A Four-Stage Algorithm for Updating a Burrows-Wheeler Transform

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

We present a four-stage algorithm that updates the Burrows-Wheeler Transform of a text T, when this text is modified. The Burrows-Wheeler Transform is used by many text compression applications and some self-index data structures. It operates by reordering the letters of a text T to obtain a new text $bwt(T)$ which can be better compressed.
Even if recent advances are offering this structure new applications, a major bottleneck still exists: $bwt(T)$ has to be entirely reconstructed from scratch whenever T is modified. We are studying how standard edit operations (insertion, deletion, substitution of a letter or a factor) that are transforming a text T into T' are impacting $bwt(T)$. Then we are presenting an algorithm that directly converts $bwt(T)$ into $bwt(T')$. Based on this algorithm, we also sketch a method for converting the suffix array of T into the suffix array of T'.
We finally show, based on the experiments we conducted, that this algorithm, whose worst-case time complexity is $O(|T| \log |T| (1 + \log \sigma / \log \log |T|)$, performs really well in practice and replaces advantageously the traditional approach.

Key words: Burrows-Wheeler Transform, Compression, Dynamic, Suffix Array, Edit Operations, Algorithm Design, Self-index Data Structures

1. Introduction

The Burrows-Wheeler Transform (BWT for short) [1] is a very interesting block-sorting algorithm that reorders the letters of a text T for easing its compression. It is used as a preprocessor by some of the most popular lossless text compression tools (such as bzip) that chain it to Run-Length Encoding, entropy encoding or Prediction by Partial Matching methods [2, 3].

Conceptually speaking, the text that is produced by the BWT is very close to the suffix array proposed in [4, 5]. Crochemore *et al.* [6] proved that this transform is a particular case of a larger family of transforms, namely the Gessel-Reutenauer transforms. Due to its intrinsic structure and its similarity with the suffix array, it has been also

*Corresponding author: Laurent.Mouchard@univ-rouen.fr
[1] Funded by the French Ministry of Research - Grant 26962-2007

used for advanced compressed index structures [7, 8] that authorize approximate pattern matching, and therefore can be used by search engines.

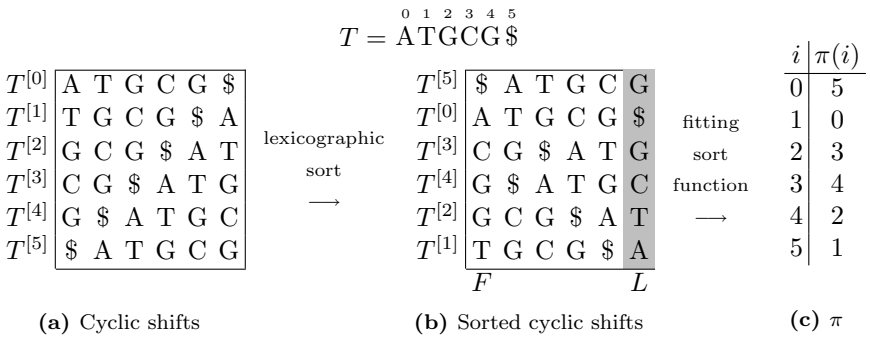
The Burrows-Wheeler Transform of a text T of length n, $bwt(T)$, is often obtained from the fitting suffix array. Its construction is based on the construction of the suffix array, usually performed in $O(n)$-time [9]. Storing the intermediate suffix array is still one of the main technological bottlenecks, as it requires $\Omega(n \log n)$ bits while storing $bwt(T)$ and T only requires $O(n \log \sigma)$ 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 altered.
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 BWT, 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 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. 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. $T_i = T_n^{[(i+1) \mod |T|]}$, for $0 \leq i \leq n$.



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

Figure 1: $bwt(\text{ATGCG\$}) = L =$**G\$GCTA**

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 cyclic shifts $T^{[i]}$ (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.

Remark 2. π is exactly equal to the suffix array of T, denoted by SA, confirming the closeness between L (letters) and SA (integers). Moreover, we simply have $L[i] = T[(SA[i]-1) \mod |T|]$, meaning we have a very simple formula for deriving L from SA.

Rebuilding the text T from π and L can be done simply. First, we start with the row corresponding to $T^{[0]}$ (and therefore to i such that $\pi(i) = 0$). It necessarily contains \$=T_n in L since \$ appears at the last position of T. Then, we consider the first row, corresponding to $T^{[n]}$ (and therefore to i such that $\pi(i) = n$). It contains \$ in F and T_{n-1} in L. Finally, we are considering the sequence of all sorted cyclic shifts in decreasing order (see Remark 1). We have:

$\pi(i)$	0	5	4	3	2	1	← decreasing order
i	1	0	3	2	4	5	← associated row positions in the matrix
$L[i]$	\$	G	C	G	T	A	← corresponding letters in L

Reading the letters from right to left, we obtain: T=ATGCG\$.

Unfortunately, the Burrows-Wheeler Transform of T consists only in L. The only column we can easily deduce from L is F. In order to navigate through L, we still need a function that tells us how the rows are ranked. This function is named LF and maps a letter in L to its equivalent in F, for example, the unique \$ in L is mapped to the unique \$ in $F, \ldots$, the second G in L to the second G in F. We are giving a formal definition thereafter.

First, we consider the first row of the conceptual matrix: it corresponds to $T^{[n]}$. It necessarily contains \$ in F and T_{n-1} in L (since \$ is the smallest letter in T and F is sorted). Consequently, the row that contains $c = T_{n-1}$ in L, has to be mapped with the row corresponding to the cyclic shift starting with c\$, that is the row where c appears for the first time in F. In order to extend our right-to-left reconstruction, we need a simple mechanism for navigating from the row corresponding to $T^{[i+1]}$ to $T^{[i]}$. By definition, if $T^{[i+1]}$ has a letter c in L, then $T^{[i]}$ has also the c in F. Generally speaking, the mechanism we have to build between F and L has to map identical letters.

From Remark 1, we know that if p is the position of $T^{[i+1]}$ in the sorted cyclic shifts, then $T_i = L[p]$. In order to map corresponding letters in L and F, we therefore need a function $rank_c(U, i)$ that returns the number of c in $U[0\,..\,i]$, for any word U over Σ. Finally, given two positions p and p' such that $F[p'] = L[p] = c$, we are connecting them if and only if $rank_c(F, p') = rank_c(L, p)$.

We denote by C_T a table storing, for each letter c of the alphabet, the number of letters smaller than c in T. Since letters are lexicographically sorted in F, the number of letters smaller than c corresponds to the position where c appears for the first time in F. Finally, the position of $T^{[i]}$ is $C_T[L[p]] + rank_{L[p]}(L, p) - 1$.

This function LF, which permits to compute the position of a cyclic shift $T^{[i]}$ from the position of the cyclic shift $T^{[i+1]}$, is defined [7] as $LF(p) = C_T[L[p]] + rank_{L[p]}(L, p) - 1$.

LF is of crucial importance, since it creates a link between two consecutive elements in T as described in Fig. 2.

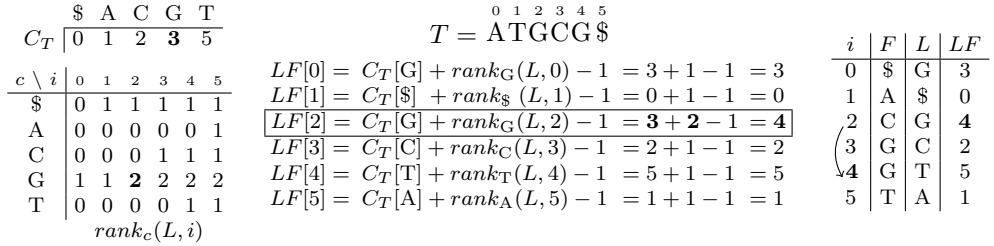


Figure 2: LF: Establishing a relation between L and F

Remark 3. Without the added sentinel letter \$, LF cannot be necessarily determined from $bwt(T)$. Let us consider T=AAA, F and L are both equal to AAA and $rank_{\mathrm{A}}(L, i) = rank_{\mathrm{A}}(F, i)$ for all $0 \leq i < 3$, annihilating all possible mapping between consecutive elements of T.

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 (see [9] for a very detailed and interesting review on that subject) 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, such as complete genome sequences, to be encoded, even if a recent promising result [11] 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 the Burrows-Wheeler Transform of T' 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.

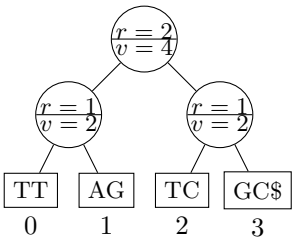
Since we are interested in managing a dynamic version of L, we need dynamic structures for storing L and C_T. Partial sums are intensively used in the following to store C_T and to compute $rank_c$ in L.

A partial sum S is a structure which stores non-negative numbers and allows to query prefix sums or to update values. The following functions are defined over S :

- $sum(S, i)$ returns the sum of the $i + 1$ first numbers in S.
- $update(S, i, \delta)$ updates the $(i + 1)$-th entry of S by adding δ to it.

Any insertion, substitution or deletion, may heavily impact C_T. We minimize this impact by using a partial sum S_{C_T} which allows to recompute any number originally stored in C_T.

Let c_i be the i-th letter of Σ in lexicographical order. We define $S_{C_T}[i]$ as the number of occurrences of c_i in T. Therefore, we have $C_T[\$] = 0$ and $C_T[c_i] = sum(S_{C_T}, i - 1)$, for any $0 < i < \sigma$.
Finally, when an edit operation modifies T, we only need to modify one value of S_{C_T} for having up-to-date values of C_T.



r: number of leaves in the left subtree.
v: number of letters in the left subtree.

	Leaves			
	0	1	2	3
$S_\$$	0	0	0	1
S_A	0	1	0	0
S_C	0	0	1	1
S_G	0	1	0	1
S_T	2	0	1	0

$$U = \overset{0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}{\mathrm{TTAGTCGC\$}}$$

Computing $rank_\mathrm{T}(U, 5)$:

The letter in position 5 in U is the sixth letter of the text.
Since $v = 4$ in the root, the sixth letter is in the right subtree.
In the right subtree, $v = 2$. Thus, the sixth letter is in its left subtree (leaf 2) and the letter is the last one in this leaf.
In this leaf, there is one T before the sixth letter of the text. $sum(S_\mathrm{T}, 1) = 2$, meaning that we have 2 Ts in the first two leaves.
Finally, $rank_\mathrm{T}(U, 5) = 3$.

Partial sums $S_\$, \ldots, S_\mathrm{T}$ store the number of occurrences of each letter in each leaf. The whole structure is managed as a collection of partial sums.

Figure 3: Dynamic structure allowing operations $rank_c$, *insert*, *delete* [12].

When L is updated, we need results of *rank* functions to be updated as well. In order to do so, we can use the structure proposed by González and Navarro [12]. This structure allows insertions (operation *insert*), deletions (operation *delete*) and supports *rank*, all in $O(\log n(1 + \log \sigma / \log \log n))$ worst-case time. This is an entropy-bound structure so that L is compressed and the whole structure needs $nH_0 + o(n \log \sigma)$ bits, where H_0 is the zero-order entropy. Their structure is based on a balanced tree and a collection of dynamic partial sums. A dynamic partial sum allows insertions and deletions of values. We present an example of this structure in Fig. 3. In this example, we store the plain text in the leaves but, theoretically, the content of the leaves is compressed. Insertions and deletions of a letter are handled by updating the leaf where the modification lies as well as the nodes on the path to the root and by updating the collection of partial sums.

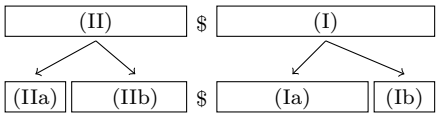
Based on these observations, we are presenting an algorithm for updating L with only a very limited extra space and prove its correctness.

3. Approach

This section can be decomposed as follows: we start with a complete study on how an edit operation, transforming T into T', is impacting the corresponding L (either directly or implicitly). To illustrate our approach, we are considering the simple case consisting of the insertion of a single letter. Based on this case, we propose a four-stage algorithm for updating L. In order to do so, we also need to update the partial sum S_{C_T}. Finally, we extend our approach to the insertion of a factor, and explain how we can extend our approach to substitutions and deletions as well.

In order to study the impact the insertion of a single letter has, we have first to recall that L will strongly depend 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 consider four cases, remembering that T_n=\$, as in Fig. 4.

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



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

Figure 4: All possible locations of c in $T'^{[j]}$ after the insertion

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

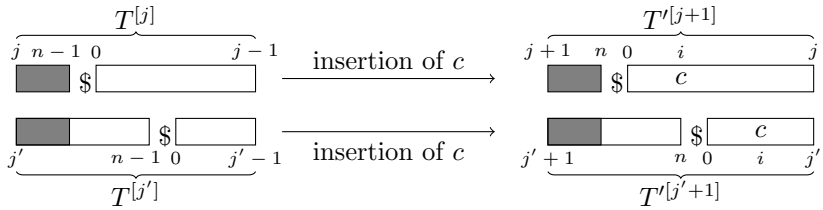
In this section, we are considering all cyclic shifts associated with positions in T that are strictly greater than i. We show that the two stages (Ia) and (Ib) are not modifying the respective ranking of the corresponding cyclic shifts.

From Fig. 4 (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 1. *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 $T^{[j]} < T^{[j']} \iff T'^{[j+1]} < T'^{[j'+1]}$.*

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 $T[j+n-j] = \$$ induces $T[j'+n-j] \neq \$$, and therefore $T[j'+n-j] > \$$. 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, $T^{[j]} < T^{[j']} \implies T'^{[j+1]} < T'^{[j'+1]}$.

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

Remark 4. This lemma can be generalized to the insertion of a factor of length k by considering $T'^{[j+k]} < T'^{[j'+k]}$ instead of $T'^{[j+1]} < T'^{[j'+1]}$.

3.1.1. Cyclic Shifts of Order $j > i+1$: (Ia) c between \$ and L

It follows, from Lemma 1, 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.

3.1.2. Cyclic Shift of Order $i+1$: (Ib) c in $L \rightarrow$ Modification of L

The ranking of this cyclic shift with respect to 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}, now equals c.

In order to perform this modification, we first have to find the position of the row corresponding to $T^{[i]}$. This is done by using a sampling of SA and the position k such that $k = SA[i]$ (see [8, 13]). The position of $T^{[i]}$ may not be sampled so we have to use the next sampled position of a cyclic shift $T^{[j]}$. The order j is such that $j > i$ and there is no position of $T^{[j']}$ sampled such that $i < j' < j$. LF allows to navigate from cyclic shift of order j to cyclic shift of order $j-1$. Therefore, using the LF function $j-i$ times, we have the position k of cyclic shift $T^{[i]}$. Finally, we can modify the corresponding letter in L: $L[k] = c$. Now the element in L at position k represents $T'^{[i+1]}$.

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

T=C T C T G C \$ (0 1 2 3 4 5 6) $\rightarrow$ T'=C T **G** C T G C \$ (0 1 2 3 4 5 6 7)

(Ia): no modification.

(Ib): $T^{[i]}$ is at position k=3 ($SA[3] = \pi(3)$=**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	(Ib) $\longrightarrow$	C	**G**
4	G	T		G	T
1	T	C		T	C
3	T	C		T	C

3.2. Cyclic Shifts of Order $j \leq i$

3.2.1. Cyclic Shift of Order i: (IIa) c in $F \rightarrow$ 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 $rank_c(L,k)$). With González and Navarro's structure, we can insert the letter c in L at position $LF(k)$ using the operation $insert(L, c, LF(k))$. The partial sum S_{C_T} has also to be updated, with the function $update(S_{C_T}, i, 1)$, if c is lexicographically the $i+1$-th letter of Σ.

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

$$\overset{0\,1\,2\,3\,4\,5\,6}{T=\mathrm{CTCTGC\$}} \rightarrow \overset{0\,1\,2\,3\,4\,5\,6\,7}{T'=\mathrm{CT\mathbf{G}CTGC\$}}$$

(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
\$	C
C	G
C	\$
C	**G**
G	T
T	C
T	C

$\xrightarrow{\text{(IIa)}}$

F	L
\$	C
C	G
C	\$
C	G
G	T
G	**T**
T	C
T	C

3.2.2. 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 a 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 we 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 the position of $T^{[j+1]}$ with the LF-value computed while considering $T^{[j+1]}$. The position of $T'^{[j]}$ is obtained from LF of the position of $T'^{[j+1]}$. 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 Reorder below).

We give the pseudocode of the reordering step. The *index* function 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(L, j, j')
6        j ← new_j
7        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.

Lemma 2. *$\forall j < i$, $\forall j' > j$, $T'^{[j]} < T'^{[j']} \iff index(T'^{[j]}) < index(T'^{[j']})$, 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 $T'^{[i+1]} < T'^{[j']} \iff T^{[i]} < T^{[j'-1]}$. Obviously, the property we want to prove is true for any j, on the text T and the corresponding $bwt(T)$. Thus $T'^{[i+1]} < T'^{[j']} \iff index(T^{[i]}) < index(T^{[j'-1]})$. Neither $T'^{[i+1]}$ nor $T'^{[j']}$ have been moved in the algorithm. Thus, $index(T'^{[i+1]}) < index(T'^{[j']}) \iff index(T^{[i]}) < index(T^{[j'-1]}) \iff T'^{[i+1]} < T'^{[j']}$.

We have shown that the lemma is true for $j = i + 1$, now let us prove it recursively for $j - 1$.

By definition, $T'^{[j-1]}_0 = T'^{[j]}_{n+1}$, let $r = rank_{T'^{[j]}_{n+1}}(L, index(T'^{[j]}))$. The index of $T'^{[j-1]}$ is

computed using the LF function with the following formula: $index(T'^{[j-1]}) = C_T[T'^{[j-1]}_0]+ r-1$. We distinguish two different cases:

- if the first letter of $T'^{[j-1]}$ is different from the first one of $T'^{[j']}$, then $C_T[T'^{[j-1]}_0] \neq C_T[T'^{[j']}_0]$. Without loss of generality, consider $T'^{[j-1]}_0 < T'^{[j']}_0$. By definition, $r \leq C_T[T'^{[j']}_0] - C_T[T'^{[j-1]}_0]$. Thus $C_T[T'^{[j-1]}_0] + r - 1 \leq C_T[T'^{[j']}_0] - 1$. However, the *rank* computed for the index of $T'^{[j']}$ is strictly positive. Finally $T'^{[j-1]}_0 < T'^{[j']}_0 \implies index(T'^{[j-1]}) < index(T'^{[j']})$.

- otherwise, both letters are equal. Then, we can write

$$\begin{aligned} T'^{[j-1]} < T'^{[j']} &\iff T'^{[j-1]}[1..n+1] < T'^{[j']}[1..n+1] \\ &\iff T'^{[j-1]}[1..n+1]T'^{[j-1]}_0 < T'^{[j']}[1..n+1]T'^{[j']}_0 \\ &\iff T'^{[j]} < T'^{[j'+1]} \end{aligned}$$

  We know that the lemma is true for j, thus we have $T'^{[j]} < T'^{[j'+1]} \iff index(T'^{[j]}) < index(T'^{[j'+1]})$.
  Let $k = index(T'^{[j]})$, $k' = index(T'^{[j'+1]})$, $r' = rank_{T'^{[j'+1]}_{n-1}}(L, k')$ and $c = T'^{[j-1]}_0 = T'^{[j']}_0$.

$$\begin{aligned} index(T'^{[j-1]}) &= C_T[c] + rank_c(L,k) - 1 \\ index(T'^{[j']}) &= C_T[c] + rank_c(L,k') - 1 \end{aligned}$$

  We know that $T'^{[j]}_{n+1} = L_k = c$, $T'^{[j'+1]}_{n+1} = L_{k'} = c$ and $k' > k$. So $rank_c(L,k') > rank_c(L,k)$ and eventually $index(T'^{[j-1]}) < index(T'^{[j']})$.
  Finally, $T'^{[j-1]} < T'^{[j']} \implies index(T'^{[j-1]}) < index(T'^{[j']})$. We can prove $T'^{[j-1]} < T'^{[j']} \impliedby index(T'^{[j-1]}) < index(T'^{[j']})$ 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 $\forall j, j'\ T'^{[j]} < T'^{[j']} \iff index(T'^{[j]}) < index(T'^{[j']})$. 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 3. *$index(T^{[k]}) = index(T'^{[k]}) \implies index(T^{[j]}) = index(T'^{[j]})$, for $j < k < i$.*

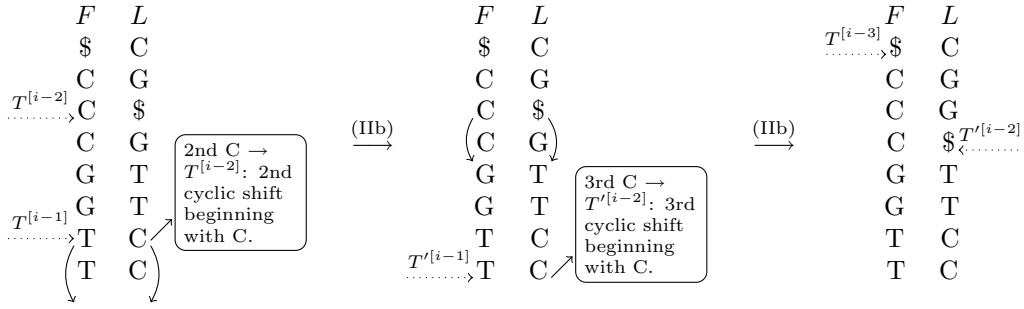
Proof. Given $index(T^{[k]})$,

$$\begin{aligned} index(T^{[k-1]}) &= C_T[T^{[k]}_n] + rank_{T^{[k]}_n}(L, index(T^{[k]})) \\ &= C_T[T'^{[k]}_{n+1}] + rank_{T'^{[k]}_{n+1}}(L, index(T'^{[k]})) = index(T'^{[k-1]}) \end{aligned}$$

Therefore, $index(T^{[k]}) = index(T'^{[k]}) \implies index(T^{[k-1]}) = index(T'^{[k-1]})$.
By induction, we prove the property for each $j < k$.

Consider a cyclic shift $T^{[j]}$ and k, the rank of $T^{[j]}_n$ in L at 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 reached 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 of length m at position i in T. Let 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 $i + 1$:
$T'^{[i]}$: **insertion** $F{=}S_{j-i}$ and $L{=}S_{j-i-1}$.
insertion $F{=}S_0$ and $L{=}T_{i-1}$.
(IIb) Cyclic shifts $T'^{[j]}$ with $j < i$: as presented in algorithm Reorder on page 8.

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 totally relying on $rank_{T_{i-1}}$ and, depending on the location we consider and the location of the original T_{i-1}, adding 1 to the obtained value.

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 Reorder on page 8.

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 REORDER on page 8.

3.6. Complexity

After the three first stages, a modification and an insertion have modified the two columns L and F. 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 $\mathrm{A}^m\$$ transformed in $\mathrm{A}^m\mathrm{C}\$$). It follows that the worst-time complexity depends on the $O(n)$ iterations presented in the algorithm page 8. Function MOVEROW operates on the dynamic structure storing L. It can be performed in at most $O(\log n(1+\log\sigma/\log\log n))$, leading to an overall practical complexity bounded by $O(n\log n(1+\log\sigma/\log\log n))$.

4. Results

In the previous section, we presented a four-stage algorithm for updating the Burrows-Wheeler Transform of a modified text. We are now conducting experiments on various types of sequences (length, type, alphabet size, entropy) and are comparing our approach with the most efficient static and dynamic existing implementations.

4.1. Material and methods

We conducted experiments on real-life texts as follows: we downloaded four texts from the Pizza&Chili corpus[1] on March, 15th 2008. 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 BWT and the suffix array, we generated, for each sample, 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.

	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
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: Average number of modifications for a random substitution of a single letter.

The results are presented in Table 1.

As shown in column ratio in the table, multiplying the size of the text by 10 does not increase by the same factor the number of differences. Moreover, the number of modifications is closer to $\log(n)$ rather than n. This observation is an explanation of the good behaviour of our algorithm in practice. 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. The first tests we conducted on an early implementation are confirming that this approach is from far faster than the quickest BWT constructions.

4.2. Comparisons

We implemented our algorithm using Gerlach's implementation [14] of Mäkinen and Navarro's dynamic structures [13]. Using these structures, $rank_c$, *insert* and *delete* operations are performed in $O(\log n \log \sigma)$ time. We compared our algorithm, allowing dynamic updates, to a reconstruction from scratch of a BWT (using dynamic or static structures). The dynamic implementation is due to Gerlach while the static one is based on one of the most time-efficient construction of the suffix array [15].

We tested our algorithm on various texts and are presenting the results for DNA and random texts. The results are very similar for the other types of text. The random text is drawn over an alphabet of size 100.

First we have inserted a single letter at a random position in texts of different lengths. Then we repeated the process for the insertion of 500 consecutive letters. Each insertion was repeated at different positions 30 times. For each considered length of the text, the insertion positions are always the same.

The programs were compiled with `gcc` version 4.1.2, and flag `-O2` (and `-DSAMPLE=0` for the implementation of BWT with dynamic structures). We achieved the tests on a computer running on Ubuntu 7.10 with an Intel Core 2, 1.83 GHz and 2 GB of RAM. In Fig. 5 we compare:

- the whole reconstruction of the BWT, due to the insertion of 500 letters in the original text (a: with dynamic structures; b: with static structures).
- our four-stage algorithm, with the insertion of a single letter. The time obtained is multiplied by 500 in order to simulate the insertion of 500 letters one at a time (c).

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

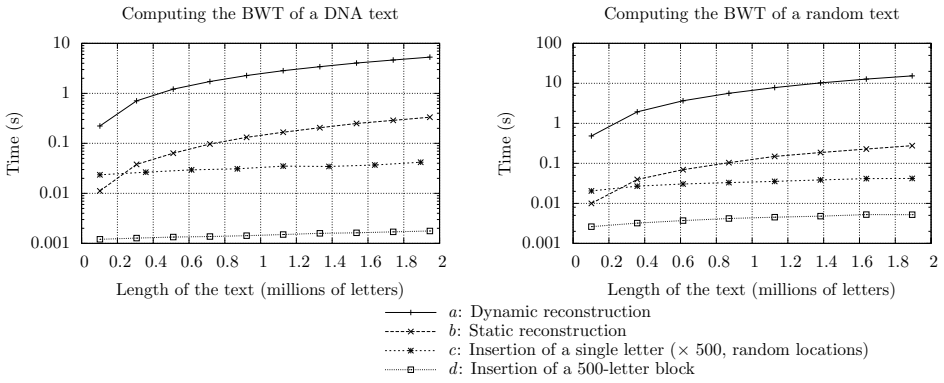


Figure 5: Comparison of the time used to construct $bwt(T')$ *vs* our dynamic update (note the logarithmic time scale)

- our four-stage algorithm with the insertion of a 500-letter block (d).

Although dynamic structures are much slower than static ones, our algorithm outperforms the static reconstruction of BWT. Moreover, dynamic-compressed structures computing *rank* operation are very recent and some further work may improve the complexity of *insert*, *delete* and *rank* operations and, thus, improve the efficiency of our implementation.

Our results show that inserting many letters at the same time is more efficient than inserting them one at a time. Consequently, we wonder how evolves the time of insertion considering different lengths of insertion. Meanwhile, we want to determine when the reconstruction of the static structure is quicker. We performed some tests on a 1 MB DNA file. We see in Fig. 6 that our algorithm is slower for insertions of more than 60 000 consecutive letters, that is 6% of the text. As we already noticed, the insertion time is not linear in the size of the word inserted for short patterns (shorter than 500 letters). For longer patterns, inserting 10 patterns of length 1 000, for example, is not worse than inserting only one pattern of length 10 000.

This is understandable since large insertions will heavily impact the lexicographical ranking of cyclic shifts and thus will lead to reorder lots of elements.

5. Conclusion

We proposed a four-stage algorithm that updates 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, despite a worst-case $O(|T| \log |T|(1 + \log \sigma / \log \log |T|)$-time complexity has been demonstrated: we selected various texts, performed random insertions and, with respect to the results, we confirmed that we are far from the mentioned worst-case bound. Yet, determining precisely the average-case bound of our algorithm still needs some extra work.

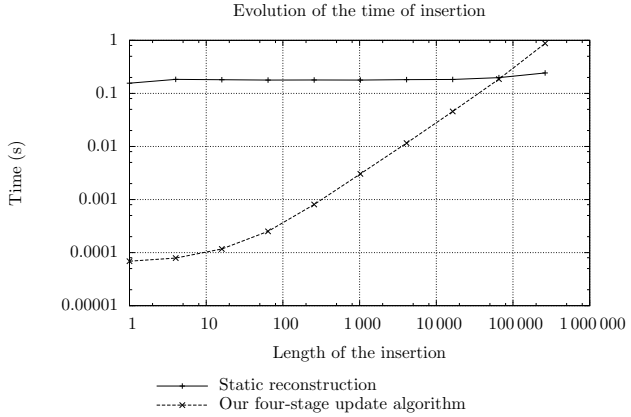


Figure 6: For which size of block should we reconstruct the BWT?

The algorithm we developed is of particular interest for data compression purposes as well as compressed indexes. Structures that are based on the Burrows-Wheeler Transform, such as FM-indexes, can be maintained in a way that is very similar to the one we developed for the transform. With little extra effort, one can probably create the first fully-dynamic compressed full-text index.

Our work may also be interesting for computing a modified suffix array without reconstructing it from scratch. Note that from the updated BWT, one can easily compute the corresponding 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 $\quad j \leftarrow LF(j)$
5 $\quad i \leftarrow (i-1) \mod (n+1)$
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.

Rather than using such a conversion for the suffix array, it is possible to update it by using a method similar to our update algorithm. Due to the equivalency between suffix sorting and cyclic shift sorting, the reordering applied to the BWT is exactly the same for the corresponding suffix array. Therefore, our plan is now to adapt our strategy for updating directly a suffix array without recomputing it entirely.

References

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

[2] J. G. Cleary, I. Witten, Data compression using adaptive coding and partial string matching, IEEE Trans. Commun. 32 (4) (1984) 396–402.
[3] J. G. Cleary, W. J. Teahan, I. Witten, Unbounded length contexts for PPM, Comput. J. 40 (2/3) (1997) 67–76.
[4] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, in: Proc. of Symposium on Discrete Algorithms (SODA), 1990, pp. 319–327.
[5] G. H. Gonnet, R. A. Baeza-Yates, T. Snider, New indices for text: Pat trees and pat arrays, Information Retrieval: Data Structures & Algorithms (1992) 66–82.
[6] M. Crochemore, J. Désarménien, D. Perrin, A note on the Burrows-Wheeler transformation, Theor. Comput. Sci. 332 (1-3) (2005) 567–572.
[7] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. of Foundations of Computer Science (FOCS), 2000, pp. 390–398.
[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] S. J. Puglisi, W. F. Smyth, A. Turpin, A taxonomy of suffix array construction algorithms, ACM Comp. Surv. 39 (2) (2007) 1–31.
[10] P. Ferragina, G. Manzini, S. Muthukrishnan, The Burrows-Wheeler Transform (special issue), Theor. Comput. Sci. 387 (3) (2007) 197–360.
[11] J. Kärkkäinen, Fast BWT in small space by blockwise suffix sorting, Theor. Comput. Sci. 387 (3) (2007) 249–257.
[12] 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, pp. 374–386.
[13] V. Mäkinen, G. Navarro, Dynamic entropy-compressed sequences and full-text indexes, ACM TALG 4 (3) (2008) article 32.
[14] 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).
[15] M. A. Maniscalco, S. J. Puglisi, Faster lightweight suffix array construction, in: Proc. of International Workshop On Combinatorial Algorithms (IWOCA), 2006, pp. 16–29.