Dynamic Burrows-Wheeler Transform

Mikaël Salson[1⋆], Thierry Lecroq[1], Martine Léonard[1], and Laurent Mouchard[1,2]

[1] LITIS EA 4108, University of Rouen, 76821 Mont Saint Aignan Cedex, France
[2] Algorithm Design Group, Department of Computer Science, King's College London, Strand, London WC2R 2LS, England
Laurent.Mouchard@univ-rouen.fr

Abstract. The Burrows-Wheeler Transform is a building block for many text compression applications and self-index data structures. It reorders the letters of a text T to obtain a new text $bwt(T)$ which can be better compressed. This forward transform has been intensively studied over the years, but a major problem still remains: $bwt(T)$ has to be entirely recomputed whenever T is modified. In this article, we are considering standard edit operations (insertion, deletion, substitution of a letter or a factor) that are transforming a text T into T'. We are studying the impact of these edit operations on $bwt(T)$ and are presenting an algorithm that converts $bwt(T)$ into $bwt(T')$. Moreover, we show that we can use this algorithm for converting the suffix array of T into the suffix array of T'. Even if the theoretical worst-case time complexity is $O(|T|)$, the experiments we conducted indicate that it performs really well in practice.

1 Introduction

Data compression plays an important role in computer science. Its main goal is to reduce the normal consumption of data storage (one can easily store a large selection of books on a single USB key or CD). Nowadays, one of its main interests is to save network bandwith, enabling fast access to large distant resources, permitting the development of services such as Video On Demand or WebTV broadcasting over DSL [2]. While efficient image, video or sound compressions are traditionnally achieved using lossy algorithms, text compression only tolerates lossless algorithms, as no letter of the text should be omitted.

Some of the most popular lossless text compression tools, such as bzip, 7Z or winzip, are using a preprocessing engine that reorders the letters of the original text and eases the compression, paving the way for Run-Length Encoding, entropy encoding or Prediction by Partial Matching methods [4,3]. This preprocessor, the Burrows-Wheeler Transform [1], is a very interesting block-sorting algorithm: conceptually speaking, it is very close to the suffix array proposed in [17,12] and has been proved to be a particular case of the Gessel-Reutenauer transforms [5].
Due to its intrinsic structure and its similarity with the suffix array, it has been also used for advanced compressed index structures [8,9] that authorize approximate pattern matching, and therefore can be used by search engines.

The Burrows-Wheeler Transform of a text T of length n, $bwt(T)$, is often obtained from the fitting suffix array. Its construction is based on the construction of the suffix array, usually performed in $O(n)$-time [19]. Storing the intermediate suffix array is still one of the main technological bottlenecks, as it requires $\Omega(n \log n)$ bits, while storing $bwt(T)$ and T only require $O(n \log \sigma)$ bits, where σ is the size of the alphabet.

Even if this transform has been intensively studied over the years [10], one essential problem still remains: $bwt(T)$ has to be totally reconstructed as soon as the text T is

⋆ Funded by the French Ministry of Research – Grant 26962-2007

Mikaël Salson, Thierry Lecroq, Martine Léonard, Laurent Mouchard: Dynamic Burrows-Wheeler Transform, pp. 13–25.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1

altered. Although some authors already addressed the issue of maintaining an index for a dynamic text [6,7,14], their answer cannot be fully applied to the Burrows-Wheeler Transform.

In this article, we are considering the usual edit operations (insertion, deletion, substitution of a letter or a factor) that are transforming T into T'. We are studying their impact on $bwt(T)$ and are presenting an algorithm for converting $bwt(T)$ into $bwt(T')$. Moreover, we show that we can use this algorithm for changing the suffix array of T into the suffix array of T'.

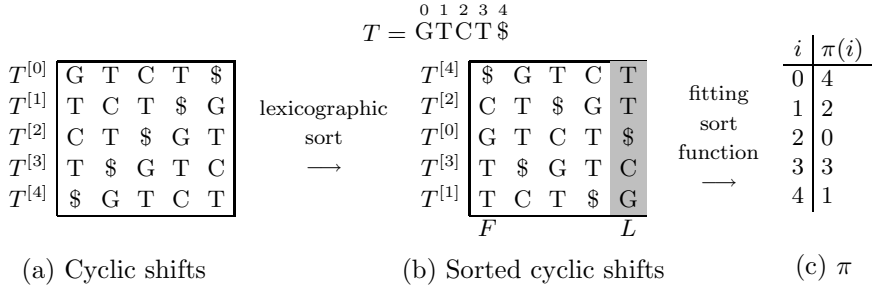
The article is organized as follows: in section 2 we introduce the Burrows-Wheeler Transform and all associated vocabulary and structures and state the formal problem we are facing. In section 3, we present a detailed explanation of the proposed algorithm when considering an insertion. We then extend the algorithm to handle the other edit operations, exhibiting their respective complexities. In section 4, we expose our results and compare them with the theoretical assumptions and finally in section 5 we conclude and draw perspectives.

2 Preliminaries

Let the text $T = T[0\,..\,n]$ be a word of length $n+1$ over a finite ordered alphabet Σ of size σ. Mimicking the suffix tree and suffix array structures, we are considering here that the rightmost letter of T is a sentinel letter \$. This letter 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. The cyclic shift of order i of the text T is $T^{[i]} = T[i\,..\,n]T[0\,..\,i-1]$ for a given $0 \leq i \leq n$.

Remark 1. $T_i = T^{[(i+1) \mod |T|]}[n]$ that will be simply denoted by $T_n^{[(i+1) \mod |T|]}$ thereafter.

The Burrows-Wheeler Transform of T, denoted $bwt(T)$, is the text of length $n+1$ corresponding to the last column L of the conceptual matrix whose rows are the lexicographically sorted $T^{[i]}$ (see Fig. 1b).Note that F, the first column of this matrix, is sorted, so can be trivially deduced from L, and that in Fig. 1c, π is the fitting sort function.



(a) Cyclic shifts (b) Sorted cyclic shifts (c) π

Figure 1. $bwt(\text{GTCT\$}) = L =$**TT\$CG**

Remark 2. We can observe that π corresponds to the suffix array of T, SA confirming the adjacency between L (letters) and SA (integers). Moreover, we simply have $L[i] = T[(SA[i]-1) \mod |T|]$, meaning we can deduce L from SA.

Combining Remarks 1 and 2, one can easily recover the original word T when considering both columns L and π. We know that: T_0=$T_4^{[1]}$, T_1=$T_4^{[2]}$, T_2=$T_4^{[3]}$, T_3=$T_4^{[4]}$ and T_4=$T_4^{[0]}$. The orders of the cyclic shifts are $(1, 2, 3, 4, 0)$ in the $\pi(i)$-column, that is $(4, 1, 3, 0, 2)$ in the i-column and finally (G, T, C, T, \$) in the L-column. We obtain T=GTCT\$.
Similarly, a right-to-left reconstruction of T will use sequence $(0, 4, 3, 2, 1)$, that is $(2, 0, 3, 1, 4)$ in the i-column and finally (\$, T, C, T, G) in the L-column. Reading this sequence from right to left, we obtain T=GTCT\$.

We clearly know how to progress in the $\pi(i)$-column, if we consider a value j in this column, its predecessor is $(j-1) \mod 5$. Starting with $j = 0$, we obtain the sequence $(0, 4, 3, 2, 1)$. We have now to study how to progress in the i-column. Considering a value j in this column, the corresponding value in $\pi(i)$-column is obviously $\pi(j)$. Its predecessor in $\pi(i)$-column is $(\pi(j)-1) \mod 5$) and finally the associated value back in the i-column is $\pi^{-1}((\pi(j)-1) \mod 5)$.

i	$\pi(i)$	$(\pi(i)-1) \mod 5$	$\pi^{-1}((\pi(i)-1) \mod 5)$
0	4	3	3
1	2	1	4
2	0	4	0
3	3	2	1
4	1	0	2

Using this formula, we obtain a permutation 0→3, 3→1, 1→4, 4→2, 2→0. We have to start with i such that $\pi(i)$ =0, that is $i = 2$, corresponding to $(2, 0, 3, 1, 4)$ in the i-column and subsequently (\$, T, C, T, G) in the L-column. Reading this sequence from right to left, we obtain T=GTCT\$.

This function is of crucial importance, since it creates a link between two consecutive elements of L or more precisely between an element of L and its equivalent in F, as described in Fig. 2. It has been shown [8] that this function, which creates a table LF of size $n+1$, can be computed using only L and the functions $rank_c(U, i)$ that return the number of c in $U[0\,..\,i]$.

i	F	L	LF
0	\$	T	3
1	C	T	4
2	G	\$	0
3	T	C	1
4	T	G	2

$$T = \overset{0\ 1\ 2\ 3\ 4}{\mathrm{GTCT\$}}$$

The second T in L ($rank_{\mathrm{T}}(L, 1)$=2) is linked to the second T in F ($rank_{\mathrm{T}}(F, 4)$=2). This specific T occurs at position 1 in the text. $LF[1]$=4 so $L[1]$=T is immediately preceded by $L[LF[1]]$=$L[4]$=G.

Figure 2. LF: Establishing a relation between L and F

Remark 3. Without the added sentinel letter \$, LF can not be necessarily determined from $bwt(T)$, *e.g.* T=AAA. It is clear that F and L would be both equal to AAA and that $rank_{\mathrm{A}}(L, i) = rank_{\mathrm{A}}(F, i)$ for all $0 \leq i < 3$, annihilating all possible relation between consecutive elements of L.

To cut a long story short, LF provides a convenient way of navigating between cyclic shifts of order i and $i-1$ and will be intensively used in this article.

We already explained that L is conceptually very close to SA, with a simple forward transform from the former to the latter. It follows that most of the algorithms constructing L are using the existing $O(n)$-time (theoretical) algorithms that build SA [19] and are applying the forward transform afterwards. Storing SA is still the main technological bottleneck, as it requires $\Omega(n \log n)$ bits while L and T only require $O(n \log \sigma)$ bits. Such a requirement prevents large texts to be encoded, even if a recent promising result [15] authorizes large texts to be processed by computing the suffix array, a block at a time.

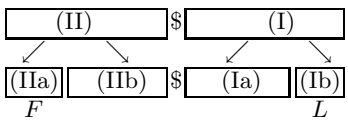
Nevertheless, L is a text that accepts no direct modification: a simple transformation of T into T' traditionally leads to the computation of its Burrows-Wheeler Transform, L', from scratch. Our goal is to study how L is affected when standard edit operations (insertion, deletion or substitution of a block of letters) are applied to T. Based on these observations, we are presenting an algorithm for transforming L into L' with only a very limited extra space and prove its correctness.

3 A Four-stage Algorithm for Updating L

We start by conducting a complete study on how an edit operation, transforming T into T', is impacting L (either directly or implicitly). To illustrate this study, we are considering the simple case consisting of the insertion of a single letter. Based on this study, we propose a four-stage algorithm for transforming L into L'. We are conducting a parallel study for F, which is required for the construction of L'. In order to do so, we are maintaining a two-column matrix gathering F and L. Each row contains the F and L values corresponding to a given cyclic shift (as described in Fig. 2). At the end of the process, L is equal to $bwt(T')$. Finally, we extend our approach to the insertion of a factor, and explain how we can consider substitutions and deletions.

In order to study the impact the insertion of a single letter has, we have first to recall that L' strongly depends on the ranking of all cyclic shifts of T'. We thus have to study how the insertion of a letter is modifying the cyclic shifts. Assume we are inserting a letter c at position i in T. Depending on the cyclic shift we are considering, we can formalize these four cases, remembering that T_n=\$, by:

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



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

Figure 3. All possible locations of c in $T'^{[j]}$ after the insertion

3.1 Cyclic Shifts of Order $j > i$ (I)

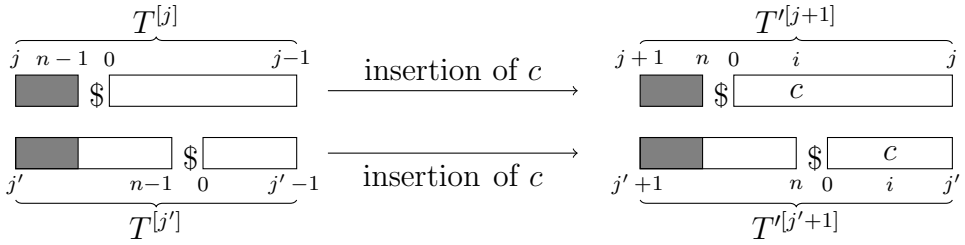
In this section, we are considering all cyclic shifts associated with positions in T that are strictly greater than i. We show that the two stages (Ia) and (Ib) are not modifying the respective ranking of the corresponding cyclic shifts.

From Fig. 3 (Ia), $T'^{[j+1]} = T[j\,..\,n-1]\$T[0\,..\,i-1]cT[i\,..\,j-1]$, $\forall j \geq i$ meaning that $T'^{[j+1]}$ and $T^{[j]}$ are sharing a common prefix $T[j\,..\,n-1]\$T[0\,..\,i-1]$.

Lemma 4. *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 $T^{[j]} < T^{[j']} \Leftarrow: T'^{[j+1]} < T'^{[j'+1]}$.*

Proof. In order to prove this lemma, we have to prove that the relative lexicographical rank of two cyclic shifts, of orders strictly greater than i is the same before and after the insertion.

Assume without loss of generality that $j > j'$ and $T^{[j]} < T^{[j']}$.
We know that for every $k < |T|$, $T^{[j]}[0\,..\,k] \leq T^{[j']}[0\,..\,k]$. The prefix of $T^{[j]}$ ending before the sentinel letter \$ is of length $n-j < |T|$, and therefore $T^{[j]}[0\,..\,n-j-1] \leq T^{[j']}[0\,..\,n-j-1]$. That is, $T[j\,..\,n-1] \leq T[j'\,..\,j'+n-j-1]$ (grey rectangles below). Moreover \$, the smallest letter of Σ, occurs only once in T. The fact that $T[j+n-j]$ is equal to \$ induces $T[j'+n-j] \neq\$$, and is therefore strictly greater than \$. It follows that $T^{[j]}[0\,..\,n-j] < T^{[j']}[0\,..\,n-j]$.



Since $T'[j+1\,..\,n]\$ = T[j\,..\,n-1]\$$ and $T'[j'+1\,..\,n+j'-j+1] = T[j'\,..\,n+j'-j]$, we have $T'[j+1\,..\,n]\$ < T'[j'+1\,..\,n+j'-j+1]\$$. So $T'[j+1\,..\,n]\$u < T'[j'+1\,..\,n+j'-j+1]v$, for all texts u, v over Σ. Finally, $T^{[j]} < T^{[j']} : T'^{[j+1]} < T'^{[j'+1]}$.

The proof of $T'^{[j+1]} < T'^{[j'+1]} : T^{[j]} < T^{[j']}$ is done in a similar way.

Remark 5. This lemma can be generalized to the insertion of a factor of length k by considering $T'^{[j+k]} < T'^{[j'+k]}$ instead of $T'^{[j+1]} < T'^{[j'+1]}$.

Cyclic Shifts of Order $j > i+1$: (Ia) c between \$ and L It follows, from Lemma 4, that the ranking of all cyclic shifts $T'^{[j+1]}$ is identical to the ranking of all cyclic shifts $T^{[j]}$. In the rows corresponding to $T'^{[j]}$, F and L are unchanged.

Cyclic Shift of Order $i+1$: (Ib) c in $L \rightarrow$ Modification of L The respective ranking of this cyclic shift with respect to the cyclic shifts of greater order is preserved. Since c is inserted at position i, it follows that $T'^{[i+1]} = T^{[i]}c$. These two cyclic shifts are sharing a common prefix $T^{[i]}$. In the row corresponding to $T'^{[i+1]}$, F is unchanged while L, which was equal to T_{i-1}, is now equal to c.
We find the position of $T'^{[i+1]}$ by using a subsampling of π (see [9,16]) and computing k such that $\pi(k)=i$.

Insertion of **G** at position i=**2** in T

T=C T C T G C \$ $\rightarrow$ T'=C T **G** C T G C \$

(Ia): no modification.

(Ib): $T^{[i]}$ is at position k=3 ($\pi(3)$=**2**), $L[3] \leftarrow$**G**.

After stage (Ib), we have: one G in F and two Gs in L, two Ts in F and one T in L.

π	F	L		F	L
6	\$	C		\$	C
5	C	G		C	G
0	C	\$		C	\$
i= **2**	C	T	$\xrightarrow{\text{(Ib)}}$	C	**G**
4	G	T		G	T
1	T	C		T	C
3	T	C		T	C

3.2 Cyclic Shifts of Order $j \leq i$

Cyclic Shift of Order i: (IIa) c in F → Insertion of a new row After considering the cyclic shift $T'^{[i+1]}$ that ends with the added letter c, we now have to consider the brand new cyclic shift that starts with the added c, that is $T'^{[i]} = cT^{[i]} = cT[i\,..\,n-1]\$T[0\,..\,i-1]$ which ends with T_{i-1}. Since $T'^{[i+1]}$ is located at position k, $T'^{[i]}$ has to be inserted in the table at position $LF[k]$ (derived from the function $rank_c(L,k)$).

Insertion of **G** at position i=**2** in T

T=C T C T G C \$ → T'=C T **G** C T G C \$

(IIa): $T'^{[i]}$ is inserted in the table at position $LF[k]$.
For this inserted row F=c=**G** and L=T_{i-1}=**T**.
$T'^{[i+1]}$ finishes with a G which is the second G in L.
$T'^{[i]}$ begins with this G which has to be the second G in F.
After stage (IIa), we have: two Gs in F and two Gs in L, two Ts in F and two Ts in L.

F	L		F	L
\$	C		\$	C
C	G		C	G
C	\$		C	\$
C	**G**	(IIa) →	C	G
G	T		G	T
T	C		**G**	**T**
T	C		T	C
			T	C

Cyclic Shifts of Order $j < i$: (IIb) c between F and \$ → Reordering So far, the L-value of one row has been updated (Ib) and one new row has been inserted (IIa). However, cyclic shifts $T'^{[j]}$, for any $j < i$, may have a different lexicographical rank than $T^{[j]}$ (e.g. AAG\$ < AG\$A but ATAG\$ > AG\$AT). Consequently, some rows corresponding to those cyclic shifts may be moved.

To know which rows have to move, we compare the position of $T^{[j]}$ with the computed position of $T'^{[j]}$, from $j = i-1$ downto 0, until these two positions are equal. The position of $T^{[j]}$ is obtained from $T^{[j+1]}$ and the LF-table we updated while considering $T^{[j+1]}$ (UpdateLF in the algorithm). The position of $T'^{[j]}$ is obtained from $T'^{[j+1]}$ and the current LF-table.
When these two positions are different, the row corresponding to $T^{[j]}$ is moved to the computed position of $T'^{[j]}$ (MoveRow in the algorithm).

We give the pseudocode of the reordering step. The function *index* returns the position of a cyclic shift in the matrix.

```
Reorder(L, i)
1  j ← index(T^[i−1])          ▷ Gives the position of T^[i−1]
2  j' ← LF[index(T'^[i])]      ▷ Gives the computed position of T'^[i−1]
3  while j ≠ j' do
4        new_j ← LF[j]
5        MoveRow(j, j')
6        UpdateLF(j', new_j)
7        j ← new_j
8        j' ← LF[j']
```

We now prove that the algorithm Reorder is correct: it ends as soon as all the cyclic shifts of T' are sorted. In the following lemma, we denote by C a succinct representation of F. Since the letters of the text are lexicographically sorted in F, we only need to store the number of times each letter appears in the text. Thus, $C[c]$ is defined as the number of letters in the text strictly lower than c, e.g. when F = \$AAACCGGT, $C[\$] = 0$ and $C[\mathrm{G}] = 6$.

Lemma 6. *$\forall j < i$, $\forall j' > j$, $T'^{[j]} < T'^{[j']}$ $\Leftarrow$: $index(T'^{[j]}) < index(T'^{[j']})$, after the iteration considering $T^{[j]}$, in* Reorder.

Proof. We prove the lemma recursively for any $j \leq i+1$.

From the previous lemma, $\forall j' \geq i+1$ we have $T'^{[i+1]} < T'^{[j']} \Leftarrow: T^{[i]} < T^{[j'-1]}$. Obviously, the property we want to prove is true for any j, on the text T and the original BWT. Thus $T'^{[i+1]} < T'^{[j']} \Leftarrow: index(T^{[i]}) < index(T^{[j'-1]})$. Neither $T'^{[i+1]}$ nor $T'^{[j']}$ have been moved in the algorithm. Thus, $index(T'^{[i+1]}) < index(T'^{[j']}) \Leftarrow: index(T^{[i]}) < index(T^{[j'-1]}) \Leftarrow: T'^{[i+1]} < T'^{[j']}$.

We have shown that the lemma is true for $j = i+1$, now let us prove it recursively for $j-1$.
By definition, $T_0'^{[j-1]} = T_{n+1}'^{[j]}$, let $r = rank_{T'^{[j]}_{n+1}}(L, index(T'^{[j]}))$. The index of $T'^{[j-1]}$ is computed using LF with the following formula:
$index(T'^{[j-1]}) = C[T_0'^{[j-1]}] + r - 1$. We distinguish two different cases:

- if the first letter of $T'^{[j-1]}$ is different from the first one of $T'^{[j']}$, then $C[T_0'^{[j-1]}] \neq C[T_0'^{[j']}]$. Without loss of generality, consider $T_0'^{[j-1]} < T_0'^{[j']}$. By definition, $r \leq C[T_0'^{[j']}] - C[T_0'^{[j-1]}]$. Thus $C[T_0'^{[j-1]}] + r - 1 \leq C[T_0'^{[j']}] - 1$. However, the $rank$ computed for the index of $T'^{[j']}$ is strictly positive. Finally $T_0'^{[j-1]} < T_0'^{[j']}$:$index(T'^{[j-1]}) < index(T'^{[j']})$.
- otherwise, both letters are equal. Then, we can write $T'^{[j-1]} < T'^{[j']} \Leftarrow: T'^{[j-1]}[1\,..\,n+1] < T'^{[j']}[1\,..\,n+1] \Leftarrow: T'^{[j-1]}[1\,..\,n+1]T_0'^{[j-1]} < T'^{[j']}[1\,..\,n+1]T_0'^{[j']} \Leftarrow: T'^{[j]} < T'^{[j'+1]}$. We know that the lemma is true for j, thus we have $T'^{[j]} < T'^{[j'+1]} \Leftarrow: index(T'^{[j]}) < index(T'^{[j'+1]})$.
  Let $k = index(T'^{[j]})$, $k' = index(T'^{[j'+1]})$, $r' = rank_{T'^{[j'+1]}_{n-1}}(L, k')$ and $c = T_0'^{[j-1]} = T_0'^{[j']}$.
  $$\begin{aligned} index(T'^{[j-1]}) &= C[c] + rank_c(L, k) - 1 \\ index(T'^{[j']}) &= C[c] + rank_c(L, k') - 1 \end{aligned}$$
  We know that $T_{n+1}'^{[j]} = L_k = c$, $T_{n+1}'^{[j'+1]} = L_{k'} = c$ and $k' > k$. So $rank_c(L, k') > rank_c(L, k)$ and eventually $index(T'^{[j-1]}) < index(T'^{[j']})$.
  Finally, $T'^{[j-1]} < T'^{[j']}$:$index(T'^{[j-1]}) < index(T'^{[j']})$. We can prove $T'^{[j-1]} < T'^{[j']} \Longleftarrow index(T'^{[j-1]}) < index(T'^{[j']})$ in a similar way.

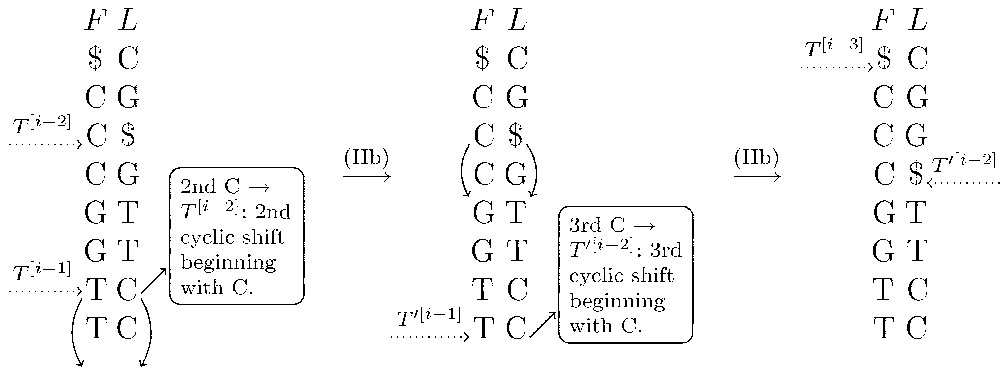
Thus, if the property is true for j, it is also true for $j-1$. Finally, when the algorithm finishes (with $j = 0$), we have $\forall j, j', \quad T'^{[j]} < T'^{[j']} \Leftarrow: index(T'^{[j]}) < index(T'^{[j']})$. In other words, at the end of the algorithm, the cyclic shifts are ordered.

We now have to prove that stopping the algorithm when the computed position and the initial one are identical is sufficient, all cyclic shifts being ordered.

Lemma 7. *$index(T^{[k]}) = index(T'^{[k]})$:$index(T^{[j]}) = index(T'^{[j]})$, for $j < k < i$.*

Proof. Given $index(T^{[k]})$,
$$\begin{aligned} index(T^{[k-1]}) &= C[T_n^{[k]}] + rank_{T_n^{[k]}}(L, index(T^{[k]})) \\ &= C[T_{n+1}'^{[k]}] + rank_{T'^{[k]}_{n+1}}(L, index(T'^{[k]})) = index(T'^{[k-1]}) \end{aligned}$$
Therefore, $index(T^{[k]}) = index(T'^{[k]})$:$index(T^{[k-1]}) = index(T'^{[k-1]})$.
By induction, we prove the property for each $j < k$.

Consider a cyclic shift $T^{[j]}$ and k the number of times $T_n^{[j]}$ appears in L from the beginning to the position of $T^{[j]}$. The LF-value for the cyclic shift $T^{[j]}$ is the position corresponding to $T^{[j-1]}$ in L which is the k-th cyclic shift beginning with a $T_n^{[j]}$.



At the position of $T'^{[i-2]}$, we have the first \$ in L, and at the position of $T^{[i-3]}$, we have the first \$ in F. Therefore, we do not need to move a cyclic shift anymore. In fact, we reach the leftmost position of the text, preventing us from considering further move.
Finally, $L = bwt(T')$.

3.3 Insertion of a Factor rather than a Single Letter

We can generalize our approach to handle the insertion of a factor S at position i in T. Consider $T' = T[0\,..\,i-1]S[0\,..\,m-1]T[i\,..\,n]$ with $m > 1$.
The four stages can be extended as follows:

(Ia) Cyclic shifts $T'^{[j]}$ with $j > i + m$: unchanged.
(Ib) Cyclic shift $T'^{[i+m]}$: modification $L{=}S_{m-1}$ instead of T_{i-1}.
(IIa) Cyclic shifts $T'^{[j]}$ from $j{=}i + m - 1$ downto $i + 1$:
insertion $F{=}S_{j-i}$ and $L{=}S_{j-i-1}$.
$T'^{[i]}$: **insertion** $F{=}S_0$ and $L{=}T_{i-1}$.
(IIb) Cyclic shifts $T'^{[j]}$ with $j < i$: as presented in algorithm on page 18.

However a problem arises: we delete T_{i-1} from L during stage (Ib), and reintroduce it after all the other insertions at the end of stage (IIa). During this stage, all $rank_{T_{i-1}}$ values that have been computed before the final insertion may be wrong. These values have to be computed only if a S_j, $j > 0$, is such that $S_j = T_{i-1}$.

A simple solution consists in not relying on $rank_{T_{i-1}}$ and, depending on the location we are considering and the location of the original T_{i-1}, adding 1 to the obtained value.

More precisely, if we are computing $LF(\ell)$ such that $L[\ell] = T_{i-1}$ and $\ell > \pi^{-1}(i)$, then we must add one to the result of $LF(i)$ (see Fig. 4).

3.4 Deletion of a Factor

Consider a deletion of m consecutive letters in T, starting at position i. The resulting text is $T' = T[0\,..\,i-1]T[i+m\,..\,n]$. The four stages can be modified as follows:

(Ia) Cyclic shifts $T'^{[j]}$ with $j > i + m$: unchanged.
(Ib) Cyclic shift $T^{[i+m]}$: modification $L{=}T_{i-1}$ instead of T_{i+m-1}.
(IIa) Cyclic shifts $T^{[j]}$ from $j{=}i + m - 1$ downto i:
deletion of the corresponding row.
We still have to pay attention to $rank_{T_{i-1}}$: during the deletion of cyclic shifts, T_{i-1} appears twice in L. Therefore, we may have to subtract one from the value returned by $rank_{T_{i-1}}$.
(IIb) Cyclic shifts $T'^{[j]}$ with $j < i$: as presented in algorithm page 18.

π	F	L		F	L	i
6	\$	C		\$	C	0
5	C	G		C	G	1
0	C	\$		C	\$	2
2	C	T		C	T	3
4	G	T		G	T	4
1	T	**C**	$\xrightarrow{\text{(Ib)}}$	T	**G**	5
3	T	C		T	C	6

Assume we are having an insertion at position 1 which causes such a modification in L.
During step (Ib) a disequilibrium is introduced between L and F (two G in L, one G in F and two C in L, three C in F).

Computing LF at position 6 gives position 2 (ie. the position of the second C in F). However it should be position 3: $\pi(6) = 3$ and $\pi(3) = \pi(6) - 1 = 2$. To correct this, we have to remember, until we insert back the original C, that at position $p = 5$ we had a C.
Using the solution we proposed, since $L[6] = \text{C}$ and $6 > \pi^{-1}(1) = 5$, we must add one to the original LFvalue obtained and finally the value is correct (that is 3).

Figure 4. Example of the problem induced by the insertion of a factor.

3.5 Substitution of a Factor

Consider the substitution of $T[i\,..\,i+m-1]$ by $S[0\,..\,m-1]$: that is T'=$T[0\,..\,i-1]S[0\,..\,m-1]T[i+m\,..\,n]$.

(Ia) Cyclic shifts $T'^{[j]}$ with $j > i+m$: unchanged.
(Ib) Cyclic shift $T'^{[i+m]}$: modification L=S_{m-1} instead of T_{i+m-1}.
(IIa) Cyclic shifts $T'^{[j]}$ from j=$i+m-1$ downto $i+1$:
substitution F=S_{j-i} and L=S_{j-i-1}
move this row to the appropriate position.
$T'^{[i]}$: modification F=S_0.
(IIb) Cyclic shifts $T'^{[j]}$ with $j < i$: as presented in algorithm on page 18.

3.6 Complexity

After the three first stages, a modification and an insertion have modified the two columns. The fourth stage, that consists in finding the new ranking of all extended cyclic shifts of order less than i, is the greediest part of the algorithm. The worst-case scenario occurs when the new ranking is obtained after each cyclic shift has been considered (e.g. $\text{A}^m\$ \rightarrow \text{A}^m\text{C}\$$). It follows that the worst-time complexity depends on the $O(n)$ iterations presented in the algorithm on page 18.

Since we are dealing with insertions and deletions, we cannot use constant-time static structures in the functions MoveRow and UpdateLF. Very recent dynamic data structures can handle insertions and deletions while allowing to perform $rank_c$, insertions and deletions in logarithmic time [16,13], leading to an overall practical complexity bounded by $O(n \log n \log \sigma)$.

These structures [16,13] can store any text in $nH_0 + o(n \log \sigma)$ bits. However Mäkinen and Navarro proved [16] that storing the BWT with such structures needs only $nH_k + o(n \log \sigma)$ bits, where H_k corresponds to the k-th order entropy of the text. C is represented in little space using $O(\sigma \log n)$ bits. Our algorithm by itself needs only constant space consisting in few variables which store values that have been replaced.

4 Experiments and Results

In the previous section, we presented a four-stage algorithm for updating the Burrows-Wheeler Transform of a modified text. We conducted experiments on real-life texts as follows: we downloaded four texts from the Pizza&Chili corpus[1] on March, 15th 2008. We added two other type of texts: a random text drawn on an alphabet of size 100 and a Fibonacci word. These texts are of various types (length, content, entropy and alphabet size). For each category, we extracted randomly 10 texts of length 100, 250 and 500 KB, and 1 MB. For each text T, the letter at a random position i was replaced by another letter c drawn from T, resulting in T'. Because of the closeness between the Burrows-Wheeler Transform and the suffix array, we generated, for each sample, the two suffix arrays, one for T and one for T'. We measured the number of differences between these two suffix arrays and repeated this operation 100 times to compute an average value. We used substitution, instead of insertion, in these tests because the number of modifications is much easier to compute: with an insertion at position i, the suffix beginning at position $j > i$ in T begins at position $j + 1$ in T'. Thus, all values greater than i in the original suffix array are incremented by one in the modified suffix array. Note that the impact an insertion or a deletion has on the lexicographical order of suffixes (or cyclic shifts) is not different from the impact of a substitution.

The results are presented in Table 1.

	Entropy H_0	100 KB	250 KB	500 KB	1 MB	Ratio 1 MB:100 KB
DNA	1.982	10.12	9.52	10.26	10.91	1.08
English	4.53	7.75	7.94	9.03	10.31	1.33
Fibo	0.96	25,414.13	63,527.09	119,780.37	261,910.49	10.31
Random	6.60	3.89	4.03	4.21	4.36	1.12
Source	5.54	92.88	55.76	118.54	72.22	0.77
XML	5.23	26.43	28.84	34.8	44.08	1.67

Table 1. Number of modifications for a random substitution of a single letter.

These results are encouraging since multiplying the size of the text by 10 does not increase by the same factor the number of differences (apart from Fibonacci). Moreover, the number of modifications is closer to $\log(n)$ rather than n. We would like to conduct an in-depth study of these experiments to examine the impact of the size of the alphabet, the entropy and other possible factors that are impacting the update.

Using dynamic structures implemented by Gerlach [11], we compare the time needed for running our update algorithm to a total reconstruction of the Burrows-Wheeler Transform (with both static and dynamic structures). The computation of the Burrows-Wheeler Transform using static structures is due to Maniscalco and Puglisi [18] and is one of the most time-efficient.

Due to technical restrictions of the implementation of the dynamic structures, we run the tests on different kinds of texts: DNA, random text and Fibonacci word. We are considering two types of updates with our algorithm: factor insertion (of length 500) and 500 insertions of a single letter.

[1] http://pizzachili.dcc.uchile.cl/texts.html

The tests are conducted on a machine under Linux 2.6.24 and the programs were compiled using `gcc` 4.2. The results are presented in Fig. 5. Note that in the graphs, y-axis uses a logarithmic scale.

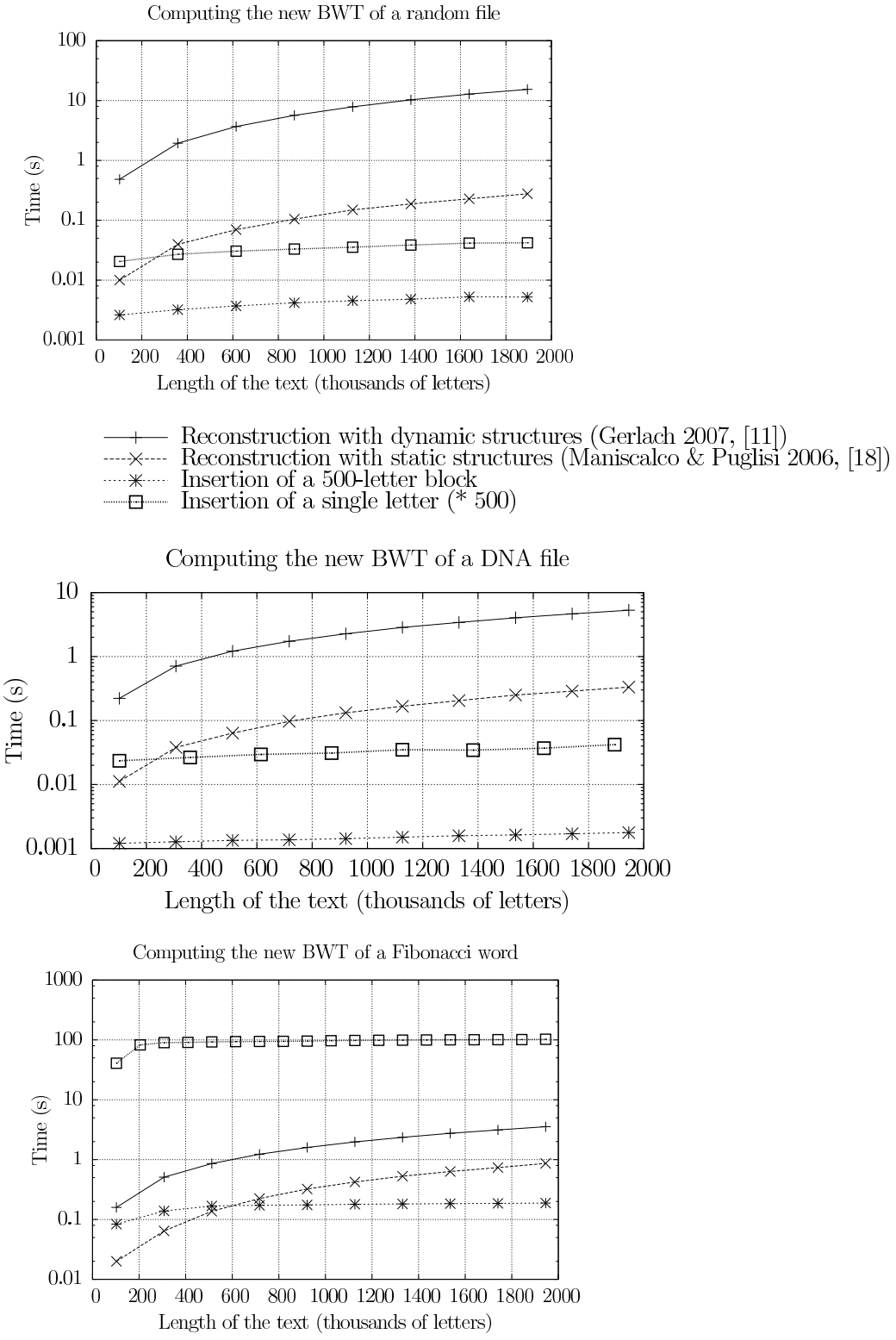


Figure 5. Time for updating and reconstructing the Burrows-Wheeler Transform.

We note that the insertion of a factor outperforms Maniscalco and Puglisi's very efficient algorithm. For the Fibonacci word, as soon as the text is long enough, our algorithm is still more efficient for the insertion of a factor although the number of iterations in step (IIb) is very high (see Table 1). However, due to the very particular structure of a Fibonacci word, one insertion of a single letter is as costly as the insertion of a 500-letter block, which explains the upper curve for Fibonacci. Note also that the reconstruction using dynamic structures is about 10 times slower than the static reconstruction, and thus our implementation may suffer from the slowdown induced by the dynamic structures.

5 Conclusions and Perspectives

We proposed an algorithm of theoretical worst-case time complexity $O(|T|)$ that modifies the Burrows-Wheeler Transform of a text T whenever standard edit operations are modifying T. The correctness of this algorithm has been proved and its efficiency in practice has been demonstrated: we selected various texts, edited randomly these texts and, with respect to the results, we confirmed that we are far from the worst-case bound. Yet, determining precisely the average-case bound of our algorithm still needs some extra work.

Moreover, this algorithm can be adapted for updating a suffix array. From a suffix array, we deduce the corresponding L, update it and retrieve the updated suffix array. Here is a pseudocode for retrieving the suffix array SA from L:

RetrieveSA(L)
1 $j \leftarrow index(L, T^{[n]})$
2 $i \leftarrow 0$
3 **repeat** $SA[j] \leftarrow i$
4 $j \leftarrow LF[j]$
5 $i \leftarrow (i-1) \mod (n+1)$
6 **until** $i = 0$

From the practical viewpoint, the dynamic structures that need to be maintained during the conversions are slowing down the process, losing the fight against "from scratch" SA constructions. Nevertheless, as far as we know, this is the first method for updating a suffix array rather than reconstructing it from scratch.

Our plan is now to adapt our strategy for updating directly a suffix array without using intermediate Burrows-Wheeler Transforms.

The algorithm we developed is also of interest for compressed indexes. Structures that are based on the Burrows-Wheeler Transform, such as FM-index, can be maintained in a way that is very similar to the one we developed for the transform, paving the way for the first fully-dynamic compressed full-text index.

References

1. M. Burrows and D. J. Wheeler: *A block-sorting lossless data compression algorithm.*, Tech. Rep. 124, DEC, Palo Alto, California, 1994.
2. C. C. Chen, C. Lee, and C. H. Ke: *Compression-based broadcast strategies in wireless information systems*, in Proc. of Advanced Information Networking and Applications (AINA), 2003, pp. 13–18.
3. J. G. Cleary, W. J. Teahan, and I. Witten: *Unbounded length contexts for PPM.* Comput. J., 40(2/3) 1997, pp. 67–76.

4. J. G. Cleary and I. Witten: *Data compression using adaptive coding and partial string matching.* IEEE Trans. Commun., 32(4) 1984, pp. 396–402.
5. M. Crochemore, J. Désarménien, and D. Perrin: *A note on the Burrows-Wheeler transformation.* Theor. Comput. Sci., 332(1-3) 2005, pp. 567–572.
6. P. Ferragina and R. Grossi: *Fast incremental text editing*, in Proc. of Symposium on Discrete Algorithms (SODA), 1995, pp. 531–540.
7. P. Ferragina and R. Grossi: *Optimal on-line search and sublinear time update in string matching*, in Proc. of Foundations of Computer Science (FOCS), 1995, pp. 604–612.
8. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in Proc. of Foundations of Computer Science (FOCS), 2000, pp. 390–398.
9. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro: *Compressed representation of sequences and full-text indexes.* ACM Trans. Alg., 3 2007, p. article 20.
10. P. Ferragina, G. Manzini, and S. Muthukrishnan: *The Burrows-Wheeler Transform (special issue).* Theor. Comput. Sci., 387(3) 2007, pp. 197–360.
11. 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.
12. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider: *New indices for text: Pat trees and pat arrays.* Information Retrieval: Data Structures & Algorithms, 1992, pp. 66–82.
13. R. González and 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, pp. 374–386.
14. W. K. Hon, T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu: *Compressed index for dynamic text*, in Proc. of Data Compression Conference (DCC), 2004, pp. 102–111.
15. J. Kärkkäinen: *Fast BWT in small space by blockwise suffix sorting.* Theor. Comput. Sci., 387(3) 2007, pp. 249–257.
16. V. Mäkinen and G. Navarro: *Dynamic entropy-compressed sequences and full-text indexes.* ACM Trans. Alg., 2008, p. , To appear.
17. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches*, in Proc. of Symposium on Discrete Algorithms (SODA), 1990, pp. 319–327.
18. M. A. Maniscalco and S. J. Puglisi: *Faster lightweight suffix array construction*, in Proc. of International Workshop On Combinatorial Algorithms (IWOCA), 2006, pp. 16–29.
19. S. J. Puglisi, W. F. Smyth, and A. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Comp. Surv., 39(2) 2007, pp. 1–31.