The Alternating BWT: an algorithmic perspective[☆]

Raffaele Giancarlo[a], Giovanni Manzini[b], Antonio Restivo[a], Giovanna Rosone[c], Marinella Sciortino[a]

[a] *University of Palermo, Italy*
[b] *University of Eastern Piedmont and IIT-CNR Pisa, Italy*
[c] *University of Pisa, Italy*

Abstract

The Burrows-Wheeler Transform (BWT) is a word transformation introduced in 1994 for Data Compression. It has become a fundamental tool for designing self-indexing data structures, with important applications in several area in science and engineering. The Alternating Burrows-Wheeler Transform (ABWT) is another transformation recently introduced in [Gessel et al. 2012] and studied in the field of Combinatorics on Words. It is analogous to the BWT, except that it uses an alternating lexicographical order instead of the usual one. Building on results in [Giancarlo et al. 2018], where we have shown that BWT and ABWT are part of a larger class of reversible transformations, here we provide a combinatorial and algorithmic study of the novel transform ABWT. We establish a deep analogy between BWT and ABWT by proving they are the only ones in the above mentioned class to be rank-invertible, a novel notion guaranteeing efficient invertibility. In addition, we show that the backward-search procedure can be efficiently generalized to the ABWT; this result implies that also the ABWT can be used as a basis for efficient compressed full text indices. Finally, we prove that the ABWT can be efficiently computed by using a combination of the Difference Cover suffix sorting algorithm [Kärkkäinen et al., 2006] with a linear time algorithm for finding the minimal cyclic rotation of a word with

Email addresses: `raffaele.giancarlo@unipa.it` (Raffaele Giancarlo), `giovanni.manzini@uniupo.it` (Giovanni Manzini), `antonio.restivo@unipa.it` (Antonio Restivo), `giovanna.rosone@unipi.it` (Giovanna Rosone), `marinella.sciortino@unipa.it` (Marinella Sciortino)

Postprint version; to appear on Theoretical Computer Science

respect to the alternating lexicographical order.

Keywords: Alternating Burrows-Wheeler Transform, Rank-invertibility, Difference cover algorithm, Galois word

---

1. Introduction

Michael Burrows and David Wheeler introduced in 1994 a reversible word transformation [4], denoted by BWT, that turned out to have "myriad virtues". At the time of its introduction in the field of text compression, the Burrows-Wheeler Transform was perceived as a magic box: when used as a preprocessing step it would bring rather weak compressors to be competitive in terms of compression ratio with the best ones available [14]. In the years that followed, many studies have shown the effectiveness of BWT and its central role in the field of Data Compression due to the fact that it can be seen as a "booster" of the performance of memoryless compressors [16, 25, 39, 19]. Moreover, it was shown in [17] that the BWT can be used to efficiently search for occurrences of patterns inside the original text. Such capabilities of the BWT have originated the field of Compressed Full-text Self-indices [32, 41]. The remarkable properties of the BWT have aroused great interest both from the theoretical and applicative points of view [33, 34, 38, 51, 30, 7, 29, 47, 8, 20, 43, 13].

In the context of Combinatorics on Words, many studies have addressed the characterization of the words that become the most compressible after the application of the BWT [37, 50, 42, 44, 45, 15]. Recent studies have focused on measuring the "clustering effect" of BWT, which is a property related to its boosting role as preprocessing of a text compressor [36, 35].

In [9], the authors characterize the BWT as the inverse of a known bijection between words and multisets of primitive necklaces [22]. From this result, in [21] the authors introduce and study the basic properties of the *Alternating BWT*, $ABWT$ from now on. It is a transformation on words analogous to the BWT but the cyclic rotations of the input word are sorted by using the *alternating* lexicographic order instead of the usual lexicographic order. The alternating lexicographic order is defined for infinite words as follows: the first letters are compared with the given alphabetic order, in case of equality the second letters are compared with the opposite order, and so on alternating the two orders for even/odd positions.

In this paper we show that the $ABWT$ satisfies most of the properties that make the BWT such a useful transformation. Not only the $ABWT$ can be computed and inverted in linear time, but also the *backward-search* procedure, which is the basis for indexed exact string matching on the BWT, can

be efficiently generalized to the $ABWT$. This implies that the $ABWT$ can be used to build an efficient compressed full text index for the transformed string, similarly to the BWT. Note that the variants of the original BWT which have been introduced so far in the literature [48, 5, 10], were either simple modifications that do not bring new theoretical insight or they were significantly different but without the remarkable compression and search properties of the original BWT (see [24, Section 2.1] for a more detailed discussion of these variants).

The existence of the $ABWT$ shows that the classical lexicographic order is not the only order relation that one can use to obtain a reversible transformation. Indeed, lexicographic and alternating lexicographic order are two particular cases of a more general class of order relations considered in [11, 46]. In a preliminary version of this paper [23] we introduce therefore a class of reversible word transformations based on the above order relations that includes both the original BWT and the Alternating BWT. Within this class, we introduce the notion of *rank-invertibility*, a property that guarantees that the transformation can be efficiently inverted using rank operations, and we prove that BWT and $ABWT$ are the only transformations within this class that are rank-invertible.

We consider also the problem of efficiently computing the $ABWT$. We first show how to generalize to the alternating lexicographic order the Difference Cover technique introduced in [28]. This result leads to the design of time optimal and space efficient algorithms for the construction of the $ABWT$ in different models of computation when the input string ends with a unique end-of-string symbol. Finally, we explore some combinatorial properties of the *Galois words*, which are minimal cyclic rotations within a conjugacy class, with respect to the alternating lexicographical order. We show that the Galois rotation of a given word can be computed in linear time and, consequently, by using the Difference Cover technique, the $ABWT$ can be computed in linear time even when the input string does not end with a unique end-of-string symbol.

Motivated by the discovering of the $ABWT$, in [24] the authors explore a class of string transformations that includes the one considered in this paper. In this larger class, the cyclic rotations of the input string are sorted using an alphabet ordering that depends on the longest common prefix of the rotations being compared. Somewhat surprisingly some of the transformations in this class do have the same properties of the BWT and $ABWT$, thus showing that our understanding of these transformations is still incomplete.

2. Preliminaries

Let $\Sigma = \{c_0, c_1, \ldots, c_{\sigma-1}\}$ be an ordered constant size alphabet with $c_0 < c_1 < \ldots < c_{\sigma-1}$, where $<$ denotes the standard lexicographic order. We denote by Σ^* the set of words over Σ. Let $w = w_0 w_1 \cdots w_{n-1} \in \Sigma^*$ be a finite word, we denote by $|w|$ its length n. We use ϵ to denote the empty word. We denote by $|w|_c$ the number of occurrences of a letter c in w. The Parikh vector P_w of a word w is a σ-length array of integers such that for each $c \in \Sigma$, $P_w[c] = |w|_c$. Given a word x and $c \in \Sigma$, we write $\text{rank}_c(x, i)$ to denote the number of occurrences of c in $x[0, i]$.

Given a finite word w, a *factor* of w is written as $w[i, j] = w_i \cdots w_j$, with $0 \leq i \leq j \leq n-1$. A factor of type $w[0, j]$ is called a *prefix*, while a factor of type $w[i, n-1]$ is called a *suffix*. The longest proper factor of w that is both prefix and suffix is called *border*. The i-th symbol in w is denoted by $w[i]$. Two words $x, y \in \Sigma^*$ are *conjugate*, if $x = uv$ and $y = vu$, where $u, v \in \Sigma^*$. We also say that x is a *cyclic rotation* of y. A word x is *primitive* if all its cyclic rotations are distinct. A primitive word is a *Lyndon word* if it is smaller than all of its conjugates. Conjugacy between words is an equivalence relation over Σ^*. A word z is called a *circular factor* of x if it is a factor of some conjugate of x.

Given two words of the same length $x = x_0 x_1 \ldots x_{s-1}$ and $y = y_0 y_1 \ldots y_{s-1}$, we write $x \preceq_{lex} y$ if and only if $x = y$ or $x_i < y_i$, where i is the smallest index in which the corresponding characters of the two words differ. Analogously, and with the same notation as before, we write $x \preceq_{alt} y$ if and only if $x = y$ or (a) i is even and $x_i < y_i$ or (b) i is odd and $x_i > y_i$. Notice that $\preceq_{lex}$ is the standard lexicographic order relation on words while $\preceq_{alt}$ is the *alternating* lexicographic order relation, in which a given alphabetic order and the opposite order are alternate for even/odd position of the words. For instance, $aba \preceq_{alt} aab$ and $aab \preceq_{lex} aba$. Standard lexicographic order and alternating order are used in Section 3 to define two different transformations on words. The alternating lexicographic order is defined for infinite words as follows: the first letters are compared with the given alphabetic order, in case of equality the second letters are compared with the opposite order, and so on alternating the two orders for even/odd positions.

The *run-length encoding* of a word w, denoted by $\text{rle}(w)$, is a sequence of pairs (w_i, l_i) such that $w_i w_{i+1} \cdots w_{i+l_i-1}$ is a maximal run of a letter w_i (i.e., $w_i = w_{i+1} = \cdots = w_{i+l_i-1}$, $w_{i-1} \neq w_i$ and $w_{i+l_i} \neq w_i$), and all such maximal runs are listed in $\text{rle}(w)$ in the order they appear in w. We denote by $\rho(w) = |\text{rle}(w)|$ i.e., is the number of pairs in w, or equivalently the number of equal-letter runs in w. Moreover we denote by $\rho(w)_{c_i}$ the number of pairs (w_j, l_j) in $\text{rle}(w)$ where $w_j = c_i$. Notice that $\rho(w) \leq \rho(w_1) + \rho(w_2) + \cdots + \rho(w_p)$,

where $w_1 w_2 \cdots w_p = w$ is any partition of w.

The zero-th order empirical entropy of the word w is defined as

$$H_0(w) = -\sum_{i=0}^{\sigma-1} \frac{|w|_{c_i}}{|w|} \log \frac{|w|_{c_i}}{|w|}$$

(all logarithms are taken to the base 2 and we assume $0 \log 0 = 0$). The value $|w|H_0(w)$ is the output size of an ideal compressor that uses $-\log(|w|_{c_i}/|w|)$ bits to encode each occurrence of symbol c_i. This is the minimum size we can achieve using a uniquely decodable code in which a fixed codeword is assigned to each symbol.

For any length-k factor x of w, we denote by x_w the sequence of characters immediately preceding the occurrences of x in w, taken from left to right. If x is not a factor of w the word x_w is empty. The k-th order empirical entropy of w is defined as

$$H_k(w) = \frac{1}{|w|} \sum_{x \in \Sigma^k} |x_w| H_0(x_w).$$

The value $|w|H_k(w)$ is a lower bound to the output size of any compressor that encodes each symbol with a code that only depends on the symbol itself and on the k preceding symbols. Since the use of a longer context helps compression, it is not surprising that for any $k \geq 0$ it is $H_{k+1}(w) \leq H_k(w)$.

3. BWT and Alternating BWT

In this section we describe two different invertible transformations on words based on the lexicographic and alternating lexicographic order, respectively. Given a primitive word w of length n in Σ^*, *the Burrows-Wheeler transform*, denoted by BWT [4] and *the Alternating Burrows-Wheeler transform*, denoted by $ABWT$ [21] for w are defined constructively as follows:

1. Create the matrix $M(w)$ of the cyclic rotations of w;
2. Create the matrix
   (a) for BWT, $M_{lex}(w)$ by sorting the rows of $M(w)$ according to $\preceq_{lex}$;
   (b) for $ABWT$, $M_{alt}(w)$ by sorting the rows of $M(w)$ according to $\preceq_{alt}$;
3. Return as output the pair
   (a) for BWT, $(bwt(w), I)$, where $bwt(w)$ is the last column L in the matrix $M_{lex}(w)$
   (b) for $ABWT$, $(abwt(w), I)$ where $abwt(w)$ is the last column L in the matrix $M_{alt}(w)$

   and, in both cases, the integer I giving the position of w in that matrix.

$$
\begin{array}{c}
\\
a\ b\ r\ a\ c\ a \\
b\ r\ a\ c\ a\ a \\
r\ a\ c\ a\ a\ b \\
a\ c\ a\ a\ b\ r \\
c\ a\ a\ b\ r\ a \\
a\ a\ b\ r\ a\ c
\end{array}
\qquad
\begin{array}{cc}
 & F\ \ \ \ \ \ L \\
 & \downarrow\ \ \ \ \ \ \downarrow \\
 & a\ a\ b\ r\ a\ c \\
 & a\ b\ r\ a\ c\ a \\
I \rightarrow & a\ c\ a\ a\ b\ r \\
 & b\ r\ a\ c\ a\ a \\
 & c\ a\ a\ b\ r\ a \\
 & r\ a\ c\ a\ a\ b
\end{array}
\qquad
\begin{array}{cc}
 & F\ \ \ \ \ \ L \\
 & \downarrow\ \ \ \ \ \ \downarrow \\
I \rightarrow & a\ c\ a\ a\ b\ r \\
 & a\ b\ r\ a\ c\ a \\
 & a\ a\ b\ r\ a\ c \\
 & b\ r\ a\ c\ a\ a \\
 & c\ a\ a\ b\ r\ a \\
 & r\ a\ c\ a\ a\ b
\end{array}
$$

$M(w)$ $\qquad$ $M_{lex}(w)$ $\qquad$ $M_{alt}(w)$

Figure 1: Left: the matrix $M(w)$ of all cyclic rotations of the word $w = acaabr$. Center: the matrix $M_{lex}(w)$; the pair $(caraab, 2)$ is the output $bwt(w)$. Right: the matrix $M_{alt}(w)$; the pair $(racaab, 0)$ is the output of $ABWT(w)$.

An example of the above process, together with the corresponding output, is provided in Figure 1.

Remark 3.1. *If two words are conjugate the BWT (resp. ABWT) will have the same column L and differ only in I, whose purpose is only to distinguish between the different members of the conjugacy class. However, I is not necessary in order to recover the matrix M from the last column L.*

The following proposition, proved in [23], states that three well known properties of the BWT hold, in a slightly modified form, for the $ABWT$ as well. Here we report the proof for the sake of completeness.

Proposition 3.2. *Let w be a primitive word and let (L, I) be the output of BWT or ABWT applied to w. The following properties hold:*

1. *Let F denote the first column of $M_{lex}(w)$ (resp. $M_{alt}(w)$), then F is obtained by lexicographically sorting the symbols of L.*
2. *For every i, $0 \leq i < n$, $L[i]$ circularly precedes $F[i]$ in the original word, for both BWT and ABWT.*
3. *For each symbol a, and $1 \leq j \leq |w|_a$, the j-th occurrence of a in F corresponds*
   (a) *for BWT, to its j-th occurrence in L;*
   (b) *for ABWT, to its $(|w|_a - j + 1)$-th occurrence in L.*

Proof. Properties 1, 2 and 3a for the BWT have been established in [4]. Properties 1 and 2 for the $ABWT$ are straightforward. To prove property 3b, consider two rows i and j in $M_{alt}(w)$ with $i < j$ starting with the symbol

a. Let w_i and w_j be the two conjugates of w in rows i and j of $M_{alt}(w)$. By construction we have $w_i = au$, $w_j = av$ and $w_i \preceq_{alt} w_j$. To establish Property 3b, we need to show that row w_j cyclically rotated precedes in the $\preceq_{alt}$ order row w_i cyclically rotated. In other words, we need to show that

$$au \preceq_{alt} av \implies va \preceq_{alt} ua.$$

To prove the above implication, we notice that if the first position in which au and av differ is odd (resp. even) then the first position in which va and ua differ will be in an even (resp. odd) position. The thesis follow by the alternate use of the standard and reverse order in $\preceq_{alt}$ (see [21] for a different proof of the same property). □

It is well known that in the BWT the occurrences of the same symbol appear in columns F and L in the **same** relative order; according to Property 3b. In the $ABWT$, the occurrences in L appear in the **reverse** order than in F. For example, in Figure 1 (right) we see that the a's of $acaabr$ in the columns F appear in the order 1st, 3rd, and 2nd, while in column L they are in the reverse order 2nd, 3rd, and 1st.

Proposition 3.2 is the key motivations to efficiently recover the original string from the output of BWT or $ABWT$, as we will see in Section 5.

Note that, although BWT and $ABWT$ are very similarly defined, they are very different combinatorial tools. Combinatorial aspects that distinguish $ABWT$ and BWT can be found in [21, 23], which makes it interesting to study $ABWT$ in terms of tool characterizing families of words.

However, in [23] we experimentally tested $ABWT$ as pre-processing of a compression tool, by comparing its performance with a BWT-based compressor. We have shown that the behaviour of the two transformations is essentially equivalent in terms of compression. Actually, such experiments confirm a theoretical result we proved in [23] for a larger class of transformations that can be seen as a generalization of the BWT and that includes the $ABWT$ as a special case. In the next section, we give a brief description of the properties we proved in [23] for such a class of transformations, all of which also hold for the $ABWT$.

4. Generalized BWTs: a synopsis

In this section we describe the class of Generalized BWTs, introduced in [23], by reporting their main properties.

Given the alphabet Σ of size σ, in the following, we denote by Π_Σ the set of $\sigma!$ permutations of the alphabet symbols. Two important permutations are distinguished in Π_Σ: the identity permutation Id corresponding to the

lexicographic order, and the reverse permutation *Rev* corresponding to the reverse lexicographic order. We consider generalized lexicographic orders introduced in [46] (cf. also [11]) that, for the purposes of this paper, can be formalized as follows.

Definition 4.1. *Given a k-tuple $K = (\pi_0, \pi_1, \ldots, \pi_{k-1})$ of elements of Π_Σ, we denote by $\preceq_K$ the lexicographic order such that given two words of the same length $x = x_0x_1 \cdots x_{s-1}$ and $y = y_0y_1 \cdots y_{s-1}$ it is $x \preceq_K y$ if and only if $x = y$ or $x_i <_i y_i$ where i is the smallest index such that $x_i \neq y_i$, and $<_i$ is the lexicographic order induced by the permutation $\pi_{i \bmod k}$. Without loss of generality, we can assume $\pi_0 = Id$.*

Using the above definition, a class of generalized BWTs can be defined as follows:

Definition 4.2. *Given a k-tuple $K = (Id, \pi_1, \ldots, \pi_{k-1})$ of elements of Π_Σ, we denote by BWT_K the transformation mapping a primitive word w to the last column L of the matrix $M_K(w)$ containing the cyclic rotations of w sorted according to the lexicographic order $\preceq_K$. The output of BWT_K applied to w is the pair $(bwt_K(w), I)$, where $bwt_K(w)$ is the last column L of the matrix and I is the row of $M_K(w)$ containing the word w.*

Note that for $K = (Id)$, BWT_K is the usual BWT, while for $K = (Id, Rev)$, BWT_K coincides with the $ABWT$ defined in Section 3.

Remark 4.3. *For most applications, it is assumed that the last symbol of w is a unique end-of-string marker smaller than each symbol of the alphabet Σ. Under this assumption, lexicographically sorting w's cyclic rotations is equivalent to building the suffix tree [31, 26] for w, which can be done in linear time. In this setting, we can compute $bwt_K(w)$ in linear time: we do a depth-first visit of the suffix tree in which the children of each node are visited in the order induced by K. In other words, the children of each node v are visited according to the order $\pi_{|v| \bmod k}$ where $|v|$ is the string-depth[1] of node v. Since the suffix tree has $O(|w|)$ nodes, for a constant alphabet the whole procedure takes linear time.*

The next result proved in [23] guarantees that the transformations BWT_K are invertible. As we specify in the next section, the inversion procedure for $ABWT$ is more efficient.

[1]The number of letters in the word obtained by concatenating the labels of the edges in the path from the root of the suffix tree to the node v

Theorem 4.4. *For every k-tuple* $K = (Id, \pi_1, \ldots, \pi_{k-1})$ *the transformation* BWT_K *is invertible in* $O(n^3)$ *time, where* $n = |w|$.

Note that recently [24], the complexity for the inversion of a generic transformation in BWT_K has been improved to $\mathcal{O}(n^2)$ time.

The following theorem proved in [23] shows that each transformation BWT_K produces a number of equal-letter runs that is at most the double of the number of equal-letter runs of the input word. This fact generalizes a result proved for BWT [36].

Theorem 4.5. *Given a k-tuple* $K = (Id, \pi_1, \ldots, \pi_{k-1})$ *and a word* w *over a finite alphabet* Σ*, then*

$$\rho(bwt_K(w)) \leq 2\rho(w).$$

A key property of BWT is that it allows to reduce the problem of compressing a string w up to its r-th order entropy to the problem of compressing a collection of factors of $bwt(w^R)$ up to their 0-th order entropy, where w^R is the reverse of the word w. This means that a BWT-based compressor combining BWT with a zero order (memoryless) compressor, is able to achieve the same high order compression typical of more complex tools such as Lempel-Ziv encoders. In [23], we prove that a similar result also holds for the transformation BWT_K.

Theorem 4.6. *Let* K *be a k-tuple and* $u = bwt_K(w^R)$*, where* w^R *is the reverse of the word* w*. For each positive integer* r*, there exists a factorization of* $u = u_1 u_2 \ldots u_m$ *such that*

$$H_r(w) = \frac{1}{|u|} \sum_{i=1}^{m} |u_i| H_0(u_i).$$

More in particular, experimental results reported in [23] show that the behavior of the transformations BWT and $ABWT$ is essentially equivalent in terms of size of compressed representation.

5. Rank-invertible transformations

It is well known that the key to efficiently compute the inverse of original BWT is the existence of a easy-to-compute permutation mapping, in the matrix $M_{lex}(w)$, a row index i to the row index $LF(i)$ containing row i right-shifted by one position. This permutation is called LF-mapping since, by Proposition 3.2, $LF(i)$ is the position in the first column F of $M_{lex}(w)$ corresponding to the i-th entry in column L: in other words, $F[LF(i)]$ is the

same symbol in w as $L[i]$. Again, by Proposition 3.2 we have that $L[LF(i)]$ is the symbol preceding $L[i]$ in the input word w. Define $LF^0(x) = x$ and $LF^{j+1}(x) = LF(LF^j(x))$. If $bwt(w) = (L, I)$ with $|w| = n$, then by construction $L[I] = w_{n-1}$ and we can recover w with the formula:

$$w_{n-1-j} = L[LF^j(I)] \tag{1}$$

Note that the inversion formula (1) only depends on Properties 1 and 2 of Proposition 3.2. Since such properties hold for every generalized transformation BWT_K, (1) provides an inversion formula for every transformation in that class. In other words, inverting a generalized BWT amounts to computing n iterations of the LF-mapping.

By Property 3a in Proposition 3.2, the LF-mapping for the original BWT can be expressed using the Parikh vector P_L of L and a rank operation over L:

$$LF(i) = \sum_{c \in \Sigma}^{c<L[i]} P_L[c] \ + \ \mathrm{rank}_{L[i]}(L, i-1) \tag{2}$$

Note that $\sum_{c\in\Sigma}^{c<L[i]} P_L[c]$ is simply the total number of occurrences of symbols smaller than $L[i]$ in L, and $\mathrm{rank}_{L[i]}(L, i-1)$ is the number of occurrences of symbol $L[i]$ in among the first i symbols of L.

By Property 3b in Proposition 3.2, for the $ABWT$, the corresponding formula is:

$$LF(i) = \sum_{c \in \Sigma}^{c\leq L[i]} P_L[c] \ - \ \mathrm{rank}_{L[i]}(L, i-1) - 1 \tag{3}$$

Since the rank operation on (compressed) arrays over finite alphabet can be computed in constant time [1] and the partial sums $\sum_{c<i} P_L[c]$ can be precomputed, the computation of the LF map for both the BWT and $ABWT$ takes $O(1)$ time. This implies that, thanks to the simple structure of its LF-mapping, also the $ABWT$ can be inverted in linear time.

The computation of the LF map is the main operation also for the so-called *backward-search* procedure which makes it possible to use (a compressed version of) $bwt(w)$ as a full text index for w [18]. The following proposition is the key to generalize the backward search procedure to the $ABWT$.

Proposition 5.1. *Given a string $p \in \Sigma^*$, let $[b, e]$ denote the range of rows of $M_{alt}(w)$ which are prefixed by p. For any $x \in \Sigma$, let*

$$b' = \sum_{c \in \Sigma}^{c\leq x} P_L[c] \ - \ \mathrm{rank}_x(L, e-1) - 1, \qquad e' = \sum_{c \in \Sigma}^{c\leq x} P_L[c] \ - \ \mathrm{rank}_x(L, b).$$

If $b' \leq e'$, then $[b', e']$ is the range of rows of $M_{alt}(w)$ which are prefixed by xp if $b' > e$ then no rows of $M_{alt}(w)$ are prefixed by xp and therefore xp is not a (circular) substring of w.

PROOF. Assume first $b' \leq e'$. It is immediate that if i, j are the positions of the first and last x's in $L[b, e]$, then $b' = LF(j)$ and $e' = LF(i)$ and every other x in $L[b, e]$ is mapped to a position between b' and e'. The thesis follows since all rows in $M_{alt}(w)$ are rotations of w. If $b' > e$, then $\text{rank}_x(L, e-1) = \text{rank}_x(L, b)$ and there are no x's in $L[e, b]$ and xp is not a circular substring of w. □

Proposition 5.1 implies that if we use a compressed representation of the last columns L of $M_{alt}(w)$ supporting constant time rank operations, then, for any pattern p, we can compute in $O(|p|)$ time the range of rows of the matrix $M_{lex}(w)$ which are prefixed by p. Hence, the $ABWT$ can be used as a compressed index in the same way as the BWT.

The above results suggest that it is worthwhile to search for other transformations in the class BWT_K which share the same properties of BWT and $ABWT$. Because of the important role played by the rank operation, we introduce the notion of rank-invertibility for the class of BWT_K transformations.

Definition 5.2. *The transformation BWT_K is rank-invertible if there exists a function f_K such that, for any word w, setting $L = bwt_K(w)$ we have*

$$LF(i) = f_K(P_L, L[i], \text{rank}_{L[i]}(L, i)).$$

In other words, $LF(i)$ only depends on the Parikh vector P_L of L, the symbol $L[i]$, and the number of occurrences of $L[i]$ in L up to position i.

Note that we pose no limit to the complexity of the function f_K, we only ask that it can be computed using only P_L and the number of occurrences of $L[i]$ in $L[0, i]$.

We observed that, for $K = (Id, Rev)$, BWT_K coincides with $ABWT$ and it is therefore rank-invertible. The main result of this section is Theorem 5.9 establishing that BWT and $ABWT$ are the only rank-invertible transformations in the class BWT_K. We start our analysis considering the case $|K| = 2$.

Lemma 5.3. *Let $\Sigma = \{a, b, c\}$, and $K = (Id, \pi)$, where π is a permutation of Σ. If there exist two pairs $t_1 = (x, y)$ and $t_2 = (z, w)$ of symbols of Σ such that*

$$x <_{Id} y, \qquad z <_{Id} w \qquad \text{and} \qquad x <_{\pi} y, \qquad z >_{\pi} w,$$

then BWT_K is not rank-invertible.

PROOF. Consider for example the case $\pi = (c, a, b)$. Two pairs satisfying the hypothesis are $t_1 = (a, b)$ and $t_2 = (b, c)$ since according to the ordering $<_\pi$ it is

$$a <_\pi b \qquad \text{and} \qquad b >_\pi c.$$

Consider now the two words $s_1 = aabcc$ and $s_2 = abacc$. Both words contain two a's. In the first word the a's are followed respectively by a, b (the symbols in t_1), and in s_2 the a's are followed by b, c (the symbols in t_2).

Let F_1, L_1 (resp. F_2, L_2) denote the first and last columns of the matrix M_K associated to $bwt_K(s_1)$ (resp. $bwt_K(s_2)$). By definition, each matrix is obtained sorting the cyclic rotations of s_1 and s_2 according to the lexicographic order $\prec_K$ where symbols in odd positions are sorted according to the usual alphabetic order, while symbols in even positions are sorted according to the ordering π. We show the two matrices in Figure 2, where we use subscripts to distinguish the two a's occurrences in s_1 and s_2.

The relative position of the two a's in L_1 is determined by the symbols following them in s_1, namely those in $t_1 = (a, b)$. Since these symbols are in the first column of the cyclic rotations matrix, which is sorted according to the usual alphabetic order, the two a's appear in L_1 in the order a_1, a_2. The same is true for L_2: since the pair t_2 is also sorted, the two a's appear in L_2 in the order a_1, a_2.

The position of the two a's in F_1 is also determined by the symbols following them in s_1; but since these symbols are now in the second column, their relative order is determined by the ordering π. Hence the two a's appear in F_1 in the order a_1, a_2. In F_2 the ordering of the a's is a_2, a_1 since it depends on the π-ordering of the symbols of t_2 which *by construction* is different than their Id-ordering.

Note that s_1 and s_2 have the same Parikh vector $\langle 2, 1, 2\rangle$. If, by contradiction, BWT_K were rank invertible, the function f_K should give the correct LF-mapping for both s_1 and s_2. This is impossible since for s_1 we should have

$$f_K(\langle 2, 1, 2\rangle, a, 1) = 1, \qquad f_K(\langle 2, 1, 2\rangle, a, 2) = 2,$$

while for s_2 we should have

$$f_K(\langle 2, 1, 2\rangle, a, 1) = 2, \qquad f_K(\langle 2, 1, 2\rangle, a, 2) = 1.$$

In the general case of an arbitrary permutation π satisfying the hypothesis of the lemma the reasoning is the same. Note that such permutations are (a, c, b), (b, a, c), (b, c, a) and (c, a, b). Given the two pairs t_1 and t_2 we build two words s_1 and s_2 with Parikh vector $\langle 2, 1, 2\rangle$ such that in s_1 (resp. s_2) the two occurrences of a are followed by the symbols in t_1 (resp. t_2). We

$$
\begin{array}{cccccc}
 & F_1 & & & & L_1 \\
 & \downarrow & & & & \downarrow \\
s_1 \rightarrow & a_1 & a_2 & b & c & c \\
 & a_2 & b & c & c & a_1 \\
 & b & c & c & a_1 & a_2 \\
 & c & c & a_1 & a_2 & b \\
 & c & a_1 & a_2 & b & c
\end{array}
\qquad
\begin{array}{cccccc}
 & F_2 & & & & L_2 \\
 & \downarrow & & & & \downarrow \\
 & a_2 & c & c & a_1 & b \\
s_2 \rightarrow & a_1 & b & a_2 & c & c \\
 & b & a_2 & c & c & a_1 \\
 & c & c & a_1 & b & a_2 \\
 & c & a_1 & b & a_2 & c
\end{array}
$$

Figure 2: Cyclic rotation matrices for the words s_1 and s_2. We use subscripts to distinguish the two occurrences of a in each word.

then build the rotation matrices as before, and we find that in both L_1 and L_2 the two a's are in the order a_1, a_2. However, in columns F_1 and F_2 the two a's are not in the same relative order since it depends on the ordering π, and, by construction, such an order is not the same. Reasoning as before, we get that there cannot exist a function f_K giving the correct LF-mapping for both s_1 and s_2. □

Lemma 5.4. *Let $|\Sigma| \geq 2$ and $K = (Id, \pi)$. Then BWT_K is rank-invertible if and only if $\pi = Id$ or $\pi = Rev$.*

Proof. If $|\Sigma| = 2$ the result is trivial since the only possible permutations on binary alphabet are the identity and reverse permutation. Let us assume $|\Sigma| \geq 3$. We need to prove that if $\pi \neq Id$ and $\pi \neq Rev$ then BWT_K is not rank-invertible.

Note that any permutation π over the alphabet Σ induces a new ordering on any triplet of symbols in Σ. For example, if $\Sigma = \{a, b, c, d, e, f\}$ the permutation $\pi = (d, e, c, f, a, b)$ induces on the triplet $\{a, b, c\}$ the ordering $\pi_{abc} = (c, a, b)$. It is easy to prove that, if $\pi \neq Id$ and $\pi \neq Rev$, then there exists a triplet $\{x, y, z\}$, with $x < y < z$, such that $\pi_{xyz} \neq (x, y, z)$ and $\pi_{xyz} \neq (z, y, x)$. That is, π restricted to $\{x, y, z\}$ is different from the identity and reverse permutation. Without loss of generality we can assume that the triplet is $\{a, b, c\}$.

For any permutation π_{abc}, different from (a, b, c) and (c, b, a), there exist two pairs of symbols satisfying the hypothesis of Lemma 5.3. Hence, we can build two words s_1 and s_2 which show that BWT_K is not rank-invertible. Note that the argument in the proof of Lemma 5.3 is still valid if we add to s_1 and s_2 the same number of occurrences of symbols in Σ different from a, b, c so that s_1 and s_2 are effectively over an alphabet of size $|\Sigma|$. □

Lemma 5.4 establishes which BWT_K transformations are rank-invertible when $|K| = 2$. To study the general case $|K| > 2$, we start by establishing a simple corollary.

Corollary 5.5. *Let $|\Sigma| \geq 3$ and $K = (Id, \pi, \pi_2, \ldots, \pi_{k-1})$. If $\pi \neq Id$ and $\pi \neq Rev$ then BWT_K is not rank-invertible.*

PROOF. We reason as in the proof of Lemma 5.4, observing that the presence of the permutations $\pi_2, \ldots, \pi_{k-1}$ has no influence on the proof since the row ordering is determined by the first two symbols of each rotation. □

The following three lemmas establish necessary conditions on the structure of the tuple K for BWT_K to be rank-invertible. In particular, the following lemma shows that BWT_K is not rank-invertible if K contains anywhere a triplet (Id, Id, π) with $\pi \neq Id$.

Lemma 5.6. *Let $|\Sigma| \geq 2$ and $K = (\pi_0, \pi_1, \ldots, \pi_{k-1})$ such that $\pi_0 = Id$ and, for some $0 \leq i \leq k-1$, $(\pi_i, \pi_{(i+1) \bmod k}, \pi_{(i+2) \bmod k}) = (Id, Id, \pi)$ with $\pi \neq Id$. Then BWT_K is not rank-invertible.*

PROOF. Note that when $i = 0$, the k-tuple K starts with the triplet (Id, Id, π), when $i = k - 1$ the k-tuple K is $(Id, \pi, \ldots, Id)$. We first analyze the case $|\Sigma| = 2$ implying that $\pi = Rev$. Let us consider the words $s_1 = a_1 b^i a_2 b^{i+1} bb$ and $s_2 = a_1 b^{i+1} a_2 b^{i+1} b$, where we use subscripts to distinguish the two different occurrences of the symbol a. It is easy to see that, in the cyclic rotations matrix for s_1, a_1 precedes a_2 in both the first and the last column. Hence if BWT_K were rank-invertible we should have

$$f_K(\langle 2, 2i+3 \rangle, a, 1) = 1, \qquad f_K(\langle 2, 2i+3 \rangle, a, 2) = 2.$$

At the same time, in the cyclic rotations matrix for s_2, a_1 precedes a_2 in the last columns, but in the first column a_2 precedes a_1 since the two rotations prefixed by a differ in the $(i+2)$-th column and $b <_{Rev} a$. Therefore we should have

$$f_K(\langle 2, 2i+3 \rangle, a, 1) = 2, \qquad f_K(\langle 2, 2i+3 \rangle, a, 2) = 1.$$

Hence BWT_K cannot be rank-invertible.

Let us consider the case $|\Sigma| \geq 3$. Since $\pi \neq Id$ there are two symbols, say b and c, such that their relative order according to π is reversed, that is, $b < c$ and $c <_\pi b$. Consider now the words $s_1 = a_1 c^i b a_2 c^i ccc$ and $s_2 = a_1 c^{i+1} b a_2 c^{i+1} c$ where we use subscripts to distinguish the two different occurrences of the symbol a. It is immediate to see that, in the cyclic rotations matrix for s_1,

a_1 precedes a_2 in both the first and the last column. Hence if BWT_K were rank-invertible we should have

$$f_K(\langle 2, 1, 2i+3\rangle, a, 1) = 1, \qquad f_K(\langle 2, 1, 2i+3\rangle, a, 2) = 2.$$

At the same time, in the cyclic rotations matrix for s_2, a_1 precedes a_2 in the last columns, but in the first column a_2 precedes a_1 since the two rotations prefixed by a differ in the $(i+3)$-th column and $c <_\pi b$. Hence we should have

$$f_K(\langle 2, 1, 2i+3\rangle, a, 1) = 2, \qquad f_K(\langle 2, 1, 2i+3\rangle, a, 2) = 1$$

hence BWT_K cannot be rank-invertible. □

The following lemma shows that BWT_K is not rank-invertible if K contains anywhere a triplet (Id, Rev, π), with $\pi \neq Id$.

Lemma 5.7. *Let $|\Sigma| \geq 2$ and $K = (\pi_0, \pi_1, \ldots, \pi_{k-1})$ such that $\pi_0 = Id$ and, for some $0 \leq i < k-2$, $(\pi_i, \pi_{i+1}, \pi_{i+2}) = (Id, Rev, \pi)$ with $\pi \neq Id$. Then BWT_K is not rank-invertible.*

Proof. As in the proof of Lemma 5.6, we can consider the words $s_1 = a_1 b^i a_2 b^{i+1} bb$ and $s_2 = a_1 b^{i+1} a_2 b^{i+1} b$ in case of binary alphabet, and the words $s_1 = a_1 c^i b a_2 c^i ccc$ and $s_2 = a_1 c^{i+1} b a_2 c^{i+1} c$ in the general case by assuming that there are two symbols, say b and c, such that their relative order according to π is reversed, that is, $b < c$ and $c <_\pi b$. Recall that we use subscripts to distinguish the two different occurrences of the symbol a. In the cyclic rotations matrix for s_1, in the first column a_2 precedes a_1 while in the last column a_1 precedes a_2. At the same time, in both the first and the last column of the cyclic rotations matrix for s_2, a_2 precedes a_1. Reasoning as in the proof of Lemma 5.6 we get that BWT_K cannot be rank-invertible. □

The following lemma shows that BWT_K is not rank-invertible if K contains anywhere a triplet (Rev, Id, π), with $\pi \neq Rev$.

Lemma 5.8. *Let $|\Sigma| \geq 2$ and $K = (\pi_0, \pi_1, \ldots, \pi_{k-1})$ such that $\pi_0 = Id$ and, for some $1 \leq i \leq k-1$, $(\pi_i, \pi_{(i+1) \bmod k}, \pi_{(i+2) \bmod k}) = (Rev, Id, \pi)$ with $\pi \neq Rev$. Then BWT_K is not rank-invertible.*

Proof. We reason as in the proof of Lemma 5.7 considering again the words $s_1 = ab^i ab^{i+1} bb$ and $s_2 = ab^{i+1} ab^{i+1} b$ in case of binary alphabet and the words $s_1 = ac^i bac^i ccc$ and $s_2 = ac^{i+1} bac^{i+1} c$ in the general case. □

We are now ready to establish the main result of this section.

Theorem 5.9. *If $|\Sigma| \geq 2$, BWT and ABWT are the only transformations BWT_K which are rank invertible.*

Proof. For $|K| = 2$, the result follows from Lemma 5.4. Let us suppose $K = (Id, \pi_1, \ldots, \pi_{k-1})$ with $k > 2$ and assume BWT_K is rank invertible. Both in the case of binary alphabet and in the general case, by Corollary 5.5, we must have $\pi_1 = Id$ or $\pi_1 = Rev$. If $\pi_1 = Id$ and $BWT_K \neq BWT$ then the k-tuple K must contain the triplet (Id, Id, π) with $\pi \neq Id$ which is impossible by Lemma 5.6. If $\pi_1 = Rev$, by Lemma 5.7 $\pi_2 = Id$. We have therefore established that K has the form $K = (Id, Rev, Id, \pi_3, \ldots, \pi_{k-1})$. We can suppose that $k > 3$. In fact, if were $k = 3$ then, by Lemma 5.8, K would cyclically contain the triplet (Rev, Id, Id), so BWT_K would not be rank-invertible, a contradiction. By Lemma 5.8 it is $\pi_3 = Rev$. By iterating the same reasoning we can conclude that k is even and BWT_K coincides with $ABWT$, concluding the proof. □

6. Efficient computation of the ABWT

The bottleneck for the computation of $ABWT$ (as well as of any transformation BWT_K) of a given string w is the $\preceq_{alt}$-based (the $\preceq_K$-based) sorting of its cyclic rotations. In Remark 4.3 we have observed that, if a unique end-of-string symbol, which is smaller than any other symbol in the alphabet, is appended to the input string, all transformations in the class BWT_K can be computed in linear time by first building the suffix tree for the input string. However, for computing the BWT this strategy has never been used in practice. The reason is that the algorithms for building the suffix tree, although they take linear time, have a large multiplicative constant and are not fast in practice. In addition, the suffix tree itself requires a space of about ten/fifteen times the size of the input which is a huge amount of temporary space that is not necessarily available (considering also that saving space is the primary reason for using the BWT). For the above reasons the BWT is usually computed by first building the Suffix Array [27, 40] which is the array giving the lexicographic order of all the suffixes of the input string.

A fundamental result on Suffix Array construction is the technique in [28] that, using the concept of *difference cover*, makes it possible to design efficient Suffix Array construction algorithms for different models of computation including RAM, External Memory, and Cache Oblivious.

In this section, we show that this technique can be adapted to compute the $ABWT$ within the same time bound of the BWT.

Firstly, in order to use the notion of suffix array for the computation of $ABWT$ we need to extend the definition of alternating lexicographic order also for strings having different length.

Definition 6.1. *Let $x = x_0x_1 \dots x_{s-1}$ and $y = y_0y_1 \dots y_{t-1}$ with $s < t$.*

1. *If x is not a prefix of y and i is the smallest index in which $x_i \neq y_i$ Then, if i is even $x \prec_{alt} y$ iff $x_i < y_i$. Otherwise, if i is odd $x \prec_{alt} y$ iff $x_i > y_i$.*
2. *If x is a prefix of y, we say that $x \prec_{alt} y$ if $|x|$ is even, $y \prec_{alt} x$ if $|x|$ is odd.*

Suffix array algorithms often assume that the input string ends with a unique end-of-string symbol smaller than any other in the alphabet Σ. Remark that if we append the end-of-string symbol \$ to the string w, the $\prec_{alt}$-order relation between two suffixes of $w\$$ is determined by using Definition 6.1 (case 1). Moreover, using the end-of-string symbol \$ implies that the $\preceq_{alt}$-based sorting of the cyclic rotations of input string is induced by the $\prec_{alt}$-based sorting of its suffixes. Note that this property does not hold in general. However, it is easy to verify that, apart from the symbol \$, the output $abwt(w\$)$ may be different from $abwt(w)$ and the number of equal letter runs can be different (see Figure 4).

Here we assume that the input string w contains a unique end-of-string symbol \$, but, in the next section, we show how to remove this hypothesis by using combinatorial properties of some special rotations of the input string.

To illustrate the idea behind difference cover algorithms, in the following, given a positive integer v, we denote by $[0, v)$ the set $\{0, 1, \dots, v-1\}$.

Definition 6.2. *A set $D \subseteq [0, v)$ is a* difference cover *modulo v if every integer in $[0, v)$ can be expressed as a difference, modulo v, of two elements of D, i.e.*

$$\{(i - j) \bmod v \mid i, j, \in D\} = [0, v).$$

For example, for $v = 7$ the set $\{0, 1, 3\}$ is a difference cover, since $0 = 0-0$, $1 = 1 - 0$, $2 = 3 - 1$, $3 = 3 - 0$, $4 = 0 - 3 \bmod 7$, and so on. An algorithm by Colbourn and Ling [6] ensures that for any v a difference cover modulo v of size at most $\sqrt{1.5v} + 6$ can be computed in $O(\sqrt{v})$ time. The suffix array construction algorithms described in [28] are based on the general strategy shown in Algorithm 1. Steps 3 and 4 rely heavily on the following property of Difference covers: for any $0 \leq i, j < n$ there exists $k < v$ such that $(i + k) \bmod v \in D$ and $(j + k) \bmod v \in D$. This implies that to compare lexicographically suffixes $w[i, n-1]$ and $w[j, n-1]$ it suffices to compare at most v symbols since $w[i + k, n-1]$ and $w[j + k, n-1]$ are both sampled suffixes and their relative order has been determined at Step 2.

To see how the algorithm works consider for example $v = 6$, $D = \{0, 1, 3\}$ and the string $w = abaacabaacab\$$. The sampled suffixes are those starting at positions $0, 1, 3, 6, 7, 9, 12$. To sort them, consider the string over Σ^v

Algorithm 1 DIFFERENCE COVER SUFFIX SORTING.

Input: A string w of length n and a modulo-v difference cover D

Output: w's suffixes in lexicographic order

1: Consider the $(n|D|)/v$ suffixes $w[i, n-1]$ starting at positions i such that $i \bmod v \in D$. These suffixes are called the *sampled suffixes*.
2: Recursively sort the sampled suffixes
3: Sort non-sampled suffixes
4: Merge sampled and non-sampled suffixes

whose elements are the v-tuples starting at the sampled positions in the order $0, 6, 12, 1, 7, 3, 9$:

$$R[0,6] = \overset{w[0,5]}{abaaca}\ \overset{w[6,11]}{baacab}\ \overset{w[12,18]}{\$\$\$\$\$\$}\ \overset{w[1,6]}{baacab}\ \overset{w[7,12]}{aacab\$}\ \overset{w[3,8]}{acabaa}\ \overset{w[9,14]}{cab\$\$\$}$$

(note we have added additional \$'s to make sure all blocks contain v symbols). The difference cover algorithm then renames each v-tuple with its lexicographic rank. Since

$$\$\$\$\$\$ \preceq_{lex} aacab\$ \preceq_{lex} abaaca \preceq_{lex} acabaa \preceq_{lex} baacab \preceq_{lex} cab\$\$\$$$

the renamed string is $R_{bwt} = [2, 4, 0, 4, 1, 3, 5]$. The crucial observation is that the suffix array for R_{bwt}, which in our example is $SA(R_{bwt}) = [2, 4, 0, 5, 1, 3, 6]$, provides the lexicographic ordering of the sampled suffixes. Indeed $R[2] = w[12, 18]$ is the smallest sampled suffix, followed by $R[4] = w[7, 12]$, followed by $R[0] = w[0, 5]$, and so on. The Suffix Array of R_{bwt} is computed with a recursive call at Step 2, and is later used in Steps 3 and 4 to complete the sorting of all suffixes.

To compute $abwt(w)$ with the difference cover algorithm, we consider the same string R but we sort the v-tuples according to the *alternating* lexicographic order. Since

$$\$\$\$\$\$ \preceq_{alt} acabaa \preceq_{alt} abaaca \preceq_{alt} aacab\$ \preceq_{alt} baacab \preceq_{alt} cab\$\$\$$$

it is $R_{abwt} = [2, 4, 0, 4, 3, 1, 5]$. Next, we compute the Suffix Array of R_{abwt} according to the *standard* lexicographic order, $SA(R_{abwt}) = [2, 5, 0, 4, 1, 3, 6]$. We now show that, since $v = 6$ is even, $SA(R_{abwt})$ provides the correct *alternating* lexicographic order of the sampled suffixes.

To see this, assume $w[i, n-1]$ and $w[j, n-1]$ are sampled suffixes with a common prefix of length ℓ. Hence $w[i, i+\ell-1] = w[j, j+\ell-1]$ while $w[i+\ell] \neq w[j+\ell]$. Let $R_{abwt}[t_i]$ and $R_{abwt}[t_j]$ denote the entries in R_{abwt} corresponding to $w[i, i+v-1]$ and $w[j, j+v-1]$. By construction, the

suffixes $R_{abwt}[t_i, r]$ and $R_{abwt}[t_j, r]$ have a common prefix of $\lfloor \ell/v \rfloor$ entries (each one corresponding to a length-v block in w) followed respectively by $R_{abwt}[t_i + \lfloor \ell/v \rfloor]$ and $R_{abwt}[t_j + \lfloor \ell/v \rfloor]$ which are different since they correspond to the v-tuples $R[t_i + \lfloor \ell/v \rfloor]$ and $R[t_j + \lfloor \ell/v \rfloor]$ which differ since they contain the symbols $w[i+\ell]$ and $w[j+\ell]$ in position (ℓ mod v). Assuming for example that $w[i+\ell] < w[j+\ell]$, it is $w[i, n-1] \prec_{alt} w[j, n-1]$ depending on whether ℓ is odd or even. Since v is even, ℓ is even iff ℓ mod v is even, hence

$$\begin{aligned} w[i, n-1] \prec_{alt} w[j, n-1] &\iff R[t_i + \lfloor \ell/v \rfloor] \preceq_{alt} R[t_j + \lfloor \ell/v \rfloor] \\ &\iff R_{abwt}[t_i + \lfloor \ell/v \rfloor] < R_{abwt}[t_j + \lfloor \ell/v \rfloor] \\ &\iff R_{abwt}[t_i, r] \preceq_{lex} R_{abwt}[t_j, r] \end{aligned}$$

which shows that the standard Suffix Array for R_{abwt} provides the alternating lexicographic order of the sampled suffixes, as claimed.

Summing up, after building the string R_{abwt}, at Step 2 we compute $SA(R_{abwt})$ using the standard Difference cover algorithm, or any other suffix sorting algorithm. Finally, Step 3 and 4 can be easily adapted to the alternating lexicographic order using its property that for any symbol $c \in \Sigma$ and strings $\alpha, \beta \in \Sigma^*$ it is

$$c\alpha \prec_{alt} c\beta \iff \beta \prec_{alt} \alpha. \tag{4}$$

For example, to compare $w[0, 12]$ with $w[5, 12]$ we notice that after $w[0] = w[5]$ we reach the sampled suffixes $w[1, 12]$ and $w[6, 12]$ corresponding to $R[3, 6]$ and $R[1, 6]$. According to $SA(R_{abwt})$ it is $R[1, 6] \preceq_{lex} R[3, 6]$ which implies $w[6, 12] \prec_{alt} w[1, 12]$, and by (4) $w[0, 12] \preceq_{lex} w[5, 12]$. Since from the alternating lexicographic order of w's suffixes $abwt(w)$ can be computed in linear time, the results in [28] can be translated as follows.

Theorem 6.3. *Given a string $w[0, n-1]$ ending with a unique end-of-string symbol, we can compute $abwt(w)$ in RAM in $\mathcal{O}(n)$ time, or in $\mathcal{O}(n \log\log n)$ time but using only $n + o(n)$ words of working space. In external memory, using D disks with block size B and a fast memory of size M, $abwt(w)$ can be computed in $\mathcal{O}(\frac{n}{DB} \log_{M/B} n/B)$ I/Os and $\mathcal{O}(n \log_{M/B} n/B)$ internal work.*

We point out that the above results cannot be easily extended to the generalized BWTs introduced in Section 4. The reason is that Step 3 and 4 of the modified Difference cover algorithm hinge on Property (4) that does not hold in general for the lexicographic orders introduced by Definition 4.1.

7. Galois words and ABWT computation for arbitrary rotations

Galois words, introduced in [46], are generalization of Lyndon words for the alternating lexicographic order. Roughly speaking, a Galois word is the smallest cyclic rotation within its conjugacy class, with respect to $\preceq_{alt}$ order. Some characterizations of Galois words by using infinite words and some properties of words obtained as a nonincreasing factorization in Galois words, are studied in [11]. Although, in general, Galois and Lyndon words are distinct within a conjugacy class, some properties that hold for Lyndon words are preserved.

In this section, we explore some combinatorial properties of Galois words and, in particular, we consider the issue of designing an efficient strategy to find the Galois rotation of a given word, as posed in [11]. The problem can be solved by two different approaches. In the first approach we show how to find the Galois rotation directly using specific combinatorial properties of Galois words. The second approach, suggested by one of the Reviewers of this paper, is outlined in Remark 7.8 and uses similar arguments as in Section 6, by reducing the problem to compute the Lyndon rotation of a new word over a new alphabet obtained by encoding factors of even length.

Which ever of the two methods is used, the results of this section allow to prove that, for the computation of the $ABWT$ of a string w, Galois words play a role similar to that of Lyndon words for BWT [25, 2], hence the computation of $ABWT$ can be linearly performed, even if no end-of-string symbol is appended to the input.

Definition 7.1. *A primitive word w is a* Galois word *if for each nontrivial factorization $w = uv$, one has $w \preceq_{alt} vu$.*

Example 7.2. *The words $w = ababba$ and $v = aababb$ are, respectively, the Galois word and the Lyndon word within the same conjugacy class. Another example is $w = ababaa$ and $v = aaabab$.*

Firstly, we show that a string w is a Galois word if it is smaller than its proper suffixes, with respect to $\prec_{alt}$ order introduced in Definition 6.1 (see Figure 3 for an example).

The following result has been proved in [46] (Proposition 3.1). Here we restate the proof by using our notation.

Lemma 7.3. *If a Galois word w has a border u, then u has odd length.*

Proof. Let u be both suffix and prefix of w. This means that $w = uv' = v''u$. By definition, $w = uv' \prec_{alt} uv''$. If $|u|$ would be even, then it should be $v' \prec_{alt} v''$. On the other hand, $w = v''u \prec_{alt} v'u$ implies that $v'' \prec_{alt} v'$, a contradiction. □

Lyndon words can be defined as the strings that are smaller of its proper suffixes. Such a characterization also holds for Galois words, as shown in the following proposition. A different proof of this result, involving infinite words, is given in [11].

Proposition 7.4. *A primitive word w is a Galois word if and only if w is smaller than any of its suffix, with respect to $\prec_{alt}$ order.*

Proof. Let w be a Galois word and let v a suffix of w. This means that $w = uv$. If v is also prefix of w, then by Lemma 7.3 v has odd length, i.e. $w \prec_{alt} v$. If v is not a prefix of w, then there exists $0 \leq i < |v| - 1$ such that $v_i \neq w_i$. Since w is a Galois word, $uv \prec_{alt} vu$. This fact implies that $w = uv \prec_{alt} v$. Conversely, let $w = uv$. Since $uv \prec_{alt} v$, we can distinguish two cases, whether v is prefix of w or not. If v is not a prefix of w then $uv \prec_{alt} vu$. If v is a prefix of w, then the length of v is odd. Therefore if it would be $vu \prec_{alt} uv = vu'$ then $u' \prec_{alt} u$ that implies $u' \prec_{alt} uv$ that is a contradiction. □

It is known that, when Lyndon words are considered, the lexicographic sorting of its suffixes induces the $\preceq_{lex}$-sorting of the conjugates. Such a result is useful to compute the BWT of a string without using any end-of-string symbol [25]. The following proposition shows that this property also holds for Galois words. In fact, the $\preceq_{alt}$-based sorting of the cyclic rotations of a primitive Galois word can be reduced to the $\prec_{alt}$-based sorting of its suffixes. An example of this property is reported in Figure 3.

$a\ b\ a\ b\ b\ a$	$a\ b\ a\ b\ b\ a$	$a\ b\ a\ b\ b$
$a\ b\ b\ a\ a\ b$	$a\ b\ b\ a$	$a\ b\ b$
$a\ a\ b\ a\ b\ b$	a	$a\ a\ b\ a\ b\ b$
$b\ b\ a\ a\ b\ a$	$b\ b\ a$	$b\ b$
$b\ a\ a\ b\ a\ b$	$b\ a$	$b\ a\ b\ b$
$b\ a\ b\ b\ a\ a$	$b\ a\ b\ b\ a$	b
$M_{alt}(ababba)$	$Suf(ababba)$	$Suf(aababb)$

Figure 3: Left: the matrix M_{alt} of all cyclic rotations of the word *ababba*. Center: the $\prec_{alt}$-sorted suffixes of the Galois word *ababba*. Right: the $\prec_{alt}$-sorted suffixes of the Lyndon conjugate *aababb*. The $\prec_{alt}$-order of the last two suffixes is different from the $\preceq_{alt}$-order of the correspondent cyclic rotations.

Proposition 7.5. *Let w be a primitive Galois word and let u', u'', v', v'' be factors of w such that $w = u'v' = u''v''$. Then, $v'u' \prec_{alt} v''u'' \iff v' \prec_{alt} v''$.*

PROOF. Let us assume that $w' = v'u' \prec_{alt} w'' = v''u''$. There exists $0 \leq i < |w|$ such that $w'_i \neq w''_i$. Firstly, let us assume that $i < \min\{|v'|, |v''|\}$. In this case $v' \prec_{alt} v''$. Let us assume now that v' is a prefix of v'', i.e. $v'' = v's$, for some non-empty string s. If $|v'|$ would be odd, then $v'u' \prec_{alt} v''u'' = v'su'' \Rightarrow su'' \prec_{alt} u' \Rightarrow su''v' \prec_{alt} u'v'$, that is a contradiction. So, $|v'|$ is even and by definition $v' \prec_{alt} v''$. Let us consider the case v'' is a prefix of v', i.e. $v' = v''t$, for some string t. If $|v''|$ would be even then $v''tu' = v'u' \prec_{alt} v''u'' \Rightarrow tu' \prec_{alt} u'' \Rightarrow tu'v'' \prec_{alt} u''v''$, that is a contradiction. So, $|v''|$ is odd then, by definition, $v' \prec_{alt} v''$.

Conversely, let us suppose that $v' \prec_{alt} v''$. If neither v' is prefix of v'' nor v'' is prefix of v', then $v'u' \prec_{alt} v''u''$. Let us suppose now that v' is prefix of v'', i.e. $v'' = v's$ then v' has even length. Since w is a Galois word, $u'v' \prec_{alt} su''v' \Rightarrow u' \prec_{alt} su'' \Rightarrow v'u' \prec_{alt} v'su'' = v''u''$. Let us suppose now that v'' is prefix of v', i.e. $v' = v''t$ then v'' has odd length. The fact that w is a Galois word implies that $u''v'' \prec_{alt} tu'v'' \Rightarrow u'' \prec_{alt} tu' \Rightarrow v''tu' = v'u' \prec_{alt} v''u''$. □

It is known that the unique Lyndon conjugate of a string w is one of the elements in the non-increasing factorization of ww into Lyndon words [12]. As proved in [11], this strategy does not work for Galois words. Hence, we propose a linear time algorithm, named FINDGALOISROTATION, to find, for each primitive string w of length n, its unique cyclic rotation that is a Galois word. Our algorithm is a variant of the one in [3, 31] to find the Lyndon conjugate of a given string. The algorithm FINDGALOISROTATION uses a *border array* B of length $n + 1$ that stores in each position $j > 0$ the length of the border of the j-length prefix of $w[k, (k + j - 1) \bmod n]$, i.e. the Galois rotation starting at position k, and $B[0] = -1$. At the end of the algorithm, B is the border array of the Galois rotation of the word.

At each iteration of the main **while** loop (lines 3–13), k is the starting position of the current candidate for the smallest cyclic rotation (with respect to $\preceq_{alt}$ order), $w[k, (k + j - 1) \bmod n]$ is a Galois word and $B[j] = i$ is the length of its border. This means that $w[k, (k + i - 1) \bmod n] = w[(k + j - i), (k+j-1) \bmod n]$. So, the characters $w[(k+i) \bmod n]$ and $w[(k+j) \bmod n]$ are compared. If those characters are equal, the length of the border is increased. If they are distinct, different alphabet orders are used depending on whether i is even or not, and the value of k is consequently updated. Note that, even if k is changed, the computed value $B[j + 1]$ is the same and the values $B[i]$, with $i \leq j$, do not need to be re-computed.

Theorem 7.6. *Given a primitive string w, its unique cyclic rotation that is a Galois word can be computed in linear time and space.*

Algorithm 2 FindGaloisRotation

Input: A primitive string w of length n

Output: The starting position $0 \leq k < n$ of the cyclic rotation of w that is a Galois word

```
1:  i ← 0; j ← 1; k ← 0;
2:  B[0] ← −1;
3:  while k + j < 2n do
4:      if j ≤ n then B[j] ← i;
5:      while i ≥ 0 and w[(k + j) mod n] ≠ w[(k + i) mod n] do
6:          if i mod 2 = 0 then
7:              if w[(k + j) mod n] < w[(k + i) mod n] then
8:                  k ← k + j − i; j ← i;
9:          else
10:             if w[(k + j) mod n] > w[(k + i) mod n] then
11:                 k ← k + j − i; j ← i;
12:         i ← B[i];
13:     i ← i + 1; j ← j + 1;
14: return k
```

Proof. We note that the auxiliary memory consists solely of the border array and that the execution time depends linearly on the number of comparisons between the characters in w. To prove that FindGaloisRotation requires at most $4n - 3$ comparisons, we consider the quantity $2(k + j) - i$ and show that it always increases after each comparison between the characters $w[(k + j) \bmod n]$ and $w[(k + i) \bmod n]$. If the two characters are equal, then both i and j are increased by one at Line 13. If the two characters are different, then the quantity $k + j$ remains unchanged and the value of i is decreased. Finally, note that if $n \geq 2$, the quantity $2(k + j) - i$ is equal to 2 for the first comparison and it is at most $2(2n - 1)$, so the overall number of comparisons is at most $4n - 3$ as claimed. □

The next corollary shows how to use the linear computation of the Galois rotation of a word to compute in linear time the $ABWT$ of an input string without using any end-of-string symbol.

Corollary 7.7. *The $ABWT$ of a generic string w can be computed in linear time.*

Proof. We apply FindGaloisRotation to w to find its Galois conjugate w'. Then we apply to $w'\$$ the algorithm described in Section 6. By using

$$
\begin{array}{cccccc}
a & n & a & n & a & b \\
a & n & a & b & a & n \\
a & b & a & n & a & n \\
b & a & n & a & n & a \\
n & a & b & a & n & a \\
n & a & n & a & b & a
\end{array}
$$

$M_{alt}(banana)$

$$
\begin{array}{ccccccc}
\$ & b & a & n & a & n & a \\
a & n & a & n & a & \$ & b \\
a & n & a & \$ & b & a & n \\
a & \$ & b & a & n & a & n \\
b & a & n & a & n & a & \$ \\
n & a & \$ & b & a & n & a \\
n & a & n & a & \$ & b & a
\end{array}
$$

$M_{alt}(banana\$)$

$$
\begin{array}{ccccccc}
\$ & a & n & a & n & a & b \\
a & n & a & n & a & b & \$ \\
a & n & a & b & \$ & a & n \\
a & b & \$ & a & n & a & n \\
b & \$ & a & n & a & n & a \\
n & a & b & \$ & a & n & a \\
n & a & n & a & b & \$ & a
\end{array}
$$

$M_{alt}(ananab\$)$

Figure 4: Left: the matrix M_{alt} of all cyclic rotations of the word $w = banana$, sorted by using $\preceq_{alt}$-order. The output is $abwt(w) = bnnaaa$. Center: the matrix M_{alt} of the word $banana\$$. The output is $abwt(banana\$) = abnn\$aa$. Right: the matrix M_{alt} of the word $ananab\$$, where $ananab$ is the Galois conjugate of w. The output is $abwt(ananab\$) = b\$nnaaa$.

Remark 3.1, we can deduce that $abwt(w)$ can be obtained from $abwt(w'\$)$ by just removing \$ from the output (see Figure 4 for an example). □

Remark 7.8. *An alternative lightweight strategy to compute the Galois rotation of a word consists in reducing the problem to find a Lyndon rotation of a new word, reasoning as in Section 6. More in details, if $|w|$ is odd, we can split $w' = ww$ into blocks of two characters and create a new word w'' of length $|w|$ over a new alphabet of size σ^2. Each cyclic rotation of w corresponds to a cyclic rotation of w''. The order on the new alphabet is given by the $\preceq_{alt}$-order on the new alphabet. If $|w|$ is even, we can find the smallest cyclic rotation that starts at an odd position and the smallest cyclic rotation that starts at an even position, and then compare them (in linear time). In both cases, we can consider blocks of two characters. By using an efficient algorithm to compute Lyndon rotations [49], such a strategy can be performed in linear time and constant space.*

The previous remark shows that the problem of finding the Galois rotation of a word can be reduced to compute Lyndon rotations of new words by using lexicographic order. It would be interesting to investigate whether similar strategies can be applied for the efficient computation of other generalized BWTs.

8. Conclusions

We have investigated the relationship between two word transformations, namely, BWT and $ABWT$. Our main contribution has been to establish a

deep connection and analogy between the two. In particular, we have shown that the $ABWT$, that originates from purely combinatorial considerations, can also be used in more practical settings such as Data Compression and Compressed Data Structures. Our results are also based on combinatorial properties of Galois words that are of independent interest.

Acknowledgements

The authors would like to thank both referees for comments and suggestions that have helped us improve the presentation of our ideas. Referee 1 provided a particularly insightful set of comments and suggestions regarding an alternative algorithm to compute the Galois rotations of a word.

RG and GM are partially supported by INdAM-GNCS project 2019 "Innovative methods for the solution of medical and biological big data" and MIUR-PRIN project "Multicriteria Data Structures and Algorithms: from compressed to learned indexes, and beyond" grant n. 2017WR7SHH.

GR and MS are partially supported by the project MIUR-SIR CMACBioSeq "Combinatorial methods for analysis and compression of biological sequences" grant n. RBSI146R5L.

References

[1] D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM T. Algorithms*, 11(4):31:1–31:21, 2015.

[2] S. Bonomo, S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. Sorting conjugates and suffixes of words in a multiset. *International Journal of Foundations of Computer Science*, 25(08):1161–1175, 2014.

[3] K. S. Booth. Lexicographically least circular substrings. *Inf. Process. Lett.*, 10(4/5):240–242, 1980.

[4] M. Burrows and D. J. Wheeler. A block sorting data compression algorithm. Technical report, DIGITAL System Research Center, 1994.

[5] B. Chapin and S. Tate. Higher Compression from the Burrows-Wheeler Transform by Modified Sorting. In *DCC*, page 532. IEEE Computer Society, 1998. Full version available from `https://www.uncg.edu/cmp/faculty/srtate/papers/bwtsort.pdf`.

[6] C. J. Colbourn and A. C. H. Ling. Quorums from difference covers. *Inf. Process. Lett.*, 75(1-2):9–12, 2000.

[7] A. Cox, M. Bauer, T. Jakobi, and G. Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012.

[8] A.J. Cox, F. Garofalo, G. Rosone, and M. Sciortino. Lightweight LCP construction for very large collections of strings. *J. Discrete Algorithms*, 37:17–33, 2016.

[9] M. Crochemore, J. Désarménien, and D. Perrin. A note on the Burrows-Wheeler transformation. *Theor. Comput. Sci.*, 332:567–572, 2005.

[10] J. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Lonard, and . Prieur-Gaston. A survey of string orderings and their application to the Burrows-Wheeler transform. *Theor. Comput. Sci.*, 2017.

[11] F. Dolce, A. Restivo, and C. Reutenauer. On generalized Lyndon words. *Theor. Comput. Sci.*, 777:232–242, 2019.

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

[13] L. Egidi, F. A. Louza, G. Manzini, and G. P. Telles. External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14(1):6:1–6:15, 2019.

[14] P. Fenwick. The Burrows-Wheeler transform for block sorting text compression: Principles and improvements. *Comput. J.*, 39(9):731–740, 1996.

[15] S. Ferenczi and L. Q. Zamboni. Clustering Words and Interval Exchanges. *Journal of Integer Sequences*, 16(2):Article 13.2.1, 2013.

[16] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *J. ACM*, 52(4):688–713, 2005.

[17] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS 2000*, pages 390–398. IEEE Computer Society, 2000.

[18] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52:552–581, 2005.

[19] P. Ferragina, I. Nitto, and R. Venturini. On Optimally Partitioning a Text to Improve Its Compression. *Algorithmica*, 61(1):51–74, 2011.

[20] T. Gagie, G. Manzini, and J. Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theor. Comput. Sci.*, 698:67–78, 2017.

[21] I. M. Gessel, A. Restivo, and C. Reutenauer. A bijection between words and multisets of necklaces. *Eur. J. Combin.*, 33(7):1537 – 1546, 2012.

[22] I. M. Gessel and C. Reutenauer. Counting permutations with given cycle structure and descent set. *J. Comb. Theory A*, 64(2):189–215, 1993.

[23] R. Giancarlo, G. Manzini, A. Restivo, G. Rosone, and M. Sciortino. Block Sorting-Based Transformations on Words: Beyond the Magic BWT. In *DLT*, volume 11088 of *LNCS*, pages 1–17. Springer International Publishing, 2018.

[24] R. Giancarlo, G. Manzini, G. Rosone, and M. Sciortino. A new class of searchable and provably highly compressible string transformations. In *CPM*, volume 128 of *LIPIcs*, pages 12:1–12:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[25] R. Giancarlo, A. Restivo, and M. Sciortino. From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization. *Theor. Comput. Sci.*, 387:236 – 248, 2007.

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

[27] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 943–955. Springer Berlin Heidelberg, 2003.

[28] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53:918–936, 2006.

[29] K. Kimura and A. Koike. Ultrafast SNP analysis using the Burrows-Wheeler transform of short-read data. *Bioinformatics*, 31(10):1577–1583, 2015.

[30] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[31] M. Lothaire. *Applied Combinatorics on Words (Encyclopedia of Mathematics and its Applications).* Cambridge University Press, New York, NY, USA, 2005.

[32] V. Mäkinen, D. Belazzougui, F. Cunial, and A. I. Tomescu. *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*. Cambridge University Press, 2015.

[33] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007.

[34] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. A new combinatorial approach to sequence comparison. *Theory Comput. Syst.*, 42(3):411–429, 2008.

[35] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. Burrows-Wheeler Transform and Run-Length Enconding. In *Combinatorics on Words - 11th International Conference, WORDS 2017. Proceedings*, volume 10432 of *LNCS*, pages 228–239. Springer, 2017.

[36] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, and L. Versari. Measuring the clustering effect of BWT via RLE. *Theor. Comput. Sci.*, 698:79–87, 2017.

[37] S. Mantaci, A. Restivo, and M. Sciortino. Burrows-Wheeler transform and Sturmian words. *Information Processing Letters*, 86:241–246, 2003.

[38] S. Mantaci, A. Restivo, and M. Sciortino. Distance measures for biological sequences: Some recent approaches. *Int. J. Approx. Reasoning*, 47(1):109–124, 2008.

[39] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[40] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.

[41] G. Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[42] I. Pak and A. Redlich. Long cycles in abc-permutations. *Functional Analysis and Other Mathematics*, 2:87–92, 2008.

[43] N. Prezza, N. Pisanti, M. Sciortino, and G. Rosone. SNPs detection by eBWT positional clustering. *Algorithms for Molecular Biology*, 14(1):3, 2019.

[44] A. Restivo and G. Rosone. Burrows-Wheeler transform and palindromic richness. *Theor. Comput. Sci.*, 410(30-32):3018 – 3026, 2009.

[45] A. Restivo and G. Rosone. Balancing and clustering of words in the Burrows-Wheeler transform. *Theor. Comput. Sci.*, 412(27):3019 – 3032, 2011.

[46] C. Reutenauer. Mots de Lyndon généralisés 54. *Sém. Lothar. Combin.*, pages 16, B54h, 2006.

[47] G. Rosone and M. Sciortino. The Burrows-Wheeler Transform between Data Compression and Combinatorics on Words. In *The Nature of Computation. Logic, Algorithms, Applications - 9th Conference on Computability in Europe, CiE 2013. Proceedings*, volume 7921 of *LNCS*, pages 353–364. Springer, 2013.

[48] M. Schindler. A fast block-sorting algorithm for lossless data compression. In *DCC*, page 469. IEEE Computer Society, 1997.

[49] Y. Shiloach. Fast canonization of circular strings. *J. Algorithms*, 2(2):107–121, 1981.

[50] J. Simpson and S. J. Puglisi. Words with simple Burrows-Wheeler transforms. *Electronic Journal of Combinatorics*, 15, article R83, 2008.

[51] L. Yang, X. Zhang, and T. Wang. The Burrows-Wheeler similarity distribution between biological sequences based on Burrows-Wheeler transform. *Journal of Theoretical Biology*, 262(4):742–749, 2010.