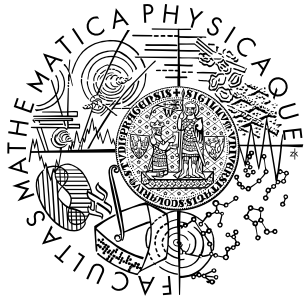
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Pavel Žoha

Algoritmy konstrukce sufixového pole

Kabinet software a výuky informatiky

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

2006

Děkuji svému vedoucímu RNDr. Tomáši Dvořákovi, CSc. a Mgr. Martinu Senftovi za ochotu a za čas, který mi věnovali.

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

V Praze dne 12. prosince 2006 Pavel Žoha

Obsah

Název práce: Algoritmy konstrukce sufixového pole
Autor: Pavel Žoha
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. Tomáš Dvořák, CSc.
e-mail vedoucího: dvorak@ksvi.mff.cuni.cz

Abstrakt: Sufixové pole je datová struktura, která se používá při operacích s řetězci jako je vyhledávání vzorků. Má uplatnění také v některých algoritmech na bezztrátovou kompresi dat. V této práci uvádím srovnání různých postupů při konstrukci sufixového pole (algoritmy Manbera a Myerse, Kärkkäinena a Sanderse, Sewarda, Manziniho a Ferraginy a Ukkonenův algoritmus na konstrukci sufixového stromu). Tyto metody jsem implementoval a spolu s běžnými třídícími algoritmy (mergesort, quicksort, heapsort a shellsort) testoval jednak na souborech běžně používaných formátů a jednak na náhodně generovaných datech nad různě velkými abecedami. Dále jsem se zabýval možností použití algoritmů pro vstupy nad abecedami většími než 256 znaků.

Klíčová slova: porovnání algoritmů, sufixové pole, sufixový strom, třídění řetězců

Title: Suffix array construction algorithms
Author: Pavel Žoha
Department: Department of Software and Informatics
Supervisor: RNDr. Tomáš Dvořák, CSc.
Supervisor's e-mail address: dvorak@ksvi.mff.cuni.cz

Abstract: Suffix array is a data structure used for string operations such as pattern matching. It is also useful for some lossless data compression algorithms. In this paper I present a comparison of several different ways to construct it (algorithms of Manber & Myers, Kärkkäinen & Sanders, Seward, Manzini & Ferragina and Ukkonen's algorithm for suffix tree construction). I implemented these methodes and together with common sorting algorithms (mergesort, quicksort, heapsort and shellsort) tested both on standard data files and on randomly generated files of different alphabet sizes. I also studied the possibility of using these algorithms for inputs of alphabet greater than 256 characters.

Keywords: algorithms comparing, suffix array, suffix tree, string sorting

Kapitola 1

Úvod

Datová struktura sufixové pole byla navržena v roce 1990 Manberem a Myersem v [16] jako prostorově úspornější alternativa k sufixovému stromu. Sufixové pole má v praxi široké uplatnění. Lze ho například využít k efektivnímu vyhledávání vzorků v řetězci. Používá se také při Burrows-Wheelerově transformaci, která je součástí kompresního algoritmu bzip2 (viz. [14]). Ukazuje se, že také libovolná úloha řešitelná za pomoci sufixového stromu lze vyřešit s použitím sufixového pole rozšířeného o dodatečné informace (viz. [10]).

Od roku 1990 do současnosti bylo publikováno několik algoritmů na konstrukci sufixového pole. Tyto algoritmy se výrazně liší svými vlastnostmi. Existují algoritmy pracující v lineárním čase, ale také algoritmy v nejhorším případě superlineární, které však v praxi pracují rychleji.

Pokusil jsem se ze známých algoritmů vybrat několik takových, které se co možná nejvíce liší myšlenkou, na které jsou založeny. Tyto algoritmy jsem studoval do hloubky i s důrazem na jejich implementační detaily. Dále jsem se zaměřoval na hledání vstupních dat, na kterých jednotlivé algoritmy selhávají.

Jednotlivé metody konstrukce jsem porovnával z teoretického hlediska podle předpokládaného chování na různých typech dat i experimentálně při skutečném výpočtu na počítači. Algoritmy jsem implementoval v jednotném prostředí výhradně podle slovního popisu jejich autorů. Při výběru dat pro experimentální vyhodnocení algoritmů jsem vycházel z obvyklého přístupu, který používá standardní textové korpusy, viz. např. [1].

V úvodní části práce uvádím definice základních pojmů a značení používané dále v textu a popisuji úlohu *konstrukce sufixového pole* spolu s jejím základním řešením.

V druhé části analyzuji různé algoritmy, které řeší tuto úlohu. Analýza se skládá z popisu myšlenky algoritmu a z jeho kostry popsané v pseudokódu, následují detaily mé implementace a na závěr uvádím jeho vlastnosti spolu s popisem typu dat, pro který je vhodný a nevhodný. Na přiloženém CD je každý algoritmus implementován v jazyce C.

Ke konstrukčním algoritmům je přidán popis efektivního algoritmu na kontrolu správnosti spočítaného sufixového pole.

Na konci teoretické části se zabývám úpravou algoritmů pro práci se vstupními soubory nad 16ti bitovou abecedou.

Na závěr práce uvádím výsledky experimentů. Měřil jsem rychlosti a množství použité paměti (mé implementace) studovaných algoritmů spolu s běžnými třídícími algoritmy na různých typech dat.

Kapitola 2

Základní pojmy a značení

Nejprve uvádím definice základních pojmů a značení, které budu v práci používat. Toto značení je shodné se značením ve většině studovaných pramenů.

2.1 Abeceda, řetězce

Základním pojmem je řetězec nad abecedou znaků.

Abeceda je neprázdná konečná setříděná množina prvků (znaků). Budu ji značit Σ a její velikost $|\Sigma|$. V tomto textu bude abeceda většinou reprezentována ASCII znaky od 0 do 255. V příkladech budu používat podmnožinu znaků a až z. Znaky budu označovat λ, případně $\lambda_1, \lambda_2, \ldots$

Řetězec je konečná posloupnost znaků abecedy (ϵ označuje prázdný řetězec). V textu se značí x, y, případně $x_0, x_1, \ldots, y_0, y_1, \ldots$. Délka řetězce x se značí $|x|$. Znaky řetězce se indexují hranatými závorkami. První znak má index jedna.

Příklad:
$\Sigma = \{a, k, o\}$
$x = kakao, |x| = 5$
$x[1] = k, x[2] = a, x[3] = k, x[4] = a, x[5] = o$

Sufix řetězce je jeho libovolná (i prázdná) přípona. Řetězec délky n má tedy $(n + 1)$ různých sufixů. Sufixy jednoho řetězce se číslují pozicí jejich začátku v tomto řetězci. Pro $i = 1, \ldots, (n + 1)$ se i-tý sufix řetězce x délky n skládá ze znaků $x[i], x[i + 1], \ldots, x[n]$ a značí se $x[i..n]$.

V textu se budou řetězce porovnávat podle ***lexikografického uspořádání***. Řetězec x je lexikograficky menší nebo roven řetězci y (značí se $x \leq y$), pokud $|x| = 0$ nebo pokud $|y| > 0$ a první znak x je menší než první znak y nebo pokud se první znaky obou řetězců rovnají a $x[2..|x|] \leq y[2..|y|]$.

Dalším pojmem je ***nejdelší společný prefix*** množiny řetězců, budu ho značit $lcp\{x_1, x_2, \ldots, x_k\}$. Je to celé číslo větší nebo rovné nule, které udává délku nejdelší společné

předpony těchto řetězců.

Jistou charakteristikou řetězce je jeho ***délka nejdelší shody***, budu ji značit $lml(x)$. Jedná se o celé číslo větší nebo rovné nule, které je maximem z lcp pro všechny dvojice sufixů x. Tedy $lml(x) = max_{i=1..n,j=1..n}\{lcp\{x[i..n], x[j..n]\}\}$.

2.2 Sufixové pole

Sufixové pole pro řetězec x délky n je pole přirozených čísel délky n, značí se SA_x nebo jen SA. Je dáno předpisem $SA_x[j] = i \iff x[i..n]$ je j-tý řetězec v lexikografickém uspořádání všech neprázdných sufixů řetězce x.

Jedná se tedy o permutaci čísel 1 až n. Průchodem sufixového pole zleva doprava získáváme sufixy původního řetězce v rostoucím lexikografickém uspořádání.

Sufixové pole budu pro přehlednost psát do tabulky, kde první řádek obsahuje indexy 1 až n, ve druhém řádku je řetězec x a ve třetím řádku je SA_x. Sufixové pole pro výše zmíněný příklad vypadá takto:

		1	2	3	4	5
x	=	k	a	k	a	o
SA	=	2	4	1	3	5

V některých ze studovaných algoritmů se používá ***inverzní sufixové pole***. Je to také pole přirozených čísel délky n, značíme ho ISA_x nebo ISA a je dáno předpisem $ISA[i] = j \iff SA[j] = i$.

Hodnota $ISA[i]$ tedy udává lexikografické pořadí i-tého sufixu. Pro zmíněný příklad vypadá ISA_x takto:

		1	2	3	4	5
x	=	k	a	k	a	o
ISA	=	3	1	4	2	5

Některé algoritmy také pracují s částečným uspořádáním sufixů ve smyslu, že porovnávají pouze prvních h znaků (***h-uspořádání***). V této souvislosti se v textu používá pojem ***h-skupina***, je to třída ekvivalence tohoto h-uspořádání.

Dalšími pojmy jsou ***h-sufixové pole*** SA_h a ***inverzní h-sufixové pole*** ISA_h. SA_h je pole, které vznikne setříděním sufixů podle prvních h znaků. Indexy sufixů, které spadají do jedné h-skupiny budu zapisovat v rostoucím pořadí, více než jednoprvkové h-skupiny budu v obrázcích uzavírat do kulatých a hranatých závorek.

ISA_h obsahuje pořadí sufixů v h-uspořádání. Pokud více sufixů spadá do stejné h-skupiny, budou všechny tyto sufixy mít v poli ISA_h pořadí (lexikograficky) nejmenšího z nich. Jako na příkladu:

		1	2	3	4	5
x	=	k	a	k	a	o
SA_1	=	(2	4)	(1	3)	5
ISA_1	=	3	1	3	1	5

Posledním pojmem je ***průměrný společný prefix*** řetězce x. Je to reálné číslo větší nebo rovné nule, které se získá jako aritmetický průměr lcp všech $(n-1)$ dvojic (neprázdných) sufixů, které v sufixovém poli leží vedle sebe. Průměrný společný prefix budu značit $\overline{lcp}(x)$.

Kapitola 3

Úloha konstrukce sufixového pole

Studované algoritmy na konstrukci sufixového pole jsou algoritmy, které řeší následující úlohu:

Máme přirozené číslo n a řetězec x délky n nad abecedou Σ. Úkolem je zkonstruovat pole celých čísel SA délky n, které je sufixovým polem pro vstupní řetězec x.

V implementaci jsem se omezil na případ, kdy $2 \leq n < 2^{31}$ a $\Sigma = \{0, 1, \ldots, \alpha\}$, kde $\alpha \in \{2^8 - 1, 2^{16} - 1\}$.

3.1 Jednoduché řešení

Nejjednodušším řešením takové úlohy je považovat neprázdné sufixy x za samostatné řetězce. Máme pak n prvkové pole řetězců, které můžeme setřídit nějakým standardním třídícím algoritmem, například algoritmem quicksort.

V první fázi se pole SA naplní hodnotami od jedné do n. V druhém kroku necháme toto pole setřídit knihovní funkci qsort. Jejím parametrem je kromě pole a jeho délky také ukazatel na funkci, která porovnává dva prvky pole. V tomto případě prvky pole SA chápu jako indexy do řetězce x na pozici, kde začíná příslušný sufix.

Zjednodušeně se dá celý algoritmus popsat takto:

procedure computeSA_quicksort()
    for $i \leftarrow 1$ **to** n **do** $SA[i] \leftarrow i$;
    qsort(SA, n, *cmp*);

procedure cmp(i, j)
    return $x[i..n] < x[j..n]$

Pro srovnání jsem tento postup implementoval se čtyřmi běžnými třídícími algoritmy a v experimentech ho porovnal s ostatními metodami.

Tento postup obecně není příliš efektivní, jeho časová složitost v nejhorším případě je pro algoritmy heapsort a mergesort $O(n^2 \log n)$ a pro quicksort a shellsort dokonce $O(n^3)$[1]. Očekávaná časová složitost tohoto postupu je $O(A.n \log n)$, kde A je počet znaků, které se v průměru musí porovnat pro zjištění který ze dvou sufixů x je lexikograficky větší.

Pro dosažení lepších výsledků je nutné využít závislostí, které mezi tříděnými řetězci existují.

[1]Pro algoritmus shellsort jsou známy lepší odhady časové složitosti v závislosti na velikosti skoku

Kapitola 4

Analýza algoritmů

4.1 Manberův a Myersův algoritmus

4.1.1 Popis algoritmu

Manber a Myers v [16] popsali výhody a nevýhody sufixového pole a navrhli algoritmus na jeho konstrukci. Jejich algoritmus není obecně rychlejší než postup, kdy se nejprve zkonstruuje sufixový strom (viz. kapitola 4.5) a z něho pak průchodem do hloubky sufixové pole. Výhodou jejich postupu je však přibližně trojnásobná úspora paměti.

Manberův a Myersův algoritmus je iterativní, postupně zpřesňuje částečné sufixové pole. V prvním kroku se spočítá SA_1. V každém dalším se z SA_h spočítá SA_{2h}. Tento postup se opakuje, dokud nejsou všechny h-skupiny jednoprvkové. Celkový počet iterací nepřevýší $\lceil \log_2(n+1) \rceil$.

SA_1 se získá jako výsledek třídění sufixů podle prvního znaku algoritmem bucketsort. V h-té iteraci jsou sufixy roztříděny podle prvních h znaků do h-skupin. Je tedy potřeba každou h-skupinu setřídit podle prvních $2h$ znaků a rozdělit ji na $2h$-skupiny. Pokud sufixy i a j patří do stejné h-skupiny, mají prvních h znaků stejných a stačí porovnat následujících h symbolů. Tyto následující znaky sufixů i a j jsou právě prvními znaky sufixů $(i+h)$ a $(j+h)$. Tedy $2h$-pořadí sufixů i a j je stejné jako h-pořadí sufixů $(i+h)$ a $(j+h)$.

Algoritmus v každé iteraci prochází SA_h zleva doprava, když narazí na sufix i, umístí sufix $(i-h)$ (pokud existuje) na začátek jeho h-skupiny a posune tento začátek o 1 doprava. Tímto způsobem jsou do SA_{2h} umístěny všechny sufixy delší než h znaků, přípony délky h a kratší je potřeba umístit zvlášť.

Celý algoritmu je popsán na obrázku 4.1.

4.1.2 Implementační detaily

Kromě vstupního řetězce x o délce n a pole SA délky n, potřebuje moje implementace pomocné pole 32-bitových celých čísel délky n. Stejné paměťové nároky uvádějí i autoři v [16]. Toto pomocné pole jsem označil ISA. Spodních 31 bitů polí SA a ISA slouží pro uložení SA_h a ISA_h, poslední bit v obou polích používám pro označování políček.

```
procedure computeSA_mm()
    spočítej SA_1
    h ← 1
    while existuje víceprvková h-skupina do
        for j ← max(n − h + 1,1) to n do
            SA_2h[head_h(j)] = j
            inc(head_h(j))
        for i ← 1 to n do
            j ← (SA_h[i] − h)
            if j ≥ 1 then
                SA_2h[head_h(j)] = j
                inc(head_h(j))
        h ← 2h
```

Obrázek 4.1: Manberův a Myersův algoritmus. Operátor $head_h(i)$ ukazuje na začátek h-skupiny v SA_h, do které patří sufix i.

Na začátku h-té iterace obsahuje pole SA přesně SA_h a pole ISA obsahuje ISA_h. Na obrázku 4.2 je příklad pro $x =$ 'mississippi' a $h = 1$. Sufixy, které jsou na prvním místě ve své h-skupině, jsou v ISA označeny. V příkladu jsou to sufixy 2, 1, 9 a 3 a označena jsou políčka $ISA[2]$, $ISA[1]$, $ISA[9]$ a $ISA[3]$.

	1	2	3	4	5	6	7	8	9	10	11
x =	m	i	s	s	i	s	s	i	p	p	i
SA =	(2	5	8	11)	1	(9	10)	(3	4	6	7)
ISA =	5	1	8	8	1	8	8	1	6	6	1

Obrázek 4.2: Manberův a Myersův algoritmus na začátku první iterace pro vstupní slovo $x =$ "mississippi".

Pomocí označených políček v ISA definuji operátor $head_h(i)$ takto:

```
procedure head_h(i)
    if ISA[i] je označeno then return ISA[i]
    else return ISA[SA[ISA[i]]]
```

Nejprve budu umisťovat sufixy délky h a kratší na začátky svých h-skupin, potom budu procházet pole $SA[i]$, pro $i = 1, .., n$ a umisťovat sufixy $(j = SA[i] - h)$. Tyto sufixy nemohu zapisovat přímo do pole SA, protože v něm je v tuto chvíli SA_h. Místo toho použiji neoznačená políčka v ISA. Pokud $ISA[j]$ není označené, uložím do něj nové pořadí sufixu j. Označená políčka se nesmí přepsat, jsou potřeba pro operátor *head*. Pokud $ISA[j]$ je označené, označím v SA novou pozici pro sufix j:

for $j \leftarrow \max(n-h+1, 1)$ **to** n **do**
  if $ISA[j]$ není označeno **then** $ISA[j] = head(j)$
  else označ $SA[head(j)]$
  inc($head(j)$)
for $i \leftarrow 1$ **to** n **do**
  $j \leftarrow SA[i] - h$
  if $ISA[j]$ není označeno **then** $ISA[j] = head(j)$
  else označ $SA[head(j)]$
  inc($head(j)$)

Obrázek 4.3 ukazuje, jak se změní uvedený příklad po tomto průchodu. V poli SA je v každé h-skupině označené jedno políčko.

	1	2	3	4	5	6	7	8	9	10	11
x =	m	i	s	s	i	s	s	i	p	p	i
SA =	(2	5	8	11)	1	(9	10)	(3	4	6	7)
ISA =	6	5	12	8	4	11	9	2	8	6	1

Obrázek 4.3: Manberův a Myersův algoritmus po prvním průchodu.

Teď se sestaví SA_{2h}. Nejdřív se na označená políčka v SA dají první prvky z příslušné h skupiny a pak se průchodem přes ISA umístí zbytek (výsledek je na obrázku 4.4):

for $i \leftarrow 1$ **to** n **do**
  $j \leftarrow SA[i]$
  if $ISA[j]$ je označeno **then** $k \leftarrow j$
  if $SA[i]$ je označeno **then** $SA[i] \leftarrow k$
for $j \leftarrow 1$ **to** n **do**
  if $ISA[j]$ není označeno **then** $SA[ISA[j]] \leftarrow j$

	1	2	3	4	5	6	7	8	9	10	11
x =	m	i	s	s	i	s	s	i	p	p	i
SA =	(11	8	[2	5])	1	(10	9)	([4	7]	[3	6])
ISA =	6	5	12	8	4	11	9	2	8	6	1

Obrázek 4.4: Manberův a Myersův algoritmus po spočítání SA_2. Kulaté závorky oddělují 1-skupiny, hranaté 2-skupiny.

Už zbývá jenom spočítat ISA_{2h}. To udělám ve dvou krocích. Nejprve zrekonstruuji ISA_h (obrázek 4.5) a z něj pak spočítám ISA_{2h} (obrázek 4.6).

Nejprve tedy projdu pole SA a do $ISA[SA[i]]$ zapíši pozici začátku příslušné h-skupiny. Využívám toho, že označená políčka v ISA ukazují na začátek následující h-skupiny:

$k \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
    if $i \geq k$ **then** $l \leftarrow k$
    if $ISA[SA[i]]$ je označeno **then** $k \leftarrow ISA[SA[i]]$
    $ISA[SA[i]] \leftarrow l$

	1	2	3	4	5	6	7	8	9	10	11
x =	m	i	s	s	i	s	s	i	p	p	i
SA =	(11	8	[2	5])	1	(10	9)	([4	7]	[3	6])
ISA =	5	1	8	8	1	8	8	1	6	6	1

Obrázek 4.5: Manberův a Myersův algoritmus po zrekonstruování ISA_1.

Pole SA teď projdu ještě jednou. Pro každé $j = SA[i]$ porovnám dvojici $(ISA[j], ISA[j+h])$ s dvojicí $(ISA[j-1], ISA[j-1+h])$ a pokud se nebudou rovnat, znamená to, že sufix j začíná novou $2h$-skupinu a políčko $ISA[j]$ označím. Nakonec projdu SA ještě jednou a podle označených začátků $2h$-skupin spočítám ISA_{2h}:

for $i \leftarrow 1$ **to** n **do**
    $j \leftarrow SA[i]$
    if $(ISA[j], ISA[j+h]) \neq (ISA[j-1], ISA[j-1+h])$ **then** označ $ISA[j]$
for $i \leftarrow 1$ **to** n **do**
    $j \leftarrow SA[i]$
    if $ISA[j]$ je označeno **then** $k \leftarrow i$
    $ISA[j] \leftarrow k$

Obrázek 4.6 ukazuje příklad na konci první iterace algoritmu. Pokud nejsou všechny $2h$-skupiny jednoprvkové, zvětší se h na dvojnásobek a pokračuje se od začátku.

x =	m	i	s	s	i	s	s	i	p	p	i
SA =	(11	8	[2	5])	1	(10	9)	([4	7]	[3	6])
ISA =	5	3	10	8	3	10	8	2	7	6	1

Obrázek 4.6: Manberův a Myersův algoritmus na konci první iterace. Pole SA obsahuje SA_2 a v ISA je ISA_2. První prvky 2-skupin jsou v ISA označené.

4.1.3 Vlastnosti

Paměťové nároky Manberova a Myersova algoritmu závisí pouze na délce vstupního slova, činí $9n$ bytů (n bytů na vstupní řetězec x, $4n$ bytů na výstupní pole SA a $4n$ bytů na pomocné pole ISA).

Jedna iterace algoritmu má lineární časovou složitost vzhledem k délce vstupního slova. Při každé další iteraci se h zvětší na dvojnásobek a algoritmus skončí nejpozději pro $h \geq n$. Teoretická složitost v nejhorším případě je tedy $O(n \log n)$.

Vzhledem k tomu, že každá iterace trvá přibližně stejně dlouho, skutečná rychlost výpočtu závisí hlavně na počtu těchto iterací. Výpočet probíhá, dokud nejsou všechny h-skupiny jednoprvkové. Doba běhu je tedy přímo úměrná logaritmu délky nejdelší shody ve vstupním řetězci, $\log(lml(x))$.

Manberův a Myersův algoritmus se tak hodí na vstupy, které mají lml malé, naopak kritický je pro tento algoritmus například řetězec DNA s dlouhými sekvencemi znaků, které se opakují. To potvrzují i měření v kapitole 6.5.

4.2 Kärkkäinenův a Sandersův algoritmus

4.2.1 Popis algoritmu

Kärkkäinen a Sanders v [15] navrhli rekurzivní algoritmus pro konstrukci sufixového pole, který má časovou složitost v nejhorším případě $O(n)$. Jeho paměťové nároky jsou menší než pro sufixový strom (viz. kapitola 4.5), ale větší než u Manberova a Myersova algoritmu.

Algoritmus pracuje ve třech krocích. Nejprve rekurzivně setřídí sufixy začínající na pozicích $i \bmod 3 \neq 1$. Potom setřídí zbylé sufixy za použití výsledků prvního kroku. Nakonec slije obě posloupnosti z předchozích kroků do výsledného sufixového pole.

Postup ukáži na příkladu vstupního slova "mississippi" (obrázek 4.7)

	1	2	3	4	5	6	7	8	9	10	11
x =	m	i	s	s	i	s	s	i	p	p	i

Obrázek 4.7: Kärkkäinenův a Sandersův algoritmus. Vstupní slovo $x =$ "mississippi".

V prvním kroku se z indexů sufixů ($i \equiv 2 \bmod 3$) a ($i \equiv 0 \bmod 3$) sestaví posloupnost s a spočítá se 3-pořadí těchto sufixů v rámci množiny s. Toto pořadí se pak uloží do vektoru x' (obrázek 4.8).

Třídění má jednu výjimku. Poslednímu sufixu ve skupině ($i \equiv 2 \bmod 3$) je vždy přiřazeno samostatné, menší pořadí než všem ostatním sufixům, které mají stejné první tři znaky jako on.

$s =$	(2	5	8	11)	(3	6	9)
$t_i =$	iss	iss	ipp	i--	ssi	ssi	ppi
$x' =$	3	3	2	1	5	5	4

Obrázek 4.8: Kärkkäinenův a Sandersův algoritmus. V prvním řádku jsou indexy sufixů ($i \equiv 2 \bmod 3$)($i \equiv 0 \bmod 3$), ve druhém jsou první tři znaky těchto sufixů a v posledním jejich 3-pořadí.

Vektor x' je vstupní slovo pro rekurzi (obrázek 4.9). Po návratu z rekurze obsahuje inverzní sufixové pole $ISA_{x'}$ lexikografické pořadí sufixů $(i \equiv 2 \bmod 3)(i \equiv 0 \bmod 3)$.

$x' =$	3	3	2	1	5	5	4
			↓ rekurze				
$ISA_{x'} =$	4	3	2	1	7	6	5

Obrázek 4.9: Kärkkäinenův a Sandersův algoritmus. Rekurze.

Je tomu tak proto, že každé dva sousední sufixy v poli s jsou od sebe vzdáleny přesně tři znaky a po 3-třídění každé číslo v x' zastupuje právě trojici znaků z x. Výjimku tvoří rozhraní skupin $(i \equiv 2 \bmod 3)$ a $(i \equiv 0 \bmod 3)$, proto bylo poslednímu sufixu v první skupině přiřazeno samostatné pořadí.

Jedním průchodem přes $ISA_{x'}$ se pak spočítá SA_{y1}, lexikograficky setříděný seznam sufixů z s (obrázek 4.10).

$s =$	(2	5	8	11)	(3	6	9)
$ISA_{x'} =$	4	3	2	1	7	6	5
			↓ inverze				
$SA_{y1} =$	11	8	5	2	9	6	3

Obrázek 4.10: Kärkkäinenův a Sandersův algoritmus. Výpočet SA_{y1}.

V druhém kroku algoritmu se setřídí zbylé sufixy do pole SA_{y2}. Průchodem přes $SA_{y1}[j]$ pro $j = 1, \ldots, |SA_{y1}|$ se získá setříděná posloupnost sufixů $(i \equiv 2 \bmod 3)$, ta určuje pořadí sufixů $(i \equiv 1 \bmod 3)$ od druhého znaku. Stabilní třídění podle prvního písmene pak určí SA_{y2} (obrázek 4.11).

$s' =$	(1	4	7	10)
↓ indukce z SA_{y1}				
$s' =$	10	7	4	1
↓ stabilní 1-třídění				
$SA_{y2} =$	1	10	7	4

Obrázek 4.11: Kärkkäinenův a Sandersův algoritmus. Výpočet SA_{y2} z množiny sufixů $s' = (i \equiv 1 \bmod 3)$.

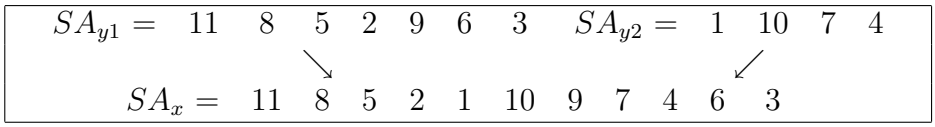
V posledním kroku algoritmu se setříděné posloupnosti SA_{y1} a SA_{y2} slijí do výsledného sufixového pole SA_x (obrázek 4.12). K tomu je potřeba v konstantním čase porovnat sufix i_1 z posloupnosti SA_{y1} se sufixem i_2 z SA_{y2}.

Pořadí sufixů z SA_{y1} je po rekurzi uloženo v poli $ISA_{x'}$, to umožňuje definovat makro rank:

procedure rank(i)
if $(i \equiv 2 \bmod 3)$ **then return** $ISA_{x'}[\lfloor (i+1)/3 \rfloor]$
if $(i \equiv 0 \bmod 3)$ **then return** $ISA_{x'}[\lfloor (i+1)/3 \rfloor + \lfloor i/3 \rfloor]$

Pro sufix i_2 platí, že $i_2 + 1 \equiv 2 \bmod 3$ a $i_2 + 2 \equiv 0 \bmod 3$. Na tomto pozorování je založena porovnávací funkce compare, která vrací true, pokud sufix i_1 je lexikograficky menší než i_2:

procedure compare(i_1, i_2)
if $(i_1 \equiv 2 \bmod 3)$ **then**
return $(x[i_1]$,rank$(i_1 + 1)) < (x[i_2]$,rank$(i_2 + 1))$
if $(i_1 \equiv 0 \bmod 3)$ **then**
return $(x[i_1], x[i_1 + 1]$,rank$(i_1 + 2)) < (x[i_2], x[i_2 + 1]$,rank$(i_2 + 2))$



Obrázek 4.12: Kärkkäinenův a Sandersův algoritmus. Slití posloupností SA_{y1} a SA_{y2}.

Celý algoritmus je na obrázku 4.13.

4.2.2 Implementační detaily

Na rozdíl od Manberova a Myersova algoritmu se v tomto případě moje implementace opírá téměř přesně o slovní popis algoritmu (viz. výše).

Jedinou komplikací je skutečnost, že vstupní řetězec x je nad abecedou 8-bitových znaků, zatímco vstupní řetězce pro rekurzivní volání jsou nad abecedou celých čísel z rozsahu 1 až n. Na zakódování jednoho znaku řetězce v rekurzi používám 32 bitů. Z tohoto důvodu moje implementace obsahuje dvě funkce computeSA_ks, jednu pro osmi a jednu pro třicetidvou bitovou abecedu.

4.2.3 Vlastnosti

Kärkkäinenův a Sandersův algoritmus má nejvyšší paměťové nároky ze všech studovaných algoritmů (mimo sufixového stromu). Paměťově nejnáročnější je závěrečné slévání. Slévá se posloupnost délky $\frac{2}{3} * 4n$ bytů s posloupností délky $\frac{1}{3} * 4n$ do výsledného sufixového pole délky $4n$ bytů. Při slévání je dále potřeba pomocné pole délky $\frac{2}{3} * 4n$ bytů a vstupní řetězec x délky n. Součtem těchto hodnot dostáváme $11,7n$ bytů paměti. Paměťové nároky spojené s rekurzí nejsou příliš velké, při rekurzivním volání je po dobu běhu hlubší iterace nutné v paměti udržovat jen vstupní řetězec. To je v první iteraci n bytů, ve druhé $\frac{2}{3} * 4n$ bytů, ve třetí $(\frac{2}{3})^2 * 4n$, atd. Spotřeba paměti v průběhu rekurze nikdy nepřevýší množství

```
procedure computeSA_ks(x, SA_x)
    s ← (i ≡ 2 mod 3)(i ≡ 0 mod 3)
    proveď 3-třídění sufixů z s a jejich pořadí zapiš do x'
    if x' obsahuje dva stejné znaky then
        computeSA_ks(x', SA_x')
        spočítej ISA_x'
    else ISA_x' ← x'
    for j ← 1 to |ISA_x'| do
        i ← ISA_x'[j]
        SA_y1[i] ← s[j]
    k ← 1
    for j ← 1 to |SA_y1| do
        i ← SA_y1[j]
        if (i ≡ 2 mod 3) then
            SA_y2[k] ← (i − 1)
            inc(k)
    proveď stabilní 1-třídění SA_y2
    slij posloupnosti SA_y1 a SA_y2 za pomoci funkce compare do pole SA_x
```

Obrázek 4.13: Kärkkäinenův a Sandersův algoritmus.

paměti použité v první iteraci. Moje implementace Kärkkäinenova a Sandersova algoritmu tedy potřebuje pro výpočet $11,7n$ bytů paměti.

Časová složitost algoritmu v nejhorším případě je $O(n)$. Rozdělení sufixů na dvě skupiny, 3-třídění a slití obou skupin dohromady má lineární časovou složitost a algoritmus potřebuje jen jedno rekurzivní volání na řetězec délky $\lfloor 2/3n \rfloor$. Časová složitost lze tedy vyjádřit rekurentním vztahem $T(n) = O(n) + T(\frac{2}{3}n)$, který má řešení $T(n) = O(n)$.

Skutečná rychlost algoritmu závisí především na počtu iterací. Čas potřebný na provedení jedné iterace je jen minimálně závislý na charakteru vstupních dat. Rekurze končí v okamžiku, kdy se ve dvou třetinách vstupního slova žádný znak neopakuje. V první iteraci každý symbol odpovídá jednomu znaku abecedy, ve druhé iteraci každý symbol zastupuje trojici znaků ze vstupního souboru, ve třetí iteraci symbol zastupuje devět znaků původního vstupu atd. Z toho vyplývá, že doba výpočtu sufixového pole je u tohoto algoritmu stejně jako u algoritmu Manberova a Myersova přímo úměrná logaritmu maxima z délek společného prefixu dvojic sufixů. V tomto případě to ale není maximum přes všechny dvojice, ale pouze přes dvojice sufixů, které se oba dostanou do nejhlubší iterace rekurze.

Kärkkäinenův a Sandersův postup je tedy opět vhodný zejména pro vstupy, ve kterých se nevyskytuje více stejných dlouhých podřetězců.

4.3 Sewardův algoritmus

4.3.1 Popis algoritmu

Sewardův algoritmus popsaný v [20] se vyznačuje malými paměťovými nároky, kromě prostoru pro vstupní řetězec a výsledné sufixové pole používá jen konstantní množství další paměti. Ačkoli má velkou časovou složitost v nejhorším případě, na většině vstupních dat je v praxi rychlejší než oba předchozí algoritmy.

Algoritmus nejprve provede 2-třídění sufixů vstupního řetězce. Vznikne tak SA_2, kde jsou sufixy rozděleny do 1-skupin a 2-skupin (viz příklad na obrázku 4.14).

Další postup je iterativní. Vždy je vybrána 1-skupina s nejmenším počtem nesetříděných sufixů (nechť její sufixy začínají znakem λ_2). Všechny její 2-skupiny (tj. sufixy začínající znaky $\lambda_2\lambda_3$), které dosud nebyly tříděny, se teď setřídí standardním algoritmem na třídění řetězců.

Nyní se průchodem přes tuto kompletně setříděnou 1-skupinu indukuje pořadí sufixů ve všech 2-skupinách začínajících znaky $\lambda_1\lambda_2$. Když se při průchodu narazí na sufix i, je sufix $(i-1)$ umístěn na začátek příslušné 2-skupiny a tento začátek je o jedničku posunut doprava.

Postup se opakuje dokud nejsou setříděné všechny 1-skupiny a tedy spočítáno SA_x.

Celý algoritmu je popsán v pseudokódu na obrázku 4.15. V závěrečné fázi každé iterace se právě setříděná 1-skupina prochází zepředu pro $j = \text{head}(\lambda_2, 0), \ldots, S[\lambda_2]$ a pak zezadu $j = \text{tail}(\lambda_2, \Sigma - 1), \ldots, E[\lambda_2]$. Na konci těchto průchodů bude platit, že $S[\lambda_2] > E[\lambda_2]$. Díky tomu není potřeba explicitně třídit 2-skupinu začínající znaky $\lambda_2\lambda_2$. Toto pozorování výrazně urychlí výpočet sufixového pole pro řetězec, ve kterém se vyskytují dlouhé sekvence stejných znaků.

Postup ukáži opět na slově mississippi. Po 2-třídění bude pole SA vypadat jako na obrázku 4.14.

	1	2	3	4	5	6	7	8	9	10	11
x =	m	i	s	s	i	s	s	i	p	p	i
SA =	(11	8	[2	5])	1	(10	9)	([4	7]	[3	6])

Obrázek 4.14: Sewardův algoritmus po počátečním 2-třídění. Kulaté závorky oddělují více než jednoprvkové 1-skupiny, hranaté více než jednoprvkové 2-skupiny.

Nyní se vybere 1-skupina s nejmenším počtem nesetříděných sufixů, tedy skupina obsahující jediný sufix, začínající znakem 'm' (1). Explicitní třídění není potřeba. Průchodem přes tuto 1-skupinu se neindukuje pořadí žádné 2-skupiny (sufix $i - 1 = 0$ není definován).

V druhé iteraci vybereme 1-skupinu začínající znakem 'p' (10 9). Obě její 2-skupiny jsou jednoprvkové, explicitní třídění není potřeba. Při průchodu zepředu umístíme sufix $10 - 1 = 9$ na začátek 2-skupiny začínající znaky 'pp' a zvětšíme index S['p'], tudíž ještě

musíme sufix $9 - 1 = 8$ umístit na začátek 2-skupiny 'ip'. Odzadu se tato 1-skupina neprochází.

Nyní je na řadě 1-skupina začínající 'i' (11 8 [2 5]). Po explicitním třídění 2-skupiny 'is' získáme kompletní pořadí (11 8 5 2). Průchodem odzadu dopředu získáme pořadí v 2-skupině 'si', které je [7 4].

Zbývá 1-skupina 's' (7 4 [3 6]), její 2-skupina [3 6] se explicitně netřídí, protože začíná dvěma stejnými znaky 'ss'. Při průchodu odpředu narazíme nejprve na 7 a pak na 4, pořadí sufixů ve 2-skupině 'ss' je tedy [6 3].

Nyní jsou všechny 1-skupiny sufixového pole úplně setříděny, přitom bylo potřeba jen jedno explicitní třídění dvouprvkové množiny sufixů.

```
procedure computeSA_s(x, SA)
    proveď 2-třídění sufixů x, výsledné SA_2 ulož do pole SA
    while v SA existuje nesetříděná 1-skupina do
        vyber 1-skupinu s nejmenším počtem nesetříděných sufixů
        λ_2 ← první znak sufixů v této 1-skupině
        for λ_3 ← 0 to |Σ| − 1, λ_3 ≠ λ_2 do
            if 2-skupina (λ_2, λ_3) není setříděná then sort(λ_2, λ_3)
        for λ_1 ← 0 to |Σ| − 1 do
            S[λ_1] ← head(λ_1, λ_2)
            E[λ_1] ← tail(λ_1, λ_2)
        j ← head(λ_2, 0)
        while j < S[λ_2] do
            i ← SA[j]
            if i > 1 then
                λ_1 ← x[i − 1]
                SA[S[λ_1]] ← (i − 1)
                inc(S[λ_1])
        j ← tail(λ_2, |Σ| − 1)
        while j > E[λ_2] do
            i ← SA[j]
            if i > 1 then
                λ_1 ← x[i − 1]
                SA[E[λ_1]] ← (i − 1)
                dec(S[λ_1])
```

Obrázek 4.15: Sewardův algoritmus. Makra head(λ_1, λ_2) a tail(λ_1, λ_2) jsou indexy do pole SA na první a poslední prvek 2-skupiny sufixů začínající znaky $\lambda_1\lambda_2$. Funkce sort(λ_2, λ_3) je standardní třídící funkce, která setřídí 2-skupinu začínající znaky $\lambda_1\lambda_2$.

4.3.2 Implementační detaily

Moji implementaci popisuje téměř přesně obrázek 4.15. Drobným detailem je poslední (n-tý) sufix, ten nemá definovaný druhý znak, proto musí být v samostatné 2-skupině. Pro jednoduchost jsem tedy makro head(λ_1, λ_2), pro 256ti znakovou abecedu, reprezentoval dvourozměrným polem 256x(256+1) indexů. Dále makro tail není potřeba definovat vůbec, protože tail(λ_1, λ_2) $\equiv$ head(λ_1, $\lambda_2 + 1$) - 1.

K určování 1-skupiny s nejmenším počtem nesetříděných sufixů používám zvláštní pole errs 32bitových celých čísel délky 256, kde errs[λ] udává počet sufixů ve všech nesetříděných 2-skupinách začínajících znakem λ.

K explicitnímu třídění 2-skupin sufixů jsem podle doporučení autora použil algoritmus popsaný v [11]. V zásadě se jedná o ternární quicksort, tedy quicksort, který ve fázi dělení pole podle mediánu umisťuje prvky rovné mediánu na okraje pole a až je všechno roztříděno, přesune tyto prvky doprostřed.

Algoritmu také popisuje způsob výběru mediánu: Pro pole s více než 40 prvky se použije pseudomedián z vybraných devíti prvků pole, pro pole s více než sedmi prvky je to medián z prvního, prostředního a posledního prvku pole a pro sedmiprvkové pole se bere prostřední prvek. Kratší pole se místo quicksortem třídí insertsortem.

4.3.3 Vlastnosti

Kromě prostoru pro x a SA a zásobníku pro explicitní třídění potřebuje algoritmus jen konstantní množství další paměti. Z toho největší je pole s ukazateli na začátky 2-skupin. Pro 256ti znakovou abecedu se jedná o 64KB.

Autor v [20] bez důkazu uvádí, že časová složitost tohoto algoritmu v nejhorším případě je $O(n^2 \log(n))$. Toto tvrzení jistě platí v případě, že algoritmus quicksort volaný v těle algoritmu bude vždy třídit v čase $O(n \log(n))$. Časová složitost Sewardova algoritmu závisí také na druhé mocnině velikosti abecedy.

V praxi je rychlost algoritmu závislá především na počtu sufixů, které je potřeba třídit explicitně (respektive na velikosti 2-skupin, které je nutné explicitně třídit) a na $\overline{lcp}$ každé z takto tříděných množin. Pro první podmínku je kritický vstupní soubor, ve kterém se některé dvojice po sobě jdoucích znaků vyskytují výrazně častěji než jiné. Druhá podmínka souvisí s $\overline{lcp}$ celého souboru. Celkově je algoritmus rychlý na vstupech s krátkým $\overline{lcp}$, s velkou abecedou a bez sekvence znaků, která by se často opakovala. Naopak patologická je pro Sewardův algoritmus posloupnost, ve které se stále opakují dva znaky (*abab...*, viz. 6.5).

4.4 Manziniho a Ferraginův algoritmus

4.4.1 Popis algoritmu

Manzini a Ferragina v [17] navrhli vylepšení Sewardova algoritmu. To spočívá v nahrazení ternárního quicksortu (používaného k explicitnímu třídění 2-skupin) pro tuto úlohu lepším

třídícím algoritmem.

Jejich algoritmus na konstrukci sufixového pole je tedy také popsán na obrázku 4.15 s tím rozdílem, že volání sort(λ_2, λ_3) je nahrazeno voláním Manziniho a Ferraginovy třídící procedury (obrázek 4.16) s parametry: sort_mf(head(λ_2, λ_3), tail(λ_2, λ_3)-head(λ_2, λ_3)+1, 2).

```
procedure sort_mf(a, n, depth)
    if depth < L then multikeyQuickSort(a, n, depth)
    else
        if existuje "vhodná" setříděná 2-skupina then
            setřiď pole a induktivně
        else
            if n ≤ M then blindSort(a, n, depth)
            else ternalQuickSort(a, n, depth)
```

Obrázek 4.16: Manziniho a Ferraginův algoritmus na třídění pole sufixů a délky n za předpokladu, že všechny sufixy z a mají společný prefix délky alespoň *depth*. Meze L a M jsou parametry algoritmu, autoři doporučují $L = 500$ a $M = |x|/2000$.

Algoritmus nejprve množinu sufixů a třídí rekurzivním algoritmem multikey quicksort (viz. kap. 4.4.2), ale když v rekurzi dosáhne hloubky L (tj. sufixy v množině a mají společný prefix délky L), přepne se na jiný třídící algoritmus. A to buď na blindsort (viz. kap. 4.4.3), pokud v množině a není více než M sufixů a nebo na ternární quicksort (viz. kapitola 4.3.2) v opačném případě. Navíc, pokud je to možné, pokouší se algoritmus setřídit množinu a induktivně na základě nějaké již setříděné 2-skupiny (viz. kap. 4.4.4).

4.4.2 Multikey quicksort

Multikey quicksort je obecně algoritmus na třídění pole záznamů s více než jedním klíčem (viz. [12]). V Manziniho a Ferraginově algoritmu je použit k třídění pole řetězců (každý znak řetězce je považován za jeden klíč).

Algoritmus je rekurzivní (viz obrázek 4.17). Na začátku každé iterace platí, že všechny řetězce mají společný prefix délky *depth*. Náhodně je jeden z řetězců vybrán jako pivot. Podle pivota je pole rozděleno na tři části: na řetězce, které mají na pozici (*depth*+1) menší znak než pivot, stejný znak jako pivot, větší znak než pivot. Na tyto tři části je pak algoritmus spuštěn rekurzivně s tím, že u prostřední skupiny je parametr *depth* zvětšen o jedničku.

V Manziniho a Ferraginově algoritmu je ve skutečnosti místo rekurzivního volání multikeyQuickSort spuštěna procedura sort_mf z obrázku 4.16. Tím je zajištěno přepnutí na jiný způsob třídění při dosažení hloubky L.

```
procedure multikeyQuickSort(a, n, depth)
    náhodně zvol pivota
    a_< ← řetězce, které mají na pozici (depth + 1) menší znak než pivot
    a_= ← řetězce, které mají na pozici (depth + 1) stejný znak jako pivot
    a_> ← řetězce, které mají na pozici (depth + 1) větší znak než pivot
    multikeyQuickSort(a_<, |a_<|, depth)
    multikeyQuickSort(a_=, |a_=|, depth + 1)
    multikeyQuickSort(a_>, |a_>|, depth)
```

Obrázek 4.17: Multikey quicksort.

4.4.3 Blindsort

Blindsort je třídící algoritmus založený na sufixovém trie (viz. kapitola 4.5). Autoři tuto datovou strukturu nazývají blind trie (viz. [17]), odtud plyne název blindsort. Algoritmus začíná s prázdným trie a postupně se do něj vkládají všechny řetězce tříděné množiny a. Na závěr se trie prochází do hloubky. Hrany vedoucí z každého uzlu se vždy navštěvují v pořadí, které odpovídá lexikografickému uspořádání symbolů, kterými jsou tyto hrany označeny. Při tomto průchodu jsou uzly zastupující řetězce z a navštíveny v lexikografickém pořadí.

4.4.4 Induktivní třídění

Pokud se má třídit pole sufixů a a víme, že tyto sufixy mají prvních $depth \geq L$ znaků stejných, pokusí se Manziniho a Ferraginova třídící procedura mezi prvními $depth$ znaky těchto sufixů najít dvojici $\lambda\mu$ takovou, že 2-skupina začínající znaky $\lambda\mu$ již byla setříděna. Taková 2-skupina indukuje pořadí sufixů v a.

4.4.5 Implementační detaily

Z implementace je asi nejzajímavější induktivní třídění. Pokud $a = \{s_1, s_2, \ldots s_m\}$ a znaky λ a μ se v sufixech z a vyskytují na l-té a $(l+1)$-ní pozici, pak by stačilo projít 2-skupinu $\lambda\mu$ a při navštívení sufixu (s_j+l) umístit s_j na začátek a a ukazatel na tento začátek zvětšit o 1. Protože ale 2-skupina začínající znaky $\lambda\mu$ bývá řádově větší než množina a a sufixy $(s_1+l), (s_2+l), \ldots (s_m+l)$ se v této 2-skupině vyskytují zpravidla blízko u sebe, doporučují autoři nalézt pouze jeden ze sufixů $\hat{s_j} = (s_j + l)$ a od něj pak procházet 2-skupinu doleva a doprava dokud nejsou nalezeny všechny.

Pro efektivní hledání sufixu $\hat{s_j}$ se používají dvě pomocná pole. Vstupní řetězec x je pomyslně rozdělen na d (autoři doporučují $d = 500$) segmentů a pro každý segment i se definují hodnoty *Offset[i]* a *Anchor*$[i]$: *Offset[i]* je index nejlevějšího sufixu, který začíná v i-tém segmentu a patří do některé již setříděné 2-skupiny. A pokud $\hat{s}_i =$ *Offset[i]*, pak *Anchor*$[i]$ udává pořadí sufixu $\hat{s}_i$ v rámci jeho 2-skupiny.

Nechť a je množina sufixů $s_1, s_2, \ldots s_m$, které mají společný prefix délky L. Při pokusu o induktivní třídění nejdříve pro každý sufix s_j z množiny a určíme segment t_j, do kterého sufix patří. Pak zjistíme, jestli nejlevější sufix segmentu t_j patřící do některé již setříděné 2-skupiny nezačíná dvojicí znaků (λ, μ), která se nachází mezi prvními L znaky sufixu s_j (tedy jestli $s_j <$ *Offset*[t_j]$< s_j + L$). Pokud ano, můžeme sufix $\hat{s_j}$ =*Offset*[t_j] použít pro induktivní třídění. V takovém případě budeme množinu a třídit induktivně na základě 2-skupiny $\lambda\mu$, tuto 2-skupinu budeme prohledávat od pozice $head[\lambda, \mu] + Anchor[t_j]$.

Tento postup není stoprocentně spolehlivý, ale většinou uspěje. Autoři v [17] argumentují tím, že sufixy $s_1, s_2, \ldots s_m$ většinou leží v odlišných segmentech a tedy pokud je induktivní třídění možné, s velkou pravděpodobností takto vhodný sufix $\hat{s_j}$ nalezneme.

4.4.6 Vlastnosti

Paměťové nároky Manziniho a Ferraginova algoritmu jsou stejně jako u Sewardova algoritmu přibližně $5n$ bytů pro vstupní řetězec délky n. Pomocná pole *Offset* a *Anchor* sice vyžadují lineární množství paměti, ale s velmi malou konstantou. Paměťově náročný blind sort je použit jen v případě, že tříděná množina je velmi malá.

Časová složitost v nejhorším případě je také stejná, tedy $O(n^3)$, ale vylepšená třídící procedura zlepšuje chování na některých vstupech, které byly pro Sewardův algoritmus patologické.

Díky sofistikovanějšímu explicitnímu třídění nejsou pro Manziniho a Ferraginův algoritmus takovým problémem vstupní soubory s velkým $\overline{lcp}$. Naopak kvůli režijním operacím je tento algoritmus na vstupech s velmi krátkým $\overline{lcp}$ pomalejší než Sewardův.

Na rychlost výpočtu má hlavním vliv velikost množin sufixů, které je potřeba třídit explicitně. Nejhorší je pro tento algoritmus vstup, ve kterém se často opakují některé sekvence znaků jako je tomu například u zdrojových kódů jazyka C nebo u dokumentů ve formátu XML (viz. výsledky měření v kapitole 6.5). Patologický případ posloupnosti "*abab...*" řeší Manziniho a Ferraginův algoritmus několikanásobně rychleji než Sewardův, přesto ale výrazně pomaleji než ostatní studované algoritmy.

4.5 Algoritmus založený na sufixovém stromu

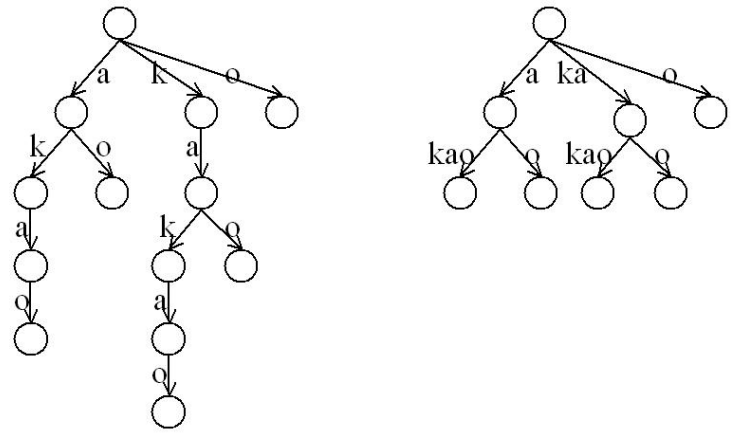
4.5.1 Popis algoritmu

Jedná se o postup, kdy se nejprve zkonstruuje sufixový strom, ten se pak prochází do hloubky tak, aby hrany vycházející z každého vrcholu byly navštíveny v pořadí odpovídajícím lexikografickému pořadí řetězců, kterými jsou tyto hrany označeny. Do sufixového pole se zapisují indexy sufixů v pořadí, ve kterém byly při průchodu stromem navštíveny.

Sufixový strom je komprimovaná varianta datové struktury trie (viz. obrázek 4.18). Trie je acyklický orientovaný graf, ve kterém každý uzel odpovídá jednomu podslovu vstupního řetězce (kořen odpovídá prázdnému slovu). Uzel odpovídající slovu s budu označovat $\overline{s}$. Hrany jsou označeny písmeny abecedy Σ, z uzlu $\overline{s}$ vede hrana označená symbolem λ do

uzlu $\bar{t}$ právě když $t = s\lambda$. V sufixovém stromu jsou vypuštěny všechny vrcholy, které mají jen jednoho potomka, hrany jsou označeny řetězci, které se skládají ze znaků na cestě mezi odpovídajícími vrcholy v trie. Tyto řetězce jsou podslova vstupního řetězce x. Při implementaci se u hran namísto řetězců ukládají jen indexy začátku a konce tohoto řetězce v x.

Uzly trie, které v sufixovém stromu chybí, se označují jako *implicitní*. Implicitní vrchol q je určen dvojicí (p, w), kde p je nejbližší předek q v trie, který je explicitní v sufixovém stromu a w je posloupnost znaků na hranách na cestě mezi p a q v trie. Dvojice (p, ϵ) popisuje explicitní vrchol p.

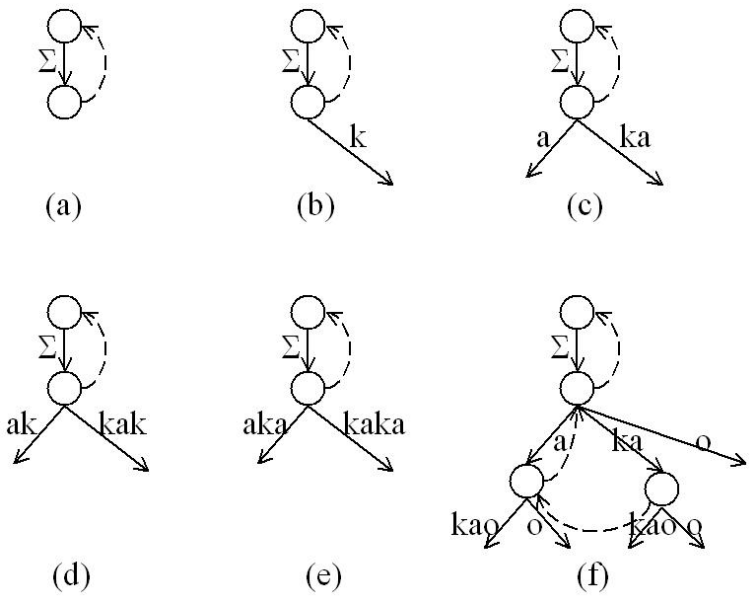


Obrázek 4.18: Vlevo trie a vpravo sufixový strom pro slovo "kakao".

Ke zkonstruování sufixového stromu jsem použil Ukkonenův algoritmus popsaný v [21]. Tento algoritmus prochází vstupní slovo zleva doprava, v každém okamžiku má kompletní sufixový strom pro již zpracovanou část řetězce. Při navštívení i-tého znaku je již zkonstruován sufixový strom pro řetězec $x[1..(i-1)]$ a znak $x[i]$ se přidá na konec každého sufixu.

Sufixový strom je pro potřeby Ukkonenova algoritmu obohacen o pomocný uzel $\perp$, do kterého nevede žádná hrana a ze kterého vede hrana pro každý znak abecedy do kořenu. Ve stromu jsou dále odstraněny listy. Hrany, které do listů vedly, zůstávají a označují se jako *otevřené*. Navíc se definují tzv. *sufixové hrany*: z uzlu $\bar{s}$ vede sufixová hrana do uzlu $\bar{t}$, právě když existuje $\lambda \in \Sigma$, že $s = \lambda t$. Příklad takto upraveného sufixového stromu je na obrázku 4.19.

Ukkonen v sufixovém stromě dále definuje tzv. *hraniční posloupnost* vrcholů a ukazuje, že při přidávání i-tého znaku (tj. znaku $x[i]$) stačí pouze projít vrcholy v této posloupnosti a za každý připojit hranu označenou $x[i]$, pokud taková hrana dosud neexistuje. Hraniční posloupnost je posloupnost explicitních nebo implicitních vrcholů $(\overline{s_1}, w_1)$, $(\overline{s_2}, w_2)$, ..., $(\overline{s_i}, w_i)$ takových, že $s_1 w_1 = x[1..(i-1)]$, $s_2 w_2 = x[2..(i-1)]$, ..., $s_{i-1} w_{i-1} = x[i-1]$, $s_i w_i = \epsilon$.



Obrázek 4.19: Sufixový strom na začátku Ukkonenova algoritmu a po vložení jednotlivých písmen slova "kakao". Přerušované šipky značí sufixové hrany.

Příklad hraniční posloupnosti v sufixovém stromě:
pro slovo "kaka" (obázek 4.19(e)):
(kořen, "kaka"), (kořen, "aka"), (kořen, "ka"), (kořen, "a"), (kořen, ϵ).
pro slovo "kakao" (obrázek 4.19(f)):
($\overline{ka}$, "kao"), ($\overline{a}$, "kao"), ($\overline{ka}$, "o"), ($\overline{a}$, "o"), (kořen, "o"), (kořen, ϵ).

K přechodu k dalšímu členu hraniční posloupnosti slouží sufixové hrany.

Při přidávání znaku $x[i]$ je hraniční posloupnost rozdělena na tři části: na vrcholy odpovídající listům, na vrcholy, ze kterých nevede hrana začínající znakem $x[i]$ a na vrcholy, ze kterých hrana začínající znakem $x[i]$ již vede.

1. Do první skupiny patří právě implicitní vrcholy na koncích otevřených hran. Pro přidání $x[i]$ na konec takovéhoto sufixu stačí prodloužit slovo, kterým je daná otevřená hrana označena o znak $x[i]$. Ve skutečnosti se ale tato operace explicitně neprovádí. Slova označující otevřené hrany mají definovanou pouze pozici ve slově x, na které začínají. Jsou to sufixy právě zpracované části x, proto na konci každé iterace končí všechna na pozici i a není potřeba to u každého zvlášť uvádět.

2. Ve druhé skupině jsou explicitní i implicitní vrcholy. K explicitním stačí přidat novou otevřenou hranu označenou slovem začínajícím na pozici i. Z implicitních vrcholů

se nejprve stanou explicitní: Hrana, na které leží se rozdělí na dvě části. K nově vzniklému vrcholu se pak přidá otevřená hrana začínající na pozici i.

3. Vrcholy ze třetí skupiny se nemusí upravovat vůbec. Jedná se buď o explicitní vrcholy, ze kterých hrana pro znak $x[i]$ už vede, nebo o implicitní vrcholy, které leží na hraně, která "pokračuje" znakem $x[i]$ (tj. pro každý takový vrchol (p, w) existuje také vrchol $(p, wx[i])$.

Každá z těchto tří skupin je v hraniční posloupnosti souvislá. První prvek druhé skupiny se označuje jako *aktivní bod*. Při přidávání znaku $x[i]$ se začne u tohoto aktivního bodu a pomocí sufixových hran se pokračuje po hraniční posloupnost, dokud se nenarazí na vrchol patřící do třetí skupiny. Ke každému navštívenému vrcholu se přidá hrana začínající na pozici i. Algoritmus je popsán na obrázku 4.20.

```
procedure buildTree(x, n)
    vytvoř uzly ⊥ a kořen
    pro všechny znaky z Σ vytvoř hranu z ⊥ do kořenu
    vytvoř sufixovou hranu z kořenu do ⊥
    (p, w) ←(kořen, ϵ)
    for i ← 1 to n do (p, w) ← update((p, w), i)

procedure update((p, w), i)
    q ← make_explicit(p, w)
    old_q ← kořen
    while z q nevede hrana začínající x[i] do
        přidej k q novou otevřenou hranu pro znak x[i]
        if old_q ≠ kořen then
            vytvoř sufixovou hranu z old_q do q
        old_q ← q
        (p, w) ← canonize(konec sufixové hrany z p, w)
        q ← make_explicit(p, w)
    if old_q ≠ kořen then
        vytvoř sufixovou hranu z old_q do p
    return canonize(p, wx[i])
```

Obrázek 4.20: Ukkonenů algoritmus na konstrukci sufixového stromu. Vrchol (p, w) je na začátku každé iterace aktivní bod. Funkce make_explicit otestuje, zda její parametry odpovídají explicitnímu vrcholu a pokud ne, udělá z tohoto implicitního vrcholu explicitní. Funkce canonize dělá z dvojice $(\overline{s}, w)$ implicitní vrchol $(\overline{s'}, w')$ tak, že $sw = s'w'$.

Když je hotov sufixový strom pro celý řetězec x, je potřeba tento strom projít do hloubky a kdykoli narazíme na uzel odpovídající sufixu x, přidáme index této přípony do sufixového pole. K tomu je výhodné, aby všechny tyto uzly byly explicitní. Vzhledem k tomu, že uzly odpovídající sufixům jsou právě uzly v hraniční posloupnosti, stačí na závěr

ještě jednou po této posloupnosti projít a udělat z každého implicitního vrcholu na této cestě explicitní. V zásadě to přesně odpovídá tomu, jako bychom na závěr do stromu přidali znak $x[n+1]$, který je lexikograficky menší než všechny znaky Σ.

4.5.2 Implementační detaily

Mým hlavním problémem při implementaci byl návrh datových struktur pro reprezentaci sufixového stromu.

Ukázalo se, že není výhodné udržovat dvě struktury - uzel a hrana, ale že rychlejší a paměťově úspornější je používat pouze strukturu pro uzly, která odkazuje přímo na potomky.

Dále bylo několik možností jak reprezentovat potomky uzlu. Nakonec jsem se rozhodl pro variantu, kdy každý uzel má ukazatel na prvního syna (pokud existuje) a každý syn ukazuje na jednoho svého bratra (pokud existuje). Tento seznam synů, především z důvodu závěrečné rekonstrukce sufixového pole, udržuji lexikograficky setříděný.

Speciálním případem je uzel $\perp$. Vzhledem k tomu, že při průchodu po hraniční posloupnosti bývá tento uzel často navštěvován a jeho potomkem je jen uzel "kořen", zdálo se výhodnější nevytvářet z něho hrany pro všechny znaky abecedy, ale ošetřit ho zvlášť.

Ukázalo se také, že je výhodné alokovat vždy více uzlů najednou. Není úplně jasné jak určit velikost těchto bloků uzlů. Čím jsou bloky větší, tím je výpočet rychlejší, ale dochází také k většímu plýtvání paměti. V implementaci, se kterou jsem prováděl experimenty, je jeden blok velký $n/32$ uzlů.

4.5.3 Vlastnosti

Pro vstup délky n může sufixový strom obsahovat až $2n$ uzlů. Jeden uzel v mojí implementaci zabírá 20 bytů. Dohromady tedy algoritmus může potřebovat až $45n$ bytů paměti ($2n * 20$ na strom + $5n$ na vstupní řetězec a sufixové pole). I když většinou je tato hodnota nižší (viz. kapitola 6.5), zůstává Ukkonenův algoritmus paměťově výrazně nejnáročnější ze všech studovaných metod.

Časová složitost algoritmu je lineární: Při procházení po hraniční posloupnosti je v každém kroku přidán jeden nebo dva vrcholy a celkem je ve stromě maximálně $2n$ vrcholů. Závěrečné procházení stromem do hloubky a konstrukce sufixového pole má také lineární časovou složitost, protože každý vrchol je navštíven jen jednou.

Pro skutečnou rychlost tohoto algoritmu se zdá rozhodující velikost abecedy vstupního řetězce, čím větší je abeceda, tím je výpočet pomalejší. Tato závislost je způsobena tím, že pro velkou abecedu má každý uzel typicky dlouhý seznam synů, který je potřeba procházet. Chování algoritmu na vstupech nad velkými abecedami by bylo možné vylepšit použitím jiné reprezentace sufixového stromu, například použitím hašovaní namísto prostého seznamu synů.

Dále algoritmu vyhovují vstupy, ve kterých se vyskytují některé podřetězce výrazně často, jako je tomu u zdrojových kódu jazyka C nebo u XML-dokumentů. V sufixovém

stromě pak vznikají hrany označené buď celými nebo částmi těchto podřetězců, tj. ve stromě je mnoho vrcholů implicitních.

Patologické jsou tedy vstupy nad velkou abecedou bez často se opakujících podřetězců jako jsou zkomprimované soubory nebo náhodné posloupnosti.

Kapitola 5

Algoritmus na kontrolu správnosti

Pro kontrolování správnosti spočítaného sufixového pole jsem použil algoritmus, který Burkhardt a Kärkkäinen popsali v [13]. Algoritmus má jako vstup řetězec x a pole celých čísel SA, oboje délky n. Na výstupu vrací *true*, v případě že SA obsahuje sufixové pole pro x, *false* v opačném případě. Algoritmus je založený na tvrzení, že SA je sufixové pole pro x právě když splňuje tyto tři podmínky:

1. $SA[i] \in \{1, 2, \ldots, n\}, \forall i = 1, 2, \ldots, n$
2. $x[SA[i-1]] \leq x[SA[i]], \forall i = 2, 3, \ldots, n$
3. pokud $(x[SA[i-1]] = x[SA[i]])$ a $(SA[i-1] \neq n)$ pro nějaké $i \in \{2, 3, \ldots, n\}$, pak existují j, k taková, že $SA[j] = SA[i-1] + 1$, $SA[k] = SA[i] + 1$ a $j < k$.

První dvě podmínky se otestují snadno. Třetí podmínka je také jednoduchá za předpokladu, že máme k dispozici inverzní sufixové pole. V [13] je popsána i varianta, která ISA nepotřebuje. Pro naše účely ale stačí tato paměťově náročnější verze. Algoritmus je blíže popsán na obrázku 5.1.

```
procedure check(x, SA, n)
    for i ← 1 to n do
        if SA[i] ∈ {1, 2, ..., n} then ISA[SA[i]] ← i
        else return false
    for i ← 2 to n do
        if x[SA[i − 1]] > x[SA[i]] then return false
        if (x[SA[i − 1]] = x[SA[i]]) and (SA[i − 1] ≠ n) then
            j ← ISA[SA[i − 1] + 1]
            k ← ISA[SA[i] + 1]
            if j ≥ k then return false
    return true
```

Obrázek 5.1: Burkhardtův a Kärkkäinenův algoritmus na kontrolu správnosti sufixového pole.

Kapitola 6

Srovnání algoritmů

Implementované algoritmy jsem porovnával z hlediska skutečné rychlosti a skutečných paměťových nároků na souboru dat různého typu.

Při výběru vstupních dat pro experimenty jsem se inspiroval prací [19], ve které pro testování používají data z běžných korpusů (viz. [1]) a prací [18], ve které algoritmy testují na náhodných datech různé délky a velikosti abecedy. Pokusil jsem se spojit oba tyto přístupy. Sestavil jsem vlastní množinu testovacích souborů, ve které jsou zastoupeny nejběžnější formáty dat. Z těchto souboru jsem pak vybral podřetězce různých délek pro zjištění chování algoritmů na různě velkých datech stejného typu. Mimoto jsem provedl experimenty i s náhodnými daty různých délek nad různě velkými abecedami stejně jako v [18].

6.1 Měření času

Do měření času jsem zahrnul jen skutečnou dobu výpočtu sufixového pole, tedy měřit jsem začal až po načtení vstupního řetězce a skončil před uložením výsledného sufixového pole.

K samotnému měření jsem použil systémové funkce jazyka C settimer() a gettimer(). Pomocí settimer() lze nastavit výchozí hodnotu časovače, tato hodnota se pak postupně snižuje. Funkcí gettimer() se získá aktuální hodnota časovače.

Na začátku výpočtu tedy nastavím časovač na dostatečně velkou hodnotu, na konci výpočtu přečtu aktuální stav a rozdíl obou čísel dává dobu výpočtu.

Tento způsob měření jsem zvolil ze dvou důvodů. Jednak hodnota časovače se mění poměrně často, přibližně každých 0.001s, na rozdíl od běžné funkce jazyka C na měření času time(). Druhý důvod je, že funkce settimer() a gettimer() umožňují pracovat i s tzv. virtuálním časovačem. Jedná se o časovač, který snižuje svou hodnotu jen po dobu, kdy je danému procesu přidělen procesor.

Hodnoty v tabulkách v kapitole 6.5 jsou naměřeny pomocí virtuálního časovače. Při použití reálného časovače jsem dostal hodnoty v průměru o dvacet procent vyšší.

Při měření velmi krátkých časových úseků jsem zpozoroval podezřelé chování těchto časovačů. Časovače těsně po začátku měření svou hodnotu vždy nejprve nepatrně zvýši

a pak teprve začnou postupně snižovat. Kvůli tomu považuji za relevantní jen naměřené doby delší než 0.001s.

6.2 Měření paměťových nároků

K měření spotřebované paměti jsem použil knihovnu libmemusage.so, která je běžnou součástí distribucí OS Linux. Knihovna se dynamicky přilinkuje k testovanému programu pomocí proměnné prostředí LD_PRELOAD:

```
LD_PRELOAD=/lib/libmemusage.so MEMUSAGE_OUTPUT=datafile prog
```

To má za následek, že po skončení programu *prog* je na terminál vypsána statistika využití haldy a zásobníku. Tato statistika je také v binární podobě uložena do souboru *datafile*, ze kterého lze vyčíst tzv. memory-peek, tj. největší množství najednou použité paměti.

Tento způsob jsem použil jednak kvůli jeho jednoduchosti a také proto, že měří zároveň paměť alokovanou na haldě i na zásobníku a nezkresluje výsledek další spotřebou paměti, která je závislá na operačním systému.

6.3 Testované algoritmy

Pro experimenty jsem použil jednak vlastní implementace pěti popsaných algoritmů a pro srovnání také volně dostupné implementace běžných třídících algoritmů: mergesort, quicksort, heapsort a shellsort. Konkrétně u algoritmu quicksort jsem použil implementaci z [5], u heapsortu implementaci z [4] a u shellsortu z [6]. Pro třídění mergesortem používám knihovní funkci qsort, která v linuxu, pokud je dostatek volné operační paměti, třídí vstup právě algoritmem mergesort.

Každý z těchto algoritmů jsem dále rozšířil pro práci se vstupním řetězcem nad až 16ti bitovou abecedou. Toto rozšíření je u běžných třídících algoritmů stejně jako u Manberova a Myersova algoritmu a Kärkkäinenova a Sandersova algoritmu i v případě konstrukce sufixového pole pomocí sufixového stromu triviální.

Problém nastává u Sewardova a také u Manziniho a Ferraginova algoritmu. Tyto dvě metody potřebují pro svou práci pole *head* velikosti $4*|\Sigma|^2$ bytů, to by v případě 16ti bitové abecedy znamenalo 16GB. Z toho důvodu jsem tyto dva algoritmy upravil tak, že zpracovávají vstup jako by se jednalo o řetězec nad osmi bitovou abecedou a z takto spočítaného sufixového pole pak do výsledku překopírují jen sudé indexy sufixů vydělené dvěma. Ve skutečnosti je před samotným výpočtem ještě potřeba 16ti bitové znaky převést z little-endianu standardně používaného v linuxu a ve windows na big-endian. Šestnácti bitové verze těchto dvou algoritmů tedy pracují s dvojnásobně dlouhým vstupem než ostatní, ale s menší abecedou.

6.4 Testovací data

Algoritmy jsem testoval na třech okruzích dat: na reálných souborech, na náhodně generovaných datech a na speciálních posloupnostech. V tabulce 6.1 je seznam těchto souborů spolu s údaji o jejich velikostech a průměrném společném prefixu. U souborů nad 16ti bitovou abecedou je uvedena velikost v bytech, počet znaků je tedy poloviční.

soubor	popis	velikost	$\overline{lcp}$
bin	program gimp-2.2 pro linux	2.94 MB	252.49
bmp	nekomprimovaná fotografie 1280x960x24	3.51 MB	5.19
csource	část zdrojového kódu programu gimp	3.64 MB	90.41
DNA	DNA bakterie E.coli	4.42 MB	17.38
text-cz	český překlad bible	3.27 MB	10.33
text-en	anglický překlad bible	3.85 MB	13.97
xml	factbook.xml z www.w3.org	4.02 MB	56.20
zip	zkomprimovaný Large Corpus (E.coli, bible.txt, world192.txt)	3.10 MB	2.07
text-jp	japonský text	2.22 MB	3.99
text-ch	čínský text	2.93 MB	9.16
r2	pseudonáhodná posloupnost dvou různých znaků	2.00 MB	19.89
r4	pseudonáhodná posloupnost čtyř různých znaků	2.00 MB	9.69
r8	pseudonáhodná posloupnost osmi různých znaků	2.00 MB	6.29
r16	pseudonáhodná posloupnost šestnácti různých znaků	2.00 MB	4.60
r32	pseudonáhodná posloupnost 32 různých znaků	2.00 MB	3.58
r64	pseudonáhodná posloupnost 64 různých znaků	2.00 MB	2.93
r128	pseudonáhodná posloupnost 128 různých znaků	2.00 MB	2,36
r256	pseudonáhodná posloupnost 256 různých znaků	2.00 MB	2.03
r512	pseudonáhodná posloupnost 512 různých znaků	4.00 MB	1.88
r1024	pseudonáhodná posloupnost 1024 různých znaků	4.00 MB	1.57
r2048	pseudonáhodná posloupnost 2048 různých znaků	4.00 MB	1.21
r4096	pseudonáhodná posloupnost 4096 různých znaků	4.00 MB	1.06
r8192	pseudonáhodná posloupnost 8192 různých znaků	4.00 MB	1.01
r16384	pseudonáhodná posloupnost 16384 různých znaků	4.00 MB	1.00
r32768	pseudonáhodná posloupnost 32768 různých znaků	4.00 MB	0.99
r65536	pseudonáhodná posloupnost 65536 různých znaků	4.00 MB	0.97
aaaa	soubor ve kterém se stále opakuje znak a	64.00 kB	32768.00
abab	soubor ve kterém se pravidelně opakují znaky a,b	64.00 kB	32767.00
abca	soubor ve kterém se pravidelně opakují znaky a,b,c	64.00 kB	32766.00
sigma	soubor ve kterém se pravidelně opakuje abeceda (256 znaků)	64.00 kB	32513.49

Obrázek 6.1: Data použitá pro testování rychlosti a paměťových nároků konstrukčních algoritmů.

Mezi reálné soubory jsem vybral osm běžných typů dat: spustitelný soubor (konkrétně gimp-2.2 distribuovaný spolu s SuSE Linuxem ver. 10.1), nekomprimovaný obrázek (fotografie o rozměrech 1280x960x24), zdrojový kód jazyka C (pospojovaná část zdrojových souborů programu GIMP v jazyce C dostupných na [9]), řetězec DNA (DNA bakterie E.coli z [1]), český text (bible z [2]), anglický text (bible z [1]), xml soubor (dostupný

na [3]) a komprimovaný soubor (zkomprimovaný Large Corpus z [1]). Velikost souborů se pohybuje od 3 do 4,5 megabytů a $\overline{lcp}$ od 2 do 252.

Z každého z těchto souboru jsem nechal náhodně vybrat vždy deset souvislých úseků délek 128, 256, 512, ... až 2 miliónů bytů (2^{21} bytů) pro zjištění chování algoritmů na datech stejného typu, ale menší velikosti.

Do testovaného skupiny reálných dat jsem přidal ještě dva soubory nad 16ti bitovou abecedou - japonský a čínský text. Je to vždy několik kratších textů pospojovaných do jednoho souboru, v případě japonštiny jsem texty čerpal z [8] a v případě čínštiny z [7]. Soubory jsou kódovány v UTF-16 (little-endian), každý znak právě 16ti bity. V souboru text-jp se vyskytuje 3850 a v souboru text-ch 4974 různých znaků.

Pro testování algoritmů na kratší vstupy stejného charakteru jsem z těchto dvou souborů náhodně vybral vždy deset podřetězců délek 128, 256, ... až 2^{19} znaků.

Druhou skupinou jsou náhodná data. Za pomoci standardního náhodného generátoru jazyka C jsem vytvořil pseudonáhodné posloupnosti délek 128, 256, 512, ... až 2^{21} znaků pro abecedy velikosti 2, 4, 8, ... až 2^{16} znaků. Pro každou dvojici (délka, velikost abecedy) mám čtyři různé posloupnosti. Údaj $\overline{lcp}$ v tabulce 6.1 je vždy průměr ze čtyř $\overline{lcp}$ posloupností délky 2^{21} bytů. Hodnoty $\overline{lcp}$ kratších souborů jsou uloženy na přiloženém CD v samostatném dokumentu.

V souborech nad abecedou velikosti 2 až 256 znaků odpovídá každý symbol jednomu bytu, ve zbývajících souborech je znak kódován vždy dvojicí bajtů v pořadí little-endian.

Třetí skupinou jsou uměle zkonstruované posloupnosti znaků, které měly odhalit slabiny některých ze studovaných algoritmů. Jsou to posloupnosti ve kterých se opakuje 1, 2, 3 nebo 256 znaků stále dokola.

Vzhledem k tomu, že u těchto dat jsou zajímavé jen algoritmy, které pro tyto vstupní posloupnosti běží výrazně dlouho a protože se tato vada ukazuje již pro krátké posloupnosti, testoval jsem všechny metody jen na souborech délek 128, 256, 512, ... až 2^{16}. (Údaj $\overline{lcp}$ v tabulce 6.1 platí pro posloupnosti délky 2^{16} bytů, pro ostaní délky je tento údaj uveden na přiloženém CD v samostatném dokumentu). V tabulkách v kapitole 6.5 jsou u některých algoritmů uvedeny doby běhu i pro posloupnosti až do délky 2^{21} bytů, není tomu tak ale u všech, vzhledem k tomu, že v některých případech byly tyto doby neúměrně dlouhé.

6.5 Výsledky

Měření jsem provedl na počítači s procesorem AMD Sempron 2600+ s operační pamětí 256 MB a operačním systémem Ubuntu Linux verze 6.06. Všechny algoritmy byly přeloženy programem gcc verze 4.0 s optimalizací -O3. Protože měření paměti může ovlivnit celkový čas výpočtu, bylo měření času a paměti provedeno zvlášť.

Každé číslo v tabulce je průměrem vždy z nejméně čtyř naměřených hodnot.

6.5.1 Doby výpočtu nad reálnými daty

Následují naměřené rychlosti algoritmů pro reálná data. Každá z tabulek 6.2 až 6.11 obsahuje doby běhu (v sekundách) všech algoritmů vždy pro jeden vstupní soubor a jeho úseky. Délky úseků jsou uvedeny v záhlaví tabulek ve znacích, *full* znamená původní soubor.

Hodnoty v tabulce pro velikosti 2^7 až 2^{20} znaků jsou průměry z naměřených hodnot pro deset úseků této velikosti, hodnota ve sloupci *full* je průměr ze čtyř běhů algoritmu nad původním souborem.

V prvním sloupci tabulek jsou uvedeny zkratky konstrukčních algoritmů (*merge* pro mergesort, *quick* pro quicksort, *heap* pro heapsort, *shell* pro shellsort, *mm* pro Manberův a Myersův algoritmus, *ks* pro Kärkkäinenův a Sandersův algoritmus, *s* pro Sewarův algoritmus, *mf* pro Manziniho a Ferraginův algoritmus a *tree* pro algoritmus založený na sufixovém stromě).

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	full
merge	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.009	0.021	0.047	0.093	0.209	0.489	1.188	32.393
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.011	0.025	0.056	0.111	0.253	0.639	1.646	56.883
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.005	0.012	0.029	0.067	0.161	0.481	1.399	3.695	66.367
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.007	0.018	0.044	0.095	0.231	0.638	2.065	6.307	99.124
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**<0.001**	0.003	0.014	0.065	0.261	0.967	2.497	5.581	31.667
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.002**	**0.008**	0.025	0.094	0.359	1.074	2.622	9.202
s	<0.001	<0.001	0.001	0.002	0.002	0.003	0.005	0.007	0.010	**0.016**	**0.028**	**0.064**	**0.152**	**0.371**	3.703
mf	<0.001	<0.001	0.001	0.002	0.002	0.004	0.005	0.008	0.011	0.019	0.034	0.073	0.163	0.373	**1.347**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.009	0.023	0.047	0.196	0.756	2.369	6.632	21.996

Obrázek 6.2: Doby výpočtu sufixového pole pro vstup **bin**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	full
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.016	0.039	0.086	0.192	0.429	0.939	3.815
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.020	0.045	0.102	0.228	0.503	1.114	4.765
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.010	0.024	0.056	0.148	0.408	1.124	2.994	14.121
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.013	0.036	0.086	0.221	0.529	1.301	3.666	18.483
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.008	0.038	0.120	0.365	0.929	3.325	14.504
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**<0.001**	**0.002**	0.007	0.020	0.065	0.196	0.532	1.858	8.002
s	<0.001	<0.001	<0.001	<0.001	0.001	0.002	0.002	0.003	**0.006**	**0.011**	**0.022**	**0.048**	0.108	0.235	1.120
mf	<0.001	<0.001	<0.001	0.001	0.001	0.002	0.003	0.005	0.007	0.012	0.023	0.048	**0.105**	**0.226**	**1.045**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.017	0.055	0.157	0.402	0.967	2.462	17.511

Obrázek 6.3: Doby výpočtu sufixového pole pro vstup **bmp**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	full
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.004	0.010	0.027	0.083	0.205	0.458	1.173	3.030	11.322
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.013	0.035	0.110	0.270	0.591	1.600	4.303	16.110
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.013	0.034	0.104	0.281	0.738	2.098	5.458	24.411
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.008	0.022	0.057	0.173	0.465	1.173	3.593	11.039	69.519
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.006	0.022	0.108	0.413	1.241	3.145	8.280	35.483
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.001**	**0.003**	0.010	**0.030**	0.112	0.365	0.967	2.333	11.063
s	<0.001	<0.001	<0.001	0.001	0.001	0.001	0.002	0.005	**0.010**	0.034	0.084	0.180	0.509	1.250	4.474
mf	<0.001	<0.001	0.001	0.001	0.002	0.002	0.004	0.010	0.039	0.083	0.143	0.250	0.634	2.233	5.823
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.013	0.032	**0.072**	**0.171**	**0.349**	**0.700**	**3.861**

Obrázek 6.4: Doby výpočtu sufixového pole pro vstup **csource**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	full
merge	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.009	0.022	0.051	0.116	0.264	0.620	1.490	9.422
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.011	0.027	0.061	0.136	0.324	0.846	2.122	13.642
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.013	0.029	0.070	0.189	0.574	1.649	4.222	27.010
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.008	0.020	0.051	0.124	0.320	0.935	2.630	7.516	73.894
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.011	0.097	0.394	1.140	3.451	8.170	47.817
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.007	0.029	0.124	0.415	1.206	2.897	15.877
s	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.002	0.006	0.014	0.032	0.080	0.232	0.611	4.009
mf	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	**0.002**	**0.005**	**0.011**	**0.024**	**0.062**	**0.195**	**0.526**	**3.258**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.005	0.019	0.054	0.135	0.311	0.695	1.553	8.824

Obrázek 6.5: Doby výpočtu sufixového pole pro vstup **DNA**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	full
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.018	0.042	0.096	0.222	0.540	1.282	5.593
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.009	0.022	0.049	0.112	0.267	0.737	1.832	8.078
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.010	0.025	0.061	0.170	0.531	1.546	3.933	17.432
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.016	0.040	0.096	0.239	0.662	2.160	6.022	34.364
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.015	0.080	0.356	1.273	2.840	6.059	24.869
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	**0.002**	0.008	0.031	0.134	0.456	1.192	2.762	10.998
s	<0.001	<0.001	0.001	0.001	0.001	0.002	0.003	0.004	**0.007**	**0.014**	**0.030**	**0.070**	**0.198**	**0.483**	2.128
mf	<0.001	<0.001	0.001	0.001	0.001	0.002	0.003	0.005	0.009	0.017	0.035	0.082	0.220	0.484	**2.047**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.006	0.019	0.060	0.160	0.389	0.812	1.933	8.460

Obrázek 6.6: Doby výpočtu sufixového pole pro vstup **text-cz**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	full
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.009	0.020	0.047	0.106	0.245	0.603	1.425	7.376
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.010	0.024	0.055	0.125	0.296	0.829	2.052	10.591
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.011	0.028	0.065	0.180	0.554	1.603	4.085	21.838
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.017	0.043	0.106	0.265	0.743	2.319	6.478	43.002
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.018	0.096	0.416	1.082	3.269	7.537	33.194
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**<0.001**	**0.003**	0.009	0.030	0.133	0.434	1.185	2.799	12.701
s	<0.001	<0.001	<0.001	<0.001	0.001	0.001	0.002	0.003	**0.007**	**0.014**	**0.033**	**0.081**	**0.225**	**0.561**	2.930
mf	<0.001	<0.001	<0.001	<0.001	0.001	0.001	0.002	0.005	0.010	0.020	0.040	0.090	0.263	0.587	**2.594**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.005	0.016	0.048	0.125	0.299	0.623	1.408	7.652

Obrázek 6.7: Doby výpočtu sufixového pole pro vstup **text-en**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	full
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.007	0.019	0.048	0.117	0.301	0.755	1.818	10.771
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.009	0.023	0.059	0.144	0.388	1.046	2.591	15.398
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.010	0.026	0.068	0.191	0.601	1.724	4.430	26.535
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.006	0.016	0.043	0.114	0.298	0.830	2.539	7.163	57.945
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.019	0.113	0.496	1.629	3.747	8.075	41.481
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.001**	**0.003**	0.009	0.031	0.128	0.421	1.158	2.773	13.194
s	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.002	0.003	**0.007**	**0.016**	**0.039**	**0.108**	**0.297**	**0.748**	**4.578**
mf	<0.001	<0.001	<0.001	0.001	0.001	0.002	0.003	0.005	0.017	0.042	0.099	0.254	0.488	1.024	4.760
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.015	0.044	0.111	0.251	0.548	1.221	5.981

Obrázek 6.8: Doby výpočtu sufixového pole pro vstup **xml**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	full
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.014	0.033	0.075	0.175	0.426	1.050	4.406
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.007	0.017	0.037	0.088	0.212	0.582	1.475	6.273
heap	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.021	0.050	0.146	0.486	1.432	3.727	15.680
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.013	0.033	0.079	0.191	0.579	1.850	5.520	30.363
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**<0.001**	**0.004**	0.033	0.175	0.553	1.278	2.706	11.884
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.031	0.119	0.344	0.856	1.903	6.732
s	0.001	0.001	0.002	0.002	0.002	0.004	0.007	0.010	0.015	**0.025**	**0.047**	**0.096**	**0.208**	**0.469**	1.846
mf	<0.001	0.001	0.002	0.002	0.003	0.004	0.007	0.011	0.017	0.028	0.055	0.108	0.223	0.483	**1.822**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.023	0.114	0.530	1.980	5.864	14.404	32.783	121.732

Obrázek 6.9: Doby výpočtu sufixového pole pro vstup **zip**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	full
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.015	**0.036**	0.083	0.203	0.505	1.402
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.007	0.017	0.041	0.098	0.275	0.704	1.957
heap	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.009	0.023	0.060	0.198	0.613	1.652	4.627
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.013	0.034	0.084	0.243	0.877	2.490	8.511
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.001**	**0.004**	0.016	0.086	0.367	1.157	2.681	6.303
ks	<0.001	<0.001	<0.001	<0.001	0.001	0.001	0.003	0.006	**0.013**	0.045	0.176	0.527	1.322	3.270
s	0.001	0.001	0.003	0.004	0.005	0.008	0.010	0.014	0.022	0.039	**0.082**	0.193	0.490	1.330
mf	0.001	0.002	0.003	0.004	0.006	0.008	0.011	0.016	0.025	0.043	0.084	**0.190**	**0.455**	**1.168**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.010	0.036	0.122	0.422	1.485	5.158	14.045	56.319

Obrázek 6.10: Doby výpočtu sufixového pole pro vstup **text-jp**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	full
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.007	0.015	0.035	0.080	0.202	0.520	2.107
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.007	0.017	0.040	0.094	0.265	0.726	2.978
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.010	0.022	0.058	0.189	0.587	1.640	6.660
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.013	0.032	0.083	0.226	0.739	2.277	10.734
mm	<0.001	<0.001	<0.001	<0.001	<0.001	**<0.001**	**0.001**	**0.003**	0.013	0.071	0.302	0.848	2.006	12.723
ks	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.002	0.005	**0.012**	0.042	0.163	0.463	1.168	5.034
s	0.001	0.002	0.002	0.004	0.006	0.008	0.009	0.012	0.019	**0.034**	**0.068**	**0.157**	0.404	**1.559**
mf	0.001	0.002	0.002	0.004	0.006	0.009	0.011	0.015	0.022	0.039	0.076	0.161	**0.375**	1.841
tree	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.015	0.057	0.277	1.182	4.339	9.972	28.606	149.271

Obrázek 6.11: Doby výpočtu sufixového pole pro vstup **text-ch**.

Z tabulek vyplývá, že jednodušší algoritmy s nízkou časovou složitostí v nejhorším případě (Manberův a Myersův algoritmus, Kärkkäinenův a Sandersův algoritmus) mají dobré výsledky na kratších vstupech, zatímco pro delší soubory vyhrávají sofistikovanější algoritmy (Sewardův a Manziniho a Ferraginův) i když jejich teoretická složitost v nejhorším případě je výrazně horší. To platí obecně bez ohledu na typ vstupních dat.

Zajímavě se chová algoritmus založený na sufixovém stromě. Pro některá data, jako *bin*, *bmp* a hlavně pro *zip*, výrazně zaostává za nejlepšími. Naopak na *csource* je jednoznačně nejrychlejší. Pravděpodobně je to pro to, že soubor *csource* má velké $\overline{lcp}$ a zároveň malý počet různých znaků a slov, která se stále opakují. Díky tomu má sufixový strom v průběhu konstrukce mnoho implicitních vrcholů. Naproti tomu v souboru *zip*, na kterém dopadl sufixový strom nejhůře, je $\overline{lcp}$ velmi krátké, soubor jistě obsahuje všech 256 znaků a posloupnosti symbolů se v něm typicky neopakují, z toho důvodu jsou ve stromě téměř všechny vrcholy explicitní.

Doby výpočtu standardních třídících algoritmů jsou podle očekávání přímo závislé na $\overline{lcp}$ vstupního řetězce. Nejpomalejší jsou na spustitelném souboru a na zdrojovém kódu jazyka C, které mají průměrný společný prefix 252 a 90, naopak poměrně rychle běží na komprimovaném souboru nebo na bitmapě a na japonském textu, které mají všechny $\overline{lcp}$ do pěti.

Manberův a Myersův algoritmus běžel rychle na krátkých souborech, je to díky jeho jednoduchosti, nepoužívá žádné režijní operace, které by se vyplatily až pro delší vstu-

py. Nejpomalejší je pro soubory s velkou délkou nejdelší shody (*lml*). Nejlépe dopadl pro *zip*-soubor, ve kterém nebývají dva sufixy s dlouhým společným prefixem.

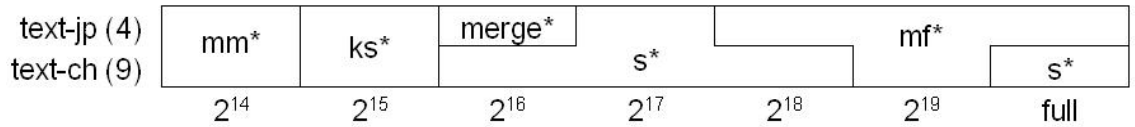
Kärkkäinenův a Sandersův algoritmus nemá tak velké rychlostní výkyvy pro různé soubory. Ze stejného důvodu jako *mm*, má i *ks* nejlepší výsledky na krátkých vstupech. Z naměřených hodnot se dá vysledovat závislost, podle které Kärkkäinenův a Sandersův algoritmus běží tím rychleji, čím je zastoupení jednotlivých znaků rovnoměrnější.

Sewardův algoritmus a Manziniho a Ferraginova varianta tohoto algoritmu byly nejrychlejší z testovaných algoritmů na většině vstupních dat. V průměru dopadl lépe Sewardův algoritmus, který má méně režijních operací. Manziniho a Ferraginovo vylepšení se projeví zejména na speciálních datech v kapitole 6.5.3. Mezi reálnými daty byl algoritmus *mf* výrazně lepší jen na spustitelném souboru (*bin*) původní velikosti.

Zajímavé je chování algoritmů na japonském a čínském textu. Přestože algoritmy *s* a *mf* u těchto souborů zpracovávají vlastně dvojnásobně dlouhý vstup oproti ostatním algoritmům, pro delší úseky dopadají nejlépe.

Nejrychlejší algoritmy pro jednotlivé vstupy jsou zachyceny na obrázcích 6.12 a 6.31. V těchto tabulkách jsou jen soubory od velikosti 2^{14} do plné velikosti, na kratších vstupech nebyl vždy jednoznačný vítěz.

Obrázek 6.12: Nejrychlejší algoritmy pro jednotlivé vstupy. Na y-ové ose jsou vstupní soubory (sežazené tak, aby oblasti, kde byl nejrychlejší jeden algoritmus byly co nejsouvislejší), v závorce je uvedeno $\overline{lcp}$ těchto souborů, na x-ové ose jsou velikosti vstupních souborů.



Obrázek 6.13: Nejrychlejší algoritmy pro japonský a čínský text.

Rychlosti algoritmů pro původní soubory (soubory plné velikosti) jsou shrnuty v tabulce 6.14. Tento seznam je jen orientační vzhledem k tomu, že vstupní soubory mají různou velikost.

	bin	bmp	csource	DNA	text-cz	text-en	xml	zip	text-jp	text-ch
merge	32.393	3.815	11.322	9.422	5.593	7.376	10.771	4.406	1.402	2.107
quick	56.883	4.765	16.110	13.642	8.078	10.591	15.398	6.273	1.957	2.978
heap	66.367	14.121	24.411	27.010	17.432	21.838	26.535	15.680	4.627	6.660
shell	99.124	18.483	69.519	73.894	34.364	43.002	57.945	30.363	8.511	10.734
mm	31.667	14.504	35.483	47.817	24.869	33.194	41.481	11.884	6.303	12.723
ks	9.202	8.002	11.063	15.877	10.998	12.701	13.194	6.732	3.270	5.034
s	3.703	1.120	4.474	4.009	2.128	2.930	**4.578**	1.846	1.330	**1.559**
mf	**1.347**	**1.045**	5.823	**3.258**	**2.047**	**2.594**	4.760	**1.822**	**1.168**	1.841
tree	21.996	17.511	**3.861**	8.824	8.460	7.652	5.981	121.732	56.319	149.271

Obrázek 6.14: Doby výpočtu sufixového pole pro reálná vstupní data původní velikosti.

6.5.2 Doby výpočtu nad náhodnými daty

Dále uvádím naměřené rychlosti algoritmů pro náhodná data. Tabulky 6.15 až 6.30 obsahují doby běhu (v sekundách) všech algoritmů pro soubory různých velikostí vždy nad abecedou jedné velikosti.

Každé z čísel v tabulkách je vždy průměr z naměřených hodnot pro čtyři náhodně vygenerované posloupnosti dané délky a velikosti abecedy.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.012	0.031	0.068	0.157	0.353	0.823	1.941	4.548
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.007	0.016	0.038	0.085	0.190	0.436	1.122	2.729	6.453
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.016	0.037	0.089	0.228	0.665	1.857	4.698	11.239
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.004	0.011	0.029	0.073	0.165	0.433	1.168	3.503	9.853	29.640
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.010	0.058	0.266	0.759	1.668	3.542	7.328
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.009	0.024	0.108	0.334	0.851	1.916	4.088
s	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.007	0.016	0.038	0.092	0.256	0.647	1.563
mf	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.002**	**0.005**	**0.011**	**0.025**	**0.063**	**0.181**	**0.451**	**1.065**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.007	0.018	0.048	0.105	0.240	0.517	1.125	2.520

Obrázek 6.15: Doby výpočtu sufixového pole pro vstup **r2**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.009	0.023	0.050	0.114	0.257	0.616	1.469	3.491
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.011	0.027	0.061	0.138	0.321	0.844	2.093	5.052
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.013	0.029	0.072	0.189	0.574	1.645	4.208	10.165
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.008	0.020	0.054	0.127	0.315	0.862	2.683	7.384	21.283
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.002**	0.009	0.054	0.275	0.712	1.591	3.376	7.042
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.026	0.114	0.349	0.870	1.911	4.007
s	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.001	0.002	0.007	0.015	0.034	0.083	0.243	0.637	1.563
mf	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.002	**0.005**	**0.012**	**0.025**	**0.063**	**0.180**	**0.451**	**1.083**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.006	0.021	0.057	0.141	0.317	0.716	1.607	3.655

Obrázek 6.16: Doby výpočtu sufixového pole pro vstup **r4**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.020	0.044	0.100	0.225	0.534	1.274	3.093
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.010	0.023	0.053	0.117	0.273	0.743	1.864	4.520
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.011	0.026	0.064	0.173	0.538	1.565	4.015	9.702
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.018	0.043	0.109	0.261	0.755	2.307	6.568	18.250
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.010	0.057	0.219	0.578	1.315	2.779	5.763
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.001**	0.007	0.027	0.118	0.361	0.898	2.015	4.263
s	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.001	0.005	0.012	0.030	0.074	0.209	0.562	1.412
mf	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.002	**0.005**	**0.011**	**0.025**	**0.061**	**0.170**	**0.428**	**1.019**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.006	0.023	0.068	0.185	0.447	1.029	2.353	5.395

Obrázek 6.17: Doby výpočtu sufixového pole pro vstup **r8**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.007	0.017	0.040	0.090	0.203	0.489	1.183	2.864
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.009	0.021	0.046	0.105	0.250	0.679	1.707	4.208
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.011	0.024	0.059	0.165	0.522	1.519	3.915	9.499
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.006	0.015	0.037	0.093	0.235	0.670	2.158	6.112	17.131
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.007	0.039	0.247	0.647	1.462	3.122	6.512
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.020	0.084	0.238	0.759	1.890	4.417
s	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.001	**0.004**	**0.010**	0.025	0.064	0.180	0.447	1.156
mf	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.002	0.005	0.010	**0.024**	**0.060**	**0.159**	**0.382**	**0.953**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.006	0.028	0.091	0.264	0.657	1.515	3.604	8.554

Obrázek 6.18: Doby výpočtu sufixového pole pro vstup **r16**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.002	0.007	0.017	0.037	0.085	0.193	0.464	1.130	2.735
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.008	0.019	0.044	0.099	0.235	0.640	1.636	3.994
heap	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.010	0.023	0.058	0.158	0.506	1.482	3.853	9.340
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.014	0.035	0.090	0.225	0.625	2.034	5.825	16.710
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.001**	0.007	0.040	0.178	0.481	1.096	2.587	6.727
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.007	0.025	0.098	0.278	0.681	1.489	3.143
s	<0.001	<0.001	<0.001	<0.001	0.001	<0.001	0.001	0.002	**0.003**	**0.009**	**0.022**	**0.056**	0.163	0.406	0.959
mf	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.002	0.005	0.011	0.024	0.057	**0.153**	**0.371**	**0.854**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.008	0.035	0.133	0.407	1.035	2.458	5.968	14.470

Obrázek 6.19: Doby výpočtu sufixového pole pro vstup **r32**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.016	0.035	0.080	0.184	0.444	1.087	2.660
quick	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.002	0.007	0.018	0.043	0.094	0.225	0.615	1.565	3.880
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.009	0.022	0.054	0.155	0.492	1.466	3.798	9.187
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.013	0.034	0.082	0.215	0.610	1.985	5.797	15.973
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.008	0.047	0.190	0.509	1.158	2.474	5.123
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.002**	0.007	0.031	0.117	0.337	0.831	1.816	3.825
s	<0.001	<0.001	<0.001	<0.001	0.001	0.001	0.001	0.002	**0.004**	**0.010**	**0.021**	**0.053**	**0.153**	0.390	0.930
mf	<0.001	<0.001	<0.001	0.001	0.001	0.001	0.002	0.003	0.007	0.012	0.026	0.061	0.158	**0.378**	**0.852**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.013	0.045	0.149	0.488	1.475	4.048	10.250	23.834

Obrázek 6.20: Doby výpočtu sufixového pole pro vstup **r64**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.015	0.034	0.078	0.176	0.434	1.066	2.569
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.007	0.018	0.040	0.092	0.218	0.595	1.511	3.740
heap	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.009	0.022	0.053	0.151	0.486	1.440	3.754	9.112
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.013	0.033	0.080	0.203	0.566	1.946	5.551	15.632
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.001**	0.008	0.049	0.204	0.544	1.245	2.648	5.494
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	**0.006**	0.033	0.121	0.348	0.854	1.899	4.065
s	0.001	0.001	0.001	0.001	0.002	0.003	0.004	0.005	0.007	**0.013**	**0.029**	**0.069**	**0.168**	**0.431**	1.002
mf	<0.001	<0.001	0.001	0.001	0.002	0.003	0.004	0.006	0.009	0.017	0.035	0.079	0.178	0.432	**0.974**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.021	0.114	0.440	1.238	2.995	7.023	17.107	44.054

Obrázek 6.21: Doby výpočtu sufixového pole pro vstup **r128**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.014	0.033	0.075	0.173	0.423	1.051	2.538
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.007	0.017	0.039	0.088	0.212	0.580	1.465	3.621
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.009	0.021	0.051	0.147	0.479	1.429	3.707	9.045
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.013	0.032	0.077	0.197	0.568	1.873	5.573	15.467
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**<0.001**	**0.004**	0.035	0.190	0.552	1.268	2.724	5.684
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.032	0.119	0.346	0.856	1.909	4.135
s	0.001	0.001	0.003	0.002	0.004	0.005	0.007	0.010	0.015	**0.026**	**0.049**	**0.097**	**0.209**	**0.464**	1.124
mf	0.001	0.001	0.002	0.002	0.003	0.005	0.008	0.012	0.017	0.030	0.056	0.111	0.224	0.481	**1.117**
tree	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.025	0.117	0.531	1.975	5.868	14.458	32.790	72.288

Obrázek 6.22: Doby výpočtu sufixového pole pro vstup **r256**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.005	0.013	**0.032**	0.075	0.185	0.469	1.159	2.790
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.015	0.036	0.089	0.252	0.639	1.579	3.718
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.008	0.020	0.054	0.192	0.598	1.616	3.993	9.466
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.012	0.030	0.078	0.222	0.745	2.237	6.292	16.557
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**<0.001**	**0.004**	0.034	0.140	0.507	1.182	2.741	5.848
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.035	0.132	0.381	0.899	1.979	4.245
s	0.002	0.003	0.004	0.007	0.007	0.008	0.009	0.012	0.018	0.033	**0.067**	**0.160**	0.366	0.897	2.273
mf	0.001	0.002	0.004	0.006	0.008	0.009	0.011	0.014	0.020	0.035	0.071	0.161	**0.352**	**0.842**	**2.047**
tree	<0.001	<0.001	<0.001	<0.001	0.001	0.006	0.015	0.037	0.119	0.460	1.784	6.526	21.732	66.815	172.853

Obrázek 6.23: Doby výpočtu sufixového pole pro vstup **r512**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.013	0.031	0.074	0.182	0.461	1.147	2.760
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.015	0.035	0.087	0.245	0.620	1.513	3.582
heap	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.020	0.054	0.189	0.595	1.609	3.973	9.412
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.012	0.030	0.073	0.213	0.746	2.233	6.093	16.846
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.004	0.034	0.139	0.412	0.872	2.119	5.840
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.002**	**0.024**	0.136	0.381	0.899	1.972	4.251
s	0.002	0.002	0.003	0.005	0.007	0.008	0.009	0.012	0.018	0.032	**0.064**	**0.152**	0.351	0.868	2.152
mf	0.001	0.002	0.004	0.006	0.007	0.009	0.010	0.014	0.021	0.034	0.068	0.153	**0.343**	**0.840**	**2.014**
tree	<0.001	<0.001	<0.001	<0.001	0.002	0.013	0.041	0.107	0.258	0.648	1.828	5.725	19.168	73.339	269.173

Obrázek 6.24: Doby výpočtu sufixového pole pro vstup **r1024**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.005	0.014	0.031	0.073	0.183	0.461	1.135	2.732
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.005	0.015	0.034	0.087	0.244	0.620	1.509	3.525
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.002	0.008	0.019	0.054	0.188	0.595	1.597	3.963	9.370
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.011	0.029	0.073	0.217	0.733	2.149	6.015	16.433
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.036	0.143	0.381	0.874	1.889	4.076
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**<0.001**	**0.002**	**0.014**	0.097	0.389	0.916	1.994	4.285
s	0.001	0.002	0.003	0.005	0.006	0.008	0.009	0.012	0.017	0.031	**0.066**	**0.153**	0.347	0.851	2.157
mf	0.001	0.002	0.003	0.005	0.007	0.009	0.011	0.013	0.020	0.035	0.069	0.154	**0.339**	**0.825**	**2.039**
tree	<0.001	<0.001	<0.001	<0.001	0.002	0.021	0.102	0.306	0.739	1.630	3.674	8.632	21.997	68.172	251.035

Obrázek 6.25: Doby výpočtu sufixového pole pro vstup **r2048**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.013	0.031	0.074	0.181	0.454	1.133	2.718
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.014	0.034	0.084	0.240	0.619	1.495	3.468
heap	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.008	0.019	0.053	0.189	0.594	1.593	3.946	9.360
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.011	0.029	0.074	0.211	0.726	2.130	6.086	16.078
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.005	0.039	0.153	0.400	0.909	1.931	4.105
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**0.002**	**0.013**	**0.060**	**0.157**	0.797	2.037	4.375
s	0.001	0.001	0.003	0.003	0.006	0.008	0.009	0.012	0.018	0.033	0.072	0.162	0.366	0.885	2.218
mf	0.001	0.002	0.003	0.004	0.006	0.009	0.010	0.014	0.021	0.038	0.076	0.166	**0.361**	**0.858**	**2.116**
tree	<0.001	<0.001	<0.001	<0.001	0.003	0.025	0.156	0.679	2.240	5.723	13.507	32.152	65.830	145.585	–

Obrázek 6.26: Doby výpočtu sufixového pole pro vstup **r4096**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.013	0.030	0.071	0.181	0.452	1.126	2.709
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.006	0.015	0.035	0.085	0.242	0.617	1.479	3.470
heap	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.019	0.053	0.185	0.586	1.581	3.949	9.324
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.011	0.029	0.071	0.210	0.721	2.109	6.076	16.340
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.007	0.041	0.162	0.421	0.952	2.015	4.255
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	**<0.001**	**0.003**	**0.016**	**0.066**	**0.173**	0.393	2.095	3.818
s	0.001	0.002	0.003	0.004	0.005	0.007	0.009	0.012	0.017	0.035	0.076	0.175	0.394	0.934	2.277
mf	0.001	0.001	0.003	0.004	0.004	0.008	0.010	0.014	0.021	0.039	0.085	0.181	**0.392**	**0.908**	**2.170**
tree	<0.001	<0.001	<0.001	<0.001	0.002	0.023	0.174	1.905	15.818	54.549	132.975	–	–	–	–

Obrázek 6.27: Doby výpočtu sufixového pole pro vstup **r8192**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.005	0.012	0.030	0.072	**0.180**	0.450	1.118	2.694
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.005	0.015	0.034	0.085	0.241	0.616	1.498	3.452
heap	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.007	0.019	0.052	0.185	0.580	1.578	3.932	9.292
shell	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.004	0.011	0.028	0.070	0.209	0.717	2.108	6.054	16.220
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<**0.001**	0.005	0.043	0.166	0.434	0.987	2.085	4.396
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	**0.004**	**0.019**	**0.071**	0.189	0.435	1.558	**1.980**
s	0.002	0.001	0.004	0.003	0.005	0.006	0.008	0.012	0.019	0.037	0.082	0.188	**0.422**	0.989	2.357
mf	0.001	0.001	0.003	0.004	0.005	0.007	0.010	0.014	0.022	0.044	0.091	0.199	0.425	**0.975**	2.265
tree	<0.001	<0.001	<0.001	<0.001	0.002	0.022	0.175	2.578	27.860	119.534	–	–	–	–	–

Obrázek 6.28: Doby výpočtu sufixového pole pro vstup **r16384**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.012	0.029	**0.071**	**0.177**	**0.446**	1.117	2.677
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.005	0.014	0.033	0.084	0.236	0.610	1.487	3.411
heap	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.019	0.052	0.182	0.582	1.575	3.922	9.289
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.011	0.028	0.070	0.211	0.726	2.121	6.018	15.926
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<**0.001**	**0.005**	0.044	0.171	0.446	1.016	2.157	4.540
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.006	**0.026**	0.089	0.219	0.493	**1.058**	**2.238**
s	0.002	0.002	0.003	0.003	0.004	0.006	0.008	0.013	0.022	0.043	0.091	0.201	0.457	1.070	2.464
mf	0.001	0.002	0.002	0.003	0.005	0.008	0.010	0.015	0.026	0.050	0.102	0.214	0.466	1.064	2.401
tree	<0.001	<0.001	<0.001	<0.001	0.002	0.020	0.163	2.386	31.654	172.821	–	–	–	–	–

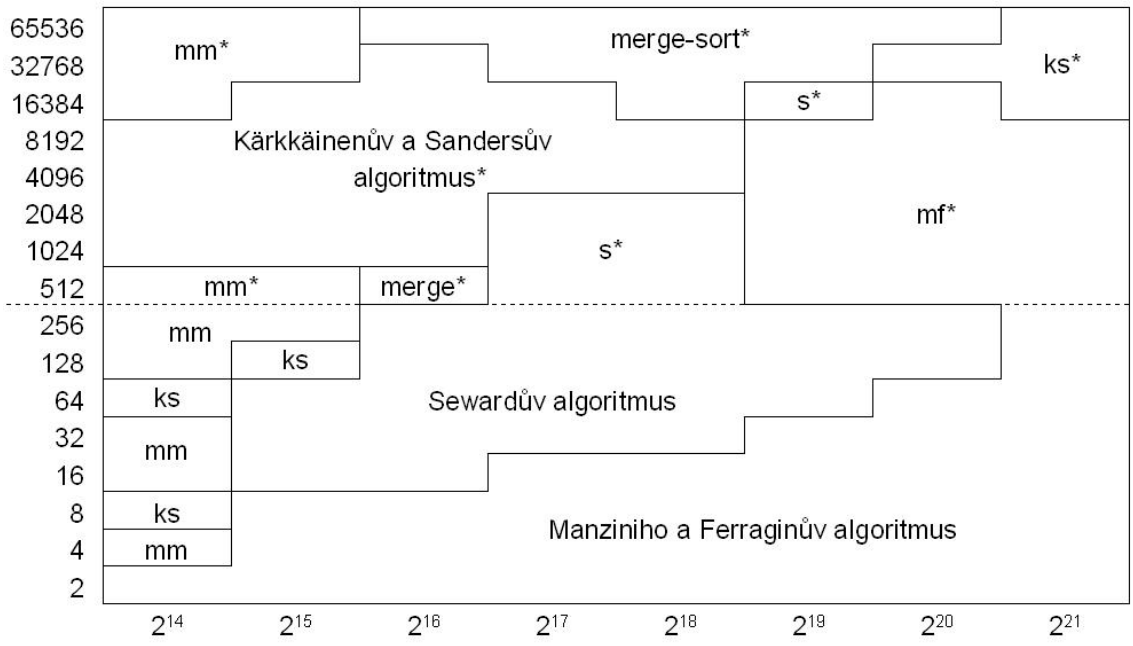
Obrázek 6.29: Doby výpočtu sufixového pole pro vstup **r32768**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.005	0.012	**0.030**	**0.069**	**0.174**	**0.442**	**1.099**	2.657
quick	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.005	0.014	0.033	0.081	0.235	0.607	1.475	3.411
heap	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.008	0.018	0.052	0.182	0.577	1.568	3.907	9.257
shell	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.011	0.028	0.070	0.196	0.720	2.104	5.902	15.780
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<**0.001**	**0.001**	**0.009**	0.037	0.179	0.458	1.046	2.224	4.709
ks	<0.001	<0.001	<0.001	<0.001	0.001	<0.001	0.002	0.005	0.014	0.039	0.107	0.254	0.561	1.194	**2.493**
s	0.001	0.002	0.002	0.004	0.005	0.008	0.011	0.017	0.028	0.051	0.101	0.213	0.478	1.144	2.575
mf	0.002	0.002	0.003	0.004	0.005	0.008	0.012	0.017	0.031	0.057	0.112	0.228	0.494	1.141	2.505
tree	<0.001	<0.001	<0.001	<0.001	0.002	0.020	0.160	2.168	31.172	195.956	–	–	–	–	–

Obrázek 6.30: Doby výpočtu sufixového pole pro vstup **r65536**.

Výsledky ukazují, že s náhodnými daty nemá žádný algoritmus s výjimkou sufixového stromu větší problémy.

Nejrychlejší algoritmy v každé kategorii shrnuje obrázek 6.31. Obecně se dá říct, že na krátké posloupnosti nad velkou abecedou je nejlepší Manberův a Myersův polu s Kärkkäinenovým a Sandersovým algoritmem, pro středně velké soubory a abecedy vyhrává Sewardův algoritmus a na dlouhé náhodné posloupnosti s malou abecedou je nejlepší Manziniho a Ferraginův algoritmus.



Obrázek 6.31: Nejrychlejší algoritmy pro jednotlivé náhodné vstupy. Na y-ové souřadnici je velikost abecedy a na x-ové velikost vstupního souboru v bytech. Šestnácti bitové verze algoritmů jsou označeny hvězdičkou.

U náhodných dat s kratšími abecedami se ukazuje, že Sewardův a Manziniho a Ferraginův algoritmus jsou nejrychlejší již pro poměrně krátké posloupnosti a přitom mezi těmito dvěma algoritmy není větší časový rozdíl. Ostatní algoritmy se s těmito dvěma mohou na krátkých posloupnostech rovnat, ale u delších výrazně ztrácejí. U větších abeced jsou rozdíly v rychlostech (s výjimkou sufixového stromu) výrazně menší.

Nejvýraznější je ztráta u sufixového stromu. A čím je abeceda větší, tím je tento postup pomalejší. Problém je zřejmě v implementaci stromu, kdy pro velkou abecedu je potřeba procházet dlouhý seznam potomků uzlů.

U Manberova a Myersova i u Kärkkäinenova a Sandersova algoritmu je patrná jen poměrně malá závislost rychlosti na velikosti abecedy.

Standardní třídící algoritmy běží na náhodných datech poměrně rychle, je to díky tomu, že tyto posloupnosti mají velmi malé $\overline{lcp}$. Čím je abeceda větší, tím jsou tyto algoritmy rychlejší. Nejrychlejší z nich je mergesort, jeho rychlost je u tohoto typu dat srovnatelná s nejrychlejšími.

Doby běhu nad náhodnými soubory velikosti 2^{21} znaků jsou shrnuty v tabulkách 6.32 a 6.33.

	r2	r4	r8	r16	r32	r64	r128	r256
merge	4.548	3.491	3.093	2.864	2.735	2.660	2.569	2.538
quick	6.453	5.052	4.520	4.208	3.994	3.880	3.740	3.621
heap	11.239	10.165	9.702	9.499	9.340	9.187	9.112	9.045
shell	29.640	21.283	18.250	17.131	16.710	15.973	15.632	15.467
mm	7.328	7.042	5.763	6.512	6.727	5.123	5.494	5.684
ks	4.088	4.007	4.263	4.417	3.143	3.825	4.065	4.135
s	1.563	1.563	1.412	1.156	0.959	0.930	1.002	1.124
mf	**1.065**	**1.083**	**1.019**	**0.953**	**0.854**	**0.852**	**0.974**	**1.117**
tree	2.520	3.655	5.395	8.554	14.470	23.834	44.054	72.288

Obrázek 6.32: Doby výpočtu sufixového pole pro náhodná vstupní data délky 2^{21} znaků.

	r512	r1024	r2048	r4096	r8192	r16384	r32768	r65536
merge	2.790	2.760	2.732	2.718	2.709	2.694	2.677	2.657
quick	3.718	3.582	3.525	3.468	3.470	3.452	3.411	3.411
heap	9.466	9.412	9.370	9.360	9.324	9.292	9.289	9.257
shell	16.557	16.846	16.433	16.078	16.340	16.220	15.926	15.780
mm	5.848	5.840	4.076	4.105	4.255	4.396	4.540	4.709
ks	4.245	4.251	4.285	4.375	3.818	**1.980**	**2.238**	**2.493**
s	2.273	2.152	2.157	2.218	2.277	2.357	2.464	2.575
mf	**2.047**	**2.014**	**2.039**	**2.116**	**2.170**	2.265	2.401	2.505
tree	172.853	269.173	251.035	–	–	–	–	–

Obrázek 6.33: Doby výpočtu sufixového pole pro náhodná vstupní data délky 2^{21} znaků.

6.5.3 Doby výpočtu nad speciálními daty

Následují naměřené rychlosti algoritmů pro speciální data. V tabulkách 6.34 až 6.37 jsou doby běhu (v sekundách) všech algoritmů vždy pro jednu speciální posloupnost o různých délkách.

Každá hodnota v tabulce je průměr ze čtyř měření nad stejnými daty. U některých algoritmů nejsou uvedeny hodnoty pro delší vstupní posloupnosti, je to z důvodu, že tyto algoritmy běžely na tomto typu dat příliš dlouho.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	0.001	0.010	0.046	0.205	0.894	3.883	16.724	71.689	–	–	–	–	–
quick	<0.001	<0.001	0.003	0.018	0.084	0.374	1.644	7.170	31.050	133.827	–	–	–	–	–
heap	<0.001	<0.001	0.002	0.017	0.078	0.346	1.521	6.585	28.395	122.301	–	–	–	–	–
shell	<0.001	<0.001	0.003	0.018	0.086	0.362	1.541	8.093	32.465	134.654	–	–	–	–	–
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.010	0.024	0.081	0.175	0.376	0.790	1.669	3.520
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.009	0.024	0.055	0.124	0.272	0.571	1.173
s	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.008	0.017	0.033	0.068
mf	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.003	0.007	0.014	0.027	0.053	0.104
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.006	0.011	0.022	0.053	0.100	0.204	0.418

Obrázek 6.34: Doby výpočtu sufixového pole pro vstup **aaaa**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	0.001	0.008	0.042	0.187	0.822	3.588	15.539	66.957	–	–	–	–	–
quick	<0.001	<0.001	0.003	0.020	0.090	0.403	1.781	7.776	33.901	146.731	–	–	–	–	–
heap	<0.001	<0.001	0.003	0.015	0.073	0.323	1.423	6.184	26.827	115.718	–	–	–	–	–
shell	<0.001	<0.001	0.004	0.021	0.100	0.443	1.944	8.779	36.697	159.421	–	–	–	–	–
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.009	0.022	0.105	0.243	0.516	1.100	2.304	4.895
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.009	0.024	0.064	0.147	0.329	0.682	1.424
s	<0.001	<0.001	0.001	0.007	0.034	0.151	0.671	2.945	12.829	55.585	–	–	–	–	–
mf	<0.001	<0.001	<0.001	0.002	0.008	0.038	0.124	0.485	1.859	7.255	27.126	–	–	–	–
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.007	0.016	0.037	0.070	0.144	0.297	0.599

Obrázek 6.35: Doby výpočtu sufixového pole pro vstup **abab**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	0.001	0.008	0.040	0.178	0.786	3.439	14.939	64.584	–	–	–	–	–
quick	<0.001	<0.001	0.003	0.020	0.089	0.402	1.760	7.716	33.137	149.987	–	–	–	–	–
heap	<0.001	<0.001	0.003	0.015	0.069	0.309	1.362	5.961	25.899	112.024	–	–	–	–	–
shell	<0.001	<0.001	0.002	0.015	0.069	0.312	1.409	5.909	26.172	116.542	–	–	–	–	–
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.009	0.024	0.118	0.299	0.642	1.355	2.847	6.053
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.008	0.020	0.052	0.116	0.251	0.534	1.086
s	<0.001	<0.001	<0.001	0.004	0.021	0.095	0.421	1.860	8.143	35.415	–	–	–	–	–
mf	<0.001	<0.001	<0.001	0.001	0.005	0.020	0.081	0.331	1.319	4.874	18.502	–	–	–	–
tree	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.002	0.008	0.020	0.044	0.091	0.185	0.372	0.754

Obrázek 6.36: Doby výpočtu sufixového pole pro vstup **abca**.

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
merge	<0.001	<0.001	<0.001	<0.001	0.010	0.059	0.314	1.562	7.469	34.856	–	–	–	–	–
quick	<0.001	<0.001	<0.001	0.002	0.021	0.129	0.689	3.518	16.901	79.923	–	–	–	–	–
heap	<0.001	<0.001	<0.001	0.001	0.014	0.091	0.500	2.532	12.403	59.041	–	–	–	–	–
shell	<0.001	<0.001	<0.001	0.002	0.024	0.139	0.749	4.345	23.371	114.707	–	–	–	–	–
mm	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.003	0.009	0.046	0.168	1.421	3.093	6.836	14.740	31.213
ks	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.003	0.009	0.028	0.136	0.420	1.082	2.386	5.051
s	0.001	0.003	0.003	0.003	0.003	0.003	0.004	0.013	0.053	0.237	1.098	4.985	22.587	100.763	445.155
mf	0.002	0.002	0.003	0.002	0.002	0.003	0.003	0.007	0.021	0.069	0.266	1.032	4.005	14.330	58.117
tree	<0.001	<0.001	<0.001	<0.001	<0.001	0.001	0.004	0.009	0.021	0.045	0.092	0.248	0.518	1.058	2.122

Obrázek 6.37: Doby výpočtu sufixového pole pro vstup **sigma**.

Tato vstupní data měla odhalit nedostatky jinak velmi rychlého Sewardova algoritmu a ukázat, jak tento problém řeší Manziniho a Ferraginovo vylepšení algoritmu.

V případě posloupnosti ze samých *áček* k problému nedošlo, oba algoritmy taková data setřídí induktivně. Pro posloupnost, kde se opakuje k znaků stále dokola, oba algoritmy sufixy rozdělí do k 1-skupin, a vždy jednu 1-skupinu (tedy n/k sufixů) musí setřídí explicitně.

To vede k velmi dlouhým dobám běhu zejména u posloupnosti *abab*. Algoritmy *s* a *mf* jsou pro tento typ vstupních dat výrazně pomalejší než ostatní (pominu-li standardní třídící

algoritmy) již pro posloupnosti délky 2^{10} bytů. Ukazuje se, že Manziniho a Ferraginova třídící procedura je v případě *abab* sedm a půlkrát rychlejší než běžné třídění u Sewardova algoritmu, přesto se ale nemůže měřit s ostatními algoritmy.

Tento problém se s rostoucí délkou posloupnosti, která se v souboru opakuje, zmenšuje. U souboru *sigma* do délky 2^{16} bytů už není skoro patrný.

Všechny tyto speciální posloupnosti jsou nevýhodné i pro standardní třídící algoritmy, které s nimi mají problém kvůli jejich velkému $\overline{lcp}$.

6.5.4 Paměťové nároky

Paměťové nároky jednotlivých algoritmů relativně vzhledem k délce vstupu (memory-peak/n) jsou uvedeny v tabulkách 6.38, 6.39, 6.40 a 6.41.

V tabulkách 6.38, 6.39 a 6.40 jsou hodnoty vždy pro největší soubor ze skupiny. Výjimku tvoří algoritmus *tree*, pro který jsou v tabulce 6.40 uvedeny údaje pro vstup délky 2^{16} znaků. V tabulce 6.41 jsou všechny hodnoty naměřeny pro posloupnosti délky 2^{16} znaků.

Tato relativní spotřeba paměti je samozřejmě pro kratší úseky souborů výrazně větší, jak je vidět v tabulce 6.41, ve které jsou naměřené hodnoty pro vstupní soubory délky 2^{16} bytů. Velikost pomocných datových struktur algoritmů *s* a *mf* je u takto krátkých souborů stejná jako paměť pro uložení x a SA.

V tabulce 6.40 je uvedena spotřeba paměti pro 16ti bitové verze algoritmů. U všech algoritmů s výjimkou *s* a *mf* jsou tyto hodnoty vždy o jedničku větší než u jejich osmi bitové verze. To proto, že vstupní řetězec v tomto případě zabírá $2n$ bytů paměti. Naproti tomu u algoritmů *s* a *mf* je množství potřebné paměti dvojnásobné z důvodu, že používají sufixové pole dvojnásobné délky.

Větší závislost spotřebované paměti na typu vstupních dat se ukazuje jen u sufixového stromu. Hodnoty v tabulce nemají jednoznačnou souvislost s velikostí stromu v průběhu výpočtu. Jsou v nich totiž započteny i uzly, které vzniknou při závěrečné expanzi stavů odpovídajících sufixům vstupního řetězce. (Například sufixový strom pro vstup *aaaa* má v průběhu výpočtu pouze stavy $\perp$ a *koren*, ze kterého vede jedna otevřená hrana, přesto nakonec algoritmus pro tento vstup potřebuje $25,6n$ bytů paměti.)

Spotřebu paměti by u sufixového stromu bylo možné snížit. Vzhledem k tomu že má ale výrazně větší paměťové nároky než ostatní algoritmy, nesnažil jsme se ho po téhle stránce maximálně optimalizovat.

	bin	bmp	csource	DNA	text-cz	text-en	xml	zip
merge	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00
quick	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
heap	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
shell	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
mm	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00
ks	11.67	11.67	11.67	11.67	11.67	11.67	11.67	11.67
s	5.17	5.14	5.14	5.11	5.16	5.13	5.13	5.16
mf	5.24	5.18	5.20	5.17	5.21	5.18	5.18	5.20
tree	33.13	34.38	39.38	38.13	35.63	36.25	36.25	27.50

Obrázek 6.38: Maximální množství najednou použité paměti pro jednotlivé reálné soubory relativně vzhledem k n.

	r2	r4	r8	r16	r32	r64	r128	r256
merge	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00
quick	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
heap	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
shell	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
mm	9.00	9.00	9.00	9.00	9.00	9.00	9.00	9.00
ks	11.67	11.67	11.67	11.67	11.67	11.67	11.67	11.67
s	5.25	5.25	5.25	5.25	5.25	5.25	5.25	5.25
mf	5.30	5.30	5.29	5.29	5.29	5.29	5.29	5.29
tree	45.00	37.50	34.38	33.13	31.88	28.75	30.63	26.88

Obrázek 6.39: Maximální množství najednou použité paměti pro jednotlivé náhodné soubory nad osmi bitovou abecedou relativně vzhledem k n.

	r512	r1024	r2048	r4096	r8192	r16384	r32768	r65536	text-jp	text-ch
merge	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00
quick	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00
heap	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00
shell	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00	6.00
mm	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00
ks	12.67	12.67	12.67	12.67	12.67	12.67	12.67	12.67	12.67	12.67
s	10.25	10.25	10.25	10.25	10.25	10.25	10.25	10.25	10.45	10.35
mf	10.34	10.33	10.33	10.33	10.33	10.33	10.33	10.33	10.55	10.47
tree	28.52	27.27	27.27	27.90	29.15	31.02	32.27	31.65	32.88	33.50

Obrázek 6.40: Maximální množství najednou použité paměti pro jednotlivé 16ti bitové soubory relativně vzhledem k n.

	aaaa	abab	abca	sigma
merge	9.03	9.03	9.03	9.03
quick	5.02	5.02	5.02	5.02
heap	5.02	5.02	5.02	5.02
shell	5.02	5.02	5.04	5.02
mm	9.01	9.01	9.01	9.01
ks	11.68	11.68	11.68	11.68
s	13.09	13.11	13.11	13.10
mf	13.13	14.12	14.12	14.11
tree	25.64	25.64	25.64	25.64

Obrázek 6.41: Maximální množství najednou použité paměti pro jednotlivé soubory relativně vzhledem k n.

Pomineme-li sufixový strom, je paměťově nejnáročnější Kärkkäinenův a Sandersův algoritmus, potřebuje více než dvakrat větší množství paměti oproti Sewardově a Manziniho a Ferraginově algoritmu. Manberův a Myersův algoritmus s $9n$ byty paměti patří spíše k paměťově náročnějším. Ze standartních třídících algoritmů má mergesort téměř dvakrát větší paměťové nároky než ostatní.

6.5.5 Rozdíly při použití různých překladačů

Všechny výše uvedené doby výpočtu jsem naměřil pro programy v jazyce C přeložené překladačem gcc s optimalizací -O3. Pro srovnání jsem algoritmy přeložil i v jazyce C++, překladačem g++ bez optimalizací. Rychlosti výpočtu sufixového pole se v obou případech výrazně lišily. Pro srovnání uvádím doby výpočtů obou verzí algoritmů nad soubory maximální délky (obrázky 6.42 až 6.44).

Rozdíly v rychlostech jsou různé v závislosti na použitém algoritmu i na typu vstupních dat. Nejvíce se zrychlení projevuje u standardních třídících algoritmů, ale také u algoritmů *s* a *mf*.

Zajímavý je například první sloupec v tabulkách na obrázku 6.42, kde všechny standardní algoritmy běží přibližně dvakrát rychleji při překladu pomocí gcc s optimalizací, naproti tomu u metod *mm*, *ks* a *tree* je rozdíl jen minimální. Rozdíl se zmenšuje také s klesajícím $\overline{lcp}$ vstupních dat, jak ukazují i tabulky s náhodnými posloupnostmi.

	bin	bmp	csource	DNA	text-cz	text-en	xml	zip	text-jp	text-ch
merge	62.872	5.053	20.571	12.510	7.076	9.705	16.407	4.993	1.879	3.043
quick	115.982	6.876	27.516	18.479	10.584	14.235	24.706	7.880	2.873	4.585
heap	130.248	18.515	35.597	33.827	21.643	27.392	36.200	19.039	5.925	8.671
shell	202.159	30.856	106.902	99.853	48.497	59.755	85.322	41.652	12.543	16.145
mm	36.682	17.406	40.146	53.512	28.180	37.404	46.260	13.194	7.155	14.359
ks	11.038	9.630	13.481	18.677	13.097	15.129	15.925	7.934	3.896	5.989
s	9.392	**1.671**	10.282	5.635	**3.017**	4.318	9.461	**2.117**	1.822	**2.289**
mf	**2.207**	1.711	10.369	**5.034**	3.128	**4.094**	8.654	2.275	**1.817**	2.811
tree	23.343	18.702	**4.650**	9.841	9.246	8.564	**6.779**	125.996	58.326	151.126

	bin	bmp	csource	DNA	text-cz	text-en	xml	zip	text-jp	text-ch
merge	32.393	3.815	11.322	9.422	5.593	7.376	10.771	4.406	1.402	2.107
quick	56.883	4.765	16.110	13.642	8.078	10.591	15.398	6.273	1.957	2.978
heap	66.367	14.121	24.411	27.010	17.432	21.838	26.535	15.680	4.627	6.660
shell	99.124	18.483	69.519	73.894	34.364	43.002	57.945	30.363	8.511	10.734
mm	31.667	14.504	35.483	47.817	24.869	33.194	41.481	11.884	6.303	12.723
ks	9.202	8.002	11.063	15.877	10.998	12.701	13.194	6.732	3.270	5.034
s	3.703	1.120	4.474	4.009	2.128	2.930	**4.578**	1.846	1.330	**1.559**
mf	**1.347**	**1.045**	5.823	**3.258**	**2.047**	**2.594**	4.760	**1.822**	**1.168**	1.841
tree	21.996	17.511	**3.861**	8.824	8.460	7.652	5.981	121.732	56.319	149.271

Obrázek 6.42: Doby výpočtu sufixového pole pro reálná vstupní data původní velikosti. Nahoře programy v jazyce C++ bez optimalizace, dole v jazyce C s optimalizací -O3.

	r2	r4	r8	r16	r32	r64	r128	r256
merge	6.818	4.716	3.960	3.511	3.294	3.115	2.995	2.907
quick	9.487	6.826	5.881	5.389	5.093	4.903	4.715	4.611
heap	15.139	12.894	12.144	11.788	11.563	11.408	11.281	11.181
shell	44.425	30.572	25.913	24.107	23.401	22.223	21.911	21.597
mm	8.343	7.951	6.595	7.330	7.718	5.849	6.188	6.376
ks	5.368	5.148	5.353	5.495	4.055	4.702	4.919	4.954
s	2.502	2.213	1.905	1.530	**1.247**	**1.164**	**1.181**	**1.302**
mf	**1.702**	**1.688**	**1.563**	**1.418**	1.248	1.210	1.280	1.394
tree	2.993	4.092	5.828	9.011	15.100	24.936	45.229	74.587

	r2	r4	r8	r16	r32	r64	r128	r256
merge	4.548	3.491	3.093	2.864	2.735	2.660	2.569	2.538
quick	6.453	5.052	4.520	4.208	3.994	3.880	3.740	3.621
heap	11.239	10.165	9.702	9.499	9.340	9.187	9.112	9.045
shell	29.640	21.283	18.250	17.131	16.710	15.973	15.632	15.467
mm	7.328	7.042	5.763	6.512	6.727	5.123	5.494	5.684
ks	4.088	4.007	4.263	4.417	3.143	3.825	4.065	4.135
s	1.563	1.563	1.412	1.156	0.959	0.930	1.002	1.124
mf	**1.065**	**1.083**	**1.019**	**0.953**	**0.854**	**0.852**	**0.974**	**1.117**
tree	2.520	3.655	5.395	8.554	14.470	23.834	44.054	72.288

Obrázek 6.43: Doby výpočtu sufixového pole pro náhodné posloupnosti délek 2^{21} bytů. Nahoře programy v jazyce C++ bez optimalizace, dole v jazyce C s optimalizací -O3.

	r512	r1024	r2048	r4096	r8192	r16384	r32768	r65536
merge	3.446	3.381	3.310	3.279	3.248	3.229	3.219	3.209
quick	5.259	5.160	5.082	5.033	5.009	4.965	4.934	4.894
heap	11.715	11.646	11.588	11.558	11.500	11.459	11.393	11.350
shell	23.356	23.826	23.230	22.650	23.033	22.939	22.490	22.221
mm	6.526	6.643	4.619	4.638	4.755	4.887	5.053	5.244
ks	5.055	5.021	5.044	5.123	4.500	**2.431**	**2.697**	2.936
s	**2.861**	**2.674**	**2.615**	**2.638**	**2.658**	2.728	2.832	**2.920**
mf	2.959	2.851	2.844	2.866	2.878	2.930	3.023	3.110
tree	180.499	284.643	286.892	–	–	–	–	–

	r512	r1024	r2048	r4096	r8192	r16384	r32768	r65536
merge	2.790	2.760	2.732	2.718	2.709	2.694	2.677	2.657
quick	3.718	3.582	3.525	3.468	3.470	3.452	3.411	3.411
heap	9.466	9.412	9.370	9.360	9.324	9.292	9.289	9.257
shell	16.557	16.846	16.433	16.078	16.340	16.220	15.926	15.780
mm	5.848	5.840	4.076	4.105	4.255	4.396	4.540	4.709
ks	4.245	4.251	4.285	4.375	3.818	**1.980**	**2.238**	**2.493**
s	2.273	2.152	2.157	2.218	2.277	2.357	2.464	2.575
mf	**2.047**	**2.014**	**2.039**	**2.116**	**2.170**	2.265	2.401	2.505
tree	172.853	269.173	251.035	–	–	–	–	–

Obrázek 6.44: Doby výpočtu sufixového pole pro náhodné posloupnosti délek 2^{21} znaků. Nahoře programy v jazyce C++ bez optimalizace, dole v jazyce C s optimalizací -O3.

Kapitola 7

Závěr

V této práci jsem popsal několik algoritmů na konstrukci sufixového pole spolu s jejich vlastnostmi a chováním na různých typech dat. Algoritmy jsem implementoval ve dvou verzích - pro osmi a šestnácti bitovou vstupní abecedu a spolu s běžnými metodami na třídění dat testoval na souborech běžně používaných v praxi a na náhodných posloupnostech.

Ukázalo se že na náhodně generovaná data lze pro konstrukci sufixového pole s úspěchem použít jednoduchý mergesort. Naopak na reálných datech z praxe jsou sofistikovanější algoritmy významně rychlejší. Obecně nejrychlejší jsou Manziniho a Ferraginův spolu se Sewardovým algoritmem, přestože jejich teoretická časová složitost v nejhorším případě je asymptoticky nejhorší. Pro tyto dva algoritmy lze ale snadno zkonstruovat vstupní posloupnost, na které selhávají. Naopak Kärkkäinenův a Sandersův algoritmus s asymptoticky lineární časovou složitostí je v průměru pomalý. Jeho výhodou je, že běží přibližně stejně dlouho pro jakýkoli vstup.

Z hlediska typu dat se ukázalo že Manberův a Myersův spolu s Kärkkäinenovým a Sandersovým algoritmem mají problémy se vstupy, ve kterých se vyskytují alespoň dva dlouhé stejné podřetězce, jako je tomu u řetězce DNA. Pro Manziniho a Ferraginův a pro Sewardův algoritmus je kritická délka průměrného společného prefixu vstupního souboru. Naopak algoritmu založeném na sufixovém stromě vstupy s velkým průměrným společným prefixem vyhovují.

Při pokusech s abecedou větší než 256 znaků se ukázalo, že takovýto typ vstupu není pro žádný ze studovaných algoritmů, s výjimkou metody založené na sufixovém stromu, problémem.

Paměťové nároky jednotlivých postupů se nezdají být významné při zpracování vstupů do velikosti řádově desítek mega bytů a pro větší data by konstrukce sufixového pole těmito algoritmy byla časově velmi náročná. Množství potřebné paměti by ale mohlo být důležité při výpočtu v mobilních zařízeních. Paměťově nejúspornější jsou algoritmy *s* a *mf* spolu s *quicksortem*, *heapsortem* a *shellsortem*, velké paměťové nároky má naopak sufixový strom.

V budoucnu by mohlo být zajímavé studovat možnost konstrukce sufixového pole pro velmi dlouhé vstupní soubory.

Dodatek A

Obsah přiloženého CD

Přiložené CD obsahuje tento text, zdrojové kódy všech algoritmů a všechna testovací data spolu s údaji o jejich $\overline{lcp}$.

Literatura

[1] Canterbury Corpus, Calgary Corpus, Large Corpus, http://corpus.canterbury.ac.nz/

[2] Český překlad bible, http://www.palmknihy.cz/

[3] factbook.xml,
http://www.w3.org/XML/Binary/2005/03/test-data/Over100K/factbook.xml

[4] Implementace heapsortu, http://ftp.osuosl.org/pub/gentoo/distfiles/openbsd-lib-3.8.tar.bz2, soubor openbsd-lib-3.8/lib/libc/stdlib/heapsort.c.

[5] Implementace quicksortu, http://gentoo.osuosl.org/distfiles/glibc-2.3.5.tar.bz2, soubor glibc-2.3.5/stdlib/qsort.c.

[6] Implementace shellsortu, http://c.snippets.org/code/rg_ssort.c.

[7] Texty v čínštině, http://ef.cdpa.nsysu.edu.tw/ccw/

[8] Texty v japonštině, http://etext.virginia.edu/japanese/

[9] Zdrojové kódy programu GIMP, http://www.gimp.org/source/

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

[11] Jon L. Bentley, M. Douglas McIlroy, Engineering a Sort Function. Software - Practice & Experience 23-11 (1993) 1249-1265.

[12] Jon L. Bentley, Robert Sedgewick, Fast algorithms for sorting and searching strings, Proc. ACM-SIAM Symp. Discrete Algs. (1997) 360-369.

[13] Stefan Burkhardt, Juha Kärkkäinen, Fast lightweight suffix array construction and checking. In Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03). LNCS 2676, Springer, 2003, pp. 55-69.

[14] M. Burrows & D. J. Wheeler, A Block-Sorting Lossless Data Compression Algorithm, Digital Systems Research Report 124 (1994).

[15] Juha Kärkkäinen and Peter Sanders, Simple linear work suffix array construction. Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03). LNCS 2719, Springer, (2003) 943-955.

[16] U.Manber, G.Myers, Suffix arrays: A new mathod for online string searches. Proc. First ACM-SIAM Symp. on Discrete Algs. (1990), 319-327.

[17] Giovanni Manzini, Paolo Ferragina, Engineering a lightweight suffix array construction algorithm, Algorithmica 40 (2004), 33-50.

[18] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings, Cambridge University Press, 2002.

[19] Simon J. Puglisi, W. F. Smyth, Andrew Turpin, A taxonomy of suffix array construction algorithms, Proceedings of the Prague Stringology Conference (PSC'05), Jan Holub (ed.), 2005, pp. 1-30.

[20] Julian Seward, On the Performance of BWT Sorting Algorithms, Proc. Data Compression Conf. (2000), 173-182.

[21] Esko Ukkonen, On-line construction of suffix trees, Algorithmica 14 (1995), 249-260.