Describing and testing two new linear Suffix Array Construction Algorithms

Aurélien Bellet

Department of Computing & Software

McMaster University, Hamilton, Ontario, Canada L8S 4K1

belleta@mcmaster.ca

April 2009

Abstract

In late 2008, Nong, Zhang and Chan proposed in [1] two new algorithms for linear suffix array construction, and demonstrate through experiments that they outperform the existinglinear SACAs (KA [2] and KS [3] algorithms). In this project, we first provide a intuitive description of these two new algorithms, making their main ideas very clear. We then test them against MP [4], an algorithm considerably more efficient than KA and KS in practice, but not linear in the worst case.

1 Introduction

In this section, we first give some basic definitions[1] to ensure the reader's understanding of this report. Then, we define the suffix array of a string, and briefly mention its algorithmic applications.

1.1 Basic definitions

The strings we are considering in this project are linear strings. A ***linear string*** is a finite sequence of characters (***letters***) that are members of a set Σ (***alphabet***). In this paper, we suppose that this alphabet has finite size $|\Sigma| = k$, and we denote a (linear) string x by an array $x[0..n-1]$ of n letters, where $n = |x|$ is the ***length*** of x. Additionally, ϵ denotes the empty string (*i.e.* of length 0), $x[i]$ represents the character at position i in the string x (for $i \in 0..n-1$ integer), and x^k denotes the string x repeated k times ($k \geq 0$).

[1]These definitions and notations are mostly borrowed from [5], with small changes being made to fit with the notations used to describe the presented algorithms.

Any ***substring*** of x can be denoted by $x[i..j]$, $0 \leq i \leq j \leq n-1$, where $x[i..j]$ is the substring of length $j-i+1$ starting at position i. For any integer $i \in -1..n-1$, we call the substring $x[0..i]$ a ***prefix*** of x. Similarly, for any integer $j \in 0..n$, we call $x[j..n-1]$ a ***suffix*** of x. For example, $x =$ `aabc` has prefixes ϵ, `a`, `aa`, `aab`, `aabc`, and suffixes ϵ, `c`, `bc`, `abc`, `aabc`.

The algorithms we will be considering require the alphabet to be ***ordered***: $\forall \lambda, \mu \in \Sigma$, it is decidable in constant time whether $\lambda < \mu$ for some order relation $<$ (for example, the ***lexicographical order***, also known as alphabetic or dictionary order). The alphabet should also be either a constant alphabet — of size $O(1)$ — or an integer alphabet (consisting of characters in $\{0, \ldots, \alpha\}$ for $\alpha \geq 1$). Finally, a string is supposed to be terminated by a ***sentinel*** \$, which is the unique smallest character in the string (in other words, all the other characters in the string are larger than \$).

1.2 Suffix Array

1.2.1 What is a suffix array ?

Definition 1.2.1 *The suffix array of a string x, denoted SA, is an array of integers giving the starting positions of the suffixes of x (ignoring ϵ), sorted ascendingly in lexicographical order.*

As an example, let $\Sigma = \{\$, \texttt{a}, \texttt{b}, \texttt{c}\}$ and $x =$ `aabbcbbccab$`. The suffix array of x is shown in Table 1.

i	**SA[i]**	**x[SA[i]..n − 1]**
0	11	`$`
1	0	`aabbcbbccab$`
2	9	`ab$`
3	1	`abbcbbccab$`
4	10	`b$`
5	2	`bbcbbccab$`
6	5	`bbccab$`
7	3	`bcbbccab$`
8	6	`bccab$`
9	8	`cab$`
10	4	`cbbccab$`
11	7	`ccab$`

Table 1: Suffix array of $x =$ `aabbcbbccab$`, and the corresponding suffixes $x[SA[i]..n-1]$.

The concept of suffix array was introduced by Myers and Manber in [6, 7] as a space-saving alternative to another structure: the ***suffix tree***, proposed about 20 years before in [8], and widely used since then.

A suffix array basically contains the same information as a suffix tree but is a lot more compact, reducing memory consumption, and making primary memory storage easier. However, it took about ten years before the first efficient suffix array construction algorithms were proposed [9, 2, 10, 11]. It took even more before it was shown in [12] that suffix arrays can work out any problem solvable using suffix trees, in an equivalent time complexity.

In the last few years, a lot of effort has been put into developing ***compressed suffix arrays*** [13, 14, 15] and ***linear SACAs*** [16, 2, 3, 1], making the structure and its construction as compact and efficient as possible, to face the ever growing need for large-scale applications such as web searching and genome databases.

1.2.2 Algorithmic applications

Thanks to their lightweight nature, suffix arrays are now used in many applications for which suffix trees were previously the data structure of choice.

The most common application is probably that a suffix array of a string x can be used as an index to efficiently find every occurrence of a substring p in x. Indeed, this is equivalent to finding every suffix for which p is a prefix. Since the suffix array is lexicographically ordered, these suffixes are following one another in the array, and can be found using a binary search. As performed in [6], this binary search takes $O(m + \log n)$ time in the worst case, where $m = |p|$ and $n = |x|$. More recently, some linear-time search algorithms for constant-size alphabet based on suffix arrays have been introduced [17, 12].

Another important application is the computation of the Burrows-Wheeler Transform (BWT) introduced in [18], widely used in data compression algorithms such as `bzip2`.

2 Description of SA-IS and SA-DS algorithms

In this section, we describe two new algorithms presented in [1]. Our purpose is to keep this description as simple as possible, using clear examples, to provide the reader with an insight into the main ideas of these algorithms. Proofs and implementation details can be found in the original paper.

Both algorithms share a divide-and-conquer approach, which includes 3 main steps :

1. **Problem reduction**: the input string is reduced into a smaller string.
2. **Recursion**: the suffix array of the reduced problem is recursively computed.
3. **Problem induction**: based on the suffix array of the reduced problem, that of the unreduced problem is induced.

2.1 SA-IS Algorithm

2.1.1 Problem reduction

In the SA-IS algorithm, the problem reduction is achieved by sampling the ***LMS-substrings*** of a string. Let $x[0, n-1]$ be a string of n characters (the last one being the sentinel \$).

Definition 2.1.1 *(S-type and L-type suffix) A suffix $x[i, n-1]$ is said to be S-type or L-type if $x[i, n-1] < x[i+1, n-1]$ or $x[i, n-1] > x[i+1, n-1]$, respectively. The last suffix $x[n-1, n-1]$, consisting of the character \$ only, is defined as S-type.*

Definition 2.1.2 *(S-type and L-type character) A character $x[i]$ is said to be S-type or L-type if the suffix $x[i, n-1]$ is S-type or L-type, respectively.*

Definition 2.1.3 *(LMS character) A character $x[i]$ is called leftmost S-type (LMS) if $x[i]$ is S-type and $x[i-1]$ is L-type.*

Definition 2.1.4 *(LMS-substring) A LMS-substring is (i) a substring $x[i..j]$ with both $x[i]$ and $x[j]$ being LMS characters, and there is no other LMS character in the substring, for $i \neq j$; or (ii) the sentinel itself.*

Table 2 illustrates these definitions, letting $x =$ `aabbcbbccab$`. Row 3 shows the type (S or L) of each character / suffix starting from this character. Row 4 identifies the LMS characters of x. Note that $x[5..11]$ is not a LMS-substring because it contains a LMS character.

0	1	2	3	4	5	6	7	8	9	10	11
a	a	b	b	c	b	b	c	c	a	b	\$
S	S	S	S	L	S	S	L	L	S	L	S
					●				●		●

Table 2: x has three LMS-substrings: $x[5..9]$, $x[9..11]$ and $x[11]$.

Intuitively, we notice that the LMS-substrings can be seen as the basic blocks of the string: within each LMS-substring, the lexicographical values of the characters keep increasing and then decreasing (*i.e*, in the form of S...LS). Not surprisingly, the next step is then to sort these LMS-substrings. We define a specific order relation to perform this sorting.

Definition 2.1.5 *(LMS-substring order $<_{LMS}$) To determine the order of any two LMS-substrings, we compare their corresponding characters from left to right: for each pair of characters, we compare their lexicographical values first, and next their types if the two characters are of the same lexicographical value, where the S-type is larger than the L-type.*

From the definition, we can see that two LMS-substrings have to same order index if and only if they are equal in terms of length, characters and types. Let n_1 denote the number of LMS-substrings of x. We use the order index of each LMS-substring as its name to generate a new string x_1 of length n_1, consisting of the sequence of the names of the LMS-substrings, preserving their original position in x. Therefore, in our example, $x_1 = \texttt{210}$, since \$ $<_{LMS}$ `ab$` $<_{LMS}$ `bbcca`.

2.1.2 Recursion

As we said before, we want to use the suffix array SA_1 of the reduced string x_1 to induce that of the original string x. Before considering this induction process, we need to calculate SA_1 recursively :

- if each character in x_1 is unique, we compute directly SA_1.
- otherwise, we use a recursive call on x_1.

In the first case, it is clear that we can compute SA_1 by scanning x_1 once: $SA_1[x_1[i]] = i$ for $i = 0...n_1$. As for the second case, we have to make sure that we reach the stopping criterion (*i.e.* the first case) after a finite number of iterations. This is confirmed by the following lemma (the proof is available in [1]).

Lemma 2.1.1 *(1/2 reduction ratio)* $|x_1|$ *is at most halt of* $|x|$, *i.e.* $n_1 \leq \lfloor n/2 \rfloor$.

2.1.3 Problem induction

In this section, we describe the algorithm for inducing the suffix array SA of the original string x from the suffix array SA_1 of the reduced string x_1. We also need the type (L or S) of each character in x, along with the array $P[0..n_1 - 1]$ containing the position of all the LMS characters in x, preserving their original positional order. Note that it has been computed before (during the reduction step), and is therefore still accessible.

Before investigating the induction process, we briefly introduce the concept of ***buckets*** in an array. It consists in partitioning an array into a number of parts (buckets). In our case, the array is SA and there is a bucket for each unique letter in x, its size being equal to the number of occurrences of this letter in x, and the buckets are ordered lexicographically. For each bucket, we can set a current position within the bucket.

The induction process consists of the four following steps :

1. Set all the items of SA to -1. Scan x to find the buckets in SA.
2. Set the current position of each bucket at its end. Scan SA_1 from right to left and put $P[SA_1[i]]$ to the current position of its bucket in SA and move the bucket position to the left.

3. Set the current position of each bucket at its start. Scan SA from left to right: for each non-negative item $SA[i]$, if $x[SA[i]-1]$ is L-type, then put $SA[i]-1$ to the current position of its bucket in SA and move the bucket position to the right.

4. Set the current position of each bucket as its end. Scan SA from right to left: if $x[SA[i]-1]$ is S-type, then put $SA[i]-1$ to the current position of its bucket and move the bucket position to the left.

A correctness proof of this induction process is given in [1]. Intuitively, step 2 simply puts all the sorted LMS suffixes into their buckets. Step 3 sorts all the L-type suffixes, using the order of the LMS suffixes. Finally, step 4 sorts all the S-type suffixes, using the order of the L-type suffixes.

Table 3 shows the intermediate suffix array $SA_{(i)}$ after each step i, letting $x = \texttt{aabbcbbccab\$}$, and using the arrays $SA_1 = [2, 1, 0]$ and $P = [5, 9, 11]$ computed before. Looking back on Table 1, we can check that $SA_{(4)}$ is indeed the suffix array of x.

bkt	i	$\mathbf{SA_{(1)}}$	$\mathbf{SA_{(2)}}$	$\mathbf{SA_{(3)}}$	$\mathbf{SA_{(4)}}$
\$	0	-1	11	11	11
a	1	-1	-1	-1	0
a	2	-1	-1	-1	9
a	3	-1	9	9	1
b	4	-1	-1	10	10
b	5	-1	-1	-1	2
b	6	-1	-1	-1	5
b	7	-1	-1	-1	3
b	8	-1	5	5	6
c	9	-1	-1	8	8
c	10	-1	-1	4	4
c	11	-1	-1	7	7

Table 3: SA-IS induction example with $x = \texttt{aabbcbbccab\$}$.

2.1.4 Algorithm framework

To summarize what we have seen, Algorithm 1 shows a simplified SA-IS framework.

2.1.5 Complexity

Theorem 2.1.1 *(Time / Space Complexities) Given x is of a constant or integer alphabet, the time and space complexities for SA-IS to compute SA are $O(n)$ and $O(n\lceil\log n\rceil)$ bits, respectively.*

```
Algorithm 1: SA-IS
   Data: $x$ input string of length $n$, $SA$: array $[0..n-1]$ of integer.
   Result: $SA$ suffix array of $x$.
1  Classify all the characters of $x$ as L-type or S-type;
2  Find all the LMS-substrings in $x$;
3  Sort all the LMS-substrings;
4  Name each LMS-substrings by its order index to get a shortened string $x_1$;
5  if each character in $x_1$ is unique then
6      Directly compute $SA_1$ from $x_1$;
7  else
8      SA-IS($x_1$,$SA_1$);
9  Induce $SA$ from $SA_1$;
10 return
```

Proof. It is clear that Lines 1-2 and 4-6 can be done in linear time. Sorting the LMS-substrings (Line 4) is also achieved in $O(n)$ time in the implementation presented in [1]. Besides, inducing SA from SA_1 (Line 9) is obviously done in linear time (it consists of four sequential linear-time stages). The problem is reduced by 1/2 in the worst case at each recursion (Lemma 2.1.1), which gives us the following equation for the time complexity: $\mathcal{T}(n) = \mathcal{T}(\lfloor n/2 \rfloor) + O(n) = O(n)$. As for the space complexity, the proof, which is of course intimately related to the implementation details, is available in [1].

2.2 SA-DS Algorithm

2.2.1 Problem reduction

In SA-DS, the problem reduction is achieved by sampling the ***d-critical substrings*** of a string. Let $x[0, n-1]$ be a string of n characters (the last one being the sentinel \$).

Definition 2.2.1 *(Critical Character) A character $x[i]$ is said to be d-critical, where $d \geq 2$, if and only if (i) $x[i]$ is a LMS character; or else (ii) $x[i-d]$ is a d-critical character, $x[i+1]$ is not a LMS character and no character in $x[i-d+1..i-1]$ is d-critical.*

Definition 2.2.2 *(Critical Substring) The substring $x[i..i+d+1]$ is said to be a d-critical substring for the d-critical character $x[i]$ in x. For $i \geq n-(d+1)$, $x[i..i+d+1] = x[i..n-2]\{x[n-1]\}^{d+1-(n-2-i)}$.*

First of all, we have the following immediate observation: the concept of d-critical character is strongly related to that of the LMS character (each LMS character is also a d-critical character). However, the main difference between LMS-substrings and d-critical subtrings is their length: LMS-substrings have a variable length while d-critical substrings have a fixed length $d+2$. These observations provide us with an insight into how SA-IS and SA-DS are linked together: SA-DS has a smaller reduction ratio (because the number of d-critical substrings is greater or equal to the number of LMS-substrings), but deals with fixed-length substrings that can be sorted more efficiently.

Table 4 illustrates the previous definitions, letting x = `aabbcbbccab`\$ and $d = 2$. Row 4 identifies the 2-critical characters of x. Among them, we recognize the three LMS characters from the Table 2, plus $x[7]$. Each d-critical substring has length $d + 2 = 4$.

0	1	2	3	4	5	6	7	8	9	10	11
a	a	b	b	c	b	b	c	c	a	b	\$
S	S	S	S	L	S	S	L	L	S	L	S
					●		●		●		●

Table 4: x has four 2-critical substrings: $x[5..8]$ = `bbcc`, $x[7..10]$ = `ccab`, $x[9..12]$ = `ab`\$\$ and $x[11..14]$ = \$\$\$\$.

Now, let n_1 denote the number of d-critical substrings of x. We apply the same strategy as in SA-IS: we sort the d-critical substrings and use the order index of each substring as its name to generate a new string x_1 of length n_1, consisting of the sequence of the names of the substrings, preserving their original position in x. In our example, x_1 = `2310`, since \$\$\$\$ $<_{LMS}$ `ab`\$ $<_{LMS}$ `bbcc` $<_{LMS}$ `ccab`.

2.2.2 Recursion

The recursion is performed in the exact same way as in SA-IS: if each character in x_1 is unique, we compute SA_1 directly from x_1, otherwise we call the algorithm recursively on x_1. The previously stated Lemma 2.1.1 about the 1/2 reduction ratio, which holds for SA-DS (proof in [1]), guarantees that the recursion stops.

2.2.3 Problem induction

SA-DS uses the same induction algorithm as SA-IS, except for a single change in step 2, which becomes :

2. Set the current position of each bucket at its end. Scan SA_1 from right to left: if $x[P[SA_1[i]]]$ is a LMS character, then put $P[SA_1[i]]$ to the current position of its bucket in SA and move the bucket position to the left.

Therefore, the d-critical characters that are not LMS are ignored in this step, as they are not needed for the induction. Letting x = `aabbcbbccab`\$, and using the arrays $SA_1 = [3, 2, 0, 1]$ and $P = [5, 7, 9, 11]$ computed before, we thus get the same result as that shown in Table 3 for the SA-IS algorithm.

2.2.4 Algorithm framework

Algorithm 2 shows a simplified SA-DS framework that sums up what we have seen. As we have already noticed, SA-DS can be seen as a modified version of SA-IS, performing a trade-off between reduction ratio and ease of sorting the substrings.

```
Algorithm 2: SA-DS
   Data: $x$ input string of length $n$, $SA$: array $[0..n-1]$ of integer.
   Result: $SA$ suffix array of $x$.
 1 Classify all the characters of $x$ as L-type or S-type;
 2 Find all the d-critical substrings in $x$;
 3 Sort all the d-critical substrings;
 4 Name each d-critical substrings by its order index to get a shortened string $x_1$;
 5 if each character in $x_1$ is unique then
 6     Directly compute $SA_1$ from $x_1$;
 7 else
 8     SA-DS($x_1$,$SA_1$);
 9 Induce $SA$ from $SA_1$;
10 return
```

2.2.5 Complexity

Theorem 2.2.1 *(Time / Space Complexities) Given x is of a constant or integer alphabet, the time and space complexities SA-DS to compute SA are $O(n)$ and $O(n\lceil\log n\rceil)$ bits, respectively.*

Proof. It is clear that Lines 1-2 and 4-6 can be done in linear time. Sorting the d-critical substrings (Line 4) is also achieved in $O(n)$ time in the implementation presented in [1]. Besides, inducing SA from SA_1 (Line 9) is performed by reusing the SA-IS linear induction step. As in SA-IS, the problem is reduced by 1/2 in the worst case at each recursion (Lemma 2.1.1), therefore we end up with the same equation for the time complexity: $\mathcal{T}(n) = \mathcal{T}(\lfloor n/2 \rfloor) + O(n) = O(n)$. The proof for the space complexity is available in [1].

3 Experiments

In [1], SA-IS and SA-DS were only tested against two linear SACAs: KA [2] and KS [3]. However, several SACAs considerably more efficient than KA and KS in practice, although not linear in the worst case, can be found in the literature. According to the experiments conducted on the occasion of a recent taxonomy of SACAs [19], MP [4] is by far the best among all algorithms tested, despite its $O(n^2 \log n)$ worst-case complexity. Therefore, it it a good candidate to gauge the practical performance of SA-IS ans SA-DS in terms of runtime and memory usage.

3.1 Experimental setup

3.1.1 Implementations

We used the implementations of SA-IS and SA-DS proposed in the appendix of the original paper [1]. Additionally, we also tested an optimized implementation of SA-IS by Mori, who has shown that his implementation, called SAIS, outperforms the original implementation [20]. As for MP, we used an implementation gracefully provided by its authors. All the algorithms were implemented in C++.

3.1.2 Platform

All tests were conducted on a dual 2.4GHz Xeon Server with 4Gb of RAM. The operating system was RedHat Enterprise Linux 5.3. The compiler was `g++` (`gcc` version 4.1.2) executed with the `-O3` option.

3.1.3 Runtime and memory usage

Running times are the average of five runs and do not include the time spent reading the input files. Times were recorded by using the C++ standard library function `clock()`. Peak memory usage was measured with the Valgrind[2] tool `massif`.

3.1.4 Test data

We chose to use the same test data as in [19], namely a selection of ten files from the Canterbury corpus[3] and the corpus compiled by Manzini[4]. Table 5 shows the details of the tested files. Mean and Max LCP (Longuest Common Prefix among all suffixes in the string) give the average and worst-case number of character comparisons respectively, required to separate two suffixes. Roughly, the higher those values, the more difficult the problem.

3.2 Test results

Results are exposed in Table 6 and 7 and summarized in Figure 1.

Algorithm MP is the fastest on eight of the ten files. On average, it outperforms SA-IS, SA-DS and SAIS by about 70%, 142% and 13% respectively. Thus, although MP has a clear advantage on SA-IS and SA-DS, SAIS is only slightly slower than MP (it is actually even a bit faster on the files `jdk13` and `reuters`). Moreover, SAIS has the smallest working memory: about 6%, 8% and 20% less than SA-IS, SA-DS and MP[5] on average, respectively.

[2]http://valgrind.org/

[3]http://corpus.canterbury.ac.nz/

[4]http://web.unipmn.it/~manzini/lightweight/corpus/

[5]Note that there exists an implementation of MP that can achieve a better memory use ($5.13n$ on average instead of $6n$ according to [19]), which is still larger than SAIS.

String	**Size (bytes)**	**$\|\Sigma\|$**	**Mean LCP**	**Max LCP**	**Description**
`bible`	4,047,392	63	14	551	King James Bible
`chr22`	34,553,758	5	1,979	199,999	Human chromosome 22
`ecoli`	4,638,690	4	17	2,815	*Escherichia Coli* genome
`etext`	105,277,340	146	1,108	286,352	Texts from Gutenberg Project
`howto`	39,422,105	197	267	70,720	Linux Howto files
`jdk13`	69,728,899	113	679	37,334	JDK 1.3 documentation
`reuters`	114,711,151	93	282	26,597	Reuters news in XML format
`rfc`	116,421,901	120	93	3,445	Concatenated IETF RFC files
`sprot`	109,617,186	66	89	7,373	SwissProt database
`world`	2,473,400	94	23	559	CIA World Fact Book

Table 5: Description of the data set used for our experiments.

Data	SA-IS	SA-DS	SAIS	MP
`bible`	4.45	6.69	3.38	**2.38**
`chr22`	46.81	67.92	32.33	**28.44**
`ecoli`	5.28	7.89	4.18	**3.10**
`etext`	189.88	268.75	127.96	**100.58**
`howto`	55.76	80.89	37.96	**26.59**
`jdk13`	81.86	117.15	**53.44**	61.80
`reuters`	167.72	233.94	**109.09**	111.76
`rfc`	162.33	234.08	107.25	**87.95**
`sprot`	173.39	240.66	113.49	**97.07**
`world`	2.32	3.66	1.81	**1.22**
TOTAL	889.80	1,261.63	590.89	**520.89**

Table 6: CPU time (seconds) on text data. Minimum is shown in bold for each string.

Another strong advantage of SAIS is the stability of its memory use ($5n$). Therefore, when memory is tight, SAIS may be a good choice, since it also achieves pretty good speed.

4 Concluding remarks

In this project, we described in simple and intuitive manner two new linear SACAs, and showed through experiments that algorithm SA-IS (with SAIS optimized coding) is lightweight, linear in the worst case and reasonably fast in practice. It outperforms the other well-known linear SACAs, but is also on a par with the fastest well-known SACA in terms of runtimes. It is therefore an algorithm of choice, especially when memory use is a significant issue.

Data	SA-IS	SA-DS	SAIS	MP
bible	21.88	22.54	**20.24**	26.38
chr22	186.74	193.40	**172.77**	208.37
ecoli	25.47	26.37	**23.19**	29.93
etext	568.51	586.74	**526.39**	632.71
howto	213.03	218.18	**197.11**	237.58
jdk13	363.69	366.03	**348.64**	419.42
reuters	603.42	610.90	**573.56**	689.32
rfc	617.28	627.65	**582.11**	699.58
sprot	581.52	587.67	**548.09**	658.75
world	13.32	13.53	**12.37**	16.94
TOTAL	3,194.86	3,253.01	**3,004.47**	3,618.98

Table 7: Peak memory usage (MBytes) on text data.

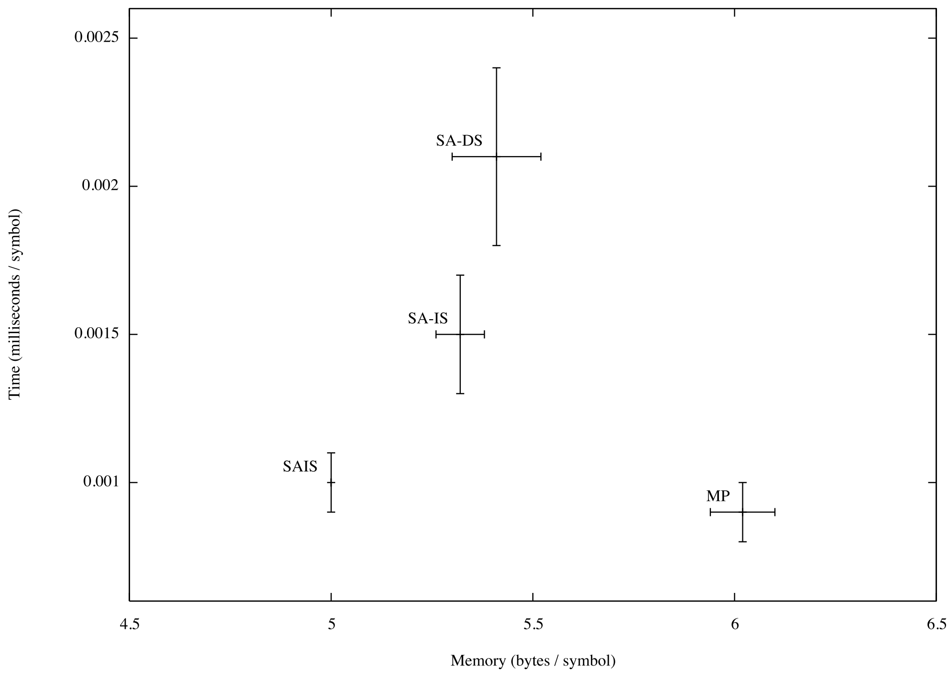

Figure 1: Resource requirements of the algorithms averaged over the data set. Error bars are a standard deviation.

On the other hand, SA-DS as implemented in [1] is clearly too slow in practice. However, since the re-implementation of SA-IS led to significant improvements both in time and space, and considering that SA-IS and SA-DS share a similar strategy (using the LMS characters), there may be room for improvement. Such a re-implementation should take advantage of the interesting feature of SA-DS: dealing with fixed-length substrings.

References

[1] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear suffix array construction. To be published, 2008.

[2] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium CPM*, pages 200–210, 2003.

[3] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[4] M. A. Maniscalco and S. J. Puglisi. Faster lightweight suffix array construction. In *17th Australasian Workshop on Combinatorial Algorithms*, pages 122–133, 2006.

[5] W. F. Smyth. *Computing Patterns in Strings*. Pearson Addison-Wesley, 2003.

[6] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[7] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 25(9):935–948, 1993.

[8] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE Computer Society, 1973.

[9] J. N. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, 1999.

[10] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Simple linear work suffix array construction. In *Proc. 30th International Col loquium Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[11] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Annual Symposium CPM*, pages 55–69, 2003.

[12] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[13] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th International Conference on Algorithms and Computation*, pages 410–421. Springer-Verlag, 2000.

[14] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th annual ACM-SIAM symposium on Discrete algorithms*, pages 225–232, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[15] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[16] D. Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symp. Combinatorial Pattern Matching*, pages 186–199. Springer-Verlag, 2003.

[17] J. S. Sim, D. K. Kim, H. Park, and K. Park. Linear-time search in suffix arrays. In *Proc. 14th Australasian Workshop on Combinatorial Algorithms*, pages 139–146, 2003.

[18] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital SRC Research, 1994.

[19] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007.

[20] Y. Mori. SAIS. http://yuta.256.googlepages.com/sais.