Sorting Suffixes of a Text via its Lyndon Factorization

Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino

University of Palermo, Dipartimento di Matematica e Informatica, Italy
{sabrina,restivo,giovanna,mari}@math.unipa.it

Abstract. The process of sorting the suffixes of a text plays a fundamental role in Text Algorithms. They are used for instance in the constructions of the Burrows-Wheeler transform and the suffix array, widely used in several fields of Computer Science. For this reason, several recent researches have been devoted to finding new strategies to obtain effective methods for such a sorting. In this paper we introduce a new methodology in which an important role is played by the Lyndon factorization, so that the local suffixes inside factors detected by this factorization keep their mutual order when extended to the suffixes of the whole word. This property suggests a versatile technique that easily can be adapted to different implementative scenarios.

Keywords: sorting suffixes, BWT, suffix array, Lyndon words, Lyndon factorization

1 Introduction

The sorting of the suffixes of a text plays a fundamental role in Text Algorithms with several applications in many areas of Computer Science and Bioinformatics. For instance, it is a fundamental step, in implicit or explicit way, for the construction of the suffix array (SA) and the Burrows-Wheeler Transform (bwt). The SA, introduced in 1990 (cf. [19]), is a sorted array of all suffixes of a string, where the suffixes are identify by using their positions in the string. Several strategies that privilege the efficiency of the running time or the low memory consumption have been widely investigated (cf. [22,16]). The bwt, introduced in 1994 (cf. [6]), permutes the letters of a text according to the sorting of its cyclic rotations, making the text more compressible (cf. [2]). A recent survey on the combinatorial properties that guarantee such a compressibility after the application of bwt can be found in [25] (cf. also [23]). Moreover, in the last years the SA and the bwt, besides being important tools in Data Compression, have found many applications well beyond its original purpose (cf. [1,13,14,20,26,8,2]).

The goal of this paper is to introduce a new strategy for the sorting of the suffixes of a word that opens new scenarios of the computation of the SA and the bwt.

Our strategy uses a well known factorization of a word W called the *Lyndon factorization* and is based on a combinatorial property proved in this paper, that allows to sort the suffixes of W ("global suffixes") by using the sorting of the suffixes inside each block of the decomposition ("local suffixes").

The Lyndon factorization is based on the fact that any word W can be written uniquely as $W = L_1 L_2 \cdots L_k$, where

- the sequence $L_1, L_2, \ldots, L_k$ is non-increasing with respect to lexicographic order;
- each L_i is strictly less than any of its proper cyclic shift (Lyndon words).

Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, Marinella Sciortino: Sorting Suffixes of a Text via its Lyndon Factorization, pp. 119–127.
Proceedings of PSC 2013, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05330-0 © Czech Technical University in Prague, Czech Republic

This factorization was introduced in [7] and a linear time algorithm is due to Duval [11]. The intuition that the knowledge of Lyndon factorization of a text can be used for the computation of the suffix array of the text itself has been introduced in [5]. Conversely, a way to find the Lyndon factorization from the suffix array can be found in [17].

If U is a factor of a word W we say that the sorting of the local suffixes of U is *compatible* with the sorting of the global suffixes of W if the mutual order of two local suffixes in U is kept when they are extended as global suffixes. The main theorem in this paper states that if U is a concatenation of consecutive Lyndon factors, then the local suffixes in U are compatible with the global suffixes. This suggests some new algorithmic scenarios for the constructions of the SA and the *bwt*. In fact, by performing the Lyndon factorization of a word W by Duval's algorithm, one does not need to get to the end of the whole word in order to start the decomposition into Lyndon factors. Since our result allow to start the sorting of the local suffixes (compatible with the sorting of the global suffixes) as soon as the first Lyndon word is discovered, this may suggest an online algorithm, that do not require to read the entire word to start sorting. Moreover, the independence of the sorting of the local suffixes inside the different Lyndon factors of a text suggests also a possible parallel strategy to sort the global suffixes of the text itself.

In Section 2 we give the fundamental notions and results concerning combinatorics on words, the Lyndon factorization, the Burrows-Wheeler transform and the suffix array. In Section 3 we first introduce the notion of global suffix on a text and local suffix inside a factor of the text. Then we prove the compatibility between the ordering of local suffixes and the ordering of global suffixes. In Section 4 we describe an algorithm that uses the above result to incrementally construct the *bwt* of a text. Such a method can be also used to explicitly construct the SA of the text. In Section 5 we discuss about some possible improvements and developments of our method, including implementations in external memory or in place constructions. Finally, we compare our strategy for sorting suffixes with the method proposed in [12] in which a lightweight computation of the *bwt* of a text is performed by partitioning it into factors having the same length.

2 Preliminaries

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite alphabet with $c_1 < c_2 < \cdots < c_\sigma$. Given a finite word $W = a_1 a_2 \cdots a_n$, $a_i \in \Sigma$ for $i = 1, \ldots, n$, a *factor* of W is written as $W[i, j] = a_i \cdots a_j$. A factor $W[1, j]$ is called a *prefix*, while a factor $W[i, n]$ is called a *suffix*. In this paper, we also denote by $suf_W(i)$ as the suffix of W starting from position i. We omit W when there is no danger of ambiguity. We say that $x, y \in \Sigma^*$ are *conjugate* (or *cyclic shift*) or *y is a conjugate of x* if $x = uv$ and $y = vu$ for some $u, v \in \Sigma^*$. Recall that conjugacy is an equivalent relation.

A *Lyndon* word is a primitive word which is also the minimum in its conjugacy class, with respect to the lexicographic order relation. In [18,11], one can find a linear algorithm that for any word $W \in \Sigma^*$ computes the Lyndon word of its conjugacy class. We call it the Lyndon word of W. Lyndon words are involved in a nice and important factorization property of words.

Theorem 1. *[7] Every word $W \in \Sigma^+$ has a unique factorization $W = L_1 L_2 \cdots L_k$ such that $L_1 \geq_{lex} \cdots \geq_{lex} L_k$ is a non-increasing sequence of Lyndon words.*

We call this factorization the *Lyndon factorization* of a word and it can be computed in linear time (see for instance [11,18]). Duval in [11] presents two variants of an algorithm of factorization of a word into Lyndon words in time linear in the length of the word. The first variant of the algorithm uses only three variables for a complete computation and it requires no more than $2n$ comparisons between two letters. The second one is slightly faster in that sense that it requires no more than $\frac{3n}{2}$ comparisons but it uses an auxiliary storage of size $\frac{n}{2}$. The basis idea for both these variants is finding each factor of the decomposition of the word W from left to right by eventually reading a long enough prefix of the next Lyndon factor.

Lyndon factorization has been realized also in parallel (cf. [3]) and in external memory (cf. [24]).

One way to define the Burrows-Wheeler Transform (*bwt*) [6] of a string W of length n (although not the most efficient way to compute it) is to construct all n cyclic shifts of W and sort them lexicographically. The output of *bwt* consists of the pair (L, I), where L is the sequence of the last character of each rotation in the sorted list and I is an integer denoting the position of the original word in the list.
Another more efficient way consists in the concatenating at the the input string W a symbol \$ that is smaller than any other letter. In this case, the *bwt* is intuitively described as follows: given a word $W \in \Sigma^*$, $bwt(W)$ is a word obtained by sorting the list of the suffixes of $W\$$ and by concatenating the symbols preceding in W each suffix in the sorted list. In both the cases, it is an invertible transform, i.e., one can recover the original text from its *bwt*.

Note that, in general, the sorting of the conjugates of a word W and the sorting of the suffixes of a word $W\$$ is different, but, as consequence of the properties of Lyndon words, when the word W is the Lyndon word, then the two sorting coincide (cf. [15, Lemma 12]). A study of the combinatorial aspects that connect these two sorting can be found in [5]. In this study an important role is played by the notion of Lyndon word.

Given a text W of length n, the suffix array (SA) for W is an array of integers of range 1 to $n+1$ specifying the lexicographic ordering of the suffixes of the string W. It will be convenient to assume that $W[n+1] = \$$, where \$ is smaller than any other letter. That is, $SA[j] = i$ if and only if $W[i, n+1]$ is the j-th suffix of W in ascending lexicographical order.

SA	bwt	$Suffixes$
12	s	\$
2	m	$a\ t\ h\ e\ m\ a\ t\ i\ c\ s$ \$
7	m	$a\ t\ i\ c\ s$ \$
10	i	$c\ s$ \$
5	h	$e\ m\ a\ t\ i\ c\ s$ \$
4	t	$h\ e\ m\ a\ t\ i\ c\ s$ \$
9	t	$i\ c\ s$ \$
1	\$	$m\ a\ t\ h\ e\ m\ a\ t\ i\ c\ s$ \$
6	e	$m\ a\ t\ i\ c\ s$ \$
10	c	s \$
2	a	$t\ h\ e\ m\ a\ t\ i\ c\ s$ \$
8	a	$t\ i\ c\ s$ \$

Figure 1. The table of the lexicographically sorted suffixes of the word *mathematics*\$ together the SA(*mathematics*\$) and the *bwt*(*mathematics*\$).

For instance, if $W = mathematics$ then $bwt(W\$) = smmihtt\$ecaa$ and $SA(W\$) = [12, 2, 7, 10, 5, 4, 9, 1, 6, 10, 2, 8]$. The table obtained by lexicographically sorting all the suffixes of $W\$$ is depicted in Figure 1.

3 Local and global suffixes of a text

Let $W \in \Sigma^*$ and let $W = L_1 L_2 \cdots L_k$ be its Lyndon Factorization. For each factor L_r, we denote by $first(L_r)$ and $last(L_r)$ the position of the first and the last character, respectively, of the factor L_r in W. Let u be a factor of W. We denote by $suf_u(i) = W[i, last(u)]$ and we call it *local suffix* at the position i with respect to u. Note that $suf_W(i) = W[i, n]$ and we call it *global suffix* of W at the position i. We write $suf(i)$ instead of $suf_W(i)$ when there is no danger of ambiguity.

Definition 2. *Let W be a word and let u be a factor of W. We say that the sorting of suffixes of u is* compatible *with the sorting of suffixes of W if for all i, j with $first(u) \leq i < j \leq last(u)$,*

$$suf_u(i) < suf_u(j) \iff suf(i) < suf(j).$$

Notice that in general taken an arbitrary factor of a word W, the sorting of its suffixes is not compatible with the sorting of the suffixes of W. Consider for instance the word $W = abababb$ and its factor $u = ababa$. Then $suf_u(1) = ababa > a = suf_u(5)$ whereas $suf(1) = abababb < abb = suf(5)$.

Theorem 3. *Let $W \in \Sigma^*$ and let $W = L_1 L_2 \cdots L_k$ be its Lyndon factorization. Let $u = L_r L_{r+1} \cdots L_s$. Then the sorting of the suffixes of u is compatible with the sorting of the suffixes of W.*

Proof. Let i and j be two indexes with $i < j$ both contained in u. We just need to prove that $suf(i) > suf(j) \iff suf_u(i) > suf_u(j)$. Let $x = W[j, last(L_s)]$ and $y = W[i, i + |x| - 1]$.

Suppose that $suf(i) > suf(j)$. Then $y \geq x$ by the definition of lexicographic order. If $y > x$ there is nothing to prove. If $x = y$, then $suf_u(j)$ is prefix of $suf_u(i)$, so by the definition of lexicographic order $suf_u(i) > suf_u(j)$.

Suppose now that $suf_u(i) > suf_u(j)$. This means that $y \geq x$. If $y > x$ there is nothing to prove. If $x = y$, the index $i + |x| - 1$ is in some Lyndon factor L_m with $r \leq m \leq s$, then $L_r \geq L_m \geq L_s$. We denote $z = W[i + |x|, last(L_m)]$. Then $suf(i) = xzL_{m+1} \cdots L_k > xL_{s+1} \cdots L_k = suf(j)$, since $z > L_m$ (because L_m is a Lyndon word) and $L_m \geq L_{s+1}$ (since the factorization is a sequence of non increasing factors). □

The above theorem states, in other words, that mutual order of the suffixes of W starting in two positions i and j is the same as the mutual order of the "local" suffixes starting in i and j inside the block obtained as concatenation of the consecutive Lyndon factors including i and j.

As particular case, the theorem is also true when the two suffixes start in the same Lyndon factor.

We recall that, if l_1 and l_2 denote two sorted lists of elements taken from any well ordered set, the operation $merge(l_1, l_2)$ consists in obtaining the sorted list of elements in l_1 and l_2

A consequence of previous theorem is stated in the following proposition.

Proposition 4. *Let* $sort(L_1L_2\cdots L_l)$ *and* $sort(L_{l+1}L_{l+2}\cdots L_k)$ *denote the sorted lists of the suffixes of* $L_1L_2\cdots L_l$ *and the suffixes* $L_{l+1}L_{l+2}\cdots L_k$*, respectively. Then* $sort(L_1L_2\cdots L_k) = merge(sort(L_1L_2\cdots L_l), sort(L_{l+1}L_{l+2}\cdots L_k))$.

This proposition suggests a possible strategy for sorting the list of the suffixes of some word W:

- find the Lyndon decomposition of W, $L_1L_2\cdots L_k$;
- find the sorted list of the suffixes of L_1 and, separately, the sorted list of the suffixes of L_2;
- merge the sorted lists in order to obtain the sorted lists of the suffixes of L_1L_2;
- find the sorted list of the suffixes of L_3 and merge it to the previous sorted list;
- keep on this way until all the Lyndon factors are processed;

This kind of strategy could have several advantages: first of all, one can work online, i.e. one can start sorting suffixes as soon as the first Lyndon factor is individuated. This also allow to integrate the sorting process with the Duval's Algorithm for Lyndon decomposition that outputs Lyndon factors online as well.

The second advantage is that this kind of strategy allows parallelization, since every Lyndon factor can be processed separately for sorting its suffixes. These kind of application would require an efficient algorithm to perform the merging of two sorted lists.

A detailed algorithmic description of this method in order to obtain the *bwt* of a text is given in next section.

4 An incremental algorithm to sort suffixes of a text

In this section we propose an algorithm that incrementally constructs the suffix array SA and the Burrows-Wheeler transform *bwt* of the text W by using its Lyndon factorization. In particular, here we detail the construction of the *bwt* but an analogous reasoning can be done in order to obtain the suffix array. We assume that $L_1L_2\cdots L_k$ is the Lyndon factorization of the word $W[1,n]$. So $L_1 \geq L_2 \geq \cdots \geq L_k$. Such an hypothesis, although strong, is not restrictive because one can obtain the Lyndon factorization of any word in linear time (cf. [11,18]). As shown in previous section, the hypothesis that W is factorized in Lyndon words suggests to connect the problem to the sorting of the local suffixes of W to the lexicographic sorting of the global suffixes of W.

Our algorithm, called BWT_LYND, considers the input text $W[1,n]$ as logically partitioned into k blocks, where each block corresponds to a Lyndon word, and computes incrementally the $bwt(W\$)$ via k iterations, one per block of W. Each block is examined from right to left so that at iteration i we compute $bwt(L_1\cdots L_i\$)$ given $bwt(L_1\cdots L_{i-1}\$)$, $bwt(L_i\$)$ and $SA(L_i\$)$. Remark that the positions in $SA(L_i\$)$ range in $[first(L_i), Last(L_i)+1]$. This means that we sum the amount $|L_1\cdots L_{i-1}|$ to the values of the usual suffix array of $L_i\$$.

The key point of the algorithm comes from Theorem 3, because the construction of $bwt(L_1\cdots L_i\$)$ from $bwt(L_1\cdots L_{i-1}\$)$ requires only the insertion of the characters of L_i in $bwt(L_1\cdots L_{i-1}\$)$ in the same mutual order as they appear in $bwt(L_i\$)$. Note that the character \$ that follows L_i is not considered in this operation.

Moreover, such an operation does not modify the mutual order of the characters already lying in $bwt(L_1\cdots L_{i-1}\$)$.

For each block L_i with i ranging from 1 to k, the algorithm BWT_LYND executes the following steps:

1. Compute the $bwt(L_i\$)$ and $SA(L_i\$)$.
2. Compute the counter array $G[1, |L_i|+1]$ which stores in $G[j]$ the number of suffixes of the string $L_1 \cdots L_{i-1}\$$ which are lexicographically smaller than the j-th suffix of $L_i\$$.
3. Merge $bwt(L_1 \cdots L_{i-1}\$)$ and $bwt(L_i\$)$ in order to obtain $bwt(L_1 \cdots L_{i-1}L_i\$)$.

Example 5. Let $W = aabcabbaabaabdabbaaabbdc$. The Lyndon factorization of W is $L_1L_2L_3$, where $L_1 = aabcabb > L_2 = aabaabdabb > L_3 = aaabbdc$. Figure 2 illustrates how Step 3 of the algorithm works. Note that the positions of the suffixes in $L_2\$$ (i.e. in $SA(L_2\$)$) are shifted of $|L_1| = 7$ positions. Notice that in the algorithm BWT_LYND we do not actually compute the sorted list of suffixes, but we show it in Figure 2 to ease the comprehension of the algorithm. Moreover, the algorithm can be simply adapt to compute the suffix array of W, so in Figure 2 the suffix arrays are also shown.

$L_1\$$

	SA	bwt	Sorted Suffixes
	8	$\underline{b}$	$
	1	$	*aabcabb*$
	5	*c*	*abb*$
	2	*a*	*abcabb*$
	7	*b*	*b*$
	6	*a*	*bb*$
	3	*a*	*bcabb*$
	4	*b*	*cabb*$

$L_2\$$

G	SA	bwt	Sorted Suffixes
0	**11+7=18**	**b**	**$**
0	**1+7=8**	**$**	**aabaabdabb$**
2	**4+7=11**	**b**	**aabdabb$**
2	**2+7=9**	**a**	**abaabdabb$**
2	**8+7=15**	**d**	**abb$**
4	**5+7=12**	**a**	**abdabb$**
4	**10+7=17**	**b**	**b$**
5	**3+7=10**	**a**	**baabdabb$**
5	**9+7=16**	**a**	**bb$**
7	**6+7=13**	**a**	**bdabb$**
8	**7+7=14**	**b**	**dabb$**

⇒

$L_1L_2\$$

SA	bwt	Sorted Suffixes
18	**b**	**$**
8	$\underline{b}$	*aabaabdabb$*
1	$	*aabcabbaabaabdabb*$
11	**b**	**aabdabb$**
9	**a**	**abaabdabb$**
15	**d**	**abb$**
5	*c*	*abbaabaabdabb*$
2	*a*	*abcabbaabaabdabb*$
12	**a**	**abdabb$**
17	**b**	**b$**
7	*b*	*baabaabdabb*$
10	**a**	**baabdabb$**
16	**a**	**bb$**
6	*a*	*bbaabaabdabb*$
3	*a*	*bcabbaabaabdabb*$
13	**a**	**bdabb$**
4	*b*	*cabbaabaabdabb*$
14	**b**	**dabb$**

Figure 2. Iteration 2 of the computation of the *bwt* of the text $W = aabcabb|aabaabdabb|aaabbdc$ on the alphabet $\{a, b, c, d\}$. The two columns represent the *bwt*s before and after the iteration. Note that the first row (the underlined letter) in the table relative to $L_1\$$ and the second row (the underlined suffix) in the table relative to $L_2\$$ flow into the second row in the table relative to $L_1L_2\$$. Indeed, the suffix *aabaabdabb*$ is preceded by the symbol b in $L_1L_2\$$. We use distinct style fonts for each Lyndon word.

Step 1 can be executed in linear time $O(|L_i|)$, if $bwt(L_i\$)$ and $SA(L_i\$)$ are stored in internal memory (see [22,16]).

During Step 2, the algorithm uses the functions C and *rank* described as follows. For any character $x \in \Sigma$, let $C(u, x)$ denote the number of characters in u that are smaller than x, and let $rank(u, x, t)$ denote the number of occurrences of x in $u[1, t]$. Such functions have been introduced in [13] for the FM-index. For sake of simplicity we can firstly construct the array $A[1, |L_i| + 1]$ which stores in $A[j]$ the number of

suffixes of the string $L_1 \cdots L_{i-1}\$$ which are lexicographically smaller than the suffix of $L_i\$$ starting at the position j. Remark that we set $A[1] = 0$ because $L_i[1, |L_i|]\$$ has the same rank of \$ between the suffixes of $L_1 \cdots L_{i-1}\$$ and it is preceded by the same symbol $L_{i-1}(|L_{i-1}|)$ in $L_1 \cdots L_{i-1}L_i\$$. Consequently, in our algorithm considers the suffixes $L_i[1, |L_i|]\$$ and the suffix \$ (of the string $L_1 \cdots L_{i-1}\$$) as the same suffix. It is easy to prove that the value $A[|L_i| + 1]$ is 0. The array A is computed from the position $|L_i|$ to 2 by using Proposition 6.

Proposition 6. *Let j be a integer ranging from $|L_i|$ to 2 and let $A[j + 1]$ be the number of suffixes of $L_1 \cdots L_{i-1}\$$ lexicographically smaller than $L_i[j + 1, |L_i|]\$$. Let c be the first symbol of the suffix $L_i[j, |L_i|]\$$. Then,*

$$A[j] = C(bwt(L_1 \cdots L_{i-1}\$), c) + rank(bwt(L_1 \cdots L_{i-1}\$), c, A[j + 1]).$$

Proof. Since c is the first symbol of the suffix $L_i[j, |L_i|]\$$, then $L_i[j, |L_i|]\$ = cL_i[j + 1, |L_i|]\$$. All the suffixes of $L_1 \cdots L_{i-1}\$$ starting with a symbol smaller than c are lexicographically smaller than $L_i[j, |L_i|]\$$. The number of such suffixes is given by $C(bwt(L_1 \cdots L_{i-1}\$), c)$. Let us count now the number of suffixes that starting with c and are smaller than $L_i[j, |L_i|]\$$. This is equivalent to counting how many c's occur in $bwt(L_1 \cdots L_{i-1}\$)[1, A[j+1]]$. Such a value is given by $rank(bwt(L_1 \cdots L_{i-1}\$), c, A[j + 1])$. □

It is easy to verify that we can obtain the array G by using the array A and the suffix array $SA(L_i\$)$, i.e. $G[i] = A[SA(L_i\$)[i]]$. Note that the array G contains the partial sums of the values of the *gap* array used in [9,12]. However, we could directly compute the array G by using the notion of inverse suffix array ISA[1]. Step 2 could be realized in $O(\sum_{j=1,\ldots,i} |L_j|)$ time because we can build a data structure supporting $O(1)$ time rank queries over $bwt(L_1 \cdots L_{i-1}\$)$. The same time complexity is obtained if the rank queries are executed over $bwt(L_i\$)$.

Step 3 uses G to create the new array $bwt(L_1 \cdots L_i\$)$ by merging $bwt(L_i\$)$ with the $bwt(L_1 \cdots L_{i-1}\$)$ computed at the previous iteration. Such a step implicitly constructs the lexicographically sorted list of suffixes starting in $L_1 \cdots L_{i-1}$ and extending up to end of L_i together with the suffixes of L_i. In order to do this we keep the mutual order between the suffixes of $L_1 \cdots L_{i-1}\$$ and $L_i\$$ thanks to Theorem 3. From the definition of the array G, it follows that the first two positions of the array $bwt(L_1 \cdots L_i\$)$ are the first symbol of $bwt(L_i\$)$ and the first symbol of $bwt(L_1 \cdots L_{i-1}\$)$, respectively. For $j = 3, \ldots, |L_i|$ we copy $G[j]$ values from $bwt(L_1 \cdots L_{i-1}\$)$ followed by the value $bwt(L_i\$)[j]$. It is easy to see that the time complexity of Step 3 is $O(\sum_{j=1,\ldots,i} |L_j|)$, too.

From the description of the algorithm and by proceeding by induction, one can prove the following proposition.

Proposition 7. *At the end of the iteration k, Algorithm* BWT_LYND *correctly computes $bwt(L_1 \cdots L_k\$)$. Each iteration i runs in $O(\sum_{j=1,\ldots,i} |L_j|)$ time. The overall time complexity is $O(k^2M)$, where $M = \max_{i=1,\ldots,k}(|L_i|)$.*

[1] The inverse suffix array ISA of a word $W\$$ is the inverse permutation of SA, i.e., $ISA[SA[i]] = i$ for all $i \in [1, |w| + 1]$. The value $ISA[j]$ is the lexicographical rank of the suffix starting at the position j.

5 Discussions and conclusions

The goal of this paper is to propose a new strategy to compute the *bwt* and the SA of a text by decomposing it into Lyndon factors and by using the compatibility relation between the sorting of its local and global suffixes. At the moment, the quadratic cost of the algorithm could make it impractical. However, from one hand, in order to improve our algorithm, efficient dynamic data structure for the rank operations and for the insertion operations could be used. Navarro and Nekrich's recent result [21] on optimal representations of dynamic sequences shows that one can insert symbols at arbitrary positions and compute the rank function in the optimal time $O(\frac{\log n}{\log \log n})$ within essentially $nH_0(s) + O(n)$ bits of space, for a sequence s of length n. On the other hand, our technique, differently from other approaches in which partitions of the text are performed, is quite versatile so that it easily can be adapted to different implementative scenarios.

For instance, in [12] the authors describe an algorithm, called BWTE, that logically partitions the input text W of length n into blocks of the same length m, i.e. $W = T_{n/m}T_{n/m-1}\cdots T_1$ and computes incrementally the *bwt* of W via n/m iterations, one per block of W. Text blocks are examined from right to left so that at iteration $h+1$, they compute and store on disk $bwt(T_{h+1}\cdots T_1)$ given $bwt(T_h\cdots T_1)$. In this case the mutual order of the suffixes in each block depends on the order of the suffixes of the next block. Our algorithm BWT_LYND builds the *bwt* of a text or its SA by scanning the text *from left to right* and it could run online, i.e. while the Lyndon factorization is realized. One of the advantages is that adding new text to the end does not imply to compute again the mutual order of the suffixes of the text analyzed before, unless for the suffixes of the last Lyndon word that could change by adding characters on the right. Moreover, as described in the previous section, the text could be partitioned into several sequences of consecutive blocks of Lyndon words, and the algorithm can be applied *in parallel* to each of those sequences. Furthermore, also the Lyndon factorization can be performed in parallel, as shown in [3]. Alternatively, since we read each symbol only once, also an in-place computation could be suggested by the strategy proposed in [10], in which the space occupied by text W is used to store the $bwt(W)$.

Finally, in the description of the algorithm we did not mention the used workspace. In fact, it could depend on the time-space trade-off that one should reach. For instance, the methodologies used in [4,12] where disk data access are executed only via sequential scans could be adapted in order to obtain a lightweight version of the algorithm. An external memory algorithm for the Lyndon factorization can be found in [24]. We remark that that the method proposed in [12] could be integrated into BWT_LYND in the sense that one can apply BWTE to compute at each iteration the *bwt* and the SA of each block of the Lyndon partition.

In conclusion, our method seems lay out the path towards a new approach to the problem of sorting the suffixes of a text in which partitioning the text by using its combinatorial properties allows it to tackle the problem in local portions of the text in order to extend efficiently solutions to a global dimension.

References

1. M. ABOUELHODA, S. KURTZ, AND E. OHLEBUSCH: *The enhanced suffix array and its applications to genome analysis*, in Algorithms in Bioinformatics, vol. 2452 of LNCS, Springer Berlin Heidelberg, 2002, pp. 449–463.

2. D. Adjeroh, T. Bell, and A. Mukherjee: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*, Springer Publishing Company, Incorporated, first ed., 2008.
3. A. Apostolico and M. Crochemore: *Fast parallel lyndon factorization with applications.* Mathematical systems theory, 28(2) 1995, pp. 89–108.
4. M. J. Bauer, A. J. Cox, and G. Rosone: *Lightweight algorithms for constructing and inverting the BWT of string collections.* Theoretical Computer Science, 483(0) 2013, pp. 134–148.
5. S. Bonomo, S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino: *Suffixes, Conjugates and Lyndon words*, in DLT, vol. 7907 of LNCS, Springer, 2013, pp. 131–142.
6. M. Burrows and D. J. Wheeler: *A block sorting data compression algorithm*, tech. rep., DIGITAL System Research Center, 1994.
7. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus. IV. The quotient groups of the lower central series.* Ann. of Math. (2), 68 1958, pp. 81–95.
8. A. J. Cox, T. Jakobi, G. Rosone, and O. B. Schulz-Trieglaff: *Comparing DNA sequence collections by direct comparison of compressed text indexes*, in WABI, vol. 7534 LNBI of LNCS, Springer, 2012, pp. 214–224.
9. A. Crauser and P. Ferragina: *A theoretical and experimental study on the construction of suffix arrays in external memory.* Algorithmica, 32(1) 2002, pp. 1–35.
10. M. Crochemore, R. Grossi, J. Kärkkäinen, and G. Landau: *Constant-Space Comparison-Based Algorithm for Computing the Burrows-Wheeler Transform*, in CPM, LNCS, Springer, 2013, In press.
11. J.-P. Duval: *Factorizing words over an ordered alphabet.* Journal of Algorithms, 4(4) 1983, pp. 363–381.
12. P. Ferragina, T. Gagie, and G. Manzini: *Lightweight Data Indexing and Compression in External Memory.* Algorithmica, 63(3) 2012, pp. 707–730.
13. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in FOCS 2000, IEEE Computer Society, 2000, pp. 390–398.
14. P. Ferragina and G. Manzini: *An experimental study of an opportunistic index*, in SODA 2001, SIAM, 2001, pp. 269–278.
15. R. Giancarlo, A. Restivo, and M. Sciortino: *From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization.* Theoret. Comput. Sci., 387(3) 2007, pp. 236–248.
16. R. Grossi: *A quick tour on suffix arrays and compressed suffix arrays.* Theoretical Computer Science, 412(27) 2011, pp. 2964–2973.
17. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theoretical Computer Science, 307(1) 2003, pp. 173–178.
18. M. Lothaire: *Applied Combinatorics on Words (Encyclopedia of Mathematics and its Applications)*, Cambridge University Press, New York, NY, USA, 2005.
19. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches*, in Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms, SODA '90, Philadelphia, PA, USA, 1990, SIAM, pp. 319–327.
20. S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino: *A new combinatorial approach to sequence comparison.* Theory Comput. Syst., 42(3) 2008, pp. 411–429.
21. G. Navarro and Y. Nekrich: *Optimal dynamic sequence representations*, in Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2013, pp. 865–876.
22. S. J. Puglisi, W. F. Smyth, and A. H. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Comput. Surv., 39 2007.
23. A. Restivo and G. Rosone: *Balancing and clustering of words in the Burrows-Wheeler transform.* Theoret. Comput. Sci., 412(27) 2011, pp. 3019–3032.
24. K. Roh, M. Crochemore, C. S. Iliopoulos, and K. Park: *External memory algorithms for string problems.* Fundam. Inf., 84(1) 2008, pp. 17–32.
25. G. Rosone and M. Sciortino: *The Burrows-Wheeler Transform between Data Compression and Combinatorics on Words*, in CiE, vol. 7921 of LNCS, Springer, 2013, In press.
26. J. T. Simpson and R. Durbin: *Efficient construction of an assembly string graph using the FM-index.* Bioinformatics, 26(12) 2010, pp. i367–i373.