Dynamic Extended Suffix Arrays

M. Salson [a,1] T. Lecroq [a] M. Léonard [a] L. Mouchard [a,b,*]

[a] *Université de Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan, France*

[b] *Algorithm Design Group, Department of Computer Science, King's College London, Strand, London WC2R 2LS, England*

Abstract

The suffix tree data structure has been intensively described, studied and used in the eighties and nineties, its linear-time construction counterbalancing his space-consuming requirements. An equivalent data structure, the suffix array, has been described by Manber and Myers in 1990. This space-economical structure has been neglected during more than a decade, its construction being too slow. Since 2003, several linear-time suffix array construction algorithms have been proposed, and this structure has slowly replaced the suffix tree in many string processing problems. All these constructions are building the suffix array from the text, and any edit operation on the text leads to the construction of a brand new suffix array. In this article, we are presenting an algorithm that modifies the suffix array and the Longest Common Prefix (LCP) array when the text is edited (insertion, substitution or deletion of a letter or a factor). This algorithm is based on a recent four-stage algorithm developed for dynamic Burrows-Wheeler Transforms (BWT). For minimizing the space complexity, we are sampling the Suffix Array, a technique used in BWT-based compressed indexes. We furthermore explain how this technique can be adapted for maintaining a sample of the Extended Suffix Array, containing a sample of the Suffix Array, a sample of the Inverse Suffix Array and the whole LCP array. Our practical experiments show that it operates very well in practice, being quicker than the fastest suffix array construction algorithm.

Key words: Dynamic, Suffix Array, Edit Operations, Algorithm Design, Self-index Data Structures, FM-index

* Corresponding author: Laurent.Mouchard@univ-rouen.fr

[1] Funded by the French Ministry of Research - Grant 26962-2007

1 Introduction

Index structures are nowadays a must-have for most of the search engines. A preprocessing stage analyzes large known texts (books, web pages, whole genome sequences) and produces index structures that will permit fast answers to numerous requests. The original question that gave birth to these structures is probably due to D. Knuth: "Given two strings T_1 and T_2, find their longest common factor". A very first sound solution appeared in Weiner [32]: he established that the lower bound for solving this problem was $O(|T_1| + |T_2|)$ and presented an algorithm for building a data structure, the suffix tree, that helps in answering this question. In 1976, McCreight [24] proposed a linear-time construction algorithm, introducing suffix links that speed up the visit and practical construction of this data structure, and in 1992, Ukkonen [31] produced an online linear-time algorithm. These three algorithms and the data structure in itself paved the way to numerous applications, as described by Gusfield [12] in 1997. He described in this book a very large range of computational biology applications that can be solved using suffix trees. Despite its very fast construction, this data structure suffers from an important drawback: its memory requirement of about 10 bytes per input letter on average. This limitation confines its use to medium size texts, preventing it from being used for complete genome or sets of large texts. Note that recent articles are describing space-economical versions of suffix trees where redundant information is encoded in a compressed way (e.g. [28,29,33]).

An early space-economical alternative to suffix tree is due to Manber and Myers [22] in 1990. They described the suffix array, an integer array that can efficiently replace the suffix tree. It consists in the list of starting positions of lexicographically sorted suffixes of the text. The $O(n \log n)$ construction algorithm they proposed was unfortunately not fast enough to challenge the suffix tree, confining this array to the role of an elegant but quite useless alternative. In 2003, three linear-time construction algorithms [16,15,14] were proposed almost simultaneously. They paved the way to numerous Suffix Array Construction Algorithms that have been recently analyzed and cleverly classified in [26]. From challenger, the status of suffix arrays is rapidly changing to leader, becoming the data structure of choice for most of the stringology problems to which suffix tree methodology is applicable. New compressed versions of the suffix arrays [17–19] are now offering a very appealing trade-off between time and space.

Nevertheless, a major problem remains: as soon as the text T is edited (insertion, substitution or deletion of a letter or a block of letters), its associated suffix array $SA(T)$ has to be rebuilt from scratch. We are presenting in this article an algorithm that maintains the existing suffix array of a text T, by considering only the edit operations that are affecting the text.

In Section 2 we describe the suffix array structure and its associated LCP (Longest Common Prefix) array. We then present the Burrows-Wheeler Transform, its closeness with the suffix array, together with a recent algorithm that maintains a dynamic Burrows-Wheeler Transform [30]. In Section 3 we present the details of our algorithm that updates simultaneously the suffix array and its associated LCP. In Section 4 we refine our algorithm by sampling the suffix array. In order to explain our algorithm, we are considering the insertion of a single letter in T. We then explain, in section 5, how our algorithm can be extended to handle the insertion of a block of letters, and other edit operations as well. In Section 6 we study the efficiency of our approach, give details about implementations, present our practical results and sketch an algorithm that can be used for producing a dynamic FM-index [7]. Finally, in Section 7 we conclude and draw perspectives.

2 Background

Let the text $T = T[0\,..\,n]$ be a word of length $n+1$ over Σ, a finite ordered alphabet of size σ. The last letter of T is a sentinel letter \$, that has been added to the alphabet Σ and is smaller than any other letter of Σ. A factor starting at position i and ending at position j is denoted by $T[i\,..\,j]$ and a single letter is denoted by $T[i]$ (or T_i to facilitate the reading). We add that when $i > j$, $T[i\,..\,j]$ is the empty word. A factor starting at position 0 is a prefix of T while a factor ending at position n is a suffix of T.

2.1 Suffix array and suffix tree

The suffix array of T, denoted by $SA(T)$, or simply SA, is the list of starting positions of lexicographically sorted suffixes of T. That is $SA[j] = i$ if and only if $T[i\,..\,n]$ is the $(j+1)^{\text{th}}$ suffix of T in ascending order. Its inverse, denoted by $ISA(T)$, indicates for each suffix of T its corresponding row in SA.
From now on, we will use S (resp. I) instead of SA (resp. ISA) in the figures.

In addition to SA, we are maintaining an integer array LCP of size n. For each $i \in [0, n-1]$, $LCP[i]$ is the length of the Longest Common Prefix between $SA[i]$ and $SA[i+1]$. The largest value in LCP corresponds to the length of the longest factor appearing at least twice in T. In our example (Fig. 1), the largest value is *2* (corresponding to **CT** that appears exactly two times, with starting positions **0** and **2**).

The Extended Suffix Array ESA, containing both SA and LCP, can easily replace the standard suffix tree. There exists a simple relationship between

0	C	T	C	T	G	C	$
1	T	C	T	G	C	$	
2	C	T	G	C	$		
3	T	G	C	$			
4	G	C	$				
5	C	$					
6	$						

(a) unsorted suffixes

	S								LCP
6	$								0
5	C	$							1
0	**C**	**T**	C	T	G	C	$		*2*
2	**C**	**T**	G	C	$				0
4	G	C	$						0
1	T	C	T	G	C	$			1
3	T	G	C	$					

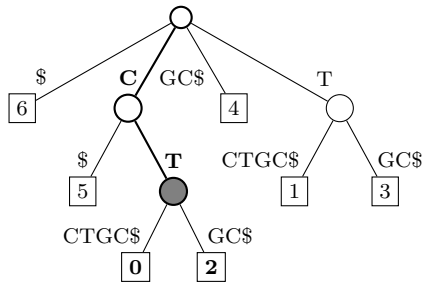
(b) sorted suffixes, SA and LCP

	S	I
0	6	2
1	5	5
2	0	3
3	2	6
4	4	4
5	1	1
6	3	0

(c) SA, ISA

Fig. 1. SA, ISA and LCP for T =CTCTGC$

the suffix tree of T and the ESA, as illustrated in Fig. 2:



	S	LCP
0	6	0
1	5	1
2	**0**	*2*
3	**2**	0
4	4	0
5	1	1
6	3	

SA corresponds to the preorder visit of the ordered suffix tree. $LCP[2] = 2$ (between leaves **0** and **2**) is the sum of length of the words labeling the path from root to least common ancestor node ● (see [5]), $LCP[2] = |\mathbf{C}| + |\mathbf{T}| = 2$.

Fig. 2. Suffix tree and suffix array of T=CTCTGC$

Computing SA and LCP from the suffix tree is straightforward. Symmetrically, reconstructing the suffix tree from the SA and LCP can be easily achieved. Note that the closeness between suffix tree and ESA explains why the problems whose solutions are using a suffix tree, can use ESA as well.

2.2 Burrows-Wheeler Transform

We are now presenting the Burrows-Wheeler Transform [2], a transform that has been intensively studied [9] and is a tool of choice for many text applications. Simply speaking, the transform reorders the text to achieve a better compression, traditionally performed by PPM [3,4]. In what follows, we will use BWT for the transform or for the text resulting from this transform.

The BWT operates on cyclic shifts: a cyclic shift of order i of T (or i^{th} cyclic shift of T) is $T^{[i]} = T[i\,..\,n]T[0\,..\,i-1]$ for a given $0 \leq i \leq n$. The BWT is the text of length $n+1$ corresponding to the last column L of the conceptual matrix M whose rows are the lexicographically sorted cyclic shifts (Figure 3(b)). Note that since M is sorted, the first column F is sorted as well.

There exist two important differences between SA and BWT. First, while SA is made of *integers*, BWT is made of *letters*. Second, while SA builds a list

0	C	T	C	T	G	C	$
1	T	C	T	G	C	$	C
2	C	T	G	C	$	C	T
3	T	G	C	$	C	T	C
4	G	C	$	C	T	C	T
5	C	$	C	T	C	T	G
6	$	C	T	C	T	G	C

(a) unsorted cyclic shifts

	F						*L*
6	$	C	T	C	T	G	C
5	C	$	C	T	C	T	G
0	C	T	C	T	G	C	$
2	C	T	G	C	$	C	T
4	G	C	$	C	T	C	T
1	T	C	T	G	C	$	C
3	T	G	C	$	C	T	C

(b) sorted cyclic shifts

	S
0	6
1	5
2	0
3	2
4	4
5	1
6	3

(c) *SA*

Fig. 3. BWT(CTCTGC$)=CG$TTCC

of *starting* positions of sorted *suffixes*, BWT concatenates *ending* letters of sorted cyclic *shifts*. The list of starting positions of ordered cyclic shifts is identical to the list of starting positions of ordered suffixes. After computing the BWT, we have a text L, which is a permuted version of T, we therefore have to reconstruct T. From L we can easily deduce (sorted) F: in our example, F =$CCCGTT (Figure 3.(b)). We can then construct a one-to-one correspondence between a letter in L and its equivalent letter in F, that is the letter having the same rank (first T in L with first T in F, second C in L with second C in F, ...) as presented in Figure 4.(a). This function, named the LF function establishes the relationship between a row and the next row to be visited for reconstructing T.

	F	*L*
0	$	C
1	C	G
2	C	$
3	C	T
4	G	T
5	T	C
6	T	C

	LF
0	1
1	4
2	0
3	5
4	6
5	2
6	3

(a) constructing LF

L	*L*	*L*	*L*	*L*	*L*	*L*
C	**C**	C	C	C	C	C
G	**G**	**G**	G	G	G	G
$	$	$	$	$	$	**$**
T	T	T	T	**T**	**T**	T
T	T	**T**	**T**	T	T	T
C	C	C	C	C	**C**	C
C	C	C	**C**	**C**	C	C

(b) using LF to reconstruct T (visit commences in $)

Fig. 4. Computing the LF function and reconstructing T using LF

Reconstructing T consists in a visit of the "cycle" we just built, starting from $ in L (we know T ends with $), and collecting letters from L at each step. The right-to-left reconstruction gives: $ → C → G → T → C → T → C. So finally, we obtain CTCTGC$.

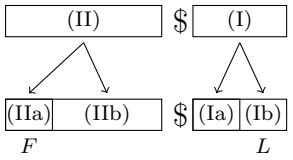
2.3 Dynamic Burrows-Wheeler Transform

The Burrows-Wheeler Transform of T is a static structure that has to be reconstructed from scratch as soon as T is edited. The edit operations we are

considering are the insertion, substitution or deletion of a letter or a factor. We recently designed an algorithm that updates the Burrows-Wheeler Transform of T when T is edited rather than rebuilding it from scratch. In what follows, we are considering, without loss of generality, the text T' that results from the insertion of a single letter c at position i in T.

BWT is operating on cyclic shifts, so we have to understand the impact the insertion of a single letter has on cyclic shifts. We are encountering four different situations, that can be summarized as follow:

$$T'^{[j]} = \begin{cases} T[j-1\,..\,n-1]\ \$\ T[0\,..\,i-1]\ c\ T[i\,..\,j-2] & \text{if } i+1 < j \leq n+1 \quad \text{(Ia)} \\ T[i\,..\,n-1]\ \$\ T[0\,..\,i-1]\ c & \text{if } j = i+1 \quad \text{(Ib)} \\ c\ T[i\,..\,n-1]\ \$\ T[0\,..\,i-1] & \text{if } j = i \quad \text{(IIa)} \\ T[j\,..\,i-1]c\ T[i\,..\,n-1]\ \$\ T[0\,..\,j-1] & \text{if } 0 \leq j < i \quad \text{(IIb)} \end{cases}$$



c appears: (I) right to \$, (II) left to \$.
More precisely:
c appears: (Ia) between \$ and L, (Ib) in L.
c appears: (IIa) in F, (IIb) between F and \$.

Fig. 5. All possible locations of c in $T'^{[j]}$ after its insertion at position i

Based on this analysis, we presented in [30] a four-stage algorithm, corresponding to these four cases. We showed that it maintains an up-to-date version of BWT, and described the various stages:

(Ia)	ignore	No direct impact on either F or L;
(Ib)	modification	For row $ISA[i]$, the letter in L is stored and replaced with c;
(IIa)	insertion	A new row is inserted at position $LF(ISA[i])$, F receives c and L receives the letter that was stored during stage (Ib);
(IIb)	reordering	We gently reorganize the rows that are affected by the insertion.

Let consider the insertion of a letter G at position $i = 2$: $T' = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7}{\text{CTGCTGC\$}}$

	F	L		F	L		F	L
0	\$	C		\$	C		\$	C
1	C	G		C	G		C	G
2	C	\$		C	\$		C	\$
3	C	T	$\overset{\text{(Ib)}}{\rightarrow}$	C	**G**		C	G
4	G	T		G	T		G	T
5	T	C		T	C	$\overset{\text{(IIa)}}{\rightarrow}$	**G**	T
6	T	C		T	C		T	C
7							T	C
	(Ia)			(Ib)			(IIa)	

(Ia) No impact;
(Ib) For $ISA[2] = 3$, the letter T in L is stored and replaced with **G**;
(IIa) Following standard BWT mechanisms, the next row to be considered is $LF(ISA[2]) = LF(3) = 5$, where the new row is inserted.
F receives **G** and L receives stored T.

	F	L		F	L		F	L
0	\$	C		\$	C		\$	C
1	C	G		C	G		C	G
2	C	\$		C	\$		C	G
3	C	G		(C	G)		C	\$
4	G	T		G	T		G	T
5	G	T		G	T		G	T
6	T	C		T	C		T	C
7	(T	C)		T	C		T	C
				(IIb)				

The fourth stage is slightly more complicated: by inserting a new row in M during stage (IIa), we somehow disrupt the LF function and create inappropriate relations between letters in F and L. We therefore have to consider the local rearrangement that might occurs (they consist in rotations, a row k moves to row k' and all the rows between k and k' are shifted by one position accordingly).

These rearrangements are performed as long as the "expected" LF value is different from the actual LF value. The "expected" LF value is computed by summing $rank_c(T, i)$ (the number of c in $T[0\,..\,i]$) and value $C(c)$ (the index of the first row where c is found in F).

This stage uses the following algorithm (as explained in [30]):

REORDER(L, i)
1 $j \leftarrow index(T^{[i-1]})$ ▷ Gives the actual position of $T^{[i-1]}$
2 $j' \leftarrow LF(index(T'^{[i]}))$ ▷ Gives the computed position of $T'^{[i-1]}$
3 **while** $j \neq j'$ **do**
4 $new\_j \leftarrow LF(j)$
5 MOVEROW(L, j, j')
6 $j \leftarrow new\_j$
7 $j' \leftarrow LF(j')$

MOVEROW(L, j, j') moves a row of M from position j to position j'.

3 Our Approach

It has been shown in [30] that the algorithm that maintains the Burrows-Wheeler Transform is very efficient in practice. Due to the closeness between the Burrows-Wheeler Transform and the suffix array, we naturally extend the

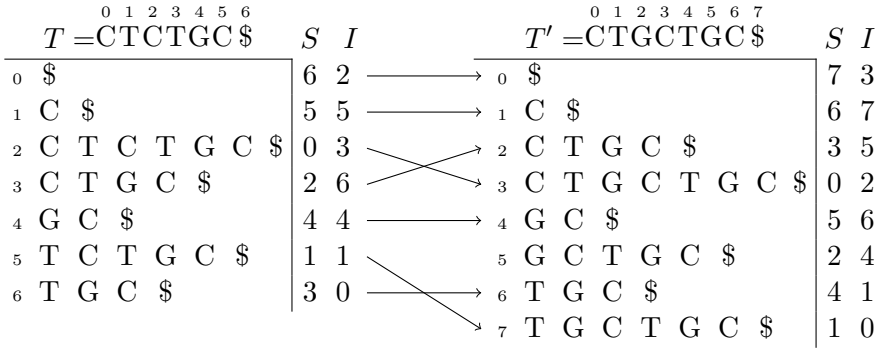
approach to the suffix array and we explain how the LCP array can be updated as well.

3.1 Basic Idea

The LF function offers a convenient way for navigating in L from the j^{th} to the $(j-1)^{\text{th}}$ cyclic shift. It can be easily computed using only the rank and C (count) functions we previously mentioned. Note also that the j^{th} cyclic shift corresponds to the suffix beginning at position j and therefore $LF(j) = j'$ iff $SA[j] = SA[j'] + 1$.
In [30], this function is particularly useful during the reordering stage. Since we have no direct way for calculating LF using only SA, we still need to store L (which gives $rank_a$ functions) and C (count function) which are fundamental for computing LF. This is just a little extra space since L can be stored using only $nH_k(T)$ bits, where H_k is the k-th order entropy, and C using $O(\sigma \log n)$ bits [21,11].

Moreover, the inverse suffix array is indispensable during the reordering stage. In order to initiate stages (Ib) and (IIa), we need to know the location of the row corresponding to $T^{[i]}$ (where i is the position in the text where the edit operation takes place) in the conceptual matrix, given by $ISA[i]$.

We will consider the four-stage algorithm that updates the Burrows-Wheeler Transform and detail, first, its impact on L, and then the induced modifications on both SA, ISA. In order to clarify our approach, we are only considering the insertion of a single letter c at position i in the text T leading to text $T' = T[0\,..\,i-1]cT[i\,..\,n]$, in our example a G at position $i = 2$.



(a) SA and ISA for T (b) SA and ISA for T'

Fig. 6. Insertion of a letter G at position 2 in T =CTCTGC\$: T' =CTGCTGC\$

We start by introducing a lemma (demonstrated in [30]) that will be useful in the following for justifying that, in some cases, there is no modification.

Lemma 1 (Lemma 1, [30]) *Inserting a letter c at position i in T has no effect on the respective ranking of cylic shifts whose orders are strictly greater than i. That is, for all $j \geq i$ and $j' \geq i$, we have $T^{[j]} < T^{[j']} \iff T'^{[j+1]} < T'^{[j'+1]}$.*

3.1.1 Stage 1 (Ia) – suffixes $T[j\,..\,n]$, $\forall j > i+1$

It has been shown in [30, Lemma 1] that the respective ranking of these cyclic shifts is conserved. Therefore, *SA* and *ISA* are not modified during this stage.

3.1.2 Stage 2 (Ib) – suffix $T[i\,..\,n]$

For $k = ISA[i]$, the letter in $L[k]$ is stored ($L_s =$ T) and replaced by c. From [30, Lemma 1], we know that respective ranking of the sorted cyclic shifts is not affected by this modification. Hence, *SA*, *ISA* are not altered.

	F	L	S	I		F	L	S	I
0	$	C	6	2		$	C	6	2
1	C	G	5	5		C	G	5	5
2	C	$	0	3		C	$	0	3
3	C	T	2	6	→	C	**G**	2	6
4	G	T	4	4		G	T	4	4
5	T	C	1	1		T	C	1	1
6	T	C	3	0		T	C	3	0

(Ia) No impact on L.
No impact on *SA* or *ISA*;
(Ib) For $ISA[2] = 3$, we store the original $LF(3) = 5$ in *pos*, the letter T in L is stored in L_s (=T) and replaced with c (=**G**);
No impact on *SA* or *ISA*.

Fig. 7. Stages (Ia) and (Ib): no impact on *SA* and *ISA*

3.1.3 Stage 3 (IIa) – suffix $cT[i\,..\,n]$

A new row is inserted at position $k' = LF(k)$. $F[k'] \leftarrow c$ and $L[k'] \leftarrow L_s = T_{i-1}$

SA: Insertion of i at index k'
(1) all values in *SA*, greater than or equal to i are incremented by 1.
(2) value i is inserted at index k'.
ISA: Insertion of k' at index i
(3) all values in *ISA*, greater than or equal to k' are incremented by 1.
(4) value k' is inserted at index i.

3.1.4 Stage 4 (IIb) – suffixes $T'[j\,..\,n]$, $j < i$

The stage 4 of the algorithm is the reordering stage, it consists in a loop statement that progressively removes discrepancies introduced in the LF function during the insertion of new elements.

	F	L	S	I		F	L	S	I
0	\$	C	6	2		\$	C	~~6~~7	2
1	C	G	5	5		C	G	~~5~~6	~~5~~6
2	C	\$	0	3		C	\$	0	**5**
3	C	G	2	6	→	C	G	~~2~~3	3
4	G	T	4	4		G	T	~~4~~5	~~6~~7
5	T	C	1	1		**G**	**T**	**2**	4
6	T	C	3	0		T	C	1	1
7						T	C	~~3~~4	0

(IIa) From (Ib), we had $pos = \mathbf{5}$,
$LF(3) = C[\mathrm{G}] + rank_{\mathrm{G}}(L, 3) - 1$,
$LF(3) = 4 + 2 - 1 = 5$.
All values greater than or equal to $i = \mathbf{2}$ are incremented in SA.
We update pos: $pos = ~~5~~6$.
All values greater than or equal to $LF(3) = \mathbf{5}$ are incremented in ISA.
A new row is inserted at position 5:
$F \leftarrow c =$ **G** and $L \leftarrow L_s =$ **T**.

Fig. 8. Stage (IIa): impact on SA and ISA

For a given iteration, it moves a row from position j to position j'. Without loss of generality, we will consider that $j < j'$.

L: the element at position j is moved at position j'.
SA: the element at position j is moved at position j'.
ISA: all values between j (excluded) and j' (included) are decremented by 1. Then, j is modified to j'.

	F	L	S	I		F	L	S	I
0	\$	C	7	2		\$	C	7	2
1	C	G	6	6		C	G	6	~~6~~7
2	C	\$	0	5		C	\$	0	5
3	C	G	3	3	→	C	G	3	3
4	G	T	5	7		G	T	5	~~7~~6
5	G	T	2	4		G	T	2	4
6	T	C	1	1		**T**	**C**	**4**	1
7	T	C	4	0		**T**	**C**	**1**	0

(IIb) $LF(5) = C[\mathrm{T}] + rank_{\mathrm{T}}(L, 5) - 1$
$LF(5) = 6 + 2 - 1 = 7$
We compare $pos = 6$ and $LF(5) = 7$
MoveRow(6,7) stores $pos = LF(6) = 2$.
All values in ISA between $6 + 1$ and 7 are decremented.
$ISA[1] = 6$ is set to 7.

	F	L	S	I		F	L	S	I
0	\$	C	7	2		\$	C	7	~~2~~3
1	C	G	6	7		C	G	6	7
2	C	\$	0	5		**C**	**G**	**3**	5
3	C	G	3	3	→	**C**	**\$**	**0**	~~3~~2
4	G	T	5	6		G	T	5	6
5	G	T	2	4		G	T	2	4
6	T	C	4	1		T	C	4	1
7	T	C	1	0		T	C	1	0

(IIb) $LF(7) = C[\mathrm{C}] + rank_{\mathrm{C}}(L, 7) - 1$
$LF(7) = 1 + 3 - 1 = 3$
We compare $pos = 2$ and $LF(7) = 3$.
MoveRow(2,3) stores $pos = LF(2) = 0$.
All values in ISA between $2 + 1$ and 3 are decremented.
$ISA[0] = 2$ is set to 3.

3.2 Updating LCP Values

If we want to deal with an enhanced/extended suffix array, we need to update the LCP array in addition to the SA and ISA. We have to estimate the impact

	F	L	S	I
0	\$	C	7	3
1	C	G	6	7
2	C	G	3	5
3	C	\$	0	2
4	G	T	5	6
5	G	T	2	4
6	T	C	4	1
7	T	C	1	0

(IIb) $LF(3) = C[\$] + rank_{\$}(L, 3) - 1$
$LF(3) = 0 + 1 - 1 = 0$
We compare $pos = 0$ and $LF(3) = 0$
$pos = LF(3) \rightarrow$ no modification left.

Fig. 9. Stage (IIb): impact on SA and ISA

a modification of the SA, ISA has on the LCP. The two first stages (Ia) and (Ib) are not modifying the SA and ISA, and therefore are not alterating LCP. The two last stages (IIa) and (IIb) are performing two different operations:
1) the insertion of a new row at position j in M;
2) the rotation that moves a given row from position j to j'.

1) the insertion of a new row at position j corresponds to the insertion of a single value in SA at index j. It has also underlying impacts (series of increments) that have been described before.

2) the rotation that moves a given row from position j to j' can be decomposed into the deletion of a row from position j, followed by the insertion of a row at position j'. It implies that, in order to deal with rotation, we have to consider the deletion of a letter. This operation supposes the deletion of one value in SA, ISA arrays and in the LCP array as well.

Finally, operations 1) and 2) can be performed using only the insertion and the deletion of a single value in these arrays.

The way the LCP array is modified by an insertion is rather simple. It consists in:

(a) adding at position $j - 1$ the LCP between the suffix starting at position $SA[j - 1]$ and the inserted suffix;
(b) updating at position j the LCP between the inserted suffix and the suffix that previously started at position $SA[j]$.

Similarly, the way the LCP array is affected by a deletion consists in:

(a) computing at position $j - 1$ the LCP between the suffix starting at position $SA[j - 1]$ and suffix starting at position $SA[j + 1]$;
(b) removing the LCP value at position j.

It means that our present goal is to compute the new LCP values. In what follows, we denote by $lcp(u, v)$ the function that returns the length of the longest common prefix between two texts u and v.

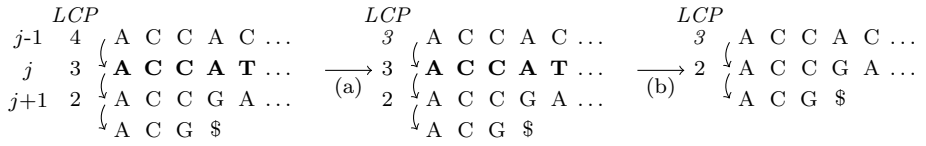
3.3 Deletion of a value in LCP

We start with the simplest operation of the two: the deletion of a value.
Let us consider three consecutive ordered suffixes u, v, w, such that v starts at position $SA[j]$. We know that:

$$lcp(u, w) = \min(lcp(u, v), lcp(v, w)) \tag{1}$$

The deletion of v from the list of sorted suffixes has a direct impact on LCP. We have to:

(a) compute, at position $j-1$, the LCP between the suffix starting at position $SA[j-1]$ and suffix starting at position $SA[j+1]$, that is:
$$LCP[j-1] = \min(LCP[j-1], LCP[j])$$
(b) remove the LCP value at position j.



Deletion of $v =$**A C C A T** and its impact on LCP:
(a) the modified value is computed, $LCP[j-1] = \min(3, 4) = 3$;
(b) $LCP[j]$ is removed from the LCP array.

3.4 Insertion of a value in LCP

Let us consider two consecutive ordered suffixes u, w, such that w starts at position $SA[j]$. We will insert a new suffix v between u and w: v will start at position $SA[j]$ and w will start at position $SA[j+1]$.
Without loss of generality, we will first compute $lcp(u, v)$ and will then deduce or compute (if needed) $lcp(v, w)$.

If $lcp(u, w) = 0$ then the two suffixes are starting with different letters, the number of comparisons that have to be performed is limited.
But it is very likely that u and v are starting with the same letter, since u and v are sorted. In that case, we can take advantage of equation (1) and we know that $lcp(u, v) \geq lcp(u, w)$.

Computing $lcp(u, v)$ can be carried out using two possible strategies, as described below:

Practically speaking, the second strategy performs quicker, the number of

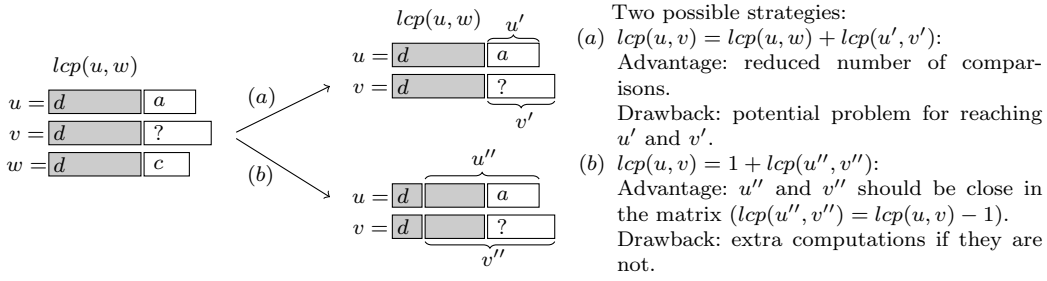
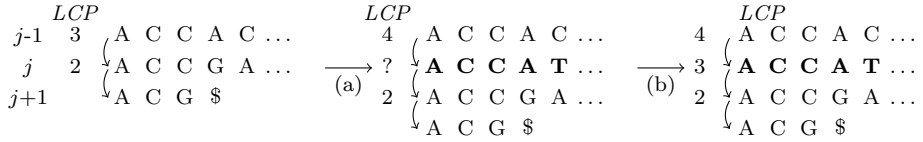


Fig. 10. Two possible strategies for computing $lcp(u, v)$

operations (series of LF^{-1}) that are needed for moving from u to u' being prohibitive.



Insertion of $v =$**A C C A T** and its impact on LCP:
(a) the modified value is computed: $LCP[j-1] = 4$;
(b) the modified $LCP[j-1]$ is larger than the former $LCP[j-1]$, it follows that $LCP[j]$ is equal to former $LCP[j-1]$, $LCP[j] = 3$.

We know that $lcp(u, w) = \min(lcp(u, v), lcp(v, w))$, which implies that either $lcp(u, v)$ or $lcp(v, w)$ is equal to $lcp(u, w)$. We therefore have:

$$lcp(v, w) \neq lcp(u, w) \implies lcp(u, v) = lcp(u, w) \quad (2)$$
$$lcp(u, v) \neq lcp(u, w) \implies lcp(v, w) = lcp(u, w) \quad (3)$$

It follows that, after computing $lcp(u, v)$, we are facing two different cases:

- $lcp(u, v) \neq lcp(u, w)$: using equation (3), we have $lcp(v, w) = lcp(u, w)$ (no further computation is required);
- $lcp(u, v) = lcp(u, w)$: we still need to compute $lcp(v, w)$

3.5 Conclusion

The structures BWT, SA, ISA are up-to-date after the reordering stage. However the LCP table may need some extra computations. Some cyclic shifts, whose lexicographical ranking has not changed, may now have a different LCP value (see Fig. 11). For this reason, we have to continue considering cyclic shifts from right to left. For each of these cyclic shifts, we recompute the LCP value with the cyclic shift preceding and succeeding it, in lexicographical order. The LCP table is finally up-to-date when both values are equal to the original

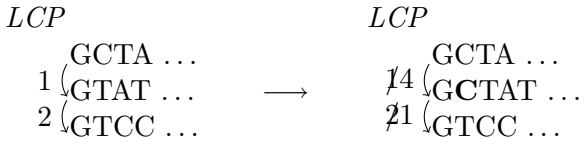


Fig. 11. Impact of a modification on the LCP array

ones.

Handling the three arrays that compose the ESA leads to an overall space-complexity that exceeds $12n$. Moreover every rotation induces the fact that a potentially large range of values in SA and ISA have to be either incremented (or decremented). These updates are both time and space-consuming. The results are therefore intrinsically not satisfying enough, but there is obviously room for improvement. Instead of considering the whole arrays, we can consider samples, that is a limited number of values that represent the two arrays (SA, ISA), reducing the space requirements and the range of values that need to be incremented (decremented).

4 Sampling *SA* and *ISA*

We previously mentioned that both SA and ISA arrays are space-consuming, each array requiring $4n$ bytes in practice. In this section, we are sampling these two arrays to drastically limit the overall space consumption. We are, first, focusing on how ISA can be sampled and are, then, extending our approach to handle SA as well.

4.1 Basic Idea

The problem of reducing the space requirement has already been addressed for existing compressed data structures such as FM-Index or Compact Compressed Suffix Array [7,18,8]. The solution that has been proposed consists in storing only few values, based on an equiprobable distribution over ISA. Storing one value out of $\log^{1+\varepsilon} n$, for $\varepsilon > 0$, makes us store only $n/\log^{1+\varepsilon} n$ values, that is $o(n)$ bits.

Let us consider ISA=$\overset{0}{2}\ \overset{1}{5}\ \overset{2}{3}\ \overset{3}{6}\ \overset{4}{4}\ \overset{5}{1}\ \overset{6}{0}$

A dynamic sample of ISA consists in two arrays: a bit vector m which indicates the positions that are sampled, an integer array v that presents the sample values.

i	0	1	2	3	4	5	6
m	1	0	1	0	1	0	1
v	2		3		4		0

Following [11,20], we are considering the two standard operations, rank/select, on a bit vector bv:
$\text{rank}_c(bv, i)$ returns the number of occurrences of c in $bv[0..i]$;
$\text{select}_c(bv, j)$ returns the position of the j-th occurrence of c in bv.

While these two operations can be performed in constant time using static structures [20], we are limited to $O(\log n)$-time in worst case because of dynamic structures [21] we have to use.

We are facing another problem: updating *SA* or *ISA* implies a lot of elementary updates such as incrementing a series of values in the arrays. Clearly, incrementing $n/\log^{1+\varepsilon} n$ sampled values, for each iteration during stage 4, is something we cannot afford. We have to adapt our current strategy and store the sample values in a structure that authorizes fast updates. For this purpose, we are considering a variant which consists in three arrays: two bit vectors and one integer array.

The first bit vector, m_{ISA}, indicates the positions where *ISA* is sampled.
The second bit vector, v_{ISA} indicates the set of values that are sampled.
The integer array, π_{ISA}, gives the respective order of the sample values: it maps a sample position to its corresponding sample value.

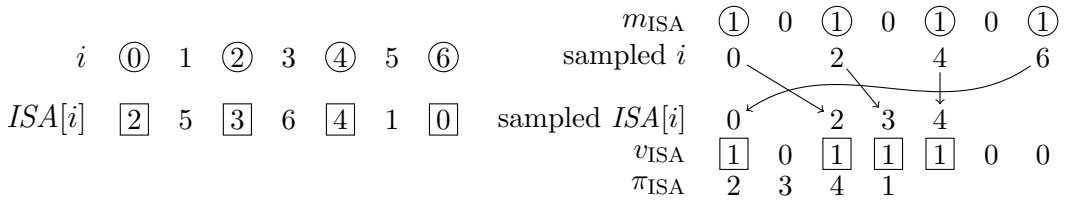
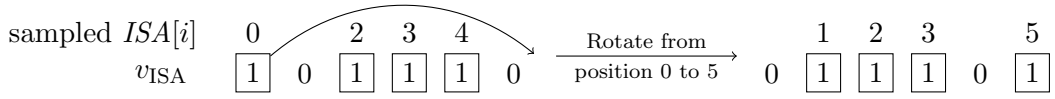


Fig. 12. Sampling *ISA*: two bit vectors and one integer array

The two bit vectors and the integer array:

$m_{\text{ISA}} = 1010101$ corresponds to sample positions $[0, 2, 4, 6]$;
$v_{\text{ISA}} = 1011100$ corresponds to sample values $[0, 2, 3, 4]$;
$\pi_{\text{ISA}} = [2, 3, 4, 1]$ corresponds to the one-to-one mapping between sample positions and associated sample values.

Decrementing all values between j (excluded) and j' (included), during stage 4 of our update algorithm, is simply done by moving the bit in v_{ISA} from position j to position j' (see Fig. 13).



Consequences of the rotation: values between 1 and 5 are decremented, sample value at position 0 is now at position 5.

Fig. 13. Decrementing values in v_{ISA}

4.2 Retrieving any Value of ISA

We replaced the complete ISA=$\overset{0\ 1\ 2\ 3\ 4\ 5\ 6}{2\ 5\ 3\ 6\ 4\ 1\ 0}$ by these three arrays. We have to explain the mechanisms that are used for accessing the original ISA values that are samples in the new structure, or for retrieving the missing value whenever the position we are requesting is not among the sampled ones. We also have to compute the cost of such an operation, the gain we achieved in terms of space should not be counterbalanced by an excessive time overload.

4.2.1 Retrieving a Value at a Sampled Position

Retrieving a value at a sample position i is a simple process. We have to determine:

(a) the rank of the sample position i, *i.e.* $r = \text{rank}_1(m_{\text{ISA}}, i)$;
(b) the rank of sample value corresponding to the sample position, *i.e.* $\pi_{\text{ISA}}(r)$;
(c) the sample value from its rank, *i.e.* $\text{select}_1(v_{\text{ISA}}, \pi_{\text{ISA}}(r))$.

i	0	1	2	3	4	5	6
m_{ISA}	1	0	1 (a)	0	1	0	1
π_{ISA}	2	3 (b)	4	1			
v_{ISA}	1	0	1	1 (c)	1	0	0

(a) m_{ISA}: $m_{\text{ISA}}[2] = 1$, 2 is a sample position.
$\text{rank}_1(m_{\text{ISA}}, 2) = 2$
(b) π_{ISA}: the 2nd value is 3
(c) v_{ISA}: $\text{select}_1(v_{\text{ISA}}, 3) = 3$
Finally, we have $ISA[2] = 3$

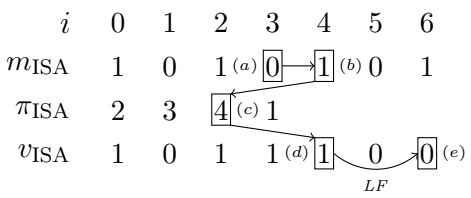
Fig. 14. Retrieving a value for a sample position: $ISA[2]$

Since rank and select functions can be performed in $O(\log n)$ worst-case time, retrieving a value for a sample position costs at most $O(\log n)$ plus the cost of accessing the π_{ISA} dynamic structure. It explains why we will have to pay a lot of attention to this specific data structure, in what follows.

4.2.2 Retrieving a Value at an Unsampled Position

Retrieving a value at an unsampled position i is a more complex process. Since the value we would like to retrieve does not correspond to a sample position, we have to locate one of the surrending positions. We can either choose preceding or succeeding position. We will consider, without loss of generality the succeeding position, that is obtained using $\text{select}_1(\text{rank}_1(m_{\text{ISA}}, i) + 1)$.

(a) determine the rank r of p, the closest sample position to the right of i, $r = \text{rank}_1(m_{\text{ISA}}, i) + 1$;
(b) determine the offset *off* between i and the position of p, $\mathit{off} = \text{select}_1(m_{\text{ISA}}, r) - i$
(c) determine the rank of sample value corresponding to p, $\pi_{\text{ISA}}(r)$;
(d) determine the sample value v corresponding to p, $v = \text{select}_1(v_{\text{ISA}}, \pi_{\text{ISA}}(r))$
(e) determine the unsampled value r' corresponding to i, $r' = LF^{\mathit{off}}(v)$;



(a) m_{ISA}: $m_{\text{ISA}}[3] = 0$, 3 is not a sample position. $\text{rank}_1(m_{\text{ISA}}, 3) + 1 = 3$
(b) m_{ISA}: $\text{select}_1(m_{\text{ISA}}, 3) = 4$. $\mathit{off} = 4 - 3 = 1$
(c) π_{ISA}: the 3rd value is 4
(d) v_{ISA}: $\text{select}_1(v_{\text{ISA}}, 4) = 4$
(e) v_{ISA}: $LF^{\mathit{off}}(4) = LF(4) = 6$.
Finally, we have $ISA[3] = 6$

Fig. 15. Retrieving a value for an unsampled position: $ISA[3]$

The cost for retrieving the sample value is described at Sec. 4.2.1. Additionally, we have to compute *off* and LF^{off}. The former only uses rank and select functions and can be performed in $O(\log n)$-time. The latter consists in *off* successive calls to the LF function and therefore costs *off* times the cost of the LF function. Since LF is calculated using rank_c on general alphabets and C (count) function, we can compute LF in $O\left(\log n \left(1 + \frac{\log \sigma}{\log \log n}\right)\right)$ time using the latest results on dynamic structures [11]. Finally, the time for retrieving a value at an unsampled position is bounded by $O\left(\mathit{off} \times \log n \left(1 + \frac{\log \sigma}{\log \log n}\right)\right)$ plus the time for accessing π_{ISA}.

4.3 Adding or Removing a Sample

The sample values must be as uniformly distributed as possible over *ISA* which means that we cannot have too many (resp. too few) samples in a given interval. More precisely, we will maintain the sample in such a way we are having exactly one sample in a sliding window of size $\alpha \log^{1+\varepsilon} n$ and at most two in a sliding window of size $\beta \log^{1+\varepsilon} n$, for any parameters $\alpha < \beta$. With such restrictions we have an upper bound for the time spent for retrieving any value of *ISA*.

Since we allow any factor to be inserted or removed at any position, we may infringe these conditions. This may happen during stage 3, when we insert or delete an element or during stage 4, when we rotate elements.

In order to insert a sample at position i, we first need to retrieve $ISA[i]$ using the method described in Sec. 4.2.

We, then, have to update the three arrays:
- $m_{\mathrm{ISA}}[i] = 1$ j-th added sample position with $j = \mathrm{rank}_1(m_{\mathrm{ISA}}, i)$
- $v_{\mathrm{ISA}}[ISA[i]] = 1$ k-th added sample value with $k = \mathrm{rank}_1(v_{\mathrm{ISA}}, ISA[i])$
- insertion of k at position j in π_{ISA}

4.4 Updating a Permutation

We have already seen that managing the updates of ISA using v_{ISA} is quite easy. Nevertheless, we also need to update the permutation π_{ISA}. It is potentially modified when we have to insert or delete a value in ISA.

In order to guarantee fast access to the ISA values we have to design an efficient way of storing and updating π_{ISA} (we recall that both rank and select on m_{ISA} and v_{ISA} can be performed in at most $\log n$ steps).

As far as we know, there exists no solution to the problem of storing the permutation π_{ISA} in such a way that the cost for updating it does not exceed the $\log n$ complexity.

Let us formalise the problem: consider a permutation π defined over $\{1, \ldots, n\}$.

(1) After inserting a value i at position j in π, the resulting permutation π' is defined over $\{1, \ldots, n+1\}$ as follows:
   (a) All values greater or equal to i in π are incremented by one in π'
   (b) Value i is inserted at position j.

(2) After deleting an element at position j in π whose value is i, the resulting permutation π' is defined over $\{1, \ldots, n-1\}$ as follows:
   (a) Element at position j is deleted.
   (b) All values greater than i in π are decremented by one in π'.

For solving this problem, we consider two structures A and B containing n nodes. We denote by $A[j]$ the j-th node in A, and $B[i]$ the i-th node in B.

Using A and B, we define a permutation π of n elements as follows: $\pi(j) = i$ if and only if a link is defined from node $A[j]$ to node $B[i]$.

4.4.1 Insertions and Deletions

The insertion of value i at position j is done by inserting a node $A[j]$, a node $B[i]$ and a link from $A[j]$ to $B[i]$.

The insertion of these nodes in A and B shifts by one position all nodes to their right, as described in Fig. 16.

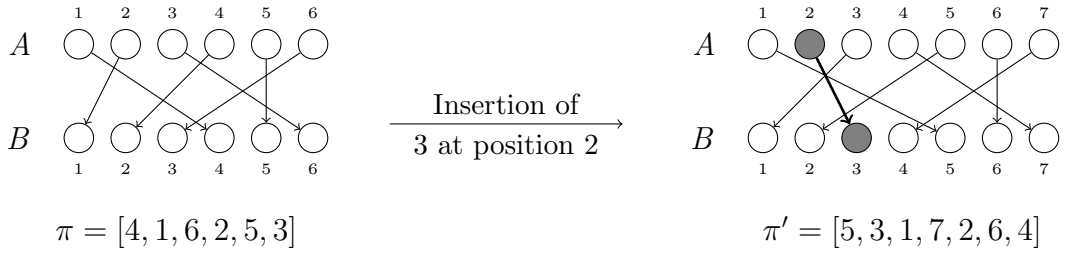


Fig. 16. Insertion in a dynamic permutation

Deleting a given value i is done in a symmetric way: we delete $B[i]$ as well as $A[j]$, with j such that $\pi(j) = i$.

4.4.2 Accessing a Value

Suppose we want to determine $\pi(j)$. First, we go to node $A[j]$, follow the link and get a corresponding node N in B. Then, we have to determine the position of N in B. This position corresponds to the value of $\pi(j)$.

Since we would like to access values in B in $\log n$ time, we are storing the nodes in a balanced binary tree such that $B[i]$ is the i-th node in the in-order traversal of the tree. In order to do so, in each node N we store its rank in the in-order traversal of the subtree rooted at N (see Fig. 17).

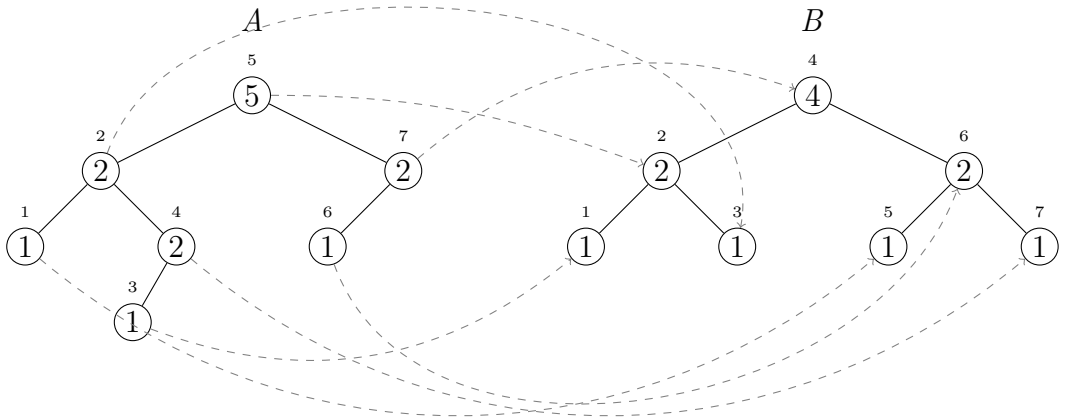


Fig. 17. Representation of the permutation π: [5, 3, 1, 7, 2, 6, 4]

This additional information allows to compute the rank of any node in the in-order traversal of the tree in at most logarithmic time. It can be done using the following algorithm:

```
inOrderRank(N)
1  rank ← N.rank
2  while N ≠ root do
3        if N is in its parent's right subtree then
4              rank ← rank + N.parent.rank
5        N ← N.parent
6  return rank
```

Conversely, given an in-order rank r, we can go to the corresponding node. Starting from the root we have to compare r with the rank R stored in the root:

- $r = R$: the root is the node we are looking for,
- $r < R$: apply the algorithm to the node at position r in the left subtree,
- $r > R$: apply the algorithm to the node at position $r - R$ in the right subtree.

In Fig. 17, for accessing the node at position $r = 6$ in A:

- $R = 5$, $r = 6$ and $r = 6 > 5 = R$: apply the algorithm to the node at position $r = R - r = 6 - 5 = 1$ in the right subtree,
- In the right subtree, we have $R = 2$, $r = 1$ and $r = 1 < R = 2$: apply the algorithm to the node at position $r = 1$ in the left subtree,
- In the left subtree, we have $R = 1$, $r = 1$ and $r = R$: it is the node we are looking for, its rank is 6 in the tree.

Following the link, we arrive at the sixth node in B, therefore $\pi(6) = 6$.

Finally, accessing the value at position j in the permutation is done by calling inOrderRank($A[j].link$).

4.5 Extension to SA

Since *ISA* is the inverse of *SA*, we can compute *SA* using a method similar to the one we used for *ISA*.

For computing *ISA*, π_{ISA} is the mapping between m_{ISA} (positions in *ISA*) and v_{ISA} (values in *ISA*). Symmetrically, for computing *SA*, the inverse of π_{ISA} is the mapping between v_{ISA} (positions in *SA*) and m_{ISA} (values in *SA*).

Therefore, the only requirement for accessing *SA* is to be able to compute π_{ISA}^{-1}. This can be easily done using the structure presented in Sec. 4.4 and by considering that links are bidirectional.

Finally, with little extra effort, we have a sample that allows us to compute

both *ISA* and *SA*.

4.6 Complexity

Without using the sample of *SA* and *ISA*, we had a space requirement of $8n$ bytes. Now, we detail the space consumption of each structure used for the sample.

The bit vectors m_{ISA} and v_{ISA} are *sparse*: they have a small number of bits equal to 1, that is $\Theta(n/\log^{1+\varepsilon} n)$. Mäkinen and Navarro developed a compressed dynamic bit vector [21] that needs $nH_0 + o(n)$ bits and handle all needed operations in $O(\log n)$ time. H_0 denotes the zero-th order entropy of the bit vector and since our bit vectors are sparse, their space consumption is $o(n)$ bits only.

The length of the permutation π_{ISA} is $n/\log^{1+\varepsilon} n$. We are storing two balanced binary trees of $n/\log^{1+\varepsilon} n$ nodes each and every node has a pointer to another node. Pointers are stored in $O(\log n)$ bits under the RAM model, and nodes in the trees can be stored in at most $O(\log n)$ bits. Note that there exist other succinct structures for storing a dynamic binary tree such as [27,1] but using them would not impact the final space complexity. Finally the permutation is stored in $O(n \log n/\log^{1+\varepsilon} n) = o(n)$ bits. The operations for retrieving, inserting or deleting a value are carried out in at most $O(\log n)$ time.

Finally the sample needs $o(n)$ bits and a value can be recovered from the sample in $O(\log n)$ worst-case time.
Since LF can be computed in $O(\log n + \frac{\log n \log \sigma}{\log \log n})$ any value from *SA* and *ISA* can be obtained in $O(\log^{2+\varepsilon} n + \frac{\log^{2+\varepsilon} n \log \sigma}{\log \log n})$ worst-case time.

4.7 Managing the Whole Suffix Array

We have explained how to manage a sampled suffix array (that is a compressed suffix array) for a space-consumption reason. However, one may want to have the whole suffix array for accessing any value more quickly.

Actually, this case is just a special case of the sampled suffix array, where each position is sampled. Therefore, the masks m_{ISA} and v_{ISA} are useless, only π_{ISA} is meaningful since it corresponds exactly to *ISA*. Clearly, the inverse permutation π_{ISA}^{-1} corresponds to *SA*.

Finally, we can access any value of *SA* or *ISA* in $O(\log n)$ worst-case time using $O(n \log n)$ bits.

5 Generalisation to Factors and Deletions/Substitutions

Until now, we only considered the insertion of a single letter in the text. Similarly to what we did for the Burrows-Wheeler Transform in [30], our approach can be extended to the insertion of a factor and other edit operations.

5.1 Inserting a Factor rather than a Single Letter

Let us consider the insertion of a text S of length m, at position i in T. One straightforward solution for inserting S in T consists in applying our algorithm successively to each letter of S. Unfortunately, this solution implies executing m times the reordering stage, which can be very time-consuming. We are presenting here an adaptation of our method, for inserting a factor, that needs only one execution of the reordering stage.

As explained in [30], stages 1, 2 and 4 remain globally the same. The main difference stands during stage 3 where all the letters of S are inserted backwards in the Burrows-Wheeler Transform.

Therefore, for $j = m - 1$ downto 0, we insert in SA the value $i + j$ at position k, the position where $S[j]$ was inserted in L, the last column of the BWT conceptual matrix. Conversely, in ISA we insert k, at position $i + j$.

5.2 Deleting a Factor

Let us consider the deletion of the substring of length m starting at position i in T. All the rows corresponding to $T^{[k]}$, where $i \leq k < i + m$ are deleted in L.

When a row r is deleted in L, we delete values at position $SA[r]$ in ISA and at position r in SA. All values in ISA greater than r are decremented by one.

5.3 Substituting a Factor

Let us consider the substitution of $T[i \,..\, i + m - 1]$ by $S[0 \,..\, m - 1]$.

In F and L the elements are substituted accordingly to S. Because of these substitutions, the modified rows may be moved as well: their lexicographic ranking has changed.

Thus, in that case, the modifications in L are equivalent to the modifications during the reordering stage (rows are moved). Therefore, the modifications in SA and ISA are done on the same principle than during the reordering stage.

5.4 Updating the LCP Table

We previously defined two basic update operations on the LCP table: inserting a new value at a given position and deleting a value from a given position.

Given the three basic modifications on the suffix array, we detail what are the operations to be used on the LCP table:

- Inserting $SA[j]$: insertion of a new value in LCP at position j (Sec. 3.4).
- Deleting $SA[j]$: deletion of $LCP[j]$ (Sec. 3.3).
- Moving $SA[j]$ at position i: deletion of $LCP[j]$ and insertion of a new value at position i (Sec. 3.3 & 3.4.)

6 Experiments and Results

Even if some work has already been done to allow any modification on an indexed text [6,13], we are not aware of any practical implementation. Therefore we compared our method to the current fastest implementation that constructs the suffix array, using different types of texts.

The reconstruction of the suffix array we considered is the very efficient algorithm by Maniscalco and Puglisi [23]. We used the implementation available on the authors website [2]: `MSufSort` version `3.1 Beta`.

Gerlach [10] kindly provided us his implementation of dynamic structures for rank and select. We used these structures in our implementation.

The C/C++ code has been compiled using `gcc` version `4.3` with option `-O2` under a Linux `2.6.27`. The execution times have been measured using the C function `gettimeofday`. The tests were performed on a laptop with 2 GB of RAM, using a single processor of an Intel Core 2 Duo at 1.83 GHz. No extra process was running on this dedicated machine during the tests.

We performed the tests on DNA and English texts from the Pizza&Chili corpus [3] and a random text drawn from an alphabet of size 100.

[2] `www.michael-maniscalco.com/msufsort.htm`
[3] `pizzachili.dcc.uchile.cl`

We studied both space and time consumptions for a variety of sampling factors. We carried out these studies by considering three different types of text of length one million symbols: DNA, random and English texts (we added for extending the scope of our study).

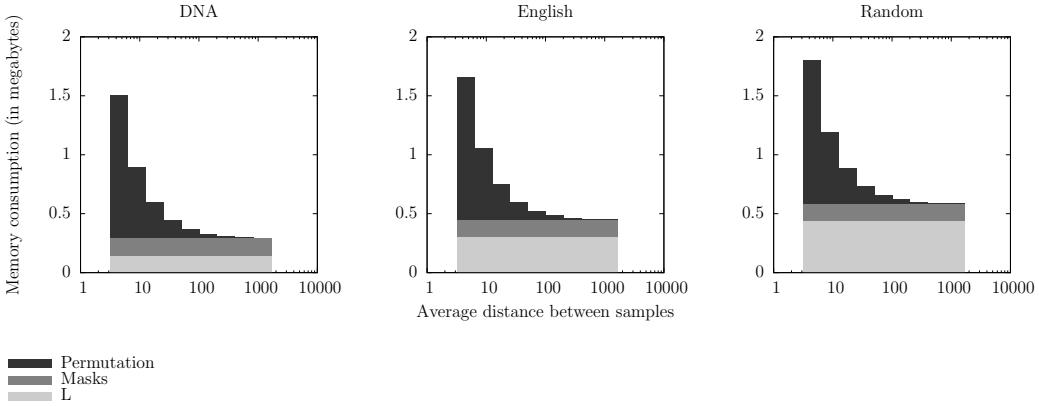


Fig. 18. Space consumption of our index, using texts of size 100 KB

In Fig. 18, in all cases we observe that most of the space used corresponds to L. It is stored in a Huffman-shaped wavelet tree, as described in [25], that explains the memory consumption of L for the different types of text. The masks, yet uncompressed, (storing m_{ISA} and v_{ISA}) depend uniquely on the length of the text. The permutations are directly impacted by the sampling factor, the largest this factor is, the smallest the size of the permutation will be.

Similarly, under the same conditions we measured the time that has been used for updating the sampled suffix array.

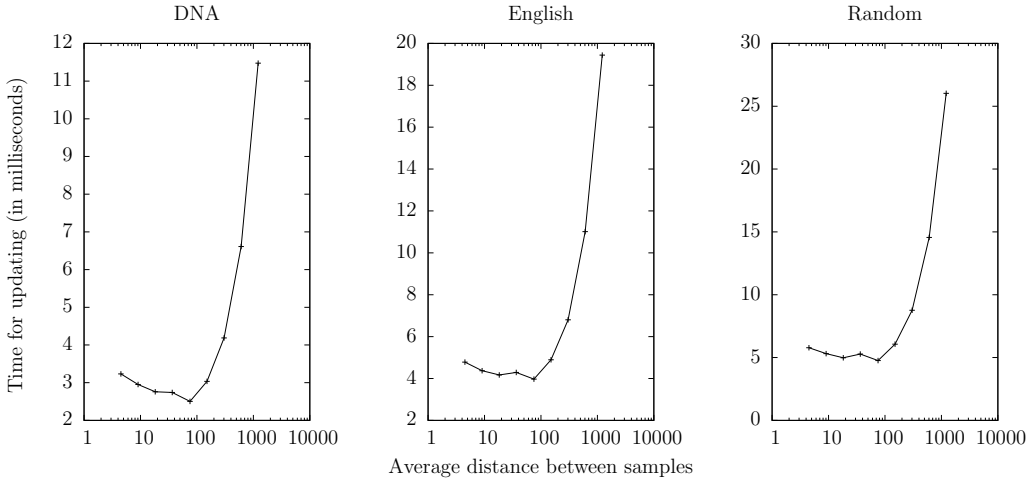


Fig. 19. Time for updating our index, using texts of size 100 KB and 50 insertions of length 10.

In all cases, we observe (in Fig. 19) a local minimum for a sampling factor less

than 100. The sampling factors less than that minimum are time-consuming because of the necessary modifications that affect the sampling. The sampling factors greater than that minimum are time-consuming because of the time that has to be used for recovering distant unsampled values.

Then, different prefixes of the same texts were considered (from size 100 KB to 50 MB) and in each of them we inserted 500 letters. These 500 letters were inserted one by one or by factors of length 10, 20, 50 or 500. The insertions were repeated 2000 times and an average value was computed over the 2000 execution times we obtained. The results are presented in Fig. 20, where the graphs use a logarithmic scale on the y-axis.

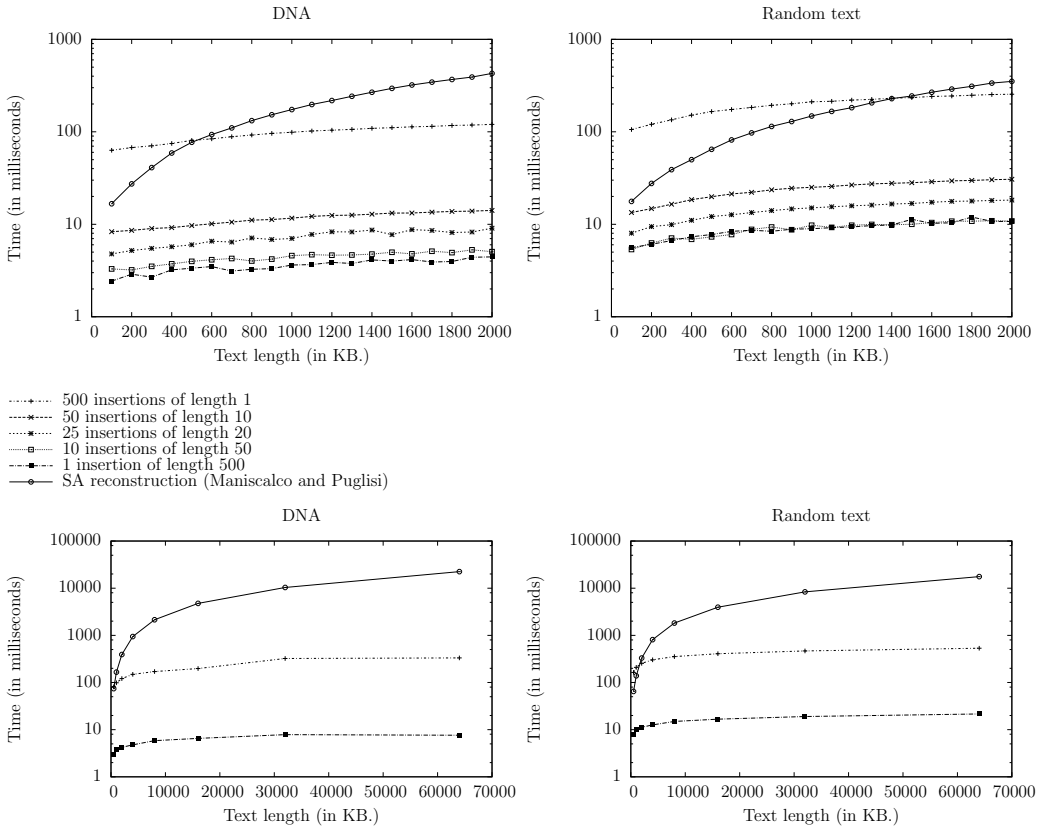


Fig. 20. Execution time for computing the modified index after the insertion of 500 letters.

The graphics show that our algorithm performs very well in practice even for many little insertions (50 insertions of size 10, for instance). More generally, the fewer insertions the quicker the algorithm is: many insertions means many reordering stages which is the most time-consuming part of our algorithm. Moreover, our algorithm is marginally dependent on the size of the text (for the texts we considered) since the execution time grows very slowly.

7 Conclusion and perspectives

In this article, we demonstrated a four-stage algorithm for updating the extended suffix array of a text T that has been edited. For clarifying our presentation, we considered a simple version that handles the insertion of a single letter in T. It can be easily adapted to deal with the insertion of a block, as well as the other two standard edit operations: deletion and substitution. This algorithm, which maintains an up-to-date version of SA, ISA and LCP, is intrinsically not satisfying enough, since potentially a large range of values in SA and ISA have to be manipulated (increment/decrement) during the most demanding stage.

We, then, presented an enhanced version that considers a sample of both SA and ISA together with the whole LCP. We explained how the SA and ISA can be represented using two bit vectors and a small integer array π_{ISA}. We also show how we can replace the series of increments or decrements on SA or ISA by a simple operation on π_{ISA}. Note that the samples that are considered are the ones that FM-index [7] is using, leading to the first dynamic FM-Index being able to handle any standard edit operations.

Finally, the tests we conducted prove that our algorithm is clearly quicker than the fastest known suffix array construction algorithm for the type of modifications that are usually observed.

In terms of perspectives, we will try to take advantage of the method presented in [6] for bounding the maximal number of modifications with a sub-linear term. Similarly to what we developed for the BWT and the ESA, we will study how we can adapt our strategy to other self index data structures. Moreover, since our technique performs very well on the DNA sequences we tested, we will use it for indexing whole genome sequences, maintaining an up-to-date version of its ESA anytime a modification arises. Finally, a promising perspective will be to reduce the space occupancy of the index by considering compressed structures described in [21]. It will directly impact both masks and L that are crucial elements in terms of storage consumption as demonstrated in Fig. 18.

Acknowledgments

The corresponding author would like to thank Gene Myers for being more than a constant source of inspiration.

References

[1] D. Arroyuelo, An improved succinct representation for dynamic k-ary trees, in: Proc. of Combinatorial Pattern Matching (CPM), vol. 5029 of Lecture Notes in Computer Science, 2008.

[2] M. Burrows, D. J. Wheeler, A block-sorting lossless data compression algorithm., Tech. Rep. 124, DEC, Palo Alto, California (1994).

[3] J. G. Cleary, W. J. Teahan, I. Witten, Unbounded length contexts for PPM, Comput. J. 40 (2/3) (1997) 67–76.

[4] J. G. Cleary, I. Witten, Data compression using adaptive coding and partial string matching, IEEE Trans. Commun. 32 (4) (1984) 396–402.

[5] T. H. Cormen, C. Leiserson, R. Rivest, Introduction to algorithms, MIT Press, 1990.

[6] P. Ferragina, R. Grossi, Optimal on-line search and sublinear time update in string matching, in: Proc. of Foundations of Computer Science (FOCS), 1995.

[7] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. of Foundations of Computer Science (FOCS), 2000.

[8] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representation of sequences and full-text indexes, ACM Trans. Alg. 3 (2007) article 20.

[9] P. Ferragina, G. Manzini, S. Muthukrishnan, The Burrows-Wheeler Transform (special issue), Theor. Comput. Sci. 387 (3) (2007) 197–360.

[10] W. Gerlach, Dynamic FM-Index for a collection of texts with application to space-efficient construction of the compressed suffix array, Master's thesis, Universität Bielefeld, Germany (2007).

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

[12] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.

[13] W. K. Hon, T. W. Lam, K. Sadakane, W. K. Sung, S. M. Yiu, Compressed index for dynamic text, in: Proc. of Data Compression Conference (DCC), 2004.

[14] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, J. ACM 53 (6) (2006) 918–936.

[15] D. K. Kim, J. S. Sim, H. Park, K. Park, Constructing suffix arrays in linear-time, J. Discrete Algorithms 3 (2–4) (2005) 126–142.

[16] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, J. Discrete Algorithms 3 (2–4) (2005) 143–156.

[17] V. Mäkinen, Compact suffix array – a space-efficient full-text index, Fundamenta Informaticae, Special Issue - Computing Patterns in Strings 56 (1-2) (2003) 191–210.

[18] V. Mäkinen, G. Navarro, Compressed compact suffix arrays, in: Proc. of Combinatorial Pattern Matching (CPM), vol. 3109 of Lecture Notes in Computer Science, 2004.

[19] V. Mäkinen, G. Navarro, Succinct suffix arrays based on run-length encoding, Nordic J. of Computing 12 (1) (2005) 40–66.

[20] V. Mäkinen, G. Navarro, Rank and select revisited and extended, Theor. Comput. Sci. 387 (3) (2007) 332–347.

[21] V. Mäkinen, G. Navarro, Dynamic entropy-compressed sequences and full-text indexes, ACM Trans. Alg. 4 (3) (2008) article 32.

[22] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, in: Proc. of Symposium on Discrete Algorithms (SODA), 1990.

[23] M. A. Maniscalco, S. J. Puglisi, Faster lightweight suffix array construction, in: Proc. of International Workshop On Combinatorial Algorithms (IWOCA), 2006.

[24] E. M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (2) (1976) 262–272.

[25] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comp. Surv. 39 (1) (2007) article 2.

[26] S. J. Puglisi, W. F. Smyth, A. Turpin, A taxonomy of suffix array construction algorithms, ACM Comp. Surv. 39 (2) (2007) 1–31.

[27] R. Raman, S. Rao, Succinct dynamic dictionaries and trees., in: Proc. of International Colloquium on Automata, Languages, and Programming (ICALP), 2003.

[28] L. Russo, G. Navarro, A. Oliveira, Fully-compressed suffix trees, in: Proc. of the Latin American Theoretical Informatics (LATIN), vol. 4957 of Lecture Notes in Computer Science, Springer, 2008.

[29] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607.

[30] M. Salson, T. Lecroq, M. Léonard, L. Mouchard, A four-stage algorithm for updating a Burrows-Wheeler Transform, Theor. Comput. Sci. To appear.

[31] E. Ukkonen, Constructing suffix trees on-line in linear time, in: Elsevier (ed.), Proc. of Information Processing, vol. 1, IFIP Transactions A-12, 1992.

[32] P. Weiner, Linear pattern matching algorithm, in: Proc. of Symp. on Switching and Automata Theory, 1973.

[33] S. Wong, W. Sung, L. Wong, CPS-tree: A compact partitioned suffix tree for disk-based indexing on large genome sequences, in: Proc. of International Conference on Data Engineering, 2007.