FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Implementace kompresní metody DCA v jazyce Java
Student:	Jakub Novák
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 metodu DCA.
2) Navrhněte, analyzujte, implementujte a testujte daný algoritmus.
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

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Implementace kompresní metody DCA v jazyce Java

Jakub Novák

Katedra softwarového inženýrství
Vedoucí práce: Ing. Radomír Polách

13. května 2018

Poděkování

Chtěl bych poděkovat vedoucímu práce Ing. Radomíru Poláchovi za námět k této práci a jeho cenné rady a připomínky. Také děkuji své rodině a Karolíně 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 13. května 2018 .....

Odkaz na tuto práci

Novák, Jakub. *Implementace kompresní metody DCA v jazyce Java.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce se zabývá vlastnostmi semi-adaptivní verze kompresní metody DCA. V práci je popsán samotný algoritmus, jeho implementace, některá vylepšení a výsledky experimentů. Předmětem práce bylo nastudovat a následně analyzovat tuto kompresní metodu a navrhnout vhodný způsob její implementace s ohledem na rychlost a kompresní poměr. Praktická část je realizována v programovacím jazyce Java, ve kterém tento algoritmus zatím nebyl dostupný.

Klíčová slova Algoritmus DCA, Java, komprese, dekomprese, antislovník, kontext

Abstract

This thesis is concerned with characteristics of a semi-adaptive version of the DCA compression method. In this thesis, one can find a description of the algorithm, its implementation, some enhancements and results of experimental testing. The subject of this thesis was to study this compression method and bring in a sufficient way for its implementation with regards to speed and compression ratio. The practical segment of this thesis is realized in the Java programming language, in which this algorithm was not yet available.

Keywords DCA algorithm, Java, compression, decompression, antidictionary, context

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Žijeme v době, kdy počítače ovlivňují život mnohým z nás. Při jejich používání denně pracujeme s velkými objemy dat, které jsou potřeba efektivním způsobem ukládat a právě proto nyní patří datová komprese mezi významné odvětví počítačových věd. V dnešní době existuje nepřeberné množství kompresních algoritmů a neméně možností jejich využití. Kromě archivace dat se velmi dobře uplatní například během přenosu dat po síti nebo v různých grafických a zvukových formátech jako jsou *JPEG, GIF, PNG, MP3, aj.* Ty nejlepší z nich se staly standardní součástí sofistikovaných softwarů především díky své spolehlivosti, rychlosti a kompresnímu poměru. Existuje ale také celá řada algoritmů, kterým se přílišné pozornosti nedostává a to zejména z důvodů jejich nízké efektivity, absence možnosti paralelizace nebo neexistence vhodných datový struktur pro jejich realizaci.

DCA (Datová komprese pomocí antislovníků) byla poprvé představena v roce 1998 M. Crochemorem, F. Mignosim, A. Restivem a S. Salemim jako textově orientovaná kontextová kompresní metoda, která pracuje s binární abecedou. Oproti algoritmům používajícím slovníky sestavené z opakujících se faktorů v textu, využívá DCA slov, jež se v textu nenacházejí. Taková slova můžeme označit za zakázaná a množinu těchto slov za antislovník [1]. Na metodě mě zaujalo, že k hledání redundance přistupuje opačným způsobem než většina kompresních metod – staví kompresi na základě řetězců, které se v textu nevyskytují. Zájem ověřit, jaký má tento přístup potenciál, byla jedním z důvodů, proč jsem si toto téma zvolil.

Kapitola 1

Cíl práce

Cílem práce je na základě analýzy požadavků vybrat a následně naimplementovat kompresní metodu DCA a otestovat ji na vhodných testovacích datech. Metoda se stane součástí open-source knihovny SCT [2] (*Small Compression Toolkit*), která je momentálně dostupná v repozitáři umístěném na fakultním serveru gitlab [3]. Tato práce je součástí snahy o rozšíření této knihovny, aby mohla sloužit jako výukový materiál komukoliv, kdo se zajímá o problematiku datové komprese. Zároveň bude knihovna nabízet příležitost mezi sebou jednoduše porovnat metody v ní obsažené.

1.1 Související práce

Tato práce částečně navazuje na diplomovou práci Jiřího Bicana *Implementace vylepšení kompresní metody ACB v jazyce Java* [4]. Svou prací tímto dal vzniknout knihovně SCT [2] a přispěl do ní algoritmem ACB, který byl až doposud jedinou kompresní metodou v této knihovně. Současně připravil v knihovně zázemí pro implementaci dalších metod – konkrétně má práce využívá funkcí pro zpracování vstupních a výstupních souborů.

1.2 Analýza požadavků

Analýza požadavků je prvním stádiem vývoje každého úspěšného softwarového projektu. Je potřeba zaručit, aby byly všechny nasbírané požadavky dobře definované a nemohlo se tedy stát, že výsledný software neodpovídá představám zadávajícího práce. Existuje mnoho způsobů jejich kategorizace, nejčastěji se však setkáme s rozdělením na požadavky funkční a nefunkční. Tohoto rozdělení využiji v následující části také, jelikož je pro potřeby této práce plně dostačující.

1.2.1 Funkční požadavky

Za funkční požadavky se označují takové nároky na software, které popisují jeho požadovanou funkčnost a chování.

Na základě zadání práce a diskuse se zadavatelem – vedoucím práce – jsem dospěl k těmto funkčním požadavkům:

- **Metoda** – Pro kompresi a dekompresi bude použita semi-adaptivní (def. 2.0.5) verze kompresní metody DCA. Ta bude mimo jména souboru ke komprimaci nebo dekomprimaci jako parametr akceptovat rovněž maximální délku antislova (def. 3.1.1).
- **Klient** – Součástí software bude uživatelské rozhraní pro příkazový řádek, které bude umožňovat používat algoritmus DCA ke komprimaci i dekomprimaci dat a upravovat parametry, které se k této metodě vztahují.
- **Segmentování** – Algoritmus musí zvládat zpracovávat vstupní soubor postupně po menších částech, aby se mohla šetřit operační paměť uživatele.

1.2.2 Nefunkční požadavky

Nefunkční požadavky se často označují za doplněk funkčních požadavků. Na rozdíl od nich kladou omezení na design a provedení daného systému.

Ze zadání mi proto vyplývají následující nefunkční požadavky:

- **Knihovna** – Implementace algoritmu musí být součástí knihovny dodané vedoucím práce. Jedná se o open-source knihovnu SCT [2] (*Small Compression Toolkit*), která je momentálně dostupná v repositáři umístěném na fakultním serveru gitlab [3]. Nový kód musí dodržovat strukturu této knihovny a využívat stejných konstrukcí, aby byla zachována její celistvost.
- **Programovací jazyk** – Realizace musí být provedena v programovacím jazyce Java, ve kterém knihovna SCT vzniká.

Kapitola 2

Definice pojmů

V této kapitole vysvětlím důležité pojmy související s datovou kompresí, stringologií a teorií automatů. Tyto pojmy jsou důležité pro pochopení principu kompresní metody DCA a jsou v práci užívány ve smyslu zde popsaném.

Definice 2.0.1. (Komprese)
Datová komprese (též komprimace) je proces konvertování vstupního datového proudu (zdroje proudu nebo originálních neupravených dat) na jiný datový proud (výstupní, zkomprimovaný proud), který má menší velikost [5].

Definice 2.0.2. (Dekomprese)
Dekomprese je proces invertní k procesu komprese. Jeho cílem je rekonstrukce originálních dat do stavu před kompresí.

Definice 2.0.3. (Bezeztrátová komprese)
Za bezeztrátovou označíme kompresi tehdy, pokud po zpětné dekompresi nedojde ke ztrátě nebo znehodnocení původních informací. Používá se v situacích, kdy by ztráta jakékoliv informace mohla nenávratně poškodit původní soubor.

Definice 2.0.4. (Kompresní poměr)
Kompresní poměr je vyjádřen podílem velikostí zkomprimovaných a původních dat. Pokud mají zkomprimovaná data menší velikost než původní, bude kompresní poměr reálné číslo menší než 1.0, v opačném případě došlo k negativní kompresi. Jeho hodnota je určená rovnicí:

$$\text{kompresní poměr} = \frac{\text{velikost po kompresi}}{\text{původní velikost}}. \tag{2.1}$$

Definice 2.0.5. (Semi-adaptivní kompresní metoda)
Semi-adaptivní kompresní metody vytvářejí při prvním průchodu souborem datový model, který je v poté v druhém průchodu použit ke kompresi. Tento model je však nutné uložit společně se zkomprimovanou správou do výsledného souboru [6].

Definice 2.0.6. (Abeceda)
Abeceda je konečná množina symbolů. Typicky se značí Σ.

Definice 2.0.7. (Symbol)
Symbol je prvek abecedy.

Definice 2.0.8. (Binární abeceda)
Binární abeceda Σ je abeceda obsahující symboly {0,1}.

Definice 2.0.9. (Řetězec)
Řetězec nad abecedou Σ je konečná posloupnost symbolů dané abecedy.

Poznámka 2.0.10. (Slovo)
Výrazem *slovo* z abecedy Σ bude myšlen řetězec z téže abecedy.

Poznámka 2.0.11. (Notace $w[a..b]$ u řetězce w)
Pokud je u řetězce w použita notace $w[a..b]$, je tím myšlen rozsah symbolů v intervalu $\langle a, b\rangle$.

Definice 2.0.12. (Délka řetězce)
Délku řetězce nebo též *slova* označujeme jako $|x|$. Vyjadřuje počet symbolů, ze kterých se řetězec skládá.

Poznámka 2.0.13. (Prázdný řetězec)
Prázdný řetězec je takový, pro který platí, že $|x| = 0$. Značíme ho ε.

Definice 2.0.14. (Deterministický konečný automat)
Deterministický konečný automat DKA je pětice $M = (Q, A, \delta, q_0, F)$, kde

- Q je konečná množina vnitřních stavů,
- A je konečná vstupní abeceda,
- δ je zobrazení z $Q \times A$ do Q,
- $q_0 \in Q$ je počáteční stav,
- $F \subset Q$ je množina koncových stavů [7].

Definice 2.0.15. (Nedeterministický konečný automat)
Nedeterministický konečný automat NKA se od deterministického popsaného v definici 2.0.14 liší pouze v přechodové funkci δ. U NKA je zobrazení δ definováno jako zobrazení z $Q \times (A \cup \{\varepsilon\})$ do množiny podmnožin Q [7].

Definice 2.0.16. (Úplný konečný automat)
Úplný konečný automat je deterministický konečný automat $M = (Q, A, \delta, q_0, F)$, pro který platí, že zobrazení $\delta(q, a)$ je definováno pro všechny stavy $q \in Q$ a všechny vstupní symboly $a \in A$ [7].

Definice 2.0.17. (Konečný překladový automat)
Konečný překladový automat(KPA) je konečný automat, který má kromě konečné vstupní abecedy i konečnou výstupní abecedu (produkuje výstup). KPA M lze formálně zapsat jako šestici $M = (Q, A, D, \delta, q_0, F)$, kde

- Q je konečná množina vnitřních stavů,
- A je konečná vstupní abeceda,
- D je konečná výstupní abeceda,
- δ je zobrazení z $Q \times (A \cup \{\varepsilon\})$ do množiny konečných podmnožin $2^{Q \times D^*}$,
- $q_0 \in Q$ je počáteční stav,
- $F \subset Q$ je množina koncových stavů.

Kapitola 3

Popis metody

Kompresní metoda DCA pracuje se vstupním souborem vždy přímo v jeho binární podobě, jinak řečeno pracuje s binární abecedou (def. 2.0.8). Je však vhodná především ke komprimaci textových souborů, kde dosahuje nejlepších výsledků. Kódování každého symbolu (def. 2.0.7) je závislé na jeho okolí (tedy symbolech, které jej obklopují), a proto spadá tato metoda do kategorie kontextových metod. Během komprese využívá takzvaného „antislovníku“ (AD) – datové struktury, která v sobě obsahuje množinu „antislov“. *Antislova* (též *zakázaná slova*) jsou řetězce (def. 2.0.9), které se ve vstupním souboru nenacházejí. Tento antislovník je specifický pro každý vstupní soubor. Prvním krokem metody DCA je proto získat takový antislovník, aby ho mohla využít ke kompresi. Nejprve vysvětlím, co jsou to *Antislova*.

3.1 Antislova

Definice 3.1.1. (Antislovo)
Antislovo, též označováno jako *zakázané slovo*, je řetězec symbolů určité maximální délky l z abecedy $\Sigma = \{0,1\}$ takový, že se v kódovaném textu[1] nevyskytuje, ale vyskytuje se v něm jeho libovolný sufix i prefix.

V definici uvedená maximální délka antislova l je parametrem metody DCA. Volba tohoto parametru ovlivňuje počet a samozřejmě délku nalezených antislov a je proto jedním z hlavních faktorů ovlivňujících výsledek komprese. Pro antislovo v tedy platí, že:

$$|v| \leq l\,. \tag{3.1}$$

Být antislovem však není dostačující podmínkou pro jeho přidání do antislovníku. Aby mělo skutečně smysl antislovo do antislovníku přidat, je náš požadavek ještě to, že antislovo musí být minimální (def. 3.1.2).

[1] Jelikož metoda pracuje s binární abecedou, za text se bude v této práci považovat libovolný binární řetězec.

Definice 3.1.2. (Minimální zakázané slovo – MFW)
Mějme antislovo u, antislovo $v \neq u$ a řetězce x, y z Σ^*, kde $\Sigma = \{0,1\}$. Řetězec u je *MFW* (Minimal forbidden word) právě tehdy, když platí:

$$\forall(v)\forall(x)\forall(y) : u \ \neq x * v * y\,. \tag{3.2}$$

Operátor $*$ znázorňuje operaci zřetězení řetězců.

Vysvětleme si nyní oba tyto pojmy na jednoduchém příkladu. Mějme vstupní řetězec $w = 100101$ a maximální délku antislova $l = 4$. V tomto řetězci můžeme nalézt 2 minimální antislova 3.1.2, konkrétně $\{11, 000\}$. Ačkoliv je řetězec 011 také antislovem vstupního řetězce w, nejedná se o minimální zakázané slovo, jelikož jsme schopní podle (3.2) najít antislovo 11, řetězec $x = 0$ a řetězec $y = \varepsilon$ a ukázat, že

$$011 = 0 * 11 * \varepsilon\,.$$

Tato rovnost však odporuje výrazu v definici 3.2, a proto se v případě antislova 011 nemůže jednat o antislovo minimální.

Pro účely antislovníku jsou antislova, která nejsou minimální, nevhodná, jelikož by do slovníku zanášela redundantní informaci. Antislovo 11 říká, že se v textu neobjevují dva po sobě jdoucí jedničkové bity. Obdobně antislovo 011 říká, že se v textu neobjeví trojice bitů v podobě 011. Tato informace má však nulovou informační hodnotu, jelikož již z prvního antislova víme, že po jedničkovém bitu nesmí následovat další jedničkový bit.

3.2 Antislovník

Antislovník *AD* považuji za nejdůležitější segment DCA metody. Protože je jeho obsah závislý na kódovaném souboru, je nutné, aby byl vytvořen před začátkem komprese. V našem případě využijeme tzv. semi-adaptivní přístup 2.0.5, což znamená, že je před začátkem komprese potřeba jeden průchod vstupními daty navíc, během kterého dojde k vytvoření antislovníku. Zadefinovat ho můžeme takto:

Definice 3.2.1. (Antislovník)
Pro vstupní řetězec w z abecedy Σ je *antislovníkem* $\mathrm{AD}(w)$ množina řetězců určité maximální délky l z téže abecedy Σ a platí, že:

$$\forall(v \in \mathrm{AD}(w)) : v \ \text{ je antislovo podle 3.1.1}$$

a zároveň platí, že:

$$\forall(v \in \mathrm{AD}(w)) : v \ \text{ je minimální zakázané slovo podle 3.1.2}\,.$$

Nalezení veškerých minimálních zakázaných slov však nemusí být zrovna jednoduchý úkol. Jedná se o časově i paměťově nejnáročnější operací celé metody, nicméně výsledek této operace je možné využít k rychlé a paměťově nenáročné kompresi.

3.3 Kódování

Pro zakódování souboru potřebujeme především dvě věci. Vstupní soubor a k němu patřičně vytvořený antislovník AD. Způsob, kterým lze AD získat je popsán níže v sekci *Tvorba antislovníku* 3.5. Jakmile jej máme k dispozici, můžeme na jeho základě vytvořit DCA kodér, který dokáže informace z AD použít k určení, které bity lze ve vstupním souboru vynechat.

Princip kodéru je následovný. Během čtení vstupního textu T kontroluje, zda $\exists(v \in \mathrm{AD(T)}, b \in \{0,1\})) : v = v' * b$ takové, že v' je sufixem aktuálně zpracovávané části textu. Pokud ano, musí v textu následovat komplement b ($\neg b$). Díky pracování s binární abecedou je jednoduché tento komplement jednoznačně určit – pro $b = 0$ je $\neg b = 1$ a pro $b = 1$ je $\neg b = 0$. Díky této znalosti jsme schopni předpovědět, že příštím bitem vstupního souboru bude právě $\neg b$ a můžeme si dovolit tento symbol z výstupu vynechat. V opačném případě je načtený bit překopírován do výstupního proudu. Jinak řečeno, pokud je libovolný sufix aktuálně načteného řetězce z T shodný s libovolným řetězcem z antislovníku (mimo jeho posledního znaku), můžeme příští symbol na vstupu vynechat, protože jsme schopni na základě antislovníku tento symbol opět zrekonstruovat.

Z předešlého odstavce zároveň vyplývá, že je nezbytné použít totožný antislovník pro kódování i dekódování, a proto je nutné si na začátek zkomprimovaného souboru tento antislovník uložit.

Následná ukázka zobrazuje pseukód výše popsaného.

vstup vstupní binární řetězec T, příslušný antislovník AD
výstup T' (zakódovaný text T)
funkce:
1: $T' \leftarrow \varepsilon$
2: **for** i od 0 do $|T| - 1$ **do**
3: *přeskoč* $\leftarrow$ *false*
4: **for** každé $mfw \in AD$ **do**
5: mfw' $\leftarrow$ mfw bez posledního bitu
6: **if** $i > 0$ **and** mfw' je sufix $T[0..i-1]$ **then**
7: *přeskoč* $\leftarrow$ *true* {signál, že je možné i-tý bit vynechat}
8: **end if**
9: **end for**
10: **if** *přeskoč* je *false* **then**
11: $T' \leftarrow T' * T[i]$ {na výstup přidej i-tý bit vstupu}
12: **else**
13: **continue**
14: **end if**
15: **end for**
16: **return** T'

Algoritmus 1: DCA kodér

Ukažme si jak tento algoritmus funguje na krátkém příkladu. Mějme vstupní řetězec $w = 1101011011$. Pro maximální délku antislova $l = 5$ nalezneme antislovník AD = {111, 00, 01010}. Algoritmus bude postupovat následovně:

Příklad 3.3.1. (ukázka kódování)

$v_0 = \varepsilon$	$\mathrm{dca}(v_0) = \varepsilon$	
$v_1 = 1$	$\mathrm{dca}(v_1) = 1$	
$v_2 = 11$	$\mathrm{dca}(v_2) = 11$	
$v_3 = 110$	$\mathrm{dca}(v_3) = 11$	$111 \in \mathrm{AD}$
$v_4 = 1101$	$\mathrm{dca}(v_4) = 11$	$00 \in \mathrm{AD}$
$v_5 = 11010$	$\mathrm{dca}(v_5) = 110$	
$v_6 = 110101$	$\mathrm{dca}(v_6) = 110$	$00 \in \mathrm{AD}$
$v_7 = 1101011$	$\mathrm{dca}(v_7) = 110$	$01010 \in \mathrm{AD}$
$v_8 = 11010110$	$\mathrm{dca}(v_8) = 110$	$111 \in \mathrm{AD}$
$v_9 = 110101101$	$\mathrm{dca}(v_9) = 110$	$00 \in \mathrm{AD}$
$v_{10} = 1101011011$	$\mathrm{dca}(v_{10}) = 1101$	

Vstupní řetězec $w = 1101011011$ v tomto příkladě algoritmus DCA zmenšil na řetězec $w' = 1101$. Výsledek se však může lišit v závislosti na parametru l použitém při tvorbě antislovníku.

Na tomto příkladě lze vidět, jakým způsobem kodér při kódování postupuje. V prvním sloupci je aktuálně kódovaná část vstupního řetězce, v prostředním sloupci odpovídající výstupní řetězec a v posledním sloupci antislovo, které bylo použito, aby mohl být aktuální symbol z výstupu vynechán. To nastane pokaždé, když se kodér při čtení vstupního textu dostaneme do situace, kdy v antislovníku existuje slovo, které (bez posledního bitu) tvoří sufix aktuálně přečtenému textu.

Tento jev je vidět například na řádku 4 příkladu 3.3.1. V dané situaci jsou již ze vstupu přečtené bity 11 a kodér se chystá načíst další vstupní symbol (v našem případě 0). V antislovníku se však nachází zakázané slovo 111, které říká, jaká posloupnost na vstupu v žádné situaci nemůže být. I bez přečtení dalšího symbolu je proto jisté, že příští bit bude nulový, a proto algoritmus tento bit do výstupního řetězce nepřidá.

Do výsledného zakódovaného souboru je nutné přidat hlavičku, která v sobě obsahuje celý antislovník a poté informaci buď o původní velikosti souboru a nebo o množství bitů, které byly z původního souboru vynechány. Bez těchto informací by nebylo možné soubor úspěšně dekódovat do původní podoby.

3.4 Dekódování

Dekódování neboli dekomprese (def. 2.0.2) je v metodě DCA velmi podobné s procesem kódování. Jak již název napovídá, je jeho cílem vrátit zkomprimovaný soubor do stavu, v jakém se nacházel před kompresí.

Pro dekódování vstupního textu T potřebujeme znát antislovník AD, který byl použit pro jeho zakódování a informaci o jeho původní velikosti nebo o počtu bitů, které byly z tohoto souboru vynechány. Při implementaci jsem využil první možnosti, tzn. uložení informace o původní velikosti souboru, a budu tedy mít na mysli tuto alternativu i při následujícím popisu. Všechny tyto informace jsou typicky uložené v hlavičce kódovaného souboru a je proto nutné z ní nejdříve tyto informace získat.

Dekodér funguje tak, že přečte symbol ze vstupního textu T, zkopíruje ho do výstupního proudu T′ a dokud $\exists(v \in \text{AD}, b \in \{0,1\})) : v = v' * b$ takové, že v' je sufixem T′, přidává na výstup komplement bitu b ($\neg b$). Tento proces je ukončen v moment, kdy |T′| dosáhne velikosti původního souboru před kompresí.

Pseudokód možné implementace dekodéru:

vstup vstupní binární řetězec T, antislovník AD, původní počet bitů *size*
výstup T' (dekódovaný text T)
funkce:

```
1:  T' ← ε; i ← 0
2:  while |T'| < size do
3:    for každý mfw ∈ AD do
4:      mfw' ← mfw bez posledního bitu
5:      b ← mfw[last]
6:      if mfw' je sufix T' then
7:        T' ← T' * ¬b {na výstup přidej komplement bitu b}
8:        continue while
9:      end if
10:   end for
11:   T' ← T' * T[i] {přidej na výstup i-tý symbol ze vstupu}
12:   i ← i + 1
13: end while
14: return T'
```

Algoritmus 2: DCA dekodér

Stojí za zmínku, že kdyby antislovník neobsahoval žádná antislova, kodér i dekodér by pouze přepsaly svůj vstup na výstup a s původními daty by se nic nestalo. Naopak zakódovaná data by navíc obsahovala hlavičku, ačkoli zanedbatelně velkou.

Na následujícím příkladě předvedu zpětnou dekompresi dat z ukázky 3.3.1. Mějme tedy zakódovaný vstup dca(w) = 1101, antislovník AD = {111, 00,

01010} a původní velikost vstupního řetězce $|w| = 10$ bitů. Dekódovací algoritmus proběhne tímto způsobem:

Příklad 3.4.1. (ukázka dekódování)

$v_0 = \varepsilon$

$v_1 = 1$

$v_2 = 11$

$v_3 = 110$ $\quad 111 \in \text{AD}$

$v_4 = 1101$ $\quad 00 \in \text{AD}$

$v_5 = 11010$

$v_6 = 110101$ $\quad 00 \in \text{AD}$

$v_7 = 1101011$ $\quad 01010 \in \text{AD}$

$v_8 = 11010110$ $\quad 111 \in \text{AD}$

$v_9 = 110101101$ $\quad 00 \in \text{AD}$

$v_{10} = 1101011011$

V této fázi je $|v_{10}| = 10$ a dekódování je ukončeno. Dekomprimovaný řetězec $w = v_{10} = 1101011011$ je shodný se vstupním řetězcem z příkladu 3.3.1, dekomprese byla úspěšná.

V prvním sloupci lze v ukázce 3.4.1 vidět aktuálně dekódovaný řetězec, pravý sloupec ukazuje, které antislovo bylo použito pro dekódování posledního bitu příslušného řetězce. Na řádcích, kde je pravý sloupec prázdný, byl dekódovaný řetězec doplněn jedním znakem vstupního řetězce. Jedná se o řádky s v_1, v_2, v_5 a v_{10}. Tyto řádky končí postupně symboly 1,1,0 a 1, což odpovídá vstupnímu řetězci pro dekódování $\text{dca}(w) = 1101$. Můžeme si tedy všimnout, že algoritmus využil jak celého vstupního řetězce, tak i všech antislov. To je dáno z definice minimálních zakázaných slov (def. 3.1.2).

Na ukázce 3.4.1 je také vidět, proč je pro korektní dekódování nutné znát počet bitů původního souboru (popřípadě počet ušetřených bitů při kompresi). Tento údaj je potřeba, aby se algoritmus mohl včas zastavit a nedekódoval navíc symboly, které v původním souboru nebyly. V ukázce je algoritmus z tohoto důvodu zastaven ve stavu $v_{10} = 1101011011$. Kdyby se tak neučinilo, našel by dekodér v antislovníku AD vyhovující antislovo 111 a následně další vyhovující antislovo 00, čímž pádem by přidal do výstupního řetězce navíc symboly 0 a 1. Tyto bity se však v původním textu nevyskytovali a je proto potřeba tomuto chování zamezit.

3.5 Tvorba antislovníku

Doposud jsme v situacích, kdy byl antislovník (def. 3.2.1) potřeba, předpokládali jeho automatickou znalost. Takový předpoklad by byl velmi naivní – v praxi k této skutečnosti nejspíše nikdy nedojde. Je proto potřeba vhodným způsobem antislovník zkonstruovat pokaždé, když se chystáme ke kompresi. Vzhledem k tomu, že se jedná o množinu slov splňující určité specifické požadavky, dalo by se vymyslet mnoho způsobů, jak antislovník vytvořit. Na konci sekce 3.2 jsem zmínil, že tvorba AD je nejsložitější operací celé metody, a to jak po stránce časové, tak paměťové. Je proto skutečně nutné vybrat pro jeho konstrukci co nejvhodnější strukturu.

V předchozích sekcích zabývajících se kódováním 3.3 a dekódováním 3.4 bylo možno si všimnout, jakým způsobem se algoritmus ke slovům z antislovníku chová. Vždy nás zajímá, zda pro některé minimální antislovo *mfw* platí, že je sufixem nějakého textu T. Z definice antislova (def. 3.1.1) víme, že každý jeho prefix se v textu zaručeně vyskytuje, a proto musí tato situace vždy alespoň jednou nastat. Zároveň je potřeba, aby byla nalezena všechna antislova určité maximální délky l a aby bylo časově nenáročné zjistit, zda je takové antislovo minimální (def. 3.1.2).

Nejdřív je tedy potřeba určit množinu všech faktorů textu délky maximálně l a poté zjistit, které chybí. Chybějící z nich můžeme označit za antislova – ne však nutně minimální.

Definice 3.5.1. (Faktor řetězce)
Faktor neboli podřetězec w' řetězce $w \in \Sigma$ je řetězec, pro který platí $w = u * w' * v$, kde $u, w', v \in \Sigma^*$.

3.5.1 Binární suffix trie

Jako vhodná datová struktura pro konstrukci antislovníku se nabízí *binarní suffix trie* [8]. Jedná se o stromovou strukturu, ve které každý uzel reprezentuje jeden faktor textu T. Omezením jeho maximální hloubky parametrem l bude *suffix trie* obsahovat všechny faktory maximální délky l. K sestavení tohoto stromu je potřeba jeden průchod vstupními daty.

Během vkládání faktorů textu do stromu se využívá takzvaných *suffix linků*. Suffix link je spoj z uzlu, který reprezentuje binární řetězec $b * u$, do uzlu reprezentujícího řetězec u, kde b představuje jeden jediný bit. Například z uzlu, který představuje faktor 0011, povede suffix link do uzlu reprezentujícího řetězec 011. Těchto linků se dá dále využít jak pro rychlejší sestavení celého stromu, tak k efektivnímu ověření, zda nalezené antislovo splňuje podmínku minimálnosti.

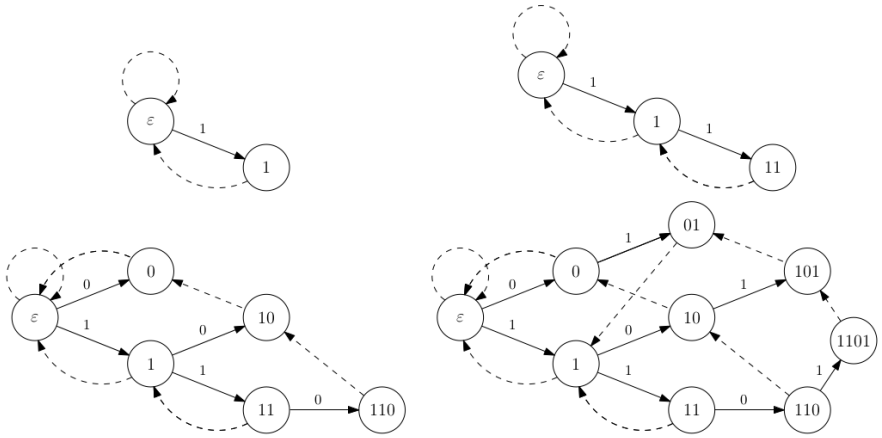
Při sestavování antislovníku si algoritmus nejprve načte prvních l bitů vstupního textu a vloží tuto sekvenci do binárního suffix trie, zároveň pro všechny uzly vytvoří příslušné suffix linky. Algoritmus si vždy pamatuje referenci na uzel, který reprezentuje poslední vložený faktor délky l, $b_1 \dots b_l$. Při

načtení nového bitu b ze vstupního textu se použije suffix link ze zmíněného uzlu $b_1 \dots b_l$ vedoucí do uzlu $b_2 \dots b_l$, tomu vytvoří se potomek $b_2 \dots b_l b$ (pokud tento potomek ještě neexistuje) a vytvoří se mu suffix link pro tohoto potomka. Potomek se použije jako reference na poslední vložený faktor délky l při dalším vkládání.

Při vytváření suffix linku pro potomka $b_2 \dots b_l b$ se může stát, že hledaný faktor $b_3 \dots b_l b$ se ve stromě nenachází. V tom případě je nutné tento uzel vytvořit a zařídit, aby i pro něj existoval patřičný suffix link. V nejhorším případě se může stát, že bude potřeba vytvořit postupně všechny uzly $b_3 \dots b_l b$, $b_4 \dots b_l b$ a tak dále, dokud se nevytvoří poslední uzel reprezentující samotný faktor b.

Poznámka 3.5.2. Vytváření těchto uzlů si můžeme dovolit, protože vytváříme vždy pouze o jedna kratší faktor k jinému faktoru, který už se ve stromě – a také textu – nachází. Máme tedy zaručeno, že do stromu pouze přidáváme další faktor textu, který v něm doposud chyběl a zatím jen nedošlo k jeho přidání.

Postupné vytváření binárního suffix trie je znázorněno na obrázku 3.1:



Obrázek 3.1: Ukázka konstrukce binárního suffix trie pro text T = ‘1101’ [9]

Na obrázku 3.1 je vyobrazeno postupné přidávání uzlů ze vstupního textu T a rekurzivní vytváření suffix linků, které k nim patří. Hrany vedoucí z rodiče do potomka (tzn. do faktoru textu o jeden znak delšího) jsou znázorněny nepřerušovanou čarou, přerušované čáry pak znázoruňují suffix linky. Uzel, označený znakem ε je kořenovým uzlem stromu představující zároveň prázdný faktor textu T. Strom znázorněný na obrázku vpravo dole je finální sestavený binární suffix trie pro T = 1101.

3.5.2 Nalezení minimálních antislov

Poté, co algoritmus projde celý vstupní text a vytvoří obdobný suffix trie, je vše připraveno k tomu, aby jsme s jeho pomocí mohli nalézt všechna minimální antislova. Pseudokód 3 ukazuje, jak může takový algoritmus fungovat.

```
   vstup suffix trie 𝒯
   výstup množina minimálních zakázaných slov MFW
   funkce:
 1: MFW ← ∅ {Prázdná množina antislov}
 2: for každý uzel n z 𝒯 do
 3:   if n má pouze 1 potomka then
 4:     vytvoř druhého potomka a označ ho jako antislovo v
 5:     přidej v do MFW
 6:   end if
 7: end for
 8: for každé antislovo v z MFW do
 9:   if !isMinimal(v) then {Funkce určující, zda je v minimální antislovo}
10:     odeber v z MFW
11:   end if
12: end for
13: return MFW
```

Algoritmus 3: Nalezení minimálních antislov pomocí suffix trie

U pseudokódu 3 nemusí být na první pohled jasné, jakým způsobem funguje. První věc, na kterou bych chtěl upozornit se nachází v podmínce na řádku číslo 3, která říká „n má pouze 1 potomka“. Je třeba si uvědomit, že se jedná o binární strom, který je vytvářen poměrně specifickým způsobem, a můžou proto nastat 3 situace.

- **Uzel n má 0 potomků** – V takovém případě se buď nacházíme v maximální hloubce stromu, nebo končil text takovým faktorem, který se nikde jinde v textu nevyskytoval. V prvním případě zde nemá smysl antislovo hledat, jelikož by bylo delší než je povolené parametrem pro maximální délku, v případě druhém ho nemá smysl vytvářet, protože by nebylo při kompresi/dekompresi ani jednou použito (o jedna větší faktor k uzlu n se v textu nenachází).

- **Uzel n má 1 potomka** – Toto je jediný stav, který nás opravdu zajímá. To, že uzel už jednoho potomka má znamená, že se ještě nenacházíme v maximální hloubce stromu, a tudíž i při vytvoření chybějícího uzlu pro antislovo nepřesáhneme maximální možnou délku. Zároveň jsem tím našli chybějící faktor, který se v celém textu nevyskytuje – antislovo. Chybějící uzel tedy vytvoříme a označíme jej za „zakázaný“. Zatím však nevíme, jestli se zároveň jedná o antislovo minimální.

- **Uzel n má 2 potomky** – Pokud má uzel oba potomky, znamená to, že ještě nejsme v maximální hloubce stromu, zároveň existují oba o jedna delší faktory k uzlu n. Tento uzel ignorujeme stejně jako ten s 0 potomky.

Tím se dostáváme do stavu, kdy máme vytvořený strom a identifikovaná zakázaná slova. Zbývá už jen zjistit, která z nich jsou zároveň minimální zakázaná slova, a která tím pádem patří do antislovníku. V pseudokódu 3 je tento proces znázorněn použitím funkce „*isMinimal*“, která zjistí, zda je antislovo minimální. Tato operace může znít složitě vzhledem k definici minimálního antislova (def. 3.1.2), která říká, že je potřeba zjistit, zda dané antislovo není složeninou jiného antislova a libovolných řetězců x, y ze stejné abecedy. Tuto informaci je velmi jednoduché ověřit právě díky použití struktury suffix trie.

Pro zjednodušení lze říci, že antislovo v je minimální, pokud neexistuje jiné antislovo u, které by bylo jeho faktorem. To, že u je faktorem v znamená jednu z těchto tří možností. Řetězec u je prefixem řetězce v, řetězec u se nachází uprostřed řetězce v a poslední možnost, u je sufixem v. První dvě možnosti však můžeme okamžitě vyloučit a to z jednoduchého důvodu. Jedná se o situaci, která nemůže v námi vytvořeném suffix trie nikdy nastat. Znamenalo by to totiž, že uzel označený jako antislovo má alespoň jednoho dalšího potomka, což je nemožné vzhledem k tomu, že by se nejednalo o faktory vstupního textu T. Navíc jsou uzly označené jako „zakázané“ až zpětně po zpracování celého vstupu. Nemůže se tedy stát, že by takový uzel ve stromě někdy vznikl.

Jediná zbývající možnost k vyloučení minimálnosti antislova v tedy je, že v textu existuje nějaké jiné antislovo u, které tvoří jeho sufix. Zde můžeme opět využít jedné z vlastností suffix trie, tentokrát suffix linků, které jsme pro každý uzel stromu vytvářeli. Pro připomenutí, suffix link je spoj z uzlu, který reprezentuje binární řetězec $b * u$, do uzlu reprezentujícího řetězec u, kde b představuje jeden jediný bit. Jak jich za tímto účelem využít si ukažme na příkladu.

Příklad 3.5.3. Mějme pro text T = 1101 suffix trie vytvořený stejným způsobem jako na obrázku 3.1 a parametr $l = 4$ definující maximální délku antislova. Za antislova algoritmus popsaný pseudokódem 3 v první části označí množinu {00,100,111,110}. Pro každé antislovo v z této množiny se poté provedou následující instrukce. Nejprve se v rozdělí na dvě části (v' a b) tak, že

$$v \ = v' * b\,, \tag{3.3}$$

kde b je poslední bit tohoto antislova. Řetězec v' je v suffix trie vyjádřen rodičovským uzlem k uzlu v. Z tohoto rodičovského uzlu poté algoritmus postupuje po suffix lincích a zkoumá, zda některý z těchto uzlů nemá potomka určeného bitem b, který je označen jako „zakázaný“. Pokud ho nalezne, nemůže být antislovo v minimální. Naopak pokud jsme žádného takového potomka nenalezli a dostali se pomocí suffix linků až k samotnému kořeni suffix trie, můžeme v označit za minimální zakázané slovo.

Pro antislovo $v = 1100$ by vypadal tento postup následovně. Slovo v se rozdělí na řetězec $v' = 110$ a bit $b = 0$ podle 3.3. Řetězec v' je v suffix trie reprezentován rodičovským uzlem pro v. Z uzlu v' vede suffix link do uzlu 10. Dále nás zajímá, zda potomek tohoto uzlu určený bitem b $(10 * b)$, je označen jako zakázaný. V tomto případě tomu tak skutečně je (100 je také antislovo), a proto řetězec 1100 nemůže být minimálním antislovem. Kdyby tento potomek antislovem nebyl, pokračoval by algoritmus z uzlu 10 pomocí jeho suffix linku na uzel 0 a případně až do uzlu ε.

Je vhodné poznamenat, že tato skutečnost nic nevypovídá o minimálnosti antislova 100. Je proto tedy nutné použít tento proces pro všechna nalezená antislova.

Výsledkem všech těchto operací je množina minimálních antislov. Jak je z popisu vidět, není zcela jednoduchým úkolem tuto množinu získat. Lze toho však dosáhnout díky datové struktuře suffix trie, která poskytuje řešení s přijatelnou časovou náročností.

Kapitola 4

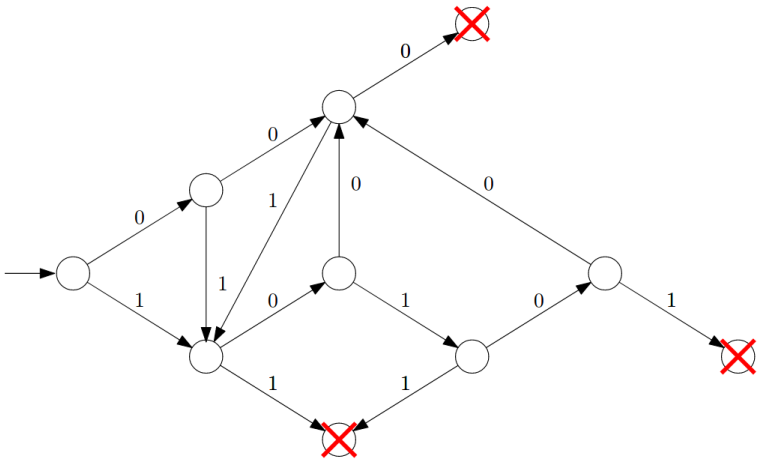
Kódovací automaty

V této kapitole bych se rád zabýval problematikou realizace kodéru a dekodéru, popsaných v předchozí kapitole v sekcích 3.3 a 3.4. Jedná se především o část, kde ukázané pseudokódy požadují, aby se pro každý bit vstupního řetězce vzala v potaz všechna antislova z AD. Taková operace by byla velmi neefektivní, především protože s velkým parametrem l pro omezení maximální velikost antislov se jich budou v textu vyskytovat tisíce až desetitisíce.

4.1 Konečné automaty

Pro tuto situaci existuje řešení v podobě deterministického konečného automatu (def. 2.0.14) sestaveného pomocí slov antislovníku tak, aby přijímal všechny jejich faktory, kromě antislov samotných. Toho lze dosáhnout tím, že tento konečný automat nebude úplný (Pro všechny stavy budou existovat přechody pro oba symboly binární abecedy, kromě přechodů do stavů reprezentujících poslední symbol antislov). Tím pádem, pokud se během komprese při čtení textu dostaneme do situace, kdy se automat nachází ve stavu, ze kterého je definován pouze jeden přechod, můžeme se do tohoto stavu přesunou a symbol na vstupu vynechat. V případě dekomprese naopak tento symbol přidáme na výstup a pokročíme přes tento přechod do dalšího stavu. Stále však zbývá dovysvětlit, jak je tento automat z antislovníku vytvořen.

Algoritmus *L-automaton* vezme za vstup binární strom $\mathcal{T}$, který by se dal také označit jako deterministický konečný automat přijímající slova z AD, a upraví ho na automat, který přijímá všechny řetězce mimo slov z antislovníku. Aby automat fungoval správně, je nutné stavům, pro které je definovaný pouze jeden přechod, přidat chybějící přechod (zpětný) pro opačný znak binární abecedy. Zpětné hrany se doplní na základě takzvané „failure funkce“ *f*. Cílem těchto hran budou stavy reprezentující nejdelší kratší sufixy k aktuálnímu stavu následované chybějícím symbolem. O „failure funkci“ lze říci, že zde má podobný význam jako suffix link v suffix trie. Jak celý algoritmus *L-automaton* funguje je naznačeno v pseudokódu 4. Na obrázku 4.1 je



Obrázek 4.1: Konečný automat vytvořený z antislovníku {000,11,10101} [10]

automat pro antislovník {000,11,10101} po aplikaci zmíněného algoritmu, popsaném detailněji v [1].

Poznámka 4.1.1. Algoritmus *L-automaton* ponechává stavy reprezentující antislova a nastavuje jim přechody pro všechny písmena abecedy jako zpětnou smyčku do sebe sama. Tyto stavy lze však z automatu zcela odstranit, jelikož by měly být z principu antislova nedosažitelné.

4.1.1 Konečný překladový automat

Až do této části je stavba kompresního i dekompresního automatu totožná. Změna nastává až při posledním kroku tohoto procesu, kterým je převod automatu vzniklého algoritmem *L-automaton* na konečný překladový automat (def. 2.0.17). Tato transformace je pro nás přínosná v tom, že během komprese i dekomprese je nám automat schopný při přesunu do dalšího stavu rovnou vrátit překlad vstupního textu. Překladem se v tomto případě rozumí zakódovaná nebo dekódovaná část textu. Nejdříve popíšu, jak tato operace vypadá v případě konstrukce kompresního automatu.

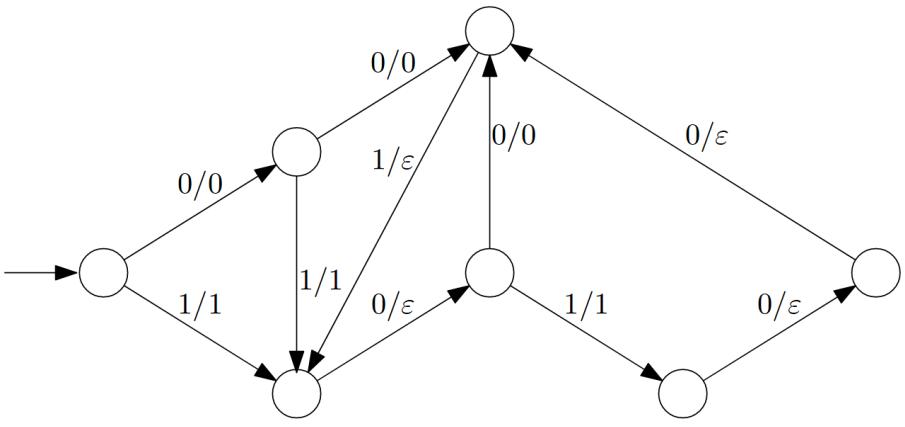
Předpokládejme, že z automatu byly odebrány stavy reprezentující antislova pro text T, jak říká poznámka 4.1.1. Poté mohou pro každý stav $p \in Q$ nastat pouze dvě situace:

- (1.) Pro stav p platí, že $(\forall a \in A) : \exists \delta(p, a)$. V tomto případě nastavíme pro oba tyto přechody překlad $\delta(p, a)$=$\{a\}$.
- (2.) Pro stav p je definován přechod pouze pro jeden symbol $a \in A$. To znamená, že druhý dříve vedl do stavu reprezentujícího některé z antislov. Tím pádem kodér při čtení vstupu ví o jaký symbol se jedná a

vstup strom $\mathcal{T} = (Q, A, \delta', q_0, F)$
výstup automat $M = (Q, A, \delta, q_0, Q \setminus F)$
funkce:
1: **for** každý $a \in A$ **do**
2: **if** existuje $\delta'(q_0, a)$ **then**
3: nastav $\delta(q_0, a) = \delta'(q_0, a)$
4: nastav $f(\delta(q_0, a)) = q_0$
5: **else**
6: nastav $\delta(q_0, a) = q_0$
7: **end if**
8: **end for**
9: **for** každý stav $p \in Q \setminus q_0$ postupem do šířky **and** každý $a \in A$ **do**
10: **if** existuje $\delta'(p, a)$ **then**
11: nastav $\delta(p, a) = \delta'(p, a)$
12: nastav $f(\delta(p, a)) = \delta(f(p, a))$
13: **else if** $p \notin F$ **then**
14: nastav $\delta(p, a) = \delta(f(p, a))$
15: **else**
16: nastav $\delta(p, a) = p$ {Smyčka v uzlu reprezentujícím antislovo}
17: **end if**
18: **end for**
19: **return** M

Algoritmus 4: L-automaton

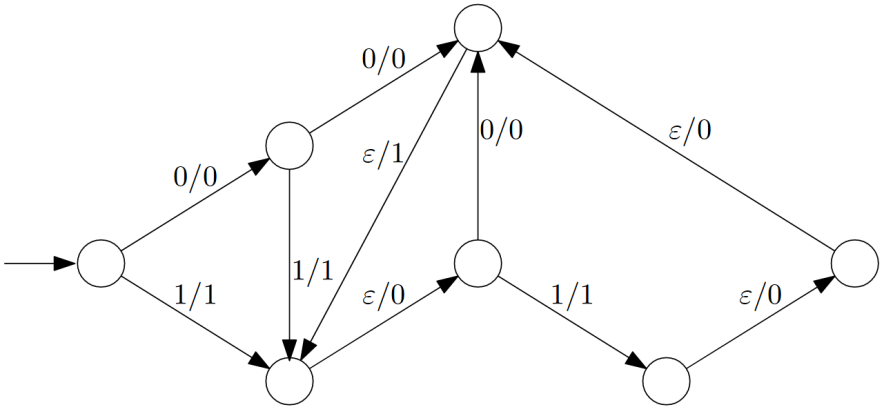
je proto možné ho z výstupu (neboli překladu) vynechat. Překlad pro existující $\delta(p, a)$ nastavíme na symbol reprezentující prázdný řetězec ε.



Obrázek 4.2: Hotový kompresní automat pro antislovník {000,11,10101} [10]

Jediným rozdílem mezi kompresním a dekompresním automatem je, že

přechodový symbol a překlad jsou všude prohozené. Toho si lze všimnou na obrázcích 4.2 a 4.3, kde jsou tyto zobrazeny hotové zkonstruované automaty. Automaty na obrázcích vznikly pomocí antislovníku {000,11,10101}, stejně jako na předchozím příkladu.



Obrázek 4.3: Hotový dekompresní automat pro antislovník {000,11,10101} [10]

Kapitola 5

Implementace

5.1 Java

Metoda DCA byla implementována v programovacím jazyce Java, verzi 8. Ten byl i jednou z podmínek definovaných v zadání práce a je v něm naprogramován základ knihovny SCT [2] včetně metody ACB. V javě nebyla implementace DCA doposud dostupná, což byl jeden z dalších důvodů této volby. Pro proces vývoje jsem zvolil vývojové prostředí NetBeans IDE 8.2 [11], především díky jeho předchozí znalosti a dobré podpory javy.

5.2 Verzovací systém

Během implementace byl použit verzovací systém GIT [12], konkrétně repozitář umístěný na fakultním serveru gitlab [3]. V tomto repozitáři je umístěna celá knihovna knihovna SCT, jejíž součástí je nyní má implementace. Vývoj probíhal v samostatné větvi s názvem DCA, aby nevznikaly konflikty s ostatními studenty přispívajícími do této knihovny.

5.3 Maven

Pro sestavování (build) knihovny byl použit nástroj Maven, který především umožňuje jednoduché sestavení projektu a zjednodušuje správu dependencí na externích knihovnách. Byly použity externí knihovny

- **Apache Log4j2** [13] – Jedná se o jednu z mnoha javovských logovacích utilit. Nabízí kvalitní a jednoduchou správu logů, které program vytváří za svého běhu. Jedna z výhod logovací utility místo běžnýc výpisů je i ta, že umožňuje omezování úrovně (levelu) logování. Tím pádem je možné například vypínat a zapínat debugovací výpisy.

- **Commons CLI** [14] – Tato knihovna byla využita při tvorbě konzolového uživatelského prostředí. Umožňuje jednoduché zpracování parametrů zadaných při spouštění přes příkazovou řádku a také nabízí možnost vypsat podrobnou *help* zprávu, aby uživatel věděl, jak má být program používán.

5.4 Adresářová struktura

Struktura knihovny je navržena tak, že se pro každou kompresní metodu vytvoří nový „package“ (adresář), ve kterém je její implementace. Části kódu, které jsou mezi některými metodami sdílené (např. zpracování vstupu/výstupu), jsou pak ve svých vlastních složkách. Metoda DCA tedy vznikla v adresáři s názvem dca. Jeho obsah vypadá následovně:

```
antidictionary
    Antidictionary.java ..... třída reprezentující antislovník
    AntidictionaryBuilder.java ..... ze vstupu sestaví antislovník
    AntidictionaryNode.java ..... třída reprezentující uzel suffix trie
    ForbiddenWord.java .. reprezentuje jedno antislovo v antislovníku
    NodeType.java ..... enum pro typ uzlu v suffix trie
automaton
    DecodingTransducer.java ..... dekódovací automat
    EncodingTransducer.java ..... kódovací automat
    StateLink.java ..... přechod mezi stavy automatu
    Symbol.java ..... enum pro symbol používaný v automatu
    TransducerBase.java ..... abstraktní třída pro společné rysy
      kodéru/dekodéru
    TransducerState.java ..... třída reprezentující stav automatu
core
    BitInputStreamReader.java ..... zpracovává vstup po bitech
    BitOutputStreamWriter.java ..... zpracovává výstup po bitech
    DCACoder.java ..... zpracovává kódování a dekódování
    DCAHeader.java ..... vytváří metadata metody DCA
    NodeBase.java ... abstraktní třída pro společné rysy Node a State
DCA.java ..... hlavní třída volající kompresi a dekompresi
DCAClient.java ..... konzolové uživatelské rozhraní
DCAParams.java ..... parametry metody DCA
DCATest.java ..... třída sloužící pro testovací účely
```

Obrázek 5.1: Adresářová struktura metody DCA

5.5 Metadata

Aby bylo možné zkomprimovaný soubor zpětně dekomprimovat, je nutné přidat během komprese na jeho začátek hlavičku s metadaty. Pro dekompresi je potřeba znát původní velikost souboru, způsob provedení komprese a obsah antislovníku použitého při kompresi.

Strukturu hlavičky jsem navrhl následovně. Nejdříve hlavička obsahuje tyto informace:

velikost	**popis**
4 byty	původní velikost souboru v bitech
4 byty	počet antislov v antislovníku
1 bit	každý segment zakódován s vlastním antislovníkem (1 = true)

Význam poslední informace je vysvětlen na v sekci 5.7. Dále je nutné do hlavičky zaznamenat obsah antislovníku. Každé antislovo se tedy do souboru uloží jako:

velikost	**popis**
6 bitů	délka antislova
délka antislova v bitech	řetězec, který tvoří toto antislovo

Antislovo 0011 je tedy v hlavičce uloženo jako 000100, což binárně reprezentuje číslo 4 (délku antislova), a hned poté následuje 0011 – samotné antislovo. Všechny antislova na sebe bez zbytečného paddingu navazují. Antislovo a tedy v hlavičce zabere $|a| + 6$ bitů paměti.

5.6 Vstup a výstup

Metoda DCA vyžaduje zpracování vstupního řetězce po jednotlivých bitech a stejným způsobem poté produkuje výstup. Implementoval jsem tedy třídy `BitInputStreamReader` a `BitOuputStreamWriter`, které navazují na kód pro zpracování vstupu a výstupu z knihovny. Třídy jsem naimplementoval jako singletony[2], aby se daly pohodlně používat z více míst a umožňovaly přitom zápis/čtení do/ze stejného bufferu. To se hodí především v situacích, kdy zápis/čtení skončí uprostřed jednoho bytu.

[2]Třída s jednou instancí vzniklou privátním konstruktorem a statickou funkcí pro získání této instance

5.7 Segmentování vstupu

Načítání souboru po menších částech původně sloužilo pouze k ušetření operační paměti při běhu programu. Vzhledem k tomu, že metoda DCA je velmi náchylná na vstupní data, rozhodl jsem se s metodou experimentovat a vytvořil verzi, která vstupní soubor rozdělí na požadovaný počet částí a každou část zakóduje, jako kdyby se jednalo o samostatný soubor. Tyto části potom spojí za sebe a vytvoří jeden výsledný soubor. Rozdíl od běžného způsobu použití DCA je v tom, že pro každou takovou část se vytváří speciální antislovník.

Tento přístup může přinést výhody i nevýhody. Výhody se objeví hlavně v situacích, kdy se struktura vstupních dat hodně mění a vznikají tím pádem pouze dlouhá a nevhodná antislova. Taková data dělají metodě DCA velký problém a není nezvyklé, že komprese je téměř nulová, ne-li negativní. Rozdělením těchto dat na menší části se můžeme těmto změnám ve struktuře vyhnout a částečně je i využít.

Nevýhodou je samozřejmě to, že hlavičku obsahující antislovník musíme do výsledného souboru uložit hned několikrát. Pokud jsou data pro metodu DCA natolik nevhodná (tzn. velmi náhodná), že i segmentování vstupu velký počet dlouhých antislov, dojde logicky k ještě většímu zhoršení komprese. Ani o jedné možnosti tedy nelze říct, že je tou lepší, jelikož výsledek závisí zcela na struktuře komprimovaných dat.

5.8 Parametry DCA

O parametry metody DCA se stará DCAParams, která je pak distribuuje do potřebných míst. Obsahuje především parametry:

- **maximální délka antislova** – Tento parametr ovlivňuje délku faktorů textu prohledávaných s cílem nalézt minimální zakázaná slova pro antislovník.

- **komprese segmentů** – Boolean hodnota určující, zda má být (případně byla) použita verze metody, která komprimuje (případně dekomprimuje) každý segment jeho vlastním antislovníkem.

- **velikost segmentů** – Velikost jednoho segmentu v bytech.

Dále se v této třídě udržuje informace o tom, kolik bitů použít na zakódování velikosti každého antislova. Tento parametr je implicitně nastaven na hodnotu 6 a měnit by se měl jen v extrémních případech, kdy je pro to dobrý důvod (šetření každého bitu při menších parametrech nebo maximální délka antislova větší než 63).

5.9 Klient

Uživatelské rozhraní bylo vytvořené pro použití přes příkazovou řádku. Poskytuje spuštění metody DCA v režimu komprese i dekomprese a samozřejmě umožňuje i nastavení všech výše zmíněných parametrů.

Klient je navržen tak, aby při každém špatném použití automaticky vypsal nápovědu, která uživatele navede k opravení chyby. Tento výpis je také možné vyvolat přepínačem `-h,--help`. Pomocný výpis vypadá takto:

```
usage: dca input output [options]
input - input file or directory
 -b,--buffer-size <N>       When processing file, each
                            partition will have N bytes
                            (default is 1 000 000)
 -de,--decompress           decompress input
                            (default is to compress)
 -h,--help                  print this help
 -l,--length <N>            Maximal forbidden word length
                            will be set to N
                            (default is 20)
 -log,--log-level <level>   sets logging level of the application
 -m,--measure               measured program process data
                            printed to standard output
                            (default: be silent)
 -p,--partitioning          antidictionary is created
                            for each file partition
                            (default creates antidictionary
                            from the whole file)
```

Pro spuštění klienta je tedy potřeba zadat 2 povinné parametry – vstupní a výstupní soubor. Zbytek parametrů je volitelný (mají implicitní hodnoty). Nastavením úrovně logovací utility klient umožňuje vypínat nebo zapínat některé informační a debugovací výpisy. Zapnutí s přepínačem `-m,--measure` způsobí, že program po dokončení požadované operace (komprese nebo dekomprese) vypíše celkovou dobu jejího běhu a také kompresní poměr spočítaný jako podíl velikostí vstupního a výstupního souboru.

5.10 Průběh metody

Komprese i dekomprese začínají inicializací třídy `DCAParams`, která se stará o parametry DCA. Slouží jako argument konstruktoru pro třídu `DCA`, která na základě těchto parametrů inicializuje potřebné struktury. Třída `DCA` obsahuje metody `createAntidictionary`, `compress`, `compressByPartitions` a `decompress`. Tyto metody jsou navržené pro používání třídou `ChainBuilder`,

která slouží k řetězení operací. Aby řetězení fungovalo správně, je potřeba, aby si navzájem odpovídaly vstupní a výstupní parametry po sobě jdoucích metod. Příklad zavolání komprese vypadá následovně:

```
ChainBuilder.create(io::openParse)
        .chain(dca::createAntidictionary)
        .accept(inputFile.toPath());

ChainBuilder.create(io::openParse)
        .chain(dca::compress)
        .end(buffer -> io.saveParsed(buffer, outputFile.toPath()))
        .accept(inputFile.toPath());
```

kde `io` je instance třídy `FileIO`, starající se o zpracování vstupu a výstupu, `dca` je instance třídy `DCA`, inicializovaná nějakými parametry `DCAParams`, `inputFile` je vstupní soubor a `outputFile` je výsledný zkomprimovaný soubor.

Na ukázce tohoto kódu je zřejmé, že se vstupní soubor musí přečíst 2x, jednou za účelem vytvoření antislovníku, podruhé dojde na základě vytvořeného antislovníku ke kompresi vstupního souboru. V případě, kdy uživatel chce zakódovat každý segment souboru jeho vlastním antislovníkem, nastaví v instanci třídy `DCAParams` atribut `compressByPartitions` na *true*. Může také upravit atribut `bufferSize` (velikost segmentu v bytech) s tím, že pokud tak neučiní, použije se implicitní hodnota 1 MB. Program se poté spustí:

```
ChainBuilder.create(io::openParse)
        .chain(dca::compressByPartitions)
        .end(buffer -> io.saveParsed(buffer, outputFile.toPath()))
        .accept(inputFile.toPath());
```

Použití dekomprese je vždy stejné, nezávisle na tom, jakým způsobem proběhla komprese. Veškeré informace si program načte z metadat uložených v hlavičce souboru (případně z hlavičky každého segmentu). Zavolání dekomprese na zkomprimovaný soubor *compressedFile* vypadá takto:

```
ChainBuilder.create(io::openParse)
        .chain(dca::decompress)
        .end(buffer -> io.saveParsed(buffer, originalFile.toPath()))
        .accept(compressedFile.toPath());
```

kde `originalFile` je zpětně dekomprimovaný původní soubor.

Kapitola 6

Výsledky

Během testování metody DCA byly použity korpusy Canterbury a Prague. Korpusy slouží k testování kompresních algoritmů a porovnávání jejich výsledků s ostatními algoritmy. Mezi algoritmy se na nich porovnává především kompresní poměr, jelikož na rozdíl od rychlosti komprese a dekomprese je nezávislý na hardwarovém vybavení testovacího zařízení.

Pro porovnatelnost časů jsem všechny měření provedl na jednom zařízení s následucjícími parametry:

- Procesor – AMD Phenom II X4 955 (3.20 GHz),
- RAM – 8 GB,
- Operační systém – Windows 7 (64 bitový).

6.1 Korpus Canterbury

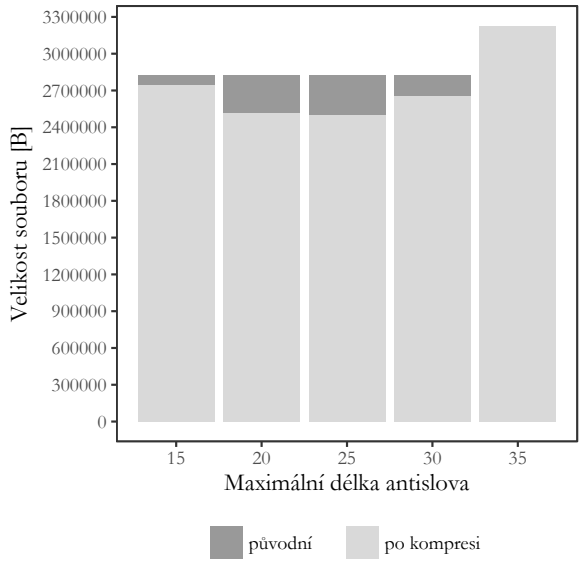
Korpus Canterbury obsahuje celkem 11 souborů o celkové velikosti 2,810,784 bytů. Vznikl jako náhrada za zastaralý korpus Calgary, aby lépe odpovídal moderním kompresním metodám. Obsah korpusu je blíže popsán v tabulce 6.1.

Komprese celého korpusu s použitím různých maximálních délek l pro antislova je znázorněna na obrázku 6.1. Z grafu je patrné, že upravováním parametru l můžeme kompresi zlepšit, ale také výrazně zhoršit.

To je způsobeno tím, že od určité velikosti l už algoritmus nachází převážně antislova nevhodná pro kompresi (jejich zapsání do hlavičky souboru zabere více paměti, než je jimi ušetřeno). V nejhorším případě může totiž antislovo ušetřit pouze jeden bit vstupního řetězce. Tomuto jevu by šlo zabránit například provedením pokusné komprese, při které by se pro každé antislovo změřila jeho efektivita (počet použití). Poté by se sestavil nový automat pouze z vhodných antislov, který by byl použit k finální kompresi. Toto řešení by zlepšilo kvalitu komprese na úkor rychlosti algoritmu.

Tabulka 6.1: Obsah korpusu Canterbury

Název	**Kategorie**	**Velikost [B]**
alice29.txt	Anglický text	152,089
asyoulik.txt	Shakespear	125,179
cp.html	zdrojový kód HTML	24,603
elds.c	zdrojový kód C	11,150
grammar.lsp	zdrojový kód LISP	3,721
kennedy.xls	Excelová tabulka	1,029,744
lcet10.txt	technický spis	426,754
plrabn12.txt	báseň	481,861
ptt5	testovací test CCITT	513,216
sum	SPARC program	38,240
xargs.1	GNU manuál	4,227



Obrázek 6.1: Komprimace korpusu canterbury

V tabulce 6.2 je zobrazena doba trvání komprese a počet vytvořených antislov v závislosti na maximální délce antislova. Při porovnání této tabulky s grafem je patrné, jak právě velký počet dlouhých antislov může mít negativní vliv na kompresní poměr.

Při kompresi celého korpusu je také vidět, že zde metoda DCA nedosahuje výrazných kompresních poměrů (nejlepší kompresní poměr 0,885 byl dosažen s parametrem $l = 25$). To může být vysvětleno tím, že DCA je velmi náchylná na změnu ve struktuře dat. V tomto případě se může jednat o to, že archiv ob-

Tabulka 6.2: Testování DCA na korpusu Canterbury

Parametr l	**Počet antislov**	**Délka komprese**
15	3,431	2,730 ms
20	42,003	4,586 ms
25	147,558	15,974 ms
30	266,165	44,914 ms
35	443,445	102,414 ms

sahuje více souborů, jejichž struktury se mohou velmi lišit. Antislovník je však vytvářen pro celý archiv najednou a není nijak zoptimalizovaný pro jednotlivé části vstupního souboru.

I to je jedním z důvodů, proč jsem vytvořil možnost komprese jednotlivých částí souboru. Přestože je potřeba provést hned několik zápisů metadat do zkomprimovaného souboru, umožňuje se tím vzniku kvalitnějších antislovníků menší části vstupního souboru, místo jednoho méně kvalitního pro celý soubor. Je tak tedy možné zlepšit kompresní poměr pro data s velkým objemem nebo pro data s nepravidelnou vnitřní strukturou.

6.2 Korpus Prague

Korpus Prague obsahuje celkem 30 souborů o celkové velikosti 58,265,600 bytů. Jedná se tedy o výrazně větší korpus v porovnání s Canterbury. Na rozdíl od něj obsahuje také několik binárních souborů, obrázků, audio souborů nebo databázových tabulek.

Kvůli velkému objemu a množství souborů je pro metodu DCA v základní podobě velmi obtížné tento korpus zmenšit. Při pokusu o kompresi tohoto korpusu s parametrem $l = 20$ nebylo nalezeno žádné antislovo. Naopak při zvětšování parametru se počet antislov rychle dostal na vysoká čísla, což vedlo k negativní kompresi. Použil jsem proto možnost segmentování 5.7 a dosáhl výsledků popsaných v tabulce 6.3.

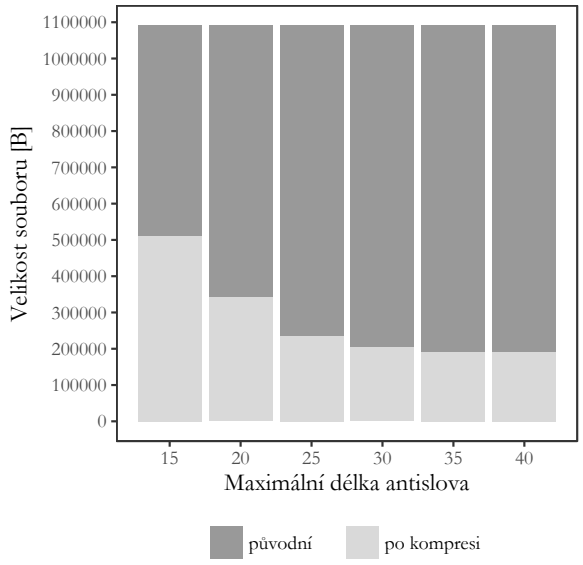
Tabulka 6.3: Testování DCA na korpusu Prague

Parametr l	**Segment**	**Velikost po kompresi**	**Čas**
15	1,500,000 B	56,521,102 B	75,629 ms
15	3,000,000 B	57,931,903 B	71,620 ms
17	1,500,000 B	55,588,245 B	122,741 ms
17	3,000,000 B	57,409,006 B	110,807 ms
20	1,500,000 B	59,692,955 B	291,767 ms
20	3,000,000 B	58,036,056 B	218,540 ms

Z tabulky 6.3 je patrné, že pro použití segmentů je vhodnější volit menší parametr l, jelikož chceme, aby vznikaly spíše menší antislovníky. Vytvoření velkého množství objemných antislovníku je samozřejmě nevýhodné i z časového hlediska.

6.3 Vhodná data

Z předchozích sekcí je zřejmé, že metoda DCA není vhodná ke kompresi libovolných souborů. Pro soubory s vysokou datovou entropií (např. obrázky nebo videa), se jeví dokonce jako velmi nevhodná. Nejlepších kompresních poměrů jsem dosáhl při komprimaci textových souborů, kde omezená sada znaků garantuje uspokojivý kompresní poměr. Pro ukázku 6.2 jsem vytvořil soubor o velikost 1,090,923 bytů, obsahující přes tisíc náhodně vygenerovaných odstavců textu *lorem ipsum*. Data byla vygenerována tak, aby se v textu žádné 2 odstavce neopakovaly. Jedná se tedy o soubor dobře reprezentující běžný text (např. kratší knihu).



Obrázek 6.2: Komprimace textového souboru

Na obrázku 6.2 je znázorněno, že textový soubor dokázala metoda DCA při použití parametru $l = 35$ zmenšit na 17,5 % původní velikosti. I přesto, že byl v tomto případě parametr l zvolen relativně velký, bylo vytvořeno pouze 3443 antislov. To je způsobeno právě malou entropií textového vstupu. Ta zároveň způsobila, že komprese trvala pouze 3572 ms, jelikož díky ní vznikl pouze velmi řídký suffix trie. Proto si myslím, že by tato metoda mohla být dobrou volbou například pro kompresi obsáhlých textových knihoven.

Závěr

Během této práce jsem se seznámil s kompresní metodou DCA a provedl její implementaci v programovacím jazyce Java tak, aby se stala součástí knihovny *Small Compression Toolkit*.

Výsledkem práce je funkční kompresní metoda DCA s některými optimalizacemi, spustitelná pomocí jednoduchého konzolového rozhraní s možností nastavení všech potřebných parametrů. Analýza a testování této metody ukázaly, že je vhodná především pro soubory s nízkou datovou entropií (např. textové soubory), kde dosahuje komprese přes 80 %. V opačných případech ji lehce předčí jiné kompresní metody. Do budoucna by tedy bylo vhodné do knihovny přidat i další verze DCA a doimplementovat její další vylepšení. Navzdory velkému prostoru k vylepšení věřím, že metoda DCA najde využití v rámci knihovny SCT a že i má práce může být užitečným materiálem pro autory dalších verzí a implementací.

Literatura

[1] Crochemore, M.; Mignosi, F.; Restivo, A.; aj.: Text Compression Using Antidictionaries. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*, ICAL '99, London, UK, UK: Springer-Verlag, 1999, ISBN 3-540-66224-3, s. 261–270. Dostupné z: https://dl.acm.org/citation.cfm?id=646229.681728

[2] Small Compression Toolkit [online]. [cit. 2018-04-25]. Dostupné z: https://gitlab.fit.cvut.cz/polacrad/sct

[3] GitLab Community Edition [online]. [cit. 2018-04-25]. Dostupné z: https://gitlab.fit.cvut.cz/help

[4] Bican, J.: *Implementace vylepšení kompresní metody ACB v jazyce Java*. Diplomová práce, České vysoké učení technické v Praze, 2017.

[5] Salomon, D.: *Data Compression: The Complete Reference*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006, ISBN 1846286026.

[6] Stringology, lexikon pojmů [online]. [cit. 2018-04-27]. Dostupné z: http://www.stringology.org/DataCompression/lexikon.html

[7] Melichar, B.: Automaty ve vyhledávání – cvičení [online]. [cit. 2018-04.05]. Dostupné z: https://edux.fit.cvut.cz/oppa/MI-AVY/cviceni/mi-avy-cviceni.pdf

[8] Crochemore, M.; Navarro, G.: Improved Antidictionary Based Compression. In *Proceedings of the XII International Conference of the Chilean Computer Science Society*, SCCC '02, Washington, DC, USA: IEEE Computer Society, 2002, ISBN 0-7695-1867-2, s. 7–. Dostupné z: http://dl.acm.org/citation.cfm?id=829496.829739

[9] Skalický, J.: *Aplikace kompresní metody DCA*. Diplomová práce, České vysoké učení technické v Praze, 2010.

[10] Holub, J.: Kontextové metody, DCA (8. přednáška) [online]. [cit. 2018-04.05]. Dostupné z: https://edux.fit.cvut.cz/courses/MI-KOD/_media/lectures/08/mi-kod-08-dca.pdf

[11] NetBeans IDE 8.2 [online]. [cit. 2018-05-06]. Dostupné z: https://netbeans.org/community/releases/82/

[12] Git [online]. [cit. 2018-06-05]. Dostupné z: https://git-scm.com

[13] Apache Log4j 2 [online]. [cit. 2018-05-06]. Dostupné z: https://logging.apache.org/log4j/2.x/

[14] Commons CLI [online]. [cit. 2018-05-06]. Dostupné z: https://commons.apache.org/proper/commons-cli/

Příloha **A**

Seznam použitých zkratek

MFW Minimal forbidden word – Minimální zakázané slovo

AD Antidictionary – Antislovník

SCT Small Compression Toolkit

Příloha **B**

Obsah přiloženého CD

Zdrojový kód metody DCA je k dispozici také v repozitáři knihovny na adrese: `https://gitlab.fit.cvut.cz/polacrad/sct/tree/DCA`.
Zdrojová forma textu práce ve formátu LaTeX je také dostupná na adrese: `https://gitlab.fit.cvut.cz/novak128/BP`.

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `BP_novak_jakub_2018.pdf` ..... text práce ve formátu PDF