



**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 SOFTWAREVÉHO INŽENÝRSTVÍ



Bakalářská práce

Implementace kompresních metod LZ77, LZ78 a LZW v jazyce Java

Ladislav Zemek

Vedoucí práce: Ing. Radomír Polách

1. května 2018

Poděkování

Rád bych poděkoval Ing. Radomírovi Poláchovi za odborné vedení, pomoc a vstřícnost při vedení bakalařské práce.

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 1. 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 a 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 vhodným návrhem a popisem daných algoritmů. U všech metod se mi podařilo při vhodně zvolených parametrech dosáhnout zmenšení testovaných korpusů vždy alespoň o 30 %. Přínosem této práce je přispění do knihovny, která může v budoucnu usnadnit některým vývojářům práci.

Klíčová slova Komprese dat, dekomprese dat, chaining, 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 in the future 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 30 %. The benefit of this work is contributing to a library that can make it easier for some developers to work in the future.

Keywords Compression, decompression, chaining, LZ77, LZ78, LZW, SCT

Obsah

Úvod	1
Pojmy	1
1 Cíl práce	3
1.1 Existující řešení	3
1.2 Analýza požadavků	3
2 Analýza algoritmů	7
2.1 LZ77	7
2.2 LZ78	10
2.3 LZW	12
2.4 Závěr analýzy	15
3 Implementace	17
3.1 Řetězení	17
3.2 Načítání a zápis do souboru	18
3.3 Triplety	18
3.4 Parametry	19
3.5 Spustitelný klient	19
3.6 Realizace LZ77	20
3.7 Realizace LZ78	23
3.8 Realizace LZW	23
4 Testování	25
Závěr	27
Literatura	29
A Seznam použitých zkratek	31

Seznam obrázků

3.1	adresářová struktura metody LZ77	20
-----	--	----

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 s kapacitou stovek gigabytů. Ovšem tehdy měli pevné disky jen desítky megabytů, což je téměř sto tisíckrát méně než dnes, paměti tedy rozhodně nebylo nazbyt. Proto bylo potřeba data nějakým způsobem zmenšit, 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 dekodovat 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.

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 obnovím ty 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 pak používá ke komprimaci. Není potřeba tyto struktury ukládat společně s komprimovanými daty.
- **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. Komprimovaná data poté obsahují indexy frází ve vytvářeném slovníku.
- **Symetrická kompresní metoda** – Komprimace i dekomprimace těchto metod má stejnou složitost.
- **Asymetrická kompresní metoda** – Komprimace i dekomprimace těchto metod má odlišnou složitost.
- **Entropie** – míra neurčitosti systému

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 následné otestování z hlediska časové a paměťové náročnosti, případně porovnání efektivity se stávajícími kompresními algoritmy v knihovně. Tato knihovna by pak měla ušetřit spoustu práce programátorům, kteří díky ní nebudou muset vymýšlet vlastní implementaci těchto algoritmů.

1.1 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.[2] Z LZ77 bylo odvozeno mnoho dalších kompresních algoritmů, které jsou velmi často používány. LZ78 byla taktéž předlohou pro jiné algoritmy například LZW nebo BTLZ. [3] Samotná se příliš nevyužívá, protože nedosahuje tak dobrého komprimačního poměru. Naopak metoda LZW bývala poměrně hojně využívána 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 nahrazena efektivnějším algoritmem.[4] 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.

1.2 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 být definovaný dostatečně detailně pro účely návrhu.[5] Obvykle se požadavky dělí na funkční a nefunkční.

1.2.1 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:

- **Dekomprese** – Kromě výchozí funkce komprese bude možné program spustit v režimu dekomprese.
- **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 spoštění přes příkazovou řádku pomocí parametrů a nebo
- **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 pustit se zvolenými parametry z příkazové řádky. Nepůjde tedy pouze o implementaci metod do knihovny, ale i o jejich interface, který bude možné prakticky využít.

1.2.2 Nefunkční požadavky

Nefunkční požadavky, jsou takové 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:

- **Triplety** – Program bude pro ukládání dat do souboru využívat již naimplementované metody třídy TripletProcesor a pro definování jednotlivých složek v tripletu třídu TripletFieldId.
- **Velikostí slovníků a bufferů** – Aby nedocházelo k nekontrolovatelnému růstu slovníku a tím i k zvyšování paměťové i časové náročnosti, budou mít algoritmy LZ78 a LZW parametr, který zhora omezí počet prvků ve slovníku. U metody LZ77 bude pomocí dvou parametrů omezen vyhledávací buffer i tzv. look ahead buffer.
- **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** – Program bude napsán v jave, 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.

1.2.3 Výjimky

??

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, náročnost na paměť, výhody a nevýhody.

2.1 LZ77

2.1.1 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 - prohlížecké 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 prohlížeckého okénka musí být vždy větší nebo rovna velikosti aktuálního okénka, nicméně v praxi je prohlížecké okénko obvykle tisíci násobně větší než aktuální.[6]

2.1.2 Komprese

Hlavní myšlenka tohoto algoritmu spočívá v nalenzení co nejdelšího prefixu z nezakódovaných dat, který je zároveň obsažený i v prohlížeckém okénku. Tato metoda umožňuje menší vylepšení efektivity, pokud se při vyhledávání neomezím pouze na prohlížecké 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 prohlížeckém okénku.

Zde je jednoduchý pseudokód, který popisuje tuto metodu:

```
prohlížeč a aktuální okénko 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 prohlížečím  
    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 následující 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

2.1.3 Složitost komprese

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

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

$$O\left(\sum_{i=1}^{|P|} mp_i\right)$$

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

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

$$O\left(m \sum_{i=1}^{|P|} p_i\right)$$

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

$$O\left(\sum_{i=1}^{|P|} mp_i\right) = O(mn)$$

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.

2.1.4 Dekomprese

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

```
prohlížečí 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 prohlížečího okénka,  $j$  je délka
     $p$  a  $k$  je první následující symbol po  $p$ ;
    if  $i$  se rovná 0 then
        zapiš  $k$ ;
        přidej na konec prohlížečího okénka  $k$ ;
    else
         $v$  := aktuální velikost prohlížečího okénka;
         $z$  :=  $v - i$ ;
         $p$  := podřetězec prohlížečího okénka od indexu  $z$  az do  $z + j$ ;
         $p$  :=  $p + k$ ;
        zapiš  $p$ ;
        do prohlížečího okénka přidej  $p$ ;
    end
end
```

Algoritmus 2: Pseudokód dekomprese metody LZ77

2.1.5 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 prohlížečího 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.

2.1.6 Výhody

- **Paměť** – Díky posuvnému okénu se zpracovává vždy pouze poměrně malá část dat, proto neroste paměťová náročnost s velikostí vstupu.
- **Rychlost dekomprese** – Při vhodné zvolené implementaci je algoritmus dekomprese lineárně závislý pouze na velikosti dat. Jeho složitost je tedy $\Theta(n)$, kde n je velikost původních dat.

2.1.7 Nevýhody

- **Náhodná data** – Pokud mají data příliš velkou entropii, 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.

2.2 LZ78

2.2.1 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.[7]

2.2.2 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

2.2.3 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:

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

2.2.4 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 zapiše 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

2.2.5 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 režiji 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 .

2.2.6 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.

2.2.7 Nevýhody

- **Paměť** – Paměťová náročnost roste s velikostí vstupu. Proto se při implementaci musí počítat s tím, že metoda vyčerpá přidělenou paměť a podle toho se zachovat.
- **Náhodná data** – Podobně jako u metody LZ77, při velké entropii dat dochází k horší kompresi, někdy může dojít až k zvětšení souboru.

2.3 LZW

Tuto slovníkovou, symetrickou, adaptivní a jednopřechodovou 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.[8]

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

2.3.1 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 hlaní změna oproti metodě LZ78, která umožňuje právě vynechání následujícího bytu v tripletu. Tento postup se opakuje dokud jsou na vstupu data.

```

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

2.3.2 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 dovoluji vynechat její odvození.

2.3.3 Dekomprese

Princip dekomprese není na první pohled 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.

Speciální 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

2.3.4 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.

2.3.5 Výhody

- **Rychlost komprese i dekomprese** – Stejně jako u LZ78 je hlavní výhodou rychlost. Při správné implementaci $\Theta(n)$.
- **Velikost tripletů** – Do souboru se ukládají pouze čísla. Na jeden triplet je tedy potřeba uložit nejmeně 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

2.3.6 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 paměť.
- **Náhodná data** – Podobně jako u předchozích metod, dochází při velké entropii dat k horší kompresi. Může dojít až k zvětšení souboru.

2.4 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 závislí na velikosti slovníku. Nicméně i přes tyto nevýhody se LZW v některých programech používá dodnes, protože je velmi rychlá a vhodná na klasický text.

Implementace

V této kapitole popisují 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. Jakým způsobem metody načítají data ze vstupu a jak zapisují triplety do souboru.

3.1 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());
```

Význam kódu není zatím relevantní, nicméně představuje použití komprese. 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) {...}
```

3. IMPLEMENTACE

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[]>>
    { ... }
```

Zde je vstupní parametr typu `ByteBuffer`, to znamená že předchozí metoda v řetězci musela mít druhý parametr typu `Consumer<ByteBuffer>`. Výstupním parametrem je `TripletSupplier`, následující metoda tedy musí mít první parametr `TripletSupplier`. V tomto případě je to objekt `t2b`, který je instancí třídy `TripletToByteConverter`, která implementuje rozhraní `Chainable<TripletSupplier, List<byte[]>`.

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

3.2 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 rozděljuje 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 vedl k sestavení špatného slovníku, což by vedlo k chybám v dekompresi.

3.3 Triplet

Nedílnou součástí všech implementovaných kompresních metod jsou triplety. Jejich spracování a následný zápis je sprostředkován pomocí rozhraní `TripletSupplier`, které se vždy používá jen pro vytvoření anonymní vnitřní třídy, 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. Druhým parametrem je hodnota zapisované složky tripletu. 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 maximálně 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. Tento postup v tomto případě ušetří na každém tripletu 22 bitů, což je velká úspora paměti, která výrazně zlepší kompresní poměr. Samotná implementace třídy `TripletFieldId` umožňuje zapsat pouze číslo o velikosti 28 bitů. Tato skutečnost ale není omezující, protože slovník s takovým počtem frází (2^{29} a více) by byl velmi neefektivní.

3.4 Parametry

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

3.5 Spustitelný klient

Každá metoda má třídu s názvem `<jméno metody>Client.java`. Tato třída obsahuje funkci `main`, to znamená, že je možné ji spustit. Jedná se o spustitelný program z příkazové řádky, který komprimuje popřípadě dekomprimuje soubor podle zvoleného přepínače a parametrů. Jednotlivé klienty a jejich použití popisují pro každou metodu zvlášť v kapitolách realizace. Klient je zatím spustitelný příkazem:

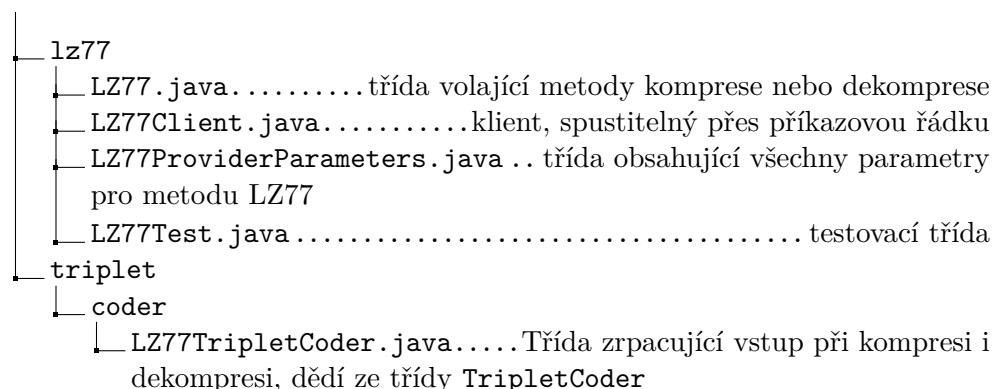
```
java -jar target/set-RELEASE.jar <jméno vstupního souboru>  
<jméno vytvořeného souboru> <případné přepínače>
```

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

3.6 Realizace LZ77

3.6.1 Struktura adresáře

Na schématu adresářové struktury můžete vidět třídu LZ77, která se stará o volání funkcí komprese a dekomprese. Také inicializuje kodér LZ77TripletCoder, který se stará o vytváření tripletů. Podrobnější popis tříd a jejich funkcí naleznete dále v textu.



Struktura 3.1: adresářová struktura metody LZ77

3.6.2 Parametry

Pro metodu LZ77 jsou důležité tři parametry. Velikost prohlížečského okna, aktuálního okna a počet bytů, který udává po jak velkých částech se bude soubor komprimovat. Velikost prohlížečského 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 jakéhokoli indexu do tripletu. Velikost třetího parametru se pak pohybuje obvykle od statisíců do desítek milionů (0,1 - 10 mb), nicméně není vyloučeno použít i jiné hodnoty. Na jak velké části soubor rozdělíte, ovlivní výslednou komprimaci. Jako parametr prohlížečského 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.

3.6.3 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 prohlížečského a aktuálního okna. Tato informace je důležitá pro načítání dat z tripletů. Udává kolik bitů je potřeba přečíst abychom získali uložené číslo. Bez této hlavičky by nebylo možné soubor dekomprimovat.

3.6.4 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 byla jen popis použití nástrojů knihovny, který je u všech metod podobný, proto ho dále nebudu popisovat. Z hlediska implementace je nejdůležitější metoda `encode`, která zajišťuje 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 odpovídali rozměrům prohlížečského a aktuálního okénka, jsem se rozhodl, použít pouze indexy v poli bytů.

První index ukazuje, kde v poli začíná prohlížečské okno. Druhý, kde je začátek aktuálního a tedy i konec prohlížečského 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 algoritmu 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 prohlížečské 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 prohlížečském okénku. Pokud je nalezena shoda, vezmu následující byte z aktuálního okénka a porovnáím ho s následujícím bytem v prohlížečském okénku. Pokud jsou schodné, postup opakuji, pokud ne, zapamatuji si toto slovo jako kandidáta na nejdelší prefix. A celý postup opakuji dokud neprojdou celé prohlížečské okno a nenajdu největší možný prefix. Prohlížečské okno se postupně zmenšuje díky skutečnosti, že v poli před kandidátem na nejdelší slovo žádné delší nemůže být, protože vím, že nejdelší nalezený prefix v předchozí části je právě kandidát. Aktuální okénko je stejné, jako na začátku, mění se až po nalezení nejdelšího prefixu.

3. IMPLEMENTACE

Po nalezení nejdelšího prefixu 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 prefixu. Počet bitů pro zápis čísla j udává druhý parametr. Poslední složkou je byte b , následující po nalezeném prefixu v aktuálním okně. Ten má vždy 8 bitů.

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

3.6.5 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 tripletty.

Implementace dekomprese je velmi jednoduchá. Na začátku si inicializují proměnou `builder` třídy `ByteBuilder`, což je třída obalující pole bytů a poskytující některé vhodné metody pro práci s polem, především pak metodu `append`, která přidá byte na konec pole.

Následující krok se opakuje dokud jsou v proměnné `input` tripletty. 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 zapídaných bytů odpovídá zakódovanému prefixu. Na konec `builderu` přidám byte b z tripletu.

Pokud dojdou tripletty, `input` vrátí číslo -1 v následující načtené složce tripletu.

V případě této metody není problém dekódovat i tripletty, které vznikli z jiné části při kompresi. Díky skutečnosti, že při dekódování tripletu mám k dispozici vždy celé pole předchozích 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.

3.6.6 Spustitelný klient

Tento klient 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 pro dekompresi
- **-sb <e>** – tento přepínač říká, jak velké bude prohlížečí okénko (2^{e-1})
- **-lb <e>** – tento přepínač říká, jak velké bude aktuální okénko (2^{e-1})
- **-p <n>** – tento přepínač říká, po jak velkých částech bude probíhat komprese (v bytech)

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

3.7 Realizace LZ78

3.8 Realizace LZW

Testování

Závěr

Literatura

- [1] *Komprese dat* [online]. [cit. 2018-04-22]. Wikipedie. Dostupné z: https://cs.wikipedia.org/wiki/Komprese_dat
- [2] *Crush* [online]. [cit. 2018-04-23]. Dostupné z: <https://sourceforge.net/projects/crush>
- [3] *LZ77 and LZ78* [online]. [cit. 2018-04-23]. Wikipedie. Dostupné z: https://en.wikipedia.org/wiki/LZ77_and_LZ78
- [4] *LZW* [online]. [cit. 2018-04-23]. Dostupné z: <https://www.prepressure.com/library/compression-algorithm/lzw>
- [5] *Analýza požadavků* [online]. [cit. 2018-04-23]. Wikipedie. Dostupné z: https://cs.wikipedia.org/wiki/Anal%C3%BDza_po%C5%BEdavk%C5%AF
- [6] *Algoritmus LZ77* [online]. [cit. 2018-04-25]. Dostupné z: <http://voho.eu/wiki/algoritmus-lz77/>
- [7] *Algoritmus LZ78* [online]. [cit. 2018-04-28]. Dostupné z: http://stringology.org/DataCompression/lz78/index_cs.html
- [8] *Algoritmus LZW* [online]. [cit. 2018-04-28]. Dostupné z: http://www.stringology.org/DataCompression/lzw-e/index_cs.html

Seznam použitých zkratek

SCT Small compression toolkit

LZ Lempel-ziv

Obsah přiloženého CD

	readme.txt.....	stručný popis obsahu CD
	exe	adresář se spustitelnou formou implementace
	src	
	impl.....	zdrojové kódy implementace
	thesis	zdrojová forma práce ve formátu L ^A T _E X
	text	text práce
	thesis.pdf	text práce ve formátu PDF
	thesis.ps	text práce ve formátu PS