

České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra softwarového inženýrství
Obor: Aplikace softwarového inženýrství



Porovnání účinnosti komprese dat ve formátech XML a JSON

Comparison of the effectiveness of data compression in XML and JSON format

DIPLOMOVÁ PRÁCE

Vypracoval: Bc. Tomáš Smola
Vedoucí práce: Ing. Tomáš Liška, Ph.D.
Rok: 2015

Před svázáním místo téhle stránky

vložte zadání práce

 s podpisem děkana (bude to jediný oboustranný list ve Vaší práci) !!!!

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v příloženém seznamu.

V Praze dne

.....
Bc. Tomáš Smola

Poděkování

Děkuji Ing. Tomáši Liškovi, Ph.D. za vedení mé diplomové práce a za podnětné návrhy, které ji obohatily.

Bc. Tomáš Smola

Název práce:

Porovnání účinnosti komprese dat ve formátech XML a JSON

Autor: Bc. Tomáš Smola

Obor: Aplikace softwarového inženýrství

Druh práce: Diplomová práce

Vedoucí práce: Ing. Tomáš Liška, Ph.D.

Katedra softwarového inženýrství, Fakulta jaderná a fyzikálně
inženýrská, České vysoké učení technické v Praze

Konzultant: —

Abstrakt: Abstrakt

Klíčová slova: Klíčová slova

Title:

Comparison of the effectiveness of data compression in XML and JSON format

Author: Bc. Tomáš Smola

Abstract: Abstract

Key words: Key words

Obsah

Úvod	1
1 Obecné seznámení s formáty XML a JSON	2
1.1 XML	2
1.1.1 Charakteristika	2
1.1.2 Syntaktická analýza	3
1.1.3 Vzorový příklad	3
1.2 JSON	4
1.2.1 Charakteristika	4
1.2.2 Syntaktická analýza	4
1.2.3 Vzorový příklad	6
1.3 Vzájemné porovnání XML a JSON	7
2 Komprese dat	8
2.1 Princip komprese dat	8
2.2 Typy kompresních metod	8
2.3 Charakteristika komprese	9
2.4 Míra informace v datech	9
2.4.1 Entropie informačního zdroje	11
2.4.2 Entropie a komprese	11
3 Statistické techniky komprese	13
3.1 Pravděpodobnostní model	13
3.1.1 Statický model	13
3.1.2 Semiadaptivní model	13
3.1.3 Adaptivní model	14
3.2 Semiadaptivní Huffmanovo kódování	14

3.2.1	Postup kódování a dekódování zprávy	14
3.2.2	Vzorový příklad	14
3.3	Adaptivní Huffmanovo kódování	15
3.3.1	Vzorový příklad	16
4	Slovníkové techniky komprese	18
4.1	Základy slovníkové komprese	18
4.1.1	Typy slovníků	19
4.2	LZ77	19
4.2.1	Princip komprese	19
4.2.2	Příklad komprese	20
4.2.3	Princip dekomprese	20
4.2.4	Variace algoritmu LZ77	20
4.3	LZ78	21
4.3.1	Princip komprese	22
4.3.2	Příklad komprese	22
4.3.3	Princip dekomprese	22
4.3.4	Variace algoritmu LZ78	22
5	Existující algoritmy pro kompresi XML	24
5.1	XMill	24
5.1.1	Popis architektury	25
5.1.2	Oddělení struktury od dat	25
5.1.3	Seskupení dat stejného významu	26
5.1.4	Sémantická komprese	26
5.1.5	Nedostatky tohoto přístupu	26
5.2	XGrind	26
5.2.1	Kompresní techniky	27
5.2.2	Popis architektury	27
5.2.3	Dotazování	28
6	Existující algoritmy pro kompresi JSON	29
6.1	JSONH	29
6.1.1	Vzorová data	29
6.1.2	Postup komprese	30

6.1.3	Data po kompresi	30
6.1.4	Postup rekonstrukce dat	30
6.2	CJSON	30
6.2.1	Vzorová data	31
6.2.2	Postup komprese	31
6.2.3	Data po kompresi	32
6.2.4	Postup rekonstrukce dat	32
7	Vlastní implementace vybraných kompresních algoritmů	33
7.1	Technická specifikace	33
7.1.1	Vývojové prostředí a programovací jazyk	33
7.1.2	Implementace a testování	33
7.1.3	Projekt	33
7.2	Vybrané algoritmy	34
7.2.1	Adaptivní Huffmanovo kódování	34
7.2.2	LZ78	35
7.2.3	JSONH	35
8	Porovnání účinnosti komprese dat ve formátu XML a JSON	37
	Závěr	38
	Seznam použitých zdrojů	39
	Přílohy	40
A	Název/obsah přílohy	41
A.1	Porovnání XML a JSON	41

Úvod

Závěrečnou diplomovou práci ke studijnímu oboru Aplikace softwarového inženýrství s názvem Porovnání účinnosti komprese dat ve formátu XML a JSON jsem si vybral z důvodu aktuálnosti, XML a JSON jsou v současnosti jedny z nejpoužívanější textových datových formátů, a také proto, že toto téma velmi dobře propojuje teoretické znalosti získané při studiu s praktickými zkušenostmi v oboru softwarového inženýrství.

Dle výzkumu International Data Corporation (IDC) The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things [8] bylo pouze v roce 2014 vytvořeno a zkonsumováno 2837 EB (exabytů) dat. Ze závěrů vyplývá, že se toto číslo každé dva roky zdvojnásobí, tedy v roce 2020 to bude již přibližně 40000 EB. Takové množství dat klade enormní požadavky na přenosové kanály a datová úložiště. Jako příklad mohou sloužit miliony shlédnutí oblíbených videí na serveru youtube.com. Pouze jedna sekunda videa v nekomprimovaném formátu CCIR 601 zabere více než 20 MB, tímto způsobem by zmiňovaná služba nemohla fungovat.

Vzhledem k tomu, že jsou technologie omezeny současnými možnostmi, znalostmi a také fyzikálními limity, je nutné hledat řešení jinde než v jejich zlepšování. Zde přichází na řadu komprese dat jako účinná metoda snížení velikosti objemu přenášených a ukládaných dat. Ve své práci bych čtenáře rád seznámil se základními principy komprese a vybranými kompresními algoritmy. Hlavním cílem je ale zodpovědět otázku, zda je možné dosáhnout dalších úspor volbou XML nebo JSON formátu a vhodného algoritmu, který využije znalosti struktury datového formátu.

Kapitola 1

Obecné seznámení s formáty XML a JSON

V této kapitole seznámím čtenáře se značkovacím jazykem XML a následně s JSON, formátem pro výměnu dat. Mým cílem je popsat základní charakteristiky a syntaxi obou formátů tak, abych byl já, a následně i čtenář, schopen pochopit v kapitolách 5 a 6 principy a výhody vybraných algoritmů využívajících znalosti struktury dat.

1.1 XML

Na základě značkovacího jazyka SGML (Standard Generalized Markup Language), jehož obecnost činí úplnou implementaci náročnou, vznikl vybráním nejpoužívanějších možností značkovací jazyk XML (eXtensible Markup Language), je tedy podmnožinou jazyka SGML. XML je obecný a otevřený, jeho vývoj a standardizaci realizovalo konsorcium W3C (World Wide Web Consortium) [11]. XML umožňuje snadné vytváření konkrétních značkovacích jazyků pro popis dokumentů a dat ve standardizované, textově orientované podobě.

1.1.1 Charakteristika

V podstatě jde o textový dokument, jenž je tvořen posloupností Unicode¹ znaků, ve kterém se rozlišují dvě základní součásti: elementy (značky) a obsah. Strukturu zapisovaných dat je možné vystihnout vzájemným vnořováním jednotlivých značek, které se ale nesmí křížit. XML je pro člověka čitelný jazyk a spousta dnešních aplikací (například internetových prohlížečů) podporou zobrazení a vhodným formátováním tuto čitelnost ještě zvyšuje.

Jazyk XML se využívá hlavně pro publikování dokumentů, při výměně dat mezi různými systémy v prostředí internetu, jako univerzální úložiště dat či jako konfigurační soubor. Obecně lze říci, že je vhodný pro strukturovaná data. Do dokumentu můžeme vložit libovolná data, jejichž význam není bez použití schématu dokumentu zřejmý. Abychom mohli definovat sadu elementů (viz následující část 1.1.2), případně v nich i kontrolovat typ dat, definujeme schéma dokumentu, které je vloženo buď přímo, nebo formou odkazu na definiční dokument.

¹Unicode je standard pro konzistentní kódování, reprezentaci a manipulaci znaků většiny světových abeced.

1.1.2 Syntaktická analýza

Jak bylo řečeno již dříve, základními kameny XML jsou elementy a jejich obsah. Při práci s XML je nutné mít na paměti, že je na dodržení syntaxe kladen velmi velký důraz. Při dodržení správného způsobu zápisu a pravidel, která budou popsána níže, lze dokument považovat za tzv. well-formed XML [11].

Element

Základním prvkem každého XML dokumentu je element, který je vyznačen pomocí takzvaných tagů², mezi které může být vložen obsah. Počáteční i ukončující tag je dle definice [11] složen z dvojice znamének < (menší než) a > (větší než), mezi kterými je zapsán název tagu a volitelně i atributy. Ukončovací tag má navíc před svým názvem znak / (lomeno). Při správné aplikaci pravidel může vypadat element například následujícím způsobem:

```
<název_elementu název_atributu="hodnota atributu"></název_elementu>.
```

V případě, že element neobsahuje žádný obsah, lze ho zkráceně zapsat jako tzv. prázdný element:

```
<název_prázdného_elementu />.
```

V případě nedodržení správné syntaxe může nastat problém při rozpoznávání zapsaných dat, což může mít za následek nekompatibilitu mezi různými systémy při výměně dat.

Atribut

Počáteční tag elementu může obsahovat atributy upřesňující jeho význam. Atribut je vždy složen ze svého názvu a hodnoty, které jsou odděleny znakem = (rovná se). Hodnota je navíc zapsána mezi dvojicí znaků " (uvozovky) nebo ' (apostrof), přičemž hodnota může obsahovat jeden z těchto znaků tak, že se syntakticky nekříží. Následuje příklad atributu, jehož hodnota obsahuje znak ':

```
název_atributu="hodnota atributu obsahující znak ' (apostrof)".
```

Obsah

Vše, co není tagem, je v dokumentu považováno za obsah. Kromě obyčejného textu mohou být obsahem další vnořené elementy, komentáře, instrukce pro zpracování, reference a další. Vzhledem k tomu, že určité znaky mají v syntaxi XML speciální význam (např. <, >), využívají se pro jejich zápis znakové entity³. Úplný výčet toho, co může XML dokument obsahovat, je včetně pravidel definován v [11].

1.1.3 Vzorový příklad

Na následujících řádcích je zapsán XML element typu **Person**, oblíbená postavička Homer Simpson ze seriálu The Simpsons, který kromě jiného obsahuje vnořený element typu **Relatives**, který zde slouží jako kontejner s Homerovými příbuznými.

²Tag definuje formu části textu.

³Pomocí znakových entit (sekvence znaků) lze zapsat znaky, které neobsahuje zvolená znaková sada, nebo mají v použitém kontextu speciální význam.

```

<Person>
  <FirstName>Homer</FirstName>
  <LastName>Simpson</LastName>
  <Relatives>
    <Relative>Grandpa</Relative>
    <Relative>Marge</Relative>
    <Relative>Bart</Relative>
    <Relative>Lisa</Relative>
    <Relative>Maggie</Relative>
  </Relatives>
</Person>

```

1.2 JSON

JSON neboli JavaScript Object Notation je odlehčený způsob zápisu (formátování) dat. Navzdory svému názvu jde o datově orientovaný textový formát nezávislý na počítačové platformě a čitelný pro člověka. Vznikl jako alternativa ke XML v oblasti výměny dat mezi systémy bez ohledu na použité technologie.

1.2.1 Charakteristika

Jak již název napovídá, je JSON velice úzce spojen s programovacím jazykem JavaScript – je na jeho syntaxi je založen. Hlavními prvky JSON jsou dvě univerzální datové struktury: kolekce dvojic klíč/hodnota s unikátními klíči (tzv. associative array) a seřazený seznam hodnot (tzv. array), které podporují v nějaké formě asi všechny známé moderní programovací jazyky.

Silnou stránkou JSON je rychlost zpracování JavaScriptem, bohužel právě přímé zpracování je zároveň potenciální hrozbou, neboť může být zneužito útočníkem k vykonání nějakého škodlivého kódu.

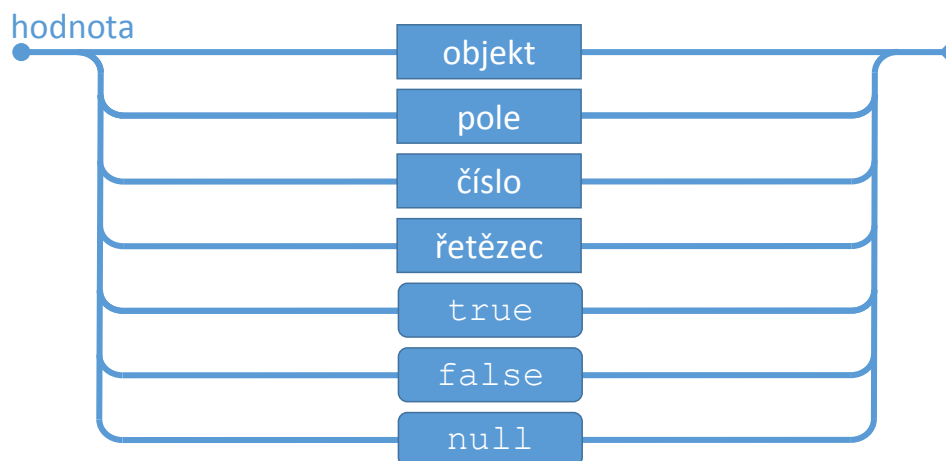
Orientace na data a zvolená syntaktická struktura umožňují přehledný zápis datových objektů. Protože co jiného než popis atributů (asociativní pole) a výčet hodnot (pole) datové objekty obsahují.

1.2.2 Syntaktická analýza

Dle definice [7] je JSON posloupností tokenů (viz 1.2.2) tvořených z Unicode znaků. Sada tokenů obsahuje šest strukturálních tokenů: [(levá hranatá závorka), { (levá složená závorka),] (pravá hranatá závorka), } (pravá složená závorka), : (dvojtečka) a , (čárka); dále obsahuje znakové řetězce, čísla a tři doslovné tokeny: **true**, **false** a **null**.

Hodnoty

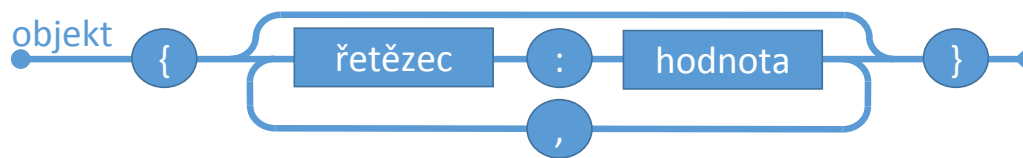
Za hodnotu je v JSON považován objekt, pole, číslo, řetězec, **true**, **false**, nebo **null**.



Obrázek 1.1: Struktura hodnoty

Objekty

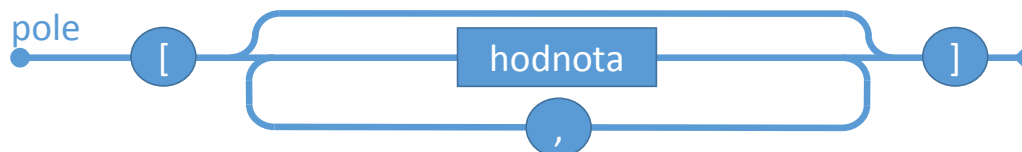
Objekt je reprezentován dvojicí složených závorek, uvnitř kterých je žádná nebo více dvojic klíč/hodnota, přičemž klíč je řetězec. Klíč a hodnota jsou odděleny dvojtečkou a jednotlivé dvojice odděluje čárka.



Obrázek 1.2: Struktura objektu

Pole

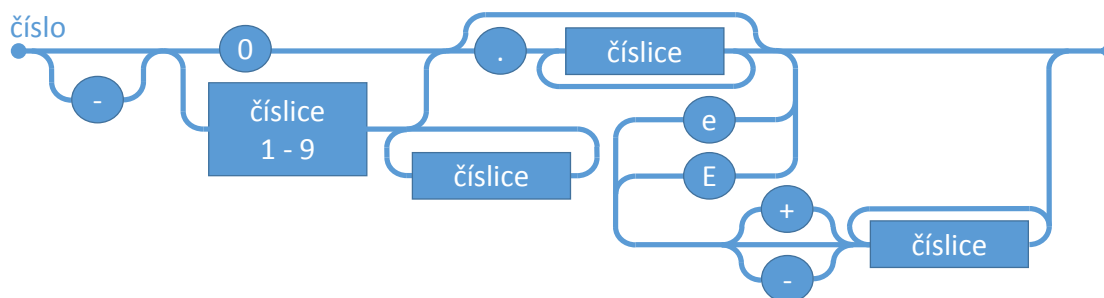
Pole je složeno z dvojice hranatých závorek, mezi kterými může být nula nebo více seřazených hodnot, které jsou odděleny čárkou.



Obrázek 1.3: Struktura pole

Čísla

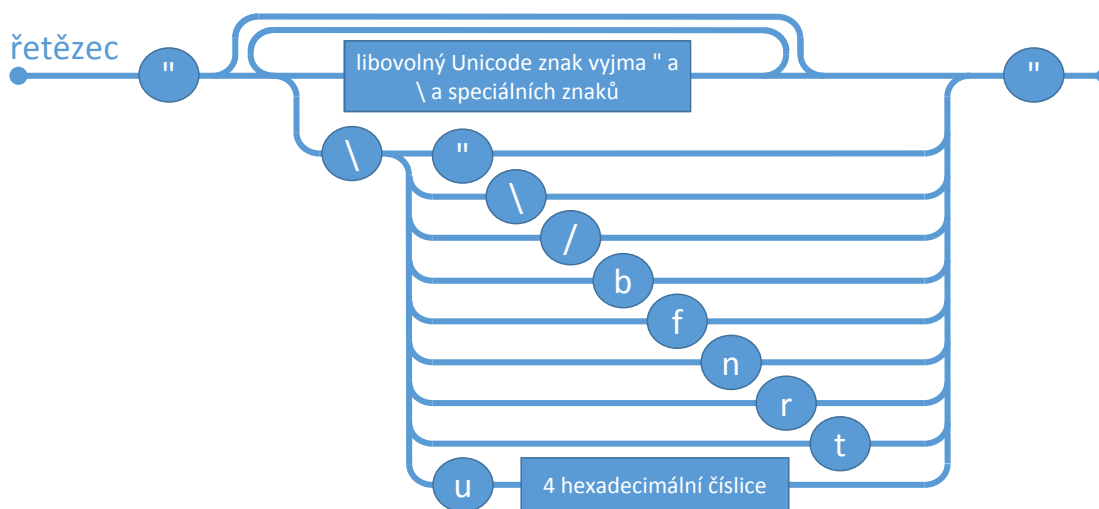
Čísla jsou v desítkové soustavě (tedy číslice 0 – 9), záporná čísla jsou uvozena znaménkem – (mínus), desetinná část je oddělena znaménkem . (tečka). Je možný i takzvaný vědecký zápis čísel s použitím symbolů *e* (malé *e*) nebo *E* (velké *e*) a volitelně lze použít u exponentu znaménka + (plus) nebo – (mínus).



Obrázek 1.4: Struktura čísla

Řetězce

Řetězec je posloupnost Unicode znaků uvozená a zakončená znakem " (uvozovky). Mezi uvozovky mohou být zapsány všechny znaky kromě speciálních znaků, které jsou uvozeny tzv. únikovým znakem \ (zpětné lomítko), těmito znaky jsou například ", \, t (znak tabulátoru), n (znak posunu kurzoru na nový řádek) a r (znak posunu kurzoru na začátek řádku, známý jako návrat vozíku) a další. Kompletní výčet je možné vidět na obrázku 1.5 nebo v [7]. Navíc je možné každý znak zapsat pomocí kombinace \u a čtyřmístného hexadecimálního čísla odpovídajícího Unicode kódu požadovaného znaku. Řetězec obsahující pouze zpětné lomítko můžeme tedy zapsat následujícími způsoby: "\\ ", "\u005c" nebo "\u005C" (u hexadecimálních čísel A-F nezáleží na velikosti).



Obrázek 1.5: Struktura řetězce

1.2.3 Vzorový příklad

Stejně jako ve vzorovém příkladu 1.1.3 ke XML, je zde popsána postavička Homera Simpsona včetně příbuzných, tentokrát ale v notaci JSON. Jde o objekt složený ze tří atributů, první dva mají hodnoty typu řetězec, k poslednímu **relatives** ale přísluší hodnota typu pole, ve kterém je vloženo pět příbuzných – hodnot typu řetězec.

```
{
  "firstName": "Homer",
```

```

"lastName": "Simpson",
"relatives": [
  "Grandpa",
  "Marge",
  "Bart",
  "Lisa",
  "Maggie"
]
}

```

1.3 Vzájemné porovnání XML a JSON

Je XML obecně lepší než JSON? Co vlastně dělá jeden formát lepší než jiný? Na toto bylo vedeno již nespočet diskusí. Argumentace obou stran diskuse je obvykle podložena jak teoretickou znalostí formátů, tak i praktickým použitím v reálných projektech. I přesto si myslím, že žádná diskuse nepřinesla jednoznačnou odpověď na výše položenou otázku, což samozřejmě pro některé konkrétní případy použití neplatí a volba jednoho z formátů je nutná nebo alespoň výhodnější. Ze svého zkoumání této problematiky si ale odnáším jeden podstatný závěr, který bych chtěl čtenáři zdůraznit: Vždy záleží na konkrétní situaci. Před samotným rozhodnutím, který z formátů využít, je vhodné zvážit důležité požadavky a do těchto úvah zahrnout klidně i další z existujících datových formátů. Zároveň lze ale říci, že pokud splňuje vybraný formát požadavky, tak ho lze považovat za dostatečný

Podíváme-li se znovu na vzorové příklady zápisu dat nebo položíme-li si vedle sebe stejná data zapsaná do obou formátů (viz příloha A.1), můžeme lehce vidět vzájemnou podobnost. Ve své práci proto vycházím z předpokladu, že jsou oba formáty z hlediska toho, jaká data lze do nich zapsat, ekvivalentní. Respektive budu pro další porovnání používat pouze takový druh dat, který bude tento předpoklad splňovat.

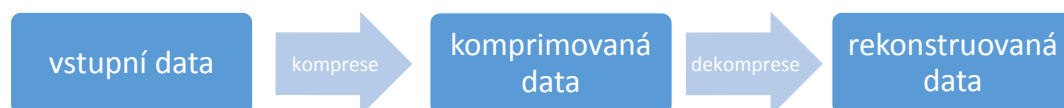
Kapitola 2

Komprese dat

Komprese nebo také komprimace dat je taková transformace dat, která má za cíl úsporu zdrojů při ukládání nebo archivaci a nebo snížení datového toku při přenosu, to vše při současném zachování informace obsažené v datech. Jinými slovy jde o redukci velikosti datových souborů, jehož následkem je úspora paměťových či přenosových kapacit. Postup, při kterém z komprimovaných dat rekonstruujeme data originální, se nazývá dekomprese.

2.1 Princip komprese dat

Data velmi často obsahují tzv. redundantní¹ informaci, toho právě využívá komprese – data jsou zpracována tak, aby byla redundance minimalizována. Jak lze vidět na obrázku 2.1, je na vstupní data použita operace komprese. Operací dekomprese dostaneme poté data rekonstruovaná – v závislosti na použité kompresní metodě, respektive na požadavcích získáme buď data přesně odpovídající původním a nebo pouze částečná. Z tohoto hlediska rozlišujeme dva typy kompresních metod: ztrátové a bezztrátové.



Obrázek 2.1: Princip komprese

2.2 Typy kompresních metod

Jak název napovídá, při ztrátové kompresi ztratíme část informace obsaženou v původních datech, respektive jsou původní data pouze aproximována. Toto nám nemusí vadit například u obrázků, zvuku a videí, kde je využito nedokonalosti lidských smyslů. Lidské ucho nedokáže například slyšet velmi vysoké frekvence. Má smysl v datech určených k poslechu zachovávat informaci, kterou nemůže člověk slyšet? Častá odpověď je „ne“. Tohoto principu využívá mnoho kompresních metod, například známý zvukový formát MP3. Odstraněním „nepotřebné“ informace z dat je dosaženo ještě větší redukce objemu.

¹Redundance znamená informační nadbytek, např. vícenásobný výskyt slov v textu.

Naopak v případě bezeztrátových metod je při kompresi zachována veškerá informace a při dekompresi jsou rekonstruována původní data. Těchto metod se využívá převážně tam, kde není možné původní data jakkoliv pozměnit. Například data ve formátech XML a JSON, kterým se věnuji v této práci, si nemůžeme dovolit pozměnit (přestanou mít původní význam), nebo dokonce ztratit.

2.3 Charakteristika komprese

Kompresní algoritmy lze hodnotit z mnoha různých úhlů pohledu. Můžeme měřit složitost algoritmu, rychlost, jakou jsou data komprimována a dekomprimována (to může být ovlivněno výkonem stroje, na kterém algoritmus běží), jak moc odpovídají rekonstruovaná data původním atd. Jednou z nejčastějších charakteristik je, logicky ze smyslu komprese vyplývající, tzv. kompresní poměr, který vyjadřuje velikost komprimovaných dat vůči původním, lze ho zapsat následujícím vztahem:

$$\text{kompresní poměr} = \frac{\text{délka původních dat}}{\text{délka komprimovaných dat}}. \quad (2.1)$$

Další sledovanou charakteristikou je tzv. úspora místa, která je vyjádřena jako:

$$\text{úspora místa} = 1 - \text{kompresní poměr}^{-1}. \quad (2.2)$$

Mějme například 2D obrázek o velikosti 256×256 pixelů, který zabírá 65536 bajtů. Obrázek zkomprimujeme a zabírá-li komprimovaná verze 16384 bajtů, můžeme říct, že kompresní poměr je 4 : 1 a úspora místa 75 %.

2.4 Míra informace v datech

Než se budeme moci věnovat jednotlivým kompresním metodám, připomeňme stručně myšlenky z teorie informace, které poskytly základ pro rozvoj metod bezeztrátové komprese. V této části předpokládám čtenářovu znalost základů teorie pravděpodobnosti a statistiky.

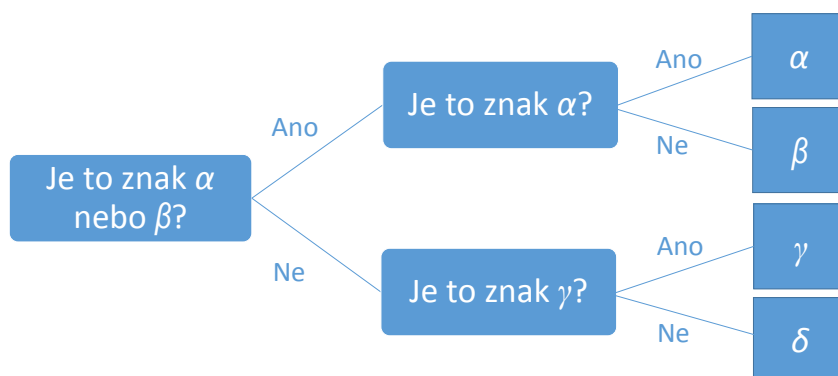
Informační entropie², jakožto míra neurčitosti v datech, nám pomůže lépe porozumět principům komprese. Představme si dva zdroje (označím je \mathbf{Z}_1 a \mathbf{Z}_2), které produkují zprávy složené ze symbolů (znaků) zdrojové abecedy $\mathbb{A} = \{\alpha, \beta, \gamma, \delta\}$. Oba zdroje generují znaky zprávy náhodně, pro zdroj \mathbf{Z}_1 však platí, že mají všechny znaky stejnou pravděpodobnost výskytu a to 0,25. Pro zdroj \mathbf{Z}_2 jsou pravděpodobnosti výskytu popsány v tabulce 2.1. Který ze zdrojů produkuje více informace?

Znak	Pravděpodobnost výskytu
α	$p_\alpha = 0,5$
β	$p_\beta = 0,3$
γ	$p_\gamma = 0,1$
δ	$p_\delta = 0,1$

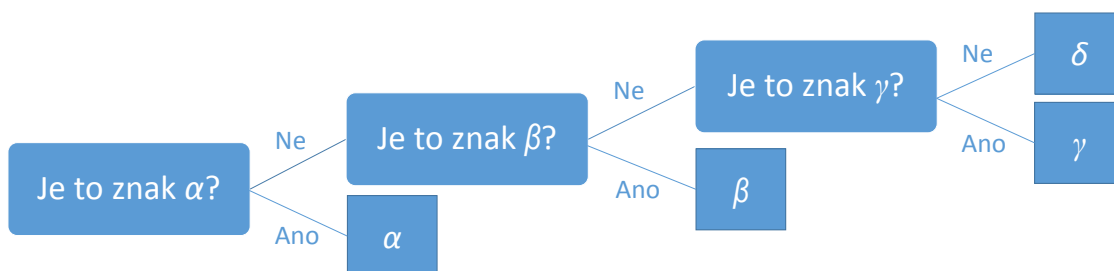
Tabulka 2.1: Pravděpodobnost výskytu znaků abecedy \mathbb{A} ve zdroji \mathbf{Z}_2

²Nazývaná též Shannonova po Claudu E. Shannonovi, který zformuloval klíčové poznatky.

Claude E. Shannon si tuto otázku položil následujícím způsobem. Jaký je nejmenší počet otázek s odpověďmi ano nebo ne, které musíme zodpovědět, abychom byli schopni rozhodnout, jaký je následující znak každého zdroje? Odpovědi na tuto zdánlivě složitou otázku je vyhledávací algoritmus binární vyhledávání. Nejeftivnější způsobem je v každém kroku položit takovou otázku, která rozdělí prohledávaný interval na dvě poloviny z hlediska pravděpodobnosti. Pro zobrazení, jaké otázky pokládat a v jakém pořadí, můžeme využít binární rozhodovací stromy (viz obrázky 2.2 pro zdroj \mathbf{Z}_1 a 2.3 pro zdroj \mathbf{Z}_2). Čteme je zleva doprava, v uzlech jsou otázky, hrany odpovídají odpovědím a v listech jsou příslušné znaky.



Obrázek 2.2: Binární rozhodovací strom pro zdroj \mathbf{Z}_1



Obrázek 2.3: Binární rozhodovací strom pro zdroj \mathbf{Z}_2

Průměrný počet otázek na zjištění jednoho znaku (tento počet označím \bar{q}) pokládaných tímto způsobem odpovídá pravděpodobnostem výskytu jednotlivých znaků a lze jej vypočítat jako $\bar{q} = \sum_{a \in \mathbb{A}} \#a \cdot p_a$, kde $\#a$ je počet otázek pro zjištění, že jde o znak a , a p_a je pravděpodobnost výskytu znaku a . Tedy pro zdroj \mathbf{Z}_1 dostaneme $\bar{q}_1 = 2p_\alpha + 2p_\beta + 2p_\gamma + 2p_\delta = 2 \cdot 0,25 + 2 \cdot 0,25 + 2 \cdot 0,25 + 2 \cdot 0,25 = 2$, tedy 2 otázky na znak, a pro zdroj \mathbf{Z}_2 dostaneme podobně $\bar{q}_2 = 1p_\alpha + 2p_\beta + 3p_\gamma + 3p_\delta = 1 \cdot 0,5 + 2 \cdot 0,3 + 3 \cdot 0,1 + 3 \cdot 0,1 = 1,7$, tedy 1,7 otázky na znak.

To znamená, že budeme-li se ptát na 1000 znaků u obou zdrojů, budeme muset položit 2000 otázek ke zjištění znaků ze zdroje \mathbf{Z}_1 a 1700 otázek ke zjištění znaků ze zdroje \mathbf{Z}_2 . Výstup zdroje \mathbf{Z}_2 obsahuje méně překvapení, nebo lépe neurčitosti, než výstup zdroje \mathbf{Z}_1 . Hovoříme o tom, že výstup zdroje \mathbf{Z}_2 poskytuje méně informace než výstup zdroje \mathbf{Z}_1 .

2.4.1 Entropie informačního zdroje

V Shannonově teorii je definice zdroje informace prakticky totožná s definicí pravděpodobnostního prostoru, tj. trojice $(\mathcal{X}, \mathcal{S}, p)$, kde \mathcal{X} je množina elementárních jevů (zpráv), \mathcal{S} je třída náhodných jevů (σ -algebra podmnožin \mathcal{X}) a p je pravděpodobnostní míra na \mathcal{S} . [10]

Pro nás bude modelem zdroje informace diskretní pravděpodobnostní prostor $(\mathcal{X}, p(x))$, který generuje zprávu X , jež je náhodnou veličinou s výběrovým prostorem $(\mathcal{X}, p(x))$. Mezi zprávou X a zdrojem $(\mathcal{X}, p(x))$ panuje ekvivalence, značíme $X \sim (\mathcal{X}, p(x))$. Základním pojmem teorie informace je entropie $H(X)$ zdroje X , jde o číslo, kterým charakterizujeme, jak obtížné je předpovědět hodnotu dané náhodné veličiny X , tj. zprávu, kterou vyprodukuje daný zdroj informace $X \sim (\mathcal{X}, p(x))$. Následuje formální definice entropie. [10]

Definice 2.1. Entropie zdroje informace $X \sim (\mathcal{X}, p(x))$ je veličina

$$H(X) = - \sum_{x \in \mathcal{X}} p(x) \log p(x),$$

kde $0 \log 0 = \lim_{t \rightarrow 0+} t \log t = 0$.

2.4.2 Entropie a komprese

Když chceme komprimovat digitální data, musíme je rozdělit na malé kousky, např. obrázek na pixely, textová data na znaky. To nám umožní zpracovat tato data jako posloupnost symbolů. Jednotlivé symboly pak můžeme reprezentovat s použitím nějakého kódu. Umíme-li zprávy vysílat a přijímat pouze v binární soustavě, vytvoříme kód³ složený ze znaků kódové abecedy $\mathbb{B} = \{0, 1\}$.

Vraťme se nyní zpět k abecedě \mathbb{A} a představme si zdroj \mathbf{Z} , který generuje zprávy složené ze znaků abecedy \mathbb{A} . Tento zdroj má pravděpodobnosti výskytu jednotlivých znaků odpovídající zdroji \mathbf{Z}_2 , což my zatím ale nevíme. Jako příklad můžeme mít za úkol komprimovat text obsahující 1000 znaků vygenerovaných zdrojem \mathbf{Z} . V takovém případě intuitivně přiřadíme jednotlivým zdrojovým znakům nejkratší možná kódová slova viz kód κ_1 v tabulce 2.2. Tato slova mají délku 2, což znamená, že potřebujeme průměrně 2 bity na znak.

Zdrojový znak	Kódové slovo
α	11
β	10
γ	01
δ	00

Tabulka 2.2: Kód κ_1 pro zdroj \mathbf{Z}_1

Pokud analýzou zdroje \mathbf{Z} ale zjistíme, že pravděpodobnosti výskytu odpovídají zdroji \mathbf{Z}_2 (viz tabulka 2.1), můžeme znakům přiřadit jiná kódová slova (viz tabulka 2.3), která budou brát ohled na nové pravděpodobnosti výskytu. Tento kód κ_2 je tzv. Huffmanův kód (viz kapitola 3 v [4]) a potřebuje průměrně 1,7 bitu na znak, způsob jeho vytvoření je popsán v části 3.2 nebo v [4].

³Takové binární kódy provází lidstvo již velice dlouho, např. kouřové signály, kódování znaků v Morseově abecedě.

Zdrojový znak	Kódové slovo
α	1
β	01
γ	000
δ	001

Tabulka 2.3: Kód κ_2 pro zdroj \mathbf{Z}_2

Podíváme-li se na obrázky 2.2 a 2.3 a příslušný výpočet průměrného počtu otázek na zjištění znaku, můžeme vidět jisté souvislosti. Například když slova Ano/Ne na hranách rozhodovacího stromu nahradíme čísly 1, resp. 0, dostaneme čtením od kořene k listům odpovídající kódová slova. Stejně tak průměrný počet otázek na zjištění znaku odpovídá střední délce zvoleného kódování (viz definice 7 v [4]).

K jaké úspoře dojde? K zakódování 1000 znaků výše zmíněného textu pomocí kódu κ_1 potřebujeme 2000 bitů, ale při použití kódu κ_2 nám bude stačit pouze 1700 bitů. Úspora je tedy 300 bitů. Na otázku, zda je možné tuto úsporu ještě zvýšit, nelze odpovědět jenom ano nebo ne. Pokud zkrátíme délku některých kódových slov v kódu κ_2 , stane se zakódovaná zpráva dvojnásobnou a zpětně nedekódovatelnou. Kódujme například znak δ kódovým slovem 0, potom sekvence 01 může znamenat jak $\delta\alpha$, tak i β . Abychom odstranili tento problém, museli bychom mezi kódová slova vložit oddělovač, čímž ale zakódovanou zprávu prodloužíme a k požadované úspoře nedojde.

Již Claude E. Shannon tvrdil, že komprese má limit a tím je vždy entropie zdrojové zprávy. Čím je entropie nižší, tím vyšší je možnost komprese. Naopak čím je entropie vyšší, kvůli nepředvídatelnosti, schopnost komprese se snižuje. Pokud bychom chtěli komprimovat až za tento limit, museli bychom nutně část informace vypustit, což je ale přesně princip ztrátové komprese, kterou si nemůžeme vždy dovolit.

Kapitola 3

Statistické techniky komprese

Statistické kompresní metody využívají znalosti tzv. pravděpodobnostního modelu komprimovaných dat. Modelem může být například četnost výskytu jednotlivých symbolů v textu. Účinnost komprese je závislá na schopnosti co nejpřesněji modelovat zpracovávaná data. Čím více se model přibližuje realitě, tím je účinnost komprese vyšší. A naopak.

3.1 Pravděpodobnostní model

Pravděpodobnostní model přiřazuje pravděpodobnosti výskytu symbolům ve zdrojové zprávě. Na tomto základě například Huffmanovo kódování (viz 3.2) přiřazuje symbolům, které se v datech objevují častěji (mají vyšší pravděpodobnost výskytu), kratší kódová slova.

Podle toho, jakým způsobem model vzniká, ho můžeme označit jako statický, semiadaptivní, nebo adaptivní.

3.1.1 Statický model

Tento model je definován při implementaci kompresní metody a při použití se již nijak nemění. Výhodou je rychlost a jednoduchost vytvoření modelu. Například v případě, že budeme zpracovávat výstup pouze jednoho známého zdroje, se tímto způsobem vyhneme opakovanému vytváření modelu, které by vedlo ke stejným, nebo alespoň velmi podobným výsledkům. V opačném případě ale model nemusí datům vůbec odpovídat, čímž bude dosažená účinnost komprese velice nízká.

3.1.2 Semiadaptivní model

Komprimovaná data jsou jedním průchodem statisticky zpracována a následně je vytvořen odpovídající model. K samotné kompresi je ale nutný ještě druhý průchod daty s již vytvořeným modelem. Aby bylo možné data dekomprimovat, je nutné připojit model k datům. Model je sice přesný, ale velikost zkomprimované zprávy je větší o model. Navíc přístup s dvěma průchody daty činí tyto algoritmy neefektivními a v praxi se moc nepoužívají.

3.1.3 Adaptivní model

Tento model vzniká v průběhu zpracování dat a je postupně aktualizován. To platí i pro dekompresní algoritmus, ten vytváří model stejně jako kompresní algoritmus z části textu, kterou již dekomprimoval. Díky tomu není nutné model připojovat k datům a také procházet data dvakrát.

3.2 Semiadaptivní Huffmanovo kódování

Huffmanovo kódování je jednou z nejstarších kompresních technik, kterou publikoval již v roce 1952 David A. Huffman. Semiadaptivní dvouprůchodová verze byla od svého vzniku podrobena dalšímu výzkumu a postupně vylepšena až na adaptivní. Je založeno na dvou vlastnostech nejkratších prefixových (jednoznačně dekódovatelných) kódů [6]:

1. Symbolům s vyšší pravděpodobností výskytu jsou přiřazována kratší kódová slova.
2. Dvěma symbolům s nejmenší pravděpodobností výskytu jsou přiřazována kódová slova stejné délky.

Algoritmus nejprve seřadí symboly sestupně podle pravděpodobnosti výskytu a poté z nich v krocích konstruuje strom od listů ke kořeni. V každém kroku jsou vybrány dva symboly s nejmenší pravděpodobností výskytu, z nich je vytvořen nový uzel, který je brán jako symbol s kumulovanou pravděpodobností odpovídající součtu dílčích pravděpodobností. Kroky opakuje až do vybrání všech symbolů, kdy je kumulovaná pravděpodobnost výskytu rovna 1. Nakonec ohodnotíme hrany vystupující z uzlu vlevo číslem 0 a vpravo číslem 1. Kódová slova získáme čtením stromu od kořene k listům. [5], [6]

3.2.1 Postup kódování a dekódování zprávy

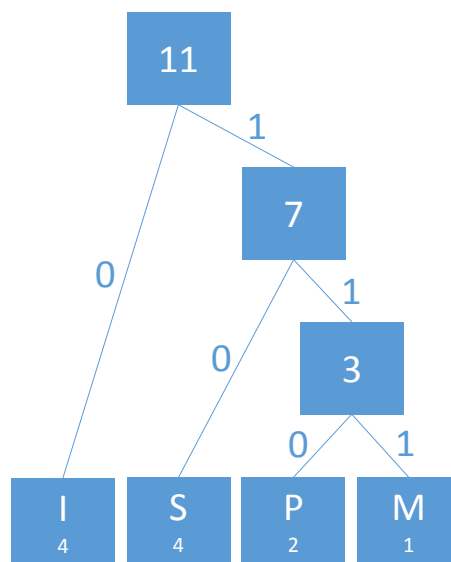
Při kódování zprávy zapisujeme na výstup kódová slova odpovídající čteným symbolům. Naopak při dekódování procházíme strom od uzlu k listům tak, jak jsou čteny jednotlivé znaky kódových slov. Pokud dojdeme do listu, přečetli jsme jeden původní symbol a můžeme jej vypsát.

3.2.2 Vzorový příklad

Postup lze nejlépe prezentovat na příkladu. Mějme za úkol zakódovat zprávu „MISSIS-SIPPI“ pomocí semiadaptivního Huffmanova kódování. Výsledný strom je zobrazen na obrázku 3.1, celá konstrukce pak v příloze [reference na konstrukci stromu v příloze](#). Místo pravděpodobností výskytu jsou použity absolutní četnosti symbolů v listech a kumulativní četnosti v dalších uzlech. Získaná kódová slova pro jednotlivé symboly jsou zobrazena v tabulce 3.1.

Data po zakódování

Zakódovaná zpráva je tvaru 111|0|10|10|0|10|10|0|110|110|0, kde je znak „|“ použit jako oddělovač jednotlivých kódových slov.



Obrázek 3.1: Strom vytvořený při semiadaptivním Huffmanově kódování

Zdrojový znak	Četnost výskytu	Kódové slovo
I	4	0
S	4	10
P	2	110
M	1	111

Tabulka 3.1: Kódová slova semiadaptivního Huffmanova kódu

3.3 Adaptivní Huffmanovo kódování

Adaptivní Huffmanovo kódování konstruuje strom při kompresi i dekompresi posloupností stejných kroků, tzv. mirroring. Strom je v každém kroku upraven tak, aby produkoval nejkratší kód pro do té doby zpracovanou část dat. Jak se strom mění, mění se i kódy přiřazené jednotlivým symbolům. Listy stromu obsahují symboly a jejich četnosti výskytu, vnitřní uzly obsahují kumulovanou četnost svých potomků.

Implementovaných verzí existuje několik. Níže je představena verze, kterou navrhl Jeffrey S. Vitter a ve které všechny uzly navíc obsahují pořadové číslo, které je využito při rozhodování při aktualizaci stromu. Nejvyšší pořadové číslo n určíme ze vztahu $n = 2m - 1$, kde m je počet znaků zdrojové abecedy. Také se zde využívají tzv. bloky – množiny uzlů se stejnou četností. Diagram aktualizace stromu po přečtení jednoho symbolu je možné vidět v příloze [odkaz](#), způsob kódování a dekódování zprávy je popsán v části 3.3.1.

Algoritmus začíná pouze s uzlem označeným NYT (not yet transmitted), který symbolizuje všechny znaky, které se zatím ve stromu nevyskytují. Je-li ke zpracování načten symbol, který se ještě ve stromě nevyskytuje, je do výstupu vypsáno kódové slovo uzlu NYT následované nezakódovaným a dohodnutým tvarem zpracovávaného symbolu. Následně se z uzlu NYT stane vnitřní uzel a jsou mu přiřazeni dva noví potomci: nový uzel NYT a list reprezentující právě zpracovávaný znak. Poté je strom aktualizován. Je-li ke zpracování načten ve stromě již obsažený symbol, je na výstup vypsáno pouze kódové slovo tohoto symbolu a strom je aktualizován.

Nezakódovaný tvar zpracovávaného symbolu může být například jeho osmibitový ASCII kód. Má-li zdrojová abeceda $\{a_1, a_2, \dots, a_m\}$ počet znaků roven m , pak zvolíme e, r tak, aby splňovala $m = 2^e + r$ a $0 \leq r < 2^e$. Znak a_k poté kódujeme jako $(e + 1)$ -bitovou reprezentaci čísla $k - 1$, je-li $1 \leq k \leq 2r$, jinak je a_k kódován jako e -bitová reprezentace čísla $k - r - 1$. [5], [6]

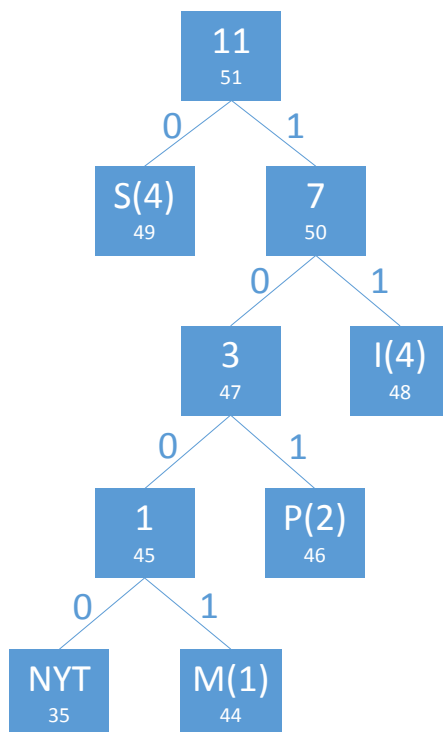
3.3.1 Vzorový příklad

Pokusme se znovu zakódovat slovo MISSISSIPPI, kde budeme uvažovat anglickou abecedu jako zdrojovou. Ta má 26 znaků, tedy $m = 26$, $e = 4$ a $r = 10$ a znaky číslujeme $a_1 = A$, $a_2 = B$ atd.

Zakódování zprávy a vytvoření stromu

Začínáme se stromem, který obsahuje pouze kořen NYT. Čteme znak M a na výstup zapíšeme jeho pětibitový nezakódovaný tvar pro $k = 13$, tedy 01100. Aktualizujeme strom. Čteme znak I, který se ve stromě také nevyskytuje. Na výstup zapíšeme kód uzlu NYT (nyní je to 0) a pětibitový nezakódovaný tvar pro $k = 9$, tedy 01000. Aktualizujeme strom. Dále čteme nový znak S. Na výstup zapíšeme kód uzlu NYT (nyní je to 00) a pětibitový nezakódovaný tvar pro $k = 19$, tedy 10010. Aktualizujeme strom. Dále čteme znak S, který ve stromu již je a tedy vypíšeme pouze jeho kód. Aktualizujeme strom. A tak pokračujeme pro všechny znaky slova.

Výsledný strom je zobrazen na obrázku 3.2, celý postup vytvoření stromu pak v příloze [odkaz](#).



Obrázek 3.2: Strom vytvořený při adaptivním Huffmanově kódování

Zakódovaná zpráva

Výsledná zakódovaná zpráva je tvaru 01100|0|01000|00|10010|101|11|0|1|01|000|01111|1001|11, kde jsem pro přehlednost vložil oddělovač „|“ oddělující jednotlivé významné bloky.

Dekódování zprávy

Přečteme $e = 4$ bitů znaků, jejichž hodnota jakožto čtyřbitového čísla je 6, což je menší než $r = 10$. Přečteme další znak, hodnota tohoto pětibitového čísla je 12 a tedy jsme přečetli symbol $a_{13} = M$. Aktualizujeme strom. Následuje 0, která je kódovým slovem pro uzel NYT, po něm vždy následuje nezakódovaný symbol. Obdobně jako pro M přečteme symbol $a_8 = I$. Aktualizujeme strom. Následuje kódové slovo uzlu NYT a symbol $a_{19} = S$. Dále čteme kódové slovo uzlu pro symbol S. Takto pokračujeme dokud nepřečteme všechna slova zakódované zprávy.

Kapitola 4

Slovníkové techniky komprese

V předchozí kapitole jsme si představili kompresní techniky, které předpokládají zdroje generující posloupnosti nezávislých symbolů a snaží se vytvořit statistický model dat. Účinnost komprese pak závisí na přesnosti modelu. Slovníkové techniky, které popíšu v této kapitole, nepoužívají statistický model ani kódy s různou délkou kódových slov. Místo toho vyhledávají opakující se části textu, tzv. fráze, ty udržují ve slovníku a na výstup vypisují pouze odkaz na příslušné místo ve slovníku. Slovníkem zde může být jak část již zpracovaných dat, tak i vlastní struktura v paměti. Podobně jako u statistických modelů existují statické a adaptivní slovníky.

4.1 Základy slovníkové komprese

Základní princip a problematiku slovníkové komprese si můžeme představit na jednoduchém příkladu. Mějme text složený ze slov délky 5, kde jsou slova libovolně složena z arabských číslic a písmen A–F. Takováto abeceda má tedy 16 znaků. Pokud budeme předpokládat, že mají všechny znaky stejnou pravděpodobnost výskytu, budeme pro vytvoření binárního kódu potřebovat 4 bity na znak a 20 bitů k zakódování každého slova (těchto slov je 2^{20}). Nyní vytvoříme statický slovník, který bude obsahovat 256 nejčastěji se vyskytujících slov.

Při kompresi budeme na výstup zapisovat jeden bit charakterizující, zda je, či není zpracovávané slovo obsaženo ve slovníku (např. 1 a 0) následovaný osmibitovým indexem slova (pokud je ve slovníku), nebo dvacetibitovým kódem slova (pokud ve slovníku není). To znamená, že pro slova ve slovníku potřebujeme pouze 9 bitů, zatímco pro slova, která ve slovníku nejsou, potřebujeme bitů 21.

Pro to, aby byl takovýto způsob komprese účinný, je rozhodující pravděpodobnost, že je zpracovávané slovo ve slovníku. To lze vyjádřit pomocí vzorce $R = Sp + T(1 - p)$, kde R je průměrný počet bitů na slovo, S počet bitů pro slovo ve slovníku, T počet bitů pro slovo mimo slovník a p je pravděpodobnost, že je zpracovávané slovo ve slovníku. V našem případě musí být $R < 20$, což je splněno pro $p \geq 0,083$.

Pro pravděpodobnost p blízkou číslu 0,083 je účinnost komprese nízká a tedy nezajímavá. Pro vysokou účinnost potřebujeme pravděpodobnost p co možná největší. Toho lze dosáhnout zvětšením slovníku, což s sebou nese vyšší paměťové nároky a časově náročnější prohledávání, nebo pečlivým výběrem slov ve slovníku.

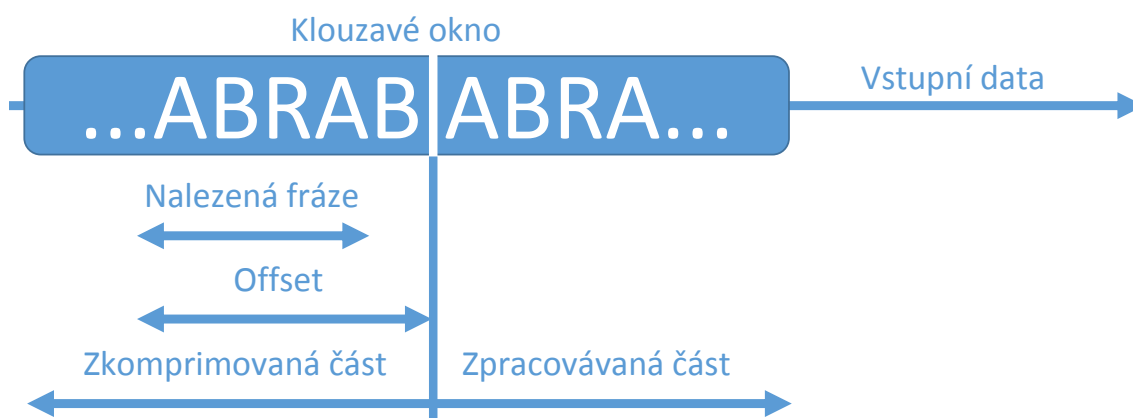
4.1.1 Typy slovníků

V praxi jsou známy dva typy slovníků: statický a adaptivní. Podobně jako v případě statistických technik (viz 3.1) komprese je volba kompresního algoritmu se statickým slovníkem nejvhodnější, pokud známe dobře zdroj a komprimujeme zprávy výhradně tohoto zdroje. Proti tomu adaptivní slovník lze použít na neznámá data.

Většina kompresních algoritmů s adaptivním slovníkem je založena na pracích pánů Abrahama Lempela a Jacoba Ziva, kteří v letech 1977 a 1978 představili dva různé přístupy k vytváření slovníku. Podle nich jsou vzniklé algoritmy označovány jako algoritmy rodiny LZ77, resp. LZ78.

4.2 LZ77

Princip tohoto algoritmu je založen na okně klouzajícím po zpracovávaných datech. Toto okno se dělí na dvě podokna (viz obrázek 4.1). V prvním je část ještě nezpracovaných dat. Ve druhém je pak část již zkomprimovaných dat, ve které jsou vyhledávány fráze a tedy funguje jako slovník. Velikost tohoto podokna je pevně daná jakožto kompromis mezi co největší velikostí, která zvyšuje pravděpodobnost nalezení fráze, a co nejmenší velikostí, která snižuje dobu vyhledávání fráze.



Obrázek 4.1: Schéma algoritmu LZ77, jenž využívá klouzavé okno

Tento algoritmus má velmi dobrý kompresní poměr pro data, ve kterých se shodné fráze vyskytují blízko sebe. To je dáno tím, že obsahem slovníku jsou poslední zpracovaná data, která se mění s posunem okna.

4.2.1 Princip komprese

V každém kroku kompresního algoritmu se v podokně zkomprimované části hledá nejdelší fráze. Znaků jsou procházeny zprava doleva až do shody prvního znaku fráze, kdy je zaznamenána délka fráze a vzdálenost fráze od pravého kraje podokna (offset), poté se pokračuje ve vyhledávání delší fráze dále vlevo. Nejdelší nalezená fráze je zakódována pomocí trojice (offset, délka fráze, následující znak (ve zpracovávaných datech)) a okno se posune o (délka fráze + 1) vpravo. Není-li nalezena shoda, je zakódován první znak ve zpracovávané části jako (0, 0, první znak). Tím je zaručeno, že se okno posune v každém

kroku alespoň o jeden znak.

4.2.2 Příklad komprese

Zkusme zakódovat část textu začínajícího znaky ABRAKADABRAK s délkou podokna slovníku 7 znaků. V prvním kroku zakódujeme znak A, protože část slovníku je zatím prázdná, a posuneme okno o jeden znak doprava. Podobně zakódujeme i následující dva znaky B a R. Kódování jednotlivých znaků s offsetem 0 a délkou 0 je typickým projevem na začátku kompresního algoritmu, kdy je slovník ještě prázdný. V kroku 4 nalezneme frázi A délky 1 s offsetem 3 a posuneme okno o 2 znaky doprava. V dalším kroku úplně stejně zakódujeme další frázi A. V posledním kroku pak nalezneme frázi ABRA délky 4 a offsetem 7, kterou zakódujeme jako (7,4,K). Celý postup je zobrazen v tabulce 4.1.

Krok	Text	Fráze	Kódování
1	<div>ABRAKADABRAK...</div>		(0,0,A)
2	<div>A BRAKADABRAK...</div>		(0,0,B)
3	<div>AB RAKADABRAK...</div>		(0,0,R)
4	<div>ABR AKADABRAK...</div>	A	(3,1,K)
5	<div>ABRAK ADABRAK...</div>	A	(2,1,D)
6	<div>ABRAKADABRAK...</div>	ABRA	(7,4,K)

Tabulka 4.1: Kódování textu pomocí algoritmu LZ77

4.2.3 Princip dekomprese

Postup dekomprese je mnohem jednodušší a rychlejší než kompresní algoritmus. V paměti si udržuje pouze data slovníku, ve kterém jednoduše vyhledává fráze dle postupně dekódovaných trojic. Offset a délka fráze určují frázi jednoznačně a není tedy nutné prohledávat celý slovník. LZ77 je tedy asymetrický kompresní algoritmus.

4.2.4 Variace algoritmu LZ77

Vylepšených verzí algoritmu LZ77 vzniklo několik a lze je rozdělit do skupin dle způsobu zlepšení. První skupina je zaměřena na efektivní kódování výstupu, tj. výše popsaných trojic, pomocí kódu s proměnlivou délkou slov. Reprezentantem takového kódování jsou semiadaptivní a adaptivní Huffmanovo kódování (viz 3.2 a 3.3). Další skupina pracuje s proměnlivou velikostí klouzavého okna, což pro případ jeho zvětšení nese požadavek na implementaci efektivnějších vyhledávacích algoritmů.

Velmi známá verze je algoritmus LZSS, který odstraňuje neefektivní chování původního algoritmu, kdy je v jednom kroku kódován i následující znak, kterým může začínat fráze existující ve slovníku. Místo toho jsou nalezené fráze kódovány pomocí trojice (příznak,

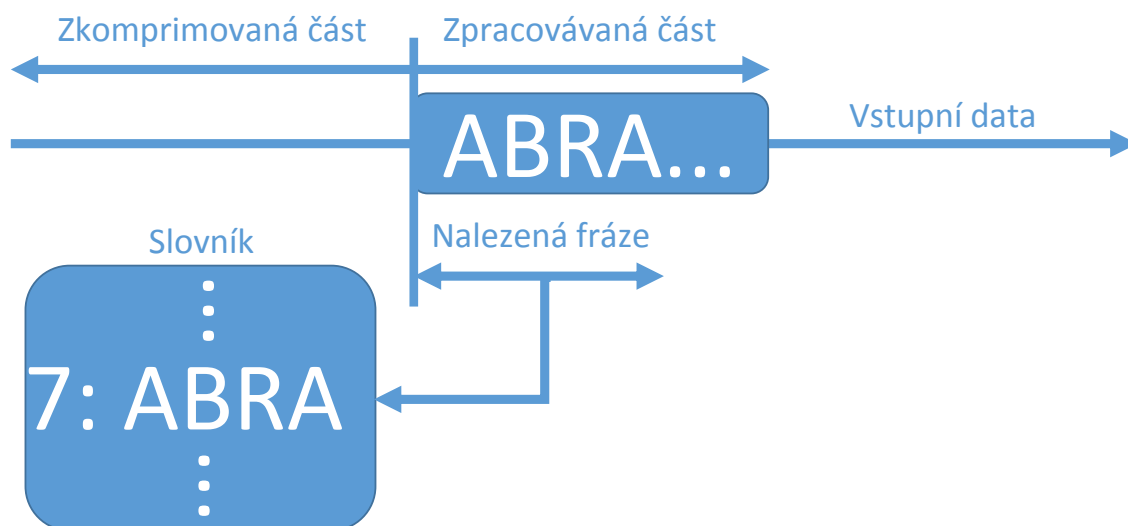
offset, délka fráze) a samotné znaky, které nejsou nalezeny ve slovníku, pomocí dvojice (příznak, znak).

4.3 LZ78

Algoritmus je založen na vyhledávání opakujících se frází ve vlastní paměťové struktuře. Tento slovník je na začátku procesu prázdný a postupně jsou do něho při zpracování na indexované pozice vkládány jednotlivé fráze. Jednoduché schéma je zobrazeno na obrázku 4.2. Podobně jako v případě LZ77 je velikost slovníku dána kompromisem mezi počtem uložených frází a rychlostí vyhledávání.

V případě, že je slovník zcela naplněn, algoritmus LZ78 neříká přesně, co v takové situaci udělat. Existuje ale několik možností, které se v praxi používají a jsou vhodné pro různé případy.

- Dále využívat slovník jako statický. To je vhodné v případě, že se stejné fráze opakují v celé šířce dat.
- Slovník úplně vymazat a začít ho plnit znovu. Takový přístup lze s výhodou aplikovat na data, ve kterých lze nalézt velké ucelené bloky, ve kterých se opakují různé fráze. Zároveň tento přístup připomíná princip algoritmu LZ77, kdy je jako slovník použita část blízkých předcházejících dat.
- Ze slovníku mazat fráze s nízkým výskytem. Takto adaptivní slovník by byl vhodný univerzálně, nicméně není znám algoritmus, který by dokázal určitě kolik a kterých frází má být smazáno.



Obrázek 4.2: Schéma algoritmu LZ78, jenž využívá vlastní slovník

Vhodnou paměťovou strukturou pro uchovávání slovníku je strom, jehož uzly mohly mít tolik potomků, kolik je znaků zdrojové abecedy. V kořeni je prázdné slovo a při vyhledávání fráze se každým načteným znakem posuneme do potomka v příslušné větvi (pokud existuje).

4.3.1 Princip komprese

Na počátku je ve slovníku na pozici 0 pouze prázdné slovo¹ ε . Při zpracování dat jsou na další pozice 1,2, atd. vkládány nové fráze. Je-li na vstupu přečten znak s , algoritmus se ve slovníku pokusí nalézt jednoznakovou frázi s . Pokud není tato fráze nalezena, je vložena na volnou pozici s nejnižším indexem a na výstup je zapsána dvojice $(0, s)$, kde 0 je index prázdného slova. Tato dvojice znamená zakódování řetězce εs (zřetězení prázdného slova a znaku s). Naopak je-li fráze s ve slovníku nalezena, je ze vstupu přečten další znak t a ve slovníku je hledána dvouznaková fráze st . Pokud je fráze nalezena, pokračuje načítání dalších znaků a hledání ve slovníku. Pokud fráze nalezena není, je na výstup zapsána dvojice (i, t) , kde i je index prefixové² fráze s . Tento proces pokračuje, dokud nejsou zpracována všechna data na vstupu.

4.3.2 Příklad komprese

Zkusme pro příklad zakódovat část textu začínající slovem ABRAKADABRAK. Stav slovníku a výstup po kompresi můžeme vidět v tabulce 4.2. V prvních třech krocích (indexy 1,2 a 3) jsou komprimovány pouze jednoznakové fráze, v dalších krocích již existují fráze A, resp. AB a tedy jich můžeme využít k zakódování delších frází.

Slovník		Výstup
Index	Fráze	
0	ε	
1	A	(0, A)
2	B	(0, B)
3	R	(0, R)
4	AB	(1, B)
5	ABR	(4, R)
6	AK	(1, K)

Tabulka 4.2: Kódování textu pomocí algoritmu LZ78

4.3.3 Princip dekomprese

Při rekonstrukci dat jsou čteny jednotlivé dvojice (index, znak) a slovník je vytvářen pomocí úplně stejných kroků jako v případě komprese. Je tedy složitější než dekompresní algoritmus LZ77. Platí, že je-li nějaká fráze ve slovníku, pak jsou v něm i všechny její prefixy. Ty jsou nutné pro hledání delších frází, neboť kompresní algoritmus se odkazuje na prefixové fráze o jeden znak kratší.

4.3.4 Variace algoritmu LZ78

Podobně jako v případě algoritmu LZ77 vzniklo i množství variant postavených na myšlence algoritmu LZ78 upravujících ty části, které považují autoři za neefektivní. Mezi nejvýznamnější se řadí metoda LZW vycházející z předpokladu, že kódovat jednotlivé znaky pomocí

¹Prázdná posloupnost znaků, slovo délky 0.

²Prefix značí několik shodných počátečních znaků různých řetězců.

výše zmíněných dvojic je neefektivní. Místo toho na počátku kompresního procesu inicializuje slovník všemi jednoznakovými frázemi znaků zdrojové abecedy. Je-li hledaná fráze nalezena, pokračuje načtením dalšího znaku a hledáním delší fráze. Naopak není-li fráze nalezena, je vložena do slovníku, ale na výstup kóduje pouze index o jeden znak kratší fráze nalezené ve slovníku (nalezení jednoznakové fráze je zaručeno díky původní inicializaci slovníku). Další hledaná fráze začíná posledním zpracovaným znakem, tj. poslední znak fráze uložené do slovníku v předchozím kroku.

Kapitola 5

Existující algoritmy pro kompresi XML

Velikost XML dokumentu danou vlastností formátu, kdy se schéma opakuje pro každý záznam, lze při kompresi redukovat pomocí například gzip¹. Jiným přístupem může být navržení kompresního algoritmu speciálně pro XML. Algoritmy využívající znalost vnitřní struktury XML se anglicky nazývají XML aware. Díky této znalosti dokážou data předpřipravit tak, aby byla komprese co nejefektivnější. Pro kompresi samotnou jsou používány algoritmy již zmiňované v kapitolách 3 a 4 a jejich upravené varianty.

XML aware algoritmy můžeme rozdělit dle několika kritérií a to zda podporují dotazování a přístup do komprimovaných dat a nebo zda je nebo není dekompresní algoritmus závislý na přístupu k XML schématu. Z druhé skupiny je častější použití algoritmů, které jsou na schématu nezávislé, protože pro dekompresi není vždy možné přístup ke schématu zaručit.

5.1 XMill

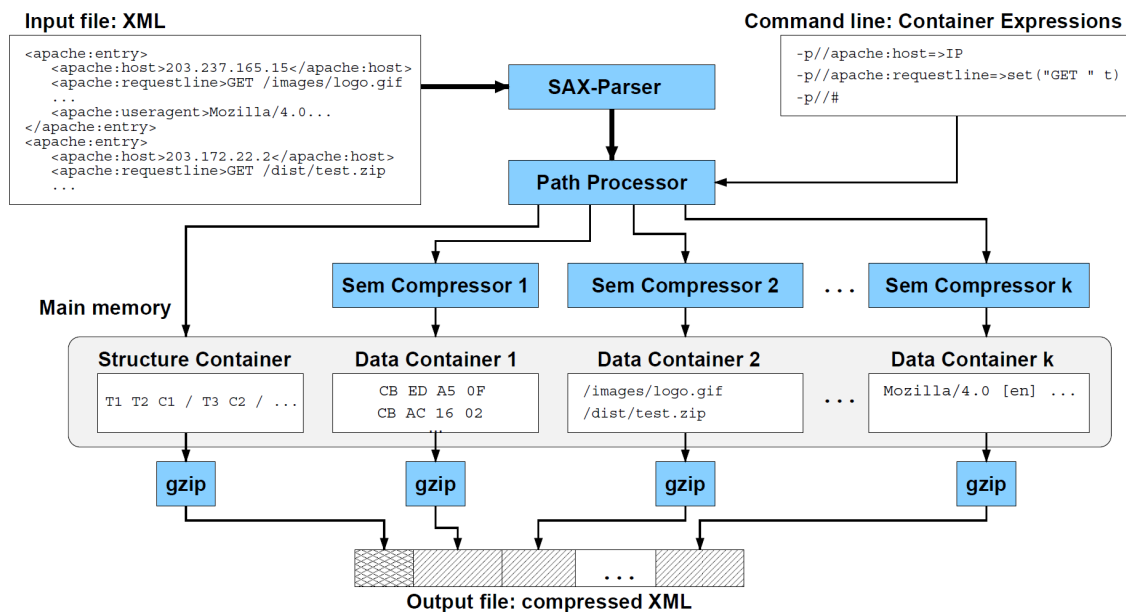
Algoritmus XMill, který nepodporuje dotazování do zkomprimovaných dat, představili pánové Hartmut Liefke a Dan Suciú v roce 2000. Jeho architektura využívá knihovnu kompresních algoritmů zlib, specifické kompresory pro určitý typ dat a navíc podporuje použití kompresorů vytvořených uživatelem pro speciální typy dat. Rozhodnutí, který kompresní algoritmus bude použit, je provedeno na základě znalosti tagů. Architektura algoritmu XMill je postavena na třech základních principech [3]:

- Struktura XML složená z tagů a atributů tvoří strom. Data, obsah elementů a hodnoty atributů, jsou reprezentovány jako řetězce. Stromová struktura a data jsou komprimovány odděleně.
- Datové položky stejných elementů jsou seskupeny do jednoho kontejneru a každý kontejner je komprimován odděleně.
- Dle typu dat (text, číslo apod.) je pro kompresi kontejneru použit vhodný kompresní algoritmus, tzv. sémantický kompresor.

¹GNU zip, software pro kompresi dat využívající LZ77 a Huffmanovo kódování.

5.1.1 Popis architektury

Architektura algoritmu XMill je zobrazena na obrázku 5.1. XML dokument je nejprve zpracován pomocí SAX² parseru, který každý prvek pošle do Path procesoru. Path procesor rozhodne, do kterého existujícího kontejneru prvek vloží, nebo zda má vytvořit kontejner nový. Ke každému kontejneru může být uživatelem přiřazen sémantický kompresor a to buď atomický implementovaný v XMill, kombinace atomických pro komplexnější typy a nebo vlastní. Kontejnery jsou plněny až do předem určené velikosti (defaultně je to 8 MB), je-li tato velikost dosažena, je kontejner zkomprimován pomocí gzip, uložen na disk a komprese dále pokračuje. Díky tomu jsou data rozdělena do vzájemně nezávislých bloků.



Obrázek 5.1: Architektura algoritmu XMill [3]

5.1.2 Oddělení struktury od dat

Struktura XML je složena z tagů a atributů. Počáteční tagy a atributy jsou kódovány slovníkovou metodou a nahrazeny indexem ve slovníku, kladným celým číslem. Ukončující tag je kódován hodnotou 0, při dekompresi je díky syntaxi zřejmé, který tag je ukončován. Datové hodnoty jsou nahrazeny záporným indexem kontejneru, do kterého byly přiřazeny. [3]

Pro příklad zpracujeme následující informace o článku:

```
<article mdate="2011-01-11" key="journals/acta/GoodmanS83">
  <author>
    <name>Nathan Goodman</name>
    <title>Mr.</title>
  </author>
  <title>NP-complete Problems Simplified on Tree Schemas.</title>
</article>
```

²Simple API for XML.

Tagy zde jsou `article`, `author` a `title` a atributy `mdate` a `key`. Po zpracování dostaneme výstup 1 2 -3 3 -4 0 4 5 -5 0 6 -6 0 0 7 -7 0 0. Lze si povšimnout, že indexování kontejnerů začíná až číslem 3, to proto, že index 0 je rezervován pro kontejner struktury, 1 pro kontejner bílých znaků (umožňuje zachování formátovacích mezer) a 2 pro kontejner obsahující procesní instrukce, DTD a jiné.

5.1.3 Seskupení dat stejného významu

Mapování mezi daty a kontejnery provádí Path procesor na základě cesty k datům (path) a na uživatelem definovaném regulárním výrazu container expression. Základní myšlenkou je hodnotám pro každý tag nebo atribut přiřadit vlastní kontejner. Cesta k datům v příkladě z předchozí části 5.1.2 vypadá například `/article/@key` pro atribut a `/article/author/name` pro tag. Pomocí container expression je možné nastavit pravidla, kdy budou například hodnoty různých tagů ukládány do stejného kontejneru. [3]

5.1.4 Sémantická komprese

Jak již bylo zmíněno v úvodu popisu tohoto algoritmu, jsou podporovány 3 druhy sémantických kompresorů: atomické, kombinované a definované uživatelem. Atomických kompresorů je 8, například textový kompresor `t`, který řetězec pouze zkopíruje na výstup (ten bude později zkomprimován pomocí `gzip`), nebo kompresor `u`, který komprimuje kladná celá čísla.

Kombinovaný kompresor je vhodný pro hodnoty, ve kterých lze nalézt nějaký vzor nebo strukturu. Jako příklad lze uvést sekvenční kompresor pro kompresi IP adresy `seq(u8 ".u8 ".u8 ".u8)`, což znamená sekvenci čtyř kladných celých čísel menších než 256 oddělených tečkou.

Vlastní kompresory lze s výhodou použít na data, která mají strukturu složitou nebo velmi těžce vyjádřitelnou pomocí kombinovaných kompresorů. Autoři jako příklad takových dat uvádějí DNA sekvence. Aby bylo možné data komprimovaná za pomoci vlastních kompresorů zpětně rekonstruovat, musí mít dekompresní algoritmus samozřejmě také přístup k použitému vlastnímu kompresoru. [3]

5.1.5 Nedostatky tohoto přístupu

Způsob rozdělení struktury a dat různých typů zvláště do jednotlivých bloků v algoritmu XMill brání úpravě komprimovaných dat nebo efektivnímu dotazování nad nimi. Možnost dotazů nad daty je ale pro některé aplikace velmi důležitá a v případě algoritmů typu LZ nebo XMill by to znamenalo zbytečně náročnou operaci: dekomprimovat data, provést dotazy nebo úpravy a data zpět zkomprimovat.

5.2 XGrind

Na základě výše zmíněných požadavků publikovali pánové Pankaj M. Tolani a Jayant R. Haritsa v roce 2002 článek, ve kterém představili nový kompresní nástroj, který přímo podporuje dotazování nad zkomprimovanými daty. Ten, namísto rozdělení struktury celého dokumentu, komprimuje obsah jednotlivých elementů a atributů pomocí semiadaptivního Huffmanova kódování (viz 3.2), které je nezávislé na kontextu.

Díky tomu podporuje dotazy typu úplná shoda nebo shoda prefixu tak, že je dotaz zakódován pomocí stromu vytvořeného při kompresi, následně jsou ve zkomprimovaném souboru vyhledána data odpovídající dotazu a pouze ta jsou dekomprimována a zobrazena uživateli. To odpovídá nejmenšímu možnému objemu dat, který je nutný dekomprimovat. Algoritmus navíc podporuje i intervalové dotazy a dotazy na podřetězec, ty ale vyžadují částečnou dekompresi příslušných elementů či atributů. [9]

5.2.1 Kompresní techniky

Způsob kódování tagů a atributů je velice podobný jako v případě algoritmu XMill (viz 5.1.2). Každý počáteční tag je kódován jako zřetězení T a unikátního indexu tagu. Podobně jsou atributy kódovány zřetězením A a unikátního indexu atributu. Koncové tagy jsou kódovány symbolem $/$, což je možné díky syntaxi. Výčtové typy, jejichž definice je uvedena v DTD nebo ve schématu dokumentu, jsou kódovány pomocí schématu $\log_2 n$, kde n je počet prvků výčtového typu.

Aby bylo možné se efektivně dotazovat do komprimovaných dat, je pro kompresi hodnot elementů a atributů použito kompresní schéma, které řetězci přiřadí kód nezávislý na místě výskytu v dokumentu. To splňuje použité semiadaptivní Huffmanovo kódování (viz 3.2), které ve dvou průchodech nejprve vytvoří kódy na základě statistiky a poté data zakóduje. V algoritmu XGrind nejsou kódovány jednotlivé znaky, ale celé hodnoty, které spolu sémanticky souvisejí. [9]

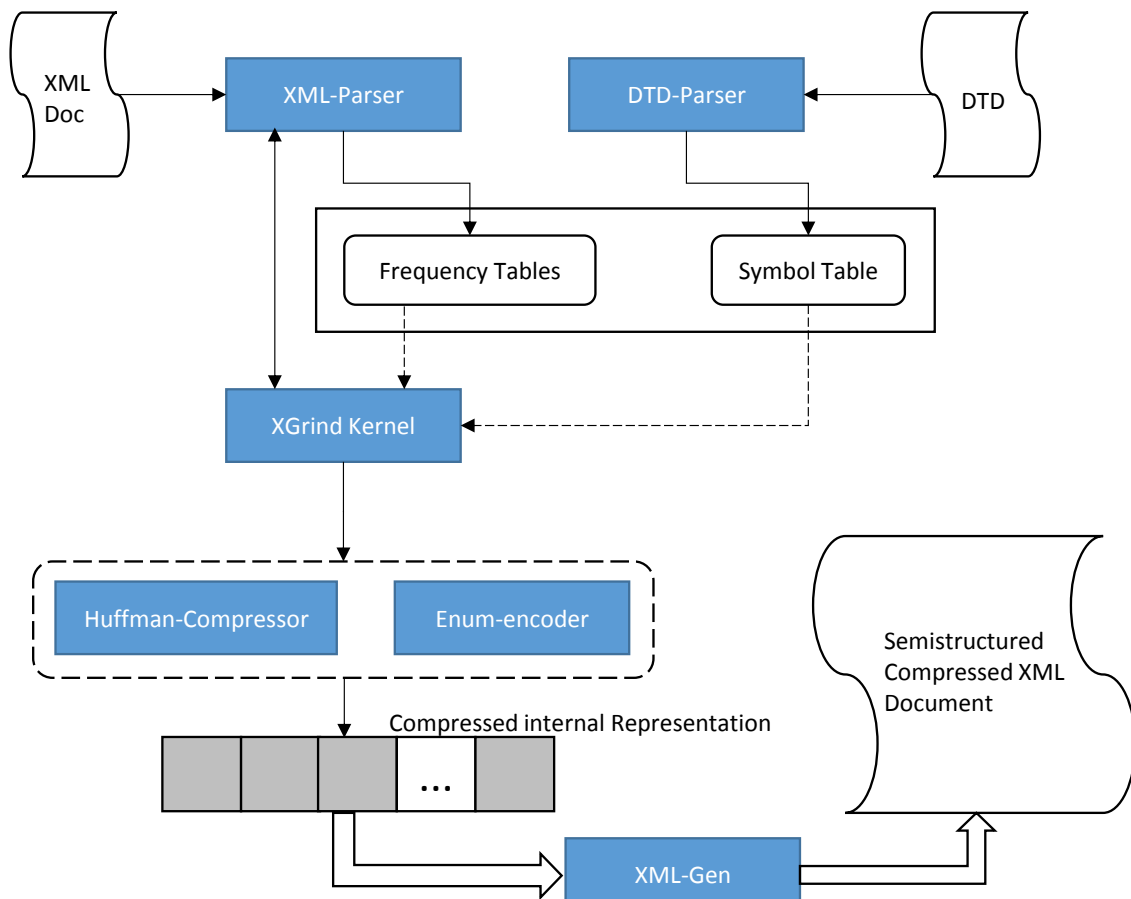
Zakódujeme nyní zkrácenou verzi příkladu k algoritmu XMill:

```
<article mdate="2011-01-11">
  <author>
    <name>Nathan Goodman</name>
  </author>
  <title>NP-complete Problems</title>
</article>
```

Výstupem bude sekvence $T0 \ A0 \ huff(2011-01-11) \ T1 \ T2 \ huff(Nathan\ Goodman) \ / \ / \ T3 \ huff(NP-complete\ Problems) \ / \ /$, kde je výrazem $huff(A)$ myšlen semiadaptivní Huffmanův kód řetězce A .

5.2.2 Popis architektury

Schéma architektury lze vidět na obrázku 5.2, jehož originální verze byla publikována v [9]. Hlavní prvkem je XGrind Kernel, který nejprve vyzve DTD-Parser k načtení DTD a inicializaci Frequency Tables naplněním všemi unikátními tagy a nevýčtovými atributy a Symbol Table naplněním výčtovými atributy. Následně XGrind Kernel vyzve XML-Parser k přečtení zpracovávaného dokumentu a naplnění Frequency Tables statistikou počtu výskytů jednotlivých prvků. Nakonec XGrind Kernel znovu vyzve XML-Parser aby vytvořil zakódovanou (viz 5.2.1) podobu XML dokumentu. [9]



Obrázek 5.2: Architektura algoritmu XGrind

5.2.3 Dotazování

Zpracování dotazů je implementováno pomocí slovníkového analyzáru, který poskytuje kódy tagů, atributů a hodnot, a parserem, který se stará o vzájemné porovnávání dotazu s daty. Parser prochází XML dokument do hloubky jako strom a uchovává si informaci o místě (cesta), kde v dokumentu se právě nachází, a obsah uzlů, které právě zpracovává.

Pro dotazy na přesnou a prefixovou shodu jsou zakódovány cesta a dotaz a při hledání jsou data označena jako vyhovující dotazu pouze v případě, že požadovaná cesta odpovídá aktuální pozici parseru v zakódovaných datech a zakódovaná verze dotazu odpovídá obsahu uzlu. Pro dotazy na rozsah nebo částečnou shodu musí být data ve vybraných uzlech dekomprimována a porovnání je provedeno na jejich původních verzích. [9]

Kapitola 6

Existující algoritmy pro kompresi JSON

JSON byl navržen jako odlehčená varianta způsobu formátování dat proti XML, čímž byla odstraněna i redundance při použití počátečního a ukončovacího tagu. Po seznámení s jeho definicí a syntaxí je zřejmé, že zde již nezbyvá moc prostoru k dalšímu odlehčení. Podíváme-li se ale na data zapsaná v tomto formátu, která mohou obsahovat například výstup SQL příkazu SELECT nad databází (viz [odkaz na testovací soubor](#)), můžeme vidět, že se zde některé prvky přeci jen opakuji. Jsou to klíče, včetně uvozovek, v jednotlivých objektech, které stačí zapsat pouze jednou. Tohoto poznatku využívají i dva vybrané algoritmy JSONH a CJSON, které jsou specifické tím, že výstupem z nich je validní JSON. Rád bych čtenáře upozornil, že algoritmů pro kompresi JSON neexistuje mnoho, resp. jsou si velice podobné myšlenkou, provedením i názvy.

S daty v tomto formátu nepracujeme jako s obyčejným textem, ale převedeme do paměti jako hodnoty (pole, objekty, atd.), což odpovídá zpracování pomocí JavaScriptu. Tomu odpovídá i terminologie použitá v této kapitole, která byla popsána v části 1.2.2.

6.1 JSONH

Tento algoritmus dovoluje komprimovat pouze homogenní kolekce dat, v terminologii JSON jde o pole objektů, které mají stejný počet klíčů se stejnými názvy. Homogenitu dat musí zaručit uživatel, algoritmus samotný toto nikterak neošetřuje. Autor projektu JSONH [1], kde je algoritmus implementován, na serveru github.com uvádí, že data mohou být v některých případech zmenšena až na 30 %.

6.1.1 Vzorová data

Mějme homogenní kolekci zaměstnanců firmy, ve které máme uložené databázové id, příjmení a pozici, na které zaměstnanec pracuje. Tato kolekce vypadá následujícím způsobem.

```
[
  { "id": 1, "name": "Sánchez", "position": "Manager" },
  { "id": 2, "name": "Duffy", "position": "Programmer" },
  { "id": 3, "name": "Tamburello", "position": "Worker" }
```

]

6.1.2 Postup komprese

1. Textová data jsou převedena na pole objektů.
2. Z prvního objektu v poli jsou určeny klíče a jejich počet.
3. Ze všech objektů v poli jsou postupně vyzvednuty hodnoty pro příslušné klíče, přičemž je zachováno pořadí klíčů, jak jsme je získali v kroku 2.
4. Je vytvořeno pole, které obsahuje prvky: počet klíčů, seznam klíčů a hodnoty vyzvednuté v kroku 3.
5. Vytvořené pole serializujeme jako JSON do řetězce nebo souboru.

6.1.3 Data po kompresi

```
[ 3, "id", "name", "position", 1, "Sánchez", "Manager", 2, "Duffy",  
  "Programmer", 3, "Tamburello", "Worker" ]
```

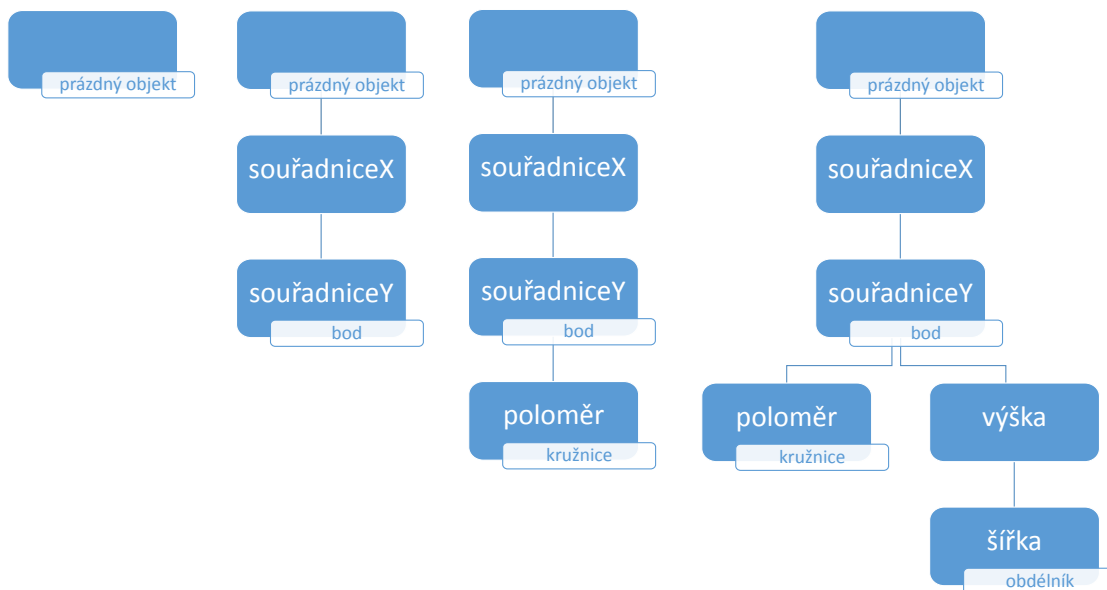
6.1.4 Postup rekonstrukce dat

1. Textová data jsou převedena na pole hodnot.
2. Z prvního prvku v poli určíme počet klíčů, označíme n .
3. Z prvků 2 až $n + 1$ určíme klíče.
4. Z následujících prvků (po $n + 1$) rekonstruujeme původní objekty až do konce pole a ukládáme je do pole nového.
5. Nové pole serializujeme jako JSON do řetězce nebo souboru.

6.2 CJSON

Proti JSONH dokáže algoritmus CJSON komprimovat i nehomogenní data. K tomu, aby bylo při kompresi dosaženo významné úspory, je nutná určitá struktura dat. Tu bych přirovnal k dědičnosti, jak ji známe z principů objektově orientovaného programování. To znamená, že data lze popsat pomocí tříd, které sdílejí některé členy. Účinnost komprese závisí na poměru počtu tříd a množství dat k nim příslušných. Platí, že čím menší počet tříd a větší množství příslušných dat, tím větší je kompresní účinnost. Za ideál lze potom považovat homogenní data, tedy případ kdy stačí k popisu pouze jedna třída, ke které přísluší všechna data.

Algoritmus v průběhu komprese vytváří postupně strom šablon, který je na závěr vložen do výstupu ve formě jednotlivých šablon, které využívají již zmiňované dědičnosti. Konstrukce stromu je zobrazena na obrázku 6.1. **Doplnit očíslování v obrázku a popis vytváření stromu.**



Obrázek 6.1: Postup vytvoření stromu šablon

6.2.1 Vzorová data

Vytvořit data, popsat, upravit obrázek konstrukce stromu!!!

Mějme homogenní kolekci zaměstnanců firmy, ve které máme uložené databázové id, příjmení a pozici, na které zaměstnanec pracuje. Tato kolekce může vypadat následujícím způsobem.

```
[
  { "id": 1, "name": "Sánchez", "position": "Manager" },
  { "id": 2, "name": "Duffy", "position": "Programmer" },
  { "id": 3, "name": "Tamburello", "position": "Worker" }
]
```

6.2.2 Postup komprese

1. Textová data jsou převedena na pole objektů.
2. Ze všech objektů v poli jsou postupně vyzvednuty hodnoty a uloženy do nového pole objektů. Tyto objekty obsahují pouze pole příslušných hodnot. Zároveň je konstruován strom šablon.
3. Ze stromu šablon je vytvořeno pole šablon. Šablona je pole obsahující identifikátor šablony, ze které dědí, a klíče. Identifikátor 0 je rezervován pro šablonu odpovídající prázdnému objektu.
4. Objekty s hodnotami jsou doplněny o identifikátor odpovídající šablony.
5. Ze vzniklých polí vytvoříme objekt obsahující identifikátor kompresního algoritmu (klíč "f"), pole šablon (klíč "t") a pole hodnot (klíč "v") (viz 6.2.3).
6. Objekt vzniklý v bodu 5 serializujeme jako JSON do řetězce nebo souboru.

6.2.3 Data po kompresi

```
{  "f": "cjson",  
  "t": [ [ 0, "x", "y"], [ 1, "width", "height"] ],  
  "v": [ { "": [ 1, 100, 100 ] }, { "": [ 2, 100, 100, 200, 150 ] } ] }
```

6.2.4 Postup rekonstrukce dat

1. Textová data jsou převedena na pole hodnot.
2. Vytvoříme pole, do kterého postupně vkládáme objekty vzniklé z prvků pole hodnot a příslušných šablon.
3. Tuto kolekci serializujeme jako JSON do řetězce nebo souboru.

Kapitola 7

Vlastní implementace vybraných kompresních algoritmů

7.1 Technická specifikace

7.1.1 Vývojové prostředí a programovací jazyk

Na základě osobních preferencí jsem zvolil programovací jazyk C# verze 5.0, který poskytuje sofistikované a otestované nástroje pro práci s XML i JSON, a vývojové prostředí Microsoft Visual Studio Ultimate 2013, které poskytuje plnou podporu pro práci se zvoleným jazykem.

Dalšími použitými nástroji jsou framework NUnit verze 2.6 pro účely testování, ReSharper 8.0.1 pro správu testů a refaktorování¹ a verzovací systém Git, jehož podpora je ve Visual Studiu přímo implementována.

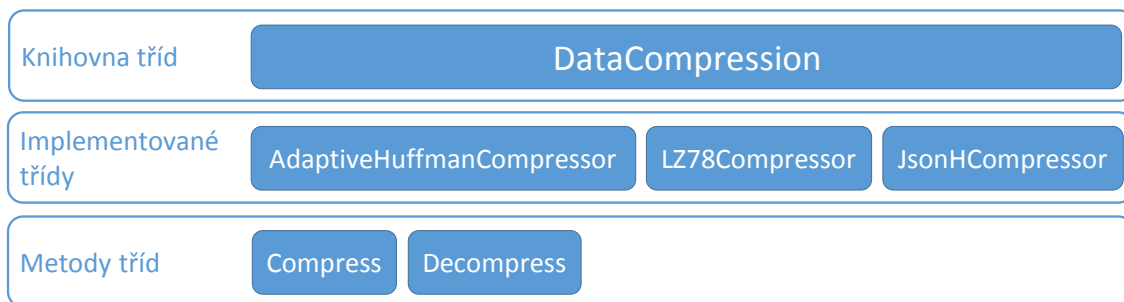
7.1.2 Implementace a testování

Při implementaci jednotlivých algoritmů jsem postupoval metodou vývoje řízeného testy (aglicky Test Driven Development). Při té programátor nejprve napíše neprocházející test, který definuje výchozí a požadovaný stav, a až poté implementuje logiku tak, aby test prošel. Díky vhodně napsané sadě testů je možné velice rychle a snadno odhalit chybu při další implementaci nebo refaktorování.

7.1.3 Projekt

Výstupem mé práce je knihovna tří vybraných kompresních algoritmů. Její struktura je zobrazena na obrázku 7.1. Každému algoritmu odpovídá jedna příznačně pojmenovaná třída, jejíž veřejné rozhraní poskytuje metody `Compress` a `Decompress`. Tyto metody na vstupu očekávají cestu k souboru, který má být zpracován, a po úspěšném zpracování všech dat na stejném umístění vytvoří soubor s komprimovanými/dekomprimovanými daty.

¹Refaktorování je proces úprav kódu vedoucí ke zlepšení jeho struktury, ale nemající vliv na funkčnost.



Obrázek 7.1: Struktura knihovny kompresních algoritmů

7.2 Vybrané algoritmy

7.2.1 Adaptivní Huffmanovo kódování

Ze statistických kompresních metod jsem zvolil Jeffreyho S. Vitterovu verzi adaptivního Huffmanova kódování, která je považována za složitější na implementaci, ale proti jiným verzím staví vyváženější stromy a tedy generuje kratší kódová slova. Uživatel má k dispozici třídu `AdaptiveHuffmanCompressor` s veřejnými metodami `Compress` a `Decompress`.

Jádrem algoritmu je ale třída `VitterTree`, která implementuje logiku chování stromu. Pro každý přečtený znak je při kompresi zavolána jeho metoda `AddChar` (viz algoritmus 1), která provede aktualizaci stromu dle algoritmu 2 a vrátí binární kód znaku ve stromu.

Algoritmus 1: AddChar
Vstupní parametry: ASCII kód znaku Z
Výstupní parametry: binární kód vloženého znaku
<pre> 1 if strom obsahuje uzel se znakem Z 2 begin 3 vrať kód uzlu se znakem Z 4 UpdateTree(Z) 5 end 6 else 7 begin 8 vrať kód uzlu NYT 9 vrať sedmibitový ASCII kód znaku Z 10 UpdateTree(Z) 11 end </pre>

Pro účely dekomprese je strom implementován jako deterministický konečný automat. Obsahuje `ukazatel`, který udržuje referenci na jeden z uzlů (stav automatu). Na počátku `ukazatel` ukazuje na kořen a s každým přečteným bitem komprimované zprávy se přesune do následujícího uzlu dle přechodové funkce. Je-li přečten kód uzlu obsahujícího znak, je tento znak vrácen a strom aktualizován. Je-li přečten kód uzlu NYT, načte algoritmus dalších 7 bitů, které reprezentují ASCII kód nového znaku, ten je vložen do stromu a vrácen. Zpracování jednoho bitu kódu provádí metoda `PushBit` (viz algoritmus 3).

Algoritmus 2: UpdateTree
Vstupní parametry: znak Z
Výstupní parametry: nemá
<pre> 1 uzel U := uzel se znakem Z 2 while uzel U není kořen 3 begin 4 if v bloku existuje uzel s vyšším indexem než má uzel U 5 begin 6 prohod' uzel U s uzlem s nejvyšším indexem v bloku 7 end 8 inkrementuj počet výskytů v uzlu U 9 U := předek uzlu U 10 end </pre>

Algoritmus 3: PushBit
Vstupní parametry: jeden bit kódu B
Výstupní parametry: přečtený znak, byl-li v tomto kroku přečten
<pre> 1 if čten znak nevyskytující se ve stromě 2 begin 3 kumuluj 7 bitů 4 if nakumulováno 7 bitů 5 begin 6 vrať přečtený znak Z 7 AddChar(Z) 8 přesměruj ukazatel na kořen 9 end 10 end 11 else 12 begin 13 dle B přesměruj ukazatel na jeho potomka 14 if ukazatel je uzel obsahující znak 15 begin 16 vrať znak Z v uzlu 17 AddChar(Z) 18 přesměruj ukazatel na kořen 19 begin 20 if ukazatel je uzel NYT 21 begin 22 přesměruj ukazatel na kořen 23 nastav stav stromu na čtení nového znaku 24 begin 25 end </pre>

7.2.2 LZ78

7.2.3 JSONH

Abych mohl ve staticky typovaném jazyce C# zpracovat libovolná data ve formátu JSON, tedy pracovat s nimi jako s objekty různých typů, využívám typ `dynamic`. Ten umožňuje

provést typovou kontrolu až v době běhu (run time), díky čemuž se s JSON v jazyce C# pracuje částečně podobně jako v JavaScriptu. Po zvládnutí transformace dat na objekty je implementace algoritmu jednoduchá.

Kompresní část algoritmu předpokládá, že všechny objekty v poli budou stejného typu. Komprese je závislá na struktuře objektů. Aby bylo dosaženo významného kompresního poměru, měly by objekty mít více atributů s jednoduchými hodnotami typu číslo, řetězec, **true**, **false** nebo **null** než méně atributů se složitými hodnotami typu pole nebo objekt. Metoda **Compress** třídy **JsonHCompressor** je popsána v algoritmu 4.

Algoritmus 4: Compress
Vstupní parametry: cesta k souboru s JSON
Výstupní parametry: nemá
1 načti JSON 2 dekoduj JSON a vlož ho do dynamického pole P 3 z prvního objektu v poli P načti klíče do kolekce K 4 vytvoř prázdnou kolekci hodnot H 5 foreach prvek p v poli P 6 begin 7 foreach klíč k v kolekci K 8 begin 9 serializuj hodnotu pro klíč k v prvku p do kolekce H 10 end 11 end 12 vytvoř dynamické pole D obsahující počet prvků v kolekci K, prvky kolekce K a prvky kolekce H 13 serializuj pole D do souboru

Dekompresní algoritmus je implementován v metodě **Decompress** třídy **JsonHCompressor**. Jeho implementace je velmi jednoduchá, což dosvědčuje i algoritmus 5.

Algoritmus 5: Decompress
Vstupní parametry: cesta k souboru s JSON
Výstupní parametry: nemá
1 načti JSON 2 dekoduj JSON a vlož ho do dynamického pole P 3 z prvního prvku v poli P načti počet klíčů do proměnné n 4 z následujících n prvků vytvoř šablonu objektu S 5 ze zbývajících prvků vytvoř pole objektů D dle šablony S 6 serializuj pole D do souboru

Kapitola 8

Porovnání účinnosti komprese dat ve formátu XML a JSON

Závěr

Seznam použitých zdrojů

- [1] GIAMMARCHI, Andrea. *JSONH: JSON Homogeneous Collections Compressor* [online]. 2014 [cit. 2015-03-17]. Dostupné na: <<https://github.com/WebReflection/JSONH>>
- [2] HANOV, Steve. Compress your JSON with automatic type extraction. *Steve Hanov's Blog* [online]. 2011 [cit. 2015-03-17]. Dostupné z: <<http://stevehanov.ca/blog/index.php?id=104>>
- [3] LIEFKE, Hartmut; SUCIU, Dan. XMill: an Efficient Compressor for XML Data. *Proceedings of the 2000 ACM SIGMOD international conference on Management of data - SIGMOD '00*. 2000, s. 153-164. DOI: 10.1145/342009.335405.
- [4] MAREŠ, Jan. *Teorie kódování*. 1. vyd. Praha: Česká technika, 2008. 120 s. ISBN 978-80-01-04203-8.
- [5] SALOMON, David. *Data Compression: the complete reference*. 4th ed. London: Springer, 2007, xxv, 1092 s. ISBN 978-1-84628-602-5.
- [6] SAYOOD, Khalid. *Introduction to data compression: the complete reference*. 4th ed. Waltham, MA: Morgan Kaufmann, c2012, xvi, 740 s. ISBN 978-0-12-415796-5.
- [7] Standard ECMA-404. *The JSON Data Interchange Format*. Geneva: Ecma International, 2013, [online], [cit. 28. října 2014]. Dostupné na: <<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>>.
- [8] The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. INTERNATIONAL DATA CORPORATION. *EMC* [online]. 2014 [cit. 2015-02-25]. Dostupné z: <<http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm>>.
- [9] TOLANI, P. M.; Haritsa, J. R. XGRIND: A Query-friendly XML Compressor. *Proceedings of the 18th International Conference on Data Engineering - ICDE '02*. 2002, s. 225–234
- [10] VAJDA, Igor. *Teorie informace*. Vyd. 1. Praha: Vydavatelství ČVUT, 2004. 109 s. ISBN 80-01-02986-7.
- [11] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 26. listopadu 2008, [online], [cit. 26. listopadu 2014]. Dostupné na: <<http://www.w3.org/TR/2008/REC-xml-20081126/>>.

Přílohy

Příloha A

Název/obsah přílohy

A.1 Porovnání XML a JSON

XML

```
<widget>
  <debug>on</debug>
  <window
    title="Sample Widget">
    <name>main_window</name>
    <width>500</width>
    <height>500</height>
  </window>
  <image
    src="Images/Sun.png"
    name="sun1">
    <hOffset>250</hOffset>
    <vOffset>250</vOffset>
    <alignment>center</alignment>
  </image>
  <text
    data="Click Here"
    size="36"
    style="bold">
    <name>text1</name>
    <hOffset>250</hOffset>
    <vOffset>100</vOffset>
    <alignment>center</alignment>
    <onMouseUp>sun1.opacity =
      (sun1.opacity / 100) * 90;
    </onMouseUp>
  </text>
</widget>
```

JSON

```
{ "widget": {
  "debug": "on",
  "window": {
    "title": "Sample Widget",
    "name": "main_window",
    "width": 500,
    "height": 500
  },
  "image": {
    "src": "Images/Sun.png",
    "name": "sun1",
    "hOffset": 250,
    "vOffset": 250,
    "alignment": "center"
  },
  "text": {
    "data": "Click Here",
    "size": 36,
    "style": "bold",
    "name": "text1",
    "hOffset": 250,
    "vOffset": 100,
    "alignment": "center",
    "onMouseUp": "sun1.opacity =
      (sun1.opacity / 100) * 90;"
  }
}}
```