VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATICKÉ SHLUKOVÁNÍ REGULÁRNÍCH VÝRAZŮ

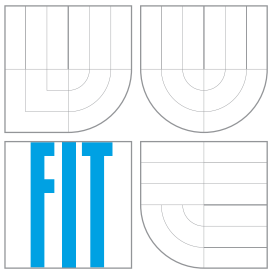
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TIMOTEJ STANEK

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AUTOMATICKÉ SHLUKOVÁNÍ REGULÁRNÍCH VÝRAZŮ

AUTOMATIC GROUPING OF REGULAR EXPRESSIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. TIMOTEJ STANEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KAŠTIL
SUPERVISOR

BRNO 2011

Automatické shlukování regulárních výrazů

Prohlášení

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

.....
Timotej Stanek
25. května 2011

Poděkování

Srdečne by som chcel poďakovať vedúcemu práce Honzovi Kaštilovi za kvalitnú odbornú pomoc a za šancu učiť sa nové znalosti a objavovať nové možnosti.

Abstrakt

Práce pojednává o detekci útoků na počítačové síti pomocí IDS systémů. IDS obsahují pravidla pro detekci ve formě regulárních výrazů, které jsou při detekci reprezentovány pomocí konečných automatů. Je vysvětlena náročnost této detekce pomocí nedeterministických a deterministických konečných automatů. Tuto náročnost je možné redukovat pomocí shlukování regulárních výrazů. Je uvedem shlukovací algoritmus a možné přístupy jak tento algoritmus vylepšit a zrychlit. Jedním z přístupů je genetický algoritmus, který dokáže pracovat v reálném čase. Nakonec je popsán přítup pomocí algoritmu Random Search. Na závěr jsou prezentovány výsledky experimentů s těmito přístupy a jsou porovnány mezi sebou.

Abstract

This project is about security of computer networks using Intrusion Detection Systems. IDS contain rules for detection expressed with regular expressions, which are for detection represented by finite-state automata. The complexity of this detection with non-deterministic and deterministic finite-state automata is explained. This complexity can be reduced with help of regular expressions grouping. Grouping algorithm and approaches for speedup and improvement are introduced. One of the approches is Genetic algorithm, which can work real-time. Finally Random search algorithm for grouping of regular expressions is presented. Experiment results with these approches are shown and compared between each other.

Klíčová slova

Systém detekce útoků, IDS, bezpečnost sítí, regulární výraz, PCRE, konečný automat, deterministický konečný automat, nedeterministický konečný automat, shlukovaní regulárních výrazů, genetický algoritmus, Random Search.

Keywords

Intrusion Detection System, IDS, network security, regular expression, PCRE, finite-state automaton, deterministic finite-state automaton, non-deterministic finite-state automaton, regular expression grouping, genetic algorithm, Random Search.

Citace

Timotej Stanek: Automatické shlukování
regulárních výrazů, diplomová práce, Brno, FIT VUT v Brně, 2011

Obsah

Kapitola 1

Úvod

V dnešnej dobe je po počítačových sieťach prenášaných stále viac informácií. Na jednej strane nám to umožňuje zasýtenie nášho hladu po informáciách, na strane druhej, ak chceme tieto informácie filtrovať v reálnom čase, či už pre zvýšenie bezpečnosti alebo získanie znalostí z informácií, tak k tomu musíme vynaložiť oveľa väčšie úsilie, ktoré musíme k tomuto vynaložiť.

Preto vznikli nástroje, ktoré umožňujú, okrem iného, detekovať útoky vyskytujúce sa v sieťovej prevádzke, takzvané IDS (Intrusion Detection System). Tieto vyhľadávajú vzory vyjadrené pomocou rôznych techník, napr. pomocou regulárnych výrazov. Regulárne výrazy budú hlavnou témou práce. Tieto výrazy sú transformované do počítaču vhodnejšej formy konečných automatov, pomocou ktorých sme schopní relatívne rýchlo vzory hľadať. Pravidlá pre systémy, ktoré budeme skúmať sú minimálne sčasti v každom z nich popísané pomocou PCRE (Perl Compatible Regular Expression) regulárnych výrazov. Až na pár výnimiek vo formáte PCRE, sú tieto výrazy prevoditeľné na konečné automaty. IDS systémom a regulárnym výrazom je venovaná druhá kapitola.

Konečné automaty, ktorými sa budeme zaoberať, budú deterministické konečné automaty (DKA), pretože pri sériovom spracovaní je ich použitie rýchlejšie ako spracovanie s pomocou nedeterministických konečných automatov (NKA), pri ktorých nám môže výrazne narásť časová zložitosť. Na NKA dokážeme previesť práve PCRE výrazy, ktoré sme následne schopní zdeterminizovať (tým vznikne DKA) a minimalizovať. Konečné automaty a ich význam je rozobratý v tretej kapitole.

Po tom, ako získame z pravidiel konečné automaty (KA), ktoré ich reprezentujú, sa vynára problém rýchleho hľadania vzorov pomocou nich. Ak hľadáme iba jedno pravidlo, tak hľadanie vzorov nie je časovo náročný problém. Ak však chceme detekovať množinu pravidiel, musíme prechádzať množinu odpovedajúcich DKA zároveň, čo nám priamo úmerne znižuje priepustnosť dát. Tento problém je možné riešiť pomocou zhlukovania KA, čo je proces, ktorý nám pre danú vstupnú množinu KA vytvorí novú množinu KA, ktoré sú ekvivalentné pravidlám vstupnej množiny. Toto prinesie menšiu časovú zložitosť hľadania vzorov, ale môže neúmerne narásť zložitosť zhlukových KA, čo sa prejaví na ich priestorovej náročnosti. Cieľom správneho zhlukovania je vytvoriť čo najmenej zhlukov tak, aby výsledný súčet veľkostí všetkých automatov bol minimálny. Problému zhlukovania sa venuje štvrtá kapitola.

Jedným z hlavných cieľov tejto práce je zhrnúť doterajšie poznatky v oblasti vysokorýchlostného vyhľadávania regulárnych výrazov a zhlukovania regulárnych výrazov. Konkrétne opísať IDS systémy a ich úlohu, vysvetliť, prečo sa zaoberáme práve nimi, zaviesť definície matematických štruktúr, ktoré sú základnými stavebnými kameňmi pri vyhľadávaní vzorov

a predstaviť základný algoritmus pre zhlukovanie regulárnych výrazov.

Po zadefinovaní a vysvetlení základných pojmov a predstavení zhlukovacieho algoritmu sa pokúsime upraviť tento algoritmus tak, aby bol efektívnejší a poskytol lepšie výsledky. Všetky tieto uplatnené prístupy otestujeme a porovnáme. Princípmi a výsledkami urýchlenia zhlukovacieho algoritmu sa zaoberá piata kapitola.

V poslednej šiestej kapitole sa pokúsime o implementáciu zhlukovacieho algoritmu, ktorý by pracoval v reálnom čase, a odstránil tak niektoré neduhy klasického zhlukovacieho algoritmu. Konkrétne sa bude jednať o použitie genetického algoritmu. Po analýze výsledkov narazíme na ďalší algoritmus schopný pracovať taktiež v reálnom čase, avšak s lepšími výsledkami.

Kapitola 2

IDS systémy

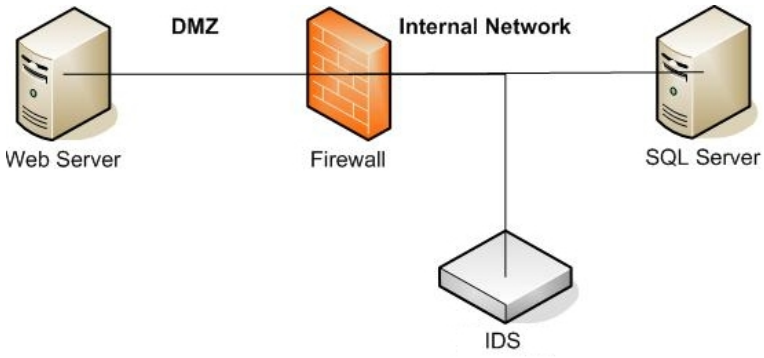
V počítačovej bezpečnosti môžeme IDS (Intrusion Detection System) systém chápať ako nástroj, ktorý nám umožní sledovaním prevádzky počítača odhaliť možné bezpečnostné útoky a riziká systému. Vďaka tomu sa IDS stávajú v dnešnej dobe veľmi dôležité. Na obrázku 2.1 je znázornený príklad použitia IDS.

2.1 Spracovávanie informácií

Intrusion Detection Systémy je možné na základe dát, ktoré spracovávajú, zaradiť do dvoch hlavných tried: [1]

- Host-based intrusion detection system (HIDS) - analyzujú na základe systémových volaní, žurnálov aplikácií, zmien v súborovom systéme, atď.,
- Network intrusion detection system (NIDS) - analyzujú sieťovú prevádzku, a to môže byť na základe anomálií alebo pomocou vopred daných pravidiel, môžu analyzovať agregované informácie o sieti, ale aj analyzovať po jednotlivých bajtoch na sieti.

V tejto práci sa budeme venovať iba NIDS systémom, resp. ich pravidlám vo forme regulárnych výrazov. V nasledujúcom texte nebudeme rozoberať HIDS, teda pojem IDS budeme chápať ako NIDS.



Obrázek 2.1: Príklad použitia sieťového IDS/IPS

2.2 Detekčné techniky

Všetky Intrusion Detection Systémy používajú jednu z týchto detekčných techník: [16]

- Štatistické IDS založené na anomáliách (A-IDS – Anomaly based IDS) – stanovujú rozhodovací limit (performance baseline) založený na hodnotení normálnej sieťovej prevádzky. Potom vzorkujú danú sieťovú aktivitu v porovnaní s týmto limitom s cieľom určiť, či táto aktivita spadá do tohto limitu. Ak je vzorkovaná prevádzka nad tento limit, tak je spustený alarm, ktorý upozorní na anomáliu.
- IDS založené na signatúrach (S-IDS – Signature based IDS) – sieťová prevádzka je skúmaná na základe preddefinovaných vzorov útokov, takzvaných signatúr (pravidiel). Veľa dnešných útokov má presne dané signatúry. Pri správnom používaní zbierky takýchto signatúr, musí byť neustále aktualizovaná pre potlačenie nových hrozieb.

Táto práca sa zaoberá vyhľadávaním regulárnych výrazov, čo je jedna z najnáročnejších operácií v S-IDS.

2.3 Softwarové nástroje

Za referenčné nástroje, ktorých pravidlá budeme spracovávať, budeme považovať nasledujúce IDS nástroje:

- Bro Intrusion Detection System – opensource unixový NIDS, najprv analyzuje sieťovú prevádzku na aplikačnej vrstve a potom použije analyzátor na základe udalosti, ktorý následne hľadá vzory podľa pravidiel [8],
- L7-filter – klasifikátor pre linuxový framework Netfilter, ktorý identifikuje pakety na základe dát z aplikačnej vrstvy; dokáže klasifikovať pakety Kazaa, HTTP, Jabber, atď. nezávisle na porte; dopĺňa klasifikátory založené na IP adresách, portoch, a pod. [17],
- Snort – opensource je sieťový IDS/IPS (Intrusion Prevention System); kombinuje používanie pravidiel a vyhľadávanie anomálií; je to celosvetovo najviac nasadzovaný IDS/IPS systém, ktorý je de facto štandard pre IPS [11].

Kapitola 3

Reprezentácia a vyhľadávanie pravidiel

Neoddeliteľnou súčasťou S-IDS systémov je preddefinovaná databáza pravidiel. Každé pravidlo unikátne definuje množinu reťazcov, ktoré reprezentujú konkrétny protokol alebo útok. Dôležitým faktom je tiež to, že základom správneho používania týchto pravidiel je častá aktualizácia databázy.

Pravidlá v IDS systémoch by mohli byť reprezentované pomocou databázy, ktorá by obsahovala pravidlá definujúce jednotlivé možné útoky ako reťazce znakov. Na detekciu útoku by teda stačilo porovnávať sieťovú prevádzku s reťazcami v databáze. Ak je však cieľom pokryť všetky útoky vrátane tých, ktoré môžu mať rôzne podoby, museli by v databáze byť všetky možné reťazce odzrkadlujúce tieto útoky. Tento problém je možné vyriešiť pomocou regulárnych výrazov, ktoré majú väčšiu vyjadrovaciu silu a existujú algoritmy ako pomocou nich vyhľadávať. [15]

3.1 Pravidlá v IDS systémoch

Pravidlá, resp. časti pravidiel v IDS bývajú často popísané pomocou regulárnych výrazov. V prípade Snortu sú to PCRE regulárne výrazy. V PCRE však existujú konštrukcie, ktoré nie je možné previesť na konečné automaty. PCRE výrazy teda majú vyššiu vyjadrovaciu silu ako regulárne výrazy. Keďže na konečných automatoch bude postavený zvyšok práce, tak budú takéto pravidlá ignorované, resp. budú uvažované len regulárne výrazy prevoditeľné na konečné automaty.

Z WWW stránok uvedených IDS nástrojov v sekcii 2.3 sme stiahli pravidlá určené pre tieto aplikácie. Keďže každý nástroj mal zoznam pravidiel v inom formáte, vytvorili sme jednoduché skripty pre extrakciu pravidiel z jednotlivých IDS systémov. Výsledkom boli 3 súbory, v ktorých sa na každom riadku nachádzal regulárny výraz popisujúci pravidlo. Z tabuľky 3.1 môžeme vidieť, že Bro a Snort obsahujú rádovo viac pravidiel ako L7-filter.

IDS systém	Bro IDS	L7-filter	Snort
Počet pravidiel	1036	278	2260
Spolu	3574		

Tabulka 3.1: Počet pravidiel v IDS systémoch (október 2010)

Vo väčšine prípadov sa pre vyhľadávanie regulárnych výrazov používajú konečné automaty, či už vo forme rôznych implementácií v hardvéri (Clark [2, 3], Sidhu-Prasanna [12]) alebo vylepšení konečných automatov (Delay DFA [5], XFA [14]).

3.2 Konečné automaty

Konečný automat (KA) je matematická štruktúra, ktorá dokáže detekovať slová z jazyka definovaného regulárnym výrazom. Regulárne výrazy sú vhodnejšie na popis množiny reťazcov. Majme však reťazec a chceme zistiť, či daný regulárny výraz tento reťazec vystihuje. K riešeniu tohto problému je vhodnejšie použiť konečný automat. Týmto konečným automatom reprezentujeme regulárny výraz a dokážeme jednoducho určiť, či reťazec náleží do množiny vyjadrenej regulárnym výrazom.

Konečné automaty je možné deliť na:

- Deterministické KA
- Nedeterministické KA

Táto vlastnosť KA je kľúčová v spojitosti s architektúrou, na ktorej chceme simulovať detekciu reťazcov. Prioritnou architektúrou v tejto práci bude pre nás sériová architektúra. Paralelnými architektúrami (napr. FPGA) sa budeme zaoberať iba okrajovo.

3.2.1 Nedeterministický konečný automat

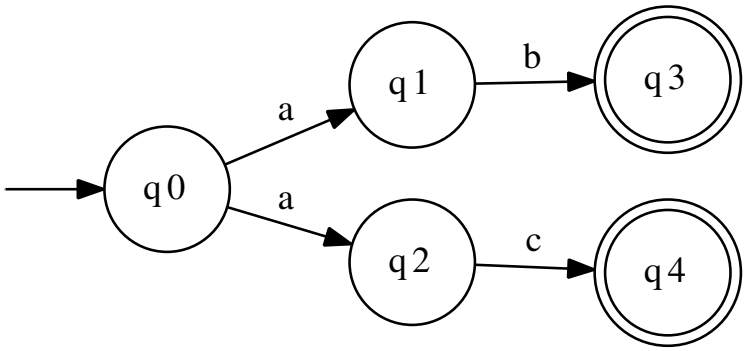
Definícia 3.1 *Nedeterministický konečný automat (NKA) je pätica* $M = (Q, \Sigma, \delta, q_0, F)$, *kde [22]*

- Q je konečná množina stavov
- Σ je konečná vstupná abeceda
- δ je zobrazenie $Q \times \Sigma \to 2^Q$
- $q_0 \in Q$ je počiatočný stav
- $F \subseteq Q$ je množina koncových stavov

Príklad 3.1 *Nech* `/^ab|ac/` *je regulárny výraz rv. Nedeterministický konečný automat, ktorý z neho zostrojíme bude* $M = (Q, \Sigma, \delta, q_0, F)$, *kde*

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $\Sigma = \{a, b, c\}$
- $\delta(q_0, a) = \{q_1, q_2\}$
  $\delta(q_1, b) = \{q_3\}$
  $\delta(q_2, b) = \{q_4\}$
- $F = \{q_3, q_4\}$

Obrázok 3.1 zobrazuje grafickú reprezentáciu automatu M.



Obrázek 3.1: Nedeterministický konečný automat reprezentujúci regulárny výraz `/^ab|ac/`

Z definície 3.1 vyplýva, že výstupom δ môže byť množina stavov, čo znamená, že KA je nedeterministický. Teda prechodom z jedného stavu sa môžeme dostať do viacerých stavov, v limitnom prípade až do všetkých stavov z množiny Q.

Pri simulácii na paralelnej architektúre nás nedeterministická vlastnosť príliš netrápi, v každom kroku (načítanie ďalšieho znaku z reťazca) počítame prechodovú funkciu pre každý stav paralelne, a to v konštantnom čase. Teda prečítanie reťazca znakov bude mať lineárnu zložitosť. Na druhej strane narazíme na podstatné obmedzujúce problémy paralelných architektúr:

- Slabé pamäťové schopnosti - na stavy a prechody sa spotrebuje podstatná časť zdrojov,
- Pomalá rekonfigurácia čipu - prejaví sa pri zmene pravidla/pravidiel.

Ak by sme chceli nedeterministický konečný automat simulovať na sériovej architektúre (napr. architektúra x86), tak sa na zložitosti prechádzania vstupného reťazca podpíše fakt, že pri každom znaku je potrebné spočítať nasledujúce prechody pre všetky aktívne stavy, čo v najhoršom prípade bude mať zložitosť $O(n)$ pre prečítaný znak, kde n je počet aktívnych stavov. Ak to spojíme s prechádzaním vstupného reťazca, ktoré má zložitosť $O(m)$, kde m je počet znakov spracovávaného reťazca, tak dostaneme celkovú zložitosť $O(n.m)$. Takáto zložitosť pri detekcii znamená, že môže nastať situácia, kedy IDS nebude schopné spracovávať dáta na rýchlosti linky a bude teda náchylné na DOS útoky. Na DOS útoky je náchylný napr. Snort, ktorý je možné ho zahltiť už pri rýchlosti 4 kbps [13]. Nedeterminizmus je možné odstrániť za cenu zvýšeného počtu stavov.

3.2.2 Deterministický konečný automat

Každý nedeterministický konečný automat je možné previesť na deterministický. Algoritmus determinizácie je relatívne jednoduchý, ale jeho časová aj priestorová náročnosť môže byť až exponenciálna (vzhľadom na počet stavov). Výhodnosť DKA oproti NKA je závislá na konkrétnom pravidle, pri niektorých pravidlách môže zložitosť KA narásť podstatne a pri niektorých môže zase klesnúť.

Definícia 3.2 *Deterministický konečný automat (DKA) je pätica* $M = (Q, \Sigma, \delta, q_0, F)$, *kde* [22]

- Q je konečná množina stavov
- Σ je konečná vstupná abeceda
- δ je zobrazenie $Q \times \Sigma \to Q$
- $q_0 \in Q$ je počiatočný stav
- $F \subseteq Q$ je množina koncových stavov

Po definícii deterministického konečného automatu je nutné uviesť aj algoritmus, ktorý prevedie nedeterministický konečný automat na ekvivalentný deterministický konečný automat.

Algoritmus 3.1 Prevod nedeterministického konečného automatu na ekvivalentný deterministický konečný automat [22]

Vstup: Nedeterministický konečný automat $M = (Q, \Sigma, \delta, q_0, F)$
Výstup: Deterministický konečný automat $M' = (Q', \Sigma, \delta', q_0', F')$
Metóda:

1. Polož $Q' = (2^Q \setminus \{\emptyset\}) \cup \{nedef\}$
2. Polož $q_0' = \{q_0\}$
3. Pre všetky $S \in 2^Q \setminus \{\emptyset\}$ a pre všetky $a \in \Sigma$ polož $\delta'(S, a) = \bigcup_{q \in S} \delta(q, a)$. Ak je $\delta'(S, a) = \emptyset$, polož $\delta'(S, a) = nedef$
4. Polož $F' = \{S | S \in 2^Q \wedge S \cap F \neq \emptyset\}$

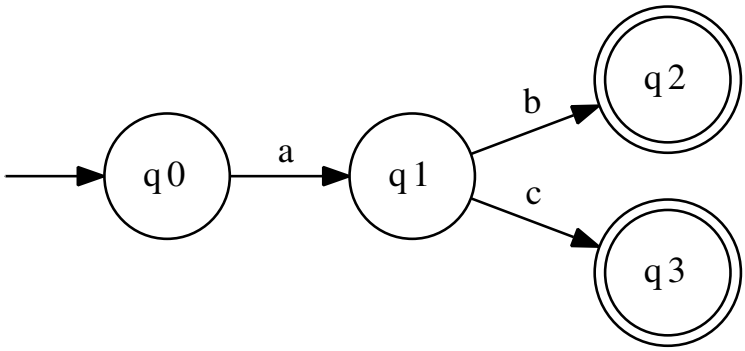
Príklad 3.2 *Nech* `/^ab|ac/` *je regulárny výraz* rv. *Deterministický konečný automat, ktorý z neho zostrojíme bude* $M(rv) = (Q, \Sigma, \delta, q_0, F)$, *kde*

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b, c\}$
- $\delta(q_0, a) = q_1$
  $\delta(q_1, b) = q_2$
  $\delta(q_1, c) = q_3$
- $F = \{q_2, q_3\}$

Obrázok 3.2 zobrazuje grafickú reprezentáciu automatu M.

Z názvu a definície deterministického KA vyplýva, že z každého stavu môžeme jedným načítaným znakom prejsť do maximálne jedného nasledujúceho stavu. Táto vlastnosť pri simulácii na sériovej architektúre umožňuje implementáciu prechodovej funkcie, ktorá bude pracovať v konštantnom čase. Pri čítaní reťazca je aktívny maximálne jeden stav, preto získame výsledok prechodovej funkcie v konštantnom čase. Prečítanie celého vstupného reťazca má lineárnu časovú zložitosť, zistenie ďalšieho stavu trvá konštantný čas, teda detekcia reťazca má celkovo lineárnu časovú zložitosť.

Spomenuli sme, že po odstránení nedeterminizmu z NKA môže narásť počet stavov. V niektorých prípadoch sa môže jednať o kvadratický nárast a v niektorých dokonca o nárast exponenciálny. Na oba prípady sa pozrieme zblízka.



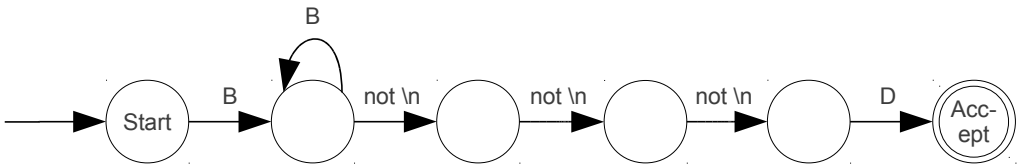
Obrázek 3.2: Deterministický konečný automat reprezentujúci regulárny výraz `/^ab|ac/`

3.2.3 Kvadratický nárast stavov po determinizácii

Medzi nevýhody determinizácie patrí, už spomenutá, časová a priestorová náročnosť. Ak hovoríme o počte stavov tak nám po determinizácii môže kvadraticky narásť počet stavov. Tento prejav si ukážeme na nasledujúcom príklade. Najprv skonštruujeme NKA, potom ho z prevedieme na DKA a ukážeme ako narástol počet stavov. Taktiež vysvetlíme, prečo k tomuto stavu prišlo.

Príklad 3.3 *Nech* `/^B+[^\n]{3}D/` *je regulárny výraz. Tento regulárny výraz vyjadruje reťazce, ktoré začínajú písmenom* `B`*, ktoré sa bude opakovať minimálne 1-krát a potom reťazec pokračuje tromi znakmi rôznymi od* `\n`*, a na poslednom mieste je znak* `D`*.*

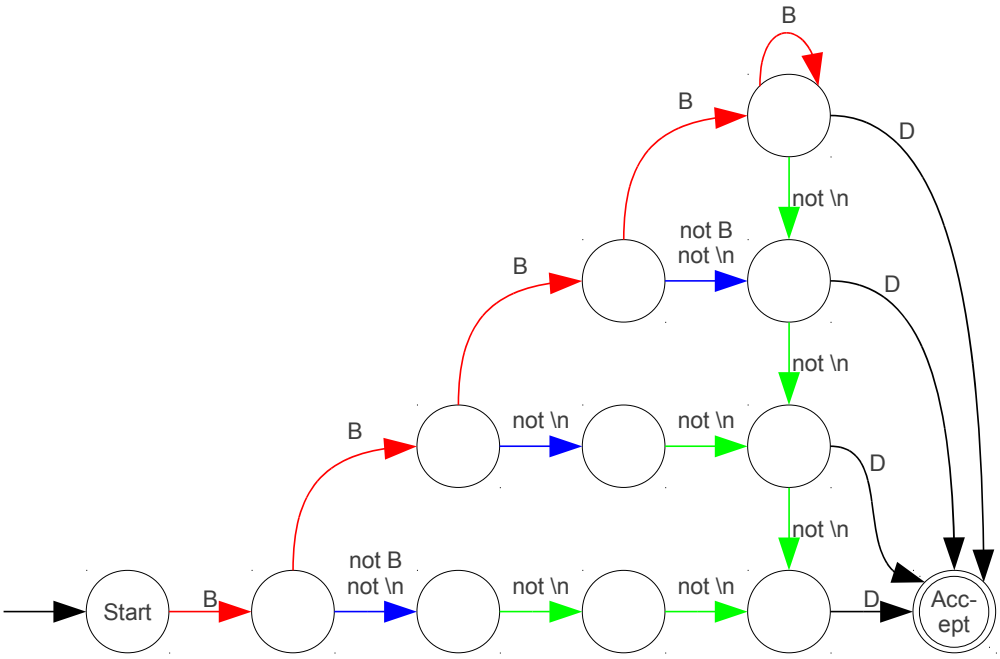
Nedeterministický konečný automat, ktorý z neho zostrojíme bude mať grafickú reprezentáciu zodpovedajúcu obrázku 3.3. Determinizáciou vznikne DKA zhodný s automatom na obrázku 3.4.



Obrázek 3.3: Nedeterministický konečný automat reprezentujúci regulárny výraz `/^B+[^\n]{3}D/`

Vytvorený NKA na obrázku 3.3 má 5 stavov, nie je vôbec zložitý a intuitívne cítime, že vyjadruje daný regulárny výraz. Tento automat zdeterminizujeme, výsledok je na obrázku 3.4.

Oproti NKA nám výrazne narástol počet stavov z 5 pôvodných na 12. Za nárast je zodpovedný tvar regulárneho výrazu: použitie znaku `B` a za ním pripojené `[^\n]]` (tiež obsahuje `B`). Znak `B` sa prekrýva s nasledujúcou množinou znakov, ktorá je použitím intervalu {3} naklonovaná 3-krát za sebou (viz obrázok 3.3). Nárast stavov je kvadratický v závislosti



Obrázek 3.4: Deterministický konečný automat reprezentujúci regulárny výraz `/^B+[^\n]{3}D/` [21]

na hodnote intervalu {n}, teda v tomto prípade 3^2. Jednoducho povedané, DKA musí mať oproti NKA stavy naviac, pomocou ktorých si pamätá, či ešte stále prijíma znaky `B` z `B+` alebo už je na niektorej z troch možných pozícií v `[^\n]{3}`. Napríklad pri reťazci BBCCC by sme po prečítaní BB nevedeli, či detekujeme druhé `B` v `B+` alebo či už detekujeme prvé `[^\n]{3}`.

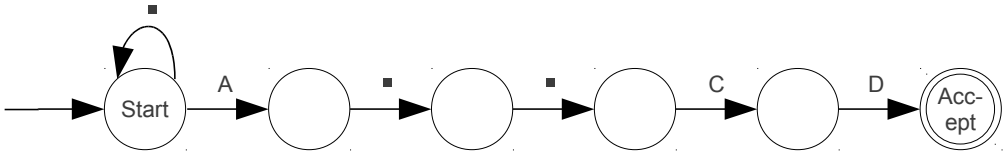
Podobnú štruktúru nájdeme aj v reálnych pravidlách IDS systémov, takže sa nejedná o teoretický hraničný prípad. Konkrétne sa jedná o výraz zo Snortu. Pravidlo pre NNTP je `/^SEARCH\s+[^\n]{1024}/`. V tomto prípade je počet stavov, ktoré vzniknú po determinizácii 1024^2, čo je obrovský nárast.

3.2.4 Exponenciálny nárast stavov po determinizácii

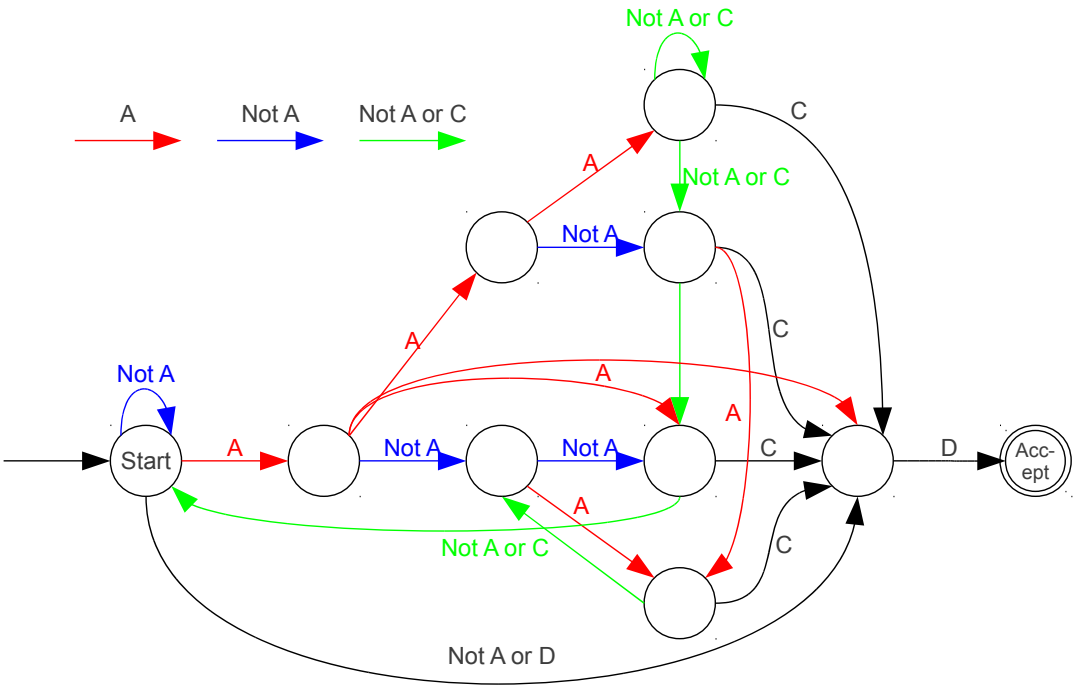
Počet stavov nám môže taktiež narásť exponenciálne. Teoreticky to je možné tak, že pri determinizácii sa môžeme za výsledný deterministický stav považovať stavy reprezentujúce všetky podmnožiny množiny Q. Takýchto podmnožín môže byť až $2^{|Q|}$. Prakticky si to predvedieme na príklade.

Príklad 3.4 *Nech* `/.*A..CD/` *je regulárny výraz. Tento regulárny výraz vyjadruje reťazce, ktoré majú na začiatku akýkoľvek reťazec (aj prázdny), potom je znak* `A`*, ďalej 2 ľubovoľné znaky a končí reťazcom* `CD`*.*

Nedeterministický konečný automat, ktorý z neho zostrojíme, bude mať grafickú reprezentáciu zodpovedajúcu obrázku 3.5. Determinizáciou nám vznikne DKA zhodný s automatom na obrázku 3.6.



Obrázek 3.5: Nedeterministický konečný automat reprezentujúci regulárny výraz `/.*A..CD/`



Obrázek 3.6: Deterministický konečný automat reprezentujúci regulárny výraz `/.*A..CD/` [21]

Nedeterministický KA (viz obrázok 3.5) je znovu jednoduchý a je ľahké si ho spojiť s daným regulárnym výrazom. Obsahuje iba 6 stavov a 6 prechodov. Po determinizácii však dostaneme komplexný a neprehľadný konečný automat (viz obrázok 3.6). Na prvý pohľad sa jedná o rovnaký problém ako v príklade 3.3, avšak problémy naviac spôsobuje znak `A`, ktorý sa nachádza medzi "`.*`" a dvomi wildcard znakmi "`..`". Determinizovaný automat si potom nepamätá iba to, či sa nachádza pred alebo po načítaní `A`, ale musí si pamätať všetky možné kombinácie stavov v časti "`..`", ktoré sa nám môžu teoreticky objaviť v NKA. Napríklad pri prečítaní `AAB` v NKA máme aktívne iné stavy, ako pri prečítaní `ABA`. DKA si doslova pamätá predchádzajúce pozície znaku `A` v načítanom reťazci. V tomto prípade to znamená nárast o 2^{2+1} stavov. Všeobecne nárast stavov podobných konštrukciách bude 2^j,

kde j je počet wildcard (.) znakov.

V reálnych pravidlách IDS systémoch nájdeme podobný regulárny výraz. Konkrétne sa jedná o výraz z pravidiel Snortu na detekciu IMAP protokolu. Reťazec `.*AUTH\s[^\n]{100}`. Po determinizácii narastie KA o 10000 stavov!

Na obrázku 3.6 je zobrazený zdeterminizovaný automat z obrázku 3.5.

Pri nasadení IDS, ktorý by simuloval NKA, môže útočník pri použití správneho reťazca zahltiť simulátor tak, že v ňom bude aktívnych mnoho stavov, čo sa odrazí na priepustnosti IDS systému. Použitím deterministických KA tento problém iba prenesieme z časovej oblasti do priestorovej, lebo DKA vyjadrujúce tieto závadné pravidlá budú zase príliš náročné na pamäť systému, i keď to ich priepustnosť neovplyvní. V prípade DKA sa tento problém prejaví už v dobe predspracovania výrazu, na rozdiel od NKA, kedy sa zahltenie prejaví až v dobe útoku.

V tejto kapitole bolo rozobrané, ako sú reprezentované pravidlá v IDS systémoch regulárnymi výrazmi. Regulárny výraz sme previedli na konečný automat, ktorý je vhodný na detekciu. Ukázali sme si, ako NKA previesť na DKA a načrtli problémy s tým spojené. V reálnom IDS systéme je však potreba detekovať celú množinu pravidiel a nielen jedno samostatné pravidlo, preto sa nasledujúca kapitola venuje problémom spojených s detekciou viacerých regulárnych výrazov a načrtnutím riešenia tohto problému.

Kapitola 4

Zhlukovanie regulárnych výrazov

Predchádzajúca kapitola sa zaoberala analýzou zložitosti DKA vytvorených z jednotlivých regulárnych výrazov. V tejto kapitole predstavíme zhlukovanie viacerých regulárnych výrazov, ktoré má za cieľ urýchliť rozpoznávanie vzorov pomocou deterministických konečných automatov.

4.1 Zložitosť spracovania textu

Zložitosť spracovania textu záleží od použitej architektúry a taktiež od použitého algoritmu. Uvažujeme sériovú architektúru a použitie konečných automatov. Potom je zložitosť ovplyvnená tým, či použijeme DKA alebo NKA a ďalej tým, či budeme rozpoznávať viac RV naraz alebo len jeden. Rozdiel zložitostí pri spracovaní pomocou DKA alebo NKA bol vysvetlený už v predchádzajúcej kapitole. Teraz si ukážeme nárast zložitosti pri rozpoznávaní viacerých výrazov.

Nižšie uvedené zložitosti vyjadrujú časovú zložitosť spracovania jedného znaku zo vstupného reťazca a nie spracovania celého reťazca. Dôvodom je vlastnosť sieťovej prevádzky, kde nás nezaujíma počet spracovaných reťazcov, ale rýchlosť načítania jednotlivých znakov, teda priepustnosť sledovanej prevádzky.

Pre hľadanie vzorov m regulárnych výrazov je na sériovej architektúre pomocou konečných automatov možné zvoliť dva hraničné prístupy. Prvým je použitie m deterministických KA reprezentujúcich regulárne výrazy, ktoré spojíme do jedného NKA, pričom nám nenarastie počet stavov, ale rýchlosť spracovania ovplyvní počet aktívnych stavov, ktorý narastie lineárne vzhľadom na m, lebo v každom čiastkovom DKA môže byť aktívny stav. Ak budeme jednotlivé výrazy reprezentovať ako NKA, tak bude priestorová zložitosť menšia, ale narastie časová zložitosť, pretože v každom čiastkovom NKA môže byť viac ako jeden aktívny stavov.

Druhým prístupom je vytvorenie DKA z predchádzajúceho NKA, ktorý obsahoval všetky výrazy. Pri tomto prístupe bude rýchlosť spracovania jedného znaku konštantná, na druhej strane nám však môže dramaticky narásť počet stavov exponenciálne a to nielen vzhľadom na m, ale aj na počet stavov čiastkových výrazov. Tabuľka 4.1 obsahuje teoretické zložitosti uvedených prípadov.

Prístup pomocou DKA má menšiu časovú zložitosť ako prístup pomocou NKA, na druhej strane má zase prístup pomocou NKA menšiu priestorovú zložitosť ako prístup pomocou DKA. Bez ohľadu na to, ktorý prístup použijeme, výsledok môže byť náročný na použitie, či už sa jedná o zložitosť časovú alebo priestorovú.

	Jeden regulárny výraz dĺžky n		m regulárnych výrazov spojených do jedného KA	
	Časová zložitosť spracovania	Priestorová zložitosť	Časová zložitosť spracovania	Priestorová zložitosť
NKA	$O(n^2)$	$O(n)$	$O(n^2m)$	$O(nm)$
DKA	$O(1)$	$O(\Sigma^n)$	$O(1)$	$O(\Sigma^{nm})$

Tabulka 4.1: Porovnanie zložitostí rôznych prístupov [21]

V ostatnom texte budeme pod pojmom spájanie regulárnych výrazov rozumieť zjednotenie ich konečných automatov a ich následná determinizácia. Ideálnym riešením by sa mohlo zdať použitie jedného DKA reprezentujúceho všetky RV, ale pri spájaní výrazov môže podstatne narásť počet stavov takéhoto automatu. Konkrétny príklad nárastu stavov pri spájaní je prezentovaný v príklade 4.1. Pri spájaní však môže počet stavov aj klesnúť, čo by znamenalo zlepšenie v časovej aj priestorovej oblasti. Najlepšou cestou by bolo spájať iba výrazy, ktoré nám prinesú spomínaný úžitok a nespájať výrazy, ktorých výsledkom by bol priestorovo náročný automat.

Keďže v nasledujúcom príklade bude použité spojenie dvoch regulárnych výrazov, je nutné zadefinovať operáciu, ktorá reprezentuje spojenie regulárnych výrazov na úrovni konečných automatov, teda operáciu zjednotenia dvoch konečných automatov (Algoritmus 4.1). Pre tento krok bude použité zovšeobecnenie nedeterministického konečného automatu, zadefinujeme si teda aj rozšírený konečný automat v Definícii 4.1. Rozšírený konečný automat je nedeterministický konečný automat, ktorého prechodová funkcia môže obsahovať špeciálny prechodový znak ε, ktorý znamená, že daný prechod môže byť vykonaný aj bez prečítania znaku zo vstupného reťazca. Túto špeciálnu funkciu využijeme práve pri zjednotení dvoch NKA. Taktiež bude nutné zadefinovať algoritmus prevodu rozšíreného konečného automatu na ekvivalentný deterministický konečný automat (Algoritmus 4.2), pre ktorý je kľúčovou funkciou výpočet, ktorý danému stavu určí množinu všetkých stavov, ktoré sú dostupné po ε hranách prechodovej funkcie δ. Označme túto funkciu ako ε-uzáver (Definícia 4.2).

Definícia 4.1 *Rozšírený konečný automat (RKA) je pätica* $M = (Q, \Sigma, \delta, q_0, F)$, *kde [22]*

- Q je konečná množina stavov
- Σ je konečná vstupná abeceda
- δ je zobrazenie $Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$
- $q_0 \in Q$ je počiatočný stav
- $F \subseteq Q$ je množina koncových stavov

Algoritmus 4.1 Zjednotenie dvoch NKA do spoločného RKA

Vstup: Nedeterministické konečné automaty $M_1 = (Q_1, \Sigma_1, \delta_1, q_1^0, F_1)$ a $M = (Q_2, \Sigma_2, \delta_2, q_2^0, F_2)$. Predpokladáme, že $Q_1 \cap Q_2 = \emptyset$, inak premenujeme stavy v Q_2 a upravíme δ_2 tak, aby táto podmienka platila

Výstup: Rozšírený konečný automat $M = (Q, \Sigma, \delta, q^0, F)$

Metóda:

1. $Q = Q_1 \cup Q_2$
2. $\Sigma = \Sigma_1 \cup \Sigma_2$
3. $\delta : Q \times \Sigma \to Q \cup \{nedef\}$ je vypočítaná takto:
   - (a) $\forall q_1 \in Q_1, \forall a_1 \in \Sigma_1, \forall \overline{Q_1} \subseteq (Q_1 \cup \{nedef\}) : \delta_1(q_1, a_1) = \overline{Q_1}$
   - (b) $\forall q_2 \in Q_2, \forall a_2 \in \Sigma_2, \forall \overline{Q_2} \subseteq (Q_2 \cup \{nedef\}) : \delta_2(q_2, a_2) = \overline{Q_2}$
   - (c) $\delta(q_0, \varepsilon) = \{q_0^1, q_0^2\}$
4. $F = F_1 \cup F_2$

Definícia 4.2 *ε-uzáver.*

$$\varepsilon\text{-uzáver}(q) = \{p | \exists w \in \Sigma^* : (q, w) \vdash^* (p, w)\}$$

Funkciu ε-uzáver zovšeobecníme tak, aby argumentom mohla byť množina $T \subseteq Q$:

$$\varepsilon\text{-uzáver}(T) = \bigcup_{s \in T} \varepsilon\text{-uzáver}(s)$$

Algoritmus 4.2 Prevod rozšíreného konečného automatu na ekvivalentný deterministický konečný automat [22]

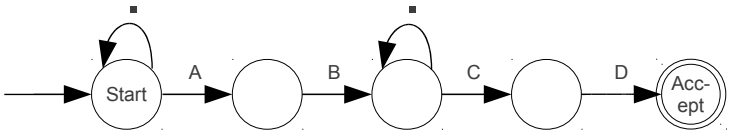
Vstup: Rozšírený konečný automat $M = (Q, \Sigma, \delta, q_0, F)$

Výstup: Deterministický konečný automat $M' = (Q', \Sigma, \delta', q_0', F')$

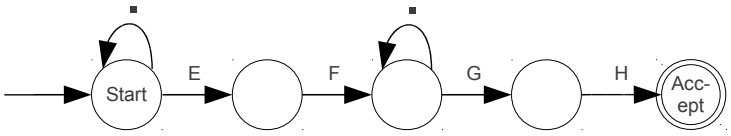
Metóda:

1. $Q' = 2^Q \setminus \{\emptyset\}$.
2. $q_0' = \varepsilon\text{-uzáver}(q_0)$
3. $\delta' : Q' \times \Sigma \to Q' \cup \{nedef\}$ je vypočítaná takto:
   - (a) Nech $\forall T \in Q', a \in \Sigma : \overline{\delta}(T, a) = \bigcup_{q \in T} \delta(q, a)$
   - (b) Potom pre každé $T \in Q', a \in \Sigma$:
     - (i) ak $\overline{\delta}(T, a) \neq \emptyset$, potom $\delta'(T, a) = \varepsilon\text{-uzáver}(\overline{\delta}(T, a))$,
     - (ii) inak $\delta'(T, a) = nedef$
4. $F' = \{S | S \in Q' \wedge S \cap F \neq \emptyset\}$

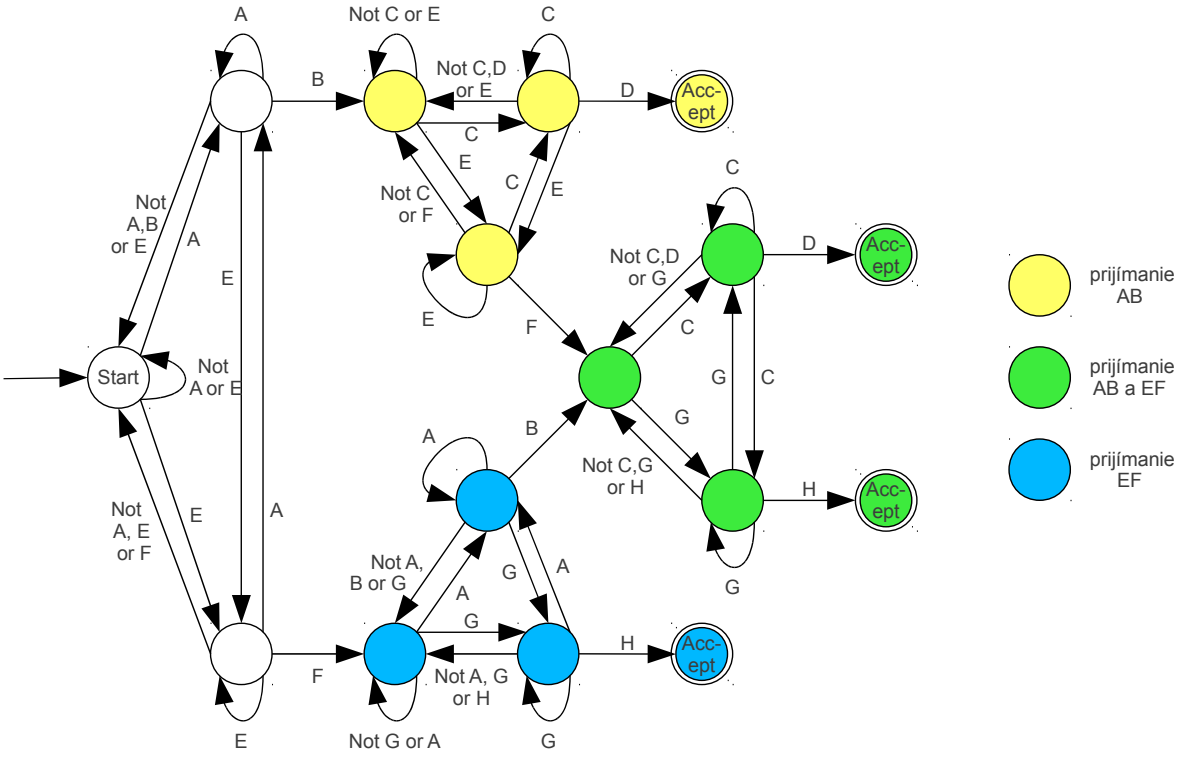
Príklad 4.1 *Nech* `/.*AB.*CD/` *a* `/.*EF.*GH/` *sú regulárne výrazy. Tieto výrazy reprezentujú 2 jednoduché KA (Obrázok 4.1 a 4.2). Po spojení a zdeterminizovaní vznikne DKA na obrázku 4.3.*



Obrázek 4.1: Nedeterministický konečný automat reprezentujúci regulárny výraz `/.*AB.*CD/`



Obrázek 4.2: Nedeterministický konečný automat reprezentujúci regulárny výraz `/.*EF.*GH/`



Obrázek 4.3: Deterministický konečný automat reprezentujúci spojené regulárne výrazy `/.*AB.*CD/` a `/.*EF.*GH/` [21]

Na príklade 4.1 je vidieť, že spájaním regulárnych výrazov do spoločného DKA môže zreteľne narásť počet stavov. V tomto prípade nárast stavov spôsobuje skutočnosť, že DKA musí obsahovať stavy, ktoré nesú informáciu o tom, či bolo prijaté `AB` alebo `EF` (žltá a modrá časť automatu na obrázku 4.3). Do žltého a modrého koncového stavu sa dá dostať prijatím `AB` a `CD`, resp. `EF` a `GH`. Do zelenej časti je možné sa dostať po prijatí dvojice `AB` a `EF` alebo v opačnom poradí po prijatí `EF` a následne `AB`. Tento stav vyjadruje situáciu, kedy sú obe prvé časti (`AB` a `EF`) regulárneho výrazu už prijaté a reťazec prijmeme bez ohľadu na to, či je na konci `CD` alebo `GH`. Vo všeobecnosti je na takúto situáciu potrebných $O(2^n)$ stavov, kde n je počet výrazov a každý výraz obsahuje práve jednu wildcard sekvenciu (`.*`). Ak výrazy obsahujú x wildcard sekvencií, tak je potrebných $(x+1)^n$ stavov.

4.2 Zhlukovanie regulárnych výrazov

Pri detekcii výrazov môžeme použiť dva prístupy založené na DKA. Prvým je jeden DKA zahrňujúci všetky RV alebo použitie n samostatných DKA pre každý výraz zvlášť. Zhlukovanie (angl. grouping) je možné chápať ako strednú cestu medzi spomenutými prístupmi. Zhlukovanie nám ponúka nižšiu priestorovú zložitosť oproti prvému prístupu a nižšiu časovú zložitosť oproti druhému prístupu. Pri spájaní viacerých výrazov nám pri spojení (spojenie viac DKA do jedného NKA a zdetermnizovanie na DKA) môže, ale nemusí narásť počet stavov. Cieľom zhlukovania je snaha spájať iba automaty, pri ktorých nám výrazne nenarastie počet stavov. Reláciu medzi dvomi výrazmi, ktorá vyjadruje či nám výrazne narastie počet stavov, nazvime interakcia.

Definícia 4.3 *Interakcia. Dva regulárne výrazy spolu interagujú ak DKA, ktorý vznikne ich spojením, obsahuje viac stavov ako suma stavov DKA reprezentujúcich samostatné regulárne výrazy. [21]*

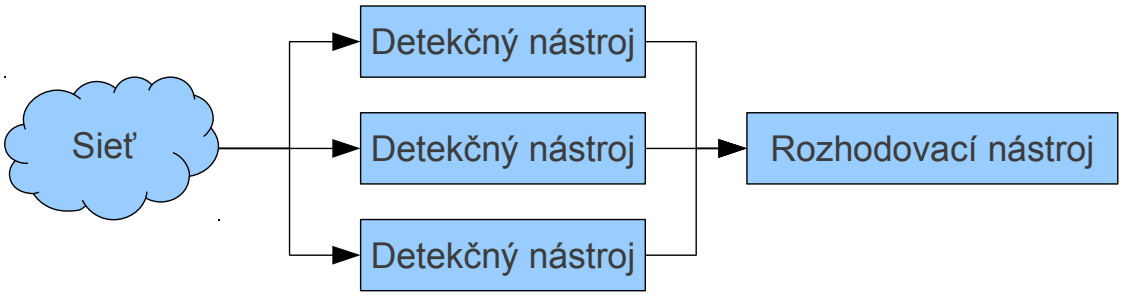
Ak teda spojíme dva regulárne výrazy, ktoré spolu neinteragujú, tak výsledkom bude DKA, ktorý má menej alebo rovnako stavov a pri detekcii bude aktívny len jeden stav. Tým pádom sa znížila časová aj priestorová zložitosť, oproti detekcii každého RV samostatne. Túto vlastnosť využijeme práve pri zhlukovaní, ktorého cieľom je nájsť množiny výrazov, ktorým po spojení a zdeterminizovaní nenarastie počet stavov.

Cieľom zhlukovania je rozdeliť množinu RV na zhluky, pri ktorých nám nenarastie priestorová zložitosť a každý zhluk bude reprezentovaný DKA. Nájsť optimálne riešenie je náročné, museli by sa vytvoriť DKA pre všetky možné zhluky (všetky možné podmnožiny z množiny RV) a sledovať ich nárast stavov. Stavový priestor možných riešení je exponenciálny vzhľadom na počet výrazov, takže naivné prehľadávanie by bolo časovo tiež exponenciálne zložité, keďže počet podmnožín množiny obsahujúcej n prvkov je 2^n. Pri väčšom počte pravidiel nepripadá takáto možnosť v úvahu, preto je nutné použiť algoritmus, ktorý dokáže nájsť riešenie blízko optimálneho riešenia v primeranom výpočetnom čase.

4.3 Príklad použitia zhlukov

Zhluky vytvorené zhlukovacím algoritmom možno využiť v distribuovanej architektúre IDS. Táto architektúra (viz obrázok 4.4) dokáže rozložiť záťaž medzi viacej detekčných nástrojov, v ktorom každý bude detekovať pomocou jedného alebo viacerých deterministických konečných automatov. Predpokladom je, že každý z nástrojov je pripravený práve na detekciu pomocou deterministického konečného automatu. Pod nástrojmi si možno predstaviť

samotné procesy v jednom počítači ale aj samotné počítače spojené pomocou počítačovej siete. Pomocou takejto paralelnej architektúry teda môžeme rozdeliť záťaž na viac strojov a vďaka zhlukovaniu minimalizovať počet takýchto výpočetných jednotiek.



Obrázek 4.4: Paralelná architektúra IDS systému [9]

4.4 Zhlukovací algoritmus

Algoritmus sme prebrali z článku [21] a bude naňho odkazované zo zvyšku práce, kde s ním budu porovnávané iné zhlukovacie algoritmy alebo jeho vylepšenia. Tento algoritmus predpokladá, že interakcia je tranzitívna relácia. Vytvára zhluky výrazov, ktoré obsahujú medzi sebou čo najmenej interakcií. Prvým krokom je teda pre všetky dvojice výrazov vypočítať ich interakcie. Prvý zhluk tvorí jeden výraz, ktorý má najmenej interakcií s ostatnými výrazmi. Do zhluku ďalej pridáva výrazy, ktoré majú najmenej interakcií s výrazmi v zhluku. Pre výrazy obsiahnuté v zhluku skonštruuje spoločný DKA, a ak jeho počet stavov prekročí vopred danú hranicu, tak sa proces vytvárania zhluku opakuje s doposiaľ nezhluknutými výrazmi.

Definícia 4.4 *Neinterakcia. Dva regulárne výrazy spolu neinteragujú práve vtedy, ak neplatí, že je medzi nimi interakcia.*

Obrázok 4.5 demonštruje jeden beh algoritmu 4.3. V prvej fáze je znázornené spočítanie interakcií. Spojnice v obrázku znamenajú, že regulárne výrazy spolu neinteragujú. Vďaka tomu je jednoduchšie popísateľná druhá fáza, kde spojnica (neinterkacia) znamená, že výrazy majú snahu sa spojiť do spoločného zhluku. Ako prvý krok je vytvorenie prvého zhluku, ktorý má najmenej interakcií (najviac neinterakcií) s ostatnými výrazmi. V ďalšom kroku sa spojí s výrazom, ktorý má s ním najviac neintarakcií, na výber má 3, v tomto prípade sa vyberie náhodný. Týmto spôsobom pokračuje algoritmus v ďalšom spájaní sa, až narazí na limit stavov, po ktorom sa vráti k predchádzajúcemu zhluku, ktorý limit nepresahoval. Zo zvyšku sa zase vyberie výraz s najmenším počtom interakcií, ktorý bude tvoriť základ nového zhluku. Tento postup sa opakuje do chvíle, pokým nie sú všetky regulárne výrazy v nejakom zhluku.

Uvažujme, že spájanie a zdeterminizovanie výrazov do jedného DKA je operácia, ktorú je možné vykonať v konštantnom čase. Potom je časová zložitosť algoritmu 4.3 kvadratická.

Algoritmus 4.3 Zhlukovací algoritmus [21]

```
for all regulárny výraz R_i z množiny do
  for all regulárny výraz R_j z množiny do
    Vypočítaj interakciu medzi R_i a R_j
  end for
end for
Zkonštruuj graf G(V, E), kde
  V je množina RV, jeden vrchol pre každý RV
  E je množina hrán medzi vrcholmi; hrana (V_i, V_j) existuje, ak R_i a R_j interagujú
repeat
  Nový zhluk NG = ∅
  Vyber z G výraz R, ktorý má najmenej interakcií s ostatnými výrazmi a pridaj ho do
  nového zhluku NG
  repeat
    Vyber z G výraz R, ktorý má najmenej hrán s NG
    Zostroj DKA pre množinu výrazov NG ∪ {R}
    if DFA je väčšie ako limit then
      break;
    else
      Pridaj R do NG
    end if
  until všetky RV z množiny G sú odskúšané
  Odober NG z množiny G
until G neobsahuje žiadny RV
```

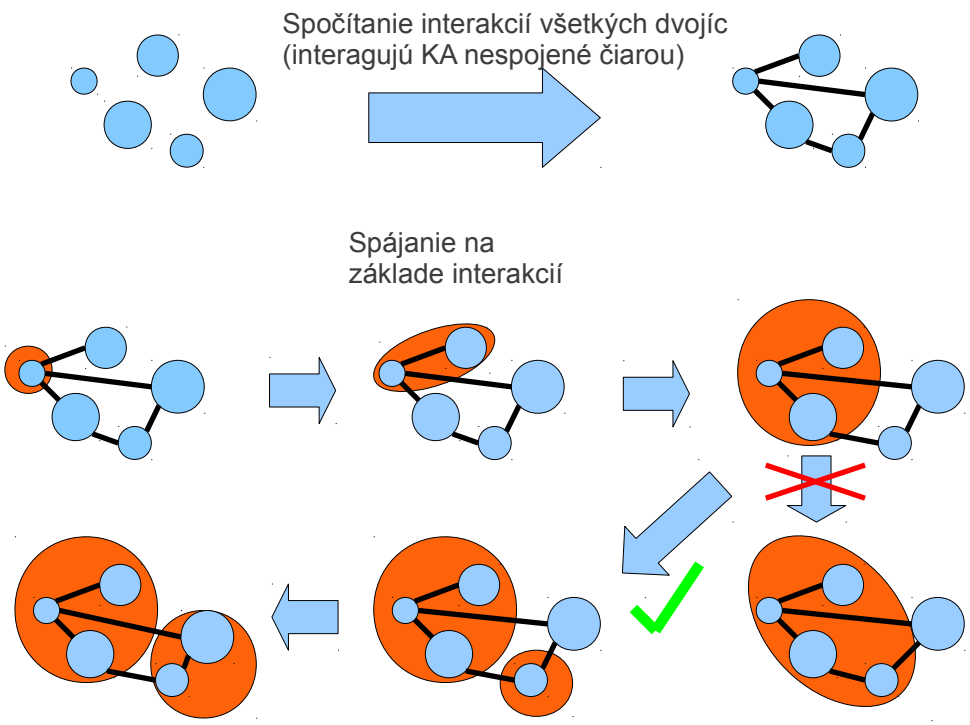
Túto operáciu by bolo možné počítať aj paralelne, keďže vypočítanie interakcie nie je závislé na ďalších výpočtoch. V inicializačnej fáze musí vypočítať interakcie medzi všetkými možnými dvojicami. Zložitosť druhej časti algoritmu je z pohľadu počtu determinizácií lineárna. V najlepšom prípade to bude n determinizácií. Toto by nastalo, ak by veľkosť automatu reprezentujúceho zhluk RV nepresiahla ani raz limit a teda výsledkom by bol jeden zhluk. Najhoršia situácia by nastala v prípade, ak by bol limit presiahnutý v každom cykle, a teda výsledkom by bolo n zhlukov a v každom by sa vykonala determinizácia jedenkrát zbytočne. Časová zložitosť najhoršieho prípadu by teda bola $2n$, čo stále spadá do triedy zložitosti $O(n)$. Nízky limit, ktorý by znamenal, že bude často presiahnutý, teda navýši čas potrebný na výpočet, ale nemá vplyv na jeho teoretickú časovú zložitosť.

Inicializačnú fázu výpočtu interakcií je možné paralelizovať. Množinu všetkých dvojíc je možné rozdeliť na podmnožiny, ktoré by sa dali vypočítať súčasne na viacerých strojoch. Druhá fáza už z princípu paralelizovateľná nie je, keďže každý krok algoritmu je závislý na predchádzajúcich výpočtoch, algoritmus musí v každom cykle vedieť, ktoré zhluky už boli spracované a ktoré ešte spracované neboli.

4.5 Iné metódy zhlukovania

Cieľom práce je navrhnúť nové spôsoby zhlukovania regulárnych, ktoré by boli efektívnejšie ako základný zhlukovací algoritmus. Ponúka sa použitie viacerých metód, ktoré by nám mohli poskytnúť isté výhody.

Prvou metódou by mohlo byť použitie genetických algoritmov alebo genetického progra-



Obrázek 4.5: Príklad zhlukovania regulárnych výrazov

movania. Toto by nám mohlo poskytnúť riešenie, ktoré by bolo robustné vzhľadom na časté zmeny databázy výrazov (pridávanie a odoberanie výrazov). Mohli by sme teda k riešeniu pridať alebo odobrať výrazy a nechať populáciu nájsť nové riešenie, ktoré by nemalo byť príliš odlišné od pôvodného. Ušetril by sa tým čas potrebný k nájdeniu riešenia.

Ďalším zlepšením, ktoré je už implementované, je možnosť zmeny ohodnocovacej funkcie za inú, oproti pôvodnej, ktorá uvažuje len počet stavov. Ďalej by bolo možné pokúsiť sa nájsť situácie, pri ktorých vzniká interakcia a snažiť sa ich predpovedať skôr ako nastane, napríklad už počas determinizácie. Takisto by mohlo byť možné nájsť situácie, pri ktorých vieme ešte pred alebo počas determinizácie určiť, že interakcia dvoch automatov nenastane. Taktiež by mohlo byť možné niektoré z týchto vylepšení použiť nie na dvojicu, ale na väčšiu množinu výrazov a získať tak informácie o vzťahoch regulárnych výrazov z množiny len za pomoci niekoľkých determininizácií.

Cieľom diplomovej práce je navrhnúť ďalšie možnosti a porovnať efektivitu uvedených metód a kvalitu ich riešení a navrhnúť algoritmus, ktorý by dokázal zhlukovať regulárne výrazy v reálnom čase.

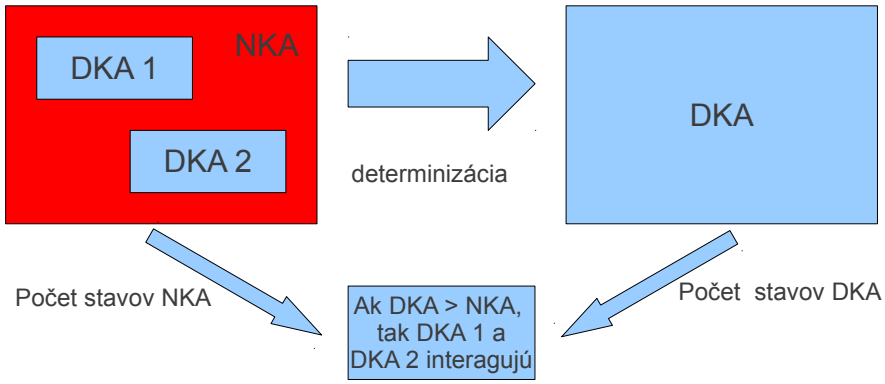
Kapitola 5

Možnosti urýchlenia zhlukovania

Táto kapitola sa bude venovať prvej fáze algoritmu 4.3. Táto fáza má najväčšiu časovú zložitosť vzhľadom k veľkosti množiny regulárnych výrazov, čiže jej úpravy by mali mať najväčší vplyv na celkové urýchlenie zhlukovacieho algoritmu. Budú rozobraté možnosti urýchlenia jednotlivých interakcií a taktiež počítanie viacerých interakcií naraz.

5.1 Výpočet jednej interakcie

Pri počítaní interakcií pre množinu RV je kľúčovou operáciou zistenie interakcie medzi dvomi regulárnymi výrazmi. Pretože je to operácia vykonaná v každom cykle, jej zrýchlenie bude mať vplyv na celkovú rýchlosť spočítania interakcií pre celú množinu. Hlavným cieľom bude analýza slabých miest pri výpočte interakcie, pri ktorých sa prevádzajú zbytočné výpočty. Princíp výpočtu interakcie vyjadruje obrázok 5.1.



Obrázek 5.1: Výpočet interakcie dvoch DKA

5.1.1 Základný výpočet interakcie

Výpočet interakcie pozostáva z prevodu NKA reprezentujúceho dva regulárne výrazy na DKA, preto si definujeme kostru algoritmu, ktorý bude tento prevod vykonávať. Algoritmus 5.1 v prvom kroku pridá do množiny S (množina nespracovaných stavov) počiatočný stav ako jednoprvkovú množinu, keďže stavy v novom automate budú reprezentované množinami

stavov aktívnych v spracovávanom NKA po prijatí nejakého reťazca. V druhom kroku sa spočítajú možné následné stavy a pridáme ich do $newS$. Stavy, do ktorých je možné sa dostať rovnakým symbolom, spojíme do jedného nového stavu. V kroku 3 označíme množinu newS za nespracovanú množinu stavov. Algoritmus končí, ak už neexistujú stavy na spracovanie.

Algoritmus presne nešpecifikuje, ako budovať výsledný automat, ale to nebude z nášho pohľadu podstatné, podstatný je spôsob prechádzania NKA smerom k výslednému DKA.

Algoritmus 5.1 Prevod NKA na ekvivalentný DKA

1. Pridaj $\{q_0\}$ do S
2. $\forall s \in S\ \forall a \in \Sigma$ pridaj $\bigcup_{q \in s} \delta(q, a)$ do $newS$ a pridaj odpovedajúce prechody a stavy do nového automatu
3. $S = newS$
4. Ak $S \neq \emptyset$, pokračuj krokom 2

5.1.2 Detekcia prekročenia stavov

Z algoritmu zhlukovania (algoritmus 4.3) a definície interakcie (Definícia 4.3) vyplýva, že na zistenie interakcie dvoch automatov musíme vytvoriť KA, reprezentujúci tieto dva automaty a následne ho previesť na DKA.

Nech m a n je počet stavov týchto dvoch automatov. Počet stavov DKA môže mať až mn stavov, teda prevod na DKA bude mať časovú zložitosť $O(mn)$. Keďže je už pred determinizáciou jasný limit stavov (konkrétne $m + n$), ktorého prekročenie znamená interakciu, môžeme pri prekročení tejto hodnoty ukončiť determinizáciu a dvojicu označiť za interagujúcu. Dopočítanie DKA by nemalo zmysel, keďže výstupom je iba informácia o interagovaní dvoch automatov. Touto jednoduchou úpravou sa zníži časová zložitosť vypočítania interakcie z $O(mn)$ na $O(m+n)$. Túto kontrolu je možné jednoducho zakomponovať do algoritmu 5.1 medzi krokom 2 a krokom 3, výsledok reprezentuje algoritmus 5.2.

Algoritmus 5.2 Prevod NKA na ekvivalentný DKA

1. Pridaj $\{q_0\}$ do S
2. $\forall s \in S\ \forall a \in \Sigma$ pridaj $\bigcup_{q \in s} \delta(q, a)$ do $newS$ a pridaj odpovedajúce prechody a stavy do nového automatu
3. Ak je počet stavov vo výslednom DKA presiahol limit, zastav determinizáciu a označ ho za interagujúci
4. $S = newS$
5. Ak $S \neq \emptyset$, pokračuj krokom 2

5.1.3 Detekcia interakcie

Ako bolo spomenuté, interakcia regulárnych výrazov znamená, že spojením a zdeterminizovaním konečných automatov reprezentujúcich tieto výrazy, narastie počet ich stavov. Z toho vyplýva, že determinizácia musí dobehnúť do konca alebo sa zastaviť na limite stavov, ako je opísané v predchádzajúcom prístupe. Bolo by pozitívne, ak by bolo možné interakciu výrazov detekovať ešte pred dokončením algoritmu determinizácie alebo ešte pred dosiahnutím limitu stavov. Nasledujúci prístup ukazuje, ako to je možné zrealizovať.

Znovu predpokladajme, že determinizujeme dva spojené DKA pomocou algoritmu 5.1. Zaveďme novú premennú *computed_states*, ktorá bude obsahovať počet stavov, ktorú sme už v priebehu výpočtu zo vstupného automatu spracovali. Nech *generated_states* je počet stavov v generovanom determininistickom konečnom automate. Ak v priebehu determinizácie nastane situácia, že $generated\_states > computed\_states$, môžeme determinizáciu ukončiť a vstupné výrazy označiť za interagujúce. Tento postup reprezentuje algoritmus 5.3.

Algoritmus 5.3 Prevod NKA na ekvivalentný DKA

1. Pridaj $\{q_0\}$ do S
2. $\forall s \in S\ \forall a \in \Sigma$ pridaj $\bigcup_{q \in s} \delta(q, a)$ do $newS$ a pridaj odpovedajúce prechody a stavy do nového automatu
3. Aktualizuj premenné *computed_states* a *generated_states*
4. Ak $generated\_states > computed\_states$, zastav determinizáciu a označ vstupné výrazy za interagujúce
5. $S = newS$
6. Ak $S \neq \emptyset$, pokračuj krokom 2

5.1.4 Detekcia neinterakcie

Pri determinizácii je možné detekovať ešte jednu vlastnosť spracovávaného KA. Ide o detekciu, keď pri spracovávaní nespracovanej časti NKA už nemôže narásť počet stavov. Myšlienka je v tom, že aktuálna množina nespracovaných stavov už neobsahuje také stavy, ktoré by znamenali, že sme v oboch DKA zároveň. Z takýchto stavov už neexistuje možnosť, aby sme sa znovu dostali do oboch automatov zároveň. Konkrétne je to možné zistiť tak, že každý nespracovaný stav nereprezentuje viac ako 1 stav v NKA, teda sa nachádza buď v jednom alebo druhom DKA. Kontrolu tejto podmienky je možné zase pridať do algoritmu 5.1 medzi kroky 2 a 3, a formálne zapísať ako $\forall s \in S : |s| \leq 1$. Výsledný algoritmus reprezentuje algoritmus 5.4.

Algoritmus 5.4 Prevod NKA na ekvivalentný DKA

1. Pridaj $\{q_0\}$ do S
2. $\forall s \in S\ \forall a \in \Sigma$ pridaj $\bigcup_{q \in s} \delta(q, a)$ do $newS$ a pridaj odpovedajúce prechody a stavy do nového automatu
3. Ak $\forall s \in S : |s| \leq 1$ zastav determinizáciu a označ automat za neinteragujúci
4. $S = newS$
5. Ak $S \neq \emptyset$, pokračuj krokom 2

5.2 Detekcia viacerých interakcií naraz

Doteraz bola spomenutá iba interakcia dvojíc regulárnych výrazov, avšak pri výpočte všetkých interakcií je možné zistiť interakciu alebo neinterakciu celej množiny výrazov. V takom prípade sa teda ušetrí čas, pretože výpočet interakcií pre celú množinu regulárnych výrazov je možné nahradiť menším počtom determininizácií. Problémom je však predpovedať tú "správnu" množinu, pre ktorú platí, že všetky jej prvky interagujú/neinteragujú. Práve preto pri nami navrhnutom algoritme nezáleží na poradí pri vyberaní regulárnych výrazov.

Skúšali sme vyberať výrazy zoradené podľa veľkosti ich automatov, ale na výsledný čas to nemalo zásadný vplyv.

5.2.1 Cyklus v automate

V istých prípadoch dokážeme už pred determinizáciou určiť, že daný regulárny výraz nebude interagovať zo žiadnym iným výrazom. V podkapitolách 3.2.3 a 3.2.4 boli spomenuté príklady veľkého nárastu stavov, či už pri determininizácii jedného automatu alebo determininizácii dvoch spojených automatov. V oboch prípadoch dochádzalo k nárastu stavov vďaka cyklom v automate, ktoré sa pri determinizácii prejavili generovaním nadmerného množstva stavov vo výstupnom automate.

Uvažujme teda, že aby došlo k nárastu stavov pri determinizácii dvoch DKA spojených do spoločného NKA, tak v oboch automatoch sa musí nachádzať cyklus. Nárast stavov síce môže nastať aj bez cyklu, ale nemal by byť taký dramatický ako nárast pomocou cyklu. Idea je v tom, že cyklus dokáže generovať nové stavy stále dokola, no v prípade automatu bez cyklu sa do tohto problémového stavu nemôžeme vrátiť. Teda problémový stav môže zvýšenie počtu stavov zaviniť iba raz.

Z vyššie uvedeného plynie poznatok, že pokiaľ automat neobsahuje cyklus, tak by nemal interagovať so žiadnym iným konečným automatom (regulárnym výrazom). Uvažujme, že pred spočítaním interakcií pre množinu, ktorá obsahuje m regulárnych výrazov detekujeme n regulárnych výrazov, ktorých konečné automaty neobsahujú cyklus. Tým by sa mala znížiť zložitosť výpočtu všetkých interakcií z $O(m)$ na $O(m - n)$. Na druhej strane treba podotknúť, že takýchto regulárnych výrazov nie je príliš veľa, ale ak vieme, či automat cyklus obsahuje alebo nie, môže byť zbytočné počítať preňho interakciu so všetkými ostatnými regulárnymi výrazmi.

5.2.2 Neinterakcia viacerých RV naraz

V tomto prístupe využijeme detekciu prekročenia počtu stavov z kapitoly 5.1.2. Prístup je založený na myšlienke, že ak spojíme viacero automatov do spoločného NKA, a pri determinizácii nebude prekročený limit daný súčtom stavov jednotlivých vstupných automatov, žiadna dvojica z týchto automatov nie je v interakcii. Týmto spôsobom je možné redukovať počet determinizácií vzhľadom k počtu získaných interakcií. Spojením n automatov pomocou $n - 1$ determinizácií dostaneme informáciu o $n^2/2$ interakciách, kým pri klasickom algoritme musíme pre získanie interakcií vykonať n^2 determinizácií.

Algoritmus 5.5 vytvára postupne automat, ktorý obsahuje navzájom neinteragujúce výrazy. Ak sa mu nepodarí pripojiť nový regulárny výraz, preskočí ho a spracuje ho v ďalšom cykle s ostatnými preskočenými výrazmi. Cyklus končí, ak sú spracované všetky výrazy a žiadny nebol preskočený. Dvojice, ktoré sa nachádzajú spoločne v niektorom $group \in groups$, spolu neinteragujú. Ostatné dvojice je nutné dopočítať pre každú dvojicu zvlášť.

Aby sme mohli spomenuté prístupy porovnať, všetky podrobíme testovaniu na rôznych sadách pravidiel a budeme skúmať, či sa potvrdia teoretické predpoklady, teda či algoritmy budú viesť k lepšiemu výpočetnému času a výsledky budú porovnateľné s klasickým zhlukovacím algoritmom.

Algoritmus 5.5 Neinterakcia viacerých RV naraz

S je množina všetkých regulárnych výrazov
$automaton$ je prázdny automat
groups $= \emptyset$ je množina neinteragujúcich množín
while S $\neq \emptyset$ **do**
  nextS $= \emptyset$
  for all a in S **do**
    $group = \emptyset$
    $new\_automaton$ = spojené a zdeterminizované $(automaton + a)$
    $limit$ = počet stavov $automaton$ + počet stavov a
    if $new\_automaton$ presiahlo $limit$ **then**
      $nextC = nextC \cup \{a\}$
      continue
    else
      a$utomaton = new_automaton$
      $group = group \cup \{a\}$
    end if
  end for
  $groups = groups \cup group$
  $S = nextS$
end while
Všetky dvojice, ktoré sa nachádzajú v $group \in groups$ spolu, neinteragujú.
Pre všetky dvojice výrazov, ktoré sa nenachádzajú spoločne v žiadnej $group \in groups$, dopočítaj interakcie klasicky.

5.3 Výsledky experimentov

V tejto časti si ukážeme výsledky testov pre spomenuté algoritmy, ktoré by mali urýchliť zhlukovací algoritmus. Zavedieme testovaciu množinu, na ktorej budú vykonané testy všetkých algoritmov. Výsledky testov porovnáme s výsledkami klasického riešenia. Najprv ukážeme rozdiel medzi klasickým zhlukovaním a zhlukovaním, kde sa pri počítaní interakcií používa limit počtu stavov. Všetky ďalšie algoritmy už budú porovnávané voči variante s limitom počtom stavov, nie voči klasickej variante, pretože sú to algoritmy, ktoré sa prejavia ešte pred dovŕšením limitu počtu stavov.

5.3.1 Testovacia množina

Ako testovaciu množinu sme zvolili súbory s výrazmi dostupnými v knižnici Netbench [20]. Každý súbor obsahuje množinu výrazov, ktoré spolu súvisia, teda je pravdepodobné, že v takejto zostave by mohli byť použité aj pri reálnej prevádzke. Z testov sme vyradili súbory, ktoré obsahovali menej ako 10 regulárnych výrazov. Kompletný prehľad o použitých súboroch s regulárnymi výrazmi je uvedený v tabuľke 5.1.

Testovacia množina obsahuje súbory s rozličnými vlastnosťami. Čas spracovania súboru nie je závislý iba na počte regulárnych výrazov, ktoré obsahuje, závisí totiž aj na počte výrazov obsahujúcich cyklus, na podiele interagujúcich dvojíc, na počte stavov, atď.

Množina regulárnych výrazov	Počet výrazov	Bez cyklu	Podiel bez cyklu	Podiel interakcií
backdoor.rules.pcre	154	12	7,79%	13,19%
chat.rules.pcre	14	1	7,14%	42,85%
dpd.reg	17	6	35,29%	20,76%
exploit.rules.pcre	40	20	50,00%	62,12%
ex.web-rules.reg	248	0	0,00%	5,05%
ftp.rules.pcre	34	4	11,76%	8,13%
imap.rules.pcre	28	1	3,57%	87,24%
misc.rules.pcre	15	3	20,00%	5,00%
netbios.rules.pcre	16	14	87,50%	15,62%
oracle.rules.pcre	12	1	8,33%	90,28%
pop3.rules.pcre	16	10	62,50%	1,56%
smtp.rules.pcre	35	5	14,28%	25,14%
snort-default.reg	743	0	0,00%	40,24%
spyware-put.rules.pcre	460	2	0,43%	26,50%
voip.rules.pcre	36	2	5,55%	11,57%
web-client.rules.pcre	30	2	6,66%	50,67%
web-misc.rules.pcre	50	6	12,00%	72,88%
web-php.rules.pcre	16	1	6.25%	50,00%

Tabulka 5.1: Prehľad a vlastnosti použitých množín pravidiel

5.3.2 Metodika testovania

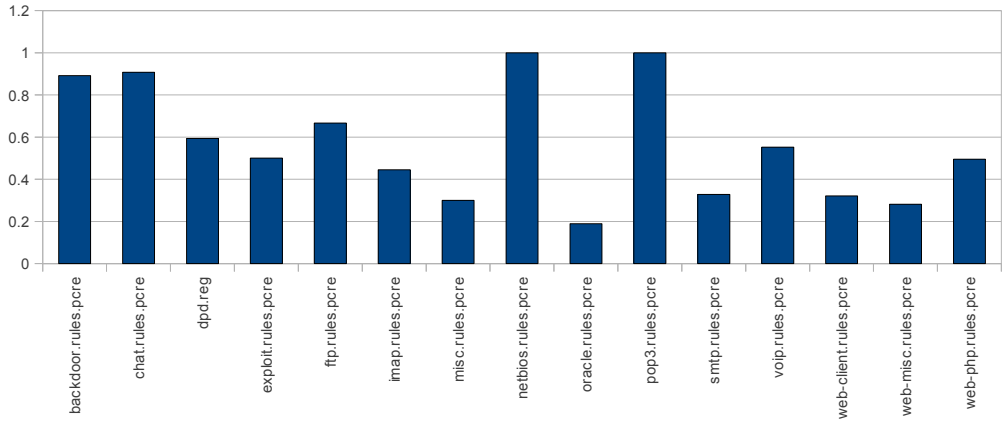
Na každý súbor s regulárnymi výrazmi sme aplikovali sadu testov. Všetky súvisiace testy boli spustené za sebou na tom istom počítači, aby sa vylúčili vonkajšie vplyvy, ktoré by mohli do výsledkov vniesť chybu. K testovaniu sme použili 6 počítačov NetFPGA Cube zapožičaných na fakulte, ktoré poskytli dostatočný výkon. Keďže prístup k počítačom sem mali výhradne my, môžeme vylúčiť vplyv cudzích procesov na výsledky.

Použitým operačným systémom bol Red Hat Enterprise Linux. Použitou verziou interpretu jazyka Python bol Python2.6.

Pri všetkých testoch bol použitý limit počtu stavov na jeden zhluk (povinný parameter pre algoritmus 4.3) na 2000 stavov. Uvedený limit je strednou cestou pre všetky množiny pravidiel tak, aby sme pri testovaní dosiahli čo najlepšie rozloženie automatov do výsledných zhlukov. Príliš vysoký limit by spôsobil, že všetky pravidlá skončia iba v jednom alebo niekoľko málo zhlukoch, naopak limit príliš nízky by sa prejavil tak, že každý automat by bol vo vlastnom zhluku, a teda by sme nedosiahli žiadny rozumný výsledok. Hodnota 2000 stavov vyplynula z práce so zhlukovacím algoritmom a množinami výrazov a nie je teda výsledkom žiadnej metodiky.

5.3.3 Detekcia prekročenia stavov

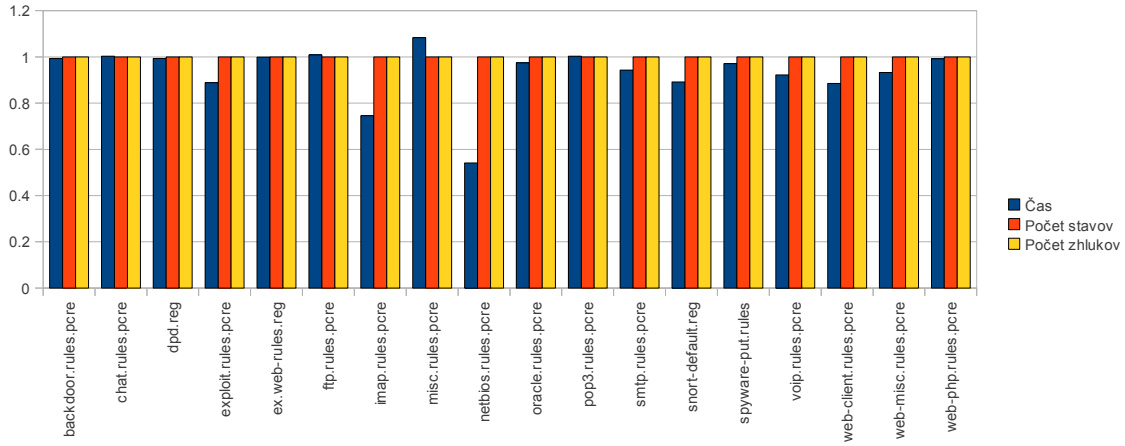
Detekcia prekročenia počtu stavov sa ukázal ako správny prístup, ktorý dokáže urýchliť spočítanie interakcií. Z grafu na obrázku 5.2 je jasne vidieť, že spočítanie interakcií trvalo pre väčšinu množín kratší čas. V prípade sady pravidiel `oracle.rules.pcre` trval výpočet dokonca 20% času oproti variante bez limitu stavov.



Obrázek 5.2: Porovnanie výpočtu interakcií pri dobehnutí celej determinizácie s variantou, ktorá výpočet ukončí po dosiahnutí limitného počtu stavov. Hodnota na ose y vyjadruje pomer času výpočtu algoritmu pomocou zavedenia limitu počtu stavov.

5.3.4 Detekcia interakcie

Porovnanie výsledkov detekcie interakcie voči klasickému zhlukovaniu s limitom počtu stavov je možné vidieť v grafe na obrázku 5.3. Vidíme, že tento prístup zachoval oproti klasickému zhlukovaniu výsledný počet stavov a počet zhlukov. To znamená, že použitím tohoto prístupu nedošlo k rozdielnemu výsledku a tým pádom môžeme porovnávať výsledné zrýchlenie.



Obrázek 5.3: Porovnanie detekcie interakcie oproti klasickému zhlukovaniu

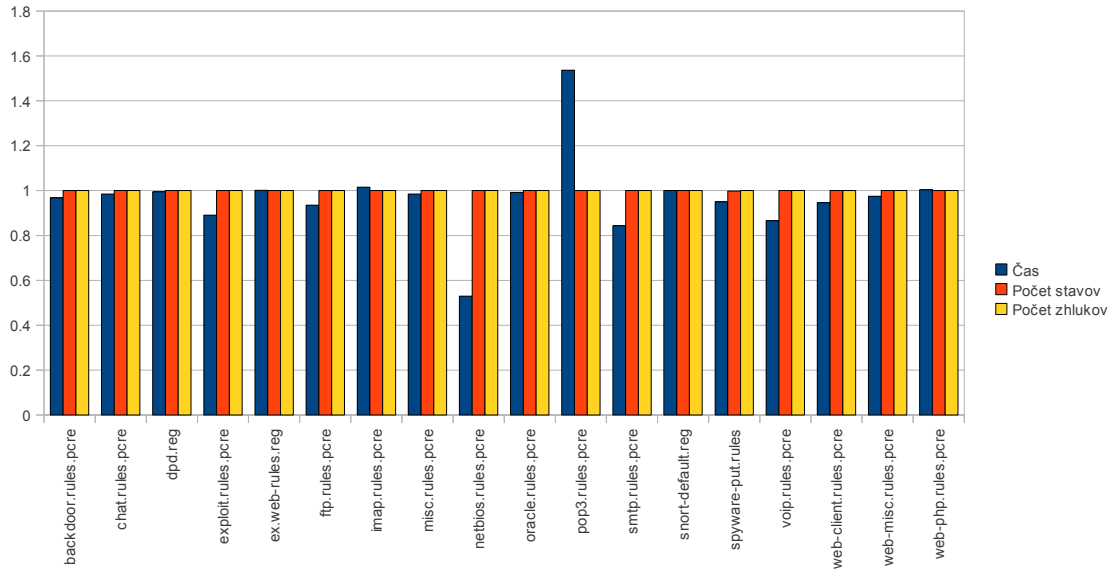
Tento prístup bol rýchlejší alebo porovnateľný na všetkých množinách regulárnych výrazov s výnimkou jednej sady pravidiel. Konkrétne sa jednalo o sadu „misc.rules.pcre“. Dôvodom je fakt, že táto sada výrazov obsahuje málo interagujúcich výrazov, a do vý-

sledku sa premietla réžia samotného detekčného algoritmu. Nie je to však možné považovať za nevýhodu tohto prístupu, lebo ak takéto spomalenie nastane, tak len pri množinách výrazov, ktoré spolu interagujú málo. Keďže interagujú málo, aj ich výpočet interakcií by mal byť rýchlejší, a teda nevýhoda tohto prístupu sa prejaví len pri množinách, ktorých výpočet interakcií aj tak nebude trvať príliš dlho. Horšie by bolo, ak by sa spomalenie prejavovalo pri sade pravidiel s opačnou charakteristikou, a teda spomaľovalo by výpočet množín, ktoré sa už z princípu počítajú dlhšie. Najviac sa zrýchlenie prejavilo na množine pravidiel „netbios.rules.pcre“.

5.3.5 Detekcia neinterakcie

Porovnanie výsledkov detekcie neinterakcie voči klasickému zhlukovaniu s limitom počtu stavov je možné vidieť v grafe na obrázku 5.4. Vidíme, že aj tento prístup si zachoval oproti klasickému zhlukovaniu výsledný počet stavov a počet zhlukov. To znamená, že použitím tohoto prístupu nedošlo k rozdielnemu výsledku a tým pádom môžeme porovnávať výsledné zrýchlenie.

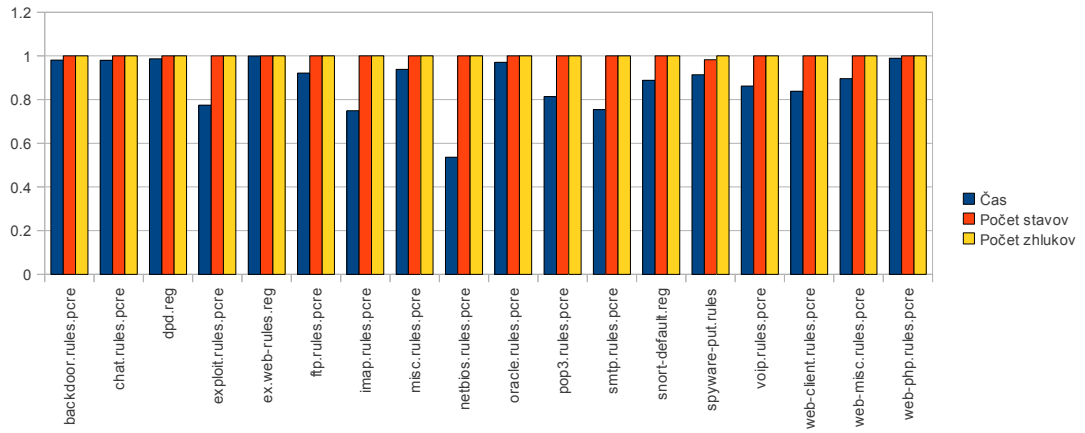
Z hľadiska samotnej rýchlosti sa tento prístup prejavil pozitívne na všetkých sadách pravidiel, okrem sady „pop3.rules.pcre“. V tomto prípade sa jedná o množinu pravidiel, ktoré medzi sebou interagujú málo. Opakuje sa teda situácia z predchádzajúceho prístupu, kedy sa do spomalenia premietla réžia pridanej časti algoritmu. Keďže sa opäť jedná o pravidlá, ktoré spolu málo interagujú, a ich výpočet bude rýchlejší oproti sadám, ktoré interagujú viac, nie je toto spomalenie až tak dôležité. Dôležité je, že pri použití na inej sade pravidiel dokážeme výsledok získať rýchlejšie alebo za rovnaký čas oproti klasickému zhlukovaciemu algoritmu s limitom počtu stavov.



Obrázek 5.4: Porovnanie detekcie neinterakcie oproti klasickému zhlukovaniu

5.3.6 Spojená detekcia interakcie aj neinterakcie

Keďže predchádzajúce dva prístupy spolu silne súvisia (oba zastavia výpočet determinizácie vo chvíli, keď už je známy výsledok), ale ich spoločné použitie sa nevylučuje, rozhodli sme sa spojiť ich do spoločného prístupu a skúsiť získať ešte lepšie výsledky. Porovnanie výsledkov spojenia detekcie neinterakcie a detekcie neinterakcie voči klasickému zhlukovaniu s limitom počtu stavov je možné vidieť v grafe na obrázku 5.5. Všimnime si, že aj tento prístup si zachoval oproti klasickému zhlukovaniu výsledný počet stavov a počet zhlukov. To znamená, že použitím tohoto prístupu nedošlo k rozdielnemu výsledku, a preto môžeme porovnávať výsledné zrýchlenie.

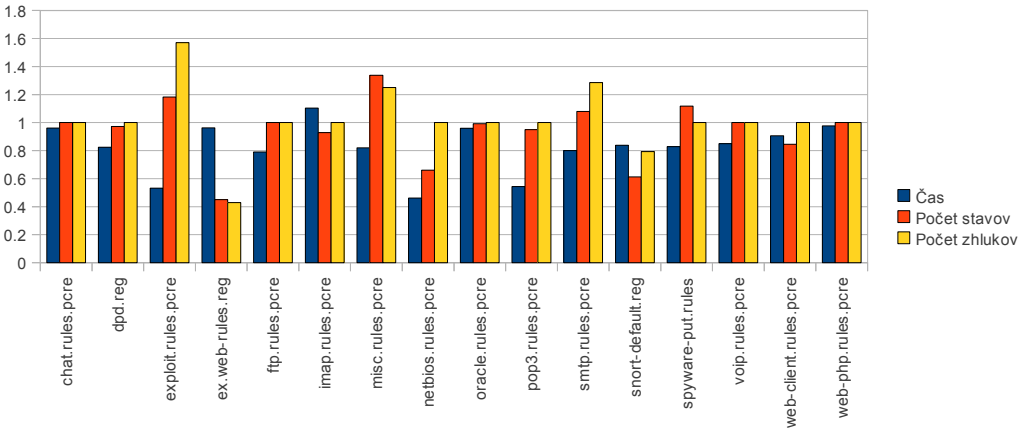


Obrázek 5.5: Porovnanie spojenia detekcie interakcie a neinterakcie oproti klasickému zhlukovaniu

Pri porovnaní oboch prístupov so spojeným prístupom je jasne vidieť, že zrýchlenie sa pozitívne prejavilo na všetkých testovaných množinách pravidiel. Oba prístupy sa dopĺňajú tam, kde druhý prístup zaostáva. Takto napríklad zmizli prípady pomalšie oproti klasickému zhlukovaniu, konkrétne u množín pravidiel „misc.rules.pcre“ a „pop3.rules.pcre“. Preto tento prístup môžeme jasne odporučiť, keďže zachováva počet stavov a počet zhlukov a zároveň poskytuje výsledok celého výpočtu v kratšom čase a na všetkých testovaných množinách pravidiel.

5.3.7 Cyklus v automate

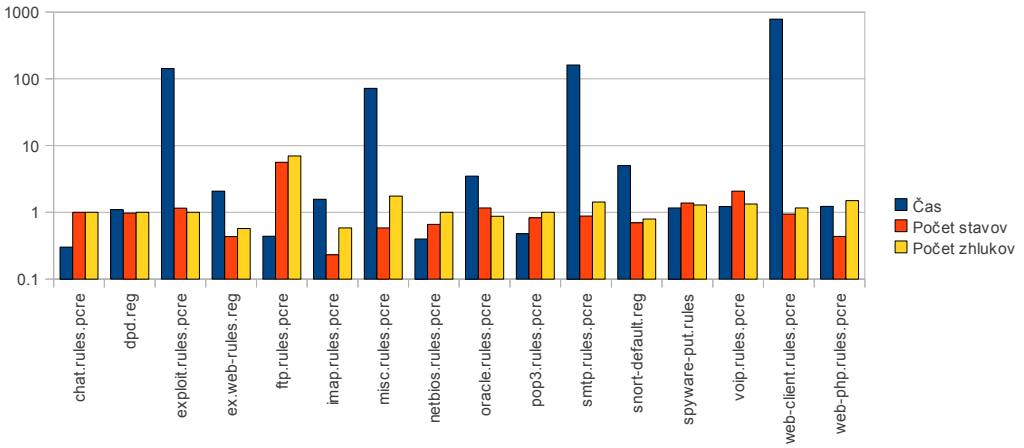
Metóda detekujúca cykly v automatoch bola rýchlejšia ako klasické zhlukovanie vo všetkých prípadoch okrem sady pravidiel `imap.rules.pcre` (viz obrázok 5.6). Aj v tomto prípade to však bolo iba o približne 10%. V troch prípadoch (`exploit.rules.pcre`, `netbios.rules.pcre` a `pop3.rules.pcre`) bol tento prístup rýchlejší približne o 50%. Tento algoritmus sa výrazne odlišoval od klasického zhlukovacieho algoritmu vo výslednom počte zhlukov a stavov. V štyroch pravidlách narástol počet stavov aj zhlukov, v ôsmich si udržal približne rovnaké hodnoty a u štyroch boli hodnoty zase lepšie. V prípade `ex.web-rules.reg` klesol počet stavov aj zhlukov o viac ako 50% pri zachovaní rovnakého času výpočtu.



Obrázek 5.6: Porovnanie zhlukovania s ohľadom na cykly oproti klasickému zhlukovaniu

5.3.8 Neinterakcia viacerých RV naraz

Výsledky tohoto prístupu (viz obrázok 5.7) nie sú veľmi presvedčivé. U troch sád pravidiel (`exploit.rules.pcre`, `misc.rules.pcre` a `smtp.rules.pcre`) trval výpočet až 100-násobne dlhšie oproti klasickému zhlukovaniu a u `web-client.rules.pcre` dokonca 1000-násobne dlhšie. V niektorých prípadoch trval výpočet síce o niečo dlhšie, ale dospel k podstatne lepšiemu riešeniu.



Obrázek 5.7: Porovnanie zhlukovania počítajúceho neinterakciu viacerých RV naraz oproti klasickému zhlukovaniu. Osa y je v logaritmickom merítku.

5.3.9 Zhodnotenie

Z testovaných prístupov by sme asi najviac vyzdvihli detekciu interakcie a detekciu neinterakcie, prípadne ich spojenie, pretože dokázali za kratší čas dospieť k lepšiemu riešeniu ako klasický zhlukovací algoritmus.

Algoritmus detekujúci cykly v automatoch dosiahol tiež zaujímavé výsledky, hlavne čo sa týka redukcie počtu stavov a zhlukov. Za klasickým zhlukovacím algoritmom značne nezaostával ani časovou náročnosťou.

Zaujímavé výsledky neinterakcie viacerých RV znehodnocuje nepredvídateľná časová náročnosť v niektorých prípadoch, Tento algoritmus by sme určite neodporučili k reálnemu použitiu.

Limit počtu stavov pri počítaní interakcie sa ukázal ako správny prístup. Ak je možné tento prístup použiť, malo by tak byť urobené vždy, pretože čas výpočtu algoritmu za každej situácie zníži. Limit počtu stavov bohužiaľ nepodporuje knižnica na prácu s konečnými automatmi FSM Library [18] od AT&T.

Všetky spomenuté prístupy sú implementované v rámci netbench formou triedy, ktorá dedí z bázovej triedy pre DKA (`b_dfa`). Trieda `interaction_dfa` rozširuje determinizáciu o nepovinné parametre, ktorými je možné uplatniť a nastaviť spomínané prístupy.

Kapitola 6

Zhlukovanie v reálnom čase

Predchádzajúci text sa zaoberal výhradne zhlukovacím algoritmom 4.3. Avšak čas potrebný pre výpočet interakcií a výpočet zhlukov v tomto algoritme môže byť nezanedbateľne dlhý. Prejavilo sa to v prípade obsiahlych množín pravidiel (niekoľko stoviek), pre ktoré trval výpočet rádovo niekoľko dní. Časová náročnosť sa javí ako výrazná nevýhoda klasického zhlukovacieho algoritmu, pretože ak chceme použiť v IDS nejakú množinu pravidiel, tak IDS môže pomocou nich začať detekovať až po vypočítaní všetkých interakcií a zhlukov.

Z uvedených príčin vyplynula nutnosť návrhu algoritmu, ktorý by bol schopný pracovať v reálnom čase. Tým máme na mysli algoritmus, ktorý by mal počas celého výpočtu k dispozícii riešenie a to už od jeho začiatku. Algoritmus by mal byť takisto robustný voči jednoduchým zmenám množiny pravidiel, ako pridanie alebo odobratie malej množiny pravidiel. Po uvážení týchto požiadaviek sme zvolil použitie genetického algoritmu.

Dôvody pre použitie genetického algoritmu sú zrejmé, genetický algoritmus spĺňa uvedené požiadavky:

- pracuje v reálnom čase,
- riešenie je dostupné takmer od začiatku algoritmu,
- poskytuje viacero riešení nielen po skončení, ale už počas výpočtu,
- pri správne zvolenej reprezentácii zhlukov je odolný voči malým zmenám vstupnej množiny výrazov.

6.1 Evolučné algoritmy

Evolučné algoritmy možno zaradiť medzi základné prostriedky modernej matematiky a informatiky slúžiace k riešeniu zložitých problémov. Najväčšie využitie majú vtedy, ak hľadáme také globálne minimum, ktoré je obklopené množstvom lokálnych miním. Používajú sa pri hľadaní riešení v extrémne ťažkých situáciách, kedy použitie deterministických metód prehľadávania nie je možné aplikovať.

Genetický algoritmus (GA) je metóda prehľadávania priestoru riešení nejakého problému. Patrí medzi základné stochastické problémy s výraznými evolučnými črtami. V súčasnosti je najčastejšie používaným evolučným optimalizačným algoritmom, so širokou paletou optimalizácie vysoko multimodálnych funkcií cez kombinatorické a grafovo-teoretické problémy až po aplikácie nazývané „umelý život“.

Ukazuje sa, že na základe analógie evolučnými procesmi prebiehajúcimi v biologických systémoch existuje alternatívna možnosť, ako usmerniť náhodné generovanie bodov k hodnotám blízkym optimálnym. Práve táto analógia sa stala základom genetického algoritmu, ktorý vylepšuje čisto stochastický slepý algoritmus prehľadávania stavového priestoru tak, že poskytuje v reálnom čase optimálne riešenie. [6]

6.1.1 Základné pojmy

Genetické algoritmy spadajú do väčšej triedy tzv. evolučných algoritmov, pomocou ktorých je možné hľadať riešenia optimalizačných problémov inšpirovaných biologickou evolúciou ako sú dedičnosť, selekcia, kríženie a mutácia. Je úžasné, že také jednoduché procesy odpozorované z reálneho sveta je možné použiť na riešenie zložitých matematických problémov. Predtým než si ukážeme, ako funguje samotný genetický algoritmus, bude potrebné opísať niekoľko dielčich častí, na ktorých je postavený. Jedná sa o pojmy: chromozóm, populácia, fitness funkcia, kríženie, mutácia a selekcia. [10]

Chromozóm Pod pojmom chromozóm sa najčastejšie myslí reťazec znakov, ktorý je reprezentáciou riešenia. Tak ako v normálnom svete chromozóm predurčuje vlastnosti živého organizmu, tak pri hľadaní riešenia chromozóm predstavuje informáciu, z ktorej dokážeme zistiť, kde sa bude riešenie nachádzať. S týmto pojmom úzko súvisia aj pojmy genotyp a fenotyp. Pod genotypom rozumieme informáciu uchovávanú v chromozóme a fenotyp určuje, ako sa chromozóm prejaví navonok v priestore hľadaných riešení. Rôzne genotypy môžu mať vo výsledku rovnaký fenotyp, naopak to však neplatí. Objekt reprezentovaný chromozómom nazývame jedincom. Množina jedincov sa nazýva populácia.

Fitness Ohodnocovacia funkcia chromozómu. Výsledok tejto funkcie predstavuje fenotyp pre daný chromozóm. Funkcia je potrebná pre odlíšenie kvality jedincov, aby bolo možné vybrať najlepšie riešenie, prípadne rozhodnúť o možnej reprodukcii jedinca v ďalšom kroku. Hodnota fitness musí jednoznačne určiť usporiadanie nad množinou jedincov. Výsledok býva väčšinou reprezentovaný číslom s pohyblivou rádovou čiarkou. Fitness určuje vyššie, resp. nižšie ohodnotenie kvalitnejším riešeniam, záleží na tom, či hľadáme v priestore maximum, resp. minimum.

Kríženie Pod pojmom kríženie si možno predstaviť nejakú funkciu, ktorej vstupom sú dva chromozómy (nazývané rodičia) a výstupom sú spravidla dva chromozómy (potomkovia). Účelom je, tak ako pri spárení dvoch organizmov v prírode, prenos kladných vlastností oboch rodičov na potomkov. Samotné kríženie je založené na náhodnosti, kedy volíme bod kríženia chromozómov. Tento náhodne zvolený bod slúži na rozdelenie chromozómov na dve časti, ktoré následne využijeme pri tvorbe nového jedinca a to takým spôsobom, že potomok zdedí časť pred bodom kríženia od jedného rodiča a ostatnú časť od druhého rodiča. Existuje viacero možností, ako aplikovať operáciu kríženia, napr. pomocou nie jedného, ale viacerých bodov kríženia.

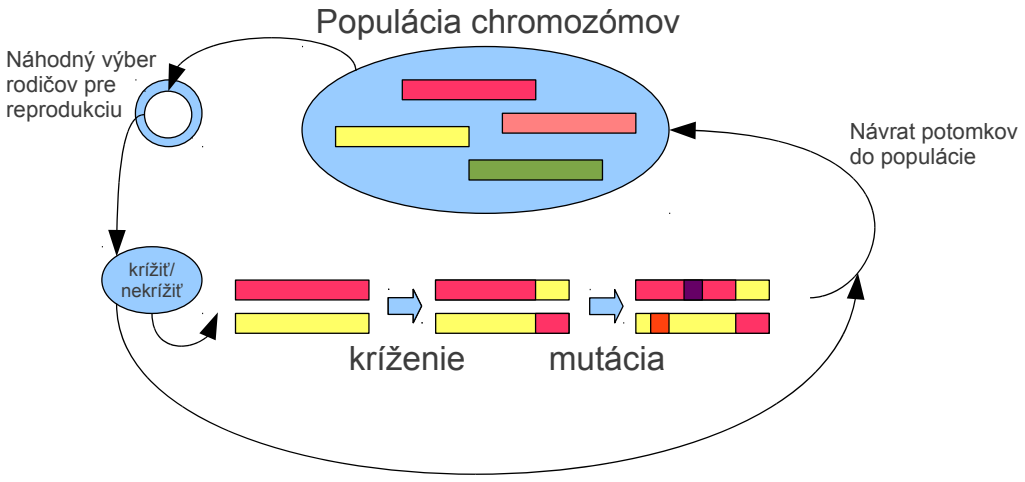
Mutácia Tak ako u živých organizmov môže nastať náhodná zmena ich genetickej informácie, takisto môže nastať rovnaká náhodná situácia aj pri chromozóme jedinca v genetickom algoritme. Vďaka mutácii nie je genetický algoritmus odkázaný iba na kríženie, lebo ak by bolo použité iba kríženie, všetky jedince mohli konvergovať k niekoľko málo riešeniam. Aj veľmi jednoduchá mutácia môže mať zásadný vplyv na fitness zmutovaného chromozómu

oproti originálnemu, preto je dôležité s ňou narábať veľmi opatrne. Mutácia by mala byť len malým usmernením cesty k riešeniu, na jednej strane takým mocným, aby sme neskončili v lokálnom minime a na druhej strane takým slabým, aby sme sa od hľadaného minima príliš nevzdialili.

Selekcia Pri genetickom algoritme je potrebné z každej generácie vybrať jedincov, ktorých genóm sa využije v ďalšej populácii ich potomkov pomocou kríženia a mutácie. Bolo by nerozumné vyberať jedincov, ktorí nie sú životaschopní a nemajú veľkú šancu na prežitie. Preto do genetických algoritmov vnášame možnosť výberu jedincov tak, aby bola nasledujúca populácia čo najviac životaschopná. Genetické algoritmy využívajú rôzne možnosti prirodzeného výberu inšpirované prírodou. Takýmto výberom môže byť napríklad výber iba tých najlepších jedincov alebo môžeme vyberať aj na základe akejsi súťaže (tournament), pri ktorej sa do výberu nemusia dostať vždy tí najlepší, ale jedinci, ktorí boli schopní obstáť v súboji s inými jedincami. Paralela s prírodou je v tomto prípade zase zrejmá.

6.1.2 Genetický algoritmus

Samotný genetický algoritmus je postavený na uvedených pojmoch. Základný algoritmus na začiatku vygeneruje náhodnú populáciu chromozómov. Ďalej sa spočíta ohodnotenie každého chromozómu fitness funkciou. Následne sa pomocou nejakej selekčnej funkcie vyberú páry, ktoré sa budú navzájom krížiť. Kríženie sa opakuje náhodne-krát, v závislosti na parametre, ktorý udáva dôležitosť kríženia. Čím väčšia je hodnota parametru, tým väčšia je pravdepodobnosť, že kríženie sa udeje viacnásobne. Potom sa zase náhodne na základe parametru použije na niektoré chromozómy mutácia. Príkladom jednoduchého genetického algoritmu môže byť napr. algoritmus reprezentovaný obrázkom 6.1.



Obrázek 6.1: Príklad jednoduchého genetického algoritmu [6]

Pri príprave konvenčného genetického algoritmu reprezentujúceho chromozómy ako reťazce fixnej dĺžky, proces prípravy zahrňuje: [4]

1. stanovenie reprezentačnej schémy chromozómu,

2. stanovenie fitness funkcie,
3. stanovenie parametrov a premenných algoritmu,
4. stanovenie cesty ako algoritmus dospeje k výsledku a
5. stanovenie kritérií pre ukončenie algoritmu.

Podľa J.R. Kozu by mal algoritmus založený na chromozómoch reťazcov fixnej dĺžky pracovať podľa nasledovného postupu: [4]

1. Náhodne vytvor počiatočnú množinu jedincov reťazcov fixnej dĺžky.
2. Iteratívne vykonávaj nasledujúce podkroky na populácii reťazcov, až pokým nie je splnené kritérium ukončenia:
   (a) Ohodnoť fitness každého jednotlivca v populácii.
   (b) Vytvor novú populáciu reťazcov použitím minimálne dvoch z troch nasledovných operácií. Tieto operácie sú použité na reťazce jednotlivcov v populácii vybraných s pravdepodobnosťou založenou na fitness.
      i. Skopíruj existujúceho jednotlivca do novej populácie.
      ii. Vytvor dva nové reťazce genetickou rekombináciou náhodne zvolených podreťazcov z dvoch existujúcich reťazcov.
      iii. Vytvor nový reťazec z existujúceho reťazca, a zmeň ho tak, že náhodne zmutuješ jeden znak na náhodnej pozícii
3. Najlepší reťazec jednotlivca, ktorý sa objavil v ktorejkoľvek generácii, je vybraný za výsledok tohto behu genetického algoritmu. Tento výsledok môže reprezentovať riešenie (alebo približné riešenie) daného problému.

Na obrázku 6.2 je zobrazený vývojový diagram týchto krokov pre konvenčný genetický algoritmus založený na reťazcoch. Index `i` značí jednotlivca v populácii o veľkosti `M`. Premenná `GEN` je aktuálne poradové číslo generácie.

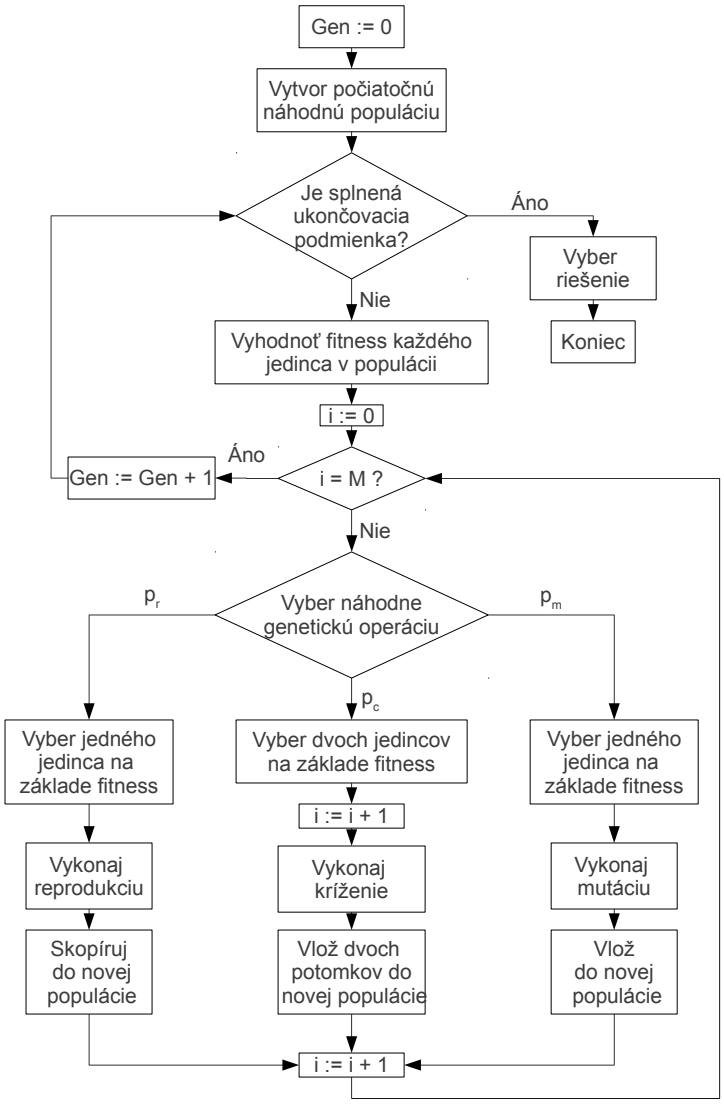
Existuje nespočetne mnoho iných možností založených na základnom genetickom algoritme; tento vývojový diagram je len jedna z nich. Napr. s mutáciou je často nakladané tak, že sa môže použiť až po reprodukcii alebo krížení (tak ako je vidieť na príklade z obrázku 6.1), takže jedinec môže byť zmutovaný a zkrížený v jednej generácii. Taktiež počet koľkokrát sú jednotlivé genetické operácie vykonané v jednej generácii, je často nastavený ako explicitná hodnota pre všetky generácie, oproti náhodnému výberu, tak ako to je vo vývojovom diagrame na obrázku 6.2. [4]

Na vývojovom diagrame taktiež nie je popísané ako krížiť dvoch jedincov. Namiesto toho je len uvedené, že sa majú náhodne vybrať dva jedince, ktoré sa budú na krížení podielať. Túto operáciu je možné nahradiť ľubovoľným zmysluplným postupom.

Genetický algoritmus hľadá reťazec znakov, ktorý by mohol mať vysokú fitness. Fitness vedie populáciu smerom k riešeniu len na základe výsledku vo forme čísla. Operácie použité pre kríženie sú taktiež veľmi jednoduché (kopírovanie, rozdeľovanie, spájanie, malé náhodné zmeny chromozómu). V praxi sú genetické algoritmy veľmi rýchle pri hľadaní v komplexných, vysoko nelineárnych, multidimenzionálnych priestoroch. Toto je o to prekvapujúcejšie, lebo samotný genetický algoritmus nevie nič o probléme, ktorý rieši, ani o použitej fitness.

Ako už bolo spomenuté, pri budovaní genetického algoritmu je najdôležitejšie zvoliť správnu štruktúru, ktorá bude reprezentovať riešenia v prehľadávanom priestore a fitness funkciu, ktorá ju správne ohodnotí. Ďalej je dôležité zvoliť veľkosť populácie, počet generácií a parametre kontrolujúce pravdepodobnosti vykonávania genetických operácií, ukončovaciu podmienku a metódu, ktorá vyberie po skončení algoritmu najlepšie riešenie. Všetky spomenuté časti významne ovplyvňujú správanie populácie počas genetického algoritmu a poskytnuté riešenie.

Po spustení genetického algoritmu je veľmi dôležité nesledovať iba výsledky, ale na nižšej úrovni sledovať prechod medzi jednotlivými generáciami a snažiť sa hľadať v použití genetických operácií a rozdielov medzi fitness jedincov vzory, ktoré by mohli pomôcť pri zmene už spomínaných parametrov tak, aby sa genetický algoritmus uberal správnym smerom. [7]



Obrázek 6.2: Vývojový diagram konvenčného genetického algoritmu [4]

6.2 Návrh GA pre zhlukovanie

V nasledujúcej podkapitole sa pokúsime definovať, ako by mal vyzerať genetický algoritmus, ktorého úlohou bude hľadanie riešenia pre zhlukovanie množiny regulárnych výrazov. Bude rozobrané, akým spôsobom sme sa rozhodli reprezentovať zhluky v chromozóme. Taktiež budú vysvetlené genetické operácie pracujúce nad množinou zhlukov, teda ich kríženie a mutácie a nakoniec si vysvetlíme parametre genetického algoritmu, ku ktorým sme dospeli.

Všetky riešenia spomenuté nižšie, nielen parametre, sú výsledkom experimentovania s rôznymi typmi operácií, zavádzaním náhodnosti do rôznych podčastí operácií. Prezentované riešenia sú teda riešenia, ku ktorým sme dospeli až po dôslednom sledovaní správania sa systému a sú to najlepšie riešenia, na ktoré sme pri skúmaní narazili.

6.2.1 Problém a jeho riešenie

Ešte pred samotným definovaním častí genetického algoritmu je nutné definovať problém, ktorý sa snažíme vyriešiť a riešenie, ku ktorému by mal genetický algoritmus dospieť. V prípade zhlukovania sa snažíme nájsť takú množinu zhlukov regulárnych výrazov, ktorá by mala byť najmenšou možnou množinou a zároveň konečný automat každého zhluku by nemal presiahnuť dopredu daný limit počtu stavov, ktorý je zavedený už v klasickom zhlukovacom algoritme. Slepé prehľadávanie stavového priestoru je NP-ťažký problém. Na druhej strane máme klasický zhlukovací algoritmus, ktorý dokáže nájsť relatívne dobré riešenie. Toto riešenie však nie je najlepším riešením, o čom sa presvedčíme pri experimentovaní s genetickým algoritmom. Cieľom genetického algoritmu je teda nájsť minimálne také dobré riešenie ako klasický zhlukovací algoritmus, avšak v kratšom čase.

Vstup a výstup genetického algoritmu je rovnaký ako pri klasickom zhlukovacom algoritme. Vstupom teda bude množina regulárnych výrazov a výstupom množina množín (zhlukov) regulárnych výrazov. Oproti klasickému zhlukovaciemu algoritmu však budeme mať v celom priebehu k dispozícii najlepšie dosiahnuté riešenie a môžeme ho použiť oveľa skorej, ako by sa dopočítal klasický zhlukovací algoritmus. Genetický algoritmus prináša ešte jednu zásadnú výhodu; vďaka tomu, že genetický algoritmus sa neukončí pri nájdení lepšieho riešenia (oproti klasickému algoritmu), je možné pokračovaním algoritmu dostať ešte ďalšie nové a lepšie riešenia. Teoreticky by genetický algoritmus mohol bežať bez prerušenia a stále poskytovať najlepšie dosiaľ nájdené riešenie.

6.2.2 Reprezentácia zhlukov

Spôsob reprezentácie zhlukov v genetickom algoritme je jeho základom. V tomto prípade sa snažíme nájsť štruktúru, ktorá bude obsahovať množinu zhlukov a každý zhluk bude obsahovať množinu regulárnych výrazov. Pri implementácii sme sa rozhodli, že množina zhlukov bude reprezentovaná ako reťazec zhlukov. Samotný zhluk bude reprezentovaný ako n-tica abecedne usporiadaných regulárnych výrazov. Vďaka tomuto usporiadaniu je možné porovnať dva zhluky medzi sebou a zistiť, či sa od seba odlišujú, čo sme využili pri fitness funkcii.

Dôležitým znakom v reprezentácii chromozómu je fakt, že každý regulárny výraz sa môže vyskytovať v práve jednom zhluku a zároveň nemôže byť viac zhlukov ako je regulárnych výrazov, teda máme presný limit dĺžky chromozómu. To znamená, že riešenie hľadáme v konečnom priestore. Zhluky v chromozóme, ktoré neobsahujú žiadne regulárne výrazy za zhluky nepovažujeme. Takéto miesta môžu vzniknúť po genetickej operácii, po ktorej sa miesto vyprázdni tým, že sa výrazy z neho prenesú do iného zhluku.

Príklad jedného chromozómu je možné vidieť na obrázku 6.3. V tomto príklade je vidieť, že vstupná množina regulárnych výrazov pozostáva z 12-tich výrazov (0 až 11), obsahuje zhluky s viacerými výrazmi, zhluky s jedným prvkom a taktiež prázdne zhluky, ktoré za zhluky nepovažujeme.

(0, 5, 11)	(1, 6, 8, 9)	(2)	(3, 4, 7)	()	(10)

Obrázek 6.3: Príklad chromozómu reprezentujúceho zhluky regulárnych výrazov

6.2.3 Fitness funkcia

Na ohodnotenie chromozómov sme použili fitness funkciu, ktorej vstupom je chromozóm a výstupom číslo. Pri hodnotení riešení vyplynulo, že sa budeme snažiť hľadať riešenie s minimálnou hodnotou fitness, teda pre lepšie riešenie vráti fitness nižšie číslo. Hodnota fitness je založená na počte znakov DKA jednotlivých zhlukov a počte zhlukov. Je navrhnutá tak, aby riešenia s menším počtom znakov a menším počtom zhlukov mali nižšiu fitness, keďže sa snažíme nájsť práve také riešenie.

Experimentovali sme s rôznymi možnosťami ako spočítať fitness, nakoniec ako najlepšie riešenie vyplynulo vypočítať ho pomocou vzťahu $pocet\_stavov * (pocet\_zhlukov^2)$. Tento vzťah rešpektuje počet stavov, ale dáva väčší dôraz na počet zhlukov. Pri experimentoch sa ukázalo, že je efektívnejšie najprv hľadať riešenia s malým počtom zhlukov a až potom hľadať riešenie s rovnakým počtom zhlukov, ale nižším počtom stavov, čomu zodpovedá aj spomenutý vzťah.

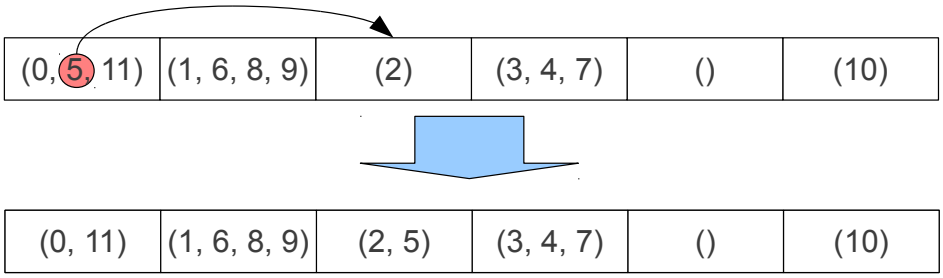
Ďalšou funkcionalitou je penalizovanie jedincov, pri ktorých minimálne jeden zhluk prekračuje limit počtu stavov takou vysokou hodnotou fitness, že daný jedinec sa nebude môcť reprodukovať v ďalšej generácii.

V rámci fitness sme zaviedli vylepšenie, ktoré si pamätalo ohodnotenie všetkých predchádzajúcich zhlukov v celom algoritme, čo bolo možné vďaka reprezentácii zhlukov a dali sa medzi sebou porovnávať na ekvivalenciu. Bez tohto vylepšenia by nebolo reálne, aby genetický algoritmus pracoval dostatočne rýchlo. Napríklad pri jednej mutácia sa nemusia počítať všetky zhluky nového chromozómu, ale len zhluky, ktoré sa mutáciou zmenili, teda iba 2 zhluky, keďže ohodnotenie ostatných zhlukov si fitness uložila už v predchádzajúcich krokoch.

6.2.4 Mutácia

Táto operácia je relatívne jednoduchá. Napriek jej jednoduchosti je ale mimoriadne dôležitá, pretože poskytuje únik z lokálneho minima, kde by samotné kríženie nemuselo tento problém vyriešiť. Mutácia spočíva v náhodnom výbere regulárneho výrazu z náhodného zhluku a následne vložením tohto výrazu do iného náhodného zhluku (viz obrázok 6.4).

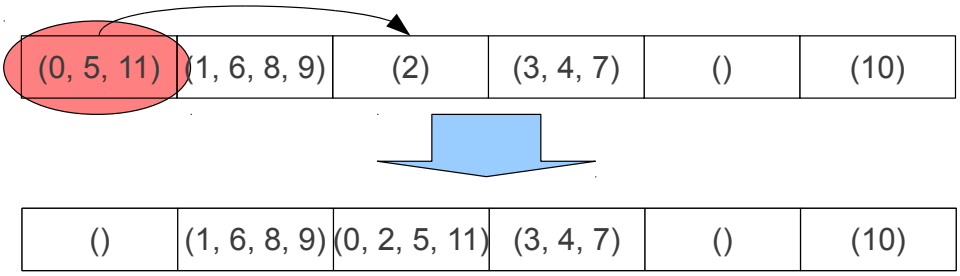
Uvedená forma mutácie je založená na fakte, že v n-dimenzionálnom priestore (n je počet regulárnych výrazov) riešení existuje lepšie riešenie minimálne jedným smerom jednej dimenzii. V prípade zhlukov to znamená, že ak odoberieme „správny“ výraz zo zhluku (jeden výraz znamená jednu dimenziu) a pridáme ho do iného „správneho“ zhluku (posun



Obrázek 6.4: Príklad mutácie chromozómu

v dimenzii), mali by sme dospieť k lepšiemu riešeniu. Ak nepracujeme už s najlepším riešením, musí existovať výraz, ktorý ak bude odobraný zo zhluku (tým zhluku klesne počet stavov), tak je možné ho pridať do druhého zhluku, kde narastie počet stavov menej ako klesol pri odobratí z prvého zhluku. Táto hypotéza by mala platiť vždy, pretože problémom sú interagujúce dvojice v zhlukoch, ktoré sa snažíme od seba oddeľovať.

Druhým spôsob mutácie, ktorý sme použili bolo spojenie dvoch zhlukov do jedného, teda presun všetkých regulárnych výrazov z jedného zhluku a ich pripojenie k druhému zhluku. Tento proces by samozrejme dokázala aj predchádzajúca forma mutácie, ale nie počas vývinu jednej generácie. Je to teda skratka, pomocou ktorej je populácia schopná dokonvergovať k riešeniu s nízkym počtom zhlukov. Túto operáciu demonštruje obrázok 6.5

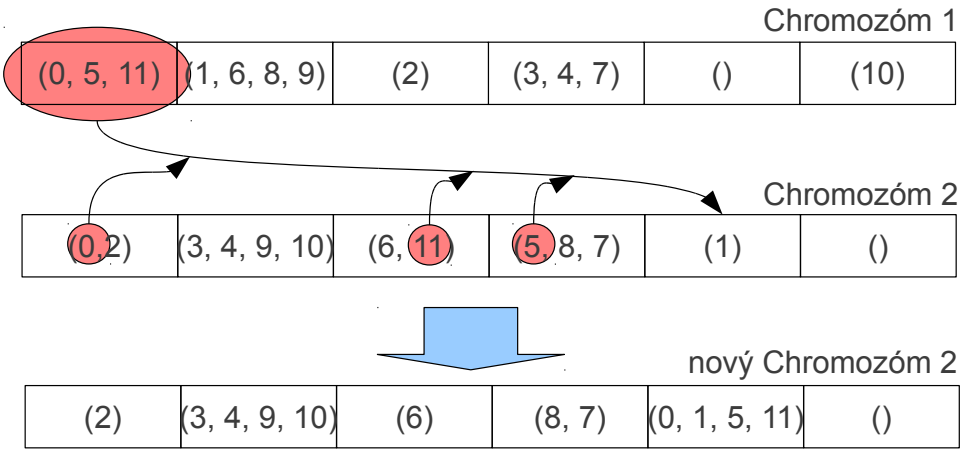


Obrázek 6.5: Príklad mutácie chromozómu - spájanie

6.2.5 Kríženie

Myšlienkou kríženia v rámci zhlukov by mal byť prenos informácie o regulárnych výrazoch, ktoré sú spolu v zhlukoch. Informácia o tom, či sú výrazy spolu v zhluku, je z hľadiska riešenia informáciou pozitívnou aj pre ostatné zhluky a teda sa prenesie pri krížení z jedného zhluku do druhého. Kríženie v rámci chromozómu je vyriešené tak, že sa vyberie náhodný zhluk z prvého chromozómu. Tento zhluk sa použije na zmenu druhého chro-

mozómu. V druhom chromozóme najprv zo všetkých zhlukov odstránime obsah spoločný s uvedeným náhodne zvoleným zhlukom a následne tento zhluk pridáme do niektorého zo zhlukov. Pri tejto operácii je možné zvoliť aj variantu, kedy nepoužijeme celý zhluk, ale len nejakú náhodnú vzorku. Proces pre takéto kríženie zobrazuje obrázok 6.6. Druhá polovica procesu prebieha rovnako, s tým rozdielom, že zameníme cieľový a zdrojový chromozóm.



Obrázek 6.6: Príklad kríženia chromozómu

6.2.6 Generovanie počiatočnej populácie

Na generovanie počiatočnej populácie sme sa rozhodli negenerovať populáciu náhodne. Problém bol v tom, že pri náhodnom generovaní sa takmer nikdy nestalo, že by žiadny zo zhlukov nepresiahol limit počtu stavov. Tým pádom by algoritmus začínal so samými nekorektnými riešeniami. Snažili sme sa taktiež tvoriť počiatočné množiny zhlukov nie náhodne, ale pomocou algoritmov, ktoré by pripravili zhluky, v ktorých by nebol limit prekročený, avšak takéto postupy boli skoro také zložité ako klasický zhlukovací algoritmus. Preto sme nakoniec nezvolili ani jedno zo spomenutých spôsobov generovania, ale vytvorili sme n zhlukov a každý obsahoval práve jeden regulárny výraz zo vstupnej množiny regulárnych výrazov. Na začiatku boli teda všetky jedince rovnaké, avšak mutácie v genetickom algoritme zaistili, že sa po pár generáciách začali navzájom líšiť a nadobúdať rôzne hodnoty fitness.

6.2.7 Parametre genetického algoritmu

K parametrom genetického algoritmu sme dospeli na základe experimentovania s rôznymi množinami výrazov a sledovaním správania genetického algoritmu na nich. Dospeli sme k hodnote 0,3 pre pravdepodobnosť mutácie a 0,2 pre pravdepodobnosť kríženia. Veľkosť populácie by mala byť v rozsahu 10 až 20 jedincov.

6.3 Implementácia

Samotný genetický algoritmus sme neimplementovali, ale použili knižnicu deap [19], ktorá je uspôsobená na prácu s evolučnými algoritmami. Túto knižnicu sme zvolili aj preto, že jej implementačný jazyk je Python, takže ju bolo možné veľmi ľahko použiť spoločne s Netbenchom [20].

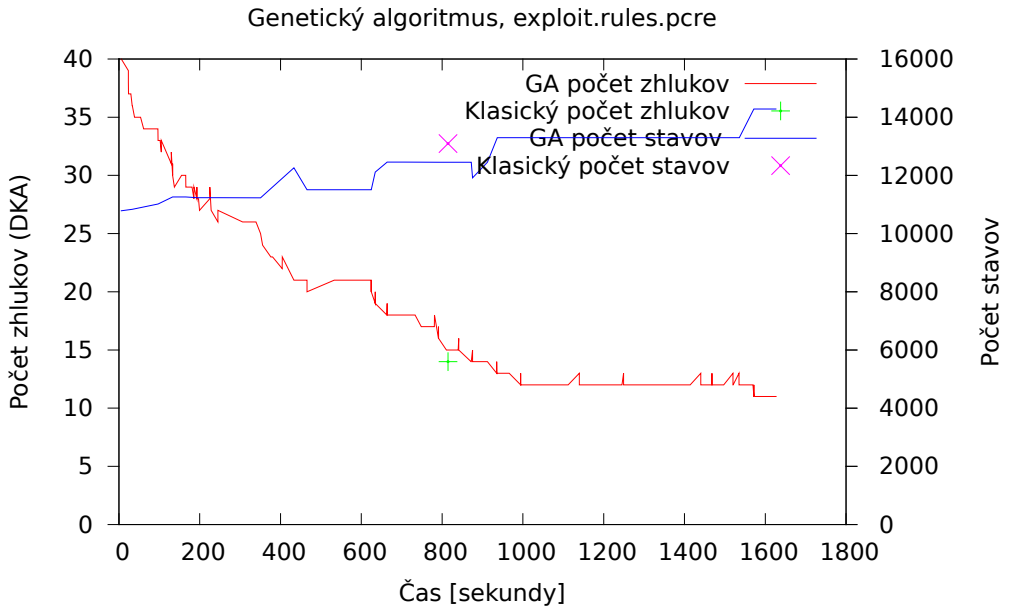
Z knižnice deap sme použili len základ genetického algoritmu a selekčné funkcie. Kríženie, mutáciu a fitness sme naprogramovali v rámci Netbench ako metódy do triedy určenej pre zhlukovanie regulárnych výrazov.

6.4 Výsledky experimentov

Pri experimentoch s genetickým algoritmom sme postupovali podobne, ako je popísané v kapitole 5.3.2. Mali sme k dispozícii tie isté súbory s množinami regulárnych výrazov a na testovanie boli použité tie isté počítače. Pri všetkých testoch bol zase použitý limitný počet 2000 stavov na jeden zhluk. Ako referenčné riešenie pre každú množinu výrazov sme zvolili riešenia získané klasickým zhlukovacím algoritmom, s ktorým sme porovnali výsledky genetického algoritmu.

Nebudeme uvádzať všetky výsledky testov, iba ich zhrnieme a rozoberieme výsledky niektorých význačných. Výsledky získané zo všetkých súborov s množinami regulárnych výrazov je možné nájsť v Prílohe B.

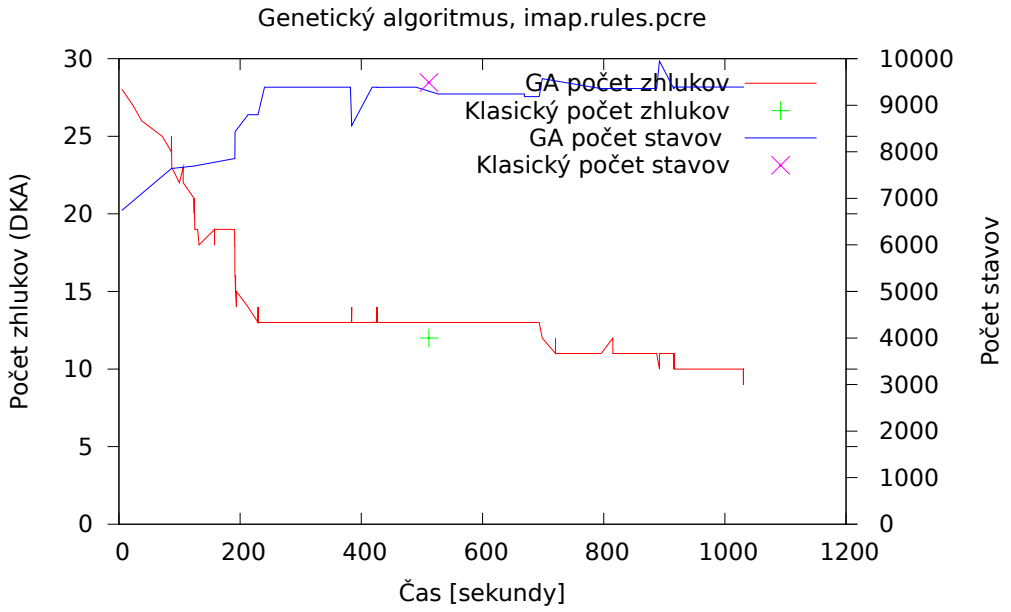
Nevyskytol sa ani jeden prípad, v ktorom by genetický algoritmus dospel k lepšiemu alebo porovnateľnému riešeniu rýchlejšie ako klasický zhlukovací algoritmus. V dvoch prípadoch však nastala situácia, kedy dospel k výrazne lepšiemu riešeniu, no za dlhšiu dobu. Konkrétne toto nastalo u sady pravidiel `exploit.rules.pcre` a `imap.rules.pcre`.



Obrázek 6.7: Porovnanie riešení genetického algoritmu a klasického zhlukovacieho algoritmu na sade pravidiel `exploit.rules.pcre`

Na obrázku 6.7 je vidieť, že genetický algoritmus dospel k lepšiemu len o málo neskôr ako klasický zhlukovací algoritmus a v dvojnásobnej dobe oproti nemu sa mu podarilo zredukovať počet zhlukov na 11 oproti 15.

Podobnú situácia je možné vidieť na grafe z obrázku 6.8. V tomto prípade genetický algoritmus dospel k lepšiemu riešeniu asi za 1,5-násobok času oproti klasickému zhlukovaciemu algoritmu, dokázal však zredukovať počet zhlukov na 9 oproti 12.



Obrázek 6.8: Porovnanie riešení genetického algoritmu a klasického zhlukovacieho algoritmu na sade pravidiel `imap.rules.pcre`

Genetický algoritmus v tejto forme nemožno podľa výsledkov vykonaných experimentov označiť za lepší ako klasický zhlukovací algoritmus. Dosiahol síce lepšie riešenie, ale iba v dvoch prípadoch a trvalo mu to dlhšie. Pri experimentovaní s genetickým algoritmom sme však narazili na variantu, ktorá sa správala podstatne lepšie.

6.5 Použitie Random-search

Pri experimentovaní s parametrami genetického algoritmu sa ukázalo, že pri nízkej pravdepodobnosti kríženia chromozómov a vyššej pravdepodobnosti mutácie sme sa rýchlejšie približovali k referenčnému riešeniu. Pri niektorých sadách pravidiel pomohlo aj znížiť počet jedincov v populácii. Tieto skúsenosti nás priviedli k použitiu algoritmu Random Search.

V tejto metóde sme využili mutácie z genetického algoritmu. Mutácia spočívala v náhodnom výbere klasickej mutácie aj mutácie spájania automatov v pravdepodobnostnom pomere výberu $0,2$ ku $0,8$. Tento pomer vyplynul z toho, že sa ukázalo lepšie najprv zhluky spájať a až potom sa snažiť nájsť lepšie rozloženie výrazov medzi zhlukmi.

Algoritmus 6.1 pracuje tak, že na začiatku vytvorí prvotné riešenie, ktoré bude obsahovať práve jeden výraz v každom zhluku. Toto riešenie označí za najlepšie doposiaľ nájdené. Potom algoritmus až do splnenia ukončovacej podmienky stále mutuje aktuálne najlepšie

riešenie. V prípade, že je zmutované riešenie lepšie ako aktuálne, označí ho za najlepšie aktuálne riešenie.

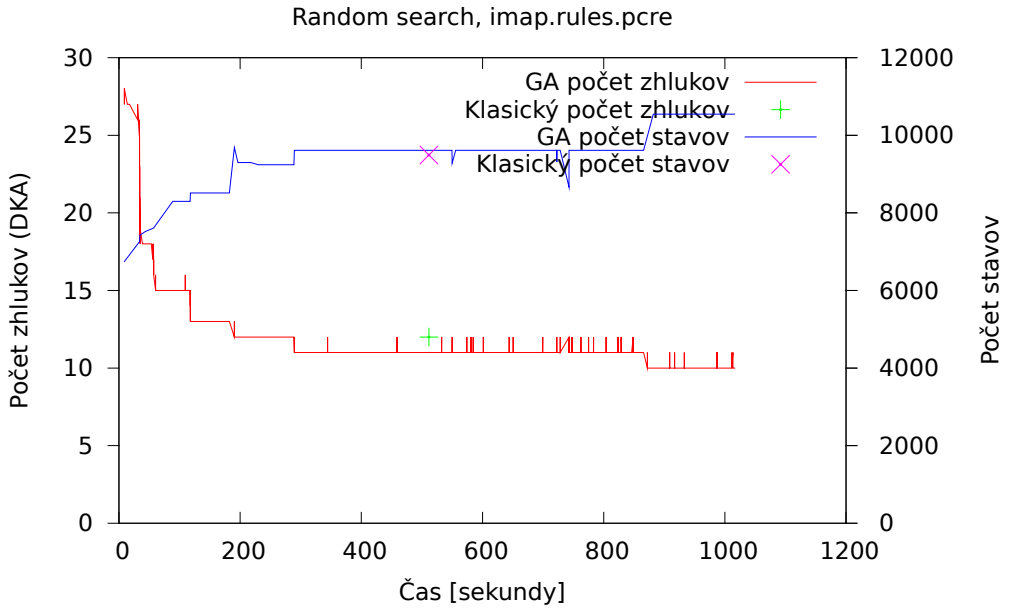
Algoritmus 6.1 Random Search pre zhlukovanie regulárnych výrazov

Vytvor počiatočné riešenie *best*, v ktorom bude každý zhluk obsahovať práve jeden regulárny výraz zo vstupnej množiny
while neplatí ukončovacia podmienka **do**
  $mutated = mutate(best)$
  if $fitness(mutated) > fitness(best)$ **then**
    $best = mutated$
  end if
end while

6.5.1 Výsledky

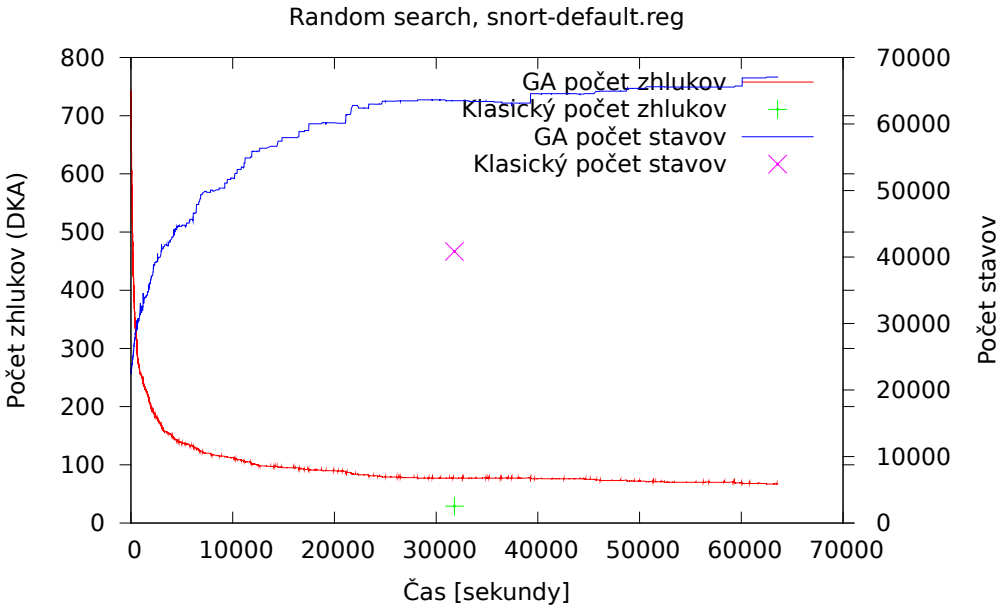
Zhlukovanie pomocou Random Search prinieslo veľmi zaujímavé výsledky. Z 18 testovaných množín výrazov dosiahol tento prístup lepší alebo porovnateľný výsledok v 10 prípadoch. U štyroch sád pravidiel dospel Random Search k porovnateľnému výsledku za menej ako polovicu trvania klasického zhlukovacieho algoritmu na danej množine pravidiel.

Na obrázku 6.9 je vidieť situácia, kedy algoritmus našiel lepšie riešenie rýchlejšie a následne sa podarilo nájsť riešenie s ešte menším počtom zhlukov.



Obrázek 6.9: Porovnanie riešení algoritmu Random Search a klasického zhlukovacieho algoritmu na sade pravidiel `imap.rules.pcre`

Prípad, v ktorom sa nepodarilo nájsť riešenie rýchlejšie nastal napr. u množiny pravidiel `snort.default.reg` (viz graf na obrázku 6.10). Na tomto grafe je dobre vidieť previazanosť počtu stavov a počtu zhlukov. Počet stavov stúpa priamo úmerne s klesaním počtu zhlukov. Tiež je vidieť ako sa pri behu algoritmu spomaľuje rýchlosť nájdenia lepšieho riešenia v čase.



Obrázek 6.10: Porovnanie riešení algoritmu Random Search a klasického zhlukovacieho algoritmu na sade pravidiel `snort.default.reg`

Grafy zo zvyšných množín pravidiel je možné nájsť v Prílohe C.

Genetický zhlukovací algoritmus, ako aj algoritmus Random Search sme implementovali v rámci knižnice Netbench [20] ako metódy triedy pre zhlukovanie regulárnych výrazov.

Kapitola 7

Záver

V práci bolo vysvetlené, prečo je detekcia útokov definovaných pomocou regulárnych výrazov v sieťovej prevádzke náročná úloha. Taktiež sme ukázali úlohu konečných automatov v IDS systémoch. Problém nastáva pri snahe detekovať viacero pravidiel súčasne, kedy môže výrazne narásť časová a priestorová zložitosť detekcie. Tento problém dokáže aspoň sčasti vyriešiť zhlukovanie, ktoré v čase predspracovania konečných automatov na detekciu môže zredukovať priestorovú náročnosť spojením viacerých pravidiel do jedného KA a súčasne používa deterministický KA, pomocou ktorého je možné spracovávať jednotlivé znaky monitorovanej prevádzky v konštantnom čase.

Zaviedli sme zhlukovací algoritmus, ktorý rieši problém nárastu stavov po determinizácii viacerých KA spojených do jedného NKA. Výsledok algoritmu vo forme množín výrazov, ktoré je možné spoločne spojiť a zdeterminizovať, dokáže riešiť problém nárastu stavov a zároveň použiť malé množstvo DKA.

Algoritmus 4.3 som implementoval v rámci diplomovej práce. Zvolený implementačný jazyk je Python, keďže cieľom bolo zakomponovať tento algoritmus do knižnice Netbench [20] vyvíjaný v rámci výskumnej skupiny ANT (Accelerated Network Technologies) na FIT VUT v Brně. Skupina ANT sa zaoberá možnosťami urýchlenia sieťových monitorovacích a bezpečnostných aplikácií.

Implementácia je zakomponovaná vo forme samostatnej triedy, ktorá by mala byť kompatibilná s ostatnými časťami knižnice. V tejto triede je možné použiť vlastnú funkciu ohodnocujúcu zložitosť konečného automatu, na rozdiel od základného algoritmu, ktorý uvažuje iba počet stavov.

Diplomová práca sa zaoberala prístupmi, ktorých cieľom bolo zlepšiť vlastnosti zhlukovania použitím nových metód. Následne sme ich výsledky porovnali medzi sebou a zhodnotili. Väčšina prístupov mala výsledky lepšie ako klasický zhlukovací algoritmu a všetky sú implementované v Netbench.

Ďalším prístupom, ktorý sme implementovali, bol genetický algoritmus na zhlukovanie regulárnych výrazov. Výsledky tohto prístupu neboli príliš presvedčivé. Detailnou analýzou populácie riešení sme dospeli k novému evolučnému algoritmu, ktorý dokázal hľadať riešenia efektívnejšie ako genetický algoritmus. Použitým algoritmom bol Random Search, ktorý dokázal nájsť lepšie alebo porovnateľné riešenie minimálne v polovici testovaných prípadov. V ostatných prípadoch sa však nedokázal tesnejšie priblížiť referenčnému riešeniu.

Výsledky Random Search sú podľa nás veľmi zaujímavé minimálne do tej miery, že takýto jednoduchý algoritmus sa reálne priblížil k sofistikovanejšiemu a zložitejšiemu algoritmu. Preto si myslíme, že použitie iných evolučných algoritmov by malo byť predmetom ďalšieho výskumu. Evolučné algoritmy ponúkajú hľadanie riešenia aj pomocou distribuova-

ných výpočtov a teda práca nie je závislá iba na jednom počítači a môže byť tak rozdelená medzi mnoho počítačov a nakoniec tak dokáže nájsť riešenie omnoho rýchlejšie ako klasický zhlukovací algoritmus.

Literatura

[1] Ciampa, M.: *Security+ Guide to Network Security Fundamentals*. Cengage Learning, 2008, ISBN 9781428340664.

[2] Clark, C. R.; Schimmel, D. E.: Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *In Proceedings of 13th International Conference on Field Program*, 2003, s. 956–959.

[3] Clark, C. R.; Schimmel, D. E.: Scalable Pattern Matching for High Speed Networks. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7695-2230-0, s. 249–257.

[4] Koza, J.: *On the programming of computers by means of natural selection*. A Bradford book, MIT Press, 1996, ISBN 9780262111706.

[5] Kumar, S.; Dharmapurikar, S.; Yu, F.; aj.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-308-5, s. 339–350.

[6] Kvasnička, V.; Pospíchal, J.; Tiňo, P.: *Evolučné algoritmy*. Vydavateľstvo STU, 2000, ISBN 8022713775.

[7] Michalewicz, Z.; Fogel, D.: *How to solve it: modern heuristics*. Springer, 2004, ISBN 9783540224945.

[8] Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks*, 1999, s. 2435–2463.

[9] Pietro, R.; Mancini, L.: *Intrusion detection systems*. Advances in information security, Springer, 2008, ISBN 9780387772653.

[10] Poli, R.; Langdon, W.; McPhee, N.: *A Field Guide to Genetic Programming*. Lulu.com, 2008, ISBN 9781409200734.

[11] Roesch, M.; Telecommunications, S.: Snort - Lightweight Intrusion Detection for Networks. 1999, s. 229–238.

[12] Sidhu, R.; Prasanna, V. K.: Fast Regular Expression Matching Using FPGAs. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA: IEEE Computer Society, 2001, ISBN 0-7695-2667-5, s. 227–238, doi:10.1109/FCCM.2001.22.

[13] Smith, R.; Estan, C.; Jha, S.: Backtracking Algorithmic Complexity Attacks against a NIDS. *Computer Security Applications Conference, Annual*, 2006, ISSN 1063-9527.

[14] Smith, R.; Estan, C.; Jha, S.; aj.: Fast Signature Matching Using Extended Finite Automaton (XFA). In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 978-3-540-89861-0, s. 158–172.

[15] Sommer, R.; Paxson, V.: Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM conference on Computer and communications security*, ACM, 2003, ISBN 1-58113-738-9.

[16] Whitman, M.; Mattord, H.: *Principles of Information Security*. Course Technology Ptr, 2011, ISBN 9781111138219.

[17] WWW stránky: Application Layer Packet Classifier for Linux [online]. `http://l7-filter.sourceforge.net/`, [cit. 2010-11-22].

[18] WWW stránky: AT&T Labs Research - FSM Library [online]. `http://www2.research.att.com/ fsmtools/fsm/`, [cit. 2011-05-10].

[19] WWW stránky: deap - Distributed Evolutionary Algorithms in Python [online]. `http://code.google.com/p/deap/`, [cit. 2010-12-10].

[20] WWW stránky: Netbench [online]. `http://merlin.fit.vutbr.cz/ant/netbench/index.html`, [cit. 2010-11-22].

[21] Yu, F.; Chen, Z.; Diao, Y.; aj.: Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection [online]. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-76.html`, 2006.

[22] Češka, M.; Vojnar, T.; Smrčka, A.: *Teoretická informatika*. Brno: FIT VUT v Brně, 2009.

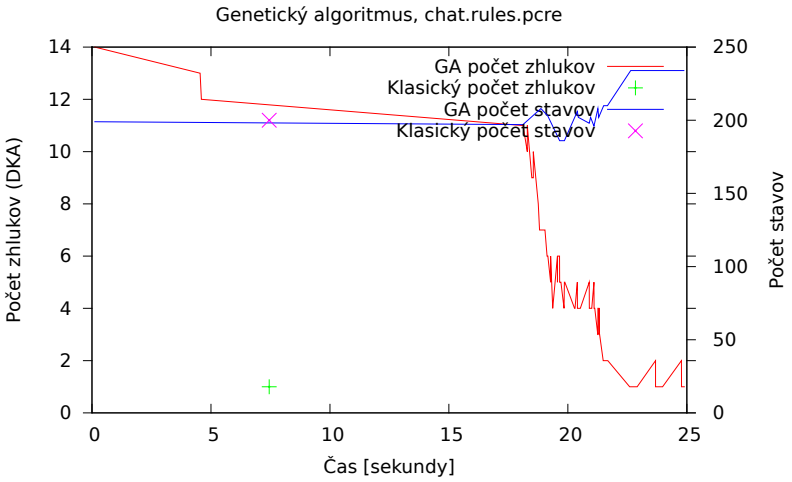
Příloha A

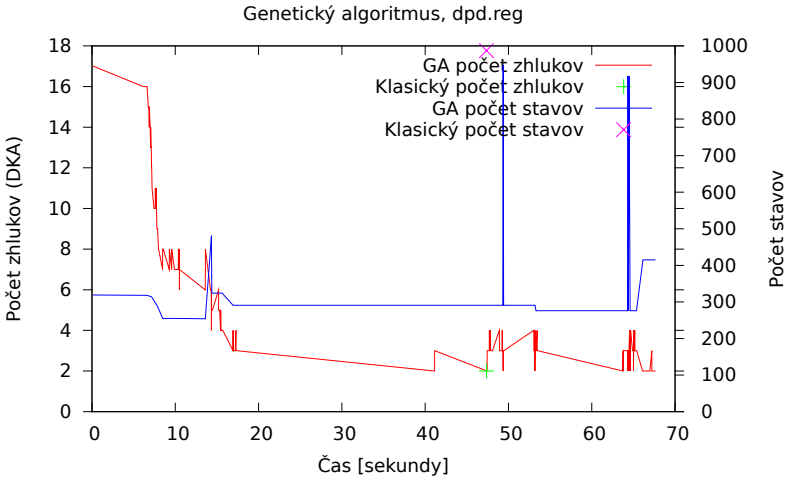
Obsah CD

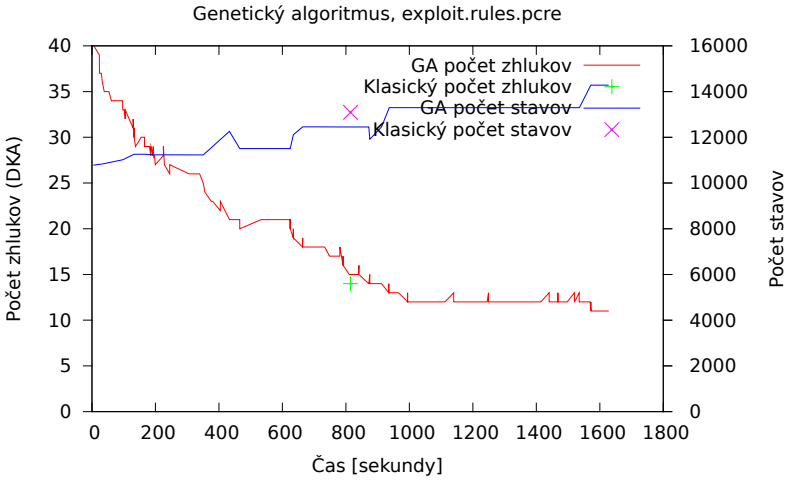
1. Text práce v zdrojových súboroch (latex),
2. Zdrojové súbory implementovaných metód,
3. Súbory s výsledkami testov.

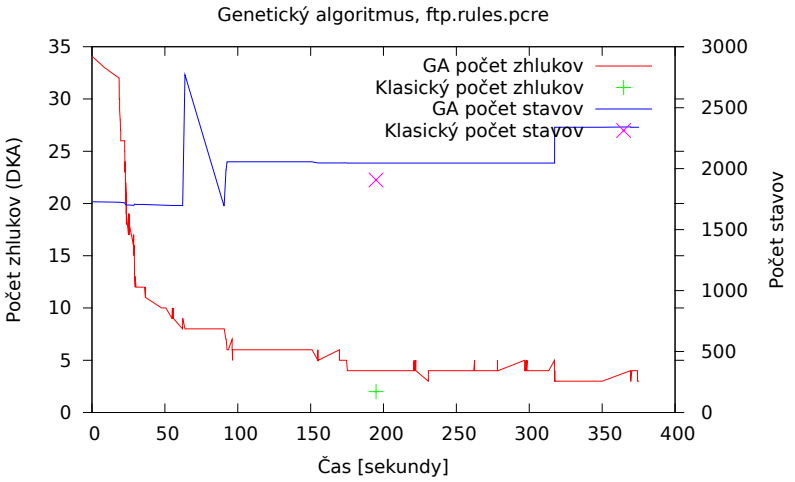
Příloha B

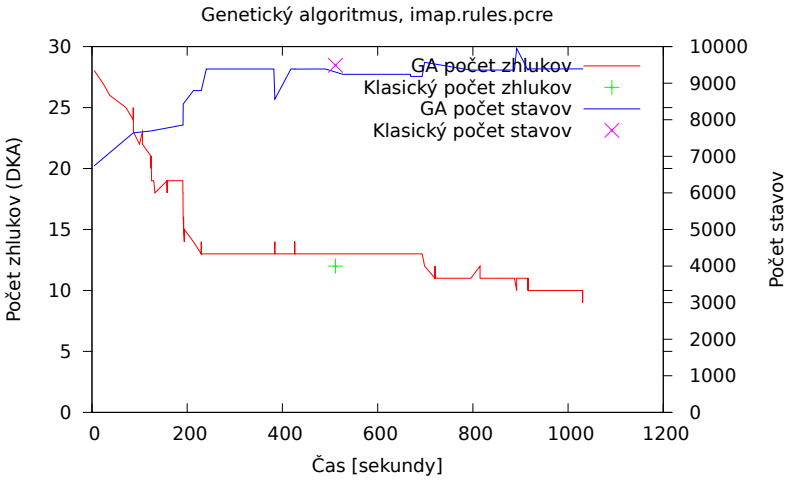
Výsledky genetického algoritmu

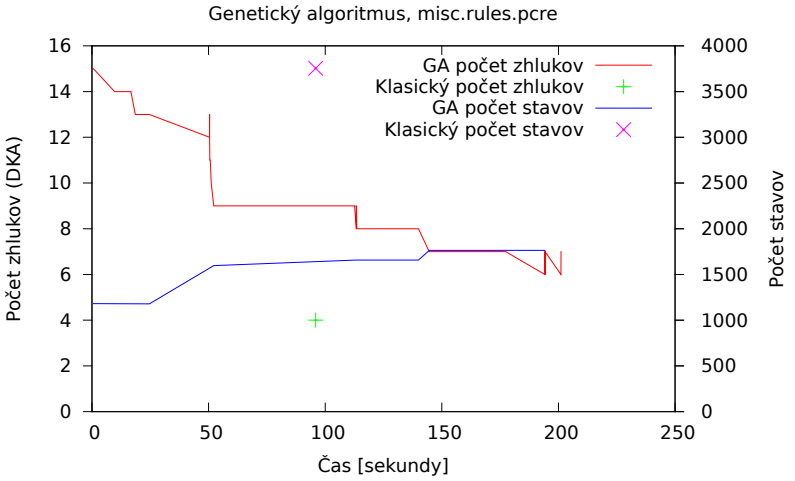


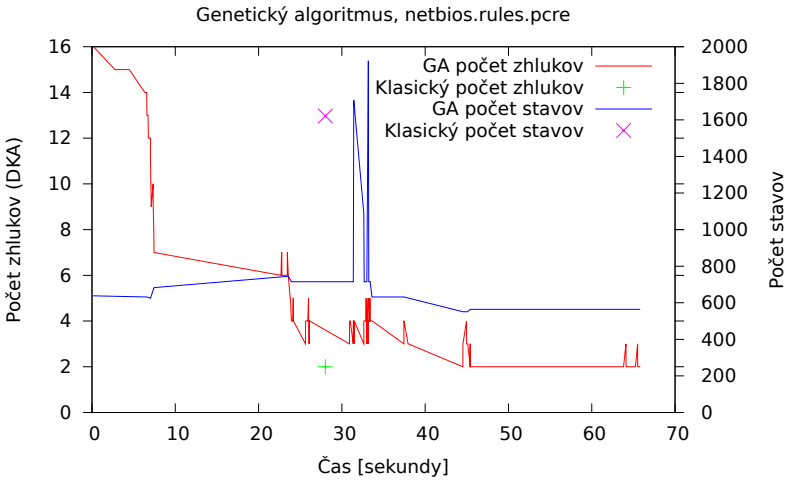


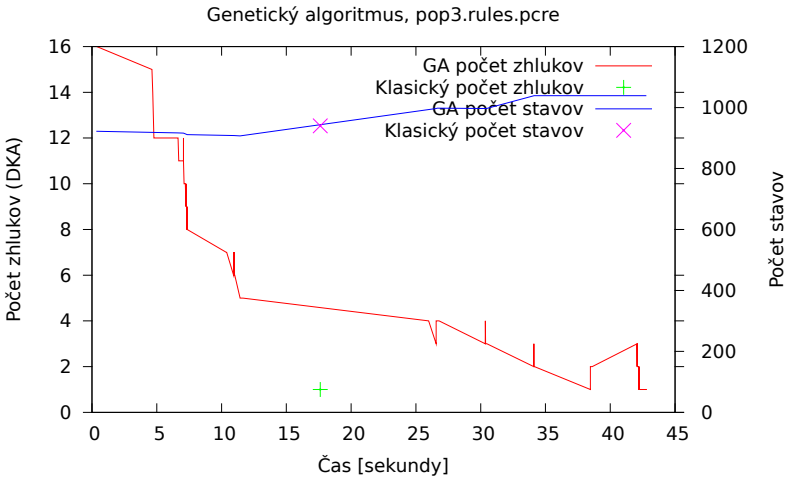

Genetický algoritmus, exploit.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
35
40
0
2000
4000
6000
8000
10000
12000
14000
16000
0
200
400
600
800
1000
1200
1400
1600
1800

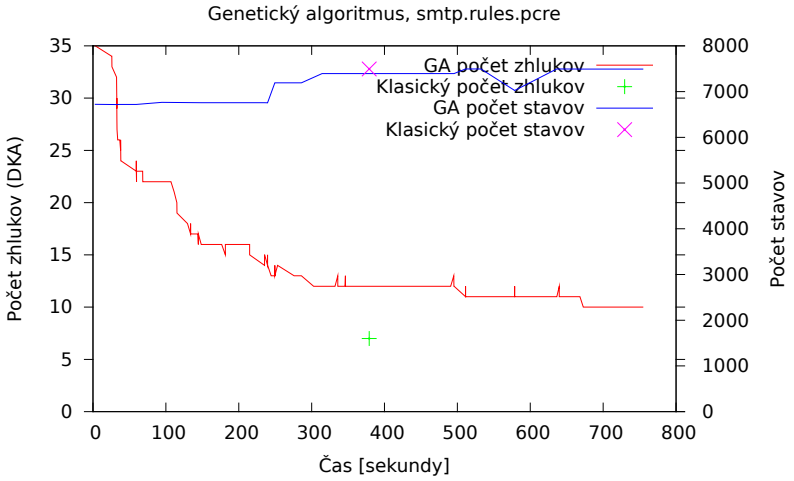

Genetický algoritmus, ftp.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
35
0
500
1000
1500
2000
2500
3000
0
50
100
150
200
250
300
350
400

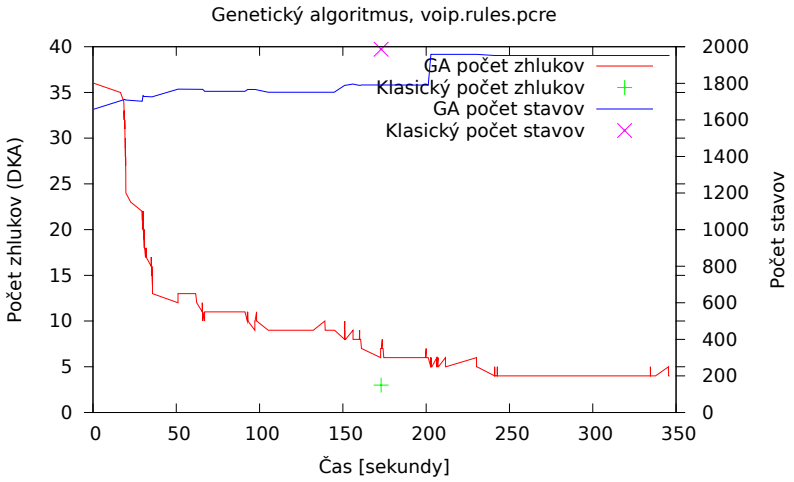

Genetický algoritmus, imap.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
200
400
600
800
1000
1200

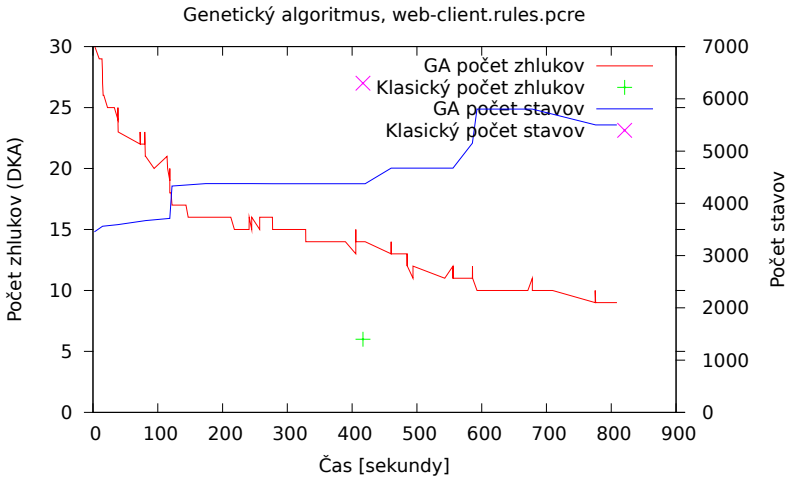

Genetický algoritmus, misc.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
2
4
6
8
10
12
14
16
0
500
1000
1500
2000
2500
3000
3500
4000
0
50
100
150
200
250

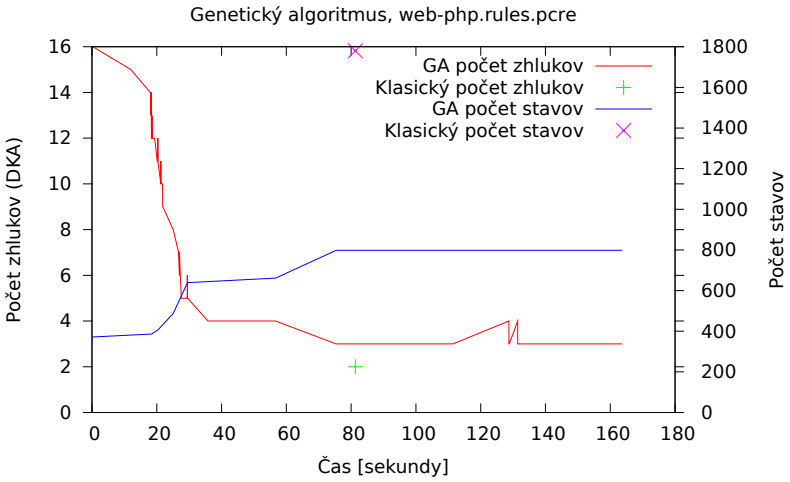

Genetický algoritmus, netbios.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
2
4
6
8
10
12
14
16
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
10
20
30
40
50
60
70


Genetický algoritmus, pop3.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
2
4
6
8
10
12
14
16
0
200
400
600
800
1000
1200
0
5
10
15
20
25
30
35
40
45


Genetický algoritmus, smtp.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
35
0
1000
2000
3000
4000
5000
6000
7000
8000
0
100
200
300
400
500
600
700
800

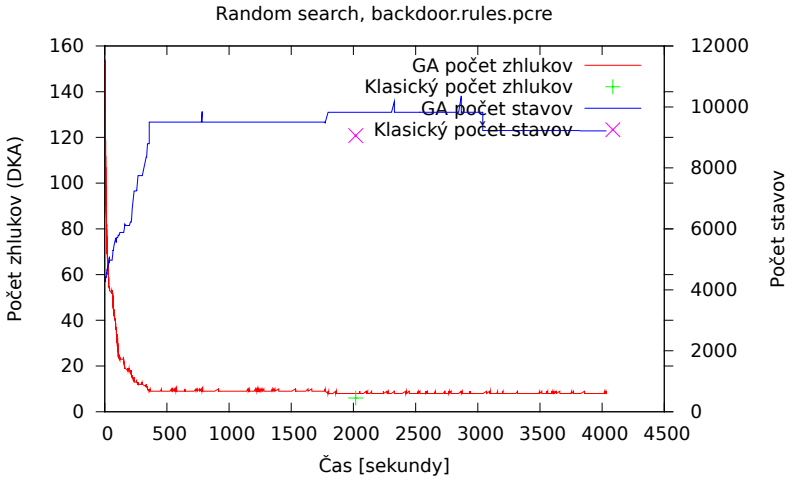

Genetický algoritmus, voip.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
35
40
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
50
100
150
200
250
300
350

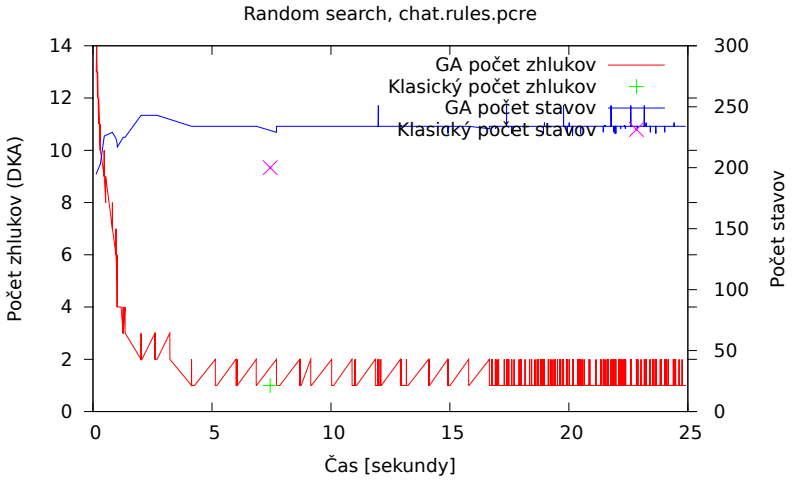

Genetický algoritmus, web-client.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
0
1000
2000
3000
4000
5000
6000
7000
0
100
200
300
400
500
600
700
800
900

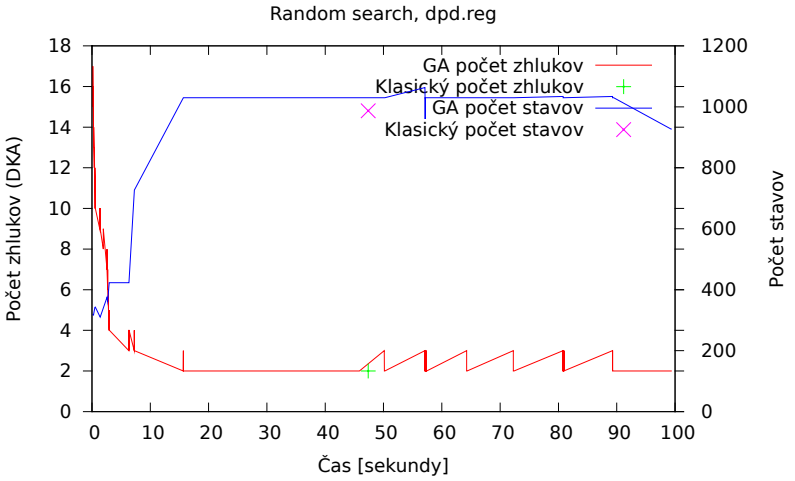

Genetický algoritmus, web-php.rules.pcre
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
0
2
4
6
8
10
12
14
16
0
200
400
600
800
1000
1200
1400
1600
1800
0
20
40
60
80
100
120
140
160
180

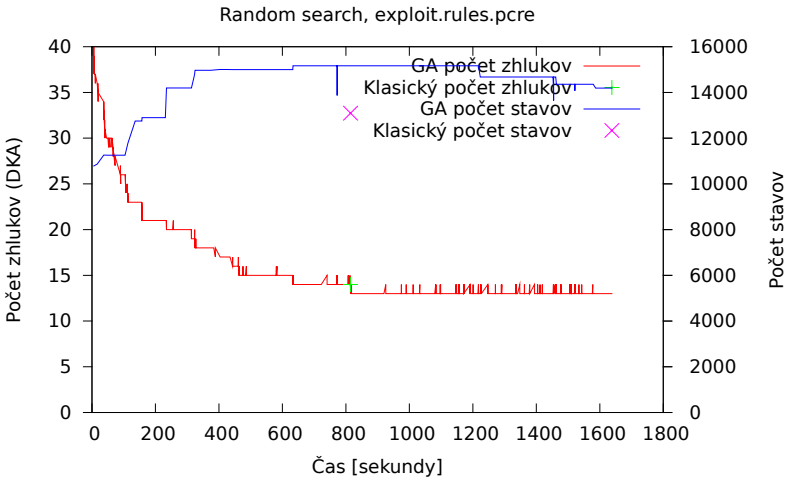
Příloha C

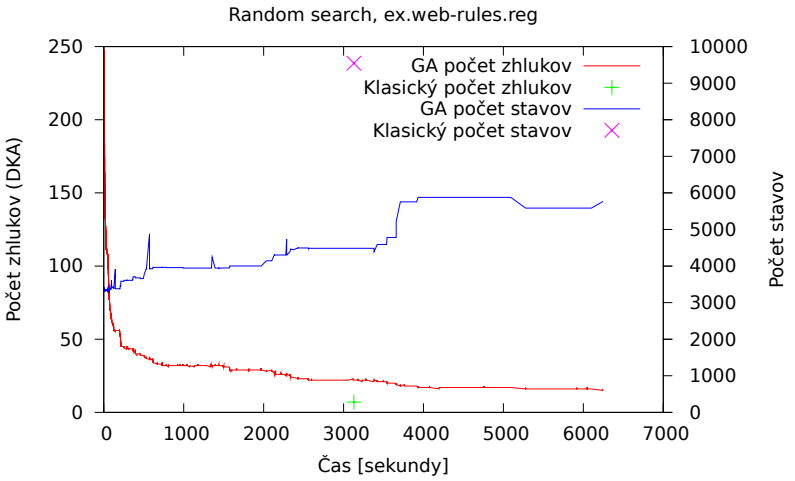
Výsledky algoritmu Random search

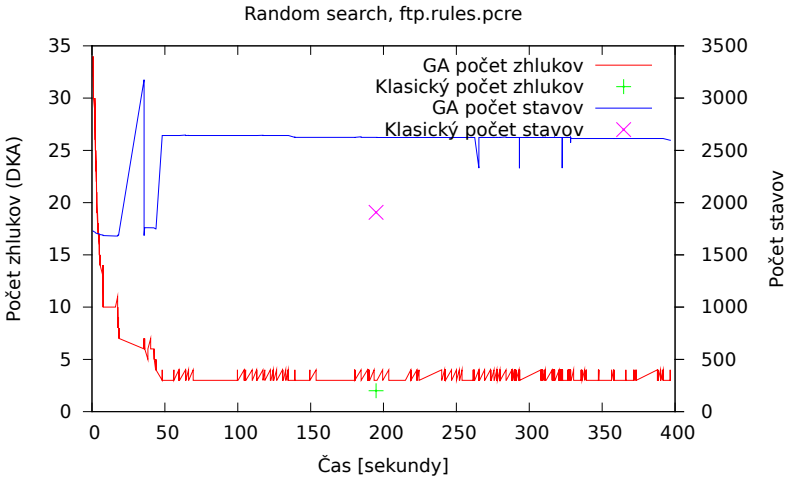


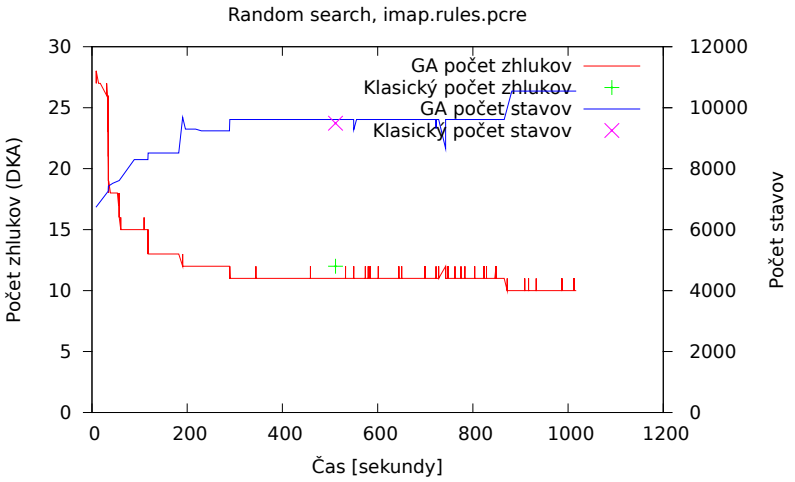


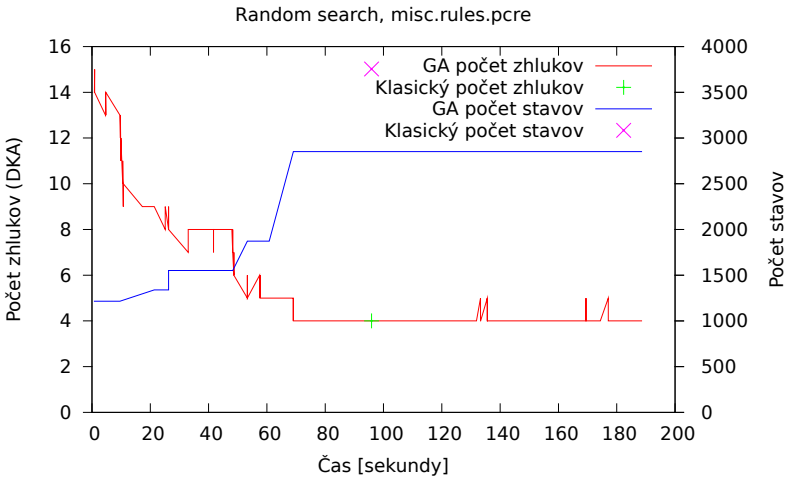

Random search, dpd.reg
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
2
4
6
8
10
12
14
16
18
0
200
400
600
800
1000
1200
0
10
20
30
40
50
60
70
80
90
100

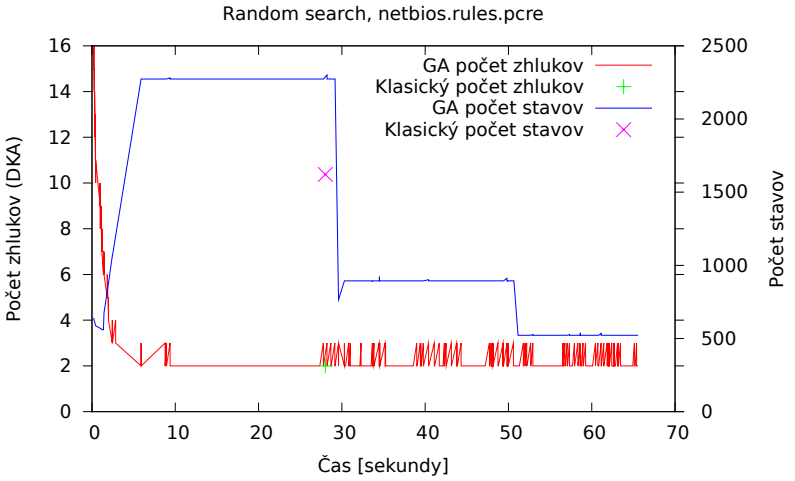

Random search, exploit.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
35
40
0
2000
4000
6000
8000
10000
12000
14000
16000
0
200
400
600
800
1000
1200
1400
1600
1800

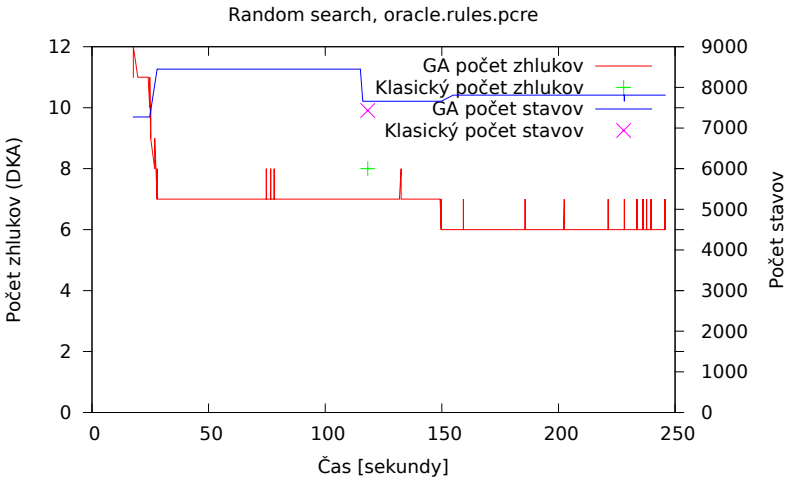

Random search, ex.web-rules.reg
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
50
100
150
200
250
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
0
1000
2000
3000
4000
5000
6000
7000

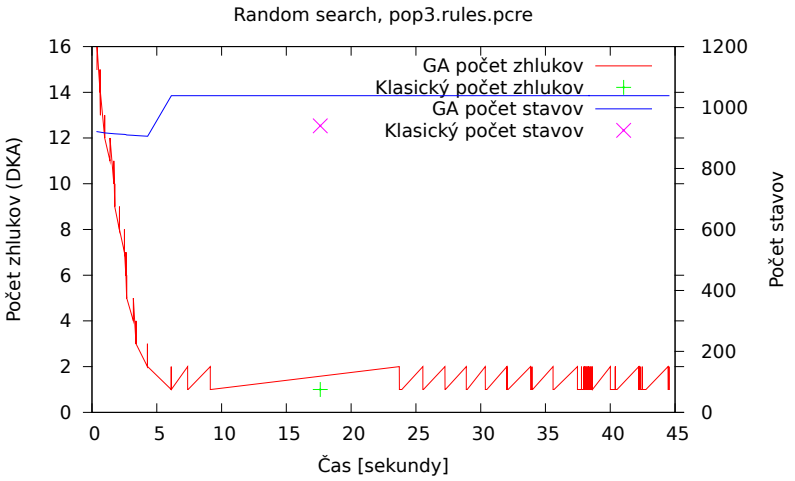

Random search, ftp.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
35
0
500
1000
1500
2000
2500
3000
3500
0
50
100
150
200
250
300
350
400

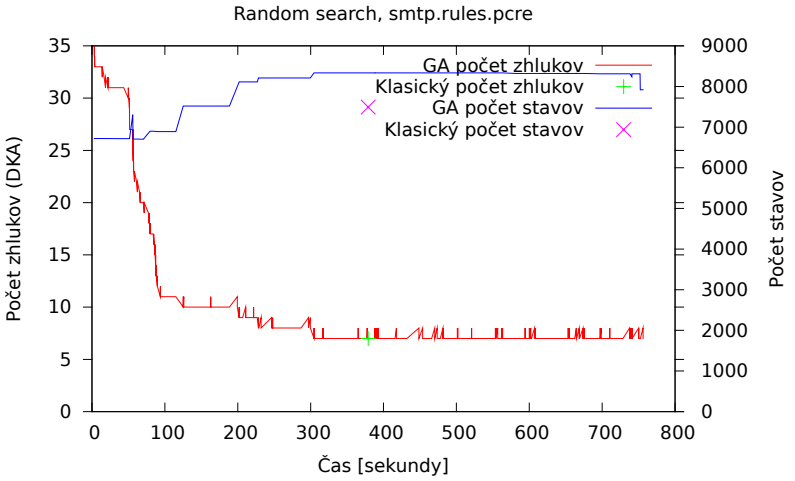

Random search, imap.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
0
2000
4000
6000
8000
10000
12000
0
200
400
600
800
1000
1200

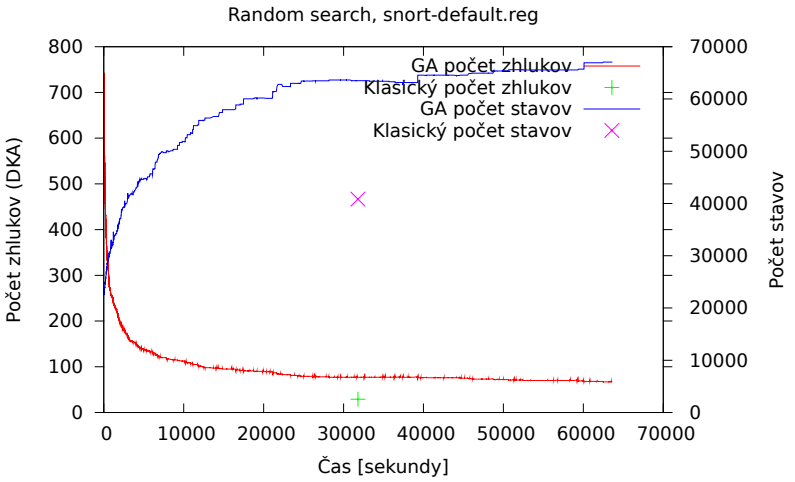

Random search, misc.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
2
4
6
8
10
12
14
16
0
500
1000
1500
2000
2500
3000
3500
4000
0
20
40
60
80
100
120
140
160
180
200

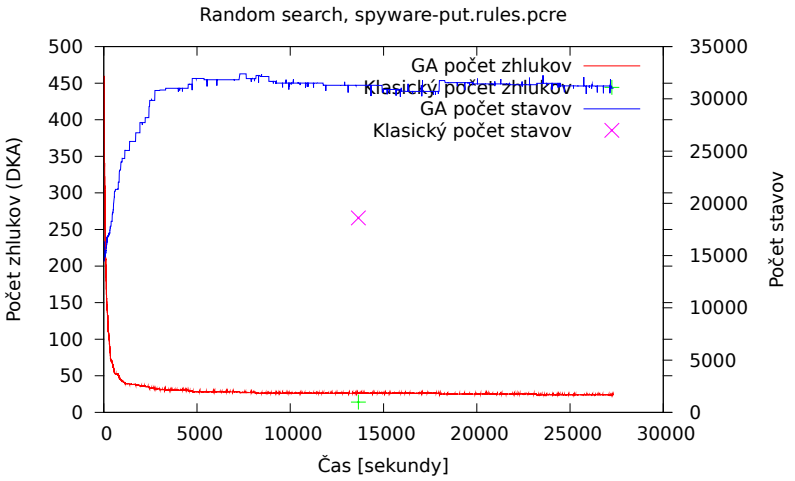

Random search, netbios.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
2
4
6
8
10
12
14
16
0
500
1000
1500
2000
2500
0
10
20
30
40
50
60
70

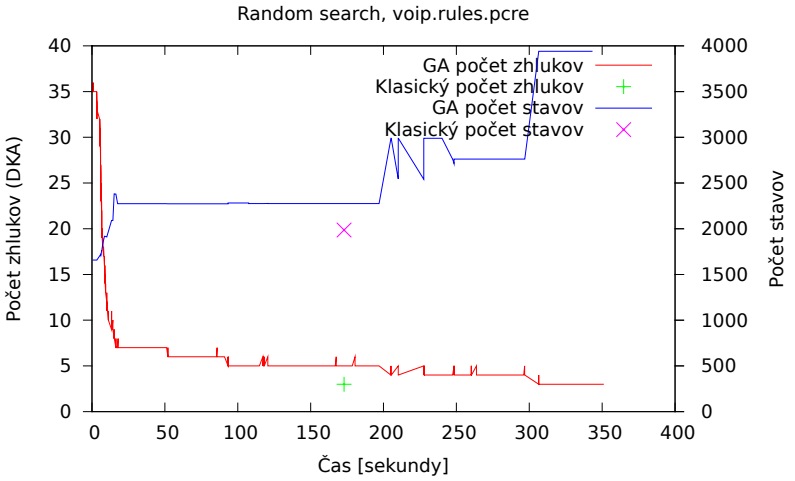

Random search, oracle.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
2
4
6
8
10
12
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
0
50
100
150
200
250

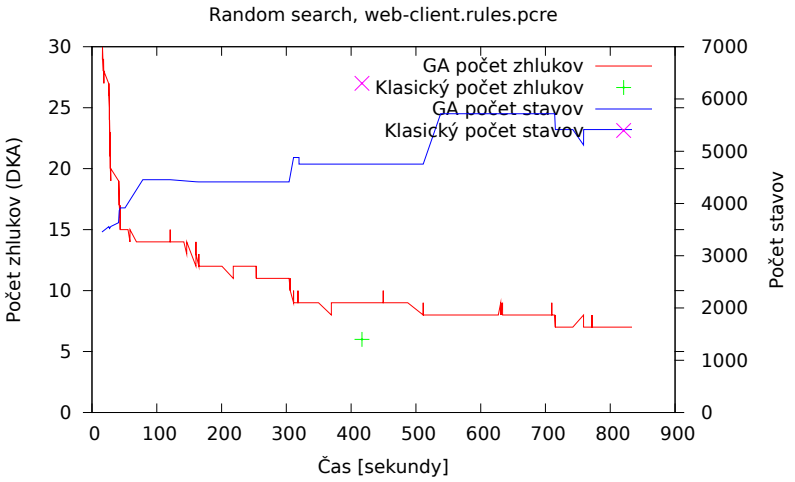

Random search, pop3.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
2
4
6
8
10
12
14
16
0
200
400
600
800
1000
1200
0
5
10
15
20
25
30
35
40
45

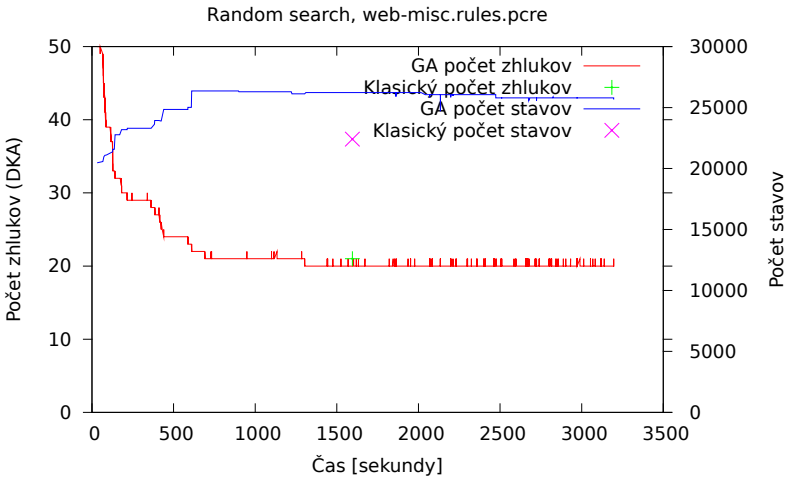

Random search, smtp.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
35
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
0
100
200
300
400
500
600
700
800

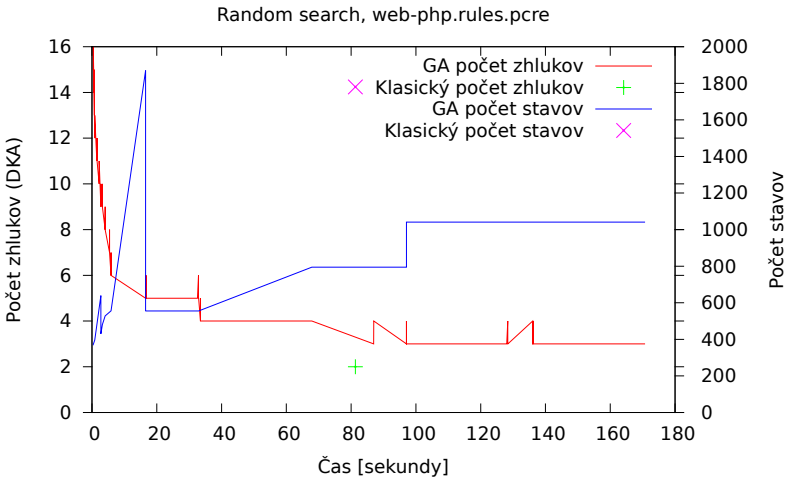

Random search, snort-default.reg
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
100
200
300
400
500
600
700
800
0
10000
20000
30000
40000
50000
60000
70000
0
10000
20000
30000
40000
50000
60000
70000


Random search, spyware-put.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
50
100
150
200
250
300
350
400
450
500
0
5000
10000
15000
20000
25000
30000
35000
0
5000
10000
15000
20000
25000
30000


Random search, voip.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
35
40
0
500
1000
1500
2000
2500
3000
3500
4000
0
50
100
150
200
250
300
350
400


Random search, web-client.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
5
10
15
20
25
30
0
1000
2000
3000
4000
5000
6000
7000
0
100
200
300
400
500
600
700
800
900


Random search, web-misc.rules.pcre
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
0
10
20
30
40
50
0
5000
10000
15000
20000
25000
30000
0
500
1000
1500
2000
2500
3000
3500


Random search, web-php.rules.pcre
Počet zhlukov (DKA)
Počet stavov
Čas [sekundy]
GA počet zhlukov
Klasický počet zhlukov
GA počet stavov
Klasický počet stavov
0
2
4
6
8
10
12
14
16
0
200
400
600
800
1000
1200
1400
1600
1800
2000
0
20
40
60
80
100
120
140
160
180