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

Zložitostné Aspekty Konečných Automatov

(bakalárska práca)

Ivana Hudáková

Odbor: Informatika 9.2.1
Vedúci: prof. RNDr. Branislav Rovan, PhD.

Bratislava, 2007

Abstrakt

V práci prinášame prehľad najdôležitejších výsledkov z oblasti zložitosti konečných automatov. Popisujeme praxou motivované problémy a prezentujeme ich riešenia na úrovni teórie formálnych jazykov. Zaoberáme sa efektívnosťou týchto riešení, t.j. ich pamäťovou a časovou zložitosťou. Tam, kde nie je známe efektívne riešenie problému, venujeme sa riešeniam pre čiastkové úlohy.

Kľúčové slová: konečné automaty, popisná zložitosť, algoritmy

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

. . . . . .

Ďakujem vedúcemu mojej bakalárskej práce profesorovi Branislavovi Rovanovi za pomoc pri hľadaní zaujímavej témy, ako aj za neoceniteľné rady pri jej spracovávaní.

Ďalej by som sa chcela poďakovať profesorovi Jurajovi Hromkovičovi, profesorovi Viliamovi Geffertovi a doktorke Galine Jiráskovej za poskytnutie kópií ich článkov.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

Idea konečnostavového zariadenia bola prvýkrát použitá na modelovanie procesov v mozgu [19]. Koncept zariadenia s konečným počtom stavov sa rýchlo ujal a bol široko využívaný. V teórii formálnych jazykov sa ukázalo, že konečné automaty sú ekvivalentné regulárnym gramatikám a teda akceptujú práve Chomského jazyky tretieho typu.

Rabin a Scott vo svojom článku [25] priniesli viacero zovšeobecnení konečných automatov. Implementácia relatívne elementárnych operácii pomocou deterministických konečných automatov vyžadovala veľké množstvo stavov a tak Rabin a Scott pridali do automatov nedeterminizmus, aby získali zariadenie s menším počtom stavov. Zaviedli tiež iné zovšeobecnenia a určili vzťah medzi novými modelmi a klasickým modelom automatu. V roku 1976 za svoj článok získali Turingovu cenu.

Konečné automaty sa využívajú v rôznych aplikáciách, napríklad na spracovanie prirodzeného jazyka, na lexikálnu analýzu v kompilátoroch, modelovanie veľkých slovníkov a editovanie textu. V poslednej dobe pribudli aplikácie v oblasti verifikácie programov. Aby implementácia riešení pomocou konečných automatov bola (pamäťovo) efektívna, snažíme sa minimalizovať pamäť potrebnú na uloženie automatu, ktorý používame na riešenie problému.

Výpočtová zložitosť konečných automatov je daná modelom (lineárny čas, konštantný priestor). Zaoberáme sa preto popisnou zložitosťou. Najčastejšie používané miery sú počet stavov, niekedy (pri nedeterministických automatoch) aj počet prechodov.

Pre programovacie účely je najvhodnejší deterministický konečný automat, pretože naprogramovanie (simulácie) jeho správania je jednoduché. Ako ukážeme v kapitole 3, nájsť minimálny DKA pre jazyk daný nejakým DKA nie je výpočtovo náročné.

Mnohé problémy, hlavne v oblasti lexikálnej analýzy a editovania textu, však na prvý pohľad nesúvisia s DKA. Pre človeka je jednoduchšie vnímať popis niektorých problémov cez regulárne výrazy. Jednoduchý problém tohto typu je napríklad otázka: "Nachádza sa v danom texte slovo z jazyka popísaného regulárnym výrazom?". Iný problém je pýtať sa: "Spĺňa táto časť kódu syntaktické pravidlá definované týmito výrazmi?". Aby počítač mohol dať odpoveď na takéto otázky, potrebujeme k regulárnemu výrazu nájsť automat. Riešenia tejto konverzie prinášame v kapitole 4.

Je tiež veľmi zaujímavé, ako je zložitosť automatu ovplyvnená inými jeho vlastnosťami. O tom ako nedeterminizmus vplýva na počet stavov si niečo povieme v kapitole 5.

Predpokladáme, že čitateľ sa už stretol s pojmom konečného automatu v rámci teórie formálnych jazykov. Aby sme však predišli nedorozumeniam, v nasledujúcej kapitole uvádzame definície a označenia, ktoré budeme používať v celej práci.

Kapitola 2

Základné definície a označenia

Ústredným pojmom celej práce je pojem konečného automatu. Uvádzame preto jeho definíciu, ktorú možno nájsť v [8], rovnako ako aj bližšie vlastnosti konečných automatov.

2.1 DKA, NKA

Definícia 2.1.1. *Deterministický konečný automat (DKA) je pätica* $A = (K, \Sigma, \delta, q_0, F)$, *kde* K *je konečná množina stavov,* Σ *je konečná vstupná abeceda,* $q_0 \in K$ *je počiatočný stav,* $F \subseteq K$ *je množina akceptačných stavov a* $\delta : K \times \Sigma \to K$ *je prechodová funkcia.*

Definícia 2.1.2. *Konfigurácia DKA je dvojica* $(q, w) \in K \times \Sigma^*$, *kde* q *je stav automatu a* w *je nespracovaná časť vstupného slova.*

Definícia 2.1.3. *Krok výpočtu DKA* A *je relácia* $\vdash_A$ *na konfiguráciách definovaná nasledovne:*

$$(q, av) \vdash_A (p, v) \iff p = \delta(q, a).$$

Označenie 2.1.1. *Budeme často písať* $\vdash$ *namiesto* $\vdash_A$, *pokiaľ je zrejmé, o ktorý automat sa jedná.*
$\vdash^*$ *bude označovať reflexívno-tranzitívny uzáver relácie a* Σ^* *bude označovať množinu všetkých slov nad abecedou* Σ.
Ak to bude potrebné, budeme symbolom Σ_L *označovať minimálnu abecedu takú, že* $L \subseteq \Sigma_L^*$.

Definícia 2.1.4. *Jazyk akceptovaný DKA* A *je množina slov* $L(A) = \{w \in \Sigma^* \mid (q_0, w) \vdash^* (q, \varepsilon) \wedge q \in F\}$.

Definícia 2.1.5. *Nedeterministický konečný automat (NKA) je pätica* $(K, \Sigma, \delta, q_0, F)$, *kde* K *je konečná množina stavov,* Σ *je konečná vstupná abeceda,* $q_0 \in K$ *je počiatočný stav,* $F \subseteq K$ *je množina akceptačných stavov a* $\delta : K \times (\Sigma \cup \{\varepsilon\}) \to 2^K$ *je prechodová funkcia.*

Definícia 2.1.6. *Krok výpočtu NKA A je relácia $\vdash_A$ na konfiguráciách definovaná nasledovne:*

$$(q, av) \vdash_A (p, v) \iff p \in \delta(q, a).$$

Poznámka 2.1.1. *Konfiguráciu NKA a jazyk akceptovaný NKA definujeme rovnako ako pre DKA.*

Definícia 2.1.7. $L(A, q) = \{w \in \Sigma^* \mid \exists p : (q, w) \vdash_A^* (p, \varepsilon) \wedge p \in F\}$ *je jazyk prislúchajúci k stavu q. Budeme používať symbol $L(q)$, ak bude z kontextu zrejmé, o ktorý automat ide.*

Automat A si môžeme predstaviť ako graf. Keď sa pozrieme na cesty, ktoré vedú zo stavu q do akceptačných stavov a zreťazíme návestia, ktoré sa nachádzajú na jednej takejto ceste, dostaneme slovo z $L(A, q)$.

Definícia 2.1.8. *Stav $q \in K$ je dosiahnuteľný, ak existuje $w \in \Sigma^*$ také, že $(q_0, w) \vdash^* (q, \varepsilon)$.*

V popisovaných algoritmoch budeme vždy pracovať s automatmi, ktoré majú len dosiahnuteľné stavy. Nedosiahnuteľné stavy nebudú nikdy použité pri žiadnom výpočte a tak ich môžeme jednoducho odstrániť a získať tak ekvivalentný automat. Nedosiahnuteľné stavy vieme nájsť pomocou prehľadávania grafu, ktorý reprezentuje zadaný automat.

Definícia 2.1.9. *Hovoríme, že konečný automat A je úplný, ak $\forall q \in K, \forall a \in \Sigma : \delta(q, a) \neq \emptyset$.*

Táto definícia vyžaduje, aby v grafe automatu viedol z každého stavu prechod na každý symbol. Aj táto vlastnosť bude pre nás veľmi dôležitá. V algoritmoch, ktoré popisujeme, budeme pracovať s grafmi automatov a budeme predpokladať, že všetky prechody sú definované (okrem časti 3.2). Pomôže nám to vyhnúť sa zbytočným komplikáciám. Ku každému automatu existuje ekvivalentný úplný automat. Získame ho tak, že doplníme nový neakceptačný stav a budeme do neho viesť všetky chýbajúce prechody.

Definícia 2.1.10. *Veľkosť DKA A je mohutnosť množiny jeho stavov. Budeme používať označenie $|A|$.*

2.2 Regulárne výrazy

Definícia 2.2.1. *Regulárny výraz α nad abecedou Σ rekurzívne definujeme nasledujúcim spôsobom:*
- *$\alpha = \emptyset$ je regulárny výraz predstavujúci jazyk $L(\alpha) = \emptyset$,*
- *$\alpha = \varepsilon$ je regulárny výraz predstavujúci jazyk $L(\alpha) = \{\varepsilon\}$,*
- *$\alpha = a$ pre $a \in \Sigma$ je regulárny výraz predstavujúci jazyk $L(\alpha) = \{a\}$.*

Nech β, γ sú regulárne výrazy a $L(\beta), L(\gamma)$ sú jazyky, ktoré predstavujú, potom
- $\alpha = \beta + \gamma$ *je regulárny výraz predstavujúci jazyk* $L(\alpha) = L(\beta) \cup L(\gamma)$,
- $\alpha = \beta\gamma$ *je regulárny výraz predstavujúci jazyk* $L(\alpha) = L(\beta)L(\gamma)$,
- $\alpha = \beta^*$ *je regulárny výraz predstavujúci jazyk* $L(\alpha) = (L(\beta))^*$.

Definícia 2.2.2. *Veľkosť regulárneho výrazu α je počet symbolov zo Σ, ktoré sa v α vyskytujú. Budeme používať označenie $|\alpha|$.*

Označenie 2.2.1. *Nech α je regulárny výraz. Symbolom $\langle\alpha\rangle_i$, $i \geq 1$, budeme označovať i-ty symbol zo Σ vyskytujúci sa v α.*

Kapitola 3

Algoritmy na DKA

3.1 Minimalizácia DKA

Ako sme už spomínali v úvode, konečné automaty majú mnohé praktické aplikácie. V praxi sa samozrejme vynára problém ich veľkosti. Minimalizácia KA sa študuje už od päťdesiatych rokov a pôvodne sa objavila v prácach Huffamana a Moora.
Algoritmy riešiace tento problém sú používané v aplikáciách, v rozsahu od konštrukcie kompilátorov po minimalizáciu hardvérových obvodov. V tejto stati uvedieme jeden z nich, ten s najlepšou známou časovou zložitosťou. Najprv sa však pozrieme na problém podrobnejšie.

3.1.1 Problém minimalizácie

Problém: K danému DKA nájsť DKA s minimálnym počtom stavov, ktorý akceptuje rovnaký jazyk ako daný DKA.

Na základe Myhill-Nerodovej vety (pôvodne sa objavila v prácach [21, 22], možno ju nájsť aj v [8]) ku každému regulárnemu jazyku L existuje práve jeden minimálny automat. Odhliadnuc od izomorfizmu je jednoznačne určený reláciou ekvivalencie R, $uRv \iff \forall x \in \Sigma^* : ux \in L \Leftrightarrow vx \in L$, nasledovne: Označme $[u] = \{v \mid vRu\}$. $A_L = (K, \Sigma, \delta, [\varepsilon], F)$, kde $K = \{[u] \mid u \in \Sigma^*\}$, $F = \{[u] \mid u \in L\}$, $\delta([u], c) = [uc]$.

V našom prípade máme jazyk L daný automatom a chceme nájsť taký automat, ktorý nebude mať viac stavov ako vyššie definovaný automat a teda je minimálny. Nebudeme ale pracovať s jazykom L ako takým a pýtať sa, či obsahuje nejaké slovo. Budeme pracovať so zadaným automatom, s jeho stavmi, a preto definujeme a budeme používať ekvivalenciu stavov.

Definícia 3.1.1. *Stavy p, q sú ekvivalentné, ak $L(p) = L(q)$. Túto skutočnosť značíme $p \sim q$.*

Tvrdenie 3.1.1. *Automat A je minimálny práve vtedy, keď*

$$\forall p, q \in K : p \neq q \Rightarrow p \not\sim q$$

Dôkaz. "⇒": Žiadne dva rôzne stavy automatu A_L nie sú ekvivalentné. Ak totiž $[u] \neq [v]$, teda $(u, v) \notin R$, musí existovať $x \in \Sigma^*$ také, že práve jedno zo slov ux, vx patrí do L. Bez ujmy na všeobecnosti nech $ux \in L$ a $vx \notin L$, potom $x \in L([u])$ a $x \notin L([v])$, preto $[u] \not\sim [v]$.
"⇐": Každému stavu q daného automatu A akceptujúceho L môžeme priradiť stav $[u_q]$ minimálneho automatu A_L tak, že $(q_0, u_q) \vdash^* (q, \varepsilon)$. Ak $p \neq q$, z predpokladu máme $p \not\sim q$ a preto $L(p) \neq L(q)$. Musí teda existovať $x \in \Sigma^*$ také, že práve jeden zo stavov $\delta(q, x), \delta(p, x)$ je akceptačný. Bez ujmy na všeobecnosti, nech $\delta(q, x) \in F$, potom $u_q x \in L$, ale keďže $\delta(p, x) \notin F$, tak $u_p x \notin L$ a teda $(u_p, u_q) \notin R$. Z toho plynie $[u_q] \neq [u_p]$. Rôznym stavom A teda prislúchajú rôzne stavy A_L. Automat A preto nemá viac stavov ako A_L a teda je minimálny. □

Relácia $\sim$ je reláciou ekvivalencie na stavoch. Naším cieľom bude hľadať triedy tejto ekvivalencie. Definujeme preto nutné pojmy.

Definícia 3.1.2. *Množiny $K_1, ..., K_n$ tvoria rozklad K, ak sú po dvojiciach disjunktné (ak $i \neq j$ tak $K_i \cap K_j = \emptyset$) a pokrývajú K ($K = K_1 \cup ... \cup K_n$). K_i je trieda rozkladu.*

Definícia 3.1.3. *Rozklad napĺňa množinu stavov M, ak M je zjednotením niektorých jeho tried.*

Definícia 3.1.4. *Kongruencia automatu A je rozklad, ktorý je zlučiteľný (kompatibilny) s jeho prechodovou funkciou. To znamená, že ak q a q' sú v jednej triede rozkladu, tak $\delta(q, a)$ a $\delta(q', a)$ sú tiež v tej istej triede pre každé $q, q' \in K$ a pre každé $a \in \Sigma$.*

Definícia 3.1.5. *Rozklad $\{K_1, ..., K_n\}$ je hrubší ako rozklad $\{K'_1, ..., K'_m\}$, ak rozklad $\{K'_1, ..., K'_m\}$ napĺňa každú triedu K_i.*

Väčšina minimalizačných algoritmov má časovú zložitosť $\mathcal{O}(|K|^2)$ a používa jeden z nasledujúcich prístupov:
(1) Začínajú z najhrubšieho rozkladu $\{F, F^c\}$ a upravujú ho, pokiaľ nedostanú kongruenciu.
(2) Začínajú z najjemnejšieho rozkladu a spájajú triedy ekvivalencie(pričom sa používa len kongruencia).

Pre deterministické automaty nad jednopísmenovou abecedou je známy minimalizačný algoritmus [24] s lineárnou časovou zložitosťou. Riešenie problému minimalizácie spočíva v nájdení najhrubšieho rozkladu s vlastnosťami kongruencie. V článku je použitý prístup (2). Triedy počiatočného rozkladu sú jednoprvkové množiny a výsledok je dosiahnutý postupnosťou krokov, v ktorých sú niektoré dve triedy spojené, pričom sa zachováva vlastnosť kongruencie.

3.1.2 Hopcroftov algoritmus

Teraz sa pozrieme na algoritmus, ktorý pracuje na všetkých kompletných deterministických konečných automatoch a rieši problém ich minimalizácie. Pre daný DKA

$A = (K, \Sigma, \delta, q_0, F)$ Hopcroftov algoritmus, ktorý sa pôvodne objavil v [7], vypočíta najhrubšiu kongruenciu, ktorá napĺňa F.

Algoritmus pracuje s momentálnym rozkladom $\mathcal{P}$ množiny stavov a množinou $\mathcal{S}$. V $\mathcal{S}$ sú páry (C, a) kde $C \in \mathcal{P}$ a $a \in \Sigma$. Je to množina dvojíc, ktoré čakajú na spracovanie.

Je použitý prístup (1) a teda začíname z rozkladu $\mathcal{P} = \{F, F^c\}$ a do $\mathcal{S}$ na začiatku uložíme dvojice $(min(F, F^c), a)\ \forall a \in \Sigma$. $\mathcal{P}$ upravuje tak, že nahrádzame jednu triedu rozkladu dvomi, ktoré vzniknú jej rozdelením.
Postupujeme nasledovne:

1. Vyberieme jednu dvojicu (C, a) z $\mathcal{S}$.
   Pre každú triedu B z rozkladu $\mathcal{P}$ vykonáme kroky 2 a 3.

2. Overíme, či (C, a) *rozdeľuje* B. To nastáva práve vtedy, keď existujú stavy $p, q \in B$ také, že $\delta(q, a) \in C$ a $\delta(p, a) \notin C$. Musíme sa teda pozrieť na prechody vedúce z B na písmeno a, či sú tam také, ktoré vedú do C aj mimo C (do C^c). Ak (C, a) nerozdeľuje B, prejdeme na spracovanie ďalšej triedy rozkladu z $\mathcal{P}$. Inak vytvoríme nové triedy $B_1 = \{q \in B \mid \delta(q, a) \in C\}$ a $B_2 = \{q \in B \mid \delta(q, a) \notin C\}$.

3. Triedu B nahradíme v $\mathcal{P}$ triedami B_1 a B_2. Pre každé $b \in \Sigma$, ak sa (B, b) nachádza v $\mathcal{S}$, nahradíme ho pármi (B_1, b) a (B_2, b). Inak vyberieme menšiu z nich $B' = min(B_1, B_2)$ a (B', b) pridáme do $\mathcal{S}$.

4. Ak je $\mathcal{S}$ prázdna, algoritmus skončí, inak pokračujeme krokom 1.

Zmienime sa o korektnosti tohto algoritmu a potom sa budeme venovať jeho časovej zložitosti.

Zastavenie: Dvojice sú pridané do $\mathcal{S}$ vtedy, keď je nejaká trieda rozdelená. To samozrejme môže pokračovať až dovtedy, kým v každej triede bude iba jeden stav (čo by tiež znamenalo, že automat na vstupe už bol minimálny). Počet pridaní je ale obmedzený tým, že triedy nemožno rozdeľovať do nekonečna. V kroku 1 je vždy nejaký pár z S vybraný, preto sa musí $\mathcal{S}$ časom vyprázdniť a teda algoritmus zastaví.

Dôkaz správnosti: Potrebujeme ukázať, že po skončení algoritmu

$$p \sim q \Leftrightarrow \exists B_i \in \mathcal{P} : p \in B_i \wedge q \in B_i$$

$\Rightarrow$:
Obmenená implikácia sa dá ľahko dokázať indukciou na počet rozdelených tried (počet vykonaní kroku 3).
$\Leftarrow$:
Pozrime sa na prípad, že $p \not\sim q \wedge \exists a \in \Sigma : q' = \delta(q, a)$ a $p' = \delta(p, a)$ sú v rôznych blokoch. Vtedy vieme, že blok obsahujúci q', p' bol rozdelený a do $\mathcal{S}$ bola vložená dvojica (C, a), kde práve jeden stav z q', p' je v C. Pri jej spracovaní bol blok obsahujúci p, q (B_i) rozdelený a p, q sa ocitli v samostatných blokoch.

Ďalej ukážeme, že také $a \in \Sigma$, pre ktoré p', q' sú v rôznych blokoch vždy existuje. Keďže $p \not\sim q$, tak $L(q) \neq L(p)$ a teda existuje najkratšie $w = w_1 w_2 ... w_k$ také, že $w \notin L(q) \cap L(p) \wedge w \in L(q) \cup L(p)$. Pozrime sa na výpočty na w, začínajúce v p a q.
$(p, w_1 ... w_k) \vdash (p_1, w_2 ... w_k) \vdash ... \vdash (p_{k-1}, w_k) \vdash (p_k, \varepsilon)$
$(q, w_1 ... w_k) \vdash (q_1, w_2 ... w_k) \vdash ... \vdash (q_{k-1}, w_k) \vdash (q_k, \varepsilon)$
Zjavne p_k, q_k sú v rôznych blokoch, keďže jeden je z F a druhý z F^c. Pre každé $1 \leq i \leq k : p_i \not\sim q_i$. Takže z platnosti prvého prípadu dostávame $p_{k-1} \not\sim q_{k-1} \wedge p_k, q_k$ sú v rôznych blokoch $\Rightarrow p_{k-1}, q_{k-1}$ sú v rôznych blokoch. Takto môžeme pokračovať, až dostaneme implikáciu $p \not\sim q \wedge p_1, q_1$ sú v rôznych blokoch $\Rightarrow p, q$ sú v rôznych blokoch.

□

3.1.3 Časová zložitosť Hopcroftovho algoritmu

Horná hranica $\mathcal{O}(n \log n)$

Nech $\mathcal{P} = \{B_1, ..., B_m\}$. Kroky 2 a 3 tvoria jednoduchý cyklus.

Tvrdíme, že čas potrebný na vykonanie tohto cyklu pre danú dvojicu $(B_i, a) \in \mathcal{S}$ je úmerný počtu prechodov na symbol a končiacich stavmi v B_i. Aby sme toto nahliadli, stačí, ak si uvedomíme, že počas cyklu nepotrebujeme testovať pre každú triedu $B_j \in \mathcal{P}$ či (B_i, a) rozbíja B_j. Stačí, keď nájdeme také stavy t, že $\delta(t, a) \in B_i$(na toto si môžeme na začiatku algoritmu zostrojiť inverznú tabuľku prechodov δ^{-1}). Keď taký stav t nájdeme, trieda, v ktorej sa nachádza, je uložená a t je pridaný na zoznam stavov, ktoré z nej treba vylúčiť (a vytvoriť tak dve nové triedy). Trieda je potom pridaná do zoznamu upravených tried. Takýchto tried nie je viac ako nájdených stavov. Triedy budeme rozdeľovať až po nájdení všetkých takých stavov.

Pre triedu B_i a symbol a budeme symbolom a_i označovať počet prechodov na symbol a končiacich stavmi v B_i a teda aj čas, ktorý treba na upravenie celého rozkladu podľa dvojice (B_i, a).

Predpokladajme, že $\mathcal{P} = \{B_1, ..., B_m\}$ a $\mathcal{S} \supseteq \{(B_j, a) \mid j \in \{i_1, ..., i_r\}\}$. Pokračujme ďalej indukciou. Tvrdením indukcie je, že čas potrebný na prejdenie cyklu pre zvolený symbol a, t.j. pre všetky možné dvojice (C, a), ktoré sa môžu vyskytnúť v $\mathcal{S}$, je ohraničený formulou, ktorú označíme ako T_a. Indukčný predpoklad:

$$T_a = \sum_{j=1}^{r} a_{i_j}.\ \log|B_{i_j}| + \sum_{j=r+1}^{m} a_{i_j}.\ \log\frac{|B_{i_j}|}{2}$$

Výraz tvaru $a_k.\ \log|B_k|$ v T_a ohraničuje čas, ktorý môžeme potrebovať na úplné spracovanie páru $(B_k, a) \in \mathcal{S}$. V tomto výraze je zahrnuté samotné spracovanie páru (B_k, a) a ďalej všetkých párov (C, a), ktoré môžu byť pridané do $\mathcal{S}$ pri rozdelení B_k a jej podmnožín, teda $C \subset B_k$. Výraz tvaru $a_k.\ \log\frac{|B_k|}{2}$ v T_a ohraničuje čas, ktorý môžeme potrebovať na rovnaké spracovanie páru (B_k, a) za predpokladu, že na začiatku $(B_k, a) \notin \mathcal{S}$. Obe sumy teda zahŕňajú čas potrebný na spracovanie všetkých párov (C, a), $C \subseteq K$.

Báza indukcie: Na začiatku algoritmu je $\mathcal{P} = \{F, F^c\}$ a $\mathcal{S} = \{(min(F, F^c), a) \mid a \in \Sigma\}$. Označme $B_1 = min(F, F^c)$ a $B_2 = B_1^c$. Pre pevne zvolené $a \in \Sigma$ platí:

$$T_a = a_1.\ \log|B_1| + a_2.\ \log\frac{|B_2|}{2} \leq a_1.\ \log|K| + a_2.\ \log|K| = (a_1 + a_2).\ \log|K| = n.\ \log n$$

Indukčný krok budeme vykonávať vzhľadom na počet prejdení cyklu tvoreného krokmi 2 a 3, čo je tiež počet vybraní nejakej dvojice z $\mathcal{S}$. Predpokladáme, že pred vybraním dvojice je čas potrebný na spracovanie všetkých párov (C, a), ktoré sa môžu počas výpočtu v $\mathcal{S}$ vyskytnúť, ohraničený výrazom T_a. Po jej spracovaní bude tento čas ohraničený novou hodnotou; označme ju $T_a^{'}$. Nech bola vybraná dvojica (B_l, b), budeme uvažovať dva prípady:

1. $b \neq a$. Čas potrebný na spracovanie tohto páru nie je zahrnutý v T_a. Musíme ale ukázať, že po spracovaní toho páru bude $T_a^{'} \leq T_a$, aj keď táto dvojica rozbije niektoré triedy. Opäť budeme uvažovať dva prípady.

   (a) Ak bola rozdelená trieda B_j taká, že $(B_j, a) \in \mathcal{S}$, tak výraz $a_j.\ \log|B_j|$ z T_a bude v $T_a^{'}$ nahradený výrazom $a_{j'}.\ \log|B_{j'}| + (a_j - a_{j'}).\ \log|B_j - B_{j'}|$, ktorý je vždy menší ako pôvodný výraz, keďže $|B_{j'}| < |B_j|$ a tiež $|B_j - B_{j'}| < |B_j|$. Takže $T_a^{'} \leq T_a$.

   (b) Ak bola rozdelená trieda B_j taká, že $(B_j, a) \notin \mathcal{S}$, tak výraz $a_j.\ \log\frac{|B_j|}{2}$ z T_a bude v $T_a^{'}$ nahradený výrazom $a_{j'}.\ \log|B_{j'}| + (a_j - a_{j'}).\ \log\frac{|B_j - B_{j'}|}{2}$, kde $(B_{j'}, a)$ bola pridaná do $\mathcal{S}$ (teda $|B_{j'}| \leq \frac{|B_j|}{2}$). Tento výraz je preto menší ako $a_{j'}.\ \log\frac{|B_j|}{2} + (a_j - a_{j'}).\ \log\frac{|B_j|}{2}$, a teda aj menší ako pôvodný výraz $a_j.\ \log\frac{|B_j|}{2}$. Takže $T_a^{'} \leq T_a$.

   Opakovaným aplikovaním týchto úvah dostávame, že $T_a^{'} \leq T_a$, aj keď (B_l, a) rozbije viac ako jednu triedu.

2. $b = a$. Na spracovanie páru (B_l, a) je potrebný čas úmerný a_l. Vieme, že po jeho spracovaní (B_l, a) už nebude v $\mathcal{S}$. To, že sa pri spracovaní páru (B_l, a) rozdelili nejaké triedy už neuvažujeme, pretože sme v prípade 1. ukázali, že to nezhorší potrebný čas. Teda po spracovaní páru (B_l, a) bude hodnota

$$T_a^{'} \leq a_l + a_l.\ \log\frac{|B_l|}{2} + \sum_{\substack{j=1 \\ i_j \neq l}}^{r} a_{i_j}.\ \log|B_{i_j}| + \sum_{j=r+1}^{m} a_{i_j}.\ \log\frac{|B_{i_j}|}{2}$$

   A keďže

$$a_l + a_l.\ \log\frac{|B_l|}{2} = a_l.(1 + \log|B_l| - \log 2) = a_l.\ \log|B_l|$$

   potom jasne aj $T_a^{'} \leq T_a$.

Tým sme ukončili dôkaz toho, že čas potrebný na spracovanie všetkých dvojíc so symbolom a je ohraničený výrazom T_a a ten na začiatku algoritmu nie je väčší ako $n.\log n$. Tento čas vynásobíme počtom symbolov a pripočítame k nemu inicializáciu (vytvorenie δ^{-1}, $\mathcal{P}$ a $\mathcal{S}$). Keďže symbolov je konštantne veľa a inicializácia nepresahuje čas výpočtu, môžeme povedať, že algoritmus pracuje v čase $\mathcal{O}(n \log n)$.

Dolná hranica $\Omega(n \log n)$

J. Berstel a O. Carton prezentujú v článku [14] triedu automatov, pre ktorú Hopcroftov algoritmus potrebuje čas $\Omega(n \log n)$. Popisujú tiež spôsob, akým vyberať dvojice $(C, a) \in \mathcal{S}$ tak, aby bol čas behu čo najhorší.

3.2 Hľadanie minimálneho acyklického DKA

V úvode sme spomínali, že konečné automaty majú svoje miesto aj v spracovaní prirodzeného jazyka. Hlavný dôvod používania konečných automatov v tejto oblasti je, že ich reprezentácia množiny slov je kompaktná a vyhľadanie slova v slovníku reprezentovanom automatom je veľmi rýchle - úmerné dĺžke slova. Špecifické postavenie v tejto oblasti majú deterministické acyklické konečné automaty, ktorými sa slovníky reprezentujú.

Definícia 3.2.1. *Deterministický konečný automat* $A = (K, \Sigma, \delta, q_0, F)$ *je acyklický, ak pre každé dva stavy* $p, q \in K$ *platí:*

$$(\exists v \in \Sigma^* : (p, v) \vdash^* (q, \varepsilon)) \Rightarrow (\nexists u \in \Sigma^* : (q, u) \vdash^* (p, \varepsilon))$$

Definícia hovorí len toľko, že ak máme v grafe automatu cestu z p do q, potom nemáme cestu späť. Čiže počas výpočtu na každom slove, automat prejde jedným stavom najviac raz. Acyklické automaty majú tento názov preto, lebo ich grafy sú acyklické.

Problém: Pre danú konečnú množinu slov nájsť(skonštruovať) minimálny DKA, ktorý akceptuje práve danú množinu.

Acyklické automaty akceptujú práve všetky konečné jazyky. Teda každý jazyk akceptovaný acyklickým automatom je konečný a ku každému konečnému jazyku existuje acyklický automat, ktorý ho akceptuje. V nasledujúcej časti (3.2.1) ukážeme, ako taký automat zostrojiť.

V prvej časti sme si ukázali minimalizačný algoritmus, ktorý pracuje na všetkých automatoch a teda aj na acyklických. Jedna možnosť ako riešiť tento problém je preto zostrojiť *nejaký* automat pre danú množinu a následne ho *minimalizovať* použitím tohto algoritmu. Toto riešenie však nie je zďaleka také efektívne ako by sa mohlo zdať. Prekvapivo je to najpomalší postup. Oproti algoritmom vyžadujúcim lineárny čas, vyššie zmienený algoritmus vyžaduje navyše $\mathcal{O}(log\ n)$ režijných výdavkov. O niečo lepšie sú na tom *poloinkrementačné* algoritmy (Watson [28], Revuz [26]).

Vo všeobecnosti najrýchlejšie a vyžadujúce najmenej pamäte sú *inkrementačné* algoritmy. Najprv si ukážeme, ako pracuje takýto algoritmus na množine slov, ktorá je utriedená. Potom si podmienky sťažíme a ukážeme čo robiť, ak sa na utriedenie nemôžeme spoliehať.

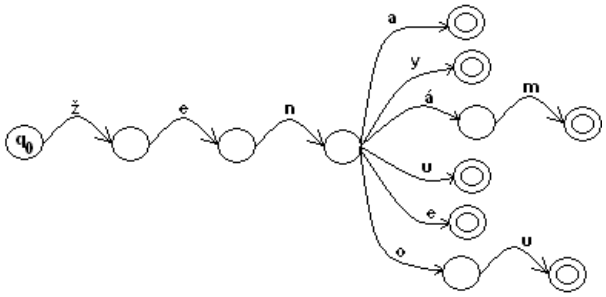
3.2.1 Budovanie stromu

Slovník alebo automat, ktorý ho modeluje, má stromovú štruktúru. Počiatočný stav q_0 je koreň, prechody sú označené písmenami abecedy, listy sú akceptačné stavy(môžeme mať aj iné akceptačné stavy) a písmená na jednej ceste z koreňa do akceptačného stavu tvoria slovo. Takýto strom budeme nazývať *písmenový strom.*

Písmenový strom pre danú množinu slov vytvoríme pridávaním stavov. Ak už máme slovník obsahujúci slová a chceme pridať ďalšie slovo, jednoducho v strome nájdeme najdlhšiu cestu, ktorá zodpovedá prefixu pridávaného slova. K tejto ceste pridáme ďalšie stavy a prechody medzi nimi označíme písmenami ostávajúceho sufixu.

Je dobré si v tomto momente uvedomiť, že automat, ktorý vznikne takýmto jednoduchým pridávaním stavov môže mať v porovnaní s minimálnym automatom veľa stavov. Keďže slovenské tvaroslovie je pomerne pravidelné, veľa slov v našom prirodzenom jazyku má rovnaké sufixy. Tieto sufixy v písmenovom strome vytvárajú izomorfné podstromy. Minimálny automat sa bude od písmenového stromu odlišovať práve tým, že nebude mať žiadne izomorfné podstromy a teda ani izomorfné (ekvivalentné) stavy.

Príklad: Keby sme chceli do slovníka pridať tvary slova *žena* v jednotnom čísle, vznikol by nám podstrom s deviatimi stavmi. A s každým slovom, ktoré sa podľa tohto vzoru skloňuje, by nám pribudol rovnaký(izomorfný) podstrom.



Ak máme v strome dva izomorfné podstromy, ich korene sú ekvivalentné stavy, lebo im prislúchajúce jazyky sa rovnajú. No takéto hľadanie ekvivalentných stavov porovnávaním množín slov je výpočtovo náročné. Preto teraz uvedieme rekurzívnu definíciu ekvivalencie stavov, ktorá nám toto hľadanie výrazne uľahčí.

$p \sim q$ práve vtedy, keď sú splnené všetky 4 nasledujúce podmienky:

1. oba stavy sú akceptačné alebo neakceptačné
2. majú rovnaký počet vychádzajúcich prechodov
3. zodpovedajúce prechody v oboch stavoch sú označené rovnakým symbolom $(\forall a \in \Sigma : \delta(p, a) \neq \emptyset \Leftrightarrow \delta(q, a) \neq \emptyset)$

4. zodpovedajúce prechody vedú do ekvivalentných stavov

Oba algoritmy prezentované ďalej budú mať spoločné to, že stavy, ktoré reprezentujú triedu rozkladu(ako sme to popisovali v časti 3.1) budú uložené v registri.

3.2.2 Utriedená množina slov

Predpokladajme, slová pridávame v lexikografickom usporiadaní, a že máme už vybudovanú časť slovníka. Pozrime sa na to, ako vyzerá pridanie jedného slova. Už vieme, ako budeme slová do automatu pridávať. Nájdeme najdlhšiu cestu od koreňa, ktorá zodpovedá prefixu slova a zvyšok stavov a prechodov pre sufix doplníme.

Slovo, ktoré sme naposledy pridali do slovníka označme u a slovo, ktoré pridávame označme v. Keďže slová pridávame v lexikografickom poradí, najdlhší prefix slova v v slovníku je tiež najdlhším spoločným prefixom u a v. Sufix slova u budeme minimalizovať (ak je zvyšok slova prázdny, teda u bolo prefixom v, minimalizácia sa nekoná). To budeme robiť smerom od posledného stavu slova u až po stav, od ktorého sa slová u a v líšia. Pre každý stav na tejto ceste budeme zisťovať, či sa v slovníku (presnejšie v registri stavov) nachádza nejaký s ním ekvivalentný. Ak áno, nahradíme ho ekvivalentným stavom, ak nie, uložíme ho do registra. Dopredný proces minimalizácie nám umožní zmeniť poslednú podmienku ekvivalencie nasledovne:

4'. zodpovedajúce prechody vedú do rovnakých stavov.

Preto bude jednoduché porovnať stav s každým stavom v registri a zistiť tak, či v slovníku nie je už ekvivalentný stav. Keď je minimalizácia ukončená, pridáme nové stavy pre sufix slova v. Pred pridaním ďalšieho slova budeme tieto stavy minimalizovať.

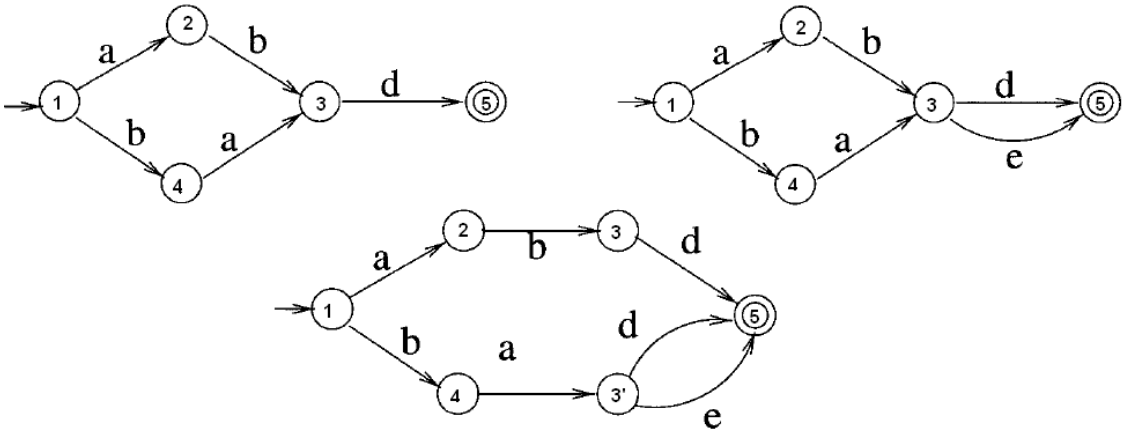
Potom, čo do slovníka pridáme všetky slová, spustíme doprednú minimalizačnú fázu ešte raz (stavy, ktoré sme naposledy pridali sme neminimalizovali). Prejdeme celú cestu, ktorá tvorí posledné pridané slovo a skontrolujeme každý stav na nej.

Pamäťová zložitosť pozostáva z pamäte potrebnej na uloženie automatu, čo je úmerné počtu stavov a celkovému počtu prechodov, a z pamäte pre register stavov. Keďže z každého stavu vychádza iba konečne veľa prechodov (pre dané Σ), na uloženie automatu potrebujeme najviac $\mathcal{O}(n)$ miesta. Rovnako v registri nebude nikdy uložených viac ako $\mathcal{O}(n)$ stavov. Pamäťová zložitosť algoritmu je preto $\mathcal{O}(n)$.

Nájdenie spoločného prefixu a pridanie nových stavov vyžaduje konštantný čas. Zistenie, či sa v registri nachádza ekvivalentný stav a jeho prípadné nahradenie vyžaduje čas $\mathcal{O}(\log n)$. Počet stavov, pre ktoré sa v registri vyhľadáva, je rovný počtu stavov v písmenom strome pre danú množinu slov. A to nie je viac ako počet symbolov vo vstupnom zozname. Celkový čas behu algoritmu je teda $\mathcal{O}(l \log n)$, kde l je celkový počet symbolov vo vstupnej množine slov.

3.2.3 Neutriedená množina slov

Nie vždy je možné predpokladať, že vstupná množina bude utriedená. Dokonca sa môže stať, že jednotlivé slová vznikajú počas konštrukcie automatu, alebo jednoducho potrebujeme do hotového automatu pridať nejaké nové slovo.



Obr. 3.1: Obrázok je uvedený v článku Incremental Construction of Minimal Acyclic Finite-State Automata [4] ako príklad.
Do automatu vľavo chceme pridať slovo bae. *Napravo vidíme automat, do ktorého bol jednoducho pridaný a následne minimalizovaný sufix. Tým sme pridali aj nechcené slovo* abe. *Stav* ***3*** *je sútok a je to zároveň aj posledný stav spoločného prefixu. Na spodnom automate je stav* ***3*** *skopírovaný a tým vznikol stav* ***3'****. Následne bol pridaný a minimalizovaný sufix.*

Na rozdiel od predchádzajúceho algoritmu v tomto prípade nemôžeme využiť spoločný prefix slov, ktoré sú po sebe pridávané. Pri pridávaní slova budeme hľadať najdlhší prefix, ktorý už je v automate. Nemá preto zmysel čakať s minimalizáciou stavov, ktoré tvoria sufix, do príchodu ďalšieho slova. Stavy sufixu budeme minimalizovať hneď potom ako ich pridáme.

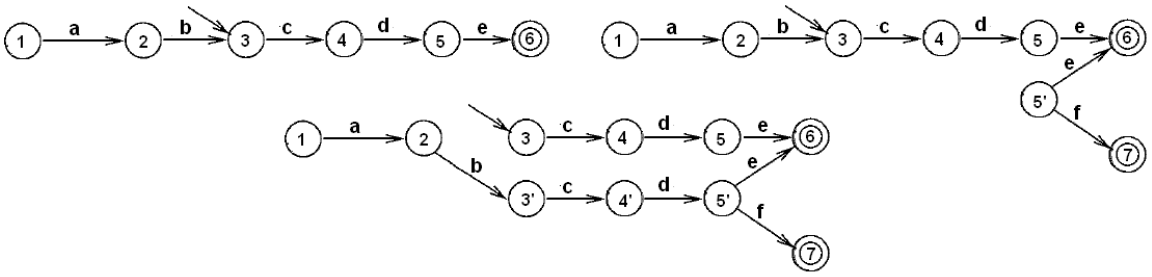
Na ceste v prefixe sa môže nachádzať stav, do ktorého vchádza viac prechodov. Takému stavu budeme hovoriť *sútok*. Vznikol pri minimalizácii nejakého sufixu spojením izomorfných podstromov, ktoré sa v písmenovom strome nachádzali viackrát(ako sme to popisovali v časti 3.2.1). My potrebujeme jeden z týchto podstromov zmeniť a teda už nebude viac izomorfný s ostatnými. V predchádzajúcom prípade sa sútok nemohol vyskytovať v prefixe, ktorý sme v automate našli. Táto komplikácia spôsobí, že nebudeme môcť stavy sufixu jednoducho pridať a minimalizovať. Jednoduchým pridaním nových stavov by sme do automatu pridali viac ako jedno slovo, ako je to ukázané na obrázku 3.1.

Aby sme zabránili pridaniu nesprávnych slov do automatu, stavy v spoločnom prefixe od posledného až po prvý sútok budeme musieť *skopírovať*. Stav skopírujeme tak, že vytvoríme nový stav, ktorý má rovnaké vychádzajúce prechody vedúce do rovnakých stavov.

Algoritmus na pridanie jedného slova, označme ho w, sa bude skladať z týchto krokov:

1. Nájdeme najdlhší spoločný prefix slova w, ktorý je v automate.

2. Zistíme, či sa na ceste v spoločnom prefixe nachádza sútok. Ak sme sútok nenašli, pridáme stavy pre sufix. Minimalizovať ich budeme od konca sufixu. Pre každý nový stav zistíme, či v registri nie je s ním ekvivalentný. Ak áno,



Obr. 3.2: Do horného automatu vľavo chceme pridať slovo abcdf. *Nájdeme spoločný prefix, ktorý tvorí slovo* abcd. *Na ceste v prefixe sa nachádza sútok (stav* 3*). Preto posledný stav v prefixe (stav* 5*) skopírujeme a vznikne stav* 5'. *K nemu pridáme stavy pre sufix a vznikne tak automat napravo. Na automate nižšie vidno, ako pokračujeme. Postupne kopírujeme stavy* 4 a 3. *Napokon zmeníme prechod zo stavu* 2 *na písmeno* b, *tak aby viedol do novej cesty.*

nahradíme ho stavom z registra, ak nie, uložíme ho do registra aby reprezentoval novú triedu ekvivalencie. Ďalej budeme pokračovať krokom 4.

Ak sme v prefixe sútok našli, musíme vytvoriť kópiu posledného stavu v prefixe a k nemu pripojiť nové stavy pre sufix a minimalizovať ich rovnako ako je to popísane vyššie. Prejdeme na ďalší krok.

3. Stavy na ceste od predposledného stavu v nájdenom prefixe až po prvý sútok budeme kopírovať. Pre každý stav vytvoríme jeho kópiu a zmeníme prechody tak aby sme vytvorili novú cestu. Tým vytvoríme kópiu cesty a prechod v prefixe, ktorý viedol do pôvodnej cesty, bude smerovať do jej kópie. Situácia je znázornená na obrázku 3.2.

4. Pokúsime sa minimalizovať prefix. Takže pre každý stav od konca až po q_0 budeme v registri hľadať s ním ekvivalentný a následne ho ním nahrádzať.

V kroku 3 je dôležité prejsť celý prefix a nájsť prvý sútok. Pri minimalizácii sufixu v kroku 2 sme mohli na ceste v prefixe vytvoriť nový sútok. Preto sútok, ktorý sme našli v kroku 2 už nemusí byť prvý. Opomenutie tejto situácie by viedlo k vytvoreniu cyklu [4]. V kroku 4 prepočítavame triedy ekvivalencie, pretože jazyky prislúchajúce stavom v prefixe sa zmenili. Tento krok môže viesť k nájdeniu menšieho automatu.

V prípade, že sa nemôžeme spoliehať na utriedenie vstupnej množiny je ťažšie spojiť proces pridávania slova s minimalizáciou. Často je však efektívnejšie použiť algoritmus opísaný vyššie, než triediť vstupné dáta, alebo oddeliť budovanie slovníka a minimalizáciu do dvoch samostatných fáz. Porovnanie algoritmov pre nájdenie minimálneho acyklického konečného automatu je možné nájsť v [3].

Kapitola 4

Algoritmy na NKA

Hlavnou motiváciou pre prácu s nedeterministickými konečnými automatmi je manipulácia s regulárnymi výrazmi [16]. Regulárne výrazy poskytujú vhodnú notáciu pre regulárne jazyky v textovom prostredí, zatiaľ, čo konečné automaty sú vhodnejšie ako dátová štruktúra pre programovacie účely. V praxi tým vzniká nasledujúci problém.

Problém: K danému regulárnemu výrazu r nájsť DKA, ktorý akceptuje jazyk popísaný výrazom r.

Tento problém budeme zapisovať ako $RV \rightarrow DKA$. Riešenie pozostáva z dvoch krokov. K regulárnemu výrazu nájdeme NKA a k tomu zostrojíme ekvivalentný DKA, t.j. $RV \rightarrow NKA \rightarrow DKA$. Stručne sa pozrime na oba podproblémy.

1. $RV \rightarrow NKA$
   Prvé algoritmy riešiace tento problém možno nájsť v [27, 25]. Tieto algoritmy sú pomerne efektívne. Z druhého podproblému ale vyplýva, že je nutné hľadať optimálnejšie riešenia. Náplňou tejto kapitoly je uviesť doterajšie výsledky.

2. $NKA \rightarrow DKA$
   Rabin a Scott vo svojom článku [25] popisujú ako transformovať NKA na DKA. Pre NKA s n stavmi dostaneme ekvivalentný DKA s najviac 2^n stavmi. Moore[20] neskôr ukázal, že táto hranica je tesná. To znamená, že existujú regulárne jazyky a k nim NKA s n stavmi, ale ich minimálne DKA majú rádovo 2^n stavov.

Pre druhý podproblém vo všeobecnosti platí, že s čím väčším NKA začneme, tým väčší DKA dostaneme. Vzniknutý DKA môžeme následne minimalizovať, ale predtým ho potrebujeme uložiť do pamäte. Už pre malé n je veľký rozdiel v tom, či skladujeme 2^n alebo 2^{n+5} stavov. Preto je výhodné začínať z čo najmenšieho NKA.

Budeme sa zaoberať prvým podproblémom. Uvedieme algoritmy, ktoré používajú rôzny prístup k jeho riešeniu. Cieľom vždy bude dopracovať sa k čo najmenšiemu NKA pre zadaný regulárny výraz.

Tento problém je podobný problému z časti 3.2. Jeho riešenie by sme mohli rozdeliť opäť na časť nájdenia nejakého NKA pre regulárny výraz a na časť minimalizácie tohto

NKA. Tento postup budeme bližšie popisovať v časti 4.2. Celkové riešenie potom bude mať nasledujúcu štruktúru:

$$\text{RV} \rightarrow \text{NKA} \rightarrow \text{malý (n-stavový) NKA} \overset{\theta(2^n)}{\rightarrow} \text{DKA} \rightarrow \text{minimálny DKA}.$$

Iný prístup používa priamu konštrukciu. Z regulárneho výrazu zostrojíme čo možno najmenší NKA v jednom kroku. Známe výsledky prinesieme v časti 4.3.2. Štruktúru riešenia možno popísať takto:

$$\text{RV} \rightarrow \text{malý (n-stavový) NKA} \overset{\theta(2^n)}{\rightarrow} \text{DKA} \rightarrow \text{minimálny DKA}.$$

4.1 Minimalizácia NKA

Definícia 4.1.1. *NKA $A = (K, \Sigma, \delta, q_0, F)$ je minimálny, ak pre každý NKA $A' = (K', \Sigma', \delta', q_0', F')$ platí: $L(A) = L(A') \Rightarrow |A'| \geq |A|$.*

Problém: K danému NKA nájsť minimálny NKA, ktorý akceptuje rovnaký jazyk ako daný NKA.

Zatiaľ, čo pri DKA sme mali jasný cieľ v podobe unikátneho minimálneho automatu, pri nedeterministických automatoch takú výhodu nemáme. Môže existovať niekoľko rôznych minimálnych NKA akceptujúcich jeden jazyk. Čo je však horšie, nepoznáme polynomiálny deterministický algoritmus ako nájsť minimálny NKA pre daný jazyk. Problém nájdenia minimálneho NKA je PSPACE-úplný a teda je považovaný za veľmi ťažký. Presnejšie, nasledujúci rozhodovací problém je PSPACE-úplný [15]:

Vstup: DKA A a celé číslo k
Otázka: Existuje NKA s najviac k stavmi akceptujúci $L(A)$?

V schémach, ktoré sme uviedli vyššie teda popisom *malý (n-stavový) NKA* nemyslíme skutočne minimálny NKA, ale rozumne malý. Taký, na ktorého nájdenie poznáme efektívny algoritmus.

4.2 Redukcia veľkosti NKA

V tejto časti vychádzame z článku [18] a z niektorých jeho referencií [12, 13, 17].

Pridaním nedeterminizmu do konečných automatov sa zmenia niektoré ich pekné vlastnosti. Počet stavov už nebude o automate vypovedať toľko ako pri deterministickom modeli. Do miery zložitosti pre NKA je vhodné zahrnúť počet prechodov, pretože práve to udáva koľko miesta potrebujeme aby sme mohli automat uložiť (napríklad do pamäte počítača). Podobne ako pre DKA (definícia 2.1.10) definujeme veľkosť pre NKA.

Definícia 4.2.1. *Veľkosť NKA A definujeme ako $|A| = |K| + |\delta|$, kde $|\delta|$ je počet prechodov.*

Pre veľkosť NKA sa používajú rôzne definície. My budeme v tejto časti používať definíciu 4.2.1, rovnako ako autori článku [12]. Spájaním stavov sa snažíme veľkosť automatu zmenšiť. Myšlienka postupu je podobná tej, ktorú sme uviedli v časti 3.2. Vtedy sme považovali dva stavy za ekvivalentné ak k nim prislúchajúce jazyky boli zhodné. To bolo práve vtedy, keď z nich viedli prechody na rovnaké prímená do ekvivalentných stavov a boli oba buď akceptačné alebo neakceptačné.

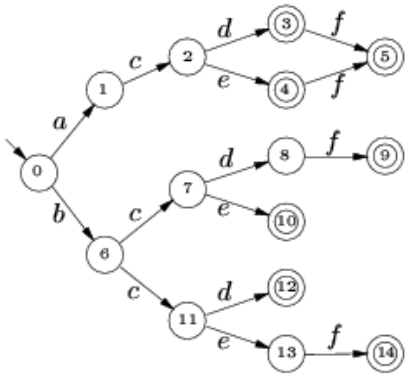
V nedeterministickom prípade máme množinu stavov, do ktorej vedú prechody z jedného stavu na jeden zvolený symbol. Budeme spájať také stavy, o ktorých vieme, že sú ekvivalentné(podľa definície 3.1.1), ale nie všetky stavy vo výslednom automate budú podľa pôvodnej definície neekvivalentné.

Budeme počítať reláciu ekvivalencie $\equiv$ na stavoch daného NKA tak, že bude platiť: $p \equiv q \Rightarrow L(p) = L(q)$. Symbolom $[q]_{\equiv}$ budeme označovať triedu ekvivalencie, v ktorej je stav q. Nech S je množina stavov, potom $S/_{\equiv} = \{[q]_{\equiv} \mid q \in S\}$ je množina tried. Definujme $\equiv$ nasledovne: $\forall p, q \in K : p \equiv q \Leftrightarrow$

1. $p, q \in F \vee p, q \notin F$
2. $\forall a \in \Sigma : (p \equiv q \Rightarrow \delta(p, a)/_{\equiv} = \delta(q, a)/_{\equiv})$

Stavy ekvivalentné podľa definície 3.1.1 volajme pre zrozumiteľnosť *neodlíšiteľné* a stavy, ktoré sú v relácii $\equiv$ volajme *ekvivalentné*. Nasledujúci obrázok ukazuje NKA, ktorý obsahuje neekvivalentné neodlíšiteľné stavy. Preto nie všetky stavy, ktoré sú neodlíšiteľné môžu byť spojené pomocou takejto relácie.

$\equiv$	
[0]	$\{0\}$
[1]	$\{1\}$
[2]	$\{2\}$
[3]	$\{3, 4\}$
[5]	$\{9, 10, 12, 14\}$
[6]	$\{6\}$
[7]	$\{7\}$
[8]	$\{8, 13\}$
[11]	$\{11\}$



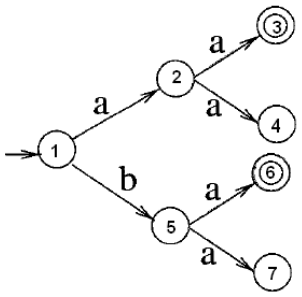
Obr. 4.1: Na obrázku vidíme stavy 3 *a* 8*, ktoré nie sú ekvivalentné (pretože jeden je akceptačný a druhý nie). Rovnako aj stavy* 4 *a* 13 *nie sú ekvivalentné. Na základe toho* 2 *a* 7 *aj* 2 *a* 11 *sú neekvivalentné. Preto stavy* 1 *a* 6 *nemôžu byť v relácii* $\equiv$ *aj keď* $L(1) = L(6) = \{cd, ce, cdf, cef\}$.

Budeme hľadať komplement relácie $\equiv$:

$$(\exists p' \in \delta(p, a) : (\forall q' \in \delta(q, a) : p' \not\equiv q')) \Rightarrow p \not\equiv q \qquad (*)$$

Na začiatku, rovnako ako pri algoritmoch na DKA, budú za neekvivalentné považované tie dva stavy, z ktorých jeden je akceptačný a druhý nie.
Teda $\forall (p, q) \in (K - F) \times F : p \not\equiv q$. Ak však poznáme dva neekvivalentné stavy

$\equiv$	
[1]	$\{1\}$
[2]	$\{2,5\}$
[3]	$\{3,6\}$
[4]	$\{4,7\}$



Obr. 4.2: Stavy 4 a 6 sú neekvivalentné, ale ich predchodcovia 2 a 5 sú ekvivalentní.

$r \not\equiv s$, nemôžeme ich predchodcov na zvolený symbol a (podľa inverznej tabuľky stavov) označiť ako neekvivalentné stavy. Ukazuje to obrázok 4.2.

Hľadanie tried $\not\equiv$ preto vyžaduje viac času, než sme potrebovali pri algoritmoch pre DKA. Aby sme prehlásili stavy za neekvivalentné, musíme zistiť, či je splnená podmienka $(*)$.

$\equiv$ je sprava invariantná vzhľadom na zadaný NKA A. Výsledný automat $A/_{\equiv} = (K/_{\equiv}, \Sigma, [q_0]_{\equiv}, \delta_{\equiv}, F/_{\equiv})$, kde pre $\delta_{\equiv}$ platí: $[q]_{\equiv} \in \delta_{\equiv}([p]_{\equiv}, a) \Leftrightarrow q \in \delta(p, a)$.

Na redukciu počtu stavov môžeme použiť aj zľava invariantnú reláciu $\equiv_L$. Najprv k A zostrojíme automat A^r. Automat A^r je *obrátený* k $A = (K, \Sigma, \delta, q_0, F)$ a získame ho tak, že v A otočíme všetky prechody(zmeníme orientáciu hrán v grafe automatu) a vymeníme počiatočný stav s akceptačnými. Formálne teda $A^r = (K, \Sigma, \delta^r, F, \{q_0\})$, kde $q \in \delta^r(p, a) \Leftrightarrow p \in \delta(q, a)$. Do A^r môžeme potom pridať nový počiatočný stav a viesť z neho ε-prechody do stavov z F. Takto vzniknutý automat zbavíme štandardnou konštrukciou ε-prechodov a doplníme chýbajúce prechody, ktoré budú viesť do nového neakceptačného stavu. Triedy $\equiv_L$ získame tak, že na automate A^r vypočítame triedy relácie $\equiv$. Autori článku [18] v časti 4 popisujú ako efektívne používať tieto dve relácie na zmenšenie veľkosti NKA.

V článku [2] sa na minimalizáciu miesto ekvivalencií používa čiastočné usporiadanie na stavoch: $q \subseteq p \Leftrightarrow L(q) \subseteq L(p)$. Obe techniky možno použiť pre ľubovoľný NKA.

4.3 Konštrukcia NKA k regulárnemu výrazu

V praktických aplikáciách často potrebujeme zostrojiť konečný automat k danému regulárnemu výrazu. Aj v prípadoch, kde potrebujeme zostrojiť deterministický konečný automat postupujeme tak, že najskôr zostrojíme nedeterministický konečný automat. Jedno riešenie je zostrojiť NKA, ktorý používa ε-prechody. Takýto automat sa nazýva Thomsonov [27]. Zaujímavé sú však konštrukcie, ktoré dávajú NKA bez ε-prechodov, pretože sa s nimi vykonávajú ľahšie niektoré operácie, ako napríklad prienik jazykov alebo zisťovanie príslušnosti slova do jazyka. Štandardná konštrukcia pre zostrojenie ε-free NKA využíva pojem pozičného automatu.

4.3.1 Pozičný automat

Pozičný automat nezávisle objavili Glushkov [5] a McNaughton a Yamada [23].

Nech α je regulárny výraz. $\overline{\alpha}$ bude znamenať regulárny výraz, ktorý vznikol z α očíslovaním všetkých symbolov zo Σ v poradí v akom sa v α vyskytujú. Každé písmeno dostane unikátny symbol podľa svojej *pozície* v α. Množina pozícií pre výraz α je $pos(\alpha) = \{1, 2, ..., |\alpha|_\Sigma\}$, označme $pos_0(\alpha) = pos(\alpha) \cup \{0\}$. Napríklad nech $\alpha = (a+b)(abb+a)^*$, potom $\overline{\alpha} = (a_1+b_2)(a_3b_4b_5+a_6)^*$. Podobný spôsob zápisu sa používa aj pre odznačenie všetkých symbolov vo výraze, teda $\overline{\overline{\alpha}} = \alpha$.

Na zostrojenie automatu ďalej potrebujeme definovať nasledujúce tri zobrazenia. Pre regulárny výraz α a $i \in pos(\alpha)$ definujeme:

$$\begin{aligned} \texttt{first}(\alpha) &= \{i \mid a_i w \in L(\overline{\alpha})\} \\ \texttt{last}(\alpha) &= \{i \mid w a_i \in L(\overline{\alpha})\} \\ \texttt{follow}(\alpha, i) &= \{j \mid u a_i a_j v \in L(\overline{\alpha})\} \end{aligned}$$

Rozšírime zobrazenie `follow` na $\texttt{follow}(\alpha, 0) = \texttt{first}(\alpha)$.

Ďalej položme $\texttt{last}_0(\alpha) = \begin{cases} \texttt{last}(\alpha) & \text{ak } \varepsilon \notin L(\alpha) \\ \texttt{last}(\alpha) \cup \{0\} & \text{ak } \varepsilon \in L(\alpha) \end{cases}$

Pozičný automat pre regulárny výraz α je

$$A_{pos}(\alpha) = (pos_0(\alpha), \Sigma, \delta_{pos}, 0, \texttt{last}_0(\alpha))$$

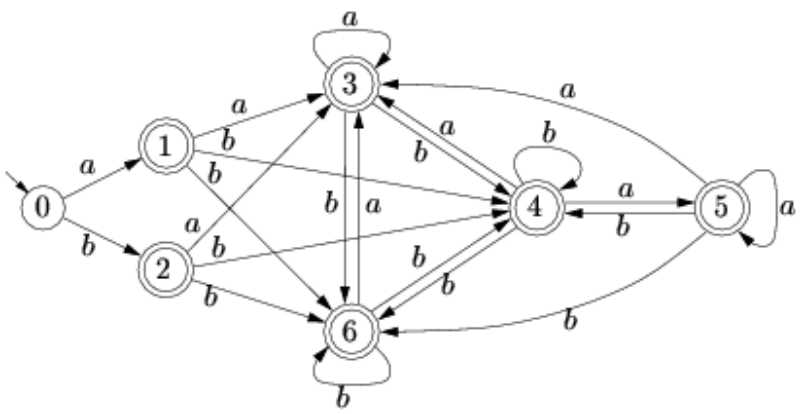
kde

$$j \in \delta_{pos}(i, a) \Leftrightarrow j \in \texttt{follow}(\alpha, i) \wedge a = \overline{a}_j$$

Na obrázku 4.3 je príklad pozičného automatu z [17].

$\text{first}(\alpha) = \{1, 2\}$
$\text{last}(\alpha) = \{1, 2, 3, 4, 5, 6\}$

i	follow(α, i)
1	$\{3, 4, 6\}$
2	$\{3, 4, 6\}$
3	$\{3, 4, 6\}$
4	$\{3, 4, 5, 6\}$
5	$\{3, 4, 5, 6\}$
6	$\{3, 4, 6\}$



Obr. 4.3: Pozičný automat $A_{pos}(\alpha)$ pre $\alpha = (a + b)(a^* + ba^* + b^*)^*$

V [5] a [23] je ukázané, že $L(\alpha) = L(A_{pos}(\alpha))$. Pozičný automat má vždy $|\alpha| + 1$ stavov a všetky prechody vchádzajúce do jedného stavu sú označené rovnakým symbolom. Tieto vlastnosti sa využívajú v algoritmoch na nájdenie špecifických reťazcov v texte, pretože tieto reťazce sa zadávajú pomocou regulárnych výrazov. $A_{pos}(\alpha)$ sa dá skonštruovať v kubickom čase pomocou induktívnych definícií zobrazení `first`, `last` a `follow`. V [1] je ukázané, ako vytvoriť tento automat v kvadratickom čase.

4.3.2 Malý NKA z regulárneho výrazu

V článku [11] je použitý iný prístup. Z regulárneho výrazu je priamo skonštruovaný automat, ktorý akceptuje jazyk definovaný daným výrazom. Autori pritom dokazujú, že vzniknutý automat je "malý". Ako miera zložitosti pre NKA je použitý počet prechodov, teda $|A| = |\delta|$. "Malý" automat budeme vnímať ako automat, ktorý ma málo prechodov, pričom o počet stavov sa nestaráme.

V predchádzajúcej časti sme sa oboznámili s pozičným automatom. Algoritmy prezentované v [11] je možné vnímať ako úpravu pozičného automatu. Najprv pre každý jeho stav vytvoríme niekoľko kópií. Každá kópia stavu bude mať rovnaké vchádzajúce prechody ako pôvodný stav. Vychádzajúce prechody pôvodného stavu budú medzi neho a jeho kópie rozdelené. Potom spojíme stavy s rovnakými prechodmi vedúce do rovnakých stavov.

Pozičný automat ale explicitne zostrojovať nemusíme. V skutočnosti budeme upravovať zobrazenie `follow`. Predpokladajme že máme zadaný výraz α a poznáme zobrazenia `first`, `follow` a `last`. Pre každé $i \in pos(\alpha)$ sa snažíme rozdeliť množinu $\texttt{follow}(\alpha, i)$ na niekoľko podmnožín. Tieto podmnožiny budú reprezentovať stavy nového automatu. Každá podmnožina C bude zodpovedná za prechody z pôvodného stavu do prvkov množiny C. Teda ak $\texttt{follow}(\alpha, i) = C_1 \cup C_2 \cup ... \cup C_k$ a pozičný automat bol v stave i potom nový automat bude v jednom zo stavov, ktoré reprezentujú množiny C_j. Nech v pozičnom automate vedie prechod z i do $i' \in C_l$ a $\texttt{follow}(\alpha, i') = C'_1 \cup C'_2 \cup ... \cup C'_{k'}$. Potom v novom automate bude viesť prechod na rovnaký symbol z C_l do C'_j, pre všetky $1 \leq j \leq k'$.

Jedna podmnožina C sa môže vyskytovať v niekoľkých dekompozíciách rôznych `follow` množín. Preto autori článku hovoria o "spoločných" množinách. Formálne teraz zadefinujeme takýto systém aj pojem dekompozície.

Definícia 4.3.1. *Nech α je regulárny výraz a $pos(\alpha)$ je množina jeho pozícií. Systém spoločných* `follow` *množín (system of common follow sets, skrátene systém CFS) pre α je daný dekompozíciou $dec(i) \subseteq 2^{pos(\alpha)}$ pre každé $i \in pos(\alpha)$. Pričom pre každé $i \in pos(\alpha)$ musí platiť*

$$dec(i) \neq \emptyset \text{ a } \texttt{follow}(\alpha, i) = \bigcup_{C \in dec(i)} C.$$

Skupina spoločných `follow` *množín (skupina CFS) $\mathcal{C}$ priradená k tomuto systému je*

$$\mathcal{C} = \{\texttt{first}(\alpha)\} \cup \bigcup_{i \in pos(\alpha)} dec(i).$$

Automat A priradený k tomuto systému (CFS automat) je $A = (K, \Sigma, \delta, q_0, F)$ kde $K = \mathcal{C} \times \{0, 1\}$, $q_0 = \begin{cases} (\texttt{first}(\alpha), 1) & ak \quad \varepsilon \in L(\alpha) \\ (\texttt{first}(\alpha), 0) & ak \quad \varepsilon \notin L(\alpha) \end{cases}$, $F = \mathcal{C} \times \{1\}$ a

$$(C', f_i) \in \delta((C, f), \langle\alpha\rangle_i) \qquad \forall i \in C, f \in \{0, 1\}, C' \in dec(i),$$

kde $f_i = 1$ ak $i \in \texttt{last}(\alpha)$ a $f_i = 0$ ak $i \notin \texttt{last}(\alpha)$.

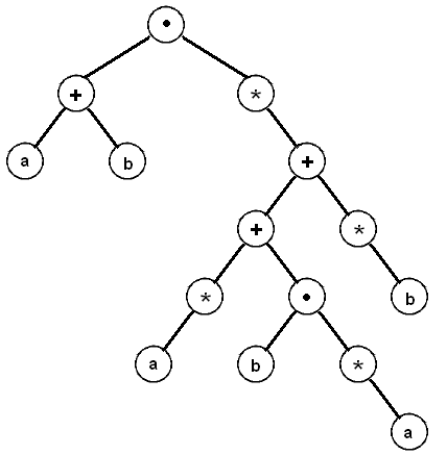
V článku [11] je ukázané, že CFS automat pre daný výraz α akceptuje práve jazyk daný výrazom α. Zložitosť (v tomto prípade počet prechodov) CFS automatu závisí od CFS systému, na ktorom je automat postavený. Naša snaha smeruje k nájdeniu takého CFS systému, ktorý dáva malý automat.

Regulárny výraz ako strom

Regulárny výraz môžeme reprezentovať stromom. Listy stromu budú symboly vyskytujúce sa vo výraze. Vnútorné uzly budú obsahovať operácie, ktoré sa vo výraze vyskytujú. Jeden vrchol (list alebo vnútorný uzol) reprezentuje podvýraz. Pojmy *vrchol* a *podvýraz* budeme teda používať v rovnakom význame.

Nech α je regulárny výraz, jemu prislúchajúci strom budeme označovať t_α. Nech β a γ sú podvýrazy α. Ak koreň t_β je otec koreňa t_γ, značíme $root(t_\beta) \prec root(t_\gamma)$ (alebo tiež $\beta \prec \gamma$), potom γ je podvýrazom výrazu β.

Keď máme strom t_α, jeho podstromom budeme rozumieť podgraf, ktorý je strom. Teda ak t je podstrom t_α, t môže byť aj jednoduchá cesta. Pre podstrom t budeme značiť ako $pos(t)$ pozície, ktoré sa v podstrome vyskytujú, t.j. pozície listov stromu t v celkovom výraze. Ak koreň t je podvýraz β potom $pos(t) \subseteq pos(\beta)$ (rovnosť nastáva ak t je celý podstrom pod β).



Obr. 4.4: Strom reprezentujúci výraz $\alpha = (a + b)(a^* + ba^* + b^*)^*$. Poziční automat a zobrazenia follow, first a last pre rovnaký výraz sú na obrázku 4.3.

Teraz uvedieme induktívne definície zobrazení `first`, `last` a `follow`, aby si čitateľ mohol urobiť predstavu, ako tieto zobrazenia súvisia so stromovou reprezentáciou regulárneho výrazu.

$$\begin{aligned} \texttt{first}(\emptyset) &= \emptyset \\ \texttt{first}(\varepsilon) &= \emptyset \\ \texttt{first}(a) &= pos(a) \qquad \text{pre } a \in \Sigma \\ \texttt{first}(\beta + \gamma) &= \texttt{first}(\beta) \cup \texttt{first}(\gamma) \end{aligned}$$

Pravidlá pre `last` získame tak, že v horeuvedených pravidlách vymeníme `first` za `last`. Ďalej platí

$$\texttt{first}(\beta\gamma) = \begin{cases} \texttt{first}(\beta) & \text{ak } \varepsilon \notin L(\beta) \\ \texttt{first}(\beta) \cup \texttt{first}(\gamma) & \text{ak } \varepsilon \in L(\beta) \end{cases}$$

$$\texttt{last}(\beta\gamma) = \begin{cases} \texttt{last}(\gamma) & \text{ak } \varepsilon \notin L(\gamma) \\ \texttt{last}(\beta) \cup \texttt{last}(\gamma) & \text{ak } \varepsilon \in L(\gamma) \end{cases}$$

$$\texttt{follow}(a, i) = \emptyset \quad \text{pre } a \in \Sigma$$

$$\texttt{follow}(\beta + \gamma, i) = \begin{cases} \texttt{follow}(\beta, i) & \text{ak } i \in pos(\beta) \\ \texttt{follow}(\gamma, i) & \text{ak } i \in pos(\gamma) \end{cases}$$

$$\texttt{follow}(\beta\gamma, i) = \begin{cases} \texttt{follow}(\beta, i) & \text{ak } i \in pos(\beta) - \texttt{last}(\beta) \\ \texttt{follow}(\beta, i) \cup \texttt{first}(\gamma) & \text{ak } i \in \texttt{last}(\beta) \\ \texttt{follow}(\gamma, i) & \text{ak } i \in pos(\gamma) \end{cases}$$

$$\texttt{follow}(\beta^*, i) = \begin{cases} \texttt{follow}(\beta, i) & \text{ak } i \in pos(\beta) - \texttt{last}(\beta) \\ \texttt{follow}(\beta, i) \cup \texttt{first}(\beta) & \text{ak } i \in \texttt{last}(\beta) \end{cases}$$

Stromová reprezentácia nám pomôže pri získavaní $\texttt{follow}(\alpha, i)$ tak, že budeme prechádzať v strome t_α od listu i až ku koreňu a budeme za istých okolností, ak $i \in \texttt{last}(\beta)$ pre nejaký podvýraz β, pridávať $\texttt{first}(\beta)$ do `follow` množiny pre i. Definujme funkciu `next`, aby sme mohli pridávanie prvkov do `follow` množiny formalizovať. Pre podvýraz β výrazu α položme

$$\texttt{next}(\beta) = \begin{cases} \beta & \text{ak } \beta \text{ je syn podvýrazu } \beta^* \text{ v } t_\alpha \\ \gamma & \text{ak } \beta \text{ je syn podvýrazu } \beta\gamma \text{ v } t_\alpha \\ \bullet & \text{inak} \end{cases}$$

Položme $\texttt{first}(\bullet) = \emptyset$ aby sme získali nasledujúce tvrdenie, časť 1.

Tvrdenie 4.3.1. *Nech α je regulárny výraz a $i \in pos(\alpha)$. Predpokladajme, že výraz α je reprezentovaný stromom t_α a že β a γ sú podvýrazy vyskytujúce sa ako vrcholy v t_α. Potom platí:*

1. $\texttt{follow}(\beta, i) = \bigcup_{\substack{\beta' \succ \beta \\ i \in last(\beta')}} \texttt{first}(\texttt{next}(\beta'))$.
2. *Ak $\beta \prec \gamma$, potom $\texttt{first}(\texttt{next}(\gamma)) \subseteq pos(\beta)$.*
3. *Ak $\beta \preceq \gamma$, potom $\texttt{last}(\beta) \cap pos(\gamma) = \texttt{last}(\gamma)$ alebo $\texttt{last}(\beta) \cap pos(\gamma) = \emptyset$. Rovnaké tvrdenie platí aj pre `first` miesto `last`.*

Takto sme získali popis $\texttt{follow}(\beta, i)$, ktorý môže byť rozšírený na ľubovoľný podstrom t stromu t_α nasledovne

$$\texttt{follow}(t, i) = pos(t) \cap \bigcup_{\substack{\beta' \succ root(t) \\ i \in last(\beta')}} \texttt{first}(\texttt{next}(\beta')).$$

Odtiaľ máme $\texttt{follow}(t_\beta, i) = \texttt{follow}(\beta, i)$ pre výraz β a $i \in pos(\beta)$.

Nájdenie CFS automatu

V [11] je prezentovaných niekoľko spôsobov ako vypočítať systém CFS, ktorý dáva malý CFS automat. Popíšeme teraz najjednoduchší z nich. Idea spočíva v rekurzívnom počítaní množín $\mathcal{C}(t) \subseteq 2^{pos(t)}$ a dekompozícií $dec(i,t)$, pre niektoré podstromy t stromu t_α, pričom pre nejaké $i \in pos(t)$ musí platiť:

$$dec(i,t) \subseteq \mathcal{C}(t) \text{ a } \texttt{follow}(t,i) = \bigcup_{C \in dec(i,t)} C$$

Ak sú tieto dve podmienky splnené, hovoríme, že dekompozícia dec je *vhodná* pre t a $i \in pos(t)$. Pre celý strom t_α potom podľa tvrdenia 4.3.1 dostávame

$$\texttt{follow}(\alpha,i) = \texttt{follow}(t_\alpha,i) = \bigcup_{C \in dec(i,t_\alpha)} C \qquad \text{pre } i \in pos(\alpha).$$

Keď sa nám podarí nájsť $\mathcal{C}(t_\alpha)$ a $dec(i,t_\alpha)$, dostaneme takýto CFS systém

$$\mathcal{C} = \{\texttt{first}(\alpha)\} \cup \mathcal{C}(t_\alpha) \text{ a } dec(i) = dec(i,t_\alpha) \qquad \text{pre } i \in pos(\alpha).$$

Dekompozíciu $dec(i,t_\alpha)$ vhodnú pre všetky $i \in pos(t_\alpha)$ budeme hľadať induktívne.

1. Pre jednoduchý strom t, ktorý obsahuje jediný list s pozíciou i je jednoduché nájsť vhodnú dekompozíciu. Jedna z možností je položiť

$$dec(i,t) = \begin{cases} \{\{i\}\} & \text{ak } \exists \gamma \in t, \gamma \neq root(t) : i \in \texttt{last}(\gamma) \cap \texttt{first}(\texttt{next}(\gamma)) \\ \emptyset & \text{inak} \end{cases}$$

2. Predpokladajme teda, že pre strom t s koreňom β platí $|t| > 1$. Nech t_1 je podstrom stromu t pod nejakým vrcholom β_1 a t_2 je zvyšok stromu t po odstránení t_1. Ak $i \in pos(t_1)$ a dec je vhodná pre t_1 a i, potom

$$dec(i,t) = \begin{cases} dec(i,t_1) \cup C_1 & \text{ak } i \in \texttt{last}(\beta_1) \\ dec(i,t_1) & \text{inak,} \end{cases}$$

kde

$$C_1 = pos(t) \cap \bigcup_{\substack{\beta \prec \gamma \preceq \beta_1 \\ last(\gamma) \cap pos(\beta_1) = last(\beta_1)}} \texttt{first}(\texttt{next}(\gamma))$$

je vhodná pre t a i. Podobne $i \in pos(t_2)$ a dec je vhodná pre t_2 a i, môžeme položiť

$$dec(i,t) = \begin{cases} dec(i,t_2) \cup C_2 & \text{ak } \texttt{first}(\beta_1) \subseteq \texttt{follow}(\alpha,i) \\ dec(i,t_2) & \text{inak,} \end{cases}$$

kde

$$C_2 = pos(t) \cap \texttt{first}(\beta_1)$$

a získať tak vhodnú dekompozíciu pre t a i.

Nasledujúci algoritmus, pozostávajúci z krokov 1 až 5, dostane ako vstup podstrom t stromu t_α a vypočíta $dec(i,t)$ pre každé $i \in pos(t)$.

1. Ak $|t| = 1$, vypočítame dekompozíciu podľa prípadu 1 vyššie.
2. Ak $|t| > 1$, vykonáme nasledujúce kroky.
3. Od vrcholu $root(t)$ smerom dole budeme hľadať vrchol β_1 a jemu prislúchajúci strom t_1 tak, aby platilo $\frac{|t|}{3} \leq |t_1| \leq \frac{2|t|}{3}$.
4. Rekurzívne voláme tento algoritmus pre stromy t_1 a t_2, kde t_2 je zvyšok stromu t po odstránení t_1.
5. Pre každé $i \in pos(t)$ získame $dec(i,t)$ podľa prípadu 2 vyššie.

Autori článku [11] dokazujú, že vyššie uvedený algoritmus vypočíta taký systém CFS, že CFS automat zostrojený k tomuto systému podľa definície 4.3.1 má najviac $6n - 2$ stavov a približne

$$\frac{6}{(\log_2 3/2)^2} n(\log n)^2$$

prechodov. Ďalej uvádzajú ako zlepšiť tento algoritmus tak, aby výsledný automat mal najviac $2n - 1$ stavov a

$$\frac{4}{(\log_2 3/2)^2} n(\log_2 n)^2$$

prechodov.

Hagenah a Muscholl [6] ukázali implementáciu tohto algoritmu, ktorá beží v čase $\mathcal{O}(n(\log n)^2)$.

Kapitola 5

Miery nedeterminizmu

Čitateľ, ktorý má prehľad v teórii jazykov, určite vie, že pre niektoré výpočtové modely pridanie nedeterminizmu znamená zvýšenie výpočtovej (akceptačnej) sily. Takým modelom sú napríklad zásobníkové automaty. Často sa tiež stretneme s tým, že nedeterminizmus pomáha znížiť výpočtovú zložitosť. Jeden z najznámejších otvorených problémov teórie jazykov a automatov je, či pridanie nedeterminizmu do turingových strojov pomáha znížiť čas výpočtu viac ako polynomiálne. Nedeterminizmus môže tiež pomôcť znížiť pamäťové nároky na výpočet (t.j. ak sme v deterministickom modeli potrebovali pre vstupy dĺžky n pamäť $S(n)$, pridanie nedeterminizmu môže spôsobiť, že nám bude stačiť $o(S(n))$ pamäte).

Už pri zavedení nedeterminizmu do konečných automatov Rabin a Scott ukázali, že to nezmení ich výpočtovú silu. Keďže čas výpočtu DKA na slove w je zhora ohraničený jeho dĺžkou, nedeterminizmus v tejto oblasti nemôže pomôcť. V konečných automatoch použitá pamäť pre slovo w dokonca ani nezáleží od jeho dĺžky (je vždy konštantná), takže ani na tento zložitostný aspekt nemá nedeterminizmus v KA vplyv.

V kapitole 4 sme už spomínali, že existujú regulárne jazyky, ktoré sa dajú akceptovať NKA s n stavmi, ale ich minimálne DKA majú 2^n stavov. Toto je presne to miesto, na ktorom nám nedetrminizmus pomáha. Jeho sila sa prejavila v popisnej zložitosti. Aby sme mohli určiť, čo spôsobuje tento dramatický (t.j. exponenciálny) rozdiel, mali by sme zistiť vzťah medzi množstvom nedeterminizmu a počtom stavov. Najskôr však potrebujeme vedieť ako budeme nedeterminizmus v KA merať.

Za veľkosť automatu A budeme považovať počet jeho stavov a budeme používať označenie $|A|$. Pre každý regulárny jazyk L budeme ako $s(L)$ označovať veľkosť minimálneho DKA pre L a podobne $ns(L)$ bude označovať veľkosť minimálneho NKA pre L.

Pre NKA A a vstup x bude $T_{A,x}$ označovať strom všetkých výpočtov A na x. *Nejednoznačnosť* A na x je počet akceptačných výpočtov A na x (v strome $T_{A,x}$ je to počet akceptačných listov). Ak nejednoznačnosť A je 1 pre všetky $x(\in L(A))$, potom hovoríme, že A je *jednoznačný*. Ako $uns(L)$ (unambigous nondeterministic size) označíme veľkosť minimálneho jednoznačného NKA(UNKA), ktorý akceptuje L. Ak má A nejednoznačnosť najviac k pre všetky vstupy, potom ho voláme *k-nejednoznačný* a ako $ns_k(L)$ označujeme veľkosť minimálneho k-nejednoznačného NKA, ktorý akceptuje L.

Stupeň nedeterminizmu meriame nasledovne. Pre každý vstup $x \in \Sigma^*$ a pre každý výpočet C automatu A na x definujeme $advice(C)$ ako počet nedeterministických rozhodnutí počas výpočtu C. V strome $T_{A,x}$ je to počet vrcholov na ceste C, ktoré majú viac ako jedného syna. Ďalej

$$advice_A(x) = \max\{advice(C) \mid C \text{ je výpočet automatu } A \text{ na } x\}$$

$$\text{a } advice_A(n) = \max\{advice_A(n) \mid x \in \Sigma^n\}.$$

Pre každé $x \in \Sigma^*$ je $leaf_A(x)$ počet listov v strome $T_{A,x}$ a

$$leaf_A(n) = \max\{leaf_A(x) \mid x \in \Sigma^n\}.$$

Stupeň nejednoznačnosti $ambig_A(n) = \max\{ambig_A(x) \mid x \in \Sigma^{\leq n}\}$, kde $ambig_A(x)$ je počet akceptačných listov v strome $T_{A,x}$.

5.1 Vzťahy medzi mierami

V tejto kapitole sa opierame o prácu [10]. Z nej čerpáme aj nasledujúce výsledky.

Tvrdenie 5.1.1. *Pre každý NKA A platí jeden z nasledujúcich vzťahov:*

1. $advice_A(n) \leq |A|$ *a* $leaf_A(n) \leq |A|^{|A|}$
2. $advice_A(n) \geq \frac{n}{|A|} - 1$ *a* $leaf_A(n) \geq \frac{n}{|A|} - 1$

Dôkaz. Druhý prípad nastáva ak v A je dosiahnuteľný stav q, ktorý leží v cykle a má dva vychádzajúce prechody na rovnaký symbol, pričom jeden z nich smeruje do cyklu. Potom existuje vstup dĺžky n, ktorého výpočet bude týmto cyklom a teda aj stavom q prechádzať. Ak taký stav v A nie je, potom každý stav s nedeterministickým rozhodnutím je prejdený najviac raz pre každé slovo teda platí vzťah 1. □

Stav q nazývame *absolútne zamietací*, ak $L(q) = \emptyset$. Teda ak sa raz výpočet ocitne v tomto stave, už nemôže nastať akceptovanie.

Tvrdenie 5.1.2. *Pre každý NKA A s najviac jedným absolútne zamietacím stavom platí*

$$leaf_A(x) \leq ambig_A(|x| + |A|).|x|.|A| + 1 \text{ pre každé } x.$$

Tvrdenie 5.1.3. *Pre každý NKA A s najviac jedným absolútne zamietacím stavom platí*

$$advice_A(n), ambig_A(n) \leq leaf_A(n) \leq \mathcal{O}(ambig_A(n).advice_A(n)).$$

Pre každý taký UNKA platí $advice_A(n) = \Theta(leaf_A(n))$.

Kapitola 6

Záver

V práci sme priniesli výsledky o popisnej zložitosti konečných automatov, ktoré majú tak praktický, ako aj teoretický význam. Čitateľ sa dozvedel o niektorých problémoch, ktoré sa riešia pomocou konečných automatov.

Medzi najzákladnejšie a najstaršie problémy z oblasti zložitosti konečných automatov patrí hľadanie minimálneho DKA. Algoritmus s najlepšou známou časovou zložitosťou, riešiaci tento problém [7], sme priniesli v časti 3.1. Deterministickými automatmi sme v ďalšej časti implementovali slovníky, t.j. ukázali sme spôsob ako nájsť acyklický automat pre konečnú množinu slov.

V kapitole 4 sme sa zaoberali problémami, ktoré súvisia s regulárnymi výrazmi. Najprv sme uviedli známe výsledky o minimalizácii NKA spolu s algoritmom na čiastočnú minimalizáciu. Potom sme ukázali, ako k regulárnemu výrazu nájsť malý NKA pomocou pozičného automatu.

Poslednú časť práce sme venovali mieram nedeterminizmu. Naznačili sme vzťahy medzi množstvom nedeterminizmu a počtom stavov.

V oblasti zložitosti konečných automatov je stále veľa otvorených problémov, ktoré by si zaslúžili viac pozornosti. Niektoré z nich je možné nájsť v [9].

Literatúra

[1] Anne Bruggemann-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.

[2] J.-M. Champarnaud and F. Coulon. Nfa reduction algorithms by means of regular inequalities. *Theor. Comput. Sci.*, 327(3):241–253, 2004.

[3] Jan Daciuk. Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings. *Theor. Comput. Sci.*, 2608:127–152, 2004.

[4] Jan Daciuk, Bruce W. Watson, Stoyan Mihov, and Richard E. Watson. Incremental construction of minimal acyclic finite-state automata. *Comput. Linguist.*, 26(1):3–16, 2000.

[5] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

[6] Christian Hagenah and Anca Muscholl. Computing ε-free nfa from regular expressions in o(n log(n)) time. In *Mathematical Foundations of Computer Science*, pages 277–285, 1998.

[7] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

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

[9] Juraj Hromkovič. Descriptional complexity of finite automata: concepts and open problems. *J. Autom. Lang. Comb.*, 7(4):519–531, 2002.

[10] Juraj Hromkovič, Juhani Karhumäki, Hartmut Klauck, Georg Schnitger, and Sebastian Seibert. Measures of nondeterminism in finite automata. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 199–210, London, UK, 2000. Springer-Verlag.

[11] Juraj Hromkovič, Sebastian Seibert, and Thomas Wilke. Translating regular expressions into small ε-free nondeterministic finite automata. *J. Comput. Syst. Sci.*, 62(4):565–588, 2001.

[12] Lucian Ilie and Sheng Yu. Algorithms for computing small nfas. In *MFCS '02: Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science*, pages 328–340, London, UK, 2002. Springer-Verlag.

[13] Lucian Ilie and Sheng Yu. Reducing nfas by invariant equivalences. *Theor. Comput. Sci.*, 306(1-3):373–390, 2003.

[14] O. Carton J. Berstel. On the complexity of hopcrofts state minimization algorithm. In *Lecture Notes in Computer Science*, volume 3317, pages 35–44. Springer-Verlag, 2005.

[15] Tao Jiang and B. Ravikumar. Minimal nfa problems are hard. *SIAM J. Comput.*, 22(6):1117–1141, 1993.

[16] S. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, Princeton, N.J., 1956.

[17] S. Yu L. Ilie, G. Navarro. On nfa reductions. In G. Paun J. Karhumäki, H. Maurer and G. Rozenberg, editors, *Theory Is Forever*, volume 3113 of *Lecture Notes in Computer Science*, pages 112 – 124, Berlin, Heidelberg, 2004. Springer-Verlag.

[18] S. Yu L. Ilie, R. Solis-Oba. Reducing the size of nfas by using equivalences and preorders. In *Combinatorial Pattern Matching, Lecture Notes in Computer Science*, pages 310 – 321. Springer Berlin / Heidelberg, 2005.

[19] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[20] F. R. Moore. On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. *IEEE Trans. Comput.*, 20:1211 – 1214, 1971.

[21] J. Myhill. Finite automata and the representation of events. Technical Report WADD TR-57-624, Wright Patterson Air Force Base, Ohio, 1957.

[22] A. Nerode. Linear automaton transformations. *Proc. Amer. Math. Soc.*, 9:541–544, 1958.

[23] H Yamada R McNaughton. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–47, 1960.

[24] R. E. Tarjan R. Paige and R. Bonic. A linear time solution to the single function coarsest partition problem. *Theor. Comput. Sci.*, 40(1):67–84, 1985.

[25] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Develop.*, 3:115–125, 1959.

[26] Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theor. Comput. Sci.*, 92(1):181–189, 1992.

[27] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[28] B. W. Watson. A fast and simple algorithm for constructing minimal acyclic deterministic finite automata. *Journal of Universal Computer Science*, 8(2):363–367, 2002.