Space Efficient Linear Time Construction of Suffix Arrays [⋆]

Pang Ko and Srinivas Aluru
Dept. of Electrical and Computer Engineering

[1] Laurence H. Baker Center for Bioinformatics and Biological Statistics
[2] Iowa State University
Ames, IA 50011
`{kopang, aluru}@iastate.edu`

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 improves upon the best known direct algorithm for suffix sorting, which takes $O(n \log n)$ time. We also show how to construct suffix trees in linear time from our suffix sorting result. Apart from being simple and applicable for alphabets not necessarily of fixed size, this method of constructing suffix trees is more space efficient.

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 $O\left(n \log |\Sigma|\right)$ time and $O(n)$ space, or in $O(n)$ time and $O(n|\Sigma|)$ space [McC76, Ukk95, Wei73]. These algorithms are suitable for small, fixed size alphabets. Subsequently, Farach [FM96] 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 [Gus97].

The suffix array of a string is the lexicographically sorted list of all its suffixes. In 1993, Manber and Myers introduced the suffix array data structure [MM93] as a space-efficient substitute for suffix trees. As a lexicographic-order 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

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

presented direct construction algorithms that run in $O(n \log n)$ worst-case time and $O(n)$ expected time, respectively. 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 [MM93]. More recently, Kasai *et al.* [KLA$^+$01] 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' pattern matching algorithm takes $O(|P| + \log n)$ time, without any restriction on Σ. Recently, Abouelhoda *et al.* [AOK02] 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 [AKO02, AOK02]. 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 [DKF$^+$99] 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 [IT99, LS99]. We close this gap by presenting a direct linear time algorithm for constructing suffix arrays over integer alphabets. Contemporaneous to our result, Kärkkänen *et al.* [KS03] and Kim *et al.* [KSPP03] also discovered suffix array construction algorithms with linear time complexity.

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 [FM96]. Because the *lcp* array can be inferred from the suffix array in linear time [KLA$^+$01], 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 [McC76, Ukk95, Wei73]. 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, this is the first suffix tree construction algorithm that achieves linear run-time without exploiting the use of suffix links.

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.

The remainder of this paper is organized as follows: In Section 2, we present our linear time suffix sorting algorithm. An implementation strategy that further improves the run-time in practice is presented in Section 3. Section 4 concludes the paper.

2 Suffix Sorting Algorithm

Consider a string $T = t_1 t_2 \dots t_n$ over the alphabet $\Sigma = \{1 \dots n\}$. 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 $T_i = t_i t_{i+1} \dots t_n$ 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.

We classify the suffixes into two types: 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.

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

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

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 $T_i, T_{i+1}, \dots, T_{j-1}$ are of type S.
if $t_j < t_i$, then suffixes $T_i, T_{i+1}, \dots, T_{j-1}$ are of type L.

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

The type of each suffix of the string MISSISSIPPI$ is shown in Figure 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. We prove this by contradiction. Suppose a type S suffix T_i and a type L suffix T_j be two suffixes that start with the same character c, such that $T_i \prec T_j$. We can write $T_i = c\alpha c_1\beta$ and $T_j = c\alpha c_2\gamma$, where $c_1 \neq c_2$ and α, β, and γ are (possibly empty) strings.

Case 1: α contains a character other than c. Let c_3 be the leftmost character in α that is different from c. Because T_i is a type S suffix, it follows that $c_3 > c$. Similarly, for T_j to be a type L suffix, $c_3 < c$, a contradiction.

Case 2: α does not contain any character other than c. In this case, we have the following:

$$T_i \text{ of type } S \Rightarrow c_1 \geq c$$
$$T_j \text{ of type } L \Rightarrow c_2 \leq c$$
$$c_2 \leq c \text{ and } c \leq c_1 \Rightarrow c_2 \leq c_1$$

But $T_i \prec T_j \Rightarrow c_1 < c_2$, a contradiction. □

Corollary 1. *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. Create an array *Rev* such that $R[i] = k$ if $A[k] = i$, i.e., $R[i]$ indicates the position where suffix T_i is stored in A. We need to keep *Rev* up-to-date, thus any change made to A is also reflected in *Rev*. 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:

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 $O(n)$ 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 that position. After the scan of B is completed, by Corollary 1, all type S suffixes are in their correct positions in A. The time taken is $O(|B|)$, which is bounded by $O(n)$.
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 $O(n)$ time. At the end of this step, A contains all suffixes of T in sorted order.

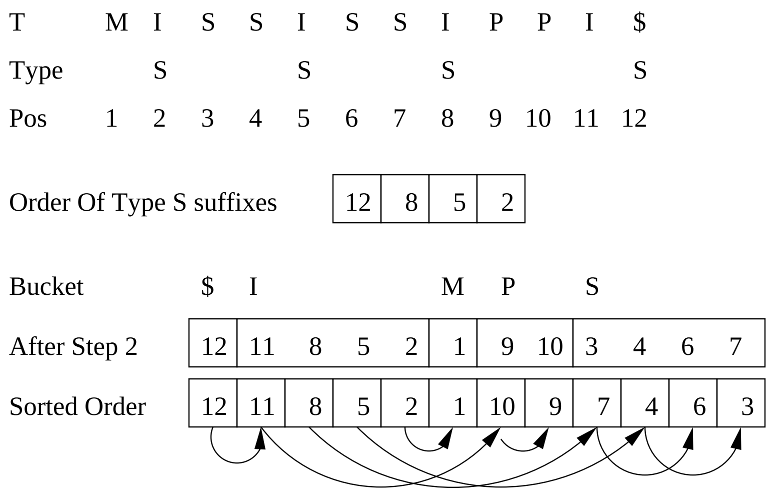
T M I S S I S S I P P I $
Type S S S S
Pos 1 2 3 4 5 6 7 8 9 10 11 12
Order Of Type S suffixes 12 8 5 2
Bucket $ I M P S
After Step 2 12 11 8 5 2 1 9 10 3 4 6 7
Sorted Order 12 11 8 5 2 1 10 9 7 4 6 3

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\$.

In Figure 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 3. *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. $\square$

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 $O(n)$ 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 $t_i \dots t_j$ 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 4, 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 $O(n)$ 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 Figure 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 $(1 \leq j \leq m)$ contains all the suffixes with an *S-distance* of j, listed in the order in which they appear in array A. 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, move the type S substring starting

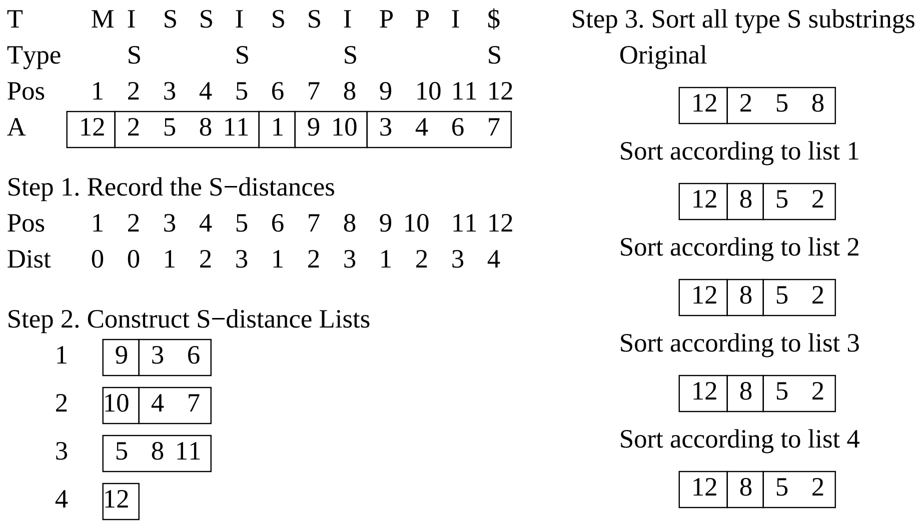
T M I S S I S S I S S I P P I $
Type S S S S
Pos 1 2 3 4 5 6 7 8 9 10 11 12
A 12 2 5 8 11 1 9 10 3 4 6 7
Step 1. Record the S−distances
Pos 1 2 3 4 5 6 7 8 9 10 11 12
Dist 0 0 1 2 3 1 2 3 1 2 3 4
Step 2. Construct S−distance Lists
1 9 3 6
2 10 4 7
3 5 8 11
4 12
Step 3. Sort all type S substrings
Original
12 2 5 8
Sort according to list 1
12 8 5 2
Sort according to list 2
12 8 5 2
Sort according to list 3
12 8 5 2
Sort according to list 4
12 8 5 2

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

at t_{i-j} to the current front of its bucket. 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 4.

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 4. *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 1, 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 2. *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 4. □

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 $O(n)$ additional work. This leads to the recurrence

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

Theorem 1. *The suffixes of a string of length n can be lexicographically sorted in $O(n)$ time and space.*

We now consider the space required for the execution of our suffix array construction algorithm. By applying several implementation strategies, some of which are similar to those presented by Manber and Myers [MM93], it is possible to derive an implementation of our algorithm that uses only 3 integer arrays of size n and 3 boolean arrays (2 of size n and one of size $\lceil \frac{n}{2} \rceil$). Assuming each integer representation takes 4 bytes of space, the space requirement of our algorithm is $12n$ bytes plus $2.5n$ bits. This compares favorably with the best space-efficient implementations of linear time suffix tree construction algorithms, which still require $20n$ bytes [AOK02]. 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. Note that the maximum space utilization by our algorithm occurs in the first iteration. As the size of the string reduces at least by half in each iteration, so does the space required by the algorithm. We take advantage of this fact by designing a more space-efficient implementation for the first iteration, which is applicable only for constant sized alphabets. The main underlying idea is to eliminate the construction of the lists used in sorting type S substrings. 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 second iteration 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' $O(n \log n)$ time algorithm for suffix array construction [MM93], 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.

3 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 $T' = b_1 b_2 \ldots b_m$. Consider a maximal substring $b_i \ldots b_j$ ($j < m$) such that each b_k ($i \leq k \leq j$) contains only one type S substring. We can shorten T' by replacing each such maximal substring $b_i \ldots b_j$ 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 is 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 $j \geq 0$ 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.

4 Conclusions

In this paper we present a linear time algorithm for sorting the suffixes of a string over 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 the first direct algorithm 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.

References

[AKO02] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *2nd Workshop on Algorithms in Bioinformatics*, pages 449–63, 2002.

[AOK02] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *International Symposium on String Processing and Information Retrieval*, pages 31–43. IEEE, 2002.

[DKF$^+$99] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27:2369–76, 1999.

[FM96] M. Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In *Proc. of 23rd International Colloquium on Automata Languages and Programming*, 1996.

[Gus97] D. Gusfield. *Algorithms on Strings Trees and Sequences.* Cambridge University Press, New York, New York, 1997.

[IT99] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix array. In *International Symposium on String Processing and Information Retrieval*, pages 81–88. IEEE, 1999.

[KLA$^+$01] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *12th Annual Symposium, Combinatorial Pattern Matching*, pages 181–92, 2001.

[KS03] J. Kärkkänen and P. Sanders. Simpler linear work suffix array construction. In *International Colloquium on Automata, Languages and Programming*, page to appear, 2003.

[KSPP03] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *14th Annual Symposium, Combinatorial Pattern Matching*, 2003.

[LS99] N. J. Larsson and 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.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–72, 1976.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line search. *SIAM Journal on Computing*, 22:935–48, 1993.

[Ukk95] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–60, 1995.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *14th Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.