Information and Computation 294 (2023) 105068

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

A new class of string transformations for compressed text indexing [☆]

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

[a] *University of Palermo, Italy*
[b] *University of Pisa, Italy*

A R T I C L E   I N F O

Article history:
Received 11 August 2022
Received in revised form 18 June 2023
Accepted 4 July 2023
Available online 11 July 2023

Keywords:
Data indexing and compression
Burrows-Wheeler transformation
Combinatorics on words

A B S T R A C T

Introduced about thirty years ago in the field of data compression, the Burrows-Wheeler Transform (BWT) is a string transformation that, besides being a *booster* of the performance of memoryless compressors, plays a fundamental role in the design of efficient self-indexing compressed data structures. Finding other string transformations with the same remarkable properties of BWT has been a challenge for many researchers for a long time. In this paper, we introduce a whole class of new string transformations, called *local orderings-based transformations*, which have all the "myriad virtues" of BWT. As a further result, we show that such new string transformations can be used for the construction of the recently introduced *r*-index, which makes them suitable also for highly repetitive collections. In this context, we consider the problem of finding, for a given string, the BWT variant that minimizes the number of runs in the transformed string.

1. Introduction

Michael Burrows and David Wheeler introduced in 1994 a reversible word transformation [2], denoted by BWT, that turned out to have "myriad virtues".[1] 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 [4]. 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 [5–7].

The importance of this transformation has been further increased when in [8] it was proven that, in addition to making easier to represent a string in space close to its entropy, it also makes easier to search for pattern occurrences in the original string. After this discovery, data transformations inspired by the BWT have been proposed for compactly representing and searching other combinatorial objects such as: trees, graphs, finite automata, and even string alignments. See [9,10] for an attempt to unify some of these results and [11] for an in-depth treatment of the field of compact data structures.

☆ A preliminary version of this paper appeared in [1].
* Corresponding author.

E-mail addresses: raffaele.giancarlo@unipa.it (R. Giancarlo), giovanni.manzini@unipi.it (G. Manzini), antonio.restivo@unipa.it (A. Restivo), giovanna.rosone@unipi.it (G. Rosone), marinella.sciortino@unipa.it (M. Sciortino).

[1] We are using the catch phrase "myriad virtues" as a tribute to Alberto Apostolico who first used this expression in the seminal paper [3].

https://doi.org/10.1016/j.ic.2023.105068
0890-5401/© 2023 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Going back to the original Burrows-Wheeler string transformation, we can summarize its salient features as follows: **1**) it can be computed and inverted in linear time, **2**) it produces strings which are provably compressible in terms of the high order entropy of the input, **3**) it supports pattern search directly on the transformed string in time proportional to the pattern length. It is the *combination* of these three properties that makes the BWT a fundamental tool for the design of compressed self-indices. In Section 2 we review these properties and also the many attempts to modify the original design. However, we recall that, despite more than twenty years of intense scrutiny, the only non trivial known BWT variant that fully satisfies properties **1**–**3** is the *Alternating BWT* (ABWT). The ABWT has been introduced in [12] in the field of combinatorics of words and its basic algorithmic properties have been described in [13,14].

In this paper we introduce a new *whole family* of transformations that satisfy properties **1**–**3** and can therefore replace the BWT in the construction of compressed self-indices with the same time efficiency of the original BWT and the potential of achieving better compression. We show that our family, supporting linear time computation, inversion, and search, is a special case of a much larger class of transformations that also satisfy properties **1**–**3** except that, in the general case, inversion and pattern search may take quadratic time. Our larger class includes as special cases also the BWT and the ABWT and therefore it constitutes a natural candidate for the study of additional properties shared by all known BWT variants.

More in detail, in Section 3 we describe a class of string transformations based on *context adaptive alphabet orderings*. The main feature of the above class of transformations is that, in the rotation sorting phase, we use alphabet orderings that depend on the context (i.e., the longest common prefix of the rotations being compared). We prove that such transformations are always invertible and provide a general inversion algorithm that runs in time quadratic with respect to the string length. We consider also some subclasses of such transformations in which the permutations associated to each prefix are defined by "simple" rules and we show that they have more efficient inversion algorithms.

In Section 4 we consider the subclass of transformations based on *local orderings*. In this subclass, the alphabet orderings only depend on a constant portion of the context. We prove that local ordering transformations can be inverted in linear time, and that pattern search in the transformed string takes time proportional to the pattern length. Thus, these transformations have the same properties **1**–**3** that were so far prerogative of the BWT and ABWT. We also show that it is always possible to implement an r-index [15] on the top of a BWT based on a local ordering, thus making this transformation suitable also for highly repetitive collections.

Having now at our disposal a wide class of string transformations with the same remarkable properties of the BWT, it is natural to use them to improve BWT-based data structures by selecting the one more suitable for the task. In this paper we initiate this study by considering the problem of selecting the BWT variant that minimizes the number of runs in the transformed string. The motivation is that data centers often store highly repetitive collections, such as genome databases, source code repositories, and versioned text collections. For such collections theoretical and practical evidence suggests that entropy underestimates the compressibility of the collection and we can obtain much better compression ratios exploiting runs of equal symbols in the BWT [15–26].

As we review in Section 2.2, run-minimization is a challenging and multifaceted problem: we believe the introduction of a new class of string transformations makes it even more interesting. In Section 5 we contribute to this problem showing that, for constant size alphabet, for the most general class of transformations considered in this paper, the BWT variant that minimizes the number of runs can be found in linear time using a dynamic programming algorithm. Although such result does not lead to a practical compression algorithm, such minimal number of runs constitutes a lower bound for the number of runs achievable by the other variants described in this paper and therefore constitutes a baseline for further theoretical or experimental studies.

A preliminary version of this paper appeared in [1]. In this new version we introduce and analyze new BWT variants, in particular the ones described in Sections 3.3 and 3.4. We also added Section 4.1 where we show that the recently introduced r-index [15] can be adapted to work with some of the proposed BWT variants. This result is particularly important since the r-index is particularly suited to highly repetitive collections which are relevant for the run minimization problem discussed in Section 5. Another new result is the characterization of local ordering transformations contained in Section 4.2. The new version also includes a more in-depth discussion of the existing literature: Section 2.1 has been largely extended and Section 2.2 is new. Finally, we have added some carefully designed examples in order to better illustrate some subtle points of the exposition. All references have been updated, and new relevant results have been discussed in the paper.

2. Notation and background

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite ordered alphabet of size σ with $c_1 < c_2 < \cdots < c_\sigma$, where $<$ denotes the standard lexicographic order. We denote by Σ^* the set of strings over Σ. Given a string $x = x_1 x_2 \cdots x_n \in \Sigma^*$, we denote by $|x|$ its length n. We use ϵ to denote the empty string. A *factor* of x is written as $x[i, j] = x_i \cdots x_j$ with $1 \le i \le j \le n$. A factor of type $x[1, j]$ is called a *prefix*, while a factor of type $x[i, n]$ is called a *suffix*. The i-th symbol in x is denoted by $x[i]$. Two strings $x, y \in \Sigma^*$ are called *conjugate*, if $x = uv$ and $y = vu$, where $u, v \in \Sigma^*$. We also say that x is a *cyclic rotation* of y. A string x is *primitive* if all its cyclic rotations are distinct. Given a string x and $c \in \Sigma$, we write $\mathsf{rank}_c(x, i)$ to denote the number of occurrences of c in $x[1, i]$, and $\mathsf{select}_c(x, j)$ to denote the position of the j-th c in x.

Given a primitive string s, we consider the matrix of all its cyclic rotations sorted in lexicographic order. Note that the rotations are all distinct by the primitivity of s. The last column of the matrix is called the Burrows-Wheeler Transform of

$$
\begin{array}{cccccccccccc}
 & F & & & & & & & & L & \\
 & \downarrow & & & & & & & & \downarrow & \\
1 & a & a & a & b & a & c & a & a & b & \\
2 & a & a & b & a & a & a & b & a & c & \leftarrow s \\
3 & a & a & b & a & c & a & a & b & a & \\
4 & a & b & a & a & a & b & a & c & a & \\
5 & a & b & a & c & a & a & b & a & a & \\
6 & a & c & a & a & b & a & a & a & b & \\
7 & b & a & a & a & b & a & c & a & a & \\
8 & b & a & c & a & a & b & a & a & a & \\
9 & c & a & a & b & a & a & a & b & a &
\end{array}
\qquad
\begin{array}{cccccccccccc}
 & F & & & & & & & & L & \\
 & \downarrow & & & & & & & & \downarrow & \\
1 & a & c & a & a & b & a & a & a & b & \\
2 & a & b & a & c & a & a & b & a & a & \\
3 & a & b & a & a & a & b & a & c & a & \\
4 & a & a & a & b & a & c & a & a & b & \\
5 & a & a & b & a & a & a & b & a & c & \leftarrow s \\
6 & a & a & b & a & c & a & a & b & a & \\
7 & b & a & a & a & b & a & c & a & a & \\
8 & b & a & c & a & a & b & a & a & a & \\
9 & c & a & a & b & a & a & a & b & a &
\end{array}
$$

Fig. 1. The original BWT matrix for the string $s = aabaaabac$ (left), and the ABWT matrix of cyclic rotations sorted using the alternating lexicographic order (right). In both matrices the horizontal arrow marks the position of the original string s, and the last column L is the output of the transformation.

the string s and it is denoted by $bwt(s)$ (see Fig. 1 (left)). It is shown in [2] that $bwt(s)$ is always a permutation of s, and that there exists a linear time procedure to recover s, given $bwt(s)$ and the row index I of s in the rotations matrix (it is $I = 2$ in Fig. 1 (left)).

In this paper we follow the assumption usually done for text indexing that the last symbol of s is a unique end-of-string symbol. This guarantees the primitivity of s. In addition, this assumption implies that, if we compare two cyclic rotations symbol-by-symbol, they differ as soon one of them reaches the end-of-string symbol (or sooner). This property ensures that all families of transformations defined in this paper can be computed in linear time using a suffix tree as described in Section 3 (note that the BWT and ABWT can be computed in linear time also for a generic primitive string [6,14,27]). The reader will notice that the algorithms computing the *inverse* transformations provided in this paper do not make use of the unique end-of-string symbol and therefore are valid for any primitive string.

The BWT has been introduced as a data compression tool: it was empirically observed that $bwt(s)$ usually contains long runs of equal symbols. This notion was later mathematically formalized in terms of the empirical entropy of the input string [5,7]. For $k \geq 0$, the k-th order empirical entropy of a string x, denoted as $H_k(x)$, is a lower bound to the compression ratio of any algorithm that encodes each symbol of x using a codeword that only depends on the k symbols preceding it in x. The simplest compressors, such as Huffman coding, in which the code of a symbol does not depend on the previous symbols, typically achieve a (modest) compression bounded in terms of the zeroth-order entropy H_0. This class of compressors are referred to as *memoryless* compressors. More sophisticated compressors, such as Lempel-Ziv compressors and derivatives, use knowledge from the already seen part of the input to compress the incoming symbols. They are slower than memoryless compressors but they achieve a much better compression ratio, which can be usually bounded in terms of the k-th order entropy of the input string for a large k [28].

It is proven in [5, Theorem 5.4] that the informal statement "the output of the BWT is highly compressible" can be formally restated saying that $bwt(s)$ can be compressed up to $H_k(s)$, for any $k > 0$, using any tool able to compress up to the zeroth-order entropy. In other words, after applying the BWT, we can achieve high order compression using a simple (and fast) memoryless compressor. This property is often referred to as the "boosting" property of the BWT. Another remarkable property of the BWT is that it can be used to build compressed indices. It is shown in [29] how to compute the number of occurrences of a pattern x in s in $\mathcal{O}(t_R|x|)$ time, where t_R is the cost of executing a rank query over $bwt(s)$. This result has spurred a great interest in data structures representing compactly a string x and efficiently supporting the queries rank, select, and access (return $x[i]$ given i, which is a nontrivial operation when x is represented in compressed form) and there are now many alternative solutions, with different trade-offs. In this paper, we assume a RAM model with word size w and an alphabet of size $\sigma = w^{\mathcal{O}(1)}$. Under this assumption, we make use of the following result (Theorem 5.1 in [30]):

Theorem 2.1. *Let s denote a string over an alphabet of size $\sigma = w^{\mathcal{O}(1)}$. We can represent s in $|s|H_0(s) + o(|s|)$ bits and support constant time* rank, select, *and* access *queries.*

The assumption $\sigma = w^{\mathcal{O}(1)}$ is only required to ensure that rank, select, and access take constant time; we use it to simplify the statements of our results. It is straightforward to verify that for larger alphabets our results still hold with the bounds on the running times multiplied by the largest of the cost of the above operations over the larger alphabets.

The properties of the BWT of being *compressible* and *searchable* combine nicely to give us *indexing capabilities* in *compressed space*. Indeed, combining a zeroth-order representation supporting rank, select, and access queries with the boosting property of the BWT, we obtain a full text self-index for s that uses space bounded by $|s|H_k(s) + o(|s|)$ bits for $k = o(\log_\sigma(n))$, see [29,31,11,32] for further details on these results and on the field of compressed data structures and algorithms that originated from this area of research.

2.1. Known BWT variants

We observed that the salient features of the Burrows-Wheeler transformation can be summarized as follows: **1**) it can be computed and inverted in linear time, **2**) it produces strings which are provably compressible in terms of the high order entropy of the input, **3**) it supports linear time pattern search directly on the transformed string. The *combination* of these three properties makes the BWT a fundamental tool for the design of compressed self-indices. Over the years, many variants of the original BWT have been proposed; in the following we review them, in roughly chronological order, emphasizing to what extent they share the features **1**–**3** mentioned above.

The original BWT is defined by sorting in lexicographic order all the cyclic rotations of the input string. In [33] Schindler proposes a *bounded context* transformation that differs from the BWT in the fact that the rotations are lexicographically sorted considering only the first ℓ symbols of each rotation. In [34,35] it has been shown that this variant satisfies properties **1**–**3**, with the limitation that the compression ratio can reach at maximum the ℓ-th order entropy and that it supports searches of patterns of length at most ℓ. Chapin and Tate [36] have experimented with computing the BWT using a different alphabet order. This simple variant still satisfies properties **1**–**3**, but it clearly does not bring any new theoretical insight. In the same paper, the authors also propose a variant in which rotations are sorted following a scheme inspired by reflected Gray codes. This variant shows some improvements in terms of compression, but it has never been analyzed theoretically and it does not seem to support property **3**.

A variant called Bijective BWT (BBWT), has been proposed in [37] as a transformation which is bijective even without assuming that the input string s be primitive. In this variant, the output consists of the last characters of the lexicographically sorted *cyclic* rotations of all factors of the Lyndon Factorization [38] of s. This variant has been recently shown in [39] to satisfy **1**, but being based on the cyclic rotations of the Lyndon factors property **2** has not been studied. As for property **3**, searching a pattern P directly on the (compressed) transformed string takes $\mathcal{O}(|P|\log|P|)$ time [40]. Note that the related variant Extended BWT [41], which takes as input a collection of strings, supports search in linear $\mathcal{O}(|P|)$ time for the problem of circular pattern matching [42–45,27,46].

More recently, some authors have proposed variants in which the lexicographic order is replaced by a different order relation. The interested reader can find relevant work in a recent review [47]. It turns out that these variants satisfy property **1** in part but nothing is known with respect to properties **2** and **3** (an outline of a substring matching algorithm is given in [48, Sec. 6] without any time analysis, but is based on substrings rather than single symbols). A major problem when using ordering substantially different from the lexicographic order is that the rotations prefixed by the same substring are not necessarily consecutive in the sorted matrix. For instance, if the cyclic rotations of Fig. 1 are sorted according to the V-order [49], the two rotations prefixed by ab are not consecutive in the ordering. Since all the BWT-based searching algorithms work by keeping track of the rows prefixed by larger and larger pattern substrings, the fact that these rows are not consecutive makes the design of an efficient algorithm extremely difficult.

To the best of our knowledge, the only non trivial BWT variant that fully satisfies properties **1**–**3** is the Alternating BWT (ABWT). This transformation has been derived in [12] starting from a result in combinatorics of words [50] characterizing the BWT as the inverse of a known bijection between words and multisets of primitive necklaces [51]. The ABWT is defined as the BWT except that when sorting rotation instead of the standard lexicographic order we use a different lexicographic order, called the *alternating* lexicographic order. In the alternating lexicographic order, the first character of each rotation is sorted according to the standard order of Σ (i.e., $a < b < c$). However, if two rotations start with the same character we compare their second characters using the reverse ordering (i.e., $c < b < a$) and so on, alternating the standard and reverse orderings in odd and even positions. Fig. 1 (right) shows how the rotations of an input string are sorted using the alternating ordering and the resulting ABWT.

The algorithmic properties of the BWT and ABWT are compared in [13,14]. It is shown that they can be both computed and inverted in linear time and that their main difference is in the definition of the LF-map, i.e. the correspondence between the characters in the first and last column of the sorted rotations matrix. In the original BWT, the i-th occurrence of a character c in the first column F corresponds to the i-th occurrence of c in the last column L, i.e., equal characters appear in the same relative order in F and L. Instead, in the ABWT, equal characters appear in the *reverse* order in F and L. That is, the i-th occurrence of c from the *top* in F corresponds to the i-th occurrence of c from the *bottom* in L. Since this modified LF-map can still be computed efficiently using rank operations, the ABWT can replace the BWT for the construction of self-indices. The experiments in [13] show that the ABWT has essentially the same compression performance of the BWT. However, we are not aware of any experiment using the ABWT for indexing purposes.

Note that in [13,14] the ABWT has been studied within a larger class of transformations in which the alphabet ordering depends on the position of the characters within any cyclic rotation. Although the "compression boosting" property holds for all transformations in this class, in [14] it is shown that the algorithmic techniques, based on the computation of the rank function on the transformed string, that allow us to invert BWT and ABWT in linear time cannot be applied to any other transformation in that class [14, Theorem 5.9]. Indeed, apart from the BWT and ABWT, for all the transformations studied in [14], the best inversion algorithm takes cubic time [14, Theorem 4.4]. In this paper we improve this result by providing a quadratic time algorithm for a general class of string transformations that includes the one studied in [13,14] (see Section 3.3).

2.2. Number of runs minimization

The problem of minimizing the number of runs in a transformed string has been studied initially for collections of (relatively short) strings because of the relevance of this setting for bioinformatics applications [52,53]. String collections are usually transformed with the Extended BWT [41] (EBWT) or with an EBWT variant in which a distinct end-marker is

$$\begin{array}{ccccccccccc}
 & F & & & & & & & & L & \\
 & \downarrow & & & & & & & & \downarrow & \\
1 & b & a & a & a & b & a & c & a & a & \\
2 & b & a & c & a & a & b & a & a & a & \\
3 & a & c & a & a & b & a & a & a & b & \\
4 & a & a & b & a & a & a & b & a & c & \leftarrow s \\
5 & a & a & b & a & c & a & a & b & a & \\
6 & a & a & a & b & a & c & a & a & b & \\
7 & a & b & a & a & a & b & a & c & a & \\
8 & a & b & a & c & a & a & b & a & a & \\
9 & c & a & a & b & a & a & a & b & a &
\end{array}$$

Fig. 2. The generalized BWT matrix for the string $s = aabaaabac$ computed using the orderings $\pi_\epsilon = (b, a, c)$, $\pi_a = (c, a, b)$, $\pi_{aa} = (b, a, c)$, $\pi_{aaba} = (a, c, b)$, and $\pi_x = (a, b, c)$ for every other substring x. The horizontal arrow marks the position of the original string s; the last column L is the output of the transformation. The range $R[aa] = [4, 3]$ of the three rows prefixed by aa are highlighted in gray.

appended to each string, making the collection ordered [54,55]. In [52] the authors introduce some heuristics for reordering the strings on-the-fly during the EBWT construction in order to reduce the number of runs. Experiments show that these heuristics yield a significant improvement in the overall compression. Recently, the authors of [56,57] extended these results obtaining an offline linear time algorithm for finding the string reordering yielding the minimum number of runs, and showing that the optimal reordering can reduce the number of runs by a factor $\Omega(\log_\sigma n)$, where n is the sum of the string lengths and σ is the alphabet size. The authors in [58] present the first tool that guarantees to output a BWT of a string collection with minimal number of runs, in terms of reordering of input strings. The authors of [59] consider instead the case in which the *same* end-marker is appended to each string and for this setting provide an $\mathcal{O}(\sigma n)$ time algorithm to find the string reordering that minimize the number of BWT runs. See [60] for a recent review of BWT variants for string collections and their properties with respect to the number of runs.

The problem of minimizing the number of runs in a single-string BWT has been studied in [57] where the authors prove that the decision problem of finding the alphabet permutation that minimizes the number of runs in the BWT is NP-complete and the optimization variant is APX-hard. Note that, by introducing new BWT variants, we are adding a new dimension to the run minimization problem: in addition to minimizing over alphabet or string reorderings, we can also minimize over a given class of BWT variants.

3. BWTs based on context adaptive alphabet orderings

In this section we introduce a class of string transformations that generalize the BWT in a very natural way. Given a primitive string s, as in the original BWT definition, we consider the matrix containing all its cyclic rotations. In the original BWT, the matrix rows are sorted according to the standard lexicographic order. We generalize this concept by sorting the rows using an ordering that *depends on their common context*, i.e., their longest common prefix. Formally, for each string x that prefixes two or more rows, we assume that an ordering π_x is defined on the symbols of Σ. When comparing two rows which are both prefixed by x, their relative rank is determined by the ordering π_x. Once the matrix rows have been ordered with this procedure, the output of the transformation is the last column of the matrix, as in the original BWT. Thus, these BWT variants are based on *context adaptive alphabet orderings*. For simplicity, in the following, we call them *context adaptive BWTs*.

We denote by $M_*(s)$ the matrix obtained using this generalized sorting procedure, and by $L = BWT_*(s)$ the last column of $M_*(s)$. Clearly L depends on s and the ordering used for each common prefix. Since we can arbitrarily choose an alphabet ordering for any substring x of s, and there are $\sigma!$ orderings to choose from, our definition includes a very large number of string transformations.

Example 3.1. In Fig. 2, the generalized BWT matrix for the string $s = aabaaabac$ is shown. The ordering associated to the empty string ϵ is $\pi_\epsilon = (b, a, c)$ so, among the rows that have no common prefix, first we have those starting with b, then those starting with a, and finally the one starting with c. Since $\pi_a = (c, a, b)$, among the rows which have a as their common prefix, first we have the ones in which c is the next letter, then the ones in which a is the next letter, followed by the ones in which b is the next letter. The complete ordering of the rows is established in a similar way on the basis of the orderings π_x. In particular, the ordering between any pair of rows is determined by π_x where x is their longest common prefix.

This class of transformations has been mentioned in [5, Sect. 5.2] under the name of *string permutations realized by a Suffix Tree* (the definition in [5] is slightly more general; for example it includes the bounded context BWT, which is not included in our class). Indeed, one can easily see that $L = BWT_*(s)$ can also be obtained by visiting the suffix tree of s in depth first order, except that when we reach a node u (including the root), we sort its outgoing edges according to their first characters using the permutation associated to the string u_x labeling the path from the root to u. During such a visit, each time we reach a leaf, we write the symbol associated to it: the resulting string is exactly $L = BWT_*(s)$ (see Fig. 3 (right)).

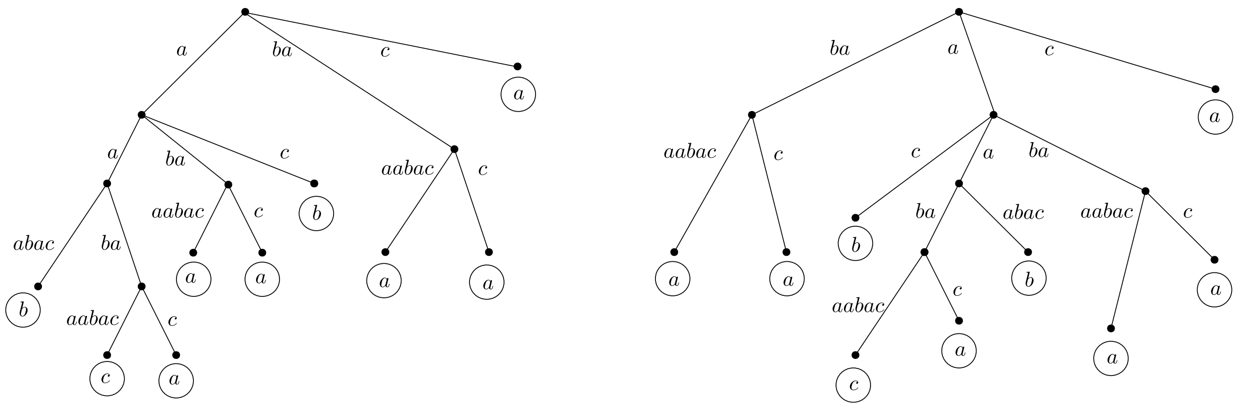


Fig. 3. Standard suffix tree for $s = aabaaabac$ with the symbol c used as a string terminator (left), and suffix tree with edges reordered using the same orderings of Figure 2 (right). To each leaf it is associated the symbol preceding in s the suffix spelled by that leaf. Note that reading left to right the symbols associated to each leaf gives $bwt(s)$ (left) and $BWT_*(s)$ (right).

Although in [5] the authors could not prove the invertibility of context adaptive transformations, which we do in Section 3.2, they observed that their relationship with the suffix tree has two important consequences: 1) they can be computed in $\mathcal{O}(n \log \sigma)$ time, and 2) they provably produce *highly compressible* strings, i.e., they have the "boosting" property of transforming a zeroth order compressor into a k-th order compressor.

Summing up, context adaptive transformations generalize the BWT in two important aspects: efficient computation (linear time in n) and compressibility. In [5], the only known instances of *reversible* suffix tree induced transformations were the original BWT and the bounded context BWT. In the following, we prove that *all* context adaptive BWTs defined above are invertible. Interestingly, to prove invertibility we first establish another important property of these transformations, namely that they can be used to count the number of occurrences of a pattern in s, which is another fundamental property of the original BWT.

We conclude this section by observing that both the BWT and ABWT belong to the class we have just defined. To get the BWT, we trivially define π_x to be the standard Σ ordering for every x, and to get the ABWT, we define π_x to be the standard Σ ordering for every x with $|x|$ even, and the reverse ordering for Σ for every x with $|x|$ odd.

3.1. Counting occurrences of patterns in context adaptive BWTs

Let $L = BWT_*(s)$ denote a context adaptive BWT. In the following, we assume that L is enriched with data structures supporting constant time rank queries as in Theorem 2.1. In this section we show that, given L and the set of alphabet permutations used to build $M_*(s)$, then we can determine in $\mathcal{O}(\sigma |x|^2)$ time the set of $M_*(s)$ rows prefixed by x, for any string x. We preliminarily observe that, by construction, this set of rows, if non-empty, form a contiguous range inside $M_*(s)$. This observation justifies the following definitions.

Definition 3.2. Given a string x, we denote by $R[x] = [b_x, \ell_x]$ the range of rows of $M_*(s)$ prefixed by x. More precisely, if $R[x] = [b_x, \ell_x]$, then row i is prefixed by x if and only if it is $b_x \le i < b_x + \ell_x$. If no rows are prefixed x, we set $R[x] = [0, 0]$. Note that ℓ_x is the number of occurrences of x in the circular string s.

For technical reasons, given x, we are also interested in the set of rows prefixed by the strings xc as c varies in Σ. Clearly, these sets of rows are consecutive in $M_*(s)$ and their union coincides with $R[x]$.

Definition 3.3. Given a string x, we denote by $R^*[x]$ the set of $\sigma + 1$ integers $[b_x, \ell_1, \ell_2, \ldots, \ell_\sigma]$ such that b_x is the lower extreme of $R[x]$ and, for $i = 1, \ldots, \sigma$, ℓ_i is the number of rows of $M_*(s)$ prefixed by xc_i.

Since $R[x]$ is the union of the ranges $R[xc]$, for $c \in \Sigma$, we have that, if $R^*[x] = [b_x, \ell_1, \ell_2, \ldots, \ell_\sigma]$, then $R[x] = [b_x, \sum_i \ell_i]$. Note also that the ordering of the ranges $R[xc]$ within $R[x]$ is determined by the permutation π_x. As observed in Section 2, we can assume that L supports constant time rank queries. This implies that, in constant time, we are also able to count the number of occurrences of a symbol c inside a substring $L[i, j]$.

Example 3.4. Let us consider the string $s = aabaaabab$ and the generalized BWT matrix illustrated in Fig. 2. Since $\pi_{aa} = (b, a, c)$, we have that $R^*[aa] = [4, 2, 1, 0]$, therefore $R[aa] = [4, 2 + 1 + 0] = [4, 3]$.

Lemma 3.5. *Given $R^*[x]$ and the permutation π_x, the set of values $R[xc_i]$ for all $c_i \in \Sigma$ can be computed in $\mathcal{O}(\sigma)$ time.*

Proof. If $R^*[x] = [b_x, \ell_1, \ell_2, \dots, \ell_\sigma]$, then $R[xc_i] = [b, \ell]$ with

$$b = b_x + \sum_{j : c_j <_{\pi_x} c_i} \ell_j, \qquad \ell = \ell_i \tag{1}$$

where the summation in (1) is done over all $j \in \{1, 2, \dots, \sigma\}$ such that c_j is smaller than c_i, according to the permutation π_x. □

Lemma 3.6. *Let $x = x_1 x_2 \cdots x_m$ be any length-m string with $m > 1$. Then, given $R^*[x_1 \cdots x_{m-1}]$ and $R^*[x_2 \cdots x_m]$, the set of values $R^*[x_1 \cdots x_m]$ can be computed in $\mathcal{O}(\sigma)$ time.*

Proof. By Lemma 3.5, given $R^*[x_1 \cdots x_{m-1}]$ and x_m, we can compute $R[x_1 \cdots x_m] = [b_x, \ell_x]$. In order to compute $R^*[x_1 \cdots x_m]$, we additionally need the number of rows prefixed by $x_1 x_2 \cdots x_m c$, for any $c \in \Sigma$. These numbers can be obtained by first computing the ranges $R[x_2 \cdots x_m c]$ using again Lemma 3.5. The number of rows prefixed by $x_1 x_2 \cdots x_m c$ can be obtained by counting the number of x_1 in the portions of L corresponding to each range $R[x_2 \cdots x_m c]$. The counting takes $\mathcal{O}(\sigma)$ time since we are assuming L supports constant time rank as in Theorem 2.1. □

Theorem 3.7. *Suppose we are given $BWT_*(s)$ with constant time rank support, and the set of permutations used to compute the matrix $M_*(s)$. Then, given any string $x = x_1 x_2 \cdots x_p$, the range of rows $R[x]$ prefixed by x can be computed in $\mathcal{O}(\sigma p^2)$ time and $\mathcal{O}(\sigma p)$ space.*

Proof. We need to compute $R[x_1 x_2 \cdots x_p]$. To this end we consider the following scheme, inspired by the Newton finite difference formula:

$$\begin{array}{llllll}
R^*[x_1] & R^*[x_1x_2] & R^*[x_1x_2x_3] & \cdots & R^*[x_1x_2\cdots x_{p-1}] & R^*[x_1x_2\cdots x_p] \\
R^*[x_2] & R^*[x_2x_3] & R^*[x_2x_3x_4] & \cdots & R^*[x_2\cdots x_p] & \\
R^*[x_3] & R^*[x_3x_4] & \cdots & & & \\
\vdots & & & & & \\
R^*[x_p] & & & & &
\end{array}$$

Using Lemma 3.6, we can compute $R^*[x_i \cdots x_j]$ given $R^*[x_i \cdots x_{j-1}]$ and $R^*[x_{i+1} \cdots x_j]$. Thus, from two consecutive entries in the same column, we can compute one entry in the following column. To compute $R[x_1 x_2 \cdots x_p]$ we can for example perform the computation bottom-up, proceeding row by row. In this case, we are essentially computing the ranges corresponding to x_p, $x_{p-1}x_p$, $x_{p-2}x_{p-1}x_p$ and so on, in a sort of backward search. However, we can also perform the computation top down, diagonal by diagonal, and in this case, we are computing the ranges corresponding to x_1, x_1x_2, and so on, up to $x_1 \cdots x_p$. In both cases, the information one needs to store from one iteration to the next is $\mathcal{O}(p)$ $R^*[\cdot]$ values, which take $\mathcal{O}(\sigma p)$ words. By Lemma 3.6, the computation of each value takes $\mathcal{O}(\sigma)$ time, so the overall complexity is $\mathcal{O}(\sigma p^2)$ time. □

Example 3.8. For the transformation described in Fig. 2, the computation of Theorem 3.7 for computing $R[aba]$ works as follows

$$\begin{array}{lll}
R^*[a] = [3,1,3,2] & R^*[ab] = [7,2,0,0] & R^*[aba] = [7,1,0,1] \\
R^*[b] = [1,2,0,0] & R^*[ba] = [1,1,0,1] & \\
R^*[a] = [3,1,3,2] & &
\end{array}$$

Finally, since $R^*[aba] = [7,1,0,1]$, we have $R[aba] = [7, 1+0+1] = [7,2]$.

Note that our scheme for the computation of $R[x]$ is based on the computation of $R^*[y]$ for $\mathcal{O}(p^2)$ substrings y of x. If x has many repetitions, the overall cost could be less than quadratic. In the extreme case, $x = a^p$, $R[x]$ can be computed in $\mathcal{O}(\sigma p)$ time.

3.2. Inverting context adaptive BWTs

We now show that the machinery we set up for counting occurrences can be used to retrieve s given $BWT_*(s)$, thus to invert any context adaptive BWT.

Lemma 3.9. *Given $R^*[x] = [b_x, \ell_1, \ell_2, \dots, \ell_\sigma]$ and a row index i with $b_x \le i < b_x + \sum_{j=1}^{\sigma} \ell_j$, the $(|x|+1)$-st character of row i can be computed in $\mathcal{O}(\sigma)$ time.*

$$
\begin{array}{cccccccccccc}
 & F & & & & & & & & L & \\
 & \downarrow & & & & & & & & \downarrow & \\
1 & c & a & a & b & a & a & a & b & a & \\
2 & a & b & a & c & a & a & b & a & a & \\
3 & a & b & a & a & a & b & a & c & a & \\
4 & a & c & a & a & b & a & a & a & b & \\
5 & a & a & b & a & c & a & a & b & a & \\
6 & a & a & b & a & a & a & b & a & c & \leftarrow s \\
7 & a & a & a & b & a & c & a & a & b & \\
8 & b & a & a & a & b & a & c & a & a & \\
9 & b & a & c & a & a & b & a & a & a &
\end{array}
$$

Fig. 4. The BWT_K matrix for the string $s = aabaaabac$ computed using the triple of alphabet permutations $K = \{(c,a,b),(b,c,a),(b,a,c)\}$. The horizontal arrow marks the position of the original string s; the last column L is the output of the transformation.

Proof. Let $\rho_1,\dots,\rho_\sigma$ denote the alphabet symbols reordered according to permutation π_x, and let $\ell'_1,\dots,\ell'_\sigma$ denote the values $\ell_1,\dots,\ell_\sigma$ reordered according to the same permutation. Since $i \in R[x]$, row i is prefixed by x. Since the rows prefixed by x are sorted in their $(|x|+1)$-st position according to π_x, the $(|x|+1)$-st symbol of row i is the symbol ρ_j such that

$$b_x + \sum_{1\le h<j} \ell'_h \le i < b_x + \sum_{1\le h\le j} \ell'_h \quad \square$$

Theorem 3.10. *Given $BWT_*(s)$ with constant time rank support, the permutations π_x used to build the matrix $M_*(s)$, and the row index I containing s in $M_*(s)$, the original string s can be recovered in $\mathcal{O}(\sigma|s|^2)$ time and $\mathcal{O}(\sigma|s|)$ working space.*

Proof. Let $s = s_1 s_2 \cdots s_n$. From $BWT_*(s)$, in $\mathcal{O}(n)$ time we retrieve the number of occurrences of each character in s and hence the ranges $R[c_1]$, $R[c_2]$, …, $R[c_\sigma]$. From those and the row index I, we retrieve the first character of s, i.e. s_1. Next, counting the number of occurrences of s_1 in the ranges of $BWT_*(s)$ corresponding to $R[c_1]$, $R[c_2]$, …, $R[c_\sigma]$, we compute $R^*[s_1]$. Finally, we show by induction that, for $m = 1,\dots,n-1$, given $R^*[s_1 s_2 \cdots s_m]$, we can retrieve s_{m+1} and $R^*[s_1 s_2 \cdots s_{m+1}]$ in $\mathcal{O}(m\sigma)$ time. By using Lemma 3.9, from $R^*[s_1 s_2 \cdots s_m]$ and i we retrieve s_{m+1}. Next, assuming we maintained the ranges $R^*[s_j \cdots s_m]$, for $j = 1,\dots,m$ we can compute $R^*[s_j \cdots s_{m+1}]$ adding one diagonal to the scheme shown in the proof of Theorem 3.7. By Lemma 3.6, the overall cost is $\mathcal{O}(\sigma|s|^2)$ time as claimed. $\square$

The above theorem establishes that all context adaptive BWTs are invertible. Note that in our definition, the alphabet ordering π_x associated to x can depend on the whole string x; in this sense the context has full memory. We consider this an important conceptual result. However, from a practical point of view, a transformation whose definition requires the specification of $\mathcal{O}(|s|)$ alphabet permutations appears rather cumbersome. For this reason, in the following, we consider some subclasses of transformations in which the permutations associated to each prefix are defined by "simple" rules. We show that, in some cases, nontrivial generalized BWTs have simpler and more efficient inversion algorithms.

3.3. Special case: ordering based on context length

In [13], the authors introduced a set of generalized transformations that turns out to be a subclass of context adaptive BWTs. Given a k-tuple of alphabet permutations $K = (\pi_0, \pi_1, \dots, \pi_{k-1})$, using the notation of this paper, the transformation BWT_K in [13] is defined as the context adaptive BWT in which the permutation π_x associated to the string x is π_ℓ where $\ell = |x| \bmod k$. Hence, the permutation associated to each string only depends on its depth, and the k-tuple $K = (\pi_0, \pi_1, \dots, \pi_{k-1})$ completely determines the transformation. In [13,14], it was shown that for every K the transformation BWT_K can be inverted in $\mathcal{O}(|s|^3)$ time; Theorem 3.10 therefore constitutes an alternative faster inversion algorithm. To our knowledge, no faster algorithm is known, even for $k = 2$, when the whole transformation depends on just two alphabet permutations.

Example 3.11. Let us consider $k = 3$ and the k-tuple of alphabet permutations $K = (\pi_0, \pi_1, \pi_2)$, where $\pi_0 = (c,a,b)$, $\pi_1 = (b,c,a)$ and $\pi_2 = (b,a,c)$. The BWT_K matrix for the string $s = aabaaabac$ is shown in Fig. 4. If x is the longest common prefix between two rows, their ordering depends on the respective characters at the $(|x|+1)$-th position, according to the permutation $\pi_{|x| \bmod 3}$. For instance, $abacaabaa < abaaabaca$, since aba is the longest common prefix and $c < a$, according to π_0.

3.4. Special case: ± ordering

Given a permutation π of the alphabet Σ, we denote by π^R the reversal of π, that is, the permutation such that, for each pair of symbols c_i, c_j in Σ,

$$\begin{array}{c|ccccccccc|c}
 & F & & & & & & & & L & \\
 & \downarrow & & & & & & & & \downarrow & \\
1 & b & a & a & a & b & a & c & a & a & \\
2 & b & a & c & a & a & b & a & a & a & \\
3 & a & c & a & a & b & a & a & a & b & \\
4 & a & a & b & a & c & a & a & b & a & \\
5 & a & a & b & a & a & a & b & a & c & \leftarrow s \\
6 & a & a & a & b & a & c & a & a & b & \\
7 & a & b & a & a & a & b & a & c & a & \\
8 & a & b & a & c & a & a & b & a & a & \\
9 & c & a & a & b & a & a & a & b & a &
\end{array}$$

Fig. 5. Example of a transformation based on a $\pm$ ordering. Let $\pi = (b, a, c)$ so that $\pi^R = (c, a, b)$. The generalized BWT matrix $M_*(s)$ for the string $s = aabaaabac$ is computed using the following permutations $\pi_\epsilon = \pi$, $\pi_a = \pi^R$, $\pi_{aa} = \pi$, $\pi_{aaba} = \pi^R$, and $\pi_x = \pi$ for every other substring x. The horizontal arrow marks the position of the original string s; the last column L is the output of the transformation. The range $R[aba] = [7, 2]$ of the rows prefixed by aba, i.e. 7 and 8, are colored in cyan. In light brown, the row 2 of the range $R[bac] = [2, 1]$. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

$$\pi^R(c_i) < \pi^R(c_j) \quad \Longleftrightarrow \quad \pi(c_i) > \pi(c_j).$$

Let π denote an arbitrary permutation of Σ. We consider the subclass of context adaptive transformations in which the permutation π_x associated to each substring x can be either π or its reversal π^R. Once π is established, for each string x we only need an additional bit to specify the ordering π_x. So, for every string x and character c, checking if $\pi_x = \pi_{xc}$ can be done in constant time. For this subclass we say that the matrix $M_*(s)$ is based on a $\pm$ *ordering* (see example in Fig. 5). In this section, we show that, for the transformations in this subclass, the space and time for inversion can be reduced by a factor σ.

Lemma 3.12. *Let $M_*(w)$ be based on a $\pm$ ordering, and let $x = x_1x_2 \cdots x_m$ be any length-m string, with $m > 1$. Then, given $R[x_1 \cdots x_{m-1}]$, $R[x_2 \cdots x_m]$ and $R[x_2 \cdots x_{m-1}]$, we can compute $R[x_1 \cdots x_m]$ in $\mathcal{O}(1)$ time assuming the permutation π or π^R associated to each substring is known, and $BWT_*(s)$ has constant time rank support.*

Proof. Let $y = x_1 \cdots x_{m-1}$ and $z = x_2 \cdots x_{m-1}$. Recall that there is a bijection between the rows in $R[y]$ and the rows in $R[z]$ whose last symbol is x_1. We exploit this bijection to find $R[yx_m]$ given $R[zx_m]$. The size of $R[yx_m]$ is equal to the number of rows in $R[zx_m]$ ending with x_1, so to completely determine $R[yx_m]$ we just need to compute how many rows in $R[y]$ precede $R[yx_m]$. If $\pi_y = \pi_z$ this number is equal to the number of rows in $R[z]$ *above* $R[zx_m]$ and ending with x_1. If $\pi_y \neq \pi_z$, since necessarily $\pi_y = \pi_z^R$, this number is equal to the number of rows in $R[z]$ *below* $R[zx_m]$ and ending with x_1. Since counting the number of rows ending in x_1 in a given range can be done using rank operations on $BWT_*(s)$, the lemma follows. □

Example 3.13. Consider again the example of Fig. 5. Let $x = abac$, so $y = aba$ and $z = ba$. It is $R[y] = [7, 2]$, $R[z] = [1, 2]$ and $R[zc] = [2, 1]$. The rows of $R[y]$ are highlighted in cyan, the rows of $R[zc]$ are highlighted in light brown in Fig. 5. The number of rows in $R[zc] = [2, 1]$ ending with a is 1. This a number gives the size of $R[yc] = R[abac]$. Moreover, since $\pi_y = \pi_z = \pi$, we need to count the number of rows in $R[z] = [1, 2]$ above $R[zc] = [2, 1]$ and ending with a. This a number is 1. This means that one row in $R[y] = [7, 2]$ precedes $R[yc] = R[abac]$. Hence, $R[abac] = [8, 1]$. If we consider $x = aa$, it is $y = a$ and $z = \epsilon$, $R[y] = R[a] = R[za] = [3, 6]$, $R[z] = R[\epsilon] = [1, 9]$. The number of rows in $R[za] = [3, 6]$ ending with a is 3. So, 3 is the size of $R[aa]$. Since $\pi_a \neq \pi_\epsilon$, we have to count the number of rows in $R[z] = [1, 9]$ below $R[za] = [3, 6]$ and ending with a. Such a number is 1. This means that there is 1 row in $R[y] = R[a] = [3, 6]$ that precedes $R[ya] = R[aa]$, hence $R[aa] = [4, 3]$.

Lemma 3.14. *Suppose $M_*(s)$ is based on a $\pm$ ordering. Given $BWT_*(s)$ with constant time rank support, and any string $x = x_1x_2 \cdots x_p$, we can compute the range of rows prefixed by x in $\mathcal{O}(p^2)$ time and $\mathcal{O}(p)$ space.*

Proof. We reason as in the proof of Theorem 3.7, except that because of Lemma 3.12 we work with $R[\cdot]$ instead of $R^*[\cdot]$. The thesis follows observing that each entry takes $\mathcal{O}(1)$ space and can be computed in $\mathcal{O}(1)$ time. □

Theorem 3.15. *Suppose $M_*(s)$ is based on a $\pm$ ordering. Given $BWT_*(s)$ with constant time rank support and the row index i containing s in $M_*(s)$, we can retrieve the original string s in $\mathcal{O}(|s|^2)$ time and $\mathcal{O}(|s|)$ working space.*

$$\begin{array}{cccccccccccc}
 & F & & & & & & & & L & \\
 & \downarrow & & & & & & & & \downarrow & \\
1 & b & a & a & a & b & a & c & a & a & \\
2 & b & a & c & a & a & b & a & a & a & \\
3 & c & a & a & b & a & a & a & b & a & \\
4 & a & b & a & a & a & b & a & c & a & \\
5 & a & b & a & c & a & a & b & a & \underline{a} & \\
6 & a & a & b & a & a & a & b & a & \underline{c} & \leftarrow s \\
7 & a & a & b & a & c & a & a & b & \underline{a} & \\
8 & a & a & a & b & a & c & a & a & b & \\
9 & a & c & a & a & b & a & a & a & \underline{b} &
\end{array}$$

Fig. 6. The local ordering BWT matrix for the string $s = aabaaabac$ computed using the orderings $\pi_\epsilon = (b, c, a)$, $\pi_a = (b, a, c)$, $\pi_b = \pi_c = (a, b, c)$. Here, the alphabet orderings associated to each non-empty string depend only on the last symbol of the string, i.e. $k = 1$. The horizontal arrow marks the position of the original string s; the last column L is the output of the transformation. The rows starting with ba (highlighted in cyan and light brown, respectively) are in a order-preserving correspondence to the rows starting with a and ending with b. The last character of each BWT-run is underlined.

4. BWTs based on local orderings

In this section, we consider the context adaptive transformations in which the alphabet ordering π_x associated to each string x only depends on the last k symbols of x, where k is fixed. In the following, we refer to these string transformations as *BWTs based on local orderings*. We show that local ordering transformations have properties very similar to the original BWT, since they can be inverted in linear time and also support the search of a pattern in the original text in time proportional to the pattern length.

We start by analyzing the case $k = 1$. For such local ordering transformations, the matrix $M_*(s)$ depends on only $\sigma + 1$ alphabet orderings: one for each symbol plus the one used to sort the first column of $M_*(s)$. See Fig. 6 for an example. The following lemma establishes an important property of local ordering transformations.

Lemma 4.1. *If $M_*(s)$ is based on a local ordering with $k = 1$, then for any pair of characters x_1 and x_2, there is an order-preserving bijection between the set of rows starting with x_1x_2 and the set of rows starting with x_2 and ending with x_1.*

Proof. Note that both sets of rows contain a number of elements equal to the number of occurrences of x_1x_2 in the circular string s. In the following, we write $s[i\cdots]$ to denote the cyclic rotation of s starting with $s[i]$. Assume that rotations $s[i\cdots]$ and $s[j\cdots]$ both start with x_2 and end with x_1 and let $h > 1$ denote the first column in which the two rotations differ. Rotation $s[i\cdots]$ precedes $s[j\cdots]$ in $M_*(s)$ if and only if $s[i+h-1]$ is smaller than $s[j+h-1]$ according to the alphabet ordering associated to symbol $s[i+h-2] = s[j+h-2]$. The two rotations $s[i-1\cdots]$ and $s[j-1\cdots]$ both start with x_1x_2 and their relative position also depends on the relative ranks of $s[i+h-1]$ and $s[j+h-1]$ according to the alphabet ordering associated to symbol $s[i+h-2] = s[j+h-2]$. Hence the relative order of $s[i-1\cdots]$ and $s[j-1\cdots]$ is the same as the one of $s[i\cdots]$ and $s[j\cdots]$. $\square$

Example 4.2. Consider the string $s = aabaaabac$ and the local ordering BWT computed using the orderings $\pi_\epsilon = (b, c, a)$, $\pi_a = (b, a, c)$, $\pi_b = \pi_c = (a, b, c)$ by using $k = 1$. As proved in Lemma 4.1, the bijection between the set of rows 1 and 2 starting with ba (highlighted in cyan and light brown, respectively) and the set of rows 8 and 9 starting with a and ending with b is order-preserving. Analogously, the bijection mapping the rows 5, 6, and 7 starting with aa to the rows 4, 5, and 7, respectively.

Armed with the above lemma, we now show that for local ordering transformations we can establish much stronger results than the ones provided in Section 3.1.

Lemma 4.3. *Suppose $BWT_*(s)$ is based on a local ordering with $k = 1$ and supports constant time rank queries. Let $x = x_1x_2\cdots x_m$ be any length-m string with $m > 1$. Then, given $R[x_1x_2]$, $R[x_2]$ and $R[x_2\cdots x_m]$, the value $R[x_1\cdots x_m]$ can be computed in $\mathcal{O}(1)$ time.*

Proof. By Lemma 4.1, there is an order preserving bijection between the rows in $R[x_1x_2]$ and those in $R[x_2]$ ending with x_1. In this bijection, the rows in $R[x_1\cdots x_m]$ correspond to those in $R[x_2\cdots x_m]$ ending with x_1. Because of this bijection, if, among the ordered set of rows starting with x_2 and ending with x_1, those prefixed by $x_2\cdots x_m$ are in positions $r+1, \ldots, r+h$, then, among the rows starting with x_1x_2, those prefixed by $x_1x_2\cdots x_m$ are *consecutive* and in positions $r+1, \ldots, r+h$. The lemma follows since if $R[x_2] = [b, \ell]$ and $R[x_2\cdots x_m] = [b', \ell']$, we have

$$r = \mathsf{rank}_{x_1}(L, b'-1) - \mathsf{rank}_{x_1}(L, b-1), \qquad h = \mathsf{rank}_{x_1}(L, b'+\ell'-1) - \mathsf{rank}_{x_1}(L, b'-1),$$

and, if $R[x_1x_2] = [\bar{b}, \bar{\ell}]$, it is $R[x_1\cdots x_m] = [\bar{b}+r, h]$. $\square$

Theorem 4.4. *Suppose $BWT_*(s)$ is based on a local ordering with $k = 1$ and supports constant time rank queries. After a $\mathcal{O}(\sigma^2)$ time preprocessing, given any string $x = x_1x_2 \cdots x_p$, the range of rows prefixed by x can be computed in $\mathcal{O}(p)$ time and $\mathcal{O}(\sigma^2 + p)$ space.*

Proof. We reason as in the proof of Theorem 3.7, except that because of Lemma 4.3 we can work with $R[\cdot]$ instead of $R^*[\cdot]$ and we only need to compute the first two columns and the diagonal. In the preprocessing step, we compute $R[c_i]$ and $R[c_ic_j]$ for any pair $(c_i, c_j) \in \Sigma^2$. During the search phase, we compute each diagonal entry in constant time. □

Another immediate consequence of Lemma 4.1 is that we can efficiently "move back in the text" as in the original BWT. Note this operation is the base for BWT inversion and for snippet extraction and locate operations on FM-indices [29].

Lemma 4.5. *Suppose $BWT_*(s)$ is based on a local ordering with $k = 1$ and supports constant time rank and access queries. Then, after a $\mathcal{O}(\sigma^2)$ time preprocessing, given a row index i we can compute in $\mathcal{O}(1)$ time the index of the row obtained from the i-th row with a circular right shift by one position.*

Proof. It suffices to compute the first and last symbol of row i and then apply Lemma 4.1. □

Corollary 4.6. *If $BWT_*(s)$ is based on a local ordering with $k = 1$ and supports constant time rank and access queries, $BWT_*(s)$ can be inverted in $\mathcal{O}(\sigma^2 + |s|)$ time and $\mathcal{O}(\sigma^2)$ working space.*

The notion of local ordering can be generalized to contexts of size $k > 1$. The resulting transformations depend on $1 + \sigma + \sigma^2 + \cdots + \sigma^k$ alphabet permutations, one for each string over Σ of length at most k. Once these permutations have been chosen, with the notation of Section 3, we consider the context adaptive transformations in which the alphabet ordering π_x associated to each string x only depends on the last k symbols of x, or to the last $|x|$ symbols if $|x| < k$. Lemma 4.1 can be generalized to show that, for any $(k+1)$-tuple $x_1, \ldots, x_{k+1}$, there is an order preserving bijection between the rows prefixed by $x_2 \cdots x_{k+1}$ and ending with x_1 and the rows prefixed by $x_1 \cdots x_{k+1}$. As a consequence, search and inversion can still be performed in linear time with the only difference that the preprocessing phase now takes $\mathcal{O}(\sigma^{k+1})$ time and space, since we need to compute and store the values $R[x]$ for all strings x of length up to $k + 1$.

4.1. Local orderings and r-index

The r-index [15] is a recent variant of the FM-index, still based on the BWT, introduced to efficiently support the locate operation on highly repetitive collections. The locate operation is the task of determining the positions, in the original input s, of all the occurrences of the pattern. This is usually done storing a subset of the Suffix Array entries (these are called Suffix Array samples). The main feature of the r-index is that it supports the locate operation using a number of Suffix Array samples equal to the number of runs in the BWT. If the input contains many repetitions such number is much smaller than the number of Suffix Array samples used by the standard FM-index, usually $\Theta(n/\log^c n)$ for some constant $c > 1$.

In this section we show that it is possible to implement an r-index also on the top of a BWT based on a local ordering. We start considering the case $k = 1$. The crucial ingredient of the r-index is the so called Toehold Lemma: this result ensures that when we search for a pattern x, in addition to the range of rows prefixed by x, we also obtain the position in s of at least one occurrence of x (assuming such occurrence exists). To prove the Toehold Lemma for local orderings, we proceed as in Lemma 2 in [61] and we logically mark every character in L which is the last character in a BWT-run, and we store the position in s of the marked characters. In addition, for each character c, we store the position in s of the last row prefixed by c. This means that the last character of each run in F is augmented with the position in s of that suffix representing this character.

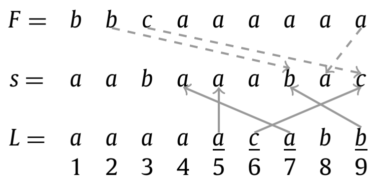
Lemma 4.7. *Let $x = x_1 \cdots x_p$ be a string that occurs in s. Then, using a BWT based on local ordering, in $\mathcal{O}(p)$ time we can compute, in addition to the range $R[x_1 \cdots x_p]$ of rows prefixed by x, the position in s of the last row in such a range.*

Proof. We proceed by induction on $j = p, p-1, \ldots, 1$. For $j = p$ the position in s of the last row prefixed by x_p is obtained by $R[x_p]$, which is computed in the preprocessing phase of Theorem 4.4. For $j < p$, assume that we know the range $R[x_{j+1} \cdots x_p]$ and the position pos_{j+1} in s of the last row in that range. Since there is an order preserving bijection between the rows in $R[x_{j+1} \cdots x_p]$ ending with x_j and the rows in $R[x_j \cdots x_p]$, the last row in $R[x_j \cdots x_p]$ corresponds to the last row in $R[x_{j+1} \cdots x_p]$ ending with x_j. If the latter coincides with the last row in $R[x_{j+1} \cdots x_p]$, i.e. x_j is the last symbol in the portion of L corresponding to $R[x_{j+1} \cdots x_p]$, then the position of the last row in $R[x_j \cdots x_p]$ in s is simply $\mathsf{pos}_{j+1} - 1$. If x_j is not the last symbol in the portion of L corresponding to $R[x_{j+1} \cdots x_p]$, then the last x_j in that range will be the last of a BWT-run and therefore will be a marked position. Its position in s will be among the ones we stored, and this will coincide with the position of the last row in $R[x_j \cdots x_p]$. □

Example 4.8. Let us consider the following string s and the column L of its local ordering BWT-matrix, as shown in Fig. 6. The column F of the matrix is here also reported.

$$
\begin{array}{r|ccccccccccc|l}
 & \overset{F}{\downarrow} & & & & & & & & & & \overset{L}{\downarrow} & \\
1 & a & a & a & b & a & a & b & a & a & c & b & \\
2 & a & a & b & a & a & b & a & a & c & b & a & \\
3 & a & a & b & a & a & c & b & a & a & a & b & \\
4 & a & a & c & b & a & a & a & b & a & a & b & \\
5 & a & b & a & a & b & a & a & c & b & a & a & \\
6 & a & b & a & a & c & b & a & a & a & b & a & \\
7 & a & c & b & a & a & a & b & a & a & b & a & \\
8 & c & b & a & a & a & b & a & a & b & a & a & \\
9 & b & a & a & c & b & a & a & a & b & a & a & \\
10 & b & a & a & a & b & a & a & b & a & a & c & \leftarrow s \\
11 & b & a & a & b & a & a & c & b & a & a & a &
\end{array}
\qquad
\begin{array}{r|ccccccccccc|l}
 & \overset{F}{\downarrow} & & & & & & & & & & \overset{L}{\downarrow} & \\
1 & a & a & a & b & a & a & b & a & a & c & b & \\
2 & a & a & b & a & a & b & a & a & c & b & a & \\
3 & a & a & b & a & a & c & b & a & a & a & b & \\
4 & a & a & c & b & a & a & a & b & a & a & b & \\
5 & a & b & a & a & b & a & a & c & b & a & a & \\
6 & a & b & a & a & c & b & a & a & a & b & a & \\
7 & a & c & b & a & a & a & b & a & a & b & a & \\
8 & c & b & a & a & a & b & a & a & b & a & a & \\
9 & b & a & a & a & b & a & a & b & a & a & c & \leftarrow s \\
10 & b & a & a & b & a & a & c & b & a & a & a & \\
11 & b & a & a & c & b & a & a & a & b & a & a &
\end{array}
$$

Fig. 7. The generalized BWT matrix for the string $s = baaabaabaac$ (left) computed using the orderings $\pi_\epsilon = (a, c, b)$, $\pi_{baa} = (c, a, b)$, and $\pi_x = (a, b, c)$ for every other substring x. The local ordering BWT matrix for the same string (right) using the orderings $\pi_\epsilon = (a, c, b)$, $\pi_a = \pi_b = \pi_c = (a, b, c)$. In both matrices the horizontal arrow marks the position of the original string s, and the last column L is the output of the transformation.



The local ordering BWT matrix for s is computed using the ordering $\pi_\epsilon = (b, c, a)$, $\pi_a = (b, a, c)$, $\pi_b = \pi_c = (a, b, c)$, as reported in Fig. 6. The last character of each BWT-run in L is underlined. They are $L[5]$, $L[6]$, $L[7]$ and $L[9]$, and the corresponding positions in s are 5, 9, 4 and 7, respectively. The correspondence between each underlined symbol in L and its position in L is represented by an arrow. In addition, for each character c, we store the position in s of the last row prefixed by c. For each of such rows, the correspondent position in s is highlighted by a dashed arrow from the column F to s. Such positions in s are 7 (row 2), 9 (row 3), and 8 (row 9).

Let us consider the string $x = baa$. For $j = 3$, the position $p_3 = 8$ in s of the last row prefixed by $x_3 = a$ is obtained by $R[a] = [4, 6]$ (i.e., the row 9). For $j = 2$, we know that there is an order preserving bijection between the rows in $R[a] = [4, 6]$ ending with a and the rows in $R[aa] = [6, 3]$. Since $x_2 = a$ *is not* the last symbol in the portion of L corresponding to $R[a]$, then the last a in that range (i.e., $L[7]$) is in a marked position since it is the last symbol of a BWT-run, and its position $p_2 = 4$ in s is precisely the position in s of the last row of the range $R[aa]$. For $j = 1$, we know that there is an order preserving bijection between the row in $R[aa] = [6, 3]$ ending with b and the row in $R[baa] = [1, 1]$. Since $x_3 = b$ *is* the last symbol in the portion of L corresponding to $R[aa]$ (i.e. $L[8]$), then its position in s is $p_1 = p_2 - 1 = 4 - 1 = 3$, which is also the position of the last, although unique, row in the range $R[baa]$.

In addition to the Toehold Lemma, the only other ingredients of the r-index are 1) the predecessor data structure $P^{\pm}$ from Lemma 3.5 in [15], which is not related to the BWT, and 2) the property that if two consecutive rows of the BWT matrix start and end with the same symbols, then rotating them rightward by one position we get two new rows which are still consecutive and in the same relative order. This property is valid for local orderings, even if it not valid for the general class of context adaptive alphabet orderings, as shown in the following example.

Example 4.9. Fig. 7 shows a generalised BWT matrix based on context adaptive alphabet orderings (on the left) and a local ordering BWT matrix (on the right) for the string $s = baaabaabaac$. Let us consider the rows 3 and 4 of the generalised BWT matrix on the left, both starting with a and ending with b, highlighted in cyan and light brown, respectively. It can be verified that, if we rotate them rightward we get rows 11 and 9, respectively, which are no longer consecutive nor in the same relative order. Instead, let consider the rows 3 and 4 of the local ordering BWT matrix, both starting with a and ending with b, highlighted in cyan and light brown, respectively. If we rotate them rightward we obtain the rows 10 and 11, respectively, which are consecutive and in the same relative order.

For the local orderings with $k > 1$ the above arguments can be generalized with the limitation that the length of the searched pattern must be at least k. The main modification is that we need to store the range $R[y]$ for all strings y of length k that occur in s. For each such range we also store the suffix array sample for the last row in the range for an overall extra space of $\mathcal{O}(\sigma^k)$. To search a pattern $x = x_1 \cdots x_p$ we consider the length-k suffix $x' = x_{p-k+1} \cdots x_p$ and we retrieve its range $R[x']$ and the position in s of the last row in the range. With this information we then use Lemma 4.7 to retrieve in $p - k$ steps the range $R[x_1 \cdots x_p]$ and the position in s of the last row of such range.

$$M(S)$$

$$\begin{array}{cccccccccccc}
 & F & & & & & & & & L & \\
 & \downarrow & & & & & & & & \downarrow & \\
1 & ba & aa & aa & ab & ba & ac & ca & aa & ab & \\
2 & ba & ac & ca & aa & ab & ba & aa & aa & ab & \\
3 & ca & aa & ab & ba & aa & aa & ab & ba & ac & \\
4 & ab & ba & aa & aa & ab & ba & ac & ca & aa & \\
5 & ab & ba & ac & ca & aa & ab & ba & aa & aa & \\
6 & aa & ab & ba & aa & aa & ab & ba & ac & ca & \leftarrow S \\
7 & aa & ab & ba & ac & ca & aa & ab & ba & aa & \\
8 & aa & aa & ab & ba & ac & ca & aa & ab & ba & \\
9 & ac & ca & aa & ab & ba & aa & aa & ab & ba &
\end{array}$$

Fig. 8. The BWT matrix for the string $S = aa\,ab\,ba\,aa\,aa\,ab\,ba\,ac\,ca$ computed using the ordering Π based on the local orderings $\pi_\epsilon = (b, c, a)$, $\pi_a = (b, a, c)$, $\pi_b = \pi_c = (a, b, c)$. The concatenation of the pairs in the last column of the matrix gives $bwt(S) = ab\,ab\,ac\,aa\,aa\,ca\,aa\,ba\,ba$. The string obtained by concatenating the first symbol of each pair in $bwt(S) = L$ (colored in red) is $BWT_*(s) = aaaaacabb$, where $s = aabaaabac$, as shown in Fig. 6.

4.2. An alternative view of local orderings

We conclude this section showing an alternative way to derive transformations based on local orderings. Consider for simplicity the case $k = 1$ and assume the transformation BWT_* is defined by the $\sigma + 1$ orderings π_ϵ and π_c for $c \in \Sigma$. Consider now the ordering Π over $\Sigma^2 = \Sigma \times \Sigma$ defined as follows: Given the pairs x_1x_2, y_1y_2 in Σ^2 it is $(x_1x_2 <_\Pi y_1y_2)$ iff $(x_1 \neq y_1,\ x_1 <_{\pi_\epsilon} y_1)$, or $(x_1 = y_1 = c,\ x_2 <_{\pi_c} y_2)$. We now show that BWT_* is equivalent to the original BWT over the alphabet Σ^2 ordered according to Π. To each string s, we associate a new string S over Σ^2, defined by $S[i] = s[i]s[i+1]$ with indices taken modulo $|s|$. Such an approach is described in Example 4.10 in which the strings $s = aabaaabac$ and $S = aa\,ab\,ba\,aa\,aa\,ab\,ba\,ac\,ca$ are considered.

There is a natural correspondence between rotations of s and S, and because of the definition of Π, the ordering of s's rotations in $M_*(s)$ coincides with the ordering of the corresponding rotations of S in $M(S)$. As a consequence, if $bwt(S)$ (the last column of $M(S)$) has the form $bwt(S) = x_1y_1x_2y_2 \cdots x_ny_n$, we have that $BWT_*(s) = x_1x_2 \cdots x_n$, and the first column of $M_*(s)$ is $y_1y_2 \cdots y_n$. The LF-map applied to $M(S)$ establishes an order preserving bijection between rows ending with $\alpha \in \Sigma^2$ and rows starting with α. If $\alpha = x_1x_2$, this translates in $M_*(s)$ to an order preserving bijection between rows starting with x_2 and ending with x_1 and rows starting with x_1x_2: this establishes an alternative proof of Lemma 4.1.

The above alternative view of local orderings has probably no practical interest: there is no need to work with the alphabet Σ^2 to emulate something we can easily do working over Σ. However, from the theoretical point of view, it is intriguing, and deserving further investigation, that a family of BWT variants can be obtained by first transforming the string and the alphabet and then applying the standard BWT followed by the string back-transformation.

Example 4.10. Let us consider the string $s = aabaaabac$. The local ordering BWT matrix for s computed using the ordering $\pi_\epsilon = (b, c, a)$, $\pi_a = (b, a, c)$, $\pi_b = \pi_c = (a, b, c)$ is reported in Fig. 6. Instead, Fig. 8 shows the BWT matrix $M(S)$ of the string $S = aa\,ab\,ba\,aa\,aa\,ab\,ba\,ac\,ca$ by using the ordering Π over the alphabet $\{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$. The last column of $M(S)$ is $bwt(S) = ab\,ab\,ac\,aa\,aa\,ca\,aa\,ba\,ba$. One can verify that there is correspondence between the rows of the local ordering BWT matrix described in Fig. 6 and the rows of $M(S)$. Moreover, the string obtained by concatenating the first symbol (colored in red) of each element in $bwt(S)$ is $BWT_*(s) = aaaaacabb$. Finally the string obtained by concatenating the last symbol of each element in $bwt(S)$ gives the first column of the local ordering BWT matrix.

5. Run minimization problem

We consider the following problem: given a string s and a class of BWT variants, find the variant that minimizes the number of runs in the transformed string. As we mentioned in the introduction, this problem is relevant for the compression of highly repetitive collections. Depending on the class of BWT variants, finding the exact minimum could be a difficult problem, so one may want to resort to heuristics. In this context, any lower bound to the minimum number of runs achievable by a class of transformations would be useful to assess the quality of the solutions found.

In this section we describe an efficient algorithm to determine a lower bound on the number of runs in the transformed string which is valid for all BWT variants discussed in this paper. The lower bound is established by computing the minimal number of runs for the most general class we considered: the class of context adaptive BWTs described in Section 3. In this class we can select an alphabet ordering π_x independently for every substring x. It is easy to see that the only orderings that influence the output of the transform are those associated to strings corresponding to the internal nodes of the suffix tree of s. Nevertheless, the number of possible choices is $\approx (\sigma!)^{\mathcal{O}(|s|)}$ which is exponential even for constant alphabets.

Given a suffix tree node v, we denote by $bw(v)$ the multiset of symbols associated to the leaves in the subtree rooted at v (see Fig. 3). We say that a string z_v is a *feasible* arrangement of $bw(v)$ if we can reorder the nodes in the subtree rooted at v so that z_v is obtained by reading left to right the symbols in the reordered subtree. For example, in the suffix tree of Fig. 3 (left), if v is the internal node with upward path aa, it is $bw(v) = \{a, b, c\}$ and bac, bca, acb, cab are feasible arrangements of $bw(v)$, while *abc* and *cba* are *not* feasible arrangements. If τ is the suffix tree root, using the above notation our problem becomes that of finding the feasible arrangement of $bw(\tau)$ with the minimal number of runs. The following theorem shows that, for constant alphabets, the optimal arrangement can be found in linear time using dynamic programming.

Theorem 5.1. *Given a string s over an alphabet of size $\sigma = \mathcal{O}(1)$, the context adaptive transformation minimizing the number of runs in $BWT_*(s)$ can be found in $\mathcal{O}(|s|)$ time.*

Proof. Let Opt denote the minimal number of runs. We show how to compute Opt with a dynamic programming algorithm; the computation of the alphabet orderings giving Opt is done using standard techniques. For each suffix tree node v and pairs of symbols c_i, c_j let $\rho(v, c_i, c_j)$ denote the minimal number of runs among all feasible arrangements of $bw(v)$ starting with c_i and ending with c_j. Clearly, if τ is the suffix tree root, then $Opt = \min_{i,j} \rho(\tau, c_i, c_j)$.

For each leaf ℓ, it is $\rho(\ell, c_i, c_j) = 1$ if $c_i = c_j = bw(\ell)$ and $\rho(\ell, c_i, c_j) = \infty$ otherwise. We need to show how to compute, for each internal node v, the σ^2 values $\rho(v, c_i, c_j)$ for c_i, c_j in Σ, given the, up to σ^3 values, $\rho(w_k, c_\ell, c_m)$, $k = 1, \dots, h$, where $w_1, \dots, w_h$ are the children of v. To this end, we show that for each ordering π of $w_1, \dots, w_h$ we can compute in constant time the minimal number of runs among all the feasible arrangements of $bw(v)$ starting with c_i and ending with c_j and with the additional constraint that v's children are ordered according to π.

To simplify the notation, assume $w_1, \dots, w_h$ have been already reordered according to π. For $k = 1, \dots, h$, let $M_\pi[k, c_\ell, c_m]$ denote the minimal number of runs among all strings x such that $x = y_1 \cdots y_k$, where y_t, for $t = 1, \dots, k$, is a feasible arrangement of $bw(w_t)$, and with the additional constraints that y_1 starts with c_ℓ and y_k ends with c_m (by construction every such x is a feasible arrangement of $bw(v)$)). We have

$$M_\pi[1, c_\ell, c_m] = \rho(w_1, c_\ell, c_m)$$

and for $k = 2, \dots, h$

$$M_\pi[k, c_\ell, c_m] = \min_{i,j} \left(M_\pi[k-1, c_\ell, c_i] + \rho(w_k, c_j, c_m) - \delta_{ij} \right) \tag{2}$$

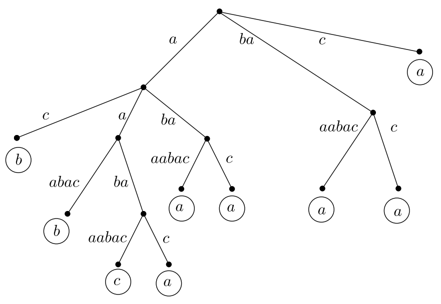
where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. Essentially, (2) states that to find the minimal number of runs for $w_1, \dots, w_k$ we consider all possible ways to combine an optimal solution for $w_1, \dots, w_{k-1}$ followed by a feasible arrangement of $bw(w_k)$. The δ_{ij} term comes from the fact that the number of runs in the concatenation of two strings is equal to the sum of the runs in each string, minus one if the last symbol of the first string is equal to the first symbol of the second string. Once we have the values $M_\pi[h, c_i, c_j]$, the desired values $\rho(v, c_i, c_j)$ are obtained taking the minimum over all possible alphabet ordering π. $\square$

Example 5.2. The run minimization algorithm described in Theorem 5.1 is applied to the standard suffix tree, depicted in Fig. 3 (left), for the string $s = aabaaabac$, where the symbol c used as a string terminator. In Fig. 9 (left) the new tree is shown. It is obtained from the standard suffix tree of s by reordering the children of the internal node with upward path to the root labeled by a. Here, we denote this node by v. It is easy to see that $bw(v) = \{a, a, a, b, b, c\}$ and that *bbcaaa* is a feasible arrangement of $bw(v)$, obtained by moving the third child (with upward path *ac*) to the left. Leaving all other edges unchanged, it is easy to verify that the number of runs remains minimized. The ordering π so obtained is defined as $\pi_a = (c, a, b)$ and $\pi_x = (a, b, c)$ for every other substring x. The generalized BWT matrix of the correspondent context adaptive BWT which minimizes the number of runs is described in Fig. 9 (right).

6. Conclusions and future directions of research

In this paper we introduced a new class of string transformations and showed that they have the same remarkable properties of the BWT: they can be computed and inverted in linear time, they support linear time pattern search directly in the compressed text, and they can transform a zeroth order compressor into a k-th order compressor ("compression boosting" property). This implies that such transformations can replace the BWT in the design of self-indices without any asymptotic loss of performance. Given the crucial role played by the BWT even outside the area of string algorithms, we believe that expanding the number of efficient BWT variants can lead to theoretical and practical advancements. A natural consequence will be the design of "personalized" transformations, where one will choose the "best" alternative to the BWT according to costs and benefits dictated by application domains. As an example, motivated by the problem of compressing highly repetitive string collections that arises in areas such as bioinformatics, we considered the problem of determining the BWT variant that minimizes the number of runs in the transformed string.

Our efficient BWT variants are a special case of a more general class of transformations that have the same properties of the BWT but for which we could not devise efficient (linear time) inversion and search algorithms. We believe this larger class of transformation should be further investigated: we have shown that some of them do have more efficient inversion and search algorithms and this suggests that there could be other subclasses of practical interest. Another possible avenue of further research would be the generalizations of our variants to the recently introduced extension of the BWT in the areas of graphs, languages and automata [9,10,62].



$$
\begin{array}{cccccccccccc}
 & F & & & & & & & & L & \\
 & \downarrow & & & & & & & & \downarrow & \\
1 & a & c & a & a & b & a & a & a & b & \\
2 & a & a & a & b & a & c & a & a & b & \\
3 & a & a & b & a & a & a & b & a & c & \leftarrow s \\
4 & a & a & b & a & c & a & a & b & a & \\
5 & a & b & a & a & a & b & a & c & a & \\
6 & a & b & a & c & a & a & b & a & a & \\
7 & b & a & a & a & b & a & c & a & a & \\
8 & b & a & c & a & a & b & a & a & a & \\
9 & c & a & a & b & a & a & a & b & a &
\end{array}
$$

Fig. 9. Suffix tree for $s = aabaaabac$, with edges reordered using the run minimization algorithm described in Theorem 5.1 (left). The generalized BWT matrix for the string s computed using the orderings $\pi_a = (c, a, b)$, $\pi_x = (a, b, c)$ for every other substring x. The horizontal arrow marks the position of s (right). Each leaf of the tree is associated with the symbol preceding the suffix in s spelled by that leaf. Note that reading left to right the symbols associated to each leaf gives $BWT_*(s)$.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Funding

GM and GR are partially supported by PNRR ECS00000017 Tuscany Health Ecosystem, Spoke 6 “Precision medicine & personalized healthcare”, CUP I53C22000780001, funded by the European Commission under the NextGeneration EU programme. GM is partially supported by the ICSC – Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing, Spoke 1 “FutureHPC & BigData”, Flagship 3, funded by European Union, NextGeneration EU programme. MS is partially supported by the PNRR project Italian Strengthening of ESFRI RI RESILIENCE (ITSERR) funded by the European Union, NextGeneration EU programme, CUP B53C22001770006 and by “FFR-Sciortino”. GR and MS are partially supported by INdAM-GNCS Project 2022 “Metodi combinatori per indicizzare e confrontare collezioni di testi altamente ripetitivi”, CUP E55F22000270001. GM, GR and MS are partially supported by INdAM-GNCS Project 2023 “Linguaggi regolari ordinati, indicizzazione e confronto di collezioni di stringhe, con applicazioni”, CUP E53C22001930001. RG is partially supported by INDAM-GNCS Project 2023 “Approcci computazionali per il supporto alle decisioni nella Medicina di Precisione”, CUP E53C22001930001. RG and GM are partially supported by MIUR-PRIN project “Multicriteria Data Structures and Algorithms: from compressed to learned indexes, and beyond” grant n. 2017WR7SHH.

References

[1] R. Giancarlo, G. Manzini, G. Rosone, M. Sciortino, A new class of searchable and provably highly compressible string transformations, in: CPM, in: LIPIcs, vol. 128, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2019, pp. 12:1–12:12.
[2] M. Burrows, D.J. Wheeler, A block sorting data compression algorithm, Tech. Rep., DIGITAL System Research Center, 1994.
[3] A. Apostolico, The myriad virtues of subword trees, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, Springer Berlin Heidelberg, Berlin, Heidelberg, 1985, pp. 85–96.
[4] P. Fenwick, The Burrows-Wheeler transform for block sorting text compression: principles and improvements, Comput. J. 39 (9) (1996) 731–740.
[5] P. Ferragina, R. Giancarlo, G. Manzini, M. Sciortino, Boosting textual compression in optimal linear time, J. ACM 52 (4) (2005) 688–713.
[6] R. Giancarlo, A. Restivo, M. Sciortino, From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization, Theor. Comput. Sci. 387 (2007) 236–248.
[7] G. Manzini, An analysis of the Burrows-Wheeler transform, J. ACM 48 (3) (2001) 407–430.
[8] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: FOCS 2000, IEEE Computer Society, Washington, DC, USA, 2000, pp. 390–398.
[9] J. Alanko, G. D’Agostino, A. Policriti, N. Prezza, Wheeler languages, Inf. Comput. 281 (2021) 104820.
[10] T. Gagie, G. Manzini, J. Sirén, Wheeler graphs: a framework for BWT-based data structures, Theor. Comput. Sci. 698 (2017) 67–78.
[11] G. Navarro, Compact Data Structures – A Practical Approach, Cambridge University Press, Cambridge, U.K., 2016.
[12] I.M. Gessel, A. Restivo, C. Reutenauer, A bijection between words and multisets of necklaces, Eur. J. Comb. 33 (7) (2012) 1537–1546.
[13] R. Giancarlo, G. Manzini, A. Restivo, G. Rosone, M. Sciortino, Block sorting-based transformations on words: beyond the magic BWT, in: DLT, in: LNCS, vol. 11088, Springer, Cham, 2018, pp. 1–17.
[14] R. Giancarlo, G. Manzini, A. Restivo, G. Rosone, M. Sciortino, The alternating BWT: an algorithmic perspective, Theor. Comput. Sci. 812 (2020) 230–243.
[15] T. Gagie, G. Navarro, N. Prezza, Fully-functional suffix trees and optimal text searching in BWT-runs bounded space, J. ACM 67 (1) (2020) 2.

[16] H. Kaplan, E. Verbin, Most Burrows–Wheeler based compressors are not optimal, in: CPM, in: LNCS, vol. 4580, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 107–118.
[17] V. Mäkinen, G. Navarro, J. Sirén, N. Välimäki, Storage and retrieval of highly repetitive sequence collections, J. Comput. Biol. 17 (3) (2010) 281–308.
[18] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, Burrows-Wheeler transform and run-length enconding, in: WORDS, in: LNCS, vol. 10432, Springer, 2017, pp. 228–239.
[19] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, L. Versari, Measuring the clustering effect of BWT via RLE, Theor. Comput. Sci. 698 (2017) 79–87.
[20] G. Navarro, Indexing highly repetitive string collections, part I: repetitiveness measures, ACM Comput. Surv. 54 (2) (2021) 29:1–29:31.
[21] G. Navarro, Indexing highly repetitive string collections, part II: compressed indexes, ACM Comput. Surv. 54 (2) (2021) 26:1–26:32.
[22] A. Restivo, G. Rosone, Balancing and clustering of words in the Burrows-Wheeler transform, Theor. Comput. Sci. 412 (27) (2011) 3019–3032.
[23] A. Frosini, I. Mancini, S. Rinaldi, G. Romana, M. Sciortino, Logarithmic equal-letter runs for BWT of purely morphic words, in: DLT, in: LNCS, vol. 13257, Springer, 2022, pp. 139–151.
[24] V. Guerrini, F.A. Louza, G. Rosone, Lossy compressor preserving variant calling through extended BWT, in: BIOSTEC/BIOINFORMATICS, INSTICC, SciTePress, 2022, pp. 38–48.
[25] G. Fici, G. Romana, M. Sciortino, C. Urbina, On the impact of morphisms on BWT-runs, in: CPM, in: LIPIcs, vol. 259, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2023, pp. 10:1–10:18.
[26] S. Giuliani, S. Inenaga, Z. Lipták, G. Romana, M. Sciortino, C. Urbina, Bit catastrophes for the Burrows-Wheeler transform, in: DLT, in: LNCS, vol. 13911, Springer, 2023, pp. 86–99.
[27] C. Boucher, D. Cenzato, Z. Lipták, M. Rossi, M. Sciortino, Computing the original eBWT faster, simpler, and with less memory, in: SPIRE, in: LNCS, vol. 12944, Springer International Publishing, Cham, 2021, pp. 129–142.
[28] R. Kosaraju, G. Manzini, Compression of low entropy strings with Lempel–Ziv algorithms, SIAM J. Comput. 29 (3) (1999) 893–911.
[29] P. Ferragina, G. Manzini, Indexing compressed text, J. ACM 52 (2005) 552–581.
[30] D. Belazzougui, G. Navarro, Optimal lower and upper bounds for representing sequences, ACM Trans. Algorithms 11 (4) (2015) 31:1–31:21.
[31] V. Mäkinen, D. Belazzougui, F. Cunial, A. Tomescu, Genome-Scale Algorithm Design, Cambridge University Press, Cambridge, U.K., ISBN 978-1-107-07853-6, 2015.
[32] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007).
[33] M. Schindler, A fast block-sorting algorithm for lossless data compression, in: DCC, IEEE Computer Society, Washington, DC, USA, 1997, p. 469.
[34] J.S. Culpepper, M. Petri, S.J. Puglisi, Revisiting bounded context block-sorting transformations, Softw. Pract. Exp. 42 (8) (2012) 1037–1054.
[35] M. Petri, G. Navarro, J.S. Culpepper, S.J. Puglisi, Backwards search in context bound text transformations, in: CCP, IEEE Computer Society, Washington, DC, USA, 2011, pp. 82–91.
[36] B. Chapin, S. Tate, Higher compression from the Burrows-Wheeler transform by modified sorting, in: DCC, IEEE Computer Society, Washington, DC, USA, 1998, p. 532.
[37] J.Y. Gil, D.A. Scott, A bijective string sorting transform, CoRR, arXiv:1201.3077 [abs], 2012.
[38] K.T. Chen, R.H. Fox, R.C. Lyndon, Free differential calculus. IV. The quotient groups of the lower central series, Ann. Math. (2) 68 (1958) 81–95.
[39] H. Bannai, J. Kärkkäinen, D. Köppl, M. Piatkowski, Constructing the bijective and the extended Burrows-Wheeler transform in linear time, in: CPM, in: LIPIcs, vol. 191, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021, pp. 7:1–7:16.
[40] H. Bannai, J. Kärkkäinen, D. Köppl, M. Piatkowski, Indexing the bijective BWT, in: N. Pisanti, S.P. Pissis (Eds.), CPM, in: LIPIcs, vol. 128, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 17:1–17:14.
[41] S. Mantaci, A. Restivo, G. Rosone, M. Sciortino, An extension of the Burrows-Wheeler transform, Theor. Comput. Sci. 387 (3) (2007) 298–312.
[42] L. Egidi, G. Manzini, Lightweight merging of compressed indices based on BWT variants, Theor. Comput. Sci. 812 (2020) 214–229.
[43] P. Ferragina, R. Venturini, The compressed permuterm index, ACM Trans. Algorithms 7 (1) (2010) 10:1–10:21.
[44] W. Hon, C. Lu, R. Shah, S.V. Thankachan, Succinct indexes for circular patterns, in: ISAAC, in: LNCS, vol. 7074, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 673–682.
[45] W. Hon, T. Ku, C. Lu, R. Shah, S.V. Thankachan, Efficient algorithm for circular Burrows-Wheeler transform, in: CPM, in: LNCS, vol. 7354, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 257–268.
[46] C. Boucher, D. Cenzato, Z. Lipták, M. Rossi, M. Sciortino, r-indexing the eBWT, in: SPIRE, in: LNCS, vol. 12944, Springer International Publishing, Cham, 2021, pp. 3–12.
[47] J. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, É. Prieur-Gaston, A survey of string orderings and their application to the Burrows-Wheeler transform, Theor. Comput. Sci. (2017).
[48] J.W. Daykin, N. Mhaskar, W.F. Smyth, Computation of the suffix array, Burrows-Wheeler transform and FM-index in V-order, Theor. Comput. Sci. 880 (2021) 82–96.
[49] J. Daykin, C. Iliopoulos, W. Smyth, Parallel RAM algorithms for factorizing words, Theor. Comput. Sci. 127 (1) (1994) 53–67.
[50] M. Crochemore, J. Désarménien, D. Perrin, A note on the Burrows-Wheeler transformation, Theor. Comput. Sci. 332 (2005) 567–572.
[51] I.M. Gessel, C. Reutenauer, Counting permutations with given cycle structure and descent set, J. Comb. Theory, Ser. A 64 (2) (1993) 189–215.
[52] A.J. Cox, M.J. Bauer, T. Jakobi, G. Rosone, Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform, Bioinformatics 28 (11) (2012) 1415–1419, availability: Code is part of the BEETL library, available as a github repository at https://github.com/BEETL/BEETL.
[53] H. Li, Fast construction of FM-index for long sequence reads, Bioinformatics 30 (22) (2014) 3274–3275, source code: https://github.com/lh3/ropebwt2.
[54] M.J. Bauer, A.J. Cox, G. Rosone, Lightweight BWT construction for very large string collections, in: CPM, in: LNCS, vol. 6661, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 219–231.
[55] M.J. Bauer, A.J. Cox, G. Rosone, Lightweight algorithms for constructing and inverting the BWT of string collections, Theor. Comput. Sci. 483 (0) (2013) 134–148, availability: Code is part of the BEETL library, available as a github repository at https://github.com/BEETL/BEETL.
[56] J.W. Bentley, D. Gibney, S.V. Thankachan, On the complexity of BWT-runs minimization via alphabet reordering, CoRR, arXiv:1911.03035 [abs], 2019.
[57] J.W. Bentley, D. Gibney, S.V. Thankachan, On the complexity of BWT-runs minimization via alphabet reordering, in: ESA, in: LIPIcs, vol. 173, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2020, pp. 15:1–15:13.
[58] D. Cenzato, V. Guerrini, Z. Lipták, G. Rosone, Computing the optimal BWT of very large string collections, in: Data Compression Conference, DCC 2023, IEEE, Snowbird, UT, 2023, pp. 71–80.
[59] B. Cazaux, E. Rivals, Linking BWT and XBW via aho-corasick automaton: applications to run-length encoding, in: CPM, in: LIPIcs, vol. 128, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019, pp. 24:1–24:20.
[60] D. Cenzato, Z. Lipták, A theoretical and experimental analysis of BWT variants for string collections, in: CPM, in: LIPIcs, vol. 223, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 25:1–25:18.
[61] H. Bannai, T. Gagie, T. I, Refining the r-index, Theor. Comput. Sci. 812 (2020) 96–108.
[62] N. Cotumaccio, N. Prezza, On indexing and compressing finite automata, in: SODA, SIAM, USA, 2021, pp. 2585–2599.