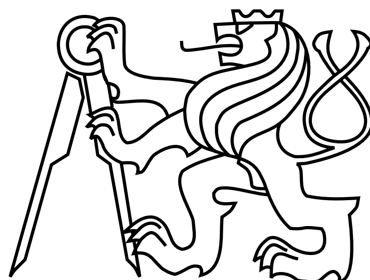
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Aplikace kompresní metody DCA

Bc. Jan Skalický

Vedoucí práce: doc. Ing. Jan Holub, Ph.D.

Studijní program: Elektrotechnika a informatika, strukturovaný, Navazující magisterský

Obor: Výpočetní technika

14. května 2010

Poděkování

Děkuji doc. Ing. Janu Holubovi, Ph.D. za vedení práce. Dále bych rád poděkoval firmě Princip, a. s., zejména jejímu vývojovému řiditeli Ing. Jaroslavu Altmannovi za poskytnutí hardwaru a Ing. Janu Krejsovi za konzultační čas ohledně jejich platformy, Petru Nohavicovi ohledně portování kódu a Filipu Šimkovi za pomoc při integraci kódu do knihovny ExCom. Dík patří rovněž všem, kteří moji práci připomínkovali a všem vývojářům svobodného software, jejichž nástroje mi napomáhaly v celém průběhu práce.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 14. 5. 2010 ....

Abstract

This thesis is concerned with applications of DCA compression method (Data Compression using Antidictionaries). The topic of the first of two processed applications is searching capabilities in a compressed text without the need of its decompression. The speed of such algorithm is determined by its limitation on work with compression metadata – an antidictionary or encoding automaton. The second part of the work is engaged in an illustrative employment of DCA compression on some practical instance. The compression algorithm is implemented to a suitable HW device.

Abstrakt

Tato práce se zabývá aplikacemi kompresní metody DCA (Data Compression using Antidictionaries). Tématem první ze dvou zpracovávaných aplikací je vyhledávání v zakomprimovaném textu bez nutnosti jeho dekomprese. Rychlost vzniklého algoritmu je určena jeho omezením na práci s kompresními metadaty – antislovníkem nebo kódovacím automatem. Druhá část práce se zabývá ilustrativní ukázkou nasazení DCA komprese na praktickém příkladě. Kompresní algoritmus je zde implementován do vhodně zvoleného HW zařízení.

Obsah

1 Úvod **1**
- 1.1 Popis problému . . . 1
- 1.2 Motivace . . . 1
- 1.3 Související práce . . . 2
- 1.4 Specifikace cílů . . . 2
  - 1.4.1 Vyhledávání v zakomprimovaném textu . . . 2
  - 1.4.2 Nasazení DCA v HW zařízení . . . 3

2 Formalismus **5**
- 2.1 Definice pojmů . . . 5
  - 2.1.1 Stringologie . . . 5
  - 2.1.2 Konečné automaty . . . 7
  - 2.1.3 Komprese dat . . . 10

3 Kompresní metoda DCA **13**
- 3.1 DCA kódování . . . 14
- 3.2 DCA dekódování . . . 16
- 3.3 Konstrukce antislovníku . . . 18
  - 3.3.1 Suffix trie . . . 18
  - 3.3.2 Selekce minimálních AW . . . 19
  - 3.3.3 Omezení délky AW . . . 20
- 3.4 Dodatky k DCA . . . 21
  - 3.4.1 DCA pomocí KA . . . 21
  - 3.4.2 Kompresní schémata DCA . . . 21
    - 3.4.2.1 Semi-adaptivně . . . 21
    - 3.4.2.2 Staticky . . . 22
    - 3.4.2.3 Dynamicky (adaptivně) . . . 22
  - 3.4.3 Prořezávání AD . . . 22
  - 3.4.4 Almost antiwords . . . 23

4 Analýza a návrh řešení **25**
- 4.1 Negativní odpověď . . . 25
  - 4.1.1 Vyhledávací automat . . . 26
  - 4.1.2 Faktorový automat . . . 28
- 4.2 DCA automat . . . 29

4.2.1 DCA kompresní automat 29
4.2.2 DCA dekompresní automat 29
4.2.3 Použití DCA automatu 30
4.3 Pozitivní odpověď 31
4.3.1 Předpony minimálních antislov 32
4.3.2 Faktory minimálních antislov 32
4.3.3 Kombinovaný věštící automat 33
4.4 Problém zarovnání dat 35
4.4.1 Zarovnání v praxi 35

5 Realizace **37**
5.1 Implementace 37
5.1.1 Programovací a vývojové prostředí 37
5.1.2 Rešerše knihoven pro práci s KA 38
5.1.2.1 AT&T FSM Library 38
5.1.2.2 Grail 38
5.1.3 Návaznost na implementaci DCA 39
5.1.4 Implementace vyhledávání 39
5.1.4.1 Třída bitstring 40
5.1.4.2 Třída ADsearcher 41
5.1.4.3 Chybová odpověď 42
5.1.5 Integrace vyhledávání do ExCom 43
5.1.6 Nástroj search_cli 43
5.2 Testování 44
5.2.1 Metodika testování 44
5.2.2 Pomocné nástroje 46
5.2.3 Testovací řetězec 46
5.2.4 Výsledky experimentů 47
5.2.4.1 Grafy výsledků a jejich interpretace 47
5.3 Zhodnocení 66

6 DCA v HW zařízení **67**
6.1 HW platforma 67
6.1.1 Výběr zařízení 67
6.1.2 Popis zařízení 68
6.2 Kompresní schéma 68
6.3 Implementace 69
6.3.1 Program dcaxgen 70
6.3.2 Program sdca 71
6.3.2.1 Výjimky 71
6.3.2.2 Kódování výjimek 72
6.3.2.3 Standardizační mezivrstva 74
6.4 Nasazení 76
6.4.1 Testy v zařízení 77

7 Závěr **81**

Literatura **83**

A Slovník zkratek **85**

B Obsah přiloženého CD **87**

C Použití vybraných programů **89**

C.1 search_cli 89

C.2 dcaxgen 90

C.3 sdca 90

D Fotografie Vetronicsu **91**

E Originální specifikace Vetronicsu **93**

Seznam obrázků

1.1 Schéma kompresního procesu a vyhledávání nad kompresními metadaty . . . 3

2.1 Příklad znázornění přechodového diagramu KA 8
2.2 Příklad znázornění přechodového diagramu PKA 9

3.1 Úplná suffix trie pro řetězec ‘1101’ 20

4.1 Základní vyhledávací automat 26
4.2 Vyhledávací automat pro více vzorků 26
4.3 Determinizovaný automat z obr. 4.1 27
4.4 Automat přijímající předpony přípon 28
4.5 Faktorový automat 28
4.6 DCA kompresní automat 29
4.7 DCA dekompresní automat 30
4.8 Nástin automatu pro pozitivní odpověď na základě předpon AW 32
4.9 Nástin automatu pro pozitivní odpověď na základě faktorů AW 33
4.10 Opravený automat pro kombinovanou odpověď 34
4.11 Problém zarovnání dat – neurčitá odpověď 36
4.12 Problém zarovnání dat – pozitivní odpověď 36

5.1 Class diagram třídy bitstring 40
5.2 Vložení hlavičky třídy bitstring ostatními moduly 41
5.3 Kolaborační diagram třídy ADsearcher 42
5.4 Volací diagram programu search_cli 44
5.5 Histogram délky antislov - malé soubory 48
5.6 Histogram délky antislov - střední soubory 49
5.7 Histogram délky antislov - velké soubory 50
5.8 Výskyt neurčité odpovědi na náhodných vzorcích - malé soubory 51
5.9 Výskyt neurčité odpovědi na náhodných vzorcích - střední soubory 52
5.10 Výskyt neurčité odpovědi na náhodných vzorcích - velké soubory 53
5.11 Výskyt pozitivní odpovědi na náhodných vzorcích - malé soubory 54
5.12 Výskyt pozitivní odpovědi na náhodných vzorcích - střední soubory 55
5.13 Výskyt pozitivní odpovědi na náhodných vzorcích - velké soubory 56
5.14 Výskyt neurčité odpovědi na vzorcích obsažených v textu - malé soubory . . . 57
5.15 Výskyt neurčité odpovědi na vzorcích obsažených v textu - střední soubory . 58
5.16 Výskyt neurčité odpovědi na vzorcích obsažených v textu - velké soubory . . 59

5.17 Počet výsledek rozhodujících AW pro negativní odpověď - malé soubory . . . 60
5.18 Počet výsledek rozhodujících AW pro negativní odpověď - střední soubory . . 61
5.19 Počet výsledek rozhodujících AW pro negativní odpověď - velké soubory . . . 62
5.20 Počet výsledek rozhodujících AW pro pozitivní odpověď - malé soubory . . . 63
5.21 Počet výsledek rozhodujících AW pro pozitivní odpověď - střední soubory . . 64
5.22 Počet výsledek rozhodujících AW pro pozitivní odpověď - velké soubory . . . 65

6.1 Schéma kompresního procesu implementovaného statického DCA 69
6.2 Kolaborační diagram třídy DCAxgen 70
6.3 Volací diagram programu sdca 74
6.4 Závislosti zdrojového souboru kompresního modulu 75
6.5 Vložení hlavičky config.h – volba platformy 75
6.6 Závislost velikosti sDCA překladače na hloubce AD 78
6.7 Kompresní poměry sDCA na testovacích datech, velikosti výjimek 79
6.8 Histogram výskytu výjimek v sDCA na datech podobných trénovacím 80

Seznam tabulek

3.1 Simulace DCA kodéru . . . 16
3.2 Simulace DCA dekodéru . . . 17
3.3 Začátek konstrukce suffix trie pro $T =$ '11011010110' . . . 19

5.1 Soubory v Canterbury Corpusu . . . 45
5.2 Velikosti antislovníků souborů z Canterbury Corpusu . . . 47

6.1 Počet stavů DCA automatu při prořezání AD (testdata1.log) . . . 71
6.2 Kódy s pohyblivou délkou – základní . . . 72
6.3 Kódy s pohyblivou délkou – Elias, Fibonacci, Golomb, Rise . . . 73
6.4 Soubory v Calgary Corpusu . . . 76

Seznam algoritmů

3.1 DCA kodér 15
3.2 DCA dekodér 17

Kapitola 1

Úvod

1.1 Popis problému

Kompresní metoda DCA (Data Compression using Antidictionaries) je relativně nová kompresní metoda prezentovaná M. Crochemorem [20, 21] na přelomu minulého století. Narozdíl od slovníkových metod (např. LZW), které pracují se slovníkem jako s množinou často používaných frází kódovaných vhodně zkrácenými obrazy, DCA pracuje naopak s řetězci nepoužívanými – tzv. „antislovníkem“. *Antislovník* (dále AD) je množina řetězců, které se v komprimovaném textu nevyskytují, a tak poskytují kódujícímu prostředku možnost některé informační jednotky (typicky bity) nekopírovat do výstupu, protože jejich hodnoty jsou deterministicky *predikovatelné*.

Dekodér této metody pak provádí činnost inverzní – na místech v datovém toku, kde kodér vynechával bity, si je dekodér obnovuje s jejich původní hodnotou, kterou lze určit opět z AD. Obě strany kompresního/dekompresního procesu tedy potřebují AD pro svoji činnost. Informace v AD tvoří jakási *metadata*, která jsou vhodným způsobem kódována v zakomprimovaném souboru spolu s vlastnímy daty.

Vzhledem k tomu, že metoda DCA je relativně nová, existuje zde prostor pro řadu aplikací, které nejsou dosud známé, otestované či nasazené. Dvě z nich, které jsme vynašli v souvislosti s činností *stringologické výzkumné skupiny* katedry počítačů FEL ČVUT (později katedry teoretické informatiky FIT ČVUT), chceme zpracovat jako téma této práce.

1.2 Motivace

Korelace obsahu AD s charakterem komprimovaného textu je z předchozích odstavců zřejmá. Je zajímavé zabývat se otázkou, kolik informace o původním textu se při DCA kompresi dostalo do jejích metadat. To záleží také na některých parametrech kompresního procesu – na hloubce komprese (max. počet bitů antislov) a na velikosti paměti, která je kompresnímu algoritmu k dispozici. Je dokonce možné, že v některých případech informace v kompresních metadatech postačí k *orientačnímu vyhledávání* v datech původních. Takové vyhledávání bude „věštící“ (angl. “oracle”) – nebude schopno vždy jednoznačně odpovědět,

zda-li se vzorek v původním textu nacházel, ale někdy bude schopno např. jeho výskyt vyloučit. Použitelnost takového přístupu v této práci prověříme.

Kompresní metoda DCA má i své odpůrce, kteří jí vytýkají zejména nízký kompresní poměr (def. 2.46). Na druhou stranu je to z podstaty mechanismu komprese metoda nenáročná na výpočetní výkon (kompresi může provádět konečný automat (def. 2.19)) a ve verzi se statickým schématem (def. 2.42) by neměla být příliš náročná ani na spotřebu paměti. Toto jsou vlastnosti, které ji předurčují k nasazení v různých menších HW aplikacích, *embedded systémech* apod. Potenciální přínosnost takového nasazení budeme zjišťovat.

1.3 Související práce

Implementaci kompresní metody DCA, na kterou budeme v této práci částečně navazovat, provedl ve své diplomové práci na této katedře Martin Fiala [23]. Tato implementace byla později přidána do univerzální kompresní knihovny *ExCom* (Extensible Compression Library), která vznikla na stejné katedře v rámci diplomové práce Filipa Šimka [26]. Obecným vyhledáváním v textu se zabývá řada předchozích prací a učebnic, např. [31], a konkrétně věštícími automaty[1] např. [16, 17]. Vyhledávání v textu simulací nedeterministických konečných automatů zpracoval ve své disertační práci Jan Holub [24].

1.4 Specifikace cílů

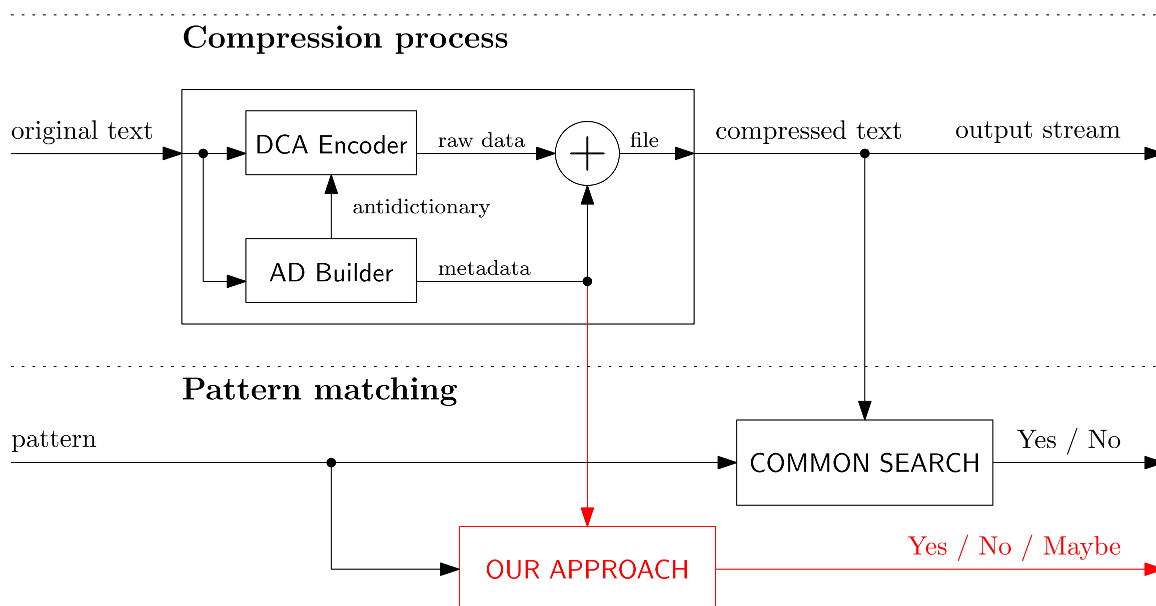
První část práce si klade za cíl zkonstruovat a prověřit možnosti využití věštícího automatu při vyhledávání vzorku v zakomprimovaném textu. Druhá část práce se zabývá ilustrativní ukázkou nasazení DCA komprese na praktickém příkladě.

1.4.1 Vyhledávání v zakomprimovaném textu

Existuje častý požadavek vyhledávání vzorku uvnitř textu – dat nad definovanou abecedou (def. 2.1) – a v případě textu zakomprimovaného je triviálním řešením této úlohy text dekomprimovat a nasadit známé algoritmy na vyhledávání. Lze ale tyto algoritmy modifikovat i pro použití přímo na textu zakomprimovaném a ušetřit tak čas a paměť potřebnou k jeho dekompresi, která při nenalezení vzorku navíc není potřeba. Takové modifikace jsou známé [37, 28]. Narozdíl od tohoto postupu, kdy se v zakomprimované podobě zpracuje celý text, naše práce se zabývá možností pracovat pouze nad kompresními metadaty – hledat v AD. Schéma tohoto přístupu vidíme na obr. 1.1.

Je jasné, že takové omezení není z principu schopno poskytnout pro libovolný vzorek závaznou informaci o jeho výskytu/nevýskytu. V některých případech však tuto otázku pouze na základě znalosti metadat zodpovědět lze a v ostatních situacích bude výstupem algoritmu odpověď neurčitého charakteru. Výhodou zpracování *pouze metadat* je rychlost, protože se

[1] automaty, jejichž odpověď není vždy určitá; mohou sjednocovat více typů možných výsledků



Obrázek 1.1: Schéma kompresního procesu a vyhledávání nad kompresními metadaty

pracuje s menším objemem dat než obsahuje celý text. Celkovou použitelností tohoto přístupu se zabývá první část práce, která provede jeho analýzu, realizaci v rámci existující implementace DCA [23], následné testování pomocí experimentů nad reálnými daty a konečné zhodnocení, jde-li o techniku přínosnou či nikoliv.

Tato část práce je tedy rozdělena do kapitoly analytické, implementační a testovací.

1.4.2 Nasazení DCA v HW zařízení

Hlavní výhoda kompresní metody DCA spočívá v její dekompresní rychlosti a relativní jednoduchosti. Obecně se jedná se o metodu asymetrickou (def. 2.39) – náročnou konstrukci AD provádí kodér – a v případě její verze se semi-adaptivním schématem (def. 2.43) je její dekomprese (v případě statické dokonce i komprese) úlohou pro jeden konečný automat, který se navíc dá jednoduše vyrobit z informací získaných během komprese. Vlastní dekompresi může zastat i vcelku jednoduchý hardware s malým výpočetním výkonem (jednočipový mikrokontrolér, hradlové pole apod.).

Druhá část práce se zabývá právě ukázkou nasazení DCA komprese na takovém příkladě, kdy malá kompresní režie není překážkou, a poměrně nenáročná komprese může pomoci ušetřit místo v úložišti nebo pásmo komunikačního kanálu v konkrétní aplikaci s omezenými prostředky. Tato část se tedy skládá z nalezení a seznámení se s vhodnou hardwarovou platformou, výběru kompresního schématu, který bude použit, jeho vlastní implementace a nakonec možnostmi nasazení DCA komprese v praxi.

Druhá část práce je koncipována jako implementační s otestováním na reálných datech.

Kapitola 2

Formalismus

2.1 Definice pojmů

2.1.1 Stringologie

Definice 2.1 (Abeceda)
Abeceda je konečná neprázdná množina symbolů. Velikost abecedy Σ značíme $|\Sigma|$.

Definice 2.2 (Binární abeceda)
Binární abeceda Σ je abeceda, pro kterou platí $|\Sigma| = 2$.

Poznámka 2.3 (Binární abeceda)
Binární abeceda používaná ve výpočetní technice typicky obsahuje prvky '0' a '1'. Pojmem binární abeceda v této práci budeme rozumět tuto množinu.

Poznámka 2.4 (Symbol)
Symbol je libovolný prvek abecedy.

Definice 2.5 (Komplement symbolu)
Komplement symbolu a v abecedě Σ je množinový doplněk k a v Σ. Značíme ho $\bar{a} = \Sigma \setminus \{a\}$.

Poznámka 2.6 (Komplement symbolu v binární abecedě)
V binární abecedě je komplement symbolu určen jednoznačně (jeho výsledkem je jediný prvek, resp. množina s mohutností 1). Typicky pro $\Sigma = \{\text{'0'}, \text{'1'}\}$ je komplementem symbolu '0' symbol '1' a naopak. Této jednoznačnosti se v DCA kompresi využívá.

Definice 2.7 (Řetězec)
Řetězec (řetěz) nad abecedou Σ je konečná posloupnost symbolů z abecedy Σ. Řetězec vypsaný po symbolech je zvykem uzavírat do jednoduchých, event. dvojitých (v případě, že obsahuje znaky mezer) uvozovek.

Definice 2.8 (Délka řetězce)
Délka řetězce $|x|$ je délka posloupnosti symbolů, ze kterých se skládá. Speciálně pro $|x| = 0$ (prázdná posloupnost symbolů) řekneme, že x je *prázdný řetězec* a značíme ho ε.

Poznámka 2.9 (Množina všech řetězců)
Množinu všech řetězců nad abecedou Σ značíme Σ^*. Libovolný řetězec x nad abecedou Σ značíme relací $x \in \Sigma^*$.

Definice 2.10 (Jazyk)
Jazyk L nad abecedou Σ je libovolná podmnožina množiny Σ^*, t.j. $L \subset \Sigma^*$. Prvek jazyka se nazývá *slovo*.

Definice 2.11 (Zřetězení)
Zřetězení je multi-ární operace s řetězci nad abecedou Σ, při které je posloupnost symbolů následujícího operandu přidána na konec posloupnosti předchozího. Zřetězení řetězců x a y zapisujeme ve tvaru xy či $x.y$ (s tečkou uprostřed).

Poznámka 2.12 (Zřetězení – vlastnosti)
Zřetězení je operace asociativní, nekomutativní. Dále platí $x\varepsilon = x$ a $\varepsilon x = x$. Zřetězení $n \geq 0$ stejných symbolů x značíme zkraceným tvarem x^n, např. $a^0b^1c^2d^3 = \varepsilon bccddd = bccddd$.

Definice 2.13 (Podřetězec)
Podřetězec (podřetěz, faktor) x řetězce $s \in \Sigma$ je řetězec, pro který platí $s = uxv$, kde $u, x, v \in \Sigma^*$.

Definice 2.14 (Předpona (Přípona))
Předpona (přípona) x řetězce $s \in \Sigma$ je podřetězec řetězce s, pro který platí navíc $u = \varepsilon$ ($v = \varepsilon$).

Definice 2.15 (Vlastní podřetězec)
Vlastní podřetězec x řetězce s je podřetězec řetězce s, pro který platí $x \neq s$.

Poznámka 2.16 (Vlastní předpona (přípona))
Analogicky k definici 2.15 použijeme adjektivum „vlastní" i pro předponu (příponu).

Poznámka 2.17 (Množina všech podřetězců)
Množinu všech podřetězců řetězce s značíme $Fact(s)$.
Množinu všech předpon řetězce s značíme $Pref(s)$.
Množinu všech přípon řetězce s značíme $Suff(s)$.
Množinu všech vlastních podřetězců řetězce s značíme $PFact(s)$.
Množinu všech vlastních předpon řetězce s značíme $PPref(s)$.
Množinu všech vlastních přípon řetězce s značíme $PSuff(s)$.

Pro více řetězců (např. pro jazyk) vzniknou tyto množiny jako sjednocení množin pro jednotlivé řetězce (např. pro slova jazyka).

Definice 2.18 (Antifaktor)
Řekneme, že řetězec x je *antifaktor* řetězce s, pokud $x \notin Fact(s)$.

2.1.2 Konečné automaty

Definice 2.19 (Konečný automat)
Konečný (stavový) automat (KA) je stroj, jehož vstupem je řetězec a výstupem logická hodnota. Obecně je to výpočetní model užívaný nejčastěji ke zpracování regulárních jazyků [33]. Formálně lze konečný automat M zapsat jako pětici $M = (Q, \Sigma, \delta, q_0, F)$, kde

- Q je konečná množina *vnitřních stavů*,
- Σ je vstupní abeceda,
- δ je relace $\delta \subseteq ((Q \times \Sigma) \times Q)$ a nazývá se *přechodová relace*,
- $q_0 \in Q$ je *počáteční stav*,
- $F \subset Q$ je množina *koncových stavů*.

Definice 2.20 (Konečný automat deterministický)
Deterministický konečný automat (DKA) je konečný automat, jehož přechodová relace δ je jednoznačná – δ je zobrazení[1] $(Q \times W) \rightarrow Q$ a nazývá se *přechodová funkce*.

Definice 2.21 (Konečný automat nedeterministický)
Řekneme, že konečný automat je nedeterministický (NKA), pokud není deterministický.

Poznámka 2.22 (Konečný automat nedeterministický)
Někdy je zvykem na přechodovou relaci NKA nahlížet jako na zobrazení $(Q \times W) \rightarrow \mathcal{P}(Q)$, kde $\mathcal{P}(Q)$ je potenční množina množiny všech stavů (např. [33, s. 31]).

Poznámka 2.23 (Výpočet konečného automatu)
Výpočet konečného automatu nad řetězcem $w \in W^*$ délky n je posloupnost přechodů mezi stavy $\{q_0, q_1 \dots q_n\}$ z množiny Q taková, že $\forall i \geq 0, i < n$ platí $q_{i+1} \in \delta(q_i, w_i)$, kde w_i značí i-tý symbol slova w (indexace od 0). „Pamětí“ automatu je tedy pouze informace o stavu, ve kterém se právě nachází.

Definice 2.24 (Konfigurace konečného automatu)
Konfigurace konečného automatu je dvojice (q, w), kde q je aktuální stav automatu a w zbylá část vstupního řetězce, kterou automat dosud nepřečetl. Relace mezi předchozí a následující konfigurací při výpočtu KA se nazývá *relace přechodu* (zkrác. *přechod*).

Poznámka 2.25 (Koncová konfigurace konečného automatu)
Koncová konfigurace KA je konfigurace (q, ε), kde q je stav z množiny koncových stavů.

Definice 2.26 (Jazyk přijímaný konečným automatem)
Řekneme, že slovo w je přijato konečným automatem M, jestliže existuje posloupnost přechodů z počáteční konfigurace (q_0, w) do koncové konfigurace. V opačném případě řekneme, že slovo w je automatem M odmítnuto. Množina všech slov přijímaných automatem M je *jazyk přijímaný konečným automatem* M a značí se $L(M)$.

[1] *Zobrazením* množiny A do množiny B nazýváme každou binární relaci $f \subseteq A \times B$, pro kterou ke každému $a \in A$ existuje nejvýše jedno $b \in B$ tak, že $(a, b) \in f$. [27, s. 10]

Definice 2.27 (Konečný automat – vlastnosti)
DKA je *úplný*, pokud jeho přechodová funkce je totálním zobrazením [27].
Stav q je *dosažitelný*, pokud existuje řetězec, při jehož zpracování dojde k průchodu přes q.
Stav, který není dosažitelný, je *nedosažitelný*.
Stav q je *užitečný*, pokud existuje řetězec w, který je automatem přijat z konfigurace (q, w).
Stav, který není užitečný, je *zbytečný*.
Konečné automaty M a N nazveme *ekvivalentní*, pokud $L(M) = L(N)$.

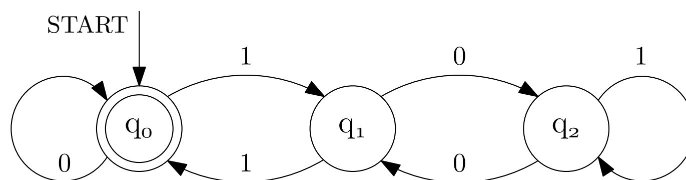
Věta 2.1 (Ekvivalence DKA a NKA)
Pro každý nedeterministický automat M existuje deterministický automat N takový, že $L(M) = L(N)$. [33]. Algoritmy převodu NKA na DKA jsou známy [33, s. 42][24, s. 8].

Poznámka 2.28 (Zobecněný NKA)
NKA může být zobecněn doplněním o přechody, při kterých se nečte žádný vstupní symbol. Takové přechody nazveme *ε-přechody*. Dále je možné ho zobecnit tím, že místo jednoho počátečního stavu bude definována množina počátečních stavů. Takový automat nazveme *automat s množinou počátečních stavů* a jeho přechodový diagram nemusí být souvislý (počátky vedou do jiných segmentů grafu).

Takto zobecněné NKA mohou být v některých situacích názornější, avšak jejich výpočetní síla je totožná s NKA bez ε-přechodů a jedním počátečním stavem. Existují tudíž techniky, jak ε-přechody odstraňovat (počítá se tranzitivní uzávěr dostupnosti přechodů bez čtení vstupu) nebo redukovat množinu počátečních stavů (zavedeme nový počáteční stav a pomocí ε-přechodů provedeme paralelizaci počátku).

Poznámka 2.29 (Znázornění konečného automatu)
Konečný automat můžeme znázornit tabulkou přechodů nebo orientovaným grafem – *přechodovým diagramem*. Přechodový diagram je vlastně grafem přechodové relace automatu, přičemž prvky z množiny vstupní abecedy tvoří ohodnocení hran grafu (ε pro ε-přechody). Koncové stavy jsou znázorněny dvojitým obrysem kolečka stavu a počáteční stav(y) šipkou „START“ (viz příklad 2.1).

Příklad 2.1 (Znázornění konečného automatu)
KA přijímající regulární výraz [33] $(0 + 11 + 101^*(00)^*01)^*$, což je jazyk všech přirozených čísel dělitelných 3 v binárním zápisu (index stavu je velikost zbytku dosud přečtené části čísla):



Obrázek 2.1: Příklad znázornění přechodového diagramu KA

Definice 2.30 (Překladový konečný automat)
Překladový konečný automat (PKA, KPA) je stroj, jehož vstupem i výstupem je řetězec. Formálně lze překladový konečný automat M zapsat jako šestici $M = (Q, T, D, \delta, q_0, F)$, kde

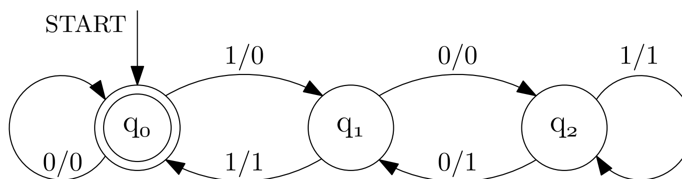
- Q je konečná množina *vnitřních stavů*,
- T je vstupní abeceda,
- D je výstupní abeceda,
- δ je relace $\delta \subseteq ((Q \times \{T \bigcup \varepsilon\}) \times \{Q \times D^*\})$[2],
- $q_0 \in Q$ je *počáteční stav*,
- $F \subset Q$ je množina *koncových stavů*.

Poznámka 2.31 (Rozdíl mezi KA a PKA)
Hlavní rozdíl mezi KA a PKA je v typu výstupu. Zatímco KA odpovídal logickou hodnotou – slovo je/není z jazyka („rozhodoval jazyk“), PKA transformuje slovo z vstupního jazyka na nějaké slovo z výstupního jazyka („překládá jazyk“). Transformace probíhá akumulací výstupních symbolů. Možnost ε na vstupu znamená, že automat při přechodu nečte symbol (pouze zapisuje). Naopak díky relaci s D^* může automat při jednom přechodu zapsat řetězec (více symbolů) nebo výstup vynechat (zapsat ε – řetězec délky 0).

Poznámka 2.32 (Překladový konečný automat – vlastnosti)
Další pojmy a speciální vlastnosti PKA definujeme analogicky jako u KA. Konfigurace PKA je trojice $(q, x, y) \in \{Q \times T^* \times D^*\}$. ε v definici relace δ poněkud komplikuje požadavek na deterministický PKA – δ musí být jednoznačná (viz def. 2.20) a navíc v každém stavu musí být možné přejít do dalšího stavu výlučně buď se čtením vstupu (jednoznačně) nebo bez něj.

PKA znázorňujeme obdobně jako KA (viz pozn. 2.29). Hrany v přechodovém diagramu ohodnocujeme kromě vstupu i výstupem automatu v „lomítkové“ syntaxi (viz příklad 2.2).

Příklad 2.2 (Znázornění překladového konečného automatu)
KA z příkladu 2.1 povýšený na PKA – automat přijímá binární čísla dělitelná 3 a jeho výstupem je podíl po dělení 3:



Obrázek 2.2: Příklad znázornění přechodového diagramu PKA

Mnoho dalších a souvisejících definic z teorie jazyků, gramatik a překladů lze nalézt v [33, 36, 29, 30].

[2] Stroj s takto koncipovaným výstupem bývá v terminologii KA nazýván *Mealyho automat*. Oproti tomu *Mooreův automat* generuje výstup pouze v závislosti na aktuálním stavu, bez ohledu na právě čtený symbol.

2.1.3 Komprese dat

Definice 2.33 (Komprese dat)
Komprese dat (také komprimace dat) je speciální postup při ukládání nebo transportu dat. Úkolem komprese dat je zmenšit datový tok nebo zmenšit potřebu zdrojů při ukládání informací. Obecně se jedná o snahu zmenšit velikost datových souborů, což je výhodné např. pro jejich archivaci nebo při přenosu přes síť s omezenou rychlostí (snížení doby nutné pro přenos). [15]

Poznámka 2.34 (Komprese dat)
Z hlediska teorie překladačů [36] je komprese dat pouze speciálním případem překladu textu (resp. binárních dat) na text (resp. binární data).

Poznámka 2.35 (Dekomprese dat)
Postup inverzní ke kompresi (rekonstruující původní data) se nazývá *dekomprese*.

Definice 2.36 (Ztrátová komprese)
Ztrátová komprese je taková komprese, při níž jsou některé informace nenávratně ztraceny a nelze je zpět rekonstruovat. Používá se tam, kde je možné ztrátu některých informací tolerovat a kde nevýhoda určitého zkreslení je bohatě vyvážena velmi významným zmenšením souboru. Používá se zejména pro kompresi zvuku a obrazu (videa), při jejichž vnímání si člověk chybějících údajů nevšimne nebo si je dokáže domyslet (do určité míry). [15]

Definice 2.37 (Bezeztrátová komprese)
Bezeztrátová komprese je taková komprese, při níž není žádná původní informace ztracena či znehodnocena. Obvykle není tak účinná jako ztrátová komprese dat. Velkou výhodou je, že zakomprimovaný soubor lze opačným postupem rekonstruovat přesně do původní podoby. To je nutná podmínka při přenášení obecných počítačových dat, výsledků měření, textu apod., kde by ztráta i jediného znaku mohla znamenat nenávratné poškození souboru. [15]

Definice 2.38 (Symetrická komprese)
Symetrická komprese je druh komprese, u které je doba trvání, resp. výpočetní složitost, procesu komprese srovnatelná s procesem dekomprese.

Definice 2.39 (Asymetrická komprese)
Řekneme, že komprese je *asymetrická*, pokud není symetrická.

Poznámka 2.40 (Asymetrická komprese)
Asymetrická komprese může vykazovat buď delší dobu trvání procesu komprese nebo procesu dekomprese. První případ je výhodné nasazovat v situacích, kdy je dostatek času na jednou prováděnou kompresi a málo na mnohokrát prováděnou dekompresi (typicky velkoobjemové archivační účely multimédií). Druhý případ je výhodné nasazovat v situacích s častou kompresí a výjimečnou dekompresí (typicky pravidelné zálohování).

Poznámka 2.41 (Kompresní model)
Proces komprese i dekomprese dat je řízen metadaty, která reprezentují informaci o překladu vstupu na výstup (např. tabulka častých frází). Tato metadata nazveme *kompresní model*.

Definice 2.42 (Statická komprese)
Statická komprese (komprese se statickým schématem) je druh komprese, jehož kompresní model je statický – konstantní v čase a nezávislý na komprimovaných datech.

Definice 2.43 (Semi-adaptivní komprese)
Semi-adaptivní komprese (komprese se semi-adaptivním schématem) je druh komprese, u kterého se kompresní model buduje při prvním průchodu dat ke kompresi. Při druhém průchodu se na data aplikuje a jeho metadata jsou kódována na začátku výstupu.

Definice 2.44 (Adaptivní komprese)
Adaptivní komprese (komprese s adaptivním schématem) je druh komprese, u kterého se kompresní model buduje dynamicky podle příchozích dat ke kompresi. Komprese probíhá v rámci jediného průchodu.

Poznámka 2.45 (Lokálně adaptivní komprese)
Kombinací schémat z def. 2.42 s 2.44 nebo 2.43 s 2.44 obdržíme *lokálně adaptivní kompresi*, jejíž model se mění v rámci malých úseků.

Definice 2.46 (Kompresní poměr)
Kompresní poměr je podíl

$$CR = \frac{LCD}{LOD}$$

kde LCD je velikost zakomprimovaných dat
a LOD velikost původních dat.

Definice 2.47 (Kompresní faktor)
Kompresní faktor je podíl

$$CF = \frac{1}{CR} = \frac{LOD}{LCD}$$

kde CR, LCD, LOD viz def. 2.46.

Poznámka 2.48 (Negativní komprese)
Z podstaty komprese jsou žádoucí situace, kdy $CR < 1$ ($CF > 1$). V situacích, kdy $CR > 1$ ($CF < 1$), hovoříme o *negativní kompresi*.

Poznámka 2.49 (Text)
Pro účely této práce se pojmem *text* rozumí data (obecná, nikoliv pouze data textového charakteru) na vstupu kompresního algoritmu (zdrojová zpráva). Značíme ho obvykle T.

Poznámka 2.50 (Vzorek)
Pro řetězec, jehož výskyt v textu chceme zodpovědět, používáme pojem *vzorek* a značíme ho obvykle P (z angl. “Pattern”).

Poznámka 2.51 (Vyhledávání v komprimovaném textu)
Vyhledávání v komprimovaném textu je hledání vzorku v textu, který je zakomprimován určitou kompresní metodou. Pro účely této prácem nebudeme uvažovat triviální řešení, kdy se text napřed dekomprimuje a následné se vyhledává v obyčejném textu.

Kapitola 3

Kompresní metoda DCA

Kompresní metoda DCA je kontextová kompresní metoda nad binární abecedou $\Sigma = \{`0', `1'\}$, a jak jsme zmínili již v úvodu, je založena na myšlence tzv. „antislovníku“. *Antislovník* (AD) je množina řetězců, které se v komprimovaném textu nevyskytují. Tyto řetězce nazveme *zakázaná slova* – „antislova“ (AW)[1]. AD tvoří *anti-faktoriální jazyk* (množinu antifaktorů textu). Formálně zavedeme pojem antislovníku definicí 3.1

Definice 3.1 (Antislovník)
Pro zadaný text T v abecedně Σ je antislovník $AD(T)$ množina řetězců v abecedě Σ , která splňuje podmínku:

$$\forall(w \in AD(T)) : w \notin Fact(T)$$

a z této definice ihned zřejmě vyplývá, že $AD(T) \subset AFact(T)$, kde $AFact(T)$ je množina *všech* antifaktorů T a dále platí:

$$\forall(r, s, t \in B^*) : rst \in Fact(T) \Rightarrow s \notin AD(T) \tag{3.1}$$

$$\forall(r, s, t \in B^*) : s \notin Fact(T) \Rightarrow rst \in AFact(T) \tag{3.2}$$

Kontrola v rovnici 3.2 nám umožňuje dále neuvažovat řetězce rst (pro $r \neq \varepsilon$ nebo $t \neq \varepsilon$) jako antislova, protože jejich informaci máme plně obsaženou v kratších antislovech s. Tato úvaha vede k definici toho, co je to *minimální antislovo*:

Definice 3.2 (Minimální antislovo)
Antislovo w řetězce T je *minimální*, pokud:

$$\forall(s \in B^*; b \in B) : s \notin Fact(T) \Rightarrow w \neq bs \wedge w \neq sb)$$

nebo intuitivněji užitím vlastních faktorů a množiny všech antifaktorů:

$$PFact(w) \cap AFact(T) = \emptyset \,, \text{resp. } PFact(w) \subset Fact(T) \tag{3.3}$$

[1] Pojem *antislovo* používáme v kontextu s DCA kompresí. Naproti tomu pojem *antifaktor* považujeme za obecný pojem stringologie. Za běžných podmínek je každé antislovo komprimovaného textu jeho antifaktorem.

Definice 3.2 v praxi znamená, že odebráním symbolu (1 bitu) ze začátku nebo konce minimálního antislova, nemůže vzniknout antifaktor – musí vzniknout faktor, čili podřetězec textu, který už antislovem z definice být nemůže. Minimální antislova jsou tedy nejkratší možná a jejich délka je „těsně za hranicí“ mezi faktory a antifaktory. Taková situace je výhodná z hlediska úspory paměti pro antislovník, ale i času algoritmů, která s antislovy pracují.

Poznámka 3.3 (Minimální antislovník)
Zavedeme konvenci, že **antislovník implicitně obsahuje právě minimální antislova** (pokud není uvedeno jinak). Toto si můžeme dovolit, protože minimalita antislov informační hodnotu antislovníku nijak nezhoršuje, avšak některé jeho vlastnosti může naopak vylepšit. Při vyhledávání v textu pomocí jeho antislovníku ji budeme někdy dokonce požadovat, jak uvidíme v analytické podkapitole 4.3.

Jako důsledek předchozího pro jeden libovolný antislovník AD (s minimálními AW) textu T platí, že žádné antislovo nemůže být faktorem jiného antislova, resp. formálně:

Lemma 3.1 (Vzájemná exkluze minimálních antislov)
Pro libovolná dvě minimální antislova w_1, w_2 v antislovníku $AD(T)$ platí:

$$\forall(w_1, w_2 \in AD(T)) : w_1 \notin Fact(w_2)$$

Důkaz 3.1 (Lemma 3.1)
Provedeme důkaz sporem. Předpokládejme negaci dokazované věty:

$$\exists(w_1, (w_2 = asb^2) \in AD(T)); a, b \in \Sigma; s \in \Sigma^* : w_1 \in Fact(w_2) \tag{3.4}$$

Potom buď $w_1 \in Fact(as)$ nebo $w_1 \in Fact(sb)$, protože AD je množina (neobsahuje stejné prvky). Dále pak jako důsledek $w_1 \in AD$ musí $as \notin Fact(T)$ nebo $sb \notin Fact(T)$ (viz lemma 4.1 pro $P = as$ nebo $P = sb$). Ovšem toto zjištění je v rozporu s definicí 3.2, neboť buď $as \in PFact(w_2)$ (chybí b na konci) a $as \notin Fact(T)$ nebo $sb \in PFact(w_2)$ (chybí a na začátku) a $sb \notin Fact(T)$, ačkoliv definice minimálního AW požaduje $PFact(w_2) \subset Fact(T)$. Obdržený spor zneplatňuje předpoklad v rovnici 3.4 a tudíž platí dokazovaná věta, čímž je důkaz ukončen.

3.1 DCA kódování

Vlastní kompresi textu T provádí DCA kodér na základě znalosti jeho antislovníku AD tak, že čtené symboly T (jednotlivé bity proudu) replikuje na výstup, avšak v situacích, kdy $\exists(w \in AD; b \in B) : w = w'b$ a zároveň $w' \in Suff(t_i)$, kde $t_i \in Pref(T)$ je dosud přečtená část textu, vstupní bit b nereplikuje. Právě tyto bity jsou ušetřeny a snižují kompresní poměr. Vynechávání těchto bitů si můžeme dovolit s ohledem na fakt, že antislovo w (které je samo

[2] Zde si můžeme dovolit předpokládat existenci alespoň jednoho symbolu z obou stran řetězce, protože pro $|w_2| < 2$ by nutně muselo $|w_1| < 1$ (AD je množina, takže neobsahuje stejné prvky a tudíž w_1 musí být vlastní faktor w_2, čili kratší), ale $w_1 \neq \varepsilon$, protože takové antislovo by nebylo antifaktorem žádného textu.

jednoznačně určeno díky minimalitě AD) jednoznačně určuje bit b a ten tudíž nemůže ve vstupním textu následovat, protože tím by byla porušena definice AD. Musí tedy následovat komplement b, ale ten je z podstaty binární abecedy určen rovněž jednoznačně – je to $\neg b$ ('1' pro $b =$ '0' a '0' pro $b =$ '1'). Řekneme, že takový bit b je *predikovatelný* (pomocí AD). Tento postup v pseudokódu formalizuje algoritmus 3.1.

Algoritmus 3.1 DCA kodér

vstup: text T, jeho antislovník AD
výstup: zakomprimovaný text T', počet vynechaných bitů *omitted*
procedura:

```
1:  T' ← ε {empty string}
2:  omitted ← 0
3:  for i from 0 to |T| − 1 do {indexation from 0}
4:    for all w ∈ AD do
5:      if butlast[3](w) ∈ Suff(T[0..i][4]) then
6:        omitted ← omitted + 1
7:        continue for 3
8:      end if
9:    end for
10:   T' ← T'.T[i] {replicating unpredictable bit}
11: end for
12: return (T', omitted)
```

Algoritmus 3.1 provádí kromě vlastní komprese ještě počítání vynechaných bitů. Toto je informace nutná k rekonstrukci správné délky původního textu při dekompresi. Komprese totiž může zobrazovat dva různé texty, kratší T_1 a delší T_2, na totožné výstupy. Stane se to tehdy, pokud všechny bity, o které je T_2 delší, jsou pomocí AD predikovatelné. Při dekompresi již není možné rozlišit, zda-li byl komprimován text T_1 nebo T_2. Informace o počtu vynechaných bitů nám však tuto „koncovou singularitu" umožní rozlišit. Alternativně lze uchovávat delší hodnotu délky původního textu [23, s. 11]. V konkrétní implementaci tuto informaci vhodně kódujeme v rámci kompresních metadat.

Ukážeme krátký příklad simulace běhu komprese pro $AD = \{\text{'00'}, \text{'111'}, \text{'01010'}\}$ a $T =$ '11011010110'. Předně můžeme vidět, že AD je skutečně antislovník pro nějaký text, protože jeho antislova nejsou vzájemnými faktory (jinak by nebyl minimální). Dále ověříme, že AD je antislovníkem zrovna pro T. K tomu musí platit:

- (1) '00'$\notin Fact(T)$ a zároveň '0'$\in Fact(T)$,
- (2) '111'$\notin Fact(T)$ a zároveň '11'$\in Fact(T)$,
- (3) '01010'$\notin Fact(T)$ a zároveň '0101', '1010'$\in Fact(T)$.

Pohledem na T vidíme, že podmínky pro všechna tři antislova jsou splněny a můžeme prohlásit, že AD je skutečně možným antislovníkem pro text T. Můžeme tedy přistoupit k vlastní simulaci, kterou zachycuje příklad 3.1.

[3] Funkce *butlast*(s) vrací řetězec s bez posledního symbolu.
[4] Indexace řetězce s v notaci $s[a..b]$ má zde význam rozsahu jeho symbolů v intervalu $\langle a, b)$

Příklad 3.1 (DCA kodér – simulace)
Simulace běhu komprese pro $AD = \{`00', `111', `01010'\}$ a $T = `11011010110'$ je v tabulce 3.1. První řádek je inicializace a každý další stav po jednom průchodu hlavní smyčkou.

i	vstup $T[0..i]$	výstup T'	použité AW	vynecháno bitů
0	ε	ε		0
1	1	1		
2	11	11		
3	110	11	111	1
4	1101	11	00	2
5	11011	111		
6	110110	111	111	3
7	1101101	111	00	4
8	11011010	1110		
9	110110101	1110	00	5
10	1101101011	1110	01010	6
11	11011010110	1110	111	7

Tabulka 3.1: Simulace DCA kodéru

Na příkladu 3.1 můžeme vidět, že text s délkou 11 bitů se podařilo zakomprimovat vynecháním 7 b na text s délkou 4 b a každé slovo z AD bylo alespoň jednou použito k ušetření bitu. Kompresní poměr však nebude $\frac{4}{11}$, protože k výstupním datům bychom museli přidat ještě vhodně kódovaný antislovník a informaci o délce původního textu. Prezentaci jiného příkladu v „pohyblivé“ verzi nalezneme v přednáškách [11, lec.8].

3.2 DCA dekódování

Od DCA dekódování se očekává zpětná rekonstrukce komprimovaného textu. Dekodér by se měl chovat obráceně vzhledem ke kodéru – v každém kroku připisovat bit do výstupu, ale v některých situacích (ve stejných místech, kde kodér vynechával zápis) vynechávat čtení. Hodnota výstupního bitu v takových situacích závisí pouze na historii vstupu (proto DCA patří mezi kontextové kompresní metody). Zápis dekodéru v pseudokódu je algoritmus 3.2 a simulaci dekomprese pro komprimovaný text z příkladu 3.1 najdeme v příkladu 3.2.

Příklad 3.2 (DCA dekodér – simulace)
Simulace běhu dekomprese pro $AD = \{`00', `111', `01010'\}$, $T' = `1110'$ a $omitted = 7$ je v tabulce 3.2. První řádek je inicializace a každý další stav po jednom průchodu hlavní smyčkou.

Jak je vidět z příkladů 3.1 a 3.2, po dekompresi produktu komprese jsme obdrželi původní text. Dekompresní algoritmus 3.2 musí v případě validního vstupu skončit s hodnotou $omitted = 0$. Tuto vlastnost bychom mohli použít k testování integrity zakomprimovaných dat. Proto ji zde, jakožto větu 3.1, dokazujeme důkazem 3.2.

[5] Funkce *last(s)* vrací poslední symbol řetězce *s*.

Algoritmus 3.2 DCA dekodér

vstup: zakomprimovaný text T', antislovník AD textu T, počet vynechaných bitů *omitted*
výstup: původní text T
procedura:

```
1: T ← ε {empty string}
2: i ← 0 {indexation in T'}
3: loop
4:   for all w ∈ AD do
5:     if butlast(w) ∈ Suff(T) then
6:       break loop if ommited = 0 {termination on predicting}
7:       T ← T.¬last^5(w) {predicting complementary bit}
8:       omitted ← omitted − 1
9:     else
10:      break loop if i = |T'| {termination by default}
11:      T ← T.T'[i]
12:      i ← i + 1
13:    end if
14:  end for
15: end loop
16: return T
```

i	vstup $T'[0..i]$	výstup T	použité AW	vynecháno bitů
0	ε	ε		7
1	1	1		
2	11	11		
2	11	110	111	6
2	11	1101	00	5
3	111	11011		
3	111	110110	111	4
3	111	1101101	00	3
4	1110	11011010		
4	1110	110110101	00	2
4	1110	1101101011	01010	1
4	1110	11011010110	111	0

Tabulka 3.2: Simulace DCA dekodéru

Věta 3.1
Algoritmus 3.2 skončí pro platný DCA komprimát (výstup z alg. 3.1) a antislovník AD původního textu T na vstupu s hodnotou $omitted = 0$.

Důkaz 3.2 (Věta 3.1)
Algoritmus 3.2 pracuje v nekonečné smyčce (řádky 3–15), kterou opouští z řádků 6 (1) nebo 10 (2). Příkaz (1) má dokazovaný předpoklad v podmínkové části, takže zbývá předpoklad dokázat pro opuštění smyčky příkazem (2). Podmínkou příkazu (2) je rovnost $i = |T'|$, kde

$|T'|$ je délka vstupu T' (zakomprimovaného řetězce). Tato rovnost ale platí pouze v případě, že bylo na výstup T zapsáno alespoň tolik bitů, kolik je délka vstupu, protože zápis bitu do T (řádek 11) je ve stejné větvi s jedinou inkremetancí i (řádek 12). Další zápisy (nad počet $|T'|$) pak nutně provedl příkaz na řádku 7 (3), protože kromě těchto dvou příkazů se do T nezapisuje a oba zapisují právě jeden bit.

Rozdíl mezi celkovým počtem zapsaných bitů a $|T'|$ je tedy nutně roven rozdílu mezi hodnotou vstupního argumentu *omitted* a aktuální hodnotou *omitted* v algoritmu (po dokončení větve), protože jediná dekrementace této proměnné (řádek 8) je ve stejné větvi se zápisem (3). Zápis (3) je však pod podmínkou na řádku 5, která znamená, že bude následovat zápis predikovatelného bitu. Tím jsme dokázali, že *aktuální* hodnota *omitted* vždy odpovídá počtu predikovatelných bitů, které ještě zbývá zapsat. Pokud algoritmus skončí s hodnotou $omitted \neq 0$, značí to buď neúplný vstup T', nekompatibilní AD nebo moc vysokou hodnotu vstupního argumentu *omitted*, což je obměněná implikace dokazované věty.

Poznámka 3.4
Důkazem 3.2 jsme dokázali i správnost ukončení algoritmu 3.2, protože kdybychom jeho správnost předpokládali, nemuseli bychom dokazovat správnost hodnoty *omitted* v jeho průběhu a důkaz by se redukoval na poslední dvě věty.

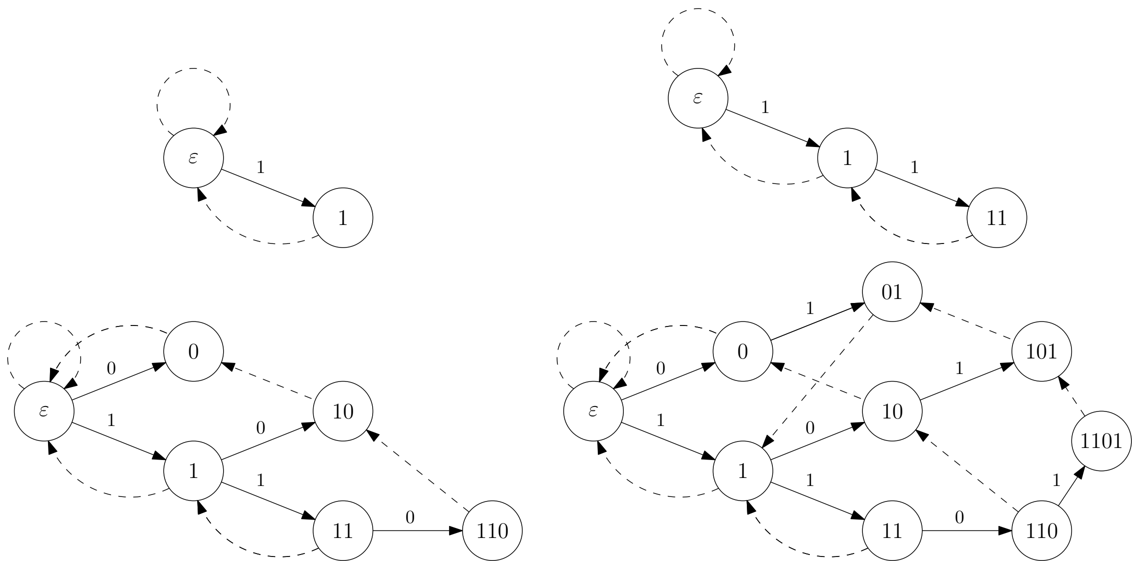
3.3 Konstrukce antislovníku

Dosud jsme předpokládali implicitní znalost AD. Prvním krokem komprese je však jeho konstrukce pro komprimovaný text. Z pohledu použití antislovníku AD při kompresi se vlastně v textu T řeší podmínka $butlast(w) \in Suff(T[0..i])$ **for all** $w \in AD$ v alg. 3.1. $Suff(T[0..i])$ jsou zde všechny přípony dosud zpracovaného textu na vstupu; přes celý běh kódu pak jeho všechny přípony všech předpon $Suff(Pref(T))$. Antislova (obecně neminimální) mohou být pouze takové řetězce, které se v této množině nevyskytují, což odpovídá jejich antifaktoriální definici ($AFact(T) = \overline{Fact(T)} = \overline{Suff(Pref(T))}$). Je tedy možné antislova hledat pomocí indexace všech přípon všech předpon komprimovaného textu, čili pomocí indexace všech jeho faktorů.

3.3.1 Suffix trie

K indexaci přípon předpon (faktorů) lze použít strukturu *suffix trie* (*trie of suffixes*, *position trie*) [18] („trie“ od slova retrieval), což je stromová struktura, ve které každý vrchol reprezentuje jednu příponu předpony, list příponu nevlastní předpony (celého textu), regulérní hrany znamenají čtení jednoho symbolu vstupního textu (přechod k delší předponě) a speciální zpětné hrany (*suffix linky*) udržují relaci mezi příponou a příponou o jeden symbol zleva kratší (tato informace se využije při vkládání nových přípon a posléze při hledání antislov, která jsou minimální). Průběh prvních čtyř kroků konstrukce [23] suffix trie struktury pro T = ‘11011010110’ (tzn. úplnou indexaci faktorů řetězce ‘1101’) můžeme sledovat na obrázcích v tab. 3.3 (suffix linky jsou vyznačeny přerušovanou čárou).

Po dokončení indexace faktorů je cesta k nalezení antislov přímočará. Hledáme vlastně řetězce, které se v indexu faktorů nevyskytují, a to jsou právě takové, jejichž poslední uzel



Tabulka 3.3: Začátek konstrukce suffix trie pro T = '11011010110'

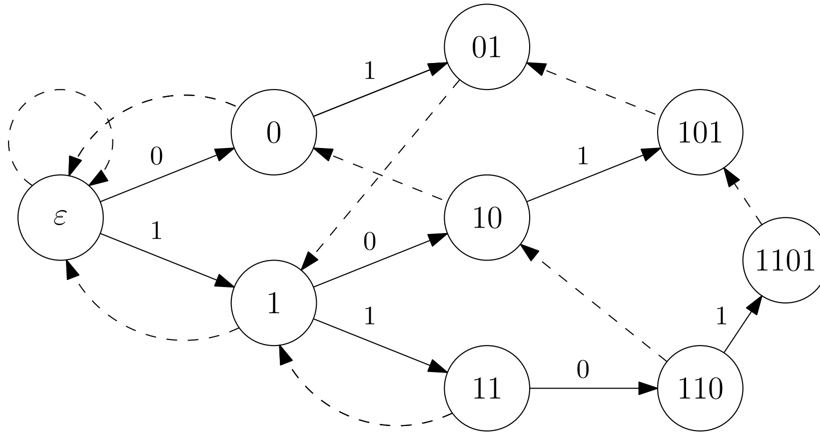
(po cestě od ε) není v grafu dopředných hran uvnitř suffix trie definován. Stačí se tedy omezit na ty vrcholy, ze kterých nevychází obě dopředné hrany (pro pokračování podřetězce oběma možnými bity) a antislova jsou pak zřetězením jejich ohodnocení s komplementem definované dopředné hrany (event. s oběma bity pro žádnou dopřednou hranu).

V posledním kroku (obrázek 3.1) jsou to řetězce '0 0', '01 0', '01 1', '10 0', '11 1', '101 0', '101 1', '110 0', '1101 0', '1101 1' (poslední bity za mezerou jsou ony nedefinované komplementy). Povšimněme si, že poslední dvě jmenovaná antislova jsou delší, než řetězec samotný (omezili jsme se na první čtyři kroky, čili na řetězec '1101'), protože vznikly doplněním uzlu pro celý řetězec (nejhlubší list v suffix trie). Takové řetězce sice splňují definici antifaktorů, ale v praxi žádnou užitečnou informaci neposkytují, protože delší řetězec je vždy antifaktorem kratšího (a zde nejsou ani minimální, neboť např. '11010'= '1'.'1010' a '1010'$\notin$ *Fact*(1101)).

V praxi je však nutné hloubku suffix trie a tudíž i délky AW omezit (viz podkapitola 3.3.3), protože paměťová složitost této struktury je $\mathcal{O}(|T|^2)$ (jeden uzel grafu za každý faktor a ten lze zvolit indexem začátku a konce – $(0..T)\times(0..T)$) a délka textu je typicky velká. Při omezení hloubky nemůžeme k prodloužení o neexistující dopředné hrany nejhlubší listy použít vůbec, protože informaci o pokračování grafu za nimi tímto omezením ztrácíme a jejich dopředné hrany mohou chybět pouze kvůli tomu. Z těchto listů jsou pak při výpočtech použity jen suffix linky k usnadnění traverzace v grafu.

3.3.2 Selekce minimálních AW

Antislova, která nalezneme jakožto antifaktory v suffix trie, však obecně nejsou *minimální* a pro správný antislovník podle konvence v pozn. 3.3 je třeba z nich ještě vybrat podmnožinu, jejíž prvky už minimální antislova budou. Toho dosáhneme použitím suffix linků. Suffix linky odkazují na o jeden symbol zleva kratší přípony a pro každý vrchol N



Obrázek 3.1: Úplná suffix trie pro řetězec ‘1101’

tedy platí, že $String^{6}(SLink^{7}(N)) \in Suff(String(N)) \in Fact(String(N))$. Z toho vyplývá, že v případě existence suffix linku mezi dvěma vrcholy, které oba generují antislova se stejným symbolem na konci, pak podle lemma 3.1 nemůže delší z nich být minimální. Takto odebíráme neminimální antislova až do doby, než zbydou pouze minimální a ty utvoří antislovník textu. V našem příkladě tímto způsobem odebereme ‘11010’, ‘11011’, ‘1100’, ‘1010’, ‘1011’, ‘100’ a zbyde $AD = \{‘00’, ‘010’, ‘011’, ‘111’\}$. Vidíme, že prvky tohoto AD nejsou vzájemné faktory a zároveň platí, že odebráním bitu z každé strany libovolného z nich vznikne faktor původního textu: $\{‘0’, ‘10’, ‘01’, ‘11’\} \subset Fact(‘1101’)$. Antislova jsou tedy skutečně minimální a AD je správně zkonstruovaný antislovník.

3.3.3 Omezení délky AW

Konstrukce AD pomocí suffix trie je, jak už bylo uvedeno výše, poměrně náročná na spotřebu paměti a pro praktické účely se indexace omezuje hodnotou k hloubky jejího grafu (nevkládají se vrcholy s řetězcem delším než k; vkládání na konce kratších přípon zůstává). Toto omezení způsobí to, že mohutnost antislovníku klesne (nevyskytují se v něm antislova delší než k), ale to nebrání jeho použití. Platí totiž věta 3.2, kterou nebudeme dokazovat, protože je zřejmá (na prvky AD jsou kladeny požadavky pouze jednotlivě).

Věta 3.2 (Neúplný antislovník)
Odebráním libovolného prvku z antislovníku AD_1 pro text T obdržíme antislovník AD_2 pro text T.

Omezení hloubky hledání faktorů – délky nalezených antifaktorů a jejich počtu, má kromě kladného vlivu na výpočetní nároky (zejména na paměť), samozřejmě záporný vliv na sílu komprese (na kompresní poměr). Problematikou konstrukce AD použitím sofistikovanějších datových struktur (*suffix arrays*) se zabývá [19]. Implementační hledisko nejen těchto modifikací, ale i řadu souvisejících aspektů podrobně rozebírá implementace DCA [23].

[6] $String(N)$ je řetězcové ohodnocení vrcholu N.
[7] $SLink(N)$ je přechod v grafu po zpětné hraně suffix linku.

3.4 Dodatky k DCA

3.4.1 DCA pomocí KA

V algoritmu 3.1 jsme na pseudokódu ukázali, jak probíhá DCA komprese. Efektivita a použitelnost konkrétní implementace bude záviset zejména na schopnosti vyhledávat přípony, které by mohly odpovídat některému z antislov po odebrání posledního bitu (řádek 5). Klíčovou výhodou metody DCA je skutečnost, že toto lze provádět paralelně pro všechny přípony (v praxi do hloubky k) v lineárním čase a navíc jednoduchým výpočetním modelem. Tímto modelem je *konečný automat*.

Konečný automat, který uspokojí potřebu DCA kodéru, bude vyhledávací automat pro více vzorků [31]. Může být zkonstruován např. algoritmem *Aho-Corasick* [15] pro „AC pattern matching“, což je v jistém smyslu zobecnění algoritmu *Knuth–Morris–Pratt* [15] pro více vzorků. Je možné automat sestavit také modifikací struktur pro konstrukci AD. Např. suffix trie je topologicky totožná s vyhledávací strukturou, kterou generuje AC algoritmus – suffix linky mají sémantický význam „fail funkce“, což je mapování na nejdelší příponu řetězce odpovídající předponě vzorku. Toto mapování (většinou realizované jako tabulka) se použije v případě, kdy aktuální symbol na vstupu neodpovídá dalšímu symbolu ve vzorku.

Záměrem dopředného hledání je totiž neporovnávat jednou přečtené symboly vícekrát, k čemuž je nutné vědět, od kterého místa ve vzorku je možné pokračovat (všechny symboly před tímto místem jsou příponou dosud přečtené části textu). Jedná se tedy o způsob předzpracování vzorku, ale v případě konstrukce KA je nutné nahradit fail funkci novými přechody. Toho dosáhneme přidáním hran pro neodpovídající symboly, jejichž cílem budou vrcholy reprezentující tyto nejdelší kratší přípony následované přechodem pro daný symbol, který se určí stejným způsobem). Bližší popis této techniky a výpočet fail funkce nalezneme v [34] (kapitola 5).

Vyhledávací KA poté povýšíme na překladový KA tak, že bude vstupní symboly replikovat na výstup pro všechny své přechody kromě těch, které vedou z koncových stavů (resp. ze stavů, které ohlašují výskyt některého z hledaných vzorků). Tím vznikne překladač pro DCA kompresi (angl. “DCA transducer”) – *DCA kompresní automat*. Jelikož operace komprese musí být (se správně zkonstruovaným AD) reverzibilní, musí existovat i automat pro zpětný překlad – *DCA dekompresní automat*. Z kompresního automatu ho vyrobíme záměnou vstupních a výstupních symbolů u všech přechodů. Toto je možné, jelikož kompresní automat zapisoval buď jeden symbol nebo žádný (ε), takže dekompresní automat bude po záměně vstupů a výstupů buď číst (po jednotlivých symbolech) nebo přecházet po ε-přechodech. Obecně tedy vznikne zobecněný nedeterministický automat. Bylo by možné jeho běh simulovat pomocí deterministických algoritmů [24], ale v tomto případě bude nejspíš lepší aplikovat známé algoritmy pro převod NKA na DKA [33] a deterministickou verzi následně použít k dekompresi.

3.4.2 Kompresní schémata DCA

3.4.2.1 Semi-adaptivně

Až dosud jsme se zabývali představou, že komprese proběhne ve dvou krocích – v prvním kroku se vytvoří antislovník textu a v druhém se použije k jeho kompresi. To odpovídá semi-

adaptivnímu schématu komprese (def. 2.43). Metadata kompresního modelu jsou v druhém kroku ve vhodném kódování (lze ho např. komprimovat sebou samým, tzv. „self-compression“ [23, s. 16], a to dokonce opakovaně) přidána na začátek komprimovaného výstupu. Jejich umístění na začátku je výhodné, protože umožní alespoň dekompresní jednotce pracovat proudově – nepoužívat náhodné čtení v souboru a přitom nečekat na přijetí všech dat.

3.4.2.2 Staticky

Pokud bychom mohli stejný antislovník používat opakovaně, a tudíž by nebylo nutné ho konstruovat znovu pro každý soubor vstupu, ani ho kódovat ke každému výstupu, odpovídalo by to kompresi se statickým schématem (def. 2.42). Antislovník by byl oběma stranám komprese předem známý a neměnný. Těžko bychom však hledali nová data s užitečným obsahem, která by odpovídala předem stanovenému antislovníku. Zde by bylo nutné použít mechanismus kódování výjimek, který by byl aplikován v případě, že by měl být komprimován bit, který ve skutečnosti predikovatelný není (díky neplatnosti daného antislova v nových datech).

Toto schéma negativně ovlivní dosahované kompresní poměry (v rámci vlastních dat – bez započítání objemu metadat), protože nebude možné komprimovat některé bity, jejichž antislova ve statickém AD nemáme, a naopak bude nutné přidávat bity kódující výjimky. Tato nevýhoda ale bude vyvážena malými hardwarovými nároky na použitou výpočetní platformu, protože komprese by mohla naplno využít všech výhod zdůrazněných v podkapitole 3.4.1 a zejména by nebylo nutné provádět nad každým vstupem konstrukci antislovníku, což je operace náročná na spotřebu operační paměti. Nutným předpokladem je však komprese podobných dat (dat jejichž antislovníky mají co nejmohutnější průnik).

3.4.2.3 Dynamicky (adaptivně)

Možný je však i pokročilejší přístup, využívající adaptivní kompresní schéma (def. 2.44). Při něm se antislovník konstruuje postupně v rámci jediného průchodu dat, společného s vlastní kompresí. I zde je ovšem nutno zpracovávat výjimky, protože v průběhu komprese stále pracujeme s nekompletním antislovníkem. Tato *dynamická* verze DCA je hlavním předmětem implementace [23].

3.4.3 Prořezávání AD

Větu 3.2 nemusíme využít jen při omezení hloubky AD (AW s nějakou maximální délkou), nýbrž pro odejmutí zcela libovolného antislova z AD. Ne všechna antislova (ne)nalezená v textu se totiž při jeho kompresi zaslouží o snížení kompresního poměru. Skutečností je, že některá antislova parametry komprese zhoršují, protože mají malou četnost použití (dají se použít třeba jen pouze jednou v rámci celého běhu komprese) a zabírají místo v AD. Rychlost zpracování a paměťové nároky každé další antislovo zhorší určitě a pokud by odměnou za to nebylo snížení kompresního poměru, nemá cenu takové antislovo uvažovat (není podmínkou, aby antislovník byl úplný). Strukturu nalezených antislov můžeme na základě libovolných kritérií prořezat a optimalizovat tak parametry komprese – hardwarovou náročnost i kompresní poměr.

3.4.4 Almost antiwords

Další možná technika optimalizace DCA komprese se nazývá *almost antiwords* („skoro antislova"). Tato technika vychází z opačné úvahy než prořezávání – zahrnout některá slova, která nejsou zakázaná (jsou to faktory vstupního textu), do antislovníku. Toto má cenu provádět při očekávání, že režie na kódování jejich výjimek (viz podkapitola 3.4.2.2) bude zabírat ve výstupu méně místa, než kolik místa nám ušetří použití těchto (skoro) antislov v rámci úseků, kde jejich sémantický význam platí (nejsou faktory velkých částí vstupního textu). Podrobněji je tato myšlenka rozebrána v [21].

Modelový příklad pro nasazení *almost antiwords* by vypadal asi takto:

$$\mathit{freq}(\mathit{butlast}(w).\neg \mathit{last}(w)) > \mathit{bitsInAD}(w) + \mathit{bitsInExc}(w), \tag{3.5}$$

kde w je sledované (skoro) antislovo, $\mathit{freq}(s)$ je četnost výskytů řetězce s v komprimovaném textu, $\mathit{butlast}()$ a $\mathit{last}()$ stejně jako u alg. 3.1 a 3.2, $\mathit{bitsInAD}(w)$ je počet bitů, které antislovo w zabírá v antislovníku ($|w|$ pro nekomprimovaný AD) a $\mathit{bitsInExc}(w)$ je počet bitů, které antislovo w zabírá v režii kódování výjimek.

Každý výskyt antislova nám šetří jeden bit na komprimovaném výstupu, takže platí-li nerovnost 3.5, (skoro) antislovo w nám kompresi zlepšuje (resp. zlepšuje kompresní poměr, avšak přidává na výpočetní režii) a můžeme ho při kompresi použít. Je vidět, že pro toto rozhodování je nutné počítat jisté ukazatele ($\mathit{bitsInAD}$, $\mathit{bitsInExc}$), což samo o sobě generuje nějakou další režii, a z nich pak počítat *zisk* pro (skoro) antislova, o jejichž použití se rozhodujeme. Podmnínka podobná nerovnosti 3.5 má význam i při prořezávání AD (podkapitola 3.4.3), avšak bez započtení členu pro režii výjimek (pracujeme stále s pravými antislovy).

DCA implementace [23] umožňuje tuto optimalizaci zapnout/vypnout makrem ve zdrojovém kódu.

Kapitola 4

Analýza a návrh řešení

V této kapitole se budeme zabývat možnostmi odpovědět na základní otázku vyhledávacího problému – jestli se hledaný vzorek P nachází v textu T, avšak v našem případě zakomprimovaném metodou DCA. Nebudeme však pracovat se zakomprimovanými daty (konvenční přístup), ale omezíme se na znalost *pouze kompresních metadat* (zde antislovníku). Jelikož je naše metoda věštící (je z principu, protože v AD je jen část původní informace), budeme se kromě definitivních odpovědí dostávat i do situací, ve kterých budeme schopni říci pouze to, že se vzorek v komprimovaném textu nacházet *může, ale nemusí.* Bylo by vhodné omezit se na takovou *neurčitou odpověď* pouze v případech, kdy lepší informaci na základě znalosti AD nelze z principu získat. Je nutné poznamenat, že kvalita informací, které lze z AD získat, bude záležet i na některých jeho vlastnostech (např. jeho minimalitě – minimalitě všech antislov v něm obsažených).

4.1 Negativní odpověď

Nejjednodušší situace, ve které můžeme *zavrhnout výskyt vzorku* v zakomprimovaném textu je taková, když v hledaném vzorku nalezneme alespoň jeden prvek AD – antislovo – jakožto jeho faktor. Protože takové podřetězce se v textu určitě nevyskytují (jsou to antislova), nevyskytuje se v textu ani hledaný vzorek. Je tomu tak díky tranzitivitě v lemma 4.1. Zjištění této skutečnosti provedeme aplikací *vyhledávacího automatu* na jednotlivá antislova.

Lemma 4.1 (Tranzitivita vlastnosti „nebýti podřetězcem“)

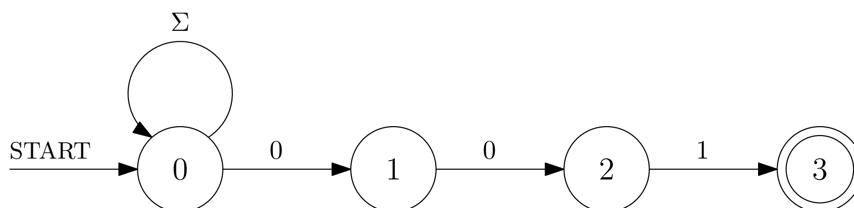
$$(w \notin Fact(T) \wedge w \in Fact(P)) \Rightarrow P \notin Fact(T)$$

Důkaz 4.1 (Lemma 4.1)
$w \notin Fact(T)$ dle def. 2.13 znamená, že $\forall u, v \in \Sigma^* : T \neq uwv$ (1) (kde Σ je abeceda pro T a w). $w \in Fact(P)$ znamená naopak, že $\exists u', v' \in \Sigma^* : P = u'wv'$ (2). Pokud by neměl platit důsledek implikace a mělo by $P \in Fact(T)$, pak by $\exists u'', v'' \in \Sigma^* : T = u''Pv''$ a kvůli (2) dále $T = u''u'wv'v''$. Jenže to by nemohlo současně platit (1), protože by $\exists u, v \in \Sigma^* : T = uwv$, a sice $u = u''u', v = v'v''$, což je v rozporu s předpokladem implikace. Důsledek implikace tedy platí a tím je důkaz ukončen.

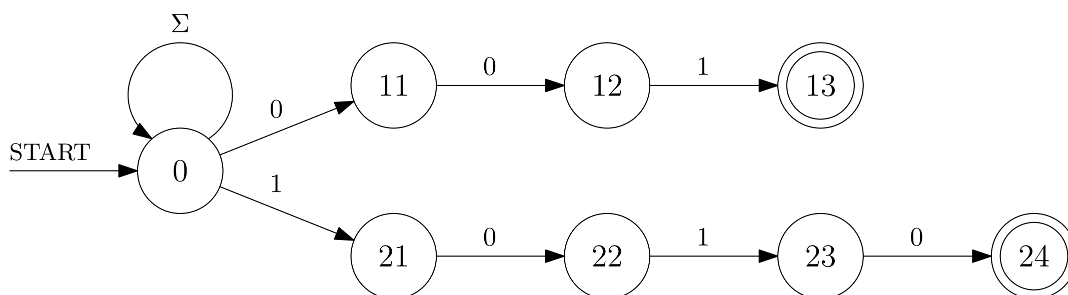
4.1.1 Vyhledávací automat

Základní schéma *vyhledávacího automatu* [31] (pro vzorek '001'), který se vyznačuje nedeterministickou smyčkou v počátečním stavu (Σ je celá abeceda, zde {'0', '1'}), je na obr. 4.1. V implementaci bude potřeba automat determinizovat či simulovat. AD má více položek, a proto je nutné základní automat upravit pro vyhledávání všech jeho prvků (antislova jsou zde vlastně vzorky vyhledávané automatem a hledaný vzorek P v zakomprimovaném textu je vstupem do automatu). Toho dosáhneme paralelizací více takových automatů, jak ukazuje obr. 4.2, který odpovídá $AD = \{$'001', '1010'$\}$.

Protože hledáme odpověď na otázku, jestli se ve vzorku antislovo nachází (a nepotřebujeme vědět kde a kolik těch výskytů je), může proces vyhledávání skončit po průchodu automatu libovolným koncovým stavem (resp. ihned po jeho dosažení). V takovém případě je některé z antislov faktorem hledaného vzorku a ten tudíž nemůže být faktorem komprimovaného textu – nevyskytuje se v něm a můžeme oznámit *negativní odpověď* („vzorek se v textu nevyskytuje“).



Obrázek 4.1: Základní vyhledávací automat

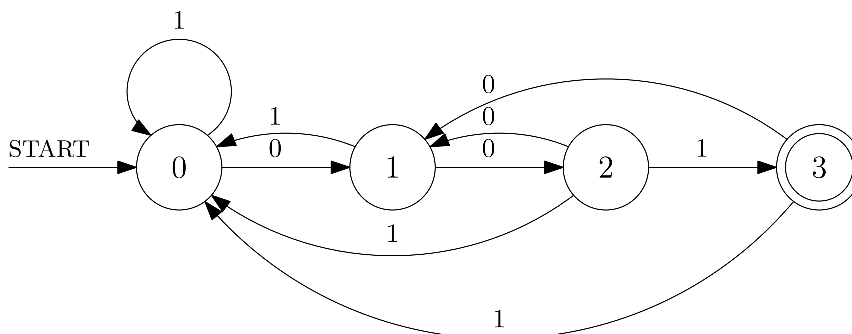


Obrázek 4.2: Vyhledávací automat pro více vzorků

V opačném případě – pokud automat již zpracoval celý vstup (hledaný vzorek), a přesto neprošel koncovým stavem, je nutné v tomto základním schématu konstatovat, že výskyt hledaného vzorku v komprimovaném textu nemůžeme vyloučit. V některých případech však lze naopak predikovat přítomnost hledaného vzorku. Jedná se o situaci, kdy kodérem poskytnutý AD je minimální a hledaný vzorek je naopak faktorem některého antislova. Tuto možnost se pokusíme rozebrat v podkapitole 4.3 a i její použitelnost eventuálně zjistit v kapitole o testování 5.2.

Takovýto vyhledávací automat je vlastně metodou, při které dochází k předzpracování vzorku/vzorků (toho, co se zde vyhledává – antislov). Předzpracování je tedy fixní vzhledem k AD a bylo by možné ho ve vhodném kódování ukládat ke komprimovaným datům, což by se mohlo ukázat jako výhodné v případě, kdybychom ve stejném komprimovaném textu chtěli často vyhledávat jiná data.

Složitost výpočtu vyhledávacího automatu je lineární vzhledem k délce vstupu (zde vzorku P) – $\mathcal{O}(|P|)$ (při čtení každého symbolu na vstupu provede automat jeden přechod), ovšem to platí v případě, že pracujeme už s jeho determinizovanou podobou. Jeho konstrukce do nedeterministické podoby je lineární vzhledem k délce antislovníku – $\mathcal{O}(\sum_{AD} |w \in AD|) = \mathcal{O}(|AD|)$ (nový stav za každý symbol všech antislov). Nepříznivá je složitost procesu determinizace, neboť počet stavů při převodu NKA→DKA roste obecně exponenciálně (skutečnost, že NKA může být naráz v množině stavů, ošetříme tím, že z celé této množiny vytvoříme jeden nový stav DKA), takže paměťová složitost konstrukce je $\mathcal{O}(2^{|P|} \times |\Sigma|)$ (vyplňujeme tabulku $Q \times \Sigma$) a výpočetní složitost je omezená nutností reprezentaci automatu v paměti naplnit – $\mathcal{O}(2^{|P|})$. Ve speciálních případech může být složitost neasymptoticky příznivější, např. pro *homogenní automaty* [35, s. 46]. Diagram automatu z obr. 4.1 po determinizaci ukazuje obr. 4.3.



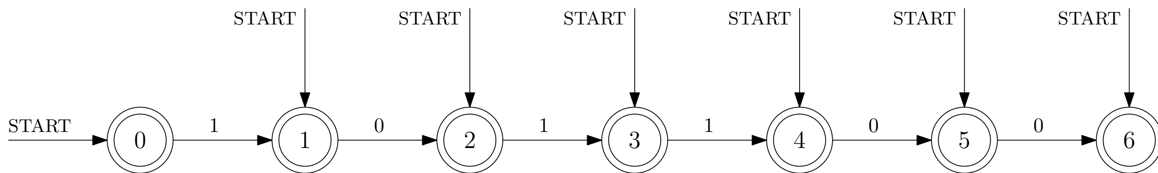
Obrázek 4.3: Determinizovaný automat z obr. 4.1

Zde nám absolutní velikost počtu stavů nevzrostla, protože automatu stačí vědět délku dosud nalezené předpony jediného vzorku, k čemuž potřebuje právě $|P| + 1$ stavů (při determinizaci jsme některé stavy z původního NKA mohli vyřadit jako nedosažitelné). Jinak by tomu bylo ve složitějším případě, např. u automatu pro více vzorků. Výpočet NKA lze, kromě determinizace a výpočtu ekvivalentního DKA, řešit také jeho simulací. V takovém případě se udržuje místo jednoho aktuálního stavu automatu informace o celé množině stavů, ve kterých automat může být (do kterých vede cesta z počátku pro dosud přečtený vstup), a toto se zohledňuje i při výpočtech přechodů. Výpočetní složitost takové simulace je $\mathcal{O}(N \times |Q|)$, kde N je délka vstupu (možnost přechodu musíme testovat pro celou množinu aktuálních stavů, jichž je nejvíce $|Q|$). Problematikou simulace NKA ve vyhledávacích úlohách se velmi podrobně zabývá [24].

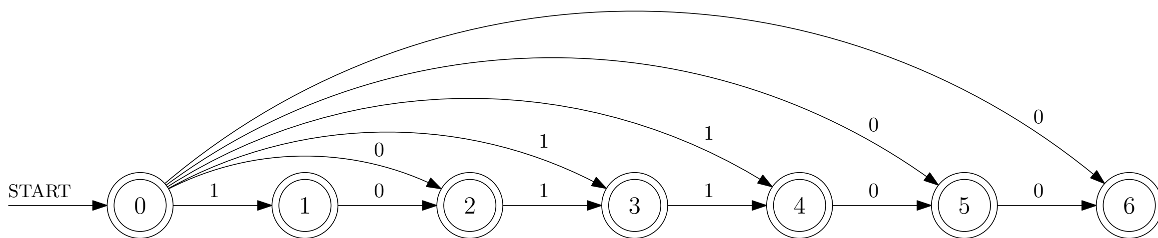
4.1.2 Faktorový automat

Vyhledávací automat zjišťoval, zda-li antislova nejsou faktory hledaného vzorku, a pracoval s předzpracováním AD (jakožto toho, co se vyhledává). Totéž lze uskutečnit i s předzpracováním vlastního hledaného vzorku P. Vyhledávací automat přejde po této úvaze v automat faktorový. Jedná se vlastně o úplnou indexaci hledaného vzorku P, ve kterém se vyhledávájí antislova jako jeho eventuální faktory. Je to tak proto, že automat přijímá předpony všech přípon (i nevlastní a prázdné – všechny stavy jsou koncové) indexovaného řetězce. Jeho podobu pro indexovaný vzorek '101100' s více počátky vidíme na obr. 4.4; po odstranění ε-přechodů a před determinizací pak na obr. 4.5.

Automat se tedy vybuduje nad vzorkem, jehož výskyt v zakomprimovaném textu se má zodpovědět, a poté se spustí nad všemi antislovy. Najde-li se některé z nich, je faktorem hledaného vzorku a proces lze ukončit s *negativní odpovědí*. Tento závěr vyplývá opět ze skutečnosti, kterou zachycuje lemma 4.1 – je-li antislovo faktorem vzorku, nemůže být vzorek faktorem komprimovaného textu.



Obrázek 4.4: Automat přijímající předpony přípon



Obrázek 4.5: Faktorový automat

Předzpracování hledaného vzorku, které konstrukcí faktorového automatu provádíme, může být výhodné naopak tehdy, hledáme-li stejná data např. ve velkém množství souborů zakomprimovaných metodou DCA. V takovém případě se automat vybuduje jednou a vstupem do něj budou antislovníky všech relevantních souborů. Narozdíl od vyhledávacího automatu zde pochopitelně nemá smysl takové předzpracování (např. ve formě tohoto faktorového automatu) ukládat ke komprimovaným datům, protože s nimi nijak nesouvisí – není vlastností jich, nýbrž jednoho konkrétního vyhledávacího dotazu. Výpočetní složitost takto realizovaného hledání (nepočítajíc konstrukci, resp. determinizaci automatu) bude $\mathcal{O}(\sum_{files^1} \sum_{AD} |w \in AD|) = \mathcal{O}(\sum_{files} |AD|)$.

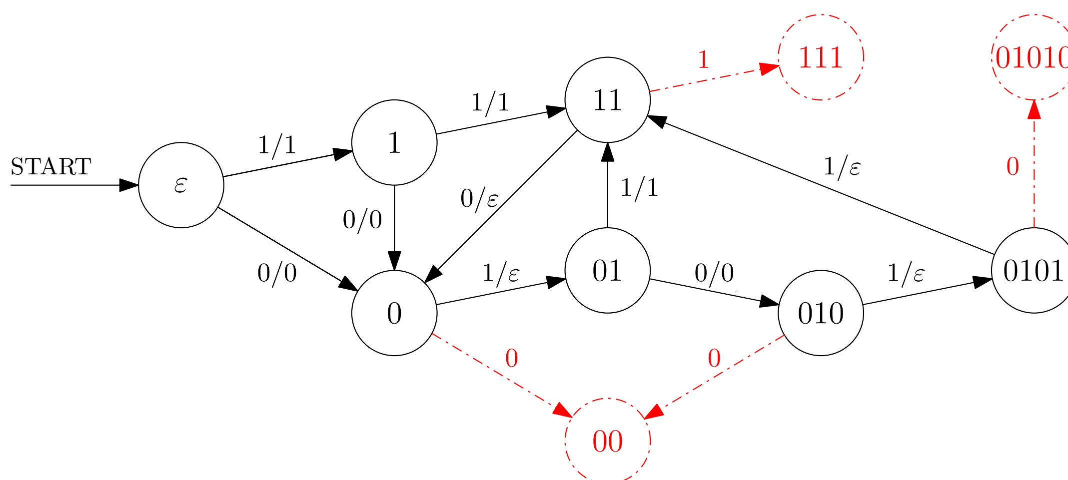
[1] množina všech relevantních souborů v případě hromadného hledání

4.2 DCA automat

Pokud bychom neměli k dispozici metadata komprese v podobě antislovníku, alternativně by se dal jako podklad pro vyhledávání vzít *DCA kompresní automat*. Jedná se o automat, který je u semi-adaptivního schématu DCA výsledkem prvního průchodu dat a vznikne např. úpravou *suffix trie* [18], viz podkapitola 3.3.1 (struktura kódující nalezená antislova). Automat je překladový, neboť realizuje vlastní kompresi a funguje dle popisu v podkapitole 3.4.1.

4.2.1 DCA kompresní automat

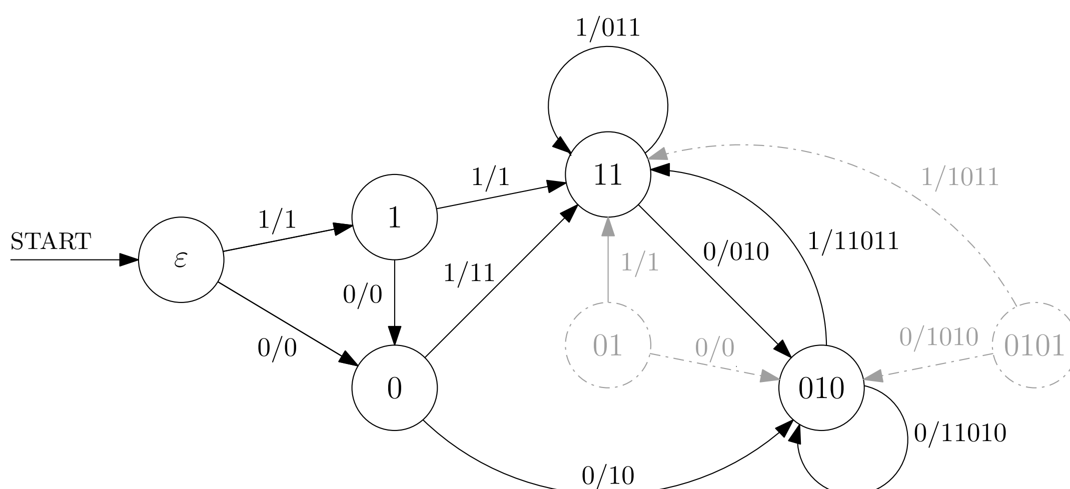
Na obr. 4.6 vidíme příklad *DCA kompresního* (překladového) *automatu* pro množinu antislov $AD = \{\text{'00'}, \text{'111'}, \text{'01010'}\}$ (odpovídá příkladům 3.1, 3.2 a např. textu '11011010110'). Stavy vyznačené červeně/čerchovaně jsou stavy odpovídající antislovům, tedy stavy „zakázané“, resp. takové, do kterých by se automat při normální činnosti (při kompresi textu) neměl dostat (výjimkou může být povolení techniky *almost antiwords* [21], představené v podkapitole 3.4.4). V pouhé implementaci komprese lze tyto zbytečné stavy, které by měly být navíc i nedosažitelné, vypustit.



Obrázek 4.6: DCA kompresní automat

4.2.2 DCA dekompresní automat

Automat pro inverzní činnost – *DCA dekompresní automat* můžeme vidět na obr. 4.7. Z kompresního automatu vznikne záměnou vstupů s výstupy pro všechny hrany jeho grafu. Hrany nereplikující vstup na výstup (právě ony provádějí reálnou kompresi) tímto přejdou v hrany s ε-přechody (podrobněji v podkapitole 3.4.1). Obrázek ukazuje situaci po jejich odstranění s adekvátní modifikací výstupní funkce (výstupní symboly přes ε-hrany jsou kumulovány do výstupního řetězce za lomítkem). Je zajímavé si povšimnout, že některé stavy se při této operaci stávají nepotřebnými a lze je pro další zpracování vypustit bez vlivu na



Obrázek 4.7: DCA dekompresní automat

překladovou relaci (zde jsou to šedé/čerchované stavy '01' a '0101'). Děje se tak proto, že stavy obsahující pouze ε-hrany jako vstupní se stávají po jejich odstranění stavy nedosažitelnými. Dekompresní automat vlastně mění informaci kódovanou tvarem grafu na složitější popis jeho hran.

4.2.3 Použití DCA automatu

Postup, kterým se pomocí DCA kompresního automatu ověří, jestli se hledaný vzorek v zakomprimovaném textu může vyskytovat, je jednoduchý. Automat vlastně detekuje stavy těsně před koncem antislova (stavy pro přípony *butlast*[2]$(w); w \in (AD)$, ze kterých vede pouze jeden přechod) a pokud se do některého z nich dostane při zpracování hledaného vzorku P (nikoliv textu T, který předtím zakomprimoval), jsou dvě možnosti následného vývoje:

Vzorek P buď dané antislovo jako svůj faktor neobsahuje a pak automat legálně přejde přes komprimující hranu (hranu nereplikující vstup na výstup), nebo ho obsahuje a automat přejít nebude moci (přechodová funkce není v tomto místě pro komplementární symbol definována). Takovou situaci při simulaci komprese odchytíme a můžeme ohlásit *negativní odpověď*. Realizaci je možno upravit i po formální stránce lépe tak, že se zakázané stavy v seznamu stavů ponechají jako stavy koncové. Činnost automatu je pak ukončena v okamžiku, kdy se do takového stavu dostaneme, a to s ohlášením *negativní odpovědi*. Zpracování celého textu bez dosažení koncového stavu značí *neurčitý výsledek* (hledaný vzorek se v zakomprimovaném textu nacházet „může, ale nemusí").

Postup s použitím DCA kompresního automatu se hodí zejména v případě, kdy již máme tento automat k dispozici (z kompresního procesu). Stačí ho totiž pustit na hledaný vzorek a nic dalšího není nutno konstruovat. Tento postup je tedy velice přímočarý a podobně jako u ostatních automatů nabízí možnost paralelizace pro více vzorků (udržujeme vícero aktuálních stavů a přechody počítáme pro každý z nich zvlášť). Má však také některé omezující vlastnosti, např. ztrátu možnosti *pozitivní odpovědi*.

[2] *butlast*() stejně jako u alg. 3.1

4.3 Pozitivní odpověď

Ačkoliv by se na první pohled mohlo zdát, že vyhledávání na základě AD metadat by mělo být schopno dávat na otázku výskytu vzorku pouze *odpověď negativní* („ne“) či *neurčitou* („možná ano“), při podrobnější analýze zjišťujeme, že za jistých okolností je možno výskyt vzorku v původním textu i s jistotou potvrdit. Tato třetí možná, *pozitivní odpověď* („určitě ano“), věštícího automatu vychází z implicitní vlastnosti AD obsahovat antislova v jejich minimální podobě (konvence v pozn. 3.3).

Minimalita antislov nám totiž zajišťuje důležitou skutečnost – jejich vlastní faktory se v původním textu vyskytovaly, neboť kdyby tomu tak nebylo, příslušné antislovo by bylo možno zkrátit odebráním bitu zpředu či zezadu a výsledkem by bylo opět antislovo (s efektivní výhodou toho, že by bylo kratší), což je v rozporu s předpokladem minimality antislova nezkráceného. Formálně zavedeme lemma 4.2 pro pozitivní odpověď.

Lemma 4.2 (Inkluze vlastních faktorů antislov)
Pro minimální[3]antislovník $AD(T)$ textu T platí:

$$\forall w \in AD(T) : PFact(w) \subset Fact(T)$$

Důkaz 4.2 (Lemma 4.2)
Dokazujeme vlastně pouze ekvivalenci def. 3.2 s 2. variantou zápisu rovnice 3.3:

Zapíšeme $w = bs\,(sb)$[4], pro $s \in B^*; b \in B$. To je ovšem negace důsledku implikace v def. 3.2, takže neplatí jeho předpoklad $s \notin Fact(T)$ a platí jeho negace $s \in Fact(T)$. Dle použité substituce je ovšem $s \in Fact(w)$ a zároveň $|s| < |w|$ (právě o krajní symbol b), takže dokonce $s \in PFact(w)$. Možina $PFact(w)$ však neobsahuje delší řetězce než s (delší je už jen w, ale to není svůj vlastní vlastní faktor) a pro všechny kratší s' je $s' \in Fact(s)$, a tudíž i $s' \in Fact(T)$ (event. můžeme použít indukci a substituovat dál na $w = bbs\,(sbb)$ až do $s = \varepsilon$). Proto musí $PFact(w) \subset Fact(T)$, čímž je důkaz ukončen.

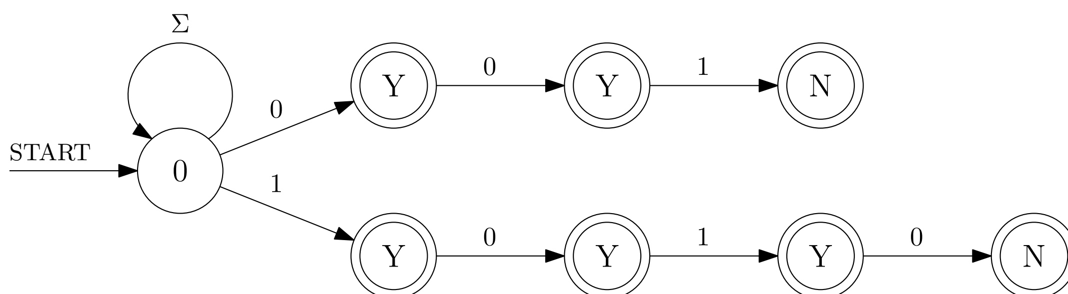
Dlužno ještě dodat, že povolení techniky *almost antiwords* nám zde, narozdíl od *negativní odpovědi*, výsledek nezkazí! Jsou totiž pouze dvě možnosti, čím může být prvek AD v případě povolení techniky *almost antiwords*. Buď je antislovem – antifaktorem, na který je kladen požadavek minimality, a tudíž dostaneme validní *pozitivní odpověď* dle lemma 4.2. Nebo antifaktorem není, tudíž je faktorem, a tak nám z principu nemůže generovat výsledek typu „false positive“. Způsobí pouze pozitivní odpověď v případě nalezení jeho faktorů, které se v textu vyskytují spolu s ním a to je žádoucí výsledek typu „true positive“. Toto je zřejmé a tudíž netřeba formalizovat.

[3]podle konvence v pozn. 3.3

[4]Zde si můžeme dovolit předpokládat existenci alespoň jednoho symbolu v řetězci, čiže $w \neq \varepsilon$, protože takové antislovo by nebylo antifaktorem žádného textu (ani pro $T = \varepsilon$).

4.3.1 Předpony minimálních antislov

Pro antislovník např. $AD = \{‘001’, ‘1010’\}$ je tedy zřejmé, že při dodržení požadavku na jeho minimalitu (konvence v pozn. 3.3) musel původní komprimovaný text obsahovat všechny vlastní faktory $PFact(AD) = \{‘0’, ‘1’, ‘00’, ‘01’, ‘10’, ‘101’, ‘010’\}$[5]. Pokud bychom se omezili z vlastních faktorů na vlastní předpony, pak automat, který by nalezení takové vlastní předpony oznámil koncovým stavem ‘Y’ („Yes“ – *pozitivní odpověď*) a nalezení celého antislova (nevlastní předpony) oznámil koncovým stavem ‘N’ („No“ – *negativní odpověď*) by byl jednoduchou modifikací vyhledávacího automatu pro více vzorků z obr. 4.2. Je naznačen na obr. 4.8.



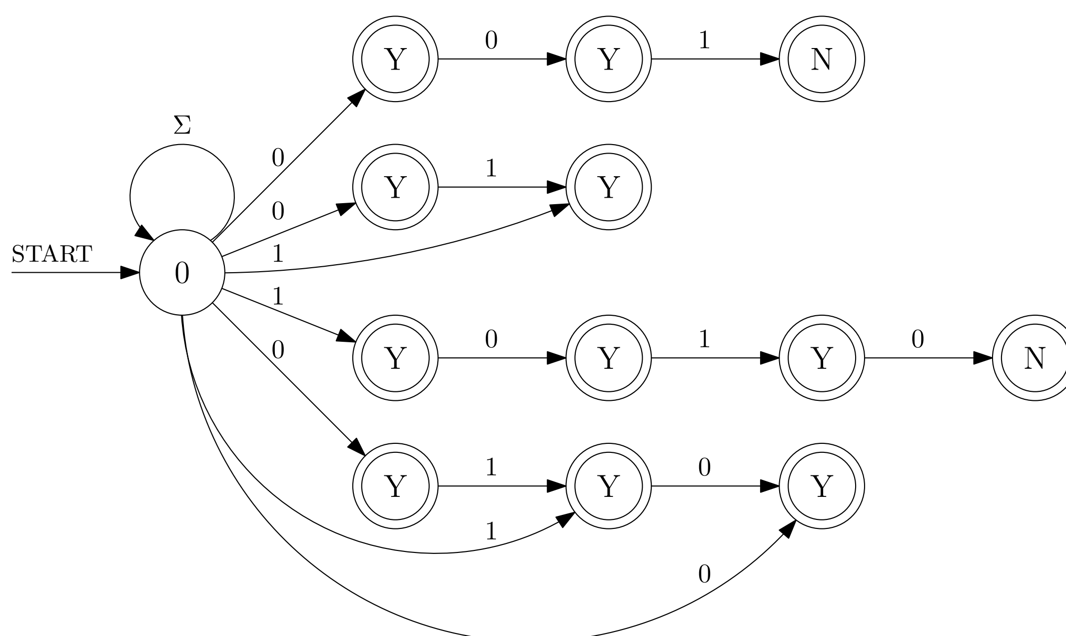
Obrázek 4.8: Nástin automatu pro pozitivní odpověď na základě předpon AW

4.3.2 Faktory minimálních antislov

Při obyčejném vyhledávání v textu lze od automatu přijímajícího jazyk všech předpon k automatu přijímajícím jazyk všech faktorů přejít pouze přidáním dalších hran (v triviálním případě pro získání více počátečních stavů). Např. automat z obr. 4.4 by bez počátků hranami „START“ shora přijímal pouze předpony. Teprve přidáním počátečních stavů doprostřed řetězce dochází k rozpoznání předpon i v rámci přípon, protože dodaná nedeterminičnost vlastně provádí paralení výpočet nejen na původním textu/vzorku, ale i na všech jeho příponách (začíná se dál než na prvním symbolu). Tok aktuálních stavů probíhá v automatu i nadále jedním směrem (v takto nakresleném doprava) a vše funguje.

V našem hledání však přechod od vlastních předpon k vlastním faktorům nelze udělat pouze přidáním hran. Toto vylepšení musí zohlednit fakt, že při nalezení antislova celého až do konce je třeba rozlišit, zda-li jsme se k němu dostali jako k příponě vlastní či nevlastní (bude rozdíl v odpovědi!). Je tedy nutné pro každé antislovo zavést nový faktorový „podautomat“ (sudá patra 2 a 4), který se bude starat o práci s vlastní příponou tak, jak naznačuje obr. 4.9. Jde o to, že pokud jsme se na konec antislova dostali odjinud, než z jeho začátku, jedná se o jeho faktor vlastní, a proto bude odpověď na konci těchto sudých pater *pozitivní*.

[5] Formálně bychom měli do množiny $PFact(AD)$ (analogicky i pro $Fact(AD)$) zahrnovat i prázdný řetězec ε. Pak by i počáteční stav odpovídajícího automatu byl koncovým s ohodnocením ‘Y’, protože vzorek ε se vyskytuje kdekoliv v každém textu. Hledání prázdného řetězce však nemá praktický význam.



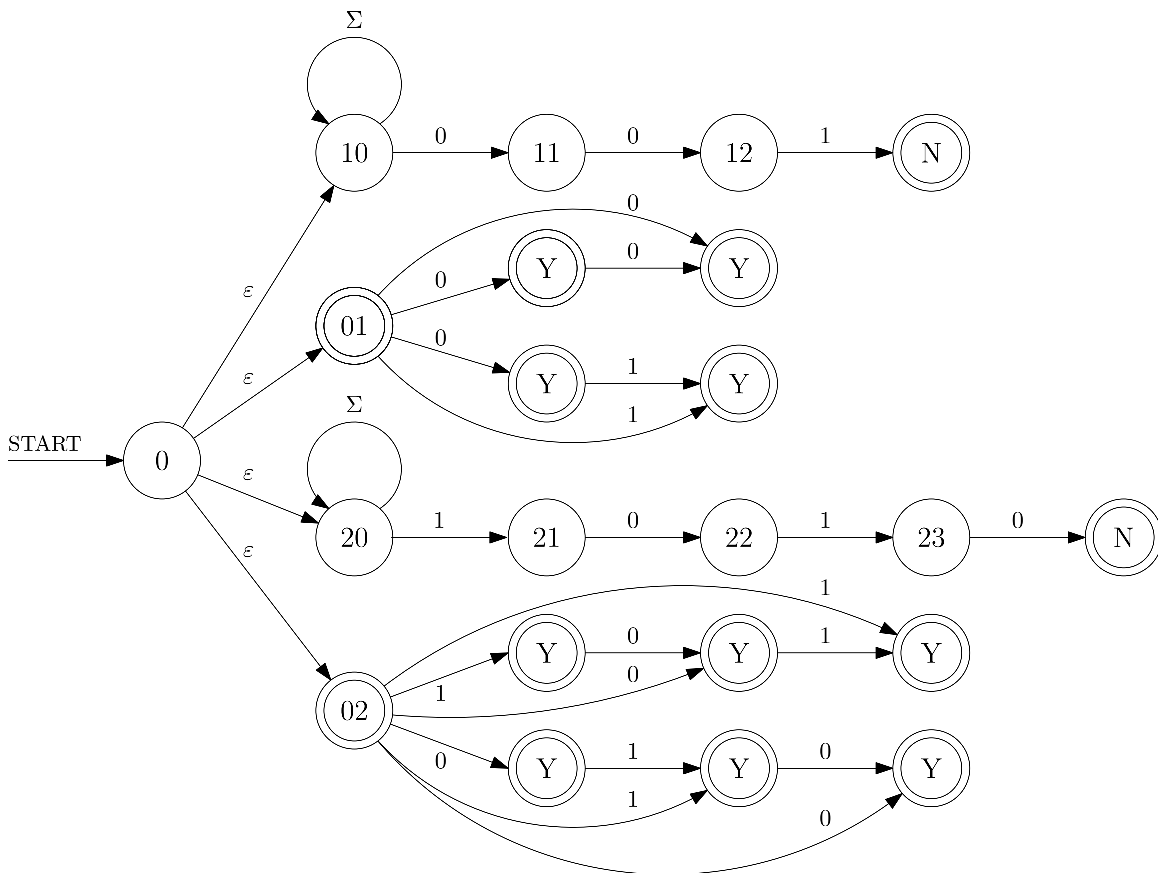
Obrázek 4.9: Nástin automatu pro pozitivní odpověď na základě faktorů AW

4.3.3 Kombinovaný věštící automat

Oba automaty z obr. 4.8 a 4.9 vykazují však jednu podstatnou závadu – řetězce pro *pozitivní odpověď* (vlastní předpony/faktory antislov) vyhledají ve vzorku P na vstupu jako jeho faktory nezávisle na tom, zda-li vzorek už před jejich začátkem nějaké symboly obsahoval. To samozřejmě odpověď automatu zcela diskvalifikuje (automat by odpovídal 'Y' i v případech neurčitého výsledku) a příčinou tohoto chování je společná *vyhledávací smyčka* v počátečním stavu automatu. Tato smyčka je pro činnost negativních větví nutná (hledaný vzorek může antislovo jako svůj faktor obsahovat kdekoliv), ale pro pozitivní větve škodlivá (při hledání vzorku jako faktoru antislova je vyčkávání v ní nežádoucí – místo vzorku se najde jeho přípona). Problém byl patrný už na první pohled, protože automaty z obr. 4.8 a 4.9 neobsahují jiné stavy než koncové s oběma typy *určitých odpovědí* ('Y' nebo 'N'). Z povahy metadat k dispozici pro rozhodování však bude mnoho situací, které nebudou moci skončit *určitou odpovědí*.

Oprava tohoto problému je založena na zajištění nezávislosti mezi pozitivími a negativními „podautomaty“ (lichá/sudá patra). Počáteční vyhledávací smyčku budou obsahovat pouze větve pro *negativní odpověď*. Souvislost grafu takto upraveného automatu udržíme doplněním ε-přechodů na jeho začátek. Jedná se vlastně o „nedeterministickou paralelizaci“ kompozicí různých typů jednodušších automatů. Toto opravené řešení ukazuje obr. 4.10 (stále pro $AD = \{\text{'001'}, \text{'1010'}\}$).

Uvedený problém by bylo možno řešit i prostým počítáním provedených přechodů, čímž bychom poznali, jak dlouho automat setrval ve vyhledávací smyčce. Tato technika však není čistá z hlediska teorie konečných automatů (KA nemá žádnou další paměť kromě své aktuální konfigurace), a proto preferujeme formální oddělení na úrovni „podautomatů“.



Obrázek 4.10: Opravený automat pro kombinovanou odpověď

Automat z obr. 4.10 je schopen *odpovídat negativně* v případě, že ve vzorku P najde antislovo jako jeho faktor – potom P se s jistotou nemůže v textu vyskytovat. Dále je schopen *odpovídat pozitivně* v případě, že P je naopak faktorem některého antislova, a to právě jeho *vlastním faktorem* (jinak by byl dotyčným antislovem a odpověď má být 'N'), protože implicitní minimalita antislov zajišťuje výstkyt takového vzorku v původním komprimovaném textu dle lemma 4.2.

Vzhledem k tomu, že ve své zobecněné nedeterministické podobě není automat z obr. 4.10 úplně definován (nemá přechody ze všech stavů pro oba možné bity na vstupu), ve zbylých případech odpovídá *neurčitou odpovědí*. Jsou to případy, kdy automat končí s přečteným vstupem ve stavech negativních větví jinde než úplně na konci.

Vzhledem k vyhledávací povaze negativních větví je jejich odpověď určující, tzn. v případě návštěvy koncového stavu 'N' je výsledkem věštícího vyhledávání *negativní odpověď* bez nutnosti dále číst vstup až do konce. Koncové stavy větví pro *pozitivní odpověď* tuto vlastnost samozřejmě mít nemohou.

Finální zobecnění automatu z obr. 4.10 vzhledem k AD je vlastně **sjednocením vyhledávacích automatů pro všechna antislova a faktorových automatů pro jejich vlastní faktory**. Formálně se nad antislovníkem $AD(T)$ a pro vstup P chová takto:

- (1) přijímá ve stavech 'N', pokud $\exists AW \in AD : AW \in \mathit{Fact}(P)$,
- (2) přijímá ve stavech 'Y', pokud $\exists AW \in AD : P \in \mathit{PFact}(AW)$,
- (3) nepřijímá ve zbylých případech (což má význam *neurčité odpovědi*).

4.4 Problém zarovnání dat

Myšlenka vyhledávání v datech zakomprimovaných metodou DCA má jeden zásadní problém. Praktické vyhledávání v textu většinou znamená vyhledávání řetězců nad abecedou ASCII znaků či některou z jejích nadstaveb pro různá kódování národních abeced (Unicode, UTF apod.). Základní entitou takového přístupu je bajt (8 bitů) a pro znaky jak textu, tak vyhledávaných vzorků proto platí, že vždy začínají na bitu s indexem celočíselného násobku 8 – data jsou typicky *zarovnána*.

Komprese DCA však pracuje nad binární abecedou – s jednotlivými bity a informace o hranicích mezi znaky není v bitovém proudu obecně nijak zaznamenána. Používaná antislova nijak nectí začátky a konce bitových jednotek, které jsou interpretovány o vrstvu výše – jedná se o kompresi na *bitové úrovni*. Informace o jednotlivých znacích je tak ztracena a její rekonstrukce by vyžadovala dekompresi dat spolu s počítáním indexu bitu, a to buď od začátku datového proudu nebo, v závislosti na možnostech implementace, od nějakých pomocných značek uvnitř proudu.

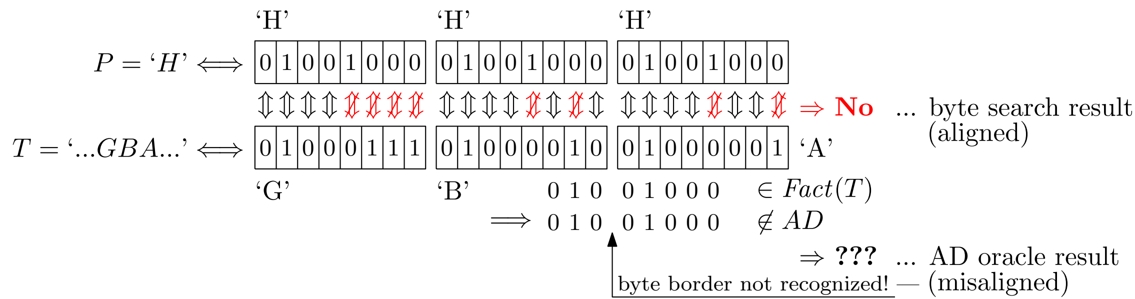
Naše situace, kdy se původní data nedekomprimují a vyhledávání je prováděno pouze na základě znalosti kompresních metadat, vylučuje možnost jakéhokoliv eventuálního dopočítávání hranic bajtů a ty jsou v tomto smyslu ztraceny zcela. Prvky antislovníku neobsahují z informace o pořadí bitů v rámci jednotlivých bajtů původního textu nic. Kvůli absenci této informace a nemožnosti ji z metadat jakkoliv rekonstruovat **je vyhledávání na základě DCA metadat omezeno na původní bitovou podstatu metody**. Výsledky vyhledávacích dotazů tak nelze použít přímo k interpretaci na bajtové, potažmo znakové úrovni.

4.4.1 Zarovnání v praxi

Výše uvedené v praxi znamená, že vzorek k vyhledání v DCA komprimovaném textu je nutno chápat jako *řetězec bitů* a nikoliv jako posloupnost bajtů s velikostí $\frac{1}{8}$ počtu bitů. Odpověď věštícího automatu je nutno chápat stejně nízkoúrovňově – tak, že daná *bitová sekvence* se v původním textu buď vyskytovat mohla, avšak na jakémkoliv místě (obecně nezarovnaném na osmice či jiné n-tice bitů), nebo se v textu nevyskytovala, a to ani nezarovnaně (případně ještě analogicky pro eventuální *pozitivní odpověď*).

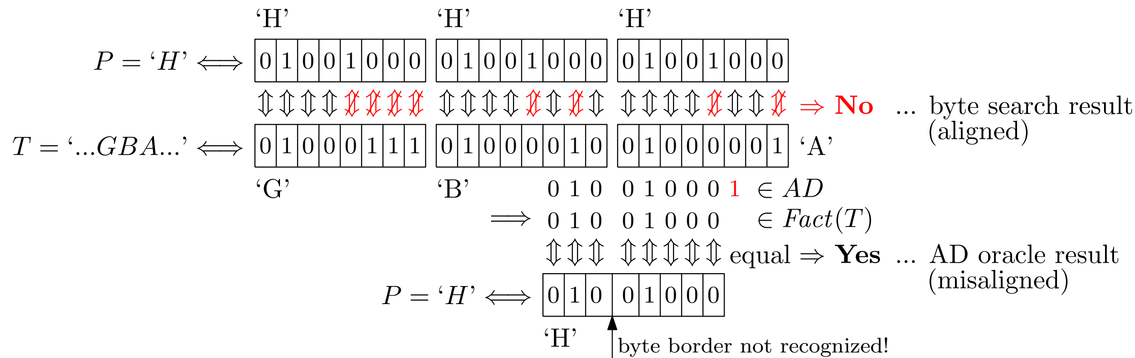
Jelikož podezření na výskyt je pouze neschopnost vyloučit opak, nejedná se o aspekt, který by automatu znemožňoval pracovat úplně. Problém se projevuje tím, že síla odpovědí se zmenšuje monotónně s velikostí bitových jednotek, které jsou interpretovatelné pro vyšší vrstvu, jež věštícímu automatu klade vyhledávací dotazy. Je tomu tak proto, že v některých situacích automat není schopen výskyt, např. znaku 'H' (z pohledu automatu řetězce

'01001000', pokud pracujeme ve standardu ASCII) vyloučit, protože tímto znakem reprezentovaný řetězec se v původním textu vyskytuje. Vyskytuje se v něm ovšem pouze v nezarovnané podobě (přes hranici více jednotek), což je ona informace, která automatu zůstává skryta. Tuto situaci ilustruje obr. 4.11.



Obrázek 4.11: Problém zarovnání dat – neurčitá odpověď

Pozor je třeba dát v případě, pokud automat dává *pozitivní odpověď*. Znamená to potvrzený výskyt v *obecně nezarovnané podobě* a bylo by třeba jinou metodou (citlivou na hranice bitových jednotek) určit, zda-li se jedná o pravý výskyt daného řetězce bitů (se správným zarovnáním), nebo jde o výskyt *nezarovnané bitové sekvence*. V tomto smyslu automat pozitivně odpovídá *podezřením z výskytu* vzorku v textu a pokud bychom toto nebrali na zřetel a nesprávně interpretovali výskyt automaticky v zarovnané podobě, jednalo by se o výsledek typu „false positive". Tuto situaci ilustruje obr. 4.12.



Obrázek 4.12: Problém zarovnání dat – pozitivní odpověď

Kapitola 5

Realizace

V následující kapitole se budeme věnovat zejména implementaci vyhledávacího automatu pro kombinovanou odpověď z podkapitoly 4.3.3 a jeho testování na referenčních datech, konkrétně na kolekci souborů obsažených v *Canterbury Corpusu*[1]. *Canterbury Corpus* je běžně používaný benchmark vzniklý v roce 1997 na univerzitě v Canterbury a slouží k testování kompresních metod (především bezeztrátových). Nahrazuje starší *Calgary Corpus*[2].

Dále zde popíšeme změny, které byly provedeny ve zdrojových kódech kompresní knihovny *ExCom* při integraci funkcionality vyhledávání nad metadaty v souborech zakomprimovaných metodou DCA.

5.1 Implementace

5.1.1 Programovací a vývojové prostředí

Vlastní implementaci jsme prováděli na CPU architektuře IA32 (x86, resp. x86-64), platformě PC s operačním systémem GNU/Linux nebo Microsoft Windows v prostředí Cygwin[3](port POSIXových nástrojů pro Windows). Jako výchozí programovací jazyk jsme zvolili C/C++ (hlavně C++ kvůli návaznosti na existující implementaci DCA [23]) a jako jeho kompilátor gcc[4]/g++ (GNU C Compiler), na Windows spolu s MinGW.

Při psaní zdrojových kódů jsme nepoužívali žádné konkrétní IDE či grafické prostředí, ale pouze textový editor (konkrétně *vim*[5]– „VI improved“), pro sestavování zdrojů konzolový nástroj *make*[6](GNU make) a pro ladění konzolový debugger *gdb*[7](GNU debugger). Multiplatformita kódů je zajištěna dodržováním standardu ANSI C, resp. ISO C 99 a ISO C++ 98, a přiložením Makefilů (pravidla pro sestavení pomocí *make*) pro nejběžnější systémy – UNIX/Linux a Windows.

[1] http://corpus.canterbury.ac.nz
[2] http://www.data-compression.info/Corpora/CalgaryCorpus/
[3] http://www.cygwin.com
[4] http://gcc.gnu.org
[5] http://www.vim.org
[6] http://www.gnu.org/software/make/
[7] http://www.gnu.org/software/gdb/

Jelikož v rámci práce nevznikal jeden větší programový celek, ale několik menších programů a dalších podpůrných utilit (použitých např. v rámci procesu automatizovaného testování, viz podkapitola 5.2.2), nebyl důvod k nasazení další úrovně automatizace sestavování kódů, např. pomocí balíku *autotools*, a zůstali jsme u modelu manuální invokace *make*u z shellu (příkazové řádky). Další úroveň automatizace jsme naopak použili při testování, a to pomocí řady konfigurovatelných shellových skriptů, konkrétně pro interpret *bash* (v případě možnosti zachování zpětné kompatibility pro *Bourne shell*) – více v podkapitole 5.2.3.

5.1.2 Rešerše knihoven pro práci s KA

Na počátku vlastní implementace jsme provedli rešerši existujícího softwaru se zaměřením na „Automata tools" (nástroje pro automaty). Velmi rozsáhlý a ucelený přehled takových nástrojů, spolu s jejich krátkými popisy a odkazy na domovské weby, jsme našli na `http://members.fortunecity.com/boroday/Automatatools.html`. Naše požadavky byly:

- implementace/rozhraní v C/C++,
- schopnost determinizace NKA,
- schopnost výpočtu překladového KA (jeho běhu).

Pro naše účely by byly použitelné knihovny *AT&T FSM Library* nebo *Grail*. Obě jsou k dispozici zdarma pro nekomerční účely.

5.1.2.1 AT&T FSM Library

`http://www.research.att.com/sw/tools/fsm/`
Jedná se o knihovnu pro všeobecné použití s API rozhraním v jazyce C. Obsahuje asi 30 operací s automaty včetně podpory pro jejich vizualizaci (v kombinaci s nějakým grafickým procesorem, např. *Graphviz*[8]). Ve vývojových laboratořích AT&T je použita např. k rozpoznávání řeči nebo analýze DNA sekvencí (typická multidisciplinární úloha pro stringologii). Její jádro je založeno na matematické teorii racionálních řad a mělo by umět pracovat řádově s desítkami milionů stavů/přechodů.

5.1.2.2 Grail

`http://www.cs.ust.hk/~dwood/grail/index.html`
Grail je symbolické výpočetní prostředí pro práci nejen s KA, ale i s regulárními výrazy apod. Mezi jednotlivými takovými objekty formálních jazyků lze převádět, minimalizovat, determinizovat, počítat doplňky atd. Software je napsán v jazyce C++ a od verze 2.4 podporuje také Mealyho automaty. Líbí se nám, že jeho stupeň parametrizace v kombinaci s velkou flexibilitou C++ umožňuje definovat abecedu KA/regexpu i pomocí uživatelského datového typu. Hlavními autory projektu jsou Darrell Raymond (University of Western Ontario) a Derick Wood (Hong Kong University of Science and Technology).

[7] `http://www.gnu.org/software/bash/`
[8] `http://www.graphviz.org`

5.1.3 Návaznost na implementaci DCA

Existující implementaci DCA [23] jsme využili pro výpočet antislovníku. Protože v závěru implementace budeme chtít integrovat vyhledávání do *ExCom*, vyšli jsme ze zdrojových kódů DCA upravených Filipem Šimkem [26]. Úpravy se týkají odstranění některých funkcionalit, které v kompresní knihovně nejsou potřeba (byly přítomné vesměs pro ladicí/testovací účely), zlepšení kompatibility pro kompilování na 64-bitových platformách a opravení několika memory leaků (paměťových úniků).

Po prostudování [23] a seznámení s kódem zde uvádíme několik zásadních poznatků:

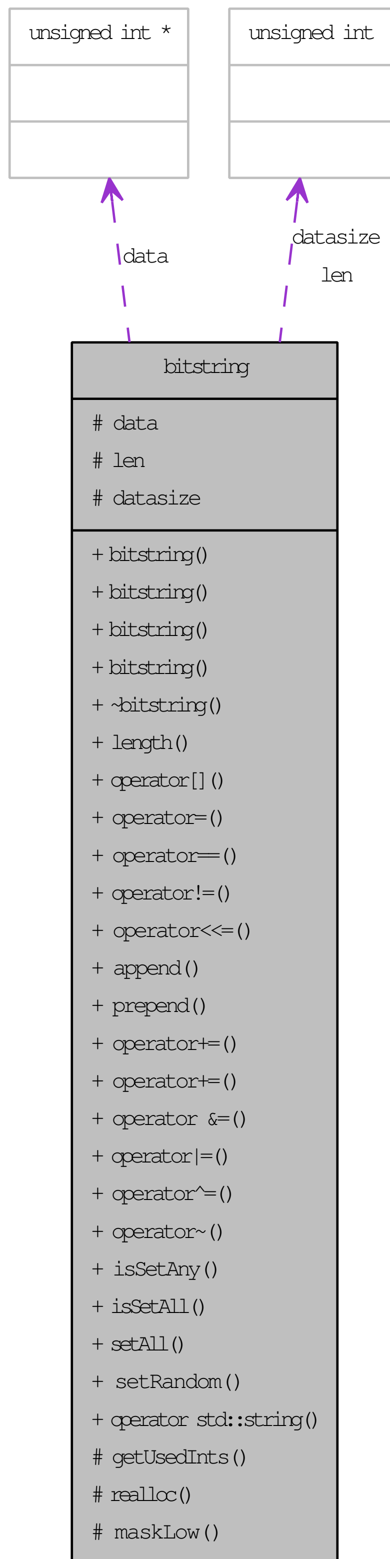
- celá implementace pracuje s bajty v režimu MSb first, tzn. v datovém proudu se zpracovává nejprve 7. bit bajtu a jako poslední bit 0 (platí pro čtení i zápis),
- po sestavení *suffix trie* se vždy provádí *simple pruning* – prořezávání stromu podle zisku jednotlivých antislov (podrobněji na [23, s. 37]),
- v semi-adaptivní verzi DCA (adaptivní je v adresáři `dynamic/`, více v podkapitole 5.1.5) je antislovník konstruován vždy jako *minimální* a lze na to tudíž spoléhat při pozitivní odpovědi vyhledávání,

a dále jsme provedli nějaké vlastní úpravy kódu, zejména:

- oprava módu otevírání souborových proudů na binární,
- nahrazení funkce `bzero()` standardní funkcí `memset()`,
- podmíněné zakázání nestandardních funkcí z hlavičkového souboru `<sys/resource.h>` makrem `PROFILING`,
- podmíněné zakázání *simple pruning* makrem `USE_SIMPLE_PRUNE`,
- deklarace spřátelených tříd (`ADaccess`, `DCAxgen`) pro přístup k interním strukturám antislovníku.

5.1.4 Implementace vyhledávání

Původně jsme zamýšleli použít knihovnu *Grail*, s její pomocí vytvořit automat podle 4.3.3, nechat zdeterminizovat a spustit nad hledaným vzorkem. Vzhledem k tomu, že automat v této podobě má mnoho stavů $((\sum_{AD}|w \in AD| + 1) \times 3 - |AD| \times 3 + 1 = \sum_{AD}|w \in AD| \times 3 + 1 = \mathcal{O}(|AD|))$ kvůli velikosti antislovníku a jeho obecná determinizace by byla paměťově i časově velmi nepříznivá, od této myšlenky jsme upustili. Velikost prořezaného antislovníku pro soubory 100–1000 KB a hloubku komprese 30 bitů se může pohybovat řádově ve stovkách tisíc (bitů). Plný (neprořezaný) AD může být ještě např. 5–10× větší.



Obrázek 5.1: Class diagram třídy bitstring

Poznatky v [34] a [24] nás inspirovaly k myšlence automat simulovat v jeho nedeterministické podobě. Jako základ jsme se rozhodli použít algoritmus 4.2 [24, s. 60] – simulace výpočtu NFA pomocí bitového vektoru. Bitový vektor zde kóduje množinu všech současně aktivních stavů automatu a jeho aktualizace se provádí logickými operacemi nad přechodovou funkcí, která zobrazuje do množiny všech stavů (z přechodové relace NFA jsme získali funkci seskupením všech přechodů do jedné množiny následujících stavů, viz. def. 2.19 a 2.20). Pokud je velikost vektoru menší než bitová šířka konkrétní procesorové architektury, je možno navíc využít *bitový paralelismus*, což ovšem není náš případ (vektor je dlouhý kvůli velkému počtu stavů).

Implementaci simulace NFA jsme se rozhodli napsat vlastní, což má oproti možnostem univerzálních knihoven několik nezanedbatelných výhod:

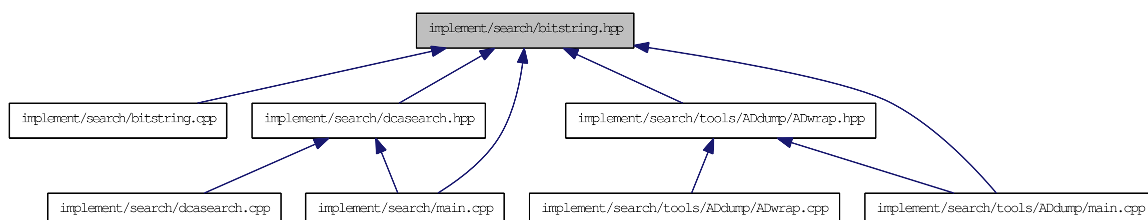
- možnost optimalizace pro binární abecedu (pracujeme na úrovni jednotlivých bitů),
- možnost předčasného konce výpočtu ('N' odpověď),
- možnost extrakce dalších informací z průběhu výpočtu, zejména počtu antislov, které rozhodly o výsledku, viz dále.

5.1.4.1 Třída bitstring

Pro reprezentaci bitového vektoru aktivních stavů jsme využili C++ třídu `bitstring`. Tuto třídu jsme vytvořili původně pro práci se vzorky, které jsou vlastně bitovými řetězci. Třída uchovává posloupnost bitů v poli celočíselných proměnných – neplýtvá místem (nejde o pole `bool`ů a jeden bit zde zabírá jen $\frac{1}{8}$ bajtu, kromě bitů v poslední buňce). Lze ji inicializovat/přetypovat na třídu `std::string`, k jednotlivým bitům lze přistupovat prostřednictvím indexovacího operátoru (výsledkem je spřátelená třída `bit`, do které je možné přiřazovat referencí) a i ostatní funkcionality jsou založeny hlavně na přetěžovaných operátorech.

Pro využití této třídy i v simulaci došlo k jejímu rozšíření. Bitový řetězec lze libovolně zvětšovat (třída provádí automatickou realokaci v případě potřeby), provádět bitové posuny, testovat nastavení všech bitů atd. Její třídní diagram ukazuje obr. 5.1.

Třídu jsme využili ve 2 programech – ve vlastním vyhledávání (`dcasearch`) a v pomocném nástroji pro textový výtisk antislovníku v semi-adaptivně komprimovaných `*.dca` souborech (`ADdump`). Pro stručnou představu přikládáme diagram jejího vložení v souborech zdrojových kódů na obr. 5.2. Kompletní výstup z dokumentačního nástroje Doxygen[9]nalezneme na CD, viz kapitola B).



Obrázek 5.2: Vložení hlavičky třídy bitstring ostatními moduly

5.1.4.2 Třída ADsearcher

Vlastní vyhledávání – simulaci běhu automatu – provádí třída `ADsearcher`. Jejím vstupem je hledaný vzorek (instance třídy `bitstring`) a antislovník původního textu. AD se dodává sekvenčně po antislovech voláním metody `addAW()`. Definována je i metoda pro mazání antislova `delAW()`, resp. celého AD `delAD()`. Tento model nám umožňuje načtení AD i z externího souboru (využíváno při testování). Navíc je možné měnit ho i mezi jednotlivými vyhledávacími dotazy, protože zde už neprobíhá žádné předzpracování (jen realokace vektoru stavů automatu v případě potřeby).

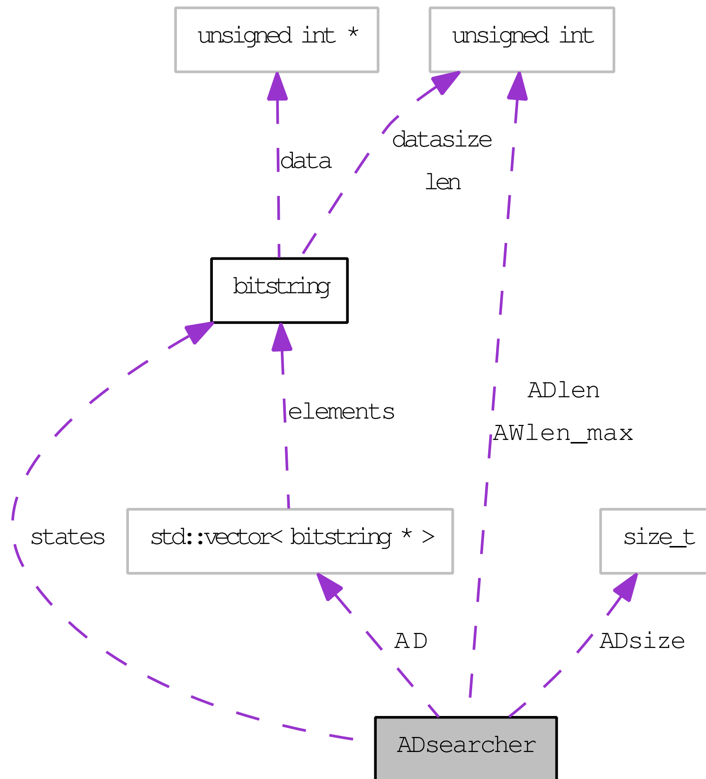
Jádrem třídy je vhodné mapování stavů automatu na index bitu v instanci `states` (typu `bitstring`) a kód pro realizaci přechodů – nastavování aktivních stavů. Jediný ε-přechod na začátku automatu jsme rozvětvili implicitně – nahradili více počátky. Rovněž neudržujeme stavy $10, 20 \ldots n0 \,; n < |AD|$, které jsou pořád aktivní kvůli přítomnosti vyhledávací smyčky (vyhledávají prázdnou předponu). Naopak duplikujeme stavy $01, 02 \ldots 0n \,; n < |AD|$ kvůli optimalizaci indexovacích výrazů. Toto nám paměťovou složitost, která je asymptoticky stále $\mathcal{O}(|AD|)$, zhorší jen zanedbatelně (pro AD s hloubkou k jen o faktor $\frac{1}{3k}$).

Časová složitost běhu simulace NKA je horší než při běhu DKA. Ušetření času za determinizaci se zde projevuje nutností pracovat s vektorovou přechodovou funkcí a z původně lineární složitosti $\mathcal{O}(|P|)$ (kde P je délka vzorku) se stává $\mathcal{O}(|P| \times |AD|)$ (máme $\sim |AD|$ stavů). Takto vzrůstají „variabilní náklady“ na hledání za současného poklesu „fixních“. V případě, že máme v plánu hledat jen jeden či několik málo vzorků (běžná situace při vyhledávání), tuto skutečnost spíše vítáme.

Výsledkem volání instanční metody `search()` je struktura `sADsearch_result`, ze které je možné vyčíst vyhledávací odpověď (‘N’, ‘Y’ nebo ‘MAYBE’) a dále počet antislov `AWcnt`, které rozhodly o výsledku (o negativním i pozitivním výsledku může současně rozhodnout více AW). Metodu je možné volat s parametrem `fast`, který povoluje možnost předčasného

[9]http://www.doxygen.org

ukončení výpočtu v případě jisté odpovědi ('N' pro antislovo jakožto faktor vzorku). Toto výrazně šetří dobu výpočtu, ale zneplatňuje hodnotu položky `AWcnt`, takže při testování jsme tuto možnost nevyužívali. Při volání z knihovny *ExCom* lze tuto optimalizaci zapnout argumentem na příkazové řádce. Kolaborační diagram třídy `ADsearcher` je na obr. 5.3. Další podrobnosti nalezneme přímo ve zdrojových kódech.



Obrázek 5.3: Kolaborační diagram třídy ADsearcher

5.1.4.3 Chybová odpověď

Za normálních okolností jsou vyhledávací odpovědi z volání `ADsearcher::search()` buď 'N', 'Y' nebo 'MAYBE' v souladu s analýzou. V případě, kdy je zakázána možnost předčasného ukončení výpočtu, je metoda schopna rozlišit ještě čtvrtou odpověď – 'ERROR'. Tato odpověď vzniká tehdy, když jsou současně splněny podmínky pro pozitivní i negativní odpověď (při předčasném ukončení výpočtu na první negativní odpovědi se toto nezjišťuje). To značí použití antislovníku, který *není minimální*, což je nesplnění předpokladu pro užití vyhledávání v DCA metadatech. Naše implementace však toto rozpozná a v případě úmyslného použití neminimálního AD lze odpověď 'ERROR' interpretovat jako odpověď 'N', protože negativní odpověď je prioritní (všechna antislova jsou antifaktory textu). Chybová odpověď je tedy informací navíc o použití nesprávného antislovníku (dle 3.3). Není to však test minimality celého AD!

5.1.5 Integrace vyhledávání do ExCom

Integrace do kompresní knihovny *ExCom* se ukázala jako problematická. *ExCom* totiž adoptuje pouze dynamickou verzi DCA (s adaptivním schématem, viz def. 2.44), která komprimuje jednoprůchodově, a kompletní antislovník je známý až po načtení celého vstupu (nebo zapsání celého výstupu při dekompresi). Navíc ani tehdy není splněna podmínka minimality antislovníku, jak vyplývá z dokumentace [23, s. 24], kde se říká: „V případě dynamické komprese nepotřebujeme množinu minimálních antislov, protože kompresor i dekompresor sdílí stejnou *suffix trie* a mohou použít všechny dostupné antifaktory přímo.“

I přes tento problém jsme vyhledávání do *ExCom* integrovali s tím, že v případě vlivu neminimality AD na výsledek vyhledávání platí vše z podkapitoly 5.1.4.3. Zběžné testování potvrdilo výše uvedené. Rovněž je nutné nejprve nechat provést dekompresi souboru pro získání jeho antislovníku (který u semi-adaptivního schématu je uložen odděleně na začátku souboru). To činí samotné vyhledávání prostřednictvím *ExCom* v praxi málo použitelné a bylo integrováno z důvodu požadavku v zadání práce.

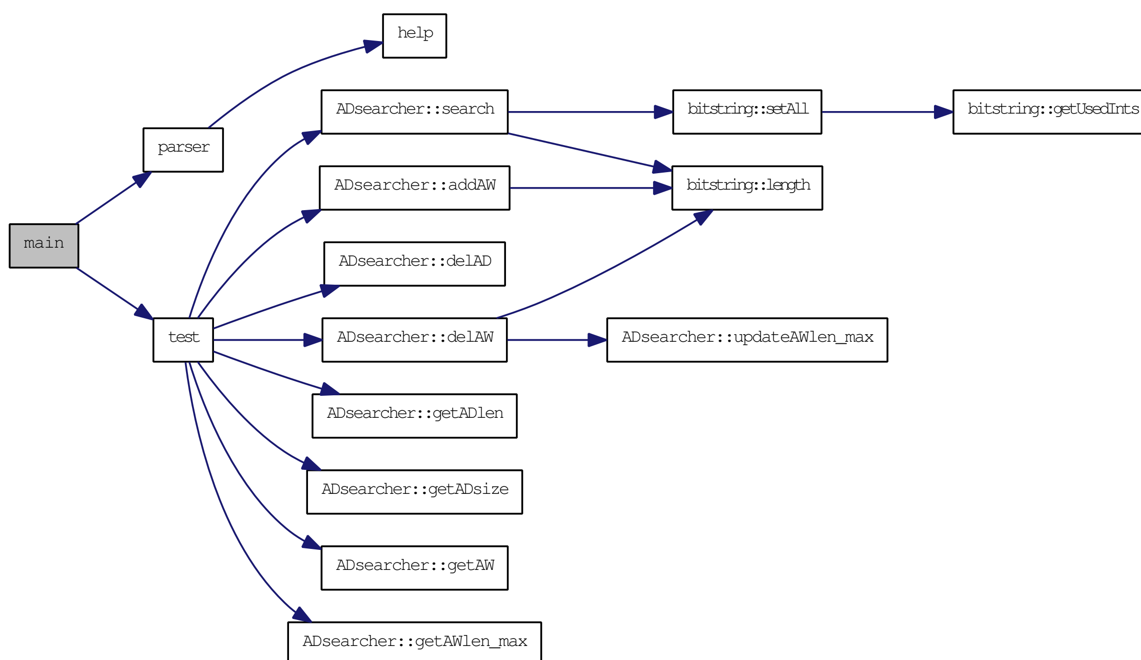
Změny ve zdrojových kódech *ExCom* lze shrnout takto:

- přidání nové operace, kterou mohou kompresní moduly poskytovat – „vyhledávání v metadatech“,
- přidání parametrů vyhledávání a hodnot výsledků modulu DCA,
- doplnění zdrojů o vlastní vyhledávací funkcionalitu v rámci metody DCA (třídy `ADsearcher`, `ADaccess` – extrakce AD pro naplnění vyhledávače),
- rozšíření testovací aplikace o podporu nové funkcionality:
  - přidání globálního argumentu `-s`, který vyvolá operaci vyhledávání (s vyšší prioritou než komprese/dekomprese),
  - přidání argumentu `p` pro hledaný vzorek a `e` pro rozšířené vyhledávání (zakázání předčasného ukončení běhu pro výpočet a výstup počtu výsledek rozhodujících antislov),
  - protože argument `p` přijímá řetězec, bylo třeba přidat makro STRING_ARG pro načítání řetězcových parametrů metody.

Podrobný návod k doplňování nových funkcionalit do *ExCom* je v příloze F práce [26].

5.1.6 Nástroj search_cli

Nástroj `search_cli` (umístění viz příloha B) je konzolová, interaktivní verze vyhledávání. Jeho rozhraní má několik příkazů (info, add, load, del, delall, get, search, mass, quit) a je navrženo tak, aby bylo možno použít jak pro „hraní“, tak i pro dávkové zpracování hromadných dat. Dávkové zpracování v tichém módu, řízené shellovým skriptem, se používá v experimentech. Ovládání je jednoduché a intuitivní, proto ho zde nebudeme popisovat. Program je možné spustit i ve verbózním módu přepínačem `-v`; nápovědu získáme přepínačem `--help` (toto platí i pro další programy, které budeme zmiňovat). Pro představu o vnitřní organizaci kódu přikládáme volací diagram programu na obr. 5.4.



Obrázek 5.4: Volací diagram programu search_cli

5.2 Testování

5.2.1 Metodika testování

Chování vyhledávání v DCA metadatech jsme testovali především na souborech z *Canterbury Corpusu*. Experimenty mají následující dimenze:

- typ textu – výběr souboru z korpusu,
- typ vzorku – bitová sekvence náhodná nebo obsažená v textu,
- délka vzorku (v bitech),
- hloubka antislovníku (maximální délka antislova),
- použití plného nebo prořezaného antislovníku.

Výčet souborů obsažených v *Canterbury Corpusu* s popisem jejich obsahu a velikostí v bytech máme v tab. 5.1. Z celkem 11 souborů je 8 textových, z čehož 4 jsou zdrojové kódy (cp.html, fields.c, grammar.lsp, xargs.1). Zbylé 3 soubory jsou binární (kennedy.xls, ptt5, sum). Všem souborům byla během experimentů věnována stejná pozornost.

Interval testovaných délek vzorků byl po úvodním experimentování zvolen tak, aby pokryl oblast změn naměřených charakteristik s malou rezervou na obou koncích. Pro zlepšení čitelnosti grafů a jejich vizuálního srovnávání byl pro podobná měření pokud možno ujednocen.

soubor	typ	kategorie	velikost
alice29.txt	text	English text	152089
asyoulik.txt	play	Shakespeare	125179
cp.html	html	HTML source	24603
fields.c	Csrc	C source	11150
grammar.lsp	list	LISP source	3721
kennedy.xls	Excl	Excel Spreadsheet	1029744
lcet10.txt	tech	Technical writing	426754
plrabn12.txt	poem	Poetry	481861
ptt5	fax	CCITT test set	513216
sum	SPRC	SPARC Executable	38240
xargs.1	man	GNU manual page	4227

Tabulka 5.1: Soubory v Canterbury Corpusu

Po úvodním experimentování jsme zjistili, že hloubka AD vesměs pouze posouvá charakteristiky po ose délky vzorku. Vzhledem k velkému počtu zbylých dimenzí a za účelem redukce počtu vzniklých grafů byla tato zafixována na 30 bitů. Taková hodnota byla zvolena s ohledem jak na přesnost měření, tak na rychlost/dobu zpracování všech experimentů a velikost vzniklých antislovníků (velikost AD roste obecně exponenciálně s jeho hloubkou). Hloubka 30 bitů je výchozí hodnotou i v implementaci DCA [23], kde nalezneme rovněž grafy kompresních poměrů, rychlostí a paměťových nároků v závislosti na ní.

Funkcionalita prořezávání AD (podkapitola 3.4.3) šetří jeho velikost a zlepšuje kompresní poměr. Na vyhledávání bude mít však negativní dopad, protože do AD se dostává méně informací z původních dat. Kritérium v implementaci [23] způsobuje jeho redukci na cca $\frac{1}{5}$–$\frac{1}{10}$ původního objemu (v bitech). Je tedy zajímavé zjišťovat i dopad této optimalizace na výsledky/použitelnost vyhledávání ve vzniklém AD. Za tímto účelem jsme do implementace dopsali možnost vypínání prořezávání (implicitně bylo vždy zapnuto, protože při kompresi/dekompresi nemá jeho vypínání smysl) a všechny experimenty jsme provedli jak na plném (full) tak prořezaném (pruned) AD.

Při experimentech byly měřeny tyto charakteristiky (v závislosti na všech dimenzích rozebraných výše):

- poměr výskytu neurčité odpovědi ('MAYBE') při vyhledávání na náhodných vzorcích,
- poměr výskytu pozitivní odpovědi ('Y') ve všech určitých odpovědích (nepočítaje 'MAYBE') při vyhledávání na náhodných vzorcích,
- poměr výskytu neurčité odpovědi ('MAYBE') při vyhledávání na vzorcích, které byly v textu obsaženy (ostatní odpovědi jsou 'Y'),
- počet antislov, které rozhodly výsledek pro negativní odpověď ('N') při vyhledávání na všech vzorcích (náhodných i pozitivních),
- počet antislov, které rozhodly výsledek pro pozitivní odpověď ('Y') při vyhledávání na všech vzorcích (náhodných i pozitivních).

Mohli bychom zjišťovat ještě citlivost vyhledávání na zarovnání dat (viz podkapitola 4.4), ale není žádný důvod, abychom naměřili jinou statistickou hodnotu pravděpodobnosti správné odpovědi i vzhledem k zarovnání na bajty, než $\frac{1}{8}$.

V rámci našich experimentů bylo provedeno celkem 187 200 vyhledávacích dotazů – 312 průchodů souborů náhodných vzorků konstantní délky po 500 kusech a 26 průchodů souborů pozitivních vzorků různých délek po 1200 kusech.

5.2.2 Pomocné nástroje

Pro usnadnění testování jak správnosti vyhledávacích odpovědí, tak vlastních experimentů, jsme naprogramovali několik dalších menších pomocných nástrojů v C/C++. Jedná se o:

- `ADdump` – nástroj pro textový výtisk antislovníku v semi-adaptivně komprimovaných `*.dca` souborech (možné plnění vyhledávací třídy `ADsearcher`),
- `patterns-gen` – nástroj pro generování náhodných bitových řetězců podle zadaných parametrů,
- `patterns-cut` – nástroj pro dodávku bitových řetězců, které existují v určitém souboru.

Nástroje `patterns-gen` a `patterns-cut` mají obdobné rozhraní jako `search_cli` z podkapitoly 5.1.6 a dají se ovládat manuálně nebo přesměrováním z shellového skriptu. Všechny podpůrné nástroje v poslední verzi, včetně jejich zdrojových kódů, nalezneme v příloze B (v adresáři implement/search/tools/). Při ladění a namátkové kontrole správnosti vyhledávání jsme využili ještě jednoduchou utilitu `conv` (raw2raw) – konvertor binárního formátu do „bitového“ (textový soubor obsahující ‘0’ a ‘1’), jejímž autorem je Petr Nohavica.

5.2.3 Testovací řetězec

Posloupnost akcí prováděných v rámci našich experimentů lze shrnout takto:
(může být použito jako návod pro přidání nového experimentu)

1. vygenerování antislovníků vstupních textů nástrojem `ADdump`,
2. úprava skriptu generujícího program (nebo přímo jeho) pro nástroje `patterns-*`,
3. vygenerování požadovaných vzorků (soubory `*.pat` v `implement/search/tests/`),
4. definice nových testů v souboru `autotest.defs` – přidání nových dvojic (texts, patts),
5. spuštění testovacího skriptu `autotest.sh`, který generuje program pro `search_cli`,
6. (po dokončení procesu testování) dle potřeby sjednotit/přepočítat výstupní data (v textovém formátu “search(pattern)=answer”) do tabulkového formátu, např. pro import ve zdrojových kódech grafů apod. – skripty `gscatter.sh`, `ggather.sh` (+ jeho konfigurační soubor `ggather.defs`).

Skript `autotest.sh` generuje program pro `search_cli` podle definic testů/experimentů v `autotest.defs` a v cyklu ho spouští s přesměrovaným výstupem (generuje soubory `*.test`). Skript automaticky rozpozná, které definované testy ještě nebyly provedeny, takže neprovádí již hotové testy znovu (dávkové hledání většího množství vzorků je časově náročné).

Testovací řetězec je navržen tak, aby bylo možno přidat nové testy/experimenty s minimálním úsilím až po vygenerování finálních grafů závislostí (např. až po shlédnutí finálních grafů jsme se rozhodli přidat experimenty ještě pro délky vzorků 10 a 18 b). Složitější funkcionality (např. sjednocování dat jinak než podle typu odpovědi nebo průměrného počtu rozhodujících AW) by bylo nutno dopsat do skriptu v řetězci či přidat nový skript/nástroj.

5.2.4 Výsledky experimentů

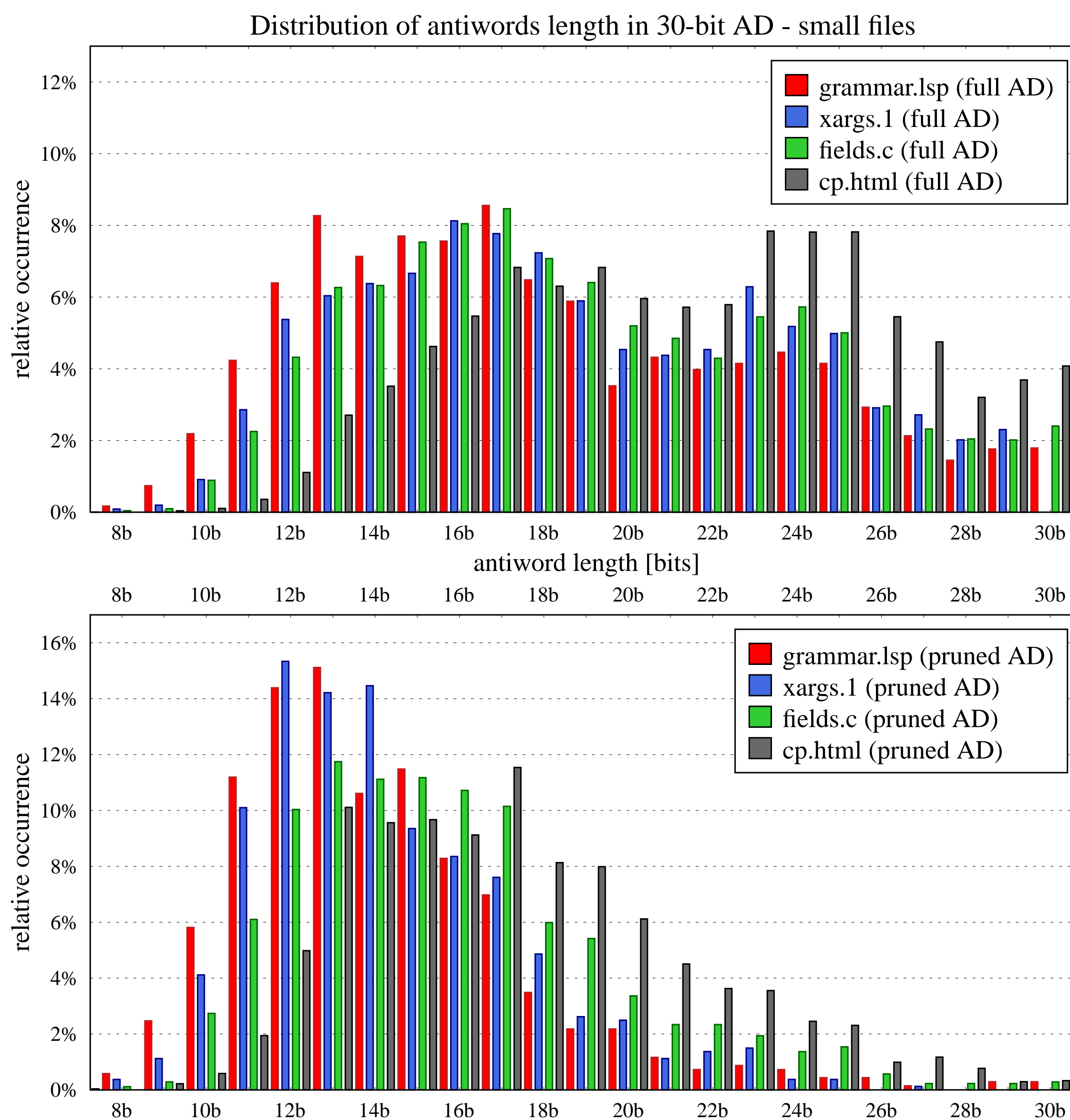
Předně se podívejme do tab. 5.2 na velikosti vygenerovaných antislovníků pro jednotlivé soubory z Canterbury Corpusu. Všechny velikosti se týkají naší výchozí hloubky 30 bitů. Ve sloupci (full) jsou velikosti neprořezaných AD, ve sloupci (pruned) velikosti prořezaných.

soubor	velikost [b]	AD (full)[b]	AD (pru.)[b]	$\frac{\lvert AD(full)\rvert}{velikost}$	$\frac{\lvert AD(pru.)\rvert}{velikost}$	$\frac{\lvert AD(pru.)\rvert}{\lvert AD(full)\rvert}$
alice29.txt	1 216 712	788 156	181 184	0.648	0.149	0.230
asyoulik.txt	1 001 432	790 303	166 394	0.789	0.166	0.211
cp.html	196 824	535 603	52 746	2.721	0.268	0.098
fields.c	89 200	151 608	31 137	1.700	0.349	0.205
grammar.lsp	29 768	70 553	11 141	2.370	0.374	0.158
kennedy.xls	8 237 952	1 411 483	302 367	0.171	0.037	0.214
lcet10.txt	3 414 032	1 389 801	390 459	0.407	0.114	0.281
plrabn12.txt	3 854 888	1 097 917	314 326	0.285	0.082	0.286
ptt5	4 105 728	1 363 504	147 704	0.332	0.036	0.108
sum	305 920	720 552	109 289	2.355	0.357	0.152
xargs.1	33 816	117 863	13 135	3.485	0.388	0.111
$\sum$ [b]	22 486 272	8 437 343	1 719 882	0.375	0.076	0.204
$\sum$ [B]	2 810 784	1 054 668	214 986			
$\sum$ [KiB]	2 745	1 030	209.947			
$\sum$ [MiB]	2.681	1.006	0.205			

Tabulka 5.2: Velikosti antislovníků souborů z Canterbury Corpusu

Velikost neprořezaných AD je v průměru cca 5× větší než prořezaných. U malých souborů je velikost neprořezaných 30-bitových AD (v syrovém stavu – nekomprimovaném pomocí *self compression*) dokonce větší než velikost vlastních dat – zde nemůžeme počítat s $CR < 1$ a tyto situace jsou do experimentů zahrnuty hlavně pro účely srovnání s použitím prořezaných AD. I přesto je vidět jednoznačná korelace velikosti vstupních dat a jejich AD. Zároveň však pozorujeme i vliv typu dat, protože např. Excel Spreadsheet (kennedy.xls) má podobně velký AD jako Technical writing (lcet10.txt), ačkoliv jde o více než 2× větší soubor.

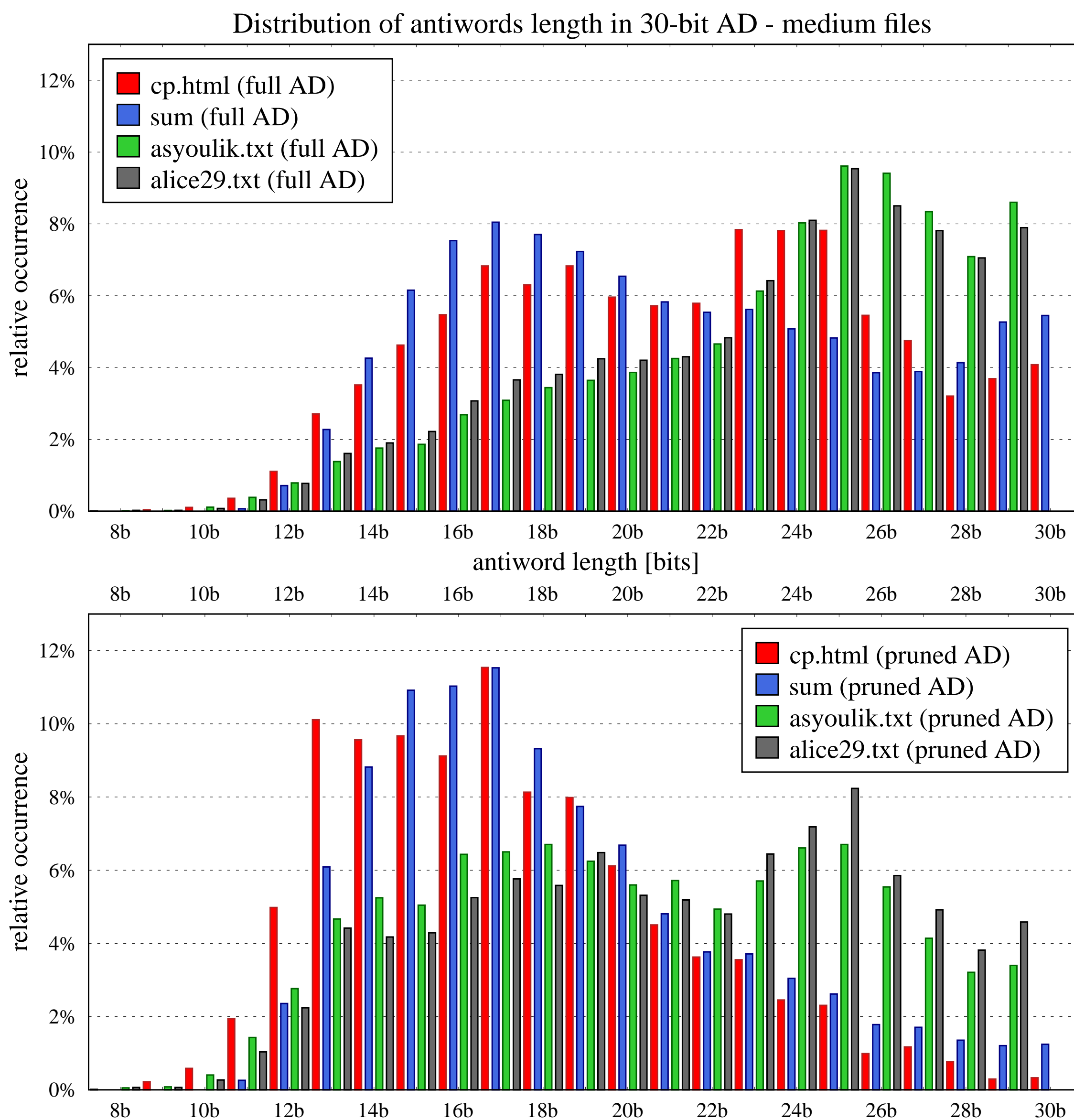
5.2.4.1 Grafy výsledků a jejich interpretace



Obrázek 5.5: Histogram délky antislov - malé soubory

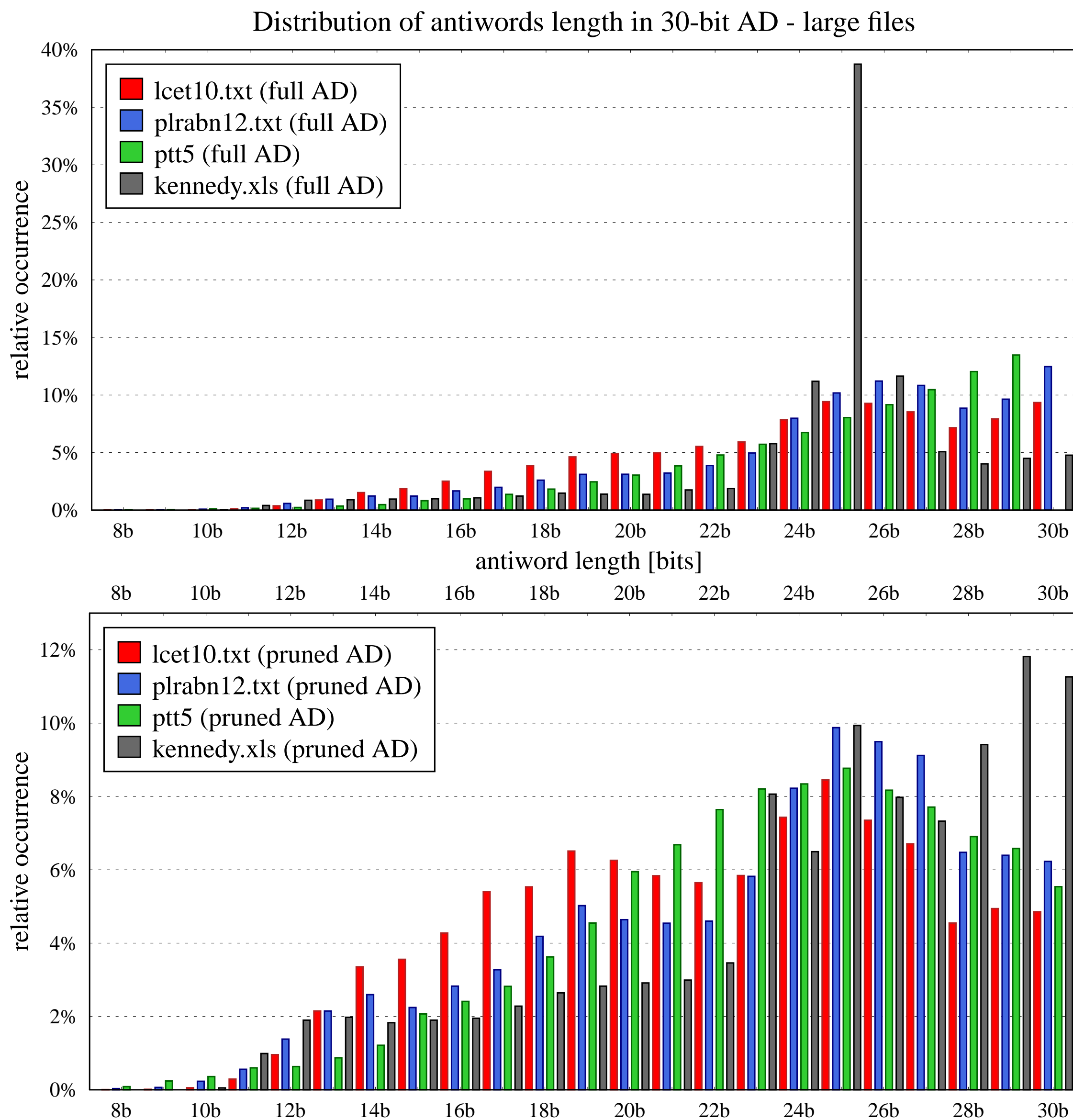
Vizualizace jednotlivých typů experimentů jsme pro dosažení co nejlepší přehlednosti rozdělili do trojic podle velikostí dotčených souborů (podobná velikost přibližuje i výsledky). Jelikož souborů v korpusu je celkem 11 a soubor cp.html je na okraji intervalů velikostí pro malé i střední soubory, jeho data se vyskytují na grafech pro obě tyto velikosti, což může poněkud zlepšovat návaznost čtení při přechodu mezi nimi. Každý obrázek je ještě vertikálně zdvojený pro měření na plném či prořezaném AD, s lícující horizontální osou, aby bylo možno výsledky lehce srovnávat i v této této dimenzi.

První, co nás zajímalo ještě před samotnými experimenty vyjmenovanými v podkapitole 5.2.1 byla distribuce délky antislov v AD.



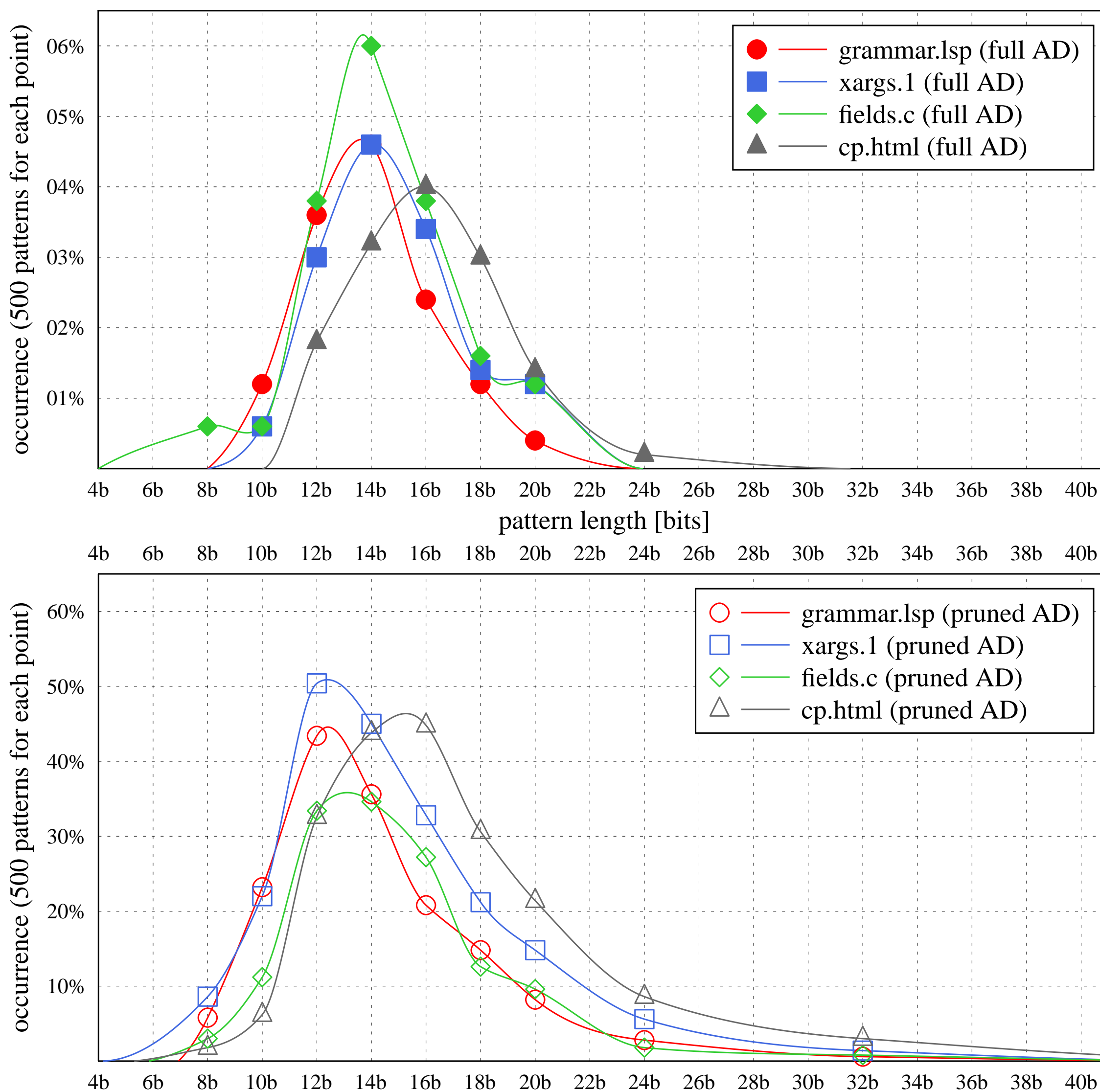
Obrázek 5.6: Histogram délky antislov - střední soubory

Histogramy délky antislov (obr. 5.5, 5.6 a 5.8) ukazují, že malé soubory obsahují obecně kratší antislova. Prořezávání rovněž posouvá průměrnou délku AW k nižším hodnotám – odstraňuje tedy hlavně delší AW. Výskyt jejich komplementů není v textu tak častý jako u kratších, a tudíž mají menší zisk (ačkoliv u stromové reprezentace zabírají v AD stejně velké místo).



Obrázek 5.7: Histogram délky antislov - velké soubory

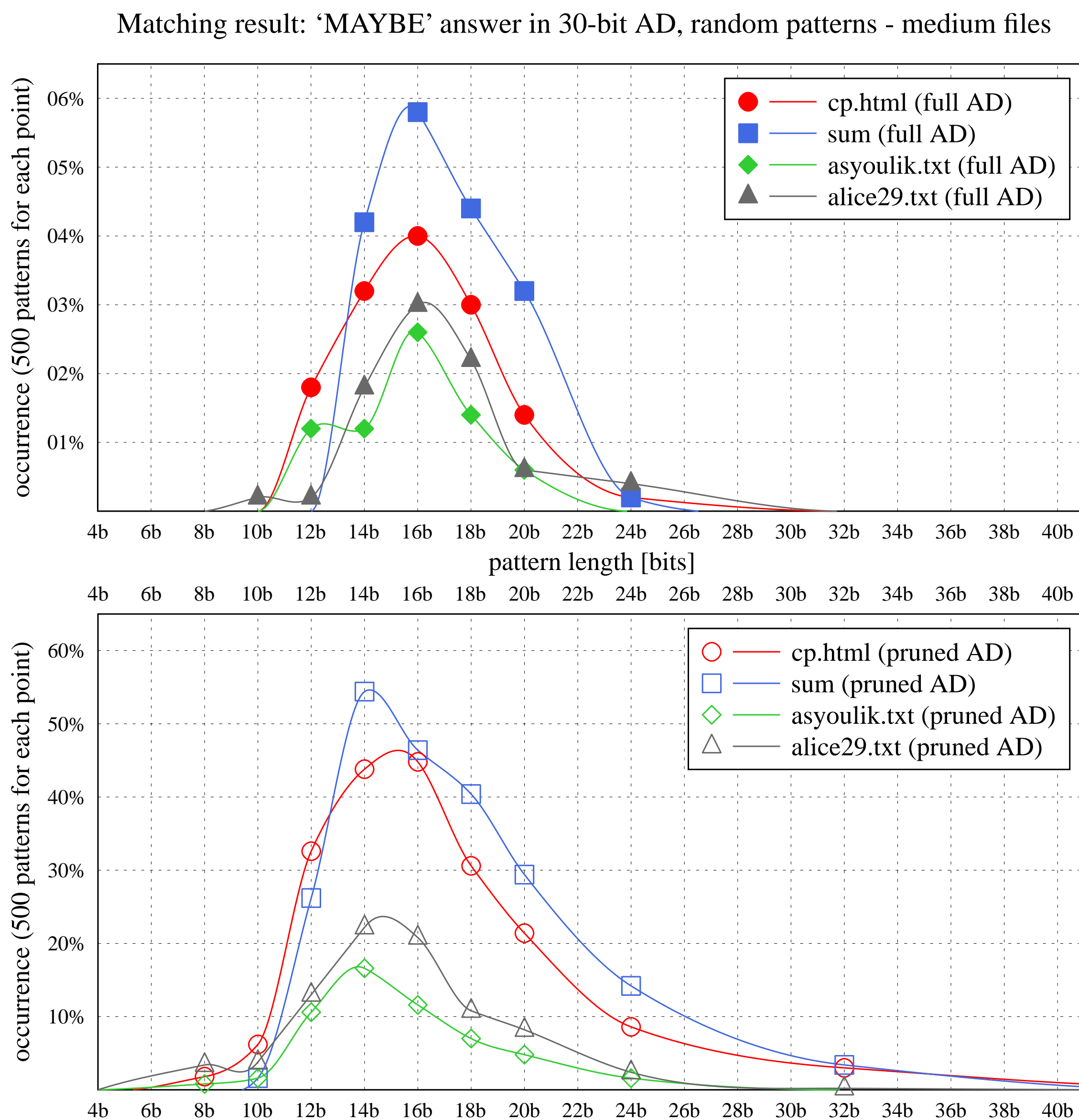
V delších souborech je dost dat a času pro nalezení i delších antislov, a to vesměs nezávisle na typu dat. Výraznou výjimku tvoří podíl 25-bitových AW v souboru kennedy.xls. Domníváme se, že toto souvisí se souborovým formátem XLS, který nejspíš provádí nějaké specifické kódování za pomocí 3 bajtů, což v souboru vytváří vzorky dlouhé 24 bitů – prvních 24 bitů antislova (poté přichází predikovatelný bit). Pohledem do bitové podoby souboru skutečně zjišťujeme pravidelnost s touto periodou. Jak je ale vidět ze spodního podobrázku, většina těchto AW je při prořezávání odstraněna.



Obrázek 5.8: Výskyt neurčité odpovědi na náhodných vzorcích - malé soubory

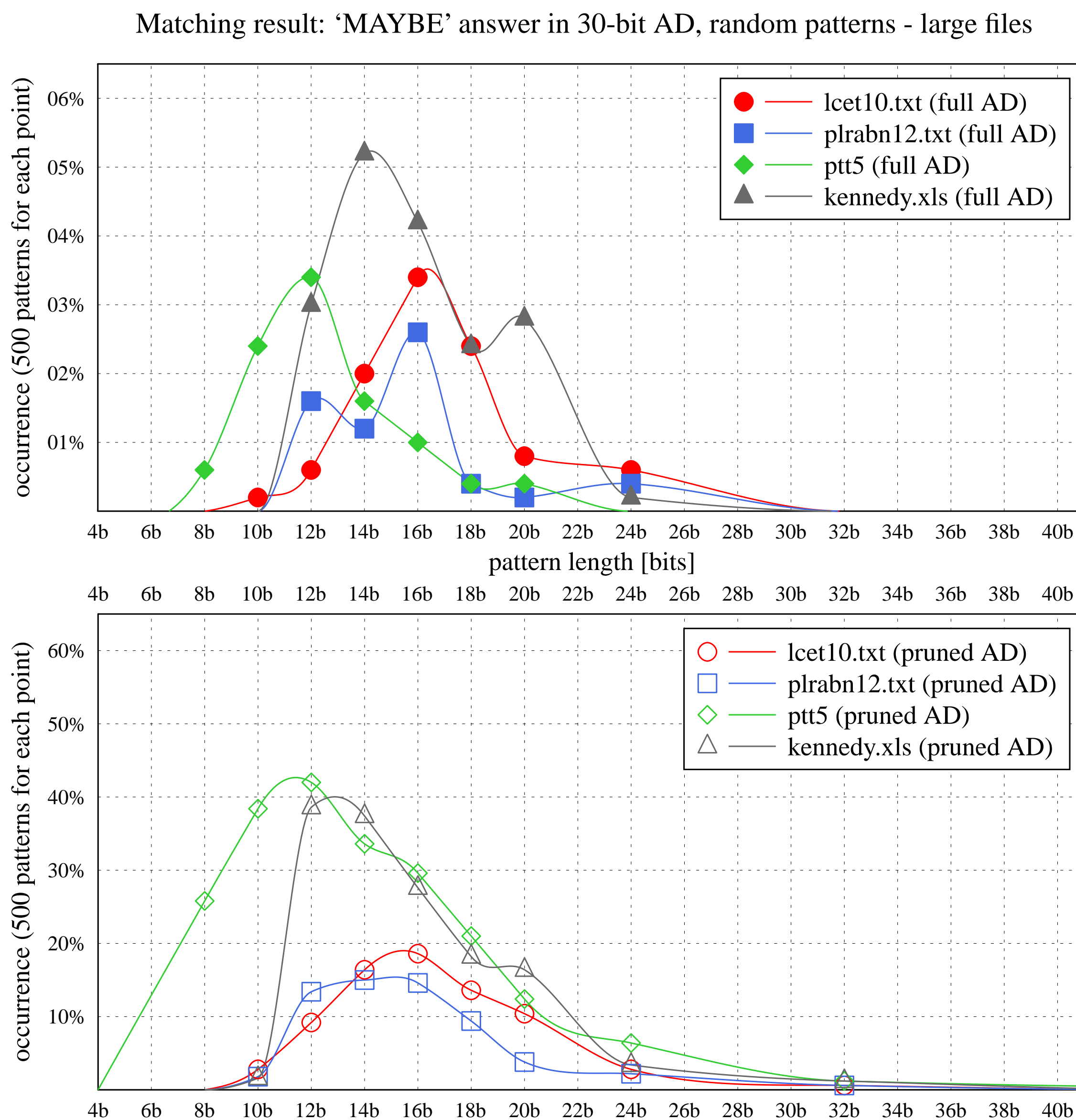
Primární kritérium pro hodnocení použitelnosti našeho vyhledávání je asi podíl výskytu neurčité odpovědi. To nám dává jasnou představu o vyhledávací síle. Podíl výskytu 'MAYBE' odpovědi pro náhodné vzorky naznačují grafy na obr. 5.8, 5.9 a 5.10.

Vidíme, že neurčitá odpověď vykazuje globální maximum, a sice mezi cca 12 až 16 bity délky vzorku, nejčastěji při 14 b. Kratší náhodné vzorky jsou v textu většinou obsaženy (jak uvidíme dále) a jsou zároveň vlastními faktory některého AW. U delších vzorků se naopak zvyšuje pravděpodobnost, že se do AD dostalo antislovo (nebo i více takových AW), které je jeho faktorem. Potom vyhledávání opět „ví“. Největší problém vzniká v této konfiguraci při délce vzorku cca poloviny bitové hloubky AD.



Obrázek 5.9: Výskyt neurčité odpovědi na náhodných vzorcích - střední soubory

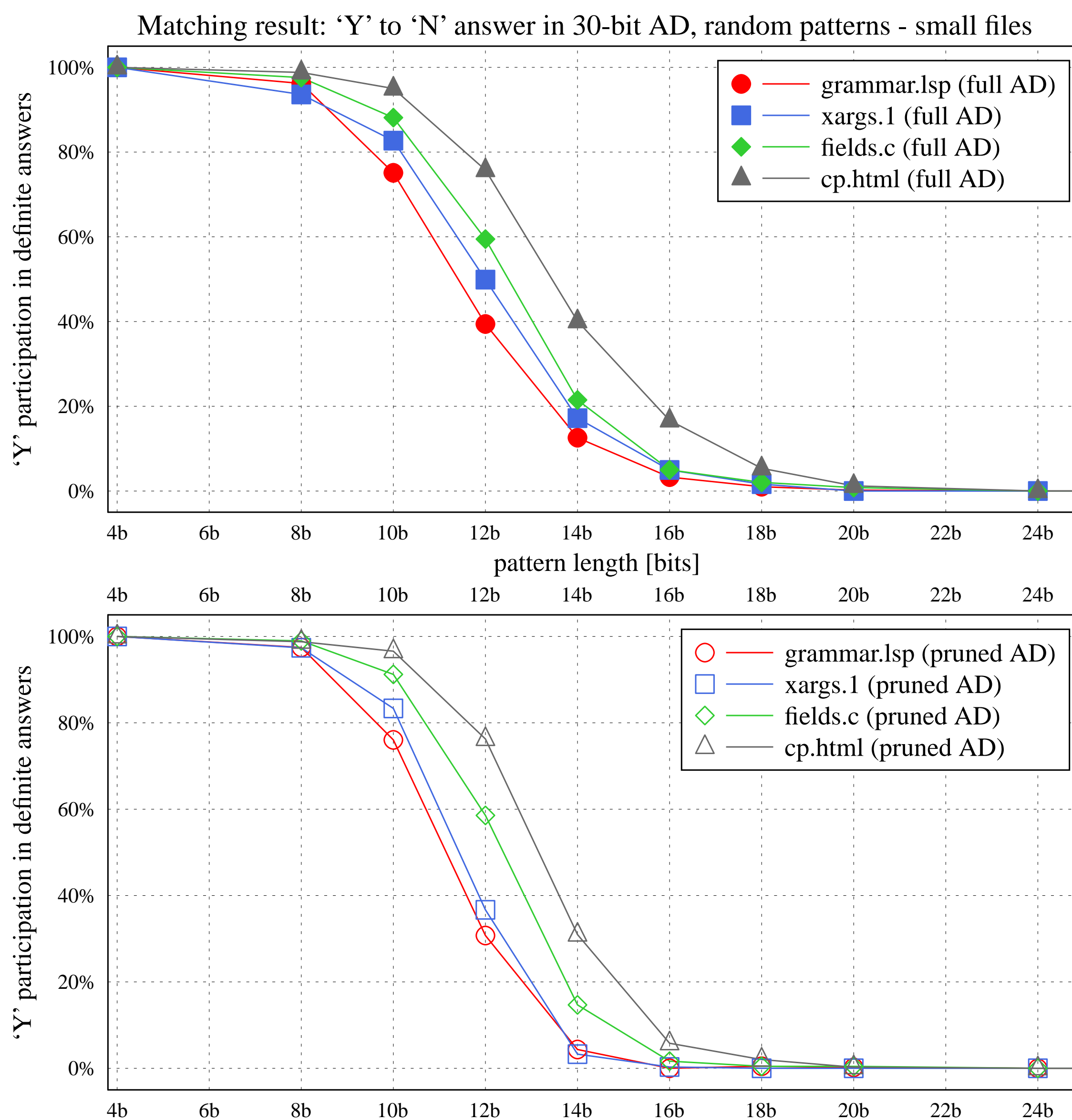
Je pozoruhodné, jak málo je v tomto ohledu vyhledávání citlivé na typ dat i jejich délku (připomeňme rozdíly ve velikostech AD pro jednotlivé soubory v tabulce 5.2). Ve všech připadech jsou výsledky dosti podobné – maxima v okolí 14 b. Velice výrazně se nám však projevuje vliv prořezání AD, které podíl neurčitých odpovědí zvedá v jejich maximech zhruba až o 1 dekadický řád! (a mírně posouvá směrem ke kratším vzorkům). Jednoznačně se tak potvrzuje naše výše vyslovená obava o negativní dopad prořezávání na vyhledávací sílu.



Obrázek 5.10: Výskyt neurčité odpovědi na náhodných vzorcích - velké soubory

Zdá se nám, že výsledky na neprořezaném AD jsou vcelku příznivé, resp. čekali jsme znatelně větší procento neurčitých odpovědí. Zejména u větších souborů, kdy se pracuje v poměru k jejich původní velikosti s menším množstvím dat. Např. u textu plrabn12.txt (báseň v angličtině) je plocha pod modrou křivkou velmi malá, ačkoliv se pracuje jen s 29 % objemu původních dat.

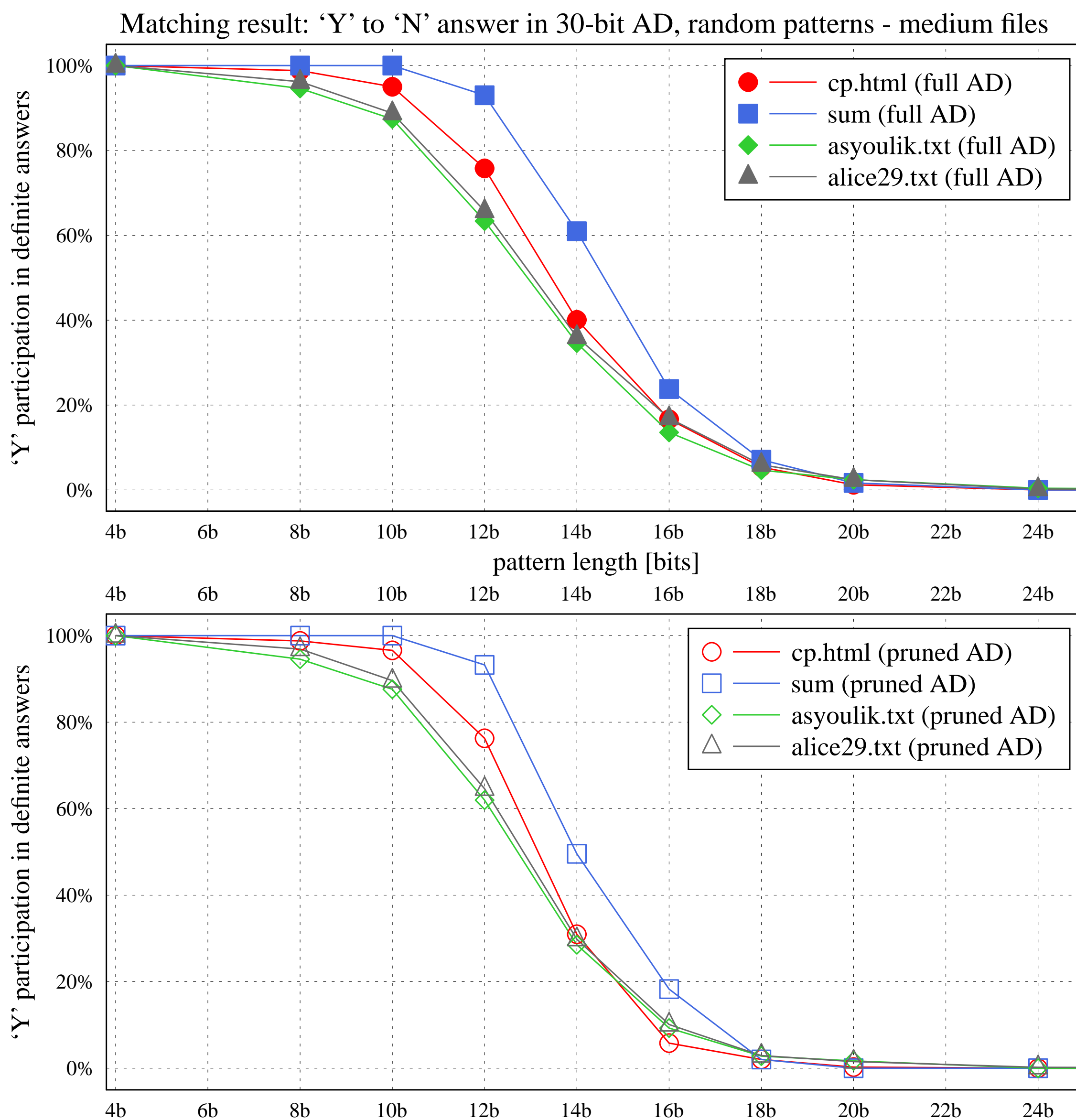
Ani v případě prořezaného AD (v praxi při zpracování přimo `.dca` souboru) není situace tak špatná, když si uvědomíme, že pracujeme jen s 8 % objemu původních dat a křivka nabývá maxima v poměrně úzkém intervalu délky vzorku. Zatím jsme se však zabývali jen vzorky náhodnými, u kterých existuje sama o sobě jistá statistická pravděpodobnost výskytu.



Obrázek 5.11: Výskyt pozitivní odpovědi na náhodných vzorcích - malé soubory

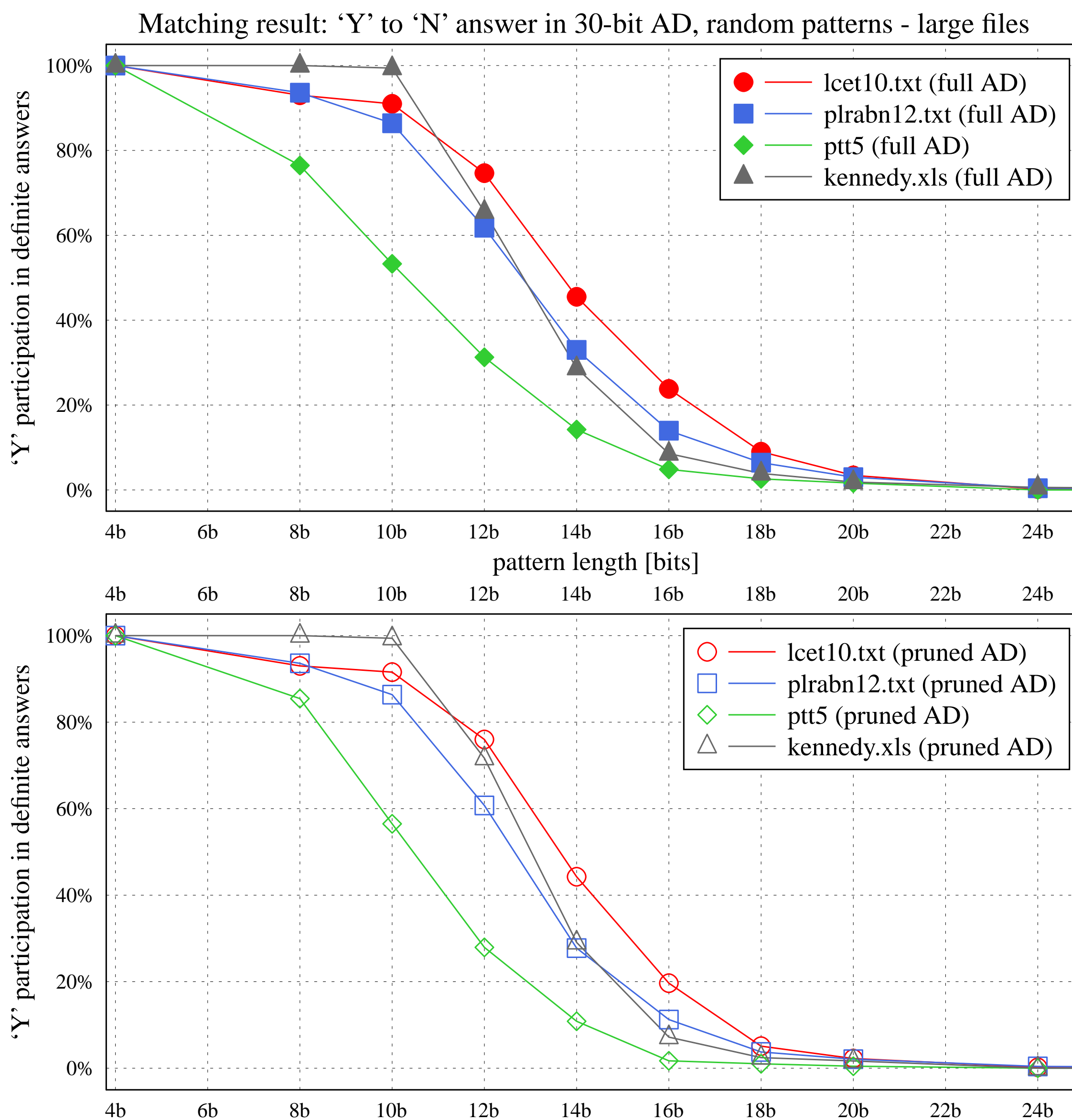
Grafy na obr. 5.11, 5.12 a 5.13 znázorňují podíl výskytu pozitivní odpovědi, a to pouze při odpovědích určitých (‘Y’/‘N’, odpovědi ‘MAYBE’ nejsou započítány). Jde vlastně o křivky distribučních funkcí obsažení faktorů jistých délek v textu (matematicky ještě po odečtení od jedničky).

Dá se předpokládat, že posun propadu od 100 % k 0 % po horizontální ose doprava znamená vyšší entropii vstupních dat a naopak (text s nízkou entropií obsahuje méně náhodných vzorků jisté délky).



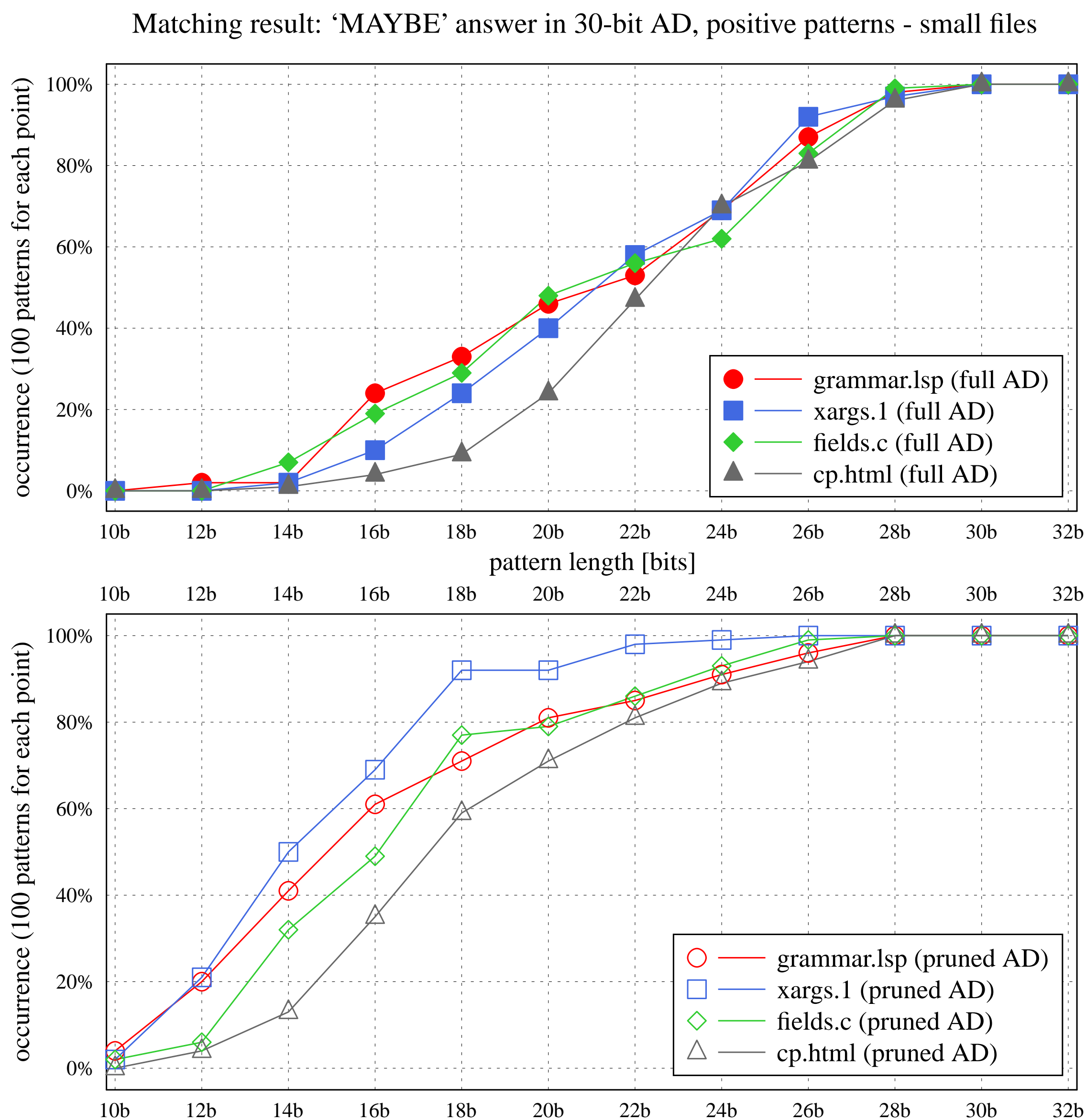
Obrázek 5.12: Výskyt pozitivní odpovědi na náhodných vzorcích - střední soubory

Tento předpoklad potvrzuje posun o cca 2 bity doprava u souborů střední velikosti, ale platí to i na malých souborech mezi sebou (legenda na všech grafech je vždy seřazena podle velikosti souborů). Binární soubory (např. sum) však mají entropii při stejné délce o něco větší než textové.



Obrázek 5.13: Výskyt pozitivní odpovědi na náhodných vzorcích - velké soubory

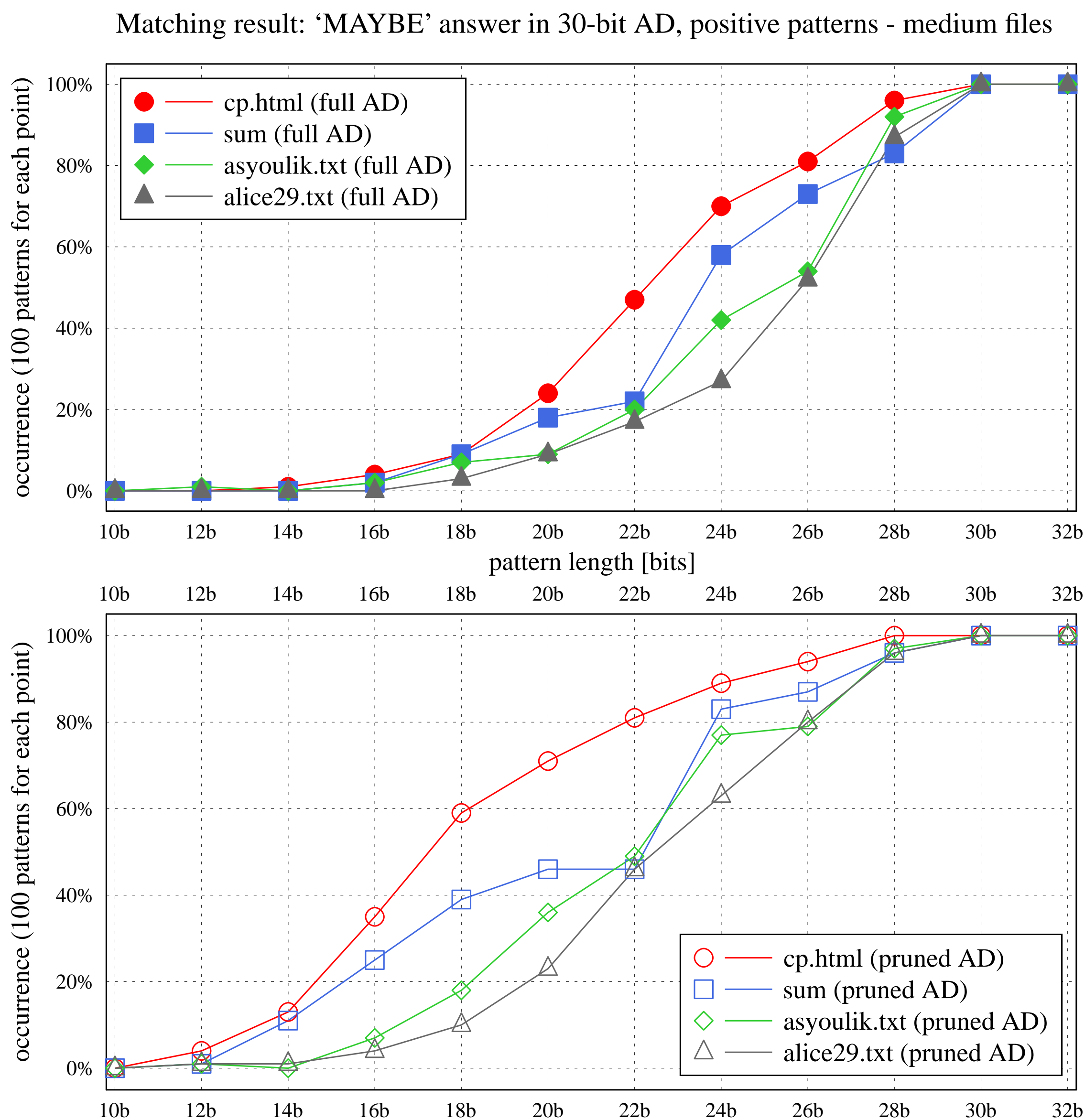
Strmost propadu by měla být vesměs podobná normálnímu rozdělení, pokud formát souboru není příliš speciální, jak je tomu např. u souboru ptt5, který obsahuje mnoho bloků nul. Tento experiment není tak zajímavý z pohledu vyhledávání v DCA metadatech, ale minimálně ověřuje funkčnost vyhledávače.



Obrázek 5.14: Výskyt neurčité odpovědi na vzorcích obsažených v textu - malé soubory

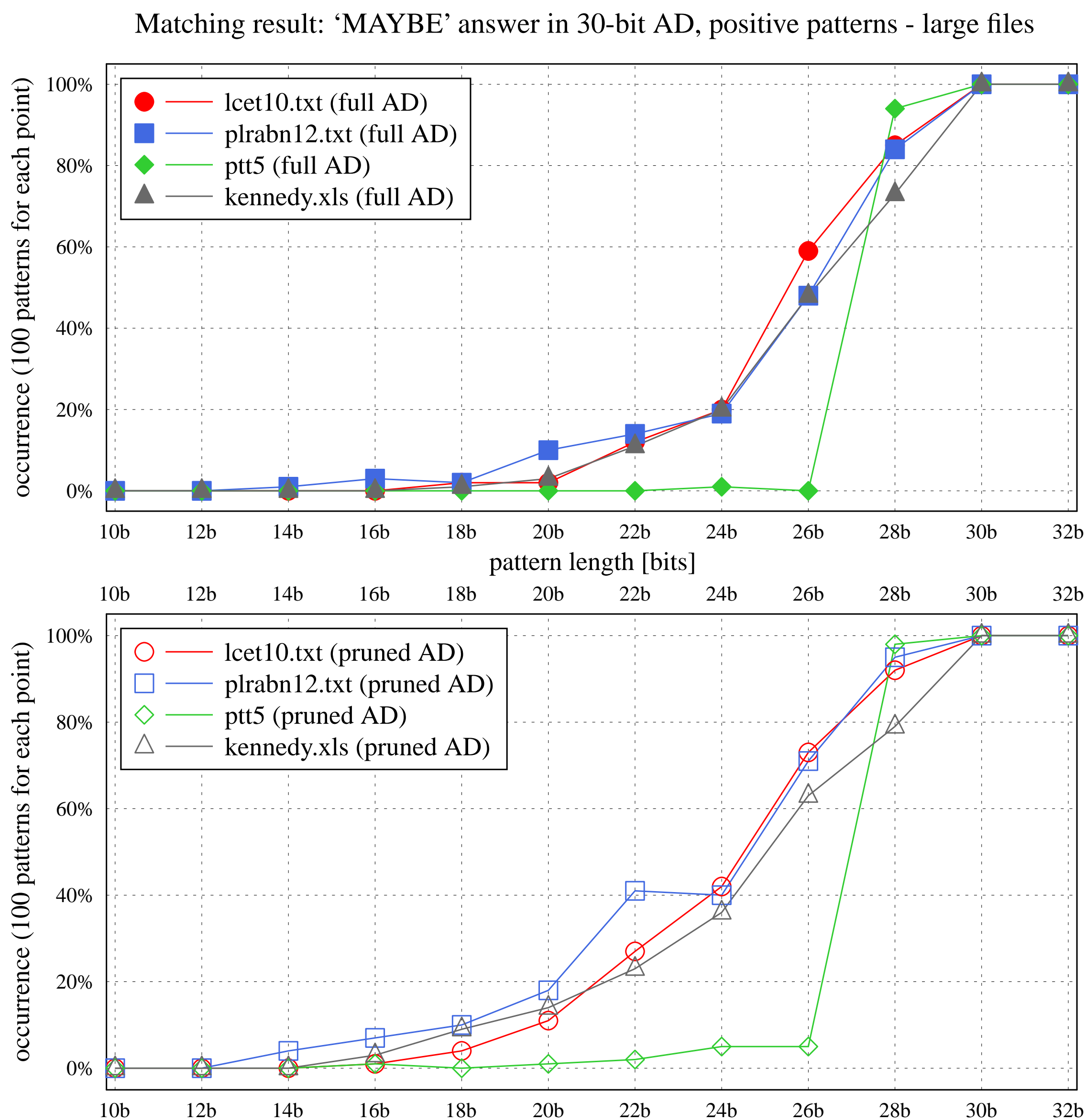
Zajímavější je podíl výskytu 'MAYBE' odpovědi pro vzorky, které jsou v textu obsaženy (narozdíl od náhodných výše). Takovou závislost zachycují grafy. 5.14, 5.15 a 5.16.

Z podstaty pozitivní odpovědi je jasné, že procento jejího výskytu bude se vzrůstající délkou vzorku klesat, protože maximální délku antislov (ve kterých vzorek hledáme jakožto vlastní faktor) máme omezenou hloubkou AD. U vzorků obsažených v textu proto musí adekvátně stoupat procento neurčité odpovědi, protože jinou odpověď než 'Y' či 'MAYBE' zde nemůžeme obdržet. Nad bitovou hloubkou AD (včetně) pak bude existovat už pouze odpověď 'MAYBE', o čemž svědčí konvergence ke 100 % hladině v grafu. Zajímavý je bod začátku a rychlost této konvergence.



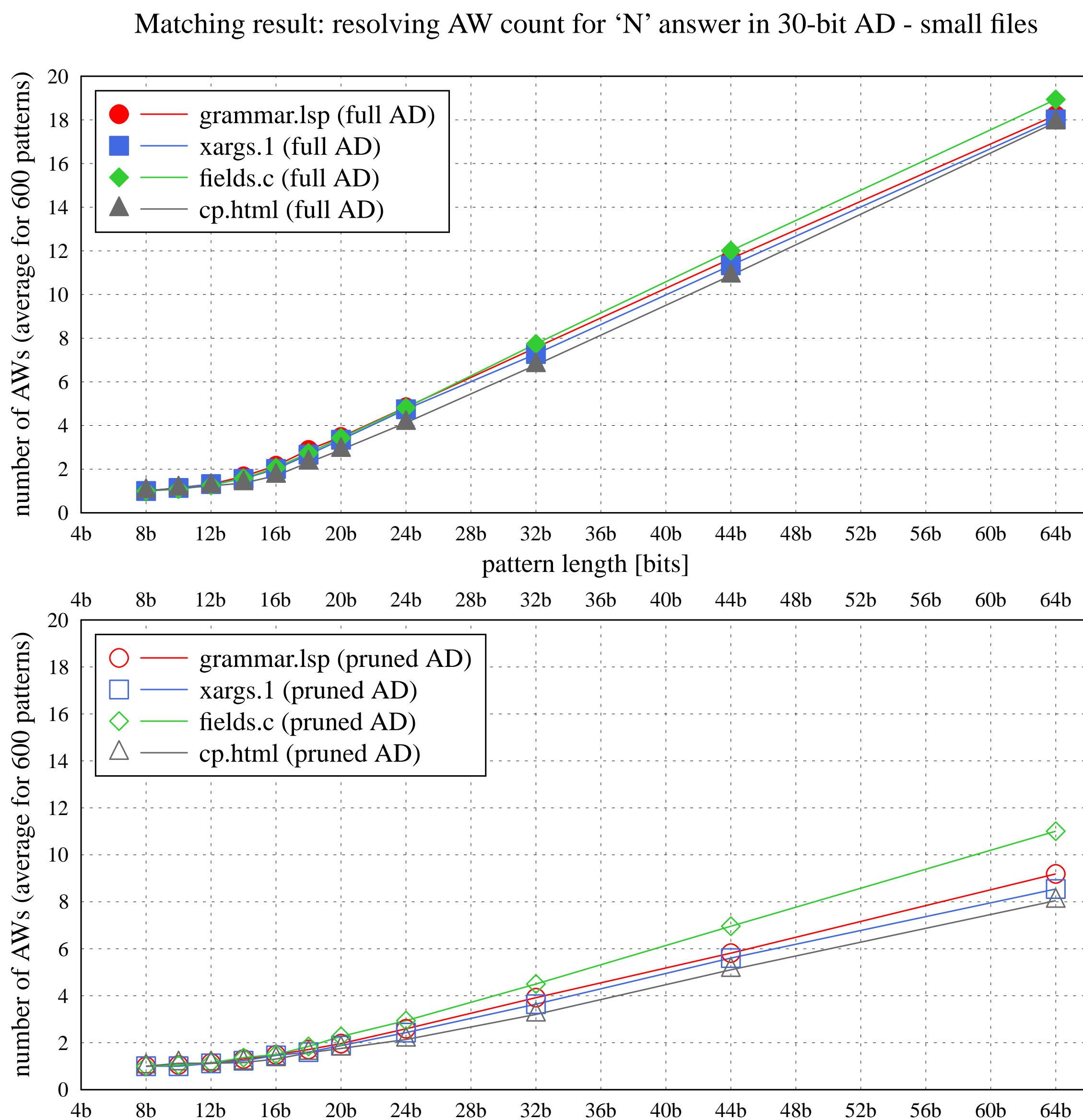
Obrázek 5.15: Výskyt neurčité odpovědi na vzorcích obsažených v textu - střední soubory

Je vidět, že u větších souborů začíná odkládnění k neurčitým odpovědím dále (vpravo po horizontální ose). Použití prořezaného AD nám charakteristiky posouvá naopak vlevo, což je v souladu se ztrátou části informace o původním textu. Zdá se, že soubory s vyšší entropií konvergují dříve a pomaleji (cp.html), což zvětšuje plochu pod křivkou a snižuje sílu vyhledávání.



Obrázek 5.16: Výskyt neurčité odpovědi na vzorcích obsažených v textu - velké soubory

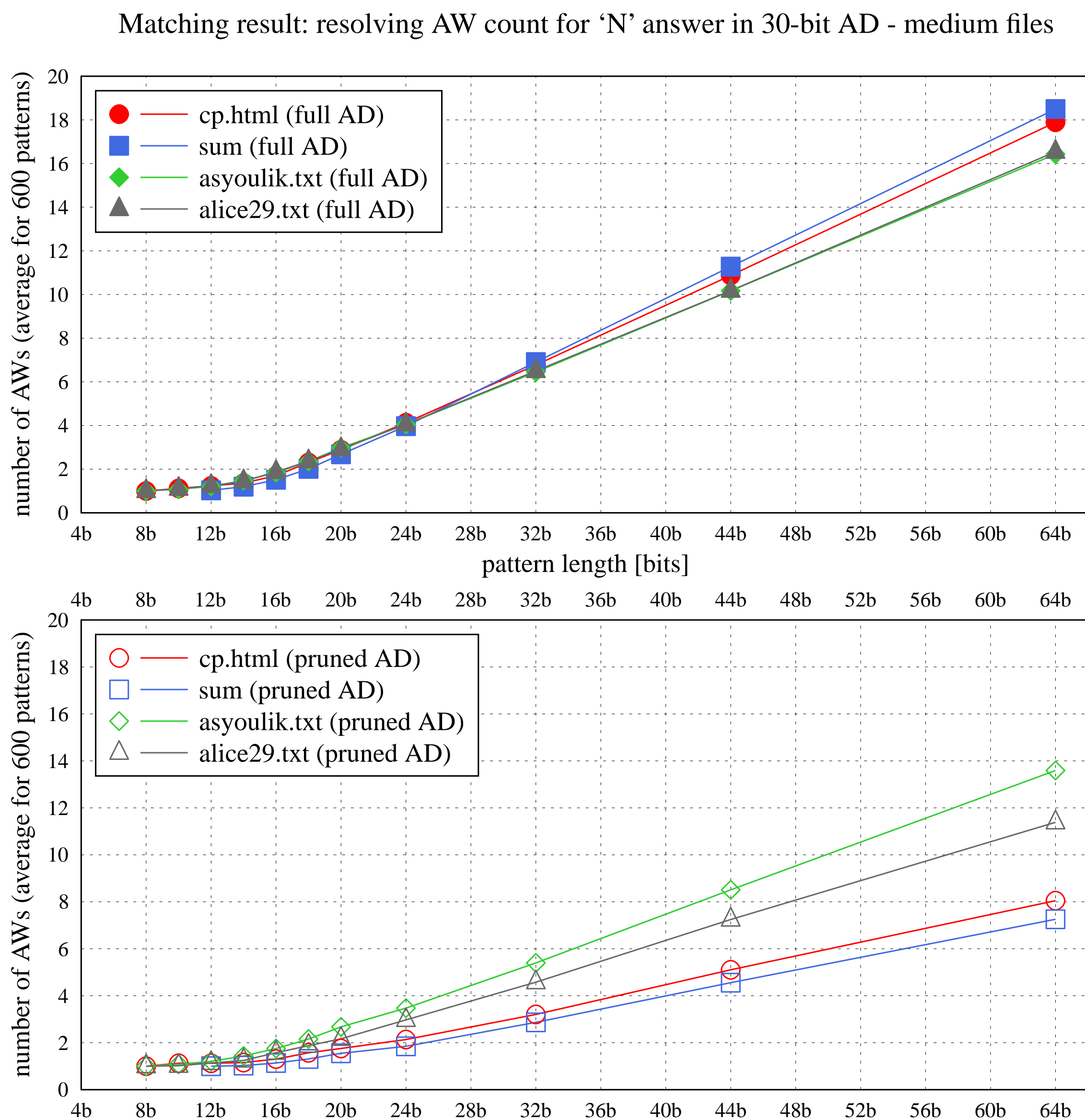
Tomu by odpovídal i vymykající se výsledek pro ptt5 (soubor obsahující mnoho bloků nul), kdy naopak k rapidní změně v typu odpovědi dojde na intervalu pouze 2 bitů. Dále zjišťujeme, že u větších souborů se zmenšuje vliv prořezání AD – zde jsou křivky a plochy pod nimi znatelně podobnější než u souborů malých. To je příznivé zjištění, protože vyhledávání tím pádem neztrácí sílu ani při práci s poměrně menším objemem dat (antislovníky větších souborů jsou menší v poměru k jejich velikosti).



Obrázek 5.17: Počet výsledek rozhodujících AW pro negativní odpověď - malé soubory

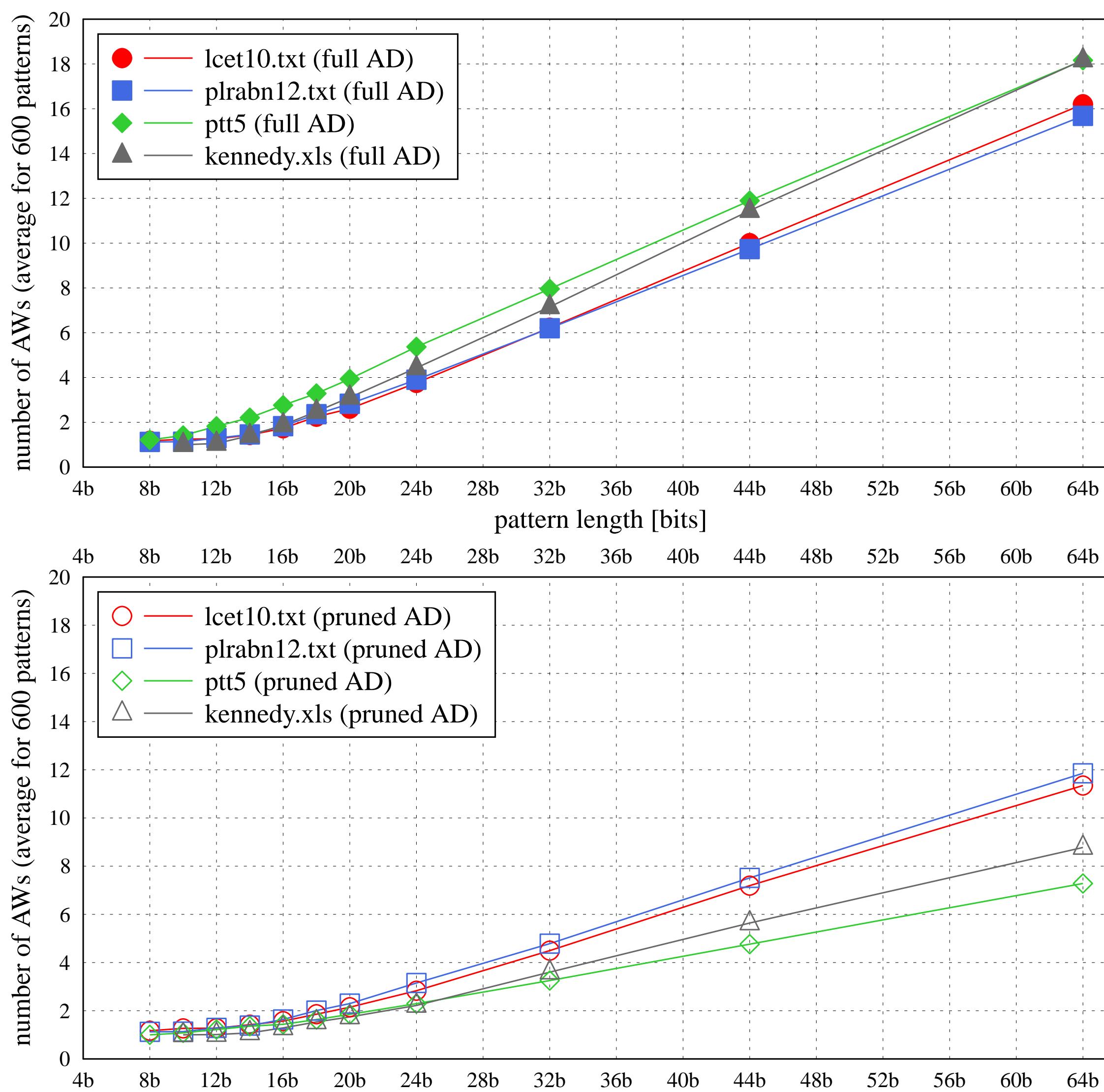
Dále jsme měřili závislost počtu antislov, které rozhodly výslednou odpověď. Její grafy pro rozhodnutí negativní odpovědi máme na obr. 5.17, 5.18 a 5.19. Jedná se o *průměrný* počet AW, která byla pro výsledek rozhodující.

Kromě samého začátku pro velmi krátké vzorky (pro vzorky menší než 8 bitů nemáme v grafu body, protože takové se v textech obvykle všechny vyskytují) jsou závislosti velice dobře lineární. S délkou vzorku totiž počet pozic, na kterých se může některé AW vyskytovat, roste lineárně (kromě začátku, kdy jsou AW delší než vzorky).

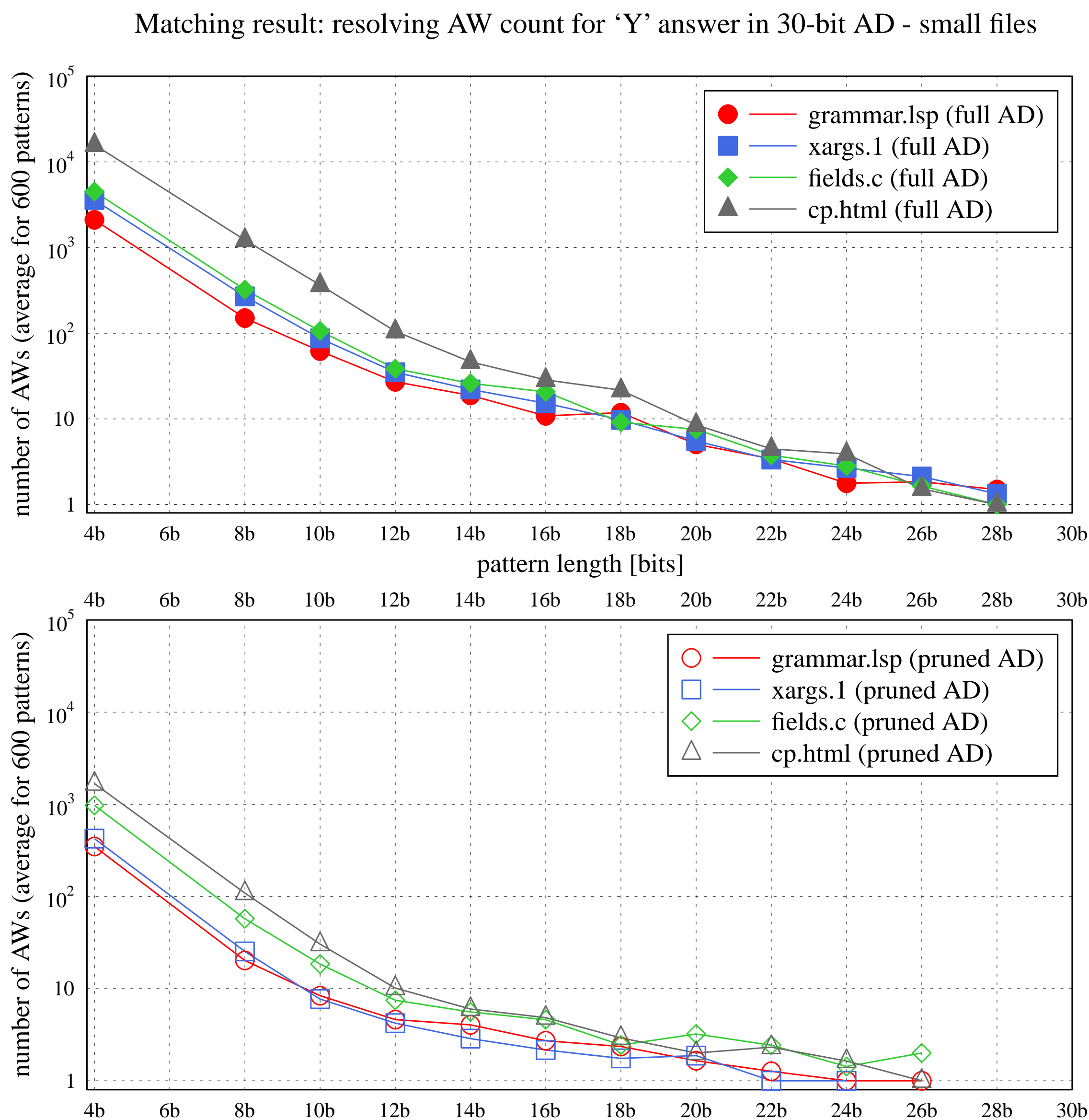


Obrázek 5.18: Počet výsledek rozhodujících AW pro negativní odpověď - střední soubory

Na těchto grafech nejsou žadná významná místa. V případě použití prořezaného AD klesá počet rozhodujících antislov zhruba na polovinu, což je menší pokles, než jsme čekali. Čekali bychom poměr bližší podílu velikostí prořezaného a plného AD (viz tab. 5.2). Dokonce ani rozptyl tohoto podílu nemá velký vliv (soubory xargs.1, cp.html, ptt5). U negativní odpovědi tedy prořezávání sílu vyhledávání příliš nesnižuje.



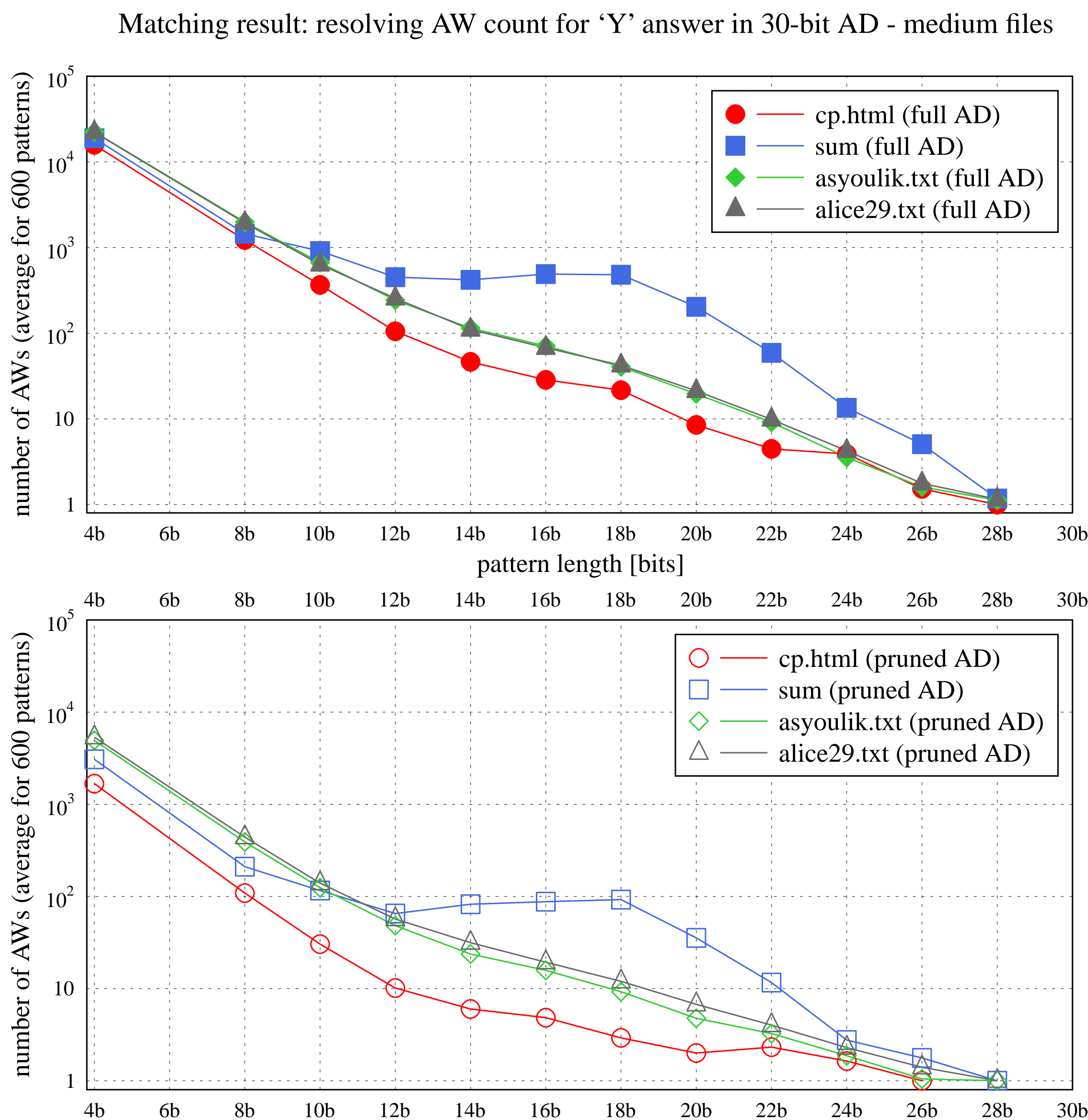
Obrázek 5.19: Počet výsledek rozhodujících AW pro negativní odpověď - velké soubory



Obrázek 5.20: Počet výsledek rozhodujících AW pro pozitivní odpověď - malé soubory

Grafy rozhodnutí pozitivní odpovědi máme na obr. 5.20, 5.21 a 5.22. Na vertikální ose máme logaritmické měřítko, protože dynamický rozsah výsledků je velký kvůli krátkým vzorkům, které jsou vlastními faktory mnoha AW současně. Křivky nesahají dále než ke 30 bitům (kromě), opět kvůli nemožnosti pozitivní odpovědi pro vzorky s délkou větší (nebo rovnou) hloubce AD.

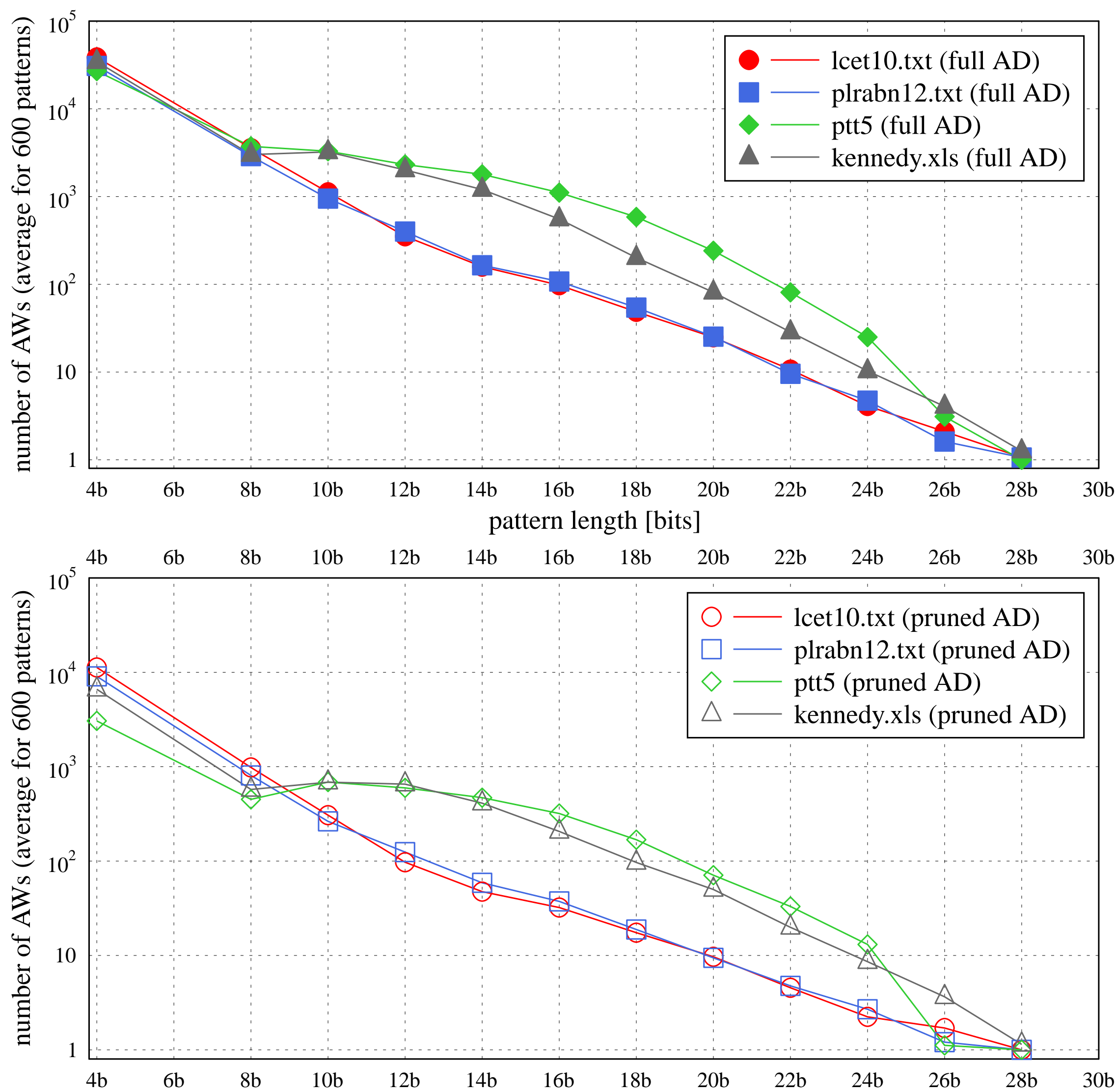
U malých souborů a prořezaném AD křivky končí dokonce ještě o 2 b dříve. Vliv prořezání vidíme na první pohled, kdy počet rozhodujících AW klesne zhruba o 1 dekadický řád. Závislost na velikosti souboru je pozorovatelná – u větších máme víc rozhodujících AW jak při použití AD plného, tak prořezaného.



Obrázek 5.21: Počet výsledek rozhodujících AW pro pozitivní odpověď - střední soubory

U větších souborů se mírně snižuje vliv prořezání a křivky se zplošťují, což znamená pomalejší pokles síly vyhledávání (vyhledávání sice dává určitou odpověd vždy, pokud počet rozhodující AW > 0, ale víc rozhodujících AW znamená větší „rezervu“ při pohybu po dalších dimenzích měření).

Narozdíl od negativní odpovědi zde však prořezávání sílu vyhledávání jenodznačně snižuje. Tento závěr lze potvrdit i z grafů na obr. 5.11, 5.12 a 5.13, kde lze pozorovat systematické snížení podílu kladných odpovědí při použití prořezaného AD. Nejméně se tento vliv projevuje opět u velkých souborů.



Obrázek 5.22: Počet výsledek rozhodujících AW pro pozitivní odpověď - velké soubory

5.3 Zhodnocení

Nejdůležitější informace a poznatky, které jsme učinili při implementaci a testování vyhledávání v DCA metadatech, lze sumarizovat takto:

- Vyhledávání v DCA metadatech je omezeno na bitovou podstatu metody a hledání zarovnaných dat dává redukované výsledky (negativní odpověď ztrácí sílu; pozitivní generuje výsledky typu „false positive“). Typické použití je tedy při hledání samostatných bitových řetězců.
- V implementaci vyhledáváme pomocí simulace NKA s paměťovou složitostí $\mathcal{O}(|AD|)$ a časovou $\mathcal{O}(|P| \times |AD|)$. Faktor $|AD|$[10]nám do výrazu pro časovou složitost přibyl kvůli provádění simulace běhu NKA. Díky tomu však nemusíme provádět determinizaci automatu, která by byla pro větší AD paměťově i časově neúnosná.
- Implementace zná oproti analýze navíc chybovou odpověď 'ERROR', která je výstupem v případě detekce neminimálního antislovníku.
- Integrace vyhledávací funkcionality do ExCom je problematická, protože ExCom implementuje pouze dynamickou verzi DCA, která ke komprimovanému souboru neukládá antislovník. Pro jeho získání je nutno provést „jalovou“ dekompresi, což omezuje využití na experimentální účely.
- Experimenty jsme prováděli s hloubkou AD 30 bitů (hloubka AD posouvá charakteristiky po horizontální ose s délkou vzorku).
- Samotné vyhledávání je citlivé na snížení velikosti AD při jeho prořezání. Má větší vyhledávací sílu při zamítání výskytu vzorku (negativní odpověď) a největší procento neurčitých odpovědí je pro délky vzorku cca poloviční vůči hloubce AD.
- Vyhledávání funguje lépe pro větší soubory (i přesto, že jejich antislovníky jsou menší v poměru k jejich velikosti).

Nárůst síly vyhledávání s velikostí AD, resp. její pokles při jeho prořezání, je ve střetu zájmů s vlastní kompresí, která se snaží do AD dostat jen taková antislova, které může často využít. V tomto ohledu by mohlo být zajímavé použít vyhledávání samostatně – bez komprese, např. pro nějaké indexovací účely. Je však nutné najít aplikaci, která nepracuje s n-ticemi zarovnaných bitů, ale s bitovým proudem.

[10]Mohutnost antislovníku (velikost množiny) je pro nějakou konkrétní bitovou hloubku k asymptoticky totožná s jeho velikostí v bitech $(\sum_{AD} |w \in AD|)$, protože se liší jen o konstantní faktor $\frac{k}{p}$, kde p je poměr průměrné a maximální délky AW.

Kapitola 6

DCA v HW zařízení

V poslední kapitole práce máme za úkol návrh a implementaci kompresní metody DCA v menším HW zařízení. Provedeme výběr vhodného zařízení, seznámíme se s jeho platformou a zajistíme možnost jeho poskytnutí pro účely této ukázky. Navrhneme kompresní schéma, které budeme implementovat, a vyzkoušíme možnosti jeho nasazení v praxi.

6.1 HW platforma

6.1.1 Výběr zařízení

Menším zařízením v úvodu máme na mysli hardware disponující omezeným výpočetním výkonem nebo malou operační pamětí. V úvahu o výběru takového zařízení přichází hardware s větším hradlovým polem (např. Xilinx Spartan), embedded aplikace s jednočipovým mikrokontrolérem (např. Atmel AVR) nebo menším RISC MCU (např. s jádrem ARM7–9) apod. Požadavkem je, aby systém generoval nějaké množství dat za účelem jejich přenosu či archivace. Zpracovávaná data by navíc měla být dobře komprimovatelná, aby se vyplatilo zkoušet nasazovat nějakou variantu metody DCA pro ušetření pásma přenosového kanálu nebo místa v datovém úložišti.

Zařízení, které splňuje uvedené požadavky, jsme našli u firmy Princip, a. s.[1] Jedná se o mobilní jednotku **MJ2732VEP – Vetronics**. Základní funkcionalitou tohoto zařízení je elektronická kniha jízd (on-line sledování vozidel). Jde tedy o telemetrickou aplikaci, u které se sbírá informace o poloze a stavu vozu. Provozovatelem monitorovací služby je firma HI Software Development, s. r. o.[2] Zákaznické specifikace pro nás nejsou důležité, proto uvedeme pro zajímavost jen to, že zařízení je napájeno z palubního napětí automobilu a je vybaveno rozhraními GPS, GSM/GPRS, RS232, CAN/FMS, RFID. Jeho celou technickou specifikaci nalezneme v příloze E a námi pořízené ilustrativní fotografie samotné jednotky v příloze D. Výrobce byl ochoten jeden kus zapůjčit pro účely naší práce.

Konkrétní zákaznická konfigurace určuje, jaké procesy v zařízení běží a jaká data sbírají. Výstupem jsou logy, které se ukádají do kruhového bufferu, a jednou za čas se po GPRS uploadují na server provozovatele služby. Logy mohou být vyčteny i z file systemu v offline režimu přes komunikační kabel.

[1] http://www.princip.cz
[2] http://www.webdispecink.cz

6.1.2 Popis zařízení

Jádro hardwaru zařízení tvoří RISCový mikrokontrolér STR911FAM44 z dílny firmy STMicroelectronics[3]. Jedná se o 16/32-bit 96 MHz MCU na bázi ARM9 s flash pamětí, DMA, IRQ, množstvím integrovaných řadičů pro externí periferie (a JTAG pro servisní účely) v pouzdru LQFP80/128 nebo LBGA144. V zapojení Vetronicsu je u něho přes SPI připojená další NOR flash paměť s kapacitou 32 Mbit. Použitý operační systém je proprietární s neoficiálním názvem „Sedlix“ (jeho autorem je Ing. Seidl, pracovník Principu). OS momentálně umožňuje multitasking až 32 procesů, a ačkoliv není prohlášen za real-timový, dle vyjádření ostatních zaměstnanců by měl být velmi optimalizovaný.

Velikost paměti SRAM tohoto procesoru je 64/96 KB (Vetronics používá verzi s 96 KB) a po odečtení jejího zaplnění kritickými procesy (nutnými pro běh aplikace) zbývá zhruba 30 KB. Toto je hlavní překážka pro portaci nějaké běžné kompresní metody. Pro rychlejší upgrade firmwaru byl v minulosti portován Gzip (GNU zip), ovšem pouze jeho dekompresní část. Bylo nám řečeno, že kompresi se kvůli vyšším požadavkům na paměť (a nutnosti nahlížení dozadu) nepodařilo převzít a zde je tedy prostor pro naše pokusy s DCA. Data k potenciální kompresi jsou soubory s logy z jednotky, jejichž kódování připomíná formát `.hex`, takže by měla být i dobře komprimovatelná.

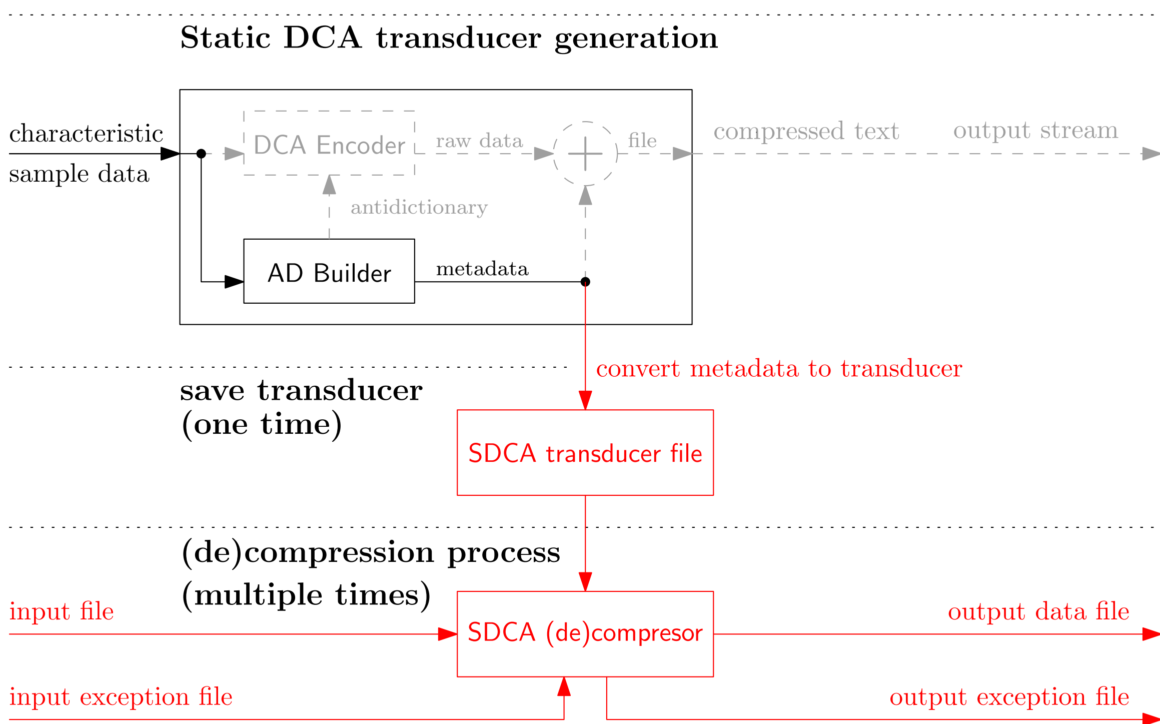
Vývojová platforma systému je postavena na jazyku C a je částečně kompatibilní se standardem ANSI. To jsme uvítali v domnění, že implementaci založíme na portaci existujícího DCA z C++ do C. Existující implementaci jsme nakonec použili jen částečně (jak uvidíme dále), ale kompatibilitu s jazykem C jsme využili k odladění většiny kódu na PC a výsledný kompresní program se dá podmíněně přeložit buď pro PC s Linux/Windows nebo pro Vetronics a jeho OS.

6.2 Kompresní schéma

Hlavní problém metody DCA spočívá v konzumaci paměti při tvorbě antislovníku. Existující implementace dynamicky alokuje pro velké soubory i stovky MB (pro hloubku AD 30 bitů). Je zřejmé, že konstrukci AD nebudeme moci v malém HW provádět. Toto předurčuje použití komprese se statickým schématem (def. 2.42). Myšlenka je tedy „outsourcovat“ konstrukci AD na výkonnou výpočetní platformu (PC) a v malém HW provádět jen vlastní kompresi/dekompresi dat DCA překladovým automatem (viz podkapitoly 3.4.1, 4.2).

Je jasné, že kompresi je nutné „natrénovat“ (vytvořit AD, potažmo DCA automat) na datech, která se budou později komprimovat, a že pro jiný charakter dat nebude použitelná. Vstupní logy jsou však dosti podobné – obsahují stejnou množinu použitých znaků, mají podobně dlouhé řádky atd., takže si myslíme, že tento přístup je zde možný. Samozřejmě je nutné počítat s tím, že natrénovaný AD bude obsahovat i antislova, které se v dalších datech ke kompresi budou vyskytovat. Taková místa (ve kterých DCA automat nebude mít definovaný přechod) musíme v komprimovaném souboru kódovat jako *výjimky* (viz dále). Podobnost trénovacích dat a dalších dat ke kompresi by měla počet takových výjimek, které budou zabírat další místo (a zhoršovat tak CR), minimalizovat.

[3] http://www.st.com



Obrázek 6.1: Schéma kompresního procesu implementovaného statického DCA

Schéma přístupu, který jsme se rozhodli implementovat, vidíme na obr. 6.1. Z existující implementace DCA přebíráme konstrukci antislovníku (resp. kódovacího automatu) a realizujeme 2 vlastní prvky – převod metadat do paměťově úsporného formátu DCA automatu a statickou kompresi/dekompresi.

Bylo by ideální, kdyby se definice DCA automatu (SDCA transducer file; S od static) celá vešla do zbytku RAM a nemuseli bychom části automatu načítat dynamicky z FS na flash (jednotka má vlastní plochý FS podobný FAT). To by velmi drasticky zhoršilo hlavní výhodu DCA, kterou je rychlost zpracování (provádí se jen výpočet PKA). Přístup k flash paměti je totiž řádově pomalejší než do RAM a počet výměn částí automatu by byl velký, protože nevykazuje žádnou prostorovou lokalitu.

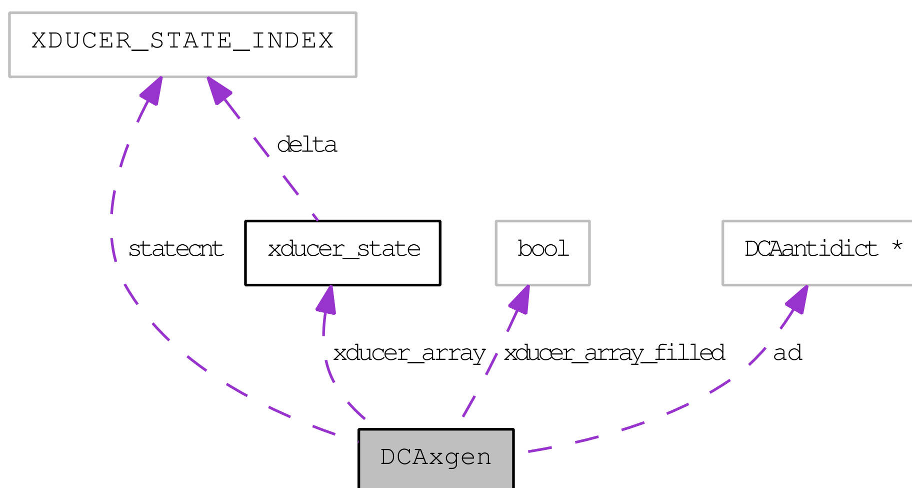
6.3 Implementace

Programovací a vývojové prostředí, které jsme používali, je podobné jako při realizaci vyhledávání v DCA metadatech. Navíc jsme měli k dispozici kompilátor pro Vetronics (modifikované gcc) a speciální kabel pro upload firmwaru. Na PC jsme se zařízením komunikovali přes sériový port.

Byly implementovány 2 programy – generátor překladového automatu pro kompresi i dekompresi (`dcaxgen`) a jeho interpret (`sdca`). Generátor je primárně určen pro platfromu PC, (de)kompresor pro platformu Vetronics (ale dá se podmíněně zkompilovat a jádro komprese odladit i na PC).

6.3.1 Program dcaxgen

Program `dcaxgen` je nástroj napsaný v C++, který využívá existující implementaci DCA, a jeho úkolem je ze vstupního souboru vygenerovat soubor DCA překladového automatu v námi stanoveném formátu. Vstupním souborem jsou charakteristická data podobná těm, které se mají v budoucnu komprimovat. Zdrojový kód modulu obsahuje zejména třídu `DCAxgen`, jejíž kolaborační diagram máme na obr. 6.2.



Obrázek 6.2: Kolaborační diagram třídy DCAxgen

Formát automatu byl zvolen s ohledem na co nejmenší velikost a obsahuje hlavičku se signaturou souboru (`SDCX`), počet stavů automatu a jeho přechodovou tabulku v 16-bitovém fixním kódování. Stav s indexem 0 značí nedefinovaný přechod a jeho buňka je v souboru rezervována pro globální informace. Datový typ `short` pro indexaci jsme zvolili proto, že je to nejbližší násobek velikosti bajtu vyšší, než binární logaritmus maximálního počtu stavů, se kterými vůbec můžeme pracovat (kvůli omezení paměti).

Během generování automatu je implicitně zapnuté prořezávání antislovníku, které lze vypnout argumentem na příkazové řádce. V tomto případě (narozdíl od vyhledávání) je prořezávání nanejvýš žádoucí, protože odstraní antislova, která se na charakteristická data dají použít jen málokdy, a na data komprimovaná by se nemusela dát použít vůbec. Počet stavů se u menších automatů (20–28-bit AD) tímto redukuje sice jen o 10–30 %, ale i to vzhledem k omezené paměti vítáme. Podrobnější hodnoty počtu stavů automatu na testovacím kompilátu logů (soubor testdata1.log) jsou v tab. 6.1.

Při snaze o minimalizaci velikosti automatu nás napadlo několik dalších možností, jak jeho reprezentaci zmenšit. Bylo by asi možné samotný automat komprimovat, kódovat diferenci přechodové funkce (místo její přímé hodnoty, protože mnoho hran je krátkých) za použití unionů, nebo se pokusit o kompaktizaci grafu přechodů pomocí genetického algoritmu. Ani jedna z těchto technik by však podle nás nebyla přínosem vzhledem k poměru ušetřeného místa a větší složitosti zpracování. Proto jsme zůstali u blokové reprezentace a volby vhodné bitové hloubky při konstrukci AD, což se ukázalo jako dostatečně jednoduché pro menší HW.

hloubka AD	stavy (full)	stavy (pruned)	ušetřeno %
30	25161	5317	78.9
29	13506	3151	76.7
28	3975	1833	53.9
25	1134	1055	7.0
20	534	483	10.0
15	278	235	15.5
10	72	52	27.8
5	1	1	0

Tabulka 6.1: Počet stavů DCA automatu při prořezání AD (testdata1.log)

6.3.2 Program sdca

Program `sdca` je samostatná aplikace napsaná v C realizující vlastní statickou kompresi/dekompresi pomocí dodaného DCA automatu. Vstupní soubor čte jako bitový proud v režimu MSb first (stejně jako implementace [23]) a používá čtecí i zapisovací buffery s velikostí nastavitelnou makrem. Na začátek výstupního souboru se ukládá velikost původního textu, aby bylo možné proces dekomprese ukončit ve správný okamžik.

V kódu jsou ošetřeny všechny chybové stavy, které během komprese mohou nastat – selhání souborových operací, nedostatek paměti pro alokaci automatu, špatně definovaný automat, přetečení při kódování výjimek atd. Použití programu máme v příloze C. V případě absence názvů výstupních souborů program automaticky[4]použije názvy podle vstupu s automatickou změnou přípon (`.sdca` pro datový proud, `.sdce` pro soubor s výjimkami).

6.3.2.1 Výjimky

Jelikož pracujeme se statickým schématem DCA, je nutné řešit výjimky ve vstupním proudu. Jedná se o situace, kdy v metadatech (zde v DCA automatu) máme definováno antislovo, které se v komprimovaném textu vyskytuje. Tuto situaci rozpoznáme tak, že překladový automat se při kompresi dostane do stavu, ze kterého nevede přechod pro symbol (bit) následující ve vstupu. Při dekompresi načítáme vzdálenost následující výjimky ze souboru výjimek.

Při detekci výjimky můžeme pokračovat buď přechodem automatu přes komplementární hranu nebo jeho restartováním do stavu počátečního. Je jasné, že stejným způsobem se musí chovat kódovací i dekódovací procedura, aby bylo zajištěno, že v každém místě dat bude pozice v automatu synchronizována pro kompresi i dekompresi[5]. Varianta s komplementární hranou má výhodu v tom, že v případě špatných/nedostupných informací o výjimkách se zachovává synchronizace komprese/dekomprese a dekomprimovaný soubor by se od původního lišil jen ve výjimkových bitech. Varianta s restartem je v tomto případě desynchronizující (dekompresní proces se nedozví o tom, že má provést restart, přejde přes dekomprimující

[4]tato funkcionalita je při kompilaci pro Vetronics zakázána pro ušetření paměti

i	$\alpha(i)$	$\beta(i)$	$\beta'(i)$	$\gamma(i)$	$\gamma'(i)$	$\delta(i)$
0	–	(0)	–	–	–	–
1	1	1	ε	1	1	1
2	01	10	0	001	010	0010
3	001	11	1	011	011	0011
4	0001	100	00	00001	00100	01100
5	00001	101	01	00011	00101	01101
6	$0^5 1$	110	10	01001	00110	01110
7	$0^6 1$	111	11	01011	00111	01111
8	$0^7 1$	1000	000	0000001	0001000	00001000
9	$0^8 1$	1001	001	0000011	0001001	00001001
10	$0^9 1$	1010	010	0001001	0001010	00001010
11	$0^{10} 1$	1011	011	0001011	0001011	00001011
12	$0^{11} 1$	1100	100	0100001	0001100	00001100
13	$0^{12} 1$	1101	101	0100011	0001101	00001101
14	$0^{13} 1$	1110	110	0101001	0001110	00001110
15	$0^{14} 1$	1111	111	0101011	0001111	00001111
16	$0^{15} 1$	10000	0000	000000001	000010000	000110000
17	$0^{16} 1$	10001	0001	000000011	000010001	000110001
18	$0^{17} 1$	10010	0010	000001001	000010010	000110010
19	$0^{18} 1$	10011	0011	000001011	000010011	000110011
20	$0^{19} 1$	10100	0100	000100001	000010100	000110100
$\sum$ [b]	210	74	54	128	128	138

Tabulka 6.2: Kódy s pohyblivou délkou – základní

hranu a pokračuje v automatu dále) a dekomprimovaný soubor by oproti původnímu vykazoval bitové posuny v místech výjimek. Uvedené se ale týká pouze situace poškozených dat výjimek. Při experimentech s počtem a vzdálenostmi výjimek jsme zjistili, že z hlediska komprese se lépe chová varianta s restartem automatu. Při přechodech přes komplementární hrany se objevovalo množství bezprostředně následujících výjimek, které automat dostaly blízko počátečnímu stavu. Restartující variantu jsme tedy zvolili za výchozí a v případě potřeby/experimentů se dá toto chování přepnout makrem `SDCA_EXCEPT_RESTART`.

6.3.2.2 Kódování výjimek

Vlastní kódování výjimek funguje tak, že se ukládá počet vynechaných bitů (hodnota *omitted* v alg. 3.1) mezi dvěma následujícími výjimkami. V tomto ohledu jsme se inspirovali z [23], kde se konstatuje, že není důvod k ukládání celkového počtu projitých bitů (tedy i replikovaných). Tím dostáváme na vstup kodéru nižší čísla. Kódujeme kódem s proměnnou délkou, protože vzdálenost může být obecně velká (obzvášť u dat podobným trénovacím –

[5] připomeňme, že pro kompresi i dekompresi používáme stejný překladový automat, pouze s obráceným vstupem a výstupem

i	$\omega(i)$	$\omega'(i)$	$Fib_2(i)$	$Gol_3(i)$	$Rise_{2^2}(i)$	$Rise_{2^3}(i)$
0	–	–	–	(10)	(100)	(1000)
1	0	00	11	110	101	1001
2	100	0100	011	111	110	1010
3	110	0110	0011	010	111	1011
4	101000	1000	1011	0110	0100	1100
5	101010	1010	00011	0111	0101	1101
6	101100	1100	10011	0010	0110	1110
7	101110	1110	01011	00110	0111	1111
8	1110000	01110000	000011	00111	00100	01000
9	1110010	01110010	100011	00010	00101	01001
10	1110100	01110100	010011	000110	00110	01010
11	1110110	01110110	001011	000111	00111	01011
12	1111000	01111000	101011	000010	000100	01100
13	1111010	01111010	0000011	0000110	000101	01101
14	1111100	01111100	1000011	0000111	000110	01110
15	1111110	01111110	0100011	0000010	000111	01111
16	10100100000	100100000	0010011	00000110	0000100	001000
17	10100100010	100100010	1010011	00000111	0000101	001001
18	10100100100	100100100	0001011	00000010	0000110	001010
19	10100100110	100100110	1001011	000000110	0000111	001011
20	10100101000	100101000	0101011	000000111	00000100	001100
$\sum$ [b]	142	135	114	117	105	98

Tabulka 6.3: Kódy s pohyblivou délkou – Elias, Fibonacci, Golomb, Rise

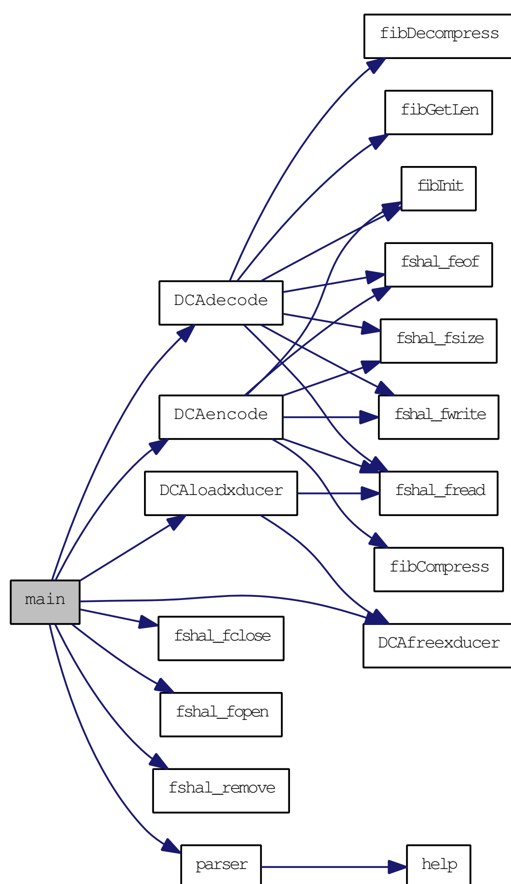
málo výjimek) a v případě širokého blokového kódu a velkého množství nízkých čísel (u dat nepodobným trénovacím – mnoho výjimek) bychom obdrželi značnou redundanci.

V případě malého počtu výjimek budeme potřebovat kódovat velká čísla, ale bude jich malé množství. Horší situace nastane v případě velkého množství malých výjimek (kódujících vzdálenosti cca 0–50). Výběr kódu tedy provádíme s ohledem na délku nízkých čísel. Do tabulek 6.2 a 6.3 jsme si vypsali základní a odvozené pohyblivé kódy pro čísla 0–20 za účelem výběru toho, který použijeme. Nejmenší součet délky zakódování prvních 20 čísel mají kódy *Fibonacci* a *Rise* (*Golombův* kód pro základ s mocninou 2). Rise kód má ale až příliš nevhodnou distribuci délek pro velmi vysoká čísla (protože obsahuje prefix tvořený α kódem) a ke kódování výjimek jsme se rozhodli použít kód Fibonacciho.

Fibonacciho kód je zápis čísla s bází Fibonacciho posloupnosti a má tu vlastnost, že se v něm nikde nevyskytují 2 znaky '1' za sebou. Sekvence '11' tak může být použita k zakódování hranic. Základní verze vychází z Fibonacciho posloupnosti 2. řádu $(1, 1, 2, 3, 5, 8, 13 \ldots)$. Posloupnosti vyšších řádů, jejich součty a součty jejich součů se používají jako báze pro Fibonacciho kódy vyšších řádů. Ty mají méně strmou závislost nárůstu délky kódových slov, ale vyšší její absolutní hodnotu pro malá čísla. Vzhledem k tomu, že my optimalizujeme hlavně délku kódování nízkých čísel, volíme základní verzi na bázi posloupnosti 2. řádu.

Proces kódování/dekódování výjimek musí, stejně jako komprese dat, pracovat s jednotlivými bity a je nutné cachovat nedokončené kousky v jednotkách po 32 bitech. V tomto ohledu je o něco komplikovanější dekódování, protože se dopředu neví, kolik bitů je třeba načíst – kde končí následující výjimka. K usnadnění načítání jsme do modulu Fibonacciho kódování dopsali funkci pro detekci délky. Pro debugovací a statistické účely je možné Fibonacciho kódování vypnout a přepnout na kódování blokovým kódem oddefinováním makra `SDCA_EXCEPT_FIBONACCI`.

Výjimky se ukládají a načítají z dedikovaného souboru. Ačkoliv takto to řeší i implementace [23], není to řešení úplně čisté, poněvadž pracuje se dvěma datovými proudy. Vetronics obsahuje jednoduchý FS, a tak si můžeme toto dovolit. Řešení nad jedním proudem by vyžadovalo znalost vzdálenosti následující výjimky ještě před její detekcí v kompresoru. K tomu by byl třeba lookahead buffer s neomezenou délkou, protože výjimka může nastat libovolně daleko. V praxi by se dal tento problém obejít známou velikostí bufferu a rezervovanou hodnotou kódující absenci výjimky v tomto intervalu. Poté by se prokládal bitový proud dat s proudem výjimek. Toto jsme však implementovat nepotřebovali.



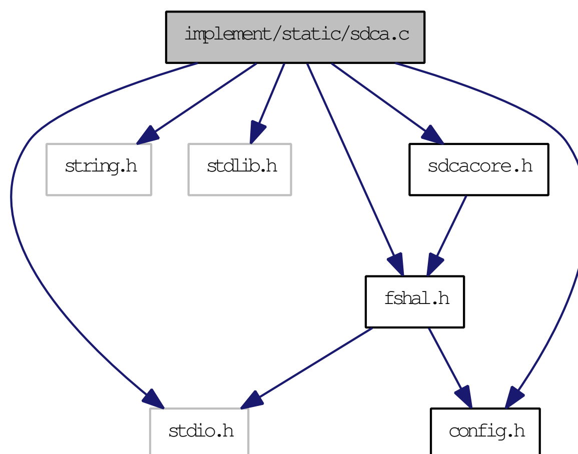
Obrázek 6.3: Volací diagram programu sdca

Dodejme ještě, že Fibonacciho kódování nemá definováno kódové slovo pro nulu, která se může ve vzdálenostech mezi výjimkami vyskytovat (a v případě dat hodně odlišných od trénovacích se vyskytuje často). Proto se ve skutečnosti kóduje číslo po přičtení jedničky. Pokud výjimky nejsou potřeba vůbec (ve vstupním souboru žádné nebyly), soubor s výjimkami se nezapíše (resp. má nulovou velikost). Prázdný soubor s výjimkami je možné smazat a `sdca` v dekompresním módu (`-x`) pak s výjimkami nepracuje vůbec (s výpisem varování, že nemusí jít o žádoucí stav).

6.3.2.3 Standardizační mezivrstva

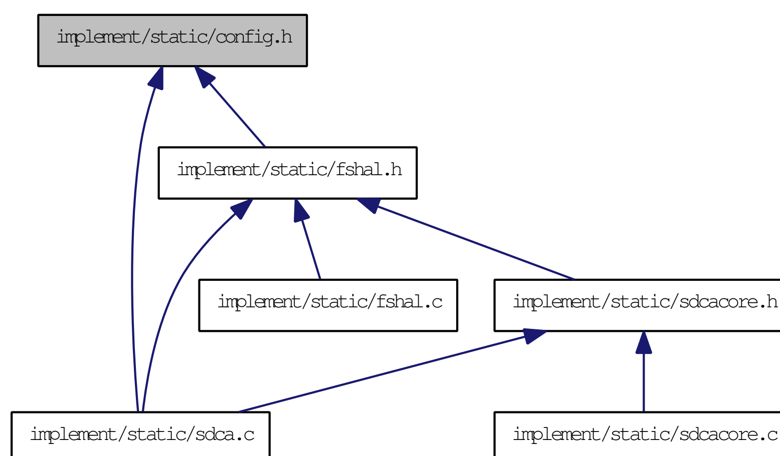
Na obr. 6.3 máme volací diagram programu `sdca`. Povšimněme si v něm funkcí s prefixem `fshal_*`. Jedná o mezivrstvu pro práci se soubory, kterou jsme museli dopsat pro dosažení multiplatformity kódu. C knihovna Vetronicsu totiž neobsahuje funkce pro práci se soubory, ani datový typ `FILE`. Tato mezivrstva obsahuje všechny základní souborové operace a při překladu pro Vetronics obhospodařuje seznam otevřených souborů s jejich aktuálními pozicemi. Typy a pořadí parametrů jejích funkcí i jejich návratové hodnoty jsou v souladu se standardem ANSI.

Moduly programu a jádra komprese (`sdcacore.c`) pak místo standardních funkcí přistupují k souborovému systému prostřednictvím této mezivrstvy, jak ukazuje graf závislostí na obr. 6.4.



Obrázek 6.4: Závislosti zdrojového souboru kompresního modulu

Volba platformy, pro kterou se má překládat, se provádí makrem `JSC_STR912` v hlavičkovém souboru `config.h`. Při kompilaci pro Vetronics se z `config.h` importuje ještě `project.h`, kde je naše makro nastaveno na 1. Závislosti modulů na `config.h` zachycuje diagram na obr. 6.5.



Obrázek 6.5: Vložení hlavičky config.h – volba platformy

Pro změnu cílové platformy tedy stačí pouze zaměnit `config.h`. Pro portaci na jinou nestandardní platformu by bylo třeba ještě dopsat deklaraci a definici nové mezivrstvy v modulu `fshal`.

6.4 Nasazení

Funkčnost a spolehlivost komprese byla nejprve pečlivě testována na PC. Testovali jsme nejen na souborech s logy z Vetronicsu, ale také na 14 referenčních souborech, tentokrát z Calgary Corpusu. Seznam jeho souborů máme v tab. 6.4. Testy probíhaly tak, že jsme si nejprve nechali vygenerovat 4 překladové automaty pro každý vstupní soubor – 2 z plného AD s hloubkou 15 a 20 bitů, 2 z prořezaného AD s hloubkou 25 a 30 bitů (velikosti souborů s automaty můžeme zjistit z výpisu adresáře `implement/static/tests/sdcx/`). Následně jsme s použitím všech vzniklých automatů zakomprimovali všechny vstupy (každý s každým) a zkontrolovali to, zda-li vzniklé soubory výjimek jsou prázdné pro dvojice vstup a jeho vlastní automat. Nakonec jsme provedli dávkovou dekompresi všech vzniklých souborů `.sdca` a výsledky binárně porovnali s původní sadou vstupů. Komprese se ukázala jako spolehlivá a od verze programu 0.2.x výše jsme neobdrželi žádný chybný výstup.

soubor	kategorie	velikost
bib	Bibliography (refer format)	111261
book1	Fiction book	768771
book2	Non-fiction book (troff format)	610856
geo	Geophysical data	102400
news	USENET batch file	377109
obj1	Object code for VAX	21504
obj2	Object code for Apple Mac	246814
paper1	Technical paper	53161
paper2	Technical paper	82199
pic	Black and white fax picture	513216
progc	Source code in "C"	39611
progl	Source code in LISP	71646
progp	Source code in PASCAL	49379
trans	Transcript of terminal session	93695

Tabulka 6.4: Soubory v Calgary Corpusu

Při testování jsme si ověřili to, že pro dosažení lepšího kompresního poměru je v případě výskytu výjimky vhodnější provádět restart automatu (soubory s výjimkami byly ve verzi bez restartu v průměru 4× větší a v součtu s datovými soubory činil rozdíl CR asi 10 %). Zároveň jsme stanovili výchozí hodnotu bitové hloubky konstrukce antislovníku v nástroji `dcaxgen` na 24 bitů. Dále jsme pozorovali, že na soubory s výjimkami je v případě velkého rozdílu trénovacích a vstupních dat možné použít kompresi RLE, protože opakované nulové vzdálenosti výjimek se v případě Fibonacciho kódování ukládají jako posloupnosti jedniček (0 se kóduje jako '11' a tyto dvojice se za sebou opakují).

Prostor pro další experimentování skýtá myšlenka komprimovat data statickým DCA vícekrát za sebou. Bylo by však nutné udržovat automaty pro několik úrovní komprese a citlivost na charakter dat by se nejspíš zvyšovala. Bylo by možné opakovanou kompresi zkoušet i na soubor s výjimkami. Toto by bylo nutno prověřit důkladněji. Zajímavá by mohla být i analýza výjimek a možnost průběžného vylepšování DCA automatu projitými daty.

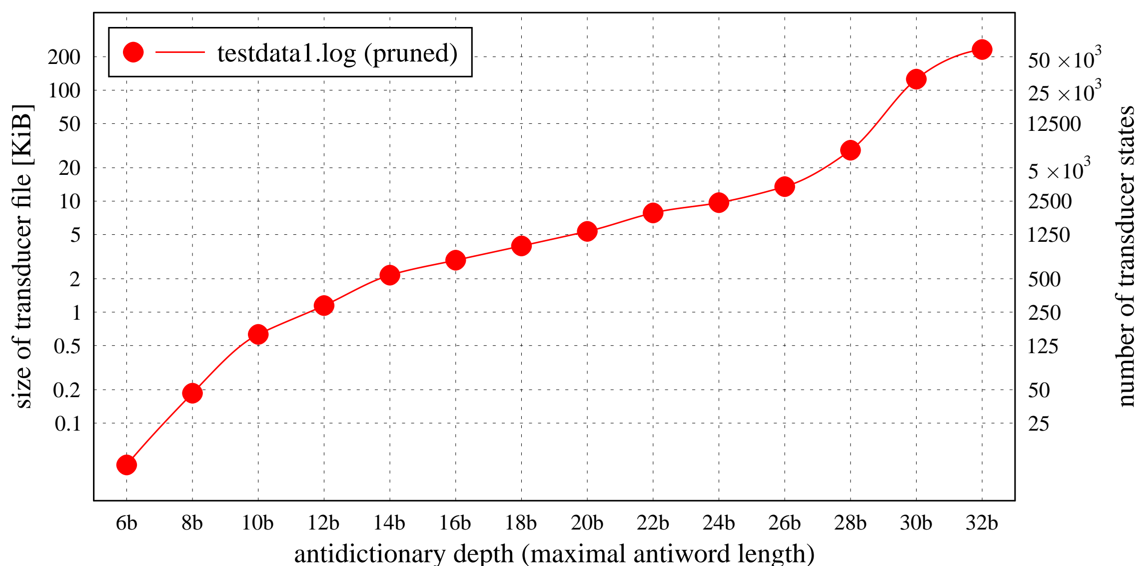
6.4.1 Testy v zařízení

S Vetronicsem komunikujeme přes sériový port v nastavení 38400 kbit/s, 8+1 bits, no parity, no handshaking. Soubory z/do PC posíláme protokolem Zmodem. V následujícím výpisu máme příklad komunikace a příkazů pro kompresi testovacího souboru logů `testdata2.log` o velikosti cca 0.5 MB pomocí automatu vygenerovaného ze souboru logů `testdata1.log`. Ve výpisu souborů (příkaz `ls`) můžeme vidět nově vytvořené soubory:

```
$ver
HW:6.11 KERN:6.1.1.18 SN:80102237 (47a16b16) t:4be0051e opt:00000001
FW:6.11 VER:6.255.1.252 t:4be02e44 REV:5157.10037
GSM: 0,0 Telit GE864-QUAD 07.02.003
GPS: ANTARIS ATR062x HW 00040001 EXT CORE 5.00 Jan 09 2006 12:00:00

$ls
 0  102     256 _ZLOG.stat.01
 1  104   13518 testdata1.26b.pruned.sdcx
 2  105    5342 testdata1.20b.pruned.sdcx
 3  100  478150 testdata2.log
 4  101 1319351 testdata1.log
Blocks: used=890 free=846(846) recy=0

$sdca testdata1.26b.pruned.sdcx testdata2.log testdata2.sdca testdata2.sdce

$ls
 0  102     256 _ZLOG.stat.01
 1  104   13518 testdata1.26b.pruned.sdcx
 2  105    5342 testdata1.20b.pruned.sdcx
 3  100  478150 testdata2.log
 4  101 1319351 testdata1.log
 5  107  288100 testdata2.sdca
 6  108     144 testdata2.sdce
Blocks: used=1032 free=702(702) recy=2

$sdca -x testdata1.26b.pruned.sdcx testdata2.sdca testdata2.out testdata2.sdce

$ls
 0  102     256 _ZLOG.stat.01
 1  104   13518 testdata1.26b.pruned.sdcx
 2  105    5342 testdata1.20b.pruned.sdcx
 3  100  478150 testdata2.log
 4  101 1319351 testdata1.log
 5  107  288100 testdata2.sdca
 6  108     144 testdata2.sdce
 7  109  478150 testdata2.out
Blocks: used=1266 free=468(468) recy=2
```

Soubor s výjimkami (`testdata2.sdce`) je velmi malý, což je dobré. Kompresní poměr je zde (bez započítání velikosti automatu) cca 0.6. Proces komprese trval asi 100 sekund, dekomprese asi 70 sekund. Rychlost vlastní komprese nemá cenu měřit, protože doba zpracování zavisí takřka výhradně na rychlosti připojené flash paměti. Navíc jsme příkaz `sdca` neimplementovali jako vlastní proces, nýbrž jako příkaz shellu (včetně parseru argumentů), jehož proces je v aktuální konfiguraci Vetronicsu omezen na využití 60 % jádra CPU. Doba zpracování zde ale není příliš důležitá, protože ke kompresi by docházelo jednou za čas (po odměření většího množství dat) a hardware nebývá plně vytížen. Pro srovnání poznamenejme, že dekomprese 0.5 MB Gunzipem zde trvá asi 10 minut. Pokud by se výrobce rozhodl kompresi adaptovat, dopsali bychom do ní ještě přítomnost kontrolního součtu (např. CRC32).

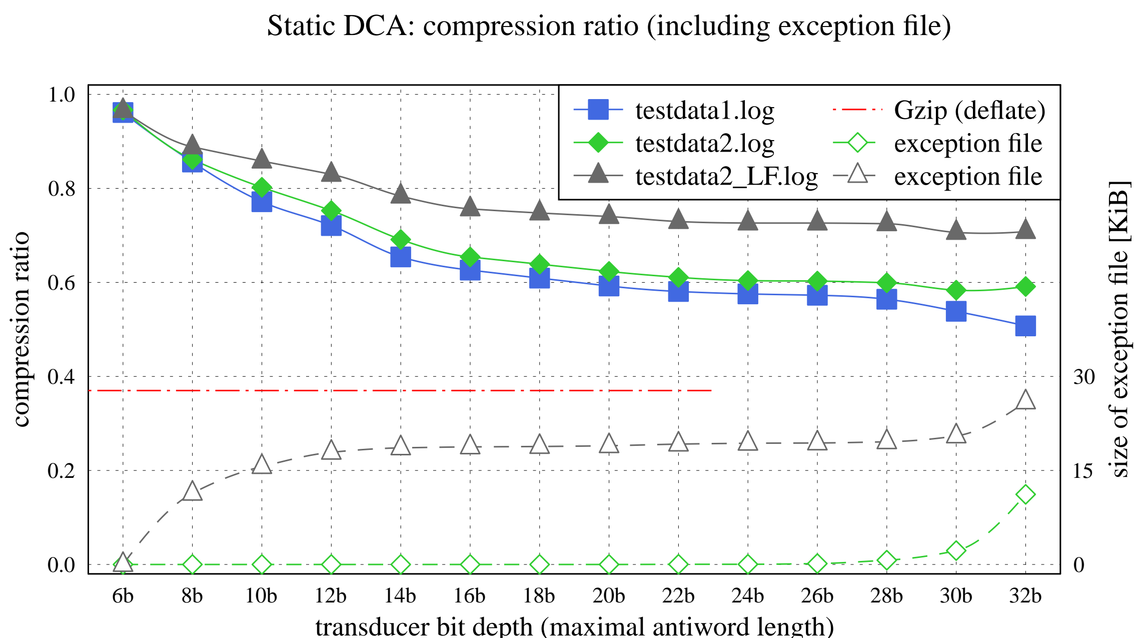
DCA překladový automat natrénovaný na souboru `testdata1.log` (ve výpisu soubory `testdata1.??b.pruned.sdcx`) se celý vejde do RAM až do AD hloubky 27 b. Na obr. 6.6 máme závislost jeho velikosti a počtu stavů na bitové hloubce. Vertikální osa má logaritmické měřítko, takže závilost je zhruba exponenciální, což odpovídá teorii.



Obrázek 6.6: Závislost velikosti sDCA překladače na hloubce AD

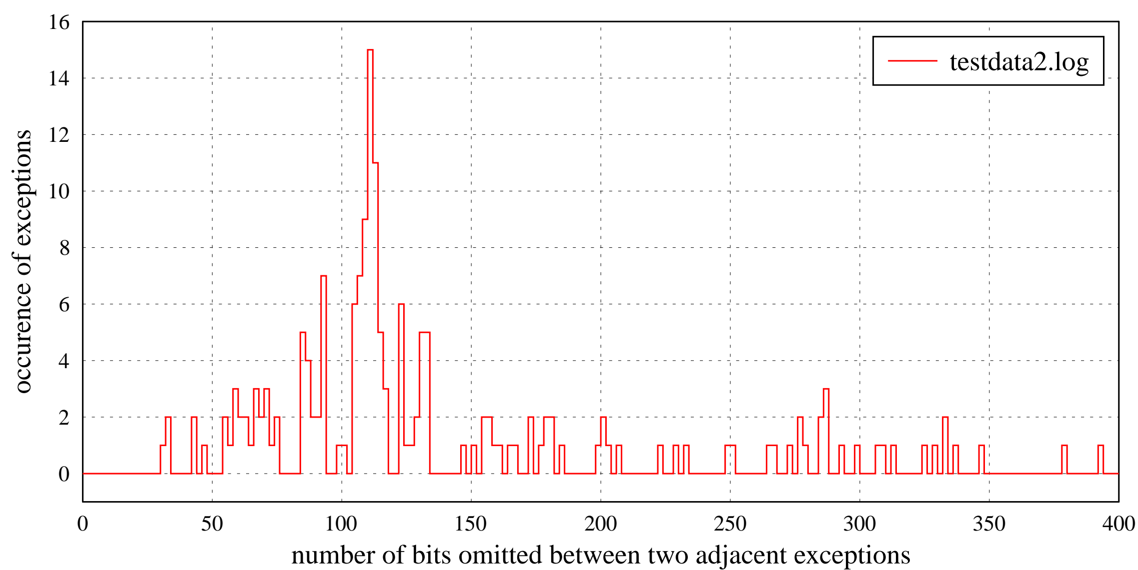
Závislost kompresního poměru testovacích dat pro různě velké automaty zobrazuje graf na obr. 6.7. Zde vidíme, že nejlépe se komprimují data, na kterých byl automat natrénován (`testdata1.log`), což jsme očekávali. Podobná data `testdata2.log`, která by se měla v budoucnu komprimovat, vykazují velmi podobné výsledky a i minimální počet výjimek. Křivka pro `testdata2_LF.log` se týká modifikovaného souboru s logem, který má řádky ukončené jen znakem `LF` narozdíl od původního `CRLF`. Tento experiment jsme provedli jako test citlivosti na zásadní změnu ve vstupních datech.

Pozorujeme, že velmi výrazně vzrostl počet výjimek a kompresní poměr se zhoršil až o 10 %. Komprese se tedy evidentně charakteristiku původních dat naučila. Ačkoliv statické DCA nedosahuje výsledků metody deflate (gzip zkomprimoval log na necelých 40 %), na dodané soubory logů Vetronicsu se zdá být použitelná (a narozdíl od gzipu jde spustit na jeho HW). Podle grafu 6.7 volíme výchozí hloubku automatu pro tuto situaci na 26 bitů. Pro vyšší hloubky se má komprese tendenci přeučovat a roste počet výjimek.



Obrázek 6.7: Kompresní poměry sDCA na testovacích datech, velikosti výjimek

Pro úplnost dodáváme ještě histogram výskytu výjimek v souboru `testdata2.sdce` na obr. 6.8. Znázorňuje naměřenou distribuci kódovaných vzdáleností mezi sousedními výjimkami.



Obrázek 6.8: Histogram výskytu výjimek v sDCA na datech podobných trénovacím

Kapitola 7

Závěr

Podrobně jsme se seznámili s kompresní metodou DCA i s její existující implementací. Na základě znalosti antislovníku jsme navrhli a zkonstruovali konečný automat, který je v určité míře schopen vyhledávat v původním textu. Automat může výskyt vzorku vyvrátit i potvrdit. Pokud se do antislovníku při jeho konstrukci nedostane potřebné množství informace, automat odpovídá neurčitou odpovědí. Jeho implementaci jsme provedli simulací v nedeterministické podobě s paměťovou složitostí $\mathcal{O}(|AD|)$ a časovou $\mathcal{O}(|P| \times |AD|)$. Implementace umí navíc detekovat antislovník, který není minimální. Byly naprogramovány nástroje pro testování a realizovány podrobné experimenty efektivity a síly vyhledávání. Vyhledávání je použitelné, ovšem nejlépe pro nezarovnaný bitový proud, protože z podstaty metody DCA nerespektuje zarovnání do bajtů. Jeho integrace do knihovny ExCom byla problematická, protože ExCom adoptuje jen dynamickou verzi DCA, u které se antislovník do souboru neukládá. Adopci semi-adaptivní verze DCA do ExCom dáváme jako námět práce do budoucna. Experimenty ukazují, že vyhledávání funguje lépe na větších datech, což nás inspiruje k dalšímu námětu budoucí práce – možnost využití metadat komprese DCA k indexaci dat.

V druhé části práce jsme za pomocí vhodného hardwaru demonstrovali nasazení komprese DCA v praxi. Navrhli jsme kompresní schéma a implementovali ho pro použitou platformu i pro PC. Hlavní řešený problém spočíval v omezení paměťových nároků komprese. Ten jsme vyřešili dvouúrovňovým schématem, kdy na výkonné platformě jednou proběhne předzpracování charakteristických dat a nízkoprostředkový hardware poté provádí opakovanou kompresi pomocí dodaného DCA překladového automatu. Takto jsme navíc získali možnost určit, kolik paměti bude moci komprese maximálně použít. Na testovacích datech jsme dosáhli kompresní poměr 0.6. Jiné bezeztrátové metody dosahovaly sice lepšího poměru, ale omezení množství paměti jejich použití vylučovalo. Výsledky nasazení naznačují, že u statického DCA existuje prostor pro experimenty s ještě omezenějším hardwarem. Při nasazení v proudově orientované aplikaci by bylo nutné provádět kódování dat i výjimek do jednoho proudu. Způsob řešení takového prokládání jsme navrhli a jeho implementaci necháváme jako možný námět na budoucí práci.

Práce nás obohatila zejména v oblasti stringologie, komprese dat a formálních úprav při publikování vědecko-akademické literatury.

Literatura

[1] LaTeX online manuál [CSTUG].
http://www.cstug.cz/latex/lm/. (14. 11. 2009).

[2] LaTeX 2_ε reference manual (September 2009).
http://home.gna.org/latexrefman/.

[3] CSTUG — CSTeX Users Group.
http://www.cstug.cz. (14. 11. 2009).

[4] GLE — graphics layout engine: Quality graphs, plots, diagrams, and figures.
http://www.gle-graphics.org. (6. 5. 2010).

[5] Instalace a počeštění MiKTeXu (Antonín Karolík).
http://www.mujweb.cz/www/miktex/. (31. 10. 2008).

[6] Ipe — extensible drawing editor with support for TeX typesetting.
http://tclab.kaist.ac.kr/ipe/. (7. 10. 2009).

[7] K336 Info — pokyny pro psaní diplomových prací.
https://info336.felk.cvut.cz/clanek.php?id=400. (7. 12. 2009).

[8] The Prague Stringology Club – doc. Ing. Jan Holub, Ph.D. (DTCS, FIT, CTU).
http://www.stringology.org/~holub.

[9] Webová stránka podpory výuky předmětu X36PJP – Programovací jazyky a překladače.
http://service.felk.cvut.cz/courses/X36PJP/, 2007.

[10] Webová stránka podpory výuky předmětu X36JPR – Jazyky a překlady.
http://service.felk.cvut.cz/courses/X36JPR/, 2008.

[11] Webová stránka podpory výuky předmětu X36KOD – Komprese dat.
http://service.felk.cvut.cz/courses/X36KOD/, 2008.

[12] Webová stránka podpory výuky předmětu X36TPR – Tvorba překladačů.
http://service.felk.cvut.cz/courses/36TPR/, 2008.

[13] Webová stránka podpory výuky předmětu X36TAL – Textové algoritmy.
http://service.felk.cvut.cz/courses/36TAL/, 2009.

[14] Wiki books — LaTeX.
http://en.wikibooks.org/wiki/LaTeX/, December 2009.

[15] Wikipedia, the free encyclopedia.
http://www.wikipedia.org, December 2009.

[16] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: a new structure for pattern matching. 1999.

[17] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle, suffix oracle. 1999.

[18] M. Crochemore. Indexing texts and data compression, 1999. (SOFSEM'99 slides).

[19] M. Crochemore. Reducing space for index implementation. 2002.

[20] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Data compression using antidictionaries. *IEEE Special Issue On Lossless Data Compression*, pages 180–192, 2000.

[21] M. Crochemore and G. Navarro. Improved antidictionary based compression. 2002.

[22] M. Davidson and L. Ilie. Fast data compression with antidictionaries. *Fundamenta Informaticae*, 2005(64):119–134, 2005.

[23] M. Fiala. Implementation of DCA compression method. Master's thesis, CTU FEE Prague, 2007.

[24] J. Holub. *Simulation of Nondeterministic Finite Automata in Pattern Matching*. PhD thesis, CTU FEE Prague, 2000.

[25] J. Holub. Simulation of NFA in pattern matching, 2004. (Athens 2004 slides).

[26] F. Šimek. Data compression library. Master's thesis, CTU FEE Prague, 2009.

[27] J. Kolář. *Teoretická informatika*. Česká informatická společnost, 2. edition, 2004.

[28] J. Lahoda, B. Melichar, and J. Žďárek. Pattern matching in DCA coded text. In *CIAA conference*, 2008.

[29] B. Melichar. *Gramatiky a automaty*. Vydavatelství ČVUT, reprint edition, 1979.

[30] B. Melichar. *Překladače - Postgraduální studium*. Vydavatelství ČVUT, 1. edition, 1986.

[31] B. Melichar. *Textové informační systémy*. Vydavatelství ČVUT, 1994.

[32] B. Melichar. Tvorba překladačů - cvičení. leden 2001.

[33] B. Melichar. *Jazyky a překlady*. Vydavatelství ČVUT, 2. edition, 2003.

[34] B. Melichar. Textové informační systémy - cvičení. červenec 2003.

[35] B. Melichar. Jazyky a překlady - cvičení. březen 2004.

[36] B. Melichar and L. Vagner. Compiler construction. March 2006.

[37] Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Pattern matching in text compressed by using antidictionaries. 1999.

Dodatek A

Slovník zkratek

AD Antidictionary

ANSI American National Standards Institute

API Application Programming Interface

ASCII American Standard Code for Information Interchange

AW Antiword

CPU Central Processor Unit

CR Compression Ratio

DCA Data Compression Using Antidictionaries

DFA Deterministic Finite Automaton

DMA Direct Memory Access

FA Finite Automaton

FAT File Allocation Table

FS File System

FSM Finite State Machine

GNU GNU's Not Unix

GPL GNU General Public License

GPRS General Packet Radio Service

GPS Global Positioning System

GSM Global System for Mobile Communications

HW Hardware

IDE Integrated Development Environment

IRQ Interrupt Request Controller

ISO International Organization for Standardization

JTAG Joint Test Action Group

LBGA Low-Profile Ball Grid Array

LQFP Low-profile Quad Flat Package

LSB Least significant byte

LSb Least significant bit

LZW Lempel–Ziv–Welch (compression method)

MSB Most significant byte

MSb Most significant bit

MinGW Minimalist GNU for Windows

MCU Microcontroller Unit

NFA Non-deterministic Finite Automaton

POSIX Portable Operating System Interface for Unix

RISC Reduced Instruction Set Computer

RLE Run-Length Encoding

SPI Serial Peripheral Interface

SRAM Static Random Access Memory

UART Universal Asynchronous Receiver/Transmitter

UCS Universal Character Set

UTF UCS Transformation Format

Dodatek B

Obsah přiloženého CD

```
+---README.TXT              Readme (základní informace a obsah CD)
+---doc-dca                 referencovaná literatura o DCA
+---doc-matching            referencovaná literatura o „pattern matching“
+---doc-other               referencovaná literatura ostatní
+---implement               implementace
|   +---Fiala               implementace DCA [23]
|   +---FSimek              implementace ExCom [26]
|   +---Princip             kompilátory pro platformu Vetronicsu
|   +---search              implementace vyhledávání v AD
|   |   +---corpuses        referenční data
|   |   |   +---cantrbry    Canterbury Corpus – původní data
|   |   |   +---cantrbry.ad.*  Canterbury Corpus – antislovníky
|   |   |   +---cantrbry.bit   Canterbury Corpus – bitový tvar
|   |   |   +---distribs    Canterbury Corpus – distribuce AW
|   |   |   +---graphs      Canterbury Corpus – grafy
|   |   +---excom           integrace vyhledávání do ExCom – změny
|   |   +---testdata        další testovací data (první log z Vetronicsu)
|   |   +---tests           experimenty s vyhledáváním v AD
|   |   |   +---graphs      experimenty – grafy
|   |   |   +---positive*   experimenty – výsledky na pozitivních vzorcích
|   |   |   +---random*     experimenty – výsledky na náhodných vzorcích
|   |   |   +---*.pat       experimenty – vzorky
|   |   +---tools           pomocné nástroje pro vyhledávání
|   |   |   +---ADdump      vypisovač antislovníku v .dca souborech
|   |   |   |   +---dcamod  modifikované zdroje implementace DCA
|   |   |   +---patterns-cut   generátor obsažených bitových vzorků
|   |   |   +---patterns-gen   generátor náhodných bitových vzorků
|   |   +---autotest.sh     hlavní skript pro automatické testování
|   |
```

```
|   |
|   +---static                   implementace statického DCA
|       +---corpuses             Calgary Corpus (vč. velké verze)
|       +---Princip              modifikované zdroje Vetronicsu
|       +---testdata             testovací data – logy z Vetronicsu
|       +---tests                testování vlastností a spolehlivosti komprese
|       |   +---input            vstupy do testování spolehlivosti
|       |   +---output           výstupy křížové komprese (input/ × sdcx/)
|       |   +---Princip          experimenty se sDCA na testovacích lozích
|       |       +---graphs       experimenty se sDCA – grafy
|       |   +---sdcx             automaty aplikované na vstupní soubory
|       +---tools                pomocné nástroje pro sDCA
|           +---dcaxgen          generátor překladače pro sDCA
|           |   +---dcamod       modifikované zdroje implementace DCA
|           +---fibonacci.sh     výpočet Fibonacciho posl. řádu n a jejích supersoučtů
+---text                         text práce
    +---diagrams                 diagramy
    +---doxygen                  dokumentace z nástroje Doxygen
    |   +---html                 Doxygen – hypertextová dokumentace
    |   +---refman.pdf           Doxygen – LaTeXová dokumentace
    +---figures                  obrázky
    |   +---vetronics            fotografie jednotky Vetronicsu
    +---graphs                   použité grafy ve vektorovém formátu
    +---Princip                  technická specifikace a montážní návody Vetronicsu
    +---dp.pdf                   diplomová práce – výstup v PDF
    +---dp.tex                   diplomová práce – zdroj v TeXu
    +---dp-zadani.rtf            diplomová práce – oficiální zadání
```

Dodatek C

Použití vybraných programů

C.1 search_cli – konzolová verze vyhledávání v AD

```
$ ./search_cli --help
search_cli 0.4.3  Copyleft (c) Jan Skalicky 2010
================
effect:   searches for a pattern in text using DCA antidictionary
usage:    search_cli [-v] [-q] [-e] [file]
----------------
<options>
 -?:      help screen
 -v:      verbose mode
 -q:      quiet mode
 -e:      extended search - counts a number of resolving AWs
<I/O>
 file:    text file with program for automatic testing
----------------
(*)       default option
example:  search_cli -q -e autotest.txt >autotest.out
```

C.2 dcaxgen – generátor PKA pro statické DCA

```
$ ./dcaxgen --help
dcaxgen 0.3.6  Copyleft (c) Jan Skalicky 2010
================
effect:   generates transducer for static DCA compression/decompression
usage:    dcaxgen [options] infile outfile
----------------
<options>
 -?:      help screen
 -v:      verbose mode
 -p:      enable simple pruning on antidictionary (*)
 -nop:    disable simple pruning (less efective, used in special cases)
 -l=%:    antidictionary depth (max. AW length in bits, 24 (*))
<I/O>
 infile:  sample data with characteristic content for antidictinary generation
 outfile: output file to store DCA transducer
----------------
(*)       default option
example:  dcaxgen -v -l=28 sample.dat sample.sdcx
```

C.3 sdca – (de)kompresor souborů statickým DCA

```
$ ./sdca --help
sdca 0.6.3  Copyleft (c) Jan Skalicky 2010
================
effect:   compress/decompress file with given DCA transducer (static method)
usage:    sdca [-x] xducer infile [outfile] [excepts]
----------------
<options>
 -?:      help screen
 -x:      decompression mode
<I/O>
 xducer:  file with stored DCA transducer
 infile:  input file
 outfile: output file (infile.sdca (*) on compress)
 excepts: file to load or save exceptions (infile.sdce (*))
----------------
(*)       default option
example:  sdca sample.xducer data.bin
```

Dodatek D

Fotografie Vetronicsu

SIM
MJ2732 VEP
e8
03 0031
S/N: 111400509020951
OPEN
SIMLOCK
AMPHENOL
s/n:
111400509020951
producer: PRINCIP a.s.
made in Czech Republic

Dodatek E

Originální specifikace Vetronicsu

(vlastní dokument, 3 strany)