FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Finite State Entropy kodér pro knihovnu SCT
Student:	Bc. Ladislav Zemek
Vedoucí:	Ing. Radomír Polách
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Prozkoumejte princip asymetrických číselných systémů a kompresní algoritmus Finite State Entropy.
Analyzujte, navrhněte a implementujte Finite State Entropy kodér pro knihovnu SCT.
Implementujte dva algoritmy založené na Finite state entropy kodéru: LZFSE a jinou vhodně zvolenou metodu.
Zajistěte, aby bylo možní Finite State Entropy kodér použít jako náhradu za Aritmetický kodér.
Otestujte implementaci vůči algoritmům založených na Aritmetickém kodéru.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 19. února 2021 v Praze.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Softwarového inženýrství

Diplomová práce

Finite State Entropy kodér pro knihovnu SCT

Bc. Ladislav Zemek

Vedoucí práce: Ing. Radomír Polách

5. května 2021

Poděkování

Rád bych poděkoval Ing. Radomírovi Poláchovi za pomoc a vstřícnost při vedení diplomové práce, Evě Filipovské za korekci textu a v neposlední řadě rodině a přátelům za psychickou a morální podporu v těžkých chvílích.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 5. května 2021 .....

Odkaz na tuto práci

Zemek, Ladislav. *Finite State Entropy kodér pro knihovnu SCT*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato diplomová práce se zabývá analýzou, návrhem, implementací a testováním algoritmů založených na metodě Finite State Entropy (FSE). Konkrétně se jedná o metodu LZFSE a ZStandard. Obě tyto metody jsou implementované v knihově SCT. Tato knihovna by měla v budoucnu nabídnout širokou škálu kompresních algoritmů. Díky mé implementaci FSE kodéru má nyní uživatel možnost vybrat si, jakým způsobem budou data po kompresi kódována. Doteď bylo možné použít pouze Aritmetický kodér. Následné kódování tripletů je stěžejní pro dosažení dobrého kompresního poměru. Všechny výše zmíněné algoritmy se mi podařilo naimplementovat s dobrými výsledky.

Tato práce přispívá do knihovny SCT, jejímž cílem je poskytnout rozmanitou nabídku kompresních algoritmů.

Klíčová slova Komprese, Dekomprese, FSE, Finite State Entropy, LZFSE, ZStandard, SCT, Small Compression Toolkit, Java, ANS

Abstract

This diploma thesis deals with the analysis, design, implementation and testing of FSE-based algorithms. Specifically, it is the LZFSE and ZStandard methods. Both of these methods are implemented in the SCT library. This library should provide a wide range of compression algorithm. Thanks to my implementation of the FSE coder, the user now has the opportunity to choose how the data will be encoded after compression. Until now, only the Arithmetic Encoder could be used in SCT. Subsequent encoding of triplets is crucial to achieving a good compression ratio. I managed to implement all the above algorithms with good results. This work contributes to the SCT library, whose goal is to provide a diverse range of compression algorithm.

Keywords Compression, Decompression, FSE, Finite State Entropy, LZFSE, ZStandard, SCT, Small Compression Toolkit, Java, ANS

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Žijeme v digitálním světě a množství dat, která potřebujeme uchovávat je obrovské a neustále narůstá. Datová úložiště se rychle plní. Stále více je potřeba data přenášet pomocí internetové sítě, její rychlost muže být značně omezená. V minulosti tedy vznikla potřeba data zmenšit, aby se práce s nimi zjednodušila a zefektivnila. Přesně takový účel plní kompresní metody. Téměř všechny kompresní metody jsou založené na faktu, že uspořádání dat není ideální a obsahuje redundance, jež je možné nějakým způsobem využít k požadovanému zmenšení souboru. Redundance si lze představit například jako opakující se bloky dat, které je následně možné vynechat a pouze se odkázat na předchozí totožný blok. Takovým způsobem funguje nejznámější kompresní metoda LZ77.

Komprese dat se stala užitečnou a nedílnou součástí našich životů. Používají ji jak profesionálové v oblasti IT, tak i obyčejní uživatelé, kteří mnohdy nemají ani tušení, že byl jejich soubor zkomprimován. Není překvapení, že díky velkému využití kompresních algoritmů neustále probíhá výzkum a hledání nových a ještě efektivnějších algoritmů. To dokládá i fakt, že hlavní algoritmus, kterým se tato práce zabývá, vznikl teprve pred pár lety, v roce 2015, jde tedy o novinku ve světě komprese. Jedná se o Finite state entropy algoritmus. Ten vychází z myšlenky metody Asymetric numeral systems (ANS) redukovat redundance pomocí nejnižších bitů, narozdíl od Aritmentického kódování, které se naopak zameřuje na nejvyšší bity. Princip FSE je popsán dále v této práci.

Díky implementaci FSE do knihovny SCT [3] bylo možné doimplementovat metody LZFSE a ZStandard. Obě tyto metody FSE algoritmus využívají pro kódování svých výstupů, aby dosáhly ještě lepšího výsledku.

Obsah této práce popisuje princip, analýzu, implementaci a testování výše zmíněných algoritmů. LZFSE a ZStandard byly porovnány s jinými metodami, které jsou již v knihovně naimplementovány. Obě metody dosáhly dobrých výsledků komprese v kratším čase, než algoritmy používající aritmetický kodér, což byl očekávaný výsledek.

Kapitola 1

Pojmy

Tato kapitola obsahuje souhrn pojmů a definic, které jsou pro pochopení této diplomové práce důležité.

- **Komprese/komprimace dat** – Zpracování počítačových dat s cílem zmenšit jejich objem při současném zachování informací v datech obsažených.
- **Bezztrátová komprese dat** – Komprese dat, při které nedochází ke ztrátám informace.
- **Dekomprese/dekomprimace dat** – Inverzní operace ke kompresi. Ze zmenšených dat obnoví původní.
- **Triplet** – V této bakalářské práci je tripletem myšlena jakákoliv n-tice. Je tomu tak proto, že v knihovně SCT je každá n-tice zpracovávána pomocí třídy `TripletProcessor`. Nejedná se tedy vždy o trojici, jak by si mohl čtenář myslet. V závislosti na metodě může jít například i o čtveřici (LZFSE).
- **Adaptivní kompresní metoda** – Tyto metody si vytváří struktury za běhu programu v závislosti na vstupních datech. Vytvořené struktury jsou používány ke komprimaci dat a není potřeba je ukládat společně s daty. [1]
- **Slovníkové kompresní metody** – Tyto metody používají pro komprimaci slovník, který v průběhu komprimace i dekomprimace roste přidáváním nových frází. Proto patří slovníkové metody mezi adaptivní kompresní metody. [1]
- **Symetrická kompresní metoda** – Komprimace i dekomprimace má stejnou složitost. [1]

- **Asymetrická kompresní metoda** – Komprimace i dekomprimace má odlišnou složitost. [1]
- **Entropie** – Míra neurčitosti systému. [2] V kontextu této práce se jedná o míru náhodnosti dat.
- **Abeceda** – Množina symbolů.
- **Řetězec** – Posloupnost symbolů z abecedy.
- **Délka řetězce** – Počet symbolů v řetězci.
- **Korpus** – Soubor dat vytvořený pro účely testování kompresních metod.
- **Kompresní poměr** – Udává efektivitu, s jakou se podařilo zkomprimovat data. Vypočte se jako:

  $$K = \frac{v_k}{v},$$

  kde v_k je velikost dat po kompresi a v je velikost původních dat. Pokud je hodnota $K < 1$, pak byl soubor zmenšen. Pokud je $K \geq 1$, data se nezmenšila a komprese byla neúspěšná.
- **Modul** – V kontextu této práce nazveme jako modul část kódu, která je na zbytku nezávislá a je možné ji použít v ostatních naimplementovaných metodách knihovny SCT, pokud to dovolují.
- **Řetězení/chain/chainovaní** – Knihovna SCT implementuje princip Chain-of-responsibility pattern, který umožnuje řetězení kompresních metod a modulů libovolně za sebe.

Kapitola 2

Cíl práce

Cílem této práce je popis, analýza a implementace algoritmu FSE a jeho následné využití při implementaci metod LZFSE a ZStandard. Dále pak testování a vyhodnocení efektivnosti těchto algoritmů v porovnání s již implementovanými metodami v knihovně SCT, které využívají pro následnou komprimaci aritmetický kodér. Testování bude probíhat na standardních datasetech, které se pro hodnocení komprimace obvykle používají. Hodnotícími kritérii pro tyto metody budou časová složitost a paměťová náročnost.

FSE metoda bude implementována jako modul, bude jí tedy možné zaměnit za aritmetický kodér u jakékoliv již existující kompresní metody v SCT. Nové algoritmy obohatí tuto knihovnu a poskytnou uživateli rozmanitost při výběru vhodné kompresní metody.

2.1 Existující řešení

Přesto, že FSE je poměrněrně nová metoda, rychle našla své využití u dvou velkých firem. Yann Collet ve společnosti Facebook vyvinul algoritmus ZStandard, který staví na metodě LZ77 a její výstup kóduje pomocí FSE algoritmu. Elektronický gigant Apple Inc. také na této metodě postavil svůj algoritmus jménem LZFSE, který modifikuje metodu LZ77 a pro kódování jejího výstupu také využívá FSE. Obě tyto metody jsou open source, takže nepodlehají autorským právům.

Metoda LZFSE je plně implementována v systémech macOS, iOS a Linux. Podobně je na tom i metoda ZStandard, která je hojně rozšířena skrze téměř všechny operační systémy. Představení obou algoritmů proběhlo v roce 2015, jedná se tedy o novinku v kompresních algoritmech. Přesto již vznikla řada implementací. Příkladem je implementace LZFSE v knihovně EXCom. Tuto knihovnu v rámci své práce vytvořil Filip Šimek v roce 2009 na ČVUT. Knihovna je psaná v jazyce C a je zaměřená na minimalizaci časové náročnosti algoritmů. LZFSE do ní byla implementována později v roce 2018 Martinem Hronem.

Celá implementace ZStandard přímo od Facebooku je volně k dispozici v repozitáři zde [9]. Je velmi propracovaná a poskytuje možnost volby velkého množství parametrů. Implemenace LZFSE od Applu je dostupná na v repozitáři na adrese zde [10].

2.2 Analýza požadavků

Každý softwarový projekt začíná sběrem a analýzou požadavků. Tedy konkrétně toho, co si zákazník přeje a očekává od vzniklého softwaru. V tomto případě je zákazníkem zadavatel práce. Požadavek musí být proveditelný, dostatečně definovaný pro účely návrhu a testovatelný. Požadavky se obvykle rozdělují na funkční a nefunkční.[4]

2.2.1 Funkční požadavky

Funkční požadavky určují, jaké má mít software funkce a jak se má chovat v určitých situacích. Na základě zadání a konzultace s vedoucím práce jsem určil tyto funkční požadavky:

- **Modulární implementace FSE** – FSE bude implementováno takovým způsobem, aby šlo jednoduše zaměnit za aritmetický kodér tam, kde se používá.
- **Implementace LZFSE a ZStandard** – Implementace dvou algoritmů, jejichž algoritmus využívá modul FSE.
- **Dekomprese FSE, LZFSE a ZStandard** – Přirozeně je potřeba doimplementovat i dekompresní část všech zmíněných algoritmů.
- **Řetězení nových metod** – Metody LZFSE a ZStandard budou implementovat princip řetězení podobně jako ostatní metody v SCT. Tato knihovna umožnuje uživateli libovolně kombinovat jiné kompresní metody a další modul, což umožňuje právě implementovaný princip řetězení.
- **Parametrizovatelnost** – Všem algoritmům bude možné měnit jejich parametry, například velikost bufferů a podobně, aby bylo možné testovat jejich závislost na výkonu komprese případně dekomprese. Parametry bude možné upravovat pomocí třídy přímo určené pro nastavení metody a pomocí uživatelského klienta spustitelného z příkazové řádky.
- **Spustitelnost** – Součástí práce bude implementace klienta pro LZFSE a ZStandard, který poskytne uživateli možnost spustit kompresi dat s vlastními parametry.

- **Komprimace po částech** – Metody LZFSE a ZStandard budou komprimovat soubor po libovolně velkých částech. Velikost těchto částí je dána parametrem při spuštění komprimace. Algoritmy dekomprese budou schopny dekomprimovat soubor bez tohoto parametru, protože FSE bude komprimovat všechna data jako celek. Jedná se tak o možnost pro uživatele omezit algoritmu velikost paměti, kterou bude využívat.

2.2.2 Nefunkční požadavky

Nefunkční požadavky přímo neříkají, co má software umět. Spíše se jedná o omezující a zpřesňující podmínky, kterými se následně implementace řídí. Následující požadavky jsem určil jako nefunkční:

- **Dodržení principů SCT knihovny** – Implementace se bude držet všech použitých paternů a frameworků, které jsou v SCT použity.
- **Jazyk Java** – Veškerá implementace bude provedena v jazyce Java.
- **Triplety** – Algoritmy LZFSE a ZStandard budou produkovat výstupy v podobě n-tic, tzv. tripletů.
- **Použití tříd `TripletProcesor` a `TripletFieldID`** – Ukládání do souboru bude zprostředkováno v metodě FSE, která bude implementovat interface `TripletProcesor`. Definování složek tripletů bude zajištěno pomocí třídy `TripletFieldID`.
- **Kontrola smysluplnosti parametrů** – Zadané parametry projdou testy, které zajistí, že kombinace takových parametrů je smysluplná a algoritmus lze s takovými parametry spustit. Například velikost automatu u FSE musí být vždy větší nebo rovna počtu symbolů, které budou pomocí dané instance FSE zakódovány. V opačném případě není možné FSE použít a uživateli bude zobrazena chyba.

Kapitola 3

Popis a analýza algoritmů

Tato kapitola se zabývá popisem implementovaných algoritmů a dalších, které jsou zásadní pro jejich pochopení nebo jinak souvisí s touto prací. Například pro pochopení metody FSE je potřeba vysvětlit základní princip Asymetric numeral systems (ASN), na kterém tato metoda staví.

Jako další algoritmy, které souvisejí s touto prací, jsou aritmetický kodér a Huffmananovo kódování. S těmito metodami bude implementace FSE srovnávána v sekci testování a proto je potřeba alespoň stručně nastínit jejich princip a silné a slabé stránky.

Součástí analýzy algoritmů bude i odhad jejich časové a paměťové náročnosti. Dále pak zhodnocení jejich výhod a nevýhod.

3.1 Metoda FSE

Jedná se o poměrně složitou metodu, pro jejíž pochopení je potřeba znalost některých pojmů týkajících se oborů statistiky a komprese dat. Před samotným popisem algoritmu a jeho analýzou vysvětlím některé důležité pojmy.

3.1.1 Entropie[2]

Entropie je jedním ze základních a nejdůležitějších pojmů ve fyzice, teorii pravděpodobnosti a teorii informace, matematice a mnoha dalších oblastech vědy teoretické i aplikované. Pro kompresní metody je důležitá především entropie dat. Jde o míru neurčitosti systému, tedy o způsob, jak měřit náhodnost dat. Ta se v kontextu této práce hodí například při komentování efektivnosti jednotlivých algoritmů. Pro kompresní algoritmy obvykle platí, že čím vyšší entropie dat, tím horší efektivita algoritmu. Informační entropie se často nazývá Shannonova entropie po Claude Elwoodovi Shannonovi, který zformuloval mnoho klíčových poznatků teoretické informatiky.

Jde o střední hodnotu informace připadající na jeden symbol generovaný stochastickým zdrojem dat. Míra informační entropie řetězce S se vypočítá

pomocí nasledujícího vzorce:

$$H(S) = -\sum_i P_i log_2 P_i,$$

kde P_i je pravděpodobnost výskytu i-tého symbolu v řetězci S.

S pojmem entropie úzce souvisí pojem redundance. Ta se dá vypočítat následujícím vzorcem:

$$R(S) = L(S) - H(S),$$

kde $L(S)$ je délka řetězce S a $H(S)$ je jeho entropie. $R(S)$ tedy říká, kolik bitů je použito navíc proti ideálnímu případu při zachování stejné informace. Úkolem kompresních algoritmů je minimalizace $R(S)$.

3.1.2 Huffmanovo kódování

Pro účely této práce není potřeba tento algoritmus znát podrobně. Důležité je, že se jedná o entropický algoritmus, který každý výskyt symbolu kóduje do stejného kódového slova. Žádné kódové slovo nikdy není prefixem jiného kódového slova. Díky této vlastnosti se dá v bitovém streamu snadno každý symbol rozpoznat.

Jedná se o jednoduchou a rychlou metodu. Na druhou stranu existují jiné metody, které dosahují lepšího kompresního poměru, například aritmetický kodér nebo FSE. Podrobný popis algoritmu je možné nalézt zde. [5]

3.1.3 Aritmetický kodér

Aritmetický kodér je o poznání složitější než Huffmanův algoritmus. Symboly kóduje do čísla v intervalu $(0, 1)$, čím více se zakóduje informací, tím delší je desetinný rozvoj tohoto čísla a tím roste i potřeba bitů pro jeho reprezentaci.

Obecně platí, že aritmetický kodér je výpočetně náročnější než Huffmanův algoritmus, ale dosahuje lepších výsledků. Podrobnější popis tohoto algoritmu je možné nalézt zde. [6]

3.1.4 Princip ANS

Jde o entropickou kódovací techniku, kterou v roce 2009 představil Jaroslaw Duda. Jak sám autor uvedl, jedná se o metodu kombinující rychlost Huffmanova kódování a efektivitu aritmetického kodéru.

Algoritmy založené na tomto principu se používají například v softwaru společnosti Facebook (metoda ZStandard) nebo Applu (metoda LZFSE), ale také v Linux kernelu, operačním systému Android nebo třeba v nových formátech obrázků, například JPEG XL.[7]

Základní myšlenka je zakódovat informace do jednoho přirozeného čísla x. Ve standardním binárním numerickém systému můžeme přidat jeden bit informace $s \in (0, 1)$ do čísla x tak, že posuneme číslo x bitově o jeden bit

doleva a přičteme s. Matematicky se dá taková operace zapsat následujícím způsobem:

$$x' = 2x + s,$$

kde x' je nové přirozené číslo obsahující předešlé informace z x a informaci nesoucí s. Z pohledu entropického kodéru je takovýto postup ideální, pokud pravděpodobnost $P(0) = P(1) = \frac{1}{2}$. ANS zobecňuje tento postup pro libovolnou množinu symbolů $s \in S$ s příslušným pravděpodobnostním rozdělením (p_s).

Pokud je x' přirozené číslo, které vzniklo z jiného přirozeného čísla x připojením informací, které nese symbol s, pak platí

$$x' \approx x p_s^{-1}.$$

A ekvivalentně

$$log_2(x') = log_2(x) + log_2(\frac{1}{p_s}),$$

kde $log_2(x)$ je počet bitů informace uložené v x a $log_2(\frac{1}{p_s})$ je počet bitů informace v symbolu s.

Pro případ, kde $s \in (0, 1)$ se množina přirozených čísel rozdělila na sudá a lichá. To v obecném případně není možné. ANS proto rozdělí množinu přirozených čísel na disjunktní podmnožiny, jejichž hustota odpovídá pravděpodobnostní distribuci symbolů. Při přidání informace symbolu s k informacím uloženým v čísle x přejde na nové přirozené číslo následovně:

$$x' = C(x, s) \approx x p_s^{-1}.$$

Kódovací funkce $C(x, s)$ vrací x-tý výskyt symbolu s v příslušné podmnožině. Číslo x' tedy roste pomaleji, pokud přidáme symbol s s velkou četností a naopak roste rychleji, pokud má s malou pravděpodobnost výskytu. [8]

Jedná se o obecný princip, který je v praxi aplikován pomocí matematických postupů pro kódování a dekódování (jako u uANS a rANS algoritmů) nebo vytvořením konečného automatu, který usnadňuje výpočty komprese i dekomprese (například Finite state entropy – FSE). Posledním zmiňovaným se zabývá tato práce a bude podrobněji analyzován.

3.1.5 Popis Finite state entrophy[10]

Metoda Finite state entrophy implementuje princip ANS, ale místo matematických postupů pracuje s konečným automatem, ve kterém je uložen veškerý postup komprese i dekomprese. Díky tomuto postupu se celý algoritmus obejde bez násobení. Tato operace je ve srovnání se sčítáním časově náročnější a její absence je tedy výhodou. Místo ní se používá operace sčítání, odčítání a bitový posun, což jsou operace jednoduché pro procesor. Konečný automat je obvykle reprezentován tabulkou, proto se také metoda FSE někdy nazývá table ANS (tANS).

Jako první krok je potřeba určit frekvenci jednotlivých symbolů a vytvořit frekvenční tabulku. Následně je nutné frekvenční tabulku znormalizovat tak, aby součet frekvencí byl rovný velikosti dekódovacího automatu. Algoritmus pro normalizaci popisuji níže. Následně se pomocí znormalizované frekvenční tabulky vytvoří konečný automat pro kódování a zároveň i konečný automat pro dekódování. Dekódovací tabulka se zapíše do souboru na začátek, aby bylo možné data později dekódovat.

Následně se pomocí operací nad kódovací tabulkou zakódují symboly ze vstupu. Při každém zakódování se dopočítá, kolik je potřeba přidat bitů k číslu x, aby se do něj uložila informace, kterou symbol obsahuje. Číslo x se posune pomocí bitového posunu o potřebný počet bitů a přičte se číslo, které reprezentuje stav v kodóvací tabulce. Poslední stav je potřeba také zapsat do souboru, aby bylo možné data dekódovat. Celý postup komprese a dekomprese je podrobněji popsán dále.

3.1.6 Normalizace frekvenční tabulky

Algoritmus FSE vyžaduje, aby byl součet frekvencí symbolů roven velikosti dekódovací tabulky. Velikost dekódovací tabulky je volitelný parametr, který pro učely této práce nazvu L.

Každá frekvence se přenásobí parametrem L a vydělí celkovým součtem frekvencí. Jelikož frekvence musí být celočíselná, může dojít díky zaokrouhlování k situaci, kdy celkový součet normalizovaných frekvencí není rovný parametru L. Pro takovou situaci následuje další část algoritmu. Nejprve se spočítá rozdíl těchto dvou čísel:

$$D = L - T,$$

kde T je součet normalizovaných frekvencí.

Pokud je číslo D menší než 0, pak je potřeba některé frekvence zvýšit a pokud je větší než 0, tak snížit. To zajišťuje smyčka, která pro každé jednotlivé zvýšení nebo snížení vypočítá, pro který symbol je to nejvhodnější. Pokud je potřeba frekvence zvyšovat, pak takový symbol nalezne pomocí následujícího vzorce:

$$C = f_s * \log \frac{f'_s}{f'_s + 1},$$

kde f_s je původní frekvence symbolu s a f'_s je normalizovaná frekvence s. V případě snižování frekvencí bude ve vzorci ve jmenovateli znaménko minus.

Jako symbol, u kterého se provede korekce, vybere ten s nejnižší hodnotou C. Při takovém postupu by se mohlo stát, že se některá z frekvencí dostane na hodnotu 0. Takové chování je nežádoucí a je potřeba takový případ ošetřit. Tento případ je ošetřen tak, že při snižování frekvence se symboly s frekvencí jedna neberou do úvahy, jednoduše se přeskočí.

3.1.7 Komprese

Před samotnou kompresí dat je potřeba předpočítat konečný automat pro kódování. Ten je reprezentován tabulkou o velikosti $|f|$. Kde f je normalizovaná frekvenční tabulka.Pro každou buňku v tabulce je pak potřeba dopočítat s_0, k, $delta_0$ a $delta_1$. Hodnota s_0 udává hranici, podle které se řídí výpočet následujícího stavu. Slouží pro výběr mezi hodnotami $delta_0$ a $delta_1$ tak, aby při jejich použití nedošlo k přetečení hodnoty stavu mimo interval dekódovací tabulky.

Hodnota k slouží kromě výpočtu s_0 také k určení počtu bitů, které je potřeba z vypočteného stavu zapsat. Stav totiž díky předpočteným tabulkám není potřeba zapisovat celý, ale pouze několik nejnižších bitů. Následující pseudokód popisuje výpočet kódovací tabulky pro normalizovanou frekvenční tabulku f:

Input : normalizovaná frekvenční tabulka f, parametr L
Output: vyplněná kódovací tabulka t
$offset := 0$;
$t :=$ prázdná tabulka velikosti $|f|$;
$l :=$ počet nul na začátku bitového zápisu čísla L;
foreach *symbol* $s \in f$ **do**
    $freq = f[s]$;
    $l' :=$ počet nul na začátku bitového zápisu čísla $freq$;
    $k := l' - l$;
    $e.s_0 := (freq << k) - L$;
    $e.k := k$;
    $e.delta_0 := offset - freq + (L >> k)$;
    $e.delta_1 := offset - freq + (L >> (k-1))$;
    $offset := offset + freq$;
    $t[s] := e$;
end

Algoritmus 1: Pseudokód výpočtu kódovací tabulky FSE [10]

Algoritmus komprese následně vypočítá i hodnoty pro dekódovací tabulku, která bude následně uložena do souboru společně se zkomprimovanými daty. Bez této tabulky by data nebylo možné dekódovat. Tabulka má velikost $|L|$ a každá buňka obsahuje informace o symbolu, který reprezentuje. Dále pak hodnotu k a $delta$, kde k udává, kolik bitů ze vstupu je potřeba přečíst. Hodnota $delta$ souží k výpočtu následujícího stavu.

Dekódovací tabulka je konečný automat, který má $|L|$ stavů. Komprese FSE vypočítává přechody mezi těmito stavy a do souboru zapisuje číslo stavu. Díky předpočítaným hodnotám, které obsahuje i dekódovací tabulka, však stačí zapsat pouze některé nejnižší bity. Ty následně využije algoritmus dekomprese pro dopočítání celého čísla stavu. Podrobnější popis lze nalézt v sekci dekomprese. Následující pseudokód popisuje výpočet dekódovací tabulky pro normalizovanou frekvenční tabulku f:

```
t := prázdná tabulka;
l := počet nul na začátku bitového zápisu čísla L;
foreach symbol s ∈ f do
    freq = f[s];
    l' := počet nul na začátku bitového zápisu čísla freq;
    k := l' − l;
    j0 := ((2 ∗ L) >> k) − freq;
    i := 0;
    while i < freq do
        e.symbol := s;
        if j < j0 then
            e.k := k;
            e.delta := ((freq + j) << k) − L;
        else
            e.k := k − 1;
            e.delta := (j − j0) << (k + 1);
        end
        t.add(e);
        i + +;
    end
end
```

Algoritmus 2: Pseudokód výpočtu dekódovací tabulky FSE [10]

Samotné kódování dat je pak poměrně triviální. K zakódování symbolu je potřeba znát aktuální stav a kódovací tabulku. Nejdříve se porovná aktuální stav s předpočítaným hraničním stavem s_0 pro daný symbol. Podle výsledku se nastaví počet nejnižších bitů potřebný pro zapsání stavu do souboru. Dále se vypočítá nový stav pro kódování následujícího symbolu. Zde uvádím pseudokód popisující jeden kompresní krok:

```
Input  : symbol, aktualní stav s a kódovací tabulka t
Output: n nejméně významných bitů stavu s, nový aktualní stav s'
e := t[symbol];
if s < e.s0 then
    n := e.k − 1;
    delta := e.delta0;
else
    n := e.k;
    delta := e.delta1;
end
s' := (s >> n) + delta;
s' je nový stav pro kódování následujícího symbolu;
```

Algoritmus 3: Pseudokód kódování FSE [10]

3.1.8 Složitost komprese

Algoritmus komprese FSE je poměrně složitý a skládá se z několika kroků. Prvním je vytvoření frekvenční tabulky. Pro tento krok je potřeba projít celý soubor, který je komprimován. Složitost této operace je tedy

$$\Theta(n),$$

kde n je velikost dat. Operaci normalizace frekvenční tabulky je možné zanedbat, protože závisí na počtu různých symbolů a je tedy řádově méně časově náročná.

Dalšími operacemi jsou výpočet kódovací a dekódovací tabulky. Složitost je závislá na jejich velikosti a dala by se souhrnně napsat takto:

$$\Theta(|S| + |L|),$$

kde S je množina všech symbolů a L je uživatelem zadaný parametr velikosti dekódovací tabulky.

Posledním krokem je samotná komprese dat. Ta pochopitelně probíhá přes všechna data souboru označená výše jako n. Složitost je tedy také

$$\Theta(n).$$

Pokud sečteme složitosti všech operací, získáme výslednou složitost:

$$\Theta(n + |S| + |L| + n) = \Theta(2n + |S| + |L|).$$

Jelikož mluvíme o asymptotické složitosti, můžeme část s $|S|$ a $|L|$ vypustit, protože n nabývá řádově vyšších hodnot. Dostaneme výslednou asymptotickou složitost

$$\Theta(n).$$

Tato složitost odpovídá předpokladu, že se jedná o velmi rychlou metodu.

3.1.9 Dekomprese

Nejprve se ze souboru načte dekódovací tabulka t a poslední stav vypočtený při kompresi s. Pomocí stavu se dekóduje symbol a počet bitů, které je potřeba přečíst ze vstupu, aby byl správně rekonstruován následující stav v tabulce. Takto se dekóduje celý soubor do původní podoby před kompresí. Toto je pseudokód jednoho kroku dekomprese:

Input : dekódovací tabulka t, stav s
Output: dekódovaný *symbol*, nový stav s'
$n :=$ přečti $t[s].k$ bitů ze vstupu;
$s' := t[s].delta + n$;
$symbol = t[s].symbol$;

Algoritmus 4: Pseudokód dekódování FSE [10]

3.1.10 Složitost dekomprese

Určit složitost dekomprese je poměrně jednoduché. Složitost načtení dekódovací tabulky můžeme zanedbat, vzhledem k její velikosti. Zbývá tedy samotná operace dekódování symbolů. Ta se provede pro každý symbol původního souboru. Z toho plyne, že výsledná složitost bude

$$\Theta(n),$$

stejně jako u komprese. I přesto, že je komprese na první pohled složitější, asymptoticky budou tyto algoritmy shodné.

3.1.11 Volba parametru a omezení

Algoritmus FSE má pouze jeden parametr, které je možné měnit. V této práci je označen jako L a určuje velikost dekódovací tabulky a celkový součet frekvencí v normalizované frekvenční tabulce. V původní práci, kde byla FSE představena, byla použita hodnota 4096. Nicméně záleží na datech, která jsou kódována. V této práci je parametr L implementován tak, aby bylo možné ho snadno upravovat.

Tento parametr musí být vždy větší než velikost množiny kódovaných symbolů. Jinak by se do dekódovací tabulky nevešly všechny symboly a tedy logicky by je nebylo možné dekódovat. Navíc musí platit, že

$$L = 2^m,$$

kde m je přirozené číslo. Parametr L má vliv na efektivitu komprese.

K výběru vhodné hodnoty L a jeho vlivu na kompresní poměr se budu věnovat v kapitole testování.

3.1.12 Výhody

- **Rychlost komprese i dekomprese** – Složitost algoritmu je velmi dobrá. Vyrovná se Huffmanovu algoritmu, který je považován za rychlý kodér.

- **Dobrý kompresní poměr** – Podle autora tohoto algoritmu se kompresním poměrem dokáže rovnat aritmetickému kodéru, jehož kompresní poměr se v ideálních případech blíží teoretické hranici, s jakou je možné data zmenšit.

- **Jednoduché matematické operace** – Algoritmus při výpočtech nepoužívá matematické operace složité pro procesor, jako je například násobení. Místo něj využívá bitové posuny, sčítání a odčítání.

3.1.13 Nevýhody

- **Adaptivní verze** – Algoritmus FSE nelze implementovat adaptivně, respektive v takovém případě ztrácí veškeré výhody oproti jiným kompresním algoritmům. Především pak časová náročnost prudce vzroste.
- **Složitost implementace** – Jak je vidět z popisu výše, algoritmus je složitý a má několik mezivýpočtů. Jeho implementace je tedy složitější v porovnání například s Huffmanovým algoritmem nebo s adaptivním kódováním.

3.2 Metoda ZStandard

Jedná se o metodu bezztrátové komprese, kterou vyvinul Yann Collet pro společnost Facebook. První verze algoritmu byla zveřejněna v lednu 2015. Metoda kombinuje algoritmus LZ77 a kodér FSE. Tato kombinace nabízí srovnatelný kompresní poměr jako algoritmus DEFLATE, jeho rychlost komprimace i dekomprimace je však vyšší. [12] Tuto metodu podporuje program GNU tar a lze ji použít přepínačem –zstd.[11] Implementace přímo od společnosti Facebook je dostupná na GitHub repozitáři. [9]

3.2.1 Popis LZ77

Algoritmus vznikl v roce 1977. Jeho tvůrci jsou Abraham Lempel a Jakob Ziv. Jde o slovníkovou metodu, která je jednoprůchodová, bezztrátová a adaptivní.

Princip metody je založen na hledání opakujících se sekvencí v datech. Ty se následně zkomprimují pomocí odkazu do již zpacovaných dat. K tomuto postupu si algoritmus vytvoří posuvné okénko (floating window), kterým se postupně posouvá daty. Posuvné okénko je dále rozděleno na prohlížecí okno (search buffer) a aktuální okno (look-ahead buffer), kde prohlížecí část slouží jako slovník. Tento slovník obsahuje všechny podřetězce, které je možné z dat v prohlížecím okně vytvořit.

Algoritmus následně vyhledá nejdelší podřetězec z aktuálního okna. Ten pak zapíše v podobě tripletu jako index ve vyhledávacím okně, délku řetězce a nasledující symbol, který se liší.[13]

3.2.2 Popis ZStandard

Metoda má dvě části, kterým se říká frontend a backend. V frontend části se využije algoritmus metody LZ77. Ten zkomprimuje data do tripletů. Následuje backend část, která jednotlivé složky tripletu zakóduje pomocí FSE a uloží do souboru. Taková komprese je rychlá a výsledekem je dobrý kompresní poměr.

3.2.3 Komprese

Zde uvádím pouze pseudokód algoritmu LZ77. Metoda FSE byla podrobně popsána výše.

```
prohlížecí a aktuální okénka jsou prázdná;
while na vstupu jsou data do
    přidej na konec aktuálního okénka symbol ze vstupu;
    najdi prefix p z aktuálního okénka, který začíná v prohlížecím
     okénku;
    if p bylo nalezeno then
        zapiš trojici (i, j, k), kde i je vzdálenost prvního znaku
         nalezeného p od začátku aktuálního okénka, j je délka p a k
         je první nasledující symbol po p;
        posuň klouzavé okénko o j + 1;
    else
        zapiš triplet (0, 0, m), kde m je první symbol v aktuálním
         okénku;
        posuň klouzavé okénko o 1;
    end
end
```

Algoritmus 5: Pseudokód komprese metody LZ77 [14]

Místo zápisu se triplet pošle metodě FSE a ta každou složku zakóduje zvlášť. Takže každá složka má svou vlastní instanci FSE, která data kóduje. Kódování pomocí FSE se provede potom, co je celý soubor převeden na triplety. Z jednotlivých složek tripletu se vytvoří tři frekvenční tabulky, ty se znormalizují, vypočtou se kódovací a dekódovací tabulky, a následně jsou všechny triplety zakódovány a zapsány do souboru s příslučnou dekódovací tabulkou a konečným stavem.

3.2.4 Složitost komprese

Výpočet složitosti se skládá ze dvou částí. Složitosti frontendu a backendu. Frontend je algoritmus LZ77, který má složitost v nejhorším případě $O(mn)$, kde m je velikost prohlížecího okna a n je velikost vstupních dat.[15] Pro úplnost je potřeba zmínit, že za normálních okolností by se m zanedbalo jako konstanta. Nicméně se jedná o parametr, který roste exponenciálně a jeho hodnota je obvykle nezanedbatelná.

Backend je metoda FSE, která má složitost $O(n)$, kde n je v tomto případě počet tripletů. Ten je ale závislý na velikosti vstupních dat, proto pro jednoduchost uvažuji n jako velikost nezakódovaných dat. Metoda FSE se provede pro každou složku tripletu. Takže celková složitost bude $O(3n)$.

Po sečtení složitosti frontendu a backendu dostanu:

$$O(mn + 3n) = O((m + 3)n) = O(mn).$$

Z analýzy složitosti vyplývá, že asymptoticky je složitost ZStandard metody ovlivněná především metodou LZ77. FSE je tedy v porovnání s LZ77 tak rychlá, že na celkovou složitost nemá téměř vliv.

3.2.5 Dekomprese

Nejdříve se pomocí dekomprese FSE rekontruují triplety, následně se dekomprimují pomocí dekomprese LZ77 a do souboru se zapíší původní data. Podrobný popis dekódování FSE je popsaný výše v kapitole o této metodě. Zde je pseudokód dekomprese LZ77:

```
prohlížecí okénko je prázdné;
while na vstupu jsou data do
    ze vstupu načti triplet (i, j, k), kde i je vzdálenost prvního znaku
     hledaného podřetězce p od konce prohlížecího okénka, j je délka
     p a k je první nasledující symbol po p;
    if i se rovná 0 then
        zapiš k;
        přidej na konec prohlížecího okénka k;
    else
        v := aktuální velikost prohlížecího okénka;
        z := v − i;
        p := podřetězec prohlížecího okénka od indexu z az do z + j-;
        p := p + k;
        zapiš p;
        do prohlížecího okénka přidej p;
    end
end
```

Algoritmus 6: Pseudokód dekomprese metody LZ77[14]

Algoritmus pomocí informací z tripletu zkopíruje podřetězec z prohlížecího okna a na konec přidá symbol k. Prohlížecím oknem jsou v podstatě celá aktuálně dekódovaná data.

3.2.6 Složitost dekomprese

Nejdříve se dekomprimují složky tripletů. To odpovídá složitosti $O(3n)$, kde n je počet tripletů. Následná dekomprese LZ77 tripletů je rychlá a spočívá pouze v kopírování již dekomprimovaných dat. Složitost je tedy $O(n)$. Stejně jako u komprese se složitosti sečtou a dostaneme:

$$O(3n + n) = O(4n) = O(n),$$

kde n je počet tripletu. Dá se tedy říci, že dekomprese je závislá na kompresním poměru. Čím méně je potřeba tripletů, tím rychlejší bude dekomprese, ale jak bylo zmíněno výše, počet tripletů je závislý na velokosti dat, takže asymptoticky bude složitost stejná.

Z analýzy dekomprese plyne, že algoritmus je velmi rychlý. Složitost komprese a dekomprese metody ZStandard není stejná, což znamená, že se jedná o asymetrickou kompresní metodu.

3.2.7 Parametry a omezení

Tento algoritmus má následující parametry:

- **Velikost prohlížecího okna** m – Udává rozsah hodnot indexů v prohledávacím okně.
- **Velikost aktuálního okna** k – Říká, jaká je maximální velikost podřetězce, který se kopíruje pomocí tripletu.
- **Velikost dekódovacích tabulek** L_0, L_1, L_2 – Každá složka tripletu má jiný rozsah, takže pro každou instanci FSE může být nastavený jiný parametr L.

Platí, že prohlížecí okno musí být větší nebo rovno aktuálnímu oknu. Parametr L musí být vždy větší nebo roven rozsahu hodnot. Takže například pro složku indexy do prohlížecího okénka musí být alespoň m, pro délku kopírovaného řetězce zase alespoň k.

Hodnota parametrů se volí tak, aby se dala zapsat jako 2^n, kde n je přirozené číslo.

Jaké parametry jsou pro algoritmus nejvhodnější, popisuji v kapitole testování.

3.2.8 Výhody

- **Rychlost komprese i dekomprese** – Složitost algoritmu ZStandard je velmi dobrá. Překoná dokonce algoritmus DEFLATE.
- **Dobrý kompresní poměr** – Co se kompresního poměru týče, dosahuje ZStandard podobných výsledků jako DEFLATE.
- **Paměť** – Paměťová náročnost roste lineárně s velikostí vstupu.

3.2.9 Nevýhody

- **Parametry** – Algoritmus má mnoho parametrů a efektivita je silně závislá na jejich volbě.

3.3 Metoda LZFSE

Jedná se o bezztrátovou metodu vyvinutou společností Apple Inc. v roce 2015. Je plně implementovaná do všech operačních systémů této společnosti. Navíc je možné ji použít i na Linuxu. Apple uvádí, že kompresní poměr je porovnatelný s metodou DEFLATE, ale dekomprese je dvakrát až třikrát rychlejší, zatímco používá méně paměti a proto nabízí vyšší efektivitu. Právě efektivita dekomprese byla hlavní cíl při vývoji této metody.

Třetí strany při testování potvrdily, že LZFSE je skutečně efektivnější než implementace DEFLATE v knihovně zlib. Nicméně také uvádí, že jiné algoritmy mohou dosahovat lepších výkonostních charakteristik jak při kompresi, tak i dekompresi.

Tato metoda není vhodná pro malé soubory, proto Apple společně s LZFSE vydal i algoritmus LZVN. Implementace LZFSE od Applu mezi těmito algoritmy přepíná podle velikosti souboru. Tento algoritmus však není součástí této práce. Implementaci LZFSE i LZVN přímo od Applu v jazyku C je možné dohledat v repozitáři na GitHubu.[10]

Podle výkonnostních testů má LZFSE stejnou rychlost komprimace a dekomprimace jako ZStandard, avšak kompresní poměr je o něco horší.

3.3.1 Popis LZFSE

Metoda se, podobně jako ZStandard, skládá ze dvou hlavních částí. Beckendovou částí je algoritmus podobný LZ77. Metoda má také posuvné okénko rozdělené na prohlížecí a aktuální část. V každém kroku se pokusí najít nejdelší podřetězec z aktuálního okna v prohlížecím okně. Hlavní rozdíl je, že pokud se nepodaří najít podřetězec delší než k, pak se tento podřetězec přeskočí a algoritmus si zapamatuje, jak velká část byla přeskočena.

Pokud je následně nalezen dostatečně dlouhý podřetězec, zapíše se triplet, který má čtyři složky. První je počet přeskočených symbolů, druhý je řetězec přeskočených symbolů, třetí je index v prohlížecím poli a čtvrtý je délka nalezeného podřetězce. Oproti LZ77 také odpadá potřeba zaznamenat následující symbol.

Každá složka tripletu se pak zakóduje pomocí FSE algoritmu ve backendové části a následně se uloží do souboru. Řetězce se kódují po jednom bytu, protože díky předchozímu údaji z tripletu se snadno při dekompresi načte správný počet symbolů a tím se daný řetězec zrekonstruuje.

3.3.2 Komprese

LZFSE nejdříve vytvoří čtveřice, jak je popsáno výše. Čtveřice se ovšem rovnou nezapisují do souboru, ale každá složka je poslána instanci FSE, která ji zakóduje a uloží do souboru. Jednotlivé složky jsou tedy uloženy odděleně

a každá má svou dekódovací tabulku. V následujícím pseudokódu je naznačen princip hledání čtveřic u LZFSE:

```
prohlížecí a aktuální okénka jsou prázdná;
řetězec přeskočených symbolů str je prázdný;
while na vstupu jsou data do
    přidej na konec aktuálního okénka symbol ze vstupu;
    najdi nejdelší prefix p z aktuálního okénka, který začíná
     v prohlížecím okénku;
    if p bylo nalezeno then
        if |p| je menší než k then
            připoj p na konec str;
        else
            zapiš čtveřici (|str|, str, i, j), kde str je řetězec vynechaných
             symbolů, i je vzdálenost prvního znaku nalezeného p od
             začátku aktuálního okénka a j je délka p;
            vyprázdni str;
        end
        posuň klouzavé okénko o j + 1;
    else
        připoj m na konec str, kde m je první symbol v aktuálním
         okénku;
        posuň klouzavé okénko o 1;
    end
end
```

Algoritmus 7: Pseudokód komprese metody LZFSE [14]

Reálná implementace má navíc ještě podmínku, že pokud je délka přeskočených symbolů větší než k, zapíše se triplet i bez nalezení dostatečně dlouhého podřetězce. Hodnotu k je možné zvolit před spuštěním komprimace.

Takový triplet má tvar $(k, str, 0, 0)$ a jde tedy v podstatě pouze o překopírování dat. Nicméně díky následnému kódování pomocí FSE i při takovém tripletu může dojít ke zmenšení. FSE jako každý kodér kóduje nejkratší sekvencí nejčetnější symboly. Je velmi pravděpodobné, že symboly k a 0 budou velmi četné a jejich zápis nebude paměťově tolik náročný.

3.3.3 Složitost komprese

Algoritmus komprese LZFSE má asymptoticky totožnou složitost jako ZStandard. Složitost se liší pouze tím, že se kóduje čtyřmi instancemi FSE a ne pouze třemi. Tedy platí, že celková složitost je

$$O(mn + 4n) = O((m + 4)n) = O(mn),$$

což odpovídá předpokladu, že LZFSE je stejně rychlá jako ZStandard.

3.3.4 Dekomprese

Nejdříve se jednotlivé složky dekoprimují pomocí FSE dekomprese. Následně se rekonstruují čtveřice vzniklé při kompresi. První složka při dekompresi poslouží jako informace, kolik má instance FSE vydat symbolů pro konkrétní triplet. U ostaních složek je to vždy jeden symbol pro jeden triplet, ale u druhé složky se jedná o řetězec, takže je potřeba dekódovat dostatečný počet symbolů. Poté následuje samotná dekomprimace pomocí LZFSE popsaná následujícím pseudokódem:

```
prohlížecí okénko je prázdné;
while na vstupu jsou data do
    ze vstupu načti čtveřici (|str|, str, i, j), kde str je řetězec
      vynechaných symbolů, i je vzdálenost prvního znaku nalezeného
      p od začátku aktuálního okénka a j je délka p;
    přidej do prohlížecího okénka str;
    if i se nerovná 0 then
        v := aktuální velikost prohlížecího okénka;
        z := v − i;
        p := podřetězec prohlížecího okénka od indexu z az do z + j-;
        p := p + k;
        zapiš p;
        do prohlížecího okénka přidej p;
    end
end
```

Algoritmus 8: Pseudokód dekomprese metody LZFSE[14]

3.3.5 Složitost dekomprese

Podobně jako komprese i dekomprese má asymptotickou složitost stejnou jako ZStandard. Liší se pouze v množství použitých instancí FSE dekomprese. Po sečtení složitosti dekomprese částí LZFSE a FSE dostaneme:

$$O(4n + n) = O(5n) = O(n),$$

kde n je počet čtveřic, které vyprodukovala frontendová část LZFSE metody.

Výsledná asymptotická složitost odpovídá faktu, že rychlost LZFSE je srovnatelná s metodou ZStandard.

3.3.6 Volba parametrů a omezení

Tento algoritmus má následující parametry:

- **Velikost prohlížecího okna** m – Udává rozsah hodnot indexů v prohledávacím okně.

- **Velikost aktuálního okna** k – Říká, jaká je maximální velikost podřetězce, který se kopíruje pomocí tripletu.
- **Velikost dekódovacích tabulek** L_0**,** L_1**,** L_2**,** L_3 – Každá složka čtveřice má jiný rozsah, takže pro každou instanci FSE může být nastavený jiný parametr L.
- **Minimální délka podřetězce** l – Pokud je nejdelší nalezený podřetězec menší než l, je přeskočen a přidán do pomocné proměnné, kde vyčká, než bude zapsán společně s dalšími daty.
- **Maximální počet přeskočených symbolů** s – Udává hranici, kdy je potřeba čtveřici zapsat bez ohledu na to, jestli byl nalezen dostatečně dlouhý podřetězec.

Platí, že prohlížecí okno musí být větší nebo rovno aktuálnímu oknu. Parametr L musí být vždy větší než rozsah hodnot. Takže například pro složku indexy do prohlížecího okénka musí být větší než m, pro délku kopírovaného řetězce zase větší než k.

Hodnota parametrů se volí tak, aby se dala zapsat jako 2^n, kde n je přirozené číslo. Pouze hodnota l je obvykle malé číslo menší než 10.

Tato metoda má poměrně hodně parametrů a jejich vhodný výběr analyzuji a komentuji v kapitole testování.

3.3.7 Výhody

- **Rychlost komprese i dekomprese** – Složitost algoritmu LZFSE je srovnatelná s ZStandard.
- **Dobrý kompresní poměr** – Co se kompresního poměru týče, dosahuje LZFSE dobrých výsledků.
- **Paměť** – Paměťová náročnost roste lieneárně s velikostí vstupu.

3.3.8 Nevýhody

- **Parametry** – Algoritmus má celkem 8 parametrů a jejich správná volba je stěžejní pro dosažení dobré komprese. Je tedy potřeba provést analýzu vhodných hodnot těchto parametů.

3.4 Závěr analýzy

Analýza všech metod naznačuje, že se jedná o velmi efektivní a rychlé algoritmy, jejichž využití v praxi je reálné. Zásadní je volba parametrů, které velmi ovlivňují rychlost komprese, dekomprese a kompresní poměr. Je tedy potřeba se na jejich analýzu zaměřit a následně zvolit nejlepší hodnoty. Implementace

v knihovně SCT umožňuje měnit parametry libovolně podle potřeb uživatele. Nebude tedy problém takovou analýzu vytvořit. Více v kapitole o testování.

Kapitola 4

Implementace

Následující kapitola pojednává o samotné implementaci metod FSE, LZFSE a ZStandard do knihovny SCT. Dále pak o principech, kterými se řídí knihovna SCT, jako je například použití frameworku pro řetězení metod, popis načítání dat ze souboru a zápis zakódovaných dat, realizace n-tic (dále tripletů), reprezentace parametrů. Součástí implementace je také spustitelný klient, což je program, který dokáže pomocí příkazové řádky zkomprimovat i dekomprimovat soubor s parametry, které zadá uživatel.

4.1 Řetězení

Jeden z požadavků na implementaci algoritmu v knihovně SCT je možnost řetězit jednotlivé metody za sebe. Takový přístup umožnuje chain of responsibility pattern framework. Jedná se o způsob, jak rozdělit úkoly metodám a následně je pozapojovat podle potřeby. Například metoda LZFSE nejdříve načte data, ty zpracuje frontendovým algoritmem a výstup pošle backendovému algoritmu, tím je FSE metoda. Její výstup se pak uloží do souboru. Všechny tyto úkony jsou vykonávány v oddělených metodách. Zde je ukázka kódu:

```
ChainBuilder.create(io::openParse)
.chain(lzfse::compress)
.chain(t2b)
.end(bytes -> io.saveObject(bytes, f2.toPath()))
.accept(f1.toPath());
```

Tento přístup umožňuje například jednoduše vyměnit modul LZFSE frontendového algoritmu za LZ77 metodu a ihned dostávám metodu ZStandard. Ukázka kódu z SCT:

```
ChainBuilder.create(io::openParse)
.chain(zStandard::compress)
.chain(t2b)
```

Stačí tedy pouze doimplementovat kompresní metodu, protože ostatní části algorimu jsou již hotové. Je však potřeba navrhnout metodu tak, aby byl tento patern zachován. Řetězitelná metoda musí mít dva parametry. Prvním parametrem je vstupní objekt a druhým výstupní. Například u metod komprese je vstupní parametr objekt `ByteBuffer` a výstupním parametrem je `TripletSupplier`. Ten je obalen objektem Consumer. Zde je ukázka hlavičky metody `compress` v LZFSE:

```
public void compress(ByteBuffer byteBuffer,
    Consumer<TripletSupplier> tripletSupplierConsumer)
    {...}
```

Tento způsob je vhodný, pokud je potřeba řetězit pouze jednu metodu v celé třídě, případně více metod, ale každá má jiný vstup a výstup. V ZStandard i v LZFSE jsou vždy dvě řetězitelné metody `compress` a `decompress`. Další možnost, jak naimplementovat řetězení, je, že objekt implementuje interface `Chainable<in, out>`, kde `in` je vstupní objekt, například instance třídy `TripletSupplier` a `out` představuje výstupní objekt, například list polí bytů `List<byte[]>`. Zde je ukázka kódu abstraktní třídy `TripletToByteConverter`, ze které dědí `FSEEncoder`:

```
public abstract class TripletToByteConverter<T>
    implements Chainable<TripletSupplier, List<byte[]>>
    {...}
```

`FSEEncoder` je objekt, který kóduje triplety pomocí FSE komprese a vytvoří pole bytů, které je možné zapsat. To pak předá metodě v řetězci, která je za zápis zodpovědná. Řetězitelná metoda musí vždy implementovat metodu `accept` a `setConsumer`, aby byly splněny požadavky rozhraní `Chainable<>`. Pomocí metody `accept` pak objekt ukládá data do `consumera`. `Consumer` je vlastně takový sklad dat pro další metodu v řetězci.

4.2 Čtení a zápis dat

Metody LZFSE a ZStandard mají totožné čtení a zápis do souboru. Nicméně metody komprese a dekomprese se z tohoto pohledu zásadně liší. Zatímco kompresní metoda čte soubor po bytech, dekompresní metoda načítá celý objekt. Je tomu tak proto, aby zůstaly rozlišeny části tripletů a jejich načítání bylo jednodušší. Takový postup není paměťově náročný a dekomprimovaný soubor výrazně nezvětší.

Kompresní metoda tedy načte data po bytech, ale uloží celý objekt. Dekompresní metoda pak načte objekt ze souboru a po dekompresi uloží byty původních dat. Výsledkem je totožný soubor jako před komprimací.

O celou práci se soubory se stará objekt `FileIO`. Na počátku si uživatel pomocí parametru zvolí, po jak velkých kusech je potřeba zvolený soubor

načítat. Následně se pak tyto menší části zkomprimují na triplety, zakódují pomocí kodéru a uloží do souboru.

4.3 Triplety

Jak bylo již zmíněno, v kontextu této práce a knihovny SCT jsou triplety všechny n-tice. Ne tedy jen trojice, ale například i čtveřice jako u LZFSE. Knihovna SCT je reprezentuje pomocí třídy `TripletSupplier`, která se používá pro vytvoření anonymní vnitřní třídy. Ta má jako parametr instanci třídy `TripletProcessor`, pomocí něhož se volá metoda `write`, která zapisuje jednotlivé složky tripletu. Zde je ukázka kódu:

```
output.accept((TripletProcessor visitor) -> {
                visitor.write(distField, k);
                visitor.write(lengField, l);
                visitor.write(byteField, b);
            });
```

Takto se uloží triplet (k, l, b) pro následné zpracování. `Write` má jako první parametr objekt `TripletFieldId,` který určuje, o jakou část tripletu se jedná. Parametry `TripletFieldId` jsou index a počet bitů potřebný po zápis jakékoliv hodnoty. Například při velikosti prohlížecího okna 2^{10} je možné zapsat hodnotu vzdálenosti až $2^{10} - 1$ a pro takové číslo je potřeba 10 bitů. Parametr tedy bude mít hodnotu 10, protože ostatní čísla jsou menší a 10 bitů k zápisu postačí. Index je identifikátor konteineru, do kterého budou data přidána. V knihovně je definován například takto:

```
byteField = new TripletFieldId(0, Byte.SIZE);
distField = new TripletFieldId(1, 10);
```

Části tripletu tedy není potřeba zadávat střídavě tak, jak jdou za sebou. Při prvním přidání dat se pro každou část tripletu s rozdílným `TripletFieldId` vytvoří konteiner, do kterého jsou data ukládána. V dalším kroku jsou pak zpracována některým z kodérů (buď aritmetický kodér, Huffmanův kodér nebo nově přidaným FSE kodér).

4.4 Reprezentace parametrů

V knihovně SCT jsou všechny implementace kompresních algoritmů parametrizovatelné. Je tedy možné u nich nastavit téměř jakékoliv parametry, pokud jejich kombinace dává logický smysl.

Pro tento účel má každá metoda implementovanou třídu s názvem `<název metody>ProviderParametrs`. Tento objekt se předává jako jediný kontruktoru při vytváření instance kompresní metody. Parametry jsou popsány dále v implementační části a jejich dopad je vysvětlen v kapitole testování a analýzy algoritmů.

4.5 Spustitelný klient

Aby bylo pro uživatele možné metody snadno používat, je v knihovně SCT pro každou kompresní metodu třída s názvem `<název metody>Client.` Každá klient třída obsahuje statickou metodu `main`, je tedy možné ji spustit z příkazové řádky. Na základě parametrů, které zadá uživatel, dokáže klient komprimovat i dekomprimovat zadaný soubor. Uživatel pak může nastavit všechny ostatní parametry, které metoda má. Pokud jejich kombinace neporušuje některá pravidla pro danou metodu, komprese se provede se zadanými parametry. Pokud jsou zadány chybně, algoritmus vyvolá chybu.

Konkrétní klienti jsou popsáni u každé implementace jednotlivých metod. Vždy je aktivní pouze jeden klient a pro jeho změnu je potřeba zadat cestu k novému klientovi v souboru `pom.xml` a následně projekt zkompilovat.

Klient je po kompilaci umístěn ve složce `sct/targets` a dá se spustit příkazem:

```
java -jar sct-RELEASE.jar <input> <output> <options>
```

Parametrem `<input>` je určen soubor, který se má zkomprimovat, `<output>` je název nového zkomprimovaného souboru a `<options>` je sada přepínačů, kterými uživatel volí hodnoty parametrů. Popis těchto přepínačů je také popsán u jednotlivých metod.

Každá spustitelná metoda má ve složce se svým jménem ukázku použití implementace v souboru s názvem `<název metody>Test`. Této testovací metodě je možné změnit název souboru, který komprimuje a kodér, který kóduje vzniklé triplety. Tyto testovací třídy jsou vhodnou demonstrací funkčnosti implementace metod LZFSE a ZStandard v SCT. Nejdříve je potřeba knihovnu zkompilovat a následně některý ze zmiňovaných testovacích objektů spustit. Ten provede kompresi zvoleného souboru, kterou nazve `<název souboru>.dec` a dekompresi souboru, kterou pojmenuje jako komprimovaný soubor a přidá koncovku jména metody, například `calgary.dec.lzfse`.

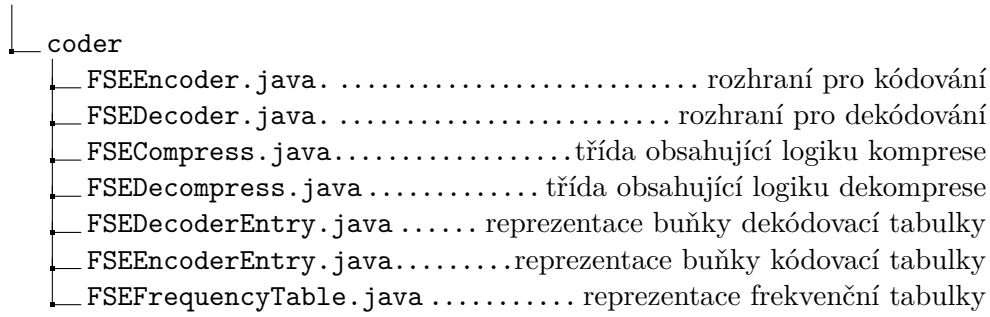
4.6 Realizace FSE jako modul

Jeden z požadavků na metodu implementaci metody FSE je snadná zaměnitelnost za jiný kodér. To se dá zajistit děděním a následnou implementací abstraktní parametrizované třídy `ByteToTripletConverter<T>`. Nový modul pak musí implementovat několik metod, jako například `write`, `read`, `compress` v případě kodéru a `decompress` v případě dekodéru a další.

Díky tomuto přístupu se následně dá všude, kde je použit Aritmetický kodér, který také dědí z `ByteToTripletConverter<T>,` nahradit za nově implementovaný modul FSE, jehož algoritmus je jiný, ale funkce z pohledu ostatních metod je stejná. Jednoduše vezme data, zakóduje je.

4.7 Realizace FSE

4.7.1 Struktura adresáře



Struktura 4.1: adresářová struktura metody FSE v SCT

4.7.2 Parametry

Metoda FSE potřebuje pouze jediný parametr a tím je velikost dekódovací tabulky. Obvykle je značen písmenem L a je velmi důležitý pro efektivnost algoritmu. V původní práci pojednávající o FSE algoritmu má L hodnotu 4096. Tato hodnota ale není jediná možná a pro některá měření, prováděná v kapitole testování, je nedostatečná.

Dá se říci, že je možné použít jakoukoliv hodnotu, která splňuje následující podmínky:

$$L = 2^n,$$

kde n je přirozené číslo a

$$L > |S|,$$

kde S je množina všech unikátních komprimovaných symbolů.

Paradoxně přestože L určuje velikost dekódovací tabulky, velký význam má i při kompresi, protože určuje celkový počet frekvencí v normalizované frekvenční tabulce. Z té se pak počítá kódovací i dekódovací tabulka.

Jeho hodnota ovlivňuje celkovou efektivitu a rychlost FSE algoritmu. Platí, že čím vyšší číslo L, tím větší časová složitost, ale také lepší kompresní poměr. Ten má ale vždy hranici, kdy se přestane zlepšovat. Je tomu tak proto, že čím větší je dekódovací tabulka, tím lépe se v ní projeví rozdíly ve frekvencích jednotlivých symbolů. Pokud je však tabulka již dostatečně velká, její zvětšení nepřidá žádné informace pro kompresi a efektivita stagnuje. Naopak dochází k lehkému zhoršení kompresního poměru, protože je potřeba zapsat do souboru větší dekódovací tabulku.

Analýza tohoto parametru je obsahem kapitoly testování.

4.7.3 Hlavička souboru

Každá instance FSE má svou vlastní hlavičku, která je potřeba pro následné dekódování souboru, bez nutnosti zadávat jakékoliv parametry. V hlavičce je uložena hodnota parametru L, celá dekódovací tabulka t a konečný stav komprese s.

4.7.4 Implementace komprese

Třída `FSEEncoder` je implementována tak, aby byla zaměnitelná s aritmetickým kodérem, který již v knihovně SCT funguje. V budoucnu bude ale možné ji zaměnit za jakýkoliv nově vzniklý kodér. Tato třída totiž dědí z třídy TripletToByteConverter a implementuje všechny jeho metody. Duležité jsou především metody `write`, `compress`, `createNew` a `terminate`.

Metoda `write` je volána při přidání dat. Jejím úkolem je vytvořit pomocí `createNew` novou instanci `FSECompress,` pokud pro dané `TripletFieldId` již nenexistuje. To znamená, že každá část tripletu má vlastní instanci FSE kodéru a data se tedy kódují nezávisle na sobě.

Dále je zavolána metoda `compress`. Ta na data zavolá funkci `increment` dané FSE instance. Jelikož se nejedná o adaptivní kodér, funkce `increment` uloží data do datového konteineru nazvaného `dataStore`, což je zásobník. Zásobník funguje na principu LIFO. Tedy data, která přijdou první, se budou kódovat jako poslední. Je tomu tak proto, že FSE při dekompresi data dekóduje v opačném pořadí, než je kóduje.

Proto jsem se rozhodl, že data rovnou prohodím při kompresi. Dekomprese je tedy pak dekomprimuje ve správném pořadí. Tato změna nijak neovlivní výpočetní náročnost nebo kompresní poměr. Metoda `increment` dále pro daný symbol upraví frekvenční tabulku, aby odpovídala četnosti symbolů. Tuto tabulku reprezentuje třída `FrequencyTable`.

`FrequencyTable` je třída obsahující tabulku všech kódovaných symbolů, jejich četnost a celkový počet v proměnné `total`. Dále pak nabízí několik pomocných funkcí, jako je například metoda výpočtu normalizované frekvenční tabulky.

Poté, co doběhne fáze přidávání tripletů, se zavolá metoda `terminate`. Ta spustí metodu `compress,` která spustí finální kódování tripletů.

Nejprve se provede normalizace frekvenční tabulky. Tu provede metoda `normalize` ve třídě `FrequencyTable`. Popis normalizace je vysvětlen ve třetí kapitole o algoritmu FSE. Vytvořená normalizovaná tabulka je uložena v proměnné `normalizedFrequencyTable` ve třídě `FrequencyTable.`

Dále se vypočte kódovací tabulka pomocí metody `computeEncodeTable` a dekódovací tabulka pomocí `computeDecodeTable.` Oba postupy jsou také podrobně popsané ve třetí kapitole o metodě FSE. Postup výpočtu v implementaci se nijak neliší od teoretického popisu, nemá tedy přiliš význam znovu výpočet popisovat. Za zmínku stojí, že každá buňka kódovací i dekódovací

tabulky je tvořena objektem, který jednotlivé řádky tabulky popisuje. Tyto objekty jsou reprezentovány třídami `FSEEncoderEntry` a `FSEDecoderEntry.`

Ihned po výpočtu dekódovací tabulky je tento objekt zapsán do hlavičky souboru společně s parametrem L. Hodnoty dekódovací tabulky jsou zapsány jako integer, jak jdou za sebou. Díky parametru L je jasné, kolik hodnot za sebou tvoří celou dekódovací tabulku. Tento postup by se dal v budoucnu upravit tak, aby tabulka zabírala méně místa. Nicméně velikost tabulky je zanedbatelná vůči velikosti dat.

Následuje samotné kódování dat z objektu `dataStore.` V každém kroku se vezme hodnota na vrcholu zásobníku a zakóduje se pomocí kódovací tabulky. Výsledkem je hodnota `state` a posun `shift.` Hodnota `shift` říká, kolik nejnižších bitů z hodnoty `state` je potřeba zapsat do souboru. Tento ořez se dělá pomocí bitového ANDu. Pro zápis slouží metoda `write` ve třídě `FSECompress`. Ta má jako parametry oříznutou hodnotu ze stavu, označenou jako `b` a počet bitů nutný pro zápis `nBits`.

`Write` funguje tak, že si drží pomocnou hodnotu `compressBuffer`, ve které jsou již zakódovaná data. Pokud je potřeba přidat zakódovaný symbol, `b` se bitově posune o velikost již zakomprimovaných dat a bitově se sečte s `compressBufferem.` Tak postupně vzniká jedno velké číslo tak, jak je popsáno v kapitole o principu ANS. Pokud již mají data v `compressBuffer` osm a více bitů, provede se oříznutí nejnižších osmi bitů. Ty jsou pak uschovány v dalším zásobníku, protože data je potřeba zapojovat, ale pořadí musí zůstat zachováno. Tedy poslední zakódovaný symbol musí být při dekódování první.

Poté, co jsou všechna data zakomprimována, se do souboru za dekódovací tabulku uloží celý poslední vypočtený stav a také číslo `bitCount`, které říká, jak velký je padding prvního načteného bytu. Tedy kolik prvních bitů v něm se má přeskočit při dekompresi.

Po zápise hlavičky se pak zapíšou ze zásobníku všechny byty. Výsledkem je zakomprimovaná sekvence dat.

4.7.5 Implementace dekomprese

Pro dekompresi FSE slouží třída `FSEDecode.` Ta podobně jako `FSEEncode` dědí z třídy `ByteToTripletConverte` a implementuje její metody tak, aby byla zaměnitelná. Hlavní metody jsou `decompress`, `read` a `createNew`.

Metoda `read` podobně jako při kompresi nejdříve zkontroluje, jestli objekt pro danou část tripletu existuje a pokud ne, vytvoří pro ni instanci `FSEDecompress.`

Při samotném vytvoření instance `FSEDecompress` dojde v konstruktoru k dekódování všech dat. Nejdříve se ze souboru načte parametr L pojmenovaný jako `sizeOfDecodeTable.` Pomocí něj se ze souboru načte celá dekódovací tabulka a uloží se do proměnné `decodeTable.` Počet hodnot, které je potřeba načíst, je přesně $3 * L$. Dále se načte finální stav a padding. Následují data, která je potřeba dekódovat.

Nejdříve se přeskočí počet bitů rovný paddingu. Teď již může začít samotná dekomprese dat. Ta je realizovaná v metodě `decodeAllData`. Jak její název napovídá, tato metoda najednou dekóduje všechna data a uloží je do fronty. Není tedy potřeba nijak měnit pořadí symbolů. Algoritmus z finálního stavu a dekódovací tabulky určí, jaký symbol dekóduje, kolik bitů je potřeba načíst ze vstupu a jaká je hodnota následujícího stavu. Tento postup se opakuje, dokud se nedekódují celá data. Podrobný popis lze nalézt v kapitole o dekódování pomocí metody FSE.

To vše se stane při vytvoření instance `FSEDecoderu`, tedy ještě v prvním volání metody `read`. Dále pak zavolá svou metodu `decompress`, která spustí dekompresi v třídě `FSEDecompress`. Ta postupně vrací již dekódovaná data tak, jak je uživatel potřebuje.

4.7.6 Spustitelný klient

Tato metoda nedisponuje spustitelným klientem, je jí však možné použít v jiných spustitelných klientech změnou konvertorů na `FSEEncoder` a `FSEDecoder`. Tato metoda nefunguje jako samostatný kompresní algoritmus. Slouží především ke kódování tripletů, které byly vytvořeny jiným algoritmem.

4.8 Realizace ZStandard

4.8.1 Struktura adresáře

- `ZStandard`
  - `ZStandard.java` ... třída volající metody komprese nebo dekomprese
  - `ZStandardClient.java` ..... klient spustitelný přes příkazovou řádku
  - `ZStandardProviderParameters.java` ..... třída obsahující všechny parametry
  - `ZStandardTest.java` ..... testovací třída
- `triplet`
  - `coder`
    - `ZStandardTripletCoder.java` . třída zpracující vstup při kompresi i dekompresi, dědí ze třídy `TripletCoder`
    - `LZ77TripletCoder.java` ..... třída zpracující vstup při kompresi i dekompresi, dědí ze třídy `TripletCoder`

Struktura 4.2: Adresářová struktura metody ZStandard v SCT

4.8.2 Parametry

Algoritmus ZStandard má celkem tři parametry pro frontendovou část metody: velikost prohlížecího okna, aktuálního okna a velikost dekódovacích tabulek pro FSE. Poslední zmiňovaný parametr se dá rozdělit a pro každou instanci FSE nastavit jinou hodnotu. V kapitole testovaní se zabývám dopadem rozdělení parametrů na efektivitu.

Všechny parametry jsou zadávány jako exponent e a jejich hodnota se spočítá jako

$$2^{e-1}.$$

Samotný exponent e říká, kolik bitů je potřeba pro zápis všech možných hodnot čísla 2^{e-1}. To je důležité především pro kodéry, u kterých je informace o rozsahu možných hodnot velmi důležitá. U prohlížecího okna se hodnota většinou pohybuje kolem 10-16 a u aktuálního okna 8-12. Vždy ale platí, že velikost prohlížecího okna je vždy větší nebo rovna velikosti aktuálního okna. Parametr L pak musí splňovat podmínku, že je větší než velikost množiny kódovaných symbolů, v opačném případě algoritmus selže a zahlásí chybu.

Dalším, spíše technickým parametrem, je velikost částí, na jaké algoritmus komprimovaný soubor rozdělí. Ten se obvykle pohybuje ve statisících až milionech bitů.

4.8.3 Hlavička souboru

Algoritmus LZ77, který je součástí ZStandard potřebuje pro dekódování uložit parametry prohlížecího a aktuálního okna. Tyto hodnoty jsou na začátku komprese zapsány jako části tripletu, přestože jimi nejsou. Je tomu tak proto, že knihovna SCT nemá zatím žádný obecný postup, jak hlavičku zapsat a tento přístup se mi jevil jako elegantní a jednoduchý. Takže před dekompresí souboru se první dvě hodnoty načtou jako parametr prohlížecího a aktuálního okna.

Tato hlavička je tedy součástí kódovaných dat. Komprimovaný soubor u ZStandard metody je vlastně rozdělen na tři části, z nichž každá představuje hodnoty jednoho ze tří segmentů tripletu. Každá část má tedy jako hlavičku vše potřebné pro dekódování dat – parametr L, dekódovací tabulku, finální stav a padding.

4.8.4 Implementace komprese

Kompresní metoda ZStandard má frontendovou část stejnou jako metoda LZ77, která byla mnou již implementována v minulosti. Bylo tedy možné její části využít pro realizaci ZStandard.

Ve třídě `ZStandard` je metoda `compress`, která zajišťuje průběh komprese. Jejím vstupním parametrem je instance třídy `ByteBuffer.` Ten obsahuje data souboru, která je potřeba zkomprimovat. Pomocí něj se inicializuje instance třídy `LZ77TripletCoder.` Ta se stará o kompresi a vytváření tripletů, které jsou následně posílány kodéru FSE. Jeho implementace je popsána v předchozí kapitole.

Metoda si nejprve inicializuje parametry ze třídy `LZ77ProviderParametr.` Vypočítá si velikosti prohlížecího a aktuálního okénka a nastaví identifikátory částí tripletu pomocí třídy `TripletFieldId.`

Následně vytvoří tři indexy. První ukazuje na začátek prohlížecího okna, druhý začátek aktuálního okna a třetí na konec aktuálního okna. Tyto indexy se poté starají o to, aby velikost oken zůstala tak velká, jak udávají omezení daná parametry. Na začátku jsou první dva indexy rovny nule. Třetí index odpovídá velikosti aktuálního okna. Postupnou iterací se indexy zvětšují. Jedná se tedy o hranice v datech, které ohraničují posuvné okénko tak, aby algoritmus nevybíral neočekávané hodnoty, například délky kopírovaných dat.

Algoritmus následně hledá první symbol z aktuálního okénka, pokud ho nalezne, podívá se, jak dlouhá sekvence znaků v aktuálním okénku odpovídá sekvenci znaků v prohlížecím okně. Poté si zapamatuje délku a index prvního znaku v prohlížecím okně. A pokračuje dál v hledání. Když najde další shodu, porovná délky podřetězců a vybere ten nejdelší. Poté, co projde celé prohlížecí okénko, pošle triplet (i, j, b) FSE kodéru, kde i je vzdálenost nalezeného podřetězce od začátku aktuálního okna, j je jeho délka a b následující byte.

Rozsah hodnot i je od 0 do velikosti prohlížecího okna. Pro jeho zapsání je tedy zapotřebí stejný počet bitů, jaký byl zadaný pomocí třídy parametrů. Podobně je tomu u j. Jeho hodnota muže být až velikost aktuálního okna. I pro toto číslo je zadán parametr. Poslední složkou je b. To je vždy byte, takže je potřeba 8 bitů na zápis. Není tedy potřeba žádný parametr.

Jak bylo zmíněno výše, u FSE algoritmu je pro každou složku možné volit jestě parametr L. Ta je standardně nastavená na 4096.

FSE kodér si triplet uloží a upraví frekvenční tabulku. O implementaci FSE algoritmu pojednává předešlá kapitola.

Poté, co je celý soubor zpracován pomocí LZ77, je vyslán signál prostřednictvím funkce `terminate.` Ta spustí kódování tripletů pomocí FSE a následně jsou data zapsána do souboru.

4.8.5 Implementace dekomprese

Podobně jako u komprese, ve třídě `ZStandard` je metoda `decompress`, která zajišťuje průběh dekomprese. Jedním jejím parametrem je instance třídy `TripletProcessor. TripletProcessor` je instancí třídy `FSEDecode`. Tato třída nejprve zrekonstruuje všechny triplety tak, aby byly připraveny pro dekompresi metodou ZStandard.

Ta využívá existující dekompresní algoritmus ve třídě `LZ77TripletCoder.` Z této třídy se zavolá metoda `decode.` Ta vezme triplet (i, j, b) a pokud je hodnota $i = 0$, pak zapíše do souboru byte b. Jinak nalezne podřetězec v již dekódovaných datech ve vzdálenosti i od konce dat s délkou j. Tento podřetězec zapíše na konec dat a přidá byte b. Takto celý algoritmus pokračuje, dokud jsou na vstupu triplety.

Nakonec se celý dekódovaný obsah zapíše do souboru a program se ukončí.

4.8.6 Spustitelný klient

Pro tuto metodu je vytvořen spustitelný klient `ZStandardClinet.` Klient metody ZStandard může být spuštěn s těmito přepínači:

- **-h** – Vypíše nápovědu.
- **-d** – Spustí dekompresi vstupního souboru. S tímto přepínačem není potřeba zadávat další parametry kromě kodéru, program bude ostatní ignorovat, protože si načte hlavičku, ve které jsou všechny potřebné parametry.
- **-sb** <e> – Tento přepínač říká, jak velké prohlížecí okénko bude použito (2^{e-1}).
- **-lb** <e> – Tento přepínač říká, jak velké aktuální okénko bude použito (2^{e-1}).

- **-p** <n> – Tento přepínač říká, po jak velkých částech bude probíhat komprese (v bytech).
- **-c** <c> – Tento přepínač říká, jaký kodér bude použit, $c = 1$ pro FSE a $c = 0$ pro Aritmetický kodér.

Poznámka: Pokud při kompresi některý z posledních dvou přepínačů chybí nebo je v některém chyba, metoda se zavolá s výchozími parametry, které jsou: `-sb 11 -lb 8 -p 1000000 -c 1`.

4.9 Realizace LZFSE

4.9.1 Struktura adresáře

- `LZFSE`
  - `LZFSE.java` ..... třída volající metody komprese a dekomprese
  - `LZFSEClient.java` ..... klient spustitelný přes příkazovou řádku
  - `LZFSEProviderParameters.java` . třída obsahující všechny parametry
  - `LZFSETest.java` ..... testovací třída
- `triplet`
  - `coder`
    - `LZFSETripletCoder.java` ..... třída zpracující vstup při kompresi i dekompresi, dědí ze třídy `TripletCoder`

Struktura 4.3: Adresářová struktura metody LZFSE v SCT

4.9.2 Parametry

Frontendová část této metody má celkem čtyři parametry: velikost prohlížecího okna m, aktuálního okna k, minimální délku podřetězce l a maximální počet přeskočených symbolů v jednom tripletu s.

Podobně jako u ZStandard platí, že m a l lze zapsat ve tvaru

$$2^n,$$

kde n je přirozené číslo. Pro l a s platí, že

$$s > l.$$

Hodnota parametru s je obvykle v nižších desítkách. Výchozí hodnota je 50. Je možné s ní experimentovat z hlediska efektivity a rychlosti. V kapitole testování uvádím, jaký má změna tohoto parametru dopad na efektivitu. Parametr l také testuji v následující kapitole.

Backendová část má také čtyři parametry. Metoda LZFSE má triplet obsahující čtyři složky. Pro každou je potřeba instance FSE a tedy i parametr velikosti dekódovací tabulky. Tyto parametry lze měnit dle přání uživatele.

4.9.3 Hlavička souboru

Hlavička souboru je totožná s hlavičkou ZStandard. Stejně ukládá maximální velikost prohlížecího a aktuálního okna. Rozdíl je pouze v tom, že zatímco ZStandard ukládala tři části dekomprimovaných dat, LZFSE ukládá čtyři. Takže místo tří hlaviček uvnitř souboru jsou čtyři. Ty obsahují stejná data, tedy parametr L, dekódovací tabulku, konečný stav a padding.

4.9.4 Implementace komprese

Rámec této metody je stejný jako u ZStandard, takže si ho dovolím přeskočit.

Hlavní logika implementace LZFSE je v `LZFSETripletCoder.` Tato třída představuje frontendovou část. Metoda načte data a parametry zadané uživatelem pomocí třídy `LZFSEProviderParametrs`. Vytvoří rámec pohyblivého okénka pomocí indexů do načtených dat. Postupuje podobně jako ZStandard, tedy, že najde nejdelší podřetězec ze začátku aktuálního okna v prohlížecím okně. Pokud je podřetězec kratší než zadaný parametr l, pak se tento podřetězec uloží do pomocného podstringu *str*. Pokud je takových podřetězců nalezeno po sobě více, připojují se za sebe.

Následně mohou nastat dva případy. Buď bude velikost *str* větší nebo rovna hranici zadané uživatelem s nebo bude nalezen dostatečně dlouhý podřetězec. V obou případech se zapíše triplet. Nicméně v prvním případě se *str* ořízne tak, aby jeho velikost byla přesně s. Indexy pohyblivého okna se upraví, tak že se celé posune zpět o velikost oříznuté části řetězce *str*. Poté se zapíše triplet $(s, str, 0, 0)$.

V druhém případě se zapíše triplet $(|str|, str, i, j)$. Kde i je vzdálenost od konce prohlížecího okna a j je délka kopírovaného podřetězce. Části tripletů jsou identifikovány pomocí třídy `TripletFieldId` jako u předchozí metody.

Triplety se posílají instanci `FSEEncoder,` která uchovává čtyři instance FSE kodéru `FSECompress,` pro každý segment tripletu. Poté, co doběhne frontendová část algoritmu, spustí se backendová část. Ta je podrobně popsána v implementaci FSE výše. Důležité je, že se všechny části tripletu kódují odděleně a následně se uloží do souboru za sebe.

4.9.5 Implementace dekomprese

Při dekompresi se nejdříve provede dekódování jednotlivých složek tripletů tak, jako u metody ZStandard. Inicializuje se instance třídy `FSEDecoder`, ve které se následně při prvním volání vytvoří čtyři instance třídy `FSEDecompression.` Ty ve svém konstruktoru postupně dekódují všechna data tripletů. Implementaci popisuji v kapitole o FSE.

Následně je pak podle potřeby volána metoda `read,` která má jako parametr instanci třídy `TripletFieldId.` Díky ní pozná, jaká čast tripletu je potřeba vrátit.

Samotný algoritmus je implementován tak, že načte první část tripletu m, která říká, kolik bytů má následující část tripletu, ta je totiž na rozdíl od ostatních částí různá. Algoritmus tedy zavolá m-krát metodu `read` pro další část tripletu označenou jako *str*, ta je kódovaná po bytech. Výsledek tohoto volání přidá na konec dekódovaných dat.

Dalším krokem je načtení vzdálenosti kopírovaného podřetězce od konce dekódovaných dat a délku podřetězce. Tyto dvě hodnoty jsou obsažené v tri-

pletu na třetím a čtvrtém místě. Po jejich získání se z dekódovaných dat zkopíruje podřetězec a připojí se na konec dat.

Takto algoritmus dekomprimuje celý soubor, následně vytvoří nový s dekódovanými daty a ukončí se.

4.9.6 Spustitelný klient

Pro tuto metodu je vytvořen spustitelný klient `LZFSEClinet.` Klient metody LZFSE může být spuštěn s těmito přepínači:

- **-h** – Vypíše nápovědu.
- **-d** – Spustí dekompresi vstupního souboru. S tímto přepínačem není potřeba zadávat další parametry kromě kodéru. Program bude ostatní ignorovat, protože si načte hlavičku, ve které jsou všechny potřebné parametry.
- **-sb** <e> – Velikost prohlížecího okénka bude (2^{e-1}).
- **-lb** <e> – Velikost aktuálního okénka bude (2^{e-1}).
- **-p** <n> – Tento přepínač říká, po jak velkých částech bude probíhat komprese (v bytech).
- **-c** <c> – Jako kodér bude použit, $c = 1$ pro FSE a $c = 0$ pro Aritmetický kodér.
- **-nl** <e> – Parametr maxima přeskočených symbolů bude mít hodnotu (2^{e-1}).
- **-ch** <n> – Parametr minima symbolů pro vytvoření tripletu bude mít hodnotu n.

Poznámka: Pokud při kompresi některý z posledních dvou přepínačů chybí nebo je v některém chyba, metoda se zavolá s výchozími parametry, které jsou: `-sb 11 -lb 8 -p 1000000 -c 1 -nl 5 -ch 4`.

Kapitola 5

Testování

V této kapitole se zaměřuji na efektivnost mé implementace algoritmů FSE, ZStandard a LZFSE. Pro určování efektivnosti kompresních algoritmů se obvykle používá hodnota kompresního poměru a časová náročnost. Kompresní poměr je podíl velikosti zkomprimovaného a původního souboru. Číslo tedy říká, na jak velkou část se podařilo soubor zmenšit. V případě, že je kompresní poměr větší než jedna, byla komprese neúspěšná a soubor se zvětšil. Časová náročnost je údaj, který říká, za kolik sekund algoritmus úspěšně doběhl.

Jako testovací soubory jsem zvolil korpusy Canterbury a Prague. Canterbury má velikost 2 760 KB a jedná se o 14 souborů anglického textu, náhodných symbolů, kódů v C a Pascalu, obrázků a podobně.

Prague je větší korpus Canterbury a má velikost 56 900 KB. Obsahuje směs 30 souborů, které jsou různorodé a obecně platí, že na Prague korpusu se dobrého kompresního poměru dosahuje velmi těžce.

Všechny naměřené hodnoty byly pořízeny pomocí počítače s následující konfigurací:

Procesor – Intel Core i5 9th Gen (frekvence jednoho jádra – 2,4 GHz)

RAM – 16 GB

Operační systém – Windows 10 64-bit (testování – příkazová řádka Git Bash)

5.1 Měření FSE

Metoda FSE má pouze jediný parametr a tím je velikost dekompresní tabulky L. Tento parametr je potřeba nastavit pro každou instanci FSE kodéru. Obvykle se používá společná hodnota 4096 nezávisle na datech, která jsou kódována. Nicméně v této práci analyzuji algoritmy LZFSE a ZStandard, které mohou mít v závislosti na nastavených parametrech různé rozsahy hodnot.

Pro některé by tedy hodnota 4096 nemusela stačit. Rozhodl jsme se tedy otestovat parametr L dvěma způsoby. V prvím případě jsem použil při kompresi pro všechny složky tripletu stejnou hodnotu L. Výsledky měření jsou v tabulce 5.1.

Hodnoty byly naměřeny na korpusu Canterbury s fixními parametry, z nichž nejdůležitější je velikost prohlížecího okna. Ta má totiž nejvíce možných hodnot a je tedy důležité, aby L bylo větší, než je velikost množiny všech jejích hodnot. Tato hodnota byla při tomto měření nastavena na 1024, proto nelze použít menší číslo pro parametr L než 2048. Naopak větší číslo pro L než 16384 nebylo potřeba testovat, protože z měření je jasně vidět, že by se výsledek komprese jen zhoršoval.

Problém ve fixně zvoleném parametru L je ten, že pro ostatní části tripletu s mnohem menším rozsahem hodnot je příliš velká dekódovací tabulka kontraproduktivní a kompresní poměr se začne zhoršovat, jak je vidět v tabulce 5.1.

Tabulka 5.1: Testování FSE na Canterbury s fixním parametrem L

Parametr L	**2048**	**4096**	**8192**	**16384**
Kompresní poměr	0,40	0,42	0,48	0,62
Čas komprese (s)	0,311	0,394	0,653	1,195
Čas dekomprese (s)	0,145	0,127	0,126	0,241

Rozhodl jsem se tedy otestovat i variantu, kdy se jednotlivé parametry L určují podle rozsahu hodnot segmentů tripletu. Naměřené hodnoty jsou vidět v tabulce 5.2 a v grafu na obrázku 5.1.

Parametr n v tabulce 5.2 říká, jakým koeficientem se přenásobil rozsah jednotlivých složek v tripletu, aby každá instance FSE metody získala svůj parametr L. Tedy například, pokud byl rozsah hodnot pro prohlížecí okénko k, pak se hodnota parametru L vypočte takto: $L = k2^n$.

Při porovnání obou způsobů je vidět, že druhý přístup má lepší výsledný kompresní poměr pro $n = 2$ a $n = 3$. Pro ostatní měření jsem tedy zvolil druhý postup s $n = 2$.

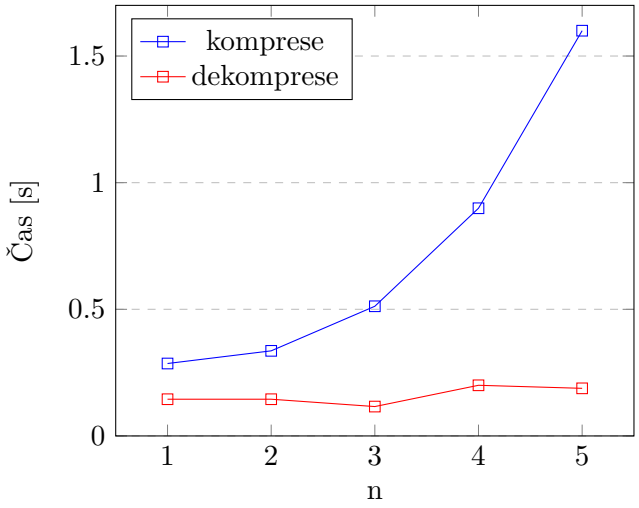
V následujícím grafu na obrázku 5.1 je vidět časová závislost komprese a dekomprese pomocí FSE. Je vidět, že časová složitost komprese roste s velikostí L. V tomto případě se nedá říci, že by časová složitost rostla exponenciálně s parametrem L. Parametr sám totiž není lineární a jeho hodnoty

jsou exponenciální. Časová složitost je tedy lineární s L, ale L nabývá pouze exponenciálních hodnot.

Z grafu je také vidět, že rychlost dekomprese dat téměř nezávisí na velikosti L respektive n. Složitost dekomprese je závislá především na velikosti dat a ta se při měření neměnila.

Tabulka 5.2: testování FSE na Canterbury s dynamickým parametrem L

Parametr n	**1**	**2**	**3**	**4**	**5**
Kompresní poměr	0,46	0,38	0,39	0,43	0,52
Čas komprese (s)	0,286	0,336	0,512	0,899	1,600
Čas dekomprese (s)	0,145	0,145	0,116	0,200	0,188



Obrázek 5.1: Graf časové závislosti komprese a dekomprese FSE na velikosti parametru L

V následujícím měření jsem se zaměřil na rychlost komprese, dekomprese a kompresní poměr. V porovnání s Aritmetickým kodérem je FSE rychlejší při dekódování souboru, naopak v porovnání s rychlostí komprese a kompresním poměrem je výkon implementované metody o něco horší.

Kodéry není možné spustit samostatně, proto jsem zvolil postup, kdy použiji naimplementovanou metodu LZ77 a pouze vyměním kodéry. Následně pak naměřím časy komprese, dekomprese a kompresní poměry u několika souborů z Prague korpusu. Aritmetický kodér je v SCT implementován adaptivně, to znemožnuje naměření čistě jen časové náročnosti kodéru. Nicméně pokud použiji stejnou frontendovou metodu s totožnými parametry a na stejném souboru, vstup pro kodéry bude identický a tedy bude možné jejich naměřené hodnoty

porovnat. Jako parametry jsem zvolil pro prohlížecí okénko 1024 a pro aktuální okénko 128.

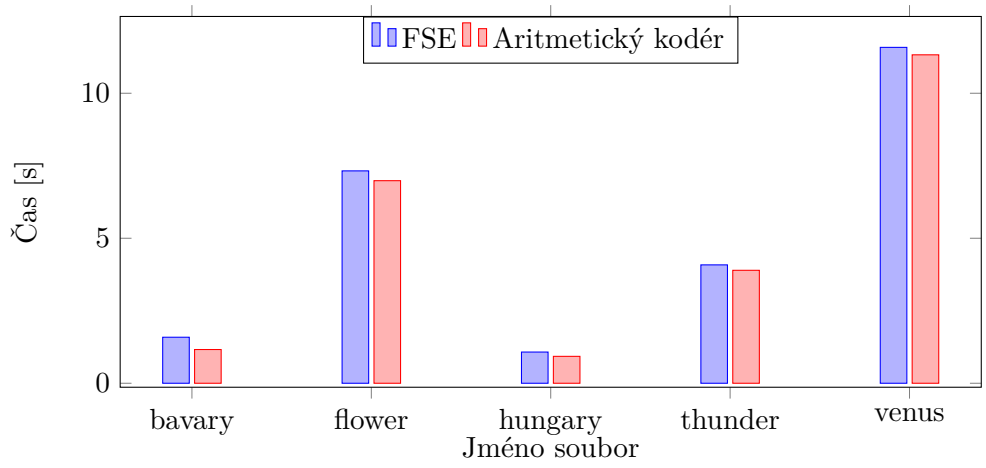
Všechny naměřené hodnoty jsou uvedené v tabulce 5.3.

Z grafu na obrázku 5.3 je vidět, že dekomprese u FSE je rychlejší než u Aritmetického kodéru. Takový výsledek byl očekávaný. Jak je zmíněno v popisu metody FSE, jde o metodu rychlejší především při dekódování. Co se kompresního poměru týká, je vždy o něco horší než u Aritmetického kodéru, ale rozdíl je poměrně malý.

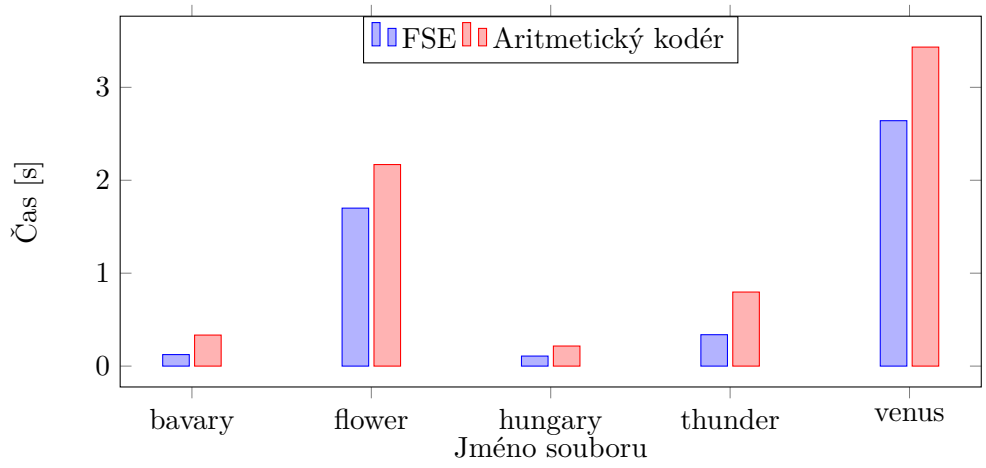
Graf měření času komprese 5.2 a kompresního poměru 5.4 ukazuje, že v těchto dvou aspektech je Aritmetický kodér efektivnější. Je ale potřeba zmínit, že implementace adaptivního aritmetického kodéru v knihovně SCT je modifikována pomocí datové struktury FenwickTree, která umožňuje efektivní prefixové sčítání, což výrazně snižuje čas komprese. Přesto, že z grafu 5.2 vyplývá, že je implementace FSE metody méně efektivní, rozdíl je poměrně malý, jen v řádech procent. Naopak u času dekomprese je FSE kodér o desítky procent rychlejší.

Tabulka 5.3: Testování FSE a AK na Canterbury

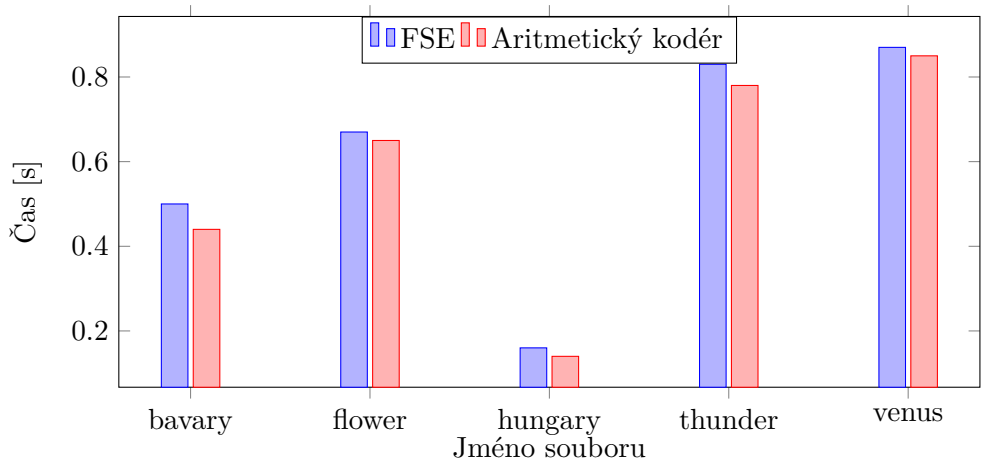
Soubor	**bavary**	**flower**	**hungary**	**thunder**	**venus**
Kom. FSE (s)	1,587	7,323	1,076	4,082	11,582
Kom. AK (s)	1,163	6,985	0,928	3,895	11,325
Dek. FSE (s)	0,124	1,696	0,108	0,338	2,641
Dek. AK (s)	0,344	2,169	0,216	0,797	3,433
Kom. p. FSE	0,50	0,67	0,16	0,83	0,78
Kom. p. AK	0,44	0,65	0,14	0,78	0,85



Obrázek 5.2: Porovní času komprese FSE a Adaptivního Aritmetického kodéru u LZ77



Obrázek 5.3: Porovní času dekomprese FSE a Adaptivního Aritmetického kodéru u LZ77



Obrázek 5.4: Porovní kompresního poměru FSE a Adaptivního Aritmetického kodéru u LZ77

5.2 Měření ZStandard a LZFSE

Tato kapitola obsahuje naměřené hodnoty při testování metody ZStandard a LZFSE. Měření se zaměřuje především na rychlost komprese, dekomprese a kompresní poměr. Dále pak je zde zkoumán vliv parametrů těchto metod na výslednou efektivitu implementovaných algoritmů. Obě dvě metody porovnávám ve stejné kapitole, protože mají mnoho společného, především pak dva parametry – velikost prohlížecího a aktuálního okna.

Dále mají společný backendový kodér FSE, jehož parametr je pro zjednodušení měření fixován tak, že pro každou část tripletu je nastaven na rozsah dané části zvětšený čtyřikrát. Tedy jde o postup popsaný v měření FSE v předchozí kapitole. Tento postup vykazoval lepší výsledky než stejný parametr pro všechny části.

5.2.1 Parametr prohlížecího okna

Jako první jsem se zaměřil na parametr ovlivňující velikost prohlížecího okna. V tabulce 5.4 jsou uvedeny všechny naměřené hodnoty pro parametr e. Ten udává exponent rozsahu všech hodnot prohlížecího okna. Samotná velikost prohlížecího okna je 2^{e-1}. V tabulce jsou uvedeny časy komprese, dekomprese a kompresní poměr metod LZFSE a ZStandard.

Kromě parametru prohlížecího okna jsou všechny ostatní parametry zafixovány. Aktuální okno je nastaveno na velikost 2^7, tedy exponent je nastaven na 8. LZFSE má pak ještě nastavený parametr, a to maximálního počtu přeskočených bytů na 2^5, což znamená, že exponent pro tento parametr je 6. Parametr pro minimální velikost podřetězce pro vytvoření tripletu je 4.

Pro lepší přehlednost jsem hodnoty zobrazil pomocí grafů na obrázcích 5.5, 5.6 a 5.7.

Tabulka 5.4: Data měření závislosti efektivnosti LZFSE a ZStandard na změně parametru velikosti prohlížecího okna

Parametr e	**10**	**11**	**12**	**13**	**14**
Komprese ZSTD (s)	1,140	1,709	2,774	7,965	17,852
Komp. poměr ZSTD	0,37	0,36	0,36	0,39	0,45
Dekomprese ZSTD (s)	0,188	0,209	0,137	0,130	0,170
Komprese LZFSE (s)	1,680	2,487	3,960	6,990	15,882
Komp. poměr LZFSE	0,39	0,38	0,38	0,41	0,47
Dekomprese LZFSE (s)	0,120	0,169	0,153	0,180	0,119

V grafu 5.5 je vidět, že časová složitost komprese pomocí metod LZFSE i ZStandard roste exponenciálně s parametrem e. Nicméně, jak bylo zmíněno v analýze LZ77, tento parametr sám roste exponenciálně. Proto není vhodné vybírat příliš velkou hodnotu e. Obvykle se volí hodnota v rozmezí 10-15.

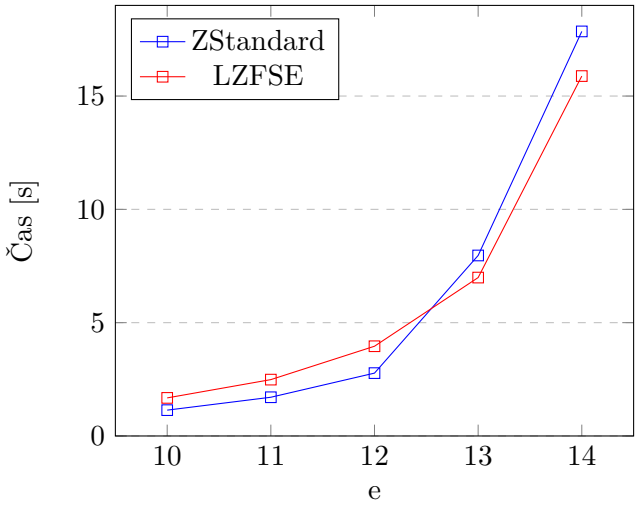
Z naměřených dat vyplývá, že pro nižší e je ZStandard rychlejší a naopak u LZFSE je pro vyšší e výsledný čas komprese o něco nižší. Nicméně také záleží na kompresním poměru a ten je pro oba algoritmy lepší při nižších e. Z hlediska časové složitosti je tedy vhodné volit parametr e, jako hodnotu v rozmezí 10-12.

Graf 5.6 zobrazuje zavislost kompresního poměru u LZFSE a ZStandard na parametru velikosti prohlížecího okna. Jak ukazují křivky v grafu, s rostoucí velikostí prohlížecího okna klesá efektivnost komprese a kompresní poměr se zhoršuje. Ideální volbou pro parametr e z hlediska kompresního poměru je tedy hodnota 11 a 12.

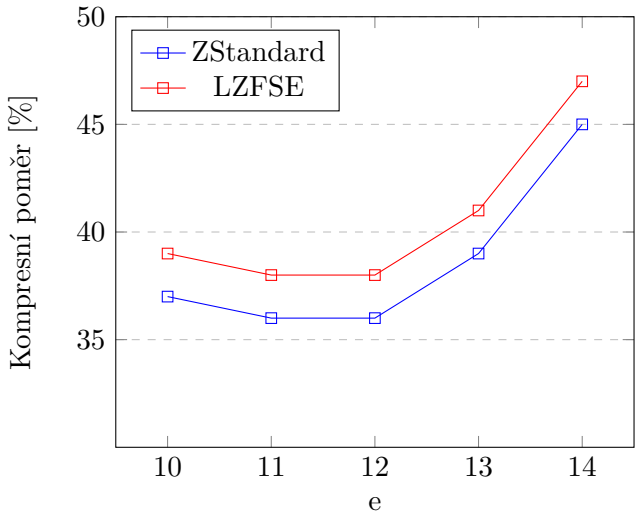
Poslední data, která jsem měřil, je rychlost dekomprese v závislosti na velikosti prohlížecího okna. Zobrazuje ji graf na obrázku 5.7. Podle předpokladu je rychlost dekomprese závislá především na počtu tripletů a ten je závislý na velikosti dat, která se neměnila.

Při výběru parametru tak, aby byla dekomprese co nejrychlejší, není potřeba se velikostí prohlížecího okna příliš zabývat. Naopak pokud jde o rychlost komprese a kompresní poměr, pak je ideální volbou hodnota 11 nebo 12.

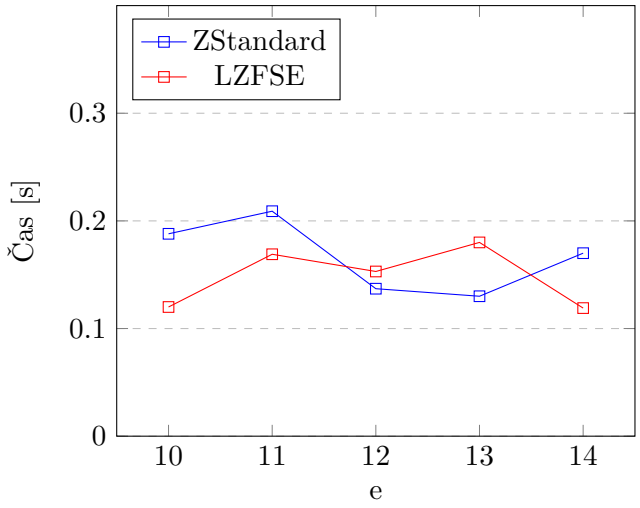
Pro další měření, kde je velikost prohlížecího okna fixovaná, jsem zvolil hodnotu 12, hodnota prohlížecího okna tedy bude 2^{11}.



Obrázek 5.5: Graf časové závislosti komprese LZFSE a ZStandard na velikosti prohlížecího okna



Obrázek 5.6: Graf závislosti kompresního poměru LZFSE a ZStandard na velikosti prohlížecího okna



Obrázek 5.7: Graf časové závislosti dekomprese LZFSE a ZStandard na velikosti prohlížecího okna

5.2.2 Parametr aktuálního okna

Dalším parametrem je velikost aktuálního okna. Ta udává, jak maximálně velký může být podřetězec kopírovaný pomocí tripletu. Jako exponent parametru prohlížecího okna jsem zvolil hodnotu 12. Tato hodnota vyplynula jako ideální z předchozí analýzy tohoto parametru. Pro LZFSE jsem pak dále zvolil

exponent parametru maximální délky přeskočených bytů jako 6 a minimální počet bytů pro vytvoření tripletu jsem zvolil hodnotu 4. V tabulce 5.5 jsou naměřená data metod LZFSE a ZStandard na testovacím korpusu Canterbury.

Tabulka 5.5: Měření závislosti efektivnosti LZFSE a ZStandard na změně parametru velikosti aktuálního okna.

Parametr e	**7**	**8**	**9**	**10**	**11**
Komprese ZSTD (s)	3,895	2,740	2,741	3,005	3,577
Komp. poměr ZSTD	0,371	0,370	0,371	0,372	0,372
Dekomprese ZSTD (s)	0,130	0,130	0,131	0,133	0,130
Komprese LZFSE (s)	4,635	3,869	3,786	3,754	3,870
Komp. poměr LZFSE	0,391	0,389	0,391	0,390	0,398
Dekomprese LZFSE (s)	0,115	0,118	0,160	0,160	0,140

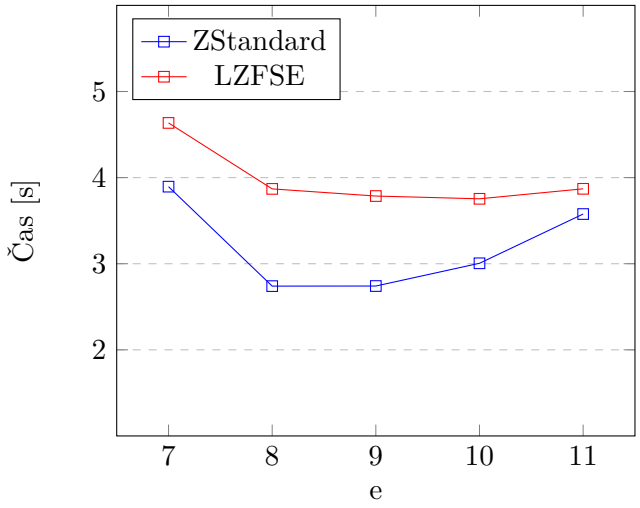
Na grafech 5.8, 5.9 a 5.10 jsou zobrazena data rychlosti komprese, dekomprese a kompresního poměru. Podle hodnot v grafu 5.8 se zdá, že parametr aktuálního okna nemá u LZFSE téměř žádný vliv, pokud je exponent větší než 7. U metody ZStandard se zdá, že ideálními hodnotami z hlediska rychlosti komprese je hodnota exponentu 8 nebo 9. U vyšších hodnot dochází k nárůstu časové složitosti.

Zajímavější je graf 5.9, na něm je vidět, že pro ZStandard je rychlost dekomprese téměř nezávislá na velikosti aktuálního okna. U metody LZFSE je naopak jasně vidět, že čím větší je aktuální okno, tím déle trvá dekomprese souboru. LZFSE je především konstruována tak, aby její dekomprese byla časově efektivní. Je tedy potřeba volit spíše menší velikost aktuálního okna, aby se rychlost dekomprese nezhoršila a metoda LZFSE tak nepřišla o hlavní vlastnost. Proto je ideální hodnota exponentu aktuálního okna 7 nebo 8. Z grafu 5.9 je vidět, že vyšší hodnota by rychlost dekomprese zvýšila na úroveň ZStandard nebo ještě výš.

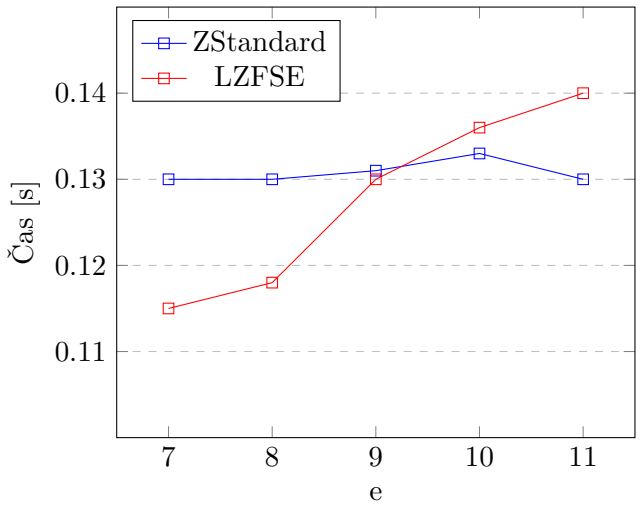
Pro metodu ZStandard je ideální hodnota exponentu 8 nebo 9.

Naopak z grafu 5.10 je zřejmé, že kompresní poměr není tímto parametrem nijak ovlivněn.

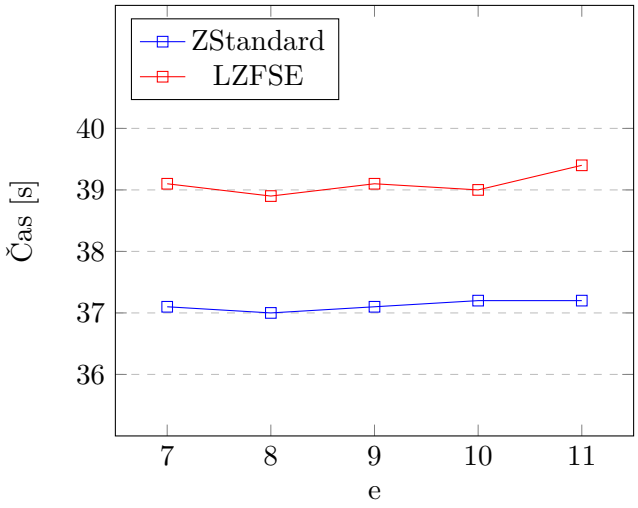
Pro další měření tedy za hodnotu exponentu aktuálního okna zvolím hodnotu 8 u LZFSE i ZStandard.



Obrázek 5.8: Graf časové závislosti komprese LZFSE a ZStandard na velikosti aktuálního okna



Obrázek 5.9: Graf časové závislosti dekomprese LZFSE a ZStandard na velikosti aktuálního okna



Obrázek 5.10: Graf závislosti kompresního poměru LZFSE a ZStandard na velikosti aktuálního okna

5.2.3 Parametry pouze u metody LZFSE

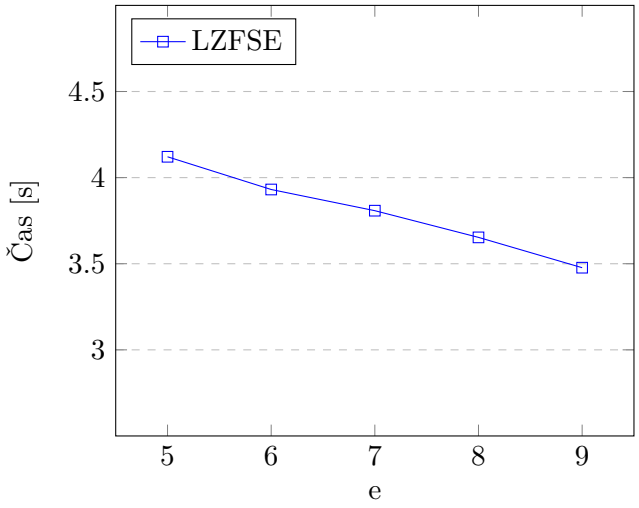
Prvním parametrem je velikost maximálního počtu přeskočených bytů. Ten se podobně jako předchozí parametry zadává pomocí exponentu. Pokud je tedy exponent $e = 5$, pak hodnota parametru bude 2^4. Jedná se o maximální počet přeskočených bytů, které mohou být zapsány v jednom tripletu. Tabulka 5.6 obsahuje naměřené hodnoty při analýze tohoto parametru. Testování proběhlo také na korpusu Canterbury. Ostatní parametry byly zafixovány na hodnotu 12 u exponentu prohlížecího okna a 8 na u hodnoty exponentu aktuálního okna.

Tabulka 5.6: Měření závislosti efektivnosti LZFSE na změně parametru maximálního počtu přeskočených bytů.

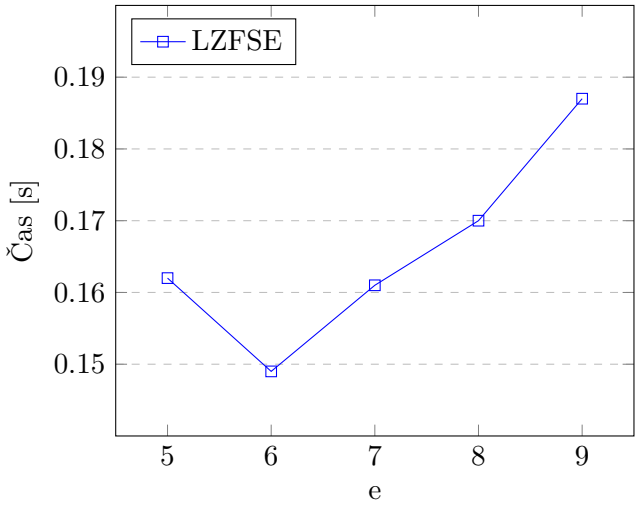
Parametr e	**5**	**6**	**7**	**8**	**9**
Komprese LZFSE (s)	4,121	3,931	3,808	3,653	3,477
Komp. poměr LZFSE	0,40	0,39	0,38	0,39	0,39
Dekomprese LZFSE (s)	0,162	0,160	0,161	0,163	0,187

Podle grafu 5.11 délka komprese klesá s rostoucím exponentem. Nicméně, jak už bylo řečeno, LZFSE se zaměřuje především na rychlost dekomprese a ta je nejlepší při hodnotě 6, jak je vidět v grafu 5.12. Kompresní poměr je sice podle grafu 5.13 lepší při hodnotě 7, ale i při hodnotě 6 je velmi dobrý a prioritou je rychlost dekomprese.

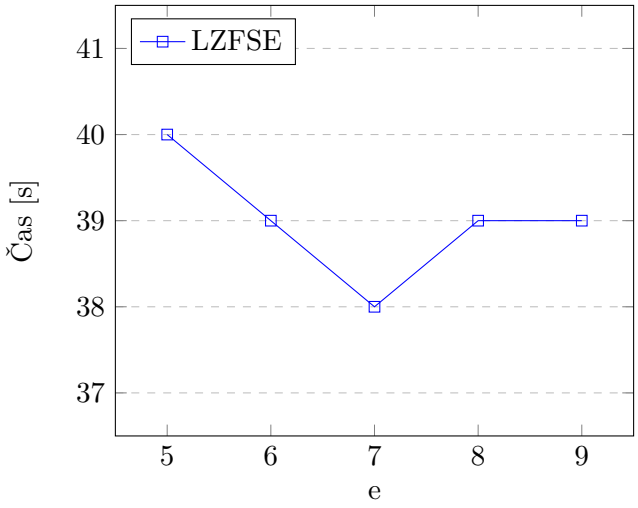
Z tohoto měření plyne, že je tento parametr velmi důležitý především pro minimalizaci délky dekomprese.



Obrázek 5.11: Graf časové závislosti komprese LZFSE na maximálním počtu přeskočených bytů



Obrázek 5.12: Graf časové závislosti dekomprese LZFSE na maximálním počtu přeskočených bytů



Obrázek 5.13: graf závislosti kompresního poměru LZFSE na maximálním počtu přeskočených bytů

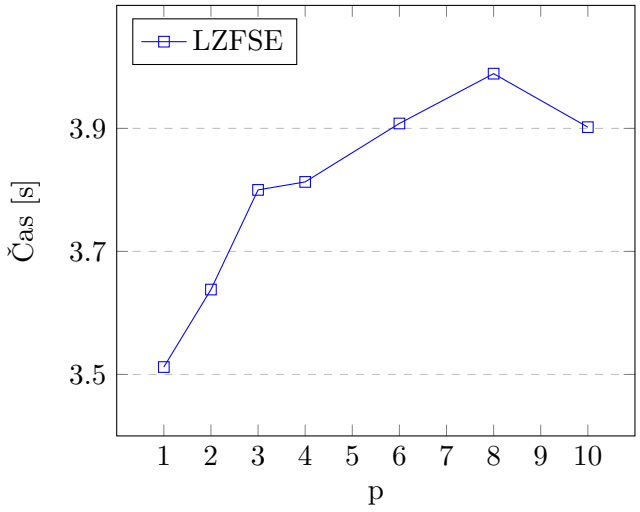
Dalším a posledním parametrem je parametr minimálního počtu zkopírovaných bytů pro vytvoření tripletu. Pro testování tohoto parametru jsem použil korpus Canterbury. Další parametry byly zafixované na hodnotu exponentu prohlížecího okna 12, hodnotu exponentu aktuálního okna na 8 a hodnotu exponentu maximálního počtu přeskočených bytů na 6. V tabulce 5.7 jsou všechna data získaná při tomto měření.

Z grafu 5.14 můžeme vyčíst, že s rostoucím parametrem p roste i časová složitost. Stejný trend je možné vidět i na grafu kompresního poměru 5.15. Lze tedy říci, že s rostoucím parametrem p se obecně zhoršuje komprese.

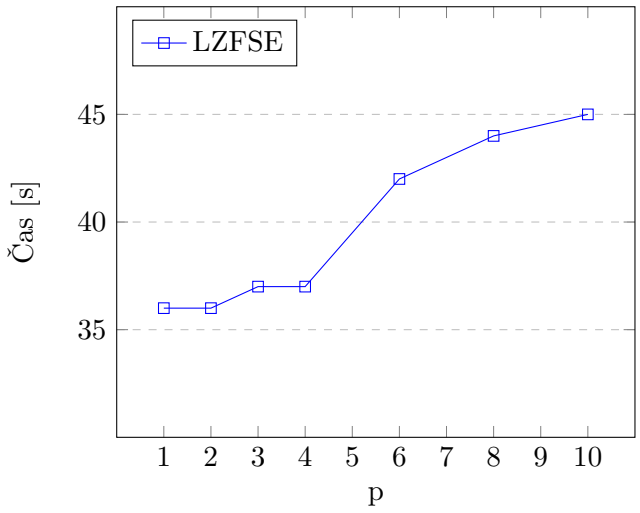
Nicméně důležitý je především čas dekomprese. Z grafu 5.16 je zřejmé, že ji tento parametr ovlivňuje. Je tedy potřeba zvolit ideální hodnotu tak, aby byla dekomprese co nejrychlejší, ale aby zároveň nedošlo k velkému zhoršení efektivity komprese. Ideální hodnotou je tedy $p = 4$, případně $p = 3$.

Tabulka 5.7: Měření závislosti efektivnosti LZFSE na změně parametru minimálního počtu zkopírovaných bytů pro vytvoření tripletu

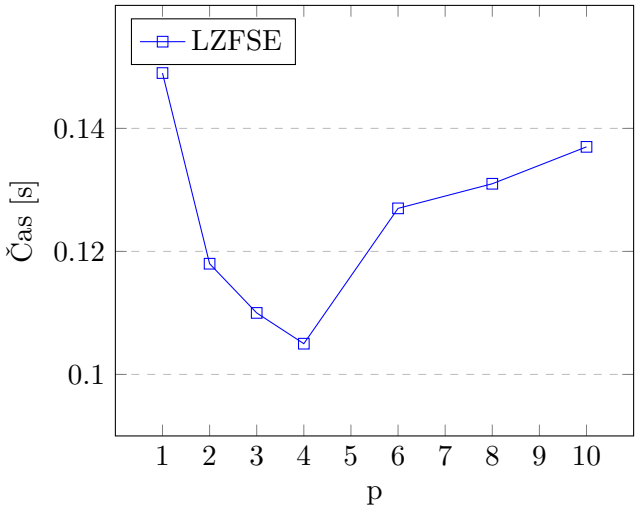
Parametr p	**1**	**2**	**3**	**4**	**6**	**8**	**10**
Kom. LZFSE (s)	3,512	3,638	3,800	3,813	3,908	3,989	3,902
Kom. p. LZFSE	0,36	0,36	0,37	0,37	0,42	0,44	0,45
Dek. LZFSE (s)	0,149	0,118	0,110	0,105	0,127	0,131	0,137



Obrázek 5.14: Graf závislosti času komprese LZFSE na změně parametru minimálního počtu zkopírovaných bytů pro vytvoření tripletu



Obrázek 5.15: Graf závislosti kompresního poměru LZFSE na změně parametru minimálního počtu zkopírovaných bytů pro vytvoření tripletu



Obrázek 5.16: Graf závislosti času dekomprese LZFSE na změně parametru minimálního počtu zkopírovaných bytů pro vytvoření tripletu

5.2.4 Porovnání LZFSE a ZStandard

V tabulce 5.8 jsou uvedeny hodnoty naměřené při testování rychlosti komprese a dekomprese na 6 náhodně vybraných testovacích souborů z korpusu Prague. ZStandard měla nastavený parametr exponentu velikosti prohlížecího okna na 12 a aktuálního okna na 8. LZFSE měla stejné parametry jako ZStandard. Parametr maximálního počtu bytů na 6 a minimální počet bytů na 4. Obě metody byly tedy nastaveny na hodnoty, jaké vyšly z měření jako ideální. Jejich FSE kodér měl hodnoty parametrů nastavené podle rozsahu hodnot v tripletu, jak je popsáno výše v kapitole o měření FSE.

Tabulka 5.8: Rychlost (s) komprese a dekomprese LZFSE a ZStandard na Prague korpusu

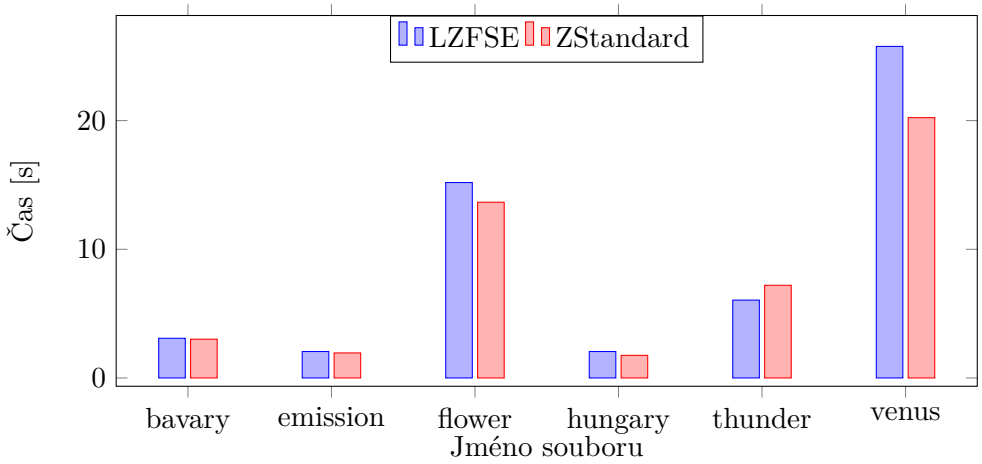
Soubor	**Dek. LZFSE**	**Dek. ZStd**	**Kom. LZFSE**	**Kom. ZStd**
bavary	0,125	0,124	3,084	3,015
emission	0,088	0,123	2,057	1,945
flower	1,544	1,867	15,188	13,659
hungary	0,111	0,139	2,055	1,756
thunder	0,501	0,489	6,051	7,202
venus	2,076	2,709	25,767	20,232

V grafech 5.17 a 5.18 jsou pro přehlednost zanesena data komprese a dekomprese z tabulky 5.8.

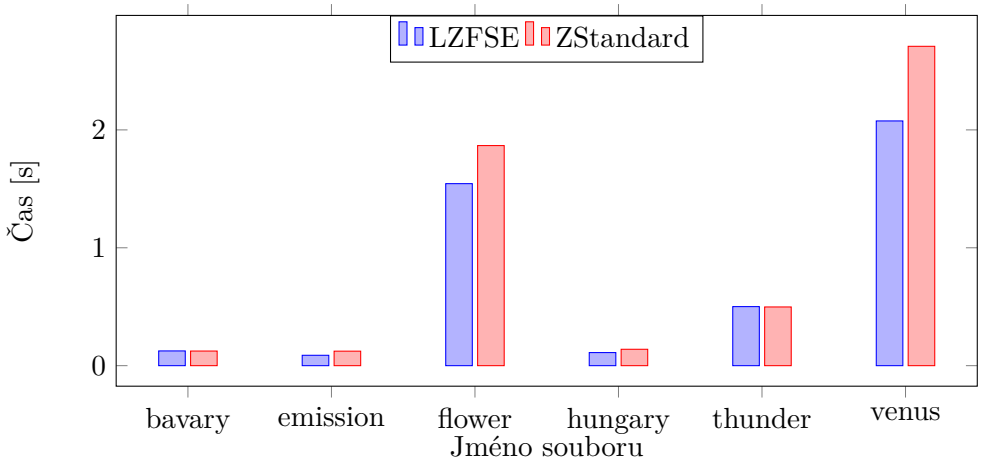
Měření potvrdilo teoretický předpoklad, že LZFSE je rychlejší při dekompresi než ZStandard. Ale pro rychlost komprese platí opak. U všech testova-

ných souborů byla rychlost komprese lepší u ZStandard. Což dokládá i graf 5.17. Naopak z grafu 5.18 je zřejmé, že všechna měření rychlosti dekomprese dopadla lépe pro metodu LZFSE.

To je přesně výsledek, který byl očekáván. Metoda byla přímo vytvořena tak, aby minimalizovala dobu dekomprese i za cenu horší efektivity komprese.



Obrázek 5.17: Porovnání rychlosti komprese LZFSE a ZStandard na Prague korpusu



Obrázek 5.18: Porovnání rychlosti dekomprese LZFSE a ZStandard na Prague korpusu

5.3 Závěr testování

Z naměřených dat u všech metod je zřejmé, že jejich implementace obstojí v porovnání s ostatními, již implementovanými metodami. Dále měření potvrdilo, že metoda FSE poskytuje především velmi rychlou dekompresi souborů.

U metod LZFSE a ZStandard analýza naměřených dat ukázala, že jejich efektivita je ovlivněna zvolenými parametry. Pro obě metody se mi podařilo nalézt optimální volbu parametrů. Uživatel ovšem může parametry zvolit libovolně, podle aspektu, který chce metoda minimalizovat, například pokud je potřeba u metody LZFSE minimalizovat čas komprese, je možné u parametru minimálního počtu bytů zvolit nižší hodnotu, než je mnou uváděná ideální hodnota 4.

Testování bylo provedeno na dvou korpusech Canterbury a Prague. Pro ještě přesnější měření by bylo vhodné výsledky ověřit na dalších korpusech, nicméně pro účely této práce považuji testování na dvou korpusech za dostatečné.

Závěr

Cílem této práce bylo seznámit se s kompresními metodami FSE, LZFSE a ZStandard a následně je implementovat do knihovny Small compression toolkit (SCT).

Kodér FSE měl být implementován jako modul tak, aby ho bylo možné jednoduše zaměnit za již implementovaný aritmetický kodér. Všechny výše zmíněné algoritmy se mi podařilo naimplementovat. Výsledkem je tedy knihovna SCT rozšířená o dva nové kompresní algoritmy a jeden kodér.

Analýza FSE ukázala, že se jedná o podobně rychlý a efektivní kompresní algoritmus jako aritmetický kodér. U LZFSE se mi podařilo testováním potvrdit, že jeho rychlost dekomprese je velmi rychlá a předčí i algoritmus ZStandard. Minimalizovat rychlost dekomprese byl hlavní účel, za kterým tento algoritmus vznikl. Dále se mi podařilo ověřit, že všechny implementované algoritmy jsou velmi efektivní a je možné je používat v praxi.

Metoda FSE byla porovnána s aritmetickým kodérem, který je již v SCT implementován a výsledkem je konstatování, že FSE má především rychlejší dekódování souborů. V ostatních parametrech je aritmetický kodér lepší, ale rozdíl není nijak výrazný.

U všech metod jsem provedl analýzu a testování jejich parametrů, především pak, jak se projeví jejich změna na kompresní poměr a časovou náročnost komprese a dekomprese.

Přes své nevýhody věřím, že FSE kodér, metoda LZFSE a metoda ZStandard najdou u uživatelů knihovny SCT své využití.

Literatura

[1] *Lexikon pojmů* [online]. [cit. 2021-04-24]. Dostupné z: http://stringology.org/DataCompression/lexikon.html

[2] *Shannonovo chápání informace* [online]. [cit. 2021-04-22]. Dostupné z: https://wikisofia.cz/wiki/Shannonovo_ch%C3%A1p%C3%A1n%C3%AD_informace

[3] *Small compression toolkit* [online]. [cit. 2021-04-24]. Dostupné z: https://gitlab.fit.cvut.cz/polacrad/sct

[4] *Functional and Nonfunctional Requirements: Specification and Types* [online]. [cit. 2021-04-23]. Dostupné z: https://www.altexsoft.com/blog/business/functional-and-non-functional-requirements-specification-and-types/

[5] *Huffmanovo kódování* [online]. [cit. 2021-04-22]. Dostupné z: http://voho.eu/wiki/kodovani-huffman/

[6] *Arithmetic coding - integer implementation* [online]. [cit. 2021-04-15]. Dostupné z: http://www.stringology.org/DataCompression/ak-int/index_en.html

[7] *Asymmetric numeral systems* [online]. [cit. 2021-04-22]. Dostupné z: https://en.wikipedia.org/wiki/Asymmetric_numeral_systems

[8] *Lossless Compression with Asymmetric Numeral Systems* [online]. [cit. 2021-04-22]. Dostupné z: https://bjlkeng.github.io/posts/lossless-compression-with-asymmetric-numeral-systems/

[9] *ZStandard repozitář od Facebooku* [online]. [cit. 2021-04-22]. Dostupné z: https://github.com/facebook/zstd

[10] *LZFSE repozitář od Apple Inc.* [online]. [cit. 2021-04-17]. Dostupné z: https://github.com/lzfse/lzfse

[11] *GNU Tar 1.31 přináší podporu zstd* [online]. [cit. 2021-04-16]. Dostupné z: https://www.root.cz/zpravicky/gnu-tar-1-31-prinasi-podporu-zstd/

[12] *Zstandard Compression and the application/zstd Media Type* [online]. [cit. 2021-04-22]. Dostupné z: https://tools.ietf.org/html/rfc8478

[13] *Algoritmus LZ77* [online]. [cit. 2021-04-22]. Dostupné z: http://voho.eu/wiki/algoritmus-lz77/

[14] *Holub J.: Data compression, dictionary metod I. (5. přednáška)* [online]. [cit. 2021-04-24]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-KOD/_media/lectures/05/mi-kod-05-dictionary.pdf

[15] *Zemek L.: Implementace kompresních metod LZ77, LZ78, LZW v jazyce Java* [online]. [cit. 2021-04-16]. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/76651/F8-BP-2018-Zemek-Ladislav-thesis.pdf?sequence=-1&isAllowed=y

Příloha **A**

Seznam použitých zkratek

AK Aritmetický kodér

ANS Asymmetric numeral systems (česky – Asymetrické číselné systémy)

FSE Finite state entropy

LZ Lempel-Ziv

SCT Small compression toolkit

ZSTD, ZStd Metoda ZStandard

Příloha **B**

Obsah přiloženého CD

Zdrojový kód knihovny SCT s implementací metod je k dispozici také na adrese: `https://gitlab.fit.cvut.cz/polacrad/sct/tree/lz`

```
/
├── readme.txt ..... stručný popis obsahu CD
├── src
│   ├── impl ..... zdrojové kódy knihovny s implementací metod
│   └── thesis ..... zdrojová forma práce ve formátu LaTeX
└── text ..... text práce
    └── DP_zemek_ladislav_2021.pdf ..... text práce ve formátu PDF
```