Linear Suffix Array Construction by Almost Pure Induced-Sorting

Ge Nong*
Computer Science Department
Sun Yat-Sen University
Guangzhou 510275, P.R.C.
issng@mail.sysu.edu.cn

Sen Zhang†
Dept. of Math., Comp. Sci. & Stat.
SUNY College at Oneonta
NY 07104, U.S.A.
zhangs@oneonta.edu

Wai Hong Chan‡
Department of Mathematics
Hong Kong Baptist University
Kowloon, Hong Kong
dchan@hkbu.edu.hk

Abstract

We present a linear time and space suffix array (SA) construction algorithm called the SA-IS algorithm. The SA-IS algorithm is novel because of the LMS-substrings used for the problem reduction and the pure induced-sorting (specially coined for this algorithm) used to propagate the order of suffixes as well as that of LMS-substrings, which makes the algorithm almost purely relying on induced sorting at both its crucial steps. The pure induced-sorting renders the algorithm an elegant design and in turn a surprisingly compact implementation which consists of less than 100 lines of C code. The experimental results demonstrate that this newly proposed algorithm yields noticeably better time and space efficiencies than all the currently published linear time algorithms for SA construction.

1 Introduction

A suffix of a string is a substring ending at the last character of the string. A suffix array for a given string is an array of specially arranged indexes pointing to all the suffixes of the string ascendingly sorted according to their lexicographical order. Suffix arrays were first introduced by Manber and Myers in SODA'90 [5] as a space efficient alternative to suffix trees. Since then, suffix arrays have been used as fundamental data structures in a broad spectrum of applications, e.g., data indexing, compressing, retrieving, storing and processing. Very recently, the research on both time and space more efficient suffix array construction algorithms (SACAs) has become an increasingly hotter pursuit due to the growing need of constructions of suffix arrays for large-scale applications, e.g., web searching and biological genome database, where the magnitude of huge datasets is measured often in billions of characters [1,2,6,7]. The fastest linear SACA algorithm among all the latest results obtained so far is the KA algorithm from Aluru and Ko [4].

The framework of the KA algorithm can be recapitulated as the following 3-step recursive process. 1) First the input string is reduced into a smaller string capturing only S-substrings using more compact coding words of them, so that the original problem is divided into a reduced part and an unreduced part. This step is commonly known as substring naming for all the algorithms sharing the same recursive framework. 2) Then the suffix array of the reduced problem is recursively computed. To be specific, if the suffix array

*Nong was partially supported by the National Science Foundation of P.R.C. (Grant No. 60873056).

†Zhang was partially supported by the SUNY College Oneonta W.B. Ford Grant.

‡Chan was partially supported by the Faculty Research Grant (FRG/07-08/II-30), HKBU.

1068-0314/09 $25.00 © 2009 IEEE
DOI 10.1109/DCC.2009.42

of the reduced problem is not immediately obtainable, it will further trigger a recursive call to solve the reduced problem. Otherwise, the recursion is terminated. 3) Finally as the recursive calls returning level by level, the order of the suffixes of the reduced problem is propagated to the suffix array of the unreduced problem all the way back to the original problem.

The elegance of the KA algorithm lies in the step 3 where a new concept called induced sorting has been proposed to efficiently sort suffixes in unreduced problems at all levels of the recursive calls. The intuition behind the induced sorting is that the significant redundancies buried in the suffixes that nest one in another can be used to propagate the order of certain representative suffixes to their left-hand longer neighbours. However, compared with its close competitor the KS algorithm [3] which shares a similar recursive framework, the KA algorithm suffers a relatively complicated step 1, where the selected varying-length S-substrings need to be sorted and re-named by their order indexes, which turns out to be the bottleneck of the KA algorithm (or any algorithm relying on induced sorting at step 3). For this task, Ko and Aluru proposed to use the S-distance lists where each list contains all the characters with the same S-distance to facilitate the sorting. Maintaining the S-distance lists, however, demands not only extra space but also an involved subroutine to process it, which is well evidenced by Ko's sample implementation of the KA algorithm with over 1000 lines in C, as opposed to only about 100 lines of the KS algorithm.

In this paper, we discuss a novel linear SACA called the SA-IS algorithm which differs from the KA algorithm in two aspects. First, the SA-IS samples the leftmost S-type (LMS) substrings (will be formally defined in section 2) instead of S-substrings (Due to space limit, we refer readers to their seminal paper [4] for its more detailed discussion). Since LMS-substrings are likely longer blocks than S-substrings, our algorithm can reduce problem faster than the KA algorithm. Second, the SA-IS uses induced-sorting NOT ONLY in step 3 to induce the ordering of the unreduced problem, BUT ALSO in step 1 to induce the ordering of the reduced problem. The new algorithm is presented and analyzed in Sections 2-3, followed by a running example in Section 4 and an experimental study for performance evaluation in Section 5. Finally, Section 6 gives the closing remarks.

2 Our Solution

```
SA-IS($S, SA$)
    ▷ $S$ is the input string;
    ▷ $SA$ is the output suffix array of $S$;
    $t$: array $[0..n-1]$ of boolean;
    $S_1$: array $[0..n_1-1]$ of integer;
    $P_1$: array $[0..n_1-1]$ of integer;
    $B$: array $[0..\|\Sigma(S)\|-1]$ of integer;
    Scan $S$ once to classify all the characters as L- or S-type into $t$;
    Scan $t$ once to find all the LMS-substrings in $S$ into $P_1$;
    Induced sort all the LMS-substrings using $P_1$ and $B$;
    Name each LMS-substring in $S$ by its bucket index to get a new shortened string $S_1$;
    if Each character in $S_1$ is unique
        then
                Directly compute $SA_1$ from $S_1$;
        else
                SA-IS($S_1, SA_1$); ▷ where recursive call happens
    Induce $SA$ from $SA_1$;
    return
```

Figure 1: The algorithm framework for induced sorting suffix array in linear time/space, where $\Sigma(S)$ denotes the alphabet of S.

2.1 Algorithm Framework Our linear suffix array sorting algorithm SA-IS is outlined in Fig. 1. Lines 1-4 first produce the reduced problem, which is then solved recursively by Lines 5-9, and finally from the solution of the reduced problem, Line 10 induces the final solution for the original problem. The time and space bottleneck of this algorithm resides at reducing the problem in Lines 1-4.

2.2 Reducing the Problem Let S be a string of n characters, represented by an array indexed by $[0..n-1]$. For presentation simplicity, S is supposed to be terminated by a sentinel \$, which is the unique lexicographically smallest character in S. Let $suf(S,i)$ be the suffix in S starting at $S[i]$ and running to the sentinel. A suffix $suf(S,i)$ is said to be S- or L-type if $suf(S,i) < suf(S,i+1)$ or $suf(S,i) > suf(S,i+1)$, respectively. The last suffix $suf(S,n-1)$ consisting of only the single character \$ (the sentinel) is defined as S-type. Correspondingly, we can classify a character $S[i]$ to be S- or L-type if $suf(S,i)$ is S- or L-type, respectively. From the above definitions, we observe the following properties: (i) $S[i]$ is S-type if (i.1) $S[i] < S[i+1]$ or (i.2) $S[i] = S[i+1]$ and $suf(S,i+1)$ is S-type; and (ii) $S[i]$ is L-type if (ii.1) $S[i] > S[i+1]$ or (ii.2) $S[i] = S[i+1]$ and $suf(S,i+1)$ is L-type. These properties suggest that by scanning S once from right to left, we can determine the type of each character in $O(1)$ time and fill out the type array t in $O(n)$ time. Further, we introduce two new concepts called LMS character and LMS-substring as following.

Definition 2.1. *(LMS character) A character $S[i]$, $i \in [1, n-1]$, is called leftmost S-type (LMS) if $S[i]$ is S-type and $S[i-1]$ is L-type.*

Definition 2.2. *(LMS-substring) A LMS-substring is (i) a substring $S[i..j]$ with both $S[i]$ and $S[j]$ being LMS characters, and there is no other LMS character in the substring, for $i \neq j$; or (ii) the sentinel itself.*

Intuitively, if we treat the LMS-substrings as basic blocks of the string, and if we can efficiently sort all the LMS-substrings, then we can use the order index of each LMS-substring as its name, and replace all the LMS-substrings in S by their names. As a result, S can be represented by a shorter string, denoted by S_1, thus the problem size can be reduced to facilitate solving the problem in a manner of divide-and-conquer. Now, we define the order for any two LMS-substrings.

Definition 2.3. *(Substring Order) 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 of higher priority than the L-type.*

From this order definition for LMS-substring, we see that two LMS-substrings can be of the same order index, i.e. the same name, iif they are equal in terms of lengths, characters and types. When $S[i] = S[j]$, we assign a higher priority to S-type because $suf(S,i) > suf(S,j)$ if $suf(S,i)$ is S-type and $suf(S,j)$ is L-type.

To sort all the LMS-substrings, we don't need extra physical space; instead, we simply maintain a pointer array P_1, which contains the pointers for all the LMS-substrings in S. This can be done by scanning S (or t) once from right to left in $O(n)$ time.

Definition 2.4. *(Sample Pointer Array) P_1 is an array containing the pointers for all the LMS-substrings in S preserving their original positional order.*

Suppose we have all the LMS-substrings sorted into the buckets in their lexicographical orders where all the LMS-substrings in a bucket are identical. Then we name each item of P_1 by the index of its bucket to produce a new string S_1. Here, we say two equal-size substrings $S[i..j]$ and $S[i'..j']$ are identical iif $S[i+k] = S[i'+k]$ and $t[i+k] = t[i'+k]$, for $k \in [0, j-i]$. We have the following observation on S_1.

Lemma 2.1. *(1/2 Reduction Ratio) $\|S_1\|$ is at most half of $\|S\|$, i.e. $n_1 \leq \lfloor n/2 \rfloor$.*

Proof. The first character in S must not be LMS and no any two consecutive characters in S are both LMS.

Lemma 2.2. *(Sentinel) The last character of S_1 is the unique smallest character in S_1.*

Proof. From the definition of LMS character, $S[n-1]$ must be a LMS character and the LMS-substring starting at $S[n-1]$ must be the unique smallest among all sampled by P_1.

Lemma 2.3. *(Coverage) For $S_1[i] = S_1[j]$, there must be $P_1[i+1] - P_1[i] = P_1[j+1] - P_1[j]$.*

Proof. Given $S_1[i] = S_1[j]$, from the definition of S_1, there must be (1) $S[P_1[i]..P_1[i+1]] = S[P_1[j]..P_1[j+1]]$ and (2) $t[P_1[i]..P_1[i+1]] = t[P_1[j]..P_1[j+1]]$. Hence, the two LMS-substrings in S starting at $S[P_1[i]]$ and $S[P_1[j]]$ must have the same length.

Lemma 2.4. *(Order Preservation) The relative order of any two suffixes $suf(S_1, i)$ and $suf(S_1, j)$ in S_1 is the same as that of $suf(S, P_1[i])$ and $suf(S, P_1[j])$ in S.*

Proof. The proof is due to the following consideration on two cases:

- Case 1: $S_1[i] \neq S_1[j]$. There must be a pair of characters in the two substrings of either different lexicographical values or different types. Given the former, the statement is obviously correct. For the latter, because we assume the S-type is of higher priority (see Definition 2.3), the statement is also correct.

- Case 2: $S_1[i] = S_1[j]$. In this case, the order of $suf(S_1, i)$ and $suf(S_1, j)$ is determined by the order of $suf(S_1, i+1)$ and $suf(S_1, j+1)$. The same argument can be recursively applied on $S_1[i+1] = S_1[j+1]$, ..., $S_1[i+k-1] = S_1[j+k-1]$ until $S_1[i+k] \neq S_1[j+k]$. Because $S_1[i..i+k-1] = S_1[j..j+k-1]$, from Lemma 2.3, $P_1[i+k] - P_1[i] = P_1[j+k] - P_1[j]$, i.e. the lengths of substrings $S[P_1[i]..P_1[i+k]]$ and $S[P_1[j]..P_1[j+k]]$ are the same. Thus, sorting $S_1[i..i+k]$ and $S_1[j..j+k]$ is equal to sorting $S[P_1[i]..P_1[i+k]]$ and $S[P_1[j]..P_1[j+k]]$. Hence, the statement is correct.

This lemma suggests that to sort all the LMS-suffixes in S, we can sort S_1 instead. As S_1 is at least 1/2 shorter than S, the computation on S_1 can be done in one half the complexity for S. Let SA and SA_1 be the suffix arrays for S and S_1, respectively. Assume SA_1 has been solved, we proceed to show how to induce SA from SA_1 in linear time/space.

2.3 Inducing SA from SA_1 As defined, SA maintains the indexes of all the suffixes of S according to their lexicographical order. Trivially, we can see that in SA, the pointers for all the suffixes starting with a same character must span consecutively. Let's call a sub-array in SA for all the suffixes with a same first character as a bucket. Further, there must be no tie between any two suffixes sharing the identical character but of different types.

Therefore, in the same bucket, all the suffixes of the same type are clustered together, and the S-type suffixes are behind, i.e. to the right of the L-type suffixes. Hence, each bucket can be further split into two sub-buckets for the L- and S-type buckets respectively.

Further, when we say to put an item $SA_1[i]$ to its bucket in SA, it means that we put $P_1[SA_1[i]]$ to the bucket in SA for the suffix $suf(S, P_1[SA_1[i]])$ in S. With these notations, we describe our algorithm for inducing SA from SA_1 in linear time/space as below:

1. Find the end of each S-type bucket; put all the items of SA_1 into their corresponding S-type buckets in SA, with their relative orders unchanged as that in SA_1;

2. Find the head of each L-type bucket in SA; scan SA from head to end, for each item $SA[i]$, if $S[SA[i]-1]$ is L-type, put $SA[i]-1$ to the current head of the L-type bucket for $S[SA[i]-1]$ and forward the current head one item to the right.

3. Find the end of each S-type bucket in SA; scan SA from end to head, for each item $SA[i]$, if $S[SA[i]-1]$ is S-type, put $SA[i]-1$ to the current end of the S-type bucket for $S[SA[i]-1]$ and forward the current end one item to the left.

Obviously, each of the above steps can be done in linear time. We now consider the correctness of this inducing algorithm by investigating each of the three steps in reversed order. First the correctness of step 3, which is about how to sort all the suffixes from the sorted L-type suffixes by induction, is endorsed by the Lemma 3 established in [4].

Lemma 2.5. *[4] Given all the L-type (or S-type) suffixes of S sorted, all the suffixes of S can be sorted in $O(n)$ time.*

In our context, Lemma 2.5 can be translated into the following statement.

Lemma 2.6. *Given all the L-type suffixes of S sorted, all the suffixes of S can be sorted by step 3 in $O(n)$ time.*

From the above lemma, we have the below result to support the correctness of step 2.

Lemma 2.7. *Given all the LMS suffixes of S sorted, all the L-type suffixes of S can be sorted by the step 2 in $O(n)$ time.*

Proof. From Lemma 2.5, if all the S-type suffixes have been sorted, we can sort all the (S- and L-type) suffixes by traversing SA once from left to right in $O(n)$ time through induction. Notice that not every S-type suffix is useful for induced sorting the L-type suffixes; instead a S-type suffix is useful only when it is also a LMS suffix. In order words, the correct order of all the LMS suffixes suffice to induce the order of all the L-type suffixes in $O(n)$ time/space.

The first step is to put all the sorted LMS suffixes into their S-type buckets in SA, from ends to heads. Hence, from lemma 2.7, step 2 will sort all the L-type suffixes correctly; and from the lemma 2.6, step 3 will sort all the suffixes from the sorted L-type suffixes.

2.4 Induced Sorting Substrings In the above discussion, we have safely assumed that how to sort substrings is not an issue. However, it turns out that this is the step that we had made the SA-IS algorithm to further optimize the KA algorithm and others. As we have mentioned earlier, sorting the variable-size S- or L-substrings constitutes the bottleneck of the KA algorithm, and solving it demands the usage of a S-distance list structure and correspondingly an involved algorithm to process the data structure. However, surprisingly we discovered that this seemingly difficult problem can be easily solved by using induced sorting too, i.e. the induced-sorting idea originally used in the KA algorithm for inducing the order of suffixes SA from SA_1 can be extended to induce the order of substrings. Specifically, we only need to make a single change to the algorithm in the section 2.3 in order to efficiently sort all the variable-length LMS-substrings. This single change is to revise step 1 to: *Find the end of each S-type bucket; put all the LMS suffixes in S into their S-type buckets in SA according to their first characters.*

To facilitate the following discussion, let's define a LMS-prefix $pre(S, i)$ to be (1) a single LMS character; or (2) the prefix $S[i..k]$ in $suf(S, i)$ where $S[k]$ is the first LMS character after $S[i]$. Similarly, we define a LMS-prefix $pre(S, i)$ to be S- or L-type if $suf(S, i)$ is S- or L-type, respectively. From this definition, any L-type LMS-prefix has at least two characters. We establish the following result for sorting all the non-size-one LMS-prefixes.

THEOREM 2.1. *The above modified induced sorting algorithm will correctly sort all the non-size-one LMS-prefixes and the sentinel.*

Proof. Initially, in step 1, all the size-one S-type LMS-prefixes are put into their buckets in SA. Now, all the LMS-prefixes in SA are sorted in their order.

We next prove, by induction, step 2 will sort all the L-type LMS-prefixes. When we append the first L-type LMS-prefix to its bucket, it must be sorted correctly with all the existing S-type LMS-prefixes. We assume this step has correctly sorted k L-type LMS-prefixes, where $k \geq 1$, and suppose contrary that when we append the $(k+1)$th L-type LMS-prefix $pre(S, i)$ to the current head of its bucket, there is already another greater L-type LMS-prefix $pre(S, j)$ in front of (i.e. on the left hand side of) $pre(S, i)$. In this case, we must have $S[i] = S[j]$, $pre(S, j+1) > pre(S, i+1)$ and $pre(S, j+1)$ is in front of $pre(S, i+1)$ in SA. This implies that when we scanned SA from left to right, before appending $pre(S, i)$ to its bucket, we must have seen the LMS-prefixes being not sorted correctly. This contradicts our assumption. As a result, all the L-type and the size-one S-type LMS-prefixes are sorted in their correct order by this step.

Now we prove step 3 will sort all the non-size-one LMS-prefixes, which is conducted symmetrically to that for step 2. When we append the first S-type LMS-prefix to its bucket, it must be sorted correctly with all the existing L-type LMS-prefixes. Notice that in step 1, all the size-one LMS-prefixes were put into the ends of their buckets. Hence, in this step, when we append a non-size-one S-type LMS-prefix to the current end of its bucket, it will overwrite the size-one LMS-prefix already there, if there is any. Assume this step has correctly sorted k S-type LMS-prefixes, for $k \geq 1$, and suppose contrary that when we append the $(k+1)$th S-type LMS-prefix $pre(S, i)$ to the current end of its bucket, there is already another less S-type LMS-prefix $pre(S, j)$ behind (i.e. on the right hand side of) $pre(S, i)$. In this case, we must have $S[i] = S[j]$, $pre(S, j+1) < pre(S, i+1)$ and $pre(S, j+1)$ is behind $pre(S, i+1)$ in SA. This implies that when we scanned SA from right to left, before appending $pre(S, i)$ to its bucket, we must have seen the non-size-one LMS-prefixes are not sorted correctly. This contradicts our assumption. As a result, all the non-size-one LMS-prefixes and the sentinel (which is unique and was sorted into its bucket in step 1) are sorted in their correct order by this step.

From this theorem, we can immediately derive the following two results. (1) Any LMS-substring is also a non-size-one LMS-prefix or the sentinel; given all the non-size-one LMS-prefixes and the sentinel ordered, all the LMS-substrings are ordered too. (2) Any S-substring is a prefix of a non-size-one LMS-prefix or the sentinel; given all the LMS-prefixes and the sentinel ordered, all the S-substrings are ordered too. Hence, our algorithm for induced sorting all the non-size-one LMS-prefixes and the sentinel can be used for sorting the LMS-substrings in our SA-IS algorithm in Fig. 1, as well as for sorting the S- or L-substrings in the KA algorithm. It is clear that the induced sorting algorithm has been heavily used in both the renaming step and the propagating step, while only one round bucket sort is needed to usher the rest induced-sorting. That is the reason why we say the SA-IS algorithm almost purely relies on induced sorting.

3 Complexity Analysis

THEOREM 3.1. *(Time/Space Complexities) Given S is of a constant or integer alphabet, the time and space complexities for the algorithm SA-IS in Fig. 1 to compute $SA(S)$ are $O(n)$ and $O(n\lceil\log n\rceil)$ bits, respectively.*

Proof. Because the problem is reduced at least 1/2 at each recursion, we have the time complexity governed by the equation below, where the reduced problem is of size at most $\lfloor n/2 \rfloor$. The first $O(n)$ in the equation counts for reducing the problem and inducing the final solution from the reduced problem.

$$\mathcal{T}(n) = \mathcal{T}(\lfloor n/2 \rfloor) + O(n) = O(n)$$

The space complexity is dominated by the space needed to store the suffix array for the reduced problem at each iteration. Because the size of suffix array at the first iteration is upper bounded by $n\lceil\log n\rceil$ bits, and decreases at least a half for each iteration thereafter, the space complexity is obvious $O(n\lceil\log n\rceil)$ bits.

To investigate the accurate space requirement, we show in Fig. 2 a space allocation scheme, where the worst-case space consumption at each level is proportional to the total length of bars at this level, and the bars for different levels are arranged vertically. In this figure, we have not shown the spaces for the input string S and the type array t, the former is fixed for a given S and the later varies from level to level. Let S_i and t_i denote the string and the type array at level i, respectively. If we keep t_i throughout the lifetime of S_i, i.e., t_i is freed only when we return to the upper level $i-1$, we need at most $2n$ bits for all the type arrays in the worst-case. However, we can also free t_i when we are going to the level $i+1$, and restore t_i from S_i when we return from the level $i+1$. In this way, we need at most n bits to reuse for all the type arrays. Because the space consumed by the type arrays is negligible when compared with SA, it is omitted in the figure.

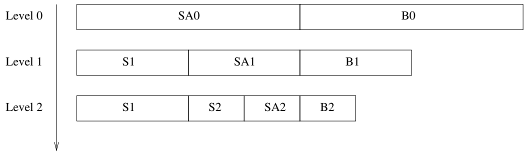

Figure 2: The worst-case space requirement at each recursion level.

The space at each level consists of two components: SA_i for the suffix array of S_i, and B_i the bucket array at level i. In the worst case, each array requires a space as large as S_i (when the alphabet of S_i is integer). For S with an integer alphabet, the peak space is at the top level. Meanwhile, for constant alphabet of S, B_0 and B_1 are $O(1)$ and $O(n)$, respectively, so the maximum space is required by the 2nd level when n increases. Hence, the space requirement is as follows, where the type arrays occupy n bits in both cases.

Corollary 3.1. *The worst-case working space requirements for SA-IS in Fig. 1 to compute the suffix array of S are: (1) $0.5n \log n + n + O(1)$ bits, for the alphabet of S is constant; and (2) $n \log n + n + O(1)$ bits, for the alphabet of S is integer.*

For the algorithm's space requirement in practice, we have the below probabilistic result.

Theorem 3.2. *Given the probabilities for each character to be S- or L-type are i.i.d as 1/2, the mean size of a non-sentinel LMS-substring is 4, i.e, the reduction ratio is at most 1/3.*

Proof. Let's consider a non-sentinel LMS-substring $S[i..j]$, where $i < j$. By the definition, $S[i]$ and $S[j]$ are LMS characters, and there must be at least one L-type character $S[k]$ in between them. Given the i.i.d probability of 1/2 for each character to be S- or L-type, the mean number of L-type characters in between $S[k]$ and $S[j]$ is governed by a geometry distribution with the mean of 1. Hence, the mean size of $S[i..j]$ is 4. Because all the LMS-substrings are located consecutively, the end of a LMS-substring is also the head of another succeeding LMS-substring. This implies that the mean size of a non-sentinel LMS-substring excluding its end is 3, resulting in the reduction ratio not greater than 1/3.

This theorem and Fig. 2 imply that if the probabilities for a character in S to be S- or L-type are equal and the alphabet of S is constant, the maximum space for our algorithm is contributed to the level 1 where $|S_1| \leq n/3$. Hence, the maximum working space is determined by the type arrays, which can be $n + O(1)$ bits in the worst case. As we will see from the experiment, this theorem well approximates the results on realistic data.

4 A Running Example

We provide below a running example of the induced sorting algorithm on naming all the LMS-substrings of a sample string $S = mmiissiissiippii\$$, where \$ is the sentinel. First, we scan S from right to left to produce the type array t at line 2, and all the LMS suffixes in S are marked by '$*$' under t. Then, we continue to run the algorithm step by step:

Step 1: The LMS suffixes are 2, 6, 10 and 16. There are 5 buckets, namely \$, i, m, p and s, for all the suffixes marked by their first characters as shown in lines 5 and 6. We initialize SA by setting all its items to be -1 and then scan S from left to right to put all the LMS suffixes into their buckets according to their first characters. Noted that the sorting in this step is not required to be stable, the suffixes belonging to a bucket can be put into the bucket in any order. In this example, we record the end of each bucket, and the LMS suffixes are put into the bucket from end to head. Hence, in the bucket for 'i', we put the suffixes first 2, next 6 and then 10. Now, SA contains all the size-one LMS-prefixes sorted.

Step 2: We first find and mark the head of each bucket with '$\wedge$', and then start to scan SA from left to right, for which the current item of SA being visited is marked by '@'. When we are visiting $SA[0] = 16$ in line 9, we check the type array t to know $S[15] = i$ is L-type. Hence, suffix 15 is appended to the bucket for 'i', and the current head of the bucket is forwarded one step to the right. In line 13, the scanning reaches $SA[2] = 14$. As $S[13] = p$ is L-type, we put suffix 13 to the bucket for 'p', and forward the bucket's head one step. To continue scanning SA, we can get all the L-type LMS-prefixes sorted in SA as shown in line 28, where '$\wedge$' between two buckets means that the left bucket is full.

Step 3: In this step, we induced sorting all non-size-one LMS-prefixes from the sorted L-type prefixes. We first mark the end of each bucket and then scan SA from right to left. At $SA[16] = 4$, we see $S[3] = i$ is S-type, and then put suffix 3 into the bucket for 'i' and forward the bucket's current end one step to the left. When we visit the next character, i.e. $SA[15] = 8$, we see $S[7] = i$ is S-type, and then we put suffix 7 to the bucket for 'i' and forward the bucket head one step to the left. To continue scanning SA in this way, all the LMS-prefixes are sorted in their order shown in line 43. Given all the non-size-one LMS-prefixes and the sentinel are sorted in SA, we scan SA once from left to right to calculate the name for each LMS-substring. As a result, we get the shortened string S_1 shown in line 45, where the names for the suffixes 2, 6, 10 and 16 are 2, 2, 1, 0, respectively.

```
00 Index: 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
01     S:  m  m  i  i  s  s  i  i  s  s  i  i  p  p  i  i  $
02     t:  L  L  S  S  L  L  S  S  L  L  S  S  L  L  L  L  S
03   LMS:        *           *           *                 *
04 Step 1:
05 Bucket:   $              i                m       p          s
06     SA: {16} {-1 -1 -1 -1 -1 10 06 02} {-1 -1} {-1 -1} {-1 -1 -1 -1}
07 Step 2:
08     SA: {16} {-1 -1 -1 -1 -1 10 06 02} {-1 -1} {-1 -1} {-1 -1 -1 -1}
09          @^    ^                         ^       ^       ^
10         {16} {15 -1 -1 -1 -1 10 06 02} {-1 -1} {-1 -1} {-1 -1 -1 -1}
11           ^   @   ^                      ^       ^       ^
12         {16} {15 14 -1 -1 -1 10 06 02} {-1 -1} {-1 -1} {-1 -1 -1 -1}
13           ^      @   ^                   ^       ^       ^
14         {16} {15 14 -1 -1 -1 10 06 02} {-1 -1} {13 -1} {-1 -1 -1 -1}
15           ^          ^       @           ^          ^    ^
16         {16} {15 14 -1 -1 -1 10 06 02} {-1 -1} {13 -1} {09 -1 -1 -1}
17           ^          ^          @        ^          ^       ^
18         {16} {15 14 -1 -1 -1 10 06 02} {-1 -1} {13 -1} {09 05 -1 -1}
19           ^          ^             @     ^          ^          ^
20         {16} {15 14 -1 -1 -1 10 06 02} {01 -1} {13 -1} {09 05 -1 -1}
21           ^          ^                  @   ^       ^          ^
22         {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 -1} {09 05 -1 -1}
23           ^          ^                        ^ @   ^          ^
24         {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 12} {09 05 -1 -1}
25           ^          ^                        ^       ^ @      ^
26         {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 12} {09 05 08 -1}
27           ^          ^                        ^       ^    @      ^
28         {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 12} {09 05 08 04}
29           ^          ^                        ^       ^             ^
30 Step 3:
31     SA: {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 12} {09 05 08 04}
32           ^                         ^       ^       ^            @^
33         {16} {15 14 -1 -1 -1 10 06 03} {01 00} {13 12} {09 05 08 04}
34           ^                      ^          ^       ^         @   ^
35         {16} {15 14 -1 -1 -1 10 07 03} {01 00} {13 12} {09 05 08 04}
36           ^                   ^             ^      @^             ^
37         {16} {15 14 -1 -1 -1 11 07 03} {01 00} {13 12} {09 05 08 04}
38           ^                ^       @        ^       ^             ^
39         {16} {15 14 -1 -1 02 11 07 03} {01 00} {13 12} {09 05 08 04}
40           ^             ^       @           ^       ^             ^
41         {16} {15 14 -1 06 02 11 07 03} {01 00} {13 12} {09 05 08 04}
42           ^          ^       @              ^       ^             ^
43         {16} {15 14 10 06 02 11 07 03} {01 00} {13 12} {09 05 08 04}
44           ^       ^                         ^       ^             ^
45     S1: 2 2 1 0
```

5 Experiments

All the datasets used were downloaded from the Manzini-Ferragina [6] corpora, the ad hoc benchmark repository for suffix array construction and compression algorithms. The experiments were performed on Linux (Sabayon Linux distribution) with AMD Athlon(tm) 64x2 Dual Core Processor 4200+ 2.20GHz and 2.00GB RAM. All the programs were implemented in C/C++ and compiled by *g++* with the -O3 option.

The performance measurements are the costs of time and space, recursion depth and mean reduction ratio. The time for each algorithm is the mean of 3 runs, and the space is the heap peak measured by using the *memusage* command to fire the running of each program. The total time (in seconds) and space (in million bytes, MBytes) for each algorithm are the sums of the times and spaces used to run the algorithm for all the input data, respectively. The mean time (in seconds per MBytes) and space (in bytes per character) for each algorithm are the total time and space divided by the total number of characters in all input data. Table 1 shows the results of the experiments. For comparison convenience, we normalize all the results by that from IS. To understand the differences of time and space used in the two algorithms we also record and compare the recursion depths (the number of iterations) and mean reduction ratios (the mean of reduction ratios for all iterations) of the two algorithms. Intuitively, the smaller the reduction ratio, the less the number of recursions, and the faster and better the algorithm. Our IS algorithm achieves all the best results among all measurements for all datasets.

Table 1: Experimental Results: Time, Space, Recursion Depth and Ratio

Data	$\|\Sigma\|$	Characters	Description	Time (Seconds)		Space (MBytes)		Depth		Ratio	
				IS	KA	IS	KA	IS	KA	IS	KA
world	94	2 473 400	CIA world fact book	**1.3**	1.9	**12.70**	21.24	**6**	6	**.32**	.42
bible	63	4 047 392	King James Bible	**2.7**	3.51	**20.86**	34.45	**6**	6	**.34**	.45
chr22	4	34 553 758	Human chromosome 22	**24.7**	33.41	**178.09**	289.97	**6**	8	**.31**	.43
E.coli	4	4 638 690	Escherichia coli genome	**2.8**	3.98	**24.29**	40.01	**7**	8	**.32**	.42
sprot	66	109 617 186	SwissProt V34 database	**94.6**	132.89	**554.58**	930.06	**7**	9	**.31**	.45
etext	146	105 277 340	Project Gutenberg	**101**	149.67	**542.17**	907.34	**11**	12	**.33**	.45
howto	197	39 422 105	Linux Howto files	**30.4**	42.85	**203.16**	331.54	**9**	13	**.32**	.45
Total		300 029 871		**257.5**	368.21	**1535.85**	2554.61	**52**	62	**2.25**	3.07
Mean				**0.86**	1.23	**5.12**	8.51	**7.43**	8.86	**.32**	.44
Norm.				**1**	1.43	**1**	1.66	**1**	1.19	**1**	1.38

In this experiment, the implementation of IS keeps the type array t_i throughout the lifetime of S_i at level i, which may lead to a usage of up to $2n$ bits in the worst case, i.e. 0.25 byte per character. As a result, the mean space is 5.12 bytes per character shown in table 1. Such a space complexity is approaching the space extreme (at least $5n$ bytes [6]) for suffix array construction, which leaves the margin for further improvement negligible.

6 Closing Remarks

For more details of the SA-IS algorithm, our technical report "Two Efficient Algorithms for Linear Suffix Array Construction" is available at http://www.cs.sysu.edu.cn/nong/. It is also worth noting that our SA-IS algorithm has been adopted in two other projects: 1)Burrows-Wheeler Alignment Tool (BWA) at http://maq.sourceforge.net/bwa-man.shtml, where IS is the default algorithm for constructing BWT index; and 2)In-place Update of Suffix Array while Recoding Words (http://www.irisa.fr/symbiose/mgalle/suffix_array_update) by the Symbiose Project Team at INRIA/Irisa. Finally, we appreciate Mori as an independent party to have re-implemented our SA-IS algorithm and performed more extensive experiments. His experimental results published at http://yuta.256.googlepages.com/sais confirm that the SA-IS algorithm is currently the most time and space efficient algorithm among all the published linear algorithms for SA construction.

References

[1] Y.-F. Chien, W.-K. Hon, R. Shah, and J. S. Vitter. Geometric burrows-wheeler transform: Linking range searching and text indexing. In *Proceedings DCC 2008 Data Compression Conference*, pages 252–261, Snowbird, UT, USA, March 2008.

[2] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of STOC'00*, pages 397–406, 2000.

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

[4] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proceedings 14th CPM, LNCS 2676, Springer-Verlag*, pages 200–210, 2003.

[5] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the first ACM-SIAM SODA*, pages 319–327, 1990.

[6] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, September 2004.

[7] S. Zhang and G. Nong. Fast and space efficient linear suffix array construction. In *Proceedings DCC 2008 Data Compression Conference*, page 553, Snowbird, UT, USA, March 2008.