Univerzita Hradec Králové

Fakulta informatiky a managementu

Bezztrátové komprimační algoritmy

Bakalářská práce

Dušan Saiko **duben 2004**

Univerzita Hradec Králové

Fakulta informatiky a managementu

Bezztrátové komprimační algoritmy

(s příklady v jazyce Java)

Bakalářská práce

Autor: **Dušan Saiko**
aplikovaná informatika
Vedoucí práce: **Ing. Stanislav Mikulecký**

Dušan Saiko **duben 2004**

Prohlášení

Prohlašuji, že jsem tuto závěrečnou práci vypracoval samostatně a s použitím uvedených informačních zdrojů.

V Hradci Králové dne 30.4.2004 _____

Dušan Saiko v.r.

Poděkování

Děkuji tímto vedoucímu práce ing. Mikuleckému za možnost výběru vlastního tématu bakalářská práce, za kreativní prostor, který mi v tvorbě práce dal, a za trpělivost s mými změnami a při neplnění původních termínů. Z vlastní zkušenosti vím, že ne vždy a na všech vysokých školách je takový přístup běžný.

Anotace

S kompresí dat se setkal snad každý, kdo někdy pracoval s omezenými disketovými mechanikami. Téma kompresních algoritmů mne již dlouhou dobu lákalo a jako téma své bakalářské práce jsem si kompresní algoritmy vybral i z důvodu akademičnosti problému – chtěl jsem něco prozkoumat, naučit se, vyzkoušet, popřípadě zkusit rozšířit; bakalářská práce, jejíž smyslem by bylo „jen“ vytvořit aplikační řešení nějaké úlohy, by mne neuspokojila.

Existenci dnešní civilizace si nelze představit bez ukládání a přenosu dat v elektronické podobě. Komprimace dat není jen o tom, jak zmenšit obsah jednotlivých souborů, ale pojednává o obecném problému, jak, pokud možno efektivně, uložit informace. Pojmy „komprimace dat“ a „kódování dat“ jsou si synonymem. Prostudování a pochopení práce s informací mi velice pomohlo při chápání celkové informační teorie a jsem rád, že jsem se tomuto tématu mohl věnovat.

Práce obsahuje úvod do teorie informace a práce s daty, přehled jednotlivých vybraných bezztrátových komprimačních algoritmů s jejich využitím a implementací v jazyce Java, testy algoritmů a srovnávací tabulky výsledků komprese. Na závěr jsou uvedeny myšlenky možného budoucího vývoje a moderních trendů v datové kompresi.

Annotation

Lossless data compression algorithms (with samples in Java)

Everybody, who (in history) used the limited space of floppy drives, has probably met with the data compression. I have had the topic of data-compression in my mind for a long time already, and I have chosen this to be the topic of my bachelor's work also for academic characteristic of the problem – I wanted to search something, to learn something; bachelor's work with subject which would be „just“ to create application solution of particular task, would not satisfy me.

We can not image any more the existence of our civilization without storing and transferring the data in electronic form. The data compression does not deal only with decreasing the size of particular file, rather it deals with the common problem how to store any information, hopefully in effective way. Therefore the names „data compression“ and „data coding“ are synonyms. Studying and understanding the work with data information helped me a lot with understanding of the whole information theory, and I am glad that I could work on this topic.

The work contains the introduction into information theory and overview of basic lossless compression algorithms, with implementation and samples in Java, also tests of achieved compression results and comparison tables. The conclusion of the work deals with the ideas of the feature evolution of the compression techniques.

Obsah

Úvod

Cílem práce je prozkoumat oblasti informační teorie související s kódováním a bezztrátovou kompresí dat, vyzkoušet si běžné postupy, popřípadě navrhnout nové, algoritmy otestovat a porovnat, a vyvodit z celé práce závěr o možnostech a dalším rozvoji kompresních metod.

Jednotlivé algoritmy jsou otestovány v jazyce Java, buď vlastní implementací, nebo u složitějších kompresních algoritmů, implementací dostupnou z internetu. Algoritmy jsou zpracovány jako vstupně-výstupní transformační streamy. Úvod do implementace kompresních algoritmů pak pojednává o využití takto navežené komprese pro komprimaci adresářové struktury a pro šifrování komprimovaného archivu.

Testy jsou prováděny na standardizované sadě souborů, na kterých je výsledek komprimace možno porovnat s výsledky různých dalších algoritmů publikovaných na internetu.

Porovnání účinností jednotlivých algoritmů je doplněno o kompresi provedenou dvěma komerčními kompresními programy – WinZip 9.0 a WinAce 2.2.

Práce končí vyhodnocením kompresních algoritmů, úvahou nad zjištěnými vlastnostmi komprese a nad dalším možným vývojem kódování a spravování informací.

Práce je vhodným čtením i pro informatiky začátečníky, kterým by po přečtení měla dát základní ale jasný přehled o teorii informace a algoritmech používaných pro její kompresi.

Součástí práce je CD s kompletními okomentovanými zdrojovými kódy a testovacími skripty implementací.

Vlastní přínos

Při navrhování a implementaci algoritmů jsem se snažil v co největší míře použít a vyzkoušet vlastní myšlenky a postupy, u jednodušších algoritmů jsem zkoušel odvodit teoretické pozadí možné komprese. Všechny teoretické příklady pracující se statistikami jsou 'lokalizovány' a předělány na podmínky českých statistik získaných ze souboru českých textů.

I. Teorie komprese dat

Shannonova teorie informace

Většina věcí má svůj počátek, ať se jedná o konkrétní událost, nebo abstraktní vývoj. Také komprese dat a vznik všech kompresních algoritmů určených k ukládání a přenosu dat má svůj základ postavený na práci Clauda Shannona, který jako první pojmenoval a konkretizoval problém přenosu a komprese informací.

Civilizační přínos teorie informace

[1]Teorie informace je jedním z mála vědních oborů, u které je možné prakticky na den přesně stanovit datum jejího vzniku. Uveřejnění práce Clauda Elwooda Shannona "Matematická teorie komunikace" [Shannon 1948] v Bell System Technical Journal v roce 1948 vymezuje jasně počátek teorie informace.

Shannon byl geniální inženýr s velkým citem pro matematiku a s ohromující schopností pronikat k jednoduchému jádru napohled velmi složitých problémů. Informaci pojal jako veličinu nezávislou na jejím hmotném nosiči a oddělil ji také od sémantického obsahu; tj. nezahrnul do tohoto pojmu důsledky, které má zpráva pro konkrétního adresáta. Informaci v datech, v signálu či zprávě jakéhokoliv druhu vymezil ve vztahu k jiné, přesně definované veličině (stavu, zvuku, obrazu) jako míru schopnosti o této veličině vypovídat, lokalizovat ji, snížit její neurčitost. Pro jednotku nové veličiny - informace - navrhl Shannon výraz bit. Prvním problémem teorie informace, který Shannon popsal, je stanovení mezních možností redukce výstupů z informačních zdrojů bez podstatného omezení informace v nich obsažených a hledání cest k jejich praktickému dosahování.

Podobně jako informaci i komunikační kanál Shannon oddělil od hmotných realizací a pojal ho jako abstraktní matematický objekt: náhodnou transformaci vstupních zpráv na zprávy výstupní. Náhoda v tomto modelu představuje šum - rušivé vlivy, kterým jsou zprávy téměř vždy v nějaké míře vystaveny. Druhým shannonovským problémem teorie informace je stanovení informační propustnosti - kapacity - komunikačního kanálu a hledání cest k jejímu

[1]Část textu je převzata z článku
Igor Vajda – Akademický buletin 4/1999
http://www.kav.cas.cz/press/stranky/archiv/ab/ab1999/ab9904/vajda.htm
Použití a změny textu proběhly po předchozím svolení autora.

dosahování, tj. k co nejlepšímu využívání kanálu. Kapacita charakterizuje maximální rychlost přenosu informace kanálem, při které lze ještě kompenzovat rušivý vliv šumu. Kompenzace se uskutečňuje nikoliv zvyšováním výkonu vysílacího zařízení, ale pouze vhodnými manipulacemi - přetvarováním přenášených zpráv, které umožňuje v přijaté zprávě opravit, anebo alespoň na postačující úroveň potlačit změny vyvolané šumem.

Další významný problém Shannon zformuloval v práci uveřejněné v roce 1950. Jde o problém svým způsobem opačný k předešlému: jak přetvarovat libovolnou zprávu, případně jak pomocí určitého klíče vytvořit kanál, aby ze zprávy na výstupu kanálu nebylo bez klíče možno rekonstruovat původní zprávu. Touto prací Shannon položil základy nové větve teorie informace – kryptografie.

Vedle problematiky vzniku, kódování a přenosu informace, se kterou přišel Shannon, zaujala důležité místo v teorii informace i problematika zpracování informace. Teorie informace umožnila zcela novým a hlubším způsobem pochopit pojem informace vypracovaný ve 30. letech R. A. Fisherem v matematické statistice, což je tradiční matematická disciplína zaměřená na zpracování informace. Pod vlivem teorie informace se sféra zájmu matematické statistiky rozšířila na oblasti, kde je informace reprezentována složitými datovými komplexy, jako jsou signály a obrazová pole. Pod vlivem Shannonova pojetí vzniklo v průběhu 50. let zcela nové chápání vzdáleností mezi pravděpodobnostními distribucemi dat. Tyto nové představy a pojmy sehrály v následujících dekádách klíčovou roli při hledání univerzálnějších metod odhadování distribucí, jejich parametrů a testování hypotéz o distribucích i parametrech. Metody zpracování informace, které jsou přímo založené na pojmech a výsledcích teorie informace, se někdy shrnují pod názvem statistická teorie informace.

Uvedené čtyři problémové okruhy tvoří dodnes základ teorie informace. Dílčí řešení za určitých speciálních podmínek našel sám Shannon, další řešení naznačil. Obecnější řešení se nacházela postupně. Zprvu v návaznosti na postupný rozvoj matematických metod a později v návaznosti na potřeby technické praxe. Po relativně mírném rozjezdu v 50. a 60. letech nastal bouřlivý rozvoj teorie informace v dekádách následujících. Prvotní tlak na její rozvoj vytvořily projekty vojenské (přenos dat mezi počítači na souši, ve vzduchu i na vodě), kosmické (přenos dat ze vzdálených vysílačů napájených slabými bateriemi), a zdokonalování počítačů (korekce chyb při čtení informace z pamětí, komprese informace při ukládání do

pamětí, např. pomocí Zip-Lampelova informačně-teoretického algoritmu). Záhy však nastoupil mnohem mohutnější tlak projektů akademických a komerčních (vytváření počítačových sítí, digitální záznam a přenos řeči, hudby a obrazu, například HI-FI kvalita hudby na CD by nebyla možná bez korekce chyb vzniklých při zápisu a čtení z disku). Počet publikací a specializovaných pracovišť prudce vzrostl. V poslední době je největší zájem o teorii informace v kruzích budujících integrované sítě pro přenos dat. Nadále však trvá zájem kosmického výzkumu, jak o blízkou komunikaci (lepší využívání kapacit kanálů na existujících komunikačních družicích, budování dokonalejších nových družic), tak i o dálkovou. Rozsáhlé komerční aplikace má také kryptografie - moderní bankovnictví si neumíme představit bez známých PIN-kódů a elektronických podpisů. Známá metoda šifrování pomocí tzv. veřejných klíčů, která se zde používá, vznikla na základě teorie informace.

Úvod do teorie informace

Co je to informace ?

Podle Příručního slovníku naučného [Slovník 1962] :

"**Informace,** lat. zpráva, popř. její obsah, smysl. V kybernetice a vědách příbuzných poučení o něčem, sloužící k vzájemnému styku živých a neživých systémů přírody. Je zde chápána v obecném matematickém smyslu."

Informace může být chápana jako sdělení poznatku a veškerý evoluční, civilizační i technologický vývoj je postaven na shromažďování, předávání a vyhledávání informací.

Informační teorie řeší z technického hlediska procesy sběru, přenosu, zpracování a uložení informace. Těmito procesy lze popsat události z technického ale i reálného světa včetně mezilidské a sociální komunikace člověka s okolím.

Každá informace má svůj počátek (záblesk světla, slovo, mechanické, chemické či elektronické jevy), nějak se šíří (nosičem, který můžeme nazvat signálem), při přenosu může být informace ovlivněna mnoha faktory a její originální kvalita může být narušena (platí opět v technickém i sociálním významu), informace může mít svého adresáta (konkrétního, neurčitého, i sebe sama) a může být adresátem různým způsobem zpracována a uložena (archivována).

Jednoduchým příkladem obecného informačního procesu mohou být:

- veškeré evoluční i sociální informace, jejich předávání a používání
- zprávy v jazykovém slova smyslu
- veškeré chemické reakce působící na člověka (city, pocity, emoce, pudy, instinkty, včetně procesu dýchání, krevního oběhu atd.)
- světelné, zvukové, mechanické a chemické procesy

a jako konkrétní příklady informací můžeme uvést např.:

- indiánské kouřové signály
- Morseovu abecedu pro usnadnění přenosu informací
- světelná a zvuková informační a výstražná znamení
- přenos rozhlasu, televize, telefonních hovorů a dalších elektronických informací

Výčet aplikací toho, co se dá nazvat "informační teorie", by byl nekonečný a je vidět, že tato teorie jen technickým způsobem popsala, kategorizovala a nově pojmenovala jevy, které provází vesmír i lidskou společnost od svého prvopočátku.

Teorie informace je tedy vědou, která popisuje přenos, uchovávání a vyhledávání informací, součástí teorie informace je i kryptografie, přenos a uchování informací v podobě, v které lze daná informace přečíst jen za určitých podmínek.

Informace je v technickém pojetí signálem s množinou po sobě jdoucích povolených stavů. Claude Shannon přiřadil technické informaci dva základní informační stavy – 1 a 0 a tuto informaci nazval "bit". V počítačové podobě se 8 bitů skládá do jednoho "bajtu", který může přenášet hodnoty 0..255, nebo alternativně významově upravené hodnoty (-128 .. 127, 1000...10255 atd ...). Naproti tomu reálný svět vnímáme jako souslednost analogových jevů, kde jednotlivé veličiny nejsou omezeny svou hodnotou a jsou spojité. Z kvantové teorie ale vyplývá, že každé vlnové šíření má částicově – vlnový charakter a tedy i námi pozorované "spojité" jevy jsou ve skutečnosti složeny z miniaturních stavových veličin pojmenovaných kvantum. Proto informační teorie a reálný svět nejsou v jakémkoliv smyslu v rozporu a záleží jen na výběru vhodného dostačujícího poměru přesnosti kódování reálných signálů do signálů elektronických.

V technické praxi víme, že například obrázek pro zobrazení počítačem je složen z určitého počtu bodů a bod má určitou barvu z dané množiny možných barev. Nebo text je za sebou následující tok znaků, které mohou mít nějakou hodnotu, podle které seskupení znaků

přiřazujeme význam. Text nemá daleko od slova a je dobré si připomenout, že i myšlení lidského mozku je tvořeno ve slovech (v lingvistickém významu slova; člověka který se pohybuje mezi několika různě mluvícími národy se má smysl ptát, "v jakém jazyce jeho mozek myslí").

To, že teorie informace není nějaký technický výmysl, ale že naopak je jednou ze základních filozofických myšlenek bytí a nebytí, potvrzuje i následující úvaha. Jestliže text je seskupení prvků z omezené množiny znaků, toto seskupení lehce můžeme vygenerovat. Můžeme generovat například jen text, který se bude skládat z malých písmen anglické abecedy, mezerou, tečkou, čárkou, otazníkem a vykřičníkem. Množina znaků bude mít tedy "*abcdefghijklmnopqrstuvwxyz .,!?*" 31 znaků. Jestliže průměrný novinový článek je rozsahu např. 80x50, tedy 4000 znaků, všechny možné, existující, neexistující, minulé i budoucí novinové články (včetně nepoměrně velkého množství nesmyslných textů) budou jednou z 2,8E+5965 možností (při použití zmiňované množiny znaků). Je to obrovské nepředstavitelné číslo, zároveň ale nepředstavitelným faktem je to, že v tomto počtu kombinací je opravdu napsané všechno, co kdy v daném rozsahu textu lidstvo mohlo a může vymyslet. V kombinacích jsou zahrnuty přední stránky zítřejších novin, výňatky textů Shakespeara, televizní program na příští rok, slovně popsaná teorie relativity, jednotlivé kapitoly této práce atd.

Tato úvaha nastoluje filozofickou otázku, zda je možná jakékoliv originální činnost lidského mozku, zda celá evoluce vesmíru a lidského druhu je jakkoliv ovlivnitelná nebo daná, a konečně by se dala rozvést i do existenční otázky, jestli člověk může hovořit o "vlastní vůli". (Dále také tato teorie nabourává smysluplnost podstaty autorských zákonů).

Informační teorie má mnoho svých aplikací v dalších vědních oborech, o tématech příbuzných informační teorii by mohlo být napsáno mnoho tisíc stran, včetně dopadu informační teorie a aplikace hromadného využívání informací na lidskou společnost ...

Z vědního hlediska lze aplikace informační teorie rozdělit podle následující tabulky [Vymětal 1999]

informační teorie:	matematická informatika
	kybernetika, umělá inteligence
	lingvistika
	teorie poznání
	některé filozofické disciplíny
informační technologie:	výpočetní technika
	spojovací technika
	teorie signálů
	reprografické techniky
informační funkce a služby:	
	knihovnictví, archivnictví
	databázová centra
	oborové informace
	informační střediska
	informační brokeři

Tabulka 1: rozdělení informační teorie z vědního hlediska

Informační teorie řeší technické postupy pro přenos a uchovávání informací, protože množství informací kolem nás je obrovské a civilizační nároky na využití informací se zvětšují, nedílnou a důležitou součástí teorie informace je efektivní zacházení s informacemi, jak pro jejich přenos, tak pro jejich uchování. Opět je možné uvést paralely z reálného světa.

- Postavený stan můžeme nazvat mechanickým stavem informace pro okolní mechanické a optické jevy. Pro efektivní přenos této informace je ale zapotřebí stan složit, zabalit a v případě potřeby podle návodu zpět složit do své původní podoby.
- Fotografie je efektivně uchovaná optická informace. Neefektivním způsobem uchovávání a reprezentace shodné optické informace by bylo sestavit reálnou scénu shodnou s originální. Reálná scéna by obsahovala mnoho informací které ve fotografii nejsou, například pohled zezadu, mechanické vlastnosti atd, ale

fotografie je značně úspornější a pro své využití obsahuje dostatečné množství informací (analogicky zvukové jevy).

- I fotografie může být někdy nepřiměřeně neefektivní reprezentace informace, a proto jsou například v elektronické komunikaci používány symboly (ikony) vyjadřující zobrazení tváře a přeneseně tak pocity původce informace ("smajlíky").

Tím se tedy dostáváme ke zefektivnění práce s informací – k datové kompresi. Stan byl příkladem bezztrátové komprese, fotografie příkladem ztrátové komprese informací.
O kompresi dat se většinou mluví v případě, že výstup kódovacího algoritmu je méně objemný než originální velikost vstupních dat.

Ve své práci o teorii informace Shannon představil existenci limitu bezztrátové komprese dat *H* a určil, že je roven míře entropie, která je závislá na charakteristice zdroje dat. Každou informace je možné bezztrátově reprezentovat objemem dat blízkým *H*, matematicky není možná lepší komprese dat.

Příruční slovník naučný [Slovník 1962] charakterizuje entropii jako:
„Matematická funkce, jejíž hodnota souvisí s hodnotou pravděpodobnosti dané soustavy tak, že maximum entropie odpovídá stavu nejpravděpodobnějšímu. Entropie vyjadřuje tendenci soustavy přecházet z méně pravděpodobných (uspořádanějších) stavů do stavů pravděpodobněji uskutečnitelných (méně uspořádaných). Entropie zdroje je číselné vyjádření množství informace, které je průměrně obsaženo ve zprávě ze zdroje informace; informační vydatnost.“

Shannon také matematicky popsal teorii ztrátové datové komprese, v které dekomprimovaná data nemusí být naprosto shodná s daty originálními a je tolerována jistá míra zkreslení informace *D*. Shannon ukázal, že pro datový zdroj se známými statistickými charakteristikami a známou mírou zkreslení *D* existuje funkce *R(D),* která určuje nejlepší míru ztrátové komprese datového zdroje. Je li míra zkreslení *D=0*, pak *R(D)=R(0)=H*.

Shannon tak formuloval existující limit pro ztrátovou i bezztrátovou kompresi, jeho teorie neobsahuje postupy, jak tohoto limitu dosáhnout, obsahuje ale některé tipy a pokyny na dosažení maximální komprese.

Shannonův statistický model zdroje dat

Jak Shannon popisuje v *Shannon's 1948 paper* [Shannon 1948], lze statistický model dat simulovat na příkladu s průměrnou, náhodně vybranou knihou, jejíž obsah lze označit symbolem *X* a kniha je tvořena množinou jednotlivých písmen *X1, X2, X3* ...

$$X = \{X1, X2, X3, \ldots\}$$

Z hlediska kódování zdroje dat jsou znaky X1, X2, X3 atd. náhodné znaky z množiny abecedy a celá kniha je náhodným zdrojem dat podle určitých statistických vlastností daného zdroje. I když u následujícího textu "*dob_ý den, mlu_í k Vám pr_zident republi_y*" jsou vynechána některá písmena, ze zkušenosti a znalosti pravděpodobných spojení jednotlivých písmen text bez problémů přečteme, i kdyby jednotlivá slova nebyla napsána v souvislém kontextu slov.

Originální Shannonovy příklady statistik a dat vycházely z anglických textů, pro následující odstavce jsem ale vytvořil české verze statistik, které budou vycházet ze souboru (přiloženého na CD) czech_text.txt, souboru o velikosti přibližně 1.5 MB, v kterém jsou obsaženy archivy českých novinových článků, rozhovorů, texty zákonů, ústavy, politických prohlášení i knihy Bible kralické. Texty jsou převedeny na malá písmena, zbaveny diakritiky a všech znaků kromě písmen anglické abecedy a mezery, bez řádkování.

Abeceda *A* analyzovaného textu obsahuje (všech) 26 abecedních znaků + mezeru

$$A = \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z, \}$$

Pro statistickou analýzu dat budeme používat třídu *saiko.compression.SourceStatistics*, kterou následně využijeme i v konkrétních statistických komprimačních metodách. Třída je schopna získat z dat statistiky pro prvky libovolného stupně (viz dále).

Statistické vlastnosti zdroje mohou být formulovány v následujících modelech.

Model nultého stupně

Každý znak textu je statisticky nezávislý na předchozím znaku a pravděpodobnost výskytu jednotlivých znaků abecedy je stejná. Jestliže by takový model odpovídal realitě, náhodný text knihy by mohl vypadat takto:

qiwrmsmotyotpwuijotyhruroowwurkulmtgxexwryrpxsywzzsnchakczzeqfrtuyjnydmt lcxlhie

Model prvního stupně

Víme, že některé znaky se v textu vyskytují častěji, jiné méně často. Můžeme proto sestavit statistiky tohoto prvního stupně pro náš ukázkový textový soubor a podle statistik nechat vygenerovat ukázkový text.

Statistiky prvního stupně pro daný textový soubor s počtem výskytů příslušného prvku:

" "	236430
"e"	128556
"o"	103838
"a"	102693
"i"	89831
"n"	80038
"s"	65486
"t"	64499
"r"	55082
"v"	53093
"l"	47944
"u"	46603
"d"	46396
"p"	41174
"k"	39786
"c"	38813
"z"	38732
"m"	38579
"y"	30805
"h"	29207
"j"	27442
"b"	23155
"g"	4141
"f"	2395
"x"	1611
"w"	77
"q"	3

Tabulka 2: statistiky prvního stupně z českého textu

sdp aeti iypr yscazaapsuns a lzeoaiavr anull voeio niano sesapaarncizmc kpd os e

Model druhého stupně

Dále následuje úvaha, že například dvojice *zz* se v textu vyskytuje méně častěji, než dvojice *za*. Můžeme proto sestavit statistiky druhého stupně a opět vygenerovat ukázkový text.

"e "	39767
"i "	35277
"a "	27443
" p"	25656
" s"	25284
"o "	21554
"ne"	19598
"u "	19348
"ni"	18881
...	
"uy"	1
"wl"	1
"wu"	1
"wz"	1
"xc"	1
"ye"	1
"yi"	1

Tabulka 3: statistiky druhého stupně z českého textu

vco dcvlseudazstchna od li hvnob aliou toknabyprli a zntkdm l ilzaapstskrotapaim

Model třetího stupně

Statistiky třetího stupně a náhodně vygenerovaný text vypadá následovně:

" pr"	13099
"ni "	10871
" a "	9537
" po"	9088
" ne"	8596
" je"	7362
" na"	7020
"ch "	6904
...	
"zyc"	1
"zyl"	1
"zza"	1
"zzo"	1

Tabulka 4: statistiky třetího stupně z českého textu

telreze usve sl posobnep ob vsmoz po zp naemuusp zejsopovtegne tvoza ti oucve p

Jde vytvořit statistický model jakéhokoliv stupně a je vidět, že se vzrůstajícími stupni jednotlivých statických modelů roste i smysluplnost generovaného textu.

Model prvního stupně je speciálním případem modelu nultého stupně, model druhého stupně je specifickým případem modelu prvního stupně atd. Jestliže bychom zvolili některý vyšší stupeň statistického modelu, například 10, generováním bychom mohli získat následující větu

i se zucaszna stale pske unie dalnicni prtizy budotovat rekldy z nas dni hotelov

Takové nahlížení na zdroj dat jako na náhodný statistický zdroj má mnoho aplikací v různých formách a variantách kompresních algoritmů.

Podle jednotlivých stupňů statistických modelů vypočítal Shannon, pro 27 znakovou anglickou abecedu, následující hodnoty entropie *H*.

model nultého stupně	4.75 bitů na znak
model prvního stupně	4.07 bitů na znak
model druhého stupně	3.36 bitů na znak
model třetího stupně	2.77 bitů na znak

Tabulka 5: hodnoty entropie pro anglickou abecedu dle Shannona

Dále Shannon rozvinul teorii statistického datového zdroje do blokového kódování tohoto zdroje přiřazováním krátkých kódů nejčastěji se vyskytujícím opakováním skupin znaků a delším kódům pro méně častěji se vyskytující prvky. S praktickou aplikací tohoto statistického kódování se setkáme v konkrétních komprimačních algoritmech.

Shannonova teorie statistického kódování dat má následující praktické omezení:

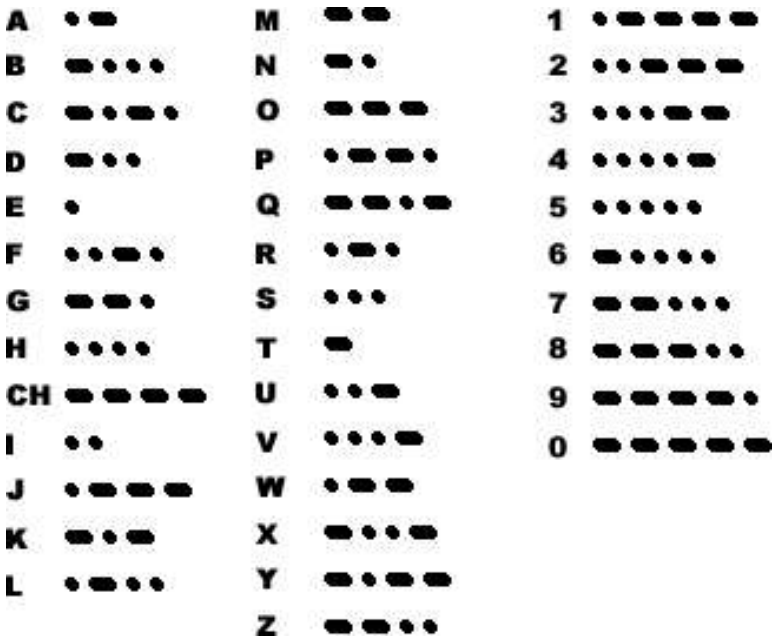
- doba prodlení při čekání na dostatečně dlouhý vzorek dat pro vytvoření statistických informací získávaných z realtimových dat může znamenat omezenou použitelnost postupu, např. pro hlasovou komunikaci
- teorie se nezabývá s velikostí ukládání statistických informací do komprimovaných dat. Čím více roste stupeň statistického modelu, tím více (často exponenciálně) roste velikost struktury tohoto modelu
- ne vždy jsou známy statistické informace zdroje dat

Morseova abeceda

[Morse] První seriózní uměle vytvořenou aplikací tehdy neexistující teorie informace bylo vytvoření Morseovy abecedy. Abecedu vytvořil amatérský fyzik Samuel Morse do soutěže vyhlášené kongresem Spojených Států v roce 1837. Soutěž byla vypsána na zlepšení tehdejší informační soustavy tehdy již známého optického telegrafu. Morse navrhl svůj telegrafní přístroj na elektromagnetickém principu a v září 1837 byl představen nový Morseův telegrafní přístroj. Proti svým současníkům, kteří navrhovali systémy použitím paralelního kódu pro více "datových linek", Morse vytvořil první aplikovaný sériový kód. Pro přenos dat na svém telegrafu vytvořil vlastní abecedu, která je ukázkovou aplikací pozdější Shannonovy teorie statistického kódování dat.

Statistické pravděpodobnosti výskytů jednotlivých znaků přiřadil daným znakům kódy s délkou nepřímo úměrnou četnosti výskytu znaku v anglické abecedě. Abecední kódy byly určeny k zvukové či vizuální interpretaci a kvůli vhodnosti k akustickému zpracování použil do kódů ternární kód (dlouhý signál, krátký signál, ticho).

Morseova abeceda našla velké uplatnění, u vojenských jednotek se používala ještě v polovině 90tých let 20tého století a pan Morse svým přístupem k problému a jeho zpracováním předběhl dobu o více jak 100 let.



Obrázek 1: Morseova abeceda

Rozdělení komprimační algoritmů

Komprimační algoritmy lze rozdělit podle mnoha úrovní, např. do následujících skupin.

Bezztrátové komprimační algoritmy

- umožňují dekódovat informaci do její přesné původní podoby

Ztrátové komprimační algoritmy

- podle druhu dat mění, či vypouští informace o nedůležitých (neviditelných, neslyšitelných) částech dat, data nejdou zrekonstruovat do úplně původního stavu, opakovaná ztrátová komprese se na datech projevuje ztrátou kvality dat, kompresní algoritmy jsou ale velice účinné.
- většina ztrátových kompresních algoritmů je kombinována s neztrátovými kompresními algoritmy pro ukládání výsledku ztrátové komprimace
- v našem přehledu kompresních algoritmů nás pouze zajímají bezztrátové kompresní algoritmy

Rozdělení podle účelu komprese

- přenosové kompresní algoritmy:
  záleží v nich na rychlosti kódování dat
- algoritmy pro archivaci dat:
  mohou více využít výpočetní sílu počítače
- algoritmy pro transparentní komprimaci:
  například diskových svazků, musí umožňovat rychlý přístup do obsahu komprimovaných dat
- algoritmy pro explicitní komprimaci:
  známé komprimační programy vyvolávané příkazovou řádkou

Rozdělení podle návrhu kompresních algoritmů

- statistické algoritmy:
  pracují se statistickými informacemi zdroje dat
- slovníkové algoritmy:
  udržují slovník vyskytujících se spojení

- analytické algoritmy:
  pro ztrátové komprimační metody
  nalézají matematické funkce pro reprezentaci obsahu dat
- aproximační metody:
  pro ztrátové komprimační metody
  nalézají aproximaci hodnot datového obsahu
- numerické algoritmy:
  pracují s kódováním v reálných číslech,
  překonávají bariéru využití zdroje i v desetinných místech jeho entropie (např. kódování znaku na 2.77 bitu)
- ve velké části kompresních algoritmů se charakteristiky jednotlivých postupů prolínají

Obrázek 2: rozdělení algoritmů komprese

II. Přehled základních bezztrátových algoritmů

Implementace kompresních algoritmů v jazyce Java

V následujících kapitolách budou probírány a implementovány příklady jednotlivých kompresních algoritmů.

Jejich implementace je samozřejmě možná v jakémkoliv programovacím jazyce, v této práci budou ale uvedeny všechny kódy v jazyce Java, a to z důvodů, které následují.

Po víceleté praxi s programovacími jazyky Assembler, C, C++, Visual Basic, Pascal/Delphi, Java, mohu o jazyce Java prohlásit:

- je universálním programovacím jazykem s plnohodnotným a neomezeným použitím ve většině programátorských oblastí
- je robustní a stabilní, nemá (větší) technologická omezení
- stále se vyvíjí, objevují se v něm nové charakteristiky vyjadřování (Java 1.5), existují různé zajímavé nadstavby jazyka, kterými se dá Java rozšiřovat (např. http://groovy.codeguards.com)
- obsahuje charakteristiky, které některé ostatní programovací jazyky dohánějí násilnými technologickými řešeními
- čitelnost a přehlednost zápisu programového kódu
- kvalitní vývojové prostředí zdarma
- kvalitní zázemí (dokumentace, rozsáhlá komunita uživatelů)
- obrovské množství použitelných transparentních knihoven
- platformě nezávislý vývoj i cílové nasazení
- má zabudovanou funkčnost pro práci s neomezeně dlouhými celočíselnými i reálnými datovými typy

Vzhledem k implementaci kompresních algoritmů využijeme velice přehledně navržený systém vstupně/výstupních streamů, který lze nezávisle kombinovat, a jednoduše tak dosahovat efektivních výsledků (při komprimaci dat, kryptování, komunikací se síťovými protokoly ...).

Komprimační algoritmy pracují často s čísly na úrovni bitů a bitových operací, zde se můžeme setkat s absencí unsigned (nezáporných) datových typů v Javě, tedy *byte* (bajt*)*

s kterým se pracuje nejčastěji, má vždy rozmezí hodnot -128 ... 127, což může být pro některé binární operace s bajty problémové, proto se často hodnoty typu *byte* převádějí na svou kladnou hodnotu a ukládají do typu *int*. Typ *int* je použit i jako parametr některých funkcí, i když se očekává číslo v rozmezí 0..255.

Další, někdy velmi nepříjemnou, nevýhodou jazyka Java je rychlost (=pomalost) některých algoritmů a nemožnost optimalizace rychlosti kódu na takové úrovni, jak to umožňuje např. C++.

Například u cyklické práce s řetězci nebo dynamicky alokovanými poli docházelo v některých algoritmech k výraznému zpomalení celého kódu.

V kódech jazyce Java bude dostatek (anglických) komentářů na pochopení funkčnosti algoritmů, jazykové konstrukce a technologie Javy (např. streamy), ale nebudou rozebírány a popisovány.

Většina kompresních algoritmů (pokud to způsob použití algoritmu dovoluje) je implementována jako vstupně-výstupní stream, který umožňuje komprimační operace s tokem dat. V terminologii jazyka Java to znamená, že naše objekty budou potomky tříd *OutputStream* (pro kompresi) a *InputStream* (pro čtení komprimovaných dat). Abychom nemuseli implementovat všechny metody tříd *InputStream* / *OutputStream*, budou objekty odvozeny od tříd *FilterOutputStream* / *FilterInputStream,* které zaobalují použití streamu předaného parametrem konstruktoru a implicitně mu předávají všechna volání. Pro třídy odvozené od *FilterOutputStream* budeme nejvíce přepisovat metody *write()* a *flush()*, pro třídy odvozené od *FilterOutputStream* nejvíce metodu *read()*. Jak uvidíme v příkladech, práce se vstupně výstupními streamy je jednoduchá a rychle přináší zajímavé výsledky.

Je ještě dobré poznamenat, že Java má v sobě již základní kompresní knihovny zabudované a podporuje práci s formátem GZip (*GZipOutputStream* a *GZipInputStream*), Zip (*ZipInputStream*, *ZipOutputStream*, *ZipEntry, ZipFile*) a Jar (*JarInputStream, JarOutputStream, JarEntry, JarFile*).

Komprimace adresářové struktury

Práce se vstupně/výstupními streamy je určena pro jednosměrný tok dat a jednoduše tak můžeme číst a zapisovat jednotlivé soubory. V praxi se ale nejvíce setkáváme s komprimací adresářových struktur, a to většinou do jednoho diskového souboru.

Postup při komprimaci adresářové struktury může být buď takový, že nejprve jsou jednotlivé soubory zkomprimovány a pak uloženy do jednoho souborového archivu, nebo takový, že je vytvořen jeden souborový archiv ze všech komprimovaných souborů a tento archiv je posléze celý zkomprimován. Oba dva postupy mohou mít svou logiku, při komprimaci jednotlivých souborů lze využít speciálních vlastností obsahu jednotlivých souborů (text, zvuk atd), v komprimaci až výsledného archivu dochází částečně ke splynutí datových charakteristik všech souborů archivu. Pokud se archiv skládá z velkého množství malých souborů, odpadne tak ale nutnost zapisování informacích o kompresi do každého jednotlivého souboru, což by v prvním případě mohlo kompresní poměr při velkém množství malých souborů znehodnotit.

Jednotlivé historické archivní programy měly většinou svůj formát ukládání komprimovaných souborů do archivu, samy implementovaly získávání informací a zkomprimovaných souborů z archivu.

Jednodušším a použitelnějším způsobem je použít standardní archivační program, který spojí soubory do jednoho souborového archivu, a starat se pouze o samotnou komprimaci datového toku (buď jednotlivých souborů, nebo celého archivu).

Takový obecný a standardní mechanizmus pro vytváření souborových archivů existuje a je dostupný i jako programová knihovna pro Javu. Je jím (starý, dobrý, kvalitní, stabilní, standardní) původem unixový archivační program TAR, popis a knihovny jsou (samozřejmě zdarma) ke stažení na http://www.trustice.com/java/tar/ [TAR].

K funkčnosti knihoven *com.ice.tar.** jsou zapotřebí knihovny *Java Activation Framework*, knihovny mohou být staženy na adrese http://java.sun.com/products/javabeans/glasgow/jaf.html.

S využitím knihoven na *com.ice.tar* lze jednoduše napsat funkci na zabalení adresářové struktury do jednoho *tar* archivu a přitom ještě daný archiv komprimovat, např. metodou GZip.

```
OutputStream output=new GZIPOutputStream(new FileOutputStream("output.tgz"));
TarArchive archive=new TarArchive(output);
archive.writeEntry(new TarEntry(new File("input_dir"),true));
archive.closeArchive();
output.close();
```

Kód 1: příklad archivace adresářové struktury pomocí knihoven com.ice.tar

Získávání souborů z archivu či dekomprimace archivu do adresářové struktury je podobně jednoduchá.

Šifrování komprimovaných dat

K šifrování komprimovaných dat můžeme v Javě použít tříd *CipherOutputStream* a *CipherInputStream*, které jsou k dispozici od Java SDK [Java] verze 1.4. Streamy umožňují šifrovat a zpětně dešifrovat tok dat pomocí certifikátu nebo hesla. Protože z klasických komprimačních programů známe často ochranu archivu souborů heslem, uvedeme si příklad kryptování výstupního datového proudu pomocí hesla. Kryptování toku dat pomocí certifikátu by bylo obdobné, změnila by se jen inicializace objektu *Cipher*.

Pro vytvoření kryptovaného datového proudu je potřeba inicializovat objekt *Cipher* na kryptování heslem (PBE – Password Based Encryption), tento objekt se pak předává do konstruktoru *CipherOutputStream* / *CipherInputStream*. Použití *CipherOutputStream* / *CipherInputStream* je pak obvyklé.

Inicializaci kryptování provedeme následovně:

```
// Begin by creating a salt of 64 bits (8 byte)
byte [] salt = aPassword.substring(0,8).getBytes();

// Create the PBEKeySpec with the given password
PBEKeySpec keySpec = new PBEKeySpec(aPassword.toCharArray());

// Create a SecretKeyFactory for PBEWithSHAAndTwofish
SecretKeyFactory keyFactory =
    SecretKeyFactory.getInstance("PBEWithMD5AndDES");

// Create our key
SecretKey key = keyFactory.generateSecret (keySpec);

// Now create a parameter spec for our salt and iterations
PBEParameterSpec paramSpec = new PBEParameterSpec (salt, ITERATIONS);

//Create a cipher and initialize it for encryption
Cipher cipher = Cipher.getInstance("PBEWithMD5AndDES");

//DECRYPT_MODE for decrypting the data
cipher.init (Cipher.ENCRYPT_MODE, key, paramSpec);
```

Kód 2: inicializace Password Based Encryption

a vytvoření streamů:

```
out=new CipherOutputStream(output, cipher);
in =new CipherInputStream(input, cipher);
```

Kompletní zdrojový kód streamů pro práci s heslem kryptovanými datovými toky je v třídě *saiko.compression.PBEOutputStream* a *saiko.compression.PBEInputStream*.

Testování kompresních algoritmů

Kompresní algoritmy, které jsou v následujících kapitolách implementované, je potřeba testovat z hlediska funkčnosti a zjistit jejich účinnost – kompresní poměr výsledného a originálního souboru.

Testování by mělo být provedeno na různých typech datových vzorů a budeme pro něj používat sadu standardních testovacích souborů, které pokrývají potřebný rozsah typů dat, a které umožňují porovnávat dosažené výsledky komprese s výsledky, jak obecných a testovaných algoritmů, tak ostatních algoritmů vytvořených různými uživateli internetu.

Projekt *The canterbury corpus* [Canterbury] nám takové soubory nabízí, na adrese http://corpus.canterbury.ac.nz je k dispozici podrobný popis projektu, testovací soubory určené ke komprimaci a výsledky testů mnoha standardních komprimačních programů.

Zde jsou jednotlivé kategorie a popisy testovaných souborů, v testech se na ně budou odkazovat přehledy výsledků komprese.

Canterbury Corpus

Sada souborů určených k testování rychlosti a účinnosti kompresních algoritmů. Kolekce byla sestavena roku 1997 z typických příkladů běžně používaných dat, na adrese http://corpus.canterbury.ac.nz/resources/report.pdf je k dispozici popis postupu vybírání jednotlivých vzorových souborů.

název souboru	popis	velikost [bajtů]
alice29.txt	anglický text	152 089
asyoulik.txt	scénář hry	125 179
cp.html	kód HTML	24 603
fields.c	kód jazyka C	11 150
grammar.lsp	kód LISP	3 721
kennedy.xls	dokument MS Excel	1 029 744
lcet10.txt	technický dokument	426 754
plrabn12.txt	knižní dokument	481 861
ptt5	faxová stránka	513 216
sum	binární soubor pro SPARC	38 240
xargs.1	GNU manuálové stránky	4 227

Tabulka 6: testovací soubory - Canterbury Corpus

The Artificial Corpus

Tato kolekce obsahuje soubory s extrémy datového obsahu – velmi malé soubory, náhodná textová data atd.

název souboru	popis	velikost [bajtů]
a.txt	soubor s jedním znakem	1
aaa.txt	soubor se 100 000 stejnými znaky	100 000
alphabet.txt	opakující se anglická abeceda	100 000
random.txt	náhodně generované textové znaky	100 000

Tabulka 7: testovací soubory - Artificial Corpus

The Large Corpus

Kolekce velkých souborů – některé algoritmy jsou vytvořeny pro velký objem dat, některé jsou moc rychlé na měření času na malých souborech.

název souboru	popis	velikost [bajtů]
E.coli	kompletní genom baktérie E. Coli	4 047 392
bible.txt	text bible	4 047 392
world192.txt	text "CIA - světová fakta"	2 473 400

Tabulka 8: testovací soubory - Large Corpus

The Miscellaneous Corpus

Kolekce speciálních souborů přidaných uživateli.

název souboru	popis	velikost [bajtů]
pi.txt	prvních milion pozic čísla Pi	1 000 000

Tabulka 9: testovací soubory - Miscellaneous Corpus

The Calgary Corpus

Historická kolekce testovacích souborů pro kompresi z osmdesátých let dvacátého století, která se stala standardem pro testování bezztrátové komprese dat, byla nahrazena kolekcí Canterbury Corpus.

název souboru	popis	velikost [bajtů]
bib	textová databáze knih	111 261
book1	kniha	768 771
book2	kniha (troff formát)	610 856
geo	geofyzikální údaje	102 400
news	komunikace USENETu	377 109
obj1	binární soubor pro VAX	21 504
obj2	binární soubor pro Apple	246 814
paper1	technický dokument	53 161
paper2	technický dokument	82 199
pic	černobílá faxová grafika	513 216
progc	kód C	39 611
progl	kód LISP	71 646
progp	kód PASCAL	49 379
trans	přepis terminálové komunikace	93 695

Tabulka 10: testovací soubory - Calgary Corpus

Testovací program

Testovací program má za úkol dle zadaného komprimovacího algoritmu provést kompresi, dekompresi a porovnání dekomprimovaného a originálního souboru.

Kompresní algoritmy se programu zadávají jako řetězcové parametry, testovací program ze zdrojového adresáře zkopíruje do cílového adresáře originální soubor a přidá mu příponu _SRC, zkomprimuje soubor do souboru s příponou podle parametricky zadané identifikace kompresního algoritmu a dekomprimuje zkomprimovaný soubor do souboru s příponou _DEST. U originálního a výsledného souboru vypočte a porovná MD5 podpis, velikost souboru a podle velikosti komprimovaného souboru vypíše kompresní poměr. Kopírování, komprese a porovnání se dělají pro celou stromovou strukturu zadanou vstupním adresářem

v parametrech programu, pro dané kompresní algoritmy zadané jako názvy tříd implementujících rozhraní *InputStream* a *OutputStream*.

K volání komprese podle zadaného názvu třídy implementující rozhraní *OutputStream* je použita funkčnost dynamické práce s objekty v Javě zpřístupněna v packagi *java.lang.reflect.*

```
// compress the file
{
     InputStream in=new FileInputStream(aInput);
     File outFile=new File(aOutputDir,aInput.getName()+"."+aAlg.ext);

     Constructor c=aAlg.outputStream.getConstructor(
         new Class[] { OutputStream.class }
     );
     OutputStream out=(OutputStream) c.newInstance(
         new Object[] { new FileOutputStream(outFile)}
     );
     int i;
     while((i=in.read())!=-1) {
          out.write(i);
     }
     out.close();
     in.close();
 }
```

Kód 3: práce s java.lang.reflection a streamy

Příklad použití testovacího programu:

```
/** runs the compression test suite */
public static void main(String aArg[]) throws Exception  {
  TestSuite tests=new TestSuite(
        "c:/benchmark_test_data",
        "c:/benchmark_test_result"
        );

  tests.addAlgorithm(
        "saiko.compression.RLEInputStream",
        "saiko.compression.RLEOutputStream",
        "RLE"
        );
  tests.addAlgorithm(
       "saiko.compression.BMInputStream",
       "saiko.compression.BMOutputStream",
       "BM"
    );

  tests.perform();
}
```

Text 4: příklad použití testovacího programu

Kompletní zdrojový kód se nachází ve třídě *saiko.compression.TestSuite*.

RLE

název komprese:	Run Length Encoding
výhody:	jednoduchý, přímý a rychlý algoritmu, nepotřebuje statistický rozbor dat, vhodný pro data s kontinuálními oblastmi opakujících se prvků, do výstupu ukládá malé množství přídavných výstupních informací týkající se algoritmu
nevýhody:	v obecných případech špatný kompresní poměr, spíše historický kompresní algoritmus
použití:	grafické formáty PCX, BMP, TIFF, PDF
vznik:	80. léta dvacátého století

Jedna z nejjednodušších forem bezztrátové komprese dat, která je pro svou jednoduchost často uváděna jako úvod do kompresních algoritmů. Komprese nahrazuje kontinuální oblast výskytu opakujícího se prvku vzorkem prvku a údajem o počtu jeho výskytů.

RLE algoritmus funguje na všech typech dat bez ohledu na jejich obsah, datový obsah ale určuje, jaký kompresní poměr může být danou metodou dosažen.

Algoritmus je používán na textových souborech s častým opakováním mezer, s textovými tabulkami, na grafických souborech s opakujícími se barvami, nejlépe na monochromatických obrázcích nebo obrázcích s malým počtem barev.

Na datech s náhodným rozložením hodnot algoritmus nedosáhne téměř žádné komprese, komprimovaný výstup může být i objemnější, než originálně komprimovaná data.

Princip RLE

V datech čtených ze vstupu jsou hledány po sobě následující výskyty opakujících se prvků (většinou znaku či bajtu), na výstup je zapsán vzorek jednoho prvku a počet jeho opakování.

Příklad RLE kódování dat:

jednorozměrné pole *input* může být zapsáno ve formě pole *output,* kde je-li prvek pole *output* dvourozměrný, první jeho část značí počet opakování příslušného prvku, druhá pak konkrétní prvek.

```
input  = {'A','A','A','A','C','A','B','B','B'}
output = {{4,'A'}, {'C'}, {'A'}, {3,'B'}}
```

Princip algoritmu je jednoduchý, jednotlivé implementace se liší hlavně tím, jakým způsobem je zapsán a označen počet opakujících se prvků.

Nejjednodušší, ale neefektivní řešení je zapisovat ke každému výskytu znaku počet jeho opakování

```
input  = AAAAAABBBBCCC4
output = 6A4B3C14
```

V případě neopakujících se dat by se nám mohl výstup zvětšit až 2x

```
input  = ABCDE
output = 1A1B1C1D1E
```

Jestliže by se vstupní data se skládala pouze z textových znaků, mohli bychom počet opakování odlišit číselnými hodnotami

```
input  = ABCDEEEEEEEE
output = ABCD8E
```

Takové optimální podmínky ale většinou nemáme, a proto se v praxi používá buď označení počtu výskytů nějakým speciálním znakem, nebo bitovými příznaky znaků.
V následujícím případě je speciálním znakem určujícím počet opakování následujícího znaku znak '*'

```
input  = 1ABCDEEEEEEEE3FFFF4
output = 1ABCD*8E3*4F4
```

Standard Compuserve pro kódování RLE souborů

Compuserve standard vznikl v osmdesátých letech 20tého století pro ukládání monochromatických obrázků. Metodou RLE kóduje stavy – bod zobrazen, bod nezobrazen. Obrázky v (tehdy) standardním "vysokém rozlišení" 256x192 bodů, nebo "středním rozlišení" 128x96 bodů byly reprezentovány jako jednorozměrný tok dat od počátku v obrázku vlevo nahoře po řádcích směrem doprava dolů. Kódování bylo tvořeno dvojicí hodnot - počtem zobrazených a následným počtem nezobrazených bodů.

Standard MS Windows pro kódování RLE souborů

[RLE spec BMP] MS Windows standart pro kódování RLE souborů je definován pro ukládání čtyř a osmi bitových obrázků.

Pro čtyřbitové obrázky obsahuje výstupní kompresní sekvence dva znaky – první značí počet vykreslovaných bodů, druhý bajt obsahuje v prvních 4 bitech barvy na lichých pozicích bodů, v druhých 4 bitech barvy bodů na sudých pozicích. Jestliže první znak je 0, druhý znak je jednou z escape sekvencí, která může značit konec řádky, konec souboru nebo posun v obrázku o relativní pozici v souřadnicích.

Pro 8 bitové obrázky je výstupní sekvence také tvořena dvěma znaky – první obsahuje počet vykreslovaných bodů, druhý znak barvu daného bodu.

Standard grafického formátu PCX

[RLE spec PCX] Formát PCX je jedním z nejstarších grafických formátů a objevil se již v počátcích 80tých let dvacátého století, umožňuje uchovávat obrázky s grafickým rozlišením od 1 do 24 bitů a pravděpodobně není žádný grafický program, který by neuměl s formátem PCX pracovat.

Komprimovaná data jsou uložena tak, že jsou-li první dva bity bajtu rovny jedné, ve zbývajících šesti bitech je uložen počet opakování následujícího znaku, jinak je daný bajt nekomprimovaný a je odeslán na výstup.

Standard Utah RLE Format

[RLE spec Utah] Utah RLE je formát pro ukládání grafického obrazu po jeho vrstvách RLE kompresním algoritmem. Obvykle jsou použity vrstvy pro červenou, zelenou a modrou barvu, může ale být použito až 255 vrstev plus jedna vrstva kanálu alfa pro transparentnost barvy. Formát podporuje libovolnou bitovou hloubku kanálu, současná implementace je ale omezena pouze na 8 bitů. Obraz je uložen po řádkách zleva doprava, jednotlivé kanály jsou uloženy odděleně a hodnoty v nich jsou komprimovány pomocí algoritmu RLE.

Implementace algoritmu

Příkladová implementace algoritmu by mohla být podobná použití RLE ve formátu PCX. První bit komprimovaného výstupu bude vyhrazen jako znak počtu opakování uloženého ve zbývajících sedmi bitech pro následující znak. Tak by se zachytilo opakování až 128 znaků (0..127)+1 (uložený nulový počet opakování nemá smysl) a komprese by se dosáhlo již při třech následných opakování jednoho znaku.

Jestliže při kompresi bude na vstupu znak s již prvním bitem nastaveným, bude muset algoritmus i při jeho jediném kontinuálním výskytu zapsat kompresní sekvenci jednoho opakování vstupního znaku. U každého znaku s nastaveným prvním bitem a pouze jedním počtem opakování dojde tedy k nárůstu výstupních dat o jeden bajt.

Pro náhodně generovaný vstup mohli bychom spočítat dosažený kompresní poměr. Je-li na vstupu znak *A*, je pravděpodobnost, že dalším znakem bude také znak *A* rovna 1/256.

Obecně pravděpodobnost *P* výskytu *n*tice shodných znaků v náhodně generovaném vstupu je na délce *l*

$$P(n,l)=\frac{l}{256^{n-1}}$$

Při výskytu *n*tice shodných znaků kde 2<*n*<=128 dostaneme úsporu *n*-2 znaků. Celková úspora U' na náhodně distribuovaných vstupních datech o délce *l* bude tedy

$$U'(l)=\sum_{n=3}^{128}\frac{l}{256^{n-1}}(n-2\ )$$

znaků.

Od dané úspory musíme ale odečíst znaky, které přibudou nutností kódovat jednotlivé výskyty znaků s nastaveným prvním bitem do dvouznakové podoby.

Pravděpodobnost, že znak bude následován dalším shodným znakem, je 1/256, že bude následován další *n*ticí znaků je *P(n,1)*. Pravděpodobnost *P1,* že znak se bude vyskytovat v náhodných datech samostatně, je

$$P1=1-\sum_{n=2}^{\infty}\frac{1}{256^{n-1}}$$

Pravděpodobnost, že znak bude mít nastavený první bit je 1/8 (platí opět pouze u náhodných dat). Přírůstek (záporná úspora) U'' dat vzniklý dvojitým kódováním samostatně se vyskytujících znaků s nastaveným prvním bitem bude na úseku délky l znaků rovna

$$U''(l)=\frac{1}{8}(1-\sum_{n=2}^{\infty}\frac{1}{256^{n-1}})l$$

celková úspora U na délce l

$$U(l)=\sum_{n=3}^{128}\frac{l}{256^{n-1}}(n-2)-\frac{l}{8}(1-\sum_{n=2}^{\infty}\frac{1}{256^{n-1}})$$

a konečně celkový kompresní poměr na délce l pro náhodně generovaná dostatečně dlouhá data bude

$$\lim_{l=>\infty}\frac{l-\sum_{n=3}^{128}\frac{l}{256^{n-1}}(n-2)+\frac{l}{8}(1-\sum_{n=2}^{\infty}\frac{1}{256^{n-1}})}{l}$$

Numerické výsledky (pro délku l=100.000):

- úspora počtu znaků: 1,54

- počet výskytů jednotlivých nekontinuálních prvků s nastaveným prvním bitem: 12450

- kompresní poměr: 1.12

Pro náhodně generovaná data dostaneme u konkrétního algoritmu nárůstek o přibližně 12%. Tento nárůstek můžeme považovat za maximální ('horší' než absolutně náhodně generovaná data nebudou), kompresní poměr bude závislý na konkrétní charakteristice dat, vypočtenou hodnotu nárůstu při náhodně distribuovaném vstupu můžeme ale porovnávat s hodnotami u jiných kompresních algoritmů a jejich variant.

Jestliže bychom například zvolili jako návěstí pro počet následně opakujících se prvků celý jeden konkrétní bajt (např. 0xFF) a celým dalším znakem zapsali počet opakování dalšího prvku, mohli bychom zachytit až 256 opakování jednoho znaku, úsporu znaků bychom dosáhli od 4 opakování výš. Výskyt samotného nebo dvou samotných znaků 0xFF bychom museli nahradit sekvencí tří znaků.

Pro naši příkladnou implementaci zvolíme deklaraci návěstí samostatným znakem 0xFF.

Specifikace vybrané implementace RLE komprese:

Komprimovaná data uvozuje hlavička dat – řetězec *DSCRLE1*. Blok dat je tvořen originálními daty, pokud se v komprimovaných datech vyskytne znak *0xFF*, následuje jednobajtový počet opakování dalšího následujícího znaku (bajtu).

Kompresní algoritmus

Kompresní algoritmus je založen na metodě *write* rozhraní *OutputStream,* která zapisuje do výstupu jednotlivé bajty vstupu. Metoda kontroluje, zda-li předchozí znak byl stejný jako znak, který se právě zapisuje. Pokud ano, uchovává si počet po sobě jdoucích stejných znaků, pokud ne, vypíše předchozí znak do výstupu voláním metody *flushBuffer().* Protože ukládáme počet opakování do jednoho znaku, je možné zkomprimovat pouze 256 opakování určitého znaku. Metoda *flushBuffer()* vypíše předchozí znak; pokud byl opakován vícekrát, nebo, pokud je to znak 0xFF, který jinak používáme k označení počtu opakování, zapíše tento znak pomocí sekvence počtu opakování, je li znak opakován méně než 4x, nedošlo by komprimací k úspoře dat a data jsou vypsána v původní podobě.

```
public synchronized void write(int aValue) throws IOException {
     if(aValue==currentChar && currentCount<256) {
         currentCount++;
     } else {
         flushBuffer();
         currentChar=aValue;
         currentCount++;
     }
 }


 private void flushBuffer() throws IOException {
     if (currentCount > 0) {
         if(currentCount<4 && currentChar!=0xFF) {
             for(int i=0; i<currentCount; i++)
                 out.write(currentChar);
         } else {
             out.write(0xFF);
             out.write(currentCount-1);
             out.write(currentChar);
         }
    currentCount = 0;
         currentChar=-1;
     }
 }
```

Kód 5: příklad RLE komprimace

Dekompresní algoritmus

Dekompresní algoritmus založený na metodě *read* rozhraní *InputStream*, která vrací další přečtený znak z komprimovaného vstupu. Metoda čte další vstupní bajt; pokud načtený bajt je příznakem počtu opakování, jsou načteny další parametry opakování a nastaveny čítače počtu opakovaného prvku.

Pokud jsou nastaveny čítače opakování, vrací se při dalším čtení přednastavený prvek a hodnota čítače je snížena.

```
public synchronized int read() throws IOException {
     //is the counter set ?
     if(currentCount>0) {
         // if yes, return the current char and decrease the counter
         currentCount--;
         return currentChar;
     } else {
         //read next char
         int c=in.read();
         if(c!=0xFF) {
             //if it is not flag, than return it
             return c;
         } else {
             //read the remaining parameters, set the counter and return the
             //value
             int c2 = in.read();
             int c3 = in.read();
             if(c2==-1 || c3==-1)
                 throw new IOException("Unexpected end of data stream");
             currentCount=c2;
             currentChar=c3;
             //do not decrease the count here - the c3 should be written c2+1
             //times
             return currentChar;
         }
     }
 }
```

Kód 6: čtení komprimovaného RLE vstupu

Kompletní zdrojový kód naleznete v příloze, a je také umístěn na doprovodném CD ve třídě *saiko.compression.RLEInputStream* a *saiko.compression.RLEOutputStream*.

Výsledky kompresního algoritmu

název souboru	komprimovaná velikost	originální velikost	kompresní poměr
alice29.txt	149 862	152 089	98.54 %
asyoulik.txt	125 001	125 179	99.86 %
cp.html	24 577	24 603	99.89 %
fields.c	10 908	11 150	97.83 %
grammar.lsp	3 637	3 721	97.74 %
kennedy.xls	10 29 867	1 029 744	100.01 %
lcet10.txt	413 649	426 754	96.93 %
plrabn12.txt	481 203	481 861	99.86 %
ptt5	113 503	513 216	22.12 %
sum	34 384	38 240	89.92 %
xargs.1	4 234	4 227	100.17 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
a.txt	8	1	800.00 %
aaa.txt	1 180	100 000	1.18 %
alphabet.txt	100 007	100 000	100.01 %
random.txt	100 007	100 000	100.01 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
E.coli	4 550 415	4 047 392	98.10 %
bible.txt	2 420 325	4 047 392	97.85 %
world192.txt	2 420 325	2 473 400	99.90 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
pi.txt	998 983	1 000 000	99.90 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
bib	111 268	111 261	100.01 %
book1	768 652	768 771	99.98 %
book2	610 568	610 856	99.95 %
geo	100 827	102 400	98.46 %
news	366 668	377 109	97.23 %
obj1	18 698	21 504	86.95 %
obj2	264 919	246 814	107.34 %
paper1	52 999	53 161	99.70 %
paper2	82 197	82 199	100.00 %
pic	113 503	513 216	22.12 %
progc	38 899	39 611	98.20 %
progl	66 619	71 646	92.98 %
progp	45 232	49 379	91.60 %
trans	89 746	93 695	95.79 %

Tabulka 11: výsledky algoritmu RLE

Z uvedených výsledků komprimace je patrné, že algoritmu RLE je vhodný pro jednoduché obrázky nebo opakující se texty, zvolený zápis informací týkající se kompresního algoritmu zajistil malou odchylku od původní velikosti pro data, které metoda RLE neuměla zkomprimovat, velikost hlavičky a dalších přidaných informací byla minimání.

Možné modifikace RLE algoritmu

Pro zdokonalení algoritmu RLE by jej bylo možné rozšířit na vyhledávání a kódování opakovaných skupin několika znaků a na analýzu vstupních dat pro výběr vhodné metody označení počtu opakování jednotlivých částí (znaků případně jejich skupin).

Bitové mapy

název komprese:	Kompresní algoritmus používající bitové mapování výskytu znaků
výhody:	přímý a rychlý algoritmus, nepotřebuje statistický rozbor dat, vhodný pro data s častým výskytem konkrétního prvku
nevýhody:	v obecných případech špatný kompresní poměr
použití:	některé starší hardwarové komunikační protokoly
vznik:	60. léta dvacátého století
zdroj:	[Komprimace 2000]

Princip algoritmu

Další jednoduchá forma bezztrátové komprese, nahrazuje výskyty nejčastějšího znaku binárními mapami s vyznačenými pozicemi těchto výskytů a ve výsledném datovém výstupu tento nejčastěji se opakující prvek vynechává. Algoritmus je jednoduše implementovatelný a rychlý, byl používán v některých hardwarových komunikačních zařízeních.

Je-li o datovém vstupu známo, že se v něm vyskytuje nejvíce konkrétní znak (např. u některých specifických komunikací to může být znak 0), může tento znak v komprimovaném výstupu být vynechán a pozice jeho originálního výskytu zaznamenány binárními příznaky do většinou osmibitové mapy. V bitové mapě nastavení jednotlivých bitů označuje, zda-li na dané pozici patří vynechaný znak (0), nebo znak, který nebyl komprimován (1).

Příklad:

vstup:	`C1 C1 C1 FA C1 C1 FB FB`
nejčastější znak	`C1`
bitová mapa:	`0 0 0 1 0 0 1 1 => M`
komprimovaný výstup:	`M FA FB FB`

Maximální teoretická komprese při použití bitové mapy kdy jeden bajt je mapován na jeden bit je 12.5 % - každých osm bajtů může být nahrazeno jedním bajtem s binární mapou.

Věcí implementace je, podobně jako u komprese RLE, zda bitové mapy budou tvořeny pro každých 8 bajtů vstupních dat, nebo, zda bude výskyt mapy uvozen speciální syntaxí.

Implementace algoritmu

Pro implementaci algoritmu bychom mohli zvolit opět návěstí v podobě samostatného symbolu (např. 0xFF), tím bychom ale při maximální teoretické kompresy zredukovali 8 bajtů na bajty 2 a dosáhli tím maximální komprese na 1/4 původního objemu dat; nebo bychom mohli vyhradit první bit znaku na příznak, zda se jedná o mapu, nebo ne; tím bychom mohli mapovat jen následujících 7 bajtů, ale teoretická maximální komprese 1/7 je větší, než v předchozím uvažovaném případě.

Pokud známe pravděpodobnost výskytu nejčastěji opakovaného prvku *P*, můžeme vypočítat teoretický kompresní poměr. Spočítáme-li pravděpodobnost *P2* toho, že v okolí 7 znaků budou právě 2 (3, 4, 5, 6, 7) výskyty daného znaku, můžeme spočítat úsporu v bajtech na konkrétní délce právě pro tuto variantu, můžeme sečíst úsporu pro všechny možné varianty a vypočítat výsledný kompresní poměr.

Kromě úspor bude mít algoritmus za následek jednobajtový přírůstek v případě, že číslo s nastaveným prvním bitem nebude mít ve svém 7 prvkovém okolí žádný výskyt daného nejčastějšího prvku.

Pro náhodně distribuované hodnoty vychází výsledný kompresní poměr cca. 109%, pro data s 25% zastoupením konkrétního znaku by kompresní poměr byl cca. 25%.

Specifikace implementace

Komprimovaná data uvozuje hlavička dat – řetězec *DSCBM1*. Hlavička dat je následována znakem nejčastějšího výskytu, v samotných datech je odlišena bitová mapa nastaveným prvním bitem bajtu dat, následujících sedm bitů v mapě specifikuje výskyt mapovaného nejčastějšího prvku v následujících sedmi bajtech. Na pozicích, kde je bit v bitové mapě nastaven na hodnotu 0 je vynechán mapovaný znak, pozice, na kterých je v bitové mapě je hodnota 1, se doplňují hodnotami uloženými za bitovou mapou.

Kompresní algoritmus

První úkol kompresního algoritmu je zjistit z dat nejčastěji se opakující prvek.

Metoda *write* rozhraní *OutputStream* načítá data do bufferu, buffer se po zaplnění nebo při uzavření zkomprimuje a vypíše.

```
public synchronized void write(int aValue) throws IOException {
    buffer.write(aValue);
    if(buffer.size()>=bufferSize) {
        flushBuffer();
    }
}
```

Kód 7: obecná konstrukce metody InputStream.write pro práci s buffery dat

Samotná metoda výstupu a komprimace dat musí ošetřit některé mezní situace, jako například vytváření map pro všechny znaky s nastaveným prvním bitem a zarovnávání dat na části, které lze mapou zpracovat (k vytvoření mapy je zapotřebí 7 dalších bajtů).

```
private synchronized void flushBuffer() throws IOException {
        ...

        int pos=0;
        //for all the data in buffer
        while(pos<=data.length-7) {
            int value=((int)(char)data[pos])&0xFF;
            pos++;

            //if the current character is not the most often used, just output it
            if((value!=theChar) && ((value&0x80)==0)) {
                out.write(value);
                continue;
            }

            //check if there is still at least 6 more bytes available at the
            // data - we need it for the map
            if(pos>data.length-6) break;

            //create the map
            int i, map;

            // we have read the most often used character, or the character with
            // first bit set
            if(value==theChar) {
                // 10111111 => first bit means this is the map, the second
                // the most often used character is next
                map=0xBF;
            } else {
                // 11111111 => first bit means this is the map, the second
                // the next character will be in the output
                map=0xFF;
            }

            // read next 6 values (7th one has been already read)
            int vals[]=new int[6];
            for(i=0; i<vals.length; i++) {
                vals[i]=((int)(char)data[pos])&0xFF;
                pos++;
            }
            // go through values and clear the bit in map on it's position
            // if it is the most often used character
            for(i=0; i<vals.length; i++) {
             if(vals[i]==theChar) {
                 map=map&~(1<<(5-i)); //clear the bit in map
             }
            } //for

            //output the map and the data
            out.write(map);

            //if we are creating the map just because of the character
```

```
            //which has first bit set, we have to output this byte
            //map already indicates it
            if(value!=theChar) {
                out.write(value);
            }

            //output the characters which differs from the most used one
            for(i=0; i<vals.length; i++) {
             if(vals[i]!=theChar) {
                 out.write(vals[i]);
             }
            } //for

        } //while

        // clear the buffer
        buffer.reset();
         ...
    }
```

Kód 8: implementace metody write pro komprimaci bitových map

Dekompresní algoritmus

Dekompresní algoritmus pro čtení následujícího znaku je postaven na uchovávání informací o případně zpracovávané bitové mapě.

```
    public synchronized int read() throws IOException {
        // if we do not have the map set, read next byte
        if(currentMapPos==0) {
            int c=in.read();
            //if this is end or if the byte does not have first bit set,
            //return in
            if((c==-1) || ((c & 0x80)==0) ) return c;
            //we have read the bit map
            currentMap=c;
            //set the current position on the second bit
            currentMapPos=(1<<6);
        }
        // we deal with the map
        int res;
        if((currentMap & currentMapPos)!=0) {
            //map indicates to read the next char from input
            res=in.read();
        } else {
            //map indicates to return the 'background' character
            res=theChar;
        }
        //shift the map position
        currentMapPos=currentMapPos>>1;
        return res;
    } //read
```

Kód 9: implementace metody read pro dekomprimaci bitových map

Kompletní zdrojový kód se nachází ve třídách *saiko.compression.BMInputStream* a *saiko.compression.BMOutputStream*.

Výsledky kompresního algoritmu

název souboru	komprimovaná velikost	originální velikost	kompresní poměr
alice29.txt	137 997	152 089	90.73 %
asyoulik.txt	116 988	125 179	93.46 %
cp.html	24 282	24 603	98.70 %
fields.c	9 909	11 150	88.87 %
grammar.lsp	3 299	3 721	88.66 %
kennedy.xls	706 107	1 029 744	68.57 %
lcet10.txt	396 831	426 754	92.99 %
plrabn12.txt	448 338	481 861	93.04 %
ptt5	138 591	513 216	27.00 %
sum	29 415	38 240	76.92 %
xargs.1	4 031	4 227	95.36 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
a.txt	9	1	900.00 %
aaa.txt	14 298	100 000	14.30 %
alphabet.txt	100 008	100 000	100.01 %
random.txt	99 863	100 000	99.86 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
E.coli	3 927 647	4 047 392	84.67 %
bible.txt	3 701 998	4 047 392	91.47 %
world192.txt	2 263 843	2 473 400	91.53 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
pi.txt	962 237	1 000 000	96.22 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
bib	107 381	111 261	96.51 %
book1	716 414	768 771	93.19 %
book2	576 822	610 856	94.43 %
geo	86 918	102 400	84.88 %
news	352 666	377 109	93.52 %
obj1	18 546	21 504	86.24 %
obj2	240 840	246 814	97.58 %
paper1	50 305	53 161	94.63 %
paper2	77 454	82 199	94.23 %
pic	138 591	513 216	27.00 %
progc	36 045	39 611	91.00 %
progl	65 283	71 646	91.12 %
progp	41 557	49 379	84.16 %
trans	88 986	93 695	94.97 %

Tabulka 12: výsledky algoritmu bitových map

Na výsledcích je vidět, že algoritmus opět dobře pracuje s černobílými faxovými obrazy (*pic* a *ppt5*), na rozdíl od RLE ale pracuje lépe i s dalšími binárně uloženými soubory a částečně komprimuje například i náhodně generovaná data.

Možné modifikace algoritmu

Algoritmus bitových map by šlo rozšířit o bitové mapy nejen pro jeden nejvyskytovanější prvek (např. pro 2 prvky v případech, kdy dva prvky současně mají statisticky v datech větší zastoupení než ostatní prvky), šlo by také vytvářet bitové mapy pro konkrétní nejvyskytovanější skupiny znaků – tím bychom ale již dostávali spíše k formě slovníkové kompresne.

Expanze bitové báze

název komprese:	expanze bitové báze v datech s omezeným číselným rozsahem
výhody:	principiálně zajímavý algoritmus, vhodný pro datové soubory s omezeným množstvím znaků
nevýhody:	pomalejší kódování i dekódování, nutnost pracovat s velkými čísly, neúčinnost algoritmu na binárních datech se zastoupením všech možných 256ti znaků
použití:	experimentální datová komprese, vhodná pro specifické data nebo části datových souborů u nichž víme o malé množině znaků
vznik:	vlastní myšlenka - konkrétní podoba algoritmu pravděpodobně není popsána

Princip algoritmu

Algoritmus je zobecněním metody, která je nazývána "metoda půlbajtové komprimace". Jestliže víme, že vstupní data budou tvořena např. jen čísly, a známe bitový rozsah těchto čísel, můžeme se rozhodnout dvě čísla ukládat do jednoho bajtu tak, že první číslo zabírá první čtyři bity bajtu, druhé číslo zbývající čtyři bity.

Algoritmus lze ale zobecnit úvahou, že, jestliže máme ve vstupním datovém proudu omezenou množinu znaků, je zbytečné kódovat každý znak do 8 bitů. Kdyby byla vstupní data tvořena jen číslicemi *"1"* a *"0"*, mohl by každý vstupní znak být mapovaný přímo na jeden výstupní bit. Problém nastane s množinou znaků, kterou nelze mapovat na přesně určitý počet bitů. Jestliže vstupní sekvence bude složena ze tří znaků, je na zakódování hodnoty v rozmezí 0,1,2 jeden bit málo a dva bity zbytečně moc, dva bity umí zapsat čtyři různá čísla, docházelo by k plýtvání bitovou kapacitou.

Vstupní množinu znaků můžeme považovat za definiční obor hodnoty znaku a znaky samotné představují čísla na základu o velikosti množiny prvků. Jestliže jsou vstupní data složena např. jen z číslic, můžeme číslice brát přímo jako znaky desítkové soustavy. Čísla 0..10 bychom mohli zapisovat do 5 bitů, a tím dosáhnout kompresního poměru 5/8, neboli 62%.

Dále můžeme číst větší skupinu znaků z dané známé množiny jako jedno velké číslo příslušné numerické soustavy, toto číslo převést a uložit v binárním zápisu. Čím větší počet vstupních znaků budeme číst, tím méně bude docházet k výkonnostním úbytkům vlivem bitového "zaokrouhlování". V ukázkových výpočtech je použito čtení čísel po 1KB znaků.

Při převádění mezi různě velkými numerickými základy s velmi rozsáhlými čísly algoritmus využije objekt *java.math.BigInteger* pro aritmetické operace s neomezeně velkým celým číslem.

Implementace algoritmu

Kompresní algoritmus

Algoritmus nejprve přečte požadované množství dat, z dat sestaví znakovou abecedu, kterou setřídí vzestupně podle hodnoty jednotlivých znaků, a vstupní data reindexuje na hodnoty indexů jednotlivých znaků v abecedě.

Jestliže na vstupu bude hodnota

"toto je zajimavy text"

bude vytvořena abeceda

"to jezaimvyx"

abeceda bude setříděna tak, aby při dalších datech s případnou shodnou množinou znaků ale s jiným pořadím výskytu, vznikla ta samá abeceda

" aeijmotvxyz"

následně budou vstupní data reindexována na hodnoty pozic výskytu jednotlivých znaků v abecedě

t	o	t	o	_	j	e	_	z	a	j	i	m	a	v	y	_	t	e	x	t
7	6	7	6	0	4	2	0	11	1	4	3	5	1	8	10	0	7	2	9	7

do výstupních dat je zapsána jedním bajtem velikost znakové abecedy (v příkladě 12) a následně celá abeceda. Při opakované komprimaci datových částí jednoho datového toku, v případě kdy je nově vytvořená abeceda shodná s abecedou předchozí části, je jako hodnota délky abecedy zapsána nula a abeceda se nezapisuje.

Do výstupu následuje zapsat 2 bajtová hodnota velikosti originálního datového bufferu, komprimační algoritmus omezuje velikost datového bufferu na hodnotu 65535.

Dále je celý buffer načtený jako číslo o bázi velikosti počtu znaků množiny (11 v příladu), pro naši ukázku od konce jako

$$7+9\ .12^1+2\ .12^2+7\ .12^3+0\ .12^4\ldots$$

Takto získané číslo je třídou *BigInteger* převedeno na pole bajtů, na výstup je zapsána dvoubajtová hodnota velikosti dat tohoto čísla a následně vlastní číslo. Protože číslo kóduje hodnoty podle velikosti datové báze, je možné, že bude kódovat u náhodných binárních dat datovou bázi o velikosti 256 znaků. Potom nedojde k žádné úspoře místa, ale kódované číslo nebude jakkoliv větší než původní data. Výstupní datový proud bude objemnější o informační data kompresního algoritmu přidávané do výstupu.

Po zpracování načteného bloku dat se pokračuje načtením a zpracováním dalšího následujícího bloku znaků.

```
//get the stats of order 0 and sort it (will be sorted 'alphabetically')
Statistics[] stats=new SourceStatistics(data).getStats(0,true);

//reindex data according to the stats
for(int i=0; i<data.length; i++) {
    for(int n=0; n<stats.length; n++) {
        if(stats[n].data[0]==data[i]) {
            data[i]=(byte)n;
            break;
        }
    }
}

//compute the number the data represents
BigInteger base=new BigInteger(String.valueOf(stats.length));

//start with zero
BigInteger value=new BigInteger("0");
for(int i=0; i<data.length; i++) {
    //multiply the previous result with the base size
    value=value.multiply(base);
    //ad new lowest value to the big number
    value=value.add(
         new BigInteger(String.valueOf(((int)(char)data[i])&0xFF)));
}

//write out the length of the number data;
data=value.toByteArray();
length=data.length;
for(int i=0; i<2; i++) {
    out.write((length>>(8*(1-i)))&0xFF);
}

//write out the number
out.write(data);
```

Kód 10: komprese expanzí bitové báze

Dekompresní algoritmus

Při dekomprimování dat čteme nejprve abecedu, která bude k dekódování použita, velikost originálního bufferu a zakódované číslo. Podle velikosti abecedy *N* pak přepočítáme přečtené číslo na číslo o základě *N* a pokračujeme čtením dalšího bloku dat.

```
//read length of the original buffer
length=0;
for(int i=0; i<2; i++) {
    length=length*256;
    int v=in.read();
    length+=v;
}
buffer = new byte[length];

//length of the  compressed data chunk
length=0;
for(int i=0; i<2; i++) {
    length=length*256;
    length+=in.read();
}

byte data[]=new byte[length];
in.read(data);
BigInteger value=new BigInteger(data);

//expand number
BigInteger base=new BigInteger(String.valueOf(alphabet.length));

//convert it to BASE radix
for(int i=0; i<buffer.length; i++) {
    BigInteger current = value.mod(base);
    value=value.divide(base);
    buffer[buffer.length-1-i]=current.byteValue();
}

//reindex the buffer
for(int i=0; i<buffer.length; i++) {
    int v=((int)(char)buffer[i])&0xFF;
    buffer[i]=alphabet[v];
}
```

Kód 11: dekomprese expandované bitové báze

Výsledky kompresního algoritmu

název souboru	komprimovaná velikost	originální velikost	kompresní poměr
alice29.txt	113 529	152 089	74.65 %
asyoulik.txt	95 537	125 179	76.32 %
cp.html	19 956	24 603	81.11 %
fields.c	9 450	11 150	80.96 %
grammar.lsp	2 822	3 721	75.84 %
kennedy.xls	922 875	1 029 744	89.62 %
lcet10.txt	320 241	426 754	75.04 %
plrabn12.txt	358 011	481 861	74.30 %
ptt5	254 348	513 216	49.56 %
sum	33 470	38 240	87.53 %
xargs.1	3 329	4 227	78.76 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
a.txt	13	1	1 300.00 %
aaa.txt	595	100 000	0.60 %
alphabet.txt	59 303	100 000	59.30 %
random.txt	75 597	100 000	75.60 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
E.coli	1 184 534	4 047 392	25.54 %
bible.txt	2 869 019	4 047 392	70.89 %
world192.txt	1 987 348	2 473 400	80.35 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
pi.txt	420 760	1 000 000	42.08 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
bib	91 330	111 261	82.09 %
book1	566 114	768 771	73.64 %
book2	459 184	610 856	75.17 %
geo	113 784	102 400	111.12 %
news	311 858	377 109	82.70 %
obj1	20 741	21 504	96.45 %
obj2	239 850	246 814	97.18 %
paper1	41 799	53 161	78.63 %
paper2	60 922	82 199	74.12 %
pic	254 348	513 216	49.56 %
progc	34 371	39 611	83.23 %
progl	53 031	71 646	74.02 %
progp	38 539	49 379	78.05 %
trans	76 917	93 695	82.09 %

Tabulka 13: výsledky algoritmu expanze bitové báze

I když počítání nebylo moc rychlé, výsledky jsou lepší než u předešlých algoritmů. Nejlépe dopadl soubor s opakovaným jedním znakem, ten byl načten a přeindexován na pole bajtů s hodnotu 0, s touto hodnotou se pak v numerické konverzi mezi bázemi čísel dobře pracovalo a celkový kompresní poměr vyšel méně než 1%. Zdrojový kód kompresního algoritmu je v třídách *saiko.compression.ExpandedBitBaseInputStream* a *saiko.compression.ExpandedBitBaseOutputStream*.

Možné modifikace algoritmu

Algoritmus je pěkným příkladem práce s daty jako jedním velkým číslem, pro praktické použití by bylo potřeba využít rychlejší knihovny pro práci s velkými čísly, pravděpodobně by se dal optimalizovat i algoritmus převodu čísel mezi jednotlivými soustavami. Dále by bylo vhodné algoritmus kombinovat s nějakým dalším druhem komprese, například s aritmetickým kódováním

Diatomické kódování (kódování bajtových párů)

název komprese:	Kompresní algoritmus používající bitové mapování výskytu znaků
výhody:	celice dobrý kompresní poměr pro textové soubory, nepotřebuje statistický rozbor dat, vhodný pro data s častým výskytem konkrétního prvku, malé množství výstupních informací týkající se algoritmu
nevýhody:	špatný kompresní poměr pro binární data
použití:	některé starší hardwarové komunikační protokoly
vznik:	60. léta dvacátého století
zdroj:	[Komprimace 2000]

Princip algoritmus

Opět principálně jednoduchá komprese s velice efektivním výsledkem hlavně pro textová data. Algoritmus nejprve vyhledá znaky, které ve vstupních datech nejsou obsaženy. Poté najde ve vstupních datech nejčastější dvojici bajtů a tuto dvojici nahradí jedním z nepoužitých znaků. Analýza nejčastější dvojice se opakuje až do vyčerpání množiny volných znaků.

Hezký efekt vzniká při komprimaci jednoduchých dat. nechť vstupní řetězec je například sekvence

A A A A A A A A A A

Prvním vybraným nepoužitým znakem může být znak 1, znakem 1 nahradíme výskyt nejčastější dvojice *„AA“*

1 1 1 1 1

Dalším nepoužitým znakem je znak 2, znakem 2 nahradíme výskyt nejčastější dvojice „11“

2 2 1

Poslední substitucí bude substituce skupiny „22“ dalším nepoužitým symbolem - 3.

3 1

Diatomická komprese vstupních dat „A A A A A A A A A A“ bude výstup „3 1“ společně se záznamem provedených substitucí.

Implementace algoritmu

Kompresní algoritmus

```
 while((available.size())>0 && data.length>1) {
    data2.reset();
    byte val=((Byte)available.get(0)).byteValue();
    byte c1, c2;

    available.remove(0);
    Statistics[] stats=new SourceStatistics(data).getStats(2,true);
    if(stats[0].count<2) break; //end, no more duplicate pair found
    c1=stats[0].data[0];
    c2=stats[0].data[1];
    substitutions.add(new DiatomicSubstitution(c1, c2,  val));

    Byte current[] = new Byte[2];
    for(int i=0; i<data.length; i++) {
        if(current[0]==null) {
            current[0]=new Byte(data[i]);
        } else { // current[1]==null
            current[1]=new Byte(data[i]);
        }

        if(current[0]!=null && current[1]!=null) {
            if(current[0].byteValue()==c1 && current[1].byteValue()==c2) {
                data2.write(((int)(char)val)&0xFF);
                current[0]=null;
                current[1]=null;
            } else {
                data2.write(((int)(char)current[0].byteValue())&0xFF);
                current[0]=current[1];
                current[1]=null;
            }
        } //write output
    } //for all data
    if(current[1]!=null)
        data2.write(((int)(char)current[1].byteValue())&0xFF);
    if(current[0]!=null)
        data2.write(((int)(char)current[0].byteValue())&0xFF);
    data=data2.toByteArray();
} //while available


//output compressed data
out.write(data);
```

Kód 12: diatomická komprese - substituce bajtových párů

Dekompresní algoritmus

Dekompresní algoritmus aplikuje na zkomprimovaných datech v opačném pořadí substituce provedené při kompresi.

```
Vector subst = new Vector();

//read the substitutions
for(int i=0; i<length; i++) {
    int pair1=in.read();
    int pair2=in.read();
    int substitute=in.read();
    subst.add(new DiatomicOutputStream.DiatomicSubstitution(
                    (byte)pair1, (byte)pair2,  (byte)substitute));
}

//apply substitutions from the back
ByteArrayOutputStream data2=new ByteArrayOutputStream(data.length);
for(int i=subst.size()-1; i>=0; i--) {
    DiatomicOutputStream.DiatomicSubstitution current=
              (DiatomicOutputStream.DiatomicSubstitution) subst.get(i);
    for(int n=0; n<data.length; n++) {
        if(current.substitute!=data[n]) {
            data2.write(((int)(char)data[n])&0xFF);
        } else {
            data2.write(((int)(char)current.pair1)&0xFF);
            data2.write(((int)(char)current.pair2)&0xFF);
        }
    }
    data=data2.toByteArray();
    data2.reset();
}
```

Kód 13: princip diatomické dekomprese

Výsledky kompresního algoritmu

název souboru	komprimovaná velikost	originální velikost	kompresní poměr
alice29.txt	75 265	152 089	49.49 %
asyoulik.txt	65 374	125 179	52.22 %
cp.html	11 372	24 603	46.22 %
fields.c	4 995	11 150	42.79 %
grammar.lsp	1 696	3 721	45.58 %
kennedy.xls	435 077	1 029 744	42.25 %
lcet10.txt	215 339	426 754	50.46 %
plrabn12.txt	243 067	481 861	50.44 %
ptt5	63 670	513 216	12.41 %
sum	33 343	38 240	87.19 %
xargs.1	2 230	4 227	52.76 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
a.txt	11	1	1100.00 %
aaa.txt	64	100 000	0.06 %
alphabet.txt	125	100 000	0.13 %
random.txt	94 094	100 000	94.09 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
E.coli	1 355 539	4 047 392	29.22 %
bible.txt	1 775 716	4 047 392	43.87 %
world192.txt	1 318 402	2 473 400	53.30 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
pi.txt	505 903	1 000 000	50.59 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
bib	54 734	111 261	49.19 %
book1	404 367	768 771	52.60 %
book2	335 592	610 856	54.94 %
geo	102 410	102 400	100.01 %
news	236 509	377 109	62.72 %
obj1	21 514	21 504	100.05 %
obj2	234 323	246 814	94.94 %
paper1	29 605	53 161	55.69 %
paper2	42 502	82 199	51.71 %
pic	63 670	513 216	12.41 %
progc	22 046	39 611	53.39 %
progl	33 856	71 646	47.25 %
progp	22 373	49 379	45.31 %
trans	49 756	93 695	53.10 %

Tabulka 14: výsledky algoritmu diatomického kódování

Přes jednoduchý princip kompresního algoritmu je na výsledcích vidět, že diatomická komprese je pro většinu textových souborů velice efektivní. Algoritmus je implementovaný ve třídách *saiko.compression.DiatomicInputStream* a *saiko.compression.DiatomicOutputStream.*

Huffmanovo statistické kódování

název komprese:	statistická komprese za použití Huffmanových kódů
výhody:	rychlé, lineární kódování, na běžných datech dobré výsledky
nevýhody:	překonáno aritmetickým kódováním, v Huffmanových kódech nelze celým počtem bitů rozlišit malé rozdíly v pravděpodobnostech výskytů jednotlivých prvků
použití:	historicky slavné a dodnes používané kódování, použito v softwarových i hardwarových kompresních algoritmech
vznik:	1952 – D. A. Huffman

Princip algoritmu

[Huffman-princip] Myšlenka statistického kódování je jednoduchá. Známe-li statistiky jednotlivých (skupin) znaků vstupního datového proudu, můžeme nejčastěji se vyskytující znaky kódovat nejkratšími sekvencemi kódu, pro nejméně často se vyskytující znaky lze použít kódování delší posloupností. Tím dostaneme výstupní datový stream s variabilními kódovými slovy. Otázkou je, jak vytvořit a odlišit kódy reprezentující jednotlivé znaky.

Stejnou teorii, jen s jiným schématem tvorby kódovacích slov, vypracovali v té samé době ještě další teoretici.

Pro kódová slova se používá variabilní binární prefixovové kódy, unikátní kódy složené z jedniček a nul, které nejsou prefixem dalšího jiného kódu. Tím s požadavkem variability jednotlivých kódů odpadá nutnost jednotlivá kódová slova ve výstupním datovém proudu oddělovat.

Máme-li vstup tvořený např. pěti znaky, a známe-li statistiky jednotlivých znaků, můžeme sestavit různé varianty prefixových kódů.

Pro následující tabulku výskytů znaků můžeme například sestavit rychle prefixové kódy uvedené v tabulce.

znak	pravděpodobnost	kód
A	0.35	1
B	0.17	01
C	0.17	001
D	0.16	0001
E	0.15	00001

Dané ukázkové kódy splňují podmínku unikátnosti v rámci prefixů a žádný z kódů nemůže být zaměněný s prefixem jiného kódu. Takové kódování ale není jakkoliv efektivní a nezachycuje vztah mezi poměry pravděpodobnosti výskytu jednotlivých znaků. Jestliže by např. všechny znaky měly pravděpodobnost výskytu 20%, bylo by výhodnější použít jiné kódování.

Optimálním řešení úlohy výběru vhodného kódu musí z těchto poměrů vycházet.

Kromě Huffmanových kódů vytvořili algoritmy na generování kódů i kódy Shannon, Fano, Gilbert a Moore. [Fano 49] [Moore 59] [Abramson 63]

Nevýhodou prefixových kódů je "zaokrouhlování" poměrů mezi jednotlivými pravděpodobnostmi výskytu na celé počty bitů. Aritmetické kódování, které bude popsáno dále a pracuje s velkým reálným číslem, toto omezení překonává.

Pro generování bitových kódů Huffman použil tvorbu binárního ohodnoceného stromu, kde vnitřní uzly nemají žádné hodnoty, zato každý list je reprezentací nějakého původního symbolu.

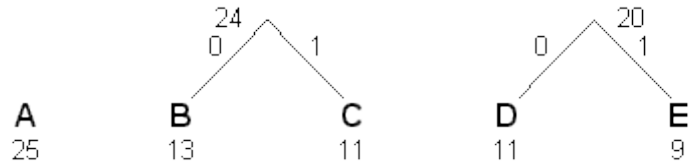
Algoritmus přetvoří jednotlivé prvky do seznamu nespojených uzlů, které setřídí podle četnosti. Dva uzly s nejmenším ohodnocením N1 a N2 spojí do jednoho N, který ohodnotí součtem hodnot těchto dvou uzlů N1 a N2, ze seznamu nespojených uzlů N1 a N2 odebere a naopak do seznamu vloží uzel N. Pokračuje se novým setříděním seznamu uzlů a odebrání dalších dvou uzlů s nejmenší hodnotou do té doby, než v seznamu zůstane jen jeden uzel – kořenový uzel celého binárního stromu. Každý vnitřní uzel binárního stromu, včetně kořenového uzlu, má dvě větve. Jednu si označíme hodnotou 0 a druhou hodnotou 1. Příslušný Hofmanův kód je pak cestou z kořene stromu až k listu obsahujícím reprezentaci jednotlivého znaku.

Můžeme vytvořit binární strom pro následující známé statistiky

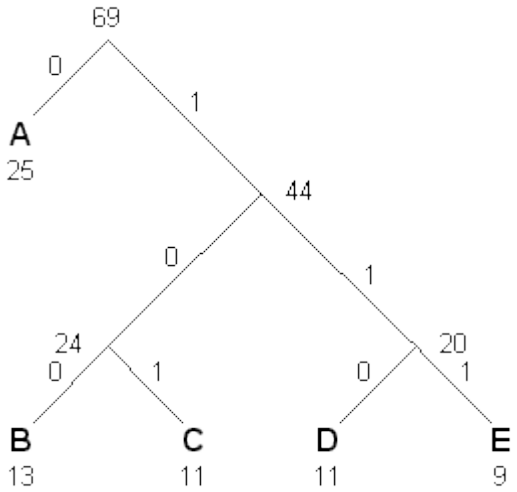
znak	počet výskytů
A	25
B	13
C	11
D	11
E	9

Nejprve spojíme do stromu symboly D a E, které mají nejmenší ohodnocení a vytvoříme jejich rodiče s ohodnoceného součtem hodnot potomků. Rodiče budeme považovat za nový symbol, dále s D a E již pracovat nebudeme a budeme se zabývat jen zbývajícími nespojenými uzly.

Spojíme další dva nejméně ohodnocené uzly – B a C.



A postup opakujeme spojením BC s DE a nakonec spojením A s BCDE.



Z daného binárního stromu vyplývá následující kódovací tabulka

znak	počet výskytů	kód
A	25	0
B	13	100
C	11	101
D	11	110
E	9	111

Pro příklad si můžeme uvést další kódovací tabulky vytvořené Hoffmanovým algoritmem, pro jiné rozložení poměrů četností jednotlivých prvků.

znak	ohodnocení	kód
A	1	01
B	1	000
C	1	001
D	1	10
E	1	11

znak	ohodnocení	kód
A	100	1
B	100	00
C	1	011
D	1	0100
E	1	0101

znak	ohodnocení	kód
A	1000	00
B	1000	01
C	1000	10
D	1	110
E	1	111

Pro kódování dat s výrazným statistickým rozložením, jako je např. text, je vhodné použít pro náhradu kódy ne jednotlivé prvky, ale jejich seskupení, statistiky druhého a vyššího řádu samozřejmě ale mohou obsahovat exponenciální množství prvků.

Implementace algoritmu

Kompresní algoritmus

Algoritmus vychází z vytvoření Huffmanova binárního stromu, čtení binárních kódů z tohoto stromu pro jednotlivé znaky a zpětně pro binární kódy získávání znaků, které kód reprezentuje.

Kódový strom má tu vlastnost, že pracuje s četnostmi reprezentovaných znaků, nezajímá se ale o konkrétní podobu jednotlivých znaků, ani o to, zdali kód reprezentuje jednotlivý znak nebo více znaků, ani jestli jsou ve stromu kombinované reprezentace jednotlivých znaků a skupin znaků.

```
/** Computes three from elements added by addElement()
     *  creates the tree, than function getCode() can be called to get code for
        particular
     *  characters and function getRepresentation() to get characters from the code
     */
    public void compute()
    {
        if(initElements==null) return;

        //transforms nodes in initElements to the tree
        //repeat while there is no more elements than the root
        //element itself
        while(initElements.size()>1) {
            //first, sort the elements in DESC order
            Collections.sort(initElements, new Comparator() {
                public int compare(Object o1, Object o2) {
                    HuffmanBinaryTree tree1=(HuffmanBinaryTree)o1;
                    HuffmanBinaryTree tree2=(HuffmanBinaryTree)o2;
                    if(tree1.count<tree2.count) return 1;
                    if(tree1.count>tree2.count) return -1;
                    return 0;
                }
            });

            //take last two and gather them
            int size=initElements.size();
            HuffmanBinaryTree tree1=(HuffmanBinaryTree)initElements.get(size-1);
            HuffmanBinaryTree tree0=(HuffmanBinaryTree)initElements.get(size-2);

            // remove the two last elements
            initElements.remove(size-1);
            initElements.remove(size-2);

            //ad the parent of those elements to the list
            initElements.add(new HuffmanBinaryTree(tree0, tree1));
        }
    } //compute
```

Kód 14: vytvoření Huffmanova binárního stromu

Jednotlivé elementy přidané do stromu jsou uchovány kromě stromu samotného v seznamu, který nám umožňuje získat tyto elementy a cestou směrem ke kořeni stromu získat odpovídající Huffmanův kód.

Opačný postup – procházení stromu od kořene směrem k listům – použijeme, jestliže známe binární kód a hledáme znaky, které tento kód reprezentuje.

Pro kódování a dekódování komprimovaných informací budeme potřebovat pracovat s tím samým binárním stromem, nemusíme ale při komprimaci ukládat přímo samotný binární strom, stačí, jestliže uložíme do výstupních dat statistiky, z kterých strom byl vytvořen, a při dekódování tyto statistiky použijeme k vytvoření shodného stromu, který budeme pro dekódování používat.

Jestliže známe statistiky, zbývá už jen číst vstupní bajty, konvertovat je na Huffmanovy kódy a kódy vypisovat do výstupního proudu dat.

```
//get statistic, do not sort them
Statistics stats[]=new SourceStatistics(data).getStats(1,false);

//create and fill the huffman binary tree
HuffmanBinaryTree tree=new HuffmanBinaryTree();
if(stats!=null) { //stats are null for 0 length file
    for(int i=0; i<stats.length; i++) {
        tree.addElement(stats[i].data, stats[i].count);
    }
    tree.compute();
}

//transform the data
String currentData="";
int i=0;
while(i<data.length) {
    String code=tree.getCode(new byte[]{data[i]});
    currentData+=code;
    if(currentData.length()>7) {
        String currentByte = currentData.substring(0,8);
        currentData=currentData.substring(8);
        int value=Integer.parseInt(currentByte,2);
        out.write(value);
    }
    i++;
}
//write out what rest in the last coded byte
while(currentData.length()>0) {
        length=Math.min(currentData.length(),8);
        String currentByte = currentData.substring(0,length);
        while(currentByte.length()<8) currentByte+="0";
        currentData=currentData.substring(length);
        int value=Integer.parseInt(currentByte,2);
        out.write(value);
}
```

Kód 15: komprimace Huffmanovým kódováním

Dekompresní algoritmus

Dekompresní algoritmus po načtení statistik a vytvoření Huffmanova binárního stromu čte vstupní znaky, které převádí na řetězec prefixového kódu a kód transformuje zpět na originální znak.

```
//create the huffman tree
HuffmanBinaryTree tree=new HuffmanBinaryTree();
if(stats.length>0) {
    for(int i=0; i<stats.length; i++) {
        tree.addElement(stats[i].data, stats[i].count);
    }
    tree.compute();
}

length=0;
int offset=0;
StringBuffer currentCode=new StringBuffer(1024);
while(length<data.length) {
    String currentByte=Integer.toBinaryString(in.read());
    while(currentByte.length()<8) currentByte="0"+currentByte;
    currentCode.append(currentByte);
    byte b[]=null;
    while(length<data.length && currentCode.length()>offset
       && (b=tree.getRepresentation(
                currentCode.toString().substring(offset)
       ))!=null) {
        data[length++]=b[0];
        String code=tree.getCode(b);
        offset+=code.length();
    }
}
```

Kód 16: dekomprimace Huffmanových kód

Výsledky kompresního algoritmu

název souboru	komprimovaná velikost	originální velikost	kompresní poměr
alice29.txt	88 204	152 089	57.99 %
asyoulik.txt	76 197	125 179	60.87 %
cp.html	16 466	24 603	66.93 %
fields.c	7 714	11 150	66.09 %
grammar.lsp	2 407	3 721	64.69 %
kennedy.xls	454 240	1 029 744	44.11 %
lcet10.txt	251 111	426 754	58.84 %
plrabn12.txt	277 055	481 861	57.50 %
ptt5	108 440	513 216	21.13 %
sum	26 419	38 240	69.09 %
xargs.1	2 833	4 227	67.02 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
a.txt	13	1	1 300.00 %
aaa.txt	12 518	100 000	12.52 %
alphabet.txt	59 784	100 000	59.78 %
random.txt	75 396	100 000	75.40 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
E.coli	1 160 744	4 047 392	25.02 %
bible.txt	2 222 120	4 047 392	54.90 %
world192.txt	1 560 693	2 473 400	63.10 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
pi.txt	424 956	1 000 000	42.50 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
bib	73 234	111 261	65.82 %
book1	440 682	768 771	57.32 %
book2	368 846	610 856	60.38 %
geo	74 065	102 400	72.33 %
news	247 500	377 109	65.63 %
obj1	16 828	21 504	78.26 %
obj2	193 714	246 814	78.49 %
paper1	33 631	53 161	63.26 %
paper2	47 968	82 199	58.36 %
pic	108 440	513 216	21.13 %
progc	27 480	39 611	66.55 %
progl	43 301	71 646	60.44 %
progp	30 490	49 379	61.75 %
trans	65 114	93 695	69.50 %

Tabulka 15: výsledky algoritmu Huffmanova kódování

Možné modifikace algoritmu

Huffmanovo kódování je rychlý a přehledný algoritmus statického kódování. Pro datové toky, u kterých nechceme bufferovat (a tím zpožďovat) data a přidávat do dat statistické informace, lze použít adaptivní obměnu Huffmanova kódování, která dynamicky vytváří a upravuje statistiky a k nim příslušné Huffmanovy kódy. Problém takového algoritmu je ale ten, že k dosažení efektivnosti algoritmu je zapotřebí po každém vstupu předělat binární kódový strom, což je velice zdlouhavé.

Jiné modifikace metody používají pravděpodobné 'známé' statistiky pro daný typ souboru, například pro anglický text, bez toho, aniž by data musely analyzovat.

Dále jsme používali jen statistiky prvního řádu, které nejsou zcela efektivní, mohli bychom zkusit využít statistiky řádů vyšších.

Jestliže ale budeme chtít implementovat Huffmanovo kódování pro statistiky vyšších řádu, dostáváme se k následujícímu problému.

Máme-li např. řetězec

CABABDABC

a pro něj statistiky druhého řádu

CA	1
AB	3
BA	1
BD	1
DA	1
BC	1

a budeme-li číst znaky po dvojicích, přečteme posloupnosti

CA BA BD AB C

Nejen, že nebudeme vědět co s posledním znakem, pro který statistiky nemáme, ale ani jsme jakkoliv nevyužili toho, že dvojice AB se vyskytuje v datech 3x, my jsme jí přečetli pouze jednou. Ve statistikách vyšších řádů by se dále zvětšovali pravděpodobnost toho, že spojitá datová část, pro níž známe statistiku, bude posunuta oproti počátku čtení dat.

Při praktických zkouškách Huffmanova kódování se statistikami vyšších řádů se u některých textových souborů dostavilo mírné zlepšení komprimace dat, u náhodných binárních souborů ale docházelo k nárůstu datových souborů i k zvýšení časové pracnosti výpočtu. Například soubor *random.txt,* obsahující náhodné textové znaky, je statistikami prvního řádu možné zkomprimovat na 75 % velikosti, při použití statistik druhého řádu ale vzrostla velikost souboru a kompresní poměr činil 106%. Inteligentní komprimovací algoritmus by se mohl umět rozhodnout, jaké statistiky pro kódování použít.

Aritmetické kódování

název komprese:	statistické aritmetické kódování
výhody:	dokonalejší formou Huffmanova kódování, velmi efektivní kódování posloupností stejných znaků
nevýhody:	strojově náročnější, práce s desetinným číslem v rozmezí 0 .. 1, která by vyžadovala použití operací s desetinnou čárkou se v praxi přepisuje do algoritmů používajích celá čísla
použití:	jako substituce namísto Huffmanova kódování, doplňková komprese k jiným kompresním mechanizmům
vznik:	počátek 90tých let 20tého století

Princip algoritmu

Nevýhoda Huffmanova kódování spočívá v tom, že pravděpodobnostní počty zapisované v celých číslech nekorespondují zcela s binárním zápisem těchto čísel. Jestliže statickým kódováním přiřadíme dvěma různým vstupním znakům pravděpodobnost 50%, každý vstupní znak můžeme zakódovat do bitu 0 nebo 1. Jestliže ale bude znaků více, například 3 různé znaky, musíme nutně kódovat do 2 bitů, které ale dávají kombinaci 4 různých čísel, a tyto dva bity nebudou tedy využity tak efektivně, jak by využity být měly.

Aritmetické kódování se (tak jako výše popsaný algoritmus expanze bitové báze) nestaví k datům jako posloupnosti jednotlivých bitů, ale k jako jednomu velkému číslu.

Jestliže známe statistiky jednotlivých znaků vstupních dat, můžeme vypočítat přesnou pravděpodobnost výskytu těchto znaků. Například pro řetězec „BILL GATES“ můžeme sestavit následující pravděpodobnostní tabulku:

znak	*počet*	*pravděpodobnost*	*rozmezí*
MEZERA	1	1/10	0,0 – 0,1
A	1	1/10	0,1 – 0,2
B	1	1/10	0,2 – 0,3
E	1	1/10	0,3 – 0,4
G	1	1/10	0,4 – 0,5
I	1	1/10	0,5 – 0,6
L	2	1/20	0,6 – 0,8
S	1	1/10	0,8 – 0,9
T	1	1/10	0,9 – 1,0

Jednotlivým znakům pak v jejich libovolném pořadí rozdělíme interval 0..1 podle jejich pravděpodobnosti. Jestliže první znak L (podle tabulky setříděné dle počtu výskytů a následně abecedně) má pravděpodobnost výskytu 0,4, připadne mu rozmezí 0,0–0,4. Další znak – s pravděpodobností 0,2 má následující část rozmezí intervalu 0,4-0,6.

Proces aritmetického kódování je takový, že rozdělujeme interval <0..1> podle pravděpodobností výskytu jednotlivých vstupních znaků a hledáme číslo, které v závislosti na pravděpodobnostní tabulce charakterizuje ve statistických závislostech nejlépe vstupní řetězec. Vstupní podmínka kódování je, že číslo leží mezi 0 a 1.

Pro uržení mezí intervalu použijeme dvě proměnné – low a high.

Pro konkrétní *„HELLO“* příklad, první vstupní znak *H* má pravděpodobnost výskytu 0,2 a vycházeje z této pravděpodobnosti, obdržel interval v rozmezí 0,6 – 0,8. Jestliže známe tento interval, nebo číslo, které v něm leží, můžeme zpětně určit, který znak tento interval reprezentuje. Při každé nové hodnotě vstupu rozdělíme stávající interval podle pravděpodobnostní tabulky do intervalu nového (užšího), určující stanovené rozmezí stávajícího intervalu.

[Arithmetic] Postup kódování řetězce „BILL GATES“ podle vytvořeného pravděpodobnostního rozdělení bude následující:

znak	low	high
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
SPACE	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

Při několikanásobné iteraci rozdělení intervalu vznikl interval 0.2572167752 – 0.2572167756.

Číslo, které reprezentuje řetězec „BILL GATES“ leží v tomto intervalu a můžeme si z intervalu vybrat libovolné číslo, například 0.2572167752.

Toto číslo unikátně definuje při znalosti pravděpodobnostní statistické tabulky vstupní řetězec „BILL GATES“.

Postup dekódování čísla bude opačný – budeme hledat znaky, v jejichž statistickém intervalu leží vstupní číslo. Nejjednodušší je nalezení prvního znaku – číslo leží v intervalu 0,2-0,3 – první znak tedy bude B.

Dále operací opačnou k výpočtu rozdělení intervalu odečteme z čísla interval 0,2-0,3 a získáme nové číslo – 0.572167752.

Postup dekódování:

číslo	výstupní znak	low	high	Range
0.2572167752	B	0.2	0.3	0.1
0.572167752	I	0.5	0.6	0.1
0.72167752	L	0.6	0.8	0.2
0.6083876	L	0.6	0.8	0.2
0.041938	SPACE	0.0	0.1	0.1
0.41938	G	0.4	0.5	0.1
0.1938	A	0.2	0.3	0.1
0.938	T	0.9	1.0	0.1
0.38	E	0.3	0.4	0.1
0.8	S	0.8	0.9	0.1
0.0				

Implementace algoritmu

Algoritmus lze podle uvedeného postupu pro testovací potřeby implementovat velice jednodušše použitím desetinných čísel. Takové použití ale bude mít své omezení v nepřesnosti a omezenosti desetinné aritmetiky. V praktickým použití je nutné použití desetinných čísel nahradit postupem, který pracuje s celými čísly. Jedním z mnou testovaných postupů bylo použití velkého desetinného čísla implementovaného Javou – java.math.BigDecimal, další testovanou myšlenkou bylo vyjádření intervalů, jakožto ryze racionálnách čísel, zlomky s čitatelem i jmenovatelem ve tvaru velkého celého čísla java.math.BigInteger. Takový postup by byl z teoretického hlediska nejpřesnější, z praktického hlediska byly výpočty velice pomalé s tím, že nárůst časové náročnosti výpočtu nebyl adekvátní úspoře místa získané přesností výpočtu. Reálné celočíselné algoritmy pracují

na omezeném rozsahu výstupního intervalu čísla a musí řešit mezní situace, kdy interval je příliš malý na popsání omezeně velkým číslem.

Jednoduchý kódovací algoritmus s použitím desetinných čísel je uveden v třídě ArithmeticTest.java a pracuje přehledně s krátkými vstupními řetězci.

Kompresní algoritmus

Ukázkový kompresní algoritmus využívá statistik prvního řádu ke zjišťování rozmezí intervalu vstupního znaku

```
//get statistics for given input - how many times each character occures
Statistics[] stats=new SourceStatistics(message).getStats(1,true);

//get the total size of the data = total number of all characters
int totalSize=message.length;

//initialize low to be 0 and high to be 1
double low=0, high=1;

//compress the data into real number
for(int i=0; i<message.length; i++) {
    byte nextChar=message[i];   //get next char from message

    //get the position of selected char from stats
    //the range where the char lies in stats
    int char_high_range=0, char_low_range=0;
    int n=0;
    do {
     char_low_range=char_high_range;
     char_high_range+=stats[n].getCount();
    } while(stats[n++].getData()[0]!=nextChar);
    //code new high and low numbers
    double range=high-low;
    high= low + range*char_high_range/(double)totalSize;
    low = low + range*char_low_range/(double)totalSize;
    System.out.println(low);
    System.out.println(high);
}

double result=(low+high)/2;

//print out result
System.out.println("low    : "+low);
System.out.println("high   : "+high);
```

Kód 17: příklad aritmetického kódování

Dekompresní algoritmus

Dekompresní algoritmus opačným postupem dekóduje výsledné desetinné číslo

```
//decode the number into message back again
for(i=0; i<message.length; i++) {
    int high_range=0;
    int low_range=0;
    for(int n=0; n<stats.length; n++) {
        low_range=high_range;
        high_range+=stats[n].getCount();
        //according to the known range search the character
        if(result*totalSize>=low_range && result*totalSize<=high_range) {
            byte data=stats[n].getData()[0];
            message[i]=data;
            double range=stats[n].getCount()/(double)totalSize;
            result=result-low_range/(double)totalSize;
            result=result/(double)range;
            break;
        }
    }
}

//print out the output of de-compression
System.out.println("output : "+new String(message));
```

Kód 18: příklad aritmetického dekódování

Ukázka výstupu testovacího algoritmu

Testovací algoritmus na různých datech vykazuje zajímavé charakteristiky aritmetického kódování. Pro „náhodná“ data je výsledné číslo dlouhé na desetinná místa, pro statisticky jednoduchý vstup je výsledné číslo (výsledný interval) jednoduché. Vzhledem k postupu kódování je zajímavý výstup u řetězce „123456789“, kterým je číslo 0,12345678905.

```
==================================
input  : HELLO
low    : 0.6851200000000001
result : 0.6855
high   : 0.6864
output : HELLO

==================================
input  : International
low    : 0.5471647089859542
result : 0.547164708986
high   : 0.5471647089873806
output : International

==================================
input  : AAAAAAAAA
low    : 0.0
result : 0.5
high   : 1.0
output : AAAAAAAAA
```

```
==================================
input  : AAAA AAAA
low    : 0.5549289573066436
result : 0.57
high   : 0.5982338843209708
output : AAAA AAAA

==================================
input  : AAAA AAAA B
low    : 0.21616497275391225
result : 0.2162
high   : 0.2164001852282435
output : AAAA AAAA B

==================================
input  : 1234567890
low    : 0.12345678900000001
result : 0.12345678905
high   : 0.1234567891
output : 1234567890
```

Praktické použití aritmetické komprese v Javě

Napsat funkční a efektivní kompresní knihovnu pro aritmetické kódování je i přes jednoduchý princip algoritmu náročný úkol.

K praktickému použití aritmetické komprese můžeme v Javě využít některou z volně dostupných knihoven, například [JavaArith]. Komprimace je vytvořena také ve formě vstupně-výstupních streamů, můžeme ji proto snadno vyzkoušet v našem testovacím schématu (na oficiálních stránkách knihoven je již testování na standardní soubory provedeno, testem si ale kompresi ověříme a rozšíříme na ostatní testovací soubory).

Výsledky kompresního algoritmu

název souboru	komprimovaná velikost	originální velikost	kompresní poměr
alice29.txt	52 376	152 089	34.44 %
asyoulik.txt	44 376	125 179	35.45 %
cp.html	8 593	24 603	34.93 %
fields.c	3 546	11 150	30.38 %
grammar.lsp	1 297	3 721	34.86 %
kennedy.xls	231 224	1 029 744	22.45 %
lcet10.txt	148 610	426 754	34.82 %
plrabn12.txt	170 969	481 861	35.48 %
ptt5	52 631	513 216	10.26 %
sum	14 742	38 240	38.55 %
xargs.1	1 772	4 227	41.92 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
a.txt	3	1	300.00 %
aaa.txt	9	100 000	0.01 %
alphabet.txt	94	100 000	0.09 %
random.txt	82 874	100 000	82.87 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
E.coli	1 138 527	4 047 392	24.54 %
bible.txt	1 285 867	4 047 392	31.77 %
world192.txt	886 675	2 473 400	35.85 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
pi.txt	416 957	1 000 000	41.7 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
bib	37 575	111 261	33.77 %
book1	283 304	768 771	36.85 %
book2	223 080	610 856	36.52 %
geo	58 766	102 400	57.39 %
news	155 721	377 109	41.29 %
obj1	10 782	21 504	50.14 %
obj2	97 429	246 814	39.47 %
paper1	19 754	53 161	37.16 %
paper2	30 023	82 199	36.52 %
pic	52 631	513 216	10.26 %
progc	14 839	39 611	35.94 %
progl	21 895	71 646	30.56 %
progp	14 460	49 379	29.28 %
trans	28 374	93 695	30.28 %

Tabulka 16: výsledky algoritmu aritmeticého kódování

V tabulce výsledků je vidět velice dobrá účinnost kompresního algoritmu, vynikající pro soubory s opakovanými daty. V celkové porovnávací tabulce kompresních metod uvidíme, jak si aritmetické kódování povede ve srovnání s dalšími algoritmy jako gzip, bzip2 atd. I když jsou knihovny na aritmetické kódování napsány třetí stranou a zjevně s velkou pečlivostí, není kompresní algoritmus v Javě rychlý, stejně tak jako předešlé, mnou napsané, algoritmy.

Možné modifikace algoritmu

Na aritmetickém kódování je stále co optimalizovat a zdokonalovat z hlediska poměru rychlost algoritmu/účinnost. Dále je možné implementovat statické či adaptabilní ukládání statistických informací – podobně jako při Hufmanovo kódování.

Slovníková komprese LZW/LZ77/LZ78

název komprese:	Lempel-Ziv-Welch komprese
výhody:	velice efektivní, obchází některé nedostatky statistických kompresí
nevýhody:	starší algoritmus
použití:	univerzální použití ve známých komprimačních programech jako ZIP, ARJ, GZIP …
vznik:	1977

Princip algoritmu

Nevýhoda všech statistických algoritmů je nutnost vytvoření statistického modelu, který je omezen svou třídou (jeden, dva znaky …) a zarovnáním dat v datovém toku (bylo popisováno také u Huffmanova kódování). Jestliže vstupní sekvenci tvoří znaky AAABBAAZ, nachází se zde tři dvojice písmen A, jedna dvojice písmen B. Jestliže ale vytvoříme statistiky druhého řádu na rozdělená data po dvou bajtech, dostaneme skupiny znaků AA AB BA AZ – tedy znaky bez jediné existující opakující se dvojice. Při pokusech vytvořit dynamický statistický model, který by obsahoval optimální výčet různě dlouhých skupin, vznikaly problémy, které spíše kompresní kód prodlužovaly, než zkracovaly. Tato úloha je přímo vzorově určena pro slovníkové kompresní metody, kdy vstupní data jsou rozsekána na co nejvíce shodné datové části a místo vícenásobného ukládání stejných datových částí vznikají mezi sebou odkazy na již použitá data. Snaha o zavedení statistických modelů proměnné délky ostatně směřuje právě k principům slovníkové komprese.

[LZW1] Algoritmus LZ77 byl publikován v roce 1977 a popisuje slovníkovou kompresi s vkládáním ukazatelů na již použitá data. Tento algoritmus je použitý například v kompresi GZip [LZW2]. Další variantou algoritmu je LZ78, vydaný v roce 1978, který popisuje algoritmus s odděleně uchovávaným slovníkem. Takový algoritmus používá například grafický formát GIF.

Jako bližší příklad fungování algoritmu si můžeme uvést specifikaci algoritmu použitého v GZip kompresi, která je součástí platformy Javy. Dle [LZW2] jde o algoritmus odvozený z LZ77, který hledá shodné skupiny a nahrazuje druhý výskyt shodné skupiny ukazatelem ve formě odkazu a velikosti odkazovaných dat. Vzdálenosti jsou limitovány 32K a velikosti dat 256 bajty. Data, která nejsou nalezena, společně s popisy ukazatelů, jsou komprimována Huffmanovým kódováním a umístěna před blok komprimovaných dat. Použití Huffmanova kódování by mohlo být nahrazeno kódováním aritmetickým.

Implementace algoritmu

Implementace algoritmu je zabudována do standardních knihoven Javy ve třídách *java.util.GZIPInputStream* a *java.util.GZIPOutputStream*.

Výsledky kompresního algoritmu

název souboru	komprimovaná velikost	originální velikost	kompresní poměr
alice29.txt	54 416	152 089	35.78 %
asyoulik.txt	48 909	125 179	39.07 %
cp.html	7 973	24 603	32.41 %
fields.c	3 207	11 150	27.48 %
grammar.lsp	1 234	3 721	33.16 %
kennedy.xls	204 004	1 029 744	19.81 %
lcet10.txt	144 916	426 754	33.96 %
plrabn12.txt	195 273	481 861	40.52 %
ptt5	56 477	513 216	11.00 %
sum	13 002	38 240	34.00 %
xargs.1	1 748	4 227	41.35 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
a.txt	21	1	2 100.00 %
aaa.txt	133	100 000	0.13 %
alphabet.txt	302	100 000	0.30 %
random.txt	75 747	100 000	75.75 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
E.coli	1 342 002	4 047 392	28.93 %
bible.txt	1 191 897	4 047 392	29.45 %
world192.txt	724 982	2 473 400	29.31 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
pi.txt	470 462	1 000 000	47.05 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
bib	35 240	111 261	31.67 %
book1	313 594	768 771	40.79 %
book2	206 676	610 856	33.83 %
geo	68 445	102 400	66.84 %
news	144 812	377 109	38.40 %
obj1	10 329	21 504	48.03 %
obj2	81 517	246 814	33.03 %
paper1	18 570	53 161	34.93 %
paper2	29 772	82 199	36.22 %
pic	56 477	513 216	11.00 %
progc	13 545	39 611	32.80 %
progl	16 267	71 646	22.70 %
progp	11 240	49 379	22.76 %
trans	19 057	93 695	20.34 %

Tabulka 17: výsledky algoritmu GZip

LZW komprese je velice efektivní, a to se také odráží v popularitě programů ZIP, ARJ atd. Svým konceptem jsou slovníkové metody dle mého názoru již na jedné straně dovedeny k hranicím svých možností, na druhé straně již nepoužívají nové numerické metody či další moderní technologie.

Transformace dat před jejich kompresí

Mnohé komprese využívají jako druhotnou kompresi již existující standardní kompresní algoritmus, například Huffmanovo nebo aritmetické kódování tak, jak tomu bylo například u LZW algoritmu. Takové kompresní algoritmy si dávají za úkol zkomprimovat data a statistické informace o kompresi, či celá komprimovaná data, ještě předat dalšímu kompresnímu algoritmu.

Prakticky podobný přístup, i když principielně z trochu jiné strany, je připravit data tak, aby existující kompresní algoritmus dosahoval na datech lepších kompresních poměrů.

Delta encoding

[TransformDelta] Jednou z nejjednodušších forem transformace dat před jejich kompresí je převedení dat na posloupnost inkrementů (delt). Tak by například vstupní data

0 1 2 3 4 5 6 7 8 9

byla převedena na posloupnost

0 1 1 1 1 1 1 1 1 1

Je vidět, že taková posloupnost půjde jednodušeji zkomprimovat. Delta transofmace je vlastně derivace diskrétních hodnot dat a lze experimentovat s transformací vyšších řádů – například delta transormace druhého řádu by pro náš příklad byla

0 1 0 0 0 0 0 0 0 0

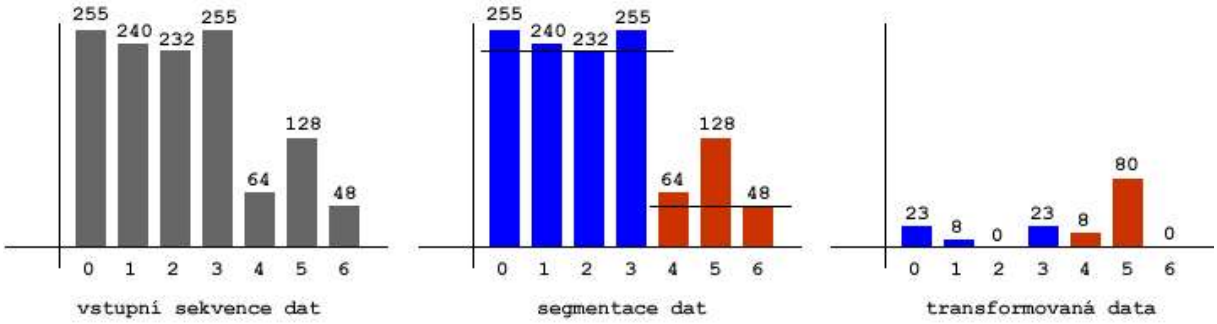
Delta transformaci lze použít například i na komprimaci setříděných slovníkových dat. Mezi setříděnými slovníkovými řetězci lze hledat přírůstky/rozdíly řetězců a zapisovat řetězce jako skupinu *číslo_řetězec,* kde číslo udává počet znaků, které se mají vzít z předchozího výrazu.

slovo	***delta transformace***
bratr	0bratr
bratranec	5anec
bratranecky	9ky
bratranek	8k
bratreni	5eni

Segmentace hladin hodnot

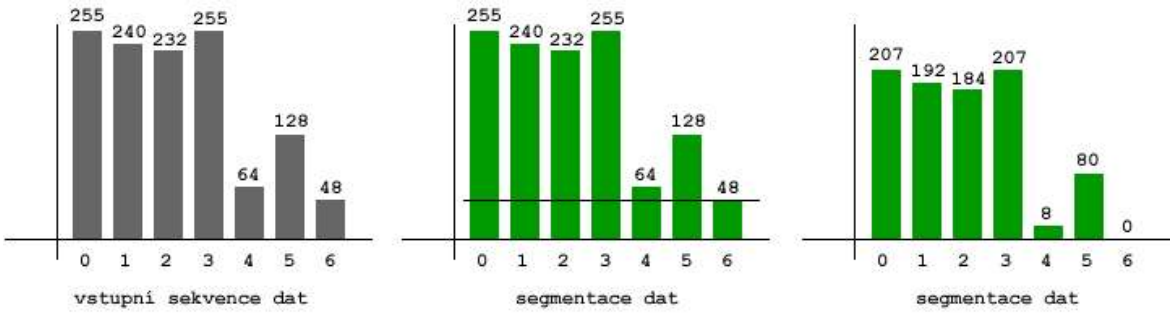
[TransformSegmentation] O této transformaci jsem uvažoval již předtím, než jsem ji našel popsanou, a i proto jsem z její existence měl radost. Myšlenka transformace vychází z předpokladu, že, jsou-li v datech kontinuální segmenty dat s přibližně stejnou hodnotou, lze z tohoto segmentu odečíst minimální hodnotu vzorku v segmentu, a snížit mu tak jeho bytový rozsah. Touto transformací lze docílit snížení počtu využitých bajtů ve vstupních datech a ke zmenšení rozdílů mezi jednotlivými částmi dat, což samo o sobě povede k možnosti lepší komprese.

Například vstupní sekvence dat 255, 240, 232, 255, 64, 128, 48 může být rozdělena na dva segmenty – jeden o společném základu 232 a druhý o společném základu 48. Tyto segmenty mohou být poníženy o své základy – a vzniknou tak data s menším číselným rozsahem, která se dají lépe komprimovat.



Obrázek 3: příklad segmentace dat

I když je také segmentace hladin hodnot v principu jednoduchá, nalézt optimální segment, který se výsledně poníží o svůj základ není jednoduchá úloha a vyžaduje značné úsilí v matematické analýze dat. Ten samý vzorek dat by totiž mohl být považován za jediný segment a celý ponížen o hodnotu 48.



Obrázek 4: příklad segmentace dat

V tomto případě můžeme odhadnout, že druhý způsob není ideální, ale rozhodnout, jak segmentovat například data vzestupného vzorku 0 1 2 3 4 5 6 7 8 9 není jednoduché.

Burrows-Wheelerova transormace

[TransformBWT] Burrows-Wheelerova transformace dat je proti delta transformaci nebo segmentaci hladin hodnot myšlenkou složitější, je ale velice účinná a dosáhla konkrétního rozsáhlého použití ve formátu bzip2.

Transformace je založena na reversibilním algoritmu, který transformuje vstupní řetězec na zjednodušený výstupní data, přitom ale zachovává možnost s minimální přídanou informací rekonstruovat zpětně výstupní data na vstupní řetězec. Zjednodušená výtupní data jsou lépe komprimovatelná a lze na ně použít i jednodušší komprimační algoritmy jako RLE. Nejlepších výsledků dosahuje Burrow-Wheelerova transformace v kombinaci s aritmetickým kódováním.

Transformace vezme vstupní řetězec a vytvoří list se všemi rotacemi tohoto řetězce. Tento list pak setřídí podle abecedy a jako transformovaný řetězec vrátí poslední znaky setříděného listu. Algoritmus je reversibilní a k zpětnému generování vstupního řetězce stačí znát pozici posledního znaku ze vstupního řetězce ve výsledné transformaci. Při zpětné rekonstrukci dat z transformovaného řetězce známe poslední sloupec transformované tabulky, který obsahuje všechny znaky vstupního řetězce. Protože tabulka rotací byla setříděna podle abecedy a my známe všechny znaky řetězce, můžeme vygenerovat první sloupec tabulky ze známých znaků tím, že je setřídíme podle abecedy. Jakmile známe první a poslední znak tabulky, tak vzhledem k vlastnostem rotace víme, že tyto znaky tvoří všechny možné dvojice ve vstupním řetězci. Setřídíme-li tyto dvojice podle abecedy, dostáváme první a také druhý sloupec převední tabulky. Po opakovaném spojování a třídění sloupců dostaneme plnou

tabulku, u které nalezneme řádek, který končí předem známým (zapamatovaným) znakem – a tento řádek je výsledek reverzní transformace na vstupní data.

Algoritmus postupu transformace:

```
BWT (STRING)
  do tabulky vytvořit list ze všech rotací řetězce STRING
  abecedně setřídit jednotlivé řádky listu
  výsledkem transformace je poslední sloupec setříděného listu


inverseBWT (STRING)
  opakovat pro počet znaků řetězce STRING
    řetězec STRING se vloží jako první levý sloupec do tabulky (která je
    z počátku prázdná)
    řádky tabulky se abecedně setřídí
  konec opakování
  výsledkem transformace je vstupní řetězec na řádce, která končí námi
    zapamatovaným znakem
```

Příklad BWT transformace pro vstupní řetězec KATANA:

Tabulka rotací vstupního řetězce:

rotace řetězce KATANA
KATANA
AKATAN
NAKATA
ANAKAT
TANAKA
ATANAK

Setříděná tabulka rotací:

setříděné rotace
AKATAN
ANAKAT
ATANAK
KATANA
NAKATA
TANAKA

Řetězec výsledné transformace je složen z posledního sloupce setříděných rotací

NTKAAA

U praktického použití transformace bude hodně záležet na velikosti transoformovaného datového bloku – u velkých bloků bude transformace velice efektivní, bude ale trvat velice dlouho.

bzip2 – použití Burrows-Wheelerovy transformace

[JavaBZip2] Bzip2 by měl být jedním z nejlepších obecných komprimačních algoritmů. Pro Javu je knihovna bzip2 opět dostupná jako vstupně/výstupní streamy *org.apache.tools.bzip2.CBZip2InputStream* a *org.apache.tools.bzip2.CBZip2OutputStream*.

Výsledky kompresního algoritmu bzip2

název souboru	komprimovaná velikost	originální velikost	kompresní poměr
alice29.txt	43 200	152 089	28.40 %
asyoulik.txt	39 567	125 179	31.61 %
cp.html	7 622	24 603	30.98 %
fields.c	3 095	11 150	26.52 %
grammar.lsp	1 281	3 721	34.43 %
kennedy.xls	130 278	1 029 744	12.65 %
lcet10.txt	107 704	426 754	25.24 %
plrabn12.txt	145 575	481 861	30.21 %
ptt5	49 757	513 216	9.70 %
sum	12 907	38 240	33.75 %
xargs.1	1 760	4 227	41.64 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
a.txt	35	1	3500.00 %
aaa.txt	45	100 000	0.05 %
alphabet.txt	538	100 000	0.54 %
random.txt	75 682	100 000	75.68 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
E.coli	1 251 002	4 047 392	26.97 %
bible.txt	845 621	4 047 392	20.89 %
world192.txt	489 581	2 473 400	19.79 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
pi.txt	431 669	1 000 000	43.17 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
bib	27 465	111 261	24.69 %
book1	232 596	768 771	30.26 %
book2	157 441	610 856	25.77 %
geo	56 919	102 400	55.58 %
news	118 598	377 109	31.45 %
obj1	10 785	21 504	50.15 %
obj2	76 439	246 814	30.97 %
paper1	16 556	53 161	31.14 %
paper2	25 039	82 199	30.46 %
pic	49 757	513 216	9.70 %
progc	12 636	39 611	30.60 %
progl	15 577	71 646	21.74 %
progp	10 708	49 379	21.69 %
trans	17 897	93 695	19.10 %

Tabulka 18: výsledky algoritmu bzip2

I přes náročnost komprimačního výpočtu, jak na CPU, tak na paměť počítače při výborných kompresních poměrech, byl bzip2 jedním z rychlejších algoritmů, i když jeho implementace v Javě je pořád pomalejší, než by měla být pro běžné použití.

Další kompresní algoritmy

Samozřejmě existují i další kompresní algoritmy či mutace existujících algoritmů, které se snaží překonat kompresní poměry stávajících mechanizmů. Samostatnou kapitolou by byly speciální komprese určené pro konkrétní druh dat – například komprese čísel, textu, komprese obrazu, zvuku atd. Většina těchto kompresí ale přechází do kompresí ztrátových, které využívají tolerovatelné míry zkreslení dat. Tyto komprimace pro mne osobně (i když je denně používám) nejsou ve středu zájmu – dochází při nich k nevratnému poškození vstupních informací. Zajímavé jsou spíše různé soutěže v komprimaci speciálního obsahu, jako například *The $5000 Compression Challenge* [Challenge1], nebo na text specializovaná *Offline Data Compression* [Challenge2].

Nejlepší kompresní algoritmy ?

Když jsem sháněl informace o potencionálně nejlepších kompresních algoritmech, zaujaly mne dva produkty. WinZip 9.0 [WinZip] – který umožňuje „enhanced compression“ a v praxi se mi opravdu osvědčil, a historicky kvalitní program WinAce [WinAce]. Protože WinZip i WinAce jsou komerční produkty, není popsaný konkrétní kompresní algoritmus, který používají, kromě toho, že jde o mutace LZW kódování společně v kombinaci se statistickým kódováním. http://compression.ca/act-index.html

Výsledky kompresního programu – WinZip 9.0

s nastavením komprese *enhanced compression*

název souboru	komprimovaná velikost	originální velikost	kompresní poměr
alice29.txt	52 949	152 089	34.81 %
asyoulik.txt	48 182	125 179	38.49 %
cp.html	8 067	24 603	32.78 %
fields.c	3 297	11 150	28.24 %
grammar.lsp	1 336	3 721	35.90 %
kennedy.xls	214 786	1 029 744	20.85 %
lcet10.txt	138 655	426 754	32.49 %
plrabn12.txt	188 050	481 861	39.02 %
ptt5	52 473	513 216	10.22 %
sum	12 818	38 240	33.51 %
xargs.1	1 842	4 227	43.57 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
a.txt	109	1	10900.00 %
aaa.txt	123	100 000	0.12 %
alphabet.txt	159	100 000	0.15 %
random.txt	75 826	100 000	75.82 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
E.coli	1 289 488	4 047 392	27.79 %
bible.txt	1 122 317	4 047 392	27.72 %
world192.txt	668 371	2 473 400	27.02 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
pi.txt	470 893	1 000 000	47.08 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
bib	33 633	111 261	30.22 %
book1	301 211	768 771	39.18 %
book2	197 855	610 856	32.38 %
geo	68 315	102 400	66.71 %
news	138 040	377 109	36.60 %
obj1	10 393	21 504	48.33 %
obj2	80 182	246 814	32.48 %
paper1	18 520	53 161	34.83 %
paper2	29 370	82 199	35.73 %
pic	52 471	513 216	10.22 %
progc	13 534	39 611	32.77 %
progl	16 209	71 646	22.62 %
progp	11 159	49 379	22.59 %
trans	18 509	93 695	19.75 %

Tabulka 19: výsledky algoritmu WinZip

WinZip neoslnil. Je možné, že je znát i určitý nárůst velikosti datových souborů z principu možnosti komprimace adresářové struktury.

Výsledky kompresního programu – WinAce 2.2

s nastavením komprese *maximum compression, buffer 2048 KB*

název souboru	komprimovaná velikost	originální velikost	kompresní poměr
alice29.txt	51 269	152 089	33.70 %
asyoulik.txt	47 034	125 179	37.57 %
cp.html	8 189	24 603	33.28 %
fields.c	3 454	11 150	29.59 %
grammar.lsp	1 529	3 721	41.09 %
kennedy.xls	36 133	1 029 744	3.50 %
lcet10.txt	127 520	426 754	29.88 %
plrabn12.txt	175 930	481 861	36.51 %
ptt5	49 230	513 216	9.59 %
sum	11 001	38 240	28.76 %
xargs.1	2 029	4 227	48.00 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
a.txt	284	1	28400.00 %
aaa.txt	413	100 000	0.41 %
alphabet.txt	486	100 000	0.48 %
random.txt	78 364	100 000	78.36 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
E.coli	1 269 208	4 047 392	27.36 %
bible.txt	964 935	4 047 392	23.84 %
world192.txt	525 998	2 473 400	21.26 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
pi.txt	468 292	1 000 000	46.82 %
název souboru	komprimovaná velikost	originální velikost	kompresní poměr
bib	33 101	111 261	29.75 %
book1	276 815	768 771	36.00 %
book2	181 883	610 856	29.77 %
geo	54 025	102 400	52.75 %
news	126 078	377 109	33.43 %
obj1	9 994	21 504	46.47 %
obj2	72 538	246 814	29.38 %
paper1	18 448	53 161	34.70 %
paper2	28 912	82 199	35.17 %
pic	49 229	513 216	9.59 %
progc	13 619	39 611	32.98 %
progl	16 107	71 646	22.48 %
progp	11 083	49 379	22.44 %
trans	18 387	93 695	19.62 %

Tabulka 20: výsledky algoritmu WinAce

WinAce má zajímavé výsledky. Většinou lepší než WinZip, v některých případech překvapivě velký kompresní poměr. Celková srovnávací tabulka, která je v příloze, porovná WinAce s dalšími kompresemi, hlavně s kompresí bzip2.

Porovnání výsledků jednotlivých kompresních algoritmů

	soubor	popis	velikost	RLE	BM	EBB	DIA	HUF	ART	GZIP	BZIP2	WINZIP	WINACE
canterbury	alice29.txt	anglický text	152 089	98.54%	90.73%	74.65%	49.49%	57.99%	34.44%	35.78%	28.40%	34.81%	33.70%
	asyoulik.txt	scénář hry	125 179	99.86%	93.46%	76.32%	52.22%	60.87%	35.45%	39.07%	31.61%	38.49%	37.57%
	cp.html	kód HTML	24 603	99.89%	98.70%	81.11%	46.22%	66.93%	34.93%	32.41%	30.98%	32.78%	33.28%
	fields.c	kód jazyka C	11 150	97.83%	88.87%	80.96%	42.79%	66.09%	30.38%	27.48%	26.52%	28.24%	29.59%
	grammar.lsp	kód LISP	3 721	97.74%	88.66%	75.84%	45.58%	64.69%	34.86%	33.16%	34.43%	35.90%	41.09%
	kennedy.xls	dokument MS Excel	1 029 744	100.01%	68.57%	89.62%	42.25%	44.11%	22.45%	19.81%	12.65%	20.85%	3.50%
	lcet10.txt	technický dokument	426 754	96.93%	92.99%	75.04%	50.46%	58.84%	34.82%	33.96%	25.24%	32.49%	29.88%
	plrabn12.txt	knižní dokument	481 861	99.86%	93.04%	74.30%	50.44%	57.50%	35.48%	40.52%	30.21%	39.02%	36.51%
	ptt5	faxová stránka	513 216	22.12%	27.00%	49.56%	12.41%	21.13%	10.26%	11.0%	9.70%	10.22%	9.59%
	sum	binární soubor pro SPARC	38 240	89.92%	76.92%	87.53%	87.19%	69.09%	38.55%	34.0%	33.75%	33.51%	28.76%
	xargs.1	GNU manuálové stránky	4 227	100.17%	95.36%	78.76%	52.76%	67.02%	41.92%	41.35%	41.64%	43.57%	48.00%
artificial	a.txt	soubor s jedním znakem	1	800.00%	900.00%	1300.00%	1100.00%	1300.00%	300.00%	2 100.0%	3500.00%	hodně	hodně
	aaa.txt	soubor se stejnými znaky	100 000	1.18%	14.30%	0.60%	0.06%	12.52%	0.01%	0.13%	0.05%	0.12%	0.41%
	alphabet.txt	opakující se abeceda	100 000	100.01%	100.01%	59.30%	0.13%	59.78%	0.09%	0.3%	0.54%	0.15%	0.48%
	random.txt	náhodně generované znaky	100 000	100.01%	99.86%	75.60%	94.09%	75.40%	82.87%	75.75%	75.68%	75.82%	78.36%
large	E.coli	genom baktérie E.coli	4 047 392	98.10%	84.67%	25.54%	29.22%	25.02%	24.54%	28.93%	26.97%	27.79%	27.36%
	bible.txt	text bible	4 047 392	97.85%	91.47%	70.89%	43.87%	54.90%	31.77%	29.45%	20.89%	27.72%	23.84%
	world192.txt	"CIA - světová fakta"	2 473 400	99.90%	91.53%	80.35%	53.30%	63.10%	35.85%	29.31%	19.79%	27.02%	21.26%
m	pi.txt	milion pozic čísla Pi	1 000 000	99.90%	96.22%	42.08%	50.59%	42.50%	41.70%	47.05%	43.17%	47.08%	46.82%
calgary	bib	textová databáze knih	111 261	100.01%	96.51%	82.09%	49.19%	65.82%	33.77%	31.67%	24.69%	30.22%	29.75%
	book1	kniha	768 771	99.98%	93.19%	73.64%	52.60%	57.32%	36.85%	40.79%	30.26%	39.18%	36.00%
	book2	kniha (troff formát)	610 856	99.95%	94.43%	75.17%	54.94%	60.38%	36.52%	33.83%	25.77%	32.38%	29.77%
	geo	geofyzikální údaje	102 400	98.46%	84.88%	111.12%	100.01%	72.33%	57.39%	66.84%	55.58%	66.71%	52.75%
	news	komunikace USENETu	377 109	97.23%	93.52%	82.7%	62.72%	65.63%	41.29%	38.40%	31.45%	36.60%	33.43%
	obj1	binární soubor pro VAX	21 504	86.95%	86.24%	96.45%	100.05%	78.26%	50.14%	48.03%	50.15%	48.33%	46.47%
	obj2	binární soubor pro Apple	246 814	107.34%	97.58%	97.18%	94.94%	78.49%	39.47%	33.03%	30.97%	32.48%	29.38%
	paper1	technický dokument	53 161	99.70%	94.63%	78.63%	55.69%	63.26%	37.16%	34.93%	31.14%	34.83%	34.70%
	paper2	technický dokument	82 199	100.00%	94.23%	74.12%	51.71%	58.36%	36.52%	36.22%	30.46%	35.73%	35.17%
	pic	černobílá faxová grafika	513 216	22.12%	27.00%	49.56%	12.41%	21.13%	10.26%	11.00%	9.70%	10.22%	9.59%
	progc	kód C	39 611	98.20%	91.00%	83.23%	53.39%	66.55%	35.94%	32.80%	30.60%	32.77%	32.98%
	progl	kód LISP	71 646	92.98%	91.12%	74.02%	47.25%	60.44%	30.56%	22.70%	21.74%	22.62%	22.48%
	progp	kód PASCAL	49 379	91.60%	84.16%	78.05%	45.31%	61.75%	29.28%	22.76%	21.69%	22.59%	22.44%
	trans	terminálová komunikace	93 695	95.79%	94.97%	82.09%	53.10%	69.50%	30.28%	20.34%	19.10%	19.75%	19.62%

Tabulka 21: celkové porovnání výsledků kompresních algoritmů

Popis tabulky:

Tabulka zobrazuje přehled zmíněných komprimačních algoritmů a jejich dosažený kompresní poměr na standardních korpusech Canterbury, Artificial, Calgary, Large, Misc. Sloupce tabulky označují jednotlivé kompresní algoritmy, žlutě (zlatě) jsou vyznačeny nejlepší dosažené hodnoty pro jednotlivé testovací soubory.

Popis zkratek testovaných kompresních algoritmů:

zkratka	popis	poznámka
RLE	Run Length Encoding	vlastní implementace
BM	bitové masky	vlastní implementace
EBB	expanze bitové báze	vlastní myšlenka, vlastní implementace
DIA	diatomické kódování	vlastní implementace
HUF	Huffmanovo kódování	vlastní implementace
ART	aritmetické kódování	knihovny v Javě http://www.colloquial.com/ArithmeticCoding
GZIP	LZW komprimace v kombinaci se statickým kódováním	standardní součást Javy
BZIP2	Burrow-Wheelerova transformace v kombinaci se statickým kódováním	knihovny v Javě http://www.kohsuke.org/bzip2
WINZIP	LZW komprimace v kombinaci se statickým kódováním	komerční produkt www.winzip.com
WINACE	LZW komprimace v kombinaci se statickým kódováním	komerční produkt www.winace.com

Výsledek:

Je vidět, že nejuniverzálnějším z testovacích algoritmů se stal algoritmus bzip2 – použití Burrows-Wheelerovy transformace v kombinaci se statistickým kódováním.

Při znovuspuštění celého komprimačního testu, jsem si všiml charakteristiky, která se asi dá vysvětlit poměrně zajímavým způsobem. Největší problém dělal všem komprimačním algoritmům, včetně WinZipu a WinAce soubor *e.coli* – čas na zpracování souboru byl někdy i několikanásobně vyšší než u jiných (i stejně velkých) souborů. Dle popisu jde o genom bakterie a z toho také můžeme usuzovat na vysvětlení daného jevu. Genom bakterie budou pravděpodobně tvořit data nabitá informacemi (těžko vytvoří něco tak složitého jako bakterie informace složená ze samých nul) s vysokou mírou neuspořádání, a proto komprimace souboru je obtížná. Dosažená úspora při komprimaci dat bude z největší části umožněna tím, že genom je zapsaný v textové podobě.

Závěr, současnost a budoucnost

Probrané kompresní algoritmy představovaly základ existujících postupů, které vznikly nad Shannonovou teorií informace. Metody probírané ke konci tohoto přehledu víceméně dosáhly dokonalosti v možném využití statistického či slovníkového přístupu k datům a nesplnil se tak můj trochu naivní předpoklad, že bych mohl přijít na nějakou novou metodu, existující vylepšit či vytvořit algoritmus kombinující dobré vlastnosti všech existujících metod.

Práce ale dobře splnila svůj účel; dovolila mi realizovat se v oblasti, která mne zajímala a našel jsem v práci i konkrétní výsledek – nalezení algoritmu, který by dosahoval kvalitnější komprese než běžné metody a který by více využíval výpočetní schopnosti dnešních počítačů. Sice jsem tento algoritmus nevymyslel sám, ale objevil jsem ho v Burrows-Wheelerově transformaci. Radost mi také způsobily některé mé nápady, u kterých jsem našel, že se jimi někdo opravdu zabývá.

Protože informací stále přibývá a nikdy nebude splněna poptávka po datovém prostoru a možnosti hledat a vyhodnocovat množství dat, je pořád vhodné ptát se, jestli jsme dosáhli limitního stavu v komprimaci dat, tak jak Shannon popisuje v souvislosti s mírou entropie. Při přemýšlení o možných modifikacích a modernizacích kompresních algoritmů a hledání moderních trendů komprese na internetu, jsem dospěl k následujícím závěrům.

- Stávající kompresní algoritmy jde ještě zlepšovat využíváním větší výpočetní síly, která je jinde než v roce 1977, kdy byl publikován algoritmus LZ77. Toto zlepšení je ale neúměrně pracné získané úspoře místa a jde v něm jen o přiblížení se hranici entropie dat.
- Všechny popisované kompresní algoritmy jsou „hloupé“. Jsou specializované na jistou charakteristiku dat a v žádném případě nejsou inteligentní. RLE kompresní algoritmus se bude pokoušet zkomprimovat zcela nevhodnou cestou náhodná data atd. Vytvořit globální kompresní algoritmus, který by vyhodnocoval, jakou metodou jakou část dat zkomprimovat, by bylo velice obtížné a stále nedokonalé. V některých případech je jednoduchý úsudek člověka stále lepší než jakýkoliv kompresní algoritmus. Jak zkomprimovat nejlépe 100.000 výskytů písmene a ? Přece zápisem „100000a“. I v takto textové podobě má náš „inteligentní“ komprimovaný výstup7 bajtů a poráží tak nejlepší kompresi, které se dosáhlo aritmetickým kódováním a měla 9 bajtů.

- Budoucnost uchovávání dat je dle mého názoru v dokonalém síťovém spojení světa, kde ukládání dat je řízeno neuronovými sítěmi. Lze si představit využití neuronové sítě pro konkrétní kompresi jednoho souboru, kde inteligentní síť vybere a aplikuje nejvhodnější komprimaci na datový celek tak, že opravdu dosáhne maximální možné míry entropie. Proč ale mít u sebe na disku stejný film jako má kolega v práci ? Ideální stav by asi byl v globálně spojené (a „100%“ propustné) síti, která by byla řízena neuronovými sítěmi vyhodnocující ideální rozmístění a distribuci dat po síti tak, aby případný výpadek části sítě neznamenal ztrátu dat, ale aby se přitom po síti data pohybovala v jen v omezeném „bezpečném“ množství kopií. Více o neuronových sítích a jejich využití ke komprimaci lze najít na internetu pod hesly „neural network compression“, o využití internetu jako obecného datového úložiště a samoučícího se komprimačního stroje pojednává *Autosophy information theory* [Authosophy].

Příloha č. 1 - Použité informační zdroje

[Abramson 63]
N. Abramson
Information Theory and Coding
McGraw-Hill, New York, 1963

[Autosophy]
Autosophy information theory
http://www.autosophy.com/information.htm

[Arithmetic]
Mark Nelson
Arithmetic Coding + Statistical Modeling = Data Compression
http://www.dogma.net/markn/articles/arith/part1.htm

[Canterbury]
The Canterbury Corpus
http://corpus.canterbury.ac.nz

[Challenge1]
The $5000 Compression Challenge
http://www.geocities.com/patchnpuki/other/compression.htm

[Challenge2]
Data Compression by Textual Substitution
http://www.cs.ucr.edu/~stelo/Offline

[Fano 49]
R.M. Fano
The Transmission of Information
Technical Report No. 65,
Research Laboratory of Electrotechnics, M.I.T, Cambridge, Massachusets

[Huffman 52]
David A. Huffman
A Method for the Construction of Minimum-Redundancy Codes
Proceedings of the IRE, volume 40, September 1952

[Huffman-princip]
Komprese pomoci Huffmanova kodovani
Viliam Holub
http://www.ms.mff.cuni.cz/~vhol8672/huff/index.html

[Java]
Java
Sun microsystems
http://java.sun.com

[JavaArith]
Compression via Arithmetic Coding in Java
http://www.colloquial.com/ArithmeticCoding/

[JavaBZip2]
bzip2 library from Apache Ant
http://www.kohsuke.org/bzip2

[Komprimace 2000]
Jan Čapek, Peter Fabian
Komprimace dat – principy a praxe
Computer Press, Praha 2000

[LZW1]
LZW Compression
L. Leurs
http://www.prepressure.com/techno/compressionlzw.htm

[LZW2]
GZip compression
http://www.gzip.org/algorithm.txt

[Moore 59]
E.N. Gilbert and E.F. Moore
Variable-Length Binary Encoding
The Bell System Technical Journal, volume 38, July 1959

[Morse]
Kódování, komprese a pan Morse
Dr. Eduard Lyko
SOŠ telekomunikační Ostrava
http://www.elektrorevue.cz/clanky/00019/

[RLE spec BMP]
Windows bitmap file format
http://www.daubnet.com/formats/BMP.html

[RLE spec PCX]
ZSoft Corporation
PCX file format – reference manual
http://www.dcs.ed.ac.uk/home/mxr/gfx/2d/PCX.txt

[RLE spec Utah]
Spencer W. Thomas
Design of the Utah RLE file format
http://netghost.narod.ru/gff/vendspec/utah/rle.txt

[Shannon 1948]
C.E. Shannon
Mathematical Theory of Communication
Bell Systems Techn. Journal, volume 27, July 1948
http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf

[Slovník 1962]
Kolektiv autorů
Příruční slovník naučný
Akademie věd, Praha 1962-67

[TransformBWT]
Burrows-Wheeler Transformation
http://en.wikipedia.org/wiki/Burrows-Wheeler_transform

[TransformDelta]
http://www.fact-index.com/d/de/delta_encoding.html

[TransformSegmentation]
Steiner, Gottfried
Statistical data compression by optimal segmentation
Wien, Wirtschaftsuniv., Diss., 1999
http://epub.wu-wien.ac.at/dyn/virlib/diss/showentry?ID=epub-wu-01_26b

[TAR]
ICE Engineering
Java TAR package
http://www.trustice.com/java/tar/

[Vymětal 1999]
Vymětal, Jan - Šilhánek, Jaroslav
Informační středisko ve firemní praxi
Ostrava: Montanex, 1996

[WinZip]
WinZip 9.0
http://www.winzip.com

[WinAce]
WinAce
http://www.winace.com

Příloha č. 2 – použité nástroje

Tak, jak na úvod patřilo poděkování vedoucímu práce, tak si myslím, že na závěr bych mohl poděkovat i kvalitním prostředkům, které jsem při práci využíval.

OpenOffice.org 1.1.1

Textový editor, který je zdarma a svými kvalitami přinejmenším na úrovni svého komerčního protivníka MS Word. Nejprve jsem se OpenOffice trochu bál a tato bakalářská práce je první delší text, který jsem v něm napsal. Editor mě mile překvapil, vše co jsem potřeboval umí, orientace v nástrojích je srozumitelná, od začátku roku již nemám na počítači nainstalovaný MS Office a nikdy se k němu nevrátím. Mezi velice milou vlastnost editoru verze 1.1.1 patří možnost exportovat dokumenty do PDF (kolik asi stojí MS Office + exportní program do PDF ?).

Mozilla 6.0

Od té doby co používám Mozillu pro prohlížení internetu a čtení pošty, nemám problémy s popup okny, mejlovými viry, práce v prostředí Mozilly je příjemná a náviková, Internet Explorer již nerad používám.

Java

Zatím stále můj velký favorit mezi programovacími nástroji. Zdarma, dostupná pro všechny OS, kvalitní.

Eclipse

Kvalitní vývojový nástroj pro Javu (opět zdarma) s velkou podporou a rozšířením. Nedosahuje kvalit komerční konkurence, ale rozdíly jsou většinou jen v drobných věcech a vynaložené peníze za případně zakoupené komerční vývojové prostředí se nevyplatí.

Příloha č. 3 - Rejstřík tabulek, obrázků a zdrojových kódů

Seznam tabulek

Seznam obrázků

Seznam zdrojových kódů

Příloha č. 4 – obsah přiloženého CD

Na příloze č. 5 – datovém CD jsou dokumenty a zdrojové kódy roztříděny do následující adresářové struktury:

<dokumenty>	adresář s různými dokumenty využitými ke studiu
<knihovny>	adresář s knihovnami třetích stran
activation.jar	knihovna potřebná k funkci JavaTAR
bzip2.src.zip	zdrovojé kódy implementace bzip2 pro Javu
colloquial_arithcode_src-1_1.zip	implementace aritmetického kódování pro Javu
javatar-2.5.zip	implementace archivátoru TAR pro Javu
<kod>	zdrojové kódy projektu
<bin>	přeložené .CLASS soubory
<lib>	knihovny potřebné k fungování zdrojových kódů
<src>	zdrojové kódy .JAVA určené pro Javu 1.4. Adresář obsahuje i zdrojové kódy knihoven pro TAR, bzip2 a aritmetické kódování
.classpath	soubor projektu pro vývojové prostředí Eclipse
.project	soubor projektu pro vývojové prostředí Eclipse
<test_files>	testovací soubory a výsledky komprimací
<The Canterbury Corpus>	popis a soubory testovací sady
<vysledky>	výsledky jednotlivých komprimací
czech_text.txt	databáze českého textu – bez diakritiky a bez konce řádků, jen anglická abeceda a mezery, text byl použit k výpočtu českých statistik
bezztratove_kombrimacni_algoritmy.sxw	bakalářská práce ve formátu OpenOffice
bezztratove_kombrimacni_algoritmy.pdf	bakalářská práce ve formátu PDF
nvynos1_2000.pdf	Metodické pokyny pro vypracování bakalářských a diplomových prací
prace_zadani.doc	zadání práce
vynos1_2004.pdf	Pokyny pro průběh státních zkoušek a obhajoby bakalářské práce

Další popisy jednotlivých souborů a případné reference na zdroje souborů jsou uvedeny předchozích částech práce.

Příloha č. 5 – opis zadání bakalářské práce

Zadání bakalářské práce

Příjmení a jméno autora: **Dušan Saiko**

Obor studia: aplikovaná informatika – kombinovaná forma studia

Příjmení a jméno vedoucího práce: **Mikulecký Stanislav, Ing.**

Přesný název práce: **Moderní algoritmy a trendy komprese dat**

Cíl práce: Zmapovat historii, současný stav a trendy v oblasti datové komprese, pojmout zajímavé a experimentální myšlenky v daném tématu.

Osnova práce:

Úvod do teorie komprese dat

Zmapování současných kompresních algoritmů

Návrh vlastního kompresního algoritmu

Implementace a zhodnocení algoritmu

Projednáno dne:

Podpis studenta: ..... Podpis vedoucího práce: .....