A New Class of Searchable and Provably Highly Compressible String Transformations

Raffaele Giancarlo
University of Palermo, Dipartimento di Matematica e Informatica, Italy
raffaele.giancarlo@unipa.it

Giovanni Manzini
University of Eastern Piedmont, Alessandria, and IIT-CNR, Pisa, Italy
giovanni.manzini@uniupo.it

Giovanna Rosone
University of Pisa, Dipartimento di Informatica, Italy
giovanna.rosone@unipi.it

Marinella Sciortino
University of Palermo, Dipartimento di Matematica e Informatica, Italy
marinella.sciortino@unipa.it

Abstract

The Burrows-Wheeler Transform is a string transformation that plays a fundamental role for the design of self-indexing compressed data structures. Over the years, researchers have successfully extended this transformation outside the domains of strings. However, efforts to find non-trivial alternatives of the original, now 25 years old, Burrows-Wheeler string transformation have met limited success. In this paper we bring new lymph to this area by introducing a whole new family of transformations that have all the "myriad virtues" of the BWT: they can be computed and inverted in linear time, they produce provably highly compressible strings, and they support linear time pattern search directly on the transformed string. This new family is a special case of a more general class of transformations based on *context adaptive alphabet orderings*, a concept introduced here. This more general class includes also the Alternating BWT, another invertible string transforms recently introduced in connection with a generalization of Lyndon words.

2012 ACM Subject Classification Theory of computation → Data compression; Mathematics of computing → Combinatorial algorithms

Keywords and phrases Data Indexing and Compression; Burrows-Wheeler Transformation; Combinatorics on Words

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.9

Funding GR and SM are partially supported by MIUR-SIR project CMACBioSeq "Combinatorial methods for analysis and compression of biological sequences" grant n. RBSI146R5L; RG and GM are partially supported by INdAM-GNCS project 2018 "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.

1 Introduction

The Burrows Wheeler Transform [2] (BWT) is a string transformation that had a revolutionary impact in the design of succinct or compressed data structures. Originally proposed as a tool for text compression, shortly after its introduction [9] it has been shown 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 search other combinatorial objects such as: trees, graphs, finite automata, and even string alignments.

30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019).
Editors: Nadia Pisanti and Solon P. Pissis; Article No. 9; pp. 9:1–9:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

See [11] for an attempt to unify some of these results and [25] for an in-depth treatment of the field of compact data structures.

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 [13] in the field of combinatorics of words and its basic algorithmic properties have been described in [15].

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). 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.

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 highly repetitive collections there is theoretical and practical evidence that the entropy underestimates the compressibility of the collection and much better compression ratios are obtained exploiting runs of equal symbols in the BWT [4, 12, 18, 19, 21, 22, 23]. In Section 5 we show 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.

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 $1 \leq i \leq j \leq n$. 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 the string s and it is denoted by $BWT(s)$ (see Figure 1 (left)). The BWT can be computed in $\mathcal{O}(|s|)$ time using any algorithm for Suffix Array construction [16, 17]. 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 position I of s in the rotations matrix (it is $I = 2$ in Figure 1 (left)).

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 [8, 24]. 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.

It is proven in [8, 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 [10] 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 7 in [1])

▶ **Theorem 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 properties of the BWT of being *compressible* and *searchable* combine nicely to give us *indexing capabilities* in *compressed space*. Indeed, combining a zero 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; see [10, 20, 25, 26] 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 [28] 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. Recent studies [6, 27] have 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 [3] 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. 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 [7]; it turns out that these variants satisfy property **1** in part but nothing is known with respect to properties **2** and **3**.

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 [13] starting from a result in combinatorics of words [5] characterizing the BWT as the inverse of a known bijection between words and multisets of primitive necklaces [14]. 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. Figure 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 [15]. 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. Instead, in the ABWT 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 be still computed efficiently using `rank` operations, the ABWT can replace the BWT for the construction of self-indices.

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

$$\begin{array}{cccccccccc}
 & F & & & & & & & & L \\
 & \downarrow & & & & & & & & \downarrow \\
 & a & a & a & b & a & c & a & a & b \\
s \rightarrow & a & a & b & a & a & a & b & a & c \\
 & a & a & b & a & c & a & a & b & a \\
 & a & b & a & a & a & b & a & c & a \\
 & a & b & a & c & a & a & b & a & a \\
 & a & c & a & a & b & a & a & a & b \\
 & b & a & a & a & b & a & c & a & a \\
 & b & a & c & a & a & b & a & a & a \\
 & c & a & a & b & a & a & a & b & a
\end{array}
\qquad
\begin{array}{cccccccccc}
 & F & & & & & & & & L \\
 & \downarrow & & & & & & & & \downarrow \\
 & a & c & a & a & b & a & a & a & b \\
 & a & b & a & c & a & a & b & a & a \\
 & a & b & a & a & a & b & a & c & a \\
 & a & a & a & b & a & c & a & a & b \\
s \rightarrow & a & a & b & a & a & a & b & a & c \\
 & a & a & b & a & c & a & a & b & a \\
 & b & a & a & a & b & a & c & a & a \\
 & b & a & c & a & a & b & a & a & a \\
 & c & a & a & b & a & a & a & b & a
\end{array}$$

Figure 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.

$$\begin{array}{cccccccccc}
 & F & & & & & & & & L \\
 & \downarrow & & & & & & & & \downarrow \\
 & b & a & a & a & b & a & c & a & a \\
 & b & a & c & a & a & b & a & a & a \\
 & a & c & a & a & b & a & a & a & b \\
s \rightarrow & a & a & b & a & a & a & b & a & c \\
 & a & a & b & a & c & a & a & b & a \\
 & a & a & a & b & a & c & a & a & b \\
 & a & b & a & a & a & b & a & c & a \\
 & a & b & a & c & a & a & b & a & a \\
 & c & a & a & b & a & a & a & b & a
\end{array}$$

Figure 2 The generalized BWT matrix for the string $s = aabaaabac$ computed using the orderings $\pi_\epsilon = (b, a, c)$, $\pi_a = (c, a, b)$, $\pi_{aa} = (c, b, a)$, 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.

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 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*.

An example is shown in Figure 2: 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 starting with c, then the ones starting with a, followed by the ones starting with b. The complete ordering of the rows is established in a similar way on the basis of the orderings π_x.

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

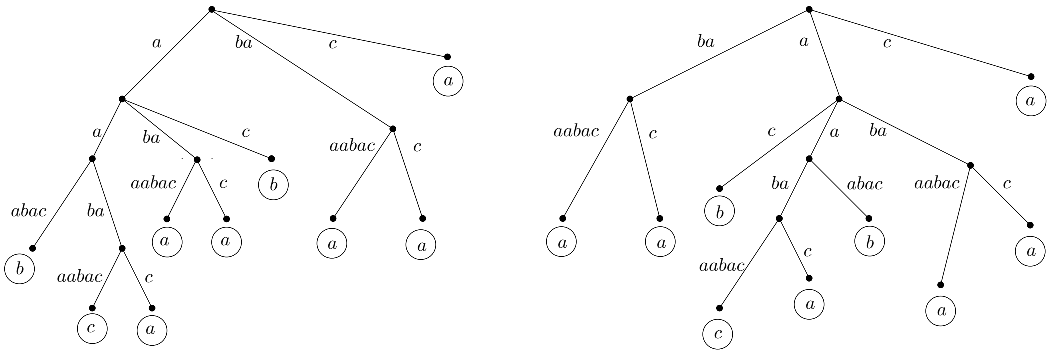

Figure 3 Standard suffix tree for $s = aabaaabac$ with the symbol c used as a string terminator (left), and suffix tree with edges reordered using the same orderings of Figure 2 (right). To each leaf it is associated the symbol preceding in s the suffix spelled by that leaf. Note that reading left to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

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. This class of transformations has been mentioned in [8, Sect. 5.2] under the name of *string permutations realized by a Suffix Tree* (the definition in [8] is slightly more general; for example it includes the bounded context BWT, which is not included in our class). Indeed, if the input string s has a unique end-of-string terminator, one can easily see that these transformations can be obtained assigning an ordering to the children of each node of the suffix tree of s

Although in [8] 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 with a proper suffix tree visit (see Figure 3), 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.

To see that the generalized BWTs can be computed in $\mathcal{O}(n \log \sigma)$ time consider first the simpler case in which the string s has a unique end-of-string terminator. To build $L = BWT_*(s)$ we first build the suffix tree for s. Then, we visit the suffix tree 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 visit, each time we reach a leaf we write the symbol associated to it: the resulting string is exactly $L = BWT_*(s)$. The above argument also shows that the number of permutations required to define a generalized BWT on a fixed string s is at most $|s|$, i.e. the number of internal suffix tree nodes. If s doesn't have a unique terminator, the argument is analogous except that we replace the suffix tree with the compressed trie containing all the cyclic rotations of s. To see that generalized BWTs have the boosting property we observe that the proof for the BWT (Theorem 5.4 in [8]) is based on structural properties of the suffix tree, and can be repeated verbatim for the generalized BWTs.

Summing up, context adaptive transformations generalize the BWT in two important aspects: efficient (linear time in n) computation and compressibility. In [8] 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 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, 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. Indeed in the full paper we will show that the complete class of transformations studied in [15] is a subclass of context adaptive transformations.

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 1. In this section we show that given L and the set of alphabet permutations used to build $M_*(s)$ then, for each string x, we can determine in $\mathcal{O}(\sigma|x|^2)$ time the set of $M_*(s)$ rows prefixed by x. We preliminary 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 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.** *Given a string x, we denote by $R^*[x]$ the set of $\sigma+1$ integers $[b_x, \ell_1, \ell_2, \dots, \ell_\sigma]$ such that b_x is the lower extreme of $R[x]$ and, for $i = 1, \dots, \sigma$, 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 $R^*[x] = [b_x, \ell_1, \ell_2, \dots, \ell_\sigma]$, 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 $L[i, j]$.

▶ **Lemma 4.** *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

$$b = b_x + \sum_j \ell_j, \qquad \ell = \ell_i \tag{1}$$

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 5.** *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 4, 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 4, and then counting the number of rows prefixed by $x_1 x_2 \cdots x_m c$, 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 1. ◀

▶ **Theorem 6.** *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 5 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 need to store from one iteration to the next is $\mathcal{O}(p)$ $R^*[\cdot]$ values, which take $\mathcal{O}(\sigma p)$ words. By Lemma 5, the computation of each value takes $\mathcal{O}(\sigma)$ time so the overall complexity is $\mathcal{O}(\sigma p^2)$ 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 7.** *Given $R^*[x] = [b_x, \ell_1, \ell_2, \ldots, \ell_\sigma]$ and a row index i with $b_x \leq i < b_x + \sum_{j=1}^{\sigma} \ell_j$, the $(|x|+1)$-st character of row i can be computed in $\mathcal{O}(\sigma)$ time.*

Proof. Let $\rho_1, \ldots, \rho_\sigma$ denote the alphabet symbol reordered according to the permutation π_x, and let $\ell'_1, \ldots, \ell'_\sigma$ denote the values $\ell_1, \ldots, \ell_\sigma$ 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

$$b_x + \sum_{1 \leq h < j} \ell'_h \;\leq\; i \;<\; b_x + \sum_{1 \leq h \leq j} \ell'_h$$

◀

▶ **Theorem 8.** *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 s's first character 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 $m = 1, \ldots, n-1$, 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 Lemma 7, 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 $j = 1, \ldots, m$ we can compute $R^*[s_j \cdots s_{m+1}]$ adding one diagonal to the scheme shown in the proof of Theorem 6. By Lemma 5, the overall cost is $\mathcal{O}(\sigma|s|^2)$ as claimed. ◀

4 BWTs based on local orderings

In our definition of context adaptive transformation, the alphabet ordering π_x associated to x can depend on the whole string x; in this sense the context has full memory. In this section we consider transformations in which the context has a bounded memory, in that it 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 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 $M_*(s)$. The following lemma establishes an important property of local ordering transformations.

▶ **Lemma 9.** *If $M_*(s)$ is based on a local ordering, then for any pair of characters x_1, x_2 there is an order preserving bijection between the set of rows starting with $x_1 x_2$ 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 $x_1 x_2$ 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 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 $s[i+h]$ is smaller than $s[j+h]$ according to the alphabet ordering associated to symbol $s[i+h-1] = s[j+h-1]$. The two rotations $s[i-1 \cdots]$ and $s[j-1 \cdots]$ both start with $x_1 x_2$ and their relative position also depends on the relative ranks of $s[i+h]$ and $s[j+h]$ according to the alphabet ordering associated to symbol $s[i+h-1] = s[j+h-1]$. 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 $s[j \cdots]$. ◀

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

▶ **Lemma 10.** *Suppose $BWT_*(s)$ supports constant time rank queries. Let $x = x_1 x_2 \cdots x_m$ be any length-m string with $m > 1$. Then, given $R[x_1 x_2]$, $R[x_2]$ 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 9 there is an order preserving bijection between the rows in $R[x_1 x_2]$ 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. Hence, if, among the rows starting with x_2 and ending with x_1, those prefixed by $x_2 \cdots x_m$ are in positions $r, r+1, \ldots, r+h$, then, among the rows starting with $x_1 x_2$, those prefixed by $x_1 x_2 \cdots x_m$ are in positions $r, r+1, \ldots, r+h$. ◀

▶ **Theorem 11.** *Suppose $BWT_*(s)$ is based on a local ordering 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 $\mathcal{O}(p)$ space.*

Proof. We reason as in the proof of Theorem 6, except that because of Lemma 10 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 9 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 [10].

▶ **Lemma 12.** *Suppose $BWT_*(s)$ is based on a local ordering 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. Compute the first and last symbol of row i and then apply Lemma 9. ◀

▶ **Corollary 13.** *If $BWT_*(s)$ is based on a local ordering 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.* ◀

In the full paper we will show that bounded context adaptive BWTs can be generalized to the case in which the ordering π_x depends only on the last $k > 1$ symbols of x. Search and inversion can still be performed in linear time with the only difference that preprocessing now takes $\mathcal{O}(\sigma^{k+1})$ time and space.

5 Run minimization problem

In this section 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.

We consider the general 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. However, 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. 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. 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 reading left to right the symbols in the reordered subtree. For example, in the suffix tree of Figure 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. For constant alphabets the following theorem, proven in the Appendix, shows that the optimal arrangement can be found in linear time using dynamic programming.

▶ **Theorem 14.** *Given a string s over a constant size alphabet, the context adaptive transformation BWT_* 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, $\rho(w_k, c_\ell, c_m)$, $k = 1, \ldots, h$, where $w_1, \ldots, w_h$ are the children of v. To this end, we show that for each ordering π of $w_1, \ldots, w_h$ 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 $w_1, \ldots, w_h$ have been already reordered according to π. For $k = 1, \ldots, h$ 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 $t = 1, \ldots, k$, 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. We have

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

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

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

where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. Essentially, (2) states that to find the minimal number of runs for $w_1, \ldots, w_k$ we consider all possible ways to combine an optimal solution for $w_1, \ldots, w_{k-1}$ 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. ◀

Note that, both the assumptions on the alphabet size and the constant-time rank operations could be relaxed without affecting the correctness of the results provided in this paper, accordingly the running time increases. For instance, in Theorem 14, the algorithm runs in $\mathcal{O}(|s|\sigma^2)$ time, for any alphabet.

Clearly the above theorem does not immediately yield a practical compressor, since the cost of specifying the alphabet ordering at each node is likely to outweigh the advantage of minimizing the number of runs. However we notice that: 1) the optimal transformation for a string will reasonably produce good results on similar strings so we can compute and store the ordering once and use it many times, 2) since Theorem 14 holds for the most general class, it provides a lower bound for the more interesting and practical BWTs based on local orderings and the ABWT.

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

3 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`.

4 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.

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

6 J. S. Culpepper, M. Petri, and S. J. Puglisi. Revisiting bounded context block-sorting transformations. *Software Pract. Exper.*, 42(8):1037–1054, 2012.

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

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

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

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

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

12 T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in bwt-runs bounded space. In *SODA*, pages 1459–1477. SIAM, 2018.

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

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

15 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 *Lecture Notes in Computer Science*, pages 1–17. Springer, 2018.

16 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.

17 J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

18 D. Kempa and N. Prezza. At the roots of dictionary compression: string attractors. In *STOC*, pages 827–840. ACM, 2018.

19 S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013.

20 V. Mäkinen, D. Belazzougui, F. Cunial, and A. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, 2015. ISBN 978-1-107-07853-6.

21 V. Mäkinen, G. Navarro, J. Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010.

22 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.

23 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.

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

25 G. Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016. ISBN 978-1-107-15238-0.

26 G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

27 M. Petri, G. Navarro, J. S. Culpepper, and S. J. Puglisi. Backwards search in context bound text transformations. In *CCP*, pages 82–91. IEEE Computer Society, 2011.

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