VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KOMPRESNÍ METODY ZALOŽENÉ NA KONTEXTOVÉM MODELOVÁNÍ

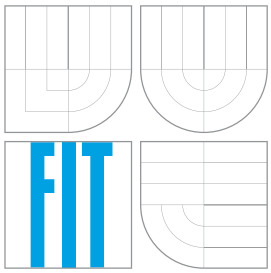
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MATÚŠ KRUPIČKA

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KOMPRESNÍ METODY ZALOŽENÉ NA KONTEXTOVÉM MODELOVÁNÍ

COMPRESSION METHODS BASED ON CONTEXT MODELLING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MATÚŠ KRUPIČKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. DAVID BAŘINA
SUPERVISOR

BRNO 2010

Abstrakt

Náplní téhle bakalářské práce je analýza kontextových kompresních metod, jejich vlastností a možností využití. Práce se zaměřuje a podáva detailní přehled statistické motody PPM (metoda predikce částečné shody). Je popsán programový návrh táto metody jako aj samotná implementace. Závěrem taky porovnání výsledků činnosti tohoto programu s již existujícími implementacemi komprimace dat.

Abstract

This thesis is dedicated to analysis of context-based compression methods, their characteristics and possibilities of their usage. Thesis focuses on and hands out detail overview of statistical method PPM (prediction by partial matching). Program design of this method is described as well as the implementation. Lastly, also the outcomes of this program are compared with already existing implementations of data compression.

Klíčová slova

Komprese dat, statistické kompresní metody, aritmetické kódování, predikce částečné shody, PPM.

Keywords

Data compression, statictical compression methods, arithmetic coding, prediction by partial matching, PPM.

Citace

Matúš Krupička: Kompresní metody založené na kontextovém modelování, bakalářská práce, Brno, FIT VUT v Brně, 2010

Kompresní metody založené na kontextovém modelování

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Davida Bařiny. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Matúš Krupička
10. května 2010

Poděkování

Ďakujem vedúcemu práce Ing. Davidovi Bařinovi za odbornú pomoc, ktorú mi poskytol pri vypracovávaní tejto práce.

Obsah

Kapitola 1

Úvod

Dnes sa stretávame s komprimovanými dátami pomerne často. Tieto techniky sa používajú hlavne na ušetrenie pamäťového priestoru v počítačoch. Metód pre kompresiu dát je mnoho. Táto práca sa venuje jednej z nich, metóde *PPM*. Ide o metódu využívajúcu aritmetické kódovanie a modelovanie kontextu. Tieto pojmy ako aj princípy a algoritmy používané pri ich implementácii sú vysvetlené v nasledujúcich kapitolách.

Druhá kapitola charakterizuje kompresiu dát, tiež podáva základné rozdelenie typov kompresných metód ako aj metriky popisujúce činnosť týchto metód, aby mal čitateľ predstavu o pojmoch použitých v ďalších kapitolách. Ďalej sa táto kapitola venuje popisu štatistických metód aj pojmami s nimi súvisiacimi ako sú modelovanie, entropia a redundancia.

Ďalšia kapitola sa venuje aritmetickému kódovaniu, ako jednej z častí metódy *PPM*. Vysvetľuje jeho princípy a venuje sa implementácii aritmetického kodeku a vysvetľuje metodiky použité pri tejto implementácii, ktorými sú celočíselná aritmetika a škálovanie.

Kaptitola 4 poskytuje detailný popis *PPM*, ako modelu pre aritmetický kóder. Opisuje princípy používané v tejto metóde. Záverom sa kapitola venuje implementácii *PPM*, dátovým typom, ktoré používa a komentuje spoluprácu s aritmetickým kóderom.

Predposledná kapitola udáva výsledky dosiahnuté implementáciou kompresnej metódy *PPM* pomocou korpusov. Venuje sa jej vylepšeniu zvolením vhodných parametrov. Nakoniec je *PPM* porovnaná s inými často využívanými metódami.

Záverom sa komentujú dosiahnuté výsledky a prípadné ďalšie možné smerovanie tejto práce.

Kapitola 2

Kompresia dát

V nasledujúcej kapitole sú uvedené niektoré dôležité pojmy kompresie, ako aj niektoré vlastnosti vyjadrujúce výkon kompresných metód. Na prípravu tejto kapitoly boli použité materiály z publikácie [11].

2.1 Charakteristika kompresie

Kompresia dát je proces premeny vstupného dátového toku na výstupný dátový tok, ktorý má menšiu veľkosť. Dátový tok môže byť buď súbor, alebo pamäťový zásobník (buffer). Inak povedané, kompresia umožňuje ukladanie informácií (do pamäte počítača), tak aby zaberali menší priestor.

Tento prístup umožňnuje akumuláciu veľkého monžstva dát. Dobrým príkladom takého využitia sú niektoré typy audio, či video súborov, alebo aj obrázkov. Rovnako nachádza využitie pri prenose dát, tým že zužuje spotrebu šírky pásma. Užívatelia teda nemusia čakať na prenos príliš dlho, keďže tieto dáta môžu byť a často aj sú skomprimované.

Na druhej strane ale komprimované dáta trpia nevýhodou, lebo z týchto dát nie je možné získať potrebné informácie. Je nutné skomprimované dáta najskôr dekomprimovať. Táto činnosť ale vyžaduje softwarové (imprelentácia algoritmu dekompresie popri kompresnom algoritme), alebo hardwarové rozšírenie.

Rôzne typy dát si vyžadujú pre kompresiu rôzne metódy, dosahujúce rôzne výsledky. Všetky tieto metódy ale pracujú na rovnakom princípe, tým je odstraňovanie redundancie zo vstupných dát. Každá ucelená kolekcia dát má svoju štruktúru, ktorá môže byť využitá na dosiahnutie zmenšenej reprezentácie týchto dát. V praxi sa s kolekciou náhodných dát nestretávame často (výnimkou sú už skomprimované dáta). Je dobré si uvedomiť, že náhodné dáta nemajú žiadnu štruktúru, teda neobsahujú žiadnu redundanciu. Teda je jasné, že skomprimované dáta prichádzajú o štruktúru (tým pádom o redundanciu) a už nie je možné ich ďalej komprimovať. Ak by tento postup bol možný, viedlo by to k tomu, že by sa dalo veľké množstvo dát niekoľkými komprimáciami zmenšiť na veľkosť jediného bitu. Ale je zrejmé, že jeden bit nedokáže udržať žiadnu komplexnú informáciu.

2.2 Základné pojmy kompresie dát

Kompresor, alebo *kóder* je program, ktorý komprimuje vstupný dátový tok na výstupný, skomprimovaný tok dát. Naopak *dekóder*, príp. *dekompresor* pracuje opačne. *Kodek* je program fungujúci aj ako kóder, aj ako dekóder.

Neadaptívna kompresia nemodifikuje svoje operácie a parametre v závislosti od druhu vstupných dát. Tieto metódy sú určené na konkrétny dátový typ a na iných nedosahujú dobré výsledky. Iný prístup majú metódy, ktoré svoje parametre a operácie nastavujú rôzne prerôz ne typy vstupných dát. Napríklad dvojitým prechodom vstupných dát, kde v prvom prechode určujú štatistiky dát a nastavujú svoje parametre a v prechode druhom podľa týchto parametrov dáta komprimujú. Ide o metódy *adaptívne*.

V istých kompresných metódach sa dá dosiahnuť lepších výsledkov, pri strate niektorých údajov, t.z. ak sú skomprimované dáta dekomprimované, nie sú s pôvodnými dátami zhodné. Nazývajú sa *stratové*. Takéto metódy majú využitie pri druhoch dát, kde je nemožné, alebo len veľmi ťažké rozpoznať rozdiely. Využívaju nedokonalosti ľudských zmyslov, teda sa používajú pri kompresii obrázkov, zvuku, či videa. Naopak pri kompresii textu by straty mohli mať za následok úplnu zmenu. Tento problém môže byť o to závažnejší, ak by šlo o zdrojový kód programu. Preto sa pri kompresii textu využívajú *bezstratové metódy*. Majme vstupný dátový tok A a kompresor X, ktorý skomprimuje A na B. Je možné, avšak zbytočné použiť ďalší kompresor Y so vstupom B a tým dostať skomprimovaný tok dát C. V prípade, že X a Y sú bezstratové, tak dekódovaním C dekóderom Y získame B, ktoré ďalším dekódovaním pomocou X podá pôvodný dátový tok A. Avšak ak je jeden z kompresorov je stratový, dekompresia C metódou Y vyprodukuje súbor dát B' (rôzne od B). Prechod B dekompresorom X vyprodukuje súbor veľmi odlišný od pôvodneho A, alebo dokonca chybný, lebo X nemusí byť schopný čítať B'. Stratová kompresia by mala byť *vnímavá*, teda sa snažiť odstrániť len tie dáta, ktoré nebudú spozorovateľné našimi zmyslami.

V prípade, že dáta je nutné komprimovať častejšie ako ich dekomprimovať, je lepšie implementovať komprimačný algoritmus, tak aby pracoval rýchlejšie a menej komplexnejšie. Takýto princíp sa nazýva *asymetrická kompresia*, pretože kóder pracuje inak ako dekóder. Inak povedané, kóder a dekóder nepoužívajú rovnaký algoritmus fungujúci opačným smerom, teda ako tzv. *symetrická metóda*. Vyššie uvedený príklad sa používa pri vytváraní zálohy. Dáta sa komprimujú a zálohujú pravidelne a predpokladá sa, že sa k nim nebude musieť pristupovať príliš často a tak sa dekóder používa len málo. Mnoho moderných metód je asymetrických. často sa stretávame s tým, že popis kompresných metód pozostáva z dekódera a formátu skomprimovaných dát. Vtedy sa akýkoľvek kóder generujúci takýto formáte nazýva *kompilant*. Výhoda tohto princípu spočíva v tom, že ktokoľvek smie vytvoriť vlastné kódovacie algoritmy. V tomto prístupe sa kóder nazýva *algoritmický* a dekóder *deterministický*.

Väčšina kompresných metód pracuje v prúdovom režime (angl. *streaming mode*), kde kodek pracuje so vstupom niekoľkých bytov, spracuje ich a pokračuje do konca dátového toku. Iný prístup je *blokový*, ktorý spracúva vstup blok po bloku, pričom každý je skomprimovaný inak. Veľkosť bloku býva zvyčajne užívateľským parametrom.

Ďalší dôležitý pojem pri kompresii je *rozdielovanie súborov*. Ide o činnosť pri ktorej sa hľadajú a komprimujú rozdiely v súboroch. Toto má využitie napr. ak sa na disku nachádzajú dve kópie rovnakého súboru. Ak sa jedna z nich zmení, ale chcem aby obe zostali identické, je nutné nahradiť aj druhú kópiu súboru. Tento súbor však môže byť príliš veľký. Tak sa zasielajú druhej kópii len samotné zmeny, ktoré sú naviac skomprimované.

Pri kompresii dát dochádza k náhrade dátového vstupu na kratší výstup bez ohľadu na to, čo znamená. Jediný spôsob ako takto zakódovaným dátam doplniť význam je ich dekomprimácia a to je možné len tak, že vieme ako boli zakódované. Tieto metódy sa nazývajú *fyzické*. Niektoré metódy sú *logické*, sú to také, pri ktorých sa individuálne prvky vstupných dát nahradzujú kratšími položkami. Dajú sa využiť však len na špeciálne typy

dát.

2.3 Metriky kompresie

Kompresný pomer (*compression ratio*) je definovaný ako

$$Kompresný\,pomer = \frac{veľkosť\,výstupných\,dát}{veľkosť\,vstupných\,dát}.$$

Očakáva sa výsledok menší ako 1. Udáva koľko percent veľkosti vstupného toku zaberá výstupný tok. V prípade, že je výsledok väčší ako 1 ide o negatívnu kompresiu, tzn. výstupný tok dát je väčší.

Tento pomer sa tiež nazýva *bit na bit, bit na Byte*, alebo *bpb, b/B* (z angl. *bit per bit, bit per Byte*), lebo v priemere udáva počet bitov komprimovaných dát potrebných na uloženie jedného bitu, príp. Bytu vstupného toku dát. Podobne pri kompresii obrázkov hovoríme o *bit na pixel* (angl. *bit per pixel*), *bpp* a pri kompresii textu *bit na znak*, alebo *bpc* (*bit per character*). Spoločne sa tieto pojmy označujú *bitová rýchlosť* (*bitrate*). *Bitová správa* (*bit budget*) popisuje funkciu jednotlivých bitov skomprimovaného toku (napr. štatistické údaje kompresie).

Opak kompresného pomeru sa nazýva *Kompresný faktor* (*compression factor*), je daný nasledujúcim vzorcom

$$Kompresný\,faktor = \frac{veľkosť\,vstupných\,dát}{veľkosť\,výstupných\,dát}.$$

Naopak, ako pri pomere, očakávaný výsledok je väčší ako 1 a znamená kompresiu, menší ako 1 znamená expanziu. Na meranie miery kompresie sa používa aj výraz

$$100 \times (1 - kompresný\,pomer),$$

vyjadruje percentuálny pomer veľkosti výstupu oproti vstupnému toku. Alebo doplnok do 100 vyjadruje percentuálne ušetrené miesto oproti pôvodným dátam.

Ďalšou metrikou výkonu kompresných metód je *kompresný zisk* (*compression gain*), je definovaný

$$Kompresný\,zisk = 100 \log_e \frac{referenčná\,veľkosť}{komprimovaná\,veľkosť},$$

kde referenčná veľkosť je veľkosť buď vstupného toku dát, alebo skomprimovaný dátový tok vytvorený štandardnou beystratovou kompresnou metódou. Jednotkou je *percentuálny logaritmický pomer*, označuje sa ÷.

Rýchlosť kompresie, prevažne na špeciálnych hardwarových zariadeniach, sa meria v *cykloch za byte*, *CPB* (*cycles per byte*). Ako názov napovedá, jedná sa o priemerný počet strojových cyklov potrebných na skomprimovanie jedného bytu.

Rozdiely zaznamenané stratovou kompresiou obrázkov a videa uvádza *mena square error* (*MSE*) a *peak signal to noise ratio* (*PSNR*). Rovnako aj relatívna kompresia sa používa na určenie kompresného zisku u stratových kompresných metód audio dát. Vyjadruje kvalitu kompresie počtom bitov o ktorý je každá zvuková vzorka zredukovaná.

2.4 Štatistické kompresné metódy

Kompresné metódy založené na kontextovom modelovaní, konkrétne aj metóda PPM, ktorú rozoberá táto práca, patria do skupiny štatistických metód. Základnou myšlienkou týchto metód je, že priradzujú vstupným symbolom kódy s premennou dĺžkou. Rovnako sa opierajú o frekvenciu výskytu jednotlivých symbolov. Táto frekvencia je akousi štatistikou vstupných symbolov udávajúca pravdepodobnosti s akými sa symbol vyskytuje vo vstupnom dátovom toku. Podľa týchto pravdepodobností sa priraďujú kratšie kódy symbolom s vysokou pravdepodobnosťou výskytu a naopak dlhšie kódy symbolom s pravdepodobnosťou nízkou.

Medzi štatistické metódy patria dve základné, aritmetické kódovanie (ktorému sa venuje kapitola 3) a Huffmanovo kódovanie (viac sa čitateľ môže dozvediet z [11][6]) a ďalšie metódy z nich odvodené.

2.4.1 Modelovanie dát

Modelovanie má na starosti určovanie a prideľovanie pravdepodobností jednotlivým vstupným symbolom, s ktorými pracujú štatistické kompresné metódy. Modelovaním vzniká *model dát*. Model dát v praxi je zvyčajne tabuľka alebo stromová štruktúra. Tieto štruktúry sa ľahko prehľadávajú príp. upravujú. Kóder a dekóder musia pracovať s rovnakým modelom, na to existuje viac možností:

- *Statické modelovanie*, podobne ako bolo vysvetlené v podkapitole 2.2, je typ modelovania, ktorý používa rovnaký model na všetky dáta bez ohľadu na ich typ. Modely sa spravidla vytvárajú z očakávaných vstupov. Kóder aj dekóder majú každý jednu kópiu modelu. Tento prístup sa využíva hlavne kôli jednoduchosti a rýchlosti, avšak dosahuje špatné výsledky, ak vstupné dáta nezodpovedajú očakávaným.

- *Semiadaptívne modelovanie* používa pre kompresiu iný model dát, v závislosti na ich type. Dáta sa pred kompresiou prečítajú a vytvorí sa z nich model. Potom sa tieto dáta skomprimujú. Model sa pred dekódovaním musí poslať dekóderu. Nevýhodou je práve táto nutnosť dvoch prechodov. Tiež sa môže stať že model dát bude väčší ako samotné skomprimované dáta.

- *Adaptívne modelovanie* pracuje viac–menej ako semiadaptívne modelovanie, ale odstraňuje nutnosť dvojitého prechodu vstupnými dátami. Na začiatku kóder a dekóder predpokladajú nejaký štartovný model, ktorý sa počas samotného kódovania upravuje. Ak sa zaručí, že algoritmus úpravy modelu pre kóder aj dekóder upravujú model rovnako, ich model bude stále zhodný. Týmto prístupom tiež odstraňuje nutnosť prenášať model od kódera k dekóderu. Metóda *PPM* patrí práve medzi tieto adaptívne modelovacie techniky.

Štatistické metódy sa ďalej dajú rozdeliť podľa princípov, na ktorých fungujú. Medzi tieto prístupy patrí:

- *Využitie pravdepodobnosti výskytu symbolov* využíva, ako názov napovedá, štatistické informácie o výskytoch symbolov vo vstupe.

- *Využitie kontextu* v modelovaní určuje pravdepodobnosti vstupných symbolov nie len na základe výskytu vo vstupe, ale aj na základe kontextu, v ktorom sa daný symbol už vyskytol. Kontextom sa bližšie venuje kapitola 4. Metóda PPM sa tiež opiera pri určovaní pravdepodobností symbolov o kontext.

- *Využitie konečných automatov v kompresií* niektorými štatistickými metódami na určenie pravdepodobnosti symbolov. Tieto metódy sú však na rámec tejto práce, ale bližšie sa im venuje aj [11].

2.4.2 Entropia a redundancia

Intuitívne vieme, čo je to *informácia*. Neustále dostávame a odosielame informácie vo forme textu, reči a obrazu. Tiež cítime, že informácia je neuchopiteľná matematická veličina, ktorá se nedá presne definovať, zachytiť ani merať. Jednou z charakteristík informácie v teórii informácie je *entropia*. Viac k teórii informácie je možné nájsť v publikáciách [11][12][5] a [8], ktoré boli použité ako podklady pre túto podkapitolu.

Kód K sa dá formálne definovať ako trojica $K = (S, C, f)$ kde S je konečná množina *zdrojových jednotiek*, C je konečná množina *kódových jednotiek* a f je injektívne zobrazenie z S do C^+.

Entropia je miera množstva informácie. Majme zdrojové jednotky $S = x_1, x_2, \ldots, x_n$, Ak pravdepodobnosť výskytu zdrojovej jednotky x_i je p_i; $1 \leq i \leq n$, potom *entropia informačného obsahu jednotky* je

$$H_i = -\log_2 p_i \quad [bit].$$

Zdrojové jednotky vyskytujúce sa s vyššou pravdepodobnosťou obsahujú teda menej informácií ako jednotky s pravdepodobnosťou menšou. *Priemerná entropia zdrojovej jednotky je z* S je

$$H_{avg}(S) = \sum_{i=1}^{n} p_i H_i = -\sum_{i=1}^{n} p_i \log_2 p_i \quad [bit].$$

Entropia zdrojovej správy $X = x_{i_1}, x_{i_2}, \ldots, x_{i_k}; X \in S^+$ je

$$H(X) = -\sum_{j=1}^{k} p_{i_j} \log_2 p_{i_j} \quad [bit].$$

Ak použijeme pre zakódovanie zdrojovej jednotky x_i; $1 \leq i \leq n$, kódové slovo s dĺžkou d_i bitov, *dĺžka zakódovanej správy* X je

$$L(X) = \sum_{j=1}^{k} d_{i_j} \quad [bit].$$

Základy teórie informácie tiež vedú k definicii *redundancie*, ako nepriaznivého javu istého nadbytku informácií v spracovávaných dátach, takže neskôr budeme môcť rozumieť spôsobu vyčíslovania redundancie a rôznym metódám jej odstraňovania.

redundancia kódu C pre správu X je rozdiel dĺžky zakódovanej správy X a entropie pôvodnej správy X a teda

$$R(X) = L(X) - H(X) \quad [bit].$$

Priemerná dĺžka kódového slova K je

$$L_{avg}(K) = \sum_{i=1}^{n} d_i p_i \quad [bit].$$

Priemerná redundancia kódu je

$$R_{avg}(K) = L_{avg}(K) - H_{avg}(S) \quad [bit].$$

Kapitola 3

Aritmetické kódovanie

Aritmetické kódovanie je metódou bezstratovej kompresie dát. Patrí medzi základné štatistické metódy kompresie, podobne ako Huffmanovo kódovanie. Na rozdiel od neho ale aritmetická metóda nepriradzuje kód každému vstupnému symbolu, ale priradí kód celému vstupnému toku. Algoritmus metódy začína s určitým intervalom a postupne číta znaky vstupného toku, pričom používa ich pravdepodobnosti (frekvencie výskytu) k zužovaniu tohoto intervalu.

Tento prístup zužovania intervalu však postupne vytvára dlhšie čísla, čo vyžaduje stále viac bitov. Aby bolo možné dosiahnuť kompresiu, algoritmus je navrhnutý tak, že symboly s vysokou pravdepodobnosťou výskytu zúžia interval menej ako symboly s nižšou pravdepodobnosťou. Takto symboly s vysokou pravdepodobnosťou výskytu prispievajú do výsledného kódu menším počtom bitov.

Z teoretického hľadiska pracuje aritmetický kóder tak, že komprimuje na, alebo veľmi blízko k entropii rovnej $-\log_2 p_i$ [11].

3.1 Priebeh kódovania

Výstupom aritmetickáho kódovania je reálne číslo v intervale $\langle 0; 1)$, (tento interval je aj počiatočný interval pri zahájení výpočtov kódera). Najskôr však je nutné stanoviť, alebo aspoň odhadnúť frekvenciu výskytu jednotlivých symbolov vstupného toku a to napríklad prečítaním v prvom prechode vstupného toku v prípade dvojprechodovej kompresie. V prípade možnosti získania údajov o frekvenciách z iného zdroja je možné prvý prechod vynechať. Ďalším krokom je rozdelenie reťazca podľa týchto frekvencií. Kóder nasledovne prečíta znak zo vstupného toku a zredukuje tento interval v závislosti na pravdepodobnosti výskytu prečítaného znaku.

Príklad: Majme 3 symboly a, b a c s ich prevdepodobnostami výskytu $p_a = 0,4$, $p_b = 0,5$ a $p_c = 0,1$. Interval $\langle 0; 1)$ sa na podľa týchto pravdepodobností rozdelí na podintervaly na ktorých poradí nezáleží. V našom prípade na $\langle 0; 0,4)$, $\langle 0,4; 0,9)$ a $\langle 0,9; 1)$. Kódovanie reťazca *accba* začne na intervale $\langle 0; 1)$. Prvý symbol vstupu a redukuje tento interval na podinterval od jeho 0% do 40%, z čoho vyjde interval $\langle 0; 0,4)$. Ďalší symbol c redukuje nový interval od jeho 90% do 100%, teda na interval $\langle 0,36; 0,40)$. Nasledujúci symbol c redukuje interval rovnakým spôsobom na inetrval $\langle 0,396; 0,400)$. Potom symbol b redukuje interval od jeho 40% do 90%, čoho výsledkom je $\langle 0,3976; 0,3996)$. Nakoniec interval symbol a redukuje na $\langle 0,3976; 0,3984)$. Výsledný kód metódy je teda číslo z tohto intervalu.

Na uloženie intervalu sa použijú dve premenné: *Low* a *High*. Tieto premenné budú definované na začiatku kódovania a budú určovať interval $\langle Low; High)$ a ich počiatočné hodnoty budú zodopovedať počiatočnému intervalu, teda $Low = 0$ a $High = 1$. Pri postupnom spracovaní vstupných symbolov sa hodnoty *Low* a *High* k sebe približujú, a tým sa interval sa zužuje. Pri čítaní ďalšieho symbolu n zo vstupu sa hodnoty *Low* a *High* aktualizujú podľa predpisov:

$$Low = Low + Range \times LowRange(n),$$

$$High = Low + Range \times HighRange(n).$$

Kde *Low* je rozdiel medzi *High* a *Low* ešte pred pred ich aktualizáciu pomocou týchto predpisov. *LowRange* je dolná hranica podintervalu pravdepodobnosti aktuálne spracovávaného symbolu a *HighRange* je jeho hornou hranicou. *Range* je šírka podintervalu do ktorého symbol spadá.

Príklad: Pri aritmetickom kódovaní anglického slova *melee* sa inicializujú premenné *Low* a *High* na hodnoty 0 a 1. Štatistické údaje reťazca (získané v prvom prechode) sú udané v tabuľke 3.1, pričom na poradí jednotlivých znakov nezáleží. Pravdepodobnosti výskytu sú určené podielom počtu výskytov daného znaku k dĺžke reťazca. Taktiež uvádza jednotlivé podintervaly intervalu $\langle 0; 1)$ určené podobne ako v predošlom príklade. Samotný postup

Symbol	**Výskyt**	**Pravdepodobnosť**	**Interval**
e	1	0,6	$\langle 0; 0,6)$
l	1	0,2	$\langle 0,6; 0,8)$
m	2	0,2	$\langle 0,8; 1)$

Tabuľka 3.1: Štatistické údaje reťazca *melee*.

kódovania ukazuje tabuľka 3.2. Výsledným kódom je číslo z intervalu $\langle 0,872; 0,88064)$. Pri strojovom spracovaní sa vyberie také číslo, ktoré je reprezentované najmenším počtom bitov. V tomto prípade číslo 0,88.

Symbol		**Výpočet *Low* a *High***
m	Low	$0 + (1 - 0) \times 0,8 = 0,8$
	High	$0 + (1 - 0) \times 1 = 1$
e	Low	$0,8 + (1 - 0,8) \times 0 = 0,8$
	High	$0,8 + (1 - 0,8) \times 0,6 = 0,92$
l	Low	$0,8 + (0,92 - 0,8) \times 0,6 = 0,872$
	High	$0,8 + (0,92 - 0,8) \times 0,8 = 0,896$
e	Low	$0,872 + (0,896 - 0,872) \times 0 = 0,872$
	High	$0,872 + (0,896 - 0,872) \times 0,6 = 0,8864$
e	Low	$0,872 + (0,8864 - 0,872) \times 0 = 0,872$
	High	$0,872 + (0,8864 - 0,872) \times 0,6 = 0,88064$

Tabuľka 3.2: Priebeh kódovania reťazca *melee*.

3.2 Priebeh dekódovania

Dekóder pracuje podobne ako kóder. Inicializuje premenné *Low* a *High* na hodnoty 0 a 1, ktoré sa ďalším dekódovaním v závislosti na práve spracovanom symbole upravujú (zužujú interval). Pri zužovaní môže nastať problém, že sa intervaly donekonečna približujú ku kódu.

Príklad: Vezmeme zakódovaný reťazec *melee* z predošlého príkladu. Dekóder zrekonštruuje tabuľku štatistík symbolov, teda tabuľku 3.1. Potom prečíta výsledný kód reťazca, čo je 0,88. Dekóder vie, že kód 0,88 spadá do intervalu $\langle 0,8;1)$, a teda určí, že prvým symbolom je *m*. Následne upraví hodnoty *Low* a *High* rovnako ako kóder. Interval sa teda zúži na $\langle 0,8;1)$. Pri dekódovaní ďalšieho symbolu sa tento interval rozdelí podľa prevdepodobností výskytu symbolov na podintervaly a určí sa do ktorého z nich patrí kód 0,88. V tomto prípade to bude symbol *e*. Ako je zrejmé, dekóder presne kopíruje úpravy hodnôt *Low* a emphHigh ako kóder počas celého procesu dekódovania a to podľa tabuľky 3.2. Pri takomto dekódovaní však vzniká problém ako určiť, že by sa malo dekódovanie skončiť. Tabuľka 3.3 ukazuje ako by mohol dekóder ďalej pokračovať, čo by viedlo k tomu, že by sa dekódovanie nikdy nezastavilo. Je na nej vidno, že sa hranice intervalu nekonečne blížia k hodnote 0,88.

Symbol		**Výpočet *Low* a *High***
m	Low	$0{,}872 + (0,88064 - 0,872) \times 0,8 = 0,878912$
	High	$0{,}872 + (0,88064 - 0,872) \times 1 = 0,88064$
l	Low	$0{,}878912 + (0,88064 - 0,878912) \times 0,6 = 0,8799488$
	High	$0{,}878912 + (0,88064 - 0,878912) \times 0,8 = 0,8802944$

Tabuľka 3.3: Prípadný postup dalšieho dekódovania reťazca *melee*.

Podobná situácia nastáva pri dekódovaní symbolov s tzv. *skosenými pravdepodobnosťami*. Pravdepodobnosti výskytu symbolov v týchto prípadoch sú veľmi rozdielne, napríklad jeden zo symbolov sa vyskytuje s veľmi vysokou pravdepodobnosťou, zatiaľ čo ostatné s veľmi nízkou. Potom pri dekódovaní môže nastať situácia, kedy presne naopak, ako v predchádzajúcom príklade, sa algoritmus ukončí predčasne.

Problém ukončenia dekódovania je možné riešiť dvoma spôsobmi. Ak je pred dekódovaním známa dĺžka reťazca, dekóder ukončí činnosť pri jej dosiahnutí. Túto dĺžku môže pri kompresii zapísať kóder na výstup pred samotnými zakódovanými dátami, dekóder ju zas prečíta pred dekompresiou. Iná možnosť je zaviesť špeciálny koncový znak *eof* (end-of-file), určujúci koniec vstupných dát pre dekóder. Tomuto symbolu sa priradí veľmi malá pravdepodobnosť, ktorá sa pripojí do tabuľky. Dekóder potom končí po dekódovaní symbolu *eof*. Pre použité príklady sa upraví tabuľka so štatistickými údajmi (tabuľka 3.1) podľa tabuľky 3.4.

3.3 Implementácia aritmetického kodeku

Doteraz popisované spôsoby aritmetického kódovania používali pre výpočty reálne čísla. Pri implementácii tejto myšlienky však nastáva problém. Tým je, že algoritmy predpokladajú, že hranice intervalu aritmetického kódovania (premenné *Low* a *High*) sa ukladajú ako

Symbol	Pravdepodobnosť	Interval
e	0,5999994	$\langle 0; 0,5999994)$
l	0,1999998	$\langle 0,5999994; 0,7999992)$
m	0,1999998	$\langle 0,7999992; 0,999999)$
eof	0,000001	$\langle 0,999999; 1)$

Tabuľka 3.4: Upravené štatistické údaje reťazca *melee*.

premenné s veľmi malými hodnotami a veľkým číselným rozvojom za rádovou čiarkou. Aj keď dnes je možné uložiť číslo s pohyblivou rádovou čiarkou do pamäte s relatívne vysokou presnosťou, postupne by kód mohol naberať veľkosť aj niekoľkých desiatok megabajtov a operácie s takými veľkými číslami by boli náročné a pomalé.

Preto sa na implementáciu využíva celočíselná aritmetika. Na rozdiel od reálnych čísel pracuje algoritmus so sekvenciou bitov. Výstupom algoritmu je teda takáto jednoznačná sekvencia, ktorá môže byť ľahko uložená alebo ďalej spracovaná.

Nasledujúca podkapitola ako aj implementácia aritmetického kódera vychádza hlavne z [2] a [8], táto problematika je tiež popísaná v [11] ako aj v [1], kde je možné najsť tiež niektoré iné implementécie.

3.3.1 Celočíselná aritmetika

Pravdepodobnosti výskytu symbolov nie je možné určiť pomocou celých čísel. Keďže sa tieto pravdepodobnosti rovnajú početnostiam symbolov, budú sa využívať namiesto nej. Každý symbol bude mať priradený jeden interval, ktorý bude reprezentovať jeho početnosti. Bude ohranečený hodnotami *lowCount* a *highCount*. Kde *lowCount*, spodná hranica, je súčtom početností všetkých predchádzajúcich symbolov a horná hranica, *highCount*, je tento súčet zvýšený ešte o samotnú početnosť daného symbolu n. Pre zoradené znaky (napríklad abecedne, alebo podľa hodnôt ASCII) platí:

$$lowCount(n) = \sum_{i=0}^{n-1} count(i),$$

$$highCount(n) = lowCount(n) + count(n).$$

Hodnota *lowCount(n)* sa nazýva *kumulatívna početnosť* symbolu n. Tieto početnosti udávajú pravdepodobnosti, hranice podintervalov intervalu $\langle Low; High)$, ako boli používané v aritmetike reálnych čísel. Takže platí, že tieto početnosti sú rovné súčinu celkovej početnosti symbolov *totalSymbols* a zodpovedajúcej hornej, príp. dolnej hranici podintervalu pre daný symbol, teda:

$$lowCount(n) = lowRange(n) \times totalSymbols,$$

$$highCount(n) = highRange(n) \times totalSymbols.$$

lowRange(n) a *highRange(n)* sú týmito hranicami (podobne ako v predpisoch na strane 10). Takže tieto hranice pravdepodobností sa dajú určiť podielom kumulatívnych početností ku celkovej početnosti symbolov:

$$lowRange(n) = \frac{lowCout(n)}{totalSymbols},$$

$$highRange(n) = \frac{highCount(n)}{totalSymbols}.$$

Premenné *Low* a *High* budú uložené v celočíselných typoch. Na to je vhodné použiť typ, ktorý je 64-bitový (alebo inak dostatočne veľký) a bezznamienkový. To nám zaručí, že v premennej nikdy nebude záporné číslo, čo je vlastne, ako už vyplýva z algoritmov aritmetického kódovania nemožné. Takže, teoreticky, by sme mohli využiť plných 64 bitov na prácu s premennými tohoto typu. Okrem *Low* a *High* sa budú využívať aj rôzne iné pomocné premenné, s ktorými sa bude pracovať podobne. Celých 64 bitov však tiež nie je možne použiť, ak chceme zabrániť pretečeniu.

V tomto prístupe sa bude s premennými pracovať ako so sekvenciami bitov. Tieto bitové sekvencie si možeme predstaviť ako číslo nachádzajúce sa za rádovou čiarkou. Napríklad spodná hranica intervalu $0,71875_{10}$ to je 0.10111_2 bude reprezentovaná v programe premennou `Low = 101110000...0`.

Na začiatku kódovania inicializujeme premennú *Low* na hodnotu 0. *High* však neinicializujeme na 1 ale na $0,99999\ldots$, keďže pracujeme s otvoreným intervalom. Táto premenná teda bude najvyšším číslom aké môžeme uložiť do našich 63 bitov a teda bude mať hodnotu `0x7FFFFFFFFFFFFFFF`. To zodpovedá jednej nule (preventujúcej pretečenie) nasledovanej 63 jednotkami. Pri výpočtoch s premennou *High* je treba vždy pripočítať hodnotu 1, pretože až to je jej skutočná hodnota zodpovedajúca hornej medzi, ale pred uložením (novej) hodnoty do *High* je nutné túto 1 zase odčítať.

Kóder získava kumulatívnu početnosť spracovávaného symbolu, podľa ktorej ho má zakódovať. Tie mu zabezpečuje model (v tomto prípade pomocou metódy *PPM* popísanej v kapitole 4). Pri určovaní podintervalov intervalu pravdepodobnosti jednotlivých symbolov pre celočíselnú aritmetiku sa využije nasledujúca premenná:

$$stepSize = \frac{(High - Low + 1)}{totalSymbols},$$

kde *totalSymbols* je počet všetkých symbolov rovnako, ako je uvedené vyššie na strane 12. *stepSize* udáva veľkosť dielov na ktoré je rozdelený celý aktuálny interval pravdepodobnosti. Po vynásobení premennými *lowCount(n)*, alebo *highCount(n)* konkrétneho symbolu n je možné určiť podintervaly prevdepodobnosi pre tento symbol.

Pomocou *stepSize* je následne možné vypočítať *Low* a *High* pre nasledujúcu iteráciu cyklu, teda pre spracovanie nasledujúceho symbolu:

$$High = Low + stepSize \times highCount(n) - 1,$$

$$Low = Low + stepSize \times lowCount(n).$$

kde n je práve spracovávaný symbol vstupu. *Low* je nutné upraviť až na koniec, lebo aj *stepSize* aj *High* závisia od jeho hodnoty.

Príklad: Kódujeme reťazec *melee*, pomocou statického modelu. Najskôr zostavíme tabuľku kumulatívnych početností, čím určíme *lowCount(n)* a *highCount(n)* symbolov, viď tabuľka 3.5. Celkový počet symbolov *totalSymbols* je 6, zostáva konštantný, keďže model je statický. Podľa vzorcov sa vypočítajú nové hodnoty *Low*, *High* a *stepSize* pri kódovaní každého symbolu, potom sa prejde sa na symbol nasledujúci (ukážka v tabuľke 3.6).

Symbol	Počet	*lowCount*	*highCount*
e	3	0	3
l	1	3	4
m	1	4	5
eof	1	5	6

Tabuľka 3.5: Kumulatívne početnosti pre reťazec *melee*.

Symbol	*stepSize*	*High*	*Low*
m	0x1555555555555555	0x6AAAAAAAAAAAAAA8	0x5555555555555554
e	0x038E38E38E38E38E	0x5FFFFFFFFFFFFFFD	0x5555555555555554
l	0x01C71C71C71C71C7	0x5C71C71C71C71C6F	0x5AAAAAAAAAAAAAA9
e	0x004BDA12F684BDA1	0x5B8E38E38E38E38B	0x5AAAAAAAAAAAAAA9
e	0x0025ED097B425ED0	0x5B1C71C71C71C719	0x5AAAAAAAAAAAAAA9

Tabuľka 3.6: Výpočet *Low*, *High* a *stepSize* pri kódovaní reťazca *melee*.

3.3.2 Škálovanie pri kódovaní

Použitím popísaných metód na zakódovanie niekoľkých symbolov nastáva problém. Tým je, že hranice intervalu *Low* a *High* konvergujú stále viac až sa budú zhodovať. To spôsobí, že ďalšie kódovanie nebude možné.

Už pri použití aritmetiky s pohyblivou rádovou čiarkou je zrejmé, že jednotlivé hodnoty hraníc intervalu sa po určitom počte opakovaných aplikácií algoritmu aritmetického kódovania líšia len v najnižších rádoch. Toto isté platí aj pri kódovaní s použitím sekvencie bitov. Bity najviac vľavo sa prestávajú meniť.

Preto je možné, ak sa hodnoty najľavejších bitov hraníc *Low* a *High* rovnajú, zapísať ich na výstup. Tieto bity sa už určite nezmenia. Keď sú tieto bity už zapísané, je nutné ich odobrať z premenných *Low* a *High* aby na dalšiu prácu kódera nemali vplyv. Tieto dve premenné teda neudržujú celý kód ale len jeho (poslednú a najviac významovú, aktívnu) časť. Pri postupnom odstraňovaní najviac významových (najľavejších) bitov premenných *Low* a *High* sa na miesta najmenej významových (najpravejších) bitov vkladajú nulové bity u premennej *Low* a jednotkové bity v prípade *High*. Napríklad:

$$\overbrace{0.101\ldots011001011010}^{výstup} \leftarrow \text{zápis 1 na výstup} \leftarrow \overbrace{\mathbf{1}00101\ldots001001}^{Low} \leftarrow \text{doplnenie 0}$$

$$\overbrace{0.101\ldots011001011010}^{výstup} \leftarrow \text{zápis 1 na výstup} \leftarrow \overbrace{\mathbf{1}10001\ldots101100}^{High} \leftarrow \text{doplnenie 1}$$

Ak sa stane, že hodnoty *Low* a *High* sú v rovnakej polovici číselného interavlu, je zaručené, že tento interval nikdy neopustia, lebo dalšie symboly tento interval môžu len zmenšiť. V tomto prípade je polovicou intervalu `0x4000000000000000`, keďže sa používa 64 bitový bezznamienkový typ s obmedzeniami, ktoré boli popísané vyššie, takže sa vlastne využíva len 63 bitov. Obe premenné spadajú do rovnakej polovice intervalu, keď sa líšia v najvýznamnejšom bite, 1 ak sú v hornej polovici intevalu a 0 ak sú v polovici dolnej. Takže, ak sú obe hodnoty *Low* a *High* menšie ako `0x4000000000000000` resp. väčšie alebo rovné `0x4000000000000000` je možné zapísať jeden bit na výstup (najľavejší, v oboch premenných rovnaký), posunúť premenné o jeden bit doľava a doplniť sprava premennej *Low* 0

a premennej *High* 1. Táto metóda sa navýva *E1* (spodná polovica intervalu) resp. *E2 škálovanie* (horná polovica intervalu).

Nasledujúci pseudokód zobrazuje toto škálovanie. Konštanta *HALF* reprezentuje polovicu intervalu s má hodnotou `0x4000000000000000`. Na posun hodnoty doľava sa dá použiť bitový posun doľava (v C++ operátor <<), alebo násobenie dvoma, čo má v dvojkovej sústave rovnaký efekt. Keďže sa v prípade *High* pracuje s otvoreným intervalom, pričíta sa po posune 1.

```
if (High < HALF)              implicitne sa predpokladá, že Low < HALF
  zapíš 0 na výstup;
else if (High >= HALF)        implicitne sa predpokladá, že Low ≤ HALF
     {
       zapíš 1 na výstup;
       Low = Low - HALF;
       High = High - HALF;
     }
posuň Low doľava;
posuň High doľava;
High++;                       otvorený interval, pričítanie 1
```

E1 a *E2* škálovanie však nie sú postačujúce. Neriešia problém, keď *Low* a *High* konvergujú do stredu intervalu, teda zostávajú vo svojich poloviciach intervalu. Dobrým príkladom by bolo keby sa *Low* rovnalo `0x3FFFFFFFFFFFFFFF` a hodnota *High* dosahovala `0x4000000000000000`. Hodnoty sa líšia vo všetkých bitoch okrem najvýznamnejších. Preto nie je možné poslať bit na výstup a previesť bitový posun. Pri ďalších iteráciách by sa k *Low* a *High* pripojovali sprava bity, ktoré by sa stratili a hodnoty by zostali nezmenené. Algoritmus by síce pokračoval až do spracovania *eof*, hodnoty a teda aj výsledný kód by však neboli správne.

Pre riešenie tohoto problému sa používa *E3 škálovanie*. Funguje podobne ako *E1* a *E2*. Na rozdiel od nich ale nepracuje s najľavejšími bitmi ale až s bitmi nasledujúcimi. Sprava sa do rovnako premenných *Low* a *High* vkladajú 0 resp 1. Toto sa môže zopakovať aj viackrát, až kým sa nebudú najvýznamnejšie bity premenných rovnať. Je nutné kontrolovať počet opakovaní. Potom sa môžu tieto bity zapísať na výstup, najvýznamnejší bit bude nasledovaný na výstupe toľkými bitmi, koľko opakovaní nastalo. Avšak, pri zápise najľavejšieho bitu 1 nasleduje zápis nulových bitov rovný počtu opakovaní a naopak. Samozrejme sa premenné posúvajú doľava a sprava vkladajú do premenných 0 resp. 1 v rovnakom počte, ako zapísaných bitov.

Ak si predstavíme konkrétnu sekvenciu bitov hraníc intervalu, pričom sa líšia v najvýznamnejších bitoch. Dostaneme podsekvenciu bitov (súčasť *Low* a *High*) u ktorých sa nedá rozhodnúť kedy ich zapísať na výstup. Rozhodne to zrovnanie menej významových bitov *Low* a *High*, ako je znázornené:

$$\overbrace{0.101\ldots1010}^{výstup}\,\overbrace{011111111}^{nerozhodnuté}\,\overbrace{\mathbf{1}\ldots01101}^{Low}$$
$$\overbrace{0.101\ldots1010}^{výstup}\,\overbrace{100000000}^{nerozhodnuté}\,\overbrace{\mathbf{1}\ldots00110}^{High}$$

V tomto prípade sa najvyššie bity nerozhodnutej sekvencie budú rovnať 1. Takže sa na výstup pošle 1 nasledovaná ôsmimi 0. Sprava sa do *Low* doplní deväť 0 a do *High* deväť 1.

$$\overbrace{0.101\ldots1010}^{výstup} \leftarrow \text{zápis } 100000000 \text{ na výstup} \leftarrow \overbrace{\ldots01101}^{Low} \leftarrow \text{doplnenie } 000000000$$

$$\overbrace{0.101\ldots1010}^{výstup} \leftarrow \text{zápis } 100000000 \text{ na výstup} \leftarrow \overbrace{\ldots01101}^{High} \leftarrow \text{doplnenie } 111111111$$

Pre implementáciu sa musia určiť dve konštanty, a to jedna pre prvú štvrtinu intervalu a druhá pre tretiu štvrtinu intervalu. Pomocou týchto konštánt budeme určovať, či *Low* a *High* konvergujú do stredu intervalu. V tomto prípade sa využíva 64 bitový bezznamienkový typ, takže tieto konštanty budú: *QUARTER1* = `0x2000000000000000` pre hranicu prvej štvrtiny a *QUARTER3* = `0x6000000000000000` pre hranicu štvrtiny tretej. Rovnako ako pri konštatne *HALF* sa využije len 63 bitov aby sa zabránilo pretečeniu.

Ako bolo spomenuté, ku škálovaniu *E3* dochádza ak hodnoty hraníc intervalu konvergujú k jeho stredu. Tomu zodpovedajú predpoklady $Low \geq QUARTER1$ $High < QUARTER3$. Intervaly sa naďalej môžu len zmenšovať, takže konvergencia k stredu intervalu je istá. Algoritmus škálovania *E3* však priamo nezapisuje bity na výstup, ale len posúva ľavé bity hodnôt *Low* a *High* doľava, pričom najvýznamnejší bit ale zostáva nezmenený. Robí to tak, že si pamätá počet prípadov, kedy by prišlo k tomuto posunu. Až po najbližšom *E1* alebo *E2* škálovaní odošle na výstup toľko bitov koľko opakovaní nastalo a nahradí ich rovnakým počtom 0 príp. 1 v závislosti od toho, či sa jedná o *Low* alebo *High*. Rovnako ako pri predošlích škálovacích algoritmoch sa na výstup odosielajú bity s negovanou hodnotou ako bit ktorý zapíše *E1* alebo *E2*. Premenná *scaleCount* udržuje počet opakovaní pri škálovaní *E3* a je inicializovaná na nulu. Celkový pseudokód pre škálovanie bude vyzerať takto:

```
while(1)                                                  aplikuj škálovanie
{
  if (High < HALF)                                        zároveň Low < HALF
  {                                                       E1
    zapíš 0 na výstup;
    for(; scaleCount > 0; scaleCount--)                   pokiaľ ScaleCount > 0
      zapíš 1 na výstup;                                  E3
  }
  else if (High >= HALF)                                  zároveň High ≤ HALF
       {                                                  E2
         zapíš 1 na výstup;
         Low = Low - HALF;
         High = High - HALF;
         for(; scaleCount > 0; scaleCount--)              pokiaľ ScaleCount > 0
           zapíš 0 na výstup;                             E3
       }
       else if (Low >= QUARTER1 and High < Quarter3)      konvergencia do stredu
            {
              scaleCount++;                               zvýš počet opakovaní
              Low = Low - QUARTER1;
              High = High - QUARTER1;
            }
       else
         break;                                           ukonči škálovanie
posuň Low doľava;
```

```
posuň High doľava;
High++;                                                    otvorený interval
}
```

Pri každej iterácii cyklu sa môže použiť práve jeden zo škálovacích algoritmov. Ak sa však nenachádzajú sučasne *Low* a *High* v jednej z polovíc intervalu, alebo nekonvergujú do jeho stredu ku škálovaniu nedôjde. V takom prípade sa prejde k spracovávaniu ďalšieho vstupného symbolu kódera.

3.3.3 Škálovanie pri dekódovaní

Dekóder pracuje pri zužovaní intervalu rovnako ako kóder, to isté platí aj o škálovaní. Avšak dekóder musí upravovať ešte aj zatiaľ nespracovanú časť zakódovanej sekvencie. Na uloženie tejto časti kódu sa zvlášť zvlášť premenná *Buffer*. S touto premennou sa pracuje pri škálovaní rovnako ako s *Low* a *High*, takže sa jej obsah posúva pri *E1* a *E2* o jeden bit doľava, pri *E3* zostáva najvýznamnejší bit nezmenený a posúvajú sa až bity nasledujúce. Po posune sa na uvoľnený bit sprava doplní další bit zo zakódovanej sekvencie. Ak sa prečíta celá táto sekvencia, dopĺňajú sa sprava nuly.

Nasledujúci pseudokód znázorňuje ako vyzerá škálovanie u dekódera. Ako je vidno je veľmi podobný škálovaniu pri kódovaní. Je súčasťou dekódovania a ak sa cyklus ukončí, škálovanie je dokončené a dekóder prechádza k spracovaniu dalšieho symbolu.

```
while(1)                                                   aplikuj škálovanie
{
  if (High < HALF)                                         zároveň Low < HALF
    ;                                                      E1, bez úprav
  else if (High >= HALF)                                   zároveň High ≤ HALF
       {                                                   E2
         Low = Low - HALF;
         High = High - HALF;
         Buffer = Buffer - HALF;
       }
       else if (Low >= QUARTER1 and High < Quarter3)       konvergencia do stredu
            {                                              E3
              Low = Low - QUARTER1;
              High = High - QUARTER1;
              Buffer = Buffer - QUARTER1;
            }
       else
         break;                                            ukonči škálovanie
posuň Low doľava;
posuň High doľava;
posuň Buffer doľava;
High++;                                                    otvorený interval
vlož zprava do Value ďalší bit z kódu;
}
```

3.3.4 Ukončenie kódovania

Ako sa hodnoty *Low* a *High* kódovaním stále zužujú, generuje sa postupne výstupný kód. Je ho však nutné správne ukončiť. Ukončiť ho tak, aby posledné bity ležali vo vnútri konečného intervalu ohraničeného konečnými hodnotami *Low* a *High*. Tieto hodnoty spadajú do nasledujúcich intervalov:

- $Low < QUARTER1 < HALF \leq High$
- $low < HALF < QUARTER3 \leq high$

Tento interval obsahuje vždy aspoň jednu celú štvrtinu. Je možné, že aj viac, čo ale nie je zaručené. Môžeme teda použiť túto hodnotu na určenie spodnej medze.

- Druhá štvrtina

  $$Low < QUARTER1 < HALF \leq High$$

  Uloží sa 0 nasledovaná 1, čo znamená že ukladáme nižšiu polovicun nasledovanú vyššou. Dekóder na koniec spodnej hranice dopĺňa nuly, zodpovedá to spodnej hranici druhej štvrtiny. Ak nastane situácia, že nebolo doriešené *E3* škálovanie, je nutné zapísať jednotkové bity, ako si to toto škálovanie žiada. V kombiácii s jednotkou, ktorú je nutné zapísať, môže sa škálovať *E3* s jedným opakovaním navyše.

- Tretia štvrtina

  $$low < HALF < QUARTER3 \leq high$$

  Pri úvahe rovnako ako v predošlom prípade sa príde k záveru, že treba na výstup zapísať 1 nasledovanú 0 a v prípade *E3* ďalšími nulovými bitmi, ale keďže nie sa doplňujú automaticky, stačí zapísať len tento jednotkový bit.

Kapitola 4

PPM

PPM je skratkou od anglického *Prediction by Partial Matching*, čo v preklade znamená predikcia čiastočnej zhody. Je to štatistická bezstratová metóda silno sa opierajúca o kontext vstupých dát. *PPM* modeluje dáta a určuje pravdepodobnosti vstupných symbolov, ktoré odovzdáva aritmetickému kóderu.

Metódu vyvinuli J. G. Cleary a I. H. Witten [4] v roku 1984. Metóda je veľmi náročná na pamäť a tak jej prvé implementácie pochádzaju až zo začiatku 90. rokov, kedy ju vylepšil a implementoval A. Moffat [7]. Dnes sa táto metóda stále rozširuje a vyvíja až vzniklo niekoľko obmien algorimtu a vznikli nové varianty (napr *PPMA*, *PPMB*, či *PPMC*, ktoré budú bližšie popísané neskôr).

Ako uvádza [11], *kontext symbolu s*, z pohľadu využitia pri modelovaní dát pre kompresiu, je vlastne tvorený niekoľkými n symbolmi, ktoré ho predchádzajú. Podľa toho sa hovorí o *kontexte n-tého rádu*. Inak povedané metóda používa *Markovský model rádu n*. Najjednoduchší model je model nultého rádu t.j. taký, ktorý k predikcii nevyužíva žiaden predchádzajúci vstupný symbol. Kompresná metóda založená na kontexte používa kontext symbolu k jeho predikcii. Predikcia sa teda chápe ako určenie pravdepodobnosti výskytu daného symbolu.

Kontextovo orientované modelovacie metódy uvažujú dva prístupy:

- *Statický model* je taký, ktorý využíva vždy rovnaké pravdepodobnosti. Obsahuje tabuľky, ktoré sú statické a obsahujú pravdepodobnosti dvojíc, alebo trojíc abecedy. Tieto tabuľky používa k priradeniu pravdepodobnosti nasledujúcemu symbolu s v závislosti na symbole, resp. kontexte c ktorý ho predchádza. Tabuľky sa vytvárajú z rozsiahlych textov (príp. iných vhodných dát) a prideľujú sa pravdepodobnosti dvoj, či trojznakovým zhlukom. Tento model je pomerne jednoduchý a dosahuje celkom dobré výsledky, má však dva hlavné problémy. Jedným je, že vstupné dáta môžu byť veľmi odlišné od dát, ktoré poslúžili na vytvorenie pravdepodobnostnej tabuľky. A druhým problémom sú nulové pravdepodobnosti. Niektoré zhluky sa nemusia vyskytnúť vôbec, čomu zodpovedá nulová prevdepodobnosť, ale artimetický kóder žiada na vstupe symboly s nenulovou pravdepodobnosťou. Tieto problémy sa dajú odstrániť dvoma spôsobmi:

  1. Po analýze veľkého množstva dát a čítaní frekvencií sa prejde tabuľka frekvencií a vyhľadajú sa prázdne bunky. Každej prázdnej bunke se pridelí hodnota 1 a celkový počet se tiež zvýši o 1. Tým sa zabezpečí, že sa každý dvojznakový alebo trojznakový zhluk vyskytol aspoň raz.

2. Celkový počet sa zvýši o 1 a táto 1 sa rozdelí medzi všetky prázdne bunky v tabuľke. Každá dostane veľmi nízku hodnotu pravdepodobnosti. Tým sa pridelí veľmi malá pravdepodobnosť každému zhluku, ktorý sa pri analýze modelovacích dát nevyskytol.

- *Adaptívny model* rovnako používa tabuľku pravdepodobnosti dvoj, troj, alebo aj viacznakových zhlukov. Rovnako túto tabuľku používa na určenie pravdepodobnosti symbolu s podľa kontextu c, ktorý ho predchádza. Tabuľka pravdepodobností sa ale počas spracovania symbolov upravuje a to v závislosti na práve kódovaných dátach. Takéto modelovanie je zložité a pomalé, ale podáva veľmi dobré výsledky z pohľadu kompresie a to aj napriek tomu, keby sa rozdelenie pradvepodobnosti dát na vstupe veľmi líšilo od priemeru.

4.1 Princíp PPM

Vysvetlenie činnosti *PPM*, ako ho podáva [11]: *PPM* využíva poznatok, že ak využije kontext určitého rádu na spracovávaný symbol s tým, že vedie k určeniu pravdepodobnosti rovnej 0, prepne na kontext o rád nižší a znova sa snaží určiť pravdepodobnosť toho istého symbolu. Začína s kontextom rádu n. Hľadá v jeho dátovej štruktúre predchádzajúci výskyt kontextu c, nasledovaného symbolom s. Ak taký kontext nenájde prepne na rád $n-1$ a pokúsi sa o to isté. Ak c' je reťazec, tvorený $n-1$ symbolmi c najviac vpravo. Kóder vyhľadá v jeho dátovej štruktúre predchádzajúci výskyt súčasného kontextu c' nasledovaný symbolom s. Metóda sa teda snaží používať menšiu časť kontextu c.

Príklad: Tento príklad ukazuje vyššie popísanú činnosť, je prebraný z [11]. Majme kontextovo orientovaný kóder používajúci kontext 3. rádu. Behom komprimovania sa niekoľkokrát vyskytne slovo *here*, ale slovo *there* sa našlo len prvýkrát. Predpokladajme, že nasledujúci symbol je r v slove *there*. Kóder nenájde žiadné výskyty kontextu *the* 3. rádu nasledované r (takže v tomto kontexte má r pravdepodobnosť rovnú 0). Kóder môže zapísať do výstupneho toku r, čo ale nedáva žiadnu komprimáciu, ale vie, že r sa vyskytlo v minulosti niekoľkokrát za kontextom 2. rádu *he* (pre tento kontext je pravdepodobnosť r nenulová).

4.2 Prieheh kódovania

Adaptívny na kontexte založený model rádu n číta vstupný symbol s a uvažuje n bezprostredne predchádzajúcich symbolov, ktoré tvoria jeho kontext c n-tého rádu. Model potom odhadne pravdepodobnosť p toho, že symbol s sa objaví vo vstupných dátach nasledujúc kontext c. Potom kóder zavolá algoritmus adaptívneho aritmetického kódovania, ktorý zakóduje symbol s s pravdepodobnosťou p.

Rád kontextu stále klesá, až kým nie je rovný 0. V tom prípade sa zisťuje, či sa vlastne daný symbol na vstupe niekedy objavil. Ak nie, prepne sa na kontext -1. Symbolu je priradená penvná pravdepodobnosť $1/|A|$, kde A je abeceda a teda $|A|$ jej veľkosť. Na začiaktu je pravdepodobnosť p daného symbolu vysoká a postupne sa znižuje. Po určení pravdepodobnosti *PPM* odovzdá túto pravdepodobnosť aritmetickému kóderu, ktorý dáta zakóduje. Kóder prijíma údaje od *PPM* v podobe *lowCount(s)*, *highCount(s)* a *highCount(totalCount)*, ako je vysvetlené v podkapitole 3.3.1. Potom prejde *PPM* na ďalší vstupný symbol $s+1$ a nastaví kontext späť na rád n.

Príklad: [11] Majme reťazec 11 znakov *xyzzxyxyzzx*. Uvažujme rády kontextu 0 až 4. Tabuľka 4.1 ukazuje jednotlivé kontexty, ktoré sa vyskytli spolu aj s ich početnosťou. Ďalej

Rád 4		**Rád 3**		**Rád 2**		**Rád 1**		**Rád 0**	
kontext	*f*	*kontext*	*f*	*kontext*	*f*	*kontext*	*f*	*kontext*	*f*
$xyzz \rightarrow x$	2	$xyz \rightarrow z$	2	$xy \rightarrow z$	2	$x \rightarrow y$	3	x	4
$yzzx \rightarrow y$	1	$yzz \rightarrow x$	2	$\rightarrow x$	1	$y \rightarrow z$	2	y	3
$zzxy \rightarrow x$	1	$zzx \rightarrow y$	1	$yz \rightarrow z$	2	$\rightarrow x$	1	z	4
$zxyx \rightarrow y$	1	$zxy \rightarrow x$	1	$zz \rightarrow x$	2	$z \rightarrow z$	2		
$xyxy \rightarrow z$	1	$xyx \rightarrow y$	1	$zx \rightarrow y$	1	$\rightarrow x$	2		
$yxyz \rightarrow x$	1	$yxy \rightarrow z$	1	$yx \rightarrow y$	1				

Tabuľka 4.1: Kontexty pre reťazec *xyzzxyxyzzx*.

uvažujme, že nasledujúcim symbolom na vstupe je *z*. Kontext štvrtého rádu je *yzxx*, ktorý sa už v minulosti vyskytol nasledovaný symbolom *y*, zatiaľ však nie *z*. Kóder preto prepne na rád 3, čo je *zzx*, ale ten ešte nebol videný nasledovaný *z*. Prechádza sa na rád 2, teda kontext *zx*. Ten sa však tiež nevyskytol nasledovaný *y*. Prechádza sa teda na kontext 1, *x*. Tento kontext sa však vyskytol len pred *y*, nie však *z*. Prechádza sa na konetxt 0, kde má *z* frekvenciu 4. Takže sa určí pravdepodobnosť dvanásteho symbolu *n*, čo je 4/11. Nakoniec *PPM* upraví tabuľku, pridaním nových kontextov a úpravou početností (viď tabuľka 4.2).

Rád 4		**Rád 3**		**Rád 2**		**Rád 1**		**Rád 0**	
kontext	*f*	*kontext*	*f*	*kontext*	*f*	*kontext*	*f*	*kontext*	*f*
$xyzz \rightarrow x$	2	$xyz \rightarrow z$	2	$xy \rightarrow z$	2	$x \rightarrow y$	3	x	4
$yzzx \rightarrow y$	1	$yzz \rightarrow x$	2	$\rightarrow x$	1	$\rightarrow z$	1	y	3
$\rightarrow x$	1	$zzx \rightarrow y$	1	$yz \rightarrow z$	2	$y \rightarrow z$	2	z	5
$zzxy \rightarrow x$	1	$\rightarrow z$	1	$zz \rightarrow x$	2	$\rightarrow z$	2		
$zxyx \rightarrow y$	1	$zxy \rightarrow x$	1	$zx \rightarrow y$	1	$z \rightarrow z$	2		
$xyxy \rightarrow z$	1	$xyx \rightarrow y$	1	$\rightarrow z$	1	$\rightarrow x$	2		
$yxyz \rightarrow x$	1	$yxy \rightarrow z$	1	$yx \rightarrow y$	1				

Tabuľka 4.2: Kontexty pre reťazec *xyzzxyxyzzxz*.

4.3 Prieheh dekódovania

Základný rozdiel činnosti pri dekódovaní je to, že kóder sa môže pozrieť na nasledujúci symbol a upraviť ďalší krok podľa neho, ale dekóder musí zistiť aký je tento nasledujúci symbol. Podľa toho aký je tento nasledujúci symbol môže kóder prepnúť na nižší rád kontextu. Dekóder to však nedokáže, keďže ten nasledujúci symbol nepozná. Preto musí byť kóder s dekóderom previazaný. To sa zaistí tým, že sa vyhradí jeden symbol abecedy pre *symbol zmeny*, alebo tiež nazývaný *escape symbol*.

Takže, keď sa kóder rozhodne prepnúť na kratší kontext, zapíše na výstup *escape symbol* (samozrejme zakódovaný). Dekóder potom pri dekompresii môže tento symbol rozpoznať, keďže je zakódovaný v súčasnom kontexte. Po jeho dekódovaní vie, že má prepnúť na kratší kontext.

Môže sa však stať, že kóder n-tého rádu narazí na symbol s, ktorý sa nikdy pred tým nevyskytol. Toto sa stáva často hlavne na začiatku kódovania. Symbol sa nevyskytol v žiadnom kontexte, ani v kontexte rádu 0, z čoho plynie, že sa doteraz nevyskytol na vstupe vôbec. Vtedy kóder postupne prejde z rádu n na rád -1 a pošle na výstup $n+1$ *escape symbolov*, a kedže je v ráde -1 nasleduje za nimi symbol s zakódovaný s pevnou pravdepodobnosťou $1/|A|$, kde $|A|$ je veľkosť abecedy.

Keďže *escape symbol* sa kóduje hlavne na začiatku procesu pomerne často, je dobré mu prideliť vhodnú pravdepodobnosť. Mala by byť spočiatku vysoká, keďže bude dochádzať k prepínaniu kontextu častejšie. Postupne bude pravdepodobnosť *escape symbolu* klesať, lebo bude prichádzať na vstup väčšie množstvo vstupných symbolov.

Príklad: Nasledujúca tabuľka 4.3 ukazuje kontexty do rádu 2 pre 11 znakový reťazec *xyzzxyxyzzx*. Každý kontext je umiestnený vo zvláštnej skupine spolu s *escape symbolom*. Kontext *yz* sa objavil na vstupe dva krát, vždy nasledovaný symbolom *z*. Jeho početnosť je teda 2 a je v skupine s *escape symbolom* s frekvenciou 1. Pravdepodobnosť *yz* s *escape*

Rád 2			**Rád 1**			**Rád 0**		
kontext	f	p	*kontext*	f	p	*kontext*	f	p
$xy \rightarrow z$	2	2/5	$x \rightarrow y$	3	3/4	x	4	2/7
$\rightarrow x$	1	1/5	*escape*	1	1/4	y	3	3/14
escape	2	2/5				z	4	2/7
			$y \rightarrow z$	2	2/5	*escape*	3	3/14
$yz \rightarrow z$	2	2/3	$\rightarrow x$	1	1/5			
escape	1	1/3	*escape*	2	2/5			
$zz \rightarrow x$	2	2/3	$z \rightarrow z$	2	1/3			
escape	1	1/3	$\rightarrow x$	2	1/3			
			escape	2	1/3			
$zx \rightarrow y$	1	1/2						
escape	1	1/2						
$yx \rightarrow y$	1	1/2						
escape	1	1/2						

Tabuľka 4.3: Kontexty s *escape symbolmi* pre reťazec *xyzzxyxyzzx*.

je 2/3 a 1/3. Kontext *xy* sa vyskytol trikrát. Raz bol nasledovaný x a dva krát symbolom z. Sú umiestnené v skupine s *escape symbolom* s početnosťou 2, kedže je v skupine s dvoma členmi. Pravdepodobnosť týchto členov je po rade 2/5, 1/5 a 2/5.

Táto metóda prideľovania prevdepodobnosti je založená na predpoklade, že ak bol konkrétny kontext c nájdený niekoľkokrát a vždy bol nasledovaný konkrétnym symbolom s, bude aj pri najbližšom výskyte kontextu znova nasledovať s. V tomto prípade bude musieť kóder málokedy znížiť rád kontextu. Tým *escape symbol* môže nadobudnúť malú pravdepodobnosť. Ale ak sa tento kontext c vyskytol vždy nasledovaný iným symbolom s_1, s_2 až s_n je veľká šanca, že kontext sa bude musieť znížiť, lebo je pradvepodobné, že nasledujúci symbol bude tiež iný, s_x. Na tomto princípe funguje varianta *PPM* tzv. *PPMC*.

Príklad: Majme reťazec *xyzzxyxyzzx*. Kontexty rádu 0 až 2 zistené kóderom sú uvedené v tabuľke 4.3. Reťazec bol zakódovaný metódou *PPMC*, aktuálny kontext je *zz* a kóder

číta ďalší symbol. Uvažujme abecedu 36 znakov (26 znakov malej abecedy a číslice 0 až 9), *escape symbol* a symbol *eof*. Veľkosť abecedy $|A| = 38$. V závislosti na aktuálnom symbole na vstupe môžu nastať tieto štyri prípady:

1. symbol x
   Kóder zistí, že kontext zz už bol nasledovaný x a pridelí mu pravdepodobnosť $2/3$. x sa potom aritmetickým kódovaním zakóduje s touto pravdepodobnosťou, k tomu bude treba približne $0,58$ bitu, keďže $-\log_2(2/3) \doteq 0,58$ (viď kapitola 3 a podkapitola 2.4.2).

2. symbol y
   z zatiaľ nenasledoval kontext zz. Aritmetickému kóderu sa odošle *escape symbol* s pravdepodobnosťou $1/3$ jeho kód zaberie asi $1,58$ bitu. Kontext rádu 1 však tiež nenájde z nasledované y. Znovu zašle aritmetickému kóderu *escape symbol*, s pravdepodobnosťou $1/3$, ako určuje kontext z. Tento kód zaberie približne $1,58$ bitov. Kóder prepína na kontext rádu 0. Ten určí, že y má pravdepodobnosť $3/14$ a aritmetický kóder ho zakóduje na $-\log_2(3/14) \doteq 2,22$ bitov. Takže celkovo sa na zakódovanie spotrebuje približne $1,58 + 1,58 + 2,12 = 5,38$ bitov.

3. symbol z
   Kontext zz zatiaľ nebol nasledovaný z. Rád kontextu sa teda zníži, takže sa artimetickému kóderu zašle *escape symbol* s pravdepodobnosťou $1/3$. Táto pravdepodobnosť je určená skupinou $zz \rightarrow$, v ktorej sa nachádza tento *escape symbol* a na jeho zakódovanie sa využije $-\log_2(1/3) \doteq 1,58$ bitov. V kontexte 1 sa zistí, že kontextu z nasledovanému symbolom z je priradená pravdepodobnosť $1/3$. z sa odošle aritmetickému kóderu. Na jeho zakódovanie sa využije $-\log_2(1/3) \doteq 1,58$, spolu s *escape symbolom* zaberie kód symbolu z približne $1,58 + 1,58 = 3,16$ bitov.

4. symbol w, alebo iný symbol, ktorý sa na vstupe ešte nevyskytol
   Rovnako ako v prípade 2 prepne kóder kontext postupne z rádu 2 na 0 a pošle aritmetickému kóderu dva *symboly escape*, s ich pravdepodobnosťami. V ráde kontextu 0 symbol w však nie je, lebo sa ešte na vstupe nikdy neobjavil. Kóder prepína na kontext rádu -1. Pritom vyšle *escape symbol* s pravdepodobnosťou $3/14$, ako určuje kontext rádu 0. Tým sa k výstupu pripojí približne $2,22$ bitov. w sa odošle aritmetickému kóderu s pravdepodobnosťou $1/|A| = 1/38$, čo zaberie $-\log_2(1/38) \doteq 5,24$ bitov. Celkovo sa na zakódovanie použije $1,58 + 1,58 + 2,22 + 4,80 = 9,89$ bitov.

Keďže abeceda má veľkosť 36, je potrebné na uloženie 1 znaku bez kompresie 6 bitov. V kroku 4 však je potrebné na uloženie znaku, ktorý sa ešte nevyskytol až $9,89$ bitov, čo je omnoho viac ako pri ukladnaí bez akejkoľvek kompresie.

4.4 Vylúčenie

Pri znižovaní rádu kontextu c z dĺžky n na $n-1$ môže kóder využiť poznatky, ktoré získal v ráde n k vylúčeniu prípadov, ktoré nemôžu v ráde $n-1$ nastať. Toto zvýši presnosť určovania pravdepodobnosti v ráde n, a tým zlepšuje kompresiu [11].

Príklad: Vrátime sa k reťazcu *xyzzxyxyzzx*. Tento reťazec je už zakódovaný pomocou *PPMC* a ďalším symbolom na vstupe je symbol z. Pri jeho spracovávaní sa najskôr kóder

zaoberá najdlhším kontextom, ktorý má dĺžku 2. Za aktuálny kontext považujme zz. Z tabuľky 4.3, vytvorenej pri kódovaní, vyberieme len dôležité kontexty, tie ktoré bude kóder pri kódovaní z na vstupe skúmať (viď tabuľka 4.4). V kontexte rádu 2 kóder zistí, že zz

Rád 2			**Rád 1**			**Rád 0**		
kontext	f	p	*kontext*	f	p	*kontext*	f	p
$zz \rightarrow x$	2	2/3	$z \rightarrow z$	2	1/3	x	4	2/7
escape	1	1/3	$\rightarrow x$	2	1/3	y	3	3/14
			escape	2	1/3	z	4	2/7
						escape	3	3/14

Tabuľka 4.4: Skúmané kontexty pre vstupný symbol z.

sa už vyskytol. Nebol však nasledovaný symbolom z. Preto vyšle *escape symbol* a zníži kontext na dĺžku 1. Zisťuje sa, či sa kontext z vyskytol pred symbolom z. Odpoveď je, že sa vyskytol s početnosťou 2. Ale to, že kontext zz sa vyskytol nasledovaný x poukazuje na skutočnosť, že kódovaným symbolom nemôže byť x, lebo by bol zakódovaný už v kontexte rádu 2. Kóder teda môže vylúčiť prípad, že kontext z bude nasledovaný symbolom x. V tabuľke pravdepodobností tak nie je nutné udržovať pre tento prípad priestor. To znižuje celkovú početnosť skupiny $z \rightarrow$ rádu kontextu 1 z 6 na 4. Tým sa zvýši pravdepodobnosť symbolu z z 1/3 na 1/2. Potom bude možné zakódovať symbol z na $-\log_2(1/2) = 1$ bit miesto pôvodných $-\log_2(1/3) = 1,58$ bitov.

4.5 Varianty PPM

PPM používa k označeniu miesta kde došlo k prepnutiu na nižší rád kontextu *escape symboly*. Ako už bolo spomenuté tým zaručuje synchronizáciu kódera a dekódera. Metódy sa väčšinou líšia v spôsobe akým prideľujú pravdepodobnosti *escape symbolom*. Najskôr je však dobré uvedomiť si niektoré skutočnosti o kontexte samotnom.

PPM model n-tého rádu číta vstupný symbol s a uvažuje n za sebou nasledujúcich znakov, nachádzajúcich sa pred symbolom s, ktoré vytvárajú kontext c tohoto symbolu. Model sa snaží z predchádzajúceho priebehu kompresie určiť symbolu s pravdepodobnosť p, s ktorou sa symbol s objaví na vstupe za kontextom c. Z toho teoreticky vyplýva, že čím je rád kontextu vyšší, tým presnejšia je predikcia pravdepodobnosti p. Veľký rád je sídce schopný určovať pravdepodobnosti s veľmi veľkou presnosťou, ale má niekoľko nevýhod:

- Ak bude n veľmi veľké číslo, je asi nemožné, že veľká skupina n symbolov, napríklad niekoľko desatok tisíc, tvoriaca konetxt by by bola viac krát nasledovaná stým istým symbolom s na vstupe.
- Ako bolo spomenuté, v prípade, že sa vstupný symbol s ešte nevyskytol prechádza kóder postupne do rádu -1 a zapisuje na výstup $n+1$ *escape symbolov*. Pri predstave, že $n = 10000$, zapíše na výstup 10001 *escape symbolov*.
- Pre veľké n existuje veľmi veľké množstvo kontextov. Tento počet exponenciálne rastie s n. Pre abecedu A s veľkosťou $|A|$ existuje $|A|^n$ možných kontextov.

V praxi sa používajú relatívne krátke kontexty. Rozmedzia sú zvyčajne od 2 do 10. Variánt *PPM* je mnoho, tak sú tu spomenuté len tie najbežnejšie.

4.5.1 PPMC

Princípy, ktoré používa táto varianta sú popísané vyššie. Prideľuje *escape symbolom* početnosti v závislosti od skupiny kontextov, v ktorých sa nachádzajú. Keď je n početnosť symbolov v danej skupine, pravdepodobnosť p akéhokoľvek symbolu (teda aj *escape symbolu*) v danej skupine je $p = \frac{f}{n+1}$. Tejto variante sa venuje aj implementácia tejto práce.

4.5.2 PPMA

Táto varianta pracuje podobne ako *PPMC*. Každý kontext je umietnený v skupine spolu so svojim *escape symbolom* (viď tabuľka 4.3). Početnosť symbolov n (v tomto prípade bez *escape symbolu*) v danej skupine určuje pradvepodobnosť symbolu $p = \frac{f}{n+1}$. Pravdepodobnosť p *escape symbolu* je však určená $p = \frac{1}{n+1}$, čo je vlastne ako by mal *escape symbol* vždy početnosť 1.

4.5.3 PPMB

Je tiež podobná *PPMC*. Rozdiel je v tom, že symbolu s za kontextom c sa priradí pravdepodobnosť až vtedy, keď sa už vyskytol aspoň dvakrát. To dosahuje zmenšenie všetkých početností o 1. Ak f je početnosť jedného symbolu s a n je početnosť všetkých symbolov v skupine, okrem *escape symbolu*, symboly skupiny majú pravdepodobnosť $p = \frac{f-1}{n}$ a *escape symbol* tejto skupiny má pravdepodobnosť $p = \frac{k}{n}$, kde k je počet členov skupiny (okrem samotného *escape symbolu*).

4.5.4 PPM*

Metódy *PPM* majú vlastnosť, že používajú vždy kontext obmedzeného a pevného rádu. Vždy začína s najdlhším kontextom c nasledovaným symbolom s. V prípade nezhody prechádza do nižšieho rádu. Metóda *PPM** sa však snaží zvyšovať hodnotu rádu kontxtu n do nekonečna. Z dôvodov popísaných vyššie to zvyšuje presnosť určenia pravdepodobnosti a metóda má lepší kompresný pomer.

4.5.5 PPMZ

Táto varianta vychádza z pôvodnej implementácie *PPM*. Sústreďuje sa na jej nevhodné javy ako sú špatné ošetrenie nedeterministických kontextov, slabá zvládnuteľnosť veľkého objemu dát, kontexty s neobmedzenou dĺžkou a lokálna estimácia rádu. *PPMZ* zameriava práve na odstránenie týchto problémov a dosahuje veľmi dobré výsledky.

4.5.6 Iné varianty

Ďalšie varianty metódy sú napríklad *PPMP*, či *PPMX*. Tieto metódy už ale používajú úplne odlišný spôsob prideľovania pravdepodobnosti a sú nad rámec tejto práce. Napriek tomu je dobré spomenúť, že napríklad *PPMP* pracuje s výskytmi symbolov ako s oddelenými Poissonvoskými procesmi. čitateľ sa o nich môže dozvedieť viac z [11].

4.6 Implementácia PPM

Najdôležitejší pri implementácií metódy *PPM* je návrh a údržba dátovej štruktúry, v ktorej by boli uložené všetky kontexty rádu 0 až n všetkých symbolov prečítaných zo vstupného

toku. V tejto je štruktúre nevynutné mať možnosť rýchleho prehľadávania. Zárovaň je nutné aby bola ľahko upravovateľná s každým nasledujúcim vstupným symbolom, aby sa zaručila adaptívnosť *PPM*. Implementácia vychádza predovšetkým z [3] a [11].

4.6.1 Trie

Náročné podmienky implementácie spĺňa a na tieo účely sa využíva dátová štruktúra *trie*. Jej názov pochádza z anglického slova re*trie*val. Popis tejto štruktúry je možne nájsť aj v [11]. *Trie* je vlastne strom, ktorý sa vetví s každým symbolom kontextu. Na každej úrovni je štruktúra vetvenia určená len časťou dátovej položky, nie položkou celou. Táto dátová položka stromu obsahuje 2 údaje, tými sú znak kontextu a početnosť kontextu. V *PPM* v sebe obsahuje každý kontext kontexty nižších rádov. Takže môže každý kontext do stromu prispieť len jedným symbolom.

Táto stromová štruktúra rastie do šírky, do hĺbky však nie. Maximálna hĺbka, ktorú môže *trie* dosiahnuť je $n + 1$, kde *trie* je rád kontextu. Čítaním dát zo vstupu strom ďalej rasie do šírky, nie však konštantnou rýchlosťou. V závislosti na vstupnom symbole sa niekedy môže pridať viac uzlov, niekedy žiadny.

Ďalšou dátovou zložkou je početnosť kontextu. Toto číslo pri symbole udáva konkrétny počet výskytov daného symbolu. Číslo pri symbole listu zas udáva, koľkokrát sa za kontextom vyskytol ďalší symbol.

Konštrukcia *trie* je znázornená na nasledujúcom príklade.

Príklad: Máme reťazec *zxzyzxxy*. Obrázok 4.1 zobrazuje *trie* pre kontexty do rádu 2. Ako je vidno 1. úroveň *trie* obsahuje uzly pre každý zatiaľ prečítaný symbol. Toto sú kontexty rádu 1. Úroveň *trie* 2 obsahuje kontexty rádu 1 atď. Prechodom *trie* z koreňa smerom dolu do jedného z listov sa získa konkrétny kontext. Teda ak prejdeme v 4.1.4 od koreňa cez *xz* ku listu *y*, dostávame *xzy*, čo je konkrétny kontext *xz* nasledovaný symbolom *y*. V 4.1.8 je v *trie* celkom 7 kontextov.

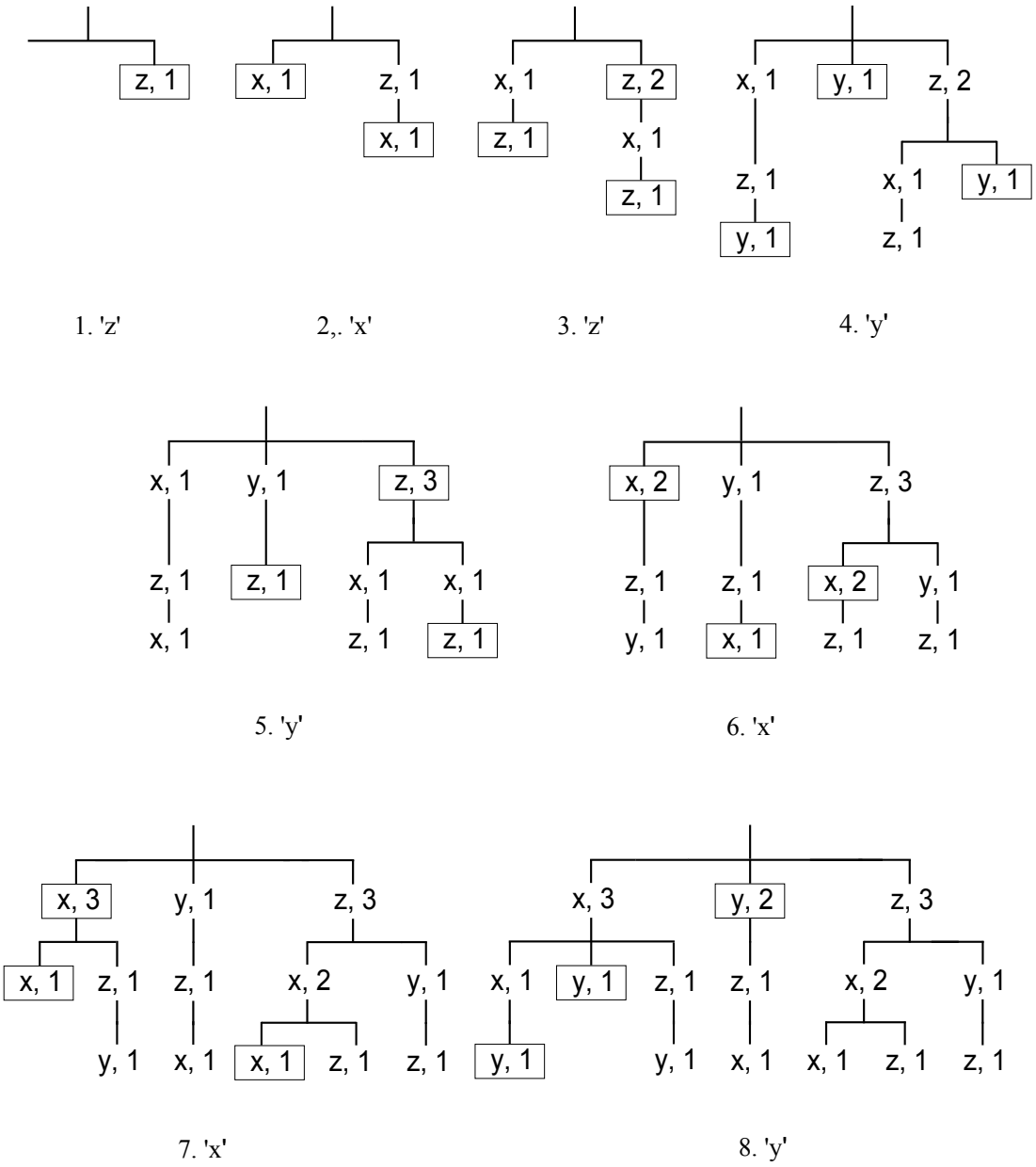
Pozrime sa na uzol *x, 3* v ľavej vetve obrázku 4.1.8. Číslo 3 znamená, že tento symbol bol videný 3 krát. Listy tohoto uzla *x, 1*, *y, 1* a *z, 1* zas určujú, že tieto 3 výskyty *x* boli v jednom prípade nasledované *x*, v jednom prípade *y* a jednom prípade symbolom *z*.

Zarámované uzly v jednotlivých obrázkoch označujú tie uzly, ktoré menia rád kontextu posledného pridaného symbolu k *trie*. Keď sa pozrieme na 4.1.3, kde sa na vstupe spracováva symbol *z*, vidíme, že *z* bol pridané dvakrát, a to v ľavej vetve ako list *x, 1* a v pravej ako list *x, 1*. To znázorňuje, že *z* nasleduje za kontextami *x* a *zx*. Rovnako však bola upravená početnosť v pravej vetve na *z, 2*.

Ako je zrejmé z príkladu, v každom kroku konštrukcie sa upravuje $n + 1$ uzlov stromu (v každel hĺbke jeden), okrem niekoľkých prvých krokov, kým strom nedosiahol plneú výšku. Niektoré z týchto uzlov sú buď nové, alebo sú to uzly, kde sa zmenila početnosť.

Tým, že sa uzly môžu so vstupným symbolom buď pridať alebo sa môže manipulovať s ich dátovým obsahom, (početnosťou symbolu v konkrétnom uzle) vzniká problém, ktorým je, že algoritmus nevie rozpoznať ktorú z týchto operácií má vykonať. Toto sa dá vyriešiť pridaním tzv. *vine pointer* (*plazivého ukazovateľa*), ktorý ukazuje späť na uzol predstavujúci kratší kontext. Ďalší ukazateľ pri konštrukcii *trie*, ktorý sa využíva, je *base potinter* alebo *ukazovateľ základu*. Ten určuje, ktorý uzol je naposledy pridaný, alebo upravovaný.

Obrázok 4.2 znázorňuje *vine* a *base ukazovatele* pre *trie* z predošlého príkladu. Čiernou farbou sú znázornené *ukazovatele base* a sivou farbou *ukazovatele vine*. V každom obrázku sú vyznačené len tie, ktoré boli vytvorené v danom kroku. To však neznamená, že tieto



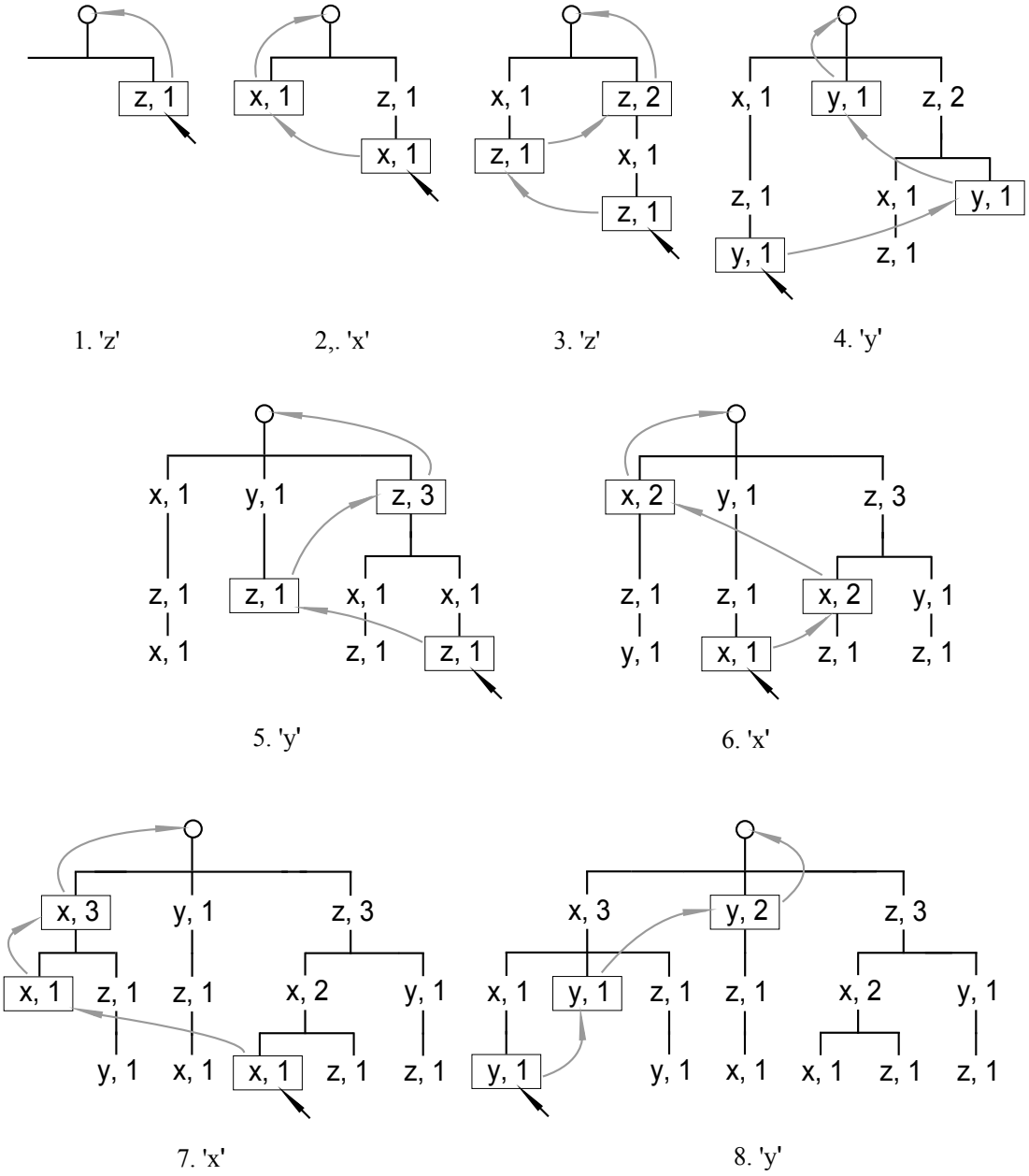
Obrázok 4.1: Trie pre reťazec *zxzyzxxy*.

ukazovateľe v nasledujúcom krokuzanikajú, nie sú vyznačené len pre prehľadnosť, ale nadaľej existujú. O *ukazovateľoch vine* všeobecne platí, že uzly so symbolom s v hĺbke h ukazujú na uzol symbolu s v hĺbke $h-1$. Z toho vyplýva aj to, že uzly v hĺbke 1 ukazujú na koreň stromu.

Algoritmus pridávajúci a upravujúci uzly *trie* pracuje nasledovne:

1. Sleduj *ukazovateľ base* uzla X. Sleduj *vine* z uzla X do uzla Y. Pridaj S ako potomka uzlu Y a nastav *base* tak, aby na neho ukazoval. Ak už Y potomka má, zvýš počítadlo tohoto uzla o 1. Nazveme tento uzol A.

2. Opakuj krok 1 bez aktualizácie *base*. Prejdi pomocou *vine* z uzla Y do uzla Z. Pridaj S ako potomka uzlu Z, alebo aktualizuj existujúceho potomka. Tento uzol označíme



Obrázok 4.2: Trie s *ukazovateľmi vine* a *base* pre reťazec *zxzyzxxy*.

B. Ak neexistuje z A do B *ukazovateľ vine*, vytvor ho.

3. Opakuj dovtedy, keď pridáš, alebo inkrementuješ uzol na úrovni 1.

Teraz je zrejmé, prečo je štruktúra *trie* taká výhodná. Pri spracovaní vstupného symbolu sa upravuje maximálne v $n + 1$ krokoch. Prepnutie kontextu je rovnako veľmi jednoduché, robí sa tak, že sa prechádza do zodpovedajúceho uzla cez *ukazovateľ vine*.

4.6.2 Spolupráca PPM s aritmetickým kóderom

V podkapitole 3.3 sú popísané princípy implementácie aritmetického kódovania. Ako už bolo spomenuté vyššie v tejto kapitole, metóda *PPM* vytvára adaptívny model určujúci pravdepodobnosti symbolov, ktoré sú vstupom pre aritmetické kódovanie, príp. dekódovanie.

Kódovanie prebieha štandardne. Model určuje početnosť daného vstupného symbolu. V prípade že tento symbol nenájde, pošle na zakódovanie *escape symbol* (jeho početnosť tiež určuje model na základe kontextu v ktorom sa nachádza). Tým sa znižuje rád kontextu. Pred tým sa však *symbol escape* aritmeticky zakóduje. Model ďalej pokračuje v kratšom kontexte. Znova zisťuje, či sa vstupný symbol v danom kontexte nájde, ak áno odošle ho spolu s jeho pravdepodobnosťou aritmetickému kóderu. Po zakódovaní všetkých symbolov sa zakóduje symbol *eof*, čím sa ukončí kódovanie.

Pri dekódovaní určí model *PPM* celkovú početnosť symbolov v aktuálnom kontexte. Potom je aritmetický kóder schopný určiť celkovú početnosť aktuálneho zakódovaného symbolu na vstupe. Podľa toho môže určiť do akého podintervalu pravdepodobnosti symbol patrí a určiť (dekódovať) ho. V prípade dekódovania *escape symbolu* pokračuje model dekóderu ďalšou iteráciou, teda prepnutím na kratší kontext a znova určuje celkovú početnosť symbolov v tomto kontexte.

Kapitola 5

Testovanie aplikácie

Dnes sú kompresné metódy veľmi rozličné a rôznorodé. Preto sa na porovnávanie ich výkonu používajú tzv. *korpusy*. Sú to kolekcie niekoľkých nemeniacich sa súborov, špeciálne zostavených na praktické testovanie kompresných algoritmov.

Hlavnou vlastnosťou kompresie na porovnávanie jednotlivých metód je *kompresný pomer*, tak ako je popísaný v podkapitole 2.3:

$$Kompresný\,pomer = \frac{veľkosť\,výstupných\,dát}{veľkosť\,vstupných\,dát},$$

tiež sa niekedy k tomuto pomeru uvádza jednotka *bit/Byte* (b/B). Pri testoch sa používa metóda *PPMC* komprimujúca celý korpus.

5.1 Calgary korpus

Tento korpus slúži na porovnávanie kompresných metód už od roku 1987 [9]. Obsahuje niekoľko typov súborov obsahujúcich prevažne textové dáta. Je to niekoľko anglicky písaných kníh (*book1* a *book2*), niekoľko tiež anglicky písaných článkov (*paper1* až *paper6*). Ďalej obsahuje súbor s príspevkami z diskusného severu (*news*), niekoľko zdrojových kódov programov (*progc*, *progl* a *progp*), súbory objektového kódu (*obj1*, *obj2*). Nakoniec obsahuje jeden súbor s geofyzikálnymi dátami (*geo*), záznam terminálovej relácie (*trans*) a jeden obrázok (*pic*). Zoznam súborov uvádza tabuľka 5.1.

Aplikácia je testovaná práve na tomto korpuse. Je už síce dosť zastaralý, ale na testovanie tejto aplikácie postačí. Existujú aj iné korpusy, treba najmä spomenúť *Canterbury korpus*. Ten má oproti *Calgary korpusu* niekoľko výhod. Mimo iné sú popísané aj v [11] ako aj v [10]. Slúži najmä na testovanie bezstratových kompresných metód a postupne v testovaní nahrádza *Calgary korpus*.

5.2 Možnosti PPM a aritmetického kódovania

Tu sú uvedené možnosti samotnej metódy *PPM* ako skvalitniť kompresiu zvolením vhodných premenných pre rád kontextu a veľkosť dátového typu pre premenné aritmetického kódera. Oba tieto aspekty môžu vplývať na kompresný pomer.

Meno súboru	Popis	Veľkosť [B]
bib	Bibliography in UNIX *refer format*	111261
book1	Far From the Medding Crowd	768771
book2	Principles of Computer Speech	610856
geo	Geological seismic data	102400
news	Usenet news file	377109
obj1	VAX object code	21504
obj2	Macintosh object code	246814
paper1	Arithmetic coding for data compression	53161
paper2	Computer(in)security	82199
paper3	In search of autonomy	46526
paper4	Programming by example revisited	13286
paper5	Logical implementation of Arithmetic	11954
paper6	Compact hash tables	38105
pic	Bitmap picture	513216
progc	C source code	39611
progl	Lisp source code	71646
progp	Pascal source code	71646
trans	Transcript using terminal	93695
	Spolu	**3255589**

Tabuľka 5.1: Calgary korpus.

5.2.1 Rád kontextu

Ako už bolo spomenuté v podkapitole 4.5, na veľkosti rádu kontextu závisí kvalita akú *PPM* podáva. Tabuľka 5.2 zhrňuje kompresné pomery pre jednotlivé dĺžky kontextov n (0 až 10).

Ako je vidno, najlepšie výsledky podáva *PPM* pri ráde kontextu 5. Potom začína kompresný pomer mierne stúpať, ale pamäťová zložitosť narastá niekoľkonásobne. Preto je zrejmé, že tak dlhé kontexty sa používať neoplatia. Keďže pri vysokých rádoch kontextu často prechádza k prepínaniu na kontexty kratšie (a musí sa zakódovať *escape symbol*), preto kompresný pomer rastie so zvyšujúcimi sa rádmi kontextu.

V tabuľke je uvedená aj dĺžka kontextu 0, kedy *PPM* takmer zodpovedá adaptívnemu aritmetickému kóderu. Vtedy metóda určuje, či bol symbol videný. Ak áno odošle aritmetickému kóderu jeho pravdepodobnosť. Ak ak sa nevyskytol, prepne sa *PPM* do rádu -1, a pošle kóderu symbol s pravdepodobnosťou $1/|A|$, kde $|A|$ je veľkosť abecedy. V ráde kontextu -1 majú všetky symboly túto pravdepodobnosť. Pri prepínaní do tohto rádu dosahuje *PPM* rádu 0 ešte horší kompresný pomer ako adaptívny aritmetický kóder sám, kvôli tomu, že musí kódovať aj *symboly escape*. V nasledujúcich testoch sa bude uvažovať kontext rádu 5.

5.2.2 Veľkosť dátových typov premenných aritmetického kódera

Premenné *low* a *High*, ktoré využíva aritmetický kóder k uloženiu hraníc aktuálneho intervalu pravdepodobnosti, využívajú na to 63-bitov, ako už bolo spomenuté a dôvody vysvetlené v podkapitole 3.3.1.

Aj veľkosť týchto dátových typov, resp. veľkosť, ktorú používajú na uloženie hraníc intervalu, prispieva na zlepšenie kompresného pomeru, ako je vidno v tabuľke 5.3. Pri príliš

rád kontextu n	**Kompresný pomer [b/B]**
0	4,37
1	3,34
2	2,64
3	2,28
4	2,13
5	2,10
6	2,12
7	2,14
8	2,15
9	2,16
10	2,17

Tabuľka 5.2: Kompresný pomer v závislosti na ráde kontextu.

Veľkosť typu [b/B]	**Kompresný pomer [b/B]**
15	1,30
20	1,19
25	0,82
30	0,81
35	0,81
40	0,81
45	0,81
50	0,81
55	0,81
60	0,81

Tabuľka 5.3: Kompresný pomer v závislosti na veľkosti typov premenných aritmetického kódera.

malých veľkostiach premenných sa kompresný pomer zhoršuje. Na hranici veľkosti 30 bitov sa však tento pomer už nezlepšuje, aj napriek zvyšovaniu veľkosti dátového typu.

Táto implementácia neuvažuje variabilitu spomínanej veľkosti, je však vhodné tento fakt spomenúť a uviesť jeho dôvody. Kompresný pomer sa zhoršuje preto, že pri výpočtoch premennej *stepSize* (viď podkapitola 3.3.1):

$$stepSize = \frac{(High - Low + 1)}{totalSymbols},$$

uvažujeme túto veľkosť ako podiel veľkosti aktuálneho intervalu ku počtu zatiaľ prečítaných symbolov. Keďže sa používa celočíselné delenie, dochádza k strate presnosti. Veľkosť tejto chyby zaokrúhľovania vzniká pri delení porovnateľných hodnôt čitateľa a menovateľa. V prípade ze by bol čitateľ oveľa väčší ako menovateľ, táto chyba by bola menšia. Keďže na premennej *stepSize* závisí aj aktualizovaná hodnota *Low* a *High*

$$High = Low + stepSize \times highCount(n) - 1,$$

$$Low = Low + stepSize \times lowCount(n)$$

je zrejmé, že použitím väčších dátových typov pre tieto premenné dochádza ku zmenšovaniu chyby pri zaokrúhľovaní. Pri veľkých dátových typoch je zaistené, že *High* bude oveľa väčšia ako *Low*, tým bude zaistená malá chyba pri zaokrúhľovaní.

5.3 Porovnanie PPM s inými kompresnými metódami

V tabuľke 5.4 je *PPMC* porovnaná s inými kompresnými metódami. Tieto údaje je možné nájsť [10], kde sú mimo iné uvedené aj výsledky týchto metód testované na iných korpusoch, nie len *Calgary korpusom*.

PPMC je porovnaná s metódou *LZ77* a *LZRW1* (slovníkové metódy), výsledkami knižnice *bzip*, ako aj s variantou *PPMD5*, podávajúcou veľmi dobré výsledky.

Ako je vidno, metódy rodiny *PPM* podávajú kvalitné výsledky hlavne pri kompresii textu, hlavne v porovnaní so slovníkovými metódami. Naopak sú menej vhodné na kompresiu objektových dát a náhoných dát (*geo*). V tomto prípade je lepšie použiť *bzip*. To je dané tým, že *PPM* uvažujú modelovanie kontexty, ktoré sa v náhoných dátach dajú tažko využiť práve preto, že sú náhodné. V praxi tieto metódy odosielajú aritmetickému kóderu príliš nízke pravdepodobnosti charakteristické pre symboly náhodných dát. Tieto pravdepodobnosti potom komprimujú na základe entropie aritmetického kódovania nekvalitne.

	Kompresný pomer [b/B]				
Súbor	**PPMC**	**bzip**	**PPMD5**	**LZ77**	**LZRW1**
bib	1,99	1,95	1,89	2,85	4,77
book1	2,34	2,40	2,34	3,44	5,47
book2	2,05	2,04	1.98	2,98	4,75
geo	4,89	4,48	4.96	6,71	6,77
news	2,45	2,51	2,42	3,51	4,94
obj1	4,05	3,87	3,70	5,11	4,93
obj2	2,61	2,46	2,35	3,51	4,13
paper1	2,45	2,46	3,36	3,24	4,63
paper2	2,45	2,42	2,34	3,20	4,90
pic	0,89	0,77	0,95	2,01	2,05
progc	2,51	2,50	2,40	3,28	4,37
progl	1,82	1,72	1,69	3,44	3,53
progp	1,82	1,71	1,72	3,42	3,43
trans	1,63	1,50	1,50	2,31	3,71
korpus	2,26	2,09	2,08	3,14	4,45

Tabuľka 5.4: Porovnanie *PPMC* s inými metódami.

Kapitola 6

Záver

Cieľom tejto aplikácie bola implementácia kompresnej metódy *PPM* ako programovej knižnice a konzolovej aplikácie. Aplikácia sa venuje implementácii varianty *PPMC*, ktorá má spomedzi základných metód (*PPMA*, *PPMB*) najlepšie výsledky. Princípy a algoritmy boli detailne popísané v predchádzajúcom texte na príkladoch. Samotná implementácia je popísaná v priloženej dokumentácií.

Po implementácii som aplikáciu otestoval na súboroch kolekcie *korpusu Calgary* a konzultoval som ako vplývajú vlastnosti *PPM* a aritmetického kódera na kvalitu kompresie. Táto kvalita je určená kompresným pomerom, ktorý metóda dosahuje pri kompresií konkrétnych súborových typov.

Experimentálne som určil najvhodnejšie hodnoty týchto vlastností. Kompresia dosahuje najlepší pomer pri využití kontextu rádu 5. Ďalej som poukázal na to, ako môže veľkosť dátového typu pre uloženie premenných aritmetického kódera ovplyvniť kompresný pomer. Ukázal som, že je dobré používať dátové typy veľké aspoň 30 bitov, kedy už zaokrúhľovanie celočíselného delenia na kompresiu vplyv nemá.

Ďalej som *PPM* porovnal s inými často využívanými metódami, resp. programami. Dospel som k záveru, že metóda je najlepšie použiteľná na kompresiu textu. Taktiež je vhodná na kompresiu nie príliš náhodných dát.

Metódy, resp. varianty na metódu *PPM*, sa dnes často využívajú a stále zdokonalujú. Ďalšie existujúce varianty ako napríklad *PPM**, *PPM+*, *PPMZ* a mnohé ďalšie rýchlo sa vyvíjajúce varianty sú považované za jedny z najlepších metód na kompresiu prirodzeného jazyka.

Jednou z možností rozšírenia tejto práce ako aj implementácie je zakomponovanie týchto nových metód s možnosťou užívateľského výberu varianty na kompresiu. Iným rozšírením by bolo doimplementovať niektoré iné kompresné metódy, pričom by sa na konkrétny druh dát využila tá, ktorá je najvhodnejšia.

Literatúra

[1] ABEL, J.: Arithmetic Coding (AC) [online]. 2010, cit. 19. 4. 2010.
URL http://www.data-compression.info/Algorithms/AC/

[2] BODDEN, E.; CLASEN, M.; KNEIS, J.: *Arithmetic Coding revealed*. McGill University, 2007.

[3] CARPENTER, B.: Compression via Arithmetic Coding in Java. Version 1.1 [online]. 2003, cit. 19. 4. 2010, posledná modif. 2003.
URL http://www.colloquial.com/ArithmeticCoding/

[4] CLEARY, J. G.; WITTEN, I. H.: Data compression using adaptive coding and partial string matching. In *IEEE Transactions on Communications*, ročník 32, 1984, ISSN 0090-6778, str. 396-402.

[5] COVER, T. M.; THOMAS, J. A.: *Elements of Information Theory*. New York: Wiley-Interscience, 2006, ISBN 0-471-24195-4.

[6] HUFFMAN, D. A.: A Method for the Construction of Minimum-Redundancy Codes. In *Proceedings of the I.R.E.*, 1952, str. 1098-1102.

[7] MOFFAT, A.: Implementing the PPM Compression Scheme. In *IEEE Transactions on Communications*, ročník 38, 1990, ISSN 0090-6778, str. 1917-1921.

[8] NOŠEK, A.: *Implementace kompresní metody PPM*. Diplomová práce, ČVUT v Praze, Praha, 2006.

[9] POWELL, M.: The Calgary Corpus [online]. 2010, cit. 19. 4. 2010, posledná modif. 8. 1. 2001.
URL http://corpus.canterbury.ac.nz/descriptions/#calgary

[10] POWELL, M.: The Canterbury Corpus [online]. 2010, cit. 19. 4. 2010, posledná modif. 8. 1. 2001.
URL http://corpus.canterbury.ac.nz/descriptions/

[11] SALOMON, D.: *Data Compression: The Complete Reference*. Springer, 2004, ISBN 0-387-40697-2.

[12] SHANNON, C. E.; WEAVER, W.: *The Mathematical Theory of Communication*. University of Illinois Press, 1998, ISBN 0-252-72546-8.

Dodatok A

Obsah CD

- Dokumenty
  - Technická správa
  - Programová dokumentácia
- Zdrojové súbory
  - Zdrojové súbory aplikácie
  - Zdrojové súbory technickej správy v LaTeX
- súbory korpusu Calgary