Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Rozpoznávanie regulárnych výrazov

(bakalárska práca)

Pavel Labath

Vedúci: RNDr. Michal Forišek

Bratislava, 2008

Abstrakt

Regulárne výrazy sa dnes používajú v textových procesoroch, lexikálnych analyzátoroch a mnohých iných oblastiach informatiky. Na rozpoznávanie regulárnych výrazov existuje viacero algortimov a každý má svoje prednosti a nedostatky. V tejto práci popisujeme algoritmus používajúci konečné automaty a experimentálne porovnávame jeho implementáciu s inými riešeniami.

Kľúčové slová: regulárne výrazy, konečné automaty, algoritmy, vyhľadávanie, zložitosť

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne s použitím citovaných zdrojov.

.....

Touto cestou by som sa chcel poďakovať môjmu školiteľovi, RNDr. Michalovi Foriškovi, za pomoc pri hľadaní zaujímavej témy, ako aj za neoceniteľné rady pri jej spracovaní.

Obsah

1 Úvod **1**

2 Základné definície a označenia **2**
2.1 Regulárne výrazy 2
2.2 Konečné automaty 3

3 Použitie regulárnych výrazov **5**
3.1 Vyhľadávanie vzorky v texte 5
3.2 Nahrádzanie vzorky 6
3.3 Validácia vstupu 7
3.4 *Neregulárne* regulárne výrazy 7
3.5 Varianty regulárnych výrazov 7

4 Algoritmus **9**
4.1 Regulárny výraz → NKA 9
4.2 NKA → DKA 12
4.3 Hľadanie podreťazcov 14
4.4 O zložitosti 16
4.5 Rozpoznávanie pomocou NKA 16

5 Testovanie **18**
5.1 Spracovanie regulárneho výrazu 18
5.1.1 Čistý text 18
5.1.2 Alternatívy 19
5.1.3 Iterácia pomocou '`{}`' 19
5.2 Vyhľadávanie v texte 19

6 Záver **23**

Literatúra **24**

A Testovacie skripty **26**

Zoznam obrázkov

4.1 Automat vyrobený naivným algoritmom 13
4.2 Automat vyrobený vylepšeným algoritmom 13
4.3 DKA pre zjednodušený zápis času 15

5.1 Čas spracovania čistého textu 21
5.2 Čas spracovania alternatív 21
5.3 Čas spracovania ⌜a.{n}a⌟ programom foogrep 22

Zoznam tabuliek

5.1 Čas spracovania čistého textu 20
5.2 Čas spracovania alternatív 20
5.3 Čas spracovania ⌜a.{n}a⌟ programom foogrep 22
5.4 Vzťah času vyhľadávania a počtu riadkov obsahujúcich hľadané reťazce 22

Kapitola 1

Úvod

Regulárne výrazy zaviedol ešte Kleene v roku 1956[1] ako prostriedok na jednoduché popisovanie udalostí v neurónových sietiach. Dnes sa regulárne výrazu používajú na rôzne činnosti súvisiace so spracovaním textu. Schopnosť jednoducho popísať niektoré množiny reťazcov im zabezpečila miesto v texových editoroch, lexikálnych analyzátoroch a v mnohých iných oblastiach informatiky. V tejto práci prinášame stručný prehľad niektorých aplikácii regulárnych výrazov. Cieľom je aj navrhnúť a implementovať algoritmus na testovanie príslušnosti slova do jazyka definovaného regulárnym výrazom.

Predpokladáme, že sa čitateľ už stretol s pojmami „regulárny výraz“, „deterministický konečný automat“ a „nedeterministický konečný automat“ v rámci teórie formálnych jazykov. Aby sme však predišli nedorozumeniam, v kapitole 2 uvádzame definície týchto pojmov, ktoré budeme ďalej v práci používať.

V kapitole 3 uvádzame príklady úloh, ktoré sa dajú ľahko riešiť pomocou regulárnych výrazov, ako aj prehľad variantov (angl. *flavour*) regulárnych výrazov, ktoré sa na to používajú.

Kapitola 4 tvorí jadro práce. Uvádzame v nej algoritmus na zostrojenie ekvivalentného deterministického konečného automatu k danému regulárnemu výrazu (použijúc nedeterministický konečný automat ako medzikrok), čím dosiahneme lineárnu časovú zložitosť testovania príslušnosti slov do jazyka. Zmienime sa aj o časovej a priestorovej zložitosti prekladu a o alternatívnych prístupoch k rozpoznávaniu.

Kapitola 5 obsahuje výsledky testov porovnávajúcich algoritmus z kapitoly 4 s programom GNU grep. Porovnanie robíme na základe času predspracovania a času rozpoznávania.

Kapitola 2

Základné definície a označenia

Ústredným pojmom celej prace je pojem regulárneho výrazu. Uvádzame preto jeho definiciu. Pre ich vizuálne odlíšenie od zvyška textu budeme regulárne výrazy uvádzať v ⌜⌟.

2.1 Regulárne výrazy

Definícia 2.1.1 (Element triedy znakov)**.** *Nech Σ je abeceda, $a <_\Sigma$ je usporiadanie na Σ. Nech $a, b \in \Sigma$. t je element triedy znakov, ak je tvaru ⌜a⌟ resp. ⌜a-b⌟. Jazyk prisluchajúci t je $L_t = \{a\}$ resp. $L_t = \{c \in \Sigma \mid a \leq_\Sigma c \leq_\Sigma b\}$.*

Definícia 2.1.2 (Trieda znakov)**.** *Nech $t_1, t_2, \ldots, t_n$ sú elementy triedy znakov. T je trieda znakov, ak je tvaru ⌜`[`$t_1 t_2 \ldots t_n$`]`⌟ resp. ⌜`[^`$t_1 t_2 \ldots t_n$`]`⌟. Jazyk prisluchajúci T je $L_T = \bigcup_{i=1}^{n} L_{t_i}$ resp. $L_T = \Sigma \setminus \bigcup_{i=1}^{n} L_{t_i}$.*

Definícia 2.1.3. *Pojmy atóm, iterovaný atóm, vetva a regulárny výraz nad abecedou Σ rekurzívne definujeme nasledovne:*

- *Nech $V_1, V_2, \ldots, V_n$ sú vetvy. Potom ⌜V_1`|`V_2`|`$\cdots$`|`V_n⌟ je regulárny výraz.*
- *Nech $I_1, I_2, \ldots, I_n$ sú iterované atómy. Potom ⌜$I_1 I_2 \cdots I_n$⌟ je vetva.*
- *Nech A je atóm. Nech $n, m \in \mathbb{N}$, $n \leq m$. Potom ⌜A⌟, ⌜A`*`⌟, ⌜A`+`⌟, ⌜A`?`⌟, ⌜A`{`n`}`⌟, ⌜A`{`n`,`m`}`⌟ a ⌜A`{`n`,}`⌟ sú iterované atómy.*
- *Nech R je regulárny výraz. Nech T je trieda znakov. Nech $a \in \Sigma$. Potom ⌜`(`R`)`⌟, ⌜T⌟, ⌜a⌟ a ⌜`.`⌟ sú atómy.*

Definícia 2.1.4. *Jazyky prislúchajúce atómom, iterovaným atómom, vetvám a regulárnym výrazom rekurzívne definujeme nasledovne:*

- *Nech* $R = \ulcorner V_1 \texttt{|} V_2 \texttt{|} \cdots \texttt{|} V_n \lrcorner$ *je regulárny výraz. Potom jazyk prislúchajúci k* R *je* $L_R = \bigcup_{i=1}^{n} L_{V_i}$.
- *Nech* $V = \ulcorner I_1 I_2 \cdots I_n \lrcorner$ *je vetva. Potom jazyk prislúchajúci k* V *je* $L_V = L_{I_1} L_{I_2} \cdots L_{I_n}$.
- *Nech* $I_1 = \ulcorner A \lrcorner$, $I_2 = \ulcorner A\texttt{*} \lrcorner$, $I_3 = \ulcorner A\texttt{+} \lrcorner$, $I_4 = \ulcorner A\texttt{?} \lrcorner$, $I_5 = \ulcorner A\texttt{\{}n\texttt{\}} \lrcorner$, $I_6 = \ulcorner A\texttt{\{}n\texttt{,}m\texttt{\}} \lrcorner$ *a* $I_7 = \ulcorner A\texttt{\{}n\texttt{,\}} \lrcorner$ *sú iterované atómy.*
  Potom $L_{I_1} = L_A$, $L_{I_2} = {L_A}^*$, $L_{I_3} = {L_A}^+$, $L_{I_4} = L_A \cup \{\varepsilon\}$, $L_{I_5} = {L_A}^n$, $L_{I_6} = \bigcup_{i=n}^{m} {L_A}^i$ *a* $L_{I_7} = \bigcup_{i=n}^{\infty} {L_A}^i$.
- *Nech* $A_1 = \ulcorner \texttt{(}R\texttt{)} \lrcorner$, $A_2 = \ulcorner T \lrcorner$, $A_3 = \ulcorner a \lrcorner$ *a* $A_4 = \ulcorner \texttt{.} \lrcorner$.
  Potom $L_{A_1} = L_R$, $L_{A_2} = L_T$, $L_{A_3} = \{a\}$ *a* $L_{A_4} = \Sigma$.

Definícia 2.1.5. *Dĺžka regulárneho výrazu* R*, v označení* $|R|$*, je počet symbolov potrebný na jeho zápis. Čísla* n *a* m *v iterovaných atómoch píšeme v desiatkovej sústave.*

Čitateľ, ktorý sa stretol s regulárnymi výrazmi v rámci teórie formálnych jazykov, si všimne, že sme nepoužili definíciu, ktorá, s zvykne uvádzať v prácach v tomto odbore (viď napríklad [1],[2]). Najväčšia časť tejto práce je venovaná použitiu regulárnych výrazov na manipuláciu textu, a preto budeme používať definície pojmov, ktoré už v tejto oblasti existujú. Uvedené definície vznikli formalizovaním (a zjednodušením) popisu „Rozšírených regulárnych výrazov“ v [3].

2.2 Konečné automaty

Definícia 2.2.1. *Deterministický konečný automat (DKA) je pätica* $A = (K, \Sigma, \delta, q_0, F)$*, kde* K *je konečná množina stavov,* Σ *je (konečná) vstupná abeceda,* $\delta : K \times \Sigma \to K$ *je prechodová funkcia,* $q_0 \in K$ *je počiatočný stav a* $F \subseteq K$ *je množina akceptačných stavov.*

Definícia 2.2.2. *Konfigurácia DKA* A *je dvojica* $(q, w) \in K \times \Sigma^*$*, kde* q *je stav automatu a* w *je nespracovaná časť vstupného slova.*

Definícia 2.2.3. *Krok výpočtu DKA A je relácia $\vdash_A$ na konfiguráciách definovaná nasledovne:*

$$\forall p, q \in K \quad \forall a \in \Sigma \quad \forall w \in \Sigma^* \qquad (q, aw) \vdash_A (p, w) \overset{\text{def}}{\iff} p = \delta(q, a)$$

Relácia $\vdash_A^*$ je reflexívno-tranzitívny uzáver relácie $\vdash_A$.

Definícia 2.2.4. *Jazyk akceptovaný DKA A je množina slov $L(A) = \{w \in \Sigma^* \mid (q_0, w) \vdash_A^* (q, \varepsilon) \wedge q \in F\}$*

Definícia 2.2.5. *Nedeterministický konečný automat (NKA) je pätica $A = (K, \Sigma, \delta, q_0, F)$, kde K je konečná množina stavov, Σ je (konečná) vstupná abeceda, $\delta : K \times (\Sigma \cup \{\varepsilon\}) \to 2^K$ je prechodová funkcia, $q_0 \in K$ je počiatočný stav a $F \subseteq K$ je množina akceptačných stavov.*

Definícia 2.2.6. *Krok výpočtu NKA A je relácia $\vdash_A$ na konfiguráciách definovaná nasledovne:*

$$\forall p, q \in K \quad \forall a \in \Sigma \cup \{\varepsilon\} \quad \forall w \in \Sigma^* \qquad (q, aw) \vdash_A (p, w) \overset{\text{def}}{\iff} p = \delta(q, a)$$

Relácia $\vdash_A^*$ je reflexívno-tranzitívny uzáver relácie $\vdash_A$.

Poznámka 2.2.1. *Konfiguráciu NKA a jazyk akceptovaný NKA definujeme rovnako ako pre DKA.*

Kapitola 3

Použitie regulárnych výrazov

V tejto časti sa stručne zmienime o úlohách, ktoré sa často vyskytujú pri spracovaní textu, a ktoré sa môžu ľahko riešiť pomocou regulárnych výrazov. Taktiež spomenieme programy, ktoré sa riešia tieto úlohy a rôzne varianty regulárnych výrazov, ktoré sú pri tom použité.

3.1 Vyhľadávanie vzorky v texte

Pri spracovaní veľkého súboru často potrebujeme nájsť výskyty nejakého reťazca. Naivný bruteforce algoritmus rieši túto úlohu v čase $O(nm)$, kde n je dĺžka vzorky a m je dĺžka textu. Pre dlhé vzorky môže však vyhľadávanie týmto algoritmom trvať neakceptovateľne dlho. Existuje rad algoritmov, ktoré majú lepšiu časovu zložitosť. Zlepšenie sa dosahuje predspracovaním. Niektoré algoritmy sú založené na predspracovaní textu (napr. suffixové stromy[4]), a iné na predspracovaní vzorky (algoritmus Knuth-Morris-Pratt[5], Boyer-Moore[6] a ich modifikácie). Vyhľadávanie pomocou regulárnych výrazov sa môže zaradiť do druhej kategórie.[1] Aj keď nedosahujú rýchlosť už spomenutých algoritmov, regulárne výrazy umožňujú hľadať aj vzorky, ktoré by sa nemohli nájsť pomocou iných algoritmov.

Veľmi často používaný je program *grep*. Grep je konzolová utilita, ktorá hľadá výskyty slov definovaných regulárnym výrazom a (v základnom režime) vypisuje riadky, ktoré také slová obsahujú. Pomocou parametrov sa dá nastaviť aj tak, aby na začiatku každého riadku vypísal názov suboru alebo číslo riadku, aby vypisoval riadky, ktoré *ne*obsahujú slová definované regulár-

[1] O spôsobe predspracovania budeme hovoriť v kapitole 4

nym výrazom, len názvy súborov, ktoré také riadky obsahujú atď. Napríklad príkazom:

```
$ grep -R / -A 3 'foo.*bar'
```

možeme najsť všetky výskyty riadky vo všetkých súboroch na disku, ktoré obsahujú reťazec 'foo' a za ním reťazec 'bar'. Grep vypíše za každým riadkom obsahujúcim tieto slová ešte tri riadky kontextu.

Možnosť vyhľadávania pomocou regulárnych výrazov nájdeme v skoro všetkých textových editoroch, najmä na unix-ových platformách (napríklad emacs, vi, ale aj Microsoft Word).

3.2 Nahrádzanie vzorky

Regulárne výrazy sa môžu používiť aj na vykonávanie jednoduchých transformácii na veľkom množstve textu. Často sa používa príkaz 's' programu *sed*[2]. Sed číta po riadkoch vstupný súbor (najčastejšie štandardný vstup) a na každom riadku vykonáva transformáciu popísanú skriptom. Nasledovným príkazom môžeme zmeniť všetky riadky obsahujúce reťazce 'foo' a 'bar' tak, že prepíšeme 'foo' na 'bar' a obrátene:

```
$ sed -re 's/foo(.*)bar/bar\1foo/'
```

Reťazec medzi prvým a druhým znakom '`/`' je regulárny výraz, ktorý sme už videli v prvom príklade. Keď sed najde reťazec ktorý patrí do jazyka definovaného regulárnym výrazom tak ho nahradí reťazcom medzi druhým a tretím znakom '`/`' v príkaze. Tu vidíme, že zátvorky v regulárnych výrazoch majú aj iný význam okrem určovania poradia aplikovania operátorov. Oni slúžia aj na označenie textu, ktorý sa potom vloží na miesto špeciálneho reťazca '`\1`'. Tiež treba spomenúť, že pri nahrádzaní je už dôležité ako funguje '`*`' a iné operátory iterovania. Keď sa pýtame iba na príslušnosť do jazyka tak nás netrápi to, že reťazec '`foofoobar`' obsahuje dve slová z $L(\ulcorner$`foo(.*)bar`$\urcorner)$ ako podreťazce ('`foofoobar`' a '`foobar`'). Pri nahrádzaní to môže byť veľmi dôležité. Väčšina implementácii si z viacerých možných podreťezcov vyberá ten, ktorý začína najľavejšie a ak je aj viac takých tak najdlhší z nich. Výsledkom substitúcie je teda '`barfoofoo`' a nie '`foobarfoo`'.

[2]Názov pochádza zo slov „Stream EDitor“

3.3 Validácia vstupu

Pomocou regulárnych výrazov môžeme ľahko overiť, či je vstup programu v požadovanom tvare. Ak áno, tak ich môžeme použiť aj na rozloženie vstupu na komponenty. Nasledovný program v *perl*-e overí, či je vstup korektný čas a ak je tak ho vypíše po zložkách.

```
if(<> =~ /^([01]?[0-9]|2[0-3]):([0-5]?[0-9]):([0-5]?[0-9])$/) {
  printf "hodiny:%02d\nminuty:%02d\nsekundy:%02d\n", $1, $2, $3
} else {
  print "Nespravny vstup\n"
}
```

Perl umožňuje odkazovať sa na text „zachytený“ zátvorkami pomocou špecialnych premenných $\$n$, a preto prvý riadok slúži nielen na validáciu vstupu, ale aj na jeho „rozparcelovanie“. V druhom riadku potom už len vypíšeme najdené hodnoty.

3.4 *Neregulárne* regulárne výrazy

V snahe zväčšiť silu regulárnych výrazov boli do nich pridané spätné odkazy (angl. *backreferences*). V regulárnom výraze `⌜(a*)b\1b\1⌟` reťazec '`\1`' hovorí, že na jeho mieste musí byť úplne rovnaký text ako ten v prvej zátvorke. Tento výraz preto definuje jazyk $\{a^n b a^n b a^n \mid n \in \mathbb{N}\}$, čo nie je ani bezkontextový jazyk.[3] Takéto rozšírenie má aj svoju cenu. Na vyhľadávanie takýchto „regulárnych“ výrazov sa už nemôžu použiť deterministické konečné automaty, čím sa zvyšuje (časová a priestová) zložitosť tejto operácie.

3.5 Varianty regulárnych výrazov

Regulárne výrazy sú reťazce textu, ktoré opisujú iné reťazce textu. Problém nastáva, keď chceme regulárnym výrazom popísať text, ktorý obsahuje metasymboly jazyka regulárnych výrazov (`*`, `+`, `{`, `}`, ...). Rôzne implementácie toto riešia na rôzne spôsoby. Najčastie sa to rieši tak, že sa zavedie ešte jeden

[3]Toto však neznamená, že takéto výrazy môžu definovať všetky bezkontextové jazyky. Napríklad jazyk $\{a^n b^n \mid n \in \mathbb{N}\}$ sa nedá definovať ani pomocou spätných odkazov.

metasymbol (‘\’). Keď sa v regulárnom výraze tento znak zjaví pred nejakým metasymbolom (teda aj pred ‘\’), tak ten metasymbol stráca špeciálny význam. Niektoré implementácie postupujú opačne a v nich majú symboly špeciálny význam, len ak je pred nimi ‘\’. Tak napríklad v programe grep regulárne výrazy ⌜`a+`⌟ a ⌜`a\+`⌟ definujú jazyky {`a+`} a $\{\ \mathtt{a}^n \mid n \in \mathbb{N}^+\}$ (v tom poradí), kým v programe egrep je to naopak. Navyše v oboch programoch aj ⌜`+`⌟ aj ⌜`\+`⌟ definujú {`+`}, lebo aj keď je ‘`+`’ resp. ‘`\+`’ metasymbol, nemá sa na čo aplikovať a preto ho programi berú doslovne. Takéto správanie sa vyžaduje v štandarde POSIX [3].

Jednotlivé implementácie sa tiež rozlišujú v správaní ⌜`.`⌟. V niektorých to definuje naozaj celé Σ, v iných $\Sigma \setminus \{'\backslash n'\}$ (znak pre koniec riadku) a v iných zase $\Sigma \setminus \{\text{ASCII NUL}\}$ (znak pre koniec reťazca v C a iných jazykoch). Tieto a mnohé iné rozdiely poukazujú na potrebu oboznámiť sa z použitým variantom regulárnych výrazov (buď čítaním dokumentácie alebo experimentovaním) pri prvom stretnutí z nejakým programom.

Kapitola 4

Algoritmus

Výhodou regulárnych výrazov je, že sa pomocou nich ľahko popisujú niektoré jazyky. Výhodou (deterministických) konečných automatov je, že sa ľahko testuje príslušnosť slov do nimi definovaných jazykov. Preto sa v tejto kapitole budeme zaoberať algoritmami na konštrukciu konečných automatov k daným regulárnym výrazom. Aj keď konečným cieľom je zostrojenie DKA, postupujeme vo fázach tak, že najprv zostrojíme NKA a následným aplikovaním štandardnej konštrukcie dostaneme žiadaný DKA.

4.1 Regulárny výraz → NKA

Jedným riešením je zostrojiť NKA z epsilonovými prechodmi. Jeho výhodou je, že sa ľahko popisuje, implementuje a dokazuje jeho korektnosť. Avšak, pre ďaľšie spracovanie (preklad na DKA) je výhodnejšie mať automat, ktorý neobsahuje ε-prechody. Tu popíšeme algoritmus, ktorý zostrojí bezepsilonový NKA akceptujúci jazyk definovaný regulárnym výrazom. Predpokladáme, že sú regulárne výrazy definované nad abecedou $\Sigma_8 = \{0, 1, \ldots, 255\}$ reprezentujúcou všetky znaky (nejakej) 8-bitovej znakovej sady na počítači.

Algoritmus je rekurzívny a automat postupne vyrába pri prehľadávaní do hĺbky stromu odvodenia regulárneho výrazu.[1] V listoch vyrába časti automatu, ktoré sa potom v nelistových vrcholoch skladajú do väčších celkov.

Štruktúra zostrojená v každom vrchole pozostáva nasledovných častí:

K množina stavov automatu

[1] Bezkontextovú gramatiku popisujúcu syntax regulárnych výrazov neuvádzame explicitne, ale veríme že jej štruktúra je zrejmá z definícii v odseku 2.1

F množina akceptačných stavov automatu

δ prechodová funkcia automatu

X množina dvojíc $(a, q) \in \Sigma \times K$ reprezentujúca šípy vedúce do automatu

Y boolovská premenná

Štruktúry sú konštruované tak, aby v každom vrchole stromu odvodenia platil nasledovný invariant:

Tvrdenie 4.1.1. *Nech α je vrchol stromu odvodenia a nech w je regulárny výraz utvorený z listov stromu pod α. Nech S je štruktúra utvorená v α. Nech A je NKA, pričom:*

- $K_A = K_S \cup \{q_0\}$ *kde q_0 je nový stav*
- $\Sigma_A = \Sigma_8$
- $\delta_A(q, a) = \delta_S(q, a)$ *pre* $q \in K_S$
  $\delta_A(q_0, a) = \{p \mid (a, p) \in X_S\}$
- $F_A = F_S$

Potom pre každé $u \in {\Sigma_8}^+$ platí: $u \in L_w \iff u \in L_A$. Navyše, $\varepsilon \in L_w \iff Y_S = true$.

Teraz popíšeme postup zostrojovania štruktúr pre jednotlivé typy vrcholov stromu odvodenia. Výsledný NKA zostrojíme zo štruktúry prislúchajúcej celému regulárnemu výrazu podľa návodu v predchádzajúcom tvrdení.

$R = \ulcorner V_1 \mid V_2 \urcorner$ Nech S_1, S_2 sú štruktúry prislúchajúce k vetvám V_1 resp. V_2. Štruktúra S prislúchajúca k regulárnemu výrazu R bude:

- $K_S = K_{S_1} \cup K_{S_2}$
- $F_S = F_{S_1} \cup F_{S_2}$
- $\delta_S(q, a) = \begin{cases} \delta_{S_1}(q, a) & q \in K_{S_1} \\ \delta_{S_2}(q, a) & q \in K_{S_2} \end{cases}$
- $X_S = X_{S_1} \cup X_{S_2}$
- $Y_S = Y_{S_1} \vee Y_{S_2}$

$V =\ulcorner I_1 I_2 \urcorner$ Nech S_1, S_2 sú štruktúry prislúchajúce k iterovaným atómom I_1 resp. I_2. Štruktúra S prislúchajúca k vetve V bude:

- $K_S = K_{S_1} \cup K_{S_2}$
- $F_S = \begin{cases} F_{S_1} \cup F_{S_2} & Y_{S_2} = true \\ F_{S_2} & Y_{S_2} = false \end{cases}$
- $\delta_S(q,a) = \begin{cases} \delta_{S_1}(q,a) & q \in K_{S_1} \setminus F_{S_1} \\ \delta_{S_1}(q,a) \cup \{p \mid (a,p) \in X_{S_2}\} & q \in F_{S_1} \\ \delta_{S_2}(q,a) & q \in K_{S_2} \end{cases}$
- $X_S = \begin{cases} X_{S_1} \cup X_{S_2} & Y_{S_1} = true \\ X_{S_1} & Y_{S_1} = false \end{cases}$
- $Y_S = Y_{S_1} \wedge Y_{S_2}$

$I_1 =\ulcorner A\texttt{*}\urcorner, I_2 =\ulcorner A\texttt{+}\urcorner, I_3 =\ulcorner A\texttt{?}\urcorner$ Nech S je štruktúra prischajúce k atómu A. Štruktúry S_1, S_2, S_3 prislúchajúca k iterovaným atómom I_1, I_2 resp. I_3 budú:

- $K_{S_1} = K_{S_2} = K_{S_3} = K_S$
- $F_{S_1} = F_{S_2} = F_{S_3} = F_S$
- $\delta_{S_1}(q,a) = \delta_{S_2}(q,a) = \begin{cases} \delta_S(q,a) & q \in K_S \setminus F_S \\ \delta_S(q,a) \cup \{p \mid (a,p) \in X_S\} & q \in F_S \end{cases}$
  $\delta_{S_3}(q,a) = \delta_S(q,a) \qquad \forall q \in K_S \quad \forall a \in \Sigma_8$
- $X_{S_1} = X_{S_2} = X_{S_3} = X_S$
- $Y_{S_1} = Y_{S_3} = true$
  $Y_{S_2} = Y_S$

$I_1 =\ulcorner A\texttt{\{}n\texttt{\}}\urcorner, I_2 =\ulcorner A\texttt{\{}n\texttt{,}m\texttt{\}}\urcorner, I_3 =\ulcorner A\texttt{\{}n\texttt{,\}}\urcorner$ Iterovaný atóm I_1 je skráteným zápisom ekvivalentného regulárneho výrazu $\ulcorner \underbrace{AA\cdots A}_{n} \urcorner$, a teda by sme štruktúru preňho (označme ju S) mohli vyrobiť tak, že urobíme n kópii štruktúry prislúchajúcej k A (ozn. $S_1, S_2, \ldots S_n$) a tie potom spojíme podľa návodu na zreťazenie. Avšak v prípade, že $\varepsilon \in L_A$, tento postup vyrobí štruktúru, ktorej graf bude mať $\Omega(n^2)$ šipiek. Tie šipky nám umožňujú prejsť cez ľubovoľnú podmnožinu n kópii automatu. Ale kópie sú ekvivalentné a nás preto zaujíma len ich počet. Môžeme ich preto pospájať tak, aby z S_i viedli šípky len do S_{i+1}, pričom akceptačne stavy S budú akceptačne stavy S_n a $X_S = \bigcup_{i=1}^{n} X_{S_i}$. Takto

výrazne zmenšíme počet šipiek vo výslednom automate, čím zmenšíme pamäť potrebnú na uloženie automatu a zrýchlime proces konštrukcie DKA. Na obrázkoch 4.1 a 4.2 sú znázornené (ekvivalentné) automaty vyrobené z regulárneho výrazu ⌜`(a?){4}b`⌟ na obidva spôsoby.

Podobný postup použijeme pre iterované atómy I_2 a I_3.

$A_1 = ⌜T⌟, A_2 = ⌜b⌟, A_3 = ⌜.⌟$ Nech T je trieda znakov, nech $b \in \Sigma_8$. Štruktúry S_1, S_2, S_3 prislúchajúce k atómom A_1, A_2 resp. A_3 sú:

- $K_{S_1} = K_{S_2} = K_{S_3} = \{p\}$ kde p je nový stav
- $F_{S_1} = F_{S_2} = F_{S_3} = \{p\}$
- $\delta_{S_1}(p, a) = \delta_{S_2}(p, a) = \delta_{S_3}(p, a) = \emptyset \qquad \forall a \in \Sigma_8$
- $X_{S_1} = \{(a, p) \mid a \in L_T\}$
  $X_{S_2} = \{(b, p)\}$
  $X_{S_3} = \{(a, p) \mid a \in \Sigma_8\}$
- $Y_{S_1} = Y_{S_2} = Y_{S_3} = false$

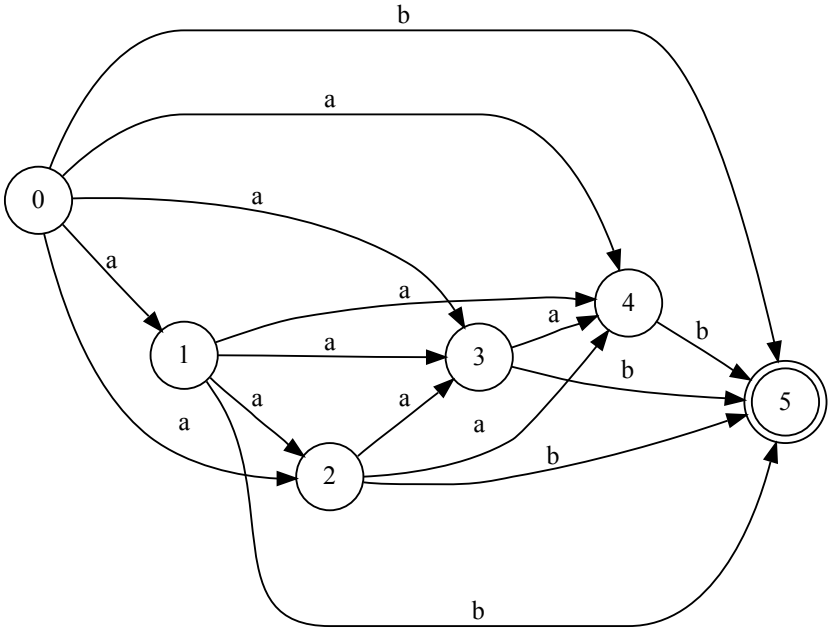
Poznámka 4.1.1. *V hore uvedenom algoritme sme uvažovali iba prípad keď regulárny výraz pozostáva z dvoch vetiev aj keď on ich v skutočnosti môže mať ľubovoľne veľa. Operácia alternatívy je však asociatívna a preto si môžeme regulárny výraz „v hlave“ uzátvorkovať zľava a tak ho spracovať. To isté platí pre vetvy pozostávajúce z viac ako dvoch iterovaných atómov.*

Poznámka 4.1.2. *V prípadoch $R = ⌜V_1⌟, V = ⌜I_1⌟, I = ⌜A_1⌟, A = ⌜(R_1)⌟$ sú štruktúry prislúchajúce objektom na ľavej strane rovnosti rovnaké ako tie prislúchajúce objektom na pravej strane.*

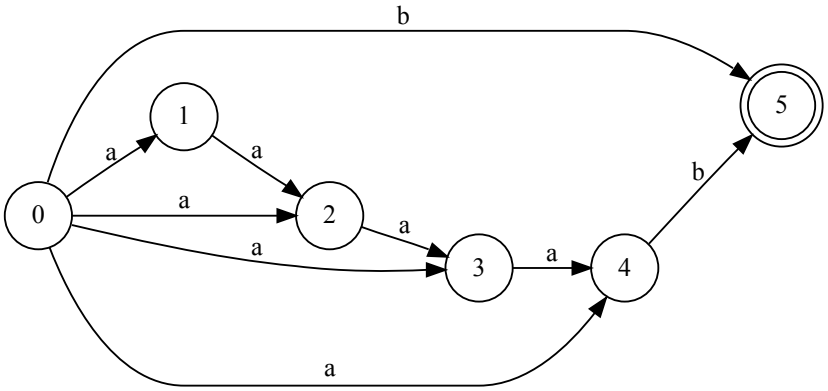
4.2 NKA → DKA

Nech A je NKA. Ekvivalentný DKA A' zostrojíme štandardnou konštrukciou. Tu uvádzame iba formálnu definíciu A', pre dôkaz správnosti a ďalšie vlastnosti odkazujeme čitateľa na [7].

- $K_{A'} = 2^{K_A}$
- $\Sigma_{A'} = \Sigma_A (= \Sigma_8)$



Obr. 4.1: Automat vyrobený naivným algoritmom



Obr. 4.2: Automat vyrobený vylepšeným algoritmom

- $q'_0 = \{q_0\}$
- $F_{A'} = \{X \in 2^{K_A} \mid X \cap F_A \neq \emptyset\}$
- $\delta_{A'}(Q, a) = \bigcup_{q \in Q} \delta_A(q, a) \qquad \forall Q \in K_{A'} \quad \forall a \in \Sigma_{A'}$

Poznámka 4.2.1. *V skutočnosti nie je potrebné konštruovať všetkých 2^{K_A} stavov, ale len tie ktoré sú dosiahnuteľné zo začiatočného stavu. Takáto optimalizácia v niektorých prípadoch výrazne zlepší časovú a priestorovú zložitosť algoritmu, lebo mnohé regulárne výrazy nepotrebujú exponenciálne veľa stavov.*

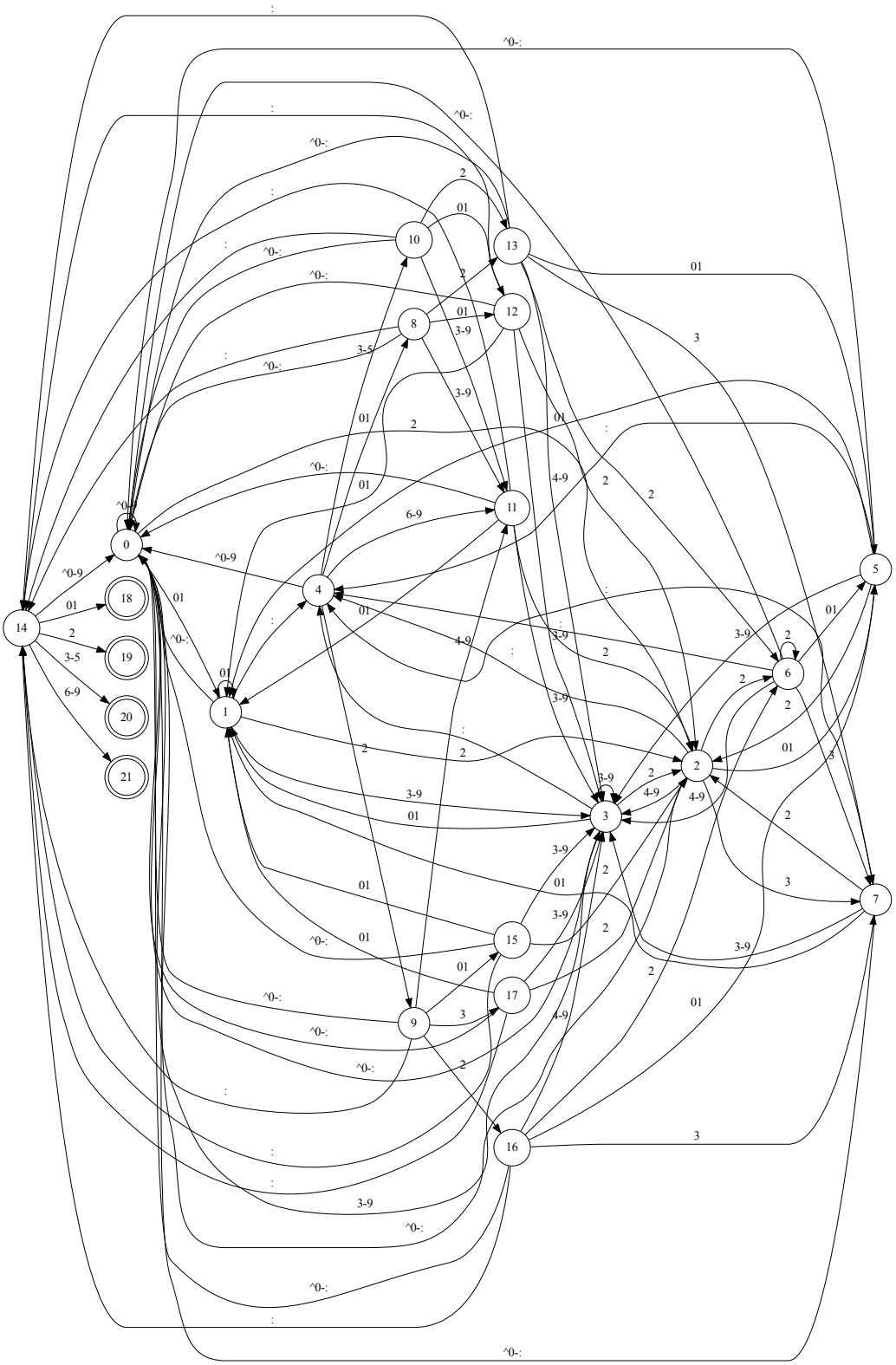
4.3 Hľadanie podreťazcov

Hore uvedený algoritmus vyrobí automat, ktorý rozpoznáva práve všetky slová z L_R (R je regulárny výraz). Grep (a mnohé iné programy) potrebuje vedieť či vstup *obsahuje* nejaké slovo z L_R ako podreťazec.[2] Jedno riešenie je zostrojiť automat, ktorý bude rozpoznávať slová definované výrazom $\ulcorner .*R.*\lrcorner$. Takýto automat vždy dočíta celý vstup, čo nemusí byť potrebné, lebo akonáhle nájdeme prvý výskyt slova z L_R môžeme vyhľadávanie prehlásiť za úspešné a skončiť. Preto je lepšie zmeniť algoritmus tak, aby tieto okolnosti zohľadňoval:

- NKA zostrojíme z R rovnako ako v odesku 4.1, ale nakoniec do začiatočného stavu pridáme slučky pre každé pismeno z abecedy.
- Konštrukciu DKA končíme v akceptačných stavoch.
- Rozponávanie končíme akonáhle sa dostaneme do akceptačného stavu.

Takto dosiahneme rýchlejšie rozpoznávanie a trochu aj zmenšíme pamäťové a časové náklady na konštrukciu DKA, keďže neuvažujeme prechody vychádzajúce z akceptačných stavov. Na obrázku 4.3 uvádzame takto zostrojený automat z výrazu $\ulcorner$`([01]?[0-9]|2[0-3]):([0-5]?[0-9]):([0-5]?[0-9])`$\lrcorner$.

[2] teda vlastne, či vstup patrí do jazyka $\Sigma^* L_R \Sigma^*$



Obr. 4.3: DKA pre zjednodušený zápis času

4.4 O zložitosti

Prvá časť algoritmu môže vyrobiť NKA s exponenciálne veľa stavmi ak výraz obsahuje operáciu iterovania pomocou '`{}`'. Tak napríklad ľubovoľný NKA rozpoznávajúci (jednoprvkový) jazyk definovaný regulárnym výrazom ⌜`a{`n`}`⌟ musí mať aspoň $n+1$ stavov. Výrazy neobsahujúce '`{}`' takýto problém nemajú, keďže jediná operácia, ktorá vytvára nové stavy, je spracovanie jednoduchého atómu[3] a výraz dĺžky n môže mať najviac n atómov.

Pri konštrukcii DKA môže nastať prípad, keď DKA bude mať exponenciálne veľa stavov vzhľadom na NKA, z ktorého je zostrojený. Napríklad, pre každé prirodzené n, náš algoritmus z regulárneho výrazu ⌜`.*a.{`n`}a`⌟ zostrojí $(n+3)$-stavový automat,[4] ale DKA pre daný jazyk bude mať $\Omega(2^n)$ stavov.

Tvrdenie 4.4.1. *Pre každé $n \in \mathbb{N}$ platí: ľubovoľný deterministický konečný automat rozpoznávajúci jazyk $L_n = L(\ulcorner$`.*a.{`n`}a`$\urcorner)$ má aspoň 2^n stavov.*

Dôkaz. Použijeme Myhill-Nerodovu vetu[8][9]. Nech $n \in \mathbb{N}$. Nech $\sim$ je sprava invariantná relácia ekvivalencie vzhľadom na operáciu zreťazenia definovaná nasledovne:

$$u \sim v \quad \overset{def}{\iff} \quad (\forall x \in {\Sigma_8}^* \quad ux \in L_n \Leftrightarrow vx \in L_n)$$

Ukážeme, že žiadne dve slová z množiny $X = \{a, b\}^n$ nemôžu byť v rovnakej triede ekvivalencie relácie $\sim$. Nech $u, v \in X$, $u \neq v$. Nech i je index prvého znaku, v ktorom sa u a v rozlišujú. i-ty znak v u je a alebo b. Bez ujmy na všeobecnosti predpokladajme, že je to a. Potom i-ty znak v v je b. Nech $x = b^{n-i}a$. Potom $ux \in L_n$, ale $vx \notin L_n$, čo znamená že neplatí $u \sim v$ a podľa Myhill-Nerodovej vety automat musí mať aspoň $|X|$ stavov. □

4.5 Rozpoznávanie pomocou NKA

V situáciach, keď je DKA príliš veľký na to, aby sa mohol celý uložiť do pamäti môže sa pristúpiť k priamej simulácii NKA: po každom načítaní znaku zo vstupu zostrojíme množinu stavov, v ktorých NKA môže byť postupným prechodom cez všetky stavy v množine utvorenej v predchádzajúcom kroku.

[3] Teda nie takého, ktorý vznikol uzátvorkovaním regulárneho výrazu

[4] existuje však aj ekvivalentný automat s $n+2$ stavmi

Tým sa vyhneme exponenciálnej priestorovej zložitosti, ale odhad časovej zložitosti sa zhorší na $O(nm)$, kde n je dĺžka vstupného slova a m — počet stavov NKA. V najhoršom prípade sa totiž môže stať, že všetky stavy budú aktívne.

GNU grep robí kompromis. Na vyhľadávanie používa DKA, ale stavy vyrába lenivo — len keď sa automat naozaj do stavu dostane. Navyše, ak počet stavov prekročí istú hranicu, tak tie najstaršie zruší a pokračuje ďalej. Ak neskoršie tie zrušené stavy bude potrebuvať, tak ich znovu vyrobí.[10] Tým dosiahne to, že pre jednoduché (a najčastejšie) regulárne výrazy sa bude správať ako DKA a v patologických prípadoch nebude potrebovať exponenciálne veľkú pamäť.

Pre špeciálne triedy regulárnych výrazov je najsť aj lepšie riešenia. Systémy na detekciu prienikov (angl. *Intrusion Detection System — IDS*) používajú aj regulárne výrazy na na odhalenie pokusov o útok. Pritom ale nepoužívajú ich plnú výrazovú silu. V [11] autori navrhujú hybridný konečný automat, ktorý vznikne kombináciou NKA a DKA. Automat sa vyrába z NKA, podobne ako DKA, ale tie stavy NKA, ktoré by spôsobili výrazné zvýšenie počtu stavov, zostávajú nedeterministické. Pre výrazy obsahujúce `.{n,m}` sa použije špeciálny stav s počítadlom. Pre takýto automat sa potom dá odhadnúť počet súčasne aktívnych stavov, čo je veľmi dôležité pre použitie v IDS.

Kapitola 5

Testovanie

Implementáciu algoritmu opísaného v kapitole 4 sme porovnali s GNU grepom. Skúmali sme dva aspekty ich práce:

1. čas potrebný na spracovanie regulárneho výrazu
2. čas potrebný na samotné vyhľadávanie

Testy sme spúšťali na počítači s procesorom Intel® Core™2 Duo E4500 (2.20GHz, 2MB L2 cache) a 1GB pamäte. Skripty, pomocou ktorých sme testovali, uvádzame v dodatku A.

5.1 Spracovanie regulárneho výrazu

5.1.1 Čistý text

Regulárny výraz pozostával z náhodne vygenerovaného reťazca nad abecedou $\Sigma_T =$ `[A-Za-z0-9]` (malé a veľké pismená anglickej abecedy a číslice). Skúmali sme ako čas spracovania závisí od dĺžky reťazca keď výraz neobsahuje žiadne metasymboly jazyka regulárnych výrazov. Merali sme čas spracovania reťazcov dĺžky 100–10000 znakov algoritmom z kapitoly 4 (foogrep) a programom GNU grep (egrep). Pre porovnanie, sme zmerali aj čas grep-u spusteného s parametrom '-f', ktorý prepne grep do špeciálneho režimu pre hľadanie jednoduchých podreťazcov (fgrep). Časť nameraných údajov je uvedená v tabuľke 5.1 a graf je znázornený na obrázku 5.1. Je vidno, že egrep potrebuje kvadraticky veľa času na rozdiel od foogrep-u, ktorého čas je lineárny a porovnateľný so špecializovaným fgrep-om.

5.1.2 Alternatívy

Regulárny výraz sa skladal s 1–100 vetiev, a každá vetva pozostávala zo 100 náhodných znakov z abecedy Σ_T. Merali sme ako závisí čas spracovania výrazu programami foogrep a egrep od počtu alternatív. Na základe tabuľky 5.2 a obrázku 5.2 usudzujeme, že v oboch prípadoch je tá závislosť kvadratická, ale egrep je citeľne rýchlejší. To sa dá vysvetliť lenivou konštrukciou DKA stavov (viď odsek 4.5).

5.1.3 Iterácia pomocou '{}'

Merali sme čas spracovania regulárneho výrazu ⌜a.{n}a␣ pre rôzne n. Zistili sme, že egrep je neporovnateľne rýchlejší vďaka lenivej konštrukcii DKA. Už pre $n = 17$ foogrep potrebuje 12.9 sekund, kým egrep zvláda aj $n = 1000$ za 5 milisekund. Namerané hodnoty sú v tabuľke 5.3 a na obrázku 5.3.

5.2 Vyhľadávanie v texte

Merali sme čas potrebný na nájdenie všetkých výskytov regulárneho výrazu v súboroch. Výraz pozostával s 1–3 vetiev a každá vetva bola náhodný reťazec nad abecedou Σ_T dĺžky 50 ± 10 znakov. Súbory mali 10000 riadkov dĺžky 100 ± 10 znakov a odlišovali sa pomerom

a) riadkov obsahujúcich hľadané vzorky

b) nahodných riadkov, ktoré hľadané vzorky neobsahujú

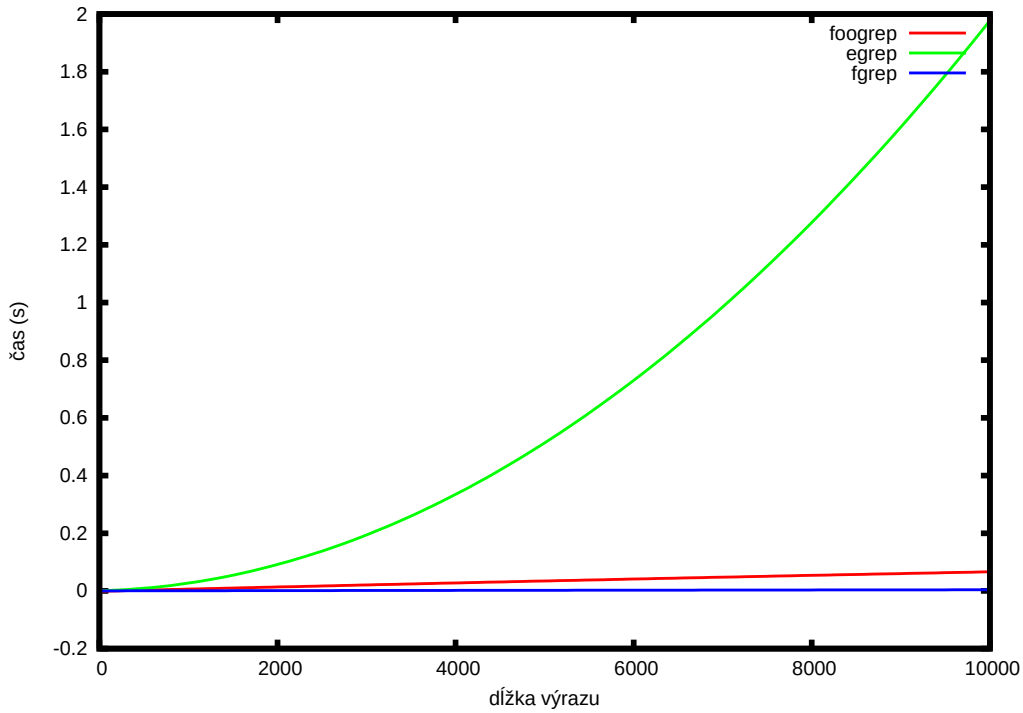
Výsledky egrep-u odhaľujú výraznú závislosť času od pomeru dvoch typov riadkov, kým časy foogrep-u sú rovnaké pre všetky pomery. Čas spracovania súboru s riadkami typu b egrep-om je porovnateľné s foogrep-om, ale súbor so 100% riadkov typu a egrep spracovával 83 sekund, kým foogrep to zvládol za 225 milisekund. Výsledky pre testy s rôznym počtom vetiev sa veľmi neodlišovali. V tabuľke 5.4 uvádzame výsledky meraní vyhľadávania regulárneho výrazu s jednou a troma vetvami na súboroch obsahujúcich 0%, 10%, 50% resp. 100% riadkov typu a. Pre prípady s jednou vetvou sme skúsili aj vyhľadávanie s fgrep-om, ale výsledky boli totožné s egrep-om. Tiež sme skúsili zmenšiť dĺžku vetiev na 5 znakov, ale ani to nezmenšilo čas spracovania súborov, ktoré obsabovali riadky typu a.

dĺžka reťazca	čas spracovania (s)		
	foogrep	egrep	fgrep
100	0.001	0.001	0.000
200	0.001	0.001	0.000
300	0.002	0.003	0.001
400	0.003	0.005	0.000
500	0.003	0.008	0.001
600	0.003	0.009	0.000
700	0.005	0.013	0.000
800	0.006	0.016	0.001
900	0.006	0.019	0.001
1000	0.006	0.023	0.001
1100	0.007	0.027	0.001
1200	0.007	0.033	0.000
1300	0.008	0.037	0.001
1400	0.009	0.043	0.001
1500	0.009	0.048	0.001
1600	0.010	0.054	0.001
1700	0.010	0.061	0.001
1800	0.009	0.068	0.001
1900	0.012	0.077	0.001
2000	0.011	0.084	0.002
⋮	⋮	⋮	⋮
9000	0.059	1.562	0.003
9100	0.058	1.612	0.003
9200	0.059	1.654	0.005
9300	0.059	1.681	0.004
9400	0.062	1.716	0.003
9500	0.062	1.736	0.004
9600	0.062	1.797	0.004
9700	0.061	1.831	0.004
9800	0.065	1.857	0.004
9900	0.065	1.906	0.004
10000	0.066	1.930	0.004

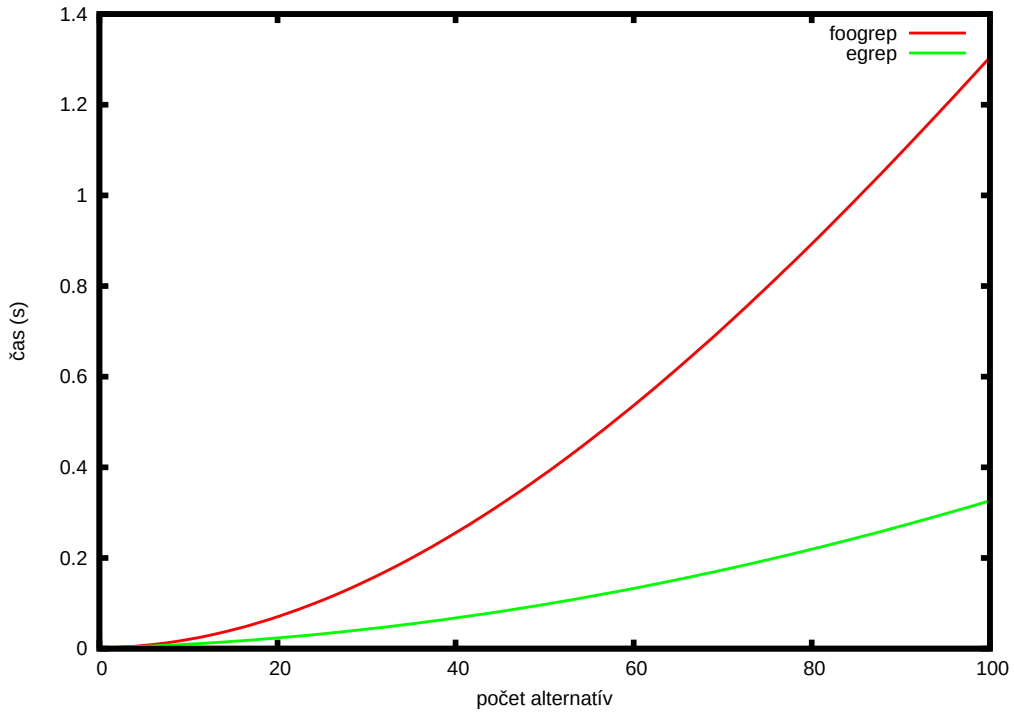
Tabuľka 5.1: Čas spracovania čistého textu

počet alternatív	čas spracovania (s)	
	foogrep	egrep
1	0.001	0.001
2	0.002	0.002
3	0.003	0.003
4	0.004	0.004
5	0.006	0.004
6	0.007	0.005
7	0.010	0.006
8	0.012	0.007
9	0.014	0.008
10	0.017	0.009
11	0.020	0.010
12	0.024	0.012
13	0.029	0.012
14	0.033	0.013
15	0.035	0.015
16	0.042	0.016
17	0.044	0.017
18	0.052	0.020
19	0.058	0.021
20	0.065	0.022
⋮	⋮	⋮
90	1.072	0.284
91	1.084	0.277
92	1.128	0.284
93	1.133	0.286
94	1.161	0.295
95	1.180	0.302
96	1.194	0.316
97	1.206	0.303
98	1.256	0.318
99	1.237	0.314
100	1.256	0.329

Tabuľka 5.2: Čas spracovania alternatív



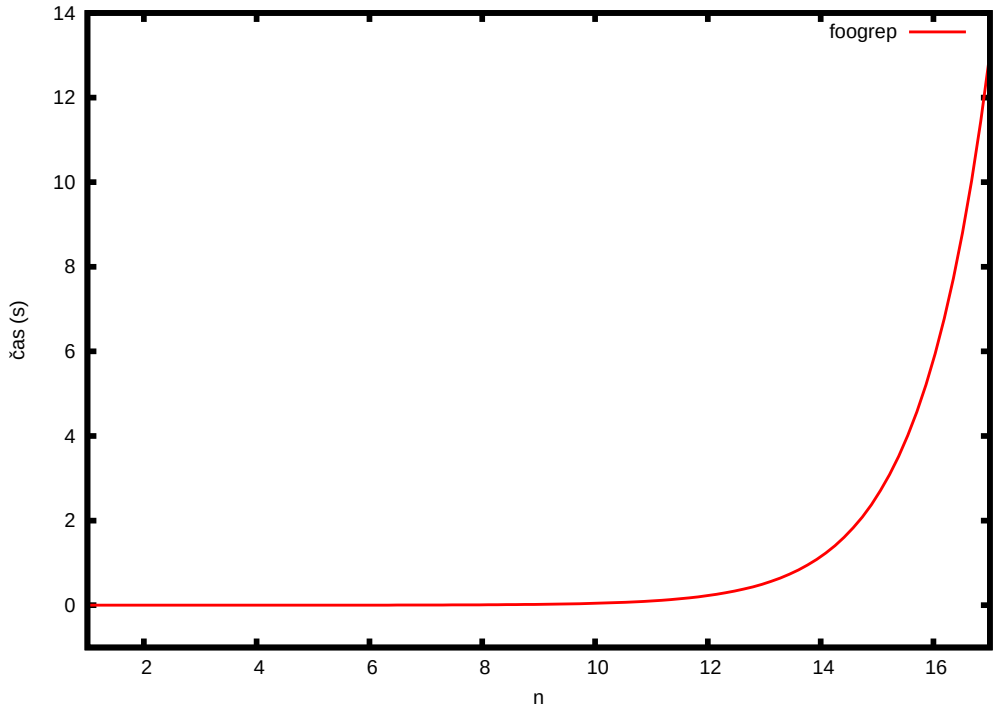
Obr. 5.1: Čas spracovania čistého textu



Obr. 5.2: Čas spracovania alternatív

n	1	2	3	4	5	6	7	8	9
čas (s)	0.001	0.001	0.001	0.001	0.002	0.002	0.004	0.009	0.018
n	10	11	12	13	14	15	16	17	
čas (s)	0.041	0.103	0.222	0.508	1.140	2.580	6.020	12.854	

Tabuľka 5.3: Čas spracovania `a.{n}a` programom foogrep



Obr. 5.3: Čas spracovania `a.{n}a` programom foogrep

počet vetiev	program	% vyhovujúcich riadkov			
		0%	10%	50%	100%
1	foogrep	0.222	0.222	0.221	0.225
1	egrep	0.510	9.000	41.379	83.283
3	foogrep	0.332	0.342	0.356	0.379
3	egrep	1.057	9.197	44.697	85.826

Tabuľka 5.4: Vzťah času vyhľadávania a počtu riadkov obsahujúcich hľadané reťazce

Kapitola 6

Záver

V tejto práci sme uviedli a otestovali algoritmus na rozpoznávanie regulárnych výrazov. Jeho najväčšia výhoda je lineárna časová zložitosť. Má to však aj svoju cenu — exponenciálna časová a priestorová zložitosť predspracovania regulárneho výrazu v najhoršom prípade. Samotné rozpoznávanie je rýchle, ale drahé predspracovanie znemožňuje použitie tekéhoto postupu pre niektoré triedy regulárnych výrazov. Tiež treba spomenúť, že tento algoritmus nedokáže spracovať „regulárne“ výrazy obsahujúce spätné odkazy, negatívny kontext a pod. ako ani zvládnuť situácie, kde sa vyžaduje „zachytávanie“ textu zátvorkami. V týchto prípadoch sa potom používa backtrackovací algoritmus, ale aj on má svoje nedostatky — exponenciálna časová zložitosť pri rozpoznávaní výrazov typu `⌜a(.*)*a⌟`, najmä v prípadoch keď text neobsahuje hľadanú vzorku.

Keďže neexistuje univerzálne najlepší algoritmus, treba pri implementovaní zvoliť ten, ktorý sa v danej situácii bude správať najlepšie. Alebo aj opačne — ak vieme aký algoritmus sa používa na rozpoznávanie, môžeme sa pokúsiť napísať regulárny výraz tak, aby proces rozpoznávania skončil čo najrýchlejšie.

Literatúra

[1] S. C. Kleene, "Representation of events in nerve nets and finite automata," in *Automata Studies* (C. E. Shannon and J. McCarthy, eds.), pp. 3–42, Princeton, N.J.: Princeton University Press, 1956.

[2] L. J. Stockmeyer and A. R. Meyer, "Word problems requiring exponential time(preliminary report)," in *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 1–9, ACM, 1973.

[3] IEEE, *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. New York, NY, USA: IEEE, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[4] P. Weiner, "Linear pattern matching algorithm," in *14th Annual IEEE Symposium on Switching and Automata Theory*, pp. 1–11, 1973.

[5] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, pp. 323–350, June 1977.

[6] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, pp. 762–772, Oct. 1977.

[7] J. E. Hopcroft and J. D. Ullman, *Formal languages and their relation to automata*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1969.

[8] J. Myhill, "Finite automata and the representation of events," Tech. Rep. WADD TR-57-624, Wright Patterson AFB, Ohio, Nov. 1957.

[9] A. Nerode, "Linear automaton transformations," in *Proceedings of the American Mathematical Society*, vol. 9, pp. 541–544, 1958.

[10] http://www.gnu.org/software/grep/.

[11] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *CoNEXT '07: Proceedings of the 2007 ACM CoNEXT conference*, (New York, NY, USA), pp. 1–12, ACM, 2007.

Dodatok A

Testovacie skripty

V tomto dodatku uvádzame listingy skriptov, ktoré sme použili na testovanie algoritmu z kapitoly 4. Opis testov, ako aj ich výsledky, sú uvedené v kapitole 5.

Najdôležitejší skript je `regex-gen.pl` (listing A.1), ktorý generuje náhodné regulárne výrazy a súbory pre testy. Testy z odseku 5.1 sa spúšťajú pomocou skriptu `runtest.sh` (listing A.2), kým `runmatch.sh` (listing A.3) spustí test z odseku 5.2. Tieto dva skripty si pomocou `regex-gen.pl` vygenerujú dáta a potom opakovane spúšťajú foogrep, egrep a fgrep a zaznamenávajú ich časy. Pri práci používajú ešte dva skripty:

- `process-time.pl` (listing A.4), skript spracujúci výstup príkazu *time*.
- `merge-results.pl` (listing A.5), skript, ktorý z nameraných hodnôt vypočíta strednú hodnotu a odchýlku.

```
#!/usr/bin/perl
use strict;
use warnings;

my $chars = '0123456789qwertyuiopasdfghjklzxcvbnm' .
            'QWERTYUIOASDFGHJKLZXCVBNM';
sub randchar {
  return substr($chars, int(rand(length $chars)), 1);
}

sub randchars {
  my $t='';
  $t .= randchar foreach 1 .. shift;
  return $t;
```

```
}

sub plaintext {
  print randchars(100*$_), "\n" foreach 1 .. shift;
}

sub alternatives {
  foreach my $i (1 .. shift) {
    my $t=randchars(100*$i);
    print substr($t, $_*100, 100), $_!=$i-1?"|":"\n"
                foreach 0 .. $i-1;
  }
}

sub braces {
  print "a.{$_}a\n" foreach 1 .. shift;
}

# param 0: number of alternatives
# param 1: number of lines
# param 2: probability that a line will contain a match
sub matching {
  my @alt;
  foreach (1 .. $_[0]) {
    my $t=randchars(40+int(rand(20)));
    print STDERR $t, $_!=$_[0]?"|":"\n";
    push(@alt, $t);
  }

  foreach my $i (1 .. $_[1]) {
    unless(rand(1)<$_[2]) {
      print randchars(90+int(rand(20))), "\n";
    } else {
      my $t=$alt[int(rand($_[0]))];
      my $str=randchars(40+int(rand(20)));
      my $pos=int(rand(length($str)+1));
      print substr($str, 0, $pos), $t,
            substr($str, $pos, 100), "\n";
    }
  }
}

$_=shift;
plaintext(shift) if /^plaintext$/i;
alternatives(shift) if /^alternatives$/i;
braces(shift) if /^braces$/i;
matching(@ARGV) if /^matching$/i;
```

Listing A.1: regex–gen.pl

```
#!/bin/bash
# param 1: test to run
# param 2: number of regexes
./regex-gen.pl $1 $2 >testdata/regexes.lst

for((j=1; j<=5; j++)); do
  for i in src/foogrep egrep fgrep; do
    while read A; do
      time $i $A </dev/null
    done <testdata/regexes.lst 2>&1 | \
    ./process-time.pl >testdata/$1-$j-`basename $i`.plt
  done
done

for i in src/foogrep egrep fgrep; do
  ./merge-results.pl testdata/$1-*-`basename $i`.plt \
     >testdata/$1-`basename $i`.plt
done
```

Listing A.2: runtest.sh

```
#!/bin/bash
# param 1: number of alternatives
# param 2: number of lines
for((j=1; j<=5; j++)); do
  for k in 0.0 0.1 0.5 1.0; do
    A=`./regex-gen.pl matching $1 $2 $k 2>&1 >testdata/grepfile`
    for i in src/foogrep egrep; do
      time $i $A testdata/grepfile >/dev/null
    done
  done 2>&1 | ./process-time.pl >testdata/matching-$1-$j.plt
done

./merge-results.pl testdata/matching-$1-*.plt \
   >testdata/matching-$1.plt
```

Listing A.3: runmatch.sh

```
#!/usr/bin/perl
use strict;
use warnings;

while(<>) {
  print $1*60 + $2, "\n" if /user\s*([0-9]+)m([0-9]+\.[0-9]+)s/;
}
```

Listing A.4: process–time.pl

```
#!/usr/bin/perl
use strict;
use warnings;

my @files;
foreach my $i (0 .. $#ARGV) {
    open($files[$i], $ARGV[$i]) ||
        die "open failed for file $ARGV[$i]: $!.";
}

outer: for(;;) {
  my $sum=0;
  my @arr;
  foreach my $file (@files) {
    if(my $in=<$file>) {
      $sum+=$in;
      push @arr, $in;
    } else {
      last outer;
    }
  }
  my $avg=$sum/@ARGV;
  $sum=0;
  $sum+=($_-$avg)**2 foreach @arr;
  my $s=@ARGV>1?sqrt($sum/(@ARGV-1)):1;
  print $avg, " ", $s, "\n";
}
```

Listing A.5: merge–results.pl