UNIVERZITA PARDUBICE

Fakulta elektrotechniky a informatiky

Bezztrátová komprimace dat

Josef Haken

Bakalářská práce

2008

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Katedra informačních technologií
Akademický rok: **2007/2008**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Josef HAKEN**

Studijní program: **B2646 Informační technologie**

Studijní obor: **Informační technologie**

Název tématu: **Bezztrátová komprimace dat**

Zásady pro vypracování:

V teoretické části bakalářské práce budou zhodnoceny současné principy komprimačních algoritmů, jejich omezení, vývoj a využití v praxi.
V implementační části bude naprogramováno komprimování a dekomprimování dat většinou dostupných algoritmů a ty pak budou vyhodnoceny dle časové náročnosti a komprimačního poměru v porovnání s komerčními komprimačními programy.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

Jan Čapek, Peter Fabian: Komprimace dat - principy a praxe, Computer press, 2000
http://www.pcsvet.cz/art/article.php?id=4603 - Komprimace dat – PC Svět

Vedoucí bakalářské práce: **Ing. Zbyněk Kopecký**
Ústav elektrotechniky a informatiky

Datum zadání bakalářské práce: **30. listopadu 2007**

Termín odevzdání bakalářské práce: **16. května 2008**

Univerzita Pardubice Česká republika

L.S.

doc. Ing. Simeon Karamazov, Dr.
děkan

V Pardubicích dne 29. dubna 2008

SOUHRN

Práce se zabývá bezztrátovou komprimací dat. Popisuje základní principy a algoritmy používané ke komprimaci dat. V závěru práce jsou popsané algoritmy vyhodnoceny na komprimační poměr, časovou náročnost a zároveň srovnány s komerčními programy používaných pro bezztrátovou komprimaci dat.

KLÍČOVÁ SLOVA

bezztrátová komprimace, komprese, algoritmy, programování, C++, komprimační programy

TITLE

Lossless data compression

ABSTRACT

The bachelor work deal with lossless data compression. It describes basic principles and algorithms used for data compression. In the final part of the work are algorithms evaluate to compression ratio, time heftiness and also compared with commercial programs used for lossless data compression.

KEYWORDS

lossless compression, compression, algorithm, programming, C++, compression programs

Obsah

Seznam obrázků

Seznam tabulek

Seznam zkratek

ACC (Advanced Audio Coding) – ztrátový zvukový kodek. Byl vyvinut jako logický následovník formátu MP3.

bit − z anglického binary digit , základní a současně nejmenší jednotka informace.

BMP (Microsoft Windows Bitmap) − počítačový formát pro ukládání rastrové grafiky.

Byte − jednotka množství dat v informatice, zpravidla označuje osm bitů. V češtině se může používat fonetický zápis bajt.

JPEG (Joint Photographic Experts Group) − standardní metoda ztrátové komprimace používaná pro ukládání počítačových obrázků ve fotorealistické kvalitě.

JPEG 2000 − formát na ztrátovou komprimaci obrazu založený na wavelet transformaci. Umožňuje i bezeztrátovou komprimaci.

MP3 − (MPEG-1 Layer 3) je formát ztrátové komprese zvukových souborů, založený na kompresním algoritmu MPEG.

MPEG (Motion Picture Experts Group) − název skupiny standardů používaných na kódování audiovizuálních informací (např. film, obraz, hudba) pomocí digitálního kompresního algoritmu.

PAL (Phase Alternation Line) − jeden ze standardů kódování barevného signálu pro televizní vysílání.

QT − multi platformová knihovna C++ pro vytváření programů s grafickým uživatelským rozhraním. Knihovnu vyvíjí norská společnost Trolltech (dříve Quasar Technologies).

STL (Standard Template Library) − standardní knihovna šablon jazyka C++. Obsahuje velké množství užitečných datových struktur a algoritmů.

WMA (Windows Media Audio) − komprimovaný zvukový formát vyvinutý jako součást Windows Media byl původně určen jako náhrada za MP3.

XML (eXtensible Markup Language) − obecný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C.

1 Úvod

V oblasti informatiky se od samých počátků řešil problém s nedostatkem paměťového prostoru a datové propustnosti. Hlavní důvod byl, že paměťový prostor byl drahý, a tak se s ním muselo šetřit a přenosová kapacita byla malá. Bylo známo, že data obsahují redundantní informace a ty je možno v nějakém přesném řádu „vypustit“ a soubor tak zmenšit, respektive komprimovat. Daný soubor nám tedy bude zabírat méně paměťového místa a přeci se neznehodnotí jeho informační hodnota a zpětným dekódováním půjde obnovit původní zprávu. Začaly se tedy vymýšlet různé komprimační techniky a algoritmy, které by data co nejvíce zmenšily tedy zkomprimovaly.

V dnešní době, kdy paměťová kapacita záznamových médií je obrovská ve srovnání s dobou například před 15 lety a každým rokem stoupá a v přepočtu na peněžní jednotky je jeden bajt čím dál levnější, se může zdát, že komprimační algoritmy již zdaleka nejsou tak potřebné jako v dobách nedávno minulých. Opak je ale pravdou. Jako příklad můžeme uvést datovou náročnost multimediálních souborů (obrázky, zvuk, video), kde bychom se bez jejich komprimace jen s těží obešli. V příkladu si uveďme video soubor ve standartě používané normě PAL. To má rozlišení 720x576 obrazových bodů, počítejme s 25 snímky za sekundu (norma PAL má frekvenci 50 půlsnímků za sekundu) a s barevnou hloubkou 24 bitů. Jednoduchým výpočtem dostaneme, že velikost nezkomprimovaného video souboru s délkou záznamu jedna hodina, by zabrala na datovém médiu přibližně 104 GiB, což i na dnešní kapacitu záznamových médiích je značně neúnosné. Multimediální soubory, ale většinou využívají ztrátové komprimace, tato práce se zabývá výhradně bezztrátovou komprimací dat, tzn. komprimací, kde si nemůžeme dovolit ztratit nějaká data. Využití bezztrátové komprimace jsou široké, například můžeme jmenovat komprimaci textu, nebo pokud s daty delší dobu nepracujeme, nebo je zálohujeme, vyplatí se využít komprimaci. Po komprimaci tak budou na paměťovém médiu zabírat méně místa.

S nárůstem výpočetního výkonu přibyla možnost dekomprimovat data takřka v reálném čase, uživatel si tedy ani nemusí uvědomovat, že data, se kterými právě pracuje, nebo je využívá, jsou na paměťovém médiu uloženy v zkomprimované

podobě. Příkladem může být formát docx, který používá MS Word 2007. Formát je založen na XML struktuře a společně s dalšími daty je následně zkomprimovaný.

Celé odvětví komprimace je velice rozsáhlé a existuje nepřeberné množství algoritmů využívaných ke komprimaci dat. Každý, kdo vyvíjí komprimační programy, se snaží buďto stávající algoritmy upravit tak, aby byly rychlejší a dosahovaly většího stupně komprimace anebo se snaží vyvinout nové. Takto vznikají další a další verze komprimačních algoritmů. V omezeném rozsahu práce tedy není možné všechny typy algoritmů popsat, proto jsem se zaměřil pouze na základní typy algoritmů.

Cílem této práce je přiblížit a popsat ty nejdůležitější komprimační algoritmy a techniku komprimace dat. Dále naprogramovat většinu těchto algoritmů a srovnat jejich efektivnost s komerčními komprimačními programy. Tato práce by měla sloužit jako východisko pro další studium či k rozšíření komprimačních algoritmů.

2 Rozdělení komprimace

2.1 Bezztrátová komprimace

Při bezztrátové komprimaci je možné rekonstruovat data do původní podoby, v jaké se nacházely před komprimací. Používá se tam, kde nemůžeme připustit jakoukoliv ztrátu dat. Žádný komprimační program nedokáže bezztrátovou metodou komprimovat všechny soubory, aby tak pokaždé došlo ke komprimaci dat. Existuje vždy taková kombinace vstupních dat, které nebude schopen komprimační algoritmus zkomprimovat ani o jediný bit [1]. V případě bezeztrátové komprimace nelze dosáhnout vyšší komprese, než je míra entropie[1] komprimovaných dat. Předností bezztrátové komprimace je očividná skutečnost, že dekompresí lze získat přesný duplikát původního souboru, nevýhodou je nepříliš vysoký poměr komprimace, což může dokonce někdy vést k větším výsledným souborům po komprimaci.

2.2 Ztrátová komprimace

Při použití ztrátové komprimace již není nikdy možná zpětná rekonstrukce dat do původní podoby. Ztrátová komprimace je nepoužitelná v případě, kdy je potřeba uchovat přesnou kopii původních dat, například text knihy, program nebo zálohu dat. Ztrátová komprimace se využívá hlavně pro komprimaci dat určených k smyslovému vnímání (obrázky, zvuk, video). Přesto, že se část informace při ztrátové komprimaci nenávratně ztrácí, je tento způsob ukládání dat často velmi výhodný. Ztráta některých informací je totiž zcela vyvážena velmi výrazným zmenšením komprimovaných dat. Obvykle je tak určitá (malá) ztráta kvality vyvážena výraznou úsporou místa [2].

Příkladem může být obrázek, který je uložen v nekomprimovaném grafickém formátu např. BMP, ten zabírá na disku mnohem více místa, než stejný obrázek ve formátu JPEG a přitom uživatel na obou obrázcích nevidí rozdíl (pokud nemá možnost je detailně porovnat). Toto je způsobeno tím, že většina obrázků převedena do digitální formy obsahuje často „nadbytečné“ informace, které není schopno lidské oko postihnout. Lidské oko má totiž omezenou rozlišovací schopnost, jak ve vztahu k barevné hloubce, tak obrysovým detailům. Na světlé ploše tedy těžko rozezná jeden

1 Střední hodnota informace jednoho kódovaného znaku. Označuje se H. Tuto veličinu zavedl Claude Elwood Shannon. Více v [4].

tmavý bod a stejně tak není schopno rozeznat milion barevných odstínů. Barvy, které leží blízko sebe, oko průměruje [3].

Číselné vyjádření jednotlivých pixelů bývá zcela odlišné, proto se komprimační algoritmy snaží najít v těchto číselných kombinacích nějaký řád, který jim umožní zmenšit složitost obrázku. Ačkoliv komprimace vede ke ztrátě, kvalita obrázku měřená subjektivním lidským vjemem, zůstává přibližně stejná [1]. Podobně je na tom komprimace u zvukových formátů či videa.

Příklady některých formátů, které ukládají informace pomocí ztrátové komprimace, jsou například pro obrázky a grafiku JPEG, JPEG 2000 (umožňuje i bezztrátově). Jeden z nejznámějších formátů využívající ztrátové komprimace pro audio soubory je MP3, WMA a AAC. Formát využívající ztrátovou komprimaci video souborů je například MPEG v různých verzích a mutacích.

Více informací o formátech a metodách ztrátové komprimaci lze najít například v knihách [1], [5], nebo v [6].

3 Druhy komprimace

3.1 Fyzická a logická komprimace

Všechny druhy komprimačních algoritmů mají za cíl, převádět data do kompaktnější formy, která mají stejnou informační hodnotu jako data původní. Toto nám zajišťuje logická nebo fyzická komprimace [1].

3.1.1 Logická komprimace

Logická komprimace využívá logické informační hodnoty komprimovaných dat. Používá nejčastěji logické substituce sekvence znaků jinou, úspornější řadou [1]. Příkladem můžou být různé zkratky či zkratková slova (např.: Čedok = Česká dopravní kancelář, ALCA = Advanced Light Combat Aircraft). Metody komprimace na logické úrovni využívají sémantických vědomostí o složení a možné následnosti kódovaných informací, díky tomu mohou dosahovat vyšších komprimačních poměrů, jsou však časově náročnější a použitelné pouze pro jistý specifický druh dat [5].

3.1.2 Fyzická komprimace

U fyzické komprimace nezáleží na sémantice dat a pracuje bez zřetele na logiku dat. Vytváříme nové sekvence symbolů[2] (bitů, bajtů apod.), pokud možno kratší, podle určitého komprimačního algoritmu bez využití sémantických vědomostí o složení a možné následnosti kódovaných informací. Fyzickou komprimaci používá většina komprimačních algoritmů [5].

3.2 Symetrická a asymetrická komprimace

3.2.1 Symetrická komprimační metoda

Pro symetrickou komprimační metodu platí, že čas potřebný pro komprimaci a dekomprimaci dat je přibližně stejný [1].

3.2.2 Asymetrická komprimační metoda

Vyznačuje se rozdílnými časovými nároky na komprimaci a dekomprimaci. U většiny komprimačních algoritmů je čas komprimace souboru delší, než v případě jeho dekomprimace, ale není to pravidlem.

3.3 Rozdělení podle způsobu sestavování modelu

Udávají nám, jakým způsobem algoritmus sestavuje pravděpodobnost k jednotlivým symbolům. Komprimace (zajišťuje nám kodér) a dekomprimace (zajišťuje nám dekodér) musí probíhat nad stejným modelem dat.

3.3.1 Statické komprimační metody

Využívají ke své činnosti předem známý model, který se v závislostech na vstupních datech nemění. Pravděpodobnostní model může být například vytvořen podle výskytu jednotlivých znaků abecedy v daném jazyku. Kodér i dekodér pracuje se stejným modelem, který musí předem znát. Model se tedy nepřenáší se zkomprimovanými daty. Příkladem může být například Morseova abeceda, Braillovo písmo [7]. Při komprimaci textu mohou také kodér a dekodér využívat tabulku pravděpodobnosti výskytu jednotlivých znaků abecedy v textu daného jazyka (příklad takové tabulky pro český text lze nalézt v [8]). Pokud bude kodér využívat tabulku pravděpodobnosti výskytu jednotlivých znaků v českém textu, bude se český text

2 Pod pojmem symbol můžeme rozumět cokoliv, např. se může jednat o znaky (třeba ASCII), nebo o sekvence bajtů apod.

komprimovat s dobrým komprimačním poměrem, ale pokud bude muset zkomprimovat soubor obsahující anglický text, komprimační poměr se nám značně zhorší. Výsledek komprimace je tedy silně závislý na vstupních datech, jejichž výběr je značně omezený, proto se toto řešení v praxi moc nevyužívá.

3.3.2 Semi-adaptivní komprimační metody

Model se vytváří na základě komprimovaných dat. Před vlastní komprimací se data projdou a vytvoří se model. Nevýhodou tohoto postupu, pokud pomineme, že soubor musíme procházet dvakrát, je přenos modelu mezi kodérem a dekodérem. To je hlavně problém, pokud komprimujeme malý objem dat, kdy velikost modelu může být i větší než samotná zkomprimovaná data.

3.3.3 Adaptivní komprimační metody

Na začátku předpokládáme obecný model, kde všechny symboly mají stejnou pravděpodobnost, ten poté v průběhu komprimace, respektive dekomprimace aktualizujeme. Výhodou je pouze jeden průchod komprimovaným souborem a také, že není nutné přenášet model s komprimovanými daty.

4 Rozdělení algoritmů podle metod komprimace

Komprimační algoritmy můžeme rozdělit do několika základních kategorií, podle toho, jaké základní metody ke komprimaci dat využívají.

4.1 Jednoduché komprimační metody

Využívají ke komprimaci dat posloupnost jednotlivých [bitů] či symbolů. Vyznačují se velkou rychlostí komprimace i dekomprimace, ale dosahují obecně špatného komprimačního poměru, který je značně závislý na typu vstupních dat. Patří mezi ně například metoda potlačení nul popsaná například v [5], nebo metoda proudového kódování (RLE).

4.2 Statistické komprimační metody

Jsou založené na jednotlivých četnostech výskytu symbolů v komprimovaném proudu dat. Většinou jde o dvou průchodové komprimační metody, kdy prvním průchodem proudu je spočítána četnost jednotlivých symbolů v komprimovaném vstupním proudu dat a při druhém se teprve provádí samotná komprimace. Patří sem

například Huffmanovo kódování, Shannon-Fanovo kódování a Aritmetické kódování.

4.3 Slovníkové komprimační metody

Využívají ke komprimaci slovník, který se na počátku komprimace inicializuje a během jejího průběhu se obohacuje o nové fráze. Patří sem například komprimační metody LZW, LZ77 a LZ78.

4.4 Kontextové komprimační metody

Kontextové komprimační metody používají při komprimaci informací o kontextu komprimovaného symbolu, tzn. tvar zkomprimované zprávy záleží také na okolních symbolech původní zprávy [7]. Patří sem například metody ACB, DCA a PMM.

4.5 Ostatní metody použité v komprimačních algoritmech

Při použití těchto metod nedochází k samotné komprimaci dat, ale data jsou pouze transformována, aby byl dosažen možná co nejlepší komprimační poměr. Patří sem například Burrows-Wheelerova transformace, nebo Move-to-front.

5 Základní ukazatelé udávající výkon komprimace

5.1 Komprimační poměr

Je hodnota, která nám vyjadřuje efektivnost komprimace. Komprimační poměr vypočítáme jako podíl velikosti výstupního souboru (zkomprimovaného) k souboru vstupnímu. Pokud je hodnota komprimačního poměru menší jak 1, dochází ke komprimaci souboru, čím se hodnota více blíží k nule, tím je komprimace účinnější. Pokud je hodnota komprimačního poměru větší jak 1, dochází k expanzi zkomprimovaného souboru, tedy k negativní komprimaci. Nejčastěji se komprimační poměr vyjadřuje v % podle vztahu [9]:

$$komprimační\ poměr = \frac{velikost\ výstupního\ souboru}{velikost\ vstupního\ souboru} * 100\,[\%]$$

5.2 Faktor komprimace

Faktor komprimace je dalším ukazatelem efektivnosti komprimace. Jedná se o převrácenou hodnotu komprimačního poměru. Hodnoty faktoru komprimace, které jsou vetší jak jedna, nám tedy udávají komprimaci dat a hodnoty menší jak jedna expanzi dat, tedy negativní komprimaci. Faktor komprimace vypočítáme ze vztahu [9]:

$$faktor\ komprimace = \frac{velikost\ vstupního\ souboru}{velikost\ výstupního\ souboru}$$

5.3 Rychlost komprimace

Jedná se o další důležitý údaj zobrazující výkon komprimačních algoritmů. Rychlost komprimace nám udává, kolik času je zapotřebí ke zkomprimování vstupního souboru.

5.4 Rychlost dekomprimace

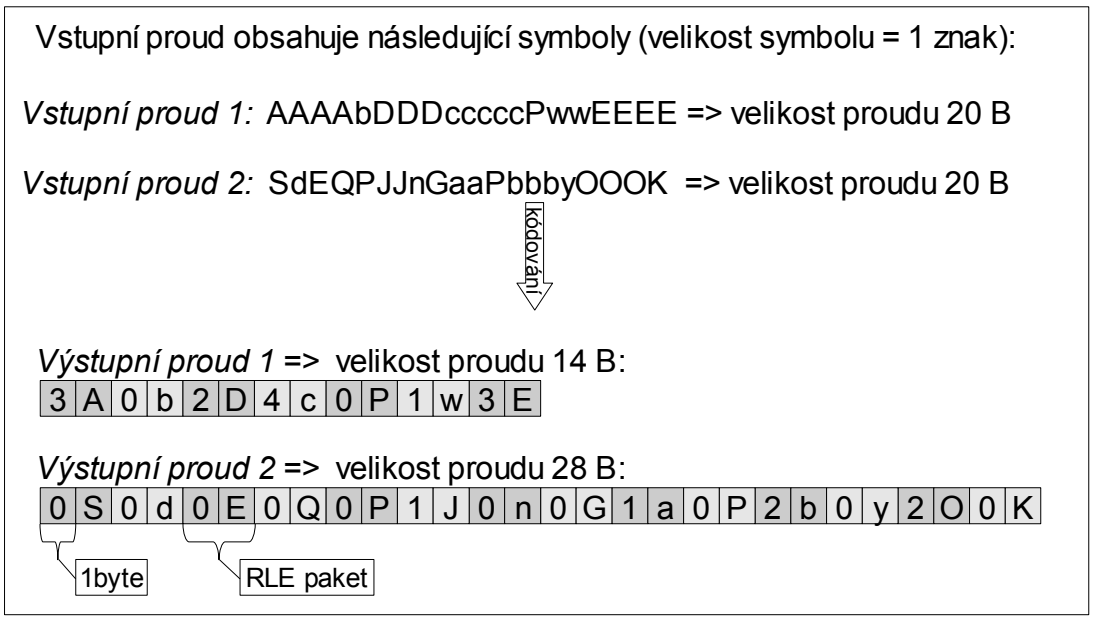
Rychlost dekomprimace nám udává, kolik času je zapotřebí k rozbalení zkomprimovaného souboru do původní podoby. Z hlediska údajů zobrazujících výkon komprimačních algoritmů se jedná o méně důležitý údaj. U většiny algoritmů je totiž rychlost dekomprimace nižší než rychlost komprimace [1].

6 Popis jednotlivých algoritmů pro bezztrátovou komprimaci dat

6.1 RLE (Run-Length Encoding)

Komprimační metoda Run-Length Encoding označována zkratkou RLE, v češtině občas nazývána metoda proudového kódování, je velice jednoduchý algoritmus pro bezztrátovou komprimaci dat. Vyznačuje se velkou rychlostí komprimace i dekomprimace, ale obecně nízkým komprimačním poměrem [1].

Hlavní myšlenkou algoritmu je zhuštění po sobě jdoucích stejných symbolů, tyto symboly se nazývají proud. Proud symbolů se komprimuje do tzv. RLE paketů. Jak můžeme vidět na obrázku 1, RLE paket je tvořen proudovým číslem, které udává počet opakujících se symbolů snížený o jedničku a proudovou hodnotou, která nám udává opakující se symbol.



Obrázek 1: Příklad komprimace pomocí základní metody RLE

Tento způsob tvorby RLE paketu není příliš vhodný, protože pokud by v souboru bylo málo po sobě opakujících se symbolů, s největší pravděpodobností by docházelo k záporné komprimaci. Proto existuje řada modifikací tvorby RLE paketů, které se snaží vyhnout záporné komprimaci při nízkém počtu opakujících se symbolů. Ty nejznámější jsou níže popsány.

6.1.1 Tvorba RLE paketu pomocí identifikátoru na bajtové úrovni

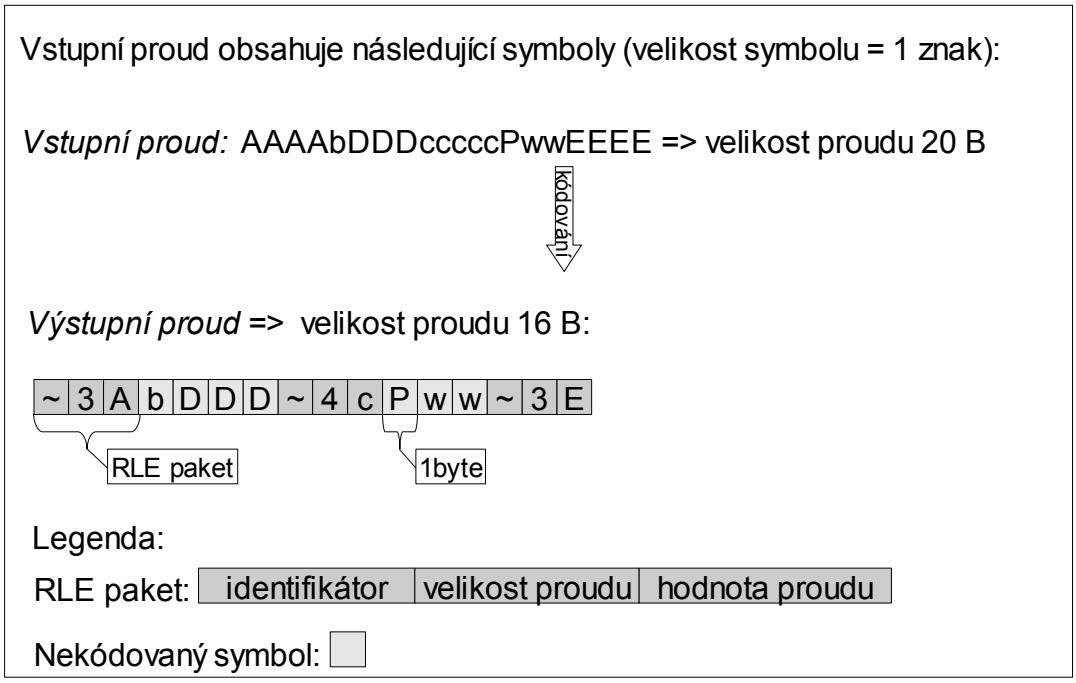
Tato modifikace algoritmu RLE předchází záporné komprimaci zavedením tzv. identifikátoru, který nám ve zkomprimovaném souboru uvozuje začátek RLE paketu.

Při komprimaci se zakódují pouze ty proudy, které obsahují více jak tři symboly. Pokud je proud dat menší, jak tři symboly, zapíší se do komprimovaného toku dat přímo. Je-li tedy proud dat vetší jak tři symboly, data se zakódují následujícím způsobem: první bajt obsahuje identifikační hodnotu, druhý proudové číslo (počet opakujících se symbolů snížený o jedničku) a třetí proudovou hodnotu (opakující se symbol). Příklad komprimace můžeme vidět na obrázku 2.

6.1.2 Tvorba RLE paketu pomocí identifikátoru na bitové úrovni

Základní myšlenka této modifikace metody RLE spočívá v tom, že nám nejvyšší bit v bajtu velikosti proudu představuje tzv. identifikátor typu proudu. Pokud je nej-

vyšší bit nastaven na hodnotu 1, jedná se o klasický RLE paket. Nižších 7 bitů nám pak udává proudové číslo a následující bajt představuje proudovou hodnotu. Maximální velikost proudu tedy může být 128 symbolů. Je-li identifikátor typu proudu nastaven na hodnotu 0, jedná se o tzv. přímý paket. V tomto případě nám proudová hodnota (opět uložená v nižších 7 bitech bajtu identifikátoru) udává přesný proud znaků, které se čtou v té podobě, jako jsou zapsána.



Obrázek 2: Příklad RLE kódování s využitím identifikátoru na bajtové úrovni

Algoritmus vyniká díky své jednoduchosti vysokou komprimační a dekomprimační rychlostí. Nevýhodou metody RLE je úzká oblast dat, na kterých dosahuje dobrých kompresních poměrů. Je vhodná například pro komprimaci obrázků s malou barevnou hloubkou či jako sekundární algoritmus pro komprimaci dat po nějaké transformaci (nejčastěji textu, na který byla aplikována Burrows-Wheelerova transformace). V praxi se využívá například ve formátu PCX, informace a popis implementace můžeme najít v [10] nebo [3].

6.2 Huffmanovo kódování

Huffmanovo kódování je neznámějším zástupcem algoritmů, které pracují na základě různých četností znaků v kódovaných datech [1]. Algoritmus v roce 1952 poprvé představil David A. Huffman [11].

Algoritmus vytváří Huffmanův kód, tj. kód s minimální délkou (ve smyslu Shannova teorému) a také se jedná o takzvaný prefixový kód[3]. Hlavní myšlenkou je přiřazení zástupného kódu k jednotlivým symbolům vstupní abecedy. Délka zástupného kódu závisí na jednotlivých četnostech výskytu symbolů. Symboly vstupní abecedy s vetší četností výskytu se zapisují na výstup s využitím malého množství bitů (nejčetnější symbol může být zapsán i v podobě jednoho bitu), naopak symboly, které se vyskytují ve vstupní abecedě méně častěji, se zapisují na výstup větším počtem bitů (není výjimkou, že datová velikost výstupního kódu může přesáhnout i původní datovou velikost vstupního symbolu).

K získání zástupného kódu se používá datová struktura binární strom, ve kterém jeho listy představují symboly vstupní abecedy, hrany jsou ohodnoceny symboly 0 (levé větve) a 1 (pravé větve) a všechny uzly stromu (včetně listů) jsou ohodnoceny pravděpodobností výskytu jednotlivých symbolů ve vstupní abecedě. Pravděpodobnost vnějšího uzlu je dána vždy součtem pravděpodobností všech vnitřních uzlů, z toho vyplývá, že kořen stromu má vždy pravděpodobnost rovnou jedné. Zástupný bitový kód vstupního symbolu se pak získá zřetězením hodnot hran, kterými projdeme průchodem stromu od kořene do daného listu.

6.2.1 Komprimace

Vlastní algoritmus pracuje tak, že nejdříve projde vstupní soubor a vytvoří statistiku četností jednotlivých symbolů obsažených ve vstupním souboru. Dále setřídíme symboly vzestupně podle pravděpodobnosti výskytu a poté vytváříme binární strom. Podle zásad Huffmanova algoritmu se binární strom vytváří od listů a od znaků s nejmenší pravděpodobností výskytu. Binární strom můžeme vytvářet několika způsoby, popíši zde pouze ten, který jsem použil ve své implementaci tohoto algoritmu.

Do datové struktury prioritní fronta budou uloženy všechny pravděpodobnosti výskytu jednotlivých symbolů, kde nejvyšší prioritu (bude odebrán jako první) má prvek s nejnižší pravděpodobností.

V dalším kroku odebereme vždy z fronty dva prvky (s nejnižší pravděpodobností) a spojíme je v jeden uzel, který bude mít ohodnocení dané součtem pravděpodobností svých synů (odebraných záznamů). Do fronty poté vložíme nový prvek

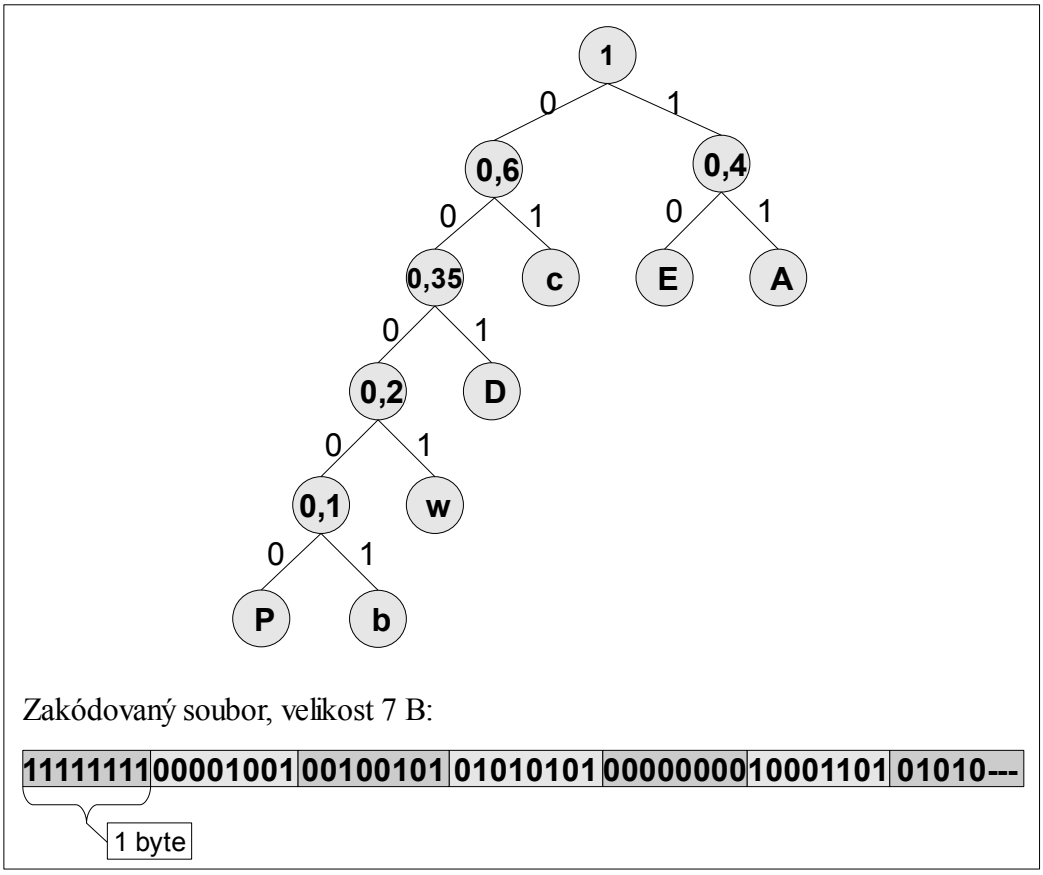
3 Podmínka pro jednoznačnou dekódovatelnost kódu. Předpona kódu znaku nesmí být kódem znaku jiného, jinak nepůjde kód správně dekódovat.

(námi vytvořený uzel). Tento postup opakujeme až do té doby, než nám ve frontě zůstane pouze jediný záznam, který nám představuje kořen stromu s pravděpodobností rovné 1.

Pro upřesnění uvedeme příklad. Vstupní proud o velikosti 20 B obsahuje následující symboly (velikost symbolu je jeden znak): *AAAAbDDDcccccPwwEEEE*. Pravděpodobnost jednotlivých znaků je uvedena v tabulce 1. Vytvořený binární strom a zakódovaný vstupní proud viz obrázek 3.

Tabulka 1: Četnost jednotlivých znaků ve vstupním proudu.

Znak	Četnost	Pravděpodobnost	Kód
c	5	0,25	01
A	4	0,2	11
E	4	0,2	10
D	3	0,15	001
w	2	0,1	0001
b	1	0,05	00001
P	1	0,05	00000
SUMA	20	1	



Obrázek 3: Příklad Huffmanova binárního stromu a zakódovaného souboru.

Po vytvoření binárního stromu máme definované kódy, které jsou přiřazené podle četnosti výskytu k jednotlivým symbolům. Kódování probíhá tak, že ke vstupnímu symbolu vyhledáme příslušný bitový kód a ten zapíšeme na výstup. Bitový kód odpovídá zřetězení hodnot hran, kterými projdeme při cestě z kořene do daného listu, který odpovídá právě kódovanému symbolu.

K dekódování vstupního proudu dat potřebujeme stejný binární strom, jako jsme měli v případě kódování. Z toho vyplývá, že se jedná o metodu semi-adaptivní. Proto dekodéru musíme předat strukturu binárního stromu, nejčastěji ji zapisujeme na začátek výstupního souboru při kódování. Po znovu sestavení binárního stromu, čteme vstupní proud dat bit po bitu a postupně od kořene procházíme stromem do té doby, než se dostaneme k listu, který nám představuje zakódovaný symbol. Symbol zapíšeme na výstup a strom začneme procházet opět od kořene. Toto opakujeme do té doby, dokud není konec vstupního proudu.

6.2.2 Huffmanovo kódování (adaptivní)

Algoritmus je založen na původním Huffmanově kódování. Jedná se o nejstarší adaptivní algoritmus, byl nezávisle publikován Fallerem (1973), později Gallagerem (1978). V roce 1985 provedl další obměnu tohoto algoritmu Knuth. Tento algoritmus dostal zkrácený název FGK [7].

Binární strom má sourozeneckou vlastnost, jestliže každý uzel (s výjimkou kořene) má sourozence a pokud lze uzly seřadit do monotónní posloupnosti (podle četnosti daného uzlu) tak, že má každý uzel v posloupnosti za souseda svého sourozence. Gallager potom dokázal, že binární prefixový kód je Huffmanovým kódem právě tehdy, když má odpovídající kódovací strom sourozeneckou vlastnost. Algoritmus potom funguje na principu vkládání symbolů do stromu a v případě, že se podaří detekovat porušení sourozenecké vlastnosti, se provede přeuspořádání stromu tak, aby byla sourozenecká vlastnost zachována [12].

Algoritmus má dvě varianty přístupu k abecedám, v prvním případě je strom na začátku nastaven tak, že obsahuje všechny znaky se zvolenou pravděpodobností. Druhý případ používá tzv. uzel *zero*. V tomto případě obsahuje počáteční strom pouze jediný uzel *zero*. Při načtení znaku, který dosud nebyl zakódován, se do výstupu vypíše kód uzlu *zero* a uzel se rozdělí na nový uzel *zero* a list s novým znakem.

Další adaptivní verzí Huffmanovo kódování je Vitterův algoritmus. Jedná se o podobný algoritmu jako FGK, ale používá trochu jiný způsob upravování stromu a v určitých případech tak dává lepší výsledky. Podrobnější popis FGK a Vitterova algoritmu můžeme nalézt například v [9] nebo [13].

6.2.3 Problém optimálního kódu a praktické využití

Huffmanovo kódování produkuje ideální kód (tzn. kód, jehož průměrná délka se rovná entropii) pouze v případě, že pravděpodobnosti výskytu jednotlivých symbolů jsou čísla o mocnině ½ (1/2, 1/4, 1/8 atd.). V praxi je to ale spíše ojedinělý jev. Huffmanova metoda každému symbolu vstupní abecedy musí přiřadit celočíselný počet bitů. Tím dochází k nedokonalé komprimaci, pokud máme například symbol s pravděpodobností výskytu 0,4, bylo by ideální přiřadit tomuto symbolu kód o velikosti 1,32 bitů ($\log_2 0{,}4 \approx 1{,}32$). To ovšem není možné, a tak Huffmanův algoritmus přiřadí s největší pravděpodobností tomuto symbolu kód jednobitový, nebo dvoubitový. Tento problém řeší až aritmetické kódování popsané v kapitole 6.4.

Huffmanova kódování se využívá například v metodě Deflate (zjednodušeně řečeno je to kombinace slovníkové metody LZ-77 a Huffmanova kódovaní). Využívá se i ve ztrátové komprimaci konkrétně v komprimaci JPEG. Zde se používá v poslední fázi, kde se pomocí Huffmanova kódování zakóduje „zig-zag“ posloupnost hodnot bloku, kde se využívá její bezztrátovost. Dále by jsme jej nalezli v komprimačních algoritmech a programech jako jsou například PKZIP, MP3 a bzip2 [18].

6.3 Shannon-Fanovo kódování

Tato metoda pochází z roku 1949. Publikovali ji nezávisle na sobě Claude Elwood Shannon (ten je označován jako otec teorie informace) s Warrenem Weaverem a Robert Mario Fano.

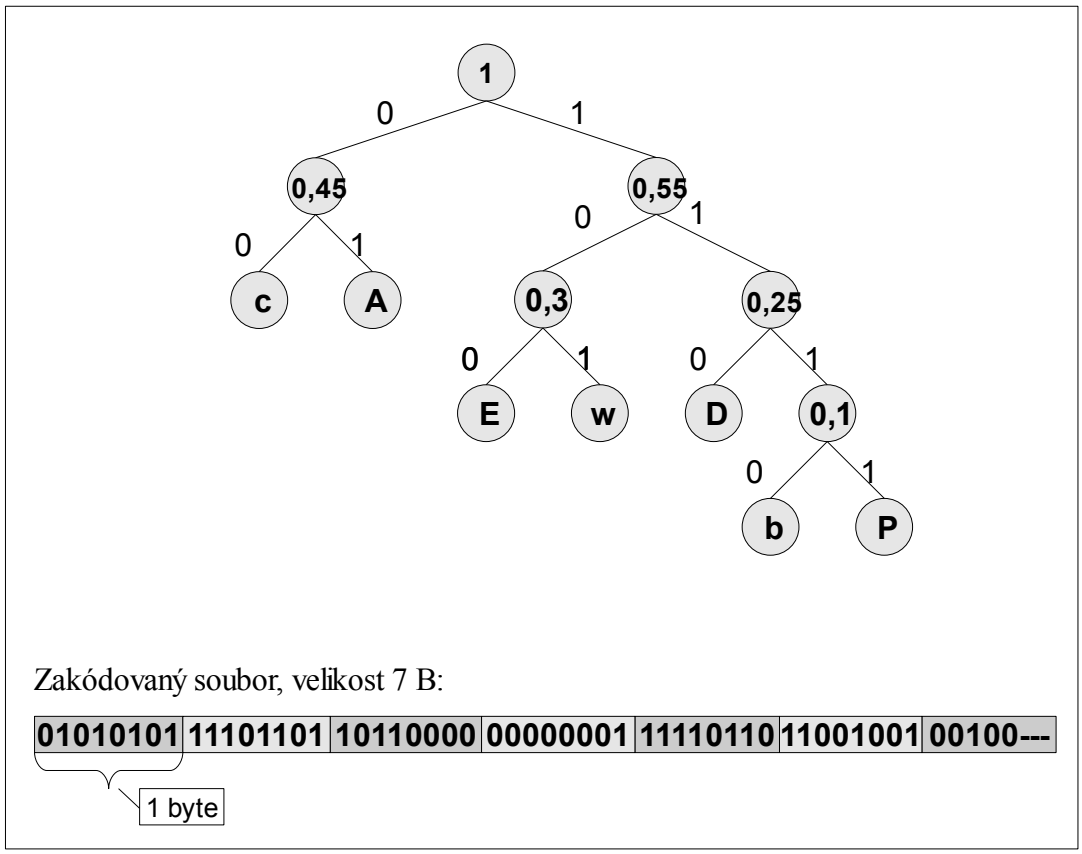
Jedná se o statistickou, semi-adaptivní metodu. Shannon-Fanovo kódování je velice podobné Huffmanovu kódování, Huffman při tvorbě algoritmu z tohoto kódování vycházel. Jediný podstatný rozdíl mezi oběma metodami je při tvorbě binárního stromu [1].

U Huffmanova kódování vytváříme binární strom od listů s nejmenší pravděpodobností, až ke kořenu, Shannon-Fanovo kódování má opačný přístup. Vytváří

tedy binární strom od kořene až ke koncovým listům, tvorba binárního stromu je tedy jednodušší.

Komprimace a dekomprimace pomocí algoritmu probíhá stejně jako u výše popsaného Huffmanova kódování (semi-adaptivního), proto ji zde nebudeme podrobněji popisovat a zaměříme se pouze na rozdílnou konstrukci binárního stromu. Ze začátku musíme opět vytvořit pravděpodobnost jednotlivých symbolů vyskytujících se ve vstupním proudu dat.

Vytváření binárního stromu, jak již bylo řečeno, probíhá od kořene k listům. V prvním kroku rozdělíme soubor vstupní abecedy na dvě skupiny se stejnou nebo co nejbližší pravděpodobností. První skupinu, která se nachází vlevo, označíme binární nulou a druhou jedničkou. Můžeme říci, že celá vytvořená skupina je nový list stromu. Toto dělení na skupiny se stejnou nebo nejvíce podobnou pravděpodobností opakujeme tak dlouho, dokud skupina nebude obsahovat pouze jediný symbol vstupní abecedy. Příklad vytvořeného binárního stromu můžeme vidět na obrázku 4. Vstupní proud symbolů je stejný jako v příkladu v kapitole 6.2 a rozdělení pravděpodobností výskytu jednotlivých symbolů můžeme vidět v tabulce 1.



Obrázek 4: Příklad Shannon-Fanova binárního stromu a zakódovaného souboru

Z uvedeného příkladu nevyplývají žádné rozdíly v délce výstupního kódu, jedině strom má jinou konstrukci. Rozdíl mezi Huffmanovým a Shannon-Fanovo kódováním se neprojevuje tedy vždy, ale zatímco Huffmanovo kódování generuje vždy optimální kódy pro jednotlivé symboly, Shannon-Fanovo kódování může v některých případech použít v kódech pár bitů navíc [1].

V praxi se Shannon-Fanovo kódování většinou nepoužívá, pro komprimaci se raději volí Huffmanovo kódování, které může dosahovat lepších výsledků, nikdy však nedosáhne horších výsledků než Shannon-Fanovo kódování.

6.4 Aritmetické kódování

Jak jsme uvedli již výše, aritmetické kódování řeší jednu z hlavních nevýhod Huffmanova kódování a to takovou, že optimální kód vzniká pouze tehdy, pokud je pravděpodobnost symbolů rovna mocnině ½. Více tento problém rozebíráme v závěru kapitoly 6.2, která se zabývá Huffmanovým kódováním [14].

Aritmetické kódování reprezentuje zprávu jako číslo z intervalu <0,1). Na začátku uvažujeme celý tento interval. Jak se zpráva prodlužuje, zpřesňuje se i výsledný interval a jeho horní a dolní mez se k sobě přibližují. Čím je kódovaný symbol pravděpodobnější, tím se interval zúží méně a k zápisu nám tedy stačí méně bitů. Nakonec stačí zapsat libovolné číslo z výsledného intervalu, které nám samo o sobě reprezentuje celou zprávu.

6.4.1 Komprimace

Algoritmus kódování můžeme zjednodušeně zapsat jako:

1. Inicializuj hodnoty:
   Dolní interval =0
   Horní interval =1
2. Opakuj tak dlouho, dokud není vstup prázdný:
   (a) přečti vstupní symbol (s)
   (b) *Rozsah= Horní interval - Dolní interval*
   (c) *Horní interval = Dolní interval + Rozsah * Horní interval symbolu(s)*
   (d) *Dolní interval = Dolní interval + Rozsah * Dolní interval symbolu(s)*
3. Výstupem je libovolná hodnota z intervalu (*Dolní interval, Horní interval*> a hodnota udávající délku vstupních dat, nebo speciální ukončovací symbol [15].

Pro podrobnější vysvětlení kódování uvádím příklad:

Vstupní soubor obsahuje znaky: *abac*

Vstupní abeceda je tedy: *a, b, c*

Pravděpodobnost znaků je a=0,5; b=0,25; c=0,25; nakonec symbolům přiřadíme intervaly a(0; 0,5>, b(0,5; 0,75>, c(0,75; 1>

Průběh kódování můžeme vidět v tabulce 4.

Tabulka 2: Příklad kódování zprávy pomocí aritmetického kódování

Krok	Interval	Symbol	Výpočet intervalu
1	H=1,0 D=0,0 R=1,0	H=0,5 a D=0,0	H= 0 + 1 * 0,5 = 0,5 D= 0 + 1 * 0,0 = 0,0
2	H=0,5 D=0,0 R=0,5	H=0,75 b D=0,5	H= 0 + 0,5 * 0,75 = 0,375 D= 0 + 0,5 * 0,50 = 0,25
3	H=0,375 D=0,25 R=0,125	H=0,5 a D=0,0	H= 0,25 + 0,125 * 0,5 = 0,3125 D= 0,25 + 0,125 * 0,0 = 0,25
4	H=0,3125 D=0,25 R=0,0625	H=1,0 c D=0,75	H= 0,25 + 0,0625 * 1,0 = 0,3125 D= 0,25 + 0,0625 * 0,75 = 0,296875
5	H=0,3125 D=0,296875	Libovolné číslo z výsledného intervalu <0,296875; 0,3125) nám reprezentuje danou zprávu. Vybereme pokud možno nejkratší reprezentaci čísla v našem případě to může být hodnota 0,3.	

6.4.2 Dekomprimace

Dekomprimace zakódované zprávy probíhá obdobným způsobem. Příklad kódování je uveden na statickém modelu, proto je třeba do zkomprimovaného souboru zapsat informace, aby dekodér dokázal znovu sestavit intervaly jednotlivých symbolů. Můžeme například uložit pravděpodobnost výskytu jednotlivých symbolů a před výpočtem intervalu seřadit symboly abecedně, to platí i v případě kódování, aby nám nedošlo např. k záměně intervalu symbolů se stejnou pravděpodobností. Dále uvádím popis dekódovacího algoritmu:

1. Načti zakódovanou hodnotu (*hodnota*) a zrekonstruuj intervaly symbolů.
2. Inicializuj hodnoty:
   Dolní interval =0.0
   Horní interval =1.0

3. Opakuj tak dlouho, dokud se počet dekódovaných symbolů nerovná počtu zakódovaných symbolů či dokud nenarazíš na konstantu „eof“:
   (a) urči, do jakého intervalu patří (*hodnota)*
   (b) zapiš na výstup symbol (*s*), kterému náleží daný interval
   (c) *hodnota = [hodnota - Dolní inter. symbolu(s)] / Pravděpodobnost . symbolu(s)*

Dekódování hodnoty 0,3 z minulého příkladu ukazuje tabulka 3

Tabulka 3: Příklad dekódování zprávy pomocí aritmetického kódování

Krok	Hodnota	Aktuální interval	Výstup	Výpočet
1	0,3	a <0; 0,5)	a	hodnota = (0,3 – 0) / 0,5 = 0,6
2	0,6	b<0,5; 0,75)	b	hodnota = (0,6– 0,5) / 0,25 = 0,4
3	0,4	a <0; 0,5)	a	hodnota = (0,4 – 0) / 0,5 = 0,8
4	0,8	c<0,75; 1)	c	---
Výstup: *abac*				

6.4.3 Reprezentace v celočíselné aritmetice

Základní myšlenku aritmetického kódování jsme vysvětlili výše, avšak při jejím převádění do počítačové praxe se setkáváme s celou řadou problémů. Nejzávažnější je asi potřeba konverze reálných čísel na bitové vyjádření. Výsledný kód reprezentující vstupní posloupnost je zapotřebí převést do formy vyjádřitelné ve dvojkové reprezentaci. Navíc s každým reálným číslem rozdělujícím základní interval <0, 1) je nezbytné pracovat s vyloučením jakéhokoliv zaokrouhlování. Dalším problémem je rychlý růst počtu platných desetinných míst v racionálních číslech, která představují horní a dolní meze intervalů. Ve většině případů již u 10 symbolů roste počet platných desetinných míst na dvacet až třicet [1].

Řešení těchto problémů je tedy několik, popíši zde v krátkosti pouze jedno z možných řešení, které je podrobněji rozebráno v [16], toto řešení jsem použil ve své implementaci tohoto algoritmu.

V implementaci jsem použil 64 bitovou ne znaménkovou proměnnou, pro jednoduchost budu v následujícím popisu používat neznaménkovou proměnnou o velikosti 8bit. K tomu, aby nedocházelo k přetečení proměnné, použijeme vždy o jeden bit méně, než je velikost proměnné. Při výpočtech a zápisech pracujeme pouze s desetinnou částí intervalu, například číslo 0,101011 zapíšeme jako 101011.

V prvním kroku si nastavíme *Dolní interval*=0x0, *Horní interval* =0x7F (což je maximální hodnota, kterou lze zapsat do 7bitů). Zde nastává hned první problém, jak jsme uvedli výše, uvažujeme pouze desetinnou část, horní interval však může nabývat i hodnoty 1,0. Tento problém lze vyřešit tak, že pro horní interval použijeme maximální hodnotu 0,9999... a vždy když budeme s horním intervalem počítat, tak k němu musíme přičíst hodnotu 1, abychom dostali jeho skutečnou hodnotu, avšak při zápisu nesmíme zapomenout tuto hodnotu opět odečíst. Poté nám stačí určit intervaly jednotlivých kódovaných symbolů (vzorce jsou vedeny níže v popisu algorimu kódování). Poté jsme již schopni zprávu zakódovat s využitím upravených vzorců použitých při práci s reálnými čísly (vzorce opět uvádíme níže v popisu algorimu kódování). Dříve nebo později bychom však narazili na další problém, jak známo intervaly se k sobě přibližují a nakonec bychom se dostali do stavu, kdy by se lišily pouze v jediném bitu a další kódování by již nebylo možné. Toto řeší takzvané E1, E2 škálování.

Jakmile hodnoty *Dolní interval* a *Horní interval* leží ve stejné polovině číselného intervalu (v našem případě jsou buď obě hodnoty menši než 0x20 anebo ≥ 0x20), je zaručeno, že tento interval nikdy neopustí, neboť další vstupní symboly interval pouze zmenšují. Zjednodušeně se dá říci, že pokud se nejvýznamnější bity proměnných rovnají, už se nikdy nezmění a mi tento bit můžeme vyjmout a zapsat na výstup. Nacházejí-li se intervaly ve spodní polovině, zapíšeme 0, E1 škálování, v horní polovině zapíšeme 1, E2 škálování. Jelikož došlo k bitovému posunu zprava (na pozici nejnižšího bitu), zapíšeme bit o hodnotě 0 u *dolního intervalu* respektive 1 u *horního intervalu.* V proměnných tedy bude uložena pouze aktivní část čísla, můžeme si je poté představit jako nekonečně velká čísla a nejsme omezeni velikostí použitého datového typu.

Problém ovšem nastává, pokud obě hodnoty *Dolní interval* a *Horní interval* konvergují ke středu intervalu, to znamená, že se liší v nejvýznamnějším bitu, program poté nemůže provést E1/E2 škálování a tím nebude generován žádný výstup. Řešením je tyto stavy detekovat včas a provést přeškálování intervalů, toto nazýváme E3 škálování. Princip spočívá ve vyjmutí druhého nejvýznamnějšího bitu, nastane bitový posun, zprava (na pozici nejnižšího bitu) zapíšeme bit o hodnotě 0 u d*olního intervalu,* respektive 1 u *horního intervalu*. Dále zavedeme proměnnou čítač, kterou zvýšíme o 1. E3 škálování provádíme do té doby, pokud hodnota *dolního intervalu* je

menší nebo rovna jedné čtvrtině počátečního intervalu (v našem případě 0x20) a zároveň *horní interval* je menší než třetí čtvrtina počátečního intervalu (v našem případě 0x60). Poté když nastane E1/E2 škálování a bit se zapisuje na výstup, musí následovat zapsání na výstup tolika bitů, jaká je hodnota čítače, a to buď o hodnotě nula pokud nastalo E2 škálování (nejvýznamnější bit=1), nebo 1, pokud nastalo E1 škálování (nejvýznamnější bit=0).

Zjednodušeně by se dal celý algoritmus kódování zapsat následovně, nerozebíráme v něm podmínky nastání škálování a jeho řešení, ty jsme podrobně vysvětlili výše.

Algoritmus kódování zprávy:

1. Inicializuj hodnoty:
   Dolní interval =0x0
   Horní interval =0x7F
2. Analyzuj soubor (pouze v případě statického kódování), potřebné informace:
   (a) celkový počet symbolů (*CelkPocSymb*), celkový počet jednotlivých symbolů (*PocSymb*)
   (b) urči intervaly každého symbolu:

$$Dolní\ inter.\ symbolu(s) = \sum_{i=0}^{s-1} PocSymb(i)$$

$$Horní\ inter.\ symbolu(s) = Dolní\ inter.symbolu(s) + PocSymb(s)$$

3. Opakuj tak dlouho, dokud není vstup prázdný:
   (a) načti symbol *s*
   (b) Vypočítej:

   krok = (Horní interval + 1 - Dolní interval) / CelkPocSymb

   *Horní interval = Dolní interval + krok * Horni inter. symbolu(s) -1*

   *Dolní interval = Dolní interval + krok * Dolní inter. symbolu(s)*

   (c) Může být provedeno E1/E2 škálování

   ANO= zapiš bit na výstup

   NE= pokračuj bodem 3

Dekódování probíhá obdobným způsobem jako kódování, máme proměnnou *buffer,* do které načteme část sekvence vstupu (v našem případě by se jednalo o 7bitů). Vypočítáme hodnotu ze vztahu: *(buffer - Dolní interval) / krok.* Do jakého intervalu hodnota spadá, tím se určí dekódovaný symbol, který se zapíše na výstup. Opět musíme hlídat, zda nenastala podmínka pro škálování, u všech třech typů škálování mě-

níme hodnotu bufferu a načítáme další bit ze souboru. Podrobnější popis dekódování nalezneme v [16].

6.4.4 Adaptivní verze aritmetického kódování

Implementace adaptivní verze není moc odlišná od své semi-adaptivní verze popsané výše. Jediný rozdíl je v přístupu k modelu. U semi-adaptivní verze musíme nejdříve projít soubor a vypočítat pravděpodobnost výskytu všech vyskytujících se symbolů, sestavený model poté musíme uložit – nejčastěji do výstupního souboru, aby byla možná dekomprimace. Toto všechno u adaptivní verze odpadá, výpočet pravděpodobnosti jednotlivých symbolů probíhá během komprimace. Kodér i dekodér začínají s modelem, jenž obsahuje všechny symboly a k nim je přiřazena stejná pravděpodobnost. Například při kódování symbolů o velikosti 8 bitů, to znamená, že vstupní proud dat může obsahovat 256 různých symbolů. Každý symbol tak bude mít na začátku kódování i dekódování přiřazenu pravděpodobnost rovnající se 1/256.

Samotné kódování probíhá způsobem, že přečteme ze vstupního proudu symbol – ten zakódujeme pomocí aritmetického kódování stejně, jak už jsme popsali výše. Dále upravíme pravděpodobnost výskytu symbolů v modelu, tento postup opakujeme do té doby, než je konec vstupního proudu symbolů. Dekódování je také stejné jako u semi-adaptivní verze, jen s rozdílem, že po dekódování symbolu upravíme pravděpodobnost v modelu.

6.4.5 Využití aritmetického kódování

Aritmetické kódování je chráněno několika patenty [17], to je jeden z důvodů, proč není tolik rozšířeno v komprimačních programech. Příkladem může být program bzip2, který v původní verzi využíval aritmetické kódování, ale díky patentům musel přejít na Huffmanovo kódování, které obecně dosahuje horších výsledků. Programy, které využívají aritmetické kódování, jsou například BMA, GRZip II [18].

Obdobou aritmetického kódování je Range encoding, rozdíl mezi nimi je v použitém intervalu, který u Range encoding je <0, 999999), dělení se provádí na celočíselné subintervaly. Výhodou je nezatíženost patenty, více o Range encoding v [19].

6.5 LZ-77

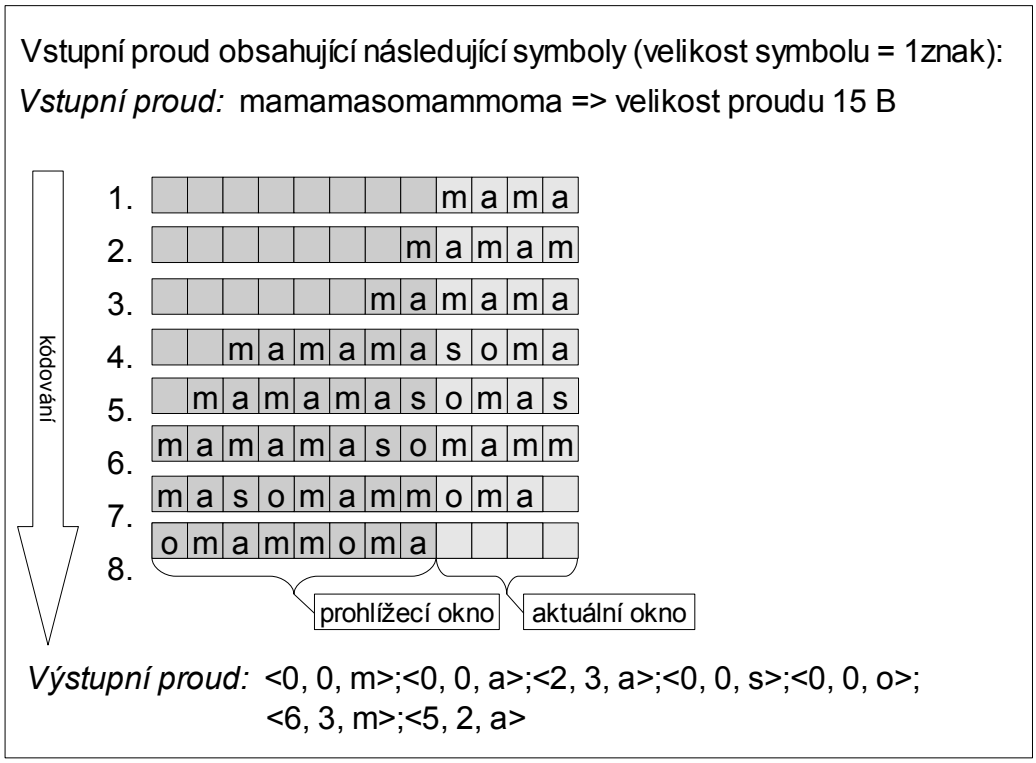
Tuto komprimační metodu uvedli poprvé v roce 1977 Abraham Lempel a Jacob Ziv. Jedná se o metodu slovníkovou, která položila základ mnoha jiným slovníkovým metodám, které jsou více či méně od této metody odvozeny.

Slovníkové metody mají několik výhod oproti metodám statistickým, které jsme popsali výše. Statistické metody nahrazují symbol kratším vyjádřením hlavně podle pravděpodobnosti výskytu, které reprezentuje model.

Algoritmus LZ-77 kóduje vstup jako posloupnost trojic. Základní myšlenkou je, že pokud kódujeme vstup od nějakého místa, pak můžeme využít předcházející vstup. Pokud najdeme nějakou shodnou sekvenci, je dán pouze odkaz na tuto sekvenci [20]. Metoda je založena na principu posuvného okna, které je rozděleno na dvě části. První část mnohonásobně větší, nám reprezentuje slovník a při dalším popisu této metody budeme tuto část nazývat jako *prohlížecí okno*. Do *prohlížecího okna* jsou postupně ukládány již zpracované symboly. Druhá kratší část posuvného okna, nazývejme ji *aktuální okno*, obsahuje přednačtené, ještě nezpracované symboly. Velikost *prohlížecího okna* se nejčastěji pohybuje od 2^{12} do 2^{16}bitů [21].

6.5.1 Komprimace

Průběh komprimace je patrný z obrázku 5. Nejdříve naplníme aktuální okno, symboly začátkem vstupního proudu, *prohlížecí okno* může zůstat neinicializováno, nebo ho můžeme naplnit například nějakým libovolně zvoleným symbolem. Poté se snažíme nalézt v *prohlížecím okně* co nejdelší předponu, která odpovídá nezkomprimované části v *aktuálním okně*. Výstup poté reprezentujeme jako trojici, která se skládá z hodnoty ukazující nám začátek nalezené předpony ve slovníku, dále délku této předpony a první nenalezený symbol. Následující symbol uvádíme z důvodu, abychom ošetřili případ, kdy nenajdeme žádnou shodu. Poté celé okno posuneme doprava o délku nalezené předpony zvětšenou o jedna. Toto opakujeme do té doby, dokud obsahuje *aktuální okno* symboly.



Obrázek 5: Příklad kódování pomocí LZ-77

Při komprimaci může nastat případ, kdy nalezená nejdelší předpona, není celá v *prohlížecím okně*, ale zasahuje nám i do nezkomprimovaných dat tzn. do *aktuálního okna*. Tuto situaci můžeme připustit, pokud splníme několik podmínek. Předpona musí vždy začínat v *prohlížecím okně*. Dále je nutné si dát pozor na to, aby výstupní trojice končila vždy nejvýše posledním načteným symbolem, a to i v případě, že by ho bylo možné reprezentovat pomocí slovníku. Tímto jsme rozšířili slovníkovou část i o nezkomprimovaná data, respektive pouze o část *aktuálního okna*, což nám pomůže zvýšit komprimační poměr.

6.5.2 Dekomprimace

Dekomprimace dat zakódovaných pomocí algoritmu LZ-77 je jednoduchá, jelikož nejsou potřeba žádné specifické informace. Přečteme vždy celou trojici zakódovaných dat, nahlédneme do slovníku (ten je reprezentován doposud dekódovanými symboly) na udávanou pozici a přečteme celý blok dat, jehož velikost nám udává druhá hodnota. K bloku dat přidáme poslední položku z trojice a vše zapíšeme na výstup (nakonec doposud dekomprimovaných dat). Pokud jsme při komprimaci připustili, aby nám předpona zasahovala od *aktuálního okna*, musíme data zapisovat na výstup, respektive do slovníku, ne po blocích, ale po jednotlivých symbolech, pro-

tože data, která se vyskytovala v *aktuálním* okně, jsou k dispozici až po částečné dekomprimaci.

Jak z výše uvedeného vyplývá, čas komprimace bude několikanásobně větší, než dekomprimace, jedná se tedy o metodu asymetrickou. Komprimaci nám ovlivňuje velikost slovníkové části, tedy prohlížecího okna. Pokud zvolíme větší datovou velikost okna, zvýší se nám pravděpodobnost nalezení delší předpony právě kódovaných dat, ale prodlouží se nám délka komprimace. Délka komprimace je tedy závislá na velikosti prohlížecího okna a zároveň na metodě jeho prohledávání. Klasický sekvenční přístup je velice časově náročný, využívají se jiné (sice paměťově náročnější) datové struktury. Pro rychlejší vyhledávání můžeme použít hashovací tabulku nebo binární vyhledávací strom.

6.5.3 Využití metody LZ-77

Metoda je vhodná pro jednorázové zakódování a četná dekódování. Používá se v algorimu zvaném Deflate, což je kombinace metody LZ-77 a Huffmanova kódování. Nejdříve je vstupní proud zakódovaný metodou LZ-77 a poté se pro zakódování pozic znaků a délek použije Huffmanovo kódování. Více o Deflate například v [22]. Tuto metodu můžeme například nalézt v komprimačních programech GZip,WinRAR, PKZip [18].

6.6 LZSS

Tento algoritmus v roce 1982 publikoval James Storer a Thomasem Szymanski. Vychází z algoritmu LZ-77, základem je opět posuvné okno, které je opět rozděleno na prohlížecí a aktuální část. Hlavní rozdíl oproti LZ-77 je v podobě výstupu, který je reprezentován buď symbolem, nebo dvojicí <*i,j*>, kde *i* nám udává pozici prvního symbolu ve slovníku (prohlížecím okně) a *j* nám představuje počet shodných symbolů. Oproti LZ-77 se neshodný symbol nezapisuje.

Pokud se nám nevyplatí kódování symbolu, zapisuje se na výstup přímo. Aby bylo možné poté data zpět dekomprimovat, musíme poznat, zda se jedná o symbol, či odkaz do slovníku, proto musíme uvádět před kódovaným blokem vždy identifikační bit [23]. Metodu LZSS využívá například komprimační program ARJ.

6.7 LZ-78

Tento algoritmus publikovali v roce 1978 opět Abraham Lempel a Jacob Ziv. Na rozdíl od předchozí metody zde nepoužíváme žádné posuvné okno, ale vytváříme si dynamický slovník z načtených (zakódovaných) symbolů. Velikost slovníku je omezena pouze velikostí paměti [9].

Výstupem kodéru je dvojice <*i, S*>, kde *i* nám představuje ukazatel na pozici ve slovníku a *S* symbol, který se již ve slovníku nenachází. Komprimaci začínáme s prázdným slovníkem, ze vstupu načteme symbol. Jelikož slovník je prázdný symbol, uložíme na pozici 1 další nenalezený symbol, uložíme na pozici 2 atd. Symboly samozřejmě zakódované zapisujeme do výstupního proudu. Nyní načteme symbol b, prohledáváme slovník, pokud není nalezen, uložíme symbol do slovníku na pozici 3 a na výstup zapíšeme <0, b>. Jiná situace nastane, pokud symbol b je ve slovníku nalezen, například na pozici 2. Poté načteme další symbol ze vstupního proudu, kterým bude například p, opět prohledáme slovník a hledáme, zda neobsahuje záznam s hodnotou bp. Hodnotu bp nenalezneme, a tak na novou pozici – v našem případě konkrétně tři, uložíme danou hodnotu bp a na výstup zapíšeme <2,p>. Příklad vytváření slovníku je uveden v tabulce 4. Budeme kódovat vstupní proud obsahující následující symboly: emamamaso

Tabulka 4: Příklad vytváření slovníku u metody LZ-78

Slovník		Výstup
Pozice	Hodnota	
1	e	<0,“e“>
2	m	<0,“m“>
3	a	<0,“a“>
4	ma	<2,“a“>
5	mas	<4,“s“>
6	o	<0,“o“>

Dekomprimace probíhá analogicky, znovu vytváříme, respektive rekonstruujeme slovník, načteme dvojici dat, podíváme se na pozici ve slovníku, na výstup zapíšeme její hodnotu plus druhou položku z načtené dvojice (tedy nenalezený symbol při komprimaci). Nakonec do slovníku vložíme novou položku, jejíž hodnota bude

reprezentována námi zapsanými daty na výstup (tedy hodnotou, na kterou ukazuje první z dvojice plus symbol S).

6.7.1 Datová struktura pro interpretaci slovníku

Jak jsme již uvedli, komprimaci začínáme s prázdným slovníkem, ze začátku ukládáme krátké fráze, ale čím vetší je datová velikost souboru (samozřejmě také záleží na druhu dat), fráze ve slovníku jsou delší, tím slovník zabírá více místa v paměti a také čím více položek je ve slovníku uloženo, tím je jeho procházení časově náročnější a prodlužuje se tak doba kódování jednotlivých symbolů. K eliminaci těchto problémů je vhodné použít pro ukládání slovníku datovou strukturu k-cestný strom.

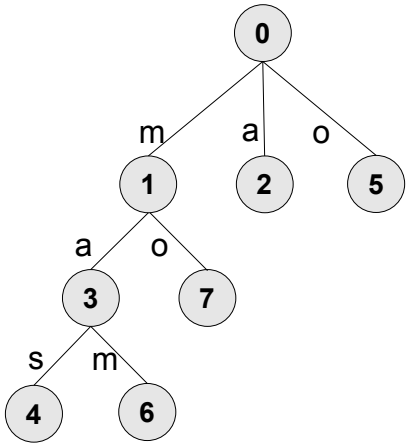
Hrany k-cestného stromu představují námi načtené symboly a listy jsou ohodnocovány vzestupnou číselnou řadou (1 až n). Komprimaci začínáme s prázdným stromem, čteme symboly ze vstupního proudu a procházíme stromem do první neshody, poté neshodný prvek uložíme jako nového syna, jehož otec je poslední shodný symbol. Na výstup zapíše klasickou dvojici <*i, S*>, kde *i* představuje hodnotu posledního načteného prvku stromu – tedy otce, u jehož synů nebyla nalezena shoda s načteným symbolem a *S* je symbol, který nebyl nalezen a byl přidán jako nový syn k prvku, jehož hodnota je *i*. Poté načteme další symbol a znovu procházíme strom od kořene. Pokud připustíme, že kódované symboly načítané ze vstupního proudu, mají velikost 8 bitů, pak každý uzel stromu (otec), může mít až 256 synů. Příklad vytváření a reprezentace slovníku pomocí k-cestného stromu a zakódování vstupního proudu dat je uveden na obrázku 6.

Dekódování je opět jednoduché, není nutné ani sestavovat strom. Strom jsme zvolili hlavně kvůli kódování, jelikož je rychlejší jeho procházení a tím hledání shodných symbolů. Při dekomprimaci je ale rychlejší (i když ne o tolik) používat tabulku, jak jsme již uvedli výše. Načteme dvojici, z uvedeného indexu načteme hodnotu, přidáme k ní druhou hodnotu z dvojice (symbol) a to celé zapíšeme na výstup a jako novou položku v tabulce. V případě stromu musíme hledat prvek, který má dané ohodnocení a musíme si pamatovat ohodnocení hran, které vedou k tomuto symbolu od kořene.

Vstupní proud obsahuje následující symboly (velikost symbolu = 1znak)

Vstupní proud: mamamasomammoma => velikost proudu 15 symbolů

abeceda: a,m,o,s



Výstupní proud: <0, m>, <0, a>, <1, a>, <3, s>, <0, o>, <3, m>, <1, o>, <1, a>

Obrázek 6: Příklad kódování pomocí metody LZ-78

Teoreticky velikost slovníku, respektive datové struktury, ve které je uložen, je omezena pouze velikostí dostupné paměti. Musíme tedy ošetřit případ, kdy nám paměť dojde. Možnosti jsou dvě a to takové, že buď smažeme celý doposud vybudovaný slovník a začneme vytvářet nový, nebo slovník zmrazíme a už jej nebudeme rozšiřovat o nové fráze, které v něm nejsou ještě uvedeny. Velikost paměti ale není jediným limitujícím faktorem, dalším je maximální počet prvků stromu. Jelikož v každém prvku je uložena indexová hodnota, je limitujícím faktorem velikost datové proměnné, do které je ukládána indexová hodnota. Pokud tedy použijeme indexovou hodnotu o velikosti 16bitů, strom může obsahovat maximálně 2^{16} prvků. Dosažením maximální hodnoty této proměnné nám představuje obdobný případ, jako kdyby nám došla paměť. Jelikož indexová hodnota představuje část, která se ukládá do výstupního proudu, musíme její rozsah zvolit rozumně, neboť nám ovlivňuje komprimační poměr. Existují také různé optimalizace, například kódování začínáme s malou datovou velikostí indexové hodnoty a poté co se slovník rozšíří natolik, že nám již datový rozsah indexové proměnné nestačí, přiřadíme indexové proměnné větší datovou velikost. Aby byla možná správná dekomprimace, musíme tyto stavy detekovat a opět měnit správně datovou velikost indexové proměnné.

V praxi se tato metoda moc nepoužívá,většinou je použito jiných modifikací slovníkových metod, například LZW.

6.8 LZW

Metodu LZW (Lempel-Ziv-Welch) publikoval v roce 1984 Terry Welch, jedná se o vylepšení metody LZ78. Na výstup zapisujeme pouze číselné označení daného listu, což je hlavní rozdíl oproti LZ-78. Před začátkem komprimace musíme inicializovat slovník, který bude obsahovat celou vstupní abecedu. Tímto je zajištěno, že každý symbol bude ve slovníku nalezen a odpadá nám tím zapisování nenalezeného symbolu na výstup, jak tomu je v případě algoritmu LZ-78 [9].

Postup komprimace je dále obdobný jako v předchozím případě, jen s malými obměnami. Jak již bylo řečeno, slovník musí být zpočátku inicializovaný, pokud velikost vstupního symbolu je 8 bitů a budeme uvažovat opět pro reprezentaci slovníku datovou strukturu k-cestný strom, kořen stromu bude mít na počátku komprimace 256 synů.

Při samotné komprimaci načteme symbol ze vstupního proudu dat a prohledáme slovník, zda se v něm nachází načtený symbol – pokud ano, načteme další symbol a prohledáváme následující symboly (v rámci struktury k-cestný strom, jsou to synové nalezeného symbolu), pokud symbol není nalezen, zapíše na výstup číselnou hodnotu posledního nalezeného symbolu. Poté prohledáváme strom znovu od začátku, od nenalezeného symbolu. Průběh komprimace můžeme vidět na obrázku 7.

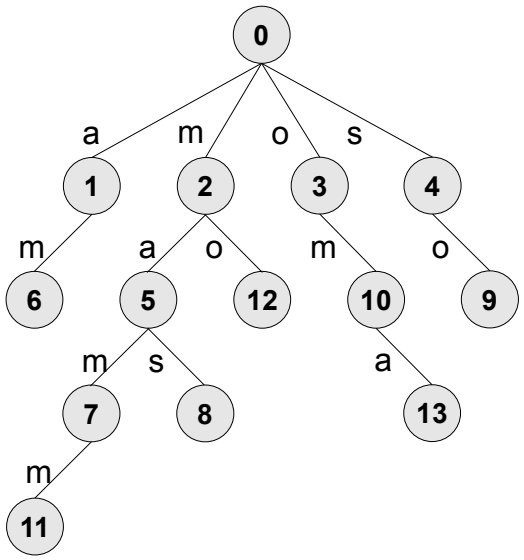
Pro dekomprimaci stačí pouze zkomprimovaný řetězec, slovník se vytváří opět postupně během dekomprimace. Na vstupu se může objevit i odkaz na frázi, která ještě ve slovníku není. Tuto frázi nám představuje poslední dekódovaná fráze a její první symbol. Takto novou frázi přidáme do slovníku a zapíšeme do výstupního proudu.

Metodu LZW využívají ke komprimaci grafické formáty GIF a TIFF, komprimační program WinZip, nebo dokumenty PS a PDF [18].

Vstupní proud obsahuje následující symboly (velikost symbolu = 1znak)

Vstupní proud: mamamasomammoma => velikost proudu 15 symbolů

abeceda: a,m,o,s



Výstupní proud: 2, 1, 5, 5, 4, 3, 7, 2, 10, 1

Obrázek 7: Příklad kódování pomocí metody LZW

6.9 Ostatní slovníkové metody

Existuje mnoho slovníkových metod, které více, či méně vycházejí z výše uvedených, respektive z metod LZ-77 a LZ-78, které můžeme považovat za základní. Příkladem může být metoda LZB, což je vlastně LZSS s úspornějším kódováním ukazatelů (i,j), kde i je binární kód rostoucí délky (nejprve 1 bit, po načtení 2. znaku dva bity, po načtení 4. znaku 3 bity atd.). LZH kombinuje metodu LZSS a Huffmanovo kódování. LZC je modifikací metody LZW, hlavní obměna je v kódování ukazatelů kódem se vzrůstající délkou. U metody LZMW se nová fráze získá zřetězením dvou posledních. Popis mnoha slovníkových metod nalezneme v [9], [30].

6.10 Metoda PPM

Metoda PPM (Prediction by Partial Matching) byla vynalezena Johnem Clearym a Ianem Wittenem v roce 1984. Jedná se o adaptivní kontextovou metodu (v některých zdrojích se uvádí, že se jedná o statistickou metodu), která je založená na kontextovém modelování a predikci. To znamená, že využívá předcházející symboly v nekomprimovaných datech k predikci symbolu následujícího [24], [25].

Interně je při komprimaci budován strom kontextů. Pro každý nový symbol je vždy od nejdelšího kontextu otestováno, zda-li se příchozí symbol v tomto kontextu již nevyskytl, pokud ne, přejde se o kontextovou úroveň níže, pokud ano, je s využitím aritmetického kódování tato informace uložena. Více o aplikaci metody PMM a jejích modifikací můžeme nalézt v [15].

V některých publikací se uvádí, že metoda PPM patří mezi nejlepší metody ke komprimaci textu [26]. Využívá ji mnoho komprimačních programů například 7Zip, WinRAR, FreeARC [18].

6.11 Burrows-Wheelerova transformace

Nejedná se o komprimační metodu v pravém slova smyslu, při aplikaci Burrows-Wheelerovi transformace na vstupní proud dat se data nezkomprimují ani o bit, právě naopak, ještě se o nějaký byte zvětší. Burrows-Wheelerova transformace mění pouze pořadí symbolů, díky tomu může poté komprimační algoritmus dosáhnout lepšího komprimačního poměru.

Burrows-Wheelerova transformace pracuje v blokovém režimu. Čteme ze vstupního proudu dat bloky symbolů o předem definované velikosti a každý zpracováváme samostatně.

Nejdříve provedeme všechny cyklické rotace vstupního bloku. Výsledné fráze poté lexikograficky setřídíme a každý řádek očíslujeme. Vznikla nám vlastně matice $n*n$, kde n je velikost vstupního bloku symbolů. V matici nalezneme řádek, ve kterém se nachází originální fráze a číslo řádku si zapamatujeme, respektive ho zapíšeme do výstupního proudu dat. Dále ze všech seřazených frází vybereme postupně poslední symbol, tedy poslední sloupec matice a ten zapíšeme do výstupního proudu dat.

Dekódování probíhá následovně – ze vstupního proudu dat přečteme číslo udávající nám číslo řádku, kde se vyskytuje originální fráze. Poté sestavíme poslední sloupec matice, který nám udává vstupní proud. Načtené symboly setřídíme opět vzestupně a tím nám vznikne první sloupec matice. Následně každému symbolu v posledním sloupci přiřadíme číselnou hodnotu, která odpovídá číslu řádku symbolu v prvním sloupci matice. Dekódování zprávy začínáme na čísle řádku, který nám představuje originální zprávu. Číslo, které je zapsáno v posledním sloupci, nám

udává první hodnotu symbolu. Hodnotu zapíšeme na výstup a pokračujeme v přečtení další číselné hodny z posledního sloupce, který se nalézá na řádku námi dekódovaného symbolu. Zpráva se dekóduje odzadu. Průběh transformace můžeme vidět na obrázku 8.

Kódování: vstupní proud dat: emamamaso

1. přehled všech frází po cyklické rotaci:

0. emamamaso
1. mamamasoe
2. amamasoem
3. mamasoema
4. amasoemam
5. masoemama
6. asoemamam
7. soemamama
8. oemamamas

2. lexikograficky seřazené:

0. amamasoem
1. amasoemam
2. asoemamam
3. emamamaso
4. mamamasoe
5. mamasoema
6. masoemama
7. oemamamas
8. soemamama

Výstupní proud dat: 3mmmoeaasa

Dekódování: vstupní proud dat: 3mmmoeaasa

0. a m 4.
1. a m 5.
2. a m 6.
3. e o 7.
4. m e 3.
5. m a 0.
6. m a 1.
7. o s 8.
8. s a 2.

Výstupní proud dat: emamamaso

Obrázek 8: Příklad kódování pomocí Burrows-Wheelerova transformace

Burrows-Wheelerova transformace má hlavní využití zejména při zpracování obrázků, hudby a textu. Po Burrows-Wheelerova transformaci vetšinou následuje zakódování algoritmem Move to front, podrobně je popsán například v [27] a následně jsou data zkomprimovaná nějakou statistickou metodou – nejčastěji Huffmanovým kódováním. Burrows-Wheelerovu transformaci využívá například komprimační program b2zip.

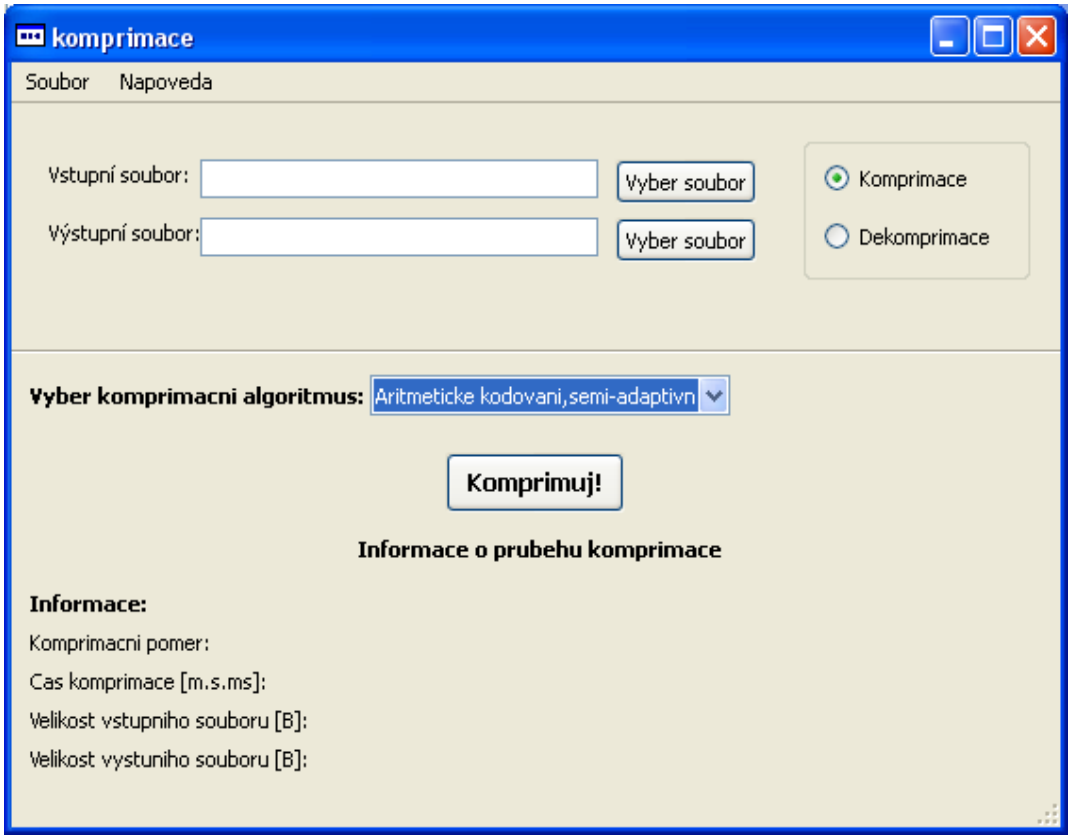
7 Implementace a popis testovací aplikace

Komprimačních algoritmů existuje nepřeberné množství, každý algoritmus má mnohé modifikace a vylepšení, jejichž hlavním cílem je dosáhnout lepšího komprimačního poměru a kratší doby komprimace. Pro implementaci jsme vybrali komprimační algoritmy, které jsou v dnešní době nejpoužívanější, respektive jejich základní verze, tak jak byly představeny svými objeviteli.

7.1 Implementace a popis programu

Pro implementaci jednotlivých algoritmů jsem zvolil programovací jazyk C++. Celý projekt byl vyvíjen ve vývojovém prostředí MS Visual Studio 2005 s integrovanou knihovou QT-4.3. Grafické uživatelské rozhraní aplikace bylo navrženo právě pomocí knihovny QT-4.3. Jelikož je knihovna QT multiplatformní, celý program by měl jít bez problémů zkompilovat i pod operačním systémem Linux. Každý komprimační algoritmus je implementovaný v samostatné třídě, tak aby bylo možno ho použít například v jiné aplikaci, případně mohl být jednoduše modifikován a upravován, aniž by to ovlivnilo celou aplikaci.

Program byl navržen pouze pro testování jednotlivých algoritmů, proto disponuje pouze základními funkcemi. Ovládání programu je velice intuitivní. Program spustíme otevřením souboru *komprimace.exe*. Vzhled aplikace můžeme vidět na obrázku 9. Vybereme vstupní a výstupní soubor, kde vstupním souborem je myšlen soubor, který chceme komprimovat, respektive dekomprimovat. Přes radio butom zvolíme, zda chceme komprimovat či dekomprimovat, a vybereme druh algoritmu. Druh algoritmu volíme pouze v případě komprimace, pokud zvolíme volbu dekomprimace, druh algoritmu se vybere automaticky podle typu souboru, pokud byl zkomprimován tímto programem, jinak program nahlásí chybu. Pod tlačítkem komprimuj – respektive dekomprimuj, se nám zobrazuje informace o průběhu komprimace, dekomprimace. V dolní části programu jsou poté vyobrazeny informace o velikosti vstupního, výstupního souboru, času komprimace – respektive dekomprimace a komprimační poměr.



Obrázek 9: Vzhled testovací aplikace

Ke každému komprimovanému souboru je na začátek zapsána jednoduchá hlavička tak, aby poté byla možná automatická dekomprimace a aplikace poznala, že se jedná o soubor, který vytvořila. Hlavička má velikost tři bajty, první dva bajty představují identifikátor, který slouží k tomu, aby aplikace poznala, že se jedná o její soubor (použity znaky JH). Třetí bajt udává druh algoritmu, jakým byl soubor zkomprimován, ten slouží pro automatickou dekomprimaci. Jelikož hlavička představuje nadbytečnou informaci v souboru, není započítávaná do jeho velikosti při výpočtu komprimačního poměru.

7.2 Přehled implementovaných algoritmů jejich datových struktur a teoretického komprimačního poměru

Každý algoritmus má svůj teoretický maximální komprimační poměr, tj. poměr, jakého může maximálně dosáhnout, pokud vstupní proud bude obsahovat pouze ideální sekvenci symbolů, ta se liší pro každý druh algoritmu.

7.2.1 RLE kódování

Implementována byla modifikace metody proudového kódování – tvorba paketu na bytové úrovni, která je popsaná v kapitole 6.1. U implementace algoritmu RLE nebyla použita žádná speciální datová struktura. RLE paket je představován třemi proměnnými typu unsigned char, které se postupně zapisují do výstupního proudu.

Maximální možný komprimační poměr, jaký lze v mé implementaci algoritmu RLE dosáhnout je: $Komprimační\ poměr_{max} = \frac{3}{256} = 0{,}011719$. Vycházíme z toho, že pro zakódování proudu symbolů potřebujeme 3 bajty. Velikost proměnné pro uložení počtu opakování je jeden bajt, z toho vyplývá, že maximální možný počet opakujících se znaků, které jsme schopny zakódovat do jednoho RLE paketu je 256. Maximálního komprimačního poměru dosáhneme za podmínky, pokud vstupní proud obsahuje na 8 bitech (omezení velikosti proměnné sloužící pro uložení proudové hodnoty) vždy shodné symboly tzn. proud 256 shodných symbolů.

Existují různé obměny – například vycházíme z toho, že proud opakujících se znaků menší jak 4 nekódujeme a zapisujeme do výstupního proudu v nezměněné podobě, tak může hodnota 0 opakování v RLE paketu představovat počet opakování 4 symbolů. Takto bychom byli schopni zakódovat proud stejných symbolů až o velikosti 259. Tuto modifikaci jsem nevyužil z důvodu, že tato varianta prochází vstupní proud pouze jednou, tedy má pevně zvolený identifikátor RLE paketu. Tímto nám odpadá jedno procházení vstupního proudu a hledání neobsazeného symbolu, který by byl poté použit jako identifikátor. Identifikátor se samozřejmě ve vstupním proudu může objevit jako vstupní symbol a pokud by byl zakódován normálně, nastaly by při dekódování problémy. Toto je vyřešeno následovně – pokud ve vstupním proudu narazíme na symbol, který má stejnou hodnotu jako identifikátor, tento symbol zakódujeme jako RLE paket s hodnotou opakovaní 0. Při dekomprimaci poté nejdříve přečteme identifikátor a víme, že nám za ním následuje počet opakování – v tomto případě 0 a poté přečteme zakódovaný symbol. Nepatrné zhoršení komprimačního poměru nám ale kompenzuje menší časová náročnost, jelikož není nutné procházet vstupní soubor dvakrát, kde bychom v prvním průchodu hledali neobsažený symbol, který by nám představoval identifikátor, čas komprimace se tedy zkrátí na polovinu, jelikož časová náročnost je *O(n)*.

7.2.2 Huffmanovo kódování a Sahannon-Fanovo kódování

U Huffmanova kódování byla použita datová struktura prioritní fronta, která je využívána při sestavování binárního stromu. K její implementaci bylo využito datového kontejneru ze standardní knihovny šablon jazyka C++ (STL) a to konkrétně *priority_queue*, řazení prvků ve frontě (tzn. určení prvku, který bude mít nejvyšší prioritu) je zabezpečeno pomocí funkčního objektu. Další datová struktura využívaná Huffmanovým kódováním je binární strom, který slouží k sestavení prefixového kódu. Datová struktura binární strom byla mnou naprogramovaná. Jedná se o abstraktní datovou strukturu, která se nalézá v samostatné třídě, její využití tedy není svázané pouze s tímto algoritmem, nebo s konkrétním datovým typem. Celý algoritmus je podrobněji popsán v teoretické části v kapitole 6.2.

Jediný podstatný rozdíl mezi Huffmanovým a Sahannon-Fanovým kódováním je v sestavování stromu. Bylo tedy využito dědičnosti, kde třída v níž je implementováno Sahannon-Fanovo kódování dědí od třídy s implementací Huffmanova kódování. Metody, které třída s implementací Sahannon-Fanova kódování využívá od svého předka slouží: k analýze souboru a ke komprimaci vstupního proudu dat. Předefinovány byly metody sloužící k sestavení stromu a k dekomprimaci proudu dat. K uložení binárního stromu je opět použito stejné datové struktury jako v případě Huffmanova kódování. Algoritmus byl implementován přesně tak jak je popsáno v kapitole 6.3.

Maximálního komprimačního poměru lze dosáhnout pouze tehdy, pokud je vstupní abeceda reprezentována jedním nebo dvěma symboly, tzn. vstupní proud obsahuje pouze jeden maximálně dva druhy symbolů. Hodnota maximálního komprimačního poměru je $Komprimační\ poměr_{max} = \frac{1}{8} = 0{,}125$. Do maximálního poměru není započítána velikost modelu, jelikož obě metody jsou semi-adaptivní, tohoto komprimačního poměru nikdy nedosáhneme, ale pokud vstupní proud dat je dostatečně velký, velice se k němu přiblížíme.

7.2.3 Aritmetické kódování

Implementována byla jak semi-adaptivní verze aritmetického kódování, tak adaptivní verze. Rozdíl mezi oběma je minimální, pouze semi-adaptivní musí procházet vstupní soubor dvakrát a ukládat do výstupního souboru informace o modelu. Není

použita žádná složitější datová struktura, pouze pole, které obsahuje prvky datového typu struktura (obsahuje symbol, četnost výskytu a dolní interval). Celé pole je poté setříděno podle četnosti výskytu pomocí algoritmu Quicksort. Obě verze aritmetického kódování byli implementovány samostatně, každá ve své třídě. Podrobnější popis algoritmu nalezneme v kapitole 6.4.

Maximální komprimační poměr se teoreticky libovolně může přiblížit nule, protože se hodnota komprimačního poměru odvíjí od délky zprávy.

7.2.4 Slovníková metoda LZ-77

Slovníková metoda LZ-77 využívá datovou strukturu oboustranná fronta, která představuje prohlížecí okno. Jako oboustranná fronta byl použit datový kontejner *deque* ze standardní knihovny šablon jazyka C++ (STL). Algoritmus byl implementován tak jak je popsáno v kapitole 6.4.

Pokud uvažujeme velikost aktuálního okna 8 bitů, tj. 256 symbolů, jsme schopni zakódovat maximální velikost kódové části 255 symbolů plus jeden symbol nenalezený, nebo poslední symbol v aktuálním okně. Dále pokud bude velikost prohlížecího okna 4096 symbolů, potřebujeme pro indexaci pozice ve slovníku proměnnou o velikosti 12 bitů. Ve své implementaci jsem použil proměnnou o velikosti 16 bitů, tu tedy budeme uvažovat pro výpočet maximálního komprimačního poměru. Velikost symbolu bude 1 bajt a velikost proměnné udávající délku předpony bude také jeden bajt. Pro zakódování jedné sekvence tedy potřebujeme 4 bajty. Maximální komprimační poměr tedy vypočítáme ze vztahu:

$$Komprimační\ poměr_{max} = \frac{4}{256} = 0{,}015625$$

Pokud začínáme komprimaci s prázdným slovníkem, komprimační poměr se nepatrně zhorší, v praxi při ověřování maximálního komprimačního poměru jsem dosáhl hodnoty 0,1572.

7.2.5 Slovníkové metody LZ-78 a LZW

Obě metody využívají pro reprezentaci slovníku datové struktury k-cestný storm, tak jak je popsáno v kapitole 6.7, respektive 6.8. Pro reprezentaci datové struktury k- cestný strom bylo použito datového kontejneru *vector* ze standardní knihovny šablon jazyka C++ (STL).

Obě metody se liší pouze ve výstupním proudu a práci se slovníkem. Maximální komprimační poměr teoreticky může být nekonečně malý, protože je determinován pouze velikostí vstupního proudu dat.

8 Testování jednotlivých algoritmů

Pro otestování výkonnosti jednotlivých komprimačních algoritmů, existují různé standardizované testovací korpusy, což jsou vlastně normované archívy, které obsahují různé typy souborů – příkladem může být Artificial korpus, Calgary korpus, nebo Canterbury korpus. Data, jaká obsahuje Canterbury korpus, můžeme vidět v tabulce 5. Přehled dalších korpusů včetně jejich popisu a obsahu souborů, můžeme nalézt v [28].

Tabulka 5: Obsah Canterbury korpusu, zdroj: [28]

název souboru	typ souboru	obsah	velikost [B]
alice29.txt	text	anglický text	152089
asyoulik.txt	play	Shakespeare	125179
cp.html	html	HTML kód	24603
fields.c	Csrc	zdrojový kód C	11150
grammar.lsp	list	zdrojový kód LISP	3721
kennedy.xls	Excl	dokument Excel	1029744
lcet10.txt	tech	technický text	426754
plrabn12.txt	poem	poezie	481861
ptt5	fax	CCITT test set	513216
sum	SPRC	SPARC spustitelný soubor	38240
xargs.1	man	GNU manuálová stránka	4227

Pro otestování výkonu komprimačních algoritmů byly zvoleny některé soubory z Canterbury korpusu, v tabulce 5 jsou řádky zvýrazněny světle šedivou barvou. Tyto soubory doplňují námi vybrané soubory, jejich seznam zobrazuje tabulka 7.

Veškeré testování probíhalo na notebooku Asus A-6M. Jeho základní hardwarové parametry nám udává tabulka 6.

Tabulka 6: Parametry PC použitého pro testování

CPU	AMD Sempron 3400+
Operační paměť	1024MB/667MHz
HDD	100GB, 5400 ot/min.
OS	MS Windows XP, SP2

Tabulka 7: Přehled vlastních testovacích souborů

název souboru	typ souboru	obsah	Velikost [B]
text.txt	textový	český neformátovaný text v rozsahu 2 A4	3894
word-old.doc	doc MS Word 2003	český formátovaný text v rozsahu 2 A4.	31744
word-new.docx	docx MS Word 2007	český formátovaný text v rozsahu 2 A4.	14506
Obrazek1.bmp	obrázek	nekomprimovaný obrazový soubor barevná hloubka 24bit, rozlišení 960x768	2211894
Obrazek2.bmp	obrázek	nekomprimovaný obrazový soubor barevná hloubka 256 barev, rozlišení 960x768	738358
Obrazek3.bmp	obrázek	nekomprimovaný obrazový soubor 16 stupňů šedi barev, rozlišení 960x768	368758
Obrazek4.jpg	obrázek	komprimovaný obrázový soubor JPGE komprimací kvalita 90%, barevná hloubka 24bit, rozlišení 960x768	162119
video.avi	video	5s nekomprimovaného videa v rozlišení 320x240 pix.	35573444

8.1 Přehled vybraných komerčních programů

Přehled komerčních programů, které jsem si vybral pro otestování s implementovanými komprimačními algoritmy je uveden v tabulce 8. Všechny programy byly pro komprimaci ponechány v defaultním nastavení. Přehled algoritmů, jaké komerční komprimační programy využívají, můžeme vidět v tabulce 8. Popis jednotlivých programů byl převzat z [29].

Tabulka 8: Přehled algoritmů využívaných v komerčních komprimačních programech, [18]

Program	algoritmy
7-Zip	f, BWT, LZMA, PPMII, LZ77,
bzip2	BWT, MTF, Huff
gzip	DEFLATE
WinZip	LZH, LZW, SF, Huff, PPMd
WinRAR	f, LZ77, PPMII, Huff
Legenda: f – různé speciální transformace BWT – Burrows-Wheelerova transformace Huff – Huffmanovo kódování SF – Shannon-Fanovo kódování MTF – Move to front	

8.1.1 Bzip2

Svobodný komprimační algoritmus a program vyvinutý Julianem Sewardem. První verzi (0.15) vydal v červenci 1996. Stabilita a popularita programu od té doby značně vzrostla. Verze 1.0 byla vydána na konci roku 2000. Bzip2 používá Burrows-Wheelerovu transformaci, která konvertuje často se opakující znakové sekvence do řetězců ze stejných písmen a poté použije Move-to-front transformaci a nakonec Huffmanovo kódování. Původně používal předek bzip2, bzip aritmetické kódování, které má lepší komprimační poměr. To ale muselo být vyměněno za méně výkonné Huffmanovo kódování, protože aritmetické kódování je patentováno. Pro testování byla použita verze programu bzip2 verze 1.0.5.

8.1.2 Gzip

Gzip je zkratka pro GNU zip; program je free software vyvíjený pod projektem GNU. Gzip je založený na algoritmu DEFLATE, který je kombinací LZ77 a Huffmanova kódování. Podobně jako bzip2 i gzip je normálně používán ke komprimaci jednotlivých souborů. Pro testování byla použita verze programu gzip 1.3.12.

8.1.3 WinRAR

Je populární proprietární souborový formát pro komprimaci dat a archivaci vyvinutý ruským programátorem Jevgenijem Rošalem (odtud pojmenování RAR: Roshal ARchive). Autor WinRARu dal k dispozici zdrojové kódy dekomprimačního programu, díky tomu je možné RAR dekomprimovat i v jiných programech na různých platformách. RAR na rozdíl od předchozích formátů je šířen jako shareware. To znamená, že po uplynutí zkušební lhůty musíme program buď zaplatit nebo odinstalovat. Pro testování byla použita verze programu WinRAR 3.71.

8.1.4 WinZip

WinZip patří mezi nejužívanější a také nejznámější komprimační programy. Jedná se o jednu z nejlepších alternativ pro komprimaci souborů ve Windows. V současnosti je program distribuován ve dvou licenčních variantách – Standard a Professional, které se neliší jen cenou, ale také nabízenými funkcemi. Pro testování byla použita verze programu WinZip 11.2 Professional.

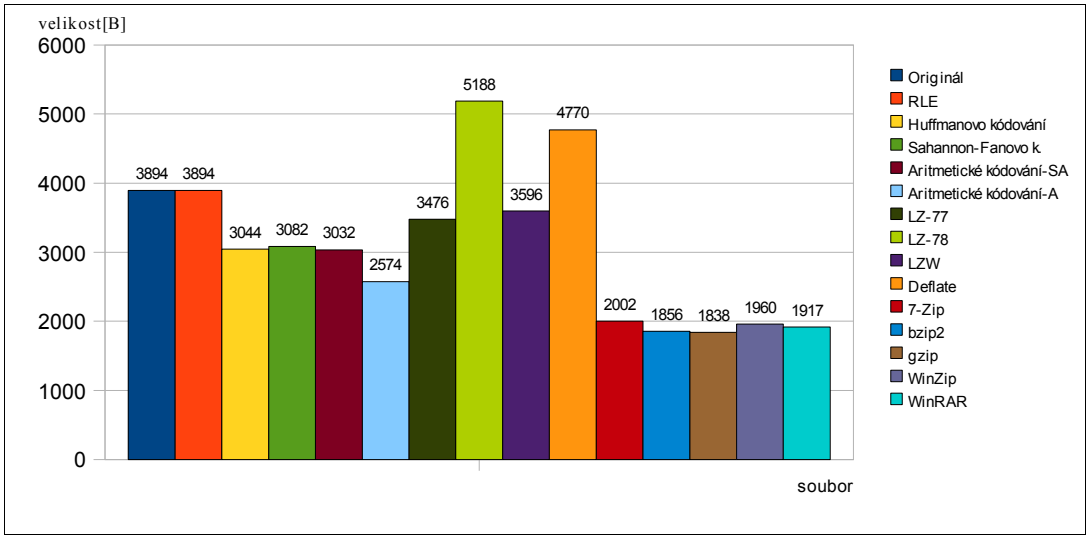
8.1.5 7-Zip

Je komprimační program určený pro různé operační systémy. 7-Zip je svobodný software, vyvíjený Igorem Pavlovem a distribuován pod licencí GNU LGPL. 7Zip používá přednostně kompresní algoritmus LZMA, nabízí však také kompresní algoritmy PPMD, bzip2, Deflate, a „store" – tedy uloženi souborů bez komprimace. Pro testování byla použita verze programu 7-Zip 4.57.

8.2 Testování komprimačního poměru

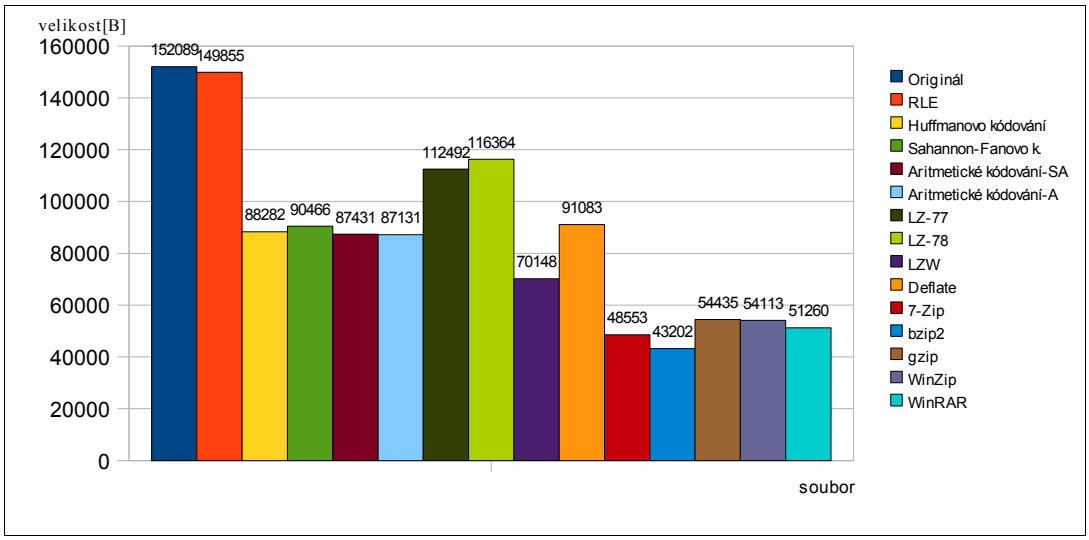
Přehled všech komprimačních poměrů dosažených jak komerčními programy, tak algoritmy, můžeme vidět v tabulce 9.

8.2.1 Soubory obsahující text

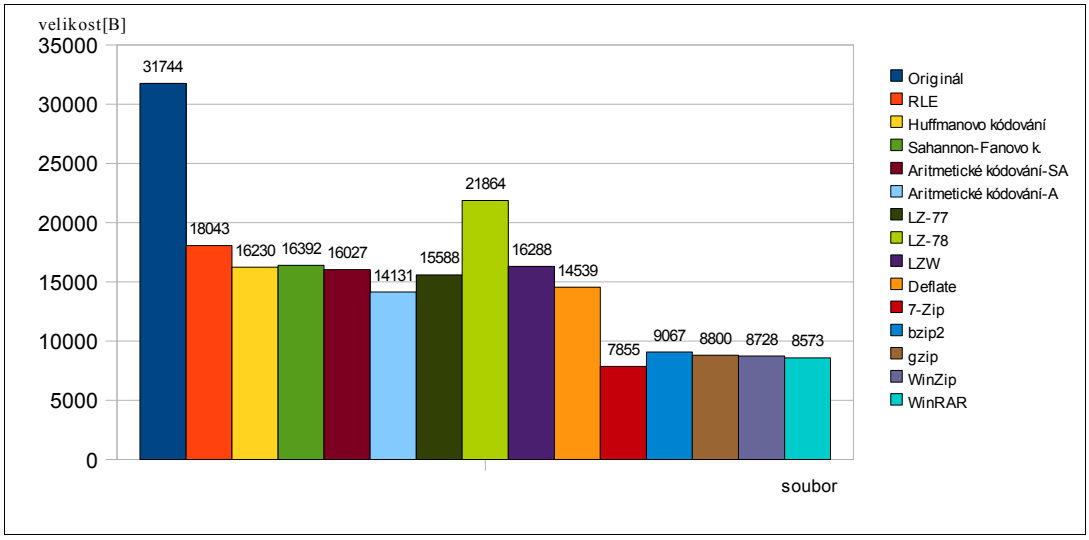


Obrázek 10: Datová velikost souboru text.txt po komprimaci

Tato podkapitola se soustřeďuje na komprimaci textu – klasického neformátovaného použitého v psaném projevu, a to jak českého, tak anglického. Dále textu ve zdrojových kódech a formátovaného textu. Při komprimaci anglického textu, kde jsou výsledky zobrazeny v obrázku 11, se obecně dosahuje lepších komprimačních poměrů, než při komprimaci textu českého, výsledky jsou vidět na obrázku 10. Je to dáno hlavně tím, že anglický text neobsahuje diakritiku, tedy vstupní abeceda je menší. Jedny z nejhorších výsledků dosahoval algoritmus LZ-78, což je způsobeno tím, že potřebuje k uložení jediného symbolu nebo fráze vždy 3 bajty a nehlídá si, zda nedosahuje záporné komprimace.



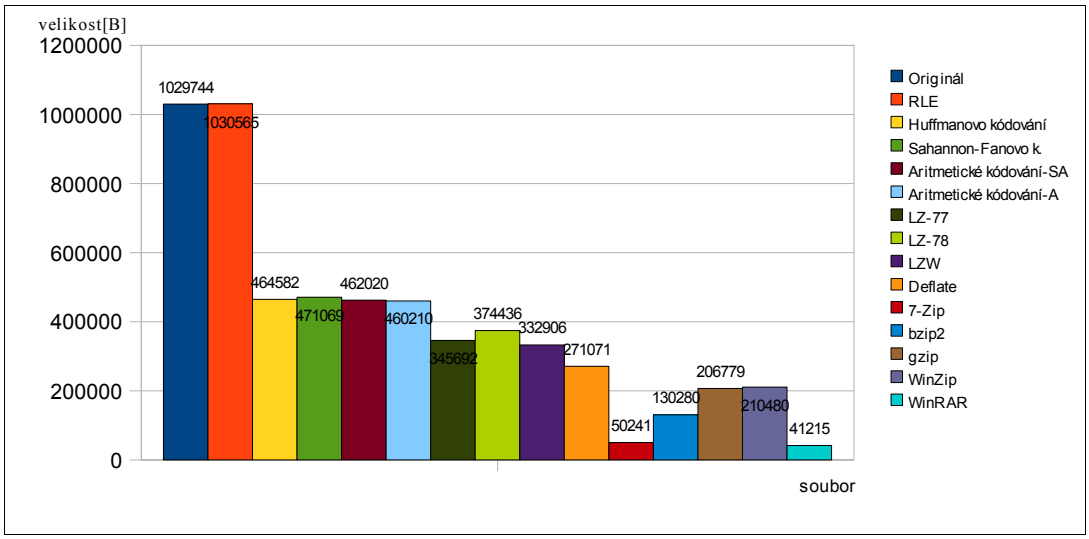
Obrázek 11: Datová velikost souboru alice29.txt po komprimaci



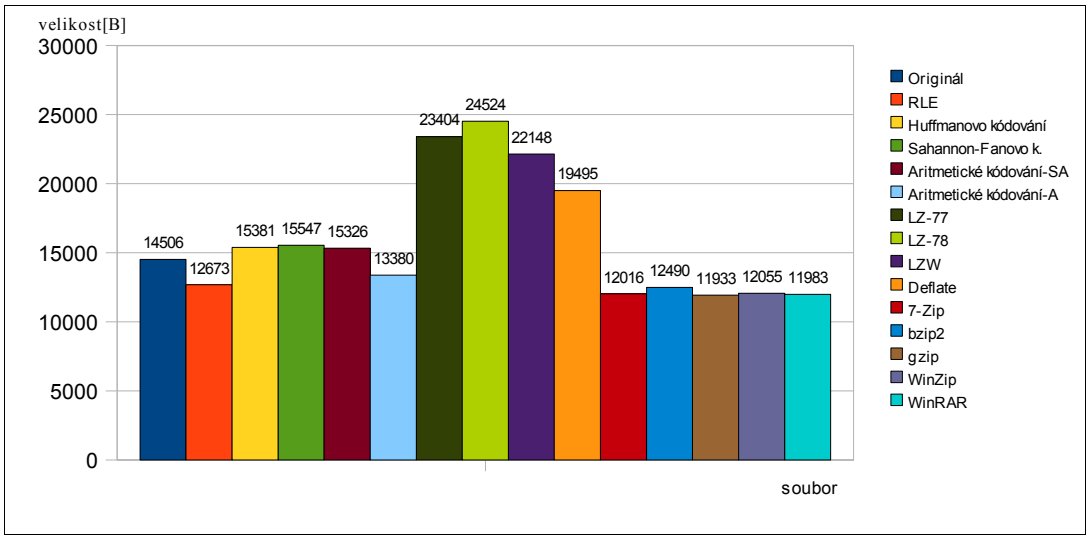
Obrázek 12: Datová velikost souboru word-old.doc po komprimaci

Za povšimnutí stojí výsledek komprimace dokumentu .docx, zobrazený na obrázku 14. Jedná se o již zkomprimovaný formát, a to konkrétně metodou Zip, proto většina algoritmů dosáhla záporné komprimace s výjimkou adaptivního aritmetického kódování a překvapivě i RLE kódování, což svědčí o faktu, že se v tomto dokumentu nachází sekvence stejných symbolů.

Pro komprimaci zdrojových souborů v jazyce C jsou výsledky vyobrazeny na obrázku 15 nebo html viz obrázek 16, dosahuje dobrých výsledků slovníkový algoritmus LZ-77, což je zřejmě dáno častým opakováním klíčových slov.

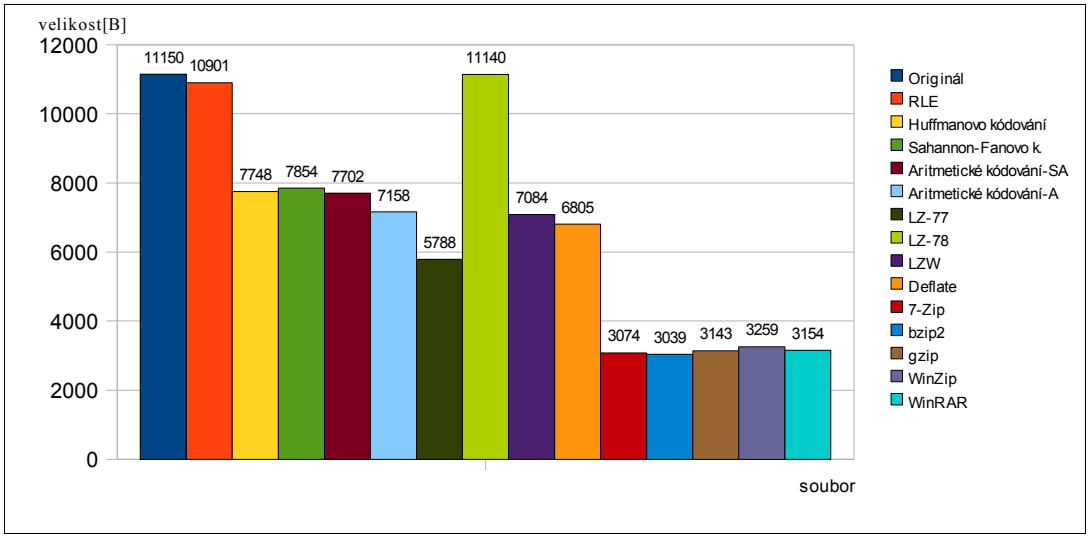


Obrázek 13: Datová velikost souboru kennedy.xls po komprimaci



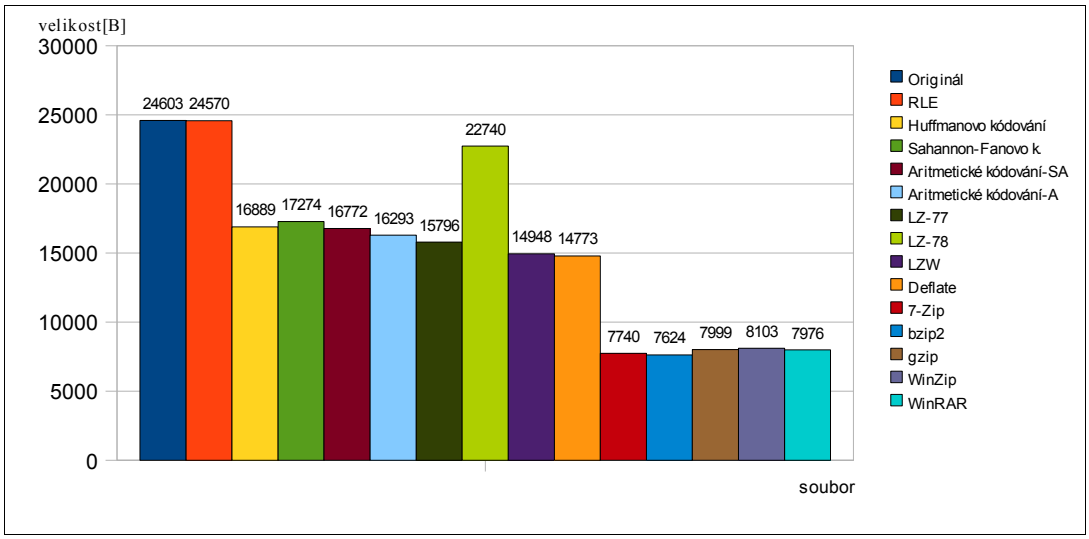
Obrázek 14: Datová velikost souboru word-new.docx po komprimaci

Komerční komprimační programy dosahují vcelku vyrovnaných výsledků mimo komprimace .xls dokumentu, viz obrázek 13. Zde programy WinRar a 7-Zip dosáhly o poznání lepších komprimačních poměrů. Oba programy používají mimo jiných komprimační algoritmus PPMII, tím bych vysvětloval dosažení výrazně nižšího komprimačního poměru oproti ostatním programům.



Obrázek 15: Datová velikost souboru fields.c po komprimaci

Algoritmy nehodící se pro komprimaci textu jsou RLE a LZ-78. Aby RLE dosáhl komprimace, musí text obsahovat sekvenci stejných symbolů, což v psaném projevu není tak obvyklé. Naopak slovníkové algoritmy LZ-77 (včetně Deflate), LZW nebo statistická metoda adaptivního aritmetického kódování dosahovaly v průměru dobrých výsledků. Samostatné algoritmy nelze ovšem srovnávat s komerčními, které ve všech souborech dosáhly lepšího komprimačního poměru, je to způsobeno tím, že využívají různé transformace a kombinace základních algoritmů.



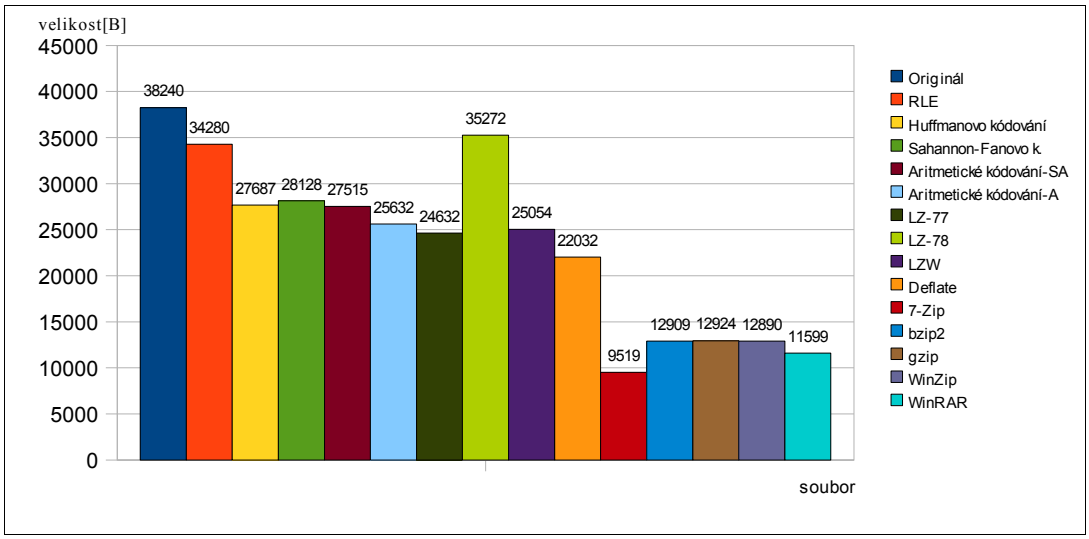
Obrázek 16: Datová velikost souboru cp.html po komprimaci

8.2.2 Binární soubory

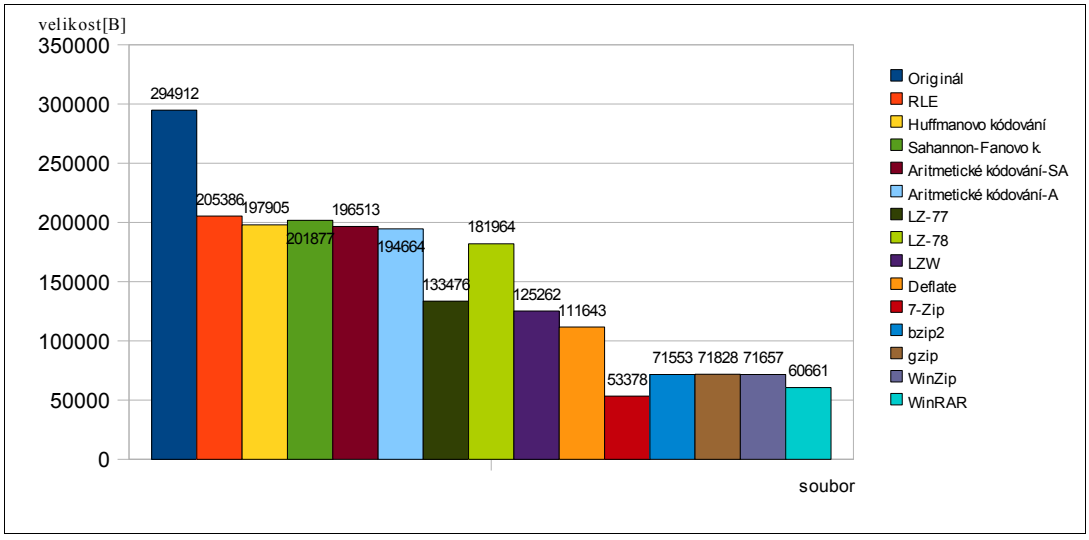
U binárních souborů viz obrázky 17 a 18 vyzvedněme algoritmus LZ-77 – respektive Deflate, který opět dosáhl jednoho z nejnižších komprimačních poměrů ná-

sledovaný LZW. Za zmínění stojí výsledek dosažený metodou RLE, která se hlavně na obrázku 18 – dosaženým komprimačním poměrem, přiblížila statistickým komprimačním algoritmům.

Komerční komprimační algoritmy byly opět vcelku vyrovnané, až na program 7Zip, který dosáhl o poznání lepšího komprimačního poměru.



Obrázek 17: Datová velikost souboru sum po komprimaci

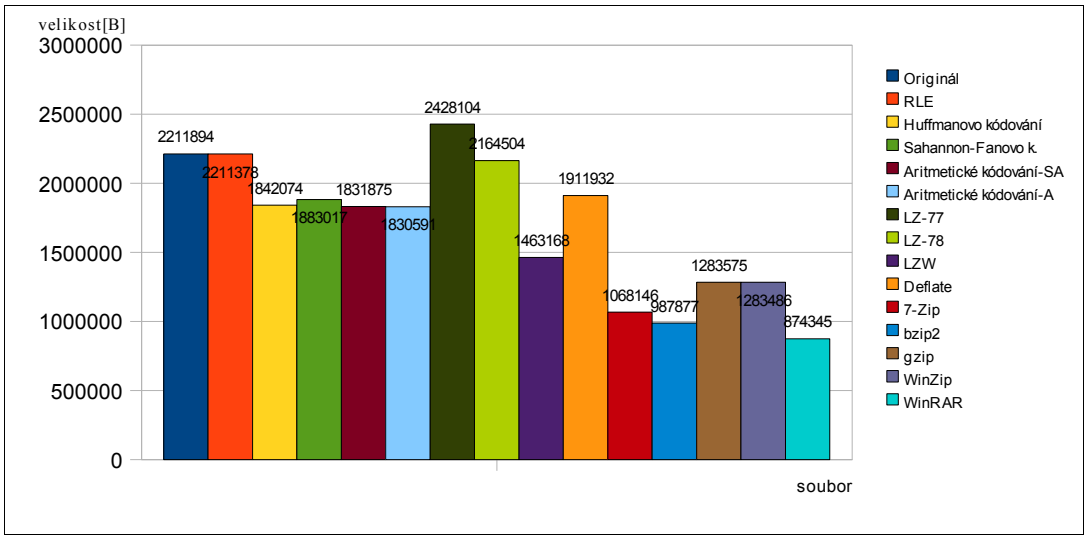


Obrázek 18: Datová velikost souboru komprimace.exe po komprimaci

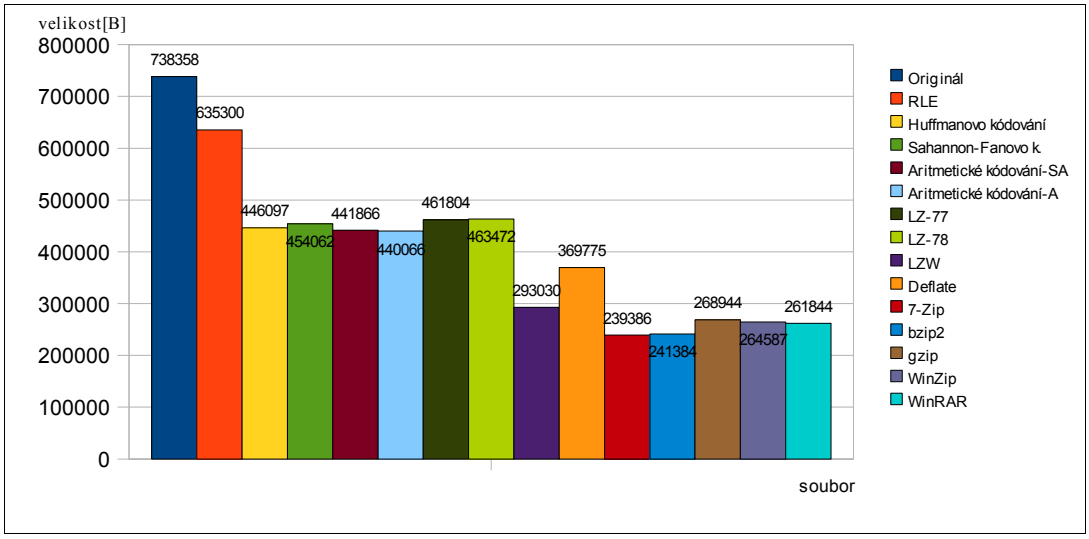
8.2.3 Obrázkové a video soubory

Motiv na obrázcích je vždy stejný, rozlišení také, liší se jen v barevné hloubce nebo formátu. Z výsledků nekomprimovaného formátu, které jsou vyobrazeny na obrázcích 19, 20 a 21 vyplývá, že se komprimační poměr odvíjí od barevné

hloubky. Nejlepších výsledků dosahuje slovníková metoda LZW, nejhorších naopak, pomineme-li metodu RLE, která dosahuje komprimace až při nižších počtu barev, dosahuje metoda LZ-77.



Obrázek 19: Datová velikost souboru Obrazek1.bmp po komprimaci

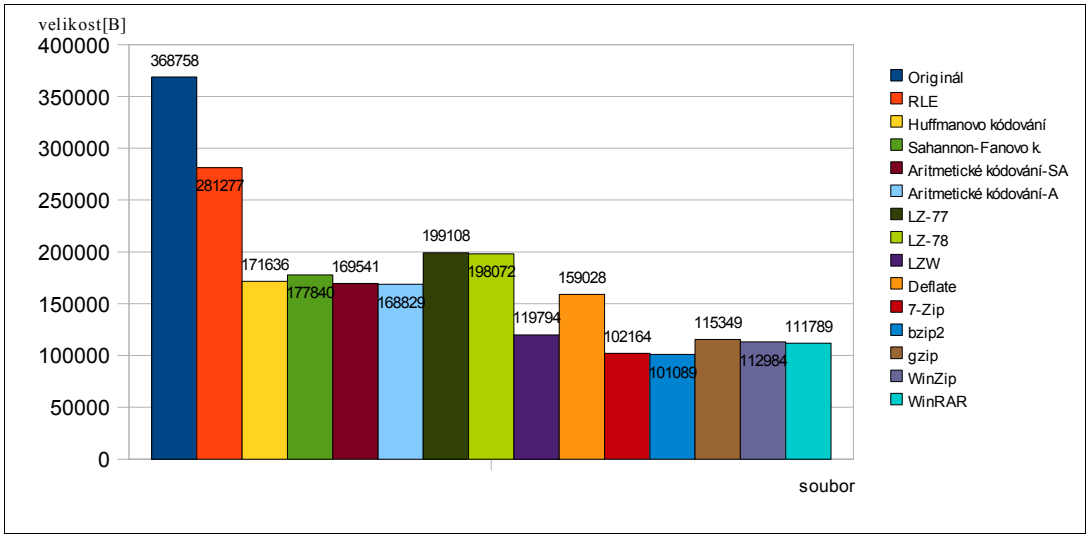


Obrázek 20: Datová velikost souboru Obrazek2.bmp po komprimaci

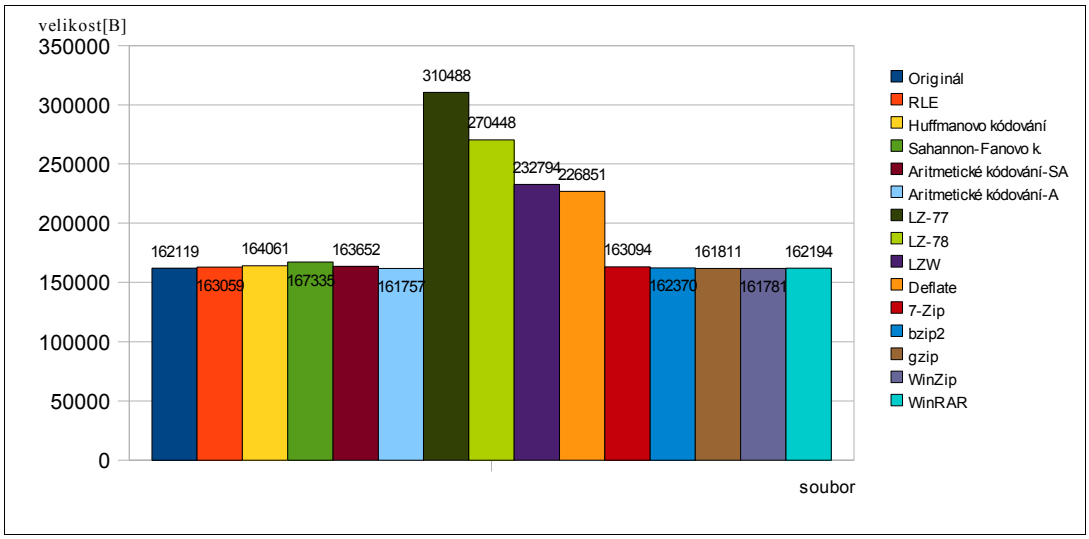
Při komprimaci již komprimovaného obrázku a to konkrétně metodou JPEG – výsledky viz obrázek 22, většina algoritmů dosáhla záporné komprimace. Výjimku tvoří adaptivní aritmetické kódování, důvodem je to, že metoda JPEG komprimuje „zig-zag“ Huffmanovým kódováním [30], které nám nevytváří optimální kód, jak tomu je v případě aritmetického kódování.

Dosažení komprimačních poměrů u komprimace videa můžeme vidět na obrázku 23. Z uvedeného grafu je zřejmé, že slovníkové algoritmy na komprimaci videa nejsou vhodné, jelikož video obsahuje velké množství různých symbolů.

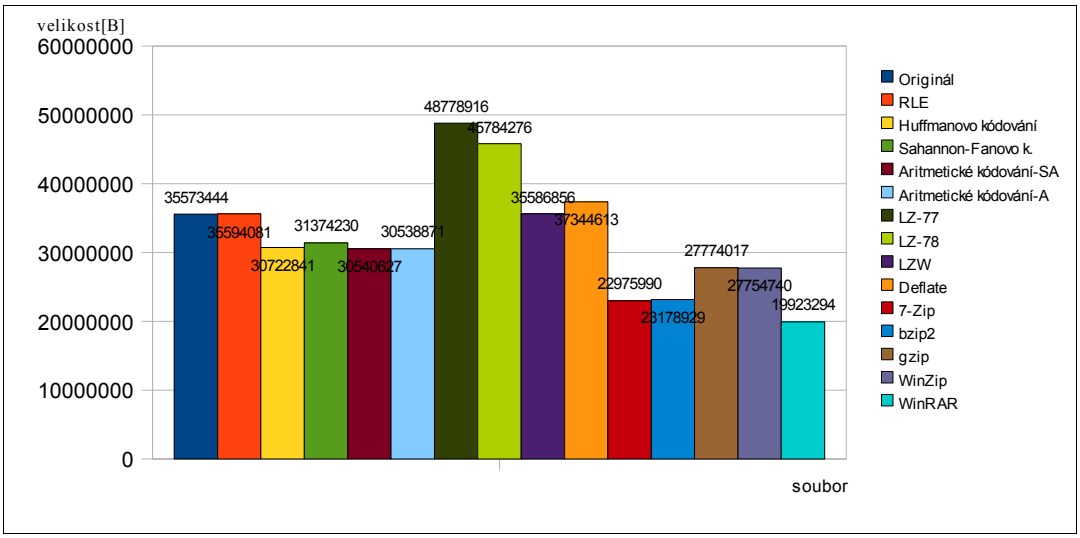
Komerční komprimační programy dosahovaly obdobných výsledků – opět lepších, než samotný algoritmus, jen na obrázku 19 – při komprimaci obrázku o barevné hloubce 24 bit, dosahovaly programy bzip2 a gzip horšího komprimačního poměru, než ostatní programy.



Obrázek 21: Datová velikost souboru Obrazek3.bmp po komprimaci



Obrázek 22: Datová velikost souboru Obrazek4.jpg po komprimaci



Obrázek 23: Datová velikost souboru video.avi po komprimaci

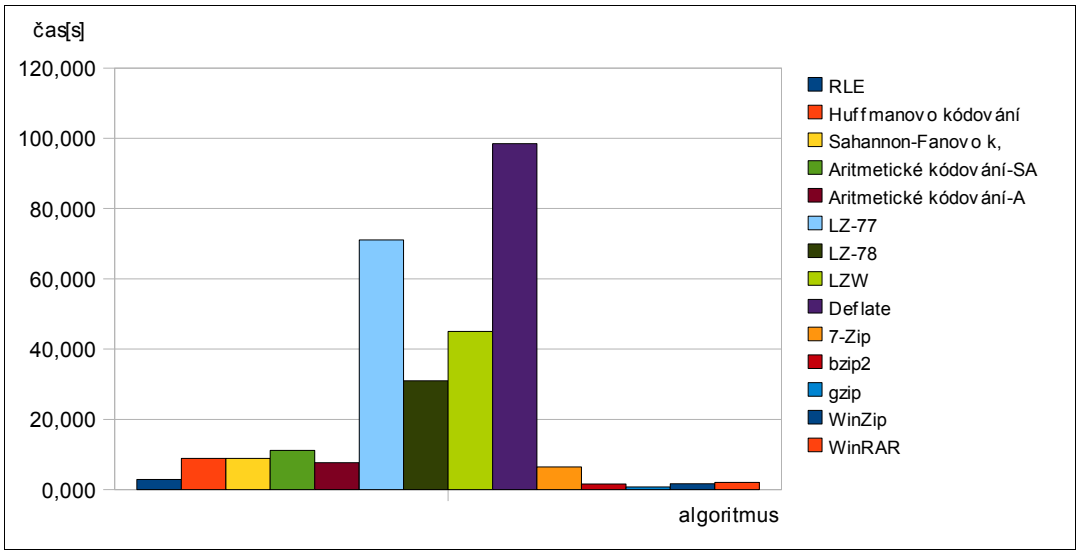
Tabulka 9: Přehled komprimačních poměrů dosažený algoritmy a programy u všech testovaných souborů

	Druh algoritmu / programu													
Název soubor	RLE	Huffmanovo kódování	Sahannon-Fanovo kódování	Aritmetické kódování, semi-adaptivní	Aritmetické kódování, adaptivní	LZ-77	LZ-78	LZW	Deflate	7-Zip	bzip2	gzip	WinZip	WinRAR
alice29.txt	98,53	58,05	59,48	57,49	57,29	73,96	76,51	46,12	59,89	31,92	28,69	36,51	36,65	35,05
text.txt	100,00	78,17	79,15	77,86	66,1	89,27	133,23	92,35	122,50	51,41	47,66	55,78	50,33	49,23
word-old.doc	56,84	51,13	51,64	50,49	44,52	49,11	68,88	51,31	45,8	24,74	28,56	27,72	27,49	27,01
word-new.docx	87,36	106,03	107,18	105,65	92,24	161,34	169,06	152,68	134,39	82,83	86,1	82,26	83,1	82,61
kennedy.xls	100,08	45,12	45,75	44,87	44,69	33,57	36,36	32,33	26,32	4,88	12,65	20,08	20,44	4
cp.html	99,87	68,65	70,21	68,17	66,22	64,2	92,43	60,76	60,05	31,46	30,99	32,51	32,94	32,42
fields.c	97,77	69,49	70,44	69,08	64,2	51,91	99,91	63,53	61,03	27,53	27,22	28,15	29,19	28,25
sum	89,64	72,4	73,56	71,95	67,03	64,41	92,24	65,52	57,62	24,89	33,76	33,8	33,71	30,33
komprimace.exe	69,64	67,11	68,45	66,63	66,01	45,26	61,7	42,47	37,86	18,1	24,26	24,36	24,3	20,57
Obrazek1.bmp	99,98	83,28	85,13	82,82	82,76	109,78	97,86	66,15	86,44	48,29	44,66	58,03	58,03	39,53
Obrazek2.bmp	86,04	60,42	61,5	59,84	59,6	62,54	62,77	39,69	50,08	32,42	32,69	36,42	35,83	35,46
Obrazek3.bmp	76,28	46,54	48,23	45,98	45,78	53,99	53,71	32,49	43,13	27,7	27,41	31,28	30,64	30,32
Obrazek4.jpg	100,58	101,2	103,22	100,95	99,78	191,52	166,82	143,6	139,93	100,6	100,15	99,81	99,79	100,05
video.avi	100,06	86,36	88,2	85,85	85,85	137,12	128,7	100,04	104,98	64,59	65,16	78,08	78,02	56,01
Legenda:	nejlepší komprimační poměr dosažený samotným algoritmem													
	nejhorší komprimační poměr dosažený samotným algoritmem													
	nejlepší komprimační poměr dosažený komerčním programem													
	nejhorší komprimační poměr dosažený komerčním programem													

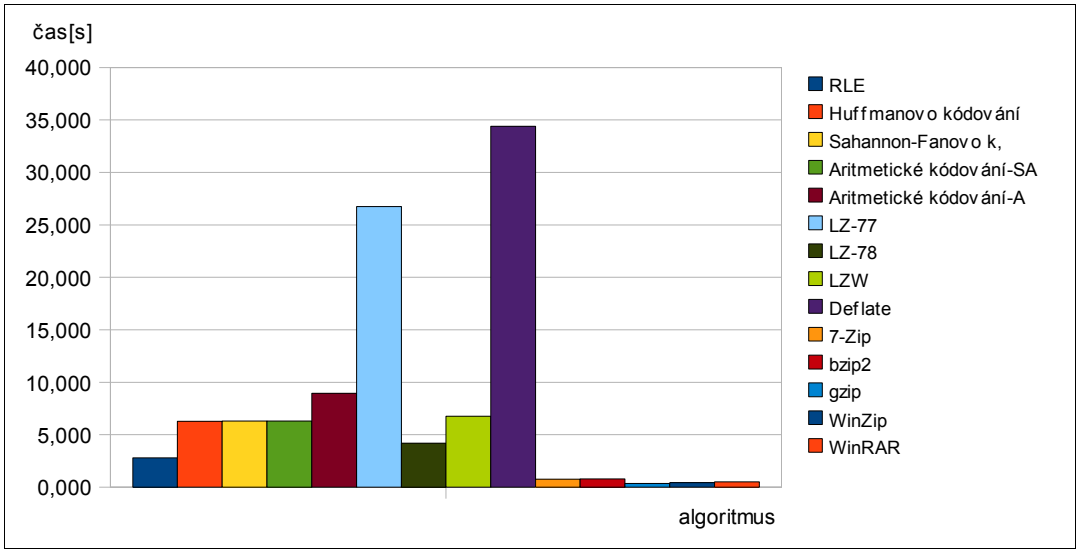
8.2.4 Testování časové náročnosti

Čas komprimace je do značné míry závislý na implementaci daného algoritmu. Komerční komprimační programy jsou vyvíjeny po několik let, a tak se dá říci, že mají dostatečně odladěný kód a datové struktury, proto dosahují o poznání lepších časů komprimace, dekomprimace. Obecně nejhůře dosahovaly, pomineme-li algoritmus Deflate, což je kombinace dvou algoritmů, výsledků LZW a LZ77. U LZ77 je to způsobeno tím, že byl implementován za pomocí různých datových kontejnerů a k vyhledávání byly použity standardní vyhledávací algoritmy z knihovny STL jazyka C++ a byly použity její vyhledávací algoritmy, což se poté ukázalo jako značně neefektivní řešení z hlediska časové náročnosti. U LZW je větší časová náročnost způsobena procházením k-cestného stromu a hledání frází ve slovníku.

Grafy uvedené na obrázcích 24 a 25 nám ukazují dosažené průměrné časy komprimace, respektive dekomprimace. Jednotlivé naměřené hodnoty časových údajů můžeme vidět v tabulce 10, kde jsou vyobrazeny dosažené časy samostatných algoritmů a v tabulce 11 jsou vyobrazeny časy dosažené komerčními komprimačními programy.



Obrázek 24: Dosažené průměrné časy při komprimaci



Obrázek 25: Dosažené průměrné časy při dekomprimaci

Tabulka 10: Časová náročnost jednotlivých algoritmů, jednotka času [s]

		Druh algoritmu								
Název soubor		RLE	Huffmanovo kódování	Sahannon-Fanovo k,	Aritmetické kódování-SA	Aritmetické kódování-A	LZ-77	LZ-78	LZW	Deflate
alice29.txt	komprimace	0,187	0,375	0,266	0,360	0,359	4,109	0,422	1,032	8,782
	dekomprimace	0,140	0,250	0,265	0,594	0,422	1,250	0,219	0,265	1,469
text.txt	komprimace	0,016	0,016	0,015	0,015	0,031	0,203	0,016	0,094	0,688
	dekomprimace	0,015	0,016	0,016	0,032	0,015	0,047	0,015	0,016	0,234
word-old.doc	komprimace	0,046	0,188	0,094	0,125	0,078	5,797	0,360	0,391	11,500
	dekomprimace	0,094	0,125	0,062	0,156	0,093	0,719	0,078	0,109	0,968
word-new.docx	komprimace	0,015	0,141	0,079	0,093	0,062	0,344	0,328	1,156	1,062
	dekomprimace	0,016	0,110	0,047	0,220	0,078	0,188	0,157	0,109	0,344
kennedy.xls	komprimace	1,047	1,828	1,828	2,265	1,656	12,015	3,531	7,875	148,656
	dekomprimace	0,984	1,469	1,437	4,094	2,047	8,032	0,938	1,312	9,187
cp.html	komprimace	0,016	0,063	0,047	0,063	0,063	0,547	0,062	0,203	1,314
	dekomprimace	0,016	0,062	0,047	0,092	0,078	0,203	0,047	0,047	0,266
fields.c	komprimace	0,031	0,062	0,047	0,047	0,031	0,500	0,047	0,140	1,782
	dekomprimace	0,032	0,047	0,047	0,062	0,031	0,125	0,031	0,047	0,391
sum	komprimace	0,063	0,219	0,141	0,312	0,187	5,078	0,719	0,672	6,782
	dekomprimace	0,046	0,157	0,218	0,594	0,312	1,000	0,172	0,110	0,609
komprimace.exe	komprimace	0,235	0,766	0,734	0,891	0,766	10,516	1,578	2,500	21,968
	dekomprimace	0,250	0,532	0,516	1,374	0,985	2,344	0,672	0,547	2,672
Obrazek1.bmp	komprimace	2,172	6,750	6,796	8,156	6,735	40,437	11,156	23,781	93,812
	dekomprimace	2,188	4,438	4,500	13,094	8,031	18,844	2,953	4,141	23,047
Obrazek2.bmp	komprimace	0,718	2,140	2,047	2,500	1,922	30,593	3,531	6,046	57,468
	dekomprimace	0,672	1,203	1,187	3,062	2,500	6,546	0,875	1,156	7,328
Obrazek3.bmp	komprimace	0,375	0,656	0,641	0,828	0,953	17,343	0,829	2,141	36,032
	dekomprimace	0,313	0,531	0,547	1,220	1,219	3,140	0,438	0,546	4,109
Obrazek4.jpg	komprimace	0,157	0,671	0,625	0,703	0,531	2,203	3,391	5,797	6,156
	dekomprimace	0,156	0,391	0,375	1,500	0,656	1,406	0,390	0,594	2,016
video.avi	komprimace	34,813	110,641	111,219	139,453	93,750	865,078	407,797	578,360	982,457
	dekomprimace	34,109	78,453	78,890	62,312	108,656	330,512	51,687	85,813	428,753

Tabulka 11: Časová náročnost jednotlivých komerčních komprimačních programů, jednotka času [s]

		Název programu				
Název soubor		7-Zip	bzip2	gzip	WinZip	WinRAR
alice29.txt	komprimace	0,406	0,204	0,141	0,390	0,313
	dekomprimace	0,140	0,140	0,109	0,250	0,234
text.txt	komprimace	0,203	0,125	0,125	0,422	0,250
	dekomprimace	0,219	0,125	0,125	0,281	0,234
word-old.doc	komprimace	0,219	0,187	0,125	0,360	0,235
	dekomprimace	0,172	0,109	0,125	0,234	0,234
word-new.docx	komprimace	0,203	0,141	0,110	0,406	0,266
	dekomprimace	0,156	0,156	0,125	0,265	0,218
kennedy.xls	komprimace	2,906	0,485	0,266	0,547	0,672
	dekomprimace	0,203	0,218	0,141	0,250	0,234
cp.html	komprimace	0,188	0,172	0,140	0,469	0,266
	dekomprimace	0,171	0,157	0,188	0,281	0,234
fields.c	komprimace	0,219	0,172	0,140	0,485	0,328
	dekomprimace	0,203	0,172	0,188	0,359	0,265
sum	komprimace	0,172	0,109	0,109	0,344	0,219
	dekomprimace	0,141	0,110	0,109	0,203	0,250
komprimace.exe	komprimace	0,421	0,187	0,140	0,516	0,391
	dekomprimace	0,204	0,266	0,235	0,281	0,265
Obrazek1.bmp	komprimace	3,969	1,109	0,594	1,156	1,688
	dekomprimace	0,500	0,594	0,187	0,547	0,515
Obrazek2.bmp	komprimace	2,000	0,437	0,360	0,750	0,890
	dekomprimace	0,250	0,281	0,125	0,282	0,266
Obrazek3.bmp	komprimace	1,032	0,265	0,250	0,500	0,531
	dekomprimace	0,203	0,188	0,141	0,281	0,266
Obrazek4.jpg	komprimace	0,375	0,156	0,157	0,500	0,328
	dekomprimace	0,172	0,140	0,109	0,281	0,219
video.avi	komprimace	77,718	18,937	7,407	16,703	21,812
	dekomprimace	8,094	8,406	3,140	2,328	3,782

9 Závěr

Prokázala se nám teorie o statistických algoritmech, kde Shannon-Fanovo kódování dosahovalo vždy horšího, v lepších případech stejného komprimačního poměru jako Huffmanovo kódování. Naopak Huffmanovo kódování dosahovalo ve většině případů horšího komprimačního poměru, než Aritmetické kódování. Jelikož oba algoritmy, respektive tři včetně Sahnnon-Fanova kódování jsou semi-adaptivní, ukládají model, což nám výsledky měření nijak nedegraduje, protože v jejich implementaci bylo použito stejného modelu. Dále je ze srovnání semi-adaptivní verze aritmetického kódování a adaptivní verze zřejmé, že adaptivní verze dosahuje ve většině případů lepšího komprimačního poměru, něž verze semi-adaptivní. Z toho může vyplývat, že by i adaptivní Huffman dosahoval lepších komprimačních poměrů, než verze semi-adaptivní. Pokud je to možné, je v praxi tedy lepší implementovat adaptivní verzi algoritmu, tím se nám zlepší komprimační poměr a zároveň se i podstatně zkrátí čas komprimace, neboť není potřeba procházet vstupní proud symbolů dvakrát.

U slovníkových metod se ukázal v praxi skoro nepoužitelný algoritmus LZ-78, také není implementován skoro v žádném komerčním programu. Oproti němu od něj odvozená verze LZW dosahuje dobrých komprimačních poměrů, jeho většímu rozšíření brání bohužel patentování.

Metoda RLE je použitelná pouze na vstupní proudy, kde se předpokládá následování více stejných symbolů za sebou. Tuto metodu lze tedy použít například na data po transformaci, např. BWT.

Jak je zřejmé z výsledků testování jednotlivých algoritmů, samotný algoritmus není schopen konkurovat komerčním programům, které jsou vyvíjeny po několik let a dá se říci, že již mají dostatečně odladěné datové struktury, aby dosahovaly co nejrychlejších komprimačních časů. Pro dosáhnutí nejlepšího komprimačního poměru je tedy nutné, použít více komprimačních algoritmů, viz např. metoda Deflate. Nejlépe je nejdříve data transformovat nějakou metodou, poté zakódovat například slovníkovou metodou a nakonec indexi slovníku zakódovat nějakou statistickou metodou.

Po té, co jsem se seznámil s odvětvím informatiky zabývající se komprimací dat a po nastudování komprimačních algoritmů, jsem byl schopen tyto algoritmy napro-

gramovat. Za pomocí jazyka C++ jsem naprogramoval komprimační algoritmy, které jsem poté mezi sebou otestoval. Využil jsem také nabytých znalostí knihovny QT-4.3, ve které jsem vytvořil grafický vzhled testovací aplikace. Na závěr bych chtěl říci, že znalosti, které jsem díky této bakalářské práci získal, jsou pro mě přínosem pro mou budoucí praxi.

10 Použitá literatura

[1] MORKES, David. *Komprimační a archivační programy*. 1. vyd. Brno : Computer Press, 1998. 177 s. ISBN 80-7226-089-8.

[2] WIKIPEDIA. *Ztrátová komprese* [online]. 2008 [cit. 2008-04-17]. Dostupný z WWW: http://cs.wikipedia.org/wiki/Ztrátová_komprese

[3] MURRAY, James, VAN RYPER, William. *Encyklopedie grafických formátů*. 1. vyd. Praha : Computer Press, 1995. 736 s. ISBN 80-85896-18-4.

[4] WIKIPEDIA. *Information entropy* [online]. 2008 [cit. 2008-04-20]. Dostupný z WWW: http://en.wikipedia.org/wiki/Information_entropy

[5] ČAPEK, Jan, FABIAN, Petr. *Komprimace dat - principy a praxe*. 1. vyd. Praha : Computer Press, 2000. 173 s. ISBN 80-7226-231-9.

[6] WIKIPEDIA. *Lossy compression* [online]. 2008 [cit. 2008-04-20]. Dostupný z WWW: http://en.wikipedia.org/wiki/Lossy_data_compression

[7] BALÍK, Miroslav , et al. *Komprese dat - Lexikon pojmů* [online]. [2007] [cit. 2008-04-15]. Dostupný z WWW: http://www.stringology.org/DataCompression/lexikon.html

[8] LABORATOŘ ZPRACOVÁNÍ PŘIROZENÉHO JAZYKA. *Frekvence písmen, bigramů, trigramů, délka slov* [online]. 2004 [cit. 2008-04-22]. Dostupný z WWW: http://nlp.fi.muni.cz/nlp/aisa/NlpCz/Frekvence_pismen_bigramu_trigramu_delka_slov.html

[9] SALOMON, David. *Data compression*. 3rd edition. New York : Springer, 2004. 920 s. ISBN 0-387-40697-2.

[10] TIŠNOVSKÝ, Pavel. *PCX prakticky - implementace komprimace RLE* [online]. 2006 [cit. 2008-04-15]. Dostupný z WWW: http://www.root.cz/clanky/pcx-prakticky-implementace-komprimace-rle/

[11] HUFFMAN, David. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*. 1952, no. 40, s. 1098-1101.

[12] BALÍK, Miroslav , HOLUB, Jan. *Komprese dat - Adaptivní Huffmanovo kódování - FGK* [online]. 2005-2006 [cit. 2008-04-24]. Dostupný z WWW: http://www.stringology.org/DataCompression/fgk/index_cs.html

[13] CODEPEDIA. *Huffman Trees for Data Compression* [online]. 2006 [cit. 2008-04-25]. Dostupný z WWW: http://www.codepedia.com/1/Art_Huffman_p1

[14] NOVÁK. *Aritmetické kódování* [online]. 1997 [cit. 2008-04-24]. Dostupný z WWW: http://atrey.karlin.mff.cuni.cz/~tnovak/compress/arit/

[15] NOSEK, Aleš. *Implementace kompresní metody PPM*. Praha, 2006. 73 s. ČVUT FEL. Diplomová práce. Dostupný z WWW: https://dip.felk.cvut.cz/browse/pdfcache/noseka1_2006dipl.pdf

[16] BODDEN, Eric, et al. *Arithmetic Coding revealed : A guided tour from theory to praxis*. 2007. 60 s. Dostupný z WWW: http://www-users.rwth-aachen.de/Eric.Bodden/files/ac/ac_en.pdf

[17] WIKIPEDIA. *Arithmetic_coding#US_patents_on_arithmetic_coding* [online]. 2008 [cit. 2008-04-27]. Dostupný z WWW: http://en.wikipedia.org/wiki/-Arithmetic_coding#US_patents_on_arithmetic_coding

[18] MAXIMUMCOMPRESSION. *Compression Programs* [online]. 2003- [cit. 2008-05-01]. Dostupný z WWW: http://www.maximumcompression.com/-programs.php

[19]WIKIPEDIA. *Range encoding* [online]. 2008 [cit. 2008-05-03]. Dostupný z WWW: http://en.wikipedia.org/wiki/Range_encoding

[20] BĚHÁLEK, Marek. *Programovací techniky.* Ostrava : [s.n.], 2006. 95 s. Dostupný z WWW: http://www.cs.vsb.cz/behalek/vyuka/pte/texty/Skripta%20PTE.pdf

[21] DVOŘÁK, Tomáš. *Algoritmy komprese dat* [online]. 2007 [cit. 2008-05-24]. Dostupný z WWW: http://ksvi.mff.cuni.cz/~dvorak/vyuka/SWI072

[22] DEUTSCH , Peter . *DEFLATE Compressed Data Format Specification version 1.3* [online]. 1996 [cit. 2008-05-06]. Dostupný z WWW: http://www.w3.org/Graphics/PNG/RFC-1951.html

[23] DIPPERSTEIN, Michael . *LZSS (LZ77) Discussion and Implementation* [online]. 2008 [cit. 2008-05-06]. Dostupný z WWW: http://michael.dipper-stein.com/lzss/

[24] WIKIPEDIA. *Prediction by Partial Matching* [online]. 2008 [cit. 2008-05-16]. Dostupný z WWW: http://en.wikipedia.org/wiki/Prediction_by_Partial_Matching

[25] BALÍK, Miroslav, et al. *PPMC* [online]. [2007] [cit. 2008-05-24]. Dostupný z WWW: http://www.stringology.org/DataCompression/ppmc/index_cs.html

[26] PC SVĚT. *Komprimace dat* [online]. [2003] [cit. 2008-05-14]. Dostupný z WWW: http://www.pcsvet.cz/art/article.php?id=4603

[27] ABEL, Jürgen. *Move To Front (MTF)* [online]. 2004-2008 [cit. 2008-05-27]. Dostupný z WWW: http://www.data-compression.info/Algorithms/MTF/index.htm

[28] POWELL , Matt . *The Canterbury Corpus* [online]. 2001- [cit. 2008-06-01]. Dostupný z WWW: http://corpus.canterbury.ac.nz/descriptions/

[29] *Archivace a komprese - který formát je pro vás nejlepší?* [online]. 2008 [cit. 2008-06-03]. Dostupný z WWW: http://www.ddworld.cz/windows/archivace-a-komprese-ktery-format-je-pro-vas-nejlepsi-2.html

[30] NELSON, Mark. *The Data Compression Book*. 2nd enl. edition. New York : M&T Books, 1996. 543 s. ISBN 1-55851-434-1.