Available online at www.sciencedirect.com
SCIENCE DIRECT®

Journal of Discrete Algorithms 3 (2005) 143–156

JOURNAL OF DISCRETE ALGORITHMS
www.elsevier.com/locate/jda

Space efficient linear time construction of suffix arrays [☆]

Pang Ko [a,*], Srinivas Aluru [a,b]

[a] *Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA*
[b] *Laurence H. Baker Center for Bioinformatics and Biological Statistics, Iowa State University, Ames, IA 50011, USA*

Available online 15 September 2004

Abstract

We present a linear time algorithm to sort all the suffixes of a string over a large alphabet of integers. The sorted order of suffixes of a string is also called suffix array, a data structure introduced by Manber and Myers that has numerous applications in pattern matching, string processing, and computational biology. Though the suffix tree of a string can be constructed in linear time and the sorted order of suffixes derived from it, a direct algorithm for suffix sorting is of great interest due to the space requirements of suffix trees. Our result is one of the first linear time suffix array construction algorithms, which improve upon the previously known $O(n \log n)$ time direct algorithms for suffix sorting. It can also be used to derive a different linear time construction algorithm for suffix trees. Apart from being simple and applicable for alphabets not necessarily of fixed size, this method of constructing suffix trees is more space efficient.

Keywords: Computational biology; Pattern matching; String algorithms; Suffix array; Suffix sorting

[☆] Research supported by IBM Faulty Award and NSF under ACI-0203782.

[*] Corresponding author.
E-mail addresses: kopang@iastate.edu (P. Ko), aluru@iastate.edu (S. Aluru).

1570-8667/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2004.08.002

1. Introduction

Suffix trees and suffix arrays are important fundamental data structures useful in many applications in string processing and computational biology. The suffix tree of a string is a compacted trie of all the suffixes of the string. The suffix tree of a string of length n over an alphabet Σ can be constructed in $O(n \log |\Sigma|)$ time and $O(n)$ space, or in $O(n)$ time and $O(n|\Sigma|)$ space [17–19]. These algorithms are suitable for small, fixed size alphabets. Subsequently, Farach [6] presented an $O(n)$ time and space algorithm for the more general case of constructing suffix trees over integer alphabets. For numerous applications of suffix trees in string processing and computational biology, see [8].

The suffix array of a string is the lexicographically sorted list of all its suffixes. Manber and Myers introduced the suffix array data structure [16] as a space-efficient substitute for suffix trees. Gonnet et al. [7] have also independently developed the suffix array, which they refer to as the PAT array. As a lexicographic-depth-first traversal of a suffix tree can be used to produce the sorted list of suffixes, suffix arrays can be constructed in linear time and space using suffix trees. However, this defeats the whole purpose if the goal is to avoid suffix trees. Hence, Manber and Myers presented a direct construction algorithm that runs in $O(n \log n)$ worst-case time and $O(n)$ expected time. Since then, the study of algorithms for constructing suffix arrays and for using suffix arrays in computational biology applications has attracted considerable attention.

The suffix array is often used in conjunction with another array, called *lcp* array, containing the lengths of the longest common prefixes between every pair of consecutive suffixes in sorted order. Manber and Myers also presented algorithms for constructing *lcp* array in $O(n \log n)$ worst-case time and $O(n)$ expected time, respectively [16]. More recently, Kasai et al. [12] presented a linear time algorithm for constructing the *lcp* array directly from the suffix array. While the classic problem of finding a pattern P in a string T of length n can be solved in $O(|P|)$ time for fixed size Σ using a suffix tree of T, Manber and Myers' suffix array based pattern matching algorithm takes $O(|P| + \log n)$ time, without any restriction on Σ. Recently, Abouelhoda et al. [2,3] have improved this to $O(|P|)$ time using additional linear time preprocessing, thus making the suffix array based algorithm superior. In fact, many problems involving top-down or bottom-up traversal of suffix trees can now be solved with the same asymptotic run-time bounds using suffix arrays [1–3]. Such problems include many queries used in computational biology applications including finding exact matches, maximal repeats, tandem repeats, maximal unique matches and finding all shortest unique substrings. For example, the whole genome alignment tool MUMmer [5] uses the computation of maximal unique matches.

While considerable advances are made in designing optimal algorithms for queries using suffix arrays and for computing auxiliary information that is required along with suffix arrays, the complexity of direct construction algorithms for suffix arrays remained $O(n \log n)$ so far. Several alternative algorithms for suffix array construction have been developed, each improving the previous best algorithm by an additional constant factor [10, 15]. We close this gap by presenting a direct linear time algorithm for constructing suffix arrays over integer alphabets. Contemporaneous to our result, Kärkkäinen et al. [11] and Kim et al. [13] also discovered suffix array construction algorithms with linear time complexity. All three algorithms are very different and are important because they elucidate different properties of strings, which could well be applicable for solving other problems. An important distinguishing feature of our algorithm is that it uses only $8n$ bytes plus $1.25n$ bits for a fixed size alphabet. Our algorithm is based on a unique recursive formulation where the subproblem size is not fixed but is dependent on the properties of the string. Recently, Hon et al. [9] discovered a linear time construction algorithm for compressed suffix array.

It is well known that the suffix tree of a string can be constructed from the sorted order of its suffixes and the *lcp* array [6]. Because the *lcp* array can be inferred from the suffix array in linear time [12], our algorithm can also be used to construct suffix trees in linear time for large integer alphabets, and of course, for the special case of fixed size alphabets. Our algorithm is simpler and more space efficient than Farach's linear time algorithm for constructing suffix trees for integer alphabets. In fact, it is simpler than linear time suffix tree construction algorithms for fixed size alphabets [17–19]. A noteworthy feature of our algorithm is that it does not construct or use suffix links, resulting in additional space advantage. To the best of our knowledge, all direct suffix tree construction algorithms that achieve linear run-time exploit the use of suffix links.

The remainder of this paper is organized as follows: In Section 2, we present our linear time suffix sorting algorithm. A detailed analysis of the space requirement of our algorithm is presented in Section 3. An implementation strategy that further improves the run-time in practice can be found in Section 4. We compare our algorithm with other previous work in Section 5. Section 6 concludes the paper.

2. Suffix sorting algorithm

Consider a string $T = t_1 t_2 \ldots t_n$ over the alphabet $\Sigma = \{1 \ldots n\}$. Without loss of generality, assume the last character of T occurs nowhere else in T, and is the lexicographically smallest character. We denote this character by '\$'. Let $T_i = t_i t_{i+1} \ldots t_n$ denote the suffix of T starting with t_i. To store the suffix T_i, we only store the starting position number i. For strings α and β, we use $\alpha \prec \beta$ to denote that α is lexicographically smaller than β. Throughout this paper the term *sorted order* refers to lexicographically ascending order.

A high level overview of our algorithm is as follows: We classify the suffixes into two types, S and L. Suffix T_i is of type S if $T_i \prec T_{i+1}$, and is of type L if $T_{i+1} \prec T_i$. The last suffix T_n does not have a next suffix, and is classified as both type S and type L. The positions of the type S suffixes in T partitions the string into a set of substrings. We substitute each of these substrings by its rank among all the substrings and produce a new string T'. This new string is then recursively sorted. The suffix array of T' gives the lexicographic order of all type S suffixes. Then the lexicographic order of all suffixes can be deduced from this order.

We now present complete details of our algorithm. The following lemma allows easy identification of type S and type L suffixes in linear time.

Lemma 1. *All suffixes of T can be classified as either type S or type L in $\mathrm{O}(n)$ time.*

Proof. Consider a suffix T_i $(i < n)$.

T	M	I	S	S	I	S	S	I	P	P	I	$
Type	L	S	L	L	S	L	L	S	L	L	L	L/S
Pos	1	2	3	4	5	6	7	8	9	10	11	12

Fig. 1. The string “MISSISSIPPI$” and the types of its suffixes.

Case 1: If $t_i \neq t_{i+1}$, we only need to compare t_i and t_{i+1} to determine if T_i is of type S or type L.
Case 2: If $t_i = t_{i+1}$, find the smallest $j > i$ such that $t_j \neq t_i$.
if $t_j > t_i$, then suffixes $T_i, T_{i+1}, \ldots, T_{j-1}$ are of type S.
if $t_j < t_i$, then suffixes $T_i, T_{i+1}, \ldots, T_{j-1}$ are of type L.

Thus, all suffixes can be classified using a left to right scan of T in $\mathrm{O}(n)$ time. □

The type of each suffix of the string MISSISSIPPI$ is shown in Fig. 1. An important property of type S and type L suffixes is, if a type S suffix and a type L suffix both begin with the same character, the type S suffix is always lexicographically greater than the type L suffix. The formal proof is presented below.

Lemma 2. *A type S suffix is lexicographically greater than a type L suffix that begins with the same first character.*

Proof. Suppose a type S suffix T_i and a type L suffix T_j are two suffixes that start with the same character c. We can write $T_i = c^k c_1 \alpha$ and $T_j = c^l c_2 \beta$, where c^k and c^l denotes the character c repeated for $k, l > 0$ times, respectively; $c_1 > c$, $c_2 < c$; α and β are (possibly empty) strings.

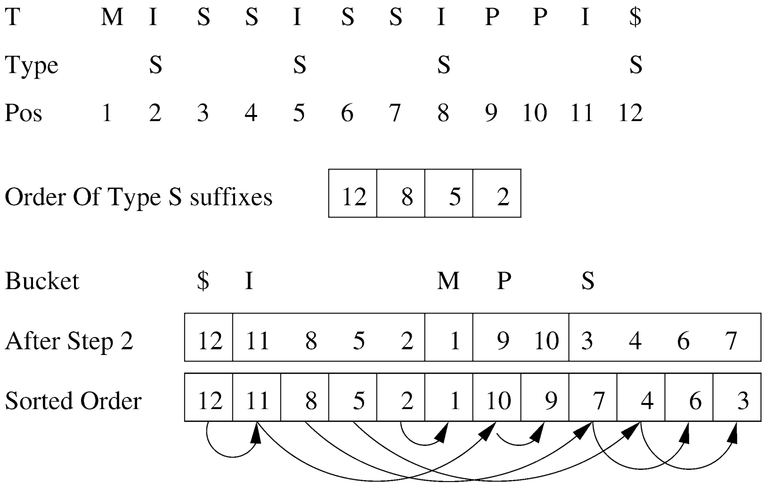
Case 1: If $k < l$ then c_1 is compared to a character c in c^l. Then $c_1 > c \Rightarrow T_j \prec T_i$.
Case 2: If $k > l$ then c_2 is compared to a character c in c^k. Then $c > c_2 \Rightarrow T_j \prec T_i$.
Case 3: If $k = l$ then c_1 is compared to c_2. Since $c_1 > c$ and $c > c_2$, then $c_1 > c_2 \Rightarrow T_j \prec T_i$.

Thus a type S suffix is lexicographically greater than a type L suffix that begins with the same first character. □

Corollary 3. *In the suffix array of T, among all suffixes that start with the same character, the type S suffixes appear after the type L suffixes.*

Proof. Follows directly from Lemma 2. □

Let A be an array containing all suffixes of T, not necessarily in sorted order. Let B be an array of all suffixes of type S, sorted in lexicographic order. Using B, we can compute the lexicographically sorted order of all suffixes of T as follows:



T	M	I	S	S	I	S	S	I	P	P	I	$
Type		S			S			S				S
Pos	1	2	3	4	5	6	7	8	9	10	11	12

Order Of Type S suffixes | 12 | 8 | 5 | 2 |

Bucket	$	I				M	P		S			
After Step 2	12	11	8	5	2	1	9	10	3	4	6	7
Sorted Order	12	11	8	5	2	1	10	9	7	4	6	3

Fig. 2. Illustration of how to obtain the sorted order of all suffixes, from the sorted order of type S suffixes of the string MISSISSIPPI$.

(1) Bucket all suffixes of T according to their first character in array A. Each bucket consists of all suffixes that start with the same character. This step takes $\mathrm{O}(n)$ time.

(2) Scan B from right to left. For each suffix encountered in the scan, move the suffix to the current end of its bucket in A, and advance the current end by one position to the left. More specifically, the move of a suffix in array A to a new position should be taken as swapping the suffix with the suffix currently occupying the new position. After the scan of B is completed, by Corollary 3, all type S suffixes are in their correct positions in A. The time taken is $\mathrm{O}(|B|)$, which is bounded by $\mathrm{O}(n)$.

(3) Scan A from left to right. For each entry $A[i]$, if $T_{A[i]-1}$ is a type L suffix, move it to the current front of its bucket in A, and advance the front of the bucket by one. This takes $\mathrm{O}(n)$ time. At the end of this step, A contains all suffixes of T in sorted order.

In Fig. 2, the suffix pointed by the arrow is moved to the current front of its bucket when the scan reaches the suffix at the origin of the arrow. The following lemma proves the correctness of the procedure in step 3.

Lemma 4. *In step* 3, *when the scan reaches* $A[i]$, *then suffix* $T_{A[i]}$ *is already in its sorted position in* A.

Proof. By induction on i. To begin with, the smallest suffix in T must be of type S and hence in its correct position $A[1]$. By inductive hypothesis, assume that $A[1], A[2], \ldots,$ $A[i]$ are the first i suffixes in sorted order. We now show that when the scan reaches $A[i+1]$, then the suffix in it, i.e., $T_{A[i+1]}$ is already in its sorted position. Suppose not. Then there exists a suffix referenced by $A[k]$ $(k > i+1)$ that should be in $A[i+1]$ in sorted order, i.e., $T_{A[k]} \prec T_{A[i+1]}$. As all type S suffixes are already in correct positions, both $T_{A[k]}$ and $T_{A[i+1]}$ must be of type L. Because A is bucketed by the first character of the suffixes prior to step 3, and a suffix is never moved out of its bucket, $T_{A[k]}$ and $T_{A[i+1]}$ must begin with the same character, say c. Let $T_{A[i+1]} = c\alpha$ and $T_{A[k]} = c\beta$. Since $T_{A[k]}$ is type L, $\beta \prec T_{A[k]}$. From $T_{A[k]} \prec T_{A[i+1]}$, $\beta \prec \alpha$. Since $\beta \prec T_{A[k]}$, and the correct sorted position of $T_{A[k]}$ is $A[i+1]$, β must occur in $A[1] \ldots A[i]$. Because $\beta \prec \alpha$, $T_{A[k]}$ should have been moved to the current front of its bucket before $T_{A[i+1]}$. Thus, $T_{A[k]}$ can not occur to the right of $T_{A[i+1]}$, a contradiction. □

So far, we showed that if all type S suffixes are sorted, then the sorted position of all suffixes of T can be determined in $\mathrm{O}(n)$ time. In a similar manner, the sorted position of all suffixes of T can also be determined from the sorted order of all suffixes of type L. To do this, we bucket all suffixes of T based on their first characters into an array A. We then scan the sorted order of type L suffixes from left to right and determine their correct positions in A by moving them to the current front of their respective buckets. We then scan A from right to left and when $A[i]$ is encountered, if $T_{A[i]-1}$ is of type S, it will be moved to the current end of its bucket.

Once the suffixes of T are classified into type S and type L, we choose to sort those type of suffixes which are fewer in number. Without loss of generality, assume that type S suffixes are fewer. We now show how to recursively sort these suffixes.

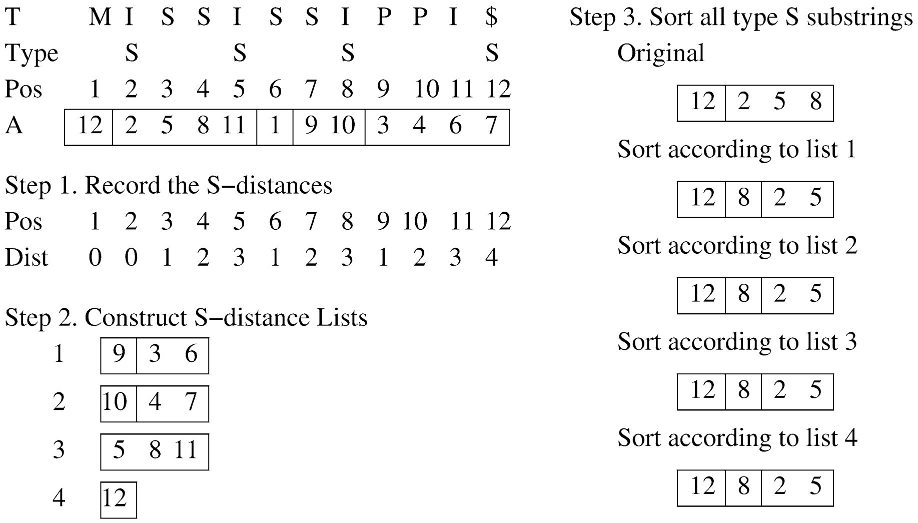
Define position i of T to be a type S position if the suffix T_i is of type S, and similarly to be a type L position if the suffix T_i is of type L. The substring $t_i \ldots t_j$ is called a type S substring if both i and j are type S positions, and every position between i and j is a type L position.

Our goal is to sort all the type S suffixes in T. To do this we first sort all the type S substrings. The sorting generates buckets where all the substrings in a bucket are identical. The buckets are numbered using consecutive integers starting from 1. We then generate a new string T' as follows: Scan T from left to right and for each type S position in T, write the bucket number of the type S substring starting from that position. This string of bucket numbers forms T'. Observe that each type S suffix in T naturally corresponds to a suffix in the new string T'. In Lemma 5, we prove that sorting all type S suffixes of T is equivalent to sorting all suffixes of T'. We sort T' recursively.

We first show how to sort all the type S substrings in $\mathrm{O}(n)$ time. Consider the array A, consisting of all suffixes of T bucketed according to their first characters. For each suffix T_i, define its *S-distance* to be the distance from its starting position i to the nearest type S position to its left (excluding position i). If no type S position exists to the left, the *S-distance* is defined to be 0. Thus, for each suffix starting on or before the first type S position in T, its *S-distance* is 0. The type S substrings are sorted as follows (illustrated in Fig. 3):

(1) For each suffix in A, determine its *S-distance*. This is done by scanning T from left to right, keeping track of the distance from the current position to the nearest type S position to the left. While at position i, the *S-distance* of T_i is known and this distance is recorded in array *Dist*. The *S-distance* of T_i is stored in *Dist*$[i]$. Hence, the *S-distances* for all suffixes can be recorded in linear time.

(2) Let m be the largest *S-distance*. Create m lists such that list j $(1 \leqslant j \leqslant m)$ contains all the suffixes with an *S-distance* of j, listed in the order in which they appear in array A.



T	M	I	S	S	I	S	S	I	P	P	I	$
Type		S			S			S				S
Pos	1	2	3	4	5	6	7	8	9	10	11	12
A	12	2	5	8	11	1	9	10	3	4	6	7

A: | 12 | 2 5 8 11 | 1 | 9 10 | 3 4 6 7 |

Step 1. Record the S−distances

Pos	1	2	3	4	5	6	7	8	9	10	11	12
Dist	0	0	1	2	3	1	2	3	1	2	3	4

Step 2. Construct S−distance Lists

1 | 9 | 3 6 |

2 | 10 | 4 7 |

3 | 5 8 11 |

4 | 12 |

Step 3. Sort all type S substrings

Original

| 12 | 2 5 8 |

Sort according to list 1

| 12 | 8 | 2 5 |

Sort according to list 2

| 12 | 8 | 2 5 |

Sort according to list 3

| 12 | 8 | 2 5 |

Sort according to list 4

| 12 | 8 | 2 5 |

Fig. 3. Illustration of the sorting of type S substrings of the string MISSISSIPPI$.

This can be done by scanning A from left to right in linear time, referring to $Dist[A[i]]$ to put $T_{A[i]}$ in the correct list.

(3) We now sort the type S substrings using the lists created above. The sorting is done by repeated bucketing using one character at a time. To begin with, the bucketing based on first character is determined by the order in which type S suffixes appear in array A. Suppose the type S substrings are bucketed according to their first $j-1$ characters. To extend this to j characters, we scan list j. For each suffix T_i encountered in the scan of a bucket of list j, move the type S substring starting at t_{i-j} to the current front of its bucket, then move the current front to the right by one. After a bucket of list j is scanned, new bucket boundaries need to be drawn between all the type S substrings that have been moved, and the type S substrings that have not been moved. Because the total size of all the lists is $O(n)$, the sorting of type S substrings only takes $O(n)$ time.

The sorting of type S substrings using the above algorithm respects lexicographic ordering of type S substrings, with the following important exception: If a type S substring is the prefix of another type S substring, the bucket number assigned to the shorter substring will be larger than the bucket number assigned to the larger substring. This anomaly is designed on purpose, and is exploited later in Lemma 5.

As mentioned before, we now construct a new string T' corresponding to all type S substrings in T. Each type S substring is replaced by its bucket number and T' is the sequence of bucket numbers in the order in which the type S substrings appear in T. Because every type S suffix in T starts with a type S substring, there is a natural one-to-one correspondence between type S suffixes of T and all suffixes of T'. Let T_i be a suffix of T and $T'_{i'}$ be its corresponding suffix in T'. Note that $T'_{i'}$ can be obtained from T_i by replacing every type S substring in T_i with its corresponding bucket number. Similarly, T_i can be obtained from

$T'_{i'}$ by replacing each bucket number with the corresponding substring and removing the duplicate instance of the common character shared by two consecutive type S substrings. This is because the last character of a type S substring is also the first character of the next type S substring along T.

Lemma 5. *Let T_i and T_j be two suffixes of T and let $T'_{i'}$ and $T'_{j'}$ be the corresponding suffixes of T'. Then, $T_i \prec T_j \Leftrightarrow T'_{i'} \prec T'_{j'}$.*

Proof. We first show that $T'_{i'} \prec T'_{j'} \Rightarrow T_i \prec T_j$. The prefixes of T_i and T_j corresponding to the longest common prefix of $T'_{i'}$ and $T'_{j'}$ must be identical. This is because if two bucket numbers are the same, then the corresponding substrings must be the same. Consider the leftmost position in which $T'_{i'}$ and $T'_{j'}$ differ. Such a position exists and the characters (bucket numbers) of $T'_{i'}$ and $T'_{j'}$ in that position determine which of $T'_{i'}$ and $T'_{j'}$ is lexicographically smaller. Let k be the bucket number in $T'_{i'}$ and l be the bucket number in $T'_{j'}$ at that position. Since $T'_{i'} \prec T'_{j'}$, it is clear that $k < l$. Let α be the substring corresponding to k and β be the substring corresponding to l. Note that α and β can be of different lengths, but α cannot be a proper prefix of β. This is because the bucket number corresponding to the prefix must be larger, but we know that $k < l$.

Case 1: β is not a prefix of α. In this case, $k < l \Rightarrow \alpha \prec \beta$, which implies $T_i \prec T_j$.

Case 2: β is a proper prefix of α. Let the last character of β be c. The corresponding position in T is a type S position. The position of the corresponding c in α must be a type L position.

Since the two suffixes that begin at these positions start with the same character, by Corollary 3, the type L suffix must be lexicographically smaller then the type S suffix. Thus, $T_i \prec T_j$.

From the one-to-one correspondence between the suffixes of T' and the type S suffixes of T, it also follows that $T_i \prec T_j \Rightarrow T'_{i'} \prec T'_{j'}$. □

Corollary 6. *The sorted order of the suffixes of T' determines the sorted order of the type S suffixes of T.*

Proof. Let $T'_{i'_1}, T'_{i'_2}, T'_{i'_3}, \ldots$ be the sorted order of suffixes of T'. Let $T_{i_1}, T_{i_2}, T_{i_3}, \ldots$ be the sequence obtained by replacing each suffix $T'_{i'_k}$ with the corresponding type S suffix T_{i_k}. Then, $T_{i_1}, T_{i_2}, T_{i_3}, \ldots$ is the sorted order of type S suffixes of T. The proof follows directly from Lemma 5. □

Hence, the problem of sorting the type S suffixes of T reduces to the problem of sorting all suffixes of T'. Note that the characters of T' are consecutive integers starting from 1. Hence our suffix sorting algorithm can be recursively applied to T'.

If the string T has fewer type L suffixes than type S suffixes, the type L suffixes are sorted using a similar procedure—call the substring $t_i, \ldots, t_j$ a type L substring if both

i and j are type L positions, and every position between i and j is a type S position. Now sort all the type L substrings and construct the corresponding string T' obtained by replacing each type L substring with its bucket number. Sorting T' gives the sorted order of type L suffixes.

Thus, the problem of sorting the suffixes of a string T of length n can be reduced to the problem of sorting the suffixes of a string T' of size at most $\lceil \frac{n}{2} \rceil$, and $\mathrm{O}(n)$ additional work. This leads to the recurrence

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathrm{O}(n).$$

Theorem 7. *The suffixes of a string of length n can be lexicographically sorted in $\mathrm{O}(n)$ time and space.*

3. Space requirement

We now consider the space requirement of our suffix array construction algorithm. The algorithm can be decomposed into the following parts:

(1) Classifying the types of all suffixes.
(2) Sorting all suffixes according to their first character.
(3) Constructing m lists according to the *S-distance* of each suffix, and the sorted order of their first character.
(4) Sorting all type S substrings by repeated bucketing using the m lists.
(5) Constructing a new string T' according to the bucket numbers of type S substrings.
(6) Recursively applying our algorithm, and obtaining the sorted order of type S suffixes.
(7) Constructing the suffix array from the sorted order of all type S suffixes.

Except for step 4, the calculation of space requirement for each of the steps listed above is straightforward, and offers little room for improvement by using a more efficient implementation. Therefore we limit the focus of our analysis to efficient implementation of step 4.

As mentioned previously, the sorting of all type S substrings is done by repeated bucketing using one character at a time. Suppose the type S substrings are bucketed according to their first $j-1$ characters. To extend this to j characters, we scan list j. For each suffix T_i encountered, move the type S substring starting at t_{i-j} to the current front of its bucket and advance the current front by one.

In Manber and Myers' algorithm [16], the suffixes are also moved to the front of their respective buckets in each iteration. However, their space-efficient scheme does not apply to our algorithm because every suffix will be moved at most once in each iteration of their algorithm. On the other hand, a type S substring may be moved multiple times in each recursion step of our algorithm. In order to achieve $\mathrm{O}(n)$ runtime, we must be able to locate the current front of the bucket containing a given type S substring in constant time.

Let array C be an array containing all type S substrings, bucketed according to their first characters. A type S substring is denoted by its starting position in T. Array C can be generated by copying from array A computed in step 2. Let R be an array of size n, such that if $C[i] = j$, then $R[j] = k$ where k is the position of the end of the bucket containing j. R can be constructed by a right to left scan of C. Let *lptr* be an array of the same size as C, such that if i is the last position of a bucket in C, then $\mathit{lptr}[i] = j$ where j is the current front of that bucket. For all other positions k, $\mathit{lptr}[k] = -1$.

Each of the m lists is itself bucketed according to the first character of the suffixes. As previously mentioned, for each suffix T_i encountered in the scan of a bucket in list j, type S substring starting at t_{i-j} is moved to the current front of its bucket. The bucket containing t_{i-j} can be found by referring to $R[i - j]$, and the current front of its bucket can then be found by referring to $\mathit{lptr}[R[i - j]]$. The current front is advanced by incrementing $\mathit{lptr}[R[i - j]]$. Note that the effect of moving a type S substring starting at t_{i-j} is achieved by adjusting the values of $R[i - j]$ and $\mathit{lptr}[R[i - j]]$ instead of actually moving it in C.

After scanning an entire bucket of list j, all the elements of C that have been moved should be in a new bucket in front of their old bucket. To accomplish this, we note that the *lptr* at the end of each old bucket in C is pointing to the current front of the old bucket, which is immediately next to the last element of the new bucket. Thus the bucket of list j is scanned again. For suffix T_i encountered in the scan, type S substring starting at t_{i-j} is moved into the new bucket by first setting $R[i - j] = \mathit{lptr}[R[i - j]] - 1$, then we set $\mathit{lptr}[R[i - j]] = R[i - j]$ if $\mathit{lptr}[R[i - j]] = -1$ or decrement $\mathit{lptr}[R[i - j]]$ by one otherwise.

It is easy to see that all the values of R and *lptr* are set correctly at the end of the second scan. The amount of work done in this step is proportional to the size of all the m lists, which is $\mathrm{O}(n)$. Two integer arrays of size n and two integer arrays of size at most $\lceil \frac{n}{2} \rceil$ are used. Assuming each integer representation takes 4 bytes of space, the total space used in this step is $12n$ bytes. Note that it is not necessary to actually move the type S substrings in C as the final positions of type S substrings after sorting can be deduced from R. In fact, we construct T' directly using R. Array C is only needed to initialize R and *lptr*. We can initialize R from C, then discard C, and initialize *lptr* from R, thus further reducing the space usage to $10n$ bytes. However, this reduction is not necessary as construction of m lists in step 3 requires $12n$ bytes, making it the most space-expensive step of the algorithm.

To construct the m lists, we use a stable counting sort on A using the *S-distance* as the key. The total amount of space used in this part of the algorithm is 3 integer arrays—one for A, one for the m lists, and a temporary array. The fact that we discard almost all arrays before the next recursion step of our algorithm except the strings, and that each subsequent step uses only half the space used in the previous step, makes the construction of the m lists in the first iteration the most space consuming stage of our algorithm.

It is possible to derive an implementation of our algorithm that uses only three integer arrays of size n and three boolean arrays[1] (two of size n and one of size $\lceil \frac{n}{2} \rceil$). The space requirement of our algorithm is $12n$ bytes plus $\frac{3}{2}n$ bits. This compares favorably with the best space-efficient implementations of linear time suffix tree construction algorithms,

[1] The boolean arrays are used to mark bucket boundaries, and to denote the type of each suffix.

which still require $20n$ bytes [2]. Hence, direct linear time construction of suffix arrays using our algorithm is more space-efficient.

In case the alphabet size is constant, it is possible to further reduce the space requirement by eliminating the calculation of the m lists in the first iteration. This is possible because the type S substrings can be sorted character by character as individual strings in $\mathrm{O}(n)$ time if the alphabet size is constant. This reduces the space required to only $8n$ bytes plus $0.5n$ bits for the first iteration. Note that this idea cannot be used in subsequent iterations because the string T' to be worked on in the subsequent iterations will still be based on integer alphabet. So we resort to the traditional implementation for this and all subsequent iterations. As a result, the space requirement for the complete execution of the algorithm can be reduced to $8n$ bytes plus $1.25n$ bits. This is competitive with Manber and Myers' $\mathrm{O}(n \log n)$ time algorithm for suffix array construction [16], which requires only $8n$ bytes. In many practical applications, the size of the alphabet is a small constant. For instance, computational biology applications deal with DNA and protein sequences, which have alphabet sizes of 4 and 20, respectively.

4. Reducing the size of T'

In this section, we present an implementation strategy to further reduce the size of T'. Consider the result of sorting all type S substrings of T. Note that a type S substring is a prefix of the corresponding type S suffix. Thus, sorting type S substrings is equivalent to bucketing type S suffixes based on their respective type S substring prefixes. The bucketing conforms to the lexicographic ordering of type S suffixes. The purpose of forming T' and sorting its suffixes is to determine the sorted order of type S suffixes that fall into the same bucket. If a bucket contains only one type S substring, the position of the corresponding type S suffix in the sorted order is already known.

Let $T' = b_1 b_2 \dots b_m$. Consider a maximal substring $b_i \dots b_j$ $(j < m)$ such that each b_k $(i \leqslant k \leqslant j)$ contains only one type S substring. We can shorten T' by replacing each such maximal substring $b_i \dots b_j$ with its first character b_i. Since $j < m$ the bucket number corresponding to '\$' is never dropped, and this is needed for subsequent iterations. It is easy to directly compute the shortened version of T', instead of first computing T' and then shortening it. Shortening T' will have the effect of eliminating some of the suffixes of T', and also modifying each suffix that contains a substring that is shortened. We already noted that the final positions of the eliminated suffixes are already known. It remains to be shown that the sorted order of other suffixes are not affected by the shortening.

Consider any two suffixes $T'_k = b_k \dots b_m$ and $T'_l = b_l \dots b_m$, such that at least one of the suffixes contains a substring that is shortened. Let $j \geqslant 0$ be the smallest integer such that either b_{k+j} or b_{l+j} (or both) is the beginning of a shortened substring. The first character of a shortened substring corresponds to a bucket containing only one type S substring. Hence, the bucket number occurs nowhere else in T'. Therefore $b_{k+j} \neq b_{l+j}$, and the sorted order of $b_k \dots b_m$ and $b_l \dots b_m$ is determined by the sorted order of $b_k \dots b_{k+j}$ and $b_l \dots b_{l+j}$. In other words, the comparison of any two suffixes never extends beyond the first character of a shortened substring.

5. Related work

In this section we compare our algorithm with some of the other suffix array construction algorithms. Since the introduction of suffix array by Manber and Myers [16], several algorithms for suffix array construction have been developed. Some of these algorithms are aimed at reducing the space usage, while others are aimed at reducing the runtime. Table 1 contains the names and descriptions of the algorithms used in our comparison. Table 2 lists the space requirement, time complexity, and restrictions on alphabet size. It is immediately clear that space is sacrificed for better time complexity. We also note that for the case of constant size alphabet, our algorithm has a better runtime, while maintaining similar memory usage compared to algorithms by Manber and Myers [16], and Larsson and Sadakane [15]. Kurtz [14] has developed a space-efficient way of constructing and storing suffix trees. Although on average it only uses 10.1 bytes per input character, it has a worst case of 20 bytes per input character. Indeed, some of the techniques used in his implementation can be applied to our algorithm as well, and this will lead to further reduction of space in practice.

Note that we have not included a comparison of the space required by other linear time algorithms [11,13] in Table 2. To achieve optimal space usage for our algorithm, it is very important that implementation techniques outlined in Section 3 are properly utilized. Due to the recent discovery of these results, a thorough space analysis of the other two linear time algorithms is not yet available in the published literature. Our analysis indicates that our space requirement would be lower than the space required by Park et al.'s algorithm [13] and is the same as the space required for Kärkkäinen and Sanders' algorithm [11]. All three algorithms depend on recursively reducing the problem size—to half the original size for Park et al.'s algorithm, to two thirds of the original size for Kärkkäinen and Sanders'

Table 1
Algorithms and their descriptions

Name	Description
Manber and Myers	Manber and Myers' original algorithm [16].
Sadakane	Larsson and Sadakane's algorithm [15].
Two-stage suffix sort	Itoh and Tanaka's two-stage suffix sorting algorithm [10].
Multikey Quicksort	Sorting suffixes as individual strings using ternary Quicksort [4].
Our algorithm	The algorithm presented in this paper.

Table 2
Comparison of different algorithms

Algorithm	Space (bytes)	Time complexity	Alphabet size
Manber and Myers	$8n$	$O(n \log n)$	Arbitrary
Sadakane	$8n$	$O(n \log n)$	Arbitrary
Two-stage suffix sort	$4n$	$O(n^2)$	$1 \ldots n$
Two-stage suffix sort	$4n$	$O(n^2 \log n)$	Arbitrary
Multikey Quicksort	$4n$	$O(n^2 \log n)$	Arbitrary
Our algorithm	$12n$	$O(n)$	$1 \ldots n$
Our algorithm	$8n$	$O(n)$	Constant

algorithm, and to at most half the original size for our algorithm. The worst-case of reducing the problem size to only half will be realized when the number of type S and type L suffixes are the same. This, coupled with the reduction technique presented in Section 4, will significantly reduce the number of levels of recursion required. For example, in an experiment to build a suffix array on the genome of *E. Coli* which is approximately 4 million base pairs (characters) long, we found the number of levels of recursion required is only 8 compared to the 22 that would be required by recursively halving.

6. Conclusions

In this paper we present a linear time algorithm for sorting the suffixes of a string over an integer alphabet, or equivalently, for constructing the suffix array of the string. Our algorithm can also be used to construct suffix trees in linear time. Apart from being one of the first direct algorithms for constructing suffix arrays in linear time, the simplicity and space advantages of our algorithm are likely to make it useful in suffix tree construction as well. An important feature of our algorithm is that it breaks the string into substrings of variable sizes, while other linear time algorithms break the string into substrings of a fixed size. A C++ implementation of our suffix array construction algorithm can be obtained by contacting the first author.

References

[1] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, The enhanced suffix array and its applications to genome analysis, in: 2nd Workshop on Algorithms in Bioinformatics, 2002, pp. 449–463.

[2] M.I. Abouelhoda, E. Ohlebusch, S. Kurtz, Optimal exact string matching based on suffix arrays, in: International Symposium on String Processing and Information Retrieval, IEEE, 2002, pp. 31–43.

[3] M.I. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, J. Discrete Algorithms 2 (1) (2004) 53–86.

[4] J.L. Bentley, R. Sedgewick, Fast algorithms for sorting and searching strings, in: Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 1997, pp. 360–369.

[5] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg, Alignment of whole genomes, Nucleic Acids Research 27 (1999) 2369–2376.

[6] M. Farach, S. Muthukrishnan, Optimal logarithmic time randomized suffix tree construction, in: Proc. of 23rd International Colloquium on Automata Languages and Programming, 1996.

[7] G.H. Gonnet, R.A. Baeza-Yates, T. Snider, New indices for text: PAT trees and PAT arrays, in: Information Retrieval: Data Structures and Algorithms, 1992, pp. 66–82.

[8] D. Gusfield, Algorithms on Strings Trees and Sequences, Cambridge University Press, New York, 1997.

[9] W.K. Hon, K. Sadakane, W.K. Sung, Breaking a time-and-space barrier in constructing full-text indices, in: 44th Annual IEEE Symposium on Foundations of Computer Science, 2003, pp. 251–260.

[10] H. Itoh, H. Tanaka, An efficient method for in memory construction of suffix array, in: International Symposium on String Processing and Information Retrieval, IEEE, 1999, pp. 81–88.

[11] J. Kärkkäinen, P. Sanders, Simpler linear work suffix array construction, in: International Colloquium on Automata, Languages and Programming, 2003, pp. 943–955.

[12] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: 12th Annual Symposium, Combinatorial Pattern Matching, 2001, pp. 181–192.

[13] D.K. Kim, J.S. Sim, H. Park, K. Park, Linear-time construction of suffix arrays, in: 14th Annual Symposium, Combinatorial Pattern Matching, 2003, pp. 186–199.

[14] S. Kurtz, Reducing the space requirement of suffix trees, Software—Practice and Experience 29 (13) (1999) 1149–1171.

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

[16] U. Manber, G. Myers, Suffix arrays: a new method for on-line search, SIAM J. Comput. 22 (1993) 935–948.

[17] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (1976) 262–272.

[18] E. Ukkonen, On-line construction of suffix-trees, Algorithmica 14 (1995) 249–260.

[19] P. Weiner, Linear pattern matching algorithms, in: 14th Symposium on Switching and Automata Theory, IEEE, 1973, pp. 1–11.