MATEMATICKO-FYZIKÁLNÍ FAKULA
UNIVERZITY KARLOVY

Kompresní a archivační program

Individuální softwarový projekt

Václav Honetschläger
Obor: informatika, magisterské studium
Vedoucí projektu: RNDr. Tomáš Dvořák, CSc.

1998

Děkuji RNDr. Tomáši Dvořákovi za vedení projektu a poskytnutí odborné literatury.

Prohlašuji, že jsem projekt vypracoval samostatně. Souhlasím s půjčováním a zveřejňováním projektu.

1. Úvod

Obsahem této bakalářské práce je popis kompresního a archivačního programu, jehož kompresní metodou je statistické modelování doplněné aritmetickým kódováním.

S rozvojem celosvětové sítě Internet narostl objem přenášených dat do obrovských rozměrů. Jelikož za využívání přenosových cest se platí a cesty mají omezenou kapacitu, vyvstala dnes více než kdy jindy potřeba zmenšit objem přenášených a uchovávaných dat. Kromě toho se charakter dat změnil — dříve byly přenášeny převážně relativně krátké texty, dnes po Internetu proudí multimediální datové soubory uchovávající animace, obrázky a zvuky. Tyto soubory mají obvykle velikost řádu jednotek až stovek megabytů. Vidíme, že zmenšením objemu dat lze ušetřit obrovskou přenosovou a paměťovou kapacitu. Způsob, jak zmenšit velikost dat odstraněním redundantní informace a tím zmenšit jejich délku, se nazývá komprese.

Současná doba kompresi dat přeje — cena strojového času je zanedbatelná a moderní počítače disponují obrovskou výpočetní silou. Je tedy užitečné věnovat strojový čas na kompresi dat a díky tomu ušetřit přenosovou a paměťovou kapacitu. Ačkoli se dnes vyvíjí velké množství kompresních algoritmů, některé dobré algoritmy byly vymyšleny již před delší dobou, ale doposud nenalezly praktické využití kvůli velké výpočetní složitosti nebo přílišným paměťovým nárokům.

Dvěma z těchto algoritmů jsou statistické modelování a aritmetické kódování. Statistické modelování spotřebuje velké množství paměti a strojového času; paměťové nároky aritmetického kódování jsou zanedbatelné, ne tak jeho výpočetní náročnost. Spojením těchto dvou metod ovšem získáme účinný kompresní algoritmus.

Ve druhé kapitole práce najdeme detailní vysvětlení obou algoritmů, které je velkou měrou převzato z knihy The Data Compression Book. Třetí kapitola popisuje mou implementaci těchto metod a archivačních schopností programu a myšlenky zde uvedené jsou mým dílem. Ve čtvrté kapitole se dozvíme, jak se program ovládá. Práce je doplněna experimentálními výsledky testování účinnosti a rychlosti programu a srovnáním s jinými běžně používanými i téměř neznámými komprimačními programy.

2. Použité kompresní metody

2.1 Aritmetické kódování

Aritmetické kódování je patří do skupiny statistických kodérů, tedy algoritmů, které při kódování vycházejí pouze z pravděpodobností výskytu znaků vstupní abecedy. Do stejné třídy patří i Huffmanovo kódování, to je však optimální jen v případě, že pravděpodobnosti výskytu znaků vstupní abecedy jsou záporné mocniny dvou. Představme si, že pravděpodobnost výskytu určitého znaku je $1/5$. Pro zakódování tohoto znaku by mělo být použito přibližně 2.3 bitu, ale Huffmanovo kódování by tomuto znaku přiřadilo kód délky 2 nebo 3 bity. Obě možnosti vedou k nižší kompresi. A kdybychom se pokusili zakódovat posloupnost, ve které by se vyskytovaly pouze dva znaky, byl by každému z nich, nezávisle na četnosti výskytu, přiřazen kód délky 1 bit, takže by ke kompresi vůbec nedošlo. Všechny tyto problémy řeší aritmetické kódování.

Aritmetické kódování přiřazuje vstupní posloupnosti znaků jedno číslo z intervalu $\langle 0, 1)$. Toto číslo je získáno následujícím způsobem:

1) Určíme četnosti znaků vstupní abecedy v kódované posloupnosti.
2) Interval $\langle 0, 1)$ rozdělíme na podintervaly příslušející ke znakům vstupní abecedy. Délka intervalu odpovídá relativní četnosti jeho znaku — čím vyšší četnost, tím větší interval. Záleží jen na délce podintervalu, nikoli na jeho umístění, musíme jen zajistit, aby umístění bylo stejné i při dekódování.
3) Současný interval nastavíme na $\langle 0, 1)$.
4) Načteme znak a současný interval upravíme podle intervalu tohoto znaku: interval příslušný ke znaku zmenšíme tak, aby jeho původní nule odpovídala dolní mez současného intervalu a jeho původní jedničce horní mez. Za současný interval prohlásíme ten, jehož dolní (horní) mez je dolní (horní) mezí intervalu příslušného ke znaku v současném intervalu.
5) Pokud je na vstupu další znak, opakujeme krok 3, jinak je výstupem libovolné číslo ze současného intervalu, například dolní mez.

Formálně můžeme kódovací proceduru popsat takto:

```
low = 0.0;
high = 1.0;
while ((c = getchar()) != EOF) {
  range = high - low;
  high = low + range * high_range(c);
  low = low + range * low_range(c);
  }
output(low);
```

Příklad:

Máme zakódovat posloupnost znaků "MAMKA METE". Určíme podintervaly příslušející těmto znakům:

znak	**četnost**	**interval**
A	0.2	$\langle 0, 0.2)$
E	0.2	$\langle 0.2, 0.4)$
K	0.1	$\langle 0.4, 0.5)$
M	0.3	$\langle 0.5, 0.8)$
T	0.1	$\langle 0.8, 0.9)$
mezera	0.1	$\langle 0.9, 1)$

Načteme první znak, "M", a omezíme současný interval $\langle 0, 1)$ na interval příslušející "M", tedy $\langle 0.5, 0.8)$. Poté načteme druhý znak "A", jemuž příslušející podinterval je $\langle 0, 0.2)$, a představíme si, že současný interval $\langle 0.5, 0.8)$ je pro tento znak zároveň intervalem $\langle 0, 1)$. V tomto intervalu vytyčíme podinterval $\langle 0, 0.2)$ a vypočteme, který interval mu v původním intervalu odpovídá. Je to $\langle 0.50, 0.56)$ a tento interval prohlásíme za současný. Tak pokračujeme, dokud nezakódujeme celou posloupnost. Uvedeme celý průběh kódování:

znak	**dolní mez**	**horní mez**
	0	1
M	0.5	0.8
A	0.50	0.56
M	0.530	0.548
K	0.537 2	0.539 0
A	0.537 20	0.537 56
mezera	0.537 524	0.537 560
M	0.537 542 0	0.537 552 8
E	0.537 544 16	0.537 546 32
T	0.537 545 888	0.537 546 104
E	0.537 545 931 2	0.537 545 974 4

Na výstup zapíšeme např. číslo 0.537 545 931 2.

Dekódování pak probíhá následujícím způsobem:

1) Znakům vstupní abecedy přiřadíme podintervaly intervalu $\langle 0, 1)$ stejně jako při kódování.
2) Přečteme číslo na vstupu a prohlásíme ho za současné číslo.
3) Najdeme znak, do jehož intervalu současné číslo patří.
4) Tento znak zapíšeme na výstup a interval příslušný ke znaku zvětšíme tak, aby jeho původní horní mezi odpovídala jednička a dolní mezi nula. V tomto intervalu $\langle 0, 1)$ najdeme současné číslo a vypočteme, které číslo v původním intervalu vytyčuje. Vypočítané číslo prohlásíme za současné.
5) Opakujeme postup počínaje krokem 3.

Formálně zapíšeme:

```
number = input_code ();
for ( ; ; ) {
  symbol = find_symbol (number);
  putc (symbol);
```

```
    number = (number - low_range(symbol)) /
        (high_range (symbol) - low_range(symbol));
}
```

Doposud nemáme vyřešeno, jak zjistíme konec vstupních dat. Dojdeme-li totiž při dekompresi k číslu 0, znamená to pouze libovolný (i nulový) počet opakování znaku, k němuž příslušný interval má jako svou dolní mez 0. Většinou se tento problém řeší přidáním speciálního znaku *end-of-file* do vstupní abecedy a jeho zakódováním na konci vstupních dat. Druhou možností je přenést do dekodéru i údaj o délce původní posloupnosti.

Příklad:

Budeme dekódovat posloupnost z předchozího příkladu. Načteme číslo 0.5375459312 a řekneme, že je současné. Toto číslo leží v podintervalu $\langle 0.5, 0.8)$, jemuž odpovídá znak "M". Zapíšeme tedy na výstup tento znak a číslo přepočítáme: představme si, že dolní mez intervalu $\langle 0.5, 0.8)$ označíme jako nulu a horní jako jedničku. Současné číslo nám v tomto intervalu vytyčuje nové současné číslo, tj. 0.125153104. Celý průběh dekódování zachycuje tabulka.

číslo	**interval**	**znak**
0.537 545 931 2	$\langle 0.5, 0.8)$	M
0.125 153 104	$\langle 0, 0.2)$	A
0.625 765 52	$\langle 0.5, 0.8)$	M
0.419 218 4	$\langle 0.4, 0.5)$	K
0.192 184	$\langle 0, 0.2)$	A
0.960 92	$\langle 0.9, 1)$	mezera
0.609 2	$\langle 0.5, 0.8)$	M
0.364	$\langle 0.2, 0.4)$	E
0.82	$\langle 0.8, 0.9)$	T
0.2	$\langle 0.2, 0.4)$	E
0		

Komprese je při aritmetickém kódování dosaženo tím, že pravděpodobnější znaky zmenšují interval pomaleji než ty méně pravděpodobné, a tedy je k jednoznačnému určení intervalu potřeba méně bitů.

2.2 Celočíselná implementace aritmetického kódování

Pokud budeme chtít implementovat algoritmus aritmetického kódování tak, jak byl popsán, narazíme na problém omezené velikosti a přesnosti reálných čísel reprezentovaných počítačem. Naštěstí není nutné ani rozumné používat pro aritmetické kódování čísla s pohyblivou řádovou čárkou, celé kódování lze provést ve standardní 16 nebo 32-bitové celočíselné aritmetice.

Všimněme si, že v průběhu kódování se čím dál více počátečních číslic horní a dolní meze současného intervalu shoduje a že, pokud se tyto číslice shodují, budou se shodovat i nadále, neboť meze se k sobě přibližují. Shodné počáteční úseky obou mezí můžeme tedy zapsat na

výstup a zapomenout. Naše registry, které pro implementaci použijeme, budou tudíž tvořit jakási okénka, ve kterých budou zachyceny nejvýznamnější dosud nezapsané bity čísel, se kterými pracujeme. Po zápisu shodné části posuneme okénko doprava, tzn. bity v něm doleva, a doplníme... Ale čím? Pokud bychom se striktně drželi předchozího algoritmu, výpočty by měly probíhat i na teoreticky nekonečné posloupnosti bitů v okénku nezachycených. Tento požadavek však není nutné dodržet — nic totiž nezkazíme, pokud interval, ve kterém má ležet hledané číslo, zvětšíme. Toto číslo v něm jistě bude ležet i nadále. Důležité je pouze, aby se dekodér choval stejně jako kodér.

Z našich úvah vyplývá, že dolní mez intervalu budeme zprava doplňovat nulovými bity, kdežto horní jedničkovými. Pro snazší implementaci budeme do registru horní meze zapisovat číslo o jedničku menší, než je skutečná horní mez. Tak snadno zapíšeme i číslo 1, například, použijeme-li 4-bitové registry, jako (0.)1111. Nesmíme pak ale zapomenout, že délka intervalu je

$$\text{horní mez} \; - \; \text{dolní mez} \; + 1.$$

Algoritmus s sebou nese jeden problém. Máme pravidlo, které říká: shodují-li se nejvýznamnější bity horní a dolní meze, vysuň je na výstup. Co když se však nejvýznamnější bity neshodují, ale meze jsou u sebe příliš blízko, jako je tomu např. u mezí 0110 a 1001? Zde již dochází ke ztrátě přesnosti. Ba co hůře, může se stát, že dolní mez bude 0111 a horní 1000. Zde jsme se dostali do mrtvého bodu. Rozsah intervalu je tak malý, že se při dalších výpočtech kvůli zaokrouhlení meze nezmění.

Tento problém může nastat pouze za následujících podmínek:

- nejvýznamnější bity mezí se liší, tedy horní mez začíná na 1 a dolní na 0
- druhý nejvýznamnější bit horní meze je 0
- druhý nejvýznamnější bit dolní meze je 1

Pokud k tomu dojde, hned zabráníme zhoršení situace provedením modifikovaného vysunutí: první bity ponecháme, ale celý zbytek registru posuneme o jednu pozici doleva. Druhý bit tím bude zapomenut a zprava doplníme horní mez jedničkou a dolní mez nulou. Pokud se problém opakuje, opakujeme i modifikované vysunutí. Musíme si ovšem zapamatovat počet bitů, které jsme vysunuli. Tuto informaci budeme ještě potřebovat — až nejvýznamnější bity obou mezí dokonvergují ke stejné hodnotě, dáme na výstup tuto hodnotu následovanou zapamatovaným počtem bitů opačné hodnoty, než jakou má nejvýznamnější bit. Pokud se totiž společnou hodnotou stane 1, byly by bity, které jsme vysunuli, nulové a obráceně. Modifikované vysunutí nám vlastně umožňuje beze ztráty přesnosti odložit rozhodnutí, co zapíšeme na výstup. Proč je vlastně můžeme použít a co přesně znamená, se dozvíme v alternativním výkladu algoritmu.

Příklad:

horní mez	1001	1011	1111	1011	0111	1111
dolní mez	0111	0110	0100	1001	0010	0100
čítač	0	1	2	2	0	0
stav	(1)	(2)	(3)	(4)	(5)	(6)

Při kódování se dostaneme do stavu (1). Provedeme dvě modifikovaná vysunutí a za-

pamatujeme si, že jsme odstranili dva bity (stavy (2),(3)). Po načtení dalšího znaku a příslušném výpočtu se dostaneme do stavu (4). Zde je již o nejvyšším bitu rozhodnuto. Na výstup tedy zapíšeme 1 (tento bit) a 00 (dva bity opačné než ten předchozí), vynulujeme čítač, provedeme posun (stav (5)) a potom zapíšeme i 0, na kterou obě meze začínají, a vysuneme ji (stav (6)).

Po ukončení kódování zůstává část informace o vstupní posloupnosti v použitých registrech. Proto je nutné dát na výstup (kromě případných dosud nezapsaných bitů, jejichž počet je uložen v čítači) ještě nejvyšší dva bity z libovolného čísla, které leží mezi mezemi.

V naší celočíselné implementaci musejí být celočíselné i meze intervalů, které slouží jako vstup do kodéru a výstup z dekodéru. Proto interval $\langle 0, 1)$ budeme považovat za interval např. $\langle 0, 16384)$ a příslušně přepočítáme i meze podintervalů. Abychom při výpočtech neztratili přesnost, musíme splnit dvě pravidla:

1) počet bitů udávající četnost musí být nejméně o dva menší než počet bitů registrů mezí
2) počet bitů udávající četnost plus počet bitů použitých v registru meze nesmí přesáhnout počet bitů proměnných použitých při výpočtech během kódování a dekódování

Pravidlo 2 můžeme splnit správnou volbou velikosti proměnných, např. četnost — 16 bitů, meze — 16 bitů, pomocné proměnné — 32 bitů. Splnění pravidla 1 musíme kontrolovat během výpočtu. V našem příkladě nesmí celková četnost přesáhnout $16 - 2 = 14$ bitů, tedy číslo 16383. Technika, kterou toho dosáhneme, se nazývá *přeškálování*. Její princip je, že jakmile celková četnost přesáhne ono číslo, četnosti jednotlivých znaků a potažmo celková četnost se zmenší pod danou mez. Pro zmenšení četnosti se obvykle používá dělení dvěma. Drobnou nevýhodou je malá ztráta přesnosti četností, výhodou však lepší adaptace na změny charakteru vstupních dat — celkovou četnost totiž více ovlivňují znaky, které kodér přečetl až po přeškálování. Kodér nemusí dekodéru sdělovat, že přeškáloval, dekodér to může zjistit sám, neboť dekódováním získává stejná data, která četl kodér ze vstupu.

2.2.1 Alternativní popis celočíselné implementace aritmetického kódování

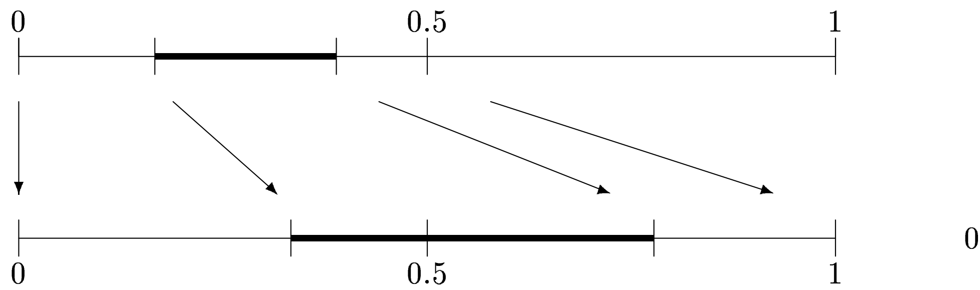
Na výše uvedený algoritmus aritmetického kódování se můžeme dívat i trochu jinýma očima:

Označme si interval, ve kterém se nachází hledané číslo, jako aktuální interval. Jeho mezemi jsou tedy horní a dolní mez z předchozího vysvětlení. Číslo, které je výstupem kodéru, hledáme půlením intervalu. Jakmile zjistíme, že se aktuální interval nachází celý v intervalu $\langle 0, 0.5)$ nebo $\langle 0.5, 1)$, expandujeme tento interval i s aktuálním intervalem tak, aby pokrýval celý interval $\langle 0, 1)$ a hledáme jen v něm. Na výstup zapíšeme 0 nebo 1 podle toho, zda jsme se zajímali o interval $\langle 0, 0.5)$ nebo $\langle 0.5, 1)$. Proberme si oba případy:

1) $0 \leq$ dolní mez $<$ horní mez < 0.5

   Lineární transformace, která splňuje

$$f : \langle 0, 0.5) \rightarrow \langle 0, 1), \text{ je } f(x) = 2x$$

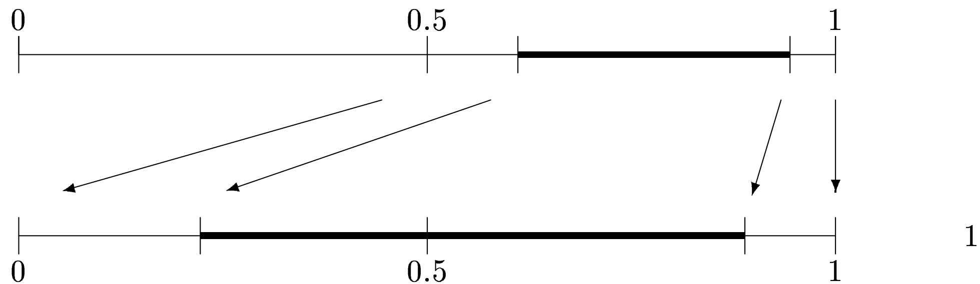


Násobení dvojkou je přesně to, co děláme, když z registrů mezí vysouváme nejvyšší nulové bity, posouváme zbytek registru o jednu pozici doleva a zprava doplňujeme dolní mez nulou a horní mez jedničkou. (Připomeňme si, že v registru horní meze není uložena horní mez intervalu, jenž je z této strany otevřen, ale číslo o 1 nižší, tedy takové největší, které do intervalu ještě náleží).

	před	**po**
horní mez	0a..a	a..a1
dolní mez	0b..b	b..b0

2) $0.5 \leq$ dolní mez $<$ horní mez < 1

$$f : \langle 0.5, 1) \to \langle 0, 1),\ \ f(x) = 2(x - 0.5)$$



Obdobně jako v předchozím případě vysunutí jedničky z nejvyšších bitů mezí, posun zbytku registru o jedno místo doleva a doplnění horní meze jedničkou a dolní nulou je přesně transformace $x \to 2(x - 0.5)$.

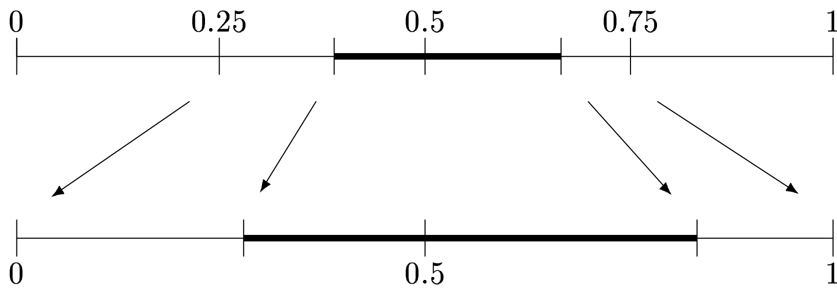
	před	**po**
horní mez	1a..a	a..a1
dolní mez	1b..b	b..b0

Vysouvání provádíme, dokud obě meze aktuálního intervalu leží ve stejné polovině intervalu $\langle 0, 1)$, tedy dokud je délka aktuálního intervalu menší než $1/2$. Pomocí vysouvání zdvojnásobujeme délku aktuálního intervalu, dokud jeho meze neleží v různých polovinách, tedy dokud není jeho délka větší než $1/2$.

Může se nám stát, že délka aktuálního intervalu je menší než $1/2$, ale jeho meze náležejí různým polovinám intervalu $\langle 0, 1)$. Kdybychom pokračovali ve výpočtech, mohli bychom ztratit přesnost, proto expandujeme interval $\langle 0.25, 0.75)$ na $\langle 0, 1)$. Tím dostáváme třetí případ.

3) $0.25 \leq$ dolní mez $<$ horní mez < 0.75

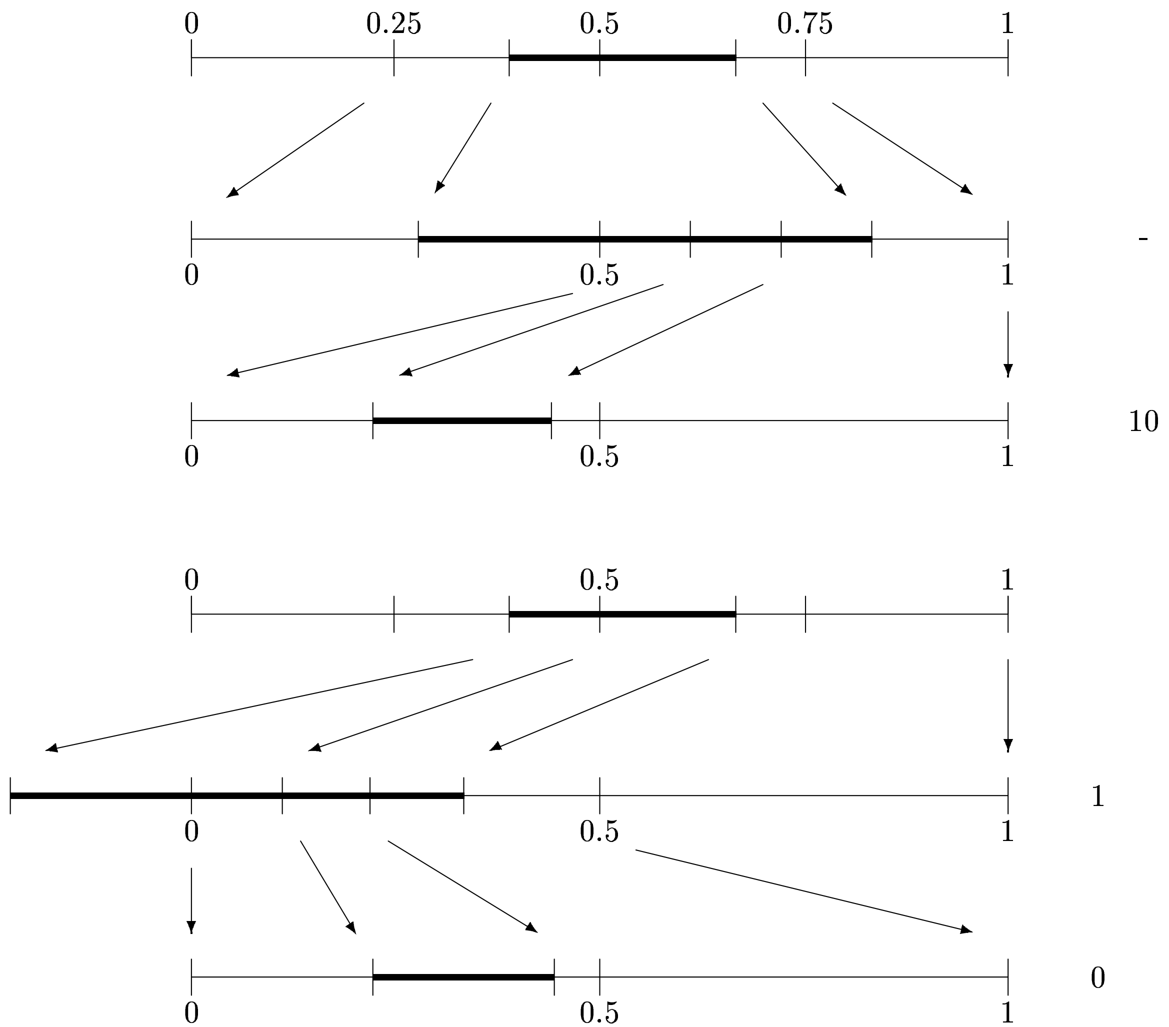
$$f : \langle 0.25, 0.75) \to \langle 0, 1),\ \ f(x) = 2(x - 0.25)$$



I zde samozřejmě odpovídají operace prováděné na registrech mezí požadované transformaci.

	před	**po**
horní mez	10a..a	1a..a1
dolní mez	01b..b	0b..b0

Zde nemůžeme na výstup zapsat, zda jsme zvolili interval $\langle 0, 0.5)$ nebo $\langle 0.5, 1)$, protože jsme ještě žádný nezvolili. Pouze si musíme zapamatovat, že jsme provedli expanzi okolo bodu 0.5. Teprve až budeme vědět, do kterého intervalu meze padnou, zapíšeme na výstup po zápisu jeho kódu tolik bitů k němu inverzních, kolik jsme provedli expanzí okolo bodu 0.5.



Na tomto místě také můžeme lépe pochopit některé věci, které jsme dříve prováděli bez podrobného vysvětlení. Tak např. po skončení kódování dáváme na výstup dva nejvýznamnější bity z nějakého čísla, které leží mezi horní a dolní mezí. Proč právě dva? Délka aktuálního intervalu je nejméně $1/2$, tedy, rozdělíme-li interval $\langle 0, 1)$ na čtyři stejné podintervaly, leží alespoň jeden z nich v aktuálním intervalu. Dva bity právě určí tento interval.

Dále jsme omezeni tím, že počet bitů udávající četnost musí být nejméně o dva menší než počet bitů registrů mezí. Je-li tomu tak a je-li délka intervalu, do kterého patří kódovaný znak, rovna jedné, budou se po přepočtu meze lišit pouze ve dvou nejméně významných bitech. Protože ale délka aktuálního intervalu mohla být pouze $1/2$, je možné, že se budou lišit pouze v posledním bitu. Rozdíl dvou bitů je tedy nejzazší hranice, kdy nám ještě obě meze nesplynou.

2.2.2 Dekomprese

Při dekódování podobně jako v neceločíselné implementaci udržujeme hodnoty horní a dolní meze a načítaného čísla. Během dekódování se hodnoty mezí mění přesně tak, jako se měnily při kódování. Jediný rozdíl je v tom, že provádíme modifikované výsuny bez toho, abychom si zapamatovali počet vysunutých bitů. Pouze je z obou mezí a načítaného čísla vysouváme.

Aritmetické kódování je pomalejší než Huffmanovo kvůli velkému počtu prováděných aritmetických instrukcí. Dává ovšem přinejmenším stejně dobré výsledky a ze všech prakticky použitelných statistických metod se kompresní poměr nejvíce přibližuje entropii.

2.3 Statistické modelování

Přeformulujme činnost aritmetického kodéru: dáme mu délku celého intervalu, který obsahuje podintervaly odpovídající četnostem znaků, a meze podintervalu odpovídající četnosti kódovaného znaku. Aritmetický kodér přepočítá meze aktuálního intervalu a případně zapíše několik bitů na výstup. Dekodéru potom předáme délku celého intervalu a dekodér podle hodnoty čísla načítaného ze vstupu určí, kam se zobrazí reprezentace tohoto čísla. Odtud určíme, do kterého podintervalu číslo patří, a tedy který znak jsme dekódovali. Potom dekodéru znovu předáme délku celého intervalu a meze našeho podintervalu a on přepočtem mezí současného intervalu "odstraní" dekódovaný znak ze vstupní posloupnosti.

Abychom dosáhli lepšího kompresního poměru, musíme buďto snížit počet znaků na vstupu, nebo zvětšit délky příslušných podintervalů, což znamená zvýšit pravděpodobnosti výskytu kódovaných znaků. Prakticky se můžeme snažit pouze o splnění druhé podmínky. Může se zdát, že pravděpodobnosti výskytů znaků ve vstupní posloupnosti se nemění, ale není to zcela pravda. Pravděpodobnosti výskytů se mohou měnit v závislosti na kontextu — na znacích, které předcházely kódovanému symbolu. Např. kódujeme-li program v jazyce C, je pravděpodobnost výskytu znaku nového řádku řekněme $1/40$, počítáme-li ji jako počet výskytů tohoto znaku dělený součtem výskytů všech znaků. Byl-li ale předcházející znak '}', je pravděpodobnost výskytu znaku nového řádku např. $1/2$. Budeme-li používat model, kde vezmeme do úvahy kontext kódovaného znaku, můžeme výrazně zlepšit kompresní poměr. Čím více bude predikce pravděpodobností odpovídat skutečnosti, tím lepší komprese dosáhneme.

Budeme se tedy zabývat tzv. *modelováním s konečným kontextem.* Myšlenka této metody je jednoduchá: pravděpodobnosti výskytu kódovaného symbolu budeme počítat na základě kontextu, ve kterém se vyskytoval. Jako *řád modelu* budeme označovat délku kontextu — počet předchozích symbolů tvořících kontext. Nejjednodušší statistické modelování je řádu 0 — pravděpodobnosti výskytu jsou počítány nezávisle na předchozích znacích. Modelování řádu 0 jsme vlastně prováděli, když jsme kódovali pouze aritmetickým kódováním a jako pravděpodobnost výskytu znaku jsme brali počet jeho výskytů dělený délkou vstupní posloupnosti.

Statistický model řádu 0 potřebuje pouze jednu tabulku pro uchovávání četností znaků vyskytujících se ve vstupní posloupnosti. Model řádu 1 jich může potřebovat až 256, jednu pro každý kontext, kterým je jeden znak. Model řádu 2 potřebuje těchto tabulek až 65536 atd. Vidíme, že roste-li řád modelu lineárně, zvyšují se paměťové nároky exponenciálně. To kromě jiného přináší potřebu přenášet obrovské množství statistických dat spolu se zkomprimovanými daty. Obvyklý způsob je totiž projít jednou vstupní data, nasbírat statistická data o výskytech a při druhém průchodu zkomprimovat data s použitím statistik. Ty je ovšem nutno přidat ke zkomprimovaným datům, aby měl dekodér podle čeho vytvořit model. V případě modelů vyšších řádů však mohou statistická data být mnohem větší než vstupní posloupnost znaků.

Řešením je *adaptivní modelování.* Při jediném průchodu daty se sbírají statistiky a zároveň se komprimuje. Kompresor i dekompresor začínají se stejným, statickým modelem, v němž je rozložení pravděpodobnosti výskytu všech znaků rovnoměrné. Kompresor zakóduje znak za použití stávajícího modelu a model aktualizuje. Dekompresor pak dekóduje znak za použití stávajícího modelu a model aktualizuje. Protože algoritmus aktualizace modelu je stejný pro kompresor i dekompresor, není nutno přenášet statistická data mezi kodérem a dekodérem. Ve stejné fázi činnosti mají kodér i dekodér vytvořen stejný model.

Drobnou nevýhodou je statický model na začátku komprese — kompresní poměr není zpočátku příliš dobrý. Eliminace přenosu statistik nicméně způsobí, že adaptivní modelování poskytuje lepší kompresi než modelování s pevnými daty.

Podstatnou nevýhodou adaptivního statistického modelování je ovšem dlouhá doba potřebná k aktualizaci modelu. Chceme-li zvýšit četnost jednoho znaku v jednom kontextu, potřebujeme, pokud se v tomto kontextu vyskytuje všech 256 znaků, přepočítat průměrně 128 kumulativních četností. Naštěstí se v kontextech vyšších řádů obvykle vyskytuje pouze několik málo znaků. Kromě toho mohou být znaky uspořádány podle četností svých výskytů, a tak většina přepočtů kumulativních četností bude probíhat na malé podmnožině znaků kontextu.

2.3.1 Použití řídícího znaku escape

Inicializaci modelu lze řešit několika způsoby. Nejjednodušší z nich, tj. v každém kontextu nastavit četnosti všech znaků na 1, abychom mohli zakódovat i znaky, které se v daném kontextu ještě nevyskytly, však není nejvhodnější. Tato metoda totiž nešetří pamětí — v každém kontextu musí být zapsány všechny znaky vstupní abecedy, kdežto ve skutečnosti se jich tam velmi pravděpodobně vyskytne pouze malá část. Druhou nevýhodou je velmi špatný kompresní poměr dosahovaný na začátku komprese. Uveďme příklad.

Příklad:

> Komprimujeme anglicky psaný text. Typicky se v textu po znaku 'q' vyskytuje znak 'u', v našem textu se v kontextu 'q' jiný znak než 'u' vyskytovat nebude. V kontextové tabulce pro znak 'q' je na počátku četnost všech 257 znaků nastavena na 1. První výskyt znaku 'u' má pravděpodobnost $1/257$ a bude tedy zakódován přibližně v 8 bitech. Druhý výskyt 'u' má pravděpodobnost $2/258$ a bude zakódován zhruba v 7 bitech. Teprve až šestnáctý výskyt znaku 'u' bude zakódován přibližně ve 4 bitech.

Důvodem tak špatné komprese je, že pravděpodobnosti ostatních 256 znaků snadno převáží pravděpodobnost výskytu 'u'. Ačkoli se jiné znaky než 'u' v kontextu 'q' pravděpodobně nevyskytnou, potřebují mít nenulové četnosti, abychom je mohli zakódovat, kdyby se v něm vyskytly.

Řešení je jednoduché: na začátku bude každý kontext obsahovat pouze jediný znak, a tím bude speciální symbol nazvaný *escape*. Potřebujeme-li do kontextu zapsat znak, který se v něm dosud nevyskytuje, zapíšeme symbol escape a přepneme se do kontextu o řád nižšího, který dostaneme ze současného kontextu vynecháním jeho prvního znaku, a postup zopakujeme. Musí ovšem existovat kontext obsahující všechny znaky, tzn. všechny znaky vstupní abecedy v něm mají četnost nastavenu na 1. Tento kontext se tradičně označuje jako kontext řádu -1, neboť do něj přecházíme, nevyskytuje-li se znak ani v kontextu řádu 0. Kontext řádu -1 není nikdy aktualizován. Například náš současný kontext je 'req' a potřebujeme zakódovat znak 'u'. Řekněme, že se znak 'u' v kódované posloupnosti dosud nevyskytoval. Aritmetickému kodéru bude předán znak escape a my přejdeme do kontextu 'eq', který je řádu 2. Ani tam ovšem 'u' nenajdeme, vygenerujeme escape a takto ještě projdeme kontexty 'q' a ''. Ani v kontextu řádu 0 'u' není, opět vygenerujeme escape a přejdeme do kontextu řádu -1, kde se znak již vyskytuje. Tato myšlenka výrazně vylepšuje dosažený kompresní poměr.

Příklad:

> Mějme stejnou situaci jako v předchozím případě. Při prvním příchozím 'u' musíme zakódovat symbol escape, jehož pravděpodobnost je ovšem $1/1$, takže na zakódování nepotřebujeme žádný bit. Pak se přepneme do kontextu nižšího řádu, v našem případě řádu 0, a hledáme 'u'. Zde ho také nenajdeme, vygenerujeme escape (tady už zabere nenulový počet bitů) a přejdeme do kontextu řádu -1. Zde se už znak 'u' vyskytuje a bude mít pravděpodobnost $1/257$, tedy na něj spotřebujeme přibližně 8 bitů. 'u' je přidáno do kontextu 'q' a je mu přiřazena četnost 1. Druhý znak 'u' se už bude v kontextu 'q' vyskytovat a jeho pravděpodobnost bude $1/2$, tedy na jeho zakódování spotřebujeme 1 bit. Až se tento znak objeví po šestnácté, bude pravděpodobnost jeho výskytu $15/16$, tudíž se na jeho zakódování spotřebuje přibližně 0.09 bitu.

Kromě toho při použití popsané myšlenky zabereme pro každý kontext jen nezbytně nutné množství paměti.

Kostru kompresního algoritmu nyní můžeme zapsat takto:

```
init_model();
init_encoder();
```

```
do {
  c = getc( input );
  if ( c == EOF ) c = EOS;
  do {
    escaped = get_values( c, &c_low, &c_high, &c_scale );
    encode( output, c_low, c_high, c_scale );
    } while ( escaped );
  update_model( c );
  if ( c != EOS ) shift_context( c );
  } while ( c != EOS );

flush_encoder( output );
done_model();
```

A takto vypadá kostra dekompresního algoritmu:

```
init_model();
init_decoder( input );

while ( 1 ) {
  do {
    get_scale( &c_scale );
    count = get_count( c_scale );
    c = find_char( count, &c_low, &c_high );
    remove_char( input, c_low, c_high, c_scale );
    } while ( c == ESC );
  if ( c == EOS ) break;
  putc( c, output );
  update_model( c );
  if ( c != EOS ) shift_context( c );
  }

done_model();
```

Dosud jsme se podrobně nezabývali tím, které kontexty aktualizujeme. Jeden přístup spočívá v přidání znaku do kontextů, ve kterých se tento znak vyskytuje, při druhém postupu se znak přidává do kontextů, podle kterých jsme znak kódovali. Znamená to aktualizovat kontexty od řádu 0 po nejvyšší kontext v prvním případě, a od kontextu, ve kterém jsme znak nalezli, po nejvyšší kontext v případě druhém.

Příklad:

> V kontextu 'req' jsme kódovali znak 'u', ten se však poprvé vyskytl až v kontextu 'eq'. V prvním případě přidáme znak 'u' do kontextů '', 'q', 'eq' a 'req', ve druhém pouze do 'eq' a 'req'.

Uvedená modifikace nejen zrychluje aktualizaci, nýbrž i mírně vylepšuje kompresní poměr.

2.3.2 Pravděpodobnost výskytu znaku escape

Pro jednoduchost jsme předpokládali, že četnost znaku escape je v každém kontextu rovna jedné. To však nemusí být dobrý odhad. Existuje několik faktorů, které ovlivňují pravděpodobnost výskytu znaku escape, to jest stavu, kdy se kódovaný znak dosud v kontextu nevyskytuje.

Prvním je počet znaků, které se v kontextu vyskytují. Přirozený požadavek je, aby v případě, že se v kontextu žádný znak nevyskytuje, byla pravděpodobnost výskytu znaku escape rovna jedné. To budeme mít zaručeno tím, že četnost escape bude kladné číslo a že četnosti ostatních znaků budou, v souladu s výše uvedenou úvahou, nulové. Jak se bude počet znaků v kontextu zvyšovat, měla by pravděpodobnost výskytu znaku escape klesat až k nule, a to v případě, že se v kontextu vyskytují všechny znaky vstupní abecedy.

Druhým faktorem je náhodnost tabulky kontextu. Jestliže některé znaky mají výrazně vyšší četnost, než je průměrná četnost znaků v tabulce, pravděpodobně se budou často vyskytovat v daném kontextu i nadále, a tím nižší bude pravděpodobnost příchodu nového znaku. Naopak, nebudou-li se četnosti znaků výrazně vychylovat od průměru, je pravděpodobnější, že se v kontextu vyskytne nový znak. Z praktického hlediska není vhodné při každé aktualizaci počítat průměr četností a odchylky znaků v celém kontextu, tedy se jako klíčový údaj uvažuje pouze četnost výskytů nejčastějšího znaku. Jeden z možných vzorců pro výpočet četnosti znaku escape v daném kontextu zde nabízíme:

c ... počet znaků vstupní abecedy
v ... počet znaků vstupní abecedy vyskytnuvších se v kontextu
n ... nejvyšší z četností znaků v kontextu

$$\text{četnost escape} = \max\left\{1, \frac{(c-v)v}{cn}\right\}$$

2.3.3 Scoreboarding

Další technikou, která sice zpomaluje použití statistického modelu, ale také zlepšuje kompresní poměr, je tzv. *scoreboarding*. Spočívá v jednoduché úvaze: jestliže v kontextu nenajdeme přišedší znak, vygenerujeme escape a přepneme se do kontextu nižšího řádu. Pokud se v něm vyskytují znaky, které se vyskytovaly v předchozím kontextu, můžeme je z tohoto kontextu dočasně vypustit. Hledaný znak se jistě mezi nimi nevyskytuje, protože v opačném případě bychom ho našli již v předchozím kontextu. Obecně, pokud v nějakém kontextu kódujeme znak, ať už je to hledaný znak nebo escape, můžeme při výpočtu intervalů dočasně zapomenout na znaky z tohoto kontextu, které se již vyskytovaly v předchozích, námi prošlých kontextech vyšších řádů. Uvedený postup dočasně zvyšuje četnost výskytu kódovaného znaku, což se příznivě projeví na kompresním poměru.

Příklad:

V kontextu 'par' se snažíme zakódovat znak 's'.

’par’		’ar’		’r’	
esc	1	esc	1	esc	1
t	4	t	3	e	31
a	1	r	2	a	17
			2		14
		a	1	o	9
		c	1	s	9
				i	8
				t	5
				m	3
				r	2
				c	2

Generování escape v kontextu ’par’ není ovlivněno, pravděpodobnost je $1/6$. V kontextu ’ar’ scoreboarding vylučuje znaky ’t’ a ’a’ a zvyšuje tak pravděpodobnost výskytu znaku escape z $1/10$ na $1/6$. V kontextu ’r’ je vyloučením znaků ’t’,’r’,’ ’,’a’ a ’c’ zvýšena pravděpodobnost výskytu ’s’ z $9/100$ na $9/60$. Scoreboarding tak snižuje počet bitů potřebných k zakódování symbolu z 9.38 na 7.91 bitu.

K mírnému vylepšení kompresního poměru můžeme využít i faktu, že řídící znaky, které slouží ke komunikaci mezi kodérem a dekodérem (jako například end-of-file), používáme velmi zřídka. Nebudeme je tedy nadále ukládat do kontextu řádu -1, kde by zbytečně snižovaly pravděpodobnosti výskytu běžných znaků, ale vyhradíme pro ně speciální kontextovou tabulku. Ta se tradičně označuje jako tabulka kontextu řádu -2, neboť do ní přecházíme, když řídící znak nenalezneme ani v tabulce řádu -1. Tato tabulka, stejně jako tabulka řádu -1, není aktualizována. Při aktualizaci tabulek kontextů musíme ovšem zajistit, aby se do nich řídící znaky nepřidávaly.

2.3.4 Další vylepšení

Při kompresi se nám může stát, že se charakter vstupních dat změnil a že data používaná pro kompresi jedné části souboru jsou pro kompresi druhé části bezcenná. V tom případě by stálo za úvahu omezit vliv současných statistik na modelování následujících dat a tím dát větší váhu statistikám získaným z dat, která teprve zkomprimujeme. Dosáhli bychom tak opětného zlepšení zhoršujícího se kompresního poměru. Tuto myšlenku můžeme uskutečnit sledováním kompresního poměru během komprese: pokud míra komprese klesne pod určitou hranici, vydělíme všechny četnosti ve všech kontextech dvěma. Bohužel není jednoduché algoritmicky odhalit změnu charakteru vstupních dat. Po vyprázdnění modelu vyšleme na výstup řídící znak flush, který je uložen v kontextu řádu -2, aby se dekodér dozvěděl, kdy má vyprázdnit model.

Poslední zlepšení se týká změn řádu modelu, podle kterého probíhá kódování. Protože na začátku komprese, než se zaplní statistické tabulky, je generováno mnoho znaků escape, je lepší začít kódovat podle modelu řádu 0 a, jak se tabulky postupně zaplňují, přecházet k modelům vyšších řádů, dokud nedosáhneme maximálního řádu. Statistiky by se ovšem sbíraly hned od začátku pro model nejvyššího řádu.

Statistické modelování ve spojení s aritmetickým kódováním poskytuje vynikající kompresní

poměr. Cenou za to je pomalá komprese a potřeba velkého množství paměti. Vhodnou implementací lze však zvýšit rychlost a snížit paměťové nároky na únosnou míru.

3. Popis programu

VA je kompresní a archivační program určený pro použití na všechny typy souborů. Má implementovánu jednu kompresní metodou, a tou je statistické modelování maximálně 4. řádu doplněné aritmetickým kódováním. Není-li ovšem komprese úspěšná, tedy má-li zkomprimovaný soubor větší velikost než originál, je původní soubor namísto komprese beze změny zkopírován.

Program je psán čistě v jazyce C. Nezvolil jsem moderní programovací jazyk C++, neboť práce s objekty snižuje rychlost provádění programu a navic se mi nepodařilo nalézt v mém programu část, v níž by mi objektové možnosti C++ poskytovaly výhodu oproti jazyku C. Taktéž jsem pro zápis časově kritických částí programu nepoužil jazyk asembleru, protože by se program stal méně srozumitelným i přenositelným.

Program se ovládá pouze nastavením parametrů na příkazové řádce. Tento nemoderní způsob ovládání jsem zvolil mj. proto, že v grafickém rozhraní je obtížné vyvolat některé akce (např. archivovat soubory z adresářů "`prog1/doc`" a "`prog2/doc`") tak, aby uživatel mohl jasně vidět, s jakými soubory se bude pracovat. Ovládání pomocí příkazové řádky je také, alespoň pro mě, přehlednější — všechny potřebné údaje vidím na jediné řádce.

3.1 Implementace aritmetického kódování

V programu je použita celočíselná implementace aritmetického kódování tak, jak je popsána ve 2. kapitole. Registry potřebné pro kódování a dekódování jsou bezznaménková 16-bitová čísla, pro pomocné proměnné potřebné při práci program používá, přesně podle popsaného postupu, bezznaménková 32-bitová čísla.

3.2 Implementace statistického modelování

Statistické modelování je založeno na všech myšlenkách uvedených ve 2. kapitole s výjimkou části 2.3.4.

3.2.1 Struktura kontextových tabulek

Kontexty jsou uloženy ve stromové struktuře, kde synovi odpovídá kontext s přidaným jedním znakem na konec; jako otce nějakého uzlu označuji uzel s odebraným jedním znakem ze začátku. V mé terminologii je tedy otec syna některého uzlu, stejně jako syn otce uzlu, pouze nějaký kontext stejného řádu, jako má náš uzel. Popsané uspořádání ukazatelů je výhodné při změně kontextu, která se děje po zakódování každého znaku.

Každá tabulka kontextu je následující struktura:

```
typedef struct _Context Context;
struct _Context {
  int     number;
  Context *parent;
  s16     *chars;
  u16     *count;
```

```
Context **sons;
};
```

number
: počet znaků vyskytujících se v kontextu, jsou počítány pouze běžné znaky bez znaku escape

parent
: ukazatel na kontext nižšího řádu, tedy, jedná-li se o kontext "aX", ukazuje "parnet" na kontext "X"

chars
: ukazatel na pole znaků, které se v kontextu vyskytly, znaky jsou setříděny podle četnosti sestupně; znak, kterému je zvyšována četnost, je zapsán před znaky se stejnou nebo menší četností, znak escape není zapisován

count
: ukazatel na pole četností, ty odpovídají znakům z pole chars, poslední zapsanou četností (tedy na pozici number) je vždy četnost znaku escape

sons
: ukazatel na pole ukazatelů na kontexty vyšších řádů, tedy, jedná-li se o kontext "X", na kontexty tvaru "Xa"

Všechny tabulky kontextů jsou dynamicky alokované. Kontexty nejvyššího řádu nemají syny a ukazatel "sons" je tedy u nich nastaven na NULL. Kontext řádu -2 nemá otce.

V kontextu řádu -1 jsou zapsány všechny znaky vstupní abecedy a symbol ESC (escape), který zde slouží k přechodu do kontextu řádu -2. V něm jsou uloženy pouze dva řídící znaky — EOS (end-of-stream) a RCN (reduce context). Význam druhého z nich bude objasněn později. Řídící znaky jsou ohodnoceny zápornými čísly.

Četnosti znaků v kontextu jsou skutečně uloženy jako četnosti, nikoli jako kumulativní četnosti — přepočet probíhá pouze v případě potřeby. Tabulka při něm není přepisována, ale výpočet je proveden v pomocných proměnných. Při přepočtu je využito scoreboardingu.

Při přeškálování je snížena četnost znaků pouze v kontextu, který přetekl. Pokud se přeškálováním sníží četnost nějakého znaku na nulu, liší se postup dle toho, jedná-li se o kontext nejvyššího řádu. V kontextu nejvyššího řádu takový znak skutečně vypustím, ovšem nejde-li o kontext nejvyššího řádu, znaku bude přiřazena četnost jedna, aby nebyly zrušeny tabulky synovských kontextů, které obsahují informace, jež se nám mohou hodit. Je samozřejmě možné i v tomto případě znak vypustit a zrušit příslušné synovské kontexty. Experimentálně jsem však ověřil, že k takovému rušení dojde málokdy, většinou v případě, že data jsou špatně zkomprimovatelná a komprese je neúspěšná. Příliš tedy nezáleží na tom, která z variant je použita.

3.2.2 Řád kontextu a paměťové nároky

Program umí používat statistické modelování řádů 0 až 4. Maximální řád kontextu je implicitně zapsán v systémově závislých souborech, je ale samozřejmě možné přenastavit jej při spuštění komprese. Ne vždy je pro predikci používán model tohoto řádu. Na úplném začátku

komprese se používají modely nižších řádů, protože žádný kontext maximálního řádu ještě není k dispozici. To znamená, že pro kódování prvního znaku se použije kontext řádu 0, pro kódování druhého kontext řádu 1 atd. a podle kontextu s maximálním řádem modelujeme teprve tehdy, až načteme dostatečný počet znaků.

Druhý důvod pro snížení maximálního řádu modelu vychází z omezeného množství dostupné operační paměti. Dosáhne-li program hranice paměti, kterou může používat, sníží o jednu řád modelu a zruší všechny kontexty nejvyššího řádu. Tím sníží množství obsazené paměti přibližně na 15–50% původní velikosti, cenou za to je horší kompresní poměr. Po snížení řádu modelu je na výstup vyslán již dříve zmiňovaný znak RCN.

Velikost paměti, kterou může program pro statistický model využít, je zapsána v systémově závislých souborech. Jako velikost obsazené paměti se bere to množství, o které program požádal operační systém. Typicky je programu přidělena větší část paměti, než kterou chtěl (ačkoli ji nemůže využít), neboť operační systém přiděluje paměť po větších jednotkách, než je 1 byte. Jistě by byl lepším měřítkem obsazení paměti objem skutečně zabraný programem, pro zjištění tohoto údaje však neexistuje jednotná strategie, a na některých operačních systémech je nemožné ho získat. Musel jsem se tedy uchýlit k měření velikosti obsazené paměti výše popsanou cestou a spoléhat na to, že, jak ukázaly experimentální výsledky, se množství skutečně přidělené paměti pohybuje přibližně okolo dvojnásobku paměti, o kterou bylo žádáno.

Přesto se může stát, že program nebude mít pro práci dostatek paměti. O paměť totiž žádá postupně s tím, jak se zvětšují kontextové tabulky. Nepožaduje na začátku maximální množství paměti, které smí použít (neboť by to bylo plýtvání), ani nekontroluje, zda by mu tolik paměti mohlo být přiděleno (to se může na víceprocesovém operačním systému změnit). Pokud dojde k vyčerpání dostupné paměti ještě před tím, než program vyčerpal množství, o které směl požádat, program skončí s chybou.

3.2.3 Pravděpodobnost výskytu znaku escape

Pro výpočet pravděpodobnosti znaku escape jsem nepoužil vzorec uvedený ve 2. kapitole, ale vzorec

c … počet znaků vstupní abecedy
v … počet znaků vstupní abecedy vyskytnuvších se v kontextu
n … nejvyšší z četností znaků v kontextu

$$p(\mathrm{ESC}) = \max\left\{1, \frac{(c-v)v}{c} - \frac{n}{4}\right\},$$

neboť ten, jak jsem po experimentech s různými vzorci zjistil, poskytuje o něco lepší kompresní poměr.

3.3 Práce s archivem a archivační funkce

Program nabízí několik základních archivačních funkcí. Mezi ně patří přidání souborů do archivu, extrakce souborů z archivu, rušení souborů z archivu, testování a výpis archivovaných

souborů a méně obvyklé přesouvání archivovaných souborů mezi archivy. Přesné provedení archivačních funkcí lze ovlivňovat přepínači.

3.3.1 Práce s archivem

V archivním souboru jsou uloženy archivované soubory. Kdykoli se s archivem pracuje, zůstává až do konce práce v původním stavu, neboť je dle požadavků vytvářen (z původního archivu a případně z diskových souborů) nový archiv jiného jména. Teprve když je práce skončena, je původní archiv zrušen a nový archiv dostane jméno starého. Takto je minimalizována ztráta dat při neobvyklé události (nedostatek paměti, přerušení od uživatele, výpadek proudu). Pokud v archivu na konci práce nezbydou žádné soubory, archiv je zrušen.

Pro přístup k archivnímu souboru je deklarována struktura ARC obsahující následující údaje:

```
typedef struct {
 FILE *file;    /* soubor - archiv */
 char *name;    /* jméno souboru - archivu */
 u16 count;     /* počet přečtených/zapsaných souborů z/do archivu */
 u32 filepos;   /* pozice v~archivu pro další čtení/zápis */
 u16 mode;      /* mód otevření archivu */
 } ARC;
```

S archivem pracuje několik funkcí z modulu archand.c. Uveďme nejdůležitější z nich:

ARC *openarc (char *arcname, u16 mode, u16 buffsize);

Funkce otevře archivní soubor se jménem *arcname* v módu *mode*, nastaví velikost bufferu pro přístup k souboru na *buffsize* a provede další počáteční nastavení.
Módy otevření archivního souboru:
A_WR … otevře archiv pro zápis
A_RD … otevře archiv pro čtení
A_UP … otevře archiv pro aktualizaci - archiv je vlastně otevřen pro čtení, ale kontroluje se, zda máme práva k následnému přepisu archivu

Funkce vrátí ukazatel na strukturu ARC, který je později využíván dalšími funkcemi pro přístup k archivu, nebo NULL, nelze-li soubor otevřít. Program také může při volání této funkce skončit s fatální chybou, pokud

- nebyl dostatek paměti
- archiv otevíraný pro aktualizaci existuje, ale nemáme k němu právo zápisu
- archiv otevřený pro čtení má špatnou hlavičku
- nelze zapsat hlavičku archivu

int readarc (ARC *arc, FileHeader *fh);

Funkce obnoví uloženou pozici v archivu *arc*, načte hlavičku souboru do *fh* a uloží pozici pro další čtení. Vrací 0, pokud v archivu už není žádný soubor, a 1 v opačném případě.

void writearc (ARC *arc, FileHeader *fh);

Funkce uloží současnou pozici a zapíše hlavičku souboru *fh* do archivu *arc*.

void writerearc (ARC *arc, FileHeader *fh);

Funkce obnoví pozici v archivu, zapíše hlavičku souboru a skočí na konec archivu.

void writeagainarc (ARC *arc, FileHeader *fh);

Funkce obnoví pozici v archivu, zapíše hlavičku souboru a zkrátí soubor na aktuální pozici.

void closearc (ARC *arc);

Funkce zavře archiv, pokud byl otevřen pro zápis, je předtím aktualizována jeho hlavička. Jsou uvolněny příslušné paměťové struktury.

Nutnost použití tří funkcí pro zápis hlavičky do souboru vyplývá z následující úvahy: před tím, než do archivu začneme zapisovat vlastní zkomprimovaná data, musí v něm být zapsána hlavička o správné velikosti. Poté zapíšeme zkomprimovaná data. Byla-li ovšem komprese neúspěšná, je nutno opět skočit těsně za hlavičku (třeba tak, že obnovíme pozici v archivu a zapíšeme stejnou hlavičku jako v předchozím případě) a zkrátit archiv tak, aby hned za hlavičkou končil. Poté musíme zkopírovat archivovaný soubor do archivu. Po zápisu dat se musíme vrátit na hlavičku, přepsat ji hlavičkou s vyplněnými údaji a opět skočit na konec archivu.

Celý zápis do archivu potom probíhá následujícím způsobem:

1) archiv se otevře funkcí openarc() v zápisovém módu
2) funkcí writearc() se zapíše hlavička souboru, která má vyplněny přinejmenším položky path a name (kvůli správné délce hlavičky)
3) zapíše se zkomprimovaný soubor
4) byla-li komprese neúspěšná, obnoví se pozice v archivu, zapíše se funkcí writeagainarc() hlavička o správné délce a překopíruje se původní soubor
5) funkce writerearc() přepíše hlavičku souboru správně vyplněnou hlavičkou a skočí se na konec archivu
6) počínaje krokem 2 se postup může opakovat
7) archiv se zavře funkcí closearc()

Čtení z archivu je mnohem jednodušší:

1) archiv se otevře funkcí openarc() ve čtecím módu
2) přečte se hlavička souboru funkcí readarc()
3) přečte se zkomprimovaný soubor
4) počínaje krokem 2 se postup může opakovat
5) archiv se zavře funkcí closearc()

3.3.2 arc_add()

Pro přidávání souborů do archivu je volána funkce arc_add(), která má následující deklaraci:

```
void arc_add (char *arcname, int order, int number, char **mask,
                                                      int mode);
```

Soubory, které jsou zadány maskami mask, funkce zkomprimuje za použití modelu řádu order do archivu arcname; number je počet masek a mode je mód ovlivňující, které soubory jsou do archivu přidány.

Přidání může probíhat ve třech módech: přidávání, aktualizace a obnova.

ADD ... přidávání:
Do archivu jsou přidány všechny soubory vyhovující maskám.

UPD ... aktualizace:
Do archivu jsou přidány všechny soubory, které vyhovují maskám a jsou novější než verze v archivu nebo v archivu vůbec nejsou.

FRS ... obnova (aktualizace archivovaných souborů):
Do archivu jsou přidány všechny soubory, které vyhovují maskám a jsou novější než verze v archivu.

V žádném svém módu funkce arc_add() neruší soubory z archivu, obecně pouze přidává další soubory nebo nahrazuje stávající jinými verzemi.

Uvedu přehlednou tabulku, zda jsou soubory z disku přidávány do archivu v jednotlivých módech.

	soubor v archivu		
mód	je	není	1 ... je přidán
ADD	1	1	0 ... není přidán
UPD	*	1	* ... je přidán, jen pokud je novější
FRS	*	0	než verze v archivu

Nyní popíšu algoritmus, na jehož základě je vybudována funkce arc_add():

- vytvořím seznam souborů na disku, které odpovídají maskám
- otevřu původní archiv pro aktualizaci a nový archiv pro zápis
- načítám postupně soubory z původního archivu a:
  - není-li soubor na disku, zkopíruji ho z původního archivu do nového
  - je-li na disku, potom:
    - je-li mód roven ADD nebo je-li soubor na disku novější než v archivu, označím v seznamu, že soubor je i v archivu
    - jinak zkopíruji soubor z původního archivu do nového a zruším ho ze seznamu
- zavřu původní archiv

Nyní jsou do nového archivu zkopírovány všechny soubory z původního archivu, které by do něj zkopírovány být měly. V seznamu jsou zapsány ty soubory, které mám do nového archivu

přikomprimovat z disku. U každého souboru je též uvedeno, zda se jeho verze v archivu nachází či nikoli.

- jsem-li v módu FRS, zruším ze seznamu soubory, které jsou jen na disku (tedy není u nich označeno, že jsou v archivu)
- postupně procházím seznam a zkouším přidat soubor z disku:
  - lze přidat: přidám a zruším ze seznamu
  - nelze přidat (soubor neexistuje, nemám práva ...):
    - vypíšu varování a:
      - pokud soubor není v původím archivu, zruším ho ze seznamu
      - pokud je původním archivu, v seznamu ho nechám

Teď mám z disku do archivu přidané všechny soubory, které jsem chtěl mít přidané a které přidat šly. V seznamu jsou soubory, které se nepovedlo přidat, ale které jsou v původním archivu.

- není-li seznam prázdný:
  - otevřu znovu původní archiv pro aktualizaci
  - postupně z něj načítám soubory a narazím-li na soubor ze seznamu, zkopíruji ho do nového archivu
- zavřu archivy a ukončím práci

Tento postup je výhodný ze dvou důvodů: původní archiv procházím sekvenčně, a to maximálně dvakrát, ve většině případů však jen jednou. Dále, pokud se v archivu vyskytuje soubor, ale má být nahrazen jinou verzí z disku a tato operace se nepovede, je alespoň v archivu zachována původní verze.

3.3.3 arc_addarc()

Tato funkce přidává soubory z jednoho archivu do druhého bez provádění dekomprese a opětovné komprese. Má následující deklaraci:

```
void arc_addarc (char *arcname, char *addname, int number,
                                                char **mask, int mode)
```

Funkce přidá soubory vyhovující masce *mask* z archivu *addname* do archivu *arcname*. Počet masek je *mask* a mód *mode* ovlivňuje, které soubory se do archivu přidají. Funkce je velmi podobná funkci arc_add(). Módy jsou stejné a mají stejné významy jako ve funkci arc_add().

Popíšu algoritmus činnosti funkce:

- vytvořím seznam souborů v původním archivu
- otevřu přidávaný archiv pro čtení a nový archiv pro zápis
- postupně procházím soubory v přidávaném archivu a:
  - není-li soubor v seznamu, potom, pokud mód není roven FRS, soubor překopíruji

z přidávaného do nového archivu
  - je-li soubor v seznamu a je-li mód roven ADD nebo je-li soubor v přidávaném archivu novější než v původním, zkopíruji ho do nového archivu a záznam ze seznamu vymažu

- otevřu původní archiv pro aktualizaci
- postupně z archivu načítám soubory a najdu-li v něm soubor vyskytující se v seznamu, zkopíruji ho do nového archivu a záznam ze seznamu vymažu
- zavřu archivy a ukončím práci

Původní archiv takto projdu dvakrát, přidávaný archiv jednou.

3.3.4 arc_extract()

Pro dearchivaci se volá funkce arc_extract() s následující deklarací:

```
void arc_extract (char *arcname, char *base, int number,
                                          char **mask, int mode);
```

Funkce z archivu se jménem *arcname* extrahuje soubory s maskami *mask* do adresářové struktury s kořenem base. Počet masek je označen jako *number* a *mode* je mód extrakce. Extrakce může probíhat ve dvou módech:

EXT
extrakce z archivu na disk

TST
testování archivovaných souborů (extrakce z archivu na nulové zařízení)

Jediný podstatný rozdíl mezi oběma módy je v místě, kam se extrahované soubory zapisují - v případě TST je to nulové zařízení.

Pokud se soubory extrahují v módu EXT, potom, existuje-li již soubor, jenž má být vytvořen, je uživatel otázán, zda má soubor být přepsán.

3.3.5 Ostatní funkce pro práci s archivem

Funkce arc_list() z archivu se jménem *arcname* vypíše všechny soubory vyhovující maskám *mask*.

```
void arc_list (char *arcname, int number, char **mask);
```

Funkce arc_delete() vymaže z archivu *arcname* soubory s maskou *mask*. Pracuje tak, že z původního do nového archivu jsou zkopírovány pouze soubory nevyhovující masce.

```
void arc_delete (char *arcname, int number, char **mask);
```

3.3.6 Formát archivního souboru

Všechny položky jsou ve formátu little endian (nižší bity - vyšší bity)

Archiv:

délka	položka
4	hlavička archivu
.	hlavička archivovaného souboru
.	archivovaný soubor
.	hlavička archivovaného souboru
.	archivovaný soubor

Hlavička archivu:

délka	položka
2	identifikátor archivu - 'VA' (0x56 0x41)
2	počet archivovaných souborů

Hlavička archivovaného souboru:

délka	položka
1	bity 7..4 ... verze formátu archivovaného souboru bity 3..0 ... použitá kompresní metoda (0..4 - řád statistického modelování, 0xE - uložení bez komprese)
4	délka zkomprimovaného souboru
4	délka původního souboru
2	CRC 16
4	čas poslední modifikace souboru v UNIXovém formátu (počet sekund od 1.1.1970)
1	délka cesty k souboru
n	cesta k souboru
1	délka jména souboru
n	jméno souboru
1	typ systémově závislých informací
1	délka systémově závislých informací
n	systémově závislé informace

3.4 Práce s cestami a názvy souborů

Programu je možno zadávat kromě obvyklých jmen souborů i tzv. žolíky, čili jména souborů, která nejsou určena jednoznačně. Program rozeznává dva speciální expanzní znaky:
'*' ... odpovídá jakémukoli řetězci (i nulové délky)
'?' ... odpovídá právě jednomu libovolnému znaku

Příklad:

> '?b?*' odpovídá všem jménům souborů, která jsou délky alespoň 3 a jako druhý znak se v nich vyskytuje 'b'.

Žolíky je možno zadávat pouze ve názvech souborů, nikoli v cestách.

Některé archivační programy vypouštějí části cest, které jsou relativní, např. "`../../prog/doc`" bývá uloženo jako "`prog/doc`". Tento přístup však vede k tomu, že pokud archivujeme

podadresářovou strukturu a pak tyto soubory opět z archivu extrahujeme a je-li náš pracovní adresář stejný jako při archivaci, soubory budou extrahovány jinam. Mně tento postup nevyhovuje, a proto jsem se rozhodl ukládat názvy cest do archivního souboru tak, jak byly zadány. Vhodným zadáním adresáře, do kterého se bude extrahovat, lze ovšem nasimulovat výše popsané chování archivačních programů. Např. cesty souborů v archivu začínají "`..`", ale my chceme tyto soubory extrahovat do pracovního adresáře. V tom případě pro dosažení žádaného výsledku zadáme jako adresář pro extrakci např. "`./tmp`". Tento adresář vůbec nemusí existovat.

3.5 Přenositelnost programu

Jedním z požadavků, které jsou čím dál více kladeny na programy, je snadná přenositelnost na jiné hardwarové platformy a jiné operační systémy. Tato snaha je pochopitelná a samozřejmá — je-li program už jednou napsán, vyplatí se, aby fungoval i na jiném typu počítačů, než pro který byl vytvořen. I já jsem šel vstříc tomuto trendu a snažil jsem se napsat program tak, aby byl lehce portovatelný. Strojově nebo systémově závislé věci jsem oddělil do souborů machine.c a machine.h. Pro každý operační systém či překladač by měl být vytořen adresář, který tyto soubory obsahuje.

Program jsem vytvářel a testoval na operačním systému Linux Red Hat s překladačem gcc a na operačním systému MS-DOS s překladačem Borland C++. Abych si ověřil, jak dobře se mi povedlo oddělit systémově závislé věci, pokusil jsem se na konci práce přenést program na překladač Watcom C++ fungující pod MS-DOSem. Po několika minutách přepisování systémově závislých souborů a změně názvu jednoho standardního hlavičkového souboru jsem program přeložil a ten bez problémů fungoval.

Obecně by program měl být snadno přenositelný na překladače a operační systémy, které respektují normu POSIX.

Systémově závislé funkce jsou deklarovány v souboru machine.h a implementovány v souboru machine.c. Jejich popis uvádím.

void md_init_environ ();

Inicializuje systémově závislé prostředí — např. nastavuje ovladače přerušení.

void md_int_handler (int intr);

Ovladač přerušení. Vypisuje zprávu a ukončuje program s návratovým kódem 128+číslo přerušení.

uchar *md_getfattr (char *name);

Zjistí systémově závislé atributy souboru zadaného jména a vrátí je jako pole typu uchar. Tento údaj bude uložen v archivním souboru. Návratová hodnota je v této funkci statická.

void md_setfattr (char *name, uchar *attr);

Obnoví atributy souboru zadaného jména tak, jak byly uloženy.

char *md_makeattrstr (uchar *attr, char mtype);

Argumentem jsou atributy souboru tak, jak byly uloženy, a identifikátor OS, pod kterým byly atributy vytořeny. Vrátí řetězec symbolických jmen atributů, který bude použit při výpisu souborů. Návratová hodnota je v této funkci statická.

char *md_stripname (char *fullname);

Vrátí část plné cesty k souboru odpovídající jeho jménu. Návratová hodnota je ve funkci alokována a po použití by měla být uvolněna.

char *md_strippath (char *fullname);

Vrátí část plné cesty k souboru odpovídající jeho cestě. Návratová hodnota je ve funkci alokována a po použití by měla být uvolněna. V případě, že cesta je prázdná, je vrácen prázdný řetězec.

void md_concatname (char *path, char *name);

Vytvoří plnou cestu k souboru ze zadané cesty a jména a uloží ji do path. Nejsou prováděny paměťové realokace.

void md_concatpath (char *base, char *path);

Spojí dvě zadané cesty a výsledek uloží do base. Nejsou prováděny paměťové realokace.

char *md_tovapath (char *path);

char *md_tovaname (char *name);

char *md_tomdpath (char *path);

char *md_tomdname (char *name);

Převede cestu/jméno souboru z/do formátu VA. Návratové hodnoty jsou v těchto funkcích statické.

int md_cmpname (char *mask, char *name);

Porovná zadanou masku obsahující žolíky se zadaným jménem souboru a zjistí, zda si odpovídají.

void md_fitdirname (char *newpath, char *path);

Upraví jméno adresáře, např. převede "a/./b" na "a/b" nebo "a/../b" na "a". Ukončující oddělovač (např. "/") je odstraněn.

char *md_fitarcname (char *name);

Přidá k zadanému jménu příponu VA archivu dle zvyklostí OS.

3.6 Návratové hodnoty programu

Správně napsaný program by měl při svém skončení předat operačnímu systému tzv. návratovou hodnotu. Ta informuje operační systém a ostatní uživatelské procesy, vyžádají-li si tuto informaci, zda program provedl úspěšně svou činnost, či zda a kvůli jaké chybě skončil předčasně. Po úspěšném průběhu se vrací 0, po neúspěšném nenulová hodnota.

Můj program rozeznává dvě úrovně závažnosti chyb: varování a závažné chyby. Varování se objeví, když program nemůže provést nějakou operaci, ale jiné operace provést může (např. nemám práva ke čtení souboru, který chci archivovat). Varování není důvodem k ukončení programu. Závažná chyba je naopak taková situace, ve které nemá smysl pokračovat v provádění programu. Vzniká např. když není možno vytvořit archivní soubor.

0 Bez chyby. Program úspěšně provedl zadanou činnost, ale je možné, že se objevila varování.
1 Nedostatek paměti. Program žádal o paměť, ale požadavek nemohl být uspokojen.
2 Nesprávné parametry. Program byl spuštěn s nesprávnými parametry.
3 Nelze otevřít vstupní archivní soubor. Tato chyba nastává, když se pokoušíme otevřít archiv, ze kterého se bude číst nebo který se bude aktualizovat, a tento archiv neexistuje.
4 Nelze číst z archivu. Chyba nastává, jestliže archiv byl sice úspěšně otevřen, ale v průběhu čtení jsme předčasně narazili na konec souboru nebo byl soubor fyzicky poškozen.
5 Nelze otevřít výstupní archivní soubor. Chyba nastává, pokoušíme-li se otevřít nový archiv pro zápis. Možných důvodů, proč chyba nastala, je mnoho, nejčastějšími z nich je neexistující cesta k souboru, práva neumožňující zápis do adresáře, ve kterém má být archiv vytvořen, a práva neumožňující čtení adresářů na cestě k archivu.
6 Nelze zapisovat do archivu. Archiv byl ovšem úspěšně otevřen pro zápis. Nejčastějším důvodem chyby je zaplněný disk.
7 Archiv je poškozen. Chyba nastane v případě, že při extrakci z archivu nesouhlasí zaznamenaný kontrolní součet s vypočteným. Archiv byl softwarově nebo hardwarově poškozen.
8 Čtený soubor není archiv VA. Soubor na svém začátku neobsahuje hlavičku typickou pro archiv VA.

Dalším důvodem k ukončení programu jsou přerušení. Na přerušení program reaguje ukončením své činnosti s návratovou hodnotou 128+číslo přerušení. Přerušení a jejich kódy se liší v závislosti na systému.

Posledním důvodem k předčasnému ukončení programu jsou situace, kdy nastane stav, ke kterému nikdy nemělo dojít - přestal platit invariant nějakého algoritmu. Znamená to, že byl špatně implementován algoritmus nebo nesprávně zvolen invariant. Obojí je důvodem k okamžitému ukončení programu. Návratová hodnota je ve všech případech 255.

3.7 Popis modulů

archand.c

Modul obsahuje funkce pro manipulaci s archivem, které byly popsány v kapitole 3.3.1, a hlavní archivační funkce popsané v kapitolách 3.3.2 až 3.3.5.

arcoder.c

Modul implementuje aritmetické kódování podle popisu v kapitolách 2.2 a 3.1. Je využíván modulem smarc.c.

bitio.c

Modul zajišťuje funkce pro přístup k souboru po bitech. Obsahuje funkce pro otevření, zavření a zjištění stavu souboru. Příslušný hlavičkový soubor obsahuje makra pro čtení a zápis po bitech. Modul je používán modulem archand.h pro čtení a zápis komprimovaných souborů.

crc16.c

Modul obstarává funkce pro výpočet 16-bitového CRC. Je používán moduly archand.c a smarc.c.

dirhand.c

Modul poskytuje funkce pro otevření, čtení obsahu a uzavření adresáře. K adresáři se vždy přistupuje rekurzivně - při jeho čtení jsou přečteny všechny jeho soubory i soubory všech jeho podadresářů.

error.c

Modul obsahuje chybové kódy a funkce pro výpis varování a závažných chyb. Příslušný hlavičkový soubor obsahuje makro assume(), které kontroluje platnost invariantů, jak bylo popsáno v posledním odstavci kapitoly 3.6. Modul je využíván většinou ostatních modulů.

files.c

Modul poskytuje převážně funkce pracující se seznamy souborů. Lze zde nalézt také funkci pro vytváření adresářů a jiné. Je využíván modulem archand.c.

machine.c

Modul implementuje systémově závislé funkce — ty z velké části slouží k manipulaci se jmény souborů a adresářů. V příslušném hlavičkovém souboru jsou uložena makra a symbolické konstanty specifické pro daný operační systém. Více informací podává kapitola 3.5.

misc.c

Modul obsahuje tématicky nezařaditelné funkce.

smarc.c

Modul zajišťuje statistického modelování, jak bylo popsáno v kapitolách 2.3 a 3.2.

va.c

Hlavní modul. Vypisuje nápovědu, překládá požadavky a volá archivační funkce.

4. Ovládání programu

Program se ovládá zadáním parametrů na příkazové řádce.

Obecná syntaxe:

`va <příkaz>[<volba>[<volba>...]] <archiv> [maska [maska ...]]`

Hranaté závorky označují volitelnou část. Není-li maska zadána, pracuje se se všemi soubory.

Příkazy:

`a` Do archivu se přidají soubory vyhovující maskám.
možné volby : `0-4efmqu`

`e` Z archivu se extrahují soubory vyhovující maskám. Soubory jsou extrahovány bez použití uložených cest.
možné volby : `mqty`

specifická syntaxe:
`va e[<volba>[<volba>...]] <archiv> [základní adresář/]`
`[maska [maska...]]`
Extrahované soubory jsou ukládány vzhledem k základnímu adresáři.

`d` Z archivu se zruší soubory vyhovující maskám. Pokud archiv neobsahuje po rušení žádné soubory, je smazán.
možné volby: `q`

`l` Vypíší se soubory v archivu, které vyhovují maskám.
možné volby: `l`

`t` Otestuje se neporušenost souborů vyhovujících maskám v archivu
možné volby: `q`

`x` Z archivu se extrahují soubory vyhovující maskám. Soubory jsou extrahovány za použití uložených cest.
možné volby: `mqty`

Syntaxe je stejná jako u příkazu "`e`".

`A` Do archivu se přidají z jiného archivu soubory vyhovující maskám.
možné volby: `fmqu`

specifická syntaxe:
`va A[<volba>[<volba>...]] <archiv> <přidávaný_archiv>`
`[maska [maska ...]]`
Soubory jsou přidávány do archivu z přidávaného archivu.

Volby:

`0..4` Určuje se použitý řád statistického modelování.

`e` Do archivu se neukládají cesty k souborům.

`f` Do archivu jsou přidány všechny soubory, které vyhovují maskám a jsou novější než verze v archivu.

`l` Výpisují se všechny informace o souborech uchovávaných v archivu

`m` Přidané soubory jsou z původního místa zrušeny.

`q` Nevypisují se nepodstatné informace jdoucí na standardní výstup. Volbu je možno zadat dvakrát a pak znamená potlačení výpisu i chybových hlášení na chybovém výstupu.

`t` Extrahovaným souborům není nastaven uložený, nýbrž aktuální čas.

`u` Do archivu se přidají všechny soubory, které vyhovují maskám a jsou novější než verze v archivu nebo v archivu vůbec nejsou.

`y` Není vyžadováno potvrzení akcí od uživatele a na všechny dotazy předpokládá kladnou odpověď.

5. Výsledky porovnání

Následující tabulka obsahuje výsledky porovnání účinnosti a rychlosti kompresních programů. Horní řádky tabulky obsahují velikost výsledného souboru v bytech, dolní řádky čas potřebný k provedení komprese v sekundách.

	gzip	**bzip2**	**rar**	**ha-1**	**ha-2**	**va-4**	**va-3**
(1)	382133 8.0	334100 7.5	375141 12.3	380631 15.9	291147 29.3	301666 33.7	296956 24.5
(2)	1230899 29.3	1101654 27.7	1208971 69.0	1205188 47.7	1084984 183.9	1116165 167.5	1115467 160.7
(3)	1404002 25.2	1066144 50.9	1195321 74.5	1396490 57.9	1174408 117.1	1164240 105.9	1226393 98.5
(4)			7480279 141		6687352 763	6347564 752	

Testovací data:

(1) celis.bmp, 256-barevný bitmapový obrázek, originální velikost 812086 B

(2) 1026-Rejuvin8.xm, soubor formátu extended module obsahující samplovaná hudební data, originální velikost 1915442 B

(3) archiv typu tar obsahující manuálové stránky Linuxových aplikací, originální velikost 4741120 B

(4) zdrojové texty jádra Linuxu, 2236 souborů, originální velikost 26264295 B

Testované programy:

gzip

gzip 1.2.4, Jean-loup Gailly, kompresní metoda: založena na LZ77,
volby programu: `-9`

bzip2

bzip2 0.1pl2, Julian Seward, kompesní metoda: třídění bloků,
volby programu: `-9`

rar

rar 2.01, Eugene Roshal, kompresní metoda: založena na LZ77,
volby prgramu: `-md1024`

ha

ha 0.999, Harri Hirvola

ha-1: kompresní metoda: založena na LZ77 a aritmetické kódování,
volby programu: `1`

ha-2: kompresní metoda: modelování s konečným kontextem a aritmetické kódování,
volby programu: `2`

va

va 0.3, Václav Honetschläger, kompresní metoda: modelování s konečným kontextem a aritmetické kódování
číslo znamená maximální použitý řád
paměťové omezení 6 MB

Měřeno velikostí výsledného souboru programy založené na modelování s konečným kontextem a aritmetickém kódování podle očekávání jasně zvítězily nad tradičními slovníkovými metodami. Cenou za to byla přibližně 2–6 krát delší doba potřebná k provedení komprese. Nepříliš často používaná kompresní metoda založená na třídění bloků překvapila výborným poměrem účinnost komprese/rychlost komprese. Můj program si v konkurenci nevedl špatně a nenechal se porazit konkurentem ze stejné třídy — ha.

Zajímavým faktem plynoucím z myšlenky konečného kontextového modelování je na první pohled neobvyklá závislost rychlosti komprese na zvoleném řádu statistického modelu. Rychlost komprese je nižší pro příliš malý nebo příliš velký řád. Vysvětlení není obtížné: při velkém řádu je nutno vytvářet a naplňovat velké množství kontextových tabulek; při malém řádu je sice tabulek málo, ale vyskytuje se v nich velké množství znaků, tedy se hodně času spotřebovává na vyhledání znaku v tabulce.

Pro ilustraci jevu uvádím tabulku výsledků testování mého programu na datech (3).

řád modelu	**4**	**3**	**2**	**1**
velikost výsledného souboru (B)	1164240	1226393	1627532	2211542
čas komprese (s)	105.9	98.5	120.1	159.5

Všechny programy byly testovány na PC s procesorem Pentium 100 MHz, 24 MB operační paměti a operačním systémem Linux Red Hat 5.0, jádro 2.0.32. Doby běhu programů byly měřeny standardním programem *time* a uvedené časy jsou doby strávené v uživatelské fázi.

6. Závěr

Program implementuje kompresní metodu založenou na statistickém modelování a aritmetickém kódování. Poskytuje lepší kompresní poměr než většina běžně používaných programů, ovšem rychlost komprese je v porovnání s ostatními programy nízká. Aby můj program mohl dosáhnout širšího použití, mělo by být omezeno velké množství paměti, kterou pro práci potřebuje. Rychlost komprese by se taktéž dala podle mého soudu o něco zvýšit. Oba problémy by mohlo vyřešit použití hashovací tabulky s omezenou velikostí a rušení nejdéle nepoužívaných nebo nejméně často používaných kontextů. Ovšem výzkumy vhodných datových struktur, algoritmů a hashovacích funkcí je téma téměř na diplomovou práci.

7. Použité zdroje

1) Mark Nelson, Jean-loup Gailly: The Data Compression Book, M&T Books, New York 1996

2) Harri Hirvola: HA, kompresní a archivační program, 1995

3) RNDr. Tomáš Dvořák: prezentace k přednášce Algoritmy komprese dat, 1997

4) P.G.Howard: The Design and Analysis of Efficient Loseless Data Compression Systems, PhD Thesis, Brown University 1993