Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Jan Chroustovský

Aritmetické kódování nad abecedou slabik

Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Jan Lánský

Studijní program: Informatika, Softwarové systémy

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne Jan Chroustovský

OBSAH

1. ABSTRAKT

Název práce: Aritmetické kódování nad abecedou slabik
Autor: Jan Chroustovský
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Jan Lánský
E-mail vedoucího: Jan.Lansky@mff.cuni.cz
Abstrakt: Slabiková komprese je nová metoda komprese textu, nabízející zajímavý kompromis mezi kompresí po znacích a po celých slovech, která by měla být výhodná především pro soubory střední velikosti. Slabiky jsou pro kódování vhodné, neboť přirozeně tvoří logické často se opakující jednotky, ze kterých se skládají celá slova. Nevýhodou slabik je, že datové struktury s nimi pracující se musí vyrovnat s teoreticky nekonečnou množinou slabik, která se může v textu vyskytnout. Tato diplomová práce se zaměřuje na využití aritmetického kódování pro slabikovou kompresi a použití rychlých a efektivních datových struktur pro statistický model a slabikový kontext. V práci je popsán nový aritmetický slabikový kompresní algoritmus AritSyll, který vychází z algoritmu HuffSyll. Značného vylepšení, oproti HuffSyll, doznal slabikový kontextový model, který je v AritSyll-u dlouhý až tři slabiky a lépe reprezentuje skladbu věty v přirozeném jazyce, a statistický model, který je inicializován statistickými informacemi lišící se podle zvoleného jazyka, pro lepší kompresi souborů malé velikosti. Kromě „klasických“ textových souborů s osmibitovou velikostí znaku, je pomocí AritSyll-u možné efektivně kódovat i textové soubory v kódování UTF-8. Implementovaný slabikový kompresní algoritmus je srovnáván s jinými znakovými i slovními kompresními algoritmy.
Klíčová slova: komprese textu, aritmetické kódování, binární indexovaný strom, slabikový statistický model

Title: Arithmetic coding with syllable alphabet
Author: Jan Chroustovský
Department: Department of Software Engineering
Supervisor: Mgr. Jan Lánský
Supervisor's e-mail address: Jan.Lansky@mff.cuni.cz
Abstract: A syllable-based compression is a new method of a text compression, offering interesting trade-off between a character-based compression and a word-based compression. Syllable-based compression should be best suited for middle sized files. Syllables are favorable, because they are forming logical units, which words are composed of. Syllables have disadvantage that data structures working with them must deal with infinite set of syllables. This thesis is focused to usage of arithmetic coding for syllable-based compression and usage of fast and effective data structures for a statistical model and a syllable context. In this thesis is described new arithmetic syllable-based compression algorithm AritSyll, which is based on the HuffSyll algorithm. A considerable improvement is new syllable context, which is up to three syllables long and which better represents sentences in a language. The next improvement is statistical model, which is initialized with statistical information based on selected language, for better compression of small sized files. With the AritSyll is possible to effectively encode text files in UTF-8 coding. Implemented syllable-based compression algorithm is compared with other character-based, word-based compression algorithms.
Keywords: text compression, arithmetic coding, binary indexed tree, syllable-based statistical model

2. ÚVOD

Tato diplomová práce navazuje na práci (Lánský, 2005), kde byla představena slabiková komprese jako nová metoda komprese textu. Slabiková komprese by měla být kompromisem mezi kompresí po znacích a po celých slovech, tudíž by měla být výhodná především pro soubory střední délky. Slabiky jsou pro kompresi výhodnější, než úseky znaků pevné délky, protože přirozeně tvoří logické často se opakující jednotky, ze kterých se skládají celá slova. Oproti slovům je počet unikátních slabik v jazyce výrazně menší, než počet unikátních slov a množiny slabik dvou různých dokumentů ve stejném jazyce jsou si mnohem podobnější, než množiny slov stejných dokumentů. Při slabikové kompresi (stejně jako slovní) hodně záleží na jazyku, ve kterém je napsán vstupní dokument. Pro slabikovou kompresi by měly být výhodné především jazyky s bohatým tvaroslovím.

V práci byl definován pojem jazyk a slabika a bylo ukázáno několik univerzálních algoritmů dělení slov na slabiky, jejichž funkčnost je založena na znalosti, která písmena jsou samohlásky a která souhlásky. Ačkoli slova nejsou těmito algoritmy rozdělena úplně gramaticky správně, na účinnost komprese to nemá velký vliv.

Součástí práce jsou i dva implementované slabikové kompresní algoritmy LZWL a HuffSyll. Algoritmus LZWL je slabikovou variací slovního algoritmu LZW. Algoritmus HuffSyll je slabikový kompresní algoritmus využívající adaptivního Huffmanova kódování inspirovaný algoritmem HuffWord. Implementované algoritmy měli ovšem pár nedostatků a vad na kráse. Například algoritmus HuffSyll nepoužíval žádný kontextový model – pouze jednoduše a nepříliš efektivně určoval očekávaný typ slabiky a případně kódoval jeho špatné určení – byl velice časově náročný a jeho kompresní výsledky byly nepříliš dobré.

V této práci bude představen nový slabikový kompresní algoritmus AritSyll, který nemá za cíl nic menšího, než zlepšení původního kompresního algoritmu HuffSyll – jak jeho funkčnosti, tak dosaženého kompresního poměru. Toho hodlá dosáhnout především pomocí aritmetického kódování, které je efektivnější statistickou metodou než Huffmanovo kódování, nového slabikového kontextového modelu, který je dlouhý až tři slabiky, a který by měl lépe odrážet skladbu vět a slov v přirozeném jazyce, a v neposlední řadě také pomocí rychlých a efektivních datových struktur (např. trie pro slovník častých slov/slabik a binární indexovaný strom pro statistický model).

Kompresní algoritmus AritSyll je také vylepšen o možnost komprimovat textové soubory v kódování UTF-8. I když ve své podstatě je každý kompresní algoritmus schopen komprimovat soubory v kódování UTF-8, textové kompresní algoritmy mají pozici mnohem těžší, protože musí v UTF-8 kódech správně rozeznávat písmena, aby mohly pracovat efektivně.

3. JAZYKY A SLABIKY

Vyčerpávající rozbor jazyků a slabik včetně jejich formálních definicí je již uveden v (Lánský, 2005). Uveďme v této kapitole alespoň jeho krátké shrnutí pro jemné uvedení do problému slabikové komprese.

3.1. JAZYKY

Jedním z kritérií rozdělení přirozených jazyků je, jestli mají jednoduché nebo bohaté tvarosloví. U jazyků s jednoduchým tvaroslovím (např. angličtina) lze od jednoho jazykového kmene odvodit jen velice málo slov. Oproti tomu u jazyků s bohatým tvaroslovím (např. čeština, němčina, ruština) lze od jednoho jazykového kmene odvodit mnoho slov. Například z jednoho jazykového kmene v češtině lze odvodit několik stovek různých slov, které jsou tvořeny pouze několika desítkami různých slabik. Navíc pro jazyky s bohatým tvaroslovím je přirozené dělení slov do slabik a slova jsou většinou víceslabičná. V (Lánský, 2005) je ukázáno, že slabiková komprese je výhodnější právě pro jazyky s bohatým tvaroslovím.

3.2. SLABIKY

Zjednodušená definice slabiky říká, že „slabika je posloupnost hlásek, která obsahuje právě jednu posloupnost samohlásek“. Ačkoli tato definice se liší od formální definice slabiky v Compact Oxford English Dictionary, pro potřebu slabikové komprese je zcela dostačující, neboť není třeba vždy rozdělit slova úplně správně.

Podle zjednodušené definice slabiky má například slovo *rostoucí* tři slabiky, neboť obsahuje tři posloupnosti samohlásek *o*, *ou* a í, což je i správný počet. Ale třeba slovo *neobletí* má podle zjednodušené definice pouze tři slabiky, protože obsahuje posloupnosti samohlásek *eo*, *e* a í. Ovšem správný počet jsou čtyři slabiky.

Jeden z důvodů, proč je dělení na slabiky výhodné, je, že počet unikátních slabik v jednom dokumentu je mnohem menší než počet unikátních slov. Například dle (Lánský, 2005) obsahuje dokument v českém jazyce (Karel Čapek: Hordubal) o velikosti 195kB 8 071 unikátních slov (z 33 135 slov) a pouze 3 187 unikátních slabik (z 61 259 slabik). Anglický překlad bible o velikosti 4MB obsahuje 13 455 unikátních slov (ze 767 857 slov) a pouze 5 604 unikátních slabik (z 1 073 882 slabik).

3.3. DĚLENÍ SLOV NA SLABIKY

Dělení slov na slabiky nemusí být v některých případech vůbec jednoduché a dokonce ani jednoznačné. Ke správnému rozdělení se často musí nejdříve určit kořen slova, neboť stejné posloupnosti písmen se často dělí rozdílně a na různý počet slabik.

Algoritmus přesného dělení slov na slabiky by byl tedy zbytečně náročný. Navíc, pro potřeby komprese stačí, aby takový algoritmus pouze vytvářel skupiny písmen, které se vyskytují často. Bude tedy stačit nějaký algoritmus, který aproximuje správné rozdělení slov na slabiky.

V (Lánský, 2005) jsou popsány čtyři typy univerzálních algoritmů dělení slov na slabiky: univerzální levý P_{UL}, univerzální pravý P_{UR}, univerzální středně-levý P_{UML} a univerzální středně-pravý P_{UMR}. Každý

algoritmus pracuje po slovech. Nepísmenná slova (číselná a speciální) se nijak nedělí. U písmenných slov se nejdříve určí, která písmena ve slově jsou souhlásky a která samohlásky. Následně se vyhledají všechny souvislé úseky samohlásek. Za slabiku je prohlášen každý takový úsek samohlásek a k těmto úsekům jsou rozděleny souhlásky. Způsob, jak jsou souhlásky přiřazeny k souhláskám, závisí na typu algoritmu:

- Algoritmus P_{UL} přidá všechny souhlásky mezi dvěma samohláskami k levé souhlásce
- Algoritmus P_{UR} přidá všechny souhlásky mezi dvěma samohláskami k pravé souhlásce
- Algoritmus P_{UML} přidá polovinu souhlásek mezi dvěma samohláskami k levé samohlásce a polovinu k pravé. Pokud je lichý počet souhlásek, pak prostřední souhlásku přidá k levé samohlásce
- Algoritmus P_{UMR} přidá polovinu souhlásek mezi dvěma samohláskami k levé samohlásce a polovinu k pravé. Pokud je lichý počet souhlásek, pak prostřední souhlásku přidá k pravé samohlásce

Pokud souhlásky neleží mezi dvěma samohláskami (na začátku a na konci slova), pak jsou připojeny k nejbližší samohlásce.

Algoritmy při určování souhlásek a samohlásek navíc využívají znalosti, v jakém jazyce je vstupní dokument. Například v českém jazyce může být písmeno l a r jak samohláskou, tak souhláskou. V angličtině je podobným případem písmeno y.

V zájmu zjednodušení (a zefektivnění) komprese byla omezena maximální délka slova. Písmenná a speciální slova mají maximální délku 10 a číselná slova délku 4. Pokud se v textu vyskytuje slovo, které je delší než 10 (resp. 4) znaků, pak se jednoduše rozdělí na několik slov o maximální délce 10 (resp. 4).

4. ARITMETICKÉ KÓDOVÁNÍ

Aritmetické kódování je zástupce bezztrátových statistických kompresních metod. Tyto metody pracují s pravděpodobností výskytu symbolů v kódované abecedě. Čím větší pravděpodobnost výskytu ve vstupní zprávě symbol má, tím menší kód mu tyto metody přiřadí. Kromě aritmetického kódování do této skupiny patří ještě například Huffmanovo kódování nebo Shannonovo kódování.

4.1. PRINCIP ARITMETICKÉHO KÓDOVÁNÍ

Na rozdíl od Huffmanova kódování nebo Shannonova kódování, kde se každému symbolu z abecedy přiřadí nějaký unikátní binární kód, v aritmetickém kódování se hledá takové reálné číslo z intervalu $\langle 0,1)$, které reprezentuje celou vstupní zprávu. Na začátku kódování se uvažuje celý interval $\langle 0,1)$. S každým dalším kódovaným symbolem se tento interval zúží na základě pravděpodobnosti výskytu právě kódovaného symbolu. Na konci kódování stačí vypsat libovolné číslo z konečného intervalu, které reprezentuje celou vstupní zprávu. Dekódování pracuje obdobně. Na začátku dekódování je k dispozici celé reálné číslo, které reprezentuje celou zakódovanou zprávu, a celý interval $\langle 0,1)$. Symbol se dekóduje tak, že se rozdělí interval v poměru pravděpodobností výskytu všech symbolů a zjistí se, do které části intervalu reálné číslo patří. K dekódování dalších symbolů je ještě třeba aktuálně dekódovaný symbol „odstranit“ z reálného čísla.

Mějme kódovou zprávu $\alpha = a_1 a_2 a_3 \dots a_n \ \forall i = 1..n; a_i \in \Sigma$, kde Σ je nějaká abeceda symbolů velikosti m. Předpokládejme, že známe pravděpodobnosti výskytu všech symbolů a_i a známe počet symbolů ve zprávě (při dekódování). Pak následující dva obrázky představují schéma algoritmů aritmetického kódování (viz Obrázek 1) a dekódování (viz Obrázek 2).

$encode(\alpha = a_1 a_2 a_3 \dots a_n)$

(1) $I \leftarrow \langle 0,1), i \leftarrow 1$
(2) **while** $i \leq n$ **do**
(3) rozděl I na m intervalů, jejichž poměr velikostí odpovídá pravděpodobnostem symbolů
(4) $I \leftarrow I_{a_i}$ interval určený symbolem a_i
(5) $i \leftarrow i + 1$
(6) **od**
(7) $output(x \in I)$

Obrázek 1: Aritmetické kódování

Při implementaci aritmetického kódování ovšem narazíme na spoustu technických problémů. Základním problémem je, že žádný počítač nemá přesná, natož nekonečná reálná čísla. Dalším problémem je, jak získat pravděpodobnosti výskytu symbolů a jak je předat dekodéru. A nakonec je ještě třeba vyřešit zastavení dekodéru, který se bez dodatečných informací (např. počet zakódovaných symbolů nebo zakódování speciálního symbolu EOF) nikdy nezastaví.

$decode(x)$

(1) $I \leftarrow \langle 0,1), i \leftarrow 1$
(2) **while** $i \leq n$ **do**
(3) rozděl I na m intervalů, jejichž poměr velikostí odpovídá pravděpodobnostem symbolů
(4) vyber interval I_{a_i} takový, že $x \in I_{a_i}$
(5) $output(a_i)$
(6) $x \leftarrow \frac{x-(I_{a_i})_L}{(I_{a_i})_H-(I_{a_i})_L}$, kde $(I_{a_i})_L$ je dolní a $(I_{a_i})_H$ horní část intervalu I_{a_i}
(7) $i \leftarrow i + 1$
(8) **od**

Obrázek 2: Aritmetické dekódování

4.2.IMPLEMENTACE ARITMETICKÉHO KÓDOVÁNÍ

Ve skutečnosti stačí pro aritmetické kódování obyčejná 16, 32 nebo 64 bitová celočíselná aritmetika. Reálná čísla se dají snadno nahradit celými čísly a i když ani celá čísla nejsou nekonečná, dá se pomocí nich a operace binárního posunu snadno nekonečné číslo simulovat, protože nikdy nebude potřeba více než pár bitů na začátku čísla. Stačí načíst nové bity na konec čísla a bitovým posunem je dostat na začátek čísla, kde se zpracují.

Celá modifikace algoritmu aritmetického kódování do celočíselné aritmetiky spočívá v několika jednoduchých tricích. Místo pravděpodobností symbolů se používají kumulované četnosti symbolů. Kumulovaná četnost f_i pro symbol a_i je suma četností s_j všech předcházejících symbolů

$$f_i = \sum_{j=1}^{i} s_j$$

S $f_0 = 0$ snadno odvodíme, že $s_i = f_i - f_{i-1}$. Pomocí kumulovaných četností lze interval I také rozdělit na podintervaly I_{a_i}, jejichž poměr velikostí odpovídá poměru jejich pravděpodobností. Navíc platí, že

$$(I_{a_i})_L = \frac{f_{i-1}}{f_n}, (I_{a_i})_H = \frac{f_i}{f_n}$$

Kde f_n je součet všech četností. K získání správného intervalu I_{a_i} pro symbol a_i tedy stačí pouze f_{i-1}, f_i a f_n.

4.2.1. ARITMETICKÉ KÓDOVÁNÍ

Označíme-li $low = f_{i-1}, high = f_i, total = f_n$, pak následující obrázek (viz Obrázek 3) představuje algoritmus aritmetického kódování podle (Moffat, Neal, & Witten, 1998) pro jeden symbol. Kodér používá dvě celočíselné b bitové proměnné L a R, které reprezentují dolní hranici aktuálního intervalu I a jeho délku. Na začátku kódování je $L = 0$ a může nabývat libovolné hodnoty z intervalu $\langle 0,2^b - 2^{b-2})$, $R = 2^{b-1}$ a $R \in (2^{b-2},2^b\rangle$. Zakódovanou zprávou je libovolné číslo z intervalu $\langle L,L + R)$.

```
encode(low, high, total)

(1) r ← R div total
(2) L ← L + r · total
(3) if high < total
(4)     R ← r · (high − low)
(5) else
(6)     R ← R − r · low
(7) while R ≤ 2^(b−2) do
(8)     encode_renormalize()
```

Obrázek 3: Implementace aritmetického kódování

Algoritmus aritmetického kódování i dekódování předpokládá, že $0 \leq low < high \leq total$, $total \leq 2^f$, kde f je počet bitů, které jsou použity k uložení kumulovaných četností, a $f \leq b - 2$. Celkový součet četností ve standardní 32 bitové aritmetice tedy může být až 2^{30}, než je nutná aktualizace statistického modelu (všechny četnosti se vydělí dvěma).

Cílem algoritmu (1 – 6) je zmenšit interval $\langle L, L + R)$ na jeho podinterval $\langle low/total, high/total)$, který reprezentuje nový kódovaný symbol. Výsledný interval je tedy $\langle L + R \cdot low/total, L + R \cdot high/total)$. Pokud by se ovšem výsledný interval stále zmenšoval, za chvíli by měl délku 1 a nebylo by možné zakódovat už žádný další symbol. Proto jsou v algoritmu řádky (7) a (8), které zajistí, že interval nikdy nebude menší než 2^{b-2}. Více o tomto postupu v Kapitole 4.2.3.

4.2.2. ARITMETICKÉ DEKÓDOVÁNÍ

Algoritmus aritmetického dekódování (viz Obrázek 4) používá stejné proměnné jako aritmetický kodér. Navíc obsahuje celočíselnou proměnnou V, která reprezentuje b bitové okénko do zakódované zprávy (přesněji je to rozdíl okénka a L).

```
decode_target(total)

(1) r ← R div total
(2) target ← min{total − 1, V div r}
(3) return target

decode(low, high, total)

(1) r ← R div total
(2) V ← V − r · low
(3) if high < total
(4)     R ← r · (high − low)
(5) else
(6)     R ← R − r · low
(7) while R ≤ 2^(b−2) do
(8)     decode_renormalize()
```

Obrázek 4: Implementace aritmetického dekódování

Aritmetické dekódování je rozděleno do dvou funkcí. První funkce $decode\_target$ nejprve nalezne z hodnot V a $total$ cílovou hodnotu (1 + 2), která je předána statistickému modelu. Ten musí najít

takový symbol s, že $low_s \leq target < high_s$. Pak už může být zavolána druhá funkce $decode$, která interval nalezeného symbolu odstraní z aktuálního intervalu.

4.2.3. RENORMALIZACE INTERVALU

Aby nedocházelo ke snížení efektivity komprese, musí být velikost intervalu R stále co největší. Dokonce, aby aritmetický kodér správně fungoval, musí být $R \geq total$. A protože $total \leq 2^f$ a $f \leq b-2$, musí platit nerovnost $2^{b-2} < R \leq 2^{b-1}$. Proto musí být velikost intervalu R pravidelně před každým krokem kódování a dekódování renormalizována. Následující dva obrázky představují schéma algoritmů renormalizace při kódování (viz Obrázek 5) a dekódování (viz Obrázek 6).

```
encode_renormalize()

(1)  if L + R ≤ 2^(b-1)
(2)     put_bit_plus_follow(0)          // vypíše bit 0 a follow_bits krát bit 1
(3)  else if 2^(b-1) ≤ L
(4)     put_bit_plus_follow(1)          // vypíše bit 1 a follow_bits krát bit 0
(5)     L ← L − 2^(b-1)
(6)  else
(7)     follow_bits ← follow_bits + 1
(8)     L ← L − 2^(b-2)
(9)  fi
(10) L ← 2 · L
(11) R ← 2 · R
```

Obrázek 5: Renormalizace intervalu při aritmetické kódování

```
decode_renormalize()

(1)  if L + R ≤ 2^(b-1)
(2)     // žádná akce
(3)  else if 2^(b-1) ≤ L
(4)     L ← L − 2^(b-1)
(5)     V ← V − 2^(b-1)
(6)  else
(7)     L ← L − 2^(b-2)
(8)     V ← V − 2^(b-2)
(9)  fi
(10) L ← 2 · L
(11) R ← 2 · R
(12) V ← 2 · V + read_next_bit()        // načte nový bit do V
```

Obrázek 6: Renormalizace intervalu při aritmetické dekódování

Pokud v algoritmu funkce $encode\_renormalize$ je $L \leq 2^{b-1}$ a $L + R \leq 2^{b-1}$ (1), pak obě čísla začínají nulovým bitem a může se tento bit vypsat (2). Podobným případem je, pokud $L \geq 2^{b-1}$ a $R < 2^{b-2}$ (3). Pak obě čísla (L a $L + R$) začínají jedničkovým bitem a opět je ho možné vypsat (4). Pokud nenastane ani jedna z těchto možností, ale přesto je $R \leq 2^{b-2}$ (6), pak je interval $\langle L, L + R)$ okolo čísla 2^{b-1}. V takovém případě není možné vypsat žádný bit, ale inkrementuje se interní proměnná $follow\_bits$ (7) a při dalším vypisování bitů (2 + 4) se společně s ním vypíše i $follow\_bits$

krát opačný bit. Výsledkem provedení celého algoritmu na Obrázku 5 je, že se velikost intervalu R zdvojnásobí.

Po každém dekódovaném symbolu musí být aktualizován interval stejně jako při kódování. Navíc ještě musí být aktualizována proměnná V tak, že se všechny bity ve V posunou o jeden bit doleva a na poslední místo se přidá nový bit (12).

4.3.STATISTICKÝ MODEL

Jak již bylo napsáno, k tomu, aby bylo možné použít libovolnou statistickou kompresní metodu, musíme znát pravděpodobnosti výskytu všech symbolů ve vstupní zprávě. Strukturu, která obsahuje pravděpodobnost p_a (nebo f_a) pro každý symbol a z abecedy Σ, nazveme statistický model. Aby mohl být statistický model použit k efektivní kompresi dat, musí splňovat dvě podmínky:

- Správně predikovat pravděpodobnosti symbolů v kódované zprávě
- Generovat pravděpodobnosti, které neodpovídají rovnoměrnému rozdělení

První podmínka je pro každé statistické kódování opravdu nezbytné, neboť myšlenka statistického kódování je založena právě na tom, že symbolům s velkou pravděpodobností přiřadí krátký kód (nebo velký interval) a symbolům s malou pravděpodobností přiřadí dlouhý kód (nebo malý interval). Pokud by model generoval pravděpodobnosti naopak, pak by místo komprese mohlo dojít k expanzi vstupní zprávy.

Pokud by pravděpodobnosti, které model generuje, odpovídali rovnoměrnému rozdělení, pak by nedocházelo vůbec k žádné kompresi – vstupní i komprimovaná zpráva by byli stejně dlouhé. Platí, že čím více se pravděpodobnosti symbolů v modelu budou lišit od rovnoměrného rozdělení, tím lepšího kompresního poměru je možné dosáhnout. Nesmí se ovšem zapomínat na první podmínku a pravděpodobnosti se musí generovat správně.

4.3.1. STATICKÉ A ADAPTIVNÍ MODELOVÁNÍ

Běžnou metodou, jak získat pravděpodobnosti symbolů ve vstupní zprávě, je projít před kódováním celou vstupní zprávu a shromáždit všechny potřebné pravděpodobnosti. Takto získaný statický statistický model je pro aktuální zprávu optimální a zakódovaná zpráva bude mít zaručeně nejmenší možnou délku. Nevýhodou je, že se takový model musí nějak předat dekodéru, aby ji mohl bezchybně dekódovat – tj. celý statistický model se musí připojit k zakódované zprávě a výsledná délka už zdaleka nemusí být optimální.

Druhou možností je generovat a aktualizovat pravděpodobnosti symbolů v modelu „za běhu“ algoritmu komprese i dekomprese. Oba algoritmy musí začínat s nějakým shodným statistickým modelem a po každém zakódovaném/dekódovaném symbolu tento model aktualizovat. Pokud kodér i dekodér aktualizují statistický model stejně, pak není třeba dekodéru předávat statistický model kodéru. Ačkoli na začátku má adaptivní statistický model nevýhodu, protože neobsahuje optimální pravděpodobnosti symbolů, po odečtení ceny předávání statistického modelu společně se zakódovanou zprávou, je výkonnost tohoto modelu většinou lepší než u statického.

4.3.2. MODELOVÁNÍ S KONEČNÝM KONTEXTEM

Metoda modelování s konečným kontextem vychází z myšlenky, že pravděpodobnost symbolů ve vstupní zprávě se může velice lišit podle toho, v jakém se nacházejí kontextu. Kontextem může být například předchozí symbol nebo skupina symbolů. Například Pravděpodobnost výskytu znaku '\n' v zdrojovém kódu jazyka C může být pouze $\frac{1}{40}$, ale pravděpodobnost výskytu znaku '\n' v kontextu znaku '}' může být až $\frac{1}{2}$.

Kontextové modely se označují řádem podle velikosti kontextu, se kterým pracují. Nejjednodušší by byl kontextový model řádu 0 – tedy takový model, který nemá vůbec žádný kontext. Kontextový model řádu 1, používá kontext jednoho symbolu, atd.

Je pochopitelné, že čím vyšší řád kontextový model má, tím větší budou jeho paměťové nároky. Pokud by byl kontextový model implementován jako obyčejné datové pole a používala by se standardní 256 znaková abeceda, pak by model řádu 0 potřeboval celkem $256 \cdot 8$ bytů, model řádu 1 celkem $256^2 \cdot 8$ bytů … a model řádu n celkem $256^n \cdot 8$ bytů. Paměťové nároky tedy v tomto příkladě rostou exponenciálně s řádem modelu. Existují pochopitelně i lepší datové struktury (například trie) s menšími paměťovými nároky. To ovšem nemění nic na tom, že všechny kontextové metody musí obezřetně sledovat velikost volné paměti a v případě potřeby se nějak vyrovnat s jejím nedostatkem.

5. DATOVÉ STRUKTURY

V následující kapitole rozebereme a popíšeme všechny zajímavější datové struktury, které byly použity a implementovány v AritSyll-u.

5.1. BINÁRNÍ INDEXOVANÝ STROM

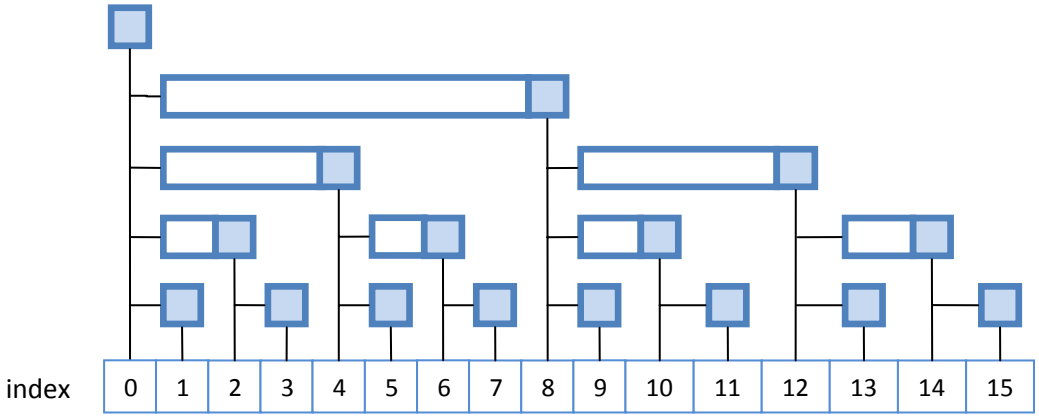
Binární indexovaný strom (nebo také Fenwickův strom) je speciální datová struktura pro ukládání kumulovaných četností (Fenwick, 1993).

Binární indexovaný strom ve skutečnosti není strom, ale pouze obyčejné datové pole. Pouze chytrým uložením kumulovaných četností simuluje činnost (i časovou složitost) binárního stromu. Místo klasických kumulovaných četností jsou v poli uložené částečné četnosti v pravidelném vzoru založeném na mocnině 2.

V Tabulce 1 je uveden příklad pro lepší ilustraci rozdělení kumulovaných četností na částečné četnosti pro pole velikosti 16. V prvním řádku je uveden index pole resp. prvku. Ve druhém řádku je obsah indexů pole. Například index 6 obsahuje součet četností prvků 5 – 6 a index 12 obsahuje součet četností prvků 9 – 12. Ve třetím řádku je četnost prvků a ve čtvrtém řádku jejich kumulovaná četnost. V posledním řádku jsou uvedeny skutečné hodnoty, které jsou uloženy v datovém poli.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Obsah	0	1	1..2	3	1..4	5	5..6	7	1..8	9	9..10	11	9..12	13	13..14	15
Četnost	0	2	0	1	1	1	0	4	4	0	1	0	1	2	3	0
Kumulovaná četnost	0	2	2	3	4	5	5	9	13	13	14	14	15	17	20	20
Uložené hodnoty	0	2	2	1	4	1	1	4	13	0	1	0	2	2	2	5

Tabulka 1: Příklad binárního indexovaného stromu



Obrázek 7: Rozdělení četností na částečné četnosti

Na Obrázku 7 je podrobně naznačena stromová struktura uložení hodnot v poli. Každý obdélník označuje interval hodnot, které jsou v poli uloženy na indexu pod tmavším (nejvíce vpravo) čtvercem. K získání jedné kumulované četnosti je třeba stromem projít od příslušného listu až ke kořeni stromu a po cestě sečíst všechny částečné četnosti.

K procházení stromem slouží dvě funkce $forward(s)$ a $backward(s)$. Pokud s je nějaký symbol v intervalu $1 \leq s \leq n$, kde n je velikost pole, pak funkce $backward(s)$ vrátí takové číslo, které se rovná rozdílu hodnot s a čísla tvořeného posledním jedničkovým bitem v s. Například číslo 14 má binární zápis 1110 tudíž $backward(14) = 1100_b = 12$; $backward(12) = 1000_b = 8$ a nakonec $backward(8) = 0$. Funkce $forward(s)$ vrátí takové číslo, které je součtem s a čísla tvořeného posledním jedničkovým bitem v s. Například číslo 6 má binární zápis 110, tudíž $forward(6) = 1000_b = 8; forward(8) = 10000_b = 16$. Pokud jsou záporná celá čísla ukládána ve dvojkovém doplňku, pak obě funkce mohou být jednoduše implementovány pomocí bitových operací. Funkce $backward(s)$ může být implementována buď jako „$s - (s \text{ AND } -s)$“ nebo „$s \text{ AND } (s-1)$“ a funkce $forward(s)$ jako $s + (s \text{ AND } -s)$, kde AND je bitový operátor „a“.

Funkce $backward(s)$ se používá pro získání kumulované četnosti symbolu s. Na následujícím obrázku (viz Obrázek 8) je ukázáno její jednoduché použití. Uvedená implementace ovšem není úplně ideální, protože pro zakódování jednoho symbolu aritmetickým kodérem jsou potřeba kumulované četnosti symbolů s a $s-1$. A je mnohem výhodnější počítat obě kumulované četnosti najednou, neboť je značná pravděpodobnost, že budou mít velkou část cesty stromem společnou.

```
fenwick_get_frequency(s)

(1) i ← s, f_s ← 0
(2) while i > 0 do
(3)     f_s ← f_s + F[i]          // F je pole s uloženými částečnými četnostmi
(4)     i ← backward(i)
(5) od
(6) return f_s                    // vrátí kumulovanou četnost f_s symbolu s
```

Obrázek 8: Získání kumulované četnosti symbolu

Funkce $forward(s)$ se používá při aktualizování četnosti symbolu s (viz Obrázek 9). Při aktualizaci četnosti se musí zvýšit četnost všech symbolů, které leží „nad“ symbolem s (mají bílý obdélník nad symbolem s na Obrázku 7).

```
fenwick_update(s, inc)            // inc je hodnota, o kterou má být inkrementována
                                  četnost symbolu s
(1) i ← s
(2) while i ≤ F.length do
(3)     F[i] ← F[i] + inc         // F je pole s uloženými částečnými četnostmi
(4)     i ← i + forward(i)
(5) od
```

Obrázek 9: Aktualizace kumulované četnosti symbolu

Poslední funkcí, která je nezbytná pro aritmetické dekódování, je funkce, která z kumulované četnosti $target$ nalezne takový symbol s, že $f_{s-1} < target \leq f_s$. Její schéma je uvedeno na Obrázku 10

```
fenwick_get_frequency(target)
(1)  s ← 0
(2)  mid ← 2^⌊log n⌋
(3)  while mid > 0 do
(4)     if F[s + mid] ≤ target          // F je pole s uloženými částečnými četnostmi
(5)        s ← s + mid
(6)        target ← target − F[s]
(7)     fi
(8)     mid ← mid/2
(9)  od
(10) return s
```

Obrázek 10: Nalezení symbolu z kumulované četnosti

5.1.1. ČASOVÁ A PAMĚŤOVÁ SLOŽITOST

Paměťová složitost binárního indexovaného stromu je $O(n)$, kde n je počet uložených prvků. Časová složitost získání a aktualizaci kumulované četnosti f_s symbolu s i nalezení symbolu s z kumulované četnosti $target$ je $O(\log n)$.

5.2. HAŠOVACÍ TABULKA

Hašovací tabulka (hashtable) je datová struktura pro efektivní ukládání a vyhledávání dat podle jejich klíče.

Každý prvek, který se do hašovací tabulky ukládá, musí mít přiřazen svůj klíč. Prvek je pak v tabulce uložen na místo, které je vypočítané hašovací funkcí z jeho klíče. Nechť U je univerzum všech klíčů a nechť T je hašovací tabulka velikosti m. Pak funkce

$$h(k): U \to \{0,1,2, \dots m-1\}$$

zobrazující univerzum U do indexů $0,1,2, \dots m-1$ je hašovací funkcí tabulky T. Pokud dva různé klíče k_1, k_2 mají stejnou hašovací hodnotu, pak nastává tzv. kolize.

5.2.1. ŘEŠENÍ KOLIZÍ

Ani se sebelepší hašovací funkce se zcela nevyhne kolizím. Dobrá hašovací funkce sice může počet kolizí minimalizovat, ale nikdy je neodstraní úplně, protože je $|U| \gg m$, pak alespoň dva klíče musí mít stejnou hašovací hodnotu. Existuje značné množství postupů, jak kolize řešit. Pro hašovací tabulku v AritSyll-u bylo vybráno a implementováno řešení kolizí pomocí řetězení. To znamená, že na místo v tabulce, kde dochází ke kolizi, se uloží lineární spojový seznam, který obsahuje všechny prvky, které mají stejnou hašovací hodnotu. Každá položka tabulky tedy buď neobsahuje nic, nebo jeden prvek, nebo lineární spojový seznam prvků, které mají stejnou hašovací hodnotu.

5.2.2. HAŠOVACÍ FUNKCE

Hašovací tabulka používá násobící hašovací funkci:

$$h(k) = \lfloor m\big((k \cdot A)\ mod\ 1\big) \rfloor$$

Kde A je nějaká vhodná konstanta v intervalu $(0,1)$ a m je velikost hašovací tabulky. Ačkoli velikost hašovací tabulky m může být libovolné číslo, je výhodné aby $m = 2^p$, pro libovolné celé číslo p. Potom lze násobící hašovací funkci implementovat velice elegantně. Nechť w je délka slova v počítači v bitech, k se vejde do jednoho slova a $A = \frac{s}{2^p}$, kde s je z intervalu $(0, 2^p)$. Vynásobíme-li nyní $k \cdot s$, $s = A \cdot 2^p$, bude výsledek dlouhý dvě slova a $h(k)$ bude p nejvýznamnějších bitů z druhého slova. Je zajímavé, že i když tato hašovací funkce funguje s libovolným $A \in (0,1)$, s některými pracuje lépe než s jinými. Například D. E. Knuth v (Knuth, 1998) doporučuje, aby $A \sim \frac{\sqrt{5}-1}{2}$, kterého bylo použito i v AritSyll-u.

5.2.3. OČEKÁVANÁ ČASOVÁ SLOŽITOST

Za předpokladů jednoduchého rovnoměrného hašování (každý klíč bude hašován se stejnou pravděpodobností do každé položky tabulky nezávisle na předchozím klíči) je v hašovací tabulce, kde jsou kolize řešené řetězením, očekávaná časová složitost operace $MEMBER$ $\Theta(1 + \alpha)$, kde $\alpha = \frac{n}{m}$ je faktor zaplnění tabulky (podíl počtu prvků v hašovací tabulce a velikosti tabulky). Pokud je počet prvků v hašovací tabulce úměrný velikosti tabulky, pak $\alpha = O(1)$ a tedy očekávaná časová složitost hledání prvku je konstantní (Cormen, Leiserson, Rivest, & Stein, 2001).

5.3. TRIE

Trie je datová struktura pro ukládání slovníku slov konečné délky.

Trie nad abecedou Σ velikosti k je konečný strom, jehož každý vnitřní vrchol má právě k synů, které jsou jednoznačně ohodnoceny prvky abecedy Σ. Každému vnitřnímu vrcholu trie odpovídá právě jedno slovo nad Σ s délkou shodnou jako je délka cesty od kořene trie k vrcholu. Kořenu trie odpovídá slovo λ (prázdné slovo) a pokud vrcholu v odpovídá slovo α, pak $v[a]$ synu v ohodnocenému písmenem a, odpovídá slovo αa.

Pole synů jsou implementovány jako hašovací tabulky s rostoucí velikostí (pokud je tabulka zaplněna z $XY\%$, pak se automaticky zvětší její velikost na dvojnásobek)

5.3.1. ČASOVÁ SLOŽITOST

Časová složitost operace $MEMBER$ i $INSERT$ je $O(l)$, kde l je maximální délka slova abecedy Σ (Koubek, 2005).

5.4. SLOVNÍK SLOV (SLABIK)

Ačkoli Trie je univerzální datová struktura pro ukládání slovníku slov, pro potřeby AritSyll-u nestačí, neboť každé slovo má přiřazený svůj unikátní index a v určité části dekomprese potřebujeme efektivně nalézt slovo podle jeho indexu. Protože slovník slov bude velice rozsáhlý, musí být všechny operace z hlediska časové složitosti co nejoptimálnější. Potřebujeme tedy operace:

- $INSERT$: Přidej nové slovo do slovníku

- $FIND$: Nalezni slovo ve slovníku
- GET: Nalezni slovo ve slovníku podle indexu

Řešením je použít dvě datové struktury:

- Trie, ve kterém jsou uloženy slova
- Vektor (datové pole, které si automaticky zvětší velikost na dvojnásobek, pokud je plné), ve kterém jsou uloženy pointery na slova podle jejich indexů. Indexy jsou slovům přiřazovány inkrementálně (první slovo má index 0, druhé slovo má index 1, …)

5.4.1. ČASOVÁ SLOŽITOST

Časová složitost operace $FIND$ je časovou složitostí operace $MEMBER$ v Trie – tedy $\mathrm{O}(l)$, kde l je maximální délka slova. Časová složitost operace GET je $\mathrm{O}(1)$ neboť na pozici i je pointer na slabiku s indexem i.

6. KONTEXT

6.1. TYP SLABIKY

Z definice slabiky a slova v (Lánský, 2005) vyplývá, že každá slabika je zároveň i slovem. Z definice dále vyplývá, že existuje pět druhů slov/slabik.

Název	Zkratka	Popis
Malá	L	Slovo/slabika složené z malých písmen
Velká	U	Slovo/slabika složené z velkých písmen
Smíšená	M	Slovo/slabika, které začíná velkým písmenem a zbytek písmen je malých
Číselná	N	Slovo/slabika složené z číslic
Speciální	O	Slovo/slabika složené z ostatních (ne-alfanumerických) znaků

Tabulka 2: Typ slov/slabik

První tři typy slov/slabik se navíc mohou označit jako písmenná a zbytek jako nepísmenná.

6.2. KONTEXT

V každém kroku komprese se musí zakódovat typ slova/slabiky a pak samotné slovo/slabiku. Jedna z možností, jak zakódovat typ slova/slabiky, je znát pravděpodobnosti výskytu všech typů slov/slabik v textu a z nich vytvořit statický model řádu 0. Kód slabiky je potom kód aritmetického kodéru používající tento model. Existuje ovšem lepší způsob zakódování typu slova/slabiky, protože věty a slova ve všech jazycích vykazují spoustu různých závislostí. Je proto mnohem výhodnější kódovat typ slov/slabik na základě předchozího kontextu.

Navržený kontextový model se snaží využít těchto jazykových závislostí:

- Pokud první slabika slova byla malá (resp. velká), pak následující slabika bude s velkou pravděpodobností opět malá (resp. velká)
- Pokud první slabika slova byla smíšená, pak následující slabika bude s velkou pravděpodobností malá
- Čím delší je sekvence slabik, tím menší je pravděpodobnost, že následující slabika bude písmenná
- Po písmenném slovu následuje speciální slovo a naopak
- Pokud speciální slovo/slabika obsahuje některý ze symbolů ukončující větu [.?!], pak následující slovo/slabika bude s největší pravděpodobností smíšené
- Pokud speciální slovo/slabika neobsahuje některý ze symbolů ukončující větu [.?!], pak následující slovo/slabika bude s největší pravděpodobností stejného typu jako poslední písmenné slovo/slabika
- Číselná slova/slabiky budou krátká a bude po nich následovat speciální slabika

Namísto vytváření složitého statistického modelu vyššího řádu postačí, pokud se bude „simulovat“ několika statistickými modely řádu 0, které se mění na základě aktuálního kontextu. Hlavním důvodem, proč byl zvolen tento přístup, je, že množina kontextů, které jsou opravdu zajímavé, je velice omezené množství (viz předchozích sedm bodů). Ve slabikové kompresi se používá kontext maximální délky 3, a pokud by byl použit statistický modelu řádu 3, pak by v něm bylo celkem $5^3 + 5^2 + 5$ různých kontextů. V AritSyll-u si ovšem vystačíme s pouhými třinácti kontexty pro slabiky a sedmi pro slova.

6.3. TYP KONTEXTU

Typy kontextu i funkce aktualizující kontext musí být různé pro slova i slabiky, neboť jen tak je možné jednoduše zachytit závislosti mezi slovy a slabikami. Navíc kontext pro slova má délku pouze 1 a kontext pro slabiky délku 1 – 3. V následujících dvou tabulkách (viz Tabulka 3 a Tabulka 4) jsou všechny typy kontextu vypsány.

Název	Zkratka	Popis
Malá_1	CT_L1	1 malá slabika
Malá_2	CT_L2	2 malé slabiky
Malá_3	CT_L3+	3 a více malých slabik
Velká_1	CT_U1	1 velká slabika
Velká_2	CT_U2	2 velké slabiky
Velká_3	CT_U3+	3 a více velkých slabik
Smíšená_1	CT_M1	1 smíšená slabika
Smíšená_2	CT_M2	1 smíšená slabika následovaná 1 malou slabikou
Smíšená_3	CT_M3+	1 smíšená slabika následovaná 2 a více malými slabikami
Číselná	CT_N	1 číselná slabika
Speciální s tečkou	CT_Ow.	1 speciální slabika, která **obsahuje** symbol ukončující větu [.?!] a následuje po písmenné slabice
Speciální bez tečky	CT_Oo.	1 speciální slabika, která **neobsahuje** symbol ukončující větu [.?!] a následuje po písmenné slabice
Speciální	CT_O	1 speciální slabika, která následuje po nepísmenné slabice

Tabulka 3: Kontext slabik

V Tabulce 5 jsou uvedeny pravděpodobnosti (v procentech) výskytů všech typů slabik, které následovali po určitém kontextu. V každém řádku je uveden aktuální kontext a pravděpodobnosti (v procentech) jakého typu bude následující slovo. Uvedená data byla získána z testovací množiny dokumentů T_{CZ} z dělení slov na slabiky MR. V tabulce jsou dobře vidět výše uvedené jazykové závislosti. Například že čím delší je slovo, tím je větší pravděpodobnost, že následující slabika bude speciální (CT_L1: pouze 40%; CT_L3+: 72%), nebo že po slabice se znakem ukončující větu následuje

Název	Zkratka	Popis
Malá_1	CT_L1	1 malé slovo
Velká_1	CT_U1	1 velké slovo
Smíšená_1	CT_M1	1 smíšené slovo
Číselná	CT_N	1 číselné slovo
Speciální s tečkou	CT_Ow.	1 speciální slovo, které **obsahuje** symbol ukončující větu [.?!] a následuje po písmenném slovu
Speciální bez tečky	CT_Oo.	1 speciální slovo, které **neobsahuje** symbol ukončující větu [.?!] a následuje po písmenném slovu
Speciální	CT_O	1 speciální slovo, které následuje po nepísmenném slovu

Tabulka 4: Kontext slov

Kontext\ Následující slabika	Speciální	Malá	Velká	Smíšená	Číselná
CT_L1	39.50	60.49	0.00	0.00	0.00
CT_L2	54.04	45.96	0.00	0.00	0.00
CT_L3+	71.84	28.16	0.00	0.00	0.00
CT_U1	41.46	0.44	57.86	0.00	0.25
CT_U2	62.29	0.04	37.67	0.00	0.00
CT_U3+	70.51	0.04	29.45	0.00	0.00
CT_M1	31.48	68.42	0.00	0.08	0.02
CT_M2	56.64	43.26	0.00	0.01	0.00
CT_M3+	74.79	25.21	0.00	0.00	0.00
CT_N	82.70	0.23	0.07	0.16	16.84
CT_Ow.	0.00	6.04	0.82	91.98	1.16
CT_Oo.	0.00	95.13	0.14	4.45	0.27
CT_O	0.06	25.20	43.19	24.20	7.34

Tabulka 5: Pravděpodobnosti (v procentech) typů slabik v kontextu

smíšená slabika (pravděpodobnost 92%).

Obdobná tabulka existuje pro všechny typy dělení slov na slabiky a jazyky. Každou tabulkou se inicializuje třináct statistických modelů řádu 0 – jeden pro každý typ kontextu. Aritmetický kodér potom používá statistický model aktuálního kontextu pro zakódování typu slova/slabiky.

7. SLOVNÍK ČASTÝCH SLABIK

V každém přirozeném jazyce existuje charakteristická množina slabik, které jsou používány mnohem více než ostatní. Bylo by velice výhodné tuto charakteristickou množinu umět získat a inicializovat s ní statistický model slabik. Počáteční inicializace statistického modelu je z pohledu dosaženého kompresního poměru velice výhodná, neboť již není třeba každou slabiku při prvním výskytu kódovat znak po znaku. Pokud ovšem bude získaná množina slabik příliš veliká, mohlo by se stát, že se spousta slabik vůbec nepoužije a zbytečně by se prodlužoval kód těch, které jsou používány často. Optimální cesta leží mezi oběma extrémními případy a je nutné najít postup, který k ní vede.

7.1. METODY VYTVÁŘENÍ SLOVNÍKU ČASTÝCH SLABIK

Slovníky slabik jsou rozdílné pro různé jazyky i pro různá dělení slov na slabiky, nicméně pravidla pro zařazování slabik do slovníku jsou pokaždé stejná. V (Lánský & Žemlička, 2006b) jsou uvedeny dvě pravidla, podle kterých lze zařazovat slabiky do slovníku:

- Kumulativní – slabika je zařazena do slovníku, pokud je její četnost dělená celkovou četností slabik větší než určité číslo
- Výskytové – slabika je zařazena do slovníku, pokud se vyskytuje (minimálně) v nějakém určitém počtu dokumentů

Slovníky slabik, které vznikly kumulativním pravidlem, budou označeny písmenem „C“ a číslem, které reprezentuje minimální podíl četností. Například slovník, kde má každá slabika četnost větší než $\frac{1}{65000}$ celkového počtu slabik, bude označen C65. Slovníky slabik vzniklé druhým pravidlem, budou označeny písmenem „A“ a číslem, které udává minimální procento dokumentů, ve kterém se slabika musí vyskytovat. Například slovník, kde se každá slabika vyskytuje v alespoň 20% dokumentů, bude označen A20. Pravidla lze pochopitelně i kombinovat a tak slovník, který bude splňovat obě výše uvedené podmínky, bude označen C65A20.

Při vytváření slovníku slabik je navíc velice důležité zvolit takovou množinu dokumentů, která je dostatečně reprezentativní pro daný jazyk. Pokud se vybere nesprávná množina dokumentů, pak se do slovníku mohou dostat slabiky, které se v daném jazyce vyskytují jen vzácně, a naopak slabiky, které se v jazyce vyskytují často, se nemusí do slovníku dostat vůbec. Výsledkem takového slovníku je pochopitelně špatný kompresní poměr, který je zvláště patrný při kompresi malých souborů.

7.2. SLOVNÍK ČASTÝCH SLABIK

Bylo testováno mnoho různých slovníků. Nejlepší kompresní poměr vycházel, pokud byl slovník inicializován slovníkem A5. Pokud byly soubory komprimovány se slovníkem A5, pak pro malé soubory do 10kB byl kompresní poměr lepší o 0.05 – 0.09 bpc, pro větší pouze o 0.01 – 0.02 bpc než u slovníku A20. Jenže velikost slovníku A5 byla mnohonásobně větší než u slovníku A20, což zbytečně prodlužovalo čas nutný ke kompresi a dekompresi souborů. Celkově nebylo zlepšení kompresního poměru tak veliké, aby se slovník A5 vyplatil. Oproti tomu slovník C65A20 byl o dost menší než slovník A20 a při kompresi souborů větších než 50kB, poskytoval prakticky stejné výsledky lišící se pouze o 0.01 – 0.03 bpc. Při kompresi malých souborů byl ovšem kompresní poměr horší o 0.5 – 0.2 bpc než u slovníku A20, což už je nezanedbatelné zhoršení. U ostatních slovníků, které byly

testovány, vycházeli výsledky přibližně stejně jako u slovníků A5, A20 nebo C65A20 a proto je zde nebudeme podrobně zmiňovat. Nakonec byl tedy vybrán slovník A20, neboť měl nejlepší poměr cena/výkon (resp. poměr velikost slovníku/kompresní poměr).

V Tabulce 6 jsou uvedeny výsledky slovníku A20 pro český i anglický jazyk z příslušné testovací množiny dokumentů T_{CZ} a T_{EN}. V prvním řádku každého jazyku jsou uvedeny počty různých slabik (slabiky, které se liší pouze velikostí písmen, se považují za shodné) ve slovníku a jejich procento z celkového počtu unikátních slabik v testovací množině dokumentů. V dalším řádku je uvedeno procento součtu četností slabik ve slovníku slabik z celkového počtu slabik v testovací množině dokumentů. Čím je toto číslo větší, tím menší je pravděpodobnost, že se při kompresi narazí na novou neznámou slabiku.

Jazyk\Alg. dělení slov		L	R	ML	MR	W
T_{CZ}	Počet slabik	5 977 (24.39%)	4 436 (21.97%)	4 803 (28.22%)	4 235 (26.74%)	8 727 (5.02%)
	Σ četností	96.46%	97.71%	97.95%	98.14%	75.40%
T_{EN}	Počet slabik	5 348 (13.55%)	5 498 (12.72%)	4 826 (16.49%)	4 736 (15.96%)	9 280 (6.00%)
	Σ četností	97.76%	97.76%	98.37%	98.34%	91.66%

Tabulka 6: Slovník častých slabik A20

Hypotézu, že v každém jazyce existuje charakteristická množina slabik, potvrzuje porovnání procenta počtu slabik a procenta součtu četností. Pouze cca 30% slabik v českém jazyce (resp. 15% slabik v anglickém jazyce) reprezentuje přes 97% všech slabik v dokumentech. U slov je rozdíl ještě větší a u anglického jazyka zvláště zajímavý.

Pro srovnání: český slovník slabik A5 obsahuje průměrně unikátních 8 500 slabik a 37000 slov a součet četností slabik/slov ve slovníku je průměrně 99.0%. Tedy za zvýšení součtu četností o 2% zaplatíme skoro dvojnásobnou (u slov dokonce čtyřnásobnou) velikostí slovníku.

7.3. SLOVNÍK ČASTÝCH ZNAKŮ

Jedním z požadavků zadání je, že program musí být schopen komprimovat soubory ve formátu UTF-8. UTF-8 je jedna z variant kódování Unicode (UTF-8 - Wikipedia) a tedy existuje celkem 2^{16} různých znaků, které musí být kodér schopný zakódovat (ve skutečnosti Unicode může mít až 2^{31} různých znaků, ale znaky od U+10000 jsou pouze grafické znaky nebo speciální znaky, které nemají s textem mnoho společného, a proto bylo rozhodnuto tyto znaky nekódovat). Je pochopitelné, že některá z písmen budou využívána poměrně často (například normální abeceda, čísla, znaky ‘ ’ a ‘\n’, atd.) a některá naopak vůbec. Proto bude k optimální kompresi souborů ve formátu UTF-8, kromě slovníku častých slabik, potřeba i slovník častých písmen.

K vytvoření slovníku častých písmen je možné použít úplně stejná pravidla jako k vytvoření slovníku častých slabik. Dá se předpokládat, že slovníky písmen budou mít obdobné vlastnosti jako slovníky slabik. A tedy pokud slovník slabik vytvořený nějakým pravidlem je dobrý (výsledná komprese nad tímto slovníkem má dobrý kompresní poměr), pak zřejmě bude dobrý i slovník písmen, který byl

vytvořený stejným pravidlem a ze stejné testovací množiny dokumentů. Proto i slovník častých písmen je vytvořený pravidlem A20.

7.3.1. UTF-8

Jak již bylo napsáno, UTF-8 je jedna z variant kódovaní znaků Unicode s proměnnou délkou. Výsledný kód znaku v UTF-8 muže mít jeden až čtyři byty. Původní specifikace UTF-8 dovolovala až šest bytů, ale v dokumentu RFC 3629 bylo rozhodnuto, že kódování nebude pokrývat všech 2^{31} znaků, ale pouze znaky U+000000 – U+10FFFF. Je samozřejmé, že znaky, které jsou často používané, mají kratší kód (jeden až tři byty) a znaky vzácné mají delší kód.

1) Jeden byte je třeba na zakódování 128 US-ASCII znaků
2) Dva byty jsou třeba k zakódování znaků s diakritikou a znaků z jiných abeced (česká, řecká, hebrejská, arabská, atd.)
3) Tři byty jsou třeba na zbytek znaků ze základní vícejazyčné stránky
4) Čtyři byty jsou třeba na znaky z ostatních stránek Unicode (ty jsou používány jen zřídka)

Pro anglickou abecedu tedy stačí pouze jediný byte. Pro českou abecedu jsou potřeba na znaky s diakritikou již dva byty.

V Tabulce 7 je detailně zaznamenán postup vytváření kódů znaků v UTF-8. Pokud kód znaku je pouze jednobytový, pak první bit má hodnotu 0. Vícebytové znaky začínají bitem 1 a podle počtu následujících bitů s hodnotou 1 se určí celkový počet bytů znaku.

Rozsah	Celkem kódů	UTF-8 hodnota
0x0000 – 0x007F	128 kódů	0zzzzzzz
0x0080 – 0x07FF	1 920 kódů	110yyyyy 10zzzzzz
0x0800 – 0xFFFF	63 488 kódů	1110xxxx 10yyyyyy 10zzzzzz
0x10000 – 0x10FFFF	1 048 576 kódů	11110www 10xxxxxx 10yyyyyy 10zzzzzz

Tabulka 7: Kódy UTF-8

Ačkoli v UTF-8 nezáleží na pořadí bytů (big-endian vs. little-endian), neboť se zpracovávají po bytech, soubory ve Windows přesto používají sekvenci tří bytů 0xEF 0xBB 0xBF, která je připojena na začátek každého souboru. Tato značka pouze identifikuje soubor jako soubor s kódováním UTF-8.

8. ARITSYLL

V předchozích kapitolách jsme se seznámili se slabikami, algoritmy dělení slov na slabiky, aritmetickým kódováním, slabikovým a slovním kontextem a slovníkem častých slabik. Nyní nezbývá nic jiného, než dát všechny tyto věci dohromady a představit nový kompresní algoritmus AritSyll.

AritSyll je bezztrátový kompresní algoritmus postavený na aritmetickém kódování, který je navržený pro kompresi textových souborů. Základním elementem, na který je dokument dělen a který je kódován, je slabika. Navíc je ještě možné kódovat po slovech, neboť slabika speciálním případem slova. Ačkoli algoritmus AritSyll-u je navržen pro kompresi textových souborů, je s ním technicky možné „komprimovat“ i binární soubory (i když doporučeníhodné to rozhodně není, neboť je značné riziko, že by mohlo dojít spíše k expanzi souboru).

AritSyll je na začátku kódování i dekódování inicializován všemožnými statistickými informacemi, které se liší podle zvoleného jazyka, typu dělení slov na slabiky a kódování vstupního souboru. Tyto informace mají za cíl především zlepšit kompresi souborů malé velikosti.

Kromě „klasických“ textových souborů s osmibitovou velikostí znaku, je pomocí AritSyll-u možné efektivně kódovat i textové soubory v kódování UTF-8.

8.1. STATISTICKÝ MODEL

Statistický model AritSyll-u musí obsahovat pravděpodobnosti pro všechny elementy, které bude třeba kódovat. Určitě tedy bude obsahovat statistický model pro kontext resp. typ slabiky (viz Kapitola 6) a statistický model slabik (viz Kapitola 7). Jaké další statistické modely v něm budou obsažené, závisí na tom, jak budou kódované nové slabiky.

8.1.1. KÓDOVÁNÍ NOVÉ SLABIKY

I když je statistický model písmenných slabik inicializován slovníkem častých slabik, může se stát, že v průběhu kódování narazíme na novou neznámou slabiku. Podle výsledků z vytváření slovníku častých slabik, by se mělo jednat přibližně o 2% písmenných slabik (viz Tabulka 6). Proto musí existovat postup, jak zakódovat novou slabiku.

Při kódování nové slabiky není třeba kódovat její typ, neboť ten už je zakódován s použitím aktuálního kontextu. Kód nové slabiky se tedy může skládat z kódu délky slabiky a kódů jednotlivých znaků slabiky. Další možnost je vynechat kód délky slabiky a za kód posledního znaku přidat speciální znak ukončující slovo. Tato možnost není úplně vhodná z důvodů malé délky slabik, protože vážený průměr délek všech slabik se pohybuje mezi 2 – 3 znaky. Poslední možností je najít ve statistickém modelu slabik takovou slabiku, která je maximálním prefixem nové slabiky, a složit kód nové slabiky z kódu vyhledané slabiky, kódu rozdílu délek nové a vyhledané slabiky a kódů zbývajících znaků slabiky. Tato možnost by mohla být velice zajímavá pro jazyky s bohatým tvaroslovím (například čeština), neboť nové písmenné slabiky velice často mají nějaký svůj prefix ve statistickém modelu. Nicméně i od této možnosti bylo nakonec upuštěno, neboť prefix nového slova nemusí ve slovníku vůbec existovat a muselo by se zavádět nové slovo λ (prázdné slovo) a pro jazyky s jednoduchým tvaroslovím není tento postup vůbec vhodný.

Kódování nové slabiky musí být vždy uvedeno zakódováním speciálního symbolu ESC, čímž se dekodéru dá najevo, že následující kódy mají jiný význam, než doposud. Tento speciální symbol musí být obsažen ve statistickém modelu slabik a je zdrojem pravděpodobně nikdy nekončící diskuze o tom, jakou by měl mít pravděpodobnost/četnost (tzv. zero-frequency problem). V této práci je použit přístup „AX“ z (Moffat, Neal, & Witten, 1998), který aproximuje počet symbolů s nulovou četností počtem symbolů s četností jedna. Pokud t_1 symbolů má četnost jedna, symboly a_i mají četnost s_i a $t = \sum_i s_i$ je celkový součet četností, pak

$$p_{ESC} = \frac{t_1 + 1}{t + t_1 + 1}$$

$$p_i = \frac{s_i}{t + t_1 + 1}$$

Výhoda tohoto přístupu spočívá v tom, že není třeba nijak speciálně reagovat na situaci, kdy neexistuje žádný symbol s četností jedna.

8.1.2. KONTEXT

Jak již bylo napsáno v Kapitole 6.2, při kódování typu slabiky se používá statistický model aktuálního kontextu. Existuje tedy třináct dynamických modelů (jeden pro každý typ kontextu), které jsou inicializovány pravděpodobnostmi získanými z testovací množiny dokumentů (viz Tabulka 5).

Počáteční součet četností je 5 000 a aktualizace statistického modelu (snížení všech četností na polovinu) se provádí, pokud součet četností přesáhne 2^{27}. Původně bylo plánováno udělat tyto statistické modely statické, neboť v běžném textu by se získané pravděpodobnosti neměli měnit. Ale výsledky dynamického modelu byly při kompresi běžného textu úplně stejné jako statického modelu a při kompresi netradičních souborů (například datové, XML nebo HTML soubory) výrazně lepší.

8.1.3. SLABIKY

Ke kódování slabik se používají dynamické statistické modely. Každý typ slabiky má vlastní statistický model. Pouze statistický model malých slabik je inicializován slovníkem častých slabik, ostatní statistické modely jsou na začátku prázdné. Byla testována i inicializace statistického modelu velkých, smíšených a speciálních slabik, ale kompresní poměr vždy vyšel výrazně horší než bez inicializace.

Počáteční součet četností je nastaven na 40 000 a aktualizace statistického modelu se provádí, pouze pokud součet četností přesáhne 2^{27}. Tyto hodnoty nejsou náhodné, neboť bylo experimentálně zjištěno, že jsou pro kódování optimální. Počáteční četnost ESC symbolu je jedna a zvyšuje/snižuje se s novými slabikami, které přidáváme do slovníku (viz Kapitola 8.1.1). Ačkoli je známé, že cca 2% kódovaných slabik budou nové slabiky, zvýšení počáteční četnosti ESC symbolu na 2% (nebo jiné číslo) celkové četnosti, zhoršuje kompresní poměr.

8.1.4. DÉLKY SLABIK

Pro statistický model délek slabik stačí obyčejný statický model (resp. jeden statický model pro každý typ slabik), který je inicializován hodnotami, které byly získány při vytváření slovníku častých slabik.

Maximální délka slabiky je deset znaků, maximální délka číselné slabiky čtyři znaky. Počáteční součet četností je 10 000 a aktualizace četností se ve statickém modelu pochopitelně neprovádí.

8.1.5. ZNAKY

Poslední částí je statistický model písmen. Ten lze ovšem ještě rozdělit na dvě části – statický model znaků a dynamický model znaků. Dynamický model je inicializován slovníkem častých znaků (viz Kapitola 7.3) včetně pravděpodobností. Statický model obsahuje všechny zbylé znaky, které nejsou v dynamickém modelu, a všem je přiřazena stejná pravděpodobnost. Při kódování znaku se nejprve zjistí, zda je znak obsažen v dynamickém modelu. Pokud tam není, pak se jeho pravděpodobnost vezme ze statického modelu (kde je určitě obsažen) a znak se přidá do dynamického modelu a odebere ze statického.

Toto rozšíření bylo nutné z důvodů podpory kódování UTF-8, neboť UTF-8 obsahuje celkem 2^{16} různých znaků, z nichž je většina využívána jen velice málo nebo dokonce vůbec. Bylo by proto opravdu hloupé, mít všech 2^{16} znaků v jednom statistickém modelu. Pak, i kdyby byl takový model inicializován slovníkem častých znaků, by délky kódů často používaných znaků byly delší, než při použití dvou statistických modelů. Za použití dvou statistických modelů se sice zaplatí nutností mít v prvním (dynamickém) statistickém modelu ESC symbol pro přepnutí do druhého (statického) statistického modelu, ale ten by měl být využíván jen velice zřídka a celkově by neměl příliš zhoršit kompresi znaků z prvního modelu (četnost ESC symbolu je opět rovná počtu znaků s četností jedna).

Tak jako pro kontexty a slabiky i pro každý typ znaků existuje vlastní statický i dynamický model. Počáteční součet četností v písmenných statistických modelech (malá + velká písmena) je 10 000. Počáteční součet v modelu jiných znaků je z důvodů rychlé adaptace na aktuální soubor pouze 2 000. Statistický model čísel je inicializován čísly 0 – 9 se stejnou pravděpodobností. Aktualizace četností se provádí, pouze pokud součet četností přesáhne 2^{27}.

Při kódování znaků se plně využívá typu slabiky. Například pokud typ slabiky je velká, pak všechny znaky jsou velká písmena a definičním oborem kódovací funkce jsou pouze velká písmena. Z tohoto důvodu není třeba kódovat typ znaku, i když existuje více typů znaků a jejich statistických modelů.

8.1.6. ALGORITMUS KÓDOVÁNÍ A DEKÓDOVÁNÍ NOVÉ SLABIKY

Se znalostí postupu kódování nové slabiky a celého statistického modelu je nyní možné podrobně popsat schéma algoritmu kódování nové slabiky, které je na následujícím obrázku (viz Obrázek 11).

Nejdříve se aritmetickým kodérem se statistickým modelem slabik zakóduje speciální symbol ESC (2), čímž se odliší normálně zakódovaná slabika a nová slabika. Potom už se může zakódovat nová slabika podle postupu z Kapitoly 8.1.1 – tedy zakóduje se délka slabiky (3) a všechny znaky slabiky (5).

Pro každý znak a_i ze slova S se musí nejdříve zjistit jeho typ (6). Na řádku (7) se pak zjistí, zda slovník znaků obsahuje znak a_i. Pokud ano, pak stačí zakódovat znak a_i aritmetickým kodérem s dynamickým modelem písmen (8) a aktualizovat jeho četnost (9). V opačném případě se opět musí zakódovat symbol ESC (11). Tentokráte je to ovšem ESC symbol z dynamického modelu znaků. Pak už je možné zakódovat znak a_i se statickým modelem znaků (12), přidat ho do slovníku písmen (13) a do dynamického modelu znaků (14) a odstranit ze statického modelu znaků (15).

```
encode_new_syllable(S = a_1 a_2 ... a_n)      // S je nová slabika se znaky a_1 a_2 ... a_n
(1)  type ← S.type
(2)  encode(Syll_type, ESC)                   // zakóduje ESC s modelem slabik
(3)  encode(Len_type, n)                      // zakóduje délku slova s modelem délek slabik
(4)  i ← 1
(5)  while i ≤ n do
(6)     stype ← a_i.type                      // stype je typ znaku a_i
(7)     if dictionary_find(a_i)               // pokusí se nalézt znak a_i ve slovníku znaků
(8)        encode(DChar_stype, a_i)           // zakóduje znak a_i s dynamickým modelem znaků
(9)        update(DChar_stype, a_i)           // aktualizuje četnost a_i v dynamickém modelu znaků
(10)    else
(11)       encode(DChar_stype, ESC)           // zakóduje ESC s dynamickým modelem znaků
(12)       encode(SChar_stype, a_i)           // zakóduje znak a_i se statickým modelem znaků
(13)       dictionary_add(a_i)                // přidá znak a_i do slovníku znaků
(14)       install(DChar_stype, a_i)          // přidá znak a_i do dynamického modelu znaků
(15)       remove(SChar_stype, a_i)           // odstraní znak a_i ze statického modelu znaků
(16)    fi
(17)    i ← i + 1
(18)od
```

Obrázek 11: Kódování nové slabiky

Algoritmus dekódování nové slabiky bude úplně stejný jako algoritmus kódování, až na to, že se všude místo zakódování elementu, element dekóduje. Tak tomu pochopitelně musí být, protože jinak by dekodér dekódoval úplně jiné věci, než před tím kodér zakódoval.

8.2. KODÉR

V této kapitole se podrobně popíše schéma algoritmu AritSyll kodéru (viz Obrázek 12).

Nejdříve musí být inicializovány všechny datové struktury (1). Kodér ke své práci potřebuje statistický model (viz Kapitola 8.1) a slovník častých slabik a znaků (viz Kapitola 7), které jsou inicializovány hodnotami pro daný jazyk, dělení slabik a kódování znaků (Windows-1250 nebo UTF-8). Všechny dílčí statistické modely (statistický model slabik, délek slabik, znaků a kontextu) jsou implementovány pomocí binárního indexovaného stromu (viz Kapitola 5.1). Slovník slabik je implementován jako datová struktura popsaná v Kapitole 5.4 a slovník písmen jako bitové pole. Dále je nutné inicializovat parser, který čte vstupní soubor po slabikách a aritmetický kodér.

V každém kroku while-cyklu kodéru se nejprve parserem načte nová slabika (3), kde se i určí její typ. Typ slabiky je následně zakódován aritmetickým kodérem se statistickým modelem aktuálního kontextu (5). Kontext je aktualizován až po zakódování slabiky S (16).

Na řádku (7) se zjistí, zda slovník slabik obsahuje nově načtenou slabiku S. Pokud ano, pak jí slovník slabik přiřadí unikátní index (kladné číslo větší než 0), pod kterým je ve slovníku uložená. Nyní je možné tuto slabiku (resp. její index) zakódovat aritmetickým kodérem se statistickým modelem slabik (8) a aktualizovat její četnost v modelu slabik (9) a četnost jejích znaků v dynamickém modelu znaků (10). Znaky by měli být v dynamickém modelu znaků přítomny, protože slovník častých písmen byl vytvořen stejným postupem, jako slovník častých slabik. Pokud slovník slabiku S neobsahuje, musí se

```
aritsyll_encode_file()

(1)  init_encoder()
(2)  while není zakódovaný celý soubor do
(3)     S ← read_syllable()
(4)     type ← S.type
(5)     encode(Ctx, type)              // zakóduje typ slabiky S modelem kontextu
(6)     index ← dictionary_find(S)     // pokusí se nalézt slabiku S ve slovníku slabik
(7)     if index > 0
(8)        encode(Syll_type, index)    // zakóduje index slabiky S s modelem slabik
(9)        update(Syll_type, S)        // aktualizuje četnost slabiky S v modelu slabik
(10)       update(DChar, S)            // aktualizuje četnosti písmen slabiky S v modelu
(11)    else                           písmen
(12)       encode_new_syllable(S)      // zakóduje S jako novou slabiku
(13)       dictionary_add(S)           // přidá slabiku S do slovníku slabik
(14)       install(Syll_type, S)       // přidá slabiku S do modelu slabik
(15)    fi
(16)    update(Ctx, S)                 // aktualizuje kontext
(17)od
(18)encode(Ctx, type)                  // zakóduje poslední typ slova S s modelem slabik
(19)encode(Syll_type, EOF)             // zakóduje znak EOF s modelem slabik
```

Obrázek 12: AritSyll kodér

zakódovat nová slabika podle algoritmu na Obrázku 11 (12). Následně se slabika musí přidat do slovníku slabik (13), kde jí je přiřazen nový unikátní index, a do statistického modelu slabik (14).

Na závěr kódování je ještě nutné zakódovat speciální symbol EOF (18 + 19), aby dekodér poznal konec soubor.

8.3. DEKODÉR

Schéma algoritmu dekodéru AritSyll-u (viz Obrázek 13) je v podstatě stejné jako schéma algoritmu kodéru.

Dekodér je na řádku (1) inicializován úplně stejně jako kodér. Tedy inicializují se statistické modely a slovníky slabik a písmen hodnotami pro daný jazyk, dělení slabik a kódování znaků, a aritmetický dekodér. Parser není pochopitelně v dekodéru potřeba.

V každém kroku while-cyklu dekodéru se nejprve dekóduje aritmetickým dekodérem se statickým modelem aktuálního kontextu typ slabiky (3) a s modelem slabik index slabiky (4).

V kroku (5) se testuje, zda dekódovaný index náhodou není indexem symbolu EOF. Pokud ano, pak se ukončí while-cyklus dekodéru (6). V opačném případě se musí otestovat, zda nebyl dekódován symbol ESC, který značí novou neznámou slabiku (7). Pak se musí dekódovat nová slabika podle algoritmu na Obrázku 11. (8) a dekódované slabika musí přidat do slovníku slabik (9), kde je jí přiřazen unikátní index, a do statistického modelu slabik (10). Pokud nebyl dekódován symbol EOF ani symbol ESC pak se ve slovníku slabik vyhledá slabika podle dekódovaného indexu (12) a aktualizuje se její četnost ve statistickém modelu slabik (13) a četnost jejích znaků v dynamickém modelu znaků (14).

```
aritsyll_decode_file()

(1)  init_decoder()
(2)  do
(3)     type ← decode(Ctx)                    // dekóduje typ slabiky s modelem kontextu
(4)     index ← decode(Syll_type)             // dekóduje index slabiky s modelem slabik
(5)     if index = EOF
(6)        break
(7)     else if index = ESC
(8)        S ← decode_new_syllable(type)      // dekóduje S jako novou slabiku
(9)        dictionary_add(S)                  // přidá slabiku S do slovníku slabik
(10)       install(Syll_type, S)              // přidá slabiku S do modelu slabik
(11)    else
(12)       S ← dictionary_get(index)          // získá slabiku s indexem indexze slovníku slabik
(13)       update(Syll_type, S)               // aktualizuje četnost slabiky S v modelu slabik
(14)       update(DChar_type, S)              // aktualizuje četnosti písmen slabiky S v modelu
                                              písmen
(15)    fi
(16)    write(S)
(17)    update(Ctx, S)                        // aktualizuje kontext
(18) while true
```

Obrázek 13: AritSyll dekodér

Nyní už zbývá jen dekódovanou slabiku S zapsat do souboru (16) a aktualizovat kontext (17).

9. VÝSLEDKY

V této kapitole jsou uvedeny výsledky všech porovnávaných algoritmů. Algoritmy jsou porovnávány z hlediska úspěšnosti komprese, která je měřena v průměrném počtu bitů nutných k zakódování jednoho bytu (bpc: bits-per-character), a z hlediska časové a prostorové náročnosti.

Algoritmus AritSyll je zastoupen hned v pěti provedeních, lišících se pouze metodou dělení slov na slabiky. Algoritmy jsou označeny zkratkou AS s indexem označující zvolenou metodu dělení (AS_L pro dělení slov na slabiky P_{UL}, AS_R pro dělení P_{UR}, AS_{ML} pro algoritmus P_{UML}, AS_{MR} pro algoritmus P_{UMR} a AS_W pro slovní verzi). S nimi jsou porovnávány další kompresní algoritmy: slovní (HS_W) a slabikový algoritmus HuffSyll s dělením slov na slabiky P_{UML} (HS_{ML}) a se slovníkem častých slabik A20 z (Lánský, 2005), vylepšená verze aritmetického kódování uvedená v (Moffat, Neal, & Witten, 1998) ve své písmenné (ACM_{CH}) a slovní verzi (ACM_W) a nakonec ještě velice populární kompresní algoritmus založený na Burrows-Wheelerově transformaci a Huffmanově kódování bzip2 z (Seward).

9.1. TESTOVACÍ A MĚŘÍCÍ MNOŽINY DOKUMENTŮ

Kompresní algoritmus AritSyll je při svém spuštění inicializován spoustou statistických informací (četností slabik, četností písmen, délek slabik, kontext) závisící na zvoleném jazyce, algoritmu dělení slabik a kódování souboru. Musí proto existovat pro daný jazyk dostatečně reprezentativní množiny dokumentů, z kterých jsou tyto informace získány.

V testovací množině českých dokumentů T_{CZ} je 69 souborů o celkové velikosti 15MB. Nachází se zde především próza, ale i několik básnických děl a historických textů. Dokumenty pochází z mnoha zdrojů, ale většina pochází ze serveru eKnihy.

V testovací množině anglických dokumentů T_{EN} je 333 souborů o celkové velikosti 143MB. Opět je většina textů próza, ale najde se i několik historických, filozofických a odborných textů a kompletní anglický překlad bible. Text bible pochází z Canterburského Corpusu a ostatní texty jsou z Projektu Gutenberg. Všechny texty z Projektu Gutenberg mají odstraněné informace o projektu a o podmínkách šíření textu, aby nezkreslovaly statistické informace, které jsou z této množiny získávány.

Kromě testovacích množin existují ještě měřící množiny dokumentů, na kterých je následně měřena úspěšnost výsledného kompresního algoritmu a porovnávána s jinými kompresními algoritmy. Testovací a měřící množiny dokumentů nejsou úplně disjunktní. Měřící množiny vždy obsahují nějaké dokumenty z testovacích množin.

V měřící množině českých dokumentů M_{CZ} je 7 000 souborů o celkové velikosti 25MB. Kromě celé testovací množiny T_{CZ} se zde nachází novinové články získané náhodně z různých internetových stránek. Velikost jednoho článku není větší než 50kB.

V měřící množině anglických dokumentů M_{EN} je 7000 souborů o celkové velikosti 181MB. Kromě několika (300) textů z Projektu Gutenberg se zde nachází i veliké množství právnických textů.

Pro každou množinu (T_{CZ}, T_{EN}, M_{CZ} a M_{EN}) ještě existují úplně stejné množiny T-U_{CZ}, T-U_{EN}, M-U_{CZ} a M-U_{EN} lišící se pouze tím, že soubory v nich obsažené jsou v kódování UTF-8 se standardní byte-order značkou 0xEF 0xBB 0xBF (viz Kapitola 7.3.1). Celková velikost českých množin dokumentů T-U_{CZ} a M-

U_{CZ} je asi o 10% větší, než celková velikost jejich předloh T_{CZ} a M_{CZ}. U anglických množin dokumentů zůstala celková velikost prakticky stejná.

9.2. ČASOVÁ A PAMĚŤOVÁ NÁROČNOST

Výkonnostní testy jsou měřeny na počítači AMD Dual Core Athlon A64 X2 3800+, 1GB DDR2 667MHz RAM.

Komprese souboru o velikosti 4MB (anglický text bible) pomocí AritSyll-u trvá necelé 3 sekundy a dekomprese 2 sekundy. Komprese stejného souboru pomocí HuffSyll-u trvá na stejném PC 9 sekund a dekomprese 8 sekund. Bzip2 zvládne tento soubor zkomprimovat za 2 sekundy a dekomprimovat za pouhou 1 sekundu.

AritSyll je paměťově poměrně náročný, protože je inicializován spoustou statistických informací (jen slovník malých slabik je trie obsahující cca 5 000 slabik). Pro zakódování výše uvedeného 4MB velkého souboru potřebuje přibližně 3,5MB paměti a pro dekódování 3MB paměti. HuffSyll potřebuje pro zakódování stejného souboru celkem 2MB paměti a pro dekódování 1,5MB paměti. Program bzip2 je z pohledu použité paměti nejvíce náročný, neboť pro zakódování stejného 4MB souboru potřebuje celých 7MB paměti a pro dekódování 4MB.

Oproti HuffSyll-u se tedy skoro dvojnásobně zvýšila paměťová náročnost, ale zase se trojnásobně zrychlil kompresní algoritmus. Zvýšení paměťových nároků je zřejmě důsledkem vylepšení kompresního algoritmu o kódování souborů v UTF-8, neboť slabiky/slova (resp. jejich znaky) musí být reprezentovány 32 bitovými čísly (namísto 8 bitových čísel).

9.3. KOMPRESNÍ VÝSLEDKY

V této kapitole jsou čtyři tabulky, které obsahují naměřený kompresní poměr (v bpc) pro všechny uvedené algoritmy. Výsledky jsou rozděleny podle jazyku (čeština je v Tabulce 8 a v Tabulce 10, angličtina v Tabulce 9 a v tabulce 11) a podle použitého kódování souborů (kódování souborů WINDOWS-1250 je v Tabulce 8 a v Tabulce 9, kódování UTF-8 v Tabulce 10 a v Tabulce 11).

Soubory v měřících množinách M_{CZ}, M_{EN}, $M\text{-}U_{CZ}$ a $M\text{-}U_{EN}$ jsou rozděleny podle velikosti do 8 kategorií: méně než 100B, 100B – 1kB, 1kB – 10kB, 10kB – 50kB, 50kB – 200kB, 200kB – 2MB, stejně nebo více než 2MB. Prázdné kategorie nejsou v tabulce vůbec uvedeny.

V posledním sloupci (soubory větší než 2MB) tabulek v anglickém jazyce (viz Tabulka 9 a Tabulka 11) není uveden kompresní poměr u slovní verze aritmetického kódování. To je z toho důvodu, že se kompresní poměr takových souborů (konkrétně jednoho 4MB souboru) nezdařilo naměřit. Chyba, spíše než v kompresním algoritmu, je na straně „živého“ CD s operačním systémem Linuxu SLAX, jehož muselo být použito při měření, neboť implementace ACM kodéru je funkční pouze pod Unixem/Linuxem. Dá se ovšem předpokládat, že dosažený kompresní poměr bude zhruba odpovídat předchozímu sloupci.

Výsledky AritSyll-u s různými metodami dělení slabik dopadly prakticky stejně. Rozdíly mezi nimi jsou v řádech setin bpc, jen výjimečně je rozdíl větší než 0.1 bpc. V každém sloupci všech tabulek je zvýrazněn výsledek, který je nejlepší (mezi různými metodami dělení na slabiky). V anglickém jazyce

Metoda\ Soubor	< 100B	100B – 1kB	1kB – 10kB	10kB – 50kB	50kB – 200kB	200kB – 2MB
AS_L	5.63	4.54	4.21	3.93	3.72	3.69
AS_R	5.49	4.47	4.12	**3.85**	**3.64**	**3.63**
AS_{ML}	5.55	4.54	4.21	3.96	3.75	3.72
AS_{MR}	**5.40**	**4.43**	**4.10**	**3.85**	3.65	3.64
AS_W	5.68	4.58	4.07	3.53	3.12	2.97
HS_{ML}	6.22	4.85	4.43	4.05	3.86	3.80
HS_W	6.66	5.43	4.92	4.32	3.83	3.55
ACM_{CH}	8.10	6.02	5.39	5.05	5.02	5.02
ACM_W	8.17	5.57	4.77	4.12	3.77	3.58
bzip2	13.05	6.05	4.62	3.54	3.06	2.79

Tabulka 8: Kompresní poměr (v bpc) vybraných algoritmů na dokumentech v českém jazyce

Metoda\ Soubor	100B – 1kB	1kB – 10kB	10kB – 50kB	50kB – 200kB	200kB – 2MB	≥ 2MB
AS_L	**3.78**	**3.38**	**3.24**	**3.10**	**2.99**	**2.94**
AS_R	3.84	3.46	3.32	3.17	3.06	3.02
AS_{ML}	3.83	3.44	3.30	3.15	3.03	3.00
AS_{MR}	3.82	3.44	3.30	3.16	3.05	3.01
AS_W	3.31	2.66	2.36	2.26	2.30	2.40
HS_{ML}	3.93	3.45	3.25	3.19	3.17	3.16
HS_W	3.72	2.94	2.50	2.46	2.50	2.55
ACM_{CH}	5.84	4.83	4.59	4.59	4.59	4.69
ACM_W	4.49	2.97	2.38	2.39	2.38	--
bzip2	5.28	3.03	2.18	2.13	2.28	2.35

Tabulka 9: Kompresní poměr (v bpc) vybraných algoritmů na dokumentech v anglickém jazyce

jasně kraloval AS_L, který je nejlepší ve všech velikostních kategoriích. V českém jazyce je nejlepší AS_{MR}, neboť i když není pro větší soubory nejlepší, rozdíl od AS_R jsou jen minimální.

Kompresní poměr slovní verze algoritmu AritSyll dopadl s přehledem lépe než u slabikové komprese. Díky inicializaci statistického modelu slovníkem častých slov má i na malých souborech AS_W lepší kompresní poměr (až na první dvě kategorie v češtině, kde je výsledek poměrně těsný) než libovolný slabikový AritSyll.

Z porovnání se slabikovou verzí HuffSyll-u, odchází slabikový AritSyll jednoznačně vítězně. Na dokumentech v českém jazyce je AritSyll lepší až o 0.8 bpc. Tento rozdíl se ovšem s rostoucí velikostí souborů snižuje až na 0.16 bpc. Na dokumentech v anglickém jazyce jsou výsledky mnohem těsnější – slabikový AritSyll je zde průměrně lepší jen o 0.14 (v kategorii 10kB – 50kB dokonce jen o 0.01). Ještě větší rozdíl je mezi slovními verzemi obou algoritmů. Slovní AritSyll je na dokumentech v českém jazyce lepší o 0.6 – 1.0 bpc, což je velice příznivý výsledek. Na dokumentech v anglickém jazyce je rozdíl „jen“ 0.15 – 0.4 bpc. V obou případech se rozdíl zmenšuje s rostoucí velikostí souborů.

Před porovnáváním výsledků kódování ACM_{CH} a ACM_{W} je nutno ještě poznamenat, že oba algoritmy nerozeznávají česká písmena s diakritikou jako písmena. Proto jsou jejich výsledky na dokumentech v českém jazyce o třídu horší než na dokumentech v anglickém jazyce.

Z porovnávaných kompresních algoritmů jednoznačně nejhůře dopadl písmenný algoritmus ACM_{CH}. Jeho výsledky jsou výrazně horší než výsledky všech slabikových (ve všech kategoriích) i slovních algoritmů (ve většině kategorií). Výsledky slovního algoritmu ACM_{W} jsou už o něco lepší, než jeho písmenné varianty. Na dokumentech v anglickém jazyce je skoro stejně dobrý jako slovní verze AritSyll-u, až na soubory malé velikosti, kde má AritSyll díky inicializaci slovníkem častých slov jednoznačně navrch. Na dokumentech v českém jazyce algoritmus ACM_{W} pochopitelně úplně propadnul a jeho výsledky jsou dokonce horší než výsledky slabikového AritSyll-u.

Algoritmus bzip2 dosáhl nejlepších kompresních výsledků v obou jazycích pro soubory nad 50kB. Nicméně odstup druhého v pořadí – slovní verze AritSyll-u – není nikdy větší než 0.18 bpc (ve většině případů spíše do 0.06 bpc). Pro menší soubory je díky inicializaci slovníkem častých slov vždy lepší slovní AritSyll a mnohdy i slabikový AritSyll.

Metoda\ Soubor	< 100B	100B – 1kB	1kB – 10kB	10kB – 50kB	50kB – 200kB	200kB – 2MB
AS_{L}	5.53	4.19	3.81	3.56	3.36	3.34
AS_{R}	5.40	4.12	3.74	**3.49**	**3.29**	**3.29**
AS_{ML}	5.46	4.18	3.82	3.58	3.38	3.37
AS_{MR}	**5.32**	**4.08**	**3.72**	**3.49**	3.30	**3.29**
AS_{W}	5.53	4.22	3.70	3.20	2.83	2.70
HS_{ML}	10.50	6.79	5.70	4.91	4.46	4.34
HS_{W}	11.11	7.32	6.09	5.06	4.35	4.03
ACM_{CH}	8.24	6.12	5.48	5.16	5.12	5.14
ACM_{W}	8.41	5.40	4.47	3.81	3.46	3.27
bzip2	13.90	5.90	4.32	3.27	2.79	2.54

Tabulka 10: Kompresní poměr (v bpc) vybraných algoritmů na dokumentech v českém jazyce v UTF-8

Metoda\ Soubor	100B – 1kB	1kB – 10kB	10kB – 50kB	50kB – 200kB	200kB – 2MB	≥ 2MB
AS_L	**3.83**	**3.39**	**3.24**	**3.10**	**2.99**	**2.94**
AS_R	3.91	3.47	3.32	3.17	3.06	3.02
AS_{ML}	3.87	3.45	3.30	3.15	3.03	3.00
AS_{MR}	3.87	3.45	3.31	3.16	3.05	3.01
AS_W	3.36	2.67	2.36	2.26	2.30	2.40
HS_{ML}	4.12	3.48	3.25	3.20	3.18	3.16
HS_W	3.92	2.97	2.50	2.46	2.50	2.55
ACM_{CH}	5.87	4.83	4.59	4.60	4.59	4.67
ACM_W	4.54	2.98	2.39	2.39	2.38	--
bzip2	5.40	3.05	2.20	2.13	2.28	2.35

Tabulka 11: Kompresní poměr (v bpc) vybraných algoritmů na dokumentech v anglickém jazyce v UTF-8

Při porovnávání výsledků dokumentů v kódování UTF-8 si nelze nevšimnout, že hodnoty uvedené v Tabulce 11 jsou prakticky stejné jako hodnoty v Tabulce 9. Důvod je velice jednoduchý – soubory v kódování UTF-8 jsou pro anglický jazyk úplně (až na úvodní byte-order značku) shodné jako soubory v normálním osmibitovém kódování. Proto tyto výsledky nejsou, z pohledu komprese souborů v kódování UTF-8, vůbec zajímavé.

Na dokumentech v českém jazyce a v kódování UTF-8 mají všechny testované slovní a slabikové algoritmy kompresní poměr o 0.1 – 0.4 bpc lepší (s rostoucí velikostí souborů se rozdíl zvětšuje), než při kódování WINDOWS-1250. Důvodem zlepšení kompresního poměru je to, že znaky s diakritikou jsou v souborech dvoubytové, kdežto slabikové i slovní metody je chápou jako jedno slovo/písmeno. Nelze se nezmínit o tom, že u AritSyll-u bylo toto zlepšení nejlepší. Jedinou výjimkou je algoritmus HuffSyll (slabiková i slovní verze), který je inicializovaný špatným slovníkem a jehož kompresní poměr se značně zhoršil (až o 4.0 bpc).

Nejlepšího kompresního poměru pro soubory nad 50kB opět dosáhl bzip2. Odstup od slovní varianty AritSyll není ovšem větší než 0.16 bpc. Pro malé soubory do 1kB je nejúspěšnější slabikový AS_{MR}. V kategoriích mezi 1kB – 50kB je nejlepší slovní AritSyll.

10.ZÁVĚR

V této práci je popsán statistický kompresní algoritmus AritSyll založený na aritmetickém kódování. Ačkoli je s ním možné komprimovat všechny soubory, je speciálně navržen pro kompresi textu. Algoritmus existuje ve slabikové a slovní verzi. Slabiková verze dělí vstupní text slova, které dále dělí na slabiky (resp. na úseky písmen, které slabiky připomínají) a jež kóduje. Slovní verze dělí text pouze na slova, která kóduje. Při kódování slov i slabik se k zakódování jejich typu používá slabikový/slovní kontext, který se snaží zachytit skladbu vět a slov v přirozeném jazyce a tím zlepšit kompresi. Pro efektivní kompresi malých souborů je statistický model na začátku komprese naplněn všemožnými statistickými informacemi (slovníkem častých slov/slabik, slovníkem častých písmen, délkami slabik a kontextem slov/slabik), které závisí na zvoleném jazyce, algoritmu dělení slov na slabiky a kódování souboru.

Pomocí kompresního algoritmu AritSyll je možné komprimovat text v libovolném jazyce, ve kterém se slova skládají z písmen. Algoritmus je implementován pro český a anglický jazyk a není obtížné ho rozšířit i pro jiné jazyky.

Algoritmus AritSyll je díky použití rychlých datových struktur (binární indexovaný strom, trie, hašovací tabulky) třikrát rychlejší, než jeho předloha HuffSyll, za cenu mírného zvýšení paměťových nároků. Ty ale nejsou nijak zásadní a souvisí především s vylepšením o kódování souborů ve formátu UTF-8. Dosažený kompresní poměr na měřící množině dokumentů v českém jazyce je oproti HuffSyll výrazně lepší a pro dokumenty v anglickém jazyce jen mírně lepší. Zlepšení lze přičíst aritmetickému kódování i novému kontextovému modelu. V porovnání s ostatními kompresními algoritmy je slabikový i slovní AritSyll vždy lepší než písmenná komprese a pro velmi malé soubory je slabikový AritSyll lepší než ostatní slovní metody. Slovní verze AritSyll je pro malé a střední soubory lepší než bzip2 a pro velké soubory přibližně stejně dobrá jako bzip2.

Výsledky slabikové komprese nejsou špatné, nejsou ovšem ani nijak skvělé. Je pravdou, že slabiková komprese má výsledky lepší než písmenná komprese a pro malé soubory i než většina testovaných slovních algoritmů. Existuje ovšem určitá skepse, zda tento výsledek je důsledek slabikové komprese, nebo pouze dobrou inicializací slovníkem častých slabik. Dosažené výsledky slovního algoritmu lze oproti tomu hodnotit velice kladně.

Nevýhodou popsaného algoritmu je, že se musí pro každý jazyk, ve kterém má kódovat, inicializovat statistickými informacemi. Konkrétně slovníkem častých slov/slabik, slovníkem častých písmen, délkami slabik a kontextem slabik/slov. Odměnou za tuto inicializaci je sice lepší komprese opravdu malých souborů, ale pro získání všech informací je nutná dostatečně velká a reprezentativní množina dokumentů pro každý jazyk a správný postup pro jejich získávání. Z tohoto důvodu se uvedený algoritmus (ani slabikový, ani slovní) nikdy nemůže stát univerzálním nástrojem pro kompresi textu. Bylo by zajímavé vyzkoušet postup z (Moffat, Neal, & Witten, 1998), kde je statistický model na začátku prázdný a symboly jsou na začátku inkrementovány o velké číslo. Při každé aktualizaci statistického modelu (půlení všech četností) se vydělí dvěma i toto číslo. Z výsledků algoritmu ACM_W je vidět, že tento postup velice rychle a správně naplní statistický model potřebnými informacemi.

V celé práci není ani zmínka o „Kompresi slovníku na základě struktury“ z (Lánský & Žemlička, 2006a), jak vyžaduje zadání, neboť tento postup není pro slabikovou kompresi (resp. pro AritSyll) vhodný. Výsledky slabikové komprese totiž ukazují, že je výhodná především pro soubory menší a střední

velikosti, kde může být konkurentem slovních metod. Ovšem postup, kdy se slovník předává dekodéru společně s kódovou zprávou je výhodný výhradně pro velké soubory. Navíc metody pro kompresi slovníku TD2 a TD3 navrhované v (Lánský & Žemlička, 2006a), které teprve začínají být zajímavé, využívají přerovnání abecedy tak, aby malá písmena, velká písmena, čísla a speciální znaky byly blízko u sebe. To je ovšem možné pouze v klasické osmibitové abecedě, v kódování UTF-8 je to už skoro nemožné.

11. CITOVANÁ LITERATURA

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). Chapter 11: Hash Tables. V *Introduction to Algorithms* (2. vyd., stránky 221-252). MIT Press and McGraw-Hill.

Fenwick, P. (1993). *A New Data Structure for Cumulative Probability Tables.* Department of Computer Science, The University of Auckland.

Knuth, D. (1998). Section 6.4: Hashing. V *The Art of Computer Programming, Volume 3: Sorting and Searching* (stránky 513–558). Addison-Wesley.

Koubek, V. (2005). *Datové struktury II, Přednáška.* Faculty of Mathematics and Physic, Charles University.

Lánský, J. (2005). *Slabiková komprese, Master Thesis.* Faculty of Mathematics and Physic, Charles University.

Lánský, J., & Žemlička, M. (2006). Compression of a Dictionary. V V. Snášel, K. Richta, & J. Pokorný, *Proceedings of the Dateso 2006 Annual International Workshop on DAtabases, TExts, Specifications and Objects* (stránky 11-20). CEUR-WS.

Lánský, J., & Žemlička, M. (2006). *Compression of Small Text Files using Syllables.* Faculty of Mathematics and Physic, Charles University.

Moffat, A., Neal, R. M., & Witten, I. H. (1998). Arithmetic Coding Revisited. *ACM Transactions on Information Systems, Vol. 16* (stránky 256–294). ACM.

Seward, J. *bzip2*. Získáno 28. 7 2007, z bzip2: http://www.bzip.org/

UTF-8 - Wikipedia. Získáno 5. 7. 2007, z Wikipedia: http://en.wikipedia.org/wiki/Utf-8