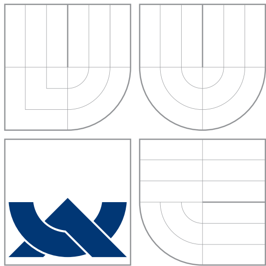
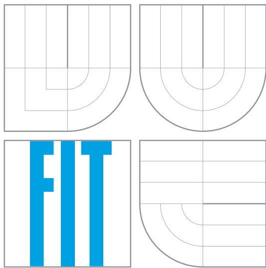


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SROVNÁNÍ IMPLEMENTAČNÍCH STRATEGIÍ DFA
COMPARISON OF IMPLEMENTATION STRATEGIES OF THE DFA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MAREK BALGAR
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KAŠTIL,
SUPERVISOR

BRNO 2009

Abstrakt

Tato bakalářská práce podrobně popisuje výběr metod pro uložení automatu na FPGA a následnou implementaci. Byly vybrány metody bit-split, compress metoda a hashovací tabulka. Dále jsou zde porovnávány jednotlivé paměti, které automaty v reprezentaci jednotlivých metod zaberou. Jsou zde prováděny různé testy s velkou škálou vzorků. Z výsledků jsou zde pak zhodnoceny výhody a nevýhody jednotlivých metod, ale hlavně je zde obsaženo rozhodnutí, která metoda je nejvýhodnější pro uložení automatu na FPGA.

Abstract

This bachelor thesis focuses upon the choices of methods, which are used to store automata on FPGA. There have been chosen these methods: bit-split, compress method and hash table. Then there are being compared the sizes of used memory, which are taken by automata in representation of these methods. The importaint part of this work is testing many kinds of samples. After the results have been achieved, I was able to examine the advantages and disadvantages of each method. Finally the analysis reveals which of the chosen methods is the best to store the automata on FPGA.

Klíčová slova

deterministický konečný automat, maticová reprezentace, bit-split, compress metoda, hash tabulka, testy, FPGA, porovnání celkové paměti, C++, regulární výrazy, stavy, symboly, přechody, testování, zaplněnost přechodů

Keywords

deterministic finite automaton, matrix representation, bit-split, compress method, hash table, tests, FPGA, comparison of the memory, C++, regular expresions, states, symbols, transitions, testing, filling of the transitions

Citace

Marek Balgar: SROVNÁNÍ IMPLEMENTAČNÍCH STRATEGIÍ DFA, bakalářská práce, Brno, FIT VUT v Brně, 2009

SROVNÁNÍ IMPLEMENTAČNÍCH STRATEGIÍ DFA

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Kaštila

. . . .
Marek Balgar
17. května 2009

Poděkování

Chtěl bych touto cestou poděkovat vedoucímu své bakalářské práce, Ing. Janu Kaštilovi, za hodnotné rady a odborné vedení během mé práce.

© Marek Balgar, 2009.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **5**

2 Úvod do problematiky **6**
2.1 Konečné automaty . . . 6
2.1.1 Tabulková reprezentace . . . 6
2.1.2 Grafická reprezentace: . . . 7
2.1.3 Reprezentace zápisem . . . 7
2.2 Vlastnosti konečných automatů . . . 8
2.3 Deterministické konečné automaty . . . 8

3 Rozbor metod **9**
3.1 Maticová reprezentace . . . 9
3.2 Bit-Split . . . 10
3.3 Hashovací tabulka . . . 11
3.4 Compress metoda (metoda prokládání řádků) . . . 12

4 Implementace **15**
4.1 Vstup . . . 15
4.2 Bit-Split . . . 16
4.2.1 Výstup metody . . . 16
4.2.2 Vstupní parametry . . . 16
4.3 Hashovací tabulka . . . 17
4.3.1 Vstupní parametry . . . 17
4.3.2 Výsledná struktura . . . 18
4.3.3 Hash funkce . . . 18
4.4 Compress metoda . . . 19
4.4.1 Posuny . . . 19
4.4.2 Vstupní parametry . . . 19
4.5 Implementované skripty . . . 19

5 Testování **21**
5.1 Výpočty pamětí . . . 21
5.1.1 Maticová reprezentace . . . 21
5.1.2 Bit-split . . . 21
5.1.3 Hash . . . 22
5.1.4 Compress . . . 22
5.2 Testované vzorky dat . . . 22
5.3 Testování různých vzorků . . . 22

5.3.1 Porovnání metod s maticovou reprezentaci . . . 22
5.3.2 Zhodnocení výsledků . . . 23
5.4 Závislost paměti na počtech stavů a přechodů . . . 24
5.4.1 Zadání . . . 24
5.4.2 Sada 1 . . . 24
5.4.3 Sada 2 . . . 25
5.4.4 Celková paměť . . . 27
5.4.5 Neukončené testy . . . 27
5.5 Závislost paměti na zaplněnosti přechodů . . . 28
5.5.1 Zadání . . . 28
5.5.2 Sada 1 . . . 28
5.5.3 Sada 2 . . . 30
5.5.4 Sada 3 . . . 30
5.5.5 Compress metoda vs. hash tabulka . . . 31
5.5.6 Celková paměť . . . 32
5.6 Typy přechodů u metody bit-split . . . 32
5.6.1 Zadání . . . 32
5.6.2 Průběhy testů . . . 32

6 Závěr **35**
6.1 Bit-split . . . 35
6.2 Compress metoda . . . 35
6.3 Hash table . . . 35
6.4 Rozhodnutí . . . 36
6.5 Další vývoj . . . 36
6.6 Zhodnocení . . . 36

Seznam obrázků

2.1 Označení stavu . . . 7
2.2 Označení počátečního stavu . . . 7
2.3 Označení koncového stavu . . . 7
2.4 Označení přechodu mezi dvěma stavy . . . 7
2.5 Takto může vypadat jednoduchý konečný automat grafem (je ekvivalentní k tabulce) . . . 7

3.1 Maticová reprezentace . . . 9
3.2 Efektivní využití . . . 10
3.3 Neefektivní využití . . . 10
3.4 Příklad původního automatu . . . 10
3.5 Vzniklé přechody v různých automatech . . . 10
3.6 Vytvoření deterministického KA z automatu vzniklého z nultého bitu. . . . 11
3.7 Uložení automatu pomocí hashovací tabulky . . . 12
3.8 Příklad původní maticové tabulky na které bude vysvětlen postup . . . 13
3.9 Výsledné pole po vložení 1 řádku . . . 13
3.10 Vložení 0 řádku . . . 13
3.11 Vkládání zbylých řádků . . . 13
3.12 Výsledná struktura . . . 14
3.13 Tabulka posunů . . . 14

5.1 Porovnání výsledků bitsplitu a maticové reprezentace . . . 23
5.2 Porovnání výsledků hash tabulky a maticové reprezentace . . . 23
5.3 Porovnání výsledků compress metody a maticové reprezentace . . . 23
5.4 Závislost výsledného počtu stavů na reg. výrazu . . . 24
5.5 Časová závislost . . . 24
5.6 Závislost počtu stavů na reg. výrazu . . . 25
5.7 Závislost počtu přechodů na reg. výrazu . . . 25
5.8 Porovnání celkové paměti jednotlivých metod . . . 25
5.9 Porovnání celkové paměti metod při původním výrazu . . . 26
5.10 Porovnání celkové paměti metod při použití upraveného výrazu . . . 27
5.11 Porovnání závislosti zaplněnosti přechodů na celkové paměti - části 1 . . . 29
5.12 Porovnání závislosti zaplněnosti přechodů na celkové paměti - části 2 . . . 29
5.13 Porovnání závislosti celkové paměti na zaplněnosti přechodu . . . 30
5.14 Porovnání celkové paměti všech testů . . . 31
5.15 Poměr pamětí . . . 31
5.16 Porovnání celkové paměti testů bitsplitu s různými přechody . . . 32
5.17 Zobrazení testů, kde je výhodnější u bitsplitu přechod o 1 bitu . . . 33
5.18 Zobrazení testů, kde je výhodnější u bitsplitu přechod o 2 bitech . . . 33

5.19 Zobrazení počtu stavů, kde je výhodnější přechod o 1 bitu 34
5.20 Zobrazení počtu stavů, kde je výhodnější přechod o 2 bitech 34

Kapitola 1

Úvod

Postupem doby jsou stále větší požadavky a nároky na vyhledávání regulárních výrazů. Jedním z hlavních problémů je rychlost vyhledávání jednotlivých výrazů. Ta je bezesporu důležitá pro plynulý chod daného zařízení. Jednou z možností, jak je tento problém řešen, je použití konečných automatů a právě ty budou základním kamenem této práce.
Jak je zmíněno v zadání, úkolem této práce je vyhledat metody vhodné pro implementaci deterministického konečného automatu. Následně je nutné vybrané metody implementovat. Poté už se provádí nejdůležitější část této práce kterou je testování. Bude nás hlavně zajímat prostor, který dané uložení zabírá. Samotné testování tedy bude probíhat s různými typy a velikostmi konečných automatů, kde se bude porovnávat použitelnost a vhodnost jednotlivých metod. Hlavním faktorem a cílem bude bezesporu zredukování množství dat, z důvodu implementace v FPGA. To znamená, že bude potřeba najít takové metody, které vhodně provedou kompresi. Samozřejmě ale budeme požadovat, aby zpětná dekomprese byla co nejjednodušší. V opačném případě by totiž dekomprese mohla způsobit nežádoucí navýšení času nutného pro zjištění např. dalšího stavu v automatu. To znamená, že nás v neposlední řadě bude také zajímat rychlost vyhledávání požadovaných dat ve výsledných strukturách.

Kapitola 2

Úvod do problematiky

2.1 Konečné automaty

Konečné automaty jsou spolu s regulárními výrazy a regulárnimi gramatikami jedním z prostředků vhodných pro specifikaci regulárního jazyka. Slovo konečný znamená, že automat má konečnou množinu stavů, ve které je specifikován jeden počáteční stav a množina koncových stavů.

Formálně je konečný automat pětice **M = (Q,** Σ**, R, s, F)**, kde:
Q .... je konečná množina stavů,
Σ.... je konečná množina vstupních symbolů neboli přechodů (nazývána také vstupní abeceda)
R .... je konečná množina pravidel tvaru **ra**$\rightarrow$**s** ,kde **r,s** jsou stavy (náleží množině Q) a symbol **a** je zde prvkem vstupní abecedy Σ
s .... je počáteční stav a patří do množiny Q
F .... je množina koncových stavů a také náleží do množiny Q
Pomocí této pětice tvoříme KA. Mezi jednotlivými stavy se přechází pomocí tzv. přechodu (jde o jeden výpočetní krok KA), což je použití jednoho stavu a vstupního symbolu např. **sc** $\rightarrow$ **p** kde **s, p** jsou stavy a **c** je vstupní symbol. Automat lze reprezentovat třemi způsoby a to tabulkou, grafem a zápisem. Nyní si je tedy rozebereme podrobněji.

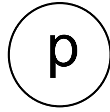
2.1.1 Tabulková reprezentace

V tomto typu reprezentace představují jednotlivé řádky stavy z Q. Každý sloupec pak symbolizuje jeden přechod neboli prvek z množiny Σ. V místě, kde se kříží řádek se sloupcem, je výsledný stav, který vznikne při použití přechodu daného sloupce a stavu daného řádku. Další důležitou věcí je, že počáteční stav je vždy na prvním řádku. Na něm se tedy bude pokaždé začínat. Koncové stavy se pak poznají tak, že jsou podtržené.

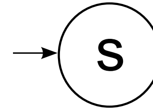
Jednoduchý konečný automat vyjádřený tabulkou může vypadat takto:

	ϵ	a	b	c
s	{r}	ϕ	{p}	{p}
r	ϕ	ϕ	{f,p}	ϕ
p	ϕ	{f}	ϕ	ϕ
f	ϕ	ϕ	ϕ	ϕ

2.1.2 Grafická reprezentace:

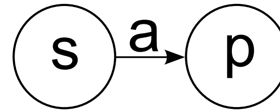


Obrázek 2.1: Označení stavu

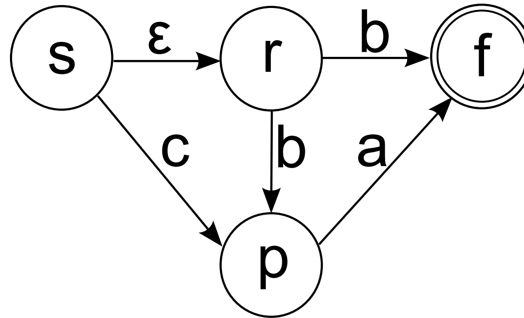


Obrázek 2.2: Označení počátečního stavu

Obrázek 2.3: Označení koncového stavu



Obrázek 2.4: Označení přechodu mezi dvěma stavy



Obrázek 2.5: Takto může vypadat jednoduchý konečný automat grafem (je ekvivalentní k tabulce)

2.1.3 Reprezentace zápisem

Tento zápis je ekvivalentní s grafem a tabulkou

Q ={s,r,p,f}
Σ = {a,b,c}
R = {s $\rightarrow$ r
sc $\rightarrow$ p
rb $\rightarrow$ f
rb $\rightarrow$ p
pa $\rightarrow$ f }
F={f}

2.2 Vlastnosti konečných automatů

Hlavním úkolem konečného automatu je určení, zda přijímá regulární jazyk, či nikoliv. Je velice důležité tuto skutečnost zjistit co nejrychleji a proto je potřeba rychlého vyhledávání v tabulce. To může být ale často problémem u velkých tabulek. Potíž zde nastává u velikosti paměti, kterou taková struktura zabere. Jde o to, že procházení tabulek a hledání odpovídajících přechodů zabere určitý čas. V tomto případě se jedná o lineární složitost.
Nevýhodná z časového hlediska by byla například metoda Backtracking. Probíhá u ní totiž nežádoucí vrácení se zpět ve stavech, což opět může navýšit čas potřebný pro zjištění, zda regulární jazyk přijímá daný konečný automat. Naší snahou je získat co nejmenší tabulku, která je z co největší části zaplněná, což způsobí efektivní vyhledávání.
Z těchto důvodů se provádí různé typy komprese, které umožní tyto nežádoucí vlastnosti eliminovat.

2.3 Deterministické konečné automaty

V předcházející kapitole šlo o konečné automaty, které nebyly deterministické. V této práci nás ale budou zajímat výhradně deterministické konečné automaty. Zde jde tedy o takový, který z každé konfigurace může přejít maximálně do jedné další. Jinými slovy pomocí jednoho přechodu se lze dostat maximálně do jednoho stavu. Formálně je deterministický konečný automat taková pětice M = (Q', Σ', R', s', F'), kde:
Q' .... je konečná množina stavů,
Σ'.... je konečná množina vstupních symbolů neboli přechodů (mezi symboly už nepatří ϵ)
R' .... je konečná množina pravidel tvaru **ra→s** (pro každé ra→s náležící do množiny R platí, že množina **R** bez **{ra→s}** neobsahuje žádné další pravidlo s levou stranou **ra**)
s' .... je počáteční stav a patří do množiny Q'
F' .... je množina koncových stavů a také náleží do množiny Q '
Je zde důležité říci, že tento typ automatu lze získat z nedeterministického KA. Tato operace se provede pomocí dvou kroků:
1. Nejdříve se eliminují všechny ϵ přechody.
2. Následně se snažíme odstranit samotný nedeterminismus tak, že docílíme situace, kdy každý stav má pro jeden symbol maximálně jeden přechod. V tomto kroku vznikají nové stavy, které mohou obsahovat více stavů původního automatu.
Z výše uvedených důvodů vyplývá, že hlavním rozdílem je, že tento automat neobsahuje ϵ - přechody a taky jsou přidány stavy, tak aby simulovaly přechody původního automatu.
V celé této kapitole jsem čerpal v publikacích [1].

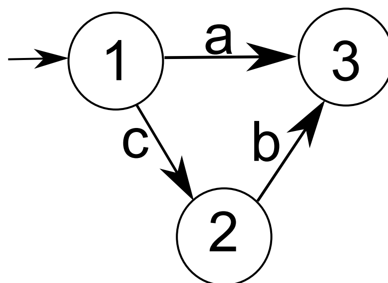
Kapitola 3

Rozbor metod

V této kapitole je cílem popsat a rozebrat do detailu všechny vybrané metody. Popíší se výhody a nevýhody jednotlivých metod, jejich předpokládanou efektivnost a rychlost.

3.1 Maticová reprezentace

Základní reprezentace se kterou budeme porovnávat všechny metody se nazývá maticová reprezentace. Jde o výchozí stav, ze kterého budeme vytvářet ostatní zbylé reprezentace. V podstatě lze říci, že jde o tabulkovou reprezentaci konečných automatů. V paměti pak půjde o dvourozměrné pole, kde indexy řádků budou představovat zdrojové stavy a indexy sloupců budou představovat symboly. Pokud tedy budu chtít zjistit do jakého stavu přejdu po použití jednoho přechodu například výchozí stav je p a použitý symbol je x, pak naleznu cílový stav na pozici [p][x].



Obrázek 3.1: Maticová reprezentace

Výhodou této metody je relativně jednoduché vyhledávání, které má konstantní časovou složitost. Stačí vědět jen dvě výchozí potřebné hodnoty a to hodnotu výchozího stavu a symbolu. Vyhledávání u této metody je tedy relativně rychlé.
Velkou nevýhodou této metody je skutečnost, že paměť nezabírají jen buňky, které jsou skutečně obsazeny, ale také ty ve kterých žádný stav není (stav ze kterého neexistuje přechod s daným symbolem). Tato skutečnost může způsobit v mnoha případech zbytečné plýtvání velké paměti. Platí to hlavně pro případy, kdy existuje mnoho symbolů, ale pro přechody ze stavu jsou vždy využity jen některé.

	a	b	c	d
s	{q}	{p,q}	{r}	Φ
p	{s}	{s,q}	{f}	{r}
r	{r,f}	{s}	{r}	{f}
q	{p,q}	{s}	{s,q}	{f}
f	{s}	Φ	{r}	{q}

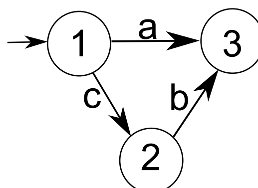
Obrázek 3.2: Efektivní využití

	a	b	c	d
s	{q}	Φ	{r}	Φ
p	Φ	Φ	{f}	{r}
r	Φ	Φ	Φ	{f}
q	Φ	{s}	Φ	{f}
f	Φ	Φ	Φ	Φ

Obrázek 3.3: Neefektivní využití

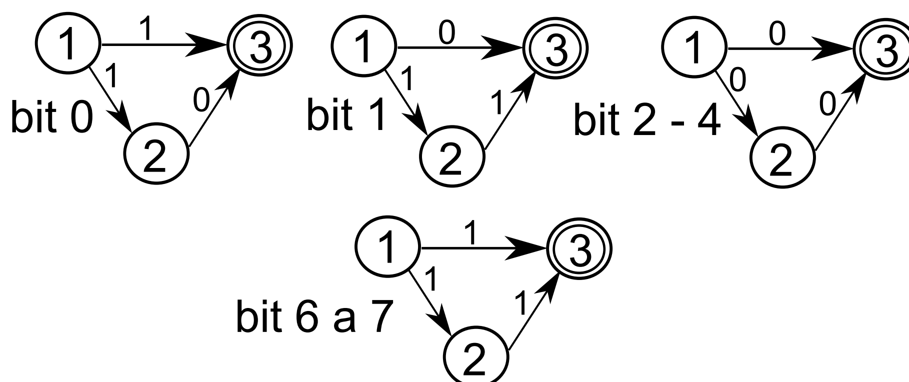
3.2 Bit-Split

Jak již bylo řečeno, u automatů je nevýhodou velký počet možných přechodů, kvůli kterých je zabíráno v paměti zbytečně mnoho prostoru. První metodou, kterou budeme implementovat, se budeme snažit tuto vlastnost eliminovat. Tato metoda se nazývá **Bit-Split**. Hlavní myšlenkou a cílem zde je co největší zaplnění tabulkové reprezentace.
Ze samotného názvu metody lze odvodit, že vzniku více automatů docílíme pomocí oddělení jednotlivých bitů přechodů do více konečných automatů. Celkový počet vzniklých automatů pak závisí na počtu bitů jednoho přechodu. Pokud má automat například přechody jako ASCII písmena, kde každé z nich zabírá 7 bitů, vznikne potom 7 samostatných přechodů v 7 automatech.



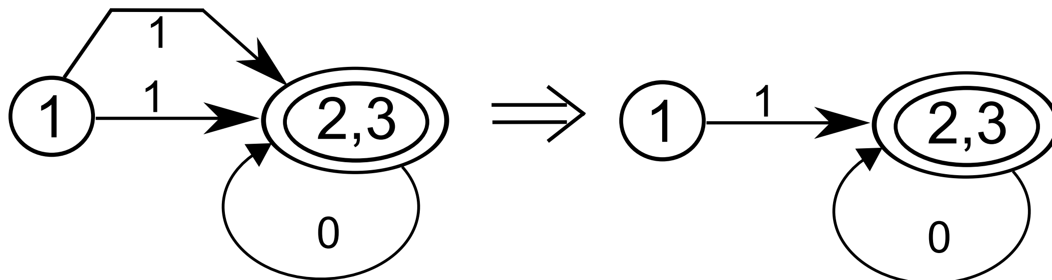
Obrázek 3.4: Příklad původního automatu

Zde máme jako symboly přechodů ASCII znaky a,b,c, kde jejich bitová reprezentace je a = 1100001 , b = 1100010, c = 1100011. Z těchto hodnot se pak postupně vytvářejí automaty podle následujícího obrázku.



Obrázek 3.5: Vzniklé přechody v různých automatech

Tímto krokem ale ještě nedocílíme požadovaného snížení počtu přechodů. Jak můžeme vidět na výsledných automatech, bude určitě docházet v jednotlivých stavech k nedeterminovanosti. Je to díky tomu, že existují maximálně dva symboly jako přechody. To nám pak způsobí vznik více přechodu se stejným symbolem při existenci více přechodů ze stavu. Musíme tedy znovu provést pro každý stav ve všech vzniklých automatech krok, který nám opět vytvoří z KA deterministický KA. Na následujícím obrázku je ukázka provedení tohoto kroku na automatu vzniklého z nultého bitu.



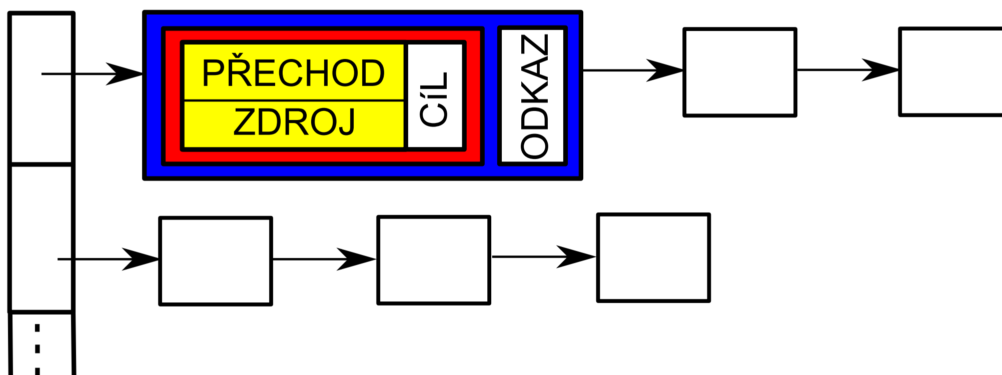
Obrázek 3.6: Vytvoření deterministického KA z automatu vzniklého z nultého bitu.

Můžeme tedy vidět, že skutečně došlo ke snížení počtu přechodů v daném automatu v porovnání s původním. U tak malého automatu je ve skutečnosti efekt opačný, neboť při sečtení všech přechodů jednotlivých automatů jejich počet narostl. Tato metoda totiž předpokládá velké automaty s mnoha stavy a hlavně s velkým počtem přechodů. Tuto vlastnost se budeme snažit zjistit pomocí testování mnoha vzorků dat.
Jak již bylo řečeno výše, imlementací této metody bude zajištěno co největší využití paměti při tabulkové reprezentaci. Tuto vlastnost si také ověříme ve fázi testování. Pro nastudování této metody jsem čerpal v těchto publikacích [3] a [4].
U této metody je důležité zmínit, že ji nelze použít pro všechny regulární výrazy. Existují pravidla, která pokud pro daný výraz platí, lze říci, že metoda je na tento výraz použitelná.
Pravidla:
$L \subseteq Alt(La, Lb)$
Zde platí, že L = L(M), kde M je DKA nad abecedou $A \times B$. Pravidlo tedy říká, že jazyk popsaný bitsplitovými automaty je roven nebo je nadmnožinou jazyka popsaného původním automatem.
$a_1y_1\ a_2y_2\ \ldots a_ny_n, x_1b_1\ x_2b_2 \ldots x_nb_n \subset L \Rightarrow a_1b_1\ a_2b_2 \ldots a_nb_n \subset L$
Zde platí, L = L(M), kde M je DKA nad abecedou $A \times B$. Dále pak platí $a_i, x_i \subset A, b_i, y_i \subset B\ pro\ i = 1, 2, \ldots, n$. Tato podmínka postačuje k určení, že metodu lze na daný výraz použít, ale není nutná. Znamená to, že může být i takový jazyk, pro který rovnost platí, ale podmínku nesplňuje. Tyto podmínky jsem našel v této publikaci [4], kde jsou také detailněji popsány.

3.3 Hashovací tabulka

U další metody, která nás bude zajímat se budeme opět snažit eliminovat neefektivní uložení automatu v paměti. Nebudeme rozdělovat KA na více dalších jako v předešlé metodě, ale naší snahou bude vypustit takové buňky (místa, kde se ukládají cílové stavy), které neobsahují žádný stav.
Do paměti se uloží předem daný počet seznamů, do kterých budeme ukládat jednotlivé

přechody. Zajistíme tím to, že se budou skutečně ukládat jen ty přechody, které existují. Ideologie této metody je založena na hashovací funkci. Ta na základě vstupních dat neboli klíče určí, do kterého seznamu bude daný prvek vložen. Pomocí tohoto klíče lze také zase zpětně zjistit, ve kterém seznamu se daný prvek nachází. Dané rozdělení slouží hlavně k tomu, aby byly prvky rovnoměrně rozmístěny v jednotlivých seznamech a prohledávání netrvalo dlouho.
Na následujícím obrázku (obr 3.7) je graficky znázorněno, jak vypadá uložení jednotlivých prvků symbolizujících přechody. Také je zde detailněji popsáno, jak vypadá jeden z prvků (ohraničen modrou barvou). Každý element obsahuje data (červeně ohraničené) a odkaz na další prvek. Data se pak dělí na klíč(ohraničeno žlutě) a cílový stav. Klíč se pak skládá ze zdrojového stavu a symbolu daného přechodu.



Obrázek 3.7: Uložení automatu pomocí hashovací tabulky

Výhodou této metody je snížení velikosti paměti nutné pro uložení automatu. Jak již bylo řečeno výše, toto je dosaženo díky eliminaci neexistujících přechodů.
Na druhou stranu paměť také trochu naroste díky nutnosti udržování odkazů na jednotlivé prvky v seznamech. Další nevýhodou je také fakt, že nelze říci, jak dlouho se bude hledat požadovaný prvek. Nevíme totiž, zda se nachází na začátku, uprostřed, či na konci seznamu. Z toho vyplývá, že nejhorší možná složitost této metody je lineární.
V našem testování nás bude zajímat, zda se paměťově vyplatí tuto metodu implementovat.

3.4 Compress metoda (metoda prokládání řádků)

Další vybraná metoda, která nás bude zajímat se bude opět snažit eliminovat zbytečně zabrané místo v paměti. Její princip je ale poněkud odlišný od předešlých metod.
Cílem je zde vytvořit z původní matice jednořádkové pole. Toho dosáhneme pomocí prokládání jednotlivých řádků následujícím způsobem. Postupně vybíráme řádky od nejobsazenějšího po nejméně zaplněný, kde řádek představuje stav. Jednotlivé prvky v řádku pak představují přechody, které z něj vychází. Poté co jsme si vybrali řádek, se budeme snažit najít vhodné místo, na které nám celý řádek zapadne. Probíhá to tak, že se hledá volné místo pro první prvek řádku. Když je místo nalezeno, podíváme se, jestli lze vložit s daným posunutím i ostatní prvky (přechody). Pokud nepůjde s daným posunutím vložit byť jediný prvek, budeme toto opakovat, až dojdeme na konec našeho jednořádkového pole. To znamená skutečnost, že musíme celý řádek uložit až na konec seznamu. Podrobněji toto bude vysvětleno na následujících obrázcích. Jak již bylo řečeno, vždy hledáme nejzaplněnější řádek v tabulce. Tím je v našem případě řádek 1, který má 3 zaplněné přechody. Jelikož jde

Přechody

Stavy	a	b	c	d
0	4			3
1	2	3		5
2		1	5	
3			0	
4	2		3	

Obrázek 3.8: Příklad původní maticové tabulky na které bude vysvětlen postup

ale o první zpracovaný řádek a naše výsledné pole je prázdné, řádek hned vložíme. Další řádek, který následně vybereme je 0 řádek a pro něj už musíme hledat pozici. Jak můžeme vidět na obrázku 3.10 vloží se s posunutím 2.

2	3		5

Obrázek 3.9: Výsledné pole po vložení 1 řádku

2	3		5		
		⇑			
		4			3

Obrázek 3.10: Vložení 0 řádku

Takto pokračujeme do té doby, až budou vloženy všechny řádky.

2	3	4	5		3						

⇓ Vložení 2. řádku

2	3	4	5		3	1	5				

⇓ Vložení 4. řádku

2	3	4	5		3	1	5	2		3	

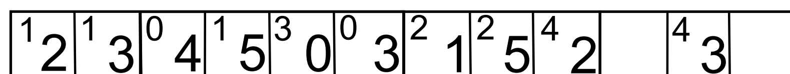
⇓ Vložení 3. řádku

2	3	4	5	0	3	1	5	2		3	

Obrázek 3.11: Vkládání zbylých řádků

Jak je vidět na obrázku 3.11 necháváme v místech, kde nejsou v daném řádku přechody prázdný prvek. Je to zapříčiněno tím, že pokud se chceme zpětně dostat k danému prvku, musíme zachovat pořadí. Dalším důležitým bodem je skutečnost, že si musíme někde ukládat jednotlivé posuny, jinak bychom nevěděli, kde přechody daného řádku začínají. Kromě výsledného pole si tedy ještě budeme ukládat tabulku posunů. Tu lze pro náš příklad

jednoduše vydedukovat z předešlého obrázku (obr 3.13).
Výsledná struktura s výchozími stavy a cílovými stavy pak vypadá takto (3.12).



Obrázek 3.12: Výsledná struktura

Řádek	Posun
0	2
1	0
2	5
3	2
4	8

Obrázek 3.13: Tabulka posunů

U této metody se proto opět budeme snažit testovat, jestli se nám skutečně paměťově vyplatí takto jednotlivé automaty ukládat.

Kapitola 4

Implementace

V této kapitole se budeme zabývat hlavně volbou vhodného jazyka, struktur a vstupních hodnot. Nejprve je vhodné zvolit jazyk pro implementaci konečných automatů. Rozhodl jsem se vybrat jazyk C++. Důležitým faktorem pro výběr tohoto jazyka je skutečnost, že jazyk C++ má vhodné volby dynamických polí, do kterých lze jednoduše vkládat předem neurčitý počet prvků. Tato vlastnost je zde velice důležitá.

4.1 Vstup

Nejdříve je nutné říci, že na počátku máme regulární výrazy, které tvoří tzv. pravidla ze kterých potřebujeme vytvořit takové automaty, jenž je přijímají. Ty vytvoříme pomocí parseru, který byl vytvořen v rámci bakalářské práce pana Andreje Hanka, Bc. [2]. Výstup tohoto parseru je pak naším vstupem. Ten je uložen v nasledující struktuře.

```
typedef map<int int> tDLine;
typedef vector<tDLine> tDTransitions;
typedef vector<vector<set<char> > > tSymbolTable2;

typedef struct tTableDFA {
        int StateCount;
        long int TransCount;
        int SymTableSize;
        tDTransitions Transitions;
        map<int  vector<int> > endStates;
        tSymbolTable2 symbolTable;
} tTableDFa
```

Už podle názvu lze odhadnout co jednotlivé části znamenají.
StateCount ...... je číslo určující počet stavů
TransCount ...... je číslo určující celkový počet přechodů
SymTableSize .... tato proměnná určuje velikost tabulky symbolů. V naší práci nás ale nebude zajímat, neboť tato část bude zůstávat nezměněna.
Transitions ..... v této části se určují jednotlivé přechody, kde jsou specifikovány v tDLine jako dvě čísla. První je zdrojovým stavem a druhé cílovým.
endStates .... zde se určuje, které stavy jsou výstupní.

symbolTable ..... zde se určuje, jaké symboly odpovidají danému přechodu. Tato část nás ale v naší práci nebude zajímat.

4.2 Bit-Split

U této metody máme ukládat více automatů, jejichž množství odpovídá počtu bitů, který zabírá jeden přechod. Bylo by tedy vhodné vytvořit pole, ve kterém se budou všechny automaty nacházet. Jelikož nám ale budou jednotlivé automaty sloužit pouze k vypočtům statistik, vystačíme si jen s jedním prvkem. S ním pak vždy provedeme požadované výpočty a zjištění, následně jej přepíšeme dalším automatem v pořadí.

4.2.1 Výstup metody

Jak jsem již zmínil, ve výsledné struktuře nás budou nejdříve zajímat vzniklé stavy nových automatů. Proto budeme muset použít takovou šablonu, která bude schopna přiřadit jednomu vzniklému stavu více stavů automatu prvního. Toto musíme uložit pro všechny nově vzniklé stavy. Zvolil jsem tedy kombinaci šablon **vector** a **set**, čímž toto umožním. Dále musíme ukládat přechody jednotlivých stavů. Ty jsem se rozhodl ukládat stejně jako u původního automatu. Další důležitou část, kterou musíme být schopni ukládat jsou výstupní stavy. Nebylo by vhodné ukládání volit stejně jako u původního automatu a to z důvodu zbytečného zabírání paměti. U bitsplitu totiž může být o hodně více výstupních stavů než u maticové reprezentace. Pokud by jsme tedy ukládali zase zvlášť stavy, které jsou výstupní, zabralo by nám to zbytečné místo navíc. Rozhodl jsem se pro efektivnější řešení a to takové, že u každého stavu bude navíc bit, určující zda je daný stav výstupní, či nikoliv. Výsledná struktura pak vypadá takto:

```
typedef struct tSummary{
    int StateCount;
    int TransCount;
    int filling;
    int symbols;
}tSummary;
```

4.2.2 Vstupní parametry

Přechody

Jak jsem již vysvětlil v úvodu, u bitsplitu v automatech představují přechody jednotlivé bity původního automatu. Bylo by ale také vhodné zkusit jiné varianty než přechod o jednom bitu. Dosáhli bychom tím jiného počtu nejen automatů, ale také stavů. Vzniknou nám tak úplně jiné automaty, které mohou být pro implementaci výhodnější. Proto jsem se rozhodl u této metody implementovat možnost výběru, zda bude přechod obsahovat 1 nebo 2 po sobě jdoucí bity původního automatu. Pokud by měl původní automat například 8-bitový přechod 11010110, pak při volbě velikosti přechodu 1 bit vznikne 8 automatů, kde každý z nich bude mít jeden z přechodů 1,1,0,1,0,1,1,0. V případě, že zvolíme velikost přechodu 2 bity, pak vzniknou jen 4 automaty a každý z nich bude mít jeden z přechodů 11,01,01,10. Toto lze nastavit vstupním parametrem **-t**, za který napíšeme čislo. Námi implementovaný

algoritmus bude schopen příjmout jen nastavení velikosti přechodů 1 nebo 2 a to z toho důvodu, že automaty s více než dvou bitovými přechody nehodláme testovat.

Stavy a paměť

Další část, kterou by bylo vhodné nastavit před jednotlivými výpočty je velikost, kterou zabírají jednotlivé stavy v paměti. Většinou platí, že se tato hodnota vypočte z celkového počtu stavů. Na druhou stranu můžeme také požadovat určení této hodnoty pomocí jiných parametrů a tudíž budeme chtít zadat tuto hodnotu ručně. Z tohoto důvodu jsem se rozhodl toto nastavení implementovat pomocí dalšího parametru **-c**, za kterým opět následuje číslo, pomocí nehož určujeme kolik bitů má zabírat jeden stav. V případě, že chceme, aby byla velikost vypočítána z celkového počtu stavů, zadáme místo čísla písmeno **a**.

Další

Většinou platí, že nás zajímají jen výsledné statistiky, ale může se také stát, že si budeme chtít prohlédnout výsledné automaty. Proto jsem zavedl další parametr **-w**. Pokud za ním následuje klíčové slovo yes, znamená to skutečnost, že chceme vypsat výslednou podobu konečných automatů.
Posledním parametrem vstupu je přepínač **-f**, za nímž následuje vstupní soubor obsahující výsledný konečný automat. Všechny parametry musí být zadány a to ve správném pořadí, jinak nedojde ke spuštění programu.
Podoba spuštění může například vypadat takto:

./BitSplit -t 2 -c a -w yes -f output.efa

4.3 Hashovací tabulka

Jelikož u hashovací tabulky půjde o seznamy, výsledná struktura bude pochopitelně odlišná. Zde máme objekt hash, který obsahuje ukazatel na pole ukazatelů. Tím dosáhneme požadovaného výsledku, aby výsledná struktura měla více seznamů, do kterých budeme rozdělovat jednotlivé prvky.

4.3.1 Vstupní parametry

Počet seznamů

Musíme se zde zamyslet nad počtem seznamů, do kterých budeme vkládat jednotlivé prvky. Pokud bychom měli jen jeden seznam, bylo by následné vyhledání prvku velice neefektivní. Prvek by se totiž mohl nacházet například až na konci seznamu, jehož průchod by znamenal navýšení času výsledné dekomprese. Naopak pokud bychom zvolili příliš velký počet seznamů, znamenalo by to naopak zbytečné zabírání paměti dalšími ukazately. Tento problém jsem se rozhodl vyřešit tak, že počet seznamů bude vkládán jako vstupní parametr. V případě testování větších automatů zadáme na vstupu vyšší počet seznamů a naopak pokud budeme mít menší automat zadáme nižší počet seznamů. Počet seznamů lze určit parametrem **-s**, za kterým následuje hodnota.

Stavy a paměť

U této metody jsem se taktéž rozhodl implementovat možnost nastavení počtu bitů, které zabírá jeden stav. Opět lze toto nastavení provést pomocí přepínače **-c**, za kterým následuje hodnota.

Ostatní parametry

Posledním parametrem vstupu je přepínač **-f** za nímž následuje vstupní soubor obsahující výsledný konečný automat.
Výsledná podoba spuštění může vypadat takto:

./HashTable -s 10 -c a -f output.efa

4.3.2 Výsledná struktura

Dalším cílem je určení vzhledu prvku. Víme, že máme jednotlivé seznamy do kterých budeme ukládat prvky. Ty zde budou představovat přechody. Jeden prvek tedy bude muset obsahovat zdrojový stav, cílový stav a v neposlední řadě přechod. Všechny tyto hodnoty samozdřejmě ponecháme jako celočíselné hodnoty. Dále také musí být ve výsledné struktuře odkaz na další prvek, abychom dosáhli lineárního seznamu. Výsledná struktura pak bude vypadat takto:

```
typedef struct Item{
    int from;
    int trans;
    int to;
    Item *next;
}Item;
```

Také je důležité vyřešit otázku co bude klíčem hashovací funkce. U dekomprese budeme vždy znát zdrojový stav a přechod a budeme chtít zjistit cílový stav. Z toho vyplývá, že klíčem bude zdrojový stav a přechod.

4.3.3 Hash funkce

Na tuto část jsme nuceni klást velký důraz, neboť je důležité vhodně zvolit hashovací funkci, která nám zajistí rovnoměrné rozložení prvků v jednotlivých seznamech. Nejdříve jsem se rozhodl pro implementaci funkce, která násobí a sčítá jednotlivé části klíče. Následně jsem ale usoudil, že toto nebude nejvhodnější varianta. Jelikož chceme implementovat tyto metody v FPGA nebylo by vhodné používat operaci násobení, která nám sice zajistí dobré rozložení prvků v seznamech, ale na druhou stranu tento výpočet spotřebuje hodně zdrojů. Po delší úvaze a nastudování více hashovacích funkcích jsem se rozhodl implementovat druhou funkci známou pod názvem **Jenkinsonova hashovací funkce**, kterou jsem čerpal z tohoto zdroje [6]. Ta pracuje s operacemi sčítání a posunů, jejichž výpočet není náročný na paměť a zároveň nám zajistí dobré rozložení prvků v seznamech.

4.4 Compress metoda

V této metodě narozdíl od těch předešlých nebude výsledná struktura dvourozměrné pole ani seznamy. Jak již bylo řečeno v předešlé kapitole, snahou bude vložit původní dvourozměrné pole do jednoho řádku a to pomocí prokládání. S tím budou ale spojeny problémy, které si popíšeme.

4.4.1 Posuny

Hlavním problémem u této metody je schopnost co nejrychleji nálézt takové posunutí řádku, aby docházelo k co největšímu překrývání s ostatními. Jelikož chceme dosáhnout co největší zaplněnosti, tak musíme hledat volné místo nejbližší začátku pole. Toho dosáhnu tak, že si vždy nejdříve vyberu první obsazený prvek daného řádku, což představuje první existující přechod daného stavu. Podle toho o který symbol jde v pořadí, tak určím počáteční posun. Následně se v tomto posunutí snažím zjistit, zda jde samotný řádek do výsledného pole vložit. Pokud toto nejde přičtu k hodnotě posunutí jedičku a opakuji celý krok až na konec pole. Tímto postupem dosáhnu toho, že budou jednotlivé řádky proloženy co nejvíce.

4.4.2 Vstupní parametry

Prvním parametrem je zde přepínač **-w**, kterým lze určit, zda chceme vypsat výsledné pole a tabulku posuntí. Pokud chceme provést tento výpis napíšeme za tento parametr klíčové slovo "yes". Dalším parametrem je stejně jako u předchozích testů možnost určení počtu bitů, které zabere jeden stav. Zadává se přepínačem **-c** následujícím hodnotou. Znovu je zde možnost nechat určení velikosti stavu na programu a to klíčovým znakem "a". Posledním vstupním parametrem je zde vstupní soubor obsahující automat, který následuje za přepínačem **-f**. Všechny tyto parametry jsou povinné, jinak nedojde ke spuštění programu. Výsledná podoba spuštění může vypadat takto:
./Compression -w no -c a -f output.efa

4.5 Implementované skripty

Souhrnný výpis

První skript, který jsem se rozhodl implementovat, provede spuštění všech programů jednotlivých metod. Dále pak vypíše statistiky a celkové paměti všech metod. To vše je následně také uloženo v souboru output.vysl, který se musí nacházet ve stejné složce jako spouštěný skript. Vstupní soubor, který obsahuje testovaný automat, se zadává hned za názvem programu. Pokud chceme, aby byla vytvořena závěrečná zpráva s výše zmíněnými statistikami a grafem celkové paměti, pak za název souboru zadáme další parametr v podobě "-yes". Vygeneruje se následně pdf s požadovaným obsahem. Součástí tohoto skriptu je další skript s názvem plot.p, který provede vykreslení grafu - porovnání celkové paměti. Příklad spuštění může vypadat takto:
./script.sh output.efa -yes

Provedení více testů

Další skript s názvem script2.sh provede hromadné převedení více regulárních výrazů v jednom souboru na automaty, které je přijímají. Tento skript produkuje soubory s příponou efa. Jeho spuštění je bez parametrů:
./script2.sh

Dále pak na tento skript navazuje další, který provede otestování jednotlivých metod na všech automatech, které určuje rozsah cyklu ve skriptu. Tento skript pro změnu produkuje soubory s příponou vysl. Jeho spuštění je taktéž bez parametrů:
./script3.sh

Posledním ze sady implementovaných skriptů má název script4.sh. Ten provede vypársování statistik ze všech testů, které určíme a také provede porovnání celkové paměti všech metod s maticovou reprezentací. Jeho spuštění je také bez parametrů:
./script4.sh

Kapitola 5

Testování

Tato kapitola je nejdůležitější částí celé práce, neboť zde budeme porovnávat pomocí vzorků dat jednotlivé metody. V prvé řadě si budeme ověřovat a porovnávat jejich předpokládané výhody, vhodnost použití a paměťovou náročnost.

5.1 Výpočty pamětí

V této části jsem se rozhodl popsat jak je vypočítávána celková paměť jednotlivých metod. Popisuji to až v této části z důvodu návaznosti na jednotlivé testy.

5.1.1 Maticová reprezentace

Zde se celková paměť vypočte jako násobek celkového počtu stavů a celkového počtu symbolů původního automatu. Tato hodnota je pak ještě vynásobena počtem bitů, které zabírá jeden stav. Nakonec se k této hodnotě ještě přičte paměť zabírající výstupní stavy. Ta se vypočte jako násobek počtu výstupních stavů a počtu bitů, které zabírá jeden stav.
Vzorec:
počet_stavů * počet_symbolů * počet_bitů_stavu
+ počet_vystupních stavů * počet_bitů_stavu

5.1.2 Bit-split

Jelikož u bitsplitu jde stejně jako u maticové reprezentace o tabulky, znovu se vynásobí celkový počet stavů (nyní již jde ale o součet stavů všech vzniklých automatů bitsplitu) celkovým počtem přechodů, což v případě bitsplitu bude hodnota 2 nebo 4 (záleží na volbě velikosti přechodu bitsplitu). Tento výsledek se pak následně znovu vynásobí hodnotou, která představuje počet bitů zabírajících jeden stav. K tomuto výsledku bude posléze přičtena paměť potřebná pro uložení výstupních stavů. Ta je ale vypočítána jinak než u maticové reprezentace. Zde jsem zvolil variantu, kde u každého stavu bude bit navíc indikující, zda se jedná o výstupní stav. Počet bitů zabraný výstupními stavy tedy bude odpovídat celkovému počtu stavů bitsplitu. Tuto variantu ukládání výstupních stavů jsem zvolil z toho důvodu, že bitsplit způsobí výskyt původních cílových stavů ve více stavech bitsplitu. Vzorec:
počet_stavů_bitsplitu * počet_symbolů * počet_bitů_stavu_bitsplitu +
počet_stavů_bitsplitu

5.1.3 Hash

U této metody je princip výpočtu celkové paměti trochu jiný. Nejdříve se vypočte paměť kterou zabírá 1 prvek. Tomu odpovídá součet dvojnásobku velkosti paměti, kterou zabírá jeden stav (počáteční a koncový stav), a paměti kterou zabírá jeden symbol. Tuto výslednou hodnotu pak vynásobím celkovým počtem prvků (neboli přechodů). Následně se opět k této vypočtené hodnotě připočte paměť zabírající výstupní stavy, která se vypočte stejně jako u maticové reprezentace (násobek počtu výstupních stavů a počtu bitů zabírajících jeden stav). V celkové paměťi nejsou započítávány jednotlivé odkazy na prvky. Vzorec:
(počet_bitů_stavu * 2 + pocet_bitů_symbolu) * pocet_přechodů +
počet_bitů_stavu * počet_vystupnich_stavu

5.1.4 Compress

Zde se celková paměť zkládá ze 3 částí. Nejdříve se vypočte násobek celého pole, velikosti paměti kterou zabírá 1 stav a čísla dvě. K této hodnotě se přičte paměť zabírající tabulku posunů, která se vypočte jako násobek počtu stavů a počtu bitů zabírající jeden stav. Nakonec se k výsledné hodnotě připočte paměť zabírající výstupní stavy. Ta se vypočte stejně jako u maticové reprezentace a to počet výstupních stavů krát počet bitů zabírajících jeden stav. Vzorec:
velikost_pole * počet_bitů_stavu * 2 +počet_stavů * počet_bitů_stavu +
počet_výstupních_stavů * počet_bitů_stavu

5.2 Testované vzorky dat

Pro různé testy je vhodné použít různé vzorky dat, kde bude středem zájmu velká škála vlastností a hlavně velikostí automatů. Rozhodl jsem se proto využít regulární výrazy vygenerované programem snort (o kterém jsem čerpal odtud [5]), který slouží pro detekci průniků v síti. Budu tak moci testovat reálné automaty, které přijímají tyto regulární výrazy.

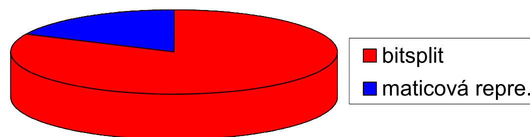
5.3 Testování různých vzorků

Od této části už začne samotné testování dat a porovnání výsledků. Rozhodl jsem se, že nejdříve otestuji co největší množství vzorků dat a z nich pak případně vyberu určitá pravidla, která budou zastupovat určité skupiny výrazů a na ně se zaměřím podrobněji. Jak jsem již psal výše testováná pravidla pochází od programu snort. Z dostupných pravidel jsem tedy nakonec výbral 176 regulárních výrazů pro testy. Na každém z nich jsem spustil skript **script.sh**, který vytvořil statistiky u všech testů. Následně jsem pak porovnal výsledky.
Z všech 176 testů jsem úspěšně získal výsledky u 166 testů. U zbylých deseti testů program běžel neúměrně dlouho a proto jsem se jej v těchto případech rozhodl ukončit násilně. Příčinu proč dochází k tak dlouhému nárůstu doby programu budu zkoumat v dalších částech.

5.3.1 Porovnání metod s maticovou reprezentaci

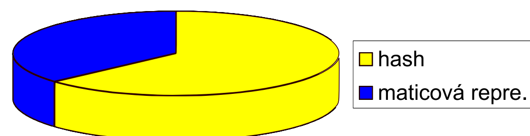
Po nastudování a porovnání všech testů jsem došel k těmto výsledkům. Při porovnání maticové reprezentace a bitsplitu byla paměť výhodnější v:

136 případech metody bitsplitu
30 případech maticové reprezentace.
Metoda bitsplitu je tedy v 81,93% testovaných dat výhodnější než maticová reprezentace.



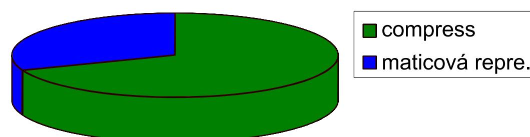
Obrázek 5.1: Porovnání výsledků bitsplitu a maticové reprezentace

Při porovnání maticové reprezentace a hashovací tabulky byla paměť výhodnější v:
105 případech metody hashovací tabulky
61 případech maticové reprezentace
Metoda hashovací tabulky je výhodnější v 63,3% testovaných dat než maticová reprezentace



Obrázek 5.2: Porovnání výsledků hash tabulky a maticové reprezentace

Při porovnání maticové reprezentace a metody prokládání řádků byla paměť výhodnější v:
115 případech metody prokládání řádků
51 případech maticové reprezentace
Metoda prokládání řádků je výhodnější v 69,3% testovaných dat než maticová reprezentace



Obrázek 5.3: Porovnání výsledků compress metody a maticové reprezentace

5.3.2 Zhodnocení výsledků

Z těchto hodnot můžeme vidět, že všechny 3 metody měly procentuálně lepší výsledky než maticová reprezentace. Nejvýhodnější metodou, co se týče procentuální úspěšnosti byla metoda bitsplitu, která měla v 81,93% případů nižší paměťové nároky, což je vysoká úspěšnost.

5.4 Závislost paměti na počtech stavů a přechodů

5.4.1 Zadání

V této sadě testů jsem se rozhodl více zaměřit na vhodnost použití bitsplitu. Postupným navyšováním počtu stavů a přechodů budu moci zjistit poměr změn stavů původního automatu a bitsplitu.

5.4.2 Sada 1

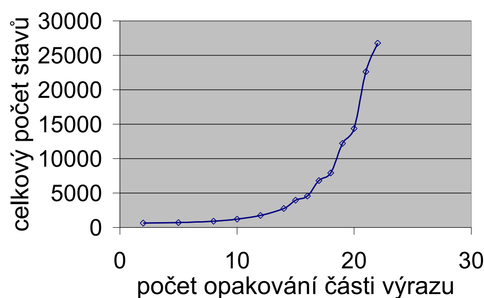
Vybral jsem si takový regulární výraz v předešlých testovaných výrazech, kde měl bitsplit větší paměťové nároky než maticová reprezentace. Výraz vypadá takto:

```
/((\w+)[\r\n\s]*\x3a=[\r\n\s]*(\x27[^\x27]{2,}\x27| \x22[^\x22]{2,}\x22)
[\r\n\s]*\x3b.*SCHEMA_NAME[\r\n\s]*=>[\r\n\s]*2| SCHEMA_NAME\s*=>\s*(\x27
[^\x27]{2,}|\x22[^\x22]{2,})|\verb|\(\s*(\x27[^\x27]{2,}
|\x22[^\x22]{2,}))/si
```

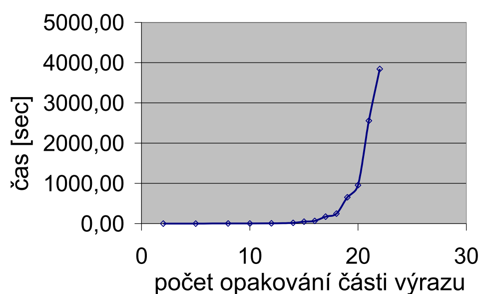
Zde před každým testem změním jen první část ve složených závorkách `{2,}`, čímž mohu určovat spodní hranici počtu opakování výrazu. Tím lze regulovat výsledný počet stavů a přechodů. Všechny pravidla se nachází v příloze v testovací části test 1-> 1 sada pod názvy test_2.pcre až test_22.pcre.

Průběhy testů

Po spuštění mnoha testů s daným pravidlem jsem vybral některé, na kterých se budu snažit popsat výsledek. U daného regulárního výrazu jsem testoval takové výrazy u kterých jsem měnil opakování výrazu 2-22 krát. Nevolil jsem vyšší počet opakování a to z důvodu, že s narůstající hodnotou nám exponenciálně narůstá nejen výsledný počet stavů bitsplitu, ale také doba vytvoření všech automatů bitsplitu. Také by to bylo zbytečné, neboť narůstání paměti bylo evidentní. Například u nastavení regulárního výrazu s hodnotou 22 trvalo samotné provedení 66 minut(viz graf 5.5). Jak jsem se již zmínil exponenciálně se zvyšuje i celkový počet stavů, což můžeme vidět na grafu 5.4. Není to ale způsobeno tím, že by navyšování hodnot v regulárním výrazu způsobilo exponenciální navýšení stavů a přechodů u původního automatu, což můžeme vidět na grafech 5.6 a 5.7.

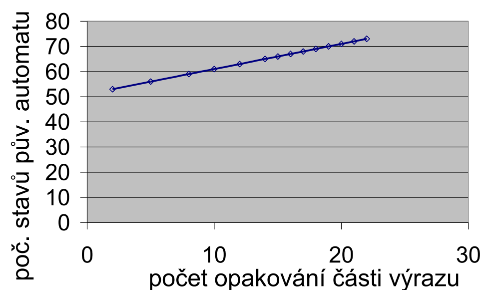


Obrázek 5.4: Závislost výsledného počtu stavů na reg. výrazu

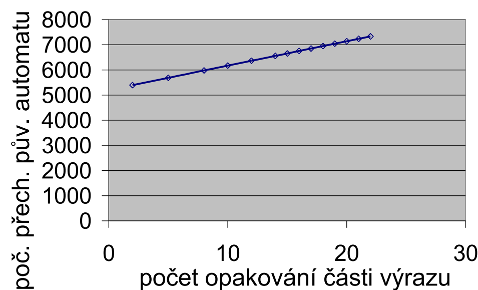


Obrázek 5.5: Časová závislost

Po delší úvaze a sledování průběhů testů jsem zjistil, že k prudkému nárůstu počtu stavů bitsplitu začne docházet až při určitém počtu stavů a přechodů. Jak můžeme na grafech také

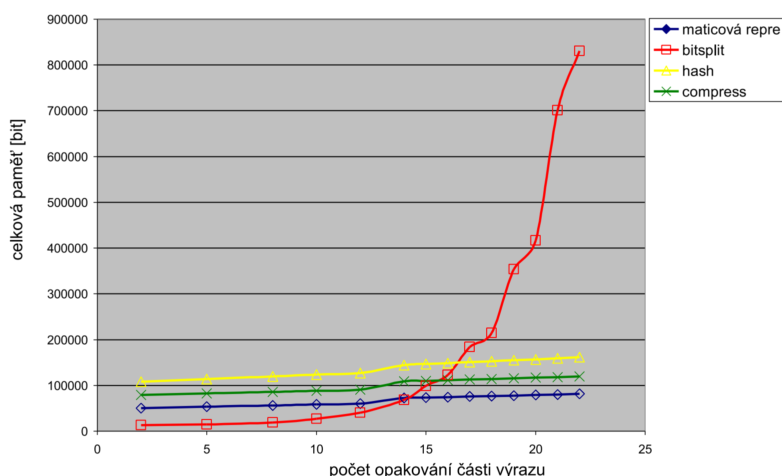


Obrázek 5.6: Závislost počtu stavů na reg. výrazu



Obrázek 5.7: Závislost počtu přechodů na reg. výrazu

vidět toto navýšení je způsobeno tím, že čím více stavů a přechodů má původní automat, tím více stavů má výsledný automat. Vyšší počet stavů totiž umožňuje existenci velkého množství potencionálních kombinací stavů, které mohou obsahovat výsledné stavy. Vyšší počet přechodů zase způsobuje, že je těchto výsledných stavů generováno více. Pokud je pak velké množství kombinací stavů, nebude tak často docházet k situaci, kdy daný stav už existuje, ale naopak se budou muset ukládat nové. Porovnání celkových pamětí můžeme vidět na obrázku 5.8.



Obrázek 5.8: Porovnání celkové paměti jednotlivých metod

5.4.3 Sada 2

V této sadě znovu provedu obdobný test, ale již s jiným regulárním výrazem. Vybral jsem takový výraz, kde měl bitsplit oproti předešlé sadě testů nižší paměťové nároky než maticová reprezentace.

Část 1

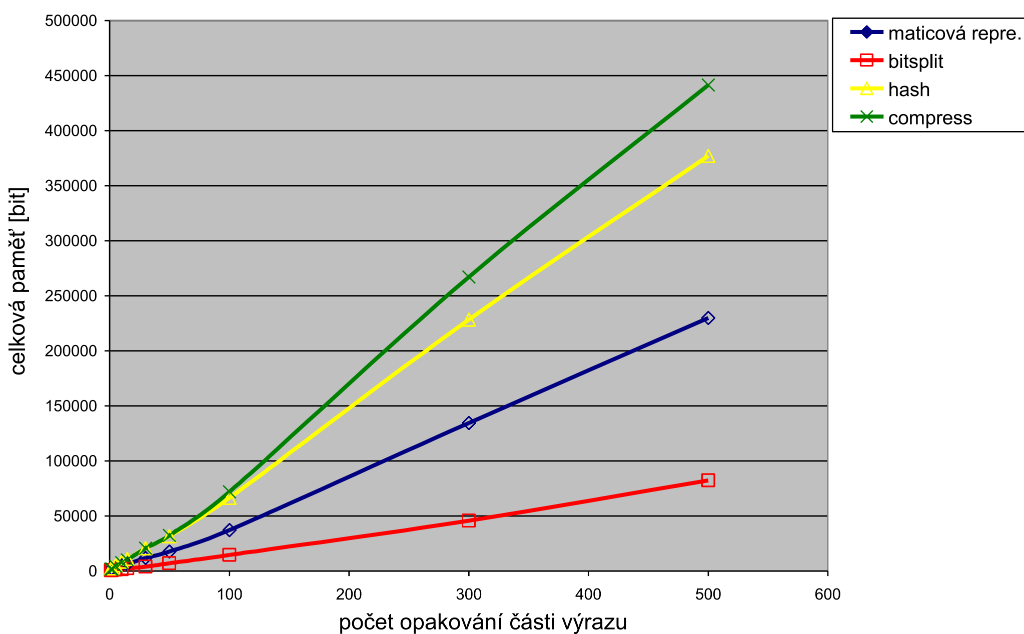
Jeho struktura vypadá takto:

```
/\x2fnds[^\r\n]{1}|\x2fnds\x2f[^&\r\n\x3b]{1}|\x3C[^\x3E\x0a]{1}/Usmi
```

Pro každý test se bude znovu měnit jen první část ve složených závorkách, která opět umožnuje určovat počet opakování výrazu. Všechny pravidla se nachází v příloze v testovací části test 1-> 2 sada ->cast 1 pod názvy test_1.pcre až test_500.pcre.

Průběhy testů

Po spuštění mnoha testů s daným pravidlem jsem vybral vzorek, na kterém se opět budu snažit popsat výsledek. Zjistil jsem, že u daného regulárního výrazu byla u všech testů celková paměť bitsplitu nižší než u maticové reprezentace (viz graf 5.9). Bez problému byla možnost vložení většího počtu opakování a bitsplit byl stále výhodnější než maticová reprezentace. Také doba provedení bitsplitu byla u všech testů řádově v sekundách. Naskytá se nám tedy otázka: "Čím je způsoben rozdíl oproti předchozí sadě testů?"Rozhodl jsem se, že budu pátrat dál než jen u původního automatu, ale pokusím se zjistit, zda na tento rozdíl nemá vliv některá část testovaných regulárních výrazů. Po porovnání reg. výrazu z obou testovaných skupin, jsem nalezl pár částí, které se nachází v prvním výrazu, ale nikoliv v druhém.



Obrázek 5.9: Porovnání celkové paměti metod při původním výrazu

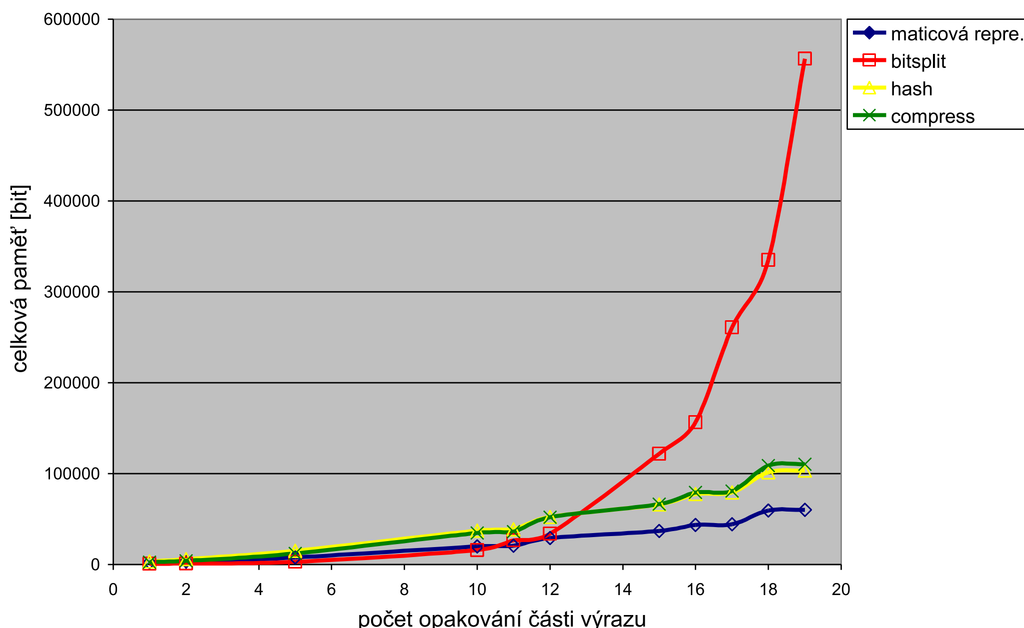
Část 2

Podstatná část, u které jsem zpozoroval, že činí rozdíl mezi jednotlivými výrazy, je `[\r\n\s]*`. Znamená to libovolný počet opakování sekvence znaků `\r\n\s`. Vložíme toto tedy do výrazu části 1 a porovedeme znovu všechny testy. Výraz pak bude vypadat takto:

```
//\x2fnds[\r\n\s]*[^\r\n]{1}|\x2fnds\x2f[^&\r\n\x3b]{1}|
\x3C[^\x3E\x0a]{1}/Usmi
```

Všechny pravidla, automaty i výsledky se nachází v příloze v testovací části test 1-> 2 sada ->cast 2 pod názvy test_1.pcre až test_19.pcre. Po provedení všech testů na automatech vzniklých z upravenéno regulárního výrazu jsem zjistil, že celková paměť bitsplitu

byla zpočátku nižší. S narůstajícím počtem opakování výrazu ale začala celková paměť bitsplitu exponenciálně růst stejně jako v sadě 1(viz graf 5.10). Oproti původnímu regulárnímu výrazu jsem testoval už jen opakování 1-19, neboť také doba provedení bitsplitu začala mít exponenciální růst.



Obrázek 5.10: Porovnání celkové paměti metod při použití upraveného výrazu

5.4.4 Celková paměť

Při porovnání celkové paměti jednotlivých metod jsem zjistil, že ve všech testech první sady mají hashovací tabulka a metoda prokládání řádků větší paměťové nároky než původní maticová reprezentace. Tato skutečnost je ale pravděpodobně způsobena tím, že u všech testů byla velká celková zaplněnost přechodů, která se pohybovala okolo **63%**.
U druhé sady testů je situace obou metod obdobná. Pouze u prvních dvou testů se paměť blížila velikosti paměti zabrané maticovou reprezentací. To bylo dle mého názoru opět způsobeno tím, že byla u těchto testů nižší celková zaplněnost než u ostatních a to 44% a 52%. Tuto teorii se budu snažit ověřit v další skupině testů.
Metoda bitsplitu zajišťuje nížší paměťové nároky v případě, že výrazy neobsahují části s velkým počtem opakování např s výrazem "*", což je libovolný počet opakování reg. výrazu. V opačném případě u ní docházelo k exponenciálnímu nárůstu paměti.

5.4.5 Neukončené testy

Zde bych se chtěl ještě vrátit k těm testům ze sady 0, u kterých byla doba provedení algoritmu bitsplitu extrémně dlouhá. Z toho důvodu jsem je ukončil a nezískal tedy statistiky těchto testů. Šlo zde o testy s čísly 29, 35, 52, 84, 132, 135, 136, 137, 138, 139 a 161(viz příloha). Při zhlédnutí jednotlivých výrazů jsem zjistil, že ve všech těchto testech se nachází velké hodnoty opakování výrazu a také libovolný počet opakování reg. výrazu značící se "*"nebo "+". Na základě zkušeností a závěrů z předešlých testů jsem provedl

s každým z těchto výrazů více experimentů, kde jsem znovu měnil počet opakování. Použil jsem u nich nižší počty opakování, abych získal výsledek v relativně krátkém čase (řádově maximálně v minutách). U všech těchto experimentů jsem zjistil, že s navyšováním počtu opakování části výrazu dochází k exponenciálnímu růstu celkové paměti a doby provedení bitsplitu. Z toho vyplývá, že u všech těchto výrazů je nevýhodné použít metodu bitsplitu pro uložení automatu.
Dále jsem také vyzkoušel otestovat tyto původní regulární výrazy s tím rozdílem, že jsem odstranil všechny výrazy libovolného opakování. Například test číslo 29:
`/^AUTHINFOs+USERs[^n]{200}/smi`
jsem upravil odstraněním znaku "+":
`/^AUTHINFOsUSERs[^n]{200}/smi`
U každého z těchto testovaných regulárních výrazů byla najednou paměť potřebná pro uložení bitsplitu nižší než u maticové reprezentace. Tato skutečnost mě jen utvrdila v závěru, že bitsplit je výhodné používat jen u těch regulárních výrazů, které neobsahují libovolné opakování.

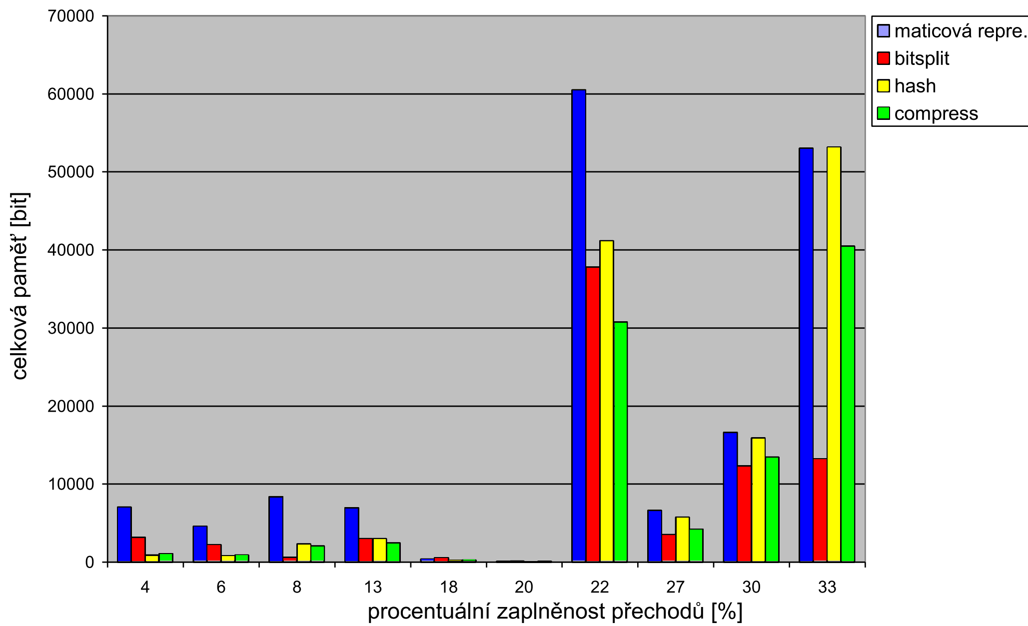
5.5 Závislost paměti na zaplněnosti přechodů

5.5.1 Zadání

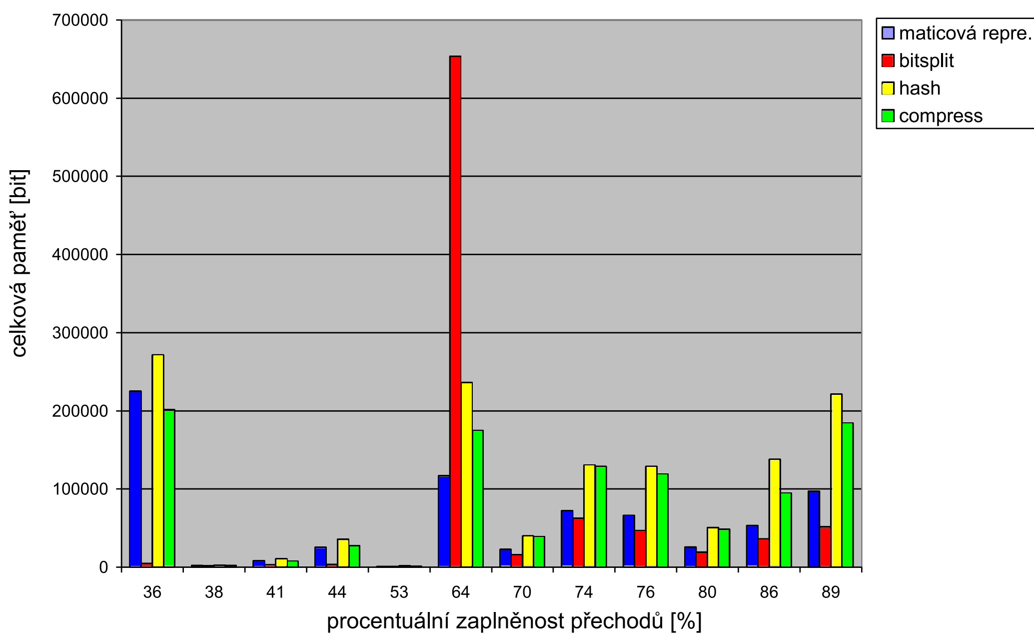
V další skupině testů bude mým cílem dosáhnout rozdílné zaplněnosti přechodů. Tentokrát mě již nebude zajímat nárůst paměti mezi jednotlivými testy a to z důvodu, že v každém testu je experimentováno s jinými automaty, kde každý z nich může mít rozdílnou velikost. Po spuštění všech testů porovnám výsledné paměti mezi jednotlivými metodami u každého z testů a určím případnou závislost na faktoru zaplněnosti. Zde jsem provedl více sad testů, abych mohl porovnávat výsledky na více vzorcích.

5.5.2 Sada 1

Rozhodl jsem se projít výsledky všech testů ze sekce testy 0 a následně najít takové vzorky, kde bude různá zaplněnost přechodů automatu. Snažil jsem se získat co největší škálu vzorků. Vybral jsem nakonec automaty se zaplněním přechodů od 4% do 89%. Všechny pravidla, automaty i výsledky se nachází v příloze v testovací části test 2-> sada 2 pod názvy test_4.pcre až test_89.pcre.
Po provedení testů jsem zjistil skutečnost, že metoda hashovací funkce a metoda prokládání řádků mají menší paměťové nároky při nižší zaplněnosti než maticová reprezentace. Metoda používající hashovací tabulku byla v porovnání s maticovou reprezentací paměťově efektivnější do testu o zaplněnosti **30%**(viz graf 5.11). S rostoucí zaplněností pak u této metody narůstal poměr paměti vzhledem k maticové reprezentaci. U metody prokládání řádků byl výsledek obdobný. Tato metoda dosáhla ale o něco vyšší úspěšnosti než hashovací tabulka. Zde celková paměť dosáhla nižších hodnot než maticová reprezentace až do zaplněnosti **41%**(viz graf 5.12). U metody bitsplitu nelze z této sady testů vyvodit jakoukoliv závislost na zaplněnosti přechodů. Jak můžeme ale vidět na grafech, tato metoda má ve vetšině testů oproti maticové reprezentaci nižší celkovou paměť potřebnou pro uložení automatu. Vyjímkou je test se zaplněností 64%. Zde jde ale o regulární výraz obsahující libovolný počet opakování, u kterého máme ověřeno z předchozí skupiny testů, že navyšuje paměť automatu při použití metody bitsplitu.



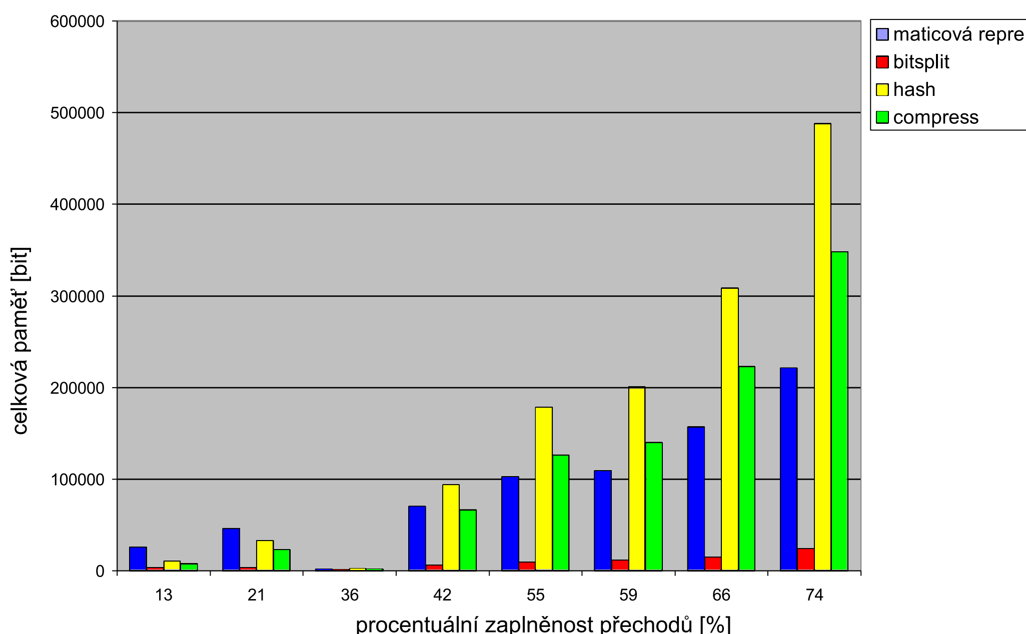
Obrázek 5.11: Porovnání závislosti zaplněnosti přechodů na celkové paměti - části 1



Obrázek 5.12: Porovnání závislosti zaplněnosti přechodů na celkové paměti - části 2

5.5.3 Sada 2

V další sadě testů jsem se rozhodl vybrat jeden regulární výraz, který budu měnit případně k němu přidávat další části výrazů tak, abych dosáhl znovu různé zaplněnosti. Vybral jsem si tento výraz:
`/((\.\.\/|\.\.\\).*|(\.(exe|dll))~~|(afregdsd|w|d|grgeregghth|dfgergerwff)`
`|\x22[^\x22]{20}/Ri`
Podařilo se mi jej upravovat tak, že jsem získal vzorky zaplněnosti v rozmezí 21% a 89%. Všechny pravidla, automaty i výsledky se nachází v příloze v testovací části test 2-> sada 2 pod názvy test_1_1.pcre až test_1_11.pcre.
Po otestování této sady jsem zjistil, že metoda bitsplitu má u daného automatu znovu nižší paměťové nároky než maticová reprezentace a ušetřená paměť oproti maticové reprezentaci je místy až několikanásobná(viz graf 5.13). Co se týče ostatních dvou metod, tak velký úspěch už se nedostavil. U hashovací tabulky byly hodnoty nižší pouze u prvních dvou testů, kde zaplněnost opět nepřesáhla **30%**. Metoda prokládání řádků byla trochu úspěšnější, neboť paměť byla nižší u čtyř prvních testů, v nichž nepřesáhla zaplněnost **42%**. Z grafu lze ale znovu vidět, že metoda prokládání řádků má ve všech testech stejně jako v první sadě nižší paměťové nároky než hashovací tabulka a tudíž je výhodnější. V grafu jsem nezobrazil poslední tři testy z důvodu extrémně velkých hodnot, které by znehodnotily graf. Tyto tři testy měly ale stejnou tendenci růstu jako předešlé.

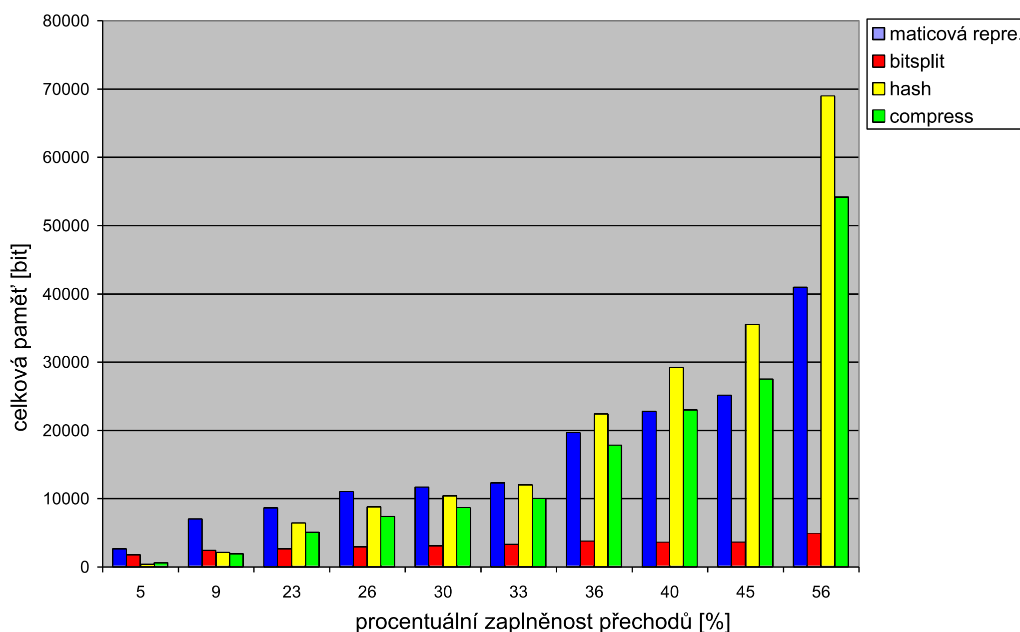


Obrázek 5.13: Porovnání závislosti celkové paměti na zaplněnosti přechodu

5.5.4 Sada 3

V poslední sadě této skupiny testů jsem znovu vybral regulární výraz a upravoval jej, abych získal různou zaplněnost. Původní regulární výraz vypadá takto:
`/^100013Agentsvr\x5E\x5EMerlin|\x11[^\x11]{12}/smi`
Rozsah zaplněnosti přechodů testovaných vzorků je v v rozmezí 5% až 79%. Rád bych zde zmínil, že bylo mým cílem získat co nevíce vzorků v rozmezí 20% až 40%, neboť u předešlých

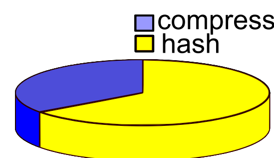
dvou sad testů se v tomto rozmezí nacházela hranice, kde se stávaly metody hashovací tabulky a prokládání řádků paměťově neefektivní. Všechna pravidla, automaty i výsledky se nachází v příloze v testovací části test 1-> sada 3->cast 3 pod názvy test_2_1.pcre až test_2_11.pcre.
Po spuštění testů zjišťuji, že situace má obdobný výsledek jako předchozí dvě sady. Metoda bitsplitu opět snižuje paměť oproti maticové reprezentaci a stejně jako v předešlé skupině testů zde nedochází k viditelným závislostem celkové paměti na zaplněnosti přechodů. U hashovací tabulky je paměť nižší u prvních šesti testů, při kterých byla maximální zaplněnost **33%**. To je přibližně stejná hodnota jako u první a druhé sady testů. Výsledky u metody prokládání řádků dosáhly znovu větších úspěchů. Zabraná paměť byla nižší u prvních sedmi testů. Zde se hranice zaplněnosti pohybovala mezi **36 až 40%**. Vše je znázorněno v grafu 5.14, kde jsem znovu nevykreslil všechny testy z důvodu větší přehlednosti a viditelnosti nižších hodnot.



Obrázek 5.14: Porovnání celkové paměti všech testů

5.5.5 Compress metoda vs. hash tabulka

Rozhodl jsem se, že ještě provedu více porovnání compress metody(metoda prokládání řádků) a hashovací tabulky. Srovnal jsem proto tyto dvě metody u všech provedených testů ze sekce **test 0** a získal jsem tím více nezávislých porovnání. Zjistil jsem skutečnost, že ze 166 bylo ve 108 případech paměťově výhodnější použít compress metodu než hashovací tabulku(viz graf 5.15).
Z tohoto porovnání a také ze třech předešlých sad testů vyplývá, že je výhodnější používat compress metodu, která měla u většiny testů jednotlivých sad nižší paměťové nároky než hashovací tabulka. V porovnání s maticovou reprezentací se hash tabulku vyplatí použít u automatů, kde zaplněnost přechodů nepřesahuje zhruba **33%**. Oproti tomu metodu prokládání řádků je vhodné použít,



Obrázek 5.15: Poměr pamětí

pokud je zaplněnost přechodů nižší než **40%**. Také tento fakt dokazuje, že je výhodnější použít compress metodu než hashovací tabulku.

5.5.6 Celková paměť

Z předešlých testů nám vyplývá, že celková zaplněnost přechodů u původního automatu má vliv na metodu prokládání řádků a na metodu hashovací tabulky. U metody bitsplitu nebyl z těchto testů prokázán vliv zaplněnosti přechodů na celkovou paměť. Z toho tedy usuzuji, že zaplněnost přechodů nehraje tak podstatnou roli, jako u ostatních dvou metod. Tento závěr platí minimálně do zaplněnosti 89%, do které jsme testovali vzorky dat. Bylo totiž velmi problematické získat vzorky dat s vyšší zaplněností.

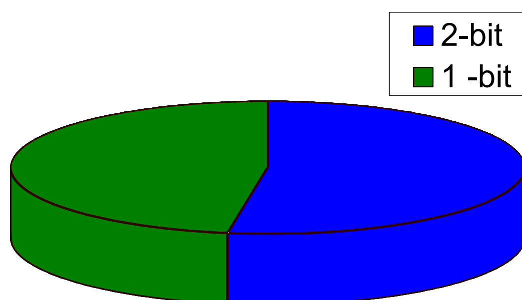
5.6 Typy přechodů u metody bit-split

5.6.1 Zadání

V této sekci jsem se rozhodl, že se zaměřím hlavně na metodu bitsplit. Cílem bude zjištění, zda není paměťově výhodnější použít přechod se dvěma bity. Test tedy provedu jednoduše aplikací této metody na stejném automatu dvakrát po sobě , kde jedna skupina výsledných automatů bude mít přechod 1 bit a druhá skupina 2 bity původního automatu. Následně se pak opět pokusím zhodnotit jednotlivé výsledky a vyvodit případné závěry.

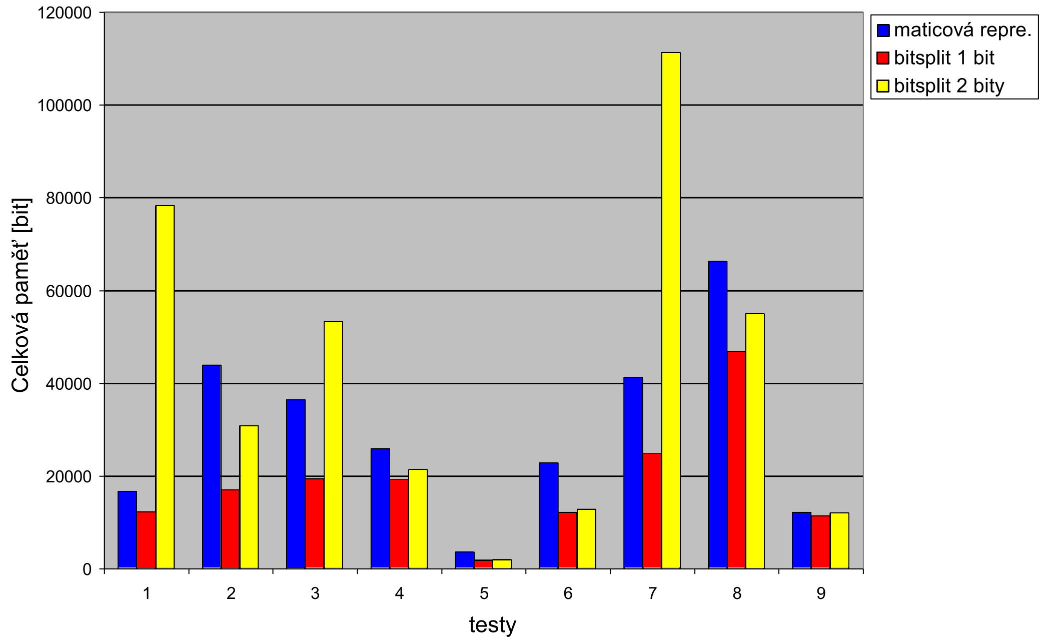
5.6.2 Průběhy testů

Zde jsem vybral více různých výrazů ze skupiny testů použitých u testu 0, tak abychom získali velkou škálu automatů. Spustil jsem dva výše zmiňované typy testů na 152 vzorcích. Po sledování výsledků testů jsem zjistil, že v 83 případech z 152 exprimentů bylo výhodnější použít přechod o 2 bitech než přechod o 1 bitu (viz graf 5.16). To je přibližně 54,6%. Z toho vyplývá, že obě metody mají přibližně stejnou úspěšnost.

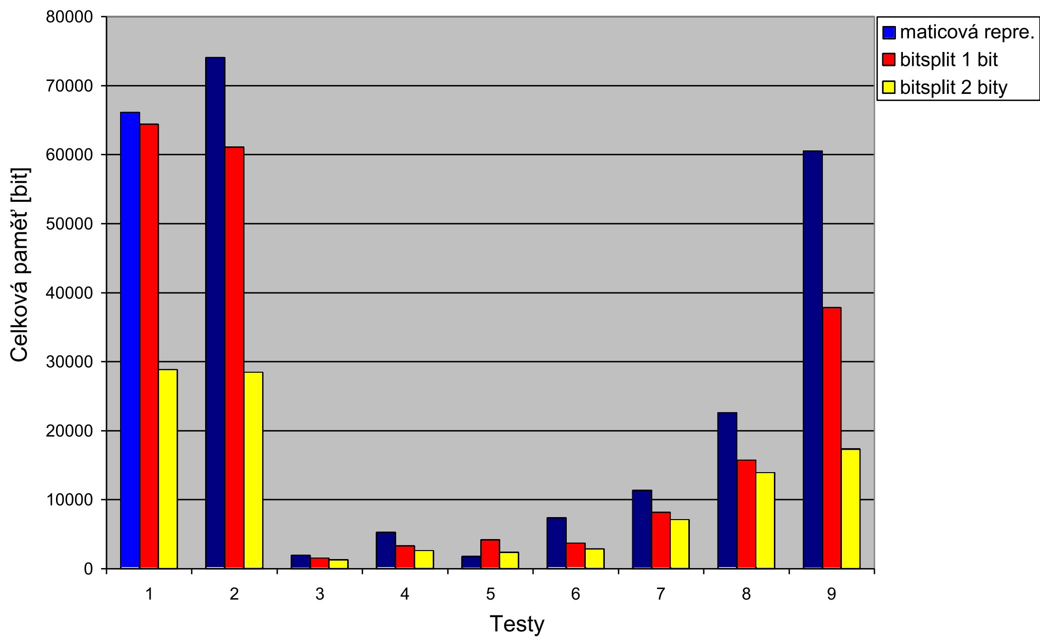


Obrázek 5.16: Porovnání celkové paměti testů bitsplitu s různými přechody

Následně jsem se vytvořil dva grafy. V prvním (5.17) jsem znázornil výběr pár testů z těch, kde je výhodnější použít přechod o jednom bitu a v druhém grafu (5.18) jsem znázornil ty testy, kde bylo výhodnější použít přechody o dvou bitech.

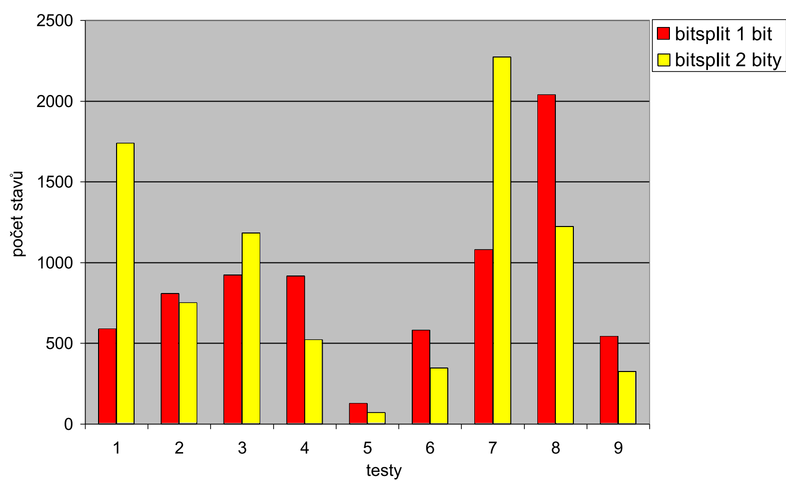


Obrázek 5.17: Zobrazení testů, kde je výhodnější u bitsplitu přechod o 1 bitu

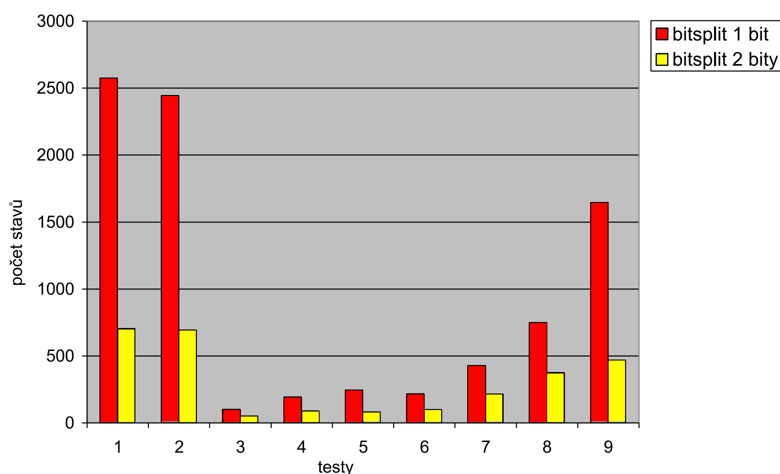


Obrázek 5.18: Zobrazení testů, kde je výhodnější u bitsplitu přechod o 2 bitech

Rozhodl jsem se více prozkoumat jednotlivé statistiky těchto vybraných testů, abych zjistil, co způsobuje paměťové rozdíly mezi jednotlivými metodami. Zjistil jsem, že u daných dvou typů testů je hlavní rozdíl v počtech stavů. U testů, kde je paměťově výhodnější přechod o 2 bitech, je podstatně nižší počet stavů (5.19). Díky tomu je pak nižší počet bitů, které zabere jeden stav. To pak v důsledku způsobí menší velikost výsledné tabulky, i když ta má větší počet symbolů(místo 2 má 4 symboly). U testů, kde je výhodnější
1-bitový přechod je sice znovu často menší počet stavů, ale ne tak podstatně jako u první skupiny testů (graf 5.20).
Dále jsem také porovnával původní regulární výrazy jednotlivých skupin a hledal jsem odlišnosti, které by případně způsobovaly paměťové rozdíly jednotlivých typů metod. Nenašel jsem ale žádnou část, u které by toto šlo prokázat.



Obrázek 5.19: Zobrazení počtu stavů, kde je výhodnější přechod o 1 bitu



Obrázek 5.20: Zobrazení počtu stavů, kde je výhodnější přechod o 2 bitech

Kapitola 6

Závěr

V této práci bylo mým cílem vyhodnotit jednotlivé metody a jejich případné použití v FPGA. Hlavním úkolem bylo porovnání celkových pamětí, které zaberou jednotlivé metody pro uložení testovaných automatů. Zaměřil jsem se na použití různých regulárních výrazů a jejich vliv na celkové paměti implementovaných metod.

6.1 Bit-split

U metody nazývané bitsplit jsem došel k závěru, že má z vybraných metod nejlepší výsledky a tudíž je nejvýhodnější ji použít. U většiny testů dosahovala bezkonkurenčně nejnižších paměťových nároků na uložení automatu (ve 136 případech ze 166 -> 81,96% úspěšnost). Na druhou stranu má i tato metoda nevýhodu. Pokud je totiž v regulárním výrazu, který přijímá ukládaný automat libovolné opakování v kombinaci s větší hodnotou jiného opakování, požaduje metoda pro uložení oproti ostatním vysoké paměťové nároky.
Dalším poznatkem u této metody je fakt, že zaplněnost přechodů nemá vliv na celkovou paměť bitsplitu.
Tuto metodu lze ale použít u takových výrazů, které splňují určitá pravidla viz. třetí kapitola, ve které jsou tyto podmínky rozebrány podrobněji. V tomto projektu jsem se těmito podmínkami nezabýval a považoval jsem je u všech výrazů za splněné.

6.2 Compress metoda

Zde co se týče testové sady 0 byla metoda u 166 testů výhodnější ve 115 případech než maticová reprezentace, což je 69,3% úspěšnost. U této metody byl zjištěn taktéž vliv zaplněnosti přechodů. V případě, že je zaplněnost přechodů nižší než 40%, je výhodnější použít pro uložení automatu tuto metodu než maticovou reprezentaci.

6.3 Hash table

Tato metoda je na základě všech provedených testů bezesporu nejméně úspěšná a vhodná v porovnání s ostatními vybranými metodami. Jak již bylo zmíněno v testové sadě 0, u 166 testů byla tato metoda ve 105 případech výhodnější než maticová reprezentace, což je 63,3% úspěšnost. U této metody jsem zjistil vliv zaplněnosti přechodů. Pokud je totiž zaplněnost přechodů nižší než 33%, je výhodnější použít pro uložení automatu tuto metodu než maticovou reprezentaci.

6.4 Rozhodnutí

Po zhodnocení výsledků práce jsem tedy přesvědčen, že při výběru metody pro uložení automatu na FPGA bych volil metodu bitsplitu. V případě, že má automat nízkou zaplněnost přechodů zvážil bych užití compress metody neboli metody prokládání řádků.

6.5 Další vývoj

Tento projekt by mohl být dále rozšířen o část, ve které by se zjišťovalo, zda testované výrazy splňují požadované vlastnosti, při kterých metoda bitsplitu funguje správně. Obecný regulární výraz totiž tyto pravidla nesplňuje.

6.6 Zhodnocení

Tato práce mi ukázala možnosti použití málo známých metod. Zdokonalil jsem zde své schopnosti programování a programovacích technik v jazyce C++. Při implementaci jednotlivých metod jsem narazil na několik problémů. Asi nejtěžší pro mě byla skutečnost, že k některým z metod nebylo mnoho dostupného materiálu, neboť jde o málo rozšířené algoritmy. Při testování jsem narazil na spoustu poznatků a problémů, které se mi podařilo zdárně vyřešit. Jsem přesvědčen, že tyto znalosti mi budou přínosem jak při tvorbě dalších prací v budoucím studiu, tak v případné profesi.

Literatura

[1] Alexander Meduna, R. L.: *Formální jazyky a překladače - Přednášky*. VUT Brno, FIT, 2006.

[2] Hank Andrej, B.: *Detekce narušení počítačové sítě*. VUT Brno, FIT, 2007.

[3] Lin Tan, B. B.; Sherwood, T.: *Bit-Split String-Matching Engines for Intrusion Detection and Prevention*. ACM New York, NY, USA, 2006, iSSN:1544-3566.

[4] Ryan Dixon, O. E.; Sherwood, T.: *Automata-Theoretic Analysis of Bit-Split Languages for Packet Scanning*. San Franciscon, CA, USA, 2008, ISBN 978-3-540-70843-8.

[5] Tripp, G.: *Regular expression matching with input compression: a hardware design for use within network intrusion detection systems*. Springer Paris, 2007.

[6] WWW stránky: Hash. `http://burtleburtle.net/bob/hash/doobs.html`.