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

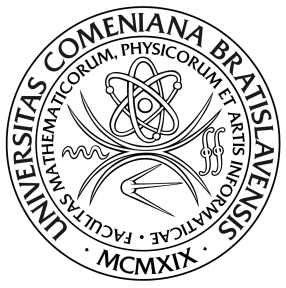
Vzdialenosti jazykov

(Bakalárska práca)

Michal Nánási

Vedúci: Prof. RNDr. Branislav Rovan, PhD.

Bratislava, 2008



Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského v Bratislave

Vzdialenosti jazykov

(bakalárska práca)

Michal Nánási

Študijný odbor: Informatika 9.2.1

Vedúci: prof. RNDr. Branislav Rovan, PhD.

Bratislava, 2008

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

.....

Poďakovanie

Chcel by som poďakovať svojmu vedúcemu bakalárskej práce profesorovi Branislavovi Rovanovi pomoc pri výbere témy práce a za cenné rady pri písaní práce.

Tiež by som chcel poďakovať svojej rodine za podporu, obzvlášť svojej mame za cennú recenziu.

Nakoniec by som chcel poďakovať svojej priateľke za to, že to so mnou vydržala.

Abstrakt

Autor:	Michal Nánási
Názov bakalárskej práce:	Vzdialenosti jazykov
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci bakalárskej práce:	Prof. RNDr. Branislav Rovan, PhD
Rozsah práce:	39 strán

Bratislava, jún 2008

Práca sa zaoberá algoritmami na vypočítanie vzdialeností a podobností slov a jazykov. Venuje sa editačnej vzdialenosti, Δ-podobnosti a ľubovoľnej vzdialenosti, ktorá môže byť definovaná a-prekladačom (prekladačová vzdialenosť). Prezentuje doteraz známe algoritmy na vypočítanie editačnej a prekladačovej vzdialenosti dvoch regulárnych jazykov. Prináša algoritmy na vypočítanie editačnej a prekladačovej vzdialenosti regulárneho jazyka od bezkontextového jazyka a algoritmus na vypočítanie Δ-podobnosti dvoch regulárnych jazykov. V práci je dokázané, že pre väčšinu tried Chomského hierarchie jazykov neexistuje algoritmus, ktorý vypočíta vzdialenosť dvoch jazykov z danej triedy.

Kľučové slová: Editačná vzdialenosť, Δ-podobnosť, vzdialenosť jazykov.

Predhovor

Vzdialenosti definované na slovách majú široké uplatnenie. Najjednoduchším príkladom je kontrola pravopisu implementovaná v rôznych textových editoroch, e-mailových klientoch či internetových prehliadačoch. V biológii, obzvlášť v genetike, sa porovnávajú DNA, či RNA reťazce, kde sa k slovu opäť dostávajú vzdialenosti slov. Vedieť porovnávať reťazce je tiež užitočné pri odhaľovaní plagiátorstva. Pri všetkých týchto aplikáciach dosiahneme zlepšenie výkonu, ak nebudeme porovnávať len jednotlivé reťazce, ale celé jazyky. Vzdialenosti jazykov majú tiež uplatnenie v rozpoznávaní reči.

Cieľom tejto bakalárskej práce je skúmať vzdialenosti definované na jazykoch a algoritmy na ich vypočítanie. Zaoberá sa algoritmami na vypočítanie vzdialeností jazykov z jednotlivých tried a dvojíc tried z Chomského hierarchie jazykov. Práca prináša algoritmy na vypočítanie editačnej a prekladačovej vzdialenosti regulárneho jazyka od bezkontextového jazyka a algoritmy na vypočítanie Δ-podobnosti dvoch regulárnych jazykov.

Obsah

Zoznam tabuliek

Zoznam obrázkov

Kapitola 1

Úvod

Podobne, ako na množine reálnych čísiel, tak aj na množine slov si môžeme definovať vzdialenosť. Vzdialenosť dvoch slov nám vlastne hovorí, ako sú dve slová podobné. Pomôže nám to vyriešiť napríklad nasledovný problém.

Problém 1.0.1 *Užívateľ píše v textovom editore text a robí preklepy. Našou úlohou je napísať program, ktorý mu bude chyby opravovať. Predpokladáme, že korektné slová sú tie, ktoré patria do jazyka L (L je konečný jazyk).*

Riešenie. V prvom rade musíme mať zadefinovanú nejakú vzdialenosť, podľa ktorej budeme porovnávať rôznosť slov. Keďže užívateľ píše na klávesnici, tak najčastejšie chyby, aké môžu nastať sú nenapísanie nejakého písmena, napísanie nejakého písmena namiesto iného písmena, a stlačenie takej klávesy, ktorú sme nechceli prípadne vymenenia poradia dvoch susedných písmen. Za vzdialenosť slov si zoberieme minimálny počet chýb, ktorý bol nutný spraviť tak, aby sme namiesto jedného slova napísali to druhé. V prípade, že by sme chceli byť ešte dôslednejší, tak by sme mohli chyby ohodnotiť. Pomýlenie si písmen „a" a „s" je totiž pravdepodobnejšie ako pomýlenie si písmen „a" a „p", keďže „a" a „s" sú na klávesnici k sebe fyzicky oveľa bližšie.

Následne sa pokúsime nájsť také slová $u \in L$, že ich vzdialenosť je najbližšie k zadanému slovu a necháme si užívateľa vybrať, ktoré slovo myslel.

Toto riešenie je efektívne (za predpokladu, že najbližšie slovo nehľadáme prezeraním všetkých slov, ale nejako efektívnejšie) a je viac menej používané. Moderné kontroly pravopisu (ako napríklad Aspell[Asp]) robia skoro to isté.

Vedieť zistiť, či sú nejaké slová podobné, nám dáva možnosť riešiť problémy aj v iných oblastiach, ako napríklad v biológií (porovnávanie DNA a RNA reťazcov), prípadne pri odhaľovaní plagiátorstva.

Pri náročnejších aplikáciach, ako napríklad rozoznávanie reči[MPR02] sa dostávame k problému, že máme dva jazyky (väčšinou regulárne) a chceme nájsť takú dvojicu slov z tých jazykov, že ich vzdialenosť je minimálna.

Na konci tejto kapitoly sa venujeme základným definíciam a neexistencii algoritmov na vypočítanie vzdialeností jazykov z niektorých tried Chomského hierarchie. V kapitole 2

sa venujeme algoritmom na vypočítanie editačnej vzdialenosti. V kapitole 3 sa venujeme zovšeobecneniu editačnej vzdialenosti pomocou a prekladača. V oboch kapitolách popisujeme známe algoritmy na výpočet vzdialenosti dvoch regulárnych jazykov a predkladáme vlastný algoritmus na vypočítanie vzdialenosti regulárneho jazyka od bezkontextového jazyka. Kapitola 4 sa venuje Δ-podobnosti a predstavuje dva vlastné algoritmy na vypočítanie Δ-podobnosti dvoch regulárnych jazykov.

1.1 Použité označenia

Predpokladáme, že čitateľ pozná pojmy ako abeceda, slovo, jazyk a pozná jednotlivé triedy Chomského hierarchie jazykov. Uvedieme len označenia, ktoré budeme používať. Označenia sme prebral z [JH79].

- Σ - konečná abeceda abeceda
- ε - prázdne slovo
- $A = (K, \Sigma, \delta, q_0, F)$ je nedeterministický konečný automat.
- $T = (K, \Sigma_1, \Sigma_2, H, q_0, F)$ je a-prekladač, kde H je množina pravidiel.
- $G = (N, T, P, \sigma)$ - G je gramatika, N je množina neterminálov, T je množina terminálov, P je množina pravidiel a $\sigma \in N$ je počiatočný symbol.
- Pre krok výpočtu automatov používam symbol $\vdash$ a pre krok odvodenia gramatiky používam symbol $\Rightarrow$.

1.2 Vzdialenosti a podobnosti

Najskôr si zadefinujeme, čo je to vzdialenosť a podobnosť.

Definícia 1.2.1 *Nech M je ľubovoľná množina, nech $u, v, w \in M$. Nech $s : M \times M \to \mathbb{R} \cup \{\infty\}$ je funkcia spĺňajúca nasledovné podmienky:*

1. $d(u, v) \geq 0$
2. $d(u, v) = 0 \Leftrightarrow u = v$ *(reflexívnosť)*
3. $d(u, v) = d(v, u)$ *(symetria)*
4. $d(u, v) + d(v, w) \geq d(u, w)$ *(trojuholníková nerovnosť)*

Potom funkciu d nazývame vzdialenosťou na množine M.

Definícia 1.2.2 *Nech M je ľubovoľná množina, nech $u, v \in M$. Nech $s : M \times M \to \mathbb{R}$ je funkcia spĺňajúca nasledovné podmienky:*

1. $0 \leq s(u,v) \leq 1$
2. $s(u,v) = 1 \Leftrightarrow u = v$ *(reflexívnosť)*
3. $s(u,v) = s(v,u)$ *(symetria)*

Potom funkciu s nazývame podobnosťou na množine M.

Definícia 1.2.3 *Nech Σ je konečná abeceda a d je vzdialenosť definovaná na Σ^*. Nech $L_1, L_2 \subseteq \Sigma^*$. Potom vzdialenosť jazykov L_1, L_2 definujeme nasledovne:*

$$d(L_1, L_2) = \inf\{d(u,v) | u \in L_1, v \in L_2\}$$

Definícia 1.2.4 *Nech Σ je konečná abeceda a s je podobnosť definovaná na Σ^*. Nech $L_1, L_2 \subseteq \Sigma^*$. Potom podobnosť jazykov L_1, L_2 definujeme nasledovne:*

$$s(L_1, L_2) = \sup\{s(u,v) | u \in L_1, v \in L_2\}$$

Ak to pôjde, tak namiesto inf budeme používať minimum a namiesto sup budeme používať maximum. Vzdialenosti dvoch jazykov by sa dali definovať aj ináč a tak, aby záviseli od viac ako dvoch slov, avšak takéto vzdialenosti sa v praxi zatiaľ vôbec nepoužívajú a preto sa im nebudeme venovať.

Prvú vec, ktorú si treba priznať je, že takto definované vzdialenosti (podobnosti) nie sú vzdialenosťami (podobnosťami) nad Σ^*, keďže nespĺňajú hneď druhú podmienku definície vzdialenosti (podobnosti). Ako príklad pre podobnosť aj vzdialenosť uvedieme jazyky $L_1 = \{a, b\}$ a $L_2 = \{a, c\}$. Napriek tomu, v praxi majú význam a preto sa nimi budeme zaoberať. Skôr ako sa budeme zaoberať algoritmami na vypočítanie vzdialeností jazykov si zistíme, čo sa už nedá vypočítať.

Veta 1.2.5 *Neexistuje algoritmus, ktorý pre ľubovoľné dva bezkontextové jazyky vypočíta ich vzdialenosť.*

Dôkaz. Ak by sme vedeli nájsť vzdialenosť dvoch bezkontextových jazykov, tak by sme vedeli rozhodovať či ich prienik je prázdny, lebo vzdialenosť dvoch jazykov je nula práve vtedy, keď obsahujú rovnaké slovo. Dôkaz toho, že problém prázdnosti prieniku dvoch bezkontextových jazykov je nerozhodnuteľný, sa dá nájsť v [JH79]. □

Veta 1.2.6 *Neexistuje algoritmus, ktorý vypočíta vzdialenosť regulárneho jazyka od kontextového jazyka.*

Dôkaz. Dôkaz je podobný, ibaže v tomto prípade by sme vedeli rozhodnúť, či je kontextový jazyk prázdny. Stačí zobrať za regulárny jazyk jazyk Σ^*. Dôkaz toho, že problém prázdnosti kontextového jazyka je nerozhodnuteľný, sa nachádza v [JH79]. □

Z posledných dvoch tvrdení môžeme vyvodiť záver, že nemá zmysel hľadať algoritmy na vypočítanie vzdialeností nielen pre bezkontextové jazyky, ale aj kontextové, rekurzívne, či rekurzívne vyčísliteľné jazyky. Najvyššie, ako sa vieme v Chomského hierarchii jazykov dostať je vzdialenosť regulárneho jazyka od bezkontextového jazyka.

Kapitola 2

Editačná vzdialenosť

Najznámejšou vzdialenosťou definovanou na slovách je nepochybne editačná vzdialenosť [Kul06, Moh03]. V princípe nám hovorí, koľko krát musíme editovať jedno slovo, aby sme dostali druhé. Na začiatku tejto kapitoly definujeme editačnú vzdialenosť a predstavíme algoritmus na vypočítanie editačnej vzdialenosti dvoch slov. V ďalšej časti sa budeme venovať už existujúcemu algoritmu na vypočítanie editačnej vzdialenosti dvoch regulárnych jazykov a na záver predstavíme vlastný algoritmus na vypočítanie editačnej vzdialenosti regulárneho jazyka od bezkontextového jazyka.

Označenie 2.0.7 *Nech Σ je abeceda. Potom množinu*

$$\Sigma_E = \{\binom{a}{b} | a, b \in \Sigma\} \cup \{\binom{\varepsilon}{a} | a \in \Sigma\} \cup \{\binom{a}{\varepsilon} | a \in \Sigma\}$$

budeme nazývať editačnou abecedou nad abecedou Σ. Prvky množiny $\Sigma_I = \{\binom{\varepsilon}{a} | a \in \Sigma\}$ budeme nazývať vloženia. Prvky množiny $\Sigma_D = \{\binom{a}{\varepsilon} | a \in \Sigma\}$ budeme nazývať vymazania. Prvky množiny $\Sigma_T = \{\binom{a}{b} | a, b \in \Sigma\}$ budeme nazývať zmeny.

Množina Σ_E nám definuje množinu transformácií. Všimnime si, že $\Sigma_E = \Sigma_I \cup \Sigma_D \cup \Sigma_T$, čo znamená, že povolené transformácie sú vloženie písmena, vymazanie písmena a zmena písmena. Taktiež si všimnite, že posledná transformácia je zbytočná a dá sa zložiť z vymazania a vloženia a teoreticky by sme ju potom nemuseli uvažovať, ale pri praktickom vyžití význam má, napríklad ak chceme mať jednoduchú kontrolu pravopisu (Aspell[Asp] vo svojich prvých verziách používal takto definovanú editačnú vzdialenosť, neskôr pridal aj možnosť výmenenia dvoch susedných písmen). Teraz si definujeme jednoduchú hodnotiacu funkciu.

Definícia 2.0.8 $\Psi : \Sigma_E^* \to \mathbb{R}$ *je funkcia definovaná nasledovne:*

$$\begin{aligned}
\Psi(\varepsilon) &= 0 \\
\Psi(\binom{a}{a}) &= 0, a \in \Sigma \\
\Psi(\binom{a}{b}) &= 1, a \neq b, a, b \in \Sigma \cup \{\varepsilon\} \\
\Psi(w) &= \Sigma_{i=1}^{|w|} \Psi(w_i); w \in \Sigma_E^* \wedge |w| \geq 2
\end{aligned}$$

Číslo $\Psi(w)$ *budeme nazývať cena* slova w. *Nech* $u, v \in \Sigma_E^*$ *a* $\Psi(u) \leq \Psi(v)$. *Potom hovoríme, že* u *je lacnejšie ako* v.

Každé pridanie, zmazanie a zmenenie písmena nás stojí 1 nezmenenie písmena nás nič nestojí. Túto hodnotiacu funkciu budeme používať v celej tejto kapitole. Tak isto, ako nasledujúce homomorfizmy h_1 a h_2.

Definícia 2.0.9 h_1, h_2 *sú homomorfizmy definované nasledovne:*

$$\begin{aligned}
h_1(\binom{a}{b}) &= a, a, b \in \Sigma \cup \{\varepsilon\} \\
h_2(\binom{a}{b}) &= b, a, b \in \Sigma \cup \{\varepsilon\}
\end{aligned}$$

Definícia 2.0.10 *Nech* Σ *je abeceda a* Σ_E *k nej prislúchajúca editačná abeceda. Funkciu* $d_E : \Sigma^* \times \Sigma^* \to \mathbb{R}$ *budeme nazývať editačnou vzdialenosťou, práve vtedy, keď*

$$d_E(u, v) = \min\{\Psi(w) | w \in \Sigma_E^* \wedge h_1(w) = u \wedge h_2(w) = v\}$$

Ak $u, v \in \Sigma^*$ *tak číslo* $d_E(u, v)$ *budeme nazývať* editačnou vzdialenosťou *slov* u *a* v. *Také slovo* $w \in \Sigma_E$, *ktoré spĺňa podmienku* $h(w) = d_E(u, v) \wedge h_1(w) = u \wedge h_2(w) = v$ *budeme nazývať* editačné slovo *slov* u *a* v.

Príklad: Nech $u = abbbb$ a $v = cbbb$. Editačná vzdialenosť slov u a v je 2. Editačné slová slov u a v sú nasledovné slová: $\binom{a}{\varepsilon}\binom{b}{c}\binom{b}{b}\binom{b}{b}\binom{b}{b}$, $\binom{a}{c}\binom{b}{\varepsilon}\binom{b}{b}\binom{b}{b}\binom{b}{b}$, $\binom{a}{c}\binom{b}{b}\binom{b}{\varepsilon}\binom{b}{b}\binom{b}{b}$, $\binom{a}{c}\binom{b}{b}\binom{b}{b}\binom{b}{\varepsilon}\binom{b}{b}$, $\binom{a}{c}\binom{b}{b}\binom{b}{b}\binom{b}{b}\binom{b}{\varepsilon}$. Každé z nich má na hornom poschodí u a na spodnom poschodí v a cena každého slova je 2. Ľubovoľné iné slovo $w \in \Sigma_E^*$ spĺňajúce $h_1(w) = u \wedge h_2(w) = v$ už nie je editačné slovo, lebo má väčšiu cenu. Ako príklad použijeme slovo $w = \binom{a}{\varepsilon}\binom{\varepsilon}{c}\binom{b}{\varepsilon}\binom{b}{b}\binom{b}{b}\binom{b}{b}$, ktorého cena je 3 a teda $\Psi(w) = 3$.

Definícia 2.0.10 sa líši od pôvodných definícii, ale pre naše potreby je vhodnejšia a až na jednu lemu, dôkazy s ňou budú jednoduchšie. Ľahko vidieť, že táto definícia je ekvivalentná s pôvodnou definíciou, ktorá sa dá nájsť napríklad v [Kul06].

Pre úplnosť a na zvyknutie čitateľa na notácie, uvádzame nasledovnú lemu. Dôkaz tejto lemy pre pôvodnú definíciu sa nachádza v [Kul06].

Lema 2.0.11 *Editačná vzdialenosť je vzdialenosťou nad* Σ^*.

Dôkaz. Najskôr si ukážeme že $d_E(u,v)$ je konečné číslo. Nech $k = |u| + |v|$. Je zrejmé, že $d_E(u,v) \leq k$ (najskôr použijeme $|u|$ krát vymazanie písmena a potom $|v|$ krát pridanie písmena). Z definície Ψ vyplýva, že $\Psi(w) \geq 0$ pre ľubovoľné $w \in \Sigma_E$.

Reflexívnosť vyplýva z toho, že $\Psi(\binom{a}{a}) = 0$. Nech $u \in \Sigma^*$. Definujme homomorfizmus $h(a) = \binom{a}{a}, a \in \Sigma$. Nech $w = h(u)$. Zrejme platí, že $u = h_1(w) = h_2(w)$. Taktiež platí $\Psi(w) = 0$, čo znamená, že $d_E(u,u) = 0$.

Symetria sa dá tiež dokázať jednoducho. Nech $u, v \in \Sigma^*$ a $w \in \Sigma_E$ je editačné slovo slov u a w. Definujme si homomorfizmus h_{rot} nasledovne: $h_{rot}(\binom{a}{b}) = \binom{b}{a}.a, b \in \Sigma \cup \{\varepsilon\}$. Nech $w' = h_{rot}(w)$. Platí, že $\Psi(w') = \Psi(w) = d_E(u,v) \wedge h_1(w') = v \wedge h_2(w') = u$. Stačí už len dokázať, že $\Psi(w') = d_E(v,u)$. To dokážeme sporom. Nech $\Psi(w') > d_E(v,u)$. Potom ale existuje editačné slovo $x \in \Sigma_E^*$ slov u a v. Avšak $\Psi(h_{rot}(x)) < \Psi(w) \wedge h_1(h_{rot}(x) = u \wedge h_2(h_{rot}(x)) = v$, čiže w nemôže byť editačným slovom slov u a v.

Dokázať trojuholníkovú nerovnosť bude zložitejšie. Nech $u, v, w \in \Sigma^*$. Nech x je editačné slovo slov u a v, y nech je editačné slovo slov v a w. Teraz nájdeme také editačné slovo z, pre ktoré platí $h_1(z) = u$ a $h_2(z) = w$ a zároveň $\Psi(x) + \Psi(y) \geq \Psi(z)$, z čoho priamo vyplýva platnosť trojuholníkovej nerovnosti. Vymažme zo slova x všetky písmená tvaru $\binom{a}{\varepsilon}, a \in \Sigma$ a zo slova y všetky písmená tvaru $\binom{\varepsilon}{a}, a \in \Sigma$. Ostanú nám slová x' a y'. Musí platiť, že $h_2(x') = h_1(y')$. Pozrime sa teraz na písmená x'_i a $y'_i, 1 \leq i \leq |x'|$. Zostrojíme slovo $z' \in (\Sigma_E \cup \{\binom{\varepsilon}{\varepsilon}\})^*, |z'| = |x'|$. Môžu nastať 4 prípady.

1. $x'_i = \binom{a}{b} \wedge y'_i = \binom{b}{c}, a, b, c \in \Sigma$: $z'_i = \binom{a}{c}$
2. $x'_i = \binom{\varepsilon}{a} \wedge y'_i = \binom{a}{\varepsilon}, a \in \Sigma : z'_i = \binom{\varepsilon}{\varepsilon}$
3. $x'_i = \binom{\varepsilon}{a} \wedge y'_i = \binom{a}{b}, a, b \in \Sigma$: $z'_i = \binom{\varepsilon}{b}$
4. $x'_i = \binom{a}{b} \wedge y'_i = \binom{b}{\varepsilon}, a, b \in \Sigma$: $z'_i = \binom{a}{\varepsilon}$

Je zrejmé, že platí že $h_1(z') = h_1(x') \wedge h_2(z') = h_2(y')$. Chceme ukázať[1] že $\Psi(z') \leq \Psi(x') + \Psi(y')$. Pozrime sa na jednotlivé písmená x'_i, y'_i, z'_i. Ak nastal prvý prípad a $a = c$, tak cena písmena v $\Psi(z'_i) = 0 \leq \Psi(x'_i) + \Psi(y'_i)$. Ak $a \neq c$, tak $\Psi(z'_i) = 1$ ale buď $\Psi(x'_i) = 1$ alebo $\Psi(y'_i) = 1$ a preto $\Psi(z'_i) \leq \Psi(x'_i) + \Psi(y'_i)$. Ak nastal druhý prípad, tak $\Psi(z'_i) = 0 \leq \Psi(x'_i) + \Psi(y'_i)$. Pri treťom a štvrtom prípade je $\Psi(z'_i) = 1$ a $\Psi(x'_i) + \Psi(y'_i) \geq 1$.

Dostali sme, že pre všetky i je $\Psi(z'_i) \leq \Psi(x'_i) + \Psi(y'_i)$ a teda aj $\Psi(z') \leq \Psi(x') + \Psi(y')$. Zoberme teraz slovo z' a vymažme z neho písmená tvaru $\binom{\varepsilon}{\varepsilon}$, čím dostaneme slovo z'', pričom slovo z'' má rovnaké vlastnosti ako slovo z'. Pridajme teraz do slova z'' tie písmená, ktoré sme predtým vymazali zo slov x a y a vytvorme nové slovo z tak, aby platilo $h_1(z) = u \wedge h_2(z) = w$. Pozrime sa na cenu slova z. Označme si počet vymazaných písmen zo slov x a y číslom k. Potom sme do slova z'' tých k písmen naspäť (Každé také písmeno má cenu 1). Čiže $\Psi(z) = k + \Psi(z'')$. Zároveň $\Psi(x) + \Psi(y) = k + \Psi(x') + \Psi(y')$. Z toho vyplýva, že $\Psi(z) = k + \Psi(z'') \leq k + \Psi(x') + \Psi(y') = \Psi(x) + h(y)$. □

[1] Za predpokladu, že Ψ rozšírime o $\Psi(\binom{\varepsilon}{\varepsilon}) = 0$.

Označenie 2.0.12 *Nech $u \in \Sigma^+$. Potom $[u]_i$ budeme nazývať prefix slova u dĺžky i, teda*

$$[u]_i = v, u = vw, v, w \in \Sigma^*, |v| = i$$

Zároveň u_i budeme označovať i-te písmeno slova u, čiže $u_i = c, u = vcw, v, w \in \Sigma^* \wedge c \in \Sigma \wedge |v| = i - 1$.

Teraz sa budeme zaoberať tým, ako vypočítať editačnú vzdialenosť dvoch slov. Efektívne vypočítanie vzdialenosti slov u a v nad abecedou Σ založíme na nasledovnej leme (dôkaz pre pôvodnú definíciu ako aj algoritmus na vypočítanie editačnej vzdialenosti sa dá nájsť v [Kul06]).

Lema 2.0.13 *Nech $u, v \in \Sigma^*$. Potom platia nasledovné vzťahy:*

1. $d_E([u]_0, [v]_0) = 0$
2. $d_E([u]_i, [v]_0) = i, i > 0$
3. $d_E([u]_0, [v]_i) = i, i > 0$
4. *Ak $u_i \neq v_j$, tak*

   $$d_E([u]_i, [v]_j) = \min\{d_E([u]_{i-1}, [v]_{j-1})+1, d_E([u]_i, [v]_{j-1})+1, d_E([u]_{i-1}, [v]_j)+1\}, i, j > 0$$

5. *Ak $u_i = v_j$, tak*

   $$d_E([u]_i, [v]_j) = \min\{d_E([u]_{i-1}, [v]_{j-1}), d_E([u]_i, [v]_{j-1}) + 1, d_E([u]_{i-1}, [v]_j) + 1\}, i, j > 0$$

Dôkaz. Prvý vzťah vyplýva priamo z definície. Pri druhom a treťom si stačí uvedomiť, že môžeme použiť buď $|u|$ krát vymazanie, alebo $|v|$ krát pridanie a že žiadnu inú operáciu nemôžeme použiť.

Štvrtý a piaty vzťah dokážeme zložitejšie. Nech w je editačné slovo slov $[u]_i$ a $[v]_j$. Posledné písmeno slova w je buď $\binom{u_i}{v_j}$ alebo $\binom{u_i}{\varepsilon}$ alebo $\binom{\varepsilon}{u_j}$. Nech $w' = [w]_{|w|-1}$. Nech $u' = h_1(w')$ a $v' = h_2(w')$. Potom w' je editačné slovo slov u' a v'. Dokážeme to sporom. Nech x je editačné slovo slov u' a v' a $\Psi(x) < \Psi(w')$ a nech s je posledné písmeno slova w. Potom slovo xs je lacnejšie ako w a $h_1(xs) = [u]_i$ a $h_2(xs) = [v]_i$ a $\Psi(xs) < \Psi(w)$, čo je v spore s tým, že w je editačné slovo slov $[u]_i$ a $[v]_j$.

Aká je cena editačného slova w? $\Psi(w) = \Psi(w') + \Psi(s)$. Ak $s = \binom{a}{a}, a \in \Sigma$, tak $\Psi(w) = \Psi(w')$ a ináč $\Psi(w) = \Psi(w') + 1$.

Jednotlivé prípady, vyzerajú nasledovne:

1. $s = \binom{a}{b}, a, b \in \Sigma, a \neq b$. Vtedy $d_E([u]_i, [v]_j) = d_E([u]_{i-1}, [v]_{j-1}) + 1$
2. $s = \binom{a}{a}, a \in \Sigma$. Vtedy $d_E([u]_i, [v]_j) = d_E([u]_{i-1}, [v]_{j-1})$
3. $s = \binom{a}{\varepsilon}, a \in \Sigma$. Vtedy $d_E([u]_i, [v]_j) = d_E([u]_{i-1}, [v]_j) + 1$

4. $s = \binom{\varepsilon}{a}, a \in \Sigma$. Vtedy $d_E([u]_i, [v]_j) = d_E([u]_i, [v]_{j+1}) + 1$

Otázkou už iba ostáva, či sa nemôže stať, že w končí písmenom $s = \binom{a}{b}, a, b \in \Sigma, a \neq b$, teda $d_E([u]_i, [v]_j) = d_E([u]_{i-1}, [v]_{j-1}) + 1$ ale $d_E([u]_{i-1}, [v]_j) + 1$ bude menšie? Toto sa stať nemôže, lebo v takom prípade vieme zostrojiť také slovo w'' s cenou $d_E([u]_{i-1}, [v]_j) + 1$ (editačné slovo slovo $[u]_{i-1}, [v]_j$ zreťazené s $\binom{u_i}{\varepsilon}$), a teda $d_E([u]_i, [v]_j) < d_E([u]_{i-1}, [v]_{j-1}) + 1)$ čo je spor. Ostatné možnosti vyzerajú analogicky. □

Teraz už máme dostatok informácii aby sme zostrojili efektívny algoritmus na vypočítanie editačnej vzdialenosti dvoch slov. Pre zjednodušenie označenia, si zavedieme funkciu $A(i, j) = d_E([u]_i, [v]_j)$. Keďže $d_E(u, v) = d_E([u]_{|u|}, [v]_{|v|})$, tak potom $d_E(u, v) = A(|u|, |v|)$. Ak chceme vypočítať editačnú vzdialenosť slov u a v, potrebujeme vypočítať $A(|u|, |v|)$. Priamym aplikovaním lemy 2.0.13 by sme dostali algoritmus s exponenciálnou časovou zložitosťou. My využijeme fakt, že výsledok $A(i, j)$ závisí len od parametrov i a j, čo znamená, že si môžeme vypočítané výsledky zapamätať a potom tú istú hodnotu nemusíme vypočítať viac krát. Takto dostaneme algoritmus s časovou zložitosťou $O(|u||v|)$ a rovnakou pamäťovou zložitosťou. Pamäťová zložitosť sa však dá zlepšiť použitím dynamického programovania. Na výpočet hodnôt $A(i, j), 0 \leq j \leq |v|$ a i je pevné, nám stačí poznať hodnoty $A(i-1, j), 0 \leq j \leq |v|$. Teda naraz si potrebujeme pamätať iba hodnoty $A(i-1, j)$ a $A(i, j), 0 \leq j \leq |v|$ a funkciu A počítať postupne, teda najskôr pre $i = 0$, potom pre $i = 1$, atď.

Kód algoritmu vyzerá nasledovne:

```
1)  distance(string u,string v):
2)      for i=0 to |v| do:
3)          A[0,i]=i;
4)      for i=1 to |u| do:
5)          A[i,0]=i;
6)          for j=1 to |v| do:
7)              A[i mod 2,j]=min(A[i mod 2,j-1]+1,A[(i+1) mod 2,j]+1)
8)              if u[i]=v[j]:
9)                  A[i mod 2,j]=min(A[i mod 2,j],A[(i+1) mod 2,j-1])
10)             else:
11)                 A[i mod 2,j]=min(A[i mod 2,j],A[(i+1) mod 2,j-1]+1)
12)     return A[|a|,|b|];
```

Dostali sme algoritmus, ktorý má časovú zložitosť $O(|u||v|)$ a spotrebuje iba $O(|v|)$ pamäte. Pamäťovú zložitosť algoritmu vieme jednoducho redukovať na $O(\min\{|u|, |v|\})$, ak algoritmus rozšírime tak, že ak $|u| < |v|$, tak nebudeme počítať A po „riadkoch", ale po „stĺpcoch".

2.1 Vzdialenosť dvoch regulárnych jazykov

Algoritmus na hľadanie editačnej vzdialenosti dvoch regulárnych jazykov sa nachádza tiež v [Moh03], avšak ten používa Dijsktrov algoritmus[CLR90] lebo uvažuje všeobecnejšie ohod-

notenia (takými sa budeme zaoberať v kapitole 3). Z toho dôvodu má horšiu časovú zložitosť ako algoritmus prezentovaný v tejto kapitole.

Definícia 2.1.1 *Nech* $L_1, L_2 \subseteq \Sigma^*$. *Potom* $d_E(L_1, L_2) = \min\{d_E(u, v) | u \in L_1 \wedge v \in L_2\}$ *a* $d_E(L_1, L_2)$ *budeme nazývať editačnou vzdialenosťou jazykov* L_1 *a* L_2.

Algoritmus na vypočítanie vzdialenosti dvoch slov pomocou dynamického programovania nám dáva návod , ako vypočítať editačnú vzdialenosť dvoch regulárnych jazykov. Predpokladajme, že máme zadané dva regulárne jazyky L_1, L_2 a k nim dva konečné automaty $A_1 = (K_1, \Sigma, \delta_1, q_1, F_1), A_2 = (K_2, \Sigma, \delta_2, q_2, F_2)$, pričom platí $L(A_i) = L_i, i \in \{1, 2\}$.

Pre jednoduchosť budúcich konštrukcii si potrebujeme dokázať nasledovné dve lemy. V dôkazoch uvedieme len konštrukcie, úplné znenia dôkazov sa nachádzajú v [JH79].

Lema 2.1.2 *Ku každému NKA* $A = (K_A, \Sigma, \delta_A, q_A, F_A)$ *existuje ekvivalentný NKA* $B = (K_B, \Sigma, \delta_B, q_B, F_B)$ *taký, ktorý neobsahuje* ε *prechody, t.j.* $L(A) = L(B) \wedge \forall q \in K_B : |\delta_B(q, \varepsilon)| = 0$.

Dôkaz. Uvedieme len konštrukciu: B bude mať rovnakú množinu stavov, rovnakú abecedu a rovnaký počiatočný symbol ako A.

Pre každý stav q, taký, že existuje $p \in \delta_A(q, \varepsilon)$ bude v $p' \in \delta_B(q, u) \Leftrightarrow (p, u) \vdash_A^* (p'', u) \vdash_A (p', \varepsilon)$. Ostatné pravidlá δ_A funkcie do δ_B funkcie skopírujeme.

Množinu akceptačných stavov automatu B bude tvoriť nasledovná množina: $F_B = \{p | (p, \varepsilon) \vdash_A^* (q, \varepsilon), q \in F_A\}$. □

Lema 2.1.3 *Ku každému NKA* $A = (K_A, \Sigma, \delta_A, q_A, F_A)$ *existuje ekvivalentný NKA* $B = (K_B, \Sigma, \delta_B, q_B, F_B)$ *taký, ktorý v každom kroku vypíše práve jedno písmeno, t.j.* $L(A) = L(B) \wedge \forall q \in K_B, \forall u \in \Sigma^* : |u| \neq 1 \Rightarrow |\delta(q, u)| = 0$.

Dôkaz. Opäť uvedieme len konštrukciu: Z predchádzajúcej lemy vieme, že existuje taký NKA A' ktorý neobsahuje ε prechody. Z dôkazu predchádzajúcej lemy preberieme označenia.

B bude obsahovať stavy, ktoré má automat A a ešte nejaké stavy navyše, ktoré sa vyskytnú v nasledovnej konštrukcií: Nech $p \in \delta_{A'}(q, u), |u| = n$. Potom do automatu B pridáme unikátne stavy $q_2, q_3, \cdots q_n$ a pravidlá $q_2 \in \delta_B(q, u_1), q_{i-1} \in \delta_B(q_i, u_i), 3 \leq i \leq n$ a pravidlo $p \in \delta_B(q_n, u_n)$. Ak pravidlo akceptuje práve jedno písmeno, do automatu B ho skopírujeme.

Akceptačné stavy automatu B budú totožné s akceptačnými stavmi automatu A'. Dostali sme ekvivalentný automat, ktorý v každom kroku vypíše práve jedno písmeno. □

Vďaka poslednej leme, môžeme odteraz predpokladať, že pracujeme iba s automatmi, ktoré v každom kroku akceptujú práve jedno písmeno, čo nám značne zjednoduší konštrukciu.

Teraz si zostrojíme nový NKA $C = (K_C, \Sigma_E, \delta_C, q_c, F_C)$. Automat bude mať ako stavy dvojice z $K_1 \times K_2$ a bude pracovať nad abecedou Σ_E. Výpočet bude vyzerať tak, že buď

sa automat „posunie" v automatoch A_1 aj A_2 (a akceptuje symbol $\binom{a_1}{a_2}$, pričom automat A_i by akceptoval symbol a_i) alebo iba v jednom z nich (vtedy v druhom z nich výpočet „stojí" a na patričnom poschodí bude ε). Formálne konštrukcia vyzerá nasledovne:

$$K_C = K_1 \times K_2$$

$$\begin{aligned}
(p,r) \in \delta_C((q,s), \binom{a}{b}) &\Leftrightarrow \exists p,q \in K_1, \exists r,s \in K_2, \exists a,b \in \Sigma : p \in \delta_1(q,a), r \in \delta_2(s,b) \\
(p,s) \in \delta_C((q,s), \binom{a}{\varepsilon}) &\Leftrightarrow \exists p,q \in K_1, \exists s \in K_2, \exists a \in \Sigma : p \in \delta_1(q,a) \\
(q,r) \in \delta_C((q,s), \binom{\varepsilon}{b}) &\Leftrightarrow \exists q \in K_1, \exists r,s \in K_2, \exists b \in \Sigma : r \in \delta_2(s,b)
\end{aligned}$$

$$\begin{aligned}
q_C &= (q_1, q_2) \\
F_C &= F_1 \times F_2
\end{aligned}$$

Tento automat hrá kľúčovú úlohu v našom algoritme. Akceptuje také slová z Σ_E^*, že ich horná časť patrí do L_1 a spodná časť patrí do L_2. Zároveň každé písmeno slova je nejaká povolená operácia. Teraz dokážeme, že sa tam nachádza aj také slovo, ktorého cena bude rovná editačnej vzdialenosti jazykov L_1 a L_2 a že lacnejšie slovo tam nebude.

Definícia 2.1.4 *Nech L_1, L_2 sú ľubovoľné jazyky nad abecedou Σ. Potom jazyk nad editačnou abecedou Σ_E definovaný nasledovne:*

$$L_{L_2}^{L_1} = \{w | w \in \Sigma_E^* \wedge h_1(w) \in L_1 \wedge h_2(w) \in L_2\}$$

nazveme editačný jazyk *jazykov L_1, L_2 a označíme ho $L_{L_2}^{L_1}$. Také slovo $w \in \Sigma_E$, pre ktoré platí $h_1(w) \in L_1 \wedge h_2(w) \in L_2 \wedge \Psi(w) = d_E(L_1, L_2)$ nazveme* editačné slovo *jazykov L_1 a L_2.*

Veta 2.1.5 *Jazyk $L(C) = L_{L_2}^{L_1}$*

Dôkaz. Najskôr si indukciou dokážeme nasledovné tvrdenie: Nech $w \in \Sigma_E^*, |w| = n$ a nech $u = h_1(w), v = h_2(w)$. Potom

$$((q_1,q_2),w)) \vdash_C^* ((p,q),\varepsilon) \Leftrightarrow (q_1,u) \vdash_1^* (p,\varepsilon) \wedge (q_2,v) \vdash_2^* (q,\varepsilon)$$

1. $n = 0$. Automat C je v počiatočnom stave (q_1, q_2) a automat A_1 je v stave q_1, automat $A_1 = q_1$ a $A_2 = q_2$. Keďže automaty A_1, A_2, C nemajú prechody na ε, tak na akceptovanie prázdnych slov nemôžu byť v iných stavoch.

2. Nech indukčný predpoklad platí pre n. Nech $w \in \Sigma^*, |w| = n + 1$. Nech $s \in \Sigma_E, w' \in \Sigma^*, w = w's$ a nech $u' = h_1(w'), v' = h_2(w')$. Teraz dokážeme jednotlivé časti ekvivalencie:

- „⇒": Nech $(q_C, w) \vdash_C^* ((p, q), \varepsilon)$. Potom $((q_C, w') \vdash_C^* ((p, q), \varepsilon)$ pre nejaké $p \in K_1, q \in K_2$. Potom podľa indukčného predpokladu $(q_1, u') \vdash_1^* (p', \varepsilon) \wedge (q_2, v') \vdash_2^* (q', \varepsilon)$. Môžu nastať 3 prípady:
  (a) $u \neq u' \wedge v \neq v'$: Keďže $((p', q'), s) \vdash_C ((p, q), \varepsilon) \Leftrightarrow (p', h_1(s)) \vdash_1 (p, \varepsilon) \wedge (q', h_2(s)) \vdash_2 (q, \varepsilon)$ (definícia δ_C funkcie), tak aj $(q_1, u) \vdash_1^* (p, \varepsilon) \wedge (q_2, u) \vdash_2^* (q.\varepsilon)$.
  (b) $u \neq u', \wedge v = v'$: Keďže $((p', q), s) \vdash_C ((p, q), \varepsilon) \Leftrightarrow (p', h_1(s)) \vdash_1 (p, \varepsilon)$ (opäť z definície δ_C funkcie), tak aj $(q_1, u) \vdash_1^* (p, \varepsilon) \wedge (q_2, u) \vdash_2^* (q.\varepsilon)$.
  (c) $u = u' \wedge v \neq v'$: Keďže $((p, q'), s) \vdash_C ((p, q), \varepsilon) \Leftrightarrow (q', h_2(s)) \vdash_1 (q, \varepsilon)$ (opäť z definície δ_C funkcie), tak aj $(q_1, u) \vdash_1^* (p, \varepsilon) \wedge (q_2, u) \vdash_2^* (q.\varepsilon)$.
- „⇐": Nech $(q_1, u) \vdash_1^* (p, \varepsilon) \wedge (q_2, v) \vdash_2^* (q, \varepsilon)$. Potom $(q_1, u') \vdash_1^* (p', \varepsilon) \wedge (q_2, v') \vdash (q', \varepsilon)$. Podľa indukčného predpokladu $(q_C, w') \vdash_C^* ((p', q'), \varepsilon)$. Tu môžu tiež nastať 3 prípady:
  (a) $u \neq u' \wedge v \neq v'$: Ak $(q_1, u) \vdash_1^* (p, \varepsilon) \wedge (q_2, v) \vdash_2^* (q, \varepsilon)$ tak aj $(q_1, u') \vdash_1^* (p', \varepsilon) \wedge (q_2, v') \vdash_2^* (q', \varepsilon)$. Z indukčného predpokladu vyplýva, že $(q_C, w') \vdash_C^* ((p', q'), \varepsilon)$. Keďže $(p', h_1(s)) \vdash_1 (p, \varepsilon) \wedge (q', h_2(s)) \vdash_2 (q, \varepsilon)$ tak aj $((p', q'), s) \vdash_C ((p, q), \varepsilon)$ a teda $(q_C, w) \vdash_C ((p, q), \varepsilon)$.
  (b) $u \neq u' \wedge v = v'$: Ak $(q_1, u) \vdash_1^* (p, \varepsilon) \wedge (q_2, v) \vdash_2^* (q, \varepsilon)$ tak aj $(q_1, u') \vdash_1^* (p', \varepsilon)$. Z indukčného predpokladu vyplýva, že $(q_C, w') \vdash_C^* ((p', q), \varepsilon)$. Keďže $(p', h_1(s)) \vdash_1 (p, \varepsilon)$ tak aj $((p', q), s) \vdash_C ((p, q), \varepsilon)$ a teda $(q_C, w) \vdash_C ((p, q), \varepsilon)$.
  (c) $u = u' \wedge v \neq v'$: Ak $(q_1, u) \vdash_1^* (p, \varepsilon) \wedge (q_2, v) \vdash_2^* (q, \varepsilon)$ tak aj $(q_2, v') \vdash_2^* (q', \varepsilon)$. Z indukčného predpokladu vyplýva, že $(q_C, w') \vdash_C^* ((p, q'), \varepsilon)$. Keďže $(q', h_2(s)) \vdash_2 (q, \varepsilon)$ tak aj $((p, q'), s) \vdash_C ((p, q), \varepsilon)$ a teda $(q_C, w) \vdash_C ((p, q), \varepsilon)$.

Teraz môžeme pristúpiť k samotnému dôkazu. Inklúzie dokážeme oddelene.

- $L(C) \subseteq L_{L_2}^{L_1}$: Nech $w \in L(C)$. Podľa predchádzajúceho tvrdenia $h_1(w) \in L_1 \wedge h_2(w) \in L_2$ a teda $w \in L_{L_2}^{L_1}$.
- $L(C) \supseteq L_{L_2}^{L_1}$: Nech $w \in L_{L_2}^{L_1}$. Teda $h_1(w) \in L_1 \wedge h_2(w) \in L_2$ a podľa predchádzajúceho tvrdenia $w \in L(C)$.

□

Nasledujúca veta je veľmi dôležitá, lebo ju využijeme aj neskôr pri hľadaní vzdialenosti regulárneho jazyka od bezkontextového jazyka.

Veta 2.1.6 *Nech L_1, L_2 sú ľubovoľné jazyky a $L_{L_2}^{L_1}$ je ich editačný jazyk. Nech $w \in L_{L_2}^{L_1}$ a $\forall v \in L_{L_2}^{L_1} : \Psi(w) \leq \Psi(v)$. Potom $\Psi(w) = d_E(L_1, L_2) = d_E(h_1(w), h_2(w))$.*

Dôkaz. Rovnosť $\Psi(w) = d_E(h_1(w), h_2(w))$ vyplýva z definície d_E.

Zvyšnú rovnosť dokážeme sporom. Nech $u \in L_1, v \in L_2$ sú také slová, ktoré majú editačné slovo w' a $\Psi(w') < \Psi(w)$. Ale $w' \in L_{L_2}^{L_1}$, čo je v spore s predpokladom, že $\Psi(w)$ je najlacnejšie slovo z $L_{L_2}^{L_1}$. □

Podľa viet 2.1.5, 2.1.6, nám stačí nájsť editačné slovo $w \in L(C)$ (t.j. najlacnejšie slovo z $L(C)$). Potom $\Psi(w)$ bude hľadanou vzdialenosťou.

Teraz si úlohu preformulujeme. Na automat C pozrieme ako na graf G, kde stavy sú vrcholy a jednotlivé časti δ funkcie sú hrany, pričom cena hrany je rovná cene písmena, ktoré sa v danom kroku výpočtu vygeneruje. Cena najlacnejšieho slova z $L(C)$ sa rovná dĺžke najkratšej cesty v grafe G. Na hľadanie najlacnejšej cesty sa obyčajne používa Dijsktrov[CLR90] algoritmus, ktorý je však v tomto prípade zbytočné použiť, keď si uvedomíme, že graf má hrany s dĺžkou 0 alebo 1. V takomto prípade stačí použiť upravené prehľadávanie do šírky s deque[Knu97] namiesto frontu. Vrcholy so vzdialenosťou 0 budeme vkladať na začiatok deque a zvyšné vrcholy budeme vkladať na koniec deque.

Graf G bude obsahovať multihrany a preto kvôli redukcii počtu hrán si zapamätáme ku každej dvojici vrcholov iba tú najkratšiu. Dĺžku najkratšej cesty to nezmení.

V nasledovnom pseudokóde algoritmu neuvádzame kód zostrojenia grafu. Vysvetlenie identifikátorov: *deque.push_front* vloží prvok na začiatok *deuqe*, *deque.pop_front* vyberie prvok zo začiatku *deque* a *deque.push_back* vloží prvok na koniec *deque*. *G.edges(vertex)* je zoznam hrán, ktoré vychádzajú z vrchola *vertex*. Ak *e* je hrana, tak *e.where* je vrchol, do ktorého hrana smeruje. *G.was[v]* je pravdivostná hodnota, ktorá hovorí o tom, či vrchol v bol navštívený.

```
1)  create_graph G.
2)  deque.push_front((q_c,0))
3)  while not deque.front.first in F_C do:
4)      (vertex,distance)=deque.pop_front
5)      G.was[verter]=true
6)      for each e in G.edges(vertex) do:
7)          if not G.was[e.where] then:
8)              if e.length = 0 then:
9)                  deque.push_front((e.where,distance))
10)             else:
11)                 deque.push_back((e.where,distance+1))
12) return deque.front.first
```

V tejto časti sa budeme zaoberať časovou a pamäťovou zložitosťou algoritmu. Najskôr potrebujeme zostrojiť automat C. Ten má $|K_1| \times |K_2|$ stavov a $|\delta_1| \times |\delta_2| + |\delta_1||\delta_2|$ pravidiel v δ_C funkcii. To znamená, že graf G bude mať $|K_1| \times |K_2|$ vrcholov a najviac $\min(|\delta_1| \times |\delta_2| + |\delta_1||\delta_2|, |K_1|^2|K_2|^2)$ hrán. Vytvorenie automatu bude v čase $O(|K_1| \times |K_2| + |\delta_1| \times |\delta_2| + |\delta_1||\delta_2|)$. Prehľadávanie má lineárnu časovú zložitosť od veľkosti grafu a preto celková časová zložitosť algoritmu je $O(|K_1| \times |K_2| + |\delta_1| \times |\delta_2| + |\delta_1||\delta_2|)$.

Pri pamäťovej zložitosti máme dva varianty. Rozdiel je v spôsobe, ako si pamätáme graf G. Ak si pamätáme hrany ako maticu susedností vrcholov, tak pamäťová zložitosť bude $O(|K_1|^2 \times |K_2|^2)$ (napriek tomu, že δ_C môže byť oveľa väčšia, stačí si pri vytváraní automatu C pamätať pre dvojicu stavov len najlacnejší prechod medzi nimi).

Ak si budeme pamätať hrany v zozname hrán pre každý vrchol, tak potom bude pamäťová zložitosť algoritmu $O(|K_1| \times |K_2| + |\delta_1| \times |\delta_2| + |\delta_1||\delta_2|)$. Tento variant sa hodí

obzvlášť ak dopredu vieme, že náš graf bude mať málo hrán, alebo napríklad vtedy, ak máme na vstupe nie nedeterministické, ale deterministické automaty. Vtedy bude C tiež deterministický automat a nebudeme musieť redukovať počet hrán.

Pamäťová zložitosť sa dá tiež redukovať použitím takzvanej „on the fly" implementácie algoritmu, a teda zostrojovaním grafu počas výpočtu. Vtedy sa spracujú len tie hrany, ktoré treba spracovať a teda nám to v niektorých prípadoch prinesie urýchlenie algoritmu. Najhorší prípad nám to ale nezlepší.

2.2 Vzdialenosť regulárneho a bezkontextového jazyka

V tejto časti prezentujeme vlastný algoritmus na vypočítanie vzdialenosti regulárneho jazyka od bezkontextového jazyka. Základná idea je rovnaká ako pri regulárnych jazykoch, ale samotný algoritmus je už iný.

Zadanie máme nasledujúce: Nech L_1 je regulárny jazyk (dostaneme ho ako NKA

$$A = (K, \Sigma, \delta, q_0, F)$$

) a L_2 je bezkontextový jazyk (daný bezkontextovou gramatikou $G = (N_G, \Sigma, P_G, \sigma_G)$). Našou úlohou je nájsť

$$d_E(L_1, L_2) = \min\{d_E(u, v) | u \in L_1 \wedge v \in L_2\}$$

Dokážeme, že ku každej bezkontextovej gramatike existuje ekvivalentná gramatika v tvare, ktorý sa podobá na prísny Chomského normálny tvar. Jediný rozdiel je, že Chomského normálny tvar nedovoľuje pravidlá tvaru $A \to B$, kde A, B sú neterminály. Nám takéto pravidlá problémy robiť nebudú a preto ich povolíme, a nebudeme sa musieť zaoberať s „*chain rules*". Navyše v „našom" tvare je možné z počiatočného neterminálu vygenerovať ε. Viac o Chomského normálnom tvare sa dá nájsť v [JH79].

Veta 2.2.1 *Pre každú bezkontextovú gramatiku $G = (N, T, P, \sigma)$ existuje gramatika $G' = (N', T, P', \sigma)$, ktorá obsahuje pravidlá z množiny $N' \to (N' \times N' \cup N' \cup T)$ alebo pravidlo $\sigma \to \varepsilon$ a navyše $L(G) = L(G')$.*

Dôkaz. Uvedieme len konštrukciu. Až na odstraňovanie „chain-rules" je naša konštrukcia rovnaká v [JH79]

Najskôr odstránime z G pravidlá tvaru $A \to \varepsilon$. Potom vypočítame množinu pravidiel $M = \{n | n \in N, n \Rightarrow^* \varepsilon\}$, ktorú vieme nájsť nasledovným postupom. Nech $M_0 = \{n | n \in N, n \to \varepsilon \in N\}$. Nech $M_i = M_{i-1} \cup \{n | n \in N, n \to u \in P, u \in M_{i-1}^*\}$, Je zrejmé, že ak $M_i = M_{i+1}$ tak $M_i = M_k, k > i$ a teda $|N| = M_{|N|+1} = M$, lebo v M_i nemôže byť viac ako $|N|$ neterminálov. Nová sada pravidiel sa bude volať P_1 a bude tvorená nasledovnú množinou:

$$\{A \to u | u \in (N \cup T)^*, u = v_0 v_1 \cdots v_l, A \to v_0 A_1 \cdots A_l v_l \in P, l \geq 0, v_i \in (N \cup T)^*, A_i \in M\}$$

Ak $\varepsilon \in L(G)$ tak pridáme do P_1 pravidlo $\sigma \to \varepsilon$.

Teraz sme dostali ekvivalentnú sadu pravidiel, ktoré negenerujú ε. Teraz odstránime z pravých strán terminály nasledovne: Pre každý terminál $t \in T$, pridáme neterminál ε_t. Zoberme si homomorfizmus h taký, že $h(n) = n, n \in N$ a $h(t) = \varepsilon_t, t \in T$. Potom $N'' = N \cup \{\varepsilon_t | t \in T\}, P'' = \{\varepsilon_t \to t | t \in T\} \cup \{A \to h(u) | A \to u \in P_1\}$, čím sme dostali gramatiku $G'' = (N'', T, P'', \sigma)$, ktorá je ekvivalentná z G.

Ostáva nám odstrániť pravidlá, ktoré majú na pravej strane viac ako 2 neterminály. To spravíme nasledovnou konštrukciou: $P' = \cup_{p \in P''} P'(p)$ kde $P'(p) = \{p\}$ ak $p = A \to u, |u| \leq 2$ a ináč $P'(p) = \{A \to B_1'^p, B_1' \to B_1 B_2'^p, B_2'^p \to B_2 B_3'^p \cdots B_{k-1}'^p \to B_{k-1} B_k | p = A \to B_1 B_2 \cdots B_k, A, B_i \in N'', k > 2\}$ a $N'(p) = \{B_1'^p, B_2'^p, \cdots, B_{k-1}'^p | p = A \to B_1 B_2 \cdots B_k, k > 2\}$ a $N'(p) = \emptyset$ ak $p = A \to u, |u| \leq 2$. Nová sada neterminálov bude $N' = N \cup (\cup_{p \in p''} N'(p))$.

Gramatika $G' = (N', T, P', \sigma)$ je v požadovanom tvare a je ekvivalentná s G. □

Pri hľadaní algoritmu na vypočítanie vzdialenosti regulárneho jazyka od bezkontextového jazyka budeme postupovať v podstate rovnako, ako v predchádzajúcej časti. Najskôr si zostrojíme bezkontextovú gramatiku G_E, ktorá bude generovať jazyk $L_{L_2}^{L_1}$ a potom v nej nájdeme najlacnejšie slovo. Pre zjednodušenie konštrukcie, budeme predpokladať, že G je v tvare podľa lemy 2.2.1. Budeme tiež predpokladať, A tvare podľa lemy 2.1.3 (v každom kroku akceptuje práve jedno písmeno).

Myšlienka konštrukcie je nasledovná. Neterminály gramatiky G_E bude tvoriť množina $K \times N_G \cup \{\xi_\varepsilon\} \times K \cup \{\sigma\}$. Neterminál ξ_ε pridáme do gramatiky G tak, že sa bude dať vygenerovať z každého neterminálu a bude vedieť vygenerovať ľubovoľne veľa neterminálov ξ_ε. Tak, ako pri hľadaní editačnej vzdialenosti regulárnych jazykov, tak neterminál ξ_ε bude fungovať rovnako ako nevykonanie kroku nejakého automatu, a teda bude na spodnom poschodí výsledného slova generovať ε. Pravidlá v gramatike G_E budú zodpovedať pravidlám v gramatike G, čo zaručí, že na spodnom poschodí výsledného slova bude slovo z L_2. Navyše, v každej vetnej forme budú dva susedné neterminály na seba nadväzovať (ak sú to neterminály (p, A, q) a (r, B, s), tak $q = r$) a prvý neterminál je tvaru (q_0, A, q) a posledný neterminál je tvaru $(p, A, q_f), q_f \in F$. Nadväzovanie neterminálov a tvar prvého a posledného neterminálu zaručí, že na hornom poschodí bude slovo z L_1. Prepisovanie neterminálov na terminály bude zodpovedať akceptovaniu automatom A_1 a generovaniu gramatikou G. Neterminál (p, A, q) sa prepíše na $\binom{a}{b}$ ak $A \to b \in P_G$ a $q \in \delta(p, a)$. Neterminál (p, A, p) sa prepíše na $\binom{\varepsilon}{b}$ ak $A \to b \in P_G$ a neterminál (p, ξ_ε, q) sa prepíše na $\binom{a}{\varepsilon}$ ak $q \in \delta(p, a)$. Formálne G_E vyzerá nasledovne:

$$G_E = (N, T, P, \sigma),$$

kde

$$N = K \times N_G \cup \{\xi_\varepsilon\} \times K \cup \{\sigma\}$$

$$T = \Sigma_E$$

$$
\begin{aligned}
P \;=\; & \{(p,A,q) \to (p,B,r)(r,C,q) | (A \to BC \in P_G \vee A = B = C = \xi_\varepsilon) \wedge a,q,r \in K\} \\
& \cup \{(p,A,q) \to (p,B,q) | p \in K \wedge A \to B \in P_G\} \\
& \cup \{(p,A,q) \to (p,A,p')(p',\xi_\varepsilon,q) | p,q \in K \wedge A \in N_G\} \\
& \cup \{(p,A,q) \to (p,\xi_\varepsilon,p')(p',A,q) | p,q \in K \wedge A \in N_G\} \\
& \cup \{(p,A,q) \to \binom{a}{b} | q \in \delta(p,a) \wedge A \to b \in P_G\} \\
& \cup \{(p,A,p) \to \binom{\varepsilon}{a} | p \in K, A \to a \in P_G\} \\
& \cup \{(p,\xi_\varepsilon,q) \to \binom{a}{\varepsilon} | q \in \delta(p,a)\} \\
& \cup \{\sigma \to (q_0,\sigma_G,f) | f \in F\}
\end{aligned}
$$

Ak $\varepsilon \in L_2$, tak navyše P obsahuje aj pravidlo $\sigma \to \varepsilon$.

Pred dôkazom toho, že gramatika G_E generuje $L_{L_2}^{L_1}$ si dokážeme nasledujúcu lemu.

Lema 2.2.2 *Nech* $s \in N^*, \sigma \Rightarrow^+ s, |s| = n$. *Potom* $s_1 = (q_0,A,p)$ *a* $s_n = (p,A,q_f), A \in N_G$ *a* $\forall i, 1 \leq i < n : s_i = (p,A,q) \wedge s_{i+1} = (q,B,r), A,B \in N_G; p,q,r \in K$.

Dôkaz. Nech m je počet krokov odvodenia. Dôkaz urobíme indukciou vzhľadom na m.

1. $m = 1$ Jediné pravidlá, ktoré sa dajú použiť sú tvaru $\sigma \to (q_0,\sigma_G,f) | f \in F$ a teda vetná forma ostane v požadovanom tvare.

2. Nech tvrdenie platí pre m. Nech $\sigma \Rightarrow^m s', |s'| = n'$ a $s' \Rightarrow s$. Podľa indukčného predpokladu, $s'_1 = (q_0,A,p)$ a $s'_{n'} = (p,A,q_f)$ a $\forall i, 1 \leq i < n' : s'_i = (p,A,q) \wedge s'_{i+1} = (q,B,r)$. Nech s'_i je ten neterminál, ktorý sa prepíše v odvodení vetnej formy s. $s'_i = (p,A,q)$ pre nejaké p,q,A. Jediné prípustné[2] pravidlá sú tvaru $(p,A,q) \to (p,B,r)(r,C,q)$ alebo $(p,A,q) \to (p,B,q)$ a teda s bude spĺňať všetky podmienky tejto lemy.

□

Veta 2.2.3 $L(G_E) = L_{L_2}^{L_1}$

Dôkaz. Dôkaz sa skladá z dôkazov menších tvrdení.

(a) Nech $w \in L(G_E), |w| = n$. Potom $h_1(w) \in L_1$: Nech S je taká vetná forma z odvodenia slova w, ktorá obsahuje iba neterminály a pri každom ďalšom kroku odvodenia slova w sa použije pravidlo tvaru $A \to t, A \in N, t \in T$. Pre s platia podmienky z lemy 2.2.2. Teraz z každého neterminálu s_i odvodíme terminál w_i a dostaneme slovo w. Tvrdíme, že $h_1(w) \in L_1$. Odvodenie sa dá zostrojiť induktívne:

1. $m = 0$. Odvodenie začína v stave q_0.

[2]Keďže, že chceme dostať len neterminály.

2. m. Spracovali sme prvých m písmen a teda sme v stave q_m. Nech $s_m = (p, A, q_m)$ a nech $a \in \Sigma, b \in \Sigma \cup \{\varepsilon\}, q_m, q_{m+1} \in K, A \in N_G$. Ak $(q_m, A, q_{m+1}) \to \binom{a}{b}$, tak $q_{m+1} = \delta(q_m, a)$. Ak $(q_m, A, q_{m+1}) \to \binom{\varepsilon}{b}$, tak $q_m = q_{m+1}$. Inými slovami, ak sme vyrobili ε, tak sa nekonal žiadny krok odvodenia a keď sme vyrobili písmeno, tak sme vedeli spraviť krok odvodenia.

Po skončení procesu skončí výpočet v akceptačnom stave a teda $h_1(w) \in L_1$.

(b) Nech $w \in L(G_E)$. Potom $h_2(w) \in L_2$: Zoberme strom odvodenia slova w. Pozrime sa na neterminál ξ_ε. Z neho na spodnom poschodí vygenerujeme len ε. To znamená, že zo stromu odvodenia môžeme vymazať vetvy, ktoré majú ako koreň ξ_ε. Nech nám ostal strom S. Zoberme si ľubovoľný vrchol v (okrem koreňa). Z tohto stromu vyrobíme strom odvodenia slova $h_2(w)$. Z vetiev tvaru $(p, A, q) \to (p, B, r)(r, C, q)$ vyrobíme vetvy tvaru $A \to BC$ a z vetvy tvaru $(p, A, q) \to (p, B, q)$ vyrobím vetvu tvaru $A \to B$ (z konštrukcie G_E je jasné, že v gramatike E také pravidlá existujú. Ostávajú nám odvodenia listov. Tie sú však tvaru $(p, A, q) \to \binom{a}{b}$ a tie nahradíme $A \to b$, ktoré zase existuje. Takto sme zostrojili strom odvodenia slova $h_2(w)$ s gramatikou G a teda $h_2(w) \in L_2$.

(c) Nech $w \in L_{L_2}^{L_1}$. Táto časť dôkazu je príliš technická a preto uvediem iba myšlienku dôkazu. Nájdeme strom odvodenia slova $h_2(w)$ v gramatike G. Pridáme odvodenia prázdnych slov pomocou neterminálu ξ_ε, tak, aby odvodené slovo sedelo na spodné poschodie slova w. Potom neterminály, z ktorých sa vygenerovali terminály, prerobíme na neterminály z G_E tak, aby sledovali akceptovanie slova na hornom poschodí w. Ostatné neterminály v strome už vieme jednoznačne prerobiť na neterminály s G_E. Nakoniec pridáme odvodenie zo σ.

Z tvrdení (a), (b) vyplýva, že $L(G) \subseteq L_{L_2}^{L_1}$. Tvrdenie (c) dokazuje opačnú inklúziu. □

Teraz nám už iba ostáva nájsť najlacnejšie slovo z $L(G_E)$. Nech $C(k)$ je cena najlacnejšieho slova odvoditeľného z neterminálu k. Ak k je terminál, tak $C(k)$ je cena terminálu, teda $C(k) = \Psi(k)$. Nech $s \in (N \cup T)^*, |s| = n$. Potom $C(s) = \Sigma_{i=1}^{n} C(s_i)$. Platí, že $C(k) = \min\{C(s) | k \to s \in P\}$. Túto rovnosť však nie je úplne vhodné použiť priamo na vytvorenie algoritmu, lebo by mal príliš veľkú časovú zložitosť. Algoritmus bude podobný Dijkstrovmu[CLR90] algoritmu na hľadanie najkratšej cesty v grafe. Najskôr vypočítame $C(k)$ pre terminály. Potom v každom kroku nájdem také pravidlo $k \to s$, že na k je nespracovaný neterminál, s obsahuje len terminály a spracované neterminály a $C(s)$ je najmenšia možná. Vtedy určíme $C(k) = C(s)$. Toto budeme postupne opakovať, až kým sa nedostaneme k neterminálu σ. Vtedy sme našli $C(\sigma) = d_E(L_1, L_2)$.

Argument prečo je algoritmus správny je podobný ako v dôkaze Dijkstrovho algoritmu [CLR90]: Nech $C(k)$ je menšie, ako sa nám podarilo vypočítať (nech sa nám podarila vypočítať cena $M > C(k)$)a nech v najlacnejšom slove odvoditeľnom z neterminálu k bolo použité pravidlo $k \to s$. Každý prvok s má menšiu cenu ako $C(k)$ a preto bol spracovaný skôr ako k a teda $M \leq C(k)$, čo je v spore s tým, že $M > C(k)$.

Predchádzajú postup nám síce dal správny algoritmus, ale vieme využiť štruktúru gramatiky G_E a dosiahnuť lepšiu časovú zložitosť. Problém je v hľadaní správneho pravidla, pozeranie všetkých pravidiel je „drahé". Keďže G je v tvare podľa lemy 2.2.1, tak každé pravidlo, ktoré má na pravej strane neterminály je tvaru $A \to BC$ alebo $A \to B$. Spracovanie druhého typu pravidiel je priamočiare na rozdiel od pravidiel tvaru $A \to BC$: Pre každú dvojicu neterminálov z N_G si budeme pamätať aké neterminály z N_G danú dvojicu generujú (a rozšírime to aj to neterminál ξ_ε). Navyše si pre každý neterminál $B \in N_G$ a stav $q \in K$ budeme pamätať, ktoré neterminály tvaru (p, B, q) a (q, B, p) sme už spracovali. Z tohto vieme potom efektívne (v lineárnom čase) vygenerovať tie neterminály, ktoré sa stali „dosiahnuteľné[3]". Dosiahnuteľné neterminály budeme držať v prioritnom fronte, aby sme vedeli rýchlo vybrať nespracovaný dosiahnutený neterminál s minimálnou cenou.

Teraz sa pozrieme na časovú zložitosť algoritmu. V najhoršom prípade každý neterminál raz vyberieme z prioritnej fronty a pri každom pravidle zmeníme hodnotu neterminálu vo fronte a preto časová zložitosť algoritmu je $O((|N| + |P|) \log(|N|))$. Vzhľadom na to, že $|N| = |K|^2|N_G| + 1$ a $|P| = O(|K|^3|P_G|)$, tak pamäťová zložitosť celého algoritmu je $O((|K|^2|N_G|+|K|^3|P_G|) \log(|K|(|N_G)+|P_G|))$. Ak si nebudeme pamätať všetky pravidlá, ale dopočítavať si ich podľa potreby, tak algoritmus bude mať pamäťovú zložitosť $O(|K|^2|N_G|+|N_G|^2)$

V [Kul06] sa nachádza algoritmus na vypočítanie vzdialenosti slova od bezkontextového jazyka. Má prirodzene lepšiu časovú zložitosť a preto je vhodnejší v prípade, že máme jedno slovo, alebo konečný[4] jazyk namiesto regulárneho jazyka.

[3]Prvky na pravej strane sú už spracované,

[4]Je jednoduché rozšíriť techniky dynamického programovania na orientované acyklické grafy.

Kapitola 3

Prekladačová vzdialenosť

V praktických aplikáciach, ako napríklad kontrola pravopisu pri písaní v textovom editore sa môže stať, že niektoré transformácie sú viac pravdepodobné, ako iné, a preto použitie obyčajnej editačnej vzdialenosti nedosahuje optimálne výsledky. Vtedy je lepšie si ohodnotiť niektoré transformácie ináč. Občas sa ale stane, že dvom susedným písmenám iba vymeníme poradie a tu nám už ani takto upravená editačná vzdialenosť nepomôže. Ak chceme pracovať s Hammingovou[AB02] vzdialenosťou, musíme niektoré transformácie zakázať. Jednou s možností by bolo robiť zvlášť algoritmus pre každý nový typ vzdialenosti, ale rozumnejšie je si všetky takéto vzdialenosti popísať a použiť všeobecný algoritmus.

V [Moh03] sa nachádza zovšeobecnenie editačnej vzdialenosti pomocou ohodnoteného a-prekladača. V celej tejto kapitole sa budeme venovať tejto vzdialenosti. Najskôr uvedieme základné definície a algoritmus na vypočítanie prekladačovej vzdialenosti dvoch slov, potom prezentujeme algoritmus na vypočítanie vzdialenosti dvoch regulárnych jazykov z [Moh03], pričom rozoberieme niekoľko variantov ohodnotenia a-prekladača. Nakoniec uvedieme vlastný algoritmus na vypočítanie vzdialenosti regulárneho jazyka od bezkontextového jazyka.

Definícia 3.0.4 *Nech $T = (K, \Sigma_1, \Sigma_2, H, q_0, F)$ je a-prekladač a $C : H^* \to R \cup \{\infty\}$ je hodnotiaca funkcia, pričom platí: $C(uv) = C(u) + C(v)$. Nech pr_1, pr_2, pr_3, pr_4 sú homomorfizmy definované nasledovne: $pr_i((a_1, a_2, a_3, a_4)) = a_i, 1 \leq i \leq 4$.*

Potom cenu prekladu slova u na slovo v definujeme ako funkciu $C_T(u, v)$, pričom platia nasledovné vzťahy:

1. *Ak $v \in T(u)$, tak*

$$C_T(u,v) = \min\{C(w) | (w \in H^*) \wedge (pr_2(w) = u) \wedge (pr_3(w) = v) \wedge (pr_1(w_1) = q_0) \wedge$$
$$(pr_4(w_{|w|}) \in F) \wedge (\forall i, 1 \leq i < |w| : pr_4(w_i) = pr_1(w_{i+1}))\}$$

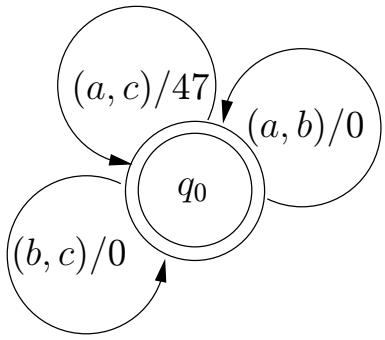
2. *Ak $v \notin T(u)$, tak*

$$C_T(u,v) = \infty$$

Hovoríme, že C_T je hodnotiaca funkcia prekladov a-prekladača T a hodnotiacej funkcie C.

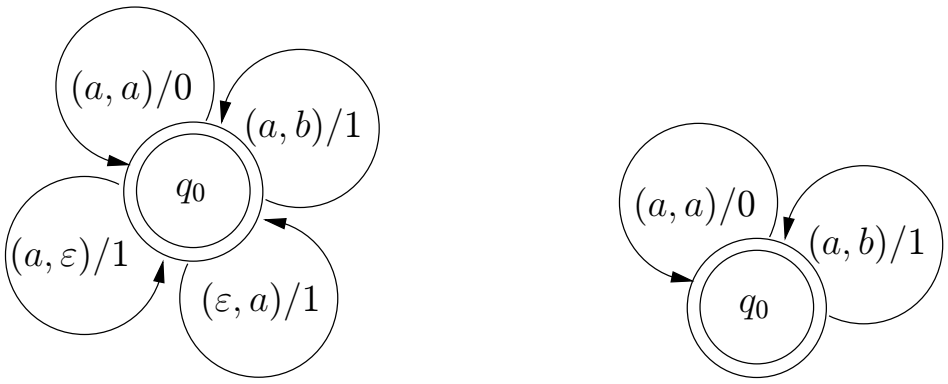
Definícia 3.0.5 *[Moh03] Nech T je a-prekladač s hodnotiacou funkciou C a nech u, w sú slová nad abecedou Σ. Potom* prekladačová vzdialenosť *slov u a v je $d_T(u, v) = C_T(u, v)$.*

Takto definovaná vzdialenosť v skutočnosti nie je vzdialenosť. Nie je ťažké vymyslieť stroj, ktorý nebude spĺňať trojuholníkovú nerovnosť: Nech $\Sigma = \{a, b, c\}$, a $d_T(a, b) = 0, d_T(b, c) = 0), d_T(a, c) = 47$. Taký stroj T ktorý toto spĺňa je na obrázku 3.1. Ten zodpovedá a-prekladaču s nasledovnou množinou $H = \{(q_0, a, c, q_0), (q_0, a, b, q_0), (q_0, b, c, q_0)\}$ a hodnotiacou funkciou $C((q_0, a, c, q_0)) = 47$, $C((q_0, a, b, q_0)) = 0$, $C((q_0, b, c, q_0)) = 0$.



Obr. 3.1: Príklad a-prekladača, ktorého vzdialenosť nespĺňa trojuholníkovú nerovnosť.

Samozrejme, dajú sa zostrojiť a-prekladače a ohodnotenia také, že d_T bude dobre definovaná vzdialenosť. Editačná vzdialenosť je v skutočnosti iba špeciálnym prípadom prekladačovej vzdialenosti. Stačí použiť nasledovný a-prekladač z obrázku 3.2. Na obrázku $a, b \in \Sigma$ a $a \neq b$. Výraz $(a, b)/x$ znamená, že a-prekladač načíta a, vypíše b a táto operácia ho stojí x.



Obr. 3.2: Automat pre editačnú vzdialenosť a automat pre Hammingovu vzdialenosť

Na obrázku 3.2 je taktiež automat, ktorý predstavuje Hammingovu vzdialenosť. Hammingova vzdialenosť je definovaná ako počet miest, kde sa slová líšia. Preto prirodzene nie

je definovaná na slovách rôznej dĺžky. Ďalšie prvky, ktoré sa dajú zakomponovať do prekladačovej vzdialenosti sú napríklad výmeny susedných písmen, zisťovanie, či slovo obsahuje nejaké iné slovo ako podslovo a podobne. Tiež sa dá spraviť automat, ktorý skontroluje, či slová nemajú len zamenené písmená alebo skupiny písmen. Ako vidíme, prekladačová vzdialenosť zastupuje širokú triedu vzdialeností definovaných na jazykoch.

Pre zjednodušenie ďalších konštrukcii si dokážeme nasledovnú lemu:

Lema 3.0.6 *Ku každému a-prekladaču* $T = (K, \Sigma_1, \Sigma_2, H, q_0, F)$ *existuje ekvivalentný a-prekladač taký, že* $H \subseteq K \times \Sigma_1 \times \Sigma_2 \times K \cup K \times \Sigma_1 \times \{\varepsilon\} \times K \cup K \times \{\varepsilon\} \times \Sigma_2 \times K$.

Dôkaz. Stačí si uvedomiť, že konštrukciu už máme napísanú v Leme 2.1.3. Je potrebné si všimnúť, že na a-prekladač sa dá pozrieť ako na nedeterministický konečný automat nad abecedou $\Sigma_E \cup \binom{\varepsilon}{\varepsilon}$, kde $\binom{\varepsilon}{\varepsilon}$ budeme považovať za ε. □

Odteraz budeme predpokladať, že každý a-prekladač s ktorým pracujeme bude tvare, o ktorom hovorí predchádzajúca lema.

3.1 Vzdialenosť dvoch slov

V tejto časti sa budeme venovať algoritmu na vypočítanie prekladačovej vzdialenosti dvoch slov. Nech $T = (K, \Sigma_1, \Sigma_2, H, q_0, F), K = \{q_0, q_1, \ldots, q_{|K|-1}\}$. Budeme postupovať podobne ako pri editačnej vzdialenosti, teda dynamickým programovaním. Postupne budeme počítať funkciu $D(i, j, k)$, ktorá hovorí aký existuje najlacnejší preklad slova $[u]_i$ na $[v]_j$ taký, že a-prekladač skončí v stave q_k (je nám jedno, či k-ty stav je akceptačný, alebo nie). Keďže a-prekladač v každom kroku načíta, alebo vypíše jedno písmeno, dá sa $D(i, j, k)$ vypočítať z $D(i-1, j-1, m), D(i-1, j, m), D(i, j-1, m), m \in K$ hľadaním minima.

Lema 3.1.1

(a) $D(0, 0, 0) = 0$

(b) $D(0, 0, k) = \infty, k > 0$

(c) $D(i, j, k) =$

$$\min\{C(t) + D(i', j', k') | t \in K, t = (p_{k'}, a, b, p_k), a, b \in \Sigma \cup \{\varepsilon\}, [u]_i = [u]_{i'}a, [v]_j = [v]_{j'}b\}$$

Dôkaz.

(a,b) Z lemy 3.0.6 vyplýva, že prázdne slovo preložíme na prázdne slovo iba tak, že nepoužijeme žiadne pravidlo. Preto ak T nič nenačítal, ani nevypísal, tak nemôže byť v inom, ako počiatočnom stave.

(c) Nech $m = \min\{C(t) + D(i', j', k') | t \in K, t = (p_{k'}, a, b, p_k), a, b \in \Sigma \cup \{\varepsilon\}, [u]_i = [u]_{i'}a, [v]_j = [v]_{j'}b\}$. To, že $D(i, j, k)$ je aspoň m vyplýva z toho, že $D(i, j, k)$ sleduje nejaký preklad slov u a v. To, že sleduje minimálny sa dá ukázať sporom. Nech $D(i, j, k)$ je menšie. Nech $t' = (k', a, b, k)$ je to pravidlo a-prekladača, ktoré bolo použité ako posledné pri danom preklade. Nech $[u]_i = [u]_{i'}a$ a $[v]_j = [v]_{j'}b$. Potom platí, že $D(i, j, k) = D(i', j', k') + C(t)$, čím sme sa dostali do sporu.

□

Poznámka: Iná možnosť, ako dokázať tvrdenie 3.1.1, je pozrieť sa na trojice (i, j, k) ako na vrcholy grafu a orientované a ohodnotené hrany budú tvoriť prechody a-prekladača. Stačí potom ukázať, že graf neobsahuje cyklus.

Algoritmus na vypočítanie prekladačovej vzdialenosti bude jednoduchý a podobný ako pri počítaní editačnej vzdialenosti. Vysvetlenie identifikátorov: *pravidla_tvaru(k,a,b,*)* je zoznam všetkých pravidiel a-prekladača T, ktoré vedú zo stavu k, načítajú a a vypíšu b . Pravidlá reprezentujeme ako pole so štyrmi prvkami (prvý index je 1).

```
1)  D(0,0,0)=0;
2)  for i=1 to |K|-1 do:
3)      D(0,0,i)=infty
4)  for i=0 to |v| do:
5)      for j=0 to |u| do:
6)          for k=0 to |K|-1 do:
7)              for l in pravidla_tvaru(k,u[i],v[j],*) do:
8)                  D(i+1,j+1,l[4])=min(D(i,j,k)+Psi(l),D(i+1,j+1,l[4]))
9)              for l in pravidla_tvaru(k,u[i],epsilon,*) do:
10)                 D(i+1,j,l[4])=min(D(i,j,k)+Psi(l),D(i+1,j,l[4]))
11)             for l in pravidla_tvaru(k,epsilon,v[j],*) do:
12)                 D(i,j+1,l[4])=min(D(i,j,k)+Psi(l),D(i,j+1,l[4]))
```

Algoritmus má časovú zložitosť $O(|u||v|(|K|+|H|))$. Pamäťová zložitosť je $O(|u||v||K|+|H|)$. V praktických aplikáciach budú dĺžky vstupných slov oveľa väčšie ako veľkosť a-prekladača, a preto v takých prípadoch môžeme veľkosť a-prekladača zanedbať. Pamäťová náročnosť algoritmu sa dá ešte vylepšiť a to na $O(|K| \min\{|u|, |v|\} + |H|)$ ak si uvedomíme, že si nepotrebujeme pamätať celú tabuľku D, ale iba „riadok" alebo „stĺpec" (podobne ako pri editačnej vzdialenosti).

3.2 Vzdialenosť dvoch regulárnych jazykov

V tejto časti sa budeme zaoberať algoritmom na vypočítanie prekladačovej vzdialenosti dvoch regulárnych jazykov. Pôvodne sa tento algoritmus vyskytol v [Moh03].

Budeme predpokladať, že obidva jazyky sú nad rovnakou abecedou. Nie je to nutné, ale uľahčí to vyjadrovanie.

Definícia 3.2.1 *Nech $L_1, L_2 \subseteq \Sigma^*$ a T je a-prekladač s ohodnotením C. Potom $d_T(L_1, L_2) = \min\{d_T(u, v) | u \in L_1 \wedge v \in L_2\}$. $d_T(L_1, L_2)$ budeme volať prekladačovou vzdialenosťou jazykov L_1 a L_2*

Teraz máme nasledujúcu úlohu: Nech L_1, L_2 sú regulárne automaty a $A_1 = (K_1, \Sigma, \delta_1, q_1, F_1)$, $A_2 = (K_2, \Sigma, \delta_2, q_2, F_2)$, pričom platí $L(A_1) = L_1$ a $L(A_2) = L_2$. Navyše nech $T = (K_T, \Sigma, \Sigma, H, q_0, F)$ je a-prekladač s hodnotiacou funkciou C. Hľadáme $d_T(L_1, L_2)$.

Budeme predpokladať, že A_1, A_2 sú v tvare podľa lemy 2.1.3 a T je v tvare podľa lemy 3.0.6.

Zavedieme nové označenie a to *prekladačovú abecedu*. Bude plniť rovnakú funkciu ako editačná abeceda.

Označenie 3.2.2 *Nech Σ je abeceda a T je a-prekladač s množinou stavov K. Potom množinu $\Sigma_T = \{\begin{pmatrix} a \\ b \\ p,q \end{pmatrix} | a, b \in \Sigma \cup \{\varepsilon\}, p, q \in K_T\} - \{\begin{pmatrix} \varepsilon \\ \varepsilon \\ p,q \end{pmatrix} | p, q \in K_T\}$ budeme nazývať* prekladačovou abecedou *nad abecedou Σ a prekladačom T.*

V skutočnosti je to iba iný zápis pravidiel a-prekladača, ale jednoduchšie v ňom vidno paralelu s editačnou vzdialenosťou[1].

Tak, ako v minulej kapitole, aj teraz si definujeme nasledovné homomorfizmy:

Definícia 3.2.3 *h_1, h_2, h_3, h_4 sú homomorfizmy definované nasledovne:*

$$h_1(\begin{pmatrix} a \\ b \\ p,q \end{pmatrix}) = a, h_2(\begin{pmatrix} a \\ b \\ p,q \end{pmatrix}) = b, h_3(\begin{pmatrix} a \\ b \\ p,q \end{pmatrix}) = p, h_4(\begin{pmatrix} a \\ b \\ p,q \end{pmatrix}) = q$$

$a, b \in \Sigma \cup \{\varepsilon\}, p, q \in K_T$

A taktiež si definujeme hodnotiacu funkciu prekladačovej abecedy.

Definícia 3.2.4

$$\Psi_C(\varepsilon) = 0, \Psi_C = \begin{pmatrix} a \\ b \\ p,q \end{pmatrix} = C((p, a, b, q)), \Psi_C(w) = \Sigma_{i=1}^{|w|} \Psi_C(w_i), w \in \Sigma_T^*$$

$\Psi_C(w)$ *budeme nazývať* cena *slova w.*

Tak ako sme si v prechádzajúcej kapitole definovali editačný jazyk, tak si tak isto definujeme prekladačový jazyk.

Definícia 3.2.5 *Nech $L_1, L_2 \subseteq \Sigma^*$ a $T = (K_T, \Sigma, \Sigma, H_T, q_T, F_T)$. Nech $w \in \Sigma_T^*$. Potom $w \in T_{A_2}^{A_1}$ práve vtedy, ak spĺňa nasledovné podmienky:*

[1] A hlavne je to čitateľnejšie.

1. $h_1(w) \in L_1, h_2(w) \in L_2$
2. $\forall 1 \leq i \leq |w| : (h_3(w_i), h_1(w_i), h_2(w_i), h_4(w_i)) \in H_T$,
3. $\forall 1 \leq i < |w| h_4(w_i) = h_3(w_{i+1})$
4. $h_3(w_1) = q_T, h_4(w_{|w|}) \in F_T$

Jazyk $T_{A_2}^{A_1}$ *budeme nazývať* prekladačový jazyk *jazykov* L_1, L_2 *a a-prekladača* T. *Slová, ktoré patria do* $T_{A_2}^{A_1}$ *budeme nazývať* prekladačové slová *jazykov* L_1, L_2 *a prekladača* T.

Všimnime si, že $w \in T_{A_2}^{A_1}$ nám jednoznačne popisuje nejaký preklad slova $h_1(w)$ na $h_2(w)$. Je to v skutočnosti iba postupnosť prvkov množiny H a-prekladača T v poradí, v akom sa použijú pri výpočte.

Dokážeme, že cena najlacnejšieho slova prekladačového jazyka je rovná vzdialenosti jazykov L_1 a L_2. Na to si ale potrebujeme najskôr dokázať nasledovnú lemu.

Lema 3.2.6 *Nech* $u \in L_1$ *a* $v \in L_2$ *a* $v \in T(u)$. *Potom existuje také slovo* $w \in T_{A_2}^{A_1}$, *že* $h_1(w) = u \wedge h_2(w) = v \wedge \Psi_C(w) = d(u, v)$ *a zároveň* w *bude prekladačové slovo.*

Dôkaz. Nech $\{p_i\}_{i=1}^{k}$ je tá najlacnejšia postupnosť stavov a-prekladača T pri preklade slova u na v. Nech $\{a_i\}_{i=1}^{k-1}$ sú tie prvky zo $\Sigma \cup \{\varepsilon\}$, ktoré sa čítali v i-tom kroku prekladu T a $\{b_i\}_{i=1}^{k-1}$ sú tie prvky, ktoré sa zapisovali. Nech

$$w = \begin{pmatrix} a_1 \\ b_1 \\ p_1, p_2 \end{pmatrix} \begin{pmatrix} a_2 \\ b_2 \\ p_2, p_3 \end{pmatrix} \cdots \begin{pmatrix} a_{k-1} \\ b_{k-1} \\ p_{k-1}, p_k \end{pmatrix}$$

Je zrejmé, že w spĺňa všetky podmienky na to, aby bolo prekladačové slovo. □

Veta 3.2.7 *Nech* L_1, L_2 *sú ľubovoľné jazyky nad abecedou* Σ *a nech* T *je a-prekladač a* $T_{A_2}^{A_1}$ *je k ním prislúchajúci prekladačový jazyk. Nech* $w \in T_{A_2}^{A_1}$ *a* $\forall v \in T_{A_2}^{A_1} : \Psi_C(w) \leq \Psi_C(v)$. *Potom* $\Psi_C(w) = d_T(L_1, L_2) = d_T(h_1(w), h_2(w))$.

Dôkaz. Každé slovo z jazyka $T_{A_2}^{A_1}$ zodpovedá nejakému korektnému[2] prekladu a-prekladača T a preto nutne $\Psi_C(w) \leq d_T(L_1, L_2)$.

Nech existujú $u' \in L_1, v' \in L_2$ také, že $d_T(L_1, L_2) = d_T(u', v') < \Psi_C(w)$. Potom ale podľa lemy 3.2.6 existuje také prekladačové slovo w', že $h_1(w') = u' \wedge h_2(w') = v' \wedge \Psi_C(w') = d_T(u', v')$, čo je spor s tým, že w je najlacnejšie slovo a teda $\Psi_C(w) = d_T(L_1, L_2) = d_T(h_1(w), h_2(w))$. □

[2]Priamo z definície.

Budeme pokračovať zostrojením automatu, ktorý bude generovať jazyk $T_{A_2}^{A_1}$. Automat má množinu stavov $K_1 \times K_T \times K_2$ a funguje podobne, ako automat z predchádzajúcej kapitoly, iba je rozšírený o stavy a-prekladača. Ak a-prekladač načíta aj vypíše písmeno, tak sa „posunieme" vo všetkých troch zložkách stavu (vykonáme krok výpočtu v A_1, T aj A_2). Ak a-prekladač iba načíta písmeno, tak sa „neposunieme" v poslednej zložke stavu (čo zodpovedá tomu, že automat A_2 stojí). Ak a-prekladač iba vypíše písmeno, tak stojí automat A_1. Formálne vyzerá konštrukcia nasledovne:

$B = (K_1 \times K_T \times K_2, \Sigma_T, \delta, (q_1, q_t, q_2), F_1 \times F_T, \times F_2)$, kde

$$(p,q,r) \in \delta((p',q',r'), \begin{pmatrix} a \\ b \\ q',q \end{pmatrix}) \Leftrightarrow (q',a,b,q) \in H_T \wedge p = \delta_1(p',a) \wedge r = \delta_2(r',b)$$

$$(p,q,r) \in \delta((p',q',r), \begin{pmatrix} a \\ \varepsilon \\ q',q \end{pmatrix}) \Leftrightarrow (q',a,\varepsilon,q) \in H_T \wedge p = \delta_1(p',a)$$

$$(p,q,r) \in \delta((p,q',r'), \begin{pmatrix} \varepsilon \\ b \\ q',q \end{pmatrix}) \Leftrightarrow (q',\varepsilon,b,q) \in H_T \wedge r = \delta_2(r',b)$$

kde $a, b \in \Sigma, p, p' \in K_1, q, q' \in K_T, r, r' \in K_2$.

Veta 3.2.8 $L(B) = T_{A_2}^{A_1}$

Dôkaz. Nech $w \in \Sigma_T^*, |w| = n, h_1(w) = u, h_2(w) = v$. Indukciou vzhľadom na n dokážeme nasledovné tvrdenie:

$$((q_1,q_t,q_2),w) \vdash_B^* ((p,q,r),\varepsilon) \Leftrightarrow (q_1,u) \vdash_{A_1}^* (p,\varepsilon) \wedge (q_2,v) \vdash_{A_2}^* (r,\varepsilon) \wedge (q_t,u,\varepsilon) \vdash_T^* (q,\varepsilon,u)$$

.

1. $n = 0$. Keďže A_1, A_2, T, B neobsahujú epsilonové prechody (pri T je to prechod čítajúci aj zapisujúci ε) a jediná možnosť, ako mať prázdne slovo w je neurobiť žiadny krok.

2. Indukčný predpoklad platí pre n. Nech $|w| = n+1, w' = [w]_{|w|-1}$ a $g = w_{|w|}, u' = h_1(w), v' = h_2(w), a = u_{|u|}, b = v_{|v|}$. Podľa indukčného predpokladu platí:

$$((q_1,q_t,q_2),w') \vdash_B^* ((p',q',r'),\varepsilon)$$

$$\Leftrightarrow$$

$$(q_1,u') \vdash_{A_1}^* (p',\varepsilon) \wedge (q_2,v') \vdash_{A_2}^* (r',\varepsilon) \wedge (q_t,u',\varepsilon) \vdash_T^* (q',\varepsilon,u')$$

Z konštrukcie automatu B vyplýva:

- $g = \begin{pmatrix} a \\ b \\ q', q \end{pmatrix} : (p,q,r) \in \delta((p',q',r'), \begin{pmatrix} a \\ b \\ q', q \end{pmatrix}) \Leftrightarrow (q',a,b,q) \in H_T \wedge p = \delta_1(p',a) \wedge r = \delta_2(r',b)$
- $g = \begin{pmatrix} a \\ \varepsilon \\ q', q \end{pmatrix} : (p,q,r) \in \delta((p',q',r), \begin{pmatrix} a \\ \varepsilon \\ q', q \end{pmatrix}) \Leftrightarrow (q',a,\varepsilon,q) \in H_T \wedge p = \delta_1(p',a)$
- $g = \begin{pmatrix} \varepsilon \\ b \\ q', q \end{pmatrix} : (p,q,r) \in \delta((p,q',r'), \begin{pmatrix} \varepsilon \\ b \\ q', q \end{pmatrix}) \Leftrightarrow (q',\varepsilon,b,q) \in H_T \wedge r = \delta_2(r',b)$

  A teda $((q_1,q_t,q_2),w) \vdash_B^* ((p,q,r),\varepsilon) \Leftrightarrow (q_1,u) \vdash_{A_1}^* (p,\varepsilon) \wedge (q_2,v) \vdash_{A_2}^* (r,\varepsilon) \wedge (q_t,u,\varepsilon) \vdash_T^* (q,\varepsilon,u)$

Teraz pristúpime k samotnému dôkazu vety:

- $L(B) \subseteq T_{A_2}^{A_1}$: Nech $w \in L(B)$. Podĺa predchádzajúceho tvrdenia $h_1(w) \in L_1 \wedge h_2(w) \in L_2 \wedge h_2(w) \in T(h_1(w))$. A teda $w \in T_{A_2}^{A_1}$.
- $L(B) \supseteq T_{A_2}^{A_1}$: Nech $w \in T_{A_2}^{A_1}$. Potom $h_1(w) \in L_1, h_2(w) \in L_2$ a $h_2(w) \in T(h_1(w))$. Použitím predchádzajúceho tvrdenia dostaneme $w \in L(B)$.

□

Ideme sa venovať hľadaniu najlacnejšieho výpočtu. Postupovať budeme podobne, ako v predchádzajúcej kapitole. Predstavíme si náš a-prekladač ako graf. Stavy budú vrcholy a prechody budú ohodnotené hrany. V takomto grafe nám stačí nájsť najkratšiu cestu z počiatočného stavu do nejakého akceptačného stavu. Tu sa nám však cesty rozdelia. Ak ohodnotenia prechodov a-prekladača sú rovnaké (ale kladné), postačí nám použiť prehľadávanie do šírky a máme algoritmus s lineárnou časovou zložitosťou od veľkosti grafu. Ak je ohodnotenie a-prekladača buď nula, alebo nejaká kladná konštanta, tak je možné použiť rovnaký algoritmus ako v predchádzajúcej kapitole. Ak sú ohodnotenia vrcholov kladné, použijeme Dijkstrov[CLR90] algoritmus a v opačnom prípade použijeme Bellman-Fordov[CLR90] algoritmus.

Graf má $|K_1||K_2||K_T|$ vrcholov a hrán je $O(|\delta_1||H_T||\delta_2|)$. Preto časové a pamäťové zložitosti algoritmov budú nasledovné: Pri odhade pamäťovej zložitosti predpokladáme, že si nepamätáme celý graf, ale hrany si vieme v konštantnom čase vypočítať (konštrukcia automatu B to umožňuje).

V prípade, že máme k dispozícii iba konečné jazyky, tak vieme zostrojiť algoritmus s lineárnou časovou zložitosťou pre ľubovoľné ohodnotenie. Treba si uvedomiť, že automat pre konečný jazyk tvorí acyklický orientovaný graf[3]. Z toho vyplýva, že automat B je tiež

[3]Ak obsahuje cyklus, tak do cyklu sa nedá dostať z počiatočného stavu, alebo sa z cyklu nedá dostať do akceptačného stavu a také cykly jednoducho z grafu odstránime.

Tabuľka 3.1: Časová zložitošť algoritmu

Algoritmus	**Časová zložitosť**	**Pamäťová zložitosť**
Prehľ. do šírky	$O(\|K_1\|\|K_2\|\|K_T\| + \|\delta_1\|\|H_T\|\|\delta_2\|)$	$O(\|K_1\|\|K_2\|\|K_T\| + \|H_T\| + \|\delta_1\| + \|\delta_2\|)$
Dijkstra	$O((\|K_1\|\|K_2\|\|K_T\|)^2)$	$O(\|K_1\|\|K_2\|\|K_T\| + \|H_T\| + \|\delta_1\| + \|\delta_2\|)$
Bellman-Ford	$O(\|K_1\|\|K_2\|\|K_T\|\|\delta_1\|\|H_T\|\|\delta_2\|)$	$O(\|K_1\|\|K_2\|\|K_T\| + \|H_T\| + \|\delta_1\| + \|\delta_2\|)$

acyklický orientovaný graf (každá hrana zodpovedá odvodeniu buď v oboch automatoch A_1, A_1, alebo aspoň v jednom z nich, pričom nezáleží, či T obsahuje cyklus alebo nie). Nájsť najkratšiu cestu v takom automate je už jednoduché. Stačí si vrcholy grafu topologicky usporiadať a potom s použitím dynamického programovania vypočítať najkratšiu cestu. Časová zložitosť takéhoto algoritmu je lineárna od veľkosti automatu B.

3.3 Vzdialenosť regulárneho od bezkontextového jazyka

V tejto časti uvedieme vlastný algoritmus na vypočítanie vzdialenosti regulárneho jazyka od bezkontextového jazyka. Algoritmus na vypočítanie editačnej vzdialenosti regulárneho jazyka od bezkontextového jazyka je špeciálnym prípadom tohto algoritmu.

Nech L_1 je regulárny jazyk, L_2 je bezkontextový jazyk a T je a-prekladač s hodnotiacou funkciou C. Našou úlohou je nájsť $d_T(L_1, L_2)$.

Nech $G = (N_G, \Sigma, P_G, \sigma_G)$ je bezkontextová gramatika $L(G) = L_2$ a $A = (K_A, \Sigma, \delta, q_A, F_A)$ je NKA $L(A) = L_1$. Nech $T = (K_T, \Sigma, \Sigma, H, q_T, F_T)$. Budeme predpokladať, že G je v tvare podľa lemy 2.2.1, A je v tvare podľa lemy 2.1.3 a T v tvare podľa lemy 3.0.6.

Teraz zostrojíme bezkontextovú gramatiku G_T takú, ktorá bude generovať jazyk $T^{A_1}_{A_2}$. Princíp konštrukcie je podobný ako v predchádzajúcej kapitole. V tomto prípade okrem stavu automatu je potrebné si v neterminái pamätať aj stav a-prekladača. Formálne konštrukcia vyzerá nasledovne:

Nech $G_T = (N, \Sigma_T, P, \sigma)$, kde

$$N = K_A \times K_T \times (N_G \cup \{\xi_\varepsilon\}) \times K_T \times K_A \cup \sigma$$

$$(p, q, A, r, s) \to (p, q, B, m, n)(n, m, C, r, s) \in P \Leftrightarrow A \to BC \in P_G \vee A = B = C = \xi_\varepsilon$$

$$\begin{aligned}
(p, q, A, r, s) &\to (p, q, A, m, n)(n, m, \xi_\varepsilon, r, s) \in P \\
(p, q, A, r, s) &\to (p, q, \xi_\varepsilon, m, n)(n, m, A, r, s) \in P \\
(p, q, A, r, s) &\to (p, q, B, r, s) \in P \Leftrightarrow A \to B \in P_G
\end{aligned}$$

$$(p,q,A,r,s) \rightarrow \begin{pmatrix} a \\ b \\ q,r \end{pmatrix} \in P \Leftrightarrow (q,a,b,r) \in H \wedge A \rightarrow b \in P_G \wedge s \in \delta(p,a)$$

$$(p,q,\xi_\varepsilon,r,s) \rightarrow \begin{pmatrix} a \\ \varepsilon \\ q,r \end{pmatrix} \in P \Leftrightarrow (q,a,\varepsilon,r) \in H \wedge s \in \delta(p,a)$$

$$(p,q,A,r,p) \rightarrow \begin{pmatrix} \varepsilon \\ b \\ q,r \end{pmatrix} \in P \Leftrightarrow (q,\varepsilon,b,r) \in H \wedge A \rightarrow b \in P_G$$

kde $p,s \in K_A, q,r \in K_T, A,B,C \in N_G \cup \{\xi_\varepsilon\}, a,b \in \Sigma$.

$$\begin{aligned} \sigma_G &\rightarrow \varepsilon \in P \Leftrightarrow \varepsilon \in L_2 \\ \sigma_G &\rightarrow (q_A,q_T,\sigma,p,q) \in P \Leftrightarrow p \in F_T \wedge q \in F_A\} \end{aligned}$$

Na to, aby sme si mohli pohodlne dokázať, že gramatika G_T generuje $T_{A_2}^{A_1}$, potrebujeme dokázať nasledujúce lemy.

Lema 3.3.1 *Nech $s \in N^*$. Ak $\sigma \Rightarrow_{G_T}^* s$, tak platia nasledovné podmienky:*

1. $s_1 = (q_A, q_T, A, p, q)$
2. $s_{|s|} = (p, q, A, q_{Tf}, q_{Af})$
3. $\forall 1 \leq i < |s| : s_i = (p,q,A,r,s) \wedge s_{i+1} = (s,r,B,q',p')$.

kde $q_{Af} \in F_A, q_{Tf} \in F_T, A, B \in N_G, p, p', s \in K_A, q, q', r \in K_T$

Dôkaz. Nech m je počet krokov odvodenia. Dôkaz urobíme matematickou indukciou vzhľadom na m.

1. Ak $m = 1$, tak $s = (q_A, q_T, \sigma_G, p, q), p \in F_T, q \in F_A$
2. Nech tvrdenie platí pre m a nech $\sigma_{G_T} \Rightarrow^m s' \Rightarrow_{G_T} s$ a s'_i je taký neterminál, ktorý sa použije v poslednom kroku odvodenia. Nech $s'_i = (p,q,A,r,s)$. Neterminál s'_i sa môže prepísať len na $(p,q,B,u,v)(v,u,C,r,s)$ alebo (p,q,B,r,s) a teda všetky podmienky lemy budú zachované.

□

Lema 3.3.2 *Nech $s \in N^*$. Nech h je homomorfizmus definovaný nasledovne:*

$$h(\begin{pmatrix} a \\ b \\ p,q \end{pmatrix}) = h_2(\begin{pmatrix} a \\ b \\ p,q \end{pmatrix}), h((p,q,A,r,s)) = A, A \in N_G, h(p,q,\xi_\varepsilon,r,s) = \varepsilon$$

Nech $s \in (N_G \cup T_G)^+, s \neq \sigma$. Potom ak $\sigma_{G_T} \Rightarrow^+ s$, tak $\sigma_G \Rightarrow^* h(s)$.

Dôkaz. Nech m je počet krokov odvodenia. Dôkaz spravíme matematickou indukciou vzhľadom na m.

1. Ak $m = 1$, tak $s = (q_A, q_T, \sigma_G, p, q), p \in F_T, q \in F_A$ a teda $h(s) = \sigma_G$ a teda platí, že $\sigma_G \Rightarrow^* h(s)$.

2. Nech tvrdenie platí pre m. Nech $\sigma \Rightarrow^m s' \Rightarrow s$. Keďže pre každé pravidlo $p \to r \in P$ platí, že
$$h(p) \to h(r) \in P_G \Leftrightarrow ((h(p) \neq \varepsilon) \vee (h(p) = h(r))),$$
tak nutne musí platiť, že $h(s') \Rightarrow_G h(s)$ alebo $h(s') = h(s)$ a z toho a z indukčného predpokladu vyplýva, že $\sigma_G \Rightarrow_G^* h(s)$.

□

Lema 3.3.3 *Nech $s \in N^+$, pričom platia nasledujúce podmienky:*

- $s_1 = (q_A, q_T, A, r, s)$
- $s_{|s|} = (p, q, B, f_T, f_A)$
- $\forall 1 \leq i < |s| : s_i = (p, q, A, r, s) \wedge s_{i+1} = (s, r, B, q', p')$

kde $f_T \in F_T, f_A \in F_A, p, p', s \in K_A, q, q', r \in K_T, A, B \in N_G$

Nech h je taký homomorfizmus, že $h((p, q, A, r, s)) = A, h((p, q, \xi_\varepsilon, r, s)) = \varepsilon$. Potom ak $\sigma_G \Rightarrow_G^* h(s)$ tak $\sigma \Rightarrow_{G_T}^* s$.

Dôkaz. Nech m je počet krokov odvodenia slova $h(s)$. Dôkaz urobíme matematickou indukciou vzhľadom na m.

1. Ak $m = 0$, tak $h(s) = \sigma_G$. Ale $\sigma \Rightarrow_{G_T} (q_A, q_T, \sigma_G, q_T, q_{Af})$ a teda tvrdenie platí.

2. Nech tvrdenie platí pre m. Nech $\sigma_G \Rightarrow_G^{m+1} h(s)$, Nech[4] $\sigma_G \Rightarrow_G^m h(s') \Rightarrow_G h(s)$. Rozdelíme to na dva prípady:

   (a) $h(s') \Rightarrow_G h(s)$ použití pravidla $A \to B$. Nech $A = h(s')_i$ a $B = h(s)_i$. Nech $s'' = us_i''v, s = us_iv, s_i = (p, q, B, r, s), s_i'' = (p, q, A, r, s)$. Je zrejmé, že s'' spĺňa predpoklady tejto lemy a $s'' \Rightarrow_{G_T} s$ a navyše $\sigma_G \Rightarrow_G^m h(s'')$ a teda z indukčného predpokladu $\sigma \Rightarrow_{G_T} s''$ a teda $\sigma \Rightarrow_{G_T}^* s$.

   (b) $h(s') \Rightarrow_G h(s)$ použití pravidla $A \to BC$. Položme $A = h(s')_i$, $B = h(s)_i$ a $C = h(s)_{i+1}$. Nech $s_i = (p, q, B, r', s')$ a $s_j = (t, o, C, r, s)$ a $\forall k, i < k < j$: $h(s_k) = \xi_\varepsilon$ (také musí existovať, lebo $h(s)$ sa dá odvodiť). Nech $s'' = us_i''v$, kde $s = us_is_{i+1} \cdots s_jv$ a $s_i'' = (p, q, A, r, s)$. Je zrejmé, že
   $$s'' \Rightarrow_{G_T} u(p, q, B, r'', s'')(s'', r'', C, r, s) \Rightarrow_{G_T}^* s$$
   (posledné odvodenie využije vygenerovanie neterminálu ξ_ε a „množenie" ξ_ε toľko krát, koľko treba) a $\sigma_G \Rightarrow_G^m h(s'')$. Podľa indukčného predpokladu $\sigma \Rightarrow_{G_T}^* s''$ a teda $\sigma \Rightarrow_{G_T}^* s$.□

[4]Treba si uvedomiť, že nemusí platiť $s' \Rightarrow_{G_T} s$

Teraz už máme dokázané všetko potrebné na to, aby sme vedeli pohodlne dokázať, že konštrukcia gramatiky G_T je správna (že $L(G_T) = T_{A_2}^{A_1}$).

Veta 3.3.4 $L(G_T) = T_{A_2}^{A_1}$

Dôkaz. Jednotlivé inklúzie si dokážeme oddelene.

- $L(G) \subseteq T_{A_2}^{A_1}$: Nech $w \in L(G)$. Potom podľa lemy 3.3.2 $h_2(w) \in L_2$.

  V bezkontextovej gramatike vieme preusporiadať kroky odvodenia tak, že prepisovanie na terminály sa použije až na konci. Nech $s \in N^*$ je taká vetná forma, že $\sigma \Rightarrow^* s \Rightarrow^* w$ a navyše pri odvodení slova w z vetnej formy s sa použijú len prepisovania na terminály. Nech $s_i = (p, q, A, r, s)$. Potom s_i sa prepíše na terminál

  $$\begin{pmatrix} a \\ b \\ q, r \end{pmatrix}, s \in \delta(p, a).$$

  Ak $s_i = (p, q, A, r, p)$ tak s_i sa prepíše na $\begin{pmatrix} \varepsilon \\ b \\ q, r \end{pmatrix}$. Ak teraz použijeme lemu 3.3.1, tak dostaneme, že $h_1(w) \in L_1$ a $h_3(w_i) = h_4(w_{i-1})$ a teda $w \in T_{A_2}^{A_1}$

- $L(G) \supseteq T_{A_2}^{A_1}$: Nech $w \in T_{A_2}^{A_1}$, $u = h_1(w), v = h_2(w)$. Zostrojíme strom odvodenia slova w. Zo slova w zostrojíme také slovo $s \in N^+$, aby spĺňal nasledovné podmienky:

  (a) $s \Rightarrow_{G_T}^* w$

  (b) $s_1 = (q_A, q_T, A, r, s) \wedge s_{|s|} = (p, q, B, f_T, f_A)$

  (c) $\forall 1 \leq i < |s| : s_i = (p, q, A, r, s) \wedge s_{i+1} = (s, r, B, q', p')$

  (d) $s_i \Rightarrow_{G_T} w_i$

  kde $p, p', s \in K_A, q, q', r \in K_T A, B \in N_G$

  Je zrejmé, že s určite existuje. Keďže $h_2(w) \in L_2$, tak s spĺňa všetky podmienky lemy 3.3.3 a teda $\sigma \Rightarrow_{G_T}^* s$ čo znamená, že $\sigma \Rightarrow_{G_T}^* w$.

□

Teraz nám už nič nebráni použiť algoritmus na nájdenie najlacnejšieho slova bezkontextovej gramatiky z časti 2.2 s jedinou zmenou: neterminál (p, q, A, r, s) budeme považovať za neterminál $((p, q), A, (s, r))$. Keďže $|N| = |K_A|^2|K^T|^2||N_G|$ a $|P| = O(|K_A|^3|K_T|^3||P_G|)$, tak časová[5] zložitosť algoritmu je

$$O((|K_A|^2|K^T|^2||N_G| + |K_A|^3|K_T|^3||P_G|) \log(|K_A||K^T|||N_G|))$$

[5]Dostali sme ju iba dosadením veľkosti gramatiky G_T do časovej zložitosti algoritmu z predchádzajúcej kapitoly.

a pamäťová zložitosť algoritmu je $O(|K_A|^2|K_T|^2|N_G| + |N_G|^2)$.

Algoritmus z časti 2.2 funguje len v prípade, že ohodnotenie a-prekladača T je všade nezáporné. Ak ohodnotenie môže byť záporné, tak použijeme podobnú myšlienku, ako v Bellman-Fordovom[CLR90] algoritme. Najkôr sa pozrime na Bellman-Fordov algoritmus.

Bellman-Fordov algoritmus je jednoduchý. Nech V je množina vrcholov. Na začiatku má každý vrchol okrem počiatočného vrcholu vzdialenosť ∞, počiatočný vrchol má 0. Potom sa $|V| - 1$ krát pokúšame použiť každú hranu a teda skúsiť zlepšiť najlepšiu cestu z počiatočného vrcholu hrany do koncového vrcholu hrany použitím danej hrany. Ak sa na konci dá použitím nejakej hrany zmenšiť vzdialenosť nejakého vrcholu, tak graf obsahuje záporný cyklus. Dôkaz prečo to funguje je jednoduchý a dá sa nájsť v [CLR90]. Pseudokód algoritmu vyzerá nasledovne (E je množina hrán):

```
1)  for v in V do:
2)      v.distance=infty
3)  start.distance=0;
4)  for i=0 to |V|-2 do:
5)      for e in E do:
6)          e.to.distance=min{e.to.distance,e.from.distance+e.length}
7)  for e in E do:
8)      if e.to.distance>e.from.distance+e.length:
9)          return -1
```

Tento istý princíp použijeme aj pri zostrojovaní algoritmu. Samozrejme, aby existovalo najlacnejšie odvodenie, tak ani gramatika G_T nesmie obsahovať „záporný cyklus". „Záporný cyklus" je neterminál, ktorý vie vygenerovať nekonečne veľa neterminálov, ktoré vedia vygenerovať slovo so zápornou cenou. Toto bude vedieť náš algoritmus zistiť.

Nech u je najlacnejšie slovo, ktoré sa dá odvodiť z neterminálu A. Potom existuje odvodenie slova u také, že v žiadnej vetnej forme, ktorá vznikne pri odvádzaní slova u sa neterminál A nenachádza. Ak by sa tam totiž nachádzal, tak by sme mohli zobrať jeho posledný výskyt a slovo v, ktoré sa z neho vygenerovalo a nahradiť celé odvodenie odvodením toho slova. Ak v je drahšie, tak to znamená, že sme našli „záporný cyklus" (lebo posledný výskyt A by sme nahradili odvodením z prvého A, teda namiesto v by sme odvodili znova u a dostali by sme ešte lacnejšie slovo). Z toho vyplýva, že ak tam nemáme „záporný cyklus", tak to slovo v musí byť rovnako lacné, alebo lacnejšie. A teda existuje strom odvodenia, v ktorom nie sú dva rovnaké neterminály pod sebou a teda nemá väčšiu hĺbku ako $|N|$.

Nasleduje presudokód algoritmu. Vysvetlivky ku kódu: Pravidlá sú reprezentované dvojicou. Ak p je pravidlo, tak*p.left* je neterminál na ľavej strane, *p.right* je pole neterminálov na pravej strane pravidla (indexované od 0). *N[0]* je počiatočný neterminál. Každý neterminál má navyše vlastnosť *n.dist*, ktorá obsahuje cenu najlacnejšieho slova, aké vieme z neho zatiaľ odvodiť. Psi je funkcia Ψ_C.

```
1)  for n in N do:
2)      n.dist=infty
3)  for p in P do:
4)      if p.right[0] is terminal:
5)          p.left.dist=Psi(p.right[0])
6)  for i=1 to |N|-1 do:
7)      for p in P do:
8)          if p.right.size=2:
9)              p.left.dist=min{p.left.dist,p.right[0].dist+p.right[1].dist}
10)         else:
11)             p.left.dist=min{p.left.dist,p.right[0].dist}
12) for p in P do:
13)     if p.right.size=2:
14)         if p.left.dist>p.right[0].dist+p.right[1].dist:
15)             return NEGATIVE_CYCLE
16)     else:
17)         if p.left.dist>p.right[0].dist:
18)             return NEGATIVE_CYCLE
19) return N[0].dist
```

Prvý cyklus nastaví každému neterminálu, že nevie odvodiť žiadne slovo. Druhý cyklus nájde najlacnejšie odvodenie pre neterminály, ktoré vedia priamo odvodiť nejaký terminál. Cyklus od riadku 6 po riadok 11 v každej iterácii vypočíta pre každý neterminál, aké je najlacnejšie odvodenie z neho so stromom odvodenia s maximálnou hĺbkou $i + 1$. Posledný cyklus kontroluje, či sa v gramatike nenachádza záporný cyklus. Časová zložitosť algoritmu je $O(|N||P|)$ a dá sa implementovať tak, aby mal pamäťovú zložitosť $O(|N| + |P|)$. Celková časová zložitosť algoritmu je $O(|K_A|^5|K_T|^5|P_G|)$ a pamäťová zložitosť je $O(|K_A|^2|K^T|^2||N_G| + |K_A|^3|K_T|^3||P_G|)$.

V tejto aj predchádzajúcej kapitole, sme hľadali vzdialenosť regulárneho jazyka od bezkontextového jazyka. Ak uvažujeme prekladačovú vzdialenosť, tak tá nemusí byť symetrická a v takom prípade, by konštrukcia gramatiky G_T vyzerala ináč a to tak, že na hornom poschodí by sa generoval bezkontextový jazyk a na strednom poschodí by sa generoval kontextový jazyk.

Kapitola 4

Δ-podobnosť

Δ-podobnosť na rozdiel od predchádzajúcich vzdialeností zohľadňuje aj dĺžku slov a teda slová s dĺžkou 100 000 líšiace sa v desiatich znakoch sú podobnejšie ako slová *bb* a *cc*, ktoré sú zase bližšie napríklad pri editačnej vzdialenosti. Pôvodne bola Δ-podobnosť definovaná iným spôsobom, ako v tejto práci, ale pôvodná definícia je zložitejšia. Dôkaz ekvivalentnosti definícii sa dá nájsť napríklad v [Kul06]. V tej istej práci sa dá tiež nájsť algoritmus na vypočítanie Δ-podobnosti slova a bezkontextového jazyka. V tejto kapitole stručne predstavíme algoritmus na vypočítanie Δ-podobnosti dvoch slov a predstavíme dva vlastné algoritmy, na vypočítanie Δ-podobnosti dvoch regulárnych jazykov.

Definícia 4.0.5 *Nech Σ je konečná abeceda a nech $u, v \in \Sigma^*$. Potom w je* najdlhšia spoločná podpostupnosť *slov u a v práve vtedy keď existujú také indexy $i_1, i_2, \cdots, i_{|w|}$ a také indexy $j_1, j_2, \cdots, j_{|w|}$, že spĺňajú nasledovné podmienky:*

1. $\forall k, 1 \leq k \leq |w| : 1 \leq i_k \leq |u| \wedge 1 \leq j_k \leq |v|$
2. $\forall k, 1 \leq k \leq |w| : w_k = u_{i_k} = v_{j_k}$
3. $\forall k, 1 \leq k < |w| : i_k < i_{k+1} \wedge j_k < j_{k+1}$.

Najdlhšiu spoločnú podpostupnosť slov u a v budeme označovať $LCS(u, v)$.

Definícia 4.0.6 *[Kul06] Nech Σ je konečná abeceda a nech $u, v \in \Sigma^*$. Potom*

$$\Delta(u, v) = \frac{2 \cdot LCS(u, v)}{|u| + |v|}$$

Hovoríme, že číslo $\Delta(u, v)$ je Δ podobnosť slov u a v.

Veta 4.0.7 *Δ-podobnosť je podobnosť definovaná na množine Σ^*.*

Dôkaz. Prvá podmienka je jednoduchá, $LCS(u, u) = |u|$, takže $\Delta(u, u) = 1$. Ak $u \neq v$, tak $LCS(u, v) \leq \max\{|u|, |v|\} - 1$ a teda $\Delta(u, v) < 0$.

Druhá podmienka vyplýva z toho, že $LCS(u, v) = LCS(v, u)$. □

4.1 Podobnosť dvoch slov

Ak vieme vypočítať $LCS(u, v)$, tak potom vieme vypočítať aj $\Delta(u, v)$. LCS sa dá vypočítať podobne ako editačná vzdialenosť.

Lema 4.1.1 *Nech $u, v \in \Sigma^*, |u| = m, |b| = n$. Potom*

1. $LCS(\varepsilon, \varepsilon) = 0$
2. $LCS(u, \varepsilon) = LCS(\varepsilon, u) = 0$
3. *Ak $u_m = v_n$, tak*
$$LCS(u, v) = \max\{LCS([u]_{m-1}, [v]_{n-1} + 1, LCS(u, [v]_{n-1}, LCS([u]_{m-1}, v)\}$$
4. *Ak $u_m \neq v_n$, tak*
$$LCS(u, v) = \max\{LCS([u]_{m-1}, [v]_{n-1}, LCS(u, [v]_{n-1}, LCS([u]_{m-1}, v)\}$$

Dôkaz. Prvé dve rovnosti sú zrejmé.

Nech $u_m = v_n$. Je zrejmé, že $LCS(u, v) = LCS([u]_{m-1}, [v]_{n-1}) + 1$.
$LCS(u, [v]_{n-1})$ ani $LCS([u]_{m-1}, v)$ nemôže byť väčšie (Sporom: nech je to väčšie a je to K, Potom $K - 1 \leq LCS([u]_{m-1}, [v]_{n-1})$ a potom $K \geq LCS([u]_{m-1}, [v]_{n-1}) + 1$).

Ak $u_m \neq v_n$, tak $LCS(u, v) = \max\{LCS(u, [v]_{n-1}), LCS([u]_{m-1}, v)\}$, vyplýva to z definície LCS. □

Táto lema nám priamo dáva návod ako vypočítať LCS v čase $O(|u||v|)$ a pamäti $O(|u|+|v|)$ rovnakým spôsobom, ako pri editačnej vzdialenosti. Existuje však lepší algoritmus, s ktorým sa dá vypočítať LCS rýchlejšie v očakávanom čase $(O(|u| + |v|) \log(|u| + |v|))$. Algoritmus sa prvý krát objavil v článku [HS77].

4.2 Podobnosť dvoch regulárnych jazykov

V tejto časti budeme prezentovať dva vlastné algoritmy na vypočítanie Δ-podobnosti. Jeden vypočíta Δ-podobnosť presne a druhý, ktorý bude rýchlejší, bude vedieť vypočítať Δ-podobnosť s ľubovoľnou presnosťou.

Definícia 4.2.1 *Nech Σ je konečná abeceda a nech $L_1, L_2 \in \Sigma^*$. Potom $\Delta(L_1, L_2) = \sup\{\Delta(u, v) | u \in L_1, v \in L_2\}$.*

Je dobré si všimnúť, že použitie maxima nie je možné. Môže sa totiž stať, že maximálny prvok neexistuje. Zoberme si nasledovný príklad: $L_1 = \{a^n | n \in N\}$, $L_2 = \{a^n b | \in N\}$. Je zrejmé, že pre každú dvojicu slov $u \in L_1, v \in L_2$ platí, že $\Delta(u, v) < 1$. Presnejšie

$$\Delta(a^n, a^m b) = \frac{2 \cdot \min\{m, n\}}{m + n + 1}$$

a teda menovateľ je vždy aspoň o jedna väčší ako čitateľ. Pre ľubovoľné $0 < k < 1$ sa dajú nájsť také slová $u \in L_1, v \in L_2$, že $\Delta(u,v) > k$. Napríklad ak $u = a^l, v = a^l b, l > \frac{k}{2-2k}$. Toto je dôvod, prečo musíme použiť suprémum a nie maximum.

$$\begin{aligned}\Delta(u,v) &= \frac{2 \cdot LCS(u,v)}{|u|+|v|} = \frac{|u|+|v|-(|u|+|v|)+2 \cdot LCS(u,v)}{|u|+|v|} \\ &= 1 - \frac{|u|+|v|-2 \cdot LCS(u,v)}{|u|+|v|}\end{aligned}$$

Nech

$$D(u,v) = \frac{|u|+|v|-2 \cdot LCS(u,v)}{|u|+|v|}$$

Keďže chceme, aby $\Delta(u,v)$ bolo čo najväčšie, tak $D(u,v)$ chceme mať čo najmenšie. Výraz v čitateli je $d_T(u,v)$, ak za T zoberieme nasledovný a-prekladač: $T = (\{q\}, \Sigma, \Sigma, H, q, \{q\})$, kde

$$H = \{(q,a,b,q) | a \in \Sigma\{\varepsilon\}, (a = b \wedge a \neq \varepsilon) \vee (a = \varepsilon \wedge b \neq \varepsilon) \vee (a \neq \varepsilon \wedge b = \varepsilon)\}$$

T je podobný ako editačná vzdialenosť, ale nepovoľujeme operáciu zmeny písmena. Ako hodnotiacu funkciu použijeme nasledovnú funkciu $C, C((q,a,a,q)) = 0, C((q,a,b,q)) = 1, a, b \in \Sigma \cup \{\varepsilon\}, a \neq b$. Na prvý pohľad by sa mohlo zdať, že už máme všetko vyriešené, ale nie je to tak. Totiž dve slová, ktoré majú najmenšiu vzdialenosť, ešte nie nutne musia mať najväčšiu Δ-podobnosť (ako príklad uvediem jazyky $\{a, a^{100\ 000}\}, \{b, a^{99\ 999}b\}$. Najbližšie slová z pohľadu prekladačovej vzdialenosti sú a a b, ale z pohľadu Δ-podobnosti to sú najpodobnejšie slová $a^{100\ 000}$ a $a^{99\ 999}b$). Napriek tomu si pomôžeme kapitolou 3.2.

Nech B je automat definovaný k automatom $A_1, A_2, L(A_i) = L_i$ a prekladaču T s ohodnotení C tak, ako v kapitole 3.2 (z tejto kapitoly tiež preberieme všetky ostatné označenia). Potom hľadáme také slovo $w \in L(B)$, že

$$\frac{\Psi_C(w)}{|h_1(w)| + |h_2(w)|}$$

je minimálne možné, a ak také slovo neexistuje, tak hľadáme aspoň

$$\inf\{\frac{\Psi_C(w)}{|h_1(w)| + |h_2(w)|}; w \in L(B)\}.$$

Z automatu B si zostrojíme graf rovnakým postupom, ako v kapitole 3.2 s jedinou výnimkou. Hrany z vrcholu x do vrcholu y, ktoré vygenerujú písmená tvaru $\binom{a}{a}$ rozdelíme na dve. Prvá bude viesť z vrcholu x do vrcholu y' (y' je nový vrchol) a bude mať cenu 0 a druhá hrana s nulovou cenou bude viesť z y' do y. Týmto sme dosiahli to, že dĺžka ľubovoľného sledu, ktorý sleduje generovanie slova w sa rovná $|h_1(w)| + |h_2(w)|$, a teda chceme nájsť čo „najpriemernejší" sled.

Nech $G = (V, E)$ je orientovaný, ohodnotený graf a c je funkcia, ktorá hranám prideľuje cenu, q_0 je počiatočný vrchol a F je množina akceptačných vrcholov. Najkôr si definujeme priemer sledu.

Definícia 4.2.2 *Nech $s = v_0e_1v_1e_1 \cdots e_kv_k$ je sled grafu G.*

$$P(s) = \frac{\Sigma_{i=1}^{k} c(e_i)}{k+1}$$

budeme volať priemer sledu.

Označenie 4.2.3 *Odteraz zavediem nasledovnú terminológiu: dĺžka sledu znamená počet použitých hrán a cena sledu, znamená súčet cien na jednotlivých hranách sledu.*

Našou úlohou je nájsť nasledovné číslo:

$$P = \inf\{P(s) | s = v_0e_1v_1e_2 \cdots e_kv_k, (k > 1) \vee (k = 1 \wedge v_0 \neq v_1), v_i \in V, e_i \in E, v_0 = q_0, v_k \in F\}$$

Nasledovná lema hovorí, že ak sled s obsahuje cyklus s', ktorý má menší priemer ako priemer celého sledu, tak potom ak budeme v slede s cyklus s' opakovať, tak priemer sledov, ktoré takto vytvárame sa bude zmenšovať a blížiť k $P(s')$.

Lema 4.2.4 *Nech $s = v_0e_1v_1 \cdots e_kv_k$ a nech $v_i = v_j, i < j$. Nech $s' = v_ie_{i+1} \cdots e_jvj$. Nech $P(s) > P(s')$. Potom*

$$P(s') = \inf\{P(t_n) | t_n = v_0e_1 \cdots e_i s'^n e_{j+1} \cdots e_kv_k, n \geq 1\}$$

kde $s'^1 = v_ie_{i+1} \cdots e_jv_j$, $s'^{n+1} = s'^n e_{i+1}v_{i+1} \cdots e_j, v_j$.

Dôkaz. Nech $P(s) = \frac{C}{D}$ a nech $P(s') = \frac{C'}{D'}$ a $P(t_n) = \frac{C_n}{D_n}$. Najskôr dokážem, že $P(t_n) > P(t_{n+1})$ a potom že $\lim_{n\to\infty} P(t_n) = P(s')$.

Prvú časť dokážeme indukciou vzhľadom na n.

1. Ak $n = 1$, tak $t_1 = s$. $P(t_2) = \frac{C+C'}{D+D'}$.

$$\begin{aligned} \frac{C'}{D'} < \frac{C}{D} &\Rightarrow C'D < CD' \Rightarrow CD + C'D < CD + CD' \\ &\Rightarrow D(C + C') < C(D + D') \\ &\Rightarrow \frac{C + C'}{D + D'} < \frac{C}{D} \end{aligned}$$

2. Indukčný predpoklad platí pre n. $P(t_{n+1}) = \frac{C_n+C'}{D_n+D'}$.

$$\begin{aligned} \frac{C'}{D'} < \frac{C_n}{D_n} &\Rightarrow C'D_n < C_nD' \Rightarrow C_nD_n + C'D_n < C_nD_n + C_nD' \\ &\Rightarrow D_n(C_n + C') < C_n(D_n + D') \\ &\Rightarrow \frac{C_n + C'}{D_n + D'} < \frac{C_n}{D_n} \end{aligned}$$

Z predchádzajúceho vyplýva, že $P(t_n) = \frac{C+(n-1)C'}{D+(n-1)D'}$.
A teda $\lim_{n\to\infty} P(t_n) = \lim_{n\to\infty} \frac{C-C'+nC}{D-D'+nD} = \frac{C}{D}$. □

Veta 4.2.5 *Nech $A(x,y) = k$ kde k je priemer sledu z vrcholu x do vrcholu y v ktorom sa neopakujú vrcholy s výnimkou prvého a posledného vrcholu. Ak také k neexistuje, tak $A(x,y) = \infty$.*

Potom $P = \min M$, kde

$$M = \{A(q_0, q_f) | q_f \in F\} \cup \{A(x,x) | A(q_0,x) < \infty \wedge \exists q_f \in F : A(x,q_f) < \infty\}$$

Dôkaz. Je zrejmé, že $P \leq \min M$. Nech s je taký sled, že začína vo vrchole q_0 a končí v F. Ak s je cesta (neopakujú sa tam vrcholy), tak potom $s \in M$ a teda $P(s) \geq \min M$. Nech sa v slede s zopakuje vrchol x. Potom podľa lemy 4.2.4 $P(s) \geq A(x,x)$ a teda $P(s) \geq \min M$. □

Ostáva nám už iba vypočítať $A(x,y)$. Budeme postupne počítať $B_k(x,y)$ čo sa bude rovnať najlacnejšej ceste z x do y s dĺžkou k. Je zrejmé, že $A(x,y) = \min\{\frac{B_k(x,y)}{k+1} | 1 \leq K \leq |V|-1\}$. Stačí si pamätať B_k a B_{k-1} a v A najmenšie hodnoty, aké sme doteraz vypočítali. Dostaneme nasledovný algoritmus:

```
1)  for i=1 to |V| do:
2)      for j=1 to |V| do:
3)          B[0,i,j]=B[1,i,j]=A[i,j]=infty
4)  for e in E do:
5)      B[0,e.from,e.to]=A[e.from,e.to]=e.length
6)  for i=1 to |V|-1 do:
7)      for j=1 to |V| do:
8)          for k=1 to |V| do:
9)              B[i mod 2,j,k]=infty
10)     for e in E do:
11)         for i=1 to |V| do:
12)             B[i mod 2,i,e.to]=
13)                 min(B[i mod 2,i,e.to],B[(i+1)mod 2,i,e.from]+e.length)
14)             A[i.e.to]=min(a[i,e.to,B[i mod 2,i,e.to]/(i+1))
```

Algoritmus má časovú náročnosť $O(|V|(|V|^2 + |V||E|))$. Nájdenie infíma (podľa vety 4.2.5) je jednoduché a časová náročnosť je menšia ako vypočítanie matice A a preto sa celková časová náročnosť nezhorší. Pamäťová náročnosť bude $O(|V|^2 + |E|)$.

Ak nám stačí vedieť výsledok s nejakou presnosťou, tak sa dá použiť o niečo rýchlejší algoritmus.

Veta 4.2.6 *Nech $k \in < 0,1 >$. Nech G' je graf, ktorý vznikol z G odrátaním čísla k od ceny každej hrany. Graf obsahuje sled zápornej ceny začínajúci v začiatočnom vrchole a končiaci v akceptačnom vrchole práve vtedy, keď $k > P$.*

Dôkaz.

- „⇒" Nech s je sled z grafu G', vedúci zo začiatočného vrchola do akceptačného vrchola, ktorý má záporný priemer. V grafe G sled s musí byť $P(s) \geq P$ (z minimality P). Z toho ale vyplýva, že $k > P$.
- „⇐" Nech s je taký sled, že $P \leq P(s) < k$. Keď od každej hrany sledu odrátame k, tak dostaneme sled so záporným priemerom. □

Z predchádzajúcej vety vyplýva, že P môžeme binárne vyhľadávať[CLR90]. Jediné, čo potrebujeme zistiť, je či graf G' obsahuje záporný sled z počiatočného vrchola do akceptačného vrchola. Na to nám poslúži Bellman-Fordov algoritmus. Časová zložitosť tohto riešenia je $O(|V||E|\log(\frac{1}{\epsilon}))$, kde ϵ je požadovaná presnosť.

Tabuľka 4.1: Časová zložitošť algoritmu

Algoritmus	**Časová zložitosť**	**Pamäťová zložitosť**
Exaktný	$O(\|K_1\|^3\|K_2\|^3 + \|K_1\|^2\|K_2\|^2\|\delta_1\|\|\Sigma\|\|\delta_2\|)$	$O(\|K_1\|^2\|K_2\|^2 + \|\delta_1\|\|\Sigma\|\|\delta_2\|)$
Približný	$O(\|K_1\|\|K_2\|\|\delta_1\|\|\Sigma\|\|\delta_2\|\log(1/\epsilon))$	$O(\|K_1\|\|K_2\| + \|\delta_1\|\|\Sigma\|\|\delta_2\|)$

Pozrime na celkovú časovú zložitosť algoritmov. Na to potrebujeme vedieť, aký veľký je graf G. Graf G má $|K_1||K_2|$ vrcholov a $O(|\delta_1||H||\delta_2|)$ hrán. Veľkosť $|H| = O(\Sigma)$ a teda graf G má $O(|\delta_1||\Sigma||\delta_2|)$ hrán. Časové a pamäťové zložitosti algoritmov sú uvedené v tabuľke 4.1.

Na záver si ukážeme pseudokód posledného algoritmu. Pre zjednodušenie programu predpokladáme, že z každého vrcholu sa dá dostať do akceptačného vrcholu a že do každého vrcholu sa dá dostať z počiatočného vrcholu.

```
1)  (low,height)=(0,1)
2)  while height-low>epsilon*2:
3)      P=(low+height)/2
4)      negative=false
5)      for v in V do:
6)          v.distance=infty
7)      V[0].distance=0;
8)      for i=0 to |V|-2 do:
9)          for e in E do:
10)             e.to.distance=min{e.to.distance,e.from.distance+e.length-P}
11)     for e in E do:
12)         if e.to.distance>e.from.distance+e.length-P:
13)             negative=true
14)     for v in V do:
15)         if v in F and v.distance<0:
16)             negative=true
```

```
17)     if negative:
18)         height=P
19)     else:
20)         low=P
21) return (height+low)/2
```

Kapitola 5

Záver

V tejto práci sme sa venovali hlavne algoritmom na vypočítanie vzdialeností jazykov. Okrem prezentácie už známych algoritmov, sme priniesli sme nové algoritmy na vypočítanie editačnej a prekladačovej vzdialenosti regulárneho jazyka od bezkontextového jazyka. Vďaka tomu, že neexistuje algoritmus na vypočítanie vzdialenosti[1] regulárneho jazyka od kontextového jazyka a že neexistuje algoritmus na vypočítanie vzdialenosti dvoch bezkontextových jazykov, sme pokryli všetky triedy Chomského hierarchie jazykov.

Venovali sme sa jednotlivým variantom algoritmu na vypočítanie prekladačovej vzdialenosti dvoch regulárnych jazykov. Varianty sa odlišovali vlastnosťami hodnotiacej funkcie a dosiahli sme tým v niektorých prípadoch zlepšenie časovej zložitosti. Navyše sme sa venovali aj úprave algoritmu na vypočítanie prekladačovej vzdialenosti, ak vieme, že na vstupe máme iba konečné jazyky. Uviedli sme dva nové algoritmy, na vypočítanie Δ-podobnosti dvoch regulárnych jazykov.

Práca odkryla niekoľko zaujímavých otázok, ktorými by sa dalo v budúcnosti zaoberať. Jednou z nich je, ako rozšíriť algoritmus, ktorý vypočíta Δ-podobnosť dvoch regulárnych jazykov na algoritmus, ktorý vypočíta Δ-podobnosť regulárneho jazyka a bezkontextového jazyka. Ďalšia otázka je, aká je očakávaná vzdialenosť jazykov náhodne vygenerovaných konečných automatov. Našou hypotézou je, že pre rovnako veľké konečné automaty a editačnú vzdialenosť je očakávaná vzdialenosť pre dostatočne veľkú abecedu rovná nule. Zaujímavé sú tiež otázky, ako je to pri ostatných vzdialenostiach.

Ďalší smer, ktorým by sa dalo uberať, je zmeniť definíciu vzdialenosti jazykov tak, aby výsledná vzdialenosť nezávisela len od dvoch slov, ale od celých jazykov, alebo aspoň od nejakej významnej podmnožiny oboch jazykov. Týmto otázkam sa plánujeme venovať v ďalšom štúdiu.

[1] editačnej aj prekladačovej

Literatúra

[AB02] Timo Raita Abraham Bookstein, Vladimir A. Kulyukin. Generalized hamming distance. *Information Retrieval*, 5(4):353–375, 2002.

[Asp] Aspell home page. http://www.aspell.net.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill, 1990.

[HS77] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[JH79] J.D. Ullman J.E. Hopcroft. *Introduction to Automata Theory, Languages and Computation*. 1979.

[Knu97] Donald E. Knuth. *The Art of Computer Programming*. Volume 1. Addison-Wesley, 1997.

[Kul06] Tomáš Kulich. *The distances on words*. Master Thesis, Comenius University, 2006.

[Moh03] Mehryar Mohri. Edit-distance of weighted automata: General definitions and algorithms. *Int. J. Found. Comput. Sci.*, 14(6):957–982, 2003.

[MPR02] Mehryar Mohri, Fernando Pereira, and Michael Riley. Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88, 2002.