VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

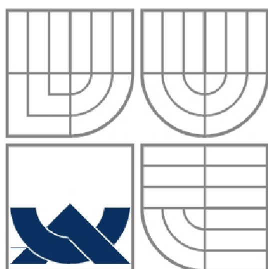
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# VYUŽITÍ BURROWS-WHEELEROVY TRANSFORMACE PRO KOMPRESI DAT

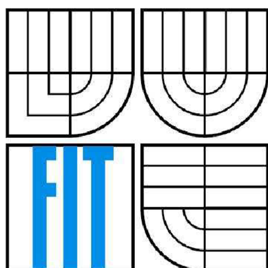
BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE KAREL SOKL
AUTHOR

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# VYUŽITÍ BURROWS-WHEELEROVY TRANSFORMACE PRO KOMPRESI DAT

UTILIZATION OF BURROWS-WHEELER TRANSFORM FOR DATA COMPRESSION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE KAREL SOKL
AUTHOR

VEDOUCÍ PRÁCE ING. VLASTIMIL KOŠAŘ
SUPERVISOR

BRNO 2012

# Abstrakt

Každý se může dostat do situace, kdy je nutné zmenšit velikost nějakého souboru. Pro tento účel existuje mnoho komprimačních algoritmů. Jak ale vybrat ten správný? V této práci je testována účinnost některých bezztrátových komprimačních algoritmů se zaměřením na Burrows-Wheelerovu transformaci. Výsledky jsou pak porovnány s programem bzip2.

# Abstract

Everyone can get into a situation when it is necessary to reduce the size of some file. For that purpose there are many compress algorithms. But how to choose the right one? In this work is tested efficiency of some of lossless compression algorithms focusing on Burrows-Wheeler transform. Then results are compared with the program bzip2.

# Klíčová slova

Burrows-Wheelerova transformace, BWT, komprese dat, Run-length kódování, RLE, Move-to-front transformace, MTF, Huffmanovo kódování, Lempel-Ziv-Welch algoritmus, LZW.

# Keywords

Burrows–Wheeler transform, BWT, data compression, Run-Length encoding, RLE, Move-to-front transform, MTF, Huffman encoding, Lempel-Ziv-Welch algorithm, LZW.

## Citace

Sokl Karel: *Využití Burrows-Wheelerovy transformace pro kompresi dat*, bakalářská práce, Brno, FIT VUT v Brně, 2012

# Využití Burrows-Wheelerovy transformace pro kompresi dat

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Vlastimila Košaře.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Karel Sokl
16. května 2012

## Poděkování

Děkuji svému vedoucímu Ing. Vlastimilovi Košařovi za poskytnuté rady a podporu během vytváření této bakalářské práce. Věnovat bych tuto práci chtěl všem vývojářům svobodného softwaru.

© Karel Sokl, 2012
*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů..*

# Obsah

Obsah ... 1
1 Úvod ... 3
2 Testované algoritmy ... 4
2.1 Burrows-Wheelerova transformace ... 4
2.1.1 Transformace BWT ... 4
2.1.2 Zpětná transformace BWT ... 5
2.2 Move-to-front transformace ... 7
2.2.1 Transformace MTF ... 8
2.2.2 Zpětná transformace MTF ... 8
2.3 Huffmanovo kódování ... 8
2.3.1 Komprimace pomocí HK ... 9
2.3.2 Dekomprimace pomocí HK ... 10
2.4 Run-length kódování ... 10
2.5 Lempel-Ziv-Welch komprese ... 11
2.5.1 Komprimace pomocí LZW ... 11
2.5.2 Dekomprimace pomocí LZW ... 12
2.6 Program bzip2 ... 12
3 Analýza problému a návrh jeho řešení ... 14
3.1 Požadavky na aplikace ... 14
3.2 Návrh ... 14
4 Popis implementace ... 15
4.1 BWT ... 15
4.2 MTF ... 15
4.3 Huffmanovo kódování ... 15
4.4 RLE ... 16
4.5 LZW ... 16
5 Návrh testů ... 17
5.1 Vybrané soubory ... 17
5.1.1 Formát TXT ... 17
5.1.2 Formáty zdrojových kódů ... 17
5.1.3 Uložené databáze ... 17
5.1.4 Multimédia ... 17
5.1.5 Ostatní formáty ... 18
5.2 Sestava testů ... 18

5.3 Očekávané výsledky ... 18
6 Výsledky testování ... 19
7 Závěr ... 21
7.1 Možná rozšíření ... 21
Příloha 1 ... 24
Výčet a popis testovaných souborů ... 24
Příloha 2 ... 25
Výsledky testů v číslech ... 25

# 1 Úvod

Samotná potřeba zhušťovat data sahá daleko před dobu, kdy byl vyroben první integrovaný obvod. Již v roce 1948 byly položeny základy teorie komprese dat vydáním práce *Matematická teorie komunikace* (A Mathematical Theory of Communication) [1], kterou zpracoval matematik a elektronik Claude Elwood Shannon spolu s matematikem Warrenem Weaverem. Později s příchodem prvních počítačů, jejichž paměť pro ukládání dat nebyla příliš velká, byla komprimace velmi důležitá a hojně používaná. Dnešní datová úložiště mají mnohem větší kapacitu a jsou cenově dostupnější než například před deseti lety. Mohlo by se tedy zdát, že potřeba nějakým způsobem zmenšovat objem dat již není aktuální. Není tomu tak. Už sice nemusíme řešit otázku kam data uložit, je ale stále potřeba informace předávat – od někud někam posílat. Obzvláště dnes, kdy je snaha přesouvat data i aplikace na nějaký server, přístupný přes internet, (tzv. *cloud computing*).

Data je možné komprimovat ztrátově (po kompresi a následné dekompresi ztratíme část informace) nebo bezztrátově (po dekompresi obdržíme informaci naprosto stejnou, jako před kompresí). Tato práce se bude zabývat především algoritmy pro bezztrátovou kompresi, konkrétně těmi, které jsou popsány v následující kapitole. Předmětem bude zkoumání, zjišťování, testování jejich vlastností. Především pak schopnosti – možnosti redukce velikosti určitých typů dat (kompresní poměr).

# 2 Testované algoritmy

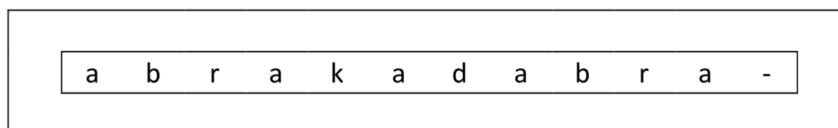
V této kapitole jsou popsány jednotlivé algoritmy, které jsou předmětem testování této práce. U každého z nich je vysvětlena podstata a princip transformace, popřípadě komprimace dat. Na závěr této kapitoly je také vysvětleno jak pracuje program *bzip2* [2], se kterým budeme dosažené výsledky testů porovnávat.

## 2.1 Burrows-Wheelerova transformace

Burrows-wheelerova transformace [3], dále jen BWT, je pomocná metoda, používaná při kompresi dat. Tento algoritmus vynalezli Michael Burrows a David Wheeler v roce 1994 v Kalifornii. Tato metoda vstupní data pouze uspořádává vhodným způsobem tak, že následná komprimace je pak s velkou pravděpodobností efektivnější než bez použití této transformace. BWT patří mezi algoritmy, které nepracují s proudem dat. Data naopak zpracovává po blocích. Metoda dosahuje lepších výsledků, pokud pracuje s bloky většími, ale během transformace se data musí třídit. Proto, pokud se bude pracovat s většími bloky dat, bude celá transformace trvat déle.

### 2.1.1 Transformace BWT

Během popisu transformace budou jednotlivé její kroky ukazovány na bloku dat, který je na obrázku 2.1.1. Jedná se o blok dat délky $N = 12$. Poslední znak „-“ reprezentuje konec souboru „EOF“. Tento znak má nejvyšší ordinální hodnotu (nejvyšší ze všech znaků, které se mohou objevit na vstupu).




**Obrázek 2.1.1** Ilustrační blok dat.

Blok, který se bude zpracovávat, se celý načte do paměti. Nejprve se vytvoří matice o velikosti $[N, N]$, kde v prvním řádku bude původní blok dat. V každém dalším řádku pak bude tento blok otočený o jeden znak doleva. Znaky, které na řádku „přetečou“ vlevo, se doplní zprava. Následně celou matici seřadíme podle obsahů jednotlivých řádků. Každý řádek je tvořen „řetězcem“ bajtů. Je tedy možné porovnávat ordinální hodnoty jednotlivých bajtů. Stav před a po seřazení je na obrázku 2.1.2.

| 0 | a | b | r | a | k | a | d | a | b | r | a | - |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | b | r | a | k | a | d | a | b | r | a | - | a |
| 2 | r | a | k | a | d | a | b | r | a | - | a | b |
| 3 | a | k | a | d | a | b | r | a | - | a | b | r |
| 4 | k | a | d | a | b | r | a | - | a | b | r | a |
| 5 | a | d | a | b | r | a | - | a | b | r | a | k |
| 6 | d | a | b | r | a | - | a | b | r | a | k | a |
| 7 | a | b | r | a | - | a | b | r | a | k | a | d |
| 8 | b | r | a | - | a | b | r | a | k | a | d | a |
| 9 | r | a | - | a | b | r | a | k | a | d | a | b |
| 10 | a | - | a | b | r | a | k | a | d | a | b | r |
| 11 | - | a | b | r | a | k | a | d | a | b | r | a |

| 0 | a | b | r | a | k | a | d | a | b | r | a | - |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | a | b | r | a | - | a | b | r | a | k | a | d |
| 9 | a | d | a | b | r | a | - | a | b | r | a | k |
| 3 | a | k | a | d | a | b | r | a | - | a | b | r |
| 8 | a | - | a | b | r | a | k | a | d | a | b | r |
| 2 | b | r | a | k | a | d | a | b | r | a | - | a |
| 7 | b | r | a | - | a | b | r | a | k | a | d | a |
| 1 | d | a | b | r | a | - | a | b | r | a | k | a |
| 6 | k | a | d | a | b | r | a | - | a | b | r | a |
| 10 | r | a | k | a | d | a | b | r | a | - | a | b |
| 4 | r | a | - | a | b | r | a | k | a | d | a | b |
| 11 | - | a | b | r | a | k | a | d | a | b | r | a |

**Obrázek 2.1.2** Vlevo: před seřazením, vpravo: po seřazení.

Také je nutné si uschovat informaci o tom, který řetězec se přesunul na který řádek. Výstupem celé transformace je pak poslední sloupec seřazené matice, dále pozice prvního a posledního (ukončovacího) znaku v tomto posledním sloupci, číslováno od nuly shora. Celý výstup je ukázán na obrázku 2.1.3.

| - | d | k | r | r | a | a | a | a | b | b | a | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Obrázek 2.1.3** Výstup. Červeně: pozice prvního a posledního znaku.

Pozice prvního a posledního znaku se ukládá proto, aby pozdější zpětná transformace byla rychlejší a hlavně paměťově méně náročnější. Po transformaci velkého bloku dat pak stejné po sobě jdoucí znaky vytváří dlouhé řady, což je vhodné pro metody typu Run-length kódování nebo Move-to-front transformace. Ve výsledku je transformovaný soubor o něco větší, ale jeho komprimační možnosti jsou o tolik lepší, že je tato skutečnost zanedbatelná.

## 2.1.2 Zpětná transformace BWT

Zpětnou transformaci je možné realizovat několika různými způsoby. V následujících podkapitolách budou předvedeny dva z nich.

#### 2.1.2.1 Rekonstrukce celé matice

Tento způsob spočívá v tom, že se postupně sestaví celá matice pro každý blok dat. Nejprve se načte transformovaný blok dat o velikosti $N$ a pozice prvního znaku. Načtený blok představuje

poslední sloupec matice. Jeho seřazením pak dostaneme první sloupec. Poslední sloupec se „nalepí“ před první sloupec matice a získá se tak část matice (2 sloupce). Dále se bude postupovat tak, že se vždy seřadí ta část matice, která je již vytvořena. A pak se před ni „nalepí“ poslední sloupec. Takto se bude „řadit a lepit“ dokud se nezíská celá matice o velikosti $[N, N]$. Tímto postupem se zpětně získá původní seřazená matice, která je vidět na obrázku 2.1.2 vpravo. Původní řetětec (blok dat) je pak umístěn na řádku, jehož index je roven pozici posledního znaku. Tento postup je nastíněn na obrázku 2.1.4.

| A | B | 1 | 1' | 2 | 2' | 3 | 3' | |
|---|---|---|---|---|---|---|---|---|
| - | a | - a | a b | - a b | a b r | - a b r | a b r a | |
| d | a | d a | a b | d a b | a b r | d a b r | a b r a | |
| k | a | k a | a d | k a d | a d a | k a d a | a d a b | |
| r | a | r a | a k | r a k | a k a | r a k a | a k a d | |
| r | a | r a | a - | r a - | a - a | r a - a | a - a b | |
| a | b | a b | b r | a b r | b r a | a b r a | b r a k | … |
| a | b | a b | b r | a b r | b r a | a b r a | b r a - | |
| a | d | a d | d a | a d a | d a b | a d a b | d a b r | |
| a | k | a k | k a | a k a | k a d | a k a d | k a d a | |
| b | r | b r | r a | b r a | r a k | b r a k | r a k a | |
| b | r | b r | r a | b r a | r a - | b r a - | r a - a | |
| a | - | a - | - a | a - a | - a b | a - a b | - a b r | |

**Obrázek 2.1.4** A je načtený blok dat, B je první sloupec.
1 … situace po "přilepení sloupce A". 1' … situace po seřazení.

#### 2.1.2.2 Rekonstrukce pomocí transformačního vektoru

Předchozí způsob rekonstrukce je jednoduchý, ale je velmi náročný na paměť. Pokud by se zpracovával blok například o velikosti 50 kB, bylo by potřeba k rekonstrukci 2,5 GB operační paměti. Proto se v praxi používá úspornější metoda, která nejprve nalezne transformační vektor $T[]$ a podle něj pak data přeuspořádá do správného tvaru. Aby bylo možné tento transformační vektor vytvořit, je nutné znát první a poslední sloupec původní matice. Poslední sloupec $L[]$ je vlastně načtený blok a první sloupec $F[]$ se získá seřazením sloupce posledního. Pak už jen stačí postupně procházet poslední sloupec $L[i]$, vždy nalézt odpovídající znak v prvním sloupci $F[j]$ ($L[i] == F[j]$) a do výsledného transformačního vektoru $T[]$ tuto pozici uložit ($T[i] = j$). Odpovídající znak v $F[j]$ se hledá vždy od zažátku sloupce a bere se první výskyt tohoto znaku. Každý znak se zpracuje pouze jednou. V případě více stejných znaků se tedy budou přeskakovat ty, které už se jednou „použily”. Vektor $T[]$ pro ilustrační příklad je vyobrazen na obrázku 2.1.5. Jakmile je získán vektor $T[]$, je možné rovnou zapsat zpětně transformovaná data. Data se začnou zapisovat z posledního sloupce

$L[]$ z pozice *index* ($L[index]$), na které je uložen první znak. Tento *index* byl načten zároveň s daty určenými ke zpětné transformaci. Po zapsání prvního znaku se přejde na *index2*, který je uložený v $T[index]$ ($index2 = T[index]$). Zapíše se druhý znak z $L[index2]$ a přejde se na další index uložený v $T[index2]$. Takto se postupně zapíšou celá původní data.

| T = | 5 | 6 | 7 | 8 | 11 | 9 | 10 | 1 | 2 | 3 | 4 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Obrázek 2.1.5** Transformační vektor *T*.

| *index* | *L*[] | *T*[] | Výstup |
|---|---|---|---|
| 0 | - | 5 | **a** |
| 1 | d | 6 | **b** |
| 2 | k | 7 | r |
| 3 | r | 8 | a |
| 4 | r | 11 | k |
| **5** | **a** | **9** | a |
| 6 | a | 10 | d |
| 7 | a | 1 | a |
| 8 | a | 2 | b |
| **9** | **b** | 3 | r |
| 10 | b | 4 | a |
| 11 | a | 0 | - |

**Obrázek 2.1.6** Červeně je vyznačen index prvního znaku, zeleně pak index druhého znaku.

# 2.2 Move-to-front transformace

Move-to-front transformace [4], dále jen MTF, je další metoda, která se používá pro uspořádání dat před jejich komprimací. Publikoval ji Boris Ryabko pod jejím původním názvem „book stack“ v roce 1980. Tento algoritmus data zpracovává proudově, proto není náročný na paměť. K transformaci používá tabulku všech symbolů, které se na vstupu mohou objevit. Tato tabulka se označuje jako vstupní abeceda. Nejčastěji se data zpracovávají po bajtech a tabulka je tak tvořena ASCII kódy jednotlivých znaků (od 0 až po 255). Výhoda této metody je, že posloupnosti stejných znaků nahrazuje posloupností nul. Obecně zvyšuje frekvenci výskytu znaků s nízkou ordinální (ASCII) hodnotou. Tato vlastnost je velmi vhodná pro další komprimaci, která je založená na statistice, například Huffmanovo kódování, které je popsáno dále.

### 2.2.1 Transformace MTF

Pro názornost zde bude ukázán postup na vzorku dat, který byl výstupem BWT v předešlé kapitole, obrázek 2.2.1.

| znaky: | - | d | k | r | r | a | a | a | a | b | b | a | 5 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ASCII: | 45 | 100 | 107 | 114 | 114 | 97 | 97 | 97 | 97 | 98 | 98 | 97 | 5 | 0 |

**Obrázek 2.2.1** Vstup pro MTF. Na druhém řádku ASCII hodnota znaků.
Červeně jsou vyznačena čísla ve své datové podobě, jak jsou uložena.

Nejprve se vytvoří tabulka znaků – vstupní abeceda. Pak se budou načítat jednotlivé znaky ze vstupu. Každý načtený znak se vyhledá v tabulce, na výstup se zapíše jeho aktuální index v tabulce (pozice od začátku tabulky) a poté se změní jeho pozice v tabulce – přesune se na začátek. Takto se pokračuje, dokud se nezpracuje celý vstup. Výstup pro ilustrační vstup je na obrázku 2.2.2.

| ASCII: | 45 | 100 | 107 | 114 | 0 | 100 | 0 | 0 | 0 | 101 | 0 | 1 | 11 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Obrázek 2.2.2** Výstup MTF v ASCII tvaru.

### 2.2.2 Zpětná transformace MTF

Zpětná transformace je prakticky stejná jako transformace samotná. S tím rozdílem, že na vstupu jsou indexy do tabulky místo znaků a na výstup se zapisují znaky místo indexů. Na začátku se vytvoří tabulka znaků (stejně jako na začátku transformace). Pak se načítají jednotlivé indexy ze vstupu. Znak, který se v tabulce nachází na aktuálně načteném indexu se zapíše na výstup. Poté se pozice tohoto znaku v tabulce změní – přesune se na začátek. Tabulka se takto postupně aktualizuje, není tedy nutné ji někam ukládat.

## 2.3 Huffmanovo kódování

Huffmanovo kódování [5, 6], dále jen HK, je jeden z bezztátových kompresních algoritmů, který využívá pravděpodobnosti výskytu jednotlivých znaků v datech. Byl vynalezen Davidem Albertem Huffmanem, který jej publikoval v roce 1952. Jeho princip zpočívá v tom, že nejprve zjistí četnost výskytu každého znaku na vstupu. Pak podle této četnosti každému znaku přidělí unikátní bitový kód (prefix). Těmito prefixy pak jednotlivé znaky nahradí. Výhoda je pak v tom, že znaky, které se v datech objevují nejčastěji, budou mít nejkratší prefix – například délky 3 bity. Takovýto

znak je pak zakódován namísto osmi bity pouze třemi. Nevýhoda pak může být skutečnost, že je nutné data projít dvakrát. V prvním průchodu zjistit četnost jednotlivých znaků a až při druhém průchodu data kódovat. Tato nevýhoda se dá řešit staticky (dopředu) vytvořenou tabulkou výskytů, která se pak bude používat pro různá data. Takováto tabulka ale musí být vytvořena speciálně prourčitý typ dat, aby byla komprese efektivní. Se statickou tabulkou bude efektivita ve velké většině případů horší, než s tabulkou dynamicky vytvořenou pro konkrétní data.

### 2.3.1 Komprimace pomocí HK

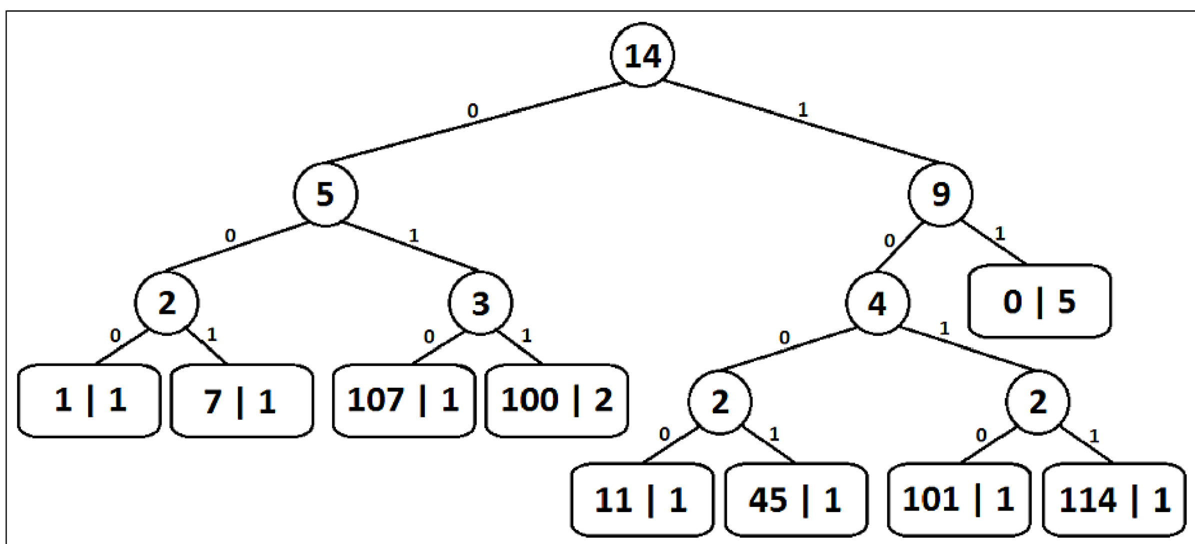
Nejprve je nutné zjistit četnost výskytů jednotlivých znaků ve vstupních datech. Jako příklad zde bude uvedena komprimace výstupu z předešlé metody MTF. Tento výstup je uveden na obrázku 2.2.2. Jakmile máme četnosti zjištěné (Obrázek 2.3.1) sestavíme Huffmanův strom následujícím způsobem.

| znaky: | 0 | 1 | 7 | 11 | 45 | 100 | 101 | 107 | 114 |
|---|---|---|---|---|---|---|---|---|---|
| četnosti: | 5 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 |

**Obrázek 2.3.1** Četnosti znaků na vstupu.

#### 2.3.1.1 Huffmanův strom

Tento binární strom se začíná vytvářet zespodu, od listů. Vždy se vezmou dva znaky s nejmenší četností a vytvoří se z nich uzel nový. Tento nový uzel pak bude mít hodnotu součtu četností znaků (popřípadě uzlů), ze kterých byl získán (vytvořen). Takto se bude postupovat, dokud se nezíská jediný uzel – kořen stromu. Vytvořený strom pro ilustrační příklad je na obrázku 2.3.2.




**Obrázek 2.3.2** Huffmanův strom. V levé části listů je hodnota znaku, v pravé pak jeho četnost.

Jakmile je strom vytvořený, přidělí se každému levému přechodu hodnota 0 a pravému hodnota 1. Kód znaku je pak tvořen váhou cesty od kořene až k listu tohoto znaku. Kódy znaků z příkladu jsou shrnuty na obrázku 2.3.3.

| znaky: | 0 | 1 | 7 | 11 | 45 | 100 | 101 | 107 | 114 |
|---|---|---|---|---|---|---|---|---|---|
| kódy: | 11 | 000 | 001 | 1000 | 1001 | 011 | 1010 | 010 | 1011 |

**Obrázek 2.3.3** Kódy znaků.

Při druhém průchodu souborem s daty se na výstup budou zapisovat kódy těchto znaků. Každý znak je obvykle uložen na jednom bajtu. Z délky jednotlivých kódů je vidět, že výsledná, zkomprimovaná data, budou zabírat méně místa, než před kompresí. Velikost dat z příkladu se podařilo zmenšit z původních 14ti bajtů na pouhých 6. Jak je vidět na obrázku 2.3.4, poslední bajt není zcela využit. Je tedy nutné do komprimovaného souboru také uložit počet platných bitů v posledním bajtu dat. Aby bylo možné zkomprimovaná data rekonstruovat, je také nutné uložit tabulku s jednotlivými kódy. To vše nám výslednou velikost zvyšuje. Je proto výhodnější, pokud se zpracovává větší množství dat v poměru k velikosti tabulky kódů.

| 1001 0110 | 1010 1111 | 0111 1111 | 1101 0110 | 0010 0000 | 1xxx xxxx |
|---|---|---|---|---|---|

**Obrázek 2.3.4** Bitový výstup Huffmanova kódování.

### 2.3.2 Dekomprimace pomocí HK

K dekomprimaci je potřeba znát tabulku, podle které se kódovalo. Ta je uložena zároveň s daty. Stačí tedy postupně načítat bit po bitu a hledat v tabulce prefix, který máme aktuálně načtený. Pokud jej v tabulce nenajdeme, načteme další bit a hledání opakujeme. Pokud jej najdeme, zapíšeme na výstup odpovídající znak. Takto pokračujeme dokud nezpracujeme celý vstup.

## 2.4 Run-length kódování

Run-length kódování [5, 7], dále jen RLE, je bezztrátový kompresní algoritmus, který nahrazuje posloupnosti stejných znaků znakem pouze jedním a počtem jeho opakování. Kompresní poměr této metody velmi závisí na charakteru dat. Pokud se ve vstupních datech posloupnosti stejných znaků nevyskytují, komprese může být i záporná.

Tento algoritmus je možné implementovat v několika verzích. Jednou z možností je nahrazovat každou, libovolně dlouhou, posloupnost dvojicí *délka posloupnosti* : *hodnota znaku*. Tato metoda ale není nejefektivnější, protože nahrazuje i posloupnosti délky 1 touto dvojicí. S velkou pravděpodobností tak bude výsledný soubor větší než před komprimací.

Jinou možností je nahrazovat pouze posloupnosti od určité délky. Zde však nastává problém jak poznat, zda načtený znak určuje počet opakování nebo představuje datový bajt. Jedna z možností jak tuto situaci vyřešit je vyčlenit jeden bit z bajtu, který udává délku posloupnosti. Pokud je pak tento bit nastavený na jedna, dané číslo uvádí počet opakujících se znaků a následovat jej bude jeden zástupný znak posloupnosti. Pokud je nastavený na nulu, bude toto číslo udávat počet neopakujících se znaků. Následovat jej pak budou jednotlivé neopakující se znaky. Příklad vstupu a výstupu pro tento typ RLE kódování je na obrázku 2.4.1. Jak je z tohoto obrázku vidět, velikost původní zprávy se zredukovala pouze o jeden bajt. Na třetím řádku tabulky na tomto obrázku je zeleně vyznačen bit, který určuje, zda bude následovat posloupnost různých znaků nebo jeden zástupný znak z opakující se posloupnosti.

| Vstup: | m | n | a | a | a | a | a | u |
|---|---|---|---|---|---|---|---|---|
| Výstup: | 2 | m | n | 133 (5+128) | a | 1 | u | |
| Bity: | 0000 0010 | | | 1000 0101 | | 0000 0001 | | |

**Obrázek 2.4.1** Vstup a výstup RLE. Červeně jsou vyznačena čísla ve své datové podobě, jak jsou uložena. Na třetím řádku je ukázána bitová podoba uložených počtů (délek bloků).

# 2.5 Lempel-Ziv-Welch komprese

Lempel-Ziv-Welch algoritmus [5, 8], dále jen LZW, je představitel bezztrátových komprimačních metod, využívajících slovník. Jedná se o vylepšenou verzi algoritmů LZ77 a LZ78. Algoritmus LZW vyvinuli Abraham Lempel, Jacob Ziv a Terry Welch a publikovali jej v roce 1984. Až do roku 2004 byl chráněný patentem. LZW data zpracovává prudově, ale během komprimace i dekomprimace si vytváří pomocnou tabulku – slovník a velikost této tabulky velmi rychle roste. Je proto vhodné data rozdělit na bloky menší a ty pak zpracovat jednotlivě.

## 2.5.1 Komprimace pomocí LZW

Nejprve je nutné si vytvořit slovník – tabulku. Do tohoto slovníku se vloží všechny znaky (řetězce délky jedna), které se mohou vyskytnout na vstupu. Poté se načte nejdelší možný řetězec, který již je uložen ve slovníku (z počátku to budou řetězce délky jedna). Na výstup se pak zapíše pozice řetězce ve slovníku. Dále se k tomuto řetězci přidá nový znak, načtený ze vstupu, a tento nový

řetězec se vloží na konec slovníku. Takto se budou ze vstupu načítat a zpracovávat řetězce dokud budou na vstupu nějaké znaky. Ukázka postupu této komprimace je na obrázku 2.5.1.

| Krok | Načtený řetězec | Následující znak | Přidáno do slovníku: | | Výstup | |
|---|---|---|---|---|---|---|
| | | | Index | Hodnota | Index | (Odpovídá) |
| 1 | a | b | 256 | ab | 97 | a |
| 2 | b | r | 257 | br | 98 | b |
| 3 | r | a | 258 | ra | 114 | r |
| 4 | a | k | 259 | ak | 97 | a |
| 5 | k | a | 260 | ka | 107 | k |
| 6 | a | d | 261 | ad | 97 | a |
| 7 | d | a | 262 | da | 100 | d |
| 8 | ab | r | 263 | abr | 256 | ab |
| 9 | ra | EOF | | | 258 | ra |

**Obrázek 2.5.1** Průběh komprimace metodou LZW. Vstup je na obrázku 2.1.1. Výstup tvoří jednotlivé indexy zobrazeny ve sloupci *Výstup (Index)*.

### 2.5.2 Dekomprimace pomocí LZW

Při dekompresi se postupuje podobně jako u komprese. Nejprve se vytvoří slovník, do nějž se vloží všechny znaky, které může obsahovat původní soubor. Pak se načítají ze vstupu indexy do tabulky a na výstup se zapisují odpovídající znaky, popřípadě řetězce. Slovník je nutné průběžně aktualizovat. Nový záznam se vytvoří konkatenací (spojením) aktuálně zapisovaného řetězce na výstup a prvního znaku z následujícího řetězce (ten je nutné dopředu zjistit).

## 2.6 Program bzip2

Autorem tohoto komprimačního programu je Julian Seward. Prvotní verzi programu autor vydal v roce 1996. Bzip2 je šířen jako open-source (BSD licence) a není zatížen patenty. Je tedy možné ho nejen bezplatně používat, ale také je k dospozici jeho zdrojový kód. Tento kód je možné si upravovat, popřípadě z něj vytvořit knihovnu, kterou pak lze přidat do nějakého vlastního projektu. Bzip2 totiž nabízí programátorské aplikační rozhraní, přes které je možné přímo číst nebo zapisovat soubory ve formátu bzip2. Program je dostupný na oficiálních stránkách autora [2].

Pro komprimaci bzip2 nejprve použije Burrows-Wheelerovu transformaci, dále aplikuje transformaci Move-to-front a nakonec data zakóduje pomocí Huffamnova kódování. Dříve pro samotnou kompresi bylo použito aritmetické kódování, ale to je dnes chráněno patentem a proto bylo nahrazeno Huffmanovým kódem. Zkomprimovaný soubor pomocí bzip2 také obsahuje kontrolní

součet CRC a podporuje kontrolu a opravu poškozených archívů. Podporuje 32 i 64 bitové systémy. Původně byl program psán pro UNIX, ale je bez problému přenositelný na windows nebo Mac OS. Obyčejně se s programem pracuje v konzoli, existují ale také různé grafické nádstavby, například *Wbzip2* pro windows nebo *7-Zip*.

Bzip2 obecně dosahuje lepšího kompresního poměru než například algoritmus *deflate*, který ke komprimaci využívá algoritmy LZ77 (předchůdce LZW) a Huffmanovo kódování. *Deflate* je použit například v obrazových formátech GIF, PNG a v programech *zip*, *gzip*.

Sám o sobě je bzip2 poměrně pomalý, ale existují vícevláknové implementace, které tento nedostatek zmírňují. Například *pbzip2* (parallel bzip2) nebo výše zmíněný *7-Zip* umí pracovat s vlákny a podporuje vícejádrové procesory. Bzip2 samotný také dokáže zpracovat pouze jeden soubor najednou. Pokud je tedy potřeba zkomprimovat více souborů dohromady, popřípadě celou strukturu nějakého adresáře, je nutné tyto soubory nejdříve spojit do jednoho. V systému typu UNIX se k tomuto účelu používá například program *tar*.

# 3 Analýza problému a návrh jeho řešení

Existuje mnoho příležitostí, kdy je velmi vhodné nebo i nutné nějaký komprimační algoritmus použít. Ať už z důvodu malé šířky komunikačního přenosového pásma, nebo dokonce kvůli nízkému limitu pro přenos dat po komunikačním kanále, nebo pro pouhé ušetření místa na datovém úložišti. V každém případě se mohou požadavky na vlastnosti komprimačního nástroje lišit. Pokud bude někdo potřebovat data dlouhodobě ukládat, pravděpodobně mu bude nejvíce záležet na samotném kompresním poměru a čas potřebný pro zkomprimování již tak důležitý nebude. Naopak v oblasti komunikace, obzvláště pokud má tato komunikace probíhat v reálném čase, je nezbytné, aby komprese byla rychlá. Každý, kdo bude psát nějakou aplikaci nebo protokol, popřípadě si jen bude chtít zálohovat obsah své databáze, bude jistě hledat takový komprimační algoritmus, který by jeho požadavkům vyhověl nejlépe. Otázkou pak tedy zůstává podle čeho si zvolit ten správný komprimační nástroj. Velmi praktické by například bylo mít k dispozici výsledky účelných testů, které by na konkrétních příkladech přehledně demonstrovaly vlastnosti vybraných kompresních algoritmů. Aby byly takovéto výsledky užitečné pro co nejvíce zájemců, měly by testy být provedeny na velkém množství různých datových typů souborů.

Aby bylo možné jednotlivé algoritmy mezi sebou porovnávat, bude nezbytné mít k dispozici jejich implementaci. Dále pro větší volnost a možnosti následného testování bude vhodné aby pro každý algoritmus byl vytvořen program zvlášť. Bude pak možné je za sebe libovolně řetězit.

## 3.1 Požadavky na aplikace

Všechny tyto programy (aplikace) by měly správně implementovat daný algoritmus. Měly by zvládnout zpracovat vstup ze standardního vstupu nebo ze zvoleného souboru. Jako výstup pak bude postačovat transformovaný (zkomprimovaný) soubor, který bude zapsán na standardní výstup nebo do zvoleného souboru.

## 3.2 Návrh

Každý algoritmus bude implementovaný jako modul v jazyce C++, který se bude dát přeložit a použít jako knihovna. Tento modul bude obsahovat třídu, která bude poskytovat veřejné metody pro nastavení vstupu/výstupu, také pro samotnou transformaci (komprimaci) a zpětnou transformaci (dekomprimaci), popřípadě pro nastavení jiných parametrů – například velikost bloku u BWT. Pro účely testování pak bude pro každý modul vytvořen program, který tento modul „oživí“.

# 4 Popis implementace

Pro implementaci byl zvolen jazyk C++. Každý algoritmus byl realizován jako třída. Rozhraní této třídy umožňuje použití daného algoritmu. U každého z nich je pak možné nastavit vstupní soubor, který obsahuje data pro zpracování, a výstupní soubor, kam se bude ukládat výsledek. Pokud tyto soubory nastaveny nejsou, automaticky se načítá ze standardního vstupu a ukládá na standardní výstup. U BWT a LZW, které pracují blokově, je také možné nastavit velikost tohoto bloku. Dále jsou v této kapitole popsány některé implementační detaily jednotlivých algoritmů.

## 4.1 BWT

Tato transformace pracuje blokově. Velikost tohoto bloku je implicitně nastavena na 512 kB, ale je možné ji změnit. Při volbě této velikosti je dobré mít na paměti, že během transformace se tento blok musí seřadit. Zvolíme-li tedy blok větší, bude čas potřebný pro zpracování souboru delší.

Řadící algoritmus byl zvolen QuickSort s průměrnou časovou složitostí $O(n * \log(n))$, jehož optimalizovaná implementace je k dispozici ve standardní knihovně jazyka C, na základě porovnání uvedených v [9].

Paměťová náročnost této metody je závislá pouze na velikosti zpracovávaného bloku. Tento blok zabírá dvojnásobek své velikosti – pro řazení je použito pomocné indexové pole.

Samotné jádro transformace je sepsáno podle implementace uvedené v [3].

## 4.2 MTF

MTF data zpracovává znak po znaku. Ke své práci využívá tabulku všech znaků, které se mohou objevit na vstupu. Časově transformaci nejvíce prodlužuje, při přesouvání prvku na začátek tabulky, nutnost posouvat všechny předcházející prvky o jedno místo doprava. Tato tabulka se k výslednému souboru ukládat nemusí.

Implementace této transformce byla sepsána podle popisu v [4].

## 4.3 Huffmanovo kódování

Toto kódování zpracovává celý vstup najednou ve dvou průchodech. Během prvního průchodu se zjišťují četnosti jednotlivých znaků a pokud se načítá ze standardního vstupu, je také vytvořen dočasný soubor, do kterého se celý vstup uloží pro pozdější zpracování. Z těchto získaných četností se pak vytvoří Huffmanův kód platný pro celý vstup. Ve druhém průchodu se pak jednotlivé znaky zakódují a zapíší na výstup.

Samotná operace vytvoření bitového kódu je časově náročná, proto má toto řešení výhodu tu, že se tento kód vytvoří pouze jednou. Navíc je nutné k výslednému souboru tuto tabulku přiložit, aby bylo možné původní soubor znovu zrekonstruovat. A pokud je tato tabulka pouze jedna, jejím přiložením k výslednému souboru se jeho velikost příliš nezvětší. Nevýhoda je pak ta, že se musí daný soubor projít dvakrát.

Implementace vytvoření huffmanova stromu byla sepsána podle kódu uvedeného v [10].

## 4.4 RLE

RLE je implementováno konečným automatem. Vstupní data zpracovává znak po znaku a průběžně si načtené znaky ukládá do pomocného pole. Na výstup pak zapíše vždy délku posloupnosti znaků a v případě, že se jedná o neopakující se posloupnost, vypíše tyto znaky. V případě, že se jedná o posloupnost stejných znaků, vypíše jeden zástupný znak. Pro rozlišení, zda uložená délka označuje počet opakujících se nebo neopakujících se znaků, se nastavuje nejvýznamější bit na $1$ nebo $0$. Tato délka je uložena jako typ *short*, který má obvykle velikost 2B. Tímto je počet znaků v posloupnosti omezen na velikost $2^{16-1}$ bajtů ($32\ 768$ B). A svým způsobem RLE tedy pracuje blokově.

## 4.5 LZW

Tato metoda si během svého zpracování vytváří slovník řetězců, které se na vstupu již vyskytly. Tyto řetězce pak nahrazuje pozicí (indexem) v tomto slovníku. Velikost tohoto slovníku velice rychle roste – při uvážení, že na vstupu se může vyskytnout 256 různých znaků, pak jen dvojic, vytvořených z těchto znaků, je $256^2$ ($65\ 535$). V paměti pak takovýto slovník zabere více jak stovky megabajtů. Z tohoto důvodu se vstupní soubor zpracovává po blocích a slovník se vytváří pro každý blok nový. Tento slovník se do výstupního souboru ukládat nemusí, proto nevadí, když bude pro každý blok nový. Výhoda tohoto řešení je ta, že pro zpracování není potřeba tolik operační paměti a že doba zpracování je kratší (slovník se musí prohledávat). Nevýhodou je pak menší výsledný kompresní poměr.

Jádro této metody bylo sepsáno podle implementace uvedené na [11].

# 5 Návrh testů

Aby byly výsledky testů co nejvíce objektivní vzhledem k typům souborů, které jsou často předmětem komprimace, je nutné pečlivě vybrat vzorové soubory, na kterých se budou testy provádět. Ač se to nemusí tak jevit, není jednoduché vybrat jediný soubor každého formátu, který by tento souborový typ plně zastoupil.

Projektů, zabývajících se výběrem vhodných souborů pro testování komprimačních algoritmů, je mnoho. Například na těchto stránkách [12, 13] jsou přímo nabídnuty soubory, vybrané a určené přímo pro testování komprimačních algoritmů. Jsou rozvážně vybrány tak, aby obsahem dostatečně pokryly oblast nejčastěji v praxi používaných souborů. Z těchto zdrojů byly vybrány některé soubory pro účely testování, k této práci jsou přiloženy na CD a stručně popsány jsou v papírové příloze 1.

## 5.1 Vybrané soubory

Byli vybráni zástupci různých obsahů a datových formátů souborů.

### 5.1.1 Formát TXT

Tento textový formát byl vybrán proto, že svůj obsah ukládá tak, jak je. Testovaných souborů s touto příponou je více, konkrétně 5. Většina z nich obsahuje anglický text různého žánru, například pohádka, divadelní hra, texhnický text nebo poezii. Jeden z nich pak obsahuje prvních milion cifer čísla *pi*.

### 5.1.2 Formáty zdrojových kódů

Zdrojové texty pro různé typy programovacích jazyků se vzájemně liší. Byl proto vybrán zástupce strukturovaných jazyků (kód jazyka C), funkcionálních jazyků (kód jazyka LISP) a značkovacích jazyků (kód v HTML).

### 5.1.3 Uložené databáze

Databáze, popřípadě struktury nebo data tříd je možné ukládat do různých formátů. Z textových byly vybrány formáty *json* a *xml*.

### 5.1.4 Multimédia

Zvukové a obrazové soubory se většinou kódují pomocí ztrátových kompresních algoritmů, které jsou určené přímo pro daný formát. Existují ale odvětví, ve kterých je potřebné obrazová nebo

zvuková data komprimovat a ukládat beze ztráty jakékoli informace (například medicínská data). Do testů byl proto také zahrnut obrazový formát *bmp* a zvukový formát *wav*.

### 5.1.5 Ostatní formáty

Pro úplnost bude do testů zahrnut tabulkový soubor programu Microsoft Excel, přeložený a spustitelný program a také soubor obsahující přes 350 tisíc abecedně seřazených anglických slov.

## 5.2 Sestava testů

Průběh testování bude nejvíce zaměřen na Burrows-Wheelerovu transformaci, protože primárním účelem této práce je zjistit jaký vliv má tato metota na výsledek komprimace. Zda je vhodné ji použít, popřípadě v jaké kombinaci s jinými algoritmy se vyplatí ji použít. Během testování budou zkoušeny kombinace různé a to ve tvaru *transformace -> komprimace*. Otestovány budou také komprimační algoritmy samotné (bez předchozí transformace).

## 5.3 Očekávané výsledky

Dle obecných popisů mají slovníkové algoritmy většinou menší kompresní poměr než metody založené na statistice. Dalo by se tedy očekávat, že při použití Huffmanova kódování by výsledný soubor měl mít menší velikost než v případě použití metody LZW.

Program bzip2 data nejprve transformuje metodami BWT a MTF. Nakonec pak použije Huffmanovo kódování. Je tedy pravděpodobné, že výsledný soubor, zkomprimovaný tímto programem, a tentýž soubor, zkomprimovaný stejnými metodami (naimplementovanými pro tuto práci), budou mít přibližně stejnou velikost.

# 6 Výsledky testování

Pro jednoduchost zde budou použity zkratky, jejichž význam je uveden na obrázku 7.1. Výčet všech testovaných kombinací a jejich výsledný průměrný komprimační poměr je na obrázku 7.2.

Jak je vidět, nejlepšího kompresního poměru dosáhl program bzip2. Na druhém místě se pak umístil kompresní poměr testované kombinace BWT-MTF-HUFF. Ten se od prvního zmíněného liší téměř o sedm procent, což je relativně hodně vzhledem k tomu, že se jedná o stejnou posloupnost použitých algoritmů. Bzip2 pravděpodobně implementuje nějaké další heuristiky, které celkový komprimační poměr zlepšují. Celkem obstojně se pak ješte umístila kombinace BWT-RLE-HUFF s poměrem 48,1%. U všech ostatních možností se ukázalo, že jsou pro běžnou kompresi nevhodné. U některých byl výsledný soubor dokonce větší než původní před zpracováním. Nejhůře pak dopadla metoda LZW.

Průběh celého testování a konkrétní hodnoty jsou ukázány tabulkou v papírové příloze 2. Celá tabulka je rozdělena na dvě části, které na sebe navazují. Pro lepší orientaci je ve druhé části znovu uveden sloupec s názvy testovaných souborů. Jsou tam ukázány původní velikosti těchto souborů a pak dále jejich velikosti po použití daného algoritmu, popřípadě kombinaci algoritmů. V každém řádku je tučně a zeleně vyznačena metoda, která daný soubor zkomprimovala nejlépe. Tučně a červeně je pak vyznačena metoda nejhorší pro daný soubor. Kromě jednoho případu si nejlépe vedl program bzip2. Ale je zajímavé se podívat také na ostatní metody, které si v mnoha případech vedly také velmi dobře. Výjimku pak tvoří kombinace algoritmů MTF-LZW, která (kromě dvou případů) velikost souboru vždy zvětšila. Velmi překvapůjící je také skutečnost, že téměř v žádném případě se nepodařilo zkomprimovat soubor typu wav. Ba naopak jeho velikost většinou vzrostla téměř trojnásobně.

| Zkratka | Význam |
|---|---|
| BWT | Burrows-Wheelerova transformace |
| MTF | Move-to-front transformace |
| HUFF | Huffmanovo kódování |
| RLE | Run-length kódování |
| LZW | Lempel-Ziv-Welch kódování |
| BWT-MTF-HUFF | Použití těchto metod na jeden soubor v pořadí zleva doprava. |

**Obrázek 7.1** Význam použitých zkratek.

| Testované kombinace algoritmů | Kompresní poměr |
| --- | --- |
| HUFF | 62,3% |
| LZW | 121,2% |
| RLE | 99,9% |
| BWT-HUFF | 62,6% |
| BWT-LZW | 105,7% |
| BWT-RLE | 65,7% |
| MTF-HUFF | 62,8% |
| MTF-LZW | **161,5%** |
| MTF-RLE | 101,7% |
| BWT-MTF-HUFF | 38,7% |
| BWT-MTF-LZW | 99,5% |
| BWT-RLE-HUFF | 48,1% |
| BWT-RLE-LZW | 116,2% |
| bzip2 | **31,8%** |

**Obrázek 7.2** Kompresní poměry testovaných kombinací algoritmů.

# 7 Závěr

Doba vzniku všech těchto algoritmů sahá do relativně daleké minulosti, a tak se popis jejich implementace objevuje v mnoha různých publikacích ve formě pseudokódu nebo přímo v podobě kódu nějakého programovacího jazyka. Ve většině případů tedy nebylo nutné znovu vymýšlet celou implementaci a postačovalo tu stávající přepsat do objektové podoby do C++.

Z výsledků je také patrné, že při použití Burrows-Wheelerovy transformace se vždy dosáhlo mnohem lepšího výsledku než bez jejího použití. Toto zlepšení v průměru činí 24,6%. BWT má velký potenciál a v oblasti komprimace je to nepostradatelný algoritmus.

Celkově testy dopadly, až na dvě výjimky, podle očekávání. Velmi překvapivých výsledků dosahovala metoda LZW, což je snad způsobeno málo optimalizovanou implementací. Také je zajímavé, že testovaný soubor typu *wav* se pomocí většiny metod nedal zkomprimovat.

## 7.1 Možná rozšíření

Do této práce nebyl zahrnut algoritmus aritmetického kódování, který obecně dosahuje lepších komprimačních výsledků než Huffmanovo kódování, protože je zatížený patentem. Výsledky této práce jsou směřovány spíše soukromníkům a zájemcům o svobodný software. Z čistě akademických důvodů by ale bylo zajímavé do testů tento agloritmus zahrnout také.

V této práci byl převážně kladen důraz na kompresní poměr jednotlivých metod. Bylo by vhodné optimalizovat implementaci stávajících algoritmů a porovnat v této sadě testů také čas, který je potřebný pro zpracování jednotlivých souborů.

Také implementovaná metoda LZW dosahovala v testech příliš špatných výsledků, což je velmi zvláštní. Bylo by tedy dobré její implementaci vylepšit a otestovat ji znovu.

# Literatura


[1] WWW stránka: *Theory of Data Compression* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://www.data-compression.com/theory.shtml>

[2] SEWARD, Julian: *Bzip2* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://www.bzip.org>

[3] NELSON, Mark: *Dr. Dobb's Journal: Data compression with the Burrows-Wheeler Transform* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://marknelson.us/1996/09/01/bwt/>

[4] CAMPOS, Arturo: *Move to front* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://www.arturocampos.com/ac_mtf.html>

[5] WXXXWW stránka: *Theory of Data Compression* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://www.data-compression.com/theory.shtml>

[6] HORDĚJČUK, Vojtěch: *Huffmanovo kódování* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://voho.cz/wiki/informatika/kodovani/huffman/>

[7] WWW stránka: *Run-Length Encoding (RLE)* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://www.fileformat.info/mirror/egff/ch09_03.htm>

[8] HORDĚJČUK, Vojtěch: *Kompresní algoritmus LZW* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://voho.cz/wiki/informatika/algoritmus/komprese/lzw/>

[9] ALLAIN, Alex: *Sorting algorithm comparsion* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html>

[10] MOL, Mike: *Rosetta code: LZW compression* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://rosettacode.org/wiki/Huffman_coding#C.2B.2B>

[11] MOL, Mike: *Rosetta code: Huffman coding* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://rosettacode.org/wiki/LZW_compression#C.2B.2B>

[12] HORLOR, J., Arnold, R., Powell, M., Bell, T.: *The Cantebury Corpus* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://corpus.canterbury.ac.nz>

[13] WWW stránka: *Maximum compression* [online]. [cit. 2012-05-15]. Dostupné na URL: <http://www.maximumcompression.com>

# Seznam příloh

Příloha 1. Výčet a popis testovaných souborů
Příloha 2. Výsledky testů v číslech
Příloha 3. DVD obsahující implementaci komprimačních metod a sadu testů.

# Příloha 1

## Výčet a popis testovaných souborů

| Název souboru | Popis obsahu souboru | Původ souboru |
|---|---|---|
| alice29.txt | Kniha v anglickém jazyce, pohádka. | The Canterbury corpus [11] |
| asyoulik.txt | Kniha v anglickém jazyce, divadelní hra. | The Canterbury corpus [11] |
| bmp.in | Obrázek ve formátu *bmp*. | Vytvořeno autorem této práce |
| cp.html | Zdrojový kód jazyka *HTML*. | The Canterbury corpus [11] |
| english.dic | Přes 350 tisíc seřazených anglických slovíček. | Maximum compression [12] |
| fields.c | Zdrojový kód jazyka *C*. | The Canterbury corpus [11] |
| grammar.lsp | Zdrojový kód jazyka *LISP*. | The Canterbury corpus [11] |
| json.in | Databáze uložená do formátu *json.* | Vybráno autorem z různých zdrojů |
| kennedy.xml | Tabulkový soubor programu Microsoft Excel. | The Canterbury corpus [11] |
| lcet10.txt | Technický text v angličtině. | The Canterbury corpus [11] |
| pi.txt | Prvních milion cifer čísla *pi.* | The Canterbury corpus [11] |
| plrabn12.txt | Poezie v angličtině. | The Canterbury corpus [11] |
| sum | Přeložený spustitelný program. | The Canterbury corpus [11] |
| wav.in | Zvukový soubor ve formátu *wav*. | Vybráno autorem z různých zdrojů |
| xml.in | Databáze uložená do formátu *xml.* | Vybráno autorem z různých zdrojů |

**Příloha 1** Seznam souborů, které byly použity v testech.

# Příloha 2

## Výsledky testů v číslech

| Soubor | Původní velikost (kB) | Velikost v kB po aplikaci následujících kombinací algoritmů | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | HUFF | LZW | RLE | BWT-HUFF | BWT-LZW | BWT-RLE | MTF-HUFF |
| alice29.txt | 152,0 | 88,0 | 164,8 | 150,4 | 88,0 | 148,1 | 116,3 | 97,3 |
| asyoulik.txt | 125,2 | 76,0 | 140,3 | 125,2 | 76,1 | 132,1 | 100,9 | 82,9 |
| bmp.in | 21 233,7 | 20 757,4 | **46 134,3** | 20 971,3 | 20 757,9 | 34 440,2 | 21 060,4 | 16 606,8 |
| cp.html | 24,6 | 16,5 | 30,0 | 24,9 | 16,5 | 28,5 | 16,3 | 17,5 |
| english.dic | 4 067,4 | 2 212,7 | 3 583,5 | 4 067,6 | 2 213,0 | 3 741,1 | 3 058,6 | 2 010,5 |
| fields.c | 11,2 | 7,3 | 14,2 | 11,4 | 7,4 | 13,1 | 7,1 | 7,9 |
| grammar.lsp | 3,7 | 2,4 | 5,6 | 3,8 | 2,5 | 5,5 | 2,7 | 2,7 |
| json.in | 26 957, 2 | 13 829, 0 | 8 311,2 | 26 452,8 | 13 830,7 | 4 270,9 | 2 557,3 | 14 360,4 |
| kennedy.xls | 1 029,7 | 463, 6 | 707,0 | **1 035,5** | 463,6 | 281,9 | 261,4 | 386,0 |
| lcet10.txt | 426,8 | 250,9 | 459,2 | 416,3 | 250,9 | 365,348 | 290,9 | 271,4 |
| pi.txt | 1 000,0 | **424,9** | 1 155,8 | 1 016,60 | 437,5 | 1 156,0 | 1 016,8 | 437,4 |
| plrabn12.txt | 481,9 | 275,9 | 538,5 | 481,2 | 275,9 | 472,8 | 383,3 | 302,1 |
| sum | 38,2 | 26,6 | 50,1 | 37,4 | 26,6 | 45,1 | 24,6 | 25,7 |
| wav.in | 47 457,8 | 46 365,0 | 135 851 | 47 183,2 | 46 366,0 | 131 385,3 | 47 092,7 | 47 299,7 |
| xml.in | 8 361,0 | 3 659,8 | 3 354,5 | 8 361,3 | 3 660,3 | 826,9 | 307,9 | 3 534,3 |

**Příloha 2** Výsledky testů, část první.

| Soubor | Velikost v kB po aplikaci následujících kombinací algoritmů | | | | | | |
|---|---|---|---|---|---|---|---|
| | MTF-LZW | MTF-RLE | BWT-MTF-HUFF | BWT-MTF-LZW | BWT-RLE-HUFF | BWT-RLE-LZW | bzip2 |
| alice29.txt | **266,4** | 152,6 | 49,6 | 131,4 | 77,3 | 171,0 | **43,3** |
| asyoulik.txt | **226,6** | 127,3 | 45,2 | 117,8 | 67,8 | 154,3 | **39,6** |
| bmp.in | 45 600,7 | 21 896,2 | 13 417,9 | 35 733,7 | 19 892,8 | 38 777,6 | **11 888,9** |
| cp.html | **50,7** | 25,1 | 9,0 | 25,8 | 11,9 | 32,3 | **7,6** |
| english.dic | **4 409,1** | 4 206,8 | 1 400,1 | 3 526,00 | 1 894,9 | 4 256,8 | **1 221,7** |
| fields.c | **23,5** | 11,9 | 3,7 | 11,2 | 5,3 | 15,0 | **3,0** |
| grammar.lsp | **8,7** | 3,9 | 1,6 | 5,0 | 2,2 | 6,4 | **1,2** |
| json.in | 18 258,1 | **27 460,9** | 4 151,0 | 3 593,2 | 1 354,9 | 3 091,9 | **960,7** |
| kennedy.xls | 378,8 | 1 032,0 | 281,4 | 282,3 | 229,7 | 259,4 | **130,2** |
| lcet10.txt | **730,3** | 421,8 | 129,0 | 328,3 | 192,9 | 416,3 | **107,7** |
| pi.txt | **1 156,3** | 1 016,80 | 437,4 | 1 156,0 | 465,6 | 1 220,2 | 431,7 |
| plrabn12.txt | **828,7** | 487,0 | 169,9 | 433,1 | 252,1 | 544,7 | **145,6** |
| sum | **64,0** | 38,3 | 15,2 | 41,6 | 18,9 | 49,7 | **12,9** |
| wav.in | **137 318,0** | 47 209,5 | 45 472,1 | 131 883,1 | 46 106,0 | 131 712,5 | **43 411,6** |
| xml.in | 6 015,3 | **8 405,3** | 1 105,5 | 579,7 | 197,9 | 431,9 | **105,0** |

**Příloha 2** Výsledky testů, část druhá.