Two Efficient Algorithms for Linear Time Suffix Array Construction

Ge Nong, *Member*, *IEEE*, Sen Zhang, *Member*, *IEEE*, and Wai Hong Chan

Abstract—We present, in this paper, two efficient algorithms for linear time suffix array construction. These two algorithms achieve their linear time complexities, using the techniques of divide-and-conquer, and recursion. What distinguish the proposed algorithms from other linear time suffix array construction algorithms (SACAs) are the variable-length leftmost S-type (LMS) substrings and the fixed-length d-critical substrings sampled for problem reduction, and the simple algorithms for sorting these sampled substrings: the induced sorting algorithm for the variable-length LMS substrings and the radix sorting algorithm for the fixed-length d-critical substrings. The very simple sorting mechanisms render our algorithms an elegant design framework, and, in turn, the surprisingly succinct implementations. The fully functional sample implementations of our proposed algorithms require only around 100 lines of C code for each, which is only $1/10$ of the implementation of the KA [1] algorithm and comparable to that of the KS [2] algorithm. The experimental results demonstrate that these two newly proposed algorithms yield the best time and space efficiencies among all the existing linear time SACAs.

Index Terms—Suffix array, linear time, divide-and-conquer.

1 Introduction

The concept of suffix arrays was introduced by Manber and Myers in SODA'90 [3] and SICOMP'93 [4] as a space efficient alternative to suffix trees, and since then has been well-recognized as a fundamental data structure, useful for a broad spectrum of applications, e.g., data indexing, retrieving, storing, and processing. For an n-character string, denoted by S, its suffix array, denoted by $SA(S)$, is an array of indices pointing to all the suffixes of S, sorted according to their ascending(or descending) lexicographical order. The suffix array of S itself requires only $n\lceil\log n\rceil$-bit space. However, different suffix array construction algorithms (SACAs) may require significantly different space and time complexities. During the past decade, a plethora of researches have been devoted to developing SACAs that are both time and space efficient, for which we recommend a thorough survey from Puglisi [5]. Very recently, the research on time and space efficient SACAs has become an even hotter pursuit; due to that, constructions of suffix arrays are needed for large-scale applications, e.g., web searching and biological genome database, where the magnitude of a huge data set is measured often in billions of characters [6], [7], [8], [9], [10]. Time and space efficient linear time algorithms are crucial for large-scale applications to have predictable worst-case performance. Our interest, herein, is *limited to linear time suffix array construction only*.

1.1 Prior-Arts

The three well-known linear time SACAs up to date are the KS [11], [2], KA [12], [1], and KSP [13] algorithms, all contemporarily reported in 2003. All of them are of linear time for an input string of either constant or integer alphabets, where a constant alphabet is of size $O(1)$ and an integer alphabet consists of the characters in $[0, n^{O(1)}]$. Among them, the KSP algorithm appears to mimic Farach's work [14] on suffix trees in using a very similar and complex merging step; thus it does not gain popularity in practice.

The KS algorithm consists of three straightforward steps [11], [2]:

1. A size-n string S (represented by an array indexed by $[0..n-1]$) is reduced to S_1 by naming each size-3 substring $S[i..i+2]$ for $i \bmod 3 \neq 0$ as an integer of size $\lceil\log n\rceil$ bits, which can be done in $O(n)$ time by simply running three passes radix sort on all the sampled fixed-size substrings. As a result, we split the original problem of size n, i.e., S, to a reduced problem of size $2n/3$, i.e., S_1, and the remaining problem of size $n/3$. Then, the suffix array of S_1 is constructed by further reductions, using $2/3$-recursion repeatedly.
2. Construct the suffix array of the remaining problem in $O(n)$ time, using induction from the suffix array of S_1.
3. Merge the two suffix arrays by a simple comparison-based algorithm in $O(n)$ time to produce the final result.

The KS algorithm requires a linear time given by $\mathcal{T}(n) = \mathcal{T}(\lceil 2n/3\rceil) + O(n) = O(n)$ and an extra working space of at least n integers, where each integer is of $\lceil\log_2 n\rceil$ bits. Herein, we define *working space* as the extra space needed in addition

- *G. Nong is with the Department of Computer Science, Sun Yat-sen University, Guangzhou 510275, P.R. China. E-mail: issng@mail.sysu.edu.cn.*
- *S. Zhang is with the Department of Mathematics, Computer Science and Statistics, State University of New York, College at Oneonta, NY 13820. E-mail: zhangs@oneonta.edu.*
- *W.H. Chan is with the Department of Mathematics and Information Technology, The Hong Kong Institute of Education, Hong Kong. E-mail: waihchan@ied.edu.hk.*

Manuscript received 21 Apr. 2009; revised 23 June 2010; accepted 30 June 2010; published online 7 Sept. 2010.
Recommended for acceptance by J. Chen.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-04-0165.
Digital Object Identifier no. 10.1109/TC.2010.188.

to the input string and the output suffix array (which are universally needed for any SACA published so far).

The key idea of the KA algorithm lies in classifying all the suffixes in the string S into two classes for problem reduction: L-type and S-type, which, to some degree, is a variant of the type-A/B suffix classification method, formerly proposed by Itoh and Tanaka [15]. The L/S-type suffix classification can be done in $O(n)$ time by simply scanning S from right to left. A character $S[i]$ is said to be L-type and S-type, if the suffix $S[i..n-1]$ is L-type and S-type, respectively. Based on the classified suffixes of L-type and S-type, a S-substring is defined as any substring $S[i..j]$, $j > i$, satisfying that $S[i]$ and $S[j]$ are the only two S-type characters in $S[i..j]$. Similarly, an L-substring $S[i..j]$, $j > i$ satisfies that $S[i]$ and $S[j]$ are the only two L-type characters in $S[i..j]$. Since the definitions of L-type and S-type substrings (see Section 3.2 for the precise definitions) are symmetric, it is safe to assume that there are fewer S-substrings; otherwise, L-substrings will be used instead. Given this assumption, the KA algorithm is composed of the following three steps [12], [1]:

1. By naming all the S-substrings in S in $O(n)$ time, the original problem S is split into a reduced problem S_1 of size at most $n/2$ and a remaining problem of size at least $n/2$, where the reduced and remaining problems consist of all the S-type and L-type suffixes in S, respectively. The suffix array of S_1 is constructed by further reductions, using 1/2-recursion repeatedly.
2. Construct in $O(n)$ time the suffix array of the remaining problem, i.e., the suffix array of all the L-type suffixes in S, using induction from the suffix array of S_1.
3. Merge the two suffix arrays for S_1 and the remaining problem, i.e., the SAs for all the S-type and L-type suffixes of S, respectively, in $O(n)$ time to produce the final result.

The merging step in the KA algorithm is very simple, benefited from this fact observed in [12], [1] for any string. That is, for any two suffixes of L-type and S-type, respectively, if their beginning characters are identical, the L-type suffix must be smaller than the S-type one. Hence, merging the two suffix arrays for the reduced and remaining problems can be done by scanning them once with simple character and type comparisons. The KA algorithm has a linear time given by $\mathcal{T}(n) = \mathcal{T}(\lceil n/2 \rceil) + O(n) = O(n)$, and a working space of $3n$ bytes plus $1.25n$ bits for a string not longer than 2^{32} [12].

Due to the space limit, we refer readers who are new to suffix arrays to [5] for more related backgrounds.

1.2 Remarks

Both the KS and KA algorithms share a similar divide-and-conquer framework, which comprises linear-time problem reduction, recursion, remaining problem induction, and merging. To be more specific, the framework works as follows:

1. First, the input string is reduced into a smaller string, so that the original problem is divided into a reduced part and a remaining part.
2. Then the suffix array of the reduced problem is recursively computed.
3. Based on the result of the previous step, the suffix array of the remaining problem is induced.
4. Finally, the two suffix arrays are merged as the final result.

In order to reduce the problem in Step 1, the selected substrings, either the triplets in the KS algorithm or the S or L-substrings in the KA algorithm, need to be sorted and renamed by their order indices. This step is commonly known as substring naming. In Step 2, if the suffix array of the reduced problem is not immediately obtainable, a recursive call is further triggered to solve the reduced problem.

The two algorithms differ from each other in how to select substrings for reducing the problem. The KS algorithm selects the fixed-length substrings that are separated by the fixed intervals; thus the problem size is reduced at each iteration in a constant reduction ratio of 2/3. In the meanwhile, the KA algorithm selects the S or L-substrings, which have varying lengths subject to the specific characteristics of a given string. The reduction ratio of the KA algorithm is always not more than 1/2 due to the symmetric definitions of L and S-type suffixes. Herein, reduction ratio is defined as the size of the new child problem against that of its parent. Due to better reduction ratio (1/2 versus 2/3), the KA algorithm is expected to run faster than the KS algorithm and use less space, which has been confirmed by the performance evaluation studies independently carried out by Puglisi [16] and Lee [17].

It appears that the KA algorithm is faster in problem reduction; however, the sampled S-substrings (or symmetrically, L-substrings) may have different and unpredictable lengths, which makes the design of algorithm for problem reduction in the KA algorithm far more complicated than that in the KS algorithm, where the fixed length substrings are sampled and sorted. For accomplishing this task, Ko and Aluru [12], [1] proposed to use the S-distance lists where each list contains all the suffixes with the same S-distance, and the S-distance for a suffix $S[i..n-1]$ is the distance from $S[i]$ to the nearest S-type character to its left (excluding $S[i]$). However, maintaining the S-distance lists demands not only extra space but also additional time. Moreover, the S-distance lists complicate the whole algorithm's design, which is well-evidenced by the sample implementations of the KS and KA algorithms; the former is embodied within only around 100 lines in C, whereas the latter uses far more than 1,000 lines. In this sense, the KS algorithm is much more elegant than the KA algorithm. Therefore, how to name the variable-length S-substrings has been identified as the performance and design bottleneck in the KA algorithm.

1.3 What Is New

Recently, we proposed in DCC'09 [18] and CPM'09 [19] two new linear time SACAs that sample the variable-length leftmost S-type (LMS) substrings (Definition 3.2) and fixed-length d-critical substrings (Definition 4.3), and use the very simple induced sorting and radix sorting methods to sort the sampled substrings, respectively. Since the LMS and d-critical substrings are statistically longer than the L or S-substrings, our algorithms achieve an even better mean reduction ratio, and thus run faster than the KA algorithm.

For our algorithm sampling the fixed-length d-critical substrings, sorting the sample substrings can be done using a very simple radix sorting algorithm, for their lengths are identical. For our another algorithm sampling variable-length LMS substrings, we do not need to use any heavy data structure like S-distance lists in the KA algorithm, but simply employ a new induced-sorting method to address the bottleneck problem of sorting the variable-length LMS substrings.

In the rest of this paper, Section 2 first introduces some basic notations for presenting our two algorithms. Further, these two new algorithms are presented and analyzed in Sections 3 and 4, respectively, followed by an extensive performance evaluation in Section 5. Finally, Section 6 concludes our results.

2 Basic Notations

We introduce, in this section, some basic notations, commonly used in the presentations of our two algorithms.

Let S be a string of n characters stored in an array $[0..n-1]$, and $\Sigma(S)$ be the alphabet of S. For a substring $S[i]S[i+1]\dots S[j]$ in S, we denote it as $S[i..j]$. For presentation simplicity, S is supposed to be terminated by a sentinel \$, which is the unique lexicographically smallest character in S (using a sentinel is widely adopted in the literatures for SACAs [5]).

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-type 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-type or L-type, if $suf(S,i)$ is S-type or L-type, respectively. To store the type of every character/suffix, we introduce an n-bit boolean array t, where $t[i]$ records the type of character $S[i]$ as well as suffix $suf(S,i)$: 1 for S-type and 0 for L-type. From the S-type and L-type 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/suffix in $O(1)$ time and fill out the type array t in $O(n)$ time.

As defined earlier, $SA(S)$ (the notation of SA is used for it when there is no confusion in the context), i.e., the suffix array of S, stores the indices 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 subarray in SA for all the suffixes with the same first character as a *bucket*, where *the head and the end of a bucket* refer to the first and the last items of the bucket, respectively. Further, there must be no tie between any two suffixes sharing the identical character but of different types. That is, 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 [12], [1]. Hence, each bucket can be further split into two subbuckets with respect to the types of suffixes inside: the L and S-type buckets, where the L-type bucket is on the left of the S-type bucket.

```
SA-IS(S, SA)
     ▷ S is the input string;
     ▷ SA is the output suffix array of S;
     t: array [0..n − 1] of boolean;
     P1, S1: array [0..n1 − 1] of integer; ▷ n1 = ||S1||
     B: array [0..||Σ(S)|| − 1] of integer;
 1   Scan S once to classify all the characters as
     L- or S-type into t;
 2   Scan t once to find all the LMS-substrings in S into P1;
 3   Induced sort all the LMS-substrings using P1 and B;
 4   Name each LMS-substring in S by its bucket
     index to get a new shortened string S1;
 5   if Each character in S1 is unique
 6      then
 7            Directly compute SA1 from S1;
 8      else
 9            SA-IS(S1, SA1); ▷ Fire a recursive call
10   Induce SA from SA1;
11   return
```

Fig. 1. The SA-IS algorithm framework.

Before going further, we would remind readers that the exact definitions of the two common symbols, P_1 and S_1, for presenting our two algorithms, are different in their respective contexts.

3 Algorithm I: Induced Sorting Variable-Length LMS-Substrings

3.1 Algorithm Framework

The framework of our linear time suffix array, sorting algorithm SA-IS that samples and sorts the variable-length LMS-substrings, 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. In the rest of this section, we further describe each step in more details.

3.2 Reducing the Problem

We start by introducing the terms of leftmost S-type (LMS) character, suffix, and substring as follows:

Definition 3.1. *(LMS Character/Suffix) A character $S[i]$, $i \in [1, n-1]$ is called LMS, if $S[i]$ is S-type and $S[i-1]$ is L-type. A suffix $suf(S,i)$ is called LMS, if $S[i]$ is an LMS character.*

Definition 3.2. *(LMS-Substring) An 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 3.3. *(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, if and only if they are equal in terms of lengths, characters, and types. Assigning the S-type character a higher priority is based on a property directly from the definitions of L-type and S-type suffixes in [12]: $suf(S,i) > suf(S,j)$, if (1) $S[i] > S[j]$, or (2) $S[i] = S[j]$, $suf(S,i)$ and $suf(S,j)$ are S-type and L-type, respectively.

To sort all the LMS-substrings, no extra physical space is needed for storing them. Instead, we simply maintain a pointer array, denoted by P_1, which contains the pointers for all the LMS-substrings in S and can be made by scanning S (or t) once from right to left in $O(n)$ time.

Definition 3.4. *(Sample Pointer Array) P_1 is an array containing the pointers for all the LMS-substrings in S with their original positional order being preserved.*

Suppose, we have all the LMS-substrings sorted in the buckets in their lexicographical order, 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, if and only if $S[i+k] = S[i'+k]$ and $t[i+k] = t[i'+k]$, for $k \in [0, j-i]$. We have the following observation on S_1.

Lemma 3.5. *(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, while the last must be LMS. Moreover, there are at least three characters in each nonsentinel LMS-substring, and any two neighboring LMS-substrings overlap on a common character. □

Lemma 3.6. *(Sentinel) The last character of S_1 must be the unique smallest character in S_1.*

Proof. From Defintion 3.2, we know that the single-character LMS-substring, i.e., the sentinel, must be the unique smallest among all the sampled LMS-substrings in P_1. □

The above two lemmas state that the size of S_1 is at most half of that of S, and S_1 is terminated by a unique smallest sentinel too.

Lemma 3.7. *(Coverage) For any two characters $S_1[i] = S_1[j]$, 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 3.8. *(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 for the following 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, it is obvious that the statement is correct. For the latter, because we assume that the S-type is of higher priority (see Definition 3.3), 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 conducted on $S_1[i+1] = S_1[j+1]$, $S_1[i+2] = S_1[j+2], \ldots S_1[i+k-1] = S_1[j+k-1]$ until $S_1[i+k] \neq S_1[j+k]$. Because that $S_1[i..i+k-1] = S_1[j..j+k-1]$, from Lemma 3.7, we must have $P_1[i+k] - P_1[i] = P_1[j+k] - P_1[j]$, i.e., the substrings $S[P_1[i]..P_1[i+k]]$ and $S[P_1[j]..P_1[j+k]]$ are of the same length. This suggests that 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 in this case, too. □

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

3.3 Inducing SA from SA_1

We describe below our algorithm for inducing SA from SA_1 in linear time.

A3.3 Algorithm for Inducing SA from SA_1 in SA-IS

1. Initialize each item of SA as -1. Find the end of each bucket in SA for all the suffixes in S. Scan SA_1 once from right to left, put $P_1[SA_1[i]]$ to the current end of the bucket for $suf(S, P_1[SA_1[i]])$ in SA and forward the bucket's end one item to the left.
2. Find the head of each bucket in SA for all the suffixes in S. Scan SA from left to right, for each *nonnegative* item $SA[i]$, if $S[SA[i]-1]$ is L-type, then put $SA[i]-1$ to the current head of the bucket for $suf(S, SA[i]-1)$ and forward that bucket's head one item to the right.
3. Find the end of each bucket in SA for all the suffixes in S. Scan SA from right to left, for each *nonnegative* item $SA[i]$, if $S[SA[i]-1]$ is S-type, then put $SA[i]-1$ to the current end of the bucket for $suf(S, SA[i]-1)$ and forward that bucket's end one item to the left.

Obviously, each of the above steps can be done in linear time $O(n)$. We now consider the correctness of this inducing algorithm by investigating each of the three steps in their 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 Lemma 3 established in [12] for supporting the KA algorithm, cited as below.

Lemma 3.9. [12] *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 3.9 can be translated into the statement below.

Lemma 3.10. *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 following result to support the correctness of Step 2.

Lemma 3.11. *Given all the LMS suffixes of S sorted, all the L-type suffixes of S can be sorted by Step 2 in $O(n)$ time.*

Proof. From Lemma 3.9, we know that given all the S-type suffixes having been sorted in SA, we can sort all the (S-type 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, an S-type suffix is useful only when it is also an LMS suffix. In order words, the correct order of all the LMS suffixes suffices to induce the order of all the L-type suffixes in $O(n)$ time. □

3.4 Induced Sorting LMS-Substrings

This part is dedicated to addressing the most challenging problem in the whole design of algorithm SA-IS: how to efficiently sort all the variable-size LMS-substrings. In the KA algorithm, sorting the variable-size S or L-substrings constitutes the bottleneck of the whole algorithm, and solving it, demands the usage of S-distance lists. Nevertheless, our solution does not need to use the cumbersome S-distance lists. Instead, we solve this once difficult problem by using the same induced sorting idea, originally used in the algorithm A3.3 in Section 3.3. Specifically, we only need to make a single change to the first step of A3.3 in order to efficiently sort all the variable-length LMS-substrings, as shown below.

A3.4 Algorithm for Induced Sorting LMS-Substrings

1. Initialize each item of SA as -1. Find the end of each bucket in SA for all the suffixes in S. Put the indices of all the LMS-suffixes in S into their buckets in SA, from the end to the head in each bucket. This is done by scanning S once from left to right (or right to left) and performing the following operations in $O(1)$ time for each scanned LMS suffix: put the suffix's index to the current end of its bucket in SA and forward that bucket's end one item to the left.
2. The same as Step 2 in the algorithm A3.3.
3. The same as Step 3 in the algorithm A3.3.

To facilitate the following discussion, let us define an LMS-prefix $pre(S,i)$ of $suf(S,i)$ to be 1) the sentinel itself when $i = n-1$; or 2) the prefix $S[i..k]$ in $suf(S,i)$, where $i \neq n-1$, $k > i$ and $S[k]$ is the first LMS character after $S[i]$. Similarly, we define an LMS-prefix $pre(S,i)$ to be S-type or L-type, if $suf(S,i)$ is S-type or L-type, respectively. We further to establish the following result for sorting all the LMS-prefixes.

Theorem 3.12. *The algorithm A3.4 for induced sorting LMS-substrings will correctly sort all the LMS-prefixes of S into SA.*

Proof. Initially, in the first step, all the LMS suffixes are put into their buckets in SA. Now, there is only one LMS-prefix in SA, i.e., the sentinel, which is sorted correctly.

We next prove, by induction, the second step 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 already in SA. Suppose, this step has correctly sorted k L-type LMS-prefixes, where $k > 1$. We show by contradiction that the next L-type LMS-prefix will be sorted correctly. Suppose 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 in SA being not sorted correctly. This contradicts our assumption. As a result, all the L-type LMS-prefixes and the sentinel are sorted in their correct order by this step.

Now we prove that the third step will further sort all the LMS-prefixes. This step is being conducted similar to what we have done in the second step. 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 already in SA. Notice that in the first step, all the LMS suffixes were put into of their buckets from the ends to the heads. Hence, in this step, when we append an S-type LMS-prefix to the current end of its S-type bucket, it will overwrite the LMS suffix already there, if there is any. Suppose, this step has correctly sorted k S-type LMS-prefixes, for $k > 1$. We show by contradiction that the next S-type LMS-prefix will be sorted correctly. Suppose 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 smaller 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 LMS-prefixes in SA being not sorted correctly. This contradicts our assumption. As a result, all the LMS-prefixes are sorted in their correct order by this step. □

From this theorem, we can immediately derive the following two results: 1) Every LMS-substring is also an LMS-prefix, given all the LMS-prefixes are ordered, all the LMS-substrings are ordered too. 2) Every S-substring is a prefix of an LMS-prefix, given all the LMS-prefixes are ordered, all the S-substrings are ordered too. Hence, our algorithm for induced sorting LMS-substrings can be used for sorting all the LMS-substrings in our SA-IS algorithm in Fig. 4, as well as for sorting the S or L-substrings in the KA algorithm.

```
00        0                   1
01 Index: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
02     S: m m i i s s i i s s i i p p i i $
03     t: L L S S L L S S L L S S L L L L S
04   LMS:     *       *       *           *

05 Step 1:
06 Bucket:   $              i                m       p          s
07     SA: {16} {-1 -1 -1 -1 -1 10 06 02} {-1 -1} {-1 -1} {-1 -1 -1 -1}

08 Step 2:
09 Bucket:   $              i                m       p          s
10     SA: {16} {-1 -1 -1 -1 -1 10 06 02} {-1 -1} {-1 -1} {-1 -1 -1 -1}
11          @^    ^                         ^       ^       ^
12         {16} {15 -1 -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 Bucket:   $              i                m       p          s
32     SA: {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 12} {09 05 08 04}
33           ^                         ^       ^       ^            @^
34         {16} {15 14 -1 -1 -1 10 06 03} {01 00} {13 12} {09 05 08 04}
35           ^                      ^          ^       ^         @   ^
36         {16} {15 14 -1 -1 -1 10 07 03} {01 00} {13 12} {09 05 08 04}
37           ^                   ^             ^      @^             ^
38         {16} {15 14 -1 -1 -1 11 07 03} {01 00} {13 12} {09 05 08 04}
39           ^                ^       @        ^       ^             ^
40         {16} {15 14 -1 -1 02 11 07 03} {01 00} {13 12} {09 05 08 04}
41           ^             ^       @           ^       ^             ^
42         {16} {15 14 -1 06 02 11 07 03} {01 00} {13 12} {09 05 08 04}
43           ^          ^                      ^       ^             ^
44         {16} {15 14 10 06 02 11 07 03} {01 00} {13 12} {09 05 08 04}
45           ^       ^          @              ^       ^             ^

46     S1: 2 2 1 0
```

Fig. 2. A running example for algorithm A3.4.

3.5 Example

We provide in Fig. 2 a running example of the algorithm A3.4 for induced sorting and 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 3, 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 five buckets for all the suffixes marked by their first characters, i.e., \$, i, m, p, and s, respectively. Each bucket is delimited by a pair of braces, as shown in lines 6 and 7. We initialize SA by setting all its items to be -1, and then scan S from left to right to put the indices of all the LMS-suffixes into their buckets. In this step, we record the end of each bucket, and the LMS-suffixes are put into the bucket from the end to the head. Hence, in the bucket for "i," we put the suffixes first 2, next 6, and last 10. Now, the sentinel, which is the only single-character LMS-prefix, is sorted to its correct position 0 in SA.
- Step 2: All the L-type LMS-prefixes are induced sorted in this step. We first find the head of each bucket. The current head of a bucket is marked by the symbol "$\wedge$" under the bucket. Now, we scan SA from left to right, for which the current item of SA being visited is marked by the symbol "@." When we are visiting $SA[0] = 16$ in line 10, we check the type array t to know $S[15] = i$ is L-type. Hence, 15 is appended to the current head of bucket for "i," and the bucket's head is forwarded one step to the right. In line 15, the scanning reaches $SA[2] = 14$, and see that $S[13] = p$ is L-type, then we put 13 to the current head of bucket for "p," and forward the bucket's head one step to the right. To repeat scanning SA in this way, we can get all the L-type LMS-prefixes and the sentinel sorted in SA, as shown in line 28, where a symbol "$\wedge$" between two buckets means that the left bucket is fully filled by L-type LMS-prefixes.
- Step 3: In this step, we induced sort all the 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, then put 3 to the current end of bucket for "i" and forward the bucket's end one step to the left. When we visit the next character, i.e., $S[15] = 8$, we see $S[7] = i$ is S-type, then we put 7 to the current end of bucket for "i," and forward the bucket's end one step to the left. Notice that the LMS-prefixes 3 and 7 overwrote the LMS-suffixes 2 and 6 that were formerly stored in the bucket by the first step, respectively. To repeat scanning SA in this way, all the LMS-prefixes are sorted in their order shown in line 44. (Notice that the sentinel was put into its bucket in the first step, and will not be overwritten by any character in this step, for, it is the last character in the string.)
- Given all the LMS-prefixes are sorted in SA, we scan SA once from left to right to compute the name for each LMS-substring starting from 0, where the order of any two neighboring LMS-substrings in SA is determined by comparing the lexicographical values and types of their characters one by one, using Definition 3.3. As a result, we get the shortened string S_1 shown in line 46, where the names for the LMS-substrings 2, 6, 10, and 16 are 2, 2, 1, 0, respectively.

3.6 Complexity Analysis for SA-IS

Theorem 3.13. *(Time/Space Complexities) Given S is of a constant or integer alphabet, the time and space complexities for the algorithm SA-IS in Fig. 4 to compute $SA(S)$ are $O(n)$ and $O(n \log n)$ 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 $O(n \log n)$ bits. □

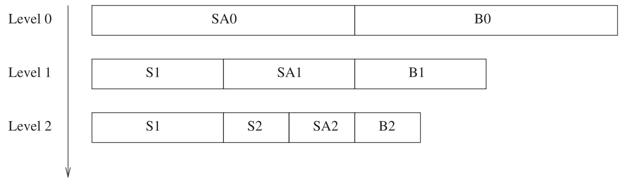

Fig. 3. The worst-case space requirement for the SA-IS algorithm at each recursion level.

To investigate the accurate space requirement, we show in Fig. 3 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 latter 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 be reused 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.

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, respectively. 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 observed at the top level. However, if the alphabet of S is constant, B_0 and B_1 are $O(1)$ and $O(n)$, respectively, resulting in the maximum space required by the second level when n increases. Hence, we have the space requirement as following, where n bits in both cases are counted for the type arrays:

Corollary 3.14. *The worst-case working space requirements for SA-IS in Fig. 4 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 space requirement of the algorithm in practice, we have the below a probabilistic result.

Theorem 3.15. *Given the probabilities for each character to be S-type or L-type are i.i.d as 1/2, the mean size of a nonsentinel LMS-substring is four, i.e, the reduction ratio is not greater than 1/3.*

Proof. Let us consider a nonsentinel LMS substring $S[i..j]$, where $i < j$. From the definition of LMS-substring, we know that this substring must contain two LMS characters: one is the head, and other is the end. Moreover, there must be at least one L-type character $S[k]$ in between $S[i]$ and $S[j]$. Given the i.i.d probability of $1/2$ for each character to be S-type 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 four. Because all the LMS-substrings are located consecutively, the end of one is also the head of another succeeding. This implies that the mean size of a nonsentinel LMS-substring excluding its end is three, resulting in the reduction ratio not greater than $1/3$. □

```
SA-DS(S, SA)
    ▷ S is the input string;
    ▷ SA is the output suffix array of S;
    t: array [0..n − 1] of boolean;
    P1, S1: array [0..n1] of integer; ▷ n1 = ||S1||
    B: array [0..||Σ(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 d-critical substrings
    in S into P1;
    Bucket sort all the d-critical substrings using P1 and B;
    Name each d-critical substring in S by its bucket
    index to get a new shortened string S1;
    if ||S1|| = Number of Buckets
        then
            Directly compute SA1 from S1;
        else
            SA-DS(S1, SA1); ▷ Fire a recursive call
    Induce SA from SA1;
    return
```

Fig. 4. The SA-DS algorithm framework.

This theorem together with Fig. 3 implies that, if the probabilities for a character in S to be S-type or L-type are equal and the alphabet of S is constant, the maximum space for our algorithm is contributed to level 1, where $|S_1| \leq n/3$. Hence, the maximum working space is determined by the type arrays, which is $n + O(1)$ bits in the worst case. As we will see in Section 5, this theorem well approximates the results on realistic data.

4 Algorithm II: Radix Sorting Fixed-Length D-Critical Substrings

In this section, the second proposed algorithm called SA-DS for linear time suffix array construction is presented. We first introduce the concept of d-critical character, which builds the basis of the SA-DS algorithm.

4.1 Critical Character

Definition 4.1. *(Critical Character/Suffix) A character $S[i]$ is said to be d-critical, where $d \geq 2$, if and only if 1) $S[i]$ is a LMS-character; or else 2) $S[i-d]$ is a d-critical character, $S[i+1]$ is not an LMS-character and no character in $S[i-d+1..i-1]$ is d-critical. A suffix $suf(S,i)$ is called d-critical if $S[i]$ is a d-critical character.*

For notation convenience, let $d_1 = d + 1$ for the rest of this section.

Definition 4.2. *(Neighboring Critical Characters) A pair of d-critical characters $S[i]$ and $S[j]$ are said to be two neighboring d-critical characters in S, if there is no other d-critical character in between them.*

Definition 4.3. *(Critical Substring) The substring $S[i..i+d_1]$ is said to be the d-critical substring for the d-critical character $S[i]$ in S. For $i \geq n - d_1$, $S[i..i+d_1] = S[i..n-2]\{S[n-1]\}^{d_1-(n-2-i)}$, where $\{S[n-1]\}^x$ denotes that $S[n-1]$ is repeated x times.*

To simplify the discussion, we use $\Psi_{C-d}(S)$ to denote the d-critical substring array for S, which contains all the d-critical substrings in S, one substring per item, consecutively arranged according to their original positional order in S. From the above definitions, we have the following immediate observations:

Proposition 4.4. *In S, 1) every LMS character is a d-critical character; and 2) the last character must be a d-critical character, and the first character must not be a d-critical character.*

Proposition 4.5. *Given $S[i]$ is a d-critical character, both $S[i-1]$ and $S[i+1]$ are not d-critical characters.*

Lemma 4.6. *The distance between any two neighboring d-critical characters $S[i]$ and $S[j]$ in S must be in $[2, d_1]$, i.e., $j - i \in [2, d_1]$, where $d \geq 2$ and $i < j$.*

Proof. From Proposition 4.5, given $S[i]$ is a d-critical character, $S[i+1]$ must not be a d-critical character. In other words, the first d-critical character on the right hand of $S[i]$ may be any in $S[i+2, i+d_1]$, but must not be $S[i+1]$. □

4.2 Algorithm Framework

Our linear time suffix array sorting algorithm SA-DS is outlined in Fig. 4. 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. In the rest of this section, we further describe in more details about the operations in each step.

4.3 Reducing the Problem

With the concept of d-critical character/suffix, here comes the key idea to reduce the problem into another that is at least half smaller. First, we introduce an integer array P_1 to maintain the pointers for all the sampled d-critical substrings for reducing the problem.

Definition 4.7. *(Sample Pointer Array) The array P_1 contains the sample pointers for all the d-critical substrings in S preserving their original positional order, i.e., $S[P_1[i]..P_1[i]+d_1]$ is a d-critical substring.*

From the definitions of P_1 and Ψ_{C-d}, immediately we have $\Psi_{C-d} = \{S[P_1[i]..P_1[i]+d_1] | i \in [0, n_1)\}$, where n_1 denotes the size (or cardinality) of Ψ_{C-d}. Hereafter, we simply consider P_1 at pointer level, but the underneath comparisons for its items lie in the substrings in Ψ_{C-d}. Provided with the type array t (defined in Section 2), we can traverse t once from left to right to compute P_1 in $O(n)$ time.

Definition 4.8. *(Siblings) $P_1[i]$ and $S[P_1[i]..P_1[i]+d_1]$ are said as a pair of siblings.*

Let $\omega(S, i)$ be the ω-weighting function of $S[i]$, defined as $\omega(S, i) = 2S[i] + t[i]$, and let S_ω denote the ω-weighted string of S, where $S_\omega[i] = \omega(S, i)$. Now, bucket sort all the items of P_1 by their ω-weighted siblings (i.e., $S_\omega[P_1[i]..P_1[i]+d_1]$ for $P_1[i]$) in increasing order. Then name each item of P_1 by the index of its bucket to produce a string S_1, where all the buckets are indexed from 0. Here, we have the following observations on S_1:

Lemma 4.9. *(Sentinel) The last character of S_1 must be the unique smallest character in S_1.*

Proof. From Proposition 4.4, we know that $S[n-1]$ must be a d-critical character, and the d-critical substring starting at $S[n-1]$ must be the unique smallest among all sampled by P_1. □

Lemma 4.10. *(1/2 Reduction Ratio) $\|S_1\|$ is at most half of $\|S\|$, i.e., $n_1 \leq \lfloor n/2 \rfloor$.*

Proof. From Proposition 4.4, $S[0]$ must not be a d-critical character. We know from Lemma 4.6 the distance between any two neighboring d-critical characters is at least two, which immediately completes the proof. □

The above two lemmas state that, S_1 is at least half smaller than S and terminated by an unique smallest sentinel too.

Theorem 4.11. *(Coverage) For any two characters $S_1[i] = S_1[j]$, 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) $t[P_1[i]..P_1[i+1]] = t[P_1[j]..P_1[j+1]]$. Given 1) and 2) are satisfied, let $i' = P_1[i] + 1$ and $j' = P_1[j] + 1$. We have the below observations:

- Any character in $S[i'..i'+d_1]$ is an LMS character. In this case, given $S_1[i] = S_1[j]$, we must have $P_1[i+1] = P_1[j+1]$.
- No character in $S[i'..i'+d_1]$ is an LMS character. In this case, both $i' + d$ and $j' + d$ must be in P_1.

In either case, we have $P_1[i+1] - P_1[i] = P_1[j+1] - P_1[j]$. □

Theorem 4.12. *(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 considerations for the following two cases:

- Case 1: $S_1[i] \neq S_1[j]$. In this case, it is trivial to see that the statement is 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 conducted on $S_1[i+1] = S_1[j+1]$, $S_1[i+2] = S_1[j+2], \ldots S_1[i+k-1] = S_1[j+k-1]$ until a k is reached that makes $S_1[i+k] \neq S_1[j+k]$. Because that $S_1[i..i+k-1] = S_1[j..j+k-1]$, from Theorem 4.11, we must have $P_1[i+k] - P_1[i] = P_1[j+k] - P_1[j]$, i.e., the substrings $S[P_1[i]..P_1[i+k]]$ and $S[P_1[j]..P_1[j+k]]$ are of the same length. This suggests that sorting $S_1[i..i+k]$ and $S_1[j..j+k]$ is equal to sorting $S[P_1[i]..P_1[i+k]+d_1]$ and $S[P_1[j]..P_1[j+k]+d_1]$. Hence, the statement is correct in this case, too. □

This theorem suggests that in order to find the orders for all the d-critical suffixes in S, we can sort S_1 instead. Because the size of S_1 is at most 1/2 of that of S, the computation on S_1 can be done within about one half the complexity for S. In the following sections, we will show

how to bucket sort and name the items of P_1, i.e., the two crucial subtasks of computing S_1.

4.4 Sorting and Naming P_1

To bucket sort and name all the items of P_1, intuitively, we need at least three integer arrays of at most $2n_1 + n$ integers in total: two arrays of size n_1 used as the alternating buffers for bucket sorting P_1, and another of size n for storing the bucket pointers, where $2n_1 \le n$. The array of bucket pointers needs to be of size n, because each character of P_1 is in the range $[0, n-1]$. The space needed for sorting P_1 constitutes the space bottleneck for our algorithm. To further improve the space efficiency, we can use the following γ-weighting scheme for bucket sorting P_1 instead.

Definition 4.13. *(γ-Weighted Substring) The γ-weighted substring $S_\gamma[i..j]$ in S is defined as $S_\gamma[i..j] = S[i..j-1]S_\omega[j]$.*

For any two γ-weighted substrings, we have the result below.

Lemma 4.14. *Given $S_\gamma[i..i+k] < S_\gamma[j..j+k]$ and $S[i..i+k] = S[j..j+k]$, we must have $t(S, i+x) \le t(S, j+x)$ for any $x \in [0, k]$.*

Proof. From the given condition, there must be $t(S, i+k) < t(S, j+k)$. If $S[i+k-1] = S[i+k]$, we must have $t(S, i+k-1) = t(S, i+k)$ and $t(S, j+k-1) = t(S, j+k)$, i.e., $t(S, i+k-1) < t(S, j+k-1)$. If $S[i+k-1] \neq S[i+k]$, because $S[i+k-1] = S[j+k-1]$, we must have $t(S, i+k-1) = t(S, j+k-1)$. Hence, in both cases, $t(S, i+k-1) \le t(S, j+k-1)$. The proof is completed by applying the analogous arguments to $t(S, i+k-2)$, $t(S, i+k-3) \ldots$, and $t(S, i)$. □

By replacing $S_\omega[i..j]$ with $S_\gamma[i..j]$ as the weight of $P_1[i]$ for bucket sorting P_1 to produce S_1, we have the following result.

Theorem 4.15. *(γ-Order Equivalence) 1) Given $S_\gamma[P_1[i]..P_1[i] + d_1] = S_\gamma[P_1[j]..P_1[j] + d_1]$, there must be $S_\omega[P_1[i]..P_1[i] + d_1] = S_\omega[P_1[j]..P_1[j] + d_1]$; and 2) given $S_\gamma[P_1[i]..P_1[i] + d_1] < S_\gamma[P_1[j]..P_1[j] + d_1]$, there must be $S_\omega[P_1[i]..P_1[i] + d_1] < S_\omega[P_1[j]..P_1[j] + d_1]$.*

Proof. Let $i' = P_1[i]$ and $j' = P_1[j]$. If $S_\gamma[i'..i'+d_1] = S_\gamma[j'..j'+d_1]$, we must have $S[i'..i'+d_1] = S[j'..j'+d_1]$ and $t(S, i'+d_1) = t(S, j'+d_1)$, i.e., $S_\omega[i'..i'+d_1] = S_\omega[j'..j'+d_1]$. Further, if $S_\omega[i'+d_1] = S_\omega[j'+d_1]$ and $S[i'+d] = S[j'+d]$, we must have $t(S, i'+d) = t(S, j'+d)$ as well as $S_\omega(i'+d) = S_\omega(j'+d)$, and so on for the other characters in the two substrings. Therefore, we must have $S_\omega[i'..i'+d_1] = S_\omega[j'..j'+d_1]$. When $S_\gamma[i'..i'+d_1] < S_\gamma[j'..j'+d_1]$, we consider these two cases:

- If $S[i'..i'+d_1] \neq S[j'..j'+d_1]$, given $S_\gamma[i'..i'+d_1] < S_\gamma[j'..j'+d_1]$, there must be $S[i'..i'+d_1] < S[j'..j'+d_1]$ from the definition of γ-weighted substring (Definition 4.13), which yields $S_\omega[i'..i'+d_1] < S_\omega[j'..j'+d_1]$ from the definition of S_ω.
- If $S[i'..i'+d_1] = S[j'..j'+d_1]$, we must have $t(S, i'+d_1) = 0$ and $t(S, j'+d_1) = 1$. Further, from Lemma 4.14, we have $t(S, i'+x) \le t(S, j'+x)$ for any $x \in [0, d_1]$, resulting in $S_\omega[i'..i'+d_1] < S_\omega[j'..j'+d_1]$.

Hence, we complete the proof. □

Theorem 4.15 suggests that, to determine the order of two ω-weighted d-critical substrings, we can use their γ-weighted counterparts instead. As a result, we need to compare the characters' types only for the last characters of two d-critical substrings. Therefore, sorting all the items of P_1, according to the last characters of their γ-weighted siblings, can be decomposed into two passes in sequence: 1) bucket sort according to the types of these characters; and 2) bucket sort according to the characters themselves. Notice that the sorting of all the γ-weighted substrings is not required to be stable, hence we can use a fast method to sort the last characters of these substrings. In Step 1, there are only two buckets, one for the L-type characters and the other for the S-type characters. This naturally suggests that Step 1 can be done by traversing all the characters only once to examine their L/S-types and put them into their buckets accordingly.

To bucket sort the γ-weighted substrings, we only need an array of $\Sigma(S)$ or n_1 integers to maintain the bucket information at the first or second iterations, respectively. Now, provided with P_1, t, and S, we can compute S_1, i.e., the reduced problem, using the two-step algorithm described below:

- Step 1: Bucket sort all the elements of P_1 into another array P_1' by their corresponding siblings (i.e., fixed-size d-critical substrings) in S, with $\Sigma(S)$ buckets. The sorting is done through $d+2$ passes, in a manner of least-significant-character-first. This step requires a time complexity of $O(dn_1) = O(n_1)$, for $d = O(1)$.
- Step 2: Compute the names for all the elements in P_1' (as well as P_1). This job can be done by a simple algorithm described as following:
  - allocate an array tmp of size n, where each item is an integer in $[0, n-1]$,
  - initialize all the items of tmp to be -1,
  - scan P_1' once from left to right to compute all the names for the items of P_1', by setting $tmp[P_1'[i]]$ with the index of bucket that $P_1'[i]$ belonging to, and
  - pack all the nonnegative elements in tmp into the buffer of P_1', by traversing tmp once. Now, the buffer of P_1' stores the string of S_1.

One problem with Step 2 in the above algorithm is that, in addition to P_1' and S_1, it uses a large space of n integers (each integer is of $\lceil \log n \rceil$ bits) for tmp. Alternatively, we can use another space-efficient algorithm for this job by reusing tmp for P_1' and S_1, described as following. Let us define a logical array $tmp_e = \{tmp[i] | i\%2 = 0\}$ for the first n_1 even items of tmp, where tmp_e is said to be a logical array for its physical buffer is distributed into the first n_1 even items of tmp, i.e., its physical buffer is not spatially continuous.

Suppose that P_1' is initially stored in the first n_1 items of tmp, we first copy P_1' into tmp_e and set $tmp[j] = -1$ for any $tmp[j] \notin tmp_e$, i.e., distribute P_1' into the first even items of tmp. Next, we scan tmp_e from left to right to compute the names for all the items of tmp_e. For each $tmp_e[i]$, we record its name as following: 1) if $tmp_e[i]$ is even, set $tmp[tmp_e[i]-1]$ with the name, or else, set $tmp[tmp_e[i]]$ with the name. Now, all the items of S_1 are stored in the nonnegative odd items of tmp in their correct relative positional orders. At last, we traverse tmp once to compact all the nonnegative odd items into S_1. Using this method for Step 2, tmp is reused for accommodating both P_1' and S_1, resulting in that only one n-integer array is required for storing them.

4.5 Inducing SA from $SA(S_1)$

Once again, let SA_1 be the suffix array of S_1. The algorithm for inducing SA from SA_1 in SA-DS is similar to the algorithm A3.3 in Section 3.3, different only in the first step as shown below:

A4.5 Algorithm for Inducing SA from SA_1 in SA-DS

1. Initialize each item of SA as -1. Find the end of each bucket in SA for all the suffixes in S. Scan SA_1 once from right to left, if $suf(S, P_1[SA_1[i]])$ is an LMS suffix then put $P_1[SA_1[i]]$ to the current end of the bucket for $suf(S, P_1[SA_1[i]])$ in SA and forward the bucket's end one item to the left.
2. The same as Step 2 in the algorithm A3.3.
3. The same as Step 3 in the algorithm A3.3.

Let us consider the correctness of the above algorithm. Notice that an LMS-suffix is also a d-critical suffix, then all the LMS-suffixes of S must be sampled in S_1, and hence ordered in SA_1. Therefore, the first steps of algorithms A.4.5 and A3.3 are equivalent in the sense that they will fill the array SA with all the LMS-suffixes of S identically. That is, the resulting SA for these two steps are the same. Because the last two steps in both algorithms are exactly the same and the algorithm A3.3 can induced sort SA from SA_1, the algorithm A.4.5 must do the same job too.

4.6 Example

To help readers grasp the core idea of the proposed algorithm, in Fig. 5, we have dumped the intermediate status of the data structures used in our SA-DS algorithm with $d = 2$ when it runs on a string $S = \texttt{mmiissiissiippii\$}$, where $ is the sentinel.

In this example, our algorithm uses only two levels of recursions, i.e., the recursion depth is two. For each recursion, the algorithm starts from sampling all the d-critical characters into P_1, then proceeds to bucket sort all the elements of P_1 by their corresponding γ-weighted siblings (2-critical substrings in S), which is done by $d + 2 = 4$ passes of bucket sort. The result for each pass is shown one after another in the figure, where the sorting is not stable. Having sorted P_1, the names for all the items of P_1 are computed, resulting in the reduced string S_1. Further, we recursively compute $SA(S_1)$ and then induce $SA(S)$ from it.

```
Recursion level 0:
                               1
 Index: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
     S: m m i i s s i i s s i i p p i i $
     t: 0 0 1 1 0 0 1 1 0 0 1 1 0 0 0 0 1
    P1: 2 4 6 8 10 12 14 16
Bucket sorting and naming P1:
Pass 1: 14 16 12  4  8 10  2  6
Pass 2: 14 16 12  4  8 10  2  6
Pass 3: 16 14 10  2  6 12  4  8
Pass 4: 16 14 10  2  6 12  4  8
    S1:  3  5  3  5  2  4  1  0

Recursion level 1:
                               1
 Index: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
     S: 3 5 3 5 2 4 1 0
     t: 1 0 1 0 1 0 0 1
    P1: 2 4 7
Bucket sorting and naming P1:
Pass 1: 4 7 2
Pass 2: 7 4 2
Pass 3: 7 4 2
Pass 4: 7 4 2
    S1: 2 1 0
   SA1: 2 1 0

Recursion ends

Recursion level 1:
Inducing SA0 from SA1:
   SA1: 2  1  0
Step 1: 7 -1  4 -1  2 -1 -1 -1
Step 2: 7  6  4 -1  2  5  3  1
Step 3: 7  6  4  2  0  5  3  1

Recursion level 0:
Inducing SA0 from SA1:
   SA1:  7  6  4  2  0  5  3  1
Step 1: 16 -1 -1 -1 -1 -1 10  6  2 -1 -1 -1 -1 -1 -1 -1 -1
Step 2: 16 15 14 -1 -1 -1 10  6  2  1  0 13 12  9  5  8  4
Step 3: 16 15 14 10  6  2 11  7  3  1  0 13 12  9  5  8  4
```

Fig. 5. A running example for the SA-DS algorithm.

4.7 Practical Strategies

We propose several techniques to further improve the time/space efficiencies of our SA-DS algorithm in practice. Without loss of generality, we assume a 32-bit machine and each integer consumes 4 bytes.

4.7.1 General Strategy: Reusing the Buffer for $SA(S)$

From the algorithm framework in Fig. 4, we see that the algorithm consists of three steps in sequence: 1) sorting P_1, 2) naming all the items of P_1 to obtain S_1, and 3) inducing $SA(S)$ from $SA(S_1)$. Notice that $SA(S)$ is an array of n integers, and both P_1 and S_1 have n_1 integers, where $2n_1 \leq n$, we can reuse the buffer for $SA(S)$ for the first two steps too.

4.7.2 Strategy 1: Storing the LS-Type Array

Each element of the LS-type array for S is 1-bit, and a total of at most $n(1 + 1/2 + 1/4 + ... + \log^{-1} n) < 2n$ bits are required by the LS-type arrays for all the recursions. Hence, we can use the two most-significant-bits (MSBs) of $SA(S)[i]$ for storing the L/S-type of $S[i]$. Recalling that the space for each integer is allocated in units of 4-byte instead of bits, the two MSBs of an integer are always available for us in this case. This is because in computing $SA(S)$, our algorithm running on a 32-bit machine that requires at least $5n$ bytes, where $4n$ for the items (each is a 4-byte integer) in $SA(S)$ and n for the input string (usually one byte per character). Therefore, the maximum size n_{max} of the input string must satisfy $5n_{max} < 2^{32}$, resulting in $n_{max} < 2^{32}/5$ and $\log n_{max} < 30$. In order words, 30-bits are enough for each item of $SA(S)$. However, for implementation convenience, we can simply store the LS-type arrays, using bit arrays of maximum $2n$ bits in total, i.e., $0.25n$ bytes.

4.7.3 Strategy 2: Bucket Sorting P_1

Given the buffers for P_1 and S_1. To bucket sort P_1, we can use another array B in Fig. 4 for maintaining the buckets, where the size of B is determined by the alphabet size of the input string S. Even the original input string S is of a constant alphabet. After the first iteration, we will have S_1 as the input string for the next iteration. Since S_1 has an integer alphabet that can be as large as n_1 in the worst case, B may require a maximum space up to $n_1 \leq \lfloor n/2 \rfloor$ integers.

To prevent B from growing with n_1, instead of sorting characters—each character is of 4 bytes—in each pass of bucket sorting the d-critical substrings, we simply sort each character with two passes, i.e., the bucket sorting is performed on units of 2-byte. The time complexity for bucket sorting all the fixed size d-critical substrings at each iteration is linearly proportional to the total number of characters for these substrings. Since each d-critical substring is of $d+2$ characters and the number of substrings decreases at least half per iteration, the total number of characters sorted at all the iterations is upper bounded by $O((d+2)(1/2+1/4+...+\log^{-1}n)) = O(dn)$, which is $O(n)$ given $d = O(1)$. Hence, the time complexity for bucket sorting in this way remains linear $O(n)$. For $n \leq 2^{32}$, the entire bucket sorting process will be half slowed down. However, the space for B can be fixed to 65,536 integers, i.e., $O(1)$. When $n > 2^{32}$, despite the size of each integer is increased, the same idea can also be applied. In respect to whether the alphabet of S is constant or integer, the peak space requirement for bucket sorting in the whole algorithm will occur as below:

- For S originated from a constant alphabet, the peak space occurs when further reducing S_1 at the second iteration, which requires an extra space of n_1 integers, where each integer is of $\lceil \log n_1 \rceil$ bits. In this case, we can bucket sort on units of $\lceil \lceil \log n_1 \rceil /2 \rceil$ bits.
- For S originated from an integer alphabet, the peak space occurs when reducing S at the 1st iteration, which requires an extra space of n integers, each integer of $\lceil \log n \rceil$ bits. In this case, we can bucket sort on units of $\lceil \lceil \log n \rceil /2 \rceil$ bits.

In both cases, given $n > 2^{32}$, the required extra spaces in the worst case are not more than $1/2^{16}$ of the spaces for their suffix arrays, respectively, and thus negligible. Hence, in summary, bucket sorting for problem reduction at each iteration can always be done using an extra working space of $O(1)$ only, independent of n.

4.7.4 Strategy 3: Inducing the Final Result

In the inducing algorithm described above, a buffer B is needed for dynamically recording the current head/end of each bucket. However, in order to save more space, we can use an alternative inducing algorithm, which requires only the buffer for $SA(S_1)$ and needs no B when inducing $SA(S_1)$. This idea is to name the elements of P_1 in a different way: once all the items of P_1 have been sorted into their buckets, we can name each item of P_1 by the end[1] of its bucket to produce S_1. To be more precise, this is because the MSB of each item in SA_1 and S_1 is unused (when the strategy 1 is not applied). Given that each item of S_1 points to the end of its bucket in the array of SA_1, the inducing can be done in this way: when an empty bucket in SA_1 is inserted the first item $S_1[i]$ at $SA_1[j]$, we set $SA_1[j] = i$ and mark the MSB of $SA_1[j]$ by 1 to indicate that $SA_1[j]$ and $S_1[i]$ are borrowed for maintaining the bucket end. At the end of each inducing stage, we can restore the items in S_1 and SA_1 to their correct values in this way: scan SA_1 from left to right, for each $SA_1[i]$ with its MSB as 1, let $S_1[SA[i]] = i$ and reset the MSB of $SA_1[i]$ as 0.

1. We can also use the head of its bucket instead.

4.8 Complexity Analysis for SA-DS

Theorem 4.16. *(Time/Space Complexities) Given S is of a constant or integer alphabet, the time and space complexities for the algorithm SA-DS in Fig. 4 to compute $SA(S)$ are $O(n)$ and $O(n \log n)$ 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 obvious $O(n \log n)$ bits, for the size of each array used at the first iteration is upper bounded by $n \lceil \log n \rceil$ bits, and decreases at least a half for each iteration thereafter. □

Corollary 4.17. *(Working Space) The SA-DS algorithm can construct the suffix array for a size-n string S with a constant or integer alphabet, using $O(n)$ time and a working space of only $0.25n + O(1)$ bytes, where the characters of the integer alphabet are in $[0..n-1]$.*

Proof. The key technique is to design the SA-DS algorithm with the general strategy and the strategies 2-3 in Section 4.7. Naturally, we can allocate an LS-type array at each iteration, which requires in total a space of $2n$ bits for the type arrays at all the iterations. However, the $2n$ bits can be further reduced to n bits by trading with time as following. At each iteration, before going to the next iteration, we release the type array for the current iteration; after returning from the next iteration, we can scan the string (of the current iteration) once to reproduce the type array for inducing the final result for the current iteration.

Despite $\Sigma(S)$ is constant or integer, after the first iteration, the SA-DS algorithm will work on the shortened strings of integer alphabets. In other words, for all the iterations except the first iteration, the SA-DS algorithm will consume the same space, no matter $\Sigma(S)$ is constant or integer. Hence, in respect to $\Sigma(S)$, we consider the following two cases at the first iteration:

- Constant alphabet: In this case, we can use an array of size $O(1)$ to maintain the bucket for inducing the final result at the first iteration, i.e., the strategy 3 is not applied at the first iteration. As a result, the least working space can be $0.125n + O(1)$ bytes.
- Integer alphabet: In this case, before the first iteration, we bucket sort all the characters of S and rename each character of S to be the end of its bucket. Under the assumption that $\Sigma(S)$ is in $[0..n-1]$, this can be done in $O(n)$ time and using only the space of $SA(S)$ plus $O(1)$. Then we execute the SA-DS algorithm to compute $SA(S)$ recursively. After returning from the second iteration, in addition to the n-bit LS-type array, we allocate one more array of n bits, one bit for use with each item of the array $SA(S)$. This n-bit array is used in combination with the array $SA(S)$ and S to apply the strategy 3. Hence, the working space is $0.25n + O(1)$ bytes.

TABLE 1
Data Used in the Experiments

Data	Characters, $\|\Sigma\|$, Description
bible.txt	4047392, 63 , King James Bible
chr22.dna	34553758, 4, Human chromosome 22
E.coli	4638690, 4, Escherichia coli genome
etext99	105277340, 146, Texts from Gutenberg project
howto	39422105, 197, Linux Howto files
pic	513216, 159, Black and white fax picture
sprot34.dat	109617186, 66, Swissprot V34 protein database
world192.txt	2473400, 94, CIA world fact book
alphabet	100000, 26, Repetitions of the alphabet [a-z]
random	100000, 64, Randomly selected from 64 characters

The peak space requirement of the whole algorithm occurs when inducing the final result at the first iteration. Hence, a working space of $0.25n + O(1)$ bytes is sufficient. □

We have coded in C a sample implementation for approaching the results stated in Corollary 4.17, i.e., the DS2 algorithm in the experiment section.

5 EXPERIMENTS

The algorithms investigated in our experiments are KS, KA and our algorithms IS, DS1, and DS2, where IS is the SA-IS algorithm, DS1 and DS2 are two variants of the SA-DS algorithm trading off differently between space and time, with $d = 3$ and enhanced by the practical strategies proposed in Section 4.7. The algorithms DS1 and DS2 use different settings of strategies: DS1 uses the general strategy only, whereas DS2 uses the strategies 2 and 3, in addition to the general strategy. Specifically, for $d = 3$, each substring sorted by the DS1 and DS2 algorithms has a fixed length of five characters, we sort the substrings at the first iteration in three passes, using a bucket of 65,536 integers (instead of sorting in five passes with a bucket of 256 integers). The performance measurements to be investigated are the time/space complexities, recursion depth, and mean reduction ratio.

The data sets in Table 1 used in our experiment were downloaded from the popular benchmark repositories for SACAs, including the Canterbury [20] and Manzini-Ferragina [6] corpora. These data sets are of constant alphabets with sizes smaller than 256, and one byte is consumed by each character. Among them, only the last two files "alphabet" and "random" are artificial. The experiments were performed on a machine with AMD Athlon(tm) 64x2 Dual Core Processor 4200+ 2.20GHz and 2.00GB RAM, and the operating system was Linux (Sabayon Linux distribution).

All the algorithms were implemented in C++ and compiled by g++ with the option of -O3. The KS algorithm was downloaded from Sanders's website [21]. For the KA algorithm, we use an improved version from Yuta Mori[2] for the original KA code (at Ko's website [22]). Our algorithms IS, DS1, and DS2 were embodied in less than 100, 150, and 250 effective lines of code, respectively; all are available on request.

2. The reason for us to use this improved version instead was that the original KA code was observed to cause segment faults or simply go dead when testing on files "howto" and "etext99," and Mori's version is the only robust implementation of the KA algorithm that we could obtain to complete our experiments. Please notice that according to one external reviewer, for all inputs Mori's implementation performed better than other known versions of the KA algorithm.

TABLE 2
Time

Data	Time (Seconds)				
	IS	DS1	DS2	KS	KA
bible	**2.7**	3.11	3.9	8.9	3.62
chr22	**24.7**	31.5	39.6	92.8	34.1
E.coli	**2.8**	3.53	4.3	10	3.98
etext	**101**	123.2	150.4	428.1	149.67
howto	**30.4**	36.3	44.05	130.4	42.85
pic	**0.06**	0.09	0.13	0.56	0.29
sprot	**94.6**	111.59	139.6	356	132.91
world	**1.3**	1.61	2	4.8	1.84
alphabet	**0.00**	0.01	0.02	0.15	0.02
random	0.02	**0.01**	**0.01**	0.06	0.02
Total	**257.58**	310.95	384.01	1031.77	369.3
Mean	**0.90**	1.08	1.34	3.60	1.29
Norm.	**1**	1.21	1.49	4.01	1.43

TABLE 3
Space

Data	Space (MBytes)				
	IS	DS1	DS2	KS	KA
bible	20.86	21.50	**20.30**	90.40	34.45
chr22	178.09	184.44	**171.41**	819.25	289.97
E.coli	24.29	25.15	**23.23**	105.93	40.01
etext	542.17	559.55	**521.85**	2369.92	907.34
howto	203.16	208.08	**195.55**	932.07	331.54
pic	**2.57**	2.76	2.79	15.51	3.11
sprot	554.58	560.44	**543.26**	2591.62	930.06
world	12.70	12.91	**12.50**	55.24	21.24
alphabet	**0.49**	0.74	0.75	3.03	0.52
random	**0.61**	0.74	0.74	2.26	0.88
Total	1539.52	1576.31	**1492.37**	6985.23	2559.12
Mean	5.37	5.50	**5.20**	24.36	8.92
Norm.	1.03	1.06	**1**	4.68	1.72

5.1 Time and Space

The time for each algorithm is the mean of three runs, and the space is the heap peak measured by using the `memusage` command to start 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 consumed by running the algorithm for all the input data, respectively. The mean time (measured in seconds per MBytes) and space (in bytes per character of the input string) for each algorithm are the total time and space divided by the total number of characters.

Tables 2 and 3 show the statistic time and space results collected from the experiments, respectively, where the best results are typeset in the bold fonts. For comparison convenience, we also normalize all the results by the best results. In the program for the KS algorithm, each character of the input string S is stored as a 4-byte integer, and the buffer for $SA(S)$ is not reused for the others.[3] For a more accurate comparison, we subtract $7n$ bytes from the space results measured for the KS algorithm in the experiments, since we are sure $7n$ space can be trivially saved using some engineering tricks.

From these two tables, we see that all the best time and space performances are achieved by our IS and DS2 algorithms, respectively. Specifically, in average, the IS algorithm is three times (300 percent) faster than the KS, and 43 percent faster than the KA. The mean space of

3. Notice that there exists a prominent discrepancy for the KS algorithm between its theoretical analysis and the results from its implementation in the experiment. As for this discrepancy, we are aware that this implementation might aim at achieving the best time complexity by pushing the space complexity to its extreme.

TABLE 4
Recursion Depth and Reduction Ratio

Data	Depth				Ratio			
	IS	DS	KS	KA	IS	DS	KS	KA
bible	**6**	**6**	**6**	7	**.34**	.37	.67	.46
chr22	**6**	10	12	9	**.31**	.36	.67	.44
E.coli	**7**	8	**7**	9	**.32**	.36	.67	.45
etext	**11**	12	12	15	**.33**	.37	.67	.45
howto	**9**	10	11	13	**.32**	.36	.67	.45
pic	**5**	9	10	**5**	**.26**	.35	.67	.39
sprot	**7**	8	9	10	**.31**	.37	.67	.45
world	**6**	7	**6**	**7**	**.32**	.37	.67	.45
alphabet	**2**	10	11	**2**	**.02**	.34	.67	**.02**
random	2	**1**	2	2	**.33**	.36	.67	.47
Total	**61**	81	86	80	**2.86**	3.61	6.7	4.03
Mean	**6.1**	8.1	8.6	8.0	**.29**	.36	.67	.40
Norm.	**1**	1.33	1.41	1.31	**1**	1.26	2.34	1.38

$24.3n$ for the KS algorithm in our experiments is about twice of the 10-$13n$ for another space efficient implementation of the KS algorithm by Puglisi [5]. Even assuming the better 10-$13n$ space, the KS algorithm still uses a space more than twice of that used by any of our algorithms. The KA algorithm in our experiments is more time and space efficient than the KS algorithm, this observation agrees with the observations from the others [5], [17], however, which still uses over 67 percent more space than ours.

In the space table, we see that DS1 and DS2 use more space than IS does for the small files "pic," "alphabet," and "random." This is due to the bucket of 65,536 integers used at the first iteration, i.e., 262,144 bytes. The size of this bucket is constant for any input string, and thus can be counted as $O(1)$. If this bucket is deducted from the total space consumption, the space used by DS1 and DS2 for these three files are around $5.2n$ bytes too, which is well coincided with the earlier analysis.

5.2 Recursion Depth and Reduction Ratio

Table 4 shows the recursion depths and problem reduction ratios. These results are machine-independent and deterministic for the given input strings. The recursion depth is defined as the number of iterations, and the mean reduction ratio is the sum of reduction ratios for all iterations divided by the number of iterations. Obviously, for the reduction ratio, the smaller, the faster and better. For an overall comparison, we also give the total for the recursion depth and reduction ratio for each algorithm and the means for both, where the former is the sum of all corresponding results and the latter is the former divided by the number of individual input data sets, i.e., 10. Because the recursion depths and reduction ratios for the algorithm DS1 and DS2 are identical for each given input string, the results for these two algorithms are listed in the two columns marked with the title of DS. As observed from this table, our IS algorithm achieves all the best results. The reduction ratio of KS is more than double of that for the IS. This well coincides with their time results in Table 2, where the IS runs more than twice faster than the KS.

In this table, the reduction ratio for IS on "alphabet" is 0.2. This is explained as following. The data set "alphabet" consists of repetitions of [a-z]. In the first iteration, it is reduced with a reduction ratio of $1/26 \approx 0.04$; in the second iteration, because all the nonsentinel characters are identical, the reduction ratio can be regarded as 0. Because the mean ratio is the average of the total ratio over the iteration number, i.e., we have $0.04/2 = 0.2$ in this case. Similarly, the reduction ratio for KA on "alphabet" can be explained in the same way.

An interesting observation also from this table is that, for the input file "random," the DS algorithm has only one recursion, which is one level less than the IS algorithm. This well explains why the DS algorithm runs faster than the IS algorithm for input file "random" in Table 2, which is the only case in our experiments that the best time was not achieved by the latter. For the random data, the DS algorithm turns out to converge faster than the IS algorithm, and hence runs faster.

5.3 Discussion

Theorem 3.15 shows that if the S-type and L-type characters are randomly distributed in the string, the reduction ratio will not be greater than 1/3. However, in practice, the characters of a string usually exhibit certain statistical correlations, which will likely render a smaller reduction ratio, e.g., the mean of 0.29 for IS in Table 4. Because all the strings in the experiments are of constant alphabets, from Fig. 3, the maximum space of our IS algorithm is observed at level 1. Given the mean reduction ratio 0.29, the space for SA is sufficient for accommodating S_1, SA_1, and B_1 of IS. In this experiment, the implementation of IS keeps the type array t_i throughout the lifetime of S_i at level i, which could lead to a usage of up to $2n$ bits in the worst case, i.e., 0.25 byte per character. Hence, we see the mean space of 5.37 bytes per character for the IS algorithm in Table 3. Such a space complexity is approaching the space extreme for suffix array construction (i.e., 5 bytes per character in this case).

6 Closing Remarks

Our proposed algorithms have been adopted by the other parties in their projects, e.g., [23], [24]. In particular, Yuta Mori has optimized the coding of the SA-IS algorithm, and conducted an extensive performance evaluation study [25] for the SA-IS algorithm versus the other well-known linear and super-linear time SACAs, i.e., the Difference-Cover [26], Deep-Shallow sorting [6], KA [1], and Larsson-Sadakane [27] algorithms. The optimized implementation of SA-IS was observed to be the most time and space efficient from his experiment results.

Appendix

I: Sample Implementation of Algorithm SA-IS

A sample natural implementation of our SA-IS algorithm can be found in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2010.188. In less than 100 lines of C code for demonstration purpose, which is also the source code used in our experiment.

II: Sample Implementation of Algorithm SA-DS

The source code located at the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2010.188 is to give a sample implementation in C for our SA-DS algorithm with $d = 3$, i.e., the length of a d-critical substring is $d + 2 = 5$. Since both the KS algorithm and ours sort fixed-size substrings, for reader's convenience of comparison, we intended to code the program with a structure similar to that for the KS algorithm [21], wherever applicable. This sample implementation uses an extra working space of at most $2.25n + O(1)$ bytes, in addition to the input string and the output suffix array.

ACKNOWLEDGMENTS

The authors are grateful to the reviewers of DCC'09 and CPM'09 for the previous presentations of the two algorithms proposed in this article, for their constructive and insightful comments that have helped improve the presentation. The authors thank Pang Ko and Simon Puglisi for the helpful discussions on their codes and works. They also thank Yuta Mori for sharing his improved version of the original KA code to complete our experiments. This research work was supported in part by the National Natural Science Foundation of P.R.C. Grant No. 60873056, the Fundamental Research Funds for the Central Universities of China Grant No. 11lgzd04 and the Faculty Research Grant No. FRG/07-08/II-30, Hong Kong Baptist University, Hong Kong.

REFERENCES

[1] P. Ko and S. Aluru, "Space-Efficient Linear Time Construction of Suffix Arrays," *J. Discrete Algorithms,* vol. 3, nos. 2-4, pp. 143-156, 2005.
[2] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear Work Suffix Array Construction," *J. ACM,* no. 6, pp. 918-936, Nov. 2006.
[3] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *Proc. First Ann. ACM-SIAM Symp. Discrete Algorithms (SODA '90),* pp. 319-327, 1990.
[4] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *SIAM J. Computing,* vol. 22, no. 5, pp. 935-948, 1993.
[5] S.J. Puglisi, W.F. Smyth, and A.H. Turpin, "A Taxonomy of Suffix Array Construction Algorithms," *ACM Computing Surveys,* vol. 39, no. 2, pp. 1-31, 2007.
[6] G. Manzini and P. Ferragina, "Engineering a Lightweight Suffix Array Construction Algorithm," *Algorithmica,* vol. 40, no. 1, pp. 33-50, Sept. 2004.
[7] R. Grossi and J.S. Vitter, "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching," *Proc. Symp. Theory of Computing (STOC '00),* pp. 397-406, 2000.
[8] W.K. Hon, K. Sadakane, and W.K. Sung, "Breaking a Time-and-Space Barrier for Constructing Full-Text Indices," *Proc. 44th Ann. IEEE Symp. Foundations of Computer Science (FOCS '03),* pp. 251-260, 2003.
[9] T.W. Lam, K. Sadakane, W.K. Sung, and S.M. Yiu, "A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays," *Proc. Int'l Conf. Computing and Combinatorics,* pp. 401-410, 2002.
[10] S. Kurtz, "Reducing the Space Requirement of Suffix Trees," *Software Practice and Experience,* vol. 29, pp. 1149-1171, 1999.
[11] J. Kärkkäinen and P. Sanders, "Simple Linear Work Suffix Array Construction," *Proc. 30th Int'l Conf. Automata, Languages, and Programming (ICALP '03),* pp. 943-955, 2003.
[12] P. Ko and S. Aluru, "Space Efficient Linear Time Construction of Suffix Arrays," *Proc. Ann. Symp. Combinatorial Pattern Matching (CPM '03),* pp. 200-210. 2003.
[13] D.K. Kim, J.S. Sim, H. Park, and K. Park, "Linear-Time Construction of Suffix Arrays," *Proc. Ann. Symp Combinatorial Pattern Matching (CPM '03),* pp. 186-199. 2003.
[14] M. Farach, "Optimal Suffix Tree Construction with Large Alphabets," *Proc. 38th Ann. IEEE Sym. Foundations of Computer Science (FOCS '97),* p. 137, 1997.
[15] H. Itoh and H. Tanaka, "An Efficient Method for In Memory Construction of Suffix Arrays," *Proc. String Processing and Information Retrieval Symp.*, 1999.
[16] S.J. Puglisi, W.F. Smyth, and A. Turpin, "The Performance of Linear Time Suffix Sorting Algorithms," *Proc. Data Compression Conf.,* pp. 358-367, Mar. 2005.
[17] S. Lee and K. Park, "Efficient Implementations of Suffix Array Construction Algorithms," *Proc. 15th Australasian Workshop Combinatorial Algorithms,* pp. 64-72, 2004.
[18] G. Nong, S. Zhang, and W.H. Chan, "Linear Suffix Array Construction by Almost Pure Induced-Sorting," *Proc. Data Compression Conf. (DCC '09),* Mar. 2009.
[19] G. Nong, S. Zhang, and W.H. Chan, "Linear Time Suffix Array Construction Using d-Critical Substrings," *Proc. 20th Ann. Symp. Combinatorial Pattern Matching (CPM '09),* pp. 54 - 67, June 2009.
[20] T. Bell andetc., "The Canterbury Corpus," http://corpus.canterbury.ac.nz, 2010.
[21] P. Sanders, "A Driver Program for the KS Algorithm," http://www.mpi-inf.mpg.de/sanders/programs/suffix/, 2007.
[22] P. Ko, "Source Codes for the KA Algorithm," http://kopang.public.iastate.edu/homepage.php?page=source, 2007.
[23] Y. Mori, "Libdivsufsort—A Lightweight Suffix Sorting Library," http://homepage3.nifty.com/wpage/software/libdivsufsort.html, 2007.
[24] H. Li, "Bwa-Burrows-Wheeler Alignment Tool," http://maq.sourceforge.net/bwa-man.shtml, 2008.
[25] Y. Mori, "SAIS-An Implementation of the Induced Sorting Algorithm," http://yuta.256.googlepages.com/sais, 2008.
[26] S. Burkhardt and J. Kärkkäinen, "Fast Lightweight Suffix Array Construction and Checking," *Proc. 14th Ann. Conf. Combinatorial Pattern Matching (CPM '03),* pp. 55-69, June 2003.
[27] N.J. Larsson and K. Sadakane, "Faster Suffix Sorting," Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1-20/(1999), Dept. of Computer Science, Lund Univ., May 1999.

Ge Nong received the BE and ME degrees in computer engineering from the NanJing University of Aeronautics and Astronautics, in 1992, and the South China University of Science and Technology, in 1995, respectively. He received the PhD degree in computer science from the Hong Kong University of Science and Technology, in 1999. He joined STMicroelectronics as a researcher with R&D on IC and system technologies for high-speed switches and routers. He is now a professor in the Department of Computer Science of Sun Yat-sen University in Guangzhou, China. His current research interests include algorithms, computer and communication networks, switching theory, and performance evaluation. He is a member of the IEEE.

Sen Zhang received the BS degree in computer science from TianJin University, TianJin, China, the ME degree in computer engineering from the South China University of Science and Technology, Guangzhou, China, and the PhD degree in computer science from New Jersey Institute of Technology, Newark, in 1992, 1995, and 2004, respectively. He joined the Department of Mathematics, Computer Science and Statistics, State University of New York College at Oneonta, in 2004, as an assistant professor. His current research interests include algorithms, data mining, database management, and bioinformatics. He is a member of the IEEE.

Wai Hong Chan received the BSc, MPhil, and PhD degrees in mathematical science from Hong Kong Baptist University, Hong Kong Special Administrative Region, China, in 1994, 1996, and 2003, respectively. He joined the Department of Mathematics, Hong Kong Baptist University in 2006. He is now an assistant professor in the Department of Mathematics and Information Technology, The Hong Kong Institute of Education, Hong Kong. His current research interests include algorithm design, quantum information, graph theory, and combinatorics.