Sādhanā (2018) 43:133
https://doi.org/10.1007/s12046-018-0832-z

Extended suffix array construction using Lyndon factors

SUNITA[1,*] and DEEPAK GARG[2]

[1]Computer Science and Engineering, Thapar University, Patiala 147004, India
[2]Computer Science Engineering Department, Bennett University, Greater Noida 201310, India
e-mail: sunita.tu@gmail.com; deepak.garg@bennett.edu.in

MS received 24 May 2017; revised 31 October 2017; accepted 4 January 2018; published online 5 July 2018

Abstract. In this paper, we are extending the novel approach of incremental suffix array construction using Lyndon factorization to the construction of extended suffix array where extended suffix array is the suffix array along with the corresponding longest common prefix (LCP) array. Main motive behind the incremental and simultaneous construction of suffix array and LCP array is that both involve in calculating the order information by considering the common prefixes of the suffixes. As local suffixes once sorted have the same sorted order when these are merged with sorted suffixes of another Lyndon factor. So, merging of Lyndon factors is simply merging of two sorted lists of suffixes of these Lyndon factors. Also, the two sorted orders coincide thus making the merging of Lyndon factors a simple merging of two sorted lists of suffixes. Incremental LCP construction simultaneously saves a lot of computation and hence time. The proposed approach has quadratic run time and the disk working space requirement is O(n). Experiments also show the performance gain of our approach in terms of time over the existing method of incremental construction.

Keywords. Computer science; data structure; suffix array; Lyndon factors; longest common prefix.

1. Introduction

Starting from the suffix tree data structure, there have been many breakthroughs in the text processing data structures like suffix array by Manber and Myers in 1990, enhanced suffix array and extended suffix array. A suffix tree is a compressed trie containing all the suffixes of the given text in the form of a tree. An array representing all the suffixes of the text in their lexicographic order is called as the suffix array (SA). Improving the space efficiency of suffix array data structure led to a new class of data structure abstract data structures like compressed suffix array [1–5] and then compressed suffix trees [6–9] with many variations in their implementations. The research in this class of data structure has been oriented towards finding an optimal space/time trade-off in their implementation either during the construction or in their use as an indexing data structure. Space/time trade-off is a major concern during the construction phase, as the size of the text which needs to be indexed is increasing day by day and far exceeding the memory capacity of today's computers. At the core of all these data structures is the suffix array which is invariably used in one form or the other in all of these variants. Also, in almost all the variants, suffix array is used with additional information like longest common prefix (LCP) array to further strengthen the index structure. The suffix array along with the LCP array is called the extended suffix array. Extended suffix array can effectively replace a suffix tree in the applications which also need some topological information of the suffix tree. The LCP array along with providing topological information of the suffix tree also helps in efficient pattern matching using suffix array [10, 11], fast disk based suffix array arrangements [12, 13] and compressed suffix trees [7].

Construction of the extended suffix array can be done in two ways: First method is to construct the suffix array and after the construction of suffix array is complete, use it to construct the LCP array using any of the existing methods. Time complexity in this case is O(n log n) for the suffix array construction and then O(n) for computing LCP from suffix array [7]. Another way is to incrementally construct the suffix array by dividing the text into blocks of suitable size and along with suffix array also compute the LCP array side by side. This is a good idea to construct the LCP as a by-product of the suffix array construction as sorting the suffixes of a text involves breaking ties after the common prefixes. Also this method is space efficient as construction is done block by block. The goal of this paper is to introduce a new strategy for the incremental construction of the extended suffix array using Lyndon Factorization that opens new scenarios for the space efficient construction of the different variants of suffix array like Extended suffix array, Compressed suffix array, etc.

*For correspondence

Chen, Fox and Lyndon [14] introduced Lyndon factorization as the factorization of a sequence of an ordered set and later Duval [15] gave a linear time algorithm to find this factorization for a word. Since then it has been used in different applications like Digital Geometry [16, 17], description of Free Lie Algebra [18], and Efficient Construction of particular de Bruijn sequence in linear time. In [19] authors gave the idea of using the Lyndon factorization of a text for the computation of the suffix array of the text. In this paper we describe and test an incremental suffix array construction approach based on Lyndon factorization of text which simultaneously constructs the LCP array. Lyndon factorization for suffix sorting helps to reduce the merging time during incremental sorting and also helps to construct the LCP array with constant additional space. The proposed approach has quadratic running time and uses O(n) bits of working space. Now we explain some of the notations and basic concepts that will help understand the paper.

1.1 *Notation and basic concepts*

We are considering a text T of size n denoted as T[0…n − 1]. All the characters of the text are from an ordered alphabet $\sum$ of size σ. The text T is appended with the character $ which is smallest of all the characters in $\sum$ and it also does not appear anywhere in the text. The suffix of a text is represented as $suf_i(T)$ which is the suffix of text T starting at position i i.e., T[i…n]. A text of size n can have n suffixes. Lexicographic order of the suffixes of text is established based on the lexicographic order of the characters of the individual suffixes as:

For any i <= n and j <= n, let $suf_i(T)$ and $suf_j(T)$ be two suffixes then $suf_i(T) < suf_j(T)$

$$\begin{aligned}&\text{if } T[i] < T[j]\\ &\text{Or if } T[i] = T[j] \text{ and } suf_{i+1}(T) < suf_{j+1}(T)\end{aligned}$$

An array representing all the suffixes of the text in their lexicographic order is called as the suffix array (SA) [20]. So, suffix at position i in the SA is i^{th} smallest suffix of text starting at position SA[i] in the text. Alternatively, $suf_{SA[0]} < suf_{SA[1]} < \cdots < suf_{SA[n]}$. As each suffix array entry is a position of the corresponding text so, can be represented in O(log n) bits and the whole suffix array takes O(n log n) bits, where n is the size of the text.

The Burrows–Wheeler Transform (BWT) [21] is a permutation of the text with some properties that make it suitable for compression as well as other text processing applications. BWT of text T of length *n* is an array bwt[0…n − 1] where bwt[i] is the last character of the i^{th} smallest cyclic shift of the text T. Sorted cyclic shifts of a text appended with a smallest character (not existing in the text) also give the sorted order of the suffixes i.e., suffix array of that text. So, the first column(F) of the sorted cyclic shifts represents the first character of the sorted suffixes and the last column(L) corresponds to BWT. There is a function LF() which gives the correspondence between the characters of the columns L and F. This LF() function helps to reconstruct the text from its BWT.

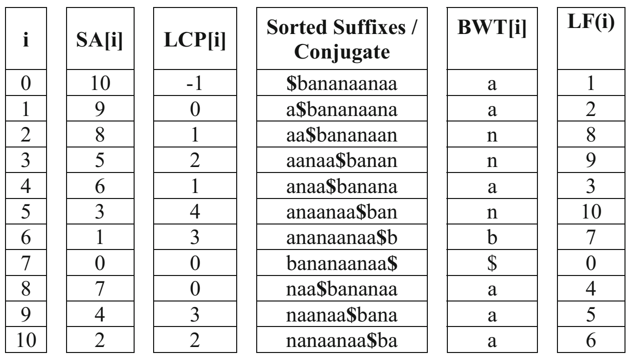
An array representing the longest common prefixes of all pairs of adjacent suffixes in SA is called as longest common prefix (LCP) array. Using LCP array as an auxiliary data structure with the SA, speeds up querying in to the text. LCP array is an array of size n whose entries are defined as:

$$\begin{aligned}LCP[i] &= -1 && \text{if } i = 0\\ &= lcp\left(suf_{SA[i-1]}, suf_{SA[i]}\right) && \text{if } 0 < i <= n\end{aligned}$$

Querying into the text using only SA takes O(m log n) time where *m* is the size of the query text. This time can be reduced to O(m + log n) by using the LCP array [20] as indicated in figure 1.

In Lyndon factorization, any string can be decomposed uniquely into a sequence of lexicographically non increasing factors i.e., $L_1L_2L_3\ldots L_k$, with the condition that $L_1 \geq L_2 \geq L_3 \geq \cdots \geq L_k$. Each factor L_i is called as a Lyndon word and has the property that each L_i is lexicographically least among its own circular shifts. For each factor L_p, let the position of the first character in that Lyndon factor is represented by F_L_p and the position of the last character of each factor L_r is given by $F\_L_{p+1} - 1$ (character preceding the first character of the next Lyndon factor). So, only the position of first character of each Lyndon factor needs to be stored. End of a Lyndon factor can be defined using the first character of the succeeding Lyndon factor. For the text *T = bananaanaa$* the Lyndon factorization is: b(L_1), an(L_2), an(L_3), aan(L_4), a(L_5), a(L_6). Each $L_i \geq L_j$ for all i < j, as b > an ≥ an > aan > a ≥ a. Also each L_i is least among its cyclic shifts.

Rest of the paper is organized as follows. Section 2 describes the related works. Section 3 describes the basic terms and concepts that will be used throughout the paper. Section 4 explains sorting the suffixes of a text using its Lyndon factorization and the proposed extensions for incremental construction of extended suffix array. Section 5 details the experimental work and results. Conclusions with



i	SA[i]	LCP[i]	Sorted Suffixes / Conjugate	BWT[i]	LF(i)
0	10	-1	$bananaanaa	a	1
1	9	0	a$bananaana	a	2
2	8	1	aa$bananaan	n	8
3	5	2	aanaa$banan	n	9
4	6	1	anaa$banana	a	3
5	3	4	anaanaa$ban	n	10
6	1	3	ananaanaa$b	b	7
7	0	0	bananaanaa$	$	0
8	7	0	naa$bananaa	a	4
9	4	3	naanaa$bana	a	5
10	2	2	nanaanaa$ba	a	6

Figure 1. Suffix array, LCP, sorted suffixes/conjugates, BWT, LF for text bananaanaa$.

applications and some possible future work are provided in section 6.

2. Related works

Suffix trees were introduced by Weiner [10] to solve the exact pattern matching problem efficiently. Later these were used to solve many other string processing problems like approximate pattern matching, longest common substring, and longest repeated substrings [11]. Suffix array data structure was introduced by Manber and Myers [20] as a space efficient alternative to suffix trees. For simple pattern searches suffix array provide same functionality as a suffix tree. For other applications suffix array can be extended with some auxiliary information like LCP array to support the functionality of suffix tree. Both these data structures have been extensively used in such diverse fields as search, indexing, plagiarism detection, information retrieval, biological sequence analysis and linguistic analysis [22]. All these fields are based on string processing in one way or the other.

In information retrieval, Suffix trees and suffix arrays are the data structures that allow searches for any text substring, not only words or phrases. So, these are suitable indexes for the applications that need to search texts where the concept of word is not well defined, as in the case of East Asian languages, biological databases or music retrieval. In data compression SA have been used to encode data with anti-dictionaries [23] and optimized for large alphabets [24]. In computational biology these data structures are used in various ways as for Exact match searches, Subsequence composition searches, Homology searches, Single sequence analysis applications, and Multiple sequence analysis applications [25]. Many bioinformatic tools using these data structures have been developed like wcd-express (an expression clustering tool which uses modified suffix array) [26], Readjoiner [27] (an efficient string graph-based sequence assembler), SHREC [28] (an algorithm that uses generalized suffix trie for correcting errors in short-read data) and essaMEM [29] (a bioinformatics tool that uses sparse enhanced suffix array for finding maximal exact matches that can be used in genome comparison and read mapping).

Almost all the applications of these data structures involve massive amount of data. So, the research in this class of data structure has been oriented towards finding an optimal space/time trade-off in their implementation either during the construction or in their use as an indexing data structure. Grossi and Vitter developed the compressed suffix array [2], a space efficient version of standard suffix array. Sadakane [3, 8] extended the work of Grossi and Vitter by developing a self- index using compressed suffix array and compressed suffix trees. Ferragina and Manzini introduced FM-index [1, 4] which is a self-indexing data structure based on Burrows Wheeler Transform which can encode the index size with respect to high-order empirical entropy.

3. Suffix sorting and Lyndon factorization

Idea of sorting the suffixes of the text using its Lyndon factors was first given by Mantaci *et al* [30] As we know that mostly all text indexes use Suffix array along with their LCP array to efficiently emulate the functionality of the index. So, we are extending the approach of suffix array construction to that of the construction of extended suffix array. Constructing the suffix array and the corresponding LCP array simultaneously will save time as well as space as incremental construction of suffix array involves identifying common prefixes and breaking ties if any which is the procedure for constructing of LCP also. Further, constructing LCP requires some additional information like inverse suffix array that is also available during incremental suffix array construction. We move on to explain our approach by first giving an overview of the existing approach. Then in the next sub-section we explain the extensions applied to the approach to use it for the construction of extended suffix array.

Lyndon factorization of a text gives a sequence of factors such that each factor in the sequence is succeeded by a factor which is smaller than it, so lexicographic ordering of suffixes and conjugates of a Lyndon factor also coincide. Based on the above discussion, we have the following lemmas [31] which define the relationship between the suffixes of a text and the corresponding conjugates and among the suffixes of adjacent Lyndon factors.

Lemma 1 *Let* $T \in \sum$ *and let* $T = L_1L_2 \ldots L_k$ *be its Lyndon factorization. For any two suffixes i and j of a Lyndon factor, if* $suf_i < suf_j$ *then the corresponding conjugates* $conj_{n-i} < conj_{n-j}$.

Lemma 2 *Lexicographic order of suffixes* $suf_i(L_m)$ *of a Lyndon factor gives the lexicographic order of the corresponding suffixes of the text i.e, for positions i and j in Lyndon factor* L_m *if* $suf_i(L_m) < suf_j(L_m)$ *then* $suf_i(T) < suf_i(T)$.

These lemmas state that lexicographic order of suffixes of a text is equivalent to the lexicographic order of corresponding suffixes in Lyndon factor. Now, based on this relationship between the conjugates for a sequence of Lyndon factors can be given as.

Theorem 1 *Let* $u = L_rL_{r+1} \ldots L_k$ *be a sequence of consecutive Lyndon factors and* L_i *and* L_j *be two Lyndon factors in u such that* $L_i < L_j$ *means* $i > j$. *Then the sorting of the conjugates of u coincides with the sorting of suffixes of u.*

Based on the above theorem, Mantaci *et al* [31] has proposed to do the suffix sorting by using the Lyndon factorization of the text. The method proposed by them is a combination of both the incremental and the divide and conquer approach for SA construction. The problem is reduced to smaller sub-problems recursively finding their SA and then induction is used on the generated solution to find the solution of the remaining problem and then finally merging the solutions of the sub-problems.

3.1 *Incremental suffix array construction*

Let $T \in \sum$ be a text with n-1 characters represented as $T[0 \ldots n-2]$ and $T[n-1] = \$$. There are many algorithms already proposed for the incremental construction of the Suffix Array (SA) for the text T. Almost all of these methods are based on dividing the text into blocks say $T_1T_2 \ldots T_k$ and then constructing the Suffix Arrays for the individual blocks say SA_1, SA_2 and merging the two giving SA for block 1 and 2 represented as SA(12), then finding SA_3 and merging it with SA(12) giving $SA(1 \ldots 3)$ and so on finally computing SA_k and merging it with $SA(1 \ldots k-1)$ giving $SA(1 \ldots k)$ which is the SA for the whole text. The complexities of these methods depend upon the complexity of the two steps mentioned above i.e. blockwise SA construction and then merging of the two partial SA's. For incremental SA construction dividing the text into blocks using the Lyndon factorization of the text greatly simplifies both these steps. Also it allows the incremental LCP construction efficiently along with the SA. First of all we will show the complexity reduction achieved by the Lyndon Factorization.

Each factor in the Lyndon factorization is succeeded by a factor that is smallest of all its preceding factors as well as its own cyclic shifts. So, each factor can be safely assumed as a text terminated by a smallest end of the text marker. And the problem of building SA of Lyndon Factor L_p is equivalent to the problem of building the suffix array of a text of size $m = |L_p|$. So, we can compute SA in linear time i.e., O(m) and O(m log m) bits of space using any of the existing linear time suffix array construction algorithm like libdivsufsort. Also, once we have sorted the suffixes of a Lyndon factor, their relative order remains same when these are merged with the sorted suffixes of adjacent Lyndon factors (as given by Theorem 1). Hence the lexicographic order between the two suffixes of T can be established by a number of symbol comparisons that is bounded by either the size of the Lyndon factor or the LCP whichever is smaller.

To incrementally construct the suffix array, the input text $T[1 \ldots n]$ is logically partitioned into k blocks based on the Lyndon Factorization [32, 33] of words. So, each block corresponds to a Lyndon Factor i.e., $T = L_1, L_2, \ldots, L_k$. The Suffix Array is computed incrementally in k passes, where each pass corresponds to a Lyndon factor. Lyndon Factors are examined from left to right so that at pass p $SA(L_1 \ldots L_p)$ is computed by first computing $SA(L_p)$ and then merging it with the $SA(L_1 \ldots L_{p-1})$ already computed. The task of computing the $SA(L_1 \ldots L_p)$ from $SA(L_1 \ldots L_{p-1})$ needs only inserting the characters from L_p in $SA(L_1 \ldots L_{p-1})$. Also, adding L_p does not modify the relative order of the suffixes already in $SA(L_{p+1} \ldots L_k)$. The algorithm can be easily adapted to external memory scenarios giving a scan-based implementation due to sequential accesses to the input text.

The crucial point of the algorithm is to compute some additional information that allows merging the two SA's efficiently. One way is to simply merge the two SA's on the basis of one to one comparison of the suffixes. But this will take much time. Another method is to find the rank of each suffix of $SA(L_p)$ in $SA(L_1 \ldots L_{p-1})$ and then simply merge the two SA's based on this rank information without any character or suffix comparison. This rank information for Lyndon factor L_p consists of an array $rank[0 \ldots |L_p|]$ which stores in rank[j] the number of suffixes of the $SA(L_1 \ldots - L_{p-1})$ which are smaller than the suffix j of Lyndon factor L_p i.e., $suf_j[L_p]$. The method given by Ferragina *et al* [34] to find the gap array is modified to compute the values for rank[i]. Rank of each suffix is calculated using the rank of its successor suffix. Following lemma gives the computation method:

Lemma 3 *Let C[a] be the number of characters smaller than a in $BWT(L_1 \ldots L_{p-1})$ and let occ(a, i) be the number of occurrences of character a in $BWT(L_1 \ldots L_{p-1})$ up to position i. Assume that $suff_{j+1}[L_p]$ be a suffix in Lyndon factor L_p whose rank in $BWT(L_1 \ldots L_{p-1})$ is r and let a be character at position j in T, so $suff_j[L_p] = a\ suff_{j+1}[L_p]$, then rank value of $suff_j[L_p]$ is given by*

$$rank[j] = C[a] + occ[a, r]$$

Based on this, Suffix Array/BWT can be constructed as:

1. *Sort the suffixes of the first Lyndon factor L_1 of the text.*
2. *Sort the suffixes of Lyndon Factor L_2.*
3. *Merge the suffixes of the two Lyndon Factors, giving the sorted suffixes of the text $T[1 \ldots F\_L_3 - 1]$.*
4. *Repeat the procedure for Lyndon Factors L_3 up to L_k.*

4. An approach for extended suffix array construction using Lyndon factors

Extended Suffix Array contains both the Suffix Array (SA) and Longest Common Prefix (LCP). So, for each Lyndon factor, we calculate both the SA as well as the LCP and merge it with the already calculated SA and LCP of succeeding Lyndon factors. The rank array is used to aid in the merging process. This array gives the rank of each new suffix of Lyndon factor L_p which needs to be merged with $SA(L_{p+1} \ldots L_k)$. The rank array gives the ranks of suffixes in their text order, but to actually use it for merging process we compute a new array called as gap() which gives the ranks of suffixes of L_p in their SA order.

$$gap\left[SA_p[i]\right] = rank[i]$$

To calculate the arrays rank[] and gap[], we need both the $BWT(L_{p+1} \ldots L_k)$ and first column(F) of the sorted suffixes. BWT() can be stored using only $nH_k(T)$ bits, where H_k is the kth order entropy. Also, column F is stored simply as the intervals of the characters instead of being stored directly using a total space of O(n). Both the values required for computation of rank(i) can be calculated in constant time [35, 36] using column F and BWT respectively.

4.1 *Computing SA for a Lyndon factor and merging with SA of another Lyndon factor*

As we know that for any Lyndon factor (L_p) the next Lyndon factor (L_{p+1}) is the smallest of all the suffixes of the factor L_p. So, we can compute $SA(L_p)$ in $O(|L_p|)$ time and $O(|L_p| \log |L_p|)$ bits of space using any of the existing linear time suffix array construction methods like Difference Cover(DC3) [37] or libdivsufsort proposed by [38] and implemented by Yuta Mori (http://code.google.com/p/libdivsufsort). Once we have calculated $SA(L_p)$, the corresponding LCP can also be computed in time linear to the size of the Lyndon factor. Now the next step is to merge both the $SA(L_p)$ and $LCP(L_p)$ with $SA(L_{p+1} \ldots L_k)$ and $LCP(L_{p+1} \ldots L_k)$ respectively.

The two SA's can simply be merged by comparing the suffixes of the SA's linearly and this merging will take time proportional to the size of the two SA's i.e., $O(|L_p| |L_{p+1} \ldots L_k|)$. But with the help of some additional information i.e., rank() we can do this merging in $O(|L_p|)$ time. To do the merging, for every suffix of L_p starting from right to left we compute its rank in the $SA(L_{p+1} \ldots L_k)$. Using array rank we compute gap() which gives the rank of suffixes of $SA(L_p)$. After calculating this, we simply merge the two SA's. Starting from gap[0], number of suffixes equal to gap[0] are shifted from $SA(L_{p+1} \ldots L_k)$ to $SA(L_p \ldots L_k)$ then $SA_p[0]$ is inserted at $SA_{p \ldots k}[gap[0] + 1]$ and so on.

4.2 *Updating the LCP*

As we are maintaining the Extended Suffix Array, we have to calculate the LCP value side by side. As is the case with suffix array, we have both $LCP(L_p)$ and $LCP(L_{p+1} \ldots .L_k)$ and we need to merge these two LCP's to get the $LCP(L_p \ldots L_k)$. The merging of LCP can be easily done side by side as we do the merging of corresponding SA's. The merging process we explained earlier can also be viewed as placing the suffixes of L_p in between the sorted suffixes of $L_{p+1} \ldots L_k$ according to their LF/rank values.

So, new $LCP(L_p \ldots L_k)$ will be the $LCP(L_{p+1} \ldots L_k)$ added with the new LCP values at the positions where new suffixes are inserted and updating the already existing values wherever required. This can easily be done in-place in the LCP array itself. Suppose a suffix from L_p has been inserted between position j and j + 1 in $SA(L_{p+1} \ldots L_k)$. Now shift all the values till position j starting from position F_L_p in the LCP array. Then we need to find LCP for the suffix at position j and the newly inserted suffix (which has been inserted at j + 1) and for the newly inserted suffix and suffix that was previously at position j + 1 i.e.,

$LCP[F\_L_p + (j + 1)] = lcp(j, j + 1)$ *where j + 1 now represents the newly inserted suffix.*
$LCP[F\_L_p + (j + 2)] = lcp(j + 1, j + 2)$ *where j + 2 represents the suffix previously existing at j + 1.*

We will show the procedure step by step using the following example. Let suffix array and corresponding LCP for a block is as under:

SA	*Suffix*	*LCP*
1	AABABABABAA$	− 1
2	ABABABAA$	1
3	ABABBBAABABABABAA$	4

Suppose suffix ABABABABAA$ is to be inserted in the suffixes of this block.

SA	*Suffix*	*LCP*
1	AABABABABAA$	− 1
2	ABABABAA$	1
3	ABABABABAA$	? need to be calculated
4	ABABBBAABABABABAA$	**4** need to be updated

Now suppose a block of say m suffixes is inserted between a pair of suffixes at position j and j + 1, even then only two LCP values need to be calculated as:

$LCP[F\_L_p + (j + 1)] = lcp(j, j + 1)$ where first suffix of the block is inserted at j + 1.
$LCP[F\_L_p + (j + m+1)] = lcp(j + m, j + m+1)$ where m is the number of suffixes in the inserted block.

SA	*Suffix*	*LCP*
1	AAABAABA$	− 1
2	ABAABA$	1
3	BAABA$	0

Now let a block of suffixes in sorted order to be inserted in the given suffixes in sorted order:

SA	*Suffix*	*LCP*
1	AABA$	− 1
2	AABAABA$	4
3	AABA$	4

New block of sorted suffixes after merging:

SA	*Suffix*	*LCP*
1	AAABAABA$	− 1
2	AABA$	? need to be calculated
3	AABAABA$	4
4	AABA$	4
5	ABAABA$	1 need to be updated
6	BAABA$	0

There is no change in the LCP value of suffixes within the block as there is no change in the sorted order of the suffixes within the block. So, we need to calculate only these two LCP values corresponding to the block of suffixes inserted. Now to calculate these two values, we use the property that lcp(i, j) of two arbitrary suffixes at position i and j in a suffix array with i < j [39]:

$$lcp(i, j) = \min((LCP[i + 1], LCP[i + 2], \ldots\ldots, LCP[j])$$

Now, suppose we have three consecutive suffixes in the suffix array at positions i, i + 1 and i + 2, and we know the LCP of suffixes at i and i + 2 i.e., lcp(i, i + 2) then lcp(i, i + 1) and lcp(i + 1, i + 2) satisfy the condition that :

$$lcp(i, i + 1) \geq lcp(i, i + 2) \, and \, lcp(i + 1, i + 2) \geq lcp(i, i + 2)$$

It is obvious from the above condition that the newly inserted suffix will have at least lcp(i, i + 2) characters in common with its adjacent suffixes. So, to find lcp(i, i + 1) we start comparing from character at position lcp(i, i + 2) + 1 in the suffixes i and i + 1. If the character compared of two suffixes is different, then LCP of two suffixes is equal to lcp(i, i + 2) otherwise LCP is computed using the LCP value of the successor suffixes. To ensure that while calculating the LCP value of two suffixes using their successor suffixes, LCP value of both the successor suffixes is available, we calculate LCP values starting from right to left in the Lyndon factor. As and when we insert a suffix in between already sorted suffixes, we know the LCP value of already sorted suffixes and using the above equation, we can find LCP of the newly inserted suffix with already existing ones. Suppose, we need to compute the LCP value of suffixes suf_i and suf_j and we already know the LCP value of suffixes suf_{i+1} and suf_{j+1} and using these values we can easily compute LCP(suf_{i+1}, suf_{j+1}) [40] as:

$$suf_i = c\, suf_{i+1}$$
$$suf_j = c\, suf_{j+1}$$
$$so, LCP\left(suf_i, suf_j\right) = LCP(suf_{i+1}, suf_{j+1}) + 1$$

Also, suffix suf_{i+1} and suf_{j+1} will be close to each other in SA($L_i \ldots L_k$). Hence, at the maximum we need character comparisons equal to the size of the Lyndon Factor being merged to compute LCP(suf_i, suf_j).

As the proposed method for updating the LCP sequentially accesses the SA as well as BWT also, it computes the LCP array sequentially. Thus, we can use this method as an external memory algorithm also in which only a portion of text and SA are in the primary memory. The proposed method uses O(n) bits of working space and time complexity is O(n^2/L) where n is the size of the text and *L* be the size of the largest Lyndon factor. Also while calculating the LCP array, partial LCP for a Lyndon factor i can be computed in O($|L_i|$) time and while merging the two LCP's we need to update at most 2$|L_i|$ LCP entries and each LCP entry can be updated in constant time (either a single character comparison or finding minimum LCP entry for the two successor suffixes).

5. Experimentation

To test how the proposed approach works in practice, we have implemented it in C language and compared it with an existing implementation of incremental construction bwt-disk by Ferragina *et al* [34]. As to the best of our knowledge no such implementation of extended suffix array exist so, we have used bwt-disk to do the incremental construction of ESA and used it to compare our approach with.

Experiments were carried out with the files of different sizes (50 MB, 100 MB and 200 MB) of the data sets mentioned in table 1 taken from Pizzza and Chilli corpus (http://pizzachili.dcc.uchile.cl/texts/). Using different data sets for the evaluation purpose helped to analyze the proposed approach for different types of data and its suitability to be used for indexing of those data.

Initially, the incremental construction of SA was done using the bwt-disk library [34] (updated to construct the SA). Afterwards, LCP array was calculated using the method of Kasai *et al* [41]. Finally, ESA was calculated for the proposed approach. We start suffix array construction starting from the last Lyndon factor moving from right to left in the text. Also along with suffix array construction we also construct the LCP array for the current Lyndon factor. While merging the partial suffix arrays we also merge the corresponding LCP arrays. In the proposed approach while merging sorted suffixes of Lyndon factor L_{k-1} with that of L_k, first suffix of L_k is the least of all the suffixes of the text. So, it is the first suffix in the suffix array and can be placed at location 0 in the resulting SA. Similarly first suffix of L_{k-1} is smallest of all the suffixes of factors $L_1$to L_{k-1}. And all the suffixes of L_k which are smaller than this suffix have their sorted order defined in the final suffix array and hence can be placed accordingly. Thus with each merging phase some suffixes are placed in their final position in the suffix

Table 1. Experimental set-up.

Data Set Name	Size (in MB)	Description
Sources	50 (File 1) 100 (File 2) 200 (File 3)	Concatenation of all the .c, .h, .C and .java files of the linux-2.6.11.6 and gcc-4.0.0 distributions.
Proteins	50 (File 1) 100 (File 2) 200 (File 3)	Sequence of newline-separated protein sequences obtained from the Swissprot database.
DNA	50 (File 1) 100 (File 2) 200 (File 3)	Sequence of newline-separated gene DNA sequences obtained from files 01hgp10 to 21hgp10, plus 0xhgp10 and 0yhgp10, from Gutenberg Project.
English	50 (File 1) 100 (File 2) 200 (File 3)	Concatenation of English text files selected from etext02 to etext05 collections of Gutenberg Project.

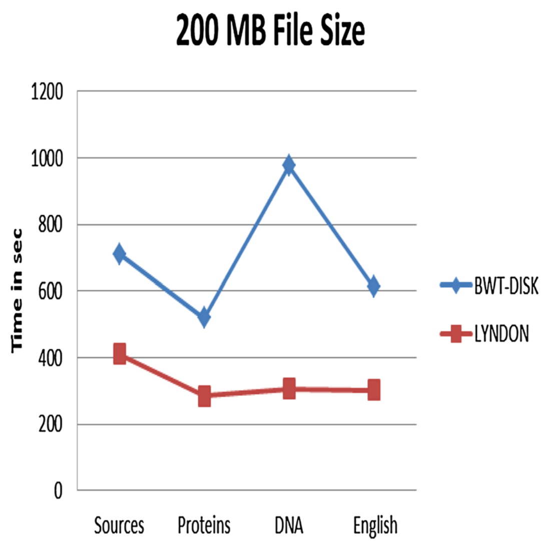


Figure 2. Construction time of BWT-DISK and LYNDON approaches for files of size 200 MB.

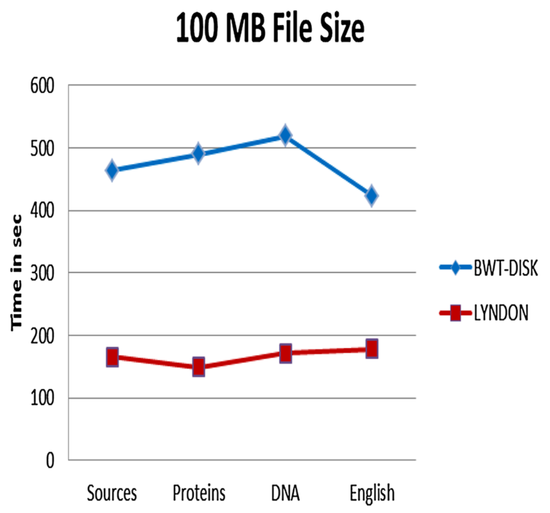


Figure 3. Construction time of BWT-DISK and LYNDON approaches for files of size 100 MB.

array and hence are excluded from rearrangements and comparisons in the successive phases.

The main point of analysis is the construction time of the method. Figures 2 and 3 indicate that our method shows improvement over the existing approach in terms of construction time for all data sets of size 200 MB and 100 MB. This performance gain is attributed to the lesser merging time of our approach as indicated in figure 2.

Figure 4 shows that the existing approach has better performance in case of files of size 50 MB. The main reason is that for small file sizes the direct construction time is so less that doing it the incremental way, simply adds the overhead. Also in our approach the number of factors is more for all the data sets and hence the overhead. Additional experiments to get the effect of number of factors on the incremental construction were conducted and the results showed that as the number of factors increased the merging time of bwt-disk increased significantly thus making it prohibitive to be used for increasingly large data sets.

But the slow increase in merging time of our approach with the increase in number of factors makes it suitable to be used for large data sets also

6. Conclusion

We have given a method here to construct the extended suffix array of a text T using the Lyndon factorization of text. This is an incremental method, which constructs the Extended SA block by block with small memory requirement, which includes space for BWT, SA and rank array of a Lyndon Factor along with the space for whole SA and LCP. Experimental analysis of the approach shows the performance gain achieved by using the Lyndon factorization for incremental construction. The proposed method can be easily adapted for the space efficient construction of

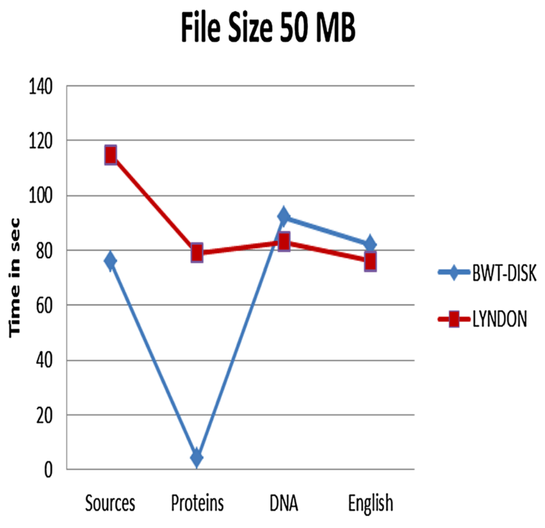


Figure 4. Construction time of BWT-DISK and LYNDON approaches for files of size 50 MB.

other variants of suffix array like compressed suffix arrays and compressed suffix trees so that it can be efficiently used in case of applications involving large data sets. Also, this method is easily adaptable to external memory or parallel scenarios. So, the proposed method can be used for distributed indexes as well as in various bioinformatics applications. Space/time trade-off can be adjusted according to the requirements by using the dynamic data structures for rank and select operations, which provide the additional information required during the merging step.

As future perspectives, we will try to take advantage of the sampling of SA to further reduce the space requirements of the method. Similarly to what we developed for the *ESA*, we will study how we can adapt this method for incremental construction of compressed Suffix array. Finally, a promising perspective will be to reduce the space occupancy of the index by considering the compressed counterparts of various additional structures used during the construction process.

References

[1] Ferragina P and Manzini G 2005 Indexing compressed text. *J. Assoc. Comput. Mach.* 52: 552–581

[2] Grossi R and Vitter J 2005 Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. 35: 378–407

[3] Hon W, Lam T, Sadakane K, Sung W and Yiu S 2007 A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica* 48: 23–36

[4] Ferragina P and Manzini G 2000 Opportunistic data structures with applications. In: *Proceedings of Annual Symposium on Foundations of Computer Science*, pp. 390–398

[5] Makinen, V. and G. Navarro 2005 Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.* 12: 40–66

[6] Canovas R and Navarro G 2010 Practical compressed suffix trees. In: *Proceedings of the International Conference on Experimental Algorithms*: LNCS. 6049, pp. 94–105

[7] Fischer J, Makinen V and Navarro G 2008 An(other) entropy-bounded compressed suffix tree. In: *CPM*: *LNCS.* 5029, pp. 152–165

[8] Sadakane K 2007 Compressed suffix trees with full functionality. *Theory Comput. Syst.* 41: 589–607

[9] Valimaki N, Makinen V, Gerlach W and Dixit K 2009 Engineering a compressed suffix tree implementation. *ACM J. Exp. Algorithmics* 14: article 2

[10] Weiner P 1973. Linear pattern matching algorithms. In: *Proceedings of the Annual Symposium on Foundations of Computer Science*, pp. 1–11

[11] Gusfield D 1997 Algorithms on strings, trees, and sequences. *Computer Science and Computational Biology.* Cambridge University Press, Cambridge

[12] Ferragina P and Grossi R 1999 The string B-Tree: a new data structure for string search in external memory and its applications. J. ACM 46: 236–280

[13] Sinha R, Puglisi S J, Moffat A and Turpin A 2008 Improving suffix array locality for fast pattern matching on disk. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 661–672

[14] Chen K T, Fox R H and Lyndon R C 1958 Free differential calculus, IV. The quotient groups of the lower central series. *Ann. Math.* 68(1): 81–95

[15] Duval J P 1983 Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4): 363–381

[16] Brlek S, Lachau J O, Provençal X and Reutenauer C 2009 Lyndon + Christoffel = Digitally Convex. *Pattern Recogn.* 42(10): 2239–2246

[17] Hohlweg C and Reutenauer C 2003 Lyndon words, permutations and trees. *Theoret. Comput. Sci.* 307(1): 173–178

[18] Berstel J, Lauve A, Reutenauer C and Saliola F 2008 *Combinatorics on words: Chritoffel words and repetition in words.* CRM Monograph Series. American Mathematical Society, 27, Providence, Rhode Island

[19] Bonomo S, Mantaci S, Restivo A, Rosone G and Sciortino M 2013 Suffixes, Conjugates and Lyndon words. *Lect. Notes Comput. Sci.* 7907: 131–142

[20] Manber U and Myers G 1993 Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.* 22(5): 935–948

[21] Burrows M, Wheeler and David J 1994 *A block sorting lossless data compression algorithm*, Technical Report 124, Digital Equipment Corporation

[22] Sadakane K 2000 Compressed text databases with efficient query algorithms based on the compressed suffix array. In: *ISAAC'00*, *LNCS* 1969: 410–421

[23] Fiala M, and Holub J 2008 DCA using suffix arrays. In: *Data Compression Conference DCC'2008*, pp. 516

[24] Sestak R, Lnsk J and Zemlicka M 2008 Suffix array for large alphabet. In: *Data Compression Conference DCC'2008*, pp. 543

[25] Bieganski P, Riedl J and Carlis J V 1994 Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation. In: *Proceedings of the 27th Annual Hawaii International Conference on System Sciences. Hawaii: IEEE*, pp. 34–55

[26] Vyverman M, De Baets B, Fack V and Dawyndt P 2013 essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics* 29(6): 802–804

[27] Schröder J, Schröder H, Puglisi SJ, Sinha R and Schmidt B 2009 SHREC: a short-read error correction method, *Bioinformatics*. 25(17): 2157–2163

[28] Gonnella G and Kurtz S 2012 Readjoiner: a fast and memory efficient string graph-based sequence assembler. *BMC Bioinform.* 13(1): 1–19

[29] Hazelhurst S and Lipák Z 2011 KABOOM! A new suffix array based algorithm for clustering expression data. *Bioinformatics* 27(24): 3348–3355

[30] Mantaci S, Restivo A, Rosone G and Sciortino M 2013 Sorting suffixes of a text via its Lyndon Factorization. *Stringology* 119–127

[31] Mantaci S, Restivo A, Rosone G and Sciortino M 2014 Suffix array and Lyndon factorization of a text. *J. Discrete Algorithms* 28: 2–8

[32] Apostolico A and Crochemore M 1995 Fast parallel Lyndon factorization with applications. *Math. Syst. Theory* 28(2): 89–108

[33] Ghuman S S, Giaquinta E and Tarhio J 2014 Alternative algorithms for Lyndon Factorization. *Stringology* 169–178

[34] Ferragina P, Gagie T and Manzini G 2012 Lightweight data indexing and compression in external memory. *Algorithmica.* 63(3): 707–730

[35] Makinen V and Navarro G 2008 Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Alg.* 4(3): article 32

[36] González R and Navarro G 2008 Improved dynamic rank-select entropy-bound structures. In: *Proceedings of the Latin American Theoretical Informatics (LATIN), Lecture Notes in Computer Science.* 4957

[37] Karkkainen J, Sanders P and Burkhardt S 2006 Linear work suffix array construction. *J. ACM* 53(6): 918–936

[38] Nong G, Zhang S and Chan W H 2009 Linear suffix array construction by almost pure induced-sorting. In: DCC, James A. Storer and Michael W. Marcellin (Eds.), *IEEE Computer Society*, pp. 193–202

[39] Bender M A and Farach-Colton M 2000 *The LCA problem revisited*, *Lecture Notes in Computer Science,* pp. 88–94

[40] Karkkainen J, Manzini G and Simon J P 2009 Permuted longest common-prefix array. In: CPM (Gregory Kucherov and Esko Ukkonen, eds.), *Lecture Notes in Computer Science*. 5577: 181–192

[41] Kasai T, Lee G, Arimura H, Arikawa S and Park K 2001 Linear-time longest-common-prefix computation in suffix arrays and its applications. In: *Proceedings of Annual Symposium on Combinatorial Pattern Matching* 2089: 181–192