase of Burrows-Wheeler transform consists of combination of Move-to-front, Run-length encoding algorithm and used to be written by Huffman or arithmetic encoding. Dictionary methods are used by means of LZ family algorithm in another lossless data compression algorithm group. This master thesis is experimentally testing suitability of integration selected dictionary methods (LZC, LZSS) in second phase of Burrows-Wheeler transform, not only over alphabet of symbols and words, but also over alphabet of syllables. This suitability is tested likewise on large XML files. It is appropriate to propose modification of Burrows-Wheeler second phase’s algorithms for large alphabets. Comparation of compression ratios not only over large XML files, but also over Calgary corpus with others programs using Burrows-Wheeler transform is presented.
Keywords: text compression, dictionary methods, Burrows-Wheeler transform, syllable-based partition, XML

Obsah

1. Bezstrátová kompresia ... 6
  1.1. Slabiky a veľká abeceda ... 7
  1.2. Testovacie množiny dokumentov ... 8
    1.2.1. Calgary korpus ... 8
    1.2.2. Korpus XMLcz ... 9
    1.2.3 Výsledky nad korpusmi ... 9
2. Abeceda ... 12
3. Algoritmy block-sorting kompresie ... 13
  3.1. Burrows-Wheelerova transformácia ... 13
  3.2. Alternatíva pre Burrows-Wheelerovu transformáciu ... 15
    3.2.1. Počiatočný úsek kontextu dĺžky 0 ... 16
    3.2.2. Počiatočný úsek kontextu dĺžky 1 ... 16
    3.2.3. Počiatočný úsek kontextu dĺžky n ... 17
4. Algoritmy po Burrows-Wheelerovej transformácii ... 19
  4.1. Algoritmus Move-to-front ... 19
  4.2. Alternatívy pre algoritmus Move-to-front ... 20
  4.3. Algoritmus Move-to-front pre veľké abecedy ... 22
    4.3.1. Splay tree ... 23
    4.3.2. Splay tree implementácia pre Move-to-front ... 25
    4.3.3. Výsledky kompresie nad korpusmi pre algoritmus Move-to-front ... 30
  4.4. Algoritmus Move-to-front-1 ... 32
    4.4.1. Splay tree implementácia pre algoritmus Move-to-front-1 ... 33
    4.4.2. Výsledky kompresie nad korpusmi pre algoritmus Move-to-front-1 ... 36
  4.5. Move-to-front-2 ... 37
    4.5.1. Výsledky kompresie nad korpusmi pre algoritmus Move-to-front-2 ... 38
  4.6. Dopad algoritmu Move-to-front a jeho variánt na komprimáciu ... 39
5. Run-length encoding ... 42
  5.1. Run-length encoding 1 ... 42
    5.1.1. Výsledky kompresie nad korpusmi pre algoritmus RLE-1 ... 45
  5.2. Run-length encoding 2 ... 47
    5.2.1. Výsledky kompresie nad korpusmi pre algoritmus RLE-2 ... 50
  5.3. Run-length encoding 3 ... 51
    5.2.1. Výsledky kompresie nad korpusmi pre algoritmus RLE-3 ... 54
  5.4. Run-length encoding EXP ... 56
  5.5. Run-length encoding BIT ... 57
  5.6. Dopad variánt algoritmu Run-length encoding na komprimáciu ... 58
6. Slovníkové algoritmy ... 61
  6.1. Algoritmus LZ78 ... 62
    6.1.1. Algoritmus LZC ... 66

6.1.3. Výsledky kompresie nad korpusmi pre algoritmus LZC....................................... 70
6.2. Algoritmus LZ77 72
6.2.1. Algoritmus LZSS... 75
6.2.2. Výsledky kompresie nad korpusmi pre algoritmus LZSS..................................... 78
6.3. Dopad slovníkových algoritmov LZC a LZSS na komprimáciu.................................. 79
7. Programy založené na Burrows-Wheelerovej transformácii... 81
7.1. Bzip2.. 81
7.2. ABC 83
7.3. Szip 84
8. Vyhodnotenie výsledkov 85
8.1. Možnosti ďalšieho vývoja 86
9. Literatúra 88

1. Bezstrátová kompresia

Bezstrátová kompresia je trieda kompresných algoritmov, ktoré dovoľujú presne zrekonštruovať pôvodné dáta zo skomprimovaných dát. Má široké použitie pre dáta, ktoré potrebujú byť po odkomprimovaní rovnaké ako pred skomprimovaním. Medzi takéto dáta patria napríklad texty, knihy, spustiteľné súbory, databázy, tabuľky a iné. Bezstrátovosť kompresie je často požadovaná, a preto sú algoritmy patriace do tejto triedy široko používané.

Momentálne používanými metódami pre bezstratovú kompresiu, obzvlášť pre kompresiu textov sú algoritmy postavené na princípe kompresných algoritmov rodiny LZ publikované Lempelom a Zivom. Základ tejto rodinu tvorí algoritmus LZ77 publikovaný Lempelom a Zivom [1] v roku 1977 a algoritmus LZ78 publikovaný Lempelom a Zivom [2] v roku 1978. Ďalšie algoritmy tvoriace túto LZ rodinu sú odvodeninami týchto dvoch algoritmov.

Medzi ďalšie metódy pre bezstrátovú komprimáciu patrí algoritmus Prediction by Partial Matching (PPM) publikovaný Bellom a kol. [3] v roku 1984 a jej ďalšie varianty. Používa adaptivnú štatistickú kompresnú metódu založenú na modelovaní kontextov a predpovedi nasledujúceho symbolu v texte. Na túto predpoveď používa množinu symbolov, ktoré už boli spracované. Na podobnom princípe pracuje algoritmus Dynamic Markov Compression publikovaný Cormackom a Horspoolom [9] v roku 1987.

V roku 1994 Burrows a Wheeler [4] publikovali novú metódu založenú na operáciach triedenia, ktoré v zhlukujú k sebe symboly, ktoré sa vyskytujú v podobnom kontexte v pôvodnom texte. Táto metóda má názov Burrows-Wheelerova transformácia a stala sa obľúbená pre veľmi dobrý pomer medzi kompresným pomerom a časom.

Medzi symbolmi sa po Burrows-Wheelerovej transformácii vytvára súvislosť, ktorá súvisí s ich zhlukovaním. Túto súvislosť využíva Move-to-front transformácia publikovaná v 1986 Bentleyom a kol. [7], jej varianty a podobné algoritmy. Po tejto transformácii často vznikajú dlhé behy symbolov, ktoré sú následne komprimované algoritmom Run-length encoding (RLE) alebo jeho variantami. Pre výstup sú používané rôzne binárne kódery, medzi najčastejšie patria Huffmanov kóder, založený na práci Huffmana [8] z roku 1952 alebo aritmetický kóder.

1.1. Slabiky a veľká abeceda

V súčasnosti sa využívajú dva základné metódy kompresie textu po symboloch a síce v prvom prípade sa za symbol berú jednotlivé znaky a v druhom prípade sa za symbol berú celé slová. Ako ukázal Lánsky [10], má zmysel rozšíriť chápanie symbolu na úroveň slabík. Chápanie symbolu na úrovni slabík sa zdá byť výhodné pre tvaroslovne bohaté jazyky akými sú napríklad slovenština, čeština, polština a iné.

Veľkosť vytváranej abecedy závisí na ponímaní symbolu. Pre znaky dostávame malé abecedy. Pri slabikách závisí veľkosť abecedy od tvaroslovnej bohatosti jazyka a pri slovách dostávame veľké abecedy. Pri chápaní symbolov ako slabík alebo slov je potrebné mať k symbolu v abecede potrebný prevod na slabiku alebo slovo, ktoré reprezentuje. Na to potrebujeme mať nejakú tabuľku alebo slovník, v ktorých budeme mať tieto prevody zaznamenané.

Veľa kompresných programov zameraných na text implementuje Burrows–Wheelerovu transformáciu kvôli dobrým vlastnostiam transformovaného reťazca. Ďalšia skupina kompresných programov používa slovníkové metódy prostredníctvom algoritmov rodiny LZ. Posledná skupina využíva štatistické kompresné metódy a algoritmy založené na algoritme PPM alebo DMC. Výstup z každej zo spomenutých metód je v ďalšom kroku zakódovaná vhodným binárnym kóderom.

Kvôli lepšiemu kompresnému pomeru sa vstupný text upravuje, aby sme ho dostali do čo najlepšie skomprimovateľnej formy. Zvolenie takéhoto postupu je omnoho efektívnejšie ako jednoduchá aplikácia jedného kompresného algoritmu, pretože použité transformácie upravujú text takým spôsobom, aby sa pri použití nasledujúcich kompresných algoritmov dosiahla čo najväčšia efektivita. Preto sa programy využívajúce Burrows–Wheelerovu transformáciu odlišujú tým, ako je spracovávaný text v jednotlivých krokoch a aké transformácie a algoritmy sú na neho použité. Spomeniem tri z nich a síce bzip2, ABC a szip.

Na testovanie rôznych použitých variánt bol v tejto diplomovej práci použitý projekt XBW, v ktorom som časti popísané v ďalších kapitolách implementoval. Konkrétne ide o algoritmus Move-to-front a jeho varianty, algoritmus Run-length encoding a jeho varianty a slovníkové metódy.

1.2. Testovacie množiny dokumentov

Testovanie výkonnosti kompresných algoritmov je veľmi špecifická záležitosť. Každý algoritmus má vstupy, na ktorých dosahuje lepší kompresný pomer a vstupy, na ktorých dosahuje horší kompresný pomer. Preto sa výkonnosť kompresných algoritmov meria na množine pevne daných súborov, a všetky algoritmy sa porovnávajú vzhľadom na ňu.

Pri našich meraniach sme použili dva rôzne korpusy, Calgary korpus je všeobecne zameraný a predstavuje štandart pre posudzovanie kompresného pomeru a XML_{cz} korpus je korpus, na ktorom sa prejaví najlepší kompresný pomer programu XBW. Oba korpusy sa nachádzajú na priloženom CD.

1.2.1. Calgary korpus

Ako prvý bol použitý Calgary korpus. Bol vytvorený Ianom Wittenom a Timom Bellom v roku 1989 a dlho bol uznávaný a používaný ako štandardný korpus na posudzovanie kompresného pomeru programov na kompresiu dát. Pri posudzovaní kompresného pomeru sa stále udávajú aj výsledky na tomto korpuse. Obsahuje týchto 18 súborov:

Názov súboru	Kategória	Veľkosť v bitoch
bib	Bibliografia	111261
book1	Beletria	768771
book2	Náučná literatúra (formátovaný dokument)	610856
geo	Geofyzikálne dáta	102400
news	USENET batch súbor	377109
obj1	Objektový kód pre VAX	21504
obj2	Objektový kód pre Apple Mac	246814
paper1	Odborný článok	53161
paper2	Odborný článok	82199
paper3	Odborný článok	46526
paper4	Odborný článok	13286
paper5	Odborný článok	11954
paper6	Odborný článok	38105
pic	Čiernobiely faxový obrázok	513216
progc	Zdrojový kód v “C”	39611
progl	Zdrojový kód v LISPe	71646
progp	Zdrojový kód v PASCALe	49379
trans	Prepis terminálovej relácie	93695

Tabuľka 1: Súbory obsiahnuté v Calgary korpuse, ich kategória a veľkosť v bitoch.

1.2.2. Korpus XMLcz

Projekt XBW bol od začiatku projektovaný na veľké XML súbory, v ktorých by mal dosahovať najlepší kompresný pomer. Preto bol účinok jednotlivých algoritmov a kompresný pomer meraný aj na množine veľkých XML súborov. Tieto súbory boli stiahnuté ako obraz celej webovej stránky, na ktorej sa používa čeština, teda stránky inštitúcií, poprípade diskusií a podobne. Následne boli uložené v XML súboroch. Stiahnutie stránky a uloženie do XML súboru bolo prevedené systémom Egothor.

XML súbory v korpuse XML_{cz} sa vyznačujú svojou veľkosťou, ktorá je priemerne okolo 20 MB na jeden súbor. Tieto XML súbory by mali predstavovať výhodu pre kompresný program projektu XBW. Vzhľadom na to, že dané súbory obsahujú češtinu, ktorá je tvaroslovne bohatá a je pre ňu vhodnejšia slabiková kompresia ako uvádza Lánsky [10].Pri takýchto veľkých súboroch takisto vzniká aj väčšia abeceda. V korpuse veľkých XML súborov v češtine sa nachádza týchto 10 súborov:

Názov súboru	Zdroj	Veľkosť v bitoch
998001.txt	www.jakpsatweb.cz	24603833
998002.txt	www.scientia.cz	23229720
998003.txt	www.vareni.cz	20148878
998004.txt	www.shop-list.cz	17795800
998005.txt	www.zaket.cz	18984137
998006.txt	www.baseballnet.cz	19452095
998007.txt	www.mojenoviny.cz	19316662
998008.txt	www.cksatur.cz	19663942
998009.txt	www.carlsbadtaxi.cz	20140780
998010.txt	www.uamk.cz	19100318

Tabuľka 2 : Súbory obsiahnuté v korpuse veľkých XML v češtine, zdroj ich stiahnutia a veľkosť v bitoch.

1.2.3 Výsledky nad korpusmi

Pre každý spomenutý algoritmus, ktorý je implementovaný v programe XBW je prítomná tabuľka zobrazujúca použité algoritmy a parsovanie pre parser. V prípade parsera sú parsovania rozdelené podľa typu a spôsobu parsovania.

Typ parsovania charakterizuje typ vstupného súboru a môže nadobúdať tieto hodnoty:

- off – nie je použitý parser, vstupný súbor je načítavaný do dátového typu char

- xml – parser očakáva na vstupe súbor s XML štruktúrou
- text – súbor očakáva na vstupe textový súbor, neočakáva žiadnu štruktúru

Spôsob parsovania charakterizuje spôsob parsovania vstupného súboru na symboly abecedy a môže nadobúdať tieto hodnoty:

- Znak – symbol abecedy je chápany ako znak, a preto je vstupný súbor parsovaný po znakoch
- Slabika – symbol abecedy je chápaný ako slabika, a preto je vstupný súbor parsovaný po slabikách, podľa definície ako uvádza Lánsky [10]
- Slovo – symbol abecedy je chápaný ako slovo, a preto je vstupný súbor parsovaný po slovách

Môžeme zvoliť varianty pre tri druhy algoritmov a síce Move-to-front, Run-length encoding a slovníkové metódy. Varianty pre algoritmus Move-to-front môžu nadobúdať tieto hodnoty:

- off – pri komprimovaní je vynechaný algoritmus Move-to-front a jeho varianty
- MTF – algoritmus Move-to-front pre veľké abecedy
- MTF-1 – algoritmus Move-to-front-1 pre veľké abecedy, ktorý je odvodený od algoritmu Move-to-front
- MTF-2 – algoritmus Move-to-front-2 pre veľké abecedy, ktorý je odvodený od algoritmu Move-to-front

Varianty pre algoritmus Run-length encoding môžu nadobúdať tieto hodnoty:

- off – pri komprimovaní je vynechaný algoritmus Run-length encoding a jeho varianty
- RLE-1 – prvá varianta algoritmu Run-length encoding
- RLE-2 – druhá varianta algoritmu Run-length encoding
- RLE-3 – tretia varianta algoritmu Run-length encoding

Slovníkové metódy môžu nadobúdať tieto hodnoty:

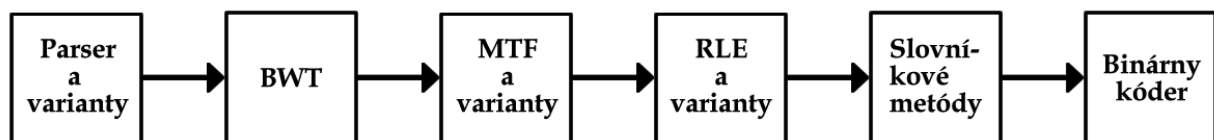
- off – pri komprimovaní je vynechaná slovníková metóda
- LZC – pri komprimovaní je použitý algoritmus LZC
- LZSS – pri komprimovaní je použitý algoritmus LZSS

Pre konkrétne nastavenie pri spúšťaní programu XBW nad korpusom je v tabuľke zobrazený aj kompresný pomer pre daný korpus. Tento kompresný pomer sa počíta z celkovej pôvodnej veľkosti korpusu a jeho veľkosti po komprimácii. Je udávaný v jednotkách bpB, čo je skratka pre anglické bit per Byte. Táto jednotka nám udáva priemerný počet bitov potrebných na skomprimovanie jedného bytu. Ak je kompresný pomer 2 bpB znamená to, že potrebujeme priemerne 2 bity na zakomprimovanie jedného bytu, teda 8 bitov. Skomprimovaný korpus potom nadobúda 2/8 = 0,25, teda 25% zo svojej pôvodnej veľkosti. Kompresný pomer CP je vypočítaný na základe nasledujúceho vzťahu

$$CP = \frac{SV \times 8}{PV}$$

kde SV je skomprimovaná veľkosť korpusu a PV je pôvodná veľkosť korpusu.

Takisto je pre konkrétne nastavenie pri spúšťaní programu XBW nad korpusom v tabuľke zaznamenaná veľkosť abecedy, s ktorou algoritmy po fáze Burrows-Wheelerovej transformácie pracujú. Vzhľadom na to, že nás zaujímajú algoritmy po fáze Burrows Wheelerovej transformácie, je pri meraní použité nasledujúce poradie jednotlivých fáz:



Obrázok 1: Poradie jednotlivých fáz pri meraní kompresných pomerov.

Uvedené kompresné pomery pri použití vypísaných algoritmov nad korpusmi sú vybrané tak, aby bol poväčšinou zachytený prvý výskyt každej varianty pre jednotlivé algoritmy, teda najlepší kompresný pomer. Poprípade sú vybrané tak, aby zachytili zaujímavé merania. V každej tabuľke je takisto zachytený najhorší kompresný pomer pre daný zafixovaný algoritmus. Kompletný zoznam všetkých meraní kompresných pomerov sa nachádza na priloženom CD.

2. Abeceda

Definujme si prostredie, pomocou ktorého budeme pracovať s algoritmami. Nech A je abeceda, na ktorej existuje usporiadanie, teda pre každé dva prvky x,y abecedy A platí:

1. $x < y \quad y < x$
2. $x < y \quad y < z \Rightarrow x < z$

Veľkosť tejto abecedy je $|A|$. Označme $X = x_0x_1x_2...x_{n-1}$ reťazec s dĺžkou n, kde $x_i \in A$.

Dané značenie a vlastnosti platia pre každú abecedu a reťazec spomenuté v tejto práci. Nech X_{metoda} je reťazec, s ktorým ma byť prevedená nejaká operácia, vstupuje do metódy *metoda*, každé x_i z daného reťazca patrí do abecedy A_x a Y_{metoda} je reťazec, ktorý dostaneme ako výstup metódy *metoda* na reťazci X_{metoda} a každé y_i z reťazca Y_{metoda} patrí do množiny B_{metoda}.

Ak v algoritme budú nasledovať za sebou metódy *metoda1* a *metoda2*, zrejme bude platiť $Y_{metoda1} = X_{metoda2}$ a $B_{metoda1}$ je podmnožinou $A_{metoda2}$ a zároveň $A_{metoda2}$ je podmnožinou $B_{metoda1}$, teda sú identické. Ako neskôr uvidíme, množiny A_{metoda} a B_{metoda} nemusia byť rovnaké. Takýto prípad môže nastať vtedy, ak si metóda *metoda* upraví abecedu. Väčšinou sa táto abeceda zväčší o nejaké špeciálne symboly, ktoré potrebuje metóda na svoj beh.

3. Algoritmy block-sorting kompresie

3.1. Burrows-Wheelerova transformácia

Myšlienka Burrows-Wheelerovej transformácie a jej hlavný prínos pre komprimáciu textov je vytváranie zhlukov rovnakých písmen, ktoré sa vyskytujú v rovnakých kontextoch. Toto zhlukovanie nie je iba jednoduché zoradenie vstupného reťazca podľa abecedy, je to špeciálna rotácia všetkých písmen reťazca tak, aby sa z nej dal dostať pri dekompresii pôvodný reťazec. Kompresia a dekompresia Burrows-Wheelerovej transformácie vyzerá v pseudokóde následovne:

```
Transformácia BWT(reťazec s)
   1. vytvor zoznam všetkých rotácií reťazca s, ku ktorému pridáme taký
      ukončovací symbol $, ktorý sa nenachádza v abecede
   2. vytvor tabuľku o veľkosti s x s a zadaj do nej všetky rotácie
   3. abecedne zoraď všetky riadky tabuľky
   4. ako výstup vráť posledný stĺpec tabuľky spolu s jeho poradovým číslom
      v tabuľke po zoradení

Detransformácia BWT(reťazec s, číslo p)
   1. vytvor prázdnu tabuľku veľkosti s x s
   2. opakuj s krát
      a. vlož reťazec s do prvého voľného stĺpca od prava
      b. abecedne zoraď riadky v tabuľke
   3. ako výstup vráť p-tý riadok z vyplnenej tabuľky bez ukončovacieho
      symbolu
```

Ukážeme si Burrows-Wheelerovu transformáciu na reťazci "BANANA":

Transformácia			
Vstup	**Všetky rotácie**	**Zoradené rotácie**	**Výstup**
BANANA	BANANA\$ \$BANANA A\$BANAN NA\$BANA ANA\$BAN NANA\$BA ANANA\$B	ANANA\$B ANA\$BAN A\$BANAN BANANA\$ NANA\$BA NA\$BANA \$BANANA	BNN\$AAA, 4

Tabuľka 3 : Burrows-Wheelerova transformácia na reťazci "BANANA"

Inverzná transformácia (detranformacia)			
Krok 2a	**Krok 2b**	**Krok 2a**	**Druhý krok 2b**

B N N $ A A A	A A A B N N $	BA NA NA $B AN AN A$	AN AN A$ BA NA NA $B
Tretí krok 2a	**Tretí krok 2b**	**Štvrtý krok 2a**	**Štvrtý krok 2b**
BAN NAN NA$ $BA ANA ANA A$B	ANA ANA A$B BAN NAN NA$ $BA	BANA NANA NA$B $BAN ANAN ANA$ A$BA	ANAN ANA$ A$BA BANA NANA NA$B $BAN
Krok 2a	**Piaty krok 2b**	**Šiesty krok 2a**	**Krok 2b**
BANAN NANA$ NA$BA $BANA ANANA ANA$B A$BAN	ANANA ANA$B A$BAN BANAN NANA$ NA$BA $BANA	BANANA NANA$B NA$BAN $BANAN ANANA$ ANA$BA A$BANA	ANANA$ ANA$BA A$BANA BANANA NANA$B NA$BAN $BANAN
Krok 2a	**Krok 2b**	**Výstup**	
BANANA$ NANA$BA NA$BANA $BANANA ANANA$B ANA$BAN A$BANAN	ANANA$B ANA$BAN A$BANAN BANANA$ NANA$BA NA$BANA $BANANA	**BANANA**	

Tabuľka 4 : Detransformácia pre reťazec transformovaný Burrows-Wheelerovou transformáciou v tabuľke 3.

Fakt, že text po aplikácii Burrows-Wheelerovej transformácie má tú vlastnosť, že zvykne zhlukovať rovnaké písmená pri viacnásobnom výskyte nejakého písmena, vyplýva z nasledujúcej analýzy.

V slove BANANA sa podreťazec “NA“ vyskytuje 2 krát a síce BANANA a BANANA. Rovnako podreťazec “AN“ sa v tomto slove vyskytuje 2 krát a síce BANANA a BANANA. Dokonca to platí aj pre podreťazce, ktoré sa vo vstupnom reťazci vzájomne prekrývajú, ako napríklad reťazec “ANA“ sa vyskytuje v slove BANANA dva krát a síce BANANA a BANANA.

Keďže v tabuľke máme všetky rotácie slova, tak sa tam vyskytujú práve dve rotácie pre podreťazec “NA“ také, že písmeno A z daného podreťazca je na začiatku rotácie, tj. je to prvé

písmeno v reťazci pre danú rotáciu a písmeno N je na konci rotácie, tj. je posledné písmeno pre danú rotáciu. Preto po abecednom zoradení všetkých rotácii je veľmi pravdepodobné, že tieto dve rotácie budú hneď pod sebou. Môže ešte nastať prípad, že v pôvodnom reťazci existuje podreťazec alebo viac podreťazcov, ktoré sa dostanú medzi tieto dve rotácie.

Pre prípad, keď skúmame podreťazec "NA", zoberme tie rotácie, pre ktoré platí, že písmeno A je prvé písmeno rotácie a písmeno N je posledné písmeno rotácie. Takéto rotácie sú v našom prípade dve. Nech "AxN" je prvá spomenutá rotácia a "AyN" je druhá spomenutá rotácia a nech bez újmy na všeobecnosti platí, že $x < y$ v abecednom zoradení, teda x bude v abecednom zoradení pred y. Potom ak existuje rotácia "Az" taká, že pre ňu platí "xN" < z < "yN", tak je zrejmé, že sa rotácia "Az" po abecednom zoradení všetkých rotácií ocitne niekde medzi rotáciami "AxN" a "AyN". Samozrejme týchto rotácií môže byť viac. Z predchádzajúcej analýzy je zrejmé, že čím dlhší podreťazec skúmame, tým je väčšia pravdepodobnosť toho, že sa pri zoradení rotácií medzi prislúchajúce reťazce nedostane ďalší, tj. na výstupe sa budú zhlukovať rovnaké písmená.

3.2. Alternatíva pre Burrows-Wheelerovu transformáciu

V roku 1997 publikoval Schindler [11] modifikáciu Burrows-Wheelerovej transformácie. Pri Burrows-Wheelerovej transformácii sa v zrotované reťazce zoradia a na výstup sa pošle posledný stĺpec. V tom sú pozhlukované symboly, ktoré mali v reťazci X_{bwt} podobné kontexty. Zoberme si posledný symbol zrotovaného reťazca. Celý reťazec pred ním nazvime jeho kontextom. Potom BWT vlastne zoradí koncové symboly podľa ich kontextov.

Schindler si uvedomil, že na to aby sa dosiahlo zhlukovanie symbolov z rovnakého kontextu vo výstupnom reťazci Y_{bwt} je často zbytočné triediť podľa celých kontextov, stačí zobrať len určitý počiatočný úsek kontextu a triediť zrotované reťazce podľa tohto začiatku.

Triedenie počiatočných úsekov kontextov prináša nový problém v oblasti Burrows-Wheelerovej transformácie. Pre každú transformáciu, pre ktorú ma existovať inverzná transformácia teda detransformácia, musí platiť, že symbol, ktorý nasleduje za kontextom musí byť jednoznačne identifikovateľný. Pri počiatočných úsekov kontextov však túto jedinečnú rozlíšiteľnosť zaručiť nevieme. Schindler preto ukázal jednoduchú metódu ako

tento problém vyriešiť. Uvedieme si princípy tranformácie a detransformácie postupne na úsekoch dĺžky 0 až n.

3.2.1. Počiatočný úsek kontextu dĺžky 0

Pri rozlíšení symbolov podľa počiatočného úseku kontextu dĺžky 0 sa nachádzajú všetky symboly X_{bwt} v tom istom a síce prázdnom kontexte. Zotriedenie podľa kontextu by teda neprinieslo žiaden výsledok. V rámci daného kontextu môžu byť zotriedené ľubovoľne, preto neexistuje detransformácia. Schindler navrhol jednoduché riešenie. Ak sa dva symboly nachádzaju v rovnakom kontexte, ten čo sa nachádza v reťazci X_{bwt} skôr nachádza sa skôr aj v danom kontexte.

Pri počiatočnom úseku kontextu dĺžky 0 sú teda všetky symboly X_{bwt} zotriedené v jednom kontexte a v tomto kontexte sú zoradené podľa svojej pozície v reťazci X_{bwt}. Z toho vyplýva, že $X_{bwt} = Y_{bwt}$, teda transformovaný reťazec je rovnaký ako pôvodný reťazec. Detransformácia je tým pádom ľahká. Symbol na vstupe sa dá na výstup.

3.2.2. Počiatočný úsek kontextu dĺžky 1

Transformácia

Transformácie pre počiatočný úsek kontextu dĺžky 1 prebieha následovne. Pri prvom prechode reťazcom X_{bwt} sa spočítajú frekvencie jednotlivých symbolov. Zrejme platí

$$X_{bwt} = \sum_{x_i \in X_{bwt}} \textit{frekvencia symbolu } x_i$$

V pamäti si alokujeme miesto o veľkosti $|X_{bwt}|$ symbolov a pre každý symbol x_i si vyhradíme miesto potrebné pre symboly, ktorým bude tento symbol x_i kontextom. Pre symbol x_i si teda vyhradíme miesto o veľkosti frekvencie symbolu x_i.

Pri druhom prechode reťazcom X_{bwt} je pri situácii $\ldots x_i x_j \ldots$ symbol x_i kontextom symbolu x_j. Vzhľadom na podmienku o jednoznačnej zotriediteľnosti podľa skoršieho výskytu v reťazci X_{bwt}, symbol x_j zaradíme na prvé voľné miesto v pamäti vyhradenej pre

kontext symbolu x_i. Po spracovaní celého reťazca X_{bwt} dostávame v pamäti výstupný reťazec Y_{bwt}.

Detransformácia

Pri detransformácii nastáva problém s tým, že nevieme kde sa v reťazci Y_{bwt} nachádzajú začiatky kontextov jednotlivých symbolov. Pri počiatočných úsekoch kontextu dĺžky 0 bol problém vyriešený hneď na začiatku, keďže sme mali iba jeden kontext, ten začínal na začiatku reťazca Y_{bwt}.

Stačí si však uvedomiť jednu základnú vec. Frekvencia jednotlivých symbolov je v reťazci X_{bwt} rovnaká ako v transformovanom reťazci X_{bwt}. Vyplýva to z toho, že symbol na vstupe je ihneď uložený v pamäti pre výstupný reťazec Y_{bwt}. Stačí preto spočítať frekvencie jednotlivých symbolov v reťazci Y_{bwt}. Tým dostaneme začiatky kontextov symbolov. Pre začiatok kontextu k_i symbolu x_i platí:

$$k_i = \sum_{j \leq i} frekvencia\ symbolu\ x_j$$

Pred detransformáciou musíme vedieť počiatočný kontext, s ktorým máme začať. Pri detransformácii následne zoberieme prvý nepoužitý symbol x_j kontextu x_i, v ktorom sa nachádzame a dáme ho na výstup. Následne sa presunieme do kontextu symbolu x_i, zoberieme prvý voľný symbol v tomto kontexte a dáme ho na výstup.

3.2.3. Počiatočný úsek kontextu dĺžky n

Transformácia

Pri jednom priechode reťazcom X_{bwt} zistíme frekvencie jednotlivých počiatočných úsekov kontextu dĺžky n. V pamäti si pripravíme potrebné miesto, celková veľkosť alokovanej pamäti je $|X_{bwt}|$. Pri druhom priechode reťazcom X_{bwt} dáme symbol x_i na vstupe na prvé prázdne miesto v pamäti vyhradenej pre počiatočný úsek kontextu pre daný symbol x_i.

Detransformácia

Pri detransformácii potrebujeme vedieť, kde sa začínajú počiatočné úseky kontextov dĺžky n. Stačí si uvedomiť, kontext k_1 dĺžky n je formovaný kontextom k_2 dĺžky n-1 a symbolom x_i, ktorého kontext o dĺžke n-1 je práve kontext k_2. To znamená, že ak poznáme v reťazci Y_{bwt} začiatky kontextov o dĺžke n-1 a kontext k_1 o dĺžke n dostaneme poskladaním kontextu k_2 a symbolu x_i, ktorý ku kontextu prislúcha, teda platí $k_1 = k_2 + x_i$, stačí nám zistiť frekvencie jednotlivých symbolov x_j, ktoré prislúchajú ku kontextu k_2. Musíme samozrejme rátať frekvencie v rámci kontextu k_2. Keďže vieme zistiť začiatky kontextov pre dĺžku kontextov 1, vieme zistiť začiatky kontextov pre ľubovoľnú dĺžku n. Pre detransformáciu nám následne stačí vedieť začiatočný kontext, s ktorým máme začať.

4. Algoritmy po Burrows-Wheelerovej transformácii

Máme teda predstavu o tom, aký reťazec Y_{BWT} dostaneme ako výstup Burrows-Wheelerovej transformácie. Keďže je to transformácia, ktorá nemení veľkosť abecedy, uskutoční len rotáciu vstupného reťazca, zjavne platí $|A_{bwt}| = |B_{bwt}|$. Výhoda, ktorú nám pri ďalšej komprimácii poskytne daná transformácia spočíva v tom, že vytvára zhluky rovnakých písmen a zachováva lokálnu štruktúru dokumentu, lebo symboly z podobných alebo rovnakých kontextov vytvárajú podreťazce kódu, ktoré sa opakujú a tým pádom vytvárajú možnosť lepšej kompresie.

Po Burrows-Wheelerovej transformácii sa ďalej nasadzuje algoritmus Move-to-front, ktorý vstupný reťazec $Y_{bwt} = X_{mtf}$ pretransformuje na číselnú radu, kde každý symbol z X_{mtf} je z množiny $\{0,1,..., | B_{bwt} | - 1\}$ a táto množina predstavuje abecedu $B_{mtf,}$,na ktorej je definovaná aj relácia porovnania.

4.1. Algoritmus Move-to-front

Move-to-front algoritmus si na začiatku vytvorí zoznam Z všetkých prvkov abecedy B_{bwt}, kde sú symboly zaradené podľa veľkosti, teda najmenší je prvý a najväčší je posledný. Na danom zozname existuje funkcia F(x), ktorá pre prvok x z daného zoznamu vráti jeho index v zozname, teda jeho poradové číslo. Prvý prvok v zozname má poradové číslo 0. Pri prítomnosti znaku y_i na vstupe sa nájde v zozname, na vstup sa vypíše poradie znaku v zozname, teda $F(y_i)$ a znak sa premiestni na prvé miesto zoznamu.

Toto je pôvodná verzia algoritmu Move-to-front, existujú ďalšie, ktoré budú opísané v časti o Move-to-front. Teda Move-to-front algoritmus transformuje jednotlivé symboly a priradzuje im čísla podľa ich lokálnej pravdepodobnosti výskytu. Častejšie sa vyskytujúcim symbolom priraďuje menšie hodnoty a tie menej často sa vyskytujúce odsúva v zozname viac dozadu, majú teda väčšie hodnoty. V istom zmysle teda napodobňuje štatistické komprimačné metódy. Podreťazce dlhšie ako $t > 2$, kde t závisí od verzie zvoleného Move-to-front algoritmu, ktoré sú tvorené identickým symbolom transformuje na beh núl. Takisto priradzuje symbolom a opakujúcim sa podreťazcom s väčšou lokálnou pravdepodobnosťou výskytu menšie hodnoty, ktoré sú následne dobre zakódovateľné binárnym kóderom.

Move-to-front algoritmus v pseudokóde vyzerá nasledovne:

```
Move-to-front(vstup, velkosť abecedy)

vytvor vzostupne zoradený zoznam L prvkov od 0 po veľkosť_abecedy - 1
while(je na vstupe symbol x){
        zisti poradie i symbolu x v zozname L
        daj na výstup číslo i - 1
        presuň symbol x v zozname L na začiatok
}
```

Jednou z vlastností algoritmu Move-to-front je vytváranie dlhých nulových behov, teda mnohonásobný výskyt čísla 0 za sebou. Pri mnohonásobnom výskyte čísla 0 sa logicky zvyšuje pravdepodobnosť výskytu v globálnom merítku. Tu ale nastáva problém, pretože vo fragmentoch reťazca, kde sa 0 až tak často nevyskytuje je jej lokálna pravdepodobnosť oveľa menšia ako celková globálna. Tento problém rieši algoritmus Run-length encoding (RLE).

4.2. Alternatívy pre algoritmus Move-to-front

Move-to-front algoritmus je jedna z možností globálnej transformácie reťazca, okrem neho sa v tejto fáze používajú napríklad tieto algoritmy:

- **Time Stamp** – pracuje na podobnom princípe ako Move-to-front, bol publikovaný Albersom [12]. Tiež si udržuje zoznam Z, má funkciu F(x), ktorá pre prvok x vracia jeho index v zozname, líši sa len v premiestňovaní prvku v rámci zoznamu. Keď je na vstupe symbol x_i, na výstup sa vypíše hodnota funkcie F(x). Prvok x_i sa presunie pred prvok, ktorý bol najčastejšie žiadaný od poslednej požiadavky na prvok x_i. Ak sa zatiaľ nevyskytla žiadna požiadavka na prvok x_i, tak ostane na svojom mieste a nie je presunutý.

- **Inversion Frequencies** – bol vyvinutý Arnavutom a Magliverasom [20]. Reťazec Y_{bwt} nad abecedou B_{bwt} pretransformuje na číselnú radu, pričom každé číslo z tejto rady patrí do množiny $\{0, ..., |B_{bwt}| - 1\}$, čo je zároveň aj abeceda A_{if}. Reťazec Y_{if} dostaneme nasledujúcim postupom. Pre každý symbol x_i abecedy B_{bwt} prehľadáme reťazec Y_{bwt}. Pri prvom výskyte symbolu x_i vypíšeme na výstup jeho pozíciu v reťazci Y_{bwt}. Pri každom ďalšom výskyte symbolu x_i vypíšeme na výstup počet väčších symbolov od posledného výskytu symbolu x_i podľa relácie zoradenia abecedy. Samotný reťazec Y_{if} nám na dekompresiu na pôvodný

reťazec Y_{bwt} stačiť nebude, potrebujeme mať ešte uložený počet výskytov v dokumente pre každý symbol.

- **Distance coding** – bol vyvinutý E. Binderom [14]. Reťazec Y_{bwt} nad abecedou B_{bwt} pretransformuje na číselnú radu, pričom každé číslo z tejto rady patrí do množiny {0, ..., n - 1}, kde n je dĺžka reťazca Y_{bwt}. Táto množina je zároveň aj abeceda A_{dc}. Reťazec Y_{if} dostaneme nasledujúcim postupom. Ak máme na vstupe symbol x_i po prvý krát, vypíšeme jeho pozíciu v reťazci Y_{bwt}. Ak máme na vstupe symbol x_i taký, že sa už v reťazci Y_{bwt} vyskytol, nájdeme jeho ďalší výskyt v reťazci Y_{bwt} a je na pozícii p_2 a momentálna pozícia symbolu x_i je na pozícii p_1, na výstup vypíšeme hodnotu p_2 - p_1. Ak sa už symbol x_i do konca reťazca Y_{bwt} nevyskytuje, vypíšeme 0. Na to, aby sme mohli korektne dekomprimovať reťazec Y_{dc} na Y_{bwt}, musíme ešte pre každý symbol z abecedy B_{bwt} poznať prvú pozíciu výskytu v reťazci Y_{bwt}.

- **Weighted Frequency Count** – vyvinutý S. Deorowiczom [15]. V podstate zovšeobecňuje algoritmy na podobnej báze ako Move-to-front algoritmus. Pre každý symbol z reťazca Y_{bwt} sa vypočíta hodnota $W_i(a_j)$, ktorá je daná predpisom

$$W_i(a_j) = \sum_{\substack{1 \le i \\ a_j = x_p}} w(i - p)$$

Reťazec Y_{wfc} dostaneme nasledujúcim spôsobom. Ak máme na vstupe symbol x_i, ktorý je na j-tej pozícii v reťazci Y_{bwt}, vypočítame pre každý symbol x_m z abecedy B_{bwt} jeho $W_j(x_m)$. Následne je zoznam Z usporiadaný podľa zostupných hodnôt $W_j(x_m)$, takže symbol x_n s najväčšou hodnotou $W_j(x_n)$ bude na prvom mieste a symbol x_k s najmenšou hodnotou $W_j(x_k)$ bude na poslednom mieste. Ak pre niektoré m rôzne od n platí, že $W_j(x_m) = W_j(x_n)$, tak sa berie $W_{j-1}(x_m)$ a $W_{j-1}(x_n)$, $W_{j-2}(x_m)$ a $W_{j-2}(x_n)$ a tak ďalej. Nakoniec $W_0(x_m) = -m$, takže určite existuje jednoznačné zoradenie pre danú abecedu a reťazec Y_{bwt}. Následne sa na výstup vypíše poradie symbolu x_i v takto zoradenom zozname Z.

Poradie symbolov v zozname závisí takisto aj od funkcie w(t). Ak definujeme funkciu w(t) = 1 pre t = 1 a w(t) = 0 pre t > 1, tak algoritmus Weighted Frequency Count presne kopíruje algoritmus Move-to-front. Slabina tohto prístupu spočíva v efektívnom zvolení funkcie w(t) pre reťazec Y_{bwt}.

4.3. Algoritmus Move-to-front pre veľké abecedy

Pri použití veľkej abecedy v algoritme Move-to-front, ktorá sa podľa skúseností z projektu XBW pohybuje až okolo hodnoty 80 000 symbolov, je hlavným problémom, ktorý nám takáto veľkosť prináša vyhľadanie pozície symbolu x_i v dátovej štruktúre, ktorou reprezentujeme Move-to-front zoznam.

Všeobecne najrozšírenejšou štruktúrou na udržiavanie Move-to-front zoznamu je spojový zoznam. Táto implementácia je vhodná pre menšie abecedy, kde sa pri operácii nájdenia pozície hľadaného symbolu x_i, ktorá sa často využíva pri transformácii reťazca $Y_{bwt} = X_{mtf}$ na reťazec Y_{mtf} , nemusí urobiť veľa porovnaní, pretože nájdenie správneho prvku sa prevádza lineárnym prehľadávaním spojového zoznamu. Takisto pri procese inverznej transformácie reťazca Y_{mtf} na reťazec Y_{bwt} sa často používa operácia nájdenia symbolu na i-tej pozícii, ktorá je pri spojovanom zozname lineárna.

Prirodzená vlastnosť Move-to-front zoznamu, výskyt najfrekventovanejších prvkov na začiatku zoznamu prirodzene znižuje nutný počet porovnaní symbolov pre malú abecedu, kde sa prvky vyskytujú s väčšou frekvenciou než pri veľkej abecede. Takisto vyhľadanie pozície dlho nepoužitého symbolu nepredstavuje rapídny nárast porovnaní, keďže sa väčšinou používa abeceda o veľkosti 256 symbolov.

Pri veľkej abecede si vyhľadanie dlho nepoužitého prvku môže vyžiadať až okolo 80 000 porovnaní a keďže frekvencia jednotlivých symbolov nie je až taká veľká ako pri menšej abecede, je väčšia pravdepodobnosť použitia dlhšie nepoužitého prvku. Preto zlepšenie vyplývajúce z prirodzenej vlastnosti Move-to-front zoznamu nedosahuje takú mieru, aby prirodzene znížila počet porovnaní pre veľké abecedy v takej miere, aby bolo použitie spojového zoznamu možné. Naopak výhodou pri spojovom zozname je jeho ľahká údržba pre potreby algoritmu Move-to-front, kde potrebujeme nájdený prvok presunúť na vrchol zoznamu.

Je preto potrebné navrhnúť spôsob, akým by sa prehľadávanie aj celého zoznamu podstatne urýchlilo a nepredstavovalo by aj pre veľké abecedy rapídne zhoršenie časovej

zložitosti. Ako ukázali Grinberg a kol. [16] vhodné vlastnosti potrebné pre udržiavanie Move-to-front zoznamu, zachytenie jeho prirodzenej vlastnosti, teda výskyt najfrekventovanejších prvkov pri vrchole zoznamu, teda aj ich rýchlejšie vyhľadanie oproti priemernej dobe na vyhľadanie prvku a rýchle vyhľadávanie vo všeobecnosti, majú dátové štruktúry nazvané Splay tree.

4.3.1. Splay tree

Splay tree je adaptabilná dátová štruktúra určená na vyhľadávanie. Splay tree má štruktúru binárneho stromu. Každý uzol okrem listov má minimálne jedného a maximálne dvoch synov. Má takisto základnú vlastnosť binárnych vyhľadávacích stromov, jeho uzly sú ohodnotené. Pre každý uzol v tejto stromovej štruktúre platí, že všetky uzly v ľavom podstrome majú menšiu hodnotu a všetky uzly v pravom podstrome majú hodnotu väčšiu.

Zaujímavosťou tejto dátovej štruktúry je algoritmus splay(x), ktorá modifikuje Splay tree tak, aby boli zachované jeho vlastnosti o usporiadaní prvkov v tejto dátovej štruktúre a zároveň zistí, či sa prvok x nachádza v strome a presunie ho do koreňa. Ak sa prvok x v strome nenachádza, do koreňa sa premiestni buď najväčší menší prvok alebo najmenší väčší prvok. Algoritmus začína od uzlu, ktorý má byť premiestnený a postupnými rotáciami spôsobí presunutie uzlu do koreňa pri zachovaní vyhľadávacej a porovnávacej schopnosti štruktúry Splay tree.

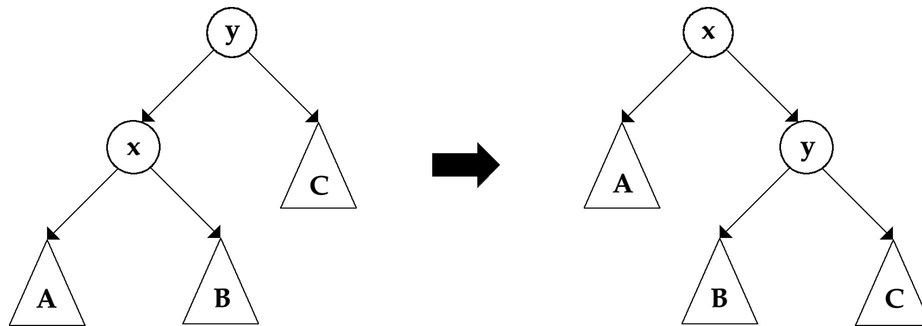
Algoritmus v pseudokóde vyzerá následovne:

splay(x):

```
while(uzol x nie je vrchol splay tree){
    if(otec uzla x je vrchol)
        rotácia(uzol x, otec uzla x)
    else
        if(uzol x, otec uzla x su obaja pravými alebo ľavými synami svojich
otcov){
            rotácia(otec uzla x, starý otec uzla x)
            rotácia(uzol x, otec uzla x)
            }
        else
            dvojitá_rotácia(uzol x, otec uzla x, starý otec uzla x)
}
```

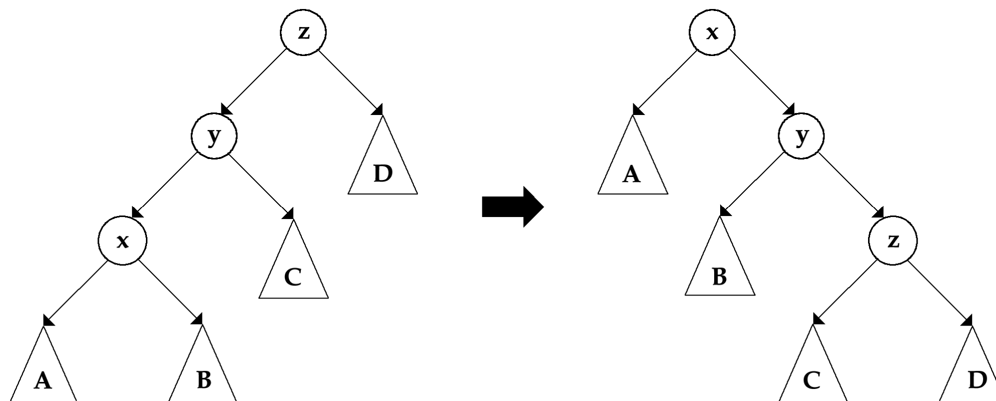
Pri algoritme splay(x) sa používajú operácie: rotácia(x,y) a dvojitá_rotácia(x,y,z), ktoré vykonávajú rotáciu uzlov a ich celých podstromov tak, aby zachovali usporiadanie v rámci dátovej štruktúry. Vyzerajú nasledovne:

rotácia(x,y):



Obrázok 2 : Popis operácie rotácia(x, y) pre datovú štruktúru Splay tree

dvojitá_rotácia(x,y,z):



Obrázok 3 : Popis operácie dvojitá_rotácia(x, y, z) pre datovú štruktúru Splay tree

Okrem toho zachovávajú tieto operácie v dátovej štruktúre Splay tree vlastnosť Move-to-front zoznamu – posledne použité prvky, ktoré boli presunuté na vrchol Splay tree, sa nachádzajú blízko vrcholu. Aj preto je to štruktúra vhodná na reprezentáciu Move-to-front zoznamu.

Algoritmus splay(x) má zároveň tendenciu vyvažovať stromovú štruktúru a tým pádom udržiavať čas potrebný na vyhľadanie uzla.

4.3.2. Splay tree implementácia pre Move-to-front

Dátová štruktúra Splay tree nám síce zabezpečuje potrebné vlastnosti pre Move-to-front zoznam a síce rýchle vyhľadávanie a presúvanie na koreň, ale chýba nám ešte vlastnosť, ktorá by zabezpečila, aby sme vedeli v rámci dátovej štruktúry Splay tree zabezpečiť usporiadanie prvkov v Move-to-front zozname. Musíme byť schopní povedať, ktorý prvok je v dátovej štruktúre Splay tree menší a ktorý väčší voči danému prvku, vzhľadom na usporiadanie v Move-to-front zozname.

Túto vlastnosť nám zabezpečí systém takzvaných časových razítok. Pre každý symbol bude existovať unikátne časové razítko, pričom na ľubovoľnej množine unikátnych časových razítok existuje usporiadanie, teda pre každé časové razítka *x*,*y*, ktoré nie sú rovnaké, platí:

1. $x < y \quad y < x$
2. $x < y \quad y < z \Rightarrow x < z$

Keď je symbol presunutý do koreňa, je mu pridelené časové razítko, ktoré je najmladšie spomedzi všetkých časových razítok v aktuálnom Splay tree. Ako časové razítko nám slúži štruktúra Splay tree, kde platí, že v ľavom podstrome sú uzly s menšou hodnotou a v pravom podstrome sú uzly s väčšou hodnotou. V našom prípade bude tá hodnota časové razítko. Zabezpečenie, aby po algoritme splay(x) ostalo označenie časovými razítkami aktuálne ukážeme neskôr.

Používanie dátovej štruktúry Splay tree sa začína jej vytvorením. V projekte XBW sú implementované dve funkcie na vytvorenie dátovej štruktúry Splay tree.

Prvá funkcia *native_init* zoberie prvý symbol v abecede, ktorý má byť v Move-to-front zozname na prvom mieste a umiestni ho do koreňa. Následne vyberie ďalší prvok abecedy, ktorý ma byť v Move-to-front zozname na druhom mieste a v Splay strome ho umiestni ako syna koreňu. Takto sa prejde cez celú abecedu a každý nový symbol je reprezentovaný uzlom, ktorý je umiestnený ako ľavý syn uzla, ktorý reprezentuje predchádzajúci symbol.

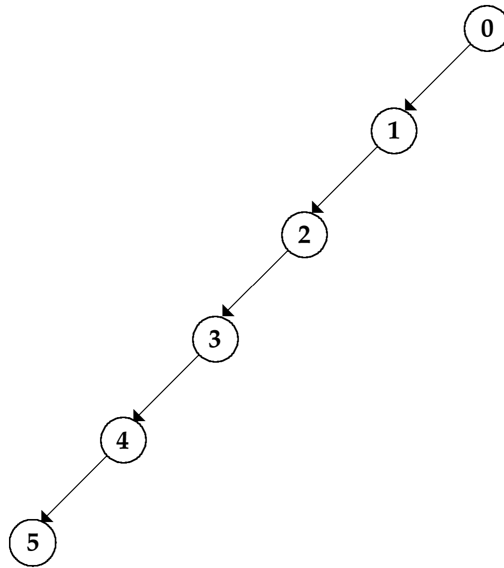
V pseudokóde vyzerá následovne:

```
native_init(abeceda):

zober prvý prvok abecedy, vytvor koreň a nastav mu nasledujúce hodnoty:
   •  jeho rodič a pravý syn je NULL
if(existuje druhý prvok)
```

```
      s_symbol nastav na druhý symbol abecedy
else
      predchádzajúci_symbol nastav na prvý symbol abecedy
while(s_symbol nie je posledný prvok abecedy){
      vytvor uzol a nastav mu nasledujúce hodnoty:
         o  jeho rodič je predchádzajúci_symbol
         o  jeho pravý syn je NULL
      predchádzajúci_symbol nastav na daný symbol
      s_symbol nastav na nasledujúci symbol abecedy
}
Vytvor uzol a nastav mu nasledujúce hodnoty:
   •  jeho rodič je predchádzajúci_symbol
   •  jeho pravý a ľavý syn sú NULL
```

Takže Splay tree by po inicializácii pre abecedu A = {0,1,2,4,5} vyzeral následovne:



Obrázok 4 : Jednoduchá inicializácia datovej štruktúry Splay tree

Splay tree vytvorená v takomto stave ale nie je ideálna, pretože máme v podstate lineárny zoznam a vôbec nevyužívame výhody, ktoré nám prináša stromová štruktúra. Po istej dobe sa vplyvom algoritmu splay(x) Splay tree dostane do viac vyváženého tvaru.

Túto počiatočnú nevýhodu sa snaží napraviť druhá viac sofistikovaná inicializácia, ktorá používa stratégiu „rozdeľuj a panuj“.

V pseudokóde vyzerá následovne:

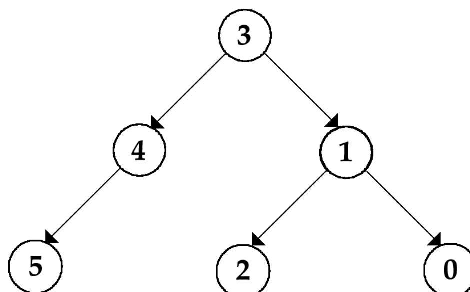
divide and conquer init(začiatok, koniec, rodič):

```
if(začiatok je rôzny od koniec){
      nájdi hodnotu symbolu v strede medzi symbolmi začiatok a koniec
      a označ ju stred
      if(symbol stred nie je rovnaký ako symbol začiatok ani symbol koniec){
        vytvor uzol Splay tree, nazvi ho s_stred a nastav mu nasledujúce
      hodnoty:
```

```
            • jeho rodič je parameter rodič
            • ľavý syn je podstrom, ktorý dostaneme zavolaním funkcie
              divide_and_conquer_init(začiatok, stred, s_stred)
            • pravý syn je podstrom, ktorý dostaneme zavolaním funkcie
              divide_and_conquer_init(stred + 1, koniec, s_stred)
         vráť odkaz na vytvorený uzol s_stred
      }
      else{
        if(symbol stred je rovnaký ako symbol začiatok){
            vytvor uzol Splay tree, nazvi ho s_stred a nastav mu nasledujúce
            hodnoty:
               • jeho rodič je parameter rodič
               • jeho ľavý syn je NULL
               • jeho pravý syn je podstrom, ktorý dostaneme zavolaním
                 funkcie divide_and_conquer_init(koniec, koniec, s_stred)
            vráť odkaz na vytvorený uzol s_stred
        }
        if(symbol stred je rovnaký ako symbol koniec){
            vytvor uzol Splay tree, nazvi ho s_stred a nastav mu nasledujúce
            hodnoty:
               • jeho rodič je parameter rodič
               • jeho pravý syn je NULL
               • jeho ľavý syn je podstrom, ktorý dostaneme zavolaním
                 funkcie divide_and_conquer_init(začiatok, začiatok,
                 s_stred)
            vráť odkaz na vytvorený uzol s_stred
        }
      }

}
else {
        vytvor uzol Splay tree, nazvi ho s_stred a nastav mu nasledujúce
hodnoty:
            • jeho rodič je parameter rodič
            • ľavý aj pravý syn je NULL
            • jeho hodnota je index symbolu začiatok v abecede
        vráť odkaz na vytvorený uzol s_stred
}
```

Takže Splay tree by po inicializácii pre abecedu A = {0,1,2,4,5} vyzeral nasledovne:



Obrázok 5 : Inicializácia dátovej štruktúry Splay tree pomocou stratégie rozďeluj a panuj

Táto druhá implementácia vytvorenia štruktúry Splay tree má výhodu v tom, že už na začiatku je vyvážená.

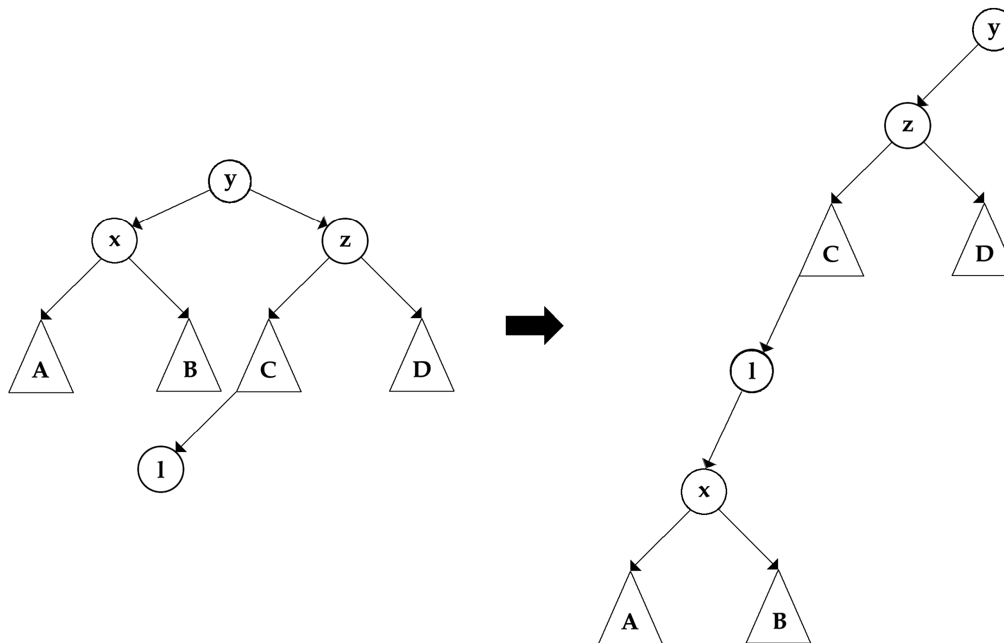
Keďže používame metódu časových razítok, je nutné po presune uzla do koreňa previesť modifikáciu dátovej štruktúry Splay tree tak, aby ostala zachovaná vlastnosť, kvôli ktorej sme metódu použili a sice, aby v ľavom podstrome uzla, v našom prípade, aby v ľavom podstrome koreňa boli uzly s menšími časovými razítkami a v pravom podstrome uzla, teda v pravom podstrome koreňa boli iba uzly s väčšími časovými razítkami. Keď uzol presunieme do koreňa, je mu pridelené najmladšie časové razítko, čo znamená, že časové razítko každého uzla v Splay tree je menšie ako nové časové razítko pridelené koreňu. Z vlastnosti časových razítok vyplýva, že ľavý podstrom koreňa bude obsahovať všetky uzly okrem koreňa a v pravom podstrome koreňa sa nebude nachádzať ani jeden prvok, lebo ak by sa nachádzal, musel by mať mladšie časové razítko než má koreň, ten má ale najmladšie časové razítko.

Musíme teda nájsť spôsob ako preorganizovať Splay tree, aby ostal koreň koreňom a aby bolo zachované usporiadanie podľa metódy časových razítok, čo rieši operácia *preorganizuj(x)*, kde x je koreň:

preorganizuj(x):

```
if(má pravý podstrom koreňa x aspoň jeden uzol)
        nájdi najľavejší list v pravom podstrome koreňa x a označ ho l
else
        l nastav na NULL
        if(l nie je NULL){
        nastav ľavý podstrom koreňa ako ľavý podstrom uzlu l a rodiča ľavého
podstromu koreňa nastav ako uzol l
        nastav pravý podstrom koreňa ako ľavý podstrom koreňa a pravý
podstrom nastav na NULL
}
```

Operáciu *preorganizuj(x)* popisuje nasledujúci obrázok:



Obrázok 6 : Preorganizovanie dátovej štruktúry Splay tree pre algoritmus MTF tak, aby koreň mal najmladšie časové razítko

Ak nemá pravý podstrom koreňa x žiaden uzol, nespraví operácia *preorganizuj(x)* nič, pretože Splay tree je už v požadovanom tvare.

Ak má pravý podstrom koreňa x nejaký uzol, ukážeme si, že operácia *preorganizuj(x)* naozaj zachovala usporiadanie podľa časových razítok a jediná zmena je tá, že koreň má najmladšie časové razítko. Značenie jednotlivých uzlov je rovnaké ako na obrázku. Nech l je uzol s najstarším časovým razítkom v pravom podstrome koreňa. Keďže sa nachádza v pravom podstrome koreňa y, pre časové razítka týchto dvoch uzloch platí, že y < l, teda uzol l má mladšie časové razítko ako koreň a zároveň pre každý uzol, ktorý je obsiahnutý v ľavom podstrome koreňa platí, že má časové razítko staršie ako koreň, teda aj ako časové razítko pre uzol l. V rámci pravého a aj ľavého podstromu koreňa y platí usporiadanie podľa časových razítok. Musíme ukázať, že pri presunutí ľavého podstromu koreňa y, od teraz o ňom budeme hovoriť ako o podstrome s koreňom x, a pri presune takto vzniknutého pravého podstromu koreňa y do ľavého podstromu koreňa y bude zachované usporiadanie podľa časových razítok. V rámci podstromu s koreňom x sme nehýbali žiadnymi uzlami, takže tam usporiadanie podľa časových razítok platí. Ako sme ukázali, uzol l ma mladšie časové razítko ako ľubovoľný uzol v podstrome, takže tým, že podstrom s koreňom x bude jeho ľavým podstromom ostane zachované usporiadanie podľa časových razítok v podstrome s koreňom l. Keďže l je uzol s najstarším časovým razítkom v bývalom pravom podstrome koreňa y, tak zrejme platí, že každý uzol z podstromov C a D spolu s uzlom z majú mladšie časové razítko, takže

usporiadanie podľa časových razítok platí v celom podstrome koreňa z. Keďže všetky uzly okrem koreňa z sa nachádzajú v ľavom podstrome koreňa z a keďže koreň z má najmladšie časové razítko, je zrejmé, že usporiadanie podľa časových razítok platí v celom Splay tree.

4.3.3. Výsledky kompresie nad korpusmi pre algoritmus Move-to-front

Calgary korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
off		MTF	RLE-2	off	2,27	142
text	Znak	MTF	RLE-2	off	2,29	142
off		MTF	RLE-1	off	2,31	142
text	Znak	MTF	RLE-1	off	2,33	142
text	Slabika	MTF	RLE-2	off	2,44	3363
off		MTF	RLE-1	LZC	2,48	142
text	Znak	MTF	RLE-3	LZC	2,49	142
text	Slovo	MTF	off	LZSS	2,66	4140
text	Slabika	MTF	RLE-3	off	20,87	3363
text	Slovo	MTF	RLE-3	off	25,22	4140

Tabuľka 5 : Kompresné pomery a veľkosti abecied pre algoritmus Move-to-front nad Calgary korpusom.

XML_{cz} korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Slabika	MTF	RLE-2	off	0,71	41542
text	Slovo	MTF	RLE-2	off	0,71	80132
text	Slabika	MTF	RLE-1	off	0,72	41542
xml	Slabika	MTF	RLE-2	off	0,72	36478
xml	Slovo	MTF	RLE-1	off	0,73	78714
off		MTF	RLE-2	off	0,76	241
text	Znak	MTF	off	LZC	0,76	394
text	Znak	MTF	RLE-3	LZC	0,76	394
text	Znak	MTF	RLE-2	LZC	0,76	394
off		MTF	RLE-3	LZC	0,77	241
off		MTF	RLE-3	off	0,77	241
text	Slovo	MTF	off	LZSS	0,96	80132
xml	Slovo	MTF	RLE-2	LZSS	0,99	78714
xml	Slovo	MTF	RLE-3	off	4,61	78714
text	Slovo	MTF	RLE-3	off	4,66	80132

Tabuľka 6 : Kompresné pomery a veľkosti abecied pre algoritmus Move-to-front nad XML_{cz} korpusom.

Z výsledkov tabuliek vieme povedať, že sa pri algoritme Move-to-front oplatí použiť metódu Run-length a to buď variantu RLE-1 alebo RLE-2. Pri nepoužití metódy Run-length a pri použití varianty RLE-3 vychádzajú horšie výsledky oproti RLE-1 a RLE-2.

Pri malých súborov vychádza najlepší kompresný pomer pre parsovanie na znaky. Za týmto výberom sa z pohľadu kompresného pomeru nachádza parsovanie na slabiky, ktorého najlepšie výsledky sú o 0,1 až 0,2 bpB horšie ako pri parsovaní na znaky. O rovnaký rozdiel v kompresnom pomere vychádza horšie parsovanie na slová. Z toto jednoznačne vyplýva, že pri malých súboroch je pre najlepší kompresný pomer pri variante Move-to-front nutné zvoliť parsovanie na znaky.

Pri veľkých súboroch je to naopak najvýhodnejšie parsovanie na slabiky alebo celé slová. Dá sa povedať, že je jedno ktorú variantu parsovania použijeme, pretože nám budú vychádzať podobné výsledky. Pri použití parsovania na znaky sa však kompresný pomer dramaticky nezmení, najlepšie výsledky vychádzajú iba o 0,05 bpB horšie, je to však 7% zhoršenie oproti najlepšiemu kompresnému pomeru pre korpus XML_{cz}.

Aj keď najlepšie výsledky pri použití algoritmu LZC sú iba o 0,05 až 0,15 bpB horšie ako najlepšie dosahované výsledky na každom korpuse, vzhľadom na to, že pri inom parsovaní pre dosiahnutie lepšieho výsledku nie je potrebné použiť tento algoritmus. Z tabuľky 6 dostávame zaujímavý výsledok a síce pre veľký súbor na malej abecede je v podstate jedno, či do kompresie zapojíme alebo nezapojíme algoritmus LZC. Z tabuliek takisto vyplýva, že algoritmus LZC dosahuje lepšie výsledky na menších abecedách.

Algoritmus LZSS sa nám neoplatí s algoritmom Move-to-front používať, pretože pri jeho použití dostávame o 0,2 až 0,4 bpB horšie kompresné pomery ako pre najlepšie kompresné pomery pre daný korpus.

Zaujímavosťou je, že pre pri použití slovníkových metód nad korpusom XML_{cz} je jedno, akú variantu algoritmu Run-length encoding použijeme, dokonca či ho vôbec použijeme. Vyplýva z toho, že kompresiu nám pri veľkých súboroch zabezpečujú ostatné zložky.

Ako si môžeme všimnúť, najlepšie kompresné pomery pre malé súbory vychádzajú pri použití malých abecied, pretože na nich dosahujú viaceré symboly väčšie frekvencie ako pri veľkých abecedách. Pri veľkých abecedách sa výhoda mnoho symbolov nestačí prejaviť. Rozdiel pre použitie malých a veľkých abecied na malých súboroch pre najlepšie kompresné pomery je o 0,15 bpB.

Naopak pri kompresii veľkých súborov je výhodnejšie použiť veľkú abecedu, pre ktorú vychádzajú lepšie kompresné pomery. Rozdiel avšak nie je až taký výrazný ako pri použití pre malé súbory. Pri použití veľkej abecedy získavame na najlepšom kompresnom pomere iba 0,05 bpB nad Calgary korpusom.

Najhoršie kompresné pomery sú spôsobené príliš veľkou abecedou, v ktorej je veľmi veľa symbolov s podobnou malou frekvenciou v rámci súboru. To spôsobí ďalším spracovaním pomocou binárneho kóderu pre malé súbory dokonca nárast súboru a to až trojnásobne. Pri veľkých súboroch sa vyprofilujú frekvencie pre jednotlivé symboly, preto nedochádza k nárastu zakomprimovaného súboru.

4.4. Algoritmus Move-to-front-1

Modifikácia známeho algoritmu Move-to-front sa zameriava na skutočnosť, že beh jedného symbolu je často prerušený iným symbolom alebo menšou skupinkou symbolov, v ktorej sa žiaden symbol neopakuje. Máme napríklad podreťazec $x_i x_i x_i x_i x_j x_i x_i x_i$ a nech pri začiatku spracovania tohto podreťazca sa nachádza symbol x_i na druhom a symbol x_i na m-tom mieste v zozname symbolov pre Move-to-front algoritmus a platí $2 < m$. Potom bude daný reťazec zakódovaný ako sekvencia 1, 0, 0, 0, m - 1, 1, 0, 0, dostávame tak sekvenciu, ktorá je síce pomerne dobre zakódovateľná, ale ak by sme mali napríklad sekvenciu 1, 0, 0, 0, m - 1, 0, 0, 0, tak tá by sa dala zakódovať lepšie. Práve takéto situácie sa pokúša lepšie riešiť algoritmus Move-to-front-1.

Modifikácia oproti Move-to-front algoritmu spočíva v tom, že symbol presunieme na začiatok zoznamu iba vtedy, ak sa momentálne nachádza na druhom mieste v zozname. Ak sa nachádza za druhým prvkom v zozname, tak ho presunieme na druhú pozíciu v zozname.

Zameriava sa na vyššie popísaný efekt, ak bol pri pôvodnom Move-to-front algoritme beh jedného symbolu x prerušený jedným alebo skupinou rôznych symbolov a opäť nasleduje beh daného symbolu x, nemusí sa najprv zapísať na výstup pozícia symbolu x v zozname, ale plynule sa pokračuje v zapísaní núl na výstup.

V pseudokóde vyzerá algoritmus Move-to-front-1 nasledovne:

move-to-front-1(vstup, velkosť abecedy):

```
vytvor vzostupne zoradený zoznam L prvkov od 0 po veľkosť_abecedy – 1
while(je na vstupe symbol x){
        zisti poradie i symbolu x v zozname L
        daj na výstup číslo i – 1
        if(i == 2)
                presuň symbol x v zozname L na začiatok
        else
                presuň symbol x v zozname L na druhú pozíciu
}
```

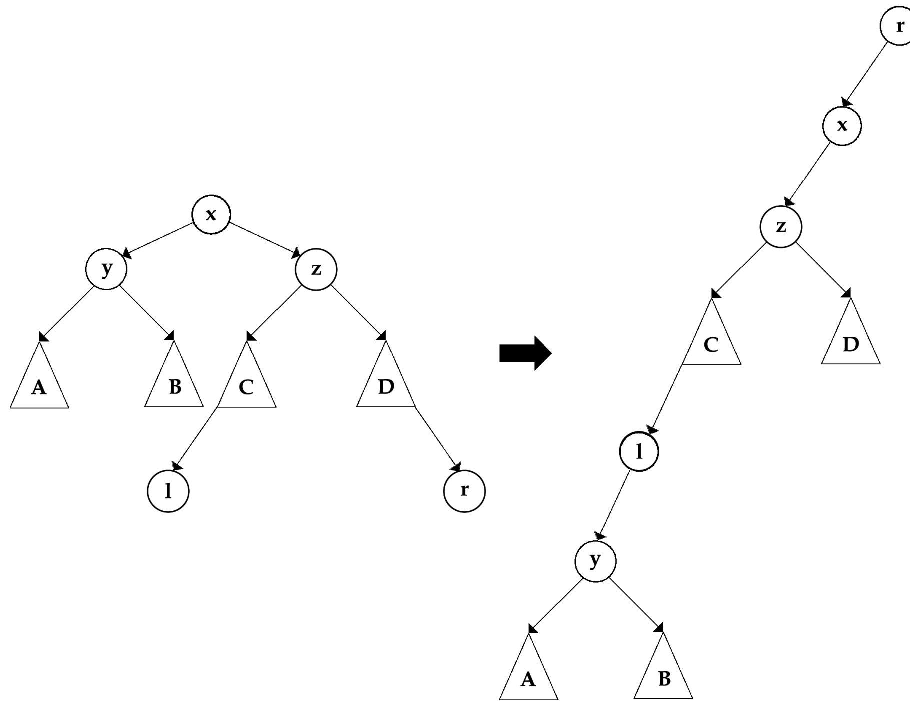
4.4.1. Splay tree implementácia pre algoritmus Move-to-front-1

Rovnako ako pri pôvodnom Move-to-front algoritme s využitím Splay tree, aj pri Move-to-front-1 algoritme využívame Splay tree a metódu časových razítok. Základná idea je rovnaká ako pri Move-to-front algoritme, zmení sa len operácia *preorganizuj(x)*, kde x je koreň. V pseudokóde vyzerá nasledovne:

preorganizuj(x):

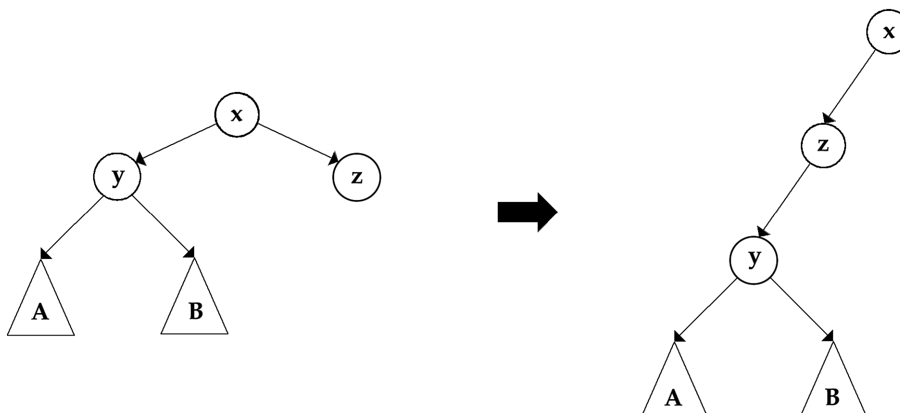
```
if(má pravý podstrom koreňa x aspoň jeden uzol)
        nájdi najľavejší list v pravom podstrome koreňa x a označ ho l
        nájdi najpravejší list v pravom podstrome koreňa x a označ ho r
else
        l a r nastav na NULL
if(l a r nie je NULL a l a r reprezentujú rôzny uzol){
        nastav ľavý podstrom koreňa ako ľavý podstrom uzlu l a rodiča ľavého
podstromu koreňa nastav ako uzol l
        nastav pravý podstrom koreňa ako ľavý podstrom koreňa a pravý
podstrom nastav na NULL
        nastav pravý podstrom rodiča uzlu r ako NULL
        nastav rodiča uzlu r ako NULL
        nastav ľavý podstrom uzla r ako koreň x
        nastav rodiča koreňa x ako uzol r
}
else
        if(l a r reprezentujú ten istý uzol){
                nastav ľavý podstrom r ako ľavý podstrom koreňa x
                nastav pravý podstrom koreňa x ako NULL
                nastav pravý ľavý podstrom uzlu r ako koreň x
                nastav rodiča koreňa x ako uzol r
        }
```

Operáciu `preorganizuj(x)` pre aspoň dva uzly v pravom podstrome koreňa x popisuje nasledujúci obrázok:



Obrázok 7 : Preorganizovanie dátovej štruktúry Splay tree pre algoritmus MTF-1 v prípade ak má koreň v pravom podstrome aspoň dva uzly tak, aby koreň mal druhé najmladšie časové razítko

Operáciu `preorganizuj(x)` pre jeden uzol v pravom podstrome koreňa x popisuje nasledujúci obrázok:



Obrázok 8 : Preorganizovanie dátovej štruktúry Splay tree pre algoritmus MTF-1 v prípade, ak má koreň v pravom podstrome práve jeden uzol tak, aby koreň mal najmladšie časové razítko

Ukážeme si, že operácia `preorganizuj(x)` zachováva usporiadanie podľa časových razítok.

Ak koreň x nemá pravý podstrom, potom je na prvom mieste v zozname pre algoritmus Move-to-front-1. Splay tree zachováva usporiadanie podľa časových razítok a nepotrebuje modifikáciu a ani metóda `preorganizuj(x)` nespraví žiadne zmeny, takže zachová usporiadanie podľa časových razítok.

Ak pravý podstrom koreňa x obsahuje jeden uzol, je tento uzol označený aj ako najpravejší a aj ako najľavejší, nazvime si ho z. Zároveň je uzol z na vrchole v zozname Move-to-front-1 a koreň x je na druhom mieste v tomto zozname. Podľa algoritmu Move-to-front-1 máme pri výskyte symbolu na vstupe, ktorý sa nachádza na druhom mieste v zozname dať tento symbol na prvé miesto v zozname, tým pádom mu dať najnovšiu časovú značku. Je zrejmé, že všetky uzly v ľavom podstrome koreňa x majú časové razítko staršie ako koreň x pred presunutím na vrchol v zozname Move-to-front-1. Je takisto jasné, že uzol z má novšie časové razítko ako koreň x, tým pádom má aj novšie časové razítko ako ľubovoľný uzol obsiahnutý v ľavom podstrome koreňa x. Preto ak dostaneme usporiadanie ako na obrázku, tak podstrom s koreňom z zachováva usporiadanie podľa časových razítok. Koreň x máme podľa algoritmu Move-to-front-1 umiestniť na vrchol zoznamu a dostane najnovšie časové razítko, čo ale operácia `preorganizuj(x)` zachováva, pretože koreň x je koreňom novovytvoreného Splay tree a všetky ostatné uzly sa nachádzajú v jeho ľavom podstrome.

Ak pravý podstrom koreňa x obsahuje viac ako jeden uzol, tak ako najpravejší a najľavejší uzol sú označené dva rôzne uzly r a l. Uzol l má najstaršie časové razítko a uzol r ma najnovšie časové razítko v rámci pravého podstromu koreňa x. Z toho vyplýva, že koreň x sa nachádza v zozname pre Move-to-front-1 na vyššom ako druhom mieste. Podľa algoritmu Move-to-front-1 má byť presunutý na druhé miesto v zozname, teda ako keby dostal najnovšie časové razítko a následne dostal uzol, ktorý bol predtým na vrchole zoznamu Move-to-front-1, takisto najnovšie časové razítko. Uzol x by sa teda ocitol na druhom mieste v zozname. Ak si v terajšej situácii odmyslíme uzol r a nazveme tento Splay strom n, dostávame situáciu podobnú ako v pôvodnom Move-to-front a nový strom n je pre Move-to-front-1 operáciou `preorganizuj(x)` modifikovaný rovnako ako pre operáciu `preorganizuj(x)` v pôvodnom algoritme. Ako sme si ukázali v pôvodnom `preorganizuj(x)`, táto operácia zachováva usporiadanie podľa časových razítok. Stačí teda ukázať, že odobratím uzla r a pridaním ako na obrázku ostane zachované usporiadanie podľa časových razítok. Uzol r má pred modifikáciu Splay tree najnovšie časové razítko v rámci pravého podstromu koreňa x, ale to znamená, že má zároveň aj najmladšie časové razítko v rámci celého Splay tree a je na vrchole zoznamu

Move-to-front-1. Teda podľa algoritmu Move-to-front-1 má naozaj ostať na vrchole zoznamu, pretože koreň x je v tomto zozname na vyššom ako druhom mieste. Teda uzol po modifikácii operáciou `preorganizuj(x)` vo fáze, keď koreň x dostane najnovšie časové razítko, teda všetky uzly okrem koreňa x budú obsiahnuté v jeho ľavom podstrome, zoberie uzol r, dá mu najnovšie časové razítko, aby Splay tree splňoval podmienku pre algoritmus Move-to-front-1 a nastaví ako jeho ľavého syna uzol x a pravý syn uzla r neobsahuje žiaden uzol, teda je naozaj na vrchole zoznamu Move-to-front-1 a uzol x sa nachádza na druhom mieste v tomto zozname a je zachované usporiadanie podľa časových razítok.

4.4.2. Výsledky kompresie nad korpusmi pre algoritmus Move-to-front-1

Calgary korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
off		MTF-1	RLE-2	off	2,26	256
text	Znak	MTF-1	RLE-2	off	2,28	142
off		MTF-1	RLE-1	off	2,30	256
text	Znak	MTF-1	RLE-1	off	2,32	142
text	Slabika	MTF-1	RLE-2	off	2,43	3363
off		MTF-1	RLE-1	LZC	2,47	256
text	Slovo	MTF-1	RLE-2	off	2,48	4141
text	Slovo	MTF-1	off	LZSS	2,65	4141
text	Slabika	MTF-1	RLE-3	off	20,86	3363
text	Slovo	MTF-1	RLE-3	off	25,20	4141

Tabuľka 7 : Kompresné pomery a veľkosti abecied pre algoritmus Move-to-front-1 nad Calgary korpusom.

XML$_{cz}$ korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Slabika	MTF-2	RLE-2	off	0,73	35435
text	Slovo	MTF-2	RLE-2	off	0,73	80132
text	Slabika	MTF-2	RLE-1	off	0,74	35435
text	Slovo	MTF-2	RLE-1	off	0,74	80132
xml	Slabika	MTF-2	RLE-2	off	0,75	34068
xml	Slovo	MTF-2	RLE-2	off	0,75	78714
off		MTF-2	RLE-3	LZC	0,77	241
text	Znak	MTF-2	off	LZC	0,77	394
xml	Znak	MTF-2	RLE-3	LZC	0,78	722
text	Slovo	MTF-2	RLE-1	LZSS	0,97	80132
text	Slovo	MTF-2	off	off	0,99	80132

xml	Slovo	MTF-2	RLE-1	LZSS	1	78714
xml	Slovo	MTF-2	off	off	1,02	78714
xml	Slovo	MTF-2	RLE-3	off	4,63	78714
text	Slovo	MTF-2	RLE-3	off	4,69	80132

Tabuľka 8 : Kompresné pomery a veľkosti abecied pre algoritmus Move-to-front-1 nad XML_{cz} korpusom.

Pre variantu Move-to-front-1 algoritmu Move-to-front platia rovnaké závery a dôsledky, aké sme vyvodili z tabuliek v kapitole 4.3.3. , avšak sa trocha zmenil kompresný pomer. Na oboch korpusoch dosahuje varianta Move-to-front-1 lepší pomer. Táto zmena ale nie je výrazna, pohybuje sa od 0,01 do 0,02 pre najlepšie kompresné pomery pre zvolený korpus. Za príčinou daného zlepšenia stojí heuristika pri posúvaní symbolu v Move-to-front zozname.

4.5. Move-to-front-2

Algoritmus Move-to-front-2 je modifikáciou algoritmu Move-to-front-1. Rovnako ako algoritmus Move-to-front-1 presúva symbol v zozname na pozícii väčšej ako 2 na druhú pozíciu, obmena spočíva v tom, že symbol je z druhého miesta v zozname presunutý na vrchol zoznamu iba v prípade, ak posledné číslo na výstupe nie je nula, čo vlastne znamená, že posledne spracovaný symbol v reťazci je momentálne spracovaný symbol, lebo inak by nemohol byť momentálne na druhej pozícii v zozname.

V pseudokóde vyzerá algoritmus Move-to-front-2 následovne:

```
move-to-front-2(vstup, velkosť abecedy):

vytvor vzostupne zoradený zoznam L prvkov od 0 po veľkosť_abecedy - 1
while(je na vstupe symbol x){
        zisti poradie i symbolu x v zozname L
        daj na výstup číslo i - 1
        if(i == 2 a posledné číslo na výstupe nie je 0)
                presuň symbol x v zozname L na začiatok
        else
                presuň symbol x v zozname L na druhú pozíciu
}
```

Operácia `preorganizuj(x)` vyzerá rovnako ako pri algoritme Move-to-front-1, jediná zmena, ktorú nám podmienka naviac v algoritme Move-to-front-2 urobila je to, kedy presunieme symbol z druhého miesta v zozname na prvé miesto, čo ale nezmení operáciu

`preorganizuj(x)`. Preto je aj pri algoritme Move-to-front-2 zachované usporiadanie podľa časových razítok.

4.5.1. Výsledky kompresie nad korpusmi pre algoritmus Move-to-front-2

Calgary korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
off		MTF-2	RLE-2	off	2,25	142
text	Znak	MTF-2	RLE-2	off	2,27	142
off		MTF-2	RLE-1	off	2,29	142
text	Slabika	MTF-2	RLE-2	off	2,42	3220
off		MTF-2	RLE-2	LZC	2,47	142
text	Slovo	MTF-2	RLE-2	off	2,48	4141
text	Slovo	MTF-2	off	LZSS	2,65	4141
text	Slabika	MTF-2	RLE-3	LZSS	2,67	3220
text	Slabika	MTF-2	RLE-3	LZC	2,75	3220
text	Znak	MTF-2	RLE-1	off	2,31	142
text	Slabika	MTF-2	RLE-3	off	20,85	3220
text	Slovo	MTF-2	RLE-3	off	25,20	4141

Tabuľka 9 : Kompresné pomery a veľkosti abecied pre algoritmus Move-to-front-2 nad Calgary korpusom.

XML$_{cz}$ korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Slabika	MTF-2	RLE-2	off	0,74	41542
text	Slovo	MTF-2	RLE-2	off	0,74	80132
text	Slabika	MTF-2	RLE-1	off	0,75	41542
text	Slovo	MTF-2	RLE-1	off	0,75	80132
xml	Slabika	MTF-2	RLE-2	off	0,75	34069
xml	Slovo	MTF-2	RLE-1	off	0,76	78714
off		MTF-2	off	LZC	0,77	241
text	Znak	MTF-2	RLE-3	LZC	0,77	394
xml	Znak	MTF-2	RLE-1	LZC	0,78	722
off		MTF-2	off	off	0,8	241
text	Slovo	MTF-2	off	LZSS	0,97	80132
xml	Slabika	MTF-2	RLE-2	LZSS	1,05	34069
text	Slovo	MTF-2	RLE-3	off	4,64	80132
xml	Slovo	MTF-2	RLE-3	off	4,69	78714

Tabuľka 10 : Kompresné pomery a veľkosti abecied pre algoritmus Move-to-front-2 nad XML_{cz} korpusom.

Pre variantu Move-to-front-1 algoritmu Move-to-front platia rovnaké závery a dôsledky, aké sme vyvodili z tabuliek v kapitole 4.3.3. , avšak sa zmenil kompresný pomer.

Na malých súboroch, teda na Calgary korpuse dosahuje varianta Move-to-front-2 lepší pomer, na XML_{cz} korpuse s veľkými súbormi horší. Táto zmena sa pohybuje od 0,01 bpB do 0,04 bpB pri zlepšení oproti algoritmu Move-to-front na Calgary korpuse pre najlepšie kompresné pomery. Na XML_{cz} korpuse je zhoršenie pre najlepšie kompresné pomery približne o 0,01 bpB oproti najlepším kompresným pomerom pre algoritmus Move-to-front.

4.6. Dopad algoritmu Move-to-front a jeho variánt na komprimáciu

Nasledujúce tabuľky ukazujú kompresný pomer pri nezapojení fázy Move-to-front alebo nejakej jeho varianty:

Calgary korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Slovo	off	RLE-2	off	2,66	4141
off		off	RLE-2	off	2,68	256
text	Znak	off	RLE-1	off	2,7	142
text	Slovo	off	RLE-1	off	2,7	4141
text	Slovo	off	off	LZSS	2,71	4141
text	Slabika	off	RLE-2	off	2,78	3220
xml	Slovo	off	off	LZC	2,82	4141
text	Slovo	off	RLE-3	off	25,31	4141

Tabuľka 11 : Kompresné pomery a veľkosti abecied bez použitia algoritmu Move-to-front alebo jeho variánt nad Calgary korpusom.

XML_{cz} korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Slovo	off	RLE-2	off	0,86	80132
off		off	RLE-3	off	0,87	241
text	Znak	off	RLE-3	off	0,87	394
xml	Slovo	off	RLE-3	off	0,87	78714
off		off	RLE-2	off	0,89	241
text	Slabika	off	RLE-2	off	0,89	35436
xml	Znak	off	RLE-3	off	0,9	722
xml	Slabika	off	RLE-2	off	0,91	34069

off		off	off	LZC	0,94	241
text	Znak	off	RLE-1	LZC	0,94	394
text	Slovo	off	off	LZSS	1,07	80132
off		off	off	LZSS	1,45	241
xml	Slabika	off	off	off	11,16	34069
text	Znak	off	off	off	11,48	394

Tabuľka 12 : Kompresné pomery a veľkosti abecied bez použitia algoritmu Move-to-front alebo jeho variánt nad XMLcz korpusom.

Pri neprítomnosti fázy algoritmu Move-to-front alebo nejakej jeho varianty dostávame horšie výsledky na oboch korpusoch. Pre malé súbory v Calgary korpuse je najlepší kompresný pomer o 0,51 bpB horší ako najlepší kompresný pomer zo všetkých variánt a síce pri algoritme Move-to-front-2. Je to podstatné zlepšenie, predstavuje zlepšenie o 19,17%.

Pre veľké súbory v korpuse XML_{cz} predstavuje nepoužitie algoritmu MTF alebo jeho varianty zhoršenie najlepšieho kompresného pomeru o 0,15 bpB oproti najlepšiemu kompresnému pomeru pri použití algoritmu MTF alebo jeho varianty. Najlepší kompresný pomer vychádza pre veľké súbory v XML_{cz} korpuse pre algoritmus MTF a je to 0,71 bpB. Dochádza teda ku 17,4% zlepšeniu.

Ak by sme mali zoradiť vhodnosť algoritmu MTF a jeho variant pre malé abecedy, poradie by vyzeralo takto:

1. MTF-2
2. MTF-1
3. MTF
4. off

Pre veľké súbory by vyzerala takto:

1. MTF
2. MTF-1
3. MTF-2
4. off

Z toho pohľadu je na všeobecné použitie, kedy nevieme veľkosť komprimovaných súborov najvhodnejšia varianta MTF-1, ktorá dosahuje dobré výsledky aj pre malé a aj pre veľké súbory. Rozdiely medzi jednotlivými variantami ale predstavujú rozdiel väčšinou okolo

0,01 bpB, čo nepredstavuje veľký rozdiel. Môžeme preto dané varianty vyhlásiť za približne rovnaké.

Veľmi zaujímavé zistenie, že kompresný pomer pri použití slovníkových metód nebol nijako zmenený algoritmom RLE alebo jeho variantou. Dokonca bolo pre kompresný pomer jedno, či sa algoritmus RLE do komprimácie zapojil alebo nie. Preto pri použití slovníkovej metódy po Burrows-Wheelerovej transformácii je zbytočné zapájať fázu RLE, jediný výsledok by bolo zväčšenie času potrebného na komprimovanie.

Naopak, pri neprítomnosti slovníkovej metódy počas komprimácie na variante algoritmu RLE alebo jeho nepoužití záleží. Z výsledkov tabuliek vieme povedať, že sa pri algoritme MTF a jeho variantách oplatí použiť metódu Run-length a to buď variantu RLE-1 alebo RLE-2. Pri nepoužití metódy Run-length a pri použití varianty RLE-3 vychádzajú horšie výsledky oproti RLE-1 a RLE-2.

Použitie slovníkovej metódy pre nastavenie algoritmov a parsera, v ktorom sa vyskytuje algoritmus MTF alebo jeho varianta zhoršuje kompresný pomer oproti prípadu keď slovníkovú metódu nepoužijeme. Výnimkou je použitie algoritmu LZC pri malých abecedách na veľkých súboroch, keď nastáva zlepšenie v kompresnom pomere približne o 3%. Tento výsledok je zachytený v tabuľke 10 pre prípad, keď nepoužívame parser. Naopak, ak používame slovníkovú metódu, pridanie algoritmu MTF alebo jeho varianty nám kompresný pomer zlepší stále.

Z tabuliek nevyplýva žiadna súvislosť medzi veľkosťou abecedy a použitím alebo vynechaním algoritmu MTF.

5. Run-length encoding

Po úprave reťazca pomocou algoritmu Move-to-front dostávame postupnosť čísel, ktorá je obvykle viac zoskupená okolo nuly ako pôvodný reťazec interpretovaný abecedou začínajúcou nulou. Takisto má tendenciu vytvárať dlhé behy, v ktorých vyniká hlavne nula, ktorá zvyčajne býva najfrekventovanejším symbolom v rámci nového reťazca. Move-to-front algoritmus a jeho variácie majú ale za následok vlastnosť, ktorá sa ešte viac prejaví pri veľkých abecedách.

Princípom algoritmu Run-length encoding je skracovanie dlhých behov tak, aby bolo možné dekomprimáciou dostať pôvodný reťazec, ale aby v komprimovanom reťazci nedochádzalo k neúmernému zvyšovaniu globálnej pravdepodobnosti nejakého symbolu kvôli jeho mnohonásobnému výskytu v určitom fragmente. Algoritmus Run-length encoding má priestorovú zložitosť O(1) a časovú O(n).

Následne je reťazec pripravený pre komprimáciu nejakým slovníkovým algoritmom, štatistickou metódou a následne binárnym kóderom, ktorý ho zapíše do binárneho súboru. Toto nie je jediná správna modifikácia reťazca, jednotlivé fázy sa môžu experimentálne prehadzovať, existujú varianty, ktoré algoritmus RLE dávajú jednak pred Move-to-front algoritmus a jednak za Move-to-front algoritmus.

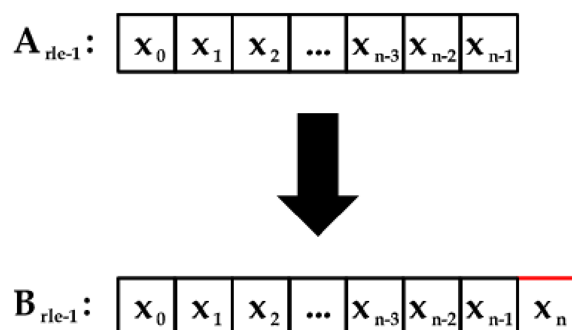
Definícia:
Reťazec r zložený z toho istého symbolu x_i s dĺžkou $d = |r|$ nazývame dlhým behom, ak $d > 2$.

V určitých prípadoch hovoríme aj o behu dĺžky dva, jedna alebo nula. Keď však nešpecifikujeme dĺžku dlhého behu, myslí sa dĺžka $d > 2$.

5.1. Run-length encoding 1

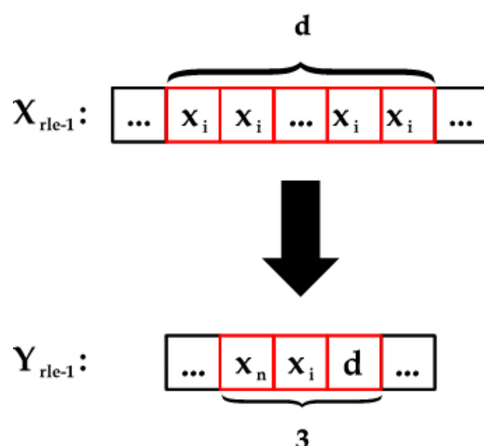
Algoritmus run-length encoding 1, ďalej budeme používať označenie RLE-1, je obľúbená a pomerne jednoduchá varianta algoritmov založených na myšlienke skracovania behov. Základný princíp tejto metódy spočíva v pridaní špeciálneho znaku do abecedy symbolov, ktoré spracúvame a tento špeciálny symbol bude signalizovať, že nasledujúci reťazec je skomprimovaný dlhý beh nejakého symbolu.

Algoritmus RLE-1 prebieha nasledovne. Zoberieme vstupný reťazec metódy RLE-1 X_{rle}, ten obsahuje symboly abecedy A_{rle-1}. Nech v reťazci X_{rle} existuje beh symbolu x_i dĺžky d, pre ktorú platí $d > 2$. Potom reťazec X_{rle-1} vyzerá $...x_j x_i ... x_i x_m...$ a dĺžka podreťazca $x_i...x_i$ je d. Ak n je veľkosť abecedy A_{rle-1}, potom abeceda $A_{rle-1} = \{ x_0, x_1, x_2, ..., x_{n-3}, x_{n-2}, x_{n-1} \}$. Do výslednej abecedy B_{rle-1} pridáme taký špeciálny symbol aký sa nevyskytuje v abecede A_{rle}, teda x_n a abeceda $B_{rle-1} = \{ x_0, x_1, x_2, ..., x_{n-3}, x_{n-2}, x_{n-1}, x_n \}$.Tým pádom bude $|B_{rle-1}| = |A_{rle-1}| + 1$.



Obrázok 9 : Modifikácia vstupnej abecedy A_{rle-1} na výstupnú abecedu B_{rle-1} algoritmom RLE-1

Skracovanie dlhých behov v pôvodnom reťazci algoritmom RLE-1 spočíva v tom, že podreťazec $x_i...x_i$, v ktorom sa symbol x_i vyskytuje d-krát, nahradíme reťazcom $x_n x_i d$. V tomto reťazci je x_n je špeciálny symbol, ktorý značí, že nasledujúci symbol, v našom prípade x_i, sa v pôvodnom reťazci opakuje d-krát.



Obrázok 10 : Zakódovanie dlhého behu na trojicu symbolov algoritmom RLE-1

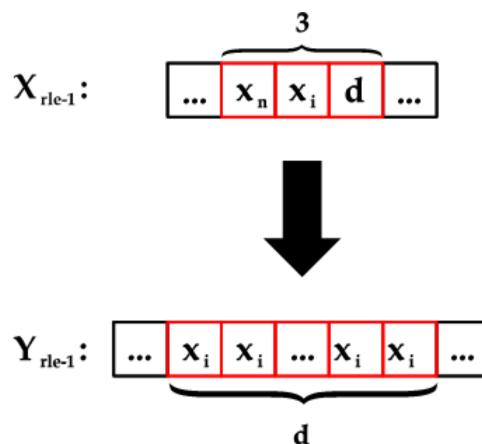
Reťazec o dĺžke d teda nahradíme reťazcom, v ktorom sa nachádzajú 3 symboly. Je zrejmé, že pre d = 2 sa nám reťazec nahradzovať neoplatí, pretože by sme pôvodný reťazec nezmenšili, ale naopak zväčšili. Pre d = 3 sa síce reťazec nezväčší, ostane rovnaký, pribudne

nám však réžia potrebná na zakódovanie a následne na odkódovanie reťazca namiesto jednoduchého prečítania symbolu a výpisu na výstup. Je preto zrejmé, že nahradzovať sa nám oplatí jedine reťazce, ktoré tvorí beh symbolu x_i aspoň o veľkosti 4. Preto je efektívne nastaviť podmienku pre dĺžku reťazca na skomprimovanie $d > 3$. Počítame s tým, že symboly abecedy A_{rle-1}, ktorej veľkosť je n, sú reprezentované ako čísla od 0 po n-1.

V pseudokóde vyzerá algoritmus RLE-1 pre zakódovanie nasledovne:

```
RLE-1_zakóduj(vstup, velkosť_abecedy):
dĺžka = 1
while(na vstupe je symbol){
        načítaj symbol a označ ho x
        while(existuje na vstupe ďalší symbol a je rovnaký ako symbol x)
                dĺžka = dĺžka + 1
        if(dĺžka > 3){
                vypíš na výstup čísla veľkosť_abecedy, symbol x, dĺžka
v tomto poradí
                dĺžka = 1
        }
        else
                while(dĺžka > 1){
                        vypíš na výstup symbol x
                        dĺžka = dĺžka - 1
                }
}
```

Pri dekódovaní vypisujeme na výstup symbol, ktorý dostaneme na vstupe, pokiaľ to nie je špeciálny symbol. Ak to je špeciálny symbol x_{y+1} vieme, že na vstupe máme reťazec $x_{y+1}x_id$, kde x_i je symbol, ktorého beh bol v pôvodnom reťazci zakódovaný a d je pôvodná dĺžka tohto behu. Takže reťazec $x_{y+1}x_id$ zameníme za beh symbolu x_i s dĺžkou d, teda na výstup vypíšeme d-krát symbol x_i.



Obrázok 11 : Odkódovanie trojice symbolov na dlhý beh algoritmom RLE-1

V presudokóde vyzerá algoritmus RLE1 pre odkódovanie nasledovne:

```
RLE-1_odkóduj(vstup, velkosť_abecedy):
while(na vstupe je symbol){
        načítaj symbol a označ ho y
        if(y == veľkosť_abecedy){
                načítaj ďalší symbol zo vstupu a označ ho x
                načítaj ďalší symbol zo vstupu a označ ho d
                for(d-krát)
                        vypíš na výstup symbol x
        }
        else
                vypíš na výstup symbol y
}
```

5.1.1. Výsledky kompresie nad korpusmi pre algoritmus RLE-1

Calgary korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
off		MTF-2	RLE-1	off	2,29	142
off		MTF-1	RLE-1	off	2,30	142
off		MTF	RLE-1	off	2,31	142
text	Znak	MTF-2	RLE-1	off	2,31	142
text	Znak	MTF-1	RLE-1	off	2,32	142
text	Slabika	MTF-2	RLE-1	off	2,43	3364
off		MTF	RLE-1	LZC	2,47	142
text	Znak	MTF-1	RLE-1	LZC	2,48	142
text	Slovo	MTF-1	RLE-1	off	2,48	4141
text	Slovo	MTF	RLE-1	LZSS	2,65	4141
text	Slovo	off	RLE-1	LZSS	2,7	4141
text	Slabika	off	RLE-1	LZC	3,03	3364

Tabuľka 13 : Kompresné pomery a veľkosti abecied pre algoritmus RLE-1 nad Calgary korpusom.

XML_{cz} korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Slabika	MTF	RLE-1	off	0,72	41542
text	Slovo	MTF	RLE-1	off	0,72	80132
xml	Slabika	MTF	RLE-1	off	0,73	34068
xml	Slovo	MTF	RLE-1	off	0,73	78714
text	Slovo	MTF-1	RLE-1	off	0,74	80132
off		MTF-1	RLE-1	LZC	0,77	241
off		MTF-2	RLE-1	LZC	0,77	241
text	Znak	MTF-2	RLE-1	LZC	0,77	394
xml	Znak	MTF-1	RLE-1	LZC	0,78	721

text	Znak	off	RLE-1	LZC	0,94	394
text	Slovo	MTF	RLE-1	LZSS	0,96	80132
off		off	RLE-1	LZSS	1,45	241

Tabuľka 14 : Kompresné pomery a veľkosti abecied pre algoritmus RLE-1 nad XML_{cz} korpusom.

Ako sme konštatovali pri výsledkov z algoritmu Move-to-front, pri použití algoritmu RLE-1 sú vhodné pre malé súbory všetky varianty algoritmu Move-to-front.

Pre veľké súbory je jednoznačne najvhodnejšie použiť pôvodný algoritmus Move-to-front alebo jeho variantu Move-to-front-1. Najlepší kompresný pomer pre algoritmus RLE-1 pri použití varianty Move-to-front-2 stráca o 0,05 bpB oproti najlepšiemu kompresnému pomeru pri použití algoritmu Move-to-front, čo predstavuje 7% zhoršenie kopresného pomeru.

Pri malých súborov vychádza najlepší kompresný pomer pre parsovanie na znaky. Za týmto výberom sa z pohľadu kompresného pomeru nachádza parsovanie na slabiky, ktorého najlepšie výsledky sú o 0,4 bpB horšie ako pri parsovaní na znaky. Približne rovnaký rozdiel v kompresnom pomere vychádza horšie parsovanie na slová. Z toto jednoznačne vyplýva, že pri malých súboroch je pre najlepší kompresný pomer pri variante RLE-1 nutné zvoliť parsovanie na symboly.

Pri veľkých súboroch je to naopak, najvýhodnejšie parsovanie na slabiky alebo celé slová. Dá sa povedať, že je jedno ktorú variantu parsovania použijeme, pretože nám budú vychádzať podobné výsledky. Pri použití parsovania na znaky sa však kompresný pomer príliš nezmení, najlepšie výsledky vychádzajú o 0,05 bpB horšie. Táto malá hodnota ale vďaka malému najlepšiemu kompresnému pomeru pre RLE-1 pri parsovaní na slabiky predstavuje približne 7% zhoršenie kompresného pomeru, čo je pomerne veľa.

Najlepšie výsledky pri použití algoritmu LZC sú o 0,05 až 0,16 bpB horšie ako najlepšie dosahované výsledky na každom korpuse, vzhľadom na to, že pri inom parsovaní pre dosiahnutie lepšieho výsledku nie je potrebné použiť tento algoritmus. Z tabuliek takisto vyplýva, že algoritmus LZC dosahuje lepšie výsledky na menších abecedách. V Calgary korpuse v tabuľke 13 sa nachádzajú dve nastavenia líšiace sa iba prítomnosťou alebo neprítomnosťou algoritmu LZC. Bez použitia algoritmu LZC dosahuje program kompresný

pomer pre algoritmus RLE-1, ktorý je 2,31 bpB. Po pridaní algoritmu LZC sa kompresný pomer dramaticky zhorší na 2,47 bpB.

Algoritmus LZSS sa nám neoplatí s algoritmom RLE-1 používať, pretože pri jeho použití dostávame o 0,24 až 0,36 bpB horšie kompresné pomery ako pre najlepšie kompresné pomery pre daný korpus. Zaujímavosťou pri algoritme LZSS je to, že dosahuje takmer rovnaké kompresné pomery pre všetky varianty algoritmu Move-to-front.

Ako si môžeme všimnúť, najlepšie kompresné pomery pre malé súbory vychádzajú pri použití malých abecied, pretože na nich dosahujú viaceré symboly väčšie frekvencie ako pri veľkých abecedách. Pri veľkých abecedách sa výhoda mnoho symbolov nestačí prejaviť. Rozdiel pre použitie malých a veľkých abecied na malých súboroch pre najlepšie kompresné pomery je okolo 0,15 bpB. Činí to zhoršenie o 6,5%.

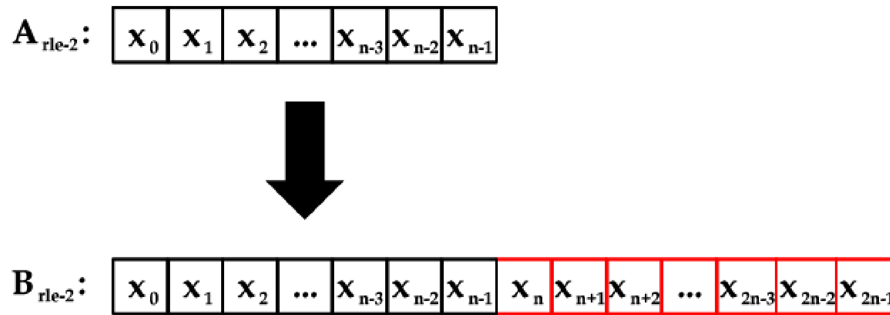
Naopak pri kompresii veľkých súborov je výhodnejšie použiť veľkú abecedu, pre ktorú vychádzajú lepšie kompresné pomery. Rozdiel avšak nie je až taký výrazný ako pri použití pre malé súbory. Pri použití veľkej abecedy získavame na najlepšom kompresnom pomere iba 0,05 bpB nad Calgary korpusom, čo je však percentuálne rovnako ako pri použití veľkých abecied nad malými súbormi.

Rozdiel medzi najlepšími a najhoršími kompresnými pomermi nie je taký výrazný ako sme sledovali pri algoritmoch Move-to-front.

5.2. Run-length encoding 2

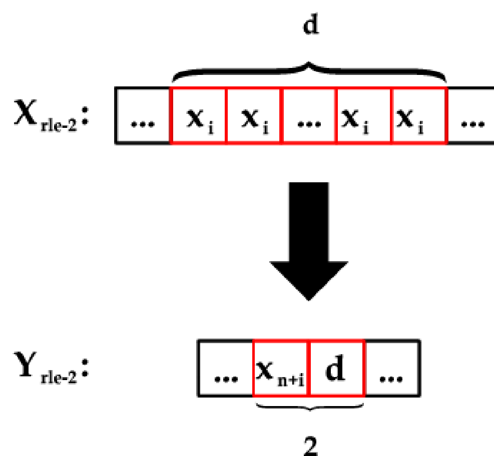
Run-length encoding 2, ďalej označované ako RLE-2 je ďalšia varianta na skracovanie dlhých behov symbolu x_i. Je založené na podobnej myšlienke ako RLE1, pridáva špeciálne symboly. Pri algoritme RLE1 sme beh symbolu x_i nahradzovali reťazcom $x_{y+1}x_id$. Teda d symbolov sme nahradili troma symbolmi. Algoritmus RLE-2 rozvíja myšlienku algoritmu RLE-1 a dáva dokopy špeciálny symbol signalizujúci, že nasledujúce symboly v sebe nesú informáciu o zakódovaní dlhého behu symbolu x_i a samotný symbol x_i. Na reprezentáciu behu dĺžky d symbolu x_i nám potom stačia dva symboly $x_{n+i}d$, kde n je veľkosť abecedy A_{rle-2} a x_{n+i} je nový špeciálny symbol.

Reťazec X_{rle-2}, ktorý je vstupom pre algoritmus RLE-2 má symboly z abecedy A_{rle-2}. Nech abeceda A_{rle-2} má veľkosť n, potom obsahuje symboly $A_{rle-2} = \{ x_0, x_1, x_2, ..., x_{n-3}, x_{n-2}, x_{n-1} \}$. Do novej abecedy B_{rle-2} pridáme pre každý symbol x_i špeciálny symbol x_{n+i}, ktorý signalizuje, že ďalší symbol na vstupe udáva dĺžku behu symbolu x_i v pôvodnom reťazci X_{rle-2}. Dostaneme tak abecedu $B_{rle-2} = \{ x_0, x_1, x_2, ..., x_{n-3}, x_{n-2}, x_{n-1}, x_n, x_{n+1}, x_{n+2}, ..., x_{2n-3}, x_{2n-2}, x_{2n-1} \}$ s dvojnásobným počtom symbolov oproti pôvodnej abecede A_{rle-2}.



Obrázok 12 : Modifikácia vstupnej abecedy A_{rle-2} na výstupnú abecedu B_{rle-2} algoritmom RLE-2

Zakódovanie reťazca X_{rle-2} algoritmom RLE-2 vyzerá podobne ako pri algoritme RLE-1. Ak máme v reťazci X_{rle-2} beh $x_i...x_i$ o dĺžke $d > 2$, tak ho vo výslednom reťazci X_{rle-2} nahradíme dvojicou špeciálny symbol a dĺžka behu $x_{n+i}d$ namiesto trojice symbolov $x_{y+1}x_id$ pri algoritme RLE-1. Zmenila sa takisto podmienka pre dĺžku behu oproti algoritmu RLE-1. Keďže behy sú zakódované dvojicou symbolov, oplatí sa nám zakódovávať behy už o dĺžke tri, pretože dosiahneme zmenšenie výsledného reťazca o d – 2 symbolov. Pre výsledný reťazec Y_{rle-2} teda po zakódovaní behu $x_i...x_i$ o dĺžke d bude platiť $| Y_{rle-2}| = | X_{rle-2}| - d + 2$.



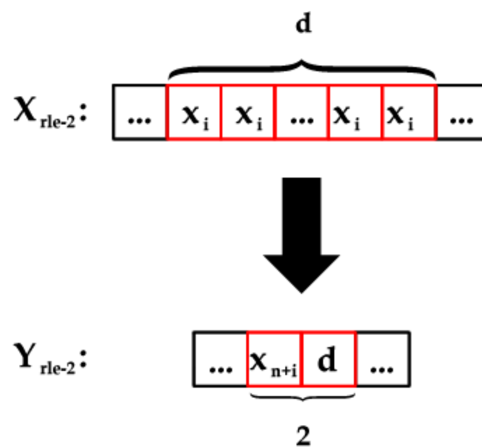
Obrázok 13 : Zakódovanie dlhého behu na dvojicu symbolov algoritmom RLE-2

Algoritmus RLE-2 na zakódovanie reťazca X_{rle-2} na výstupný reťazec Y_{rle-2} v pseudóde:

RLE-2 zakóduj(vstup, velkosť abecedy):

```
dĺžka = 1
while(na vstupe je symbol){
        načítaj symbol a označ ho x
        while(existuje na vstupe ďalší symbol a je rovnaký ako symbol x)
                dĺžka = dĺžka + 1
        if(dĺžka > 2){
                špeciálny_symbol = x + veľkosť_abecedy
                vypíš na výstup čísla špeciálny_symbol, dĺžka v tomto poradí
                dĺžka = 1
        }
        else
                while(dĺžka > 1){
                        vypíš na výstup symbol x
                        dĺžka = dĺžka - 1
                }
}
```

Pri dekódovaní vypisujeme na výstup symbol, ktorý nám príde na vstup, pokiaľ to nie je niektorý zo špeciálnych symbolov. Keď dostaneme na vstupe špeciálny symbol, prečítame zo vstupu ďalší symbol d a tým zistíme dĺžku behu v pôvodnom reťazci. Zo špeciálneho symbolu dekódujeme symbol, ktorý následne d-krát vypíšeme na výstup.



Obrázok 14 : Odkódovanie dvojice symbolov na dlhý beh algoritmom RLE-2

V presudokóde vyzerá algoritmus RLE-2 pre odkódovanie nasledovne:

RLE-2 odkóduj(vstup, velkosť abecedy):

```
while(na vstupe je symbol){
        načítaj symbol a označ ho y
        if(y >= veľkosť_abecedy){
                načítaj ďalší symbol zo vstupu a označ ho d
                pôvodný_symbol = y – veľkosť_abecedy
                for(d-krát)
                        vypíš na výstup symbol pôvodný_symbol
        }
        else
                vypíš na výstup symbol y
}
```

5.2.1. Výsledky kompresie nad korpusmi pre algoritmus RLE-2

Calgary korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
off		MTF-2	RLE-2		2,25	142
off		MTF-1	RLE-2	off	2,26	142
off		MTF	RLE-2	off	2,27	142
text	Znak	MTF-2	RLE-2	off	2,27	142
text	Znak	MTF-1	RLE-2	off	2,28	142
text	Znak	MTF	RLE-2	off	2,29	142
text	Slabika	MTF-2	RLE-2	off	2,42	3364
off		MTF-1	RLE-2	LZC	2,47	142
text	Slovo	MTF-2	RLE-2	off	2,47	4141
text	Slovo	MTF	RLE-2	LZSS	2,65	4141
text	Slovo	off	RLE-2	off	2,66	4141
text	Slabika	off	RLE-2	LZC	3,03	3364

Tabuľka 15 : Kompresné pomery a veľkosti abecied pre algoritmus RLE-2 nad Calgary korpusom.

XML$_{cz}$ korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Slabika	MTF	RLE-2	off	0,71	41542
text	Slovo	MTF-1	RLE-2	off	0,71	80132
xml	Slabika	MTF	RLE-2	off	0,72	34068
xml	Slovo	MTF	RLE-2	off	0,73	78714
off		MTF	RLE-2	off	0,76	241
text	Znak	MTF	RLE-2	LZC	0,76	394
text	Znak	MTF	RLE-2	off	0,76	394
off		MTF-2	RLE-2	LZC	0,77	241
xml	Slovo	off	RLE-2	off	0,86	78714
text	Slovo	MTF	RLE-2	LZSS	0,96	80132
text	Znak	off	RLE-2	LZSS	1,44	394
off		off	RLE-2	LZSS	1,45	241

Tabuľka 16 : Kompresné pomery a veľkosti abecied pre algoritmus RLE-2 nad XMLcz korpusom.

Algoritmus RLE-2 dosahuje oproti algoritmu RLE-1 na Calgary korpuse o 0,04 bpB lepší najlepší kompresný pomer. O 0,01 až 0,02 pbB sa zlepšil kompresný pomer na XML$_{cz}$ korpuse. V percentuálnom vyjadrení došlo k rovnakému, približne 1,5% zlepšeniu.

Je veľmi zaujímavé, že toto zlepšenie sa týka iba nastavení, v ktorých sa nepoužívajú slovníkové metódy. Toto zlepšenie je spoločné pre všetky nastavenia, ktoré nevyužívajú slovníkové metódy. Keďže algoritmus RLE-2 predstavuje jedinú zmenu oproti nastaveniam testovaných s algoritmom RLE-1 a pri absencii slovníkovej metódy sa za ním nachádza len binárny kóder, je potrebné hľadať príčinu tohto zlepšenia v ňom. Pri algoritme RLE-2 zrejme dochádza k lepšiemu prerozdeleniu frekvencií jednotlivých symbolov abecedy, ktoré sú následne lepšie zakódovateľné binárnym kóderom.

Pri slovníkových metódach nám ale výrazné zväčšenie abecedy algoritmom RLE-2 oproti abecede používanej pri ostatných rovnakých nastaveniach algoritmom RLE-1 spôsobilo rovnaké rozhádzanie kontextu a nemohli sa využiť výhody slovníkových metód.

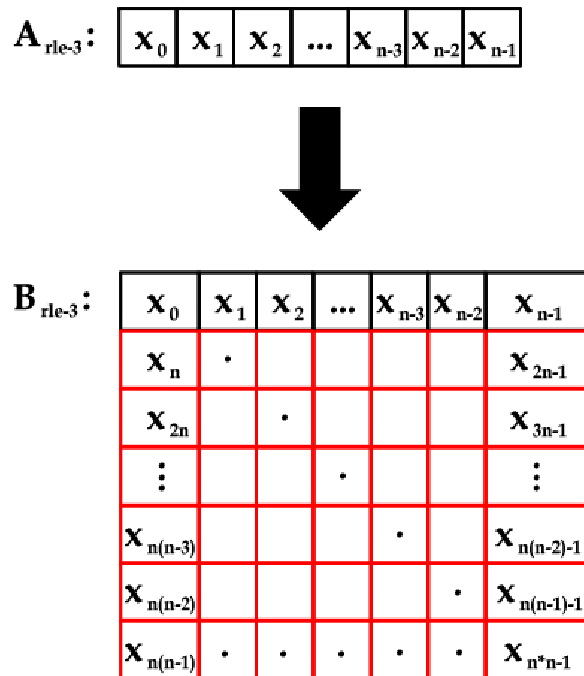
Okrem popísanej zmeny platia všetky pozorovania ako pri algoritme RLE-1 z kapitoly 5.1.1.

5.3. Run-length encoding 3

Algoritmus Run-length encoding 3, ďalej nazývaný ako RLE-3, v podstate ďalej rozvíja myšlienku RLE-2. Podobne ako pri RLE-2, ktoré modifikovalo algoritmus RLE-1 takým spôsobom, aby došlo k zníženiu počtu znakov, ktoré sú potrebné k zakódovaniu dlhého behu symbolu x_i, algoritmus RLE-3 modifikuje RLE-2 tak, aby nám namiesto dvoch symbolov stačil symbol jeden.

To sa dosiahne pridaním veľkého počtu špeciálnych symbolov. Každý špeciálny symbol bude niesť informáciu o tom, akého symbolu x_i dlhý beh kóduje a takisto aj dĺžku tohto behu. Ukážeme si najprv, ako vznikne výsledná abeceda. Pred algoritmom RLE-3 máme reťazec X_{rle-3}, v ktorom sú symboly z abecedy $A_{rle-3} = \{ x_0, x_1, x_2, ..., x_{n-3}, x_{n-2}, x_{n-1} \}$. Nová abeceda B_{rle-2} , z ktorej prvkov sa bude reťazec X_{rle-3} skladať bude vyzerať: $B_{rle-3} = \{ x_0, x_1, x_2, ..., x_{n-3}, x_{n-2}, x_{n-1}, x_n, x_{n+1}, x_{n+2}, ..., x_{2n-3}, x_{2n-2}, x_{2n-1}, x_{2n}, x_{2n+1}, x_{2n+2}, ..., x_{3n-3}, x_{3n-2}, x_{3n-1}, x_{n(n-1)}, x_{n(n-1)+1}, x_{n(n-1)+2}, ..., x_{n\times n-3}, x_{n\times n-2}, x_{n\times n-1} \}$.

Do abecedy B_{rle-3} teda pridáme $n(n - 1)$ špeciálnych symbolov. Čo reprezentuje špeciálny symbol x_i? Nech $i = k\times n + j$, kde $n = |A_{rle-3}|$, $0 < k < n$ a $0 \leq j \leq |A_{rle-3}| - 1$. Potom špeciálny symbol x_i reprezentuje dlhý beh symbolu x_j s dĺžkou k.



Obrázok 15 : Modifikácia vstupnej abecedy Arle-2 na výstupnú abecedu Brle-2 algoritmom RLE-2

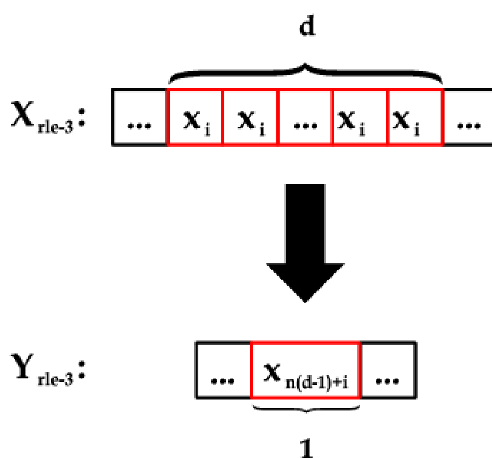
Vzhľadom na to, že pracujeme s veľkou abecedou, ktorá dosahuje až približne 80 000 symbolov, implementácia, ktorá by dodržovala vytváranie abecedy špeciálnych symbolov o takejto veľkosti by bola priestorovo veľmi zložitá. Vzhľadom na použitie štatistických kóderov na konci procesu kódovania ako Huffmanove kódovanie alebo aritmetického kódera je potrebné udržiavať tabuľku frekvencií jednotlivých symbolov.

Ako riešenie by sa dala uplatniť bitová mapa, ktorú by sme si na začiatku pripravili a každé políčko by predstavovalo napr. 64 symbolov. Prvé políčko by nám reprezentovalo, či má aspoň jeden symbol z množiny $\{ x_0, x_1, x_2, ..., x_{61}, x_{62}, x_{63} \}$ nenulovú frekvenciu. Druhé políčko by nám reprezentovalo, či má aspoň jeden symbol z množiny $\{ x_{64}, x_{65}, x_{66}, ..., x_{125}, x_{126}, x_{127} \}$ nenulovú frekvenciu. Ďalšie políčka ďalej, až pokiaľ by sme neprekročili veľkosť abecedy B_{rle-3}. Následne, ak bude v nejakom políčku mať hodnotu NULL, budeme vedieť, že každý symbol z množiny, ktorú dané políčko reprezentuje má zatiaľ nulovú frekvenciu. Ak bude mať niektorý prvok nenulovú frekvenciu, políčko bude obsahovať ukazovateľ na alokovanú pamäť, v ktorej bude uložená reprezentácia frekvencií pre danú množinu.

V projekte XBW je zvolený iný prístup. Vzhľadom na to, že iba málo dlhých behov dosahuje dĺžku viac ako 100, stačí nám obmedziť pridanie nových symbolov na reprezentovanie dlhých behov o dĺžke menej ako 1000.

Pri zakódovaní rozdeľujeme výstup z kódera algoritmu RLE-3 nasledovne:

- ak je na vstupe beh symbolu x_i s dĺžkou 1, čo je vlastne situácia, keď je na vstupe symbol x_i a za ním je symbol x_j a j sa nerovná i. Potom na výstup dáme symbol x_i.
- ak je na vstupe beh symbolu x_i s dĺžkou d, na výstup dáme špeciálny symbol x_j, ktorého index vypočítame podľa vzorca $j = n\times(d - 1) + i$, kde $n = |A_{rle-3}|$. Ako si môžeme všimnúť, tak daný vzorec platí aj pre beh symbolu x_{64} s dĺžkou $d = 1$, pretože $j = n\times(1-1) + i = i$.



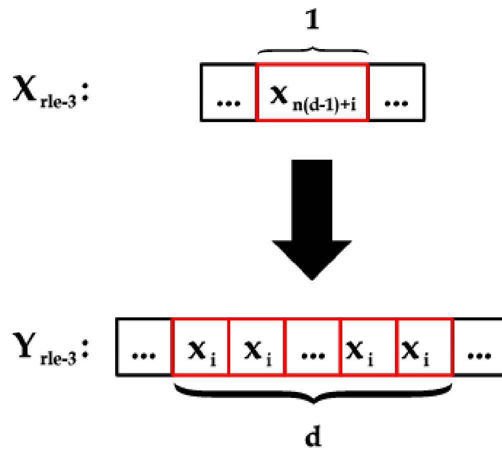
Obrázok 16 : Zakódovanie dlhého behu jedným symbolom algoritmom RLE-3.

Takto vyzerá algoritmus RLE-3 na zakódovanie reťazca X_{rle-3}:

RLE-3 zakóduj(vstup, velkosť abecedy):

```
dĺžka = 1
while(na vstupe je symbol){
        načítaj symbol a označ ho x
        while(existuje na vstupe ďalší symbol a je rovnaký ako symbol x
a dĺžka < veľkosť_abecedy)
                dĺžka = dĺžka + 1
        nový_symbol = veľkosť_abecedy*(dĺžka - 1) + index symbolu x
        na výstup vypíš nový_symbol
}
```

Pri odkódovaní načítame symbol x_j. Index j je v tvare $j = n\times k + i$, kde $0 \leq k < n$ a $0 \leq i < n$. Podľa toho ako sme všetky, aj jednotkové behy zakódovali vieme, že dĺžka behu $d = k + 1$ a jedná sa o beh symbolu x_i. Na výstup teda d-krát vypíšeme symbol x_i.



Obrázok 17 : Odkódovanie symbolu na dlhý beh algoritmom RLE-2

Algoritmus pre odkódovanie reťazca X_{rle-3} vyzerá nasledovne:

RLE-3 odkóduj(vstup, velkosť abecedy):

```
while(na vstupe je symbol){
        načítaj symbol a označ ho y
        i = y mod veľkosť_abecedy
        d = (y - i)/veľkosť_abecedy + 1
        for(d-krát)
                vypíš na výstup symbol z abecedy s indexom i
        }
        else
                vypíš na výstup symbol y
}
```

5.2.1. Výsledky kompresie nad korpusmi pre algoritmus RLE-3

Calgary korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
off		MTF-2	RLE-3	off	2,44	142
text	Znak	MTF-2	RLE-3	off	2,44	142
xml	Znak	MTF-2	RLE-3	off	2,44	142
off		MTF-1	RLE-3	off	2,45	142
text	Znak	MTF-1	RLE-3	off	2,45	142
xml	Znak	MTF-1	RLE-3	off	2,45	142
off		MTF-2	RLE-3	LZC	2,47	142
text	Slovo	MTF	RLE-3	LZSS	2,65	4141
xml	Slovo	MTF-1	RLE-3	LZSS	2,65	4141
xml	Slovo	MTF-2	RLE-3	LZSS	2,65	4141
xml	Slabika	MTF-1	RLE-3	LZSS	2,67	3363
text	Slabika	MTF-2	RLE-3	LZSS	2,67	3364
xml	Slabika	MTF	RLE-3	LZSS	2,68	3363
off		off	RLE-3	off	2,76	142

text	Slovo	MTF	RLE-3	off	25,22	4141
xml	Slovo	MTF	RLE-3	off	25,22	4141

Tabuľka 17 : Kompresné pomery a veľkosti abecied pre algoritmus RLE-3 nad Calgary korpusom.

XML$_{cz}$ korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Znak	MTF	RLE-3	LZC	0,76	394
off		MTF	RLE-3	LZC	0,77	241
off		MTF-1	RLE-3	LZC	0,77	241
off		MTF-2	RLE-3	LZC	0,77	241
off		MTF	RLE-3	off	0,77	241
text	Slovo	MTF	RLE-3	LZC	0,78	80132
xml	Znak	MTF-1	RLE-3	LZC	0,79	722
text	Slabika	MTF-2	RLE-3	LZC	0,79	35436
xml	Znak	MTF	RLE-3	off	0,79	722
xml	Slovo	MTF-1	RLE-3	LZC	0,8	78714
xml	Slabika	MTF	RLE-3	LZC	0,81	34068
off		off	RLE-3	off	0,87	241
text	Slovo	MTF	RLE-3	LZSS	0,96	80132
text	Slovo	off	RLE-3	off	4,75	80132

Tabuľka 18 : Kompresné pomery a veľkosti abecied pre algoritmus RLE-3 nad XML$_{cz}$ korpusom.

Podobne ako pri algoritme RLE-2 nám algoritmus RLE-3 zmenil kompresné pomery iba pri nastaveniach, v ktorých sa nevyužívali slovníkové metódy. Preto výrazné zlepšenie nastavení programu používajúcich slovníkové metódy nemôžeme pripísať ich úspešnosti pre algoritmus RLE-3, ale musíme ich pripísať neúspešnosti nastavení programov, ktoré nepoužívajú slovníkové metódy.

Nastavenia používajúce algoritmus RLE-3 dosahujú pri najlepšom kompresnom pomere na Calgary korpuse o 0,19 bpB horší kompresný pomer ako najlepší pomer pre algoritmus RLE-1. Pre XML$_{cz}$ korpus je to 0,07 bpB. Pre obe to znamená približne 11% zhoršenie oproti kompresnému pomeru pre algoritmus RLE-1.

Ako vidno v oboch tabuľkách, algoritmus Move-to-front a jeho varianty pri použití algoritmu RLE-3 majú na kompresný pomer slovníkových metód rovnaký charakter. Pri absencii algoritmu Move-to-front alebo jeho varianty ale nastáva zhoršenie kompresného pomeru slovníkových metód. Dá sa to vysvetliť veľkosťou abecedy, ktorá je pri algoritme RLE-3 obrovská a tým pádom varianty algoritmu Move-to-front, ktoré sa zameriavajú skôr na lokálne opakujúce sa symboly sú pri takejto veľkej abecede neúčinné.

Až trojnásobné zväčšenie skomprimovaného súboru pre najhoršie nastavenie obsahujúce algoritmus RLE-3 môžeme pripísať ohromne veľkej abecede, ktorej symboly sú následne málo frekventované. Binárny kóder dostane veľa symbolov, ktoré nedokáže efektívne zakódovať.

Celkovo sa ukázal algoritmus RLE-3 ako zlou variantou, ktorá prináša oproti ostatným variantám algoritmu Run-length encoding iba zhoršenie pre nastavenia nepoužívajúce slovníkové metódy. Nemala by byť preto používaná.

5.4. Run-length encoding EXP

Algoritmus Run-length encodin EXP, ďalej označovaný ako RLE-EXP, navrhol vo svojej práci Juergen Abel [5]. Pri použití špeciálneho symbolu na označenie a zakódovanie dlhého behu tento špeciálny symbol naruší prirodzený lokálny kontext. Namiesto označenia dlhého behu špeciálnym symbolom sa používa takzvaný prahový beh. Ako ukázal Maniscalca [6] vo svojej práci, implementácie, ktoré používajú prahový beh dosahujú lepšie výsledky. Algoritmus RLE-EXP využíva prahový beh.

Prahový beh pozostáva z dvoch častí a síce dlhého behu pevnej dĺžky t a dlhého behu premenlivej dĺžky e a slúži na zakódovanie dlhého behu symbolu x_i. Obe časti sú behmi symbolu x_i. Premenlivá dĺžka e je definovaná rovnicou:

$$e = \lfloor \log_2 (L-1) \rfloor$$

Parameter L je dĺžka dlhého behu symbolu x_i. Aby sa zakódovala celá informácia o dĺžke dlhého behu je potrebné zachytiť ešte nejakú informáciu, pretože momentálne parametre dávajú dĺžku pre pevnú hodnotu parametrov t a e v rozmedzí $\{t + 2^e - 1, t + 2^e, ..., t + 2^{e+1} - 2\}$. Vieme, že výsledná dĺžka sa nachádza v tejto množine, ale nevieme, ktorá to je. Autor preto použil dodatočnú informáciu v podobe behu bitov b s dĺžkou e, ktorý sa podobá na druhú časť Eliášovho kódovania. Beh bitov b zakóduje dĺžku v rozmedzí $\{0, ..., 2^e - 1\}$. Preto výsledná informácia zakóduje dĺžku pre pevnú hodnotu parametrov t a e v rozmedzí $\{t + 2^e - 1, t + 2^e, ..., t + 2^{e+1} - 2\}$ tak, že vieme presne určiť, ktorá akú dĺžku z danej množiny dlhý beh nadobúda.

Ak by sme beh bitov b s dĺžkou e dali v zakódovanej správe za prahový beh, narušil by prirodzený lokálny kontext symbolov. Preto sú jednotlivé behy bitov ukladané do separátneho dátového prúdu, ktorý sa nazýva RMB, podľa anglického RLE Mantissa Buffer. Dáta, ktoré sú ukladané do RMB sa neposúvajú do ďalších fáz kódovania, sú rovno dané koncovému binárnemu kóderu na spracovanie.

Dlhý beh symbolu x_i s dĺžkou L je teda zakódovaný prahovým behom symbolu x_i, kde pevná časť má dĺžku t a premenlivá má dĺžku e a behom bitov b s dĺžkou e. Hlavne pri dlhých behoch veľkej dĺžky je vďaka svojej logaritmickej štruktúre obzvlášť účinný.

Pôvodný dlhý beh	L	e	b (binárne zapísaný)	Prahový beh
aa	2	0	-	aa
aaa	3	1	0	aaa
aaaa	4	1	1	aaa
aaaaa	5	2	00	aaaa
aaaaaa	6	2	01	aaaa
aaaaaaa	7	2	10	aaaa
aaaaaaaa	8	2	11	aaaa
aaaaaaaaa	9	3	000	aaaaa
aaaaaaaaaa	10	3	001	aaaaa

Tabuľka 19 : Prahový beh pre parameter t = 2

5.5. Run-length encoding BIT

V tejto metóde popísanej Juergenom Abelom [5] sa používajú špeciálne symboly. Vzhľadom na to, že pridanie špeciálneho symbolu naruší lokálny kontext symbolov, ak je následne spracovaný algoritmom Move-to-front alebo nejakou jeho alternatívou, Abel navrhol spôsob ako k takému prípadu nedôjsť.

Algoritmus Run-length encoding BIT, ďalej nazývanom RLE-BIT, je rozdelený na dva algoritmy a síce na RLE-BIT0 a RLE-BIT1. Algoritmus RLE-BIT0 je použitý pred a algoritmus RLE-BIT1 je použitý po algoritme Move-to-front alebo jeho alternatíve.

Aby sa zabránilo narušeniu lokálneho kontextu symbolov a zároveň zmenšil tlak vytváraný dlhými behmi, sú tieto dlhé behy algoritmom RLE-BIT0 uložené dočasne v bufferi TB, kde sa uchováva výskyt dlhého behu v pôvodnom reťazci a takisto jeho dĺžka. Dlhý

reťazec je následne z reťazca vymazaný, až na prvý symbol. Výstup z algoritmu RLE-BIT0 preto neobsahuje žiadne dlhé behy.

Po fáze algoritmu Move-to-front alebo jeho alternatívy sú do reťazca na základe údajov z dočasného buffera TB algoritmom RLE-BIT1 vložené informácie o dlhých behov. Na správne rozpoznanie pôvodných dlhých behov algoritmom RLE-BIT1 potrebujeme, aby vstupný reťazec X_{gtr} bol rovnako dlhý výstupný reťazec Y_{gtr} , kde GTR je globálna transformácia reťazca zabezpečovaná algoritmom Move-to-front alebo jeho variantmi. Preto sa algoritmus RLE-BIT1 nemôže použiť napríklad s algoritmom Inversion Frequencies namiesto algoritmu Move-to-front.

Dlhé behy sú zakódované špeciálnymi symbolmi 0 a 1. Ak je dĺžka dlhého behu L, je spôsobom, ktorý sa podobá na druhú časť Eliášovho kódu zakódované číslo L – 1. V nasledujúcej tabuľke môžeme vidieť zopár príkladov kódovania algoritmom RLE-BIT1:

Pôvodný dlhý beh	L	Kódovanie dlhého behu
aa	2	a0
aaa	3	a1
aaaa	4	a00
aaaaa	5	a01
aaaaaa	6	a10
aaaaaaa	7	a11
aaaaaaaa	8	a000
aaaaaaaaa	9	a001
aaaaaaaaaa	10	a010

Tabuľka 20 : Kódovanie dlhých behov algoritmom RLE-BIT1

5.6. Dopad variánt algoritmu Run-length encoding na komprimáciu

Nasledujúce tabuľky ukazujú kompresný pomer pri nezapojení algoritmu Run-length encoding alebo nejakej jeho varianty:

Calgary korpus

Parser	Varianta	Varianta	Slovníková	Kompresný	Priemerná

Typ parsovania	Spôsob parsovania	MTF	RLE	metóda	pomer	veľkosť abecedy
off		MTF-2	off	LZC	2,47	142
off		MTF-1	off	LZC	2,47	142
off		MTF	off	LZC	2,48	142
text	Znak	MTF-1	off	LZC	2,48	142
text	Slabika	MTF-1	off	off	2,64	3236
text	Slabika	MTF-2	off	off	2,64	3236
text	Slovo	MTF-2	off	off	2,64	4141
text	Slovo	MTF-1	off	LZSS	2,65	4141
text	Slovo	MTF-3	off	LZSS	2,65	4140
text	Slovo	off	off	LZSS	2,71	4141
off		off	off	LZC	2,84	142
text	Znak	off	off	off	10,06	394

Tabuľka 21 : Kompresné pomery a veľkosti abecied bez prítomnosti algoritmus RLE alebo nejakej jeho varianty nad Calgary korpusom.

XML_{cz} korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	Spôsob parsovania					
text	Znak	MTF	off	LZC	0,76	394
off		MTF-1	off	LZC	0,77	241
text	Znak	MTF-2	off	LZC	0,77	394
text	Slovo	MTF	off	LZC	0,78	80132
xml	Znak	MTF-2	off	LZC	0,78	722
text	Slabika	MTF-1	off	LZC	0,79	35436
xml	Slovo	MTF	off	LZC	0,8	78714
xml	Slabika	MTF-2	off	LZC	0,81	34068
off		off	off	LZC	0,94	241
text	Slovo	MTF-1	off	LZSS	0,97	80132
xml	Slovo	MTF	off	off	0,97	78714
text	Slabika	MTF	off	off	0,98	35436
xml	Slabika	off	off	off	11,16	34068
text	Znak	off	off	odff	11,48	394

Tabuľka 22 : Kompresné pomery a veľkosti abecied bez prítomnosti algoritmus RLE alebo nejakej jeho varianty nad XML_{cz} korpusom.

Ako sa ukazuje pri použití algoritmu Run-legth encoding a jeho variánt dostávame lepšie výsledky na oboch korpusoch. To platí dokonca aj pre algoritmus RLE-3, ktorý oproti algoritmom RLE-1 a RLE-2 dostáva podstatne horšie výsledky.

Vzhľadom na to, zlepšenie alebo zhoršenie kompresného pomeru pre jednotlivé varianty bol rovnomerný aj na malých súboroch, aj na veľkých súboroch, môžeme jednotlivé varianty algoritmu Run-length encoding zoradiť nasledovne:

1. RLE-2
2. RLE-1 dosahuje zhoršenie kompresného pomeru oproti RLE-2 približne o 1,5%
3. RLE-3 dosahuje zhoršenie kompresného pomeru oproti RLE-2 približne o 19%

Preto najlepšou variantou algoritmu Run-length encoding je varianta RLE-2.

Pri prítomnosti slovníkovej metódy počas komprimácie na variante algoritmu RLE alebo jeho nepoužití nezáleží. Kompresný pomer slovníkových metód ostáva stále rovnaký. Z toho sa dá usúdiť, že prítomnosť či neprítomnosť dlhých behov slovníkové metódy nevedia nijako využiť.

Z tabuliek neplynie žiadna súvislosť medzi veľkosťou abecedy a použitím alebo vynechaním algoritmu RLE alebo jeho variánt.

6. Slovníkové algoritmy

Slovníkové algoritmy sú bezstratové kompresné algoritmy založené na myšlienke využitia odkazov do slovníka. Pri zakódovaní a takisto pri odkódovaní si obe metódy vytvárajú slovník na základe už spracovaného textu. V tomto slovníku sa nachádzajú časti už spracovaného textu a pri výskyte reťazca, ktorý sa už nachádza v slovníku sa pri zakódovaní jeho výskyt v reťazci nahradí odkazom na jeho umiestnenie v slovníku. Pri odkódovaní sa naopak v slovníku nájde reťazec, na ktorý sa odkazujeme a odkaz sa nahradí reťazcom.

Práve slovník má pri kódovaní hlavnú úlohu. Je dôležité, aby bol slovník v rovnakých fázach pri zakódovaní a odkódovaní rovnaký, aby sa nestalo, že sa odkazujeme na neexistujúci reťazec. Takisto dôležitá vlastnosť, ktorá výrazne zvyšuje mieru kompresie je vytváranie slovníka počas procesu zakódovania alebo odkódovania. Nie je teda potrebné prikladať k výslednému zakódovanému reťazcu slovník, pretože ten sa vytvára za behu.

Medzi algoritmy, ktoré využívajú slovník, patria aj algoritmy LZ77 a LZ78 vymyslené Abrahamom Lempelom a Jacobom Zivom v rokoch 1977 a 1978. Obidva algoritmy tvoria základ pre veľkú rodinu bezstratových kompresných algoritmov známu ako rodinu LZ algoritmov. Všetky algoritmy v tejto rodine sú určitou modifikáciou zmienených algoritmov.
Tieto algoritmy sú veľmi populárne a často používané v kompresných programoch pre ich veľmi dobrý pomer medzi časovou náročnosťou a kompresiou. Nie sú náročné na systémové požiadavky, špeciálne nemajú veľkú pamäťovú zložitosť. V pôvodných implementáciach sa pracuje s abecedou pevnej dĺžky, väčšinou o veľkosti 256 znakov. V našej implementácii používame abecedu premenlivej veľkosti, preto je pamäťová zložitosť O(n), kde n je veľkosť abecedy.

Algoritmus LZ77 nemá slovník, má takzvané pohyblivé okno, v ktorom vyhľadáva a odkazuje sa na reťazce v už spracovanom texte. Je to ale ekvivalentné použitiu slovníka.

Zameriame sa na dve odvodeniny týchto algoritmov a síce LZC a LZSS, ktoré som implementoval v projekte XBW a modifikoval pre veľké abecedy.

6.1. Algoritmus LZ78

Algoritmus LZC prvý krát uviedli Lempel a Ziv [2]. Vznikol ako jednoduchá modifikácia algoritmu LZW a takisto ako algoritmus LZW je odvodeninou algoritmu LZ78. Najprv vysvetlíme algoritmus LZ78, algoritmus LZC dostaneme jeho modifikáciou.

Kompresný algoritmus LZ78 patrí medzi algoritmy, ktoré na uschovanie indexov používajú indexáciu podľa prefixu a zväčšujúci sa slovník. Slovník je tvorený nenulovou množinou prefixov a každý prefix obsiahnutý v slovníku má unikátny index. To neplatí na začiatku, kedy je slovník pre algoritmus LZ78 prázdny.

Hlavná idea algoritmu LZ78 je založená na pozorovaní, že určité podreťazce sa zvyknú v reťazci opakovať. Toto tvrdenie platí najmä pre textové súbory, keďže tie sú tvorené slovami a tie sa zvyknú opakovať. Samozrejme to nie je doménou len textových súborov, platí to pre väčšinu reťazcov. To nám dáva možnosť takéto podreťazce dávať do slovníka, indexovať ich a následne vo výstupnom reťazci namiesto daného podreťazca použiť adekvátny index. Samozrejme univerzálne rozlíšiť slová nie je ľahké, pretože tento algoritmus nie je využívaný len pre texty. Využíva sa preto metóda prefixov a namiesto slov sa do slovníku ukladajú prefixy. Pre lepšie pochopenie metódy prefixov si predstavme, že sme v procese zakódovania reťazca. Máme vytvorený slovník podreťazcov na základe už spracovanej časti reťazca. Snažíme sa nájsť čo najdlhší reťazec v slovníku, taký, že je zároveň začiatkom reťazca na vstupe. Tvorí teda prefix pre zatiaľ nespracovanú časť reťazca. Odtiaľ pochádza pomenovanie prefixovej metódy. Do výsledného reťazca potom dáme index na daný prefix.

Na začiatku je slovník prefixov prázdny. Pre lepšiu pochopiteľnosť si predstavme, že sa nachádzame v štádiu, keď slovník už obsahuje nejaké prefixy. Začneme analyzovať reťazec na vstupe a začneme ho porovnávať s prefixom P, ktorý je na začiatku prázdny reťazec. Označme si prvý symbol v spracovávanom reťazci S. Skúsime nájsť v slovníku prefix P+S, ktorý vytvoríme zreťazením prefixu P a symbolu S, teda v prvom kroku to je iba prvý symbol práve spracovaného reťazca. Ak taký prefix nájdeme v slovníku, nastavíme P = S a do S priradíme nasledujúci symbol v spracovávanom reťazci. Takto pokračujeme, až nebudeme vedieť v slovníku nájsť prefix P+S. Našli sme teda v slovníku prefix P s maximálnou dĺžkou taký, že je zároveň začiatkom spracovaného reťazca. Na výstup dáme dvojicu (I, S), kde I je index prefixu P v slovníku a S je prvý symbol za prefixom v spracovávanom reťazci. Do slovníka

pridáme prefix v tvare P+S, ktorý dostaneme zreťazením prefixu P a symbolu S. V spracovávanom reťazci sa posunieme na symbol S a znova začneme hľadať najdlhší prefix v slovníku, ktorý sa zhoduje so začiatkom spracovávaného reťazca.

Vďaka tomu, že do slovníka postupne pridávame prefixy v tvare P+S, kde P je prefix, ktorý sa už v slovníku nachádza, tak v prípade, ak sa v slovníku nachádza prefix v tvare $x_0x_1...x_{n-1}x_n$, tak sa tam určite nachádza aj každý prefix v tvare $x_0x_1...x_{i-1}x_i$, kde $0 < i < n$, teda každý prefix reťazca $x_0x_1...x_{n-1}x_n$. Preto ak nájdeme prefix s maximálnou dĺžkou, môžeme si byť istý, že tam neexistuje dlhší prefix $x_0x_1...x_{m-1}x_m$ ktorý by bol rovnaký ako začiatok spracovaného reťazca, ako ten čo sme našli. Ak by existoval, existovala by aj postupnosť prefixov $x_0x_1...x_{i-1}x_i$ taká, že $n \le i \le m$, ktorou by sme sa naším algoritmom dostali od reťazca $x_0x_1...x_{i-1}x_i$ k reťazcu $x_0x_1...x_{m-1}x_m$.

Špeciálny prípad nastáva na začiatku, keď nemáme ešte žiaden prefix v slovníku. V tom prípade nemôžeme nájsť žiaden prefix, na ktorý by začínal spracovávaný reťazec. Preto na výstup dáme dvojicu (0, S), kde S je prvý symbol spracovávaného reťazca. V podstate to znamená, že prázdny reťazec má v slovníku index 0, ale tento prefix sa v slovníku neudržuje.

V pseudokóde vyzerá zakódovanie algoritmom LZ78 nasledovne:

lz78 zakóduj(vstup):

```
slovník a prefix P sú prázdne
while(je na vstupe symbol){
        S = symbol na vstupe
        while(reťazec P+S je v slovníku){
                P = P+S
                S = nasledujúci znak na vstupe za symbolom S
        }
        daj na výstup dvojicu (index prefixu P v slovníku, S)
        pridaj do slovníka prefix P+S
}
```

Definícia:

Krok v algoritme LZ78 nazývame postupnosť operácií vykonávaných algoritmom LZ78. Krok začína načítaním prvého symbolu v spracovávanom reťazci, pokračuje nájdením najdlhšieho reťazca, výpisom dvojice na výstup a končí pridaním nového prefixu do slovníka.

Pre názornosť ukážeme, ako sa zakóduje reťazec “cabbadcabbadcab“

Pozícia symbolu v reťazci	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Symbol na danej pozícii	c	a	b	b	a	d	c	a	b	b	a	d	c	a	b

Tabuľka 23 : Reťazec pred zakódovaním algoritmom LZ78

Číslo kroku	Pozícia	Výstup
1	1	(0, c)
2	2	(0, a)
3	3	(0, b)
4	5	(3, a)
5	6	(0, d)
6	8	(1, a)
7	10	(3, b)
8	12	(2, d)
9	13	(6, b)

Tabuľka 24 : Proces zakódovania reťazca z tabuľky 23 algoritmom LZ78. V tabuľke sa nachádza pre každé číslo kroku aktuálna pozícia v zakodúvanom reťazci a výstup v danom kroku

Index prefixu	Prefix
1	c
2	a
3	b
4	ba
5	d
6	ca
7	bb
8	ad
9	cab

Tabuľka 25 : Slovník, ktorý vznikne pri zakódovaní reťazca z tabuľky 23 algoritmom LZ78

Odkódovanie je pomerne jednoduché. Načítavame dvojice (I, S) zo vstupu, v slovníku si nájdeme podľa indexu I prislúchajúci prefix P, na výstup dáme prefix P a do slovníka pridáme prefix P+S. Ako sme už poznamenali index 0 reprezentuje prázdny reťazec.

V pseudokóde vyzerá odkódovanie pre algoritmus LZ78 nasledovne:

```
lz78 odkóduj(vstup):

slovník a prefix P sú prázdne
while(je na vstupe dvojica (I, S)){
        načítaj dvojicu (I, S) zo vstupu
        nájdi v slovníku prefix P prislúchajúci indexu I
        daj na výstup prefix P
        pridaj do slovníka prefix P+S
}
```

Zásadná vec pri odkódovaní je budovanie slovníka tak, aby sme v rovnakom kroku pri zakódovaní a odkódovaní mali slovník v rovnakej podobe. To, že slovník budeme mať identický v každom kroku dokážeme matematickou indukciou.

Veta 1: Slovníky pre zakódovanie a odkódovanie sú v rámci jedného kroku rovnaké.

Dôkaz:

V prvom kroku máme aj pri zakódovaní a aj pri dekódovaní prázdny slovník a vieme, že prázdny reťazec je v tomto slovníku reprezentovaný indexom 0. V prvom kroku indukcie teda máme slovníky rovnaké.

Zoberme si teraz n-tý krok zakódovania a odkódovania a predpokladajme, že slovníky v týchto krokoch sú rovnaké. Keďže dvojica (I, S) je pri zakódovaní daná na výstup na konci kroku, takže prefix s indexom I musí byť prítomný v slovníku pre odkódovanie už na začiatku kroku, teda podľa indukčného predpokladu sa nachádza na začiatku kroku aj v slovníku pre odkódovanie. Takže na konci kroku pri odkódovaní môžeme do slovníka pridať prefix P+S, pretože obidva reťazce poznáme, rovnako ako pri zakódovaní. Na konci kroku sú teda obidva slovníky rovnaké.

Pre názornosť odkódujeme reťazec "0a0b0c3a0d1a3b2d6b", ktorý sme dostali ako výsledok operácie zakóduj pre algoritmus LZ78:

Pozícia v reťazci	1-2	3-4	5-6	7-8	9-10	9-10	11-12	13-14	15-16
Dvojica (I, S)	(0, c)	(0, a)	(0, b)	(3, a)	(0, d)	(1, a)	(3, b)	(2, d)	(6, b)

Tabuľka 26 : Zakódovaný reťazec z tabuľky 23 algoritmom LZ78

Číslo kroku	Dvojica (I, S)	Výstup
1	(0, c)	c
2	(0, a)	a
3	(0, b)	b
4	(3, a)	ba
5	(0, d)	d
6	(1, a)	ca
7	(3, b)	bb
8	(2, d)	ad
9	(6, b)	cab

Tabuľka 27 : Proces odkódovania reťazca z tabuľky 26 algoritmom LZ78. V tabuľke sa nachádza výstup pre každú zakódovanú dvojicu.

Index prefixu	Prefix
1	c
2	a
3	b
4	ba
5	d
6	ca
7	bb
8	ad
9	cab

Tabuľka 28 : Slovník, ktorý vznikne pri odkódovaní reťazca z tabuľky 26 algoritmom LZ78

Okrem popísaných vlastností je v algoritme LZ78 zachytená aj správa slovníka. Pre slovník je vyhradená určitá pamäť a pri vyčerpaní pamäte sú použité rôzne stratégie. Zvykne sa vymazať celý slovník a začína sa budovať nový, vypúšťajú sa frázy, ktoré sa vyskytovali najmenej alebo frázy, ktoré neboli najdlhšie použité. Vymazanie celého slovníka a budovanie slovníka od začiatku nastáva tiež pri zhoršení kompresného pomeru.

V projekte XBW ale tieto vlastnosti neboli implementované. Pamäťová náročnosť slovníka je oproti potrebám najviac pamäťovo náročných komponentov zanedbateľná.

6.1.1. Algoritmus LZC

Pri algoritme LZ78 sú potrebné na zakódovanie podreťazca, ktorý sa nachádza ako prefix v slovníku, index I daného prefixu P a nasledujúci symbol S v reťazci. Ak by sa podarilo zakódovať podreťazec iba pomocou indexu do slovníka, bolo by to výrazne zlepšenie kompresného pomeru. Zakódovanie dvojicou (I, S) nastáva hlavne kvôli prázdnemu slovníku na začiatku. Nemôžeme dávať na výstup iba indexy I prefixov P, pokiaľ sa žiaden prefix v strome na začiatku nevyskytuje. Túto situáciu ale rieši algoritmus LZC, spolu s ďalšími vylepšeniami.

Algoritmus LZC sa oproti algoritmu LZ78 líši v týchto bodoch:

- na začiatku sa začína so slovníkom, v ktorom sú všetky symboly použitej abecedy
- na výstup sa dávajú iba indexy prefixov v slovníku, nedáva sa nasledujúci symbol
- indexy sú kódované binárnym kódom s rastúcou dĺžkou

Oproti algoritmu LZ78 je teda potrebné na začiatku poznať abecedu. Ak je abeceda pevnej dĺžky 256 znakov, na začiatku je slovník naplnený všetkými 256-timi znakmi a indexy prefixov sa odkazujú na ne.

V prípade abecedy premenlivej dĺžky ako je to v projekte XBW je to však zložitejšie. Nevieme, ktoré znaky sú na začiatku v abecede a posielanie zakódovanej abecedy na začiatku zakódovaného reťazca tiež nie je najšťastnejšie riešenie. V projekte XBW preto existuje parser, ktorý vytvorí abecedu tak, aby tvorili množinu prirodzených čísel v tvare {0, 1, 2, ..., n – 2, n – 1}, kde n je počet znakov v abecedy. Daná množina spĺňa podmienku, že sa tam nachádza

každé prirodzené číslo medzi najväčším a najmenším z tejto abecedy. Pre vytvorenie abecedy nám tak stačí vedieť veľkosť abecedy.

Pri vypnutí parsera pri komprinovaní je súbor načítaný ako reťazec znakov z množiny {0, 1, 2, ..., 254, 255}, ktorá tiež spĺňa podmienku, že sa v nej nachádza každé prirodzené číslo medzi 0 a 255.

Zakódovanie je pomerne jednoduché. Na začiatku inicializujeme slovník vložením všetkých znakov z abecedy $A_{lzc} = \{0, 1, 2, ..., n - 2, n - 1\}$. Nájdeme najdlhší prefix P v slovníku, ktorý je zároveň začiatok spracovávaného reťazca, vypíšeme jeho index I a do slovníka pridáme nový prefix P+S, kde S je nasledujúci symbol za prefixom P v spracovávanom reťazci.

V pseudokóde vyzerá funkcia zakóduj pre algoritmus LZC nasledovne:

lzc zakóduj(vstup, veľkosť abecedy):

```
daj do slovníka všetky symboly abecedy  Alzc = {0, 1, 2, ..., veľkosť_abecedy
- 2, veľkosť_abecedy - 1}
prefix P je prázdny
while(je na vstupe symbol){
        S = symbol na vstupe
        while(reťazec P+S je v slovníku){
                P = P+S
                S = nasledujúci znak na vstupe za symbolom S
        }
        daj na výstup index prefixu P v slovníku
        pridaj do slovníka prefix P+S
}
```

Rovnako ako pri algoritme LZ78 si ukážeme zakódovanie reťazca “cabbadcabbadcab”

Pozícia symbolu v reťazci	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Symbol na danej pozícii	c	a	b	b	a	d	c	a	b	b	a	d	c	a	b

Tabuľka 29 : Reťazec pred zakódovaním algoritmom LZC

Index v slovníku	1	2	3	4
Symbol na danej pozícii	a	b	c	d

Tabuľka 30 : Slovník pre algoritmus LZSS po inicializácii

Číslo kroku	Pozícia	Výstup

Číslo	Index prefixu	Prefix

1	1	3
2	2	1
3	3	2
4	4	2
5	5	1
6	6	4
7	7	5
8	9	7
9	11	9
10	13	11

Tabuľka 31 : Proces zakódovania reťazca z tabuľky 29 algoritmom LZC. V tabuľke sa nachádza pre každé číslo kroku aktuálna pozícia v zakodúvanom reťazci a výstup v danom kroku

kroku		
0	1	a
0	2	b
0	3	c
0	4	d
1	5	ca
2	6	ab
3	7	bb
4	8	ba
5	9	ad
6	10	dc
7	11	cab
8	12	bba
9	13	adc

Tabuľka 32 : Slovník, ktorý vznikne pri zakódovaní reťazca z tabuľky 29 algoritmom LZC

Pri odkódovaní pre algoritmus LZC prečítame index I_i zo vstupu, nájdeme príslušný prefix P v slovníku a dáme ho na výstup. Prečítame ďalší index $I_{i+1} = x_0x_1...x_{n-1}x_n$ zo vstupu, zoberieme prvý symbol x_0 a do slovníka dáme prefix $I_i + x_0$. Môže sa však stať, že prefix s indexom I_{i+1} sa v slovníku ešte nevyskytuje. Potom platí nasledujúca veta.

Veta 2:

Nech posledný pridaný prefix P má index I_i, je v tvare aW, kde a je symbol z abecedy A_{lzc} a W je reťazec symbolov z abecedy A_{lzc}. Potom, ak pri dekódovaní algoritmom LZC dostaneme na vstupe index I_{i+1}, ktorý sa nevyskytuje v abecede, je to index prefixu v tvare aWa.

Dôkaz:

Nech I_i je index posledne pridaného prefixu v slovníku pri odkódovaní algoritmom LZC. Zrejme pre všetky ostatné indexy I_j v slovníku platí, že $j < i$. Nech prefix pre index I_i v slovníku má tvar $x_0x_1...x_{n-1}x_n$. Nachádzame sa v momente, keď sme si zo vstupu prečítali index I_i, našli si odpovedajúci reťazec v slovníku, vypísali ho na výstup a chceme pridať nový prefix P+S do slovníka. Na to potrebujeme prečítať prvý symbol z prefixu, ktorý je

reprezentovaný ďalším indexom na vstupe, pretože presne takýto prefix je v tejto fáze vložený do slovníka pri zakódovaní.

Pozrime sa podrobnejšie na to, čo sa deje pri zakódovaní. Najdlhší prefix, ktorý je zhodný so začiatkom spracovaného reťazca má index I_i v slovníku a tvar $x_0x_1...x_{n-1}x_n$. Za týmto reťazcom je symbol x_j. Do slovníka sa preto dá prefix $x_0x_1...x_{n-1}x_nx_j$ a dostane index I_{i+1}. Aký index pre najdlhší prefix zo slovníku, ktorý je rovnaký ako začiatok ďalej spracovaného reťazca teda kóder vypíše na výstup, ak ho ešte dekóder vo svojom slovníku nemá a aký reťazec reprezentuje?

Keďže pri odkódovaní sme už mali prefix s indexom I_i v slovníku, tak je zrejmé že jediný prefix, ktorý ešte v slovníku dekóder nemá je prefix s indexom I_{i+1}. Preto v pôvodnom reťazci musí byť za reťazcom $x_0x_1...x_{n-1}x_n$ prefix s indexom I_{i+1}, ktorý sme práve pridali. Ale ako vieme, prefix s indexom I_{i+1} je v tvare $x_0x_1...x_{n-1}x_nx_j$. V pôvodnom reťazci teda zakodúvavame reťazec $x_0x_1...x_{n-1}x_nx_0x_1...x_{n-1}x_nx_j$, ktorý je tvorený prefixmi s indexmi I_i a I_{i+1}. Keď si ale uvedomíme, že symbol za prefixom I_i je symbol x_0, tak je zrejmé že prefix s indexom I_{i+1} je v tvare $I_i + x_0 = x_0x_1...x_{n-1}x_nx_0$. Z toho vyplýva, že $x_j = x_0$ a prefix s indexom I_{i+1} má tvar $x_0x_1...x_{n-1}x_nx_0$. Potom $a = x_0$ a $W = x_0x_1...x_{n-1}x_n$, čím sme dokázali vetu 2.

Z vety 2 vyplýva, že pri odkódovaní dostaneme na vstupe index na prefix, ktorý sa v slovníku ešte nevyskytuje, tak zoberieme naposledy pridaný prefix P, zoberieme jeho prvý symbol x_0 a do slovníka pridáme prefix $P + x_0$.

V pseudokóde vyzerá odkódovanie nasledovne:

```
lzc odkóduj(vstup, veľkosť abecedy):

daj do slovníka všetky symboly abecedy  Alzc = {0, 1, 2, ..., veľkosť_abecedy
- 2, veľkosť_abecedy - 1}
prefix P je prázdny
reťazec posledný_prefix je prázdny
while(je na vstupe index I){
        nájdi v slovníku prefix P prislúchajúci indexu I
        daj na výstup prefix P
        if(je na vstupe další index I2){
                if(existuje prefix pre index I2 v slovníku){
                        nájdi v slovníku prefix P2 prislúchajúci indexu I2
                        označ prvý symbol prefixu P2 ako a
                        pridaj do slovníka prefix P+a
                        posledný_prefix = P+a
                }
                else{
                        označ prvý symbol reťazca posledný_prefix ako a
                        pridaj do slovníka prefix posledný_prefix+a
                        posledný_prefix = posledný_prefix+a
                }
```

```
        }
}
```

Pre názornosť ukážeme proces odkódovania na výstupe operácie zakóduj algoritmu LZC. Slovník pre proces odkódovania je po inicializácii rovnaký ako pri procese kódovania, môžeme ho nájsť v tabuľke 32.

Pozícia v reťazci	1	2	3	4	5	6	7	8	9	10
Dvojica (I, S)	3	1	2	2	1	4	5	7	9	11

Tabuľka 33 : Zakódovaný reťazec z tabuľky 12 algoritmom LZC

Číslo kroku	Pozícia	Výstup
1	1	c
2	2	a
3	3	b
4	4	b
5	5	a
6	6	d
7	7	ca
8	8	bb
9	9	ad
10	10	cab

Tabuľka 34 : Proces odkódovania reťazca z tabuľky 33 algoritmom LZC. V tabuľke sa nachádza pozícia, na ktorej sa nachádzame v zakódovanom reťazci a výstup pre každú zakódovanú dvojicu.

Číslo kroku	**Index prefixu**	**Prefix**
0	1	a
0	2	b
0	3	c
0	4	d
1	5	ca
2	6	ab
3	7	bb
4	8	ba
5	9	ad
6	10	dc
7	11	cab
8	12	bba
9	13	adc

Tabuľka 35 : Slovník, ktorý vznikne pri odkódovaní reťazca z tabuľky 33 algoritmom LZC

6.1.3. Výsledky kompresie nad korpusmi pre algoritmus LZC

Calgary korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					

off		MTF-1	RLE-1	LZC	2,47	142
off		MTF-1	RLE-2	LZC	2,47	142
off		MTF-1	RLE-3	LZC	2,47	142
off		MTF-2	off	LZC	2,47	142
off		MTF-2	RLE-1	LZC	2,47	142
off		MTF	RLE-2	LZC	2,48	142
text	Znak	MTF-2	RLE-3	LZC	2,48	142
text	Slabika	MTF-1	off	LZC	2,75	3364
text	Slovo	MTF-1	RLE-1	LZC	2,75	4141
text	Slovo	off	RLE-2	LZC	2,8	4141
off		off	RLE-3	LZC	2,82	142
text	Slabika	off	off	LZC	3,03	3364

Tabuľka 36 : Kompresné pomery a veľkosti abecied pre algoritmus LZC nad Calgary korpusom.

XML_{cz} korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Znak	MTF	RLE-1	LZC	0,76	394
off		MTF	RLE-2	LZC	0,77	241
off		MTF-1	RLE-3	LZC	0,77	241
off		MTF-2	off	LZC	0,77	241
xml	Znak	MTF	RLE-1	LZC	0,78	722
xml	Znak	MTF-1	RLE-2	LZC	0,78	722
xml	Znak	MTF-2	RLE-3	LZC	0,78	722
text	Slabika	MTF	off	LZC	0,79	35436
xml	Slovo	MTF-1	RLE-1	LZC	0,8	78714
xml	Slabika	MTF-2	RLE-2	LZC	0,81	34068
off		off	RLE-3	LZC	0,94	241
xml	Slabika	off	off	LZC	1,05	34068

Tabuľka 37 : Kompresné pomery a veľkosti abecied pre algoritmus LZC nad XML_{cz} korpusom.

V kapitole 4.6. sme dospeli k záveru, že použitie algoritmu Move-to-front alebo nejakej jeho varianty zlepšuje kompresný pomer algoritmu LZC, ale na konkrétnej variante nezáleží. V kapitole 5.6. sme dospeli k záveru, že pri použití algoritmu LZC nezáleží na variante alebo prítomnosti algoritmu RLE, kompresný pomer pre algoritmus LZC sa nezmení.

Kompresné pomery a tabuľky pre algoritmus LZC len potvrdzujú tieto závery, ktoré sme spravili v predchádzajúcich kapitolách. Okrem prítomnosti alebo neprítomnosti algoritmu MTF alebo jeho varianty ovplyvňuje výsledný kompresný pomer len veľkosť abecedy, ktorú dostaneme po fáze Burrows-Wheelerovej transformácie. Vo všeobecnosti pre väčšiu abecedu je dosahovaný algoritmom LZC horší kompresný pomer.

6.2. Algoritmus LZ77

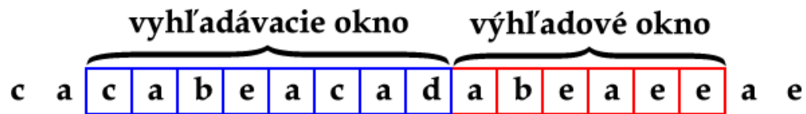
Algoritmus LZSS bol vyvinutý Storerom a Szymanskim [17] v roku 1982. Patrí do rodiny bezstrátových kompresných algoritmov LZ. Je to preto slovníková metóda a konkrétne modifikácia algoritmu LZ77. Vysvetlíme si preto algoritmus LZ77 a následné zmeny tak, aby sme dostali algoritmus LZSS.

Algoritmus LZ77 funguje na rozdiel od algoritmu LZ78 na princípe posuvného okna. Pri algoritme LZ78 sa snaží reťazec, ktorý má na vstupe zakódovať pomocou prefixov zo slovníka, jeho záujem sa v podstate sústreďuje na budúce symboly. Pri algoritme LZ77 sa záujem sústreďuje na znaky, ktoré už boli spracované, hľadá a reťace sa zakóduje pomocou nich.

Algoritmus LZ77 kóduje odkázaním sa na podreťazec v minulosti, teda taký, ktorý bol v danom momente spracovaný kóderom, aj dekóderom. Zakódovaný je najdlhší podreťazec zo začiatku práve spracovaného reťazca na vstupe taký, že sa vyskytuje v už spracovanom texte. Podreťazec je vlastne zakódovaný odkazom na svoju kópiu spracovanú v minulosti. Zakóduváva sa dvojicou čísel (V, D), kde V je vzdialenosť začiatku kópie v spracovanom texte od začiatku práve spracovaného reťazca na vstupe a D je dĺžka kópie v spracovanom texte.
Táto interpretácia je síce jednoduchšia na pochopenie, ale správnejšia je interpretácia, že pre symbol na začiatku spracovávaného reťazca a D-1 nasledujúcich symbolov platí, že V symbolov dozadu je v reťazci ten istý znak. Jasnejšie to ukazuje príklad, kedy pre dvojicu (V, D) platí V < D.

Pre kóder aj dekóder je definované takzvané posuvné okno, v rámci ktorého sa vykonáva vyhľadávanie vhodného podreťazca v spracovanom reťazci. Toto okno je pevnej dĺžky. Vyhľadávanie v už spracovanom reťazci bez obmedzenia by bolo síce výhodne, mali by sme možnosť nájsť dlhšie podreťazce, ale pre zväčšujúcu sa veľkosť súboru by bolo pomalé. Jeho rozdelenie na vyhľadávacie a výhľadové okno určuje maximálnu veľkost pre V a takisto pre D. Tým pádom vieme maximálnu dĺžku V a D nastaviť pre lepšie zakódovanie binárnym kóderom. V projekte XBW sa používa 4kB okno.

Kódovaným reťazcom sa pohybuje posuvné okno, ktoré je rozdelené na vyhľadávacie okno a výhľadové okno.



Obrázok 18 : Rozdelenie posuvného okna na vyhľadávacie a výhľadové

V rámci výhľadového okna sa vyhľadávajú podreťazce v spracovanom texte a určuje sa maximálna vzdialenosť V, v ktorej sa môže nachádzať začiatok najdlhšieho reťazca v spracovanom texte, ktorý je zhodný so začiatkom spracovávaného reťazca.
Výhľadové okno začína prvým symbolom spracovávaného reťazca a určuje maximálnu dĺžku nájdeného podreťazca v spracovanom texte.

Myšlienka zakódovania je pomerne jednoduchá. Na začiatku vyplníme vyhľadávacie okno so samými medzerami a výhľadové okno vyplníme symbolmi zo vstupného reťazca. Potom nájdeme vo vyhľadávacom okne maximálny reťazec taký, aby bol rovnaký ako začiatok spracovávaného reťazca, teda reťazca vo výhľadovom okne. Na výstup dáme trojicu (V, D, S), kde (V, D) je dvojica charakterizujúca nájdený maximálny reťazec vo vyhľadávacom okne a S je symbol, ktorý za týmto reťazcom nasleduje v spracovávanom reťazci.

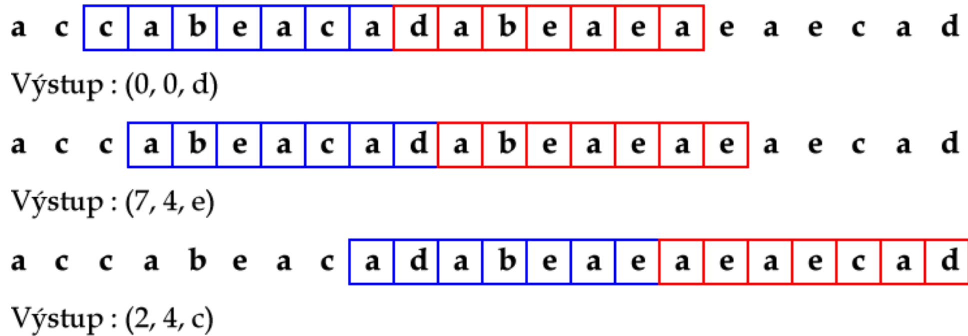
V pseudokóde vyzerá funkcia zakóduj pre algoritmus LZ77 takto:

lz77 zakóduj(vstup):

```
naplň vyhľadávacie okno samými nulami a výhľadové okno symbolmi z vstupu
D = 0
while(je na vo výhľadovom okne symbol){
        V = dĺžka výhľadového okna
        maximálna_dvojica = (0,0)
        while(V > 0){
                V1 = V
                if(symbol na mieste D v spracovávanom reťazci nie je za
                koncom spracovávaného reťazca, je rovnaký ako symbol
                V symbolov dozadu a D <= dĺžka výhľadového okna)
                        D = D + 1
                if(druhá zložka v dvojici maximálna_dvojica < D)
                        maximálna_dvojica = (V, D)
                V = V - 1
                D = 0
        }
        V1 = prvá zložka z dvojice maximálna_dvojica
        D1 = druhá zložka z dvojice maximálna_dvojica
        S = symbol na mieste D1 + 1 v spracovávanom reťazci
        daj na výstup trojicu ( V1, V2, S)
        posuň vstup o D1 + 1 symbolov, za symbol S
}
```

Najviac časovo náročnou časťou je vyhľadanie najdlhšieho reťazca vo vyhľadávacom okne. Vyhľadávanie ako je naznačené v pseudokóde je sekvenčné, a preto aj veľmi pomalé.

Pre názornosť ukážeme zopár krokov v procese zakódovania:



Obrázok 19 : Proces zakódovania reťazca algoritmom LZ77

Odkódovanie prebieha oveľa rýchlejšie a jednoduchšie ako zakódovanie. Na vstupe dostane dekóder trojicu (V, D, S). Vráti o V symbolov v odkódovanom reťazci naspäť a na koniec momentálne odkódovaného reťazca sa pridá postupne D nasledujúcich symbolov. Potom sa na koniec momentálne odkódovaného reťazca pridá symbol S a načíta sa ďalšia trojica.

V pseudokóde funkcia odkóduj vyzerá takto:

```
lz77 odkóduj(vstup):

while(je na vstupe trojica (V, D, S)){
        načítaj trojicu (V, D, S) zo vstupu
        označ pozíciu o V symbolov dozadu oproti koncu odkódovaného reťazca
        ako P
        for(D-krát){
                vypíš na koniec reťazca symbol na pozícii P
                P = pozícia symbolu za symbolom na pozícii P
        }
        vypíš na koniec reťazca symbol S
}
```

Ukážeme si proces odkódovania časti reťazca, ktorý sme zakódovávali:

Momentálne odkódovaný reťazec:

a c c a b e a c a

Vstup : (0, 0, d)

a c c a b e a c a d

Vstup : (7, 4, e)

a c c a b e a c a d a b e a e

Vstup : (2, 4, c)

a c c a b e a c a d a b e a e a

a c c a b e a c a d a b e a e a e

a c c a b e a c a d a b e a e a e a

a c c a b e a c a d a b e a e a e a e

a c c a b e a c a d a b e a e a e a e c

Obrázok 20 : Proces odkódovania reťazca zakódovaného v obrázku 18

6.2.1. Algoritmus LZSS

Algoritmus LZSS je odvodený od algoritmu LZ77. Oproti algoritmu LZ77 má tieto modifikácie:

- namiesto trojicou (V, D, S) kódujeme len dvojicou čísel (V, D)
- zakódovajú sa len podreťazce, ktorých vypísanie by zabralo viac ako ich zakódovaná verzia
- bitový indikátor, či na vstupe následuje dvojica (V, D) alebo symbol
- výhľadové okno je reprezentované ako cyklický buffer
- slovník je reprezentovaný ako binárny vyhľadávací strom

Podobne ako pri jednotlivých variáciách pre algoritmus run-length encoding a aj ako pri algoritme LZC je tu tendencia čo najviac zmenšiť výstup. V podstate prvé tri modifikácie dokopy smerujú k tomu, aby zbytočne nekódovali reťazce, ktoré sa výhodnejšie dajú zakódovať ako znaky. Výhodu nám dáva zmenšenie z trojice symbolov na dvojicu symbolov alebo jeden symbol. Používa sa na to bitový indikátor b_symbol, ktorý má hodnotu 0, ak nasleduje symbol a bitový indikátor b_dvojica, ktorý má hodnotu 1, ak nasleduje dvojica symbolov. V podstate dostávame v prípade b_symbol dvojicu symbolov (b_symbol, x_i) alebo trojicu symbolov (b_dvojica, V, D) v prípade b_dvojica. Nesmieme zabudnúť na to, že binárny indikátor je zakódovaný jediným bitom.

Ak ide o zakódovanie jedného symbolu, teda ak nenájdeme vo vyhľadávacom okne žiaden reťazec a na vstupe je symbol x_0, tak by sme ho pomocou LZ77 zakódovali ako (0, 0, x_0), ale pomocou LZSS by sme ho zakódovali ako (b_symbol, x_0). Bitový indikátor je reprezentovaný jedným bitom. Dosahuje dva stavy a špecifikuje či nasleduje symbol alebo dvojica. Aj keby sme pomocou binárneho kódera v ďalšej fáze kódovania zakódovali 0 iba jedným bitom, zapísanie trojice by bolo priestorovo náročnejšie ako zapísanie bitového indikátoru a symbolu.

Pre zakódovanie dvoch symbolov x_0x_1, ktoré v LZ77 zakódujeme ako (V, 1, x_1), kde V je vzdialenosť symbolu x_0 od začiatku výhľadového okna. Pomocou LZ77 ich už zakódujeme ako (b_dvojica, V, 2). Vzhľadom na to, že v projekte XBW sú vzdialenosti a dĺžky ukladané do dvoch rôznych adaptívnych Huffmanových stromov, bude bitový zápis trojice (b_dvojica, V, 2) pre LZSS menší ako bitový zápis trojice (V, 1, x_1) pre LZ77.
Už pri zakódovaní dvoch symbolov je pri algoritme LZSS výhodnejšie použiť binárny indikátor b_dvojica a zakódovať to ako trojicu namiesto zakódovania týchto symbolov ako dvoch samostatných symbolov.

Pre funkciu zakóduj vyzerá pseudokód takto:

lzss zakóduj(vstup):

```
naplň vyhľadávacie okno samými nulami a výhľadové okno symbolmi z vstupu
D = 0
while(je na vo výhľadovom okne symbol){
        V = dĺžka výhľadového okna
        maximálna_dvojica = (0,0)
        while(V > 0){
                V1 = V
                if(symbol na mieste D v spracovávanom reťazci nie je za
                koncom spracovávaného reťazca, je rovnaký ako symbol
                V symbolov dozadu a D <= dĺžka výhľadového okna)
                        D = D + 1
                if(druhá zložka v dvojici maximálna_dvojica < D)
                        maximálna_dvojica = (V, D)
                V = V - 1
                D = 0
        }
        V1 = prvá zložka z dvojice maximálna_dvojica
        D1 = druhá zložka z dvojice maximálna_dvojica
        if(D1 < 2){
                S = symbol na mieste V symbolov od začiatku spracovávaného
                reťazca
                daj na výstup dvojicu (0, S)
        }
        else
                daj na výstup trojicu (0, V1, D1)
```

```
        posuň vstup o D1 symbolov
}
```

V pseudokóde sme pre jednoduchosť uviedli sekvenčné vyhľadávanie, algoritmus LZSS ale implementuje binárne vyhľadávacie stromy. Ukážeme si príklad kódovania pomocou algoritmu LZSS na rovnakom príklade ako pri algoritme LZ77:

a c | c | a | b | e | a | c | a | d | a | b | e | a | e | a | e a e c a d

Výstup : (0, d)

a c c | a | b | e | a | c | a | d | a | b | e | a | e | a | e | a e c a d

Výstup : (1, 7, 4)

a c c a b e a | c | a | d | a | b | e | a | e | a | e | a | e | c | a | d

Výstup : (1, 2, 5)

Obrázok 21 : Zakódovanie reťazca pomocou algoritmu LZSS

Rovnako ako pri algoritme LZ77 je odkódovanie jednoduché a omnoho rýchlejšie ako zakódovanie. Pseudokód pre funkciu odkóduj algoritmu LZSS vyzerá takto:

lzss odkóduj(vstup):

```
while(na vstupe je binárny indikátor){
        if(binárny indikátor = 0){
                načítaj symbol S
                vypíš symbol S na koniec odkódovaného reťazca
        }
        else{
                načítaj dvojicu (V, D)
                označ pozíciu o V symbolov dozadu oproti koncu odkódovaného
                reťazca ako P
                for(D-krát){
                        vypíš na koniec odkódovaného reťazca symbol na
                        pozícii P
                        P = pozícia symbolu za symbolom na pozícii P
                }
        }
}
```

Ukážeme si proces odkódovania časti reťazca, ktorý sme zakódovávali:

Momentálne odkódovaný reťazec:

a c c a b e a c a

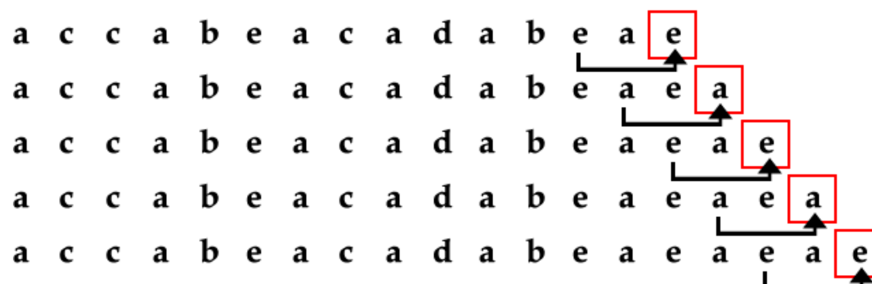
Vstup : (0, d)

a c c a b e a c a d

Vstup : (1, 7, 4)

a c c a b e a c a d a b e a

Vstup : (1, 2, 5)



Obrázok 22 : Odkódovanie zakódovaného reťazca podľa obrázku 21 algoritmom LZSS

6.2.2. Výsledky kompresie nad korpusmi pre algoritmus LZSS

Calgary korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Slovo	MTF-1	RLE-1	LZSS	2,65	4141
text	Slovo	MTF-2	RLE-2	LZSS	2,65	4141
text	Slovo	MTF	RLE-3	LZSS	2,66	4141
text	Slabika	MTF-1	off	LZSS	2,67	3364
text	Slabika	MTF	RLE-1	LZSS	2,69	3364
text	Slovo	off	RLE-2	LZSS	2,71	4141
text	Znak	MTF-3	RLE-3	LZSS	2,84	142
off		MTF-2	off	LZSS	2,85	142
off		MTF-1	RLE-1	LZSS	2,85	142
text	Slabika	off	RLE-2	LZSS	2,85	3364
text	Znak	off	RLE-3	LZSS	2,95	142

Tabuľka 38 : Kompresné pomery a veľkosti abecied pre algoritmus LZSS nad Calgary korpusom.

XML_{cz} korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Slovo	MTF	RLE-1	LZSS	0,96	80132
text	Slovo	MTF-1	RLE-2	LZSS	0,97	80132
text	Slovo	MTF-2	RLE-3	LZSS	0,97	80132
xml	Slovo	MTF	RLE-1	LZSS	0,99	78714

xml	Slovo	MTF-1	RLE-2	LZSS	1,00	78714
xml	Slovo	MTF-2	RLE-3	LZSS	1,00	78714
text	Slabika	MTF	off	LZSS	1,03	35436
xml	Slabika	MTF	RLE-1	LZSS	1,03	34068
text	Slovo	off		LZSS	1,07	80132
xml	Znak	MTF-1	RLE-2	LZSS	1,3	722
xml	Znak	MTF-2	RLE-3	LZSS	1,31	722
off		MTF	off	LZSS	1,42	241
text	Znak	MTR-2	RLE-1	LZSS	1,42	394
off		MTF	RLE-2	LZSS	1,42	241
off		off	RLE-3	LZSS	1,45	241

Tabuľka 39 : Kompresné pomery a veľkosti abecied pre algoritmus LZSS nad XML_{cz} korpusom.

V kapitole 4.6. sme dospeli k záveru, že použitie algoritmu Move-to-front alebo nejakej jeho varianty zlepšuje kompresný pomer algoritmu LZSS, ale na konkrétnej variante nezáleží. V kapitole 5.6. sme dospeli k záveru, že pri použití algoritmu LZSS nezáleží na variante alebo prítomnosti algoritmu RLE, kompresný pomer pre algoritmus LZSS sa nezmení.

Kompresné pomery a tabuľky pre algoritmus LZSS len potvrdzujú tieto závery, ktoré sme spravili v predchádzajúcich kapitolách. Okrem prítomnosti alebo neprítomnosti algoritmu MTF alebo jeho varianty ovplyvňuje výsledný kompresný pomer len veľkosť abecedy, ktorú dostaneme po fáze Burrows-Wheelerovej transformácie. Vo všeobecnosti pre väčšiu abecedu je dosahovaný algoritmom LZSS lepší kompresný pomer.

6.3. Dopad slovníkových algoritmov LZC a LZSS na komprimáciu

Nasledujúce tabuľky ukazujú kompresný pomer pri nezapojení slovníkových metód:

Calgary korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
off		MTF	RLE-2	off	2,25	142
off		MTF-1	RLE-2	off	2,26	142
off		MTF-2	RLE-2	off	2,27	142
text	Znak	MTF-2	RLE-2	off	2,27	142
text	Znak	MTF-1	RLE-2	off	2,27	142
off		MTF-2	RLE-2	off	2,29	142
text	Slabika	MTF-1	RLE-1	off	2,42	3364
off		MTF-2	RLE-3	off	2,44	142
off		MTF-1	RLE-3	off	2,45	142
text	Slovo	MTF-2	RLE-2	off	2,47	4141

text	Slovo	off	RLE-3	off	25,31	4141

Tabuľka 40 : Kompresné pomery a veľkosti abecied bez použitia slovníkových algoritmov nad Calgary korpusom.

XML$_{cz}$ korpus

Parser		Varianta MTF	Varianta RLE	Slovníková metóda	Kompresný pomer	Priemerná veľkosť abecedy
Typ parsovania	**Spôsob parsovania**					
text	Slabika	MTF	RLE-2	off	0,71	35346
text	Slovo	MTF	RLE-1	off	0,71	80132
xml	Slabika	MTF	RLE-1	off	0,72	34068
xml	Slovo	MTF	RLE-1	off	0,73	78714
text	Slabika	MTF-1	RLE-2	off	0,73	35346
xml	Slabika	MTF-2	RLE-2	off	0,75	34068
off		MTF	RLE-2	off	0,76	241
text	Znak	MTF	RLE-2	off	0,76	394
off		MTF	RLE-3	off	0,77	241
xml	Znak	MTF	RLE-2	off	0,77	722
text	Slovo	off	RLE-2	off	0,86	80132
text	Slovo	MTF	off	off	0,93	80132
xml	Slabika	off	off	off	11,16	34068
text	Znak	off	off	off	11,48	394

Tabuľka 41 : Kompresné pomery a veľkosti abecied bez použitia slovníkových algoritmov nad XMLcz korpusom.

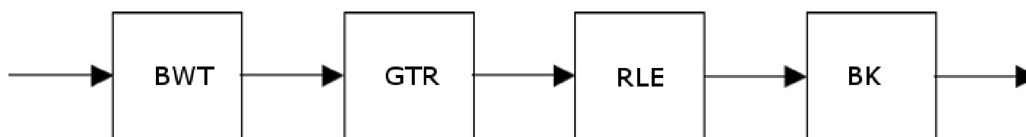
Najlepšie kompresné pomery pri použití algoritmu LZC sú horšie o 0,22 bpB na Calgary korpuse a o 0,05 bpB na XML$_{cz}$ korpuse. Je to približne o 11% horšia hodnota oproti najlepším kompresným pomerom bez použitia algoritmu LZC.

Najlepšie kompresné pomery pri použití algoritmu LZSS sú horšie o 0,4 bpB na Calgary korpuse a o 0,25 bpB na XML$_{cz}$ korpuse. Je to zhoršenie o 17 až 34% oproti najlepším dosahovaným kompresným pomerom.

S veľkou abecedou sa dobre vysporiadajú aj ostatné fázy pred a po Burrows-Wheelerovej transformácii. Nie je na to potrebná slovníková metóda. Oproti najlepším kompresným pomerom zapojením slovníkového algoritmu tieto kompresné pomery pokazíme. Pri veľkých súboroch je bez použitia slovníkového algoritmu vhodnejšia veľká abeceda a pri malých naopak malá abeceda.

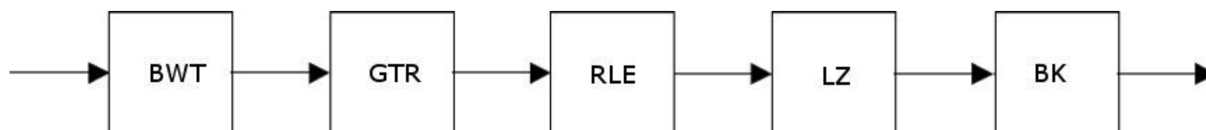
7. Programy založené na Burrows-Wheelerovej transformácii

Môžeme povedať, že momentálne najrozšírenejší model pre spracovanie reťazca po Burrows-Wheelerovej transformácii má nasledujúci charakter:



kde BWT je Burrows-Wheelerova transformácia, GTR je globálna transformácia reťazca, v našom prípade je to Move-to-front algoritmus a jeho rôzne modifikácie, nasleduje fáza RLE a ukončuje to binárny kóder (BK).

V našom prípade máme pred binárnym kóderom slovníková metóda z rodiny LZ algoritmov a síce buď LZSS alebo LZW modifikovanými pre zvládnutie veľkej abecedy. Tieto metódy používajú na zápis do súborov metódy binárneho kódera, v našom prípade to je aritmetický kodér a Huffmanové stromy.



Medzi momentálne používané programy, ktoré využívajú ako najdôležitejší algoritmus pri kompresii práve Burrows-Wheelerovu transformáciu patria bzip2, ABC alebo szip. V krátkosti si predstavíme dané programy a pozrieme sa na postupnosť algoritmov, ktoré nasledujú po Burrows-Wheelerovej transformácii.

7.1. Bzip2

Bzip2 je open-sourcový projekt voľne šíriteľný pod BSD licenciou. Je vyvíjaný Julianom Sewardom a jeho prvá verzia sa objavila v júli roku 1996, odvtedy je neustále vyvíjaný a zlepšovaný. Momentálne posledná verzia 1.0.4 bola vydaná 20. decembra 2006.

V porovnaní so známymi kompresnými programami ako gzip alebo ZIP je síce pomalší, ale zato oveľa efektívnejší na väčšine súborov, najmä veľkých textových súborov. Keď ho porovnáme s kompresnými metódami, ktoré využívajú algoritmy LZMA alebo PPM, tieto ho porážajú v ohľade kompresnej efektivity. Podľa autora Juliana Sewarda je bzip2 o 10 až 15 percent horší ako programy používajúce PPM, vynahrádza si to však časom, ako dlho beží a pri kompresii je bzip2 približne 2 krát rýchlejší a pri dekompresii dokonca až 6 krát[1].

Text prechádza počas komprimácie programom bzip2 viacerými vrstvami, ktoré ho upravujú a komprimujú. Takto vyzerá postupnosť jednotlivých vrstiev po načítaní textu:

1. **Run-length Encoding**
2. **Burrows-Wheelerova transformácia**
3. **Move-to-front**
4. **Run-length Encoding**
5. **Huffmanovo kódovanie** – je to druh štatistického kódovania. Po RLE fáze máme postupnosť čísel, väčšinou interpretovanú 8-bitovým typom int. V Huffmanovom kódovaní zistíme frekvenciu každého symbolu v kódovanom reťazci X_{hk}. Na základe tejto frekvencie priradíme najfrekventovanejším symbolom bitovú reprezentáciu s pomerne malým počtom bitov, pohybuje sa okolo 2-3 bitov a symbolom s nízkou frekvenciou priradíme bitovú reprezentáciu s vyšším počtom bitov.
6. **Viaceré Huffmanové stromy** – v bzip2 sa používa viacero Huffmanových stromov, môže ich byť až 6. Abeceda A_{hk} obohacuje abecedu B_{rle} o symbol, ktorý symbolizuje ukončenie reťazca, takže ju zväčšuje o 1, teda $|B_{rle}| = |A_{hk}| + 1$. Algoritmus Move-to-front a Burrows-Wheelerova transformácia zachovávajú veľkosť abecedy a RLE v prvom a štvrtom kroku zhodne zväčšujú abecedu o špeciálny escape symbol, ktorý značí začiatok behu. Takže zjavne $|B_{1.rle}| = |A_{1.rle}| + 1 = |A_{bwt}| = |B_{bwt}| = |A_{mtf}| = |B_{mtf}| = |A_{2.rle}|$, $|B_{2.rle}| = |A_{2.rle}| + 1 = |A_{hk}|$ a $|B_{hk}| = |A_{hk}| + 1 = |A_{hs}| = |B_{hs}|$. Keďže do prvého RLE vstupuje abeceda znakov, je zrejmé že $|B_{1.rle}| < 257$, takže o abecede B_{hs} vieme povedať, že $|B_{hs}| < 260$. V žiadnom Huffmanovom strome nie je viac ako 50 symbolov. Pred zakódovaním nasledujúcich 50 znakov je zvolený Huffmanov strom, pre ktorý sa dosiahne najlepšia kompresia.
7. **Unárne kódovanie pre číslo Huffmanovho stromu** – jednotlivé Huffmanové stromy sú očíslované a uložené v Move-to-front zozname. Ich indexy môžu byť 0 až 5. Na

[1] http://en.wikipedia.org/wiki/Bzip2

začiatku každého 50 fragmentu o veľkosti 50 symbolov je unárne zakódovaný index Huffmanovho stromu, ktorý sa má pre daný fragment použiť. Tieto stromy sú udržiavané pomocou algoritmu Move-to-front, ktorý využíva skutočnosť, že ak je na predchádzajúci fragment použitý Huffmanov strom s indexom i, je pravdepodobné, že ďalší fragment bude zakódovaný tiež Huffmanovým stromom s indexom i.

8. **Delta kódovanie** – pomocou delta kódovania bzip2 zakóduje jednotlivé Huffmanové stromy. Tie sú reprezentované ako tabuľky, v ktorých sú zoradené prvky podľa dĺžky kódu ich bitovej reprezentácie. Ako sme spomenuli pri Huffmanovom kódovaní, niektoré symboly sú reprezentované 2-3 bitmi iné napríklad 20 bitmi. Pri delta kódovaní je každý symbol uložený ako prírastok hodnoty oproti hodnote predošlého symbolu. Ak sú za sebou dva symboly s rovnakou dĺžkou kódu ich bitovej reprezentácie, na výstup sa vypíše číslo 0. Prvé číslo má svoju vlastnú hodnotu. Takže číselná postupnosť 4,4,5,6,8,11 by bola zakódovaná ako 4,0,1,2,2,3.
9. **Riedka bitová mapa** – ako poslednú fázu používa bzip2 riedku bitovú mapu na zachytenie toho, či sa daný znak nachádza v niektorom Huffmanovom strome. Uloženie do obyčajnej bitovej mapy, kde nám 256 bitov určuje, či je prislúchajúci znak prítomný v niektorom strome alebo nie, je pri použití znakov z pomerne malej oblasti, ako to býva pri obyčajnom texte, pomerne neefektívne. Preto sa využíva metóda riedkej bitovej mapy a bitová mapa je rozdelená na 16 podoblastí a daná podoblasť je zahrnutá jedine v prípade, ak je v reťazci prítomný aspoň jeden znak z tejto oblasti. Preto je pred reprezentáciou bitovej mapy 16 bitov, ktoré reprezentujú riedku bitovú mapu.

7.2. ABC

ABC je skratka pre Advanced Blocksorting Compressor. Je to voľne šíriteľný projekt pod licenciou Abel Public License (APL). Autorom je Dr. Jürgen Abel a prvá verzia 1.0 bola vydaná 22.11.2002. Posledná publikovaná verzia je 2.4 vydaná 24.04.2003. Podľa autora je ABC rýchlejší ako gzip –9 a bzip2 a na Calgary Corpuse dosahuje kompresný pomer 2.238 bpB.

Používa implementáciu Burrows-Wheelerovej transformácie s veľkosťou 5 MB pre triediaci buffer. Pri súboroch, ktoré sú väčšie alebo rovné ako 256 KB používa program ABC nasledujúce poradie algoritmov:

1. **Burrows-Wheelerova transformácia**

2. **Run-length encoding EXP** popísaný v kapitole 5.4.
3. algoritmus **Inversion Frequencies**
4. **aritmetický kóder**

Pre súbory menšie ako 256 KB používa nasledujúce poradie algoritmov:

1. **Burrows-Wheelerova transformácia**
2. **Run-length encoding BIT0** popísaný v kapitole 5.5.
3. **Algoritmus WFC**, ktorý je alternatívou pre algoritmus Move-to-front
4. **Run-length encoding BIT1** popísaný v kapitole 5.5.
5. **aritmetický kóder**

7.3. Szip

Szip je algoritmus vyvíjaný Michaelom Schindlerom od roku 1997.

Používa alternatívu k Burrows-Wheelerovej transformácii, ktorá bola vyvinutá Schindlerom, ktorá je popísaná v odstavci 3.2.. Táto varianta operuje s maximálnou veľkosťou bloku 4,7 MB, na ktorom prevádza transformáciu reťazca. Prednastavená je hodnota 1,7 MB. S týmto väčším blokom vychádza lepší kompresný pomer pre veľké súbory.

Pri kódovaní používa nasledujúce poradie algoritmov:

1. **Alternatíva k Burrows-Wheelerovej transformácii** popísaná v odstavci 3.2. Používa prediktívny model, ktorý bol špeciálne vyvinutý pre tento algoritmus.
2. **Bytovo orientovaný aritmetický kóder**

8. Vyhodnotenie výsledkov

Varianta	Calgary korpus	XML_{cz} korpus
MTF	2,27	0,71
MTF-1	2,26	0,73
MTF-2	2,25	0,74
MTF=off	2,66	0,86
RLE-1	2,29	0,72
RLE-2	2,25	0,71
RLE-3	2,44	0,76
RLE=off	2,47	0,76
LZC	2,47	0,76
LZSS	2,65	0,96
LZ=off	2,25	0,71

Tabuľka 42 : Najlepší kompresný pomer pre danú variantu na Calgary a XML_{cz} korpuse.

Obsahom tejto diplomovej práce bolo skúmanie možnosti zapojenia slovníkových metód do druhej fázy Burrows-Wheelerovej transformácie nad abecedami slov, slabík a slov.

Ako sme sa mohli presviedčať pri všetkých meraniach a výsledkoch, zapojenie slovníkových metód do druhej fázy Burrows-Wheelerovej transformácie nám neprinieslo žiadne zlepšenie výsledkov, naopak takmer vždy sme dosiahli horšie kompresné pomery.

Jediné prípady, kedy boli slovníkové metódy lepšie, teda konkrétne algoritmus LZC nastal pri neprítomnosti algoritmu RLE a pri variante RLE-3. Nedá sa však povedať, že jeho lepší kompresný pomer bol dôsledkom algoritmu LZC samotného, ide o zhoršenie kompresných pomerov pre všetky varianty nepoužívajúce slovníkové metódy.

Zaujímavým zistením bolo, že varianty RLE nemajú na slovníkové metódy takmer žiadny účinok. Algoritmus Move-to-front síce zlepšujúci účinok na slovníkové metódy má, žiadna z uvedených variánt nezmenila kompresný pomer slovníkových metód. Možno práve nevhodnosť algoritmov, ktoré predchádzali slovníkovým metódam môže za ich nie veľmi vydarené kompresné pomery.

V podstate jedinou premennou, podľa ktorej sa riadil kompresný pomer u slovníkových metód bola veľkosť použitej abecedy. Algoritmus LZC dosahoval lepšie výsledky pre menšie abecedy a algoritmus LZSS naopak.

Dá sa teda konštatovať, že slovníkové metódy pri daných nastaveniach nemôžu dosahovať ani porovnateľné výsledky pre najlepšie kompresné pomery, ktoré sa dosahujú bez nich.

Ako vidno z tabuľky 38, najlepší kompresný pomer v programe XBW dostávame pre túto postupnosť algoritmov ako druhú fázu Burrows-Wheelerovej transformácie:

1. **MTF**
2. **RLE-2**
3. **LZ=off**

Ako sa ukázalo, ani použitie slabík neprinieslo veľkú výhodu. Dôvod môžeme hľadať vo veľkostiach súborov v oboch korpusoch. Súbory v Calgary korpuse sú príliš malé, slabiky sa na nich správajú podobne ako slová a súbory v XML_{cz} korpuse su príliš veľké, slabiky sa tam správajú opäť ako slová. Podľa Lánskeho [10], výhoda slabík vynikne pri súboroch strednej veľkosti.

Program	Calgary korpus	XML_{cz} korpus
ABC	2,23	1,00
bzip2	2,19	1,3
szip	2,13	0,9
XBW	2,25	0,71

Tabuľka 43 : Najlepšie kompresné pomery pre programy používajúce Burrows-Wheelerovu transformáciu nad Calgary a XML_{cz} korpusom.

Vzhľadom ku kompresným pomerom programov, ktoré používajú Burrows-Wheelerovu transformáciu je síce program XBW posledný, ale vynahrádza si to na veľkých XML súboroch na XML_{cz} korpuse, kde ma najlepší kompresný pomer. Môžeme za tým hľadať veľký blok, na ktorý sa aplikuje Burrows-Wheelerova transformácia.

8.1. Možnosti ďalšieho vývoja

Určite zaujímavou by bola implementácia alternatívy pre Burrows-Wheelerovu transformáciu a jej prediktívneho modelu. Pri algoritme Move-to-front by bolo vhodné implementovať

nejakú jeho alternatívu napríklad algoritmus Inversion Frequencies alebo Weighted Frequecny Count, ktoré dosahujú lepší kompresný pomer ako algoritmus Move-to-front.

Pri algoritme Run-length encoding bola zaujimavá myšlienka použitá pri RLE-BIT0, kde boli dĺžky dlhých behov ukladané do samostatného dátového prúdu, ktorý bol následne pridaný do reťazca po fáze Move-to-front. K zlepšeniu by mohlo viesť ukladanie dĺžok dlhých behov do statického Huffmanovho stromu alebo aritmetického kódovania.

9. Literatúra

[1] Ziv J., Lempel A. (1977): A Universal Algorithm for Sequential Data Compression. *IEEE transactions on Information Theory* Vol. IT-23, No. 3, pp. 337-343.

[2] Ziv J., Lempel A. (1978): Compression of Individual Sequences Via Variable-Rate Coding. *IEEE Transactions on Information Theory*, Vol. IT-24, No. 5, pp. 530-535

[3] Bell T.C., Cleary J.G. and Witten I.H. (1984) : Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, Vol. 32, No. 4, pp. 396-402

[4] Burrows M. and Wheeler D.J. (1994) : A Block-sorting Lossles Data Compression Algorithm. *Digital Systems Research Center,* Research Report 124

[5] JUERGEN ABEL (2003) : Improvements to the Burrows-Wheeler Compression Algorithm: After BWT Stages

[6] MANISCALCO M.A. (2000) : A Run Length Encoding Scheme For Block Sort Transformed Data. *Technical paper*, http://www.geocities.com/m99datacompression/papers/rle/rle.html.

[7] Bentley J. L., Sleator D. D., Tarjan R. E. and Wei V. K. (1986) : A Locally Adaptive Data Compression Scheme. *Communications of the ACM*, Vol. 29, No. 4

[8] Huffman D. A. (1952) : A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the I.R.E.*, pp. 1098-1102

[9] Cormack G. and Horspool N. (1987) : Data Compression using Dynamic Markov Modelling, *Computer Journal*, December, No. 30, pp. 6

[10] Lánsky J. (2005) : Slabiková komprese, *Faculty of Mathematics and Physics*, Charles University

[11] Schindler M. (1994) : A Fast Block-sorting Algorithm for lossless Data Compression, *Proceedings of the Conference on Data Compression*, pp. 469

[12] Albers S. (1995) : Improved randomized on-line algorithms for the list update problem. *Proceeding of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 412-419

[13] Arnavut Z. , Magliveras S. S. (1997): Blocking Sorting and Compression. *Proceeding of the Data Compression Conference*, pp. 181- 190

[14] Binder E. (2001) : Distance Coder. *Usenet group: comp. Compression*

[15] Deorowicz S. (2001) : Second step algorithms in the Burrows-Wheeler compression algorithm, *Software – Practice and Experience*, pp. 99 - 111

[16] Grinberg D., Rajagopalan S., Venkatesan R. and Wei V. K. (1995) : Splay Trees for Data Compression. *Symposium on Discrete Algorithms*, pp. 522 - 530

[17] Storer J.A. and Szymanski T.G. (1982) : Data Compression via Textual Substitution. *Journal of ACM*, Vol. 29, No. 4, pp. 928-951.