Two Efficient Algorithms for Linear Suffix Array Construction

Ge Nong
Computer Science Department
Sun Yat-Sen University
GuangZhou 510275, PRC.
issng@mail.sysu.edu.cn

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

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

Abstract

We present in this paper two efficient algorithms for linear suffix array construction. These two algorithms archive their linearities using the techniques of divide-and-conquer and recursion. What distinguish the proposed algorithms from the other linear time suffix array construction algorithms are the variable-length LMS-substrings and the fixed-length d-critical substrings sampled for problem reduction, and the very 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 algorithms for linear suffix array construction.

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 indexes pointing to all the suffixes of S sorted according to their ascending(or descending) lexicographical order. The SA 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 research has been conducted on 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 huge datasets is measured often in billions of characters [6, 7, 8, 9, 10]. Time and space efficient linear algorithms are not only crucial for large-scale applications to have predicable worst-case performance, but also important for computer scientists to gain insights into the problem from theoretical point of view. **In this article, our interest is limited to linear suffix array construction only**.

Prior-Arts The three well known linear SACAs up to date are KS [11, 2], KA [12, 1] and KSP [13], all contemporarily reported in 2004. The three algorithms are linear for input strings of both constant and 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 merge step, thus KSP does not gain popularity in practice.

The key idea of the KS algorithm is straightforward: a size-n string S can be reduced by naming each size-3 substring $S[i..i+2]$ for $i \mod 3 \neq 2$ as an integer of size $\lceil \log n \rceil$ bits. Clearly, this method can reduce the problem size of n to a smaller one S_1 of size $2n/3$. The reduction can be done in a time complexity $O(n)$ by simply running 3 passes of radix sort on all the selected substrings. Furthermore, the reduction will be recursively conducted down to a point where calculating the suffix array of the reduced problem becomes trivial. Then the process backtraces to the original problem level by level. At each level, the suffix array of the reduced problem can help sort the unselected elements at that level. Finally, using the suffix array of S_1, the KS algorithm can compute the suffix array of the original S. Hence, the time complexity is given by a recursion formula of $\mathcal{T}(n) = \mathcal{T}(\lceil 2n/3 \rceil) + O(n) = O(n)$. The KS algorithm is claimed to have a time complexity $O(vn)$ and need an extra working space of $O(n/\sqrt{v})$, where $v = O(\sqrt{n})$. Herein, we define **working space** as the extra space needed in addition to the input string and the output suffix array (which are universally needed for any SACA).

The key idea of the KA algorithm [12] lies in classifying all the characters in the string into two classes: L- and S-types, which, to some degree, is a variant of the type-A/B suffix classification method proposed by Itoh and Tanaka [15]. Based on the L- and S-types, a S-substring is defined as any substring $S[i..j]$, that has $S[i]$ and $S[j]$ to be the only S-type characters in the range of i to j and all the other characters in between i and j, if any, to be L-type. Since the definitions of L- and S-types (see 2.2 for the precise definition) are symmetric, it is safe to assume that there are fewer S-substrings; otherwise, L-substrings will be used instead. Therefore, by naming all the S-substrings, the KA algorithm can reduce the problem to a shorter string with at most $\lceil n/2 \rceil$ characters, where each character is the name of size $\lceil \log n \rceil$ bits for each S-substring in the original problem. As a result, the time complexity is $\mathcal{T}(n) = \mathcal{T}(\lceil n/2 \rceil) + O(n) = O(n)$. According to Ko and Aluru [12], their algorithm needs a working space of $3n$ bytes plus $1.25n$ bits for a string not longer than 2^{32}.

Remarks

What Is Common The KS and the KA algorithms share a similar divide-and-conquer framework, which comprises problem reduction, recursion and unreduced problem induction. To be more specific, the framework works as following. 1) First the input string is reduced into a smaller string, so that the original problem is divided into a reduced part and an unreduced part; 2) then the suffix array of the reduced problem is recursively computed; 3) finally based on the result of the previous step, the suffix array of the unreduced problem is induced. In order to reduce the problem at the step 1, the selected substrings, either the triplets in the KS algorithm or the S-substrings in the KA algorithm, need to be sorted and re-named by their order indexes. This step is commonly known as substring naming. At the step 2, if the suffix array of the reduced problem is not immediately obtainable, it will further trigger a recursive call to solve the reduced problem. Due to this similar divide-and-conquer framework, both algorithms have the same linear behavior in terms of time and space complexities.

What Is Different 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 reduces the problem size at each iteration in a constant reduction ratio of 2/3; whereas the KA algorithm selects the S-substrings, which have varying lengths subject to the specific characteristics of the given string. The reduction ratio of the KA algorithm is always not more than 1/2 due to the symmetric definition of L- and S-types. Herein, reduction ratio is defined as the the size of the new child problem against that of its parent. Due to the better reduction ratio (1/2 vs. 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 may have different and unpredictable lengths, which makes the design of problem reduction in the KA algorithm more complicated than that in the KS algorithm where the fixed length blocks are sampled. 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, and the S-distance for a character $S[k]$ in a S-substring $S[i..j]$ is $k-i$. However, maintaining the S-distance lists demands not only extra space but also additional time. Moreover, the S-distance lists complicate the algorithm's design, which is well evidenced by the sample implementations of the KA and the KS algorithms: the KS algorithm can be implemented within only around 100 lines in C; whereas the source codes for the KA algorithm have more than 1000 lines. In this sense, the KS algorithm is far 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. To summarize, the KA algorithm has been observed to be faster than the KS algorithm; but the KS algorithm beats the KA in terms of algorithm design elegancy and implementation simplicity.

What Is New Recently, we proposed two new linear suffix array construction algorithms that sample the variable-length leftmost S-type (LMS) substrings (Definition 2.2) and the fixed-length d-critical substrings (Definition 3.3), and use very simple induced sorting and radix sorting methods to sort the sampled substrings, respectively. Since the LMS and the d-critical substrings are longer blocks than the S-substrings, our algorithm achieves an even better mean reduction ratio, and thus faster, than the KA algorithm. For our algorithm sampling the fixed-length d-critical substrings, sorting the sample substrings can be done using very simple radix sorting algorithm, for their lengths are identical. For our another algorithm sampling variable LMS substrings, however, the same bottleneck problem that has enforced the KA algorithm to adopt the S-distance lists based solution may also demand a similar cumbersome solution from us. But interestingly enough, we need not use any heavy data structures like the S-distance lists, but simply used a pure induced-sorting method to address this bottleneck problem in a way that has never been used before. The reason for us to call this algorithm pure induced-sorting, is that induced-sorting is used NOT ONLY in the step 3 to induce the ordering of the unreduced problem, BUT ALSO used in the step 2 to induce the ordering of the reduced problem. This is dramatically different from both the KS and the KA algorithms which use induced sorting only in the step 3, but not in the step 2.

In the rest of this article, these two new algorithms are presented and analyzed in Section 2 and 3, respectively, followed by an experimental study for performance evaluation in Section 4. Finally, Section 5 concludes the paper.

Before going further, we would ask the reader to remind that two common symbols, P_1 and S_1, are used for presenting the two algorithms, however, their exact definitions are different and given in the contexts of the two algorithms, respectively.

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

2.1 Algorithm Framework The framework of our linear suffix array sorting algorithm that samples and sorts 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.

2.2 Reducing the Problem Let S be a string of n characters, represented by an array indexed by $[0..n-1]$, For a substring of $S[i]S[i+1]...S[j]$ 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]). In other words, all the other characters in S are larger than \$.

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. 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, respectively. From the S- 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 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

```
SA-IS($S, SA$)
     ▷ $S$ is the input string;
     ▷ $SA$ is the output suffix array of $S$;
     $t$: array $[0..n-1]$ of boolean;
     $P_1$, $S_1$: array $[0..n_1-1]$ of integer;
     $B$: array $[0..\|\Sigma(S)\|-1]$ 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 $P_1$;
 3   Induced sort all the LMS-substrings using $P_1$ and $B$;
 4   Name each LMS-substring in $S$ by its bucket index to get a new shortened string $S_1$;
 5   if Each character in $S_1$ is unique
 6      then
 7           Directly compute $SA_1$ from $S_1$;
 8      else
 9           SA-IS($S_1, SA_1$); ▷ where recursive call happens
10   Induce $SA$ from $SA_1$;
11   return
```

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

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. Assigning the S-type a higher priority is based on a property directly from the definitions of L- and S-types [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- and L-type, respectively.

To sort all the LMS-substrings, we don't have to use extra physical space to store them; instead, we simply maintain a pointer array, denoted by 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, the last must be LMS. Moreover, there are at least three characters in each non-sentinel LMS-substring, and any two neighboring LMS-substrings overlap on a common character.

LEMMA 2.2. *(Sentinel) The last character of S_1 must be the unique smallest character in S_1.*

Proof. From the definition of LMS character, we know that $S[n-1]$ 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.

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

LEMMA 2.3. *(Coverage) For any two characters $S_1[i] = S_1[j]$, there must be $P_1[i+1] - P_1[i] = P_1[j+1] - P[j]$.*

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 conducted on $S_1[i+1] = S_1[j+1]$, $S_1[i+2] = S_1[j+2]$,...$S_1[i+k-1] = S_1[j+k-1]$ 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 2.3, 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 smaller 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, Now, let's 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, $SA(S)$ (the notation of SA is used for it when there is no confusion in the context), i.e. the suffix array of S, maintains 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 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 with respect to the types of suffixes inside: the L- and S-type buckets.

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 the head to the 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 the end to the 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 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 the Lemma 3 established in [12], cited as below.

Lemma 2.5. *[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 2.5 can be translated into the below statement.

Lemma 2.6. *Given all the L-type suffixes of S sorted, all the suffixes of S can be sorted by the 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, we know that given all the S-type suffixes having 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 simply put all the sorted LMS suffixes into their S-type buckets in SA, from the ends to the heads. Hence, from the lemma 2.7, we see that the step 2 will sort all the L-type suffixes correctly; and then from the lemma 2.6, we see that the step 3 will sort all the suffixes from the sorted L-type suffixes.

2.4 Induced Sorting LMS-Substrings This part is dedicated to addressing the most challenging problem in our whole algorithm design: 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. But, our solution does not need to use the cumbersome s-distance lists. Instead, we solved this once difficult problem by using the same induced-sorting idea originally used in the algorithm for inducing SA from SA_1. 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, we see that any L-type has at least two characters and the sentinel. 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 the first step, 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, the 2nd 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-prefix. 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 being not sorted correctly. This contradicts to 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 the 3rd step will sort all the non-size-one LMS-prefixes, which is conducted symmetrically to that for the 2nd 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. Notice that in the first step, 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. 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 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 to 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 are 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 are 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 the figure 3, as well as for sorting the S- or L-substrings in the KA algorithm.

2.5 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 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 5 buckets for all the suffixes marked by their first characters, i.e. \$, i, m, p and s, respectively. Each bucket is bordered 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 all the LMS suffixes into their buckets according to their first characters. Please be reminded 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 the end to the 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: 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 bucket for 'i', and the current head of the bucket 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 bucket for 'p', and forward the bucket's head one step. To repeat scanning SA in this way, we can get all the L-type LMS-prefixes sorted in SA as shown in line 28, where a symbol between two buckets means that the left bucket is full.

- Step 3: In this step, we induced sort 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, then put 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. $S[15] = 8$, we see $S[7] = i$ is S-type, then we put 7 to the bucket for 'i' and forward the bucket head one step to the left. Notice that the LMS-prefixes 3 and 7 overwrote the size-one LMS-prefix 2 and 6 already in the bucket, respectively. To repeat scanning SA in this way, all the non-size-one LMS-prefixes and the sentinel (which was put into its bucket in the 1st step, and will not be overwritten by any character in this step, for it is the last character in the string) are sorted in their order shown in line 44.

- 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 46, where the names for the LMS-substrings 2, 6, 10 and 16 are 2, 2, 1, 0, respectively.

```
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
```

2.6 Complexity Analysis for IS

THEOREM 2.2. *(Time/Space Complexities) Given S is of a constant or integer alphabet, the time and space complexities for the algorithm SA-IS in Fig. 3 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.

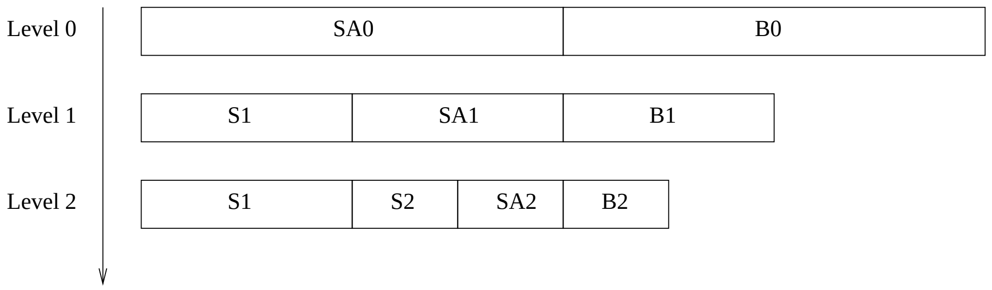

Figure 2: The worst-case space requirement at each recursion level.

To investigate the accurate space requirement, we show in the figure 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.

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 is required by the 2nd 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 2.1. *The worst-case working space requirements for SA-IS in Fig. 3 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 2.3. *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 not greater than 1/3.*

Proof. Let's consider a non-sentinel 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 another 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- 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 that 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 together with the figure 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 section, this theorem well approximates the results on realistic data.

3 Algorithm II: Radix Sorting Fixed-Length d-Critical Substrings

In this section, our another algorithm for linear suffix array construction is presented. We first introduce the concept of d-critical characters, which builds the basis of our algorithm presented in this section.

3.1 Critical Characters

Definition 3.1. *(Critical Character) A character $S[i]$ is said to be d-critical, where $d \geq 2$, iif (1) $S[i]$ is a LMS character; or else (2) $S[i-d]$ is a d-critical character, $S[i+1]$ is not a LMS character and no character in $S[i-d+1..i-1]$ is d-critical.*

Definition 3.2. *(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 characters between them.*

Definition 3.3. *(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 3.1. *In S, (1) all LMS characters are d-critical characters, and (2) the last character must be a d-critical character, and the first character must not be a d-critical character.*

Proposition 3.2. *Given $S[i]$ is a d-critical character, neither $S[i-1]$ nor $S[i+1]$ is a d-critical character.*

Lemma 3.1. *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 3.2, given $S[i]$ is a d-critical character, $S[i+1]$ must not be a d-critical character. In other words, the first d-character on the right hand of $S[i]$ may be any in $S[i+2, d+1]$, but must not be $S[i+1]$.

3.2 Algorithm Framework Our linear suffix array sorting algorithm SA-DS is outlined in Fig. 3. Lines 1-3 first produce the reduced problem, which is then solved recursively by Lines 4-8, and finally from the solution of the reduced problem, Line 9 induces the final solution for the original problem. The time and space bottleneck of this algorithm resides at reducing the problem in Lines 1-3, which is $O(n)$. In the rest of this section, we further describe in more details about the operations in each step.

3.3 Reducing the Problem With the introduced d-critical concept, here comes the key idea to reduce the problem of computing $SA(S)$ into a simpler one 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 3.4. *(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}. To compute P_1 from S, we need to know the LS-type of each character in S. This can be done by scanning S once from right to left in $O(n)$ time, by utilizing these properties: (i) $S[i]$ is type-S if (i.1) $S[i] < S[i+1]$ or (i.2) $S[i] = S[i+1]$ and $suf(S, i+1)$ is type-S; and (ii) $S[i]$ is type-L if (ii.1) $S[i] > S[i+1]$ or (ii.2) $S[i] = S[i+1]$ and $suf(S, i+1)$ is type-L. Provided with the LS-type of each character is known, we can traverse S once from right to left to compute P_1 in $O(n)$ time.

```
SA-DS($S, SA$)
     ▷ $S$ is the input string;
     ▷ $SA$ is the output suffix array of $S$;
     $t$: array $[0..n-1]$ of boolean;
     $P_1$, $S_1$: array $[0..n_1]$ of integer;
     $B$: array $[0..\|\Sigma(S)\|-1]$ of integer;
 1   Find the sample pointer array $P_1$ for all the fixed-size d-critical substrings in $S$;
 2   Bucket sort all the fixed-size d-critical substrings in $P_1$;
 3   Name each d-critical substring in $S$ by its bucket index to get a new shortened string $S_1$;
 4   if $\|S_1\|$ = Number of Buckets
 5      then
 6            Directly compute $SA_1$ from $S_1$;
 7      else
 8            SA-DS($S_1, SA_1$);
 9   Induce $SA$ from $SA_1$;
10   return
```

Figure 3: The algorithm framework for linear suffix array construction by recursive sorting fixed-length d-critical substrings.

Definition 3.5. *(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 3.2. *(Sentinel) The last character of S_1 must be the unique smallest character in S_1.*

Proof. From Proposition 3.1, 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 3.3. *(1/2 Reduction Ratio) $\|S_1\|$ is at most half of $\|S\|$, i.e. $n_1 \leq \lfloor n/2 \rfloor$.*

Proof. From Proposition 3.1, $S[0]$ must not be a d-critical character. We know from Lemma 3.1 the distance between any two neighboring d-critical character is at least 2, 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 3.1. *(Coverage) For any two characters $S_1[i] = S_1[j]$, there must be $P_1[i+1] - P_1[i] = P_1[j+1] - P[j]$.*

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 in $S[i'..i'+d+1]$ is a LMS character. In this case, given $S_1[i] = S_1[j]$, we must have $P_1[i+1] = P_1[j+1]$.
- None in $S[i'..i'+d+1]$ is a LMS character. In this case, both $i'+d$ and $j'+d$ must be in P_1.

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

Theorem 3.2. *(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: $S_1[i] \neq S_1[j]$. 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 $S_1[i+1] = S_1[j+1]$, $S_1[i+2] = S_1[j+2]$,...$S_1[i+k-1] = S_1[j+k-1]$ 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 3.1, 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 d-critical suffixes in S, we can sort S_1 instead. Because S_1 is at least 1/2 smaller than S, the computation on S_1 can be done within about one half the complexity for S. In the following subsections, we show how to bucket sort and name the items of P_1, i.e. the two crucial subtasks of computing S_1.

3.4 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 size-n_1 used as the alternating buffers for bucket sorting P_1, and another size-n for the bucket pointers, where $2n_1 \leq n$. The array of bucket pointers needs a size of 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 3.6. *(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 below result.

Lemma 3.4. *Given $S_\gamma[i..i+k] < S_\gamma[j..j+k]$ and $S[i..i+k] = S[j..j+k]$, we must have $t(S, i+x) \leq t(S, j+x)$ 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 all cases, $t(S, i+k-1) \leq t(S, j+k-1)$. The proof is completed by applying the analogous arguments to $t(S, i+k-2)$, $t(S, i+k-3)$..., 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 3.3. *(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 3.6), which immediately 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 3.4, we have $t(S, i'+x) \leq t(S, j'+x)$ 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 3.3 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 compare the characters' types only for the last characters of the 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 these characters themselves. Notice that the sorting of all the γ-weighted substrings is not required to be stable, we can use a fast method to sort the last characters of these substrings. At step (1), there are only two buckets, one for the type-L and another for the type-S characters. This naturally suggests that the task of step (1) can be done by traversing all the characters only once to examine their LS-types and put them into their buckets accordingly, as embodied in the `bucketSortLS` procedure of the codes in the appendix.

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 1st or 2nd 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 elements in P'_1 (as well as P_1). This job can be done by a simple algorithm described as following: (i) allocate a size-n array tmp, where each item is an integer in $[0, n-1]$; (ii) initialize all items of tmp to be -1; (iii) 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; (iv) pack all non-negative 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 by reusing tmp for P'_1 and S_1, described as following. Let's define a logical array $\widetilde{tmp_e} = \{tmp[i] | i\%2 = 0\}$ for the first n_1 even items of tmp, where $\widetilde{tmp_e}$ 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 $\widetilde{tmp_e}$ and set $tmp[j] = -1$ for any $tmp[j] \notin \widetilde{tmp_e}$, i.e., distribute P'_1 into the first even items of tmp. Next, we scan $\widetilde{tmp_e}$ from left to right to compute the names for all the the items of $\widetilde{tmp_e}$. For each $\widetilde{tmp_e}[i]$, we record its name as following: (1) if $\widetilde{tmp_e}[i]$ is even, set $tmp[\widetilde{tmp_e}[i]-1]$ with the name; or else set $tmp[\widetilde{tmp_e}[i]]$ with the name. Now, all the items of S_1 are stored in the non-negative odd items of tmp in their correct relative positional orders. Last, we traverse tmp once to compact all the non-negative 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 all of them.

3.5 Inducing SA from $SA(S_1)$ For denotation simplicity, let $SA_0 = SA(S)$ and $SA_1 = SA(S_1)$. Furthermore, let us define two arrays as following: $SA_{lms} = \{SA_0[i] | \mathrm{S[SA_0[i]]\ is\ a\ LMS\ character}\}$ and $SA_l = \{SA_0[i] | \mathrm{S[SA_0[i]]\ is\ a\ type-L\ character}\}$. From SA_1, we can derive SA_{lms} and then induce SA_l from SA_{lms} and SA_0 from SA_l, as described below. The algorithm for inducing SA_0 from SA_1 consists of four sequential stages in $O(n)$ time/space, where the last stage uses the inducing method of the KA algorithm in [12](p. 203).

1. Initialization: (1) Set all the items of SA_0 to be negative. (2) Scan S at most twice to find the buckets in SA_0 for all the suffixes in S according to their first characters.

2. Deriving SA_{lms} from SA_1: (1) Initialize all the buckets in SA_0 as empty by setting the start of each bucket as its end. (2) Scan SA_1 once from right to left, if $S[SA_1[i]]$ is a LMS character then put $SA_1[i]$ to the current start of its bucket in SA_0 and move the bucket start one item to the left.

3. Inducing SA_l from SA_{lms}: (1) For each bucket in SA_0, set the end as its start. (2) Scan SA_0 from left to right, for each non-negative item $SA_0[i]$, if $S[SA_0[i]-1]$ is type-L then put $SA_0[i]-1$ to the current end of its bucket and move the bucket end one item to the right.

4. Inducing SA from $SA_l(S)$: (1) For each bucket in SA_0, set the start as its end. (2) Scan SA_0 from right to left, for each item $SA_0[i]$, if $S[SA_0[i]-1]$ is type-S then put $SA_0[i]-1$ to the current start of its bucket and move the bucket start one item to the left.

In the above algorithm, in addition to SA_0, we need another array B for maintaining the start/end of each bucket on-the-fly in each stage, where B has $\|\Sigma(S)\|$ items and each item is of $\lceil \log n \rceil$ bits.

We now consider the correctness of the inducing algorithm for the stages 2-4. Given SA_1, the correctness of stage 2 is obvious, for we just simply copy all LMS items of SA_1 into the ends of their corresponding buckets (notice that in a bucket, a type-L suffix is less than a type-S suffix), keeping their relative orders unchanged. The correctness of stage 3 and 4 can be trivially proven as we did for Lemma 2.6 and 2.7, respectively.

3.6 Example To help the reader grasp the core idea of the proposed algorithm, 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 $S = mmiissiippii\$$, where \$ is the sentinel.

```
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
SAlms: 7 -1  4 -1  2 -1 -1 -1
  SAl: 7  6  4 -1  2  5  3  1
  SA0: 7  6  4  2  0  5  3  1

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

In this example, our algorithm turns out to use only two levels of recursions i.e. the recursion depth is 2. 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 S_1—the reduced string. Further, we recursively compute $SA(S_1)$ and then induce $SA(S)$ from it.

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

General Strategy: Reusing the Buffer for $SA(S)$ From our algorithm framework in Fig.3, we see that the algorithm consists of three steps in sequence: (1) sorting P_1; (2) naming 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 re-use the buffer for $SA(S)$ for the steps (1) and (2) too. For more details, the reader is referred to the sample codes in the appendix.

Strategy 1: Storing the LS-Type Array Each element of the LS-type array for S is one-bit and a total of at most $n(1 + 1/2 + 1/4 + ... + \log^{-1} n) < 2n$ bits are required by the LS-arrays for all recursions. Hence, we can use the two most-significant-bits (MSBs) of $SA(S)[i]$ for storing the LS-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 is always available for us in this case. This is because that to compute $SA(S)$, our algorithm running on a 32-bit machine 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.

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. 3 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 S 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 linear proportional to the total number of characters for these substrings. Since each d-critical substring is of fixed-size $d + 2$ characters and the number of substrings decreases at least half per iteration, the total number of characters sorted at all 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 65536 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 is of a constant alphabet, occur when further reducing S_1 at the 2nd 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 is of an integer alphabet, occur when reducing S at the 1st iteration, which requires an extra space of n integers, where each integer is 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.

Strategy 3: Inducing the Final Result In the inducing algorithm we described before, a buffer B is needed for dynamically recording the current start/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 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 and reset the MSB of $SA_1[i]$ as 0.

3.8 Complexity Analysis for DS

Theorem 3.4. *(Time/Space Complexities) Given S is of a constant or integer alphabet, the time and the space complexities for the algorithm DS in Fig. 3 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 obvious $O(n\lceil \log n\rceil)$ 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 3.1. *(Working Space) The 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 algorithm DS with the general strategy and the strategies 2-3 in Section 3.7. Naturally, we can allocate a LS-type array at each iteration, which requires in total a space of $2n$ bits for the type arrays at all 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 re-produce the type array for inducing the final result for the current iteration.

Despite $\Sigma(S)$ is constant or integer, after the first iteration, the algorithm DS will work on the shortened strings of integer alphabets. In other words, for all the iterations except the 1st iteration, the 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 information 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 DS algorithm to compute $SA(S)$ recursively. After returning from the 2nd 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.

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

[1] We can also use the start of its bucket instead.

We have coded in C a sample implementation for approaching the results stated in Corollary 3.1, i.e. the DS2 algorithm in the experiment section. Interested readers are welcome to contact us for the details/codes.

4 Experiments

The algorithms investigated in our experiments are KS, KA and our algorithms IS, DS1 and DS2, where DS1 and DS2 are two variants of the DS algorithm trading off differently between space and time, with $d = 3$ and enhanced by the practical strategies proposed in Section 3.7. The algorithms DS1 and DS2 use different settings of strategies: DS1 uses the general strategy only, whereas DS2 uses the strategies 2-3 in addition to the general strategy. Specifically, for $d = 3$, each substring sorted by the DS1 and DS2 algorithms is has a fixed length of 5 characters, we sort the substrings at the 1st iteration in 3 passes using a bucket of 65536 integers (instead of sorting in 5 passes with a bucket of 256 integers). The performance measurements to be investigated are the time/space complexities, recursion depth and mean reduction ratio.

Environment All the datasets used in our experiment were downloaded from the ad hoc benchmark repositories for suffix array construction algorithms, including the Canterbury [18] and the Manzini-Ferragina [6] corpora. These datasets 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 is 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 [19]. For the KA algorithm, we use an improved version of the original KA code (at Ko’s website [20]) from Yuta Mori. The source code of our algorithm IS is given in the appendix. Our algorithms DS1 and DS2 were embodied in less than 150 and 250 effective lines of code, respectively, both are available upon request.

Table 1: Data Used in the Experiments

Data	Characters	$\|\Sigma\|$	Description
bible.txt	4 047 392	63	King James Bible
chr22.dna	34 553 758	4	Human chromosome 22
E.coli	4 638 690	4	Escherichia coli genome
etext99	105 277 340	146	Texts from Gutenberg project
howto	39 422 105	197	Linux Howto files
pic	513 216	159	Black and white fax picture
sprot34.dat	109 617 186	66	Swissprot V34 protein database
world192.txt	2 473 400	94	CIA world fact book
alphabet	100 000	26	Repetitions of the alphabet [a-z]
random	100 000	64	Randomly selected from 64 characters

Time and Space 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 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 in all input data.

Table 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 [2]. To be fair, we subtract $7n$ bytes from the space results we measured for the KS algorithm in the experiments, for we are sure $7n$ space can be trivially saved using

[2]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 of this implementation might aim at achieving the best time complexity by pushing the space complexity to its extreme.

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 3 times (300%) faster than the KS, and 43% faster than the KA. The mean space of $24.3n$ 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 all 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% more space than ours.

In the space table, we see that DS1 and DS2 use more space than IS for the small files "pic", "alphabet" and "random". This is due to the bucket of 65536 integers used at the 1st iteration, i.e., 262144 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 3 files are around $5.2n$ bytes too, which is well coincided with the analysis before.

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 the reduction ratio for each algorithm and the mean for both, where the former is the sum of all corresponding results and the later is the former divided by the number of individual input data, i.e. 10. Because the recursion depths and the 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 dataset "alphabet" consists of repetitions of [a-z]. In the 1st iteration, it is reduced with a reduction ratio of $1/26 \approx 0.04$; in the 2nd iteration, because all the non-sentinel 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 from this table and the time 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 in the time table, which is the only case in our experiments that the best time was not archived by the latter. For the random data, the DS algorithm turns out to converge faster than the IS algorithm, and hence runs faster.

Discussion Theorem 2.3 shows that if the S- 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 renders a smaller reduction ratio, e.g. 0.29 in the table 4. Because all the strings in the experiments are of constant alphabets, from the figure 2, 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. 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, we see the mean space of 5.37 bytes per character for the IS algorithm in the table 2. Such a space complexity is approaching the space extreme for suffix array construction (i.e. 5 bytes per character in this case), which leaves the margin for further improvement negligible.

5 Conclusion

We have presented two efficient linear time algorithms for suffix array construction. Our algorithms have the following noticeable merits. (1) All the crucial sorting tasks are achieved using very simple induced sorting or radix sorting. (2) The simple induced sorting and the radix sorting methods coupled with the recursive framework render the elegant algorithm designs. (3) Compared with both the KS and the KA algorithms, our new algorithms achieve not only much better time and space efficiencies, but also enjoy the elegancy of 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

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

design and implementation, which is well evidenced by the experimental results and the sample code provided in the appendix.

Acknowledgment

We are grateful to the anonymous reviewers for the previous version of this article, for their constructive and insightful comments that have helped improve the presentation of this work. Meanwhile, we would also like to thank Pang Ko and Simon Puglisi for the helpful discussions on their codes and works. We also thank Yuta Mori for sharing with us his revised version of the original KA code to complete our experiments.

APPENDIX

A Sample Implementation of Algorithm SA-IS

A sample natural implementation of our IS algorithm is embodied below in less than 100 lines C code for demonstration purpose, which is also the source code used in our experiment.

```
unsigned char mask[]={0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01};
#define tget(i) ( (t[(i)/8]&mask[(i)%8]) ? 1 : 0 )
#define tset(i, b) t[(i)/8]=(b) ? (mask[(i)%8]|t[(i)/8]) : ((~mask[(i)%8])&t[(i)/8])
#define chr(i) (cs==sizeof(int)?((int*)s)[i]:((unsigned char *)s)[i])
#define isLMS(i) (i>0 && tget(i) && !tget(i-1))

// find the start or end of each bucket
void getBuckets(unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
  int i, sum=0;
  for(i=0; i<=K; i++) bkt[i]=0; // clear all buckets
  for(i=0; i<n; i++) bkt[chr(i)]++; // compute the size of each bucket
  for(i=0; i<=K; i++) { sum+=bkt[i]; bkt[i]=end ? sum : sum-bkt[i]; }
}

// compute SAl
void induceSAl(unsigned char *t, int *SA, unsigned char *s, int *bkt,
               int n, int K, int cs, bool end) {
  int i, j;
  getBuckets(s, bkt, n, K, cs, end); // find starts of buckets
  for(i=0; i<n; i++) {
      j=SA[i]-1;
      if(j>=0 && !tget(j)) SA[bkt[chr(j)]++]=j;
  }
}

// compute SAs
void induceSAs(unsigned char *t, int *SA, unsigned char *s, int *bkt,
               int n, int K, int cs, bool end) {
  int i, j;
  getBuckets(s, bkt, n, K, cs, end); // find ends of buckets
  for(i=n-1; i>=0; i--) {
      j=SA[i]-1;
      if(j>=0 && tget(j)) SA[--bkt[chr(j)]]=j;
  }
}

// find the suffix array SA of s[0..n-1] in {1..K}^n
// require s[n-1]=0 (the sentinel!), n>=2
// use a working space (excluding s and SA) of at most 2.25n+O(1) for a constant alphabet
void SA_IS(unsigned char *s, int *SA, int n, int K, int cs) {
  int i, j;
  unsigned char *t=(unsigned char *)malloc(n/8+1); // LS-type array in bits

  // Classify the type of each character
  tset(n-2, 0); tset(n-1, 1); //  the sentinel must be in s1, important!!!
  for(i=n-3; i>=0; i--)
    tset(i, (chr(i)<chr(i+1) || (chr(i)==chr(i+1) && tget(i+1)==1))?1:0);

  // stage 1: reduce the problem by at least 1/2
  // sort all the S-substrings
  int *bkt = (int *)malloc(sizeof(int)*(K+1)); // bucket array
  getBuckets(s, bkt, n, K, cs, true); // find ends of buckets
  for(i=0; i<n; i++) SA[i]=-1;
  for(i=1; i<n; i++)
    if(isLMS(i)) SA[--bkt[chr(i)]]=i;

  induceSAl(t, SA, s, bkt, n, K, cs, false);
  induceSAs(t, SA, s, bkt, n, K, cs, true);
  free(bkt);

  // compact all the sorted substrings into the first n1 items of SA
  // 2*n1 must be not larger than n (proveable)
  int n1=0;
  for(i=0; i<n; i++)
    if(isLMS(SA[i])) SA[n1++]=SA[i];

  // find the lexicographic names of all substrings
  for(i=n1; i<n; i++) SA[i]=-1; // init the name array buffer
  int name=0, prev=-1;
  for(i=0; i<n1; i++) {
    int pos=SA[i]; bool diff=false;
    for(int d=0; d<n; d++)
      if(prev==-1 || chr(pos+d)!=chr(prev+d) || tget(pos+d)!=tget(prev+d))
      { diff=true; break; }
      else if(d>0 && (isLMS(pos+d) || isLMS(prev+d))) break;

    if(diff) { name++; prev=pos; }
    pos=(pos%2==0)?pos/2:(pos-1)/2;
    SA[n1+pos]=name-1;
  }
  for(i=n-1, j=n-1; i>=n1; i--)
      if(SA[i]>=0) SA[j--]=SA[i];

  // stage 2: solve the reduced problem
  // recurse if names are not yet unique
  int *SA1=SA, *s1=SA+n-n1;
  if(name<n1)
    SA_IS((unsigned char*)s1, SA1, n1, name-1, sizeof(int));
  else // generate the suffix array of s1 directly
    for(i=0; i<n1; i++) SA1[s1[i]] = i;

  // stage 3: induce the result for the original problem
  bkt = (int *)malloc(sizeof(int)*(K+1)); // bucket array
  // put all left-most S characters into their buckets
  getBuckets(s, bkt, n, K, cs, true); // find ends of buckets
  for(i=1, j=0; i<n; i++)
    if(isLMS(i)) s1[j++]=i; // get p1
  for(i=0; i<n1; i++) SA1[i]=s1[SA1[i]]; // get index in s
  for(i=n1; i<n; i++) SA[i]=-1; // init SA[n1..n-1]
  for(i=n1-1; i>=0; i--) {
      j=SA[i]; SA[i]=-1;
      SA[--bkt[chr(j)]]=j;
  }
  induceSAl(t, SA, s, bkt, n, K, cs, false);
  induceSAs(t, SA, s, bkt, n, K, cs, true);
  free(bkt); free(t);
}
```

B Sample Implementation of Algorithm SA-DS

The below source code is to give a sample implementation in C for our 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 [19] 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.

```
unsigned char mask[]={0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01};
// get type in bit
#define tget(i) ( (t[(i)/8]&mask[(i)%8]) ? 1 : 0 )
// set type in bit
#define tset(i, b) t[(i)/8]=(b) ? (mask[(i)%8]|t[(i)/8]) : ((~mask[(i)%8])&t[(i)/8])
```

```
// read 1-byte or 4-byte character
#define chr(i) (cs==sizeof(int)?((int*)s)[i]:((unsigned char *)s)[i])
// omega-weight
#define omegaWeight(x) ((int)(chr(x)*2+tget(x)))

// get into p1 the pointers for all the d-critical substrings in s[0..n-1]
int dCriticalChars(unsigned char *s, unsigned char *t, int n, int *p1, int d) {
  int i=-1, j=0;
  while(i<n-1) {
      int h, isLMS=0;
      for(h=2; h<=d+1; h++) // the next d-critical character must be in s[i+2..i+d+1]
          if(tget(i+h-1)==0 && tget(i+h)==1) { isLMS=1; break; }
      if(j==0 && !isLMS) { i+=d; continue; }
      i=(isLMS)?i+h:i+d;    // move to the next d-critical character
      if(p1!=0) p1[j]=i;    // record pointer
      j++;
  }
  return j;
}
// sort src[0..n1-1] to dst[0..n1-1] according to the LS-types of characters in s,
// cs gives the character size, which is 1 for char and 4 for integer.
static void bucketSortLS(int *src, int *dst, unsigned char *s, unsigned char *t,
                         int n, int cs, int n1, int h) {
  int i, j, c[]={0, n1-1};
  for (i=0; i<n1; i++) {
      j=src[i]+h;
      if(j>n-1) j=n-1;
      if(tget(j)) dst[c[1]--]=src[i]; // type-S
      else dst[c[0]++]=src[i];        // type-L
  }
}
// sort src[0..n1-1] to dst[0..n1-1] by d-critical substrings in s with characters in [0, K]
static void bucketSort(int *src, int *dst, unsigned char *s, unsigned char *t,
                       int n, int cs, int n1, int K, int *c, int d) {
  int i, j, sum=0;
  for (i=0; i<(K+1); i++) c[i] = 0; // init counters
  for (i=0; i<n1; i++) {
      if((j=src[i]+d)>n-1) j=n-1;   // s[n-1] is the unique smallest sentinel
      c[chr(j)]++;                  // increase counter
  }
  for (i=0; i<(K+1); i++) {
     int len=c[i]; c[i]=sum; sum+=len;  // running length
  }
  for (i=0; i<n1; i++) {
      if((j=src[i]+d)>n-1) j=n-1;
      dst[c[chr(j)]++]=src[i];      // bucket sort
  }
}
// compute the start/end of each bucket
void getBuckets(unsigned char *s, int *bkt, int n, int K, int cs, bool end) {
  int i, sum=0;
  for(i=0; i<=K; i++) bkt[i]=0;     // clear all buckets
  for(i=0; i<n; i++) bkt[chr(i)]++; // compute the size of each bucket
  for(i=0; i<=K; i++) { sum+=bkt[i]; bkt[i]=end ? sum : sum-bkt[i]; } // compute start or end
}

// compute the suffix array SA of s[0..n-1]={1..K}^n+0, require s[n-1]=0 (sentinel) and n>=2.
void SA_DS(unsigned char *s, int *SA, int n, int K, int cs) {
  int i, j;
  unsigned char *t=(char *)malloc(n/8+1); // LS-type array in bits

  // Stage 1: reduce the problem by at least 1/2
  // Classify the type of each character
  tset(n-2, 0); tset(n-1, 1); //  the sentinel must be in s1, important!!!
```

```
for(i=n-3; i>=0; i--) tset(i, (chr(i)<chr(i+1) || (chr(i)==chr(i+1) && tget(i+1)==1))?1:0);

// 2n1 must be <= n
int *SA1=SA, n1=dCriticalChars(s, t, n, SA1, 3), *s1=SA+n-n1;
// bucket array for bucket sorting and final solution inducing
int *bkt = (int *)malloc(sizeof(int)*(K+1));
// bucket sort the gamma-weighted fixed-size 3-critical substrings
bucketSortLS(SA1, s1 , s, t, n, cs, n1, 4);
bucketSort(s1 , SA1, s, t, n, cs, n1, K, bkt, 4);
bucketSort(SA1, s1 , s, t, n, cs, n1, K, bkt, 3);
bucketSort(s1 , SA1, s, t, n, cs, n1, K, bkt, 2);
bucketSort(SA1, s1 , s, t, n, cs, n1, K, bkt, 1);
bucketSort(s1, SA1 , s, t, n, cs, n1, K, bkt, 0);
free(bkt);
// distribute s1 into the first n1 even elements in SA
for(i=n1-1; i>=0; i--) { j=2*i; SA[j]=SA1[i]; SA[j+1]=-1; }
for(i=2*(n1-1)+3; i<n; i+=2) SA[i]=-1;
// name the sorted substrings
int name = 0, c[] = {-1, -1, -1, -1, -1};
for(i=0; i<n1; i++) {
    int h, pos=SA[2*i], diff=0;
    for(h=0; h<4; h++)
        if(chr(pos+h)!=c[h]) {diff=true; break;}
    if(omegaWeight(pos+4)!=c[4]) diff=true;
    if(diff) {
        name++;
        for(h=0; h<4; h++)
            c[h]=(pos+h<n)?chr(pos+h):-1;
        c[h]=(pos+h<n)?omegaWeight(pos+h):-1;
    }
    if(pos%2==0) pos--;  // even item
    SA[pos]=name-1;
}
// pack s1
for(i=n/2*2-1, j=n-1; i>=0 && j>=0; i-=2)
    if(SA[i]!=-1) SA[j--]=SA[i];

// Stage 2: solve the reduced problem
if(name<n1) { // recurse if each names is not yet unique
  SA_DS((unsigned char*)s1, SA1, n1, name-1, sizeof(int));
} else  // generate the suffix array of s1 directly
  for(i=0; i<n1; i++) SA1[s1[i]] = i;

// Stage 3: induce the final result
dCriticalChars(s, t, n, s1, 3); // get p1 into s1
bkt = (int *)malloc(sizeof(int)*(K+1));
// put all left-most S characters into their buckets
getBuckets(s, bkt, n, K, cs, true); // find ends of buckets
for(i=0; i<n1; i++) SA1[i]=s1[SA1[i]]; // get index in s1 which stores p1 now
for(i=n1; i<n; i++) SA[i]=-1; // init SA[n1..n-1]
for(i=n1-1; i>=0; i--) {
    j=SA[i]; SA[i]=-1;
    if(j>0 && tget(j) && !tget(j-1)) SA[--bkt[chr(j)]]=j;
}
// compute SAl
getBuckets(s, bkt, n, K, cs, false); // find starts of buckets
for(i=0; i<n; i++) {
    j=SA[i]-1;
    if(j>=0 && !tget(j)) SA[bkt[chr(j)]++]=j;
}
// compute SAs
getBuckets(s, bkt, n, K, cs, true); // find ends of buckets
for(i=n-1; i>=0; i--) {
    j=SA[i]-1;
    if(j>=0 && tget(j)) SA[--bkt[chr(j)]]=j;
  }

  free(bkt); free(t);
}
```

References

[1] P. Ko and S. Aluru, "Space-efficient linear time construction of suffix arrays," *Journal of Discrete Algorithms*, vol. 3, no. 2-4, pp. 143–156, 2005.

[2] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear work suffix array construction," *Journal of the ACM*, no. 6, pp. 918–936, Nov. 2006.

[3] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," in *Proceedings of the first ACM-SIAM Symposium on Discrete Algorithms*, 1990, pp. 319–327.

[4] ——, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on 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 Comput. Surv.*, 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," in *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC'00)*, 2000, pp. 397–406.

[8] W. K. Hon, K. Sadakane, and W. K. Sung, "Breaking a time-and-space barrier for constructing full-text indices," in *Proceedings of FOCS'03*, 2003, pp. 251–260.

[9] T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu, "A space and time efficent algorithm for constructing compressed suffix arrays," in *Proceedings of International Conference on Computing and Combinatorics, pages*, 2002, pp. 401–410.

[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," in *30th International Colloquium on Automata, Languages and Programming (ICALP '03)*, 2003, pp. 943–955.

[12] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," in *Proceedings 14th Annual Symp. Combinatorial Pattern Matching, LNCS 2676, Springer-Verlag*, 2003, pp. 200–210.

[13] D. K. Kim, J. S. Sim, H. Park, and K. Park, "Linear-time construction of suffix arrays," in *Proceedings 14th Annual Symp. Combinatorial Pattern Matching, LNCS 2676, Springer-Verlag*, 2003, pp. 186–199.

[14] M. Farach, "Optimal suffix tree construction with large alphabets," in *FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, 1997, p. 137.

[15] H. Itoh and H. Tanaka, "An efficient method for in memory construction of suffix arrays," in *Proceedings of String Processing and Information Retrieval Symposium*, 1999.

[16] S. J. Puglisi, W. F. Smyth, and A. Turpin, "The performance of linear time suffix sorting algorithms," in *Proceedings DCC '05 Data Compression Conference*, Snowbird, UT, USA, Mar. 2005, pp. 358–367.

[17] S. Lee and K. Park, "Efficient implementations of suffix array construction algorithms," in *Proceedings of the 15th Australasian Workshop on Combinatorial Algorithms*, 2004, pp. 64–72.

[18] T. Bell and etc., "The canterbury corpus," http://corpus.canterbury.ac.nz.

[19] P. Sanders, "A driver program for the KS algorithm," *http://www.mpi-inf.mpg.de/ sanders/programs/suffix/*, 2007.

[20] P. Ko, "Source codes for the KA algorithm," *http://kopang.public.iastate.edu/homepage.php?page=source*, 2007.