VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

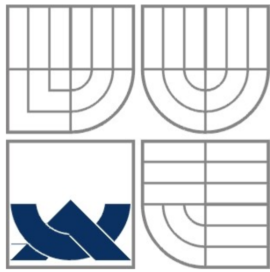
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KOMPRESE OBRAZOVÝCH DAT V MEDICÍNĚ

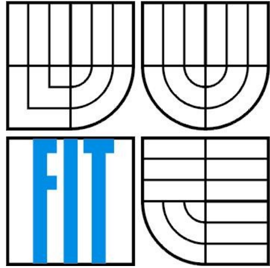
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JAKUB BALARIN
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KOMPRESE OBRAZOVÝCH DAT V MEDICÍNĚ

MEDICAL IMAGE DATA COMPRESION

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JAKUB BALARIN
AUTHOR

VEDOUCÍ PRÁCE Ing. Ph.D. PŘEMYSL KRŠEK
SUPERVISOR

BRNO 2007

Abstrakt

Práce zkoumá, jak se projeví účinek různých komprimačních algoritmů na obrazových datech v medicíně. Snaží se najít algoritmus nebo skupinu algoritmů, které budou mít největší kompresní účinek. Kromě použití klasických algoritmů je snaha využít vlastností medicínských dat (tj. že obsahují hodně podobných obrazových bodů) pro jejich lepší kompresi. Ověříme si účinnek delta kódování na výsledný kompresní poměr a na závěr uvedeme naši nejlepší nalezenou metodu.

Klíčová slova

RLE, BWT, Huffmanovo kódování, Aritmetické kódování, Delta kódování, Huffmanův strom, PPM, kontextově orientovaný model, LZW, LZ78, LZ77, Lineární interpolace, Burrows-Wheelerova transformace

Abstract

The efficiency of various compress algorithms on medical images is explored in this paper. It try's to find algorithm or group of algorithms with the best compress efficiency. Except classic algorithms the medicine data properties (it contains many similar image points) can be used to reach higher compress level. We verify delta coding efficienty to final compress ratio. At the end the best founded method is presented.

Keywords

RLE, BWT, Huffman coding, Arithmetic coding, Delta coding, Huffman tree, PPM, kontext-based, LZW, LZ78, LZ77, Linear interpolation, Burrows Wheeler Transformation

Citace

Jakub Balarin: Komprese obrazových dat v medicíně, bakalářská práce, Brno, FIT VUT v Brně, 2007

Komprese obrazových dat v medicíně

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Ph.D. Přemysla Krška

Další informace mi poskytli Ing. Michal Španěl

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jakub Balarin
10.května 2007

Poděkování

Děkuji Ing. Ph.D. Přemyslu Krškovi za odborné vedení bakalářské práce a konzultace.

Děkuji Ing. Michalu Španělovi za poskytnutý toolkit MDTSK a pomoc s některými implementacemi.

© Jakub Balarin, 2007.

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

Obsah

Obsah ... 1
1 Úvod ... 3
2 Bezeztrátové komprimační algoritmy ... 4
2.1 Základní pojmy ... 4
2.1.1 Redundance ... 4
2.1.2 Entropie ... 4
2.1.3 Prefixový kód ... 4
2.1.4 Minimální prefixový kód ... 5
2.1.5 Bezprefixový kód ... 5
2.1.6 BPC (Bits Per Charakter) ... 5
2.2 RLE (Run Length Encoding) ... 5
2.2.1 RLE0 ... 6
2.3 MTF Move To Front ... 7
2.4 Huffmanovo kódování ... 7
2.4.1 Huffmanův strom ... 8
2.4.2 Dekódování ... 9
2.5 Aritmetické kódování ... 9
2.6 Lempel-Ziv-Welch algoritmus ... 12
2.6.1 LZ77 ... 12
2.6.2 LZ78,LZW ... 14
2.7 BWT (Burrows Wheeler Transformation) ... 16
2.7.1 Transformace ... 16
2.7.2 Zpětná Transformace ... 17
2.7.3 Další úpravy po transformaci ... 19
2.8 PPM Prediction by Partial Matching ... 20
2.8.1 Statistický model ... 20
2.8.2 Kontextově orientovaný model (context-based) ... 20
2.8.3 Principy PPM ... 20
2.8.4 Implementační detaily ... 22
2.8.5 Různé varianty PPM ... 22
2.9 Speciální komprimace ... 23
2.9.1 Statická ... 23
2.9.2 Delta kódování ... 23
2.9.3 Lineární interpolace ... 23
3 Návrh ... 24

3.1 Použité prostředky 24
3.2 Vybrané algoritmy 24
4 Implementace 25
4.1 RLE 25
4.2 Huffmanovo kódování 25
4.3 BWT 25
4.4 MTF 25
4.5 PPM 26
4.6 Jiné použité techniky 26
4.6.1 Delta kódování 26
4.6.2 Rozdělení na sudé a liché 26
4.6.3 BME (Binary Matrix Exchange) 26
5 Testování 27
5.1 Postup a parametry 27
5.2 Testy 27
5.2.1 Základní sady algoritmů 27
5.2.2 Delta kódování 29
6 Závěr 30
Literatura 31
Seznam příloh 32

1 Úvod

Medicínská data se skládají převážně z obrázků (příloha č.2) radiologistických vyšetření. Nejčastěji se jedná o obrázky, reprezentované 12bity šedi. Tyto obrázky jsou velmi velké a díky komprimaci se zvyšuje datová propustnost a lepší archivovatelnost.

Existují dva druhy komprimací ztrátová a bezeztrátová:

Ztrátová komprimace – tento přístup využívá faktu, že lidský zrak je nedokonalý a rozlišuje hůře jednotlivé barvy. Díky této znalosti, můžeme podobné obrazové body nahradit jednou barvou. Ve výsledku pak dostáváme stejný obrázek, na první pohled nerozlišitelný od originálu, s daleko menší velikostí. Při dekomprimaci obrazu není výsledek na 100% totožný s originálem. Dochází tím ke ztrátě dat.

Bezeztrátová komprimace – jak už název napovídá, při opětovném rozbalení dat, dostáváme obrázek stejný jako originál. Na rozdíl od ztrátové komprimace dat, nedosahuje takových kompresních poměrů. Čím je obrázek složitější, tím se zhoršuje kompresní poměr této metody.

Které z těchto metod jsou nejvhodnější pro komprimaci obrazových medicínských dat? V podstatě se používají obě tyto metody. Ztrátová komprimace se používá v kombinaci z bezeztrátovou a to tak, aby při ztrátové komprimaci nedocházelo k vizuální ztrátě. Proto se bezeztrátová komprimace v medicínských datech, používá jako jeden z části celého komprimačního postupu a to ještě převážně k archivaci. Pro přenos po síti, se kvůli rychlosti používá čistá bezeztrátová varianta.

Mým úkolem tedy bude ukázat jak funguje bezztrátová komprimace dat, navrhnout a implementovat některé z postupů, otestovat jak veliký vliv budou mít při komprimaci medicínských dat. Dále se pokusit pomocí nějakých úprav , dostat lepší kompresní poměr při použití standardních algoritmů.

2 Bezeztrátové komprimační algoritmy

2.1 Základní pojmy

2.1.1 Redundance

Redundancí se označuje nadbytečnost některých informací v datech. Příkladem může být třeba hodnota 33, která je v jednom bajtu uložena takto : 00100001. Zde je vidět, že první dvě nuly se na výsledné hodnotě nějak nepodílí, proto je nazýváme redundantními. Naší snahou je získat kód s minimální redundanci, což je postata komprimace.

2.1.2 Entropie

Entropie je obecně veličina udávající míru neuspořádanosti zkoumaného systému nebo také míru neurčitosti daného procesu. Informační entropie často je také nazývana *Shannonovou entropií* po Claude E. Shannonovi, který zformuloval mnoho klíčových poznatků teoretické informatiky. Obecně pro systém s konečným počtem možných stavů S a pravděpodobnostní distribucí $P(s_i)$ je informační entropie definována jako střední hodnota

2.1.3 Prefixový kód

Je to podmínka pro jednoznačnou dekódovatelnost kódu. Předpona kódu znaku nesmí být kódem znaku jiného, jinak nepůjde kód správně dekódovat. Příklad:

Text : CAAABDAC

A	0
B	110
C	111
D	10

Výsledný prefixový kód: 10000110111010

Je vidět, že při dekódování, vždy jednoznačně určíme znak, který bude následovat.

2.1.4 Minimální prefixový kód

Využívá všech možných možností. Pro každou předponu kódu znaku. Platí, že pokud k ní přidáme znak z množiny {0,1}, jedná se vždy o kód znaku, nebo předponu kódu znaku.

2.1.5 Bezprefixový kód

U Bezprefixového kódu nelze jednoznačně dekódovat znak. Máme znaky A – 1, B – 11, C – 111 D - 1111. Je vidět, že i když má každý znak přiřazen svůj vlastní kód, přesto bychom nemohli ve zprávě 1111 říci, jestli obsahuje zakódované znaky AAAA nebo BB nebo CA nebo dokonce D atd. Přidáním jednoho speciálního symbolu do abecedy, který nám vždy oddělí jednotlivé znaky, problém vyřešíme.

Př.: speciální symbol pro oddělení jednotlivých slov bude v našem případě 0. Zpráva ABA, zakódovaná pomocí tohoto speciálního symbolu, bude vypadat takto 1011010. Bezprefixové kódování používá třeba Morseova abeceda.

2.1.6 BPC (Bits Per Charakter)

Také označováno jako průměrná délka kódu znaku. Udává průměrnou velikost kódu jednoho znaku v bitech. Je to jedno z nejčastějších výkonnostních měřítek komprimačních algoritmů.

2.2 RLE (Run Length Encoding)

Jednou z nejstarších metod komprese je metoda proudového kódování RLE, která se snaží v datovém toku objevit a redukovat posloupnosti opakujících se znaků. Místo této posloupnosti znaků je uložen pouze speciální znak představující indikátor znaku a počet jeho opakování.

Účinnost této metody je velice závislá na charakteru dat které komprimujeme. V případě že máme nějaké obrázky, které obsahují velké barevné plochy, můžeme dosáhnout velice dobrého výsledku. Na druhou stranu, na obrázcích s velkou barevnou hloubkou, asi těžko můžeme očekávat nějakou kompresi.

Příklad komprese řetězce délky 16 znaků:

vstup	**výstup**	**výsledná velikost**
`xyzzzzyyyyyxwwww`	`x0y0z4y5x0w4`	12
`xyzzxyzzxyzzzzzz`	`x0y0z1x0y0z1x0y0z5`	17

Tabulka komprimace metody rle

Největší nevýhoda této kompresní metody je, že při komprimaci nevhodného druhu dat, můžeme dosáhnout záporné komprese a soubor místo zmenšení zvětšit. Existuje několik způsobů, jak tento problém řešit:

1. pokud jsou alespoň dva po sobě jdoucí znaky stejné, následuje za nimi číslo, které udává,kolik těchto znaků ještě po nich následuje. Takhle dopadne předchozí příklad: xyzz2yy3xww2 a xyzz0xyzz0xyz5 . Výsledná velikost je 12 a 14 znaků. Pravděpodobnost výskytu dvou a více stejných znaků je rozhodně nižší, než výskyt různých po sobě jdoucích znaků.

   Je vidět, že problém záporné velikosti, jsme částečně odstranili, ale pouze pokud se v textu nebude vyskytovat velké množství dvou stejných po sobě následujících znaků. Tento jednoduchý postup se používá, nejčastěji třeba u modemů. Pouze se znaky kódují při více jak třech stejných po sobě jdoucích znaků.

2. Trochu pozměníme formát v jakém se ukládá výstup. Budeme rozlišovat dva případy, zda následující text lze komprimovat či ne. Například:

   CA5 - nám říká, že C vyskytují se stejné znaky, A jaké znaky to jsou, 5 kolik jich je.
   N4abcd - nám říká,že N znaky nejsou stejné, 4 jsou čtyři a za nimi následují ty znaky.
   Výstup předchozího příkladu by vypadal takto:

výstup	výsledná velikost s úpravou ukládání
N2xyCz4Cy5N1xCw4	11
N10xyzzxyzzxyCz6	13

   Na první pohled, to vypadá, že jsme si moc nepomohli. Ale když si představíte, že informaci o počtu znaků a o tom zda jsou stejné nebo ne, lze uložit do jednoho bajtu místo dvou, vychází vám v prvním případě délka 11 a ve druhém 13. Navíc, lepší implementace si průběžně hlídá, zdali se komprese nezhoršuje, a pokud ano, tak i některé stejné po sobě jdoucí znaky, dává do bloku nekomprimovaných.

2.2.1 RLE0

Jedná se o jednu z modifikací standardního algoritmu RLE. V tomto případě se ukládají pouze posloupnosti nul jdoucích bezprostředně za sebou. Nejčastější použití této varianty je v datech s velkým výskytem nul, případně se v souboru nějakým způsobem zvedne četnost nul, pro dosažení větší účinnosti.

2.3 MTF Move To Front

MTF není komprimační metoda, ale slouží k překódování dat, aby se dali potom úsporněji uložit. Metoda funguje na principu zásobníku.

Vezmeme první znak z textu a podíváme se do zásobníku, na jaké pozici, se od vrcholu zásobníku, znak nalézá. Hodnotu zapíšeme na výstup, znak ze zásobníku vyjmeme a vložíme ho na vrchol. Tento postup opakujeme pořád, dokud jsou na vstupu nějaké znaky.

vstupní řetězec v ASCI tvaru:

97	109	109	116	116	32	97	97	115	115

výstup po aplikaci metody MTF:

97	109	0	116	0	35	3	0	116	0

Jak je patrné z příkladu, tato metoda, nejen nahrazuje po sobě jdoucí znaky nulami, ale i snižuje ASCI hodnoty skupin znaků, často se opakujících, a tím současně zvyšuje četnost jednotlivých znaků. Tohoto jevu, se s výhodou využívá, pro pozdější aplikaci některé statistické metody.

Dekódování probíhá prakticky stejným způsobem, pouze se postup obrátí. Vždy načteme hodnotu, která nám současně říká, pozici znaku v zásobníku. Nalezený znak zapíšeme na výstup a opět ho přesuneme na počátek zásobníku.

2.4 Huffmanovo kódování

Algoritmus byl navržen Davidem Huffmanem roku 1952. Je asi nejznámějším zástupcem skupiny algoritmů, které využívají pravděpodobnost výskytu jednotlivých znaků. Toto kódování využívá optimálního (nejkratšího) prefixového kódu. Přiřadí tedy častěji se vyskytujícím znakům kratší kódy (např. 2 bity místo původních 8) a naopak. Délky kódů jednotlivých znaků jsou celočíselné, to však nemusí být ideální (místo 6 bitů by bylo výhodnější např. 5.25 bitu). Lepšího výsledku se tedy dosáhne kódováním n-tic znaků (k "zaokrouhlování" na celý bit dochází méně často), tím však prudce rostou nároky na paměť, zatímco výsledný kompresní poměr se zlepšuje jen velmi pomalu. Nejčastěji se používá jako poslední stupeň v kódování dat.

2.4.1 Huffmanův strom

Pro sestavení Huffmanova stromu je nutné, nejdříve zjistit četnost vstupních znaků. Můžeme ji zjistit jedním průchodem daty, nebo použít tabulku výskytů jednotlivých symbolů.

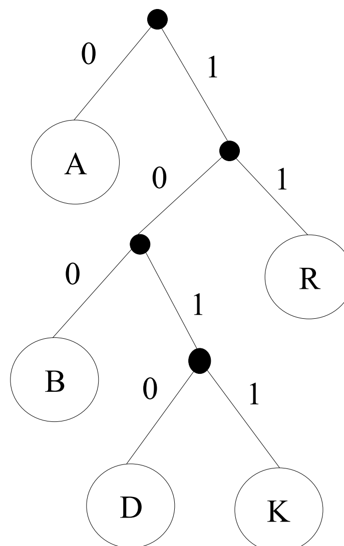
Huffmanův strom začínáme tvořit od počátečních listů (tj. Znaky s nejmenší četností. Vždy vezmeme dva nejméně četné symboly, a nahradíme je společným uzlem. Hodnota tohoto uzlu je rovna součtu četností jeho dvou potomků. Tento postup opakujeme, dokud nám nezůstane jeden uzel. Říkáme mu kořenový uzel.

Vstupní řetězec:: ABRAKADABRA

Zjistili jsme četnost výskytu jednotlivých znaků:

A	B	D	K	R
5	2	1	1	2

Na základě výše popsaného postupu, sestavíme Huffmanův strom:



2.1 Huffmanův strom

Z obrázku je vidět, že nejčetnější znaky opravdu dostávají nejkratší kódy. Vidíme zde také, jak se zaokrouhlování projeví na kódování jednotlivých znaků. Znaky B a R mají stejnou četnost výskytu, ale nemají stejně dlouhé výsledné kódy. Je vidět, jak se projevují zaokrouhlovací chyby. Pro pořádek jen uveďme, že Huffmanovo kódováni je nejefektivnější, když symboly mají pravděpodobnosti výskytu rovné záporným mocninám 2 (tj. číslům, jako 1/2, 1/4, 1/8 atd.).

2.4.2 Dekódování

Dekódování probíhá v podstatně stejně. Pouze procházíme Huffmanův strom od kořene k listům a jednotlivé symboly zapisujeme na výstup.

2.5 Aritmetické kódování

Aritmetické kódování, se stejně jako Huffmanovo kódování, řadí mezi statistické metody. Huffmanova metoda je jednoduchá, účinná a produkuje nejlepší kódy pro jednotlivé datové symboly. Jediným případem, kdy vytváří ideální kódy s proměnnou délkou (kódy, jejichž střední délka je rovna entropii) je, když symboly mají pravděpodobnosti výskytu rovné záporným mocninám 2 (tj. číslům, jako 1/2, 1/4, 1/8 atd.). Je to proto, že Huffmanova metoda přiděluje všem symbolům abecedy kódy s celočíselným počtem bitů. Symbol s pravděpodobnostní 0,4 by měl mít přidělen v ideálním případě kód s 1,32 bity. Avšak Huffmanova metoda normálně přidělí takovému symbolu kód dlouhý 1 nebo 2 bity.

Aritmetické kódování překonává problém přidělování celočíselných kódů jednotlivým symbolům tak, že přidělí jeden kód (velmi dlouhý) celému vstupnímu souboru. Metoda startuje s jistým intervalem, čte vstupní soubor symbol za symbolem a používá pravděpodobnosti jednotlivých symbolů k zužování intervalu. Určení užšího intervalu vyžaduje další bity, takže délka čísla, které algoritmus konstruuje průběžně roste. Aby se dosáhlo komprese, je algoritmus navržen tak, že symbol s vysokou pravděpodobností zúží interval méně, než symbol s nízkou pravděpodobností. Výsledkem je , že symboly s vysokou pravděpodobností přispívají do výstupu méně bity.

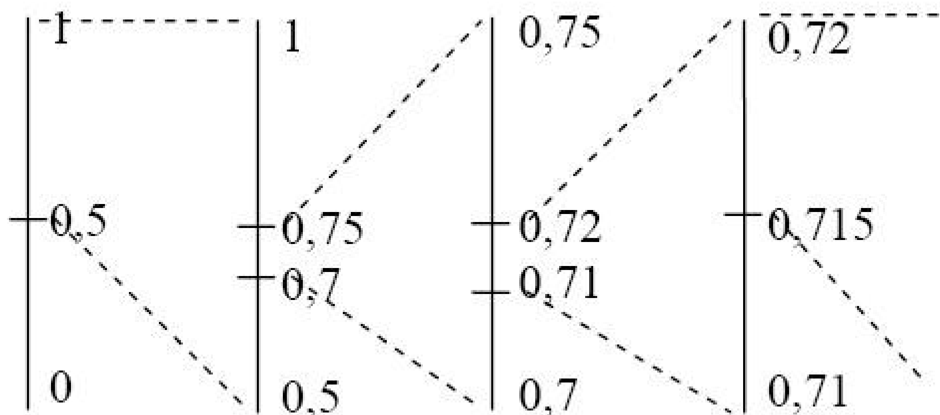
Příklad komprimace řetězce „SWISS□MISS“:

Jako první krok, provedeme zjištění četnosti jednotlivých znaků. Na základě zjištěných četností, rozdělíme interval na jednotlivé pod intervaly. Vytvoříme tak tabulku, kde každý znak bude mít svoji četnost a interval do kterého spadá.

Znak	Četnost	interval
S	5	<0,5;1.0)
W	1	<0.4;0.5)
I	2	<0,2;0,4)
M	1	<0,1;0,2)
(mezera)	1	<0,0;0,1)

2.2 Tabulka četností a intervalů jednotlivých znaků

S takto vytvořenou tabulkou 2.2 můžeme začít vstupní text komprimovat. Postupně budeme číst jednotlivé znaky a na základě intervalu, který jim byl přidělen, postupně budeme náš výsledný interval zužovat.



2.3 *Tento obrázek ilustruje postup komprimace prvních 3 znaků.*

Čteme první znak S a náš počáteční interval je <0,0;1,0). Koukneme se, jaký interval přísluší tomuto znaku. V našem případě je to interval <0,5;1.0). Náš počáteční interval <0,0;1,0) zúžíme o interval znaku, který jsme načetli a dostaneme výsledný interval <0,5;1.0). Načteme další symbol W, jehož interval je <0.4;0.5). Protože jsou jednotlivým znakům z tabulky přiděleny intervaly z intervalu <0,0;1.0), a my chceme zpřesnit interval <0,5;1.0), musíme výsledek správně přepočítat. Při přepočítávání uvažujeme interval <0,5;1.0) jako tento <0,0;1.0). Díky tomuto kroku, je výsledný zúžený interval <0,7;0.75). Načteme další znak I, který má interval <0,2;0,4). Zpřesníme náš interval o tento znak a dostaneme <0,71;0.72). Takhle pokračujeme stále, dokud je nějaký znak na vstupu.

Výsledná hodnota je vždy kterékoliv číslo z výsledného intervalu. V našem případě nám vyšlo 0,71753375. Tohle číslo se patřičně zakóduje na výstup. Ukázka celého postupu viz příloha č.1.

Dekodér pracuje opačně obr 2.4: Začne načítáním symbolů a jejich rozsahů a vytvoří si znovu tabulku s jednotlivými intervaly. Pak načte zbytek kódu. První číslice je „7“, takže dekodér již ví, že celý kód je číslo ve tvaru 0,7.... Toto číslo je uvnitř pod intervalu <0,5;1,0) znaku S, takže první s posloupnosti znaků je znak S. Dekodér eliminuje vliv znaku S na kód odečtením dolní meze intervalu 0,5 znaku S a dělí ho šířkou pod intervalu S (0,5). Výsledkem je 0,4350675. Dekodér z výsledku pozná, že následující symbol je W (protože pod interval W je <0,4, 0,5)). Takto dekodér pokračuje, dokud není celý řetězec dekódovaný.

Znak	Kód – dolní mez	Šířka pod intervalu	výsledek
S	0,71753375 – 0,5 = 0,21753375	/0,5	0,4350675
W	0,4350675 – 0,4 = 0,0350675	/0,1	0,350675
I	0,350675 – 0,2 = 0,150675	/0,2	0,753375
S	0,753375 – 0,5 = 0,253375	/0,5	0,50675
S	0,50675 – 0,5 = 0,00675	/0,5	0,0135
(mezera)	0,0135 – 0 = 0,0135	/0,1	0,135
M	0,135 – 0,1 = 0,035	/0,1	0,35
I	0,35 – 0,2 = 0,15	/0,2	0,75
S	0,75 – 0,5 = 0,25	/0,5	0,5
S	0,5 – 0,5 = 0	/0,5	0

2.4 Ukázka dekódování celého testovacího řetězce

Některé implementační detaily:

- I když počítáme s reálnými čísly, v praxi se výpočty převádí na čísla celá. Je to mimo jiné kvůli rychlosti, protože práce s reálnými čísly je několikanásobně náročnější.
- Konečný výsledek se neukládá jako jedno reálné číslo např. 0,78546876, ale jako číslo celé 78546876. Je to kvůli tomu, že uložení desetinné části v reálných číslech, nemusí být přesné.
- Musíme se vyvarovat veškerých zaokrouhlovacích chyb, jinak výsledek nepůjde dekódovat. To samé platí i o poškození dat, potom vůbec nejde data zpětně rekonstruovat.
- V případě, že některý znak je mnohonásobně četnější než všechny ostatní, může dojít k situaci, že se data špatně zakódují. Proto se do tabulky vkládá ještě jeden pomocný symbol „eof", který má malou pravděpodobnost a ukončuje kódovaný řetězec.
- Při kódování velkých oběmů dat, můžeme narazit na problém, že nepůjdou celé zakódovat jako jedno slovo. Proto se v praxi vstupní data kódují po blocích nejčastěji v řádech Kb.

2.6 Lempel-Ziv-Welch algoritmus

Jeho počátky jsou spojovány se jmény Abraham Lempel a Jakob Ziv. Oba v letech 1977 a 1978 vypracovali kompresní algoritmy, které jsou známy pod zkratkami LZ77 a LZ78. V roce 1984 na jejich práci navázal Terry Welch, který modifikoval kompresní algoritmus LZ78 pro potřeby hardwarových zařízení, konkrétně diskových řadičů. Tak vznikla podoba Lempel-Ziv-Welch algoritmu známém pod zkratkou LZW.

Základním principem toho komprimačního algoritmu je vyhledávání stejných posloupností bajtů v originálním souboru. Pomocí odkazů na tyto posloupnosti dat si algoritmus buduje datový slovník.

Komprimace pak probíhá v těchto krocích:

- Pokud se posloupnost bajtů (řetězec) ve vytvářeném slovníku nevyskytuje, je tato posloupnost přidána do slovníku a ve stejném tvaru zapsána do komprimovaného výstupu.
- Pokud se posloupnost bajtů (řetězec) ve slovníku již nachází, zapíše se na výstup zástupná hodnota odpovídající nalezené vstupní posloupnosti. Zástupná hodnota je vždy menší než načtený řetězec bajtů, a tím dochází ke kompresi.

2.6.1 LZ77

Základní myšlenkou této metody je použít části dříve načtených vstupních dat jako slovníku. Důležitou součástí této metody je pohyblivé okno. Algoritmus udržuje nad vstupními daty okno a nasouvá vstup do tohoto okna zprava doleva tak, jak se čtou řetězce symbolů. Okno je rozděleno na dvě části. Levá část se nazývá *vyhledávací buffer* a pravá je *předvídací buffer.*

vyhledávací buffer - vždy obsahuje symboly, které byly zakódovány v poslední době. Je to slovník algoritmu. Jeho velikost se pohybuje v řádech KB.

předvídací buffer - obsahuje text, který se bude teprve komprimovat.

Komprimace:

mějme řetězec „leze leze po železe“ .

Po komprimaci bude vypadat takhle „leze [0,5] po že [0,4]“

Zástupný symbol znamená vždy pozici a délku v předchozím vyhledávacím bufferu.

Vyhledávací buffer	předvídací buffer	Nej. Symbol	výstup
l	e ze leze po železe	e	le
le	ze leze po železe	z	lez
lez	e leze po železe	e	leze
leze	leze po železe	(mezera)	leze
leze	leze po železe	leze	leze [0,5]
leze [0,5]	po železe	(mezera)	leze [0,5]
leze [0,5]	po železe	p	leze [0,5] p
leze [0,5] p	o železe	o	leze [0,5] po
leze [0,5] po	železe	(mezera)	leze [0,5] po
leze [0,5] po	železe	ž	leze [0,5] po ž
leze [0,5] po ž	eleze	e	leze [0,5] po že
leze [0,5] po že	leze	leze	leze [0,5] po že[0,4]
leze [0,5] po že[0,4]			leze [0,5] po že[0,4]

2.5 Tabulka průběhu celé komprimace řetězce

Jak je vidět z tabulky 2.5, tak algoritmus ukládá do souboru jako zástupné znaky, pouze pozici a offset. Rychlost a účinnost závisí na velikosti vyhledávacího bufferu. Při moc velkém vyhledávacím bufferu je dlouhá doba hledání a doba komprese se značně prodlužuje. Dekomprese je je velice jednoduchá. Dekompresor prochází vstupní soubor a jednotlivé offsety nahrazuje za text, na který ukazují.

Implementační detaily:

- pro zvýšení účinnosti komprese, se někdy offsety kódují některou statistickou metodou (huffman nebo aritmetická komprese).
- *Vyhledávací buffer* se implementuje pro zvýšení rychlosti pomocí kruhové fronty, kdy z jedné strany prvky vstupují a druhou opouští.
- Velikost slovníku se volí nejčastěji od 1KB – 16KB. Větší hodnoty neúměrně prodlužují dobu komprimace za cenu trochu lepšího výsledku.
- Pro rychlé hledání v bufferu, se dá použít jednoduché finty. Vytvořit si pomocný buffer který bude příslušet ke každému znaku v bufferu a bude mu říkat, kde je následující stejný symbol popřípadě dvojce symbolů. Při hledání nejdelšího slova prostě jen vezmeme jeho první písmeno a skáčeme po všech těchto písmenech v bufferu. Tímto výrazně urychlíme hledání nejdelšího slova v bufferu.
- Je více způsobů jak rozeznat jestli se jedná o znak nebo odkaz na slovo. Nejpoužívanější je zápis, kdy první bit nám říká, zda jde o samostatný znak , nebo odkaz na slovo.

2.6.2 LZ78,LZW

Zatímco metoda LZ77 vytváří svůj slovník pomocí odkazů do již zkomprimovaného textu, vylepšený algoritmus LZW patřící do rodiny algoritmů LZ78 používá slovníkové odkazy odlišně. Metoda LZW vytváří dynamicky slovník opakujících se řetězců v průběhu komprimace. Existují různé varianty rodiny algoritmů LZ78, liší se pouze ve způsobu vytváření slovníku.

Algoritmus čte postupně řetězce znaků a nahrazuje je ve výstupu čísly z předem definovaného intervalu. Jednotlivé nalezené řetězce si ukládá do slovníku a přiděluje jim zástupná čísla. Celá tabulka se vytváří za běhu dynamicky a proto není nutné ji nikam ukládat.

Postup komprimace obr 2.6:

Algoritmus začíná s prázdným slovníkem a řetězcem *L* obsahujícím první znak zdrojového souboru. Vždy po přečtení dalšího znaku *c* zjistí, jestli se řetězec *L+c* již nevyskytuje ve slovníku. Pokud ano, algoritmus pouze prodlouží řetězec *L* o znak *c.* Pokud ne, algoritmus zapíše např. 12-ti bitový odkaz do slovníku nebo (pokud řetězec *L* obsahuje jediný znak) pouze znak, ale opět 12-ti bitově. V tomto případě tedy čísla 0-255 jsou jednotlivé znaky a čísla 256-4095 indexy do slovníku. Pokud se slovník zaplní dříve než je přečten celý soubor, je celý slovník smazán a začíná se plnit znovu. Vzhledem k tomu, že slovník se plní od nejnižších pozic, je pro zlepšení kompresního poměru výhodné začít kódy ukládat pouze 9-ti bitově a teprve když se slovník více zaplní, postupně je rozšiřovat až na těch např. 12 bitů. Slovník dekompresoru se totiž plní stejně a tak dekompresor snadno pozná, kdy začít kódy číst 10-ti bitově.

Řetězec L	Znak C	Výstup	Položka v slovníku
		W	
W	E	W	(256)=WE
E	B	E	(257)=EB
B	/	B	(258)=B/
/	W	/	(259)=/W
W	E		
WE	B	(256)	(260)=WEB
B	/		
B/	W	(258)	(261)=B/W
W	E		
WE	B		
WEB	!	(260)	(262)=WEB!
!	(eof)	!	

2.6 Ukázka průběhu komprese řetězce „WEB/WEB/WEB!“

Dekomprese:

Algoritmus postupně čte kódy, zapisuje jim příslušející řetězce a přidává nové řetězce do slovníku. Do slovníku je vždy přidán řetězec reprezentovaný předcházejícím kódem a první znak z řetězce s aktuálním kódem. V případě, že byl přečten kód, kterému ještě nebyl přiřazen řetězec (to se stane např. pokud původní text obsahoval několik stejných znaků za sebou), je do slovníku přidán řetězec, skládající se z předcházejícího řetězce a jeho posledního znaku. V průběhu dekomprese má dekompresor stejný slovník jako kompresor.

Dekompresní schéma algoritmu LZW:

```
read last
write last
while (!eof(input)) {
  read code
  write řetězec table[code] (nebo znak code)
  přidání řetězce table[last]+první znak table[code] do slovníku
  last = code
}
```

Jednou nezanedbatelnou výhodou je využitelnost algoritmů LZW u takových zařízení, kterými data pouze procházejí a kde je nutné zpracovat data rychle. Takovými zařízeními jsou například modemy. Většina jiných komprimačních algoritmů pracuje po blocích a data se vyšlou až po komprimaci celého bloku. Naproti tomu algoritmus LZW může dada číst a vysílat postupně, a tím umožnit zařízení plynulý provoz.

Implementační detaily:

- Velikost slovníku se volí v počtech kolik slov dokáže pojmout. Nejčastěji 8-16K slov. Větší slovník sice dává větší šanci na lepší kompresní poměr, ale zvyšují se paměťové nároky a rychlost vyhledávání klesá.
- Pokud se slovník úplně zaplní, může se kromě jeho úplného vymazání, zvolit taktika LRU (last-recently-used) pro odstranění nepoužívaných řetězců ze slovníku. Další možností je slovník vůbec nemazat, tím se slovník stane statický.
- Algoritmus LZW se nejčastěji používá v kombinaci s některou statistickou metodou. Pro zvýšení účinnosti se někdy zaznamenávají kolikrát dané slovo bylo nalezeno a díky tomu huffman nebo aritmetická komprese dokáže lépe uložit zástupné symboly.

2.7 BWT (Burrows Wheeler Transformation)

Burrows-Akcelerovat transformace (BWT) je metoda, jak změnit hůře komprimovatelná data na lépe komprimovatelná. Celá BWT spočívá ve vratném setřídění bloku dat. Cílem celé transformace je využití základní vlastnosti většiny dobře zabalitelných souborů k seřazení stejných znaků. U této transformace se využívá základní vlastnosti textu tj. opakování stejných řetězců. Výsledkem BWT transformace bude původní text, ale s velkou pravděpodobností budou stejné znaky pohromadě.

2.7.1 Transformace

originál	Po setřízení
naprogramovaný program	programnaprogramovaný
aprogramovaný programn	amnaprogramovaný progr
programovaný programna	amovaný programnaprogr
rogramovaný programnap	aný programnaprogramov
ogramovaný programnapr	aprogramovaný programn
gramovaný programnapro	gramnaprogramovaný pro
ramovaný programnaprog	gramovaný programnapro
amovaný programnaprogr	mnaprogramovaný progra
movaný programnaprogra	movaný programnaprogra
ovaný programnaprogram	naprogramovaný program
vaný programnaprogramo	ný programnaprogramova
aný programnaprogramov	ogramnaprogramovaný pr
ný programnaprogramova	ogramovaný programnapr
ý programnaprogramovan	ovaný programnaprogram
programnaprogramovaný	programnaprogramovaný
programnaprogramovaný	programovaný programna
rogramnaprogramovaný p	ramnaprogramovaný prog
ogramnaprogramovaný pr	ramovaný programnaprog
gramnaprogramovaný pro	rogramnaprogramovaný p
ramnaprogramovaný prog	rogramovaný programnap
amnaprogramovaný progr	vaný programnaprogramo
mnaprogramovaný progra	ý programnaprogramovan

2.7 Ukázka transformace BWT

Nejprve vytvoříme N-tici znaků, kde N je velikost transformovaného souboru. Prvním řádkem této N-tice bude obsah celého souboru v podobě řetězce znaků. Následně tento řádek posuneme o jeden znak doleva a umístíme jej do druhého řádku. Celý postup opakujeme, dokud nedokončíme celou N-tici o N^2 znacích. Poté všechny řádky abecedně seřadíme (mezi sebou, nikoliv jejich obsahy). Do výstupního souboru zapíšeme poslední sloupec a aktuální pozici prvního řádku (originálního souboru) v seřazené N-tici (číslo 0..N-1).

Výsledek našeho příkladu dopadl takto:

Před transformací:	naprogramovaný program
Po transformaci:	Ýrrvnooaamarrm aggppon

Jak je vidět, stejné znaky se opravdu seskupují k sobě. Což je výhodné, protože při transformaci velkého souboru, se vytváří velké skupiny stejných po sobě jdoucích znaků. A jak víme, tohle velice vyhovuje metodě RLE.

2.7.2 Zpětná Transformace

Ač se to na první pohled nezdá, tak BWT je skutečně vratná. Existují dva způsoby, jak transformaci dostat do původní podoby. Při prvním jednodušším způsobu však musíte zrekonstruovat celou matici.

Rekonstrukce 1:

Viz obrázek 2.8. První sloupec matice získáte obyčejným abecedním seřazením posledního sloupce, který už máte. Ke každému i-tému znaku vpravo existuje znak vlevo. Pokud např. k prvnímu znaku "a" náleží znak "b" vlevo, musíte pro rekonstrukci druhého sloupce najit první znak "a" vlevo a doplnit do druhého sloupce vedle tohoto "a" "b". Pokud se některý znak vyskytuje několikrát, můžete ho použít vždy jenom jednou. Musíte rovněž zachovat směr řazení tedy shora dolů. Pokud tedy hovořím o prvním "a", myslím první "a" shora. Prvnímu a druhému písmeni "o" vlevo náleží písmena "g" vpravo => za prví dvě "o" vpravo následují písmena "g". Třetímu "o" vpravo náleží však už písmeno "v". Tímto způsobem zrekonstruujeme celý druhý sloupec. Při rekonstrukci třetího hledáme pro příslušné znaky ve sloupci posledním znaky ve sloupci druhém a tímto způsobem dokončíme celou matici a řádek číslo 9 (počínaje nulou) bude odpovídat našemu původnímu řetězci.

Tento způsob je téměř nepoužitelný. K rekonstrukci 32MB^2 matice je potřeba 1024PB (2^50 bajtů) Ram. Paměťové nároky je možné sice snížit (stačí si uchovávat jenom poslední dva protilehlé sloupce a pochopitelně části originálního řádku), ale výpočet bude stejně pomalý, protože každý znak matice musíte zpracovávat samostatně.

První sloupec získáme seřazením sloupce posledního	Postupné doplňování ostatních řádků:
--------------------ý	--=>----------------ý
a--------------------r	a--=>----------------r
a--------------------r	a--=>----------------r
a--------------------v	a--=>----------------v
a--------------------n	a--=>----------------n
g--------------------o	g--=>----------------o
g--------------------o	g--=>----------------o
m--------------------a	m--=>----------------a
m--------------------a	m--=>----------------a
n--------------------m	n--=>----------------m
n--------------------a	n--=>----------------a
o--------------------r	og-=>----------------r
o--------------------r	og-=>----------------r
o--------------------m	ov-=>----------------m
p--------------------	p--=>----------------
p--------------------a	p--=>----------------a
r--------------------g	r--=>----------------g
r--------------------g	r--=>----------------g
r--------------------p	r--=>----------------p
r--------------------p	r--=>----------------p
v--------------------o	v--=>----------------o
ý--------------------n	ý--=>----------------n

2.8 Ukázka rekonstrukce původního řetězce

Rekonstrukce 2 obr. 2.9:

Druhý způsob umožňuje vytvořit originální řetězec pouze z jednoho sloupce. Znovu získáme první sloupec seřazením posledního. Transformovat budeme vždy pomocí prvního a posledního sloupce. Další nebudeme potřebovat. Řazení můžeme začít kdekoliv, ale je nejlepší začít u desátého znaku vlevo (v prvním sloupci), protože víme, že je částí původního řetězce. k tomuto znaku najdeme příslušný znak vpravo, zjistíme o kolikátý znak se jedná (opět číslujeme shora dolů) a k příslušnému stejnému znaku vlevo najdeme znak vpravo. Tento znak nyní doplníme do předposledního sloupce 9 řádku (číslováno od nuly) a k příslušnému znaku vlevo najdeme příslušný znak vpravo a ten opět doplníme před předposlední znak. Tímto způsobem můžeme rovnou doplnit celý původní řetězec. Pokud jste začali vyplňovat např. nultý řádek namísto devátého, nic se neděje stačí jenom zjistit transformaci 10 znaku pravého sloupce tj. začátek původního řetězce. pokud jste transformovali první řádek, stačí od transformace 10 znaku vpravo odečíst 1 a zjistit počátek originálu. Doporučuji si vytvořit ještě jeden sloupec obsahující transformace levých znaků vpravo. Vzniklý řetězec můžete doplňovat pozpátku, nebo jej vypište N-pozice aktuálního znaku. Nezdržujte se zbytečným převracením.

Postupné doplňování řádku:
<pre> --------------------ý a--------------------r a--------------------r a--------------------v a--------------------n g--------------------o g--------------------o m--------------------a m--------------------a n-------------<=-ogram n--------------------a o--------------------r o--------------------r o--------------------m p-------------------- p--------------------a r--------------------g r--------------------g r--------------------p r--------------------p v--------------------o ý--------------------n</pre>

2.9 Ukázka rekonstrukce původního řetězce

Doplňování tedy začíná u desátého znaku vlevo příslušným znakem vpravo je první "m". Najdeme tedy první "m" vlevo. Tento znak tedy doplníme na poslední místo v 10. řádku a jelikož tam už je, přejdeme k prvnímu znaku "m" vlevo. Příslušný znak vpravo je první znak "a". Na místo předposledního znaku v 10. řádku doplníme "a". Tímto způsobem doplníme celý řádek.

2.7.3 Další úpravy po transformaci

Jak už bylo řečeno Burrows-Wheelerova transformace data pouze předpřipraví pro pozdější lepší zkomprimování. Dále se použije úprava pomocí již zmíněného algoritmu MTF. Tímto dostaneme velký počet 0. Dále se použije varianta RLE0 která dobře komprimuje velké řetězce 0. A jako poslední část se použije algoritmus Huffman nebo Aritmetické kódování, pro konečné optimální zakódování dat.

2.8 PPM Prediction by Partial Matching

Metoda PPM je důmyslná moderní metoda, kterou původně vyvinul J. Cleary a I. Witten,rozšířil ji a implementoval A. Moffat. Zkratka PPM znamená „prediction with partial string matching“, tedy predikce se srovnáváním částečných řetězců. Metoda je založena na kodéru, který udržuje statistický model textu. Kodér čte jednotlivé symboly *S*, přiděluje jim pravděpodobnosti *P*. Na základě těchto pravděpodobností, je aritmetický kodér zapisuje na výstup.

2.8.1 Statistický model

Je to nejjednodušší model. Počítá počet výskytů jednotlivých symbolů v minulosti a podle toho přiřadí symbolům pravděpodobnost. Dejme tomu, že zatím vstoupilo a bylo zakódováno 1007 symbolů a 20 z nich bylo písmeno e. Je-li následující symbol q, dostane pravděpodobnost 20/1007 a jeho počet se zvýší o 1. Když se narazí na e příště, dostane pravděpodobnost $31/t$, kde t je celkový počet symbolů, které dosud vstoupily (bez posledního e).

2.8.2 Kontextově orientovaný model (context-based)

Dalším modelem je ***kontextově orientovaný*** (context-based) statistický model. Základní myšlenka je přidělit symbolu *S* pravděpodobnost, která nezávisí jenom na četnosti výskytu symbolu, ale i na předcházejícím kontextu, ve kterém se doposud vyskytl. Např. písmeno h se vyskytuje v textu s pravděpodobností asi 5%. V průměru tedy očekáváme výskyt h asi v 5% času. Je-li však předcházející symbol t, potom je pravděpodobnost výskytu písmena h 45%.

2.8.3 Principy PPM

Hlavní myšlenka PPM využívá tohoto poznatku. Kodér PPM přepne na kratší kontext v případě, že delší vedl na pravděpodobnost rovnou 0. PPM začíná s kontextem řádu *N*. Hledá v jeho datové struktůře předchozí výskyt současného kontextu *C* a následující symbol *S*. Když žádný takový výskyt nenajde (tj. když je pravděpodobnost je nulová), přepne kontext na řád $N - 1$ a pokusí se o totéž znovu. PPM se tedy snaží používat menší a menší části kontextu *C*, což je důvod jeho názvu.

Detailnější postup:

Kodér čte ze vstupu následující symbol *S*, zjistí si na aktuální kontext *C* řádu *N* (posledních *N* přečtených symbolů) a na základě vstupních dat, které viděl v minulosti určí pravděpodobnost *P*, že se za určitým kontextem *C* objeví *S*. Kodér pak zavolá algoritmu(s) adaptivního aritmetického kódování, aby zakódoval symbol *S* s pravděpodobností *P*.

Řád 4	Řád 3	Řád 2	Řád 1	Řád 0
xyzz→x 2	xyz→z 2	xy→z 2	x→y 3	x 4
yzzx→y 1	yzz→x 2	→x 1	y→z 2	y 3
zzxy→x 1	zzx→y 1	yz→z 2	→x 1	z 4
zxyx→y 1	zxy→x 1	zz→x 2	z→z 2	
xyxy→z 1	xyx→y 1	zx→y 1	→x 2	
yxyz→x 1	yxy→z 1	yx→y 1		

Řád 4	Řád 3	Řád 2	Řád 1	Řád 0
xyzz→x 2	xyz→z 2	xy→z 2	x→y 3	x 4
yzzx→y 1	yzz→x 2	xy→x 1	→z 1	y 3
→z 1	zzx→y 1	yz→z 2	y→z 2	z 5
zzxy→x 1	→z 1	zz→x 2	→x 1	
zxyx→y 1	zxy→x 1	zx→y 1	z→z 2	
xyxy→z 1	xyx→y 1	→z 1	→x 2	
yxyz→x 1	yxy→z 1	yx→y 1		

2.10 Vlevo ukázka kontextů a četností řetězce „xyzzxyxyzzx“, vpravo po aktualizaci znakem z

Obrázek 2.10 vlevo nám ukazuje všechny kontexty jednotlivých řádů a četnosti znaků, které po nich následují pro žetězec „*xyzzxyxyzzx*“. Obrázek vlevo nám ukazuje právě zpracovávaný znak z. Pro lepší pochopení činnosti kodéru PPM, předpokládejme že následující symbol je z. Kontext yzzx 4. řádu už byl dříve nalezen, ale nikdy po něm nenásledoval zank z. Kodér proto přepne na kontext 3. řádu, což je zzx, ale ten taky ještě nebyl nikdy následovaný z. Opět přepneme kontext. Následující nižší kontext zx je 2. řádu také neexistuje. Kodér přepne na řád 1 a testuje kontext x. Symbol x se v minulosti vyskytl třikrát, ale vždy byl následován písmenem y. Nyní se zkusí poslední řád 0, kdy z má četnost výskytu 4 (z celkového počtu 11). Takže do adaptivního aritmetického kodéru se zašle z, aby se zakódovalo s pravděpodobností 4/11.

Dekódování:

Je zde základní rozdíl mezi činností kodéru a dekodéru. Kodér se vždy může podívat na následující symbol, a další krok upravit podle tohoto symbolu. Úlohou dekodéru je najít, který ten příští symbol je. Podle toho, jaký ten příští symbol je, se kodér může rozhodnout přepnout na kratší kontext. To dekodér dělat nemůže, protože neví, jaký je následující symbol. Algoritmus potřebuje nějaký mechanizmus, kterým by to mohl určit. Vlastnost, kterou používá PPM je vyhradit jeden symbol abecedy pro *symbol změny* (escape). Když se kodér rozhodne přepnout na kratší kontext, tak nejdříve zapíše do výstupního toku symbol změny (zakódovaný aritmeticky). Dekodér symbol změny může dekódovat, protože je zakódován v současném kontextu. Po dekódování symbolu změny přepne dekodér také na kratší kontext.

2.8.4 Implementační detaily

- Pro ukládání jednotlivých řádů kontextů je nutné vybrat vhodnou datovou struktůru. Datová struktůra by měla splňovat dvě hlavní kritéria: rychlost hledání a velikost zabírané paměti. Proto se jako nejvhodnější používá imlementace pomocí stromové struktůry trie. Trie ukládá úsporněji kontexty v paměti (využívá vlastnosti, že jednotlivé kontexty mají stejné předpony) a současně díky své stromové struktůře, je čas hledání v ní, roven délce kontextu.
- Pro zlepšení účinnosti PPM se používá metoda elimininace. Při přepnutí kontextu, se v následujícím kontextu neberou v úvahu (eliminují se) znaky, které se vyskytly v kontextech předcházejících. Důvod tohoto počínání je prostý, v předcházejících řádech kontextu se znak nevyskytl a proto tyto znaky nemusíme brát v potaz při výpočtu pravděpodobnosti výskytu znaku. Výsledkem je, že se pravděpodobnost výskytu znaku výrazně zvětší.
- Velikost maximálního řádu kontextu, který predikujeme, se nastavuje nejčastěji na hodnotu 5 nebo 6. Různým testováním se zjsistilo, že množství spotřebované paměti ku velikosti kompresního poměru, se pohybuje optimálně právě mezi těmito hodnotami. Při volbě vyšších maximálních řádů kontextů, dosáhneme malého zlepšení a paměťové nároky velice rychle narůstají.

2.8.5 Různé varianty PPM

Jednotlivé varianty se od sebe liší, tím jakým způsobem přidělují váhu symbolu pro přepínání kontextu.

PPMA:

PPMA přidělí symbolu escape pravděpodobnost $1/(n + 1)$. To je ekvivalentní případu, kdy bychom mu vždy přiřazovali počet 1.

PPMB:

Přidělí pravděpodobnost symbolu *S*, následujícímu za kontextem *C* pouze poté, co *S* bylo nalezeno v kontextu *C* dvakrát. To se provádí odečtením jedničky od frekvenčního počtu.

PPMP:

PPMP je založena na odlišném principu. Pokládá každý výskyt symbolu za oddělený Poissonovský proces. PPMP předpokládá, že se symbol *i* objevuje v souladu s Poissonovým rozdělením s (průměrnou) očekávanou hodnotou λi.

2.9 Speciální komprimace

Do tohoto názvu by se dal zahrnout, jakýkoliv postup, který využívá určitý specifický tvar vstupních dat. Tímto způsobem můžeme soubor 1Mb nul zkomprimovat klidně na 1b. Podmínkou je, aby ten soubor opravdu obsahoval 1Mb nul. Při vstupu jiných dat tato komprimace se v podstatě míjí účinkem.

2.9.1 Statická

Když se v datech na stejném místě vyskytuje vždy stejná posloupnost znaků, můžeme tyto znaky nahradit jedním zástupným symbolem. Tohle můžeme uplatnit pouze na specifických datech, proto se jí říká statická. Například první řádek obrazových dat tvoří vždy bílá barva, můžeme ho rovnou vynechat. Dekompresor pak automaticky tento řádek vytvoří a zbytek dat dekóduje.

2.9.2 Delta kódování

Není komprimační metoda, ale způsob úpravy souboru pro jeho pozdější lepší komprimaci. Využívá vlastnosti, že jednotlivé pixely, které spolu sousedí, se liší jen málo. Proto nám stačí si zapamatovat pouze jejich rozdíl a tím zvýšit redundantost dat a tím i jejich lepší komprimovatelnost.

2.9.3 Lineární interpolace

Stejně jako delta kódování, data nekomprimuje, pouze připravuje ke komprimaci. Lineární interpolace předpokládá velký výskyt klesajících nebo rostoucích hodnot dat. Stačí jí zapamatovat si parametry interpolační funkce , díky které dopočítá rozdíly jednotlivých hodnot. V podstatě je to Delta kódování, ale dopočítávání rozdílů se provádí na více než dvou znacích.

3 Návrh

3.1 Použité prostředky

Jako programovací jazyk jsem zvolil C++. Pro čtení DICOM obrázků a tvorbu jednotlivých modulů, jsem použil program MDTSK vyvíjený na fakultě FIT. K automatickému testování jsem použil emulátor linuxového prostředí CYGWIN. Skripty pro automatické testování jsou napsány pro interpret BASH. Jednotlivé algoritmy jsou umístěny v knihovnách. Pro testování algoritmů jsem použil systém jednotlivých modulů. Každý modul jeden algoritmus a jeho modifikace. Jednotlivé moduly čtou data ze standartního vstupu a posílají je na standartní výstup.

3.2 Vybrané algoritmy

Zde následuje seznam vybraných algoritmů, které jsem se rozhodl implementovat.

RLE - tato metoda tu rozhodně nemůže chybět, díky tomu, že u medicíských dat je výskyt stejných obrazových bodu velmi pravděpodobný. Pokusím se zde ukázat, rozdíl mezi komprimací klasickou *RLE, RLE0,* a *RLE* jak je použita v modemech.

Huffmanovo kódování – nemůžeme opomenout klasického zástupce statistické komprimace. Ač je artitmeticé kódování lepší než huffman, je náročnější na implementaci. Zatímco huffman je na implementaci jednodušší, i když ne vždy úplně optimální.

BWT – porovnáme jak moc bude tato metoda úspěšná. Její nesporná výhoda se projevuje především na textových datech, proto uvidíme, jak si povede na medicínských obrázcích.

PPM – tato metoda zde taky nemůže chybět, je hodně univerzální a pracuje dobře pro širokou škálu různých vstupních dat.

MTF – tuto metodu zde nelze opominout, protože je jedním ze stavebních kamenů komprese, při spolupráci s BWT

Speciální komprimace – zde zkusíme některé postupy, při kterých se pokusíme zlepšit kompresní poměr využitím znalostí, které o medicínských datech známe. Patří sem Delta kódování, potom některé úpravy jako třeba: rozdělení dat na sudé a liché bajty a každou část komprimovat odděleně.

4 Implementace

Všechny algoritmy, které jsou zde zmiňovány, jsou implementovány jako blokové. Proto jsou pro všechny jednotně v rámci srovnání zvolen blok 1Mb.

4.1 RLE

Zahrnuje implementaci všech tří variant. U varianty, která se používá v modemech, jsem zvolil komprimaci až od tří po sobě jdoucích stejných výskytů znaků.

4.2 Huffmanovo kódování

Je implementována jeho polostatická 8 bitová varianta . Zatímco v dynamické variantě jednou vytvoříme strom a pak pomocí určitých úprav, pouze udržujeme jeho správnou podobu po každém přidání nového znaku. V této polostatické variantě sestavujeme celý strom vždy znovu. Jelikož tato operace je náročná na čas, provádí se jednou za 2^(bitová varianta) znaků (v našem případě 256). Podle pár testů je rozdíl mezi dynamickým huffmanem a touto verzí pouze setiny procent. I po čtyř násobném zvýšení doby sestavení stromu, se rozdíl pohybuje, oproti dynamické verzi v řádech desetin procent.

4.3 BWT

Základním kamenem této metody je algoritmus, který co nejrychleji setřídí vytvořenou matici. Pro třízení jsem zvolil algoritmus *Merge Sort*, který zaručenou dobu běhu NlogN. Nejrychlejším zástupcem ze své skupiny je sice *Quick Sort* , ale ten má při nevhodných datech kvadratickou dobu běhu. Při nejlepší implementaci je standartní *Quick sort* pouze 2x rychlejší než *Merge sort*.

Pro odstranění vlivu dlouhé doby řazení, pokud se v datech nachází velké množství stejných po sobě jdoucích znaků, používám napřed metodu RLE, kterou tento nedostatek eliminuji.

4.4 MTF

Tato metoda je implementována hned v několika variantách. První varianta, která přesouvá symboly na počatek zásobníku. Druhá varianta, která na poprvé přesouvá symboly těsně před počátek a při opětovném výskytu symbolu ho přesune na začátek. Třetí varianta, stejně jako druhá, přesouvá symbol těsně před počátek, ale pouze pokud předcházející symbol nebyl na počátku, přesouvá znak na počátek.

4.5 PPM

Vhodným kandidátem pro uložení jednotlivých řádů kontextů je *USS* (univerzální slovníková struktůra). Je to stromová struktůra, označována jako jedna z variant *Trie*. Pro jednosušší implementaci jsem zvolil variantu, kdy místo aritmetického kodéru používám huffmanovo kódování. Výsledek proto bude podávat horší výkony.

Přepínání kontextu je zde řešeno trochu jinak. Místo vyslání speciálního znaku při každém přepnutí kontextu, existuje tolik speciálních znaků, kolikrát je možno kontext přepnout. Poté se vždy vyšle daný speciální znak, podle toho, kolik bylo nutné přepnout kontextů.

Dále je zde implementována, eliminace stejných znaků při přepínání kontextů. Tato eliminace zlepšuje účinnost algoritmu.

4.6 Jiné použité techniky

4.6.1 Delta kódování

Je implementováno jak již bylo popsáno výše. Ukládají se pouze rozdíly sousedních symbolů. První bit určuje znaménko a zbývající, jak veliký je rozdíl. Ač jsou medicínská data 16 bitová, je implementováno pouze 8 bitové delta kódování. Jeli rozdíl větší než v rozsahu <-128,127>, pak je uložen na dvou bajtech.

4.6.2 Rozdělení na sudé a liché

Jak již víme, medicínská data jsou 12 bitové odstíny šedi a v souboru jsou uloženy na 16 bitech. První 4bity jsou nevyužity, čímž první bajt dosahuje maximální hodnoty 16. S využitím tohoto poznatku můžeme soubor přepsat tak, aby nejprve po sobě následovali všechny liché bajty a za nimi všechny sudé. Díky této transformaci, by jsme po následném aplikování delta nebo MTF kódování, měli dosáhnout lepší komprimovatelnosti souboru.

4.6.3 BME (Binary Matrix Exchange)

Můj vlastní vynález. Metoda sepíše za sebou nejdříve všechny 8bity pak 7bity … . V podstatě je to převrácení matice o rozměrech 8x(velikost souboru). Například pokud soubor obsahuje znaky o hodnotách v rozmezí 0-15, potom po aplikaci této metody bude první polovina souboru samé nuly. To vyhovuje metodě rle a některé z jejích variant.

5 Testování

5.1 Postup a parametry

Všechny moduly zpracovávají jednotně data po blocích velikosti 1MB. Maximální operační paměť, kterou by měli moduly při svém běhu obsadit, je 64MB. Pro testování jednotlivých modulů a jejich kombinací jsem použil sadu vlastních testovacích scriptů. Testování je zaměřeno výhradně na zjištění kompresního poměru, protože jednotlivé moduly jsou relativně stejně rychlé. Jedinou výjimku tvoří algoritmus PPM, který je výrazně pomalejší než ostatní. Ukazately pro zhodnocení kompresního poměru jsou procenta a BPC (Bit Per Charakters).

Na jednotlivých testech si zkusíme ukázat, jak velkou mají samostatné algoritmy, komprimační účinnost a jakou účinnost mají jejich kombinace. Dále si ukážeme, jak moc se dá zlepšit kompresní poměr s využitím znalosti o medicínských obrazových datech.

5.2 Testy

5.2.1 Základní sady algoritmů

Algoritmus	%	BPC
RLE	66.035	5.282
rle_modem	66.458	5.316
rle_zero	66.000	5.280
Huffman	55.950	4.476
PPM	44.250	3.540
BWT_mtf_rle0_huffman	42.658	3.412
BWT_mtf2_rle0_huffman	42.205	3.376

Zde vidíme vliv základních samostatných algoritmů na kompresní poměr. Nejlépe si zde vedla metoda PPM. V tabulce chybí porovnání s dalším významným zástupcem, a to je BWT. Jelikož BWT je skupina algoritmů, zkusíme v následujícím testu najít její nejlepší implementaci.

Algoritmus	%	BPC
BWT_mtf_bme_rle0_huffman	40.730	3.258
BWT_mtf2_bme_rle0_huffman	40.338	3.227
BWT_bme_rle0_huffman	43.320	3.465
BWT_bme_rle_huffman	43.175	3.454
BWT_mtf_rle0_huffman	42.658	3.412
BWT_mtf2_rle0_huffman	42.205	3.376

Metoda BWT jak již bylo řečeno se používá v kombinaci s jinými algorimtmy. Proto je název obsahuje současně pořadí aplikování jednotlivých algoritmů. Z tabulky lze vyčíst že Burrows-Wheelerova transformace má lepší výsledek než PPM. To je zapřičiněno pravděpodobně používáním huffmanova kodéru namísto aritmetického. Dále se zde ukazuje, že lepší výsledky poskytuje v kombinaci s MTF2 než MTF.

Dále zkusíme nějaký vlastní postup. Víme, že data jsou 16bitová a vrchních 8bitů nabývajích malých hodnot. Proto zkusíme všech vrchních 8bitů dát za sebe a všech spodních 8bitů. Následuje tabulka testů na takto upravené souboru:

Algoritmus	%	BPC
mtf_rle0	51.654	4.132
mtf_rle0_huffman	41.620	3.329
mtf_huffman	47.576	3.806
mtf_bwt_mtf2_bme_rle0_huffman	44.044	3.523
PPM	49.019	3.921
PPM_rle	44.883	3.590

Je vidět že kromě Burrows-Wheelerovy transformace a metody PPM, kterým změny zhoršily komprimovatelnost souboru, se ostatním algoritmů dařilo lépe než v předcházejících testech. Zhoršení v případě Burrows-Wheelerovy transformace a metody PPM lze vysvětlit tím, že pracují na základě statistického modelu dat. Samotná úprava samotný statistický model mění, proto je výsledek většinou horší.

5.2.2 Delta kódování

Nyní zkusíme využít faktu, že jednotlivé body poblíž sebe jsou si podobné. Zkusíme uložit pouze jejich rozdíly ve snaze dosáhnout lepší komprimovatelnosti. Použijeme 16bitové delta kódování, které ukládá rozdíl do dvou bajtu a v případě že se tam nevejde, do do čtyř bajtů.

Algoritmus	%	BPC
RLE	67.305	5.384
rle_modem	67.207	5.376
rle_zero	66.939	5.355
Huffman	51.477	4.118
PPM	45.396	3.631
PPM_rle	42.081	3.366
BWT_mtf_bme_rle0_huffman	42.680	3.414
BWT_mtf2_bme_rle0_huffman	42.282	3.382

Tabulka základních algoritmů a nejlepší metody BWT

Algoritmus	%	BPC
remixdata_rle0	66.736	5.338
remixdata_rle0_huffman	41.127	3.290
remixdata_huffman	46.207	3.696
remixdata_bwt_mtf2_bme_rle0_huffman	43.342	3.467
remixdata_PPM	49.258	3.940
remixdata_PPM_rle	46.212	3.697

Tabulka algoritmů po rozdělení dat na vyšší a nižší bajty

Z jednotlivých výsledků je patrné, že nejlépe si vedly *BWT*, *PPM_rle*, a vlastní algoritmus *remixdata_rle0_huffman.* Ostatní vykazují většinou horší výsledky, ale v některých případech jsme zaznamenali i zlepšení (např. Huffman).

6 Závěr

Ač podle teorie by PPM měla mít nejlepší kompresní poměr, tak metoda BWT ji předbíhá. Je to způsobeno volbou Huffmanova kodéru místo Aritmetického kodéru. Dále jsme zjistili, že díky velkému výskytu nul dosahuje varianta RLE0 nejlepší účinnosti. Následný pokus o zlepšení kompresního poměru pomocí *delta kódování,* přinesl mimo očekávání ve většině případů horší kompresní poměr. Jednou z výjimek byla čisté huffmanovo kódování, to dosáhlo po upravení dat lepšího kompresního poměru.

Další možné rozšíření práce je možné ve vylepšení stávajících algoritmů. Doplnění PPM o aritmetický kodér nebo i samotné *aritmetické kódování* by mohlo dávat lepší výsledky. Dále modifikování stávajícího huffmanova alogritmu na jeho dynamickou verzi, by mohlo přinést určité zlepšení. V neposlední řadě taky nahrazení nebo vylepšení delta kódování, které bude vždy zlepšovat kompresní poměr. Komerční programy pro komprimaci medicínských dat dosahují 3.9 – 3.0 násobku zmenšení souboru. Mě se podařilo dostat nejlépe na násobek 2.479 s metodou *BWT_mtf2_bme_rle0_huffman*.

Literatura

[1] Piotr Wróblewski. *Algoritmy, datové struktury a programovací techniky*. Computer Press, 2004. ISBN 80-251-0343-9.

[2] David Morkes. *Komprimační a archivační programy*. Computer Press, 1998. ISBN 80-7226-089-8.

[3] Robert Sedgewick. *Kalgoritmy v C – datové struktůry, třídění, vyhledávání,* Soft Press, 2003. ISBN 80-86497-56-9.

[4] WWW stránky. Bezeztrátová komprimační algoritmy *http://www.compression-links.info/Lossless* dostupná v květnu 2007

[5] WWW stránky. Bezeztrátová komprimace *http://en.wikipedia.org/wiki/Lossless* dostupná v květnu 2007

[6] WWW stránky. Algoritmus LZW *http://atrey.karlin.mff.cuni.cz/~tnovak/compress/lz78/* dostupná v květnu 2007

[7] WWW stránky. Algoritmus BWT *http://atrey.karlin.mff.cuni.cz/~tnovak/compress/blocksort/* dostupná v květnu 2007

[8] WWW stránky. Standard DICOM *http://www.sph.sc.edu/comd/rorden/dicom.html* dostupná v květnu 2007

[9] WWW stránky. Algoritmus PPM *http://www.cs.waikato.ac.nz/~ihw/papers/95JC-WT-IHW-Unbound.pdf* dostupná v květnu 2007

[10] WWW stránky. Algoritmus Aritmetické kódování *http://www-users.rwth-achen.de/eric.bodden/files/ac/ac_en.pdf* dostupná v květnu 2007

Seznam příloh

Příloha 1. Ukázka celého postupu aritmetického kódování

Příloha 2. Ukázka medicínského obrázku

Příloha 3. CD s moduly, knihovnou algoritmů, a testovacími skripty.

Příloha č.1

Celý postup aritmetického kódování řetězce SWISS MISS

Znak		Výpočet low a high
S	L	0,0 + (1,0 – 0,0) x 0,5 = 0,5
	H	0,0 + (1,0 – 0,0) x 1,0 = 1,0
W	L	0,5 + (1,0 – 0,5) x 0,4 = 0,70
	H	0,5 + (1,0 – 0,5) x 0,5 = 0,75
I	L	0,7 + (0,75 – 0,70) x 0,2 = 0,71
	H	0,7 + (0,75 – 0,70) x 0,4 = 0,72
S	L	0,71 + (0,72 – 0,71) x 0,5= 0,715
	H	0,71 + (0,72 – 0,71) x 1,0= 0,72
S	L	0,715 + (0,72 – 0,715) x 0,5= 0,7175
	H	0,715 + (0,72 – 0,715) x 1,0= 0,72
▫	L	0,7175 + (0,72 – 0,7175) x 0,0= 0,7175
	H	0,7175 + (0,72 – 0,7175) x 0,1= 0,71775
M	L	0,7175 + (0,71775 – 0,7175) x 0,1= 0,717525
	H	0,7175 + (0,71775 – 0,7175) x 0,2= 0,717550
I	L	0,717525 + (0,71755 – 0,717525) x 0,2= 0,717530
	H	0,717525 + (0,71755 – 0,717525) x 0,4= 0,717535
S	L	0,717530 + (0,717535 – 0,717530) x 0,5= 0,7175325
	H	0,717530 + (0,717535 – 0,717530) x 1,0= 0,717535
S	L	0,7175325 + (0,717535 – 0,7175325) x 0,5= 0,71753375
	H	0,7175325 + (0,717535 – 0,7175325) x 1,0= 0,717535

Příloha 2.

Ukázka medicínského obrázku: