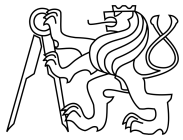
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Automatová knihovna - Komprese dat
Student:	Jan Parma
Vedoucí:	Ing. Tomáš Pecka
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Seznamte se se současnou architekturou a implementací Automatové knihovny [1].

Nastudujte Huffmanovo kódování a kompresní algoritmy LZ77 a LZ78 [2]. Tyto kompresní metody implementujte do Automatové knihovny.

Analyzujte současné řešení testování Automatové knihovny, které je prováděno pomocí shellových skriptů. Navrhněte nové řešení.

Otestujte vaší implementaci pomocí vhodně navržených testů.

Seznam odborné literatury

[1] ŽÁK, Martin: *Automatová knihovna – vnitřní a komunikační formát.* Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2014.[2] SALOMON, David. Data compression: the complete reference. 4th ed. London: Springer, 2007. ISBN 978-1-84628-603-2.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 6. ledna 2017

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

Bakalářská práce

Automatová knihovna – Komprese dat

Jan Parma

Vedoucí práce: Ing. Tomáš Pecka

15. května 2017

Poděkování

Děkuji Ing. Tomáši Peckovi za cenné rady, ochotu konzultovat a hlavně trpělivost při vedení této bakalářské práce. Dále bych rád poděkoval svým rodičům za podporu během studia.

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 15. května 2017 .....

Odkaz na tuto práci

Parma, Jan. *Automatová knihovna – Komprese dat.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato bakalářská práce se zabývá implementací kompresních algoritmů LZ77, LZ78 a implementací Huffmanova kódování. Dále analyzuje současné řešení testování již existující Automatové knihovny, které je prováděno pomocí shellových skriptů. Zároveň bylo přidáno testování i na nově vytvořené algoritmy.

Kompresní algoritmy jsou naprogramovány v jazyce C++, jelikož dosavadní verze knihovny je v témže jazyce, a testování je nově řešeno ve skriptovacím jazyce Python.

Hlavním výsledkem práce je funkční rozšíření knihovny o již zmíněné algoritmy a metody. Dále prvotní návrh a implementace sjednoceného testovacího programu pro všechny části Automatové knihovny.

Klíčová slova Automatová knihovna, komprese dat, testování, C++, Python

Abstract

This bachelor's thesis deals with the implementation of compression algorithms LZ77, LZ78 and implementation of Huffman coding. It also analyzes the current testing solution for an existing Automata Library, which is performed using shell scripts. At the same time, testing was added to newly created algorithms.

Compression algorithms are written in C++, as the current version of the library is written in the same programming language, and testing is newly solved using the Python language.

The main result of the thesis is the extension of the library with the already mentioned compression algorithms and methods. Also the initial design and implementation of a unified test suite for all parts of the Automata Library.

Keywords Automata library, data compression, testing, C++, Python

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Pracujeme-li s počítačovými daty, které mají velký objem, je pravděpodobné, že brzy bude třeba daná data archivovat nebo je přenést po síti. Vzhledem k faktu, že kapacita disku je omezená, může být proces archivace velkých počítačových dat problém. Současně, při omezené přenosové rychlosti, může proces přenosu dat zabrat velké množství času.

Oba uvedené problémy lze vyřešit díky dokoupení dalších počítačových disků a zvýšit tak celkovou úložnou kapacitu nebo zřízení připojení k síti s větší přenosovou rychlostí. Obě tyto možnosti jsou však jen dočasné řešení a navíc obě tato řešení mohou být poměrně drahá. Dalším a mnohem efektivnějším způsobem, jak archivovat objemově velká počítačová data, je komprimace pomocí kompresních algoritmů.

Automatová knihovna a komprese

Automatová knihovna je sada programů a knihoven pro zpracování automatů, gramatik a regulárních výrazů. S nápadem na vytvoření Automatové knihovny přišel ing. Jan Trávníček [1]. Automatová knihovna, do které přidávám možnosti kompresí dat, umí mnohem více. Umí potřebné věci z látky z předmětů BI-AAG (Automaty a gramatiky), BI-PJP (Programovací jazyky a překladače), BI-GRA (Grafové algoritmy a základy teorie složitosti) a MI-EVY (Efektivní vyhledávání v textech). To znamená, že si Automatová knihovna dokáže poradit například s konverzí jazyků, převodů gramatik, s grafovými algoritmy a podobně.

Pro komplexnější úlohy můžeme jednotlivé operace spojit unixovou rourou. Takovýto přístup je v této oblasti zatím ojedinělý. [3]

[1] Odborný asistent Fakulty informačních technologií, ČVUT v Praze

Tato knihovna zatím nemá možnost použití kompresních algoritmů. Cílem mé bakalářské práce je tyto kompresní algoritmy do Automatové knihovny přidat.

Automatová knihovna a testování

V Automatové knihovně je testování jednotlivých modulů řešeno skripty v shellu. Tyto skripty často obsahují z více než poloviny duplicitní kód. Cílem mé práce je tyto skripty sjednotit a vytvořit testovací program, který pomůže testování zpřehlednit.

Kapitola 1

Komprese dat

Komprese dat je proces, při němž se zpracovávaná počítačová data nahrazují, neboli kódují tak, aby výsledná velikost zpracovávaných počítačových dat byla menší než na začátku. Kompresí dat je tedy možné ušetřit místo na úložném disku a současně urychlit proces přenosu dat po síti. Metody datových kompresí je možné porovnávat podle jejich rychlostí, kvalit a také podle jejich efektivity. Kvalita komprese závisí na druhu použité komprese.

Kompresi počítačových dat je možné rozdělit na dva druhy, a to na kompresi ztrátovou a kompresi bezztrátovou.

1.1 Ztrátová komprese

Při použití ztrátové komprese je určitá část komprimovaných dat vymazána a už není možné je zpětně zrekonstruovat. Díky tomuto faktu dojde k tomu, že počítačová data zkomprimovaná ztrátovou kompresí a výsledná data následně dekomprimovaná, již nejsou totožná. Ztrátová komprese je používána v případech, kde získání malé velikosti komprimovaných dat je přednější a je možné částečnou ztrátu dat tolerovat.

Hlavní použití ztrátové komprese je například u komprimování zvuku, obrazu nebo videa. V těchto případech není lidské oko nebo ucho schopno rozeznat úbytek dat způsobený ztrátovou kompresí a je možné ho proto tolerovat. Např. lidské oko má omezenou rozlišovací schopnost, jak ve vztahu k barevné hloubce, tak i k obrysovým detailům. Pokud body leží dostatečně blízko sebe, tak oko jejich barvy průměruje. Jeden z nejčastěji používaných formátů v grafice je komprimovaný formát JPG, kde se využívá algoritmus diskrétní kosinové transformace (DCT).

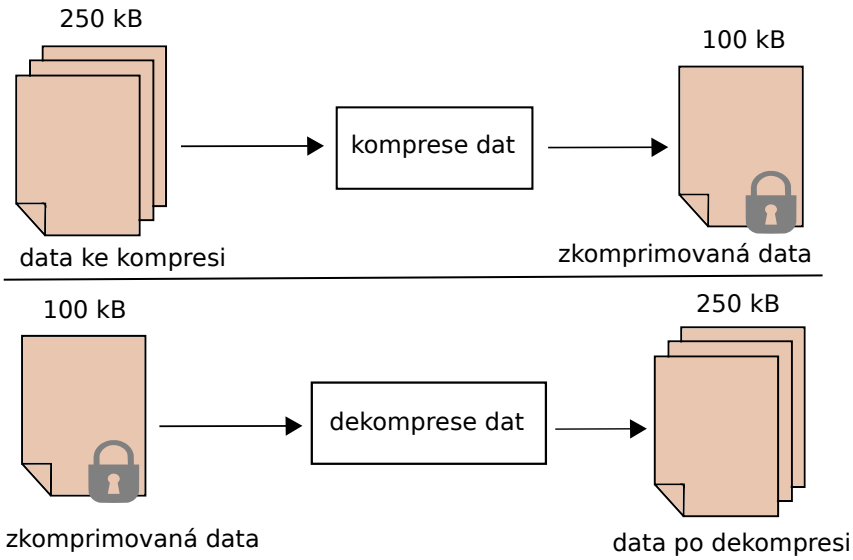
Při využívání ztrátové komprese dat se dá často zvolit účinnost této komprese. Tím se ovlivňuje kvalita obrazu po kompresi. Čím větší účinnost

komprese, tím horší kvalita obrazu je. Pro kompresi videosekvencí se nejčastěji používá formát MPEG, popřípadě MPEG-1 a MPEG-2. Efektivnější metoda, než použití formátu MPEG-2, je použití standardu pro kompresi videa H.264, který je ekvivalentní k MPEG-4 part 10, též zvané MPEG-4 AVC. Popřípadě použití standardu H.265. Pro kompresi zvuku používáme nejčastěji formáty WMA, MP3, AAC a OGG. [1]

1.2 Bezztrátová komprese

Bezztrátová komprese umožňuje komprimovaná data opět rekonstruovat opačným postupem do přesného původního stavu. Tento druh komprese dat není tak účinný jako komprese ztrátová, ale je výhodná v případech, kde ztráta dat není žádoucí. Princip bezztrátové komprese viz Obrázek 1.1 Bezztrátová komprese dat využívá faktu, že některá data se v souborech opakují. Ta jsou pak například nahrazena odkazy na stejná předchozí slova.

Bezztrátová komprese počítačových dat je využívána například při kompresi textů a při přenosu komprimovaných počítačových dat. V těchto případech není možné tolerovat sebemenší ztrátu dat. Nejznámější a nejpoužívanější programy pro bezztrátovou kompresi jsou WinRAR, GZip a podobně. Program WinRAR například používá ke kompresi predikci částečné shody spolu s algoritmem LZ77 a Huffmanovým kódováním. GZip využívá pouze algoritmu LZ77. [1]



Obrázek 1.1: Princip bezztrátové komprese [1]

Kapitola 2

Kompresní algoritmy

2.1 Vymezení pojmů

Dříve než se pustíme do problematiky datové komprese a kompresních algoritmů, je nutné si nejdříve definovat některé pojmy, které v této práci budou často zmiňovány.

Abeceda je neprázdná konečná množina symbolů.

Řetězec (neboli slovo nad abecedou), je posloupnost symbolů z abecedy.

Délka řetězce je počet znaků, které řetězec obsahuje.

Prázdný řetězec neobsahuje žádný znak, neboli je jeho délka rovna 0.

Kompresní poměr je definován jako poměr délek komprimovaného řetězce a původního řetězce. Tedy:

$$\text{Kompresní poměr} = \frac{\text{Délka komprimovaného řetězce}}{\text{Délka původního řetězce}}$$

Entropie je množství informace, které zpráva obsahuje. Entropie zprávy se měří průměrným počtem bitů, které jsou nezbytné k zakódování této zprávy při optimálním kódování. Optimálním kódováním se rozumí kódování, které použije nejmenší možný počet bitů k zakódování této zprávy. Entropie zprávy ze zdroje se spočítá jako:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

kde $p_i, ..., p_n$ jsou pravděpodobnosti všech zpráv $X_1, ..., X_n$ zdroje X a $-p_i \log_2 p_i$ je počet bitů potřebných k optimálnímu zakódování zprávy X_i [4].

Redundance neboli nadbytečnost jazyka vzhledem k jednomu symbolu, nám vyjadřuje kolik bitů je v tomto symbolu daného jazyka nadbytečných [4].

Průměrná velikost kódu znamená, kolik v průměru zabere daný kód bitů.

Buffer neboli vyrovnávací paměť, je oblast fyzické paměti sloužící k dočasnému ukládání dat během jejich přesunu z jednoho místa na druhé.

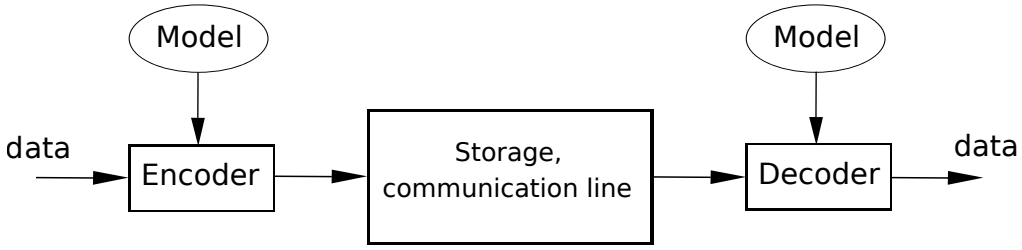
2.2 Kompresní model

Kompresní model jsou nějaké další informace použité při běhu komprese a dekomprese. Mohla by to být například tabulka, která přiřazuje určitý kód ke každému symbolu vstupní abecedy. Kompresní model můžeme rozdělit do tří kategorií, a to statický model, semi-adaptivní model a model adaptivní.[5]

2.2.1 Statický model

Metody používající statický model používají stejný model při datové kompresi a dekompresi. Tento model není závislý na vstupních datech a nemusí být přidáván ke komprimovaným datům. Kompresní metody využívající statický model dat obvykle nedávají nejlepší kompresní poměr z důvodů nedostatku schopnosti přizpůsobit se datům, která jsou právě komprimována.

Tento model může být však užitečný v případě, kde máme jen jeden známý typ dat ke zpracování. Statický model dat využívá například algoritmus ITU-T T4, který je využíván ke kompresi bitmapových obrázků, které jsou posílány faxem. [5]



Obrázek 2.1: Schéma statického modelu. [2]

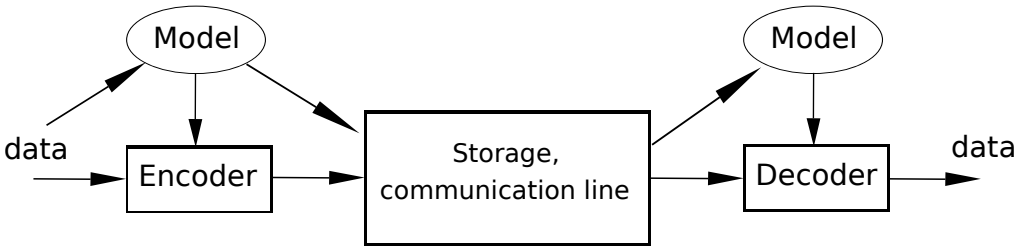
2.2.2 Semi-adaptivní model

Metody využívající semi-adaptivního modelu provádí kompresi dat ve dvou průchodech. Nejprve analyzují vstupní data a vytváří model v prvním průchodu. V druhém průchodu pak komprimují data s využitím již vytvořeného modelu.

Komprimovaná data jsou obvykle velmi malá, jelikož byla komprese dobře přizpůsobena vstupním datům. Výsledný kompresní poměr ale nemusí být tak dobrý, jelikož výsledný model musí být přidán na výstup společně s komprimovanými daty, jinak by zpětná dekomprese nebyla možná.

Také komprese dat za pomocí semi-adaptivního modelu zabere více času, jelikož se vstupní data musí procházet ve dvou krocích.

Semi-adaptivní model využívají například algoritmy Shannon-Fanova kódování a nebo Huffmanova kódování. [5]

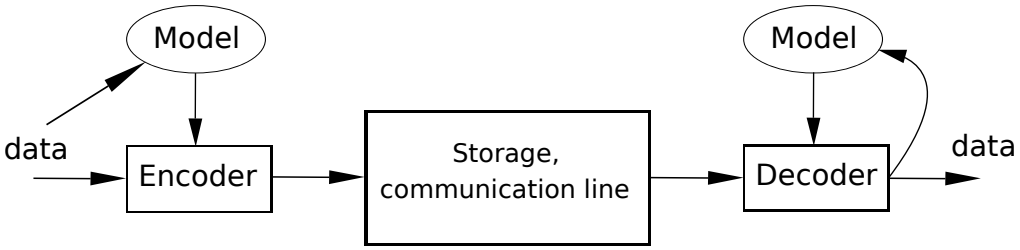


Obrázek 2.2: Schéma semi-adaptivního modelu. [2]

2.2.3 Adaptivní model

Metody využívající adaptivní model vytváří a mění model během komprese a dekomprese. Výhodou adaptivního modelu, oproti semi-adaptivnímu modelu, je možnost rekonstrukce modelu z komprimovaných dat během dekomprese. Model tedy nemusí být přidáván k výsledným komprimovaným datům.

Metody využívající adaptivní model jsou obvykle rychlejší oproti metodám se semi-adaptivním modelem, jelikož se vstupní data mohou procházet pouze jedenkrát. Adaptivní model využívají například algoritmy LZ77 nebo LZ78. [5]



Obrázek 2.3: Schéma adaptivního modelu. [2]

2.3 Statistické metody

Statistické kompresní metody používají statistické modely dat, kde kvalita komprese je závislá na kvalitě daného modelu. Tyto metody přiřazují variabilní velikost kódu symbolu, který je právě kódován. Kódovaným symbolem může být například znak nebo pixel.

Kódy s kratší velikostí jsou přiřazovány symbolům, které se v datech objevují častěji (mají větší pravděpodobnost výskytu). V tomto případě je zapotřebí se vypořádat se s dvěma problémy. Prvním problémem je nutnost každý kód dekódovat jednoznačně. Jsou-li data zakódována vybraným algoritmem, musí po dekódování být získána data původní. Druhým problémem je potřeba přiřazovat kódy k symbolům tak, aby měli minimální průměrnou velikost. Čím je průměrná velikost menší, tím je komprese efektivnější.

Telegrafní kódování, vymyšlené Samuelem Morsem, je příkladem prvního kódování. První verze, která vyšla v roce 1832, byla mnohem rozsáhlejší a komplexnější než verze další, vydaná v roce 1843. V první verzi bylo kódové číslo přiřazováno celému kódovanému slovu, tato slova pak byla dekódována pomocí vydaného slovníku. V novější verzi je přiřazováno kódové číslo pouze konkrétnímu symbolu. Například symbolu *A* je přiřazeno *10*, neboli „.-“. Tato verze telegrafního kódování je známá dodnes.

Známé algoritmy, které využívají statistické kompresní metody, jsou například Shannon-Fanovo kódování nebo Huffmanovo kódování. [6]

2.3.1 Shannon-Fanovo kódování

Shannon-Fanovo kódování získalo své jméno podle jeho tvůrců Claude Shannona a Roberta Fana. Využívá statistické metody komprese dat a současně se jedná o bezztrátovou kompresní metodu. Shannon-Fanovo kódování bylo prvním algoritmem, díky kterému bylo možné sestavit sadu nejlepších kódů s proměnnou velikostí.

2.3.1.1 Komprese

Algoritmus začíná sadou symbolů na vstupu společně s jejich pravděpodobnostmi výskytu, podle kterých jsou symboly seřazeny v sestupném pořadí. Následně je sada symbolů rozdělena na dvě části, které mají stejnou nebo alespoň přibližnou pravděpodobnost výskytu. Všem symbolům, které se vyskytují v první oddělené části, je přiřazený kód 0, zatímco druhé části je přiřazený kód 1. Každá z těchto částí je opět rekurzivně dělena na další dvě části se stejnou nebo podobnou pravděpodobností. Druhý bit kódu je určen

stejnou metodou, jako v předchozím kroku. Tento proces je prováděn tak dlouho, dokud nezůstane žádná další podmnožina. [6]

Výsledkem Shannon-Fanova kódování a i Huffmanova kódování jsou prefixové kódy. To znamená, že žádný kód nemůže být předponou jiného kódu. Kdyby to tak nebylo, výsledný kód by nešlo jednoznačně dekódovat na původní slovo, aniž bychom použili oddělovače mezi jednotlivé kódy. Pokud bychom již měli uložené kódy a k nim přiřazené symboly jako například v tabulce 2.1 a přišel by nám vstupní řetězec *1010*, neuměli bychom ho jednoznačně dekódovat. Vstupní kód by mohl znamenat buď řetězec *baba*, řetězec *bac*, řetězec *cba*, nebo řetězec *cc*. Proto se pro Shannon-Fanovo kódování a Huffmanovo kódování používají prefixové kódy.

Tabulka 2.1: Špatný příklad Shannon-Fanova kódu.

Kód	**Symbol**
0	a
1	b
10	c

Algoritmus 1: Pseudokód Shannon-Fanova kódování [7].

Input : řetězec určený ke kompresi
Output: Shannon-Fanův strom

```
1  begin
2      spočti počet prvků
3      seřaď prvky v vzestupném pořadí
4      SF-Split(S)
5      výstup(počet symbolů, kódový strom, symboly)
6      zapiš výstup
7  end
8  Procedure SF-Split(S)
9  if /S/>1 then
10     rozděl S na S1 a S2 se stejným počtem prvků
11     přidej 1 do kódu S1
12     přidej 0 do kódu S2
13     SF-Split(S1)
14     SF-Split(S2)
15 end
```

	Znak	Pravděpodobnost	Kroky				Výsledný kód
1	a	$\frac{1}{23}$	0	0	0	0	0000
2	b	$\frac{1}{23}$	0	0	0	1	0001
3	c	$\frac{1}{23}$	0	0	1	0	0010
4	d	$\frac{1}{23}$	0	0	1	1	0011
5	e	$\frac{3}{23}$	0	1	0		010
6	f	$\frac{3}{23}$	0	1	1		011
7	g	$\frac{3}{23}$	1	0	0		100
8	h	$\frac{3}{23}$	1	0	1		101
9	i	$\frac{7}{23}$	1	1			11

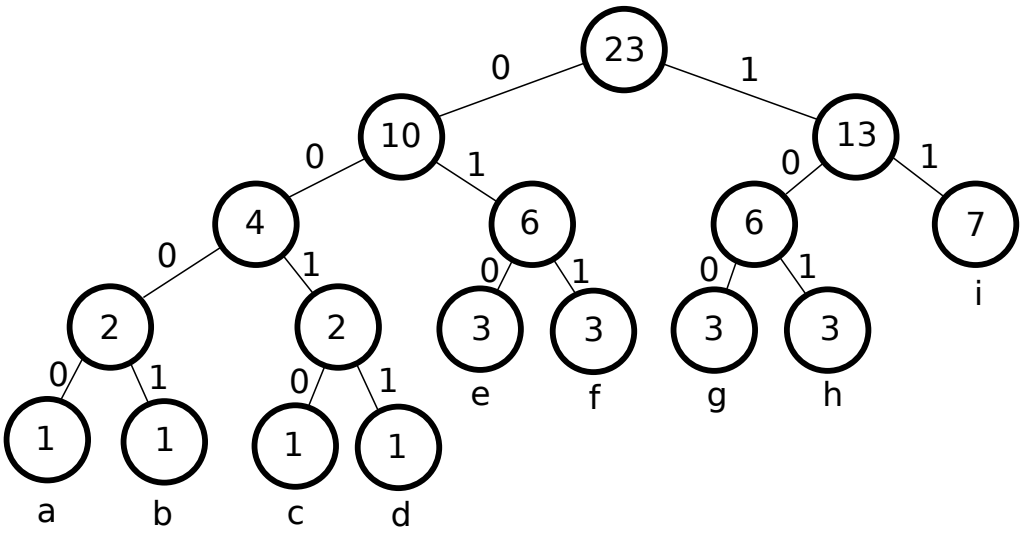
Obrázek 2.4: Průběh algoritmu Shannon-Fanova kódování.

Průběh algoritmu Shannon-Fanova kódování si ukážeme na příkladu. Předpokládáme vstupní řetězec: *abcdeeefffggghhhiiiiiii*. V prvním průchodu vstupními daty si algoritmus nejprve spočte jednotlivé pravděpodobnosti výskytu všech symbolů. Místo pravděpodobností můžeme taktéž spočítat frekvence jednotlivých symbolů, výsledek se tím nezmění.

Pro symboly *a, b ,c ,d* nám vyjde pravděpodobnost výskytu $\frac{1}{23}$. Pro symboly *e, f ,g ,h* máme pravděpodobnost $\frac{3}{23}$ a pro symbol i je výsledná pravděpodobnost výskytu symbolu $\frac{7}{23}$. V druhém průchodu rozdělíme vstupní abecedu na dvě části s přibližně stejným součtem pravděpodobností jednotlivých symbolů. To je v našem případě mezi symbolem f a g. První část řetězce má součet pravděpodobností symbolů $\frac{10}{23}$, těm je přiřazen kód 0. Druhá část má součet pravděpodobností $\frac{13}{23}$, těm je přiřazen kód 1 (viz obrázek 2.4). Takto rekurzivně dělíme řetězce na dvě poloviny s přibližně stejnou pravděpodobností dokud řetězec neobsahuje pouze jeden symbol (viz obrázek 2.5). Sečteme zapsáné kódy ke každému znaku a uložíme.

2.3.1.2 Dekomprese

Pokud chceme komprimovaná data dekomprimovat, musíme na vstupu dostat jednak dekomprimovaná data a jednak i kompresní model. Kompresní



Obrázek 2.5: Vytvořený strom pro vstup „abcdeeefffggghhhiiiiiii“.

Tabulka 2.2: Výsledná tabulka Shannon-Fannových kódů pro vstupní řetězec „abcdeeefffggghhhiiiiiii“.

Kód	**Symbol**
0000	a
0001	b
0010	c
0011	d
010	e
011	f
100	g
101	h
11	i

model může být v našem případě například mapa[2] obsahující Shannon-Fanův kód jako klíč a k němu příslušný symbol (viz tabulka 2.2). Dekomprese poté probíhá tak, že čteme komprimovaný řetězec postupně a ten hledáme v naší mapě. Pokud jej nalezneme, na výstup je zapsán symbol reprezentující dosud přečtený komprimovaný podřetězec. Pokud jej nenalezneme, přečteme další symbol a ten přidáme k dosud přečteným. Prvních 8 číslic komprimovaného řetězce *abcdeeefffggghhhiiiiiii* by bylo: *00000001*. Pro jednoduchost zde uvádím pouze prvních 8 bitů, ve skutečnosti by kom-

[2]Mapa je seřazený asociativní kontejner obsahující páry - klíč, hodnota, kde klíč je v mapě unikátní.

primovaný řetězec byl 66 bitů dlouhý.

Dekompresní algoritmus si načte první číslo, to je *0* a začne s prohledáváním mapy. Jelikož žádný symbol nevlastní klíč *0*, algoritmus přečte další číslo a přidá již k přečtenému. Nyní algoritmus hledá klíč *00* v naší mapě. Ten opět není nalezen. Situace se opakuje dokud algoritmus nepřečte sekvenci *0000*, ta přísluší symbolu *a*. Tento symbol zapíše na výstup, zahodí dosud přečtené znaky a hledá znovu. Opět načte *0* a hledá ji. poté *00* , *000* a *0001*. Zapíše na výstup symbol *b* a pokračuje dále. [6]

Algoritmus 2: Pseudokód dekomprese Shannon-Fanova kódování.

Input : tabulka reprezentující znak a přiřazený kód, výsledný Shannon-Fanův kód
Output: dekomprimovaný řetězec

```
1 vytvoř pomocný řetězec tmp
2 for i = 0; i < velikost Shannon-Fanova kódu; i++ do
3 |   přidej do tmp Shannon-Fanův kód[i]
4 |   if tmp je v tabulce then
5 |   |   přidej na výstup znak patřící klíči tmp
6 |   |   vymaž obsah tmp
7 |   end
8 end
```

2.3.1.3 Implementace

V mé implementaci je vstup realizován jako lineární string obsahující datové typy *SymbolType* [3]. Vstup je u všech algoritmů stejný, tak jej nebudu dále uvádět. Na výstup je pak posílána dvojice, která obsahuje vektor typu *bool*, reprezentující výsledný Shannon-Fanův kód, a mapa obsahující *SymbolType* jako klíč a k němu přiřazený Shannon-Fanův kód (opět vektor typu *bool*). Pro sestavení Shannon-Fanova stromu jsem si nejprve vytvořil strukturu s názvem *ShannonFanoTreeNode* obsahující váhu (*unsigned int*), hodnotu (*SymbolType*) a odkazy na levého a pravého potomka stromu.

Nejprve si spočítám jednotlivé četnosti znaků a ty k nim přiřadím. Poté si spočítám součet všech četností výskytů jednotlivých znaků a naleznu místo, kde se dá vstup rozdělit na dvě části s přibližně stejnou četností výskytu. Přitom četnosti výskytů jednotlivých podřetězců ukládám do struktur, abych vytvořil Shannon-Fanův strom. Výsledné podřetězce znovu

[3] *SymbolType* je datový typ Automatové knihovny, který reprezentuje symbol.

rekurzivně dělím, dokud nemá podřetězec délku 1. V tuto chvíli je strom kompletní a s jeho pomocí uložím do mapy postupně symboly a jejich kódy. Za pomocí mapy vytvořím výsledný Shannon-Fanův kód, který pak s mapou pošlu na výstup.

2.3.2 Huffmanovo kódování

Huffmanovo kódování bylo vymyšleno a popsáno v roce 1952 Davidem Albertem Huffmanem. Jedná se o velice populární a často používanou metodu datové komprese. Některé programy používají pro datovou kompresi pouze Huffmanovo kódování a jiné ho používají jen jako jeden z mnoha dalších kroků při kompresním procesu (například WinZip).

2.3.2.1 Komprese

Algoritmus začíná sestavením seznamu všech použitých symbolů ve vzestupném pořadí podle jejich pravděpodobností výskytů. Poté následuje konstrukce samotného stromu od spodních listů směrem nahoru. Listy stromu reprezentují použité symboly. Sestavení stromu je prováděno v krocích. V každém kroku algoritmus vybere dva symboly s nejmenší pravděpodobností, přidá je do horní části dílčího stromu, vymaže je ze seznamu a nahradí je symbolem reprezentující oba původní symboly. Poté, co v seznamu zůstane již jen jeden symbol, strom je kompletní.

Pokud je potřeba kódovému symbolu přiřadit Huffmanův kód, je nutné projít sestavený strom od kořene až po list reprezentující daný symbol. Číslo nula je zapsáno v případě, že je po cestě k listu přistupováno přes levého potomka. V případě, že je přistupováno přes pravého potomka, je zapsáno číslo jedna. Huffmanův kód je poté posloupnost těchto daných čísel, čtená od shora dolů.

Huffmanovo kódování je podobné Shannon-Fanově kódování. Obě tyto metody dávají nejlepší výsledky v případě, že pravděpodobnosti symbolů jsou záporné mocniny dvou. Hlavní rozdíl mezi Huffmanovým a Shannon-Fanovým kódováním spočívá v rozdílném sestavování kódových stromů. Huffmanova metoda sestavuje kódový strom odspodu nahoru, kdežto Shannon-Fanova metoda odshora dolu. [6]

Průběh Huffmanova algoritmu si ukážeme na příkladu. Vezmeme například vstupní řetězec „abcddeeffffffgggggg“. Frekvence a pravděpodobnost výskytu jednotlivých symbolů je zapsána v tabulce 2.3.

Nyní začneme s vytvořením Huffmanova stromu. Vezmeme si jednotlivé symboly, ty zapíšeme jako listy vytvářeného Huffmanova stromu a k nim si zapíšeme jejich příslušné frekvence výskytu. Nyní algoritmus pokračuje tak,

Algoritmus 3: Pseudokód Huffmanova kódovacího algoritmu [2].

Input : data určená ke kompresi
Output: Huffmanův strom

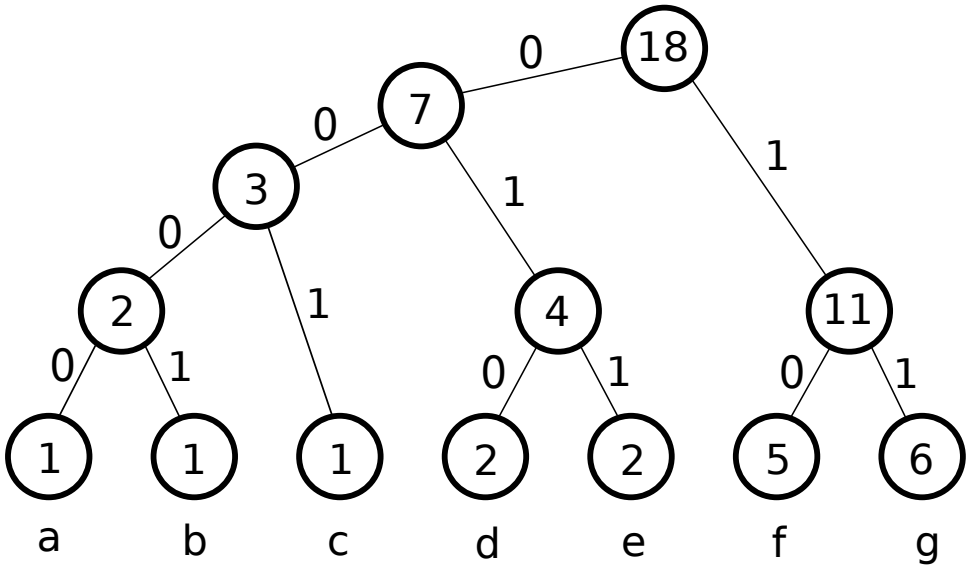
```
1  spočítání četností jednotlivých symbolů
2  for ∀x ∈ S do
3  |  newl ← (x, p(x), NULL, NULL)
4  |  Vlož(L, newl)                    ▷ Vlož tento uzel do seřazeného listu L
5  end
6  j ← n;
7  while j ≥ 2 do
8  |  l1 ← Vyjmi(L); l2 ← Vyjmi(L)     ▷ Vyjmi první dva uzly z L
9  |  newl ← (ϵ, p(l1) + p(l2), l1, l2)   ▷ Vytvoř nový uzel...
10 |  Vlož(L, newl)                    ▷ ...a vlož do listu L
11 |  j ← j − 1
12 end
```

Tabulka 2.3: Tabulka symbolů pro vstup „abcddeeffffffgggggg“.

Symbol	**Frekvence**	**Pravděpodobnost**
a	1	$\frac{1}{18}$
b	1	$\frac{1}{18}$
c	1	$\frac{1}{18}$
d	2	$\frac{2}{18}$
e	2	$\frac{2}{18}$
f	5	$\frac{5}{18}$
g	6	$\frac{6}{18}$

že si najde dva uzly s nejmenší frekvencí výskytu a jim přiřadí jednoho rodiče, který bude mít výslednou frekvenci součtu původních frekvencí. V našem případě je to například list reprezentující symbol „a“ s frekvencí 1 a list reprezentující symbol „b“ s frekvencí 1. Výsledný rodič těchto symbolů má tedy frekvenci 2. Algoritmus tyto dva použité uzly označí za již zpracované a

dále je již nezpracovává. Přidá však do seznamu pro zpracování výsledného rodiče. Takto algoritmus pokračuje dokud algoritmu zbyde pouze jeden jediný uzel. Tento uzel bude kořenovým uzlem Huffmanova stromu. V našem případě bude mít frekvenci rovnou součtu všech frekvencí listů, tedy 18 (viz Obrázek 2.6).



Obrázek 2.6: Výsledný Huffmanův strom po vstupu „abcddeeffffffgggggg“.

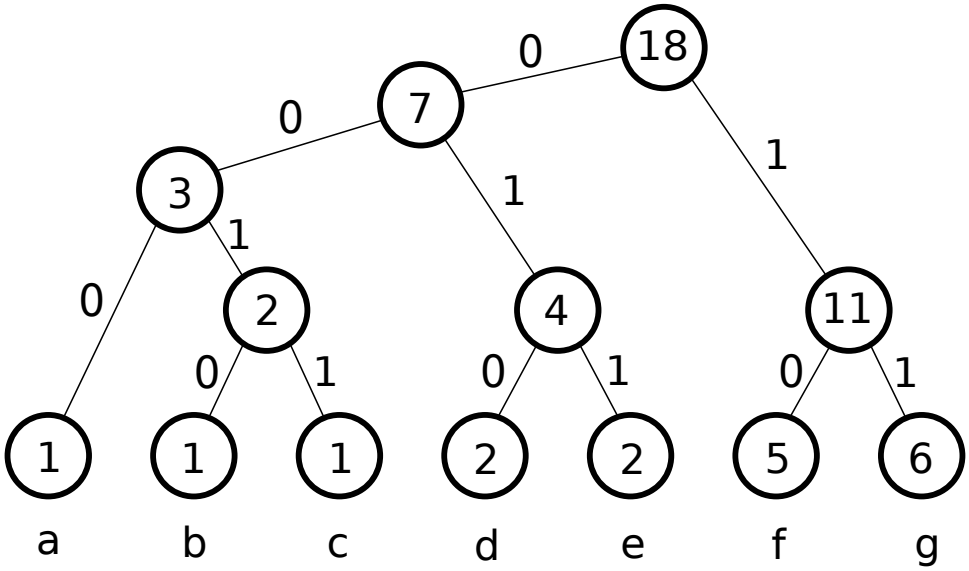
Dále algoritmus přiřadí Huffmanův kód pro každý z listů stromu. Začínáme u kořenového uzlu a pokud jsme přistoupili blíže k listu přes levého potomka uzlu, zapíšeme si číslo 0, pokud přes pravého, zapíšeme číslo 1. Například symbol „a“ bude mít Huffmanův kód 0000 (viz Tabulka 2.4).

Tabulka 2.4: Tabulka Huffmanových kódů pro vstup „abcddeeffffffgggggg“.

Kód	**Symbol**
0000	a
0001	b
001	c
010	d
011	e
10	f
11	g

Průměrná velikost kódu je: $\frac{1}{18} \times 4 + \frac{1}{18} \times 4 + \frac{1}{18} \times 3 + \frac{2}{18} \times 3 + \frac{2}{18} \times 3 + \frac{5}{18} \times 2 + \frac{6}{18} \times 2 = 2.5$ bitů na symbol.

Co je však důležité zmínit je, že Huffmanův kód není unikátní. Některé z výše uvedených kroků byly zvoleny libovolně, jelikož jsme měli více než dva symboly s nejmenší frekvencí. Pokud bychom zvolili jiné dva minimální uzly k výběru, některé symboly by měly ve výsledku různé Huffmanovy kódy. Výsledný poměr bitů na symbol by však měl zůstat zachován. [6]



Obrázek 2.7: Výsledný Huffmanův strom po vstupu „abcddeeffffffgggggg“ s jiným výběrem uzlů.

Na obrázku 2.7 jsme vybrali jiné dva uzly s minimální frekvencí výskytů. Vidíme, že například symbol „a“ má nyní Huffmanův kód 000 místo původního 0000. Výsledný poměr bitů na symbol však zůstal stejný (viz níže).

Průměrná velikost kódu pro jiný strom je: $\frac{1}{18} \times 3 + \frac{1}{18} \times 4 + \frac{1}{18} \times 4 + \frac{2}{18} \times 3 + \frac{2}{18} \times 3 + \frac{5}{18} \times 2 + \frac{6}{18} \times 2 = 2.5$ bitů na symbol.

2.3.2.2 Dekomprese

Zpětná dekomprese funguje stejně jako v případe Shannon-Fanova kódování. Opět musíme na vstup dostat komprimovaná data a model, například formou tabulky. Algoritmus opět čte komprimovaná data, ta hledá v tabulce a zapisuje na výstup příslušné symboly. [6]

2.3.2.3 Implementace

Výstup je v mé implementaci realizován stejně jako u Shannon-Fanova kódování. Struktura jednotlivých uzlů stromu s názvem *HuffmanTreeNode* je

Algoritmus 4: Pseudokód dekomprese Huffmanova kódování.

Input : tabulka reprezentující znak a přiřazený kód, výsledný Huffmanův kód
Output: dekomprimovaný řetězec

```
1 vytvoř pomocný řetězec tmp
2 for i = 0; i < velikost Huffmanova kódu; i++ do
3 |   přidej do tmp Huffmanův kód[i]
4 |   if tmp je v tabulce then
5 |   |   přidej na výstup znak patřící klíči tmp
6 |   |   vymaž obsah tmp
7 |   end
8 end
```

též realizována stejně jako u Shannon-Fanova kódování. Ze vstupu si taktéž spočítám četnosti výskytů jednotlivých znaků, které pak k nim přiřadím. Tyto znaky vkládám do prioritní fronty, která je řadí v pořadí od nejmenšího po největší (řazeno dle jejich četností výskytů). Ve *while* cyklu procházím prioritní frontu. Během běhu cyklu odeberu dva nejmenší prvky, sečtu jejich četnosti a výsledné číslo uložím do nově vytvořeného uzlu. Zároveň tomuto uzlu uložím naše dva odebrané uzly jako levý a pravý potomek. Výsledný uzel znovu vložím do prioritní fronty. Cyklus jede do té doby, dokud prioritní fronta neobsahuje právě jeden prvek. Tento prvek je kořenem Huffmanova stromu.

Stejně jako u Shannon-Fanova kódování si vytvořím mapu reprezentující symbol jako klíč a k němu odpovídající Huffmanův kód. Tuto mapu za pomocí stromu naplním daty, díky kterým pak vytvořím výsledný Huffmanův kód. Tento výsledný kód společně s mapou pak posílám na výstup.

2.4 Slovníkové metody

Slovníkově založené kompresní metody nepoužívají statistické modely dat, ani variabilní velikost kódů. Místo toho používají řetězce symbolů a tento řetězec kódují pomocí slovníků. Slovník obsahuje seznam řetězců symbolů. V případě, že se konkrétní řetězec ve slovníku nenalézá, je možné jej do slovníku přidat.

Kompresní algoritmus, využívající slovníkové metody, dokáže předložený řetězec délky n symbolů v principu zkomprimovat na velikost n * H bitů, kde H je entropie řetězce. Slovníkové metody jsou postaveny na kódování

entropie, ale pouze pokud je vstupní soubor určený ke kompresi dostatečně velký. Tato metoda je vhodná jak pro kódování obrázků a audio záznamů, tak i pro pouhý text.

Princip slovníkové metody lze vysvětlit za pomoci jednoduchého příkladu, kde slovník používaný algoritmem nahradíme například seznamem českých slov pro kódování textu v českém jazyce. Pokud se kódované slovo textu ve slovníku (seznamu českých slov) nachází, na výstup je zapsán pouze index odkazující do slovníku (seznam), ve kterém je dané kódové slovo. Pokud takové slovo ve slovníku chybí, je následně přidáno a je mu přiřazen příslušný index. Přiřazený index je volen tak, aby byl jedinečný.

Výsledkem je, že výstupní proud obsahuje indexy do slovníku a slova samotná. Je důležité je rozlišovat mezi sebou. Jeden způsob, jak toho dosáhnout, je rezervovat si extra jeden bit v každé zapsané položce. Máme-li index o velikosti 19 bitů, tak tento index je dostatečný pro specifikaci položky ve slovníku o velikosti $2^{19} = 524\ 288$ slov. Nyní budeme mít dvacetibitový index, u kterého první byt určuje zda byla nebo nebyla nalezena shoda. Pokud shoda byla nalezena, zapíšeme například 0, a dále dalších 19 bitů značí index. Pokud shoda nalezena nebyla, zapíšeme například 1, poté následuje velikost neshodného slova a v poslední řadě slovo samotné. Pokud shoda nalezena nebyla, tak číslo může být větší než 20 bitů. Velikost omezující výsledné číslo na 19 bitů je pouze v případě shodného slova.

Například si vezměme slovo *bet*, které je ve slovníku na pozici 1025. Slovo je zakódováno jako dvacetibitový index: *0| 00000001000000001*. Další slovo si vezměme například *xet*, které ve slovníku nalezeno nebylo. Toto slovo je zakódováno jako: *1| 0000011| 01111000| 01100101| 01110100*. Toto je 4-bajtové číslo kde 7-bitový blok *0000011* určuje kolik následuje ještě dalších bajtů.

Za předpokladu, že je velikost slova zapsána jako 7-bitové číslo a průměrná velikost slova je pět symbolů, tak nekomprimované slovo zabírá v průměru šest bajtů (= 48 bitů) ve výstupním proudu. Komprese 48 bitů na 20 je vynikající, za předpokladu, že se tato komprese děje dost často. Musíme ale zodpovědět otázku, kolik shod potřebujeme mít, abychom měli celkovou kompresi? Označíme si pravděpodobnost shody (případ kdy je slovo nalezeno ve slovníku) jako **P**. Po přečtení a kompresi **N** slov, velikost výstupního proudu bude N[20P+48(1-P)] = N[48-28P] bitů. Velikost vstupního proudu (za předpokladu že slovo má pět symbolů) 40N bitů. Komprese dosáhneme pokud: N[48-28P] $<$ 40N, což znamená že P $>$ 0.29. Z toho vyplývá že musíme mít shodu minimálně v 29 % případů, abychom dosáhli komprese. [6]

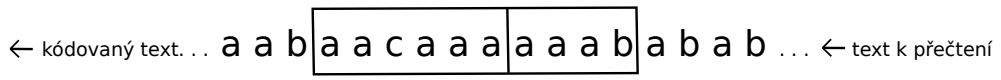
2.4.1 LZ77

LZ77 je metoda bezztrátové komprese dat publikována roku 1977 Abrahamem Lempelem a Jacobem Zivem. Princip této metody (často nazývané jako LZ1) je použití části již přečteného vstupního proudu jako slovníku. Algoritmus si udržuje okno do vstupního proudu, které prochází zprava doleva, jak jsou řetězce kódovány.

Tento kompresní algoritmus je založen na principu „posuvného okna“ (neboli *sliding window*). Posuvné okno je rozděleno na dvě části. Prví část, která se nachází vlevo, se nazývá „vyhledávací pole“ (neboli *search buffer*). Toto je současný slovník a ten obsahuje symboly, které byly nedávno zpracovány a kódovány. Druhá část posuvného okna, nacházející se vpravo, se nazývá „dopředu se dívající pole“ (neboli *look-ahead buffer*), obsahující vstup, který teprve bude kódován. V praktické implementaci bývá search-buffer několik tisíc bajtů dlouhý, kdežto look-ahead buffer bývá v řádu několik desítek bajtů dlouhý. [6]

2.4.1.1 Komprese

Vezmeme si příklad vstupního řetězce *aabaacaaaaaababab*. Pro jednoduchost zvolíme velikost search bufferu šest znaků a velikost look-ahead bufferu čtyři znaky. Vertikální pruh v ukázce níže mezi devátým a desátým symbolem reprezentuje současné rozdělení mezi naším search bufferem a look-ahead bufferem. Předpokládáme, že text *aabaacaaa* již byl komprimován, kdežto text *aaababab* ještě zkomprimován nebyl (viz Obrázek 2.8).



Obrázek 2.8: Náznak průchodu posuvného okna.

Algoritmus projíždí slovník (search-buffer) pozpátku, zprava doleva, hledající shodu s prvním podřetězcem „*a*“ v look-ahead bufferu. Jediná podmínka je, že hledaný podřetězec musí začínat v search bufferu, nicméně může libovolně přecházet z search bufferu do look-ahead bufferu. Algoritmus najde podřetězec „*a*“ hned na první pozici search bufferu (čteno zprava doleva). Poté algoritmus přidá další symbol, který je za ním, k hledanému řetězci (opět „*a*“) a stejně tak i k nalezenému řetězci. Algoritmus zjistí zdali se tento podřetězec „*aa*“ opět shoduje s tímto řetězcem. Takto algoritmus pokračuje dál, dokud nenarazí na neshodu, nebo na konec look-ahead bufferu.

Nyní si algoritmus poznamená délku shodného řetězce a pokračuje dále v prohledávání search-bufferu, zda-li se tam nevyskytuje jiný, delší shodný řetězec. V našem případě se zde vyskytují tři nejdelší shodné řetězce o délce třech symbolů. Algoritmus vybere řetězec s nejdelší shodou, a nebo, pokud je více stejně dlouhých shod, ten který nalezl jako první. Pokud tedy byl nalezen další podřetězec se stejnou délkou shody jako v předchozím kroku, je tato shoda ignorována a algoritmus pokračuje dál.

Na výstup je pak poslána trojice, v našem případě (1,3,„*b*“). První z čísel trojice udává pozici shodného podřetězce v search-bufferu (opět zprava doleva). Druhé číslo pak udává délku shody dvou řetězců. Třetí znak je znak, který se vyskytuje hned za kódovaným řetězcem. Pokud algoritmus žádnou shodu nenalezne, zapíše trojici (0, 0, *symbol*), kde symbol značí nenalezený symbol v search bufferu. Výstupem algoritmu LZ77 je pak posloupnost těchto trojic.

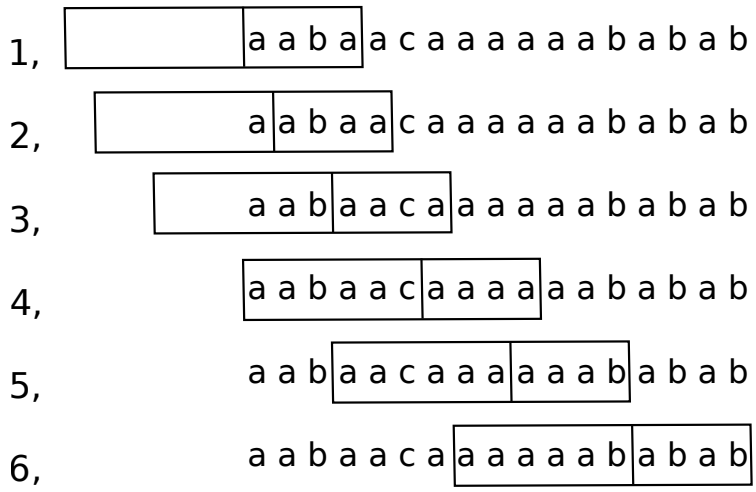
Algoritmus 5: Pseudokód algoritmu LZ77 [7].

Input : data určená ke kompresi
Output: seznam trojic (i, j, k)

```
1 naplnit výhled ze vstupu
2 while (výhled není prázdný) do
3 |   najít nejdelší předponu p z výhledu začínající v kódované části
4 |   i := pozice p v okně
5 |   j := délka p
6 |   x := první symbol za p ve výhledu
7 |   výstup(i, j, x)
8 |   přidat j + 1 symbolů do výhledu
9 end
```

Vybrání první shody má jednu výhodu a to, že první číslo, z výsledných trojic, je nejmenší. Nemusí se to zdát jako výhoda, jelikož bychom měli mít dost místa k uložení větších číslic. Nicméně, je možné spojit algoritmus LZ77 s Huffmanovým kódováním, nebo jiným algoritmem využívající statistické metody, kde menším číslům je přiřazován kratší kód. Tato metoda, kterou navrhl Bernd Herd, je nazývána LZH. V této metodě platí, že pokud máme hodně malých čísel, tak dosahujeme lepších kompresních výsledků, jelikož kódujeme i výstupní trojice Huffmanovým kódováním. [6]

Označíme-li si délku search bufferu S, délku look-ahead bufferu L a výstupní trojici (i, j, a) pak:
i = $\lceil\ log_2$ S $\rceil$ bitů



Obrázek 2.9: Postup algoritmu LZ77 nad vstupem „aabaacaaaaaababab“.

Tabulka 2.5: Výstup algoritmu LZ77 nad vstupem „aabaacaaaaaababab“.

Pořadí	**Výstup**
1	(0, 0, „a“)
2	(1, 1, „b“)
3	(3, 2, „c“)
4	(3, 2, „a“)
5	(1, 3, „b“)
6	(2, 3, „b“)

j = $\lceil log_2$ L - 1 $\rceil$ bitů
a = typicky 8 bitů. (uvažujeme ASCII znak)
Velikost zkomprimované zprávy v našem případě je: (3 + 2 + 8) × 6 = 78 bitů.
Velikost původní zprávy je: 8 × 17 = 136 bitů
Kompresní poměr tedy je: $\frac{78}{136} = 0.57$ [2].

2.4.1.2 Dekomprese

Dekompresní algoritmus LZ77 na vstupu dostává vektor trojic (*i, j, k*). Algoritmus čte každou trojici zvlášť a každou zvlášť dekóduje. Pokud *i* a *j* v trojici se rovnají nule, můžeme beztrestně na výstup zapsat symbol *k*. V jiném případě přečteme číslo *i*. Toto číslo indikuje kolik znaků zpět se musíme ve výstupu vrátit. Poté přečteme číslo *j*, to značí velikost kódového

slova. Tedy z toho co jsme již poslali na výstup přečteme to, co se nachází i kroků zpět a jeho délka rovna j. Na konci ještě přidáme symbol k. Takto dekompresní algoritmus pokračuje, dokud nezpracuje všechny vstupní trojice. [6]

Algoritmus 6: Pseudokód dekompresního algoritmu LZ77.

Input : seznam trojic (i, j, k)
Output: dekomprimovaný řetězec

```
1  vytvoření slovníku
2  while (není konec vstupu) do
3  |  přečti trojici (i, j, k)
4  |  if i a j jsou rovny 0 then
5  |  |  přidej k na výstup
6  |  else
7  |  |  for l = 0; l > j; l++ do
8  |  |  |  přidej na výstup: výstup[velikost výstupu - i z vstup[l]]
9  |  |  end
10 |  |  přidej k na výstup
11 |  end
12 end
```

Pro příklad si vezmeme výstup z předchozího příkladu jako vstup do našeho dekompresního programu (viz tabulka 2.5). Nejprve přečteme trojici *(0, 0, a)*. Jelikož první i druhé číslo je *0*, zapíšeme symbol *a* na výstup a čteme další trojici. Tentokrát přečteme ze vstupu trojici *(1, 1, b)*. Ve výstupu se vrátíme o *1* symbol zpět a přečteme podřetězec o délce *1*, tedy *a*. K tomuto podřetězci přidáme symbol *b* a opět zapíšeme na výstup. Nyní máme na výstupu řetězec *aab*. Další trojice k přečtení je *3, 2, c*. Ve vstupu se vrátíme o tři symboly zpět a přečteme podřetězec o délce *2*, tedy podřetězec *aa*, přidáme k němu symbol *c* a pošleme na výstup. V tuto chvíli máme na výstupu řetězec *aabaac*. Takto postupujeme dál, dokud nepřečteme všechny vstupní trojice.

2.4.1.3 Implementace

V mé implementaci ve *while* cyklu procházím vstupní řetězec od začátku do konce a postupně zaplňuji a uvolňuji *look-ahead* buffer společně s *search* bufferem. Načtu si řetězec z *look-ahead* bufferu a ten postupně hledám v *search* bufferu. V průběhu si ukládám nejdelší shodný řetězec a ten na konci zakóduji jako trojici *(i,j,k)*. Tato trojice je reprezentována formou *std::tuple*

[4] a všechny tyto trojice jsou uloženy v *std::vector* [5]. Na konci algoritmu je tento vektor poslán na výstup.

2.4.2 LZ78

LZ78 je bezztrátová kompresní metoda taktéž publikována Abrahamem Lempelem a Jacobem Zivem v roce 1978. Tento algoritmus též využívá slovníkové metody. Algoritmus LZ78 se často nazývá jako LZ2.

Tento kompresní algoritmus nevyužívá žádný vyhledávací buffer, ani dopředu se dívající buffer a ani posuvné okno jako kompresní algoritmus LZ77. Místo toho využívá slovník řetězců, které byly v předchozích krocích použity. Tento slovník je na začátku běhu programu prázdný (obsahuje pouze prázdný řetězec na nulové pozici) a jeho velikost je omezena pouze dostupnou pamětí počítače.

Výsledkem algoritmu je posloupnost dvojic. První z dvojice je ukazatel odkazující se na příslušné místo do slovníku a druhý z dvojice je kód symbolu. Dvojice neobsahují žádnou informaci o délce řetězce, jelikož ta se dá získat ze slovníku. Každá dvojice koresponduje se vstupním řetězcem symbolů a tento řetězec je přidán do knihovny poté, co je dvojice zpracována. [6]

2.4.2.1 Komprese

Slovník začíná pouze s prázdným řetězcem na pozici „*0*“. Poté co jsou vstupní symboly zakódovány, řetězce jsou přidávány na další volné pozice ve slovníku. Když je přečten další vstupní symbol „*x*“, celý slovník je prohledán, zda-li tam již tento vstupní symbol není. Pokud tam tento vstupní symbol není, je přidán do slovníku a kódovým výstupem je dvojice (*0, x*). Tato dvojice indikuje spojení „prázdný řetězec a symbol x“, jelikož nula odkazuje do slovníku na prázdný řetězec. Pokud vstupní symbol „*x*“ již ve slovníku obsažený je (například na pozici 7), tak je přečten další symbol „*y*“ a slovník je znovu celý prohledán, zda-li neobsahuje řetězec „*xy*“. Pokud slovník tento hledaný řetězec neobsahuje, pak je tento řetězec přidán na další dostupnou pozici ve slovníku. Výstupní dvojice je pak dvojice (*7, x*). Tato dvojice indikuje řetězec „*xy*“, jelikož na sedmé pozici ve slovníku leží symbol „*x*“. Takto na sebe jednotlivé řetězce navazují. Proces pokračuje dokud algoritmus nenarazí na konec vstupu.

[4] Šablona std::tuple je kolekce pevné velikosti heterogenních hodnot. Jedná se o zobecnění šablony std::pair.

[5] Šablona std::vector je datové pole s vestavěnými metodami, které se dají nad ním volat.

Obecně platí, že aktuálně přečtený symbol se stává jedno-symbolovým řetězcem. Algoritmus se jej pak snaží najít ve slovníku. Pokud je symbol ve slovníku nalezen, další symbol je přečten ze vstupu a spojen se symbolem předchozím. Tvoří tedy dvou-symbolový řetězec, který se algoritmus nyní snaží najít ve slovníku. Do té doby dokud jsou symboly nacházeny ve slovníku, algoritmus přidává více symbolů ze vstupu a přidává do hledaného řetězce. V určitém bodě řetězec nebude nalezený ve slovníku, tak jej algoritmus do slovníku přidá a výstup bude dvojice poslední shody ve slovníku a poslední symbol řetězce (ten, který způsobil nenalezení řetězce). [6]

Algoritmus 7: Pseudokód algoritmu LZ78 [7].

Input : data určená ke kompresi
Output: seznam dvojic (i, c)

```
1  vytvoření slovníku
2  while (není konec vstupu) do
3  |  přečti symbol(X)
4  |  if F.X je ve slovníku then
5  |  |  F = F.X
6  |  else
7  |  |  výstup(ukazatel(F), X)
8  |  |  přidat X do slovníku
9  |  |  inicializovat F pomocí prázdného symbolu
10 |  end
11 end
```

Příklad

Vezmeme si příklad a na něm si ukážeme průběh algoritmu LZ78 a jeho chování. Řekněme že budeme chtít zakódovat slovo *abracadabra* tímto algoritmem. Tabulka 2.6 ukazuje všechny postupné kroky algoritmu. Sloupec *Výstup* obsahuje jednotlivé výstupní dvojice seřazené tak jak z algoritmu vycházejí. Sloupec *Pozice* obsahuje čísla, která indikují v kolikátém kroku byla dvojice poslána na výstup. Poslední sloupec „Řetězec“ obsahuje jednotlivé podřetězce, které byly uloženy do slovníku během chodu kompresního algoritmu LZ78.

V typickém běhu kompresního algoritmu obsahuje slovník nejprve krátké řetězce, ale čim více textu je na vstupu tím větší jsou přidávané řetězce. Velikost slovníku může být buď fixní, nebo může být určená velikostí dostupné paměti pokaždé když je algoritmus LZ78 spuštěn.

Tabulka 2.6: Slovník algoritmu LZ78 po vstupu „abracadabra“.

Výstup	**Pozice**	**Řetězec**
(0 , a)	1	a
(0 , b)	2	b
(0 , r)	3	r
(1 , c)	4	ac
(1 , d)	5	ad
(1 , b)	6	ab
(3 , a)	7	ra

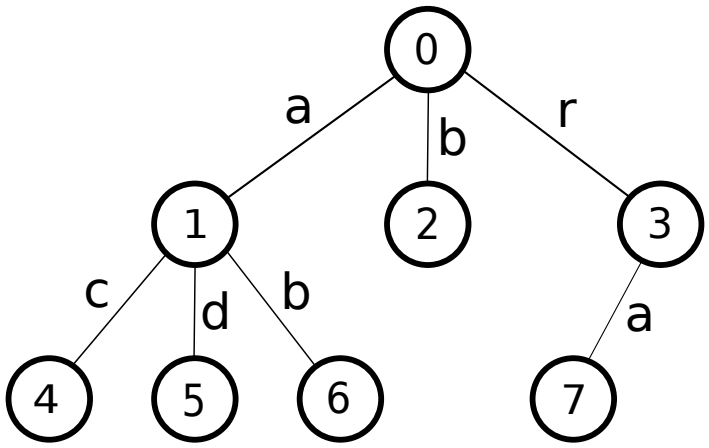
Datová struktura slovníku může být reprezentovaná více způsoby. Jeden ze způsobů jak data ve slovníku reprezentovat je pomocí tabulky. Tabulka vypadá stejně jako tabulka 2.6. Toto řešení jsem zvolil do své své implementace. Druhá možnost jak data ve slovníku strukturovat je ukládání do stromu. Každý uzel stromu může mít tolik potomků, kolik je různých symbolů v abecedě.

Strom začíná pouze s jedním kořenovým uzlem. Tento uzel reprezentuje prázdný řetězec. Všechny ostatní řetězce, které začínají prázdným řetězcem (řetězce u nichž je ukazatel nula) jsou přidány do stromu jako potomci kořenového uzlu. V našem příkladu to jsou řetězce *a , b , r*. Každý z nich se automaticky stane kořenovým uzlem podstromu (viz obrázek 2.10). Všechny ostatní řetězce, které začínají na řetězec *a* (ac, ad, ab) pokračují v podstromu z uzlu *a*.

Předpokládáme-li abecedu obsahující symboly délky 8 bitů (ASCII), tak v této abecedě je 256 různých symbolů. To znamená že v principu každý uzel stromu může mít maximálně 256 potomků. Proces přidávání potomků by měl být dynamický. Když je uzel nově vytvořen, nemá žádné potomky, tudíž by neměl zabírat žádnou paměť pro ně. Paměť by měla být přiřazena až poté, co je potomek přidán k uzlu.

Takovýto strom usnadňuje hledání a přidávání jednotlivých řetězců. Chceme-li například najít řetězec „*ac*“, program najde potomka „*a*“ kořenového uzlu. Poté nalezne potomka „*c*“ uzlu „*a*“ a je hotovo. Zde je popis našeho příkladu se vstupním řetězcem "abracadabra".

Počáteční strom začíná s pouze jedním, a to kořenovým uzlem, ohodnoceným indexem nula, reprezentující prázdný řetězec. Algoritmus nejprve přečte ze vstupního řetězce podřetězec „*a*“ a vyhledá jej ve stromě. Hledání začíná u potomků kořenového uzlu a jelikož ten žádné zatím nemá, je přidán a ohodnocen indexem *1*. Další načtený podřetězec ze vstupu je řetězec „*b*“. Tento podřetězec také není reprezentován žádným uzlem a tak



Obrázek 2.10: Postupné zřetězení vstupních symbolů algoritmu LZ78 po vstupu „abracadabra“.

je přidán jako potomek kořenového uzlu, ohodnocen indexem 2. Stejné je to i u podřetězce „r“, ten má index *3*. V dalším kroku načte algoritmus podřetězec „a“. Ten je ale již ve stromu nalezen jako potomek kořenového uzlu. Proto si algoritmus tento podřetězec uchová a přidá k němu další načtený podřetězec, a to „c“. Nyní algoritmus hledá uzel reprezentující řetězec „c“ u potomka uzlu „a“. Ten nalezen není a tak je přidán s indexem *4*. Stejné je to i pro řetězec „ad“ s indexem *5* a „ab“ s indexem *6*. V posledním kroku je přidán řetězec „a“ s indexem *7* jako potomek uzlu reprezentující řetězec „r“.

V případě LZ78, každý řetězec přidán do stromu, přidá pouze jeden symbol, a to přidáním větve do stromu. Jelikož je celková maximální velikost stromu omezena, může se zaplnit během komprese celá. To se ve skutečnosti stává pokaždé, když zrovna není vstupní řetězec nezvykle malý. Původní metoda LZ78 neupřesňuje, co dělat v takovém případě, a tak uvádím několik možných řešení.

1. Nejjednodušší řešení je slovník v tomto bodě zastavit. Žádné další uzly by neměly být přidány. Strom se sice stane statickým slovníkem, ale může být nadále použit ke kompresi řetězců.

2. Vymazat strom jakmile se zaplní a začít zase s novým, prázdným stromem. Toto řešení rozdělí efektivně vstup do bloků, kde každý blok má vlastní slovník. V případě, že obsah vstupu se pohybuje od bloku k bloku, toto řešení bude vytvářet dobrou kompresi, jelikož se odstraní slovník s řetězci, které v budoucnu pravděpodobně nebudou použity. Můžeme říct, že toto řešení předpokládá, že v budoucnu příchozí symboly budou více těžit z nových dat než ze starých (stejný předpoklad používaný u LZ77).

3. Použití UNIXové kompresní služby, která poskytuje více komplexnější řešení.

4. Když je slovník plný, vymažeme některé uzly, které jsou nejméně používané, abychom uvolnili místo pro nové. Bohužel neexistuje žádný dobrý algoritmus, který rozhodne, jaké uzly a kolik jich odstranit. [6]

2.4.2.2 Dekomprese

U dekompresního algoritmu LZ78 na vstupu dostáváme vektor dvojic (*i, s*), kde *i* je odkaz na pozici ve slovníku a *s* je přidávaný symbol. Při průchodu dekomprese si algoritmus vytváří slovník obsahující podřetězce zkomprimovaného slova. Pokud na vstupu přijde dvojice kde *i* je rovno nule, můžeme do slovníku přidat znak *s* na nejmenší možnou pozici. Pokud na vstupu přijde dvojice kde *i* je nenulové, algoritmus si najde podřetězec ve slovníku na pozici *i*, k němu přidá symbol *s* a tento výsledný podřetězec opět vloží do slovníku na nejbližší možnou pozici.

Poté co dekompresní algoritmus zpracuje všechny vstupní dvojice, projede slovník od začátku do konce a postupně pošle na vstup veškeré zapsané podřetězce. Výsledné slovo je tedy postupné zřetězení všech podřetězců ve slovníku od počátku do konce. [6]

Algoritmus 8: Pseudokód dekompresního algoritmu LZ78 [7].

Input : seznam dvojic indexu a znaku (i, c)
Output: dekomprimovaný řetězec

```
1 vytvoření slovníku
2 while (není konec vstupu) do
3 |   přečti dvojici index a znak(i, X) ze vstupu
4 |   vlož nový podřetězec (i).X do slovníku
5 |   vygeneruj řetězec na výstup
6 end
```

Pro příklad si vezmeme předchozí výstup algoritmu LZ78 (viz tabulka 2.6). Nejprve algoritmus přečetl dvojici (*0, a*). Jelikož je první číslo nulové, přidáme symbol *a* na nejbližší, tedy první, pozici ve slovníku. Stejné to bude pro vstup *0, b* a *0, r*. Dále algoritmus přečte dvojici *1, c*, jelikož je první číslo 1, vezmeme podřetězec ze slovníku na první pozici, tedy *a* a přidáme k němu symbol *c*. Řetězec tedy bude *ac*. Takto projedeme veškeré vstupní dvojice. Přečtením slovníku od začátku do konce tedy dostaneme řetězec *abracadabra*.

2.4.2.3 Implementace

V mé implementaci je výstup reprezentovaný jako *std::vector < std::pair <...> >*, kde pár obsahuje číslo *unsigned int* značící odkaz do slovníku a poslední načtený symbol. Na začátku programu si vytvořím prázdný slovník reprezentovaný jako vektor jednotlivých podřetězců vstupního řetězce. Ve *while* cyklu procházím vstupní řetězec a postupně načítám jednotlivé symboly do pomocného řetězce. Ten pak hledám ve svém vytvořeném slovníku. Pokud tam tento řetězec je, přečtu další symbol, který přidám k řetězci a hledám ve slovníku znovu. Zároveň si při tomto průchodu držím v paměti poslední shodu ve slovníku. Toto číslo odkazující do slovníku na poslední shodu posílám na výstup při první nalezené neshodě. Zároveň s ním posílám i poslední načtený symbol, který tuto neshodu způsobil.

Kapitola 3

Testování

V první řadě si musíme uvědomit, co je vlastně softwarové testování.

1. Testování vykonává či simuluje systémovou nebo programovou operaci.

2. Testování zjišťuje zda program dělá to, co dělat má a nedělá to, co dělat nemá.

3. Testování analyzuje program s cílem najít problémy a chyby.

4. Testováním se měří funkčnost a kvalita programu.

5. Testování slouží k vyhodnocení vlastností a schopností programů a posouzení toho, zda dosahují požadovaných nebo přijatelných výsledků.

6. Testování zahrnuje inspekce požadavků a designu. [8]

Všechny tyto položky jsou důležité při vývoji softwaru. Následně uvádím pět důležitých důvodů, proč by mělo být testování zařazeno jako nejdůležitější věc v projektech, zejména v jejich začátcích.

1. Chudý testovací program může způsobit časté selhávání ve vývoji. Může výrazně ovlivnit provozní výkonnost a spolehlivost. Může také zdvojnásobit nebo ztrojnásobit náklady na podporu a údržbu.

2. Dobrý testovací program je jednou z hlavních položek při ceně vývoje. Abychom udělali testování efektivní, je třeba ho řádně naplánovat a zorganizovat.

3. Dobrý testovací program výrazně pomůže při definování základních požadavků a tvorbě designu. Tato pomoc je velice potřebná k správnému zahájení projektu a může mít celkový vliv na výsledný úspěch programu.

4. Dobrý testovací program nás nutí čelit a vypořádat se s problémy za běhu vývoje, kterým bychom museli čelit na konci. Oprava těchto chyb na konci vývoje by byla jak časově, tak cenově náročná.

5. Dobrý testovací program nemůže zcela zachránit špatný softwarový projekt, ale pomáhá předejít mnoha chybám. Navíc nám dá brzo vědět, že ve vývoji softwaru jsou potíže. Testování softwaru je pojištění že vývoj probíhá správně. [8]

3.1 Shellové testovací skripty

V automatové knihovně jsou umístěny skripty začínající na předponu *tests.*, umístěné v kořenovém adresáři. Tyto skripty spouští předdefinované testy nad moduly, kde tyto testy jsou uložené v adresáři *examples2*. Zároveň s těmito testy jsou spouštěné vygenerované náhodné testy. Skript nejprve zavolá funkci *runTest* a předá jí posloupnost programů, které se mají nad vstupem spustit. Funkce *runTest* nejprve projde adresář *examples2* a podívá se po předefinovaných testech, ty pak spouští jako první. Poté funkce vygeneruje určité množství náhodných vstupů pro náhodné testy a spustí je. Spouštění algoritmů se vstupy zajišťuje druhá funkce *runTest2*, která následně i porovná vstup a výstup přes program *./acompare2*. Výsledná návratová hodnota celkové operace je pak jak zobrazena na výstupu, tak uložena do souboru. Na konci všech testů se pak návratové hodnoty sečtou a zobrazí se celkový výsledek všech spuštěných testů. Právě funkce jako například *registerResult*, *initResult*, *clearResult*, *outputResult* a podobně se často ve skriptech opakují.

3.2 Testovací program

Nově navržené a implementované testování se skládá ze dvou základních částí. První částí je testovací skript samotný, který si v sobě uchovává přidané testy, volá je, zapisuje si výsledky a podobně. Druhá část jsou skripty, které si testovací program importují a zasílají mu jaké testy se mají generovat a jak se mají porovnávat. Je to takto rozděleno z důvodu, že každý program požaduje jiný druh testování.

Algoritmus 9: Ukázka testovacího skriptu pro kompresní algoritmus LZ77.

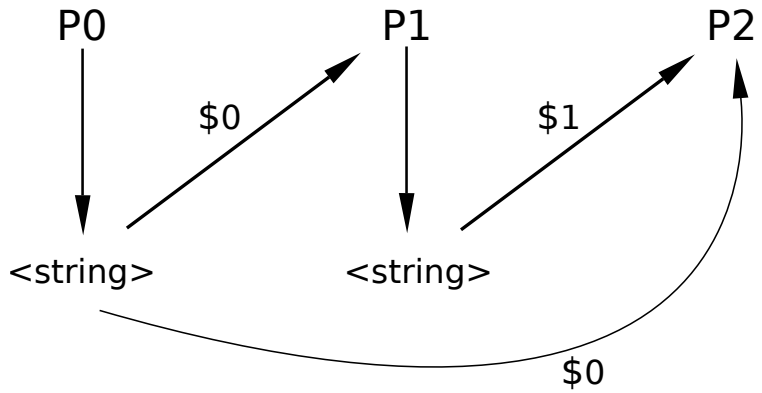
```
P0 = ["./arand2 -t ST --length 50 lz77"]
P1 = ["./acompress2 -a  lz77 -i $0",
        "./acompress2 -d -a lz77"]
P2 = ["./acompare2 $0 $1"]

controller.addRandTest([P0, P1, P2], 100)

controller.runAll()
```

V algoritmu 9 můžeme vidět, jak testovací skript vypadá například pro kompresní algoritmus LZ77. Nejprve je nutné si nadefinovat, jaké příkazy se mají spouštět. Tyto příkazy jsou uloženy v proměnných *P0, P1, P2* a tak dále. Pokud některá proměnná obsahuje více příkazů oddělených čárkou, tak se tyto příkazy zřetězují po vzoru „unixových rour“. To znamená že výstup z předchozího příkazu je vstupem nadcházejícímu příkazu.

V proměnných se mohou vyskytnout zástupné symboly jako například *$0 , $1 , $2* a podobně. Tyto zástupné proměnné reprezentují výstup z příkazu, který je obsažen v jiné proměnné. Vezmeme-li si náš příklad testovacího skriptu pro kompresní algoritmus LZ77 (viz algoritmus 9), tak vidíme, že příkaz v proměnné *P1* obsahuje zástupný symbol *$0*. Chceme totiž, aby vygenerovaný řetězec, který je uložený v proměnné *P0* byl vstupem volání kompresního algoritmu LZ77.

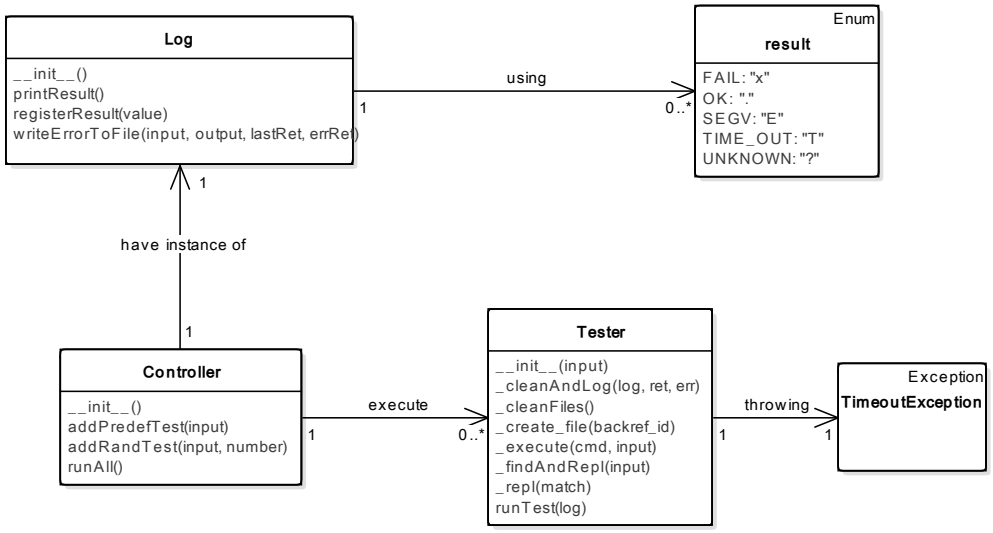
Dále se výsledek z kompresního algoritmu zřetězí s dekompresním algoritmem. V poslední proměnné pak porovnáváme výsledky z *P0* a *P1*. Tedy původní vygenerovaný řetězec s řetězcem, který prošel kompresí a dekompresí (viz Obrázek 3.1). Tyto řetězce musí být shodné. Poté se všechny tyto proměnné předají controlleru pomocí metody *addRandTest* společně s konstantou, určující kolik těchto náhodných testů se má vygenerovat. V posledním kroku se zavolá metoda controlleru *runAll*, která všechny tyto testy spustí.

Testovací program se skládá ze 3 hlavních tříd. Třída *Controller, Tester* a *Log* viz obrázek 3.2. Testy nadefinované v testovacím skriptu se přidávají do třídy *Controller*. Tato třída si veškeré přiřazené testy v sobě ukládá. Po zavolání metody *runAll()* třídy *Controller*, začne tato třída rozesílat jednotlivé testy do třídy *Tester*. Každé nově vytvořené instanci třídy *Tester* je přiřazen pouze jeden test. Tento test je přiřazen pomocí třídního konstruktoru. Toto je z důvodu jednodušší paralelizace programu. Provedení



Obrázek 3.1: Ukázka průběhu předávání vstupů a výstupů v testovacím skriptu.

testu se pak zavolá pomocí metody *runTest()* třídy *Tester*. Po provedení testu třída *Tester* vrací číslo jako návratovou hodnotu reprezentující výsledek testu. Tato hodnota je pak pomocí třídy *Log* zapsána do celkových výsledků, které jsou zobrazeny najedou po doběhnutí všech testů.



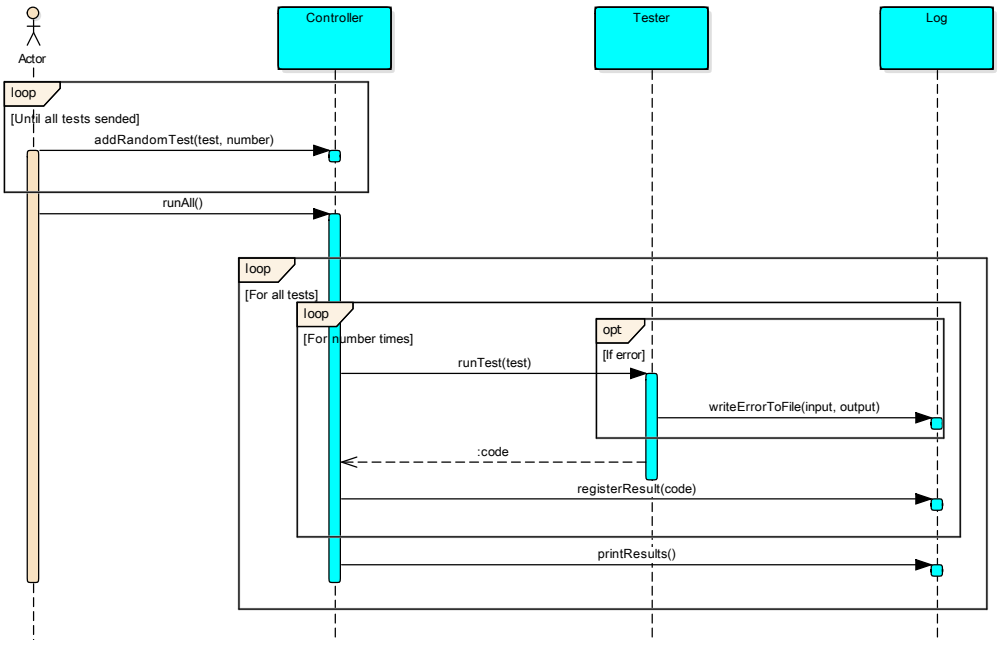
Obrázek 3.2: Diagram tříd testovacího programu.

Když třídě *Tester* přijde test, rozdělí si ho na části podle příchozí struktury, tedy na *P0, P1* a tak dále. Každý výsledek z těchto částí si v sobě ukládá pro pozdější využití. Dříve než test spustí pomocí metody *execute()*, prohledá ho, zda-li se v něm nevyskytují zastupující symboly jako například *$0 , $1* a podobně. Pokud je tento zástupný symbol nalezen, je nahrazen

výstupem z konkrétní části testu, který má třída *Tester* v sobě uložen. Pokud právě zpracovávaná část testu obsahuje více testů oddělených čárkou, je výstup z předchozího testu vstupem následujícímu testu pomocí volitelného parametru metodou *execute()*. Průběh běhu testovacího programu je znázorněn v sekvenčním diagramu viz obrázek 3.3.

Pokud při testování dojde k chybě, třída *Tester* pošle aktuální vstup a výstup přímo třídě *Log*. Ta chybu přidá do souboru s názvem *LOG_FILE*. Toto pomůže programátorovi rychleji odhalit chybu v programu. Soubor *LOG_FILE* je vždy na začátku spuštění testovacího programu promazán. Měl jsem možnost toto zapsat dvěma způsoby. Zapisovat chyby do souboru by mohla i třída *Tester* sama o sobě, ale pro přehlednost testovacího programu jsem pro zapisování chyb využil třídu *Log*.

Python nemá možnost rozdělení metod na privátní a veřejné. Proto jsem využil konvenci, která se v Pythonu používá [9]. To znamená, že všechny metody co mají být privátní mají předponu „_“. Například pokud by existovala metoda *foo()* a byla by privátní, jmenovala by se *_foo()*. Všechny ostatní veřejné metody jsou pojmenovány bez této předpony.



Obrázek 3.3: Sekvenční diagram testovacího programu.

Kapitola 4

Testování implementace

Pro testování jednotlivých modulů přidávaných do automatové knihovny se používají takzvané *unit testy*. Tyto testy jsou často používány v automatové knihovně a není problém je zde přidávat. Nejprve si něco povíme o tom co to unit testy vlastně jsou.

4.1 Unit testy

Toto testování se zabývá funkční správností a úplností jednotlivých programových jednotek, například funkcí, metod, tříd nebo jiných systémových komponent. Jednotky mohou být libovolné velikosti. Unit testování není fáze projektu, je to perspektiva, z níž se hodnotí a testují softwarové komponenty. V Automatové knihovně se unit testy používají k otestování algoritmů, datových struktur (jako je například automat, gramatika), výpis ve formátu *XML* a jiné. [10]

Pro mnou implementované algoritmy jsem vytvořil unit testy, které pokrývají i hraniční situace. Ke každému algoritmu jsou implementovány minimálně dva unit testy, a to jak k algoritmům komprese, tak algoritmům dekomprese. Jsou zde vytvořeny testy, které zahrnují prázdný vstup, vstup o délce *1* (tento vstup je problematický pro algoritmy, které si vytváří strom ve své vnitřní reprezentaci), nebo vstup větší délky. Některé z mých unit testů využívají i zpětné dekomprese. Tedy na vstup je zavolán kompresní algoritmus a na jeho výstup je zavolán dekompresní algoritmus. Pak už jen stačí porovnat prvotní vstup a výstup z dekompresního algoritmu.

Automatová knihovna využívá k unit testování framework vytvořený pro programy napsané v jazyce *C++*, knihovnu *CppUnit*. [11]

4.2 Testovací skript

Pro otestování implementovaných kompresních algoritmů, kromě vytvoření unit testů, jsem vytvořil skript *tests.acompress2.sh*. Tento skript po vzoru ostatních, již implementovaných testovacích skriptů, testuje kompresní algoritmy pomocí náhodně vygenerovaných řetězců. Tyto řetězce jsou předány na vstup příslušnému kompresnímu algoritmu. Výstup z tohoto programu je vstupem do stejného, ale dekompresního algoritmu. Výsledný výstup je pak porovnán s prvotním vygenerovaným řetězcem, který je vygenerován programem *arand2*. Porovnání výsledného výstupu probíhá přes program *acompare2*. Takto se dá testovat komprese i dekomprese algoritmu v jednom testu. Pro každý ze čtyř implementovaných algoritmů je vygenerováno 100 náhodných testovacích řetězců. Tento skript, společně se skriptem *tests.aconversion2.sh*, byl následně převeden do nového testovacího programu.

Závěr

Svou prací jsem navázal na stávající verzi Automatové knihovny a tu jsem rozšířil o možnost využití datové komprese a dekomprese.

Dle zadání byly implementovány kompresní i dekompresní algoritmy LZ77, LZ78 a Huffmanovo kódování. Navíc jsem implementoval i kompresní a dekompresní metodu Shannon-Fanova kódování, jelikož úzce souvisí s Huffmanovým kódováním. Pro všechny tyto algoritmy byly vytvořeny unit testy pro otestování správnosti. Implementace byla zvolena s ohledem na čitelnost a znovupoužitelnost algoritmů.

Vytvořil jsem program s názvem *acompress2*, pro řádné volání těchto algoritmů dle vzoru ostatních programů v Automatové knihovně. Následně byl vytvořen testovací program pro sjednocené testování všech částí Automatové knihovny. Pro ukázku jak použít tento program jsem vytvořil dva testovací skripty *tests.acompress.py* a *tests.aconversion.py*. Kompresní i dekompresní algoritmy byly následně otestovány i tímto testovacím programem.

Unit testy společně s vytvořeným testovacím programem mi pomohly odhalit pár chyb v mé implementaci. Nyní všechny testy prochází bez jediné chyby.

V budoucnu je možné dopsat zbylé testovací skripty a tím pokrýt kompletní testování Automatové knihovny. Dále je možnost paralelizace testovacího programu. Tento testovací program byl napsán tak, aby paralelizace programu byla snadno implementovatelná.

Literatura

[1] Krejčí, M.: *Komprese dat.* Magisterská práce, Fakulta elektrotechniky a komunikačních technologií - Vysoké učení technické v Brně, Brno, 2009.

[2] doc. Ing. Jan Holub, P.: *Komprese dat (nepublikované materiály k přednášce).* Praha, České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

[3] Pecka, I. T.: *Automatová knihovna – převody mezi regulárními výrazy, regulárními gramatikami a konečnými automaty.* Bakalářská práce, Fakulta informačních technologií - Vysoké učení technické v Praze, Praha, 2014.

[4] prof. Ing. Róbert Lórencz, C.: *Bezpečnost (nepublikované materiály k přednášce).* Praha, České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

[5] Fiedler, O.: *Parallel lossless data compression.* Magisterská práce, Fakulta informačních technologií - České vysoké učení technické v Praze, Praha, 2016.

[6] Salmon, D.: *Data compression.* Springer, čtvrté vydání, prosinec 2007, ISBN 9781846286025.

[7] ČVUT - Fakulta informačních technologií: *Data Compression Applets Library [online].* [cit. 2017-05-06]. Dostupné z: `http://www.stringology.org/DataCompression/index_en.html`

[8] Network, S. P. M.: Little book of testing. *Airlie Software Council*, červen 1998, [cit. 2017-05-09]. Dostupné z: `http://profinit.eu/wp-content/uploads/2016/03/LittleBookOfTesting_VolumeI.pdf`

[9] Python Software Foundation: *Python documentation [online]*. [cit. 2017-05-07]. Dostupné z: `https://docs.python.org/`

[10] Rätzmann, M.; Young, C. D.: *Software-Testing*. Galileo Computing, říjen 2002, ISBN 9783898422710.

[11] Sommerlade, E.; Feathers, M.; Lacoste, J.; aj.: *CppUnit Documentation [online]*. [cit. 2017-05-02]. Dostupné z: `http://cppunit.sourceforge.net/doc/1.8.0/index.html`

Příloha A

Návod na použití programů

A.1 Kompresní algoritmy

Nejprve musíme celou Automatovou knihovnu zkompilovat. To provedeme příkazem *make* z kořenového adresáře. Po kompilaci nalezneme spustitelné programy v adresáži *bin-debug/*.
Použití:

```
./acompress2  [-a <lz77|lz78|huffmann|shannonfano>] [-v]
[-m] [-i <file>] [-d] [--] [--version] [-h]

-a <lz77|lz78|huffman|shannonfano>,  --algorithm <lz77
   |lz78|huffman|shannonfano>
  Určuje algoritmus k~použití.

-v,  --verbose
  Vypíše více informací.

-m,  --measure
  Měří čas.

-i <file>,  --input <file>
  Použije vstup ze souboru.

-d,  --decompress
  Použije dekompresní algoritmus

--,  --ignore_rest
```

```
    Ignoruje zbytek označených argumentů za tímto příznakem.

--version
    Zobrazí informace o~verzi a skončí.

-h,  --help
    Zobrazí informace o~použití a skončí.
```

A.2 Testovací program

Testovací program se nachází v kořenovém adresáři Automatové knihovny pod názvem *tester.py*. Chceme-li napsat test k otestování nějaké části automatové knihovny, vytvoříme si nový soubor, který *tester.py* do sebe importuje. Podle vzoru již vytvořených testovacích programů nadefinujeme testy, které pošleme třídě *Controller* pomocí metody *addRandTest()*, jedná-li se o náhodné testy, nebo *addPredefTest()*, jedná-li se o předdefinované testy. Na konci je nutné všechny tyto přidané testy spustit. To učiníme pomocí metody *runAll()* třídy *Controller*.

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `automata-library` ..... adresář s Automatovou knihovnou
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `acompress` ..... program pro volání kompresních algoritmů
    - `compress` ..... kompresní algoritmy
    - `cppunit-tests` ..... cppunit testy
    - `decompress` ..... dekompresní algoritmy
    - `tester` ..... tester a testovací skripty
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF