An Optimal Suffix Array Construction Algorithm

Ge Nong
Department of Computer Science
Sun Yat-sen University, Guangzhou 510275, PRC
E-mail: issng@mail.sysu.edu.cn

Abstract—This article presents a linear-time and $O(1)$ working space algorithm for constructing the suffix array of an input string over a constant alphabet, where the working space is defined as the total space excluding that for storing the input string and the output suffix array. This new algorithm is optimal for suffix array construction in terms of its linear time and constant working space, hence called OSACA (optimal suffix array construction algorithm). In our experiment, our sample implementation of OSACA in C requires a constant working space of only 1029 bytes for any string of size $< 2^{32}$ and of an alphabet size ≤ 256. Such a small and constant working space has approached the limit for a linear-time SACA. The proposed OSACA not only runs the fastest and uses the least space among all the existing linear-time SACAs, but also achieves a deterministic constant working space in both theory and practice.

Index Terms—Suffix array construction, optimal algorithm, linear time, $O(1)$ working space.

1 Introduction

Given a string $S[0, n-1]$ of n characters from an ordered alphabet $[0, k-1]$, the suffix array of S, proposed by Manber and Myers in SODA'90, is an array $[0, n-1]$ of integers storing the suffix indices in their non-decreasing order [1], [2]. For presentation simplicity, we assume that there is always $S[n-1] = 0$ which is the *unique smallest* character in S and called the *sentinel*. Equipped with the sentinel, any two suffixes in S must be different, and their lexicographical order is determined by comparing their characters one by one, from left to right, until we see a difference. Let $suf(S, i)$ denote the suffix $S[i, n-1]$ in S, given that all the suffixes of S have been sorted into an array $SA[0, n-1]$, there must be $suf(S, SA[i]) < suf(S, SA[j])$ for $i < j$.

A plethora of suffix array construction algorithms (SACAs) have been proposed to solve the problem in different time and space complexities, e.g. [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], see [12] for a thorough survey up to 2007. Specifically, it has been known that the suffix array can be computed in linear time [13], [14], [15], [16], [17], [18]. So far, the best time and space performances for linear-time SACAs were evaluated to be achieved by our two previously published linear-time algorithms called SA-IS and SA-DS [18]. Both algorithms use a common conquer-and-divide method to recursively compute the suffix array in linear time. In general, SA-IS runs faster but SA-DS can use less space in the worst case.

Of particular interest to us in this article is to further elaborate the induced sorting technique in SA-IS for running faster and using less space. The key for SA-IS to achieve linear time is a combined use of the linear-time problem reduction and solution induction methods. The time complexity of SA-IS is given by $T(n) = T(\lfloor n/2 \rfloor) + O(n) = O(n)$, where $T(\lfloor n/2 \rfloor)$ counts for reducing S into a new shortened string S_1 which size is not greater than half of S (see Lemma 3.5 in [18]), and $O(n)$ is due to inducing the suffix array of S from that of S_1. The core of the whole SA-IS algorithm is the induced sorting technique for sorting the suffixes as well as the sampled substrings, which is developed on top of the following classification of L-type and S-type suffixes [4], [19], [14], [16], [17], [18].

The suffix composed of only the sentinel itself, i.e. $suf(S, n-1)$, is S-type. For $i \in [0, n-2]$, a suffix $suf(S, i)$ is defined as L-type or S-type if $suf(S, i) > suf(S, i+1)$ or $suf(S, i) < suf(S, i+1)$, respectively. Equivalently, $suf(S, i)$ is L-type if and only if (1) $S[i] > S[i+1]$ or (2) $S[i] = S[i+1]$ and $suf(S, i+1)$ is L-type; and $suf(S, i)$ is S-type if and only if (1) $S[i] < S[i+1]$ or (2) $S[i] = S[i+1]$ and $suf(S, i+1)$ is S-type. From the suffix type definitions, a L-type or S-type suffix is larger or smaller than its succeeding, respectively. Further, a S-type suffix $suf(S, i)$ is called a LMS-suffix (leftmost S-type) if $suf(S, i-1)$ is L-type. Given the type of a suffix, we further define the type of a character: $S[i]$ is L-type or S-type if $suf(S, i)$ is L-type or S-type, respectively. Furthermore, $S[i]$ is called a LMS-character if $suf(S, i)$ is a LMS-suffix. A substring $S[i, j]$ is called a LMS-substring if: (1) $i = j = n-1$; or (2) $i < j$, both $S[i]$ and $S[j]$ are LMS-characters, and there is no other LMS-character in between them. Let $lms(S, i)$ denote the LMS-substring staring at $S[i]$.

The following diagram is an example for illustrating the concepts of suffix/character type and LMS-substring. Given a string $S = ''\mathrm{ococonut0}''$, by scanning the string from right to left, we find the type of each suffix as well as character and store it into an n-bit type array $t = [0, 1, 0, 1, 0, 1, 0, 0, 1]$, where $t[i]$ gives the type of $suf(S, i)$: 1 for S-type and 0 for L-type, respectively. Also, all the LMS-substrings, in their positional order from left to right in S, are found to be $\{''\text{coc}'', ''\text{con}'', ''\text{nut0}'', ''0''\}$ (notice that the sentinel itself must be a LMS-string), where each pair of neighboring LMS-substrings overlap on a common LMS-character.

S:	o c o c o n u t 0
character type:	L S L S L S L L S
t:	0 1 0 1 0 1 0 0 1
LMS–substrings:	coc, con, nut0, 0

The induced sorting methods in SA-IS is a kind of bucket sorting developed in the context of suffix array construction. Given a set of elements sorted in an array, each subset of equivalent elements must locate consecutively in a sub-array called bucket. Hence, if we sort all the characters of S into SA, we will see a set of *bucket(s) in* SA, where each bucket comprises a set of equivalent characters(s). Notice that if we sort all the suffixes of S into SA, then all the suffixes with a common first character must fall into the bucket for their first characters. Let $bucket(SA, S, i)$ denote the bucket in SA for character $S[i]$ as well as suffix $suf(S, i)$. Furthermore, the first and the last items of a bucket are called the *head* and the *end* of the bucket, respectively.

An important property utilized to develop the algorithms in [14], [18] is that in each bucket in SA, a L-type suffix of S must be less than and hence locate before a S-type suffix. This property was exploited by SA-IS to induced sort both the sampled sub-substrings and the suffixes at each recursion level. The induced sorting algorithms in SA-IS are bucket sorting in principle, using a bucket counter array bkt for keeping track the status of each bucket on-the-fly when sorting. In this article, we will show how to design a new algorithm called OSACA (optimal SACA), by removing from SA-IS these two arrays: t from the top recursion level and bkt from the deeper levels. This algorithm is termed as optimal in the sense of that it achieves a linear time complexity of $O(n)$ and a constant working space of $O(1)$, for any size-n string S over a constant alphabet of size $O(1)$, where the **working space** is the total space excluding that for the input S and the output SA.

In the rest of this article, we present the OSACA algorithm framework in Section 2, and explain the underlying key ideas in Section 3-5. The practical time and space performances of OSACA are evaluated by the experiment on a set of typical corpora in Section 6, and the main results are summarized in Section 7.

2 OSACA

2.1 Framework

Fig. 1 shows the framework of OSACA, which is similar to that of SA-IS in [18]. In short, both OSACA and SA-IS first sample all the LMS-substrings of S, sort them, and then replace each LMS-substring by an integer name to produce a new shortened string S_1 (which is at least 1/2 shorter than S, i.e. $n_1 \leq \lfloor n/2 \rfloor$, see Lemma 3.5 in [18]) for computing the suffix array of S recursively. Given a

OSACA($S, SA, k, n, level$)
▷ S: input string;
▷ SA: array for storing $SA(S)$;
▷ K: alphabet size of S;
▷ n: size of S;
▷ $level$: recursion level;

▷ Stage 1: induced sort the LMS-substrings of S
1 **if** $level = 0$
then
2 Allocate an array of K integers for bkt;
3 Induced sort all the LMS-substrings of S, using bkt for bucket counters;
else
4 Induced sort all the LMS-substrings of S, reusing the head or the end of each bucket as the bucket's counter;

▷ SA is reused for storing S_1 and SA_1

▷ Stage 2: name the sorted LMS-substrings of S
5 Compute the lexicographic names for the sorted LMS-substrings to produce the reduced string S_1;

▷ Stage 3: sort recursively
6 **if** $K_1 = n_1$ ▷ each character in S_1 is unique
then
7 Directly compute $SA(S_1)$ from S_1;
else
8 OSACA($S_1, SA_1, k_1, n_1, level + 1$);

▷ Stage 4: induced sort $SA(S)$ from $SA(S_1)$
9 **if** $level = 0$
then
10 Induced sort $SA(S)$ from $SA(S_1)$, using bkt for bucket counters;
11 Free the space allocated for bkt;
else
12 Induced sort $SA(S)$ from $SA(S_1)$, reusing the head or the end of each bucket as the bucket's counter;
13 **return** ;

Fig. 1: The OSACA algorithm framework.

same S, both SA-IS and OSACA will sample the same set of LMS-substrings to compute the new shortened string S_1. Hence from level to level, S_1 produced by OSACA is identical with that from SA-IS. As a result, OSACA will output the same suffix array as that from SA-IS, in the same linear time complexity of $T(n) = T(\lfloor n/2 \rfloor) + O(n) = O(n)$ as that of SA-IS too.

In SA-IS, the working space is mainly composed of the bucket counter array bkt and the type array t at each recursion level. However, in OSACA, bkt is used only at the top recursion level (i.e. level 0), and t is used only at the deeper recursion levels (i.e. 1, 2 and so on). This enables the algorithm to achieve an $O(1)$ working space for an input string S over a constant alphabet of size $K = O(1)$. In Fig. 1, we allocate an array of K integers for bkt in line 2 only at level 0, where each integer is of a constant size[1]. In addition, no space is allocated for t in this algorithm. This is because that wherever t is needed at the deeper recursion levels, as will be shown, we can always reuse a space in SA for it. Hence, the working space for OSACA is dominated by bkt allocated at level 0 with a size of K integers, which is $O(1)$ for a constant alphabet.

Some more notations are introduced here for further presentation of OSACA. To denote a symbol in OSACA at level $i \geq 0$, we add "(i)" to the symbol's subscript, e.g. $S_{(i)}$ and $S_{1(i)}$ for S and S_1 at level i, respectively. Further, let $SA(S_{(i)})$ denote the suffix array of $S_{(i)}$, and $SA_{(i)}$ be the space for storing $SA(S_{(i)})$. That is, the notation $SA(S_{(i)})$ means that all the suffixes of $S_{(i)}$ are already sorted and stored in $SA_{(i)}$; however, the notation $SA_{(i)}$ means *only* the space for storing $SA(S_{(i)})$, regardless of what and how the data are stored. Notice that due to the recursion, $S_{1(i)}$ and $SA_{1(i)}$ are actually $S_{(i+1)}$ and $SA_{(i+1)}$, respectively.

2.2 Reusing $SA_{(0)}$

The space of SA at level 0, i.e. $SA_{(0)}$, is reused throughout all the recursion levels of OSACA in Fig. 1 to provide the space demanded at each level. In Fig. 2, the upper and the lower 3 rows show the statuses of $SA_{(0)}$ immediately before and after the recursive call at line 8 on levels 0-2, respectively.

At level 0 shown in the 1st row, $S_{(1)}$ (i.e. $S_{1(0)}$) is stored in the rightmost $n_{(1)}$ items in $SA_{(0)}$ (i.e. $SA_{(0)}[n_{(0)} - n_{(1)}, n_{(0)} - 1]$), where $n_{(i)}$ is the size of $S_{(i)}$, and the first $n_{(0)} - n_{(1)} \geq n_{(1)}$ items in $SA_{(0)}$ are unoccupied and can be reused for $SA_{(1)}$ (recalling $n \geq 2n_1$ at each level). At level 1 shown in the 2nd row, $S_{(2)}$ is stored immediately on the left hand side of $S_{(1)}$, and the leftmost $n_{(0)} - n_{(1)} - n_{(2)} \geq n_{(2)}$ items are free and can be reused for $SA_{(2)}$. Keep on recursively reducing the string from level to level, at level i, the sub-array $SA_{(0)}[0..n_{(i+1)} - 1]$ is always free and enough for the space required for $SA_{(i+1)}$.

Suppose that we are now at line 9 (in Fig. 1) for level 2. At this point, $SA(S_{(3)})$ has been computed and stored in $SA_{(3)}$ which is reusing $SA_{(0)}[0, n_{(3)} - 1]$ as shown by row 4. Then, $SA(S_{(2)})$ is induced sorted from $SA(S_{(3)})$ and stored in $SA_{(2)}$ which is reusing $SA_{(0)}[0, n_{(2)} - 1]$ by line 12. Further in line 13, we return to the upper recursion level and reach line 9 for level 1, and now the status of $SA_{(0)}$ is shown by row 5. Then, we continue to compute $SA(S_{(1)})$ from $SA(S_{(2)})$ by line 12, and get $SA(S_{(1)})$ stored in $SA_{(1)}$ shown in the last row when we reach line 9 at level 0. Finally, $SA(S_{(0)})$ is induced sorted from $SA(S_{(1)})$ by line 10 to produce the output suffix array.

1. To be precisely, each integer requires $\lceil \log_2 n \rceil$ bits. However, 64 bits are sufficient for $n \leq 2^{64}$ that is huge enough to cover any application in concern on modern computer architectures. In other words, the size of each item of *bkt* can be safely counted as $O(1)$.

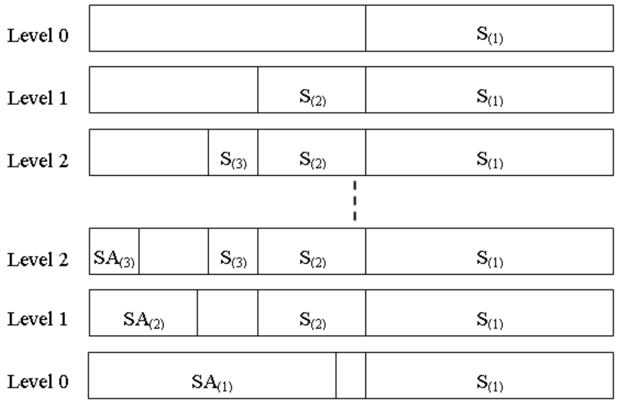

Fig. 2: Reusing $SA_{(0)}$ in the OSACA algorithm.

2.3 Induced Sorting

Notice that after level 0, OSACA will follow a common execution path for levels 1, 2 and etc. Hence, it is enough for us to explain OSACA for levels 0 and 1 only. The detail algorithms for induced sorting the suffixes at levels 0 and 1 are different, however, they can be fit into the following common algorithm framework. At each level, provided that all the LMS-suffixes of S have been sorted and stored in SA_1 which is reusing $SA[0, n_1 - 1]$, we can induced sort all the suffixes of S by this 4-step procedure:

1) Initialize each item of $SA[n_1, n-1]$ as empty;
2) Scan $SA[0, n_1 - 1]$ once from right to left to put all the sorted LMS-suffixes of S into their buckets in SA, from the end to the head in each bucket.
3) Scan SA once from left to right, for each $SA[i] > 0$ and $j = SA[i] - 1$, if $S[j]$ is L-type, then put $suf(S, j)$ into the current leftmost empty position in $bucket(SA, S, j)$.
4) Scan SA once from right to left, for each $SA[i] > 0$ and $j = SA[i] - 1$, if $S[j]$ is S-type, then put $suf(S, j)$ into the current rightmost empty position in $bucket(SA, S, j)$.

The above algorithm can be reused to induced sort all the LMS-substrings of S, by keeping the last two steps unchanged and modifying the first two steps as following:

1) Initialize each item of $SA[0, n-1]$ as empty;
2) Scan S once from right to left to put all the LMS-substrings of S into the buckets for their first characters, i.e., $lms(S, i)$ is put into $bucket(SA, S, i)$, from the end to the head in each bucket.

Each individual step of the aforementioned algorithms for induced sorting suffixes and LMS-substrings is trivially observed to run in $O(n)$ time, resulting in a total time complexity of $O(n)$ for both algorithms. Because we assume neither the type array t for level 0 nor the bucket counter array bkt for level 1, the last 3 steps for the induced sorting algorithms on levels 0 and 1 are different. Specifically, the major differences reside in the last 2 steps for (1) how to determine the type of $S[j]$

when we are scanning a non-empty item $SA[i]$, and (2) how to keep track of the current leftmost or rightmost positions of each bucket. Since the last 2 steps for induced sorting suffixes and LMS-substrings at each level are identical, and the first 2 steps are straightforward, we will concentrate on presenting the algorithms for induced sorting suffixes at levels 0 and 1, respectively.

3 Sorting Suffixes at Level 0

Different from SA-IS where an n-bit type array t is available at each level for induced sorting suffixes, there is no the type array t in OSACA at level 0. With this constraint, the general algorithm for induced sorting suffixes in Section 2.3 is further developed as following:

1) Initialize each item of $SA[n_1, n-1]$ as empty.
2) Compute into bkt the end position of each bucket in SA. Scan $SA[0, n_1-1]$ once from right to left to put all the sorted LMS-suffixes of S into their buckets in SA, from the end to the head in each bucket, in this way: for each scanned item $SA[i]$, let $j = SA[i]$ and $c = S[j]$, set $SA[bkt[c]] = j$ and decrease $bkt[c]$ by 1.
3) Compute into bkt the head position of each bucket in SA. Scan SA once from left to right to induced sort the L-type suffixes of S into their buckets in SA, from the head to the end in each bucket, in this way: for each scanned non-empty item $SA[i] > 0$, let $j = SA[i]-1$ and $c = S[j]$, if $S[j]$ is L-type, then set $SA[bkt[c]] = j$ and increase $bkt[c]$ by 1.
4) Compute into bkt the end position of each bucket in SA. Scan SA once from right to left to induced sort the S-type suffixes of S into their buckets in SA, from the end to the head in each bucket, in this way: for each scanned non-empty item $SA[i] > 0$, let $j = SA[i]-1$ and $c = S[j]$, if $S[j]$ is S-type, then set $SA[bkt[c]] = j$ and decrease $bkt[c]$ by 1.

In the last two steps of the above algorithm, how to determine the type of $S[j]$ without the type array t? For the 3rd step, because each non-empty item $SA[i]$ stores either a LMS-suffix or a L-type suffix, $S[j]$ must be L-type if we see $S[j] \geq S[SA[i]]$. However, for the 4rd step, we need to utilize the following property to help determine if $S[j]$ is S-type or not when we see $S[j] = S[SA[i]]$. In this property, $bkt[S[j]] < i$ means that the newly induced sorted S-type suffix must be stored into an item in front of (i.e. on the left hand side of) the non-empty item $SA[i]$ that we are currently scanning.

Property 3.1: When induced sorting the S-type suffixes of S from the sorted L-type suffixes, for each $SA[i] > 0$ and $j = SA[i]-1$, $suf(S, j)$ is S-type if and only if (1) $S[j] < S[SA[i]]$ or (2) $S[j] = S[SA[i]]$ and $bkt[S[j]] < i$.

4 Sorting Suffixes at Deeper Levels

Recalling that at level 0, we didn't need any type array t for sorting LMS-substrings and suffixes. However, because we reuse a sub-array of n_1 items in SA for storing the reduced string S_1, one item per character of S_1, there is always at least a bit unused in each item. Refer to Lemma 3.5 in [18], there must be $n_1 \leq \lfloor n/2 \rfloor$, which implies that a size of $\lceil \log_2 n \rceil - 1$ bits is enough for coding each character of S_1. Because each item of SA has a size of at least $\lceil \log_2 n \rceil$ bits, at least one bit per item is therefore not used for storing a character of S_1. Without loss of generality, let's suppose that the highest bit of each item is not used. Hence, for a string S at level $i > 0$, each character is stored with its type together in $\lceil \log_2 n \rceil$ bits: the highest bit for the type of the character, and the rest bits for the character.

At each recursion level of OSACA, bucket sorting is employed to induced sort both LMS-substrings and suffixes. At level 0, since an alphabet size of $K = O(1)$ is assumed for S, we can use $O(1)$ space to store a bucket counter array bkt for induced sorting when reducing S into S_1, as well as augmenting $SA(S_1)$ to $SA(S)$. However, at level 1, if we still use a specific bucket counter array for bucket sorting, the bucket counter array will require $O(n \log_2 n)$ bits in total. In order to achieve a working space of $O(1)$, no specific bucket counter array should be used for bucket sorting at levels 1, 2 and thereafter. Fortunately, we have found a novel way for induced sorting using no specific bucket counter array, in case of the following property is held.

Property 4.1: For $OSACA$ at level $i > 0$, each L-type or S-type character in S points to the head or the end of its bucket in SA, respectively.

In Section 5, we will show how to produce S with this property. Now, given this property for S at level 1, we show how to compute $SA(S)$ without using a separate bucket counter array. At level 1, after we have returned from the recursion call, line 12 in Fig. 1 will be executed to induced sort $SA(S)$ from $SA(S_1)$ (notice that S holds Property 4.1 now). As shown in Section 2.3, induced sorting suffixes consists of 4 steps, where the first 2 steps for initialization and putting all the sorted LMS-suffixes into their buckets in SA are not hard to be done at level 1, even without the bucket counter array. The tough task is the last 2 steps for induced sorting the L-type and S-type suffixes, respectively. We continue to describe the detail algorithms for these two steps.

4.1 Induced Sorting L-Type Suffixes

Without the bucket counter array bkt that we had for induced sorting the L-type suffixes at level 0 in Section 3, the algorithm for induced sorting the L-type suffixes at level 1 relies on Property 4.1. The key idea is to reuse the head item of each bucket in SA to maintain a counter for tracking the location where a L-type suffix being sorted into this bucket should be stored. Notice that at any level $i > 0$, each item of SA is reusing an item of $SA_{(0)}$ and the highest bit in each item is not needed to store the index of a suffix in S (due to $n_1 \leq \lfloor n/2 \rfloor$ at each level). Hence, at level $i > 0$, the highest bit of $SA[i]$ is always available to be used for indicating what data is currently stored in the rest bits of $SA[i]$: 0 for a suffix index, 1 for a bucket counter or empty value.

At the beginning of line 12 in Fig. 1, an item in SA may be empty (marked by the least negative integer denoted by $EMPTY$) or store a non-negative value for the index of a LMS-suffix in S, and all the LMS-suffixes stored in SA have been sorted in their correct order. To induced sort all the L-type suffixes, we scan SA once from left to right to do the following. For each $SA[i] > 0$ being scanned, let $j = SA[i] - 1$, if $S[j]$ is L-type[2], we will put $suf(S, j)$ into its bucket in SA. Recalling that S in this case holds Property 4.1, so $S[j]$ points to the head of its bucket in SA. That is, let $c = S[j]$, the head of $bucket(SA, S, j)$ is $SA[c]$. To indicate an item in SA is being reused as a bucket counter, the value stored in this item is set as a non-empty negative value. Now, we check the value of $SA[c]$ for these cases:

1) If we see $SA[c]$ empty, then $suf(S, j)$ is the 1st suffix being put into its bucket. In this case, we further check $SA[c+1]$ to see if it is empty or not. If it is, we sort $suf(S, j)$ into $SA[c+1]$ by setting $SA[c+1] = j$ and start to reuse $SA[c]$ as a counter by setting $SA[c] = -1$. Otherwise, $SA[c+1]$ may be non-negative for a suffix index or negative for a counter, and $suf(S, j)$ must be the only element of its bucket, we hence simply put $suf(S, j)$ into its bucket by setting $SA[c] = j$.
2) If we see $SA[c]$ non-negative, then $SA[c]$ is "borrowed" by the left-neighboring bucket (of $bucket(SA, S, j)$). In this case, $SA[c]$ is storing the largest item in the left-neighboring bucket, and we need to shift-left one step of all the items in the left-neighboring bucket to their correct locations in SA. The head item of the left-neighboring bucket can be found by scanning from $SA[c]$ to the left, until we see the first item $SA[x]$ that is negative for being reused as a counter. That is, x is the largest for $SA[x] < 0$, $SA[x] \neq EMPTY$ and $x < c$. Having found $SA[x]$, we shift-left one step all the items in $SA[x+1, c]$, and set $SA[c]$ as empty. Now, we see the same condition as that in case 1, hence the operations in case 1 are performed to further sort $suf(S, j)$ into its bucket.
3) If we see $SA[c]$ negative and non-empty, then $SA[c]$ is being reused as a counter for $bucket(SA, S, j)$. In this case, let $d = SA[c]$ and $pos = c - d + 1$, then $SA[pos]$ is the item that $suf(S, j)$ should be stored into. However, $suf(S, j)$ may be the largest suffix in its bucket. Therefore, we further check the value of $SA[pos]$ to proceed as following. If $SA[pos]$ is empty, we simply put $suf(S, j)$ into its bucket by setting $SA[pos] = j$, and increase the counter of its bucket by 1, i.e. $SA[c] = SA[c] - 1$ (notice that $SA[c]$ is negative for a counter). Otherwise, it indicates that $SA[pos]$ is the head item of the right-neighboring bucket, which must be currently non-negative for a suffix index or negative for a counter. Hence, we need to shift-left one step the items in $SA[c+1, pos-1]$, then sort $suf(S, j)$ into its bucket by setting $SA[pos-1] = j$.

In the algorithm described above, because we reuse the head item of a bucket as a counter for recording how many L-type suffixes are already stored in the bucket, it is possible that the largest suffix of a bucket is temporarily put into the head item of its right-neighboring bucket. In other words, the rightmost item of a bucket runs into the head item of the right-neighboring bucket. Hence, in case 2, if we see $SA[c]$ non-negative for a suffix index, it means that $SA[c]$ is borrowed by the largest suffix in the left-neighboring bucket (of $bucket(SA, S, j)$). Hence, we need to adjust all the items of the left-neighboring bucket to their correct locations. This is done by shifting left one step all the items in the left-neighboring bucket, where the head of the left-neighboring bucket is currently the 1st non-empty negative item in front of $SA[c]$. Notice that in cases 2 and 3, the suffixes in a bucket are shifted left only when the bucket is fully filled. In other words, no other suffix will be sorted into the bucket thereafter. Hence, the counter for this bucket is not needed any more. Shifting left a bucket in case 3 is simpler than that in case 2, for we have already known the exact positions for the first and the last items of the bucket.

The time complexity of this algorithm is determined by the loop for scanning SA once to perform the induced sorting operations. Each iteration of this loop will sort at most a L-type suffix into SA, and each L-type suffix already sorted into SA can be shifted at most once. Hence, this loop has a time complexity dominated by the loop's size, i.e. $O(n)$.

4.2 Induced Sorting S-Type Suffixes

Given all the L-type suffixes of S are already sorted into their correct positions in SA, we can scan SA once from right to left to induced sort all the S-type suffixes. When induced sorting the L-type suffixes, the head item of each bucket is reused as a counter for the bucket. However, to induced sort the S-type suffixes, because we are now scanning SA in a reverse direction, i.e. from right to left, and each S-type character of S points the end of its bucket in SA, it is now the end item instead of the head item of a bucket is reused as the counter for the bucket. Hence, with some minor and symmetric changes to that for induced sorting the L-type suffixes, here comes the algorithm for induced sorting the S-type suffixes from the sorted L-type suffixes.

We scan SA once from right to left to do the following. For each non-negative $SA[i]$, let $j = SA[i] - 1$, if $S[j]$ is S-type, we will put the suffix $suf(S, j)$ into its bucket in SA. Recalling that S in this case holds Property 4.1, so $S[j]$ points to the end of its bucket in SA. That is, let $c = S[j]$, the end of $bucket(SA, S, j)$ is $SA[c]$. Now, we check the value of $SA[c]$ for these cases:

2. Notice that at each level after the top level, the type of each character in S is stored together with the character itself, see the 1st paragraph of Section 4.

1) If we see $SA[c]$ empty, then $suf(S, j)$ is the first suffix being put into its bucket. In this case, we further check $SA[c-1]$ to see if it is empty or not. If it is, we sort $suf(S, j)$ into $SA[c-1]$ by setting $SA[c-1] = j$ and start to reuse $SA[c]$ as a counter by setting $SA[c] = -1$. Otherwise, $SA[c-1]$ may be non-negative for a suffix index or negative for a counter, and $suf(S, j)$ must be the only element of its bucket, we hence simply put $suf(S, j)$ into its bucket by setting $SA[c] = j$.
2) If we see $SA[c]$ non-negative, then $SA[c]$ is "borrowed" by the right-neighboring bucket (of $bucket(SA, S, j)$). In this case, $SA[c]$ is storing the smallest item in the right-neighboring bucket, and we need to shift-right one step all the items in the right-neighboring bucket to their correct locations in SA. The end item of the right-neighboring bucket can be found by scanning from $SA[c]$ to the right, until we see the first item $SA[x]$ that is negative for being reused as a counter. That is, x is the smallest for $SA[x] < 0$, $SA[x] \neq EMPTY$ and $x > c$. Having found $SA[x]$, we shift-right one step all the items in $SA[c, x-1]$, and set $SA[c]$ as empty. Now, we see the same condition as that in case 1, hence the operations in case 1 are performed to further sort $suf(S, j)$ into its bucket.
3) If we see $SA[c]$ negative and non-empty, then $SA[c]$ is reused as a counter for $bucket(SA, S, j)$. In this case, let $d = SA[c]$ and $pos = c + d - 1$, then $SA[pos]$ is the item that $suf(S, j)$ should be stored into. However, $suf(S, j)$ may be the smallest S-type suffix in its bucket. Therefore, we further check the value of $SA[pos]$ to proceed as following. If $SA[pos]$ is empty, we simply put $suf(S, j)$ into its bucket by setting $SA[pos] = j$, and increase the counter of its bucket by 1, i.e. $SA[c] = SA[c]-1$ (notice that $SA[c]$ is negative for a counter). Otherwise, $SA[pos]$ must be currently non-negative for a suffix index or negative for a counter. Hence, we need to shift-right one step the items in $SA[pos+1, c-1]$, then sort $suf(S, j)$ into its bucket by setting $SA[pos+1] = j$.

5 NAMING SORTED LMS-SUBSTRINGS

We now turn to the key issue of how to calculate the names for the sorted LMS-substrings of S to get a new reduced string S_1 (which is the input string S at the next level) with Property 4.1 . Given that all the LMS-substrings of S have been sorted into SA_1 (which is reusing $SA[0, n_1 - 1]$), we employ the following novel naming method to produce S_1 in a linear time complexity of $O(n)$. **Notice that in this section, each set of identical LMS-substrings in S constitutes a bucket in SA_1**, such a bucket definition for LMS-substrings is different from that for suffixes and characters in Section 1.

1) Scan SA_1 once from left to right to name each LMS-substring of S by the head position of the substring's bucket in SA_1, resulting in an interim reduced string denoted by Z_1 (where each character points to the head of its bucket in SA_1);
2) Scan Z_1 once from right to left to replace each S-type character in Z_1 by the end position of its bucket in SA_1.

As a result, we now get the reduced string S_1, in which each L-type or S-type character points to the head or the end of the character's bucket in SA_1, respectively. However, there is still a key problem to be solved in this naming algorithm. To detect the head of each bucket in the first step, we need to compare any two neighboring LMS-substrings of S stored in SA_1. At recursion level 0, without the type array t, how to determine the ends of two LMS-substrings when they are compared? Because the type of $suf(S, i-1)$ is relied on the type of $suf(S, i)$ when $S[i-1] = S[i]$ (see Section 1), this constitutes a difficulty for determining the end of a LMS-substring by traversing from the head of the LMS-substring. However, fortunately, we can still traverse a LMS-substring from its head to detect its end by utilizing the following observation.

A LMS-substring has a type pattern governed by this regular expression S^+L^+S, where S^+ and L^+ mean a string of one or multiple S-type and L-type characters, respectively. In other words, a LMS-substring consists of three segments in sequence: one or multiple S-type characters, one or multiple L-type characters, and a single S-type character. Suppose that we are going to retrieve the length of $lms(S, x)$ (i.e. a LMS-substring starting at $S[x]$), this LMS-substring together with its succeeding LMS-substring will follow such a pattern $S^+L^+S^+L^+S$ (notice that two neighboring LMS-substrings must overlap on a common LMS-character). This fact is utilized to design the following 2-step algorithm for retrieving $lms(S, x)$ from $S[x]$. (1) Traverse the LMS-substring from its first character $S[x]$ until we see a character $S[x+i]$ less than its preceding $S[x+i-1]$. Now, $S[x+i-1]$ must be a L-type character. (2) Continue to traverse the rest character(s) of the LMS-substring and terminate when we see a character $S[x+i]$ greater than its preceding $S[x+i-1]$ or $S[x+i]$ is the sentinel. At this point, we know that the head of the succeeding LMS-substring has been traversed and its position was previously recorded when we saw $S[x+i] < S[x+i-1]$ the last time.

Here comes a running example for the above algorithm. Suppose that we have two neighboring LMS-substrings "suffix0", where the 1st and the 2nd LMS-substrings are "suf" and "ffix0", respectively. Starting from the character "s", the 1st step traverses the character "u", then breaks when the 1st character "f" is seen, for "f" < "u". Further in the 2nd step, the next two characters "f", "i" are traversed. When the 1st "f" is visited, its position is saved, for "f" < "u" and it is probably the head of the 2nd LMS-substring. However, when the 2nd "f" is approached, we don't save its position, for it must not be the head of the 2nd LMS-substring (suppose that it is, then the 1st "f" must be S-type and hence the head of the 2nd LMS-substring instead, resulting in a contradiction). When we reach the character "i", because "i" > "f", the traversing is terminated and the 1st "f" is confirmed to be the end of the 1st LMS-substring.

5.1 Correctness

In the SA-IS algorithm [18], having sorted and stored in SA_1 all the LMS-substrings of S, we name each LMS-substring by the *index of its bucket* in SA_1 to produce the reduced string called Y_1 here, where the buckets in SA_1 are indexed from 0. If we name each LMS-substring by the *head position of its bucket* instead to produce another string Y_2 (i.e. Z_1 in our new naming algorithm), then for any $Y_1[i] < Y_1[j]$ or $Y_1[i] = Y_1[j]$, we must have $Y_2[i] < Y_2[j]$ or $Y_2[i] = Y_2[j]$, respectively. Therefore $SA(Y_1)$ and $SA(Y_2)$ must be identical. Further, we rename each S-type character in Y_2 by the end position of its bucket instead to produce yet another string called Y_3. Now for any $Y_2[i] < Y_2[j]$, there must be $Y_3[i] < Y_3[j]$. In case of $Y_2[i] = Y_2[j]$, we further look into two more cases in respect to whether the types of $Y_2[i]$ and $Y_2[j]$ are the same. If so, we must have $Y_3[i] = Y_3[j]$; or else without loss of generality, suppose $Y_2[i]$ and $Y_2[j]$ are L-type and S-type, respectively, we must have $Y_3[i] < Y_3[j]$, $suf(Y_2, i) < suf(Y_2, j)$ and $suf(Y_3, i) < suf(Y_3, j)$. Hence $SA(Y_2)$ and $SA(Y_3)$ must be identical too. Given $SA(Y_1)$ and $SA(Y_3)$ are identical, because Y_1 and Y_3 are in effect the two S_1 produced by SA-IS and OSACA, respectively, we get that $SA(S_1)$ and therefore $SA(S)$ computed by both algorithms must be identical.

6 Performance

In [18], the time and space performances of our SA-IS were compared with that of KS [15] and KA [14], and observed to take much advantages over the latter. Hence, OSACA is evaluated against SA-IS under the same settings. The datasets used in this experiment are listed in Table 1, they are a subset of the Canterbury [20], Calgary[21] and Manzini [9] corpora. The performance measures to be investigated are the time and space consumptions for each algorithm running on the datasets. The machine is a DELL(R) PowerEdge(R) 1950 server with such a configuration: 1 Intel(R) Xeon(R) CPU (E5410, 2.33GHz, cache size 6144 KB), 4GB DDR2 667MHz ECC RAM, Red Hat Enterprise Linux Server release 5.2 (Tikanga) 32-bit. Both algorithms are compiled by `g++` with options "-fomit-frame-pointer -W -Wall -Winline -DNDBUG -O3".

With these settings, each integer has a size of 4 bytes. In this case, OSACA always uses a total space of $5n + O(1)$ bytes, whereas SA-IS may require a total space up to $7.125n + O(1)$ bytes (refer to Corollary 3.14 in [18]) in the worst case. In other words, **the largest working spaces for OSACA and SA-IS are $O(1)$ and $2.125n + O(1)$ bytes, respectively**. In Table 2, for each algorithm running on a corpus, Φ is the heap peak and $\Delta = \Phi - 5n$ is the working space, where the heap peak is collected by using the command `memusage` to start running the algorithm. For each algorithm, the total space is the sum of all the space consumptions for running the algorithm on the corpora, and the mean is the total space divided by the total number of characters. We see that the working spaces of SA-IS on the corpora are varying with a mean of $0.323n$ bytes. However, the working space of OSACA on each corpus remains a constant of 1029 bytes, this is well coincident with our previous analysis for the space consumption of OSACA, i.e. $O(1)$ working space.

Notice that the working space of OSACA is mainly composed of bkt at recursion level 0. In this experiment, a fixed-size bkt of 256 integers is employed for all the corpora independent of the exact alphabet size of each individual corpus. Given 4 bytes for each integer, bkt in this case consumes 1024 bytes in total. In the introduction section, we assumed a sentinel of numeric value 0 in S for presentation simplicity. However, when we coded the program used in our experiment, we allowed the numeric value 0 to appear inside S, and we appended an additional 0 to S as the *virtual* sentinel. Hence, we actually computed the suffix array for an input string of $n + 1$ characters. This results in 5 more bytes for the working space: 1 byte for the added virtual sentinel, 4 bytes in the suffix array for recording the suffix starting at the virtual sentinel. As a result, we have a total of 1029 bytes for the working space.

To see the time performances of both algorithms, we show in Table 3 the running times for both algorithms on the corpora, where the time for each algorithm on a corpus is the mean of 3 runs, and the speedup is defined as the speed ratio of OSACA vs. SA-IS at each row (including the two rows of "Total" and "Mean"), i.e. $Time_{sa-is}/Time_{osaca}$. For each algorithm, the total time is the sum of times for running the algorithm on the corpora, and the mean is the total time divided by the total number of characters in units of MB. From this table, OSACA is observed to be running about 30% faster than SA-IS on average, i.e. a mean speedup of 1.29. The speed improvement is mainly due to that at each level $i > 0$, we need not scan S to find the head or the end of each bucket in SA, for Property 4.1 is held for S and hence each L-type or S-type character in S directly tells the head or the end of its bucket in SA, respectively. However, in SA-IS, we need to scan S 6 times to compute the bucket counter array: 3 times for induced sorting the LMS-substrings, and 3 times for induced sorting the suffixes. As a summary, OSACA not only consumes less space than SA-IS, but also runs faster.

7 Conclusion

Each step of OSACA in Fig. 1 has a linear time complexity of $O(n)$, so the total time remains linear as that of SA-IS, i.e. $T(n) = T(\lfloor n/2 \rfloor) + O(n) = O(n)$. For the space complexity of OSACA, besides S and SA, we have an

TABLE 1: Corpora

Corpus	Characters; Alphabet Size; Description
alphabet.txt	100,000; 26; Repetitions of the alphabet [a-z]
bible.txt	4,047,392; 63; King James Bible
chr22.dna	34,553,758; 5; Human chromosome 22
E.coli	4,638,690; 4; Escherichia coli genome
etext99	105,277,340; 146; Texts from Gutenberg project
gcc-3.0.tar	86,630,400; 150; Tar archive of gcc 3.0 source files
howto	39,422,105; 197; Linux Howto files
linux-2.4.5. tar	116,254,720; 256; Tar archive of Linux kernel 2.4.5 source files
obj2	246,814; 256; Object code for Apple Mac
pic	513,216; 159; Black and white fax picture
pi.txt	1,000,000; 10; The first million digits of π
random.txt	100,000; 64; Randomly chosen from 64 characters
rfc	116,421,901; 120; Concatenation of RFC text files
sprot34.dat	109,617,186; 66; Swissprot V34 protein database
w3c2	104,201,579; 256; Concatenation of html files from www.w3c.org
world192.txt	2,473,400; 94; CIA world fact book

TABLE 2: Space (Bytes)

Corpus	SA-IS		OSACA	
	Φ	Δ	Φ	Δ
alphabet.txt	513530	13530	501029	1029
bible.txt	21878497	1641537	20237989	1029
chr22.dna	186744357	13975567	172769819	1029
E.coli	25472640	2279190	23194479	1029
etext99	568510724	42124024	526387729	1029
gcc-3.0.tar	459740230	26588230	433153029	1029
howto	213026794	15916269	197111554	1029
linux-2.4.5.tar	619235576	37961976	581274629	1029
obj2	1354252	120182	1235099	1029
pic	2690903	124823	2567109	1029
pi.txt	5591695	591695	5001029	1029
random.txt	644271	144271	501029	1029
rfc	617284514	35175009	582110534	1029
sprot34.dat	581517422	33431492	548086959	1029
w3c2	544641749	23633854	521008924	1029
world192.txt	13318739	951739	12368029	1029
Total	3862165893	-	3627508969	-
Mean	5.323	-	5.000	-

TABLE 3: Time (Seconds)

Corpus	SA-IS	OSACA	Speedup
alphabet.txt	0.01	0.006	1.67
bible.txt	0.928	0.789	1.18
chr22.dna	18.107	14.661	1.24
E.coli	1.145	0.974	1.18
etext99	79.612	58.75	1.36
gcc-3.0.tar	41.25	33.145	1.24
howto	20.464	16.701	1.23
linux-2.4.5.tar	58.427	45.415	1.29
obj2	0.037	0.034	1.09
pic	0.05	0.039	1.28
pi.txt	0.173	0.151	1.15
random.txt	0.016	0.015	1.07
rfc	69.044	51.737	1.33
sprot34.dat	73.835	55.18	1.34
w3c2	53.29	44.341	1.20
world192.txt	0.443	0.375	1.18
Total	416.829	322.315	1.29
Mean	0.602	0.466	1.29

additional array bkt of size $O(k)$ at recursion level 0 *only*. Hence we have the following result:

Lemma 7.1: For a string S of n characters over a constant alphabet of size $K = O(1)$, OSACA needs only a working space of $O(1)$ for constructing the suffix array of S in a linear time of $O(n)$.

A number of linear-time algorithms have been proposed for computing the suffix array of S, among them, our previously proposed algorithm SA-IS was evaluated to achieve the best time and space performances. Compared to SA-IS, the novelty of OSACA consists in a number of new techniques for removing the type array from sorting at the top level and the bucket counter array from sorting at the deeper levels, respectively. OSACA overwhelms SA-IS by running much faster and using only a constant working space $O(1)$ for a string of an alphabet size ≤ 256, e.g. $256 \times 4 + 4 + 1 = 1029$ and $256 \times 5 + 5 + 1 = 1286$ bytes for the string size less than 2^{32} and 2^{40}, respectively. Such a small and constant working space has approached the limit for a linear-time SACA. The proposed OSACA not only runs the fastest and uses the least space among all the existing linear-time SACAs, but also achieves a deterministic constant working space in both theory and practice.

REFERENCES

[1] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," in *Proceedings of SODA*, 1990, pp. 319–327.

[2] ——, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[3] K. Sadakane, "A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation," in *Proceedings DCC '98 Data Compression Conference*, Snowbird, UT, USA, Mar. 1998, pp. 129–138.

[4] H. Itoh and H. Tanaka, "An efficient method for in memory construction of suffix arrays," in *Proceedings of String Processing and Information Retrieval Symposium*, Sep. 1999, pp. 81–88.

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

[6] G. Manzini and P. Ferragina, "Engineering a lightweight suffix array construction algorithm," in *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02, Lecture Notes in Computer Science Vol.2461)*, Sep. 2002, pp. 698–710.

[7] S. Burkhardt and J. Kärkkäinen, "Fast lightweight suffix array construction and checking," in *Proceedings of CPM'03, LNCS 2676*, Jun. 2003, pp. 55–69.

[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] G. Manzini and P. Ferragina, "Engineering a lightweight suffix array construction algorithm," *Algorithmica*, vol. 40, no. 1, pp. 33–50, Sep. 2004.

[10] K. B. Schürmann and J. Stoye, "An incomplex algorithm for fast suffix array construction," in *Proceedings of 7th Workshop on Algorithm Engineering and Experiments (ALENEX/ANALCO 2005)*, 2005, pp. 77–85.

[11] M. A. Maniscalco and S. J. Puglisi, "Faster lightweight suffix array construction," in *Proceedings of 17th Australasian Workshop on Combinatorial Algorithms (AWOCA'06)*, 2006, pp. 16–29.

[12] 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.

[13] D. K. Kim, J. Jo, H. Park, and K. Park, "Constructing suffix arrays in linear time," *Journal of Discrete Algorithms*, vol. 3, no. 2-4, pp. 126–142, 2005.

[14] 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.
[15] J. Kärkkäinen, P. Sanders, and S. Burkhardt, "Linear work suffix array construction," *JACM*, vol. 53, no. 6, pp. 918–936, Nov. 2006.
[16] G. Nong, S. Zhang, and W. H. Chan, "Linear suffix array construction by almost pure induced-sorting," in *Proceedings of DCC*, U.S.A., Mar. 2009.
[17] ——, "Linear time suffix array construction using d-critical substrings," in *Proceedings of CPM*, France, Jun. 2009.
[18] ——, "Two efficient algorithms for linear time suffix array construction," *IEEE Transactions on Computers*, vol. 60, no. 10, Oct. 2011.
[19] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," in *Proceedings of CPM*, 2003, pp. 200–210.
[20] "Canterbury corpus." [Online]. Available: http://corpus.canterbury.ac.nz/
[21] I. Witten and T. Bell, "Calgary text compression corpus." [Online]. Available: ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/