Enhanced and Extended Suffix Arrays

Adrian Regenfuß

July 31, 2020

Abstract

In this report, I review the literature on enhanced and extended suffix arrays in the context of matching long strings. I examine the different algorithms used for both constructing enhanced and extended suffix arrays and for using them in searching long strings. In the end, I compare enhanced and extended suffix arrays with suffix arrays and suffix trees.

Introduction

Finding the occurences of one string in another string, longest repeated substrings and longest shared substrings of two different strings are fundamental problems for many kinds of computing systems.

As a result, many different algorithms have been developed for these kinds of problems: For finding the occurences of one string in another one the naive algorithm and the Boyer-Moore algorithm (Boyer and Moore 1977), and for all three of these problems (and more) three different data structures: the suffix tree (Weiner 1973), the suffix array (Manber and Myers 1993) and the enhanced suffix array (Abouelhoda et al. 2002).

Suffix trees, suffix arrays and enhanced suffix arrays have the disadvantage of requiring to be constructed for a specific string, which has time and space requirements. Because of this, they are better suited for tasks where immutable strings have to be searched or matched repeatedly, although there has been some work to extend the suffix array to dynamic strings (Salson et al. 2010).

Since searching and matching very long immutable strings is very common in genome analysis, it doesn't surprise that both suffix arrays and enhanced suffix arrays were developed in that context.

This report first describes the enhanced suffix array as a data structure, then sketches the algorithms used for constructing it, and afterwards describes different string matching problems and how they are solved by enhanced and extended suffix arrays. Finally, it compares enhanced suffix arrays to normal suffix arrays and suffix trees, and closes with an overview of tools that implement enhanced and extended suffix arrays.

Enhanced and Extended Suffix Arrays

Enhanced suffix arrays were first proposed in Abouelhoda et al. 2002 as an improvement over normal suffix arrays. An enhanced suffix array contains a suffix array together with the LCP-array of the string, and sometimes a Burrows-Wheeler transformation and an inverse of the suffix table.

The term "Extended Suffix Array" is rarely used in the literature (e.g. in Salson et al. 2010) and seems to refer to a datastructure combining the LCP array with the suffix array of a string. Since algorithms that work on extended suffix arrays also work on enhanced suffix arrays, I will focus on enhanced suffix arrays in this review.

For the following, let S be a finite string of length n over the finite alphabet Σ.

suftab

The suffix array suftab is an array of integers describing the positions of of sorted suffixes of S in S.

More formally, let Suf_S be the set of suffixes of S. Let then $\mathrm{SortSuf}_S$ be the array of lexically sorted suffixes of S. Then $\mathrm{suftab}[i] = k$ if and only if $S[k..n] = \mathrm{SortSuf}[i]$.

For a string S with $n = |S| < 2^{32}$, suftab usually uses $4n$ bytes.

lcptab

The LCP (longest common prefix) table describes the length of the longest common prefix of two neighbouring entries in the array of sorted suffixes.

Formally, $\mathrm{lcptab}[i] = k$ iff $\mathrm{SortSuf}[i][0..k] = \mathrm{SortSuf}[i-1][0..k]$. The zeroth entry in lcptab is always 0.

For a string S with $n = |S|$, suftab usually uses n bytes, assuming that the length of longest common prefixes of two suffixes are less than 255. If this is not the case, Abouelhoda et al. 2004, sec. 8.1 describes some practical workarounds based on a secondary array.

bwttab

Informally, bwttab contains the character before the suffix in suftab. It is derived from the Burrows-Wheeler transformation used in text compression. By containing bwttab, the enhanced suffix array contains a complete copy of the original string which can be reconstructed in linear time.

Formally, bwttab is defined as follows:

$$\mathrm{bwttab}[i] = \begin{cases} S[\mathrm{suftab}[i]-1] & \mathrm{if} \mathrm{suftab}[i] > 0 \\ \bot & \mathrm{otherwise} \end{cases} \tag{1}$$

In theory, the size of bwttab depends on the size of the alphabet Σ, but usually it is assumed that $|\Sigma| = 256$, one ASCII character. The space requirement of bwttab then is $|S| = n$ bytes.

suftab^{-1}

suftab^{-1} is the inverse of the suffix array: $\mathrm{suftab}^{-1}[\mathrm{suftab}[q]] = q$, or, in other words, when viewing suftab and suftab^{-1} as permutations, then the concatenation $\mathrm{suftab}^{-1} \circ \mathrm{suftab} = S_{id}$ (S_{id} is the identity permutation, also denoted by (1) or ()).

suftab^{-1} requires the same amount of space as suftab, $4n$ bytes.
The inverse suffix array is used in the tandem repeat finding algorithm.

Space requirements

The space requirement of an enhanced suffix array is $10n$ bytes, $4n$ for each the suffix array and the inverse, and n for burrows-wheeler transformation and the lcp array.

Example

For example, let $S =$ "cagccacat". Then suftab, lcptab, bwttab, suftab^{-1} and suftab are the following:

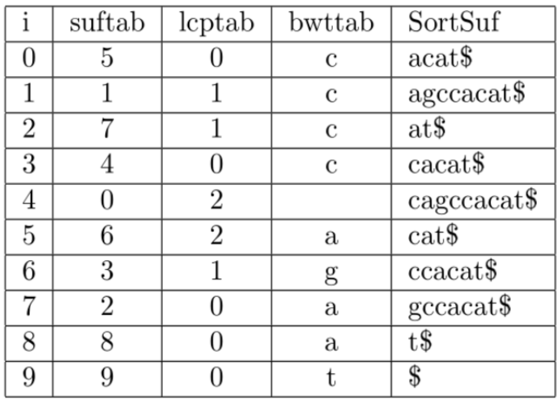
i	suftab	lcptab	bwttab	suftab^{-1}	SortSuf
0	5	0	c	4	acat$
1	1	1	c	1	agccacat$
2	7	1	c	7	at$
3	4	0	c	6	cacat$
4	0	2	$\perp$	3	cagccacat$
5	6	2	a	0	cat$
6	3	1	g	5	ccacat$
7	2	0	a	2	gccacat$
8	8	0	a	8	t$
9	9	0	t	9	$

LCP-Interval Trees

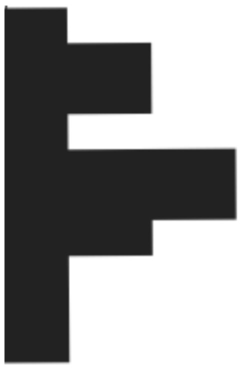
An LCP-Interval of value ℓ ($\ell - [i..j]$) is an interval $[i..j]$ ($i \leq j$) for which holds:

- $\text{lcptab}[i] < \ell$
- $\text{lcptab}[j+1] < \ell$
- $\exists k : i < k \leq j \wedge \text{lcptab}[k] = \ell$
- $\forall k : i < k \leq j \Rightarrow \text{lcptab}[k] \geq \ell$

One can visualize the lengths of the longest common prefixes as a landscape, and LCP-intervals being regions in that landscape that have a minimum height ℓ.



i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$



LCP-intervals can be embedded into each other, specifically, an $\hbar - [i..j]$ is embedded in $\ell - [k..l]$ iff $k \leq i \leq j \leq l$ and $\hbar < \ell$.

Due to this, the enhanced suffix array of a string implicitely contains a data structure called the LCP-interval tree. The root of the tree contains information on the 0-interval: $0 - [0..n]$ with $n = |S|$. The leaves are the 1-intervals, each containing the starting and ending position of the corresponding LCP interval.

The LCP-interval tree is just the suffix tree (Weiner 1973) without leaves, and its traversal enables linear time solutions to some string matching problems using the enhanced suffix array.

The LCP-interval tree is not saved in memory, but reconstructed by using the suffix array and the LCP array during execution (Abouelhoda et al. 2002).

Different String Matching Problems

A plethora of different string matching problems have been identified by computer scientists, for many of which suffix arrays and enhanced suffix arrays are useful.

Abouelhoda et al. 2004, pg. 2 summarizes suffix tree applications from Gusfield 1997, chap. 2 and classify them after their type of tree traversal.

Exact String Matching

Description

Give a string S of length n and a string T of length m with $m \leq n$ both using them same alphabet Σ, the exact matches of T in S is the set of indices $I = \{i_1, \ldots, i_k\}$ for which holds that $\forall i \in I : S[i..i + m] = T$.

In other words, I is the set of indices where T is a substring in S.

Suffix Array Algorithm

Manber and Myers 1993 describes an exact string matching algorithm that runs in $\mathcal{O}(m \log n)$ time and constant space.

Their proposed algorithm uses binary search to sequentially find the first (smallest) index L_W and the last (biggest) index R_W in suftab so that $S[\text{suftab}[L_W]]$ and $S[\text{suftab}[R_W]]$ have the prefix T.

First, it searches L_W using binary search on the whole array suftab and then uses L_W as a left boundary to find R_W, again by binary search. Using L_W as a left boundary improves runtime as opposed to two independent searches over the whole array, although the latter might be easier to parallelize.

Extended Suffix Array Algorithm

Manber and Myers 1993 then propose a speed improvement based on longest common prefixes that reduces the runtime to $\mathcal{O}(m + \log n)$. Their method attempts to reduce the number of single-character comparisons by only comparing characters that occur after the longest common prefix of T and $S[M_W]$ (M_W being the index in the middle between L_W and R_W).

Gusfield 1997, p. 152 describes another speed-up called the super-accelerant, which uses LCP-arrays to in practice reduce runtime even further. It doesn't improve worst-case time complexity.

I have not come across a proposal to use interpolation search first described in Perl et al. 1978 to search the suffix array with an improved $\mathcal{O}(\log \log n)$ runtime. This perhaps stems from the fact that interpolation search assumes uniform distribution of the alphabet, and has a worst-case runtime of $\mathcal{O}(n)$. It still might be useful to empirically test speed differences in binary and interpolation search.

Supermaximal and Maximal Repeats

Enhanced suffix arrays were first designed to solve problems in genome analysis, especially finding segmental duplications (Lander et al. 2001). Due to this, many algorithms have been devised for finding different kinds of repeated substrings in a string S.

Description

Two substrings $S_1 = S[i_1..j_1]$ and $S_2 = S[i_2..j_2]$ are called a repeated pair if $S_1 = S_2$ and $i_1 \neq i_2$ and $j_1 \neq j_2$. S_1 and S_2 are furthermore a maximal repeat iff $S[i_1 - 1] \neq S[i_2 - 1]$ and $S[j_1 + 1] \neq S[j_2 + 1]$. A supermaximal repeat is a maximal repeat that does not occur as a substring of another maximal repeat.

Let S_a and S_b be two distinct strings over Σ. Let $\# \notin \Sigma$ be a character. Then a maximum unique match (MUM) is a supermaximal repeat $((i_a, j_a)(i_b, j_b))$ of $S_a\#S_b$ so that $j_a < |S_b|$ and $i_b > |S_a|$.

For example, the string "xabyabwabyz" contains the maximal repeat "ab" (at positions $((1,2),(4,5))$ and $((4,5)(7,8)))$ and the maximal repeat "aby" at positions $((1,3),(7,9))$, as well as the supermaximal repeat "aby" as positions

$((1,3),(7,9))$. Note that the set of supermaximal repeats is a subset of the set of maximal repeats.

Enhanced Suffix Array Algorithm for Finding Supermaximal Repeats

Finding supermaximal repeats using an enhanced suffix array is comparatively simple; the process can be visualized as finding local maxima in lcptab with pairwise distinct values in bwttab.

```
maxstart ← 0
result ← $\emptyset$
**for** $i$ in $0..n-1$ **do**
    **if** lcptab[$i$] > lcptab[$i-1$] and $i > 0$ **then**
        maxstart ← $i$
        supmaxrep ← $true$
        preceding ← $\emptyset$
    **else if** lcptab[$i$] < lcptab[$i-1$] and supmaxrep **then**
        $\omega \leftarrow S[\text{suftab}[i-1]..\text{suftab}[i-1]+\text{lcptab}[i-1]]$
        result ← result $\cup$ {($\omega$, maxstart, $i-1$)}
        supmaxrep ← $false$
    **end if**
    **if** bwttab[$i$] $\in$ preceding **then**
        supmaxrep ← $false$
    **else**
        preceding ← preceding $\cup$ bwttab[$i$]
    **end if**
**end for**
```

This algorithm runs in $\mathcal{O}(n)$ time and is described first by Abouelhoda et al. 2002.

Finding Maximum Unique Matches

Finding MUMs is just a special case of finding supermaximal repeats:

- The enhanced suffix array of $S_a\#S_b$ is generated
- The algorithm for finding supermaximal repeats is executed
- The set of supermaximal repeats is scanned for instance where $j_a < |S_b|$ and $i_b > |S_a|$

This algorithm also runs in $\mathcal{O}(n)$ time ($n = |S_a\#S_b|$).

While this is theoretically good, in practice the construction of the enhanced suffix array for the two strings provides some hurdles. Especially in the case of sequence assembly (Myers et al. 2000), where the maximum unique matches of sometimes tens of thousands of reads have to be assembled, re-constructing the enhanced suffix arrray for each pair of reads can be computationally quite intensive. Salson et al. 2010 describes techniques for updating modified suffix arrays and LCP arrays, which could be a useful starting point for finding methods of combining enhanced suffix arrays of concatenated strings.

LCP-interval tree Traversal

Since the LCP-interval tree is simply the suffix tree without leaves, it is very useful to being able to construct and traverse the LCP-interval tree using the enhanced suffix array.

Abouelhoda et al. 2002 expand on an algorithm proposed in Kasai et al. 2001 that traverses the LCP-interval tree bottom up and calls a function *process* for each node in the tree.

The information passed to *process* consists of a tuple $(\ell, lb, rb, children)$, where lb is the left boundary of the ℓ-interval, rb is the right boundary, and $children$ is a list containing the child intervals of the given ℓ-interval (possibly being the empty list $[]$).

The algorithm traverses the LCP-interval linearly and has two different components:

- While the ℓ-value of the interval on the top of the stack is greater than lcptab$[i]$, pop the interval and process it, then add it to the children of the new top of the stack
- If the ℓ-value of the interval on the top of the stack is smaller than lcptab$[i]$, create a new interval at the top of the stack $(lb = i, rb = \bot, children = [])$

This algorithm runs in $\mathcal{O}(n)$ time.

Enhanced Suffix Array Algorithm for Finding Maximal Repeats

One possible application of LCP-interval tree traversal is the detection of maximal repeats in a string.

Abouelhoda et al. 2002 describe an algorithm that finds maximal repeats in $\mathcal{O}(kn + z)$ time, where $k = |\Sigma|$ and z is the amount of maximal repeats. For this, it uses lcp-interval tree traversal to build position sets for an interval and its child interval and to output position sets for distinct characters. It uses lcptab, bwttab and suftab.

A position set $\mathcal{P}_{[i..j]}(a)$ is the set of all positions in an ℓ-interval $[i..j]$ that are preceded by the character $a \in \Sigma \cup \{\bot\}$:

$$\mathcal{P}(a) = \{p \in \ell - [i..j] | \text{bwttab}[p] = a\}$$

This means that for a 1-interval $[i..i+1]$, $\mathcal{P}(a)$ is $\emptyset$ for all a except bwttab$[i]$.

The current position set is saved on a per-function persistent stack. Let $\mathcal{P}_{[i'..j']}(a)$ be the position set at the top of the stack for $a \in \Sigma \cup \bot$ (position sets for all $a \in \Sigma \cup \bot$ are saved on the stack).

Given the child position sets $\mathcal{P}_{[i'..j']}(a)$ and the current position sets $\mathcal{P}^q_{[i..j]}(a)$ $(i < i' < j' < j)$ for all $a \in \Sigma \cup \bot$ ($[i..j]$ being an ℓ-interval, $[i'..j']$ being an $\ell+1$-interval), output $(p, p + \ell - 1), (p', p' + \ell - 1))$ for $p \in \mathcal{P}^q_{[i..j]}(a)$ and $p' \in \mathcal{P}^q_{[i..j]}(b)$ for $a, b \in \Sigma \cup \bot$, $a \neq b$.

The position sets are then combined and saved on the stack: $\mathcal{P}^{q+1}_{[i..j]}(e) := \mathcal{P}^q_{[i..j]}(e) \cup \mathcal{P}_{[i'..j']}(e)$ for all e in the alphabet.

Construction

Of the Suffix Array

The naive construction of the suffix array (sorting the suffixes lexically while saving their positions) uses $\mathcal{O}(n^2 \log n)$ time ($n^2 \log n$ instead of $n \log n$ because each suffix comparison can at worst use $\mathcal{O}(n)$ time). If done in place (e.g. using pointers), it uses $\mathcal{O}(n)$ space. Manber and Myers 1993 use radix sort.

Several $\mathcal{O}(n)$ algorithms have been developed for sorting suffixes: the skew algorithm described by Kärkkäinen and Sanders [2003] or the pure-induced sorting from Nong et al. 2009.

Of the Inverse Suffix Array

Neither Abouelhoda et al. 2004 nor Abouelhoda et al. 2002 describe an algorithm for constructing inverse suffix arrays. Since the algorithm is quite straightforward, it can be listed here:

```
for $i$ in $0..n-1$ do
    $\text{suftab}^{-1}[\text{suftab}[i]] \leftarrow i$
end for
```

This is a straightforward implementation of the definition of the inverse suffix array, and the algorithm has time complexity $\mathcal{O}(n)$.

Of the LCP Array

The LCP array can be constructed in linear time from the suffix tree by first constructing the suffix tree in linear time (Giegerich and Kurtz [1997]), then removing the leaves to create the LCP interval tree, and then traversing the LCP interval tree to generate the LCP array. All steps run in $\mathcal{O}(n)$, so in combination they also run in $\mathcal{O}(n)$.

Alternatively, one can use the induced sorting algorithm (Fischer [2011]) to compute both suffix arrays and LCP-arrays.

Of the Burrows-Wheeler Transform Array

The BWT array can be constructed in linear time from the suffix array using the naive algorithm (just linearly traversing the suffix array and executing $\text{bwttab}[i] = S[\text{suftab}[i] - 1]$).

Comparison

The enhanced suffix array was developed as an alternative to the Suffix Tree, which has been a successful and fast data structure in genome analysis, but uses a lot of memory. Specifically, the suffix tree uses about 20 bytes per input character (Kurtz [1999]). The goal was to create a datastructure with which one can solve the same problems as the suffix tree, with the same time complexity, but which uses less space.

The enhanced suffix array uses 10 bytes per input character under plausible assumptions:

- suftab uses $4n$ bytes (assuming $|S| \leq 2^{32}$)
- suftab^{-1} uses $4n$ bytes as well
- lcptab uses n bytes (usually assuming LCP values < 255, although Abouelhoda et al. 2004 describes how to use a secondary array to manage extreme cases)
- bwttab uses n bytes (assuming $|\Sigma| \leq 255$)

On disk, an enhanced suffix array then uses $10n$ bytes instead of $20n$ bytes for a suffix tree, but most algorithms for Enhanced Suffix Arrays don't use every component, but are usually restricted (e.g. exact string matching only needing suftab and lcptab, and suftab^{-1} only being required for finding tandem repeats). This means that practical working memory requirements during runtime are usually even smaller.

Enhanced suffix arrays also carry the benefit of a linear layout in memory, which increases cache performance as compared to less linear suffix trees, and often offer practical speedups to existing implementations (see Abouelhoda et al. 2002, sec. 7).

Applications

Enhanced suffix arrays have been used in bioinformatics software such as VMatch (see Kurtz 2003), subsuming the REPuter project (see Kurtz et al. 2001).

However, many genome comparison software projects use suffix trees instead of enhanced or extended suffix array, such as MUMmer (Kurtz et al. 2004).

Conclusion

I have presented the enhanced suffix array and the algorithms it uses to solve several string matching problems, among them finding maximal and supermaximal repeats and exact string matching. The enhanced suffix array presents a viable alternative to other string matching datastructures such as the suffix tree: the construction algorithms have a comparable time complexity ($\mathcal{O}(n)$), as well as the the specific string matching problems ($\mathcal{O}(m + \log n)$ for exact string matching, $\mathcal{O}$ for finding supermaximal repeats and MUMs, and $\mathcal{O}(|\Sigma|n + z)$ for finding maximal repeats). Beyond that, in practice enhanced suffix arrays use less working memory than suffix trees and often have faster execution times due to linear layout in memory.

I have not covered algorithms for finding tandem repeats, methods for updating enhanced suffix arrays when the underlying string S changes, and compressed enhanced suffix arrays.

References

Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *International Workshop on Algorithms in Bioinformatics*, pages 449–463. Springer, 2002.

Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1): 53–86, 2004.

Robert S Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

Johannes Fischer. Inducing the lcp-array. In *Workshop on Algorithms and Data Structures*, pages 374–385. Springer, 2011.

Robert Giegerich and Stefan Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3): 331–353, 1997.

Dan Gusfield. Algorithms on strings, trees, and sequences. 1997. *Computer Science and Computational Biology. New York: Cambridge University Press*, 1997.

Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International colloquium on automata, languages, and programming*, pages 943–955. Springer, 2003.

Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Annual Symposium on Combinatorial Pattern Matching*, pages 181–192. Springer, 2001.

Stefan Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

Stefan Kurtz. The vmatch large scale sequence analysis software. *Ref Type: Computer Program*, 412:297, 2003.

Stefan Kurtz, Jomuna V Choudhuri, Enno Ohlebusch, Chris Schleiermacher, Jens Stoye, and Robert Giegerich. Reputer: the manifold applications of repeat analysis on a genomic scale. *Nucleic acids research*, 29(22):4633–4642, 2001.

Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. Versatile and open software for comparing large genomes. *Genome biology*, 5(2):R12, 2004.

Eric S Lander, Lauren M Linton, Bruce Birren, Chad Nusbaum, Michael C Zody, Jennifer Baldwin, Keri Devon, Ken Dewar, Michael Doyle, William FitzHugh, et al. Initial sequencing and analysis of the human genome. 2001.

Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

Eugene W Myers, Granger G Sutton, Art L Delcher, Ian M Dew, Dan P Fasulo, Michael J Flanigan, Saul A Kravitz, Clark M Mobarry, Knut HJ Reinert, Karin A Remington, et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, 2000.

Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 Data Compression Conference*, pages 193–202. IEEE, 2009.

Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log log n search. *Communications of the ACM*, 21(7):550–553, 1978.

Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. Dynamic extended suffix arrays. *Journal of Discrete Algorithms*, 8(2):241–257, 2010.

Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.