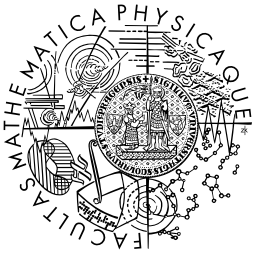


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

David Tvrdý

Lineární verze Holubova algoritmu

Katedra algebry

Vedoucí bakalářské práce: doc. Mgr. Štěpán Holub, Ph.D.

Studijní program: Matematika

Studijní obor: Matematika pro informační technologie

Praha 2020

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V Praze dne 4.6.2020 David Tvrdý

Na tomto místě bych rád poděkoval všem, kteří přispěli ke vzniku této práce, zejména pak svému vedoucímu Štěpánu Holubovi za poskytnuté konzultace, komentáře a připomínky.

Název práce: Lineární verze Holubova algoritmu

Autor: David Tvrdý

Katedra: Katedra algebry

Vedoucí bakalářské práce: doc. Mgr. Štěpán Holub, Ph.D., Katedra algebry

Abstrakt: Tato práce studuje lineární algoritmus, který pro zadané slovo rozhodne, zda existuje netriviální homomorfismus, jehož je dané slovo pevným bodem. Dále jsou v práci popsány pomocné datové struktury, které jsou pro lineární časovou složitost důležité. Součástí práce je i vlastní implementace tohoto algoritmu v jazyce Java včetně vizualizace chodu algoritmu pro konkrétní vstupy.

Klíčová slova: morfický rozklad, pevný bod, lineární algoritmus

Title: Linear version of Holub's algorithm

Author: David Tvrdý

Department: Department of Algebra

Supervisor: doc. Mgr. Štěpán Holub, Ph.D., Department of Algebra

Abstract: This work studies a linear agorithm which decides if a given word is a fixed point of any nontrivial morphism. This work also contains a description of auxiliary data structures which are crucial for linear time complexity of the algorithm. A Java implementation of the algorithm is provided along with a step-by-step visualization for particular input words.

Keywords: morphic factorization, fixed point, linear algorithm

Obsah

Úvod

Mějme konečnou abecedu Σ. Konečné posloupnosti prvků této abecedy, včetně prázdné posloupnosti, budeme nazývat slova. Na množině slov zavedeme strukturu monoidu s binární operací zřetězením a neutrálním prvkem prázdným slovem.

Dále mějme monoidový homomorfismus $h\colon \Sigma^* \to \Sigma^*$. Slovo $w \in \Sigma^*$ je *pevným bodem* homomorfismu h, pokud $h(w) = w$. Slovo $w \in \Sigma^*$ je *triviálním* pevným bodem, pokud $h(a) = a$ pro každý znak $a \in \Sigma$, který se ve slově w vyskytuje. Pokud pro každý homomorfismus takový, že slovo w je jeho pevným bodem, je slovo w jeho triviálním pevným bodem, pak řekneme, že slovo je *morficky primitivní*. Jinak řekneme, že slovo w je *morficky složené*.

Například, slovo *acacabab* je morficky složené, jelikož je netriviálním pevným bodem pro $h\colon h(c) = ac, h(b) = ab$. Naopak slovo *acacababa* je morficky primitivní.

První polynomiální algoritmus, který testuje, zda pro dané slovo w existuje homomorfismus takový, že w je jeho netriviálním pevným bodem, představil v roce 2009 Štěpán Holub ve svém článku [1]. Vojtěch Matocha ve své bakalářské práci [2] poté zlepšil horní odhad časové složitosti tohoto algoritmu na $O(mn)$ pro slovo délky n nad abecedou o velikost m.

Později T. Kociumaka, J. Radoszewski, W. Rytter a T. Waleń [3] představili novou verzi tohoto algoritmu, která má časovou složitost $O(n)$ pro slovo délky n nezávisle na velikosti abecedy. Tuto verzi algoritmu popíši spolu s datovými strukturami, které jsou pro lineární časovou složitost klíčové. Součástí práce je i přiložená vlastní implementace této verze algoritmu spolu s vizualizací chodu algoritmu pro konkrétní vstupy.

1. Morfický rozklad

Následující teorie je převzata z [3]. Pro přehlednost budeme také používat stejné značení, jaké je používáno v tomto článku.

V této kapitole si představíme základní definice spojené se slovem a pozicemi ve slově.

1.1 Klíčové znaky

Uvažujme slovo w délky n. Jednotlivé znaky ve slově budeme indexovat čísly od 1 (první znak) do n (poslední znak), tedy $w[i]$ bude značit i-tý znak ve slově a $w[i,j] = w[i]w[i+1]\dots w[j]$. Prázdné slovo budeme značit ϵ.

Definice 1 (Výskyty znaku ve slově)**.** *Množinu indexů slova w, na kterých se vyskytuje znak a, budeme značit* $\mathrm{Occ}(a,w)$ *nebo jednoduše* $\mathrm{Occ}(a)$*. Dále definujeme* $|w|_a = |\mathrm{Occ}(a,w)|$.

Množinu znaků abecedy Σ, které se ve slově w vyskytují, budeme značit $\mathrm{alph}(w)$.

Definice 2 (Klíčový znak)**.** *Mějme $F = (w_1,...,w_k)$ rozklad slova w, tedy $w = w_1 \dots w_k$ a buď $e \in \mathrm{alph}(w)$ znak slova w. Pokud se znak e v každém faktoru (rozkladu F) vyskytuje nejvýše jednou a pokud všechny faktory, ve kterých se znak e vyskytuje právě jednou, jsou shodné, pak řekneme, že e je* klíčový znak *(rozkladu F).*

Mějme e klíčový znak rozkladu F slova w. Takový faktor, že se v něm e vyskytuje právě jednou, budeme značit F_e. Všimněme si, že takový faktor je jednoznačný, jelikož všechny faktory, kde se e vyskytuje, jsou podle Definice 2 stejné. Zároveň, pro každý klíčový znak takový faktor existuje.

Definice 3 (Morfický rozklad)**.** *Buď $F = (w_1, \dots, w_k)$ rozklad slova w. Pokud každý faktor rozkladu F obsahuje alespoň jeden klíčový znak, pak řekneme, že F je* morfický rozklad *slova w.*

Definice 4 (Expanzní znak)**.** *Mějme rozklad F slova w a klíčový znak e. Pokud se ve faktoru F_e před výskytem znaku e nevyskytuje žádný jiný klíčový znak, pak řekneme, že e je* expanzní znak *(rozkladu F).*

Tedy, klíčový znak je expanzní, pokud se ve faktoru vyskytuje nejvíce nalevo ze všech klíčových znaků. Množinu expanzních znaků morfického rozkladu F slova w budeme značit $\mathcal{E}_F$. Nyní, pokud si vezmeme homomorfismus h splňující $h(e) = F_e$ pro $e \in \mathcal{E}_F$ a $h(a) = \epsilon$ pro $a \notin \mathcal{E}_F$, pak slovo w je pevným bodem tohoto homomorfismu.

Definice 5 (Standardní homomorfismus)**.** *Mějme nějaký morfický rozklad F slova w. Homomorfismus h splňující $h(e) = F_e$ pro všechna $e \in \mathcal{E}_F$ a $h(a) = \epsilon$ pro všechna $a \notin \mathcal{E}_F$ nazveme* standardním homomorfismem slova w (indukovaným rozkladem F).

Štěpán Holub ve svém článku dokázal ekvivalentní podmínku pro to, kdy je slovo morficky primitivní.

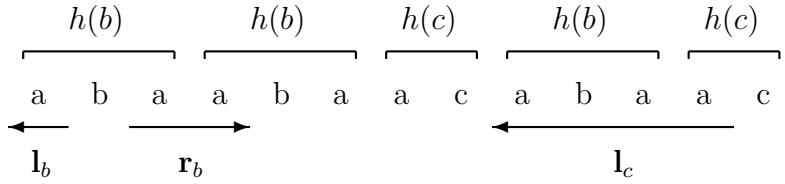
Tvrzení 1 (Věta 1 z [1])**.** *Slovo w je morficky složené právě tehdy, když existuje nějaký jeho netriviální morfický rozklad F, tedy když existuje nějaký jeho morfický rozklad, který má méně než $|w|$ faktorů.*

1.2 Okolí písmene

Definice 6 (Okolí písmene)**.** *Pro znak $a \in \mathrm{alph}(w)$ definujeme jeho* levé okolí $\mathbf{l}_a$ *ve slově w tak, že $\mathbf{l}_a a$ je nejdelší společný sufix všech prefixů slova w končících znakem a. Dále definujeme jeho* pravé okolí $\mathbf{r}_a$ *tak, že $a\mathbf{r}_a$ je nejdelší společný prefix všech sufixů slova w začínajících znakem a.*

Okolím znaku $\mathbf{n}_a$ *myslíme slovo $\mathbf{l}_a a \mathbf{r}_a$. Velikost okolí budeme značit*

$$n_a = |\mathbf{n}_a|,\ l_a = |\mathbf{l}_a|,\ r_a = |\mathbf{r}_a|.$$



Obrázek 1.1: Příklad zavedených pojmů. Pro slovo *abaabaacabaac* je $\mathbf{n}_a = a$, $\mathbf{n}_b = abaa$, $\mathbf{n}_c = abaac$. Slovo je morficky složené, je netriviálním pevným bodem homomorfismu $h(a) = \epsilon$, $h(b) = aba$, $h(c) = ac$.

Definice 7. *Buď $A = \mathrm{alph}(w[i+1, j])$. Pro $0 \le i < j \le n$ definujeme $\alpha(i, j)$ jako takový znak z A, který se ze všech znaků z A vyskytuje v celém slově w nejméněkrát. V případě nejednoznačnosti vybereme ten znak, který se ve slově $w[i+1, j]$ vyskytuje nejvíce vlevo.*

1.3 Pozice ve slově

V této sekci se budeme věnovat pozicím ve slově. Pozici ve slově si lze představovat jako mezeru mezi jednotlivými znaky, mezeru před slovem a mezeru za posledním znakem slova. Pro slovo w délky n budeme pozice ve slově číslovat od 0 (před prvním znakem) do n (za posledním znakem). Tedy i-tou pozicí ve slově myslíme mezeru za i-tým znakem slova.

Nyní si zadefinujeme dvě podmnožiny pozic ve slově, tzv. *levé* a *pravé* pozice.

Definice 8 (Levé a pravé pozice)**.** *Mějme nějaký morfický rozklad F slova w s expanzními znaky $\mathcal{E}_F$ a odpovídající standardní homomorfismus h.* Levé pozice *morfického rozkladu F definujeme jako $\mathcal{L}_F = \{i \colon |h(w[1, i])| \le i\}$.* Pravé pozice *morfického rozkladu F definujeme jako $\mathcal{R}_F = \{i \colon |h(w[1, i])| \ge i\}$.*

Definice 9. *Trojici (E, L, R) nazveme* korektní*, pokud pro všechny morfické rozklady F slova w platí $E \subseteq \mathcal{E}_F$, $L \subseteq \mathcal{L}_F$ a $R \subseteq \mathcal{R}_F$.*

2. Algoritmus

Popis algoritmu a důkazy Tvrzení 4 a 6 jsou převzaty z [3]. V této kapitole se budeme detailněji věnovat struktuře a fungování lineárního algoritmu.

2.1 Struktura algoritmu

Nejprve se podíváme na Holubovu verzi algoritmu.

Algoritmus 1: Holubův algoritmus

Inicializujme trojici množin $(E, L, R) = (\emptyset, \emptyset, \emptyset)$. Pomocí pravidel $(a) - (e)$ rozšiřujme množiny do té doby, dokud už žádné pravidlo trojici nezmění nebo dokud $E = \text{alph}(w)$.

(a) $L := L \cup \{0, n\}$; $R := R \cup \{0, n\}$
(b) **if** $w[i] \in E$ **then** $L := L \cup \{i - 1\}$; $R := R \cup \{i\}$
(c) **if** $w[i, j] = \mathbf{n}_a$ pro nějaké $a \in E$ **then** $R := R \cup \{i - 1\}$; $L := L \cup \{j\}$
(d) **if** $w[i, j] = w[i', j'] = \mathbf{n}_a$ pro nějaké $a \in E$ **then**
– **if** $i + k \in R$ pro nějaké $-1 \leq k \leq j - i$ **then** $R := R \cup \{i' + k\}$
– **if** $i + k \in L$ pro nějaké $-1 \leq k \leq j - i$ **then** $L := L \cup \{i' + k\}$
(e) **if** $i < j$ **and** $i \in L$ **and** $j \in R$ **then** $E := E \cup \{\alpha(i, j)\}$

if $E \neq \text{alph}(w)$ **then return** *w je morficky složené* **else return** *w je morficky primitivní*

Lemmata 4, 6, 7 a 8 z [1] dávají dohromady následující tvrzení.

Tvrzení 2 (Fakt 4 z [3])**.** *Pokud korektní trojici rozšíříme použitím jednoho z pravidel $(a) - (e)$ z Algoritmu 1, dostaneme opět korektní trojici.*

Pokud se tedy používáním pravidel $(a) - (e)$ dostaneme do situace, že $E = \text{alph}(w)$, tak podle Definice 9 jsou všechny znaky z $\text{alph}(w)$ expanzní pro všechny morfické rozklady slova w, a tedy w je podle Tvrzení 1 morficky primitivní.

Kociumaka, Radoszewski, Rytter a Waleń vylepšili Holubovu verzi algoritmu tak, aby počet pokusů o vkládání prvků do množin L a R odpovídal $O(n)$. Toho dosáhli změnou logiky operací $(a) - (e)$. To jim, spolu s použitím efektivních datových struktur, umožnilo vytvořit lineární verzi tohoto algoritmu.

Místo pravidel $(a) - (e)$ se v druhé verzi algoritmu používají nová pravidla $(a') - (e')$, která jsou podmnožinou pravidel $(a) - (e)$ postačující k sestavení funkčního algoritmu. K formulaci nových pravidel budeme ještě potřebovat definovat další funkce a zavést nová značení.

Definice 10 (Předchůdce a následník)**.** *Mějme A množinu celých čísel. Definujeme* předchůdce *v A a* následníka *v A jako* $\text{pred}_A(x) = \max\{y \colon y \in A, y < x\}$, $\text{succ}_A(x) = \min\{y \colon y \in A, y > x\}$. *Zároveň předpokládáme, že* $\min \emptyset = \infty$ *a* $\max \emptyset = -\infty$.

Pro *expanzní výskyty* $\mathrm{Occ}(E) = \{i \colon w[i] \in E\}$ budeme používat značení succ_E a pred_E namísto $\mathrm{succ}_{\mathrm{Occ}(E)}$ a $\mathrm{pred}_{\mathrm{Occ}(E)}$.

Hlavní vylepšení Holubova algoritmu spočívá ve zefektivnění pravidla (d). Toho je v druhé verzi algoritmu docíleno tím, že se „zmenší“ okolí expanzních znaků tak, aby nikdy nepřesahovala přes jiné expanzní znaky (formálně viz následující definice). Tak bude každá pozice ve slově ležet v dosahu maximálně dvou výskytů expanzních znaků. Navíc se pomocí pravidla (d') budou propagovat pouze prvky z R na vybraných pozicích.

Definice 11 (Dosah indexu)**.** *Pro* $i \in \mathrm{Occ}(E)$ *definujeme*

$$\mathrm{left}(i) = \min\{l_{w[i]}, i - \mathrm{pred}_E(i) - 1\},\ \ \mathrm{right}(i) = \min\{r_{w[i]}, \mathrm{succ}_E(i) - i - 1\}.$$

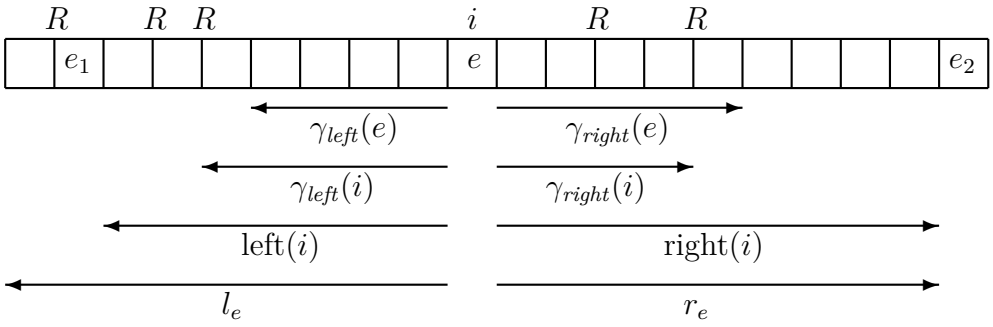
Definice 12. *Pro* $i \in \mathrm{Occ}(E)$ *definujeme*

$$\gamma_{left}(i) = i - \mathrm{pred}_R(i) - 1,\ \gamma_{right}(i) = \mathrm{pred}_R(i + \mathrm{right}(i) + 1) - i.$$

Definice 13. *Pro* $e \in E$ *definujeme*

$$\gamma_{left}(e) = \min\{\gamma_{left}(i) \colon i \in \mathrm{Occ}(e)\},\ \gamma_{right}(e) = \max\{\gamma_{right}(i) \colon i \in \mathrm{Occ}(e)\}.$$

Méně formálně, funkce left (resp. right) tedy zmenšuje okolí znaku tak, aby nepřesahovalo přes sousední expanzní znaky.
Hodnota $\gamma_{left}(i)$ (resp. $\gamma_{right}(i)$) říká, jak daleko je od indexu i ta pozice v množině R, která je nejvíce napravo v rámci levého (pravého) dosahu indexu i. Hodnota $\gamma_{left}(e)$ (resp. $\gamma_{right}(e)$) je pak minimum (maximum) přes všechny výskyty daného expanzního znaku e.



Obrázek 2.1: Ilustrace zavedeného značení.

Nyní si představíme pravidla $(a') - (e')$, která tvoří druhou verzi algoritmu, a ukážeme si, že pravidla $(a') - (e')$ skutečně tvoří podmnožinu pravidel $(a) - (e)$.

Algoritmus 2: Nová pravidla

Inicializujme trojici množin $(E, L, R) = (\emptyset, \emptyset, \emptyset)$. Pomocí pravidel $(a') - (e')$ rozšiřujme množiny do té doby, dokud už žádné pravidlo trojici nezmění.

(a') $L := L \cup \{0, n\}$; $R := R \cup \{0, n\}$
(b') **if** $w[i] \in E$ **then** $R := R \cup \{i\}$
(c') **if** $w[i] \in E$ **then** $R := R \cup \{i - \text{left}(i) - 1\}$; $L := L \cup \{i + \text{right}(i)\}$
(d') **if** $w[i] \in E$ **then** $R := R \cup \{i - 1 - \gamma_{left}(w[i]), i + \gamma_{right}(w[i])\}$
(e') **if** $i < j$ **and** $\text{succ}_R(i) = j$ **and** $\text{pred}_L(j) = i$ **and** $\{w[i+1], \ldots, w[j]\} \cap E = \emptyset$ **then** $E := E \cup \{\alpha(i, j)\}$

if $E \neq \text{alph}(w)$ **then return** *w je morficky složené* **else return** *w je morficky primitivní*

Tvrzení 3 (Fakt 5 z [3])**.** *Pokud korektní trojici rozšíříme použitím jednoho z pravidel $(a') - (e')$ z Algoritmu 2, dostaneme opět korektní trojici.*

Důkaz.
(a') provádí totéž, co (a).
(b') vynechává oproti (b) operaci vložení do množiny L.
(c') přidává do obou množin R a L jeden prvek, v některých případech stejný jako (c) (když $\text{left}(i) = l_{w[i]}$, respektive $\text{right}(i) = r_{w[i]}$), jindy jiný. Všimněme si, že v tom případě se ale do množin R, respektive L, přidává prvek, který do těchto množin patří podle pravidla (b).
(d') vkládá do R jen některé z pozic, které tam vkládá (d).
(e') klade náročnější podmínky než (e) na provedení stejné operace.

Tedy do množin E, L, R se přidají vlivem $(a') - (e')$ pouze některé z prvků, které by se tam přidaly vlivem $(a) - (e)$.

□

Opět, pokud prováděním pravidel $(a') - (e')$ nastane situace, že $E = \text{alph}(w)$, tak je slovo w morficky primitivní. Následující Tvrzení a Věta ověřují správnost Algoritmu 2 a ukazují, jak pomocí množin E, L, R získat morfický rozklad slova. Příklad takového rozkladu ukazuje Obrázek 2.2.

Tvrzení 4 (Lemma 6 v [3])**.** *Pokud běh Algoritmu 2 pro slovo w délky n skončí s trojicí (E, L, R), pak pro všechna i, j taková, že $1 \leq i < j \leq n$, $w[i] \in E$ a $j = \text{succ}_E(i)$, platí, že*

$$i + \gamma_{right}(w[i]) = j - 1 - \gamma_{left}(w[j]).$$

Důkaz. Nejprve si všimneme, že $\gamma_{right}(w[i])$ a $\gamma_{left}(w[j])$ jsou konečné. Hodnota $\gamma_{right}(w[i])$ je konečná, protože $i \in R$. Hodnota $\gamma_{left}(w[j])$ je konečná, protože pozice $j - 1 - \text{left}(j)$ leží v R.

Pro spor budeme nyní předpokládat, že pro nějaká i, j rovnost neplatí. Nejprve předpokládejme, že $i + \gamma_{right}(w[i]) < j - 1 - \gamma_{left}(w[j])$. Nechť $u = i + \text{right}(i)$ a $v = j - 1 - \gamma_{left}(w[j])$. Dostáváme dva případy. Pro $u < v$ můžeme provést operaci (e) na $u \in L$ a $v \in R$. Pokud zvolíme $u' = \text{pred}_L(v)$ a $v' = \text{succ}_R(u')$,

dostáváme dvojici pozic $u', v' \in [u,v]$, která nám umožňuje provést operaci (e'), tudíž běh algoritmu ještě neskončil. Pro $u \geq v$ je $v \in R$ ve sporu s definicí $\gamma_{right}(w[i])$.

Nyní předpokládejme, že $i+\gamma_{right}(w[i]) > j-1-\gamma_{left}(w[j])$. Pak pozice ve slově $i + \gamma_{right}(w[i]) \in R$ je ve sporu s definicí $\gamma_{left}(w[j])$.

□

Věta 5. *Algoritmus 2 správně rozhodne, zda je slovo w morficky primitivní. Navíc buď $F = (w_1, \ldots, w_{|\operatorname{Occ}(E)|})$, kde pro $j = 1, \ldots, |\operatorname{Occ}(E)|$ platí*

$$w_j = w[i_j - \gamma_{left}(w[i_j]), i_j + \gamma_{right}(w[i_j])],$$

kde i_j je j-tý nejmenší index z $\operatorname{Occ}(E)$. Pak F je morfický rozklad slova w.

Důkaz. Předpokládejme, že algoritmus skončil s trojicí (E, L, R).

Nejprve ukážeme, že F je opravdu rozklad slova w. Podle Tvrzení 4 na sebe každé dva takto popsané sousední faktory navazují, ale nepřekrývají se.
Pokud si vezmu nejmenší index i takový, že $w[i] \in E$, tak jelikož už není možné provést pravidlo (e'), tak $\operatorname{pred}_R(i) = 0$. Protože už nelze provést pravidlo (d'), tak $\gamma_{left}(w[i]) = \gamma_{left}(i) = i - 1$, a tedy $i - \gamma_{left}(w[i]) = 1$.
Podobně, pokud si vezmu největší index i takový, že $w[i] \in E$, tak jelikož už není možné provést pravidlo (e'), tak $i + \operatorname{right}(i) = n$ (protože $n \in L$ z pravidla (a') a $i+\operatorname{right}(i) \in L$ z pravidla (c')). A tedy $\gamma_{right}(w[i]) = \gamma_{right}(i) = \operatorname{pred}_R(n+1)-i = n - i$, a tedy $i + \gamma_{right}(w[i]) = i + n - i = n$.

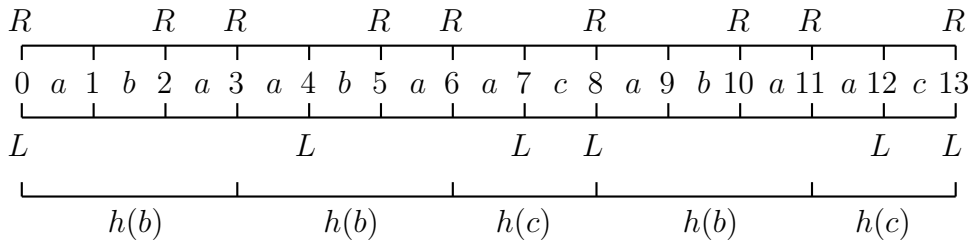
Nyní si ukážeme, že rozklad F je morfický. K tomu stačí ukázat, že znaky z E jsou klíčovými znaky rozkladu F. Mějme znak $e \in E$. Z definice $\gamma_{left}(w[i])$ (resp. $\gamma_{right}(w[i])$) pro $w[i] \in E$ plyne, že se znak e ve faktorech určených jiným znakem než e nevyskytuje.
Naopak, v každém faktoru rozkladu F určeném znakem e se znak e vyskytuje právě jednou, protože pro $w[i] \in E$ je $w[i - \gamma_{left}(w[i]), i + \gamma_{right}(w[i])]$ podslovo slova $w[i-l_{w[i]}, i+r_{w[i]}]$. Ze stejného důvodu jsou všechny faktory určené znakem e stejné.

Nakonec ověříme korektnost algoritmu. Pokud pro trojici (E, L, R) platí, že $|E| < |\operatorname{alph}(w)|$, pak výše popsaný morfický rozklad má méně než $|w|$ faktorů, a tedy podle Tvrzení 1 je slovo w morficky složené.
Z Tvrzení 3 víme, že (E, L, R) je korektní trojice. Pokud platí, že $|E| = |\operatorname{alph}(w)|$, tak podle Definice 9 pro všechny morfické rozklady F slova w platí $|\mathcal{E}_F| = |\operatorname{alph}(w)|$, a tedy každý morfický rozklad F slova w má $|w|$ faktorů. Tedy podle Tvrzení 1 je slovo w morficky primitivní.

□

2.2 Efektivní verze

V Algoritmu 2 jsou operace $(a') - (e')$ prováděny nedeterministicky, dokud se trojice (E, L, R) neustálí. Kociumaka, Radoszewski, Rytter a Waleń ve svém článku představili efektivní deterministickou implementaci. V této verzi algoritmu se pravidla uplatňují lokálně mezi sousedními expanzními znaky. Nové prvky



Obrázek 2.2: Algoritmus 2 pro slovo *abaabaacabaac* skončí s množinou $E = \{b, c\}$. Tedy slovo je morficky složené a $\gamma_{left}(b) = \gamma_{right}(b) = 1$, $\gamma_{left}(c) = 1$, $\gamma_{right}(c) = 0$. Podle Věty 5 získáváme homomorfismus $h\colon h(a) = \epsilon, h(b) = aba, h(c) = ac$.

vložené do množiny R ovlivňují své sousední expanzní znaky, které poté mohou generovat nové prvky R v dosahu jejich výskytů. Sousední expanzní znak pro pozici i se nachází na indexu $\text{pred}_E(i+1)$, resp. $\text{succ}_E(i)$.

Definice 14 (Volný interval)**.** *Mějme pozice i, j. Řekneme, že interval $[i, j]$ je* volný*, pokud můžeme na dvojici (i, j) aplikovat pravidlo (e') z Algoritmu 2.*

Volný interval tedy vždy dokáže vygenerovat nový expanzní znak $\alpha(i, j)$. Volný interval si můžeme představovat jako úsek mezi pozicí z L a pozicí z R, ve kterém se nevyskytuje žádná jiná pozice v L, žádná jiná pozice v R a žádný znak z E.

Popis efektivní verze algoritmu.

Délku vstupního slova w budeme značit n. Algoritmus začíná s $L = R = \{0, n\}$, $E = \emptyset$ (tedy bylo provedeno pravidlo (a')) a končí, pokud už nemůže být provedeno žádné z pravidel $(b') - (e')$.

Pro každý expanzní znak $e \in E$ si uchováváme aktuální hodnoty $\gamma_{left}(e)$, $\gamma_{right}(e)$, zatímco hodnoty $\gamma_{left}(i), \gamma_{right}(i)$, $\text{left}(i)$, $\text{right}(i)$ pro $i \in \text{Occ}(E)$ počítáme vždy, když potřebujeme, pomocí dotazů na předchůdce či následníky (více v sekcích 3.2 a 3.3). Pro každý expanzní znak $e \in E$ si také uchováváme množinu jeho výskytů $\mathit{ActiveSet}(e)$ (více v sekci 3.4), pro které by pravidlo (d') vygenerovalo nové pozice v R.

$$\mathit{ActiveSet}(e) = \{i \in \text{Occ}(e)\colon \gamma_{right}(i) < \gamma_{right}(e) \vee \gamma_{left}(i) > \gamma_{left}(e)\}.$$

Pravidlo (d') přidává pro $w[i] \in E$ do množiny R pozice $i - 1 - \gamma_{left}(w[i])$ a $i + \gamma_{right}(w[i])$, a tedy po provedení pravidla (d') na expanzní znak e platí

$$\forall i \in \text{Occ}(e)\colon \gamma_{right}(i) = \gamma_{right}(e) \wedge \gamma_{left}(i) = \gamma_{left}(e).$$

V proměnné *IntervalsSet* si udržujeme seznam volných intervalů (více v sekci 3.5). Upravujeme ho vždy po přidání prvků do E, L, R.

Jedna iterace Algoritmu 3 vygeneruje pomocí metody *ProcessInterval* jeden nový expanzní znak e (více v sekci 3.1). Tento znak vygeneruje podle pravidel (b') a (c') několik nových prvků R a L. Nově vložené prvky do R následně způsobují řetězovou reakcí další vkládání do R spojené s jinými expanzními znaky. Každý prvek vložený do R může díky pravidlu (d') ovlivnit všechny výskyty svých

expanzních sousedů; expanzní sousedy, kterých se to týká, vkládáme do fronty *LettersQueue*, která řetězovou reakci řídí. O vkládání znaků do fronty se stará metoda *Propagate*, která zároveň aktualizuje jejich hodnoty γ a zajišťuje, aby do fronty nebyly přidávány zbytečné znaky. Pokud pro expanzní znak e platí $\mathit{ActiveSet}(e) \neq \emptyset$, pak se určitě znak e vyskytuje ve frontě *LettersQueue*. Funkce *ProcessInterval* také hlídá, zda se nezměnil pravý dosah některého z levých expanzních sousedů vygenerovaného expanzního znaku.
Po proběhnutí metody *ProseccInterval* postupně odebíráme z fronty *LettersQueue* znaky e a do R přidáváme nové prvky způsobené pravidlem (d').

Algoritmus 3: Efektivní verze

$L := R := \{0, n\}$; $E := \emptyset$
$\mathit{IntervalsSet} := \{[0, n]\}$
while *IntervalsSet* not empty **do**
  buď $[i, j]$ jakýkoliv prvek z *IntervalsSet*
  $\mathit{NewR} :=$ ProcessInterval(i, j)
  foreach $k \in \mathit{NewR}$ **do** Propagate(k)
  while *LettersQueue* not empty **do**
    $e :=$ dequeue(*LettersQueue*)
    foreach $i \in \mathit{ActiveSet}(e)$ **do**
      $\mathit{NewR} := \{i + \gamma_{\mathit{right}}(e), i - 1 - \gamma_{\mathit{left}}(e)\} \setminus R$
      $R := R \cup \mathit{NewR}$
      foreach $k \in \mathit{NewR}$ **do** Propagate(k)
if $E \neq \mathrm{alph}(w)$ **then return** w je morficky složené **else return** w je morficky primitivní

Funkce Propagate(i)

$e_1 := w[\mathrm{pred}_E(i + 1)]$; $e_2 := w[\mathrm{succ}_E(i)]$
přidej e_1, e_2 do *LettersQueue*
pokud je to nutné, aktualizuj $\gamma_{\mathit{right}}(e_1)$ a $\gamma_{\mathit{left}}(e_2)$

Funkce ProcessInterval(i, j)

$e := \alpha(i, j)$; $E := E \cup \{e\}$; $\mathit{NewR} := \emptyset$
foreach $p \in \mathrm{Occ}(e)$ **do**
  $\mathit{NewR} := \mathit{NewR} \cup \{p, p - \mathrm{left}(p) - 1\}$
  $L := L \cup \{p + \mathrm{right}(p)\}$
$\mathit{NewR} := \mathit{NewR} \setminus R$; $R := R \cup \mathit{NewR}$
vypočítej $\gamma_{\mathit{right}}(e), \gamma_{\mathit{left}}(e)$; přidej e do *LettersQueue*
foreach $p \in \mathrm{Occ}(e)$ **do**
  $e' := w[\mathrm{pred}_E(p)]$
  if $\gamma_{\mathit{right}}(e') > \mathrm{right}(\mathrm{pred}_E(p))$ **then**
    $\gamma_{\mathit{right}}(e') := \max\{\mathrm{pred}_R(p' + \mathrm{right}(p') + 1) - p' : p' \in \mathrm{Occ}(e')\}$
return *NewR*

Tvrzení 6. *Algoritmus 3 správně rozhodne, zda je slovo morficky primitivní.*

Důkaz. Jedná se o první část důkazu Věty 8 v [3].

Pravidla (a'), (b') a (c') se v Algoritmu 3 aplikují co nejdříve. Pravidlo (e') nelze použít, pokud je množina *IntervalsSet* prázdná. Pravidlo (d') nelze použít, pokud je fronta *LettersQueue* prázdná. Tedy po skončení běhu Algoritmu 3 již nelze aplikovat žádné pravidlo $(a')-(e')$. Tudíž podle Věty 5 funguje Algoritmus 3 správně.

□

3. Pomocné datové struktury

V této kapitole si představíme pomocné datové struktury, které se v efektivní verzi algoritmu pro dosažení lineární časové složitosti používají. Problém Range Minimum Query, který se využívá také v sekci Lowest Common Prefix, zde popisuji detailněji než ostatní problémy. Ty jsou zde popsány podobně podrobně jako v článku [3].

3.1 Range Minimum Queries

Nejprve se budeme věnovat tomu, jak efektivně počítat $\alpha(i, j)$. V zadaném podslově slova w chceme najít znak, který se ve slově w vyskytuje nejméněkrát. Efektivní počítání $\alpha(i, j)$ odpovídá efektivnímu řešení problému RMQ (Range Minimum Query), kdy v zadaném intervalu pole porovnatelných objektů hledáme minimální prvek.

V našem případě nahradíme každý znak slova počtem jeho výskytů a budeme pracovat s takto vzniklou posloupností. Potřebovali bychom, aby $\alpha(i, j)$ bylo možné počítat pokaždé v konstantním čase po $O(n)$ předzpracování.

Definice 15. *Mějme posloupnost čísel* $A = (a_1, \ldots, a_n)$. *Pro* $1 \leq i \leq j \leq n$ *definujeme* $\mathrm{RMQ}_A(i, j) = \operatorname{argmin}\{a_i, \ldots, a_j\}$. *V případě nejednoznačnosti vybereme libovolný takový index.*

Například, pro $A = (5, 2, 3, 5, 6, 4, 1, 2, 7)$ platí $\mathrm{RMQ}_A(2, 8) = 7$. Pokud budeme v Algoritmu 3 používat RMQ k počítání $\alpha(i, j)$, tak musíme zajistit, aby výsledkem RMQ byl nejmenší takový index. Jak uvidíme v následující sekci, volba indexu v případě nejednoznačnosti závisí na tom, jakým způsobem zkonstruujeme odpovídající kartézský strom. Při konstrukci popsané v důkazu Tvrzení 7 například dostaneme vždy největší takový index, v tom případě můžeme najít i nejmenší takový index aplikací postupu na otočenou posloupnost.

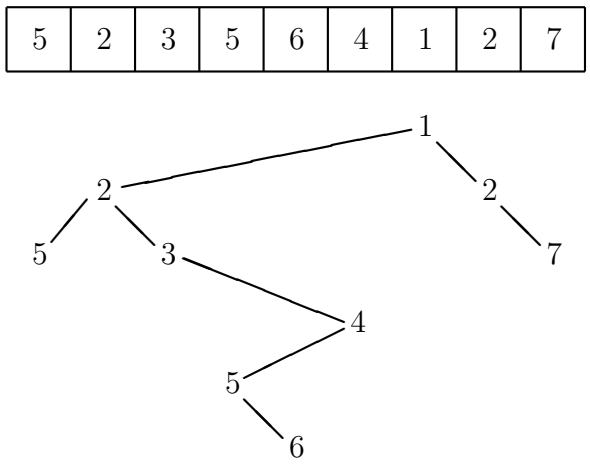
Následující postupy a důkazy jsou převzaty z [4] a [5].

3.1.1 Kartézský strom

Mějme posloupnost celých čísel. Kartézský strom této posloupnosti je binární strom mající následující vlastnosti.

1) Hodnota každého uzlu je větší nebo rovna hodnotě jeho rodiče.
2) Symetrickým (in-order) procházením stromu získáme původní posloupnost čísel.

Příklad takového stromu ukazuje Obrázek 3.1. Všimněme si, že pokud pro posloupnost čísel postavíme odpovídající kartézský strom, můžeme převést problém RMQ na jiný známý problém, a to na problém LCA (Lowest Common Ancestor — hledání nejnižšího společného předchůdce).



Obrázek 3.1: Kartézský strom pro pole $(5, 2, 3, 5, 6, 4, 1, 2, 7)$.

Definice 16 (LCA)**.** *Mějme binární strom C. Nechť u, v jsou nějaké jeho uzly. Potom* $\mathrm{LCA}_C(u, v)$ *je nejnižší uzel (tj. uzel nejdále od kořene), který má u i v jako své následníky.*

Tedy pro získání $\mathrm{RMQ}_A(i, j)$ nejprve postavíme pro posloupnost A kartézský strom C, a poté v C najdeme nejnižšího společného předchůdce uzlů odpovídajících indexům i, j v A. Bender a Farach-Colton ve svém článku [4] ukázali postup, jakým lze pro posloupnost čísel postavit v lineárním čase odpovídající kartézský strom a dokázali tím následující tvrzení.

Tvrzení 7 (Lemma 7 v [4])**.** *Pokud umíme řešit LCA problém v konstantním čase po $O(n)$ přípravě, umíme také řešit RMQ problém v konstantním čase po $O(n)$ přípravě.*

Důkaz. Mějme kartézský strom C_i pole $A[1, \ldots, i]$. Pro zkonstruování stromu C_{i+1} si všimneme, že nově přidávaný uzel $i + 1$ musí patřit do cesty nejvíce napravo od kořene stromu. Vezmeme tedy uzel $i + 1$ a postupným porovnáváním s uzly z nejvíce pravé cesty (nejprve s uzlem i, pak případně s jeho předchůdcem, a tak dále, až nakonec s uzlem, který má menší hodnotu než uzel $i + 1$ a je na nejvíce pravé cestě od kořene nejdále) zjistíme, kam uzel $i + 1$ patří. Každé takové porovnání s uzlem i' způsobí buď umístění uzlu $i + 1$ do nejvíce pravé cesty stromu (když hodnota uzlu $i+1$ je větší než hodnota uzlu i'), nebo odebrání uzlu i' z nejvíce pravé cesty (když hodnota uzlu $i + 1$ je menší nebo rovna hodnotě uzlu i'). Každý uzel navíc může být do nejvíce pravé cesty přidaný nejvýše jednou a může z ní být nejvýše jednou odebraný.

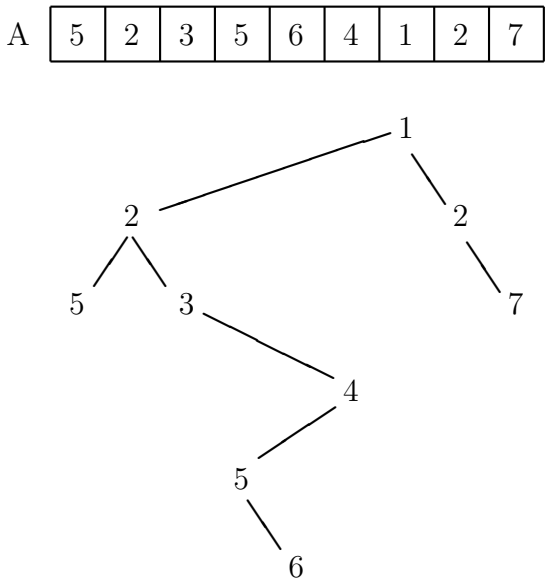
□

3.1.2 Farach-Coltonův a Benderův algoritmus

Bender a Farach-Colton ve svém článku [4] ukázali, jak lze problém LCA redukovat zpět na speciální případ problému RMQ. Dále prezentovali algoritmus, který tento speciální případ řeší. Nejprve se podíváme na redukci problému LCA

na speciální případ problému RMQ.
Mějme kartézský strom C s počtem uzlů n. Eulerova cesta stromem C je posloupnost uzlů stromu, kterou získáme, pokud si popořadě vypíšeme každý uzel, který navštívíme během prohledávání stromu do hloubky.

1) Vyrobíme pole $T[1, \ldots, 2n-1]$, které bude postupně skladovat uzly navštívené při Eulerově cestě stromem C.
2) Buď *úroveň* uzlu jeho vzdálenost od kořene stromu C. Vypočítáme pole úrovní $U[1, \ldots, 2n-1]$, kde $U[i]$ je úroveň uzlu $T[i]$.
3) Buď *reprezentant* uzlu v Eulerově cestě index jeho prvního výskytu v Eulerově cestě. Vypočítáme pole reprezentantů $V[1, \ldots, n]$, kde $V[i]$ je reprezentant uzlu i.



T	1	2	5	2	3	4	5	6	5	4	3	2	1	2	7	2	1
U	0	1	2	1	2	3	4	5	4	3	2	1	0	1	2	1	0
V	3	2	5	7	8	6	1	14	15								

Obrázek 3.2: Kartézský strom pro pole $A = (5, 2, 3, 5, 6, 4, 1, 2, 7)$ a odpovídající pole T, pole úrovní U a pole reprezentantů V. Pole T formálně obsahuje uzly stromu, v obrázku jsou místo nich pro přehlednost zobrazeny jejich hodnoty.

Každý z těchto kroků má lineární časovou složitost. Pro získání hodnoty $\mathrm{LCA}_C(u, v)$ si všimneme následujících skutečností:

1) Uzly v Eulerově cestě mezi u a v jsou $T[V[u], \ldots, V[v]]$ (nebo naopak).
2) Index v T uzlu s nejmenší úrovní z uzlů v této části Eulerovy cesty je $\mathrm{RMQ}_U(V[u], V[v])$. Jedná se tedy o uzel $T[\mathrm{RMQ}_U(V[u], V[v])]$, což je výsledek $\mathrm{LCA}_C(u, v)$.

Klíčovou vlastností pole U je to, že dva sousední prvky se liší pouze o jednotku. To je způsobené tím, že pole U vzniklo procházením stromu do hloubky. Takto vzniklému speciálnímu případu problému RMQ budeme říkat problém ±1-RMQ.

Tvrzení 8 (Důsledek Lemmatu 3 v [4])**.** *Pokud umíme řešit problém ±1-RMQ v konstantním čase s lineární přípravou, potom umíme řešit i problém LCA v konstantním čase s lineární přípravou.*

Důsledek. A tedy podle Tvrzení 7 umíme řešit i původní problém RMQ v konstantním čase s lineární přípravou.

Nyní se posuneme k tomu, jak řešit zmiňovaný problém ±1-RMQ pro naše pole U. Budeme používat tabulku ST (Sparse Table). Pro pole $X[1, \ldots, m]$ platí

$$ST[i][j] = \text{argmin}\,\{X[i], X[i+1], \ldots, X[i+2^j-1]\}.$$

Velikost této tabulky je $O(m \log m)$ a lze jí postavit v čase $O(m \log m)$ tak, že $ST[i][j] = ST[i][j-1]$, pokud $X[ST[i][j-1]] \leq X[ST[i+2^{j-1}][j-1]]$, a $ST[i][j] = ST[i+2^{j-1}][j-1]$ jinak.

Nejprve si rozdělíme naše pole U do bloků o velikosti $\log(N)/2$, kde N je velikost pole U. Definujeme pole $A'[1, \ldots, 2N/\log N]$, kde $A'[i]$ je minimální prvek i-tého bloku pole U. Dále definujeme pole $B[1, \ldots, 2N/\log N]$, kde $B[i]$ je index v i-tém bloku pole U, na kterém se hodnota $A'[i]$ vyskytuje. Pro A' nyní můžeme vytvořit tabulku ST v čase $O(N)$.

Nyní se podívejme na to, jak získat výsledek $\text{RMQ}_U(i, j)$. Indexy i, j můžou být buď ve stejném bloku tabulky U, nebo v jiných blocích tabulky U. Pokud $i < j$ jsou v jiných blocích tabulky, pak výsledek $\text{RMQ}_U(i, j)$ získáme jako index, na kterém se vyskytuje minimum z následujících hodnot:

1) Minimum mezi i a koncem jeho bloku.
2) Minimum z hodnot ve všech blocích mezi blokem, kam patří i, a blokem, kam patří j.
3) Minimum mezi j a začátkem jeho bloku.

Hodnotu 2) umíme získat v konstantním čase pomocí tabulky ST jako

$$\min\{A'[ST[i][k]], A'[ST[j-2^k+1][k]]\},$$

kde k je $\lfloor \log(j-i) \rfloor$, ale v každém případě budeme potřebovat umět ještě vypočítat minimální hodnotu uvnitř bloku.

Nyní můžeme využít toho, že pole U má výše zmiňovanou ±1 vlastnost. Každý blok pole U můžeme normalizovat, tj. odečíst od všech jeho prvků jeho první prvek. Tímto způsobem můžeme každý blok reprezentovat posloupností čísel ±1 o velikosti $\log(N)/2 - 1$ (protože první prvek po takové úpravě bude ve všech blocích 0). Počet takových možných posloupností je $2^{\log(N)/2-1} = \sqrt{N}/2$. Taková úprava bloku nám nezmění výsledky dotazů RMQ v něm provedených.

Vytvoříme tedy $O(\sqrt{N})$ tabulek, jednu pro každý možný normalizovaný blok, a do každé z nich umístíme všech $(\log(N)/2)^2 = O(\log^2(N))$ odpovědí na dotazy RMQ uvnitř bloku. Nakonec už jen pro každý blok pole U zjistíme, kterému normalizovanému bloku odpovídá. Dohromady tedy předpočítání všech minimálních hodnot uvnitř bloku bude mít časovou složitost $O(\sqrt{N}\log^2(N)) = O(N)$. Výše zmiňovaný postup vede k následujícímu tvrzení.

Tvrzení 9. *Problém ±1-RMQ lze řešit v konstantním čase po $O(n)$ předzpracování.*

Důsledek. Podle Důsledku Tvrzení 8 tedy umíme řešit problém RMQ v konstantním čase po lineárním předzpracování.

3.2 Longest Common Prefix

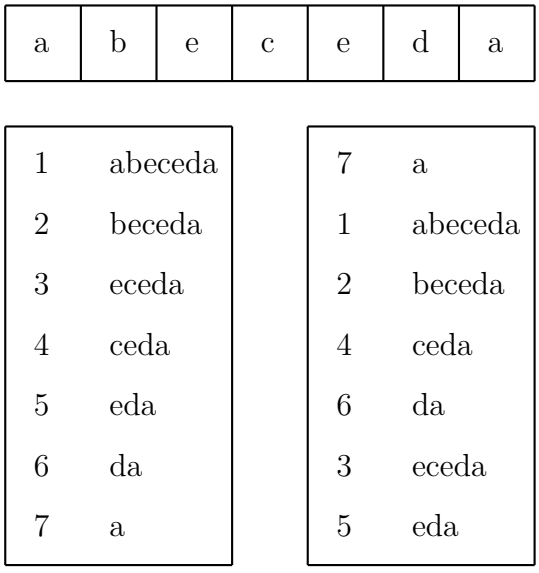
V této sekci bude našim cílem najít způsob, jak efektivně počítat hodnoty l_a, r_a pro $a \in \text{alph}(w)$. Připomeňme si, že l_a (resp. r_a) je maximální velikost podslova, které se vyskytuje bezprostředně před (za) všemi výskyty písmene a.

Definice 17 (Longest Common Prefix)**.** *Mějme slovo w délky n. Pro indexy i, j, $1 \leq i, j \leq n$ definujeme* $\text{LCP}(i, j)$ *(ve slově w) jako délku nejdelšího společného prefixu slov $w[i, n]$ a $w[j, n]$.*

Následující postupy a důkazy jsou převzaty z [6] a [7].

3.2.1 Suffix Array

Suffix Array je lexikograficky seřazené pole sufixů slova. Prvky *Suffix Array* budeme reprezentovat indexy, na kterých ve slově začínají (viz Obrázek 3.3). J.Kärkkäinen, P.Sanders a S.Burkhardt publikovali v [6] algoritmus, který dokáže toto pole pro slovo zkonstruovat v lineárním čase.



Obrázek 3.3: *Suffix Array* slova *abeceda* je $(7, 1, 2, 4, 6, 3, 5)$.

3.2.2 LCP Array

Longest Common Prefix Array je pole, do kterého ukládáme délky nejdelších společných prefixů mezi každou dvojicí po sobě jdoucích sufixů ze *Suffix Array* (viz Obrázek 3.4). V [8] byl publikován Kasaiův algoritmus, který dokáže ze slova a jeho *Suffix Array* postavit odpovídající *LCP Array*, a to v lineárním čase. Nyní si ukážeme, jak pomocí těchto struktur získat hodnotu $\mathrm{LCP}(i, j)$.

Tvrzení 10 (Sekce LCP queries for arbitrary suffixes v [7])**.** *Buď SA Suffix Array slova w a buď LA odpovídající LCP Array. Pak*

$$\mathrm{LCP}(i, j) = LA[\mathrm{RMQ}_{LA}(SA^{-1}[i] + 1, SA^{-1}[j])],$$

kde SA^{-1} *je inverzní pole k SA a* $SA^{-1}[i] < SA^{-1}[j]$.

Důkaz. Buď n délka slova w. Sufix $w[i, n]$ začínající na indexu i ve slově w se nachází na indexu $SA^{-1}[i]$ v poli SA. Každý společný prefix sufixů $w[i, n]$ a $w[j, n]$ musí být společný prefix všech sufixů mezi i-tou a j-tou pozicí pole SA^{-1}, protože je SA lexikograficky seřazené pole. Délka nejdelšího prefixu společného pro všechny tyto sufixy je minimum z intervalu $LA[SA^{-1}[i] + 1, SA^{-1}[j]]$.

□

	a	b	e	c	e	d	a
SA	7	1	2	4	6	3	5
SA^{-1}	2	3	6	4	7	5	1
LA	0	1	0	0	0	0	1

Obrázek 3.4: Slovo *abeceda* a jeho *Suffix Array SA*, SA^{-1} a *LCP Array LA*.

Jelikož umíme pole SA, SA^{-1} a LA postavit v lineárním čase a hodnoty RMQ umíme počítat po lineárním předzpracování v konstantním čase, tak hodnoty l_a a r_a, které získáme opakovanou aplikací LCP na všechny pozice z $\mathrm{Occ}(a)$, umíme vypočítat pro všechny znaky slova celkem v lineárním čase.

3.3 Marked Ancestor Probem

V této sekci se budeme věnovat tomu, jak odpovídat na dotazy ohledně předchůdců a následníků v množinách L, R, E z Algoritmu 3. Následující postup je převzatý z [3].

Nejprve si definujeme *Marked Ancestor Problem* (problém označeného předchůdce). Mějme strom T s n uzly, ve kterém každý z uzlů může být buďto *označený*, nebo *neoznačený*. Každý uzel v můžeme označit nebo se zeptat na první

označený uzel na cestě ke kořeni stromu od v. Sekce 2 v [9] ukazuje, jak m označení a dotazů na první označený uzel můžeme provést v čase $O(n+m)$.

V našem případě jsou množiny L, R, $\mathrm{Occ}(E)$ podmnožiny $\{0,\ldots,n\}$ a jedinou operací, kterou na těchto množinách provádíme, je vložení. Pokud jako strom T použijeme cestu délky $n+1$, bude vložení do jedné z množin odpovídat označení odpovídajícího uzlu v T. Dále, dotaz na první označený uzel na cestě ke kořeni bude odpovídat hledání předchůdce v množině. Tím pádem, pokud bude počet vložení do množin a dotazů na předchůdce a následníky $O(n)$, potom bude celková časová složitost všech dotazů na předchůdce a následníky lineární.

3.4 Decremental-Maxima

Struktura *Decremental-Maxima* slouží v efektivní verzi algoritmu pro správu množin $\mathit{ActiveSet}(e)$ pro expanzní znaky e. Následující postupy a důkazy jsou převzaty z [3].

Mějme pole celých čísel $t[1,\ldots,m]$ inicializované na $t[i] = -\infty$ pro všechna $i = 1,\ldots,m$. Chceme provádět následující operace:

increasevalue(i,v): $t[i] := \max\{v, t[i]\}$;
max(): **return** $\max\{t[i] : i = 1,\ldots,m\}$;
notmaximal(): **return** nějaké i takové, že $t[i] \neq \max()$ nebo **null**, pokud takové i neexistuje;
reset(): $t[i] := -\infty$ pro všechna $i = 1,\ldots,m$.

Tvrzení 11 (Lemma 12 v [3])**.** *Struktura Decremental-Maxima může být inicializována v lineárním čase tak, aby operace* reset() *probíhala v lineárním čase a aby všechny ostatní operace probíhaly v konstantním čase.*

Důkaz. Budeme si udržovat pole t, aktuální maximum M a seznamy S, S' prvků z $(1,\ldots,m)$, kde každý prvek patří právě do jednoho z těchto seznamů. Dále budeme mít $p[i]$ jako ukazatel na umístění prvku i v jeho seznamu. Seznam S bude obsahovat všechny takové prvky, že $t[i] = M$. Operaci max() provedeme jednoduše vrácením hodnoty M. Operaci increasevalue(i,v) uděláme tak, že aktualizujeme hodnotu $t[i]$, pokud $v > t[i]$. Pokud $v = M$, pak přesuneme i do seznamu S. Pokud $v > M$, pak $M := v$, seznam S připojíme k seznamu S' a v seznamu S bude pouze prvek i. Operace notmaximal() vrátí libovolný prvek seznamu S, pokud není prázdný.

□

Tvrzení 12 (Lemma 13 v [3])**.** *Všechny operace se strukturami ActiveSet lze dohromady provést v lineárním čase.*

Důkaz. Mějme $e \in E$. Všechna $i \in \mathit{ActiveSet}(e)$ můžeme rozdělit na taková, pro která platí $\gamma_{\mathit{right}}(i) > \gamma_{\mathit{right}}(e)$, a taková, pro která platí $\gamma_{\mathit{left}}(i) < \gamma_{\mathit{left}}(e)$. Pro každý případ vytvoříme vlastní strukturu Decremental-Maxima.

Vezměme si první případ, tedy $\gamma_{\mathit{right}}(i) > \gamma_{\mathit{right}}(e)$ a $m = |\mathrm{Occ}(e)|$. Pokaždé, když vkládáme nový znak e do množiny E, postavíme strukturu Decremental-Maxima tak, že provedeme nejprve operaci reset() a následně operaci

increasevalue() pro všechna $p \in \mathrm{Occ}(e)$ pomocí hodnoty $\gamma_{right}(p)$. Pro všechna písmena dohromady to lze podle Tvrzení 11 udělat v lineárním čase.

Strukturu je potřeba aktualizovat ve chvíli, kdy se pro nějaké $p \in \mathrm{Occ}(e)$ změní hodnota $\gamma_{right}(p)$. To může nastat pouze, když je do množiny R vložena nová pozice ve slově, která je v pravém dosahu znaku e. Dohromady máme $O(n)$ takových aktualizací.

V posledním cyklu metody *ProcessInterval* se může stát, že se díky vložení nového expanzního znaku e do E zmenší hodnota $\gamma_{right}(e')$ pro nějaký jiný expanzní znak e'. V tom případě pro e' postavíme strukturu znovu od začátku v čase $O(|\mathrm{Occ}(e')|)$. Pro znaky $e'_1, \ldots, e'_l$, kterým se hodnota $\gamma_{right}(e'_i)$ zmenšila díky přidání znaku e, platí, že každý výskyt e'_i je předchůdcem v E odpovídajícího výskytu e, a tedy

$$\sum_{i=1}^{l} |\mathrm{Occ}(e'_i)| \leq |\mathrm{Occ}(e)|.$$

Tedy postavení struktur od začátku trvá pro každé nové písmeno e maximálně $O(|\mathrm{Occ}(e)|)$, tedy $O(n)$ celkem.

Operace notmaximal() se použije pro vybrání prvku z *ActiveSet(e)* v příkazu **foreach** $i \in$ *ActiveSet*(e). Časová složitost operace notmaximal() je konstantní a celkový počet kroků těchto cyklů bude $O(n)$, protože každý průběh přidá nějakou pozici do R.

Druhý případ, kdy $\gamma_{left}(i) < \gamma_{left}(e)$ je podobný, místo maxima používáme minimum. V tomto případě také není nikdy potřeba přepočítávat strukturu od začátku. Všechny operace se strukturami *ActiveSet* můžeme tedy provést celkem v lineárním čase.

□

3.5 Implementace množiny *IntervalsSet*

Nakonec si ještě ukážeme, jak je v [3] v efektivní verzi algoritmu implementována množina *IntervalsSet*.

Množinu *IntervalsSet* si udržujeme jako spojový seznam volných intervalů. K tomu si navíc uchováváme pro každou pozici ve slově ukazatel na takový volný interval, který na dané pozici ve slově končí, pokud takový interval existuje. Vždy může pro každou pozici ve slově existovat maximálně jeden takový interval.

Připomeňme, že volný interval je takový interval, který začíná pozicí ve slově, která se nachází v množině L, a končí pozicí ve slově, která se nachází v množině R. Přičemž mezi těmito pozicemi nesmí být žádná jiná pozice obsažená v L ani R, a zároveň se mezi těmito pozicemi nesmí vyskytovat žádný expanzní znak.

Každé přidání prvku do L nebo R může způsobit nejvýše jedno odebrání z množiny *IntervalsSet* a nejvýše jedno přidání do množiny *IntervalsSet*. Každé přidání znaku e do množiny expanzních znaků způsobí nejvýše $|w|_e$ odebrání z množiny *IntervalsSet*.

Všechny operace spojené s množinou *IntervalsSet* tedy mohou být implementované celkově v lineárním čase.

4. Časová složitost algoritmu

Nyní probereme celkovou časovou složitost Algoritmu 3.

Věta 13. *Časová složitost Algoritmu 3 je lineární.*

Důkaz. Vybrání jednoho prvku ze seznamu *IntervalsSet* je možné v konstantním čase (3.5).

V metodě *ProcessInterval* umíme nový expanzní znak najít v konstantním čase (3.1), stejně tak umíme v konstantním čase přidávat nové prvky do množin E, L, R (3.3). Prvků do těchto množin přidáváme celkem $O(n)$.

Úprava seznamu *IntervalsSet* probíhá vždy po přidání prvku do E, L, R pomocí dotazů na předchůdce (resp. následníky) v konstantním čase (3.3, 3.5), dohromady v čase $O(n)$. Všechny operace se strukturami *ActiveSet* dohromady probíhají v čase $O(n)$ (3.4).

V metodě *Propagate* přidáváme do fronty *LettersQueue* pro každou z $O(n)$ nových pozic R maximálně dva expanzní znaky, dohromady v čase $O(n)$.

Odebrání prvku z fronty *LettersQueue* probíhá v konstantním čase. Důležité je, že po odebrání prvku z fronty se vlivem pravidla (d') nesnažíme díky strukturám *ActiveSet* přidávat do množiny R zbytečné prvky, a tedy celkový počet přidávaných prvků do R zůstává $O(n)$.

□

Věta 5 ukazuje, jak jednoduše nalézt netriviální morfický rozklad slova w, které není morficky primitivní, pomocí množiny E a hodnot $\gamma_{left}(e), \gamma_{right}(e)$. Časová složitost zkonstruování takového morfického rozkladu je zřejmě lineární, stejně tak i časová složitost zkonstruování odpovídajícího standardního homomorfismu.

5. Implementace v jazyce Java

V příloze A.1 přikládám zdrojový kód vlastní implementace algoritmu v jazyce Java. Jazyk Java byl zvolen hlavně pro množnost snadného vytvoření grafického okna s vizualizací algoritmu.

Implementace třídy `LCPArray` je převzata z [6] a [8]. Implementace třídy `CartesianTree` je převzata z [4], [5] a [10]. Implementace třídy `DisjointSet` je z větší části převzata z [11]. Implementace všech ostatních tříd jsou mojí vlastní prací.

5.1 Datové struktury

Algoritmus je reprezentován třídou `FixedPoint`. Konstruktor umožňuje zapnout či vypnout textový výstup a zaznamenávání kroků pro vizualizaci. Samotný běh algoritmu probíhá v metodě `MorphicFactorization()`.

Metoda `MorphicFactorization()` nejprve provede v metodě `alphabetIni()` inicializaci abecedy a přípravu slova, následně provede inicializaci pomocných datových struktur v metodě `structIni()`, a nakonec spustí hlavní iteraci algoritmu, tedy metodu `mainIteration()`.

Pomocné datové struktury jsou reprezentovány v dalších jednotlivých třídách. Třída `CartesianTree` reprezentuje Kartézský strom, třída `LCPArray` řeší Lowest Common Prefix problém, třída `MAPStructure` řeší Marked Ancestor Problem a třída `DecrementalMaxima` obsluhuje množiny *ActiveSet*. Třídu `Visual` obsluhující vizualizaci podrobněji probereme v následující sekci. Ostatní třídy jsou pomocné. Detailnější popis tříd a jejich metod je možné nalézt v technické dokumentaci, která se nachází v příloze A.2.

5.2 Vizualizace

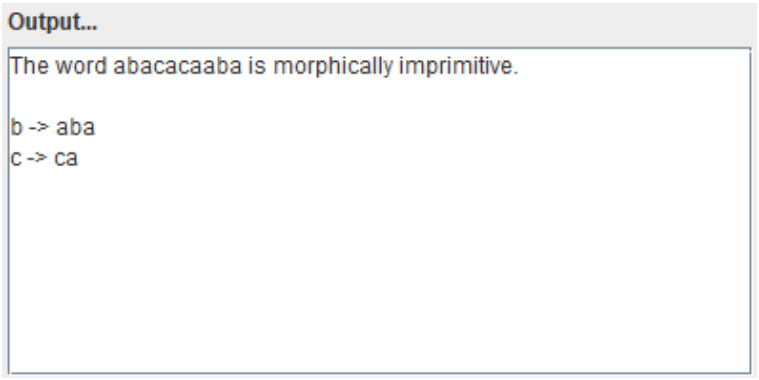
Vizualizaci chodu algoritmu přikládám ve formě `.jar` souboru v příloze A.3. Vizualizace slouží zejména ke snazšímu pochopení pravidel používaných v této verzi algoritmu a k demonstraci systému, jakým se pravidla postupně uplatňují.

Vizualizace chodu algoritmu je obsluhována třídou `Visual`. Každý krok algoritmu je uložený jako jedna instance třídy `VisualisationStep`, třída `Visual` tyto kroky zobrazuje. Pro tvorbu grafického rozhraní jsem používal knihovny AWT a Swing. Pokud chceme při běhu algoritmu uchovávat i data pro jednotlivé kroky vizualizace, tak nedosáhneme lineární časové složitosti. Časovou složitost algoritmu je tedy potřeba měřit bez zapnutého zaznamenávání kroků pro vizualizaci.

K zadání vstupu slouží textové pole označené textem *Input word...* (viz Obrázek 5.1). Ihned po zadání vstupního slova proběhne algoritmus a zobrazí se v textové podobě jeho výstup a v grafické podobě jeho první krok.

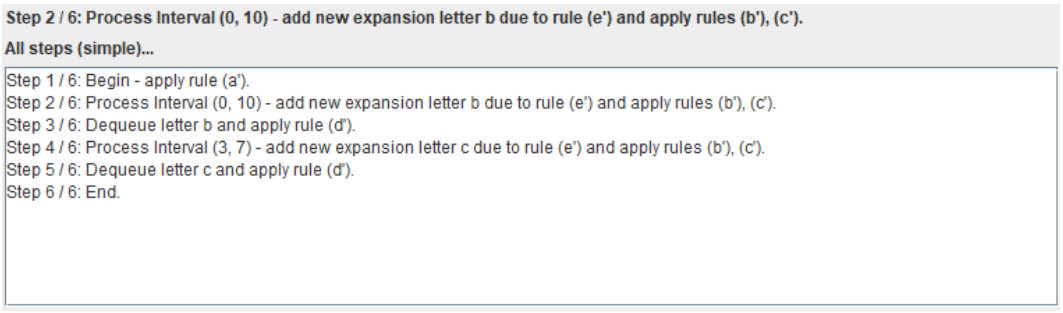
Textové pole označené textem *Output...* slouží k vypsání textového výstupu (viz Obrázek 5.2). V tomto poli je uvedeno, zda je testované slovo morficky primitivní, či složené, a případně se zde nachází definice homomorfismu, jehož je vstupní slovo netriviálním pevným bodem.

Obrázek 5.1: Textové pole pro vstupní slovo.



Obrázek 5.2: Textové pole pro výstup algoritmu.

Textové pole označené textem *All steps (simple)...* nebo *All steps (detailed)...* vypisuje všechny kroky, které algoritmus provedl (viz Obrázek 5.3). Možnost *simple* vypisuje kroky podle toho, ze které části algoritmu daný krok pochází (metoda *ProcessInterval* nebo odebírání znaků z fronty *LettersQueue*). Možnost *detailed* vypisuje jeden krok pro každé provedení nějakého pravidla. Nad textem *All steps...* je pak napsaný ten krok, který je zrovna graficky znázorněn. U každého kroku je uvedeno, o kolikátý krok se jedná, jaké části algoritmu se krok týká (jaká se uplatňují pravidla) a jakého znaku (případně intervalu) se krok týká.

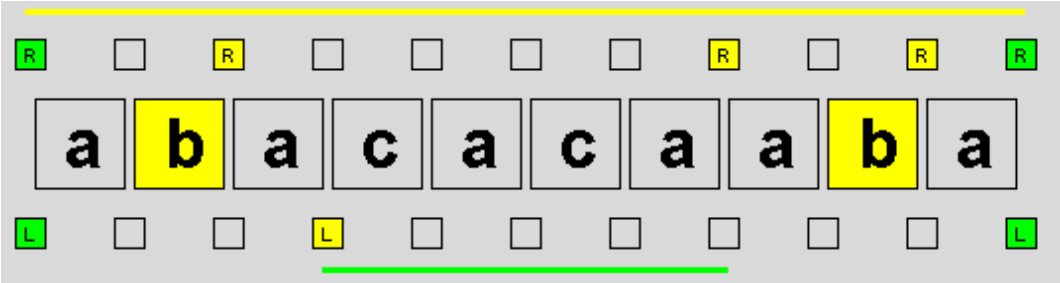


Obrázek 5.3: Textové pole pro detailní informace o jednotlivých krocích.

K přepínání mezi jednotlivými kroky slouží tlačítka *First* (první krok), *Previous* (předchozí krok), *Next* (následující krok) a *Last* (poslední krok). K přepínání mezi možnostmi *simple* a *detailed* slouží tlačítko *Details On/Off*.

Grafický výstup zobrazuje jednotlivé znaky ve slově, jednotlivé pozice ve slově a případné volné intervaly (viz Obrázek 5.4). Zvýrazněné znaky ve slově jsou ty, které už patří do množiny E. Pokud je znak zvýrazněn žlutou barvou, tak s ním algoritmus v aktuálním kroku pracuje. Zvýrazněné pozice ve slově jsou ty, které už jsou v množinách L (resp. R). Žlutě zvýrazněné pozice ve slově jsou ty, které algoritmus do množin L, R přidal v aktuálním kroku. Čáry pod slovem zobrazují aktuální intervaly v množině *IntervalsSet*. Čára nad slovem zobrazuje aktuálně zpracovávaný volný interval. Poslední krok také znázorňuje netriviální morfický

rozklad morficky složeného vstupního slova či triviální morfický rozklad morficky primitivního vstupního slova.



Obrázek 5.4: Grafická reprezentace konkrétního kroku algoritmu.

5.3 Testování výkonu

Testování výkonu obsluhuje třída `NewMain1`. K dispozici jsou metody `TimeTestLength`, `TimeTestAlphabet` a `TimeTestBoth`, které vypisují výsledky do textového souboru a metoda `TimeTest`, která vypisuje výsledky na standardní výstup. K získání přesného času se využívá metoda `System.nanoTime()`.

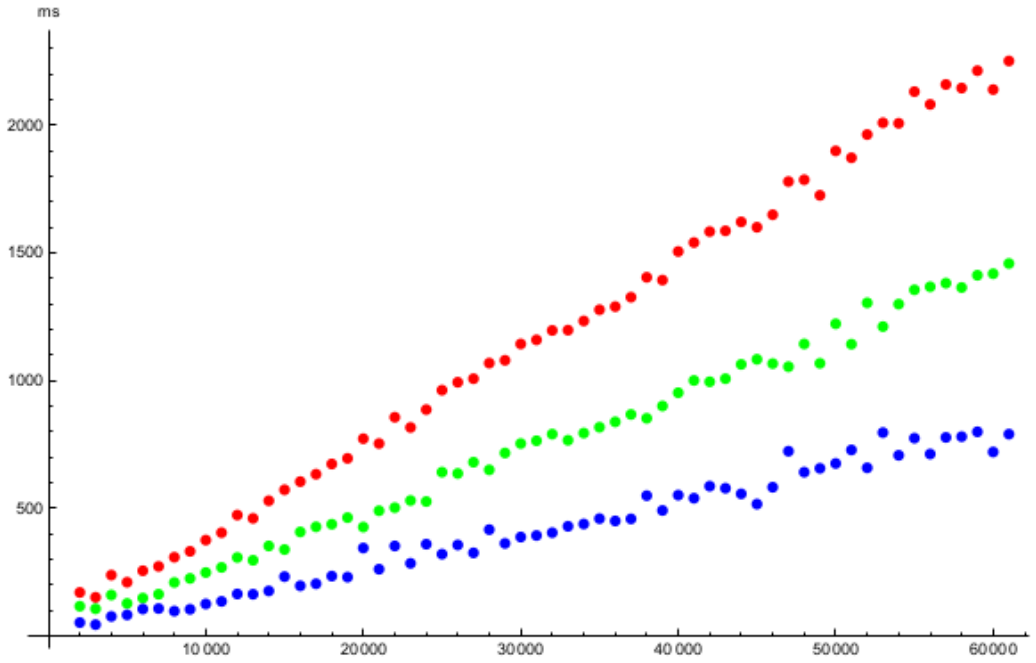
Pro následující měření jsem používal metodu `TimeTest`. Testování výkonu vždy probíhalo na 5 náhodných slovech, výsledný čas je průměrem časů běhu algoritmu na jednotlivých slovech. Červené hodnoty ukazují celkový čas běhu algoritmu. Zelené hodnoty představují čas přípravy slova a inicializace datových struktur, modré hodnoty představují čas běhu hlavní iterace. Naměřená data jsou také přiložená v příloze A.4.

Obrázek 5.5 ukazuje čas běhu algoritmu v milisekundách v závislosti na délce vstupního slova, přičemž velikost abecedy byla fixní (20). Testována byla slova o délce $n = \{2000, 3000, \ldots, 62000\}$.

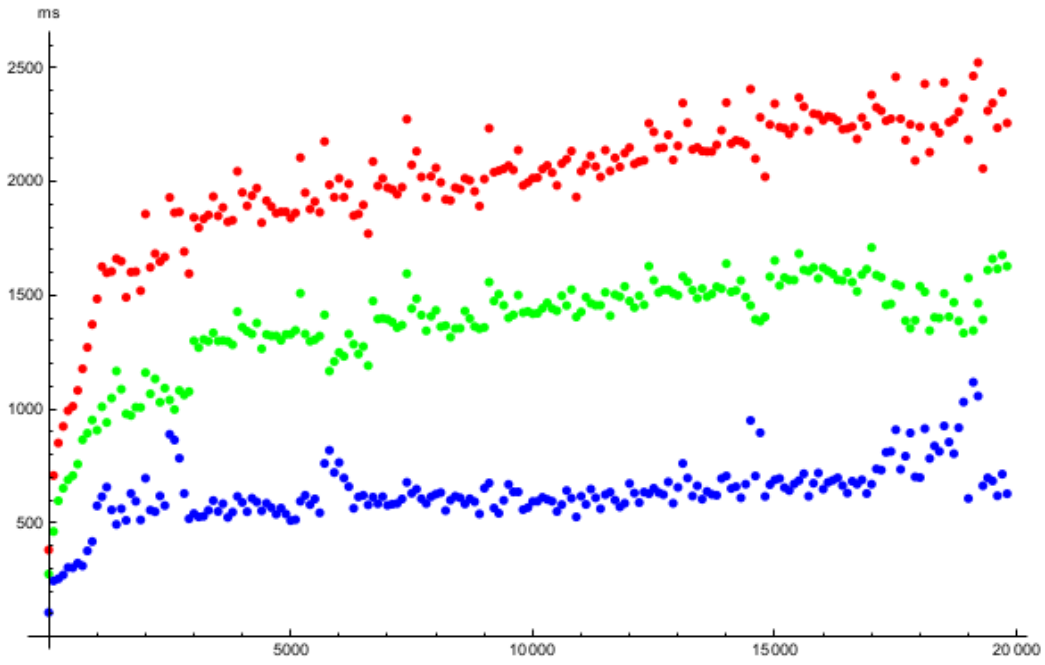
Obrázek 5.6 ukazuje čas běhu algoritmu v milisekundách v závislosti na velikosti abecedy, přičemž délka slova byla fixní (10000). Testovány byly abecedy o velikosti $m = \{5, 105, \ldots, 19905\}$.

Obrázek 5.7 ukazuje čas běhu algoritmu v milisekundách v závislosti na délce vstupního slova, přičemž velikost abecedy byla podobně veliká jako délka vstupního slova. Testována byla slova o délce $n = \{2000, 2200, \ldots, 9000\}$.

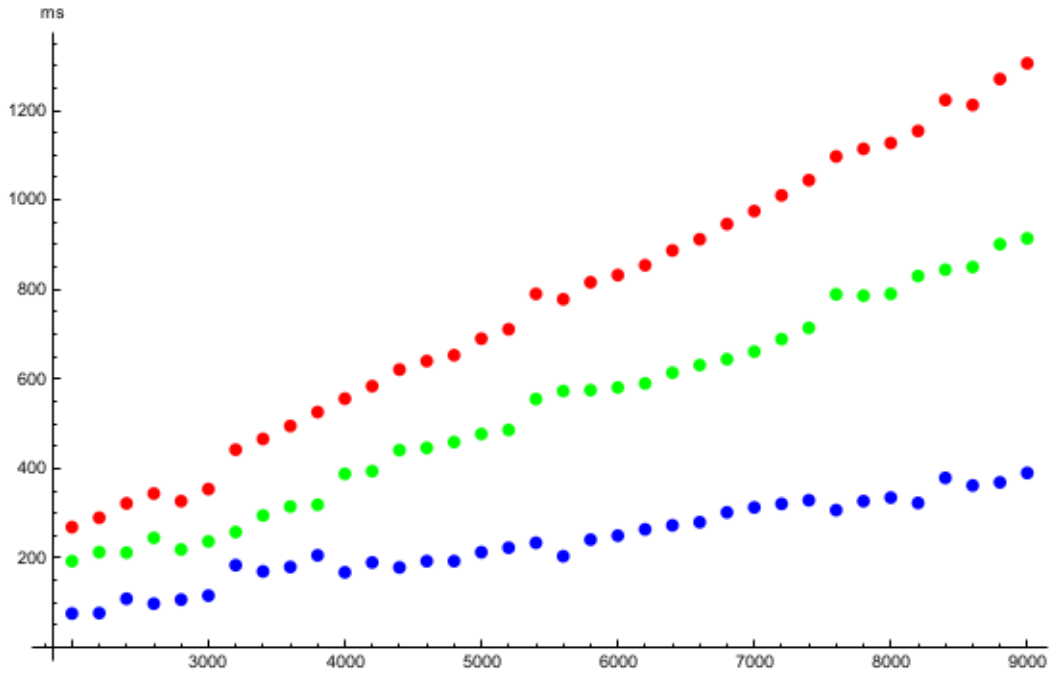
Z grafů je vidět, že v průměrném případě je složitost tohoto algoritmu přibližně lineární vůči délce slova, a to jak při fixní velikosti abecedy, tak při rostoucí velikosti abecedy. Dále si můžeme všimnout, že příprava datových struktur je časově výrazně náročnější než samotná hlavní část algoritmu. To odpovídá skutečnosti, že po většině pomocných datových struktur představených v kapitole 3 požadujeme konstantní časovou složitost odpovědí na dotazy po lineárním předzpracování. Oproti měřením, která prováděl ve své bakalářské práci Vojtěch Matocha [2], dosahuje tato implementace při pevné velikosti abecedy horších absolutních výsledků. To poukazuje na vysokou multiplikativní konstantu, způsobenou například již zmiňovanou náročnou inicializací datových struktur.



Obrázek 5.5: Doba běhu algoritmu v milisekundách v závislosti na délce slova.



Obrázek 5.6: Doba běhu algoritmu v milisekundách v závislosti na velikosti abecedy.



Obrázek 5.7: Doba běhu algoritmu v milisekundách v závislosti na délce slova a velikosti abecedy.

Závěr

V této práci jsem vysvětlil principy a fungování lineární verze Holubova algoritmu testujícího morfickou primitivnost slova. Dále jsem popsal pomocné datové struktury používané k dosažení lineární časové složitosti. Nakonec přikládám vlastní implementaci této verze algoritmu spolu s jeho vizualizací v jazyce Java.

Seznam použité literatury

[1] Štěpán Holub. Polynomial-time algorithm for fixed points of nontrivial morphisms. *Discrete Mathematics*, 309(16):5069–5076, 2009.

[2] Vojtěch Matocha. Algoritmus pro pevné body homomorfismů na slovech. Bakalářská práce, Univerzita Karlova, 2011.

[3] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń. Linear-time version of Holub's algorithm for morphic imprimitivity testing. *Theoretical Computer Science*, 602:7–21, 2015.

[4] Michael A. Bender and Martín Farach Colton. The LCA Problem Revisited. *LATIN*, 1776:88–94, 04 2000.

[5] CP-Algorithms. Lowest Common Ancestor - Farach-Colton and Bender Algorithm [online]. `https://cp-algorithms.com/graph/lca_farachcoltonbender.html`. Citováno 5.2.2020.

[6] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear Work Suffix Array Construction. *J. ACM*, 53(6):918—936, 11 2006.

[7] Wikipedia, The Free Encyclopedia. LCP Array [online]. `https://en.wikipedia.org/wiki/LCP_array`. Citováno 4.2.2020.

[8] Toru Kasai, Gunho Lee, Hiroki Arimura, Arikawa Setsuo, and Kunsoo Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching*, 2089:181–192, 06 2001.

[9] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209 – 221, 1985.

[10] CP-Algorithms. Solve RMQ by finding LCA [online]. `https://cp-algorithms.com/graph/rmq_linear.html`. Citováno 5.2.2020.

[11] Wikipedia, The Free Encyclopedia. Disjoint-set data structure [online]. `https://en.wikipedia.org/wiki/Disjoint-set_data_structure`. Citováno 4.2.2020.

Seznam obrázků

A. Přílohy

A.1 Příloha 1 - zdrojový kód v jazyce Java

Zdrojový kód implementace algoritmu v jazyce Java se nachází ve složce FixedPoint\src.

A.2 Příloha 2 - technická dokumentace

Technická dokumentace ve formě standardního Javadocu se nachází ve složce FixedPoint\dist\javadoc.

A.3 Příloha 3 - vizualizace

Spustitelný soubor `FixedPoint.jar`, který zobrazí okno s vizualizací, se nachází ve složce FixedPoint\dist.

A.4 Příloha 4 - naměřená data

Textové soubory s naměřenými daty se nachází ve složce FixedPoint\test. Data jsou ve formátu '{délka, celkový čas běhu algoritmu v milisekundách, čas běhu metody `alphabetIni`, čas běhu metody `structIni`, čas běhu metody `mainIteration`}'.