FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Implementace kompresních metod LZY, LZMW a LZAP v jazyce Java
Student:	Ján Bobot
Vedoucí:	Ing. Radomír Polách
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2019/20

Pokyny pro vypracování

1) Nastudujte pokročilé varianty metody LZW - jmenovitě LZY, LZMW a LZAP.
2) Dané metody analyzujte a navrhněte vhodný způsob jejich implementace v jazyce Java jako součást knihovny SCT vyvíjené na Katedře teoretické informatiky FIT ČVUT.
3) Metody implementujte a testujte na vhodných korpusech.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 6. března 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalárska práca

Implementace kompresních metod LZY, LZMW a LZAP v jazyce Java

Ján Bobot

Katedra teoretické informatiky
Vedúci práce: Ing. Radomír Polách

4. januára 2019

Poďakovanie

Rád by som poďakoval Ing. Radomírovi Poláchovi za rady a ústretovosť pri vedení bakalárskej práce.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov. V súlade s ustanovením § 46 odst. 6 tohoto zákona týmto udeľujem bezvýhradné oprávnenie (licenciu) k užívaniu tejto mojej práce, a to vrátane všetkých počítačových programov ktoré sú jej súčasťou alebo prílohou a tiež všetkej ich dokumentácie (ďalej len „Dielo“), a to všetkým osobám, ktoré si prajú Dielo užívať.

Tieto osoby sú oprávnené Dielo používať akýmkoľvek spôsobom, ktorý nezhoršuje hodnotu Diela, a za akýmkoľvek účelom (vrátane komerčného využitia). Toto oprávnenie je časovo, územne a množstevne neobmedzené. Každá osoba, ktorá využije vyššie uvedenú licenciu, sa však zaväzuje priradiť každému dielu, ktoré vznikne (čo i len čiastočne) na základe Diela, úpravou Diela, spojením Diela s iným dielom, zaradením Diela do diela súborného či zpracovaním Diela (vrátane prekladu), licenciu aspoň vo vyššie uvedenom rozsahu a zároveň sa zaväzuje sprístupniť zdrojový kód takého diela aspoň zrovnateľným spôsobom a v zrovnateľnom rozsahu ako je zprístupnený zdrojový kód Diela.

V Prahe 4. januára 2019 ……

Odkaz na túto prácu

Bobot, Ján. *Implementace kompresních metod LZY, LZMW a LZAP v jazyce Java*. Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Táto práca sa zaoberá slovníkovými kompresnými metódami LZMW, LZAP a LZY, ktoré sú variantami metódy LZW. Práca popisuje na akých princípoch dané metody fungujú a zaoberá sa tiež ich implementáciou. Algoritmy sú integrované do knižnice SCT a zároveň sú otestované na dátach Pražského korpusu. Keďže implementované algoritmy obsahujú rôzne nastaviteľne parametre, ich vliv na výkon algoritmov je tiež preskúmaný.

Klíčová slova kompresia dát, dekompresia dát, Java, LZAP, LZMW, LZY, SCT, slovník

Abstract

The thesis is concerned with the variants of the LZW dictionary compression method, namely LZMW, LZAP and LZY. It describes the principles upon which these methods function and also deals with their implementation. The algorithms are integrated into the SCT library and tested using the data from the Prague Corpus. Implemented algorithms also contain various adjustable parameters, which influence upon the performance of them is examined as well.

Keywords data compression, data decompression, dictionary, Java, LZAP, LZMW, LZY, SCT

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

V súčasnej dobe sa v informačnej technike pracuje s dátami rôznych veľkostí a tieto dáta sa veľmi často potrebujú niekde uložiť a premiestniť. Je zmysluplné, aby sa tieto dva operácie vykonávali čo najefektívnejšie. To znamená, že sa dáta uložia tak, aby zaberali čo najmenej miesta a premiestnia sa čo najrýchlejšie. Hlavným faktorom pri týchto úkonoch je koľko dát je potrebné premiestniť, čím menej, tým to bude rýchlejšie. Oblasť v informačnej technike, ktorá tento problém rieši je kompresia. Zmenšenie počtu dát je dosiahnuté snahou minimalizovať ich redundanciu, t.j. ich transformovanie do efektívnejšej reprezentácie.

Dodnes sa vymyslel veľký počet rôznych kompresných metód, ktoré sú založené na špecifických princípoch, majú zameranie na špecifické typy dát a vykonávajú určitý kompromis medzi rýchlosťou a účinnosťou kompresie. Pri kompresných metódach je to väčšinou tak, že algoritmus buď môže byť rýchlejší, ale zase kompresia je menej účinná alebo algoritmus je pomalší, ale kompresia je viac účinná alebo niečo medzi tým. Hlavný dôvod pre takú množnú existenciu kompresných metód je ten, že pre isté situácie je potrebná kompresia čo najrýchlejšia aj na úkor účinnosti a pri iných na rýchlosti až tak nezáleží, ale je potrebné aby účinnosť bola čo najvyššia. Pre prvý prípad by to mohlo byť napríklad streamovanie nejakého videa a pre druhý prípad, by to mohla byť archivácia veľkého počtu záznamov, ku ktorím sa až tak často nepristupuje.

LZW je slovníková kompresná metóda, ktorá kompresiu vykonáva odstránením opakujúcich sa fráz v texte a ich nahradením slovníkovými indexmi. Nové frázy sa postupne ukladajú do slovníka, popritom ako algoritmus dáta číta a na základe toho je schopný nahradiť frázy, ktoré už predtým videl, len tým indexom. Táto metóda je známa tým, že je veľmi rýchla, ale naopak jej kompresia nie je až tak účinná. Preto existovala snaha túto metódu modifikovať a odstrániť tento nedostatok, aj keď to znamená obetovanie jej rýchlosti. Vzniklo niekoľko možných variant, ale táto práca sa bude zaoberať tými naj-

známejšími: LZAP, LZMW a LZY.

Knižnica SCT, vyvíjaná na Fakulte Informačných technológií, tieto tri algoritmy ešte neobsahuje, a tak sa táto práca nebude zaoberať len ich implementáciou, ale aj ich integráciou do tejto knižnice.

Cieľ práce

Cieľom tejto práce je analyzovať známe varianty kompresnej metódy LZW, čiže LZAP, LZMW a LZY, a implementovať ich do knižnice SCT.

Ďalším krokom je vykonanie testovania daných algoritmov na vhodných dátach a porovnať ich nielen medzi sebou, ale aj s ich predchodcom LZW. Keďže algoritmy obsahuje rôzne nastaviteľné parametre, je potrebne ukázať vplyv ich zmeny na algoritmy.

Štruktúra práce

Prvá kapitola objasní potrebné základné koncepty a pojmy, ktoré čitateľ potrebuje pre pochopenie špecifických kompresných metód popísaných neskôr v tejto práci.

Druhá kapitola sa zameriava na slovníkovú kompresnú metódu LZW, ktorá je základom pre metódy, o ktorých sa bude hovoriť v nasledujúcej kapitole.

Tretia kapitola sa zaoberá analýzou kompresných metód LZWM, LZAP a LZY.

Štvrtá kapitola popisuje implementáciu kompresných metód z predošlej kapitoly do knižnice SCT. Hovorí, akým spôsobom sa dané metódy implementovali a zahrňuje tiež popis spustiteľného klienta, ktorý sa používa pre vykonanie danej kompresnej metódy na špecifikovanom súbore.

Piata kapitola sa zaoberá testovaním implementovaných metód. Popisuje, ako bolo testovanie vykonané a tiež prezentuje jeho výsledky. Diskutuje aj o tom, aký dopad majú meniace sa nastaviteľné parametre algoritmov na ich rýchlosť a kompresný pomer.

Kapitola 1

Definície pojmov

Táto kapitola sa zaoberá definovaním základných pojmov z kompresie dát, ktoré sú relevantné k témam, o ktorých sa bude hovoriť v nasledujúcich kapitolách.

1.1 Kompresia dát

Kompresia dát alebo *kódovanie*, je proces transformácie sekvencie bitov (vstupnej alebo nezakódovanej) do inej sekvencie bitov (výstupnej alebo zakódovanej), ktorá má v ideálnom prípade kratšiu dĺžku ako tá vstupná. Informácia zachovaná v nezakódovanej sekvencii sa týmto procesom nestratí vôbec (*bezstratová kompresia*) alebo len čiastočne stratí (*stratová kompresia*). Celková vstupná a výstupná sekvencia sa tiež môže označovať ako vstupné a výstupné dáta.

1.2 Dekompresia dát

Dekompresia dát alebo *dekódovanie*, je inverzný proces kompresie, ktorý transformuje zakódovanú sekvenciu bitov do jej nezakódovanej podoby.

1.3 Kompresný algoritmus

Kompresný algoritmus alebo *Kódovací algoritmus*, je sekvencia krokov, ktoré celkovo vykonávajú proces kompresie dát.

1.4 Dekompresný algoritmus

Dekompresný algoritmus alebo *dekódovací algoritmus*, je sekvencia krokov, ktoré celkovo vykonávajú proces dekompresie dát.

1.5 Kompresná metóda

Kompresná metóda je spoločný názov, ktorý sa používa k označeniu určitého kompresného a jeho príslušného dekompresného algoritmu.

1.6 Symbol

Keďže sú počítačové pamäte organizované v bajtoch a nie v bitoch, budú sa jednotlivé elementy vo vstupných a výstupných sekvenciach pre kompresné metódy označovať ako *symboly*, ktoré reprezentujú jeden bajt v sekvencii, čiže skupinu ôsmych bitov. Pri textovej reprezentácii týchto symbolov sa bude používať kódovanie ASCII.

1.7 Model

Kompresná metóda, aby mohla vykonať kompresiu, používa pri svojich algoritmoch štruktúru, ktorej sa hovorí *model* a ten reprezentuje určité informácie o práve spracovávanom vstupe.

Pri kódovaní je model použitý pre výpočet samotného kódu individuálnych skupín symbolov rôznych dĺžok zo vstupu. Celková dĺžka týchto kódov zapísaná na výstup by v ideálnom prípade mala dosahovať menších rozmerov ako dĺžka vstupu.

Pri dekódovaní je použitý pre generáciu skupín symbolov rôznych dĺžok na základe kódov zo vstupu. Preto aby bolo dekódovanie úspešné, musí dekódovací algoritmus použiť rovnaký model ako ten kódovací, inak nebude schopný vygenerovať presnú sekvenciu symbolov, ktorá bola vstupom pre kódovací algoritmus.

1.8 Adaptívna kompresná metóda

Adaptívna kompresná metóda používa pri svojich algoritmoch takzvaný *adaptívny model*. Tento model je na úplnom začiatku algoritmu úplne prázdny a algoritmus si ho postupne vytvára popritom, ako ho používa pre kódovanie/dekódovanie. Výhodou tohto prístupu je to, že sa model samotný nemusí uložiť ako metadáta s výstupnými kódmi vyprodukovanými kódovacím algoritmom. To samozrejme znižuje celkovú dĺžku výstupných dát a tiež umožňuje model prispôsobovať práve spracovávanému okoliu vstupných symbolov.

1.9 Slovníková kompresná metóda

Slovníková kompresná metóda reprezentuje model pomocou dátovej štruktúry, ktorej sa hovorí *slovník*. Algoritmus do slovníka ukladá určitým spôsobom

sekvencie symbolov, na ktoré predtým narazil vo vstupe. Na výstup algoritmus zapisuje kódy alebo *indexy*, ktoré označujú umiestnenie určitých sekvencií symbolov, ďalej len *fráz*, v slovníku.

1.10 Symetrická kompresná metóda

Symetrická kompresná metóda je taká, pri ktorej zložitosť kódovacieho algoritmu je podobná zložitosti algoritmu dekódovacieho. Kódovací a dekódovací algoritmus používajú veľmi často skoro identický model, len s tým rozdielom, že ho používajú opačným spôsobom. To znamená, že kódovací algoritmus do modelu posiela určité symboly zo vstupu a vypočítava kódy týchto symbolov, a dekódovací algoritmus potom neskôr tieto kódy pošle naspäť do modelu a on vygeneruje symboly.

1.11 Kompresný pomer

Kompresný pomer je definovaný ako:

$$\text{Kompresný pomer} = \frac{\text{Dĺžka výstupných dát}}{\text{Dĺžka vstupných dát}}$$

To znamená, že čím je dĺžka výstupných dát menšia, tým je menší aj kompresný pomer. Snaha je dosiahnuť čo najmenšieho kompresného pomeru, ktorý je v ideálnom prípade menší ako 1. Keď je kompresný pomer rovný alebo väčší ako 1, znamená to, že sa kompresnému algoritmu nepodarilo zmenšiť dĺžku vstupných dát, pravdepodobne z toho dôvodu, že algoritmus nebol vhodný pre dané vstupné dáta.

Napríklad kompresný pomer s hodnotou 0,75 znamená, že sa podarilo zmenšiť vstupné dáta na 75% originálnej veľkosti.

1.12 Dátový korpus

Dátové korpusy sa používajú, aby bolo možné otestovať výkon rôznych kompresných metód. Sú to štandardizované množiny súborov rôzneho typu. Nad týmito súbormi sa testuje kompresný pomer a čas kódovania/dekódovania.

Existuje niekoľko takýchto korpusov, ako napríklad Calgary (1987), Canterbury (1997) a Pražský korpus (2011). Pražský korpus bol vyvinutý na ČVUT v Prahe [1].

1.13 Knižnica SCT

Small Compression Toolkit [2] je knižnica obsahujúca rôzne kompresné metódy, ktorá je vyvíjaná na Katedre teoretickej informatiky FIT ČVUT v Prahe.

Je implementovaná v jazyku Java a je to tiež knižnica, do ktorej sa implementované kompresné metódy v tejto práci integrujú.

Kapitola 2

Kompresná metóda LZW

Ešte predtým ako sa môže začať hovoriť o metódach LZMW, LZAP, LZY, ktorými sa táto práca zaoberá, je potrebné najskôr spomenúť metódu LZW. Ide o základnú metódu od ktorej boli odvodené vyššie uvedené metódy. Táto kapitola sa bude venovať práve jej analýze.

2.1 Kompresná metóda LZW

LZW je kompresná metóda, ktorú vyvinul *Terry Welch* v roku 1984 [3]. Jedná sa o populárnu variantu metódy LZ78 z rodiny LZ algoritmov, ktorú publikovali *Abraham Lempel* a *Jacob Ziv* v roku 1978 [4]. Metóda LZW je klasifikovaná ako slovníková, adaptívna, bezstratová a symetrická kompresná metóda. Mala praktické využitie vo formátoch, ako napríklad GIF, PDF a v Unixovom programe *compress*.

2.1.1 Algoritmus kompresie

Algoritmus začína inicializáciou svojho slovníka, čo v prípade LZW znamená, že do neho vloží všetky frázy dĺžky jedna, čiže celú abecedu alebo konkrétne všetky hodnoty jedného symbolu (od 0 do 255). Po inicializácii algoritmus začne čítať vstupné dáta po jednom symbole v cykle až dokým sa nedostane na koniec. Každý načítaný symbol B si postupne reťazí do frázy G dovtedy dokým zistí, že daná fráza G sa v slovníku už nenachádza. Keď sa to stane, tak algoritmus zistí index frázy G bez toho posledného symbolu B, čo znamená najdlhšiu frázu, ktorú sa mu podarilo nájsť v slovníku a na výstup tento index zapíše. Následne sa fráza G do slovníka vloží. Posledný symbol B frázy G, sa nastaví ako nová fráza G a postup sa opakuje.

Pre lepšie pochopenie je tento algoritmus znázornený v pseudo kóde 1 a tiež je ďalej uvedený príklad behu 2.1.

Algorithm 1 Pseudo-kód kompresného algoritmu metódy LZW

```
1: inicializuj slovník s prvými 256 frázami
2: G ← ϵ
3: while vstup nie je na konci do
4:     načítaj symbol B zo vstupu
5:     if fráza G+B sa nenachádza v slovníku then
6:         zapíš index I frázy G na výstup
7:         pridaj frázu G+B do slovníka
8:         G ← B
9:     else
10:        G ← G + B
11: zapíš index I frázy G na výstup
```

Ako je lepšie vidieť zo pseudo-kódu 1, tento algoritmus si frázy v slovníku rozširuje stále len po jednom symbole. Z toho vyplýva, že frázy rastú celkom pomaly, čo má za následok podstatné oneskorenie, než sa začnú indexovať frázy akceptovateľnej dĺžky. Ale na druhej strane sa kvôli tomuto faktu zaručuje, že slovník obsahuje všetky prefixy každej jeho frázy, čo má pozitívny efekt pri implementácii dátovej štruktúry slovníka.

Nemusí byť tiež jasné, prečo sa slovník musí inicializovať pre celú abecedu. Nesmie sa zabúdať na to, že sa na výstup zapisujú len indexy a keby sa slovník na začiatku neinicializoval, tak by dekompresný algoritmus nevedel ani ako začať, pretože by mu ten prvý index nič nehovoril. Pri metóde LZ78, ktorý si slovník na začiatku neinicializuje, je táto situácia vyriešená tak, že si spolu s indexom ešte na výstup algoritmus zapisuje nejaké ďalšie informácie, čo má ale nepriaznivý dopad na kompresný pomer a metóda LZW je v tom oproti nej výhodnejšia.

2.1.2 Príklad kompresie

Vstup: Reťazec „yabbadabbadabbadoo“ v ASCII kódovaní.

Tabuľka 2.1: Príklad použitia kompresného algoritmu metódy LZW

Krok	**G**	**B**	**G+B v slovníku?**	**I**	**Frázy pridané do slovníka**
1	ϵ	y	Áno	-	-
2	y	a	Nie	121	ya(256)
3	a	b	Nie	97	ab(257)
4	b	b	Nie	98	bb(258)
5	b	a	Nie	98	ba(259)
6	a	d	Nie	97	ad(260)
7	d	a	Nie	100	da(261)
8	a	b	Áno	-	-
9	ab	b	Nie	257	abb(262)
10	b	a	Áno	-	-
11	ba	d	Nie	259	bad(263)
12	d	a	Áno	-	-
13	da	b	Nie	261	dab(264)
14	b	b	Áno	-	-
15	bb	a	Nie	258	bba(265)
16	a	d	Áno	-	-
17	ad	o	Nie	260	ado(266)
18	o	o	Nie	111	oo(267)
19	o	-	-	111	-

Výstup: Postupnosť indexov (121, 97, 98, 98, 97, 100, 257, 259, 261, 258, 260, 111, 111).

2.1.3 Algoritmus dekompresie

Dekompresný algoritmus potrebuje vytvoriť presne ten istý slovník ako kompresný algoritmus a tým istým spôsobom, aby bol schopný zrekonštruovať originálny vstup. Ale keďže má k dispozícii na vstupe len indexy do slovníka, musí ísť na to inak.

Podobne ako pri kompresnom algoritme je slovník na začiatku inicializovaný celou abecedou. Po inicializácii algoritmus číta postupne indexy v cykle a pre každý index I si v slovníku zistí danú frázu G nachádzajúcu sa pod týmto indexom a túto frázu zapíše na výstup. Kompresný algoritmus by v tomto kroku do slovníka pridával frázu G zreťazenú s ďalším symbolom B zo vstupu, ale dekompresný algoritmus k tomuto symbolu B nemá prístup a tak musí tento symbol dostať z prvého symbolu ďalšej frázy G, na ktorú ukazuje ďalší

index I. Takže z hľadiska pridávania do slovníka je dekompresný algoritmus o jeden krok pozadu za kompresným. Fráza G sa musí zapamätať do ďalšieho kroku a tak si ju algoritmus zapamätá ako frázu F. Potom ako si načíta novú frázu G a z nej si zoberie prvý symbol G[0], môže konečne do slovníka pridať frázu F+G[0], ktorú by kompresný algoritmus pridal už v minulom kroku. Týmto spôsobom sa pokračuje až dokonca vstupu.

Existuje jeden špeciálny prípad keď sa fráza G na indexe I ešte nenachádza v slovníku. Tento prípad nastane keď kompresný algoritmus na výstup zapíše index frázy, ktorú pridal posledne. Dekompresný algoritmus v tomto prípade musí do slovníka pridať frázu, ktorú zapísal na výstup v minulom kroku zreťazenú s jej prvým symbolom, takže frázu F+F[0].

Tento postup môžeme odpozorovať v pseudo-kóde 2 a v príklade behu algoritmu 2.2.

Algorithm 2 Pseudo-kód dekompresného algoritmu metódy LZW

```
1: inicializuj slovník s prvými 256 frázami
2: F ← ϵ
3: while vstup nie je na konci do
4:     načítaj index I zo vstupu
5:     if fráza na indexe I sa nachádza v slovníku then
6:         načítaj frázu G na indexe I zo slovníka
7:         zapíš frázu G na výstup
8:         pridaj frázu F+G[0] do slovníka
9:         F ← G
10:    else
11:        F ← F + F[0]
12:        zapíš frázu F na výstup
13:        pridaj frázu F do slovníka
```

2.1.4 Príklad dekompresie

Vstup: Postupnosť indexov (121, 97, 98, 98, 97, 100, 257, 259, 261, 258, 260, 111, 111).

Tabuľka 2.2: Príklad použitia dekompresného algoritmu metódy LZW

Krok	I	F	G	Výstup	Fráza pridaná do slovníka
1	121	ϵ	y	y	-
2	97	y	a	a	ya(256)
3	98	a	b	b	ab(257)
4	98	b	b	b	bb(258)
5	97	b	a	a	ba(259)
6	100	a	d	d	ad(260)
7	257	d	ab	ab	da(261)
8	259	ab	ba	ba	abb(262)
9	261	ba	da	da	bad(263)
10	258	da	bb	bb	dab(264)
11	260	bb	ad	ad	bba(265)
12	111	ad	o	o	ado(266)
13	111	o	o	o	oo(267)

Výstup: Reťazec „yabbadabbadabbadoo“ v ASCII kódovaní.

2.1.5 Dátová reprezentácia slovníka

2.1.5.1 Kompresný slovník

Hlavným cieľom kompresného slovníka je umožniť kompresnému algoritmu, aby bol schopný si zapamätať určité reťazce alebo frázy a tiež overiť, či daná fráza sa v ňom nachádza alebo nie. Úplným najjednoduchším spôsobom ako implementovať takúto dátovú štruktúru s touto funkcionalitou, by bolo vytvoriť pole o fixnej veľkosti, danou maximálnym počtom fráz v slovníku, a každý element by obsahoval jednu frázu. Hľadanie v takomto slovníku by prebiehalo tak, že by sa hľadaná fráza postupne porovnávala s každou frázou v slovníku, dokým by nastala zhoda alebo v najhoršom prípade, dokým by sa porovnali úplne všetky a žiadna zhoda nenastala. Pridávanie by prebiehalo tak, že by sa daná fráza pridala do poľa na pozíciu tesne za predošle pridanou frázou. Reprezentovať slovník takto by bolo veľmi jednoduché na implementáciu, ale na druhú stranu by operácia hľadania bola veľmi pomalá, nehovoriac o tom, že by slovník zaberal veľmi veľa miesta v pamäti, keďže by si pamätal všetky symboly všetkých svojich fráz.

Pre efektívnu implementáciu tohoto algoritmu je potrebná štruktúra, ktorá je o niečo efektívnejšia oproti predošlému príkladu. Našťastie z analýzy tohoto algoritmu je známe, že algoritmus do slovníka pridáva frázy tak, že si najprv

nájde frázu zo slovníka, ktorá odpovedá reťazcu zo vstupu a pridá do neho túto frázu zreťazenú s ďalším symbolom zo vstupu. To znamená, že slovník obsahuje prefixy všetkých svojich fráz. Tento fakt umožní vytvoriť slovník s rýchlym vyhľadávaním, pridávaním a tiež to podstatne zníži množstvo pamäte, ktorý slovník zaberá. Meno štruktúry, ktorá si dokáže efektívne pamätať reťazce spolu so všetkými ich prefixmi, je *trie*.

Trie je stromová štruktúra, v ktorej každý uzol reprezentuje jeden symbol a existencia určitého reťazca sa overí sledovaním rodičovských uzlov až do koreňa, ktorý reprezentuje prázdny reťazec. Keďže je toto možné pre každý uzol v strome, je možné každý uzol považovať za unikátnu frázu. Tiež je dôležité poznamenať, že rekonštrukcia reťazca sledovaním rodičovských uzlov, daný reťazec zrekonštruuje v opačnom poradí, takže je potom potrebné tento reťazec čítať opačným smerom od konca.

Každý uzol, alebo inak povedané fráza, si pamätá maximálne 256 referencií na uzly reprezentujúce frázu o jeden symbol dlhšiu, kde súčasný uzol je ich prefixom, pravé kvôli tomu, že symbol môže maximálne zakódovať 256 hodnôt. Keďže kompresný algoritmus tiež potrebuje na výstup zapisovať indexy týchto fráz, je potrebné, aby si každý uzol pamätal svoje identifikačné číslo, a tým bolo možné jednotlivé frázy unikátne rozlíšiť.

Hlavnou výhodou slovníka s trie oproti slovníku s poľom reťazcov je ten, že namiesto toho, aby si slovník zapamätal každý jeden symbol každého reťazca, si slovník zapamätá len tie symboly, ktoré sú potrebné. Napríklad, keď si slovník chce zapamätať množinu reťazcov S=„a“, „ab“, „abc“, zapamätá si fyzicky len symboly „abc“, z čoho si vie odvodiť, že si tiež pamätá aj predošlé dva prefixy „ab“ a „a“. Výhodou nie je len úspora pamäte, ale aj to, že algoritmus môže hľadať v slovníku frázy tak, že si pamätá ukazateľ na súčasnú frázu, čiže na jej posledný symbol. Existenciu frázy, ktorá je o jeden ľubovoľný symbol dlhšia si overí tak, že sa pozrie, či uzol obsahuje referenciu na susedný uzol práve s týmto symbolom. V prípade, že sa fráza nenachádza v slovníku, môže algoritmus tento uzol hneď využiť na to, že túto referenciu hneď aj vytvorí a tak pridá novú frázu do slovníka.

Aj napriek všetkým týmto výhodám, má slovník s trie ešte malý problém. Tým problémom je to, že každý uzol trie si pri jeho vytvorení musí alokovať pole o veľkosti 256, do ktorého bude neskôr ukladať referencie na podradené uzly. Vo väčšine času toto pole obsahuje neplatné referencie, čo nie je veľmi efektívne z hľadiska pamäte a mohlo by spôsobiť ešte väčšiu pamäťovú náročnosť ako pole s reťazcami. Samozrejme jedným riešením by mohlo byť to, že by sa pole nealokovalo na veľkosť 256 ihneď, ale len na veľkosť práve potrebnú, čiže by sa stalo dynamickým. Potom ale by bolo potrebné pole zase realokovať, čo by spomalilo pridávanie do slovníka a tiež hľadanie, pretože by sa teraz nemohlo hľadať tak, že by sa daný symbol použil priamo ako index do poľa referencií, ale musel by sa symbol porovnať v najhoršom prípade so všetkými referenciami v poli, pretože nie je zaručené, že sú pridávané vo vzostupnom poradí.

Existuje viacero riešení, ako tento problém s pamäťou vyriešiť, ale pre účely jednoduchosti sa bude hovoriť len o jednom riešení, ktoré je asi najlepším kompromisom medzi úsporou pamäte a rýchlosťou vyhľadávania a pridávania fráz. Toto riešenie publikoval *Juha Nieminen* v roku 2007 [5]. Juha tento problém vyriešil tak, že namiesto toho, aby každý uzol obsahoval 256 referencií, bude uzol obsahovať len 3 referencie na špecifické 3 uzly. Prvá referencia odkazuje na ľubovoľnú frázu, ktorej je súčasný uzol prefixom, druhá referencia odkazuje na uzol s rovnakým prefixom ako súčasná fráza, ale so symbolom menším a tretia referencia odkazuje na uzol s rovnakým prefixom ako súčasná fráza, ale so symbolom väčším. Čiže uloženie referencií je vyriešené použitím binárneho vyhľadávacieho stromu. Na druhú stranu je teraz vyhľadávanie fráz o jeden symbol dlhších o trochu pomalšie kvôli tomu, že sa teraz symbol nemôže použiť priamo ako index do poľa referencií ako predtým, ale musí sa hľadať v strome v najlepšom prípade v čase $\theta(\log(n))$ a v najhoršom prípade v čase $\theta(n)$, keďže sa nejedná o vyvážený binárny strom. Ale v porovnaní s dynamickým poľom referencií, spomenutým predtým, je to stále celkovo rýchlejšie.

2.1.5.2 Dekompresný slovník

Dekompresný algoritmus, narozdiel od toho kompresného, nepotrebuje testovať existenciu fráz v slovníku, ale vyžaduje rýchle vyhľadávanie fráz na základe ich indexov, schopnosť zrekonštruovať znenie celej frázy a pridať frázy do slovníka. Implementácia je možná použitím niečoho podobného, ako tomu bolo pri kompresnom slovníku, ale nie je potreba pre uzly si pamätať referencie na frázy o symbol dlhšie, keďže nie je vyžadované testovanie existencie fráz. Kompresný algoritmus si stále nájde najdlhšiu frázu G, ktorá je rovná reťazcu zo vstupu a k nej pridá symbol, ktorý tvorí frázu, ktorá sa v slovníku ešte nenachádza. Týmto spôsobom algoritmus stále pridáva frázy, ktoré ešte neexistujú, čo eliminuje potrebu pre dekompresný algoritmus kontrolovať, či frázy, ktoré pridáva, už existujú v slovníku. Čo si uzly budú musieť pamätať sú referencie na svoje prefixy a to z dôvodu, aby bol algoritmus schopný zrekonštruovať znenie celej frázy. Keďže index na frázu do slovníka odkazuje na jej posledný symbol, bude musieť algoritmus sledovať tieto prefixy až do koreňa a potom zapísať danú frázu na výstup v opačnom poradí. Samotné uzly je postačujúce ukladať postupne do poľa, ktoré sa alokuje na maximálny počet uzlov, ktoré slovník môže obsahovať. Tým sa zabráni zbytočným realokáciam. Potom je možné dané frázy získať jednoduchým indexovaním v tomto poli na základe indexov zo vstupu.

2.1.5.3 Naplnenie slovníka

Ako už bolo predtým naznačené, slovník má svoju maximálnu veľkosť, do ktorej môže rásť. Dôvodom je to, že pamäť počítača nie je neobmedzená, a tak nemôže slovník rásť donekonečna. Je dobré si maximálnu veľkosť slovníka

odvodiť ako nejakú mocninu dvoch, a to tak, že sa stanoví maximálna bitová dĺžka indexu do slovníka. Veľkosť slovníka sa tak odvodí ako 2^n, kde n je počet bitov. Dôvodom prečo sa maximálna veľkosť neodvodzuje z veľkosti operačnej pamäte je ten, že rôzne stroje môžu mať iné veľkosti pamätí a tak by na strojoch s malou pamäťou nemohol slovník narásť do takej veľkosti ako na strojoch s väčšou pamäťou.

Existuje veľa stratégií, ako vyriešiť situáciu po naplnení slovníka, ale jednými z tých najjednoduchších sú vyčistenie slovníka, jeho zmrazenie alebo kombinácia oboch.

Zmrazenie slovníka je to najrýchlejšie riešenie, pretože po naplnení sa frázy už do slovníka pridávať nemusia a tak jediné čo algoritmus vykonáva je len overovanie existencií fráz v slovníku. Na druhú stranu, by to mohlo v niektorých prípadoch zvýšiť kompresný pomer, ako napríklad vo vstupe, ktorý na začiatku obsahuje anglický text a potom hneď za tým slovenský. V tomto prípade by sa slovník na začiatku naplnil anglickými frázami, ale potom, ako by sa vstup dostal k slovenskému textu, by sa v slovníku pravdepodobne nenašli žiadne dlhšie frázy a tak by kompresný pomer mohol byť vyšší.

Vyčistenie slovníka by tento problém mohlo vyriešiť, pretože by sa po komprimovaní anglických fráz vyčistil a naplnil tými slovenskými. To by malo dobrý efekt na kompresný pomer. Na druhú stranu by mohli existovať aj prípady, kedy vyčistenie slovníka nemá dobrý efekt na kompresný pomer, z dôvodu, že by vstup obsahoval dáta rovnakého typu. Napríklad v prípade vstupu plného anglického textu, by po vyčistení slovníka, mohol kompresný pomer byť o trochu vyšší, pretože by algoritmus začal so svojím slovníkom stále od znova po určitých úsekoch a chvíľu by trvalo, dokým by sa slovník znovu naplnil frázami a začali sa znova kódovať dlhšie frázy. V tomto prípade by zmrazenie slovníka dosahovalo lepších výsledkov.

Kombinácia týchto dvoch spôsobom by teoretický mala byť schopná dosiahnuť dobrých výsledkov pri obidvoch spomenutých príkladoch, pretože keby algoritmus zistil, že sa mu nedarí kódovať dostatočne dlhé frázy so súčasným slovníkom, tak by ho zmazal a skúsil ho naplniť znova. Samozrejme problém je v tom, ako niečo také zistiť. Existuje veľa možností ako toho dosiahnuť, ale jeden zo základných spôsobov by postupne meral kompresný pomer a keby sa prekročila nejaká jeho hranica, tak by sa slovník vyčistil.

Kapitola 3

Kompresné metódy LZMW, LZAP, LZY

Kompresná metóda LZW je síce rýchla, ale na druhej strane nemá veľký potenciál dosahovať nízkeho kompresného pomeru. Je to z toho dôvodu, že sa do slovníka pridáva stále len po jednej fráze, ktoré rastú len o jeden symbol. Z toho vyplýva, že adaptácia k vstupným dátam je pomalšia a väčšinou sa slovník plní úplne nezmyselnými frázami, ktoré napríklad pri kompresii prirodzeného jazyka nie sú veľmi užitočné. Varianty tejto metódy, ktoré sa predstavia v tejto kapitole, sa snažia tento problém aspoň v menšej miere odstrániť. Pokúšajú sa to dosiahnuť tým, že pridávajú viac než jednu frázu naraz a frázy rastú po viacerých symboloch než jeden. To znamená, že sa slovník k vstupným dátam bude adaptovať rýchlejšie, čo by malo pomôcť znížiť kompresný pomer, ale na druhej strane to tiež znamená, že dátová reprezentácia slovníka a/alebo algoritmus samotný budú zložitejší, čo má nepriaznivý vplyv na rýchlosť algoritmu a tiež na jeho pamäťovú zložitosť.

3.1 Kompresná metóda LZMW

LZMW je kompresná metóda, ktorú vyvinuli *V. Miller* a *M. Vegman* v roku 1985 [6]. Túto metódu definovali na dvoch princípoch:

- Keď sa slovník naplní, odstráni sa z neho najstaršia použitá fráza. Existuje viacero spôsobom ako toto dosiahnuť, ale jeden odporúčaný spôsob je identifikovať v slovníku frázy, ktoré nie sú prefixom žiadnej inej frázy, čo znamená, že od momentu, kedy sa vložili do slovníka, nebolo nič k ním zreťazené a tak neboli vôbec použité. Z toho tiež vyplýva, že v strome tieto frázy nemajú žiadnych potomkov a tak ich odstránenie nebude narušovať žiadne iné frázy, ktoré by v strome za nimi boli zreťazené. Nakoniec z týchto identifikovaných fráz sa odstráni tá, ktorá bola pri-

daná najskôr. Prvých 256 fráz, pridaných pri inicializácii, by sa odstrániť nikdy nemali.

- Každá fráza pridaná do slovníka je zreťazenie dvoch reťazcov. Prvý reťazec je ten, ktorý sa predošle našiel v slovníku (F v pseudo-kóde) a druhý je ten súčasný, ktorý sa našiel teraz (G v pseudo-kóde).

3.1.1 Algoritmus kompresie

Algoritmus začína inicializáciou slovníka presne ako pri metóde LZW. Potom sa snaží nájsť najdlhší reťazec zo vstupu, ktorý sa nachádza v slovníku. Táto fráza sa označí písmenom G. Po jej nájdení sa do slovníka pridá fráza F+G, kde F je fráza, ktorá bola nájdená v slovníku predošle. Vzápätí sa index frázy G zapíše na výstup a fráza G je zapamätaná ako predošlá fráza F pre ďalší krok. Táto sekvencia krokov sa opakuje do konca vstupu. Na začiatku je fráza F rovná prázdnemu reťazcu. Pred pridávaním do slovníka si algoritmus musí tiež overiť, že fráza F+G sa v slovníku ešte nenachádza. Tým, že sa do slovníka pridáva zreťazenie predošlej a súčasnej nájdenej frázy, je algoritmus schopný pridávať viac zmysluplné frázy, ktoré môžu aj napríklad tvoriť celé slová v prípade, že vstup je prirodzený jazyk.

Presné kroky algoritmu sú vypísané v pseudo-kóde 3 a jeho beh je simulovaný v príklade 3.1.

Algorithm 3 Pseudo-kód kompresného algoritmu metódy LZMW

```
1: inicializuj slovník s prvými 256 frázami
2: F ← ϵ
3: G ← ϵ
4: while vstup nie je na konci do
5:     načítaj symbol B zo vstupu
6:     if fráza G+B sa nenachádza v slovníku then
7:         if fráza F+G sa nenachádza v slovníku then
8:             pridaj frázu F+G do slovníku
9:         zapíš index I frázy G na výstup
10:        F ← G
11:        G ← B
12:    else
13:        G ← G + B
14: zapíš index I frázy G na výstup
```

Tiež je potreba podotknúť, že pri implementácii tohoto algoritmu sa musí dávať pozor na fakt, že sa nezaručuje existencia všetkých prefixov frázy G v slovníku ako to bolo pri metóde LZW. Keďže sa fráza G v slovníku hľadá postupne, symbol po symbole, môže nastať situácia, že nastane len čiastočná

zhoda a tak sa algoritmus bude musieť vrátiť späť vo vstupnom reťazci (backtracking).

Napríklad, keby slovník obsahoval frázy „abc“, „abcdef“ a vstupný reťazec by bol „abcdeg“, tak by fráza G bola nastavená ako „abc“. Keďže sa ale v slovníku nenachádzajú všetky prefixy, tak by algoritmus pokračoval porovnávať vstupný reťazec až po písmeno „f“ vo vstupnom reťazci, kde by zistil, že fráza G dlhšia ako „abc“ neexistuje a tak pri hľadaní ďalšej frázy G by sa musel vrátiť naspäť a začať znova od písmena „d“.

3.1.2 Príklad kompresie

Vstup: Reťazec „yabbadabbadabbadoo“ v ASCII kódovaní.

Tabuľka 3.1: Príklad použitia kompresného algoritmu metódy LZMW

Krok	**F**	**G**	**B**	**G+B v slovníku?**	**I**	**Fráza pridaná do slovníka**
1	ϵ	ϵ	y	Áno	-	-
2	ϵ	y	a	Nie	121	-
3	y	a	b	Nie	97	ya(256)
4	a	b	b	Nie	98	ab(257)
5	b	b	a	Nie	98	bb(258)
6	b	a	d	Nie	97	ba(259)
7	a	d	a	Nie	100	ad(260)
8	d	a	b	Áno	-	-
9	d	ab	b	Nie	257	dab(261)
10	ab	b	a	Áno	-	-
11	ab	ba	d	Nie	259	abba(262)
12	ba	d	a	Nie	100	bad(263)
13	d	a	b	Áno	-	-
14	d	ab	b	Nie	257	-
15	ab	b	a	Áno	-	-
16	ab	ba	d	Áno	-	-
17	ab	bad	o	Nie	263	abbad(264)
18	bad	o	o	Nie	111	bado(265)
19	o	o	-	-	111	-

Výstup: Postupnosť indexov (121, 97, 98, 98, 97, 100, 257, 259, 100, 257, 263, 111, 111).

3.1.3 Algoritmus dekompresie

Algoritmus je veľmi priamočiary. Funguje podobne ako jeho kompresná varianta. Začína sa opäť inicializáciou slovníka. Potom sa prečíta index I frázy G zo vstupu a fráza G sa zapíše na výstup. Vzápätí sa do slovníka pridá fráza F+G, kde fráza F hraje rolu frázy G z predošlého kroku. Fráza G sa zapamätá ako fráza F pre ďalší krok a to všetko sa opakuje dokým algoritmus prečíta všetky indexy na vstupe. Samozrejme je fráza F nastavená počiatočne na prázdny reťazec. A aj tu je potrebné, aby sa v slovníku nenachádzali žiadne duplicitné frázy.

Kroky algoritmu je možné lepšie pochopiť zo pseudo-kódu 4 a z príkladu jeho behu 3.2.

Algorithm 4 Pseudo-kód dekompresného algoritmu metódy LZMW

```
1: inicializuj slovník s prvými 256 frázami
2: F ← ϵ
3: while vstup nie je na konci do
4:     načítaj index I zo vstupu
5:     načítaj frázu G na indexe I zo slovníka
6:     zapíš frázu G na výstup
7:     if fráza F+G sa nenachádza v slovníku then
8:         pridaj frázu F+G do slovníka
9:     F ← G
```

Ako je lepšie vidieť zo pseudo-kódu 4, dekompresný algoritmus netrpí problémom, ktorým trpela jeho kompresná varianta. To znamená, že sa algoritmus nemusí vracať naspäť vo vstupnom reťazci, pretože frázu G si dokáže okamžite zistiť potom ako prečíta jej index I zo vstupu a nemusí ju tak hľadať.

3.1.4 Príklad dekompresie

Vstup: Postupnosť indexov (121, 97, 98, 98, 97, 100, 257, 259, 100, 257, 263, 111, 111).

Tabuľka 3.2: Príklad použitia dekompresného algoritmu metódy LZMW

Krok	I	F	G	Výstup	Fráza pridaná do slovníka
1	121	ϵ	y	y	-
2	97	y	a	a	ya(256)
3	98	a	b	b	ab(257)
4	98	b	b	b	bb(258)
5	97	b	a	a	ba(259)
6	100	a	d	d	ad(260)
7	257	d	ab	ab	dab(261)
8	259	ab	ba	ba	abba(262)
9	100	ba	d	d	bad(263)
10	257	d	ab	ab	-
11	263	ab	bad	bad	abbad(264)
12	111	bad	o	o	bado(265)
13	111	o	o	o	oo(266)

Výstup: Reťazec „yabbadabbadabbadoo“ v ASCII kódovaní

3.1.5 Dátová reprezentácia slovníka

3.1.5.1 Kompresný slovník

Nároky kompresného slovníka metódy LZMW sú veľmi podobne nárokom pri metóde LZW, až na to, že pri metóde LZMW môžu frázy narastať o viac než jeden symbol, kvôli tomu, že sa pridávajú zreťazenia dvoch celých fráz. To tiež znamená, že teraz nebude každý uzol plnohodnotnou frázou a kvôli tomu je potrebné, aby si algoritmus dával pozor na to, či identifikačné číslo uzla je platné alebo nie. Dôvod, prečo sa algoritmus musí vracať naspäť vo vstupnom reťazci, je presne kvôli tomuto faktu, pretože algoritmus na to, aby našiel frázu G musí nájsť uzol s platným identifikačným číslom a v prípade, že sa mu nakoniec nepodarí úspešne porovnať všetky symboly až po takýto uzol, sa bude musieť vrátiť vo vstupe na ten symbol, v ktorom sa mu podarilo nájsť uzol s platným identifikačným číslom a to považovať za najdlhšiu možnú frázu G. Potom bude musieť pri hľadaní ďalšej frázy G porovnávať symboly, na ktoré už predtým narazil, keď sa snažil hľadať predošlú frázu G, čo beh algoritmu trochu spomaľuje.

3.1.5.2 Dekompresný slovník

Pri metóde LZW, mohol dekompresný algoritmus do slovníka pridávať frázu bez toho, aby sa najprv uistil, že daná fráza sa v ňom už náhodou nenachádza. Mohol to tak robiť, kvôli tomu, pretože kompresný algoritmus sa stále snažil do slovníka vložiť len unikátne frázy. V prípade metódy LZMW, to tak bohužiaľ nie je. Kompresný algoritmus sa môže do slovníka snažiť pridávať aj frázy, ktoré tam už existujú a v tom prípade sa fráza do slovníka nepridá. Ale keďže dekompresný algoritmus nevie, ktoré tieto frázy sú, musí sa najprv uistť, podobne ako kompresný algoritmus, že daná fráza v slovníku neexistuje predtým ako sa tam pridá. A tak dekompresný algoritmus od svojho slovníka tiež vyžaduje, aby bol schopný testovať existenciu fráz. Kvôli tomu je potrebné, aby dekompresný algoritmus použil slovník používaný jeho kompresnou variantou.

Uzly tohoto slovníka si tiež potrebujú naviac pamätať referenciu na prefix a tiež sú tieto uzly ukladané do poľa, aby bolo možné ich rýchle vyhľadávanie pomocou indexov do slovníka, presne ako to bolo pri dekompresnom slovníku metódy LZW. Je potrebné tiež poznamenať, že do tohoto poľa sa nebudú ukladať všetky nové uzly, ale len tie, ktoré majú platný príznak, čo znamená, že sú plnohodnotnými frázami. To tiež platí aj pre uzly, ktoré platný príznak predtým nemali, ale ho práve získali. Pri kompresnom slovníku bol tento príznak reprezentovaný platnosťou identifikačného čísla, ale keďže dekompresný slovník tieto čísla nepotrebuje, je postačujúce tento príznak reprezentovať jedným bitom.

3.2 Kompresná metóda LZAP

Metóda LZAP je rozšírenie metódy LZMW. Túto metódu publikoval J. A. Storer v roku 1988 [7]. LZMW mala problém v tom, že sa kompresný algoritmus musel stále vracať späť vo vstupnom reťazci kvôli tomu, že slovník neobsahoval všetky prefixy svojich fráz. LZAP to vyriešil tak, že namiesto pridávania zreťazenia predošlej frázy F a súčasnej frázy G sa do slovníka pridá zreťazenie frázy F so všetkými prefixmi frázy G. To znamená, že slovník bude obsahovať všetky prefixy svojich fráz a tak nie je nutné pre kompresný algoritmus sa vracať späť vo vstupe pri hľadaní frázy G. V tomto zmysle je LZAP celkom podobný LZW, len s tým rozdielom, že do slovníka pridáva frázy iným spôsobom. Dokonca aj „AP“ v názve tejto metódy je skratka pre „All Prefixes“ (všetky prefixy).

3.2.1 Algoritmus kompresie

Algoritmus kompresie je identický s algoritmom kompresie metódy LZMW, až na krok pridávania do slovníka. Tento krok je iný v tom, že sa do slovníka nepridá fráza F+G, ale pridajú sa zreťazenia frázy F so všetkými prefixmi frázy G. Čiže keby fráza F=„lzmw“ a fráza G=„lzap“, tak by sa do slovníka

pridali frázy: „lzmwl“, „lzmwlz“, „lzmwlza“, „lzmwlzap“. Na poradí, v ktorom by sa mali tieto frázy pridať do slovníka teoreticky nezáleží, ale odporúčané poradie je také ako v príklade vyššie, kvôli tomu ako je slovník v skutočnosti implementovaný.

Pre lepšie pochopenie je tu pseudo-kód 5.

Algorithm 5 Pseudo-kód kompresného algoritmu metódy LZAP

```
1: inicializuj slovník s prvými 256 frázami
2: F ← ϵ
3: G ← ϵ
4: while vstup nie je na konci do
5:     načítaj symbol B zo vstupu
6:     if fráza G+B sa nenachádza v slovníku then
7:         zapíš index I frázy G na výstup
8:         for každý symbol C frázy G do
9:             F ← F + C
10:            if fráza F sa nenachádza v slovníku then
11:                pridaj frázu F do slovníka
12:        F ← G
13:        G ← B
14:    else
15:        G ← G + B
16: zapíš index I frázy G na výstup
```

Zo pseudo-kódu 5 sa môže odpozorovať, že pri pridávaní nových fráz do slovníka, algoritmus musí prejsť všetkými symbolmi frázy G, od začiatku do konca. Priamo zo pseudo-kódu to nemusí byť hneď zrejmé, ale implementácia tohoto algoritmu si v skutočnosti v cykle nepamätá presné znenie frázy G, ale len ukazovateľ na jej uzol. A keďže tento algoritmus tiež používa dátovú štruktúru slovníka veľmi podobnú so štruktúrami slovníkov pri predošlých dvoch metódach, je známe, že ukazovateľ na uzol frázy G v skutočnosti odkazuje na posledný symbol frázy G. To znamená, že by sa fráza G musela najprv prečítať zo slovníka v opačnom poradí, uložiť do medzi-pamäte a až potom ju použiť pri pridávaní nových fráz do slovníka.

Spôsob, ako sa vyhnúť zbytočnému čítaniu frázy G zo slovníka v čase $\theta(n)$, je ten, že sa znenie frázy G uloží do medzi-pamäte hneď pri tom ako sa fráza G hľadá v slovníku. Je to možne preto, pretože fráza G je sekvencia samotných symbolov B zo vstupu a tak sa tieto symboly zapamätajú bokom do medzi-pamäte pre následné použitie pri pridávaní do slovníka.

Toto vylepšenie je možné vidieť v pseudo-kóde 6.

V pseudo-kóde 6 sa znenie frázy G zapamätá do medzi-pamäte H a tak sa pri pridávaní dá v implementácii vyhnúť zbytočnému čítaniu frázy G zo slovníka.

Algorithm 6 Pseudo-kód kompresného algoritmu metódy LZAP (1. vylepšenie)

```
1: inicializuj slovník s prvými 256 frázami
2: F ← ϵ
3: G ← ϵ
4: H ← ϵ
5: while vstup nie je na konci do
6:     načítaj symbol B zo vstupu
7:     if fráza G+B sa nenachádza v slovníku then
8:         zapíš index I frázy G na výstup
9:         for každý symbol C z medzi-pamäte H do
10:            F ← F + C
11:            if fráza F sa nenachádza v slovníku then
12:                pridaj frázu F do slovníka
13:        F ← G
14:        G ← B
15:        H ← B
16:    else
17:        G ← G + B
18:        H ← H + B
19: zapíš index I frázy G na výstup
```

Fakt, že sa do slovníka pridáva až potom ako je fráza G nájdená, nebol negatívny pre výkon kompresného algoritmu metódy LZW z toho dôvodu, že sa pridávala len jedna fráza. Ale pri kompresnom algoritme metódy LZAP sa do slovníka pridáva viacero fráz naraz a kvôli, tomu sa v cykle musí prejsť celá nájdená fráza G ešte raz, potom ako ju algoritmus prečítal prvý-krát, keď ju ešte len hľadal. Čiže sa musí prečítať fráza G dva-krát, namiesto jeden-krát.

Riešením ako toto opakované čítanie frázy G odstrániť by mohlo byť to, že by sa fráza G používala pre pridávanie do slovníka popritom ako by sa ešte len hľadala. Čiže by sa v pseudo-kóde 6, cyklus na riadku 9 zlúčil s hlavným cyklom na riadku 5 a vstupný symbol B by sa použil namiesto symbolu C z riadku 10, keďže ako už bolo predtým spomenuté, je fráza G samotnou sekvenciou symbolov zo vstupu. Je potrebné podotknúť, že teraz je medzi-pamäť H úplne zbytočná, pretože znenie frázy G dostaneme priamo zo vstupu.

Vyššie popísané skutočnosti sú naznačené v pseudo-kóde 7.

Týmto spôsobom teraz algoritmus číta frázu G len raz a tým je tento pseudo-kód 7 oproti predošlej verzii pseudo-kódu 6 rýchlejší, ale pseudo-kód 7 má stále ešte jeden malý problém. A to ten, že si algoritmus pod rukami mení slovník, v ktorom ešte len zisťuje existenciu frázy G. Tento fakt samotnému kompresnému algoritmu nevadí, ale vadiť to bude tomu dekompresnému a to z toho dôvodu, že nové prídavky, ktoré kompresný algoritmus vykoná počas

Algorithm 7 Pseudo-kód kompresného algoritmu metódy LZAP (neúplné 2. vylepšenie)

```
1: inicializuj slovník s prvými 256 frázami
2: F ← ϵ
3: G ← ϵ
4: while vstup nie je na konci do
5:     načítaj symbol B zo vstupu
6:     if fráza G+B sa nenachádza v slovníku then
7:         zapíš index I frázy G na výstup
8:         F ← G + B
9:         G ← B
10:    else
11:        G ← G + B
12:        F ← F + B
13:    if fráza F sa nenachádza v slovníku then
14:        pridaj frázu F do slovníka
15: zapíš index I frázy G na výstup
```

hľadania frázy G by mohli ovplyvniť jej samotné hľadanie a tak by mohol algoritmus vyprodukovať index na výstup, ktorý dekompresný algoritmus vo svojom slovníku pred čítaním indexu frázy G ešte nemôže obsahovať. Preto je potrebné zaistiť, aby tieto nové prídavky neovplyvnili hľadanie frázy G a to tak, že algoritmus bude mať k dispozícii dve verzie toho istého slovníka. Prvá verzia je tá, do ktorej algoritmus práve pridáva nové frázy a tá druhá je tá, ktorú algoritmus používa pre hľadanie frázy G, čiže verzia obsahujúca stav slovníka predtým, ako sa do neho začali pridávať nové frázy. Potom ako algoritmus vyprodukuje index frázy G na výstup, sa samozrejme druhá verzia slovníka aktualizuje z tej prvej, aby nové pridané frázy boli dostupné pre nasledujúce hľadanie ďalšej frázy G. Spôsob, akým sa môže implementovať takýto slovník s dvoma verziami, je popísaný v sekcii o dátovej štruktúre kompresného slovníka 3.2.5.1.

Pseudo-kód 8 znázorňuje vyššie uvedené opravy pseudo-kódu 7 a tiež príklad 3.3 simuluje beh pseudo-kódu 8.

Algorithm 8 Pseudo-kód kompresného algoritmu metódy LZAP (2. vylepšenie)

```
1: inicializuj slovník D1 a slovník D2 s prvými 256 frázami
2: F ← ϵ
3: G ← ϵ
4: while vstup nie je na konci do
5:     načítaj symbol B zo vstupu
6:     if fráza G+B sa nenachádza v slovníku D2 then
7:         zapíš index I frázy G na výstup
8:         D2 ← D1
9:         F ← G + B
10:        G ← B
11:    else
12:        G ← G + B
13:        F ← F + B
14:    if fráza F sa nenachádza v slovníku D1 then
15:        pridaj frázu F do slovníka D1
16: zapíš index I frázy G na výstup
```

Slovníky D1 a D2 v pseudo-kóde 8 sú v skutočnosti ten istý slovník, ale pre účely názornosti a všeobecnosti sa tieto dve verzie slovníka považujú v pseudo-kóde ako dva rozdielne slovníky.

3.2.2 Príklad kompresie

Vstup: Reťazec „yabbadabbadabbadoo“ v ASCII kódovaní.

Tabuľka 3.3: Príklad použitia kompresného algoritmu metódy LZAP

Krok	**G**	**B**	**G+B v D2?**	**I**	**Frázy pridané do D2**	**F**	**Fráza pridaná do D1**
1	ϵ	y	Áno	-	-	y	-
2	y	a	Nie	121	-	ya	ya(256)
3	a	b	Nie	97	ya(256)	ab	ab(257)
4	b	b	Nie	98	ab(257)	bb	bb(258)
5	b	a	Nie	98	bb(258)	ba	ba(259)
6	a	d	Nie	97	ba(259)	ad	ad(260)
7	d	a	Nie	100	ad(260)	da	da(261)
8	a	b	Áno	-	-	dab	dab(262)
9	ab	b	Nie	257	da(261), dab(262)	abb	abb(263)
10	b	a	Áno	-	-	abba	abba(264)
11	ba	d	Nie	259	abb(263), abba(264)	bad	bad(265)
12	d	a	Áno	-	-	bada	bada(266)
13	da	b	Áno	-	-	badab	badab(267)
14	dab	b	Nie	262	bad(265), bada(266), badab(267)	dabb	dabb(268)
15	b	a	Áno	-	-	dabba	dabba(269)
16	ba	d	Áno	-	-	dabbad	dabbad(270)
17	bad	o	Nie	265	dabb(268), dabba(269), dabbad(270)	bado	bado(271)
18	o	o	Nie	111	bado(271)	oo	oo(272)
19	o	-	-	111	-	-	-

Výstup: Postupnosť indexov (121, 97, 98, 98, 97, 100, 257, 259, 262, 265, 111, 111).

3.2.3 Algoritmus dekompresie

Dekompresný algoritmus to má ako obvykle oveľa jednoduchšie ako ten kompresný. Funguje podobne ako dekompresný algoritmus pri metóde LZMW, až na to, že po zápise frázy G na výstup ju následne tiež použije spolu s frázou F pri pridávaní nových fráz do slovníka, tak ako to bolo vysvetlené pri kompresnej variante tohoto algoritmu.

Je potrebné tiež podotknúť, že dekompresný algoritmus nepotrebuje používať dve rozdielne verzie slovníka, pretože frázu G je možné priamo zistiť na základe jej indexu I a nemusí sa tak hľadať vo vstupnom reťazci, ako tomu bolo pri kompresnom algoritme.

Kroky tohoto algoritmu je možné vidieť v pseudo-kóde 9 a jeho funkcionalitu otestovanú v príklade 3.4.

Algorithm 9 Pseudo-kód dekompresného algoritmu metódy LZAP

```
1:  inicializuj slovník s prvými 256 frázami
2:  F ← ϵ
3:  while vstup nie je na konci do
4:      načítaj index I zo vstupu
5:      načítaj frázu G na indexe I zo slovníka
6:      zapíš frázu G na výstup
7:      for každý symbol C frázy G do
8:          F ← F + C
9:          if fráza F sa nenachádza v slovníku then
10:             pridaj frázu F do slovníku
11:     F ← G
```

3.2.4 Príklad dekompresie

Vstup: Postupnosť indexov (121, 97, 98, 98, 97, 100, 257, 259, 262, 265, 111, 111).

Tabuľka 3.4: Príklad použitia dekompresného algoritmu metódy LZAP

Krok	I	F	G	Výstup	Frázy pridané do slovníka
1	121	ϵ	y	y	-
2	97	y	a	a	ya(256)
3	98	a	b	b	ab(257)
4	98	b	b	b	bb(258)
5	97	b	a	a	ba(259)
6	100	a	d	d	ad(260)
7	257	d	ab	ab	da(261), dab(262)
8	259	ab	ba	ba	abb(263), abba(264)
9	262	ba	dab	dab	bad(265), bada(266), badab(267)
10	265	dab	bad	bad	dabb(268), dabba(269) dabbad(270))
11	111	bad	o	o	bado(271)
12	111	o	o	o	oo(272)

Výstup: Reťazec „yabbadabbadabbadoo“ v ASCII kódovaní

3.2.5 Dátová reprezentácia slovníka

3.2.5.1 Kompresný slovník

Keďže z analýzy algoritmu je známe, že slovník v tomto prípade bude obsahovať frázy a všetky ich prefixy, je možne pre implementáciu slovníka použiť presne ten istý slovník, ako pri kompresnom algoritme metódy LZW ale s menšou modifikáciou. Tá modifikácia spočíva v tom, že rýchla implementácia LZAP vyžaduje, aby si uchovávala dve verzie slovníka pre súčasné pridávanie nových fráz a hľadanie frázy G. Možnosť, ako dosiahnuť to, aby nové frázy, ktoré boli pridané do slovníka pri hľadaní frázy G sa tvárili súčasne, ako keby pridané ešte neboli a tak neovplyvnili jej samotne hľadanie, je to, že si slovník bude pamätať svoj vek a bude si ho zväčšovať o jedničku, stále potom ako algoritmus na výstup zapíše index I frázy G. Každá fráza v slovníku si tiež pamätá vek slovníka, v ktorom bola pridaná a tak, keď vek, ktorý si pamätá fráza je rovný veku slovníka, t.j fráza bola pridaná práve pri hľadaní súčasnej

frázy G, bude algoritmus predstierať, že táto fráza v slovníku ešte neexistuje. Potom keď sa na výstup zapíše index I frázy G, sa vek slovníka zväčší o jedna. Takto frázy pridané počas hľadania poslednej frázy G sa pre ďalšie hľadanie budú tváriť, ako keby už v slovníku boli. Týmto spôsobom môže algoritmus rozdeliť frázy na dve časti: frázy, ktorých vek je aspoň o jedna menší ako vek slovníka a frázy, ktorých vek je rovný veku slovníka. Algoritmus bude pre hľadanie frázy G využívať len tú prvú časť a druhú časť potom neskôr pridá do prvej časti len jednoduchým pripočítaním jedničky k veku slovníka.

Pri tomto riešení môže problém nastať v prípade, že toto číslo je reprezentované fixným počtom bitov a tak sa môže potencionálne naskytnúť „overflow“. To by spôsobilo frázam, ktoré by už mali byť považované za pridané, že budú považované za nepridané. Tento problém by sa napríklad mohol vyriešiť tým, že by vek nebol reprezentovaný fixným počtom bitov, ale dynamickým počtom, takže by sa z neho stalo číslo s ľubovoľným rozsahom. To by ale samozrejme vyžadovalo viac pamäte.

Iným spôsobom by mohlo byť to, že by algoritmus dával pozor na to, kedy „overflow“ nastane, a tak by vek slovníka nastavil znova na najmenšiu možnú hodnotu a to isté by spravil aj so všetkými uzlami v slovníku. Funguje to preto, pretože slovníku záleží len na tom, že pridaná fráza musí mať vek aspoň o jedna menší ako súčasný vek slovníka, takže nevadí, že tieto hodnoty budú resetované. Nevýhoda tohoto riešenia je, že by to mohlo trochu spomaliť beh algoritmu.

V skutočnosti tieto riešenia ani nie sú potrebné, pretože sa nesmie zabúdať na to, že slovník je limitovaný maximálnym počtom fráz, ktoré môže obsahovať. Jediné, čo je potrebné spraviť, je to, že sa bitová dĺžka veku nastaví na takú hodnotu, aby „overflow“ nikdy nenastal, pretože dovtedy by sa slovník už vyčistil, pri použití stratégie vyčistenia slovníka, t.j. by sa vek už dávno nastavil naspäť na minimálnu hodnotu. Pri vyčistení slovníka nie je potrebné tento vek každému uzlu prepisovať, takže to ani nespomalí beh algoritmu.

3.2.5.2 Dekompresný slovník

Pri dekompresnom slovníku sa situácia veľmi nemení oproti tomu kompresnému, pretože algoritmus, podobne ako pri metóde LZMW, nezaručuje unikátnosť pridaných fráz, a tak je potrebné aj pre dekompresný algoritmus ich existenciu testovať. Kvôli tomu sa tu musí využiť kompresný slovník, ale nie je už potrebné uchovávať dva verzie, pretože fráza G sa na vstupe nemusí hľadať, je k dispozícii priamo jej index.

Podobne ako pri predchádzajúcich dekompresných algoritmoch, slovník tiež potrebuje ukladať uzly do poľa pre rýchle vyhľadávanie na základe indexov a každý uzol si tiež potrebuje uchovávať referenciu na prefix pre spätnú rekonštrukciu fráz.

3.3 Kompresná metóda LZY

Metóda LZW a LZAP sa snažila zachovať štruktúru svojho slovníka v takom stave, že keby sa z neho vybrala arbitrárna fráza v ľubovoľnom kroku algoritmu, tak by bolo možné tiež v slovníku nájsť všetky frázy, ktoré by tvorili jej prefix. Metóda LZY na to ide podobne, ale ešte sa dodatočne snaží zaručiť, že slovník súčasne obsahuje všetky sufixy každej frázy. Kompresnú metódu LZY vytvoril *Dan Bernstein* [8]. „Y“ v názve je odvodené od reťazca „Yabba“, ktorý pochádza zo vstupného textu originálne použitého pri testovaní algoritmu.

3.3.1 Algoritmus kompresie

Algoritmus začína inicializáciou slovníka identickou s predošlými metódami. Potom sa v cykle stále na začiatku prečíta zo vstupu symbol B, ktorý sa postupne reťazí do frázy G, dokým sa reťazec G+B v slovníku nenachádza. Takto si algoritmus nájde najdlhší reťazec zo vstupu, ktorý existuje v slovníku v podobe frázy G. Potom ako sa index frázy G zapíše na výstup, nasleduje, ako v predošlých metódach, fáza pridávania do slovníka. Ako už bolo spomenuté, metóda LZY sa do slovníka snaží pridávať frázy, tak, aby zaistila, že sa v slovníku nachádzajú tiež aj všetky ich sufixy. Kvôli tomu algoritmus používa ešte jednu pomocnú frázu F, ktorá je inicializovaná na úplnom začiatku na prázdny reťazec. Postupne sa na koniec frázy F reťazia individuálne symboly z frázy G. Po každom zreťazenom symbole sa algoritmus chce uistiť v tom, že fráza F a všetky jej sufixy existujú v slovníku. Dosiahne toho tak, že frázu F pridá do slovníka v prípade, že sa tam ešte nenachádza a potom zo začiatku frázy F odoberie jeden symbol a tieto dva kroky opakuje, pokým fráza F neexistuje v slovníku. Odobraním symbolu zo začiatku reťazca sa získa jeho sufix. V najhoršom prípade sa fráza F degraduje do frázy o dĺžke jedna. Existencia tejto frázy je zabezpečená inicializáciou na začiatku. Po prejdení všetkými symbolmi z frázy G, sa G nastaví na symbol B a opakuje sa to všetko do konca vstupného reťazca.

Je možné si všimnúť, že tento algoritmus, podobne ako LZAP, pridáva do slovníka viacero fráz naraz a na pridávanie sa používajú symboly z frázy G. To znamená, že sa tiež potrebuje zaistiť to, aby nové prídavky do slovníka neovplyvňovali hľadanie frázy G, čo by spôsobilo problémy pre dekompresný algoritmus, presne ako to bolo diskutované pri metóde LZAP.

Tento problém je teda možné vyriešiť tiež troma rôznymi spôsobmi:

- Nové frázy sa do slovníka pridávajú až potom, ako sa na výstup zapíše index I nájdenej frázy G.
- Podobne ako predošlý spôsob, ale fráza G je zapamätaná do medzipamäte popritom ako je hľadaná. Týmto sa potom nemusí zrekonštruovať zo slovníka, čo by zabralo čas $\theta(n)$, kde n je dĺžka frázy G.
- Použiť dve verzie toho istého slovníka, jednu do ktorej sa práve pridáva a druhú, ktorá je použitá pre hľadanie frázy G. Tak je možné pridávať do slovníka popritom, ako sa fráza G hľadá.

Z predošlej analýzy na túto problematiku nachádzajúcej sa v kapitole o metóde LZAP už je známe, že posledný uvedený spôsob je ten najlepší. Z tohto dôvodu je už priamo použitý v pseudo-kóde 10, ktorý zaznamenáva kroky tohoto algoritmu. Tiež je jeho beh simulovaný na príklade 3.5.

Algorithm 10 Pseudo-kód kompresného algoritmu metódy LZY

```
1: inicializuj slovník D1 a slovník D2 s prvými 256 frázami
2: F ← ϵ
3: G ← ϵ
4: while vstup nie je na konci do
5:     načítaj symbol B zo vstupu
6:     if fráza G+B sa nenachádza v slovníku D2 then
7:         zapíš index I frázy G na výstup
8:         G ← B
9:         D2 ← D1
10:    else
11:        G ← G + B
12:    F ← F + B
13:    while fráza F sa nenachádza v slovníku D1 do
14:        pridaj frázu F do slovníka D1
15:        F ← F[1], ..., F[F.length − 1]
16: zapíš index I frázy G na výstup
```

3.3.2 Príklad kompresie

Vstup: Reťazec „yabbadabbadabbadoo“ v ASCII kódovaní.

Tabuľka 3.5: Príklad použitia kompresného algoritmu metódy LZY

Krok	**G**	**B**	**G+B v D2?**	**I**	**Frázy pridané do D2**	**F**	**Fráza pridaná do D1**
1	ϵ	y	Áno	-	-	y	-
2	y	a	Nie	121	-	$ya \rightarrow a$	ya(256)
3	a	b	Nie	97	ya(256)	$ab \rightarrow b$	ab(257)
4	b	b	Nie	98	ab(257)	$bb \rightarrow b$	bb(258)
5	b	a	Nie	98	bb(258)	$ba \rightarrow a$	ba(259)
6	a	d	Nie	97	ba(259)	$ad \rightarrow d$	ad(260)
7	d	a	Nie	100	ad(260)	$da \rightarrow a$	da(261)
8	a	b	Áno	-	-	ab	-
9	ab	b	Nie	257	da(261)	$abb \rightarrow bb$	abb(262)
10	b	a	Áno	-	-	$bba \rightarrow ba$	bba(263)
11	ba	d	Nie	259	abb(262), bba(263)	$bad \rightarrow ad$	bad(264)
12	d	a	Áno	-	-	$ada \rightarrow da$	ada(265)
13	da	b	Nie	261	bad(264), ada(265)	$dab \rightarrow ab$	dab(266)
14	b	b	Áno	-	-	abb	-
15	bb	a	Áno	-	-	$abba \rightarrow bba$	abba(267)
16	bba	d	Nie	263	dab(266), abba(267)	$bbad \rightarrow bad$	bbad(268)
17	d	o	Nie	100	bbad(268)	$bado \rightarrow$ $ado \rightarrow$ $do \rightarrow o$	bado(269), ado(270), do(271)
18	o	o	Nie	111	bado(269), ado(270), do(271)	$oo \rightarrow o$	oo(272)
19	o	-	-	111	-	-	-

Výstup: Postupnosť indexov (121, 97, 98, 98, 97, 100, 257, 259, 261, 263, 100, 111, 111).

3.3.3 Algoritmus dekompresie

Potom ako sa inicializuje slovník, algoritmus prečíta index I zo vstupu, nájde frázu G na danom indexe a zapíše ju na výstup, podobne ako tomu bolo pri predošlých dekompresných algoritmoch. Potom frázu G tiež použije, spolu s frázou F, pri pridávaní do slovníka presne tým spôsobom, akým sa frázy pridávali pri jeho kompresnej variante. Postup sa opakuje pre všetky indexy I zo vstupu.

Podobne ako dekompresný algoritmus metódy LZAP, aj tento algoritmus nepotrebuje dve rozdielne verzie slovníka, pretože fráza G je ihneď známa z jej indexu I.

Chovanie tohoto algoritmu je znázornené v pseudo-kóde 11 a simulované v príklade 3.6.

Algorithm 11 Pseudo-kód dekompresného algoritmu metódy LZY

```
1: inicializuj slovník s prvými 256 frázami
2: F ← ϵ
3: while vstup nie je na konci do
4:     načítaj index I zo vstupu
5:     načítaj frázu G na indexe I zo slovníka
6:     zapíš frázu G na výstup
7:     for každý symbol C frázy G do
8:         F ← F + C
9:         while fráza F sa nenachádza v slovníku do
10:            pridaj frázu F do slovníka
11:            F ← F[1], ..., F[F.length − 1]
```

3.3.4 Príklad dekompresie

Vstup: Postupnosť indexov (121, 97, 98, 98, 97, 100, 257, 259, 261, 263, 100, 111, 111).

Tabuľka 3.6: Príklad použitia dekompresného algoritmu metódy LZY

Krok	**I**	**G**	**Výstup**	**F**	**Frázy pridané do slovníka**
1	121	y	y	y	-
2	97	a	a	$ya \rightarrow a$	ya(256)
3	98	b	b	$ab \rightarrow b$	ab(257)
4	98	b	b	$bb \rightarrow b$	bb(258)
5	97	a	a	$ba \rightarrow a$	ba(259)
7	100	d	d	$ad \rightarrow d$	ad(260)
8	257	ab	ab	$da \rightarrow a \rightarrow ab$	da(261)
9	259	ba	ba	$abb \rightarrow bb \rightarrow$ $bba \rightarrow ba$	abb(262), bba(263)
10	261	da	da	$bad \rightarrow ad \rightarrow$ $ada \rightarrow da$	bad(264), ada(265)
11	263	bba	bba	$dab \rightarrow ab \rightarrow abb \rightarrow$ $abba \rightarrow bba$	dab(266), abba(267)
12	100	d	d	$bbad \rightarrow bad$	bbad(268)
13	111	o	o	$bado \rightarrow$ $ado \rightarrow$ $do \rightarrow o$	bado(269), ado(270), do(271)
14	111	o	o	$oo \rightarrow o$	oo(272)

Výstup: Reťazec „yabbadabbadabbadoo“ v ASCII kódovaní.

3.3.5 Dátová reprezentácia slovníka

3.3.5.1 Kompresný slovník

Z inšpekcie kompresného algoritmu sa dá odpozorovať, že pridáva do slovníka frázy tak, že stále expanduje nejakú frázu o jeden symbol B, a tak sa dá povedať, že v slovníku sa budú nachádzať frázy a všetky ich prefixy, podobne ako pri metódach LZW a LZAP. Keďže okrem toho rýchla implementácia tiež potrebuje dva verzie slovníka, kvôli súčasnému pridávaniu a hľadaniu frázy G, využije sa pre slovník tá istá dátová štruktúra ako pri metóde LZAP, ale s jednou malou modifikáciou. Tá modifikácia je potrebná kvôli tomu, že sa dá pridávanie symbolu k fráze G vnímať ako pridávanie symbolu B k všetkým sufixom frázy G. Aby bol algoritmus schopný tieto sufixy nájsť rýchlo, po-

trebuje si každá fráza alebo uzol pamätať referenciu na frázu, ktorá tvorí jej najdlhší sufix a tiež referenciu na svoj prefix.

Takže algoritmus bude pridávať frázy tak, že si zapamätá ukazovateľ uzlu novej pridanej frázy a potom bude nasledovať referenciu sufixu prefixu daného uzla, ktorý následne rozšíri o symbol B a tak pokračuje ďalej. Algoritmus si musí dávať pozor na to, aby bola referencia sufixu platná a tiež či fráza, ktorú sa snaží pridať, už v slovníku náhodou neexistuje. V prípade, že zistí, že daný sufix rozšírený o symbol B už v slovníku existuje, môže pridávanie ukončiť, pretože už je garantované, že ďalšie sufixy by dopadli rovnako. Keďže slovník sa na začiatku inicializuje na všetky frázy dĺžky jedna, v najhoršom prípade algoritmus skončí na sufixe o dĺžke jedna. Zistiť, či je sufix fráza o dĺžke jedna, je možné tým, že majú svoju referenciu na sufix neplatnú. Tiež je nutné nezabudnúť nastaviť sufix novej frázy na novú pridanú frázu, ktorá je rozšírením frázy nachádzajúcej sa na sufixe prefixu danej frázy o symbol B.

Napríklad, keď by algoritmus rozširoval všetky sufixy frázy „ab“ o symbol „c“, najprv by pridal do slovníka frázu „abc“, potom by nasledoval sufix prefixu frázy „ab“, čiže „b“, a ten by rozšíril o symbol „c“, takže by do slovníka pridal frázu „bc“ a nastavil by sufix frázy „abc“ na frázu „bc“. Keďže fráza „c“, už je frázou o dĺžke jedna, pridávanie by skončilo, pretože frázy o dĺžke jedna nemajú platný sufix. Ale ešte predtým by sa nastavil sufix frázy „bc“ na frázu „c“.

3.3.5.2 Dekompresný slovník

Dekompresný algoritmus bude musieť použiť rovnakú dátovú štruktúru pre slovník ako kompresný algoritmus, pretože aj on pridáva frázy úplne rovnakým spôsobom. To tiež vyžaduje testovanie existencie fráz v slovníku, ale nie je potrebné používať dve verzie slovníka a to z rovnakého dôvodu, ako to bolo pri metóde LZAP. A ako obvykle, je potrebné ukladať uzly do poľa pre rýchle vyhľadávanie pomocou indexu do slovníka a uzly už majú referenciu na prefix z kompresného slovníka.

Kapitola 4

Implementácia

Cieľom tejto práce bolo implementovať kompresné metódy LZMW, LZAP a LZY do knižnice SCT. Táto kapitola sa zaoberá práve týmto procesom a vysvetľuje kroky, ktoré boli potrebné na jeho uskutočnenie.

4.1 Programovacie prostredie

Metódy boli implementované v programovacom jazyku Java, verzia 8. Dôvodom voľby tohoto jazyka je to, že samotná knižnica SCT je naprogramovaná práve v tomto jazyku. Hlavným vývojovým prostredím bol **NetBeans IDE 8.2**, kvôli dobrej podpore jazyka Java a nástrojov **Maven** a **GIT**, ktoré knižnica SCT [2] používa ako zostavovací a verzovací systém projektu.

4.2 Štruktúra súborov zdrojových kódov

Knižnica je štruktúrovaná tak, že sa pre každú novú kompresnú metódu vytvorí nový Java „package“ (balík). V prípade tejto práce sú tieto balíky dodatočne vložené ako podbalíky balíka **cz.cvut.fit**, naznačujúci fakultu, ktorá metódu implementovala. To znamená, že sa vytvorili balíky: **cz.cvut.fit.lzap**, **cz.cvut.fit.lzmw** a **cz.cvut.fit.lzy**.

Všetky kompresné metódy sú implementované pomocou 9 tried, z ktorých 6 sú umiestnené v **cz.cvut.fit.<meno metódy>**, 2 triedy sú umiestnené v **cz.cvut.fit.<meno metódy>.dictionary** a 1 trieda je umiestnená v **cz.cvut.fit.triplet.coder**.

Štruktúru balíkov spolu s názvami všetkých tried je možné odpozorovať na obrázku 4.1.

Obr. 4.1: Adresárová a súborová štruktúra implementovaných metód

- `cz.cvut.fit.lzap`
  - `LZAP.java` ..... hlavná trieda metódy LZAP
  - `LZAPClient.java` ..... konzolová aplikácia LZAP
  - `LZAPProvider.java` ..... rozhranie poskytovateľa parametrov LZAP
  - `LZAPProviderImpl.java` ..... implementácia LZAPProvider
  - `LZAPProviderParameters.java` ..... štruktúra parametrov LZAP
  - `LZAPTest.java` ..... testovacia aplikácia pre LZAP
  - `dictionary`
    - `LZAPDecodeDictionary.java` ..... dekompresný slovník LZAP
    - `LZAPEncodeDictionary.java` ..... kompresný slovník LZAP
- `cz.cvut.fit.lzmw`
  - `LZMW.java` ..... hlavná trieda metódy LZMW
  - `LZMWClient.java` ..... konzolová aplikácia LZMW
  - `LZMWProvider.java` ..... rozhranie poskytovateľa parametrov LZMW
  - `LZMWProviderImpl.java` ..... implementácia LZMWProvider
  - `LZMWProviderParameters.java` ..... štruktúra parametrov LZMW
  - `LZMWTest.java` ..... testovacia aplikácia pre LZMW
  - `dictionary`
    - `LZMWDecodeDictionary.java` ..... dekompresný slovník LZMW
    - `LZMWEncodeDictionary.java` ..... kompresný slovník LZMW
- `cz.cvut.fit.lzy`
  - `LZY.java` ..... hlavná trieda metódy LZY
  - `LZYClient.java` ..... konzolová aplikácia LZY
  - `LZYProvider.java` ..... rozhranie poskytovateľa parametrov LZY
  - `LZYProviderImpl.java` ..... implementácia LZYProvider
  - `LZYProviderParameters.java` ..... štruktúra parametrov LZY
  - `LZYTest.java` ..... testovacia aplikácia pre LZY
  - `dictionary`
    - `LZYDecodeDictionary.java` ..... dekompresný slovník LZY
    - `LZYEncodeDictionary.java` ..... kompresný slovník LZY
- `cz.cvut.fit.triplet.coder`
  - `LZAPTripletCoder.java` ..... kodér a de-kodér metódy LZAP
  - `LZMWTripletCoder.java` ..... kodér a de-kodér metódy LZMW
  - `LZYTripletCoder.java` ..... kodér a de-kodér metódy LZY

4.3 Implementované triedy

4.3.1 Trieda <meno metódy>

Táto trieda slúži ako hlavná trieda danej kompresnej metódy, ktorú je možné použiť pre vykonanie kompresie/dekompresie. Jej konštruktér prijíma inštanciu triedy **<meno metódy>ProviderParameters**, ktorá je vzápätí poslaná do inštancie triedy, ktorá implementuje rozhranie **<meno metódy> Provider**. V tomto prípade je to **<meno metódy>ProviderImpl**, ktorá sa nakonfiguruje na základe týchto parametrov a neskôr slúži na poskytovanie kodéra, ktorý vykonáva samotný kompresný/dekompresný algoritmus.

Pre účely kompresie, trieda obsahuje metódu **compress**, ktorá prijíma ako parametre inštancie tried **ByteBuffer** a **Consumer<TripletSupplier>**. Prvý parameter reprezentuje pole bajtov, ktoré sa majú kompresným algoritmom spracovať a druhý parameter reprezentuje objekt, ktorý prijíma výstupné dáta. Dôvodom prečo je tento objekt typu **Consumer** je ten, že knižnica takto môže dosiahnuť možnosť reťazenia viacero metód za seba pomocou triedy **ChainBuilder**, ktorá dovoľuje pomocou metódy **chain** zaradenie procesu do dátového toku. Analógia je podobná reťazeniu procesov v Unixom systéme pomocou „pipeliningu“. Vo vnútri metódy sa zavolá metóda **getCoder**, patriaca poskytovateľovi, ktorá vráti nakonfigurovanú inštanciu kodéra a ten je vzápätí hneď aj použitý pre vykonanie kompresného algoritmu. Pre tento účel slúži jeho metóda **encode**, ktorá príjme vstupné dáta a objekt, ktorý má prijať výstup.

Pre účely dekompresie, trieda obsahuje metódu**decompress**, ktorá príjme ako parametre inštancie tried **TripletProcessor** a **Consumer <ByteBuffer>**. Prvý parameter reprezentuje objekt, ktorý poskytuje vstupné indexy, zapísané do výstupného súboru pri kompresii, a druhý parameter reprezentuje objekt, ktorý prijíma dekódované dáta. Podobne ako pri kompresii, je tu použitý princíp reťazenia. Vo vnútri metódy je opäť zavolaný poskytovateľ, ktorý dodá vhodný nakonfigurovaný kodér, ktorého metóda **decode** je vzápätí zavolaná pre vykonanie dekompresného algoritmu.

Z typu výstupného parametra kompresie a vstupného parametra dekompresie, je možné si všimnúť, že sú indexy do slovníka pri zapisovaní a čítaní vnímané ako takzvané **triplety**. *Triplet* reprezentuje sekvenciu údajov, ktoré kodéri zapisujú na výstup a dekodéri čítajú zo vstupu v jednom kódovacom alebo dekódovacom kroku. Každý jeden z týchto údajov môže mať rôznu bitovú dĺžku a na ich počte nezáleží, aj napriek tomu, že sa nazývajú „triplety“, ktoré z názvu naznačujú, že ich je stále tri. Každý kodér si musí nadefinovať jednotlivé údaje tripletu pomocou triedy **TripletFieldId**, ktorá reprezentuje jeden údaj. Táto trieda si pamätá bitovú dĺžku daného údaju. V tomto prípade každá metóda obsahuje len jeden **TripletFieldId**, ktorého dĺžka závisí na konfigurácii algoritmu a tento údaj reprezentuje index do slovníka.

4.3.2 Trieda <meno metódy>Client

Táto trieda je použiteľná ako spustiteľná aplikácia, ktorá vykonáva kompresiu a dekompresiu pomocou danej implementovanej kompresnej metódy. Umožňuje spracovanie argumentov z príkazovej riadky, ktoré dovoľujú špecifikovať vstupný a výstupný súbor, nastaviť rôzne parametre kompresných metód a tiež špecifikovať, či sa má súbor komprimovať alebo dekomprimovať.

Spracovanie parametrov je dosiahnuté pomocou knižnice **Commons CLI**, ktorá tiež podporuje zobrazenie nápovedy, ktorá je vygenerovaná zo špecifikovaných parametrov. Tiež zaisťuje správnosť zadaných parametrov a v prípade chyby, je automaticky zobrazená nápoveda a program sa hneď ukončí. Keďže sú parametre pre všetky implementované kompresné metódy identické, sú rovnaké aj ich nápovedy, ktoré vyzerajú takto:

```
usage: java -jar sct-RELEASE.jar inFile outFile [options]
 inFile                input file name
 outFile               output file name
Options:
 -ar,--arithmetic      use ADAPTIVE_ARITHMETIC converter
                       for triplets
                       (this is default)
 -ba,--bitArray        use BIT_ARRAY converter for triplets
 -d,--decompress       decompress inFile (defaul=compress)
 -h,--help             print help
 -l,--length <L>       max bit length of dictionary index
                       (default=16)
 -log,--log-level <L>  sets logging level of the application
 -p,--partition <P>    partition size for dividing inFile
                       (default=1_000_000 B)
 -vba,--varBitArray    use BIT_ARRAY_VARIABLE converter
                       for triplets
```

Z nápovedy je možné vidieť, že je knižnica SCT kompilovaná ako celok do súboru **sct-RELEASE.jar**. Keďže knižnica obsahuje viacero kompresných metód a tak aj klientov, je nutné pri spúšťaní programu z príkazovej riadky, ako parameter zadať hlavnú triedu klienta. Príkaz, ktorý umožňuje spustiť ľubovoľnú kompresnú metódu vyzerá takto:

```
java sct–RELEASE.jar –cp <trieda klienta> [parametre]
```

4.3.3 Rozhranie <meno metódy>Provider

Toto rozhranie definuje tri metódy:

- **getCoder**,
- **getT2BConverter**,
- **getB2TConverter**.

Malo by byť implementované triedou, ktorá poskytuje nakonfigurované kodéry na základe parametrov implementovaných kompresných metód. Metóda **getCoder** by mala dodať kodér, ktorý je použitý pre vykonanie kompresného/dekompresného algoritmu danou kompresnou metódou a metódy **getT2BConverter** a **getB2TConverter** poskytujú kodér, ktorý je použitý pre spracovanie tripletov, buď tých, ktoré sú výstupom kompresného algoritmu alebo tých, ktoré sú vstupom algoritmu dekompresného. Dôvodom používania tohoto rozhrania je to, aby sa oddelila logika inicializácie a konfigurácie algoritmu od tej vykonávacej.

4.3.4 Trieda <meno metódy>ProviderImpl

Táto trieda implementuje rozhranie **<meno metódy>Provider>**. Konštruktér prijíma inštanciu triedy **<meno metódy>ProviderParameters**. Na základe týchto parametrov nakonfiguruje potrebné kodéry, ktoré potom poskytuje pomocou rozhrania. Ako už bolo spomenuté, toto rozhranie je volané hlavnou triedou kompresnej metódy, ktorá využije nakonfigurované kodéry pre vykonanie algoritmov.

4.3.5 Trieda <meno metódy>ProviderParameters

Táto trieda reprezentuje parametre kompresných metód. Parametre všetkých troch implementovaných metód sú rovnaké a majú takýto tvar:

Tabuľka 4.1: Zoznam parametrov implementovaných kompresných metód

dátový typ	**názov**
číslo (int)	length
enumerácia (enum)	converter
číslo (int)	partition

4.3.5.1 Length

Tento parameter definuje maximálnu bitovú dĺžku, ktorá je použitá pri zápise indexu do slovníka na výstup. Ovplyvňuje tiež maximálny počet fráz, ktoré môže slovník obsahovať. Veľkosť slovníka sa tak odvodí ako 2^l, kde l je dĺžka

indexu. Dôvodom, prečo sa používa mocnina dvoch, je to, aby sa využil maximálny rozsah hodnôt, ktoré je možné touto sekvenciou bitov takejto dĺžky zakódovať. Čím viac fráz slovník obsahuje, tým je väčší potenciál na dosiahnutie nižšieho kompresného pomeru.

4.3.5.2 Converter

Ako už bolo predtým spomenuté, indexy, ktoré kompresná metóda zapíše na výstup, sú považované za triplety a tento parameter slúži pre výber spôsobu, ako tieto triplety zakódovať predtým, ako sú zapísané do výstupného súboru. Existujú tri možné spôsoby:

- kódovanie adaptívnym aritmetickým kodérom,
- kódovanie ako celé čísla o fixnej bitovej dĺžke,
- kódovanie ako celé čísla o variabilnej bitovej dĺžke, ktorá závisí na súčasnej veľkosti slovníka.

Prvý spôsob je ten najpomalší, pretože aritmetické kódovanie je narozdiel od ostatných spôsobov kompresnou metódou sama o sebe. Na druhej strane by to mohlo ešte dodatočne znížiť kompresný pomer.

Druhý spôsob je ten najrýchlejší, pretože sú indexy do súboru zapisované priamo tak, ako sú uložené v pamäti, len s menšou fixnou bitovou dĺžkou. Na druhej strane to môže dosahovať vyššieho kompresného pomeru ako prvý spôsob.

Tretí spôsob je na tom rýchlostne podobne ako ten druhý. Na rozdiel od neho sa snaží kompresný pomer ešte znížiť tým, že je bitová dĺžka indexu odvodená od súčasnej veľkosti slovníka. Tomuto spôsobu sa stále podarí dosiahnuť nižšieho kompresného pomeru, pretože fixná bitová dĺžka je odvodená od maximálnej veľkosti slovníka a maximálna variabilná bitová dĺžka bude rovná práve tejto fixnej. Na druhej strane si tento spôsob musí vždy pri pridávaní nových fráz do slovníku strážiť a postupne inkrementovať súčasnú bitovú dĺžku indexu, aby sa vyhlo zbytočnému počítaniu logaritmu pri každom zápise indexu na výstup. Na rýchlosť to bude mať dopad veľmi minimálny.

4.3.5.3 Partition

Tento parameter definuje veľkosť segmentov, do ktorých sa vstupný súbor pred kompresiou rozdelí. Každá segment je komprimovaný individuálne, takže v prípade, že sa súbor rozdelí do n-segmentov, tak je kompresná metóda celkovo spustená n-krát. Dôvodom je to, aby sa zaistila možnosť spracovania súboru ľubovoľnej veľkosti a tiež to umožňuje kompresnú metódu jednoduchšie spustiť paralelne prostredníctvom viacerých vlákien.

4.3.6 Trieda <meno metódy>Test

Táto spustiteľná trieda slúži pre testovacie účely. Bola hlavne použitá počas doby implementácie pre progresívne overovanie funkčnosti danej kompresnej metódy.

4.3.7 Trieda <meno metódy>EncodeDictionary

Táto trieda predstavuje implementáciu slovníka, ktorý je použitý kompresným algoritmom danej kompresnej metódy. Ako už bolo naznačené v druhej kapitole, slovník je implementovaný ako stromová štruktúra *trie*. Uzly, ktoré majú spoločného rodiča sú usporiadané do binárneho vyhľadávacieho stromu, ktorý zaručuje, že hľadanie frázy o jeden symbol dlhšej prebehne väčšinou v logaritmickom čase. Je to z dôvodu, že BST nezaručuje vždy logaritmickú hĺbku, kvôli tomu, že nie je vyvážený. Maximálne môže obsahovať 256 uzlov, takže aj keby sa niektoré stromy degradovali na spojený list, nebude to mať až taký negatívny efekt na celkovú rýchlosť algoritmu.

Slovník obsahuje dva podstatné metódy: **search** a **add**. Vyhľadávanie je vo väčšine prípadov vykonané pomocou zisťovania existencie uzla so špecifikovaným symbolom v BST jeho prefixu, čiže rodiča. Pridanie je vykonané zaradením nového uzla do BST jeho špecifikovaného prefixu. Pred pridaním je tiež overené, či uzol už náhodou nie je súčasťou slovníka, v ktorom prípade nový uzol nie je vytvorený.

Každé uzly kódovacích slovníkov obsahujú nasledujúce položky:

Tabuľka 4.2: Položky uzlov kompresných slovníkov

LZMW		**LZAP**		**LZY**	
dátový typ	**názov**	**dátový typ**	**názov**	**dátový typ**	**názov**
bajt (byte)	symbol	bajt (byte)	symbol	bajt (byte)	symbol
číslo (int)	id	číslo (int)	id	číslo (int)	id
ukazovateľ	koreň	ukazovateľ	koreň	ukazovateľ	koreň
ukazovateľ	ľavý	ukazovateľ	ľavý	ukazovateľ	ľavý
ukazovateľ	pravý	ukazovateľ	pravý	ukazovateľ	pravý
		číslo (int)	vek	číslo (int)	vek
				ukazovateľ	prefix
				ukazovateľ	sufix

Kódovací uzol každej metódy si potrebuje pamätať, ktorý symbol reprezentuje pomocou položky **symbol** a tiež svoj index do slovníka pomocou položky **id**, ktorú kompresný algoritmus potrebuje pre zápis na výstup. Pri metóde LZMW táto položka môže obsahovať neplatnú hodnotu, ktorá značí, že uzol nie je v skutočnosti frázou, ale len časťou nejakej inej frázy.

Položka **koreň** odkazuje na uzol, ktorý reprezentuje koreň BST. Položky **ľavý** a **pravý** reprezentujú susedné uzly v BST, ktorého je uzol súčasťou.

To znamená, že sa vyhľadávanie frázy o jeden symbol dlhšej musí stále začať v uzle, na ktorý odkazuje položka **koreň** a potom sa postupne strom prechádza pomocou položiek **ľavý** a **pravý**.

Uzol slovníka používaného pri metóde LZY si potrebuje dodatočne pamätať tiež frázy, ktoré tvoria jeho prefix a sufix, kvôli spôsobu, ako táto metóda pridáva frázy do slovníka. Preto existujú položky **prefix** a **sufix**.

Uzol pri metóde LZAP a LZY si, narozdiel od LZMW, musí tiež pamätať položku **vek**. Toto celé číslo reprezentuje, kedy sa daný uzol alebo fráza do slovníka pridala. Ako bolo spomínané v tretej kapitole, tieto dva kompresné algoritmy potrebujú pre rýchlejšiu kompresiu dve verzie alebo časti slovníka. Jedna časť slúži pre vyhľadávanie a tá druhá pre pridávanie. To dovoľuje algoritmu pridávať nové frázy v tom istom cykle, ako sa hľadá nová najdlhšia fráza v slovníku. Pre rýchle pridanie týchto nových fráz do pridávacej časti slovníka sa použije táto položka, ktorá je porovnaná so súčasným vekom celého slovníka. Keď je táto položka menšia, znamená to, že bola fráza pridaná aspoň v minulom cykle pridávania a tak je fráza súčasťou vyhľadávacej časti slovníka. Inkrementovanie súčasného veku slovníka je zaistené kompresným algoritmom pomocou volania metódy slovníka **incrementAge**.

4.3.8 Trieda <meno metódy>DecodeDictionary

Táto trieda predstavuje implementáciu slovníka, ktorý je použitý dekompresným algoritmom danej kompresnej metódy. Štruktúra reprezentácie slovníka a spôsobu vyhľadávania a pridávania je podobná kompresnému slovníku.

Hlavný rozdiel spočíva v tom, že tieto slovníky potrebujú prístup k danému uzlu na základe jeho indexu do slovníka. Kvôli tomu sú všetky uzly pri všetkých algoritmoch uložené v poli, ktoré umožňuje rýchle indexovanie. Je potrebné spomenúť, že pri metóde LZMW sa do poľa ukladajú len uzly s platnou položkou **id**.

Ďalším rozdielom je to, že slovník tiež potrebuje byť schopný zrekonštruovať znenie fráz. Kvôli tomu je potrebná položka **prefix**, ktorá odkazuje na rodičovsky uzol. Metóda LZY túto položku pri uzloch jej kompresného slovníka už obsahovala, ale ostatné metódy nie a kvôli tomu je potrebné túto položku do ich uzlov pridať.

Každé uzly dekódovacích slovníkov obsahujú nasledujúce položky:

Tabuľka 4.3: Položky uzlov kompresných slovníkov

LZMW		**LZAP**		**LZY**	
dátový typ	**názov**	**dátový typ**	**názov**	**dátový typ**	**názov**
bajt (byte)	symbol	bajt (byte)	symbol	bajt (byte)	symbol
ukazovateľ	koreň	ukazovateľ	koreň	ukazovateľ	koreň
ukazovateľ	ľavý	ukazovateľ	ľavý	ukazovateľ	ľavý
ukazovateľ	pravý	ukazovateľ	pravý	ukazovateľ	pravý
ukazovateľ	prefix	ukazovateľ	prefix	ukazovateľ	prefix
boolean	fráza			ukazovateľ	sufix

Oproti kompresným slovníkov pri metódach LZAP a LZY tiež ubudla položka **vek**, pretože nie je potrebné mať dve verzie slovníka, presne ako to bolo vysvetlené v tretej kapitole.

Keďže dekompresné slovníky dostávajú indexy do slovníka priamo na vstupe, nie je potrebné si pre uzly udržiavať položku **id**, ale pri slovníku metódy LZMW je stále potrebné rozlišovať, ktoré uzly sú frázy, a ktoré nie a preto majú položku **fráza**, ktorá je binárnou hodnotou slúžiaca ako príznak.

4.3.9 Trieda <meno metódy>TripletCoder

Táto trieda reprezentuje kodér, ktorý vykonáva samotný algoritmus kompresie/dekompresie danej kompresnej metódy. Pre tieto účely obsahuje metódy **encode** a **decode**. Algoritmy sú implementované na základe pseudokódov, o ktorých sa hovorilo v tretej kapitole. Ako už bolo predtým spomenuté, kodér je nakonfigurovaný pomocou triedy, ktorá implementuje rozhranie **<meno metódy>Provider** a hlavnej triede kompresnej metódy poskytne tento kodér, ktorá potom použije jeho metódy **encode** a **decode** pre vykonanie príslušných algoritmov.

4.4 Stratégia pri naplnení slovníka

V druhej kapitole sa spomenulo niekoľko rôznych stratégii pri naplnení slovníka:

- vyčistenie slovníka,
- zmrazenie slovníka,
- kombinácia vyčistenia a zmrazenia,
- odstránenie najstarších použitých fráz (pri metóde LZMW).

Aj napriek týmto možnostiam, všetky tri implementované metódy používajú stratégiu vyčistenia slovníka. Platí to aj pre metódu LZMW, aj napriek tomu, že jej definícia vyžaduje, že sa po naplnení budú odstraňovať len najstaršie použité frázy. Na druhej strane je vyčistenie slovníka určitou formou odstránenia najstarších použitých fráz, len všetkých.

Použitie tohoto špecifického spôsobu je z dôvodu výsledkom testovania týchto spôsobov na rovnakých algoritmoch v práci [9]. Autor tejto práce vyskúšal všetky spomenuté stratégie a dopad na kompresný pomer bol len veľmi minimálny. Stratégia vyčistenia je z nich najjednoduchšia a najuniverzálnejšia a kvôli tomu bola zvolená práve ako predvolená.

4.5 Vstup a výstup

O vstup a výstup sa stará trieda **FileIO**. Táto trieda vyžaduje vo svojom konštruktéri parameter, ktorý špecifikuje veľkosť segmentov, do ktorých sa vstupný súbor má rozdeliť. Tento parameter je dodaný parametrom **partition**. Výhodou tejto triedy je to, že je možné ju reťaziť ako proces do dátového toku triedy **ChainBuilder**.

4.6 Dátový tok

Ako už bolo naznačené, kompresia a dekompresia implementovaná v knižnici SCT zahŕňa určitú sekvenciu krokov alebo procesov, ktorá je reprezentovaná pomocou triedy **ChainBuilder**.

Napríklad dátový tok kompresie metódy LZAP je reprezentovaný takto:

```
ChainBuilder.create(io::openParse)
        .chain(lzap::compress)
        .chain(tripletToByteConverter)
        .end(bytes -> io.saveObject(bytes, cmpFile.toPath()))
        .accept(origFile.toPath());
```

Tento dátový tok vykonáva nasledujúce kroky:

- načítanie originálneho súboru,
- vykonanie kompresného algoritmu,
- kódovanie tripletov,
- zapísanie do výsledného súboru.

Načítanie súboru je vykonané pomocou inštancie triedy **FileIO**, ktorá súbor rozdelí na segmenty a potom sa postupne posielajú kompresnej metóde, ktorá vykoná kompresný algoritmus a na výstup pošle indexy do slovníka v podobe tripletov, ktoré sú pred zápisom do výstupného súboru dodatočne

zakódované triplet kodérom, ktorého typ je definovaný parametrom **converter** .

Je možné si všimnúť, že vykonanie dátového toku je zahájené zavolaním metódy **accept**, ktorá príjme inštanciu vstupného súboru, ktorá sa predá prvému procesu toku a kompresia sa tak zaháji.

Dátový tok dekompresie vyzerá veľmi podobne:

```
ChainBuilder.create(io::openObject)
        .chain(byteToTripletConverter)
        .chain(lzap::decompress)
        .end(byteBuffer -> io.saveParsed(byteBuffer,
                                         decFile.toPath()))
        .accept(cmpFile.toPath());
```

Tento dátový tok vykonáva nasledujúce kroky:

- načítanie komprimovaného súboru,
- dekódovanie tripletov,
- vykonanie dekompresného algoritmu,
- zapísanie do výsledného súboru.

V porovnaní s kompresiou, je nutné aby sa dáta spracovali v opačnom poradí, takže dekódovanie tripletov sa musí vykonať pred samotným dekompresným algoritmom.

4.7 Logovanie

Implementované metódy využívajú knižnicu **Apache Log4j2**, ktorá slúži ako systém logov. Jej výhodou je to, že je možné vytvárať výpisy rôznych stupňov a je možné tak, kontrolovať čo sa má vypisovať a čo nie v rôznych situáciach bez toho, aby táto logika musela byť súčasťou samotných implementovaných metód.

Príklad použitia je napríklad to, že sa pri overení funkčnosti algoritmu nastaví systém do stavu **TRACE**, čo dovoľuje výpis logov, ktoré detailne zachytávajú operácie, ktoré algoritmus vykonáva.

Na druhej strane, pri testovaní algoritmu na rýchlosť sa systém nastaví do stavu **OFF** a tak sa nebudú vypisovať výpisy žiadne, čo umožní testovať algoritmus bez odozvy, ktoré tieto výpisy spôsobujú.

Kapitola 5

Testovanie

Táto kapitola sa zaoberá testovaním výkonu novo implementovaných metód LZAP, LZMW a LZY, ktoré sú tiež porovnané s metódou LZW, ktorej implementácia v knižnici SCT už existovala. Výkonom metódy sa myslí čas kompresie/dekompresie a hodnota kompresného pomeru. Tiež je skúmané, aký vplyv majú na výkon rôzne nastaviteľné parametre kompresných metód.

5.1 Metodika

Pre spustenie jednotlivých metód boli použité triedy typu **<meno metódy> Client**, ktoré umožňujú vykonať kompresiu/dekompresiu na špecifikovanom súbore spolu s nastavením rôznych parametrov, ktoré tento proces modifikujú, ako to bolo popísané v predošlej kapitole. Ako testovacie dáta boli použité rozmanité súbory Pražského korpusu [1], ktorý zabezpečuje, že sú metódy otestované na rôznych typoch dát. Keďže čas kompresie a dekompresie môže byť ľahko ovplyvnený operačným systémom, v ktorom bežia aj ostatné procesy, je každý test na jednotlivých súboroch vykonaný niekoľko krát, z ktorých je vybraný ten najmenší čas ako ten výsledný.

Pre automatizáciu testovania boli vytvorené dva bash skripty. Prvý skript testuje metódy LZAP, LZMW, LZY a LZW na všetkých súboroch Pražského korpusu. Metódy sú v tomto prípade spúšťané s východnými hodnotami parametrov. Druhý skript testuje vplyv nastaviteľných parametrov na výkon špecifikovanej kompresnej metódy na špecifikovanom súbore.

5.2 Testovacia platforma

Zostavenie knižnice SCT a testovanie výkonu kompresných metód bolo vykonané na systéme s nasledujúcou špecifikáciou:

- Operačný systém – Ubuntu 16.04 LTS,
- Procesor – AMD FX-7500 Radeon R7, 10 Compute Cores 4C+6G 2.10 GHz,
- Architektúra – 64 bit,
- RAM – 8 GB.

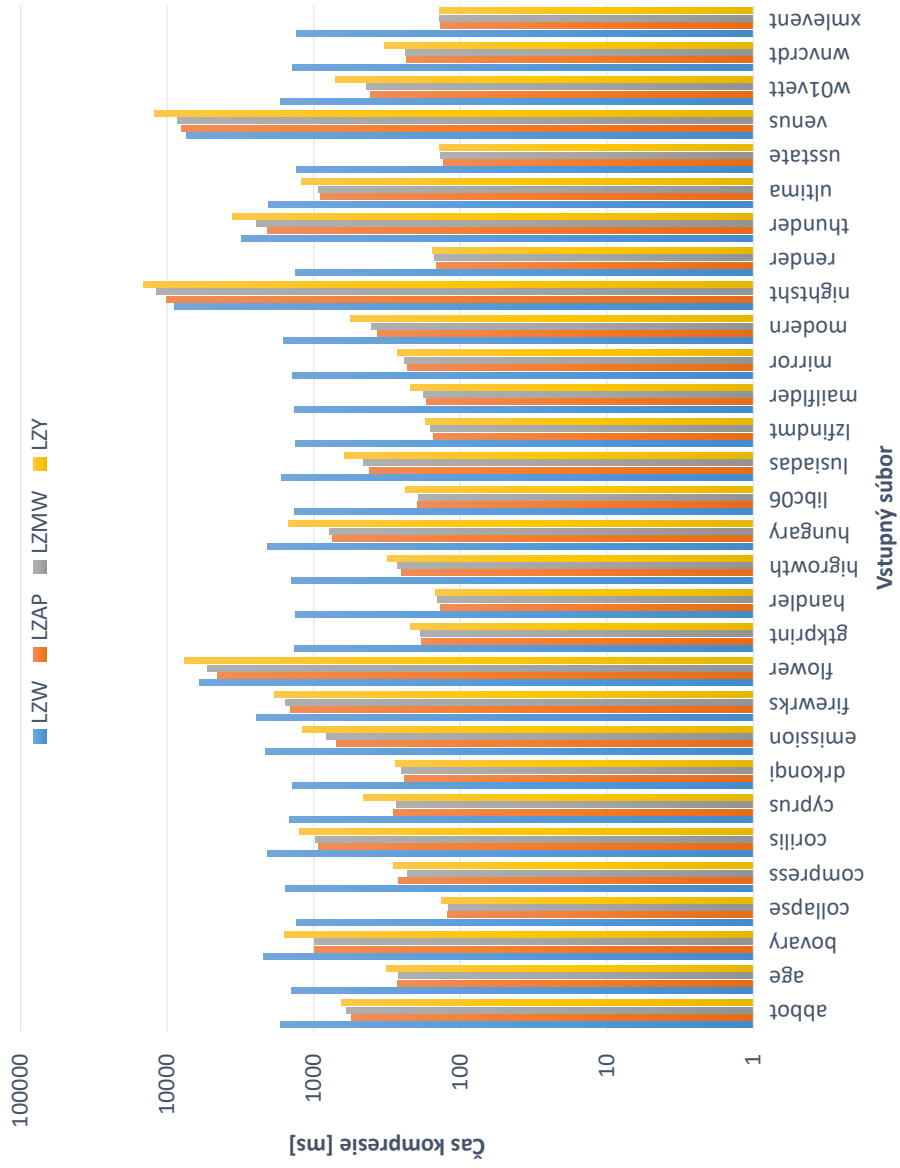
5.3 Výsledky

5.3.1 Čas kompresie

Z troch implementovaných metód, bola v prípade času kompresie najrýchlejšia metóda LZAP. Pri 25 z 30 testovaných súboroch mala najmenší čas kompresie. V porovnaní s LZMW bola v priemere o približne 7% rýchlejšia (s najväčšou odchýlkou 16% pri súbore **venus**) a v porovnaní s LZY bola rýchlejšia v priemere o približne 25% (s najväčšou odchýlkou 50% pri súbore **hungary**). Pri súboroch **age**, **collapse**, **compress**, **cyprus** a **libc06**, bola najrýchlejšia metóda LZMW, ale oproti LZAP bola na týchto súboroch rýchlejšia v priemere len o 2,5%. Najpomalšia bola teda metóda LZY, čo nie je ani také prekvapivé, keďže je zo všetkých najkomplexnejšia (ako to bolo vysvetlené v tretej kapitole). LZMW bola oproti nej v priemere o 21% rýchlejšia (s najväčšou odchýlkou 48% pri súbore **hungary**).

V porovnaní s metódou LZW boli metódy LZAP a LZMW prekvapivo v priemere o 72% rýchlejšie pri všetkých súboroch okrem **nighsht** a **venus**, kde LZW bolo rýchlejšie približne oproti LZAP o 13% a 9% a oproti LZMW o 31% a 16%. Podobne na tom bola aj metóda LZY, ktorá bola rýchlejšia v priemere o 68% pri všetkých súboroch okrem **flower**, **nighsht**, **thunder**, **venus**, kde bola pomalšia oproti LZW o približne 26%, 62%, 14% a 64%. Aj napriek tomu, že teoreticky by mala byť LZW najrýchlejšia, sú tieto metódy rýchlejšie ako LZW. Je to pravdepodobne kvôli tomu, že implementácia LZW používa o trochu inú reprezentáciu kompresného slovníka, ktorá nie je veľmi rýchla pri menších súboroch, ale na druhej strane čím sa veľkosť súboru zväčšovala, tým sa jej rýchlosť priblížila k rýchlosti ostatných metód. Až pri súboroch **flower**, **nighsht**, **thunder**, **venus** sa jej podarilo poraziť aj ostatné metódy, pretože tieto súbory patria medzi tie najväčšie z celého Pražského korpusu.

Vo výsledku je pri súčasnom stave týchto algoritmov vhodné používať metódu LZAP pri menších súboroch a metódu LZW pri väčších súboroch, keď nám ide o čo najrýchlejšiu kompresiu.



Obr. 5.1: Porovnanie kompresného času metód LZW, LZAP, LZMW, LZY

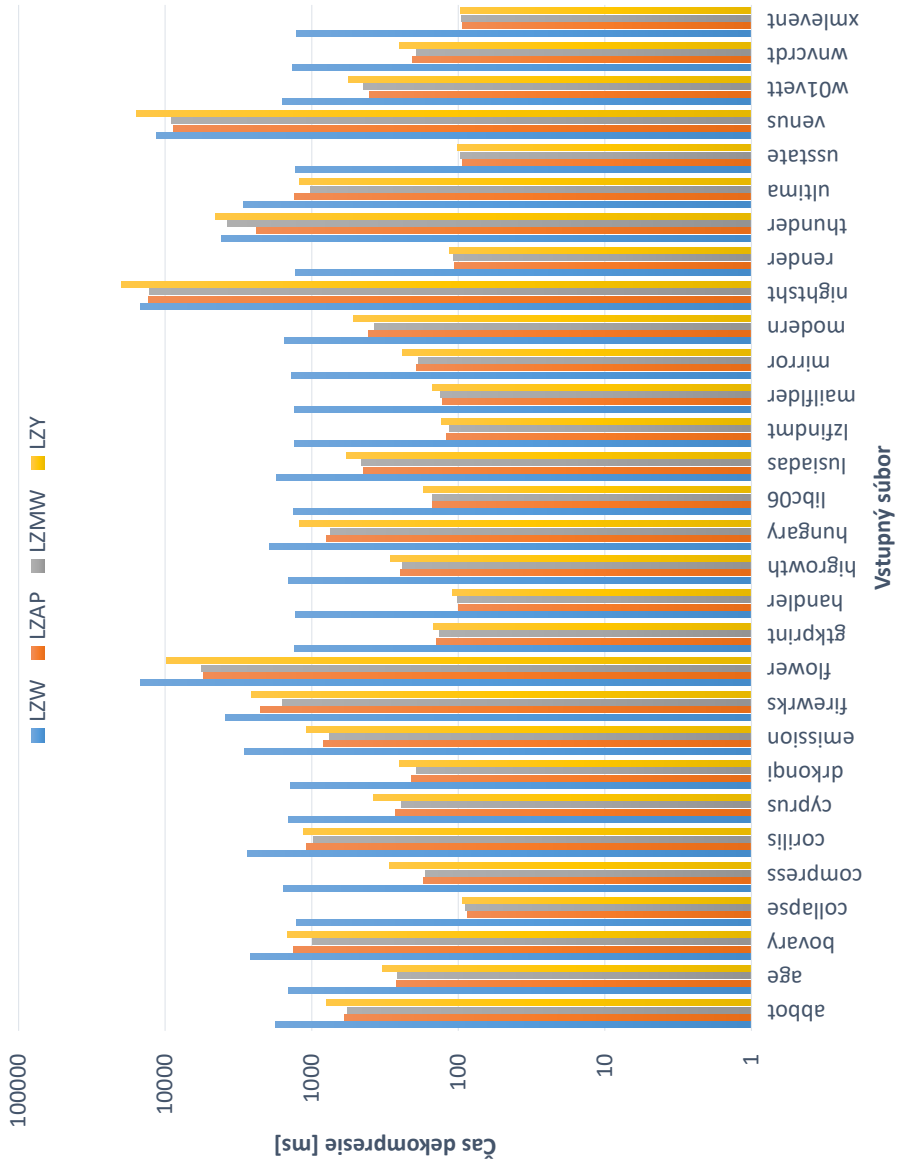
5.3.2 Čas dekompresie

Z troch implementovaných metód bola v prípade času dekompresie najrýchlejšia metóda LZMW. Pri 19 z 30 testovaných súboroch bola rýchlejšia ako LZAP v priemere o 10% (s najväčšou odchýlkou 40% pri súbore **firewrks**). Pri ostatných súboroch bola pomalšia oproti LZAP maximálne len o približne 10%, až na jednu výnimku pri súbore **thunder**, kde bola pomalšia až o 36%. V porovnaní s LZY, bolo LZMW pri všetkých súboroch rýchlejšie v priemere o 21% (s najväčšou odchýlkou 42% pri súbore **venus**). LZAP bolo rýchlejšie než LZY skoro pri všetkých súboroch v priemere o 20% (s najväčšou odchýlkou 47% pri súbore **thunder**), až na jeden súbor **ultima**, kde sa LZY podarilo byť rýchlejšie o skoro 8%.

V porovnaní s metódou LZW boli metódy LZAP a LZMW rýchlejšie na všetkých súboroch v priemere o približne 72% (s najmenšími odchýlkami okolo 10% a 20% pri súboroch **nightsht** a **venus**, resp.). Metóda LZY bola rýchlejšia ako LZW skoro pri všetkých súboroch o tiež približne 72% (s najmenšou odchýlkou 37% pri súbore **hungary**). Pri súboroch **nightsht**, **thunder** a **venus** bola metóda LZW rýchlejšia oproti LZY o približne 35%, 10% a 38%, respektívne.

Podobne ako pri čase kompresie, je LZW pomalšie ako jej varianty aj napriek teórii, pravdepodobne opäť kvôli odlišnej reprezentácii dekompresného slovníka, ktorá nie je vhodná pre menšie súbory. Ale na druhej strane je vidieť, že implementácia metódy LZW svoje varianty postupne dobieha, ako sa zvyšuje veľkosť súboru a pri najväčšom súbore **nighsht** sa jej podarilo poraziť metódu LZY o celkom podstatnú hodnotu.

Vo výsledku je v prípadoch, že nás zaujíma čo najmenší čas dekompresie pri súčasnom stave implementovaných algoritmov, vhodné pre menšie súbory použiť buď metódu LZMW alebo LZAP a pri väčších súboroch pravdepodobne metódu LZW. Čas dekompresie tiež podstatne ovplyvňuje kompresný pomer, pretože čím menej vstupných indexov sa musí spracovať, tým rýchlejšia dekompresia. Presne kvôli tomu sa podarilo metóde LZMW poraziť metódu LZAP pri väčšine súboroch, aj napriek tomu, že zložitosť ich algoritmov je skoro identická.



Obr. 5.2: Porovnanie dekompresného času metód LZW, LZAP, LZMW, LZY

5.3.3 Kompresný pomer

Z troch implementovaných metód dosahovala najnižší kompresný pomer väčšinou metóda LZMW. Pri 17 z 30 testovaných súboroch mala v priemere o približne 11% nižší kompresný pomer ako LZAP (s najväčšou odchýlkou okolo 77% pri súbore **hungary**). Na druhej strane pri ostatných 13 súboroch jej kompresný pomer nebol vyšší než 9,5%. V porovnaní s LZY, sa podarilo LZMW dosiahnuť pri 26 z 30 súboroch nižší kompresný pomer, v priemere to bolo o 12% nižšie (s najväčšou odchýlkou okolo 74% pri súbore **hungary**). Pri ostatných 4 súboroch, rozdiel nepresiahol 4%.

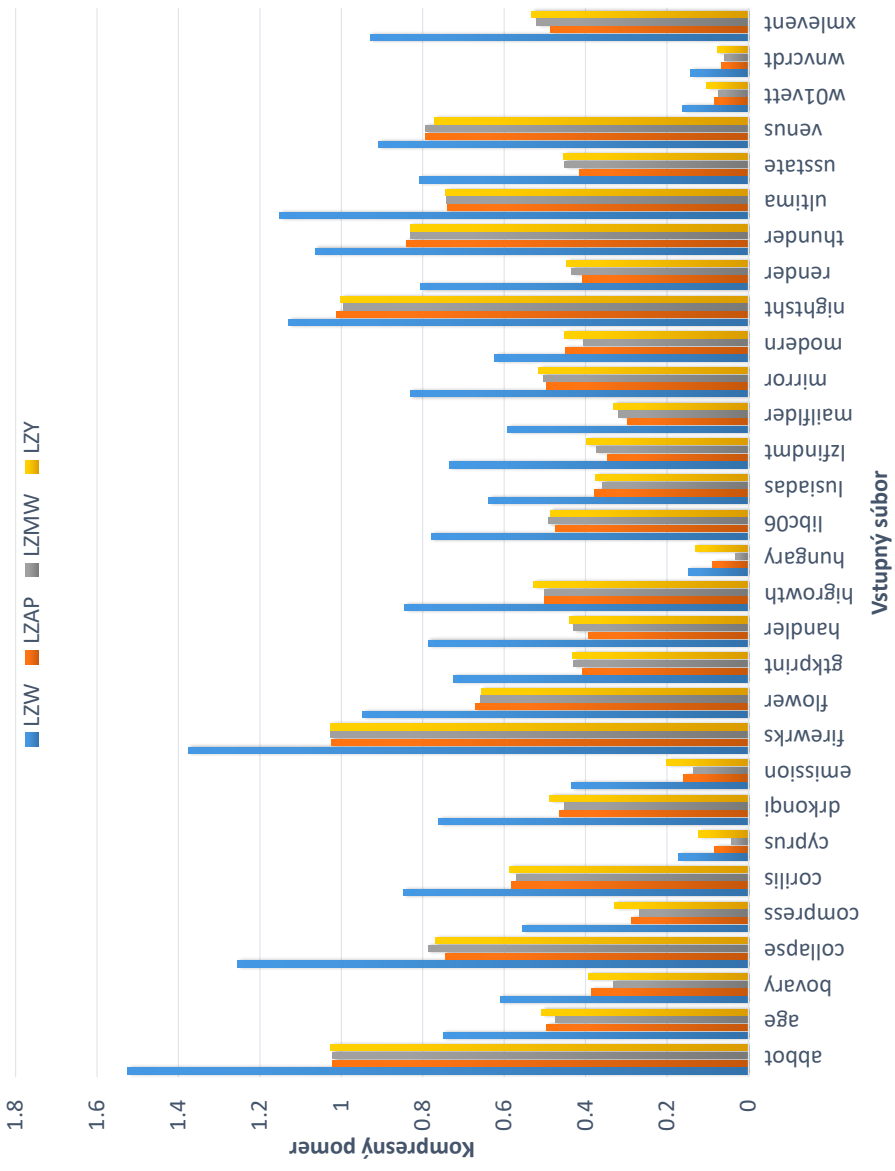
V porovnaní s metódou LZW dosahovali jej varianty v priemere o 40% nižšieho kompresného pomeru na všetkých testovaných súboroch, čo dokazuje, že jej varianty sú skutočne podstatne nadradené v prípade dosahovania nižšieho kompresného pomeru. Najväčší rozdiel sa podarilo spraviť metóde LZMW na súbore **emission**, pri ktorom dosiahla o približne 68% nižší kompresný pomer ako metóda LZW.

Najnižších kompresných pomerov sa podarilo dosiahnuť metóde LZMW na súboroch **hungary**, **cyprus**, **wnvcrdt**, **w01vett**, ktorých hodnoty boli približne 3,3%, 4,2%, 6%, 7,4% veľkosti originálneho súboru. Tieto súbory obsahujú dáta typu **markup language** a **database**, ktoré sú výstižné tým, že obsahujú opakujúce sa reťazce a to znamená, že sú schopné byť efektívne komprimované slovníkovými metódami.

Na druhej strane, najvyšších kompresných pomerov dosahovala metóda LZW na súboroch **abbot**, **firewrks**, **collapse**, **ultima**, **nightsht**, **thunder**, pri ktorých dosiahla kompresného pomeru nad hodnotu 1, čo znamená nad hodnotu 100% originálnej veľkosti a tak dané súbory zväčšila. Tieto súbory sú typu **audio**, **graphics** a **binary**, ktoré nie sú známe tým, že obsahujú opakujúce sa reťazce, a tak nie sú vhodné pre komprimovanie slovníkovými metódami.

Pre zaujímavosť, sa všetkým variantám LZW dokonca podarilo dosiahnuť kompresného pomeru pod hodnotu 1 pri súboroch **collapse**, **ultima** a **thunder**, pri ktorých sa to metóde LZW nepodarilo. Ale podobne ako LZW, dosiahli všetky varianty kompresného pomeru nad hodnotu 1 pri súboroch **abbot**, **firewrks** a **nightsht**

Vo výsledku, je vhodné pre dosiahnutie čo najmenšieho kompresného pomeru použiť vo väčšine prípadov metódu LZMW alebo aj prípadne LZAP a LZY, ale vyhnúť sa súborom, ktoré obsahujú zvukové, grafické a binárne dáta, pretože pri nich to pravdepodobne dopadne kompresným pomerom väčším ako 1.



Obr. 5.3: Porovnanie kompresného pomeru metód LZW, LZAP, LZMW, LZY

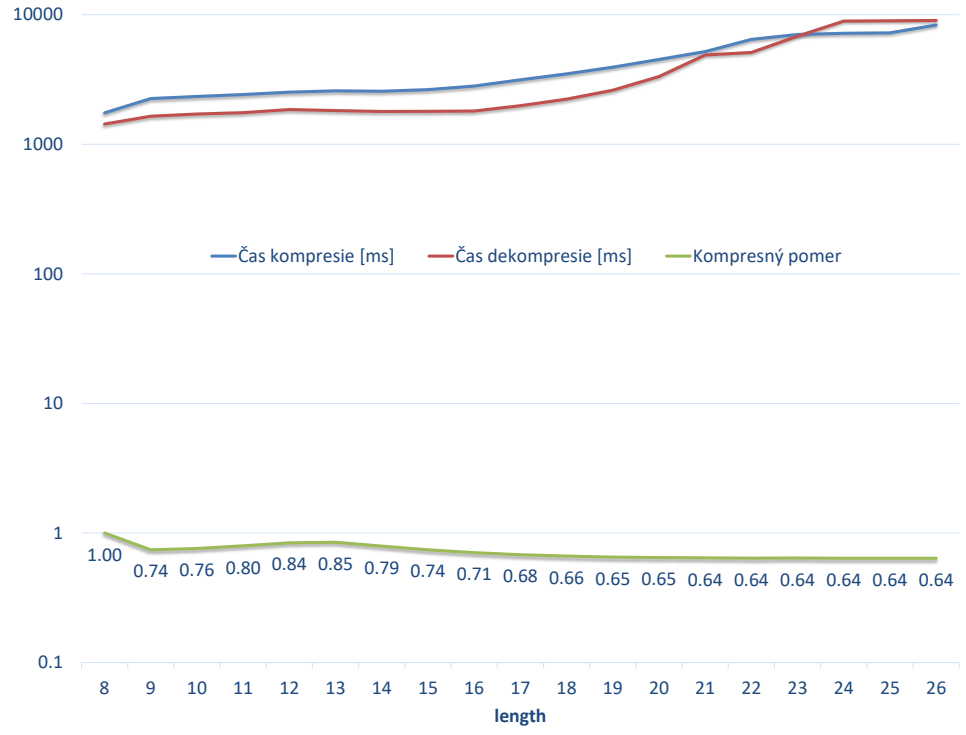
5.3.4 Nastaviteľné parametre

5.3.4.1 Length

Ako bolo spomenuté predtým, tento parameter kontroluje maximálnu bitovú dĺžku výstupných indexov do slovníka a tak aj maximálny počet fráz, ktorý môže slovník obsahovať. Zvyšovanie hodnoty tohoto parametra tak zvyšuje kapacitu slovníka, čo by teoreticky malo prispieť k dosiahnutiu menšieho kompresného pomeru, pretože čím viac fráz slovník má, tým je väčšia pravdepodobnosť, že sa podarí zakódovať frázy väčšej dĺžky a tiež to znižuje koľko krát je slovník vyčistený, čo znamená, že sa daný obsah slovníka môže používať pre kódovanie fráz o niečo dlhšiu dobu.

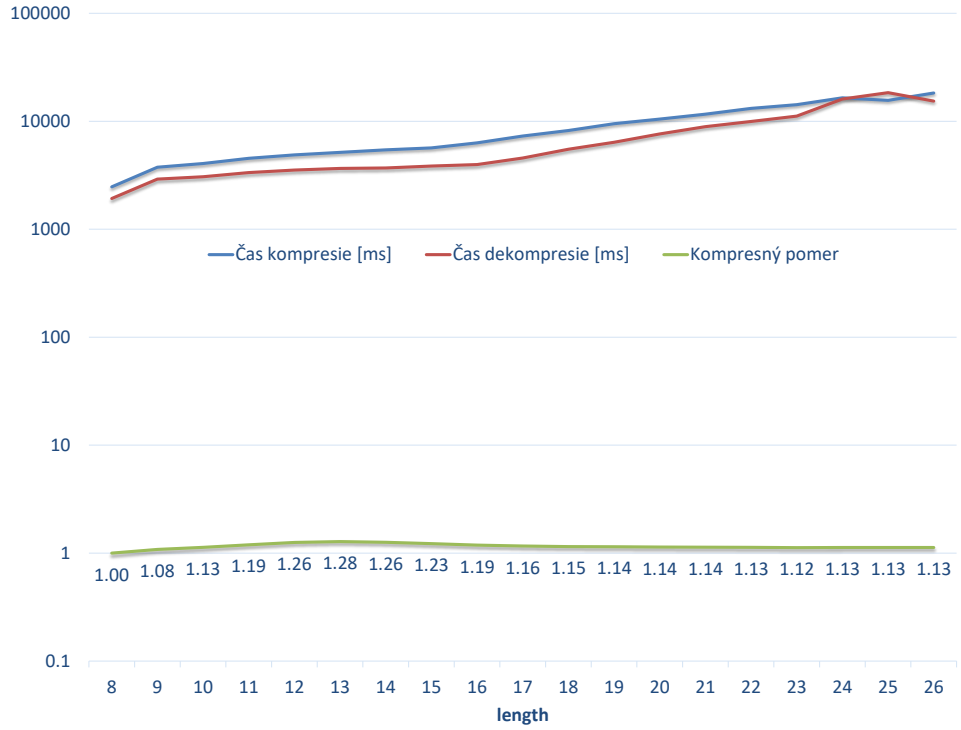
Obrázok 5.4 znázorňuje aký dopad mal tento parameter na výkon metódy LZAP na súbore **flower**. Je možné si všimnúť, že zvyšovanie parametra dosiahlo celkovo zníženie kompresného pomeru o hodnotu 0,36 oproti minimálnej hodnoty parametra, pri ktorej slovník obsahuje stále len prvých 256 fráz, s ktorými sa inicializuje a tak žiadne nové frázy do slovníka pridané nie sú. Od hodnoty parametra 21 a ďalej sa kompresný pomer prestal znižovať, pravdepodobne preto, pretože dĺžka indexu začala byť tak dlhá, že vynulovala výhody, ktoré prišli s vyššou kapacitou slovníka.

Obr. 5.4: Vplyv parametra **length** na čas kompresie, dekompresie a kompresný pomer metódy **LZAP** na súbore **flower**



Na druhej strane zvyšovanie tohoto parametra môže mať aj nepriaznivé účinky voči kompresnému pomeru, hlavne keď sa komprimuje súbor, pre ktorý nie je metóda vhodná, čiže súbor obsahuje dáta, ktoré sa veľmi neopakujú v sekvenciach. Príkladom takéhoto súboru je **nightsht**, ktorého výsledky je možné vidieť na obrázku 5.5. Je možné si všimnúť, že zvyšovanie parametra kompresný pomer naopak zhoršilo oproti minimálne hodnote, pretože aj napriek zvyšovaniu kapacity slovníka sa veľmi úspešnosť nachádzania fráz v slovníku nezlepšila a vyššie bitové dĺžky indexov len spôsobili progresívne zvýšenie kompresného pomeru, aj napriek tomu, že od hodnoty parametra 14 sa kompresný pomer začal trošku znižovať oproti predošlým hodnotám, ale stále to nebolo dostatočné sa dostať pod hodnotu kompresného pomeru 1.

Obr. 5.5: Vplyv parametra **length** na čas kompresie, dekompresie a kompresný pomer metódy **LZAP** na súbore **nightsht**



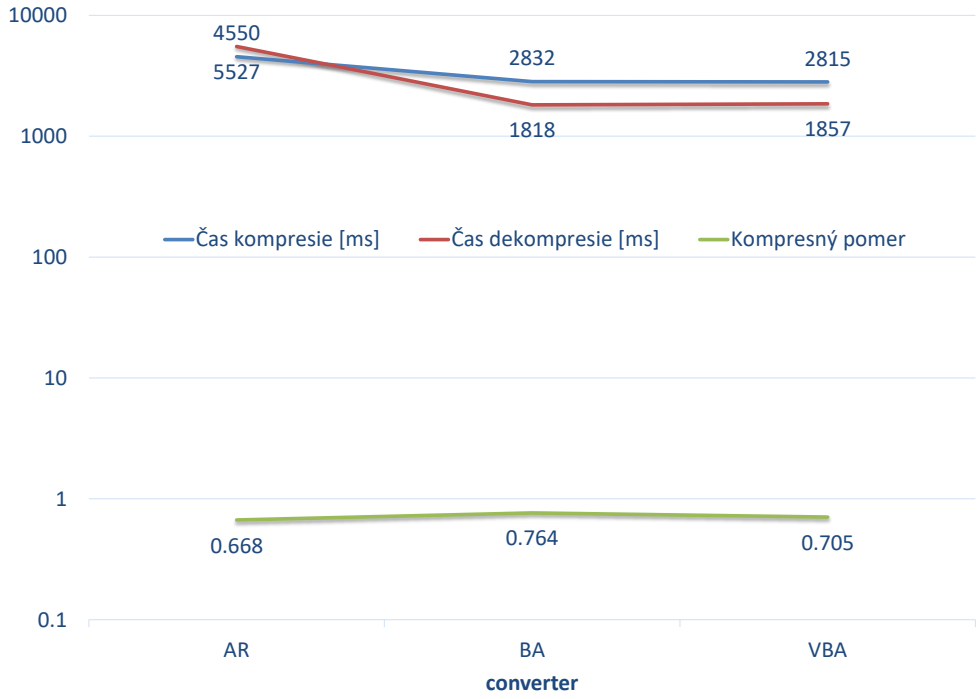
Na obrázkoch 5.4 a 5.5 si je možné tiež všimnúť, že zvyšovanie parametra tiež spôsobilo zvýšenie času kompresie/dekompresie. Dôvodom je to, že slovníky používajú BST pre uchovávanie potomkov uzla, čo znamená, že čím väčšia kapacita slovníka, tým viac sa tieto stromy naplňujú, čo ich viac prehlbuje, a tak vyhľadávanie a pridávanie uzlov zaberá viac času. Preto je vhodné urobiť určitý kompromis medzi kompresným pomerom a časom behu algoritmu a nastaviť tak parameter na vhodnú hodnotu. Na základe týchto obrázkov to vyzerá, že tento parameter je vhodné nastaviť najlepšie na hodnotu 16 až 17,

pretože pri vyšších hodnotách bola zmena kompresného pomeru len zanedbateľná.

5.3.4.2 Converter

Tento parameter nastavuje ako sa kódujú triplety, čiže výstupné indexy do slovníka. Sú podporované tri rôzne spôsoby, čiže kódovanie aritmetickým kodérom (AR), kódovanie ako celé čísla o fixnej dĺžke (VA) alebo kódovanie ako celé čísla o variabilnej dĺžke odvodenej od súčasnej veľkosti slovníka (VBA). Vplyv týchto rôznych spôsobov kódovania bol otestovaný na metóde LZAP pre súbor **flower**, ktorého výsledky sú na obrázku 5.6.

Obr. 5.6: Vplyv parametra **converter** na čas kompresie, dekompresie a kompresný pomer metódy **LZAP** na súbore **flower**



Ako je možné vidieť, najnižšieho kompresného pomeru sa podarilo dosiahnuť pri kódovaní aritmetickým kodérom, čo nie je vôbec prekvapenie, pretože tento spôsob je na rozdiel od ostatných kompresnou metódou sama o sebe. Na druhej strane to ale podstatne spomalilo čas behu algoritmu, hlavne toho dekompresného.

Pri ostatných dvoch spôsoboch, kde sa index zapíše na výstup priamo ako celé číslo, je čas behu algoritmu skoro identický. Ako je možné vidieť, keď sa dĺžka indexu odvodí od súčasnej veľkosti slovníka, je dosiahnutý samozrejme o trochu menší kompresný pomer. Keďže tieto dva spôsoby sú časovo identické,

je preferované použiť spôsob VBA, keďže dosiahne stále menší kompresný pomer ako VA.

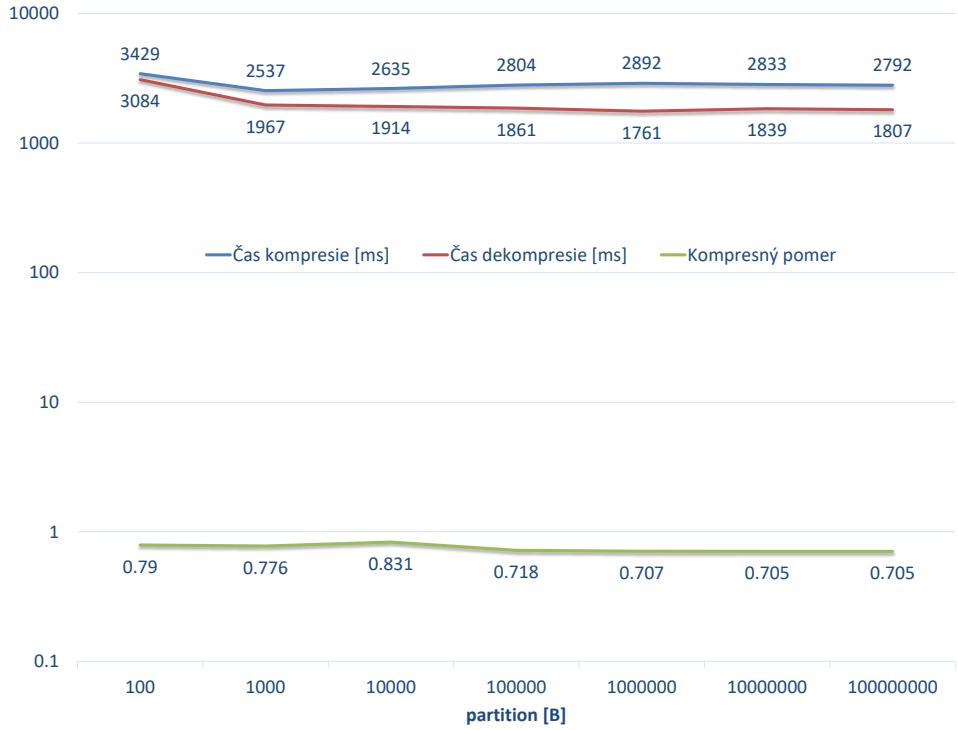
Vo výsledku je asi lepšie používať spôsob VBA, keďže aritmetický kodér celkom podstatne zvýši čas behu algoritmu a výhody, ktoré prináša kompresnému pomeru nie sú oproti spôsobu VBA až tak veľmi výrazné.

5.3.4.3 Partition

Tento parameter ovplyvňuje veľkosť partícií v bajtoch, do ktorých sa rozdelí vstupný súbor a tieto partície sú komprimované individuálne. Efekt, ktorý tento parameter má, je podobný parametru **length**, pretože znižovanie tohoto parametra môže ovplyvniť veľkosť, do ktorého slovník rastie v prípadoch keď je jeho kapacita väčšia ako veľkosť tohoto parametra, keďže sa pri ďalšej partícii slovník znova inicializuje.

Napríklad pri $length = 16$, je kapacita 2^{16} fráz, čiže 65536, takže keď nastavíme tento parameter na hodnotu 40000 B, nevyužije sa celková kapacita slovníka, čo môže mať nepriaznivý vplyv na kompresný pomer, pretože sa nevyužíva celkový potenciál danej bitovej dĺžky indexu. To hlavne platí keď sa indexy kódujú s fixnou bitou dĺžkou. Obrázok 5.7 znázorňuje podobnú situáciu.

Obr. 5.7: Vplyv parametra **partition** na čas kompresie, dekompresie a kompresný pomer metódy **LZAP** s **length** = 16 na súbore **flower**



Keďže bol algoritmus na obrázku 5.7 testovaný s *length* = 16, po dosiahnutí veľkosti partície s hodnotou 100000 B, dosiahol kompresný pomer o približne 0,1 nižšiu hodnotu ako keď bol spustený s veľkosťou partície len 100 B. Dokonca kompresný pomer klesol aj pri vyšších hodnotách tohoto parametra, pravdepodobne kvôli tomu, že pri hodnote 100000 B sa slovník inicializoval jeden krát a slovník po tom sa nestihol úplne naplniť dokým partícia skončila. Preto je lepšie partíciu nastaviť na nejaký násobok kapacity slovníka, aby sa kapacita vždy využila naplno.

Tiež je si možné všimnúť, že čas behu algoritmu bol pri minimálnej hodnote kapacity o trochu vyšší ako pri nasledujúcich hodnotách. Dôvodom je pravdepodobne réžia s tvorby a inicializácie objektov na začiatku každej partície, na ktorej až tak veľmi nezáleží pri vyšších hodnotách parametru, pretože sa vykonáva menej často.

Záver

Hlavným cieľom tejto práce bolo analyzovať a implementovať kompresné metódy LZAP, LZMW a LZY. Ich implementácia bola vykoná ako súčasť knižnice **Small Compression Toolkit (SCT)**.

Súčasne boli vykonané testy vyššie uvedených metód na súboroch Pražského korpusu z hľadiska kompresného pomeru a času kompresie/dekompresie. Metódy boli tiež porovnané s metódou LZW a ďalej sa preskúmali vplyvy rôznych nastaviteľných parametrov kompresných metód na testované parametre.

V prípade času kompresie bola v priemere najrýchlejšia metóda LZAP, tesne za ňou bola metóda LZMW a ako najpomalšia skončila metóda LZY, kvôli jej komplexnosti.

V prípade času dekompresie skončila v priemere najrýchlejšou metóda LZMW, aj napriek tomu, že metóda LZAP bolo tesne za ňou a dokonca na niektorých súboroch skončila o trochu lepšie. Najpomalšou metódou bola znovu LZY, zase kvôli jej komplexnej štruktúre.

Všetky metódy dosiahli porovnateľného kompresného pomeru, aj napriek tomu, že metóde LZMW sa väčšinou podarilo dosiahnuť kompresného pomeru o trochu menšieho ako ostatným, kvôli jej schopnosti pridávať do slovníka dlhé reťazce symbolov. V porovnaní s metódou LZW, sa všetkým metódam podarilo dosiahnuť menšieho kompresného pomeru, čo bolo ich hlavnou črtou.

Testy tiež ukázali, že testované algoritmy nie sú vhodné na komprimovanie dát, ktoré neobsahujú dostatočne dlhé často sa opakujúce sekvencie symbolom, ktoré sú potrebné na to, aby dané algoritmy dosahovali nezanedbateľného kompresného pomeru. V prípade Pražského korpusu takéto dáta obsahujú súbory, ako napríklad **abbot**, **nightsht** a **firewrks**, pri ktorých všetky testované kompresné metódy dosiahli kompresného pomeru skoro rovného alebo väčšieho ako 1, čo znamená, že veľkosť originálnych súborov zostala prakticky identická alebo ešte väčšia.

Implementované algoritmy sú využiteľné hlavne pre študijne účely, ale s trochou optimalizácií je ich možno využiť aj v praxi prostredníctvom knižnice SCT.

Do budúcna by mohlo byť zaujímavé knižnicu rozšíriť o nejaké ďalšie kompresné metódy a porovnať ich s metódami implementovanými v tejto práci.

Literatúra

[1] Řezníček, J.: *Corpus for comparing compression methods and an extension of a ExCom library*. Diplomová práce, Czech Technical University in Prague, Faculty of Electrical Engineering, Květen 2017. Dostupné z: `http://www.stringology.org/projects/PragueCorpus/data/papers/Reznicek-MSc_thesis-2010.pdf`

[2] Small Compression Toolkit [online]. [cit. 2018-05-18]. Dostupné z: `https://gitlab.fit.cvut.cz/polacrad/sct`

[3] Welch, T. A.: A Technique for High-Performance Data Compression. *Computer*, ročník 17, 1984: s. 8–19.

[4] Ziv, J.; Lempel, A.: Compression of Individual Sequences via Variable-Rate Coding. 1978.

[5] Nieminen, J.: An efficient LZW implementation [online]. 2007, [cit. 2018-06-21]. Dostupné z: `http://warp.povusers.org/EfficientLZW`

[6] Miller, V. S.; Wegman, M. N.: Variations on a theme by Ziv and Lempel. 1988.

[7] Storer, J. A.: Data Compression: Methods and Theory. *Computer Science Press*, 1988.

[8] Salomon, D.: *Data compression - The Complete Reference, 4th Edition*. Springer, 2007, ISBN 978-1-84628-602-5.

[9] Zemančík, T.: *Slovníkové metody LZ*. Diplomová práce, ČVUT v Praze, Fakulta informacních technologií, 2012.

[10] Bican, J.: *Implementation of the ACB compression method improvements in the Java language*. Diplomová práce, Czech Technical University in Prague, Faculty of Information Technology, 2017.

Dodatok A

Zoznam použitých skratiek

BST Binary Search Tree

GIF Graphics Interchange Format

LZ Lempel-Ziv, označenie rodiny slovníkových kompresných metód

LZ78 Lempel-Ziv, kompresná metóda z roku 1978

LZAP Modifikácia LZMW, AP znamená „All prefixes“ (všetky prefixy)

LZMW Lempel-Ziv-Miller-Wegman, Varianta LZW

LZW Lempel-Ziv-Welch, kompresná metóda založená na LZ78

LZY Lempel-Ziv-Yabba, Varianta LZW

PDF Portable Document Format

SCT Small Compression Toolkit

Dodatok B

Skripty použité pre testovanie

Pre automatizáciu testovania boli vytvorené dva *bash* skripty, ktoré vykonávajú proces merania výkonu kompresných a dekompresných operácii implementovaných metód.

Prvý skript **test.sh** bol použitý pre vzájomné porovnanie výkonu implementovaných kompresných metód. Druhý skript **test_parameters.sh** bol použitý pre meranie vplyvu rôznych nastaviteľných parametrov implementovaných kompresných metód na ich výkon.

Obidve skripty merajú výkon kompresných metód na súboroch korpusu a namerané výsledky zapíšu do súborov vo formáte **csv** (comma-separated values), ktorý uľahčuje neskoršie spracovanie týchto údajov v *spreadsheet* aplikáciach. Skripty sú tiež súčasťou priloženého CD.

B.1 Skript test.sh

Tento skript vykoná kompresiu a dekompresiu na všetkých súboroch korpusu a zmerá čas kompresie, dekompresie a hodnotu kompresného pomeru. Každý test na individuálnom súbore pre každú metódu je predvolene vykonaný 10-krát a minimálny čas zo všetkých týchto behov je považovaný ako výsledný čas. Namerané časy kompresie, dekompresie a hodnoty kompresného pomeru sa zapíšu do súborov:

- **compression_time.csv**,
- **decompression_time.csv**,
- **compression_ratio.csv**.

Časy sú namerané v milisekundách a kompresný pomer je v podobe čísla zaokrúhleného na tri desatinné miesta. Je tiež dôležité dávať pozor na to, že v prípade, že súbory s takýmito menami už existujú, budú týmto skriptom prepísané.

Skript by sa mal volať z adresára, ktorý obsahuje zostavený **Maven** projekt knižnice SCT. Spustiteľná knižnica by sa mala nachádzať vo formáte **jar** na relatívnej ceste **target/sct-RELEASE.jar**. V adresári skriptu by sa mal tiež nachádzať adresár obsahujúci súbory korpusu, predvolene to je **Pražský korpus** v adresári **prague**.

Skript je možné spustiť pomocou príkazu:

```
./test [parametre]
```

Možné parametre sú nasledujúce:

- **-h** – Zobraz nápovedu.
- **-r R** – Každá operácia bude opakovaná R-krát. Predvolená hodnota je **10**.
- **-f** – V prípade, že sa kompresia a dekompresia nevykoná na niektorom súbore úspešne, to znamená, že originálny súbor nebude úplne rovnaký ako ten dekomprimovaný, testovanie sa neukončí. Hodnota kompresného pomeru sa nahradí znakom „?“.
- **-d D** – Testovanie bude vykonané na všetkých súboroch, ktoré adresár D obsahuje. Predvolený adresár je **prague**.
- **-e E** – Relatívna cesta k zostavenej knižnici SCT vo formáte **jar**. Predvolená cesta je **target/sct-RELEASE.jar**.
- **-w** – Metóda LZW bude súčasťou testovania.
- **-m** – Metóda LZMW bude súčasťou testovania.
- **-p** – Metóda LZAP bude súčasťou testovania.
- **-y** – Metóda LZY bude súčasťou testovania.

Príklad použitia:

```
./test.sh -r 15 -f -d ./pragueCorpus -w -m -p -y
```

Tento príkaz vykoná testovanie metód LZW, LZMW, LZAP a LZY na súboroch nachádzajúcich sa v adresári **pragueCorpus**, pričom každý test metódy na súbore je vykonaný 15-krát a testovanie bude pokračovať aj napriek prípadným neúspešným testom.

B.2 Skript test_parameter.sh

Tento skript je podobný skriptu **test.sh**, ale namiesto vykonávania testov niekoľkých kompresných metód naraz, vykonáva testovanie špecifickej metódy a testuje jej výkon na súboroch korpusu pre rôzne hodnoty jej špecifikovaného nastaviteľného parametra. Výsledky testov vyprodukuje v podobnom formáte ako skript **test.sh**, ale súbory pomenuje vo forme:

- **compression_time_<metóda>_<parameter>.csv**,
- **decompression_time_<metóda>_<parameter>.csv**,
- **compression_ratio_<metóda>_<parameter>.csv**.

Skript je možné spustiť pomocou príkazu:

```
./test_parameters.sh [parametre]
```

Možné parametre sú nasledujúce:

- **-h** – Zobraz nápovedu.
- **-r R** – Každá operácia bude opakovaná R-krát. Predvolená hodnota je **10**.
- **-f** – V prípade, že sa kompresia a dekompresia nevykoná na niektorom súbore úspešne, to znamená, že originálny súbor nebude rovnaký ako ten dekomprimovaný, testovanie sa neukončí. Hodnota kompresného pomeru sa nahradí znakom „?“.
- **-d D** – Testovanie bude vykonané na všetkých súboroch, ktoré adresár D obsahuje. Predvolený adresár je **prague**.
- **-e E** – Relatívna cesta k zostavenej knižnici SCT vo formáte **jar**. Predvolená cesta je **target/sct-RELEASE.jar**.
- **-m** – Testuj metódu LZMW.
- **-p** – Testuj metódu LZAP.
- **-y** – Testuj metódu LZY.
- **-c C** – Testuj na parameter C. Možné hodnoty parametra sú **length**, **converter** a **partition**.

Príklad použitia:

```
./test.sh -r 10 -m -c length
```

Tento príkaz vykoná testovanie metódy LZMW na parameter **length**, pričom každý test na individuálnom súbore bude spustený 10-krát.

Dodatok C

Súbory Pražského korpusu

Tabuľka C.1: Zoznam súborov Pražského korpusu [1]

File	Size [Bytes]	Description	Type
firewrks	1,440,054	Sound of fireworks	Audio
thunder	3,172,048	Sound of thunder	Audio
drkonqi	111,056	KDE crash handler	Binary
libc06	48,120	A dynamic-link library	Binary
mirror	90,968	A part of the software package	Binary
abbot	349,055	Part of interior design application	Binary
gtkprint	37,560	A shared object	Binary
wnvcrdt	328,550	A database file	Database
w01vett	1,381,141	A database file	Database
emission	2,498,560	Waterbase emissions data	Database
bovary	2,202,291	Gustave Flaubert: *Madame Bovary*, in German	Documents
modern	388,909	Axel Lundegård, Ernst Ahlgren: *Modern. En berättelse*, in Swedish	Documents
ultima	1,073,079	Mack Reynolds: *Ultima Thule*, in English	Documents
lusiadas	625,664	Luís Vaz de Camões: *Os Lusíadas*, in Portuguese	Documents
venus	13,432,142	Ultraviolet image of Venus' clouds	Graphics
nightsht	14,751,763	A photo of a city at night	Graphics
flower	10,287,665	A photo of a flower	Graphics
corilis	1,262,483	CORILIS land cover data	Graphics
cyprus	555,986	Air Quality Monitoring in Cyprus	Markup languages
hungary	3,705,107	Air Quality Monitoring in Hungary	Markup languages
compress	111,646	Wikipedia page about data compression	Markup languages
lzfindmt	22,922	C source code from a file archiver	Scripts
render	15,984	C++ source code from an action game	Scripts
handler	11,873	Java source code from the GPS tracking system	Scripts
usstate	8,251	Java source code from the GPS tracking system	Scripts
collapse	2,871	JavaScript source code from the project management framework	Scripts
xmlevent	7,542	PHP source code from the calenedar generator	Scripts
mailflder	43,732	Python source code from the ECM framework	Scripts
age	137,216	Age structure in the world	Spreadsheets
higrowth	129,536	Financial calculations	Spreadsheets
Total	58,233,774		

Dodatok D

Detailne testovacie výsledky

Tabuľka D.1: Detailne **časy kompresie** všetkých testovaných kompresných metód na súboroch Pražského korpusu v **milisekundách**

Súbor	**LZW**	**LZAP**	**LZMW**	**LZY**
abbot	1693	549	601	646
age	1420	270	265	318
bovary	2200	987	991	1593
collapse	1318	122	120	134
compress	1559	266	231	287
corilis	2080	926	979	1260
cyprus	1470	285	274	458
drkonqi	1406	239	251	276
emission	2129	701	818	1196
firewrks	2445	1434	1563	1858
flower	6060	4509	5301	7649
gtkprint	1352	183	188	218
handler	1333	137	143	148
higrowth	1426	253	269	315
hungary	2079	749	776	1484
libc06	1360	196	193	236
lusiadas	1655	418	459	616
lzfindmt	1338	152	160	172
mailflder	1352	169	179	217
mirror	1404	229	241	268
modern	1602	369	404	557
nightsht	8953	10124	11764	14570
render	1340	145	151	155
thunder	3123	2061	2456	3565
ultima	2045	894	921	1220
usstate	1318	130	137	138
venus	7340	7990	8532	12108
w01vett	1689	411	437	705
wnvcrdt	1396	232	235	328
xmlevent	1316	136	138	139

Tabuľka D.2: Detailne **časy dekompresie** všetkých testovaných kompresných metód na súboroch Pražského korpusu v **milisekundách**

Súbor	**LZW**	**LZAP**	**LZMW**	**LZY**
abbot	1759	595	574	791
age	1437	264	259	327
bovary	2621	1334	987	1453
collapse	1271	87	90	93
compress	1553	173	167	295
corilis	2747	1087	975	1132
cyprus	1436	267	244	380
drkonqi	1405	209	192	250
emission	2877	833	753	1094
firewrks	3903	2230	1583	2574
flower	14729	5498	5670	9728
gtkprint	1315	140	135	147
handler	1290	100	101	109
higrowth	1446	248	240	292
hungary	1946	793	742	1220
libc06	1339	151	150	172
lusiadas	1733	444	455	576
lzfindmt	1308	120	114	131
mailflder	1318	129	132	149
mirror	1384	194	186	240
modern	1545	412	376	521
nightsht	14647	13046	12790	19847
render	1292	106	108	115
thunder	4099	2386	3745	4519
ultima	2939	1304	1015	1210
usstate	1285	94	96	102
venus	11413	8813	9028	15776
w01vett	1580	401	445	558
wnvcrdt	1354	205	193	250
xmlevent	1276	94	95	96

Tabuľka D.3: Detailne hodnoty **kompresných pomerov** všetkých testovaných kompresných metód na súboroch Pražského korpusu

Súbor	**LZW**	**LZAP**	**LZMW**	**LZY**
abbot	1.524	1.022	1.022	1.027
age	0.75	0.495	0.474	0.508
bovary	0.61	0.386	0.331	0.394
collapse	1.255	0.745	0.786	0.768
compress	0.554	0.288	0.267	0.328
corilis	0.847	0.581	0.569	0.587
cyprus	0.171	0.083	0.042	0.122
drkonqi	0.761	0.463	0.453	0.488
emission	0.434	0.159	0.135	0.201
firewrks	1.376	1.025	1.026	1.027
flower	0.947	0.67	0.659	0.656
gtkprint	0.725	0.408	0.43	0.432
handler	0.786	0.393	0.43	0.439
higrowth	0.844	0.5	0.5	0.527
hungary	0.148	0.088	0.033	0.131
libc06	0.779	0.474	0.492	0.487
lusiadas	0.638	0.377	0.358	0.375
lzfindmt	0.735	0.345	0.374	0.397
mailflder	0.592	0.298	0.318	0.331
mirror	0.831	0.495	0.504	0.515
modern	0.624	0.45	0.406	0.451
nightsht	1.129	1.011	0.995	1.002
render	0.805	0.407	0.434	0.446
thunder	1.064	0.841	0.829	0.831
ultima	1.152	0.74	0.741	0.743
usstate	0.808	0.416	0.452	0.454
venus	0.909	0.792	0.794	0.77
w01vett	0.162	0.083	0.074	0.102
wnvcrdt	0.142	0.066	0.06	0.077
xmlevent	0.929	0.486	0.52	0.532

Dodatok **E**

Obsah priloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `sct` ..... maven projekt knižnice SCT s metódami LZMW, LZAP a LZY
  - `prague` ..... súbory Pražského korpusu
  - `test.sh` ..... skript porovnania výkonu kompresných metód
  - `test_parameters.sh` . skript merania vplyvu nastaviľných parametrov
- `text` ..... text práce
  - `BP_bobot_jan_2018.pdf` ..... text práce vo formáte PDF
  - `src` ..... zdrojová forma práce vo formáte LaTeX