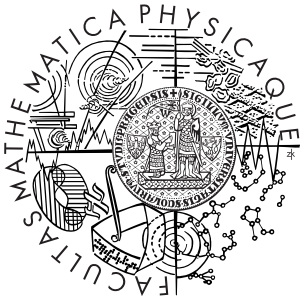
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jindřich Šedek

Algoritmy nad rozšířeným sufixovým polem

Kabinet software a výuky informatiky

Vedoucí diplomové práce: RNDr. Tomáš Dvořák, CSc.
Studijní program: Informatika, ISS

2009

Děkuji svému vedoucímu RNDr. Tomáši Dvořákovi, CSc. za čas, který mi věnoval a za literaturu, kterou mi zapůjčil.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Jindřich Šedek

Obsah

Název práce: Algoritmy nad rozšířeným sufixovým polem
Autor: Jindřich Šedek
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. Tomáš Dvořák, CSc.
e-mail vedoucího: Tomas.Dvorak@mff.cuni.cz

Abstrakt: Sufixový strom je v oblasti efektivních řešení vyhledávacích problémů jednou z nejdůležitějších datových struktur. Hlavní nevýhodou sufixového stromu je jeho prostorová složitost. Sufixové pole je naproti tomu prostorově úsporná datová struktura, která však nemá tak široké aplikace. Aby bylo možné využít sufixové pole v pokročilejších aplikacích, je nutné jej rozšířit dalšími pomocnými informacemi. Tato práce je zaměřena na experimentální srovnání rozšířeného sufixového pole se sufixovým stromem ve vybraných aplikacích z hlediska časové a prostorové složitosti.

Klíčová slova: datové struktury, sufixové pole, sufixový strom, analýza textů

Title: Algorithms for enhanced suffix array
Author: Jindřich Šedek
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Tomáš Dvořák, CSc.
Supervisor's e-mail address: Tomas.Dvorak@mff.cuni.cz

Abstract: Suffix tree is one of the most important data structures in string processing. However, the space consumption of the suffix tree is a bottleneck in large scale applications. Suffix array is by contrast a space efficient data structure having not so large domain of applications. To allow usage of the suffix array in more complex applications, we have to enhance it with an additional information. This work is aimed at a study of the enhanced suffix array and suffix tree in selected applications.

Keywords: data structures, suffix array, suffix tree, string processing

Kapitola 1

Úvod

Sufixový strom je velmi oblíbená datová struktura určená pro analýzu textů či jiných znakových posloupností. Mezi nejčastější aplikace patří například vyhledávání vzorku v textu, určení počtu různých podslov daného slova nebo nalezení nejdelšího opakujícího se podřetězce v textu.

V bioinformatice jsou DNA sekvence velmi často chápány jako slova nad abecedou nukleotidů {a, c, d, t}. V tomto pojetí se stávají předmětem zájmu lingvistické a statistické analýzy. I pro tyto účely je sufixový strom velmi užitečnou datovou strukturou.

Mezi největší nevýhody sufixového stromu patří jeho prostorová náročnost. Tento problém se snaží řešit pokročilejší datová struktura založená na konečných automatech zvaná CDAWG. CDAWG sice odstraňuje velké množství redundantních informací obsažených v sufixovém stromě, ale nepokrývá všechny aplikace sufixového stromu a jeho prostorová náročnost stále není ideální.

Konkurenční strukturou k sufixovému stromu a CDAWGu je sufixové pole. Sufixové pole je z hlediska paměťové složitosti výrazně úspornější struktura než sufixový strom i CDAWG. Sufixové pole však doposud nemělo tak velké využití. Na místě zde tedy přichází otázka, zda by nebylo možné nahradit sufixový strom sufixovým polem alespoň v některých aplikacích. Pozitivní odpověď na tuto otázku dávají Abouelhoda, Kurtz a Ohlebusch ve svém článku [1].

V této práci se pokusíme ověřit či vyvrátit, zda je možné nahradit sufixový strom sufixovým polem v jeho aplikacích a zaměříme se na porovnání jejich časové a paměťové složitosti. Pro účely porovnávání jsem mezi aplikacemi sufixového stromu vybral pět reprezentantů různého typu. Pro každou

z aplikací jsem implementoval algoritmus, který ji řeší pomocí sufixového stromu a algoritmus, který ji řesí pomocí sufixového pole. Oba algoritmy jsem poté porovnal z hlediska časové i prostorové složitosti na náhodně generovaných vzorcích dat a na vzorcích získaných z reálných dat.

Kapitola 2

Základní pojmy

Nejprve zavedeme několik základních pojmů a poté s jejich pomocí zadefinujeme datové struktury sufixové pole, sufixový strom a CDAWG.

Abeceda je neprázdná, konečná a uspořádaná množina znaků. Obvykle ji budeme značit Σ.

Slovo je konečná posloupnost znaků abecedy Σ. Při indexaci jednotlivých znaků slova budeme $i$tý znak slova S značit $S[i-1]$ a $|S|$ budeme označovat délku slova S. První znak slova S tedy budeme označovat $S[0]$.

Řekneme, že slovo x je **podslovo** slova S, pokud existuje index i takový, že $x = S[i] \dots S[i+|x|-1]$. Zkráceně budeme zapisovat $x = S[i \dots i+|x|-1]$.

Slovo S nazveme **zřetězením** slov x a y (zapisujeme xy) pokud platí:

- $|S| = |x| + |y|$
- $\forall i \in [0 \dots |x| - 1] : S[i] = x[i]$
- $\forall j \in [0 \dots |y| - 1] : S[|x| + j] = y[j]$

Přípona nebo **sufix** slova S je podslovo $S[i \dots n-1]$, kde n udává délku slova S. V textu budeme příponu slova S začínající znakem na pozici i označovat S_i.

Předpona nebo **prefix** slova S je podslovo $S[0 \dots i]$, pro $i < n$.

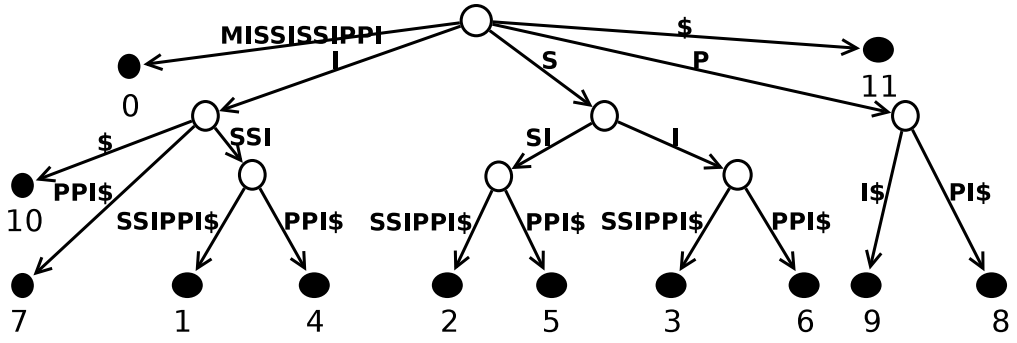
Nejdelší společná předpona slov x a y (zapisujeme $lcp(x, y)$) je slovo $x[0 \dots i]$ takové, že $x[0 \dots i] = y[0 \dots i]$ a navíc buď alespoň jedno ze slov má délku $i+1$ nebo jsou obě slova delší než $i+1$ znaků a $x[i+1] \neq y[i+1]$. *Nejdelší společnou předponou* množiny slov $x_1, x_2, \dots x_n$ je minimum z nejdelších společných předpon všech dvojic slov a značit jí budeme $lcp(x_1, x_2, \dots x_n)$.

Předpokládejme existenci speciálního znaku abecedy \$, který je větší, než všechny ostatní znaky abecedy a nenachází se v žádném slově S.

Sufixový strom pro slovo S délky n je kořenový strom s $n+1$ listy číslovanými od 0 do n. Každý vnitřní vrchol stromu různý od kořene má alespoň dva syny a každá hrana stromu odpovídá nějakému neprázdnému podslovu slova $S\$$. Každá dvě slova odpovídající hranám vedoucím z jednoho vrcholu začínají různým počátečním znakem abecedy Σ. Klíčovou vlastností sufixového stromu je, že zřetězení slov odpovídajících hranám na cestě z kořene do libovolného listu i tvoří právě příponu slova $S\$$ začínající znakem $S[i]$, tedy příponu $S[i \ldots n-1]\$$.

Hrany sufixového stromu je možné reprezentovat jako ukazatele do slova S na počáteční a koncový znak hrany. Tím je zajištěna lineární velikost sufixového stromu vzhledem k velikosti slova S [13].

O vrcholu v budeme říkat, že **reprezentuje** slovo T, podslovo slova S, pokud T vznikne zřetězením slov, která odpovídají hranám na cestě z kořene do v.



Obrázek 2.1: Sufixový strom pro slovo `MISSISSIPPI`

Konstrukci sufixového stromu je možné provádět několika způsoby. Nejjednodušší variantou je konstruovat strom postupným přidáváním jednotlivých přípon slova $S\$$ do rozrůstajícího se stromu.

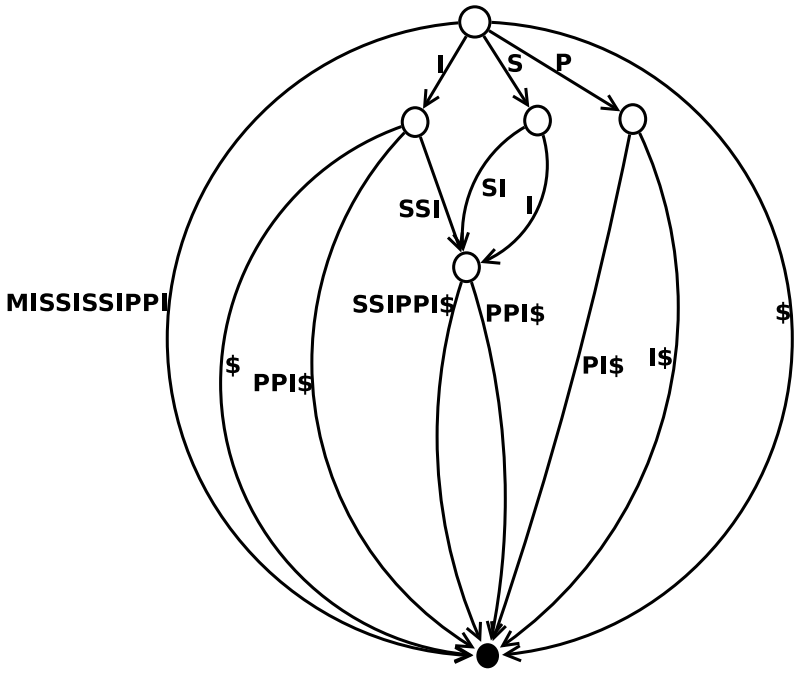
Algoritmus v prvním kroku vytvoří iniciální strom tvořený kořenem a jednou hranou odpovídající řetězci $S\$$ (tedy celému slovu). Poté pro každé i od 1 do n průchodem stromu od kořene k listům nalezne předponu slova $S[i \ldots n-1]\$$, která se již ve stromě nachází. Když nalezne vrchol, ze kterého již není možné dále pokračovat ve shodě s přidávanou příponou, vytvoří novou hranu, která bude odpovídat zbytku přidávané přípony. Pokud je místem

neshody místo na hraně, algoritmus v tomto místě rozdělí existující hranu novým vrcholem a do nově vzniklého vrcholu přidá novou hranu odpovídající zbytku vkládané přípony.

Tento kvadratický algoritmus není zdaleka optimální a jeho úpravou je možné získat lineární metodu konstrukce sufixového stromu známou jako McCreightův algoritmus [10]. Další metodou konstrukce sufixového stromu v lineárním čase je algoritmus Ukkonenův [15].

Sufixový strom je velmi užitečná datová struktura v mnoha úlohách zabývajících se vyhledáváním či jinou analýzou slov nad různými abecedami.

Hlavní nevýhodou sufixového stromu je jeho velká prostorová složitost. Tento problém se snaží odstranit pokročilejší datová struktura zvaná **Compact Directed Acyclic Word Graph** (CDAWG). CDAWG pro slovo S je konečný automat, který přijímá všechny přípony slova S. Jeho prostorová náročnost je menší, než prostorová náročnost sufixového stromu, protože CDAWG odstraňuje redundantní data obsahovaná sufixovým stromem.



Obrázek 2.2: CDAWG pro slovo `MISSISSIPPI`

Mezi známé algoritmy na konstrukci CDAWGu patří například Crochemorův algoritmus [4] nebo Inenagův algoritmus [8]. Konstrukcí CDAWGu se

i	*Sufixové pole*	$S[suftab[i] \dots n]$
0	7	`IPPI$`
1	4	`ISSIPPI$`
2	1	`ISSISSIPPI$`
3	10	`I$`
4	0	`MISSISSIPPI$`
5	9	`PI$`
6	8	`PPI$`
7	6	`SIPPI$`
8	3	`SISSIPPI$`
9	5	`SSIPPI$`
10	2	`SSISSIPPI$`
11	11	`$`

Obrázek 2.3: Sufixové pole pro slovo `MISSISSIPPI`

v tomto textu podrobněji zabývat nebudeme, protože vybočuje ze zaměření této práce a lze ji naleznout zpracovanou v [14].

Řekneme, že slovo T **lexikograficky předchází** slovo S (značíme $T \leq S$), pokud platí libovolný z následujících bodů:

- $|T| = 0$
- $(|S| > 0) \wedge (|T| > 0) \wedge (T[0] < S[0])$
- $(|S| > 0) \wedge (|T| > 0) \wedge (S[0] == T[0]) \wedge (T_1 \leq S_1)$

Pořadí dané předchozí relací budeme nazývat **lexikografické uspořádání** slov.

Sufixové pole pro slovo S délky n je číselné pole délky $n + 1$ udávající lexikografické uspořádání všech $n + 1$ neprázdných přípon slova $S\$$. Nechť $S[i \dots n - 1]\$$ je lexikograficky $k$tá přípona slova $S\$$, pak pro sufixové pole $suftab$ platí: $suftab[k - 1] = i$.

Nejjednodušší způsobem, jak zkonstruovat sufixové pole je pomocí libovolného třídícího algoritmu lexikograficky setřídit všechny přípony slova $S\$$. Podrobněji se konstrukcí sufixového pole zabývat nebudeme, protože vybočuje ze zaměření této práce. Velmi podrobnou analýzu několika známých konstrukčních algoritmů je možné naleznout v [16].

Kapitola 3

Rozšířené sufixové pole

Sufixové pole bez dalších pomocných struktur je možné použít například k rozhodnutí o existenci podřetězce či nalezení všech pozic jeho výskytů. Ve složitějších aplikacích je však samotné sufixové pole nedostatečná datová struktura a je nutné ho rozšířit o přídavné informace. V této kapitole postupně zavedeme tři přídavná pole, která nám umožní používat sufixové pole i k dalším aplikacím.

3.1 Lcptab

Mějme slovo S délky n nad abecedou Σ a sufixové pole *suftab* pro toto slovo. **Lcptab** je číselné pole délky $n+1$ obsahující čísla od 0 do n. Definujeme $lcptab[0] = 0$ a pro všechna $1 \leq i \leq n$ $lcptab[i]$ udává nejdelší společnou předponu lexikograficky i-té a $i-1$-ní přípony.

$$lcptab[i] = lcp(S[suftab[i] \ldots n-1], S[suftab[i-1] \ldots n-1]), 1 \leq i \leq n$$

Pole *lcptab* je možné konstruoval pomocí kvadratického algoritmu, který pro každé i, $1 \leq i \leq n$, postupným porovnáváním i-té a $i-1$-ní přípony nalezne nejdelší společnou předponu a její délku zapíše do pole $lcptab[i]$. Tento algoritmus je však možné vylepšit na lineární [7] použitím pomocného pole *ISA*.

Pole *ISA* je číselné pole délky $n+1$, které je inverzní k poli *suftab*. Pro jeho prvky tedy platí následující rovnost:

$$ISA[suftab[i]] = i$$

i	*Sufixové pole*	*lcptab*	$S[suftab[i] \dots n]$
0	7	0	`IPPI$`
1	4	1	`ISSIPPI$`
2	1	4	`ISSISSIPPI$`
3	10	1	`I$`
4	0	0	`MISSISSIPPI$`
5	9	0	`PI$`
6	8	1	`PPI$`
7	6	0	`SIPPI$`
8	3	2	`SISSIPPI$`
9	5	1	`SSIPPI$`
10	2	3	`SSISSIPPI$`
11	11	0	`$`

Obrázek 3.1: Sufixové pole rozšířené o pole *lcptab* pro slovo `MISSISSIPPI`

Pole *ISA* zkonstruujeme v lineárním čase pomocí předchozí definice.

Nyní nejprve uvedeme dvě základní vlastnosti, které platí pro nejdelší společné předpony přípon jednoho slova a poté popíšeme algoritmus konstrukce *lcptab*.

Lemma 1 *Nejdelší společná předpona lexikograficky $i-1$-ní a i-té přípony je větší nebo rovna nejdelší společné předponě lexikograficky j-té a i-té přípony pro všechna $0 \leq j \leq i-1$.*

Předchozí lemma přímo plyne ze skutečnosti, že pokud by lexikograficky j-tá přípona měla s i-tou delší společný prefix než $i-1$-ní, nutně by nemohla lexikograficky předcházet $i-1$-ní, což je spor s pořadím přípon.

Lemma 2 *Pro dvě libovolná slova S a T, jejichž nejdelší společná předpona je větší než 1 platí, že odebráním prvního znaku obou slov se jejich lexikografické pořadí nezmění a nejdelší společná předpona se zmenší o jedničku.*

Lemma přímo plyne z definice lexikografického uspořádání a definice *lcp*. V následující části popíšeme algoritmus konstrukce pole *lcptab* [1].

Algoritmus postupně porovnává přípony vstupního slova s jejich lexikograficky předchozími příponami od nejdelší po nejkratší. První iteraci začne nejdelší příponou, tedy celým řetězcem $S = S_0$. Pomocí pole *ISA*

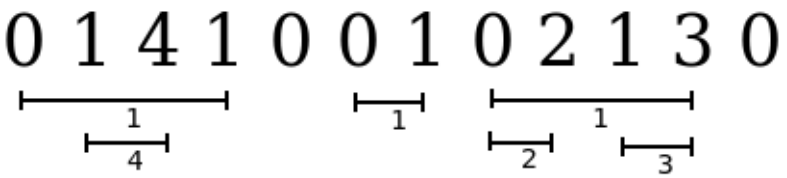
si zjistí, kolikátá je tato přípona v lexikografickém uspořádání a pomocí pole *suftab* poté nalezne k S_0 příponu lexikograficky předchozí. Označme ji Sp_0. Tyto dvě přípony znak po znaku porovná a získá jejich společný prefix $lcp_0 = lcptab[suftab[0]]$.

V druhém kroku se posune na druhou příponu S_1, tedy S bez prvního znaku. Algoritmus stejným postupem jako v předchozí iteraci nalezne k S_1 lexikograficky předchozí příponu Sp_1. Díky lemmatu 2 víme, že existuje přípona, která lexikograficky předchází S_1 a má s ní společný prefix délky $lcp_0 - 1$ (je to minule porovnávaná přípona Sp_0 zkrácená o první znak). Dle lemma 1 platí $lcp(S_1, Sp_1) \geq lcp_0 - 1$. Algoritmus proto v druhém kroku nemusí porovnávat S_1 a Sp_1 od začátku, ale může začít porovnávat od lcp_0-tého znaku.

V i-tém kroku pracuje algoritmus obdobně a využívá znalosti výsledku z $i-1$-ního kroku. Algoritmus pracuje v čase $O(n)$ [7] a mimo polí *suftab*, *ISA* a výsledného *lcptab* vyžaduje pouze konstantní množství paměti.

Pokud bychom pole *lcptab* reprezentovali jako pole čtyřbytových čísel, potřebovali bychom na jeho uložení $4n$ bytů. Čtyřbytová čísla však jsou zbytečně veliká a hodnoty pole *lcptab* je možné reprezentovat méně byty. Abychom získali lepší přehled o hodnotách pole *lcptab*, provedl jsem měření, jehož výsledky je možné naleznout v příloze v tabulkách B.1 a B.2. Sloupec `Max_LCP` udává největší hodnotu obsaženou v poli *lcptab* pro daný vstup, sloupec `count` udává počet hodnot pole *lcptab*, které jsou větší, než 254 a sloupec `promiles` udává kolik promile z celkového počtu lcp hodnot tvoří hodnoty větší než 254.

Z tabulek je možné vyčíst, že většina vzorků buď neobsahuje žádné hodnoty *lcptab* větší než 254 nebo jich obsahuje jenom velmi malé množství. Z tohoto důvodu je vhodné reprezentovat pole *lcptab* pouze jednobytovými čísly v rozsahu 0-254 a hodnoty větší než 254 označit zbývající hodnotou 255 a uložit je v pomocné datové struktuře. Paměťová složitost pole *lcptab* je $n+K$, kde K udává velikost pomocné paměti určené pro velké lcp hodnoty.



Obrázek 3.2: Struktura lcp-intervalů pro slovo `MISSISSIPPI`. Číselná řada udává hodnoty pole lcptab, vyznačené intervaly níže jsou lcp-intervaly s vyznačením jejich hodnoty

3.2 Lcp-intervaly

Nyní pomocí pole *lcptab* zavedeme lcp-intervaly a ukážeme jejich vlastnosti.

Interval $[i \ldots j], 0 \leq i < j \leq n$ splňující následující čtyři podmínky se nazývá **lcp-interval** s **hodnotou** l.

1. $lcptab[i] < l$
2. $lcptab[j+1] < l$
3. $lcptab[k] \geq l, \forall k \in [i+1 \ldots j]$
4. $\exists k \in [i+1 \ldots j] : lcptab[k] = l$

Lcp-interval s *hodnotou* l je maximální interval, jehož všechny prvky vyjma levé hranice jsou větší nebo rovné l. Interval nelze dále prodloužit (podmínky 1, 2) a l nelze zvýšit (podmínka 4). Číslo k ze čtvrté podmínky budeme nazývat *pevným bodem lcp-intevalu*. Upozorněme, že jeden *lcp-interval* může mít několik *pevných bodů*.

Lcp-intervaly se do sebe vnořují a tvoří dobré uzávorkování (nedochází ke křížení intervalů). Vnořené intervaly mají vždy vyšší hodnotu než obklopující interval, a proto pevné body nikdy nemohou být vnitřním bodem vnořeného intervalu, ale mohou se nacházet jedině na hranici vnořených intervalů. Podívejme se tedy na oblast, která se nachází mezi dvěma pevnými body.

Lemma 3 *Mějme lcp-interval* $[i \ldots j]$ *s hodnotou* l *a jeho dva po sobě následující pevné body* p *a* q, $p < q-1$. *Nechť mezi* p *a* q *není žádný jiný pevný bod. Pak interval* $[p \ldots q-1]$ *tvoří lcp-interval s hodnotou* $h = min\{p+1 \ldots q-1\}$.

Díky podmínce $p < q - 1$ je interval $[p + 1 \dots q - 1]$ neprázdný a h je tedy jednoznačně definováno. Z třetí podmínky *lcp-interval*u $[i \dots j]$ a z toho, že mezi body p a q neleží žádný pevný bod *lcp-interval*u $[i \dots j]$ plyne, že $h > l$. Body p a q tedy splňují první dvě podmínky na hranice intervalu $[p \dots q - 1]$. Protože mezi body p a q neleží žádný pevný bod a $h > l$ platí i třetí podmínka *lcp-interval*u $[p \dots q - 1]$. Čtvrtá podmínka je splněna přímo z výpočtu h.

Věta 4 *Mějme lcp-interval* $[i \dots j]$ *a nechť* $k_1 < k_2 < k_3 < \dots < k_n$ *jsou všechny jeho pevné body v rostoucím pořadí. Pak* $[i, k_1 - 1], [k_1, k_2 - 1], [k_2, k_3 - 1], \dots [k_n, j]$ *jsou buď jednoprvkové intervaly nebo vnořené lcp-intervaly.*

Pro intervaly mezi dvěma pevnými body věta plyne přímo z lemmatu 3. Krajní body i a j se z hlediska vlastností vnořených intervalů chovají stejně jako pevné body - hodnoty $lcptab[i]$ i $lcptab[j]$ jsou menší, než hodnota intervalu $[i \dots j]$, a pro krajní intervaly $[i, k_1 - 1]$ a $[k_n, j]$ tak platí všechny podmínky *lcp-interval*u. Krajní intervaly $[i, k_1 - 1]$ a $[k_n, j]$ jsou tedy buď jednoprvkové nebo také *lcp-interval*y. Jednoprvkové intervaly nesplňují čtvrtou podmínku *lcp-interval*u, a proto je není možné považovat za *lcp-interval*y. V některých aplikacích se však jednoprvkové intervaly také využívají. Příkladem takové aplikace je hledání nejkratších unikátních podslov, ke kterému se vrátíme v sekci 6.4.

V textu budeme dále používat pojem vnořené intervaly ve smyslu vnořených *lcp-interval*ů dle předchozí věty a nikoli ve smyslu všech podintervalů, jak je tento pojem často používán.

3.3 Childtab

Nyní se pokusíme naleznout analogii mezi *lcp-interval*y a vrcholy sufixového stromu. Každý vnitřní vrchol sufixového stromu lze jednoznačně identifikovat množinou listů, které se nacházejí v jeho podstromě. Tato množina listů reprezentuje množinu přípon, pro které platí, že délka jejich nejdelší společné předpony je k. k se často nazývá *výška vrcholu* a udává délku slova reprezentovaného vrcholem. Podobné rysy je možné naleznout i u *lcp-interval*ů.

Každý *lcp-interval* s hodnotou l je intervalem přípon, jejichž nejdelší společná předpona má délku l. Vnořené *lcp-interval*y se chovají stejně jako potomci vrcholů stromu. Množina přípon reprezentovaných vnořeným lcp-intervalem je podmnožinou množiny přípon reprezentované rodičovským lcp-intervalem. Hodnota vnořeného intervalu je větší, než hodnota rodičovského

intervalu (podobnost s tím, že potomci uzlů ve stromě reprezentují delší slovo, než jejich rodiče).

Přirozenou schopností sufixového stromu je naleznout k vrcholu seznam jeho potomků v čase úměrném k velikosti abecedy, ale nezávislém na délce vstupního slova. Každého potomka jednoho vrcholu sufixového stromu lze navíc jednoznačně identifikovat prvním znakem hrany, která k němu vede. *Lcp-interval*y takovéto schopnosti nemají, ale lze je o tuto funkcionalitu rozšířit. My to provedeme zavedením dalšího pomocného pole.

Mějme slovo S délky n nad abecedou Σ a sufixové pole *suftab* pro toto slovo rozšířené o pole *lcptab*. **Childtab** je pole velikosti $n+1$, jehož každá položka je tvořena třemi hodnotami *up*, *down* a *next*. Jednotlivé hodnoty pole jsou v rozsahu $0 \ldots n$ a zavedeme je následující definicí. Předpokládejme $\min(\emptyset) = \max(\emptyset) = \perp$.

$$\texttt{childtab[i].up} = \min \{q \in [0 \ldots i-1] \mid lcptab[q] > lcptab[i]\} \texttt{ and } \forall k \in [q+1 \ldots i-1] : lcptab[q] \leq lcptab[k]\}$$

$$\texttt{childtab[i].down} = \max \{q \in [i+1 \ldots n] \mid lcptab[q] > lcptab[i] \texttt{ and } \forall k \in [i+1 \ldots q-1] : lcptab[q] < lcptab[k]\}$$

$$\texttt{childtab[i].next} = \min \{q \in [i+1 \ldots n] \mid lcptab[q] = lcptab[i] \texttt{ and } \forall k \in [i+1 \ldots q-1] : lcptab[i] < lcptab[k]\}$$

V poli $childtab[i].down$ a v poli $childtab[j+1].up$ udržujeme informaci o prvním pevném bodu intervalu $[i \ldots j]$ a v poli $childtab[k].next$ budeme pro každý pevný bod k udržovat odkaz na následující pevný bod odpovídajícího intervalu nebo $\perp$, pokud následující pevný bod neexistuje.

Z věty 4 víme, že pro nalezení vnořených intervalů nám stačí naleznout pevné body *lcp-interval*u a tuto operaci nám pole *childtab* umožní. Tím bude možné naleznout i vnořené intervaly pro daný *lcp-interval* $[i \ldots j]$.

Předchozí definice pole *childtab* je z hlediska spotřeby paměti velmi nehospodárná a dá se výrazně zlepšit. Pole $childtab[i].down$ a $childtab[j+1].up$ obsahuje úplně stejnou informaci a tuto informaci pochopitelně není nutné uchovávat na dvou místech zároveň. Použít pouze pole $childtab[i].down$ nebo pole $childtab[j+1].up$ sice nestačí, protože dva lcp-intervaly mohou začínat i končit na i-tém poli, ale autoři článku [1] prezentují metodu, jak celé pole *childtab* uložit do jednoho číselného pole velikosti n. Pole *childtab* lze tedy implementovat pouze do $4n$ bytů paměti.

Konstrukce pole *childtab* se provádí pomocí lineárního průchodu polem *lcptab* za pomoci zásobníku, který modeluje hloubku zanoření *lcp-interval*ů. Algoritmus je velmi podobný algoritmu průchodu zdola nahoru, kterým se budeme podrobněji zabývat v sekci 4.1. Zde se z důvodu rozsahu a zaměření

			childtab			
i	*Sufixové pole*	*lcptab*	*up*	*down*	*next*	$S[suftab[i] \dots n]$
0	7	0	0	1	4	`IPPI$`
1	4	1	0	2	3	`ISSIPPI$`
2	1	4	0	0	0	`ISSISSIPPI$`
3	10	1	2	0	0	`I$`
4	0	0	1	0	5	`MISSISSIPPI$`
5	9	0	0	6	7	`PI$`
6	8	1	0	0	0	`PPI$`
7	6	0	6	9	11	`SIPPI$`
8	3	2	0	0	0	`SISSIPPI$`
9	5	1	8	10	0	`SSIPPI$`
10	2	3	0	0	0	`SSISSIPPI$`
11	11	0	9	0	0	`$`

Obrázek 3.3: Sufixové pole rozšířené o pole *childtab* pro slovo `MISSISSIPPI`

této práce přesným algoritmem konstrukce pole *childtab* zabývat nebudeme. V případě zájmu ho je možné naleznout v [1].

3.4 Suflinktab

Každý vrchol sufixového stromu velmi často udržuje kromě seznamu potomků také takzvaný sufixový odkaz. V této sekci pojem sufixového odkazu zavedeme a ukážeme, jak je možné rozšířit sufixové pole tak, aby umožňovalo ukládání sufixových odkazů.

Nechť u je libovolný vnitřní vrchol stromu T různý od kořene a x je slovo reprezentované vrcholem u. Protože u je vnitřní vrchol různý od kořene, je slovo x neprázdné a lze tedy psát $x = ay$, kde a je prvním znakem slova x. **Sufixový odkaz** je funkce, která pro daný vnitřní vrchol u nalezne vrchol reprezentující slovo y. Sufixový odkaz budeme zapisovat $v = suflink(u)$, kde v je vrchol reprezentující slovo y. Jednou z vlastností sufixového stromu je, že cílový vrchol sufixového odkazu vždy existuje.

Nyní přeformulujeme definici sufixového odkazu pomocí sufixového pole a lcp-intervalů. Mějme slovo S nad abecedou Σ a sufixové pole *suftab* pro toto slovo rozšířené o pomocné pole *lcptab*. Nechť platí $S_{suftab[i]} = ay$ a nechť j je index, který splňuje $S_{suftab[j]} = y$, kde a je libovolný znak a

			childtab			*suflinktab*		
i	*suftab*	*lcptab*	*up*	*down*	*next*	i	j	$S[suftab[i] \dots n]$
0	7	0	0	1	4	0	0	`IPPI$`
1	4	1	0	2	3	0	11	`ISSIPPI$`
2	1	4	0	0	0	9	10	`ISSISSIPPI$`
3	10	1	2	0	0	0	0	`I$`
4	0	0	1	0	5	0	0	`MISSISSIPPI$`
5	9	0	0	6	7	0	0	`PI$`
6	8	1	0	0	0	0	11	`PPI$`
7	6	0	6	9	11	0	0	`SIPPI$`
8	3	2	0	0	0	0	3	`SISSIPPI$`
9	5	1	8	10	0	0	11	`SSIPPI$`
10	2	3	0	0	0	7	8	`SSISSIPPI$`
11	11	0	9	0	0	0	0	`$`

Obrázek 3.4: Sufixové pole rozšířené o pole *suflinktab* pro slovo `MISSISSIPPI`

y libovolné slovo. Index j budeme nazývat sufixovým odkazem indexu i a značit $j = suflink(i)$. Nechť $[i \dots j]$ je libovolný lcp-interval s hodnotou $l \geq 1$. Pak nejmenší lcp-interval $[a \dots b]$ pro který platí: $a \leq suflink(i) < suflink(j) \leq b$ budeme nazývat **sufixovým odkazem intervalu** $[i \dots j]$ a značit $[a \dots b] = suflink([i \dots j])$.

Následující lemma 5 dává návod, jak určit sufixový odkaz libovolného indexu i.

Lemma 5 *Nechť* $S_{suftab[i]} = ay$, *pak* $suflink(i) = ISA[suftab[i] + 1]$.

Protože $S_{suftab[i]+1} = y$, musí platit, že $suftab[suflink[i]] = suftab[i] + 1$. Pak díky $ISA[suftab[i]] = i$ platí i $suflink[i] = ISA[suftab[i] + 1]$.

Pole **suflinktab** je pole velikosti $n + 1$ jehož každá položka je tvořena dvěma hodnotami. Pro každý lcp-interval $[i \dots j]$ ukládá levou a pravou hranici lcp-intervalu $suflink([i \dots j])$. Aby nedocházelo ke kolizím při ukládání informací do pole *suflinktab* jsou údaje odpovídající intervalu $[i \dots j]$ uloženy na pozici prvního pevného bodu lcp-intervalu $[i \dots j]$. První pevný bod je pro daný lcp-interval dán jednoznačně a žádné dva lcp-intervaly nemají žádný společný pevný bod. K nalezení prvního pevného bodu lcp-intervalu použijeme hodnoty *up* nebo *down* pole *childtab*.

Nyní se zaměříme na konstrukci pole *suflinktab*. Protože lcp-interval $suflink([i \dots j])$ reprezentuje slovo o jedna kratší, než interval $[i \dots j]$, je

$1 - [0 \ldots 3]$, $[5 \ldots 6]$, $[7 \ldots 10]$
$2 - [7 \ldots 8]$
$3 - [9 \ldots 10]$
$4 - [1 \ldots 2]$

Obrázek 3.5: Rozdělení lcp-intervalů dle jejich hodnoty setříděné dle levé hranice pro slovo `MISSISSIPPI`

i jeho hodnota o jedna menší. Algoritmus konstrukce pole suflinktab z této vlastnosti vychází a navíc využívá lemma 5.

V první fázi algoritmus prochází všechny lcp-intervaly a vytváří seznamy intervalů podle jejich hodnoty. Seznamy navíc udržuje setříděné podle rostoucí levé hranice intervalu.

V druhé fázi algoritmus pro každý seznam intervalů s hodnotou l a pro každý interval $[i \ldots j]$ v seznamu podle lemmatu 5 spočítá $suflink(i)$ a poté pomocí binárního vyhledávání v setříděném seznamu intervalů s hodnotou $l - 1$ nalezne interval, do kterého $suflink(i)$ patří. Nalezený interval je sufixovým odkazem intervalu $[i \ldots j]$.

Vzhledem k tomu, že počet všech intervalů ve všech seznamech je lineární vzhledem k velikosti vstupního slova S má první fáze algoritmu časovou složitost $O(n)$. Složitost druhé fáze je vzhledem k binárnímu vyhledávání $O(n\ log(n))$. Nicméně tuto fázi je ještě možné vylepšit odstraněním binárního vyhledávání tak, aby pracoval v čase lineárním $O(n)$ [2].

Protože pole *suflinktab* je pole velikosti $n + 1$ jehož každá položka je tvořena dvěma číselnými hodnotami, je jeho paměťová náročnost $8(n + 1)$ bytů.

Kapitola 4

Simulace průchodu sufixovým stromem pomocí sufixového pole

V předchozí kapitole jsme uvedli několik pomocných polí, kterými jsme rozšířili *sufixové pole*, abychom byli schopni zaznamenat informace, které si udržuje *sufixový strom*. Nyní se zaměříme na konkrétní algoritmy, které využívají těchto přídavných polí.

4.1 Průchod zdola nahoru

Průchod sufixovým stromem zdola nahoru je označení pro takové zpracovávání vrcholů sufixového stromu, při kterém zpracovávání daného vrcholu probíhá až ve chvíli, kdy již byli zpracováni všichni jeho potomci. Implementaci tohoto algoritmu pomocí sufixového pole poprvé uvedli Kasai et al. [7]. Na ně následně navázali Abouelhoda et al.[2]. Já zde uvedu ukázku novějšího algoritmu, protože je přehlednější byť je jeho princip zcela stejný.

Algoritmus generuje všechny *lcp-interval*y pomocí lineárního průchodu polem *lcptab* a zásobníku. Na zásobník si ukládá trojice $< i, j, l >$, které charakterizují *lcp-interval* $[i \ldots j]$ s hodnotou l. Pro práci se zásobníkem algoritmus používá dvě operace. Operaci *vloz*, která přidá novou trojici na zásobník a operaci *smaz*, která odstraní vrchol zásobníku. Proměnná *vrchol* přistupuje k vrcholu zásobníku.

Algoritmus pracuje v lineárním čase a důkaz jeho správnosti je možné naleznout v [7]. Algoritmus ke své práci vyžaduje zkonstruované pole *suftab*

```
vloz(< 0, ⊥, 0 >)
for k := 1 to n do
   x := k − 1;
   while lcptab[k] < vrchol.l do
         vrchol.j:= k − 1;
         zpracuj(vrchol)
         x := vrchol.i;
         smaz
   done
   if lcptab[k] > vrchol.l then begin
         vloz(< x, ⊥, lcptab[k] >)
   end
done
```

Algoritmus 1: Algoritmus průchodu sufixovým stromem zdola nahoru pomocí sufixového pole. Operace vloz a smaz pracují se zasobníkem tvořeným trojicemi $< i, j, l >$, které charakterizují *lcp-interval* $[i \ldots j]$ s hodnotou l. Proměnná *vrchol* přistupuje k vrcholu zásobníku.

a pole *lcptab*. Pole *childtab* ani *suflinktab* nejsou potřebné. Příkladem aplikace sufixového stromu, která využívá průchodu zdola nahoru, je například hledání nejdelší společné podposloupnosti dvou slov. V sekci 6.3 se této aplikaci budeme hlouběji věnovat a ukážeme její implementaci užitím sufixového pole.

4.2 Průchod shora dolů

Průchod sufixovým stromem shora dolů je asi nejčastější použití sufixového stromu. Příkladem aplikací, které tento průchod používají, je rozhodnutí o existenci vzorku či nalezení všech jeho výskytů. Algoritmus průchodu shora dolů implementovaný pomocí sufixového pole se opírá o pole *childtab*, pomocí kterého je pro daný lcp-interval možné naleznout všechny jemu vnořené intervaly.

Algoritmus 2 pro daný lcp-interval $[i \ldots j]$ vrátí seznam jemu vnořených lcp-intervalů. Některé intervaly mohou být jednoprvkové. V případě, že by byly požadovány pouze lcp-intervaly, bylo by nutné z výsledku jednoprvkové intervaly odfiltrovat. My je nyní zachováváme, protože je v některých aplikacích také využijeme.

$output$ = [ ];
if $(i < childdab[j+1].up < j)$ **then**
  $i1 = childtab[j+1].up$;
else
  $i1 = childtab[i].down$;
add($output$, i, $i1 - 1$);
while $(childtab[i1].next \neq \perp)$ **do begin**
  $i2 = childtab[i1].next$;
  add($output$, $i1$, $i2 - 1$);
  $i1 = i2$;
done
add($output$, $i1$, j);
return $output$.

Algoritmus 2: Algoritmus založený na poli *childtab*, který k danému lcp-intervalu $[i \ldots j]$ vrátí seznam jemu vnořených lcp-intervalů.

Pomocí algoritmu 2 již snadno naimplmenentujeme algoritmus průchodu shora dolů, který pracuje v lineárním čase. Pro jeho běh je nutné zkonstruovat pole *suftab*, *lcptab* a *childtab*. Pole *suflinktab* není nutné k průchodu shora dolů konstruovat.

4.3 Zpracovávání přípon slova od nejdelší po nejkratší

Další užitečnou funkcí sufixového stromu je možnost procházet všechny přípony vstupního slova postupně od nejdelší přípony po nejkratší. Tento přístup nachází své uplatnění například při počítání takzvaných *matching statistics*, které se dále používají při aproximativním vyhledávání vzorků. K implementaci této funkcionality používá sufixový strom *sufixové odkazy*, které jsme definovali v sekci 3.4. Způsob užití uvedeme přímo na aplikaci výpočtu *matching statistics* v sekci 6.5.

Kapitola 5

Implementace a provádění měření

Pro účely porovnání jednotlivých datových struktur jsem provedl jejich implementaci v jazyce C. Všechny implementace jsem provedl sám bez cizích kódů tak, aby byly pokud možno co nejjednotnější a aby implementační detaily nezvýhodňovali žádnou z použitých struktur.

Vrchol sufixového stromu jsem reprezentoval jako strukturu obsahující sufixový odkaz a ukazatel na začátek seznamu hran. Hrany stromu jsem reprezentoval strukturou obsahující index začátku hrany ve vstupním slově, index konce hrany ve vstupním slově, ukazatel na cílový vrchol hrany a ukazatel na následující hranu v seznamu. Konstrukci sufixového stromu jsem provedl pomocí Ukkonenova algoritmu [15].

Vrcholy CDAWGu jsem musel na rozdíl od sufixového stromu rozšířit o informaci vzdálenosti uzlu od kořene tak, jak to vyžaduje Inenagův algoritmus [8], který jsem na konstrukci CDAWGu použil.

Sufixové pole jsem konstruoval setříděním přípon vstupního slova pomocí třídícího algoritmu QuickSort a poté jsem k němu sestrojil pomocná pole lcptab, childtab i suflinktab dle algoritmů popsaných v předchozích kapitolách. U pole lcptab používám implementaci pomocí pole malých hodnot a velké hodnoty ukládám do pomocné hašovací tabulky. Pole childtab ukládám do pole se čtyřbytovými položkami jak je popsáno v sekci 3.3 a pole suflinktab ukládám do pole s osmibytovými položkami dle popisu v sekci 3.4. Konstrukci všech pomocných polí provádím také dle výše popsaných algoritmů. V případě více uvedených možností konstrukce používám varianty s nejlepší časovou složitostí.

Všechny uvedené implementace včetně dalších podrobností je možné naleznout ve zdrojových kódech a jejich dokumentaci na přiloženém CD.

V předchozích kapitolách jsme zavedli rozšíření sufixového pole o další přídavná pole tak, aby pomocí něho bylo možné implementovat algoritmy původně určené pro sufixový strom. Vzhledem k tomu, že jedním z hlavních důvodů, proč se snažíme nahradit sufixový strom sufixovým polem, je paměťová náročnost sufixového stromu, vypracoval jsem tabulky obsahující srovnání paměťové náročnosti všech použitých struktur. Tabulky je možné naleznout v přílohách B.5 a B.6.

Sloupce *Suffix Tree* a *CDAWG* udávají počet bytů spotřebovaných sufixovým stromem, respektive CDAWGem. Sloupec *Childtab Suffix Array* udává počet bytů spotřebovaných sufixovým polem rozšířeným o pomocná pole lcptab a childtab a sloupec *Full Suffix Array* udává počet bytů spotřebovaných sufixovým polem rozšířeným o pole lcptab, childtab i suflinktab. Všechny hodnoty udávají počet bytů potřebných na jeden znak vstupní posloupnosti a velikost vstupní posloupnosti v hodnotách není započítána.

Velikost rozšířeného sufixového pole jsem rozdělil do dvou sloupců, protože většina aplikací, které sufixové pole využívají, nevyžadují konstrukci plně rozšířeného sufixového pole a postačuje jim pouze sufixové pole rozšířené o pole lcptab a childtab. Pole suflinktab je mezi mnou vybranými aplikacemi potřebné pouze k výpočtu matching statistics, kterým se budeme věnovat v sekci 6.5.

Z tabulek je vidět, že spotřeba sufixového stromu se pohybuje v rozsahu dvacet až třicet byte na jeden znak vstupu, ale sufixové pole potřebuje pouze devět respektive sedmnáct byte na jeden znak vstupu, takže paměťová úspora je značná a nahrazení sufixového stromu pomocí sufixového pole přináší z hlediska paměťová náročnosti významné zlepšení. Některé z algoritmů užívajících sufixové pole (např. hledání supermaximálních opakování, kterému se budeme věnovat v sekci 6.2) navíc nevyžadují ani konstrukci pole childtab, což ušetří další čtyři byte a stlačí paměťovou spotřebu sufixového pole na pět byte na jeden znak vstupu.

Dalším významným parametrem datových struktur je jejich časová složitost. Sufixový strom je datová struktura, která vyniká v mnohých aplikacích velmi dobrou časovou složitostí a to nejen teoretickou, ale i praktickou. Na místě je tedy otázka, zda o tuto vlastnost nepřijdeme, rozhodneme-li se nahradit sufixový strom sufixovým polem. Aby bylo možné na tuto otázku odpovědět, vybral jsem pět různých aplikací sufixového stromu, pro tyto aplikace jsem implementoval algoritmy, které je pomocí sufixového stromu

řeší a následně jsem je naimplementoval tak, aby využívaly sufixové pole. Pro každou z aplikací jsem provedl experimentální srovnání obou implementací.

Experimentální měření jsem prováděl na náhodně generovaných datech i na reálných vzorcích, které jsem čerpal převážně z Canterburských korpusů [3]. Canterburské korpusy jsou data různého charakteru určená převážně pro testování komprimačních algoritmů. Vzhledem k tomu, že komprese dat je jednou z aplikací sufixového stromu a jedním z odvětví zpracovávání posloupností znaků, považuji tyto vzorky za vhodné i pro toto srovnání.

Mezi daty nalezneme například soubor *E.coli*, což je DNA sekvence bakterie Escherichia coli, soubor *bible.txt*, který obsahuje úplný anglický text Bible nebo soubor *pi.txt*, který obsahuje prvních milión číslic nekonečného rozvoje iracionálního čísla π.

Reálná data z Canterburských korpusů jsem ještě rozšířil o soubory *data.tar*, *data.zip*, *data.tar.gz* a *data.tar.bz2*. Tyto soubory jsem získal archivací všech zdrojových kódů této práce včetně metadat Subversion repository.

Náhodně generovaná data jsem generoval pomocí pseudonáhodného generátoru, který je součástí překladače. Zvažoval jsem též užití lepších pseudonáhodných generátorů, ale vzhledem k tomu, že mou snahou bylo využít náhodně generovaná data pouze jako vzorek dat, kterého si mohu vygenerovat neomezené množství s různými parametry a na kvalitě náhodnosti mi nezáleželo, tuto variantu jsem nakonec nezrealizoval.

Jako jednotné prostředí všech implementací jsem použil jazyk C. K překladu zdrojových kódů jsem použil překladač gcc verze 4.3 a kompilaci jsem prováděl s optimalizacemi "O2". Všechna měření jsem prováděl na počítači s procesorem Intel(R) Core(TM)2 Duo 2.53Ghz s operačním systémem Debian Linux 2.6.26-1-686. Použitý procesor je sice dvoujádrový, ale vzhledem k jednovláknové implementaci všech algoritmů měření probíhala pouze na jednom jádře procesoru. Na přiloženém CD (viz příloha A) je možné nalezнout všechny zdrojové kódy včetně Makefile.

Měření času jsem prováděl pomocí funkce `gettimeofday`, což je standardní unixové systémové volání, které vrací aktualní čas s přesností na milisekundy. Tuto funkci jsem použil před spuštěním měřeného algoritmu a po jeho skončení. Rozdíl obou časů prezentuji ve výsledcích jako čas běhu algoritmu. Tato metoda bohužel neodstíní čas běhu jádra systému, případné střídání procesů či jiné okolní vlivy, ale podmínky pro všechny měřené algoritmy jsou z tohoto pohledu stejné. Vnější vlivy jsem se navíc snažil snížit užitím velkých dat, na kterých se drobné výkyvy neprojeví a násobným prováděním jednotlivých měření.

Kapitola 6

Užití sufixového pole v aplikacích sufixového stromu

V této kapitole se zaměříme na některé typické aplikace sufixového stromu, ukážeme algoritmy, které tyto aplikace řeší pomocí sufixového stromu a poté ukážeme, jak je možné nahradit sufixový strom sufixovým polem. Na experimentálních výsledcích zhodnotíme vlastnosti obou struktur.

6.1 Hledání podslova a všech jeho výskytů

Mějme dané slovo P velikosti m zvané **vzorek** a slovo S velikosti n zvané **text**. Úlohou **nalezení vzorku** v textu je rozhodnout, zda existuje index i takový, aby platila rovnost:

$$\forall k \in 0 \ldots m-1 : P[k] = S[i+k].$$

Úlohou **nalezení všech výskytů vzorku** v textu je naleznout všechny indexy i, pro které platí předchozí rovnost.

V obou úlohách většinou předpokládáme, že velikost vzorku je výrazně menší, než velikost textu. Všechny námi zkoumané sufixové struktury jsou vhodné hlavně v případě, kdy je text pevně daný a známý předem, ale vzorky postupně přicházejí a požadavkem je jejich co nejrychlejší vyhodnocování. Z tohoto pohledu tedy budeme struktury hodnotit.

Obě úlohy jsou si dosti podobné a většinou se řeší i podobnými algoritmy, nicméně je nutné mezi nimi rozlišovat, protože ne všechny struktury a algoritmy jsou stejně vhodné pro obě úlohy. Například struktura CDAWG podporuje pouze úlohu nalezení vzorku a nalezení všech výskytů vzorku není

možné jednoduše implementovat bez dalších rozšíření či zvýšení složitosti algoritmu.

Algoritmus, který řeší obě úlohy pomocí sufixového stromu používá průchod stromem shora dolů, tedy od kořene k listům. Při průchodu hranami porovnává znaky odpovídající hranám se znaky vzorku. Pokud se mu podaří vyčerpat všechny znaky vzorku, znamená to, že vzorek se ve textu nachází a všechny hodnoty v listech podstromu, který se nachází pod místem poslední shody, udávají místa všech výskytů vzorku. Pokud se algoritmu nepodaří vyčerpat všechny znaky vzorku a narazí na místo, ze kterého není možné dále pokračovat, hledaný vzorek se v textu nevyskytuje.

Za předpokladu, že nezapočítáváme čas konstrukce sufixového stromu ani velikost nalezeného výsledku, pracuje algoritmus v čase $O(m)$ a je tedy plně nezávislý na velikosti textu S.

Nalezení vzorku pomocí struktury CDAWG probíhá v podstatě úplně stejně jako u sufixového stromu. Během průchodu CDAWGu shora dolů probíhá porovnávání znaků vzorku se znaky na hranách. Úlohu nalezení vzorku je možné rozhodnout stejně jako u sufixového stromu. V případě úspěšného nalezení celého vzorku však již není možné jednoduše naleznout všechny výskyty vzorku, protože CDAWG nemá listy a informace o počátcích přípon tak není zachována.

Pro řešení problému hledání vzorku a všech jeho výskytů pomocí sufixového pole si uvedeme dva algoritmy. První z nich je založen pouze na sufixovém poli bez dalších přídavných informací [9]. Tento algoritmus budeme označovat **SA_Simple**. Druhý využívá pomocných polí *lcptab* a *childtab* a pomocí lcp-intervalů provádí průchod shora dolů [1]. Označovat ho budeme **SA_ChildTab**.

Algoritmus *SA_Simple* je založený na lexikografickém uspořádání sufixového pole. Toto uspořádání nám zajišťuje, že pokud se vzorek v textu nachází, jsou všechny jeho výskyty shromážděny v podintervalu sufixového pole. Úlohou nalezení všech výskytů vzorku je tedy najít interval sufixového pole, jehož společný prefix odpovídá hledanému vzorku. U úlohy nalezení vzorku nám stačí najít libovolný prvek zmíněného intervalu.

K vyhledávání v poli se nejčastěji používá binární vyhledávání. Algoritmus začíná pracovat s intervalem $[0 \dots n]$, který postupně zmenšuje tak dlouho, dokud nenarazí na výskyt hledaného vzorku nebo dokud interval nezmenší na jednoprvkový a na něm neověří, že se vzorek v textu nenachází.

Trochu konkrétněji uveďme krok algoritmu při zpracovávání intervalu $[i \dots j]$. Algoritmus nejprve určí střed intervalu $m = \lceil (i + j)/2 \rceil$ a lexiko-

graficky porovná vzorek s příponou $S_{suftab[m]}$. Pokud je vzorek lexikograficky větší, pokračuje v dalším kroku s intervalem $[m \dots j]$. Pokud je lexikograficky menší pokračuje s intervalem $[i \dots m]$.

Předpokládejme, že se algoritmus nachází ve stavu zpracovávání intervalu $[i \dots j]$, v jehož středu m nalezne výskyt vzorku. Aby nalezl všechny výskyty pokračuje binárním vyhledáváním v obou intervalech $[i \dots m]$ i $[m \dots j]$. V intervalu $[i \dots m]$ hledá minimální k takové, že $S_{suftab[k]}$ odpovídá hledanému vzorku. V intervalu $[m \dots j]$ hledá maximální l takové, $S_{suftab[l]}$ odpovídá hledanému vzorku. Interval $[k \dots l]$ poté obsahuje pozice všech výskytů hledaného vzorku.

Algoritmus při hledání provede maximálně $O(log(n))$ porovnání řetězců, jejichž délka je shora omezena délkou hledaného vzorku, jeho složitost je tedy $O(m\ log(n))$. Pokud si algoritmus pamatuje, kolik znaků se mu podařilo úspěšně porovnat pro levou i pravou hranici zpracovávaného intervalu, nemusí tyto znaky porovnávat ani ve středu intervalu a díky lexikografickému uspořádání přípon má zajištěnou jejich shodu. Tímto způsobem je možné složitost algoritmu zlepšit na $O(m + log(n))$ [9].

$l = 0$; // levá hranice
$r = n$; // pravá hranice
$LP = 0$; // délka prefixu shodného s levou hranicí
$RP = 0$; // délka prefixu shodného s pravou hranicí
while $(r - l \geq 0)$
  $c = (r + l)/2$;
  $CP = min(LP, RP)$;
  $CP \mathrel{+}= lcp(vzorek[CP \dots m], S_{suftab[c]}[CP \dots m])$;
  if $(CP == m)$ **then**
    return TRUE;
  elseif $vzorek[CP + 1] < S_{suftab[c]}[CP + 1]$
    $l = c + 1$; $LP = CP$;
  else // $vzorek[CP + 1] > S_{suftab[c]}[CP + 1]$
    $r = c - 1$; $RP = CP$;
endwhile
return FALSE;

Algoritmus 3: Algoritmus *SA_Simple* sloužící k hledání výskytu vzorku v textu pomocí sufixového pole bez dalších pomocných struktur.

Algoritmus *SA_ChildTab* používá ke své činnosti sufixové pole rozšířené o pole *lcptab* a *childtab*. Pomocí algoritmu průchodu shora dolů prochází

lcp-intervaly. Začíná s intervalem $[0 \dots n]$ a zmenšuje ho tak dlouho, dokud společný prefix všech přípon intervalu neodpovídá hledanému vzorku. Všechny přípony reprezentované prvky tohoto intervalu jsou výskyty hledaného vzorku.

Konkrétněji předpokládejme, že se algoritmus nachází ve stavu, kdy zpracovává interval $[i \dots j]$ s hodnotou l. Pomocí algoritmu 2 získá seznam všech jemu vnořených intervalů. O všech vnořených intervalech ví, že jimi reprezentované přípony mají společný prefix délky l a že $l+1$-ní znak jednoznačně identifikuje nejvýše jeden vnořený interval. Pokusí se tedy mezi vnořenými intervaly naleznout ten, pro který platí, že $l+1$-ní znak libovolné jím reprezentované přípony je shodný s $l+1$-ním znakem hledaného vzorku. Pokud takový interval nenajde, hledaný vzorek se v textu nenachází. Pokud takový interval najde, nechť je to interval $[i1 \dots j1]$ s hodnotou $l1$.

Pokud $l1 = l+1$, algoritmus může přejít do stavu zpracovávání intervalu $[i1 \dots j1]$. Pokud platí $l1 > l+1$, je nutné ještě ověřit, že znaky vzorku na pozicích $l+1 \dots l1$ odpovídají znakům libovolné přípony reprezentované intervalem. Pokud ne, hledaný vzorek se v textu nenachází a pokud ano, algoritmus může přejít do stavu zpracovávání intervalu $[i1 \dots j1]$.

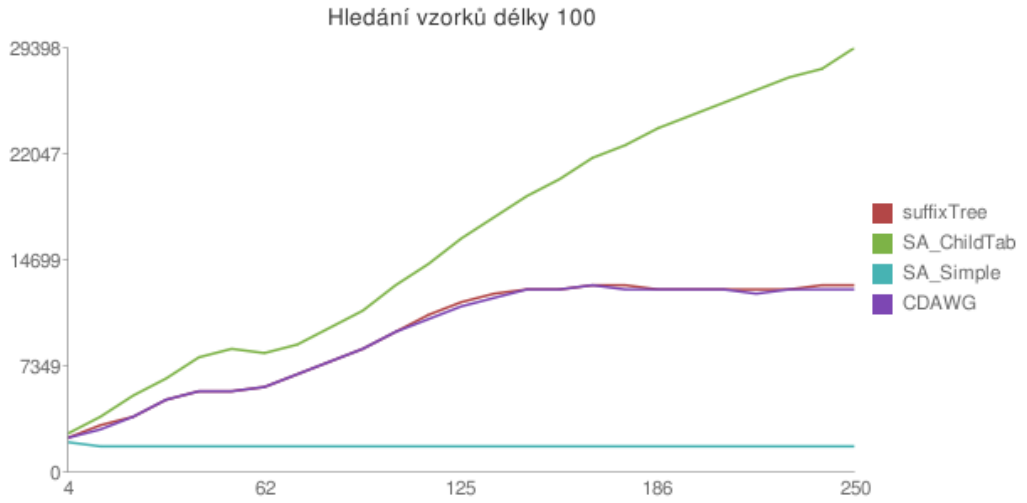
6.1.1 Experimentální výsledky

Abych mohl dobře posoudit vlastnosti jednotlivých algoritmů na nalezení vzorku a hledání všech jeho výskytů, implementoval jsem nejprve několik měření s náhodnými daty a poté jsem provedl několik měření s reálnými daty.

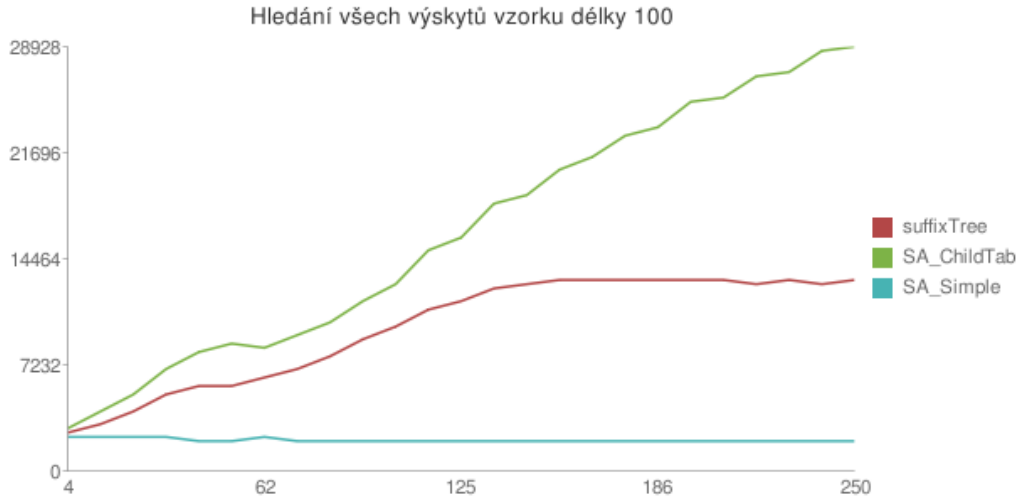
Výhodou náhodných dat je, že je možné vygenerovat libovolné množství dat různého charakteru a porovnat tak algoritmy z různých pohledů. Jejich nevýhodou je, že se svými vlastnostmi od reálných dat dost liší a jednotlivé algoritmy se na nich mohou chovat značně odlišně.

Jednou z vlastností vstupních dat, která má vliv na výkonnost jednotlivých algoritmů, je velikost abecedy Σ. První studii jsem tedy zaměřil na porovnání algoritmů z tohoto pohledu.

Obrázek 6.1 zobrazuje graf času vyhledávání 10^6 vzorků délky 100 znaků v závislosti na velikosti abecedy generovaných dat. Hledané vzorky jsou volené náhodně mezi podřetězci vstupního řetězce, takže všechna hledání musejí provézt kontrolu shodnosti všech 100 znaků. Všechny porovnávané algoritmy provádějí vyhledávání stejné množiny vzorků. Délka generovaného řetězce, nad kterým analýza probíhala, je $5 * 10^6$ znaků. Obrázek 6.2



Obrázek 6.1: Závislost času hledání vzorků délky 100 znaků v závislosti na velikosti abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas vyhledávání v milisekundách.



Obrázek 6.2: Závislost času hledání všech výskytů vzorku délky 100 v závislosti na velikosti abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas vyhledávání v milisekundách.

zobrazuje stejnou charakteristiku pro algoritmus vyhledávání všech výskytů vzorku.

Jednotlivé parametry měření jsem volil tak, aby naměřené časy byly v rozsahu jednotek až desítek sekund. Snažil jsem se tím dosáhnout kompromisního přístupu mezi velkou celkovou dobou provádění měření a dostatečnou přesností měření.

Měření neuspěšných vyhledávání jsem neprováděl, protože neuspěšné vyhledání je úloha shodná s úlohou úspěšného nalezení kratšího řetězce (shodné předpony) a následné nalezení neshody jednoho znaku, což již dělají všechny algoritmy zcela stejně a tak by toto měření nic nového nepřineslo.

Z grafů je patrné, že nejvhodnějším algoritmem pro vyhledávání vzorků v textu je algoritmus *SA_Simple*. Navíc je dobře vidět, že je tento algoritmus v obou případech plně nezávislý na velikosti abecedy.

Z obrázku 6.1 je také patrné, že rychlost hledání za pomocí sufixového stromu je v podstatě totožná, jako rychlost hledání pomocí struktury CDAWG. U obou struktur je zajímavé pozorovat, že čas hledání pro abecedu velikosti 4 až 100 téměř lineárně roste, ale pro abecedu od velikosti přibližně 140 dále už je téměř konstantní. Ve srovnání s algoritmem *SA_Simple* je však hledání pomocí sufixového stromu i struktury CDAWG výrazně pomalejší.

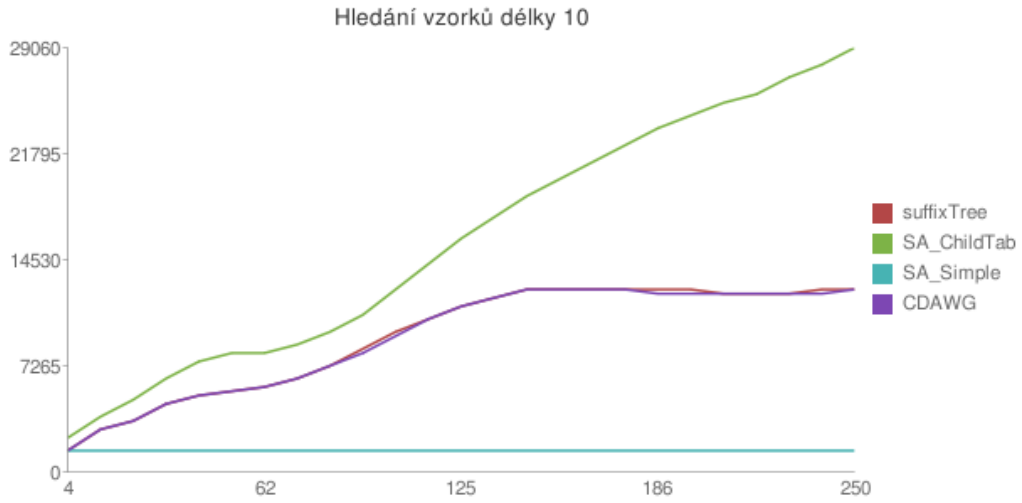
Nejhorší výsledky v tomto srovnání dává algoritmus *SA_ChildTab*, který je pro všechny velikosti abecedy horší, než všechny ostatní algoritmy. Jeho rychlost navíc téměř lineárně klesá s rostoucí velikostí abecedy.

Dalším parametrem, který by mohl mít vliv na vlastnosti algoritmů, je očekávaná délka hledaného řetězce. V předchozí části jsme algoritmy porovnávali při hledání vzorků délky 100 znaků. Obrázky 6.3 a 6.4 zobrazují závislost času vyhledávání vzorku na velikosti abecedy při hledání vzorků délky 10 respektive 1000 znaků. Velikost vstupních dat byla opět $5 * 10^6$ znaků.

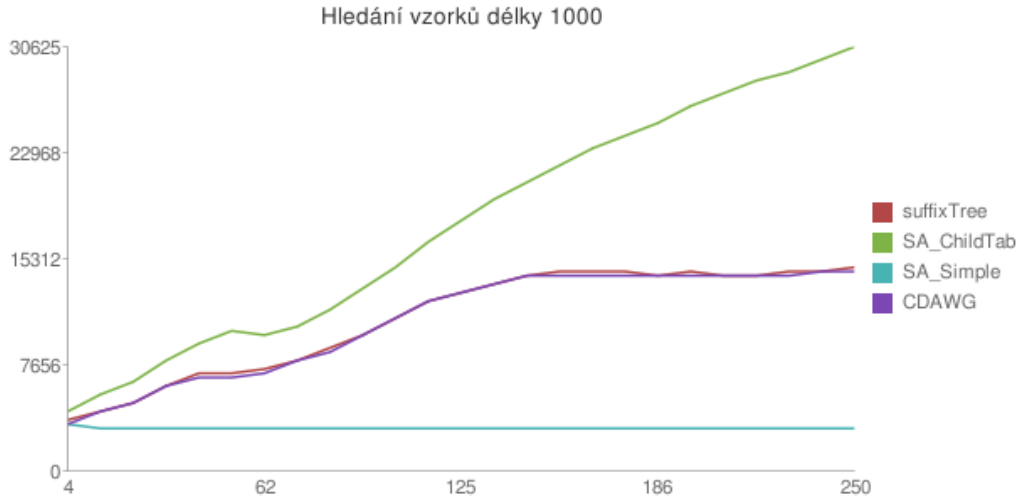
Tato měření však ukazují, že vliv délky hledaného vzorku na čas hledání je velmi malý a potvrzují, že vlastnosti algoritmů, které platí při hledání vzorků délky sto znaků platí také při hledání vzorků jiných délek.

Pro úplnost jsem ještě provedl podobné měření pro algoritmus hledání všech výskytů vzorku. Výsledek zobrazují grafy na obrázcích 6.5 a 6.6. Toto měření však žádné překvapení nepřineslo a pouze potvrdilo dříve zjištěné poznatky.

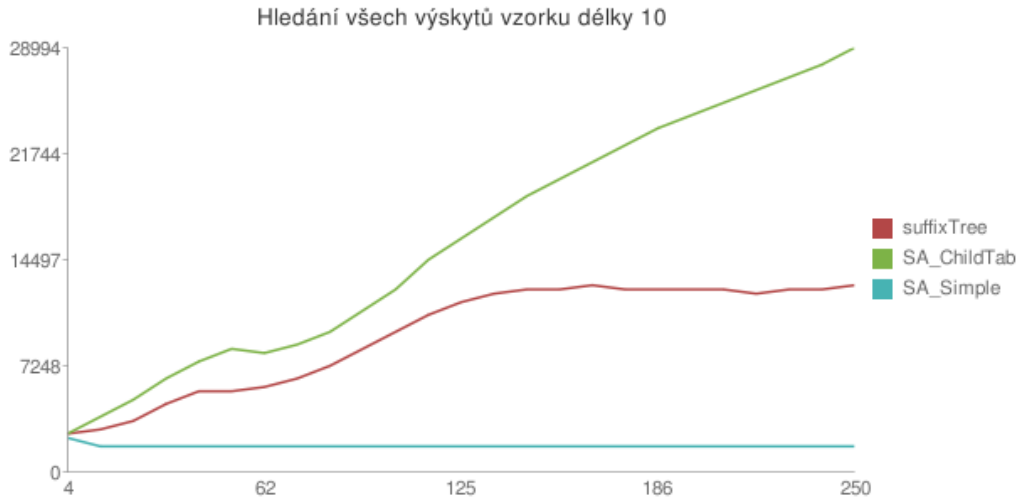
Vzhledem k tomu, že uvedené struktury je možné použít i pro práci s daty nad dvou a více bytovou abecedou, provedl jsem ještě měření zaměřené na velké abecedy. Obrázky 6.7 a 6.8 zobrazují výsledky obou měření.



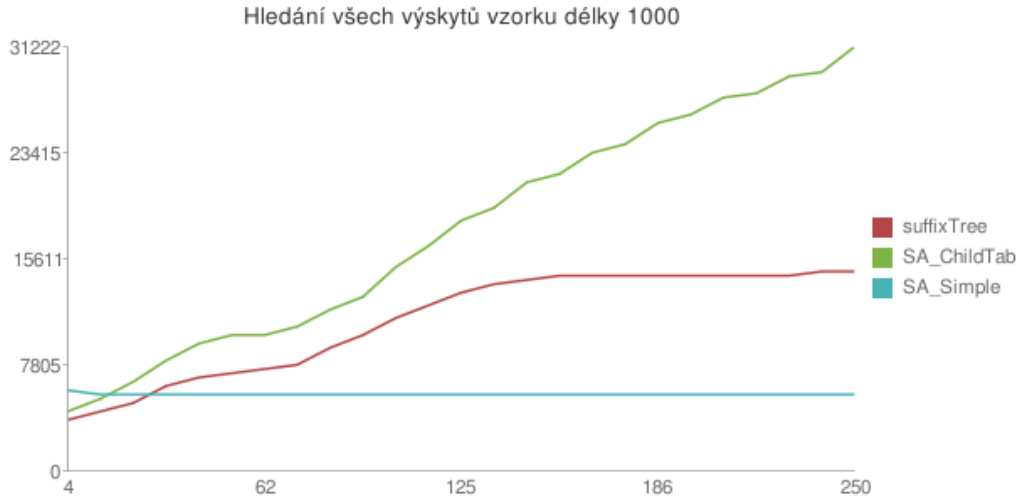
Obrázek 6.3: Závislost času hledání vzorků délky 10 v závislosti na velikosti abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas vyhledávání v milisekundách.



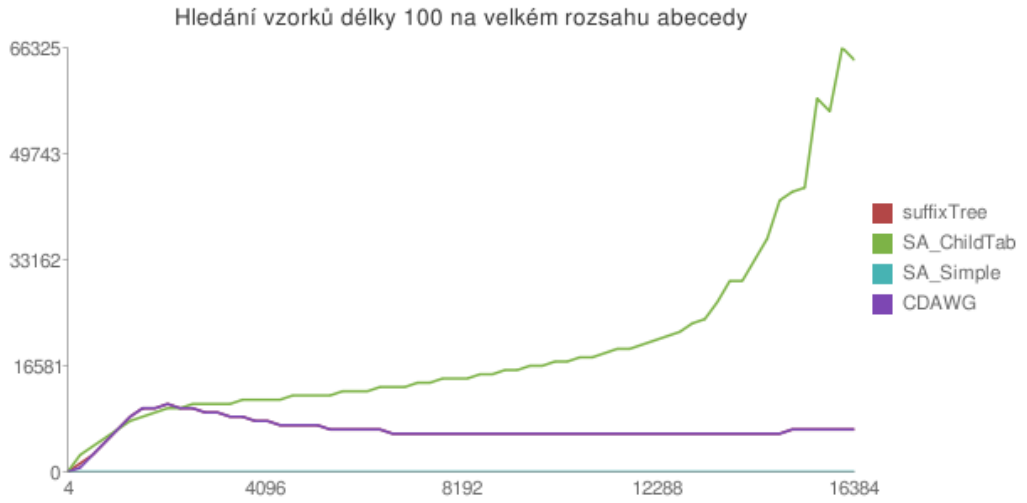
Obrázek 6.4: Závislost času hledání vzorků délky 1000 v závislosti na velikosti abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas vyhledávání v milisekundách.



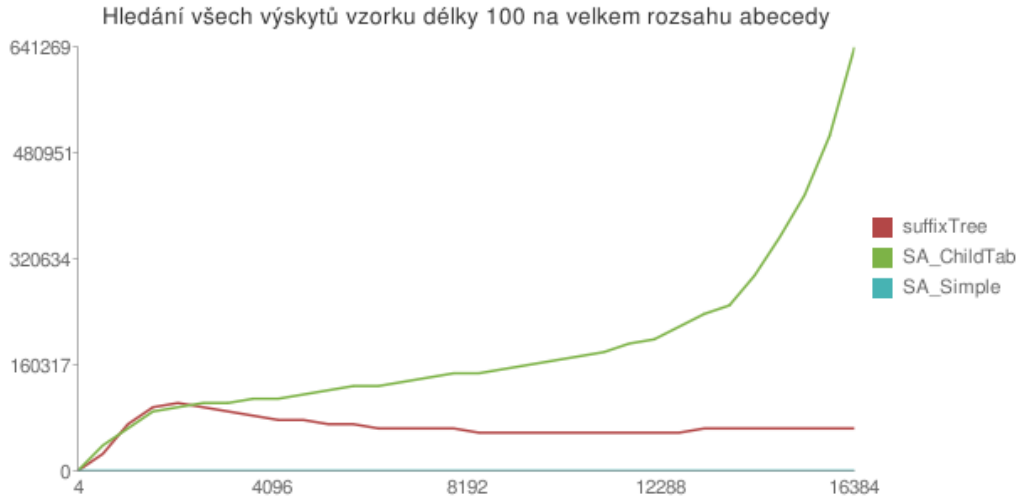
Obrázek 6.5: Závislost času hledání všech výskytů vzorku délky 10 v závislosti na velikosti abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas vyhledávání v milisekundách.



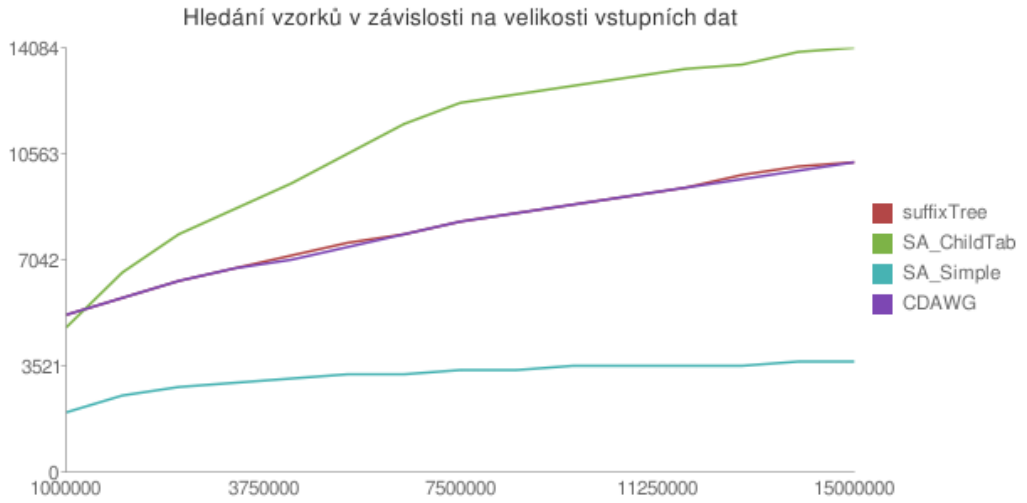
Obrázek 6.6: Závislost času hledání všech výskytů vzorku délky 1000 v závislosti na velikosti abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas vyhledávání v milisekundách.



Obrázek 6.7: Závislost času hledání vzorků délky 100 v závislosti na velikosti abecedy při velkém rozsahu abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas vyhledávání v milisekundách.



Obrázek 6.8: Závislost času hledání všech výskytů vzorku délky 100 v závislosti na velikosti abecedy při velkém rozsahu abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas vyhledávání v milisekundách.

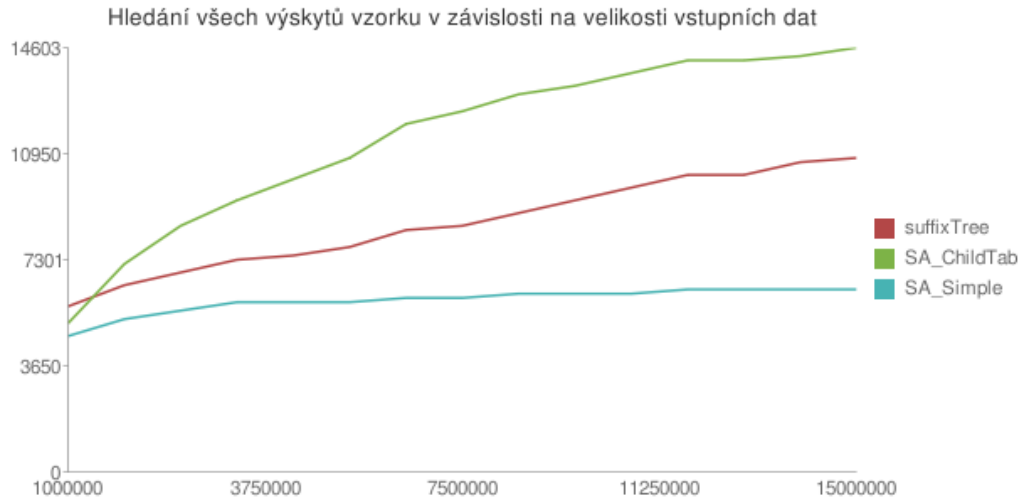


Obrázek 6.9: Závislost času hledání vzorků délky 100 znaků v závislosti na délce vstupních dat při abecedě o velikosti 60 znaků. Na ose x je vyznačena délka vstupních dat v rozsahu 10^6 až $15 * 10^6$ a osa y udává čas vyhledávání v milisekundách.

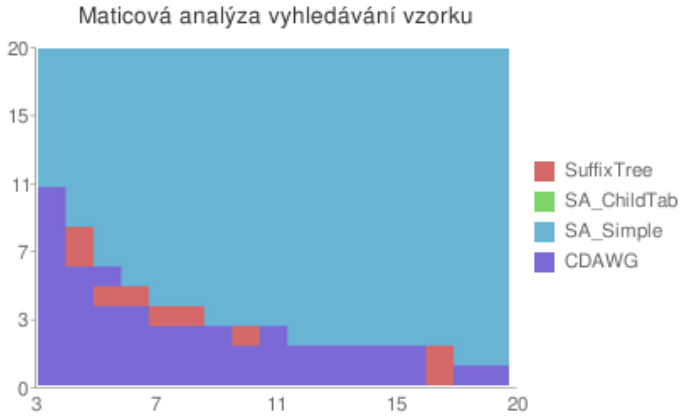
Z grafů lze vyčíst, že vlastnosti datových struktur sufixový strom a CDAWG jsou v kontextu této úlohy zcela vyrovnané. Navíc je z grafu velmi dobře vidět, že algoritmy využívající sufixové pole bez dalších pomocných struktur velmi dobře zvládají jakýkoliv rozsah abecedy. Naopak u algoritmů, které využívají sufixové pole rozšířené o další pomocné struktury, je znatelný výrazný pokles výkonu s rostoucí velikostí abecedy.

Dalším parametrem, který by mohl mít vliv na výkonnost jednotlivých algoritmů, je velikost vstupních dat. K analýze vlivu tohoto parametru jsem změřil závislost času hledání jednoho milionu vzorků délky 100 na velikosti vstupního řetězce dat při abecedě velikosti 60 znaků. Výsledek tohoto měření zobrazují grafy na obrázcích 6.9 a 6.10.

Z grafů lze vyčíst, že čas vyhledávání vzorku roste s rostoucí velikostí vstupních dat, což by se dalo předpokládat u všech použitých struktur. Vzhledem k tomu, že teoretická časová složitost algoritmu SA_Simple je logaritmicky závislá na velikosti vstupních dat, dalo by se předpokládat, že pokles výkonu bude s rostoucí velikostí dat nejvýraznější, ale z grafů je vidět, že velikost vstupních dat má na algoritmus SA_Simple menší vliv než na algoritmy používající rozšířené sufixové pole, sufixový strom či CDAWG.



Obrázek 6.10: Závislost času hledání všech výskytů vzorku délky 100 znaků v závislosti na délce vstupních dat při abecedě o velikosti 60 znaků. Na ose x je vyznačena délka vstupních dat v rozsahu 10^6 až $15 * 10^6$ a osa y udává čas vyhledávání v milisekundách.



Obrázek 6.11: Graf zobrazuje nejrychlejší algoritmus na vyhledávání vzorku v textu délky $5 * 10^6$ znaků v závislosti na velikosti abecedy a na délce hledaného slova. Osa x udává velikost abecedy a osa y délky hledaného slova.

Jako poslední experiment na náhodně generovaných datech jsem se pokusil na grafu znázornit, který algoritmus je nejlepší v konkrétní situaci při malých abecedách a krátkých hledaných vzorcích. Vytvořil jsem tedy graf zobrazený na obrázku 6.11. Graf zobrazuje měření vyhledávání vzorku v textu délky $5 * 10^6$ znaků v závislosti na velikosti abecedy a na délce hledaného řetězce. Jednotlivé barvy na grafu znázorňují, který algoritmus dává v dané situaci nejlepší výsledky.

Z grafu je vidět, že pro velmi malé abecedy a velmi krátké vzorky je nejvhodnější použít datovou strukturu CDAWG, což nebylo při předchozích měřeních vidět. Při větších abecedách či větších vzorcích se však potvrzuje, že nejrychlejším algoritmem je algoritmus SA_Simple. V několika případech, které se nacházejí převážně na hranici mezi sufixovým polem a CDAWGem, se podařilo prosadit i sufixovému stromu. Algoritmus SA_Childtab se neprosadil v žádné z měřených situací, což potvrzuje předchozí výsledky.

Podobné měření jsem chtěl udělat i pro hledání všech výskytů vzorku, ale tato úloha dává při malých abecedách a malých délkách vzorků příliš mnoho výsledků a měření pak je v podstatě neproveditelné kvůli spotřebě času a paměti. Pro větší abecedy a delší vzorky již výsledek známe z předchozích měření - dominuje v nich algoritmus SA_Simple.

Nyní se zaměříme na porovnání rychlosti algoritmů při práci s reálnými daty. Výsledky měření vyhledávání vzorku a všech jeho výskytů na reálných datech je možné naleznout v příloze v tabulkách B.7 a B.8. Obě měření uvádějí časy vyhledáváním 10^6 vzorků délky 10^3 znaků. Všechny uvedené časy jsou v milisekundách.

Z tabulky B.7 je dobře vidět, že na reálných datech již není dominance algoritmu SA_Simple nad ostatními algoritmy tak výrazná jako tomu bylo při zpracovávání náhodných dat. Velmi často ho totiž dokázal překonal algoritmus využívající strukturu CDAWG a velmi často jen těsně následovaný algoritmem využívajícím strukturu sufixový strom. Algoritmus využívající rozšířené sufixové pole se v této úloze stejně jako při zpracovávání náhodných dat příliš neosvědčil.

Z tabulky lze také dobře vyčíst, že rychlost hledání vzorku pomocí sufixového stromu je ve většině případů jenom o velmi málo větší, než při použití struktury CDAWG. I toto pozorování se shoduje s výsledky naměřenými při analýze s náhodnými daty.

Ještě jednoho pozorování si je možné všimnout. Na vzorcích, na kterých vychází struktura CDAWG lépe než sufixové pole, je rozdíl mezi strukturami velmi malý, ale v opačném případě je rozdíl mnohdy až dvojnásobný. Dalo

by se tedy říci, že algoritmus SA_Simple dává stabilnější výsledky, než konkurenční struktury, což může být v některých případech užití také užitečná vlastnost.

Z tabulky B.8, která obsahuje časy hledání všech výskytů vzorku na reálných datech, je na první pohled vidět, že tentokrát algoritmus využívající sufixové pole velmi zaostává za algoritmem, který využívá sufixový strom. Pouze v několika případech, uveďme například soubor *bible.txt*, se úlohu nalezení všech výskytů vzorku podařilo algoritmu založenému na sufixovém poli provézt rychleji, než algoritmu založenému na sufixovém stromě.

Toto měření je tedy ve sporu s měřeními, která byla provedena s náhodnými daty a ukazuje, že měření provedená na náhodných datech ne vždy popisují reálnou situaci.

6.2 Hledání supermaximálního opakování

Další obvyklou úlohou, pro kterou se používá sufixový strom, je hledání různě definovaných opakování vzorků. Jmenujme například hledání všech dvojic řetězců, které jsou shodné a nedají se prodloužit nebo hledání všech řetězců maximální délky, které se v textu nacházejí alespoň $n$krát. Nejčastější užití těchto úloh je při analýze genetických vzorků DNA.

My se v této sekci zaměříme na jednoho reprezentanta této skupiny úloh, jímž je hledání supermaximálního opakování slova. Tuto úlohu jsem mezi ostatními vybral z toho důvodu, že velikost výsledku je obvykle malá ve srovnání s velikostí vstupního slova. Průběh konstrukce výsledku tak má jen velmi malý vliv na čas běhu algoritmů a neovlivní výsledky měření.

Mějme slovo S, dvojici různých indexů i a j a číslo z takové, že platí následující podmínky:

1. $(i + z < |S|) \wedge (j + z < |S|)$
2. $\forall k \in 0 \ldots z - 1 : S[i + k] = S[j + k]$
3. $(i \neq 0) \wedge (j \neq 0) \Rightarrow S[i - 1] \neq S[j - 1]$
4. $(i + z < |S| - 1) \wedge (j + z < |S| - 1) \Rightarrow S[i + z] \neq S[j + z]$

Maximálním opakováním nazveme podslovo $R = S[i \ldots i + z]$ slova S. Pokud indexy i, j splňují třetí podmínku, budeme o podslovech, která na indexech i a j začínají, říkat, že jsou **zleva různá**. Obdobně budeme **zprava**

různá nazývat taková slova, která začínají na indexech i a j, mají délku z a splňují čtvrtou podmínku.

Supermaximální opakování ve slově S je takové maximální opakování, které se v S nikdy nenachází jako podslovo žádného jiného maximálního opakování.

Jako příklad uveďme slovo $S = abccbccabc$. Podslovo bc je maximální opakování, protože indexy 4 a 9 splňují všechny podmínky maximálního opakování. Není to však supermaximální opakování, protože slovo bc je podslovo slova abc a slovo abc je také maximální opakování. Slovo abc je dokonce supermaximální opakování.

Nyní se zaměříme na řešení úlohy nalezení supermaximálních opakování pomocí sufixového stromu. Mějme libovolný vnitřní vrchol v sufixového stromu různý od kořene. Nechť T je slovo, které vznikne zřetězením hran vedoucích z kořene do v. Protože v je vnitřní vrchol, vedou z něho alespoň dvě hrany, které začínají různými znaky abecedy. Nutně tedy existují alespoň dva výskyty slova T, které jsou zprava různé. Abychom byli schopni rozhodnout, zda jsou dvě přípony slova S zleva různé budeme potřebovat vědět, jaký znak předchází jejich začátku. Začátky přípon jsou u sufixového stromu uchovávány v listech.

Pokud bychom tedy hledali maximální opakování, stačí pro každý list naleznout znak, který předchází počátek jím reprezentované přípony a poté pro každý vnitřní vrchol určit množinu znaků reprezentovaných ve všech listech jeho podstromu. Pokud je mohutnost množiny větší než 1, pak řetězec, který vznikne zřetězením slov na hranách po cestě od kořene do vrcholu tvoří maximální opakování.

Protože nás zajímají pouze supermaximální opakování, musíme ještě zajistit, aby žádné nalezené opakování nebylo podslovem jiného maximálního opakování. Nechť v je libovolný vnitřní vrchol stromu, který reprezentuje maximální opakování a nechť w je potomek vrcholu v, který není list. Protože w je potomkem v, je slovo reprezentované v podslovem slova reprezentovaného w. Protože w není list, existují alespoň dva výskyty slova reprezentovaného vrcholem w, které jsou zprava různé. Tyto výskyty sice ještě nemusí tvořit maximální opakování, protože nemusí být zleva různé, ale pokud nejsou, jistě je lze zleva prodloužit, aby různé byly. w tedy dokazuje, že vrchol v nereprezentuje supermaximální opakování.

Mezi kandidáty na supermaximální opakování nám tedy zbývají pouze vrcholy, které jsou maximálním opakováním a všichni jejich potomci jsou listy. Jistě je snadné nahlédnout, že pokud má v alespoň dva listy, které re-

prezentují zleva shodné přípony, pak je lze doleva prodloužit, aby byly zleva různé a tím získat maximální opakování. Takový vrchol tedy také netvoří supermaximální opakování.

Nakonec nám zbývají vrcholy, jejichž všichni potomci jsou listy a navíc jsou všechny jimi reprezentované přípony zleva různé. Tyto vrcholy již tvoří supermaximální opakování.

Naleznout všechny zmíněné vrcholy je pomocí sufixového stromu možné například průchodem do hloubky. Ověření vlastností vrcholu probíhá v čase úměrném počtu potomků a ten je při předpokladu konstantní abecedy konstantní. Celková složitost algoritmu je tedy $O(n)$.

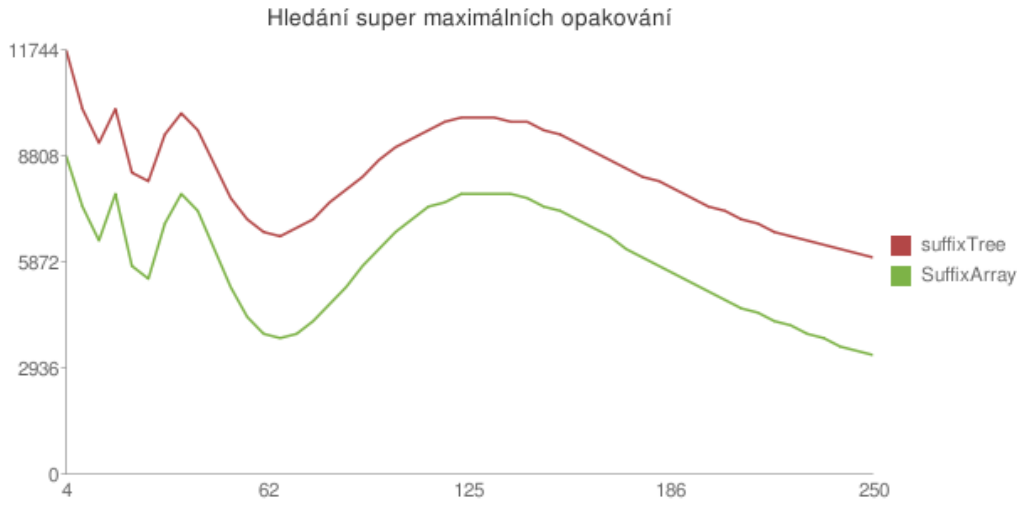
Není těžké nahlédnout, že všechny výše zmíněné skutečnosti platící o vrcholech sufixového stromu platí též o lcp-intervalech. S použitím sufixového pole se tedy úloha nalezení všech supermaximálních opakování zredukuje na úlohu nalezení lcp-intervalu, který nemá žádné vnořené lcp-intervaly a všechny znaky, které předcházejí přípony reprezentované prvky intervalu jsou různé. Průchod všemi lcp-intervaly sufixového pole lze realizovat průchodem zdola nahoru i průchodem shora dolů.

6.2.1 Experimentální výsledky

Ve své implementaci pro průchod sufixovým stromem používám rekurzivní průchod do hloubky. Při zpracovávání listu naleznu znak, který předchází jím reprezentovanou příponu a ten předávám ke zpracování rodiči. V uzlu, jehož všichni potomci jsou listy, pak tyto znaky z listů zkoumám, zda se mezi nimi nenachází žádný dvakrát.

Pro průchod pomocí lcp-intervalů jsem se snažil použít přístup co nejpodobnější přístupu použitém pro průchod sufixového stromu, aby naměřené hodnoty odpovídaly vlastnostem struktur a ne konkrétním implementačním detailům. Pomocí pole childtab a algoritmu 2 pro daný lcp-interval rekurzivně procházím všechny jemu vnořené lcp-intervaly. V okamžiku, kdy narazím na lcp-interval, který již nemá žádné vnořené lcp-intervaly, ověřím jsem, zda jsou všechny jím reprezentované přípony zleva různé.

Tabulka v příloze B.9 udává časy hledání supermaximálních opakování na reálných datech. Z tabulky je vidět, že ve většině případů je algoritmus využívající sufixové pole rychlejší, než algoritmus založený na sufixovém stromě. Pouze u několika málo souborů je tomu naopak. Vzorky *data.tar* a *data.zip* jsou příkladem dat, na kterých je výhodnější využít algoritmus se sufixovým stromem, než se sufixovým polem. Upozorněme, že tyto vzorky se od



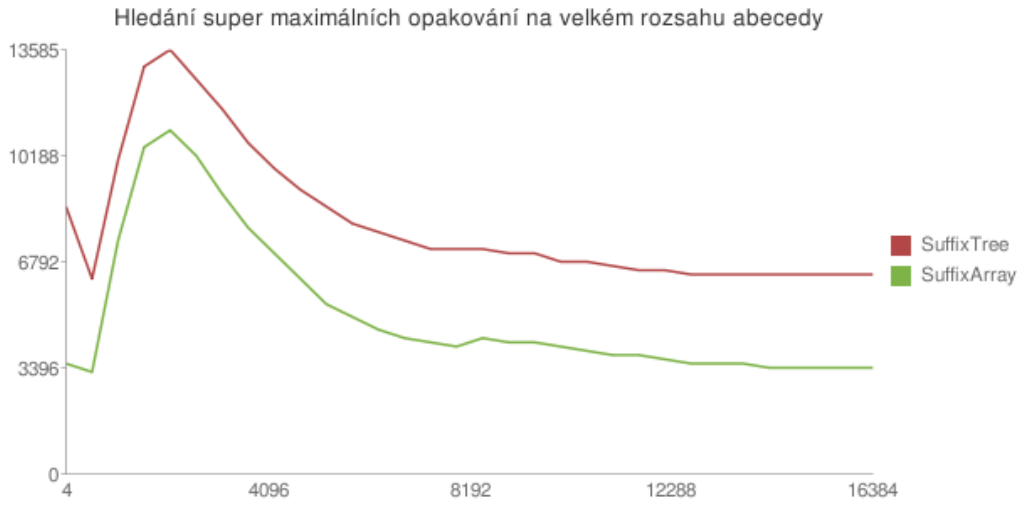
Obrázek 6.12: Závislost času hledání supermaximálních opakování v závislosti na velikosti abecedy vstupního řetězce. Na ose x je vyznačen počet znaků abecedy a osa y udává čas pěti hledání supermaximálních opakování v milisekundách.

ostatních odlišují vysokým procentem vysokých lcp hodnot, což je pravděpodobně důvodem, proč na těchto datech algoritmus založený na sufixovém poli selhává.

Pro názornější srovnání obou algoritmů jsem ještě provedl měření na náhodně generovaných datech. Obrázek 6.12 zobrazuje čas hledání supermaximálních opakování v závislosti na velikosti abecedy vstupního řetězce. Měření jsou provedena na datech délky $5 * 10^6$ znaků a pro jednotlivé velikosti abecedy jsem je prováděl pětkrát, abych snížil chybu způsobenou přesností měření a pseudonáhodností generovaných vzorků. Čas uvedený na ose y grafu na obrázku 6.12 udává celkový čas hledání na všech pěti vzorcích.

Graf je na první pohled dosti nevyrovnaný a kolísavý, ale i při vyšším počtu nezávislých měření je výsledek totožný. Navíc je dobře vidět, že oba algoritmy popisují tvarem velmi podobnou křivku. Z grafu lze také vyčíst, že algoritmus založený na sufixovém poli dává stejně jako při měření na reálných datech lepší výsledky, než algoritmus založený na sufixovém stromě.

Vzhledem ke kolísavosti grafu na malých abecedách jsem se rozhodl udělat ještě jedno měření, které by lépe odkrylo vlastnosti obou algoritmů na velkých abecedách. Opět jsem prováděl pět nezávislých měření na náhodně



Obrázek 6.13: Závislost času hledání supermaximálních opakování v závislosti na velikosti abecedy vstupního řetězce při velkém rozsahu abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas pěti hledání supermaximálních opakování v milisekundách.

generovaných vzorcích délky $5 * 10^6$ znaků. Výsledek měření zobrazuje graf na obrázku 6.13.

Graf potvrzuje, že časová náročnost algoritmu využívajícího sufixový strom je větší, než časová náročnost algoritmu využívajícího sufixové pole. Na grafu je také zajímavé pozorovat, že s rostoucí abecedou zpočátku roste čas potřebný k nalezení všech supermaximálních opakování. Od abecedy velikosti přibližně tisíc znaků se však grafy zlomí a s rostoucí abecedou začne čas potřebný k nalezení supermaximálních opakování klesat. Tento jev je možné pozorovat u obou sledovaných struktur.

6.3 Hledání nejdelšího společného podslova

Hledání nejdelšího společného podslova dvou či více slov je další z častých aplikací sufixového stromu. V této úloze však nebudeme pracovat pouze s jedním slovem, ale s množinou slov a to bude vyžadovat určité úpravy do námi používaných struktur. Nechť $M = \{S_1, S_2, \ldots, S_n\}$ je množina slov nad abecedou Σ. **Zobecněným sufixovým stromem** nad množinou M

nazveme takový sufixový strom, který reprezentuje všechny přípony všech slov množiny M.

Pro zjednodušení budeme velikost množiny M považovat za konstantní, protože předpokládáme, že počet slov množiny M je zanedbatelný vzhledem k délce jejich zřetězení. Existují i jiné aplikace, například slovníkové úlohy, kde tento předpoklad splněn není a konstrukci zobecněného sufixového stromu je nutné takovým podmínkám přizpůsobit.

Nalezení společného podslova množiny slov je s pomocí zobecněného sufixového stromu snadnou úlohou. Stačí naleznout vrchol, který má ve svém podstromě listy reprezentující přípony ze všech vstupních slov. K nalezení nejdelšího společného podslova poté stačí mezi vybranými vrcholy najít takový vrchol v, pro který platí, že délka zřetězení slov na cestě z kořene do v je maximální.

Jedním ze způsobů, jak konstruovat zobecněný sufixový strom, je ukončit každé slovo množiny M jiným oddělovacím znakem abecedy. Žádný z oddělovacích znaků se v žádném ze slov množiny M nesmí nacházet. Takto ukončená slova poté stačí zřetězit a vytvořit sufixový strom pro celé zřetězení $S_1\$S_2\#S_3@\ldots S_n\&$. Nevýhodou této konstrukce je, že pro n slov je nutné rezervovat n speciálních znaků abecedy a to může být implementačně nepříjemné. Pokud se velikost abecedy pohybuje v okolí hranice 256 znaků, může rozšíření abecedy vyžadovat dvoubytovou reprezentaci znaku řetězce místo jednobytové.

Druhým způsobem konstrukce je upravit konstrukční algoritmus tak, aby postupně přidal do stromu přípony všech slov. Tato varianta sice nevyžaduje žádné přídavné znaky abecedy, ale problém nastává v listech, protože několik přípon různých slov může končit ve stejném listě. V listech tedy nestačí uchovávat pořadí přípony, ale je nutné uchovávat seznam dvojic hodnot, kde každá dvojice identifikuje vstupní slovo a pořadí přípony v tomto slově.

Libovolný z obou konstrukčních algoritmů pracuje v čase $O(n)$, kde n je délka zřetězení slov. Nalezení nejdelšího společného podslova pomocí sufixového stromu tedy lze implementovat v čase $O(n)$, protože následný průchod stromem s nalezením nejdelšího společného podslova lze implementovat například průchodem do hloubky v čase $O(n)$.

Situace sufixového pole je velmi podobná situaci sufixového stromu. K řešení problému nalezení nejdelšího společného podslova množiny slov stačí sestavit zobecněné sufixové pole, které bude udávat pořadí všech přípon všech vstupních slov množiny M, naleznout lcp-intervaly obsahující přípony všech slov množiny M a mezi nimi vybrat ty, jejichž hodnota je maximální.

Úlohu konstrukce rozšířeného sufixového pole se mi nepodařilo v dostupných materiálech naleznout, nicméně dle mého názoru přicházejí v úvahu následující dvě varianty.

Jednou variantou, jak zkonstruovat rozšířené sufixové pole je vytvořit pole dvojic, kde každá dvojice udává identifikaci slova v množině M a pořadí přípony. Toto pole dvojic poté stačí setřídit pomocí libovolného třídícího algoritmu.

Druhou variantou je použít zřetězení slov a setřídit všechny přípony zřetězeného slova.

6.3.1 Implementace a experimentální výsledky

Ve své implementaci zobecněného sufixového stromu používám kompromisní variantu výše zmíněných možností. Všechna slova množiny M prodloužím o ukončovací znak \$ a poté provádím zřetězení těchto slov. Na toto zřetězení aplikuji Ukkonenův algoritmus [15] na konstrukci sufixového stromu s drobnou úpravou.

Úprava spočívá v tom, že v okamžiku, kdy algoritmus přidá do postupně vytvářeného stromu slovo $S_i\$$ včetně ukončovacího znaku \$, provedu změnu tohoto posledního přidaného znak \$ za jiný znak #. Vzhledem k tomu, že hrany sufixového stromu jsou ukazatelé do vstupního řetězce, toto přejmenování znaku se promítne do všech míst, kde se znak \$ nachází. V okamžiku, kdy algoritmus následně přidává slovo $S_{i+1}\$$, je pro něho znak \$ novým znakem, který se v textu dříve neobjevil, protože všechny předchozí \$ byly přejmenovány. Pro každé nově přidané slovo tedy algoritmus vytvoří nový list. Do tohoto listu je možné uložit buď dvě hodnoty udávající pořadí slova a pořadí reprezentované přípony v něm nebo pouze jednu hodnotu udávající index do zřetězení slov. V druhém případě však je nutné udržovat seznam poloh ukončovacích znaků jednotlivých slov ve zřetězení, aby k danému indexu do zřetězení bylo možné dohledat původní slovo.

Výhodou této implementace je, že nevyžaduje $|M|$ rezervovaných ukončovacích znaků, ale stačí mu pouze 2 ukončovací znaky. V listech také není nutné uchovávat seznam indexů, ale dostačující je jedna hodnota. Nevýhodou této implementace je, že je nutné udržovat seznam poloh ukončovacích znaků jednotlivých přípon, aby bylo možné rozlišit, ze kterého slova pochází která přípona. Některé vrcholy navíc mohou mít více hran do listů označených znakem #, s čímž je nutné počítat při následných průchodech stromem.

Při implementaci zobecněného sufixového pole nejprve provádím zřetězení všech slov množiny M, oddělených pomocí jednoho speciálního oddělovacího znaku \$ a poté setřídím všechny přípony tohoto zřetězení. K určení slova, ze kterého pocházejí jednotlivé přípony, využívám, stejně jako u sufixového stromu, seznamu poloh znaků \$ oddělujících jednotlivá slova.

Při implementaci obou struktur jsem se snažil dosáhnout řešení, ve kterém by se implementační detaily pokud možno co nejvíce podobaly a výsledky porovnání struktur byly co nejméně závislé na konkrétních implementačních detailech. Proto jsem v obou případech použil výše uvedené řešení se zřetězením vstupních slov a s referencováním jednotlivých vstupních slov pomocí poloh znaků oddělujících jednotlivá slova ve zřetězení.

Hledání nejdelších společných podslov v zobecněném sufixovém stromě jsem implementoval rekurzivním průchodem do hloubky. Při zpracovávání listu si algoritmus zaznamenává, do kterého slova patří přípona reprezentovaná listem. Během návratu z rekurze pak algoritmus pro každý vnitřní vrchol stromu určuje množinu slov, jejichž přípony jsou reprezentovány v listech jeho podstromu. Pokud množina obsahuje všechna vstupní slova, došlo k nalezení podslova společného všem vstupním slovům. Podle délky tohoto podslova jej buď označí jako nové nejdelší společné podslovo, nebo ho přidá k již nalezeným podslovům stejné délky, případně ho vyřadí jako nedostatečně dlouhé. Jako drobnou optimalizaci algoritmus vyřazuje z množiny zkoumaných vrcholů ty vnitřní vrcholy, jejichž výška je menší, než délka doposud nejdelšího nalezeného výsledku.

Hledání nejdelších společných podslov v zobecněném sufixovém poli provádím pomocí průchodu zdola nahoru (viz. algoritmus 1 v předchozí kapitole). Tento algoritmus postupně prochází lcp-intervaly sufixového pole v takovém pořadí, aby v okamžiku zpracovaní lcp-intervalu $[i \ldots j]$ již byly zpracovány všechny jemu vnořené intervaly.

Pokud dochází ke zpracovávání intervalu, který nemá žádné vnořené intervaly, průchodem všech jím reprezentovaných přípon určíme množinu všech vstupních slov, která obsahují podslovo společné celému intervalu. Při zpracovávání lcp-intervalu, který má vnořené podintervaly, již máme k dispozici informace o množinách vstupních slov reprezentovaných vnořenými podintervaly a jejich sjednocením získáme množinu vstupních slov reprezentovaných zpracovávaným intervalem. Pokud množina vstupních slov reprezentovaných příponami daného intervalu obsahuje všechna vstupní slova, je slovo reprezentované lcp-intervalem obsaženo ve všech vstupních slovech. Další postup je již shodný s postupem použitým při práci se sufixovým stro-

mem. Nalezené podslovo je buď nové nejdelší podslovo, nebo je přidáno k již dříve nalezeným společným podslovům stejné délky, nebo je zahozeno jako krátké.

Měření na reálných datech jsem prováděl pro tři, pět a deset slov. Jednotlivá slova jsem získal tak, že jsem data vstupního souboru rozdělil na tři, pět respektive deset dílů a prováděl jsem hledání nejdelší společné podposloupnosti těchto dílů. Výsledky provedených měření jsou zaznamenány v tabulce v příloze B.10.

Z tabulky je dobře vidět, že ve všech třech případech jsou časy naměřené algoritmu založenému na sufixovém poli menší, než časy naměřené algoritmu založenému na sufixovém stromě. V některých případech je tomu až desetkrát. Z tabulky je také možné pozorovat, že ve většině případů roste čas potřebný k nalezení nejdelší společné podposlopnosti s rostoucím počtem vstupních slov. Je tomu tak navzdory skutečnosti, že s rostoucím počtem slov klesá jejich délka, protože jsou všechna rozdělením jednoho vstupního slova.

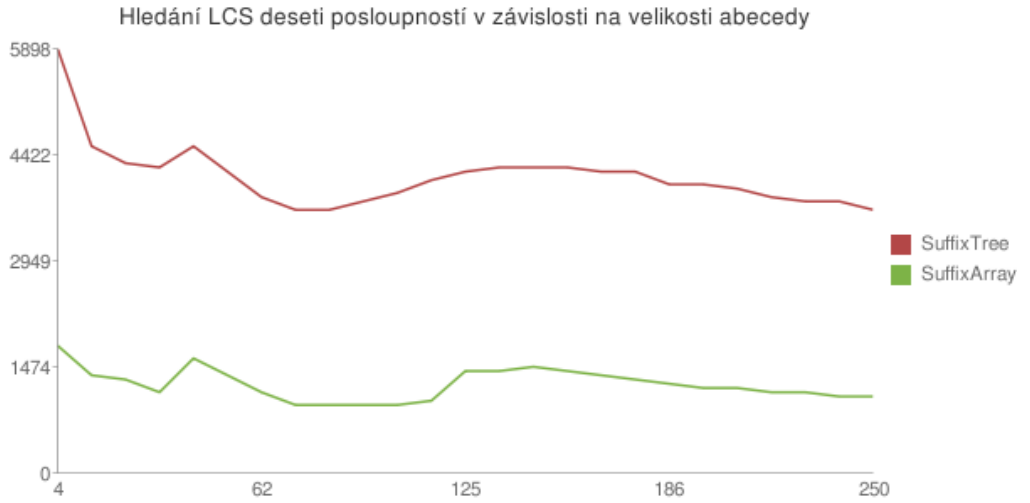
Za upozornění stojí vzorky *data.tar* a *data.zip*, které jako jediné vykazují výrazně lepší čas při použití algoritmu se sufixovým stromem než při použití algoritmu se sufixovým polem. Pravděpodobně je to opět způsobeno vysokými hodnotami a kompresí pole lcptab.

S náhodně generovanými daty jsem provedl tři měření. Nejprve jsem se zaměřil na srovnání algoritmů při hledání nejdelšího společného podslova tří slov v závislosti na velikosti vstupní abecedy, poté jsem měření opakoval s deseti slovy a na závěr jsem provedl měření závislosti času hledání nejdelší společné podposloupnosti v závislosti na počtu vstupních slov.

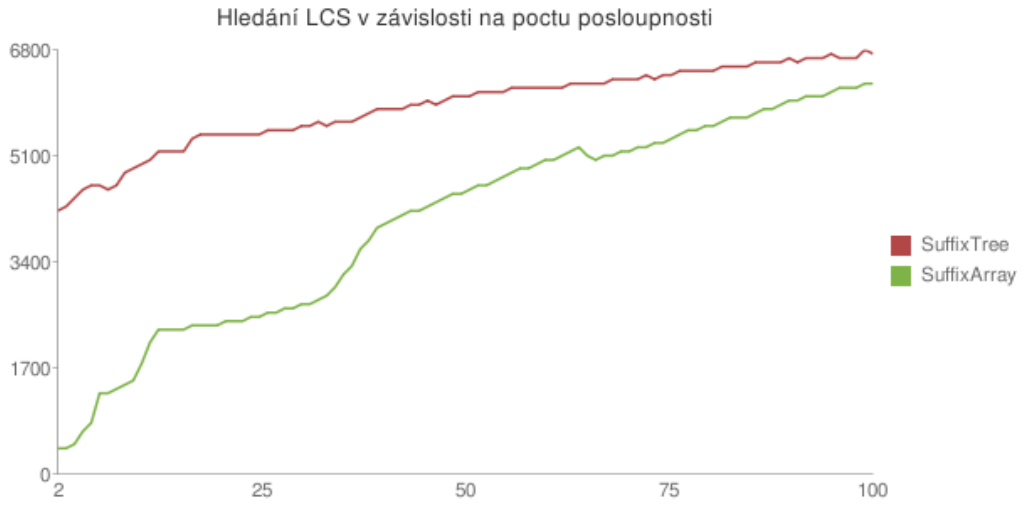
Grafy zobrazující výsledky prvních dvou měření je možné naleznout na obrázcích 6.14 a 6.15. Oba grafy potvrzují poznatky získané při měřeních provedených na reálných datech. Algoritmus založený na sufixovém poli je schopen úlohu řešit ve výrazně menším čase. Navíc je možné si všimnout, že čas hledání nejdelší společné podposloupnosti slov je téměř nezávislý na velikosti vstupní abecedy.

Třetí provedené měření je spíše orientační, protože použité implementace nejsou příliš vhodné pro vyhledávání nejdelšího společného podslova mnoha slov. Měření jsem provedl pro dvě až sto náhodně generovaných slov a výsledný graf je zobrazen na obrázku 6.16. Z obrázku je vidět, že graf zobrazující algoritmus založený na sufixovém poli roste strměji, než graf zobrazující algoritmus založený na sufixovém stromě a tím se zmenšuje rozdíl mezi oběma algoritmy. Do sta vstupních posloupností je však stále výhodnější užít algoritmus založený na sufixovém poli.

Obrázek 6.14: Závislost času hledání nejdelšího společného podslova tří slov v závislosti na velikosti abecedy vstupního řetězce. Na ose x je vyznačen počet znaků abecedy a osa y udává čas hledání nejdelšího společného podslova v milisekundách.



Obrázek 6.15: Závislost času hledání nejdelšího společného podslova deseti slov v závislosti na velikosti abecedy vstupního řetězce. Na ose x je vyznačen počet znaků abecedy a osa y udává čas hledání nejdelšího společného podslova v milisekundách.



Obrázek 6.16: Závislost času hledání nejdelšího společného podslova slov v závislosti na počtu vstupních slov při abecedě o velikosti deset znaků. Na ose x je vyznačena velikost množiny slov a osa y udává čas hledání nejdelšího společného podslova všech slov množiny v milisekundách.

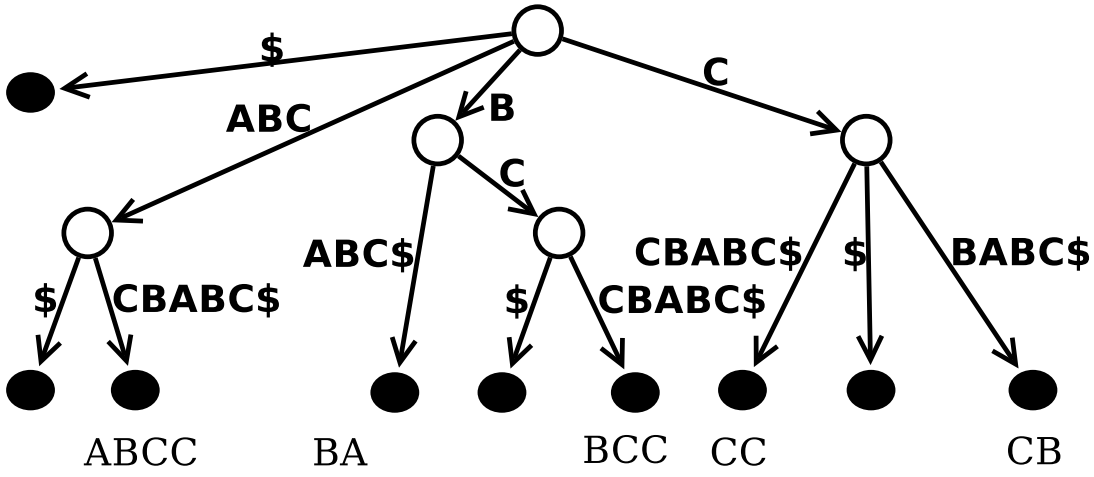
6.4 Hledání nejkratších unikátních podslov

Další zajímavou aplikací sufixového stromu je hledání nejkratších unikátních podslov.

Řekneme, že slovo T je **unikátním** podslovem slova S, pokud existuje právě jeden index i takový, že $S[i \dots i + |T| - 1] = T[0 \dots |T| - 1]$. Úlohou nalezení **nejkratších unikátních podslov** je nalezení všech unikátních podslov slova S, jejichž délka je minimální.

Například nejkratšími unikátními podslovy slova *abccbabc* jsou slova *cc*, *cb* a *ba*. Všechny se ve slově *abccbabc* nacházejí právě jednou, žádné unikátní podslovo délky 1 neexistuje a žádné další unikátní podslovo délky 2 také neexistuje.

V sekci 6.2 týkající se úlohy hledání supermaximálních opakování jsme ukázali, že libovolný vnitřní vrchol stromu reprezentuje opakující se podslovo vstupního slova. Jistě není těžké nahlédnout, že libovolná předpona slova, které se ve vstupním slově opakuje, je také opakováním. Unikátní podslova tedy nemohou být předpony slov reprezentovaných vnitřními vrcholy, ale musejí to být jejich prodloužení.



Obrázek 6.17: Nejkratší unikátní předpony přípon slova *abccbabc* reprezentované listy sufixového stromu

Nechť u je libovolný list a v je jeho rodič v sufixovém stromě. Nechť a je první znak hrany z v do u různý od znaku \$ a slovo A je slovo reprezentované vrholem v. Nyní uvažujme slovo B, které vznikne zřetězením slova A se znakem a. Toto slovo je jistě podslovem vstupního slova, protože je předponou slova, které je reprezentováno listem u. B je navíc unikátní, protože pokud by se ve vstupním slově vyskytovalo více než jednou, pak by existovaly alespoň dva listy jimiž reprezentovaná slova mají předponu B. Vzhledem k tomu, že jsme slovo A prodloužili pouze o jeden znak, je slovo B nejkratší unikátní předpona slova reprezentovaného listem v.

Příklad nejkratších unikátních předpon slov reprezentovaných listy pro slovo *abccbabc* je znázorněn na obrázku 6.17.

Předchozí úvaha nám tedy dává postup, jak naleznout všechna nejkratší unikátní podslova. Hledáme takové listy u, pro které platí, že slovo reprezentované rodičem u má minimální délku. Každý nalezený list u nám reprezentuje jedno unikátní podslovo, které je předponou přípony reprezentované listem u a jeho délka je délka slova reprezentovaného rodičem u zvětšená o jedničku.

Použití sufixového pole v této aplikaci se opět opírá o lcp-intervaly. Již jsme ukázali, že vlastnosti lcp-intervalů jsou podobné vlastnostem vnitřních vrcholů sufixového stromu. Zbývá nám však ještě ukázat, jak je možné

pomocí lcp-intervalů provádět operace, ke kterým se v sufixovém stromě používají jeho listy.

Algoritmus 2 k danému lcp-intervalu $[i \dots j]$ vrací seznam jemu vnořených podintervalů. Některé tyto podintervaly jsou lcp-intervaly a některé mohou být jednoprvkové. Jednoprvkový interval vznikne například pokud lcp-interval obsahuje dva bezprostředně následující pevné body.

O lcp-intervalech víme, že lcp-interval $[i \dots j]$ s hodnotou l reprezentuje množinu přípon, které mají společnou předponu délky l. Pokud je podinterval lcp-intervalu jednoprvkový, reprezentuje příponu, která se na $l + 1$-ní pozici neshoduje s žádnou jinou příponou lcp-intervalu $[i \dots j]$. Pokud nemá lcp-interval žádné lcp-podintervaly, pak se všechny jím reprezentované přípony liší na $l + 1$-ní pozici a jsou tedy jednoprvkové.

Úlohou nalezení nejkratších unikátních podslov pomocí sufixového pole tedy znamená naleznout jednoprvkové intervaly a vybrat mezi nimi takové, které jsou podintervalem intervalů s minimální hodnotou.

6.4.1 Implementace a experimentální výsledky

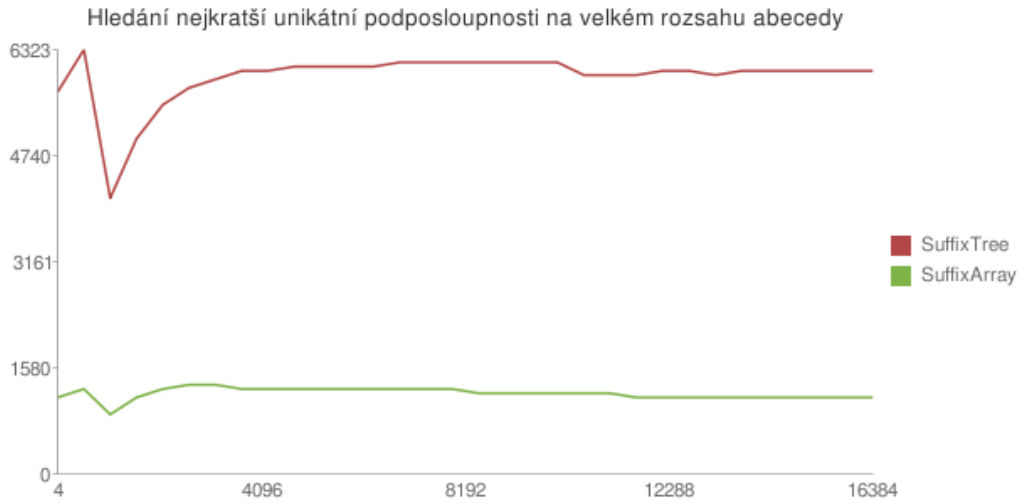
Implementačně nejvhodnějším algoritmem k hledání nejkratších unikátních podslov je průchod stromu do šířky. Pokud se nám totiž podaří naleznout nejkratší unikátní podslovo délky k, nemusíme už dále procházet vrcholy, které reprezentují slova delší než k. Tyto vrcholy sice mohou reprezentovat unikátní podslova, ale ne nejkratší unikátní podslova.

Pro hledání nejkratších unikátních podslov pomocí sufixového pole platí stejná myšlenka jako u sufixového stromu. Je vhodné procházet lcp-intervaly průchodem do šířky, hledat jednoprvkové intervaly a nezpracovávat intervaly, jejichž hodnota je vyšší, než hodnota nejkratšího doposud nalezeného unikátního podřetězce.

Výsledky měření hledání nejkratších unikátních podslov na reálných datech je možné naleznout v tabulce B.11. Drobnou nepříjemností tohoto měření je, že časy běhu algoritmu jsou velmi malé a často nerozlišitelné od 0 ms. Bylo by možné měření opakovat vícekrát a měřit součet časů, aby byly hodnoty lépe měřitelné, nicméně opakování měření na jedné datové struktuře by mohlo zkreslit výsledky kvůli procesorovým cache a podobným vlivům. Opakovaná měření proto provádím pouze s náhodně generovanými daty vždy pro jinou vstupní posloupnost a nově vytvořenou datovou strukturu.

I přes nízké naměřené časy je z tabulky dobře vidět, že čas hledání pomocí sufixového pole je výrazně nižší, než čas hledání pomocí sufixového stromu.

Obrázek 6.18: Závislost času hledání nejkratších unikátních podslov v závislosti na velikosti abecedy vstupního řetězce. Na ose x je vyznačen počet znaků abecedy a osa y udává čas pěti nezávislých hledání v milisekundách.



Obrázek 6.19: Závislost času hledání nejkratších unikátních podslov v závislosti na velikosti abecedy vstupního řetězce při velkém rozsahu abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas pěti nezávislých hledání v milisekundách.

Pouze při hledání na textech *data.tar* a *data.zip* se algoritmu, který využívá sufixový strom, podařilo překonat algoritmus založený na sufixovém poli, ale ve všech ostatních případech je použití sufixového pole výhodnější.

Měření provedená na náhodně generovaných datech jsou podobně jako při měření supermaximálních opakování dosti kolísavá. Opět jsem tato měření několikrát opakoval na různě dlouhých vstupních datech s různým počtem opakování měření, ale výsledky jsou vždy téměř totožné. Měření uvedené na obrázku 6.18 bylo prováděno na náhodně generovaných posloupnostech délky $5 * 10^6$ znaků a stejně jako při měření supermaximálních opakování jsem pro každou velikost abecedy provedl pět nezávislých měření na nezávisle generovaných vstupních posloupnostech. Graf zaznamenává celkový čas potřebný k nalezení všech pěti nejkratších unikátních podslov.

Z grafu je velmi zřetelně vidět rozdíl mezi algoritmem využívajícím sufixový strom a algoritmem využívajícím sufixové pole. Stejně jako při měření na reálných datech i zde je algoritmus využívající sufixové pole značně rychlejší na celém měřeném rozsahu abecedy.

Abych ověřil platnost zjištěných vlastností algoritmů i na velkém rozsahu abecedy, provedl jsem ještě jedno měření, jehož graf je na obrázku 6.19. Jedná se o graf času pěti hledání nejkratších unikátních podslov v závislosti na velikosti abecedy. Tentokrát jsem volil abecedu v rozsahu od čtyř do přibližně $16*10^3$ znaků a posloupnosti délky $5*10^6$ znaků. Z grafu je zřetelně vidět, že rozdíl ve výkonnosti algoritmů, který jsme pozorovali pro malé abecedy platí i pro velké abecedy. Navíc je z grafu čitelné, že čas hledání nejkratšího unikátního podslova je téměř nezávislý na velikosti abecedy a to jak pro algoritmus využívající sufixové pole, tak pro algoritmus využívající sufixový strom.

6.5 Výpočet matching statistics

Matching statistics byly zavedeny v [6] k řešení úlohy aproximativního vyhledávání v sublineárním čase za pomoci sufixového stromu.

Nechť S a T jsou dvě libovolná slova nad abecedou Σ. **Matching statistics** slova T vzhledem ke slovu S je dvojice číselných vektorů $Len(T, S)$ a $Pos(T, S)$. Délka obou vektorů je $|T|$.

$Len(T, S)[i]$ udává délku nejdelšího podslova slova T, které začíná ve slově T na pozici i a zároveň je podslovem slova S. $Pos(T, S)[i]$ udává pozici začátku libovolného výskytu výše zmíněného podslova ve slově S.

i	0	1	2	3	4	5	6	7	8	9	10
$T[i]$	I	P	P	I	S	S	I	S	S	I	M
$Len(T,S)[i]$	4	3	2	7	6	5	4	3	2	1	1
$Pos(T,S)[i]$	7	8	9	1	2	3	4	5	6	7	0

Obrázek 6.20: Matching statistics slova `IPPISSISSIM` vzhledem ke slovu `MISSISSIPPI`

Pro každé $0 \leq i \leq |T| - 1$ tedy platí:

$$T[i \ldots i+Len(T,S)[i]-1] = S[Pos(T,S)[i] \ldots Pos(T,S)[i]+Len(T,S)[i]-1]$$

K výpočtu matching statistics použijeme algoritmus uvedený v [6], využívající sufixového stromu a pracující v čase $O(|S| + |T|)$.

V původní definici matching statistics v [6] je číselný vektor Pos nahrazen vektorem ukazatelů na vrcholy sufixového stromu pro slovo S reprezentující výskyty $T[i \ldots i + Len(T,S)[i] - 1]$ ve slově S. Abychom z vektoru ukazatelů na vrcholy sufixového stromu získali vektor Pos, využijeme následující postup.

Mějme $Len(T,S)[i]$ a vrchol v_i, který reprezentuje výskyty hledaného podslova. Pokud je v_i list, pak reprezentuje právě jedno slovo a tím je přípona, jejíž počátek je v listu udržován. $Pos(T,S)[i]$ tedy nastavíme na počátek této přípony.

Pokud je v_i vnitřní vrchol, pak z něj vycházejí alepoň dvě hrany. Vyberme libovolnou z nich a nechť p je pozice jejího počátku ve slově S. Pak jeden z výskytů hledaného podslova začíná na pozici $Pos(T,S)[i] = p - Len(T,S)[i]$, protože slovo $S[p - Len(T,S)[i] \ldots p - 1]$ je slovo reprezentované vrcholem v_i.

Druhou možností, jak naleznout hledanou pozici pro vnitřní vrchol v_i je vzít libovolný list v jeho podstromě a použít začátek jím reprezentované předpony. Průchod do listu však není možné realizovat v konstatním časem, a proto je první zmíněná varianta efektivnější.

Nyní se vrátíme k algoritmu pro výpočet matching statistics. Algoritmus nejprve sestrojí sufixový strom pro slovo S libovolným algoritmem, který ke konstrukci používá sufixové odkazy. Je možné použít například Ukkonenův algoritmus [15] nebo McCreightův algoritmus [10].

Po konstrukci stromu algoritmus postupně pro každé $0 \leq i \leq |T| - 1$ spočítá $Len(T,S)[i]$, nalezne vrchol v_i a z něho výše uvedeným postupem určí $Pos(T,S)[i]$.

Naivním způsobem, jak vypočítat všechny uvedené hodnoty, je pro každé i procházet sufixovým stromem od kořene směrem do listů a hledat nejdelší předponu slova $T[i \dots |T|-1]$, která se ve slově S nachází. Vrcholem v_i pak označíme poslední navštívený vrchol. Tento algoritmus však pracuje v čase $O(|S| * |T|)$. Vylepšení naivního algoritmu spočívá ve využití sufixových odkazů definovaných v sekci 3.4 a je velmi podobné myšlence použité v Ukkonenově algoritmu [15] na konstrukci sufixového stromu.

Pro připomenutí zopakujme, že sufixový odkaz nalezne pro vrchol u reprezentující slovo x takový vrchol v, který reprezentuje slovo x zkrácené o první znak.

Při výpočtu $Len(T,S)[0]$ a v_0 algoritmus pracuje stejně jako naivní algoritmus a prochází celý strom postupným porovnáváním znaků od kořene. Při výpočtu $Len(T,S)[i]$ a v_i pro $i > 0$ již algoritmus zná v_{i-1} a o vrcholu $u = suflink(v_{i-1})$ ví, že reprezentuje předponu slova $T[i \dots |T|-1]$, a proto nemusí procházet celým stromem od kořene, ale může začít průchod až ve vrcholu u.

Korektnost a důkaz časové složitosti algoritmu je možné naleznout v [6].

Nyní se zaměříme na výpočet matching statistics pomocí sufixového pole. Nejprve je nutné sestavit sufixové pole suftab pro slovo S rozšířené o pomocná pole lcptab, childtab a suflinktab. Algoritmus poté pracuje velmi podobně jako algoritmus využívající sufixový strom. Algoritmem průchodu shora dolů prochází lcp-intervaly a pro každé $0 \leq i \leq |T|-1$ hledá nejdelší předponu slova $T[i \dots |T|-1]$, která se nachází ve slově S.

Nechť lcp-interval $[i_1 \dots j_1]$ s hodnotou l_1 je poslední lcp-interval navštívený při průchodu shora dolů. Tento interval tedy reprezentuje slovo odpovídající prefixu slova $T[i \dots |T|-1]$ a žádný lcp-interval s hodnotou větší než l_1 již nereprezentuje slovo tvořící prefix slova $T[i \dots |T|-1]$.

Nechť interval $[k \dots k]$ je jednoprvkovým podintervalem lcp-intervalu $[i_1 \dots j_1]$, který reprezentuje předponu, jež se se slovem $T[i \dots |T|-1]$ shoduje na $l_k > l_1$ znacích. Pokud takový interval existuje, pak $Len(T,S)[i] = l_k$ a $Pos(T,S)[i] = suftab[k]$. Pokud neexistuje, tak $Len(T,S)[i] = l_1$ a $Pos(T,S)[i] = suftab[i]$.

6.5.1 Implementace a experimentální výsledky

Implementaci algoritmu využívajícího sufixový strom jsem provedl přesně podle výše popsaného postupu.

Implementaci založenou na sufixovém poli jsem provedl ve dvou variantách. První variantu jsem čerpal z článku [1] a budu ji označovat stejně jako její tvůrci *greedymatch*, ve druhé variantě jsem se pokusil algoritmus *greedymatch* mírně vylepšit a urychlit.

Algoritmus *greedymatch* reprezentuje vyhledávací polohu v sufixovém poli pomocí trojice $([i \ldots j], q, [r \ldots s])$, kde $[i \ldots j]$ je lcp-interval s hodnotou l a platí jedna z následujících tří vlastností:

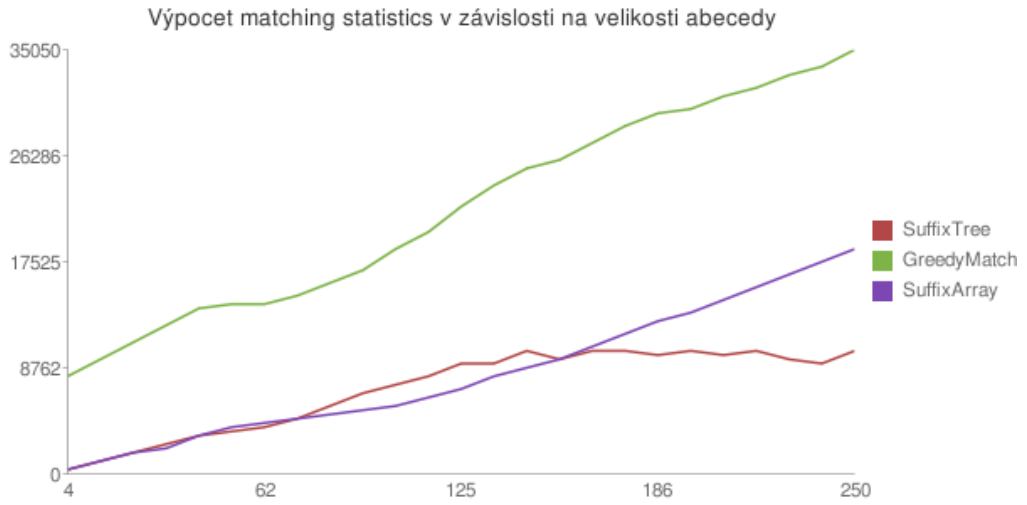
- $q = l$ a $[i \ldots j] = [r \ldots s]$
- $[r \ldots s]$ je vnořený lcp-interval intervalu $[i \ldots j]$ s hodnotou m a platí $l < q < m$
- $[r \ldots s]$ je jednoprvkový podinterval intervalu $[i \ldots j]$ a platí $l < q < n - suftab[l]$.

Tato trojice jednoznačně definuje právě jedno podslovo vstupního slova S, konkrétně $S[suftab[l] \ldots suftab[l] + q - 1]$ a algoritmus je s touto reprezentací schopen postupně procházet všechny přípony slova T přesně dle algoritmu popsaného v předchozí sekci.

Poté, co jsem algoritmus *greedymatch* implementoval, překvapilo mě o kolik je pomalejší, než implementace využívající sufixový strom. Za pomoci Linuxového profilovacího nástroje gprof [11] jsem odhalil, že problémem této implementace je velmi časté počítání hodnoty lcp-intervalů. Hodnotu lcp-intervalu si totiž algoritmus *greedymatch* ve své poloze neudržuje, a proto ji je nutné vypočítat vždy, když chceme přistupovat ke znakům odlišujícím jednotlivé vnořené intervaly lcp-intervalu. Provedl jsem tedy implementaci, která tento nedostatek odstraňuje a hodnotu lcp-intervalů si uchovává v paměti a aktualizuje ji průběžně během celého výpočtu stejně tak, jako si sufixový strom udržuje výšku posledního navštíveného uzlu.

Všechny tři algoritmy jsem porovnal na reálných i na náhodných datech.

Při měření na reálných datech jsem provedl výpočet matching statistics 100 generovaných posloupností délky 10^4 znaků vzhledem k reálným posloupnostem. Výsledky tohoto měření je možné nahlédnout v tabulce v příloze B.12. Sloupec *Suffix Tree* udává hodnoty naměřené při použití sufixového stromu, sloupec *SA Greedy* hodnoty naměřené při použití algoritmu



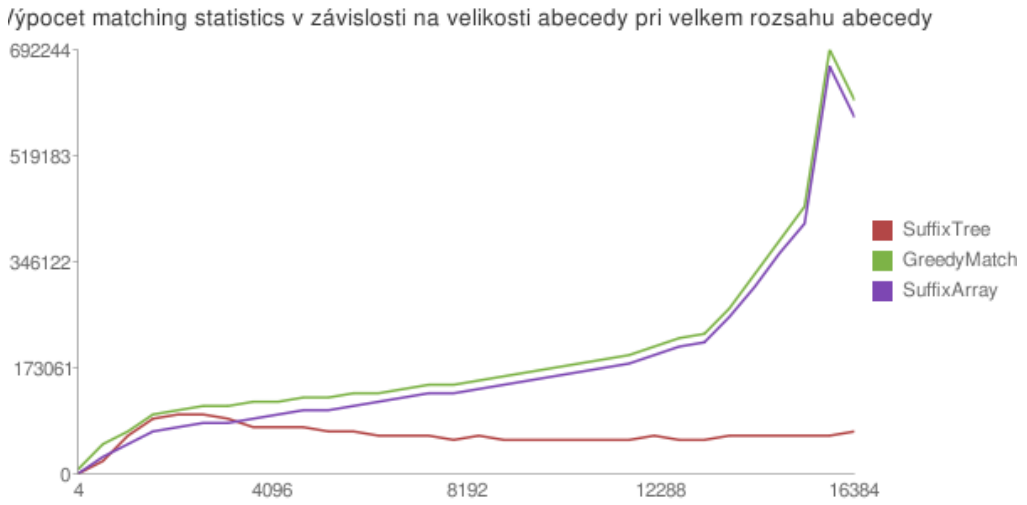
Obrázek 6.21: Závislost času výpočtu matching statistics na velikosti abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas výpočtu matching statistics 100 náhodně generovaných slov délky 10^4 vzhledem k náhodně generovanému slovu délky $5 * 10^6$ v milisekundách.

greedymatch a sloupec *SA Optim* udává hodnoty naměřené při použití vylepšeného algoritmu pro sufixové pole.

Z tabulky je vidět, že v téměř na všech vzorcích je výpočet matching statistics za pomoci sufixového stromu rychlejší než obě varianty založené na sufixovém poli. Pouze u souboru *pi.txt* je algoritmus využívající sufixové pole rychlejší. Z tabulky je také vidět obrovský rozdíl mezi původní variantou *greedymatch* a jeho vylepšenou variantou.

Měření na náhodně generovaných datech jsem podobně jako u všech předchozích aplikací prováděl v závislosti na velikosti abecedy, protože je to jeden z parametrů, který se zdá mít na všechny používané struktury významný vliv.

První měření jsem provedl pro malé abecedy v rozsahu 4 až 250 znaků a prováděl jsem výpočet matching statistics 100 náhodně generovaných slov délky 10^4 vzhledem k náhodně generovanému slovu délky $5 * 10^6$. Výsledek zobrazuje graf na obrázku 6.21. Na grafu se potvrdily výsledky zjistěné při měření nad reálnými daty a ukázalo se, že sufixový strom dává na většině případů nejlepší výsledky. Sufixové pole je lepší než sufixový strom pouze v omezeném intervalu abeced a to přibližně od abecedy velikosti 80 znaků



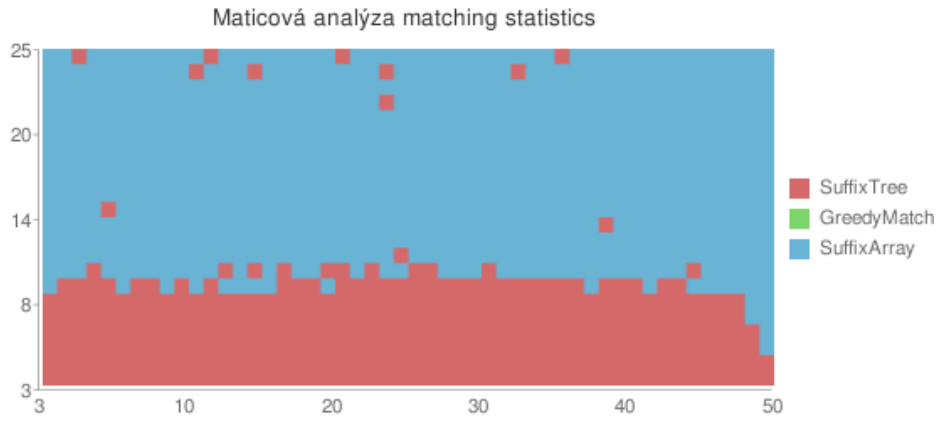
Obrázek 6.22: Závislost času výpočtu matching statistics na velikosti abecedy. Na ose x je vyznačen počet znaků abecedy a osa y udává čas výpočtu matching statistics 100 náhodně generovaných slov délky 10^4 vzhledem k náhodně generovanému slovu délky $5 * 10^6$ v milisekundách.

do abecedy velikosti 140 znaků. Z grafu je také vidět citelný rozdíl mezi algoritmem *greedymatch* a jeho vylepšenou variantou.

Druhé měření na náhodně generovaných datech jsem provedl se stejnými velikostmi dat, ale pro velký rozsah abecedy a výsledný graf je zobrazen na obrázku 6.22. Z tohoto měření ještě výrazněji vynikají výsledky sufixového stromu a ukazuje se, že při velkých abecedách je sufixové pole pro tuto aplikaci zcela nevhodné. Dokonce i provedené optimalizace při vyšších abecedách ztrácejí význam.

Vzhledem k tomu, že obě předchozí měření nezohledňují délku slova, pro které jsou matching statistics vypočítávány a z grafů navíc není příliš zřetelné, který z algoritmů dává nejlepší výsledky pro malé abecedy, rozhodl jsem se udělat ještě jedno měření, ve kterém jsem se zaměřil na tyto parametry. Graf na obrázku 6.23 zobrazuje maticové měření času výpočtu matching statistics v závislosti na šířce abecedy a velikosti vstupního slova vůči náhodně generovanému slovu délky $5 * 10^6$ znaků.

Z grafu je vidět, že pro abecedy do velikosti přibližně 10 znaků je lepší využít sufixový strom, ale pro abecedy v intervalu 10 až 25 znaků je vhodnější sufixové pole. Z předchozích měření však víme, že od abecedy velikosti



Obrázek 6.23: Maticové měření výpočtu matching statistics v závislosti na velikosti abecedy a délce slova, pro které se matching statistics vypočítávají. Na ose x je délka slova, pro které jsou matching statistics vypočítávány v 10^3 znaků a osa y udává velikost abecedy. Výpočet matching statistic probíhal vůči slovu délky $5 * 10^6$ znaků.

přibližně 4000 znaků již je sufixové pole zcela nevhodné a za sufixovým stromem začne velmi rychle ztrácet. Z grafu je navíc dobře pozorovatelné, že vhodnost jednotlivých algoritmů je v měřeném rozsahu $3 * 10^3$ až $5 * 10^4$ téměř nezávislá na délce slova, pro které jsou matching statistics vypočítávány.

Celkově si myslím, že pro řešení úlohy výpočtu matching statistics je výhodnější použít algoritmus založený na sufixovém stromě. Pro malé abecedy je rozdíl mezi oběma algoritmy zanedbatelný a pro velké abecedy je užití sufixového stromu výrazně lepší. Určitou výhodou sufixového pole v této aplikaci je jeho menší paměťová náročnost. Sufixové pole totiž pro tuto aplikaci vyžaduje pouze sedmnáct bytů na jeden znak vstupního textu, ale sufixový strom vyžaduje dvacet až třicet bytů na jeden znak vstuního textu.

Kapitola 7

Závěr

V této práci jsem se zaměřil na srovnání sufixového stromu a sufixového pole z hlediska jejich aplikací. Vybral jsem pět různých aplikací sufixového stromu a implementoval algoritmy, které dané aplikace řeší pomocí obou datových struktur. U aplikace hledání vzorku jsem řešení rozšířil ještě o algoritmus využívající datovou strukturu CDAWG. Algoritmy jsem testoval na různých datech běžně používaných v praxi a na náhodně generovaných posloupnostech znaků.

Ukázalo se, že pro hledání výskytu vzorku či všech jeho výskytů je nejvhodnější použít algoritmus založený na sufixovém poli bez dalších pomocných struktur, který nejen že je paměťově zcela nejlepší, ale ve většině měření se ukazuje i jako nejrychlejší. Naopak algoritmus založený na rozšířeném sufixovém poli se v této aplikaci ukazuje jako zcela nevhodný.

V aplikacích hledání supermaximálních opakování, nejkratších unikátních podslov a nejdelšího společného podslova se rozšířené sufixové pole ukazuje jako prostorově úspornější a časově výhodnější datová struktura než sufixový strom.

Jedinou aplikací, pro kterou se ukázalo jako vhodnější využít sufixový strom je výpočet matching statistics. Rozšířené sufixové pole pro tuto aplikaci již vyžaduje množství paměti srovnatelné s potřebami sufixového stromu a z hlediska časové složitosti je sufixový strom výhodnější. V aplikaci hledání všech výskytů vzorku na reálných datech se sufixový strom ukázal jako konkurence schopný sufixovému poli z hlediska časové složitosti, ale z hlediska paměťové složitosti je mnohokrát náročnější.

Celkově se nám podařilo ukázat, že sufixový strom je možné v mnoha aplikacích nahradit rozšířeným sufixovým polem a tato náhrada navíc přináší značné výhody.

Oblastí, ve které je sufixový strom sufixovým polem doposud nenahraditelným, jsou aplikace využívající sufixový strom nad posuvným oknem [12]. Sufixové pole je na rozdíl od stromu velmi statickou datovou strukturou a možnosti dynamických modifikací v něm zatím nejsou objasněny.

V oblasti experimentálních měření by ještě bylo zajímavé zaměřit se na porovnání obou struktur z hlediska časové složitosti jejich konstrukce. Čas konstrukce totiž může hrát velkou roli při rozhodování o vhodnosti dané struktury v aplikacích, kde je struktura použita jen několikrát nebo dokonce pouze jednou. Příkladem takové aplikace je hledání supermaximálních opakovaní. Několik poznatků sice ve své práci přináší Pavel Žoha [16], ale jeho práce pokrývá pouze konstrukci sufixového pole bez pomocných polí lcptab, childtab a suflinktab.

Literatura

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, Enno Ohlebusch, Replacing suffix trees with enhanced suffix arrays, J. Discrete Algs. 2 (2004), 53-86.

[2] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, Enno Ohlebusch, The enhanced suffix array and its application to genome analysis, Proc. Second Workshop on Algorithms in Bioinformatics, LNCS 2452 (2002), 449-463.

[3] The Canterbury Corpus, http://corpus.canterbury.ac.nz

[4] M. Crochemore, R. Vérin, On compact directed acyclic word graphs, Structures in Logic and Computer Science, Lecture Notes in Computer Science, vol. 1261, Springer, Berlin, 1997, 192-211.

[5] Dan Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.

[6] W.I.Chang, E.L. Lawler, Sublinear expected time approximate string matching and biological applications, Algorithmica 12 (4-5) 327-344 (1994).

[7] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, Lecture Notes in Computer Science, vol 2089, Springer-Verlag, Berlin, 2001, pp. 181-192.

[8] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, G. Pavesi: On-Line Construction of Compact Directed Acyclic Word Graphs, Discrete Applied Mathematics 146(2005), 156-179.

[9] U. Manber and E. W. Myers, Sufix arrays: A new method for on-line string searches, SIAM Journal on Computing, 22(5), 935-948 (1993).

[10] E.McCreight, A space-economical suffix tree construction algorithm. Jurnal of the ACM, 23(2):262-272, Apr. 1976.

[11] The GNU Profiler, http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

[12] M. Senft, Suffix Tree for a Sliding Window: An Overview, In: Šafránková, J. (ed.) WDS 2005, pp. 41-46, Matfyzpress, Praha 2005.

[13] Bill Smyth, Computing Patterns in Strings, Addison Wesley, 2003.

[14] Jindřich Šedek, Konstrukční algoritmy pro sufixové datové struktury, bakalářská práce MFF UK, Praha 2007.

[15] E. Ukkonen, On-line construction of suffix trees. Algorithmica 14(3):249-260, 1995

[16] Pavel Žoha, Algoritmy konstrukce sufixového pole, diplomová práce MFF UK, Praha 2006.

Příloha A

Obsah přiloženého CD

- Zdrojové kódy implementace všech algoritmů v jazyce C včetně Makefile.
- Dokumentace ke zdrojovým kódům implementace v jazyce C.
- Zdrojové kódy tohoto textu pro sázecí systém LaTeX.
- Použitá testovací data.
- Výsledky provedených měření v podobě označkovaných souborů ve formátu LaTeX, které jsou použity při vytváření tabulek v následující příloze této práce.
- Soubor README obsahující podrobnější instrukce, jak zkompilovat zdrojové kódy, spustit testy implementovaných algoritmů, či spustit měření vlastností algoritmů včetně generování grafů použitých v této práci.

Příloha B

Výsledky měření

Zdroj	$\|\Sigma\|$	n (B)	Max_LCP	Big LCPs	
				count	promiles
alice29.txt	74	152089	177	0	0
asyoulik.txt	68	125179	147	0	0
bib	81	111261	156	0	0
bible.txt	63	2097152	551	2752	1
book1	82	768771	104	0	0
book2	96	610856	246	0	0
cp.html	86	24603	141	0	0
data.tar	256	2498560	149881	1295737	518
data.tar.bz2	256	967840	38	0	0
data.tar.gz	256	1270025	27363	123968	97
data.zip	256	1387111	148048	627204	452
E.coli	4	4638690	2815	36600	7
fields.c	90	11150	195	0	0
geo	256	102400	61	0	0
lcet10.txt	84	426754	228	0	0
news	98	377109	1029	3417	9
paper1	95	53161	104	0	0
paper2	91	82199	115	0	0
paper3	84	46526	48	0	0
paper4	80	13286	36	0	0
pi.txt	10	1000000	12	0	0
plrabn12.txt	81	481861	163	0	0
progc	92	39611	156	0	0
progl	87	71646	560	627	8
progp	89	49379	1631	2592	52
sum	255	38240	738	3013	78
trans	99	93695	1706	3911	41
world192.txt	90	1572864	559	2290	1

Tabulka B.1: Vlastnosti reálných vstupních dat. $|\Sigma|$ - velikost abecedy, `n` - délka souboru dat v bytech, `Max_LCP` - maximální nejdelší společná předpona lexikograficky sousedních dvojic přípon, `Big LCPs` - počet dvojic lexikograficky sousedních přípon, jejichž nejdelší společná předpona je delší než 254 (`count` jako celkový počet a `promiles` jako počet promile mezi všemi dvojicemi sousedních přípon).

Zdroj	$\|\Sigma\|$	n (B)	Max_LCP	Big LCPs count	Big LCPs promiles
random	3	5000000	29	0	0
random	4	5000000	24	0	0
random	5	5000000	18	0	0
random	6	5000000	17	0	0
random	7	5000000	15	0	0
random	8	5000000	15	0	0
random	9	5000000	15	0	0
random	10	5000000	13	0	0
random	15	5000000	11	0	0
random	20	5000000	10	0	0
random	25	5000000	9	0	0
random	30	5000000	9	0	0
random	35	5000000	8	0	0
random	40	5000000	8	0	0
random	45	5000000	7	0	0
random	50	5000000	7	0	0
random	60	5000000	7	0	0
random	70	5000000	6	0	0
random	80	5000000	7	0	0
random	90	5000000	6	0	0
random	100	5000000	6	0	0
random	120	5000000	6	0	0
random	140	5000000	6	0	0
random	160	5000000	6	0	0
random	180	5000000	5	0	0
random	200	5000000	5	0	0
random	220	5000000	5	0	0
random	240	5000000	5	0	0

Tabulka B.2: Vlastnosti gererovaných vstupních dat. $|\Sigma|$ - velikost abecedy, `n` - délka souboru dat v bytech, `Max_LCP` - maximální nejdelší společná předpona lexikograficky sousedních dvojic přípon, `Big LCPs` - počet dvojic lexikograficky sousedních přípon, jejichž nejdelší společná předpona je delší než 254 (`count` jako celkový počet a `promiles` jako počet promile mezi všemi dvojicemi sousedních přípon).

Zdroj	$\|\Sigma\|$	n (B)	Suffix Tree states	Suffix Tree trans	CDAWG states	CDAWG trans
alice29.txt	74	152089	80860	232947	41292	137895
asyoulik.txt	68	125179	62746	187923	35310	120446
bib	81	111261	59845	171104	20569	71641
bible.txt	63	2097152	1179962	3277112	429342	1395370
book1	82	768771	385283	1154052	231364	777811
book2	96	610856	324528	935382	148741	498377
cp.html	86	24603	12813	37414	4588	16832
data.tar	256	2498560	1748291	4246849	173079	875352
data.tar.bz2	256	967840	96945	1064783	96490	1063865
data.tar.gz	256	1270025	238482	1508505	110828	1253161
data.zip	256	1387111	780943	2168052	86248	758298
E.coli	4	4638690	2978798	7617486	2491062	6613302
fields.c	90	11150	6588	17736	1963	6783
geo	256	102400	27712	130110	18041	93750
lcet10.txt	84	426754	226487	653239	103390	348115
news	98	377109	196337	573444	82162	298053
paper1	95	53161	29040	82199	12742	43601
paper2	91	82199	43213	125410	21391	72303
paper3	84	46526	23922	70446	12634	43191
paper4	80	13286	6877	20161	3675	12669
pi.txt	10	1000000	404236	1404234	382077	1359455
plrabn12.txt	81	481861	237075	718934	138560	468812
progc	92	39611	21174	60783	8649	30536
progl	87	71646	46507	118151	12234	39772
progp	89	49379	33068	82445	7765	26302
sum	255	38240	18452	56690	6187	26522
trans	99	93695	66610	160303	12110	40369
world192.txt	90	1572864	843443	2416305	234588	808562

Tabulka B.3: Vlastnosti datových struktur na reálných vstupních datech. $|\Sigma|$ - velikost abecedy, `n` - délka souboru dat v bytech, `SuffixTree states/trans` - počet vrcholů/hran datové struktury SuffixTree na vstupních datech, `CDAWG states/trans` - počet vrcholů/hran datové struktury CDAWG na vstupních datech.

Zdroj	$\|\Sigma\|$	n (B)	Suffix Tree states	Suffix Tree trans	CDAWG states	CDAWG trans
random	3	5000000	3690498	8690495	3092242	7466299
random	4	5000000	3109955	8109952	2734279	7340190
random	5	5000000	2787595	7787592	2514887	7229041
random	6	5000000	2566227	7566224	2361138	7146543
random	7	5000000	2337425	7337422	2174128	7005691
random	8	5000000	2300462	7300459	2160528	7014903
random	9	5000000	2106553	7106550	1982472	6855345
random	10	5000000	2007586	7007583	1916890	6823972
random	15	5000000	1651032	6651029	1599868	6548130
random	20	5000000	1793321	6793318	1744181	6694085
random	25	5000000	1343921	6343918	1312555	6281011
random	30	5000000	1276938	6276935	1261967	6246872
random	35	5000000	1528066	6528063	1513289	6498226
random	40	5000000	1655087	6655084	1634880	6614403
random	45	5000000	1551457	6551454	1530110	6508573
random	50	5000000	1344238	6344235	1325825	6307326
random	60	5000000	974047	5974044	963058	5952040
random	70	5000000	803215	5803212	797135	5791046
random	80	5000000	799329	5799326	795965	5792595
random	90	5000000	911889	5911886	909766	5907637
random	100	5000000	1089961	6089958	1087856	6085733
random	120	5000000	1425031	6425028	1421168	6417274
random	140	5000000	1539048	6539045	1533508	6527942
random	160	5000000	1451077	6451074	1445419	6439748
random	180	5000000	1275769	6275766	1270662	6265541
random	200	5000000	1085761	6085758	1081483	6077197
random	220	5000000	914255	5914252	910952	5907639
random	240	5000000	772151	5772148	769461	5766768

Tabulka B.4: Vlastnosti datových struktur na generovaných vstupních datech. $|\Sigma|$ - velikost abecedy, `n` - délka souboru dat v bytech, `SuffixTree states/trans` - počet vrcholů/hran datové struktury SuffixTree na vstupních datech, `CDAWG states/trans` - počet vrcholů/hran datové struktury CDAWG na vstupních datech.

Zdroj	$\|\Sigma\|$	n (B)	Suffix Tree	CDAWG	Suffix Array	
					Childtab	Full
alice29.txt	74	152089	28	17	9	17
asyoulik.txt	68	125179	28	18	9	17
bib	81	111261	28	12	9	17
bible.txt	63	2097152	29	13	9	17
book1	82	768771	28	19	9	17
book2	96	610856	28	15	9	17
cp.html	86	24603	28	13	9	17
data.tar	256	2498560	32	6	9	17
data.tar.bz2	256	967840	18	18	9	17
data.tar.gz	256	1270025	20	16	9	17
data.zip	256	1387111	29	9	9	17
E.coli	4	4638690	31	29	9	17
fields.c	90	11150	30	11	9	17
geo	256	102400	22	16	9	17
lcet10.txt	84	426754	28	15	9	17
news	98	377109	28	15	9	17
paper1	95	53161	29	15	9	17
paper2	91	82199	28	17	9	17
paper3	84	46526	28	18	9	17
paper4	80	13286	28	18	9	17
pi.txt	10	1000000	25	26	9	17
plrabn12.txt	81	481861	27	19	9	17
progc	92	39611	28	14	9	17
progl	87	71646	31	10	9	17
progp	89	49379	32	10	9	17
sum	255	38240	27	13	9	17
trans	99	93695	33	8	9	17
world192.txt	90	1572864	28	10	9	17

Tabulka B.5: Velikost paměti spotřebované datovými strukturami na reálných vstupních datech v bytech na jeden znak vstupu. `Suffix Tree` - sufixový strom, `CDAWG` - CDAWG, `Suffix Array - Childtab` - sufixové pole rozšířené o pole lcptab a childtab, `Suffix Array - Full` - sufixové pole rozšířené o pole lcptab, childtab a suflinktab

Zdroj	$\|\Sigma\|$	n (B)	Suffix Tree	CDAWG	Suffix Array	
					Childtab	Full
random	3	5000000	33	31	9	17
random	4	5000000	30	30	9	17
random	5	5000000	29	29	9	17
random	6	5000000	28	28	9	17
random	7	5000000	27	27	9	17
random	8	5000000	27	27	9	17
random	9	5000000	26	26	9	17
random	10	5000000	25	26	9	17
random	15	5000000	23	24	9	17
random	20	5000000	24	25	9	17
random	25	5000000	22	23	9	17
random	30	5000000	22	23	9	17
random	35	5000000	23	24	9	17
random	40	5000000	23	25	9	17
random	45	5000000	23	24	9	17
random	50	5000000	22	23	9	17
random	60	5000000	20	21	9	17
random	70	5000000	19	20	9	17
random	80	5000000	19	20	9	17
random	90	5000000	20	21	9	17
random	100	5000000	21	22	9	17
random	120	5000000	22	23	9	17
random	140	5000000	23	24	9	17
random	160	5000000	22	24	9	17
random	180	5000000	22	23	9	17
random	200	5000000	21	22	9	17
random	220	5000000	20	21	9	17
random	240	5000000	19	20	9	17

Tabulka B.6: Velikost paměti spotřebované datovými strukturami na náhodně generovaných vstupních datech v bytech na jeden znak vstupu. `Suffix Tree` - sufixový strom, `CDAWG` - CDAWG, `Suffix Array - Childtab` - sufixové pole rozšířené o pole lcptab a childtab, `Suffix Array - Full` - sufixové pole rozšířené o pole lcptab, childtab a suflinktab

Zdroj	$\|\Sigma\|$	n (B)	ST	SA_CT	SA_Simple	CDAWG
alice29.txt	74	152089	2134	3241	**1748**	1917
asyoulik.txt	68	125179	1863	3020	**1730**	**1730**
bib	81	111261	1856	3010	1732	**1693**
bible.txt	63	2097152	6822	6756	**3041**	6210
book1	82	768771	4874	5059	**2015**	4683
book2	96	610856	4537	4988	**1930**	4222
cp.html	86	24603	1540	2901	1642	**1480**
data.tar	256	2498560	14844	1164861	**3167**	13589
data.tar.bz2	256	967840	8745	14501	**2020**	8563
data.tar.gz	256	1270025	9226	26386	**2245**	8846
data.zip	256	1387111	9309	273499	**2385**	8848
E.coli	4	4638690	3990	4613	**3562**	3800
fields.c	90	11150	1348	2481	1564	**1337**
geo	256	102400	3651	4743	**1666**	3524
lcet10.txt	84	426754	3513	4101	**1860**	3113
news	98	377109	4000	4698	**1841**	3570
paper1	95	53161	1678	3207	1685	**1621**
paper2	91	82199	1811	3384	**1701**	1711
paper3	84	46526	1617	3127	1673	**1571**
paper4	80	13286	1339	2754	1569	**1326**
pi.txt	10	1000000	2873	3203	**2125**	2818
plrabn12.txt	81	481861	3544	3958	**1867**	3312
progc	92	39611	1640	2934	1658	**1566**
progl	87	71646	1859	3065	1728	**1725**
progp	89	49379	1834	2963	1734	**1730**
sum	255	38240	3530	3348	**1715**	3476
trans	99	93695	1873	3341	1771	**1689**
world192.txt	90	1572864	6601	6720	**2680**	5805

Tabulka B.7: Naměřené časy vyhledávání 10^6 vzorků délky 10^3 znaků v milisekundách. `ST` - sufixový strom, `SA_CT` - sufixové pole rozšířené a pole childtab, `SA_Simple` - algoritmus SA_Simple využívající sufixového pole, `CDAWG` - CDAWG.

Zdroj	$\|\Sigma\|$	n (B)	ST	SA	SA_Simple
alice29.txt	74	152089	**2224**	3316	4260
asyoulik.txt	68	125179	**1935**	3080	4240
bib	81	111261	**1936**	3090	4228
bible.txt	63	2097152	6909	6871	**5595**
book1	82	768771	5045	5323	**4507**
book2	96	610856	4647	5076	**4444**
cp.html	86	24603	**1612**	2973	4128
data.tar	256	2498560	16437	1160450	**6551**
data.tar.bz2	256	967840	8815	14684	**4567**
data.tar.gz	256	1270025	9404	26668	**4842**
data.zip	256	1387111	9732	272719	**5061**
E.coli	4	4638690	**4044**	4674	6111
fields.c	90	11150	**1415**	2544	4065
geo	256	102400	**3778**	4902	4179
lcet10.txt	84	426754	**3609**	4199	4366
news	98	377109	**4107**	4811	4340
paper1	95	53161	**1760**	3270	4188
paper2	91	82199	**1897**	3470	4213
paper3	84	46526	**1692**	3184	4170
paper4	80	13286	**1400**	2807	4075
pi.txt	10	1000000	**2919**	3268	4644
plrabn12.txt	81	481861	**3625**	4323	4384
progc	92	39611	**1717**	3001	4161
progl	87	71646	**1952**	3137	4214
progp	89	49379	**1920**	3068	4194
sum	255	38240	3628	**3461**	4163
trans	99	93695	**1959**	3472	4220
world192.txt	90	1572864	6681	6853	**5196**

Tabulka B.8: Naměřené časy vyhledávání všech výskytů 10^6 vzorků délky 10^3 znaků v milisekundách. `ST` - sufixový strom, `SA_CT` - sufixové pole rozšířené a pole childtab, `SA_Simple` - algoritmus SA_Simple využívající sufixového pole.

Zdroj	$\|\Sigma\|$	n (B)	ST	SA
alice29.txt	74	152089	36	**24**
asyoulik.txt	68	125179	27	**20**
bib	81	111261	21	**17**
bible.txt	63	2097152	749	**460**
book1	82	768771	265	**157**
book2	96	610856	196	**117**
cp.html	86	24603	4	**3**
data.tar	256	2498560	**914**	9926
data.tar.bz2	256	967840	221	**93**
data.tar.gz	256	1270025	293	**171**
data.zip	256	1387111	**447**	1694
E.coli	4	4638690	2176	**1637**
fields.c	90	11150	**2**	**2**
geo	256	102400	14	**10**
lcet10.txt	84	426754	125	**80**
news	98	377109	106	**67**
paper1	95	53161	**9**	**9**
paper2	91	82199	15	**13**
paper3	84	46526	**7**	**7**
paper4	80	13286	**2**	3
pi.txt	10	1000000	362	**253**
plrabn12.txt	81	481861	149	**94**
progc	92	39611	7	**6**
progl	87	71646	14	**13**
progp	89	49379	**9**	**9**
sum	255	38240	6	**5**
trans	99	93695	21	**19**
world192.txt	90	1572864	518	**299**

Tabulka B.9: Naměřené časy vyhledávání supermaximálních opakování v milisekundách. `ST` - sufixový strom, `SA` - sufixové pole.

Zdroj	$\|\Sigma\|$	n (B)	3 slova		5 slov		10 slov	
			ST	SA	ST	SA	ST	SA
alice29.txt	73	152089	31	4	31	3	40	3
asyoulik.txt	68	125179	30	3	31	3	35	3
bib	81	111269	25	3	18	3	28	4
bible.txt	63	2097159	412	50	429	54	465	63
book1	82	768779	151	16	149	18	165	19
book2	96	610859	120	14	125	14	139	15
cp.html	86	24609	3	5	7	3	8	4
data.tar	256	2498569	489	8178	506	8372	555	8292
data.tar.bz2	256	967849	125	9	125	18	131	37
data.tar.gz	256	1270029	175	41	190	54	202	81
data.zip	256	1387119	238	1194	274	1151	300	1202
E.coli	4	4638699	973	111	1018	120	1129	141
fields.c	90	11159	4	0	5	0	7	0
geo	256	102409	15	4	19	3	19	2
lcet10.txt	84	426759	85	10	87	12	98	12
news	98	377109	68	9	68	10	82	12
paper1	95	53169	12	7	13	6	19	4
paper2	91	82199	19	7	22	8	26	3
paper3	84	46529	12	5	13	7	12	8
paper4	80	13289	8	0	6	0	6	0
pi.txt	10	1000009	169	21	186	38	190	68
plrabn12.txt	81	481869	94	11	98	11	109	11
progc	92	39619	11	3	9	6	14	5
progl	87	71649	19	4	21	10	25	7
progp	89	49379	15	3	18	5	18	8
sum	255	38249	11	3	8	8	12	9
trans	99	93699	30	5	25	5	36	6
world192.txt	90	1572869	303	35	299	35	339	38

Tabulka B.10: Naměřené časy hledání nejdelší společné podposloupnosti tří, pěti a deseti slov v milisekundách. `ST` - sufixový strom, `SA` - sufixové pole.

Zdroj	$\|\Sigma\|$	n (B)	ST	SA
alice29.txt	74	152089	13	**0**
asyoulik.txt	68	125179	11	**1**
bib	81	111261	9	**0**
bible.txt	63	2097152	388	**1**
book1	82	768771	131	**0**
book2	96	610856	97	**1**
cp.html	86	24603	2	**0**
data.tar	256	2498560	**436**	9446
data.tar.bz2	256	967840	109	**26**
data.tar.gz	256	1270025	159	**49**
data.zip	256	1387111	**281**	2380
E.coli	4	4638690	1000	**13**
fields.c	90	11150	**0**	**0**
geo	256	102400	6	**1**
lcet10.txt	84	426754	60	**0**
news	98	377109	52	**1**
paper1	95	53161	3	**1**
paper2	91	82199	6	**0**
paper3	84	46526	2	**0**
paper4	80	13286	1	**0**
pi.txt	10	1000000	163	**29**
plrabn12.txt	81	481861	72	**0**
progc	92	39611	2	**0**
progl	87	71646	5	**0**
progp	89	49379	3	**0**
sum	255	38240	2	**0**
trans	99	93695	8	**1**
world192.txt	90	1572864	266	**1**

Tabulka B.11: Naměřené časy vyhledávání nejkratších unikátních podslov v milisekundách. `ST` - sufixový strom, `SA` - sufixové pole.

Zdroj	$\|\Sigma\|$	n (B)	Suffix Tree	SA Greedy	SA Optim
alice29.txt	74	152089	**129**	7823	198
asyoulik.txt	68	125179	**133**	7698	211
bib	81	111261	**97**	7594	174
bible.txt	63	2097152	**346**	9616	485
book1	82	768771	**485**	9329	524
book2	96	610856	**320**	9197	384
cp.html	86	24603	**84**	7543	207
data.tar	256	2498560	**1677**	497226	342551
data.tar.bz2	256	967840	**4903**	18090	7587
data.tar.gz	256	1270025	**5046**	28346	15884
data.zip	256	1387111	**2984**	162346	158987
E.coli	4	4638690	**579**	8231	597
fields.c	90	11150	**48**	7088	145
geo	256	102400	**594**	9615	973
lcet10.txt	84	426754	**253**	8522	303
news	98	377109	**292**	9107	324
paper1	95	53161	**77**	7843	187
paper2	91	82199	**99**	7976	193
paper3	84	46526	**101**	7756	198
paper4	80	13286	**92**	7409	200
pi.txt	10	1000000	817	7809	**750**
plrabn12.txt	81	481861	**385**	8912	398
progc	92	39611	**82**	7517	183
progl	87	71646	**70**	7472	138
progp	89	49379	**55**	7297	132
sum	255	38240	**202**	7961	445
trans	99	93695	**51**	7591	130
world192.txt	90	1572864	**312**	9912	514

Tabulka B.12: Naměřené časy výpočtu matching statistics 100 náhodně generovaných slov délky 10^4 znaků vzhledem k datům z uvedeného zdroje. `ST` - sufixový strom, `SA Greedy` - algoritmus greedymatch využívající sufixové pole, `SA Optim` - vylepšený algoritmus greedymatch