FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Implementace kompresních metod LZ77, LZ78, LZW v jazyce Java
Student:	Ladislav Zemek
Vedoucí:	Ing. Radomír Polách
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

1) Nastudujte kompresní metody LZ77, LZ78, LZW.
2) Navrhněte, analyzujte, implementujte a testujte dané algoritmy.
3) Implementaci proveďte jako součást knihovny dodané vedoucím práce.
4) Implementaci testujte na vhodných korpusech.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 22. ledna 2018

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

Bakalářská práce

Implementace kompresních metod LZ77, LZ78, LZW v jazyce Java

Ladislav Zemek

Vedoucí práce: Ing. Radomír Polách

13. května 2018

Poděkování

Rád bych poděkoval Ing. Radomírovi Poláchovi za pomoc a vstřícnost při vedení bakalařské práce a také Evě Filipovské za korekci textu.

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 13. května 2018 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 Ladislav Zemek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Zemek, Ladislav. *Implementace kompresních metod LZ77, LZ78, LZW v jazyce Java.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato bakalářská práce se zabývá návrhem, analýzou, implementací a testováním tří kompresních metod LZ77, LZ78 a LZW do knihovny SCT. Knihovna je koncipována tak, aby v budoucnu nabídla uživatelům velkou škálu kompresních metod a tak usnadnila výběr vhodné metody pro jejich projekt. Uvnitř této práce se zabývám popisem a vhodnou implementací daných algoritmů. U všech metod se mi podařilo při vhodně zvolených parametrech dosáhnout zmenšení testovaných korpusů vždy alespoň o 25 %. Přínosem této práce je přispění do knihovny, která usnadní vývojářům práci s kompresními metodami.

Klíčová slova Komprese dat, Dekomprese dat, Java, LZ77, LZ78, LZW, SCT

Abstract

This bachelor thesis deals with the design, analysis, implementation and testing of three compression methods LZ77, LZ78 and LZW in the SCT library. The library is designed to offer users a wide range of compression methods to facilitate the choice of the right method for their project. Inside this thesis I deal with suitable design and description of given algorithms. For all methods, I managed to reduce the tested corpuses with appropriately selected parameters at least by 25 %. The benefit of this work is contributing to a library that can make it easier for developers to work with compression metod.

Keywords Compression, Decompression, Java, LZ77, LZ78, LZW, SCT

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V době, kdy se začali počítače rozšiřovat do běžného života, vznikla potřeba ukládat čím dál větší objem dat. Z dnešního pohledu to nevypadá jako problém, jednoduše si uložíte několika gigabytový soubor na svůj pevný disk, který má kapacitu stovky gigabytů. Dříve ovšem měly pevné disky kapacitu jen desítky megabytů, což je téměř sto tisíckrát méně než dnes. Paměti tedy rozhodně nebylo nazbyt. Proto bylo potřeba data nějakým způsobem zmenšit a tím šetřit místo na disku. Samozřejmě za předpokladu, že nedojde ke ztrátě informací.

Právě za tímto účelem vznikla bezztrátová komprese dat, která zakóduje data pomocí důmyslných algoritmů. Nová data pak mají výrazně menší velikost než ta původní [1]. Jsou tedy vhodnější pro uložení na disk nebo k odeslání přes internet. Data je následně možné dekódovat pomocí dekomprese, což je inverzní operace ke kompresi. Výstupem dekomprese jsou původní data.

Jelikož v knihovně SCT nejsou naimplementovány zmíněné algoritmy, rozhodl jsem se je do knihovny doplnit.

Kapitola 1

Pojmy

V této kapitole bych rád vysvětlil a definoval některé pojmy.

- **Komprese/komprimace dat** – Zmenšování objemu dat.
- **Bezztrátová komprese dat** – Komprese dat, při které nedochází ke ztrátě 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`. Neznamená to tedy vždy trojici (LZ77). V závislosti na metodě může jít i o dvojici (LZ78) nebo jednici (LZW).
- **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 [2].
- **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 [2].
- **Symetrická kompresní metoda** – Komprimace i dekomprimace má stejnou složitost [2].
- **Asymetrická kompresní metoda** – Komprimace i dekomprimace má odlišnou složitost [2].
- **Entropie** – Míra neurčitosti systému [3]. 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.

Kapitola 2

Cíl práce

Cílem této bakalářské práce je seznámení se s kompresními algoritmy LZ77, LZ78, LZW a jejich následná implementace do knihovny SCT (Small Compression Toolkit). Součástí této práce je také jejich otestování z hlediska časové a paměťové náročnosti, případně porovnání efektivity s ostatními implementovanými metodami. Knihovna SCT by pak měla ušetřit práci programátorům, kteří díky ní nebudou muset vymýšlet vlastní implementaci těchto algoritmů.

2.1 Existující řešení

Jedná se o poměrně zastaralé metody, které vznikly v 80. letech 20. století a dnes je již v efektivitě překonávají jiné. Nicméně metodu LZ77 je možné použít například v programu CRUSH [4]. Z LZ77 bylo odvozeno mnoho dalších kompresních algoritmů, které jsou velmi často používány. LZ78 byla taktéž předlohou pro jiné algoritmy například LZW nebo LZMA [5]. Samotná se příliš nevyužívá, protože nedosahuje tak dobrého komprimačního poměru. Naopak metoda LZW bývala poměrně hojně využívá při komprimaci monochromatických obrázků a obecně je velmi efektivní tam, kde se opakuje velké množství dat. LZW se používala v dřívějších verzích formátu PDF, později byla nahrazená efektivnějším algoritmem [6]. Tyto algoritmy patří mezi základní kompresní metody a jelikož je hlavním cílem SCT poskytnout co největší spektrum kompresních algoritmů, rozhodl jsem se je do knihovny doimplementovat.

2.2 Analýza požadavků

V této části bych se rád věnoval požadavkům, které mi zadal vedoucí práce a jejich analýze. Každý softwarový projekt by měl začínat analýzou toho, co si zákazník přeje, aby program dělal nebo dodržoval. Požadavek musí být proveditelný, testovatelný a také musí byt definovaný dostatečné detailně pro účely návrhu [7]. Obvykle se požadavky dělí na funkční a nefunkční.

2.2.1 Funkční požadavky

Funkční požadavky objasňují, jaké má mít požadovaný software funkce, co má umět a jak se má chovat v daných situacích. Podle zadání a diskuse s vedoucím práce jsem určil tyto požadavky jako funkční:

- **Implementace LZ77, LZ78, LZW**
- **Dekomprese** – Práce zahrnuje i implementaci dekompresních algoritmů ke všem zmíněným metodám.
- **Parametry** – Algoritmům bude možné změnit jejich parametry, jako je například velikost bufferů, počet prvků ve slovníku nebo jak velké části souboru se budou komprimovat najednou. Parametry bude možné měnit při spouštění přes příkazovou řádku a nebo přímo v knihovně pomocí tříd, které poskytují parametry metodám.
- **Komprimace po částech** – Algoritmy budou schopné rozdělit soubor na menší části a ty pak komprimovat. Dekomprimace bude probíhat nad celým souborem, ne po částech, nicméně algoritmy budou schopny dekomprimovat i soubor, který byl předtím zakomprimován po částech. Velikost částí si určí uživatel pomocí parametru, případně využije výchozích nastavených hodnot.
- **Spustitelnost** – Program bude možné spustit se zvolenými parametry z příkazové řadky. Nepůjde tedy pouze o implementaci metod do knihovny, ale i o vytvoření jejich klienta, kterého bude možné prakticky využít.

2.2.2 Nefunkční požadavky

Jedná se o požadavky, které přímo neříkají, co má program umět. Jedná se převážně o omezující nebo zpřesňující podmínky, kterými se má návrh řídit. Tyto požadavky jsem určil jako nefunkční:

- **Knihovna SCT** – Implementace bude provedena do knihovny SCT [8]. Kód této open-source knihovny je dostupný na fakultním gitlabu. Adresu naleznete v kapitole popisující obsah CD na konci této práce.
- **Triplety** – Program bude pro ukládání dat do souboru využívat již naimplementované metody třídy `TripletProcesor` a pro definování jednotlivých složek v tripletu třídu `TripletFieldId`.
- **Velikost slovníků a bufferů** – Aby nedocházelo k nekontrolovatelnému růstu slovníku a tím i k zvyšování pamětové i časové náročnosti, budou mít algorytmy LZ78 a LZW parametr, který zhora omezí počet prvků ve slovníku. U metody LZ77 bude pomocí dvou parametrů omezeno prohlížecí i aktuální okénko.

- **Velikosti prvků v tripletu** – Toto omezení vychází z implementace třídy `TripletProcessor`, která dokáže zpracovávat pouze čísla, která je možné zapsat pomocí 28 bitů, tedy maximálně číslo 134 217 728.
- **Jazyk Java** – Algoritmy budou implementovány v jazyce Java, protože celá knihovna SCT je v tomto jazyce napsána.
- **Řetězení** – Implementace metody bude dodržovat myšlenku řetězení, tedy že různé kompresní metody bude možné za sebe zapojovat a docílit tak několikanásobné komprese.

Kapitola 3

Analýza algoritmů

V této kapitole pojednávám přímo o jednotlivých metodách, na jakém principu fungují a v čem se liší. Také zde určím jejich asymptotickou složitost komprese i dekomprese a jejich výhody a nevýhody.

3.1 LZ77

3.1.1 Popis

Tento kompresní algoritmus vynalezli v roce 1977 pánové Abraham Lempel a Jakob Ziv. Jedná se o slovníkovou, jednoprůchodovou, adaptivní a bezztrátovou metodu. Je založena na principu klouzavého okénka (floating window), které si lze představit jako vyříznutou část dat. Při běhu programu se okénko posouvá v datech pouze jedním směrem a to od začátku do konce. Je rozděleno na dvě části – prohlížecí okénko (search buffer) a aktuální okénko (look-ahead buffer). Jejich velikost je velmi zásadní pro efektivitu celého algoritmu. V zásadě se dá říct, že velikost prohlížecího okénka musí být vždy větší nebo rovna velikosti aktuálního okénka, nicméně v praxi je prohlížecí okénko obvykle tisíci násobně větší než aktuální. [9]

3.1.2 Komprese

Hlavní myšlenka tohoto algoritmu spočívá v nalezení co nejdelšího prefixu z nezakódovaných dat, který je zároveň obsažený i v prohlížecím okénku. Tato metoda umožňuje menší vylepšení efektivity, pokud se při vyhledávání neomezím pouze na prohlížecí okénko, ale jen na podmínku, že prefix v něm musí začínat. V některých případech je možné nalézt nejdelší prefix, který přesahuje do aktuálního okénka. Je tedy možné zakódovat větší část dat. Slovník této metody je tvořen všemi podřetězci, které začínají v prohlížecím okénku.

Zde je jednoduchý pseudokód, který popisuje tuto metodu:

```
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;
        posun 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 1: Pseudokód komprese metody LZ77 [11].

3.1.3 Složitost komprese

Dle mého názoru je složitost tohoto algoritmu $O(nm)$, kde n je délka vstupních dat a m je velikost prohlížecího okénka. Na první pohled není uplně zřejmé, jak jsem k tomuto závěru došel. Rozhodl jsem se tedy své tvrzení potvrdit pomocí matematického důkazu.

Mějme asymptotickou složitost znázorněnou sumou, která vyjadřuje nejvyšší možný počet operací potřebných pro komprimaci souboru:

$$\sum_{i=1}^{|P|} mp_i \,,$$

kde P je množina všech prefixů nalezených v datech, m je velikost prohlížecího okénka a p_i velikost i-tého prefixu z P.

Protože součet sumy nezáleží na m, mohu ho vytknout před sumu:

$$m \sum_{i=1}^{|P|} p_i \,.$$

Součet sumy je vždy rovný velikosti dat, protože procházím celá vstupní data a rozděluji je na podřetězce, které najdu v prohlížecím okénku. Nezáleží na délce jednotlivých podřetězců, jejich součet velikostí bude vždy roven velikosti vstupních dat, tedy n. Proto:

$$\sum_{i=1}^{|P|} mp_i = mn \,,$$

z čehož plyne, že složitost komprese je $O(nm)$.

Je třeba říci, že na vyhledání prefixu nebude vždy potřeba stejný počet operací, proto jsem se rozhodl použít tzv. omikron O, který složitost omezuje shora.

3.1.4 Dekomprese

Algoritmus je poměrně jednoduchý. Nejdříve načte ze vstupu triplet, který v sobě obsahuje informaci o tom, odkud a jak dlouhou část prohlížecího okénka je potřeba zapsat na výstup. Poslední část tripletu je jeden byte, který je taktéž potřeba zapsat. Následně zapsaná data rovněž přidáme na konec prohlížecího okénka. Zde je jednoduchý pseudokód:

```
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;
        pridej 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 2: Pseudokód dekomprese metody LZ77.[11]

3.1.5 Složitost dekomprese

Oproti kompresi je složitost dekomprese vcelku trivialní. Řekněme, že t je počet tripletů, pro každý triplet bude potřeba p_i operací, při čtení podřetězce z prohlížecího okénka. To znamená, že celkový počet operací bude roven sumě všech délek podřetězců. Tato suma se však rovná velikosti původního nekomprimovaného souboru, tedy n, čímž dostáváme:

$$\sum_{i=1}^{t} p_i = n\,,$$

z čehož plyne, že složitost dekomprese je $\Theta(n)$.

Složitosti komprese i dekomprese jsou rozdílné, jedná se tedy o asymetrickou metodu.

3.1.6 Výhody

- **Paměť** – Díky posuvnému okénku se v jednu chvíli zpracovává poměrně malá část dat, proto neroste paměťová náročnost s velikostí vstupu.

- **Rychlost dekomprese** – Při vhodně zvolené implementaci je algoritmus dekomprese lineárně závislý pouze na velikosti dat. Jeho složitost je tedy $\Theta(n)$, kde n je velikost původních dat.

3.1.7 Nevýhody

- **Náhodná data** – Pokud mají data příliš velkou míru entropie, algoritmus dosahuje velmi špatných výsledků, často i komprimovaný soubor zvětší. To se týká například některých formátů obrázku (jpeg).

- **Rychlost komprese** – Složitost $O(nm)$ je oproti dekompresi m krát větší. To znamená, že algoritmus bude pomalejší, protože m je obvykle velké číslo v řádech tisíců až desetitisíců.

3.2 LZ78

3.2.1 Popis

Metoda byla vynalezena stejnými autory jako LZ77 v roce 1978. Z počátku se jednalo o velmi oblíbenou komprimační metodu až do doby, než byly některé její části patentovány. Jedná se o paměťově velmi náročnou metodu. Je tomu tak hlavně kvůli obvykle velkému slovníku. Metoda totiž v původní verzi přidává do slovníku položku vždy, když zapisuje triplet, což se děje velmi často. Existuje mnoho způsobů, jak tento problém vyřešit. Například v případě vyčerpání paměti tzv. zamrazit slovník nebo smazat a vytvořit nový. Nejznámější modifikací je LZW, kterou se také zabývám v této práci. [10]

3.2.2 Komprese

Prvním krokem tohoto algoritmu je inicializace slovníku. Tento krok je velmi důležitý pro správný průběh dekomprese. Do slovníku se přidá slovo délky jedna za každé písmeno abecedy souboru. Abeceda je množina všech znaků, které se mohou v souboru objevit. Obvykle se používá množina všech bytů (0 - 255), pokud komprese kóduje po bytech. Dále nastaví řetězec *str* jako prázdný.

Následující krok se opakuje pro každý byte ze vstupu. Pokud spojení řetězce *str* a načtený byte *b* nejsou ve slovníku, na výstup se přidá triplet obsahující index *str* ve slovníku a *b*. V opačném případě se na konec řetězce *str* přidá byte *b*. Výstupem této metody je posloupnost dvojic – číslo, byte.

Zde je jednoduchý pseudokód, popisující kompresi:

```
inicializace slovníku;
str := prázdný řetězec;
while na vstupu jsou data do
    ze vstupu načti byte b;
    if str + b není ve slovníku then
        zapiš triplet (i, b), kde i je index str ve slovníku;
        přidej na konec slovníku slovo str + b ;
        str := prázdný řetězec;
    else
        str = str + b;
    end
end
```

Algoritmus 3: Pseudokód komprese metody LZ78 [11].

3.2.3 Složitost komprese

Tento algoritmus velmi ovlivňuje zvolená implementace. Pokud se slovník reprezentuje pomocí datové struktury strom, složitost je $\Theta(n)$, kde n je velikost vstupních dat v bytech. Protože dokáže v konstantní složitosti $\Theta(1)$ říci, jestli slovník již obsahuje daný řetězec či nikoliv. Pro každý vstupní symbol vykoná $\Theta(1)$ operací. Pro úplnost doplním jednoduchou rovnici, která ukazuje, že v asymptotické složitosti se konstantní složitost neprojeví:

$$\Theta(1 * n) = \Theta(n)\,.$$

3.2.4 Dekomprese

Jak je vidět na pseudokódu níže, algoritmus je velmi jednoduchý. Prvním a zásadním krokem je inicializace slovníku přesně stejnou abecedou jako v kompresi. Následně se načte triplet ze vstupu, podle něj se najde slovo ve slovníku a zapíše do souboru společně s následujícím bytem. Tento zapsaný řetězec je přidán do slovníku na konec (přiřadí se mu nejvyšší index).

```
inicializace slovníku;
while na vstupu jsou data do
    ze vstupu načti triplet (i, b), kde i je index do slovníku a b je
     následující byte;
    str := slovník[i] + b;
    zapiš str;
    přidej na konec slovníku slovo str ;
end
```

Algoritmus 4: Pseudokód dekomprese metody LZ78 [11].

3.2.5 Složitost dekomprese

Jak bylo řečeno v popisu, jedná se o symetrickou metodu. Složitost je tedy stejná jako u komprese a tedy $\Theta(n)$, kde n je velikost původních dat. Podobně jako u metody LZ77, máme sice menší vstup, nicméně je potřeba přičíst i režii slovníku na zrekonstruovaná slova. Jedná se o strom, to znamená, že metoda pro rekonstrukci slova udělá počet operací rovný délce slova. Pokud tedy všechny délky posčítáme, dostaneme n.

3.2.6 Výhody

- **Rychlost komprese i dekomprese** – Při vhodně zvolené implementaci je algoritmus komprese i dekomprese lineárně závislý pouze na velikosti dat. Jejich složitost je $\Theta(n)$, kde n je velikost původních dat.

- **Implementace** – Metoda je poměrně jednoduchá na implementaci, především pak dekomprese.

- **Velikost tripletů** – Do souboru se ukládají pouze dvojice – číslo, byte.

3.2.7 Nevýhody

- **Paměť** – Paměťová náročnost roste s velikostí vstupu. Proto se při implementaci musí počítat s tím, že metoda vyčerpá přidělenou pamět a podle toho se zachová.

- **Náhodná data** – Podobně jako u metody LZ77, při velké entropii dat dochází k horší kompresi, někdy muže dojít až k zvětšení souboru.

3.3 LZW

Tuto slovníkovou, symetrickou, adaptivní a jednoprůchodovou metodu vytvořili Abraham Lempel, Jacob Ziv a Terry Welch v roce 1984. Jedná se o modifikaci metody LZ78. Hlavním rozdílem je, že algoritmu pro dekomprimaci dat stačí pouze indexy ve slovníku. Již tedy nepotřebuje následující byte jako jeho předchůdce. To má ale také za následek rychlejší růst počtu frází ve slovníku. Slovník tedy typicky bývá větší než u LZ78 a spotřebovává rychleji paměť. Je tedy potřeba s tím počítat při implementaci a tento problém vhodně vyřešit. Obecně platí, že čím větší slovník, tím lepší kompresní poměr. Algoritmus je navržen tak, aby byl rychlý, nicméně ne vždy je optimální, protože neprovádí žádnou analýzu dat během komprese. [13]

Metoda se stala velmi oblíbenou kolem roku 1987, protože poskytovala nejlepší kompresní poměr. Byla využita například ve formátu GIF, později ve formátu PDF nebo v unixovém nástroji *compress*. Stala se první široce využívanou kompresní metodou na počítačích. [13]

3.3.1 Komprese

První krok algoritmu je inicializace, která probíhá totožně jako v metodě LZ78. Následně se načte byte b ze vstupu. Pokud je řetězec $str + b$ ve slovníku, nastaví se jako str. Pokud není, do souboru se zapíše triplet (i), kde i je index fráze str. Do slovníku se pak přidá fráze $str + b$ a jako nové str se nastaví b. To je hlavní změna oproti metodě LZ78, která umožnuje právě vynechání nasledujícího bytu v tripletu. Tento postup se opakuje, dokud jsou na vstupu data. Zde je pseudokód komprese LZW:

```
inicializace slovníku;
str := prázdný řetězec;
while na vstupu jsou data do
    ze vstupu načti byte b;
    if slovo str + b není ve slovníku then
        zapiš triplet (i), kde i je index str ve slovníku;
        přidej na konec slovníku slovo str + b ;
        str := b ;
    else
        str = str + b;
    end
end
```

Algoritmus 5: Pseudokód komprese metody LZW [12].

3.3.2 Složitost komprese

Metoda obsahuje pouze drobné změny oproti LZ78, které nemají vliv na asymptotickou složitost. Je totožná se složitostí komprese LZ78, tedy $\Theta(n)$. Proto si dovolím vynechat její odvození.

3.3.3 Dekomprese

Princip dekomprese není na první pohle pochopitelný, protože nepřidávám do slovníku slovo tvořené právě zapisovaným řetězcem, ale tím, který byl zapsaný v minulém běhu a pouze prvním znakem zapisovaného řetězce. Tato skutečnost je dána tím, že komprimace přidává do slovníku slovo délky n, končící právě načteným bytem, ale do souboru zapisuje index předchozího slova délky $n - 1$, tedy bez načteného bytu. Dekomprimační algoritmus pak vždy ví, že do slovníku je potřeba přidat slovo z minulého běhu zakončené prvním znakem z fráze nalezené ve slovníku.

Specialní případ nastane, pokud algoritmus ze vstupu načte triplet s indexem, který není ve slovníku. Toto číslo však může být pouze o jedna větší než nejvyšší obsazený index, jinak se jedná o chybu kompresního algoritmu. V tomto případě do slovníku vloží slovo, které bylo vloženo v předchozím kroku zakončené jeho prvním znakem.

K tomuto případu dochází, pokud bylo pro komprimaci použito slovo na posledním indexu (naposledy přidaná fráze do slovníku).

Zde je pseudokód, popisující tento algoritmus:

```
inicializace slovníku;
str := prázdný řetězec;
while na vstupu jsou data do
    ze vstupu načti triplet (i);
    if slovník obsahuje slovo s indexem i then
        fráze := slovník[i];
        zapiš fráze;
        přidej na konec slovníku slovo str + první znak fráze ;
    else
        fráze = str + první znak ze str;
        zapiš str;
        přidej na konec slovníku slovo fráze;
    end
    str := fráze
end
```

Algoritmus 6: Pseudokód komprese metody LZW [12].

3.3.4 Složitost dekomprese

V závislosti na implementaci se může složitost tohoto algoritmu lišit. Nicméně pokud strukturu slovníku reprezentuje strom, je možné dosáhnout složitosti $\Theta(n)$, kde n je velikost původních dat před kompresí. Metoda LZW je navíc symetrická, takže složitost je stejná jako u komprese. Důvod, proč jsem došel k této složitosti, je analogický s určením složitosti dekomprese u LZ78. Nejvíce kroků bude potřeba udělat tam, kde se ze slovníku rekonstruuje řetězec, což odpovídá celkové velikosti dat. Ostatní operace mají zanedbatelnou režii a při určování asymptotické složitosti se zanedbávají.

3.3.5 Výhody

- **Rychlost komprese i dekomprese** – Stejně jako u LZ78 je hlavní výhodou rychlost. Při správné implementaci je složitost obou $\Theta(n)$.
- **Velikost tripletů** – Do souboru se ukládají pouze čísla. Na jeden triplet je tedy potřeba uložit nejméně dat z metod, kterými se zabývá tato práce.
- **Kompresní poměr** – Na běžném anglickém textu dosahuje metoda velmi dobrých výsledků, typicky zmenší soubor zhruba o 50 %.

3.3.6 Nevýhody

- **Paměť** – Paměťová náročnost roste s velikostí vstupu. Roste dokonce rychleji než u LZ78. Proto se při implementaci musí počítat s tím, že metoda vyčerpá přidělenou pamět.
- **Náhodná data** – Podobně jako u předchozích metod, dochází při velké entropii dat k horší kompresi. Může dojít až k zvětšení souboru.

3.4 Závěr analýzy

Hlavním přínosem těchto metod je samotný princip, na jakém fungují. Především pak LZ77, z které vznikla řada algoritmů, které se dnes využívají v praxi. Největší nevýhodou těchto algoritmů je špatný kompresní poměr na datech, ve kterých se neopakují delší řetězce znaků. Další nevýhodou LZ78 a LZW je velká náročnost na paměť. Jejich kompresní poměr je obvykle závislý na velikosti slovníku. Nicméně i přes tyto nevýhody se LZW v některých programech používá dodnes, protože je velmi rychlá a vhodná na klasický text.

Kapitola 4

Implementace

V této kapitole popisuji realizaci metod přímo v knihovně SCT. Zásadní věcí v implementaci těchto metod je reprezentace slovníků, ať už u komprese nebo dekomprese. Efektivní přidávání a hledání ve slovníku je zásadní pro efektivitu těchto metod. Vysvětluji, jaké parametry bude moci nastavit uživatel a jak ovlivní efektivitu nebo paměťovou náročnost, dále jakým způsobem metody načítají data ze vstupu a jak zapisují triplety do souboru.

4.1 Řetězení

V knihovně se používá princip řetězení (chaining). To znamená, že některé metody se dají zapojovat za sebe. Zde je ukázka kódu:

```
ChainBuilder.create(io::openParse)
.chain(lz77::compress)
.chain(t2b)
.end(bytes -> io.saveObject(bytes, f2.toPath()))
.accept(f1.toPath());
```

Výše uvedený kód ukazuje použití implementovaných metod. Důležitý je princip, podle kterého je naimplementovaná metoda `compress`. Hlavní podmínkou pro chainování jsou dva parametry metody. První je vstupní parametr a druhý výstupní. Respektive druhým parametrem je objekt consumer, který má za parametr datový typ výstupu. Ukázka kódu:

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

Zde je vstupní parametr typu `ByteBuffer`, to znamená, že předchozí metoda v řetězci musela mít druhý parametr typu `Consumer<ByteBuffer>`. Vystupním parametrem je `TripletSuplier`, následující metoda tedy musí mít první parametr `TripletSuplier`. V tomto případě je to objekt `t2b`, který je

instancí třídy `TripletToByteConverter`, která implementuje rozhraní `Chainable<TripletSupplier, List<byte[]»`.

Další možností je implementovat celý objekt tak, aby mohl být řetězený, pomocí implementace rozhraní `Chainable<x, y>`, kde x je datový typ vstupního parametru a y datový typ pro výstup. Zde je hlavička třídy, kterou je možné řetězit:

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

Uvnitř metody se pak volá třídní metoda consumera `accept`, která posílá data do další metody v řetězci. Objekt musí mít vždy implementovanou metodu `setConsumer`, která nastaví consumera. Ten funguje stejně jako u metody.

4.2 Načítání a zápis do souboru

Načítání ze souboru se u komprese a dekomprese liší. Zatím, co v prvním případě načítám ze souboru byty, u dekomprese se jedná o načtení celého objektu ze souboru. Je tomu tak proto, že po kompresi se ukládá celé pole bytů jako objekt, aby bylo možné správně načítat uložené triplety. Dekomprese ukládá data do souboru po bytech, aby byl soubor stejný jako před komprimací.

O vše se stará třída `FileIO`, která má jako parametr velikost, podle které rozděluje načítaný soubor na menší části a ty se následně zkomprimují a v podobě tripletů uloží do souboru. Zkomprimované soubory se načítají i dekomprimují jako celek, aby nedocházelo k narušení slovníků. Mohlo by se stát, že načteme menší nebo větší část tripletů, než jaká byla vytvořena z jedné části při kompresi. Takový případ by pak způsobil špatné sestavení slovníku, což by vedlo k chybám v dekompresi.

4.3 Triplety

Nedílnou součástí všech implementovaných kompresních metod jsou triplety. Jejich zpracování a následný zápis je zprostředkován pomocí rozhraní `TripletSupplier`, které se vždy používá jen pro vytvoření anonymní vnitřní trídy, která implementuje metodu `visit`, jejímž parametrem je `TripletProcessor`. Ten dokáže pomocí metody `write` zapsat jednotlivé části tripletu do souboru.

Zde je ukázka kódu:

```
output.accept((TripletSupplier) new TripletSupplier() {
        @Override
        public void visit(TripletProcessor visitor) {
        visitor.write(nodeFieldId, num);
        visitor.write(characterFieldId, b);
        }
});
```

Metoda `write` má dva parametry. Prvním je objekt reprezentovaný třídou `TripletFieldId`, který popisuje, kolik bitů má zapisovaná hodnota. `TripletProcessor` se postará o správný zápis bitů. To je velká výhoda, protože u všech metod vždy dopředu vím, jaké největší číslo může být zapsáno.

Například, pokud metoda pracuje se slovníkem, jehož maximální počet frází je 2^{10}, není potřeba zapisovat indexy jako integer, který má 32 bitů. Pro zapsání jakéhokoliv indexu je potřeba nejvýše 11 bitů, zbytek jsou redundantní data. Zmíněný postup v tomto případě ušetří na každém tripletu 22 bitů, což je velká úspora paměti, která výrazně zlepší kompresní poměr. Samotná implementace třídy `TripletFieldId` umožnuje zapsat pouze číslo o velikosti 28 bitů. Tato skutečnost ale není omezující, protože slovník s takovým počtem frází (2^{27} a více) by byl velmi neefektivní.

Druhým parametrem je hodnota zapisované složky tripletu.

4.4 Parametry

Každá metoda má třídu s názvem `<jméno metody>ProviderParametrs`. Tato třída reprezentuje všechny nastavitelné parametry, které ovlivňují kompresi i dekompresi. Metodám se předává jako jediný parametr v konstruktoru. Tyto parametry jsou podrobně popsány dále v realizaci metod.

4.5 Spustitelný klient

Každá metoda má třídu s názvem `<jméno metody>Client`. Tato třída obsahuje statickou funkci *main*, to v Javě znamená, že je možné jí spustit. Jedná se o spustitelný program z příkazové řádky, který komprimuje popřípadě dekomprimuje soubor podle zvoleného přepínače a parametrů. Jednotlivé klienty a jejich použití popisuji pro každou metodu zvlášť v kapitolách realizace. Klient je prozatím spustitelný ve složce `sct/targets` příkazem:

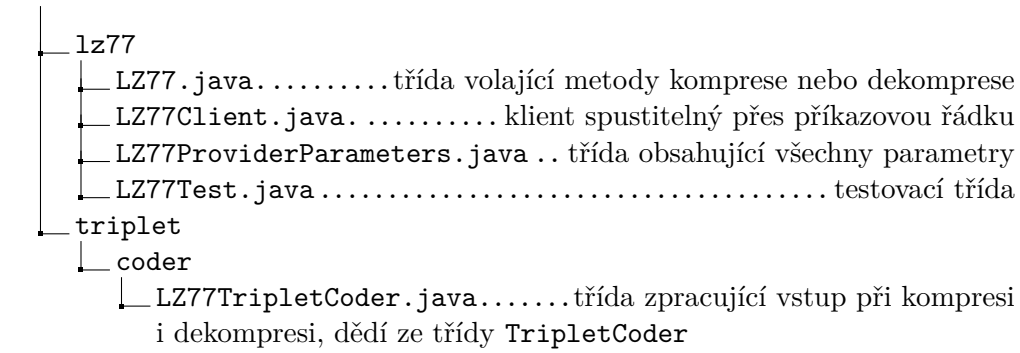
```
java -jar sct-RELEASE.jar <input> <output> <options>
```

Při změně klienta je potřeba v souboru `pom.xml` změnit cestu k správné *main* funkci, kterou chcete spustit a následně zkompilovat projekt.

4.6 Realizace LZ77

4.6.1 Struktura adresáře

Na schématu adresářové struktury můžete vidět třídu LZ77, která inicializuje kodér `LZ77TripletCoder`. Ten se stará o kompresi i dekompresi a zápis tripletů. Podrobnější popis tříd a jejich funkcí naleznete dále v textu.



Struktura 4.1: adresářová struktura metody LZ77

4.6.2 Parametry

Pro metodu LZ77 jsou důležité tři parametry, velikost prohlížecího okna, aktuálního okna a počet bytů, který udává, po jak velkých částech se bude soubor komprimovat. Velikost prohlížecího i aktuálního okna se zadává jako exponent e. Následně se pak velikost oken vypočte jako 2^{e-1}, e je pak nejvyšší počet bitů, potřebný pro zapsání jakehokoliv indexu do tripletu. Velikost třetího parametru se pak pohybuje obvykle od statisíců do jednotek milionů (0,1 - 1 mb), nicméně není vyloučeno použít i jiné hodnoty. Na jak velké části se soubor rozdělí, ovlivní výslednou komprimaci a to jak pozitivně tak i negativně. Jako parametr prohlížecího okna se obvykle volí čísla od 11 do 16, tj. velikost okna 1024 – 32768 znaků. Aktuální okno je typicky mnohem menší, obvykle 64 – 256 znaků, tomu odpovídá parametr od 7 do 9.

4.6.3 Hlavička souboru

Na začátku komprese se uloží jako první speciální triplet, který v této práci označuji jako hlavička souboru. V případě LZ77 obsahuje informaci o tom, jaký byl exponent velikosti prohlížecího a aktuálního okna. Tato informace je důležitá pro načítání dat z tripletů při dekompresi. Udává, kolik bitů je potřeba přečíst, abychom získali uložené číslo. Bez této hlavičky by nebylo možné soubor dekomprimovat.

V budoucnu by bylo vhodné implementovat jiný způsob, jakým se hlavička ukládá do souboru. Implementace třídy `FileIO`, která se stará o načítání a zapisování data, to zatím neumožňuje.

4.6.4 Implementace komprese

Ve třídě `LZ77` je implementována metoda `compress`, která zajišťuje správný průběh komprese. Vstupním parametrem je `byteBuffer`, což je instance třídy `ByteBuffer`. Pokud je vstupní parametr null, je to signál, že je komprese u konce. Na úplném začátku algoritmu se do souboru zapíše hlavička.

Pomocí `byteBufferu` a parametrů se inicializuje instance třídy `LZ77TripletCoder`, která se stará o samotnou kompresi a zápis tripletů. Komprese souboru se volá pomocí metody `encode`, ta má jako parametr konsumera, do kterého se zapisují triplety.

Většina předchozího textu je jen popis použití nástrojů knihovny, který je u všech metod podobný, proto ho dále nebudu popisovat. Z hlediska implementace je nejduležitější metoda `encode`, která zajištuje celý algoritmus komprese LZ77.

Protože načítání vstupů funguje tak, že program načte celou část souboru po bytech do pole, rozhodl jsem se toho využít při implementaci. Místo pole do kterého bych byty přidával a odebíral tak, aby vždy odpovídaly rozměrům prohlížecího a aktuálního okénka, jsem se rozhodl, použít pouze indexy v poli bytů, které metoda dostane jako vstupní parametr.

První index ukazuje, kde v poli začíná prohlížecí okno. Druhý, kde je začátek aktuálního a tedy i konec prohlížecího okna a třetí, kde je konec aktuálního okna. Těmito indexy si tedy ohraničím klouzavé okénko. To se pak velmi snadno posouvá daty. Vždy jen ke všem indexům přičtu velikost zakódovaného řetězce.

Na počátku algorimu jsou první dva indexy rovny nule, druhý se pak s prvním komprimovaným řetězcem začne zvětšovat. Index, který ukazuje na začátek, se začne měnit, až když prohlížecí okénko odpovídá požadované velikosti. Poslední index, který ukazuje na konec aktuálního okénka, je na začátku rovný požadované velikosti, v průběhu komprimace se zvětšuje, dokud nenarazí na konec pole.

Samotné hledání prefixu z aktuálního okénka probíhá tak, že vezmu první byte v aktuálním okénku a pokusím se ho najít v prohlížecím okénku. Pokud je nalezena shoda, vezmu následující byte z aktuálního okénka a porovnám ho s následujícím bytem v prohlížecím okénku. Pokud jsou schodné, postup opakuji, pokud ne, zapamatuji si toto slovo jako kandidáta na nejdelší prefix. A celý postup opakuji, dokud neprojdu celé prohlížecí okno a nenajdu největší možný prefix v aktuálním okně. Prostor pro hledání nejdelšího slova se postupně zmenšuje díky skutečnosti, že v poli před kandidátem na nejdelší slovo žádné delší nemůže být, protože vím, že nejdelší nalezené slovo v před-

chozí časti je právě kandidát. Aktuální okénko je stejné jako na začátku, mění se až po nalezení nejdelšího slova.

Po nalezení nejelšího slova se všechny indexy posunou o jeho velikost zvětšenou o 1 a do konsumera se uloží triplet, který je tvořen třemi složkami:

$$(i, j, b)\,.$$

Kde i je vzdálenost začátku slova od počátku aktuálního okénka. Na zapsání je potřeba počet bitů, který udává první parametr zadaný uživatelem. Další složkou je číslo j, což je délka slova. Počet bitů pro zápis čísla j udává druhý parametr. Poslední složkou je byte b, což je následující byte po nalezeném slově v aktuálním okně. Ten má vždy 8 bitů.

Po kompresi se všechny triplety převedou pomocí třídy `TripletToByteConverter` na byty a následně zapíšou do souboru.

4.6.5 Implementace dekomprese

Uvnitř třídy `LZ77TripletCoder` je naimplementovaná metoda `decode`, která má jako vstupní parametr input třídy `TripletProcessor`, což je výsledek zpracování vstupu třídou `ByteToTripletConverter`, která převádí vstupní byty na triplety.

Implementace dekomprese je velmi jednoduchá. Na začátku si inicializuji proměnnou *builder* třídy `ByteBuilder`, čož je třída obalující pole bytů a poskytující některé vhodné metody pro práci s polem, především pak metodu `appent`, která přidá byte na konec pole.

Následující krok se opakuje, dokud jsou v proměnné *input* triplety. Načtu triplet (i, j, b). Pokud jsou první i druhá složka rovny nule, pak na konec *builderu* přidám byte b z třetí složky tripletu, jinak najdu byte vzdálený i bytů od konce *builderu*. Od tohoto bytu přidám j znaků na konec *builderu*. Sekvence zapsaných bytů odpovídá zakódovanému prefixu. Na konec *builderu* přidám byte b z načteného tripletu.

Pokud dojdou triplety, *input* vratí číslo -1 v následující načtené složce tripletu.

V případě této metody není problém dekódovat i triplety, které vznikly z jiné části při kompresi. Díky skutečnosti, že při dekódování tripletu mám k dispozici vždy celé pole předchozích dekódovaných bytů, vždy správně najdu index, od kterého kopírovat data. To je také důvod, proč nemusí být v hlavičce velikost částí na rozdíl od dalších dvou metod.

Po dekomprimaci se pole bytů zapíše do souboru. Nový soubor je totožný s tím před komprimací.

4.6.6 Spustitelný klient

Klient metody LZ77 může být spuštěn s těmito přepínači:

- **-h** – Vypíše nápovědu.
- **-d** – Spustí dekompresi vstupního souboru. S tímto přepínačem není potřeba zadávat další parametry, program je bude 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).

Poznámka: Pokud při kompresi některý z posledních tří přepínačů chybí nebo je v některém chyba, metoda se zavolá s výchozími parametry, které jsou: -sb 14 -lb 8 -p 1000000.

4.7 Realizace LZ78

4.7.1 Struktura adresáře

Na schématu adresářové struktury můžete vidět třídu LZ78, která inicializuje kodér `LZ78TripletCoder`. Ten se stará o kompresi i dekompresi a zápis tripletů. Podrobnější popis tříd a jejich funkcí naleznete dále v textu.

Struktura 4.2: adresářová struktura metody LZ78

4.7.2 Parametry

Metoda má pouze dva parametry. Prvním je velikost slovníku, tedy počet frází, které je možné uložit. Druhým parametrem je velikost částí (v bytech). Jeho význam je stejný jako u metody LZ77. Podobně jako u LZ77 se první parametr zadává pomocí exponentu e. Velikost slovníku se pak vypočte jako 2^{e-1}, e je počet bitů potřebný pro zapsání jakehokoliv indexu ze slovníku. Obvykle se e pohybuje kolem hodnot 12-19, což odpovídá 2048 – 262 144 frází ve slovníku.

4.7.3 Hlavička souboru

Prvním údajem v hlavičce je parametr e, který udává, kolik bitů má číslo indexu v tripletu. Druhým údajem je velikost částí. Důležitost druhého parametru vysvětlím v části o implementaci dekomprese.

4.7.4 Implementace komprese

Kompresi zajišťuje metoda `encode` uvnitř třídy `LZ78TripletCoder`.

Jako strukturu pro slovník jsem se rozhodl použít strom, jehož uzly jsou reprezentováni pomocí třídy `EncodeNode`. Strom je implementovaný tak, že každý uzel má v sobě mapu potomků. To umožňuje velmi rychlé vyhledávání frází.

Slovník reprezentuje metoda `EncodeDictionary`, `compress` vždy načte byte a předá ho slovníku, aby ho zpracoval pomocí metody `processByte`.

Slovník si drží ukazatel na kořen stromu a na aktuální uzel. Kořen sám funguje jako prázdný znak, který má v mapě potomků všech 256 bytů. Jedná se o inicializaci slovníku, která je popsaná v kapitole o analýze algoritmu LZ78 a probíhá v konstruktoru slovníku.

Na začátku je ukazatel na aktuální uzel rovný ukazateli kořenu. Vždy, když se zavolá metoda `processByte`, zkontroluje se, zda má aktuální uzel v mapě potomků uzel na hodnotě načteného bytu. Pokud ano, ukazatel na aktuální uzel se posune do potomka. Pokud ne, zapíše se triplet

$$(i, b)\,.$$

Kde i je číslo aktuálního uzlu a b je načtený byte. Aktuálnímu uzlu se přidá potomek do mapy na hodnotu b. Ukazatel na aktuální uzel se posune do kořene.

Tímto způsobem se ve slovníku zjišťuje, jestli obsahuje hledanou frázi. Pokud slovník zjistí, že ji neobsahuje, přidá jí.

V průběhu komprese se kontroluje, kolik frází je již ve slovníku. Pokud počet frází dosáhne maxima, které udává parametr, slovník se zamrazí. Komprese pokračuje dál, nicméně slovník se už nezvětšuje.

4.7.5 Implementace dekomprese

Kompresi zajišťuje metoda `decode` uvnitř třídy `LZ78TripletCoder`.

Pro účely testování jsem naimplementoval dva slovníky pro dekompresi. Liší se především tím, s jakou složitostí přidávají a získávají ze slovníku fráze.

Na úplném začátku se načte hlavička a inicializuje slovník jako u komprese. Dále se načte triplet ze souboru. Jeho složky jsou index i a následující byte b. Index určuje, kde se nachází fráze ve slovníku. Zde se implementace liší.

Třída `DecodeList` reprezentuje slovník jako pole frází. Pomocí indexu z tripletu najdu frázi a tu poté zapíšu do souboru. Následně je na konec slovníku přidána fráze složená ze zapsaného řetězce a bytu b.

Třída `DecodeTree` si slovník udržuje jako strom, podobně jako při kompresi. Rozdíl je v tom, že všechny uzly jsou uloženy v poli ve stejném pořadí, v jakém byly přidány, aby bylo možné hledat uzel na indexu i v konstatní složitosti. Fráze se pak zrekonstruuje rekurzivně pomocí odkazů na rodiče, který

každý uzel osahuje. Kořen funguje jako zarážka rekurze, protože jeho rodič je null. Ukončovací podmínka tedy zní: Pokud je rodič null, zastav rekurzi.

Poté, co je fráze získaná ze slovníku, zapíše se na konec výstupního souboru společně s následujícím bytem b. Následně je do stromu vložen potomek uzlu s indexem i na hodnotou b, což představuje novou frázi.

V přidávání frází do slovníku se tyto implementace také liší. Zatímco třída `DecodeList` vytvoří novou frázi tak, že zkopíruje řetězec a přidá k němu byte b, třída `DecodeTree` přidá uzlu na indexu i pouze potomka do mapy s klíčem hodnoty b.

Třída `DecodeList` má tedy složitější přidávání frází, ale jednodušší hledání ve slovníku. U třídy `DecodeTree` je tomu přesně naopak. Tato třída sice najde uzel v konstantním čase, ale renstrukce slova je podobně složitá, jako kopírování u první metody. Nicméně testování těchto dvou implementací ukázalo, že se v rychlosti dekomprese neprojeví žádné velké rozdíly. Je to způsobeno tím, že počet kopírování a rekonstrukcí slov je stejný.

V průběhu dekomprese algoritmus zaznamenává, kolik dat bylo dekódováno. Pokud počet dekódovaných dat dosáhne velikosti druhého parametru z hlavičky, stávající slovník se vymění za nově vytvořený. Nový slovník je vytvořen pomocí konstruktoru a obsahuje pouze fráze vzniklé při inicializaci. Proces tvorby slovníku začne znovu stejně jako při kompresi nové části. Tímto zpusobem se správně dekódují jednotlivé části souboru.

4.7.6 Spustitelný klient

Klient metody LZ78 může být spuštěn s těmito přepínači:

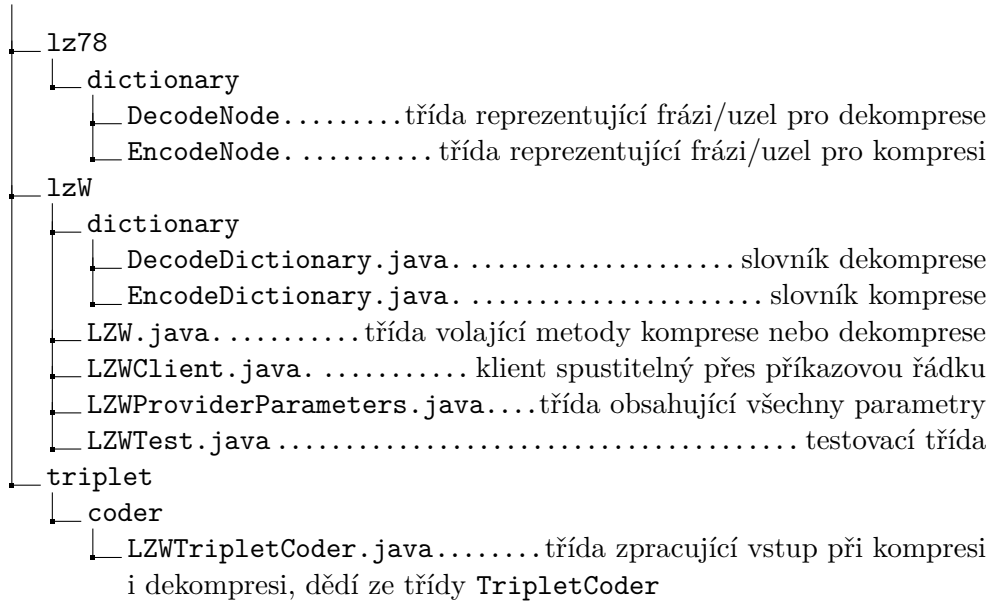
- **-h** – Vypíše nápovědu.
- **-d** – Spustí dekompresi vstupního souboru. S tímto přepínačem není potřeba zadávat další parametry, program je bude ignorovat, protože si načte hlavičku, ve které jsou všechny potřebné parametry.
- **-nc** <e> – Tento přepínač říká, kolik maximálně frází muže být ve slovníku (2^{e-1}).
- **-p** <n> – Tento přepínač říká, po jak velkých částech bude probíhat komprese (v bytech).

Poznámka: Pokud při kompresi některý z posledních dvou přepínačů chybí nebo je v některém chyba, metoda se zavolá s výchozími parametry, které jsou: -nc 17 -p 1000000.

4.8 Realizace LZW

4.8.1 Struktura adresáře

Na schématu adresářové struktury můžete vidět třídu LZW, která inicializuje kodér `LZWTripletCoder`. Ten se stará o kompresi i dekompresi a zápis tripletů. Metoda LZW využívá pro reprezentaci uzlů ve slovníku stejné třídy jako LZ78. Podrobnější popis tříd a jejich funkcí naleznete dále v textu.



Struktura 4.3: adresářová struktura metody LZW

4.8.2 Parametry a hlavička souboru

Metoda má totožnou hlavičku i nastavitelné parametry jako LZ78.

4.8.3 Implementace komprese

Kompresi zajišťuje metoda `encode` uvnitř třídy `LZWTripletCoder`.

Od imlementace komprese LZ78 se tato implementace liší pouze v chování při zápisu tripletu, který má pouze jedinou složku:

$$(i)\,.$$

Zapisuje totiž pouze index ve slovníku. Poté přidá nového potomka aktuálnímu uzlu s hodnotou posledního načteného bytu. Následně neposune ukazatel do kořene stromu, jako je tomu u LZ78, ale do potomka kořene, který má hodnotu jako zpracovávaný byte.

4.8.4 Implementace dekomprese

Dempresi zajišťuje metoda `decode` uvnitř třídy `LZWTripletCoder`.

Třída `DecodeDictionary` ve složce `lzw` reprezentuje slovník, který používá strukturu stromu. Jeho uzly jsou uloženy v poli, aby bylo možné hledat index fráze v konstantní složitosti. Fráze se pak sestaví pomocí rekurzivního procházení přes rodiče až do kořene.

Hlavní rozdíl proti dekompresi LZ78 je, že načtený index z tripletu nemusí být ve slovníku. V takovém případě je dekódovaný řetězec stejný jako poslední přidaná fráze. Do stromu se přidá uzel s hodnotou, která odpovídá prvnímu znaku naposledy přidané fráze. Tento uzel je potomkem naposledy přidaného uzlu.

Pokud index ve slovníku je, načte se příslušná fráze. Tato fráze se zapíše do souboru. Do slovníku se přidá předchozímu použitému uzlu potomek na hodnotou prvního znaku fráze.

4.8.5 Spustitelný klient

Klient metody LZW může být spuštěn s těmito přepínači:

- **-h** – Vypíše nápovědu.
- **-d** – Spustí dekompresi vstupního souboru. S tímto přepínačem není potřeba zadávat další parametry, program je bude ignorovat, protože si načte hlavičku, ve které jsou všechny potřebné parametry.
- **-nc** <e> – Tento přepínač říká, kolik maximálně frází muže být ve slovníku (2^{e-1}).
- **-p** <n> – Tento přepínač říká, po jak velkých částech bude probíhat komprese (v bytech).

Poznámka: Pokud při kompresi některý z posledních dvou přepínačů chybí nebo je v některém chyba, metoda se zavolá s výchozími parametry, které jsou: -nc 17 -p 1000000.

Kapitola 5

Testovaní

Při určování kvality algoritmu se obvykle zjišťuje efektivita a časová náročnost. Efektivitu jsem se rozhodl vyjádřit pomocí kompresního poměru. Ten se vypočítá jako podíl velikostí zkomprimovaného a původního souboru. Časová naročnost je údaj, za kolik sekund program dosáhne správného výstupu.

Všechny tři metody jsou efektivní na datech, kde se opakují podřetězce, tedy například na běžném anglickém textu. Proto jsem se rozhodl k testování použít korpus Calgary. Ten se skládá z 18 souborů s anglickým textem o celkové velikosti 3 190 KB. Abych demonstroval i nevýhody těchto algoritmů, jako druhý korpus použiji ten s názvem Prague. Takzvaný pražský korpus je specifický svou různorodostí. Obsahuje totiž soubory s textem i s náhodnými daty o celkové velikosti 56 900 KB. Dosáhnout dobrého kompresního poměru na pražském korpusu je obecně velmi těžké.

Všechna měření uvedená v této kapitole byla pořízena na počítači s těmito parametry:

Procesor – Intel Core i7 (frekvence jednoho jádra - 2,5 GHz)
RAM – 8 GB
Operační systém – Windows 10 (testování – příkazová řádka Git Bash)

5.1 Měření LZ77

V tabulkách 5.1 a 5.2 můžete vidět statistiku běhu algoritmu LZ77 na korpusech Calgary a Prague v závislosti na vstupním parametru e. Ostatní parametry se při testování neměnily. Velikost aktuálního okénka je 8 tedy 128 znaků a velikost částí je 2 miliony bytů (2 MB).

Jak je vidět v obou tabulkách měření (5.1 a 5.2), hlavní problém metody LZ77 je rostoucí časová složitost s velikostí prohlížecího okna. Zjednodušeně by se dalo říct, že s velikostí parametru e roste složitost komprese exponenciálně. Tomu odpovídají i naměřené časy. Komprese s parametrem e trvá téměř dvojnásobný čas než s $e - 1$ a čtyřnásobný než s $e - 2$. Nicméně kompresní poměr je poměrně dobrý. Především u korpusu Calgary.

Zajímavým poznatkem měření je mírně klesající doba dekomprese. S větším parametrem klesá kompresní poměr, což znamená, že se zmenšuje i počet tripletů v souboru. Klesající tendence doby dekomprese je způsobená menší režií na načítání a zpracování tripletů. Z dat v tabulkách je také vidět, že časová složitost dekomprese nezávisí na e.

Z obou tabulek je patrné, že s rostoucím parametrem e, se zlepšuje kompresní poměr, ovšem doba běhu exponenciálně roste. Nicméně na korpusu Calgary došlo k poměrně dobré kompresi již při menším parametru 12, což odpovídá prohlížecímu oknu velikosti 2048 znaků. Oproti tomu na Prague není komprese dobrá ani při parametru 15, což odpovídá 16384 znakům v prohlížecím okně. Je také patrné, že při testování na Calgary se kompresní poměr zlepšuje rychleji než na Prague. Jinými slovy pomyslná křivka kompresního poměru klesá rychleji u Calgary.

Neefektivní kompresi na korpusu Prague způsobuje především velká náhodnost dat. Algoritmus nenachází delší prefixy a ve spoustě případů zapisuje triplet, který zabírá více místa než řetězec, který kóduje.

V tabulce 5.1 uvádím pouze hodnoty parametru e od 10 do 17. Je tomu tak proto, že $e < 10$ se téměř nepoužívá, kvůli špatné efektivitě algoritmu a $e > 17$ už kompresi v tomto případě nezlepšilo. V tabulce 5.2 je tomu pro $e < 10$ obdobně. Pro $e > 15$ běží již algoritmus příliš dlouho a je tedy nepoužitelný. Pro představu pokud $e = 16$ algoritmus běží 10 minut a pro $e = 17$ zhruba 20 minut.

Ačkoliv se může z tabulek zdát, že čím větší e, tím lepší kompresní poměr, není tomu tak. Obvykle pro každý soubor existuje hranice, kdy už se komprese přestane zlepšovat. Způsobuje to zapisování čísel do tripletů. Parametrem e se nespecifikuje pouze velikost oken, ale i počet bitů potřebný pro zápis čísel do tripletu. Při příliš velkém e se stane, že pro zapisování tripletů bude používáno více bitů, ale jejich počet zůstane téměř stejný jako u menších e.

Parametr e	10	11	12	13	14	15	16	17
Kompresní poměr	0,75	0,65	0,60	0,57	0,53	0,51	0,49	0,47
Čas komprese (s)	3,9	4,4	5,5	7,5	12,0	20,5	32,6	57,0
Čas dekomprese (s)	3,5	3,1	3,0	2,9	3,1	3,1	3,0	2,8

Tabulka 5.1: testování LZ77 na Calgary

Parametr e	10	11	12	13	14	15
Kompresní poměr	0,85	0,85	0,84	0,83	0,80	0,78
Čas komprese (s)	30,9	42,6	62,5	103,0	180,0	330,5
Čas dekomprese (s)	34,6	32,8	31,5	30,5	27,5	26,9

Tabulka 5.2: testování LZ77 na Prague

5.2 Měření LZ78

V tomto měření byla pro dekompresi použita implementace pomocí stromu.

Podobně jako u měření LZ77 i zde dopadlo lépe testování na Calgary. Korpus Prague má příliš náhodná data a slovník vytváří pouze krátké fráze.

U této a LZW metody jsem měřil i hloubku stromů, které reprezentují slovník. Zatímco u Calgary měly slovníky jednotlivých částí souboru obvykle hloubku kolem sta uzlů, u Prague ve většině případů nepřesáhla deset. V některých případech byla hloubka pouze tři. Což znamená, že nejdelší zakódovaný řetězec měl délku tři. To se muže zdát jako nesmysl, ale do stromu hloubky tři se vejde $(256^2 + 256^3)$ podřetězců délky dva a tři. To je 16 842 752 uzlů, což více než osmkrát přesahuje maximální počet uzlů ve slovníku pro parametr 22. K těmto případům obvykle dochází na datech s vysokou entropií. Z toho důvodu se tyto metody nepoužívají pro komprimaci obrázků. Ve vetšině případů je totiž zvětšují.

Z naměřených dat v tabulkách 5.3 a 5.4 je vidět, že mé odhady složitosti byly správné. Rychlost komprese i dekomprese je přibližně stejná a nezávisí na velikosti parametru e. Nicméně čas komprese mírně roste s vyšším parametrem e, kdežto dekomprese mírně klesá. Zatímco u komprese je to způsobeno větší režií při přidávání frází do slovníku, u dekomprese se jedná o stejný důvod jako u LZ77. S vyšším parametrem se snižuje počet tripletů a tím i režie s jejich načítáním a zpracováním.

Korpus Calgary dokázala metoda LZ77 zmenšit na 47 % původní velikosti, což je nejlepší výsledek z měření, nicméně metoda LZ78 zmenší Calgary na 51 % za 35 krát kratší dobu než LZ77. Dá se tedy říci, že metoda LZ78 provádí podobně efektivní kompresi v mnohem lepším čase.

Parametr e	16	17	18	19	20	21
Kompresní poměr	0,67	0,62	0,60	0,54	0,53	0,51
Čas komprese (s)	1,4	1,5	1,5	1,5	1,6	1,6
Čas dekomprese (s)	1,9	1,8	1,8	1,7	1,7	1,7

Tabulka 5.3: testování LZ78 na Calgary

Parametr e	17	18	19	20	21	22
Kompresní poměr	0,81	0,76	0,74	0,75	0,75	0,75
Čas komprese (s)	13,4	13,9	13,9	14,5	14,6	15,0
Čas dekomprese (s)	24,5	23,8	21,8	21,0	22,7	22,1

Tabulka 5.4: testování LZ78 na Prague

5.3 Měření LZW

Metoda LZW je v mnoha ohledech podobná jako LZ78. Není tomu jinak ani u výsledků měření. Ty dopadly velmi podobně co se rychlosti komprese týká, nicméně LZW dosáhla stejného komprimačního poměru s nižším parametrem. Při porovnání tabulek 5.5 a 5.3 můžeme pozorovat, že při $e = 20$ byl komprimační poměr u LZW na korpusu Calgary lepší, než u LZ78.

Podobně jako u ostatních metod není čas dekomprese závislý na vstupním parametru e. Je ovlivněn počtem tripletů na vstupu, nicméně, jak jsem již psal u předchozích metod, ani zde neznamená vyšší parametr lepší kompresi.

V tabulce 5.6 je vidět, že od parametru $e = 20$ se kompresní poměr nezlepšuje. Příčinou je rostoucí velikost tripletů zatímco jejich množství se zmenšuje jen velmi málo nebo vůbec. K tomu obvykle dochází, pokud se slovník nenaplní frázemi. To nastane v případě, že je parametr příliš velký oproti velikosti části souboru, který komprimuje. Tento případ nastal u měření s parametrem $e = 21$ a $e = 22$ v tabulce 5.6.

Z dat je patrné, že komprese i dekomprese probíhají velmi rychle.

Parametr e	14	15	16	17	18	19	20
Kompresní poměr	0,76	0,73	0,68	0,64	0,59	0,54	0,50
Čas komprese (s)	1,4	1,4	1,4	1,5	1,5	1,6	1,6
Čas dekomprese (s)	2,1	2,0	1,9	1,9	1,8	1,8	1,7

Tabulka 5.5: testování LZW na Calgary

Parametr e	17	18	19	20	21	22
Kompresní poměr	0,89	0,86	0,80	0,75	0,75	0,75
Čas komprese (s)	12,3	12,8	14,2	14,7	15,3	15,5
Čas dekomprese (s)	29,5	28,3	25,3	24,2	24,1	24,1

Tabulka 5.6: testování LZW na Prague

Závěr

Cílem této práce bylo navrhnout, analyzovat, implementovat a testovat kompresní algoritmy LZ77, LZ78 a LZW. Implementace byla provedena jako součást knihovny SCT.

Výsledkem práce je knihovna SCT rozšířená o výše zmíněné algoritmy. Analýza těchto metod mimo jiné ukázala, že nejsou vhodné pro data s vysokou datovou entropií. Nicméně i přesto se hodí pro komprimaci běžného textu, případně i dat, kde se opakují delší podřetězce. Metody LZ78 a LZW dosahují v kratším čase podobného kompresního poměru jako LZ77. Je tedy možné je označit za efektivnější.

Všechny metody byly testovány na předem zvolených korpusech Calgary a Prague, aby bylo možné výsledky porovnávat. Všem metodám se povedlo korpus Calgary zmenšit o zhruba 50 % a Prague o 25 %.

I přes své nevýhody jsou metody poměrně efektivní a je možné je prostřednictvým knihovny SCT užívat v praxi.

Literatura

[1] *Data Compression* [online]. [cit. 2018-04-22]. Dostupné z: https://www.techopedia.com/definition/884/data-compression

[2] *Lexikon pojmů* [online]. [cit. 2018-05-07]. Dostupné z: http://stringology.org/DataCompression/lexikon.html

[3] *Informace, entropie a fyzika* [online]. [cit. 2018-05-07]. Dostupné z: https://popelka.ms.mff.cuni.cz/~lessner/mw/index.php/U%C4%8Debnice/Informace/Informace,_entropie_a_fyzika

[4] *Crush* [online]. [cit. 2018-04-23]. Dostupné z: https://sourceforge.net/projects/crush

[5] *LZMA Compression* [online]. [cit. 2018-04-23]. Dostupné z: https://www.advancedinstaller.com/user-guide/lzma-compression.html

[6] *LZW Compression* [online]. [cit. 2018-04-23]. Dostupné z: https://www.prepressure.com/library/compression-algorithm/lzw

[7] *Modelování požadavků* [online]. [cit. 2018-04-23]. Dostupné z: https://edux.fit.cvut.cz/oppa/BI-SI1/prednasky/BI-SI1-P03m.pdf

[8] *Small compression toolkit* [online]. [cit. 2018-05-06]. Dostupné z: https://gitlab.fit.cvut.cz/polacrad/sct

[9] *Algoritmus LZ77* [online]. [cit. 2018-04-25]. Dostupné z: http://voho.eu/wiki/algoritmus-lz77/

[10] *Algoritmus LZ78* [online]. [cit. 2018-04-28]. Dostupné z: http://stringology.org/DataCompression/lz78/index_cs.html

[11] *Holub J.: Data compression, dictionary metod I. (5. přednáška)* [online]. [cit. 2018-04-28]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-KOD/_media/lectures/05/mi-kod-05-dictionary.pdf

[12] *Holub J.: Data compression, dictionary metod II. (6. přednáška)* [online]. [cit. 2018-04-28]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-KOD/_media/lectures/06/mi-kod-06-dictionary.pdf

[13] *Algoritmus LZW* [online]. [cit. 2018-04-28]. Dostupné z: http://www.stringology.org/DataCompression/lzw-e/index_cs.html

Příloha A

Seznam použitých zkratek

SCT Small compression toolkit

LZ Lempel-Ziv

LZW Lempel-Ziv-Welch

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`
Zdrojová forma práce ve formátu LaTeX je také dostupná z: `https://gitlab.fit.cvut.cz/zemeklad/BP`

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy knihovny s implementací metod
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `BP_zemek_ladislav_2018.pdf` ..... text práce ve formátu PDF